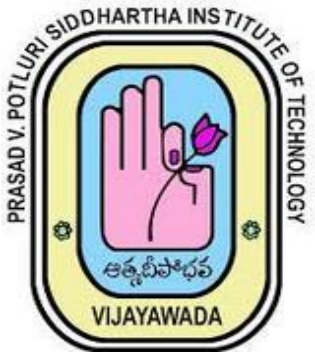
DATA STRUCTURES LAB RECORD
(20ES1356)

II B. TECH I SEM
CSE-S1

ACADEMIC YEAR: 2023-24

SUBMITTED BY
E. HARI KRISHNA
REGD. NO:22501A0545



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PRASAD V. POTLURI SIDDHARTHA INSTITUTE OF TECHNOLOGY

SIGNATURE OF STUDENT
E. Hari Krishna

SIGNATURE OF FACULTY IN-CHARGE

INDEX

S.No	Experiment No.	Topic	Date	Page No
1	2	**Searching**	11/8/2023	10-24
	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)**		
		Additional Problems		
	g	Missing-numbers - Hackerrank **(program, output,screenshot)**		
	h	Maximum-subarray-sum - Hackerrank **(program, output,screenshot)**		
	h	Difference between linear search and binary search		
	i	Viva Questions and Answers on all searching techniques		
2	1	**Recursion**	18/08/2023	25-30
	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)**		
		Additional Problems		
	e	Recursive-digit-sum -Hackerrank **(program, output,screenshot)**		
	f	the-power-sum-Hackerrank **(program, output,screenshot)**		
	g	permutations-of-strings-Hackerrank **(program, output,screenshot)**		
	h	Difference between Recursion and Iteration **(program, output,screenshot)**		
	i	Viva Questions and Answers		
3	3	**Sorting**	25/08/2023	31-46
	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		
		Additional Problems		
	l	Big-sorting -Hackerrank **(program, output,screenshot)**		
	m	Lilys-homework -Hackerrank **(program, output,screenshot)**		
	n	Almost-sorted -Hackerrank **(program, output,screenshot)**		
	o	Viva Questions and Answers on all Sorting techniques		
4	4	**Linked List**	08/09/2023	47-67
	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	68-74
	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	75-83
	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	84-100
	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	101-106
	A,B	ANY 2 PROBLEMS FROM 1-4 IN UNIT-3 PROBLEMS ON STACKS AND QUEUES.DOCX FILE **(program, output,screenshot for two test cases)**		
	C,D	ANY 2 PROBLEMS FROM 5-8 UNIT-3 PROBLEMS ON STACKS AND QUEUES.DOCX FILE **(program, output,screenshot for two test cases)**		
9	**9**	**Binary Trees**	03/11/ 2023	107-122
	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	123-136
	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	137-142
	a	dfs		
	b	bfs		
11	**11**	**Graphs - Minimum cost spanning trees**	17/11/ 2023	143-148
	a	prims		
	b	kruskals		

22501A0545

Date: 11/8/23

EXPERIMENT-2

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

(i) Sequential search

(ii) Binary search

Description:

Searching techniques are used to find particular information in the given set of data. Searching algorithms vary from data structure to data structure. When a list of data elements is represented as a linear list, the search algorithms like sequential search, binary search and other searching algorithms are used.

1. Linear/Sequential Search:

Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found.The linear search algorithm searches in sorted or unsorted data. In linear search we traverse from start to end.

Algorithm:

/*given unsorted data to search, Algorithm for linear search***/**

1. Read the length of the array
2. Read data from user store it in an Array, say a[ ]
3. Read element to be searched
4. Initialise i to 0
5. Initialise count to -1
6. Traverse the array from index 0 to n
7. If element is found increment count
8. break the loop
9. Else increment i
10. If count is -1, print element is not found
11. Else print element is found

Example:

A[5] = {5,3,2,7,6} count = -1

Length of the array n = 5

Search element x = 2

Step1: i = 0 , i< n, true

A[i] == x (5==2) false

i = i+1=1

Step2: i = 1 , i< n, true

A[i] == x (3==2) false

i = i+1=1

Step3: i = 2 , i< n, true

A[i] == x (2==2) true

Count = count +1 = 0

Step4: count == -1 false

prints : The element is found

Advantages:

1. Will perform fast searches of small to medium lists.
2. The list does not need to be sorted.
3. Not affected by insertions and deletions.

Disadvantages:

Linear search needs greater time complexities compared to other searching algorithms, so it consumes more time.

Applications:

It is used to search items when we are given an unsorted list of items; considering that we need to find a book in box of randomly arranged books, we use linear search.

Linear Search on Integers:

Code:

```
/*linear Search on numericals*/
#include<stdio.h>
int main(){
    int a[]={22,55,44,66,75,6354};
    int n=sizeof(a)/sizeof(a[0]);
    int count=0,target;
    printf("enter the target:");
    scanf("%d",&target);
    int i=0;
    while(i<n){
        if (a[i]==target){
            count++;
            printf("\nelement is found at %d position",i);
            break;
        }
        i++;
    }
    if (count==0)
        printf("\nelement not  found");
    return 0;

}
```

Output:

Element Found:

enter the target:55

element is found at 1 position

Element not found:

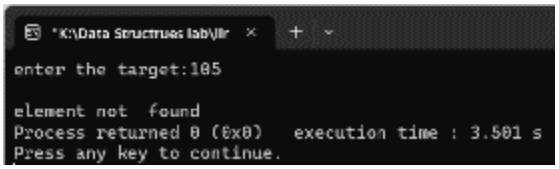
enter the target:1320

element not found

Screenshots:

Element found:

Element not found:



22501A0545

Linear Search characters

Code:

```
#include<stdio.h>
int main(){
    char a[]={'1','s','f','7','10'};
    int n=sizeof(a)/sizeof(a[0]);
    char target;
    printf("enter the element:");
    scanf("%c",&target);
    int i=0,count=0;
    while(i<n){
        if (a[i]==target){
            printf("\nchar found at %d position",i);
            count++;
            break;
        }
        i++;
    }
    if(count==0)
        printf("\nchar not found!");
    return 0;
}
```

Output:

Element found:

```
enter the element:1
char found at 0 position
```

22501A0545

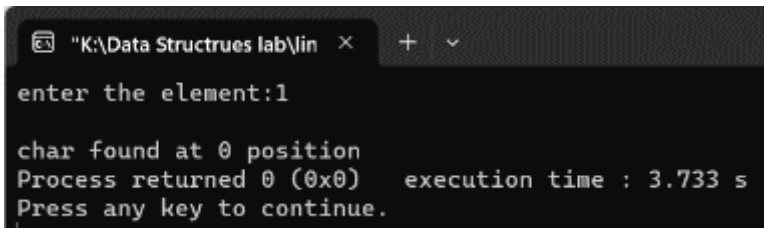
Element not found:

enter the element:56

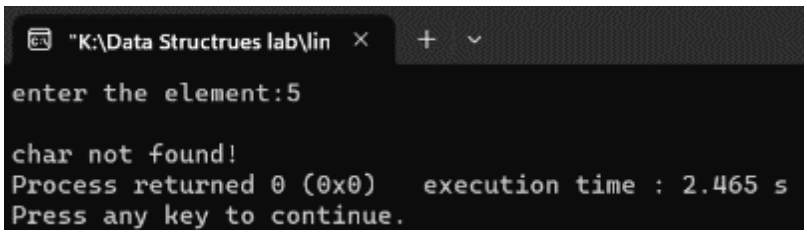
char not found!

Screenshots:

Element Found:



Element not found



Linear Search Strings

Code:

```
#include<stdio.h>
int main(){
    char a[50][50];
    int k;
    printf("no.of strings:\n");
    scanf("%d",&k);
    printf("entering the strings:");

    for(int i=0;i<k;i++){
        scanf("%s",&a[i]);
    }

    char target[50];
    printf("\nenter the target:");
    scanf("%s",&target);
    int i=0,count=0;
    while(i<k){
        if (strcmp(target,a[i])==0){
            printf("\nString is found at %d position",i);
            count++;
            break;
        }
        else {
            i++;
        }
    }
    if(count==0)
        printf("string not found");
    return 0;
}
```

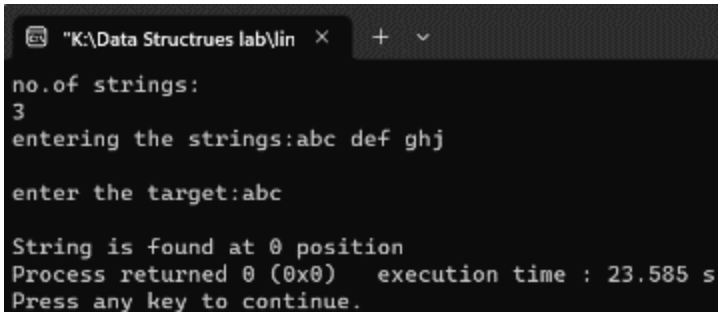
Output:

```
Element found
no.of strings:
3
entering the strings:abc efi ghi
enter the target:efi
String is found at 1 position

Element Not found
no.of strings:
3
entering the strings:abc efi ghi
enter the target:asd
```

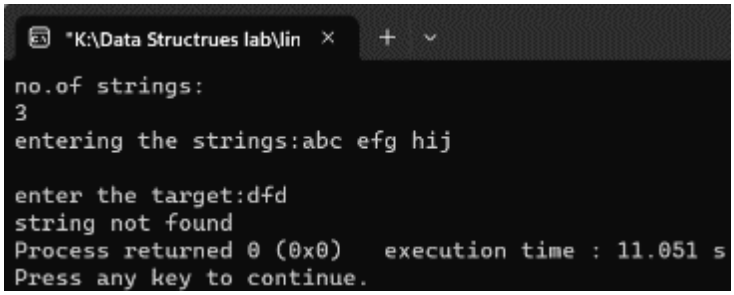
22501A0545

string not found

Screenshots:
Element found



Element not found



2. Binary search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity.

Algorithm:

/*given unsorted data to search, Algorithm for linear search***/**

1. Read the size of the array n
2. Read the elements in the array,say A]
3. Read the search element x
4. Initialize count=-1,low=0,high= n-1,mid
5. while(low<=high){
6. mid = (low+high)/2
7. If A[mid] == x
8. Increment the count and break the loop
9. Else if A[mid] < x
10. Low = mid +1
11. Else
12. High = mid-1
13. If count == -1 print element is not found
14. Else print element is found

Example:

A[7] = {2,4,6,9,10,15,45} count = -1

Length of the array n = 7

Search element x = 4 ; low = 0 and high = n-1=6

Step1: low<=high true

Mid = (low + high)/2 = 3

A[mid] = 10 not equal to4

A[mid] > search element so high = mid - 1 = 2

Step2: low<high true

Mid = (low + high)/2 = 1

A[mid] = 4 equal to 4

Prints element is found

Advantages:

1. When it comes to comparing large data, it is quite efficient as it works on the technique to eliminate half of the array element.
2. It has less compilation time and thus better time complexity.

3. As it breaks the array in half, it is considered an improvement over linear search

Disadvantages:

The process of sorting and searching in an unsorted array will take time

Applications:

1. Used to search words in dictionary.
2. Used to search items in E-commerce websites, and other online websites

Binary Search for Numericals

```
//binary search for numericals
#include<stdio.h>
int main(){
    int a[10] = {1,2,8,68,99};
    int n = sizeof(a)/sizeof(a[0]);
    int target;
    scanf("%d",&target);
    int low=0,high=n-1,mid;
    while(low<=high){
        mid = (low+high)/2;
        if(a[mid]==target){
            printf("target found at %d ",mid);
            return 0;
        }
        if(a[mid] >target)
        high = mid-1;
        else
        low = mid +1;

    }
    printf("not found");
    return 0;
}
```

Output:

Element Found

2

target found at 1

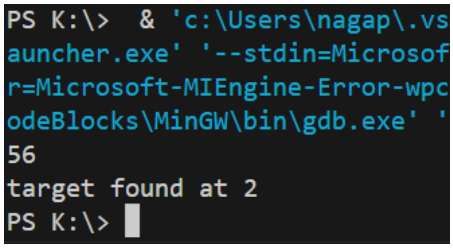
Element not found

15

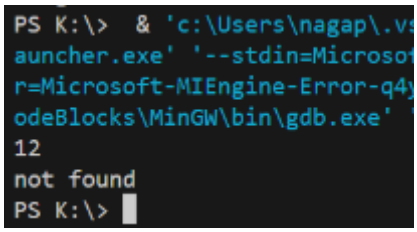
not found

Screenshots:

Element found



Element Not Found



Binary search for characters

```
//binary search for characters
#include<stdio.h>
int main ()
{
  char a[] = { 'a', 'b', 'c', 'd' };
  int n = sizeof (a) / sizeof (a[0]);
  int i = 0, count = 0;

  char target;
  printf ("enter the target:");
  scanf ("%c", &target);
  int low = 0, high = n - 1, mid;

  while (low <= high){
     mid = (low + high) / 2;
       if (a[mid] == target){
         printf ("\ntarget found at %d position", mid);
         count++;
         break;
            }
         else if (a[mid] < target)
                low = mid + 1;
         else
               high = mid - 1;
  }
  if (count == 0)
     printf ("\nelement not found !");
  return 0;
}
```

Output:

Element Found:

enter the target:a

target found at 0 position

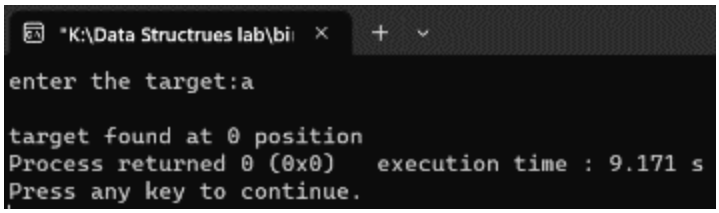
Element not found:

enter the target:s

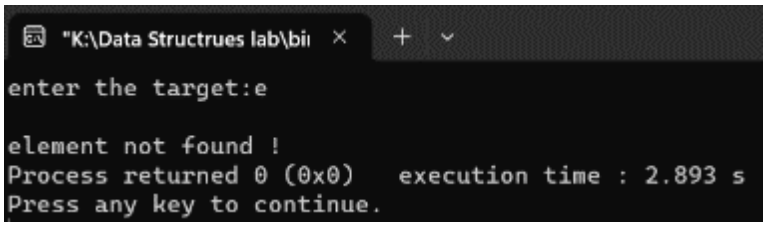
element not found !

Screenshots:

Element found:



Element not found



Binary search for strings

```
#include<stdio.h>
#include<string.h>
int main(){
    char a[50][50];
    int k;
    printf("no.of strings:\n");
    scanf("%d",&k);
    printf("entering the strings:");
    for(int i=0;i<k;i++){
        scanf("%s",&a[i]);
    }
    char target[50];
    printf("enter the string to be search:");
    scanf("%s",&target);
    int low=0,high=k-1,mid;
    while(low<=high){
        mid=(low+high)/2;

        if (strcmp(target,a[mid])==0){
            printf("\nstring found at %d position",mid);
            return 0;
        }
        else if(strcmp(target,a[mid])>0)
        low=mid+1;
        else
        high=mid-1;
    }
    printf("\nelement not found");
    return 0;
}
```

Screenshots

Element found

```
PS K:\>  & 'c:\Users\nagap\.vscode
auncher.exe' '--stdin=Microsoft-MI
r=Microsoft-MIEngine-Error-hnmme40
odeBlocks\MinGW\bin\gdb.exe' '--in
no.of strings:
3
entering the strings:abc def ghi
enter the string to be search:def

string found at 1 position
```

Element not found:

```
PS K:\>  & 'c:\Users\nagap\.vscode\
auncher.exe' '--stdin=Microsoft-MIE
r=Microsoft-MIEngine-Error-t1naew0f
odeBlocks\MinGW\bin\gdb.exe' '--int
no.of strings:
3
entering the strings:abc def ghi
enter the string to be search:jik

element not found
```

Date: 18/08/2023

EXPERIMENT-1

Aim: Demonstrate recursive algorithms with examples.

(i) Factorial of a Number

(ii) Towers of Hanoi Problem

Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH),and many other iterative problems. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

Limitations:

Each time you call a function you use up some of your memory. If there are large number of recursive calls – then you may run out of memory.

1. Factorial of a given number:

The factorial is the product of all positive integers less than or equal to n. The Factorial of a number can be represented as n!
EX: 5! =5*4*3*2*1=120

Algorithm:

1. Initialise a function, say fact taking n as arguments
2. Base condition, if n is equal to 1 then return 1
3. Recursive condition, return n*fact(n1)
4. Read the number n
5. Pass n in parameters for the function fact
6. Print the factorial

Factorial

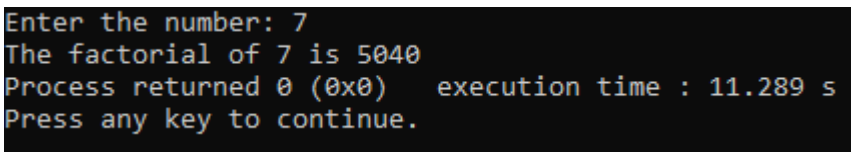
Code:

```
//Factorial using recursion
#include<stdio.h>

int fact(int);

int main(){
    int n;
    printf("Enter the number: ");
    scanf("%d",&n);
    printf("The factorial of %d is %d",n,fact(n));
    return 0;
}
int fact(int n){
    if(n==1)
        return 1;
    else
        return n*fact(n-1);
}
```

Output:

```
Enter the number: 7
The factorial of 7 is 5040
```

Screenshots:



Towers Of Hanoi

Towers of hanoi is a mathematical puzzle which consists of three rods, and a number of disks of different sizes which should be transferred from source to destination using the intermediate rod.

Rules:

1. Only one disk may be moved at a time
2. Each move consists of taking the upper disk from one of the rods and sliding into another rod, on the top of the other disks that may be already present on the rod.
3. No disk may be placed on top of a smaller disk.

Algorithm:

1. Start
2. Read N value as the no. of disks
3. Call TOH(N, S, I, D).
4. Stop

```
TOH (N, S, I, D)
{
        if n = 1 then
                Transfer disk from S to D and stop
        Else
                Call TOH(N-1, S, D, I)
                Transfer disk from S to D
                Call TOH(N-1, I, S, D);
}
```

Towers Of Hanoi

```
#include<stdio.h>

void towers(int ,char, char, char);

int main(){
    //The no of disks
    int n;
    printf("Enter the no of Disks: ");
    scanf("%d",&n);
    towers(n,'A','B','C');
    return 0;
}
// a = source, b= destination, c= aux
void towers(int n,char a,char b,char c){
    if(n==1){
        printf("\nMove disk 1 from %c to %c",a,b);
    }
    else{
        towers(n- 1, a, c, b);
        printf("\nMove disk %d from %c to %c",n,a,b);
        towers(n- 1, c, b, a);
    }
}
```

Output:

```
Enter the no of Disks: 3
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
```

Screenshots:

Binary Search Using Recursion
Code:

```
#include<stdio.h>

int binary(int[],int,int,int);
int mid;

int main(){
    int a[] = {2,8,10,15,25,45,48};
    int search,size;

    size = sizeof(a)/sizeof(a[0]);

    printf("Enter the number to be searched: ");
    scanf("%d",&search);

    int res = -1;
    res = binary(a,search,0,size-1);

    if(res >= 0)
        printf("The element is found at %d",mid);
    else if(res < 0)
        printf("The element is not found");
    return 0;

}

int binary(int a[], int search,int low, int high){

    mid = (low + high)/2;
    if(low <= high){
    if(a[mid] == search)
        return mid;
    else if(a[mid] < search)
        return binary(a, search,(mid+1), high);
    else if(a[mid] > search)
        return binary(a, search,low, (mid-1));
    }
    return -1;
}
```

Outputs:

Elements found

```
Enter the number to be searched: 8
The element is found at 1
```

Elements not found

```
Enter the number to be searched: 325
```

22501A0545

The element is not found

Screenshots:

found

```
Enter the number to be searched: 15
The element is found at 3
Process returned 0 (0x0)   execution time : 6.823 s
Press any key to continue.
```

Not found:

```
Enter the number to be searched: 455
The element is not found
Process returned 0 (0x0)   execution time : 3.520 s
Press any key to continue.
```

22501A0545

Date: 25/08/2023

EXPERIMENT-3

Aim: Implement different types of sorting techniques on a given Array

(i) Bubble sort
(ii) Insertion sort
(iii) Selection sort
(iv) Merge sort
(v) Quick Sort

Description:

Sorting algorithms are a set of instructions that take an array or list as an input and arrange the items into a particular order. Sorting is most commonly used for numerical or for characters.We can distinguish two types of sorting:

1. Internal Sorting, when data is in main memory
2. External Sorting, When data doesn't fit into main memory we sort it in the external memory.

