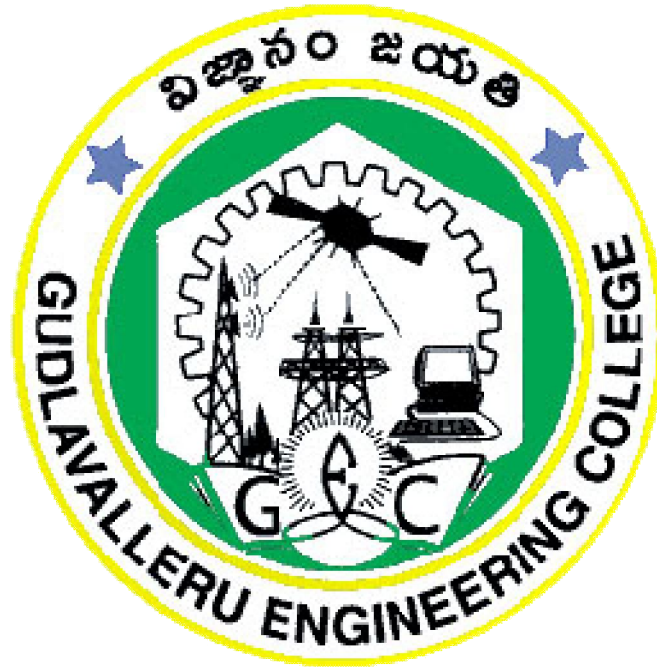


DATA STRUCTURES LAB FACULTY MANUAL
II Year I Semester
2020-21



Prepared by
Dr. B. Sai Chandana
Associate Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
GUDLAVALLERU ENGINEERING COLLEGE
(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)
Seshadri Rao Knowledge Village, Gudlavalleru – 521356.

GUDLAVALLERU ENGINEERING COLLEGE
(An Autonomous Institution with Permanent Affiliation to JNTUK, Kakinada)
Seshadri Rao Knowledge Village, Gudlavalleru – 521356
DEPARTMENT OF INFORMATION TECHNOLOGY

INSTITUTE VISION & MISSION

Institute Vision:

To be a leading institution of engineering education and research, preparing students for leadership in their fields in a caring and challenging learning environment.

Institute Mission:

- To produce quality engineers by providing state-of-the-art engineering education.
- To attract and retain knowledgeable, creative, motivated and highly skilled individuals whose leadership and contributions uphold the college tenets of education, creativity, research and responsible public service.
- To develop faculty and resources to impart and disseminate knowledge and information to students and also to society that will enhance educational level, which in turn, will contribute to social and economic betterment of society.
- To provide an environment that values and encourages knowledge acquisition and academic freedom, making this a preferred institution for knowledge seekers.
- To provide quality assurance.
- To partner and collaborate with industry, government, and R&D institutes to develop new knowledge and sustainable technologies and serve as an engine for facilitating the nation's economic development.
- To impart personality development skills to students that will help them to succeed and lead.
- To instil in students the attitude, values and vision that will prepare them to lead lives of personal integrity and civic responsibility.
- To promote a campus environment that welcomes and makes students of all races, cultures and civilizations feel at home.
- Putting students face to face with industrial, governmental and societal challenges.

DEPARTMENT VISION & MISSION**VISION**

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.

MISSION

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth

PROGRAMME EDUCATIONAL OBJECTIVES(PEOs):-

PEO1: Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate

PEO2: Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations.

PEO3: Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the public.

PROGRAM OUTCOMES (POs)

Engineering students will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

Students will be able to

PSO1: Design, develop, test and maintain reliable software systems and intelligent systems.

PSO2: Design and develop web sites, web apps and mobile apps.

Course Objectives:

- To implement different searching and sorting algorithms.
- To implement linear and non-linear data structures.

Course Outcomes:

Students will be able to

- Demonstrate linear and binary search techniques to find an element in a given list of numbers.
- Select an appropriate sorting technique to sort the given list of numbers.
- Develop suitable code to simulate the operations on linked lists.
- Determine the suitable ways to implement Stacks and Queues.
- Choose appropriate data structures for evaluation of arithmetic expressions.
- Demonstrate the operations on Binary Search Trees and Graphs.
- Determine the use of hashing in implementing dictionaries.

Mapping Of Course Outcomes With Program Outcomes:

DATA STRUCTURES LAB	1	2	3	4	5	6	7	8	9	10	11	12	PSO1	PSO2
CO1: Demonstrate linear and binary search techniques to find an element in a given list of numbers.	3	3	3					2	2	2	2	2	2	3
CO2: Select an appropriate sorting technique to sort the given list of numbers.	3	3	3					2	2	2	2	2	2	3
CO3: Develop suitable code to simulate the operations on linked lists.	3	2	2					2	2	2	1	2	2	2
CO4: Determine the suitable ways to implement Stacks and Queues.	3	3	3					2	2	2	1	2	2	2
CO5: Choose appropriate data structures for evaluation of arithmetic expressions.	3	3	3					2	2	2	2	2	2	3
CO6: Demonstrate the operations on Binary Search Trees and Graphs.	3	3	3					2	2	2	2	2	2	3
CO7: Determine the use of hashing in implementing dictionaries.	3	3	3					2	2	2	2	2	2	3

LIST OF EXPERIMENTS

Exercise	Program Name	Mapping Of CO's	Page No
I	1. Develop recursive and non-recursive functions to perform search for a key value in a given list using i. Linear Search ii. Binary Search	CO1	7
II	2. Implement the following sorting techniques to sort a given list of integers in ascending order i. Bubble Sort ii. Insertion Sort iii. Selection Sort	CO2	31
III	3. Use functions to i. Create a singly linked list. ii. Insert an element into a singly linked list. iii. Delete an element from a singly linked list.	CO3	53
IV	4. Use functions to i. Create a doubly linked list. ii. Insert an element into a doubly linked list. iii. Delete an element from a doubly linked list.	CO3	80
V	5. Implement stack (its operations) using arrays. 6. Implement queue (its operations) using linked list.	CO4	108
VI	7. To convert infix expression into postfix expression. 8. To evaluate postfix expression.	CO5	121
VII	9. Create a binary search tree of integers and perform the following operations i. insert ii. traversal (pre-order, in-order, post-order)	CO6	135
VIII	10. Implement the DFS and BFS traversals on graphs.	CO6	146
IX	11. Create a hash table to perform the following operation. i. insertion ii. Display iii. Search	CO7	181

EXERCISE-I**1. (i) (a)****AIM:**

Write C programs that use recursive functions to perform Linear search for a Key value in a given list.

DESCRIPTION:**Searching:**

- ❖ Searching is the process of finding a given value position in a list of values.
- ❖ It decides whether a search key is present in the data or not.
- ❖ It is the algorithmic process of finding an element in a collection of elements.
- ❖ It can be done on internal data structure or on external data structure.

To search an element in a given array, it can be done in following ways:

1. Linear Search
2. Binary Search

Linear Search

- Linear search is also called as Sequential Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

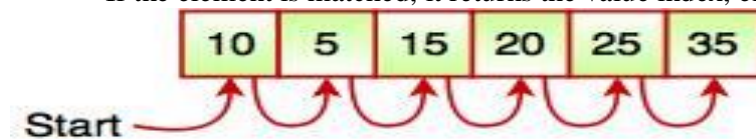


Fig. Sequential Search

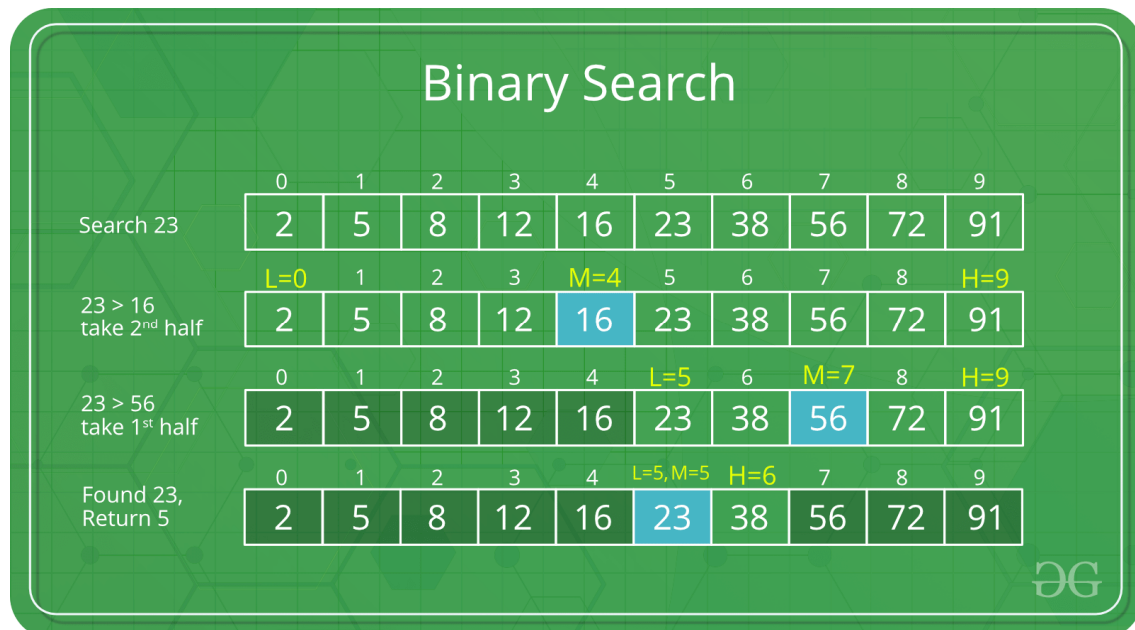
The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

The time complexity of linear search is $O(n)$. Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Binary Search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example:



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Important Differences between Linear Search and Binary Search:

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search $-O(n)$, Binary search has time complexity $O(\log n)$.
- Linear search performs equality comparisons and Binary search performs ordering comparisons

Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Algorithm:

Algorithm Linearsearch_Recursion(a<array>,n,ele,i)

Input: a is an array with n elements,ele is the element to be search, i is starting index.

Output: position of required element in array, if it is available.

1. found = 0
2. if($i < n$ and found == 0)
 - a) if($a[i] == ele$)
 - i) print(required element was found at position i)
 - ii) found=1

- b) else
 - i) $i = i + 1$
 - ii) Linearsearch_Recursion (a,n,ele,i)
- c) end if
- 3. end if
- 4. if(found!=1)
 - a) print(required element is not available)
- 5. end if

End Linearsearch_Recursion

Source Code:

```

/* LINEAR SEARCH USING RECURSION */
int linearrec(int [],int,int,int);// linearrec() function declaration// This function to search an
element
void main()
{
    int a[20],n,i,flag=0,ele;                //variables declaration
    clrscr();
    printf("Enter number of element to array");
    scanf("%d",&n);//reading number of elements of an array
    printf("\n Enter elements into array");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);// reading 'n' values
    }
    printf("\n Enter element to search");
    scanf("%d",&ele);//reading which ele value to be searched
    flag=linearrec(a,n,ele,0);// function linearrec() calling
    if(flag==1)//Check flag value 1 or not if flag==1 displays as Successful Search
    {
        printf("\n Successful search");
    }
    else //if flag!=1 executes the following statement will be executed
    {
        printf("\n The given element was not found in the array");
    }
    getch();
}
int linearrec(int a[],int n,int ele,int i)//function to search an element
{
    if(i<n)
    {
        if(a[i]==ele)
        {
            printf("\n Element found at %d location",i);
            return 1;
        }
        else

```

```
        {  
            i=i+1;  
            linearrec(a,n,ele,i);//recursive function calling  
        }  
    }  
}
```

Output 1:

Enter number of element to array5
Enter elements into array
10 2 20 3 11
Enter element to search2
Element found at 1 location
Successful search

Output 2:

Enter number of element to array6
Enter elements into array
12 36 14 10 2 6
Enter element to search
42
The given element was not found in the array.

Viva Questions:

1. What is recursion?
2. Define Searching?
3. What are different types of Searching Techniques?
4. What is linear searching?
5. Find the number of elements comparisons to check whether element value 32 is in the following list or not.
List: 12,34,67,32,69,99,45
6. Define binary search?
7. What is best case efficiency of linear search?
8. What do you understand by the term “linear search is unsuccessful”?

Lab Assignment:

1. Apply the Linear Search on the following elements 21,67,5,98,32, 43,50,74 for checking whether element 32 is presented in the given list of elements.
2. Apply the Linear Search on the following elements 21,11,5,78,49, 54,72,88 for checking whether element 6 is presented in the given list of elements.
3. Suppose an array A with elements indexed 1 to n is to be searched for a value x. Write pseudo code that performs a forward search, returning n + 1 if the value is not found.
4. Write an algorithm to search for an employee ID in an array

Conclusion:

Designed program for Linear Search using Recursive function.

1. (i) (b)

AIM:

Write C programs that use non-recursive functions to perform Linear search for a Key value in a given list.

DESCRIPTION:**Searching:**

- ❖ Searching is the process of finding a given value position in a list of values.
- ❖ It decides whether a search key is present in the data or not.
- ❖ It is the algorithmic process of finding an element in a collection of elements.
- ❖ It can be done on internal data structure or on external data structure.

To search an element in a given array, it can be done in following ways:

1. Linear Search
2. Binary Search

Linear Search:

- Linear search is also called as Sequential Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

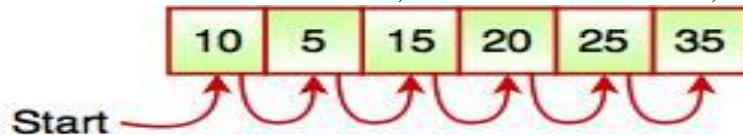


Fig. Sequential Search

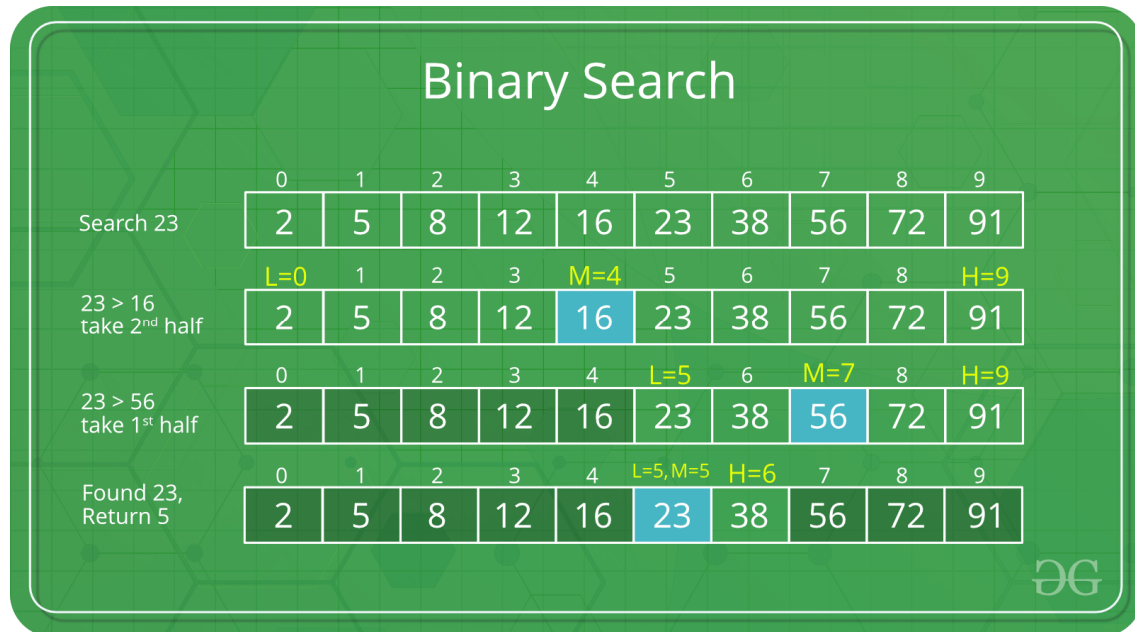
The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

The time complexity of linear search is $O(n)$. Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Binary Search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So, we recur for right half.
4. Else (x is smaller) recur for the left half.

Important Differences between Linear Search and Binary Search:

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search $-O(n)$, Binary search has time complexity $O(\log n)$.
- Linear search performs equality comparisons and Binary search performs ordering comparisons.

Non Recursive Function:

Non-Recursive Function are procedures or subroutines implemented in a programming language, whose implementation does not reference itself.

Some other points are

- A recursive function in general has an extremely high time complexity while a non-recursive one does not.
- A recursive function generally has smaller code size whereas a non-recursive one is larger.
- In some situations, only a recursive function can perform a specific task, but in other situations, both a recursive function and a non-recursive one can do it.

Algorithm:

Algorithm Linearsearch($a<array>,n,ele$)

Input: a is an array with n elements, ele is the element to be search.

Output: position of required element in array, if it is available.

1. i = 0
2. found = 0
3. while(i < n and found == 0)
 - a) if(a[i] == ele)
 - i) print(required element was found at position i)
 - ii) found=1
 - b) else
 - i) i = i + 1
 - c) end if
4. end if
5. if(found!=1)
 - a) print(required element is not available)
6. end if

End Linearsrch

Source Code:

```

/* LINEAR SEARCH USING NON RECURSION */
int linear(int [],int,int);// This function declaration for searching an element
void main()
{
    int a[20],n,i,flag=0,ele;//variables declaration
    clrscr();
    printf("Enter number of element to array");
    scanf("%d",&n);//reading number of elements of an array
    printf("\n Enter elements to array");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);//reading 'n' element values
    }
    printf("\n Enter element to search");
    scanf("%d",&ele);//reading which ele value to be searched
    flag=linear(a,n,ele
    if(flag!=0)//Check flag value not equal to 0 or not if flag!=0 displays as Successful
Search);          // function linear() calling
    {
        printf("\n Successful search");
    }
    else//if flag==0 executes the following statement will be executed
    {
        printf("\n The given element was not found in the array");
    }
    getch();
}

```

```

int linear(int a[],int n,int ele)//function to search an element
{
    int i;
    for(i=0;i<n;i++)
    {
        if(a[i]==ele)
        {
            printf("\n Element found at %d location",i);
            return 1;
        }
    }
    return 0;
}

```

Output1:

Enter number of element to array6
Enter elements to array
15 22 10 3 4 6
Enter element to search10
Element found at 2 location
Successful search

Output2:

Enter number of element to array5
Enter elements to array
1 6 4 3 7
Enter element to search10
The given element was not found in the array

Lab Questions:

1. What is an array?
2. What is the time complexity of Linear Search?
3. Why Searching is required?
4. What are the differences between recursive function and Non-recursive function?
5. What are the various applications of linear search?
6. Linear search is special case of which search?
7. What is the worst case for linear search?
8. What is the disadvantage of linear search?
9. During linear search, when the record is present in middle position then how many comparisons are made?

Lab Assignment:

1. Apply the Linear Search on the following elements 25,67,5,432,342, 143,350,274 for checking whether element 342 is presented in the given list of elements.
2. Apply the Linear Search on the following elements215,171,5,708,459, 514,724,88 for checking whether element 651 is presented in the given list of elements.

Conclusion:

Successfully implemented Linear Search using Non-Recursive function.

1. (ii) (a)

AIM:

Write C programs that use recursive functions to perform Binary search for a Key value in a given list.

DESCRIPTION:

Searching:

- ❖ Searching is the process of finding a given value position in a list of values.
- ❖ It decides whether a search key is present in the data or not.
- ❖ It is the algorithmic process of finding an element in a collection of elements.
- ❖ It can be done on internal data structure or on external data structure.

To search an element in a given array, it can be done in following ways:

1. Linear Search
2. Binary Search

Linear Search

- Linear search is also called as Sequential Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

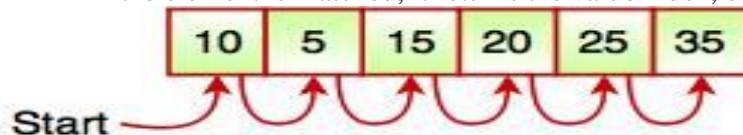


Fig. Sequential Search

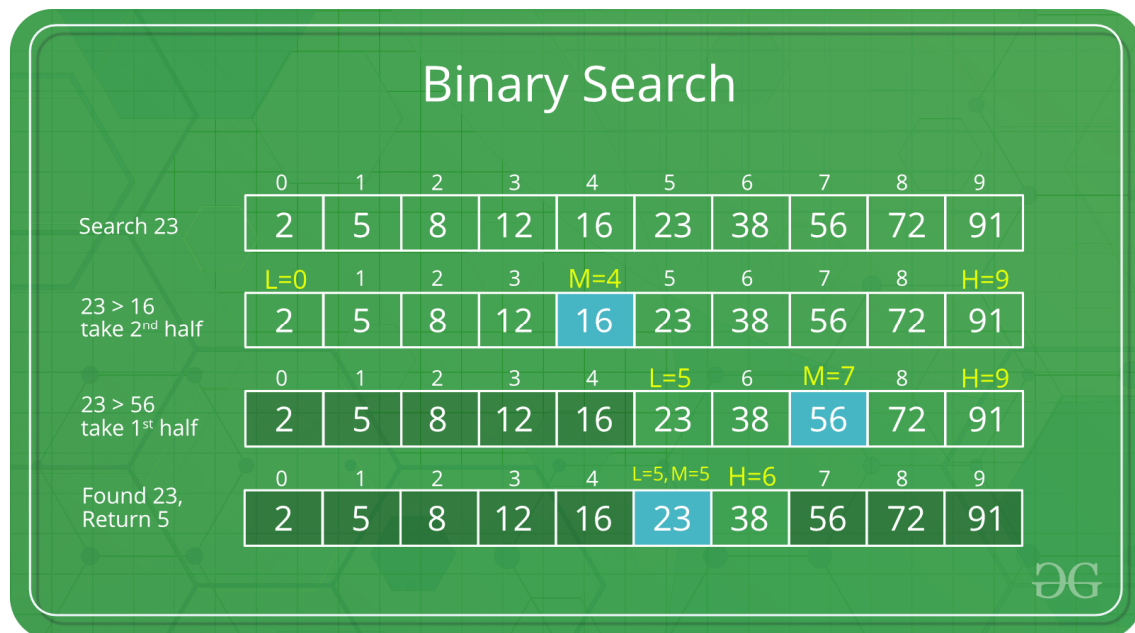
The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

The time complexity of linear search is $O(n)$. Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Binary Search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So, we recur for right half.
4. Else (x is smaller) recur for the left half.

Important Differences between Linear Search and Binary Search:

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search $-O(n)$, Binary search has time complexity $O(\log n)$.
- Linear search performs equality comparisons and Binary search performs ordering comparisons.

Recursion:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain

problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Algorithm:

AlgorithmBinarysearch_Recursion(a<array>,ele,low,high)

Input: a is array with n elements,ele is the element to be search, low is starting index, high is ending index.

Output: position of required element in array, if it is available.

