

Data Structures and Algorithms – 1

Array Implementation :

AIM: To implement the Array and its various functions Such as insertion, deletion, and traversal with the user Inputs using a c programming language.

Procedure:

1. initialize an array with a given capacity and set the size to 0.
2. Repeat the following steps until the user chooses to exit:
3. Display the menu options to the user.
4. Read the user's choice.
5. Perform the corresponding operation based on the user's choice:
 - a. If the user chooses to insert an element, read the element from the user and insert it into the array.
 - b. If the user chooses to delete an element, read the element from the user and delete it from the array.
 - c. If the user chooses to traverse the array, display all the elements in the array.
 - d. If the user chooses to exit, end the loop.
6. Display the updated array after each operation.

Code :

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

```
void insert(int arr[], int* size, int capacity, int element);
```

```
void deleteElement(int arr[], int* size, int element);
```

```
void traverse(int arr[], int size);
```

```
void insert(int arr[], int* size, int capacity, int element) {  
    if (*size >= capacity) {  
        printf("Array is full. Cannot insert more elements.\n");  
        return; }  
  
    arr[*size] = element;  
    (*size)++; }
```

```
void deleteElement(int arr[], int* size, int element) {  
    int found = 0;  
  
    for (int i = 0; i < *size; i++) {  
        if (arr[i] == element) {  
            found = 1;  
  
            for (int j = i; j < *size - 1; j++) {  
                arr[j] = arr[j + 1];  
            }  
  
            (*size)--; break;  
        } }  
}
```

```
    if (!found) {  
        printf("Element not found in the array. Cannot delete.\n");  
    }  
}
```

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

```
int main() {  
    int capacity, size = 0;  
  
    printf("Enter the capacity of the array: ");  
    scanf("%d", &capacity);  
  
    int arr[capacity];  
  
    int choice, element;  
    do {  
        printf("Array Operations:\n");  
        printf("1. Insert an element\n");  
        printf("2. Delete an element\n");  
        printf("3. Traverse the array\n");  
        printf("4. Exit\n"); printf("Enter
```

```

your choice: "); scanf("%d",
&choice);
switch (choice) {
    case 1:
        printf("Enter the element to insert: "); scanf("%d",
            &element);
        insert(arr, &size, capacity, element); break;
        case 2: printf("Enter the
            element to delete: "); scanf("%d",
            &element); deleteElement(arr, &size,
            element); break;
    case 3:
        traverse(arr, size); break;
    case 4:
        printf("Exiting...\n"); break;
    default: printf("Invalid choice. Try
        again.\n"); break;
}

printf("\n");
} while (choice != 4);

return 0;
}

```

Output :

```

Enter the capacity of the array: 5
Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 1
Enter the element to insert: 4

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 1
Enter the element to insert: 7

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 1
Enter the element to insert: 8

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 3
Array elements: 7 8 3 9

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: █

```

Result :

The Array and its operations were successfully implemented and executed with the c programming language.

1. Multidimensional Array :

AIM: To implement the Multidimensional array and its implementations using c programming language.

Procedure :

1. Initialize a multi-dimensional array with a given number of rows and columns.
2. Repeat the following steps until the user chooses to exit:
3. a. Display the menu options to the user.
4. b. Read the user's choice.
5. c. Perform the corresponding operation based on the user's choice:
6. - If the user chooses to insert an element, read the row, column, and element from the user and insert it into the array at the specified position.

7. - If the user chooses to delete an element, read the element from the user and delete all occurrences of it from the array.
8. - If the user chooses to traverse the array, display all the elements in the array.
9. - If the user chooses to exit, end the loop.
- 10.d. Display the updated array after each operation.

Code :

```
#include <stdio.h>

#define ROWS 3

#define COLS 3

void insert(int arr[][COLS], int row, int col, int element); void
deleteElement(int arr[][COLS], int row, int col, int element); void
traverse(int arr[][COLS], int row, int col);

void insert(int arr[][COLS], int row, int col, int element) {
    arr[row][col] = element;
}

void deleteElement(int arr[][COLS], int row, int col, int element) {
    int found = 0;
    for (int i = 0; i < row; i++)
        { for (int j = 0; j < col; j++)
            {
                if (arr[i][j] == element) {
                    found = 1;
                    arr[i][j] = 0; break;
                }
            }
        }
```

```
} }
```

```
if (!found) {  
    printf("Element not found in the array. Cannot  
delete.\n"); } }
```

```
void traverse(int arr[][COLS], int row, int col) {  
    printf("Array elements:\n");  
    for (int i = 0; i < row; i++)  
    { for (int j = 0; j < col; j++)  
        { printf("%d ", arr[i][j]);  
        }  
        printf("\n"); }  
    }
```

```
int main() {  
    int arr[ROWS][COLS] = {0};  
  
    int choice, row, col, element;  
    do {  
        printf("Array Operations:\n");  
        printf("1. Insert an element\n");  
        printf("2. Delete an element\n");  
        printf("3. Traverse the array\n");  
        printf("4. Exit\n"); printf("Enter  
your choice: "); scanf("%d",  
        &choice);
```

```

switch (choice) {
    case 1:
        printf("Enter the row and column to insert: ");
        scanf("%d %d", &row, &col); printf("Enter
        the element to insert: "); scanf("%d",
        &element); insert(arr, row, col, element);
        break;
    case 2:
        printf("Enter the element to delete: ");
        scanf("%d", &element); deleteElement(arr,
        ROWS, COLS, element); break;
    case 3:
        traverse(arr, ROWS, COLS);
        break;
    case 4:
        printf("Exiting...\n"); break;
    default: printf("Invalid choice. Try
        again.\n"); break;
}

printf("\n");
} while (choice != 4);

return 0;
}

```


Output :

```
Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 1
Enter the row and column to insert: 1 1
Enter the element to insert: 12

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 1
Enter the row and column to insert: 1 2
Enter the element to insert: 2

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 1
Enter the row and column to insert: 1 3
Enter the element to insert: 4

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 1
Enter the row and column to insert: 2 1
Enter the element to insert: 4

Array Operations:
1. Insert an element
4. Exit
Enter your choice: 1
Enter the row and column to insert: 2 1
Enter the element to insert: 4

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 3
Array elements:
0 0 0
0 12 2
4 4 0

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 2
Enter the element to delete: 12

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
Enter your choice: 3
Array elements:
0 0 0
0 0 2
4 4 0

Array Operations:
1. Insert an element
2. Delete an element
3. Traverse the array
4. Exit
```

Result :

The Multidimensional Array was successfully implemented and executed with user inputs using C programming language.

3. Linked List Implementation :

AIM: To implement the Linked list data structure with its various functions such as insertion, deletion and display using C language.

Procedure/Algorithm :

1. Start with an empty linked list ('head' is 'NULL').
2. Enter a loop until the user chooses to exit.
3. Display a menu of options for the user to choose from.
4. Read the user's choice.
5. Based on the user's choice, - Option 1: Create a list:
 - Read the number of nodes to be created ('n').
 - Create the list by iterating 'n' times:
 - Read the data for the current node.
 - Create a new node ('ptr') and assign the data to it.
 - If it's the first node, set 'head' to 'ptr'.
 - Otherwise, append 'ptr' to the end of the list.
- Option 2: Insert a node:
 - Read the data for the node to be inserted ('da').
 - Read the insertion position choice ('ch').
 - Based on 'ch', perform one of the following:
 - Insert at the beginning:
 - Create a new node ('np') with 'da' as data.
 - Set 'np->next' to the current 'head'.
 - Update 'head' to 'np'.
 - Insert at an index:
 - Read the index of the node to be inserted.
 - Traverse the list to the previous node of the given index.

- Create a new node (`np`) with `da` as data.
- Update the next pointers to insert `np` at the specified index.
- Insert at the end:
 - Create a new node (`np`) with `da` as data.
 - Traverse the list to the last node.
 - Set the next pointer of the last node to `np`.
- Option 3: Delete a node:
 - Read the deletion position choice (`ch`).
 - Based on `ch`, perform one of the following:
 - Delete the first node:
 - Update `head` to the next node.
 - Free the memory of the previous head node.
 - Delete at an index:
 - Read the index of the node to be deleted.
 - Traverse the list to the previous node of the given index.
 - Update the next pointer to skip the node to be deleted.
 - Free the memory of the node to be deleted.
 - Delete the last node:
 - Traverse the list to the second-to-last node.
 - Update its next pointer to `NULL`.
 - Free the memory of the last node.
 - Option 4: Display the list:
 - Traverse the list from `head` to the last node.
 - Print the data of each node.
 - Option 5: Exit the program.

6. End the loop and the program.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

struct node { int data; struct
                node* next;
            };

struct node* createList(struct node* head, int n);
struct node* insertNode(struct node* head, int n);
struct node* deleteNode(struct node* head, int n);
void displayList(struct node* head);

int main()
{ int n; int
  ch = 0;
  struct node* head = NULL;
  while (ch != 5) {
      printf("\nEnter\n1. Create a list\n2. Insert a node\n3. Delete a
node\n4. Display the list\n5. Exit\n");
      scanf("%d", &ch);
      switch (ch) {
          case 1:
              printf("Enter the number of nodes to be created: ");
              scanf("%d", &n);
              head = createList(head, n); break;
          case 2: head =
              insertNode(head, n); break;
```

```

        case 3: head =
            deleteNode(head, n); break;
        case 4:
            displayList(head); break;
        case 5: printf("Exiting the
            program.\n"); break;
        default: printf("\nEnter a valid
            input.\n"); break;
    } } return 0; } struct node* createList(struct
node* head, int n) { struct node* temp, * ptr; int i,
d;
    for (i = 1; i <= n; i++) {
        printf("Enter the data for node %d: ", i);
        scanf("%d", &d); ptr = (struct
node*)malloc(sizeof(struct node)); ptr->data =
d; ptr->next = NULL; if (head == NULL) {
            head = ptr;
            temp = ptr;
        } else { temp->next
            = ptr; temp = ptr;
        } }
    return head; }

```