1. Bubble Sort:

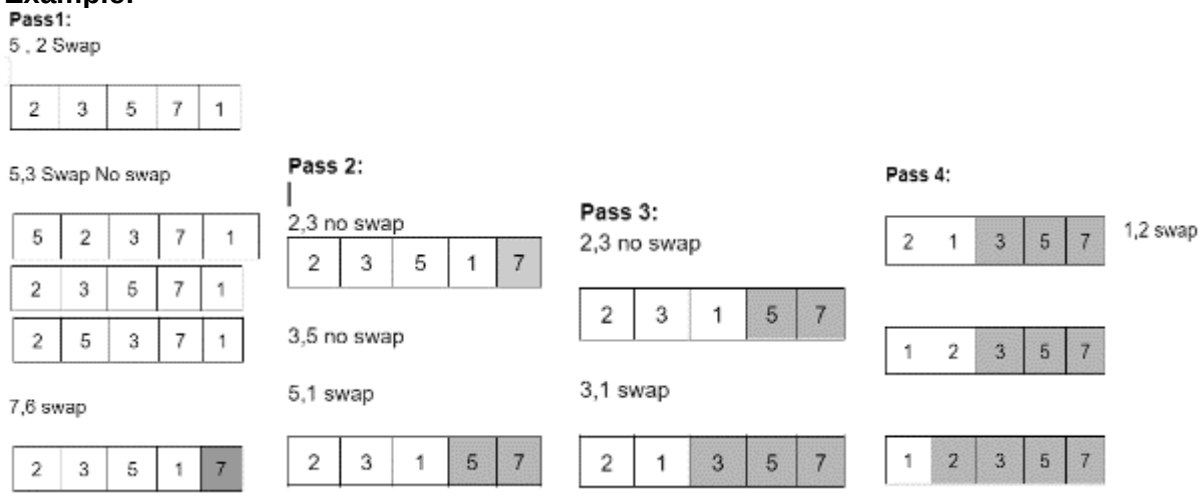
Bubble sort algorithm works by comparing each item in the list with the item next to it, and swapping them if the next element is smaller than the previous. the largest element has bubbled And placed at the last position of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

Example:

Consider the array a[ ] = {5,2,3,7,8} size of the array n = 5

5	2	3	7	1

Example:



Pass 5:

1	2	3	5	7

22501A0545

Algorithm:

1. Start by comparing 1st and 2nd element and swap if the 1st element is greater.
2. After that do the same all the elements
3. At the end of the cycle you will get the max element at the end of the list.
4. Now do the same in all subsequent cycles.
5. Perform this of the size of the array -1 times.
6. Array is sorted in place

Applications:

1. Bubble sort is a sorting algorithm that is used to sort the elements in an ascending order.
2. Bubble sort can be beneficial to sort the unsorted elements in a specific order.

Advantages of Bubble Sort:

1. Bubble sort is easy.
2. It does not require any additional memory space.

Disadvantages of Bubble Sort:

1. Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
2. Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

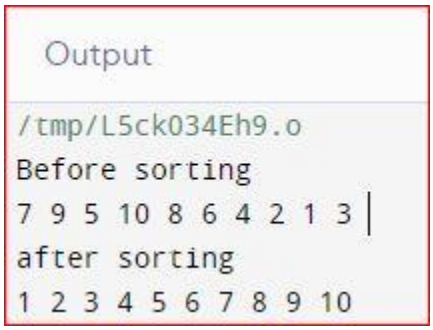
Performance:

Time Complexity: $O(N^2)$ for average, worst for best case O(N)
Space Complexity: O(1)

Bubble Sort:

```
#include<stdio.h>
int main(){
    int temp,a[] = {7,9,5,10,8,6,4,2,1,3};
    int n = sizeof(a)/sizeof(a[0]);
    printf("Before sorting\n");
    for(int i = 0;i<n;i++){
        printf("%d ",a[i]);
    }

    for(int i=0; i<n-1; i++){
        for(int j=0; j<n-i-1; j++){
            if(a[j] > a[j+1]){
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
    printf("\nafter sorting\n");
    for(int i = 0;i<n;i++){
        printf("%d ",a[i]);
    }
    return 0;
}
```

ScreenShots:



22501A0545

Insertion Sort:

Insertion sort is a simple sorting algorithm. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part. The pass adds the element from the unsorted array to the sorted array. After n-1 passes the elements are placed in sorted order.

Algorithm:

1. If it is the first element, it is already sorted. return 1;
2. Pick next element
3. Compare with all elements in the sorted sub-list
4. Shift all the elements in the sorted sub-list that is greater than the value to be sorted
5. Insert the value in the sorted position
6. Repeat the steps until all elements are sorted.

Example:

3	2	4	1	10

Pass1 :

3	2	4	1	10

2	3	4	1	10

Pass2:

2	3	4	1	10

Pass 3:

Key < a[j] key = 4

2	3	4	1	10

2	3	4	4	10

2	3	3	4	10

2	2	3	4	10

A[j+1] = key :

1	2	3	4	10

Pass 4:

Key > a[j] no swapping

1	2	3	4	10

Advantages:

1. It's efficient for small data sets.
2. It just necessitates a constant amount of O(1) extra memory space.
3. It works well with data sets that are partially sorted.

Disadvantages:

1. Insertion sort is inefficient against more extensive data sets

2. The insertion sort exhibits the worst-case time complexity of O(n2)

Performance:
Time Complexity: for worst and average case $O(N^2)$, for best case O(N)
Space Complexity: O(1)

Applications:

1.Real world Application of insertion sort isTailors arrange shirts in a closet.
2.Sorting “almost sorted” lists.
3.Sorting Small Sub-lists Merge Sort; although it may not be the best way to use insertion sort but it gives a clear idea about Complex sorting process in Merge Sort

Insertion Sort:

```
#include<stdio.h>
int main(){
    int temp,a[] = {5,8,71,0,12,1};
    int n = sizeof(a)/sizeof(a[0]);
    printf("Before sorting\n");
    for(int i = 0;i<n;i++){
        printf("%d ",a[i]);
    }
    int i,j,key;

    for(i=1; i<n; i++){
        key = a[i];
        j = i-1;
        while(j>=0 && key<a[j]){
                a[j+1] = a[j];
                j--;
        }
        a[j+1] = key;
    }

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

Output:

```
Output
/tmp/L5ck034Eh9.o
Before sorting
5 8 71 0 12 1
after sorting
0 1 5 8 12 71
```

22501A0545

Selection Sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison based algorithm.The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving the unsorted array boundary by one element to the right.

Algorithm:

1. Set minIndex to position 0
2. Search for the smallest element in the unsorted subarray and update minIndex
3. Swap the element at the position minIndex with the first element of the unsorted subarray.
4. Again set minIndex to the first position of the unsorted subarray
5. Repeat until all the elements are sorted.

Example:

Array a[5] =

84	4	21	41	14

Pass1:

84	4	21	41	14

Minval = 84, after comparing all the elements 4 is the new min value
Swap 84 and 4

4	84	21	41	14

Pass 2:

4	84	21	41	14

Minval = 21, after comparing all the elements 14 is the new min value
Swap 84 and 14

Pass 3:

4	84	21	41	14

Minval = 41, after comparing all the elements 14 is the new min value
Swap 84 and 14

4	14	21	41	84

Pass 4:

4	14	21	41	84

Minval =41, after comparing all the elements No swap

4	14	21	41	84

Applications:

1. For small unsorted data sets
2. If in a given problem checking of all the elements is compulsory we can go with Selection sort

Advantages:

1. It's efficient for small data sets.
2. It just necessitates a constant amount of O(1) extra memory space.

Disadvantages:

1. Selection sort is inefficient on large data sets since its worst case time complexity is O(n2)
2. The Selection sort's the worst-case time complexity of O(n2)

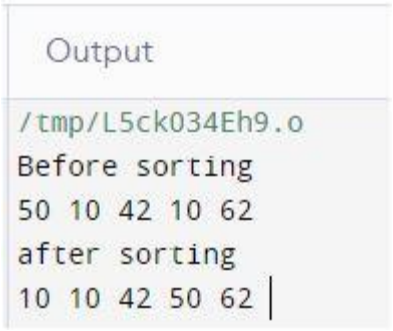
Performance:
Time Complexity: $O(N^2)$ for best, average ,worst cases
Space Complexity: O(1)

Selection Sort:

```
#include<stdio.h>
int main(){
   int temp,a[] = {50,10,42,10,62};
   int n = sizeof(a)/sizeof(a[0]);
   printf("Before sorting\n");
   for(int i = 0;i<n;i++){
      printf("%d ",a[i]);
   }
   int i,j,index;

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

```
        index=i;
        for(j=i+1;j<n;j++){
            if(a[index]>a[j]){
                index=j;
            }
        }
        if(index!=i){
            temp=a[i];
            a[i]=a[index];
            a[index]=temp;
        }
    }

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

Screenshots:



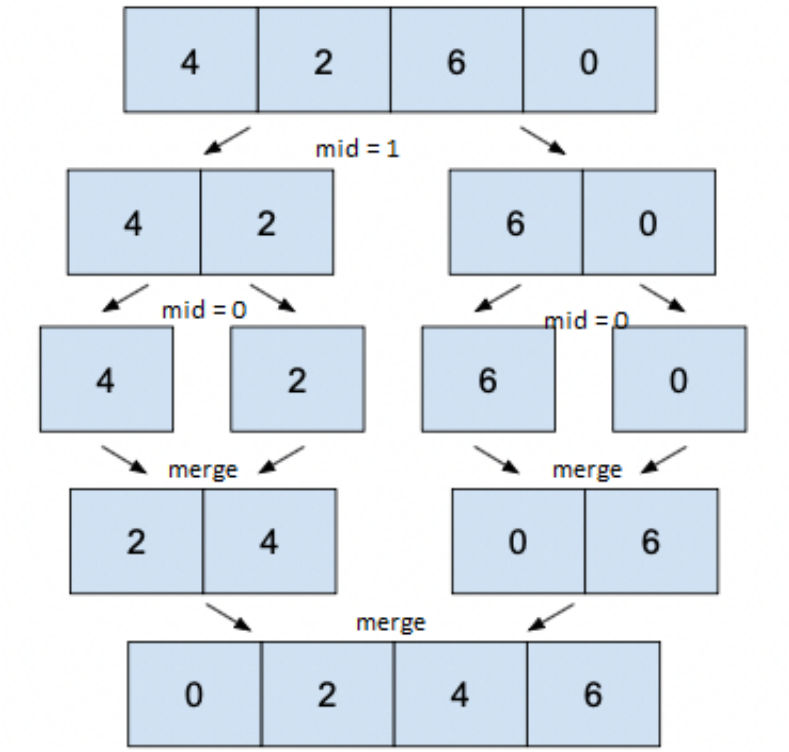
Merge Sort:

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. Merge sort follows the divide and conquer algorithm to sort the array. Merge sort has a worst case scenario of O(nlogn), which means it can perform well even on large data sets.

Algorithm:

1. Read and pass the array to a function,say mergesort() which takes array first index and last index as arguments
2. Calculate mid in mergesort
3. In mergesort divide the array into 2 sub arrays from first to mid and mid to last index
4. Call a new function, say merge
5. The merge function sorts the array and combines two array
6. The recursive function calls continue until all the elements are sorted

Example:



Procedure:

First array is divided into 2 halves first half containing **4,2** and second half **6,0**
4,2 is further divided and it is again divided into **4 and 2** they were merged as **2,4**
In the 2nd subarray **6 and 0** are divided into individual elements 6 and 0 and merged as **0 and 6**. The 2 sub arrays are merged into a sorted array.

Advantages:

1. Stability: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.

2. Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
3. Parallelizable: Merge sort is a parallelizable algorithm, which means it can take advantage of multiple processors or threads.

Disadvantages:
1. Require additional memory .
3. Not efficient for small data sets .

Applications:
1. Sorting large datasets.
2. External sorting: Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
3. Custom sorting: Merge sort can be adapted to handle different input distributions, such as partially sorted,nearly sorted or completely sorted data.

Performance:
Time Complexity: O(nlogn)
Space Complexity: O(n)

MergeSort:
```
#include<stdio.h>
void mergesort(int a[],int l,int h){
   if(l<h){
      int mid = (l+h)/2;
      mergesort(a,l,mid);
      mergesort(a,mid+1,h);
      merge(a,l,mid,h);
   }
}

void merge(int arr[],int low,int mid,int high){
   int n1 = mid-low+1;
   int n2 = high-mid;
   int a[n1],b[n2];
   for(int i=0;i<n1;i++)
      a[i] = arr[low+i];
   for(int i=0;i<n2;i++)
      b[i] = arr[mid+i+1];

   int i=0,j=0,k=low;
   while(i<n1 && j<n2){
      if(a[i] > b[j]){
         arr[k] = b[j];
         k++;
         j++;
      }
      else{
         arr[k] = a[i];
         k++;
         i++;
      }
   }
   while(i<n1){
      arr[k] = a[i];
      k++;
```

```
        i++;
    }
    while(j<n2){
        arr[k] = b[j];
            k++;
            j++;
    }
}

int main(){
    int a[] = {9,5,4,7,9};
    int n = sizeof(a)/sizeof(a[0]);
    printf("Before sorting\n");
    for(int i = 0;i<n;i++){
        printf("%d ",a[i]);
    }

    mergesort(a,0,n-1);

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

Screenshots:

```
Output

/tmp/bbFeSQ1Frq.o
Before sorting
9 5 4 7 9
after sorting
4 5 7 9 9
```

Quick Sort:

Each iteration of the quick sort selects an element, known as pivot, and divides the list into three groups: a partition of elements whose keys are less than the pivot's key, the pivot element that is placed in its ultimately correct location in the list, and a partition of elements greater than or equal to the pivot's key. The sorting then continues by quick sorting the left partition followed by quick sorting the right partition.

Algorithm:

1. Construct two functions, say qs() and part()
2. Consider a pivot element as the last element in the array
3. The part function sets the pivot element to the correct position as it was in a sorted array
4. The qs function does the recursive function calls
5. Qs splits the array after the partition is done from the pivot +1 to highest index and lowest index to pivot -1
6. The sorting is done after the recursive function calls

Example:

3 j	1	7	8	4 high

Consider array,

Pivot = a[4] = 4
J = 0, i=-1,a[high] = 4;
1. (A[j] < = pivot) =>

3 i	1 j	7	8	4 high

a[j]< = 4 swap a[i] and a[j] (3,3)
*963.

3	1 i	7 j	8	4 high

a[j]< = 4 1<=4 true swap a[i] and a[j]

(1,1)

3	1	4	8	7

A[j] <= 4 7<=4 false
Swap (a[i+1], a[high])

Left sub array from 0 index to 1

3	1 pivot	4	8	7

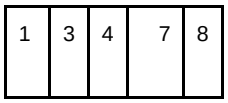
Pivot = 1

1. A[j] <= pivot 3<= 1 false j++
1<1 false
Swap a[high], a[i+1]

1	3	4	8	7

Right sub array:

A[j] < =pivot 8<=7 false
Swap a[high], a[i+1] 8,7



Quick Sort:

```
#include<stdio.h>
int partition(int a[],int low,int high){
    int i=low-1,pivot = a[high];
    for(int j=low;j<high;j++){
        if(a[j] <= pivot){
            i++;
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    int temp = a[high];
    a[high] = a[i+1];
    a[i+1] = temp;
    return i+1;
}
void quicksort(int a[], int l,int h){
    if(l<h){
        int p = partition(a,l,h);
        quicksort(a,l,p-1);
        quicksort(a,p+1,h);
    }
}
int main(){
    int a[] = {9,6,2,1};
    int n = sizeof(a)/sizeof(a[0]);
    printf("before Quick sort: \n");
    for(int i =0;i<n;i++)
        printf("%d ",a[i]);
    quicksort(a,0,n-1);
    printf("\nAfter Quick sort: \n");
    for(int i =0;i<n;i++)
        printf("%d ",a[i]);
}
```

Output:

```
Output

/tmp/vkPvKdEpyG.o
before Quick sort:
4 5 8 9 1 2 3 6 4
After Quick sort:
1 2 3 4 4 5 6 8 9
```

Advantages:

1. It is in place sorting algorithm
2. Quicksort is a cache-friendly algorithm
3. Quicksort has a best and average-case time complexity of O(N logN), which means it performs well even on large datasets.
4. It works rapidly and effectively.

Disadvantages:

1. It's difficult to implement since it's a recursive process, especially if recursion isn't available.
2. It is not stable sorting algorithm it changes the relative position of elements after sorting
3. When the pivot element or when all of the elements have the same size, the performance of the quicksort is impacted by worst-case scenarios.
4. Not efficient for small data sets .

Applications:

1. Sorting large datasets.
2. It is tail -recursive and hence all the call optimization can be done
3. Variants of Quicksort are used to separate the Kth smallest or largest elements.

Comparison between all 5 sorting Techniques

	Bubble Sort	Selection Sort	Insertion Sort	Merge SOrt	Quick Sort
No of passes	Best and Worst cases n-1	Best and Worst cases n-1	Best and Worst cases n-1	Best and Worst cases logn	Bestcase logn Worstcase n-1
Stability	stable	Not stable	stable	stable	Not stable
Real-World Usage	Rarely used	Rarely used	Small data sets	Large Data sets,external sorting	Large data sets for rapid sorting
Ease of implementation	simple	Simple	simple	moderate	moderate

Time Complexity	Best Case: 0(n) Avg case: $0(n^2)$ Worst Case: $0(n^2)$	Best Case: $0(n^2)$ Avg case: $0(n^2)$ Worst Case: $0(n^2)$	Best Case: 0(n) Avg case: $0(n^2)$ Worst Case: $0(n^2)$	Best Case: 0(nlogn) Avg case: 0(nlogn) Worst Case: 0(nlogn)	Best Case: 0(nlogn) Avg case: 0(nlogn) Worst Case: $0(n^2)$
Space Complexity	0(1)	0(1)	0(1)	O(n)	O(1)

22501A0545

Viva Questions:

1. What is Bubble Sort?
a. Bubble sort algorithm compares the elements and puts the largest element at the last position for each pass. The algorithm continues until all the elements are sorted.

2. Differences between selection sort and insertion sort?
a. Selection sort in each pass puts the smallest element in the first position and continues till data is sorted. Insertion sort separates the array into sorted and unsorted elements, the element in the unsorted position is placed into sorted position in each pass.

3. Explain Merge sort and quicksort?
a. Merge sort divides the array into individual elements and sorts the array and merges it. Quick sort Takes an element and arranges it to the correct position as it was in a sorted arrangement, quick sort is unstable and merge sort is stable. Merge sort uses extra o(N) space quick sort performs in place.

4.Which sorting algorithm is the best sorting algorithm?
a. Sorting algorithms may be divided as quadratic time complexity and logarithmic time complexity, definitely logarithmic performs better than quadratic and it also depends on the length of the data, merge and quick sorts can be used for large data sets and bubble, selection and insertion for the other.

