

DATA STRUCTURE

Experiment No. 1:

Aim:

1. Implementation of Binary Search.
2. Implementation of Selection & Quick Sort [Menu Driven Program].

BINARY SEARCH

AIM:

To search an element in array using binary search.

THEORY:

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

ALGORITHM:

A-ARRAY

N-Array size

X-value to be search

SET end=n-1; beg=0;

SET mid=(beg+end)/2

WHILE: beg<=end

 If a[mid]>x

 Set end=mid-1;

 If a[mid]==x;

 Value found at position mid+1;

 If a[mid]<x

 Set beg=mid+1;

END While loop

If beg>end

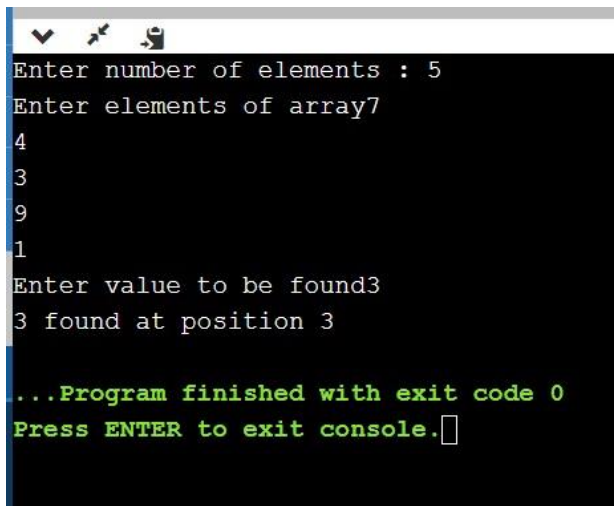
Value not present;

CODE:

```
#include <stdio.h>
int main()
```

```
{
int i, beg,end, mid, n,num,a[10];
printf("Enter number of elements : ");
scanf("%d",&n); printf("Enter
elements of array"); for(i = 0; i < n;
i++) scanf("%d",&a[i]); printf("Enter
value to be found");
scanf("%d",&num);
beg = 0; end = n - 1;
mid = (beg+end)/2;
while (beg <=end)
{
if(a[mid] < num) beg=
mid + 1;
else if (a[mid] ==num)
{
printf("%d found at position %d",num, mid+1); break;
}
else end =
mid - 1;
mid = (beg+end)/2;
}
if(beg>end)
printf(" %d is not present in the array",num);
return 0;
}
```

OUTPUT:



```
Enter number of elements : 5
Enter elements of array
4
3
9
1
Enter value to be found3
3 found at position 3

...Program finished with exit code 0
Press ENTER to exit console.
```

CONCLUSION:

Hence the number to be searched can be found using binary search.

SELECTION SORT

AIM:

To sort an array using selection sort.

THEORY:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

ALGORITHM:

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list **Step 3** –
Swap with value at location MIN

Step 4 – Increment MIN to point to next element

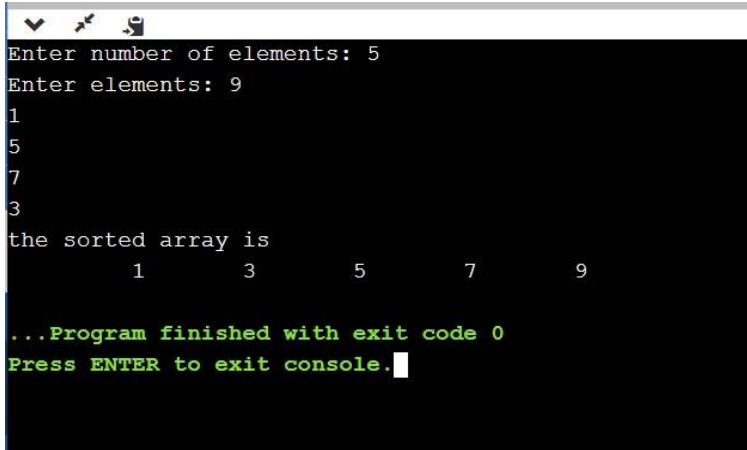
Step 5 – Repeat until list is sorted

CODE:

```
#include <stdio.h>
int main ()
{
    int n, i, j, iMin, temp, a[20];
    printf ("Enter number of elements: ");
    scanf ("%d", &n);
    printf ("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
    for (i = 0; i < n - 1; i++)
    {
        iMin = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[j] < a[iMin])
                iMin = j;
        }
        temp = a[i];
        a[i] = a[iMin];
        a[iMin] = temp;
    }
    printf ("the sorted array is \n");
```

```
for (i = 0; i < n; i++)  
printf ("\t %d", a[i]);  
return 0;  
}
```

OUTPUT:



```
Enter number of elements: 5  
Enter elements: 9  
1  
5  
7  
3  
the sorted array is  
1    3    5    7    9  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

CONCLUSION:

Selection sort is sorting algorithm known by its simplicity. Unfortunately, it lacks efficiency on huge lists of items, and also, it does not stop unless the number of iterations has been achieved even though the list is already sorted.

QUICK SORT

AIM: To sort an array using the Quick Sort .

THEORY:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worstcase complexity are $O(n \log n)$ and $O(n^2)$, respectively. On the basis of Divide and Conquer approach, quicksort algorithm can be explained as:

Divide :The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right.

Conquer :The left and the right subparts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.

Algorithm:

PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0

Step 2: Repeat Steps 3 to 6 while FLAG = 0

Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
SET RIGHT = RIGHT-1

[END OF LOOP]

Step 4: IF LOC = RIGHT

SET FLAG=1

ELSE IF ARR[LOC] > ARR[RIGHT]

SWAP ARR[LOC] with ARR[RIGHT]

SET LOC = RIGHT

[END OF IF]

Step 5: IF FLAG = 0

Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT

SET LEFT = LEFT+1

[END OF LOOP]

Step 6: IF LOC = LEFT

SET FLAG=1

ELSE IF ARR[LOC] < ARR[LEFT]

SWAP ARR[LOC] with ARR[LEFT]

SET LOC = LEFT

[END OF IF]

[END OF IF]

Step 7: [END OF LOOP]

Step 8: END

QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)

CALL PARTITION (ARR, BEG, END, LOC)

CALL QUICKSORT (ARR, BEG, LOC-1)

CALL QUICKSORT (ARR, LOC+1, END)

[END OF IF]

Step 2: END

CODE:

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

```
#include<conio.h>
```

```
int partition (int a[], int start, int end);
```

```
void quick_sort (int a[], int start, int end);
```

```
int main ()
```

```
{
```

```
int a[100], n, i;
```

```
//clrscr();
```

```
printf ("Enter number of elements : ");
```

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

```
for (i = 0; i <= n - 1; i++)
```

```
{
```

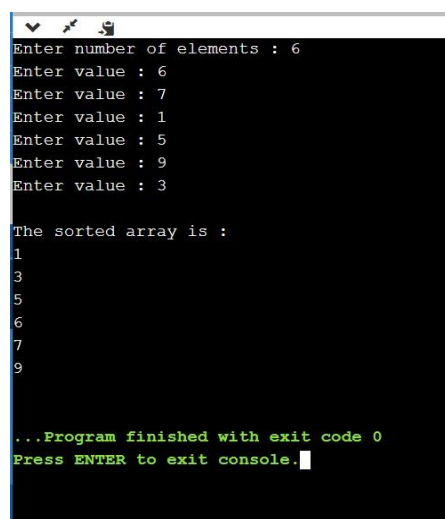
```
printf ("Enter value : ");
```

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

