EXPERIMENT - 2

Aim - To search an array for the given element using linear (or sequential) and binary search

Pre-Lab Work: Write the linear and binary search algorithms .

Linear Search : *Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.*

Algorithm:

Start with the first array element (index 0)
while(more elements in array)
{
    if value found at current index, return index;
    Try next element (increment index);
}
Value not found, return -1;

Binary search: Binary search is based on the "divide-and-conquer" strategy which works as follows: Start by looking at the middle element of the array
1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.
2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.
Repeat this process until the element is found, or until the entire array has been eliminated

Algorithm:

Set first and last boundary of array to be searched
Repeat the following:
    Find middle element between first and last boundaries;
    if (middle element contains the search value)
  return middle_element position;
    else if (first >= last)
  return –1;
    else if (value < the value of middle_element)
        set last to middle_element position – 1;
    else
  set first to middle_element position + 1;

***Programming Assignment**: Write a menu driven program in C which searches an array for the input element.*

Program:

```
#include <stdio.h>
int main()
{
int a[100] , i,j,s,c,f,l,m,n,swap;
printf("Enter the no of element of array \n");
scanf("%d",&n);
printf("Please enter the ARRAY \n");
for(i=0;i<n;i++)
{
printf("enter the %d element",i+1);
 scanf("%d",&a[i]);
}
for(i=0;i<n;i++)
{
   for(j=i+1;j<n;j++)
   {       if(a[i]>a[j])
      {
      swap=a[i];
a[i]=a[j];
a[j]=swap;
      }
   }
}
printf("Sorted array");
for(i=0i<n;i++)
{
   printf("%d ",a[i]);
}
printf("Please enter the no you want to search ");

scanf("%d",&s);
printf("which type of search you want to perform \n");
printf("enter 1 for liner search \n");
printf("enter 2 for binary search \n");
scanf("%d", &c);
switch(c)
{
   case 1 : for(i=0;i<n;i++)
```

```
    {
       if(a[i]==s)
       {
          printf(" your no present at no %d ", i+1);
          break;
       }
       if(i==n)
       {   printf(" sorry your no is not present ");
       }
    }
    break;
    case 2 : f=0;
    l=n-1;
    m=(f+l)/2;
    while(f<=l)
    {
       if(a[m]==s)
       {
       printf(" Your no is present at %d ", m+1);
       break;
       }
       if (a[m]<s)
      f=m+1;
       if(a[m]>s)
          l=m-1;
       if (f>l)
          printf("no is not present");
       m=(f+l)/2;
    }
    break;
    default :
    printf("sorry wrong choice");
    break;
}
return 0;
}
```

Output

1) Binary search

```
Enter the no of element of array
5
Please enter the ARRAY
enter the 1 element99
enter the 2 element87
enter the 3 element105
enter the 4 element12
enter the 5 element81
Sorted array12 81 87 99 105 Please enter the no you want to search 87
which type of search you want to perform
enter 1 for liner search
enter 2 for binary search
2
 Your no is present at 3
Process returned 0 (0x0)   execution time : 30.896 s
```

2) Linear search

```
Enter the no of element of array
5
Please enter the ARRAY
enter the 1 element44
enter the 2 element25
enter the 3 element97
enter the 4 element12
enter the 5 element11
Sorted array11 12 25 44 97 Please enter the no you want to search 44
which type of search you want to perform
enter 1 for liner search
enter 2 for binary search
1
 your no present at no 4
Process returned 0 (0x0)   execution time : 21.483 s
```

Experiment 1

Aim: To use functions for array handling

Pre-Lab Work:

Q1) How are single dimensional arrays used in C programs?

Ans - A One-Dimensional Array is the simplest form of an Array in which the elements are stored linearly and can be accessed individually by specifying the index value of each element stored in the array.
A One-Dimensional Array is a group of elements having the same data type which are stored in a linear arrangement under a single variable name.

Declaration Syntax: data_type array_name [array_size] ;

Initialization Syntax: data_type array_name [array_size] = {comma_separated_element_list};

Q2) How is an array passed as an argument to a function in C programs?