```

struct node* insertNode(struct node* head, int n) {
    struct node* np = (struct node*)malloc(sizeof(struct node)); int
da, ch; struct node* temp = head; printf("\nEnter the data for

```

```

the node to be inserted: "); scanf("%d", &da); printf("\nEnter 1
for insertion at the beginning, 2 for insertion at
an index, or 3 for insertion at the end: "); scanf("%d",
&ch);
np->data = da; if
(ch == 1) { np-
>next = head; head
= np;
} else if (ch == 2) { int index, i; printf("\nEnter the index
at which to insert the node: "); scanf("%d", &index);
for (i = 1; i < index; i++) {
    if (temp == NULL) {
        printf("\nInvalid index. Insertion not possible.\n");
        return head; } temp = temp->next;
    } np->next = temp-
>next; temp->next = np;
} else if (ch == 3) { if
(head == NULL)
{ head = np; np-
>next = NULL;
} else { while (temp->next !=
NULL) {
    temp = temp->next;
} temp->next = np;
np->next =
NULL; }
} else {

```

```

        printf("\nInvalid choice. Insertion not possible.\n");
    } return head; } struct node* deleteNode(struct
node* head, int n) {
    if (head == NULL) {
        printf("\nThe list is empty. Deletion not possible.\n"); return
head; } int ch; struct node* temp = head; struct node* prev =
NULL; printf("\nEnter 1 to delete the first node, 2 to delete a
node at an
index, or 3 to delete the last node: "); scanf("%d",
    &ch);

    if (ch == 1) { head =
        head->next;
        free(temp);
    } else if (ch == 2) { int index, i; printf("\nEnter the
index of the node to be deleted: "); scanf("%d",
    &index); for (i = 1; i < index; i++) {
        if (temp == NULL) {
            printf("\nInvalid index. Deletion not possible.\n");
            return head; }
        prev = temp; temp
        = temp->next;
    } if (temp == NULL)
    {
        printf("\nInvalid index. Deletion not possible.\n");
        return head; }

```

```

        prev->next = temp->next; free(temp);
    } else if (ch == 3) { while (temp-
        >next != NULL) {
            prev = temp; temp
            = temp->next;
        } prev->next =
        NULL;
        free(temp);
    } else { printf("\nInvalid choice. Deletion not
        possible.\n");
    } return
head; }

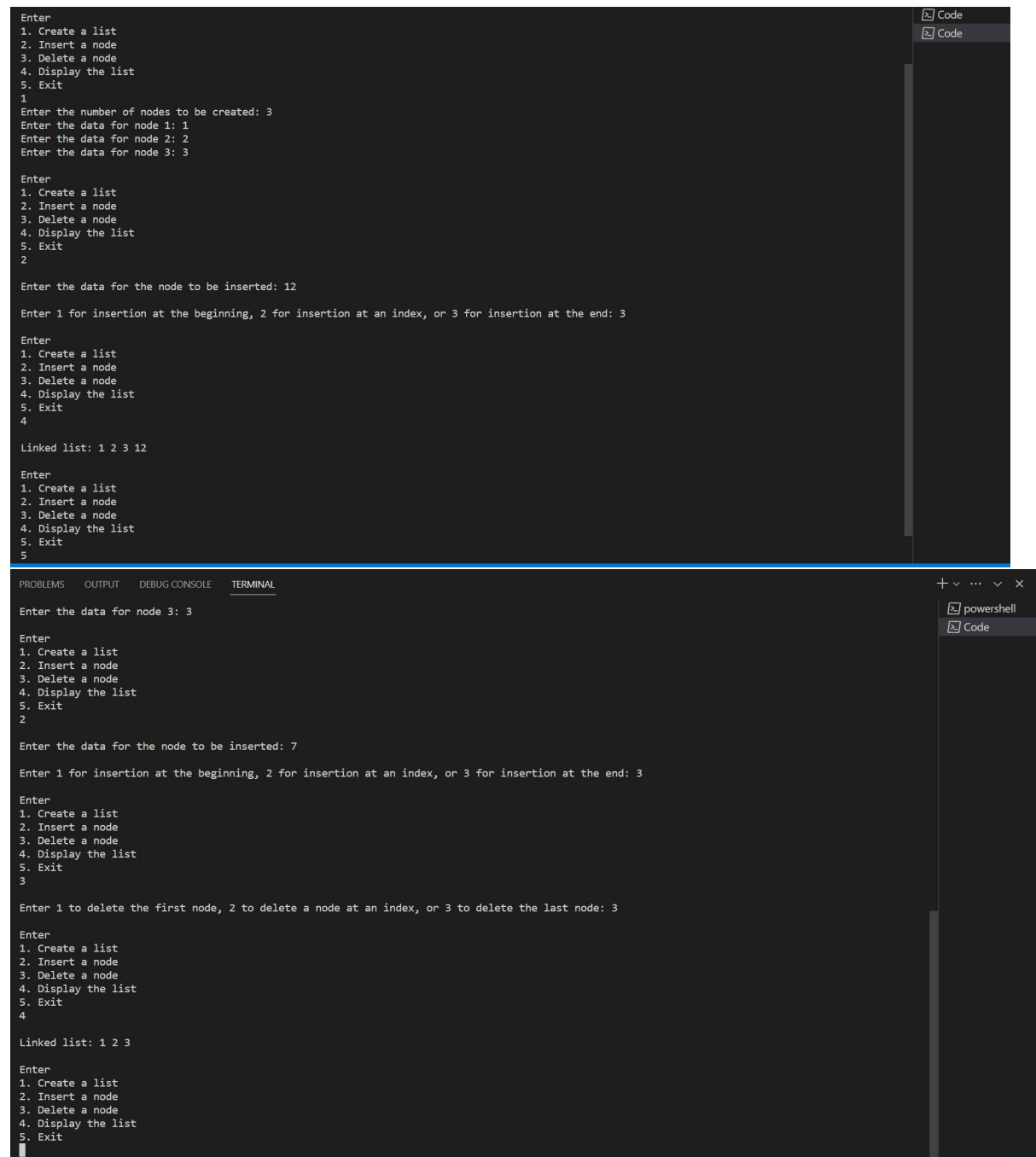
```

```

void displayList(struct node* head) {
    if (head == NULL) {
        printf("\nThe list is empty.\n");
        return; } struct node* temp =
head; printf("\nLinked list: ");
while (temp != NULL) {
    printf("%d ", temp->data);
    temp = temp->next;
}
printf("\n"); }

```


OUTPUT :



```
Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
1
Enter the number of nodes to be created: 3
Enter the data for node 1: 1
Enter the data for node 2: 2
Enter the data for node 3: 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
2

Enter the data for the node to be inserted: 12

Enter 1 for insertion at the beginning, 2 for insertion at an index, or 3 for insertion at the end: 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
4

Linked list: 1 2 3 12

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
5

5

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Enter the data for node 3: 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
2

Enter the data for the node to be inserted: 7

Enter 1 for insertion at the beginning, 2 for insertion at an index, or 3 for insertion at the end: 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
3

Enter 1 to delete the first node, 2 to delete a node at an index, or 3 to delete the last node: 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
4

Linked list: 1 2 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
5

5
```

Result :

The Single Linked list data structure with all its functions was execute .

Double Linked List :

AIM:

To implement the Double Linked List data structure and its functionalities such as insertion , deletion and traversal using the C programming language.

Procedure:

1. Creating a Doubly Linked List:

- a. Start with an empty head node.
- b. For each node to be inserted:
- c. Create a new node with the given data.
- d. If the head is empty, set the new node as the head.
- e. Otherwise, set the new node as the next node of the current tail node.
- f. Update the previous pointer of the new node to point to the current tail node.
- g. Update the tail node to the new node.

2. Inserting a Node:

- a. Create a new node with the given data.
- b. Based on the desired position of insertion:
- c. For insertion at the beginning:
- d. Set the next pointer of the new node to the current head.
- e. Update the previous pointer of the current head to the new node.
- f. Set the new node as the new head.
- g. For insertion at an index:
- h. Traverse the list to the desired index.
- i. Set the next pointer of the new node to the next node of the current node.
- j. Set the previous pointer of the new node to the current node.
- k. Update the next pointer of the current node to the new node.

- l. If the next node is not NULL, update its previous pointer to the new node.
- m. For insertion at the end:
- n. Traverse the list to the last node.
- o. Set the next pointer of the last node to the new node.
- p. Set the previous pointer of the new node to the last node.
- q. Set the next pointer of the new node to NULL.

3. Deleting a Node:

- a. Traverse the list to find the node to be deleted:
- b. If the node is found:
- c. If the node is the head:
- d. Update the next node as the new head.
- e. If the new head is not NULL, update its previous pointer to NULL.
- f. Otherwise, update the next pointer of the previous node to the next node of the current node.
- g. If the next node is not NULL, update its previous pointer to the previous node.
- h. Free the memory occupied by the current node.
- i. If the node is not found, display an appropriate message.

4. Displaying the Doubly Linked List:

- a. Traverse the list in forward order:
- b. Start from the head node.
- c. Print the data of each node.
- d. Move to the next node.
- e. Traverse the list in reverse order:
- f. Start from the last node.
- g. Print the data of each node.
- h. Move to the previous node.

5. Exit.

Code:

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

struct node {
    int data; struct
    node* prev;
    struct node* next;
};

struct node* createNode(int data)
{ struct node* newNode = (struct
node*)malloc(sizeof(struct node));
newNode->data = data; newNode->prev
= NULL; newNode->next = NULL;
return newNode; }

struct node* createList(struct node* head, int n)
{ if (n <= 0) { printf("Invalid number of nodes.
Creation not
possible.\n");
return head; }

int i, data; struct node*
newNode, * temp;
```

```
printf("Enter the data for node 1: ");  
scanf("%d", &data); head =  
createNode(data); temp = head;
```

```
for (i = 2; i <= n; i++) { printf("Enter the  
data for node %d: ", i); scanf("%d",  
&data);
```

```
newNode = createNode(data); temp-  
>next = newNode; newNode->prev =  
temp;  
temp = newNode;  
}
```

```
return head; }
```

```
struct node* insertNode(struct node* head, int data) {  
    if (head == NULL) {  
        printf("The list is empty. Insertion not  
possible.\n");  
        return head; }
```

```
struct node* newNode = createNode(data);  
int ch;
```

```
printf("\nEnter 1 for insertion at the beginning, 2 for  
insertion at an index, or 3 for insertion at the end: ");  
scanf("%d", &ch);
```

```

if (ch == 1) {
    newNode->next = head;
    head->prev = newNode;
    head = newNode;
} else if (ch == 2) { int index, i; printf("\nEnter
the index at which to insert the node: ");
    scanf("%d", &index);

    struct node* temp = head;
    for (i = 1; i < index; i++) {
        if (temp == NULL) {
            printf("\nInvalid index. Insertion not
possible.\n"); return head; }
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("\nInvalid index. Insertion not
possible.\n");
        return head; }

    newNode->next = temp->next;
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    } temp->next =
    newNode; newNode->
    >prev = temp;