4. Linked List

Date: 08/09/23

Singly LinkedList:

```
#include<stdio.h>
#include<stdlib.h>
struct node{
    int data;`
    struct node *next;
};
struct node *head;
int count=0;
void insert_begin(){
    int value;
    printf("Enter value:\n");
    scanf("%d",&value);
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    if (temp == NULL)
        printf("memory insufficient:\n");
    else{
        temp->data = value;
        if(head == NULL){//IF NO NODES IN LL
            temp->next = NULL;
        }else{
            temp->next = head;
        }
    }
    head = temp;
    count++;
}
void insert_end(){
    int value;
    printf("Enter value:\n");
    scanf("%d",&value);
    struct node *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{
            struct node *p = head;
            while(p->next != NULL){
                p=p->next;
            }
            p->next = temp;
```

```
        }
        count++;
    }
}
void insert_at_part(){
    int value,pos,i;
    printf("Enter Position:\n");
    scanf("%d",&pos);
    if(pos<=count+1 && pos>0){
        struct node *temp;
        temp = (struct node *)malloc(sizeof(struct node));
        if (temp == NULL)
            printf("memory insufficient:\n");
        else{
            if(pos == 1)
                insert_begin();
            else if(pos == count+1)
                insert_end();
            else{
                printf("Enter value:\n");
                scanf("%d",&value);
                temp->data = value;
                temp->next = NULL;
                struct node *p = head;
                for(i=1;i<=pos-2;i++)
                    p=p->next;
                temp->next = p->next;
                p->next=temp;
                count++;
            }
        }
    }
}
void display(){
    struct node *p = (struct node *)malloc(sizeof(struct node));
    p = head;
    if(head == NULL){
        printf("List is empty\n");
    }else{
        while(p->next != NULL){
            printf("%d -> ",p->data);
            p = p->next;
        }
        printf("%d -> NULL\n",p->data);
    }
}
void delete_begin(){
    if(head==NULL)
        printf("List is empty\n");
```

```
    else{
        struct node *p = head;
        head = head->next;
        free(p);
        count--;
    }
}
void delete_end(){
    if(head==NULL)
        printf("List is empty\n");
    else{
        struct node *p = head;
        while(p->next->next != NULL)
            p=p->next;
        struct node *q = p->next;
        p->next = NULL;
        free(q);
        count--;
    }
}
void delete_at_part(){
    int pos,i;
    printf("Enter position\n");
    scanf("%d",&pos);
    if(pos>0 && pos<=count){
        if(pos == 1)
            delete_begin();
        else if(pos == count)
            delete_end();
        else{
            struct node *p = head;
            for(i=1;i<=pos-2;i++)
                p=p->next;
            struct node *q;
            q = p->next;
            p->next = q->next;
            free(q);
            count--;
        }
    }
}
void search(){
    int x,i,found=0;
    printf("enter value to search:\n");
    scanf("%d",&x);
    struct node *p=head;
    i=1;
    while(p->next !=NULL){
        if(p->data == x){
```

```
            found++;
            break;
        }
        i++;
        p = p->next;
    }
    if(p->data == x){
        found++;
    }
    if(found != 0){
        printf("found at %d\n",i);
    }
    else{
        printf("not found\n");
    }
}
int main(){
    int choice;
    while(1){
        printf("Insertion choice: (1.insert, 4.End ,5.Particular position)\n");
        printf("Deletion choice: 6.Begin 7.End 8.Particular position \n");
        printf("2.display 9.Search ,3.Size of List, default exit\n");
        scanf("%d",&choice);
        switch(choice){
            case 1: insert_begin();
                break;
            case 2: display();
                break;
            case 3: printf("no of node: %d\n",count);
                break;
            case 4: insert_end();
                break;
            case 5: insert_at_part();
                break;
            case 6: delete_begin();
            break;
            case 7: delete_end();
            break;
            case 8: delete_at_part();
            break;
            case 9: search();
            break;
            default: exit(0);
        }
    }
}
```

22501A0545

Screenshots:

Inserting At the Beginning:

```
Output
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
10
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
20
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
30
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
30 -> 20 -> 10 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
```

Inserting at the End:

```
Output
1
Enter value:
20
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
30
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
30 -> 20 -> 10 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
4
Enter value:
5
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
30 -> 20 -> 10 -> 5 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
```

Inserting At a particular position:

```
Terminal
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
20
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
5
Enter Position:
2
Enter value:
58
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
20 -> 58 -> 10 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
```

Delete at beginning:

```
Terminal
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
20
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
20 -> 10 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
6
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
10 -> NULL
```

Delete End:

```
Terminal
30
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
7
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
30 -> 20 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
7
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
30 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
```

Delete at particular Position:

```
Terminal
1
Enter value:
24
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
24 -> 65 -> 65 -> 58 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
8
Enter position
3
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
24 -> 65 -> 58 -> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
```

Display Number of nodes:

```
Terminal
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
84
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
651
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
3
no of node: 2
```

Search:

```
Terminal
1
Enter value:
65
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
48
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
9
enter value to search:
48
found at 1
```

22501A0545

Display:

Terminal

```
Enter value:
10
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
69
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
58
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.Particular position
2.display 9.Search ,3.Size of List, default exit
2
58 -> 69 -> 10 -> NULL
```

Doubly Linked List:

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

struct node *head;
int count=0;

void insert_begin(){
    int value;
    printf("Enter value:\n");
    scanf("%d",&value);
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    if (temp == NULL)
        printf("memory insufficient:\n");
    else{
        temp->data = value;
        temp->nn = NULL;
        temp->pn = NULL;
        if(head == NULL){
            temp->nn = NULL;
        }else{
            temp->nn = head;
            head->pn = temp;
        }
    }
    head = temp;
    count++;
}
void insert_end(){
    int value;
    printf("Enter value:\n");
    scanf("%d",&value);
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    //for checking memory allocation
    if (temp == NULL)
        printf("memory insufficient:\n");
    else{
        temp->data = value;
        temp->nn = NULL;
        temp->pn = NULL;
        if(head == NULL)
            head = temp;
        else{
            struct node *p = head;
            while(p->nn != NULL){
                p=p->nn;
            }
            p->nn = temp;
            temp->pn = p;
```

```
        }
        count++;
    }
}
void insert_at_part(){
    int value,pos,i;
    printf("Enter Position:\n");
    scanf("%d",&pos);
    if(pos<=count+1 && pos>0){
        struct node *temp;
        temp = (struct node *)malloc(sizeof(struct node));
        if (temp == NULL)
            printf("memory insufficient:\n");
        else{
            if(pos == 1)
                insert_begin();
            else if(pos == count+1)
                insert_end();
            else{
                printf("Enter value:\n");
                scanf("%d",&value);
                temp->data = value;
                temp->nn = NULL;
                temp->pn = NULL;
                struct node *p = head;
                for(i=1;i<=pos-2;i++)
                    p=p->nn;
                temp->pn = p;
                temp->nn = p->nn;
                p->nn=temp;
                temp->nn->pn = temp;
                count++;
            }
        }
    }
}
void display(){
    struct node *p = (struct node *)malloc(sizeof(struct node));
    p = head;
    if(head == NULL){
        printf("List is empty\n");
    }else{
        while(p->nn != NULL){
            printf("%d <-> ",p->data);
            p = p->nn;
        }
        printf("%d <-> NULL\n",p->data);
    }
}

void delete_begin(){
    if(head==NULL)
        printf("List is empty\n");
    else{
        struct node *p = head;
        head = head->nn;
```

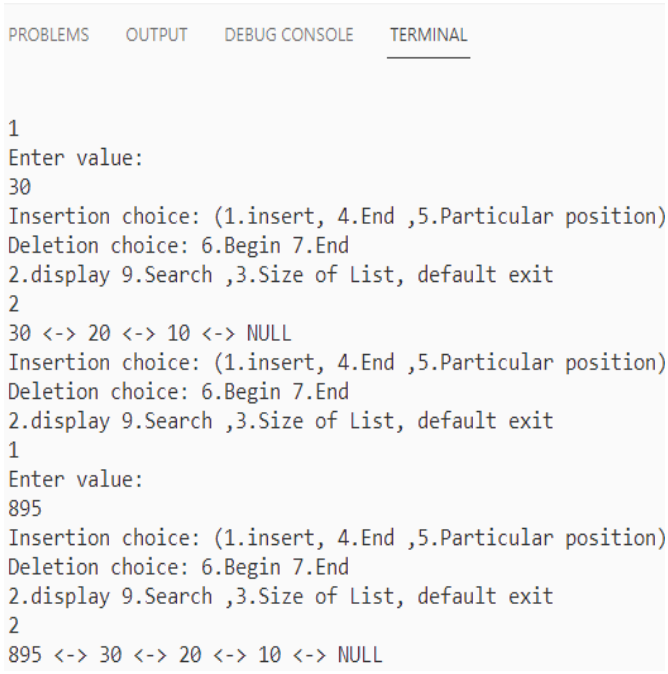
```
        head->pn = NULL;
        free(p);
        count--;
    }
}
void delete_end(){
    if(head==NULL)
        printf("List is empty\n");
    else{
        struct node *p = head;
        while(p->nn != NULL)
            p=p->nn;
        p->pn->nn = NULL;
        free(p);
        count--;
    }
}

void delete_at_part(){
    int pos,i;
    printf("Enter position\n");
    scanf("%d",&pos);
    if(pos>0 && pos<=count){
        if(pos == 1)
            delete_begin();
        else if(pos == count)
            delete_end();
        else{
            struct node *p = head;
            for(i=1;i<=pos-1;i++)
                p=p->nn;
            p->pn->nn = p->nn;
            p->nn->pn = p->pn;
            free(p);
            count--;
        }
    }
}
void search(){
    int x,i,found=0;
    printf("enter value to search:\n");
    scanf("%d",&x);
    struct node *p=head;
    i=1;
    while(p->nn !=NULL){
        if(p->data == x){
            found++;
            break;
        }
        i++;
        p = p->nn;
    }
    if(p->data == x){
        found++;
    }
    if(found != 0){
```

```
        printf("found at %d\n",i);
    }
    else{
        printf("not found\n");
    }
}
void reverse(){
    struct node *p = (struct node *)malloc(sizeof(struct node));
    p = head;
    if(head == NULL){
        printf("List is empty\n");
    }else{
        while(p->nn != NULL)
            p = p->nn;
        while(p->pn != NULL){
            printf("%d <-> ",p->data);
            p = p->pn;
        }
        printf("%d <-> NULL\n",p->data);
    }
}


int main(){
    int choice;
    while(1){
        printf("Insertion choice: (1.insert, 4.End ,5.Particular position)\n");
        printf("Deletion choice: 6.Begin 7.End 8.Particular position\n");
        printf("2.display 9.Search ,3.Size of List, 10.reverse default exit\n");
        scanf("%d",&choice);
        switch(choice){
            case 1: insert_begin();
                break;
            case 2: display();
                break;
            case 3: printf("no of node: %d\n",count);
                break;
             case 4: insert_end();
                break;
            case 5: insert_at_part();
                break;
            case 6: delete_begin();
                break;
            case 7: delete_end();
                break;
            case 8: delete_at_part();
                break;
            case 9: search();
                break;
            case 10: reverse();
            break;
            default: exit(0);
        }
    }
}
```

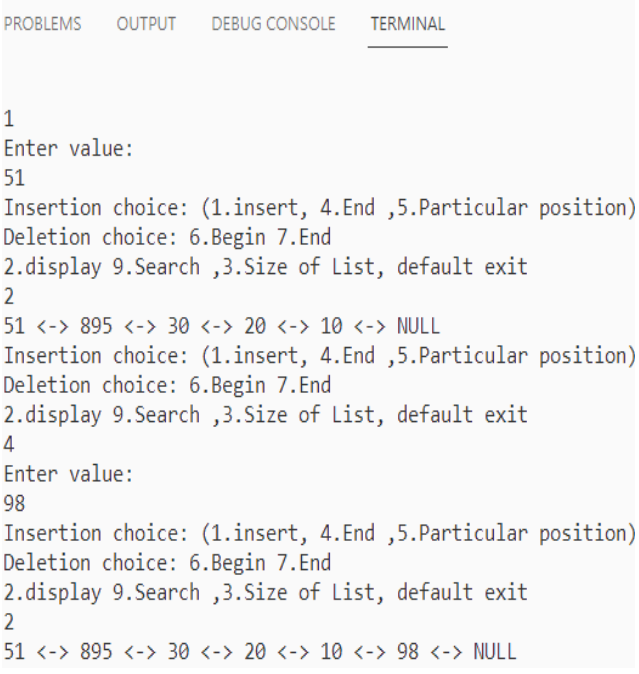
22501A0545

Screenshots:

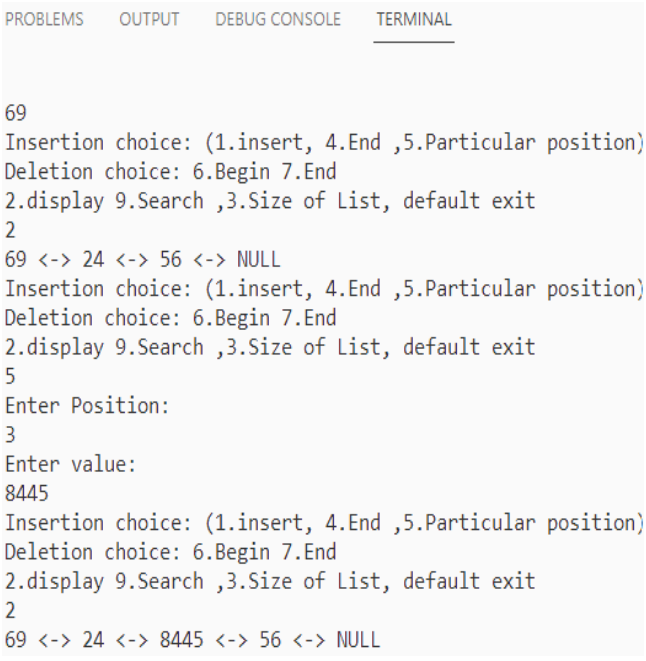
Insert at beginning:



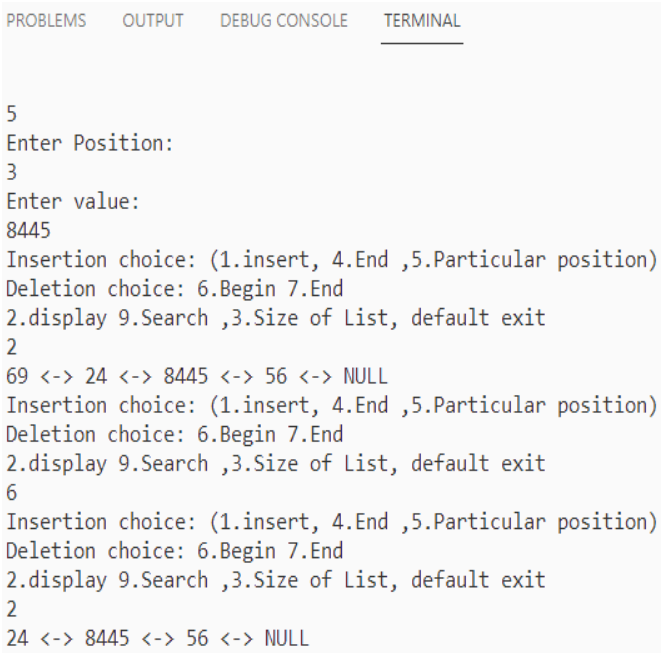
Insert at end:



Insert at particular position:



Delete at beginning:



Delete at end:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

69 <-> 24 <-> 8445 <-> 56 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
6
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
2
24 <-> 8445 <-> 56 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
6
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
2
8445 <-> 56 <-> NULL
```

Delete At particular position:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

2
Enter value:
54
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
2
36 <-> 54 <-> 57 <-> 8445 <-> 56 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
8
Enter position
3
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
2
36 <-> 54 <-> 8445 <-> 56 <-> NULL
```

Search:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

2
10 <-> 20 <-> 30 <-> 40 <-> 50 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
10
50 <-> 40 <-> 30 <-> 20 <-> 10 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
3
no of node: 5
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
9
enter value to search:
30
found at 3
```

Number of Node:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
1
Enter value:
10
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
2
10 <-> 20 <-> 30 <-> 40 <-> 50 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
10
50 <-> 40 <-> 30 <-> 20 <-> 10 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
3
no of node: 5
```

22501A0545

Display:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Enter value:
10
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
20
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
30
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End
2.display 9.Search ,3.Size of List, default exit
2
30 <-> 20 <-> 10 <-> NULL
```

Reverse Display:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL


1
Enter value:
20
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
1
Enter value:
10
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
2
10 <-> 20 <-> 30 <-> 40 <-> 50 <-> NULL
Insertion choice: (1.insert, 4.End ,5.Particular position)
Deletion choice: 6.Begin 7.End 8.particular position
2.display 9.Search ,3.Size of List, 10.reverse, default exit
10
50 <-> 40 <-> 30 <-> 20 <-> 10 <-> NULL
```

Circular Linked List:–

```
#include<stdio.h>
#include<stdlib.h>
struct node{
    int data;
    struct node *next;
};
struct node *head;
int count=0;
void insert_begin(){
    int value;
    printf("Enter value:\n");
    scanf("%d",&value);
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    //for checking memory allocation
    if (temp == NULL)
        printf("memory insufficient:\n");
    else{

        temp->data = value;
        if(head == NULL){//IF NO NODES IN LL
            temp->next = temp;
        }else{
            temp->next = head;
            struct node *p = head;
            while(p->next != head)
                p=p->next;
            p->next = temp;
        }
    }
    head = temp;
    count++;
}
void display(){
    struct node *p = (struct node *)malloc(sizeof(struct node));
    p = head;
    if(head == NULL){
        printf("List is empty\n");
    }else{
        if(count >1){
            printf("head -> ");
            while(p->next != head){
                printf("%d -> ",p->data);
                p = p->next;
            }
            printf("%d -> Head\n",p->data);
        }else if(count==1){
            printf("head %d -> Head\n",p->data);
```

```
        }
    }
}
void insert_end(){
    int value;
    printf("Enter value:\n");
    scanf("%d",&value);
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    //for checking memory allocation
    if (temp == NULL)
        printf("memory insufficient:\n");
    else{
        temp->data = value;
        //temp->next = NULL;
        if(head == NULL){
            head = temp;
            head->next = head;
        }
        else{
            struct node *p = head;
            while(p->next != head){
                p=p->next;
            }
            p->next = temp;
            temp->next = head;

        }
        count++;
    }
}

void insert_at_part(){
    int value,pos,i;
    printf("Enter Position:\n");
    scanf("%d",&pos);
    if(pos<=count+1 && pos>0){
        struct node *temp;
        temp = (struct node *)malloc(sizeof(struct node));
        if (temp == NULL)
            printf("memory insufficient:\n");
        else{
            if(pos == 1)
                insert_begin();
            else if(pos == count+1)
                insert_end();
            else{
                printf("Enter value:\n");
                scanf("%d",&value);
```

```
            temp->data = value;
            temp->next = NULL;
            struct node *p = head;
            for(i=1;i<=pos-2;i++)
                p=p->next;
            temp->next = p->next;
            p->next=temp;
            count++;
          }
        }
    }
}

void delete_begin(){
    if(head==NULL)
        printf("List is empty\n");
    else{
        struct node* q = head;
        struct node *p = head;
        while(p->next != head)
            p=p->next;
        p->next = head->next;
        head=head->next;
        free(q);
        count--;
    }
}

void delete_end(){
    if(head==NULL)
        printf("List is empty\n");
    else{
        struct node *p = head;

        while(p->next->next != head)
            p=p->next;
        struct node *q = p->next;
        p->next = head;
        free(q);
        count--;
    }
}
void delete_at_part(){
    int pos,i;
    printf("Enter position\n");
    scanf("%d",&pos);
    if(pos>0 && pos<=count){
        if(pos == 1)
            delete_begin();
```

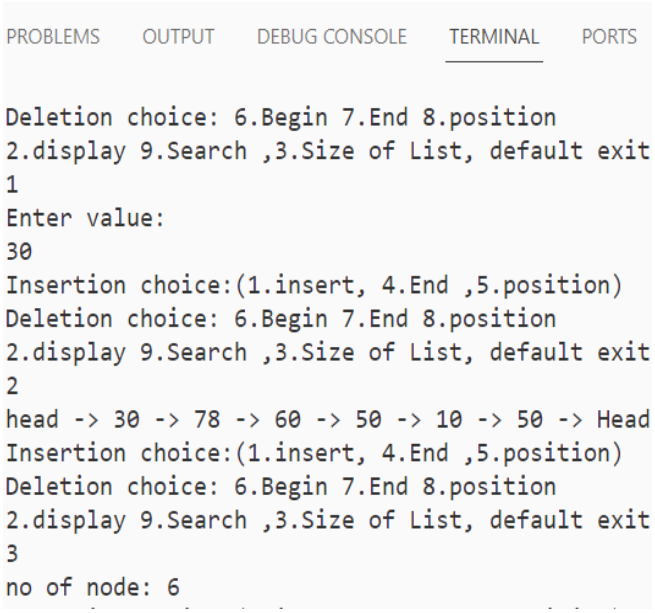
```
        else if(pos == count)
            delete_end();
        else{
            struct node *p = head;
            for(i=1;i<=pos-2;i++)
                p=p->next;
            struct node *q;
            q = p->next;
            p->next = q->next;
            free(q);
            count--;
        }
    }
}
void search(){
    int x,i,found=0;
    printf("enter value to search:\n");
    scanf("%d",&x);
    struct node *p=head;
    i=1;
    while(p->next !=head){
        if(p->data == x){
            found++;
            break;
        }
        i++;
        p = p->next;
    }
    if(p->data == x){
        found++;
    }
    if(found != 0){
        printf("found at %d\n",i);
    }
    else{
        printf("not found\n");
    }
}

void remove_dup(){
    if(head == NULL || head->next == head)
    return;
    struct node* current = head->next;
    do {
        struct node* runner = current;
        do {
            if (runner->next->data == current->data) {
                struct node* temp = runner->next;
                runner->next = temp->next;
```

