NAME: TUSHAR SAINI

BATCH: 46

DSA LAB FILE

SUBJECT FACULTY: MR. VIRENDAR KADYAN

←GitHUB

Q1) *C program to perform operation on single linked list(L.L) using pointers*

```
1.
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.
5.  // Define the structure for a node. It is a self-referential structure
6.  struct node {
7.      int data;
8.      struct node *next;
9.  };
10.
11. int main() {
12.     struct node *start = NULL;
13.     int option; // Changed from char to int
14.
15.     do {
16.         printf("\n\n\t\t----------MAIN MENU----------"
17.                "\n 1:) Create a Linked-List"
18.                "\n 2:) Display the Linked-List"
19.                "\n 3:) Insert a node in the LL"
20.                "\n 4:) Delete a node in the LL"
21.                "\n 5:) Exit the program"
22.                "\n\n Enter your choice : ");
23.         if (scanf("%d", &option) != 1) {
24.             printf("Invalid input. Please enter a number.\n");
25.             while(getchar() != '\n'); // clear the input buffer
26.             continue;
27.         }
28.         if(option < 1 || option > 5) {
29.             printf("Invalid choice. Please try again.\n");
30.             continue;
31.         }
32.
33.         switch(option) {
34.             case 1: {
35.                 // Create a linked list
36.                 struct node *newNode, *ptr;
37.                 int num;
38.
39.                 printf("\n Type numbers to create a Linked list (enter -1 to end):\n ");
40.                 scanf("%d", &num);
41.
42.                 while (num != -1) {
43.                     newNode = (struct node *)malloc(sizeof(struct node));
44.
```

```
45.                 if (newNode == NULL) {
46.                     printf("Memory allocation failed. \n");
47.                     break;
48.                 }
49.
50.                 newNode->data = num;
51.                 newNode->next = NULL;
52.
53.                 if (start == NULL) {
54.                     start = newNode;
55.                 } else {
56.                     ptr = start;
57.                     while (ptr->next != NULL){
58.                         ptr = ptr->next;
59.                     }
60.                     ptr->next = newNode;
61.                 }
62.
63.                 printf(" Enter data to create the Linked-List (enter -1 to end): ");
64.                 scanf("%d", &num);
65.             }
66.             break;
67.         }
68.         case 2: {
69.             // Display the linked list
70.             struct node *ptr = start;
71.             if (start == NULL) {
72.                 printf("\nList is empty.");
73.             } else {
74.                 printf("\nLinked List: ");
75.                 while (ptr != NULL) {
76.                     printf("%d -> ", ptr->data);
77.                     ptr = ptr->next;
78.                 }
79.                 printf("NULL\n");
80.             }
81.             break;
82.         }
83.         case 3: {
84.             // Insert a node into the linked list
85.             struct node *newNode, *ptr, *prev;
86.             int num, val;
87.             int choice;
88.
89.             printf("\n Choose method of insertion:"
90.                    "\n 1:) Insert at the beginning"
```

```
91.                 "\n 2:) Insert at the end"
92.                 "\n 3:) Insert after a specific value"
93.                 "\n 4:) Insert before a specific value"
94.                 "\n\n Enter your choice: ");
95.             if (scanf("%d", &choice) != 1) {
96.                 printf("Invalid input. Please enter a number.\n");
97.                 while (getchar() != '\n'); // clear the input buffer
98.                 continue;
99.             }
100.                    if(option < 1 || option > 4) {
101.                        printf("Invalid choice. Please try again.\n");
102.                        continue;
103.                    }
104.
105.                    switch(choice) {
106.                        case 1:
107.                            printf("\n Enter the data to be inserted at the beginning: ");
108.                            scanf("%d", &num);
109.
110.                            newNode = (struct node *)malloc(sizeof(struct node));
111.                            if (newNode == NULL) {
112.                                printf("Memory allocation failed. Unable to insert node.\n");
113.                                break;
114.                            }
115.                            newNode->data = num;
116.                            newNode->next = start;
117.                            start = newNode;
118.                            break;
119.
120.                        case 2:
121.                            printf("\n Enter the data to be inserted at the end: ");
122.                            scanf("%d", &num);
123.
124.                            newNode = (struct node *)malloc(sizeof(struct node));
125.                            if (newNode == NULL) {
126.                                printf("Memory allocation failed. Unable to insert node.\n");
127.                                break;
128.                            }
129.                            newNode->data = num;
130.                            newNode->next = NULL;
131.
132.                            if (start == NULL) {
133.                                start = newNode;
134.                            } else {
135.                                ptr = start;
136.                                while (ptr->next != NULL) {
```

```
137.                         ptr = ptr->next;
138.                      }
139.                      ptr->next = newNode;
140.                   }
141.                   break;
142.
143.                case 3:
144.                   printf("\n Enter the data to be inserted  after a specific value: ");
145.                   scanf("%d", &num);
146.                   printf(" Enter the value after which you wanna insert: ");
147.                   scanf("%d", &val);
148.
149.                   ptr = start;
150.                   while (ptr != NULL && ptr->data != val) {
151.                      ptr = ptr->next;
152.                   }
153.
154.                   if (ptr == NULL) {
155.                      printf("Value not found in the list.\n");
156.                   } else {
157.                      newNode = (struct node *)malloc(sizeof(struct node));
158.                      if (newNode == NULL) {
159.                         printf("Memory allocation failed. Unable to insert
    node.\n");
160.                         break;
161.                      }
162.                      newNode->data = num;
163.                      newNode->next = ptr->next;
164.                      ptr->next = newNode;
165.                   }
166.                   break;
167.
168.                case 4:
169.                   printf("\n Enter the data to be inserted before a specific value:
    ");
170.                   scanf("%d", &num);
171.                   printf(" Enter the value before which to insert: ");
172.                   scanf("%d", &val);
173.
174.                   ptr = start;
175.                   prev = NULL;
176.                   while (ptr != NULL && ptr->data != val) {
177.                      prev = ptr;
178.                      ptr = ptr->next;
179.                   }
180.
```

```
181.                     if (ptr == NULL) {
182.                         printf("Value not found in the list.\n");
183.                     } else {
184.                         newNode = (struct node *)malloc(sizeof(struct node));
185.                         if (newNode == NULL) {
186.                             printf("Memory allocation failed. Unable to insert
    node.\n");
187.                             break;
188.                         }
189.                         newNode->data = num;
190.                         newNode->next = ptr;
191.                         if (prev == NULL) {
192.                             start = newNode;
193.                         } else {
194.                             prev->next = newNode;
195.                         }
196.                     }
197.                     break;
198.                 default:
199.                     printf("\nInvalid choice.");
200.             }
201.             break;
202.         }
203.         case 4: {
204.             // Delete a node from the linked list
205.             struct node *ptr = start, *prev = NULL;
206.             int  pos;
207.             int choice;
208.
209.             if (start == NULL) {
210.                 printf("\nList is empty. Nothing to delete.");
211.                 break;
212.             }
213.
214.             printf("\n Choose method of deletion:"
215.                    "\n 1:) Delete at the beginning"
216.                    "\n 2:) Delete at the end"
217.                    "\n 3:) Delete at a specific position"
218.                    "\n\n Enter your choice: ");
219.             if (scanf("%d", &choice) != 1) {
220.                 printf("Invalid input. Please enter a number.\n");
221.                 while (getchar() != '\n'); // clear the input buffer
222.                 continue;
223.             }
224.             if(option < 1 || option > 3) {
225.                 printf("Invalid choice. Please try again.\n");
```

```
226.                 continue;
227.             }
228.
229.             switch(choice) {
230.                 case 1:
231.                     // Delete at the beginning
232.                     start = start->next;
233.                     free(ptr);
234.                     break;
235.
236.                 case 2:
237.                     // Delete at the end
238.                     while (ptr->next != NULL) {
239.                         prev = ptr;
240.                         ptr = ptr->next;
241.                     }
242.                     if (prev != NULL) {
243.                         prev->next = NULL;
244.                         free(ptr);
245.                     } else {
246.                         // If there is only one node
247.                         free(start);
248.                         start = NULL;
249.                     }
250.                     break;
251.
252.                 case 3:
253.                     // Delete at a specific position
254.                     printf("\n Enter the position to be deleted: ");
255.                     scanf("%d", &pos);
256.
257.                     if (pos < 1) {
258.                         printf("\nInvalid position.");
259.                         break;
260.                     }
261.
262.                     int count = 1;
263.                     ptr = start;
264.                     while (ptr != NULL && count < pos) {
265.                         prev = ptr;
266.                         ptr = ptr->next;
267.                         count++;
268.                     }
269.
270.                     if (ptr == NULL) {
271.                         printf("\nPosition not found in the list.");
```

```
                        } else {
                            if (prev != NULL) {
                                prev->next = ptr->next;
                            } else {
                                start = ptr->next;
                            }
                            free(ptr);
                        }
                        break;
                    default:
                        printf("\nInvalid choice.");
                }
                break;
            }
            case 5:
                printf("\nExiting...");
                break;
            default:
                printf("\nTry again!");
        }
    } while(option != 5);

    return 0;
}
```

Q2) ***C program to perform operation on single linked list(L.L) using pointers and functions:***

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

// Define the structure for a node. It is a self-referential structure
struct node {
    int data;
    struct node *next;
};

// Function prototypes
struct node *create_ll();
struct node *display_ll(struct node *);
struct node *insert_Node(struct node *);
struct node *delete_Node(struct node *);

int main() {
    struct node *start = NULL;
```

```
18.    int option;
19.
20.    printf("Namaskaram! This code performs both insertion and deletion on a single
       linked list.\n");
21.
22.    // Main menu loop
23.    do {
24.       printf("\n\n\t\t----------MAIN MENU----------"
25.             "\n 1:) Create a Linked-List"
26.             "\n 2:) Display the Linked-List"
27.             "\n 3:) Insert a node in the LL"
28.             "\n 4:) Delete a node in the LL"
29.             "\n 5:) Exit the program"
30.             "\n\n Enter your choice : ");
31.       if (scanf("%d", &option) != 1) {
32.          printf("Invalid input. Please enter a number.\n");
33.          while (getchar() != '\n'); // clear the input buffer
34.          continue;
35.       }
36.       if(option > 5 || option < 1) {
37.          printf("Invalid choice. Please try again.\n");
38.          continue;
39.       }
40.
41.       switch(option) {
42.          case 1:
43.             start = create_ll();
44.             break;
45.          case 2:
46.             display_ll(start);
47.             break;
48.          case 3:
49.             start = insert_Node(start);
50.             break;
51.          case 4:
52.             start = delete_Node(start);
53.             break;
54.          case 5:
55.             printf("\nExiting...\n");
56.             break;
57.       }
58.    } while(option != 5);
59.
60.    return 0;
61. }
62.
```

```
63. struct node *create_ll() {
64.     struct node *start = NULL, *newNode, *ptr;
65.     int num;
66.
67.     printf("\n Type numbers to create a Linked list (enter -1 to end):\n ");
68.     while (scanf("%d", &num) == 1 && num != -1) {
69.         newNode = (struct node *)malloc(sizeof(struct node));
70.         if (newNode == NULL) {
71.             printf("Memory allocation failed. \n");
72.             return start; // Return the current state of the linked list
73.         }
74.
75.         newNode->data = num;
76.         newNode->next = NULL;
77.
78.         if (start == NULL) {
79.             start = newNode;
80.         } else {
81.             ptr = start;
82.             while (ptr->next != NULL){
83.                 ptr = ptr->next;
84.             }
85.             ptr->next = newNode;
86.         }
87.
88.         printf(" Enter next number to add to the Linked-List (enter -1 to end): ");
89.     }
90.     return start;
91. }
92.
93. struct node *display_ll(struct node *start) {
94.     if (start == NULL) {
95.         printf("\nList is empty.\n");
96.         return;
97.     }
98.
99.     struct node *ptr = start;
100.            printf("\nLinked List: ");
101.            while (ptr != NULL) {
102.                printf("%d -> ", ptr->data);
103.                ptr = ptr->next;
104.            }
105.            printf("NULL\n");
106.        }
107.
108.        struct node *insert_Node(struct node *start) {
```

```
109.        struct node *newNode, *ptr, *prev;
110.        int num, val, choice;
111.
112.        printf("\n Choose method of insertion:"
113.              "\n 1:) Insert at the beginning"
114.              "\n 2:) Insert at the end"
115.              "\n 3:) Insert after a specific value"
116.              "\n 4:) Insert before a specific value"
117.              "\n\n Enter your choice: ");
118.        scanf("%d", &choice);
119.        if(choice < 1 || choice > 4) {
120.           printf("Invalid choice. Please try again.\n");
121.           return start;
122.        }
123.
124.        printf("\n Enter the data to be inserted: ");
125.        scanf("%d", &num);
126.
127.        newNode = (struct node *)malloc(sizeof(struct node));
128.        if (newNode == NULL) {
129.           printf("Memory allocation failed. Unable to insert node.\n");
130.           return start;
131.        }
132.        newNode->data = num;
133.
134.        switch(choice) {
135.           case 1:
136.              newNode->next = start;
137.              start = newNode;
138.              break;
139.           case 2:
140.              if (start == NULL) {
141.                 start = newNode;
142.                 newNode->next = NULL;
143.              } else {
144.                 ptr = start;
145.                 while (ptr->next != NULL) {
146.                    ptr = ptr->next;
147.                 }
148.                 ptr->next = newNode;
149.                 newNode->next = NULL;
150.              }
151.              break;
152.           case 3:
153.              printf(" Enter the value after which to insert: ");
154.              scanf("%d", &val);
```

```
155.                 ptr = start;
156.                 while (ptr != NULL && ptr->data != val) {
157.                     ptr = ptr->next;
158.                 }
159.                 if (ptr == NULL) {
160.                     printf("Value not found in the list.\n");
161.                     free(newNode);
162.                 } else {
163.                     newNode->next = ptr->next;
164.                     ptr->next = newNode;
165.                 }
166.                 break;
167.             case 4:
168.                 printf(" Enter the value before which to insert: ");
169.                 scanf("%d", &val);
170.                 ptr = start;
171.                 prev = NULL;
172.                 while (ptr != NULL && ptr->data != val) {
173.                     prev = ptr;
174.                     ptr = ptr->next;
175.                 }
176.                 if (ptr == NULL) {
177.                     printf("Value not found in the list.\n");
178.                     free(newNode);
179.                 } else {
180.                     newNode->next = ptr;
181.                     if (prev == NULL) {
182.                         start = newNode;
183.                     } else {
184.                         prev->next = newNode;
185.                     }
186.                 }
187.                 break;
188.         }
189.
190.         return start;
191.     }
192.
193.     struct node *delete_Node(struct node *start) {
194.         if (start == NULL) {
195.             printf("\nList is empty. Nothing to delete.\n");
196.             return start;
197.         }
198.
199.         struct node *ptr = start, *prev = NULL;
200.         int choice, pos;
```

```
201.
202.        printf("\n Choose method of deletion:"
203.            "\n 1:) Delete at the beginning"
204.            "\n 2:) Delete at the end"
205.            "\n 3:) Delete at a specific position"
206.            "\n\n Enter your choice: ");
207.        scanf("%d", &choice);
208.        if(choice < 1 || choice > 3) {
209.           printf("Invalid choice. Please try again.\n");
210.           return start;
211.        }
212.
213.        switch(choice) {
214.           case 1:
215.              start = start->next;
216.              free(ptr);
217.              break;
218.           case 2:
219.              while (ptr->next != NULL) {
220.                 prev = ptr;
221.                 ptr = ptr->next;
222.              }
223.              if (prev != NULL) {
224.                 prev->next = NULL;
225.              } else {
226.                 start = NULL; // If there was only one node
227.              }
228.              free(ptr);
229.              break;
230.           case 3:
231.              printf("\n Enter the position to be deleted: ");
232.              scanf("%d", &pos);
233.
234.              if (pos < 1) {
235.                 printf("\nInvalid position.\n");
236.                 return start;
237.              }
238.
239.              int count = 1;
240.              ptr = start;
241.              while (ptr != NULL && count < pos) {
242.                 prev = ptr;
243.                 ptr = ptr->next;
244.                 count++;
245.              }
246.
```

```
                if (ptr == NULL) {
                    printf("\nPosition not found in the list.\n");
                } else {
                    if (prev != NULL) {
                        prev->next = ptr->next;
                    } else {
                        start = ptr->next; // If deleting the first node
                    }
                    free(ptr);
                }
                break;
            }
            return start;
        }
```

Q3) ***C program to perform operation on single linked list(L.L) without using pointers and functions:(It is just a try to make such type of program)***

```
#include <stdio.h>
#define MAX_SIZE 1000

typedef struct Node {
    int data[MAX_SIZE];
    int size;
} Node;

void clear_input_buffer() {
    while (getchar() != '\n');
}

int main() {
    Node list = {.size = 0};
    int option, num, location;

    // Main menu loop
    do {
        printf("\n\n\t\t----------MAIN MENU----------"
            "\n 1:) Create a List"
            "\n 2:) Display the List"
            "\n 3:) Insert a node in the List"
            "\n 4:) Delete a node from the List"
            "\n 5:) Exit the program"
            "\n\n Enter your choice: ");

        if (scanf("%d", &option) != 1) {
            printf("Invalid input. Please enter a number.\n");
            clear_input_buffer();
            continue;
        }
```

```
32.
33.     if (option < 1 || option > 5) {
34.         printf("Invalid choice. Please try again.\n");
35.         continue;
36.     }
37.
38.     clear_input_buffer(); // Clear any excess input from the buffer
39.
40.     switch(option) {
41.         case 1: {
42.             if (list.size >= MAX_SIZE) {
43.                 printf("List is already full.\n");
44.                 break;
45.             }
46.             printf("\n Type data to create a list (enter -1 to end):\n ");
47.             while (scanf("%d", &num) == 1 && num != -1 && list.size < MAX_SIZE) {
48.                 list.data[list.size++] = num;
49.                 if (list.size < MAX_SIZE) {
50.                     printf(" Enter more data (enter -1 to end): ");
51.                 }
52.             }
53.             clear_input_buffer();
54.             break;
55.         }
56.         case 2: {
57.             if (list.size == 0) {
58.                 printf("Linked list is Empty\n");
59.             } else {
60.                 for (int i = 0; i < list.size; i++) {
61.                     printf("%d -> ", list.data[i]);
62.                 }
63.                 printf("NULL\n");
64.             }
65.             break;
66.         }
67.         case 3: {
68.             if (list.size >= MAX_SIZE) {
69.                 printf("List is full. Cannot insert new elements.\n");
70.                 break;
71.             }
72.             printf("\n Enter a number to insert in the list: ");
73.             if (scanf("%d", &num) != 1) {
74.                 printf("Invalid input.\n");
75.                 clear_input_buffer();
76.                 break;
77.             }
78.             printf("\n Where do you want to insert the number? (1: Beginning, 2: End, 3:
    Specific Location): ");
79.             if (scanf("%d", &location) != 1 || location < 1 || location > 3) {
80.                 printf("Invalid location choice.\n");
81.                 clear_input_buffer();
```

```
82.                 break;
83.             }
84.             clear_input_buffer();
85.
86.             // Assume adding to the end for simplicity
87.             if (location == 1) { // Beginning
88.                 for (int i = list.size; i > 0; i--) {
89.                     list.data[i] = list.data[i - 1];
90.                 }
91.                 list.data[0] = num;
92.                 list.size++;
93.             } else if (location == 2) { // End
94.                 list.data[list.size++] = num;
95.             } else { // Specific Location
96.                 int specific_location;
97.                 printf("Enter the specific position (1 to %d): ", list.size + 1);
98.                 if (scanf("%d", &specific_location) != 1 || specific_location < 1 ||
    specific_location > list.size + 1) {
99.                     printf("Invalid position.\n");
100.                        clear_input_buffer();
101.                        break;
102.                    }
103.                    for (int i = list.size; i >= specific_location; i--) {
104.                        list.data[i] = list.data[i - 1];
105.                    }
106.                    list.data[specific_location - 1] = num;
107.                    list.size++;
108.                }
109.                break;
110.            }
111.            case 4: {
112.                if (list.size == 0) {
113.                    printf("Linked list is empty. Nothing to delete.\n");
114.                    break;
115.                }
116.                printf("\n Where do you want to delete a number from? (1: Beginning, 2:
    End, 3: Specific Location): ");
117.                if (scanf("%d", &location) != 1 || location < 1 || location > 3) {
118.                    printf("Invalid location choice.\n");
119.                    clear_input_buffer();
120.                    break;
121.                }
122.                clear_input_buffer();
123.
124.                // Actual deletion logic (simplified for specific cases)
125.                if (location == 1) { // Beginning
126.                    for (int i = 0; i < list.size - 1; i++) {
127.                        list.data[i] = list.data[i + 1];
128.                    }
129.                    list.size--;
130.                } else if (location == 2) { // End
```

```
                    list.size--;
                } else { // Specific Location
                    int specific_location;
                    printf("Enter the specific position to delete (1 to %d): ", list.size);
                    if (scanf("%d", &specific_location) != 1 || specific_location < 1 ||
specific_location > list.size) {
                        printf("Invalid position.\n");
                        clear_input_buffer();
                        break;
                    }
                    for (int i = specific_location - 1; i < list.size - 1; i++) {
                        list.data[i] = list.data[i + 1];
                    }
                    list.size--;
                }
                break;
            }
            case 5:
                printf("Exiting the program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while(option != 5);

    return 0;
}
```

Q4) ***C program to perform operation on single linked list(L.L) using functions:(It is just a try to make such type of program)***

```
#include <stdio.h>
#define MAX_SIZE 1000

typedef struct Node {
    int data[MAX_SIZE];
    int size;
} Node;

void insertatbeginning(Node* list, int num) {
    if (list->size < MAX_SIZE) {
        for (int i = list->size; i > 0; --i) {
            list->data[i] = list->data[i - 1];
        }
        list->data[0] = num;
        list->size++;
    } else {
        printf("List is full. Cannot insert more elements.\n");
    }
```

```
}

void insertatend(Node* list, int num) {
    if (list->size < MAX_SIZE) {
        list->data[list->size] = num;
        list->size++;
    } else {
        printf("List is full. Cannot insert more elements.\n");
    }
}

void insertatlocation(Node* list, int num, int location) {
    if (list->size < MAX_SIZE && location >= 0 && location < list->size) {
        for (int i = list->size; i > location; --i) {
            list->data[i] = list->data[i - 1];
        }
        list->data[location] = num;
        list->size++;
    } else {
        printf("Invalid location or list is full. Cannot insert more elements.\n");
    }
}

void deletefrombeginning(Node* list) {
    if (list->size > 0) {
        for (int i = 0; i < list->size - 1; i++) {
            list->data[i] = list->data[i + 1];
        }
        list->size--;
    } else {
        printf("List is empty. Cannot delete elements.\n");
    }
}

void deletefromend(Node* list) {
    if (list->size > 0) {
        list->size--;
    } else {
        printf("List is empty. Cannot delete elements.\n");
    }
}

void deletefromlocation(Node* list, int location) {
    if (list->size > 0 && location >= 0 && location < list->size) {
        for (int i = location; i < list->size - 1; i++) {
            list->data[i] = list->data[i + 1];
        }
        list->size--;
    } else {
        printf("Invalid location or list is empty. Cannot delete elements.\n");
    }
```

```
70. }
71.
72. void displaylist(Node list) {
73.     if (list.size == 0) {
74.         printf("Linked list is empty.\n");
75.         return;
76.     }
77.     printf("Linked List: ");
78.     for (int i = 0; i < list.size; i++) {
79.         printf("%d -> ", list.data[i]);
80.     }
81.     printf("NULL\n");
82. }
83.
84. int main() {
85.     Node list = {.size = 0};
86.     int option, num, location;
87.
88.     do {
89.         printf("\n\n\t\t----------MAIN MENU----------"
90.                "\n 1:) Create a List"
91.                "\n 2:) Display the List"
92.                "\n 3:) Insert a node in the List"
93.                "\n 4:) Delete a node from the List"
94.                "\n 5:) Exit the program"
95.                "\n\n Enter your choice: ");
96.         if (scanf("%d", &option) != 1) {
97.             printf("Invalid input. Please enter a number.\n");
98.             while (getchar() != '\n'); // clear the input buffer
99.             continue;
100.                }
101.
102.                switch(option) {
103.                    case 1:
104.                        printf("\n Type data to create a list (enter -1 to end):\n ");
105.                        while (scanf("%d", &num) == 1 && num != -1) {
106.                            if (list.size < MAX_SIZE) {
107.                                insertatend(&list, num);
108.                                printf(" Enter more data (enter -1 to end): ");
109.                            } else {
110.                                printf("List is full. Cannot insert more elements.\n");
111.                                break;
112.                            }
113.                        }
114.                        while (getchar() != '\n'); // clear the buffer
115.                        break;
116.                    case 2:
117.                        displaylist(list);
118.                        break;
119.                    case 3:
120.                        printf("\n Enter a number to insert in the list: ");
```

```
121.                 if (scanf("%d", &num) != 1) {
122.                     printf("Invalid input. Please enter a number.\n");
123.                     while (getchar() != '\n'); // clear the buffer
124.                     continue;
125.                 }
126.                 printf("\n Where do you want to insert the number? (1: Beginning, 2: End,
    3: Specific Location): ");
127.                 if (scanf("%d", &location) != 1 || location < 1 || location > 3) {
128.                     printf("Invalid choice. Please enter 1, 2, or 3.\n");
129.                     while (getchar() != '\n'); // clear the buffer
130.                     continue;
131.                 }
132.                 if (location == 1) {
133.                     insertatbeginning(&list, num);
134.                 } else if (location == 2) {
135.                     insertatend(&list, num);
136.                 } else {
137.                     printf("\n Enter the specific location (0 to %d): ", list.size);
138.                     if (scanf("%d", &location) != 1 || location < 0 || location > list.size) {
139.                         printf("Invalid location.\n");
140.                         while (getchar() != '\n'); // clear the buffer
141.                         continue;
142.                     }
143.                     insertatlocation(&list, num, location);
144.                 }
145.                 break;
146.             case 4:
147.                 printf("\n Where do you want to delete a number? (1: Beginning, 2: End, 3:
    Specific Location): ");
148.                 if (scanf("%d", &location) != 1 || location < 1 || location > 3) {
149.                     printf("Invalid choice. Please enter 1, 2, or 3.\n");
150.                     while (getchar() != '\n'); // clear the buffer
151.                     continue;
152.                 }
153.                 if (location == 1) {
154.                     deletefrombeginning(&list);
155.                 } else if (location == 2) {
156.                     deletefromend(&list);
157.                 } else {
158.                     printf("\n Enter the specific location to delete (0 to %d): ", list.size - 1);
159.                     if (scanf("%d", &location) != 1 || location < 0 || location >= list.size) {
160.                         printf("Invalid location.\n");
161.                         while (getchar() != '\n'); // clear the buffer
162.                         continue;
163.                     }
164.                     deletefromlocation(&list, location);
165.                 }
166.                 break;
167.             case 5:
168.                 printf("Exiting the program.\n");
169.                 break;
```