```

```

    } else if (ch == 3) { struct node*
        temp = head; while (temp-
        >next != NULL) {
            temp = temp->next;
        }

        temp->next = newNode; newNode-
        >prev = temp;
    } else { printf("\nInvalid choice.
        Insertion not
possible.\n"); }

    return head; }

struct node* deleteNode(struct node* head, int data) {
    if (head == NULL) {
        printf("\nThe list is empty. Deletion not
possible.\n");
        return head; }

    struct node* temp = head;

    while (temp != NULL) {
        if (temp->data == data) { if
            (temp->prev == NULL) {
                head = temp->next;
                if (head != NULL) {
                    head->prev = NULL;

```

```

    }
    } else if (temp->next == NULL) { temp->prev-
        >next = NULL;
    } else { temp->prev->next = temp-
        >next; temp->next->prev =
        temp->prev;
    }

```

```

        free(temp); printf("Node with data %d
        deleted from the
list.\n", data);
        return head; }

```

```

        temp = temp->next;
    }

```

```

        printf("Node with data %d not found in the list.\n",
data);
        return head; }

```

```

void displayList(struct node* head)
{ if (head == NULL) {
    printf("\nThe list is empty.\n");
return; }

```

```

printf("\nLinked list in forward order: ");
struct node* temp = head;
while (temp != NULL) {

```



```
    printf("%d ", temp->data);  
    temp = temp->next;  
}
```

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

```
int main() {  
    int n, data, ch = 0; struct  
    node* head = NULL;  
  
    while (ch != 5) {  
        printf("\nEnter\n1. Create a list\n2. Insert a  
node\n3. Delete a node\n4. Display the list\n5.  
Exit\n"); scanf("%d",  
        &ch);  
  
        switch (ch) {  
            case 1:  
                printf("Enter the number of nodes to be  
created: "); scanf("%d", &n); head =  
                createList(head, n); break;  
            case 2:  
                printf("Enter the data for the node to be  
inserted: "); scanf("%d", &data); head =  
                insertNode(head, data); break;  
            case 3:
```

```

        printf("Enter the data of the node to be
deleted: "); scanf("%d", &data); head =
        deleteNode(head, data); break;

case 4:
        displayList(head); break;
case 5: printf("Exiting the
program.\n"); break;
default: printf("\nEnter a valid
input.\n"); break;
    } }

return 0;
}

```

Output :

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
1
Enter the number of nodes to be created: 3
Enter the data for node 1: 1
Enter the data for node 2: 2
Enter the data for node 3: 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
2
Enter the data for the node to be inserted: 12

Enter 1 for insertion at the beginning, 2 for insertion at an index, or 3 for insertion at the end: 3

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
3
Enter the data of the node to be deleted: 12
Node with data 12 deleted from the list.

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
4

```

Result :

The doubly linked list data structure was executed with all its functions user defined using C programming language.

Circular Linked List :

AIM:

To implement the Circular linked list data structure and its functionalities such as insertion , deletion and display with users input , using the C programming language.

Procedure :

1. Create a Circular Linked List:

- Initialize the head pointer as NULL.
- For each node to be added:
 - Create a new node.
 - Read data for the new node.
 - If the head is NULL (empty list):
 - Set the head to point to the new node.
 - Make the new node's next pointer point to itself.
 - Otherwise:
 - Traverse to the last node in the list.
 - Make the last node's next pointer point to the new node.

- Make the new node's next pointer point to the head (to maintain circularity).

2. Insert a Node:

- Read the data and position where the node should be inserted.
- Create a new node with the given data.
- If the position is the first node:
 - Make the new node's next pointer point to the head.
 - Set the head to point to the new node.
- Otherwise:
 - Traverse to the node at the given position.
 - Make the new node's next pointer point to the next node.
 - Make the previous node's next pointer point to the new node.

3. Delete a Node:

- Read the position of the node to be deleted. - If the list is empty, return an error message.
- If the position is the first node:
 - Traverse to the last node in the list.
 - Make the last node's next pointer point to the second node.
 - Set the head to point to the second node.
- Otherwise:
 - Traverse to the node at the given position.
 - Make the previous node's next pointer point to the next node.
 - Free the memory allocated for the deleted node.

4. Display the Circular Linked List:

- If the list is empty, print an appropriate message.
- Otherwise:
- Initialize a pointer to the head node.
- Repeat the following steps until the pointer reaches the head again:
- Print the data of the current node.
- Move the pointer to the next node.
- Print a newline character to complete the display.

Code :

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

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

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

```
struct node* createList(struct node* head, int n); struct  
node* insertNode(struct node* head, int data, int  
position); struct node* deleteNode(struct node* head,  
int position); void displayList(struct node* head);
```

```
int main() {  
    int n, data, position, ch = 0;  
    struct node* head = NULL;  
  
    while (ch != 5) {
```

```
printf("\nEnter\n1. Create a list\n2. Insert a node\n3.  
Delete a node\n4. Display the list\n5. Exit\n"); scanf("%d",  
&ch);
```

```
switch (ch) {
```

```
case 1:
```

```
printf("Enter the number of nodes to be created: ");  
scanf("%d", &n); head =  
createList(head, n); break;
```

```
case 2:
```

```
printf("Enter the data and position of the node to  
be inserted: "); scanf("%d%d", &data, &position);  
head = insertNode(head, data, position);  
break;
```

```
case 3:
```

```
printf("Enter the position of the node to be deleted:  
");  
scanf("%d", &position); head =  
deleteNode(head, position); break;
```

```
case 4:
```

```
displayList(head); break;
```

```
case 5: printf("Exiting the  
program.\n"); break;
```

```
default: printf("\nEnter a valid  
input.\n"); break;
```

```
} }
```

```
return 0;
```

```
}
```

```

struct node* createList(struct node* head, int n) {
    if (n <= 0) {
        printf("Invalid number of nodes.\n");
        return head; }

    struct node* prev = NULL;
    struct node* current = NULL;

    for (int i = 0; i < n; i++) {
        struct node* newNode = (struct
node*)malloc(sizeof(struct node)); printf("Enter
        the data for node %d: ", i + 1); scanf("%d",
        &(newNode->data));

        if (head == NULL) {
            head = newNode;
        } else { prev->next =
            newNode;
        }

        newNode->next = NULL;
        prev = newNode;

        if (i == n - 1) {
            newNode->next = head; // Make the last node point
to the head, creating a circular link.
        } }

```

```
    printf("Circular linked list created with %d nodes.\n", n);  
return head; }
```

```
struct node* insertNode(struct node* head, int data, int  
position) {
```

```
    if (position <= 0) {  
        printf("Invalid position.\n");  
        return head; }
```

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

```
    if (head == NULL) {  
        head = newNode; newNode->next = head; // Make the  
        new node point to  
itself since it is the only node.
```

```
        printf("Node inserted at position %d.\n", position);  
        return head;
```

```
    }  
    if (position == 1) { newNode-  
        >next = head; head =  
        newNode;
```

```
    } else { struct node* current =  
        head; struct node* prev =  
        NULL; int count = 1;
```



```
while (count < position && current->next != head) {  
    prev = current; current  
= current->next;  
count++; }  

```

```
if (count < position) {  
    printf("Invalid position. Node not inserted.\n");  
free(newNode); return head; }  

```

```
prev->next = newNode; newNode-  
>next = current;  
}
```

```
printf("Node inserted at position %d.\n", position);  
return head;  
}
```

```
struct node* deleteNode(struct node* head, int position)  
{ if (head == NULL) {  
    printf("The list is empty. Deletion not possible.\n");  
return head; }  

```

```
if (position <= 0) {  
    printf("Invalid position.\n");  
return head; }  

```

```

struct node* current = head;
struct node* prev = NULL;
int count = 1;

while (count < position && current->next != head) {
    prev = current; current
= current->next;
count++; }

if (count < position) {
    printf("Invalid position. Node not found.\n");
    return head;
}
if (prev == NULL) {
    // Deletion of the head node
    struct node* tail = head;
    while (tail->next != head) {
        tail = tail->next;
    } tail->next = head-
>next; head = head-
>next;
} else { prev->next = current-
>next;
}

free(current); printf("Node deleted from
position %d.\n", position); return head; }

```

```
void displayList(struct node* head)
{ if (head == NULL) {
    printf("The list is empty.\n");
    return; }

    struct node* current = head;

    printf("Circular linked list: ");
    do {
        printf("%d ", current->data);
        current = current->next; }
    while (current != head);
    printf("\n"); }
```

Output :

```
PS C:\Users\91912\Documents\DSA> cd "c:\Users\91912\Documents\DSA\" ; if ($?) { gcc circularlinkedlist.c -o circularlinkedlist }; if ($?) { .\circularlinkedlist
}

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
1
Enter the number of nodes to be created: 4
Enter the data for node 1: 1
Enter the data for node 2: 2
Enter the data for node 3: 7
Enter the data for node 4: 9
Circular linked list created with 4 nodes.

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
2
Enter the data and position of the node to be inserted: 12 2
Node inserted at position 2.

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
4
Circular linked list: 1 12 2 7 9

Enter
1. Create a list
4. Display the list
5. Exit
2
Enter the data and position of the node to be inserted: 12 2
Node inserted at position 2.