```
            if (temp == head)
                head = runner->next; // for tail head = runner
            free(temp);
            count--;
        }else
            runner = runner->next;
    } while (runner->next != current);
    current = current->next;
  } while (current != head);
}

int main(){
  int choice;
  while(1){
    printf("Insertion choice:(1.insert, 4.End ,5.position)\n");
    printf("Deletion choice: 6.Begin 7.End 8.position \n");
    printf("2.display 9.Search ,3.Size of List, default exit\n");
    scanf("%d",&choice);
    switch(choice){
      case 1: insert_begin();
        break;
      case 2: display();
        break;
      case 3: printf("no of node: %d\n",count);
        break;

      case 4: insert_end();
        break;
      case 5: insert_at_part();
        break;
      case 6: delete_begin();
      break;
      case 7: delete_end();
      break;
      case 8: delete_at_part();
      break;
      case 9: search();
      break;
      case 10: remove_dup();
      break;
      default: exit(0);
    }
  }
}
```

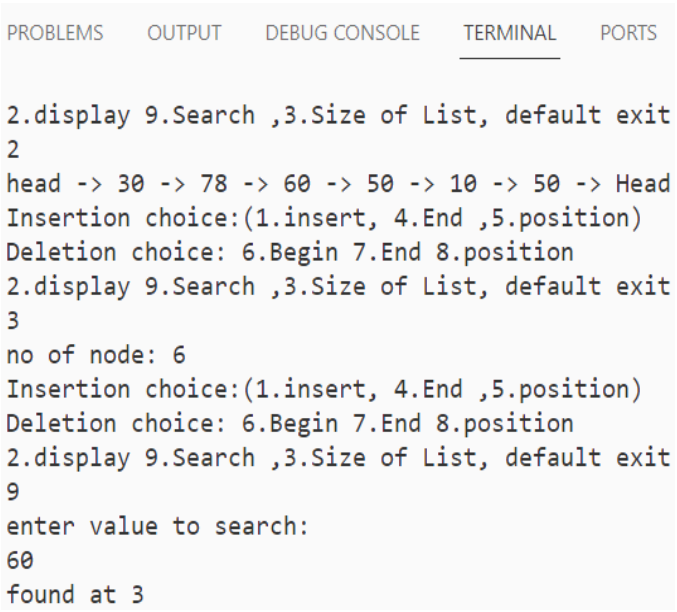
22501A0545

Screenshots:

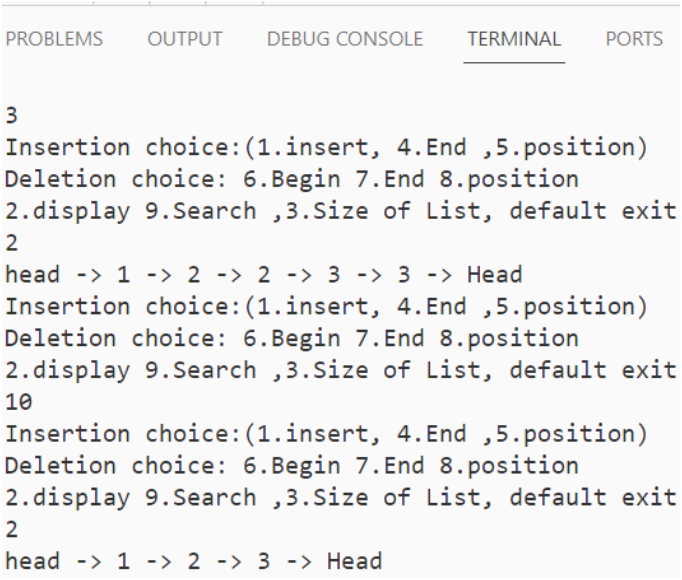
Number of nodes:



Search:



Remove Duplicates:



Insert at the beginning:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
1
Enter value:
14
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
1
Enter value:
252
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 252 -> 14 -> 16 -> Head
```

Insert at End:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 252 -> 14 -> Head
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
4
Enter value:
32
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 252 -> 14 -> 32 -> Head
```

Insert at Particular Position:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
2
head -> 252 -> 14 -> 32 -> Head
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
5
Enter Position:
2
Enter value:
50
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 252 -> 50 -> 14 -> 32 -> Head
```

delete in beginning:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
50
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 252 -> 50 -> 14 -> 32 -> Head
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
6
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 50 -> 14 -> 32 -> Head
```

Delete at end:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
6
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 50 -> 14 -> 32 -> Head
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
7
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 50 -> 14 -> Head
```

Delete At a particular position:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 10 -> 50 -> 14 -> Head
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
8
Enter position
3
Insertion choice:(1.insert, 4.End ,5.position)
Deletion choice: 6.Begin 7.End 8.position
2.display 9.Search ,3.Size of List, default exit
2
head -> 10 -> 50 -> Head
```

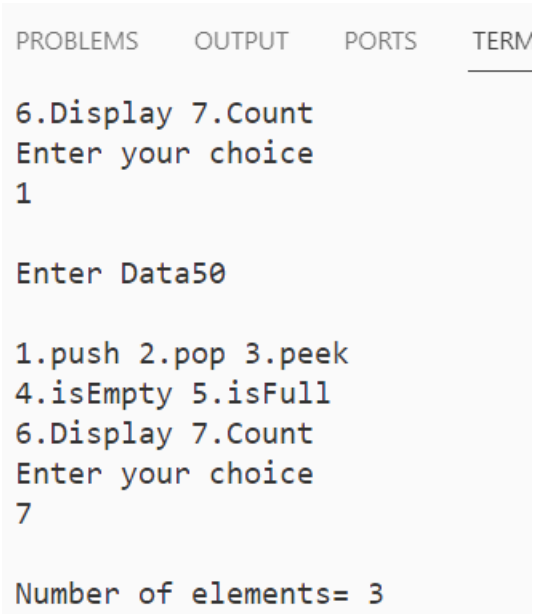
5. STACKS date: 20/09/23

Stacks Using Arrays:

```
#include<stdio.h>
#include<stdlib.h>
int size,top=-1,choice;
   int *a;
   void push(){
      int data;
      printf("\nEnter Data");
      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",a[top]);
         top--;
   }
   }
   void peek(){
      if(top==-1)
         printf("Stack is empty\n");
      else
         printf("%d is the top",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("\nNumber of elements= %d",top+1);
```

```
    }
void main(){
    printf("enter the size of stack");
    scanf("%d",&size);
    a=(int *)malloc(sizeof(int)*size);
    while(1){
    printf("\n1.push 2.pop 3.peek\n");
    printf("4.isEmpty 5.isFull\n");
    printf("6.Display 7.Count\n");
    printf("Enter your choice\n");
    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);
    }
}
}
```

ScreenShots:



22501A0545

1.Push

```
PROBLEMS     OUTPUT     PORTS


1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
1

Enter Data20

1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
6
20
```

2.Pop

```
PROBLEMS     OUTPUT     PORTS

Enter Data30

1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
6
20
30

1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
2
30 data is removed
```

3.peek

```
PROBLEMS     OUTPUT     PORTS


6.Display 7.Count
Enter your choice
1

Enter Data40
Stack is full

1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
3
20 is the top
```

4.isEmpty

```
PROBLEMS     OUTPUT     PORTS


1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
3
20 is the top
1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
4
Stack is not empty
```

5.isFull

6.Display

```
PROBLEMS     OUTPUT     PORTS

Enter your choice
1

Enter Data12
Stack is full

1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
5
Stack is full
```

```
PROBLEMS     OUTPUT     PORTS


Enter Data50

1.push 2.pop 3.peek
4.isEmpty 5.isFull
6.Display 7.Count
Enter your choice
6
20
30
40
50
```

Stacks using Linked lists:

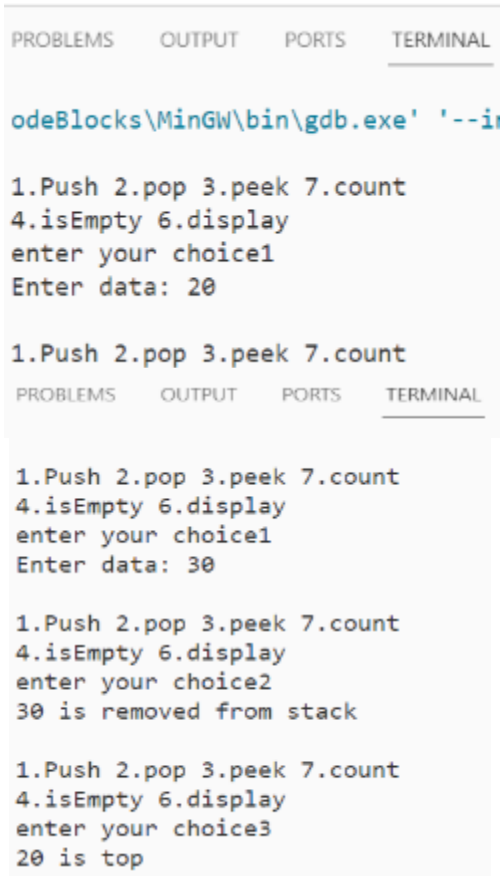
```
#include<stdio.h>
#include<stdlib.h>
struct node{
int data;
struct node *next;
};
struct node *top;
int count;
void push(){
    int value;
    struct node *temp;
    temp = (struct node*)malloc(sizeof(struct node));
    printf("Enter data: ");
    scanf("%d",&value);
    temp->data = value;
if(top==NULL)
    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 \n",top->data);
    top=NULL;
    free(p);
    count--;
}
}
void peek(){
if(top==NULL)
    printf("stack is empty\n");
else
    printf("%d is top ",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 2.pop 3.peek 7.count\n");
printf("4.isEmpty 5.isfull 6.display\n");
printf("enter your choice");
scanf("%d",&choice);
switch(choice){
case 1: push();
        break;
case 2: pop();
```

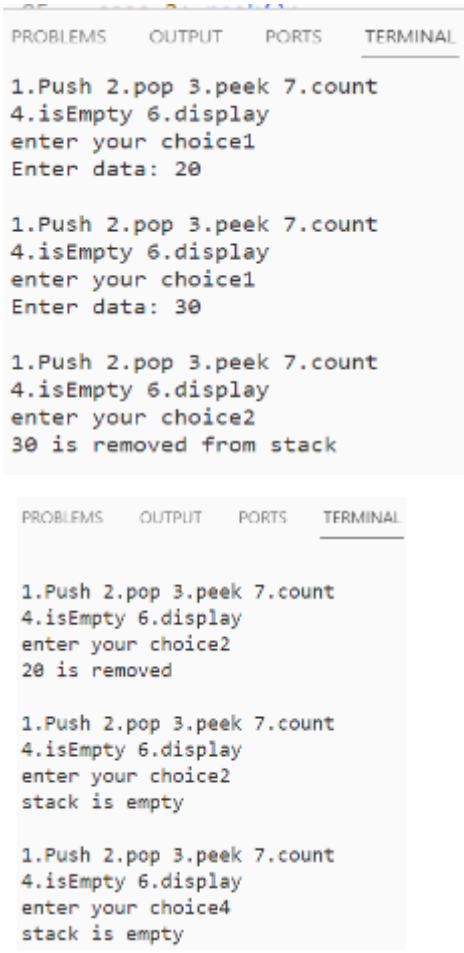
```
        break;
case 3: peek();
        break;
case 4: isEmpty();
        break;
case 6: display();
        break;
case 7: total();
        break;
default: exit(0);
}
}
}
```

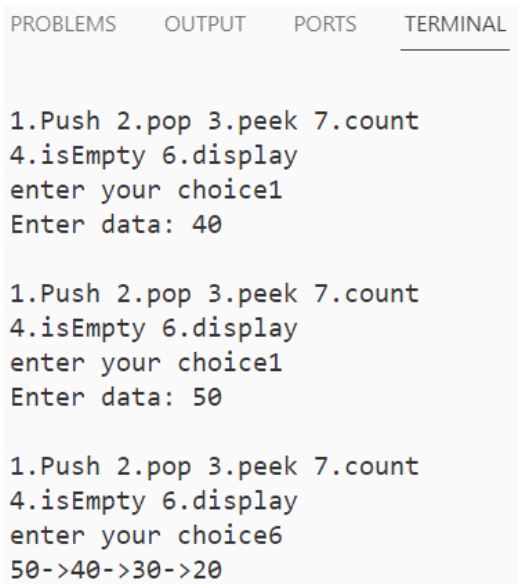
ScreenShots:

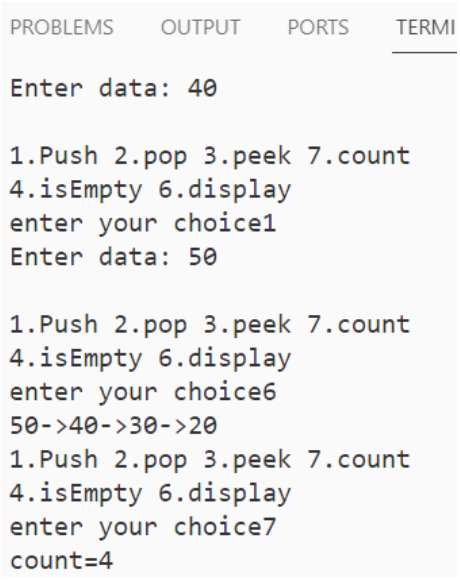
1.Push



2.Pop




PROBLEMS OUTPUT PORTS TERMINAL
1.Push 2.pop 3.peek 7.count
4.isEmpty 6.display
enter your choice1
Enter data: 40
1.Push 2.pop 3.peek 7.count
4.isEmpty 6.display
enter your choice1
Enter data: 50
1.Push 2.pop 3.peek 7.count
4.isEmpty 6.display
enter your choice6
50->40->30->20


PROBLEMS OUTPUT PORTS TERMI
Enter data: 40
1.Push 2.pop 3.peek 7.count
4.isEmpty 6.display
enter your choice1
Enter data: 50
1.Push 2.pop 3.peek 7.count
4.isEmpty 6.display
enter your choice6
50->40->30->20
1.Push 2.pop 3.peek 7.count
4.isEmpty 6.display
enter your choice7
count=4

6.STACK APPLICATIONS date:22/09/2023

Balancing of parathesis:

```
#include <stdio.h>
#include <stdlib.h>
struct sNode {
	char data;
	struct sNode* next;
};
void push(struct sNode** top_ref, int new_data);
int pop(struct sNode** top_ref);
int isMatchingPair(char character1, char character2)
{
	if (character1 == '(' && character2 == ')')
		return 1;
	else if (character1 == '{' && character2 == '}')
		return 1;
	else if (character1 == '[' && character2 == ']')
		return 1;
	else
		return 0;
}

// Return 1 if expression has balanced Brackets
int areBracketsBalanced(char exp[])
{
	int i = 0;

	// Declare an empty character stack
	struct sNode* stack = NULL;

	// Traverse the given expression to check matching
	// brackets
	while (exp[i]) {
		// If the exp[i] is a starting bracket then push
		// it
		if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
			push(&stack, exp[i]);

		// If exp[i] is an ending bracket then pop from
		// stack and check if the popped bracket is a
		// matching pair*/
		if (exp[i] == '}' || exp[i] == ')'|| exp[i] == ']') {

			// If we see an ending bracket without a pair
			// then return false
			if (stack == NULL)
				return 0;

			// Pop the top element from stack, if it is not
			// a pair bracket of character then there is a
			// mismatch.
			// his happens for expressions like {(})
```

```
                        else if (!isMatchingPair(pop(&stack), exp[i]))
                                return 0;
                }
                i++;
        }

        // If there is something left in expression then there
        // is a starting bracket without a closing
        // bracket
        if (stack == NULL)
                return 1; // balanced
        else
                return 0; // not balanced
}

// Driver code
int main()
{
        char exp[100] = "{()}{([{}])}[]";

        // Function call
        if (areBracketsBalanced(exp))
                printf("Balanced \n");
        else
                printf("Not Balanced \n");
        return 0;
}

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data)
{
        // allocate node
        struct sNode* new_node
                = (struct sNode*)malloc(sizeof(struct sNode));

        if (new_node == NULL) {
                printf("Stack overflow n");
                getchar();
                exit(0);
        }

        // put in the data
        new_node->data = new_data;

        // link the old list of the new node
        new_node->next = (*top_ref);

        // move the head to point to the new node
        (*top_ref) = new_node;
}

// Function to pop an item from stack
int pop(struct sNode** top_ref)
{
```

```
        char res;
        struct sNode* top;

        // If stack is empty then error
        if (*top_ref == NULL) {
                printf("Stack overflow n");
                getchar();
                exit(0);
        }
        else {
                top = *top_ref;
                res = top->data;
                *top_ref = top->next;
                free(top);
                return res;
        }
}
```

22501A0545

Infix to postfix:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
int precedence(char operator)
{
	switch (operator) {
	case '+':
	case '-':
		return 1;
	case '*':
	case '/':
		return 2;
	case '^':
		return 3;
	default://FOR NO OPERATOR
		return -1;
	}
}
int isOperator(char ch)
{
	return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}
char* infixToPostfix(char* infix)
{
	int i, j;
	int len = strlen(infix);
	char* postfix = (char*)malloc(sizeof(char) * (len + 2));
	char stack[MAX];//OPERATOR STACK
	int top = -1;
	for (i = 0, j = 0; i < len; i++) {
		if (infix[i] == ' ' || infix[i] == '\t')
			continue;
		if (isalnum(infix[i])) {
			postfix[j++] = infix[i];
		}
		else if (infix[i] == '(') {
			stack[++top] = infix[i];
		}
		else if (infix[i] == ')') {
			while (top > -1 && stack[top] != '(')
				postfix[j++] = stack[top--];
			if (top > -1 && stack[top] != '(')
				return "Invalid Expression";
			else
				top--;
		}
		else if (isOperator(infix[i])) {
			while (top > -1 && (precedence(stack[top]) >= precedence(infix[i])))
				postfix[j++] = stack[top--];
			stack[++top] = infix[i];
		}
	}
```

```
        while (top > -1) {
                if (stack[top] == '(') {
                        return "Invalid Expression";
                }
                postfix[j++] = stack[top--];
        }
        postfix[j] = '\0';
        return postfix;
}
int main()
{
        char infix[MAX] = "a+b*c+d";
        char* postfix = infixToPostfix(infix);
        printf("%s\n", postfix);
        free(postfix);
        return 0;
}
```

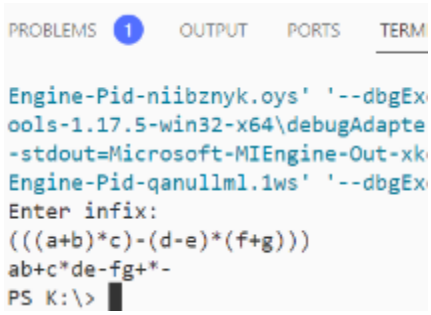
Evaluation of postfix:

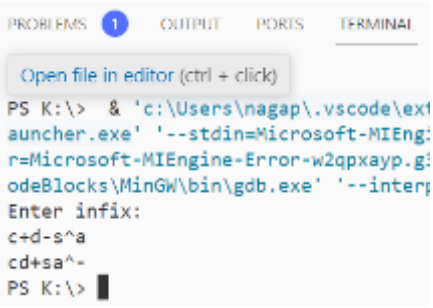
```
// C program to evaluate value of a postfix expression
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Stack type
struct Stack {
	int top;
	unsigned capacity;
	int* array;
};

// Stack Operations
struct Stack* createStack(unsigned capacity)
{
	struct Stack* stack
		= (struct Stack*)malloc(sizeof(struct Stack));

	if (!stack)
		return NULL;

	stack->top = -1;
	stack->capacity = capacity;
	stack->array
		= (int*)malloc(stack->capacity * sizeof(int));

	if (!stack->array)
		return NULL;

	return stack;
}

int isEmpty(struct Stack* stack)
{
	return stack->top == -1;
}

char peek(struct Stack* stack)
{
	return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
	if (!isEmpty(stack))
		return stack->array[stack->top--];
	return '$';
}

void push(struct Stack* stack, char op)
{
	stack->array[++stack->top] = op;
```