Ans.- If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1
Formal parameters as a pointer −

```
void myFunction(int *param) {
  .
  .
  .
}
```

Way-2
Formal parameters as a sized array −

```
void myFunction(int param[10]) {
  .
  .
  .
}
```

Way-3
Formal parameters as an unsized array −

```
void myFunction(int param[]) {
  .
  .
  .
}
```

Programming Assignment: Write a program using functions to find

- **the total number of even elements in a given array**
- **the maximum element in a given array**

Program

```
#include <stdio.h>
 int even(int size , int*A);
int max(int size , int*A);
void main()
{
 int arr[50],n,i;
 printf("Enter the no of element in array \n");
 scanf("%d",&n);
 printf("enter the element of array \n");
 for(i=0;i<n;i++)
 {
    printf("enter the %d element",i+1);
    scanf("%d",&arr[i]);
 }
  printf("total even the element in array is %d \n",even(n,arr));
  printf("maximum element of array is %d \n",max(n,arr));
}

int even(int size ,int*A)
{
   int j,c;
   for(j=0;j<size;j++)
   {
      if(A[j]%2==0)
         c++;
   }
   return c;
}
int max(int size, int*A)
{
 int m=0 ,j;
 for(j=0;j<size;j++)
 {
    if(m<A[j])
    {
       m=A[j];
```

```
    }
 }
 return m;
}
```

OUTPUT

```
Enter the no of element in array
5
enter the element of array
enter the 1 element2
enter the 2 element4
enter the 3 element9
enter the 4 element7
enter the 5 element8
total even the element in array is 44
maximum element of array is 9

Process returned 31 (0x1F)   execution time : 17.551 s
```

Experiment 3

Aim: To sort an array using iterative sorting algorithms.

Pre-Lab Work: Write selection, bubble and insertion sorting algorithms

Bubble sort - Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array (in case of arranging elements in ascending order)

In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at lower index is greater than the element at the higher index, the two elements are interchanged so that the smaller element is placed before the bigger one. This process is continued till the list of unsorted elements exhaust**.**

Algorithm:

```
BubbleSort(A, n)
{ i = 1
while(i < n)
        {
                for (j = 1 to n-i)
                        if (A[j] > A[j+1])
                                Swap A[j] & A[j+1]
        i = i+1
}
}
```

Selection Sort - Consider an array ARR with N elements. The selection sort takes N-1 passes to sort the entire array and works as follows. First find the smallest value in the array and place it in the first position. Then find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
- In pass 2, find the position POS of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now, A[0] and A[1] is sorted
- In pass 3, find the position POS of the smallest value in sub-array of N-2 elements. Swap ARR[POS] with ARR[2]. Now, ARR[0], ARR[1] and ARR[2] is sorted
- In pass N-1, find the position POS of the smaller of the elements ARR[N-2] and ARR[N-1}. Swap ARR[POS] and ARR[N-2] so that ARR[0], ARR[1], … , ARR[N-1] is sorted.

Algorithm:

```
SelectSort(A, n)
{
for (i = 1 to n )
        {          min = A[i]
                   pos=i;
                   for (j = i+1 to n)
                            if (A[j] < min)
                                     {min=A[j]
                                     pos=j
                                     }
                   A[pos]=A[i]
                   A[i]=min
        }
}
```

Insertion Sort - Insertion sort is a very simple sorting algorithm, in which the sorted array (or list) is built one element at a time.

- Insertion sort works as follows.
- The array of values to be sorted is divided into two sets. One that stores sorted values and the other contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially the element with index 0 (assuming LB, Lower Bound = 0) is in the sorted set, rest all the elements are in the unsorted set
- The first element of the unsorted partition has array index 1 (if LB = 0)
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Algorithm:

```
InsertionSort(A, n) {
for ( i = 2 to n) {
        key = A[i]
        j = i - 1
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
}
}
```

Programming Assignment: Write a menu driven program in C which sorts an array using selection, bubble and insertion sorting algorithms.

Program:

```
#include<stdio.h>
void main()
{
   int a[10],i,j,c,n,x,k,m;
   printf("enter the no element in array \n");
   scanf("%d",&n);
   printf("ENTER THE ELEMENT OF ARRAY\n");
   for(i=0;i<n;i++)
   {
      printf("Enter the element at %d ",i+1);
    scanf("%d",&a[i]);
   }
   printf("which type of shorting you want to do\n");
   printf("Enter 1 for bubble \n 2 for insertion sort \n 3 for selection short \n");
   scanf("%d",&c);
   switch(c)
   {
      case 1:  for (i=0;i<n;i++)
      {
      for (j=i+1;j<n;j++)
      {
         if (a[i]>a[j])
         {
            x=a[i];
            a[i]=a[j];
            a[j]=x;
         }
      }
      }
      for (i = 0; i < n; i++)
      printf("%d ", a[i]);
      break;
      case 2:  for (i = 1; i < n; i++)
      {
      k = a[i];
      j = i - 1;
      while (j >= 0 && a[j] > k)
```

```
      {
         a[j + 1] = a[j];
         j = j - 1;
      }
      a[j + 1] = k;
   }
   for (i = 0; i < n; i++)
      printf("%d ", a[i]);
   break;
   case 3 : for(i=0;i<n-1;i++)
   {
      m=i;
      for(j=i+1;j<n;j++)
      {
         if(a[j]<a[m])
         m=j;
      }
      if(m!=i)
      {
            x=a[i];
            a[i]=a[m];
            a[m]=x;
      }
   }
   for (i = 0; i < n; i++)
      printf("%d ", a[i]);
      break ;
    default: printf("wrong choice" );
    break;
   }
}
```