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
4
Circular linked list: 1 12 2 7 9

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
3
Enter the position of the node to be deleted: 4
Node deleted from position 4.

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
4
Circular linked list: 1 12 2 9

Enter
1. Create a list
2. Insert a node
3. Delete a node
4. Display the list
5. Exit
```

Result :

The circular linked list data structure was successfully executed.

Stack Using array :

AIM:

To implement the Stack data structure using the Arrays and implementing its various functions such as push , pop, and peek using the C programming language.

Procedure :

1. Initialize the Array with a specific size on the top of the program.
2. Start the main function.
3. Declare a variable n and initialize it to 0.
4. Enter a while loop with the condition $n \geq 0$ to continue until the user enters a negative number.
5. Print the menu options for the user: "1. push", "2. pop", "3. display".
6. Read the user's choice into the variable n.
7. Use a switch statement to perform the corresponding action based on the user's choice:
8. Case 1: Call the push function.
9. Case 2: Call the pop function.
10. Case 3: Call the display function.
11. Default: Print "Enter valid input" if the user enters an invalid choice.
12. End the switch statement.
13. End the while loop.
14. Return 0 to indicate successful execution of the program.
15. Implement the push function:
16. Increment the top variable.
17. Check if top is equal to 6 (indicating the stack is full):
18. If true, print "Overflow".
19. If false, continue.
20. Read an integer x from the user.
21. Store x in the stack array at the current top position.
22. Implement the pop function:
23. Declare a variable item to store the popped element.
24. Check if top is equal to -1 (indicating the stack is empty):
25. If true, print "Underflow".
26. If false, continue.
27. Assign the value at stack[top] to item.
28. Decrement the top variable.
29. Print "The item item was popped".
30. Implement the display function:
31. Start a loop from top to 0:
32. Print the value at stack[i].

Code :

```
#include<stdio.h>

int stack[6];
int top=-1;

void push();
void pop();
void display();

int main(){
    int n=0;
    while(n>=0){
        printf("1.push\n2.pop\n3.display\n");
        scanf("%d",&n); switch(n){ case 1:
            push(); break;
            case 2: pop(); break;
            case 3: display(); break;
            default:printf("Enter valid input\n");
        } }
    return 0; }

void push(){
    top++; int
    x;
```

```
if(top==6)
{
    printf("Overflow"); }
else{ printf("Enter the element :
\n"); scanf("%d",&x);
stack[top]=x;
} }
```

```
void pop(){ int
    item;
    if(top==-1){
        printf("Underflow\n");
    }

    else{
        item=stack[top];
        top--;
        printf("The item %d was popped\n",item);
    } }
```

```
void display(){

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

Output :

```
C:\Users\91912\Documents\D x + v
1.push
2.pop
3.display
1
Enter the element :
12
1.push
2.pop
3.display
1
Enter the element :
34
1.push
2.pop
3.display
1
Enter the element :
7
1.push
2.pop
3.display
3
7      34      12
1.push
2.pop
3.display
2
The item 7 was popped
1.push
2.pop
3.display
|
```

Result :

The Stack data structure using an array was successfully executed.

Stack using Linked list :

AIM:

To implement the Stackdata structure and its various functionalities such as push ,pop ,and display using linked list using C programming language.

Algorithm :

1. 1. Define a structure `node` with two fields: `data` to store the value and `link` to store the address of the next node.
2. 2. Implement two functions `push` and `pop` to add and remove elements from the stack, respectively.
3. 3. In the `main` function:
4. a. Create a pointer `head` and initialize it to `NULL`.
5. b. Create a new node `ptr` dynamically, set its `data` to 7, and `link` to `NULL`.
6. c. Assign the `ptr` to `head`.
7. d. Create another node `nptr`, set its `data` to 9, and `link` to the current `head`.
8. e. Assign `nptr` to `head`.
9. f. Prompt the user to choose between push or pop.
10. g. If the user selects push:
11. - Call the `push` function and pass `head` as an argument.
12. - Update `head` with the returned value from `push`.
13. h. Otherwise, if the user selects pop:
14. - Call the `pop` function and pass `head` as an argument.
15. - Update `head` with the returned value from `pop`.
16. i. Traverse the linked list starting from `head` and print the values.
- 17.
- 18.4. Implement the `push` function:
19. a. Create a new node `ptr` dynamically.
20. b. Prompt the user to enter the data value for the new node.
21. c. Assign the entered value to `ptr`'s `data`.
22. d. Set `ptr`'s `link` to the current `head`.
23. e. Update `head` with `ptr`.
24. f. Return `head`.
- 25.5. Implement the `pop` function:
26. a. Create a temporary node `temp` and assign `head` to it.
27. b. Update `head` with the address stored in `temp`'s `link`.
28. c. Free the memory occupied by `temp`.
29. d. Return `head`.

Code :

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data; struct
    node *link;
};

struct node *push(struct node *head);
struct node *pop(struct node *head);
int main(){
    struct node *head; struct node
    *ptr; ptr=malloc(sizeof(struct
    node)); ptr->data=7; ptr-
    >link=NULL; head=ptr; struct
    node *nptr;
    nptr=malloc(sizeof(struct node ));
    nptr->data=9; nptr->link=head;
    head=nptr;

    int in;
    printf("1.push\n2.pop\n");
    scanf("%d",&in);
    if(in==1){
        head=push(head); }
    else
    { head=pop(head); }
```

```

    struct node *temp;
    temp=head;
    while(temp !=
    NULL){
        printf("%d\t",temp->data); temp=temp->link;
    }
    return 0;
}

```

```

struct node *push(struct node *head){
    struct node *ptr; int n; printf("Enter
    the data for the node : ");
    scanf("%d",&n);
    ptr=malloc(sizeof(struct node)); ptr-
    >data=n; ptr->link=head; head=ptr;
    return head;
} struct node *pop(struct node
*head){
    struct node *temp;
    temp=head;
    head=head->link;
    free(temp); return
    head; }

```

Output :

Result :

The Stack data structure using Linked List was successfully executed.

Queue using array :

AIM: To implement Queue data structure using array implementing its basic functions such as enqueue , dequeue and display using C programming language.

Algorithm :

Declare global variables `front` and `rear` and initialize them to 0.

1. Declare a global array `queue` with a size of 5 and `num` set to 5 to represent the maximum number of elements in the queue.
2. Implement functions `queueisfull` and `queueisempty` to check if the queue is full or empty, respectively.
3. Implement functions `enqueue`, `dequeue`, and `display` to add elements to the queue, remove elements from the queue, and display the queue, respectively.
4. In the `main` function:
 - a. Declare a variable `n` and initialize it to 1.
 - b. Enter a while loop that continues as long as `n` is greater than 0.
 - c. Inside the loop, prompt the user to choose between enqueue, dequeue, display, or exit.
 - d. Use a switch statement to perform the appropriate action based on the user's input:
 - i. If the user chooses 1, call the `enqueue` function.
 - ii. If the user chooses 2, call the `dequeue` function.
 - iii. If the user chooses 3, call the `display` function.
 - iv. If the user chooses -1, exit the loop.
 - v. For any other input, display an error message.
5. Implement the `queueisfull` function:
 - a. Check if `rear` is equal to `num`.

- i. If true, return -1 to indicate that the queue is full.
 - ii. If false, return 1 to indicate that the queue is not full.
- 6. Implement the `'queueisempty'` function:
 - a. Check if `'front'` is equal to `'rear'` and `'front'` is not 0.
 - i. If true, return 1 to indicate that the queue is empty.
 - ii. If false, return -1 to indicate that the queue is not empty.
- 7. Implement the `'enqueue'` function:
 - a. Check if the queue is not full by calling the `'queueisfull'` function.
 - i. If true, prompt the user to enter a value and store it in a variable `'n'`.
 - ii. Assign `'n'` to `'queue[rear]'`.
 - iii. Increment `'rear'` by 1.
 - b. If the queue is full, display a message indicating that the queue is full.
- 8. Implement the `'dequeue'` function:
 - a. Check if the queue is not empty by calling the `'queueisempty'` function.
 - i. If true, display the element at the front of the queue (`'queue[front]'`). ii. Increment `'front'` by 1 to remove the element from the queue.
 - b. If the queue is empty, display a message indicating that the queue is empty.
- 9. Implement the `'display'` function:
 - a. Use a loop to iterate over the elements in the queue, starting from `'front'` and ending at `'rear'`.
 - b. Display each element of the queue (`'queue[i]'`).

Code :

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

```
int front=0;
int rear=0;
int queue[5];
int num=5;
```

```
int queueisfull(int num);
int queueisempty();
```

```
int enqueue();
int dequeue();
void display();
int main(){
    int n=1;
```

```
    while(n>0){
        printf("1.enqueue\n2.dequeue\n3.display\n-
1.exit\n"); scanf("%d",&n);
        switch(n){ case 1:
            enqueue(); break;
            case 2: dequeue();
            break; case 3:
            display(); break;

            default:printf("Invalid input !\n");
        }
    }
```

```
} }
```

```
int queueisfull(int num){  
    if (rear==num){  
        return -1;  
    }  
    else{ retur  
n 1;  
    } }
```

```
int queueisempty(){  
    if(front==rear && front !=0){  
        return 1; }  
    else{ return  
-1;  
    } }
```

```
int enqueue(){  
    if(queueisfull(num)==1){  
        printf("Enter the value\n");  
        int n; scanf("%d",&n);  
        queue[rear]=n;  
        rear=rear+1;  
  
    }  
    else{  
        printf("Queue is full!\n");
```

```

    } }

int dequeue(){
    if(queueisempty()==1){
        printf("The element %d is deleted\n",queue[front]);
        front++; }

    else{
        printf("Queue is empty\n");
    } }

void display(){
    for(int i=front;i<=rear;i++){
        printf("%d\n",queue[i]);
    }
}

```

Output :


