DATA STRUCTURES LAB RECORD
(20ES1356)

II B. TECH I SEM
CSE-S1

ACADEMIC YEAR: 2023-24

SUBMITED BY

Anjusri Kandi
22501A0512

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PRASAD V. POTLURI SIDDHARTHA INSTITUTE OF TECHNOLOGY

SIGNATURE OF STUDENT **SIGNATURE OF FACULTY IN-CHARGE**

S.No	Experiment No.	Topic	Date	Page No	Day to Day Marks	Record Marks	Signature of Faculty
1	2	**Searching**	11/8/2023	1-15			
	a	Linear Search (Description, example, algorithm, advantages,disadvantages,applications)					
	b	Perform Linear Search on Integers **(program, output,screenshot)**					
	c	Perform Linear Search on characters **(program, output,screenshot)**					
	d	Binary Search (Description, example, algorithm, advantages,disadvantages,applications)					
	e	Perform Binary Search on Integers **(program, output,screenshot)**					
	f	Perform Binary Search on Strings **(program, output,screenshot)**					
	i	Viva Questions and Answers on all searching techniques					
2	1	**Recursion**	18/08/2023	16-23			
	a	Description (Explanation,Limitations)					
	b	Factorial of a given number					

		(Algorithm, program, output)					
	c	Towers of Hanoi Problem **(Algorithm, program, output)**					
	d	Program to perform Binary Search using Recursion. **(program, output,screenshot)**					
3	3	**Sorting**	25/08/2023	24-48			
	a	Bubble Sort (Description, example, algorithm, advantages,disadvantages,applications, performance(time and space complexity)					
	b	Perform Bubble Sort on Integers **(program, output,screenshot)**					
	c	Selection Sort (Description, example, algorithm, advantages,disadvantages,applications, performance(time and space complexity)					
	d	Perform Selection Sort on Integers **(program, output,screenshot)**					
	e	Insertion Sort (Description, example, algorithm, advantages,disadvantages,applications, performance(time and space complexity)					
	f	Perform Insertion Sort on Integers **(program, output,screenshot)**					
	g	Merge Sort (Description, example, algorithm, advantages,disadvantages,applications, performance(time and space complexity)					

	h	Perform Merge Sort on Integers **(program, output,screenshot)**					
	i	Quick Sort (Description, example, algorithm, advantages,disadvantages,applications, performance(time and space complexity)					
	j	Perform Quick Sort on Integers **(program, output,screenshot)**					
	k	Comparison between all 5 sorting algorithms					
	o	Viva Questions and Answers on all Sorting techniques					
4	4	**Linked List**	08/09/2023	49-102			
	a	Single Linked List (Description, example, advantages,disadvantages,applications, performance(time and space complexity)					
	b	Operations on SLL (Algorithm or Pseudocode,diagram, time complexity for each operation) 1. insert at beginning 2. Display LIst 3. Count number of nodes 4.insert at end 5.insert at position 6. delete at the beginning 7. delete at the end 8. delete at position 9. Search for data					
	c	Program to perform operations on SLL **(program, output,screenshot for each operation)**					
	d	Double Linked List (Description, example,					

		advantages,disadvantages,applications, performance(time and space complexity)					
	e	Operations on DLL (Algorithm or Pseudocode, diagram, time complexity for each operation) 1. insert at beginning 2. Display LIst 3. Count number of nodes 4.insert at end 5.insert at position 6. delete at the beginning 7. delete at the end 8. delete at position 9. Search for data 10. Print elements in reverse order					
	f	Program to perform operations on DLL **(program, output,screenshot for each operation)**					
	g	Circular Single Linked List (Description, example, advantages,disadvantages,applications, performance(time and space complexity)					
	h	Operations on CLL (Algorithm or Pseudocode, diagram, time complexity for each operation) 1. insert at beginning 2. Display LIst 3. Count number of nodes 4.insert at end 5.insert at position 6. delete at the beginning 7. delete at the end 8. delete at position 9. Search for data 10. Delete duplicates in list					
	i	Program to perform operations on CLL **(program, output,screenshot for each operation)**					

	j	Comparison of Linked Lists with Arrays & Dynamic Arrays					
	k	Viva Questions and Answers					
5	**5**	**STACKS**	20/09/ 2023	103 - 122			
	A	STACKS (Description, example, advantages,disadvantages,applications)					
	B	Operations on Stack using Arrays (Algorithm or Pseudocode,diagram, time complexity for each operation) 1. Push 2 pop 3. Peek 4. Display 5. Count 6. IsFull 1. IsEmpty					
	C	Program to perform operations on stack using arrays **(program, output,screenshot for each operation)**					
	D	Operations on Stack using LinkedList (Algorithm or Pseudocode,diagram, time complexity for each operation) 1. Push 2 pop 3. Peek 4. Display 5. Count 6. IsEmpty					
	E	Program to perform operations on stack using linked list					

		(program, output,screenshot for each operation)					
	F	Comparison of Stack operations using Linked Lists with Arrays					
	G	Viva Questions and Answers					
6	**6**	**STACK APPLICATIONS**	22/09/ 2023	123 - 136			
	A	Program to perform **Infix to postfix conversion** (Description, example, program, output, screenshot of 2 examples)					
	B	Program to perform **balancing of parenthesis** (Description, example, program, output, screenshot of 2 examples)					
	C	Program to perform **postfix evaluation** (Description, example, program, output, screenshot of 2 examples)					
7	**7**	**QUEUES**	13/10/ 2023	137 - 159			
	A	QUEUES (Description, example, advantages,disadvantages,applications)					
	B	Operations on QUEUES using Arrays (Algorithm or Pseudocode,diagram, time complexity for each operation) 1. INSERT 2 DELETE 3. SIZE 4. Display 5. FRONT					

		6. REAR 7. IsEmpty 8. IsFull					
	C	Program to perform operations on queues using arrays **(program, output,screenshot for each operation)**					
	D	Operations on QUEUES using Linked List (Algorithm or Pseudocode,diagram, time complexity for each operation) 1. INSERT 2 DELETE 3. SIZE 4. Display 5. FRONT 6. REAR 7. IsEmpty					
	E	Program to perform operations on QUEUES using linked list **(program, output,screenshot for each operation)**					
	F	Operations on CIRCULAR QUEUES using Arrays (Algorithm or Pseudocode,diagram, time complexity for each operation) 1. INSERT 2 DELETE 3. SIZE 4. Display 5. FRONT 6. REAR 7. IsEmpty 8. IsFull					
	G	Program to perform operations on CIRCULAR QUEUES using arrays **(program, output,screenshot for each operation)**					

8	**8**	**APPLICATIONS OF QUEUES AND STACKS**	16/10/ 2023	160 - 168			
9	**9**	**Binary Trees**	03/11/ 2023	168 - 186			
	A	Implement Binary trees using Arrays and perform the following operations - inorder,preorder,postorder,levelorder traverslas -print leaf nodes -height of tree -min -max -search					
	B	Implement Binary trees using Linked Lists and perform the following operations -insert -delete - inorder,preorder,postorder,levelorder traverslas -print leaf nodes -height of tree -min -max -search -find height and depth of each node					
10	**10**	**Binary Search Trees**	06/11/ 2023	187 - 200			
	A	Implement Binary search trees using Arrays and perform the following					

		operations - inorder,preorder,postorder,levelorder traverslas -height of tree -min -max -search					
		Implement Binary Search trees using Linked Lists and perform the following operations -insert -delete - inorder,preorder,postorder,levelorder traverslas -height of tree -min -max -search -find height and depth of each node					
11	**11**	**Graphs**	10/11/2023	201 - 205			
	a	DFS					
	b	BFS					
11	**11**	**Graphs - Minimum cost spanning trees**	17/11/2023	206 - 213			
	a	Prim's Algorithm					
	b	Kruskals Algorithm					

Experiment 1: SEARCHING

Aim: Implement different types of searching techniques on a given list

(i) Linear search

(ii) Binary search

Description:

Searching is done to find an element or information needed in a group of data. There are several search algorithms to find data in linear list while mostly used search algorithms are linear search and binary search.

a.Linear Search:

In linear search, in order to find an element in an array, element to be searched is compared with each and every element of the array from the starting index to last.

Algorithm:

Step 1: Elements of array are scanned as per choice

Step 2: Element to be searched in the array(i.e key) is also taken

Step 3: Initializaton of looping variable to 1

Step 4: Until I becomes n, repeat:

4.1: Check if element at index I is equal to key value if yes return index i

Step 5: if element is not found -1 is returned

Example:

i	0	1	2	3	4
Arr[]	87	36	5	12	9

Here number of elements in the given array n=5

Key=9

i	0	1	2	3	4
Arr[i]	87	36	5	12	9

1.At i=0, Arr[0]=87------87=9? ---False Hence i=i+1=1

2. At i=1 , Arr[1]=36 -------36=9?---- False Hence i=i+1=2

3.At i=2 , Arr[2]=5 --------5=9?-----False Hence i=i+1=3

4.At i=3 , Arr[3]=12 --------12=9?----False Hence i=i+1=4

5.At i=4 , Arr[4]=9 ---------9=9?-----True Hence i=4

Element searched is found at index 4 i.e at 5^{th} position.

Advantages:

- Element in small to medium sized arrays can be searched relatively quickly.
- List does not need to be sorted.
- It is not affected by insertions or deletions.

Disadvantages:

- It has greater time complexity compared to other searching algorithms.
- Worst case of time complexity is to search element at last position.

Applications:

- Phonebook Search
- Spell checkers

b. Linear Search on Integers

Source Code:

```
#include<stdio.h>
int linear_integers(int[],int,int);
int main()
{
  int n,target;
  printf("Enter the size of array");
  scanf("%d",&n);
  int a[n];
  for(int i=0;i<n;i++)
 {
   scanf("%d",&a[i]);
 }
 printf("Enter the element to be searched:");
 scanf("%d",&target);
 int index=linear_integers(a,n,target);
 printf("Element is found at %d",index);
 return 0;
}
int linear_integers(int a[],int n,int target)
{
  for(int i=0;i<n;i++)
 {
   if(target==a[i])
        return i;
 }
 return -1; }
```

Input:

n=4, a[]={6,14,18,24},target=9

Output:

Element is found at -1

c. Linear Search on Characters

Source Code:

```
#include<stdio.h>
int linear_characters(char[],int,char);
int main()
{
   Int n;
   char target;
  char a[]={'a','k','j','s'};
  n=sizeof(a)/sizeof(a[0]);
  printf("Enter the element to be searched:");
  scanf("%c",&target);
  int index=linear_characters(a,n,target);
  printf("Element is found at %d",index);
  return 0;
}
int linear_characters(char a[],int n,char target)
{
   for(int i=0;i<n;i++)
  {
     if(target==a[i])
          return i;
  }
  return -1;
}
```

Input:

target =j

Output:

Element is found at 2

d. Linear Search on Strings

Source Code:

```
#include<stdio.h>
#include<string.h>
int linear_string(char[][],int,char[]);
int main()
{
  int n;
 char target[10];
 char a[][8]={“Anju”,”Jyothi”,”Srinivas”,”Alekhya”};
 n=sizeof(a)/sizeof(a[0]);
 printf(“Enter the string to be searched:”);
 scanf(“%s”,&target);
 int index=linear_string(a,n,target);
 printf(“Element is found at %d”,index);
 return 0;
}
int linear_string(char a[][],int n,char target[])
{
  for(int i=0;i<n;i++)
 {
   if(strcmp(target,a[i])==0)
        return i;
 }
 return -1;
}
```

Input:

target =sujatha

Output:

Element is found at -1

e. Binary Search:

In binary search, in order to find an element in an array, since array is sorted array is divided into two parts so to go traverse the array lesser number of times compared to linear search. Element is compared to the element at mid index. if the key value is greater than mid value right of the mid is searched.