1. found=0.
2. if(low<=high)
 - a) mid=(low+high)/2
 - b) if(ele==a[mid])
 - i) print(required element was found at mid position)
 - ii) found=1
 - c) else if(ele<a[mid])
 - i) Binarysearch_Recursion(a,ele,low,mid-1)
 - d) else if(ele>a[mid])
 - i) Binarysearch_Recursion(a,ele,mid+1,high)
 - e) end if
- 3.end if
- 4.if(found!=1)
 - a) print(required element is not available)
5. end if

End Binarysearch_Recursion

Source Code:

```

/* BINARY SEARCH USING RECURSION */
int binaryrec(int [],int,int,int,int);//function declaration for searching an element
void main()
{
    int a[20],n,i,flag=0,ele;//variables declaration
    clrscr();
    printf("Enter number of element to array");
    scanf("%d",&n);//reading number of elements of an array
    printf("\n Enter elements to array");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);//reading 'n' element values
    }
    printf("\n Enter element to search");
    scanf("%d",&ele);//reading which ele value to be searched
    flag=binaryrec(a,n,ele,0,n-1); // function binaryrec() calling

    if(flag==1)//Check flag value 1 or not if flag==1 displays as Successful Search
    {
        printf("\n Successful search");
    }
}

```

```

    Else//if flag! =1 executes the following statement will be executed
    {
        printf("\n The given element was not found in the array");
    }
    getch();
}

int binaryrec(int a[],int n,int ele,int first,int last)//function to search an element
{
    int mid;
    if(first<=last)
    {
        mid=(first+last)/2;
        if(ele==a[mid])
        {
            printf("\n The giveelement was found at %d location",mid);
            return 1;
        }
        else if(ele<a[mid])
        {
            binaryrec(a,n,ele,first,mid-1);//recursive function call
        }
        else if(ele>a[mid])
        {
            binaryrec(a,n,ele,mid+1,last);//recursive function call
        }
    }
}
}

```

Output 1:

Enter number of element to array5
Enter elements to array
22 33 44 55 66
Enter element to search
33
The given element was found at 1 location
Successful search

Output 2:

Enter number of element to array6
Enter elements to array
10 20 112 123 145 368
Enter element to search23
The given element was not found in the array

VIVA Questions:

1. What is the worst-case complexity of binary search using recursion?

2. Given an array $arr = \{45,77,89,90,94,99,100\}$ and $key = 99$; what are the mid values (corresponding array elements) in the first and second levels of recursion?
3. What is the average case time complexity of binary search using recursion?
4. What are the applications of binary search?
5. When is a binary search best applied?

Lab Assignment:

1. What will be the maximum number of guesses required by Binary Search, to search a number in a list of 2,097,152 elements?
2. Find number of element comparisons required by Binary Search, to search value 34 in a list of **45,67,78,89,99,112,134,178** elements?
3. Find number of element comparisons required by Binary Search, to search value 708 in a list of **101,202,303,404,505,607,708,909** elements?
4. Find average number of element comparisons required for successful search for the following list of 5,11,18,23,45,67,89,99,102 elements
5. Find average number of element comparisons required for unsuccessful search for the following list of 5,11,18,23,45,67,89,99,102 elements.

Conclusion:

Successfully implemented Binary Search using Recursive function.

1. (ii) (b)

AIM:

Write C programs that non-recursive functions to perform Binary search for a Key value in a given list.

DESCRIPTION:

Searching:

- ❖ Searching is the process of finding a given value position in a list of values.
- ❖ It decides whether a search key is present in the data or not.
- ❖ It is the algorithmic process of finding an element in a collection of elements.
- ❖ It can be done on internal data structure or on external data structure.

To search an element in a given array, it can be done in following ways:

1. Linear Search
2. Binary Search

Linear Search

- Linear search is also called as Sequential Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

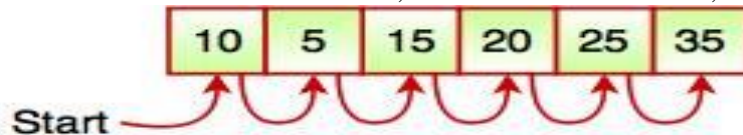


Fig. Sequential Search

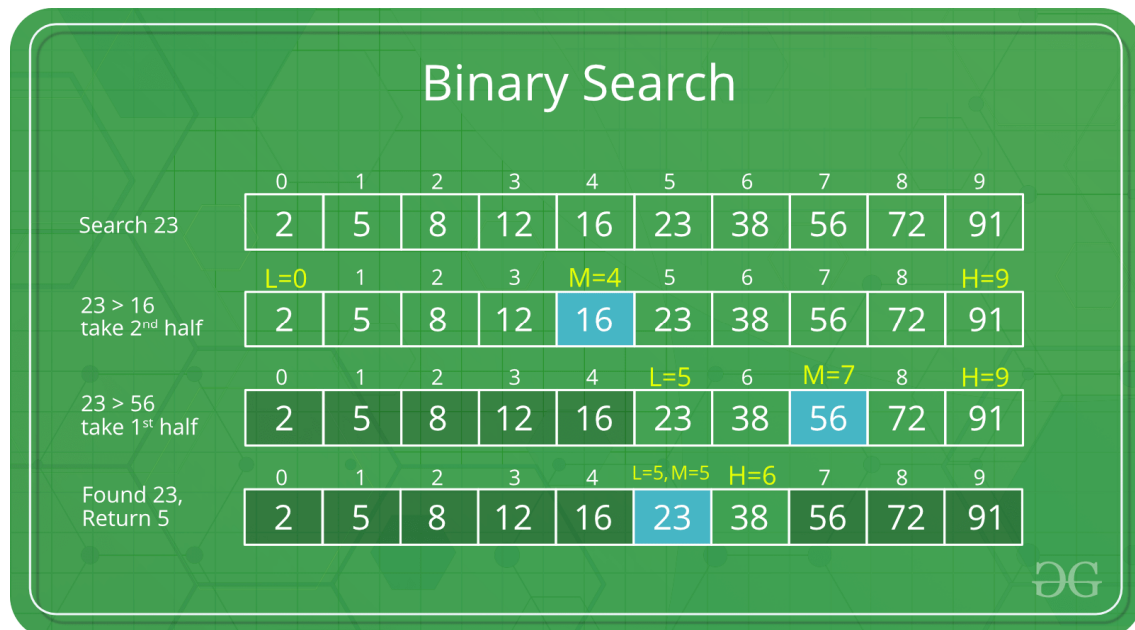
The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

The time complexity of linear search is $O(n)$. Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Binary Search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So, we recur for right half.
4. Else (x is smaller) recur for the left half.

Important Differences between Linear Search and Binary Search:

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search $-O(n)$, Binary search has time complexity $O(\log n)$.
- Linear search performs equality comparisons and Binary search performs ordering comparisons.

Non-Recursive Function:

Non-Recursive Function are procedures or subroutines implemented in a programming language, whose implementation does not reference itself.

Some other points are

- A recursive function in general has an extremely high time complexity while a non-recursive one does not.
- A recursive function generally has smaller code size whereas a non-recursive one is larger.
- In some situations, only a recursive function can perform a specific task, but in other situations, both a recursive function and a non-recursive one can do it.

Algorithm:

AlgorithmBinarysearch (a<array>, ele)

Input: a is array with n elements,ele is the element to be search.

Output: position of required element in array, if it is available.

1. found=0.
- 2.while(low<=high)
 - a) mid=(low+high)/2
 - b) if(ele==a[mid])
 - i) print(required element was found at mid position)
 - ii) found=1
 - c) else if(ele<a[mid])
 - i) high = mid - 1
 - d) else if(ele>a[mid])
 - i) low = mid + 1
 - e) end if
- 3.end if
- 4.if(found!=1)
 - a) print(required element is not available)
5. end if

End Binarysearch

Source Code:

```

/* BINARY SEARCH USING NON-RECURSION */
int binarysearch(int [],int,int);
void main()
{
    int a[20],n,i,flag=0,ele;
    clrscr();
    printf("Enter number of element to array");
    scanf("%d",&n);
    printf("\n Enter elements to array");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\n Enter element to search");
    scanf("%d",&ele);
    flag=binarysearch(a,n,ele);
    if(flag!=0)
    {
        printf("\n Successful search");
    }
    else
    {
        printf("\n The given element was not found in the array");
    }
    getch();
}

```

```
int binarysearch(int a[],int n,int ele)
{
    int first,last,mid;
    first=0;
    last=n-1;
    while(first<=last)
    {
        mid=(first+last)/2;
        if(ele==a[mid])
        {
            printf("\n The given element was found at %d location",mid);
            return 1;
        }
        else if(ele<a[mid])
        {
            last=mid-1;
        }
        else if(ele>a[mid])
        {
            first=mid+1;
        }
    }
    return 0;
}
```

Output 1:

Enter number of element to array5
Enter elements to array
12 15 17 18 25
Enter element to search18
The given element was found at 3 location
Successful search

Output 2:

Enter number of element to array4
Enter elements to array
1 2 10 15
Enter element to search22
The given element was not found in the array

VIVA Questions:

1. What is binary search?
2. What are the applications of binary search?
3. What is the advantage of recursive approach than an iterative approach?
4. Binary Search can be categorized into which of the Techniques?
5. Given an array arr = {5,6,77,88,99} and key = 88; How many iterations are done until the element is found?
6. What is the time complexity of binary search with iteration?

Lab Assignment:

1. Given an array arr = {45,77,89,90,94,99,100} and key = 100; What are the mid values(corresponding array elements) generated in the first and second iterations?

Conclusion:

Successfully implemented Binary Search using Non-Recursive function.

EXERCISE-II

2. (i)

AIM:

Write C program to sort a given list of integers in ascending order using Bubble sort.

DESCRIPTION:**Sorting:**

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

- **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

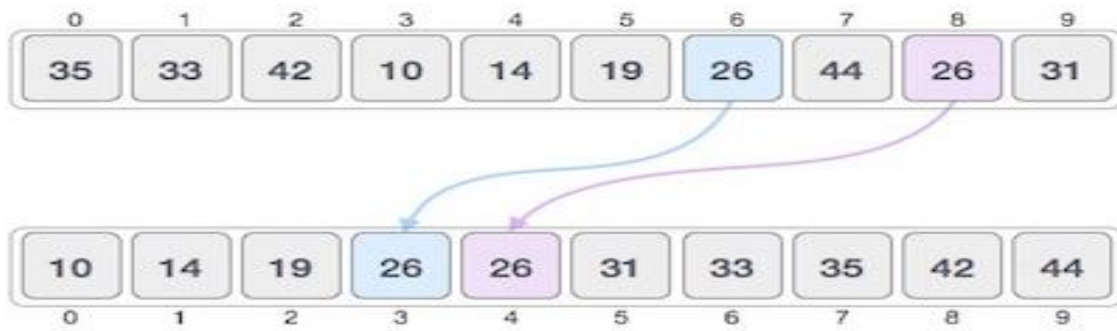
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.

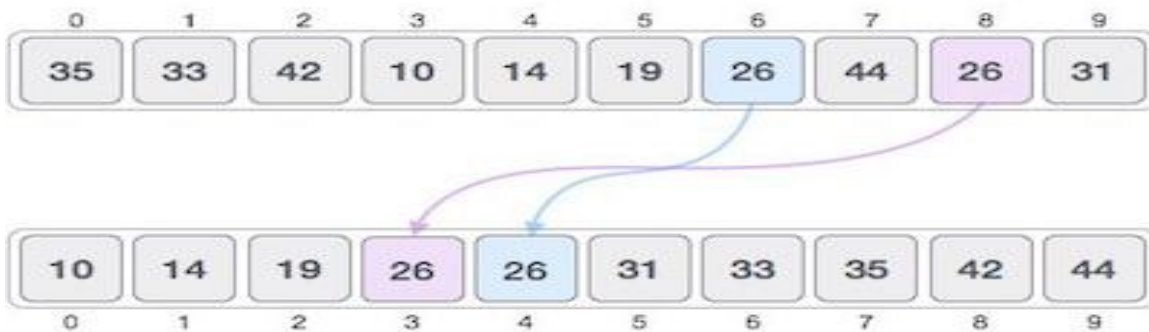
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Types of Sorting Algorithms:

1. Quick Sort
2. Bubble Sort
3. Merge Sort
4. Insertion Sort
5. Selection Sort
6. Heap Sort
7. Radix Sort
8. Bucket Sort

Algorithm	Best	Average	Worst
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$

Time Complexities of Sorting Algorithms:

Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$

Bubble Sort:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

How can we sort an array?

In Bubble Sort, we take the simplest possible approach to sort an array.

- We look through the array in an orderly fashion, comparing only adjacent elements at a time.
- Whenever we see two elements which are out of order (refer to the picture below), we swap them so that the smaller element comes before the greater element.
- We keep performing the above steps over the array again and again till we get the sorted form.

When should we swap?

ADJACENT ELEMENTS IN THE CORRECT ORDER (10 < 23)



ADJACENT ELEMENTS IN THE INCORRECT ORDER (56 > 10)



SWAP 10 AND 56 (TO GET CORRECT ORDER)

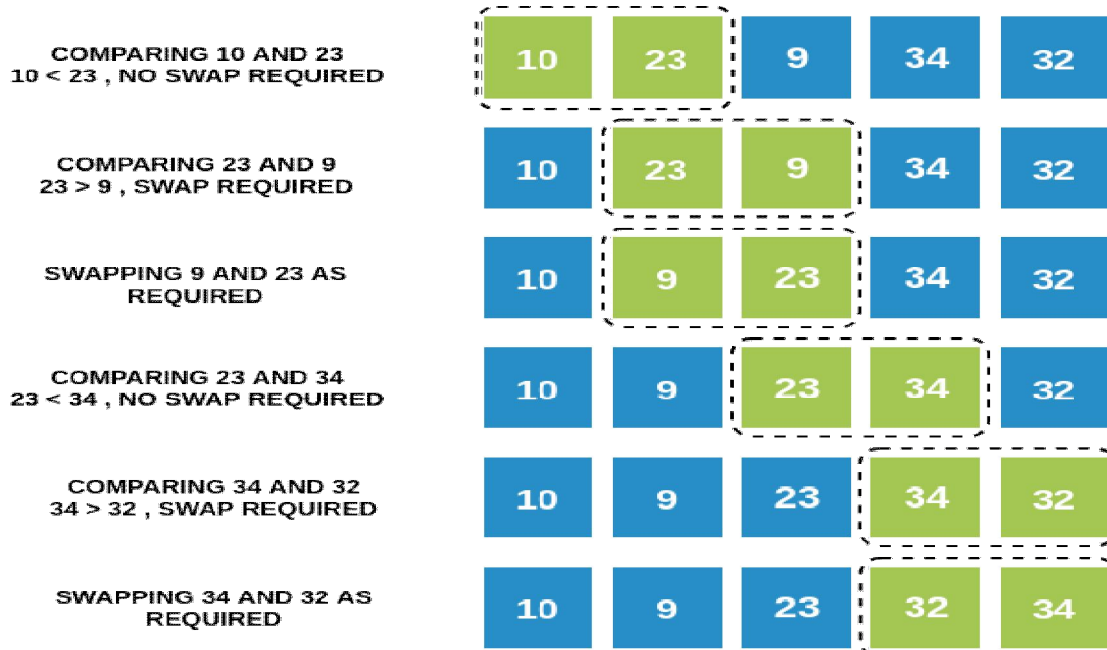


Let's take note of a few important observations:

- If we start from the first index and keep comparing the i^{th} and $(i+1)^{\text{th}}$ element (where i varies from 1st to the second last index), at the end of one iteration, we see that the **greatest** element in the entire array has reached the **last** position.

- This is because no matter where the greatest element was, it gets swapped repeatedly to reach its correct position. Refer to the picture below!
- Similarly, if we do a **second** iteration, we will end up with the **second greatest element** in the second last position.

Step by Step Process for One Iteration:



Algorithm:

Algorithm bubblesort(a<array>, n)

Input: a is array with n elements to be sort.

Output: array elements in ascending order.

1. i = 0
2. while(i < n)
 - a) j = 0
 - b) while (j < n-i-1)
 - i) if (a[j] > a[j+1])
 - A) t = a[j]
 - B) a[j] = a[j+1]
 - C) a[j+1] = t
 - ii) end if
 - iii) j = j + 1
 - c) end loop
 - d) i = i + 1
3. end loop

Endbubblesort

Source Code:

```
/* BUBBLE SORT */
```

```

#include<stdio.h>
void bubblesort(int [],int);
void main()
{
    int a[20],n,i;
    clrscr();
    printf("Enter number of element to array");
    scanf("%d",&n);
    printf("\n Enter elements to array");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    bubblesort(a,n);
    printf("\n After sorting\n");
    for(i=0;i<n;i++)
    {
        printf("\t %d",a[i]);
    }
    getch();
}

```

```

void bubblesort(int a[],int n)
{
    int i,j,t;
    for(i=0;i<n;i++)
    {
        for(j=0;j<n-i-1;j++)
        {
            if(a[j]>a[j+1])
            {
                t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
            }
        }
    }
}

```

Output 1:

```

Enter number of element to array5
Enter elements to array
12 10 11 13 9
After sorting
    9    10    11    12    13

```

Output 2:

```

Enter number of element to array4

```

Enter elements to array

-9 -3 0 14

After sorting

-9 -3 0 14

VIVA Questions:

1. What is the difference between a stable and unstable sorting algorithm?
2. What is sorting process?
3. Which element reaches its correct position after 1st iteration?
4. How many comparisons do I need to make for one iteration on an array of size 4?
5. How can the efficiency of bubble sort algorithm be determined?
6. How do we define bubble sort?
7. What main properties are considered in bubble sort?
8. What is the complexity of the bubble sort algorithm?
9. What is an internal sorting algorithm?
10. What is the worst case complexity of bubble sort?
11. What is the average case complexity of bubble sort?

Lab Assignment:

1. The given array is arr = {1, 2, 4, 3}. Bubble sort is used to sort the array elements. How many iterations will be done to sort the array?
2. Find the number of swappings needed to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order using bubble sort.
3. Arrange list of 21,67,5,98,32, 43,50,7,88 elements in Ascending order using bubble sort.
4. Arrange list of 21,67,5,98,32, 43,50,7,88 elements in Descending order using bubble sort.

Conclusion:

Designed a program for bubble sort.

2. (ii)

AIM:

Write C program to sort a given list of integers in ascending order using Insertion sort.

DESCRIPTION:

Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

- **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

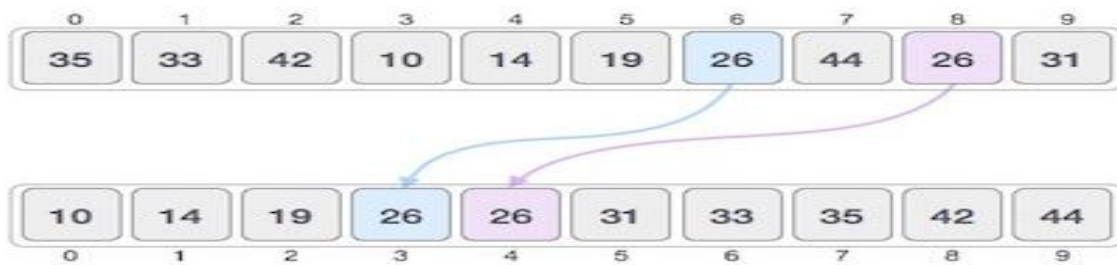
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.

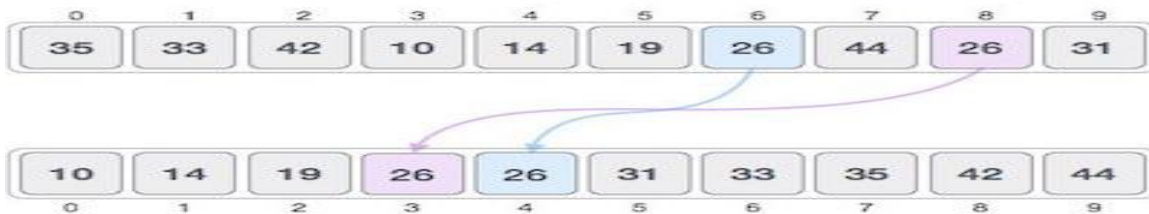
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Types of Sorting Algorithms:

1. Quick Sort
2. Bubble Sort
3. Merge Sort
4. Insertion Sort
5. Selection Sort
6. Heap Sort
7. Radix Sort
8. Bucket Sort

Time Complexities of Sorting Algorithms:

Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where **n** is the number of items.

insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(kn)$ when each element in the input is no more than k places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount $O(1)$ of additional memory space
- Online; i.e., can sort a list as it receives it.

How Insertion Sort Works?

We take an unsorted array for our example.



Algorithm	Best	Average	Worst
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$

Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm:

Algorithminsertionsort(a<array>, n)

Input: a is array with n elements to be sort.

Output:array elements in ascending order.

1. i = 1
2. while(i < n)
 - a) x = a[i]
 - b) j = i -1
 - c) while(j >= 0 and a[j] > x)
 - i) a[j] = a[j+1]
 - ii) j = j -1
 - d) end loop
 - e) a[j+1] = x
 - f) i = i + 1
3. end loop

Endinsertionsort

Program:

/* SORT THE GIVEN ELEMENTS USING INSERTION SORT */

void insertion(int [],int);

void main()

```
{
    int a[2],n,i;
    clrscr();
    printf("\n Enter no.of elements to array");
    scanf("%d",&n);
    printf("\n Enter elements to array");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\n After sorting");
    insertion(a,n);
    for(i=0;i<n;i++)
    {
```

```

        printf("\t %d",a[i]);
    }
    getch();
}

void insertion(int a[],int n)
{
    int i,j,x;
    for(i=1;i<n;i++)
    {
        x=a[i];
        j=i-1;
        while(j>=0 && a[j]>x)
        {
            a[j+1]=a[j];
            j=j-1;
        }
        a[j+1]=x;
    }
}

```

Output 1:

Enter no.of elements to array5
Enter elements to array
12 10 4 6 9
After sorting 4 6 9 10 12

Output 2:

Enter no.of elements to array4
Enter elements to array-3 -1 0 -6
After sorting -6 -3 -1 0

VIVA Questions:

1. what is sorting?
2. List the various sorting algorithms?
3. What is the advantage of insertion sort?
4. How insertion sort works?
5. Is insertion sort is stable algorithm?
6. What is the time complexity of insertion sort?
7. Why sorting is required?
8. List the application of Insertion sort?

Lab Assignment:

1. Apply the insertion sort on the following elements 21,11,5,78,49, 54,72,88
2. Apply the insertion sort on the following elements 21,11,5,78,49, 54,72,88 and 56,28,10.

Conclusion:

Designed program for Insertion sort.

2. (iii)

AIM:

Write C program to sort a given list of integers in ascending order using Selection sort.

DESCRIPTION:

Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

- **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

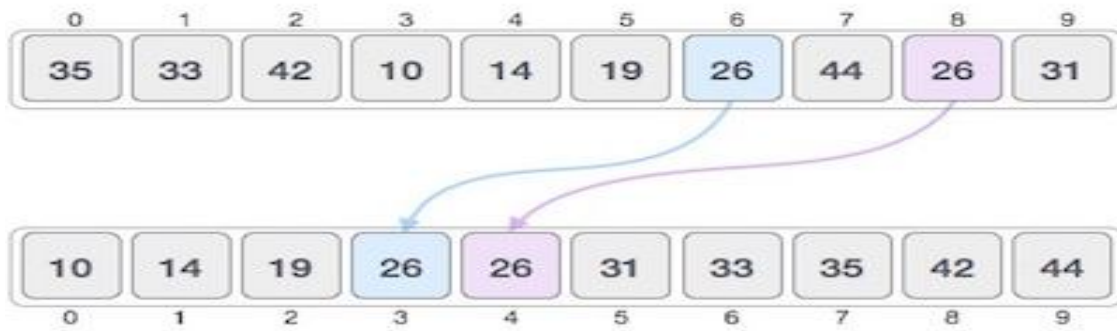
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.