```
1.enqueue
2.dequeue
3.display
-1.exit
1
Enter the value
12
1.enqueue
2.dequeue
3.display
-1.exit
1
Enter the value
7
1.enqueue
2.dequeue
3.display
-1.exit
3
12
7
0
1.enqueue
2.dequeue
3.display
-1.exit
2
Queue is empty
1.enqueue
2.dequeue
3.display
-1.exit
[]
```

Result :

The Queue data structure using array was successfully executed.

Queue using linked list :

AIM: To implement the Queue data structure using the linked list and its functions such as enqueue and dequeue programming language .

Algorithm :

node with fields data and next to store the value and the address of the next node, respectively.

- 1.Declare global pointers front and rear and initialize them to NULL.

ment functions enqueue, dequeue, and peek to add elements to the queue, remove elements from the queue, and display the queue, respectively. main function: a loop that continues until the user chooses to exit. the user for an option: enqueue, dequeue, display, or exit. on the user's choice, call the corresponding function or exit the loop. ment the enqueue function: a new node ptr dynamically using malloc. the user to enter a value and store it in ptr->data. ptr->next to NULL. queue is empty, set front and rear to ptr. queue is not empty, update rear->next to ptr and update rear to ptr. ment the dequeue function: queue is empty (front is NULL), display "Queue is empty". wise: a temporary node temp and set it to front. e front to front->next. ay the dequeued element temp->data. he memory allocated for temp. ment the peek function: queue is empty (front is NULL), display "Queue is empty". wise: a temporary node temp and set it to front. ay "The Queue is:". temp is not NULL, display temp->data and update temp to temp->next.

Code:

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

```

struct node
{
    int data;
    struct node *next;
};

struct node *front = NULL;

struct node *rear = NULL;

void enqueue();

void dequeue();

void peek();

int main()
{
    int n = 1;

    while (n <= 4) { printf("Enter:\n1. Enqueue\n2.
        Dequeue\n3. Display\n4.
        Exit\n"); scanf("%d",
        &n); switch (n) {
        case 1: enqueue();
            break;
        case 2: dequeue();
            break;
        case 3: peek();
            break;
        case 4:
            exit(0);
        default:
            printf("Enter a valid input\n");
        }
    }
}

void enqueue() {
    struct node *ptr; ptr =
    malloc(sizeof(struct node)); if
    (ptr == NULL) {
        printf("Memory allocation failed\n");
    }
}

```

```

        return; } int n;
printf("Enter the data: ");
scanf("%d", &n); ptr-
>data = n; ptr->next =
NULL;
if (front == NULL) {
    front = ptr;
    rear = ptr;
} else { rear->next
    = ptr; rear = ptr;
}
}

void dequeue() {
    if (front == NULL) {
        printf("Queue is empty\n");
    } else { struct node
        *temp; temp =
        front; front = front-
        >next;
        printf("Dequeued element: %d\n", temp->data);
        free(temp);
    }
}

void peek() {
    if (front == NULL) {
        printf("Queue is empty\n");
    } else { struct node *temp =
        front; printf("The Queue
        is: ");
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

```

Output :

```
Warning: PowerShell detected that you might be using a screen reader and has disabled PSReadLine for compatibility purposes. If you want to re-enable it, run 'Import-Module PSReadLine'.

PS C:\Users\91912\Documents\DSA> cd "c:\Users\91912\Documents\DSA\" ; if ($?) { gcc queue_using_linkedlist.c -o queue_using_linkedlist } ; if ($?) { .\queue_using_linkedlist }
Enter:
1. Enqueue
2. Dequeue
3. Display
4. Exit
1
Enter the data: 12
Enter:
1. Enqueue
2. Dequeue
3. Display
4. Exit
1
Enter the data: 7
Enter:
1. Enqueue
2. Dequeue
3. Display
4. Exit
1
Enter the data: 77
Enter:
1. Enqueue
2. Dequeue
3. Display
4. Exit
3
The Queue is: 12 7 77
Enter:
1. Enqueue
2. Dequeue
3. Display
4. Exit
2
Dequeued element: 12
Enter:
```

Result :

The Queue data structure using Linked List was successfully executed.

Circular Queue using array :

AIM: To implement the Circular queue data structure using array the C programming language.

Procedure :

- 1) Define the maximum size of the circular queue using #define.
- 2) Declare global variables front and rear and initialize them to -1.
- 3) Declare an array cirq to store the elements of the circular queue.

- 4) Implement functions insertion, deletion, and display to add elements to the circular queue, remove elements from the circular queue, and display the circular queue, respectively.
- 5) In the main function:
- 6) a. Enter a loop that continues until the user chooses to exit.
- 7) b. Prompt the user for an option: enqueue, dequeue, display, or exit.
- 8) c. Based on the user's choice, call the corresponding function or exit the loop.
- 9) Implement the insertion function:
- 10) a. Prompt the user to enter a number and store it in a variable n.
- 11) b. If $(\text{rear} + 1) \% \text{maxsize}$ is not equal to front, increment rear using modulo arithmetic and assign n to `cirq[rear]`.
- 12) c. If front is -1, set front to 0.
- 13) d. If $(\text{rear} + 1) \% \text{maxsize}$ is equal to front, display "Overflow".
- 14) Implement the deletion function:
- 15) a. If front is equal to -1, display "Underflow".
- 16) b. Otherwise, display the element deleted (`cirq[front]`), increment front using modulo arithmetic, and check if front becomes equal to $\text{rear} + 1$.
- 17) If true, set front and rear to -1.
- 18) Implement the display function:
- 19) a. If front is equal to -1, display "Circular queue is empty".
- 20) b. Otherwise, iterate over the elements in the circular queue starting from front and ending at $(\text{rear} + 1) \% \text{maxsize}$, and display each element (`cirq[i]`).

Code :

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

```
#define maxsize 5
```

```
int front=-1;
```

```
int rear=-1;
```

```
int cirq[5];
```

```
void insertion();
```

```
void deletion();
```

```
void display();
```

```
int main(){
```

```
    int n=0;
```

```
    while(n!=4){
```

```
        printf("1.enqueue\n2.dequeue\n3.display\n4.exit\n");
```

```
        scanf("%d",&n); switch (n) { case 1: insertion();
```

```
            break;
```

```
            case 2 : deletion();
```

```
            break; case 3 :
```

```
                display(); break;
```

```
            case 4 : printf("Exiting...\n");
```

```
                break;
```

```
            default: printf("Enter valid input \n");
```

```
                break;
```

```
        } }
```

```
    return 0; }
```

```
void insertion(){
```

```
int n; printf("Enter the number to be  
inserted : \n"); scanf("%d",&n);
```

```
if(rear == -1 && front == -1){  
    rear=0;  
    front=0;  
    cirq[rear]=n; }
```

```
else if(front==0 && rear != maxsize-1){  
    rear=(rear+1)%maxsize; cirq[rear]=n;  
}  
else if(front != 0 && rear==maxsize-1){  
    rear=(rear+1)%maxsize;  
    cirq[rear]=n; } else{  
    printf("Overflow\n");  
} }
```

```
void deletion(){  
    if(front == -1 && rear == -1){  
        printf("Underflow\n");  
  
    }  
  
    else if(front == rear){  
        printf("The element deleted is : %d\n",cirq[front]);  
        front=-1; rear=-1;
```

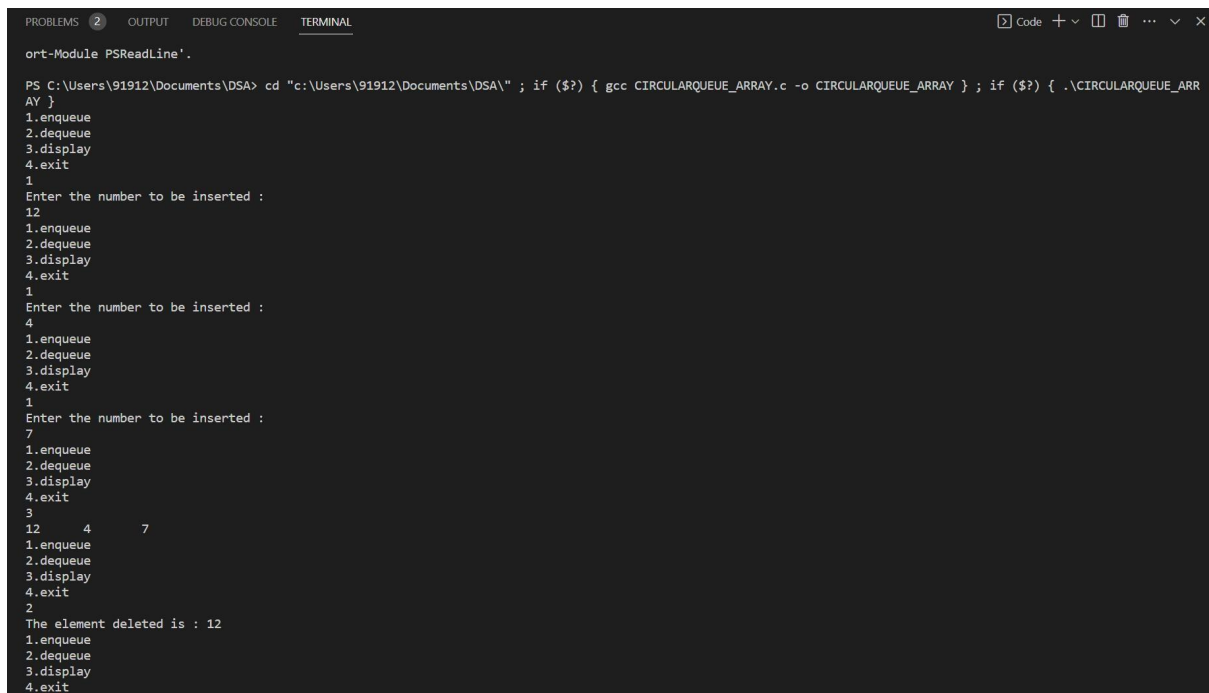


```
}
```

```
else if(front==maxsize-1){ printf("The element  
deleted is : %d\n",cirq[front]);  
front=(front+1)%maxsize;  
}  
else{  
printf("The element deleted is : %d\n",cirq[front]);  
front++;  
} }
```

```
void display(){  
if(front==-1 && rear==-1){  
printf("Circular queue is empty\n");  
return; } for(int i=front;i<=rear;i++){  
printf("%d\t",cirq[i]);  
}  
printf("\n"); }
```

Output :