Algorithm:

Step 1: Elements of array are scanned as per choice

Step 2: Element to be searched in the array(i.e key) is also taken

Step 3: Initializaton of two variables low and high to 0 and (n-1) and declaration of mid

Step 4: Until low<=high, repeat:

4.1:mid value is calculated i.e (low+high)/2

4.2: Check if element at mid is equal to key value if yes return index mid

4.3:if element at mid is greater then low is initialized to mid-1

4.4:else high is initialized to mid+1

Step 5: if element is not found -1 is returned

Example:

i	0	1	2	3	4
Arr[]	1	3	5	12	20

Here number of elements in the given array n=5

Key=9

i	0	1	2	3	4
Arr[i]	1	3	5	12	20

1.At low=0, high=5-1=4------mid=(0+4)/2=2 ---Arr[2]=5----9=5?----False-----5>9? --False Hence low=mid+1=3

2. At low=3 , high=4-----mid=(3+4)/2=3----Arr[3]=12-------12=9?---- False---12>9---True Hence high=mid-1=2

3.At low=3 ,high=2-----low<=high----False

Element searched is found at index -1 i.e not found.

Advantages:

- Element in large arrays can be searched relatively quickly.
- It does not scan each element in the list.
- Time Complexity: O(log n)

Disadvantages:

- List need to be sorted.
- The process of sorting and searching in an unsorted array will take time. Thus, binary search is not a good option in such cases.

Applications:

- Spell checkers
- Huffman Coding algorithm

f. Binary Search on Integers

Source Code:

```
#include<stdio.h>
int binary_integers(int[],int,int);
int main()
{
  int n,target;
  printf("Enter the size of array");
  scanf("%d",&n);
  int a[n];
  for(int i=0;i<n;i++)
 {
   scanf("%d",&a[i]);
 }
 printf("Enter the element to be searched:");
 scanf("%d",&target);
 int index=binary_integers(a,n,target);
 printf("Element is found at %d",index);
 return 0;
}
int binary_integers(int a[],int n,int target)
{
  int low=0;
  int high=n-1;
  int mid;
  while(low<=high)
  {
    mid=(low+high)/2;
    if(target==a[mid])
        return i;
    else if(target>a[mid])
        low=mid+1;
    else
         high=mid-1;
```

```
        }
        return -1; }
```

Input:

n=4

a[]={6,14,18,24}

target=9

Output:

Element is found at -1

g. Binary Search on Characters

Source Code:

```
#include<stdio.h>
int binary_characters(char[],int,char);
int main()
{
  char a[]={'a','l','v','w'};
  int n=sizeof(a)/sizeof(a[0]);
 printf("Enter the element to be searched:");
 scanf("%c",&target);
 int index=binary_characters(a,n,target);
 printf("Element is found at %d",index);
 return 0;
}
int binary_characters(char a[],int n,char target)
{
  int low=0;
  int high=n-1;
  int mid;
  while(low<=high)
  {
    mid=(low+high)/2;
    if(target==a[mid])
        return i;
    else if(target>a[mid])
        low=mid+1;
    else
         high=mid-1;
 }
 return -1;
}
```

Input:

target=c

Output:

Element is found at -1

h. Binary Search on Strings

Source Code:

```
#include<stdio.h>
#include <string.h>
int binary_search(char a[][30], int n, char *search){
int low=0,high=n-1;
int mid;
while(low<=high)
{
mid=(low+high)/2;
if(strcmp(search,a[mid])==0)
 return mid;
 else if(search>a[mid])
 low=mid+1;
 else
 high=mid-1;
  }
return -1;
}
int main(char argv[], int argc){
char arr[][30]={"hi", "hello","Namaste","maa"};
 int size=sizeof(arr)/sizeof(arr[0]);
char *target;
```

```
int result;
printf("Enter the value to search: ");
 scanf("%s", target);
result=binary_search (arr, size, target);
 if (result==-1)
printf("%s is not found in the array", target);
 else
printf("%s is found at %d", target, result);
return 0;
}
```

Input:

target=c

Output:

Element is found at -1

Linear Search	Binary Search
In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.
It is also called sequential search.	It is also called half-interval search.
The time complexity of linear search **O(n)**.	The time complexity of binary search **O(log n)**.
Multidimensional array can be used.	Only single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is very slow process.	It is very fast process.

j. Viva Questions and Answers

1. What is the time complexity of linear search and binary search?

Ans: Time complexity of linear search : O(n)

Time complexity of binary search:O(log n)

2. What is the disadvantage of linear search?

Ans: Even if duplicates are present only first appeared is considered

3. What is the disadvantage of binary search?

Ans: Given array must be sorted otherwise to do binary search first programmer has to sort the array which makes the code more complex

4. When was linear search preferred to use?

Ans: For small sized arrays linear search can be used while for large sized Arrays binary search is used.

Result:

Thus the programs for implementing searching techniques like Sequential search and Binary search are executed.

Experiment 2: RECURSION

Aim:

Demonstrate recursive algorithms with examples.

(i) Factorial of a Number
(ii) Towers of Hanoi Problem
(iii) Binary search

a.Description:

A function that is called repeatitively directly or indirectly by itself is known as recursion. Recursion actually reduces the complexity of code for the programmer but not for the computer.

Recursion has three basic necessities:

1. Base condition
2. Recursive call
3. Recursive call must incline to the base condition

Limitations:

- Recursion increases the space complexity and time complexity.
- Recursive solution should be shorter and understandable only then it can be used.
- Recursion should be used only then when it is suitable for the algorithm.

Syntax:

```
Return_type function_name(arguments)

{

if(base condition)

    return res;

else

   Recursive call;
```

```
}
```

b. Factorial of a given number

Algorithm:

Step 1: Read the value of n from the main function, number to which factorial need to be calculated.

Step 2: if n becomes zero return 1

Step 3: Otherwise return n*function_name(n-1)

Since Factorial of n=n*(n-1)!=n*(n-1)*(n-2)!=…….

Source Code:

```
#include<stdio.h>
int fact(int);
int main()
{
   int n;
   printf(“Enter an integer:”);
   scanf(“%d”,&n);
   int factorial=fact(n);
   printf(“Factorial of %d is %d”,n,factorial);
   return 0;
}
int fact(int n)
{
```

```
if(n==0)       //Base condition
    return 1;
else
    return n*fact(n-1);     //Recursive call
}
```

Input:

n=6

Output:

Factorial of 6 is 720

c. Towers of Hanoi

Algorithm:

Step 1: Scan the values of n(number of disks),src,dest,aux-pole names

Step 2: While n greater than or equal to one repeat

Step 2.1: (n-1)th disk is placed in auxiliary rod

Step 2.2: Write move top disk from pole src to top of pole dest

Step 2.3: (n-1)th disk is passed to destination pole from auxiliary rod

Source Code:

```
#include<stdio.h>
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
   if (n == 1)
   {
      printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
      return;
   }
   towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
   printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
   towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
   int n ;    // Number of disks
   printf(“Enter number of disks:”);
   scanf(“%d”,&n);
   towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
   return 0;
}
```

Output:

```
Move disk 1 from rod A to rod B
 Move disk 2 from rod A to rod C
```

Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C

d. Program to perform Binary Search using Recursion.

Algorithm:

Step 1: Initialize l to 0 and r to n-1
Step 2: if r is greater than or equal to l
    Step 2.1: Calculate the value of mid
    Step 2.2: Check if element present at mid index is equal to x,if yes return mid
    Step 2.3: Otherwise if element present at mid index is greater than x recursive call is made with r as mid-1
    Step 2.4: Else l to mid+1
Step 3: if element is not present -1 is returned

Source Code:

```
int binarySearch(int arr[], int l, int r, int x)
{
if (r >= l)
{
int mid = l + (r - l)/2;
if (arr[mid] == x)
    return mid;
if (arr[mid] > x)
    return  binarySearch(arr, l, mid-1, x);
return  binarySearch(arr, mid+1, r, x);
}
return -1;
}
int main()
{
int arr[] = {2, 3, 4, 10, 40};
int n = sizeof(arr)/ sizeof(arr[0]);
int x = 10;
int result = binarySearch(arr, 0, n-1, x);
(result == -1)? printf("Element is not present in array")
                : printf("Element is present at index %d", result);
return 0;
}
```

Output:

Element is present at index 3

e. Differences between Recursion and Iteration

Recursion	Iteration
Function calls itself.	A set of instructions repeatedly executed.
For functions.	For loops.
Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Smaller code size	Larger Code Size.
Very high(generally exponential) time complexity.	Relatively lower time complexity (generally polynomial-logarithmic).
The space complexity is higher than iterations.	Space complexity is lower.
Here the stack is used to store local variables when the function is called.	Stack is not used.
Execution is slow since it has the overhead of maintaining and updating the stack.	Normally, it is faster than recursion as it doesn't utilize the stack.
Recursion uses more memory as compared to iteration.	Iteration uses less memory as compared to recursion.
Possesses overhead of repeated function calls.	No overhead as there are no function calls in iteration.
If the recursive function does not meet to a termination condition or the base case is not defined or is never reached then it leads to a stack overflow error and there is a chance that the an system may crash in infinite recursion.	If the control condition of the iteration statement never becomes false or the control variable does not reach the termination value, then it will cause infinite loop. On the infinite loop, it uses the CPU cycles again and again.

f. Viva Questions and Answers

1. What is recursion?

The process of calling of function by itself is known as recursion.

2. How is recursion different from iteration?

Recursion is a repetitive function call by the same function while iteration is a execution of statements repeatitively controlled by a looping variable.

3.What are the three basic necessities of recursion?

- A recursive algorithm must have a base case.
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself, recursively.

4.How many types of recursive call are there?

Two types of recursive call are there

- Direct recursive call
- Indirect recursive call

5.Which data structure is used in recursion?

Stack data structure is used for implementation of recursion.

6. Why the space complexity of recursion is relatively higher ?

For every function call, it stores data and hence occupies more space.

7. What is the limitation of recursion?

Recursive solutions may involve extensive overhead as they use function calls.

Result:

Thus the programs for implementing Factorial, Towers of Hanoi & Binary search using recursion is executed.

Experiment 3: SORTING

a.Bubble Sort

Description:

In bubble sort, in each and every pass one element gets to its position.In 1st pass maximum element gets to its place.Every element is compared with its adjacent element and gets swapped if not present in ascending order.

Example:

I	0	1	2	3	4
A	89	12	-54	99	0

Here n=5

Pass 1: i. 89 is compared with 12. Since 89 is greater, 89 is swapped with 12

A[0]>A[1](T)

12 89 -54 99 0

ii. 89 is compared with -54. Since 89 is greater , 89 is swapped with -54

A[1]>A[2](t)

12 -54 89 99 0

iii. 89 is compared with 99. Since 99 is greater , 99 is not swapped with 89

A[2]>A[3](F)

12 -54 89 99 0

iv. 99 is compared with 0. Since 99 is greater , 99 is swapped with 0

A[3]>A[4](t)

12 -54 89 0 99

Pass 2: i. 12 is compared with -54. Since 12 is greater , 12 is swapped with -54

A[0]>A[1](t)

-54 12 89 0 99

ii. 89 is compared with 12. Since 89 is greater , 89 is not swapped with 12

A[1]>A[2](F)

-54 12 89 0 99

iii. 89 is compared with 0. Since 89 is greater , 89 is swapped with 0

A[2]>A[3](t)

-54 12 0 89 99

Pass 3: i. 12 is compared with -54. Since 12 is greater , 12 is not swapped with -54

A[0]>A[1](F)

-54 12 0 89 99

ii. 12 is compared with 0. Since 12 is greater ,12 is swapped with 0

A[1]>A[2](T)

-54 0 12 89 99

Pass 4: i. 0 is compared with -54. Since 0 is greater , 0 is not swapped with -54

A[0]>A[1](F)

-54 0 12 89 99

Algorithm:

- traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

Advantages:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages:

- Bubble sort has a time complexity of $O(N^2)$ which makes it very slow for large data sets.
- It makes more number of swaps.

Applications:

- Phone contacts
- It is used to sort shorter datasets

Performance:

Time Complexity: $O(N^2)$
Auxiliary Space: O(1)

b. Bubble Sort on integers:

Source Code:

```
#include<stdio.h>
int main()
{
   int j,i,n;
   printf("Enter the size if array:");
   scanf("%d",&n);
   int a[n];
   printf("Enter the elements of array:");
   for(i=0;i<n;i++)
   {
      scanf("%d",&a[i]);
   }
   for(i=1;i<=n-1;i++)
   {
       for(j=0;j<n-i;j++)
       {
          if(a[j]>a[j+1])
          {
             a[j]=a[j]+a[j+1]-(a[j+1]=a[j]);
          }
       }
   }
   printf("Elements in the array after sorting:");
   for(i=0;i<n;i++)
   {
      printf("%d ",a[i]);
   }
   return 0;
}
```

Input:

N=6 a[]={39 50 14 9 -1 0}

Output:

-1 0 9 14 39 50

c. Selection Sort

Description:

In selection sort, in each and every pass one element gets to its position.In 1st pass minimum element gets to its place.