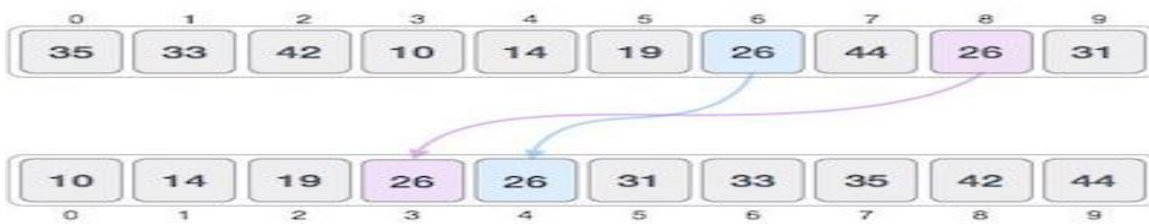
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Types of Sorting Algorithms:

9. Quick Sort
10. Bubble Sort
11. Merge Sort
12. Insertion Sort
13. Selection Sort
14. Heap Sort
15. Radix Sort
16. Bucket Sort

Algorithm	Best	Average	Worst
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$

Time Complexities of Sorting Algorithms:

Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$

Selection Sort:

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

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

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

How Selection Sort Works?

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

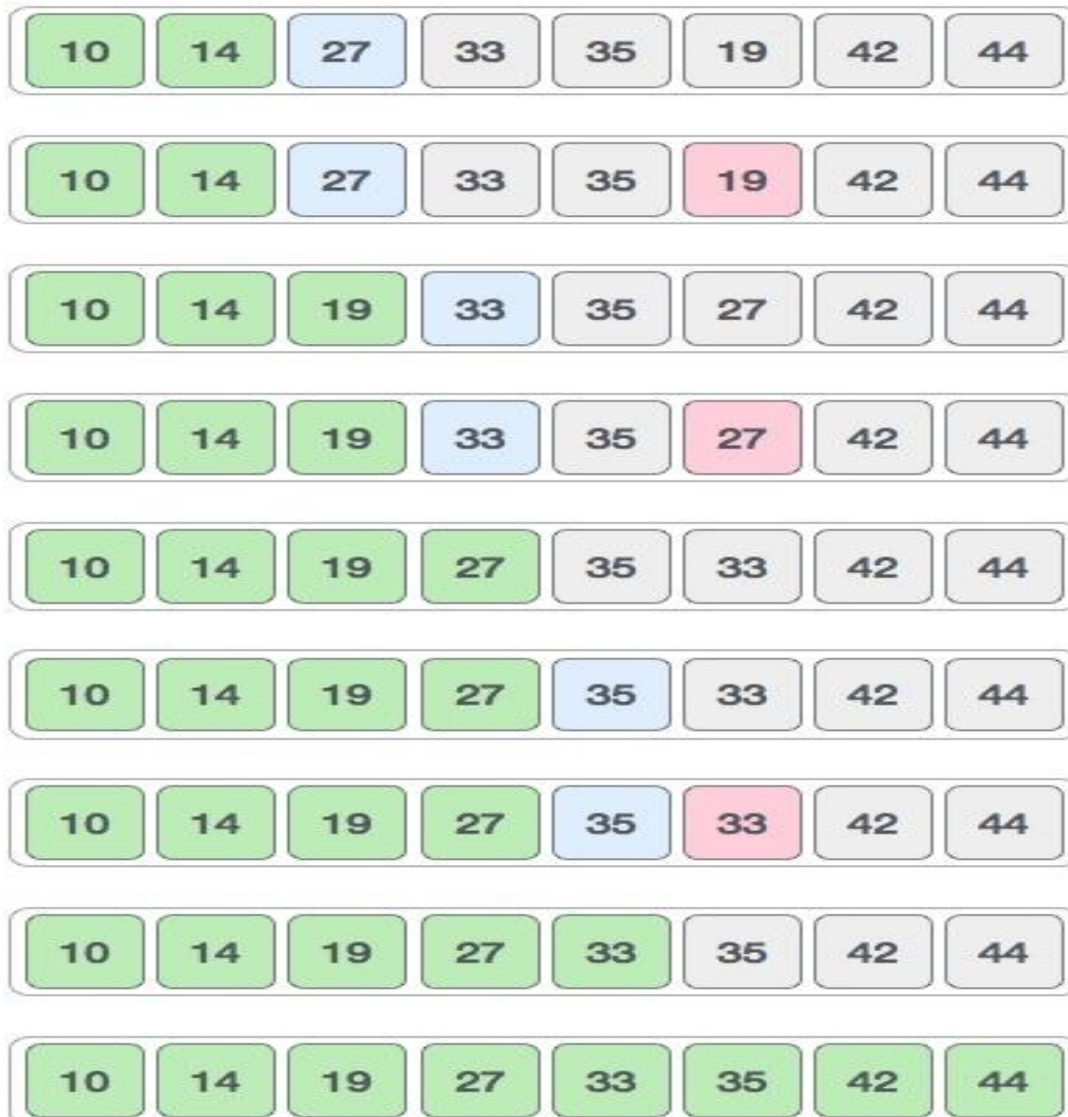


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –

**Algorithm:****Algorithm**selectionsort (a<array>, n)**Input:** a is array with n elements to be sort.**Output:**array elements in ascending order.

1. i = 0
2. while(i < n)
 - a) pos = i
 - b) j = i+1
 - c) while (j < n)
 - i) if (a[j] < a[pos])
 - A) pos = j
 - ii) end if
 - d) end loop

```

e) t = a[pos]
f) a[pos] = a[i]
g) a[i] = t
h) i= i+1

```

3. end loop

Endselectionsort

Source Code:

```

/* SELECTION SORT */
#include<stdio.h>
void selectionsort(int [],int);
void main()
{
    int a[20],n,i;
    clrscr();
    printf("Enter number of element to array");
    scanf("%d",&n);
    printf("\n Enter elements to array");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    selectionsort(a,n);
    printf("\n After sorting\n");
    for(i=0;i<n;i++)
    {
        printf("\t %d",a[i]);
    }
    getch();
}
void selectionsort(int a[],int n)
{
    int i,j,pos,t;
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(a[j]<a[pos])
            {
                pos=j;
            }
        }
        t=a[pos];
        a[pos]=a[i];
        a[i]=t;
    }
}

```

```
}

```

Output 1:

Enter number of element to array6

Enter elements to array

2 1 9 3 4 0

After sorting

0 1 2 3 4 9

Output 2:

Enter number of element to array4

Enter elements to array-6 -4 2 1

After sorting

-6 -4 1 2

VIVA Questions:

1. What is an in-place sorting algorithm?
2. What is the worst-case complexity of selection sort?
3. What is the disadvantage of selection sort?
4. What is the average case complexity of selection sort?
5. What is the advantage of selection sort over other sorting techniques?
6. What is the best-case complexity of selection sort?
7. Is selection sort having same efficiency as bubble sort?
8. Selection sort is suitable for which type of list?
9. How many comparisons are performed in the second pass of the selection sort algorithm?

Lab Assignment:

1. The given array is arr = {3,4,5,2,1}. Find number of iterations in bubble sort and selection sort respectively.
2. The given array is arr = {1,2,3,4,5}. (bubble sort is implemented with a flag variable) Find number of iterations in selection sort and bubble sort respectively.
3. Apply Selection sort on the following array. Show the array after each swap that takes place {30,60,20,50,40,10}.

Conclusion:

Designed program for Selection sort.

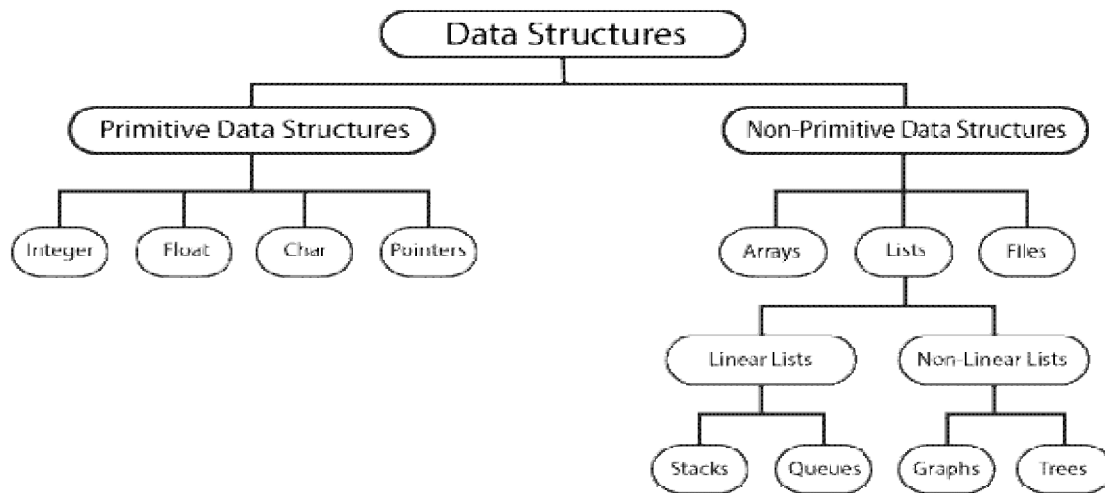
EXERCISE-III

3(i)

AIM:**Write a C program that uses functions to create a singly linked list.****DESCRIPTION:****Data Structure:**

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Types of Data Structures:**Types Of Data Structure****Linked List:**

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

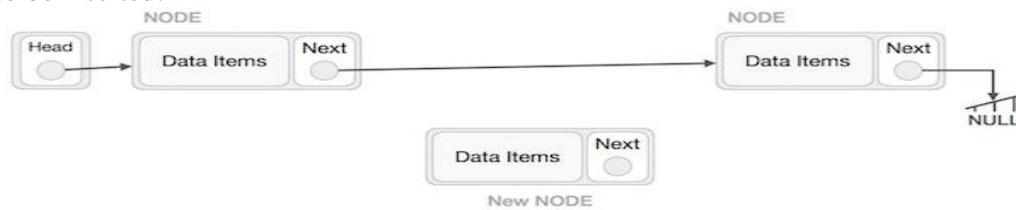
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Insertion Operation

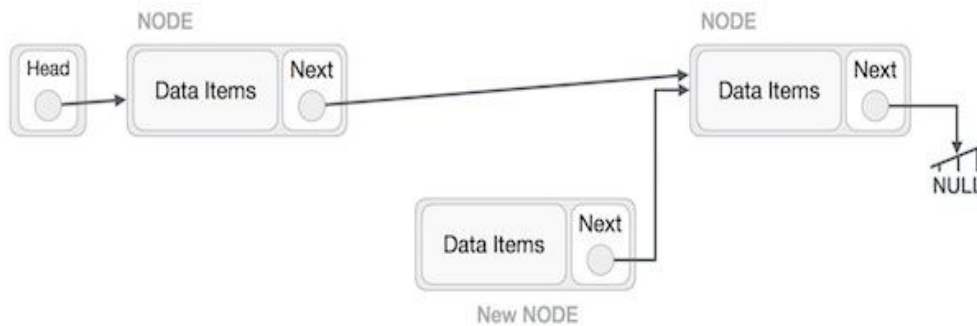
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C –

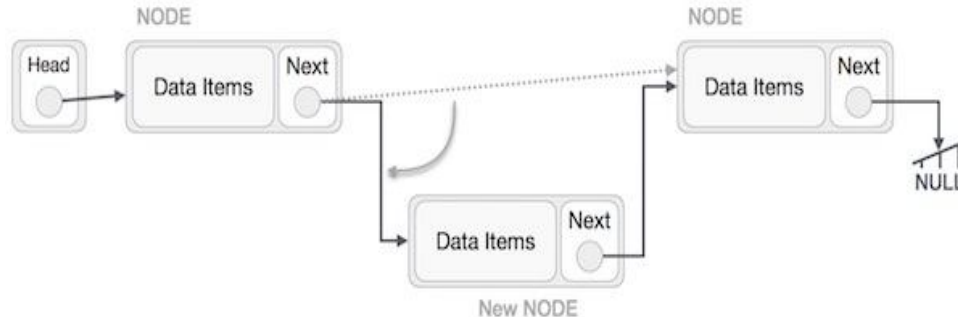
NewNode.next \rightarrow RightNode;

It should look like this –

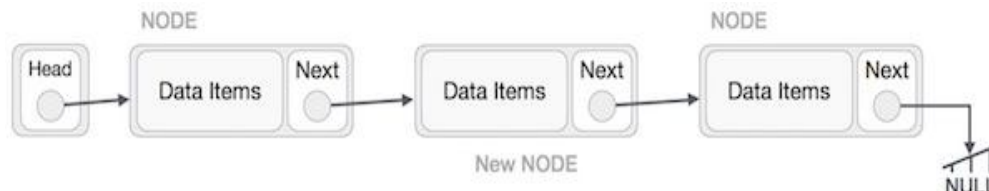


Now, the next node at the left should point to the new node.

LeftNode.next \rightarrow NewNode;



This will put the new node in the middle of the two. The new list should look like this –



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Algorithm:

Algorithm SLL_Create(header,x)

Input: header is a header node, x is data part of new node to be insert.

Output: SLL with new node inserted at beginning.

1. new=getnewnode()
2. if(new == NULL)
 - a) print(required node was not available in memory, so unable to process)
3. else
 - a) new.link=header.link /* 1 */
 - b) header.link = new /* 2 */
 - c) new.data=x
4. end if

End SLL_Create

Source Code:

```

/*Creation of a single linked list*/
#include<stdio.h>
#include<alloc.h>
void creation();
void traversal();
struct node
{
    int data;
    struct node *link;
} *ptr,*header,*new;
void main()
{
    int ch;
    clrscr();
    header->data=0;
    header->link=NULL;
    while(1)
    {
        printf("\n Enter the choice of operation 1.creation 2.traversal: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:creation();
                break;
            case 2:traversal();
                break;
            default:exit(0);
        }
    }
}
void creation()
{
    int item,x,key,pos;
    printf("enter the data value to insert");
    scanf("%d",&x);

```



```

        new=malloc(sizeof(struct node));
        new->link= header->link;
        header->link=new;
        new->data=x;
    }
void traversal()
{
    printf("\nelements in the list are");
    ptr=header;
    while(ptr->link!=NULL)
    {
        ptr=ptr->link;
        printf("\t%d",ptr->data);
    }
}

```

Output:

enter the choice of operation 1.Creation 2.traversal: 1
enter the data value to insert10

enter the choice of operation 1.Creation 2.traversal: 2
elements in the list are 10

enter the choice of operation 1.Creation 2.traversal: 1
enter the data value to insert20

enter the choice of operation 1.Creation 2.traversal: 1
enter the data value to insert30

enter the choice of operation 1.Creation 2.traversal: 2
elements in the list are 30 20 10

enter the choice of operation 1.Creation 2.traversal: 1
enter the data value to insert40

enter the choice of operation 1.Creation 2.traversal: 2
elements in the list are 40 30 20 10

enter the choice of operation 1.Creation 2.traversal:0

VIVA Questions:

1. What is a Data Structure?
2. What are linear and nonlinear data Structures?
3. What are the various operations that can be performed on different Data Structures?
4. How is an Array different from Linked List?
5. What is a linked list?
6. In what areas do data structures are applied?

Lab Assignment:

1. Write a Function to check if a singly linked list is palindrome.

2. Write a program toPrint reverse of a Linked List without actually reversing.
3. Write a program to remove duplicates from a sorted linked list
4. Write a program to move last element to front of a given Linked List

Conclusion:

Designed program for single linked list

3.(ii)

AIM:

Write a C program that uses functions to insert an element into a singly linked list

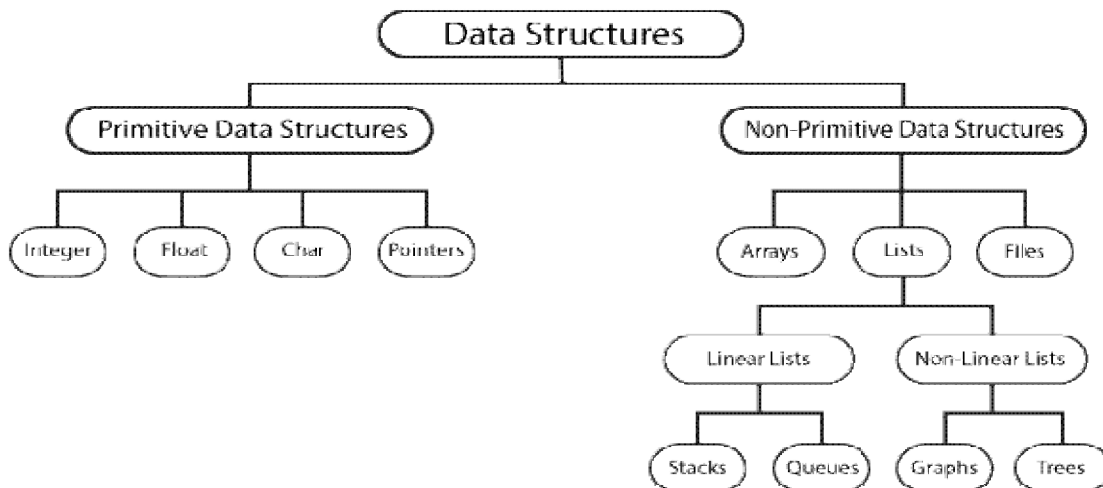
DESCRIPTION:

Data Structure:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Types of Data Structures:



Types Of Data Structure

Linked List:

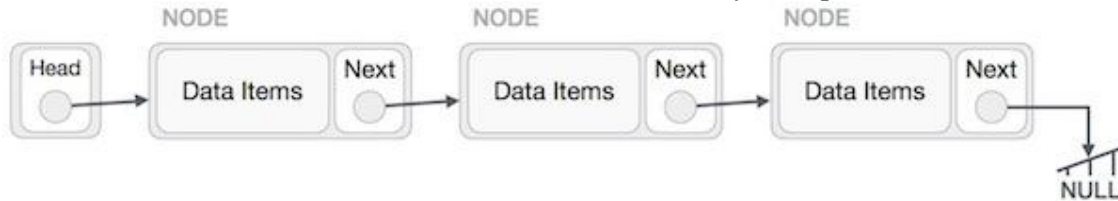
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Algorithm:

Algorithm SLL_Insert_Begin(header,x)

Input: header is a header node, x is data part of new node to be insert.

Output: SLL with new node inserted at beginning.

1. new=getnewnode()
2. if(new == NULL)
 - a) print(required node was not available in memory, so unable to process)
3. else
 - a) new.link=header.link /* 1 */
 - b) header.link=new /* 2 */
 - c) new.data=x
4. end if

End SLL_Insert_Begin**Algorithm SLL_Insert_Ending(header,x)****Input:**header is header node,x is data part of new node to be insert.**Output:**SLL with new node at ending.

```

1.new=getnewnode()
2.if(new ==NULL)
    a)print(Required node was not available in memory bank, so unable to
        process)
3.else
    a)ptr=header
    b)while(ptr.link!=NULL)
        i)ptr=ptr.link go to step(b)
    c)end loop
    d)ptr.link=new          /* 1 */
    e)new.link=NULL        /* 2 */
    f)new.data = x
4.end if

```

End_SLL_Insert_Ending**Algorithm SLL_Insert_ANY(header,x,key)****Input:**header is header node,x is data part of new node to be inserting, key is the data part of a node, after this node we want to insert new node.**Output:**SLL with new node at ANY

```

1.new= getnewnode()
2.if(new== NULL)
    a) print(required node was not available in memory bank,so unable to process)
3.else
    a) ptr=header
    b) while(ptr.data!=key and ptr.link!=NULL)
        i) ptr=ptr.link
    c) end loop
    d) if(ptr.link== NULL and ptr.data!=key)
        i) print(required node with data part as key value is not available, so unable to process)
    e) else
        ii) new.link=ptr.link          /* 1 */
        iii) ptr.link=new             /* 2 */
        iv) new.data=x
    f) end if
4. end if

```

End SLL_insert_ANY**Source Code:**

```

/*insertion of a single linked list*/
#include<stdio.h>
#include<alloc.h>
void insertion();
void traversal();

```

```
struct node
{
    int data;
    struct node *link;
} *ptr, *header, *new;

void main()
{
    int ch;
    clrscr();
    header->data=0;
    header->link=NULL;
    while(1)
    {
        printf("\nenter the choice of operation 1.insert 2.traversal: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:insertion();
                break;
            case 2:traversal();
                break;
            default:exit(0);
        }
    }
}

void insertion()
{
    int item,x,key,pos;
    printf("enter the data value to insert");
    scanf("%d",&x);
    new=malloc(sizeof(struct node));
    printf("enter the position for insertion 1.begining 2.ending 3.At any position");
    scanf("%d",&pos);
    /* insertion at beginning*/
    if(pos==1)
    {
        new->link= header->link;
        header->link=new;
        new->data=x;
    }
    /* insertion at ending*/
    else if(pos==2)
    {
        ptr=header;
        while(ptr->link!=NULL)
        {
```

```

        ptr=ptr->link;
    }
    ptr->link=new;
    new->link=NULL;
    new->data=x;
}
/* insertion at any pos*/
else if(pos==3)
{
    printf("\nenter key value");
    scanf("%d",&key);
    ptr=header;
    while(ptr->link!=NULL && ptr->data!=key)
    {
        ptr=ptr->link;
    }
    if(ptr->link==NULL)
    {
        /* Special case i.e. insertion of a node at any position that leads to insertion at end*/
        if(ptr->data==key)
        {
            new->link=ptr->link;
            ptr->link=new;
            new->data=x;
        }
        else
        {
            printf("\n Key not available");
        }
    }
    else
    {
        new->link=ptr->link;
        ptr->link=new;
        new->data=x;
    }
}
}
void traversal()
{
    printf("\nelements in the list are");
    ptr=header;
    while(ptr->link!=NULL)
    {
        ptr=ptr->link;
        printf("\t%d",ptr->data);
    }
}

```

```
}

```

Output:

```
enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert10
enter the position for insertion 1.begining 2.ending 3.At any position1
enter the choice of operation 1.insert 2.traversal: 2
elements in the list are    10
enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert20
enter the position for insertion 1.begining 2.ending 3.At any position2
enter the choice of operation 1.insert 2.traversal: 2
elements in the list are    10  20
enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert30
enter the position for insertion 1.begining 2.ending 3.At any position1
enter the choice of operation 1.insert 2.traversal: 2
elements in the list are    30  10  20
enter the choice of operation 1.insert 2.traversal: 1
enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert40
enter the position for insertion 1.begining 2.ending 3.At any position3
enter key value10
enter the choice of operation 1.insert 2.traversal: 2
elements in the list are    30  10  40  20
enter the choice of operation 1.insert 2.traversal:0
```

VIVA Questions:

1. What is a Data Structure?
2. What are linear and nonlinear data Structures?
3. What are the various operations that can be performed on different Data Structures?
4. How is an Array different from Linked List?
5. What is a linked list?
6. In what areas do data structures are applied?