```
}
```

```
quick_sort (a, 0, n - 1);
printf ("\nThe sorted array is :\n");
for (i = 0; i <= n - 1; i++)
{
    printf ("%d\n", a[i]);
}
//getch();
}
int partition (int a[], int start, int end)
{
    int i, pivot = a[end], pindex = start, temp;
    for (i = start; i <= end - 1; i++)
    {
        if (a[i] < pivot)
        {temp = a[i];
        a[i] = a[pindex];
        a[pindex] = temp;
        pindex++;}}
    temp = a[pindex];
    a[pindex] = a[end];
    a[end] = temp;
    return pindex;}
void quick_sort (int a[], int start, int end)
{int pindex;
    if (start >= end)
        return;
    pindex = partition (a, start, end);
    quick_sort (a, start, pindex - 1);
    quick_sort (a, pindex + 1, end);
}
```

OUTPUT:



```
Enter number of elements : 6
Enter value : 6
Enter value : 7
Enter value : 1
Enter value : 5
Enter value : 9
Enter value : 3

The sorted array is :
1
3
5
6
7
9

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: Hence Quick sort is implemented.

Experiment No. 2:

Aim:

1. Implementation of Queues using Arrays.
2. Implementation of Circular Queues using Arrays.

Theory:

1. Queues:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

ALGORITHM:

Algorithm to insert an element in a queue:-

```
Step 1: IF REAR = MAX-1
        Write OVERFLOW
        Goto step 4
    [END OF IF]
Step 2: IF FRONT=-1 and REAR=-1
        SET FRONT = REAR =0
    ELSE
        SET REAR = REAR+1
    [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to delete an element from a queue:-

```
Step 1: IF FRONT = -1 OR FRONT > REAR
        Write UNDERFLOW
    ELSE
        SET VAL = QUEUE[FRONT]
        SET FRONT = FRONT+1
    [END OF IF]
Step 2: EXIT
```

CODE:

```
#include <stdio.h>
#define MAX 50
void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
Int main()
{
    int choice;
    while (1)
    {
        printf("1.Insert element to queue \n");
```

```
printf("2.Delete element from queue \n");
printf("3.Display all elements of queue \n");
printf("4.Quit \n");
printf("Enter your choice : ");
scanf("%d", &choice);
switch (choice)
{
    case 1:
        insert();
        break;
    case 2:
        delete();
        break;
    case 3:
        display();
        break;
    case 4:
        exit(1);
    default:
        printf("invalid \n");
}
}
}

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

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



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

OUTPUT:

```
Enter your choice : 1
Inset the element in queue : 8
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 9
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
6Inset the element in queue :
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 5
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
8 9 6
```

Conclusion: Circular queue have been inserted successfully using arrays.

Circular Queues

THEORY:

A circular queue in C stores the data in a very practical manner. It is a linear data structure. It is very similar to the queue. The only difference is that the last node is connected back to the first node. Thus it is called a circular queue.

ALGORITHM:

Algorithm to insert an element in a circular queue:-

Step 1: IF FRONT = and Rear = MAX-1

Write OVERFLOW

Goto step 4

[END OF IF]

Step 2: IF FRONT=-1 and REAR=-1

SET FRONT = REAR =0

```
        ELSE IF REAR = MAX-1 and FRONT !=0
            SET REAR =0
        ELSE
            SET REAR = REAR+1
        [End OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
Algorithm to delete an element from a circular queue:-
Step 1: IF FRONT=-1
        Write UNDERFLOW
        Goto Step 4
    [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
        SET FRONT = REAR=-1
    ELSE
        IF FRONT = MAX -1
            SET FRONT =0
        ELSE
            SET FRONT = FRONT+1
        [END of IF]
    [END OF IF]
Step 4: EXIT
```

CODE:

```
#include <stdio.h>
#define size 20
void insert(int[], int);
void delete(int[]);
void display(int[]);
int front = - 1;
int rear = - 1;
int main()
{
    int n, ch;
    int queue[size];
    do
    {
        printf("\n Circular Queue:\n1. Insert \n2. Delete\n3. Display\n 4. Exit");
        printf("\nEnter Choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("Enter number: ");
                scanf("%d", &n);
                insert(queue, n);
                break;
            case 2:
                delete(queue);
                break;
```

```
        case 3:
            display(queue);
            break;
        }
    }while (ch != 4);
}
void insert(int queue[], int item)
{
    if ((front == 0 && rear == size - 1) || (front == rear + 1))
    {
        printf("queue is full");
        return;
    }
    else if (rear == - 1)
    {
        rear++;
        front++;
    }
    else if (rear == size - 1 && front > 0)
    {
        rear = 0;
    }
    else
    {
        rear++;
    }
    queue[rear] = item;
}

void display(int queue[])
{
    int i;
    if (front > rear)
    {
        for (i = front; i < size; i++)
        {
            printf("%d ", queue[i]);
        }
        for (i = 0; i <= rear; i++)
            printf("%d ", queue[i]);
    }
    else
    {
        for (i = front; i <= rear; i++)
            printf("%d ", queue[i]);
    }
}

void delete(int queue[])
{
    if (front == - 1)
    {
        printf("Queue is empty ");
    }
}
```

```
else if (front == rear)
{
    printf(" %d deleted", queue[front]);
    front = - 1;
    rear = - 1;
}
else
{
    printf(" %d deleted", queue[front]);
    front++;
}
}
```

```
Circular Queue:
1. Insert
2. Delete
3. Display
4. Exit
Enter Choice : 1
Enter number: 5

Circular Queue:
1. Insert
2. Delete
3. Display
4. Exit
Enter Choice : 1
Enter number: 8

Circular Queue:
1. Insert
2. Delete
3. Display
4. Exit
Enter Choice : 1
Enter number: 9

Circular Queue:
1. Insert
```

```
Circular Queue:
1. Insert
2. Delete
3. Display
4. Exit
Enter Choice : 2
5 deleted
Circular Queue:
1. Insert
2. Delete
3. Display
4. Exit
Enter Choice : 3
8 9

Circular Queue:
1. Insert
2. Delete
3. Display
4. Exit
```

Conclusion: Circular queue have been inserted successfully using arrays.

Experiment No. 3:

Aim: Implementation of Linked List.

THEORY:

The first node is called the head; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the tail, points to *NULL* which helps us in determining when the list ends.

- 1) A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is *NULL*. Each node in a list consists of at least two parts: data , Pointer (Or Reference) to the next node.

ALGORITHM:

INSERT A NEW NODE BEFORE A NODE HAVING VALUE NUM

STEP 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL-> NEXT

Step 4: SET NEW_NODE->DATA = VAL

Step 5: SET PTR = START

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while PTR-> DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR ->NEXT

[END OF LOOP]

Step 10 : PREPTR ->NEXT = NEW_NODE

Step 11: SET NEW_NODE-> NEXT = PTR

Step 12: EXIT

INSERT A NODE AFTER A GIVEN NODE HAVING VALUE NUM

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL-> NEXT

Step 4: SET NEW_NODE-> DATA = VAL

Step 5: SET PTR = START

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while PREPTR-> DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR ->NEXT

[END OF LOOP]

Step 10 : PREPTR- > NEXT =NEW_NODE

Step 11: SET NEW_NODE ->NEXT = PTR

Step 12: EXIT

SEARCH

Step 1: [INITIALIZE] SET PTR = START

Step 2: Repeat Step 3 while PTR != NULL

Step 3: IF VAL = PTR-> DATA

SET POS = PTR

Go to Step 5

ELSE

SET PTR = PTR -> NEXT

[END OF IF]

[END OF LOOP]

Step 4: SET POS = NULL

Step 5: EXIT

DELETE A NODE

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 1

[END OF IF]

Step 2: SET PTR = START

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA! = NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR-> NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

Step 8: SET PREPTR -> NEXT = PTR -> NEXT

Step 9: FREE TEMP

Step 10 : EXIT

CODE:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
} *start = NULL;
void create ()
{
    int val, n, i;
    char c = 0;
    struct node *temp, *newnode;
    start = (struct node *) malloc (sizeof (struct node));
    temp = start;
    printf
        ("Creating linked list\nEnter number of elements you want in the linked list.\n");
    scanf (" %d", &n);
    for (i = 0; i < n; i++)
    {
        printf ("Enter value: ");
        scanf ("%d", &val);
        if (temp == start)
        {
            start->data = val;
        }
        else
            newnode->data = val;
        if (i != n)
        { newnode = (struct node *) malloc (sizeof (struct node));
          temp->next = newnode;
          temp = newnode;}}
    temp->next = NULL;
}
void display ()
{
    if (start != NULL)
    {
        struct node *temp = start;
        printf ("Display linked list\n");
```

```
        while (temp->next != NULL)
    {
        printf ("%d, ", temp->data);
        temp = temp->next;
    }
    printf ("%d.", temp->data);
    }
    else
        printf ("Linked List is not created");
    }

void search ()
{
    if (start != NULL)
    {
        int val, i = 0, flag;
        struct node *temp = start;
        printf ("Enter the value to be searched:\n");
        scanf ("%d", &val);
        while (temp->next != NULL)
    {
        if (temp->data == val)
        {
            printf ("%d is the %d number element in the linked list.\n ",
                val, i + 1);
            flag = 0;
            break;
        }
        else
        {
            flag = 1;
        }
        i++;
        temp = temp->next;
    }
    if (flag == 1)
    {
        printf ("%d not found.\n", val);
    }
    }
    else
        printf ("Linked List is not created");
    }

void insertafter ()
{
    if (start != NULL)
    {
        int val;
        struct node *temp = start, *newnode;
```



```
    printf ("Inserting After an element\n");
    printf ("Enter the value after which you want to insert: ");
    scanf ("%d", &val);
    while (temp->next != NULL && temp->data != val)
    {
        temp = temp->next;
    }
    if (temp->data != val)
    printf ("value not found");
    else
    {
        newnode = (struct node *) malloc (sizeof (struct node));
        printf ("Enter a value to be inserted: ");
        scanf ("%d", &val);
        newnode->data = val;
        newnode->next = temp->next;
        temp->next = newnode;
    }
    }
    else
    printf ("Linked List is not yet");
}

void insertbefore ()
{
    if (start != NULL)
    {
        int val;
        struct node *temp = start, *pre, *newnode;
        printf ("Inserting Before an element\n");
        printf ("Enter the value before which you want to insert: ");
        scanf (" %d", &val);
        while (temp->next != NULL && temp->data != val)
        {
            pre = temp;
            temp = temp->next;
        }
        if (temp->data != val)
        printf ("value not found");
        else if (temp == start)
        {
            newnode = (struct node *) malloc (sizeof (struct node));
            printf ("Enter value to be inserted: ");
            scanf ("%d", &val);
            start = newnode;
            newnode->data = val;
            newnode->next = temp;
        }
        else
        {

```

```
newnode = (struct node *) malloc (sizeof (struct node));
printf ("Enter value to be inserted: ");
scanf ("%d", &val);
newnode->data = val;
newnode->next = temp;
pre->next = newnode;
}
}
else
    printf ("Linked List is not created ");
}

void delete ()
{
    if (start != NULL)
    {
        int val;
        struct node *temp = start, *pre, *newnode;
        printf ("Deleting an element\n");
        printf ("Enter the value to be deleted : ");
        scanf (" %d", &val);
        while (temp->next != NULL && temp->data != val)
        {
            pre = temp;
            temp = temp->next;
        }
        if (temp->data != val)
            printf ("value not found");
        else if (temp == start)
        {
            start = start->next;
            free (temp);
        }
        else
        {
            pre->next = temp->next;
            free (temp);
        }
    }
    else
        printf ("Linked List is not created");
}

int main ()
{
    int choice = 0;

    printf
    ("Choose: \n1) Create\n2) Display\n3) Search\n4) Insert Before\n5) Insert After\n6)
    Delete\n7) Exit\n");
```

```
while (1)
{
    printf ("Your choice is: ");
    scanf ("%d", &choice);
    switch (choice)
    {
    case 1:
        create ();
        break;
    case 2:
        display ();
        break;
    case 3:
        search ();
        break;
    case 4:
        insertbefore ();
        break;
    case 5:
        insertafter ();
        break;
    case 6:
        delete ();
        break;
    case 7:
        exit (0);
        break;
    default:
        printf ("NOT APPLICABLE");
    }
    if (choice == 6)
    break;
    printf ("\n");
}
return 0;
}
```

OUTPUT:

```
Choose:
1) Create
2) Display
3) Search
4) Insert Before
5) Insert After
6) Delete
7) Exit
Your choice is: 1
Creating linked list
Enter number of elements you want in the linked list.
5
Enter value: 6
Enter value: 7
Enter value: 8
Enter value: 9
Enter value: 4

Your choice is: 2
Display linked list
6, 7, 8, 9, 4, 0.
Your choice is: 3
Enter the value to be searched:
8
8 is the 3 number element in the linked list.
```

```
Your choice is: 4
Inserting Before an element
Enter the value before which you want to insert:
9
Enter value to be inserted: 2

Your choice is: 5
Inserting After an element
Enter the value after which you want to insert: 7
Enter a value to be inserted: 5

Your choice is: 6
Deleting an element
Enter the value to be deleted : 8

...Program finished with exit code 0
```

CONCLUSION:

Operations of single linked list have been performed successfully.

EXPERIMENT 4:

Aim:

Implementation of Stacks using Arrays.

THEORY:

Stack is a linear data structure.It allows all data operations at one end only. At any given time, we can only access the top element of a stack.This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing

Algorithm:

PUSH OPERATION:

```
STEP 1: IF TOP = MAX -1
PRINT " OVERFLOW"
GOTO STEP 4
[END IF]
STEP 2: SET TOP = TOP + 1
STEP 3: SET STACK[TOP] = VALUE
STEP 4: END
```

POP OPERATION:

```
STEP 1: IF TOP == NULL
PRINT "UNDERFLOW"
GOTO STEP 4
[END IF]
STEP 2: SET VAL = STACK[TOP]
STEP 3: SET TOP = TOP - 1
STEP 4: END
```

PEEK OPERATION:

```
STEP 1: IF TOP == NULL
PRINT "STACK IS EMPTY"
GOTO STEP 3
STEP 2: RETURN STACK[TOP]
STEP 3: END
```

CODE:

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void push(int);
void pop();
void display();
int stack[SIZE], top = -1;
void main()
{
    int value, choice;
    // clrscr();
    while(choice!=4){
        printf("Stack using array\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
```

```
scanf("%d",&choice);
switch(choice){
case 1: printf("Enter the value to be insert: ");
        scanf("%d",&value);
        push(value);
        break;
case 2: pop();
        break;
case 3: display();
        break;
case 4: printf("Invalid"); }}}
void push(int value){
    if(top == SIZE-1)
        printf("\nStack is Full");
    else{
        top++;
        stack[top] = value;
        printf("Inserted successfully\n"); }}
void pop(){
    if(top == -1)
        printf("\nStack is empty");
    else{
        printf("\nDeleted : %d\n", stack[top]);
        top--; }}
void display(){
    if(top == -1)
        printf("\nStack is empty");
    else{
        int i;
        printf("Stack elements are:\n");
        for(i=top; i>=0; i--)
            printf("%d\n",stack[i]); }}
```

OUTPUT:

```
Stack using array
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 5
Inserted successfully
Stack using array
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 99
Inserted successfully
Stack using array
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 36
Inserted successfully
Stack using array
1. Push
2. Pop
```

```
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 45
Inserted successfully
Stack using array
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Deleted : 45
Stack using array
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements are:
36
99
5
Stack using array
1. Push
2. Pop
3. Display
```

Conclusion: Stacks have been implemented using arrays.

EXPERIMENT 5:

Aim:

Implement Stack and Queue using Linked List.

Theory:

1. Stacks:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Algorithm to insert an element in a linked stack:-

Step 1: Allocate memory for the new node and name it as NEW_NODE

Step 2: SET NEW_NODE->DATA = VAL

Step 3: IF TOP = NULL

 SET NEW_NODE->NEXT=NULL

 SET TOP = NEW_NODE

ELSE

 SET NEW_NODE->NEXT=TOP

 SET TOP = NEW_NODE

[END OF IF]

Step 4: END

Algorithm to delete an element from a linked stack:-

Step 1: IF TOP = NULL

 PRINT UNDERFLOW

 GO TO Step 5

[END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP=TOP->NEXT

Step 4: FREE PTR

Step 5: END

CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<malloc.h>
struct stack
{
    int data;
    struct stack *next;
};
struct stack *top=NULL;
struct stack *push(struct stack *,int);
struct stack *pop(struct stack *);
struct stack *display(struct stack *);
int peek(struct stack *);
int main(int argc,char *argv[]){
    int val,choice;
    do{
```

```
printf("\n 1.Push");
printf("\n 2.Pop");
printf("\n 3.Peek");
printf("\n 4.Display");
printf("\n 5.Exit");
printf("\n Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
    case 1:
        printf("Enter the number to be pushed into the stack : ");
        scanf("%d",&val);
        top=push(top,val);
        break;
    case 2:
        top=pop(top);
        break;
    case 3:
        val=peek(top);
        if(val!=-1)
            printf("The value at the top of the stack is %d",val);
        else
            printf("\n Stack is empty");
        break;
    case 4:
        top=display(top);
        break;
}
}while(choice!=5);
return 0;
}
struct stack *push(struct stack *top,int val)
{
    struct stack *temp;
    temp=(struct stack *)malloc(sizeof(struct stack));
    temp->data=val;
    if(top==NULL)
    {
        temp->next=NULL;
        top=temp;
    }
    else
    {
        temp->next=top;
        top=temp;
    }
    return top;
}
struct stack *pop(struct stack *top)
{
    struct stack *temp;
    temp=top;
    if(top==NULL)
```



```
printf("\n Stack Underflow");
else
{
    top=top->next;
    printf("The deleted value is %d",temp->data);
    free(temp);
}
return top;
}
struct stack *display(struct stack *top)
{
    struct stack *temp;
    temp=top;
    if(top==NULL)
        printf("\n Stack is empty");
    else
    {
        while(temp!=NULL)
        {
            printf("\n %d",temp->data);
            temp=temp->next;
        }
    }
    return top;
}
int peek(struct stack *top)
{
    if(top==NULL)
        return -1;
    else
        return top->data;
}
```

OUTPUT:

```
Enter your choice : 1
Enter the number to be pushed into the stack : 3

1.Push
2.Pop
3.Peek
4.Display
5.Exit
Enter your choice : 4

3
9
5
1.Push
2.Pop
3.Peek
4.Display
5.Exit
Enter your choice : 2
The deleted value is 3
1.Push
2.Pop
3.Peek
4.Display
5.Exit
Enter your choice : 3
The value at the top of the stack is 9
```

```
Enter your choice : 1
Enter the number to be pushed into the stack : 3

1.Push
2.Pop
3.Peek
4.Display
5.Exit
Enter your choice : 4

3
9
5
1.Push
2.Pop
3.Peek
4.Display
5.Exit
Enter your choice : 2
The deleted value is 3
1.Push
2.Pop
3.Peek
4.Display
5.Exit
Enter your choice : 3
The value at the top of the stack is 9
```

Conclusion: Stacks have been implemented using linked list.

Queue:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

ALGORITHM:

Algorithm to insert an element in a linked queue:-

Step 1: Allocate memory for the new node and name it as PTR

Step 2: SET PTR->DATA=VAL

Step 3: IF FRONT = NULL

SET FRONT = REAR = PTR

ELSE

SET REAR->NEXT=PTR

SET REAR = PTR

SET REAR->NEXT=NULL

[END OF IF]

Step 4: END

Algorithm to delete an element from a linked queue:-

Step 1: IF FRONT = NULL

Write Underflow

Go to Step 5

[END OF IF]

Step 2: SET PTR = FRONT

Step 3: SET FRONT = FRONT->NEXT

Step 4: FREE PTR

Step 5: END

CODE:

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

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

```
#define N 20
```

```
typedef struct node
```

```
{
```

```
    int data;
```

```
    struct node *next;
```

```
}node;
```

```
typedef struct ll
```

```
{
```

```
    node *start;
```

```
}ll;
```

```
void create(ll *l,int n)
```

```
{
```

```
    node *newnode,*p;
```

```
    int val,i;
```

```
if(n==1){
    printf("Enter the value :");
    scanf("%d",&val);
    newnode = (node *)malloc(sizeof(node));
    newnode->data = val;
    newnode->next = NULL;
    l->start = newnode;
}
else{
    printf("Enter a value:");
    scanf("%d",&val);
    newnode = (node *)malloc(sizeof(node));
    newnode->data = val;
    newnode->next = NULL;
    l->start = newnode;
    for(i=1;i<=n-1;i++){
        printf("Enter a value:");
        scanf("%d",&val);
        newnode = (node*)malloc(sizeof(node));
        newnode->data = val;
        newnode->next = NULL;
        p=l->start;
        while(p->next!=NULL){
            p=p->next;
        }
        p->next = newnode;
    }
}
}
void enqueue(ll *l,int ele)
{
    node *newnode,*p;
    newnode=(node*)malloc(sizeof(node));
    newnode->data=ele;
    newnode->next=NULL;
    if(l->start==NULL)
    {
        l->start=newnode;
    }
    else
    {
        p=l->start;
        while(p->next!=NULL)
            p=p->next;

        p->next=newnode;
    }
}

void dequeue(ll *l)
{
    node *p;
    if(l->start==NULL)
        printf("\nQueue is Empty\n");
```

```
    else{
        p=l->start;
        l->start=p->next;
        free(p);
    }
}

void display(ll *l)
{
    node *p;
    p=l->start;
    while(p!=NULL)
    {
        printf("\t%d",p->data);
        p=p->next;
    }
}

int main()
{
    ll l;
    l.start=NULL;
    int ele,option,n;
    while(1)
    {
        printf("\n Menu");
        printf("\n 1. CREATE");
        printf("\n 2. ENQUEUE ");
        printf("\n 3. DEQUEUE ");
        printf("\n 4. DISPLAY ");
        printf("\n 5. Exit");
        printf("\n Enter your choice");
        scanf("%d",&option);
        if(option==5)
            break;
        switch(option)
        {
            case 1:
                printf("Enter the number of nodes: ");
                scanf("%d",&n);
                create(&l,n);
                break;
            case 2:
                printf("\nEnter the value to be inserted\n");
                scanf("%d",&ele);
                enqueue(&l,ele);
                break;
            case 3:
                dequeue(&l);
                break;
            case 4:
                display(&l);
                break;
            default:
                printf("\nInvalid Input\n"); }}}
}
```

OUTPUT:

```
Menu
1. CREATE
2. ENQUEUE
3. DEQUEUE
4. DISPLAY
5. Exit
Enter your choice1
Enter the number of nodes: 4
Enter a value:5
Enter a value:8
Enter a value:9
Enter a value:1

Menu
1. CREATE
2. ENQUEUE
3. DEQUEUE
4. DISPLAY
5. Exit
Enter your choice2

Enter the value to be inserted
3

Menu
1. CREATE
```

```
1. CREATE
2. ENQUEUE
3. DEQUEUE
4. DISPLAY
5. Exit
Enter your choice3
```

```
Menu
1. CREATE
2. ENQUEUE
3. DEQUEUE
4. DISPLAY
5. Exit
Enter your choice3
```

```
Menu
1. CREATE
2. ENQUEUE
3. DEQUEUE
4. DISPLAY
5. Exit
Enter your choice4
          9          1          3

Menu
1. CREATE
2. ENQUEUE
3. DEQUEUE
```

Conclusion: Queues have been implemented using linked list.

Experiment No. 6:

Aim:

Implement Infix to Postfix Transformation.

Theory:

To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Higher priority : *, /, %

Lower priority : +, -

Algorithm:

Step 1: Add ") " to the end of the infix expression

Step 2: Push " (" on to the stack

Step 3: Repeat until each character in the infix notation is scanned

IF a " (" is encountered, push it on the stack

IF an operand (whether a digit or a character) is encountered, add it postfix expression.

IF a ") " is encountered, then

a. Repeatedly pop from stack and add it to the postfix expression until a " (" is encountered.

b. Discard the " (" . That is, remove the " (" from stack and do not add it to the postfix expression

IF an operator is encountered, then

a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than 0

b. Push the operator to the stack

[END OF IF]

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: EXIT.

CODE:

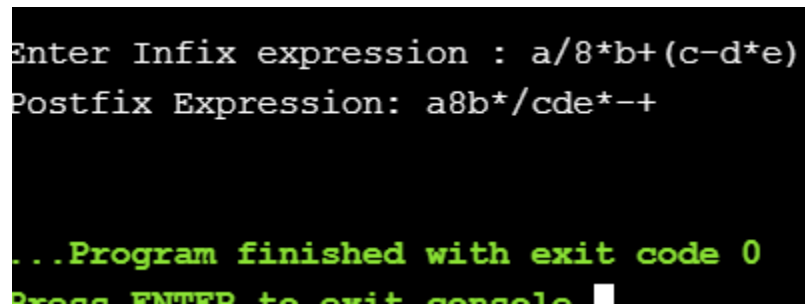
```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
#define SIZE 100
char stack[SIZE];
int top = -1;

void push(char val)
{
    if(top >= SIZE-1)
    {
        printf("\nStack Overflow.");
    }
}
```

```
    else {
        top = top+1;
        stack[top] = val; }}
char pop()
{ char val ;
  if(top == -1)
  {
      printf("stack under flow: invalid infix expression");
  }
  else
  {
      val = stack[top];
      top = top-1;
      return(val); }}
int precedence(char symbol)
{
    if(symbol == '%' || symbol == '*' || symbol == '/')
    {
        return 1;
    }
    else if ( symbol == '+' || symbol == '-')
        return 0;
}
void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
    int i=0, j=0;
    char item;
    strcpy(postfix_exp, " ");
    char x;
    while(infix_exp[i] != '\0')
    {
        if(infix_exp[i] == '(')
        {
            push(infix_exp[i]);
            i++;
        }
        else if( isdigit(infix_exp[i]) || isalpha(infix_exp[i]))
        {
            postfix_exp[j] = infix_exp[i];
            j++;
            i++;
        }
        else if(infix_exp[i] == '%' || infix_exp[i] == '*' || infix_exp[i] == '/' || infix_exp[i] == '+'
|| infix_exp[i] == '-')
        {
            while(top !=-1 && stack[top]!='(' &&
precedence(stack[top])>precedence(infix_exp[i]))
            {
                postfix_exp[j] = pop();
                j++;
            }
            push(infix_exp[i]);
            i++;
        }
    }
}
```

```
else if(infix_exp[i] == ')')
{
    while(top !=-1 && stack[top]!='(')
    {
        postfix_exp[j] = pop();
        j++;
    }
    if(top== -1)
    {
        printf("\nInvalid infix Expression.\n");
        // getchar();
        exit(1);
    }
    x = pop();
    i++;
}
else
{
    printf("\nInvalid infix Expression.\n");
    exit(1);
}
}
while(top !=-1 && stack[top]!='(')
{
    postfix_exp[j] = pop();
    j++;
}
postfix_exp[j] = '\0';
}
int main()
{
    char infix[100], postfix[100];
    printf("\nEnter Infix expression : ");
    gets(infix);
    InfixToPostfix(infix,postfix);
    printf("Postfix Expression: ");
    puts(postfix);
    return 0;
}
```

OUTPUT:



```
Enter Infix expression : a/8*b+(c-d*e)
Postfix Expression: a8b*/cde*-+

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: Infix expression has been converted to postfix expression.

Experiment No. 7:

Aim:

Implement Double Ended Queue.

Theory:

Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends, i.e., front and back.

A deque (pronounced as 'deck' or 'dequeue') is a list in which the elements can be inserted or deleted at either end. It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end. However, no element can be added and deleted from the middle. In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list. In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N-1] is followed by Dequeue[0].

There are two variants of a double-ended queue. They include :

- Input restricted deque: In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
- Output restricted deque: In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

CODE:

```
include<stdio.h>
#define MAX 5
int deque_arr[MAX];
int left = -1;
int right = -1;
void insert_right ()
{
    int x;
    if ((left == 0 && right == MAX - 1) || (left == right + 1))
    {
        printf ("Queue Overflow\n");
        return;
    }
    if (left == -1)
    {
        left = 0;
        right = 0;
    }
    else if (right == MAX - 1)
        right = 0;
    else
        right = right + 1;
    printf ("Input the element for adding in queue : ");
    scanf ("%d", &x);
    deque_arr[right] = x;
}

void insert_left ()
{
    int x;
```

```
if ((left == 0 && right == MAX - 1) || (left == right + 1))
{
    printf ("Queue Overflow \n");
    return;
}
if (left == -1)
{
    left = 0;
    right = 0;
}
else if (left == 0)
    left = MAX - 1;
else
    left = left - 1;
printf ("Input the element for adding in queue : ");
scanf ("%d", &x);
deque_arr[left] = x;
}

void delete_left ()
{
    if (left == -1)
    {
        printf ("Queue Underflow\n");
        return;
    }
    printf ("Element deleted from queue is : %d\n", deque_arr[left]);
    if (left == right)
    {
        left = -1;
        right = -1;
    }
    else if (left == MAX - 1)
        left = 0;
    else
        left = left + 1;
}

void delete_right ()
{
    if (left == -1)
    {
        printf ("Queue Underflow\n");
        return;
    }
    printf ("Element deleted from queue is : %d\n", deque_arr[right]);
    if (left == right)
    {
        left = -1;
        right = -1;
    }
    else if (right == 0)
        right = MAX - 1;
    else
```

```
    right = right - 1;
}

void display_queue ()
{
    int front_pos = left, rear_pos = right;
    if (left == -1)
    {
        printf ("Queue is empty\n");
        return;
    }
    printf ("Queue elements :\n");
    if (front_pos <= rear_pos)
    {
        while (front_pos <= rear_pos)
        {
            printf ("%d ", deque_arr[front_pos]);
            front_pos++;
        }
    }
    else
    {
        while (front_pos <= MAX - 1)
        {
            printf ("%d ", deque_arr[front_pos]);
            front_pos++;
        }
        front_pos = 0;
        while (front_pos <= rear_pos)
        {
            printf ("%d ", deque_arr[front_pos]);
            front_pos++;
        }
    }
    printf ("\n");
}

void input_q ()
{
    int choice;
    do
    {
        printf ("1.Insert at right\n");
        printf ("2.Delete from left\n");
        printf ("3.Delete from right\n");
        printf ("4.Display\n");
        printf ("5.Quit\n");
        printf ("Enter your choice : ");
        scanf ("%d", &choice);

        switch (choice)
        {
            case 1:
                insert_right ();
```

```
        break;
    case 2:
        delete_left ();
        break;
    case 3:
        delete_right ();
        break;
    case 4:
        display_queue ();
        break;
    case 5:
        break;
    default:
        printf ("Wrong choice\n");
    }
}
while (choice != 5);
}

void output_q ()
{
    int choice;
    do
    {
        printf ("1.Insert at right\n");
        printf ("2.Insert at left\n");
        printf ("3.Delete from left\n");
        printf ("4.Display\n");
        printf ("5.Quit\n");
        printf ("Enter your choice : ");
        scanf ("%d", &choice);
        switch (choice)
        {
            case 1:
                insert_right ();
                break;
            case 2:
                insert_left ();
                break;
            case 3:
                delete_left ();
                break;
            case 4:
                display_queue ();
                break;
            case 5:
                break;
            default:
                printf ("Wrong choice\n");
        }
    }
    while (choice != 5);
}

void main ()
```

```
{  
    int choice;  
    printf ("1.Input restricted dequeue\n");  
    printf ("2.Output restricted dequeue\n");  
    printf ("Enter your choice : ");  
    scanf ("%d", &choice);  
    switch (choice)  
    {  
        case 1:  
            input_q ();  
            break;  
        case 2:  
            output_q ();  
            break;  
        default:  
            printf ("Wrong choice\n");  
    }  
}
```

OUTPUT:

```
1.Input restricted dequeue  
2.Output restricted dequeue  
Enter your choice : 1  
1.Insert at right  
2.Delete from left  
3.Delete from right  
4.Display  
5.Quit  
Enter your choice : 1  
Input the element for adding in queue : 5  
1.Insert at right  
2.Delete from left  
3.Delete from right  
4.Display  
5.Quit  
Enter your choice : 1  
Input the element for adding in queue : 9  
1.Insert at right  
2.Delete from left  
3.Delete from right  
4.Display  
5.Quit  
Enter your choice : 1  
Input the element for adding in queue : 55  
1.Insert at right  
2.Delete from left  
3.Delete from right
```

```
5.Quit  
Enter your choice : 4  
Queue elements :  
5 9 55  
1.Insert at right  
2.Delete from left  
3.Delete from right  
4.Display  
5.Quit  
Enter your choice : 2  
Element deleted from queue is : 5  
1.Insert at right  
2.Delete from left  
3.Delete from right  
4.Display  
5.Quit  
Enter your choice : 3  
Element deleted from queue is : 55  
1.Insert at right  
2.Delete from left  
3.Delete from right  
4.Display  
5.Quit  
Enter your choice : 5  
...Program finished with exit code 5
```

```
1.Input restricted dequeue
2.Output restricted dequeue
Enter your choice : 2
1.Insert at right
2.Insert at left
3.Delete from left
4.Display
5.Quit
Enter your choice : 1
Input the element for adding in queue : 5
1.Insert at right
2.Insert at left
3.Delete from left
4.Display
5.Quit
Enter your choice : 1
Input the element for adding in queue : 99
1.Insert at right
2.Insert at left
3.Delete from left
4.Display
5.Quit
Enter your choice : 2
Input the element for adding in queue : 88
1.Insert at right
2.Insert at left
3.Delete from left
```

```
4.Display
5.Quit
Enter your choice : 2
Input the element for adding in queue : 11
1.Insert at right
2.Insert at left
3.Delete from left
4.Display
5.Quit
Enter your choice : 4
Queue elements :
11 88 5 99
1.Insert at right
2.Insert at left
3.Delete from left
4.Display
5.Quit
Enter your choice : 3
Element deleted from queue is : 11
1.Insert at right
2.Insert at left
3.Delete from left
4.Display
5.Quit
Enter your choice : 5
```

Conclusion: Double ended queue has been implemented successfully.

Experiment No. 8:

Aim:

Implement Binary Search Tree.

Theory:

Algorithm to search for a given value in a binary search tree:-

SearchElement (TREE, VAL)

Step 1: IF TREE DATA = VAL OR TREE = NULL

Return TREE

ELSE

IF VAL < TREE DATA

Return searchElement(TREE LEFT, VAL)

ELSE

Return searchElement(TREE RIGHT, VAL)

[END OF IF]

[END OF IF]

Step 2: END

ALGORITHM:

Algorithm to insert a given value in a binary search tree:-

Insert (TREE, VAL)

Step 1: IF TREE = NULL

Allocate memory for TREE

SET TREE ->DATA = VAL

SET TREE-> LEFT = TREE-> RIGHT = NULL

ELSE

IF VAL < TREE->DATA

Insert(TREE-> LEFT, VAL)

ELSE

Insert(TREE->RIGHT, VAL)

[END OF IF]

[END OF IF]

Step 2: END

Algorithm to delete a node from a binary search tree:-

Delete (TREE, VAL)

Step 1: IF TREE = NULL

Write "VAL not found in the tree"

ELSE IF VAL < TREE-> DATA

Delete(TREE->LEFT, VAL)

ELSE IF VAL > TREE-> DATA

Delete(TREE-> RIGHT, VAL)

ELSE IF TREE ->LEFT AND TREE->RIGHT

SET TEMP = findLargestNode(TREE-> LEFT)

SET TREE-> DATA = TEMP-> DATA

Delete(TREE ->LEFT, TEMP-> DATA)

ELSE

SET TEMP = TREE

IF TREE ->LEFT = NULL AND TREE ->RIGHT = NULL

SET TREE = NULL

ELSE IF TREE-> LEFT != NULL

```
        SET TREE = TREE ->LEFT
    ELSE SET TREE = TREE ->RIGHT
    [END OF IF]
    FREE TEMP
[END OF IF]
Step 2: END
```

CODE:

```
#include <stdio.h>
#include <malloc.h>
struct node {
    int data;
    struct node *left;
    struct node *right; };
struct node *tree;
void create_tree(struct node *tree)
{ tree = NULL;
}
struct node *insertElement(struct node *tree, int val)
{

    struct node *ptr, *nodeptr, *parentptr;
    ptr = (struct node *)malloc(sizeof(struct node));

    ptr->data = val;

    ptr->left = NULL;

    ptr->right = NULL;

    if (tree == NULL)
    { tree = ptr;

        tree->left = NULL;

        tree->right = NULL; }

    else {

        parentptr = NULL;

        nodeptr = tree;

        while (nodeptr != NULL)

        { parentptr = nodeptr;

            if (val < nodeptr->data)

                nodeptr = nodeptr->left;

            else nodeptr = nodeptr->right;
```



```
if (val < parentptr->data)
    parentptr->left = ptr;
else parentptr->right = ptr; }
return tree; }

void preorderTraversal(struct node *tree)
{
    if (tree != NULL)
    {
        printf("%d\t", tree->data);
        preorderTraversal(tree->left);
        preorderTraversal(tree->right); }}

void inorderTraversal(struct node *tree)
{
    if (tree != NULL)
    {
        inorderTraversal(tree->left);
        printf("%d\t", tree->data);
        inorderTraversal(tree->right);
    }
}

void postorderTraversal(struct node *tree)
{
    if (tree != NULL)
    {
        postorderTraversal(tree->left);
        postorderTraversal(tree->right);
        printf("%d\t", tree->data);
    }
}
```

```
struct node *deleteElement(struct node *tree, int val)
{
    struct node *cur, *parent, *suc, *psuc, *ptr;
    if (tree->left == NULL)
    {
        printf("\n The tree is empty ");
        return (tree);
    }
    parent = tree;
    cur = tree->left;
    while (cur != NULL && val != cur->data)
    {
        parent = cur;
        cur = (val < cur->data) ? cur->left : cur->right;
    }
    if (cur == NULL)
    {
        printf("\n The value to be deleted is not present in the tree");
        return (tree);
    }
    if (cur->left == NULL)
        ptr = cur->right;
    else if (cur->right == NULL)
        ptr = cur->left;
    else
    {
        psuc = cur;
```

```
    cur = cur->left;
    while (suc->left != NULL)
    {
        psuc = suc;
        suc = suc->left;
    }
    if (cur == psuc)
    {
        suc->left = cur->right;
    }
    else
    {
        suc->left = cur->left;
        psuc->left = suc->right;
        suc->right = cur->right;
    }
    ptr = suc;
}
if (parent->left == cur)
    parent->left = ptr;
else
    parent->right = ptr;
free(cur);
return tree;
}

struct node *deleteTree(struct node *tree)
{

```

```
if (tree != NULL)
{
    deleteTree(tree->left);
    deleteTree(tree->right);
    free(tree);
}
}
int main()
{
    int option, val;
    struct node *ptr;
    create_tree(tree);
    do
    {
        printf("\nMAIN MENU");
        printf("\n 1. Insert Element");
        printf("\n 2. Preorder Traversal");
        printf("\n 3. Inorder Traversal");
        printf("\n 4. Postorder Traversal");
        printf("\n 5. Delete an element");
        printf("\n 6. Delete the tree");
        printf("\n 7. Exit");
        printf("\nEnter your option : ");
        scanf("%d", &option);
        switch (option)
        {
            case 1:
                printf("\n Enter the value of the new node : ");
```

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

tree = insertElement(tree, val);

break;

case 2:

printf("\n The elements of the tree are : ");

preorderTraversal(tree);

break;

case 3:

printf("\n The elements of the tree are : ");

inorderTraversal(tree);

break;

case 4:

printf("\n The elements of the tree are : ");

postorderTraversal(tree);

break;

case 5:

printf("\n Enter the element to be deleted : ");

scanf("%d", &val);

tree = deleteElement(tree, val);

break;

case 6:

tree = deleteTree(tree);

break;

}

} while (option != 7);

return 0;

}
```

OUTPUT:

```
MAIN MENU
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Delete an element
6. Delete the tree
7. Exit
Enter your option : 1

Enter the value of the new node : 8
```

```
MAIN MENU
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Delete an element
6. Delete the tree
7. Exit
Enter your option : 1

Enter the value of the new node : 6
```

```
MAIN MENU
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Delete an element
6. Delete the tree
7. Exit
Enter your option : 1

Enter the value of the new node : 7
```

```
MAIN MENU
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Delete an element
6. Delete the tree
7. Exit
Enter your option : 2

The elements of the tree are : 8      6      7

MAIN MENU
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
```

```
The elements of the tree are : 6      7      8

MAIN MENU
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Delete an element
6. Delete the tree
7. Exit
Enter your option : 4

The elements of the tree are : 7      6      8

MAIN MENU
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Delete an element
6. Delete the tree
7. Exit
Enter your option : 5

Enter the element to be deleted : 6
```

Conclusion: Binary tree has been implemented.

Experiment No. 9:

Aim:

Implement Hashing Techniques.

Theory:

Hash Function:

Division Method

It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

$$h(x) = x \text{ mod } M$$

The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M.

The division method is extremely simple to implement. The following code segment illustrates how to do this:

```
int const M = 97; // a prime number
int h (int x)
{ return (x % M); }
```

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

Open Addressing Technique:

Linear Probing

The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision: $h(k, i) = [h(k) + i] \text{ mod } m$ www.EngineeringBooksPdf.com 470 Data Structures Using C Where m is the size of the hash table, $h(k) = (k \text{ mod } m)$, and i is the probe number that varies from 0 to m-1. Therefore, for a given key k, first the location generated by $[h(k) \text{ mod } m]$ is probed because for the first time $i=0$. If the location is free, the value is stored in it, else the second probe generates the address of the location given by $[h(k) + 1] \text{ mod } m$. Similarly, if the location is occupied, then subsequent probes generate the address as $[h(k) + 2] \text{ mod } m$, $[h(k) + 3] \text{ mod } m$, $[h(k) + 4] \text{ mod } m$, $[h(k) + 5] \text{ mod } m$, and so on, until a free location is found.

Code:

```
#include<stdio.h>

/*Code for linear probing*/

void insert(int ary[],int hashFn, int size){
    int element,pos,n=0;

    printf("Enter key element to insert\n");
```

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

pos = element%hashFn;

while(ary[pos]!=NULL)
{ // NULL indicates that cell is empty. So if cell is empty loop will break and goto bottom
of the loop to insert element
    if(ary[pos]==ary[size+1])
        break;
    pos = (pos+1)%hashFn;
    n++;
    if(n==size)
        break;    // If table is full we should break, if not check this, loop will go to infinite
loop.
}

if(n==size)
    printf("Hash table was full of elements\nNo Place to insert this element\n\n");
else
    ary[pos] = element;    //Inserting element
}

void delete(int ary[],int hashFn,int size){
/*
very careful observation required while deleting. To understand code of this delete function
see the note at end of the program
*/
int element,n=0,pos;

printf("Enter element to delete\n");
scanf("%d",&element);

pos = element%hashFn;

while(n++ != size){
    if(ary[pos]==NULL){
        printf("Element not found in hash table\n");
        break;
    }
    else if(ary[pos]==element){
        ary[pos]=NULL;
        printf("Element deleted\n\n");
        break;
    }
    else{
        pos = (pos+1) % hashFn;
    }
}
if(--n==size)
    printf("Element not found in hash table\n");
}
```



```
void search(int ary[],int hashFn,int size){
int element,pos,n=0;
printf("Enter element you want to search\n");
scanf("%d",&element);
pos = element%hashFn;

while(n++ != size)
{
    if(ary[pos]==element)
    {
        printf("Element found at index %d\n",pos);
        break;
    }
    else
    {
        if(ary[pos]==ary[size+1] ||ary[pos]!=NULL)
            pos = (pos+1)%hashFn;
    }
}
if(--n==size)
    printf("Element not found in hash table\n");
}

void display(int ary[],int size)
{
    int i;
    printf("Index\tValue\n");
    for(i=0;i<size;i++)
        printf("%d\t%d\n",i,ary[i]);
}

int main()
{
    int size,hashFn,i,choice;
    printf("Enter size of hash table\n");
    scanf("%d",&size);
    int ary[size];
    printf("Enter hash function [if mod 10 enter 10]\n");
    scanf("%d",&hashFn);
    for(i=0;i<size;i++)
        ary[i]=NULL; //Assigning NULL indicates that cell is empty
    do{
        printf("Enter your choice\n");
        printf(" 1-> Insert\n 2-> Delete\n 3-> Display\n 4-> Searching\n 0-> Exit\n");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                insert(ary,hashFn,size);
                break;
            case 2:
                delete(ary,hashFn,size);
                break;
```

```
        case 3:
            display(ary,size);
            break;
        case 4:
            search(ary,hashFn,size);
            break;
        default:
            printf("Enter correct choice\n");
            break;
    }
}while(choice);

return 0;
}
```

OUTPUT:

```
Enter size of hash table
11
Enter hash function [if mod 10 enter 10]
11
Enter your choice
1-> Insert
2-> Delete
3-> Display
4-> Searching
0-> Exit
1
Enter key element to insert
36
Enter your choice
1-> Insert
2-> Delete
3-> Display
4-> Searching
0-> Exit
1
Enter key element to insert
80
Enter your choice
1-> Insert
2-> Delete
```

```
3-> Display
4-> Searching
0-> Exit
3
Index  Value
0      0
1      0
2      0
3      36
4      80
5      0
6      0
7      0
8      0
9      0
10     0
Enter your choice
1-> Insert
2-> Delete
3-> Display
4-> Searching
0-> Exit
2
Enter element to delete
36
```

```
Element deleted
Enter your choice
1-> Insert
2-> Delete
3-> Display
4-> Searching
0-> Exit
3
Index  Value
0      0
1      0
2      0
3      0
4      80
5      0
6      0
7      0
8      0
9      0
10     0
Enter your choice
```

Conclusion: Hence Hashing technique is implemented.

Experiment No. 10:

Aim:

Implement BFS & DFS [Menu Driven].

Theory:

1. BFS:

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Algorithm for breadth-first search:-

Step 1: SET STATUS=1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS=2
(waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS=3
(processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2
(waiting state)

[END OF LOOP]

Step 6: EXIT

2. DFS:

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Algorithm for depth-first search:-

Step 1: SET STATUS=1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS=2
(waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS=3
(processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS=1) and set their STATUS=2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Code:

```
#include <stdio.h>
#define MAX 10
void breadth_first_search(int adj[][MAX],int visited[],int start)
{
    int queue[MAX],rear = -1,front = -1, i;
    queue[++rear] = start;
```

```
visited[start] = 1;
while(rear != front)
{
    start = queue[++front];
    printf("%c \t",start + 65);
    for(i = 0; i < MAX; i++)
    {
        if(adj[start][i] == 1 && visited[i] == 0)
        {
            queue[++rear] = i;
            visited[i] = 1;
        }
    }
}
}

void depth_first_search(int adj[][MAX],int visited[],int start)
{
    int stack[MAX];
    int top = -1, i;
    printf("%c",start + 65);
    visited[start] = 1;
    stack[++top] = start;
    while(top != -1)
    {
        start = stack[top];
        for(i = 0; i < MAX; i++)
        {
            if(adj[start][i] && visited[i] == 0)
            {
                printf("->");
                stack[++top] = i;
                printf("%c", i + 65);
                visited[i] = 1;
                break;
            }
        }
        if(i == MAX)
            top--;
    }
}

int main()
{
    int visited[MAX] = {0};
    int adj[MAX][MAX], i, j, choice;
    printf("1. BFS\n");
    printf("2. DFS\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
            printf("Enter the adjacency matrix:\n");
```

```
for(i = 0; i < MAX; i++)
{
    for(j = 0; j < MAX; j++)
        scanf("%d", &adj[i][j]);
}
printf("BFS Traversal: ");
breadth_first_search(adj,visited,0);
break;
case 2:
printf("Enter the adjacency matrix:\n");
for(i = 0; i < MAX; i++)
{
    for(j = 0; j < MAX; j++)
        scanf("%d", &adj[i][j]);
}
printf("DFS Traversal: ");
depth_first_search(adj,visited,0);
break;
default:
printf("Invalid Input!!");
}
}
```

Output:

```
1. BFS
2. DFS
Enter your choice: 1
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B C D E
BFS Traversal: A      B      D      F      H      I      E      G
                C
```

```
1. BFS
2. DFS
Enter your choice: 2
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B C D E
DFS Traversal: A->B->E->C->D->F->G->H->I->J
```

Conclusion: BFS AND DFS have been implemented.