Output:

1. **Bubble sort**

```
enter the no element in array
5
ENTER THE ELEMENT OF ARRAY
Enter the element at 1 7
Enter the element at 2 5
Enter the element at 3 2
Enter the element at 4 1
Enter the element at 5 4
which type of shorting you want to do
Enter 1 for bubble
 2 for insertion sort
 3 for selection short
1
1 2 4 5 7
```

2. **Insertion Sort**

```
enter the no element in array
5
ENTER THE ELEMENT OF ARRAY
Enter the element at 1 5
Enter the element at 2 1
Enter the element at 3 2
Enter the element at 4 9
Enter the element at 5 7
which type of shorting you want to do
Enter 1 for bubble
 2 for insertion sort
 3 for selection short
2
1 2 5 7 9
```

3. **Selection Sort**

```
enter the no element in array  5
ENTER THE ELEMENT OF ARRAY
Enter the element at 1 8
Enter the element at 2 1
Enter the element at 3 6
Enter the element at 4 4
Enter the element at 5 3
which type of shorting you want to do
Enter 1 for bubble
 2 for insertion sort
 3 for selection short
3
1 3 4 6 8
```

Experiment 4

Aim: To sort an array using recursive sorting algorithms

Pre-Lab Work: Write the Merge Sort and Quick Sort algorithms.

Merge Sort . Merge sort is a sorting algorithm that uses the divide, conquer and combine algorithmic paradigm. Where,
Divide means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements in each sub-array. (If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A).
Conquer means sorting the two sub-arrays recursively using merge sort.
Combine means merging the two sorted sub-arrays of size n/2 each to produce the sorted array of n elements.

Algorithm:

```
Merge (A, left, mid, right)    { i = left
 j =mid+1
k=1
while (i<=mid AND j<=right)
{ if( A[i] < =A[j])
                    { temp [k] = A[i]
                      i=i+1
                      k=k+1}
else { temp [k] = A[j]
                      j=j+1
                      k=k+1}
}
while (i <= mid)
          { temp[k] = A[i]
          k=k+1
          i=i+1
          }
while (j<= right)
          { temp[k] = A[j]
          k=k+1
          j=j+1
          }
 }
```

Quick Sort : Another divide-and-conquer algorithm

The array A[p..r] is *partitioned* into two possibly empty subarrays A[p..q-1] and A[q+1..r]

All elements in A[p..q] are less than all elements in A[q+1..r]

The subarrays are recursively sorted by calls to quicksort

Unlike merge sort, no combining step: two subarrays form an already-sorted array.

Algorithm:

```
Quicksort(A, p, r)
{
   if (p < r)
   {
      q = Partition(A, p, r);
      Quicksort(A, p, q-1);
      Quicksort(A, q+1, r);
   }
}
Initial call is quicksort(A, 1, n)
PARTITION(A, p, r )
{
        x = A[r ]
        i = p − 1
        for j = p to r − 1
                { if A[ j ] ≤ x
                        {i ← i + 1
                        exchange A[i ] ↔ A[ j ]
                        }
                }
        exchange A[i + 1] ↔ A[r ]
        return i + 1
}
```

Programming Assignment: Write a menu driven program in C which sorts an array using Merge Sort and Quick Sort Algorithms.