Lab Assignment:

1. Write a Function to check if a singly linked list is palindrome.
2. Write a program toPrint reverse of a Linked List without reversing.
3. Write a program to remove duplicates from a sorted linked list
4. Write a program to move last element to front of a given Linked List

Conclusion:

Designed program for insertion into single linked list

3. (iii)

Aim:

Write a C program that uses functions to delete an element from a singly linked list

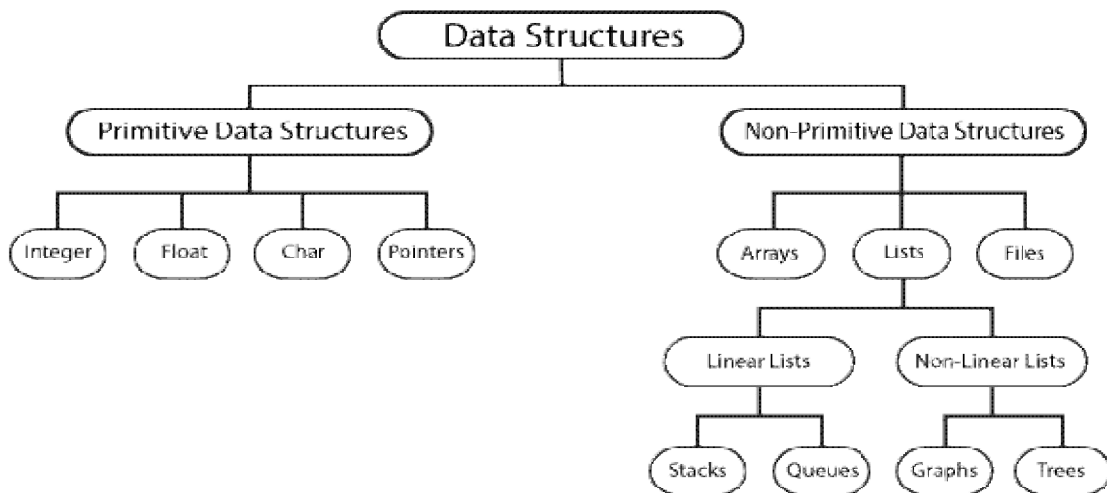
Description:**Data Structure:**

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are

widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Types of Data Structures:



Types Of Data Structure

Linked List:

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List:

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Algorithm:

Algorithm SLL_Delete_Begin(header)

Input: Header is a header node.

Output: SLL with node deleted at Beginning.

1. if(header.link == NULL)
 - a) print(SLL is empty,so unable to delete node from list)
2. else */*SLL is not empty*/*
 - a) ptr=header.link */*ptr points to first node into list*/*
 - b) header.link=ptr.link */* 1 */*
 - c) return(ptr) */*send back deleted node to memory bank*/*
3. end if

End SLL_Delete_Begin

Algorithm SLL_Delete_End (header)

Input:header is a header node

Output:SLL with node deleted at ending.

1. if(header.link == NULL)
 - a)print(SLL is empty, so unable to delete the node from list)
2. else /*SLL is not empty*/
 - a) ptr=header /*ptr initially points to header node*/
 - b)while(ptr.link !=NULL)
 - i)ptr1=ptr
 - ii) ptr=ptr.link /*go to step b*/
 - c)end loop
 - d)ptr1.link=NULL /* 1 */
 - e) return(ptr)
3. end if

End SLL_Delete_End

Algorithm SLL_Delete_ANY (header,key)

Input:header is a header node, key is the data part of the node to be delete.

Output:SLL with node deleted at Any position. i.e. Required element.

1. if(header.link == NULL)
 - a)print(SLL is empty, so unable to delete the node from list)
2. else /*SLL is not empty*/
 - a) ptr=header /*ptr initially points to header node*/
 - b)while(ptr.link!=NULL and ptr.data!=key)
 - i) ptr1 = ptr
 - ii) ptr=ptr.link go to step b
 - c)end loop
 - d) if(ptr.link == NULL and ptr.data!=key)
 - i) print(Required node with data part as key value is not available)
 - e) else /* node with data part as key value available */
 - i) ptr1.link= ptr.link /* 1 */
 - ii) return(ptr)
 - f) end if
3. end if

End SLL_Delete_ANY

Source Code:

```

/* SLL DELETION */
#include<stdio.h>
#include<malloc.h>
void create();
void delete();
void traverse();
struct node
{
    int data;
    struct node *link;
} *header,*new,*ptr,*ptr1;
void main()

```

```
{
    int ch;
    clrscr();
    header->data=0;
    header->link=NULL;
    while(1)
    {
        printf("\n\nEnter the choice of operation");
        printf("\n1.Create \t 2.Delete \t 3. Traversal\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: create();
                    break;
            case 2: delete();
                    break;
            case 3: traverse();
                    break;
            default:exit(0);
        }
    }
    getch();
}
void create()
{
    int x;
    new=malloc(sizeof(struct node));
    printf("\nEnter the data value");
    scanf("%d",&x);
    if(header->link==NULL)
    {
        header->link=new;
        new->link=NULL;
        new->data=x;
    }
    else
    {
        ptr=header;
        while(ptr->link!=NULL)
        {
            ptr=ptr->link;
        }
        ptr->link=new;
        new->link=NULL;
        new->data=x;
    }
}
```

```

void delete()
{
    int pos,x,key;
    printf("\nEnter the position for deletion");
    printf("\n 1.Begining 2.Ending\t3.At any Position\n");
    scanf("%d",&pos);
    if(pos==1) /*Deletion at beginning*/
    {
        ptr=header;
        if(ptr->link==NULL)
        {
            printf("\n SLL is empty");
        }
        else
        {
            ptr1=ptr;
            ptr=ptr->link;
            ptr1->link=ptr->link; /* Address of second node i.e link part of first node is copied to link part of header node*/
            printf("\nNode deleted is %d",ptr->data);
            free(ptr);
        }
    }
    else if (pos==2) /* Deletion at Ending */
    {
        ptr=header;
        if(ptr->link==NULL)
        {
            printf("\n SLL is empty, unable to perform deletion");
        }
        else
        {
            ptr1=ptr;
            ptr=ptr->link;
            while(ptr->link!=NULL)
            {
                ptr1=ptr;
                ptr=ptr->link;
            }
            ptr1->link=NULL; /* Last but one node link part is replaced with NULL */
            printf("\nDeleted Node is %d",ptr->data);
            free(ptr);
        }
    }
    else if(pos==3) /* Deletion at any position */
    {
        ptr=header;
        if(ptr->link==NULL)

```

```

        {
            printf("\n SLL is empty, unable to perform deletion");
        }
    else
    {
        printf("\nEnter the data value to delete");
        scanf("%d",&key);
        while(ptr->data!=key && ptr->link!=NULL)
        {
            ptr1=ptr;
            ptr=ptr->link;        /* Move to the next node*/
        }

        if(ptr->link==NULL)
        {
            printf("\n Node with key was not found");
        }
        else
        {
            ptr1->link=ptr->link;
            printf("\nDeleted node is %d",ptr->data);
            free(ptr);
        }
    }
}

void traverse()
{
    printf("\n Elements in the list are:\n");
    ptr=header;
    while(ptr->link!=NULL)
    {
        ptr=ptr->link;
        printf("\t%d",ptr->data);
    }
}

```

Output:

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value10

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value20

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value30

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value40

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value50

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value60

Enter the choice of operation

1.Create 2.Delete 3. Traversal

3

Elements in the list are:

10 20 30 40 50 60

Enter the choice of operation

1.Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

1

Node deleted is 10

Enter the choice of operation

1.Create 2.Delete 3. Traversal

3

Elements in the list are:

20 30 40 50 60

Enter the choice of operation

1.Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

2

Deleted Node is 60

Enter the choice of operation

1.Create 2.Delete 3. Traversal

3

Elements in the list are:

20 30 40 50

Enter the choice of operation

1.Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

3

Enter the data value to delete30

Deleted node is 30

Enter the choice of operation

1.Create 2.Delete 3. Traversal

3

Elements in the list are:

20 40 50

Enter the choice of operation

1.Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

3

Enter the data value to delete65

Node with key was not found

Enter the choice of operation

1.Create 2.Delete 3. Traversal

0

VIVA Questions:

1. What is a Data Structure?
2. What are linear and nonlinear data Structures?
3. What are the various operations that can be performed on different Data Structures?
4. How is an Array different from Linked List?
5. What is a linked list?
6. In what areas do data structures are applied?

Lab Assignment:

1. Write a Function to check if a singly linked list is palindrome.

2. Write a program to Print reverse of a Linked List without reversing.
3. Write a program to remove duplicates from a sorted linked list
4. Write a program to move last element to front of a given Linked List

Conclusion:

Designed program for deletion of element into single linked list

EXERCISE-IV

4. (i)

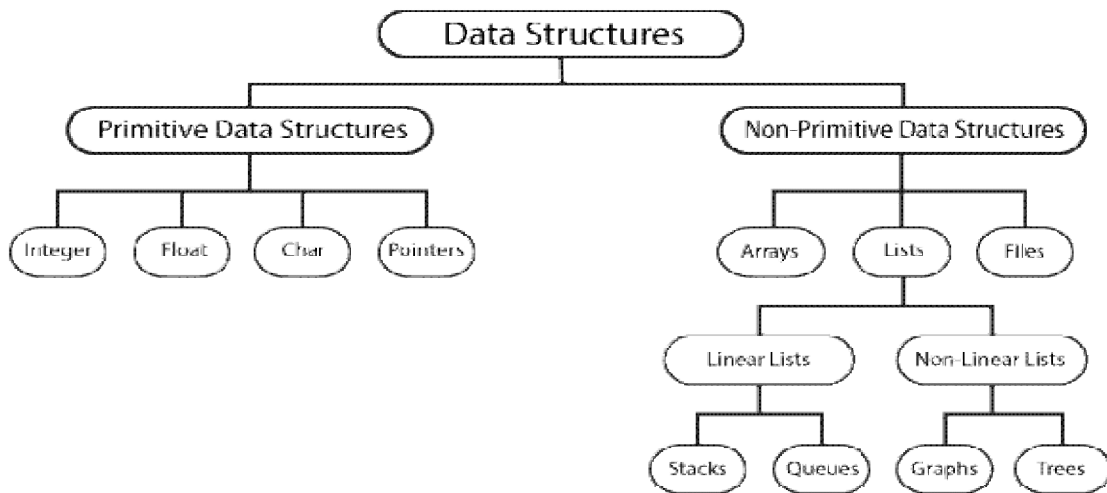
AIM:

Write a C program that uses functions to Create a Doubly linked list.

DESCRIPTION:**Data Structure:**

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Types of Data Structures:**Types Of Data Structure****Linked List:**

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

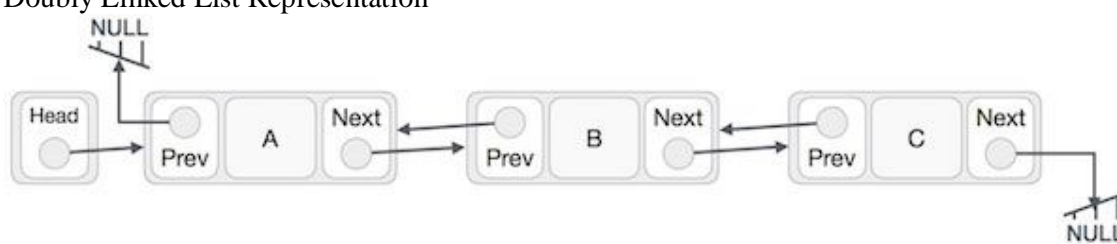
- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Doubly Linked List:

It is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Delete Last** – Deletes an element from the end of the list.
- **Insert After** – Adds an element after an item of the list.

- Delete – Deletes an element from the list using the key.
- Display forward – Displays the complete list in a forward manner.
- Display backward – Displays the complete list in a backward manner.

Algorithm:**Algorithm DLL_Creation(header,X)****Input:** header is a header node.**Output:** DLL with new node at begin.

- 1.new=getnewnode()
- 2.if(new= = NULL)
 - a) print(required node is not available in memory)
- 3.else
 - a)ptr=header.rlink
 - b)new.rlink=ptr /* 1 */
 - c)new.llink=header /* 2 */
 - d)header.rlink=new /* 3 */
 - e) ptr.llink=new /* 4 */
4. end if

End DLL_Creation**Program:**

```

/* Creation of double linked list*/
#include<stdio.h>
#include<alloc.h>
void creation();
void traversal();
struct node
{
    int data;
    struct node *llink;
    struct node *rlink;
} *ptr,*ptr1,*header,*new1;
void main()
{
    int ch;
    clrscr();
    header->llink=NULL;
    header->rlink=NULL;
    header->data=0;
    while(1)
    {
        printf("\n Enter the choice of operation 1.creation 2.traversal: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:creation();
                break;

```

```

        case 2:traversal();
            break;
        default:exit(0);
    }
}
}
void creation()
{
    int item,x,key,pos;
    printf("enter the data value to insert");
    scanf("%d",&x);
    new1=malloc(sizeof(struct node));
    ptr=header;
    ptr1=ptr->rlink;
    new1->llink=header;
    new1->rlink= ptr1;
    ptr->rlink=new1;
    ptr1->llink=new1;
    new1->data=x;
}
void traversal()
{
    printf("\nelements in the list are");
    ptr=header;
    while(ptr->rlink!=NULL)
    {
        ptr=ptr->rlink;
        printf("\t%d",ptr->data);
    }
}

```

Output:

enter the choice of operation 1.creation 2.traversal: 1
enter the data value to insert10

enter the choice of operation 1.creation 2.traversal: 1
enter the data value to insert20

enter the choice of operation 1.creation 2.traversal: 1
enter the data value to insert30

enter the choice of operation 1.creation 2.traversal: 1
enter the data value to insert40

enter the choice of operation 1.creation 2.traversal: 2

elements in the list are 40 30 20 10

enter the choice of operation 1.creation 2.traversal:0

VIVA Questions:

1. What is an abstract data type?
2. What are linear and non linear data Structures?
3. What are the various operations that can be performed on double linked list?
4. How is an Array different from Linked List?
5. Define double linked list?
6. What are the applications of double linked list?

Lab Assignment:

1. Write a Function to check if a double linked list is palindrome.
2. Write a program toPrint reverse of a double Linked List without reversing.
3. Write a program to remove duplicates from a sorted double linked list
4. Write a program to move last element to front of a given double Linked List

Conclusion:

Designed program for concept of double linked list

4. (ii)

AIM:

Write a C program that uses functions to Insert an element into a Doubly linked list.

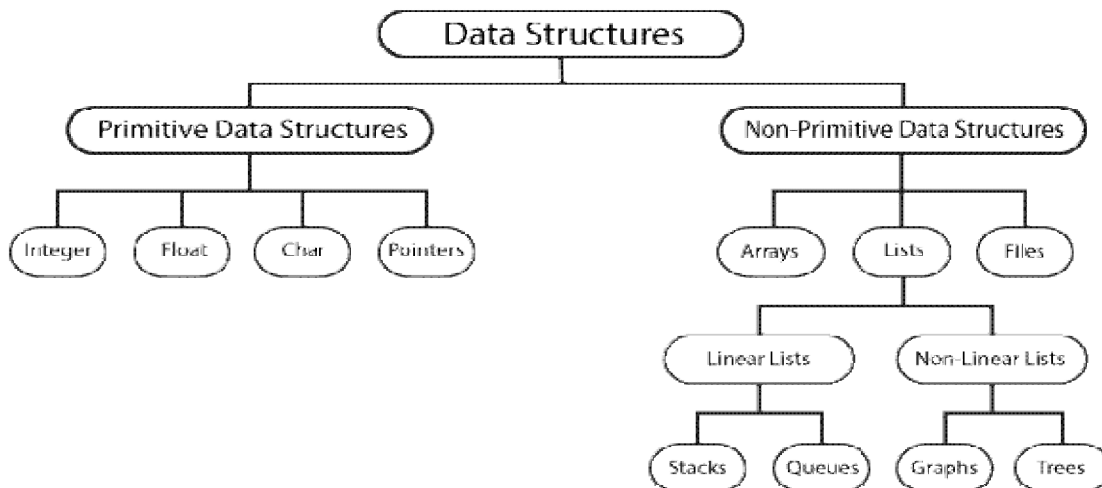
DESCRIPTION:

Data Structure:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Types of Data Structures:



Types Of Data Structure

Linked List:

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

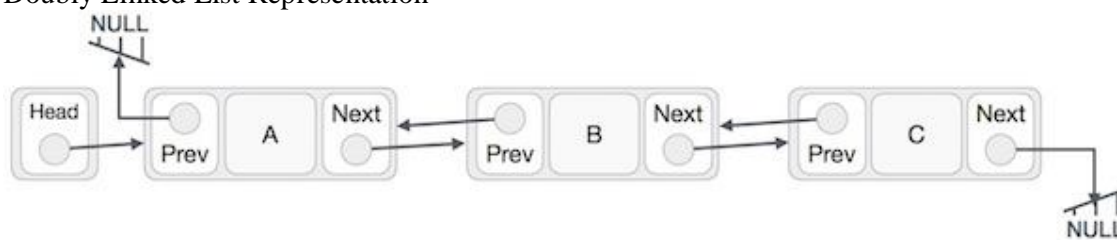
- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Doubly Linked List:

It is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations:

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Delete Last** – Deletes an element from the end of the list.
- **Insert After** – Adds an element after an item of the list.

- Delete – Deletes an element from the list using the key.
- Display forward – Displays the complete list in a forward manner.
- Display backward – Displays the complete list in a backward manner.

Algorithm:**Algorithm DLL_insertion_Begin(header,X)****Input:** header is a header node.**Output:** DLL with new node at begin.

```

1.new=getnewnode()
2.if(new= = NULL)
    a) print(required node is not available in memory)
3.else
    a)ptr=header.rlink
    b)new.rlink=ptr          /* 1 */
    c)new.llink=header      /* 2 */
    d)header.rlink=new     /* 3 */
    e) ptr.llink=new       /* 4 */
4. end if

```

End DLL_insertion_Begin**Algorithm DLL_Insert_Ending(Header,x)****Input:**Header is the header node,x is the data part of new node to be inserted.**Output:**DLL with new node inserted at the ending.

```

1.new=getnewnode()
2.if(new= = NULL)
    a)print(Required node was not available)
3.else
    a)ptr=header
    b)while(ptr.rlink != NULL)
        i)ptr=ptr.rlink      goto step(b)
    c)end while loop
    d)ptr.rlink=new          /* 1 */
    e)new.llink=ptr          /* 2 */
    f)new.rlink=NULL        /* 3 */
    g)new.data=x
4.end if

```

End DLL_Insertion_Ending**Algorithm DLL_Insertion_ANY(header,x,key)****Input:**Header is a header node, key is the data part of a node,after that node new node is inserted, x is data part of new node to be insert.**Output:**DLL with new node inserted after the node with data part as key value

```

1.new=getnewnode()
2.if(new= = NULL)
    a)print (required node is not available in memory)
3.else
    a)ptr=header
    b)while(ptr.data!=key and ptr.rlink!=NULL)

```

```

        i)ptr=ptr.rlinkgo to step(b)
    c)end loop
    d)if(ptr.rlink ==NULL and ptr.data != key)
        i)print(required node with key value was not available)
    e)else
        i)ptr1=ptr.rlink
        ii) new.rlink=ptr1          /* 1 */
        iii)new.llink=ptr          /* 2 */
        iv)ptr.rlink=new          /* 3 */
        v)ptr1.llink=new          /* 4 */
        vi)new.data=x
    f)end if
4. end if

```

End DLL_Insertion_ANY

Source Code:

```

/*insertion of a node into double linked list*/
#include<stdio.h>
#include<alloc.h>
void insertion();
void traversal();
struct node
{
    int data;
    struct node *llink;
    struct node *rlink;
} *ptr,*ptr1,*header,*new1;

void main()
{
    int ch;
    clrscr();
    header->llink=NULL;
    header->rlink=NULL;
    header->data=0;
    while(1)
    {
        printf("\nenter the choice of operation 1.insert 2.traversal: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:insertion();
                break;
            case 2:traversal();
                break;
            default:exit(0);
        }
    }
}

```

```

}
void insertion()
{
    int item,x,key,pos;
    printf("enter the data value to insert");
    scanf("%d",&x);
    new1=malloc(sizeof(struct node));
    printf("enter the position for insertion 1.begining 2.ending 3.At any position");
    scanf("%d",&pos);
    /* insertion at beginning*/
    if(pos==1)
    {
        ptr=header;
        ptr1=ptr->rlink;
        new1->llink=header;
        new1->rlink= ptr1;
        ptr->rlink=new1;
        ptr1->llink=new1;
        new1->data=x;
    }
    /* insertion at ending*/
    else if(pos==2)
    {
        ptr=header;
        while(ptr->rlink!=NULL)
        {
            ptr=ptr->rlink;
        }
        ptr->rlink=new1;
        new1->llink=ptr;
        new1->rlink=NULL;
        new1->data=x;
    }
    /* insertion at any pos*/
    else if(pos==3)
    {
        printf("\nenrter key value");
        scanf("%d",&key);
        ptr=header;
        while(ptr->rlink!=NULL && ptr->data!=key)
        {
            ptr=ptr->rlink;
        }
        if(ptr->rlink==NULL && ptr->data !=key)
        {
            printf("\n Key not available");
        }
    }
}

```

```

        else if(ptr->rlink==NULL && ptr->data ==key)
        {
            ptr->rlink=new1;
            new1->llink=ptr;
            new1->data=x;
        }
        else
        {
            ptr1=ptr->rlink;
            new1->llink=ptr;
            new1->rlink=ptr1;
            ptr->rlink=new1;
            ptr1->llink=new1;
            new1->data=x;
        }
    }
}
void traversal()
{
    printf("\nelements in the list are");
    ptr=header;
    while(ptr->rlink!=NULL)
    {
        ptr=ptr->rlink;
        printf("\t%d",ptr->data);
    }
}

```

Output:

```

enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert10
enter the position for insertion 1.begining 2.ending 3.At any position1

enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert20
enter the position for insertion 1.begining 2.ending 3.At any position1

enter the choice of operation 1.insert 2.traversal: 2

elements in the list are    20    10

enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert30
enter the position for insertion 1.begining 2.ending 3.At any position2

enter the choice of operation 1.insert 2.traversal: 2
elements in the list are    20    10    30

```

enter the choice of operation 1.insert 2.traversal: 1
enter the data value to insert:55
enter the position for insertion 1.beginning 2.ending 3.At any position:3
enter key value:10

enter the choice of operation 1.insert 2.traversal: 2
elements in the list are 20 10 55 30

enter the choice of operation 1.insert 2.traversal:0

VIVA Questions:

7. What is an abstract data type?
8. What are linear and non-linear data structures?
9. What are the various operations that can be performed on a double linked list?
10. How is an array different from a linked list?
11. Define a double linked list?
12. What are the applications of a double linked list?

Lab Assignment:

5. Write a function to check if a double linked list is a palindrome.
6. Write a program to print the reverse of a double linked list without reversing.
7. Write a program to remove duplicates from a sorted double linked list.
8. Write a program to move the last element to the front of a given double linked list.

Conclusion:

Designed program for the concept of a double linked list

4. (iii)

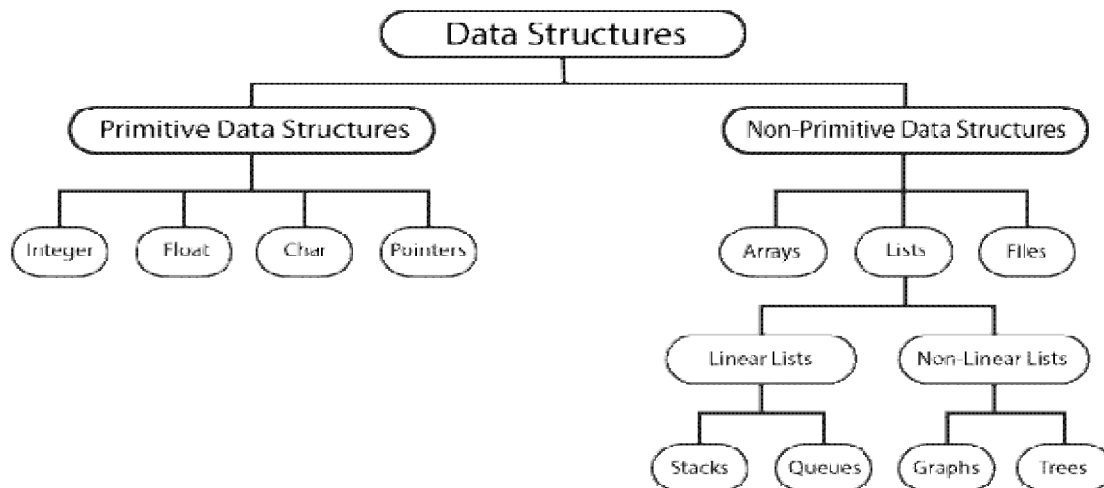
AIM:

Write a C program that uses functions to Delete an element from a Doubly linked list.

DESCRIPTION:**Data Structure:**

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Types of Data Structures:

Types Of Data Structure

Linked List:

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

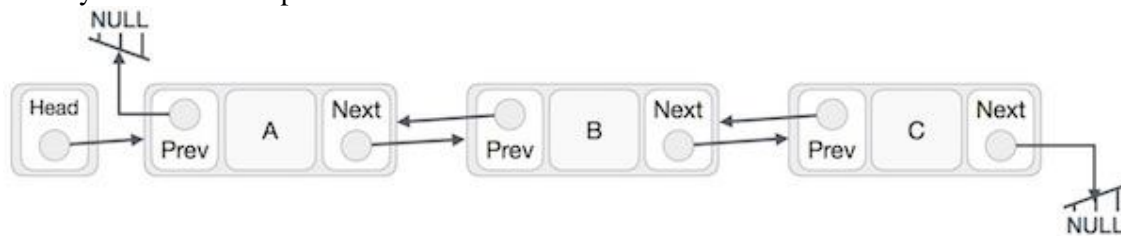
- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Doubly Linked List:

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- Link – Each link of a linked list can store a data called an element.
- Next – Each link of a linked list contains a link to the next link called Next.
- Prev – Each link of a linked list contains a link to the previous link called Prev.
- LinkedList – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by a list.

- Insertion – Adds an element at the beginning of the list.
- Deletion – Deletes an element at the beginning of the list.
- Insert Last – Adds an element at the end of the list.
- Delete Last – Deletes an element from the end of the list.
- Insert After – Adds an element after an item of the list.
- Delete – Deletes an element from the list using the key.
- Display forward – Displays the complete list in a forward manner.
- Display backward – Displays the complete list in a backward manner.

Algorithm:

Algorithm DLL_Deletion_Begin(header)

Input: header is a header node

Output: DLL with node deleted at begin

1. if(header.rlink == NULL)
 - a) print(DLL is empty, not possible to perform deletion operation)
2. else
 - a) ptr=header.rlink
 - b) ptr1=ptr.rlink
 - c) header.rlink=ptr1 /* 1 */
 - d) ptr1.llink=header /* 2 */
 - e) return(ptr)
3. end if

End DLL_Deletion_Begin

Algorithm DLL_Deletion_End(header)

Input: header is a header node.

Output: DLL with deleted node at ending.

1. if(header.rlink == NULL)
 - a) print(DLL is empty, not possible to perform deletion operation)
2. else
 - a) ptr=header
 - b) while(ptr.rlink != NULL)
 - i) ptr1=ptr
 - ii) ptr=ptr.rlink
 - c) end loop
 - d) ptr1.rlink=NULL /* 1 */
 - e) return(ptr)
3. end if

End DLL_Deletion_Ending

Algorithm DLL_Deletion_Any(header,key)

Input: header is header node, key is the data part of a node to be delete.

Output: DLL without node as data part is key value.

1. if(header.rlink == NULL)
 - a) print(DLL is empty, not possible for deletion operation)
2. else
 - a) ptr=header
 - b) while(ptr.data != key and ptr.rlink != NULL)
 - i) ptr=ptr.rlink
 - c) end loop
 - d) if(ptr.rlink == NULL and ptr.data != key)
 - i) print(required node was not available in list)
 - e) else
 - i) ptr1 = ptr.llink
 - ii) ptr2 = ptr.rlink
 - iii) ptr1.rlink = ptr2 /* 1 */
 - iv) ptr2.rlink = ptr1 /* 2 */
 - f) end if
3. end if

End DLL_Deletion_Any

Source Code:

```
/* DELETION OF A NODE FROM DLL */
#include<stdio.h>
#include<malloc.h>
void create();
void delete();
void traverse();
struct node
{
    int data;
    struct node *llink,*rlink;
}*header,*new1,*ptr,*ptr1,*ptr2;

void main()
{
    int ch;
    clrscr();
    header->data=0;
    header->llink=NULL;
    header->rlink=NULL;

    while(1)
    {
        printf("\n\nEnter the choice of operation");
        printf("\n1.Create \t 2.Delete \t 3. Traversal\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:create();
                break;
            case 2:delete();
                break;
            case 3:traverse();
                break;
            default:exit(0);
        }
    }
    getch();
}

void create()
{
    int x;
    new1=malloc(sizeof(struct node));
    printf("\nEnter the data value");
    scanf("%d",&x);
    if(header->rlink==NULL)
```

```

    {
        header->rlink=new 1;
        new 1->llink=header;
        new 1->rlink=NULL;
        new 1->data=x;
    }
else
{
    ptr=header;
    while(ptr->rlink!=NULL)
    {
        ptr=ptr->rlink;
    }
    ptr->rlink=new 1;
    new 1->llink=ptr;
    new 1->rlink=NULL;
    new 1->data=x;
}
}
void delete()
{
    int pos,x,key;
    printf("\nEnter the position for deletion");
    printf("\n 1.Begining 2.Ending\t3.At any Position\n");
    scanf("%d",&pos);
    if(pos==1)          /*Deletion at beginning*/
    {
        ptr=header;
        if(ptr->rlink==NULL)
        {
            printf("\n DLL is empty");
        }
        else
        {
            ptr1=ptr;
            ptr=ptr->rlink;
            ptr2=ptr->rlink;
            ptr1->rlink=ptr2;          /* Address of second node is copied to right link part of header node*/
            ptr2->llink=ptr1;        /* Address of header node is copied to left link part of second node*/
            printf("\nNode deleted is %d",ptr->data);
            free(ptr);
        }
    }
    else if (pos==2) /* Deletion at Ending */
    {
        ptr=header;
        if(ptr->rlink==NULL)

```

```

    {
        printf("\n DLL is empty, unable to perform deletion");
    }
    else
    {
        while(ptr->rlink!=NULL)
        {
            ptr1=ptr;
            ptr=ptr->rlink;
        }
        ptr1->rlink=NULL; /* Last but one node link part is replaced with NULL */
        printf("\nDeleted Node is %d",ptr->data);
        free(ptr);
    }
}
else if(pos==3) /* Deletion at any position */
{
    ptr=header;
    if(ptr->rlink==NULL)
    {
        printf("\n DLL is empty, unable to perform deletion");
    }
    else
    {
        printf("\n Enter the data value to delete");
        scanf("%d",&key);
        while(ptr->data!=key && ptr->rlink!=NULL)
        {
            ptr=ptr->rlink; /* Move to the next node*/
        }
        if(ptr->rlink==NULL&& ptr->data!=key)
        {
            printf("\n Node with key was not found");
        }
        else if(ptr->rlink==NULL && ptr->data==key)
        {
            ptr1=ptr->llink;
            ptr1->rlink=NULL;
            printf("\nDeleted node is %d ",ptr->data);
            free(ptr);
        }
        else
        {
            ptr1=ptr->llink;
            ptr2=ptr->rlink;
            ptr1->rlink=ptr2; /* Next node address is copied to the rlink part of previous node */
            ptr2->llink=ptr1; /* Previous node address is copied to the llink part of next node */
        }
    }
}

```

```

                printf("\n Deleted node is %d",ptr->data);
                free(ptr);
            }
        }
    }
}
void traverse()
{
    printf("\n Elements in the list are:\n");
    ptr=header;
    while(ptr->rlink!=NULL)
    {
        ptr=ptr->rlink;
        printf("\t%d",ptr->data);
    }
}

```

Output:

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value10

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value20

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value30

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value40

Enter the choice of operation

1.Create 2.Delete 3. Traversal

1

Enter the data value50

Enter the choice of operation

1. Create 2. Delete 3. Traversal

1

Enter the data value60

Enter the choice of operation

1.Create 2. Delete 3. Traversal

3

Elements in the list are:

10 20 30 40 50

Enter the choice of operation

1.Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

1

Node deleted is 10

Enter the choice of operation

1.Create 2. Delete 3. Traversal

3

Elements in the list are:

20 30 40 50 60

Enter the choice of operation

1.Create 2.Delete 3. Traversal

3

Elements in the list are:

20 30 40 50 60

Enter the choice of operation

1.Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

2

Deleted Node is 60

Enter the choice of operation

1. Create 2.Delete 3. Traversal

3

Elements in the list are:

20 30 40 50

Enter the choice of operation

1. Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

3

Enter the data value to delete30

Deleted node is 30

Enter the choice of operation

1.Create 2.Delete 3. Traversal

Elements in the list are:

20 40 50

Enter the choice of operation

1.Create 2.Delete 3. Traversal

2

Enter the position for deletion

1.Begining 2.Ending 3.At any Position

55

Enter the choice of operation

1.Create 2.Delete 3. Traversal

0

VIVA Questions:

1. What is a abstract data type?
2. What are linear and nonlinear data Structures?
3. What are the various operations that can be performed on double linked list?
4. What is a memory efficient double linked list?
5. Define double linked list?
6. What are the applications of double linked list?
7. How do you calculate the pointer difference in a memory efficient double linked list?

Lab Assignment:

1. Write a program to find size of Doubly Linked List
2. Write a function in double linked list to perform the following operations
 - a. Return the element at the tail of the list but do not remove it
 - b) Return the element at the tail of the list and remove it from the list
 - c) Return the last but one element rom the list but do not remove it
 - d) Return the last but one element at the tail of the list and remove it from the list
3. Write a program to Insert value in sorted way in a sorted doubly linked list
4. Write a program to Print reverse of a double Linked List without reversing.
5. Write a program to remove duplicates from a unsorted double linked list
6. Write a program to move last element to front of a given double Linked List

Conclusion:

Designed program for deletion of element in double linked list

EXERCISE-V

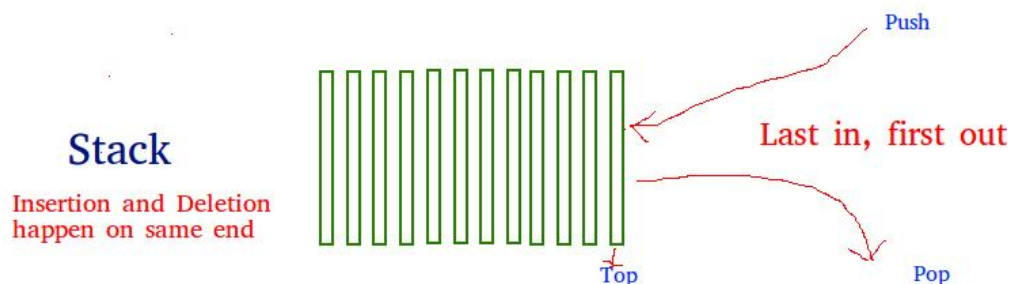
5.

AIM:**Write a C program that implement stack (its operations) using arrays.****DESCRIPTION:**

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

Mainly the following three basic operations are performed in the stack:

- Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Peek or Top: Returns top element of stack.
- isEmpty: Returns true if stack is empty, else false.



How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Algorithm:**Algorithm Stack_PUSH(item)****Input:** item is new item to push into stack

Output:pushing new item into stack at top whenever stack is not full.

- 1.if(top>=size)
 - a) print(stack is full, not possible to perform push operation)
- 2.else
 - a) top=top+1
 - b) s[top]=item
3. end if

End Stack_PUSH

Algorithm Stack_POP()

Input: Stack with some elements.

Output:item deleted at top most end.

- 1.if(top<1)
 - a) print(stack is empty not possible to pop)
- 2.else
 - a) item=s[top]
 - b) top=top-1
 - c) print(deleted item)
3. end if

End Stack_POP

Source Code:

```
#include<stdio.h>
#define size 5
int top=0,s[5];
void push(int);
void pop();
void traverse();

void main()
{
    int i,item,ch;
    clrscr();
    while(1)
    {
        printf("\n Enter your choice 1.push 2.pop 3.traverse");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\n Enter the item ");
                    scanf("%d",&item);
                    push(item);
                    break;
            case 2: pop();
                    break;
            case 3: traverse();
                    break;
        }
    }
}
```



```
                default:exit(0);
            }
        }
    }
void push(int item)
{
    if(top==size)
    {
        printf("\n stack is full ");
    }
    else
    {
        top=top+1;
        s[top]=item;
    }
}
void pop()
{
    if(top < 1)
    {
        printf("\n Stack is empty");
    }
    else
    {
        printf("Poped item is %d",s[top]);
        top=top-1;
    }
}
void traverse()
{
    int i;
    if(top < 1)
    {
        printf("\n Stack is empty");
    }
    else
    {
        printf("\n Items of stack are ");
        for(i=1;i<=top;i++)
        {
            printf("%d\t",s[i]);
        }
    }
}
}
```

Output:

Enter your choice 1.push 2.pop 3.traverse
1
Enter the item 10

Enter your choice 1.push 2.pop 3.traverse1
Enter the item 20

Enter your choice 1.push 2.pop 3.traverse1
Enter the item 30

Enter your choice 1.push 2.pop 3.traverse3
Items of stack are 10 20 30

Enter your choice 1.push 2.pop 3.traverse1
Enter the item 40

Enter your choice 1.push 2.pop 3.traverse3
Items of stack are 10 20 30 40

Enter your choice 1.push 2.pop 3.traverse2
Popped item is 40

Enter your choice 1.push 2.pop 3.traverse3
Items of stack are 10 20 30

Enter your choice 1.push 2.pop 3.traverse2
Poped item is 30

Enter your choice 1.push 2.pop 3.traverse3
Items of stack are 10 20

Enter your choice 1.push 2.pop 3.traverse1
Enter the item 55

Enter your choice 1.push 2.pop 3.traverse3
Items of stack are 10 20 55

Enter your choice 1.push 2.pop 3.traverse0

\

VIVA Questions:

1. What is an abstract data type?
2. What are linear and non linear data Structures?
3. Define stack?
4. What are the various operations that can be performed on stack?
5. What are the applications of stacks?

Lab Assignment:

1. Write a program to reverse the string using stacks

2. Write a program to check whether the given string is palindrome or not
3. Write a program to evaluate a postfix expression using stacks
4. Write a program to find whether the equation is perfectly balanced or not using stacks

Conclusion:

Designed program for implementation of stacks using arrays

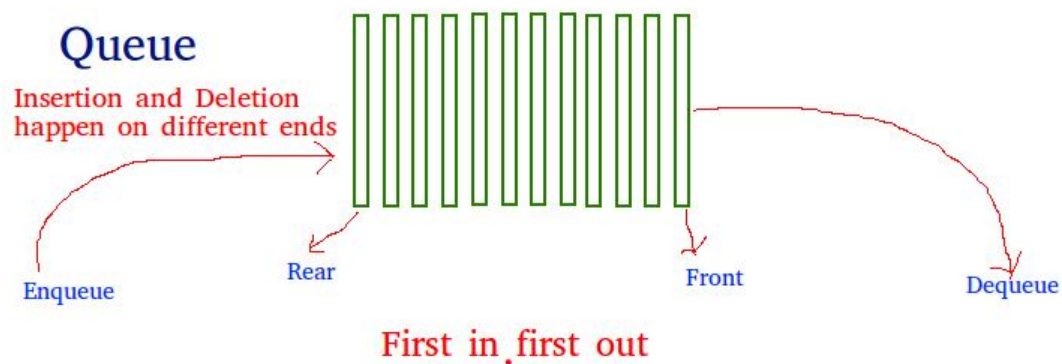
6.

AIM:

Write C programs that implement Queue (its operations) using linked lists.

Description:

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Algorithm:

Algorithm Enqueue_LL(item)

Input: item is new item to be insert.

Output: new item i.e new node is inserted at rear end.

```

1.new=getnewnode()
2.if(new ==NULL)
    a)print(required node is not available in memory)
3.else
    a)if(front ==NULL and rear ==NULL)    /* Q is EMPTY */
        i)header.link=new
        ii)new.link=NULL
        iii)front=new
        iv)rear=new
        v)new.data=item
    b)else                                /* Q is not EMPTY */
        i)rear.link=new                    /* 1 */
        ii)new.link=NULL                  /* 2 */
        iii)rear=new                      /* 3 */
        iv)new.data=item
    c)end if
4.end if

```

End_Enqueue_LL

Algorithm Dequeue_LL()

Input: Queue with some elements

Output: Element is deleted at front end if queue is not empty.

```

1.if(front==NULL and rear==NULL)
    a)print(queue is empty,not possible to perform dequeue operation)
2.else
    a)if(front==rear)                    /* Q has only one element */
        i)header.link=NULL
        ii)item=front.data
        iii)front=NULL
        iv)rear=NULL
    b)else                                /* Q has more than one element */
        i)header.link=front.link        /* 1 */
        ii)item=front.data
        iii)free(front)
        iv)front=header.link           /* 2 */
    c)end if
    d)print(deleted element is item)
3.end if

```

End_Dequeue_LL

Source Code:

```

/* QUEUE OPERATIONS USING LINKED LISTS */
#include<stdio.h>
#include<malloc.h>
void enqueue();

```

```
void dequeue();
void traverse();
struct node
{
    int data;
    struct node *link;
} *front, *rear, *ptr, *ptr1, *header, *new;
void main()
{
    int i, ch;
    clrscr();
    header->data=NULL;
    header->link=NULL;
    front=NULL;
    rear=NULL;
    while(1)
    {
        printf("\n Enter your choice 1.Enqueue 2.Dequeue 3.traverse");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: enqueue();
                    break;
            case 2: dequeue();
                    break;
            case 3: traverse();
                    break;
            default:exit(0);
        }
    }
}
void enqueue()
{
    int item;
    new=malloc(sizeof(struct node));
    printf("\n enter item to enqueue");
    scanf("%d",&item);
    if(front==NULL && rear==NULL)
    {
        header->link=new;
        new->link=NULL;
        front=new;
        rear=new;
        new->data=item;
    }
    else
    {
```

```
        rear->link=new;
        new->data=item;
        new->link=NULL;
        rear=rear->link;
    }
}
void dequeue()
{
    if(front==NULL && rear==NULL)
    {
        printf("\n Queue is empty");
    }
    else
    {
        if(front==rear) /* Q has only one item */
        {
            ptr=front->link;
            header->link=ptr;
            printf("Dequeue item is %d",front->data);
            front=rear=NULL;
        }
        else
        {
            header->link=front->link;
            printf("Dequeue item is %d",front->data);
            free(front);
            front=header->link;
        }
    }
}
void traverse()
{
    ptr=header;
    if(front==NULL && rear==NULL)
    {
        printf("\n Queue is empty");
    }
    else
    {
        printf("\n Items of Queue are ");
        while(ptr->link!=NULL)
        {
            ptr=ptr->link;
            printf("%d\t",ptr->data);
        }
    }
}
```

Output:

Enter your choice 1.Enqueue 2.Dequeue 3.traverse1
enter item to enqueue10

Enter your choice 1.Enqueue 2.Dequeue 3.traverse1
enter item to enqueue20

Enter your choice 1.Enqueue 2.Dequeue 3.traverse1
enter item to enqueue30

Enter your choice 1.Enqueue 2.Dequeue 3.traverse1
enter item to enqueue40

Enter your choice 1.Enqueue 2.Dequeue 3.traverse3
Items of Queue are 10 20 30 40

Enter your choice 1.Enqueue 2.Dequeue 3.traverse2
Dequeue item is 10

Enter your choice 1.Enqueue 2.Dequeue 3.traverse3
Items of Queue are 20 30 40

Enter your choice 1.Enqueue 2.Dequeue 3.traverse1

enter item to enqueue55

Enter your choice 1.Enqueue 2.Dequeue 3.traverse3
Items of Queue are 20 30 40 55

Enter your choice 1.Enqueue 2.Dequeue 3.traverse2
Dequeue item is 20

Enter your choice 1.Enqueue 2.Dequeue 3.traverse3
Items of Queue are 30 40 55

Enter your choice 1.Enqueue 2.Dequeue 3.traverse0

VIVA Questions:

1. What is an abstract data type?
2. What are linear and non linear data Structures?
3. Define queue?
4. What are the various operations that can be performed on queue?
5. What are the applications of queue?
6. Define dequeue?

Lab Assignment:

1. Write a program to implement queue using two stacks
2. Write a program to implement circular queue
3. Write a program to reverse the first K elements of the queue

4. Write a program to implement priority queue using linked list

Conclusion:

Designed program for implementation of queue using linked list

EXERCISE-VI

7.

Aim:

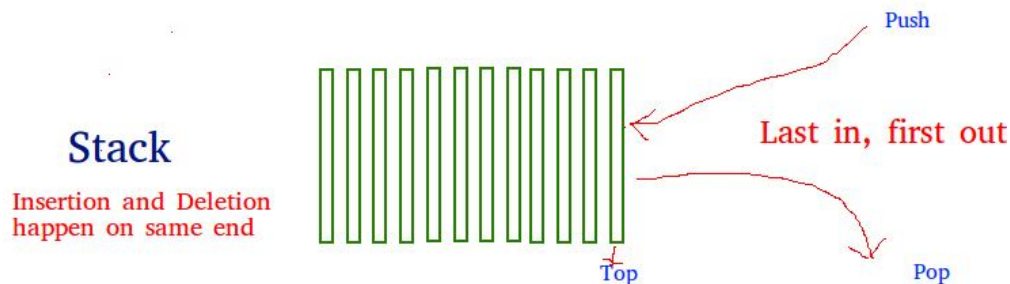
Write a C program that uses Stack operations to convert infix expression into postfix expression.

Description:

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

Mainly the following three basic operations are performed in the stack:

- Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Peek or Top: Returns top element of stack.
- isEmpty: Returns true if stack is empty, else false.



How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Let's convert a little complex expression with parentheses. Below is the given infix expression,

$$((A + B) - C * (D / E)) + F$$

The given expression has parentheses to denote the precedence. So let's start with the conversion with two empty elements respectively,

1. An empty expression string
2. An empty operator stack

The first token to encounter is an open parenthesis, add it to the operator stack.

1. Expression string:
2. Operator Stack: (
3. Remaining expression: (A + B) - C * (D / E)) + F

The second token to encounter is again an open parenthesis, add it to the stack.

1. Expression string:
2. Operator Stack: ((

3. Remaining expression: $A + B) - C * (D / E) + F$

Next token un the expression is an operand “A”, so add it to the expression string.

1. Expression string: **A**
2. Operator Stack: ((
3. Remaining expression: $+ B) - C * (D / E) + F$

Afterward, we have an operator “+”, so add it to the stack.

1. Expression string: **A**
2. Operator Stack: ((+
3. Remaining expression: $B) - C * (D / E) + F$

Then we have an operand, so add it to the expression string.

1. Expression string: **A B**
2. Operator Stack: ((+
3. Remaining expression: $) - C * (D / E) + F$

Next token in the given infix expression is a close parenthesis, as we encountered a close parenthesis we should pop the expressions from the stack and add it to the expression string until an open parenthesis popped from the stack.

1. Expression string: **A B +**
2. Operator Stack: (
3. Remaining expression: $- C * (D / E) + F$

Notice here we didn’t push the close parenthesis to the stack, instead, we popped out the operator “+” and added it to the expression string and popped out one open parenthesis from the stack as well.

Next, we are encountering with an operator “-”, so push it to the stack.

1. Expression string: **A B +**
2. Operator Stack: (-
3. Remaining expression: $C * (D / E) + F$

Next is an operand “C”, so add it to the expression string,

1. Expression string: **A B + C**
2. Operator Stack: (-
3. Remaining expression: $* (D / E) + F$

Next is an operator “*”, so push it to the stack.

1. Expression string: **A B + C**
2. Operator Stack: (- *
3. Remaining expression: $(D / E) + F$

Next is an open parenthesis, so add it to the stack.

1. Expression string: **A B + C**
2. Operator Stack: (- * (
3. Remaining expression: $D / E) + F$

Next is an operand “D”, so add it to the expression string.

1. Expression string: **A B + C D**
2. Operator Stack: (- * (
3. Remaining expression: $/ E) + F$

Next we encounter an operator “/”, so push it to the stack.

1. Expression string: **A B + C D**
2. Operator Stack: (- * (/
3. Remaining expression: $E) + F$

Then an operand “E”, add it to the expression string.

1. Expression string: **A B + C D E**
2. Operator Stack: (- * (/
3. Remaining expression:)) + **F**

Then a close parenthesis, as we saw earlier, we should not push it to the stack instead we should pop all the operators from the stack and add it to the expression string until we encounter an open parenthesis. Then pop the open parenthesis from the stack but don't add it to the expression string.

1. Expression string: **A B + C D E /**
2. Operator Stack: (- *
3. Remaining expression:) + **F**

Next token is again a close parenthesis, so we will pop all the operators and add them to the expression string until we reach the open parenthesis and we will pop the open parenthesis as well from the operator stack.

1. Expression string: **A B + C D E / * -**
2. Operator Stack:
3. Remaining expression: + **F**

Next token is an operator "+", so push it to the stack.

1. Expression string: **A B + C D E / * -**
2. Operator Stack: +
3. Remaining expression: **F**

Next token is an operand, "F". Add it to the expression string.

1. Expression string: **A B + C D E / * - F**
2. Operator Stack: +
3. Remaining expression:

As we processed the whole infix expression, now the operator stack has to be cleared by popping out each remaining operator and adding them to the expression string.

Here we have the operator "+" on the stack, so we will pop out the operator "+" from the stack and will add it to the expression string. So the resultant Postfix expression would look like below,

*Final Postfix expression: A B + C D E / * - F +*

Algorithm:

Algorithm Conversion of infix to postfix

Input: Infix expression.

Output: Postfix expression.

1. Perform the following steps while reading of infix expression is not over
 - a) if symbol is left parenthesis then push symbol into stack.
 - b) if symbol is operand then add symbol to post fix expression.
 - c) if symbol is operator then check stack is empty or not.
 - i) if stack is empty then push the operator into stack.
 - ii) if stack is not empty then check priority of the operators.
 - (I) if priority of current operator >priority of operator present at top of stack then push operator into stack.
 - (II)else if priority of operator present at top of stack >=priority of current operator then pop the operator present at top of stack and add popped operator to postfix expression (go to step I)

- d) if symbol is right parenthesis then pop every element from stack up corresponding left parenthesis and add the popped elements to postfix expression.
2. After completion of reading infix expression, if stack not empty then pop all the items from stack and then add to post fix expression.

End conversion of infix to postfix

Source Code:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
char s[20];
int top=0;
int priority(char);
void main()
{
    char infix[20],postfix[20],ch;
    int i,j,l;
    clrscr();
    printf("\n Enter infix expression: ");
    scanf("%s",infix);
    l=strlen(infix);
    for(i=0,j=0;i<l;i++)
    {
        if(infix[i]=='(')
        {
            top=top+1;
            s[top]=infix[i];
        }
        else if(isalpha(infix[i]) || isdigit(infix[i]))
        {
            postfix[j]=infix[i];
            j=j+1;
        }
        else if(infix[i]=='+' || infix[i]=='-' || infix[i]=='*' || infix[i]=='/' || infix[i]=='%' || infix[i]=='$')
        {
            if(top<1)
            {
                top=top+1;
                s[top]=infix[i];
            }
            else
            {
                if(priority(infix[i]) > priority(s[top]))
                {
                    top=top+1;
                    s[top]=infix[i];
                }
            }
        }
    }
}
```

```

else if(priority(s[top]) >= priority(infix[i]))
{
    while(priority(s[top]) >= priority(infix[i]))
    {
        postfix[j]=s[top];
        top=top-1;
        j=j+1;
    }
    top=top+1;
    s[top]=infix[i];
}
}
else if(infix[i]==')')
{
    while(s[top]!='(')
    {
        postfix[j]=s[top];
        top=top-1;
        j=j+1;
    }
    top=top-1;
}
else
{
    printf("\n Invalid Infix Expression");
}
}
while(top!=0)
{
    postfix[j]=s[top];
    top=top-1;
    j=j+1;
}
postfix[j]='\0';
printf("\nEquivalent infix to postfix expression is %s",postfix);
getch();
}
int priority(char c)
{
    switch(c)
    {
        case '(':      return 0;
        case '+':
        case '-':      return 1;
        case '*':
        case '/':
    }
}

```

```

        case '%':    return 2;
        case '$':    return 3;
    }
    return 0;
}

```

Output 1:

Enter infix expression: (a+b)

Equivalent infix to postfix expression is ab+

Output 2:

Enter infix expression: m+n-o

Equivalent infix to postfix expression is mn+o-

Output 3:

Enter infix expression: (a+b*(c-d))

Equivalent infix to postfix expression is abcd-*+

VIVA Questions:

1. What is an abstract data type?
2. What are linear and non linear data Structures?
3. Define stack?
4. What are the various operations that can be performed on stack?
5. What are the applications of stacks?

Lab Assignment:**Execute the program for the following inputs**

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	++ A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+++ A B C D	A B + C + D +

Conclusion:

Designed program for infix expression to postfix expression using stack

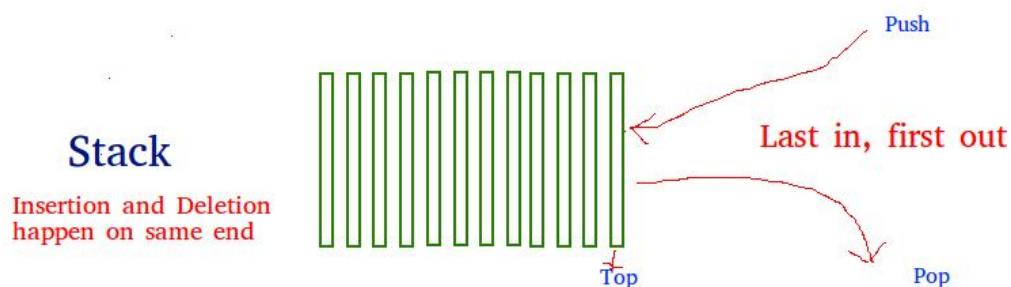
8.

Aim:**Write a C program that uses Stack operations to evaluate postfix expression.****Description:**

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

Mainly the following three basic operations are performed in the stack:

- Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Peek or Top: Returns top element of stack.
- isEmpty: Returns true if stack is empty, else false.



How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...*Operand1 Operand2 Operator*

Example



Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+, -, *, / etc.), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Algorithm:

Algorithm PostfixExpressionEvaluation

Input: Postfix expression

Output: Result of Expression



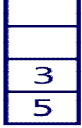
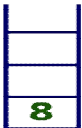

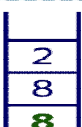

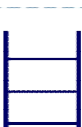

1. Repeat the following steps while reading the postfix expression.
 - a) if the read symbol is operand, then push the symbol into stack.
 - b) if the read symbol is operator then pop the top most two items of the stack and apply the operator on them, and then push back the result to the stack.
2. Finally stack has only one item, after completion of reading the postfix expression. That item is the result of expression.

End PostfixExpressionEvaluation

Infix Expression $(5 + 3) * (8 - 2)$

Postfix Expression $5 3 + 8 2 - *$

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing
+	value1 = pop() value2 = pop() result = value2 + value1 push(result) 	value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push(8) (5 + 3)
8	push(8) 	(5 + 3)
2	push(2) 	(5 + 3)
-	value1 = pop() value2 = pop() result = value2 - value1 push(result) 	value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push(6) (8 - 2) (5 + 3), (8 - 2)
*	value1 = pop() value2 = pop() result = value2 * value1 push(result) 	value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push(48) (6 * 8) (5 + 3) * (8 - 2)
\$ End of Expression	result = pop() 	Display (result) 48 As final result

Infix Expression $(5 + 3) * (8 - 2) = 48$

Postfix Expression $5 3 + 8 2 - *$ value is **48**

Source Code:

```
/* EVALUATION OF POSTFIX EXPRESSION */
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<math.h>
char s[20];
int top=0;
void main()
{
    char postfix[20],symb;
    int i=0,l,op1,op2,res;
    clrscr();
    printf("\n Enter postfix expression");
    scanf("%s",postfix);
    l=strlen(postfix);
    for(i=0;i<l;i++)
    {
        if(isdigit(postfix[i]))
        {
            top=top+1;
            s[top]=postfix[i]-48;
        }
        else
        {
            op2=s[top];
            top=top-1;
            op1=s[top];
            top=top-1;
            switch(postfix[i])
            {
                case '+':
                    res=op1+op2;
                    break;
                case '-':
                    res=op1-op2;
                    break;
                case '*':
                    res=op1*op2;
                    break;
                case '/':
                    res=op1/op2;
                    break;
                case '%':
                    res=op1%op2;
                    break;
            }
        }
    }
}
```

```

                case '$':
                    res=pow(op1,op2);
                    break;
            }
            top=top+1;
            s[top]=res;
        }
    }
    printf("\nResult postfix expression is %d",s[top]);
    getch();
}

```

Output 1:

Enter postfix expression56-
Result postfix expression is -1

Output 2:

Enter postfix expression56*2-
Result postfix expression is 28

Output 3:

Enter postfix expression56+37-.*
Result postfix expression is -44

VIVA Questions:

1. What is an abstract data type?
2. What are linear and non linear data Structures?
3. Define stack?
4. What are the various operations that can be performed on stack?
5. What are the applications of stacks?

Lab Assignment:

Execute the program for the following inputs with A=1, B=2, C=3 and D=4.

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	++ A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+++ A B C D	A B + C + D +

Conclusion:

Designed program for postfix evaluation using stacks

EXERCISE-VII

9.

AIM:

Write a C program to create a Binary Search Tree of integers and perform insert, traversal operations.

DESCRIPTION:

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

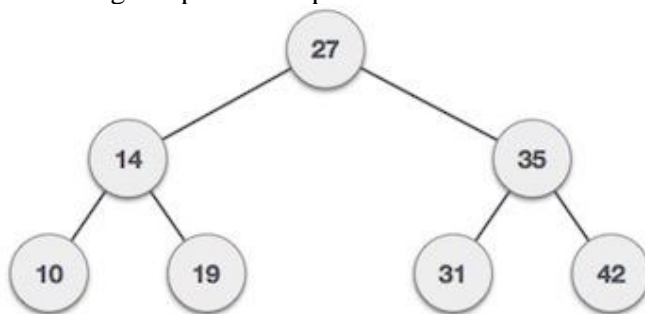
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as

left_subtree (keys) < node (key) ≤ right_subtree (keys)

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree –

- Search – Searches an element in a tree.
- Insert – Inserts an element in a tree.

- Pre-order Traversal – Traverses a tree in a pre-order manner.
- In-order Traversal – Traverses a tree in an in-order manner.
- Post-order Traversal – Traverses a tree in a post-order manner.

Insert Operation:

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Traversal Operation:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

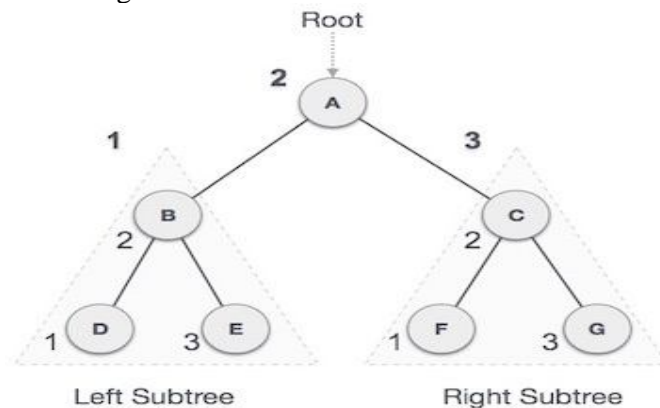
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal:

In this traversal method, the left subtree is visited first, then the root and later the right subtree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

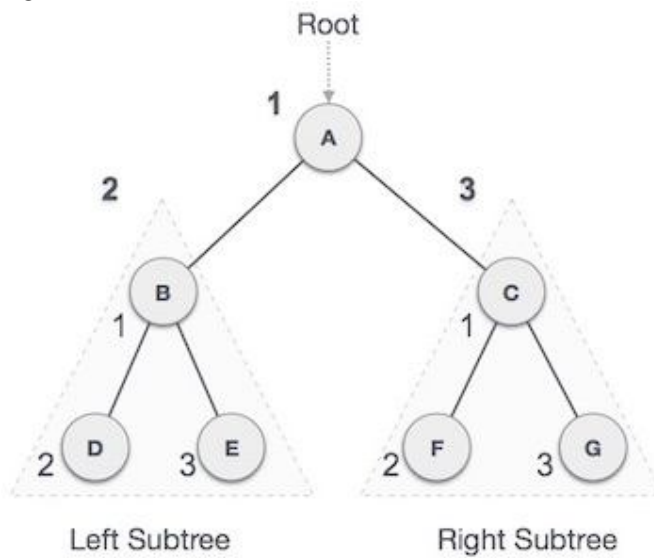


We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Pre-order Traversal:

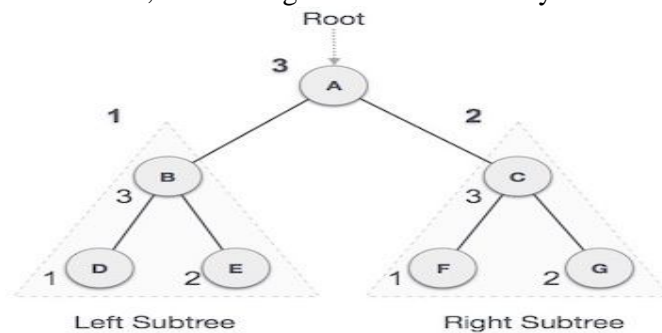
In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$
Post-order Traversal:

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$
Algorithm:**Algorithm BST_Insert(item)**

Input: item is data part of new node to be insert into BST.

Output:BST with new node has data part item.

1. ptr = Root
2. flag = 0
3. while(ptr != NULL and flag == 0)
 - a) if(item == ptr.data)
 - i) flag = 1
 - ii) print(item already exist)
 - b) else if(item < ptr.data)
 - i) ptr1 = ptr
 - ii) ptr = ptr.LCHILD
 - c) else if(item > ptr.data)
 - i) ptr1 = ptr
 - ii) ptr = ptr.RCHILD
 - d) end if
4. end loop
5. if(ptr == NULL)
 - a) new = getnewnode()
 - b) new.data=item
 - c) new.lchild=NULL
 - d) new.rchild=NULL
 - e) if(root.data==NULL)
 - i) root=new;
 - A) print(New node inserted successfully as ROOT Node)
 - f) else if(item<ptr1.data) /* inserting new node as left child to its parent*/
 - i) ptr1.lchild=new
 - ii) print(New Node is inserted successfully as LEFT child)
 - g) else /* inserting new node as right child to its parent*/
 - i) ptr1->rchild=new;
 - ii) print(New Node is inserted successfully as Right Child)
 - h) end if
6. end if

End BST_Insert

Source Code:

```

/* BST INSERTION AND TRAVERSAL */
#include<stdio.h>
#include<malloc.h>
struct node
{
    int data;
    struct node *lchild,*rchild;
} *ptr,*ptr1,*root,*new,*parent,*ptr2,*ptr3,*ptr4;
void insertion();
void preorder (struct node *);
void inorder (struct node *);
void postorder (struct node *);

```

```
void main()
{
    int ch,c,item;
    clrscr();
    root->data=NULL;
    root->lchild=NULL;
    root->rchild=NULL;
    ptr=root;
    while(1)
    {
        printf("\n enter your choice of operation");
        printf("\n 1. insertion \t 2.Traverse");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insertion();
                break;
            case 2:
                printf("\n Enter your traversal");
                printf("\n 1. Preorder 2. Inorder 3. Postorder");
                scanf("%d",&c);
                if(c==1)
                {
                    printf("\n Nodes in BST are");
                    preorder(root);
                }
                else if(c==2)
                {
                    printf("\n Nodes in BST are");
                    inorder(root);
                }
                else if(c==3)
                {
                    printf("\n Nodes in BST are");
                    postorder(root);
                }
                break;
            default : exit(0);
        }
    }
    getch();
}

void insertion()
{
```



```
int item,flag=0;
ptr=root;
printf("\n Enter data part of node to be insert");
scanf("%d",&item);
while(ptr!=NULL && flag==0)
{
    if(item==ptr->data)
    {
        printf("\n Item already exist in BST");
        flag=1;
    }
    else if(item<ptr->data)
    {
        ptr1=ptr;
        ptr=ptr->lchild;
    }
    else if(item>ptr->data)
    {
        ptr1=ptr;
        ptr=ptr->rchild;
    }
}

if(ptr==NULL)
{
    new=malloc(sizeof(struct node));
    new->data=item;
    new->lchild=NULL;
    new->rchild=NULL;
    if(root->data==NULL)
    {
        root=new;
        printf("\n Node %d is inserted successfully as ROOT Node",item);
    }
    else if(item<ptr1->data)    /* inserting new node as left child to its parent*/
    {
        ptr1->lchild=new;
        printf("\n Node %d is inserted successfully LEFT child",item);
    }
    else    /* inserting new node as right child to its parent*/
    {
        ptr1->rchild=new;
        printf("\n Node %d is inserted successfully Right Child",item);
    }
}
}

void preorder(struct node *p)
```

```

{
    if(p!=NULL)
    {
        printf("\t %d",p->data);
        preorder(p->lchild);
        preorder(p->rchild);
    }
}
void inorder(struct node *p)
{
    if(p!=NULL)
    {
        inorder(p->lchild);
        printf("\t %d",p->data);
        inorder(p->rchild);
    }
}