Example:

I	0	1	2	3	4
A	89	12	-54	99	0

Here n=5

Pass 1: i. Min=a[0]=89 --- Min>a[1](T) --- Min=a[1]=12

89 12 -54 99 0

Min>a[2](T) --- Min=a[2]=-54

89 12 -54 99 0

Min>a[3](F) --- Min=a[2]=-54

89 12 -54 99 0

Min>a[4](F) --- Min=a[2]=-54

89 12 -54 99 0

Swap(-54,89)

-54 12 89 99 0

Pass 2: i. Min=a[1]=12 --- Min>a[2](F)

-54 12 89 99 0

Min> a[3](F)

Min>a[4](T) ---Swap (a[4],a[2])

-54 0 89 99 12

Pass 3: i. Min=a[2]=89 ----Min>a[3](F)

-54 0 89 99 12

Min>a[4](T)

Swap(Min,a[2])

-54 0 12 99 89

Pass 4: Min=a[3]=99 --- Min>a[4](T)

Swap(a[3],Min)

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

Advantages:

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages:

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Applications:

- It is used to sort smaller datasets.

Performance:

Time complexity: O(n^2)

Space complexity: O(1)

d. Selection Sort on Integers:

Source Code:

```
#include<stdio.h>
int main()
{
   int n,i,j;
   printf("Enter the size of array:");
   scanf("%d",&n);
   int a[n];
   printf("Enter the elements of an array:");
   for(i=0;i<n;i++)
   {
      scanf("%d",&a[i]);
   }
   int index,min;
   for(i=0;i<n-1;i++)
   {
      index=i;
      min=a[index];
      for(j=i+1;j<n;j++)
      {
         if(min>a[j])
         {
           index=j;
           min=a[index];
         }
      }
      a[i]=a[i]+a[index]-(a[index]=a[i]);
   }
   printf("Elements in the array after swapping:");
   for(i=0;i<n;i++)
   {
      printf("%d ",a[i]);
   }
   return 0;
}
```

Input:

N=6 a[]={39 50 14 9 -1 0}

Output:

-1 0 9 14 39 50

e. Insertion Sort:

Description:

In insertion sort , given list is divided into two parts: sorted and unsorted, the first value of unsorted list is considered as key. It is compared with elements in the sorted list and is paced in its right palce. Hence in each pass one element is placed in its right position.

Example:

12 | 6 9 -2 42 -1

↓

key

Pass 1: 6 12 | 9 -2 42 -1

6 is compared with 12 and is placed ina position before 12

Pass 2: 6 9 12 |-2 42 -1

9 is compared with 12 and then with 6 since it is between 6 and 12 it is placed in between 6 and 12

Pass 3: -2 6 9 12 | 42 -1

-2 is compared with all elements in sorted list one by one and is placed at its right position

Pass 4: -2 6 9 12 42 | -1

Since 42 is greater than all elements in sorted list it is placed at the end of the sorted list

Pass 5: -1 -2 6 9 12 42

Since -1 is smaller than all elements it is places at the starting position

All elements are sorted.

Algorithm:

Step 1: Initialize I to 0

Step 2:while I is less than n

Step 2.1: Initialize j to I and key to a[i+1]

Step 2.2:while(j>=0 && a[j]>key)

Step 2.2.1:Place a[j] in [j+1]th position and key value in jth index

Step 2.2.2: Decrement j by 1

Step 3: Print array elements

Advantages:

- Simple implementation
- Insertion sort is efficient for small data
- It is adaptable
- It is more efficient than selection and bubble sort because of less number of comparisons
- It is stable
- It is an in-place algorithm

Disadvantages:

- It is not suitable for large datasets.
- Worst case time complexity is O(n^2)

Applications:

- It is used to sort smaller datasets

Performance:

Time Complexity: O(n^2)
Auxiliary Space: O(1)

f. Insertion sort on integers

Source Code:

```
#include<stdio.h>
int main()
{
   int j,i,n,key;
   printf("Enter the size of array:");
   scanf("%d",&n);
   int a[n];
   printf("Enter the elements of array:");
   for(i=0;i<n;i++)
   {
      scanf("%d",&a[i]);
   }
   i=0;
   while(i<n)
   {
      j=i;
      key=a[i+1];
      while(j>=0)
      {
         if(a[j]>key)
         {
            a[j+1]=a[j];
            a[j]=key;
            j--;
         }
         else
            break;
      }
      i++;
   }
   printf("Elements in the array after sorting:");
   for(i=0;i<n;i++)
```

```
    {
        printf("%d ",a[i]);
    }
    return 0;
}
```

Input:

N=7 a[]={8,67,34,0,-12,90,198}

Output:

0 -12 8 34 67 90 198

g. Merge Sort

Description:

It follows divide and conquer rule, Given data is divided into smaller data and after sorting smaller datasets , entire data is merged .

Example:

Algorithm:

Step 1:Initialize low to 0 and hight to n-1

Step 2:if low< high

Step 2.1: calculate mid if low and high

Step 2.2: Sort the elements of 1^{st} half array i.e from 0 to mid

Step 2.3: Sort the elements of 2^{nd} half of the array from mid+1 to high

Step 2.4:Merge both the halves of array

Advantages:

- **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Disadvantages:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.

- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

Applications:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

Performance:

Time complexity: $T(n) = 2T(n/2) + \theta(n)$

Source Code:

```
#include <stdio.h>
void merge(int arr[],int low,int mid,int high){
   int arr1[high-low+1];
   int i=low,j=mid+1,k=0;
   while(i<=mid&&j<=high){
      if(arr[i]>arr[j])
      arr1[k++]=arr[j++];
      else
     arr1[k++]=arr[i++];

   }
   while(i<=mid)
   arr1[k++]=arr[i++];

  while(j<=high)
   arr1[k++]=arr[j++];
   for(int i=0;i<high-low+1;i++){
      arr[i+low]=arr1[i];
   }
}
void merge_sort(int arr[],int low,int high){
   if(low>=high){
```

```
    }
else{
   int mid=low+(high-low)/2;
   merge_sort(arr,low,mid);
   merge_sort(arr,mid+1,high);
   merge(arr,low,mid,high);
}
}
int main() {
   printf("Enter the array size\n");
   int n;
   scanf("%d",&n);
   int arr[n];
   printf("Enter the array elements\n");
   for(int i=0;i<n;i++)
     scanf("%d",&arr[i]);
     merge_sort(arr,0,n-1);
     printf("The sorted array elements using merge\n");
   for(int i=0;i<n;i++)
   printf("%d\n",arr[i]);
   return 0;
}
```

j. Quick Sort:

Source Code:

```
#include <stdio.h>
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
int partition(int arr[], int low, int high)
{
   int pivot = arr[high];
   int i = (low - 1);
   for (int j = low; j <= high - 1; j++) {
      if (arr[j] < pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
void quickSort(int arr[], int low, int high)
```

```
{
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main()
{
   int n;
    printf("Enter the array size\n");
    scanf("%d",&n);
    printf("Enter array elements\n");
    int arr[n];
    for(int i=0;i<n;i++)
    scanf("%d",&arr[i]);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Experiment 4: Linked List

c. PROGRAM TO PERFORM OPERATIONS ON SLL

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

structnode{ int
data;
structnode *next;
};

structnode*head;
int count=0;

voidinsert_begin(){
   int value;
   printf("enterdata to be inserted");
   scanf("%d",&value);
   structnode *temp;
   temp = (struct node *)malloc(sizeof(struct node *));
   if(temp==NULL)
      printf("memory insufficient\n");
   else
   {

       temp->data=value;
    if(head==NULL){
         temp->next=NULL;
    }

    else{
         temp->next=head;
    }

    head=temp;
    count++;
   }

}
```

```
void display(){

if(head==NULL)
   printf("listisempty\n");
else{
   structnode *p;
    p=head;
   while(p->next!=NULL)
   {

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

   printf("%d\n",p->data);
}

}


voidinsert_end(){ int
   value;
   printf("enterdata to be inserted");
   scanf("%d",&value);
   structnode *temp;
   temp = (struct node *)malloc(sizeof(struct node *));
   if(temp==NULL)
      printf("memory insufficient\n");
   else{
      temp->data=value;
    temp->next=NULL;
    if(head==NULL){
         head=temp;
    }
    else{
        structnode*p;
        p=head;
        while(p->next!=NULL)
            p = p->next;
        p->next=temp;
```

```
    }

    count++;
   }

}


voidinsert_position(){

   int value,loc,i;

   printf("enterwhichlocationdatatobeinserted");
   scanf("%d",&loc);
   if(loc>0&&loc<=count+1){
   struct node *temp;
   temp = (struct node *)malloc(sizeof(struct node *));
   if(temp==NULL)
     printf("memory insufficient\n");
   else
   {

     if(loc==1)
        insert_begin();
     elseif(loc==count+1)
         insert_end();
     else
     {

        printf("enterdata to be inserted");
        scanf("%d",&value);
        struct node *p;
        p=head;
        for(i=1;i<=loc-2;i++)
        p = p->next;
        temp->data=value;
        temp->next= p->next;
        p->next=temp;
        count++;
     }

   }
```

```
    }

    else
      printf("cant insert inthegivenlocation");

  }

  Void delete_begin(){
  if(head==NULL)
    printf("list is empty can not perofrm delete operation");
  else{
    structnode*p;
    p=head;
    head=head->next;
free(p);
    count--;
    printf("firstnodeisdeletedsuccessfully");
  }

  }


  voiddelete_end(){
  if(head==NULL)
    printf("list is empty can not perform delete operation\n");
  else{
    structnode*p,*q; p =
     head;
     while(p->next->next != NULL)
       p=p->next;
     q=p;
     p=p->next;
     q->next=NULL;
     free(p);
     count--;
     printf("lastnodeisdeletedsuccessfully\n");
  }

  }


  voiddelete_position(){
  int loc,i;
  printf("Enter which location data to be deleted");
```

```
scanf("%d",&loc);
if(loc>=1&&loc<=count){
   if(loc==1)
      delete_begin();
   elseif(loc==count)
      delete_end; else{
   struct node *p,*q;
   p=head;
   for(i=1;i<=loc-2;i++)
   p=p->next;
   q=p;
   p=p->next;
   q->next=p->next;
      count--;
   printf("deletion isperformedatgiven location");
   }

}

else
    printf("deletionisnotperformedatgivenlocation");

}

voidsearch(){
int value,loc=1,found=0; printf("Enter
the data to search:");
scanf("%d",&value);
structnode*p;
p=head;
while(p->next!=NULL){
   if(p->data==value){
      printf("Found the data at position %d",loc); found=1;
      break;
}

loc++;
p=p->next;
}

if(p->data==value&&found==0)
   {

       printf("found the data at position %d",loc); found=1;
```

```
    }

if(found==0)
    printf("dataisnot found\n");

}

intmain(){
intchoice;
while(1){
   printf("1.insert at begining\n");
   printf("2. display\n");
   printf("3. print number of nodes\n");
   printf("4.insert at end\n"); printf("5.insert
   at position\n");
        printf("6. delete at the
   begining\n"); printf("7. delete at the
   end\n"); printf("8. delete at
   pos\n");printf("9.
   search\n");printf("default. exit\n");
   printf("enter the choice:");
   scanf("%d",&choice);switch(choice){
   case1:insert_begin(); break;
    case2:display();
          break;
    case 3: printf("no of nodes=%d\n",count); break;
    case4:insert_end(); break;
    case5:insert_position();
          break;
    case6:delete_begin();
          break;
    case7:delete_end(); break;
    case8:delete_pos(); break;
    case9:search();
           break;
    default: exit(0);
   }

}

return0;
}
```

f.ProgramtoperformoperationsonDLL

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

typedefstructnode{ int
   data;
   structnode*next;
   structnode*prev;
}node;

node* head;
intcount=0;

void insert_begin();
void insert_end(); void
insert_rapo();
voiddelete_begin();
void delete_end(); void
delete_rapo(); void
search();
void display();
 void displayRev();
int main()
```

```
{
    charchoice;
    while(1){
        printf("A. Insert at Beginning\n");
        printf("B. Insert at End\n");
        printf("C. Insert at random position\n");
        printf("D. Delete at Beginning\n");
        printf("E. Delete at End\n");
        printf("F. Delete at random position\n");
        printf("G. Search for an element\n");
        printf("X. Display list in reverse\n");
        printf("Y. Display\n");
        printf("Z. Print number of nodes\n");
        printf("Others to exit\n");
        printf("Enter the choice: ");
        scanf("%c",&choice);
        switch(choice)
        {
            case'A':insert_begin();
                break;
            case'B':insert_end();
                break;
            case'C':insert_rapo();
```

```
            break;
        case'D':delete_begin();
            break;
        case'E':delete_end();
            break;
        case 'F':delete_rapo();
            break;
        case'G':search();
            break;
        case'X':displayRev();
            break;
        case'Y':display();
            break;
        case 'Z': printf("Number of nodes are %d\n",count);
            break;
        default:exit(0);
      }
   }
   return 0;
}


voidinsert_begin()
{
   int value;
```

```
    printf("Enter value to be inserted: ");
    scanf("%d",&value);
    node*temp;
    temp = (node*)malloc(sizeof(node*));
    if(temp == NULL)
       printf("memory insufficient\n");
    else
    {
       temp->data=  value;
       temp->prev=NULL;
       if(head==NULL)
          temp->next=NULL;
       else
       {
          temp->next=head;
          head->prev=temp;
       }
       head=temp;
       count++;
    }


}
voidinsert_end()
{
```

```
int value;
printf("Enter value to be inserted: ");
scanf("%d",&value);
node*temp;
temp = (node*)malloc(sizeof(node));
if(temp == NULL)
   printf("memory insufficient\n");
else
{
   temp->data=value;
   temp->next=NULL;
   temp->prev=NULL;
   if(head==NULL)
      head=temp;
   else
   {
      node* p;
      p=head;
      while(p->next!=NULL)
         p = p->next;
      p->next=temp;
      temp->prev=p;
   }
   count++;
}
```

```
}
voidinsert_rapo()
{
   int value,pos;
   printf("Enter value to be inserted: ");
   scanf("%d",&value);
   printf("Enter the position:");
   scanf("%d",&pos);
   node*temp;
   temp = (node*)malloc(sizeof(node));
   if(temp == NULL)
      printf("memory insufficient\n");
   else
   {
      temp->data=value;
      temp->next=NULL;
      temp->prev=NULL;
      if(head==NULL)
         head=temp;
      else
      {
         node* p;
         p=head;
```

```
            for(inti=0;i<(pos-2);i++)
                p = p->next;
            temp->next=p->next;
            p->next = temp;
            temp->prev = p;
            if(temp->next!=NULL)
                temp->next->prev=temp;
        }
        count++;
    }
}


voiddelete_begin()
{
    if(head == NULL)
        printf("List isempty\n");
    else
    {
        node* p;
        p= head;
        head=head->next;
        free(p);
        count--;
        printf("Firstnode isdeletedsuccessfuly\n");
```

```
    }
}


voiddelete_end()
{
    if(head == NULL)
        printf("List isempty\n");
    else
    {
        node* p;
        p= head;
        while(p->next->next!=NULL)
            p = p->next;
        free(p->next);
        p->next=NULL;
        printf("Endingnode isdeleted\n");
        count--;
    }
}


voiddelete_rapo()
{
    intpos;
```

```
    printf("Enter the position:");
    scanf("%d",&pos);
    node*temp;
    temp = (node*)malloc(sizeof(node));
    if(temp == NULL)
        printf("memory insufficient\n");
    else
    {
        node*t,*q; t
        = head;
        for(int i=0;i<(pos-2);i++) t
            = t->next;
        q=t->next;
        t->next= t->next->next;
        if(t->next != NULL)
            t->next->prev=t;
        free(q);
        count--;
    }
}

voidsearch()
{
    node *t;
    int c = 0, p = -1, value;
```

```
    printf("Enterthevalue:");
    scanf("%d", &value);
    for(t=head;1;t=t->next)
    {
        c++;
        if(t->data==value)
        {
            printf("%d is in the list at %d position\n", t->data, c);
            p = 1;
            break;
        }
        if(t->next==NULL)
            break;
    }
    if(p==-1)
        printf("Elementisnotinthelist\n");
}


voiddisplayRev()
{
    if(head == NULL)
        printf("List isempty\n");
```

```
    else {
        node* p;
        p= head;
        while(p->next!=NULL)
            p = p->next;
        while(p->prev  !=  NULL){
            printf("%d",p->data);
            p = p->prev;
        }
        printf("%d\n",p->data);
    }
}
voiddisplay()
{
    if(head == NULL)
        printf("List isempty\n");
    else {
        node* p;
        p= head;
        while(p->next  != NULL) {
            printf("%d",p->data);
            p = p->next;
        }
```

```
        printf("%d\n",p->data);

    }

}
```

i.Program to perform operations on CLL

SourceCode:

```
/*OperationsonCircularLinkedList*/

#include<stdio.h>

#include<stdlib.h>

typedefstructnode{

int data;

structnode*next;
}node;node*

head;

intcount=0;

void insert_begin();

void insert_end();

voiddelete_begin();

void delete_end();

void sort();

voidremove_duplicates();

void display();

intmain()

 {

charchoice;

while(1){

printf("A.InsertatBeginning\n");
```

```
printf("B. Insert at End\n");
printf("C.DeleteatBeginning\n");
printf("D. Delete at End\n"); printf("E.
Sort the list\n"); printf("F. Delete
duplicates\n"); printf("Y. Display\n");
printf("Z.Printnumberofnodes\n");
printf("Others to exit\n"); printf("Enter
the choice: ");
scanf("%c",&choice);
switch(choice)
 {
case'A':insert_begin(); break;
case'B':insert_end(); break;
case'C':delete_begin();
break;
case'D':delete_end(); break;
case'E':sort(); break;
```

```
case'F':remove_duplicates(); break;

case'Y':display(); break;

case'Z':printf("Numberofnodesare%d\n",count); break;

default:exit(0);

 }

 }

return0;

 }

voidinsert_begin(){ int

value;

printf("Entervaluetobeinserted:");

scanf("%d", &value);

node*temp;
temp=(node*)malloc(sizeof(node));

if(temp == NULL) {

printf("Memoryinsufficient\n");
}else{
temp->data=value;

if(head == NULL) {
```

```
head=temp;
temp->next=head;
}else{

node* last = head;

while(last->next!=head){ last

= last->next;

 }

temp->next=head;

last->next = temp;

head = temp;

 }

count++;

 }

 }

voidinsert_end(){ int

value;

printf("Entervaluetobeinserted:");

scanf("%d", &value);

node*temp;
temp=(node*)malloc(sizeof(node));

if(temp == NULL) {

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

```
}else{
temp->data=value;

if(head == NULL) { head

= temp;

temp->next=head;

}else{
node* last = head;

while(last->next!=head){ last

= last->next;

 }

last->next = temp;

temp->next=head;

 }

count++;

 }

 }

void delete_begin() { if(head

== NULL) {

printf("Listisempty\n");

}else{
node* temp = head;

if(head->next==head){
```

```
head=NULL;
}else{
node* last = head;

while(last->next!=head){ last

= last->next;

 }

head=head->next;

last->next = head;

 }

free(temp);

count--;

printf("Firstnodeisdeletedsuccessfully\n");

 }

 }

void delete_end() { if(head

== NULL) {

printf("Listisempty\n");

}else{
node* temp = head;

if(head->next==head){

head = NULL;

}else{
```

```
node* prev = NULL;

while(temp->next!=head){

prev = temp;

temp=temp->next;

 }

prev->next=head;

free(temp);

 }

count--;
printf("Endingnodeisdeleted\n");

 }

 }

void sort() {

if(head==NULL){

printf("Listisempty\n");
}else{
node*current=head,*index=NULL; int

temp;

if(head->next!=head){ do {

index=current->next;

while(index != head) {
```

```
if(current->data>index->data){ temp =

current->data;

current->data=index->data;

index->data = temp;

 }

index=index->next;

 }

current=current->next;
}while(current->next!=head);

 }

 }

 }

voidremove_duplicates(){

if(head == NULL) { printf("List

is empty\n");

}else{
node*current=head,*next_next;

if(head->next != head) {

do{
if(current->data==current->next->data){

next_next = current->next->next;

free(current->next);
```

```
current->next=next_next;
count--;//decrementcountwhenaduplicatenodeisremoved
}else{

current=current->next;

 }

}while(current->next!=head);

 }

 }

 }

voiddisplay()

 {

if(head == NULL)

printf("Listisempty\n"); else

 {

node* p;

p=head;

while(p->next!=head)

 {

printf("%d",p->data); p

= p->next;

 }
```

```
printf("%d\n",p->data);
 }
 }
```

TestCase-1:CheckiftheLISTisempty Input:

A. InsertatBeginning

B. Insertat End

C. DeleteatBeginning

D. DeleteatEnd

E. Sortthe list

F. Deleteduplicates

Y.Display
Z.Printnumberofnodes

Others to exit

Enterthechoice:Y

Output:

Listis empty

Experiment 5: STACKS

c. Program to perform operations on stacks using arrays

```
#include<stdio.h>
#include<stdlib.h>
int size,top=-1;
int *a;

void push(){
int data;
printf("enter the data to insert into the stack");
scanf("%d",&data);
 if(top==size-1)
     printf("stack is full\n");
else
     a[++top]=data;
}

void pop(){
if(top==-1)
   printf("stack is empty\n");
else{
     printf("%d data is removed from stack",a[top]);
     top--;
```

```
}
}

void peek(){
if(top==-1)
   printf("stack is empty\n");
else{
     printf("%d is top element in the stack",a[top]);
}
}

void isEmpty(){
if(top==-1)
   printf("stack is empty\n");
else
     printf("stack is not empty\n");
}

 void isFull(){
     if(top==size-1)
         printf("stack is full\n");
else
```

```
    printf("stack is not full\n");
}

void display(){
int i;
if(top==-1)
   printf("stack is empty\n");
else{
 for(i=0;i<=top;i++)
    printf("%d\n",a[i]);
}
}

void count(){
printf("Number of elements in the stack = %d",top+1);
}



void main(){
int choice;
printf("enter the size of stack");
scanf("%d",&size);
```

```
a=(int*)malloc(sizeof(int)*size);
while(1){
printf("\n1. push\n");
printf("2. pop\n");
printf("3. peek\n"); printf("4.
isEmpty\n"); printf("5.
isFull\n"); printf("6.
Display\n"); printf("7.
count\n"); printf("enter your
choice");scanf("%d",&choice);
switch(choice){
  case 1: push();
          break;
case 2: pop();
        break; case
3: peek();
          break;
  case 4: isEmpty();
          break;
  case 5: isFull();
          break;
```

```
  case 6: display();
          break;
  case 7: count();
          break;
default: exit(0);
}
}
}
```

e. Program to perform operations on stacks using linked list

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


 void push(){
     int value;
     struct node *temp;
     temp = (struct node*)malloc(sizeof(struct node));
     printf("enter the data to insert in the stack");
     scanf("%d",&value);
    temp->data = value;
if(top==NULL)
    temp->next= NULL;
else
     temp->next=top;
```

```
    top=temp;
    count++;
}
void pop(){
if(top==NULL)
   printf("stack is empty\n");
else if(top->next!=NULL)
{
    struct node *p;
    p=top;
    printf("%d is removed from stack\n",top->data);
    top=top->next;
    free(p);
    count--;
}
else{
    struct node *p;
    p=top;
    printf("%d is removed from stack\n",top->data);
    top=NULL;
    free(p);
    count--;
```

```
}
}

void peek(){
if(top==NULL)
   printf("stack is empty\n");
else
     printf("%d is top element in the stack",top->data);
}

void isEmpty(){
if(top==NULL)
   printf("stack is empty");
else
     printf("stack is not empty");
}

void display(){
if(top==NULL)
   printf("stack is empty");
else{
       struct node *p;
```

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

void total(){
if(top==NULL)
    printf("stack is empty");
else
     printf("count=%d",count);
}

void main(){
int choice;
while(1){
printf("\n1. push\n");
printf("2. pop\n");
printf("3. peek\n");
```

```
printf("4. isEmpty\n");
printf("6. Display\n");
printf("7. count\n");
printf("enter your choice");
scanf("%d",&choice);
switch(choice){
 case 1: push();
        break;
case 2: pop();
       break; case
3: peek();
        break;
 case 4: isEmpty();
        break;
 case 6: display();
        break;
 case 7: total();
        break;
default: exit(0);
}
}
}
```

Experiment 7: QUEUES

A. Queues

Description:

Queue is a linear data structure that inserts the element at one end known as rear and deletes from another end known as front. Itfollows the property First In First Out (FIFO).

Advantages:

- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
- Queues are useful when a particular service is used by multiple consumers.
- Queues are fast in speed for data inter-process communication.
- Queues can be used in the implementation of other data structures.

Disadvantages:

- The operations such as insertion and deletion of elements from the middle are time consuming.
- Limited Space.
- In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
- Searching an element takes O(N) time.
- Maximum size of a queue must be defined prior.

Applications:

- **Multi programming:** Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
- **Network:** In a network, a queue is used in devices such as a router or a switch. another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.
- **Job Scheduling:** The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
- **Shared resources:** Queues are used as waiting lists for a single shared resource.

B. Operations on Queues using arrays

1. *Insert*

Algorithm:

- ❖ Check the queue is full or not.
- ❖ If Full , print overflow and exit
- ❖ If queue is not full, increment tail and add the element.

2. *Delete*

Algorithm:

- ❖ Check if queue is empty or not.
- ❖ If empty, print underflow and exit.
- ❖ If not empty,print element at the head and increment head.
- ❖ If, the value of front is -1 or value of front is greater than rear,write an underflow message and exit
- ❖ Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time

3. *Display*

Algorithm:

- ❖ Initialize a looping variable I to front and traverse through the queue

  To print elements until it becomes rear

4. *Front*

Algorithm:

- ❖ Check if queue is empty or not
- ❖ If queue is empty ,print queue is empty
- ❖ Otherwise, access the front element by queue[front]

5. *Rear*

Algorithm:

- ❖ Check if queue is empty or not
- ❖ If queue is empty ,print queue is empty
- ❖ Otherwise, access the last element by queue[rear]

6. *IsEmpty*

Algorithm:

- ❖ If front is equal to -1 or front>rear queue is empty
- ❖ Otherwise queue is not empty

7. *isFull*

Algorithm:

- ❖ If rear is equal to size-1 then queue is full
- ❖ Otherwise queue is not full

Operation	*Best Time Complexity*	*Average Time Complexity*	*Worst Time Complexity*
Insert	*O(1)*	*O(1)*	*O(1)*
Delete	*O(1)*	*O(1)*	*O(1)*
Display	*O(n)*	*O(n)*	*O(n)*
Front	*O(1)*	*O(1)*	*O(1)*
Rear	*O(1)*	*O(1)*	*O(1)*
IsEmpty	O(1)	O(1)	*O(1)*
isFull	O(1)	O(1)	*O(1)*

C. Program to perform operations on queues using arrays

```
#include<stdio.h>
#define size 10
int front=-1,rear=-1;
int queue[size];
void enqueue()
{
   int item;
   printf("Enter the element to be inserted:");
   scanf("%d",&item);
   if(front==-1)
    front=0;
   else if(rear==size-1)
   {
      printf("Queue is full\n");
      return;
   }
   queue[++rear]=item;
}
void dequeue()
{
   if(front==-1 || front>rear)
    printf("Queue is empty\n");
    else
    {
       printf("Removed element is %d",queue[front]);
       front++;
    }
}
```

```
void Front()
{
   if(front==-1 || front>rear)
      printf("Queue is empty\n");
      else
       printf("Element at front is %d\n",queue[front]);
}
void Rear()
{
   if(rear==-1 || front>rear)
    printf("Queue is empty\n");
      else
       printf("Element at rear is %d\n",queue[rear]);
}
void isFull()
{
   if(rear==size-1)
     printf("Queue is full\n");
   else
     printf("Queue is not full\n");
}
void isEmpty()
{
    if(rear==-1 || front>rear)
    printf("Queue is empty\n");
     else
      printf("Queue is not empty\n");
}
```

```
void display()
{
   int i;
 if(rear==-1 || front>rear)
    printf("Queue is empty\n");
    else
    {
       for(i=front;i<=rear;i++)
          printf("%d ",queue[i]);
    printf("\n");
    }
}
int main()
{
   int choice;
   while(1)
   {
      printf("1.Enqueue\n");
      printf("2.Dequeue\n");
      printf("3.Front\n");
      printf("4.Rear\n");
      printf("5.isFull\n");
      printf("6.isEmpty\n");
      printf("7.Display\n");
      printf("8.Exit\n");
      printf("Enter your choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:enqueue();
               break;
         case 2:dequeue();
         break;
```

```
        case 3:Front();
         break;
         case 4:Rear();
         break;
         case 5:isFull();
         break;
         case 6:isEmpty();
         break;
         case 7:display();
         break;
         case 8:return 0;
         default: printf("Entered choice is not in the given range\n");
      }
   }
}
```

Output:

```
1.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull
6.isEmpty
7.Display
8.Exit
Enter your choice:1
Enter the element to be inserted:10
1.Enqueue
2.Dequeue
3.Front
4.Rear
```

5.isFull
6.isEmpty
7.Display
8.Exit
Enter your choice:1
Enter the element to be inserted:20
1.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull
6.isEmpty
7.Display
8.Exit
Enter your choice:2
Removed element is 101.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull
6.isEmpty
7.Display
8.Exit
Enter your choice:3
Element at front is 20
1.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull

6.isEmpty
7.Display
8.Exit
Enter your choice:4
Element at rear is 20
1.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull
6.isEmpty
7.Display
8.Exit
Enter your choice:5
Queue is not full
1.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull
6.isEmpty
7.Display
8.Exit
Enter your choice:6
Queue is not empty
1.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull

6.isEmpty
7.Display
8.Exit
Enter your choice:7
20
1.Enqueue
2.Dequeue
3.Front
4.Rear
5.isFull
6.isEmpty
7.Display
8.Exit
Enter your choice:8

D. Operations on QUEUES using Linked List

1. *Insert*

Algorithm:

- ❖ Check the queue is full or not.
- ❖ If Full , print overflow and exit
- ❖ If queue is not full, increment tail and add the element.

2. *Delete*

Algorithm:

- ❖ Check if queue is empty or not.
- ❖ If empty, print underflow and exit.
- ❖ If not empty,print element at the head and increment head.
- ❖ If, the value of front is -1 or value of front is greater than rear,write an underflow message and exit
- ❖ Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time

3. *Display*

Algorithm:

- ❖ Initialize a looping variable I to front and traverse through the queue
  To print elements until it becomes rear

4. *Front*

Algorithm:

- ❖ Check if queue is empty or not
- ❖ If queue is empty ,print queue is empty
- ❖ Otherwise, access the front element by queue[front]

5. *Rear*

Algorithm:

- ❖ Check if queue is empty or not
- ❖ If queue is empty ,print queue is empty
- ❖ Otherwise, access the last element by rear->data

6. *IsEmpty*

Algorithm:

- ❖ If count is equal to NULL or queue is empty
- ❖ Otherwise queue is not empty

7. *isFull*

Algorithm:

- ❖ If count is equal to size-1 then queue is full
- ❖ Otherwise queue is not full

Operation	*Best Time Complexity*	*Average Time Complexity*	*Worst Time Complexity*
Insert	*O(1)*	*O(1)*	*O(1)*
Delete	*O(1)*	*O(1)*	*O(1)*
Display	*O(n)*	*O(n)*	*O(n)*
Front	*O(1)*	*O(1)*	*O(1)*
Rear	*O(1)*	*O(1)*	*O(1)*
IsEmpty	O(1)	O(1)	*O(1)*
isFull	O(1)	O(1)	*O(1)*

E. Program to perform operations on Queues using linked list

```
#include<stdio.h>
#include<stdlib.h>
int size=10;
struct node
{
   int data;
   struct node*next;
};
struct node* front=NULL;
struct node*rear=NULL;
int count=0;
void enqueue()
{
   struct node*temp;
   if(count==size)
   {
      printf("Queue is full\n");
      return;
   }
   else
   {
      int value;
      temp=(struct node*)malloc(sizeof(struct node));
      printf("Enter the data to be inserted:");
      scanf("%d",&value);
      temp->data=value;
      temp->next=NULL;
```

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

```
void Front()
{
   if(front==NULL)
     printf("Queue is empty\n");
   else
     printf("%d\n",front->data);
}
void Rear()
{
   if(rear==NULL)
      printf ("Queue is empty\n");
   else
      printf("%d",rear->data);
}
void isFull()
{
   if(count==size)
      printf("Queue is full\n");
   else
      printf("Queue is not full\n");
}
void isEmpty()
{
   if(count==0)
      printf("Queue is empty\n");
   else
      printf("Queue is not empty\n");
}
```

```
void display()
{
   if(count==0)
     printf("Queue is empty\n");
   else
   {
   struct node*t;
   t=front;
   while(t->next!=rear)
   {
      printf("%d",t->data);
   }
   printf("%d",rear->data);
   }
}
int main()
{
   int choice;
   while(1)
   {
      printf("1.Enqueue\n");
      printf("2.Dequeue\n");
      printf("3.Front\n");
      printf("4.Rear\n");
      printf("5.isFull\n");
      printf("6.isEmpty\n");
      printf("7.Display\n");
      printf("8.Exit\n");
      printf("Enter your choice:");
      scanf("%d",&choice);
```

```
 switch(choice)
     {
        case 1:enqueue();
             break;
        case 2:dequeue();
        break;
        case 3:Front();
        break;
        case 4:Rear();
        break;
        case 5:isFull();
        break;
        case 6:isEmpty();
        break;
        case 7:display();
        break;
        case 8:return 0;
        default: printf("Entered choice is not in the given
range\n");
     }
   }
}
```

Operation	*Best Time Complexity*	*Average Time Complexity*	*Worst Time Complexity*
Insert	*O(1)*	*O(1)*	*O(1)*
Delete	*O(1)*	*O(1)*	*O(1)*
Display	*O(n)*	*O(n)*	*O(n)*
Front	*O(1)*	*O(1)*	*O(1)*
Rear	*O(1)*	*O(1)*	*O(1)*
IsEmpty	O(1)	O(1)	*O(1)*
isFull	O(1)	O(1)	*O(1)*

F. Operations on Circular Queues using arrays

1.Insert

- Step 1: Check if the queue is full (Rear + 1 % Maxsize = Front)
- Step 2: If the queue is full, there will be an Overflow error
- Step 3: Check if the queue is empty, and set both Front and Rear to 0
- Step 4: If Rear = Maxsize - 1 & Front != 0 (rear pointer is at the end of the queue and front is not at 0th index), then set Rear = 0
- Step 5: Otherwise, set Rear = (Rear + 1) % Maxsize
- Step 6: Insert the element into the queue (Queue[Rear] = x)

2. Delete

- Step 1: Check if the queue is empty (Front = -1 & Rear = -1)
- Step 2: If the queue is empty, Underflow error
- Step 3: Set Element = Queue[Front]
- Step 4: If there is only one element in a queue, set both Front and Rear to -1 (IF Front = Rear, set Front = Rear = -1)
- Step 5: And if Front = Maxsize -1 set Front = 0
- Step 6: Otherwise, set Front = Front + 1

3.Front

```
if (front == -1)
   print Queue is empty
else
   print  a[front]
```

4.Rear

```
if (rear == -1)
   print Queue is empty
else
   print  a[rear]
```

5. isFull

```
if ((rear + 1) % MAX_SIZE == front)
        print  queue is full
   else
        print   Queue is not full
```

6. isEmpty

```
            if (rear == -1)
             print  Queue is empty
            else
             printf   Queue is not empty
```

7.Display

```
  for (i = 0; i < count; i++)
      Print  a[(front+i)%MAX_SIZE]
     printf("\n");
```

G. Program to perform operations on Circular queues using arrays

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 5
int a[MAX_SIZE];
int front = -1;
int rear = -1;

void enqueue()
{
      int item;
   printf("Enter the element to be inserted:");
   scanf("%d",&item);
if (front == -1 && rear == -1)
{
front = rear = 0;
}
else if ((rear + 1) % MAX_SIZE == front)
{
printf("queue is full\n");
return;
}
else
rear = (rear + 1) % MAX_SIZE;
a[rear] = item;
}
void dequeue()
{
if (front == -1 && rear == -1)
{
printf("queue is empty \n");
return;
}
```

```
else if (front == rear)
{
front = rear = -1;
}
else
front = (front + 1) % MAX_SIZE;
}

void Front()

{

if (front == -1)
    printf("Queue is empty\n");
else
    printf("%d", a[front]);

}
void Rear()
{
   if (rear == -1)
    printf("Queue is empty\n");
else
    printf("%d", a[rear]);
}
void display()

{

  int count = ((rear + MAX_SIZE - front) % MAX_SIZE)+1;
  int i;
  for (i = 0; i < count; i++)
  {
   printf("%d ", a[(front+i)%MAX_SIZE]);
  }
```

```
  printf("\n");
}
void isEmpty()
{
   if (rear == -1)
    printf("Queue is empty\n");
   else
    printf("Queue is not empty");
}
void isFull()
{
   if ((rear + 1) % MAX_SIZE == front)
       printf("queue is full\n");
   else
       printf("Queue is not full\n");
}

int main()
{
   int choice;
   while(1)
   {
      printf("1.Enqueue\n");
      printf("2.Dequeue\n");
      printf("3.Front\n");
      printf("4.Rear\n");
      printf("5.isFull\n");
      printf("6.isEmpty\n");
      printf("7.Display\n");
      printf("8.Exit\n");
      printf("Enter your choice:");
      scanf("%d",&choice);
```

```
 switch(choice)
     {
        case 1:enqueue();
             break;
        case 2:dequeue();
        break;
        case 3:Front();
        break;
        case 4:Rear();
        break;
        case 5:isFull();
        break;
        case 6:isEmpty();
        break;
        case 7:display();
        break;
        case 8:return 0;
        default: printf("Entered choice is not in the given
range\n");
     }
   }
}
```

Experiment 8: APPLICATIONS OF QUEUES AND STACKS

Problem 1:

Problem

You are given an array A of Q integers and Q queries. In each query, you are given an integer i $(1 \leq i \leq N)$.

Your task is to find the minimum index greater than i $(1 \leq i \leq N)$ such that:

1. Sum of digits of A_i is greater than the sum of digits of A_j
2. $A_i < A_j$

If there is no answer, then print **-1**.

Input format

- The first line contains two numbers N and Q.
- The next line contains N numbers.
- Next Q lines contain Q queries.

Output format

Print the answer as described in the problem

Constraints

$(1 \leq N,\ Q \leq 10^5)$
$(1 \leq A_i \leq 10^9)$
$(1 \leq Q_i \leq N)$

```
#include<stdio.h>

int main()

{

   int n,q;

   printf("Enter n,q:");
```

```
scanf("%d %d",&n,&q);
   int a[q];
   int Q[q];
   printf("Enter the elements of a of size %d:",n);
   for(int i=0;i<n;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("Enter the elements of q of size %d",q);
   for(int i=0;i<q;i++)
   {
      scanf("%d",&Q[i]);
   }
   for(int i=0;i<q;i++)
   {
       int k=Q[i]-1;

     int j,flag=0;
     for(j=k+1;j<q;j++)
     {
         int sum1=0,sum2=0;
```

```
    if(a[k]>=a[j])
       continue;
    else
    {
       int p=a[k],q=a[j];
     while(p>0)
     {
      sum1=sum1+p%10;
      p/=10;
     }
     while(q>0)
     {
        sum2=sum2+q%10;
        q/=10;
     }
     if(sum1>sum2)
     {
        flag=1;
        printf("%d\n",j+1);
        break;
     }
   }
}
```

```
if(flag==0) printf("-1\n");


   }
   return 0;
}
```

Output:

```
Enter n,q:5 5
Enter the elements of a of size 5:62 70 28 62 92
Enter the elements of q of size 51 5 3 4 2
2
-1
4
-1
-1
```

Problem 2:

Problem

You are given a stack of **N** integers such that the first element represents the top of the stack and the last element represents the bottom of the stack. You need to pop at least one element from the stack. At any one moment, you can convert stack into a queue. The bottom of the stack represents the front of the queue. You cannot convert the queue back into a stack. Your task is to remove exactly **K** elements such that the sum of the **K** removed elements is maximised.

Input format :

- The first line consists of two space-separated integers **N** and **K**.
- The second line consists of **N** space-separated integers denoting the elements of the stack.

Output format :

- Print the maximum possible sum of the **K** removed elements

Constraints :

- $1 \leq N \leq 10^5$
- $1 \leq K \leq N$
- $1 \leq A_i \leq 10^9$

Sample Input	Sample Output
10 5 10 9 1 2 3 4 5 6 7 8	40

```
#include <stdio.h>

   int main() {

   int n,k;

   int temp;

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

    int arr[n+1];
```

```
    for(int i=0;i<n;i++)
       scanf("%d",&arr[i]);
    int prefix[n+1];
    prefix[0]=arr[0];
    for(int i=1;i<n;i++)
        prefix[i]=prefix[i-1]+arr[i];
    int ans=0;
    for(int i=0;i<k;i++)
    {
   temp=prefix[i]+prefix[n-1]-prefix[n-k+i];
      ans=(ans>temp)?ans:temp;
    }
    printf("%d",ans);
        return 0;
   }
```

Output:

```
10 5
10 9 1 2 3 4 5 6 7 8
**************************
40
```

Problem 3:

Problem

Your task is to construct a tower in N days by following these conditions:

- Every day you are provided with one disk of distinct size.
- The disk with larger sizes should be placed at the bottom of the tower.
- The disk with smaller sizes should be placed at the top of the tower.

The order in which tower must be constructed is as follows:

- You cannot put a new disk on the top of the tower until all the larger disks that are given to you get placed.

Print N lines denoting the disk sizes that can be put on the tower on the i^{th} day.

Input format

- First line: N denoting the total number of disks that are given to you in the N subsequent days
- Second line: N integers in which the i^{th} integers denote the size of the disks that are given to you on the i^{th} day

Note: All the disk sizes are distinct integers in the range of $1\ to\ N$.

Output format

Print N lines. In the i^{th} line, print the size of disks that can be placed on the top of the tower in descending order of the disk sizes.

If on the i^{th} day no disks can be placed, then leave that line empty.

```
#include<stdio.h>

int main()

{

int disk, temp[100001] = {0};

scanf("%d", &disk);

int min = disk, size = disk;

int q;

for (int i = 0; i < disk; i++)

{
```

```
scanf("%d", &q);
temp[q] = q;
if(q == min)
{
while(temp[size])
{
printf("%d ", size);
size--;
}
min = size;
printf("\n");
}
}
}
```

Output:

```
7
1 4 7 6 3 2 5
*********
7
6
5 4 3 2 1
```

Problem 4:

Problem

You are given two arrays each of size n, a and b consisting of the first n positive integers each exactly once, that is, they are permutations.

Your task is to find the minimum time required to make both the arrays empty. The following two types of operations can be performed any number of times each taking 1 second:

- In the first operation, you are allowed to rotate the first array clockwise.
- In the second operation, when the first element of both the arrays is the same, they are removed from both the arrays and the process continues.

Input format

- The first line contains an integer n, denoting the size of the array.
- The second line contains the elements of array a.
- The third line contains the elements of array b.

Output format

Print the total time taken required to empty both the array.

Constraints

$1 \leq n \leq 100$

Sample Input	Sample Output
3 1 3 2 2 3 1	6

Time Limit: 1

```
#include <stdio.h>

 int main(){

   int num;

   scanf("%d", &num);

   int *arr1,*arr2;

   arr1=(int*)malloc(sizeof(int)*num);

   arr2=(int*)malloc(sizeof(int)*num);
```

```
   for(int i=0;i<num;i++)
    scanf("%d",&arr1[i]);
    for(int i=0;i<num;i++)
    scanf("%d",&arr2[i]);
    int time=0;
    int start=0;
    for(int i=0;i<num;i++)
    {
       while(arr1[start]!=arr2[i])
       {
          if(arr1[start]!=0)
          time++ ;


          start=(start+1)%num;
       }
       arr1[start]=0;
       time++;
    }
    printf("%d",time);
}
```

Output:

3

1 3 2

2 3 1

****

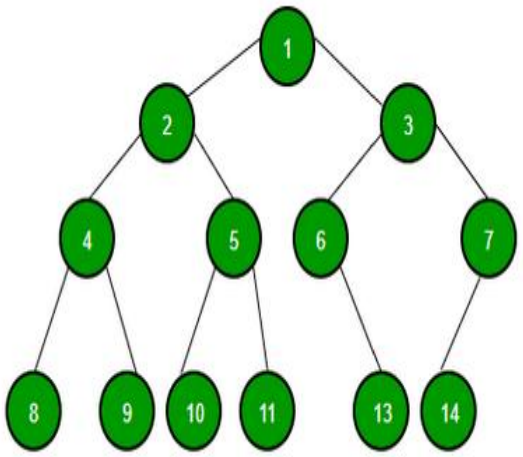
6

Experiment 9: Binary Trees

Description:

Binary Tree is defined as a tree data structure where each node has at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Example:



Applications:

- Searching and Sorting Algorithms
- Expression Trees
- Huffman Coding
- Decision Trees
- File Systems. Binary trees are used in file systems to organise and store files

A. Implement Binary trees using arrays

Source Code:

```
/*
implement a binary tree using arrays
perform tree traversals
print leaf nodes
print height of tree
*/

#include<stdio.h>
int complete_node = 15;
int tree[15]={15,16,10,45,20,'\0',24,34,54,64,74};
int max,min,sum;
int print_tree() {
  printf("\n");
  for (int i = 0; i < 15; i++) {
    if (tree[i] != '\0')
      printf(" %d ",tree[i]);
    else
      printf(" - ");
  }
  return 0;
}
```

```
int get_right_child(int index)
{
   if(tree[index]!='\0' && ((2*index)+2)<complete_node)
      return (2*index)+2;
   return -1;
}

int get_left_child(int index)
{
   if(tree[index]!='\0' && (2*index)+1<complete_node)
      return (2*index)+1;
   return -1;
}

void preorder(int index)
{
   if(index>=0 && tree[index]!='\0')
   {
      printf(" %d ",tree[index]);
      preorder(get_left_child(index));
      preorder(get_right_child(index));
   }
}
void postorder(int index)
{
   if(index>=0 && tree[index]!='\0')
   {
      postorder(get_left_child(index));
      postorder(get_right_child(index));
      printf(" %d ",tree[index]);
```

```
    }
}

void inorder(int index)
{
    if(index>=0 && tree[index]!='\0')
    {
        inorder(get_left_child(index));
        printf(" %d ",tree[index]);
        {
            if(max<tree[index])
                max=tree[index];
            if(min>tree[index])
                min=tree[index];
            sum=sum+tree[index];
        }
        inorder(get_right_child(index));
    }
}
void levelorder()
{
    int j;
    for(j=0;j<complete_node;j++)
    {
        if(tree[j]!='\0')
            printf(" %d ",tree[j]);
    }
}
```

```
int is_leaf(int index)
{
   if(!get_left_child(index) && !get_right_child(index)){
      return 1;
   }
   if(tree[get_left_child(index)]=='\0' &&
tree[get_right_child(index)]=='\0'){
      return 1;
   }
   return 0;
}

int get_max(int a, int b)
{
   return (a>b) ? a : b;
}

int get_height(int index)
{
   if(tree[index]=='\0' || index<0 || is_leaf(index))
      return 0;
   return(get_max(get_height(get_left_child(index)),
get_height(get_right_child(index)))+1);
}
```

```
int search(int x,int index)
{
        int flag=0;
        if(x==tree[index])
        {
         return 1;
         flag=1;
         }
        if(get_left_child(index)!=NULL)
        search(x,get_left_child(index));
        if(get_right_child(index)!=NULL)
        search(x,get_right_child(index));
        if(flag==0) return 0;
}
int main()
{
   print_tree();
   max=tree[0];
   min=tree[0];
   sum=0;
   printf("\nPreorder:\n");
   preorder(0);
   printf("\nPostorder:\n");
   postorder(0);
   printf("\nInorder:\n");
   inorder(0);
   printf("\n max=%d",max);
   printf("\n min=%d",min);
   printf("\n sum=%d",sum);
   printf("\nLevelorder:\n");
```

```
  levelorder();
   printf("\n");
   printf("\nheight of the tree is:%d\n",get_height(0));
   for(int i=0;i<complete_node;i++)
   {
    if(tree[i]!='\0'){
     if(is_leaf(i))
       printf("\n%d is a leaf node",tree[i]);
    }
   }
   int x;
   printf("\nEnter the element to be searched:");
   scanf("%d",&x);
   if(search(x,0)==1) printf("Element found");
   else printf("Element not found");
   return 0;
}
```

B. Implement binary tree using Linked list

Source Code:

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

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

struct node *root = NULL;

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

   return newNode;
}

struct queue {
   int front, rear, size;
   struct node* *arr;
};
```

```
struct queue* createQueue() {
   struct queue* newQueue = (struct
queue*)malloc(sizeof(struct queue));

   newQueue->front = -1;
   newQueue->rear = 0;
   newQueue->size = 0;

   newQueue->arr = (struct node**)malloc(100 * sizeof(struct
node*));

   return newQueue;
}

void enqueue(struct queue* queue, struct node *temp) {
   queue->arr[queue->rear++] = temp;
   queue->size++;
}

struct node *dequeue(struct queue* queue) {
   queue->size--;
   return queue->arr[++queue->front];
}

void insertNode(int data) {
   struct node *newNode = createNode(data);

   if(root == NULL){
      root = newNode;
      return;
   }
```

```
    else {
        struct queue* queue = createQueue();
        enqueue(queue, root);

        while(true) {
            struct node *node = dequeue(queue);

            if(node->left != NULL && node->right != NULL) {
                enqueue(queue, node->left);
                enqueue(queue, node->right);
            }
            else {
                if(node->left == NULL) {
                    node->left = newNode;
                    enqueue(queue, node->left);
                }
                else {
                    node->right = newNode;
                    enqueue(queue, node->right);
                }
                break;
            }
        }
    }
}

void inorderTraversal(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return;
    }
```

```
  else {
      if(node->left != NULL)
         inorderTraversal(node->left);
      printf("%d ", node->data);
      if(node->right != NULL)
         inorderTraversal(node->right);
   }
}

void preorderTraversal(struct node *node) {
   if(root == NULL){
      printf("Tree is empty\n");
      return;
   }
   else {
      printf("%d ", node->data);
      if(node->left != NULL)
         preorderTraversal(node->left);
      if(node->right != NULL)
         preorderTraversal(node->right);
   }
}

void postorderTraversal(struct node *node) {
   if(root == NULL){
      printf("Tree is empty\n");
      return;
   }
   else {
      if(node->left != NULL)
         postorderTraversal(node->left);
```

```
  if(node->right != NULL)
        postorderTraversal(node->right);
     printf("%d ", node->data);
   }
}

void levelorderTraversal(struct node *node) {
   if(root == NULL){
      printf("Tree is empty\n");
      return;
   }
   else {
      struct queue* queue = createQueue();
      enqueue(queue, root);

      while(queue->size > 0) {
         struct node *current = dequeue(queue);
         printf("%d ", current->data);

         if(current->left != NULL)
            enqueue(queue, current->left);
         if(current->right != NULL)
            enqueue(queue, current->right);
      }
   }
}
void printLeafNodes(struct node *node) {
   if(root == NULL){
      printf("Tree is empty\n");
      return;
   }
```

```
    else {
        if (node == NULL)
            return;
        if (node->left == NULL && node->right == NULL)
            printf("%d ", node->data);
        else {
            printLeafNodes(node->left);
            printLeafNodes(node->right);
        }
    }
}

int findMax(struct node *node) {
    if(root == NULL){
        printf("Tree is empty\n");
        return -1;
    }
    else {
        if(node == NULL)
            return INT_MIN;
        int max = node->data;
        int leftMax = findMax(node->left);
        int rightMax = findMax(node->right);

        if (leftMax > max)
            max = leftMax;
        if (rightMax > max)
            max = rightMax;
        return max;
    }
}
```

```
int findMin(struct node *node) {
   if(root == NULL){
      printf("Tree is empty\n");
      return -1;
   }
   else {
      if(node == NULL)
         return INT_MAX;
      int min = node->data;
      int leftMin = findMin(node->left);
      int rightMin = findMin(node->right);

      if (leftMin < min)
         min = leftMin;
      if (rightMin < min)
         min = rightMin;

      return min;
   }
}

bool searchElement(struct node *node, int target) {
   if(root == NULL){
      printf("Tree is empty\n");
      return false;
   }
   else {
      if (node == NULL)
         return false;
      if (node->data == target)
         return true;
```

```
  return searchElement(node->left, target) ||
searchElement(node->right, target);
   }
}

int getHeight(struct node *node) {
   if(root == NULL){
      printf("Tree is empty\n");
      return -1;
   }
   else {
      if(node == NULL)
         return -1;
      int leftHeight = getHeight(node->left);
      int rightHeight = getHeight(node->right);
      return 1 + (leftHeight > rightHeight ? leftHeight :
rightHeight);
   }
}

void getBreadth(struct node *node, int height, int *breadth) {
   if(root == NULL){
      printf("Tree is empty\n");
      return;
   }
   else {
      if(node == NULL)
         return;

      if(height == 0)
         (*breadth)++;
```

```
 else {
        getBreadth(node->left, height - 1, breadth);
        getBreadth(node->right, height - 1, breadth);
      }
   }
}

void breadthOfEachNode(struct node *node) {
   if(root == NULL){
      printf("Tree is empty\n");
      return;
   }
   else {
      int height = getHeight(node);
      for (int i = 0; i <= height; i++) {
         int breadth = 0;
         getBreadth(node, i, &breadth);
         printf("Node at height %d: %d\n", i, breadth);
      }
   }
}

struct node* deleteNode(struct node* root, int key) {
   if(root == NULL)
      return root;

   if(key < root->data)
      root->left = deleteNode(root->left, key);
   else if(key > root->data)
      root->right = deleteNode(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 && temp->left != NULL)
         temp = temp->left;

      root->data = temp->data;
      root->right = deleteNode(root->right, temp->data);
   }
   return root;
}

int main(){
   int choice, value;

   printf("\n----- Binary Tree -----\n");
   while(1){
      printf("\n***** MENU *****\n");
      printf("1. Insert\n2. Display Inorder\n3. Display
Preorder\n4. Display Postorder\n");
```

```
 printf("5. Display Levelorder\n6. Print Leaf Nodes\n7. Find
Max\n8. Find Min\n");
 printf("9. Search Element\n10. Print Height\n11. Print
Breadth of Each Node\n");
 printf("12. Delete Node\n13. Exit\n");
 printf("Enter your choice: ");
 scanf("%d",&choice);

      switch(choice){
         case 1:
            printf("\nEnter the value to be insert: ");
            scanf("%d", &value);
            insertNode(value);
            break;
         case 2:
            printf("\nInorder Traversal: ");
            inorderTraversal(root);
            break;
         case 3:
            printf("\nPreorder Traversal: ");
            preorderTraversal(root);
            break;
         case 4:
            printf("\nPostorder Traversal: ");
            postorderTraversal(root);
            break;
         case 5:
            printf("\nLevelorder Traversal: ");
            levelorderTraversal(root);
            break;
```

```
case 6:
        printf("\nLeaf Nodes: ");
        printLeafNodes(root);
        break;
    case 7:
        printf("\nMax Element: %d", findMax(root));
        break;
    case 8:
        printf("\nMin Element: %d", findMin(root));
        break;
    case 9:
        printf("\nEnter element to be searched: ");
        scanf("%d", &value);
        if (searchElement(root, value))
            printf("Element found in the tree.\n");
        else
            printf("Element not found in the tree.\n");
        break;
    case 10:
        printf("\nHeight of the Tree: %d", getHeight(root));
        break;
    case 11:
        breadthOfEachNode(root);
        break;
    case 12:
        printf("\nEnter the value to be deleted: ");
        scanf("%d", &value);
        root = deleteNode(root, value);
        break;
    case 13:
        exit(0);
```

```
    default:
            printf("\nPlease select correct operations!!!\n");
        }
    }

    return 0;
}
```

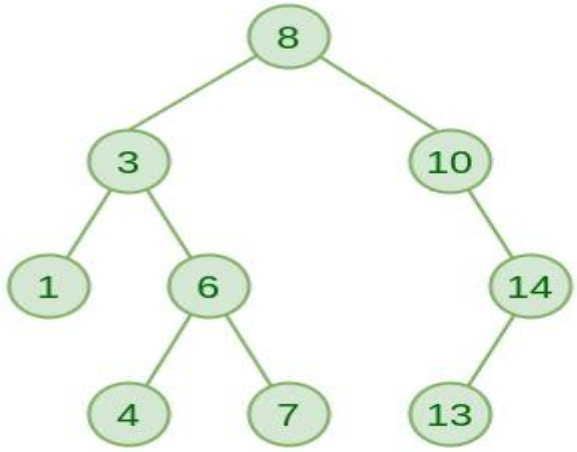
Experiment 10: Binary Search Trees

Description:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Example:



Applications:

- BSTs are used for indexing and multi-level indexing.
- They are also helpful to implement various searching algorithms.
- It is helpful in maintaining a sorted stream of data.
- TreeMap and TreeSet data structures are internally implemented using self-balancing BSTs.

A. Implement Binary Search Tree using arrays

Source Code:

```
#include <stdio.h>
#define MAX_SIZE 100
int tree[MAX_SIZE];

// Function to initialize the tree
void initializeTree() {
   for (int i = 0; i < MAX_SIZE; i++) {
      tree[i] = -1; // Assuming -1 represents an empty node
   }
}

// Function to insert a value into the tree
void insert(int value, int index) {
   if (tree[index] == -1) {
      tree[index] = value;
   } else if (value < tree[index]) {
      insert(value, 2 * index + 1); // Go to the left subtree
   } else {
      insert(value, 2 * index + 2); // Go to the right subtree
   }
}

// Function to perform inorder traversal
void inorder(int index) {
   if (tree[index] != -1) {
```

```
    inorder(2 * index + 1);
      printf("%d ", tree[index]);
      inorder(2 * index + 2);
   }
}

// Function to perform preorder traversal
void preorder(int index) {
   if (tree[index] != -1) {
      printf("%d ", tree[index]);
      preorder(2 * index + 1);
      preorder(2 * index + 2);
   }
}

// Function to perform postorder traversal
void postorder(int index) {
   if (tree[index] != -1) {
      postorder(2 * index + 1);
      postorder(2 * index + 2);
      printf("%d ", tree[index]);
   }
}

// Function to perform level order traversal
void levelorder(int height) {
   for (int i = 0; i <= height; i++) {
      printLevel(i, 0);
   }
}
```

```
// Helper function to print nodes at a given level
void printLevel(int level, int index) {
   if (index < MAX_SIZE && tree[index] != -1) {
      if (level == 0) {
         printf("%d ", tree[index]);
      } else {
         printLevel(level - 1, 2 * index + 1);
         printLevel(level - 1, 2 * index + 2);
      }
   }
}

// Function to calculate the height of the tree
int height(int index) {
   if (tree[index] == -1) {
      return -1;
   } else {
      int leftHeight = height(2 * index + 1);
      int rightHeight = height(2 * index + 2);
      return 1 + (leftHeight > rightHeight ? leftHeight :
rightHeight);
   }
}

// Function to find the minimum value in the tree
int findMin(int index) {
   while (tree[2 * index + 1] != -1) {
      index = 2 * index + 1;
   }
```

```
  return tree[index];
}

// Function to find the maximum value in the tree
int findMax(int index) {
   while (tree[2 * index + 2] != -1) {
      index = 2 * index + 2;
   }
   return tree[index];
}

// Function to search for a value in the tree
int search(int value, int index) {
   if (tree[index] == -1) {
      return 0; // Value not found
   } else if (tree[index] == value) {
      return 1; // Value found
   } else if (value < tree[index]) {
      return search(value, 2 * index + 1); // Search in the left
subtree
   } else {
      return search(value, 2 * index + 2); // Search in the
right subtree
   }
}
```

```
int main() {
    initializeTree();

    // Insert values into the tree
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++) {
        insert(values[i], 0);
    }

    // Perform traversals
    printf("Inorder: ");
    inorder(0);
    printf("\n");

    printf("Preorder: ");
    preorder(0);
    printf("\n");

    printf("Postorder: ");
    postorder(0);
    printf("\n");

    printf("Level order: ");
    int treeHeight = height(0);
    levelorder(treeHeight);
    printf("\n");

    // Display height of the tree
    printf("Height of the tree: %d\n", treeHeight);
```

```
    // Find min and max values
    printf("Minimum value: %d\n", findMin(0));
    printf("Maximum value: %d\n", findMax(0));

    // Search for a value
    int searchValue = 40;
    if (search(searchValue, 0)) {
        printf("%d found in the tree.\n", searchValue);
    } else {
        printf("%d not found in the tree.\n", searchValue);
    }

    return 0;
}
```

B. Implement Binary Search Trees using Linked list

Source Code:

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

// Node structure for the binary search tree
struct Node {
   int data;
   struct Node *left;
   struct Node *right;
};

// Function to create a new node
struct Node *createNode(int value) {
   struct Node *newNode = (struct Node
*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->left = newNode->right = NULL;
   return newNode;
}

// Function to insert a value into the tree
struct Node *insert(struct Node *root, int value) {
   if (root == NULL) {
      return createNode(value);
   }
   if (value < root->data) {
      root->left = insert(root->left, value);
   }
```

```
 else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}

// Function to find the minimum value in the tree
struct Node *findMin(struct Node *root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

// Function to delete a value from the tree
struct Node *deleteNode(struct Node *root, int value) {
    if (root == NULL) {
        return root;
    }

    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    } else {
        // Node with only one child or no child
        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;
        }

        // Node with two children, get the inorder successor
        struct Node *temp = findMin(root->right);

        // Copy the inorder successor's data to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}

// Function to perform inorder traversal
void inorder(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

```
// Function to perform preorder traversal
void preorder(struct Node *root) {
   if (root != NULL) {
      printf("%d ", root->data);
      preorder(root->left);
      preorder(root->right);
   }
}

// Function to perform postorder traversal
void postorder(struct Node *root) {
   if (root != NULL) {
      postorder(root->left);
      postorder(root->right);
      printf("%d ", root->data);
   }
}

// Function to perform level order traversal
void levelorder(struct Node *root) {
   if (root == NULL) {
      return;
   }

   // TODO: Implement level order traversal using a queue
}

// Function to calculate the height of the tree
int height(struct Node *root) {
```

```
    if (root == NULL) {
        return -1;
    } else {
        int leftHeight = height(root->left);
        int rightHeight = height(root->right);
        return 1 + (leftHeight > rightHeight ? leftHeight :
rightHeight);
    }
}

// Function to find the depth of a specific node
int depth(struct Node *root, int value, int currentDepth) {
    if (root == NULL) {
        return -1;
    } else if (value == root->data) {
        return currentDepth;
    } else if (value < root->data) {
        return depth(root->left, value, currentDepth + 1);
    } else {
        return depth(root->right, value, currentDepth + 1);
    }
}

// Function to find the maximum value in the tree
struct Node *findMax(struct Node *root) {
    while (root->right != NULL) {
        root = root->right;
    }
    return root;
}
```

```
// Function to search for a value in the tree
struct Node *search(struct Node *root, int value) {
    if (root == NULL || root->data == value) {
        return root;
    }

    if (value < root->data) {
        return search(root->left, value);
    }

    return search(root->right, value);
}

int main() {
    struct Node *root = NULL;

    // Insert values into the tree
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++) {
        root = insert(root, values[i]);
    }
    // Perform traversals
    printf("Inorder: ");
    inorder(root);
    printf("\n");

    printf("Preorder: ");
    preorder(root);
    printf("\n");
```

```
    printf("Postorder: ");
    postorder(root);
    printf("\n");

    // Display height of the tree
    printf("Height of the tree: %d\n", height(root));

    // Find min and max values
    printf("Minimum value: %d\n", findMin(root)->data);
    printf("Maximum value: %d\n", findMax(root)->data);

    // Search for a value
    int searchValue = 40;
    struct Node *searchResult = search(root, searchValue);
    if (searchResult != NULL) {
        printf("%d found in the tree.\n", searchValue);
    } else {
        printf("%d not found in the tree.\n", searchValue);
    }

    // Find height and depth of each node
    for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++) {
        int nodeDepth = depth(root, values[i], 0);
        printf("Node %d: Height=%d, Depth=%d\n", values[i],
height(search(root, values[i])), nodeDepth);
    }

    return 0;
}
```

Experiment 11: GRAPHS

A. Depth First Search(DFS)

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
int vis[100];
struct Graph {
   int V;
   int E;
   int** Adj;
};
struct Graph* adjMatrix(int vertices, int edges, int edgeList[][2])
{
   struct Graph* G = (struct Graph*)malloc(sizeof(struct Graph));
   if (!G) {
      printf("Memory Error\n");
      return NULL;
   }
   G->V = vertices;
   G->E = edges;
   G->Adj = (int*)malloc((G->V) * sizeof(int));
   for (int k = 0; k < G->V; k++) {
      G->Adj[k] = (int*)malloc((G->V) * sizeof(int));
   }
   for (int u = 0; u < G->V; u++) {
      for (int v = 0; v < G->V; v++) {
```

```
    G->Adj[u][v] = 0;
        }
    }
    for (int i = 0; i < G->E; i++) {
        int u = edgeList[i][0];
        int v = edgeList[i][1];
        G->Adj[u][v] = G->Adj[v][u] = 1;
    }
    return G;
}
void DFS(struct Graph* G, int u) {
    vis[u] = 1;
    printf("%d ", u);
    for (int v = 0; v < G->V; v++) {
        if (!vis[v] && G->Adj[u][v])
            DFS(G, v);
    }
}
void DFStraversal(struct Graph* G) {
    for (int i = 0; i < 100; i++) {
        vis[i] = 0;
    }
    for (int i = 0; i < G->V; i++) {
        if (!vis[i])
            DFS(G, i);
    }
}
```

```
int main() {
    int vertices, edges;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    int edgeList[edges][2];
    printf("Enter the edges (vertex pairs):\n");
    for (int i = 0; i < edges; i++) {
        scanf("%d %d", &edgeList[i][0], &edgeList[i][1]);
    }
    struct Graph* G = adjMatrix(vertices, edges, edgeList);
    DFStraversal(G);
    return 0;
}
```

B. Breadth First Search(BFS)

Source Code:

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#define VERTS 50
typedef struct Graph_t {
        int V;
        bool adj[VERTS][VERTS];
} Graph;
Graph* Graph_create(int V){
        Graph* g = malloc(sizeof(Graph));
        g->V = V;
        for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
                        g->adj[i][j] = false;
                }
        }
        return g;
}
void Graph_addEdge(Graph* g, int v, int w){
        g->adj[v][w] = true;
}
void Graph_BFS(Graph* g, int s){
        bool visited[VERTS];
        for (int i = 0; i < g->V; i++) {
                visited[i] = false;
        }
```

```
int queue[VERTS];
        int front = 0, rear = 0;
        visited[s] = true;
        queue[rear++] = s;

        while (front != rear) {
                s = queue[front++];
                printf("%d ", s);
                for (int adjacent = 0; adjacent < g->V;
                        adjacent++) {
                        if (g->adj[s][adjacent] && !visited[adjacent]) {
                                visited[adjacent] = true;
                                queue[rear++] = adjacent;
                        }
                }
        }
}
int main(){
        Graph* g = Graph_create(4);
        Graph_addEdge(g, 0, 2);
        Graph_addEdge(g, 0, 3);
        Graph_addEdge(g, 0, 1);
        Graph_addEdge(g, 1, 0);
        Graph_addEdge(g, 2, 0);
        Graph_addEdge(g, 2, 0);
        Graph_addEdge(g, 3, 0);
        printf("bfs start from root 3 \n");
        Graph_BFS(g, 3);
        return 0;
}
```

Experiment 12: Graphs – Minimum Cost Spanning Trees

A. Prim's Algorithm

Description:

This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

Algorithm:

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step 4: Find the minimum among these edges.
Step 5: Add the chosen edge to the MST if it does not form any cycle.
Step 6: Return the MST and exit

Source Code:

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

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

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
   // Initialize min value
   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;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d \n", parent[i], i,
            graph[i][parent[i]]);
}
```

```
// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
   // Array to store constructed MST
   int parent[V];
   // Key values used to pick minimum weight edge in cut
   int key[V];
   // To represent set of vertices included in MST
   bool mstSet[V];

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

   // Always include first 1st vertex in MST.
   // Make key 0 so that this vertex is picked as first
   // vertex.
   key[0] = 0;

   // First node is always root of MST
   parent[0] = -1;

   // The 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 value and parent index of
      // the adjacent vertices of the picked vertex.
      // Consider only those vertices which are not
      // yet included in MST
      for (int v = 0; v < V; v++)

         // graph[u][v] is non zero only for adjacent
         // vertices of m mstSet[v] is false for vertices
         // not yet included in MST Update the key only
         // if graph[u][v] is smaller than key[v]
         if (graph[u][v] && mstSet[v] == false
            && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
   }

   // print the constructed MST
   printMST(parent, graph);
}

// Driver's code
int main()
{
   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 solution
   primMST(graph);

   return 0;
}
```

B. Kruskal's Algorithm

Description:

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

Algorithm:

- Sort all the edges in non-decreasing order of their weight.
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
- Repeat step#2 until there are (V-1) edges in the spanning tree

Source Code:

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

// Comparator function to use in sorting
int comparator(const void* p1, const void* p2)
{
   const int(*x)[3] = p1;
   const int(*y)[3] = p2;

   return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
   for (int i = 0; i < n; i++) {
      parent[i] = i;
      rank[i] = 0;
   }
}

// Function to find the parent of a node
int findParent(int parent[], int component)
{
   if (parent[component] == component)
      return component;

   return parent[component]
         = findParent(parent, parent[component]);
}
```

```
// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    // Finding the parents
    u = findParent(parent, u);
    v = findParent(parent, v);

    if (rank[u] < rank[v]) {
        parent[u] = v;
    }
    else if (rank[u] > rank[v]) {
        parent[v] = u;
    }
    else {
        parent[v] = u;

        // Since the rank increases if
        // the ranks of two sets are same
        rank[u]++;
    }
}

// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
    // First we sort the edge array in ascending order
    // so that we can access minimum distances/cost
    qsort(edge, n, sizeof(edge[0]), comparator);

    int parent[n];
    int rank[n];

    // Function to initialize parent[] and rank[]
    makeSet(parent, rank, n);
```

```
  // To store the minimun cost
    int minCost = 0;

    printf(
       "Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++) {
       int v1 = findParent(parent, edge[i][0]);
       int v2 = findParent(parent, edge[i][1]);
       int wt = edge[i][2];

       // If the parents are different that
       // means they are in different sets so
       // union them
       if (v1 != v2) {
          unionSet(v1, v2, parent, rank, n);
          minCost += wt;
          printf("%d -- %d == %d\n", edge[i][0],
                edge[i][1], wt);
       }
    }

    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}
  // Driver code
int main()
{
    int edge[5][3] = { { 0, 1, 10 },
                       { 0, 2, 6 },
                       { 0, 3, 5 },
                       { 1, 3, 15 },
                       { 2, 3, 4 } };

    kruskalAlgo(5, edge);
      return 0;
}
```