```
170.                 default:
171.                     printf("Invalid choice. Please try again.\n");
172.                 }
173.             } while(option != 5);
174.
175.             return 0;
176.         }
```

Algorithms for question 1-4

1] p

1. Start
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for a linked list node, containing an integer data field and a pointer to the next node.
4. In the `main` function, initialize a pointer `start` to `NULL` to represent the start of the linked list.
5. Declare an integer variable `option` to store the user's menu choice.
6. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message and clear the input buffer.
   - If the choice is not within the valid range, display an error message.
7. Use a `switch` statement to handle the user's choice:
   - Case 1: Create a Linked List
     - Declare pointers for the new node and a temporary pointer.
     - Prompt the user to enter numbers to create the linked list, ending with `-1`.
     - For each number entered:
       - Allocate memory for a new node.
       - If memory allocation fails, display an error message and break the loop.
       - Assign the entered number to the new node's data field and set its `next` pointer to `NULL`.
       - If the list is empty, set the `start` pointer to the new node.
       - Otherwise, traverse the list to find the last node and set its `next` pointer to the new node.
   - Case 2: Display the Linked List
     - If the list is empty, display a message indicating that the list is empty.
     - Otherwise, traverse the list and print each node's data followed by `->`.
   - Case 3: Insert a Node in the Linked List
     - Declare pointers for the new node, a temporary pointer, and a previous pointer.
     - Prompt the user to choose the method of insertion.
     - Use a nested `switch` statement to handle the insertion method:
       - Case 1: Insert at the Beginning
         - Allocate memory for a new node.
         - If memory allocation fails, display an error message.
         - Assign the entered number to the new node's data field and set its `next` pointer to the current `start` node.
         - Update the `start` pointer to the new node.
       - Case 2: Insert at the End
         - Allocate memory for a new node.
         - If memory allocation fails, display an error message.
         - Assign the entered number to the new node's data field and set its `next` pointer to `NULL`.
         - Traverse the list to find the last node and set its `next` pointer to the new node.

- Case 3: Insert After a Specific Value
  - Traverse the list to find the node with the specified value.
  - If the value is not found, display an error message.
  - Otherwise, allocate memory for a new node, assign the entered number to its data field, and insert it after the found node.
- Case 4: Insert Before a Specific Value
  - Traverse the list to find the node with the specified value.
  - If the value is not found, display an error message.
  - Otherwise, allocate memory for a new node, assign the entered number to its data field, and insert it before the found node.

- Case 4: Delete a Node in the Linked List
  - Declare pointers for the current node, a previous pointer, and an integer for the position.
  - Prompt the user to choose the method of deletion.
  - Use a nested `switch` statement to handle the deletion method:
    - Case 1: Delete at the Beginning
      - Update the `start` pointer to the second node in the list and free the memory of the first node
    - Case 2: Delete at the End
      - Traverse the list to find the last node and free its memory.
      - If there is only one node, set the `start` pointer to `NULL`.
    - Case 3: Delete at a Specific Position
      - Traverse the list to find the node at the specified position and free its memory.
      - If the node is the first one, update the `start` pointer to the second node.
      - Otherwise, update the `next` pointer of the previous node to skip the deleted node.
- Case 5: Exit the Program
  - Display a message indicating that the program is exiting.
- If the user's choice is not within the valid range, display an error message.

8. End

2] f p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for a linked list node, containing an integer data field and a pointer to the next node.
4. Declare function prototypes for creating a linked list, displaying the linked list, inserting a node, and deleting a node.
5. In the `main` function, initialize a pointer `start` to `NULL` to represent the start of the linked list.
6. Declare an integer variable `option` to store the user's menu choice.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message and clear the input buffer.
   - If the choice is not within the valid range, display an error message.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a Linked List--

- Declare pointers for the new node and a temporary pointer.
- Prompt the user to enter numbers to create the linked list, ending with `-1`.
- For each number entered:
- Allocate memory for a new node.
- If memory allocation fails, display an error message and return the current state of the linked list.
- Assign the entered number to the new node's data field and set its `next` pointer to `NULL`.
- If the list is empty, set the `start` pointer to the new node.
- Otherwise, traverse the list to find the last node and set its `next` pointer to the new node.
- --Case 2: Display the Linked List--
- If the list is empty, display a message indicating that the list is empty.
- Otherwise, traverse the list and print each node's data followed by `->`.
- --Case 3: Insert a Node in the Linked List--
- Declare pointers for the new node, a temporary pointer, and a previous pointer.
- Prompt the user to choose the method of insertion.
- Use a nested `switch` statement to handle the insertion method:
- --Case 1: Insert at the Beginning--
- Allocate memory for a new node.
- If memory allocation fails, display an error message.
- Assign the entered number to the new node's data field and set its `next` pointer to the current `start` node.
- Update the `start` pointer to the new node.
- --Case 2: Insert at the End--
- Allocate memory for a new node.
- If memory allocation fails, display an error message.
- Assign the entered number to the new node's data field and set its `next` pointer to `NULL`.
- Traverse the list to find the last node and set its `next` pointer to the new node.
- --Case 3: Insert After a Specific Value--
- Traverse the list to find the node with the specified value.
- If the value is not found, display an error message.
- Otherwise, allocate memory for a new node, assign the entered number to its data field, and insert it after the found node.
- --Case 4: Insert Before a Specific Value--
- Traverse the list to find the node with the specified value.
- If the value is not found, display an error message.
- Otherwise, allocate memory for a new node, assign the entered number to its data field, and insert it before the found node.
- --Case 4: Delete a Node in the Linked List--
- Declare pointers for the current node, a previous pointer, and an integer for the position.
- Prompt the user to choose the method of deletion.
- Use a nested `switch` statement to handle the deletion method:
- --Case 1: Delete at the Beginning--
- Update the `start` pointer to the second node in the list and free the memory of the first node.
- --Case 2: Delete at the End--
- Traverse the list to find the last node and free its memory.

- If there is only one node, set the `start` pointer to `NULL`.
- --Case 3: Delete at a Specific Position--
- Traverse the list to find the node at the specified position and free its memory.
- If the node is the first one, update the `start` pointer to the second node.
- Otherwise, update the `next` pointer of the previous node to skip the deleted node.
- --Case 5: Exit the Program--
- Display a message indicating that the program is exiting.
- If the user's choice is not within the valid range, display an error message.

9. --End--

3] without f p aka normal(n)

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Define a constant `MAX_SIZE` to represent the maximum size of the list.
4. Define a structure `Node` for the list, containing an integer array `data` of size `MAX_SIZE` and an integer `size` to keep track of the current size of the list.
5. Declare a function `clear_input_buffer` to clear the input buffer.
6. In the `main` function, initialize a `Node` variable `list` with `size` set to `0`.
7. Declare integer variables `option`, `num`, and `location` to store the user's menu choice, the number to be inserted or deleted, and the location for insertion or deletion.
8. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
9. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a List--
     - If the list is already full, display a message indicating that the list is full.
     - Otherwise, prompt the user to enter numbers to create the list, ending with `-1`.
     - For each number entered, add it to the list's `data` array and increment the `size`.
   - --Case 2: Display the List--
     - If the list is empty, display a message indicating that the list is empty.
     - Otherwise, iterate through the list's `data` array and print each element followed by `->`.
   - --Case 3: Insert a Node in the List--
     - If the list is full, display a message indicating that the list is full.
     - Otherwise, prompt the user to enter a number and choose the location for insertion.
     - Use a nested `if` statement to handle the insertion location:
       - --Beginning--: Shift all elements to the right and insert the new number at the beginning.
       - --End--: Add the new number at the end of the list.
       - --Specific Location--: Shift elements to the right starting from the specified location and insert the new number.
   - --Case 4: Delete a Node from the List--
     - If the list is empty, display a message indicating that the list is empty.

- Otherwise, prompt the user to choose the location for deletion.
- Use a nested `if` statement to handle the deletion location:
- --Beginning--: Shift all elements to the left.
- --End--: Simply decrement the `size`.
- --Specific Location--: Shift elements to the left starting from the specified location.

- --Case 5: Exit the Program--
- Display a message indicating that the program is exiting.

- If the user's choice is not within the valid range, display an error message.

10. --End--

4] with f

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Define a constant `MAX_SIZE` to represent the maximum size of the list.
4. Define a structure `Node` for the list, containing an integer array `data` of size `MAX_SIZE` and an integer `size` to keep track of the current size of the list.
5. Declare functions for inserting at the beginning, inserting at the end, inserting at a specific location, deleting from the beginning, deleting from the end, and deleting from a specific location.
6. Declare a function `displaylist` to display the contents of the list.
7. In the `main` function, initialize a `Node` variable `list` with `size` set to `0`.
8. Declare integer variables `option`, `num`, and `location` to store the user's menu choice, the number to be inserted or deleted, and the location for insertion or deletion.
9. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
10. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a List--
     - Prompt the user to enter numbers to create the list, ending with `-1`.
     - For each number entered, call `insertatend` to add it to the list.
   - --Case 2: Display the List--
     - Call `displaylist` to display the contents of the list.
   - --Case 3: Insert a Node in the List--
     - Prompt the user to enter a number and choose the location for insertion.
     - Use a nested `if` statement to handle the insertion location:
       - --Beginning--: Call `insertatbeginning`.
       - --End--: Call `insertatend`.
       - --Specific Location--: Call `insertatlocation`.
   - --Case 4: Delete a Node from the List--
     - Prompt the user to choose the location for deletion.
     - Use a nested `if` statement to handle the deletion location:
       - --Beginning--: Call `deletefrombeginning`.
       - --End--: Call `deletefromend`.
       - --Specific Location--: Call `deletefromlocation`.

- --Case 5: Exit the Program--
    - Display a message indicating that the program is exiting.
- If the user's choice is not within the valid range, display an error message.

11. --End--

Q5) *C program to perform operation on doubly linked list(L.L) using pointers:*

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

// Define the structure for a node. It is a self-referential structure
struct node {
    int data;
    struct node *prev;
    struct node *next;
};

// Function to clear the input buffer
void clear_input_buffer() {
    while (getchar() != '\n');
}

int main() {
    struct node *start = NULL;
    int option;

    printf("Namaskaram! This program demonstrates insertion and deletion in a doubly linked list.\n");

    // Main menu loop
    do {
        printf("\n\n\t\t----------MAIN MENU----------"
               "\n 1:) Create a Doubly Linked List"
               "\n 2:) Display the Doubly Linked List"
               "\n 3:) Insert a node in the DLL"
               "\n 4:) Delete a node in the DLL"
               "\n 5:) Exit the program"
               "\n\n Enter your choice : ");

        if (scanf("%d", &option) != 1) {
            printf("Invalid input. Please enter a number.\n");
            clear_input_buffer(); // clear the input buffer
            continue;
        }
        if (option < 1 || option > 5) {
            printf("Invalid choice. Please try again.\n");
            continue;
        }

        switch(option) {
            case 1: {
                // Create a doubly linked list
                struct node *newNode, *ptr;
                int num;

```

```
48.             printf("\nEnter numbers to create a Doubly Linked List (enter -1 to end):\n");
49.             while (scanf("%d", &num) == 1 && num != -1) {
50.                 newNode = (struct node *)malloc(sizeof(struct node));
51.                 if (newNode == NULL) {
52.                     printf("Memory allocation failed. Doubly linked list creation aborted.\n");
53.                     break;
54.                 }
55.
56.                 newNode->data = num;
57.                 newNode->prev = NULL;
58.                 newNode->next = NULL;
59.
60.                 if (start == NULL) {
61.                     start = newNode;
62.                 } else {
63.                     ptr = start;
64.                     while (ptr->next != NULL) {
65.                         ptr = ptr->next;
66.                     }
67.                     ptr->next = newNode;
68.                     newNode->prev = ptr;
69.                 }
70.                 printf("Enter data to create the Doubly Linked List (enter -1 to end): ");
71.             }
72.             clear_input_buffer();
73.             break;
74.         }
75.         case 2: {
76.             // Display the doubly linked list
77.             struct node *ptr = start;
78.             if (start == NULL) {
79.                 printf("\nList is empty.");
80.             } else {
81.                 printf("\nDoubly Linked List: NULL <- ");
82.                 while (ptr != NULL) {
83.                     printf("%d <-> ", ptr->data);
84.                     ptr = ptr->next;
85.                 }
86.                 printf("NULL\n");
87.             }
88.             break;
89.         }
90.         case 3: {
91.             // Insert a node into the doubly linked list
92.             struct node *newNode, *ptr;
93.             int num, val, choice;
94.
95.             printf("\nChoose method of insertion:"
96.                    "\n 1:) Insert at the beginning"
97.                    "\n 2:) Insert at the end"
98.                    "\n 3:) Insert after a specific value"
```

```
99.             "\n 4:) Insert before a specific value"
100.                  "\n\nEnter your choice: ");
101.            if (scanf("%d", &choice) != 1 || choice < 1 || choice > 4) {
102.               printf("Invalid input. Please enter a valid choice.\n");
103.               clear_input_buffer();
104.               continue;
105.            }
106.
107.            switch(choice) {
108.               case 1:
109.                  printf("\nEnter the data to be inserted at the beginning: ");
110.                  if (scanf("%d", &num) != 1) {
111.                     printf("Invalid input.\n");
112.                     clear_input_buffer();
113.                     continue;
114.                  }
115.
116.                  newNode = (struct node *)malloc(sizeof(struct node));
117.                  if (newNode == NULL) {
118.                     printf("Memory allocation failed. Unable to insert node.\n");
119.                     break;
120.                  }
121.                  newNode->data = num;
122.                  newNode->prev = NULL;
123.                  newNode->next = start;
124.                  if (start != NULL) {
125.                     start->prev = newNode;
126.                  }
127.                  start = newNode;
128.                  break;
129.
130.               case 2:
131.                  printf("\nEnter the data to be inserted at the end: ");
132.                  if (scanf("%d", &num) != 1) {
133.                     printf("Invalid input.\n");
134.                     clear_input_buffer();
135.                     continue;
136.                  }
137.
138.                  newNode = (struct node *)malloc(sizeof(struct node));
139.                  if (newNode == NULL) {
140.                     printf("Memory allocation failed. Unable to insert node.\n");
141.                     break;
142.                  }
143.                  newNode->data = num;
144.                  newNode->next = NULL;
145.
146.                  if (start == NULL) {
147.                     newNode->prev = NULL;
148.                     start = newNode;
149.                  } else {
```

```
                ptr = start;
                while (ptr->next != NULL) {
                    ptr = ptr->next;
                }
                ptr->next = newNode;
                newNode->prev = ptr;
            }
            break;

        case 3:
            printf("\nEnter the data to be inserted after a specific value: ");
            if (scanf("%d", &num) != 1) {
                printf("Invalid input.\n");
                clear_input_buffer();
                continue;
            }
            printf("Enter the value after which you want to insert: ");
            if (scanf("%d", &val) != 1) {
                printf("Invalid input.\n");
                clear_input_buffer();
                continue;
            }

            ptr = start;
            while (ptr != NULL && ptr->data != val) {
                ptr = ptr->next;
            }

            if (ptr == NULL) {
                printf("Value not found in the list.\n");
            } else {
                newNode = (struct node *)malloc(sizeof(struct node));
                if (newNode == NULL) {
                    printf("Memory allocation failed. Unable to insert node.\n");
                    break;
                }
                newNode->data = num;
                newNode->next = ptr->next;
                newNode->prev = ptr;
                if (ptr->next != NULL) {
                    ptr->next->prev = newNode;
                }
                ptr->next = newNode;
            }
            break;

        case 4:
            printf("\nEnter the data to be inserted before a specific value: ");
            if (scanf("%d", &num) != 1) {
                printf("Invalid input.\n");
                clear_input_buffer();
```

```
                    continue;
                }
                printf("Enter the value before which you want to insert: ");
                if (scanf("%d", &val) != 1) {
                    printf("Invalid input.\n");
                    clear_input_buffer();
                    continue;
                }

                ptr = start;
                while (ptr != NULL && ptr->data != val) {
                    ptr = ptr->next;
                }

                if (ptr == NULL) {
                    printf("Value not found in the list.\n");
                } else {
                    newNode = (struct node *)malloc(sizeof(struct node));
                    if (newNode == NULL) {
                        printf("Memory allocation failed. Unable to insert node.\n");
                        break;
                    }
                    newNode->data = num;
                    newNode->prev = ptr->prev;
                    newNode->next = ptr;
                    if (ptr->prev != NULL) {
                        ptr->prev->next = newNode;
                    } else {
                        start = newNode;
                    }
                    ptr->prev = newNode;
                }
                break;

            default:
                printf("\nInvalid choice.\n");
        }
        break;
    }
    case 4: {
        // Delete a node from the doubly linked list
        struct node *ptr;
        int choice, val;

        if (start == NULL) {
            printf("\nList is empty. Nothing to delete.");
            break;
        }

        printf("\nChoose method of deletion:"
               "\n 1:) Delete at the beginning"
```

```
252.                 "\n 2:) Delete at the end"
253.                 "\n 3:) Delete a specific value"
254.                 "\n\nEnter your choice: ");
255.             if (scanf("%d", &choice) != 1 || choice < 1 || choice > 3) {
256.                 printf("Invalid input. Please enter a valid choice.\n");
257.                 clear_input_buffer();
258.                 continue;
259.             }
260.
261.             switch(choice) {
262.                 case 1:
263.                     ptr = start;
264.                     start = start->next;
265.                     if (start != NULL) {
266.                         start->prev = NULL;
267.                     }
268.                     free(ptr);
269.                     break;
270.
271.                 case 2:
272.                     ptr = start;
273.                     while (ptr->next != NULL) {
274.                         ptr = ptr->next;
275.                     }
276.                     if (ptr->prev != NULL) {
277.                         ptr->prev->next = NULL;
278.                     } else {
279.                         start = NULL;
280.                     }
281.                     free(ptr);
282.                     break;
283.
284.                 case 3:
285.                     printf("\nEnter the value to be deleted: ");
286.                     if (scanf("%d", &val) != 1) {
287.                         printf("Invalid input.\n");
288.                         clear_input_buffer();
289.                         continue;
290.                     }
291.
292.                     ptr = start;
293.                     while (ptr != NULL && ptr->data != val) {
294.                         ptr = ptr->next;
295.                     }
296.
297.                     if (ptr == NULL) {
298.                         printf("Value not found in the list.\n");
299.                     } else {
300.                         if (ptr->prev != NULL) {
301.                             ptr->prev->next = ptr->next;
302.                         } else {
```

```
                start = ptr->next;
            }
            if (ptr->next != NULL) {
                ptr->next->prev = ptr->prev;
            }
            free(ptr);
        }
        break;

    default:
        printf("\nInvalid choice.\n");
    }
    break;
}
case 5:
    printf("\nExiting...\n");
    break;
default:
    printf("\nInvalid option. Please try again.");
}
} while(option != 5);

return 0;
}
```

Q6) ***C program to perform operation on doubly linked list(L.L) using functions-pointers:***

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

// Define the structure for a node. It is a self-referential structure
struct node {
    int data;
    struct node *prev;
    struct node *next;
};

// Function to clear the input buffer
void clear_input_buffer() {
    while (getchar() != '\n');
}

// Function prototypes for managing the doubly linked list
struct node *create_dll();
void display_dll(struct node *);
struct node *insert_Node(struct node *);
```

```
20. struct node *delete_Node(struct node *);
21.
22. int main() {
23.     struct node *start = NULL;
24.     int option;
25.
26.     printf("Namaskaram! This program demonstrates insertion and deletion in a doubly
    linked list.\n");
27.
28.     // Main menu loop
29.     do {
30.         printf("\n\n\t\t----------MAIN MENU----------"
31.                "\n 1:) Create a Doubly Linked List"
32.                "\n 2:) Display the Doubly Linked List"
33.                "\n 3:) Insert a node in the DLL"
34.                "\n 4:) Delete a node in the DLL"
35.                "\n 5:) Exit the program"
36.                "\n\n Enter your choice : ");
37.         if (scanf("%d", &option) != 1) {
38.             printf("Invalid input. Please enter a number.\n");
39.             clear_input_buffer();
40.             continue;
41.         }
42.         if (option < 1 || option > 5) {
43.             printf("Invalid choice. Please try again.\n");
44.             continue;
45.         }
46.
47.         switch(option) {
48.             case 1:
49.                 start = create_dll();
50.                 break;
51.             case 2:
52.                 display_dll(start);
53.                 break;
54.             case 3:
55.                 start = insert_Node(start);
56.                 break;
57.             case 4:
58.                 start = delete_Node(start);
59.                 break;
60.             case 5:
61.                 printf("\nExiting...\n");
62.                 break;
63.             default:
64.                 printf("\nInvalid option. Please try again.\n");
```

```
65.         }
66.     } while(option != 5);
67.
68.     return 0;
69. }
70.
71. struct node *create_dll() {
72.     struct node *start = NULL, *newNode, *ptr;
73.     int num;
74.
75.     printf("\nEnter numbers to create a Doubly Linked List (enter -1 to end):\n");
76.     while (scanf("%d", &num) == 1 && num != -1) {
77.         newNode = (struct node *)malloc(sizeof(struct node));
78.         if (newNode == NULL) {
79.             printf("Memory allocation failed. Doubly linked list creation aborted.\n");
80.             return start;
81.         }
82.
83.         newNode->data = num;
84.         newNode->prev = NULL;
85.         newNode->next = NULL;
86.
87.         if (start == NULL) {
88.             start = newNode;
89.         } else {
90.             ptr = start;
91.             while (ptr->next != NULL) {
92.                 ptr = ptr->next;
93.             }
94.             ptr->next = newNode;
95.             newNode->prev = ptr;
96.         }
97.         printf("Enter data to create the Doubly Linked List (enter -1 to end): ");
98.     }
99.     clear_input_buffer();
100.     return start;
101. }
102.
103. void display_dll(struct node *start) {
104.     struct node *ptr = start;
105.     if (ptr == NULL) {
106.         printf("\nList is empty.\n");
107.         return;
108.     }
109.
110.     printf("\nDoubly Linked List: NULL <- ");
```

```
111.    while (ptr != NULL) {
112.        printf("%d <-> ", ptr->data);
113.        ptr = ptr->next;
114.    }
115.    printf("NULL\n");
116.}
117.
118.struct node *insert_Node(struct node *start) {
119.    struct node *newNode, *ptr;
120.    int num, val, choice;
121.
122.    printf("\nChoose method of insertion:"
123.           "\n 1:) Insert at the beginning"
124.           "\n 2:) Insert at the end"
125.           "\n 3:) Insert after a specific value"
126.           "\n 4:) Insert before a specific value"
127.           "\n\nEnter your choice: ");
128.    if (scanf("%d", &choice) != 1 || choice < 1 || choice > 4) {
129.        printf("Invalid input. Please enter a valid choice.\n");
130.        clear_input_buffer();
131.        return start;
132.    }
133.
134.    printf("\nEnter the data to be inserted: ");
135.    if (scanf("%d", &num) != 1) {
136.        printf("Invalid input. Please enter a number.\n");
137.        clear_input_buffer();
138.        return start;
139.    }
140.
141.    newNode = (struct node *)malloc(sizeof(struct node));
142.    if (newNode == NULL) {
143.        printf("Memory allocation failed. Unable to insert node.\n");
144.        return start;
145.    }
146.    newNode->data = num;
147.
148.    switch(choice) {
149.        case 1: // Insert at the beginning
150.            newNode->prev = NULL;
151.            newNode->next = start;
152.            if (start != NULL) {
153.                start->prev = newNode;
154.            }
155.            start = newNode;
156.            break;
```

```
157.
158.     case 2: // Insert at the end
159.         newNode->next = NULL;
160.         if (start == NULL) {
161.             newNode->prev = NULL;
162.             start = newNode;
163.         } else {
164.             ptr = start;
165.             while (ptr->next != NULL) {
166.                 ptr = ptr->next;
167.             }
168.             ptr->next = newNode;
169.             newNode->prev = ptr;
170.         }
171.         break;
172.
173.     case 3: // Insert after a specific value
174.         printf("Enter the value after which you want to insert: ");
175.         if (scanf("%d", &val) != 1) {
176.             printf("Invalid input. Please enter a number.\n");
177.             clear_input_buffer();
178.             free(newNode);
179.             return start;
180.         }
181.         ptr = start;
182.         while (ptr != NULL && ptr->data != val) {
183.             ptr = ptr->next;
184.         }
185.         if (ptr == NULL) {
186.             printf("Value not found in the list.\n");
187.             free(newNode);
188.         } else {
189.             newNode->prev = ptr;
190.             newNode->next = ptr->next;
191.             if (ptr->next != NULL) {
192.                 ptr->next->prev = newNode;
193.             }
194.             ptr->next = newNode;
195.         }
196.         break;
197.
198.     case 4: // Insert before a specific value
199.         printf("Enter the value before which you want to insert: ");
200.         if (scanf("%d", &val) != 1) {
201.             printf("Invalid input. Please enter a number.\n");
202.             clear_input_buffer();
```

```
203.             free(newNode);
204.             return start;
205.         }
206.         ptr = start;
207.         while (ptr != NULL && ptr->data != val) {
208.             ptr = ptr->next;
209.         }
210.         if (ptr == NULL) {
211.             printf("Value not found in the list.\n");
212.             free(newNode);
213.         } else {
214.             newNode->next = ptr;
215.             newNode->prev = ptr->prev;
216.             if (ptr->prev != NULL) {
217.                 ptr->prev->next = newNode;
218.             } else {
219.                 start = newNode;
220.             }
221.             ptr->prev = newNode;
222.         }
223.         break;
224.
225.     default:
226.         free(newNode);
227.         printf("\nInvalid choice.\n");
228.         return start;
229. }
230.
231. return start;
232.}
233.
234.struct node *delete_Node(struct node *start) {
235.  struct node *ptr;
236.  int choice, val;
237.
238.  printf("\nChoose method of deletion:"
239.       "\n 1:) Delete at the beginning"
240.       "\n 2:) Delete at the end"
241.       "\n 3:) Delete a specific value"
242.       "\n\nEnter your choice: ");
243.  if (scanf("%d", &choice) != 1 || choice < 1 || choice > 3) {
244.     printf("Invalid input. Please enter a valid choice.\n");
245.     clear_input_buffer();
246.     return start;
247.  }
248.
```

```
249.    if (start == NULL) {
250.        printf("\nList is empty. Nothing to delete.\n");
251.        return start;
252.    }
253.
254.    switch(choice) {
255.        case 1: // Delete at the beginning
256.            ptr = start;
257.            start = start->next;
258.            if (start != NULL) {
259.                start->prev = NULL;
260.            }
261.            free(ptr);
262.            break;
263.
264.        case 2: // Delete at the end
265.            ptr = start;
266.            while (ptr->next != NULL) {
267.                ptr = ptr->next;
268.            }
269.            if (ptr->prev != NULL) {
270.                ptr->prev->next = NULL;
271.            } else {
272.                start = NULL;
273.            }
274.            free(ptr);
275.            break;
276.
277.        case 3: // Delete a specific value
278.            printf("\nEnter the value to be deleted: ");
279.            if (scanf("%d", &val) != 1) {
280.                printf("Invalid input. Please enter a number.\n");
281.                clear_input_buffer();
282.                return start;
283.            }
284.            ptr = start;
285.            while (ptr != NULL && ptr->data != val) {
286.                ptr = ptr->next;
287.            }
288.            if (ptr == NULL) {
289.                printf("\nValue not found in the list.\n");
290.            } else {
291.                if (ptr->prev != NULL) {
292.                    ptr->prev->next = ptr->next;
293.                } else {
294.                    start = ptr->next;
```

```
295.                }
296.                if (ptr->next != NULL) {
297.                    ptr->next->prev = ptr->prev;
298.                }
299.                free(ptr);
300.            }
301.            break;
302.
303.        default:
304.            printf("\nInvalid choice.\n");
305.            return start;
306.    }
307.
308.    return start;
309.}
```

Algorithms for question 5 and 6

1] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for the doubly linked list, containing an integer `data`, and two pointers `prev` and `next` to the previous and next nodes.
4. Declare a function `clear_input_buffer` to clear the input buffer in case of invalid input.
5. In the `main` function, initialize a `node` pointer `start` to `NULL`.
6. Declare an integer variable `option` to store the user's menu choice.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a Doubly Linked List--
     - Prompt the user to enter numbers to create the list, ending with `-1`.
     - For each number entered, allocate memory for a new node, set its `data` to the entered number, and link it to the end of the list.
   - --Case 2: Display the Doubly Linked List--
     - Traverse the list from the start, printing each node's `data`.
   - --Case 3: Insert a Node in the Doubly Linked List--
     - Prompt the user to choose the method of insertion and enter the necessary data.
     - Use a nested `switch` statement to handle the insertion method:
       - --Beginning--: Allocate memory for a new node, set its `data`, and link it to the beginning of the list.
       - --End--: Allocate memory for a new node, set its `data`, and link it to the end of the list.
       - --After a Specific Value--: Traverse the list to find the specified value, allocate memory for a new node, set its `data`, and link it after the found node.
       - --Before a Specific Value--: Traverse the list to find the specified value, allocate memory for a new node, set its `data`, and link it before the found node.
   - --Case 4: Delete a Node from the Doubly Linked List--
     - Prompt the user to choose the method of deletion and enter the necessary data.
     - Use a nested `switch` statement to handle the deletion method:
       - --Beginning--: Remove the first node from the list and adjust the `start` pointer.
       - --End--: Traverse the list to find the last node, remove it, and adjust the `next` pointer of the second-to-last node.
       - --Specific Value--: Traverse the list to find the specified value, remove the found node, and adjust the `next` and `prev` pointers of the adjacent nodes.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.

- If the user's choice is not within the valid range, display an error message.
9. --End--

2] fp

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for the doubly linked list node, containing an integer `data` and pointers to the previous and next nodes.
4. Declare a function `clear_input_buffer` to clear the input buffer.
5. Declare function prototypes for managing the doubly linked list, including creating the list, displaying its contents, inserting nodes, and deleting nodes.
6. In the `main` function, initialize a pointer `start` to `NULL` to represent the start of the doubly linked list.
7. Declare an integer variable `option` to store the user's menu choice.
8. Display a welcome message to the user.
9. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
10. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a Doubly Linked List--
     - Call `create_dll` to create the list.
   - --Case 2: Display the Doubly Linked List--
     - Call `display_dll` to display the contents of the list.
   - --Case 3: Insert a Node in the DLL--
     - Call `insert_Node` to insert a node in the list.
   - --Case 4: Delete a Node in the DLL--
     - Call `delete_Node` to delete a node from the list.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
   - If the user's choice is not within the valid range, display an error message.
11. --End--

Function Logic:

- --create_dll--:
   - Initialize `start` to `NULL` and `newNode` to a new node.
   - Prompt the user to enter numbers to create the list, ending with `-1`.

- For each number entered, allocate memory for a new node, set its `data` to the entered number, and adjust its `prev` and `next` pointers to insert it at the end of the list.
- Return the `start` pointer.

- --display_dll--:
- Initialize a pointer `ptr` to `start`.
- If `ptr` is `NULL`, display a message indicating the list is empty.
- Otherwise, traverse the list from `start` to the end, printing each node's `data`.

- --insert_Node--:
- Prompt the user to choose the method of insertion (beginning, end, after a specific value, or before a specific value).
- Allocate memory for a new node and set its `data` to the entered number.
- Based on the user's choice, adjust the `prev` and `next` pointers of the new node and the surrounding nodes to insert the new node at the chosen position.
- Return the `start` pointer.

- --delete_Node--:
- Prompt the user to choose the method of deletion (beginning, end, or a specific value).
- Based on the user's choice, find the node to be deleted and adjust the `prev` and `next` pointers of the surrounding nodes to remove the node from the list.
- Free the memory allocated for the deleted node.
- Return the `start` pointer.

Q7) ***C program to perform operation on circular singly linked list(L.L) using pointers:***

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

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

int main() {
    struct node *start = NULL;
    int option;

    printf("Namaskaram! This program demonstrates insertion and deletion in a circular linked list (SINGLE).\n");

    // Main menu loop
    do {
        printf("\n\n\t\t----------MAIN MENU----------"
               "\n 1:) Create a Circular Linked List"
               "\n 2:) Display the Circular Linked List"
               "\n 3:) Insert a node in the CLL"
               "\n 4:) Delete a node in the CLL"
               "\n 5:) Exit the program"
               "\n\n Enter your choice : ");

        if (scanf("%d", &option) != 1) {
            printf("Invalid input. Please enter a number.\n");
            while (getchar() != '\n'); // Clear the input buffer directly here
            continue;
        }
        if (option < 1 || option > 5) {
            printf("Invalid choice. Please try again.\n");
            continue;
        }

        switch(option) {
            case 1: {
                struct node *newNode, *ptr;
                int num;

                printf("\nEnter numbers to create a Circular Linked List (enter -1 to end):\n");
                while (scanf("%d", &num) == 1 && num != -1) {
```

```
43.                 newNode = (struct node *)malloc(sizeof(struct node));
44.                 if (newNode == NULL) {
45.                     printf("Memory allocation failed. Circular linked list creation
    aborted.\n");
46.                     break;
47.                 }
48.                 newNode->data = num;
49.                 if (start == NULL) {
50.                     start = newNode;
51.                     newNode->next = start; // Circular link
52.                 } else {
53.                     ptr = start;
54.                     while (ptr->next != start) {
55.                         ptr = ptr->next;
56.                     }
57.                     ptr->next = newNode;
58.                     newNode->next = start; // Circular link
59.                 }
60.                 printf("Enter data to create the Circular Linked List (enter -1 to end): ");
61.             }
62.             while (getchar() != '\n'); // Clear the input buffer directly here
63.             break;
64.         }
65.         case 2: {
66.             struct node *ptr = start;
67.             if (start == NULL) {
68.                 printf("\nList is empty.");
69.             } else {
70.                 printf("\nCircular Linked List: ");
71.                 do {
72.                     printf("%d -> ", ptr->data);
73.                     ptr = ptr->next;
74.                 } while (ptr != start);
75.                 printf("... (circular)\n");
76.             }
77.             break;
78.         }
79.         case 3: {
80.             struct node *newNode, *ptr;
81.             int num, val, choice;
82.
83.             printf("\nChoose method of insertion:"
84.                    "\n 1:) Insert at the beginning"
85.                    "\n 2:) Insert at the end"
86.                    "\n 3:) Insert after a specific value"
87.                    "\n 4:) Insert before a specific value"
```

```
88.                 "\n\nEnter your choice: ");
89.             if (scanf("%d", &choice) != 1 || choice < 1 || choice > 4) {
90.                 printf("Invalid input. Please enter a valid choice between 1 and 4.\n");
91.                 while (getchar() != '\n'); // Clear the input buffer directly here
92.                 continue;
93.             }
94.
95.             printf("\nEnter the data to be inserted: ");
96.             if (scanf("%d", &num) != 1) {
97.                 printf("Invalid input. Please enter a number.\n");
98.                 while (getchar() != '\n'); // Clear the input buffer directly here
99.                 continue;
100.                }
101.
102.                newNode = (struct node *)malloc(sizeof(struct node));
103.                if (newNode == NULL) {
104.                    printf("Memory allocation failed. Unable to insert node.\n");
105.                    break;
106.                }
107.                newNode->data = num;
108.
109.                if (start == NULL) {
110.                    start = newNode;
111.                    newNode->next = start;
112.                } else {
113.                    ptr = start;
114.                    // Insertion logic according to the chosen option
115.                    switch(choice) {
116.                        case 1: // Insert at the beginning
117.                            while (ptr->next != start) {
118.                                ptr = ptr->next;
119.                            }
120.                            ptr->next = newNode;
121.                            newNode->next = start;
122.                            start = newNode;
123.                            break;
124.                        case 2: // Insert at the end
125.                            while (ptr->next != start) {
126.                                ptr = ptr->next;
127.                            }
128.                            ptr->next = newNode;
129.                            newNode->next = start;
130.                            break;
131.                        case 3: // Insert after a specific value
132.                            printf("Enter the value after which you want to insert: ");
133.                            if (scanf("%d", &val) != 1) {
```

```
134.                         printf("Invalid input. Please enter a number.\n");
135.                         while (getchar() != '\n'); // Clear the input buffer directly
    here
136.                         free(newNode);
137.                         continue;
138.                     }
139.                     do {
140.                         if (ptr->data == val) {
141.                             newNode->next = ptr->next;
142.                             ptr->next = newNode;
143.                             break;
144.                         }
145.                         ptr = ptr->next;
146.                     } while (ptr != start);
147.                     break;
148.                 case 4: // Insert before a specific value
149.                     printf("Enter the value before which you want to insert: ");
150.                     if (scanf("%d", &val) != 1) {
151.                         printf("Invalid input. Please enter a number.\n");
152.                         while (getchar() != '\n'); // Clear the input buffer directly
    here
153.                         free(newNode);
154.                         continue;
155.                     }
156.                     if (start->data == val) {
157.                         // Insert before the first node
158.                         newNode->next = start;
159.                         while (ptr->next != start) {
160.                             ptr = ptr->next;
161.                         }
162.                         ptr->next = newNode;
163.                         start = newNode;
164.                     } else {
165.                         while (ptr->next != start && ptr->next->data != val) {
166.                             ptr = ptr->next;
167.                         }
168.                         if (ptr->next == start) {
169.                             printf("Value not found in the list.\n");
170.                             free(newNode);
171.                         } else {
172.                             newNode->next = ptr->next;
173.                             ptr->next = newNode;
174.                         }
175.                     }
176.                     break;
177.             }
```

```
178.                 }
179.                 break;
180.             }
181.             case 4: {
182.                 // Delete a node from the circular linked list
183.                 struct node *ptr, *prev;
184.                 int choice, val;
185.
186.                 if (start == NULL) {
187.                     printf("\nList is empty. Nothing to delete.");
188.                     break;
189.                 }
190.
191.                 printf("\nChoose method of deletion:"
192.                     "\n 1:) Delete at the beginning"
193.                     "\n 2:) Delete at the end"
194.                     "\n 3:) Delete a specific value"
195.                     "\n\nEnter your choice: ");
196.                 if (scanf("%d", &choice) != 1 || choice < 1 || choice > 3) {
197.                     printf("Invalid input. Please enter a valid choice between 1 and
    3.\n");
198.                     while (getchar() != '\n'); // Clear the input buffer directly here
199.                     continue;
200.                 }
201.
202.                 switch(choice) {
203.                     case 1: // Delete at the beginning
204.                         if (start->next == start) {
205.                             free(start);
206.                             start = NULL;
207.                         } else {
208.                             ptr = start;
209.                             while (ptr->next != start) {
210.                                 ptr = ptr->next;
211.                             }
212.                             ptr->next = start->next;
213.                             free(start);
214.                             start = ptr->next;
215.                         }
216.                         break;
217.                     case 2: // Delete at the end
218.                         if (start->next == start) {
219.                             free(start);
220.                             start = NULL;
221.                         } else {
222.                             ptr = start;
```

```
223.                     while (ptr->next->next != start) {
224.                         ptr = ptr->next;
225.                     }
226.                     free(ptr->next);
227.                     ptr->next = start;
228.                 }
229.                 break;
230.             case 3: // Delete a specific value
231.                 printf("Enter the value to be deleted: ");
232.                 if (scanf("%d", &val) != 1) {
233.                     printf("Invalid input. Please enter a number.\n");
234.                     while (getchar() != '\n'); // Clear the input buffer directly here
235.                     continue;
236.                 }
237.                 ptr = start;
238.                 prev = NULL;
239.                 do {
240.                     if (ptr->data == val) {
241.                         if (prev == NULL) { // Deleting the head
242.                             if (ptr->next == start) { // Only one node
243.                                 free(ptr);
244.                                 start = NULL;
245.                             } else { // More than one node, head needs to be deleted
246.                                 prev = start;
247.                                 while (prev->next != start) {
248.                                     prev = prev->next;
249.                                 }
250.                                 prev->next = start->next;
251.                                 free(start);
252.                                 start = prev->next;
253.                             }
254.                         } else { // Deleting other than head
255.                             prev->next = ptr->next;
256.                             free(ptr);
257.                         }
258.                         break;
259.                     }
260.                     prev = ptr;
261.                     ptr = ptr->next;
262.                 } while (ptr != start);
263.                 if (ptr == start) {
264.                     printf("Value not found in the list.\n");
265.                 }
266.                 break;
267.         }
268.         break;
```

```
                }
                case 5:
                    printf("\nExiting...");
                    break;
                default:
                    printf("\nInvalid option. Please try again.");
            }
        } while(option != 5);

        return 0;
    }
```