void postorder(struct node *p)
{
    if(p!=NULL)
    {
        postorder(p->lchild);
        postorder(p->rchild);
        printf("\t %d",p->data);
    }
}

```

Output:

enter your choice of operation
1. insertion 2. Traverse1
Enter data part of node to be insert45
Node 45 is inserted successfully as ROOT Node

enter your choice of operation
1. insertion 2. Traverse1
Enter data part of node to be insert15
Node 15 is inserted successfully LEFT child

enter your choice of operation
1. insertion 2. Traverse1
Enter data part of node to be insert48
Node 48 is inserted successfully Right Child

enter your choice of operation
1. insertion 2. Traverse1
Enter data part of node to be insert22
Node 22 is inserted successfully Right Child

enter your choice of operation

1. insertion 2. Traverse1

Enter data part of node to be insert46

Node 46 is inserted successfully LEFT child

enter your choice of operation

1. insertion 2. Traverse 2

Enter your traversal

1. Preorder 2. Inorder 3. Postorder1

Nodes in BST are 45 15 22 48 46

enter your choice of operation

1. insertion 2. Traverse1

Enter data part of node to be insert11

Node 11 is inserted successfully LEFT child

enter your choice of operation

1. insertion 2. Traverse1

Enter data part of node to be insert43

Node 43 is inserted successfully Right Child

enter your choice of operation

1. insertion 2. Traverse1

Enter data part of node to be insert55

Node 55 is inserted successfully Right Child

enter your choice of operation

1. insertion 2. Traverse 2

Enter your traversal

1. Preorder 2. Inorder 3. Postorder3

Nodes in BST are 11 43 22 15 46 55 48 45

enter your choice of operation

1. insertion 2. Traverse 2

Enter your traversal

1. Preorder 2. Inorder 3. Postorder2

Nodes in BST are 11 15 22 43 45 46 48 55

enter your choice of operation

1. insertion 2. Traverse 2

Enter your traversal

1. Preorder 2. Inorder 3. Postorder1

Nodes in BST are 45 15 11 22 43 48 46 55

VIVA Questions:

1. What is a tree?
2. What is the speciality about the inorder traversal of a binary search tree?
3. The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?
4. Which traversals are sufficient to construct BST?

Lab Assignment:

1. We are given a set of n distinct elements and an unlabeled binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a binary search tree?
2. Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree?
3. The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the postorder traversal sequence of the same tree?
4. Write a program to find K^{th} smallest and K^{th} largest elements in BST.

Conclusion:

Designed program for Binary Search Tree of integers and perform insert, traversal operations.

EXERCISE-VIII

10. (i)

AIM:

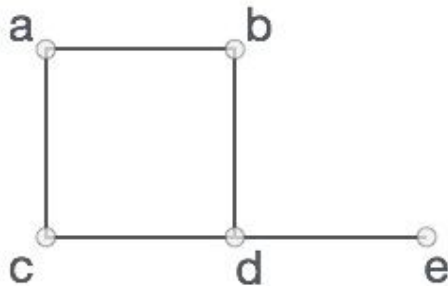
Write a C program to implement Depth First Search for a graph.

DESCRIPTION:

Graph:

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

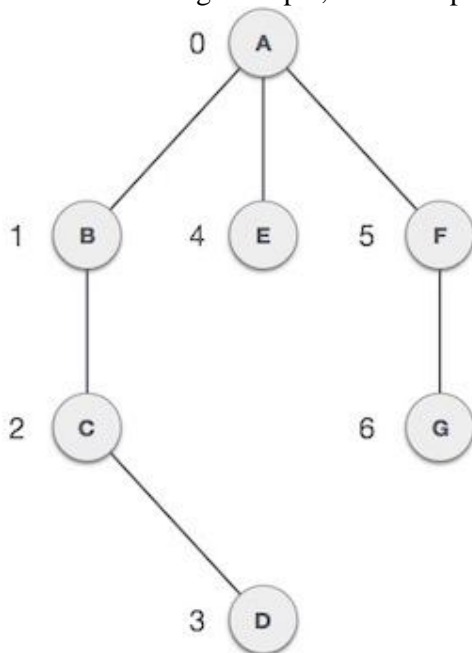
$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labelled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

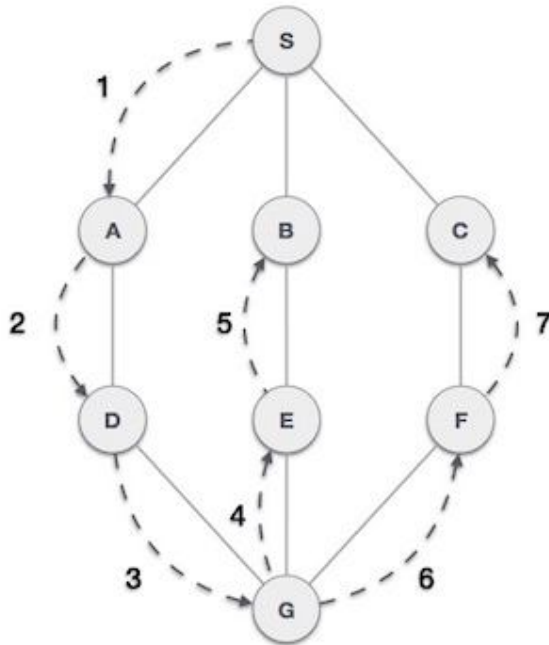
Following are basic primary operations of a Graph –

- Add Vertex – Adds a vertex to the graph.
- Add Edge – Adds an edge between the two vertices of the graph.
- Display Vertex – Displays a vertex of the graph.

The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.

Depth First Search (DFS):

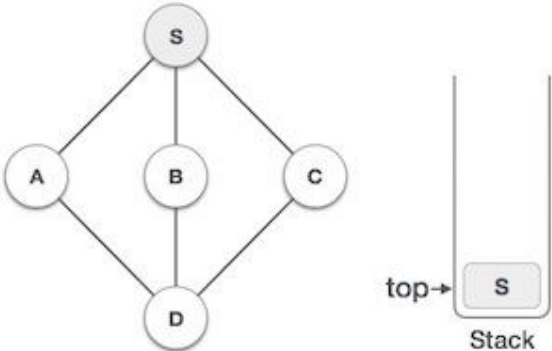
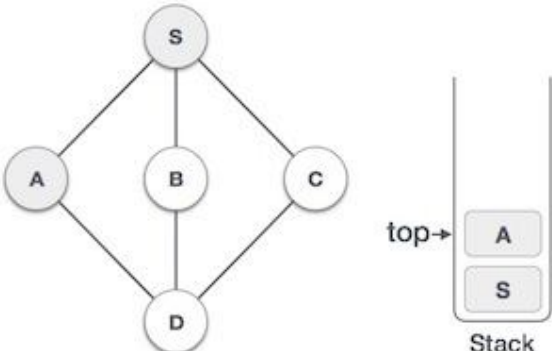
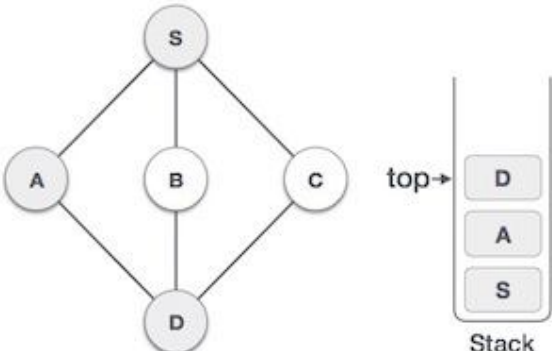
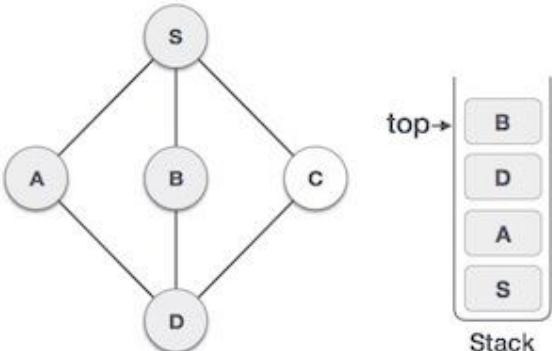
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

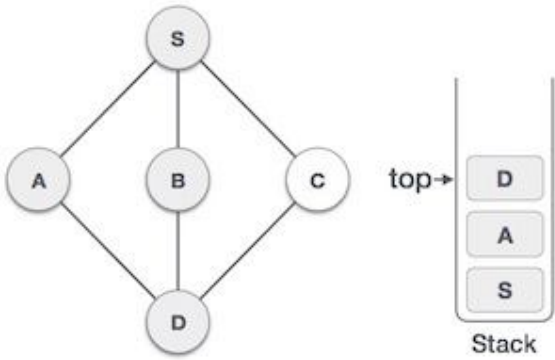
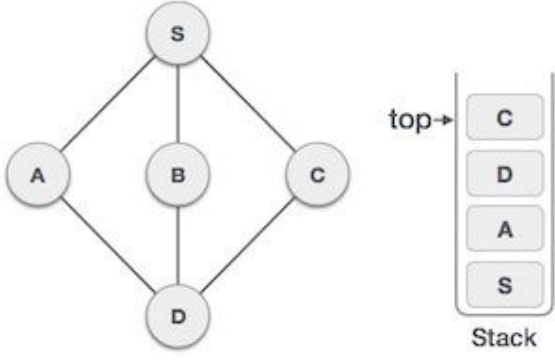


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

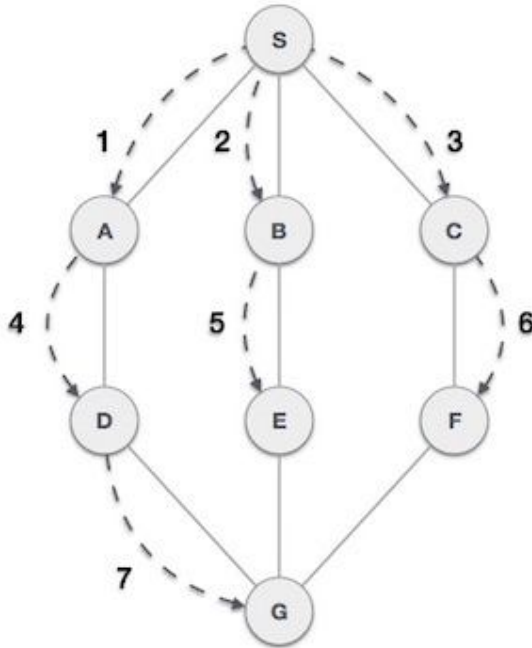
2		<p>Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p>
3		<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>

6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Breadth First Search(BFS):

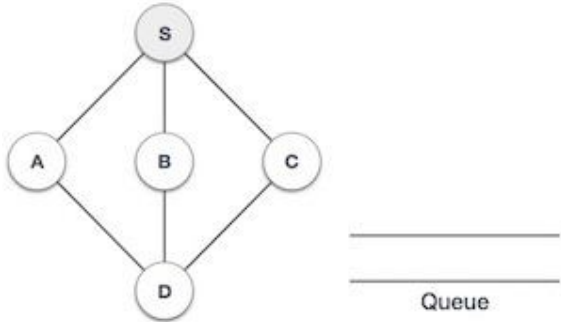
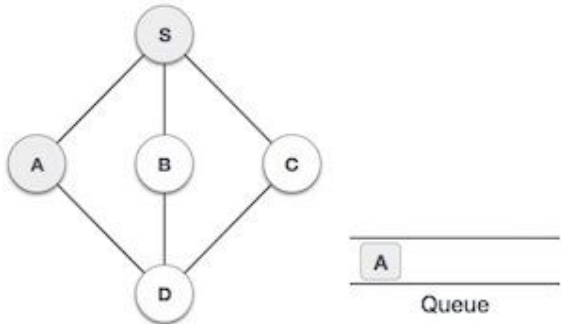
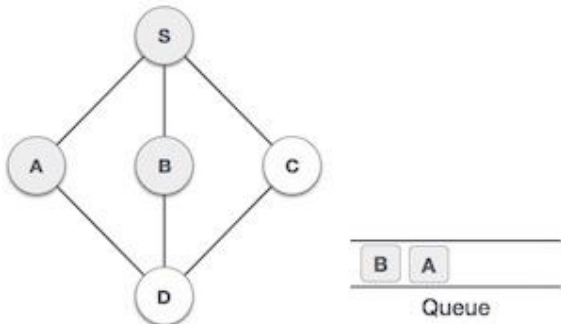
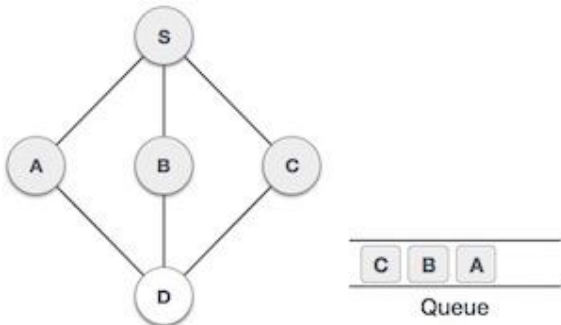
Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

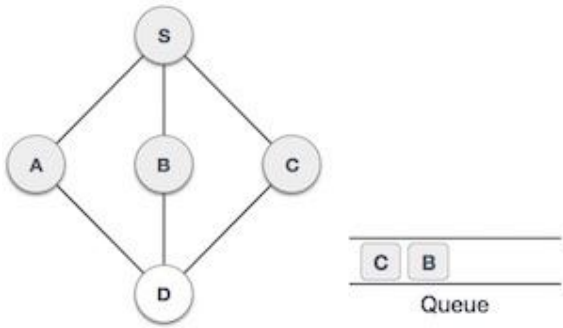
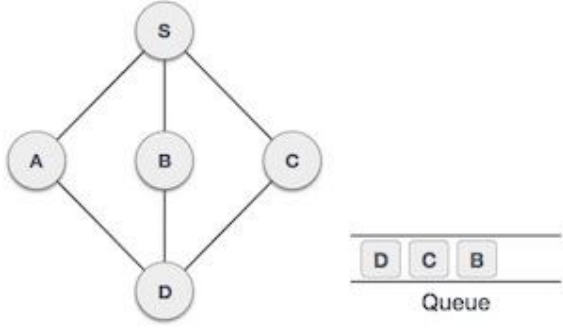


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.

2		<p>We start from visiting S (starting node), and mark it as visited.</p>
3		<p>We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.</p>
4		<p>Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.</p>
5		<p>Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.</p>

6		<p>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</p>
7		<p>From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Difference between BFS and DFS Binary Tree

BFS	DFS
BFS finds the shortest path to the destination.	DFS goes to the bottom of a subtree, then backtracks.
The full form of BFS is Breadth-First Search.	The full form of DFS is Depth First Search.
It uses a queue to keep track of the next location to visit.	It uses a stack to keep track of the next location to visit.
BFS traverses according to tree level.	DFS traverses according to tree depth.
It is implemented using FIFO list.	It is implemented using LIFO list.
It requires more memory as compare to DFS.	It requires less memory as compare to BFS.

This algorithm gives the shallowest path solution.

This algorithm doesn't guarantee the shallowest path solution.

There is no need of backtracking in BFS.

There is a need of backtracking in DFS.

You can never be trapped into finite loops.

You can be trapped into infinite loops.

If you do not find any goal, you may need to expand many nodes before the solution is found.

If you do not find any goal, the leaf node backtracking may occur.

Applications of DFS:

Here are Important applications of DFS:

Weighted Graph:

In a weighted graph, DFS graph traversal generates the shortest path tree and minimum spanning tree.

Detecting a Cycle in a Graph:

A graph has a cycle if we found a back edge during DFS. Therefore, we should run DFS for the graph and verify for back edges.

Path Finding:

We can specialize in the DFS algorithm to search a path between two vertices.

Topological Sorting:

It is primarily used for scheduling jobs from the given dependencies among the group of jobs. In computer science, it is used in instruction scheduling, data serialization, logic synthesis, determining the order of compilation tasks.

Searching Strongly Connected Components of a Graph:

It used in DFS graph when there is a path from each and every vertex in the graph to other remaining vertexes.

Solving Puzzles with Only One Solution:

DFS algorithm can be easily adapted to search all solutions to a maze by including nodes on the existing path in the visited set.

Algorithm:

Algorithm DFS()

Input: Adjacent matrix representation of a graph.

Output: DFS traversal of graph.

1. PUSH the starting vertex into stack
 2. While stack is not empty
 - a) POP a vertex v from stack
 - b) if vertex v is not visited
 - i) visit the vertex v
 - ii) PUSH all the adjacent vertices of v into stack
 - c) end if
 3. end loop
- End DFS

Source Code:

```

/* DFS USING ADJACENT MATRIX using RECURSION */
#include<stdio.h>
#include<conio.h>
int a[20][20],s[20],visited[20],n,i,j,item,st,v,top=0,size=20;
void push(int);
int pop();
void dfs();
void main()
{
    clrscr();
    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        visited[i]=0;
    }
    printf("\n Enter Adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\n Enter the starting vertex:");
    scanf("%d",&st);
    push(st); /* PUSH the starting vertex into the STACK */
    printf("\n DFS for Given graph is");
    dfs();
    getch();
}
void dfs()
{
    if(top!=0) /* While the STACK is not EMPTY */
    {
        v=pop(); /* POP vertex from STACK */
    }
}

```

```

        if(visited[v]==0)                /* If the popped vertex is not visited */
        {
            visited[v]=1;                /* Visit the popped vertex */
            printf("\n %d",v);
            for(i=1;i<=n;i++)
            {
                if(a[v][i]==1)
                {
                    push(i);             /* PUSH ALL THE ADJACENT VERTICES OF vertex V in to STACK */
                }
            }
        }
        dfs();
    }
}
void push(int item)
{
    if(top==size)
    {
        printf("\n Stack is full ");
    }
    else
    {
        top=top+1;
        s[top]=item;
    }
}

int pop()
{
    int item;
    if(top==0)
    {
        printf("\n Stack is empty");
        return 0;
    }
    else
    {
        item=s[top];
        top=top-1;
        return item;
    }
}

/* DFS USING ADJACENT MATRIX NON RECURSION */
#include<stdio.h>

```

```

#include<conio.h>
int top=0,size=20,v;
int a[20][20],s[20],visited[20],n,i,j,item,st;
void push(int);
int pop();
void dfs();
void main()
{
    clrscr();
    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        visited[i]=0;
    }
    printf("\n Enter Adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\n Enter the starting vertex:");
    scanf("%d",&st);
    push(st); /* PUSH the starting vertex into the STACK */
    printf("\n DFS for Given graph is");
    dfs();
    getch();
}

void dfs()
{
    while(top!=0) /* While the STACK is not EMPTY */
    {
        v=pop(); /* POP vertex from STACK */
        if(visited[v]==0) /* If the popped vertex is not visited */
        {
            visited[v]=1; /* Visit the popped vertex */
            printf("\n %d",v);
            for(i=1;i<=n;i++)
            {
                if(a[v][i]==1)
                {
                    push(i); /* PUSH ALL THE ADJACENT VERTICES OF vertex V in to STACK */
                }
            }
        }
    }
}

```

```
        }
    }
}

void push(int item)
{
    if(top==size)
    {
        printf("\n Stack is full ");
    }
    else
    {
        top=top+1;
        s[top]=item;
    }
}

int pop()
{
    int item;
    if(top==0)
    {
        printf("\n Stack is empty");
        return 0;
    }
    else
    {
        item=s[top];
        top=top-1;
        return item;
    }
}
```

Output 1:

Enter the number of vertices:5

Enter Adjacency matrix:

```
0  1  0  0  1
1  0  1  0  0
0  1  0  1  1
0  0  1  0  1
1  0  1  1  0
```

Enter the starting vertex:1

DFS for Given graph is

```
1
5
```


4
3
2

Output 2:

Enter the number of vertices:5

Enter Adjacency matrix:

```
0  1  1  0  0
1  0  1  1  1
1  1  0  1  1
0  1  1  0  1
0  1  1  1  0
```

Enter the starting vertex:1

DFS for Given graph is

1
3
5
4
2

VIVA Questions:

1. List out applications of depth first search?
2. Define a graph?
3. What are different types of traversal in graph?
4. Tell Difference between DFS and BFS
5. Which data structure is used for implementing DFS?
6. **Why** not use a queue for implementing DFS?
7. What is the time complexity of DFS?

Lab Assignment:

1. Write a program for Iterative Depth First Traversal of Graph

Conclusion:

Designed program for depth first search.

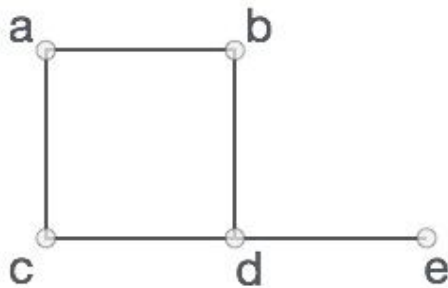
10. (ii)**Aim:**

Write a C program to implement Breadth First Search for a graph.

Description:**Graph:**

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

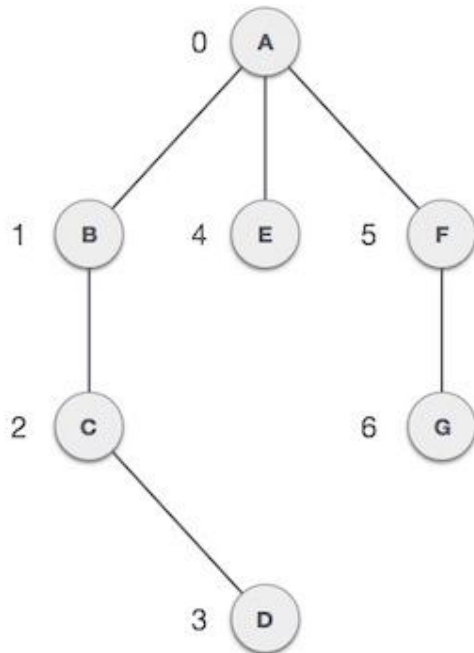
$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

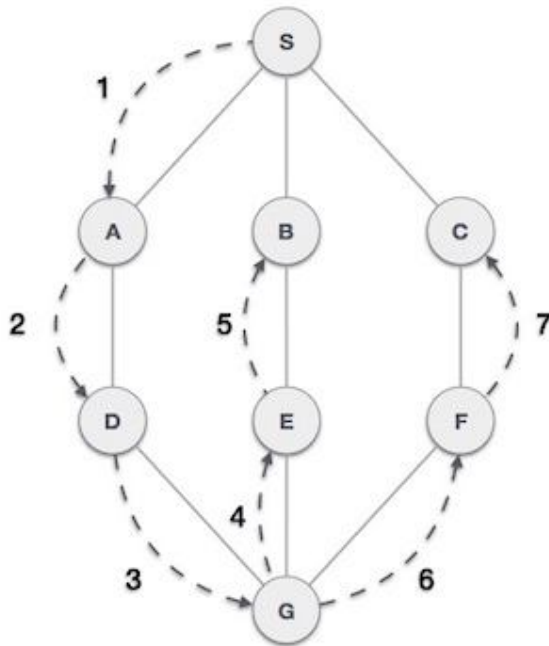
Following are basic primary operations of a Graph –

- Add Vertex – Adds a vertex to the graph.
- Add Edge – Adds an edge between the two vertices of the graph.
- Display Vertex – Displays a vertex of the graph.

The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.

Depth First Search(DFS):

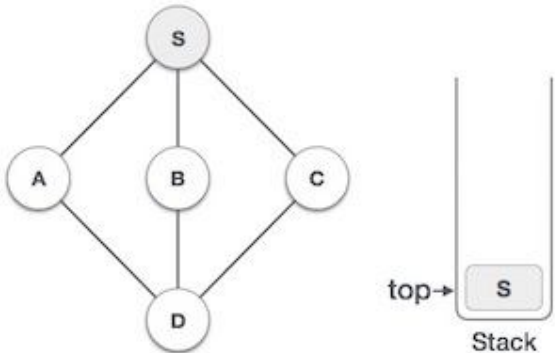
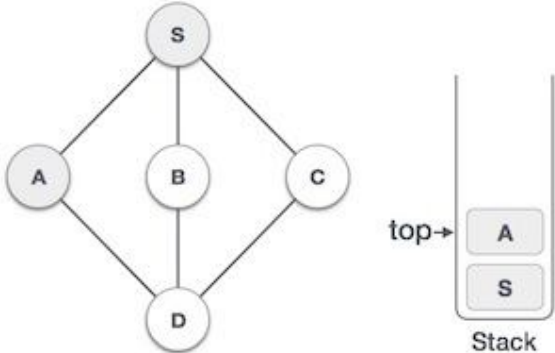
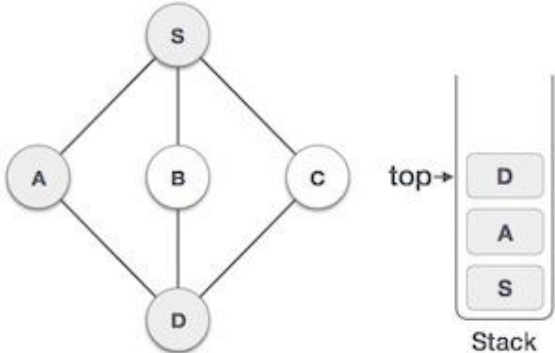
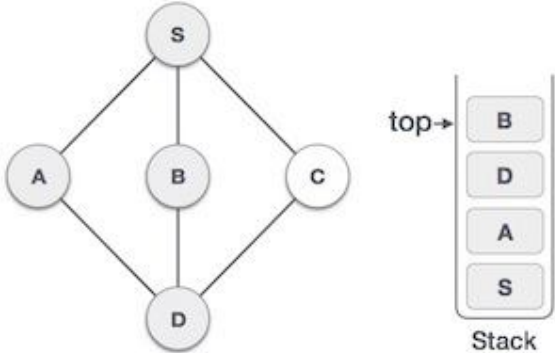
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

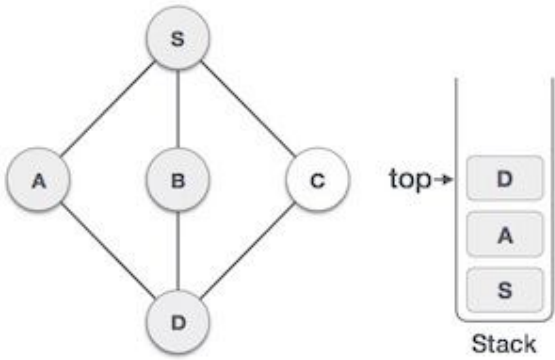
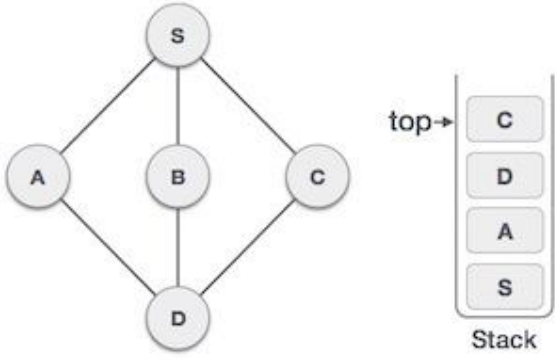


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

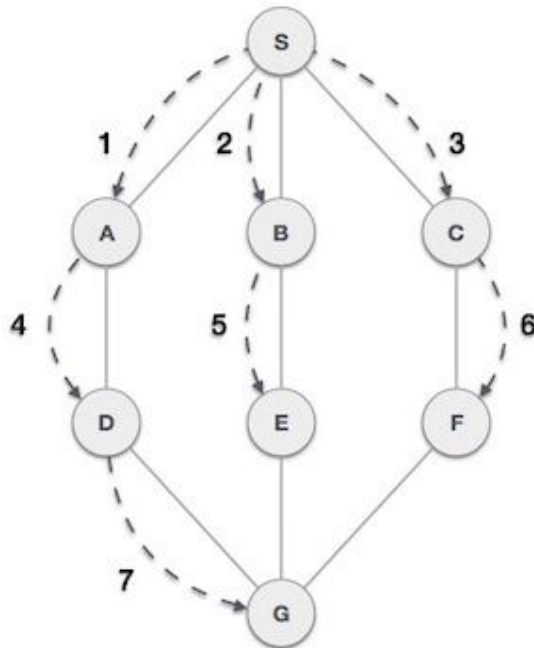
2		<p>Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p>
3		<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5		<p>We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>

6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Breadth First Search(BFS):

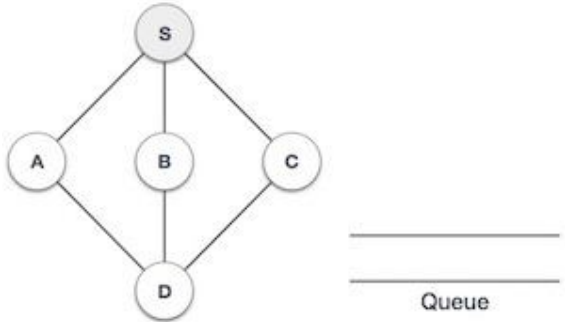
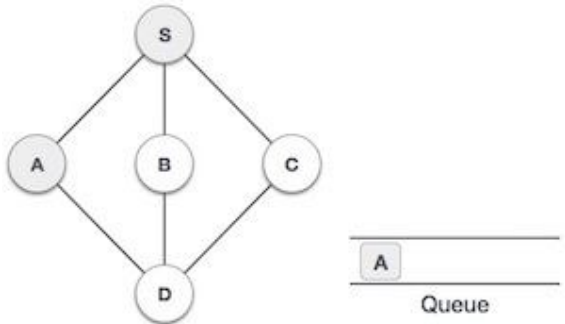
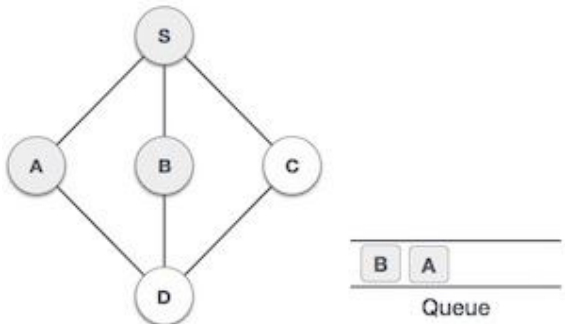
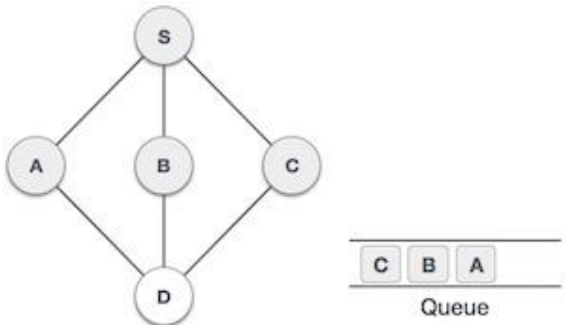
Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

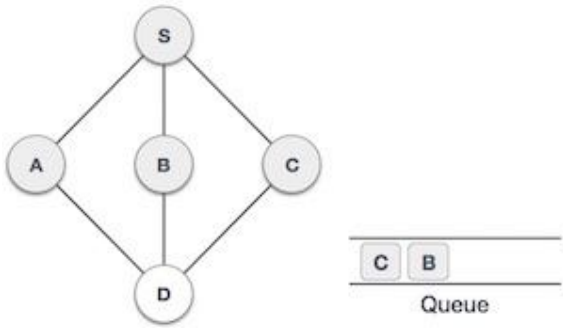
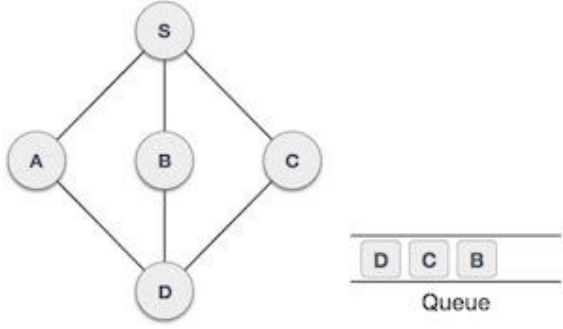


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.

2		<p>We start from visiting S (starting node), and mark it as visited.</p>
3		<p>We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.</p>
4		<p>Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.</p>
5		<p>Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.</p>

6		<p>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</p>
7		<p>From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Difference between BFS and DFS Binary Tree

BFS	DFS
BFS finds the shortest path to the destination.	DFS goes to the bottom of a subtree, then backtracks.
The full form of BFS is Breadth-First Search.	The full form of DFS is Depth First Search.
It uses a queue to keep track of the next location to visit.	It uses a stack to keep track of the next location to visit.
BFS traverses according to tree level.	DFS traverses according to tree depth.
It is implemented using FIFO list.	It is implemented using LIFO list.
It requires more memory as compare to DFS.	It requires less memory as compare to BFS.

This algorithm gives the shallowest path solution.

This algorithm doesn't guarantee the shallowest path solution.

There is no need of backtracking in BFS.

There is a need of backtracking in DFS.

You can never be trapped into finite loops.

You can be trapped into infinite loops.

If you do not find any goal, you may need to expand many nodes before the solution is found.

If you do not find any goal, the leaf node backtracking may occur.

Applications of BFS:

Like DFS, the BFS (Breadth First Search) is also used in different situations. These are like below –

- In peer-to-peer network like bit-torrent, BFS is used to find all neighbor nodes
- Search engine crawlers are used BFS to build index. Starting from source page, it finds all links in it to get new pages
- Using GPS navigation system BFS is used to find neighboring places.
- In networking, when we want to broadcast some packets, we use the BFS algorithm.
- Path finding algorithm is based on BFS or DFS.
- BFS is used in Ford-Fulkerson algorithm to find maximum flow in a network.

Algorithm BFS():

Input: Adjacent matrix representation of a graph.

Output: BFS traversal of graph.

1. ENQUEUE the starting vertex into queue
2. While queue is not empty
 - a) DEQUEUE a vertex v from queue
 - b) if vertex v is not visited
 - i) visit the vertex v
 - ii) ENQUEUE all the adjacent vertices of v into queue
 - c) end if
3. end loop

End BFS

Program:

```
/* BFS using RECURSION*/
#include<stdio.h>
#include<conio.h>
int front=0,rear=0,size=20;
int a[20][20],q[20],visited[20],n,i,j,item;
void bfs();
void enqueue(int);
```

```

int dequeue();
void main()
{
    int s;
    clrscr();
    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        visited[i]=0;
    }
    printf("\n Enter Adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\n Enter the starting vertex:");
    scanf("%d",&s);
    enqueue(s);                /* Enqueue the starting vertex into the QUEUE */
    printf("\n BFS for Given graph is");
    bfs(s);
    getch();
}
void bfs()
{
    int v;
    if(front!=0 && rear !=0)    /* While the QUEUE is not EMPTY */
    {
        v=dequeue();           /* Dequeue vertex from QUEUE */
        if( visited[v]==0)     /* If Dequeued vertex is not visited */
        {
            visited[v]=1;
            printf("\n %d",v);
            for(i=1;i<=n;i++)
            {
                if(a[v][i]==1)
                {
                    enqueue(i); /* Enqueue ALL THE ADJACENT VERTICES OF vertex V */
                }
            }
        }
        bfs();
    }
}

```

```
void enqueue(int item)
{
    if(rear==size)
    {
        printf("\n Queue is full ");
    }
    else
    {
        if(front==0 && rear==0)
        {
            front=1;
            rear=rear+1;
            q[rear]=item;
        }
        else
        {
            rear=rear+1;
            q[rear]=item;
        }
    }
}

int dequeue()
{
    int item;
    if(front==0 && rear==0)
    {
        printf("\n Queue is empty");
        return 0;
    }
    else
    {
        if(front==rear)
        {
            item=q[front];
            front=0;
            rear=0;
            return item;
        }
        else
        {
            item=q[front];
            front=front+1;
            return item;
        }
    }
}
```

```

/* BFS using NON RECURSION*/
#include<stdio.h>
#include<conio.h>
int front=0,rear=0,size=20;
int a[20][20],q[20],visited[20],n,i,j,item;
void bfs();
void enqueue(int);
int dequeue();
void main()
{
    int s;
    clrscr();
    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        visited[i]=0;
    }
    printf("\n Enter Adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("\n Enter the starting vertex:");
    scanf("%d",&s);
    enqueue(s);           /* Enqueue the starting vertex into the QUEUE */
    printf("\n BFS for Given graph is");
    bfs();
    getch();
}
void bfs()
{
    int v;
    while(front!=0 && rear !=0)   /* While the QUEUE is not EMPTY */
    {
        v=dequeue();           /* Dequeue vertex from QUEUE */
        if( visited[v]==0)     /* If Dequeued vertex is not visited */
        {
            visited[v]=1;
            printf("\n %d",v);
            for(i=1;i<=n;i++)
            {

```

```

                if(a[v][i]==1)
                {
                    enqueue(i); /* Enqueue ALL THE ADJACENT VERTICES OF vertex V */
                }
            }
        }
    }
}
void enqueue(int item)
{
    if(rear==size)
    {
        printf("\n Queue is full ");
    }
    else
    {
        if(front==0 && rear==0)
        {
            front=1;
            rear=rear+1;
            q[rear]=item;
        }
        else
        {
            rear=rear+1;
            q[rear]=item;
        }
    }
}
int dequeue()
{
    int item;
    if(front==0 && rear==0)
    {
        printf("\n Queue is empty");
        return 0;
    }
    else
    {
        if(front==rear)
        {
            item=q[front];
            front=0;
            rear=0;
            return item;
        }
        else

```

```

        {
            item=q[front];
            front=front+1;
            return item;
        }
    }
}

```

Output 1:

Enter the number of vertices:5

Enter Adjacency matrix:

```

0  1  0  0  1
1  0  1  0  0
0  1  0  1  1
0  0  1  0  1
1  0  1  1  0

```

Enter the starting vertex:1

BFS for Given graph is

```

1
2
5
3
4

```

Output 2:

Enter the number of vertices:5

Enter Adjacency matrix:

```

0  1  1  0  0
1  0  1  1  1
1  1  0  1  1
0  1  1  0  1
0  1  1  1  0

```

Enter the starting vertex:1

BFS for Given graph is

```

1
2
3
4
5

```

Advantages:

- A BFS will find the **shortest path** between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.

Disadvantages

- A BFS on a binary tree *generally* requires more memory than a DFS.

VIVA Questions:

1. List out applications of breadth first search?
2. Define a graph?

3. What are different types of traversal in graph?
4. Tell Difference between DFS and BFS
5. Which data structure is used for implementing BFS?
6. **Why** not use a queue for implementing BFS?
7. What is the time complexity of BFS?

Lab Assignment:

1. Write a program for Iterative Breadth First Traversal of Graph

Conclusion:

Designed program for breadth first search.

EXERCISE-IX

11.

AIM:

Write a C program to create a hash table and perform insert, display and search operations.

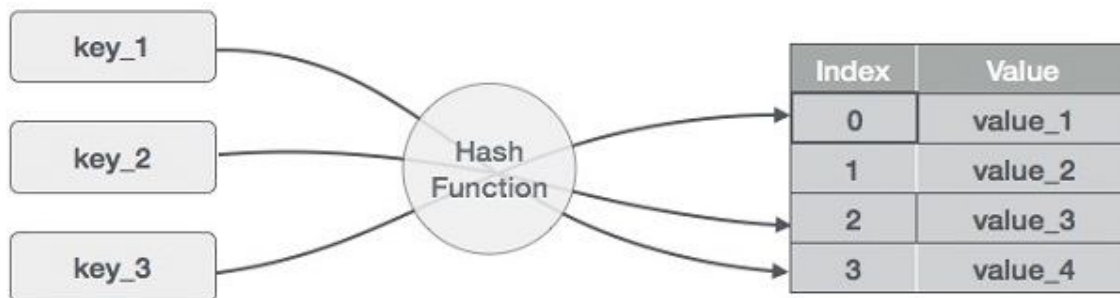
DESCRIPTION:

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

S.No	Key	Hash	Array Index
1	1	$1 \% 20 = 1$	1
2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14

7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** – Searches an element in a hash table.
- **Insert** – inserts an element in a hash table.
- **delete** – Deletes an element from a hash table.

Data Item

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Applications of hashing.

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc.

There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

Message Digest

Password Verification

Data Structures (Programming Languages)

Compiler Operation

Rabin-Karp Algorithm

Linking File name and path together

Message Digest:

This is an application of cryptographic Hash Functions. Cryptographic hash functions are the functions which produce an output from which reaching the input is close to impossible. This property of hash functions is called **irreversibility**.

Lets take an **Example**:

Suppose you have to store your files on any of the cloud services available. You have to be sure that the files that you store are not tampered by any third party. You do it by computing “hash” of that file using a Cryptographic hash algorithm. One of the common cryptographic hash algorithms is **SHA 256**. The hash thus computed has a maximum size of 32 bytes. So a computing the hash of large number of files will not be a problem. You save these hashes on your local machine.

Now, when you download the files, you compute the hash again. Then you match it with the previous hash computed. Therefore, you know whether your files were tampered or not. If anybody tamper with the file, the hash value of the file will definitely change. Tampering the file without changing the hash is nearly impossible.

Password Verification

Cryptographic hash functions are very commonly used in password verification. Let's understand this using an Example:

When you use any online website which requires a user login, you enter your E-mail and password to authenticate that the account you are trying to use belongs to you. When the password is entered, a hash of the password is computed which is then sent to the server for verification of the password. The passwords stored on the server are actually computed hash values of the original passwords. This is done to ensure that when the password is sent from client to server, no sniffing is there.

Data Structures (Programming Languages):

Various programming languages have hash table-based Data Structures. The basic idea is to create a key-value pair where key is supposed to be a unique value, whereas value can be same for different keys. This implementation is seen in unordered set & unordered map in C++, HashSet & HashMap in java, dict in python etc.

Compiler Operation:

The keywords of a programming language are processed differently than other identifiers. To differentiate between the keywords of a programming language (if, else, for, return etc.) and other identifiers and to successfully compile the program, the compiler stores all these keywords in a set which is implemented using a hash table.

Rabin-Karp Algorithm:

One of the most famous applications of hashing is the Rabin-Karp algorithm. This is basically a string-searching algorithm which uses hashing to find any one set of patterns in a string. A practical application of this algorithm is detecting plagiarism. To know more about Rabin-Karp algo go through Searching for Patterns | Set 3 (Rabin-Karp Algorithm).

Linking File name and path together:

When moving through files on our local system, we observe two very crucial components of a file i.e. file_name and file_path. In order to store the correspondence between file_name and file_path the system uses a map (file_name, file_path) which is implemented using a hash table.

Advantages

Main advantage is synchronization.

In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer softwares, particularly for associative arrays, database indexing, caches and sets.

Disadvantages

Hash collisions are practically unavoidable. when hashing a random subset of a large set of possible keys.

Hash tables become quite inefficient when there are many collisions.

Hash table does not allow null values, like hash map.

Algorithm:

Algorithm Hash_Division(key)

Input: Key is new is inserting into hash table.

Output: index for inserting key into hash table.

1. return (key % m)

End Hash_Division

Algorithm Hash_Multiplication(key)

Input: Key is new is inserting into hash table.

Output: index for inserting key into hash table.

1. $A = 0.61804$
2. $t1 = key * A$ /* t1 is integer variable*/
3. $t2 = key * A$ /* t2 is floating point variable */
4. $t2 = t2 - t1$ /* getting fractional part */

```

5. pos = t2 * m          /* Multiplying fractional part with m value */
6. return (pos)

```

End Hash_Multiplication

Algorithm Hash_Universal(key)

Input: Key is new is inserting into hash table.

Output: index for inserting key into hash table.

```

1. return ( ( ( k*a + b ) % p ) % m)

```

End Hash_Universal

Source Code:

/* Operations on Hash tables i.e. Insertion, Display, Search

Using Different Hash methods, i.e. DIVISION METHOD, MULTIPLICATION METHOD, UNIVERSAL Hashing */

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

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

```

int htable[50],h,ch,k,m=11,i,pos,count=0,flag;
void insert(int);
int search(int);
void display();
void main()
{
    clrscr();
    printf("-----HASH FUNCTIONS-----");
    printf("\n[1].Division\n[2].Multiplication\n[3].Universal");
    printf("\nChoose Hash Function:");
    scanf("%d",&h);
    while(1)
    {
        printf("\nEnter choice of OERATIONS ON DICTIONARIES");
        printf("\n[1].INSERT\t[2].SEARCH\t[3].DISPLAY\t[0].EXIT");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                if(count>=m)
                {
                    printf("\nHash Table is Full, SO INSERTION NOT POSSIBLE");
                }
                else
                {
                    printf("Enter Key to be insert:");
                    scanf("%d",&k);
                    insert(k);
                }
                break;
            case 2:
                if(count==0)

```

```

        {
            printf("\nHash Table is Empty");
        }
        else
        {
            printf("Enter Key to be search:");
            scanf("%d",&k);
            pos=search(k);
            if(htable[pos]==k)
            {
                printf("\nKey Found At Location:%d",pos);
            }
            else
            {
                printf("\nKey Not Found in the Hash Table");
            }
        }
        break;
    case 3:
        if(count==0)
        {
            printf("\nHash Table is Empty");
        }
        else
        {
            display();
        }
        break;
    default:
        exit(0);
    }
}
getch();
}

int division(int k)
{
    return (k%m);
}

int multiply(int k)
{
    float A=0.61804,t2,fa;
    int t1;
    t1=(k*A);
    t2=(k*A);
    fa=t2-t1;
    return fa*m;
}

```

```
}
int universal(int k)
{
    int p=13,a=7,b=5;
    pos=((k*a+b)%p)%m;
    return pos;
}

void insert(int k)
{
    pos=search(k);
    if(htable[pos]==k)
    {
        printf("\nKey Already Found in the Hash Table");
    }
    else
    {
        htable[pos]=k;
        printf("\nKey Inserted at Location: %d",pos);
        count++; /* Increment number of keys present in hash table bu 1 */
    }
}

int search(int k)
{
    int tc;
    if(h==1)
    {
        pos=division(k);
    }
    else if(h==2)
    {
        pos=multiply(k);
    }
    else
    {
        pos=universal(k);
    }
    if(ch==1)
    {
        flag=0;
        while(flag==0)
        {
            if(htable[pos]==k || htable[pos]==0)
            {
                flag=1;
                break;
            }
        }
    }
}
```

```

        else
        {
            printf("\n Collision Occured at Location: %d",pos);
            pos=(pos+1)%m;
        }
    }
}
else /* Other than insertion operation */
{
    flag=0;
    tc=0; /* Temporary count whenever u r started for next index to be search*/
    while(flag==0)
    {
        if(htable[pos]==k || tc==m)
        {
            flag=1;
            break;
        }
        else
        {
            pos=(pos+1)%m;
            tc++;
        }
    }
}
return pos;
}

void display()
{
    printf("\nElements in the Hash Table are:\n");
    printf("\nIndex \t Key");
    for(i=0;i<m;i++)
    {
        printf("\n%d\t %d",i,htable[i]);
    }
}

```

Output:

-----HASH FUNCTIONS-----

[1].Division
 [2].Multiplication
 [3].Universal
 Choose Hash Function:1

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2].SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:23

Key Inserted at Location: 1

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:42

Key Inserted at Location: 9

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:31

Collision Occurred at Location: 9

Key Inserted at Location: 10

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:66

Key Inserted at Location: 0

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:44

Collision Occurred at Location: 0

Collision Occurred at Location: 1

Key Inserted at Location: 2

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:23

Key Already Found in the Hash Table

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:34

Collision Occurred at Location: 1

Collision Occurred at Location: 2

Key Inserted at Location: 3

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:45

Collision Occurred at Location: 1

Collision Occurred at Location: 2

Collision Occurred at Location: 3

Key Inserted at Location: 4

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

1

Enter Key to be insert:77

Collision Occurred at Location: 0

Collision Occurred at Location: 1

Collision Occurred at Location: 2

Collision Occurred at Location: 3

Collision Occurred at Location: 4

Key Inserted at Location: 5

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

3

Elements in the Hash Table are:

Index	Key
0	66
1	23
2	44
3	34
4	45
5	77
6	0
7	0
8	0
9	42
10	31

Enter choice of OERATIONS ON DICTIONARIES[1].INSERT

[2]. SEARCH

[3].DISPLAY [0].EXIT

2

Enter Key to be search:34

Key Found At Location:3

Enter choice of OERATIONS ON DICTIONARIES

[1].INSERT [2]. SEARCH [3].DISPLAY [0].EXIT

2

Enter Key to be search:99

Key Not Found in the Hash Table
Enter choice of OPERATIONS ON DICTIONARIES

VIVA Questions:

1. What is hashing?
2. Define linear probing?
3. What is direct addressing?
4. What is the search complexity in direct addressing?
5. What is a hash function?
6. What is simple uniform hashing?

Conclusion:

Designed program to create a hash table and perform insert, display and search operations.