```
ort-Module PSReadLine'.
PS C:\Users\91912\Documents\DSA> cd "c:\Users\91912\Documents\DSA\" ; if ($?) { gcc CIRCULARQUEUE_ARRAY.c -o CIRCULARQUEUE_ARRAY } ; if ($?) { .\CIRCULARQUEUE_ARRAY }
1.enqueue
2.dequeue
3.display
4.exit
1
Enter the number to be inserted :
12
1.enqueue
2.dequeue
3.display
4.exit
1
Enter the number to be inserted :
4
1.enqueue
2.dequeue
3.display
4.exit
1
Enter the number to be inserted :
7
1.enqueue
2.dequeue
3.display
4.exit
3
12    4    7
1.enqueue
2.dequeue
3.display
4.exit
2
The element deleted is : 12
1.enqueue
2.dequeue
3.display
4.exit
```

Result :

The Circular Queue data structure using array was successfully executed.

Double Ended Queue :

AIM: To implement the Double Endedqueue data structure and its functionality using C programming language.

Procedure :

1. Define the maximum size of the circular queue using #define.
2. Declare an array doq and variables front and rear, initialized to -1.
3. Implement insertion to read n and num, handle overflow conditions, and store num in doq[front] or doq[rear].

4. Implement deletion to read n, handle underflow conditions, and delete the element at doq[front] or doq[rear].
5. Implement display to print the elements of doq from front to rear.
6. Implement the main function with a loop that reads n and performs corresponding operations based on user input.
7. End the program.

Code :

```
#include<stdio.h>

#define maxsize 5

int doq[5];
int front=-1;
int rear=-1;

int insertion();
int deletion();
void display();

int
main(){ in
    t n=0;
    while(n!=4){

printf("1.enqueue\n2.dequeue\n3.display\n4.exit\n");
    scanf("%d",&n); switch (n)
    {
```

```

        case 1: insertion(); break;
        case 2 : deletion(); break;
        case 3 : display();
                break;
        case 4 : printf("Exiting...\n");
                break;
        default: printf("Enter valid input \n");
                break;
    } }
return 0; }

```

```

int insertion(){
    int n,num; printf("1.insertion from beginnng  \n2.
        insertion
from rear\n");
    scanf("%d",&n);

    printf("Enter the element to be inserted \n");
    scanf("%d",&num); if(n==1){ if(front== -1
    && rear== -1){
        front =0; rear=0;
        doq[front]=num
        ; }

    else if(front>0){ front=maxsize-1;
        doq[front]=num;
    } }
}

```

```

else{
    if(front==-1&&rear==-1){
        front=0; rear=0;
        doq[rear]=num; }

    else if (front==rear){
        printf("Overflow\n")
        ; } else {
        rear=(rear+1)%maxsize; doq[rear]=num;
        }
    }
}
int deletion(){
    int n; printf("1.deletion from beginning/n2.deletion
    from
end");
    scanf("%d",&n);
    if(n==1){
        if(front==-1 &&rear==-1){
            printf("Underflow\n");
        } else
        {
            printf("The element %d is deleted",doq[front]);
            front=(front+1)%maxsize;
        } }

    else{
        if(front==-1 &&rear==-1){

```

```

        printf("Underflow\n");
    }
    else{
        printf("the element %d is deleted",doq[rear]);
        rear=(rear-1)%maxsize;
    }
}
}
void display(){

    if(front==-1 && rear==-1){
        printf("Circular queue is empty\n");
        return; } for(int i=front;i<=rear;i++){
        printf("%d\t",doq[i]);
    }
    printf("\n"); }

```

Output :

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
Warning: PowerShell detected that you might be using a screen reader and has disabled PSReadLine for compatibility purposes. If you want to re-enable it, run 'Import-Module PSReadLine'.

PS C:\Users\91912\Documents\DSA> cd "c:\Users\91912\Documents\DSA\" ; if ($?) { gcc double_Queue.c -o double_Queue } ; if ($?) { .\double_Queue }
1.enqueue
2.dequeue
3.display
4.exit
1
1.insertion from begininng /n2. insertion from rear
12
Enter the element to be inserted
1
1.enqueue
2.dequeue
3.display
4.exit
1
1.insertion from begininng /n2. insertion from rear
2
Enter the element to be inserted
23
Overflow
1.enqueue
2.dequeue
3.display
4.exit
1
1.insertion from begininng /n2. insertion from rear
1
Enter the element to be inserted
23
1.enqueue
2.dequeue
3.display
4.exit
3
1
1.enqueue
2.dequeue
3.display
```

Result :

The Double Ended Queue data structure using array was successfully executed.

Priority Queue :

AIM : to implement the priority queue data structure in C programming language.

Procedure :

1. Define a structure Element with data and priority fields.
2. Declare a global array queue and initialize rear to -1.
3. Implement enqueue function to insert an element based on priority.
4. Implement dequeue function to remove and return the highest priority element.
5. Implement display function to print the elements in the priority queue.

6. In the main function, use a loop to interactively perform enqueue, dequeue, and display operations.
7. Terminate the loop when the user chooses to exit.

Code:

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

#define MAX_SIZE 100
typedef struct
{
    int data;
    int priority;
} Element;

Element queue[MAX_SIZE];
int rear = -1;

void enqueue(int data, int priority);
void dequeue(); void display();

int main() { int choice,
             data, priority;

    while (1) {
        printf("\nPriority Queue Operations:\n");
        printf("1. Enqueue\n"); printf("2.
        Dequeue\n"); printf("3. Display\n");
```



```

printf("4. Exit\n"); printf("Enter your
choice: "); scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter data to enqueue: ");
        scanf("%d", &data);
        printf("Enter priority: ");
        scanf("%d", &priority);
        enqueue(data, priority);
        break;
    case 2: dequeue();
        break;
    case 3: display();
        break;
    case 4:
        printf("Exiting...\n"); exit(0);
    default:
        printf("Invalid choice! Please enter a valid
option.\n");
} }

return 0; }

void enqueue(int data, int priority)
{ if (rear == MAX_SIZE - 1) {
    printf("Priority Queue is full. Cannot
enqueue.\n"); return;

```

```
}
```

```
int i = rear; while (i >= 0 &&  
queue[i].priority > priority) {  
    queue[i + 1] = queue[i];  
    i--;  
}
```

```
    queue[i + 1].data = data;  
    queue[i + 1].priority = priority;  
    rear++; }
```

```
void dequeue() {  
    if (rear == -1) {  
        printf("Priority Queue is empty. Cannot  
dequeue.\n");  
        return; }
```

```
    printf("Dequeued element: %d\n", queue[rear].data);  
    rear--;  
}
```

```
void display() {  
    if (rear == -1) {  
        printf("Priority Queue is empty.\n"); return;  
    }
```

```
printf("Priority Queue elements:\n");  
for (int i = rear; i >= 0; i--) {  
    printf("Data: %d\tPriority: %d\n", queue[i].data,  
queue[i].priority); }  
}
```

Output :

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
Warning: PowerShell detected that you might be using a screen reader and has disabled PSReadLine for compatibility purposes. If
you want to re-enable it, run 'Import-Module PSReadLine'.

PS C:\Users\91912\Documents\DSA> cd "c:\Users\91912\Documents\DSA\" ; if ($?) { gcc sample.c -o sample } ; if ($?) { .\sample }

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter data to enqueue: 12
Enter priority: 1

Priority Queue Operations:
1. Enqueue
2. Dequeue
Data: 3 Priority: 2
Data: 12 Priority: 1

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Priority Queue elements:
Data: 7 Priority: 3
Data: 3 Priority: 2
Data: 12 Priority: 1

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Priority Queue elements:
Data: 7 Priority: 3
Data: 3 Priority: 2

Enter your choice: 3
Priority Queue elements:
Data: 7 Priority: 3
Data: 3 Priority: 2
Data: 12 Priority: 1

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Priority Queue elements:
Data: 7 Priority: 3
Data: 3 Priority: 2
Data: 12 Priority: 1

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Priority Queue elements:
Data: 7 Priority: 3
Data: 3 Priority: 2
Data: 12 Priority: 1

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued element: 7

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Priority Queue elements:
Data: 3 Priority: 2
Data: 12 Priority: 1

Priority Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 5
```

Result :

The Priority Queue data structure using array was successfully executed.

Binary Tree :

AIM:

To implement the binary tree data structure and its functionality using C programming language.

Procedure :

1. Create a structure for a node with integer data and left and right pointers.
2. Define a function to create a new node with the given data and return it.
3. Define a function to insert a node into the binary tree by recursively traversing the tree based on the data value.
4. Define a function to search for a node with the given data by recursively traversing the tree.
5. Define a function for in-order traversal by recursively traversing the left subtree, printing the node's data, and then recursively traversing the right subtree.
6. In the main function, initialize the root pointer as NULL and perform insertions to build the binary tree.
7. Print the in-order traversal of the binary tree.
8. Optionally, perform a search for a specific data value and print whether it is found or not.

Code :

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

```
struct Node {
```

```

    int data; struct
    Node* left; struct
    Node* right;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data; newNode->left = NULL; newNode->right =
    NULL; return newNode;
}

struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) {
        root = createNode(data);
        return root; } else {
        struct Node* curr; if
        (data <= root->data) {
            curr = insertNode(root->left, data); root-
            >left = curr;
        }
        else {
            curr = insertNode(root->right, data); root-
            >right = curr;
        } return
        root;
    } } struct Node* searchNode(struct Node* root, int
data) {

```

```

    if (root == NULL || root->data == data)
        return root;
    else if (data < root->data)
        return searchNode(root->left, data);
    else
        return searchNode(root->right, data);
}

void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    } } int

main() {
    struct Node* root = NULL;
    root = insertNode(root, 50);
    insertNode(root, 30);
    insertNode(root, 20);
    insertNode(root, 40);
    insertNode(root, 70);
    insertNode(root, 60);
    insertNode(root, 80);

    printf("In-order traversal: "); inorderTraversal(root); int
    searchData = 40; struct Node* searchResult =
    searchNode(root, searchData); if (searchResult != NULL)

```

```
        printf("\n%d found in the tree.\n", searchData);
    else
        printf("\n%d not found in the tree.\n", searchData);

        return 0;
}
```

Output :

```
Warning: PowerShell detected that you might be using a screen reader and has disabled PSReadLine for compatibility purposes. If you want to re-enable it, run 'Import-Module PSReadLine'.

PS C:\Users\91912\Documents\DSA> cd "c:\Users\91912\Documents\DSA\" ; if ($?) { gcc sample.c -o sample } ; if ($?) { .\sample }
In-order traversal: 20 30 40 50 60 70 80
40 found in the tree.
PS C:\Users\91912\Documents\DSA>
```


Result :

The Binary tree data structure was successfully executed.