Q8) ***C program to perform operation on circular singly linked list(L.L) using functions-pointers:***

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

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

// Function prototypes
struct node *create_cll();
void display_cll(struct node *);
struct node *insert_Node(struct node *);
struct node *delete_Node(struct node *);

int main() {
    struct node *start = NULL;
    int option;

    printf("NAMASKARAM! This program demonstrates insertion and deletion
in a circular linked list (SINGLE).\n");

    // Main menu loop
    do {
        printf("\n\n\t\t----------MAIN MENU----------\n");
        printf(" 1:) Create a Circular Linked List\n");
        printf(" 2:) Display the Circular Linked List\n");
        printf(" 3:) Insert a node in the CLL\n");
        printf(" 4:) Delete a node in the CLL\n");
        printf(" 5:) Exit the program\n");
        printf("\n Enter your choice : ");

```

```
32.         if (scanf("%d", &option) != 1) {
33.             printf("Invalid input. Please enter a number.\n");
34.             while (getchar() != '\n'); // clear the input buffer
35.             continue;
36.         }
37.
38.         if (option < 1 || option > 5) {
39.             printf("Invalid choice. Please try again.\n");
40.             continue;
41.         }
42.
43.         switch(option) {
44.             case 1:
45.                 start = create_cll();
46.                 break;
47.             case 2:
48.                 display_cll(start);
49.                 break;
50.             case 3:
51.                 start = insert_Node(start);
52.                 break;
53.             case 4:
54.                 start = delete_Node(start);
55.                 break;
56.             case 5:
57.                 printf("\nExiting...\n");
58.                 break;
59.             default:
60.                 printf("\nInvalid option. Please try again.\n");
61.         }
62.     } while(option != 5);
63.
64.     return 0;
65. }
66.
67. struct node *create_cll() {
68.     struct node *start = NULL, *newNode, *ptr;
69.     int num;
70.
71.     printf("\nEnter numbers to create a Circular Linked List (enter -1 to
    end):\n");
72.     while (scanf("%d", &num) == 1 && num != -1) {
73.         newNode = (struct node *)malloc(sizeof(struct node));
74.         if (newNode == NULL) {
75.             printf("Memory allocation failed. Circular linked list creation
    aborted.\n");
```

```
76.             return start;
77.         }
78.
79.         newNode->data = num;
80.         if (start == NULL) {
81.             start = newNode;
82.             newNode->next = start;
83.         } else {
84.             ptr = start;
85.             while (ptr->next != start) {
86.                 ptr = ptr->next;
87.             }
88.             ptr->next = newNode;
89.             newNode->next = start;
90.         }
91.
92.         printf("Enter data to create the Circular Linked List (enter -1 to end): ");
93.     }
94.     while (getchar() != '\n'); // Clear the input buffer
95.     return start;
96. }
97.
98. void display_cll(struct node *start) {
99.     if (start == NULL) {
100.             printf("\nList is empty.\n");
101.             return;
102.         }
103.         struct node *ptr = start;
104.         printf("\nCircular Linked List: ");
105.         do {
106.             printf("%d -> ", ptr->data);
107.             ptr = ptr->next;
108.         } while (ptr != start);
109.         printf("... (circular)\n");
110.     }
111.
112.     struct node *insert_Node(struct node *start) {
113.         struct node *newNode, *ptr;
114.         int num, val, choice;
115.
116.         printf("\nChoose method of insertion:\n");
117.         printf(" 1:) Insert at the beginning\n");
118.         printf(" 2:) Insert at the end\n");
119.         printf(" 3:) Insert after a specific value\n");
120.         printf(" 4:) Insert before a specific value\n");
121.         printf("\nEnter your choice: ");
```

```
122.
123.        if (scanf("%d", &choice) != 1 || choice < 1 || choice > 4) {
124.            printf("Invalid choice. Please enter a number between 1 and 4.\n");
125.            while (getchar() != '\n'); // Clear the input buffer
126.            return start;
127.        }
128.
129.        printf("\nEnter the data to be inserted: ");
130.        if (scanf("%d", &num) != 1) {
131.            printf("Invalid input. Please enter a number.\n");
132.            while (getchar() != '\n'); // Clear the input buffer
133.            return start;
134.        }
135.
136.        newNode = (struct node *)malloc(sizeof(struct node));
137.        if (newNode == NULL) {
138.            printf("Memory allocation failed. Unable to insert node.\n");
139.            return start;
140.        }
141.        newNode->data = num;
142.
143.        if (start == NULL) {
144.            start = newNode;
145.            newNode->next = start;
146.            return start;
147.        }
148.
149.        ptr = start;
150.        switch(choice) {
151.            case 1: // Insert at the beginning
152.                while (ptr->next != start) ptr = ptr->next;
153.                ptr->next = newNode;
154.                newNode->next = start;
155.                start = newNode;
156.                break;
157.            case 2: // Insert at the end
158.                while (ptr->next != start) ptr = ptr->next;
159.                ptr->next = newNode;
160.                newNode->next = start;
161.                break;
162.            case 3: // Insert after a specific value
163.                printf("Enter the value after which you want to insert: ");
164.                if (scanf("%d", &val) != 1) {
165.                    printf("Invalid input. Please enter a number.\n");
166.                    while (getchar() != '\n'); // Clear the input buffer
167.                    free(newNode);
```

```
168.                 return start;
169.             }
170.             do {
171.                 if (ptr->data == val) {
172.                     newNode->next = ptr->next;
173.                     ptr->next = newNode;
174.                     break;
175.                 }
176.                 ptr = ptr->next;
177.             } while (ptr != start);
178.             break;
179.         case 4: // Insert before a specific value
180.             if (start->data == val) { // If the node is to be inserted before the start
    node
181.                 newNode->next = start;
182.                 while (ptr->next != start) ptr = ptr->next;
183.                 ptr->next = newNode;
184.                 start = newNode;
185.             } else {
186.                 while (ptr->next != start && ptr->next->data != val) ptr = ptr-
    >next;
187.                 if (ptr->next == start) {
188.                     printf("Value not found in the list.\n");
189.                     free(newNode);
190.                 } else {
191.                     newNode->next = ptr->next;
192.                     ptr->next = newNode;
193.                 }
194.             }
195.             break;
196.         default:
197.             free(newNode);
198.             printf("\nInvalid choice.\n");
199.     }
200.
201.     return start;
202. }
203.
204. struct node *delete_Node(struct node *start) {
205.     if (start == NULL) {
206.         printf("\nList is empty. Nothing to delete.\n");
207.         return start;
208.     }
209.
210.     struct node *ptr, *prev;
211.     int choice, val;
```

```
212.
213.        printf("\nChoose method of deletion:\n");
214.        printf(" 1:) Delete at the beginning\n");
215.        printf(" 2:) Delete at the end\n");
216.        printf(" 3:) Delete a specific value\n");
217.        printf("\nEnter your choice: ");
218.
219.        if (scanf("%d", &choice) != 1 || choice < 1 || choice > 3) {
220.            printf("Invalid choice. Please enter a number between 1 and 3.\n");
221.            while (getchar() != '\n'); // Clear the input buffer
222.            return start;
223.        }
224.
225.        switch(choice) {
226.            case 1: // Delete at the beginning
227.                ptr = start;
228.                while (ptr->next != start) ptr = ptr->next;
229.                if (ptr == start) {
230.                    free(start);
231.                    start = NULL;
232.                } else {
233.                    ptr->next = start->next;
234.                    free(start);
235.                    start = ptr->next;
236.                }
237.                break;
238.            case 2: // Delete at the end
239.                ptr = start;
240.                if (ptr->next == start) {
241.                    free(start);
242.                    start = NULL;
243.                } else {
244.                    while (ptr->next->next != start) ptr = ptr->next;
245.                    free(ptr->next);
246.                    ptr->next = start;
247.                }
248.                break;
249.            case 3: // Delete a specific value
250.                printf("Enter the value to be deleted: ");
251.                if (scanf("%d", &val) != 1) {
252.                    printf("Invalid input. Please enter a number.\n");
253.                    while (getchar() != '\n'); // Clear the input buffer
254.                    return start;
255.                }
256.                ptr = start;
257.                prev = NULL;
```

```
258.            do {
259.                if (ptr->data == val) {
260.                    if (prev == NULL) { // Deleting the head
261.                        if (ptr->next == start) { // Only one node in list
262.                            free(ptr);
263.                            start = NULL;
264.                        } else { // More than one node, head is to be deleted
265.                            while (ptr->next != start) ptr = ptr->next;
266.                            ptr->next = start->next;
267.                            free(start);
268.                            start = ptr->next;
269.                        }
270.                    } else { // Deleting other than head
271.                        prev->next = ptr->next;
272.                        free(ptr);
273.                    }
274.                    break;
275.                }
276.                prev = ptr;
277.                ptr = ptr->next;
278.            } while (ptr != start);
279.            if (ptr == start) {
280.                printf("Value not found in the list.\n");
281.            }
282.            break;
283.        default:
284.            printf("\nInvalid choice.\n");
285.    }
286.
287.    return start;
288. }
```

ALGORITHMS FOR question 7 and 8

1] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for the linked list node, containing an integer `data` and a pointer `next` to the next node.
4. In the `main` function, initialize a pointer `start` to `NULL` to represent the start of the circular linked list.
5. Declare an integer variable `option` to store the user's menu choice.
6. Display a welcome message and the main menu options to the user.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a Circular Linked List--
     - Prompt the user to enter numbers to create the list, ending with `-1`.
     - For each number entered, dynamically allocate memory for a new node, set its `data` to the entered number, and insert it into the circular linked list.
   - --Case 2: Display the Circular Linked List--
     - Traverse the circular linked list starting from `start` and print each node's `data` until reaching the start node again.
   - --Case 3: Insert a Node in the Circular Linked List--
     - Prompt the user to choose the method of insertion (beginning, end, after a specific value, or before a specific value).
     - Dynamically allocate memory for a new node, set its `data` to the entered number, and insert it into the circular linked list according to the chosen method.
   - --Case 4: Delete a Node from the Circular Linked List--
     - Prompt the user to choose the method of deletion (beginning, end, or a specific value).
     - Delete the node from the circular linked list according to the chosen method.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
   - If the user's choice is not within the valid range, display an error message.
9. --End--

2] fp

1. --Start--

2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for the circular linked list node, containing an integer `data` and a pointer `next` to the next node.
4. Declare function prototypes for creating a circular linked list (`create_cll`), displaying the list (`display_cll`), inserting a node (`insert_Node`), and deleting a node (`delete_Node`).
5. In the `main` function, initialize a `node` pointer `start` to `NULL` and declare an integer `option` to store the user's menu choice.
6. Display a welcome message to the user.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a Circular Linked List--
      - Call `create_cll` to create the list and assign the returned pointer to `start`.
   - --Case 2: Display the Circular Linked List--
      - Call `display_cll` with `start` as the argument to display the list.
   - --Case 3: Insert a Node in the CLL--
      - Call `insert_Node` with `start` as the argument to insert a node. The returned pointer is assigned back to `start`.
   - --Case 4: Delete a Node in the CLL--
      - Call `delete_Node` with `start` as the argument to delete a node. The returned pointer is assigned back to `start`.
   - --Case 5: Exit the Program--
      - Display a message indicating that the program is exiting.
   - If the user's choice is not within the valid range, display an error message.
9. --End--

// Function Logic:

- --create_cll--:
   - Initialize `start` to `NULL` and declare a `newNode` pointer and an integer `num`.
   - Prompt the user to enter numbers to create the list, ending with `-1`.
   - For each number entered, allocate memory for a new node, assign the number to the node's `data` field, and insert the node into the circular linked list.
   - If the list is empty, make the new node point to itself. Otherwise, insert the new node at the end of the list.
   - Return the `start` pointer.

- --display_cll--:
   - Check if `start` is `NULL`. If so, print that the list is empty and return.
   - Initialize a `ptr` pointer to `start`.
   - Traverse the list by following the `next` pointers until reaching the `start` node again, printing each node's `data` field.

- Print a message indicating the list is circular.

- --insert_Node--:
  - Prompt the user to choose the method of insertion and enter the data to be inserted.
  - Allocate memory for a new node and assign the entered data to the node's `data` field.
  - Depending on the user's choice, insert the new node at the beginning, at the end, after a specific value, or before a specific value in the circular linked list.
  - Return the `start` pointer.

- --delete_Node--:
  - Check if `start` is `NULL`. If so, print that the list is empty and return.
  - Prompt the user to choose the method of deletion.
  - Depending on the user's choice, delete the node at the beginning, at the end, or with a specific value from the circular linked list.
  - Return the `start` pointer.

Q9) ***C program to perform operation on circular doubly linked list(L.L) using pointers functions:***

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  // Define the structure for a node
5.  struct node {
6.      int data;
7.      struct node *prev;
8.      struct node *next;
9.  };
10.
11. // Function prototypes for linked list operations
12. struct node *create_cll();
13. void display_cll(struct node *);
14. struct node *insert_Node(struct node *);
15. struct node *delete_Node(struct node *);
16.
17. int main() {
18.     struct node *start = NULL;
19.     int option;
20.
21.     printf("NAMASKARAM! This program demonstrates insertion and deletion in a
    circular doubly linked list.\n");
22.
23.     // Main menu loop
24.     do {
25.         printf("\n\n\t\t----------MAIN MENU----------\n");
26.         printf(" 1:) Create a Circular Doubly Linked List\n");
27.         printf(" 2:) Display the Circular Doubly Linked List\n");
28.         printf(" 3:) Insert a node in the CDLL\n");
29.         printf(" 4:) Delete a node in the CDLL\n");
30.         printf(" 5:) Exit the program\n");
31.         printf("\n Enter your choice : ");
32.
33.         if (scanf("%d", &option) != 1) {
34.             printf("Invalid input. Please enter a number.\n");
35.             while (getchar() != '\n'); // clear the input buffer
36.             continue;
37.         }
38.         if (option < 1 || option > 5) {
39.             printf("Invalid choice. Please try again.\n");
40.             continue;
41.         }
```

```
42.
43.         switch(option) {
44.             case 1:
45.                 start = create_cll();
46.                 break;
47.             case 2:
48.                 display_cll(start);
49.                 break;
50.             case 3:
51.                 start = insert_Node(start);
52.                 break;
53.             case 4:
54.                 start = delete_Node(start);
55.                 break;
56.             case 5:
57.                 printf("\nExiting...\n");
58.                 break;
59.             default:
60.                 printf("\nInvalid option. Please try again.\n");
61.         }
62.     } while(option != 5);
63.
64.     return 0;
65. }
66.
67. struct node *create_cll() {
68.     struct node *start = NULL, *newNode, *ptr;
69.     int num;
70.
71.     printf("\nEnter numbers to create a Circular Doubly Linked List (enter -1 to
    end):\n");
72.     while (scanf("%d", &num) == 1 && num != -1) {
73.         newNode = (struct node *)malloc(sizeof(struct node));
74.         if (newNode == NULL) {
75.             printf("Memory allocation failed. Circular doubly linked list creation
    aborted.\n");
76.             return start;
77.         }
78.
79.         newNode->data = num;
80.         if (start == NULL) {
81.             start = newNode;
82.             newNode->prev = newNode;
83.             newNode->next = newNode;
84.         } else {
85.             ptr = start->prev;
```

```
86.             newNode->next = start;
87.             newNode->prev = ptr;
88.             ptr->next = newNode;
89.             start->prev = newNode;
90.         }
91.
92.         printf("Enter data to create the Circular Doubly Linked List (enter -1 to end): ");
93.     }
94.     while (getchar() != '\n'); // Clear the input buffer
95.     return start;
96. }
97.
98. void display_cll(struct node *start) {
99.     if (start == NULL) {
100.                 printf("\nList is empty.\n");
101.                 return;
102.             }
103.             struct node *ptr = start;
104.             printf("\nCircular Doubly Linked List: ");
105.             do {
106.                 printf("%d <-> ", ptr->data);
107.                 ptr = ptr->next;
108.             } while (ptr != start);
109.             printf("(head)\n");
110.         }
111.
112.         struct node *insert_Node(struct node *start) {
113.             struct node *newNode, *ptr;
114.             int num, val, choice;
115.
116.             printf("\nChoose method of insertion:\n");
117.             printf(" 1:) Insert at the beginning\n");
118.             printf(" 2:) Insert at the end\n");
119.             printf(" 3:) Insert after a specific value\n");
120.             printf(" 4:) Insert before a specific value\n");
121.             printf("\nEnter your choice: ");
122.
123.             if (scanf("%d", &choice) != 1 || choice < 1 || choice > 4) {
124.                 printf("Invalid input. Please enter a valid choice between 1 and 4.\n");
125.                 while (getchar() != '\n'); // Clear the input buffer
126.                 return start;
127.             }
128.
129.             printf("\nEnter the data to be inserted: ");
130.             if (scanf("%d", &num) != 1) {
131.                 printf("Invalid input. Please enter a number.\n");
```

```
132.             while (getchar() != '\n'); // Clear the input buffer
133.             return start;
134.         }
135.
136.         newNode = (struct node *)malloc(sizeof(struct node));
137.         if (newNode == NULL) {
138.             printf("Memory allocation failed. Unable to insert node.\n");
139.             return start;
140.         }
141.         newNode->data = num;
142.
143.         if (start == NULL) {
144.             start = newNode;
145.             newNode->next = newNode;
146.             newNode->prev = newNode;
147.         } else {
148.             switch(choice) {
149.                 case 1: // Insert at the beginning
150.                     newNode->next = start;
151.                     newNode->prev = start->prev;
152.                     start->prev->next = newNode;
153.                     start->prev = newNode;
154.                     start = newNode;
155.                     break;
156.                 case 2: // Insert at the end
157.                     newNode->next = start;
158.                     newNode->prev = start->prev;
159.                     start->prev->next = newNode;
160.                     start->prev = newNode;
161.                     break;
162.                 case 3: // Insert after a specific value
163.                     printf("Enter the value after which you want to insert: ");
164.                     if (scanf("%d", &val) != 1) {
165.                         printf("Invalid input. Please enter a number.\n");
166.                         while (getchar() != '\n'); // Clear the input buffer
167.                         free(newNode);
168.                         return start;
169.                     }
170.                     ptr = start;
171.                     do {
172.                         if (ptr->data == val) {
173.                             newNode->next = ptr->next;
174.                             newNode->prev = ptr;
175.                             ptr->next->prev = newNode;
176.                             ptr->next = newNode;
177.                             break;
```

```
                    }
                    ptr = ptr->next;
                } while (ptr != start);
                if (ptr == start) {
                    printf("Value not found in the list.\n");
                    free(newNode);
                }
                break;
            case 4: // Insert before a specific value
                printf("Enter the value before which you want to insert: ");
                if (scanf("%d", &val) != 1) {
                    printf("Invalid input. Please enter a number.\n");
                    while (getchar() != '\n'); // Clear the input buffer
                    free(newNode);
                    return start;
                }
                ptr = start;
                do {
                    if (ptr->data == val) {
                        newNode->next = ptr;
                        newNode->prev = ptr->prev;
                        ptr->prev->next = newNode;
                        ptr->prev = newNode;
                        if (ptr == start) {
                            start = newNode;
                        }
                        break;
                    }
                    ptr = ptr->next;
                } while (ptr != start);
                if (ptr == start) {
                    printf("Value not found in the list.\n");
                    free(newNode);
                }
                break;
        }
    }
    return start;
}

struct node *delete_Node(struct node *start) {
    if (start == NULL) {
        printf("\nList is empty. Nothing to delete.\n");
        return start;
    }

```

```
224.        struct node *ptr;
225.        int choice, val;
226.
227.        printf("\nChoose method of deletion:\n");
228.        printf(" 1:) Delete at the beginning\n");
229.        printf(" 2:) Delete at the end\n");
230.        printf(" 3:) Delete a specific value\n");
231.        printf("\nEnter your choice: ");
232.
233.        if (scanf("%d", &choice) != 1 || choice < 1 || choice > 3) {
234.           printf("Invalid choice. Please enter a number between 1 and 3.\n");
235.           while (getchar() != '\n'); // Clear the input buffer
236.           return start;
237.        }
238.
239.        switch(choice) {
240.           case 1: // Delete at the beginning
241.              if (start->next == start) {
242.                 free(start);
243.                 start = NULL;
244.              } else {
245.                 ptr = start;
246.                 start->next->prev = start->prev;
247.                 start->prev->next = start->next;
248.                 start = start->next;
249.                 free(ptr);
250.              }
251.              break;
252.           case 2: // Delete at the end
253.              ptr = start->prev;
254.              if (ptr == start) {
255.                 free(start);
256.                 start = NULL;
257.              } else {
258.                 ptr->prev->next = start;
259.                 start->prev = ptr->prev;
260.                 free(ptr);
261.              }
262.              break;
263.           case 3: // Delete a specific value
264.              printf("Enter the value to be deleted: ");
265.              if (scanf("%d", &val) != 1) {
266.                 printf("Invalid input. Please enter a number.\n");
267.                 while (getchar() != '\n'); // Clear the input buffer
268.                 return start;
269.              }
```

```
                ptr = start;
                do {
                    if (ptr->data == val) {
                        if (ptr->next == ptr) { // Only one node
                            free(ptr);
                            start = NULL;
                        } else {
                            ptr->prev->next = ptr->next;
                            ptr->next->prev = ptr->prev;
                            if (ptr == start) {
                                start = ptr->next;
                            }
                            free(ptr);
                        }
                        break;
                    }
                    ptr = ptr->next;
                } while (ptr != start);
                if (ptr == start) {
                    printf("Value not found in the list.\n");
                }
                break;
            default:
                printf("\nInvalid choice.\n");
        }

        return start;
    }
```

Q10) ***C program to perform operation on circular doubly linked list(L.L) using pointers:***

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

// Define the structure for a node
struct node {
    int data;
    struct node *prev;
    struct node *next;
};

int main() {
    struct node *start = NULL;
    struct node *newNode, *ptr;
```

```
14.    int option, num, val, choice;
15.
16.    printf("NAMASKARAM! This program demonstrates insertion and deletion in a
    circular doubly linked list.\n");
17.
18.    // Main menu loop
19.    do {
20.       printf("\n\n\t\t----------MAIN MENU----------"
21.       "\n 1:) Create a Circular Doubly Linked List"
22.       "\n 2:) Display the Circular Doubly Linked List"
23.       "\n 3:) Insert a node in the CDLL"
24.       "\n 4:) Delete a node in the CDLL"
25.       "\n 5:) Exit the program"
26.       "\n\n Enter your choice : ");
27.       if (scanf("%d", &option) != 1) {
28.          printf("Invalid input. Please enter a number.\n");
29.          while (getchar() != '\n'); // clear the input buffer
30.          continue;
31.       }
32.
33.       if (option < 1 || option > 5) {
34.          printf("Invalid choice. Please try again.\n");
35.          continue;
36.       }
37.
38.       switch(option) {
39.          case 1:
40.             printf("\nEnter numbers to create a Circular Doubly Linked List (enter -1
    to end):\n");
41.             while (scanf("%d", &num) == 1 && num != -1) {
42.                newNode = (struct node *)malloc(sizeof(struct node));
43.                if (newNode == NULL) {
44.                   printf("Memory allocation failed. Circular doubly linked list creation
    aborted.\n");
45.                   break;
46.                }
47.                newNode->data = num;
48.                if (start == NULL) {
49.                   start = newNode;
50.                   newNode->next = newNode;
51.                   newNode->prev = newNode;
52.                } else {
53.                   newNode->next = start;
54.                   newNode->prev = start->prev;
55.                   start->prev->next = newNode;
56.                   start->prev = newNode;
```

```
57.                 }
58.                 printf("Enter data to create the Circular Doubly Linked List (enter -1
    to end): ");
59.             }
60.             while (getchar() != '\n'); // clear the input buffer
61.             break;
62.         case 2:
63.             if (start == NULL) {
64.                 printf("\nList is empty.\n");
65.             } else {
66.                 printf("\nCircular Doubly Linked List: ");
67.                 ptr = start;
68.                 do {
69.                     printf("%d <-> ", ptr->data);
70.                     ptr = ptr->next;
71.                 } while (ptr != start);
72.                 printf("(head)\n");
73.             }
74.             break;
75.         case 3:
76.             printf("\nChoose method of insertion:"
77.             "\n 1:) Insert at the beginning"
78.             "\n 2:) Insert at the end"
79.             "\n 3:) Insert after a specific value"
80.             "\n 4:) Insert before a specific value"
81.             "\n\nEnter your choice: ");
82.             if (scanf("%d", &choice) != 1) {
83.                 printf("Invalid input. Please enter a number.\n");
84.                 while (getchar() != '\n');
85.                 continue;
86.             }
87.             if (choice < 1 || choice > 4) {
88.                 printf("Invalid choice. Please try again.\n");
89.                 continue;
90.             }
91.             printf("Enter the data to be inserted: ");
92.             if (scanf("%d", &num) != 1) {
93.                 printf("Invalid input. Please enter a number.\n");
94.                 while (getchar() != '\n');
95.                 continue;
96.             }
97.             newNode = (struct node *)malloc(sizeof(struct node));
98.             if (newNode == NULL) {
99.                 printf("Memory allocation failed. Unable to insert node.\n");
100.                     break;
101.                 }
```

```
102.                 newNode->data = num;
103.                 if (start == NULL) {
104.                     start = newNode;
105.                     newNode->next = newNode;
106.                     newNode->prev = newNode;
107.                 } else {
108.                     ptr = start;
109.                     switch (choice) {
110.                         case 1: // Insert at the beginning
111.                             newNode->next = start;
112.                             newNode->prev = start->prev;
113.                             start->prev->next = newNode;
114.                             start->prev = newNode;
115.                             start = newNode;
116.                             break;
117.                         case 2: // Insert at the end
118.                             newNode->next = start;
119.                             newNode->prev = start->prev;
120.                             start->prev->next = newNode;
121.                             start->prev = newNode;
122.                             break;
123.                         case 3: // Insert after a specific value
124.                             printf("Enter the value after which you want to insert: ");
125.                             if (scanf("%d", &val) != 1) {
126.                                 printf("Invalid input. Please enter a number.\n");
127.                                 while (getchar() != '\n');
128.                                 free(newNode);
129.                                 continue;
130.                             }
131.                             do {
132.                                 if (ptr->data == val) {
133.                                     newNode->next = ptr->next;
134.                                     newNode->prev = ptr;
135.                                     ptr->next->prev = newNode;
136.                                     ptr->next = newNode;
137.                                     break;
138.                                 }
139.                                 ptr = ptr->next;
140.                             } while (ptr != start);
141.                             if (ptr == start) {
142.                                 printf("Value not found in the list.\n");
143.                                 free(newNode);
144.                             }
145.                             break;
146.                         case 4: // Insert before a specific value
147.                             printf("Enter the value before which you want to insert: ");
```

```
148.                     if (scanf("%d", &val) != 1) {
149.                         printf("Invalid input. Please enter a number.\n");
150.                         while (getchar() != '\n');
151.                         free(newNode);
152.                         continue;
153.                     }
154.                     do {
155.                         if (ptr->data == val) {
156.                             newNode->next = ptr;
157.                             newNode->prev = ptr->prev;
158.                             ptr->prev->next = newNode;
159.                             ptr->prev = newNode;
160.                             if (ptr == start) {
161.                                 start = newNode;
162.                             }
163.                             break;
164.                         }
165.                         ptr = ptr->next;
166.                     } while (ptr != start);
167.                     if (ptr == start) {
168.                         printf("Value not found in the list.\n");
169.                         free(newNode);
170.                     }
171.                     break;
172.                 }
173.             }
174.             break;
175.         case 4:
176.             if (start == NULL) {
177.                 printf("\nList is empty. Nothing to delete.\n");
178.                 break;
179.             }
180.             printf("\nChoose method of deletion:"
181.             "\n 1:) Delete at the beginning"
182.             "\n 2:) Delete at the end"
183.             "\n 3:) Delete a specific value"
184.             "\n\nEnter your choice: ");
185.             if (scanf("%d", &choice) != 1) {
186.                 printf("Invalid input. Please enter a number.\n");
187.                 while (getchar() != '\n');
188.                 continue;
189.             }
190.             switch (choice) {
191.                 case 1:
192.                     ptr = start;
193.                     if (start->next == start) {
```

```
194.                     free(start);
195.                     start = NULL;
196.                 } else {
197.                     start = start->next;
198.                     start->prev = ptr->prev;
199.                     ptr->prev->next = start;
200.                     free(ptr);
201.                 }
202.                 break;
203.             case 2:
204.                 ptr = start->prev;
205.                 if (start->next == start) {
206.                     free(start);
207.                     start = NULL;
208.                 } else {
209.                     start->prev = ptr->prev;
210.                     ptr->prev->next = start;
211.                     free(ptr);
212.                 }
213.                 break;
214.             case 3:
215.                 printf("Enter the value to be deleted: ");
216.                 if (scanf("%d", &val) != 1) {
217.                     printf("Invalid input. Please enter a number.\n");
218.                     while (getchar() != '\n');
219.                     continue;
220.                 }
221.                 ptr = start;
222.                 do {
223.                     if (ptr->data == val) {
224.                         if (ptr == start) {
225.                             if (start->next == start) {
226.                                 free(start);
227.                                 start = NULL;
228.                             } else {
229.                                 start = ptr->next;
230.                                 start->prev = ptr->prev;
231.                                 ptr->prev->next = start;
232.                                 free(ptr);
233.                             }
234.                         } else {
235.                             ptr->prev->next = ptr->next;
236.                             ptr->next->prev = ptr->prev;
237.                             free(ptr);
238.                         }
239.                         break;
```