Program:

```
include <stdio.h>

void printArray(int *A, int n)
{
   for (int i = 0; i < n; i++)
   {
      printf("%d ", A[i]);
   }
   printf("\n");
}

void merge(int A[], int mid, int low, int high)
{
   int i, j, k, B[100];
   i = low;
   j = mid + 1;
   k = low;

   while (i <= mid && j <= high)
   {
      if (A[i] < A[j])
      {
         B[k] = A[i];
         i++;
         k++;
      }
      else
      {
         B[k] = A[j];
         j++;
         k++;
      }
   }
   while (i <= mid)
   {
      B[k] = A[i];
      k++;
      i++;
   }
   while (j <= high)
   {
      B[k] = A[j];
```

```
        k++;
        j++;
    }
    for (int i = low; i <= high; i++)
    {
        A[i] = B[i];
    }

}

void mergeSort(int A[], int low, int high)
{
    int mid;
    if(low<high)
    {
        mid = (low + high) /2;
        mergeSort(A, low, mid);
        mergeSort(A, mid+1, high);
        merge(A, mid, low, high);
    }
}
int partition(int A[], int low, int high)
{
    int pivot = A[low];
    int i = low + 1;
    int j = high;
    int temp;

    do
    {
        while (A[i] <= pivot)
        {
            i++;
        }

        while (A[j] > pivot)
        {
            j--;
        }

        if (i < j)
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    } while (i < j);
```

```
    temp = A[low];
    A[low] = A[j];
    A[j] = temp;
    return j;
}

void quickSort(int A[], int low, int high)
{
    int partitionIndex;

    if (low < high)
    {
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1);
        quickSort(A, partitionIndex + 1, high);
    }
}

void main()
{

    int A[100],n,i,c;
    printf("enter the no of element ");
    scanf("%d",&n);
    printf("Enter the element of array");
    for(i=0;i<n;i++)
    {
        printf("Enter the element at %d",i+1);
        scanf("%d",&A[i]);
    }
    printArray(A, n);
    printf("whitch sort you want to perform \n press 1 for merg sort \n press 2 for quick sort \n");
    scanf("%d",&c);
    switch(c)
    {
        case 1 :  mergeSort(A, 0, n-1);
                  printArray(A, n);
                  break ;

        case 2 :  quickSort(A, 0, n - 1);
                   printArray(A, n);
                   break;
        default : printf("Wrong choice ");
                   break;
    }
}
```

O

Output

1. **Merge sort**

```
enter the no of element 10
Enter the element of array
 Enter the element at 1 25
Enter the element at 2 42
Enter the element at 3 1
Enter the element at 4 6
Enter the element at 5 88
Enter the element at 6 23
Enter the element at 7 99
Enter the element at 8 54
Enter the element at 9 2
Enter the element at 10 9
25 42 1 6 88 23 99 54 2 9
whitch sort you want to perform
 press 1 for merg sort
 press 2 for quick sort
1
1 2 6 9 23 25 42 54 88 99
```

2. **Quick sort**

```
enter the no of element 5
Enter the element of array
 Enter the element at 1 2
Enter the element at 2 7
Enter the element at 3 2
Enter the element at 4 5
Enter the element at 5 3
2 7 2 5 3
whitch sort you want to perform
 press 1 for merg sort
 press 2 for quick sort
2
2 2 3 5 7
```

Experiment 5

Aim: To implement singly linked lists

Pre-Lab Work: Write the algorithms for Insert before a given item in the linked list , Delete an item from the linked list and Display the linked lit functions .

Insert before a given item in the linked list ->

1. Create a new node with the given value

2. If the linked list is empty, set the new node as the head of the list and return

3. If the value to insert before is the head of the list, set the next pointer of the new node to point to the current head of the list and set the new node as the head of the list

4. Otherwise, traverse the list until a node with the value to insert before is found or the end of the list is reached

5. If a node with the value to insert before is found, set the next pointer of the new node to point to this node and set the next pointer of the previous node to point to the new node

6. If the end of the list is reached without finding a node with the value to insert before, insert the new node at the end of the list.

Delete an item from the linked list ->

1. If the linked list is empty, return

2. If the value to delete is the head of the list, set the head of the list to be the next node and return

3. Traverse the list until a node with the value to delete is found or the end of the list is reached

4. If a node with the value to delete is found, set the next pointer of the previous node to point to the next node and delete the current node

5. If the end of the list is reached without finding a node with the value to delete, return

Display the linked list ->

1. Starting from the head of the linked list, traverse the list and print the value of each node

2. Continue traversing until the end of the list is reached

Programming Assignment: Write a program to implement singly linked lists using the following functions

1. Create a linked list

2. Insert before a given item in the linked list

3. Delete an item from the linked list

4. Display the linked lit

Program

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
   int data;
   struct Node*next;
};
void linklist(struct Node *ptr)
{
   while(ptr!= NULL)
   {
      printf("Element %d\n",ptr->data);
      ptr = ptr->next;
   }
}

struct Node* insertaftervalue(struct Node * head, int d,int data)
{
   struct Node *ptr =(struct Node *)malloc(sizeof(struct Node));
   ptr->data=data;
   struct Node*p=head;
   while(p->next!=NULL&&p->data!=d)
   {
      p=p->next;
   }
   if(p->data==d)
   {
      ptr->next=p->next;
      p->next=ptr;
   }
   return(head);
};
struct Node* deletevalue(struct Node * head, int d)
{
   struct Node*p=head;
```