Binary Search Tree :

AIM: To implement the BST and its basic functions using C programming language.

PROCEDURE:

1. Define a structure for a node with integer data and left and right pointers.
2. Define a function to create a new node with the given data and return it.
3. Define a function to insert a node into the binary search tree by recursively traversing the tree based on the data value.
4. Define a function to perform an in-order traversal of the tree by recursively traversing the left subtree, printing the node's data, and then recursively traversing the right subtree.
5. In the main function, create an empty root pointer.
6. Prompt the user to enter the number of elements they want to insert into the tree.
7. Prompt the user to enter the elements one by one and insert them into the tree using the insertNode() function.
8. Print the in-order traversal of the tree.

Code :

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

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

```
struct Node { int data;  
    struct Node* left;  
    struct Node* right;  
};
```

```
struct Node* createNode(int data) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    newNode->data = data; newNode->left = NULL; newNode->right =  
    NULL; return newNode;  
}
```

```
struct Node* insertNode(struct Node* root, int data) {  
    if (root == NULL) {  
        root = createNode(data);  
    }  
    return root; } else {  
    if (data <= root->data) {  
        root->left = insertNode(root->left, data);  
    } else  
    {  
        root->right = insertNode(root->right, data);  
    }  
    return  
    root;  
} }
```

```

struct Node* searchNode(struct Node* root, int data) {
    if (root == NULL || root->data == data)
        return root; else if
    (data < root->data) return
    searchNode(root->left,
    data);

    else
        return searchNode(root->right, data);
}

```

```

void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    } }

```

```

int main() {
    struct Node* root = NULL; int
    numElements, data; printf("Enter the
    number of elements: "); scanf("%d",
    &numElements);

    printf("Enter the elements:\n"); for
    (int i = 0; i < numElements; i++) {
        scanf("%d", &data); root =
        insertNode(root, data);
    }
}

```

```
}

printf("In-order traversal: "); inorderTraversal(root);
int searchData; printf("\nEnter a
value to search: "); scanf("%d",
&searchData);
Struct Node* searchResult = searchNode(root, searchData);
if (searchResult != NULL)
    printf("%d found in the tree.\n", searchData);
else
    printf("%d not found in the tree.\n", searchData);

return 0;
}
```

Output :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Enter the number of elements: 3
Enter the elements:
1
2
3
In-order traversal: 1 2 3
Enter a value to search: 3
3 found in the tree.
PS C:\Users\91912\Documents\DSA> 
```

Result :

The Binary Search tree data structure was successfully executed.

AVLTree :

AIM: To implement the AVL tree data structure along with its functions such as insertin , deletion, and rotation using C programming language .

Procedure :

1. Define a structure for a node with integer data, left and right pointers, and height.
2. Implement functions to get the maximum of two integers, get the height of a node, get the balance factor of a node, create a new node, perform a right rotation, perform a left

- rotation, insert a node into the AVL tree while maintaining balance, find the minimum value node, delete a node from the AVL tree while maintaining balance, and print the tree in-order.
3. In the main function, create an empty root pointer.
 4. Prompt the user to enter the number of elements they want to insert into the tree.
 5. Prompt the user to enter the elements one by one and insert them into the tree using the insertNode() function.
 6. Print the in-order traversal of the tree.
 7. Prompt the user to enter a value to delete from the tree.
 8. Delete the specified value from the tree using the deleteNode() function and print the updated in-order traversal of the tree.

Code :

```
#include <stdio.h>

#include <stdlib.h>

struct Node {
    int data; struct
    Node* left; struct
    Node* right; int
    height; };

int max(int a, int b); int getHeight(struct Node*
node); int getBalance(struct Node* node); struct
Node* createNode(int data); struct Node*
rotateRight(struct Node* y); struct Node*
rotateLeft(struct Node* x); struct Node*
insertNode(struct Node* root, int data); struct Node*
deleteNode(struct Node* root, int data); struct
```

```
Node* minValueNode(struct Node* node); void  
printTree(struct Node* root);
```

```
int max(int a, int b) {  
    return (a > b) ? a : b;  
}  
int getHeight(struct Node* node) {  
    if (node == NULL)  
        return 0;  
    return node->height;  
}
```

```
int getBalance(struct Node* node) {  
    if (node == NULL)  
        return 0;  
    return  getHeight(node->left)  -  getHeight(node->right);  
}
```

```
struct Node* createNode(int data) {  
    struct    Node*    newNode    =    (struct  
Node*)malloc(sizeof(struct Node));  
    newNode->data = data;  
    newNode->left = NULL;  
    newNode->right = NULL;  
    newNode->height = 1;  
    return newNode;  
}
```

```

struct Node* rotateRight(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;
    x->right = y; y->left = T2;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1; x->height = max(getHeight(x->left),
        getHeight(x->right)) + 1;
    return x; }

```

```

struct Node* rotateLeft(struct Node* x) {
    struct Node* y = x->right; struct Node* T2 = y->left; y->left = x; x->right = T2; x->height =
        max(getHeight(x->left), getHeight(x->right)) + 1; y->height = max(getHeight(y->left),
        getHeight(y->right)) + 1;
    return y; }

```

```

struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL)
        return createNode(data);
    if (data < root->data)
        root->left = insertNode(root->left, data);
    else if (data > root->data)
        root->right = insertNode(root->right, data);
    else

```



```

        return root;
    root->height = 1 + max(getHeight(root->left),
getHeight(root->right));
    int balance = getBalance(root); if
    (balance > 1 && data < root->left->data)
    return rotateRight(root);
    if (balance < -1 && data > root->right->data)
        return rotateLeft(root);
    if (balance > 1 && data > root->left->data)
        { root->left = rotateLeft(root->left); return
        rotateRight(root);
    } if (balance < -1 && data < root->right->data)
    {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    } return
root; }

```

```

struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current; }

```

```

struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL)
        return root;
    if (data < root->data)

```

```

        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else { if ((root->left == NULL) ||
              (root->right ==
NULL)) { struct Node* temp = root->left ? root->left :
root->right;
        if (temp == NULL) {
            temp = root;
            root = NULL; }
        else
            *root = *temp; free(temp); } else { struct
Node* temp = minValueNode(root-
>right); root->data = temp-
>data;
        root->right = deleteNode(root->right, temp-
>data);
    } } if (root ==
NULL) return
root;
    root->height = 1 + max(getHeight(root->left),
getHeight(root->right));
    int balance = getBalance(root); if (balance > 1
&& getBalance(root->left) >= 0) return
rotateRight(root);
    if (balance > 1 && getBalance(root->left) < 0)
        { root->left = rotateLeft(root->left); return
rotateRight(root);

```

```

    } if (balance < -1 && getBalance(root->right) <=
0) return rotateLeft(root);
if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rotateRight(root->right);
    return rotateLeft(root);
} return
root; }

```

```

void printTree(struct Node* root) {
    if (root != NULL) {
        printTree(root->left);
        printf("%d ", root->data);
        printTree(root->right);
    } }

```

```

int main() {
    struct Node* root = NULL;
    int numElements, data;
    printf("Enter the number of
elements: "); scanf("%d",
&numElements);

    printf("Enter the elements:\n"); for
(int i = 0; i < numElements; i++) {
        scanf("%d", &data); root =
insertNode(root, data);
    }
}

```

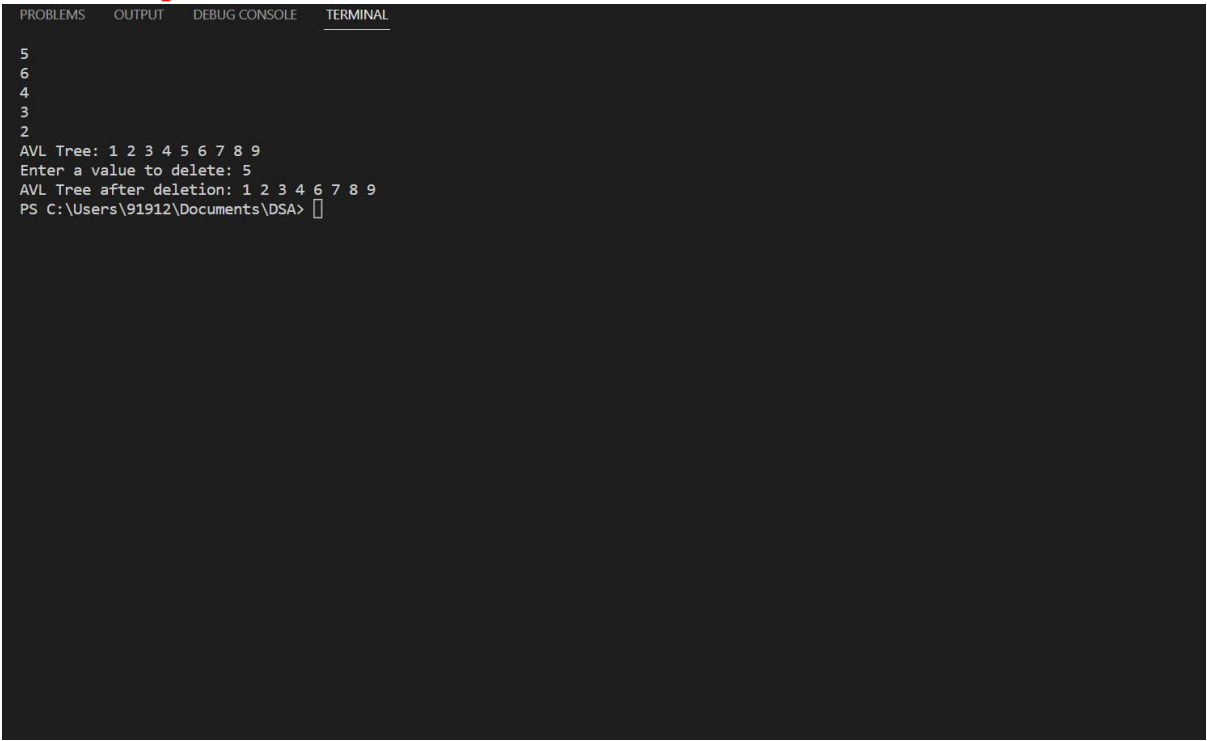
```
printf("AVL Tree: "); printTree(root);
```