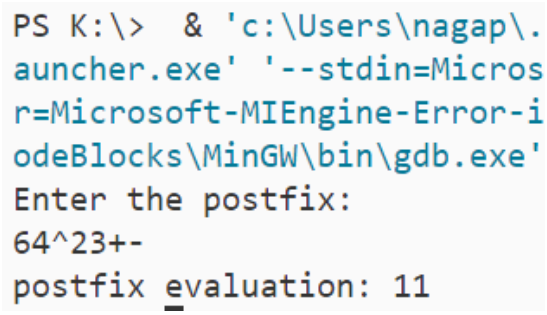
```
}

// The main function that returns value
// of a given postfix expression
int evaluatePostfix(char* exp)
{
	// Create a stack of capacity equal to expression size
	struct Stack* stack = createStack(strlen(exp));
	int i;

	// See if stack was created successfully
	if (!stack)
		return -1;

	// Scan all characters one by one
	for (i = 0; exp[i]; ++i) {

		// If the scanned character is an operand
		// (number here), push it to the stack.
		if (isdigit(exp[i]))
			push(stack, exp[i] - '0');

		// If the scanned character is an operator,
		// pop two elements from stack apply the operator
		else {
			int val1 = pop(stack);
			int val2 = pop(stack);
			switch (exp[i]) {
			case '+':
				push(stack, val2 + val1);
				break;
			case '-':
				push(stack, val2 - val1);
				break;
			case '*':
				push(stack, val2 * val1);
				break;
			case '/':
				push(stack, val2 / val1);
				break;
			}
		}
	}
	return pop(stack);
}

// Driver code
int main()
{
	char exp[] = "231*+9-";

	// Function call
	printf("postfix evaluation: %d", evaluatePostfix(exp));
	return 0;}
```

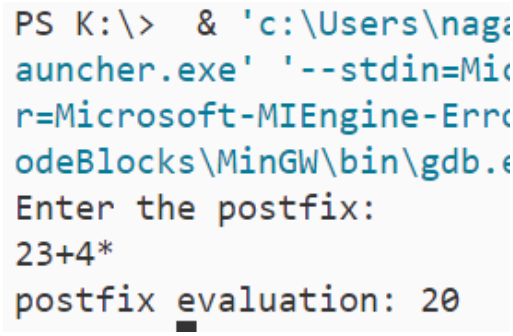
22501A0545

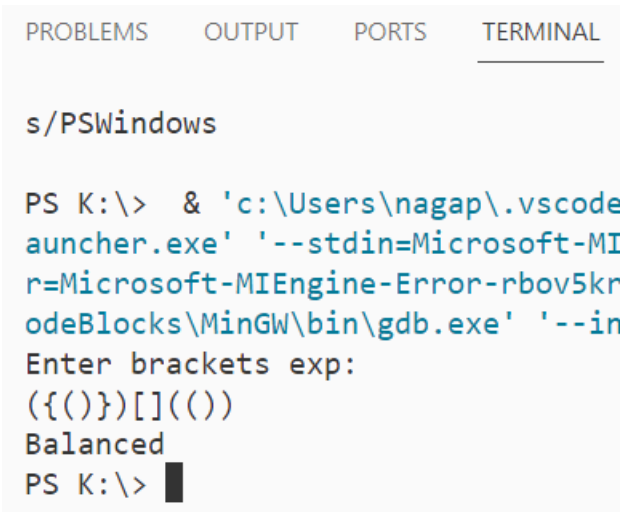
ScreenShot:

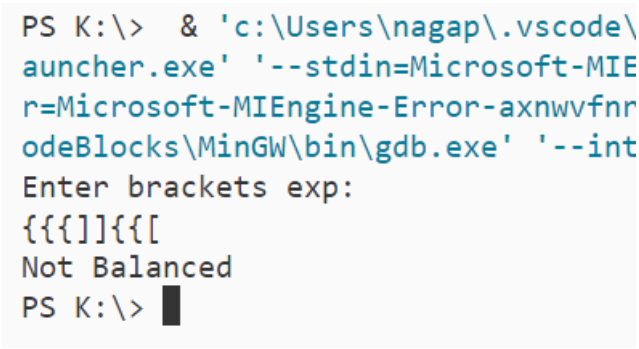









PROBLEMS
OUTPUT
PORTS
TERMINAL
s/PSWindows
PS K:\> & 'c:\Users\nagap\.vscode
auncher.exe' '--stdin=Microsoft-MI
r=Microsoft-MIEngine-Error-rbov5kr
odeBlocks\MinGW\bin\gdb.exe' '--in
Enter brackets exp:
({()})[](())
Balanced
PS K:\>


PS K:\> & 'c:\Users\nagap\.vscode\
auncher.exe' '--stdin=Microsoft-MIE
r=Microsoft-MIEngine-Error-axnwvfnr
odeBlocks\MinGW\bin\gdb.exe' '--int
Enter brackets exp:
{{{]]{{[
Not Balanced
PS K:\>

22501A0545

Date:13/10/23

7. Queues

Description:

A queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence. The operations of a queue make it a first-in-first-out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. A queue is an example of a linear data structure, or more abstractly a sequential collection.

Example:

Front <- 10	20	30	40	50 <- rear

Advantages:

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

Disadvatages:

1. The operations such as insertion and deletion of elements from the middle are time consuming.
2. Limited Space.
3. In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
4. Searching an element takes O(N) time.

Applications:

1. Multi programming: Multi programming means when multiple programs are running in the main memory.
2. Network: In a network, a queue is used in devices such as a router or a switch.
3. 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.
4. ATM
5. Ticket Counter Line
6. Key press sequence on the keyboard

B. Operations on Queue using arrays Algorithms:

1.insert

a) if (rear == size - 1)
b) Display queue is in overflow condition
c) if ((front == -1) and (rear = -1))
d) front = rear = 0
e) else

f) set rear = rear + 1
g) set queue [rear] = element

Example: front=rear=-1 insert element 10

Front <- 10 Rear<-				

Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

2. Delete

a) if ((front == -1) or (front > rear))
b) write *queue is in underflow*
c) else
d) set element = queue [front]
e) set front = front + 1

Example :

Front <- 10 Rear<-				

Front=rear=-1
10 removed

Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

3.Size

a) Check if the queue is empty display empty
b) Else display the value of = rear – front +1

Example:

Front <- 10	20	30	40	50 <- rear

The no of elements are 5
Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

4. Display

a) Check if the queue is empty if it is display empty
b) Else traverse the queue using a loop upto the rear and print the elements

Example:

Front <- 10	20	30	40	50 <- rear

The elements in the queue are
Front<- 10 20 30 40 50 <-rear

Time Complexity:

Best case: O(1) Worst Case: O(n) Average Case: O(n)

5. front

a) Check if the queue is empty if it is display empty
b) Else print the value in the index of front

Example:

Front <- 10	20	30	40	50 <- rear

The front element is 10

Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

6. rear

a) Check if the queue is empty if it is display empty
b) Else print the value in the index of rear

Example:

Front <- 10	20	30	40	50 <- rear

The rear element is 50

Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

7. IsEmpty

a) If rear and front are -1 then print the queue is empty
b) Else print queue is not empty

Example:

Front <- 10	20	30	40	50 <- rear

The queue is not empty
Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

8. IsFull

a) If rear is size -1 then print the queue is full
b) Else print queue is not empty

Example:

Front <- 10	20	30	40	50 <- rear

The queue is full
Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

C. Program on queues using arrays:

```
#include<stdio.h>
#include<stdlib.h>
int f=-1,r=-1,size,choice;
int *queue;
void enqueue(){
    int val;
    printf("Enter value: \n");
    scanf("%d",&val);
    if(isEmpty()){
        queue[++f] = val;
        r++;
    }
    else
        queue[++r] = val;
}
void dequeue(){
    if(f==-1 && r==-1) printf("Queue is empty\n");
    else
        printf("removed: %d\n",queue[f++]);
}
void display(){
    if(isEmpty()) printf("Queue is empty\n");
    else{
        printf("<- f");
        for(int i=f;i<=r;i++){
            printf(" %d ",queue[i]);
        }
        printf("r <-\n");
    }
}
int isEmpty(){
    if(f==-1 && r==-1) return 1;
    else return 0;
}
int isFull(){
    if(r==size-1) return 1;
    else return 0;
}
void front(){
    if(isEmpty()) printf("Queue is empty\n");
    else printf("Front: %d",queue[f]);
}
void rear(){
    if(isEmpty()) printf("Queue is empty\n");
    else printf("Rear: %d",queue[r]);
}
int main(){
```

```
    printf("Enter size of Queue: \n");
    scanf("%d",&size);
    queue = (int*)malloc(size*sizeof(int));
    while(1){
        printf("1.enqueue 2.dequeue 3.display\n");
        printf("4.isEmpty,5.isFull,6.front,7.rear \n");
        printf("8.no of elements,default exit\n");
        printf("Enter choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                if(r==size-1){
                    printf("Queue is full\n");
                    continue;
                }else enqueue();
            break;
            case 2: dequeue();
            break;
            case 3: display();
            break;
            case 4:
                if(isEmpty()) printf("\nqueue is empty");
                else printf("\nQueue is not empty");
                break;
            case 5:
                if(isFull()) printf("\nqueue is full");
                else printf("\nQueue is not Full");
                break;
            case 6: front();
            break;
            case 7: rear();
            break;
            case 8:
                printf("No of elements: %d\n",r-f+1);
                break;
            default: exit(0);
        }
    }
}
```

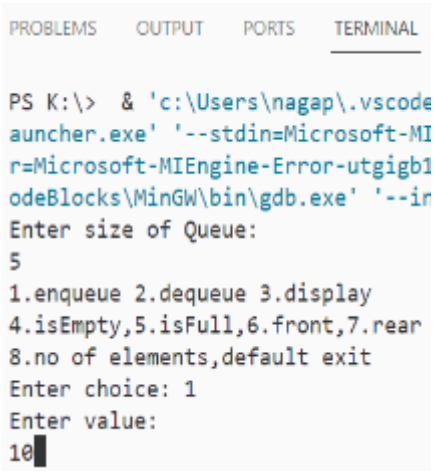
Output:

```
Enter size of Queue:
5
1.enqueue 2.dequeue 3.display
4.isEmpty,5.isFull,6.front,7.rear
8.no of elements,default exit
Enter choice:
```

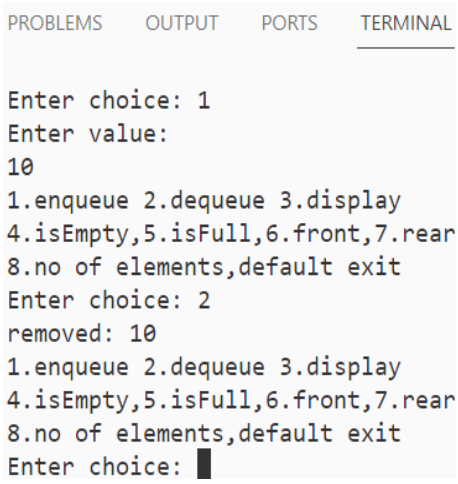
22501A0545

Screenshots:

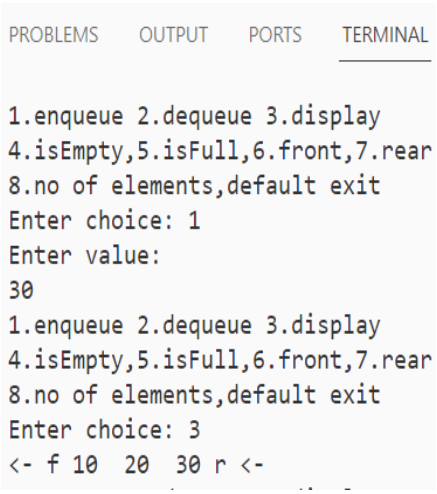
1.Enqueue



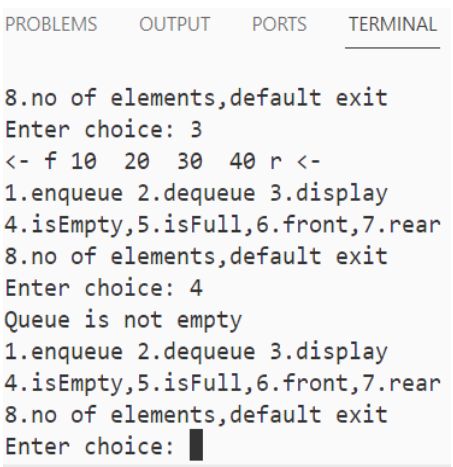
2.Dequeue



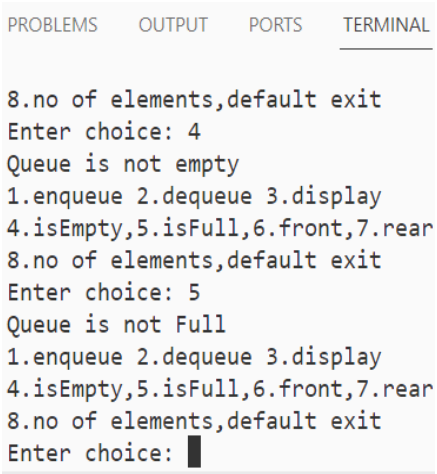
3.Display



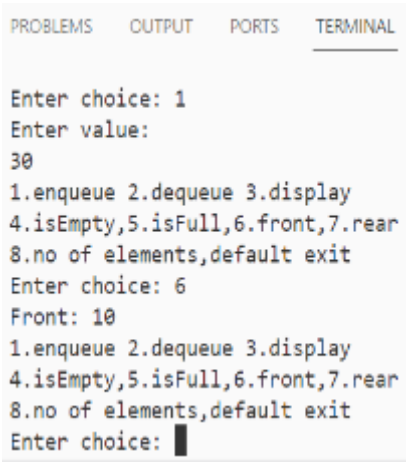
4.isEmpty



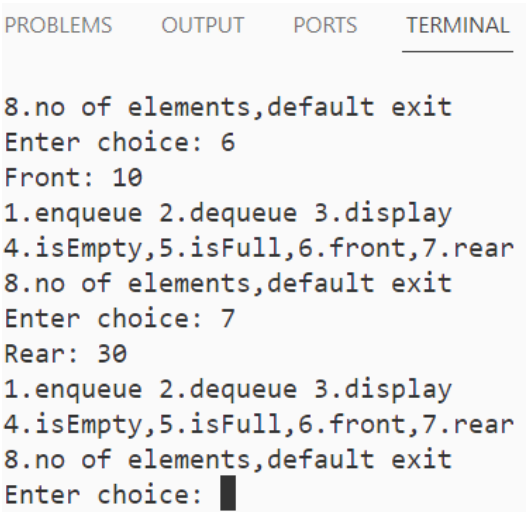
5.isFull



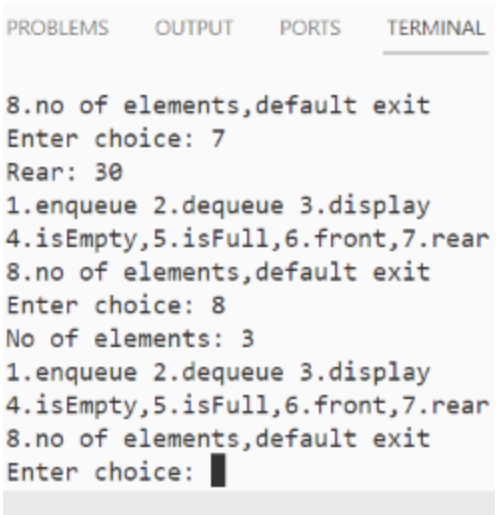
6.front



7.rear



8.Size

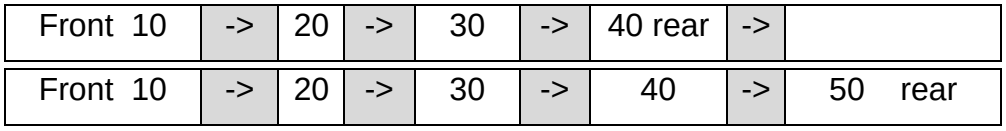


D. Operations on QUEUES using Linked List ,pseudo code

1. Insert

```
Set temp->data = val;
temp->next =NULL;
if(f==NULL && r==NULL){ //inserting first element
   r=temp;
   f=temp;
}
else{
   r->next = temp;
   r = temp;
}
```

Example: insert 50



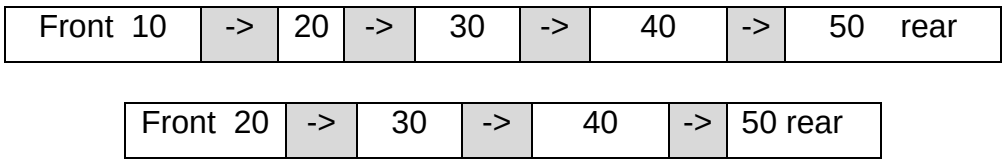
Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

2.Delete

```
if(r==NULL && f==NULL) printf("Queue is empty\n");
else if(r==f)
   print removed r->data
   struct node* temp = f;
   r=NULL;   f=NULL;
   delet temp;
else
   Print  removed  f->data
   struct node* temp = f;
   f=f->next;
   delete temp
```

Example: delete



Time Complexity:

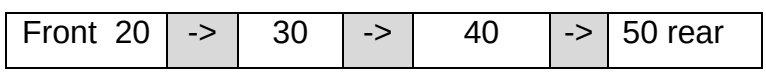
Best case: O(1) Worst Case: O(1) Average Case: O(1)

3.Size

Check if the queue is empty print empty is it is

Else Print No of elements: count

Example:



No of elements are 4

Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

4.Display:

If isEmpty print Queue is empty
else
struct node* temp=f
while(temp->next!=NULL){
print temp->data
temp=temp->next;
}
Print temp->data

Example:

Front 20	->	30	->	40	->	50 rear

The elements in the queue are f-> 20 30 40 50 ->rear

Time Complexity:

Best case: O(1) Worst Case: O(n) Average Case: O(n)

5. FRONT

Check If the queue is empty print queue is empty
Print the data in front i.e., front->data

Example:

Front 20	->	30	->	40	->	50 rear

The front value is 20

Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

6. rear

Check If the queue is empty print queue is empty
Print the data in rear i.e., rear->data

Example:

Front 20	->	30	->	40	->	50 rear

The front value is 50

Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

7. IsEmpty

If the front and rear are NULL then print the queue is empty
Else print queue is not empty

Example:

Front 20	->	30	->	40	->	50 rear

Not empty

Time Complexity:

Best case: O(1) Worst Case: O(1) Average Case: O(1)

E. Programs on queues using linked lists:

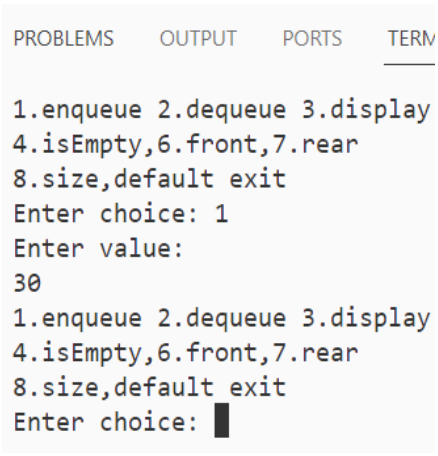
```
#include<stdio.h>
#include<stdlib.h>
int count=0,choice;
struct node{
    int data;
    struct node* next;
};
struct node* f=NULL;
struct node* r=NULL;
void enqueue(){
    int val;
    printf("Enter value: \n");
    scanf("%d",&val);
    struct node *temp = (struct node*)malloc(sizeof(struct node));
    temp->data = val;
    temp->next =NULL;
    if(f==NULL && r==NULL){
        r=temp;
        f=temp;
    }
    else{
        r->next = temp;
        r = temp;
    }count++;
}
void dequeue(){
    if(r==NULL && f==NULL) printf("Queue is empty\n");
    else if(r==f){//ONLY ONE ELEMNT
        printf("removed: %d\n",r->data);
        struct node* temp = f;
        r=NULL;
        f=NULL;
        free(temp);
        count--;
    }else{
        printf("removed: %d\n",f->data);
        struct node* temp = f;
        f=f->next;
        free(temp);
        count--;
    }
}
void display(){
    if(isEmpty()) printf("Queue is empty\n");
    else{
        struct node* temp=f;
```

```
        printf("<- f");
        while(temp->next!=NULL){
            printf(" %d ",temp->data);
            temp=temp->next;
        }
        printf("%d r <-\n",temp->data);
    }
}

int isEmpty(){
    if(r ==NULL && f==NULL) return 1;
    else return 0;
}

void front(){
    if(isEmpty()) printf("Queue is empty\n");
    else printf("Front: %d\n",f->data);
}
void rear(){
    if(isEmpty()) printf("Queue is empty\n");
    else printf("Rear: %d\n",r->data);
}
int main(){
    while(1){
        printf("1.enqueue 2.dequeue 3.display\n");
        printf("4.isEmpty,6.front,7.rear \n");
        printf("8.size,default exit\n");
        printf("Enter choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: enqueue();
            break;
            case 2: dequeue();
            break;
            case 3: display();
            break;
            case 4:
                if(isEmpty()) printf("queue is empty\n");
                else printf("Queue is not empty\n");
                break;
            case 6: front();
            break;
            case 7: rear();
            break;
            case 8:
                printf("No of elements: %d\n",count);
                break;
            default: exit(0);
        } } }
```