```
   struct Node*q=head->next;
   while(q->next!=NULL&&q->data!=d)
   {
      p=p->next;
      q=q->next;
   }
   if(q->data==d)
   {
      p->next=q->next;

      free(q);
   }
   return (head);

};
int main()
{
   struct Node*head;
   struct Node*second;
   struct Node*third;
   struct Node*fourth;

   head = (struct Node*)malloc(sizeof(struct Node));
   second = (struct Node*)malloc(sizeof(struct Node));
   third = (struct Node*)malloc(sizeof(struct Node));
   fourth= (struct Node*)malloc(sizeof(struct Node));
   int n;
   printf("Enter the first no ");
   scanf("%d",&n);
   head->data=n;
   head->next=second;
   printf("Enter the second no ");
   scanf("%d",&n);
   second->data=n;
   second->next=third;
   printf("Enter the third no ");
  scanf("%d",&n);
   third->data=n;
   third->next=fourth;
   printf("Enter the fourth no ");
   scanf("%d",&n);
   fourth->data=n;
   fourth->next=NULL;
   linklist(head);
   int data,d;
```

```
   int data2;
   printf("Enter the data after that you want to insert \n");
      scanf("%d",&d);
   printf("Enter the data you want to insert \n");
      scanf("%d",&data2);
     head= insertaftervalue(head,d,data2);
      linklist(head) ;
   printf("Enter the data you want to delete \n");
      scanf("%d",&data);
      head = deletevalue(head,data);
      linklist(head);
   return 0;
}
```

Output

```
Enter the first no 5
Enter the second no 6
Enter the third no 8
Enter the fourth no 9
Element 5
Element 6
Element 8
Element 9
Enter the data after that you want to insert
6
Enter the data you want to insert
7
Element 5
Element 6
Element 7
Element 8
Element 9
Enter the data you want to delete
6
Element 5
Element 7
Element 8
Element 9
```

Experiment 6

Aim: To implement stack operations using linked lists

Pre-Lab Work: How is a stack implemented using linked lists?

A stack is a data structure that follows the Last-In-First-Out (LIFO) principle, where the last element added to the stack is the first one to be removed. It can be implemented using linked lists, which is a data structure made up of a chain of nodes, where each node stores a value and a pointer to the next node in the list.

To implement a stack using linked lists, we can define a new data type called "stackNode," which contains two fields:

- **"value" to store the actual value of the node**
- **"next" to point to the next node in the stack**

We also need to define a new data type called "stack," which has a single field, the "top" field, that points to the topmost node in the stack.

Here's how the basic operations of a stack can be implemented using linked lists:

1. **Push: To add an element to the top of the stack, we create a new stackNode with the given value and set its "next" field to the current top of the stack. Then we update the "top" field of the stack to point to the new node. This operation takes O(1) time.**

2. **Pop: To remove the top element from the stack, we simply update the "top" field of the stack to point to the next node in the stack. We also need to return the value of the node that was removed. If the stack is empty, we return an error or throw an exception. This operation takes O(1) time.**

3. **Peek: To return the value of the top element without removing it, we simply return the "value" field of the node that is pointed to by the "top" field of the stack. This operation takes O(1) time.**

Programming Assignment: Write a program to implement a stack of integers using linked lists. The program should contain following functions

1. IsEmpty() – to check if the stack is empty

2. Push an item on stack

3. Pop an item from stack

4. Display the items of the stack from top to bottom of the stack.

Program

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

struct Node{
    int data;
    struct Node * next;
};
void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
int isEmpty(struct Node* top){
    if (top==NULL){
        return 1;
    }
    else{
        return 0;
    }
}
int isFull(struct Node* top){
    struct Node* p = (struct Node*)malloc(sizeof(struct Node));
    if(p==NULL){
        return 1;
    }
 else{
        return 0;
    }
}
```

```
struct Node* push(struct Node* top, int x){
    if(isFull(top)){
        printf("Stack Overflow\n");
    }
    else{
        struct Node* n = (struct Node*) malloc(sizeof(struct Node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
}
int pop(struct Node** top){
    if(isEmpty(*top)){
        printf("Stack Underflow\n");
    }
    else{
        struct Node* n = *top;
        *top = (*top)->next;
        int x = n->data;
        free(n);
        return x;
    }
}
int main(){
    struct Node* top = NULL;
    int e,n;
    printf("enter the no of element you want to insert ");
    scanf("%d",&n);
     for (int i = 0; i < n; i++)
     {
        /* code */
        printf("Enter the %d element you want to insert ",i+1);
        scanf("%d",&e);
        top = push(top, e);
     }

    linkedListTraversal(top);

      int element = pop(&top);
    printf("Popped element is %d\n", element);
    linkedListTraversal(top);
    return 0;
}
```

Output

enter the no of element you want to insert 5

Enter the 1 element you want to insert 25

Enter the 2 element you want to insert 26

Enter the 3 element you want to insert 27

Enter the 4 element you want to insert 28

Enter the 5 element you want to insert 29

Element: 29

Element: 28

Element: 27

Element: 26

Element: 25

Popped element is 29

Element: 28

Element: 27

Element: 26

Element: 25

Experiment 7

Aim: To implement queues using arrays.

Pre-Lab Work: How are queues implemented using arrays? What are the conditions for overflow and underflow?

Queues can be implemented using arrays as a circular buffer. In this implementation, the front and rear of the queue are tracked using indices in the array, and when an element is enqueued, it is placed at the rear of the queue, and when an element is dequeued, it is removed from the front of the queue.

Algorithm

1. Initialize an array of size n to represent the queue

2. Initialize front and rear indices to 0

3. To enqueue an element:

   a. Check if the rear index is equal to n-1. If it is, the queue is full and we cannot enqueue any more elements.

   b. Otherwise, increment the rear index and set the value at that index to the new element.

4. To dequeue an element:

   a. Check if the front index is equal to the rear index. If it is, the queue is empty and we cannot dequeue any elements.

   b. Otherwise, retrieve the element at the front index, increment the front index, and return the retrieved element.

5. To check if the queue is empty, check if the front index is equal to the rear index.

6. To check if the queue is full, check if the rear index is equal to n-1.

Conditions for overflow and underflow:

- Overflow: Occurs when we try to enqueue an element into a full queue, i.e., when the rear index is equal to the size of the array minus one.

- Underflow: Occurs when we try to dequeue an element from an empty queue, i.e., when the front index is equal to the rear index.

Programming Assignment: Write a program to implement a queue of alphabets using circular arrays. The program should contain following functions.

1. IsEmpty() – to check if the queue is empty

2. Add an item to the queue

3. Remove an item from the queue

4. Display the items of the queue from first to last.

Code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10

// Initialize queue and its variables
char queue[MAX_SIZE];
int front = -1, rear = -1;

// Function to check if the queue is empty
int isEmpty() {
   if (front == -1 && rear == -1)
      return 1;
   else
      return 0;
}

// Function to add an element to the rear of the queue
void enqueue(char element) {
   if ((rear + 1) % MAX_SIZE == front) {
      printf("Queue is full.\n");
      return;
   } else if (isEmpty()) {
      front = 0;
      rear = 0;
   } else {
      rear = (rear + 1) % MAX_SIZE;
   }
   queue[rear] = element;
   printf("%c has been enqueued.\n", element);
}

// Function to remove an element from the front of the queue
void dequeue() {
   if (isEmpty()) {
```

```
        printf("Queue is empty.\n");
        return;
    } else if (front == rear) {
        front = -1;
        rear = -1;
    } else {
        front = (front + 1) % MAX_SIZE;
    }
    printf("%c has been dequeued.\n", queue[(front + MAX_SIZE - 1) % MAX_SIZE]);
}

// Function to display the elements in the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
        return;
    }
    printf("The elements in the queue are: ");
    int i;
    for (i = front; i != rear; i = (i + 1) % MAX_SIZE) {
        printf("%c ", queue[i]);
    }
    printf("%c\n", queue[i]);
}