```
int deleteData; printf("\nEnter a  
value to delete: "); scanf("%d",  
&deleteData);
```

```
root = deleteNode(root, deleteData);  
printf("AVL Tree after deletion: ");  
printTree(root);
```

```
return 0; }
```

Output :



```
5  
6  
4  
3  
2  
AVL Tree: 1 2 3 4 5 6 7 8 9  
Enter a value to delete: 5  
AVL Tree after deletion: 1 2 3 4 6 7 8 9  
PS C:\Users\91912\Documents\DSA> 
```

Result :

The AVL Tree data structure was successfully executed.

Binary Heap :

AIM : To implement the binary heap data structure and its functionality using C programming language.

Procedure :

1. Create a binary heap array of integers with a given capacity.
2. Insert elements into the heap using the insert function.
3. Extract the minimum value from the heap using the extractMin function.
4. To insert an element, add it to the end of the heap array and perform the heapify-up operation.
5. To extract the minimum value, swap it with the last element, reduce the heap size, and perform the heapify-down operation.
6. The heapify-up operation compares the inserted element with its parent and swaps them if necessary until the heap property is satisfied.
7. The heapify-down operation compares the root element with its children and swaps it with the smallest child if necessary until the heap property is satisfied.
8. Repeat steps 2 and 3 until the heap is empty or the desired condition is met.

Code :

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

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

```
struct BinaryHeap {  
    int* heap;  
    int size; int  
    capacity;  
};
```

```
struct BinaryHeap* createHeap(int capacity) {  
    struct BinaryHeap* heap = (struct  
BinaryHeap*)malloc(sizeof(struct BinaryHeap)); heap-  
>heap = (int*)malloc(capacity * sizeof(int)); heap->size =  
0; heap->capacity = capacity; return heap; }
```

```
int parent(int index) {  
    return (index - 1) / 2;  
}
```

```
int leftChild(int index) {  
    return 2 * index + 1;  
}
```

```
int rightChild(int index) {  
    return 2 * index + 2;  
}
```

```

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

```

```

void heapifyUp(struct BinaryHeap* heap, int index) {
    while (index > 0 && heap->heap[index] < heap-
>heap[parent(index)]) {
        swap(&heap->heap[index],                &heap-
>heap[parent(index)]);
        index = parent(index);
    } }

```

```

void heapifyDown(struct BinaryHeap* heap, int index) {
    int smallest = index; int left
    = leftChild(index); int right
    = rightChild(index);

    if (left < heap->size && heap->heap[left] < heap-
>heap[smallest])
        { smallest = left;
        }

    if (right < heap->size && heap->heap[right] < heap-
>heap[smallest])
        { smallest = right;
        }

    if (smallest != index) {

```

```
        swap(&heap->heap[index], &heap->heap[smallest]);
        heapifyDown(heap, smallest);
    } }
```

```
void insert(struct BinaryHeap* heap, int value) {
    if (heap->size == heap->capacity) {
        printf("Heap is full. Cannot insert more elements.\n");
        return; }

    heap->heap[heap->size] = value;
    heapifyUp(heap, heap->size);
    heap->size++; }
```

```
int extractMin(struct BinaryHeap* heap) {
    if (heap->size == 0) {
        printf("Heap is empty. Cannot extract
        minimum element.\n"); return -1; }
```

```
    int min = heap->heap[0]; heap->heap[0] =
    heap->heap[heap->size - 1]; heap->size--;
    heapifyDown(heap, 0); return min; }
```

```
int main() {
    int capacity, n, element;

    printf("Enter the capacity of the heap: ");
    scanf("%d", &capacity);
    struct BinaryHeap* heap = createHeap(capacity);
```



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

```
printf("Enter the elements:\n");
for (int i = 0; i < n; i++)
{ scanf("%d", &element);
insert(heap, element);
}
```

```
printf("Minimum values in the heap:\n");
while (heap->size > 0) { int min =
extractMin(heap); printf("%d\n", min);
}
```

```
free(heap->heap); free(heap);
```

```
return 0;
```

```
}
```

Output :

```
Enter the elements:
1
2
3
Minimum values in the heap:
1
2
3
PS C:\Users\91912\Documents\DSA> █
```

Result :

The Binary heap data structure was successfully executed.

Heap sort :

AIM: To implement the Heap sort data structure and its fuctions using C programming language.

Procedure:

- Build a max heap from the input array using the heapify function.
- Starting from the last element, swap it with the root element and reduce the heap size by 1.
- Perform the heapify operation on the reduced heap.
- Repeat step 2 and step 3 until the heap size is 1. The array is now sorted in ascending order.

Code :

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

```
void swap(int* a, int* b); void
heapify(int arr[], int n, int i);
void heapSort(int arr[], int n);
```

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

```
void heapify(int arr[], int n, int i)
{ int largest = i; int left = 2 * i +
1;
int right = 2 * i + 2;
```

```
if (left < n && arr[left] > arr[largest])
```

```
    largest = left;
```

```
if (right < n && arr[right] > arr[largest])
```

```
    largest = right;
```

```
if (largest != i) {
```

```
    swap(&arr[i], &arr[largest]); heapify(arr,  
    n, largest);
```

```
} }
```

```
void heapSort(int arr[], int n) {
```

```
    for (int i = n / 2 - 1; i >= 0; i--) heapify(arr,  
    n, i);
```

```
    for (int i = n - 1; i >= 0; i--) {
```

```
        swap(&arr[0], &arr[i]); heapify(arr,  
        i, 0);
```

```
    } }
```

```
int main()
```

```
{ int n;
```

```
    printf("Enter the
```

```
number of
```

```
elements:
```

```
");
```

```
scanf("%d",
&n);

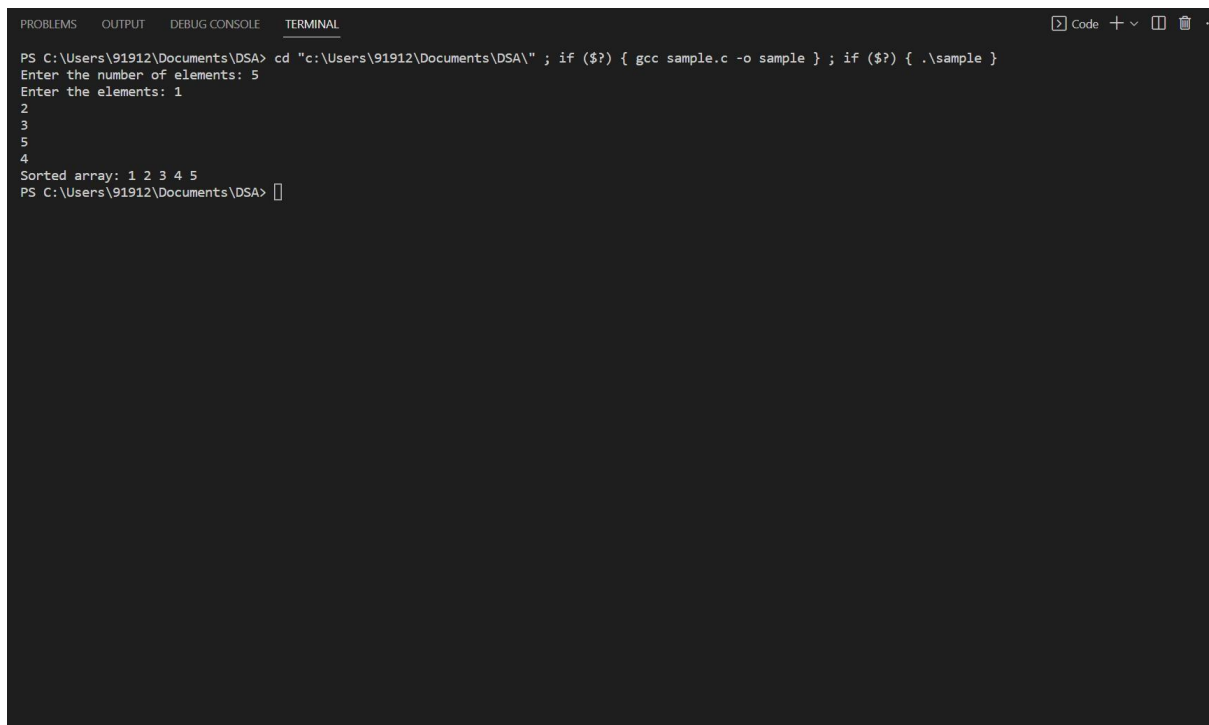
int arr[n]; printf("Enter the
elements: "); for (int i = 0; i <
n; i++) scanf("%d", &arr[i]);

heapSort(arr, n);

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

return 0; }
```

Output:



The image shows a terminal window with a dark background. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL', with 'TERMINAL' being the active tab. The terminal displays the following text:

```
PS C:\Users\91912\Documents\DSA> cd "c:\Users\91912\Documents\DSA\" ; if ($?) { gcc sample.c -o sample } ; if ($?) { .\sample }
Enter the number of elements: 5
Enter the elements: 1
2
3
5
4
Sorted array: 1 2 3 4 5
PS C:\Users\91912\Documents\DSA> 
```

Result:

The Heap sort data structure was successfully executed.