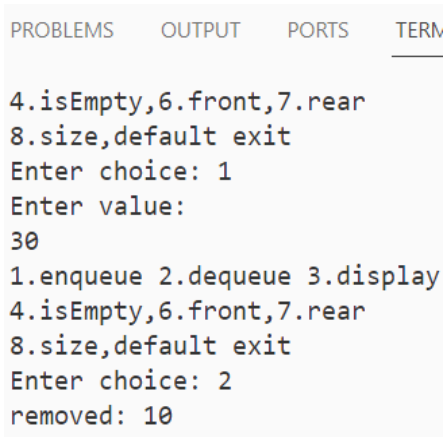
22501A0545

Screenshots:

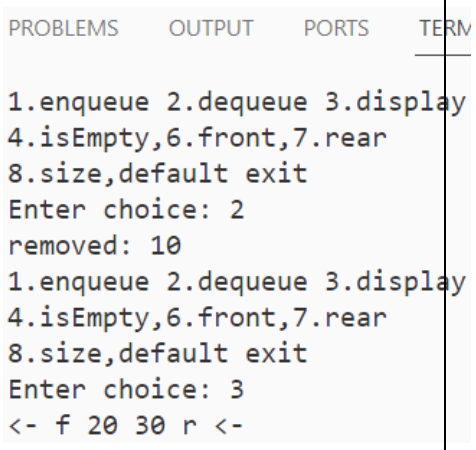
1.Enqueue



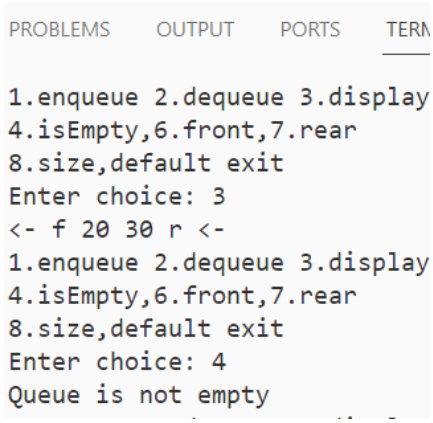
2.dequeue



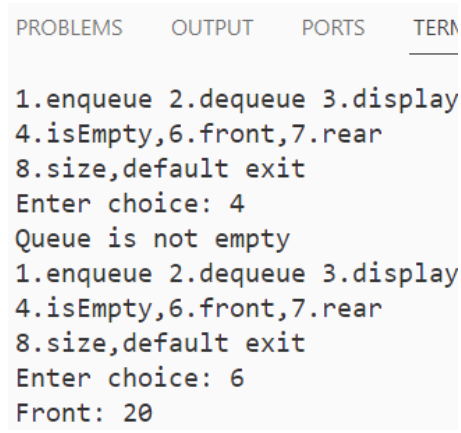
3.Display



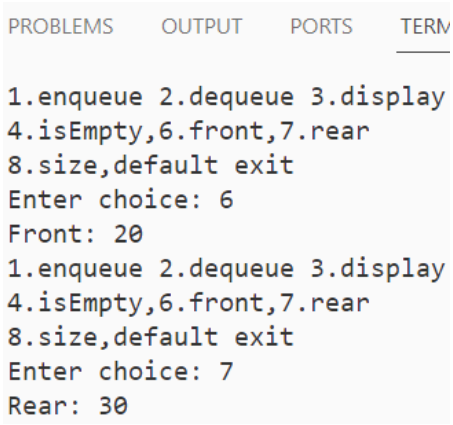
4.isEmpty



5.front



6.rear



7.size

PROBLEMS OUTPUT PORTS TERM

```
4.isEmpty,6.front,7.rear
8.size,default exit
Enter choice: 7
Rear: 30
1.enqueue 2.dequeue 3.display
4.isEmpty,6.front,7.rear
8.size,default exit
Enter choice: 8
No of elements: 2
```

F. Operations on CIRCULAR QUEUES using Arrays, pseudo code

1.insert

h) if (rear == size - 1)
i) Display queue is in overflow condition
j) if queue is empty
k) queue[++f] = val;
l) r++;
m) set rear = rear + 1 %size
n) set queue [rear] = element

Example: front=rear=-1 insert element 10

Front <- 10 Rear<-				

Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

2. Delete

f) if ((front == -1) or (front > rear))
g) write *queue is in underflow*
h) else if front==rear set front=-1 and rear=-1
i) else if front=size-1 front++
j) set front = front + 1

Example :

Front <- 10 Rear<-				

Front=rear=-1
10 removed

Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

3.Size

c) Check if the queue is empty display empty
d) Else display the value of = rear – front +1

Example:

Front <- 10	20	30	40	50 <- rear

The no of elements are 5
Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

4. Display

c) Check if the queue is empty if it is display empty
d) Else traverse the queue using a loop upto the rear and print the elements

Example:

Front <- 10	20	30	40	50 <- rear

The elements in the queue are
Front<- 10 20 30 40 50 <-rear

Time Complexity:
Best case: O(1) Worst Case: O(n) Average Case: O(n)

5. front

c) Check if the queue is empty if it is display empty
d) Else print the value in the index of front

Example:

Front <- 10	20	30	40	50 <- rear

The front element is 10

Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

6. rear

c) Check if the queue is empty if it is display empty
d) Else print the value in the index of rear

Example:

Front <- 10	20	30	40	50 <- rear

The rear element is 50

Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

7. IsEmpty

c) If rear and front are -1 then print the queue is empty
d) Else print queue is not empty

Example:

Front <- 10	20	30	40	50 <- rear

The queue is not empty
Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

8. IsFull

c) If front = (rear+1)%size then print the queue is full
d) Else print queue is not empty

Example:

Front <- 10	20	30	40	50 <- rear

The queue is full
Time Complexity:
Best case: O(1) Worst Case: O(1) Average Case: O(1)

Circular Queues:

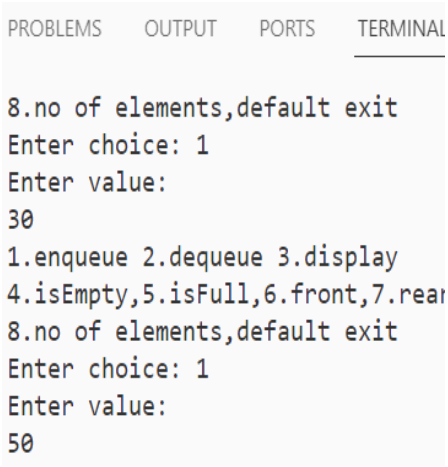
```
#include<stdio.h>
#include<stdlib.h>

int f=-1,r=-1,size,choice;
int *queue;

void enqueue(){
    int val;
    printf("Enter value: \n");
    scanf("%d",&val);
    if(isEmpty()){
        queue[++f] = val;
        r++;
    }
    else{
        r = (r+1)%size;
        queue[r] = val;
    }
}
void dequeue(){
    if(f==-1) printf("Underflow\n");
    if(f==r){
            printf("removed: %d\n",queue[f]);
        f =-1;
        r=-1;
    }else if(f==size-1){
        printf("removed: %d\n",queue[f]);
        f=0;
    }
    else {
        printf("removed: %d\n",queue[f]);
        f++;
    }
}
void display(){
    if(isEmpty()) printf("Queue is empty\n");
    else{
        printf("<- f");
        for(int i=f;i<=r;i++){
            printf(" %d ",queue[i]);
        }
        printf("r <-\n");
    }
}

int isEmpty(){
    if(f==-1 && r==-1) return 1;
```

```
    else return 0;
}
int isFull(){
    if((r+1)%size == f) return 1;
    else return 0;
}
void front(){
    if(isEmpty()) printf("Queue is empty\n");
    else printf("Front: %d\n",queue[f]);
}
void rear(){
    if(isEmpty()) printf("Queue is empty\n");
    else printf("Rear: %d\n",queue[r]);
}
int main(){
    printf("Enter size of Queue: \n");
    scanf("%d",&size);
    queue = (int*)malloc(size*sizeof(int));
    while(1){
        printf("1.enqueue 2.dequeue 3.display\n");
        printf("4.isEmpty,5.isFull,6.front,7.rear \n");
        printf("8.no of elements,default exit\n");
        printf("Enter choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                if(isFull()){
                    printf("Queue is full\n");
                    continue;
                }else enqueue();
            break;
            case 2: dequeue();
            break;
            case 3: display();
            break;
            case 4:
                if(isEmpty()) printf("queue is empty\n");
                else printf("Queue is not empty\n");
                break;
            case 5:
                if(isFull()) printf("queue is full\n");
                else printf("Queue is not Full\n");
                break;
            case 6: front();
            break;
            case 7: rear();
            break;
            case 8:
                printf("No of elements: %d\n",r-f+1);
```

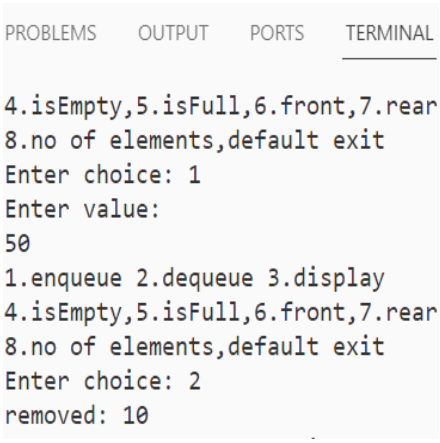
```
            break;
        default: exit(0);
      }
   }
}
```

Screenshots:

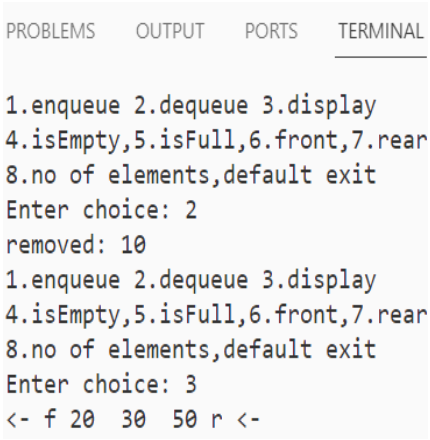
1.Enqueue



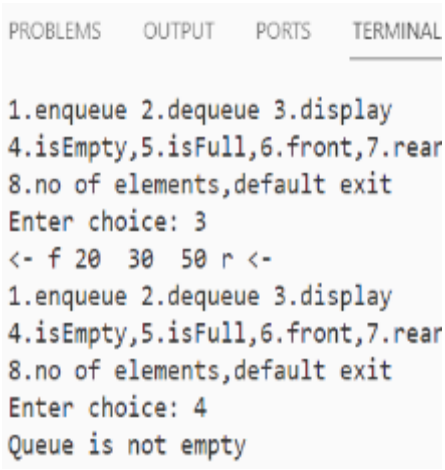
2.Dequeue



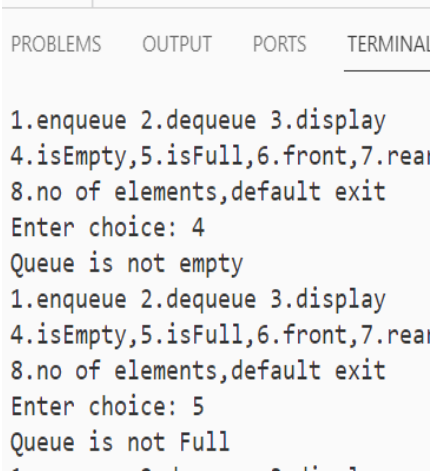
3.Display



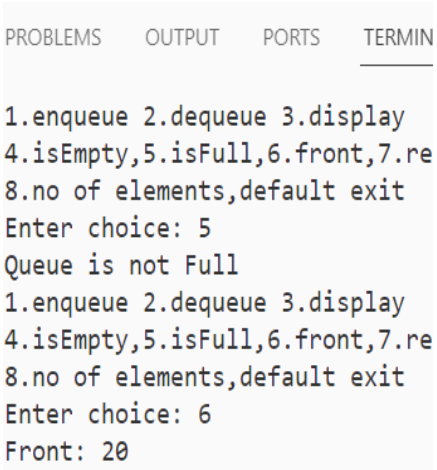
4.isEmpty

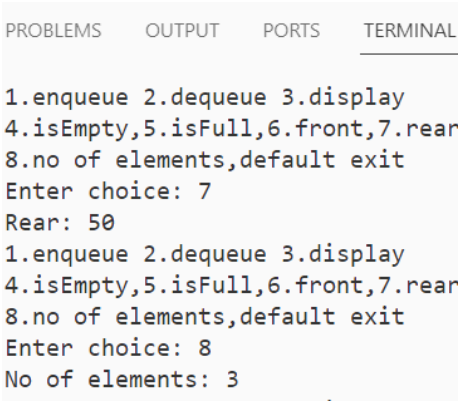


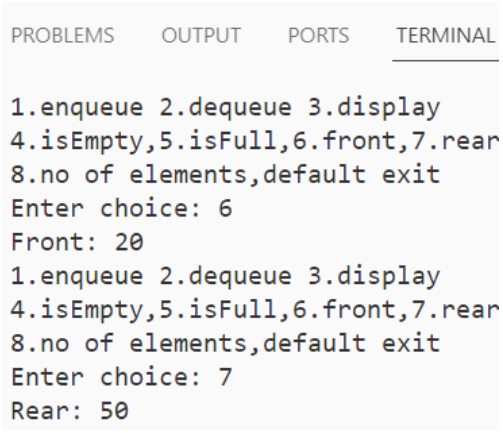
5.isFull



6.front







8.APPLICATIONS OF QUEUES AND STACKS date: 16/10/23

A. Problem-2:

```
#include<stdio.h>
int main(){
    int n,k;
    printf("Enter the value of n \n");
    scanf("%d",&n);
    printf("Enter the value of k \n");
    scanf("%d",&k);
    printf("Enter Elements \n");
    int arr[n];
    for(int i=0; i<n; i++){
        scanf("%d",&arr[i]);
    }
    int sum = 0,ans=0;
    for(int i=0; i<k; i++){
        sum = sum + arr[i];
    }
    ans=sum;
    for(int i=0; i<k-1; i++){
        sum = sum + arr[n-i-1] - arr[k-1-i];
        if(sum>=ans)
        ans=sum;
    }
    printf("the max sum is %d",ans);
    return 0;
}
```

Output:

```
Enter the value of n
10
Enter the value of k
5
Enter Elements
1 2 3 4 6 5 89 7 5 4
the max sum is 106
```

Screenshots:

Output

```
/tmp/G91mNeDJ8V.o
Enter the value of n
10
Enter the value of k
5
Enter Elements
10 9 1 2 3 4 5 6 7 8
the max sum is 40
```

Output

```
/tmp/G91mNeDJ8V.o
Enter the value of n
5
Enter the value of k
2
Enter Elements
12 45 82 -1 65
the max sum is 77
```

22501A0545

B. Problem – 3:

```
#include <stdio.h>
int main() {
    int n;
    printf("Enter size: \n");
    scanf("%d", &n);
    int a[n];
    printf("Enter elements\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    int st[n];
    int top = -1;
    int ans[n];
    for (int i = 0; i < n; i++) {
        ans[i] = a[i];
    }
    for (int i = n - 1; i >= 0; i--) {
        while (top > -1 && a[st[top]] < a[i]) {
            top--;
        }
        if (top > -1) {
            ans[i] ^= ans[st[top]];
        }
        top++;
        st[top] = i;
    }
    int max = ans[0];
    for (int i = 1; i < n; i++) {
        if (ans[i] > max) {
            max = ans[i];
        }
    }
    printf("the result is %d\n", max);
    return 0;
}
```

Outputs:

Enter size:

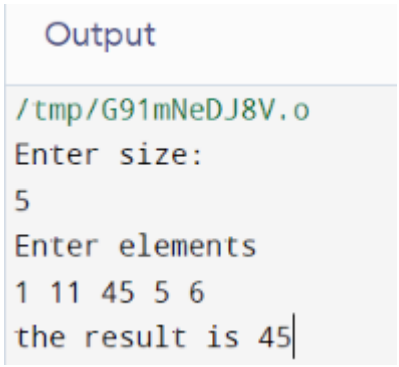
5

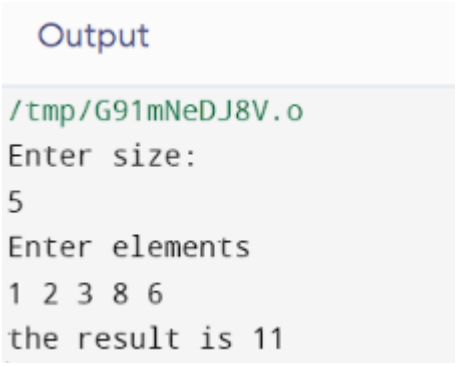
Enter elements

1 11 45 5 6

22501A0545

the result is 45
Screenshots:





22501A0545

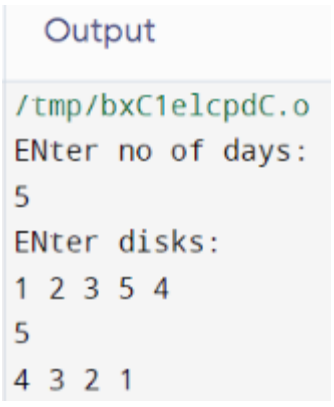
C. Problem-6

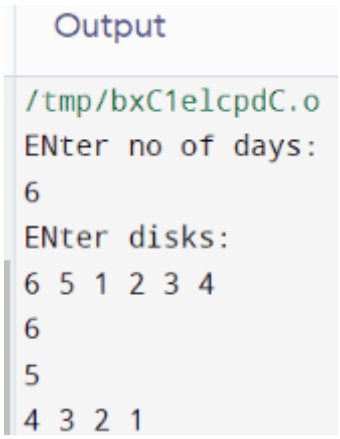
```
#include<stdio.h>
int main()
{
    int disk, temp[100001] = {0};
    printf("ENter no of days: \n");
    scanf("%d", &disk);
    int min = disk, size = disk;
    int q;
    printf("ENter disks: \n");
    for (int i = 0; i < disk; i++)
    {
        scanf("%d", &q);
        temp[q] = q;
        if(q == min)
            {
            while(temp[size])
            {
                printf("%d ", size);
                size--;
            }
            min = size;
        printf("\n");
    }
}
}
```

Outputs:

ENter no of days:
6
ENter disks:
5 4 2 3 1 6
6 5 4 3 2 1

Screenshots





22501A0545

D. problem 7

```
#include <stdio.h>
int main(){
    int num;
    printf("enter size: \n");
    scanf("%d", &num);
    int *arr1,*arr2;
    arr1=(int*)malloc(sizeof(int)*num);
    arr2=(int*)malloc(sizeof(int)*num);
    printf("enter array1 \n");
    for(int i=0;i<num;i++)
    scanf("%d",&arr1[i]);
    printf("enter array2 \n");
    for(int i=0;i<num;i++)
    scanf("%d",&arr2[i]);
    int time=0;
    int p=0;//pointer
    for(int i=0;i<num;i++)
    {
        while(arr1[p]!=arr2[i])
        {
            if(arr1[p]!=0)
            time++ ;
            p=(p+1)%num;
        }
        arr1[p]=0;
        time++;
    }
    printf("the time required is %d",time);
}
```

Output:

```
enter size:
3
enter array1
1 2 3
enter array2
3 2 1
the time required is 6
```

Screenshots:

Output

```
/tmp/v8pN5hoA6V.o
enter size:
5
enter array1
1 2 3 4 5
enter array2
1 3 2 4 5
the time required is 8
```

Output

```
/tmp/v8pN5hoA6V.o
enter size:
3
enter array1
1 2 3
enter array2
3 2 1
the time required is 6
```

9. Binary Trees

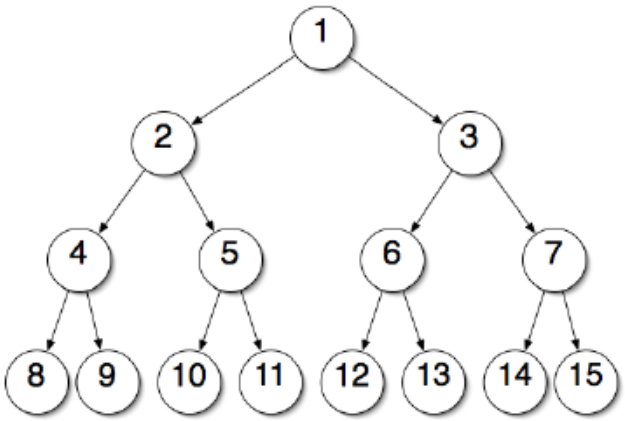
Date:3/11/23

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. A Binary tree is represented by a pointer to the topmost node (commonly known as the “root”) of the tree. If the tree is empty, then the value of the root is NULL. Each node of a Binary Tree contains the following parts:

Data Pointer to left child Pointer to right child

Example:



Advantages:

Efficient searching, Ordered traversal, Memory efficient, Fast insertion and deletion, Easy to implement, Useful for sorting

Disadvatages:

Limited structure, Unbalanced trees, Space inefficiency, Slow performance in worst-case scenarios.