int main() {
    enqueue('A');
    enqueue('B');
    enqueue('C');
    display();
    dequeue();
    display();
    enqueue('D');
    enqueue('E');
    enqueue('F');
    display();
    dequeue();
    dequeue();
    dequeue();
    dequeue();
    dequeue();
    dequeue();
    display();
    return 0;
}
```

Output

A has been enqueued.

B has been enqueued.

C has been enqueued.

The elements in the queue are: A B C

A has been dequeued.

The elements in the queue are: B C

D has been enqueued.

E has been enqueued.

F has been enqueued.

The elements in the queue are: B C D E F

B has been dequeued.

C has been dequeued.

D has been dequeued.

E has been dequeued.

F has been dequeued.

Queue is empty**.**

Experiment 8

Aim: To implement binary search trees.

Pre-Lab Work: What are the cases encountered during deletion of a key from a BST? How are they handled?

When deleting a key from a Binary Search Tree (BST), there are three cases to consider:

1. Node has no children (leaf node): In this case, we simply remove the node from the tree.

2. Node has one child: In this case, we replace the node with its child. If the node is the left child of its parent, then the child becomes the new left child of the parent. If the node is the right child of its parent, then the child becomes the new right child of the parent.

3. Node has two children: In this case, we need to find the node's inorder successor (the smallest node in the right subtree) or inorder predecessor (the largest node in the left subtree), and replace the node with the inorder successor/predecessor. Then we need to delete the inorder successor/predecessor from its original position in the tree.

Here's how these cases can be handled:

1. If the node has no children (leaf node), simply remove the node from the tree.

2. If the node has one child, replace the node with its child.

3. If the node has two children, find the inorder successor/predecessor of the node, replace the node with the inorder successor/predecessor, and delete the inorder successor/predecessor from its original position in the tree. The inorder successor/predecessor can be found recursively by traversing either the left subtree (in case of inorder predecessor) or the right subtree (in case of inorder successor).

During the deletion process, we need to make sure that the BST property is preserved at all times, i.e., the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree of a node contains only nodes with keys greater than the node's key

Programming Assignment: Write a program to implement a binary search tree containing numbers as key elements using linked implementation.

The program should contain following functions

1. **Inorder traversal**
2. **Preorder traversal**
3. **Postorder traversal**
4. **Insert a key**
5. **Delete a key**

Code

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

struct node {
   int key;
   struct node *left;
   struct node *right;
};

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

struct node *insert(struct node *root, int key) {
   if(root == NULL) {
      return createNode(key);
   }
   if(key < root->key) {
      root->left = insert(root->left, key);
   }
   else if(key > root->key) {
      root->right = insert(root->right, key);
   }
   return root;
}

struct node *delete(struct node *root, int key) {
   if(root == NULL) {
      return root;
   }
   if(key < root->key) {
```

```
        root->left = delete(root->left, key);
    }
    else if(key > root->key) {
        root->right = delete(root->right, key);
    }
    else {
        if(root->left == NULL) {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if(root->right == NULL) {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        struct node *temp = root->right;
        while(temp->left != NULL) {
            temp = temp->left;
        }
        root->key = temp->key;
        root->right = delete(root->right, temp->key);
    }
    return root;
}

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

void preorderTraversal(struct node *root) {
    if(root != NULL) {
        printf("%d ", root->key);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

void postorderTraversal(struct node *root) {
    if(root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->key);
```

```
    }
}

int main() {
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\nPreorder Traversal: ");
    preorderTraversal(root);
    printf("\nPostorder Traversal: ");
    postorderTraversal(root);
    root = delete(root, 20);
    printf("\nInorder Traversal after deleting 20: ");
    inorderTraversal(root);
    root = delete(root, 30);
    printf("\nInorder Traversal after deleting 30: ");
    inorderTraversal(root);
    return 0;
}
```

OUTPUT

```
Inorder Traversal: 20 30 40 50 60 70 80
Preorder Traversal: 50 30 20 40 70 60 80
Postorder Traversal: 20 40 30 60 80 70 50
Inorder Traversal after deleting 20: 30 40 50 60 70 80
Inorder Traversal after deleting 30: 40 50 60 70 80
```

Experiment 9

Aim: To implement graphs & traversals

Pre-Lab Work:

1-> Compare the two graph representation methods:

The adjacency matrix is a two-dimensional array that represents the presence or absence of edges between vertices. It requires O(V^2) space and is faster for checking the presence or absence of an edge between two vertices, but slower for iterating over adjacent edges.

The adjacency list is a collection of lists where each list represents a vertex and contains the vertices that are adjacent to it. It requires O(E + V) space and is faster for iterating over adjacent edges, but slower for checking the presence or absence of an edge.

2-> Differentiate between BFS and DFS of graphs

BFS (Breadth-First Search) is an algorithm that starts at a specified vertex of a graph and visits all the vertices at the same level before moving to the next level. It explores the graph in a breadth-first manner, i.e., it visits all the neighbors of a vertex before moving to the next level of vertices. BFS uses a queue data structure to keep track of the vertices that need to be visited next. The time complexity of BFS is O(V + E), where V is the number of vertices and E is the number of edges in the graph.

DFS (Depth-First Search) is an algorithm that starts at a specified vertex of a graph and explores as far as possible along each branch before backtracking. It explores the graph in a depth-first manner, i.e., it visits all the vertices in a path before backtracking to explore another path. DFS uses a stack data structure to keep track of the vertices that need to be visited next. The time complexity of DFS is also O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Programming Assignment: Write a program to implement a graphs containing numbers as key elements using adjacency lists and adjacency matrix. Write functions for breadth first and depth first graph traversal.

Code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

// Structure for adjacency list node
struct AdjListNode {
   int dest;
   struct AdjListNode* next;
};

// Structure for adjacency list
struct AdjList {
   struct AdjListNode* head;
};

// Structure for graph
struct Graph {
   int V;
   struct AdjList* array;
};

// Create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest) {
   struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
   newNode->dest = dest;
   newNode->next = NULL;
   return newNode;
}

// Create a new graph
struct Graph* createGraph(int V) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
   for (int i = 0; i < V; ++i)
      graph->array[i].head = NULL;
```

```
    return graph;
}

// Add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->V; ++v) {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (pCrawl) {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

// Depth-First Search Traversal
void DFS(struct Graph* graph, int v, int visited[]) {
    visited[v] = 1;
    printf("%d ", v);
    struct AdjListNode* pCrawl = graph->array[v].head;
    while (pCrawl) {
        int adjacentNode = pCrawl->dest;
        if (!visited[adjacentNode]) {
            DFS(graph, adjacentNode, visited);
        }
        pCrawl = pCrawl->next;
    }
}

// Breadth-First Search Traversal
void BFS(struct Graph* graph, int start) {
    int visited[MAX_SIZE] = { 0 };
    int queue[MAX_SIZE];
```

```
    int front = 0;
    int rear = 0;
    queue[rear] = start;
    visited[start] = 1;
    while (front <= rear) {
        int v = queue[front];
        printf("%d ", v);
        front++;
        struct AdjListNode* pCrawl = graph->array[v].head;
        while (pCrawl) {
            int adjacentNode = pCrawl->dest;
            if (!visited[adjacentNode]) {
                visited[adjacentNode] = 1;
                rear++;
                queue[rear] = adjacentNode;
            }
            pCrawl = pCrawl->next;
        }
    }
}

int main() {
    int V = 5;
    struct Graph* graph = createGraph(V);

    // Adding edges to the graph
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1,
```

]

output

Adjacency list of vertex 0
head -> 4-> 1

Adjacency list of vertex 1
head

Adjacency list of vertex 2
head

Adjacency list of vertex 3
head

Adjacency list of vertex 4
head -> 1

Depth-First Traversal:
0 1 4

Breadth-First Traversal:
0 1 4

Experiment 10

Aim: To find minimum spanning tree of given graph

Pre-Lab Work:

1. What is minimum spanning tree?

A minimum spanning tree (MST) is a subset of the edges of an undirected weighted graph that connects all the vertices together without forming any cycles and has the minimum possible total edge weight. In other words, it is the tree that has the minimum weight among all possible spanning trees of the graph.
MSTs have a wide range of applications in computer science and beyond, including network design, clustering, and optimization. Some of the well-known algorithms for finding MSTs include Prim's algorithm and Kruskal's algorithm.

2. Differentiate between Prim's and Kruskal's algorithms?

Feature	Prim's Algorithm	Kruskal's Algorithm
Approach	Greedy, grows MST from a single vertex	Greedy, merges trees together
Data Structure	Priority Queue or Min-Heap	Disjoint-Set Data Structure
Time Complexity	O(E log V) with Min-Heap; O(V log V + E) with Fibonacci Heap	O(E log E) with Disjoint-Set
Suitable for	Dense graphs with many edges	Sparse graphs with few edges

Programming Assignment: Write a program to find MST of a given graph.

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

#define V 5 // number of vertices in the graph

int minKey(int key[], bool mstSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}

int printMST(int parent[], int graph[V][V]) {
   int minWeight = 0;
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++) {
      printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
      minWeight += graph[i][parent[i]];
   }
   return minWeight;
}

void primMST(int graph[V][V]) {
   int parent[V]; // array to store MST
   int key[V]; // keys used to pick minimum weight edge
   bool mstSet[V]; // set of vertices not yet included in MST

   // initialize all keys as infinite and mstSet[] as false
   for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = false;

   // always include first vertex in MST
   key[0] = 0; // make key 0 so that this vertex is picked as first vertex
   parent[0] = -1; // first node is always root of MST

   // MST will have V vertices
```

```
    for (int count = 0; count < V - 1; count++) {
        // pick the minimum key vertex from the set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // add the picked vertex to the MST set
        mstSet[u] = true;

        // update key and parent arrays of adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the MST
    int minWeight = printMST(parent, graph);
    printf("Minimum Weight of the MST is %d\n", minWeight);
}

int main() {
    // create the graph
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // print the MST using Prim's algorithm
    primMST(graph);

    return 0;
}
```

OUTPUT

Edge Weight
0 - 1 2
1 - 2 3
0 - 3 6
1 - 4 5
Minimum Weight of the MST is 16