```
                    }
                    ptr = ptr->next;
                } while (ptr != start);
                if (ptr == start) {
                    printf("Value not found in the list.\n");
                }
                break;
            default:
                printf("\nInvalid choice.\n");
        }
        break;
    case 5:
        printf("\nExiting...");
        break;
    default:
        printf("\nInvalid option. Please try again.");
    }
} while(option != 5);

return 0;
}
```

ALGORITHM FOR Question 9and 10

1] fp

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for the nodes of the circular doubly linked list, containing an integer `data` and pointers to the previous and next nodes.
4. Declare function prototypes for creating a circular doubly linked list (`create_cll`), displaying the list (`display_cll`), inserting a node (`insert_Node`), and deleting a node (`delete_Node`).
5. In the `main` function, initialize a pointer `start` to `NULL` to represent the start of the list.
6. Display a welcome message to the user.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a Circular Doubly Linked List--
     - Call `create_cll` to create the list and assign the returned pointer to `start`.
   - --Case 2: Display the Circular Doubly Linked List--
     - Call `display_cll` with `start` as the argument to display the list.
   - --Case 3: Insert a Node in the CDLL--
     - Call `insert_Node` with `start` as the argument to insert a node and update `start` with the returned pointer.
   - --Case 4: Delete a Node in the CDLL--
     - Call `delete_Node` with `start` as the argument to delete a node and update `start` with the returned pointer.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
   - If the user's choice is not within the valid range, display an error message.
9. --End--

Function Logic:

- --create_cll--:
   - Initialize `start` to `NULL` and `newNode` to `NULL`.
   - Prompt the user to enter numbers to create the list, ending with `-1`.
   - For each number entered, allocate memory for a new node, set its `data` to the entered number, and insert it into the list.
   - If the list is empty, make the new node point to itself for both `next` and `prev`.
   - If the list is not empty, insert the new node at the end of the list, updating the `next` and `prev` pointers accordingly.
   - Return the `start` pointer.

- --display_cll--:
  - Check if `start` is `NULL`. If so, print that the list is empty and return.
  - Initialize a pointer `ptr` to `start`.
  - Traverse the list, printing each node's `data` until `ptr` reaches `start` again, indicating the end of the list.

- --insert_Node--:
  - Prompt the user to choose the method of insertion (beginning, end, after a specific value, or before a specific value).
  - Allocate memory for a new node and set its `data` to the entered number.
  - Based on the user's choice, insert the new node at the beginning, end, after a specific value, or before a specific value, updating the `next` and `prev` pointers accordingly.
  - Return the updated `start` pointer.

- --delete_Node--:
  - Check if `start` is `NULL`. If so, print that the list is empty and return.
  - Prompt the user to choose the method of deletion (beginning, end, or a specific value).
  - Based on the user's choice, delete the node at the beginning, end, or a specific value, updating the `next` and `prev` pointers accordingly.
  - Return the updated `start` pointer.

2] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `node` for the doubly linked list node, containing an integer `data`, and pointers to the previous and next nodes.
4. In the `main` function, initialize a pointer `start` to `NULL` to represent the start of the circular doubly linked list.
5. Declare pointers `newNode` and `ptr` for creating new nodes and traversing the list.
6. Declare integer variables `option`, `num`, `val`, and `choice` to store the user's menu choice, the number to be inserted or deleted, and the choice for insertion or deletion method.
7. Display a welcome message to the user.
8. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
9. Use a `switch` statement to handle the user's choice:
   - --Case 1: Create a Circular Doubly Linked List--
     - Prompt the user to enter numbers to create the list, ending with `-1`.

- For each number entered, allocate memory for a new node, set its `data` to the entered number, and insert it into the list.
- --Case 2: Display the Circular Doubly Linked List--
- If the list is empty, display a message indicating that the list is empty.
- Otherwise, traverse the list from the start node to the end, printing each node's data.
- --Case 3: Insert a Node in the CDLL--
- Prompt the user to choose the method of insertion and enter the data to be inserted.
- Based on the choice, insert the new node at the beginning, end, after a specific value, or before a specific value.
- --Case 4: Delete a Node in the CDLL--
- Prompt the user to choose the method of deletion.
- Based on the choice, delete the node at the beginning, end, or a specific value.
- --Case 5: Exit the Program--
- Display a message indicating that the program is exiting.
- If the user's choice is not within the valid range, display an error message.

10. --End--

Q-11) C program for searching in an array [without using functions and pointers]

```
#include <stdio.h>

int main() {
    int size, key, choice, found = 0, index = -1;

    printf("Enter the size of the array: ");
    if (scanf("%d", &size) != 1 || size <= 0) {
        printf("Invalid size. Please enter a positive integer.\n");
        while (getchar() != '\n');  // Clear the input buffer
        return 1;  // Exit the program due to invalid input
    }

    int arr[size];

    printf("Enter %d elements in sorted order: ", size);
    for (int i = 0; i < size; i++) {
        if (scanf("%d", &arr[i]) != 1) {
            printf("Invalid input. Please enter integer values.\n");
            while (getchar() != '\n');  // Clear the input buffer
            return 1;  // Exit the program due to invalid input
        }
    }

    printf("Enter the element to search: ");
    if (scanf("%d", &key) != 1) {
        printf("Invalid input. Please enter an integer.\n");
        while (getchar() != '\n');  // Clear the input buffer
        return 1;  // Exit the program due to invalid input
    }

    do {
        printf("\nChoose the search algorithm:"
               "\n1. Linear Search"
               "\n2. Binary Search"
               "\nEnter your choice: ");
        if (scanf("%d", &choice) != 1) {
            printf("Invalid input. Please enter a number.\n");
```

```
38.             while (getchar() != '\n');  // Clear the input buffer
39.             continue;
40.         }
41.         if (choice < 1 || choice > 2) {
42.             printf("Invalid choice. Please try again.\n");
43.             while (getchar() != '\n');  // Clear the input buffer
44.             continue;
45.         }
46.         break;  // Exit the loop if input is valid
47.     } while (1);
48.
49.     switch (choice) {
50.         case 1:
51.             printf("\nLinear Search:");
52.             for (int i = 0; i < size; i++) {
53.                 if (arr[i] == key) {
54.                     found = 1;
55.                     index = i;
56.                     break;
57.                 }
58.             }
59.             if (found) {
60.                 printf("\nElement found at index %d", index);
61.             } else {
62.                 printf("\nElement not found in the array");
63.             }
64.             break;
65.         case 2:
66.             printf("\nBinary Search:");
67.             int low = 0, high = size - 1;
68.             while (low <= high) {
69.                 int mid = (low + high) / 2;
70.                 if (arr[mid] == key) {
71.                     found = 1;
72.                     index = mid;
73.                     break;
74.                 } else if (arr[mid] < key) {
75.                     low = mid + 1;
76.                 } else {
77.                     high = mid - 1;
```

```
            }
        }
        if (found) {
            printf("\nElement found at index %d", index);
        } else {
            printf("\nElement not found in the array");
        }
        break;
    default:
        printf("\nInvalid choice.");
        break;
  }

  return 0;
}
```

Q-12) C program for searching in an array [using functions and pointers]

```
#include <stdio.h>

// Function for Linear Search
int linearSearch(int *arr, int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Return the index if found
        }
    }
    return -1; // Return -1 if not found
}

// Function for Binary Search
int binarySearch(int *arr, int low, int high, int key) {
    if (high >= low) {
        int mid = low + (high - low) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == key)
            return mid;

        // If element is smaller than mid, then it can only be present in left subarray
        if (arr[mid] > key)
            return binarySearch(arr, low, mid - 1, key);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid + 1, high, key);
```

```
28.     }
29.
30.     // We reach here when element is not present in array
31.     return -1;
32. }
33.
34. int main() {
35.     int size, key, choice, index = -1;
36.
37.     printf("Enter the size of the array: ");
38.     scanf("%d", &size);
39.
40.     int arr[size];
41.
42.     printf("Enter %d elements in sorted order: ", size);
43.     for (int i = 0; i < size; i++) {
44.         scanf("%d", &arr[i]);
45.     }
46.
47.     printf("Enter the element to search: ");
48.     scanf("%d", &key);
49.
50.     do {
51.         printf("\nChoose the search algorithm:"
52.                "\n1. Linear Search"
53.                "\n2. Binary Search"
54.                "\nEnter your choice: ");
55.         if (scanf("%d", &choice) != 1) {
56.             printf("Invalid input. Please enter a number.\n");
57.             while (getchar() != '\n'); // clear the input buffer
58.             continue;
59.         }
60.
61.         if (choice < 1 || choice > 2) {
62.             printf("Invalid choice. Please try again.\n");
63.             while (getchar() != '\n'); // clear the input buffer to handle further wrong inputs
64.             continue;
65.         }
66.
67.         // Valid input, exit the loop
68.         break;
69.     } while (1);
70.
71.     switch (choice) {
72.         case 1:
73.             printf("\nLinear Search:");
```

```
            index = linearSearch(arr, size, key);
            break;
        case 2:
            printf("\nBinary Search:");
            index = binarySearch(arr, 0, size - 1, key);
            break;
    }

    if (index != -1) {
        printf("\nElement found at index %d", index);
    } else {
        printf("\nElement not found in the array");
    }

    return 0;
}
```

Q-13) C program for searching in an array [using functions]

```
#include <stdio.h>

// Function for Linear Search
int linearSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Return the index if found
        }
    }
    return -1; // Return -1 if not found
}

// Function for Binary Search
int binarySearch(int arr[], int low, int high, int key) {
    while (high >= low) {
        int mid = low + (high - low) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == key) {
            return mid;
        }

        // If element is smaller than mid, then it can only be present in left subarray
        if (arr[mid] > key) {
            high = mid - 1;
        } else {
            // Else the element can only be present in right subarray
```

```
            low = mid + 1;
        }
    }

    // We reach here when element is not present in array
    return -1;
}

int main() {
    int size, key, choice, index = -1;

    printf("Enter the size of the array: ");
    if (scanf("%d", &size) != 1 || size <= 0) {
        printf("Invalid input. Please enter a positive integer for size.\n");
        return 0; // Exit on invalid input for size
    }

    int arr[size];

    printf("Enter %d elements in sorted order: ", size);
    for (int i = 0; i < size; i++) {
        if (scanf("%d", &arr[i]) != 1) {
            printf("Invalid input. Please enter integers only.\n");
            return 0; // Exit on invalid input during array entry
        }
    }

    printf("Enter the element to search: ");
    if (scanf("%d", &key) != 1) {
        printf("Invalid input. Please enter a valid integer for the search key.\n");
        return 0; // Exit on invalid input for key
    }

    printf("\nChoose the search algorithm:"
           "\n1. Linear Search"
           "\n2. Binary Search"
           "\nEnter your choice: ");
    if (scanf("%d", &choice) != 1) {
        printf("Invalid input. Please enter a number.\n");
        while (getchar() != '\n'); // clear the input buffer
        return 0; // Exit on invalid input for choice
    }

    switch (choice) {
        case 1:
            printf("\nLinear Search:");
```

```
            index = linearSearch(arr, size, key);
            break;
        case 2:
            printf("\nBinary Search:");
            index = binarySearch(arr, 0, size - 1, key);
            break;
        default:
            printf("\nInvalid choice. Please choose either 1 for Linear Search or 2 for Binary Search.");
            return 0; // Exit if an invalid choice is made
    }

    if (index != -1) {
        printf("\nElement found at index %d", index);
    } else {
        printf("\nElement not found in the array");
    }

    return 0;
}
```

Q-14) C program for searching in an array [using pointers]

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

int main() {
    int size, key, choice, found = 0, index = -1;

    printf("Enter the size of the array: ");
    if (scanf("%d", &size) != 1 || size <= 0) {
        printf("Invalid input. Please enter a positive integer.\n");
        while (getchar() != '\n'); // Clear the input buffer
        return 0; // Exit if input is not valid
    }

    // Dynamically allocate memory for the array using malloc
    int *p = (int *)malloc(size * sizeof(int));
    if (p == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Return with error code
    }

    printf("Enter %d elements in sorted order: ", size);
    for (int i = 0; i < size; i++) {
```

```
23.     if (scanf("%d", p + i) != 1) { // Directly storing into dynamically allocated array
24.         printf("Invalid input. Please enter integers only.\n");
25.         while (getchar() != '\n'); // Clear the input buffer
26.         free(p);
27.         return 0;
28.     }
29. }
30.
31. printf("Enter the element to search: ");
32. if (scanf("%d", &key) != 1) {
33.     printf("Invalid input. Please enter an integer.\n");
34.     while (getchar() != '\n'); // Clear the input buffer
35.     free(p);
36.     return 0;
37. }
38.
39. printf("\nChoose the search algorithm:"
40.        "\n1. Linear Search"
41.        "\n2. Binary Search"
42.        "\nEnter your choice: ");
43. if (scanf("%d", &choice) != 1) {
44.     printf("Invalid input. Please enter a number.\n");
45.     while (getchar() != '\n'); // Clear the input buffer
46.     free(p);
47.     return 0;
48. }
49.
50. switch (choice) {
51.     case 1:
52.         printf("\nLinear Search:");
53.         for (int i = 0; i < size; i++) {
54.             if (*(p + i) == key) {
55.                 found = 1;
56.                 index = i;
57.                 break;
58.             }
59.         }
60.         if (found) {
61.             printf("\nElement found at index %d", index);
62.         } else {
63.             printf("\nElement not found in the array");
64.         }
65.         break;
66.     case 2:
67.         printf("\nBinary Search:");
68.         int low = 0, high = size - 1;
```

```
69.             while (low <= high) {
70.                 int mid = (low + high) / 2;
71.                 if (*(p + mid) == key) {
72.                     found = 1;
73.                     index = mid;
74.                     break;
75.                 } else if (*(p + mid) < key) {
76.                     low = mid + 1;
77.                 } else {
78.                     high = mid - 1;
79.                 }
80.             }
81.             if (found) {
82.                 printf("\nElement found at index %d", index);
83.             } else {
84.                 printf("\nElement not found in the array");
85.             }
86.             break;
87.         default:
88.             printf("\nInvalid choice. Please choose 1 for Linear Search or 2 for Binary
    Search.");
89.             break;
90.     }
91.
92.     free(p); // Free the dynamically allocated memory
93.     return 0;
94. }
```

ALGORITHMS FOR QUESTION 11-14

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare integer variables `size`, `key`, `choice`, `found`, and `index` to store the size of the array, the key to search for, the user's choice of search algorithm, a flag indicating if the key is found, and the index of the found key.
4. Prompt the user to enter the size of the array.
   - If the input is not an integer or is less than or equal to 0, display an error message, clear the input buffer, and exit the program.
5. Declare an integer array `arr` of size `size`.
6. Prompt the user to enter `size` elements in sorted order.
   - If the input is not an integer, display an error message, clear the input buffer, and exit the program.
7. Prompt the user to enter the element to search for.
   - If the input is not an integer, display an error message, clear the input buffer, and exit the program.
8. Enter a `do-while` loop to prompt the user to choose the search algorithm.
   - If the input is not an integer or is not within the valid range, display an error message, clear the input buffer, and continue.
9. Use a `switch` statement to handle the user's choice:
   - --Case 1: Linear Search--
     - Iterate through the array using a `for` loop.
     - If the current element equals the key, set `found` to `1`, store the index, and break the loop.
     - If the key is found, display the index. Otherwise, display a message indicating the key is not found.
   - --Case 2: Binary Search--
     - Initialize `low` to `0` and `high` to `size - 1`.
     - Enter a `while` loop that continues as long as `low` is less than or equal to `high`.
     - Calculate the middle index `mid`.
     - If the element at `mid` equals the key, set `found` to `1`, store the index, and break the loop.
     - If the element at `mid` is less than the key, update `low` to `mid + 1`.
     - If the element at `mid` is greater than the key, update `high` to `mid - 1`.
     - If the key is found, display the index. Otherwise, display a message indicating the key is not found.
   - If the user's choice is not within the valid range, display an error message.
10. --End--

2] fp

1. --Start--
2. Include necessary header files for standard input/output operations.

3. Declare a function `linearSearch` that takes an array, its size, and a key as parameters. This function iterates through the array to find the key. If found, it returns the index of the key; otherwise, it returns `-1`.
4. Declare a function `binarySearch` that takes an array, the lower and upper bounds of the array, and a key as parameters. This function uses a divide-and-conquer approach to find the key. If the key is found, it returns the index of the key; otherwise, it returns `-1`.
5. In the `main` function, declare variables for the size of the array, the key to search for, the user's choice of search algorithm, and the index where the key is found.
6. Prompt the user to enter the size of the array and read it.
7. Declare an array of the specified size.
8. Prompt the user to enter the elements of the array in sorted order and read them.
9. Prompt the user to enter the element to search for and read it.
10. Enter a `do-while` loop to ensure the user enters a valid choice for the search algorithm.
   - Display the options for the search algorithm.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message, clear the input buffer, and continue.
   - If the input is valid, exit the loop.
11. Use a `switch` statement to handle the user's choice:
   - --Case 1: Linear Search--
      - Call `linearSearch` with the array, its size, and the key.
   - --Case 2: Binary Search--
      - Call `binarySearch` with the array, the lower bound (`0`), the upper bound (`size - 1`), and the key.
12. If the index is not `-1`, display the message that the element was found at the given index.
   - Otherwise, display the message that the element was not found in the array.
13. --End--

3] f

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare a function `linearSearch` that takes an array, its size, and a key as parameters. It iterates through the array to find the key, returning the index if found or `-1` if not found.
4. Declare a function `binarySearch` that takes an array, the low and high indices, and a key as parameters. It performs a binary search to find the key, returning the index if found or `-1` if not found.
5. In the `main` function, declare integer variables `size`, `key`, `choice`, and `index` to store the array size, the element to search for, the user's choice of search algorithm, and the index of the found element.
6. Prompt the user to enter the size of the array.

- If the input is not an integer or is less than or equal to `0`, display an error message and exit the program.
7. Declare an integer array `arr` of size `size`.
8. Prompt the user to enter `size` elements in sorted order.
   - If the input is not an integer, display an error message and exit the program.
9. Prompt the user to enter the element to search for.
   - If the input is not an integer, display an error message and exit the program.
10. Prompt the user to choose the search algorithm, displaying options for Linear Search and Binary Search.
   - If the input is not an integer, clear the input buffer and exit the program.
11. Use a `switch` statement to handle the user's choice:
   - --Case 1: Linear Search--
     - Call `linearSearch` with the array, its size, and the key.
   - --Case 2: Binary Search--
     - Call `binarySearch` with the array, `0` as the low index, `size - 1` as the high index, and the key.
   - If the user's choice is not within the valid range, display an error message and exit the program.
12. If the `index` is not `-1`, display a message indicating that the element was found at the given index.
   - Otherwise, display a message indicating that the element was not found in the array.
13. --End--

4] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. In the `main` function, declare integer variables `size`, `key`, `choice`, `found`, and `index`.
4. Prompt the user to enter the size of the array.
   - If the input is not a positive integer, display an error message, clear the input buffer, and exit the program.
5. Dynamically allocate memory for the array using `malloc`.
   - If memory allocation fails, display an error message and exit the program.
6. Prompt the user to enter elements in sorted order.
   - If the input is not valid, display an error message, clear the input buffer, free the allocated memory, and exit the program.
7. Prompt the user to enter the element to search for.
   - If the input is not valid, display an error message, clear the input buffer, free the allocated memory, and exit the program.
8. Prompt the user to choose the search algorithm: Linear Search or Binary Search.
   - If the input is not valid, display an error message, clear the input buffer, free the allocated memory, and exit the program.
9. Use a `switch` statement to handle the user's choice:

- --Case 1: Linear Search--
  - Iterate through the array from the beginning.
  - If the current element matches the key, set `found` to `1`, store the index, and break the loop.
  - If the element is found, display the index. Otherwise, display a message indicating the element is not found.
- --Case 2: Binary Search--
  - Initialize `low` to `0` and `high` to `size - 1`.
  - While `low` is less than or equal to `high`:
    - Calculate the middle index `mid`.
    - If the element at `mid` matches the key, set `found` to `1`, store the index, and break the loop.
    - If the element at `mid` is less than the key, update `low` to `mid + 1`.
    - Otherwise, update `high` to `mid - 1`.
  - If the element is found, display the index. Otherwise, display a message indicating the element is not found.
- If the user's choice is not within the valid range, display an error message.

10. Free the dynamically allocated memory.
11. --End--

Q-15) C program to perform operations in stack using array[without use of functions and pointers]

```
#include <stdio.h>

int main() {
    int choice, value, index = -1, found = 0;

    printf("Enter the size of the stack: ");
    int size;
    if (scanf("%d", &size) != 1 || size <= 0) {
        printf("Invalid input. Please enter a positive integer.\n");
        while (getchar() != '\n'); // Clear the input buffer
        return 0; // Exit on invalid size input
    }

    int stackArray[size]; // Declare VLA with user-specified size
    int top = -1; // Initialize top to -1 to indicate the stack is empty

    do {
        printf("\t\tMAIN MENU for operations of Stack in array\n"
               "\n1. Push the element\n"
               "2. Pop the element\n"
               "3. Display the elements\n"
               "4. Peek the element\n"
               "5. EXIT\n"
               "\nEnter your choice (1-5): ");
        if (scanf("%d", &choice) != 1) {
            printf("Invalid input. Please enter a number.\n");
            while (getchar() != '\n'); // Clear the input buffer
            continue; // Skip to the next iteration of the loop
        }
        if (choice < 1 || choice > 5) {
            printf("Invalid choice. Please try again.\n");
            continue; // Skip to the next iteration of the loop
        }

        switch (choice) {
            case 1:
                if (top >= size - 1) {
                    printf("Stack is Full\n");
                } else {
                    printf("Enter the value: ");
                    if (scanf("%d", &value) != 1) {
                        printf("Invalid input. Please enter a number.\n");
                        while (getchar() != '\n'); // Clear the input buffer
                        continue; // Skip this iteration, don't push garbage into the stack
```

```
45.                 }
46.                 stackArray[++top] = value; // Push value onto the stack
47.             }
48.             break;
49.         case 2:
50.             if (top == -1) {
51.                 printf("Stack is Empty\n");
52.             } else {
53.                 printf("Popped value is: %d\n", stackArray[top--]); // Pop the value from
    the stack
54.             }
55.             break;
56.         case 3:
57.             if (top == -1) {
58.                 printf("Stack is empty\n");
59.             } else {
60.                 printf("Stack elements:\n");
61.                 for (int i = top; i >= 0; i--) {
62.                     printf("%d\n", stackArray[i]);
63.                 }
64.             }
65.             break;
66.         case 4:
67.             if (top == -1) {
68.                 printf("Stack is empty\n");
69.             } else {
70.                 printf("Top element is: %d\n", stackArray[top]); // Peek the top element
71.             }
72.             break;
73.         case 5:
74.             printf("Exiting...\n");
75.             break;
76.       }
77.   } while (choice != 5);
78.
79.   return 0;
80. }
```

Q-16) C program to perform operations in stack using array[use of functions and pointers]

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int* stackArray;   // Pointer for dynamic array
5. int* topPtr = NULL;  // Pointer to the top of the stack
```

```
6.  int maxSize;        // Maximum size of the stack
7.
8.  int peekElement() {
9.      if (topPtr == NULL) {
10.         printf("Stack is empty\n");
11.     } else {
12.         printf("Top element is: %d\n", *topPtr);
13.     }
14.     return 0;
15. }
16.
17. void displayElements() {
18.     if (topPtr == NULL) {
19.         printf("Stack is empty\n");
20.     } else {
21.         printf("Stack elements:\n");
22.         for (int* ptr = topPtr; ptr >= stackArray; ptr--) {
23.             printf("%d\n", *ptr);
24.         }
25.     }
26. }
27.
28. int popElement() {
29.     if (topPtr == NULL) {
30.         printf("Stack is Empty\n");
31.         return 0;
32.     } else {
33.         int item = *topPtr;  // Get the value at the top position
34.         topPtr--;  // Move the pointer down to the previous position
35.         if (topPtr < stackArray - 1) topPtr = NULL; // Check if the stack is now empty
36.         printf("Popped value is: %d\n", item);
37.         return item;
38.     }
39. }
40.
41. int pushElement(int value) {
42.     if (topPtr == stackArray + maxSize - 1) {
43.         printf("Stack is Full\n");
44.         return -1;
45.     } else {
46.         if (topPtr == NULL) topPtr = stackArray - 1;  // Initialize topPtr if it's the first
    push
47.         topPtr++;  // Move the pointer to the next available position
48.         *topPtr = value;  // Store the value at the top position
49.         return 0;
50.     }
```

```
51. }
52.
53. int main() {
54.     int choice, value;
55.
56.     printf("Enter the maximum size of the stack: ");
57.     if (scanf("%d", &maxSize) != 1 || maxSize <= 0) {
58.         printf("Invalid input. Please enter a positive integer.\n");
59.         return 0; // Exit on invalid size input
60.     }
61.
62.     stackArray = (int*) malloc(maxSize * sizeof(int));  // Dynamically allocate memory
    for the stack
63.     if (stackArray == NULL) {
64.         printf("Failed to allocate memory for the stack.\n");
65.         return 1; // Exit if memory allocation fails
66.     }
67.     topPtr = NULL;  // Initialize topPtr to NULL indicating stack is empty
68.
69.     do {
70.         printf("\t\tMAIN MENU for operations on Stack in array\n"
71.                "1. Push the element\n"
72.                "2. Pop the element\n"
73.                "3. Display the elements\n"
74.                "4. Peek the element\n"
75.                "5. EXIT\n"
76.                "\nEnter your choice (1-5): ");
77.         if (scanf("%d", &choice) != 1) {
78.             printf("Invalid input. Please enter a number.\n");
79.             while (getchar() != '\n'); // Clear the input buffer
80.             continue;
81.         }
82.         if (choice < 1 || choice > 5) {
83.             printf("Invalid choice. Please try again.\n");
84.             continue;
85.         }
86.
87.         switch (choice) {
88.             case 1:
89.                 printf("Enter the value: ");
90.                 if (scanf("%d", &value) != 1) {
91.                     printf("Invalid input. Please enter a number.\n");
92.                     while (getchar() != '\n'); // Clear the input buffer
93.                     continue;
94.                 }
95.                 pushElement(value);
```

```
            break;
        case 2:
            popElement();
            break;
        case 3:
            displayElements();
            break;
        case 4:
            peekElement();
            break;
        case 5:
            printf("Exiting...\n");
            break;
    }
} while (choice != 5);

free(stackArray); // Free the dynamically allocated memory
return 0;
}
```

Q-17) C program to perform operations in stack using array[use of pointers]

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

int main() {
    int choice, value, stackSize;
    int* stackArray;
    int* topPtr;

    printf("Enter the stack size: ");
    scanf("%d", &stackSize);

    // Allocate memory for the stack array
    stackArray = (int*)malloc(stackSize * sizeof(int));
    if (stackArray == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    topPtr = stackArray; // Initialize topPtr to point to the start of stackArray

    do {
        printf("\t\tMAIN MENU for operations on Stack in array\n"
               "\n1. Push the element\n"
               "2. Pop the element\n"
```

```
25.             "3. Display the elements\n"
26.             "4. Peek the element\n"
27.             "5. EXIT\n"
28.             "\nEnter your choice (1-5): ");
29.         scanf("%d", &choice);
30.
31.         if (choice > 5 || choice < 1) {
32.             printf("Invalid choice\n");
33.         }
34.
35.         switch (choice) {
36.             case 1: {
37.                 printf("Enter the value: ");
38.                 scanf("%d", &value);
39.
40.                 // Push logic
41.                 if (topPtr == stackArray + stackSize) {
42.                     printf("Stack is Full\n");
43.                 } else {
44.                     *topPtr = value; // Store the value at the top position
45.                     topPtr++; // Move the pointer to the next available position
46.                 }
47.                 break;
48.             }
49.             case 2: {
50.                 // Pop logic
51.                 if (topPtr == stackArray) {
52.                     printf("Stack is Empty\n");
53.                 } else {
54.                     topPtr--; // Move the pointer down to the previous position
55.                     value = *topPtr; // Get the value at the top position
56.                     printf("Popped value is: %d\n", value);
57.                 }
58.                 break;
59.             }
60.             case 3: {
61.                 // Display logic
62.                 int* ptr = stackArray;
63.                 if (ptr == topPtr) {
64.                     printf("Stack is empty\n");
65.                 } else {
66.                     printf("Stack elements:\n");
67.                     while (ptr < topPtr) {
68.                         printf("%d\n", *ptr);
69.                         ptr++; // Move the pointer up
70.                     }
```

```
            }
            break;
        }
        case 4: {
            // Peek logic
            if (topPtr == stackArray) {
                printf("Stack is empty\n");
            } else {
                printf("Top element is: %d\n", *(topPtr - 1));
            }
            break;
        }
        case 5: {
            printf("Or bhai aa gye sawad DSA ke\n");
            break;
        }
      }
    } while (choice < 5);

    // Free the allocated memory
    free(stackArray);

    return 0;
}
```

Q-18) C program to perform operations in stack using array[use of functions]

```
// C program for Implementation of stack in array  using functions

#include <stdio.h>
#define MAX_SIZE 1000

int peekElement();
int popElement();
void displayElements();
int pushElement(int);

int stackArray[MAX_SIZE];
int topIndex = -1;

int peekElement() {
    if (topIndex == -1) {
        printf("Stack is empty\n");
    } else {
```

```
18.         printf("%d\n", stackArray[topIndex]);
19.     }
20.     return 0;
21. }
22.
23. int main() {
24.     int choice;
25.     do {
26.         printf("\t\tMAIN MENU for operations on Stack in array\n"
27.                 "\n1. Push the element\n"
28.                 "2. Pop the element\n"
29.                 "3. Peek the element\n"
30.                 "4. Display the element\n"
31.                 "5. EXIT\n"
32.                 "\nEnter your choice (1-5): ");
33.         scanf("%d", &choice);
34.
35.         if (choice > 5 || choice < 0) {
36.             printf("Invalid choice\n");
37.         }
38.
39.         switch (choice) {
40.             case 1: {
41.                 int value;
42.                 printf("Enter the value: ");
43.                 scanf("%d", &value);
44.                 pushElement(value);
45.                 break;
46.             }
47.             case 2: {
48.                 popElement();
49.                 break;
50.             }
51.             case 3: {
52.                 peekElement();
53.                 break;
54.             }
55.             case 4: {
56.                 displayElements();
57.                 break;
58.             }
59.             case 5: {
60.                 printf("Exit...\n");
61.                 break;
62.             }
63.         }
```

```
64.     } while (choice < 5);
65.
66.     return 0;
67. }
68.
69. int popElement() {
70.     int item;
71.     if (topIndex == -1) {
72.         printf("Stack is Empty\n");
73.         return 0;
74.     } else {
75.         item = stackArray[topIndex];
76.         topIndex--;
77.         printf("Popped value is: %d\n", item);
78.     }
79.     return 0;
80. }
81.
82. void displayElements() {
83.     for (int i = topIndex; i >= 0; i--) {
84.         printf("%d\n", stackArray[i]);
85.     }
86. }
87.
88. int pushElement(int value) {
89.     if (topIndex == MAX_SIZE - 1) {
90.         printf("Stack is Full\n");
91.     } else {
92.         topIndex++;
93.         stackArray[topIndex] = value;
94.     }
95.     return 0;
96. }
```

Q-19) C program to perform operations in stack using LL[use of functions and pointers]

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  struct NodeStruct {
5.      int value;
6.      struct NodeStruct *nextPtr;
7.  };
8.  struct NodeStruct *topPtr = NULL;
9.
```

```
10. int peekElement();
11. void displayElements();
12. int popElement();
13. int pushElement(int);
14.
15. int peekElement() {
16.     if (topPtr == NULL) {
17.         printf("Stack is Empty\n");
18.         return 0;
19.     } else {
20.         printf("The top of the Data is %d\n", topPtr->value);
21.     }
22.     return 1;
23. }
24.
25. int main() {
26.     int choice;
27.     do {
28.         printf("\t\tMAIN MENU for operations on Stack in array\n");
29.         printf("\n1. Push the element\n");
30.         printf("2. Peek the element\n");
31.         printf("3. Pop the element\n");
32.         printf("4. Display\n");
33.         printf("5. EXIT\n");
34.         printf("\nEnter your choice (1-5): ");
35.         scanf("%d", &choice);
36.
37.         if (choice > 5 || choice < 0) {
38.             printf("Invalid choice\n");
39.         }
40.
41.         switch (choice) {
42.             case 1: {
43.                 int data;
44.                 printf("Enter the data: ");
45.                 scanf("%d", &data);
46.                 pushElement(data);
47.                 break;
48.             }
49.             case 2: {
50.                 peekElement();
51.                 break;
52.             }
53.             case 3: {
54.                 popElement();
55.                 break;
```

```
56.             }
57.             case 4: {
58.                 displayElements();
59.                 break;
60.             }
61.             case 5: {
62.                 printf("hmm bhadiya execute hua hai tera code. Virendra ka khoof kuch
    kaam to aaya \n");
63.                 break;
64.             }
65.         }
66.     } while (choice < 5);
67.
68.     return 0;
69. }
70.
71. void displayElements() {
72.     struct NodeStruct *temp;
73.     if (topPtr == NULL) {
74.         printf("Stack is Empty\n");
75.         return;
76.     } else {
77.         temp = topPtr;
78.         while (temp != NULL) {
79.             printf("%d\n", temp->value);
80.             temp = temp->nextPtr;
81.         }
82.     }
83. }
84.
85. int popElement() {
86.     struct NodeStruct *temp;
87.     if (topPtr == NULL) {
88.         printf("The stack is Empty\n");
89.         return 0;
90.     } else {
91.         temp = topPtr;
92.         topPtr = topPtr->nextPtr;
93.         free(temp);
94.     }
95.     return 1;
96. }
97.
98. int pushElement(int data) {
99.     struct NodeStruct *newNode;
100.             newNode = (struct NodeStruct *)malloc(sizeof(struct NodeStruct));
```

```
            newNode->value = data;
            newNode->nextPtr = topPtr;
            topPtr = newNode;
            return 1;
        }

```

Q-20) C program to perform operations in stack using LL[use of pointers]

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

struct NodeStruct {
    int value;
    struct NodeStruct *nextPtr;
};

int main() {
    struct NodeStruct *topPtr = NULL;
    int choice;

    do {
        printf("\t\tMAIN MENU for operations on Stack in array\n"
            "\n1. Push the element\n"
            "2. Peek the element\n"
            "3. Pop the element\n"
            "4. Display\n"
            "5. EXIT\n"
            "\nEnter your choice (1-5): ");
        scanf("%d", &choice);

        if (choice > 5 || choice < 0) {
            printf("Invalid choice\n");
        }

        switch (choice) {
            case 1: {
                int data;
                printf("Enter the data: ");
                scanf("%d", &data);

                // Push logic
                struct NodeStruct *newNode = (struct NodeStruct *)malloc(sizeof(struct
    NodeStruct));
                newNode->value = data;
                newNode->nextPtr = topPtr;
                topPtr = newNode;
```

```
38.             break;
39.         }
40.         case 2: {
41.             // Peek logic
42.             if (topPtr == NULL) {
43.                 printf("Stack is Empty\n");
44.             } else {
45.                 printf("The top of the Data is %d\n", topPtr->value);
46.             }
47.             break;
48.         }
49.         case 3: {
50.             // Pop logic
51.             if (topPtr == NULL) {
52.                 printf("The stack is Empty\n");
53.             } else {
54.                 struct NodeStruct *temp = topPtr;
55.                 topPtr = topPtr->nextPtr;
56.                 free(temp);
57.             }
58.             break;
59.         }
60.         case 4: {
61.             // Display logic
62.             struct NodeStruct *temp = topPtr;
63.             if (topPtr == NULL) {
64.                 printf("Stack is Empty\n");
65.             } else {
66.                 while (temp != NULL) {
67.                     printf("%d\n", temp->value);
68.                     temp = temp->nextPtr;
69.                 }
70.             }
71.             break;
72.         }
73.         case 5: {
74.             printf("Exit.....\n");
75.             break;
76.         }
77.     }
78. } while (choice < 5);
79.
80. return 0;
81. }
```

ALGORITHM FOR QUESTION 15-20

1] normal,array

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare the main function.
4. Initialize variables for user choice (`choice`), value to be pushed or popped (`value`), index for search operations (`index`), and a flag to indicate if a value is found (`found`).
5. Prompt the user to enter the size of the stack.
6. Validate the input for the size of the stack. If the input is invalid (not a positive integer), display an error message, clear the input buffer, and exit the program.
7. Declare a variable-length array (VLA) `stackArray` with the user-specified size and initialize the `top` index to `-1` to indicate the stack is empty.
8. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
9. Use a `switch` statement to handle the user's choice:
   - --Case 1: Push the Element--
     - If the stack is full, display a message indicating that the stack is full.
     - Otherwise, prompt the user to enter a value and push it onto the stack.
   - --Case 2: Pop the Element--
     - If the stack is empty, display a message indicating that the stack is empty.
     - Otherwise, pop the top element from the stack and display it.
   - --Case 3: Display the Elements--
     - If the stack is empty, display a message indicating that the stack is empty.
     - Otherwise, display all elements in the stack from top to bottom.
   - --Case 4: Peek the Element--
     - If the stack is empty, display a message indicating that the stack is empty.
     - Otherwise, display the top element of the stack without removing it.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
10. --End--

2] fp, array

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Declare a global pointer `stackArray` for the dynamic array representing the stack.

4. Declare a global pointer `topPtr` to keep track of the top of the stack. Initialize it to `NULL`.
5. Declare a global integer `maxSize` to store the maximum size of the stack.
6. Declare functions for peeking at the top element (`peekElement`), displaying all elements (`displayElements`), popping an element (`popElement`), and pushing an element (`pushElement`).
7. In the `main` function, prompt the user to enter the maximum size of the stack.
   - If the input is not a positive integer, display an error message and exit.
8. Dynamically allocate memory for the stack using `malloc`, based on the user-specified `maxSize`.
   - If memory allocation fails, display an error message and exit.
9. Initialize `topPtr` to `NULL` to indicate that the stack is empty.
10. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
11. Use a `switch` statement to handle the user's choice:
   - --Case 1: Push an Element--
     - Prompt the user to enter a value.
     - Call `pushElement` with the entered value.
   - --Case 2: Pop an Element--
     - Call `popElement` to remove and display the top element.
   - --Case 3: Display the Elements--
     - Call `displayElements` to display all elements in the stack.
   - --Case 4: Peek the Element--
     - Call `peekElement` to display the top element without removing it.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
12. After the loop, free the dynamically allocated memory for the stack using `free`.
13. --End--

3] p,array

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. In the `main` function, declare variables for the user's choice, the value to be pushed or popped, the size of the stack, and pointers for the stack array and the top of the stack.
4. Prompt the user to enter the stack size and allocate memory for the stack array using `malloc`.
5. Check if the memory allocation was successful. If not, display an error message and exit the program.
6. Initialize `topPtr` to point to the start of the `stackArray`.

7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the choice is not within the valid range, display an error message.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Push the Element--
     - Prompt the user to enter the value to be pushed.
     - Check if the stack is full. If so, display a message indicating that the stack is full.
     - Otherwise, store the value at the top position of the stack and increment `topPtr`.
   - --Case 2: Pop the Element--
     - Check if the stack is empty. If so, display a message indicating that the stack is empty.
     - Otherwise, decrement `topPtr`, retrieve the value at the top position, and display it.
   - --Case 3: Display the Elements--
     - Check if the stack is empty. If so, display a message indicating that the stack is empty.
     - Otherwise, iterate through the stack from the bottom to the top, displaying each element.
   - --Case 4: Peek the Element--
     - Check if the stack is empty. If so, display a message indicating that the stack is empty.
     - Otherwise, display the value at the top position of the stack.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
9. After the loop, free the allocated memory for the stack array using `free`.
10. --End--

4] f, array

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Define a constant `MAX_SIZE` to represent the maximum size of the stack.
4. Declare an integer array `stackArray` of size `MAX_SIZE` to store the stack elements and an integer `topIndex` initialized to `-1` to represent the top of the stack.
5. Declare functions for peeking at the top element (`peekElement`), popping the top element (`popElement`), displaying all elements (`displayElements`), and pushing a new element onto the stack (`pushElement`).
6. In the `main` function, declare an integer variable `choice` to store the user's menu choice.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Push the Element--
     - Prompt the user to enter a value.

    - Call `pushElement` with the entered value to add it to the stack.
  - --Case 2: Pop the Element--
    - Call `popElement` to remove the top element from the stack.
  - --Case 3: Peek the Element--
    - Call `peekElement` to display the top element of the stack without removing it.
  - --Case 4: Display the Element--
    - Call `displayElements` to display all elements in the stack from top to bottom.
  - --Case 5: Exit--
    - Display a message indicating that the program is exiting.
9. --End--

Function Logic:

- --peekElement--:
  - Checks if the stack is empty (`topIndex == -1`).
  - If not empty, prints the top element of the stack (`stackArray[topIndex]`).
  - Returns `0`.

- --popElement--:
  - Checks if the stack is empty.
  - If not empty, removes the top element by decrementing `topIndex` and prints the popped value.
  - Returns `0`.

- --displayElements--:
  - Iterates from `topIndex` to `0` (inclusive) and prints each element of the stack.

- --pushElement--:
  - Checks if the stack is full (`topIndex == MAX_SIZE - 1`).
  - If not full, increments `topIndex` and adds the new element to the stack at the new top index.
  - Returns `0`.

5] fp, linked list

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `NodeStruct` for the stack nodes, containing an integer `value` and a pointer `nextPtr` to the next node.
4. Declare a global pointer `topPtr` initialized to `NULL` to represent the top of the stack.
5. Declare functions for peeking the top element, displaying all elements, popping the top element, and pushing a new element onto the stack.
6. In the `main` function, declare an integer variable `choice` to store the user's menu choice.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.

- Display the main menu options to the user.
- Prompt the user to enter their choice.
- If the input is not within the valid range, display an error message and continue.

8. Use a `switch` statement to handle the user's choice:
- --Case 1: Push an Element--
  - Prompt the user to enter the data to be pushed onto the stack.
  - Call `pushElement` with the entered data to add it to the stack.
- --Case 2: Peek the Top Element--
  - Call `peekElement` to display the top element of the stack.
- --Case 3: Pop the Top Element--
  - Call `popElement` to remove the top element from the stack.
- --Case 4: Display All Elements--
  - Call `displayElements` to display all elements in the stack.
- --Case 5: Exit the Program--
  - Display a message indicating that the program is exiting.

9. --End--

Function Descriptions

- --`peekElement`--: Checks if the stack is empty. If not, it prints the value of the top element. Returns `1` if successful, `0` otherwise.
- --`displayElements`--: Iterates through the stack from the top to the bottom, printing each element's value. If the stack is empty, it prints a message indicating that the stack is empty.
- --`popElement`--: Checks if the stack is empty. If not, it removes the top element from the stack by updating `topPtr` to point to the next element and frees the memory of the removed node. Returns `1` if successful, `0` otherwise.
- --`pushElement`--: Allocates memory for a new node, sets its `value` to the given data, and its `nextPtr` to the current top of the stack. Then, updates `topPtr` to point to the new node. Returns `1` if successful.

6] p, linked-list

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `NodeStruct` for the stack nodes, containing an integer `value` to store the data and a pointer `nextPtr` to the next node in the stack.
4. In the `main` function, initialize a pointer `topPtr` to `NULL` to represent the top of the stack.
5. Declare an integer variable `choice` to store the user's menu choice.
6. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the choice is not within the valid range, display an error message.

7. Use a `switch` statement to handle the user's choice:
   - --Case 1: Push the Element--
     - Prompt the user to enter the data to be pushed onto the stack.
     - Allocate memory for a new node using `malloc`.
     - Assign the entered data to the `value` field of the new node.
     - Set the `nextPtr` of the new node to the current `topPtr`.
     - Update `topPtr` to point to the new node.
   - --Case 2: Peek the Element--
     - If the stack is empty (`topPtr` is `NULL`), display a message indicating that the stack is empty.
     - Otherwise, display the value of the top element.
   - --Case 3: Pop the Element--
     - If the stack is empty (`topPtr` is `NULL`), display a message indicating that the stack is empty.
     - Otherwise, remove the top element by updating `topPtr` to point to the next node and free the memory of the removed node using `free`.
   - --Case 4: Display--
     - If the stack is empty (`topPtr` is `NULL`), display a message indicating that the stack is empty.
     - Otherwise, iterate through the stack from the top to the bottom, printing the value of each node.
   - --Case 5: Exit--
     - Display a message indicating that the program is exiting.
8. --End--

Q-21) C program for infix to postfix and prefix conversion[use of functions]

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

#define MAX_SIZE 1000  // Define a maximum size for expressions and stack usage

// Function to return precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3;
    return 0;
}

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

int main() {
    int choice;
    char exp[MAX_SIZE];  // Declare the expression array with a fixed size

    do {
        printf("\t\tMain menu\n"
               "1. Infix to Postfix\n"
               "2. Postfix to Prefix\n"
               "3. EXIT\n"
               "Enter your choice (1-3): ");
        scanf("%d", &choice);
        while (getchar() != '\n');  // Clear input buffer to prevent leftover input from affecting future reads

        if (choice < 1 || choice > 3) {
            printf("Invalid choice\n");
            continue;  // Prompt again if choice is invalid
        }

        if (choice == 3) {
            printf("Exiting...\n");
            break;  // Exit the loop and program
        }
```

```
41.
42.     printf("Enter the expression: ");
43.     scanf("%s", exp);
44.     while (getchar() != '\n');  // Clear the buffer after reading the expression
45.
46.     switch (choice) {
47.         case 1: {
48.             char result[MAX_SIZE];
49.             int resultIndex = 0;
50.             int len = strlen(exp);
51.             char stack[MAX_SIZE];
52.             int stackIndex = -1;
53.
54.             for (int i = 0; i < len; i++) {
55.                 char c = exp[i];
56.
57.                 if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')) {
58.                     result[resultIndex++] = c;
59.                 } else if (c == '(') {
60.                     stack[++stackIndex] = c;
61.                 } else if (c == ')') {
62.                     while (stackIndex >= 0 && stack[stackIndex] != '(') {
63.                         result[resultIndex++] = stack[stackIndex--];
64.                     }
65.                     stackIndex--; // Pop '(' from the stack
66.                 } else {
67.                     while (stackIndex != -1 && precedence(stack[stackIndex]) >=
    precedence(c) && stack[stackIndex] != '(') {
68.                         result[resultIndex++] = stack[stackIndex--];
69.                     }
70.                     stack[++stackIndex] = c;
71.                 }
72.             }
73.
74.             while (stackIndex >= 0) {
75.                 result[resultIndex++] = stack[stackIndex--];
76.             }
77.
78.             result[resultIndex] = '\0';  // Null-terminate the string
79.             printf("Postfix expression: %s\n", result);
80.             break;
```

```
81.         }
82.         case 2: {
83.             char stack[MAX_SIZE][MAX_SIZE]; // Stack to hold elements
84.             int top = -1;
85.             int len = strlen(exp);
86.
87.             for (int i = 0; i < len; i++) {
88.                 if (exp[i] >= 'a' && exp[i] <= 'z' || exp[i] >= 'A' && exp[i] <= 'Z' || exp[i] >=
    '0' && exp[i] <= '9') {
89.                     // Push operand to stack
90.                     top++;
91.                     stack[top][0] = exp[i];
92.                     stack[top][1] = '\0';
93.                 } else {
94.                     // Operator encountered
95.                     char op1[MAX_SIZE], op2[MAX_SIZE];
96.                     strcpy(op2, stack[top--]);  // Pop two operands
97.                     strcpy(op1, stack[top--]);
98.                     char expr[MAX_SIZE] = {exp[i]}; // Start with operator
99.                     strcat(expr, op1);            // Concatenate first operand
100.                        strcat(expr, op2);            // Concatenate second operand
101.                        top++;
102.                        strcpy(stack[top], expr);      // Push result back to stack
103.                    }
104.                }
105.
106.                // The final prefix expression is on top of the stack
107.                printf("Prefix expression: %s\n", stack[top]);
108.                break;
109.            }
110.            case 3:
111.                break;
112.            default:
113.                printf("Invalid choice\n");
114.        }
115.    } while (choice != 3);
116.
117.    return 0;
118. }
```

Q-22) C program for infix to postfix and prefix conversion[use of functions and pointer]

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

#define MAX_SIZE 1000  // Define a maximum size for expressions and stack usage

// Function to return precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3;
    return 0;
}

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

int main() {
    int choice;
    char exp[MAX_SIZE];  // Declare the expression array with a fixed size
    char *expPtr = exp;  // Pointer to the expression array

    do {
        printf("\t\tMain menu\n"
               "1. Infix to Postfix\n"
               "2. Postfix to Prefix\n"
               "3. EXIT\n"
               "Enter your choice (1-3): ");
        scanf("%d", &choice);
        while (getchar() != '\n');  // Clear input buffer to prevent leftover input from affecting future reads

        if (choice < 1 || choice > 3) {
            printf("Invalid choice\n");
            continue;  // Prompt again if choice is invalid
        }

        if (choice == 3) {
            printf("Exiting...\n");
            break;  // Exit the loop and program
        }

        printf("Enter the expression: ");
```

```
44.         scanf("%s", expPtr);
45.         while (getchar() != '\n');  // Clear the buffer after reading the expression
46.
47.         switch (choice) {
48.             case 1: {
49.                 char result[MAX_SIZE];
50.                 char *resultPtr = result;
51.                 char stack[MAX_SIZE];
52.                 char *stackPtr = stack - 1;  // Stack pointer initialized to just before the stack
    start
53.
54.                 for (int i = 0; i < strlen(expPtr); i++) {
55.                     char c = expPtr[i];
56.
57.                     if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
    {
58.                         *resultPtr++ = c;  // Directly append the operand to the result
59.                     } else if (c == '(') {
60.                         *(++stackPtr) = c;  // Push '(' onto the stack
61.                     } else if (c == ')') {
62.                         while (stackPtr >= stack && *stackPtr != '(') {
63.                             *resultPtr++ = *(stackPtr--);  // Pop all until '('
64.                         }
65.                         stackPtr--; // Pop '(' from the stack
66.                     } else {
67.                         while (stackPtr != -1 && precedence(*stackPtr) >= precedence(c) &&
    *stackPtr != '(') {
68.                             *resultPtr++ = *(stackPtr--);
69.                         }
70.                         *(++stackPtr) = c;
71.                     }
72.                 }
73.
74.                 while (stackPtr >= stack) {
75.                     *resultPtr++ = *(stackPtr--);  // Pop all operators left in the stack
76.                 }
77.
78.                 *resultPtr = '\0';  // Null-terminate the string
79.                 printf("Postfix expression: %s\n", result);
80.                 break;
81.             }
82.             case 2: {
83.                 char stack[MAX_SIZE][MAX_SIZE]; // Stack to hold elements
84.                 int top = -1;
85.                 int len = strlen(expPtr);
86.
```

```
87.             for (int i = 0; i < len; i++) {
88.                 if (expPtr[i] >= 'a' && expPtr[i] <= 'z' || expPtr[i] >= 'A' && expPtr[i] <=
    'Z' || expPtr[i] >= '0' && expPtr[i] <= '9') {
89.                     // Push operand to stack
90.                     top++;
91.                     stack[top][0] = expPtr[i];
92.                     stack[top][1] = '\0';
93.                 } else {
94.                     // Operator encountered
95.                     char op1[MAX_SIZE], op2[MAX_SIZE];
96.                     strcpy(op2, stack[top--]);  // Pop two operands
97.                     strcpy(op1, stack[top--]);
98.                     char expr[MAX_SIZE] = {expPtr[i]}; // Start with operator
99.                     strcat(expr, op1);            // Concatenate first operand
100.                            strcat(expr, op2);            // Concatenate second operand
101.                            top++;
102.                            strcpy(stack[top], expr);      // Push result back to stack
103.                        }
104.                    }
105.
106.                    // The final prefix expression is on top of the stack
107.                    printf("Prefix expression: %s\n", stack[top]);
108.                    break;
109.                }
110.                case 3:
111.                    break;
112.                default:
113.                    printf("Invalid choice\n");
114.            }
115.        } while (choice != 3);
116.
117.        return 0;
118.    }
```

ALGORITHM FOR QUESTION 21 and 22

1] f

1. --Start--
2. Include necessary header files for standard input/output operations and string manipulation.
3. Define a constant `MAX_SIZE` to represent the maximum size for expressions and stack usage.
4. Declare a function `precedence` to return the precedence of operators.
5. Declare a function `isOperator` to check if a character is an operator.
6. In the `main` function, declare an integer variable `choice` for the user's menu choice and a character array `exp` to store the expression.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Infix to Postfix Conversion--
     - Initialize a character array `result` to store the postfix expression and variables to track the result index and stack index.
     - Iterate through the expression:
       - If the character is an operand, add it to the result.
       - If the character is an opening parenthesis, push it onto the stack.
       - If the character is a closing parenthesis, pop operators from the stack and add them to the result until an opening parenthesis is encountered, then pop the opening parenthesis.
       - If the character is an operator, pop operators from the stack with higher or equal precedence and add them to the result, then push the current operator onto the stack.
     - After iterating through the expression, pop any remaining operators from the stack and add them to the result.
     - Null-terminate the result string and display the postfix expression.
   - --Case 2: Postfix to Prefix Conversion--
     - Initialize a stack to hold elements of the expression.
     - Iterate through the expression:
       - If the character is an operand, push it onto the stack.
       - If the character is an operator, pop two operands from the stack, concatenate them with the operator to form a new expression, and push the new expression back onto the stack.
     - After iterating through the expression, the final prefix expression is on top of the stack. Display the prefix expression.
   - --Case 3: Exit the Program--
     - Display a message indicating that the program is exiting.
   - If the user's choice is not within the valid range, display an error message.
9. --End--

2] fp

1. --Start--
2. Include necessary header files for standard input/output operations and string manipulation.
3. Define a constant `MAX_SIZE` to represent the maximum size for expressions and stack usage.
4. Declare functions `precedence` to return the precedence of operators and `isOperator` to check if a character is an operator.
5. In the `main` function, declare integer variable `choice` for the user's menu choice, and character arrays `exp` and `result` for the expression input and output, respectively. Also, declare pointers `expPtr` and `resultPtr` to navigate through these arrays.
6. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer or not within the valid range, display an error message and continue.
   - If the choice is to exit, display a message and break the loop.
   - Prompt the user to enter an expression.
   - Use a `switch` statement to handle the user's choice:
     - --Case 1: Infix to Postfix--
       - Initialize a stack for operators and a pointer to navigate through it.
       - Iterate through the expression:
         - If the character is an operand, append it to the result.
         - If the character is an opening parenthesis, push it onto the stack.
         - If the character is a closing parenthesis, pop operators from the stack and append them to the result until an opening parenthesis is encountered.
         - If the character is an operator, pop operators from the stack with higher or equal precedence and append them to the result, then push the current operator onto the stack.
       - After the iteration, pop any remaining operators from the stack and append them to the result.
       - Display the postfix expression.
     - --Case 2: Postfix to Prefix--
       - Initialize a stack to hold elements of the expression.
       - Iterate through the expression:
         - If the character is an operand, push it onto the stack.
         - If the character is an operator, pop two operands from the stack, concatenate them with the operator, and push the result back onto the stack.
       - After the iteration, the final prefix expression is on top of the stack.
       - Display the prefix expression.
     - --Case 3: Exit the Program--
       - Display a message indicating that the program is exiting.
     - If the user's choice is not within the valid range, display an error message.
8. --End--

Q-23) C program for parenthesis checking [without use of functions and pointer]

```
#include <stdio.h>

#define MAX_SIZE 1000  // Maximum size of the stack

int main() {
    char expression[MAX_SIZE];  // Input expression array
    char stack[MAX_SIZE];       // Stack to hold open parentheses and brackets
    int top = -1;               // Stack pointer initialized to indicate empty stack
    int i;                      // Loop variable
    char ch;                    // Character variable to store each character from input

    printf("Enter an expression: ");
    scanf("%s", expression);

    for (i = 0; expression[i] != '\0'; i++) {
        ch = expression[i];

        // Check if the character is an opening brace, bracket, or parenthesis
        if (ch == '(' || ch == '[' || ch == '{') {
            if (top == MAX_SIZE - 1) {
                printf("Stack overflow\n");
                return 1;
            }
            // Push onto stack
            stack[++top] = ch;
        } else if (ch == ')' || ch == ']' || ch == '}') {
            // If stack is empty at this point, it's an unbalanced expression
            if (top == -1) {
                printf("Unbalanced expression\n");
                return 1;
            }
            // Check for matching parentheses
            if ((ch == ')' && stack[top] == '(') ||
                (ch == ']' && stack[top] == '[') ||
                (ch == '}' && stack[top] == '{')) {
                top--;  // Pop the stack
            } else {
                printf("Unbalanced expression\n");
                return 1;
            }
        }
    }

    // If stack is not empty at the end, it's an unbalanced expression
```

```
    if (top != -1) {
        printf("Unbalanced expression\n");
    } else {
        printf("Balanced expression\n");
    }

    return 0;
}
```

Q-24) C program for parenthesis checking [use of functions and pointer]

```
#include <stdio.h>

#define MAX_SIZE 1000  // Maximum size of the stack

int main() {
    char expression[MAX_SIZE];  // Input expression array
    char stack[MAX_SIZE];       // Stack to hold open parentheses and brackets
    int top = -1;               // Stack pointer initialized to indicate empty stack
    int i;                      // Loop variable
    char ch;                    // Character variable to store each character from input

    printf("Enter an expression: ");
    scanf("%s", expression);

    for (i = 0; expression[i] != '\0'; i++) {
        ch = expression[i];

        // Check if the character is an opening brace, bracket, or parenthesis
        if (ch == '(' || ch == '[' || ch == '{') {
            if (top == MAX_SIZE - 1) {
                printf("Stack overflow\n");
                return 1;
            }
            // Push onto stack
            stack[++top] = ch;
        } else if (ch == ')' || ch == ']' || ch == '}') {
            // If stack is empty at this point, it's an unbalanced expression
            if (top == -1) {
                printf("Unbalanced expression\n");
                return 1;
            }
            // Check for matching parentheses
            if ((ch == ')' && stack[top] == '(') ||
                (ch == ']' && stack[top] == '[') ||
                (ch == '}' && stack[top] == '{')) {
```

```
            top--;  // Pop the stack
        } else {
            printf("Unbalanced expression\n");
            return 1;
        }
    }
}

// If stack is not empty at the end, it's an unbalanced expression
if (top != -1) {
    printf("Unbalanced expression\n");
} else {
    printf("Balanced expression\n");
}

return 0;
}
```

Q-25) C program for parenthesis checking [use of pointers]

```
#include <stdio.h>

#define MAX_SIZE 1000  // Maximum size of the stack

int main() {
    char expression[MAX_SIZE];  // Input expression array
    char stack[MAX_SIZE];       // Stack to hold open parentheses and brackets
    char *top = stack - 1;      // Stack pointer initialized to just before the stack start
    int i;                      // Loop variable
    char *ch;                   // Pointer to traverse the expression

    printf("Enter an expression: ");
    scanf("%s", expression);

    for (ch = expression; *ch != '\0'; ch++) {
        // Check if the character is an opening brace, bracket, or parenthesis
        if (*ch == '(' || *ch == '[' || *ch == '{') {
            if (top == stack + MAX_SIZE - 1) {
                printf("Stack overflow\n");
                return 1;
            }
            // Push onto stack using pointer increment
            *(++top) = *ch;
        } else if (*ch == ')' || *ch == ']' || *ch == '}') {
            // If stack is empty at this point, it's an unbalanced expression
            if (top == stack - 1) {
```

```
            printf("Unbalanced expression\n");
            return 1;
        }
        // Check for matching parentheses using pointer to access top element
        if ((*ch == ')' && *top == '(') ||
            (*ch == ']' && *top == '[') ||
            (*ch == '}' && *top == '{')) {
            top--;  // Pop the stack by moving the pointer
        } else {
            printf("Unbalanced expression\n");
            return 1;
        }
    }
}

// If stack is not empty at the end, it's an unbalanced expression
if (top != stack - 1) {
    printf("Unbalanced expression\n");
} else {
    printf("Balanced expression\n");
}

return 0;
}
```

Q-26) C program for parenthesis checking [use of functions]

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

#define MAX_SIZE 1000  // Maximum size for the stack

char stack[MAX_SIZE];       // Stack to hold open parentheses and brackets
int top = -1;               // Stack pointer initialized to indicate empty stack

// Function to push a character onto the stack
void push(char ch) {
    if (top < MAX_SIZE - 1) {
        stack[++top] = ch;
    } else {
        printf("Stack overflow\n");
    }
}

// Function to pop the top character from the stack
char pop() {
```

```
    if (top != -1) {
        return stack[top--];
    } else {
        printf("Stack underflow\n");
        return '\0';  // Return a null character if underflow occurs
    }
}

// Function to check if the stack is empty
int isEmpty() {
    return top == -1;
}

int main() {
    char expression[MAX_SIZE];  // Input expression array
    char ch;                    // Character variable to store each character from input
    int i;                      // Loop variable

    printf("Enter an expression: ");
    scanf("%s", expression);

    for (i = 0; expression[i] != '\0'; i++) {
        ch = expression[i];

        if (ch == '(' || ch == '[' || ch == '{') {
            push(ch);
        } else if (ch == ')' || ch == ']' || ch == '}') {
            if (isEmpty()) {
                printf("Unbalanced expression\n");
                return 1;
            }
            char last = pop();
            if (!((ch == ')' && last == '(') ||
                  (ch == ']' && last == '[') ||
                  (ch == '}' && last == '{'))) {
                printf("Unbalanced expression\n");
                return 1;
            }
        }
    }

    if (!isEmpty()) {
        printf("Unbalanced expression\n");
    } else {
        printf("Balanced expression\n");
    }
```

```
66.
67.     return 0;
68. }
```

ALGORITHM FOR QUESTION 23-26

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Define a constant `MAX_SIZE` to represent the maximum size of the stack.
4. In the `main` function, declare an array `expression` of size `MAX_SIZE` to store the input expression.
5. Declare an array `stack` of size `MAX_SIZE` to hold open parentheses, brackets, and braces.
6. Initialize a variable `top` to `-1` to indicate an empty stack.
7. Declare loop variable `i` and character variable `ch` to store each character from the input expression.
8. Prompt the user to enter an expression.
9. Read the expression from the user using `scanf`.
10. Enter a loop that iterates through each character in the expression until the null terminator is encountered.
    - For each character:
        - If the character is an opening brace, bracket, or parenthesis:
            - Check if the stack is full. If so, display "Stack overflow" and exit the program.
            - Push the character onto the stack.
        - If the character is a closing brace, bracket, or parenthesis:
            - Check if the stack is empty. If so, display "Unbalanced expression" and exit the program.
            - Check if the top of the stack matches the closing character. If not, display "Unbalanced expression" and exit the program.
            - If the characters match, pop the stack.
11. After the loop, check if the stack is empty. If not, display "Unbalanced expression". Otherwise, display "Balanced expression".
12. --End--

2]fp

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Define a constant `MAX_SIZE` to represent the maximum size of the stack.
4. In the `main` function, declare an array `expression` of size `MAX_SIZE` to store the input expression.
5. Declare an array `stack` of size `MAX_SIZE` to hold open parentheses, brackets, and braces.
6. Initialize a variable `top` to `-1` to indicate an empty stack.

7. Declare loop variable `i` and character variable `ch` to store each character from the input expression.
8. Prompt the user to enter an expression and store it in `expression`.
9. Iterate through each character in `expression`:
   - If the character is an opening brace (`(`, `[`, or `{`), check if the stack is full. If so, print "Stack overflow" and exit the program. Otherwise, push the character onto the stack.
   - If the character is a closing brace (`)`, `]`, or `}`), check if the stack is empty. If so, print "Unbalanced expression" and exit the program. Otherwise, check if the top of the stack matches the closing brace. If it does, pop the stack. If it doesn't, print "Unbalanced expression" and exit the program.
10. After iterating through all characters in the expression, check if the stack is empty. If it is, print "Balanced expression". If it's not, print "Unbalanced expression".
11. --End--

3] p

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Define a constant `MAX_SIZE` to represent the maximum size of the stack.
4. In the `main` function, declare a character array `expression` to hold the input expression and a character array `stack` to act as the stack for holding open parentheses, brackets, and braces.
5. Initialize a pointer `top` to point just before the start of the stack, indicating that the stack is initially empty.
6. Declare integer variable `i` for loop control and a character pointer `ch` to traverse the expression.
7. Prompt the user to enter an expression and read it into the `expression` array.
8. Enter a loop to traverse the expression character by character.
   - If the current character is an opening brace, bracket, or parenthesis:
     - Check if the stack is full. If so, display "Stack overflow" and exit the program with an error code.
     - Push the character onto the stack by incrementing the `top` pointer and assigning the current character to the new top of the stack.
   - If the current character is a closing brace, bracket, or parenthesis:
     - Check if the stack is empty. If so, display "Unbalanced expression" and exit the program with an error code.
     - Check if the top of the stack matches the closing character. If not, display "Unbalanced expression" and exit the program with an error code.
     - If the characters match, pop the stack by decrementing the `top` pointer.
9. After traversing the entire expression, check if the stack is empty.
   - If the stack is not empty, it indicates an unbalanced expression. Display "Unbalanced expression" and exit the program.
   - If the stack is empty, it indicates a balanced expression. Display "Balanced expression".

10. --End--

4] f

1. --Start--
2. Include necessary header files for standard input/output operations and string manipulation.
3. Define a constant `MAX_SIZE` to represent the maximum size of the stack.
4. Declare a character array `stack` of size `MAX_SIZE` to hold open parentheses and brackets, and an integer `top` initialized to `-1` to indicate an empty stack.
5. Declare functions for pushing a character onto the stack, popping the top character from the stack, and checking if the stack is empty.
6. In the `main` function, declare a character array `expression` of size `MAX_SIZE` to store the input expression, and a character variable `ch` to store each character from the input.
7. Prompt the user to enter an expression and read it into the `expression` array.
8. Iterate through each character in the `expression` array:
   - If the character is an opening parenthesis (`(`, `[`, or `{`), call `push` to add it to the stack.
   - If the character is a closing parenthesis (`)`, `]`, or `}`), check if the stack is empty by calling `isEmpty`.
      - If the stack is empty, display "Unbalanced expression" and exit the program with a return code of `1`.
      - Otherwise, call `pop` to remove the top character from the stack and store it in `last`.
      - Compare the popped character with the current character to ensure they form a valid pair of parentheses, brackets, or braces.
         - If they do not form a valid pair, display "Unbalanced expression" and exit the program with a return code of `1`.
9. After processing all characters in the `expression` array, check if the stack is empty by calling `isEmpty`.
   - If the stack is not empty, display "Unbalanced expression".
   - Otherwise, display "Balanced expression".
10. --End--

Q-27) C program for Evaluation of postfix and prefix expressions[without use of functions and pointers]

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

#define MAX_SIZE 100  // Maximum size for the stack

int main() {
    char exp[MAX_SIZE];
    int stack[MAX_SIZE];
    int top = -1;
    int choice, i, op1, op2, result;

    do {
        printf("\t\tMain Menu\n"
               "1. Evaluate Prefix Expression\n"
               "2. Evaluate Postfix Expression\n"
               "3. EXIT\n"
               "Enter your choice (1-3): ");
        scanf("%d", &choice);

        if (choice < 1 || choice > 3) {
            printf("Invalid choice\n");
            continue;
        }

        if (choice == 3) {
            printf("Exiting...\n");
            break;
        }

        printf("Enter the expression: ");
        scanf("%s", exp);

        switch (choice) {
            case 1: // Evaluate Prefix
                for (i = strlen(exp) - 1; i >= 0; i--) {
                    if (isdigit(exp[i])) {
                        stack[++top] = exp[i] - '0';
                    } else {
                        op1 = stack[top--];
                        op2 = stack[top--];
                        switch (exp[i]) {
                            case '+': stack[++top] = op1 + op2; break;
```

```
                    case '-': stack[++top] = op1 - op2; break;
                    case '*': stack[++top] = op1 * op2; break;
                    case '/': stack[++top] = op1 / op2; break;
                }
            }
        }
        result = stack[top--];
        printf("Result of Prefix Evaluation: %d\n", result);
        break;

    case 2: // Evaluate Postfix
        for (i = 0; i < strlen(exp); i++) {
            if (isdigit(exp[i])) {
                stack[++top] = exp[i] - '0';
            } else {
                op2 = stack[top--];
                op1 = stack[top--];
                switch (exp[i]) {
                    case '+': stack[++top] = op1 + op2; break;
                    case '-': stack[++top] = op1 - op2; break;
                    case '*': stack[++top] = op1 * op2; break;
                    case '/': stack[++top] = op1 / op2; break;
                }
            }
        }
        result = stack[top--];
        printf("Result of Postfix Evaluation: %d\n", result);
        break;
    }
} while (choice != 3);

return 0;
}
```

Q-28) C program for Evaluation of postfix and prefix expressions[use of functions and pointers]

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_SIZE 100  // Maximum size for the stack

// Stack operations
void push(int stack[], int *top, int value) {
    stack[++(*top)] = value; // Push value onto stack
}
```

```
11.
12. int pop(int stack[], int *top) {
13.     return stack[(*top)--]; // Pop and return the top value from the stack
14. }
15.
16. // Evaluates a prefix expression
17. int evaluatePrefix(char *exp) {
18.     int stack[MAX_SIZE];
19.     int top = -1;
20.     int length = strlen(exp);
21.     int op1, op2;
22.
23.     // Loop from right to left
24.     for (int i = length - 1; i >= 0; i--) {
25.         if (isdigit(exp[i])) {
26.             push(stack, &top, exp[i] - '0');
27.         } else {
28.             op1 = pop(stack, &top);
29.             op2 = pop(stack, &top);
30.             switch (exp[i]) {
31.                 case '+': push(stack, &top, op1 + op2); break;
32.                 case '-': push(stack, &top, op1 - op2); break;
33.                 case '*': push(stack, &top, op1 * op2); break;
34.                 case '/': push(stack, &top, op1 / op2); break;
35.             }
36.         }
37.     }
38.     return pop(stack, &top); // Return the result from the stack
39. }
40.
41. // Evaluates a postfix expression
42. int evaluatePostfix(char *exp) {
43.     int stack[MAX_SIZE];
44.     int top = -1;
45.     int length = strlen(exp);
46.     int op1, op2;
47.
48.     // Loop from left to right
49.     for (int i = 0; i < length; i++) {
50.         if (isdigit(exp[i])) {
51.             push(stack, &top, exp[i] - '0');
52.         } else {
53.             op2 = pop(stack, &top);
54.             op1 = pop(stack, &top);
55.             switch (exp[i]) {
56.                 case '+': push(stack, &top, op1 + op2); break;
```

```
57.                 case '-': push(stack, &top, op1 - op2); break;
58.                 case '*': push(stack, &top, op1 * op2); break;
59.                 case '/': push(stack, &top, op1 / op2); break;
60.             }
61.         }
62.     }
63.     return pop(stack, &top); // Return the result from the stack
64. }
65.
66. int main() {
67.     int choice;
68.     char exp[MAX_SIZE];
69.
70.     do {
71.         printf("\t\tMain menu\n"
72.                "1. Evaluate Prefix Expression\n"
73.                "2. Evaluate Postfix Expression\n"
74.                "3. EXIT\n"
75.                "Enter your choice (1-3): ");
76.         scanf("%d", &choice);
77.         while (getchar() != '\n');  // Clear input buffer
78.
79.         if (choice < 1 || choice > 3) {
80.             printf("Invalid choice\n");
81.             continue;
82.         }
83.
84.         if (choice == 3) {
85.             printf("Exiting...\n");
86.             break;
87.         }
88.
89.         printf("Enter the expression: ");
90.         scanf("%s", exp);
91.         while (getchar() != '\n');
92.
93.         int result;
94.         switch (choice) {
95.             case 1:
96.                 result = evaluatePrefix(exp);
97.                 printf("Result of Prefix Evaluation: %d\n", result);
98.                 break;
99.             case 2:
100.                    result = evaluatePostfix(exp);
101.                    printf("Result of Postfix Evaluation: %d\n", result);
102.                    break;
```

```
            }
        } while (choice != 3);

        return 0;
    }
```

Q-29) C program for Evaluation of postfix and prefix expressions[use of functions]

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_SIZE 100  // Maximum size for the stack

int stack[MAX_SIZE];  // Global stack
int top = -1;         // Global stack top index

// Stack operations without pointers
void push(int value) {
    if (top < MAX_SIZE - 1) {
        stack[++top] = value;
    } else {
        printf("Stack overflow\n");
    }
}

int pop() {
    if (top != -1) {
        return stack[top--];
    } else {
        printf("Stack underflow\n");
        return -1;  // Return -1 if underflow occurs, which should be handled appropriately if possible
    }
}

// Evaluates a prefix expression
int evaluatePrefix(char exp[]) {
    int length = strlen(exp);
    int op1, op2;

    // Loop from right to left
    for (int i = length - 1; i >= 0; i--) {
        if (isdigit(exp[i])) {
            push(exp[i] - '0');
        } else {
```

```
            op1 = pop();
            op2 = pop();
            switch (exp[i]) {
                case '+': push(op1 + op2); break;
                case '-': push(op1 - op2); break;
                case '*': push(op1 * op2); break;
                case '/': push(op1 / op2); break;
            }
        }
    }
    return pop(); // Return the result from the stack
}

// Evaluates a postfix expression
int evaluatePostfix(char exp[]) {
    int length = strlen(exp);
    int op1, op2;

    // Loop from left to right
    for (int i = 0; i < length; i++) {
        if (isdigit(exp[i])) {
            push(exp[i] - '0');
        } else {
            op2 = pop();
            op1 = pop();
            switch (exp[i]) {
                case '+': push(op1 + op2); break;
                case '-': push(op1 - op2); break;
                case '*': push(op1 * op2); break;
                case '/': push(op1 / op2); break;
            }
        }
    }
    return pop(); // Return the result from the stack
}

int main() {
    int choice;
    char exp[MAX_SIZE];

    do {
        printf("\t\tMain menu\n"
               "1. Evaluate Prefix Expression\n"
               "2. Evaluate Postfix Expression\n"
               "3. EXIT\n"
               "Enter your choice (1-3): ");
```

```
84.     scanf("%d", &choice);
85.     while (getchar() != '\n');  // Clear input buffer
86.
87.     if (choice < 1 || choice > 3) {
88.         printf("Invalid choice\n");
89.         continue;
90.     }
91.
92.     if (choice == 3) {
93.         printf("Exiting...\n");
94.         break;
95.     }
96.
97.     printf("Enter the expression: ");
98.     scanf("%s", exp);
99.     while (getchar() != '\n');
100.
101.            int result;
102.            switch (choice) {
103.                case 1:
104.                    result = evaluatePrefix(exp);
105.                    printf("Result of Prefix Evaluation: %d\n", result);
106.                    break;
107.                case 2:
108.                    result = evaluatePostfix(exp);
109.                    printf("Result of Postfix Evaluation: %d\n", result);
110.                    break;
111.            }
112.        } while (choice != 3);
113.
114.        return 0;
115.    }
```

Q-30) C program for Evaluation of postfix and prefix expressions[use of pointers]

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <ctype.h>
4.
5.  #define MAX_SIZE 100  // Maximum size for the stack
6.
7.  int main() {
8.      char expression[MAX_SIZE];  // Input expression array
```

```
9.    int stack[MAX_SIZE];        // Stack to hold values for operations
10.   int *top = stack;           // Pointer to the top of the stack (start initially)
11.   int choice;
12.   char *ch;                   // Pointer to traverse the expression
13.
14.   do {
15.      printf("\t\tMain menu\n"
16.           "1. Evaluate Prefix Expression\n"
17.           "2. Evaluate Postfix Expression\n"
18.           "3. EXIT\n"
19.           "Enter your choice (1-3): ");
20.      scanf("%d", &choice);
21.      while (getchar() != '\n');  // Clear input buffer
22.
23.      if (choice < 1 || choice > 3) {
24.         printf("Invalid choice\n");
25.         continue;
26.      }
27.
28.      if (choice == 3) {
29.         printf("Exiting...\n");
30.         break;
31.      }
32.
33.      printf("Enter the expression: ");
34.      scanf("%s", expression);
35.      while (getchar() != '\n');
36.
37.      int op1, op2, result;
38.
39.      switch (choice) {
40.         case 1: {  // Evaluate Prefix Expression
41.            int length = strlen(expression);
42.            // Process from right to left
43.            for (ch = expression + length - 1; ch >= expression; ch--) {
44.               if (isdigit(*ch)) {
45.                  *top++ = *ch - '0';  // Push converted digit onto stack
46.               } else {
47.                  op2 = *--top;  // Pop first operand
48.                  op1 = *--top;  // Pop second operand
49.                  switch (*ch) {
50.                     case '+': *top++ = op1 + op2; break;
51.                     case '-': *top++ = op1 - op2; break;
52.                     case '*': *top++ = op1 * op2; break;
53.                     case '/': *top++ = op1 / op2; break;
54.                  }
```

```
55.                 }
56.             }
57.             result = *(top - 1);  // The result is at the top of the stack
58.             printf("Result of Prefix Evaluation: %d\n", result);
59.             break;
60.         }
61.         case 2: {  // Evaluate Postfix Expression
62.             for (ch = expression; *ch != '\0'; ch++) {
63.                 if (isdigit(*ch)) {
64.                     *top++ = *ch - '0';  // Push digit onto stack
65.                 } else {
66.                     op2 = *--top;  // Pop first operand
67.                     op1 = *--top;  // Pop second operand
68.                     switch (*ch) {
69.                         case '+': *top++ = op1 + op2; break;
70.                         case '-': *top++ = op1 - op2; break;
71.                         case '*': *top++ = op1 * op2; break;
72.                         case '/': *top++ = op1 / op2; break;
73.                     }
74.                 }
75.             }
76.             result = *(top - 1);  // The result is at the top of the stack
77.             printf("Result of Postfix Evaluation: %d\n", result);
78.             break;
79.         }
80.     }
81. } while (choice != 3);
82.
83. return 0;
84. }
```

ALGORITHM FOR QUESTION 27-30

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations, dynamic memory allocation, string manipulation, and character type checking.
3. Define a constant `MAX_SIZE` to represent the maximum size of the expression and the stack.
4. In the `main` function, declare variables for the expression (`exp`), the stack (`stack`), and the stack's top index (`top`).
5. Declare integer variables for the user's choice (`choice`), loop counter (`i`), operands (`op1`, `op2`), and the result (`result`).
6. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the choice is not within the valid range, display an error message and continue.
   - If the choice is to exit, display a message indicating that the program is exiting and break the loop.
   - Prompt the user to enter the expression.
7. Use a `switch` statement to handle the user's choice:
   - --Case 1: Evaluate Prefix Expression--
     - Iterate through the expression from the end to the beginning.
     - If the character is a digit, push it onto the stack.
     - If the character is an operator, pop two operands from the stack, perform the operation, and push the result back onto the stack.
     - After the loop, the result of the expression evaluation is at the top of the stack.
   - --Case 2: Evaluate Postfix Expression--
     - Iterate through the expression from the beginning to the end.
     - If the character is a digit, push it onto the stack.
     - If the character is an operator, pop two operands from the stack, perform the operation, and push the result back onto the stack.
     - After the loop, the result of the expression evaluation is at the top of the stack.
8. Display the result of the expression evaluation based on the user's choice.
9. --End--

2] fp

1. --Start--
2. Include necessary header files for standard input/output operations, string manipulation, and character type checking.
3. Define a constant `MAX_SIZE` to represent the maximum size of the stack and the expression.
4. Declare functions for stack operations:

- `push` to add an element to the top of the stack.
- `pop` to remove and return the top element from the stack.

5. Declare functions for evaluating prefix and postfix expressions:
   - `evaluatePrefix` to evaluate a prefix expression.
   - `evaluatePostfix` to evaluate a postfix expression.
6. In the `main` function, declare variables for user choice and the expression to be evaluated.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer or not within the valid range, display an error message and continue.
   - If the choice is to exit, display a message indicating that the program is exiting.
   - If the choice is to evaluate an expression, prompt the user to enter the expression.
   - Use a `switch` statement to handle the user's choice:
     - --Case 1: Evaluate Prefix Expression--
       - Call `evaluatePrefix` with the entered expression and display the result.
     - --Case 2: Evaluate Postfix Expression--
       - Call `evaluatePostfix` with the entered expression and display the result.
8. --End--

3] f

1. --Start--
2. Include necessary header files for standard input/output operations, string operations, and character type checking.
3. Define a constant `MAX_SIZE` to represent the maximum size of the stack and the expression.
4. Declare a global integer array `stack` of size `MAX_SIZE` and a global integer `top` to keep track of the top of the stack.
5. Declare functions for stack operations: `push` and `pop`.
6. Declare functions for evaluating prefix and postfix expressions: `evaluatePrefix` and `evaluatePostfix`.
7. In the `main` function, declare integer variables `choice` for the user's menu choice and `exp` for the expression to be evaluated.
8. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer or not within the valid range, display an error message and continue.
   - If the choice is to exit, display a message indicating that the program is exiting.
   - Prompt the user to enter the expression to be evaluated.
   - Use a `switch` statement to handle the user's choice:

- --Case 1: Evaluate Prefix Expression--
  - Call `evaluatePrefix` with the entered expression and display the result.
- --Case 2: Evaluate Postfix Expression--
  - Call `evaluatePostfix` with the entered expression and display the result.

9. --End--

Logic of User-Defined Functions

- --push(int value)--: This function is used to add an element to the top of the stack. It checks if the stack is not full before adding the element. If the stack is full, it prints "Stack overflow" and does not add the element.
- --pop()--: This function is used to remove and return the top element of the stack. It checks if the stack is not empty before removing the element. If the stack is empty, it prints "Stack underflow" and returns `-1`.
- --evaluatePrefix(char exp[])--: This function evaluates a prefix expression. It iterates through the expression from right to left. If the character is a digit, it pushes it onto the stack. If the character is an operator, it pops two operands from the stack, performs the operation, and pushes the result back onto the stack. Finally, it returns the result from the stack.
- --evaluatePostfix(char exp[])--: This function evaluates a postfix expression. It iterates through the expression from left to right. If the character is a digit, it pushes it onto the stack. If the character is an operator, it pops two operands from the stack, performs the operation, and pushes the result back onto the stack. Finally, it returns the result from the stack.

4] p

1. --Start--
2. Include necessary header files for standard input/output operations, string manipulation, and character type checking.
3. Define a constant `MAX_SIZE` to represent the maximum size of the expression and the stack.
4. In the `main` function, declare variables for the expression input, the stack to hold values for operations, a pointer to the top of the stack, and a variable for the user's menu choice.
5. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer or not within the valid range, display an error message and continue.
   - If the choice is to exit, display a message indicating that the program is exiting and break the loop.
6. Prompt the user to enter the expression.
7. Declare variables for the operands and the result of the operation.
8. Use a `switch` statement to handle the user's choice:

- --Case 1: Evaluate Prefix Expression--
    - Iterate through the expression from right to left.
    - If the character is a digit, convert it to an integer and push it onto the stack.
    - If the character is an operator, pop the top two elements from the stack, perform the operation, and push the result back onto the stack.
    - After processing the entire expression, the result is at the top of the stack. Display the result.
- --Case 2: Evaluate Postfix Expression--
    - Iterate through the expression from left to right.
    - If the character is a digit, convert it to an integer and push it onto the stack.
    - If the character is an operator, pop the top two elements from the stack, perform the operation, and push the result back onto the stack.
    - After processing the entire expression, the result is at the top of the stack. Display the result.

9. --End--

Q-31) C program to perform operations in queue using array[use of functions]

```
#include <stdio.h>
#define MAX_SIZE 1001

int addElement(int);
void showElements(int, int);
int removeElement(int, int);
void peekElement(int);

int queueArray[MAX_SIZE];
int frontIndex = -1;
int rearIndex = -1;

int peek() {
    if (frontIndex == -1 && rearIndex == -1) {
        printf("Queue is Empty\n");
    } else {
        printf("%d\n", queueArray[frontIndex]);
    }
    return 0;
}

int main() {
    int choice;

    do {
        printf("\t\tMain Menu for Opertions on queue\n"
               "\n1. Insert Element.\n"
               "2. Display the elements\n"
               "3. Delete the element\n"
               "4. Peek the element\n"
               "5. EXIT\n"
               "\nEnter your choice (1-5): ");
        scanf("%d", &choice);

        if (choice > 5 || choice < 1) {
            printf("Invalid choice\n");
        }

        switch (choice) {
            case 1: {
                int value;
                printf("Enter the value: ");
                scanf("%d", &value);
                addElement(value);
```

```
            break;
        }
        case 2: {
            showElements(frontIndex, rearIndex);
            break;
        }
        case 3: {
            removeElement(frontIndex, rearIndex);
            break;
        }
        case 4: {
            peek();
            break;
        }
        case 5: {
            printf("Or Bhai aa gye Sawad DSA ke\n");
            break;
        }
    }
} while (choice < 5);

return 0;
}

int addElement(int value) {
    if (rearIndex == MAX_SIZE - 1) {
        printf("Queue is FULL\n");
    } else if (frontIndex == -1 && rearIndex == -1) {
        frontIndex = rearIndex = 0;
        queueArray[rearIndex] = value;
    } else {
        rearIndex++;
        queueArray[rearIndex] = value;
    }
    return 0;
}

void showElements(int front, int rear) {
    if (front == -1 && rear == -1) {
        printf("Queue is Empty(just like your brain rn\n");
    } else {
        for (int i = front; i <= rear; i++) {
            printf("%d\n", queueArray[i]);
        }
    }
}
```

```
91.
92. int removeElement(int front, int rear) {
93.     if (front == -1 && rear == -1) {
94.         printf("Queue is Empty\n");
95.     } else if (front == rear) {
96.         printf("Deleted Element is : %d\n", queueArray[front]);
97.         front = rear = -1;
98.     } else {
99.         printf("Deleted Element is : %d\n", queueArray[front]);
100.                front++;
101.            }
102.            return 0;
103.        }
```

Q-32) C program to perform operations in queue using array[use of pointers]

```
1.  #include <stdio.h>
2.
3.  int main() {
4.      int choice, size, value;
5.      printf("Enter the size of the queue: ");
6.      scanf("%d", &size);
7.
8.      int queueArray[size];
9.      int* front = &queueArray[0];
10.     int* rear = &queueArray[0];
11.
12.     do {
13.         printf("\n1. Insert the element in the queue.\n"
14.                "2. Display the elements\n"
15.                "3. Delete the element from the Queue\n"
16.                "4. Peek the element\n"
17.                "5. EXIT\n"
18.                "\nEnter your choice (1-5): ");
19.         scanf("%d", &choice);
20.
21.         if (choice > 5 || choice < 1) {
22.             printf("Invalid choice\n");
23.         }
24.
25.         switch (choice) {
26.             case 1: {
27.                 printf("Enter the value: ");
28.                 scanf("%d", &value);
29.
```

```
            // Enqueue logic
            if (rear - queueArray == size - 1) {
                printf("Queue is FULL\n");
            } else if (front == &queueArray[0] && rear == &queueArray[0]) {
                *front = *rear = value;
            } else {
                rear++;
                *rear = value;
            }
            break;
        }
        case 2: {
            // Display Queue logic
            if (front == &queueArray[0] && rear == &queueArray[0]) {
                printf("Queue is Empty(like your brain rn)\n");
            } else {
                int* temp = front;
                while (temp <= rear) {
                    printf("%d\n", *temp);
                    temp++;
                }
            }
            break;
        }
        case 3: {
            // Dequeue logic
            if (front == &queueArray[0] && rear == &queueArray[0]) {
                printf("Queue is Empty\n");
            } else if (front == rear) {
                printf("Deleted Element is : %d\n", *front);
                front = rear = &queueArray[0];
            } else {
                printf("Deleted Element is : %d\n", *front);
                front++;
            }
            break;
        }
        case 4: {
            // Peek logic
            if (front == &queueArray[0] && rear == &queueArray[0]) {
                printf("Queue is Empty\n");
            } else {
                printf("%d\n", *front);
            }
            break;
        }
```

```
            case 5: {
                printf("Exit....\n");
                break;
            }
        }
    } while (choice < 5);

    return 0;
}
```

Q-33) C program to perform operations in queue using array[use of functions and pointers]

```
#include <stdio.h>

int main() {
    int choice, size, value;
    printf("Enter the size of the queue: ");
    scanf("%d", &size);

    int queueArray[size];
    int front = -1;
    int rear = -1;

    void enqueueElement(int*, int*, int*, int);
    void displayQueue(int*, int, int);
    void dequeueElement(int*, int*);
    void peekQueue(int*, int);

    do {
        printf("\n1. Insert the element in the queue.\n"
               "2. Display the elements\n"
               "3. Delete the element from the Queue\n"
               "4. Peek the element\n"
               "5. EXIT\n"
               "\nEnter your choice (1-5): ");
        scanf("%d", &choice);

        if (choice > 5 || choice < 1) {
            printf("Invalid choice\n");
        }
```

```
29.
30.         switch (choice) {
31.             case 1: {
32.                 printf("Enter the value: ");
33.                 scanf("%d", &value);
34.                 enqueueElement(queueArray, &front, &rear, size);
35.                 break;
36.             }
37.             case 2: {
38.                 displayQueue(queueArray, front, rear);
39.                 break;
40.             }
41.             case 3: {
42.                 dequeueElement(&front, &rear);
43.                 break;
44.             }
45.             case 4: {
46.                 peekQueue(queueArray, front);
47.                 break;
48.             }
49.             case 5: {
50.                 printf("THANK YOU FOR USING\n");
51.                 break;
52.             }
53.         }
54.     } while (choice < 5);
55.
56.     return 0;
57. }
58.
59. void dequeueElement(int* frontPtr, int* rearPtr) {
60.     if (*frontPtr == -1 && *rearPtr == -1) {
61.         printf("Queue is Empty\n");
62.     } else if (*frontPtr == *rearPtr) {
63.         printf("Deleted Element is : %d\n", *frontPtr);
64.         *frontPtr = *rearPtr = -1;
65.     } else {
66.         printf("Deleted Element is : %d\n", *frontPtr);
67.         (*frontPtr)++;
68.     }
69. }
70.
71. void peekQueue(int* queueArray, int front) {
72.     if (front == -1) {
73.         printf("Queue is Empty\n");
74.     } else {
```

```
        printf("%d\n", queueArray[front]);
    }
}

void enqueueElement(int* queueArray, int* frontPtr, int* rearPtr, int size) {
    int value;
    printf("Enter the value: ");
    scanf("%d", &value);

    if (*rearPtr == size - 1) {
        printf("Queue is FULL\n");
    } else if (*frontPtr == -1 && *rearPtr == -1) {
        *frontPtr = *rearPtr = 0;
        queueArray[*rearPtr] = value;
    } else {
        ++(*rearPtr);
        queueArray[*rearPtr] = value;
    }
}

void displayQueue(int* queueArray, int front, int rear) {
    if (front == -1 && rear == -1) {
        printf("Queue is Empty\n");
    } else {
        for (int i = front; i <= rear; i++) {
                    printf("%d\n", queueArray[i]);
                }
            }
        }
```

Q-34) C program to perform operations in queue using array[without use of functions and pointers]

```
// c program for  implementation of queue in array without functions and pointers

#include <stdio.h>
#include <stdlib.h>
int main() {
    int choice, size, value, front = -1, rear = -1;
    printf("Enter the size of the queue: ");
    scanf("%d", &size);

    int queueArray[size];

    do {
```

```
13.     printf("\n1. Insert the element in the queue.\n"
14.         "3. Delete the element from the Queue\n"
15.         "4. Peek the element\n"
16.         "5. EXIT\n"
17.         "\nEnter your choice (1-5): ");
18.     scanf("%d", &choice);
19.
20.     if (choice > 5 || choice < 1) {
21.        printf("Invalid choice\n");
22.     }
23.
24.     switch (choice) {
25.        case 1: {
26.           printf("Enter the value: ");
27.           scanf("%d", &value);
28.
29.           // Enqueue logic
30.           if (rear == size - 1) {
31.              printf("Queue is FULL\n");
32.           } else if (front == -1 && rear == -1) {
33.              front = rear = 0;
34.              queueArray[rear] = value;
35.           } else {
36.              rear++;
37.              queueArray[rear] = value;
38.           }
39.           break;
40.        }
41.        case 2: {
42.           // Display Queue logic
43.           if (front == -1 && rear == -1) {
44.              printf("Queue is Empty(like your brain rn)\n");
45.           } else {
46.              for (int i = front; i <= rear; ++i) {
47.                 printf("%d\n", queueArray[i]);
48.              }
49.           }
50.           break;
51.        }
52.        case 3: {
53.           // Dequeue logic
54.           int* removed;
55.           if (front == -1 && rear == -1) {
56.              printf("Queue is Empty\n");
57.           } else if (front == rear) {
58.              printf("Deleted Element is : %d\n", queueArray[front]);
```

```
                front = rear = -1;
            } else {
                printf("Deleted Element is : %d\n", queueArray[front]);
                removed = (int *) front;
                front++;
                free(removed);
            }
            break;
        }
        case 4: {
            // Peek logic
            if (front == -1 && rear == -1) {
                printf("Queue is Empty\n");
            } else {
                printf("%d\n", queueArray[front]);
            }
            break;
        }
        case 5: {
            printf("Or aa gaye sawad DSA ke saath \n");
            break;
        }
    }
} while (choice < 5);

return 0;
}
```

Q-35) C program to perform operations in queue using LL[use of functions and pointers]

```
1. //c program for implementation of queue in Linked list using functions and pointers
2.
3. #include <stdio.h>
4. #include <stdlib.h>
5.
6. struct QueueNode {
7.     int value;
8.     struct QueueNode* next;
9. };
10.
11. struct QueueNode* head = NULL;
12. struct QueueNode* tail = NULL;
13. struct QueueNode* newNode, * temp;
14.
15. int enqueueElement(int);
```

```
16. void displayQueue();
17. int dequeueElement();
18. void peekQueue();
19.
20. int main() {
21.     int choice;
22.     do {
23.         printf("\n1. Insert the element in the Queue.\n"
24.                "2. Display\n"
25.                "3. Delete the element from Queue\n"
26.                "4. Peek the element\n"
27.                "5. EXIT\n"
28.                "\nEnter your choice (1-5): ");
29.         scanf("%d", &choice);
30.
31.         if (choice > 5 || choice < 0) {
32.             printf("Invalid choice\n");
33.         }
34.
35.         switch (choice) {
36.             case 1: {
37.                 int x;
38.                 printf("Enter the data: ");
39.                 scanf("%d", &x);
40.                 enqueueElement(x);
41.                 break;
42.             }
43.             case 2: {
44.                 displayQueue();
45.                 break;
46.             }
47.             case 3: {
48.                 dequeueElement();
49.                 break;
50.             }
51.             case 4: {
52.                 peekQueue();
53.                 break;
54.             }
55.             case 5: {
56.                 printf("THANKYOU FOR USING");
57.                 break;
58.             }
59.         }
60.     } while (choice < 5);
61.
```

```
62.     return 0;
63. }
64.
65. int dequeueElement() {
66.     temp = head;
67.     if (head == NULL && tail == NULL) {
68.         printf("Queue is Empty\n");
69.     } else {
70.         printf("Deleted element %d\n", head->value);
71.         head = head->next;
72.         free(temp);
73.     }
74.     return 0;
75. }
76.
77. void peekQueue() {
78.     if (head == NULL && tail == NULL) {
79.         printf("Queue is Empty\n");
80.     } else {
81.         printf("%d\n", head->value);
82.     }
83. }
84.
85. int enqueueElement(int x) {
86.     newNode = (struct QueueNode*)malloc(sizeof(struct QueueNode));
87.     newNode->value = x;
88.     newNode->next = NULL;
89.
90.     if (head == NULL && tail == NULL) {
91.         head = tail = newNode;
92.     } else {
93.         tail->next = newNode;
94.         tail = newNode;
95.     }
96.     return 0;
97. }
98.
99. void displayQueue() {
100.             if (head == NULL && tail == NULL) {
101.                 printf("Queue is Empty\n");
102.             } else {
103.                 temp = head;
104.                 while (temp != NULL) {
105.                     printf("%d\n", temp->value);
106.                     temp = temp->next;
107.                 }
```

```
        }
    }
```

Q-36) C program to perform operations in queue using LL[use of pointers]

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

struct QueueNode {
    int value;
    struct QueueNode* next;
};

struct QueueNode* head = NULL;
struct QueueNode* tail = NULL;

int main() {
    int choice, x;
    do {
        printf("\n1. Insert the element in the Queue.\n"
               "2. Display\n"
               "3. Delete the element from Queue\n"
               "4. Peek the element\n"
               "5. EXIT\n"
               "\nEnter your choice (1-5): ");
        if (scanf("%d", &choice) != 1) {
            printf("Invalid input. Please enter a number.\n");
            while (getchar() != '\n'); // Clear the input buffer
            continue;
        }

        if (choice < 1 || choice > 5) {
            printf("Invalid choice\n");
            continue;
        }

        switch (choice) {
            case 1: {
                printf("Enter the data: ");
                scanf("%d", &x);
                struct QueueNode* newNode = (struct QueueNode*)malloc(sizeof(struct QueueNode));
                if (newNode == NULL) {
                    printf("Memory allocation failed.\n");
                    continue;
                }
```

```
41.             newNode->value = x;
42.             newNode->next = NULL;
43.
44.             if (head == NULL) {
45.                 head = tail = newNode;
46.             } else {
47.                 tail->next = newNode;
48.                 tail = newNode;
49.             }
50.             break;
51.         }
52.         case 2: {
53.             if (head == NULL) {
54.                 printf("Queue is Empty\n");
55.             } else {
56.                 struct QueueNode* temp = head;
57.                 while (temp != NULL) {
58.                     printf("%d\n", temp->value);
59.                     temp = temp->next;
60.                 }
61.             }
62.             break;
63.         }
64.         case 3: {
65.             if (head == NULL) {
66.                 printf("Queue is Empty\n");
67.             } else {
68.                 struct QueueNode* temp = head;
69.                 printf("Deleted element %d\n", head->value);
70.                 head = head->next;
71.                 if (head == NULL) tail = NULL; // Update tail if last element is removed
72.                 free(temp);
73.             }
74.             break;
75.         }
76.         case 4: {
77.             if (head == NULL) {
78.                 printf("Queue is Empty\n");
79.             } else {
80.                 printf("%d\n", head->value);
81.             }
82.             break;
83.         }
84.         case 5:
85.             printf("THANK YOU FOR USING\n");
86.             break;
```

```
87.         }
88.     } while (choice != 5);
89.
90.     // Clean up remaining nodes to prevent memory leaks
91.     while (head != NULL) {
92.         struct QueueNode* temp = head;
93.         head = head->next;
94.         free(temp);
95.     }
96.
97.     return 0;
98. }
```

ALGORITHM FOR QUESTION 31-36

1] f,array

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Define a constant `MAX_SIZE` to represent the maximum size of the queue.
4. Declare global variables for the queue array `queueArray` of size `MAX_SIZE`, and two integers `frontIndex` and `rearIndex` to keep track of the front and rear of the queue.
5. Declare functions for adding an element to the queue (`addElement`), displaying elements in the queue (`showElements`), removing an element from the queue (`removeElement`), and peeking at the front element of the queue (`peekElement`).
6. In the `main` function, declare an integer variable `choice` to store the user's menu choice.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the choice is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Insert Element--
     - Prompt the user to enter a value.
     - Call `addElement` with the entered value to add it to the queue.
   - --Case 2: Display the Elements--
     - Call `showElements` with `frontIndex` and `rearIndex` to display the elements in the queue.
   - --Case 3: Delete the Element--
     - Call `removeElement` with `frontIndex` and `rearIndex` to remove the front element from the queue.
   - --Case 4: Peek the Element--
     - Call `peekElement` to display the front element of the queue.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
9. --End--

Function Logic:

- --addElement(int value)--:
  - Check if the queue is full by comparing `rearIndex` with `MAX_SIZE - 1`.
  - If the queue is empty (both `frontIndex` and `rearIndex` are `-1`), initialize `frontIndex` and `rearIndex` to `0` and add the value at `rearIndex`.
  - Otherwise, increment `rearIndex` and add the value at the new `rearIndex`.

- --showElements(int front, int rear)--:
  - Check if the queue is empty by comparing `front` and `rear` with `-1`.
  - If the queue is not empty, iterate from `front` to `rear` and print each element.

- --removeElement(int front, int rear)--:
  - Check if the queue is empty by comparing `front` and `rear` with `-1`.
  - If the queue has only one element (`front` equals `rear`), print the front element and reset `front` and `rear` to `-1`.
  - Otherwise, print the front element and increment `front`.

- --peekElement()--:
  - Check if the queue is empty by comparing `frontIndex` and `rearIndex` with `-1`.
  - If the queue is not empty, print the front element.

2]p, array

1. --Start--
2. Include necessary header files for standard input/output operations.
3. In the `main` function, declare integer variables `choice`, `size`, and `value` to store the user's menu choice, the size of the queue, and the value to be inserted or deleted.
4. Prompt the user to enter the size of the queue and store it in `size`.
5. Declare an integer array `queueArray` of size `size` to represent the queue.
6. Initialize pointers `front` and `rear` to point to the first element of `queueArray`.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not within the valid range, display an error message.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Insert the element in the queue--
     - Prompt the user to enter a value.
     - If the queue is full, display a message indicating that the queue is full.
     - If the queue is empty, set both `front` and `rear` to point to the entered value.
     - Otherwise, increment `rear` and set its value to the entered value.
   - --Case 2: Display the elements--
     - If the queue is empty, display a message indicating that the queue is empty.
     - Otherwise, iterate through the queue from `front` to `rear` and print each element.
   - --Case 3: Delete the element from the Queue--
     - If the queue is empty, display a message indicating that the queue is empty.
     - If the queue has only one element, reset `front` and `rear` to point to the first element of `queueArray`.
     - Otherwise, increment `front` and display the deleted element.
   - --Case 4: Peek the element--
     - If the queue is empty, display a message indicating that the queue is empty.
     - Otherwise, display the element at `front`.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
9. --End--

3] fp, array

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare the `main` function where the program execution begins.
4. Inside `main`, declare integer variables `choice`, `size`, and `value` to store the user's menu choice, the size of the queue, and the value to be enqueued.
5. Prompt the user to enter the size of the queue and store it in `size`.
6. Declare an integer array `queueArray` of size `size` to represent the queue.
7. Initialize `front` and `rear` pointers to `-1` to indicate an empty queue.
8. Declare functions `enqueueElement`, `displayQueue`, `dequeueElement`, and `peekQueue` for queue operations.
9. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the choice is not within the valid range, display an error message and continue.
10. Use a `switch` statement to handle the user's choice:
   - --Case 1: Insert an Element in the Queue--
     - Prompt the user to enter a value.
     - Call `enqueueElement` to add the value to the queue.
   - --Case 2: Display the Elements--
     - Call `displayQueue` to display the elements in the queue.
   - --Case 3: Delete an Element from the Queue--
     - Call `dequeueElement` to remove an element from the queue.
   - --Case 4: Peek the Element--
     - Call `peekQueue` to display the front element of the queue without removing it.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.
11. --End--

Function Descriptions:

- --enqueueElement--:
   - Takes the queue array, front and rear pointers, and the size of the queue as arguments.
   - Prompts the user to enter a value.
   - Checks if the queue is full. If so, displays a message indicating that the queue is full.
   - If the queue is empty, initializes the front and rear pointers to `0` and adds the value to the queue.
   - Otherwise, increments the rear pointer and adds the value to the queue.

- --displayQueue--:

- Takes the queue array, front, and rear pointers as arguments.
- Checks if the queue is empty. If so, displays a message indicating that the queue is empty.
- Otherwise, iterates through the queue from the front to the rear and prints each element.

- --dequeueElement--:
- Takes the front and rear pointers as arguments.
- Checks if the queue is empty. If so, displays a message indicating that the queue is empty.
- If the queue has only one element, deletes it and resets the front and rear pointers to `-1`.
- Otherwise, deletes the front element and increments the front pointer.

- --peekQueue--:
- Takes the queue array and the front pointer as arguments.
- Checks if the queue is empty. If so, displays a message indicating that the queue is empty.
- Otherwise, prints the front element of the queue.

4] normal, array

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. In the `main` function, declare integer variables `choice`, `size`, `value`, `front`, and `rear` to manage the queue operations and state.
4. Prompt the user to enter the size of the queue and store it in `size`.
5. Declare an integer array `queueArray` of size `size` to represent the queue.
6. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not within the valid range, display an error message.
7. Use a `switch` statement to handle the user's choice:
   - --Case 1: Insert the element in the queue--
     - Prompt the user to enter the value to be inserted.
     - If the queue is full, display a message indicating that the queue is full.
     - If the queue is empty, set `front` and `rear` to `0` and insert the value at `rear`.
     - Otherwise, increment `rear` and insert the value at the new `rear` position.
   - --Case 2: Display the Queue--
     - If the queue is empty, display a message indicating that the queue is empty.
     - Otherwise, iterate through the queue from `front` to `rear` and print each element.

- --Case 3: Delete the element from the Queue--
  - If the queue is empty, display a message indicating that the queue is empty.
  - If the queue has only one element, display the deleted element and reset `front` and `rear` to `-1`.
  - Otherwise, display the deleted element at `front`, increment `front`, and free the memory pointed to by `removed` (note: this step is incorrect as `removed` is not a pointer to dynamically allocated memory but rather an integer).
- --Case 4: Peek the element--
  - If the queue is empty, display a message indicating that the queue is empty.
  - Otherwise, display the element at `front`.
- --Case 5: Exit the Program--
  - Display a message indicating that the program is exiting.

8. --End--

5] fp, LL

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `QueueNode` for the queue nodes, containing an integer `value` and a pointer `next` to the next node in the queue.
4. Declare global pointers `head` and `tail` to keep track of the beginning and end of the queue, and `newNode` and `temp` for temporary node references.
5. Declare functions for enqueuing an element, dequeuing an element, peeking at the queue, and displaying the queue.
6. In the `main` function, declare an integer variable `choice` to store the user's menu choice.
7. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the choice is not within the valid range, display an error message and continue.
8. Use a `switch` statement to handle the user's choice:
   - --Case 1: Insert an Element--
     - Prompt the user to enter the data.
     - Call `enqueueElement` with the entered data.
   - --Case 2: Display the Queue--
     - Call `displayQueue` to display the contents of the queue.
   - --Case 3: Delete an Element--
     - Call `dequeueElement` to remove the front element of the queue.
   - --Case 4: Peek at the Queue--
     - Call `peekQueue` to display the front element of the queue without removing it.
   - --Case 5: Exit the Program--
     - Display a message indicating that the program is exiting.

9. --End--

Function Logic:

- --enqueueElement(int x)--:
  - Allocate memory for a new node.
  - Set the `value` of the new node to `x`.
  - If the queue is empty (`head` and `tail` are `NULL`), set both `head` and `tail` to the new node.
  - Otherwise, set the `next` of the current `tail` to the new node and update `tail` to the new node.

- --dequeueElement()--:
  - If the queue is empty, display a message indicating that the queue is empty.
  - Otherwise, display the value of the front element, update `head` to the next node, and free the memory of the dequeued node.

- --peekQueue()--:
  - If the queue is empty, display a message indicating that the queue is empty.
  - Otherwise, display the value of the front element.

- --displayQueue()--:
  - If the queue is empty, display a message indicating that the queue is empty.
  - Otherwise, iterate through the queue from `head` to `tail`, displaying the value of each node.

6] p, LL

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Define a structure `QueueNode` for the queue nodes, containing an integer `value` and a pointer `next` to the next node in the queue.
4. Declare global pointers `head` and `tail` to the first and last nodes of the queue, initially set to `NULL`.
5. In the `main` function, declare integer variables `choice` and `x` to store the user's menu choice and the value to be inserted into the queue.
6. Enter a `do-while` loop that continues until the user chooses to exit the program.
   - Display the main menu options to the user.
   - Prompt the user to enter their choice.
   - If the input is not an integer, display an error message, clear the input buffer, and continue.
   - If the choice is not within the valid range, display an error message and continue.

7. Use a `switch` statement to handle the user's choice:
   - --Case 1: Insert the element in the Queue--
      - Prompt the user to enter a value.
      - Allocate memory for a new `QueueNode` and check if the allocation was successful.
      - Assign the entered value to the new node's `value` and set its `next` pointer to `NULL`.
      - If the queue is empty, set both `head` and `tail` to the new node.
      - Otherwise, append the new node to the end of the queue by updating the `next` pointer of the current `tail` and then updating `tail` to the new node.
   - --Case 2: Display--
      - If the queue is empty, display a message indicating that the queue is empty.
      - Otherwise, iterate through the queue from `head` to `tail`, printing each node's `value`.
   - --Case 3: Delete the element from Queue--
      - If the queue is empty, display a message indicating that the queue is empty.
      - Otherwise, remove the node at the front of the queue by updating `head` to the next node and freeing the memory of the removed node.
      - If the queue becomes empty, also update `tail` to `NULL`.
   - --Case 4: Peek the element--
      - If the queue is empty, display a message indicating that the queue is empty.
      - Otherwise, print the value of the node at the front of the queue.
   - --Case 5: EXIT--
      - Display a message indicating that the program is exiting.
8. After the loop, ensure to free any remaining nodes in the queue to prevent memory leaks.
9. --End--

Q-37) C program to perform bubble sort [without using functions and pointers]

```
#include <stdio.h>

int main() {
    int arr[100], size;

    // Input the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    // Input array elements
    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    // Display the unsorted array
    printf("Unsorted array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Bubble sort logic
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    // Display the sorted array
    printf("Sorted array:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Q-38) C program to perform bubble sort [using functions]

```
#include <stdio.h>

// Function to input array elements
void inputArray(int arr[], int size) {
    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
}

// Function to display the array
void displayArray(int arr[], int size) {
    printf("Array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Function to perform bubble sort
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[100], size;

```

```
    // Input the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    // Input array elements
    inputArray(arr, size);

    // Display the unsorted array
    displayArray(arr, size);

    // Perform bubble sort
    bubbleSort(arr, size);

    // Display the sorted array
    displayArray(arr, size);

    return 0;
}
```

Q-39) C program to perform bubble sort [using pointers]

```
// c program for bubble sort using pointers only

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

int main() {
    int *arr, size;

    // Input the size of the array
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    // Allocate memory for the array dynamically
    arr = (int*)malloc(size * sizeof(int));

    // Input array elements
    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", arr + i);
    }

    // Display the unsorted array
    printf("Unsorted array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", *(arr + i));
```

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

    // Bubble sort logic
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (*(arr + j) > *(arr + j + 1)) {
                // Swap *(arr + j) and *(arr + j + 1)
                int temp = *(arr + j);
                *(arr + j) = *(arr + j + 1);
                *(arr + j + 1) = temp;
            }
        }
    }

    // Display the sorted array
    printf("Sorted array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", *(arr + i));
    }
    printf("\n");

    // Free the dynamically allocated memory
    free(arr);

    return 0;
}
```

Q-40) C program to perform bubble sort [using functions and pointers]

```
#include <stdio.h>

void inputArray(int *arr, int *size);
void displayArray(int *arr, int size);
void bubbleSort(int *arr, int size);

int main() {
    int arr[100], size;

    inputArray(arr, &size);

    printf("Unsorted array:\n");
    displayArray(arr, size);

    bubbleSort(arr, size);

```

```
17.     printf("Sorted array:\n");
18.     displayArray(arr, size);
19.
20.     return 0;
21. }
22.
23. void inputArray(int *arr, int *size) {
24.     printf("Enter the size of the array: ");
25.     scanf("%d", size);
26.
27.     printf("Enter %d elements:\n", *size);
28.     for (int i = 0; i < *size; i++) {
29.         scanf("%d", arr + i);
30.     }
31. }
32.
33. void displayArray(int *arr, int size) {
34.     for (int i = 0; i < size; i++) {
35.         printf("%d ", *(arr + i));
36.     }
37.     printf("\n");
38. }
39.
40. void bubbleSort(int *arr, int size) {
41.     for (int i = 0; i < size - 1; i++) {
42.         for (int j = 0; j < size - i - 1; j++) {
43.             if (*(arr + j) > *(arr + j + 1)) {
44.                 // Swap *(arr + j) and *(arr + j + 1)
45.                 int temp = *(arr + j);
46.                 *(arr + j) = *(arr + j + 1);
47.                 *(arr + j + 1) = temp;
48.             }
49.         }
50.     }
51. }
```

ALGORITHM FOR QUESTION 37-40

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare functions for inputting array elements, displaying the array, and performing bubble sort.
4. In the `main` function, declare an integer array `arr` of size `100` and an integer variable `size` to store the size of the array.
5. Prompt the user to enter the size of the array and store it in `size`.
6. Call the `inputArray` function to input the array elements.
7. Call the `displayArray` function to display the unsorted array.
8. Call the `bubbleSort` function to sort the array using the bubble sort algorithm.
9. Call the `displayArray` function again to display the sorted array.
10. --End--

Function Descriptions

- --inputArray(int arr[], int size)--
  - This function takes an array `arr` and its size `size` as parameters.
  - It prompts the user to enter `size` number of elements for the array.
  - It uses a loop to read each element from the user and stores it in the array.

- --displayArray(int arr[], int size)--
  - This function takes an array `arr` and its size `size` as parameters.
  - It iterates through the array and prints each element followed by a space.
  - After printing all elements, it prints a newline character to end the line.

- --bubbleSort(int arr[], int size)--
  - This function takes an array `arr` and its size `size` as parameters.
  - It implements the bubble sort algorithm to sort the array in ascending order.
  - The outer loop iterates over each element of the array.
  - The inner loop compares adjacent elements and swaps them if they are in the wrong order.
  - This process continues until the entire array is sorted.

2] f

1. --Start--
2. Include necessary header files for standard input/output operations.

3. Declare functions for inputting array elements, displaying the array, and performing bubble sort.
4. In the `main` function, declare an integer array `arr` of size `100` and an integer `size` to store the size of the array.
5. Prompt the user to enter the size of the array.
6. Call `inputArray` to input the elements of the array.
7. Call `displayArray` to display the unsorted array.
8. Call `bubbleSort` to sort the array using the bubble sort algorithm.
9. Call `displayArray` again to display the sorted array.
10. --End--

Detailed Function Descriptions:

- --inputArray(int arr[], int size)--
  - This function takes an array `arr` and its size `size` as parameters.
  - It prompts the user to enter `size` number of elements.
  - It uses a `for` loop to read each element from the user and stores it in the array.

- --displayArray(int arr[], int size)--
  - This function takes an array `arr` and its size `size` as parameters.
  - It prints the elements of the array in a single line, separated by spaces.
  - It uses a `for` loop to iterate through each element of the array and print it.

- --bubbleSort(int arr[], int size)--
  - This function takes an array `arr` and its size `size` as parameters.
  - It implements the bubble sort algorithm to sort the array in ascending order.
  - It uses two nested `for` loops to compare each pair of adjacent elements and swap them if they are in the wrong order.
  - The outer loop runs `size - 1` times, and the inner loop runs `size - i - 1` times, where `i` is the current iteration of the outer loop.
  - The inner loop's condition ensures that the largest element is always moved to the end of the array in each iteration of the outer loop.

3] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. In the `main` function, declare a pointer to an integer `arr` and an integer `size` to store the size of the array.
4. Prompt the user to enter the size of the array and read it using `scanf`.
5. Allocate memory dynamically for the array using `malloc`, with the size being `size * sizeof(int)`.
6. Prompt the user to enter the elements of the array.

- Use a `for` loop to iterate through each element of the array.
- Use pointer arithmetic to access and store the input values in the dynamically allocated array.
7. Display the unsorted array to the user.
- Use a `for` loop to iterate through each element of the array.
- Use pointer arithmetic to access and print the elements of the array.
8. Implement the Bubble Sort algorithm using pointers.
- Use a nested `for` loop to iterate through the array.
- For each iteration, compare adjacent elements using pointer arithmetic.
- If the current element is greater than the next element, swap their values.
- Continue this process until the entire array is sorted.
9. Display the sorted array to the user.
- Use a `for` loop to iterate through each element of the array.
- Use pointer arithmetic to access and print the elements of the array.
10. Free the dynamically allocated memory using `free`.
11. Return `0` to indicate successful execution of the program.
12. --End--

4] fp

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare functions for inputting an array, displaying an array, and sorting an array using the Bubble Sort algorithm.
4. In the `main` function, declare an integer array `arr` of size `100` and an integer variable `size` to store the size of the array.
5. Call `inputArray` to input the array elements and size.
6. Display the unsorted array by calling `displayArray`.
7. Sort the array using the `bubbleSort` function.
8. Display the sorted array by calling `displayArray`.
9. --End--

Function Descriptions

- --inputArray(int *arr, int *size)--
  - Prompt the user to enter the size of the array.
  - Read the size from the user and store it in `size`.
  - Prompt the user to enter the elements of the array.
  - Read each element from the user and store it in the array.

- --displayArray(int *arr, int size)--
  - Iterate through each element of the array.
  - Print each element followed by a space.
  - Print a newline character after all elements have been printed.

- --bubbleSort(int *arr, int size)--
  - Iterate through each element of the array.
  - For each element, iterate through the remaining unsorted elements.
  - Compare each element with its adjacent element.
  - If the current element is greater than the adjacent element, swap them.
  - Continue this process until the array is sorted.

Q-41) C program to perform insertion sort [without using functions and pointers]

```
#include <stdio.h>

int main() {
    int n, i, j, key;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Using VLA for the array
    int arr[n];

    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("Array before Sorting: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    // Implementing insertion sort directly in main
    for (i = 1; i < n; i++) {
        key = arr[i]; // Take the value
        j = i;
        while (j > 0 && arr[j - 1] > key) {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = key; // Insert in the right place
    }

    printf("Array after Sorting: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Q-42) C program to perform insertion sort [using functions]

```
#include <stdio.h>

```

```
3.  // Function to sort an array using insertion sort
4.  void insertionSort(int arr[], int n) {
5.      int i, key, j;
6.      for (i = 1; i < n; i++) {
7.          key = arr[i];
8.          j = i - 1;
9.          while (j >= 0 && arr[j] > key) {
10.             arr[j + 1] = arr[j];
11.             j = j - 1;
12.         }
13.         arr[j + 1] = key;
14.     }
15. }
16.
17. // Function to print an array
18. void printArray(int arr[], int n) {
19.     for (int i = 0; i < n; i++)
20.         printf("%d ", arr[i]);
21.     printf("\n");
22. }
23.
24. int main() {
25.     int n, i;
26.
27.     printf("Enter the number of elements: ");
28.     scanf("%d", &n);
29.
30.     // Using VLA for the array
31.     int arr[n];
32.
33.     printf("Enter elements: ");
34.     for (i = 0; i < n; i++)
35.         scanf("%d", &arr[i]);
36.
37.     printf("Array before Sorting: ");
38.     printArray(arr, n);
39.
40.     // Call the insertion sort function
41.     insertionSort(arr, n);
42.
43.     printf("Array after Sorting: ");
44.     printArray(arr, n);
45.
46.     return 0;
47. }
```

Q-43) C program to perform insertion sort [using pointers]

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

int main() {
    int n, i, j, key;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory for the array
    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("Array before Sorting: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    // Implementing insertion sort directly in main
    for (i = 1; i < n; i++) {
        key = arr[i]; // Take the value
        j = i;
        while (j > 0 && arr[j - 1] > key) {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = key; // Insert in the right place
    }

    printf("Array after Sorting: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    // Free dynamically allocated memory
    free(arr);

    return 0;
```

```
46. }
```

Q-44) C program to perform insertion sort [using functions and pointers]

```
1.  #include <stdio.h>
2.  #include <stdlib.h> // For malloc and free
3.
4.  // Function to sort an array using insertion sort
5.  void insertionSort(int* arr, int n) {
6.      int i, key, j;
7.      for (i = 1; i < n; i++) {
8.          key = arr[i];
9.          j = i - 1;
10.         while (j >= 0 && arr[j] > key) {
11.             arr[j + 1] = arr[j];
12.             j = j - 1;
13.         }
14.         arr[j + 1] = key;
15.     }
16. }
17.
18. // Function to print an array
19. void printArray(int* arr, int n) {
20.     for (int i = 0; i < n; i++)
21.         printf("%d ", arr[i]);
22.     printf("\n");
23. }
24.
25. int main() {
26.     int n, i;
27.
28.     printf("Enter the number of elements: ");
29.     scanf("%d", &n);
30.
31.     // Dynamically allocate memory for the array
32.     int* arr = (int*)malloc(n * sizeof(int));
33.     if (arr == NULL) {
34.         printf("Memory allocation failed.\n");
35.         return 1;
36.     }
37.
38.     printf("Enter elements: ");
39.     for (i = 0; i < n; i++)
40.         scanf("%d", &arr[i]);
41.
42.     printf("Array before Sorting: ");
```

```
43.     printArray(arr, n);
44.
45.     // Call the insertion sort function
46.     insertionSort(arr, n);
47.
48.     printf("Array after Sorting: ");
49.     printArray(arr, n);
50.
51.     // Free dynamically allocated memory
52.     free(arr);
53.
54.     return 0;
55. }
56.
```

ALGORITHM FOR QUESTION 41-44

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations.
3. In the `main` function, declare integer variables `n`, `i`, `j`, and `key` to store the number of elements, loop counters, and the key element for insertion sort.
4. Prompt the user to enter the number of elements `n` and read it using `scanf`.
5. Declare an array `arr` of size `n` using Variable Length Array (VLA) to store the elements.
6. Prompt the user to enter the elements of the array and read them using a `for` loop.
7. Display the array before sorting to the user.
8. Implement the insertion sort algorithm directly in the `main` function:
   - Start a loop from the second element (index `1`) to the last element of the array.
   - For each element, store its value in `key`.
   - Initialize `j` to the current index `i`.
   - While `j` is greater than `0` and the element at index `j - 1` is greater than `key`, shift the element at index `j - 1` to index `j` and decrement `j`.
   - Insert `key` at the correct position found by `j` in the sorted part of the array.
9. Display the array after sorting to the user.
10. Return `0` to indicate successful execution.
11. --End--

2] f

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare a function `insertionSort` to sort an array using the insertion sort algorithm.
   - The function takes an integer array `arr` and its size `n` as parameters.
   - It iterates over the array starting from the second element (index 1).
   - For each element, it compares it with the elements before it and shifts them to the right until it finds the correct position for the current element.
   - It then inserts the current element at the found position.
4. Declare a function `printArray` to print the elements of an array.
   - The function takes an integer array `arr` and its size `n` as parameters.
   - It iterates over the array and prints each element followed by a space.
5. In the `main` function, declare integer variables `n` and `i` to store the number of elements in the array and the loop counter.
6. Prompt the user to enter the number of elements `n`.
7. Declare a variable-length array `arr` of size `n` to store the elements.
8. Prompt the user to enter the elements of the array.

9. Display the array before sorting by calling `printArray`.
10. Call `insertionSort` to sort the array.
11. Display the array after sorting by calling `printArray`.
12. --End--

3] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. In the `main` function, declare integer variables `n`, `i`, `j`, and `key` to store the number of elements, loop counters, and the key element for insertion sort.
4. Prompt the user to enter the number of elements `n`.
5. Dynamically allocate memory for an integer array `arr` of size `n` using `malloc`.
   - If memory allocation fails, display an error message and return `1` to indicate an error.
6. Prompt the user to enter the elements of the array.
7. Display the array before sorting.
8. Implement the insertion sort algorithm directly in the `main` function:
   - For each element in the array starting from the second element (`i = 1`):
     - Store the current element as `key`.
     - Compare `key` with the previous elements.
     - If `key` is smaller than the previous element, shift the previous element to the right.
     - Continue this process until `key` is in its correct position in the sorted part of the array.
     - Insert `key` in its correct position.
9. Display the array after sorting.
10. Free the dynamically allocated memory for the array using `free`.
11. Return `0` to indicate successful execution.
12. --End--

4] fp

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Declare a function `insertionSort` to sort an array using the insertion sort algorithm.
   - The function takes an integer array `arr` and its size `n` as parameters.
   - It iterates through the array starting from the second element.
   - For each element, it compares it with the elements before it and shifts them to the right until it finds the correct position for the current element.
   - It then inserts the current element at the found position.
4. Declare a function `printArray` to print the elements of an array.
   - The function takes an integer array `arr` and its size `n` as parameters.
   - It iterates through the array and prints each element followed by a space.

5. In the `main` function, declare integer variables `n` and `i` to store the number of elements in the array and the loop counter.
6. Prompt the user to enter the number of elements `n`.
7. Dynamically allocate memory for the array `arr` of size `n` using `malloc`.
   - Check if the memory allocation was successful. If not, display an error message and exit the program.
8. Prompt the user to enter the elements of the array.
9. Display the array before sorting using the `printArray` function.
10. Call the `insertionSort` function to sort the array.
11. Display the array after sorting using the `printArray` function.
12. Free the dynamically allocated memory for the array using `free`.
13. --End--

Q-45) C program to perform merge sort [without using functions and pointers]

```
#include <stdio.h>

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Using VLA for array size and elements
    int arr[n];

    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    // Merge Sort logic directly in main
    if (n > 1) {
        int m = n / 2;

        // Sort first and second halves
        for (i = 0; i < m; i++) {
            int L[m], R[n - m];
            int j, k = 0;

            // Copy data to temporary arrays
            for (j = 0; j < m; j++)
                L[j] = arr[j];
            for (j = m; j < n; j++)
                R[j - m] = arr[j];

            // Merge the temporary arrays back into arr[l..r]
            i = 0; // Initial index of first subarray
            j = 0; // Initial index of second subarray
            k = 0; // Initial index of merged subarray
            while (i < m && j < n - m) {
                if (L[i] <= R[j]) {
                    arr[k] = L[i];
                    i++;
                } else {
                    arr[k] = R[j];
                    j++;
                }
                k++;
            }
```

```
45.
46.             // Copy the remaining elements of L[], if there are any
47.             while (i < m) {
48.                 arr[k] = L[i];
49.                 i++;
50.                 k++;
51.             }
52.
53.             // Copy the remaining elements of R[], if there are any
54.             while (j < n - m) {
55.                 arr[k] = R[j];
56.                 j++;
57.                 k++;
58.             }
59.         }
60.     }
61.
62.     printf("\nSorted array is: \n");
63.     for (i = 0; i < n; i++)
64.         printf("%d ", arr[i]);
65.
66.     return 0;
67. }
```

Q-46) C program to perform merge sort [using functions and pointers]

```
1.  #include <stdio.h>
2.  #include <stdlib.h> // For malloc and free
3.
4.  // Function to merge two subarrays
5.  void merge(int* arr, int l, int m, int r) {
6.      int i, j, k;
7.      int n1 = m - l + 1;
8.      int n2 = r - m;
9.
10.     // Dynamically allocate memory for temporary arrays
11.     int* L = (int*)malloc(n1 * sizeof(int));
12.     int* R = (int*)malloc(n2 * sizeof(int));
13.     if (L == NULL || R == NULL) {
14.         printf("Memory allocation failed.\n");
15.         return;
16.     }
17.
18.     // Copy data to temporary arrays
```

```
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into arr[l..r]
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    // Free dynamically allocated memory
    free(L);
    free(R);
}

// Function to perform the merge sort
void mergeSort(int* arr, int l, int r) {
    if (l < r) {
        // Find the middle point
        int m = l + (r - l) / 2;

        // Sort first and second halves
```

```
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory for the array
    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    // Perform merge sort
    mergeSort(arr, 0, n - 1);

    printf("\nSorted array is: \n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    // Free dynamically allocated memory
    free(arr);

            return 0;
        }
```

Q-47) C program to perform merge sort [using functions]

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
```

```
5.      int n1 = m - l + 1;
6.      int n2 = r - m;
7.
8.      // Create temporary arrays
9.      int L[n1], R[n2];
10.
11.     // Copy data to temporary arrays
12.     for (i = 0; i < n1; i++)
13.        L[i] = arr[l + i];
14.     for (j = 0; j < n2; j++)
15.        R[j] = arr[m + 1 + j];
16.
17.     // Merge the temporary arrays back into arr[l..r]
18.     i = 0; // Initial index of first subarray
19.     j = 0; // Initial index of second subarray
20.     k = l; // Initial index of merged subarray
21.     while (i < n1 && j < n2) {
22.        if (L[i] <= R[j]) {
23.           arr[k] = L[i];
24.           i++;
25.        } else {
26.           arr[k] = R[j];
27.           j++;
28.        }
29.        k++;
30.     }
31.
32.     // Copy the remaining elements of L[], if there are any
33.     while (i < n1) {
34.        arr[k] = L[i];
35.        i++;
36.        k++;
37.     }
38.
39.     // Copy the remaining elements of R[], if there are any
40.     while (j < n2) {
41.        arr[k] = R[j];
42.        j++;
43.        k++;
44.     }
45. }
46.
47. void mergeSort(int arr[], int l, int r) {
48.     if (l < r) {
49.        // Find the middle point
50.        int m = l + (r - l) / 2;
```

```
51.
52.         // Sort first and second halves
53.         mergeSort(arr, l, m);
54.         mergeSort(arr, m + 1, r);
55.
56.         // Merge the sorted halves
57.         merge(arr, l, m, r);
58.     }
59. }
60.
61. int main() {
62.     int n, i;
63.
64.     printf("Enter the number of elements: ");
65.     scanf("%d", &n);
66.
67.     // Using VLA for array size and elements
68.     int arr[n];
69.
70.     printf("Enter elements: ");
71.     for (i = 0; i < n; i++)
72.         scanf("%d", &arr[i]);
73.
74.     mergeSort(arr, 0, n - 1);
75.
76.     printf("\nSorted array is: \n");
77.     for (i = 0; i < n; i++)
78.         printf("%d ", arr[i]);
79.
80.     return 0;
81. }
```

Q-48) C program to perform merge sort [using pointers]

```
1.  #include <stdio.h>
2.  #include <stdlib.h> // For malloc and free
3.
4.  int main() {
5.      int n, i;
6.
7.      printf("Enter the number of elements: ");
8.      scanf("%d", &n);
9.
10.     // Dynamically allocate memory for the array
11.     int* arr = (int*)malloc(n * sizeof(int));
12.     if (arr == NULL) {
```

```
13.     printf("Memory allocation failed.\n");
14.     return 1;
15. }
16.
17. printf("Enter elements: ");
18. for (i = 0; i < n; i++)
19.     scanf("%d", &arr[i]);
20.
21. // Merge Sort logic directly in main
22. if (n > 1) {
23.     int m = n / 2;
24.
25.     // Dynamically allocate memory for temporary arrays
26.     int* L = (int*)malloc(m * sizeof(int));
27.     int* R = (int*)malloc((n - m) * sizeof(int));
28.     if (L == NULL || R == NULL) {
29.         printf("Memory allocation failed.\n");
30.         free(arr); // Free previously allocated memory
31.         return 1;
32.     }
33.
34.     // Copy data to temporary arrays
35.     for (i = 0; i < m; i++)
36.         L[i] = arr[i];
37.     for (i = m; i < n; i++)
38.         R[i - m] = arr[i];
39.
40.     // Merge the temporary arrays back into arr[l..r]
41.     i = 0; // Initial index of first subarray
42.     int j = 0; // Initial index of second subarray
43.     int k = 0; // Initial index of merged subarray
44.     while (i < m && j < n - m) {
45.         if (L[i] <= R[j]) {
46.             arr[k] = L[i];
47.             i++;
48.         } else {
49.             arr[k] = R[j];
50.             j++;
51.         }
52.         k++;
53.     }
54.
55.     // Copy the remaining elements of L[], if there are any
56.     while (i < m) {
57.         arr[k] = L[i];
58.         i++;
```

```
59.             k++;
60.         }
61.
62.         // Copy the remaining elements of R[], if there are any
63.         while (j < n - m) {
64.             arr[k] = R[j];
65.             j++;
66.             k++;
67.         }
68.
69.         // Free dynamically allocated memory
70.         free(L);
71.         free(R);
72.     }
73.
74.     printf("\nSorted array is: \n");
75.     for (i = 0; i < n; i++)
76.         printf("%d ", arr[i]);
77.
78.     // Free dynamically allocated memory
79.     free(arr);
80.
81.     return 0;
82. }
```

ALGORITHM FOR QUESTION 45-48

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations.
3. In the `main` function, declare integer variables `n` and `i` to store the number of elements and the loop counter.
4. Prompt the user to enter the number of elements `n`.
5. Declare an array `arr` of size `n` to store the elements.
6. Prompt the user to enter the elements of the array.
7. Check if the number of elements `n` is greater than `1`. If not, the array is already sorted.
8. Calculate the middle index `m` of the array.
9. Enter a loop that iterates from `0` to `m` to sort the first and second halves of the array.
   - For each iteration, declare two temporary arrays `L` and `R` to store the left and right halves of the array.
   - Copy the elements from the original array `arr` to the temporary arrays `L` and `R`.
   - Initialize three indices `i`, `j`, and `k` to `0`, `0`, and `0` respectively.
   - Enter a loop to merge the temporary arrays back into the original array `arr`.
     - Compare the elements at indices `i` and `j` of `L` and `R` respectively.
     - If `L[i]` is less than or equal to `R[j]`, copy `L[i]` to `arr[k]` and increment `i`.
     - Otherwise, copy `R[j]` to `arr[k]` and increment `j`.
     - Increment `k` after each copy operation.
   - After the merge loop, copy any remaining elements from `L` and `R` to `arr`.
10. After the sorting loop, print the sorted array.
11. --End--

2] fp

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Declare a function `merge` to merge two subarrays of the given array.
   - This function takes the array, the starting index `l`, the middle index `m`, and the ending index `r` as parameters.
   - It dynamically allocates memory for two temporary arrays `L` and `R` to store the two halves of the array.
   - It then copies the elements from the original array into these temporary arrays.
   - It merges the two temporary arrays back into the original array in sorted order.
   - Finally, it frees the dynamically allocated memory for the temporary arrays.
4. Declare a function `mergeSort` to perform the merge sort on the given array.

- This function takes the array, the starting index `l`, and the ending index `r` as parameters.
- If `l` is less than `r`, it calculates the middle index `m`.
- It then recursively calls `mergeSort` on the first half of the array (from `l` to `m`) and the second half (from `m+1` to `r`).
- After sorting the two halves, it calls `merge` to merge them into a sorted array.

5. In the `main` function, declare integer variables `n` and `i` to store the number of elements and the loop counter.
6. Prompt the user to enter the number of elements and store it in `n`.
7. Dynamically allocate memory for the array `arr` of size `n`.
8. Prompt the user to enter the elements of the array.
9. Call `mergeSort` on the array with `l` set to `0` and `r` set to `n-1`.
10. Display the sorted array.
11. Free the dynamically allocated memory for the array.
12. --End--

3] f

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare a function `merge` that takes an array, two indices `l` and `m`, and another index `r`. This function is responsible for merging two subarrays of the given array.
   - Initialize variables `i`, `j`, and `k` for indexing the left subarray, right subarray, and the merged subarray, respectively.
   - Calculate the sizes of the left and right subarrays.
   - Create temporary arrays `L` and `R` to hold the left and right subarrays.
   - Copy the elements of the left and right subarrays into `L` and `R`.
   - Merge the temporary arrays back into the original array by comparing the elements of `L` and `R` and placing the smaller element into the original array.
   - Copy any remaining elements from `L` and `R` into the original array.
4. Declare a function `mergeSort` that takes an array and two indices `l` and `r`. This function is responsible for sorting the array using the merge sort algorithm.
   - If `l` is less than `r`, find the middle point `m` of the array.
   - Recursively sort the first half of the array using `mergeSort`.
   - Recursively sort the second half of the array using `mergeSort`.
   - Merge the two halves using the `merge` function.
5. In the `main` function, declare variables `n` and `i` for the number of elements in the array and for loop control, respectively.
6. Prompt the user to enter the number of elements in the array and read it into `n`.
7. Declare a variable-length array `arr` of size `n` to hold the elements of the array.
8. Prompt the user to enter the elements of the array and read them into `arr`.
9. Call `mergeSort` to sort the array.

10. Display the sorted array to the user.
11. --End--

4] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. In the `main` function, declare integer variables `n` and `i` to store the number of elements and loop counter.
4. Prompt the user to enter the number of elements and read it into `n`.
5. Dynamically allocate memory for an integer array `arr` of size `n` using `malloc`.
   - If memory allocation fails, display an error message, free any previously allocated memory, and exit the program with an error code.
6. Prompt the user to enter the elements of the array and read them into `arr`.
7. Check if `n` is greater than `1`. If so, proceed with the merge sort logic.
   - Calculate `m` as half of `n`.
   - Dynamically allocate memory for two temporary arrays `L` and `R` to hold the left and right halves of the original array.
      - If memory allocation fails for either `L` or `R`, display an error message, free previously allocated memory, and exit the program with an error code.
   - Copy the first half of `arr` into `L` and the second half into `R`.
8. Implement the merge logic:
   - Initialize three indices: `i` for the first subarray, `j` for the second subarray, and `k` for the merged subarray.
   - While `i` is less than `m` and `j` is less than `n - m`:
      - If the current element of `L` is less than or equal to the current element of `R`, copy the element from `L` to `arr` at index `k`, increment `i`, and `k`.
      - Otherwise, copy the element from `R` to `arr` at index `k`, increment `j`, and `k`.
   - After the loop, if there are any remaining elements in `L`, copy them to `arr`.
   - If there are any remaining elements in `R`, copy them to `arr`.
9. Free the dynamically allocated memory for `L` and `R`.
10. Print the sorted array.
11. Free the dynamically allocated memory for `arr`.
12. --End--

Q-49) C program to perform quick sort [without using pointer functions]

```
1.  // took help from algorithm provided by visualgo
2.
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.  #include <time.h>
6.
7.  int main() {
8.      // Initialize random seed
9.      srand(time(0));
10.
11.     // Prompt user for array size
12.     int n;
13.     printf("Enter the size of the array: ");
14.     scanf("%d", &n);
15.
16.     // Use VLA for the array
17.     int arr[n];
18.
19.     // Prompt user for array elements
20.     printf("Enter the elements of the array: ");
21.     for (int i = 0; i < n; i++) {
22.         scanf("%d", &arr[i]);
23.     }
24.
25.     // Print original array
26.     printf("Original Array: \n");
27.     for (int i = 0; i < n; i++) {
28.         printf("%d ", arr[i]);
29.     }
30.     printf("\n");
31.
32.     // QuickSort without functions and pointers
33.     int pivotIndex = 0; // Starting pivot index
34.     while (pivotIndex < n - 1) {
35.         int pivot = arr[pivotIndex];
36.         int storeIndex = pivotIndex + 1;
37.
38.         for (int i = pivotIndex + 1; i < n; i++) {
39.             if (arr[i] < pivot || (arr[i] == pivot && rand() % 2 == 0)) {
40.                 // Swap arr[i] and arr[storeIndex]
41.                 int temp = arr[i];
42.                 arr[i] = arr[storeIndex];
43.                 arr[storeIndex] = temp;
44.                 storeIndex++;
45.             }
```

```
        }

        // Swap pivot with arr[storeIndex - 1]
        int temp = arr[pivotIndex];
        arr[pivotIndex] = arr[storeIndex - 1];
        arr[storeIndex - 1] = temp;

        // Move to the next partition
        pivotIndex = storeIndex;
    }

    // Print sorted array
    printf("Sorted Array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}

```

Q-50) C program to perform quick sort [using functions]

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

// Function to prompt user for array size and elements
void promptArray(int arr[], int n) {
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
```

```
23.     if (low < high) {
24.         int pivotIndex = partition(arr, low, high);
25.         quickSort(arr, low, pivotIndex - 1);
26.         quickSort(arr, pivotIndex + 1, high);
27.     }
28. }
29.
30. // Partition function for QuickSort
31.
32.
33. // Swap function
34. void swap(int arr[], int a, int b) {
35.     int t = arr[a];
36.     arr[a] = arr[b];
37.     arr[b] = t;
38. }
39. int partition(int arr[], int low, int high) {
40.     int pivot = arr[high];
41.     int i = (low - 1);
42.
43.     for (int j = low; j <= high - 1; j++) {
44.         if (arr[j] < pivot) {
45.             i++;
46.             swap(arr, i, j);
47.         }
48.     }
49.     swap(arr, i + 1, high);
50.     return (i + 1);
51. }
52. int main() {
53.     // Initialize random seed
54.     srand(time(0));
55.
56.     // Prompt user for array size
57.     int n;
58.     printf("Enter the size of the array: ");
59.     scanf("%d", &n);
60.
61.     // Use VLA for the array
62.     int arr[n];
63.
64.     // Prompt user for array elements
65.     promptArray(arr, n);
66.
67.     // Print original array
68.     printf("Original Array: \n");
```

```
    printArray(arr, n);

    // QuickSort
    quickSort(arr, 0, n - 1);

    // Print sorted array
    printf("Sorted Array: \n");
    printArray(arr, n);

    return 0;
}
```

Q-51) C program to perform quick sort [using pointer functions]

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

// Function to swap two elements using pointers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function
int partition(int *arr, int low, int high) {
    int pivot = arr[low];
    int i = low + 1;
    int j = high;

    while (i <= j) { // Corrected condition
        while (i <= j && arr[i] <= pivot) {
            i++;
        }
        while (i <= j && arr[j] > pivot) {
            j--;
        }
        if (i < j) { // Ensure i is not equal to j to avoid unnecessary swap
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[low], &arr[j]); // Move pivot to its final position
    return j;
}

```

```
// QuickSort function using pointers
void quickSort(int *arr, int low, int high) {
    if (low < high) {
        // Partition the array
        int pivotIndex = partition(arr, low, high);
        // Recursively sort elements before and after the pivot
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}


int main() {
    // Initialize random seed
    srand(time(0));

    // Prompt user for array size
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    // Dynamically allocate memory for the array
    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Return an error code
    }

    // Prompt user for array elements
    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Print original array
    printf("Original Array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Call QuickSort function
    quickSort(arr, 0, n - 1);

    // Print sorted array
    printf("Sorted Array: \n");
```

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

    // Free dynamically allocated memory
    free(arr);

    return 0;
}
```

Q-52) C program to perform quick sort [using pointer]

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

int main() {
    srand(time(0));

    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int* arr = (int*)malloc(n * sizeof(int));
    if (!arr) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter the elements of the array: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Original Array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Simulate the stack with array indices
    int start = 0;
    int end = n - 1;
```

```
    int stack[n], top = -1;
    stack[++top] = start;
    stack[++top] = end;

    while (top >= 0) {
        end = stack[top--];
        start = stack[top--];

        int pivot = arr[end];
        int i = (start - 1);

        for (int j = start; j <= end - 1; j++) {
            if (arr[j] <= pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[end];
        arr[end] = temp;

        int p = i + 1;
        if (p - 1 > start) {
            stack[++top] = start;
            stack[++top] = p - 1;
        }
        if (p + 1 < end) {
            stack[++top] = p + 1;
            stack[++top] = end;
        }
    }

    printf("Sorted Array: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

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

ALGORITHM FOR QUESTION 49-52

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations, dynamic memory allocation, and time-related functions.
3. Initialize the random seed using `srand(time(0))` to ensure different random numbers are generated each time the program runs.
4. Prompt the user to enter the size of the array `n`.
5. Declare a variable-length array `arr` of size `n` to store the array elements.
6. Prompt the user to enter the elements of the array.
7. Print the original array to the console.
8. Implement the QuickSort algorithm without using functions and pointers:
   - Initialize a pivot index `pivotIndex` to `0`.
   - Enter a `while` loop that continues until `pivotIndex` is less than `n - 1`.
      - Inside the loop, set `pivot` to the value of `arr[pivotIndex]`.
      - Initialize `storeIndex` to `pivotIndex + 1`.
      - Enter a `for` loop that iterates from `pivotIndex + 1` to `n`.
         - Inside the loop, check if `arr[i]` is less than `pivot` or if it's equal to `pivot` and a random number is even.
         - If the condition is true, swap `arr[i]` and `arr[storeIndex]` and increment `storeIndex`.
      - Swap `arr[pivotIndex]` with `arr[storeIndex - 1]` to move the pivot to its correct position.
      - Move to the next partition by setting `pivotIndex` to `storeIndex`.
9. Print the sorted array to the console.
10. --End--

2] f

1. --Start--
2. Include necessary header files for standard input/output operations, dynamic memory allocation, and time-related functions.
3. Declare a function `promptArray` to prompt the user for the size of the array and its elements.
4. Declare a function `printArray` to print the elements of the array.
5. Declare a function `quickSort` to perform the QuickSort algorithm on an array.
6. Declare a function `partition` to partition the array around a pivot element.
7. Declare a function `swap` to swap two elements in the array.
8. In the `main` function, initialize the random seed using the current time.
9. Prompt the user to enter the size of the array.
10. Use Variable Length Array (VLA) to create an array of the specified size.
11. Prompt the user to enter the elements of the array using `promptArray`.

12. Print the original array using `printArray`.
13. Call `quickSort` to sort the array.
14. Print the sorted array using `printArray`.
15. --End--

Detailed Function Descriptions

- --promptArray(int arr[], int n)--
  - Prompts the user to enter the elements of the array.
  - Iterates through each element of the array, using `scanf` to read the user's input.

- --printArray(int arr[], int n)--
  - Prints the elements of the array.
  - Iterates through each element of the array, using `printf` to display the element.

- --quickSort(int arr[], int low, int high)--
  - Implements the QuickSort algorithm.
  - If the `low` index is less than the `high` index, it partitions the array around a pivot element and recursively sorts the sub-arrays.

- --partition(int arr[], int low, int high)--
  - Partitions the array around a pivot element.
  - Initializes a pivot element as the last element of the array.
  - Iterates through the array, swapping elements that are less than the pivot to the left side of the array.
  - Swaps the pivot element to its correct position in the sorted array.
  - Returns the index of the pivot element.

- --swap(int arr[], int a, int b)--
  - Swaps two elements in the array.
  - Uses a temporary variable to hold the value of the first element, then assigns the value of the second element to the first, and finally assigns the value of the temporary variable to the second.

3] fp

1. --Start--
2. Include necessary header files for standard input/output operations, dynamic memory allocation, and time-related functions.
3. Declare a function `swap` to swap two elements using pointers.
4. Declare a function `partition` to partition the array around a pivot element, returning the index of the pivot element after partitioning.

5. Declare a function `quickSort` to perform the QuickSort algorithm on the array, using the `partition` function to recursively sort the elements.
6. In the `main` function, initialize the random seed using the current time.
7. Prompt the user to enter the size of the array and store it in a variable `n`.
8. Dynamically allocate memory for the array of size `n` and store the pointer to the first element in a variable `arr`.
9. Check if the memory allocation was successful. If not, display an error message and return an error code.
10. Prompt the user to enter the elements of the array and store them in `arr`.
11. Print the original array to the console.
12. Call the `quickSort` function to sort the array.
13. Print the sorted array to the console.
14. Free the dynamically allocated memory for the array.
15. --End--

Detailed Function Logic:

- --swap(int *a, int *b)--:
  - Swaps the values of two integers pointed to by `a` and `b` by using a temporary variable `temp`.

- --partition(int *arr, int low, int high)--:
  - Selects the first element of the array as the pivot.
  - Initializes two pointers, `i` and `j`, to the elements immediately after the pivot and the last element of the array, respectively.
  - Increments `i` while the element pointed to by `i` is less than or equal to the pivot.
  - Decrements `j` while the element pointed to by `j` is greater than the pivot.
  - If `i` is less than `j`, swaps the elements pointed to by `i` and `j`.
  - Continues this process until `i` is not less than `j`.
  - Swaps the pivot with the element pointed to by `j`.
  - Returns the index of the pivot element.

- --quickSort(int *arr, int low, int high)--:
  - If `low` is less than `high`, partitions the array using the `partition` function and recursively sorts the elements before and after the pivot.
  - The base case for the recursion is when `low` is not less than `high`, indicating that the array is already sorted.

4] p

1. --Start--
2. Include necessary header files for standard input/output operations, dynamic memory allocation, and time-related functions.

3. In the `main` function, initialize the random number generator with the current time to ensure different random numbers each run.
4. Declare an integer variable `n` to store the size of the array.
5. Prompt the user to enter the size of the array and read it.
6. Allocate memory dynamically for an integer array `arr` of size `n` using `malloc`.
   - If memory allocation fails, display an error message and return `1` to indicate an error.
7. Prompt the user to enter the elements of the array and read them.
8. Display the original array to the user.
9. Initialize variables `start`, `end`, and `top` for simulating a stack with array indices.
   - `start` is initialized to `0`.
   - `end` is initialized to `n - 1`.
   - `top` is initialized to `-1`.
10. Push `start` and `end` onto the stack.
11. Enter a `while` loop that continues until the stack is empty.
   - Pop `end` and `start` from the stack.
   - Select the last element of the current subarray as the pivot.
   - Initialize `i` to one position before the start of the current subarray.
   - Iterate through the current subarray from `start` to `end - 1`.
      - If the current element is less than or equal to the pivot, increment `i` and swap the elements at indices `i` and `j`.
   - Swap the pivot element with the element at index `i + 1`.
   - Calculate the partition index `p`.
   - If there are elements before `p`, push `start` and `p - 1` onto the stack.
   - If there are elements after `p`, push `p + 1` and `end` onto the stack.
12. After the loop, the array is sorted.
13. Display the sorted array to the user.
14. Free the dynamically allocated memory for the array.
15. --End--

Q-53) C program to perform selection sort [without using pointer functions]

```
1.  // took help from visualalgo
2.  #include <stdio.h>
3.
4.  int main() {
5.      int numOfElements;
6.      int i, j, minIndex, temp;
7.
8.      // Prompt user for array size
9.      printf("Enter the size of the array: ");
10.     scanf("%d", &numOfElements);
11.
12.     // Declare VLA with size determined by user input
13.     int arr[numOfElements];
14.
15.     // Prompt user for array elements
16.     printf("Enter the elements of the array: ");
17.     for (i = 0; i < numOfElements; i++) {
18.         scanf("%d", &arr[i]);
19.     }
20.
21.     // Implementing Selection Sort
22.     for (i = 0; i < numOfElements - 1; i++) {
23.         minIndex = i; // Assume the first unsorted element is the minimum
24.
25.         // Find the minimum element in the unsorted part of the array
26.         for (j = i + 1; j < numOfElements; j++) {
27.             if (arr[j] < arr[minIndex]) {
28.                 minIndex = j; // Update the minimum index
29.             }
30.         }
31.
32.         // Swap the minimum element with the first unsorted position
33.         temp = arr[minIndex];
34.         arr[minIndex] = arr[i];
35.         arr[i] = temp;
36.     }
37.
38.     // Print the sorted array
39.     printf("Sorted array: \n");
40.     for (i = 0; i < numOfElements; i++) {
41.         printf("%d ", arr[i]);
42.     }
43.     printf("\n");
44.
```

```
45.     return 0;
46. }
```

Q-54) C program to perform selection sort [using functions]

```
1.  #include <stdio.h>
2.
3.  // Function to read the array size and elements from the user
4.  int readArray(int arr[]) {
5.      int numOfElements;
6.      printf("Enter the size of the array: ");
7.      scanf("%d", &numOfElements);
8.
9.      printf("Enter the elements of the array: ");
10.     for (int i = 0; i < numOfElements; i++) {
11.         scanf("%d", &arr[i]);
12.     }
13.
14.     return numOfElements;
15. }
16.
17. // Function to perform Selection Sort
18. void selectionSort(int arr[], int numOfElements) {
19.     int i, j, minIndex, temp;
20.
21.     for (i = 0; i < numOfElements - 1; i++) {
22.         minIndex = i;
23.
24.         for (j = i + 1; j < numOfElements; j++) {
25.             if (arr[j] < arr[minIndex]) {
26.                 minIndex = j;
27.             }
28.         }
29.
30.         temp = arr[minIndex];
31.         arr[minIndex] = arr[i];
32.         arr[i] = temp;
33.     }
34. }
35.
36. // Function to print the sorted array
37. void printArray(int arr[], int numOfElements) {
38.     printf("Sorted array: \n");
39.     for (int i = 0; i < numOfElements; i++) {
40.         printf("%d ", arr[i]);
```

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

int main() {
    int arr[100]; // Assuming a maximum size of 100 for simplicity
    int numOfElements = readArray(arr);
    selectionSort(arr, numOfElements);
    printArray(arr, numOfElements);

    return 0;
}
```

Q-55) C program to perform selection sort [using pointers]

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

int main() {
    int numOfElements;
    int i, j, minIndex, temp;
    int *arr; // Pointer to dynamically allocated array

    // Prompt user for array size
    printf("Enter the size of the array: ");
    scanf("%d", &numOfElements);

    // Dynamically allocate memory for the array
    arr = (int*)malloc(numOfElements * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1; // Return an error code
    }

    // Prompt user for array elements
    printf("Enter the elements of the array: ");
    for (i = 0; i < numOfElements; i++) {
        scanf("%d", &arr[i]);
    }

    // Implementing Selection Sort
    for (i = 0; i < numOfElements - 1; i++) {
        minIndex = i; // Assume the first unsorted element is the minimum

        // Find the minimum element in the unsorted part of the array
        for (j = i + 1; j < numOfElements; j++) {
```

```
            if (arr[j] < arr[minIndex]) {
                minIndex = j; // Update the minimum index
            }
        }

        // Swap the minimum element with the first unsorted position
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }

    // Print the sorted array
    printf("Sorted array: \n");
    for (i = 0; i < numOfElements; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Free dynamically allocated memory
    free(arr);

    return 0;
}
```

Q-56) C program to perform selection sort [using functions pointers]

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

// Function to read the array size and elements from the user
int* readArray(int* arr, int* numOfElements) {
    printf("Enter the size of the array: ");
    scanf("%d", numOfElements);

    // Dynamically allocate memory for the array based on the user's input
    arr = (int*)malloc(*numOfElements * sizeof(int));

    printf("Enter the elements of the array: ");
    for (int i = 0; i < *numOfElements; i++) {
        scanf("%d", &arr[i]);
    }

    return arr;
}

// Function to perform Selection Sort
```

```
21. void selectionSort(int* arr, int numOfElements) {
22.     int i, j, minIndex, temp;
23.
24.     for (i = 0; i < numOfElements - 1; i++) {
25.         minIndex = i;
26.
27.         for (j = i + 1; j < numOfElements; j++) {
28.             if (arr[j] < arr[minIndex]) {
29.                 minIndex = j;
30.             }
31.         }
32.
33.         temp = arr[minIndex];
34.         arr[minIndex] = arr[i];
35.         arr[i] = temp;
36.     }
37. }
38.
39. // Function to print the sorted array
40. void printArray(int* arr, int numOfElements) {
41.     printf("Sorted array: \n");
42.     for (int i = 0; i < numOfElements; i++) {
43.         printf("%d ", arr[i]);
44.     }
45.     printf("\n");
46. }
47.
48. int main() {
49.     int numOfElements;
50.     int* arr; // Declare a pointer to an int
51.
52.     arr = readArray(arr, &numOfElements); // Dynamically allocate memory for the
    array
53.     selectionSort(arr, numOfElements);
54.     printArray(arr, numOfElements);
55.
56.     // Free the dynamically allocated memory
57.     free(arr);
58.
59.     return 0;
60. }
```

ALGORITHM FOR QUESTION 53-56

1] normal

1. --Start--
2. Include necessary header files for standard input/output operations.
3. In the `main` function, declare integer variables `numOfElements`, `i`, `j`, `minIndex`, and `temp` to store the number of elements in the array, loop counters, the index of the minimum element, and a temporary variable for swapping elements.
4. Prompt the user to enter the size of the array and store it in `numOfElements`.
5. Declare a variable-length array `arr` with a size determined by the user's input.
6. Prompt the user to enter the elements of the array and store them in `arr`.
7. Implement the Selection Sort algorithm:
   - For each element in the array (except the last one), starting from the first element:
     - Assume the current element is the minimum.
     - Iterate through the unsorted part of the array (from the next element to the end of the array).
       - If a smaller element is found, update the `minIndex` to the index of the smaller element.
     - Swap the element at the `minIndex` with the first unsorted element.
8. After sorting, print the sorted array to the console.
9. --End--

2] f

1. --Start--
2. Include necessary header files for standard input/output operations.
3. Declare a function `readArray` to read the array size and elements from the user.
4. Declare a function `selectionSort` to perform the Selection Sort algorithm on an array.
5. Declare a function `printArray` to print the sorted array.
6. In the `main` function, declare an integer array `arr` with a maximum size of 100 for simplicity.
7. Call `readArray` to read the array size and elements from the user, storing the number of elements in `numOfElements`.
8. Call `selectionSort` to sort the array using the Selection Sort algorithm.
9. Call `printArray` to print the sorted array.
10. --End--

Function Descriptions

- --readArray(int arr[])--
  - Prompt the user to enter the size of the array.

- Read the size of the array from the user and store it in `numOfElements`.
- Prompt the user to enter the elements of the array.
- Read each element of the array from the user and store it in the array `arr`.
- Return the number of elements in the array.

- --selectionSort(int arr[], int numOfElements)--
- Iterate through the array from the first element to the second-to-last element.
- For each element, find the smallest element in the unsorted part of the array.
- Swap the current element with the smallest element found.
- Continue this process until the entire array is sorted.

- --printArray(int arr[], int numOfElements)--
- Print the sorted array to the console.
- Iterate through the array and print each element followed by a space.
- Print a newline character after printing all elements to end the line.

3] p

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. In the `main` function, declare integer variables `numOfElements`, `i`, `j`, `minIndex`, and `temp`.
4. Declare a pointer `arr` to an integer, which will be used to dynamically allocate memory for the array.
5. Prompt the user to enter the size of the array and store it in `numOfElements`.
6. Dynamically allocate memory for the array using `malloc`, with the size being `numOfElements` times the size of an integer.
   - If memory allocation fails (`malloc` returns `NULL`), display an error message and return an error code.
7. Prompt the user to enter the elements of the array and store them in `arr`.
8. Implement the Selection Sort algorithm:
   - For each element in the array (except the last one), find the minimum element in the unsorted part of the array.
      - Initialize `minIndex` to the current index.
      - Iterate through the unsorted part of the array, comparing each element with the current `minIndex`.
         - If a smaller element is found, update `minIndex` to the index of the smaller element.
      - Swap the element at `minIndex` with the element at the current index.
9. Print the sorted array.
10. Free the dynamically allocated memory using `free`.
11. Return `0` to indicate successful execution.
12. --End--

4] fp

1. --Start--
2. Include necessary header files for standard input/output operations and dynamic memory allocation.
3. Declare a function `readArray` to read the array size and elements from the user, dynamically allocate memory for the array based on the user's input, and return the array.
4. Declare a function `selectionSort` to perform the Selection Sort algorithm on the given array.
5. Declare a function `printArray` to print the sorted array.
6. In the `main` function, declare an integer `numOfElements` and a pointer to an integer `arr`.
7. Call `readArray` to dynamically allocate memory for the array and read the array size and elements from the user.
8. Call `selectionSort` to sort the array using the Selection Sort algorithm.
9. Call `printArray` to print the sorted array.
10. Free the dynamically allocated memory for the array.
11. --End--

Detailed Function Descriptions:

- --readArray(int* arr, int* numOfElements)--
  - Prompt the user to enter the size of the array.
  - Dynamically allocate memory for the array based on the user's input.
  - Prompt the user to enter the elements of the array.
  - Return the array.

- --selectionSort(int* arr, int numOfElements)--
  - Iterate through the array from the first element to the second-to-last element.
  - For each element, find the smallest element in the unsorted part of the array.
  - Swap the current element with the smallest element found.
  - Repeat the process until the entire array is sorted.

- --printArray(int* arr, int numOfElements)--
  - Print the sorted array elements.

Execution Flow:

1. The program starts by declaring a pointer to an integer `arr` and an integer `numOfElements`.
2. The `readArray` function is called, which prompts the user for the array size and elements. It dynamically allocates memory for the array and returns it.
3. The `selectionSort` function is called with the array and its size as arguments. It sorts the array using the Selection Sort algorithm.

4. The `printArray` function is called to print the sorted array.
5. The dynamically allocated memory for the array is freed to prevent memory leaks.
6. The program ends.