Complex balancing algorithms

Applications:

Database systems, File system, Compression algorithm , Decision trees, Game AI

Algorithms for binary trees using arrays

1. preorder traversal
    1. check if the current value is null, if it is terminate recursion
    2. print the root value
    3. recursively call the function with left node value
    4. recursively call the function with left node value

Time complexity :- O(n)

2. post order traversal
    1. check if the current value is null, if it is terminate recursion
    2. recursively call the function with left node value
    3. recursively call the function with left node value
    4. print the root value

Time complexity :- O(n)

3. Inorder traversal

1. check if the current value is null, if it is terminate recursion
2. recursively call the function with left node value
3. print the root value
4. recursively call the function with left node value

Time complexity :- O(n)

4.Searching

1. Set found value to 0
2. Scan the required value to search in the binary tree
3. Traverse the binary tree until all the values in the tree are checked if found increment a found value
4. Else set found to 0

Time complexity :- O(n)

5.level order traversal

1. Traverse the binary tree until all the values in the tree are scaned
2. Print the value in the root
3. Stop if last node is reached

Time complexity :- O(n)

6.print leaf nodes

1. Traverse the tree level by level using level order traversal
2. Check if the node's left and right node values are out of bounds, if it is then don't traverse for the next node for the current node
3. Print if the condition is true

Time complexity :- O(n)

8.MINIMUM VALUE :-

1. set a global min value
2. check is the current root value is less than the min value update the min value
3. recursively call the function with root left passing until the passed value is not NULL
4. recursively call the function with root left passing until the passed value is not NULL

Time complexity :-O(n)

9.Maximum value:-

1. set a global Max value
2. check is the current root value is greater than the Max value update the min value
3. recursively call the function with root left passing until the passed value is not NULL
4. recursively call the function with root left passing until the passed value is not NULL

Time complexity :- O(n)

A.Program on binary trees using arrays

```
#include<stdio.h>
#include<limits.h>
int complete_node = 15;
int MAX = INT_MIN,MIN = INT_MAX;
char tree[15]="1234567\0\0\0\0\0\089" ;
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(" %c ",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(" %c ",tree[index]);
    }
}
void inorder(int index){
    if(index>=0 && tree[index]!='\0'){
        inorder(get_left_child(index));
        printf(" %c ",tree[index]);
        inorder(get_right_child(index));
    }
}
void levelorder(){
    int j;
    for(j=0;j<complete_node;j++){
        if(tree[j]!='\0' )
            printf(" %c ",tree[j]);
    }
}
void printleafs(){
    int j;
    for(j=0;j<complete_node;j++){
        if(is_leaf(j) && tree[j] != '\0')
            printf(" %c ",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);
}
void maxe(int index){
    int present = tree[index] - '0';
    if(index>=0 && tree[index]!='\0'){
        if(present > MAX) MAX = present;
        maxe(get_left_child(index));
        maxe(get_right_child(index));
    }
}
void mine(int index){
    int present = tree[index] - '0';
    if(index>=0 && tree[index]!='\0'){
        if(present < MIN) MIN = present;
        mine(get_left_child(index));
        mine(get_right_child(index));
    }
}
int Search(int val){
    int j,f=0;
    for(j=0;j<complete_node;j++){
        int v = tree[j] - '0';
        if(v == val )
            f = 1;
    }
    if(f==0)  printf("\nnot found");
    else printf("\nFound");
    return 0;
}
int main(){
    int choice,s;;
    while(1){
        printf("\n1.preorder 2.postorder 3.inorder\n");
        printf("5.leafs 6.height 7.min 8.max\n");
        printf("4.levelorder 9.search\n");
        printf("Enter your choice:\n");
      scanf("%d",&choice);
        switch(choice){
            case 1:printf("\nPreorder:\n");
                   preorder(0);break;
            case 2:printf("\nPostorder:\n");
                   postorder(0);break;
            case 3:printf("\nInorder:\n");
                   inorder(0);
```

```
                break;
        case 4:printf("\nLevelorder:\n");
                levelorder();break;
        case 5:printf("\nLeafs:\n");
                printleafs();break;
        case 6:printf("\nheight of the tree is:%d\n",get_height(0)+1);
                break;
        case 7:mine(0);printf("\nthe min of tree: %d",MIN);
                break;
        case 8:maxe(0);
                printf("\nthe min of tree: %d",MAX);break;
        case 9:printf("Enter search element: \n");
                scanf("%d",&s);
           Search(s);break;
        default: exit(0);
      }
   }
}
```

Output:

Enter your choice:

1.preorder 2.postorder 3.inorder 4.levelorder

5.leafs 6.height 7.min 8.max of tree

9.search

ScreenShots:

1.Preorder:

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
1

Preorder:
 1  2  4  5  3  6  7  8  9
```

2.postorder

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
2

Postorder:
 4  5  2  6  8  9  7  3  1
```

3.inorder

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
3

Inorder:
 4  2  5  1  6  3  8  7  9
```

5.leafs

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
5

Leafs:
 4  5  6  8  9
```

6.height

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
6

height of the tree is:4
```

7.min

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
7

the min of tree: 1
```

8.Max

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
8

the max of tree: 9
```

4.levelorder

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
4

Levelorder:
 1  2  3  4  5  6  7  8  9
```

9.search

```
1.preorder 2.postorder 3.inorder
5.leafs 6.height 7.min 8.max
4.levelorder 9.search
Enter your choice:
9
Enter search element:
4

Found
```

22501A0545

Algorithms for binary tree using linked list

1.PREORDER TRAVERSAL :-

ALGORITHM:-

1. Follow step 2 to 4 until root != NULL
2. Write root -> data
3. Preorder (root -> left)
4. Preorder (root -> right)

Time complexity:- O(n)

2.INORDER TRAVERSAL:-

1. Follow step 2 to 4 until root != NULL
2. Inorder (root -> left)
3. Write root -> data
4. Inorder (root -> right)

TIME COMPLEXITY:-O(n)

3.POSTORDER TRAVERSAL:-

1. Follow step 2 to 4 until root != NULL
2. Postorder (root -> left)
3. Postorder (root -> right)
4. Write root -> data

Time complexity :-O(N)

4.LEVEL ORDER :-

1. If the root is NULL, return.
2. Otherwise push the root in queue.
3. Pop the node from the queue.
4. Print the node’s data and add its left and right child.
5. Repeat until the queue is empty.

TIMECOMPLEXITY:-O(n)

5.LEAFNODE :-

1. Check if the given node is null. If null, then return from the function.
2. Check if it is a leaf node. If the node is a leaf node, then print its data.
3. If in the above step, the node is not a leaf node then check if the left and right children of node exist. If yes then call the function for left and right child of the node recursively.

TIME COMPLEXITY :-O(n)

6.HEIGHT OF TREE :-

1. If we find an empty root node, we will say that the height of the tree is 0.
2. Otherwise, we will find the height of the left subtree and right subtree recursively.
3. After finding the height of the left subtree and right subtree, we will calculate their maximum height.
4. We will add 1 to the maximum height. That will be the height of the binary tree.

TIME COMPLEXITY :-O(n)

7.SEARCH :-

1. Define Node class which has three attributes namely: data left and right. Here, left represents the left child of the node and right represents the right child of the node.
2. When a node is created, data will pass to data attribute of the node and both left and right will be set to null.
3. Define another class which has two attribute root and flag.
4. Root represents the root node of the tree and initializes it to null.
5. The Flag will be used to check whether the given node is present in the tree or not. Initially, it will be set to false.
6. searchNode() will search for a particular node in the binary tree:
7. It checks whether the root is null, which means the tree is empty.
8. If the tree is not empty, it will compare temp?s data with value. If they are equal, it will set the flag to true and return.
9. Traverse left subtree by calling searchNode() recursively and check whether the value is present in left subtree.
10. Traverse right subtree by calling searchNode() recursively and check whether the value is present in the right subtree.

Time complexity -O(n)

8.MINIMUM VALUE :-

1. set a global min value
2. check is the current root value is less than the min value update the min value
3. recursively call the function with root left passing until the passed value is not NULL
4. recursively call the function with root left passing until the passed value is not NULL

Time complexity :-O(n)

9.Maximum value:-

1. set a global Max value
2. check is the current root value is greater than the Max value update the min value
3. recursively call the function with root left passing until the passed value is not NULL
4. recursively call the function with root left passing until the passed value is not NULL

Time complexity :- O(n)

10.insert :-

1. Start with the root node of the binary tree.
2. If the root is NULL, create a new node with the given data and set it as the root node.
3. Otherwise, create a queue data structure to hold the nodes during traversal.
4. Enqueue the root node into the queue.
5. Repeat the following steps until the queue becomes empty:
    a. Dequeue a node from the front of the queue.
6. Check if the left child of the dequeued node is NULL.
7. If so, create a new node with the given data and set it as the left child.
8. If not, enqueue the left child into the queue.
9. Check if the right child of the dequeued node is NULL.
10. If so, create a new node with the given data and set it as the right child.
11. If not, enqueue the right child into the queue.
12. Once the queue is empty, the insertion process is complete.

Time complexity :-O(n)

11.delete :-

1. Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.

Time complexity :-O(n)

12.Height and depth of a particular node:-

1. Initialize height and depth variable with -1;
2. Initialize a queue and a level variable with 0 and push the root in the queue.
3. Perform level order traversal and if value of frontNode is equal to the target(K) node then value of depth will be equal to the level value and continue traversing.
4. After completion we can calculate the value of height using height = level – depth – 1;
5. Print the value of height and depth variable.

Time complexity :-O(n)

B.Program on Binary trees using linked list

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
#include <stdbool.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;
                else node->right = newNode;
                break;
            }
        }
    }
}
void printlevelorder(struct node* r){
    struct queue* queue = createQueue();
    enqueue(queue, r);
    while(queue->size != 0) {
        struct node *node = dequeue(queue);
        printf("%d ",node->data);
        if(node->left != NULL) enqueue(queue, node->left);
        if(node->right != NULL)enqueue(queue, node->right);
    }
}
void leafnodes(struct node* r){
    struct queue* queue = createQueue();
    enqueue(queue, r);
    while(queue->size != 0) {
        struct node *node = dequeue(queue);
        if(node->left == NULL && node->right == NULL) printf("%d ",node->data);
        if(node->left != NULL) enqueue(queue, node->left);
        if(node->right != NULL)enqueue(queue, node->right);
    }
}
void deletetree(struct node *r){
    if(r != NULL){
        deletetree(r->left);
        deletetree(r->right);
        free(r);
    }
}
```

```
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);

        }
}
int treeheight(struct node *Root){
    int leftheight,rightheight;
    if(Root == NULL) return 0;
    else{
        leftheight = treeheight(Root->left);
        rightheight = treeheight(Root->right);
        if(leftheight > rightheight) return leftheight+1;
        else return rightheight+1;
    }
}
void search(struct node* r,int val){
    int f=0;
    struct queue* queue = createQueue();
    enqueue(queue, r);
    while(queue->size != 0) {
        struct node *node = dequeue(queue);
        if(node->data == val) f++;
```

```
        if(node->left != NULL) enqueue(queue, node->left);
        if(node->right != NULL)enqueue(queue, node->right);
    }
    if(f==0) printf("not found\n");
    else printf("Found \n");
}
void Min(struct node* r){
    int min = r->data;
    struct queue* queue = createQueue();
    enqueue(queue, r);
    while(queue->size != 0) {
        struct node *node = dequeue(queue);
        if(node->data < min) min=node->data;
        if(node->left != NULL) enqueue(queue, node->left);
        if(node->right != NULL)enqueue(queue, node->right);
    }
    printf("\nminimum in tree: %d",min);
}
void Max(struct node* r){
    int max= r->data;
    struct queue* queue = createQueue();
    enqueue(queue, r);
    while(queue->size != 0) {
        struct node *node = dequeue(queue);
        if(node->data > max) max=node->data;
        if(node->left != NULL) enqueue(queue, node->left);
        if(node->right != NULL)enqueue(queue, node->right);
    }
    printf("\nmaximum in tree: %d",max);
}

void eachnode(struct node* r){
    int offheight = treeheight(root);
    struct queue* queue = createQueue();
    enqueue(queue, r);
    while(queue->size != 0) {
        struct node *node = dequeue(queue);
        int h = treeheight(node);
        printf("The height of %d is %d\n",node->data,h);
        if(node->left != NULL) enqueue(queue, node->left);
        if(node->right != NULL)enqueue(queue, node->right);
    }
}
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 = treeheight(node);
        printf("\nDepths: \n");
        for (int i = 0; i <= height; i++) {
            int breadth = 0;
            getBreadth(node, i, &breadth);
            if(breadth >= height ) 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,val,del;
    while(1){
      printf("\n1.Insert 2.Inorder 3.Preorder");
      printf("\n 4.Postorder 5.level order 6.height ");
      printf("\n7.delete tree 8.leafs 9.Search ");
```

```
      printf("\n10.min 11.max");
      printf("\nEnter 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("\n"); inorderTraversal(root);; break;
        case 3:printf("\n"); preorderTraversal(root); break;
        case 4:printf("\n"); postorderTraversal(root); break;
        case 5:printf("\n"); printlevelorder(root);break;
        case 6:printf("\n Height: %d",treeheight(root)); break;
         case 7:printf("\nENter node to be deleted: ");
         scanf("%d",&del);
             root = deleteNode(root,del); break;
         case 8: printf("\n");leafnodes(root);break;
         case 9:printf("enter val: ");
                scanf("%d",&val);
                search(root,val); break;
        case 10: Min(root); break;
        case 11: Max(root); break;
        case 12: eachnode(root);
        breadthOfEachNode(root);break;
    default: exit(0);
      }
   }
}
```

Output:

```
1.Insert 2.Inorder 3.Preorder
 4.Postorder 5.level order 6.height
7.delete tree 8.leafs 9.Search
10.min 11.max
Enter your choice:
```

22501A0545

Screenshots:

1.insert

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height.ght
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 3

1 2 3
```

2.inorder

```
1.Insert 2.Inorder 3.Preorder
1.Insert 2.Inorder 3.Preorder    ;ht
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 4

2 3 1
```

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 5

1 2 3
```

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 6

 Height: 2
```

```
ENter node to be deleted: 3

1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 2

2 1
```

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 8

2 3
```

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 9
enter val: 1
Found
```

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 10

minimum in tree: 1
```

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 11

maximum in tree: 3
```

```
1.Insert 2.Inorder 3.Preorder
4.Postorder 5.levelorder 6.height
7.delete node 8.leafs 9.Search
10.min 11.max
Enter your choice: 12
The height of 1 is 2
The height of 2 is 1
The height of 3 is 1

Depths:
Node at height 0: 1
Node at height 1: 1
Node at height 2: 0
```

10.Binary Search Trees Date: 06/11/23

AIM: Implement Binary search trees using Arrays and Linked

DEFINITION:

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

Applications of Binary Trees:

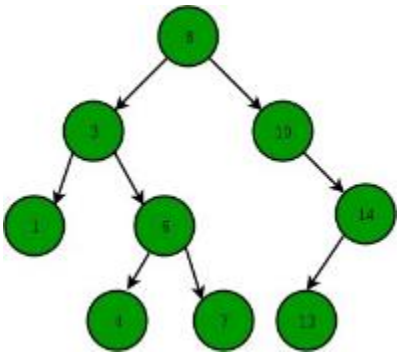
1. BSTs are used for indexing.
2. It is also used to implement various searching algorithms.
3. IT can be used to implement various data structures.
4. BSTs can be used in decision support systems to store and quickly retrieve data.

Advantages of Binary Tree:

1. BST is fast in insertion and deletion when balanced.
2. BST is also for fast searching
3. BST is efficient. It is efficient because they only store the elements and do not require additional memory for pointers or other data structures.

Disadvantages of Binary Tree:

1. Limited structure
2. Unbalanced trees
3. Space inefficiency
4. Slow performance in worst-case scenarios
5. Complex balancing algorithms



OPERATIONS ON BINARY SEARCH TREES:

Inorder Traversal:

1. Traverse the left subtree in inorder.
2. Visit the current node.
3. Traverse the right subtree in inorder.

Preorder Traversal:

1. Start from the root.
2. Visit the current node.
3. Traverse the left subtree in preorder.
4. Traverse the right subtree in preorder.

Postorder Traversal:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.

3. Visit the current node.

Level Order Traversal:

1. Use a queue data structure.
2. Enqueue the root node.
3. While the queue is not empty, dequeue a node, visit it, and enqueue its children.

Print Leaf Nodes:

1. Traverse the tree.
2. For each encountered node, if it is a leaf node (has no children), print its value.

Height of a Tree:

1. Recursively calculate the height of the left and right subtrees.
2. The height of the tree is the maximum of the heights of the left and right subtrees, plus 1.

Find the Min in Tree:

1. Start at the root of the BST.
2. Traverse the BST by moving to the left child until you reach a node with no left child.
3. The value of the node reached in step 2 is the minimum value in the BST.

Find the Max in Tree:

1. Start at the root of the BST.
2. Traverse the BST by moving to the right child until you reach a node with no right child.
3. The value of the node reached in step 2 is the maximum value in the BST.

Searching an Element in the Tree:

1. Start at the root of the BST.
2. While the current node is not null:
    a. If the value of the current node matches the target, return True (element found).
    b. If the target is less than the current node's value, move to the left child.
    c. If the target is greater than the current node's value, move to the right child.
3. If the loop exits without finding the target, return False (element not found).

A.Program on binary search trees using Arrays

```
#include <stdio.h>
int complete_node = 10;
int bst[10] = {20, 10, 25, 8, 12, '\0', 17, 25, '\0', '\0'};
int max, min, sum;
int print_tree(){
  printf("\n");
  for (int i = 0; i < 11; i++) {
    if (bst[i] != '\0') printf(" %d ", bst[i]);
    else printf(" - ");
  }
  return 0;
}
int get_right_child(int index){
  if (bst[index] != '\0' && ((2 * index) + 2) < complete_node)
    return (2 * index) + 2;
  return -1;
}
int get_left_child(int index){
  if (bst[index] != '\0' && (2 * index) + 1 < complete_node)
    return (2 * index) + 1;
  return -1;
}
void preorder(int index){
  if (index >= 0 && bst[index] != '\0') {
    printf(" %d ", bst[index]);
    preorder(get_left_child(index));
    preorder(get_right_child(index));
  }
}
void postorder(int index){
  if (index >= 0 && bst[index] != '\0') {
    postorder(get_left_child(index));
    postorder(get_right_child(index));
    printf(" %d ", bst[index]);
  }
}
void inorder(int index) {
  if (index >= 0 && bst[index] != '\0') {
    inorder(get_left_child(index));
    printf(" %d ", bst[index]);
    {
      if (max < bst[index])
        max = bst[index];
      if (min > bst[index])
        min = bst[index];
      sum = sum + bst[index];
    }
    inorder(get_right_child(index));
```

```
  }
}
void levelorder() {
  int j;
  for (j = 0; j < complete_node; j++) {
    if (bst[j] != '\0')
      printf(" %d ", bst[j]);
  }
}
int is_leaf(int index) {
  if (!get_left_child(index) && !get_right_child(index))
    return 1;
  if (bst[get_left_child(index)] == '\0' && bst[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 (bst[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 main() {
    printf("Binary search tree array representation: \n");
  print_tree();
  max = bst[0];
  min = bst[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));
  printf("\nLeaf nodes are: ");
  for (int i = 0; i < complete_node; i++) {
    if (bst[i] != '\0') {
      if (is_leaf(i))
        printf("\n%d is a leaf node", bst[i]);
    }
```

```
  }
  int valuee,ch=0;
  printf("enter element to search: \n");
  scanf("%d",&valuee);
  for (int i = 0; i < complete_node; i++) {
    if (bst[i] == valuee){
        ch++;
      break;
    }
  }
  ch != 0 ? printf("\nFound!"):printf("\nNot Found!");
  return 0;
}
```

Output:

```
Binary search tree array representation:
 20  10  25  8  12  -  17  25  -  -  -
Preorder:
 20  10  8  25  12  25  17
Postorder:
 25  8  12  10  17  25  20
Inorder:
 25  8  10  12  20  25  17
 max=25
 min=8
 sum=117
Levelorder:
 20  10  25  8  12  17  25
Height of the tree is: 3
Leaf nodes are:
12 is a leaf node
17 is a leaf node
25 is a leaf node
enter element to search:
```

Screenshots:

```
Binary search tree array representation:

 20  10  25  8  12  -  17  25  -  -  -
Preorder:
 20  10  8  25  12  25  17
Postorder:
 25  8  12  10  17  25  20
Inorder:
 25  8  10  12  20  25  17
 max=25
 min=8
 sum=117
Levelorder:
 20  10  25  8  12  17  25

Height of the tree is: 3

Leaf nodes are:
12 is a leaf node
17 is a leaf node
25 is a leaf node
enter element to search:
```

Search

```
enter element to search:
25

Found!
```

```
enter element to search:
120

Not Found!
PS K:\>
```

OPERATIONS ON BINARY SEARCH TREES USING LINKED LISTS:

(i) Insertion:

a. Create a new node with the data to be inserted.
b. If the tree is empty, set the root to the new node and return.
c. Otherwise, start from the root and traverse down the tree.
d. At each node, if the data to be inserted is less than the current node's data, move to the left child. If greater, move to the right child.
e. Repeat steps 4 until you reach a null pointer.
f. Set the left child of the null pointer to the new node if the data is less than the current node's data, and set the right child if the data is greater.

(ii) Deletion:

a. Search for the node containing the data to be deleted.
b. If the node has no children (leaf node), simply remove it.
c. If the node has one child, replace the node with its child.
d. If the node has two children, find the node's in-order successor or in-order predecessor
e. Replace the node to be deleted with the in-order successor/predecessor.
f. Delete the in-order successor/predecessor from its original position.

(ii) Preorder Traversal:

a. Start at the root node.
b. Visit the current node.
c. Recursively traverse the left subtree.
d. Recursively traverse the right subtree.

(iv) Inorder Traversal:

a. Start at the root node.
b. Recursively traverse the left subtree.
c. Visit the current node.
d. Recursively traverse the right subtree.

(v) Postorder Traversal:

a. Start at the root node.
b. Recursively traverse the left subtree.
c. Recursively traverse the right subtree.
d. Visit the current node.

(vi) Levelorder Traversal:

a. Enqueue the root node.
b. While the queue is not empty:
c. Dequeue a node and visit it.
d. Enqueue its left child if it exists.
e. Enqueue its right child if it exists.

(vii) Print the Leaf Nodes:

a. Start at the root node.
b. If the current node is a leaf node (both left and right children are null), print its data.
c. Recursively check the left and right subtrees.

(viii) Print the Height of the Tree:

a. If the tree is empty, the height is -1.

b. Otherwise, recursively find the height of the left and right subtrees.
c. Return the maximum height of the left and right subtrees, plus 1.

(ix) Find the Max in Tree:

a. Start at the root node.
b. If the current node's data is greater than the maximum value, update the maximum value.
c. Recursively check the right subtree.

(x) Find the Min in Tree:

a. Start at the root node.
b. If the current node's data is less than the minimum value, update the minimum value.
c. Recursively check the left subtree.

(xi) Searching an Element in the Tree:

a. Start at the root node.
b. If the current node's data matches the target value, return true.
c. If the target value is less than the current node's data, recursively search the left subtree.
d. If the target value is greater than the current node's data, recursively search the right subtree.
e. If a null pointer is reached, return false.

(xii) Find Height and Breadth of Each Node:

a. Start at the root node with height and breadth values initialized to 0.
b. For each node, update its height as the height of its parent node plus 1.
c. For each node, update its breadth as the breadth of its parent node plus 1 if it is a right child, or the same breadth if it is a left child.
d. Recursively apply steps 2 and 3 to the left and right subtrees.

B. Program on binary search tree using linked list:

```
#include <stdio.h>
#include <stdlib.h>
#include<limits.h>
int min = INT_MAX,max = INT_MIN;
struct node {
    int data;
    struct node *left;
    struct node *right;
};
struct node *createnode(int val) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = val;
    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 insert(struct node **r, int val) {
    struct node *Node = createnode(val);
    if(*r == NULL) {
        *r = Node;
    }else {
        struct node *current = *r;
        while (1) {
            if (val < current->data && current->left == NULL) {
                current->left = Node;
                break;
            } else if (val > current->data && current->right == NULL) {
                current->right = Node;
```

```
                break;
            }
            if (val < current->data) {
                current = current->left;
            } else {
                current = current->right;
            }
        }
    }
}
void minMAX(struct node *r) {
    if (r != NULL) {
        minMAX(r->left);
        if(min > r->data) min = r->data;
        if(max < r->data) max = r->data;
        minMAX(r->right);
    }
}
void inorder(struct node *r) {
    if (r != NULL) {
        inorder(r->left);
        printf("%d ", r->data);
        inorder(r->right);
    }
}
void preorder(struct node *r) {
    if (r != NULL) {
        printf("%d ", r->data);
        inorder(r->left);
        inorder(r->right);
    }
}
void postorder(struct node *r) {
    if (r != NULL) {
        inorder(r->left);
        inorder(r->right);
        printf("%d ", r->data);
    }
}
int search(struct node *r,int searchele){
    while(r->left != NULL && r->right != NULL){
        if(r->data == searchele) return 1;
        else if(r->data > searchele){
            r = r->left;
        }else r = r->right;
    }
    if(r->data == searchele) return 1;
    return 0;
}
```

```
void printLeafNodes(struct node* r) {
    if (r != NULL) {
        if(r->left == NULL && r->right == NULL)
        printf("%d ", r->data);
        inorder(r->left);
        inorder(r->right);
    }
}
int treeheight(struct node *Root){
    int leftheight,rightheight;
    if(Root == NULL) return 0;
    else{
        leftheight = treeheight(Root->left);
        rightheight = treeheight(Root->right);
        if(leftheight > rightheight) return leftheight+1;
        else return rightheight+1;
    }
}
void eachnode(struct node* r){
    int offheight = treeheight(r);
    struct queue* queue = createQueue();
    enqueue(queue, r);
    while(queue->size != 0) {
        struct node *node = dequeue(queue);
        int h = treeheight(node);
        printf("The height of %d is %d\n",node->data,h);
        if(node->left != NULL) enqueue(queue, node->left);
        if(node->right != NULL)enqueue(queue, node->right);
    }
}
void getBreadth(struct node *node, int height, int *breadth) {
    if(node== NULL){
        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(node == NULL){
```

```
        return;
    }
    else {
        int height = treeheight(node);
        printf("\nDepths: \n");
        for (int i = 0; i <= height; i++) {
            int breadth = 0;
            getBreadth(node, i, &breadth);
            if(breadth >= height ) 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() {
    struct node* root = NULL;
    int choice, value,key,del;
    while (1) {
        printf("\n1.insert 2.inorder 3.search");
        printf("\n4.preorder 5.postorder 6.min");
        printf("\n7.max 8.printleafs 9.tree height\n");
        printf("\n10.height depth 11.delete");
        printf("Enter choice:\n");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:printf("Enter data: ");
                scanf("%d", &value);
                insert(&root, value);break;
            case 2: inorder(root);break;
            case 3:printf("Enter data: ");
                scanf("%d", &value);
                key = search(root,value);
                if(key == 0) printf("Not found");
                else printf("Found");
                break;
            case 4:preorder(root);break;
            case 5:postorder(root);break;
            case 6:minMAX(root);
                printf("\nMin is %d",min);break;
            case 7:minMAX(root);
                printf("\nMax is %d",max);break;
            case 8:printLeafNodes(root);break;
            case 9:printf("Height of tree is: %d",treeheight(root));break;
            case 10: eachnode(root);
                    breadthOfEachNode(root);break;
            case 11:printf("\nENter node to be deleted: ");
                scanf("%d",&del);
             root = deleteNode(root,del); break;
            default:exit(0);
        }
    }
}
```

Output:

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
```

22501A0545

Screenshots:

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:

1
Enter data: 10
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
2
5 10 12
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
3
Enter data: 10
Found
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
4
10 5 12
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
5
5 12 10
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
6

Min is 5
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
7

Max is 12
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
8
5 12
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
9
Height of tree is: 2
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
10
The height of 10 is 2
The height of 5 is 1
The height of 12 is 1

Depths:
Node at height 0: 1
Node at height 1: 1
Node at height 2: 0
```

```
1.insert 2.inorder 3.search
4.preorder 5.postorder 6.min
7.max 8.printleafs 9.tree height
10.height depth 11.delete
Enter choice:
11

ENter node to be deleted: 12
```

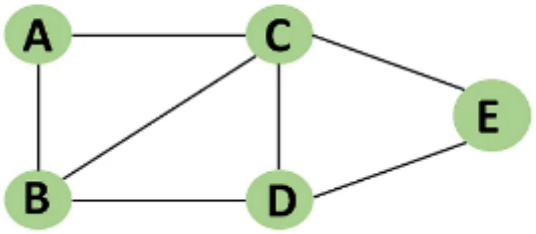
11.Graphs

Date:10/11/23

Depth first search:

Depth-first search (DFS) is an algorithm for traversing graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

Example:



A given root to algorithm
Dfs for example is **E D C B A**

Algorithm:

1. Initially all vertices are marked unvisited (false).
2. The DFS algorithm starts at a vertex u in the graph. By starting at vertex u it considers the edges from u to other vertices.
3. If the edge leads to an already visited vertex, then backtrack to current vertex u.
4. If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current root for traversal.
5. Follow this process until a vertices are marked visited.

Advantages of Depth First Search:

A. If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.
B. DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all.

Disadvatages:

A.
Depth-First Search is not guaranteed to find the solution.
B. And there is no guarantee to find a minimal solution, if more than one solution.

22501A0545

A.Program for depth first search:

```
#include <stdio.h>
#include <stdlib.h>
int vis[100];
struct Graph {
	int V;
	int E;
	int** Adj;
};
struct Graph* adjMatrix(){
	struct Graph* G = (struct Graph*)malloc(sizeof(struct Graph));
	if (!G) {
		printf("Memory Error\n");
		return NULL;
	}
	G->V = G->E = 7;
	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;
		}
	}
	G->Adj[0][1] = G->Adj[1][0] = 1;
	G->Adj[0][3] = G->Adj[3][0] = 1;
	G->Adj[1][3] = G->Adj[3][1] = 1;
	G->Adj[1][4] = G->Adj[4][1] = 1;
	G->Adj[1][5] = G->Adj[5][1] = 1;
	G->Adj[1][4] = G->Adj[4][1] = 1;
	G->Adj[6][2] = G->Adj[2][6] = 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(){
	struct Graph* G;
	G = adjMatrix();
	DFStraversal(G);
}
```

Output:

The dfs output for given Adj Matrix is:

0 1 3 4 5 2 6

Screenshots:

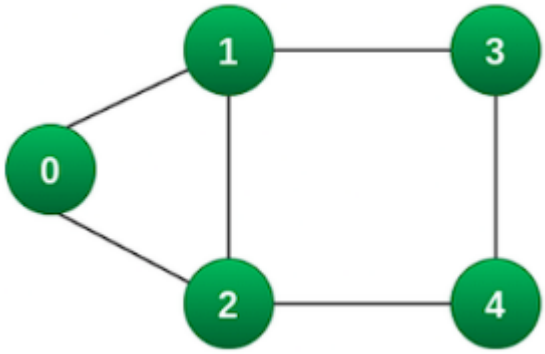
22501A0545


PROBLEMS
OUTPUT
PORTS
TERMINAL
auncher.exe' '--stdin=Microsoft-MIEngine
r=Microsoft-MIEngine-Error-s4o3kbm4.ct3'
odeBlocks\MinGW\bin\gdb.exe' '--interpre
The dfs output for given Adj Matrix is:
0 1 3 4 5 2 6
PS K:\>

22501A0545

Breadth first Search:

The Breadth First Search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.



Example:
BFS is 0 1 2 3 4

Algorithm

1. Starting from the root,
2. all the nodes at a particular level are visited first
3. the nodes of the next level are traversed till all the nodes are visited.
4. All the adjacent unvisited nodes of the current level are pushed into the queue
5. nodes of the current level are marked visited and popped from the queue.

Advantages

1. Simplicity in understanding and implementation.
2. Assurance of finding the shortest path from the start to the destination.
3. Tendency to find paths with fewer steps compared to other algorithms like Depth First Search.
4. The potential for parallelization, allowing it to utilize multiple processors and speed up the search process.

Disadvantages

1. It can be memory-intensive, as it needs to maintain information about all nodes in the search tree.
2. It may be slow in certain scenarios as it explores all nodes at each depth level before moving to the next.
3. It might not always find the most optimal solution since it doesn't explore all possible paths through the search tree.

22501A0545

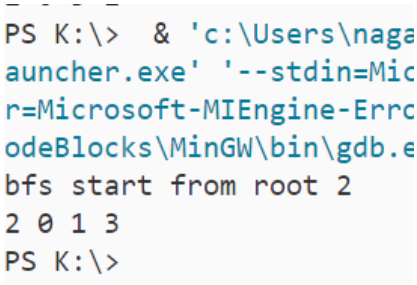
B.Program for Breadth first search

```
#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);
```

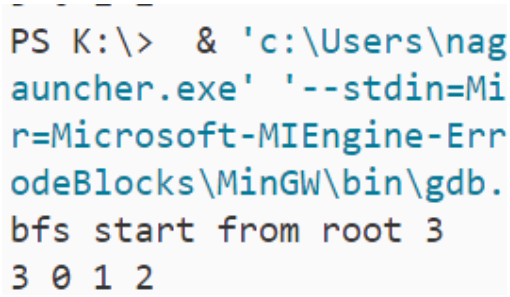
```
        Graph_addEdge(g, 1, 0);
        Graph_addEdge(g, 2, 0);
        Graph_addEdge(g, 2, 0);
        Graph_addEdge(g, 3, 0);
        printf("bfs start from root 3 \n");
        Graph_BFS(g, 3);
        return 0;
}
```

Screenshot:

Start from 2



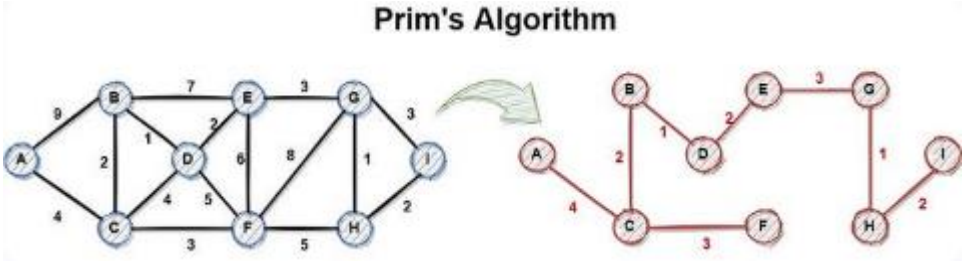
start from 3



12.Graphs - Minimum cost spanning trees date:17/11/23

Prims Algorithm:

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.



Algorithm:

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

Advantages:

1. It guarantees to find the minimum spanning tree for any connected, weighted, undirected graph.
2. It is a simple algorithm to understand and implement.
3. It works well on dense graphs, where the number of edges is close to the number of vertices.

Disadvantages:

1. It may not work efficiently on sparse graphs, where the number of edges is much smaller than the number of vertices, because of the use of priority queue.
2. It is not suitable for dynamic graphs where the edges can be added or removed frequently, as it requires the entire graph to be present upfront.
3. It requires the graph to be connected. If the graph is not connected, the algorithm will only find the minimum spanning tree of one of its connected components.

22501A0545

A.Program for Prims algorithm:

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 5
int minKey(int key[], bool mstSet[]){
	int min = INT_MAX, min_index;
	for (int v = 0; v < V; v++)
		if (mstSet[v] == false && key[v] < min)
			min = key[v], min_index = v;
	return min_index;
}
int printMST(int parent[], int graph[V][V]){
	printf("Edge \tWeight\n");
	for (int i = 1; i < V; i++)
		printf("%d - %d \t%d \n", parent[i], i,
			graph[i][parent[i]]);
}
void primMST(int graph[V][V]){
	int parent[V];
	int key[V];
	bool mstSet[V];
	for (int i = 0; i < V; i++)
		key[i] = INT_MAX, mstSet[i] = false;
	parent[0] = -1;
	for (int count = 0; count < V - 1; count++) {
		int u = minKey(key, mstSet);
		mstSet[u] = true;
		for (int v = 0; v < V; v++)
			if (graph[u][v] && mstSet[v] == false
				&& graph[u][v] < key[v])
				parent[v] = u, key[v] = graph[u][v];
	}
	printMST(parent, graph);
}
int main(){
	int graph[V][V] = { { 0, 3, 0, 6, 0 },
			{ 2, 0, 3, 8, 4 },
			{ 1, 3, 0, 0, 7 },
			{ 6, 8, 0, 0, 9 },
			{ 0, 8, 7, 8, 0 } };
	primMST(graph);
	return 0;
}
```

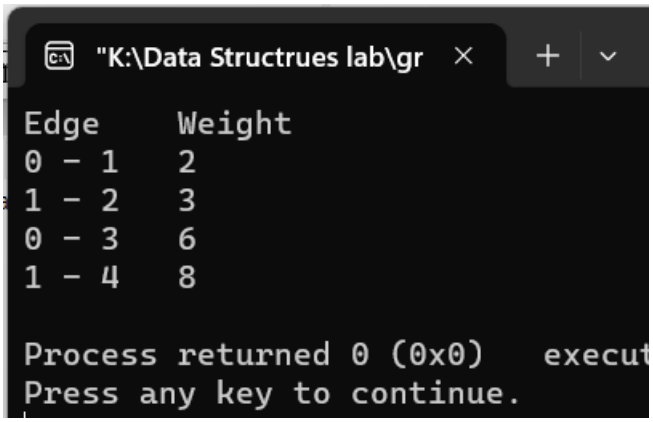
Output:

```
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
```

22501A0545

1 - 4 8

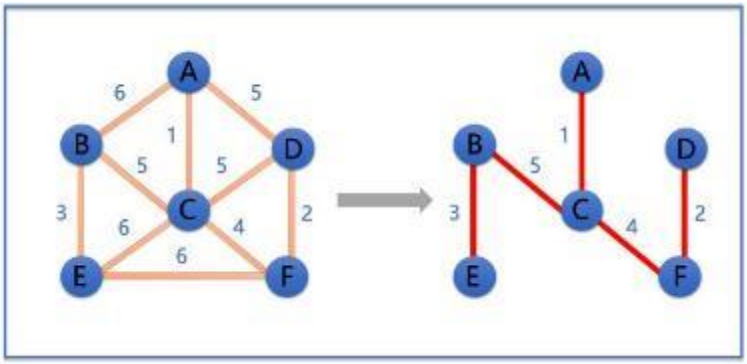
Screenshot:



Kruskal's Minimum Spanning Tree Algorithm:

Kruskal's algorithm is a greedy approach used to find the minimum spanning tree (MST) in a connected, undirected graph. The algorithm involves sorting all edges in ascending order of their weights and then iteratively adding edges to the MST, avoiding cycles. The goal is to select the smallest edge at each step that maintains acyclicity, resulting in a globally optimal MST when (V-1) edges, where V is the number of vertices, have been added.

Example:



Algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. 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.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Advanatages:

1. It runs faster than many other algorithms
2. it can solve complex problems very efficiently.
3. It requires less memory than other algorithms because it is initialized to only include part of the search space for a particular problem.

Disadvatages:

1. The running time is quadratic in the size of the data set.
2. The complexity of the algorithm is O(n).
3. The algorithm does not take into account the order of the items in the data set.
4. It is inefficient in the case of large arrays.

B.Program for Krushals algorithm:

```
#include <stdio.h>
#include <stdlib.h>
int comparator(const void* p1, const void* p2){
        const int(*x)[3] = p1;
        const int(*y)[3] = p2;
```

```
	return (*x)[2] - (*y)[2];
}
void makeSet(int parent[], int rank[], int n){
	for (int i = 0; i < n; i++) {
		parent[i] = i;
		rank[i] = 0;
	}
}
int findParent(int parent[], int component){
	if (parent[component] == component)
		return component;
	return parent[component]= findParent(parent, parent[component]);
}
void unionSet(int u, int v, int parent[], int rank[], int n){
	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;
		rank[u]++;
	}
}
void kruskalAlgo(int n, int edge[n][3]){
	qsort(edge, n, sizeof(edge[0]), comparator);
	int parent[n];
	int rank[n];
	makeSet(parent, rank, n);
	int minCost = 0;
	printf("Edges in 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 (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);
}
```

```
int main(){
	int edge[5][3] = { { 0, 1, 10 },
					{ 0, 2, 5},
					{ 0, 3, 5 },
					{ 1, 3, 5 },
					{ 6, 3, 4 } };
	kruskalAlgo(5, edge);
	return 0;
}
```

Screenshot: