Lab Manual: Data Structure and algorithms

[DSAL – 2022-26]

Dr. Kailash Shaw, Prof. Hema karande

TE7546

Vision of Institute

To be a center of excellence for creation and dissemination of knowledge by imparting life skills and experiential learning for a promising future in the areas of engineering and technology.

Mission of Institute

- To promote professional ethics and experiential learning for better employability.
- To contribute towards knowledge generation and dissemination in the field of engineering.
- To address societal problems by promoting research, innovation and entrepreneurship.
- To develop global competencies amongst students by fostering value-based education.
- To strengthen industrial, Institutional, and international collaborations for synergetic relations.

Vision of Department

To impart quality education with research insights for developing competent global engineers in the field of Artificial Intelligence and Machine Learning to solve societal problems.

Mission of Department

- To educate students on cutting-edge AIML technologies with strong industry connections to develop problem-solving capabilities, leadership, and teamwork skills.
- To produce quality research through national and international collaborations leading to publications, IPR, and sponsored/funded projects.
- To inculcate professional values with lifelong learning through curricular and co-curricular activities and create globally-aware citizens.

PROGRAM OUTCOMES (PO):

Engineering Graduates 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
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 (PSOs):

1. To apply the concepts of Artificial Intelligence and Machine Learning with practical knowledge in analysis, design and development of intelligent systems and applications to multi-disciplinary problems
2. To provide a concrete foundation to the students in the cutting edge areas Artificial Intelligence and Machine Learning and excelling in the specialized areas like Natural Language Processing, Computer Vision, Reinforcement Learning, Internet of Things, Cloud computing, Data Security and privacy etc.

GENERAL LABORATORY INSTRUCTIONS

1) **Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5 minutes will not be allowed into the lab.**
2) **Plan your task properly much before to the commencement, come prepared to the lab with the program / experiment details.**
3) **Student should enter into the laboratory with:**
    a. **Laboratory Record updated up to the last session experiments.**
    b. **Proper Dress code and Identity card.**
4) **Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.**
5) **All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.**
6) **Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.**
7) **Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviours with the staff and systems etc., will attract severe punishment.**
8) **Students must take the permission of the faculty in case of any urgency to go out; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.**
9) **Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.**

Lab Duration: 60 hours

Lab Objectives

1) Understand the basic concepts of data structures and algorithms.
2) Gain hands-on experience with various data structures and their operations.
3) Develop programming skills for implementing data structures and algorithms.
4) Analyze the time and space complexity of different data structures and algorithms.
5) Solve real-world problems using appropriate data structures and algorithms.

Lab Outcome

CO 1: Gain understanding and practical implementation of searching (linear, binary) and sorting (bubble, selection, insertion, merge, quick) techniques. Analyze complexity, evaluate efficiency, and recognize importance of data structures and algorithms.

CO 2: Gain understanding and practical implementation of singly linked lists, doubly linked lists, and circularly linked lists. Implement menu-driven programs for create, insert, delete, reverse, and concatenate operations. Analyze efficiency and apply knowledge to solve real-world problems.

CO 3: students will gain the ability to design and implement menu-driven programs to create a binary search tree, conduct inorder, preorder, and postorder traversals, and perform efficient node searches within the tree structure.

Co 4: Gain practical implementation skills in graph algorithms, including insertion and deletion using adjacency list, Dijkstra's shortest path algorithm, BFS, and DFS. Apply knowledge to solve real-world problems and analyze algorithm efficiency.

Lab Equipment and Software:

1) Computers with the required programming environment (e.g., C/C++ compiler).
2) Integrated Development Environment (IDE) such as Visual Studio Code, Eclipse, or Code, Blocks, DEV C++. (Recommended DEV C++)
3) Sample input/output files for testing purposes.

Lab Activities:

Experiment 1: Implement following searching techniques:

i. Linear

ii. Binary

Experiment 2: Implement following sorting techniques:

i. Bubble

ii. Selection

iii. Insertion

Experiment 3: Implement following sorting techniques:

i. Merge

ii. Quick

Experiment 4: Write a menu driven program that implements singly linked list for the following operations:

Create, insert, delete, reverse, concatenate.

Experiment 5: Write a menu driven program that implements doubly linked list for the following operations:

Create, insert, delete, reverse, concatenate.

Experiment 6: Write a menu driven program that implements Circular linked list for the following operations:

Create, insert, delete, reverse, concatenate.

Experiment 7: Write a menu driven program to

i. Create a binary search tree

ii. Traverse the tree in inorder, preorder and postorder

iii. Search the tree for a given node

Experiment 8: Write a program to insert and delete nodes in graph using adjacency list.

Experiment 9: Write a program to implement Dijkstra "s shortest path algorithm for a given directed Graph.

Experiment 10: Write a program in C to implement Breadth First Search using linked representation of graph.

Experiment 11: Write a program in C to implement Depth First Search using linked representation of graph.

Lab Report:

- **Each student should submit a lab report documenting their implementation, including code snippets and output screenshots.**
- **The report should include a description of the implemented data structures, algorithms, and their time complexity analysis.**
- **Students should provide a detailed explanation of the testing performed and the results obtained.**
- **The report should also include any challenges faced during the lab and their solutions.**

Lab Evaluation:

- **Students will be evaluated based on their implementation of the required data structures and algorithms.**
- **The correctness and efficiency of the implemented solutions will be assessed.**
- **The quality and completeness of the lab report will also be considered for evaluation.**

Assessment	Marks	Total
Continuous Assessment (CA)	20	50
End Sem Exam (ESE)	30	

Continuous Assessment (CA):

Each Lab will be evaluated based on following Rubrics. Finally, all the marks should be scale down to maximum 20 only.

Lab Performance for each experiment				Viva	Total
Timely Lab report Submission	Correctness of Code	Code Quality	Testing and Validation	Based on the Experiment Performed	
2	6	4	2	6	20

Timely Lab Report Submission (10%): If a student fails to submit the handwritten report by the specified due date, a penalty will be imposed. The marks for report submission will decrease by 1 marks if submitted within next five days, and after 5 days you will get 0 marks. However, it is still mandatory for the student to submit the report, as it is crucial for claiming other marks allocated to the laboratory. Failure to submit the report will result in a total score of 0 for the entire lab. In addition to the handwritten notes, students are required to submit a single PDF copy containing the executable code, description, and output screen to the designated Google Classroom shortly after the experiment concludes. **Note: It is compulsory for you to add your name and PRN in the top left corner of each page of the report. Additionally, please be aware that in case of plagiarized code, no marks will be awarded.**

Correctness of Code (30%): Marks will be awarded based on

- Implements required algorithm correctly.
- Handles edge cases (e.g., empty array, target not found) appropriately.
- Returns the correct value or output as required

Code Quality (30%):

- Follows proper coding conventions (e.g., indentation, variable naming).
- Uses appropriate data types and memory management.
- Includes comments for better code understanding.
- Avoids code redundancy and follows good programming practices.

Testing and Validation (10%):

- Tests the program with various test cases, including edge cases.
- Produces correct output for all test cases.

Viva (30%): Based on the Experiment performed.

End Sem Exam (ESE): To be evaluated at the end of semester

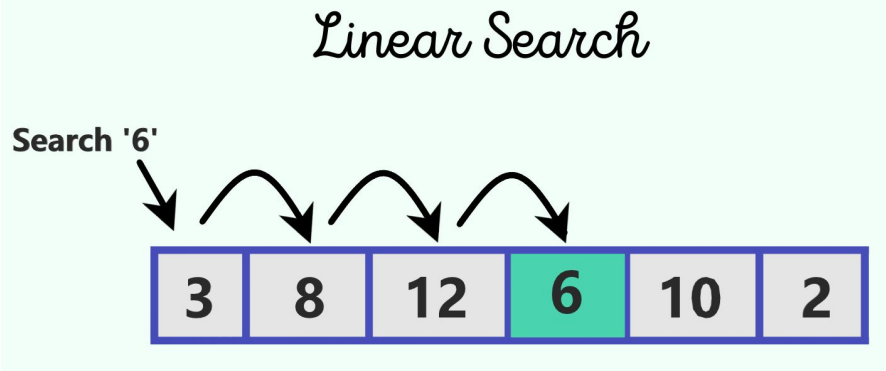
Lab Performance				Viva	Total
Timely Completion of Code with write Up	**Correctness of Code**	**Code Quality**	**Testing and Validation**	**Based on the Experiment Performed**	
2	**10**	**4**	**4**	**10**	**30**

Experiment 1

Linear Search

In computer science, a linear search or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.

A linear search runs in at worst linear time and makes at most n comparisons, where n is the length of the list. If each element is equally likely to be searched, then linear search has an average case of n+1/2 comparisons, but the average case can be affected if the search probabilities for each element vary. Linear search is rarely practical because other search algorithms and schemes, such as the binary search algorithm and hash tables, allow significantly faster searching for all but short lists.



For a list with *n* items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case *n* comparisons are needed.

If the value being sought occurs *k* times in the list, and all orderings of the list are equally likely, the expected number of comparisons is

$$\begin{cases} n & \text{if } k = 0 \\ \frac{n+1}{k+1} & \text{if } 1 \leq k \leq n. \end{cases}$$

For example, if the value being sought occurs once in the list, and all orderings of the list are equally likely, the expected number of comparisons is n+1/2. However, if it is *known* that it occurs once, then at most *n* - 1 comparisons are needed, and the expected number of comparisons is

$$\frac{(n+2)(n-1)}{2n}$$

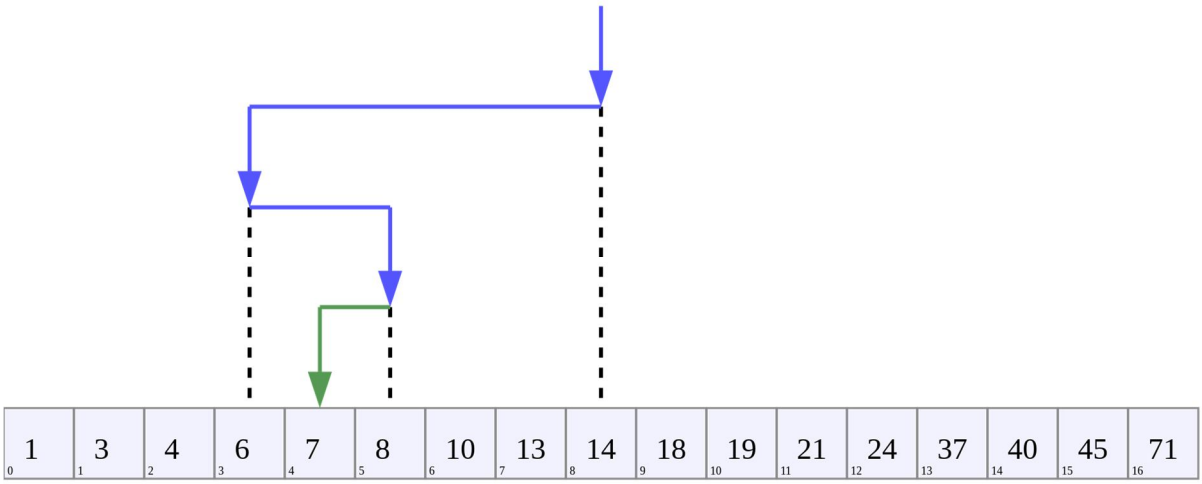
(for example, for n = 2 this is 1, corresponding to a single if-then-else construct).Either way, asymptotically the worst-case cost and the expected cost of linear search are both O(*n*).

Binary Search

In computer science, binary search, also known as half-interval search,[1] logarithmic search,[2] or binary chop,[3] is a search algorithm that finds the position of a target value within a sorted array.[4][5] Binary search compares the target value to the middle element of the array. If

they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Binary search runs in logarithmic time in the worst case, making O(log n) comparisons, where *n* is the number of elements in the array. Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search.



A. Problem: Find the smallest positive integer missing from an unsorted array of integers.

Description: You are given an array of integers, which may contain both positive and negative numbers. However, the array does not contain any duplicate elements. Your task is to write a C program to find the smallest positive integer that is missing from the array.

Write a function int findMissingInteger(int arr[], int size) that takes the following parameters:

- **arr: An array of integers.**
- **size: The size of the array.**

The function should return the smallest positive integer that is missing from the array. If no positive integer is missing, the function should return the next positive integer after the maximum element in the array.

Example:

Input array: {-2, 1, 3, 4, 6, 2}

Output: The smallest positive missing integer is: 5

Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Input and Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

B. Problem: **Write a C program to count the occurrences of each word in a given text.**

Description: Your program should take a text string as input and count the occurrences of each word in the text. Words are separated by spaces or punctuation marks (such as commas or periods). The program should then display the results, listing each unique word along with its corresponding count.

Write a function void countWordOccurrences(char text[]) that takes the following parameter:

- **text: A null-terminated string representing the input text.**

Example:

Input char text [] = "This is a sample text. It contains sample words, some of which are repeated.";

countWordOccurrences(text);

Output:

This: 1
is: 1
a: 1
sample: 2
text: 1
It: 1
contains: 1
words: 1
some: 1
of: 1
which: 1
are: 1
repeated: 1

Objective of Experiment:

Define Linear search:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

C. Problem: **Write a C program to find the smallest missing positive integer from a sorted array of integers. The array may contain duplicates and negative numbers, but it is guaranteed to be sorted in ascending order. Your program should return the smallest missing positive integer not present in the array.**

Write a function int findSmallestMissingPositive(int arr[], int size) that takes the following parameters:

arr: A sorted array of integers.
size: The number of elements in the array.
The function should return the smallest missing positive integer.

Input: **{-4, -2, 0, 1, 2, 4, 5};**
Output:
Smallest missing positive integer: 3

Objective of Experiment:

Define Binary search:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

D. Problem: **Write a C program to find the index of a target word in a sorted array of words. The array is sorted in lexicographic (dictionary) order. Your program should perform a binary search to find the index of the target word in the array, if it exists. If the target word is not present in the array, your program should return -1.**

Write a function int binarySearchWords(char* words[], int size, char* target) that takes the following parameters:

words: An array of strings representing the sorted words.
size: The number of elements in the array.
target: The target word to search for.
The function should return the index of the target word in the array, or -1 if it is not found.

Input:

words[] = {"apple", "banana", "cherry", "grape", "orange", "peach"};
char* target = "grape";

Output: **Target word 'grape' found at index 3**

Objective of Experiment:

Define Binary search:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

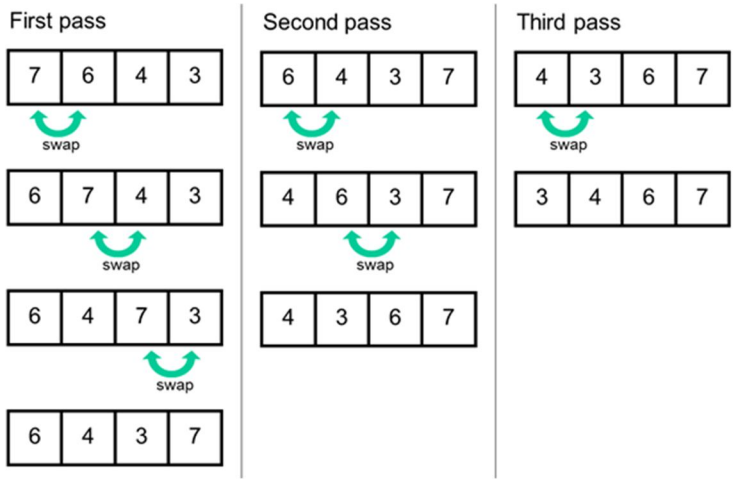
Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 2

Bubble Sort:

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed. These passes through the list are repeated until no swaps had to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a comparison sort, is named for the way the larger elements "bubble" up to the top of the list.

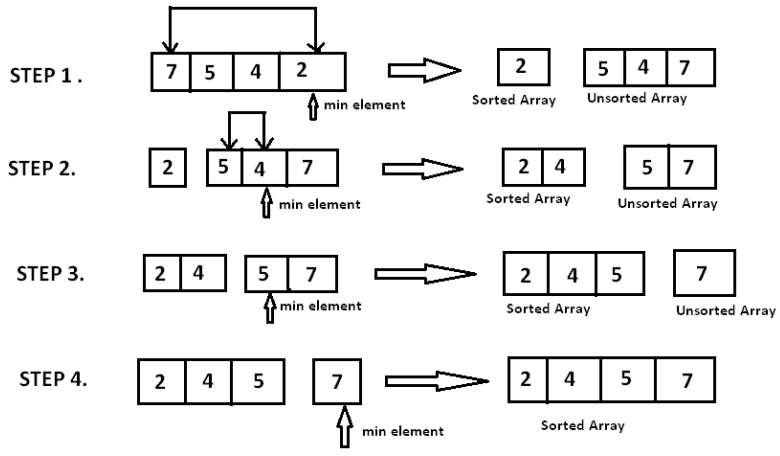
Bubble sort has a worst-case and average complexity of O(n^2), where *n* is the number of items being sorted.



Selection Sort:

In computer science, selection sort is an in-place comparison sorting algorithm. It has an O(n2) time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

Selection sort is not difficult to analyze compared to other sorting algorithms, since none of the loops depend on the data in the array. Selecting the minimum requires scanning *n* elements (taking *n-1* comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining *n-2* elements and so on. Therefore, the time complexity is O(n^2)

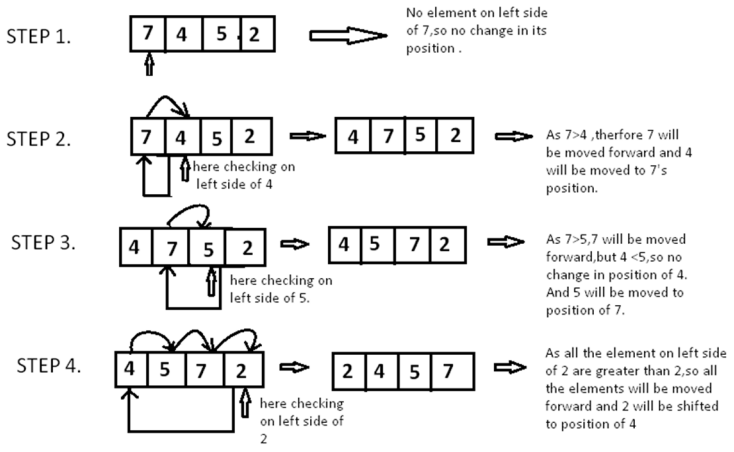


Insertion Sort:

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons. Insertion sort iterates, consuming one input element each repetition, and grows a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n^2)).



A. Problem: **Write a C program that implements the Bubble Sort algorithm to sort an array of integers in ascending order.**

Requirements:

- **The program should prompt the user to enter the size of the array (maximum size: 100) and the elements of the array.**
- **The program should implement the Bubble Sort algorithm to sort the array in ascending order.**
- **After sorting, the program should print the sorted array.**

Example:

Enter the size of the array: 6

Enter the elements of the array:

Element 1: 9

Element 2: 2

Element 3: 7

Element 4: 4

Element 5: 1

Element 6: 5

Output

Array before sorting: 9 2 7 4 1 5

Array after sorting: 1 2 4 5 7 9

Objective of Experiment:

Define Bubble Sort:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

B. Problem: **Write a C program that implements the Bubble Sort algorithm to sort an array of integers in descending order. However, there is a catch. The program should terminate the sorting process early if it detects that the array is already sorted in descending order. In other words, if the array is already in descending order, the program should stop the sorting process and print the sorted array.**

Requirements:

- **The program should prompt the user to enter the size of the array (maximum size: 100) and the elements of the array.**
- **The program should implement the Bubble Sort algorithm to sort the array in descending order.**
- **If the program detects that the array is already sorted in descending order during the sorting process, it should terminate the sorting process and print the sorted array.**
- **If the array is not already sorted, the program should continue sorting until the entire array is sorted in descending order.**
- **After sorting, the program should print the sorted array.**

Example:

Input
Enter the size of the array: 5
Enter the elements of the array:
Element 1: 9
Element 2: 7
Element 3: 5
Element 4: 3
Element 5: 1
Output
Array before sorting: 9 7 5 3 1
Array after sorting: 9 7 5 3 1
Explanation:

In this example, the array is already sorted in descending order. As a result, the program terminates the sorting process early and prints the sorted array without performing any unnecessary comparisons or swaps.

Note: This problem requires you to implement the Bubble Sort algorithm and add an additional check to terminate the sorting process early if the array is already sorted.

Objective of Experiment:

Define Optimized bubble Sort:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

C. Problem: Do a Time Complexity Comparison on the Code (B), when implemented using Bubble, Selection and Insertion Sort, on random data (10000).

Requirements:

- **Generates an array of Random Integer and save it in arr[].**
- **Copy the content of arr[] into arr1[], arr2[], arr3[].**
- **performs Bubble Sort, Selection Sort, and Insertion Sort on the array, and measures the execution time of each algorithm. The time taken for each sorting**
- **Use #include<time.h> header file, and its library clock() to capture the current time. Use following syntax**
  clock_t start = clock();
  / / write your code or call process
  clock_t end = clock();
  //calculate the time difference as
  double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

Objective of Experiment:

Define Bubble, Selection and Insertion Sort:

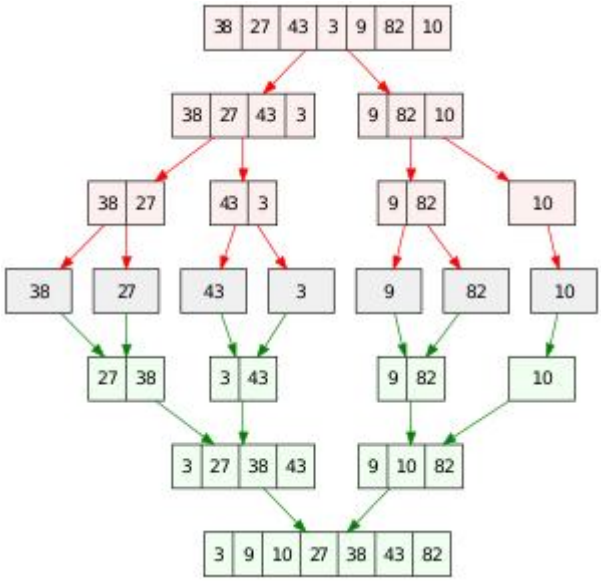
Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 3

Merge Sort:



In computer science, merge sort (also commonly spelled as mergesort) is an efficient, general-purpose, and comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output. Merge sort is a divide-and-conquer algorithm that was invented by John von Neumann in 1945. merge sort's worst case complexity is O(n log n)

Conceptually, a merge sort works as follows:

1) Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
2) Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

Quick Sort:

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons.

A. Problem: **Write a C program to sort an array of integers using the Merge Sort algorithm. However, the program should implement an optimized version of Merge Sort that skips the merge step if the two subarrays are already sorted in ascending order. The program should also keep track of the number of comparisons performed during the sorting process.**

Requirements:

- **The program should prompt the user to enter the size of the array (maximum size: 100) and the elements of the array.**
- **The program should implement the Merge Sort algorithm to sort the array in ascending order.**
- **If the program detects that the two subarrays in the merge step are already sorted in ascending order, it should skip the merge step for that subarray.**
- **The program should keep track of the number of comparisons performed during the sorting process and print the total number of comparisons at the end.**
- **After sorting, the program should print the sorted array.**

Example:

Enter the size of the array: 7
Enter the elements of the array:
Element 1: 5
Element 2: 2
Element 3: 7
Element 4: 1
Element 5: 4
Element 6: 3
Element 7: 6
Output:

Array before sorting: 5 2 7 1 4 3 6
Array after sorting: 1 2 3 4 5 6 7
Total comparisons: 10

Objective of Experiment:

Define Merge Sort:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

B. Problem: Do a Time Complexity Comparison on the Code (A), when implemented using Quick Sort with Merge Sort on random data (10000).

Requirements:

- **Generates an array of Random Integer and save it in arr[].**
- **Copy the content of arr[] into arr1[], arr2[], arr3[].**
- **performs Bubble Sort, Selection Sort, and Insertion Sort on the array, and measures the execution time of each algorithm. The time taken for each sorting**
- **Use #include<time.h> header file, and its library clock() to capture the current time. Use following syntax**
  clock_t start = clock();
  / / write your code or call process
  clock_t end = clock();
  / /calculate the time difference as
  double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

Objective of Experiment:

Define Qsort:

Flow Chart/Pseudo Code/Algorithm:

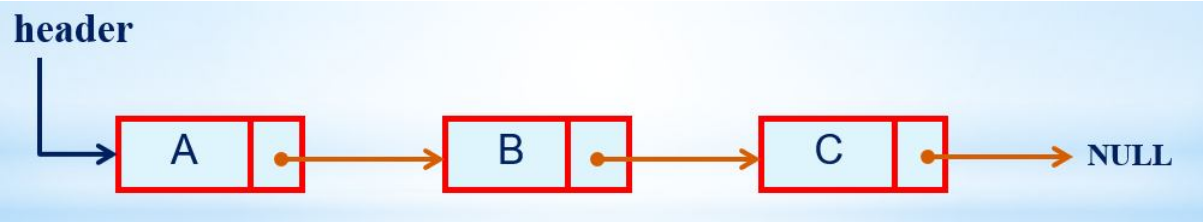
Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 4

Linked List

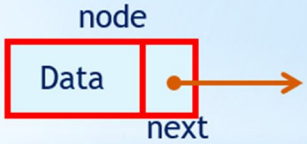
- A linked list is a data structure which allows to store data dynamically and manage data efficiently.
- Typically, a linked list, in its simplest form looks like the following



- **Few salient features**
  - There is a pointer (called header) points the first element (also called node)
  - Successive nodes are connected by pointers.
  - Last element points to NULL.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space, consume exactly what it needs.
- **Defining a node in Linked List**

  Each structure of the list is called a node, and consists of two fields:
  - Item (or) data
  - Address of the next item in the list (or) pointer to the next node in the list
  - **How to define a node in Linked List**

```
struct node
{
   int data;            /* Data */
   struct node *next;   /* pointer*/
} ;
```



Single Linked List

Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

Single linked list (or simply linked list)

- A head pointer addresses the first element of the list.

- Each element points at a successor element.
- The last element has a link value NULL.

To start with, we have to create a node (the first node), and make header point to it.

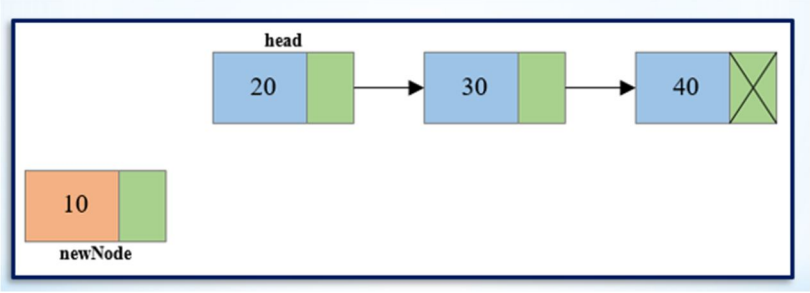
```
newNode = (struct node *)malloc(sizeof(struct node));
newNode->data = data;          //Links the data field with data
newNode->next = NULL;          //Links the address field to NULL
header = newNode;
temp = newNode;
```

It creates a single node. For example, if the data entered is 100 then the list look like

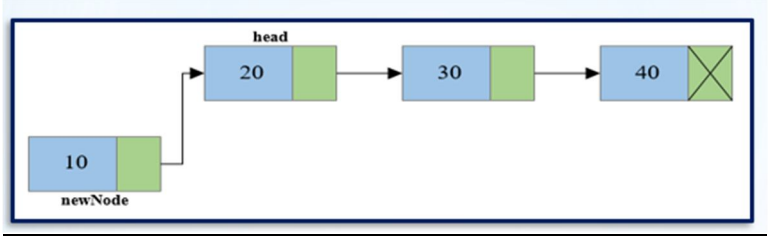
Insertion at Front

Steps to insert node at the beginning of singly linked list

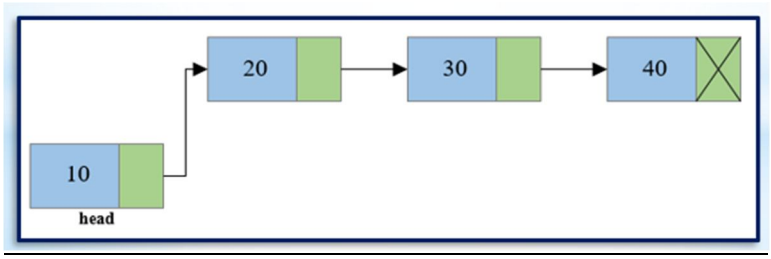
Step 1: Create a new node.



Step 2: Link the newly created node with the head node, i.e. the **newNode** will now point to **head** node.



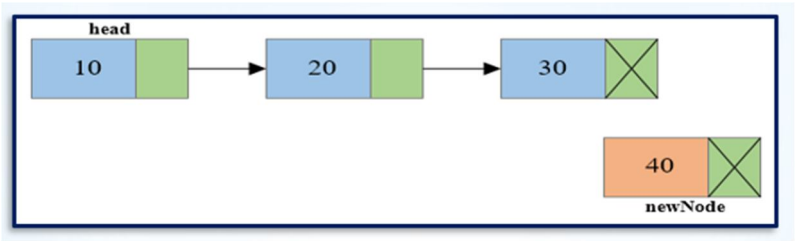
Step 3: Make the new node as the head node, i.e. now **head** node will point to **newNode**.



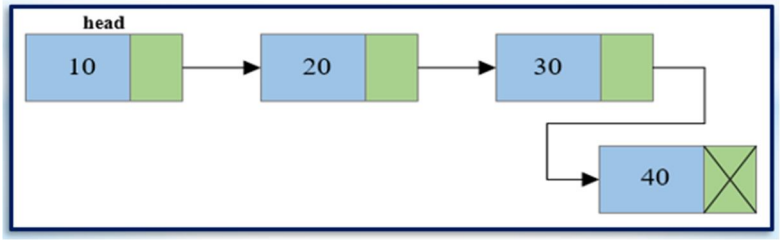
Insertion at the end

Steps to insert node at the end of Singly linked list

Step 1: Create a new node and make sure that the address part of the new node points to NULL. i.e. newNode->next=NULL



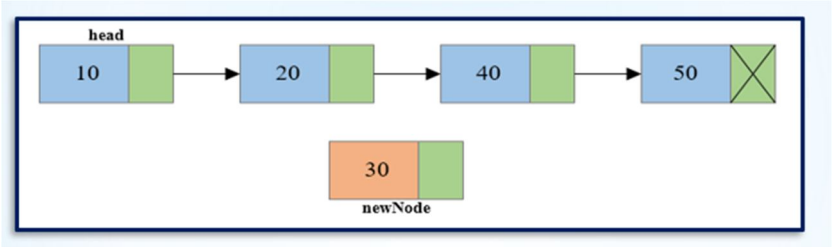
Step 2: Traverse to the last node of the linked list and connect the last node of the list with the new node, i.e. last node will now point to new node. (lastNode->next = newNode).



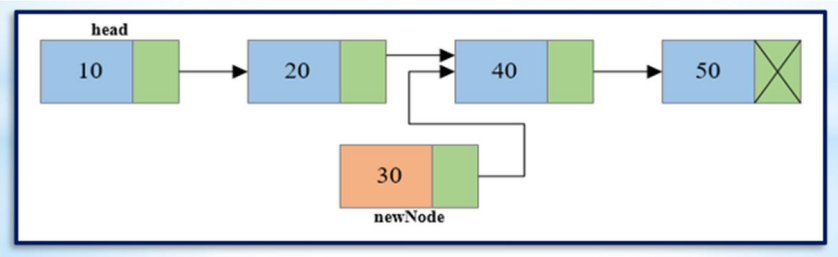
Insertion at a position

Steps to insert node at any position of Singly Linked List

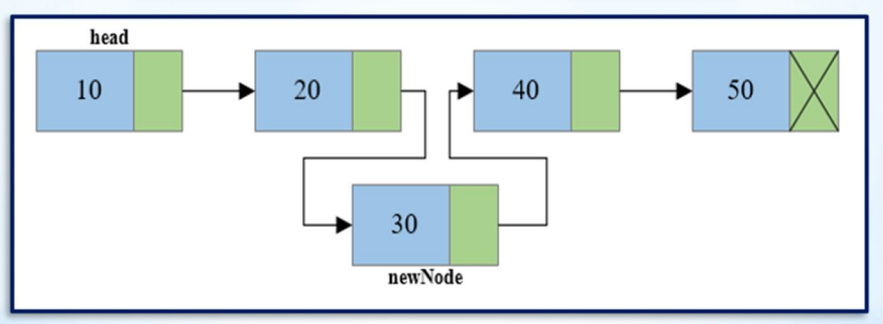
Step 1: Create a new node.



Step 2: Traverse to the n-1th position of the linked list and connect the new node with the n+1th node. (newNode->next = temp->next) where temp is the n-1th node.



Step 3: Now at last connect the n-1th node with the new node i.e. the n-1th node will now point to new node. (temp->next = newNode) where temp is the n-1th node.



Sample Program: Create a Linked list to store and print roll number of n students.

```
#include<stdio.h>
#include<malloc.h>
struct student
{
        int roll;
        char name[30];
        struct student *link;
}*start;
main()
{
        int choice, n, i, data;
        start=NULL;
        printf("How many students' details you want to add:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter the roll:");
                scanf("%d",&data);
                struct student *q, *tmp;
        tmp=(struct student*)malloc(sizeof(struct student));
        tmp->roll=data;
        tmp->link=NULL;
        if(start==NULL)
        {
                start=tmp;
        }
        else
        {
                q=start;
                while(q->link!=NULL)
                        q=q->link;
                        q->link=tmp;
        }
```

```
        }
        struct student *d;
        if(start==NULL)
        {
                printf("No data");
        }
        else
        {
                d=start;
                printf("List is:");
                while(d!=NULL)
                {
                        printf("\n%d", d->roll);
                        d=d->link;
                }
        }

}
```

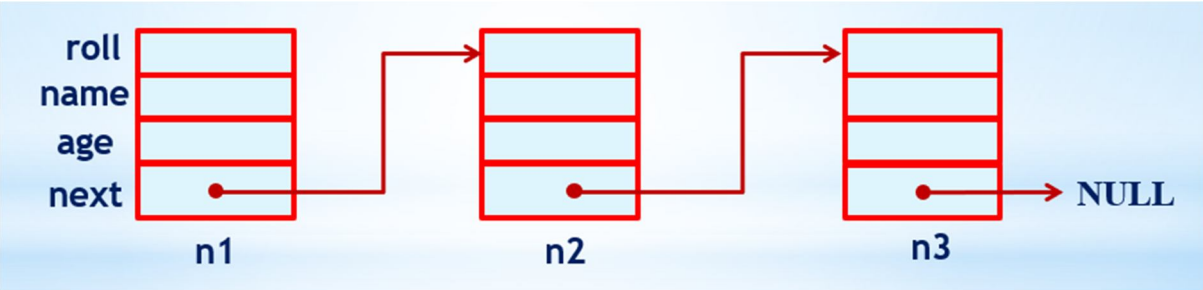
A. Problem: Create a Linked list to store and print roll number, name and age of 3 students.

Description:

Assume the list with three nodes n1, n2 and n3 for 3 students.

Create a structure as a collection three variables – Name, Roll No & Age. Structure will also contain a pointer variable which will point to the next node. Pointer of the last will element will point to NULL.

List will look like:



Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Input and Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

B. Problem: Write a C program to create a Linked list of PAN numbers of 4 employees. Then insert the 5th employee's PAN number at the beginning of the list.
Description:

Create a list of 5 nodes containing integer numbers. Add another node at the beginning of the list.

A header pointer addresses the first node in the list. Each node points at the next node. While adding a node at 1st position, we need to make the header pointer addresses the new node and the new node points at the 1st node.

Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

C. Problem: Write a C program to create a Linked list of Aadhar number of 4 employees. Then insert the 5th employee's Aadhar number in the middle of the list.

Description:

Create a list of 5 nodes containing integer numbers. Add another node at the 3rd position.

A header pointer addresses the first node in the list. Each node points at the next node. While adding a node at 3rd position, we need to make the 2nd node points at new node and pointer of new node will point at the 3rd node. In this way we can add an additional node in between middle of the list.

Objective of Experiment:

Define Linked List:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

D. Problem: Write a C program to create Linked list of storing heights of 10 students. Then (i) delete one student data from middle of the list, (ii) delete one node from the beginning of the remaining list, (iii) delete one node from the end of the remaining list.

Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

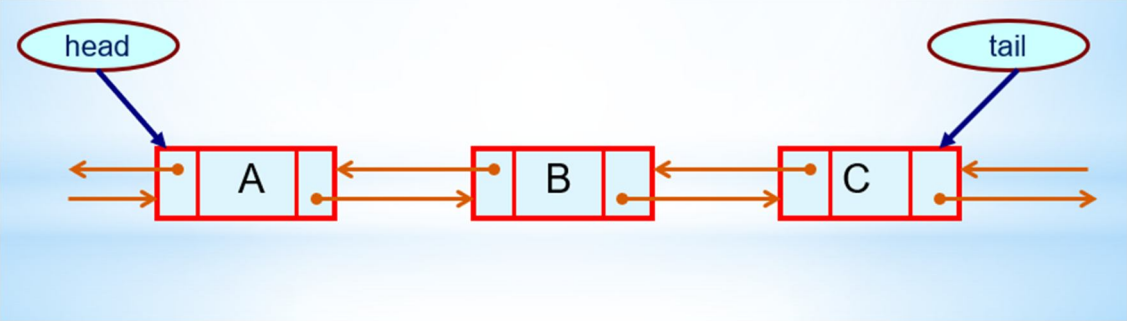
Experiment 5

Double Linked List

Pointers exist between adjacent nodes in both directions.

The list can be traversed either forward or backward.

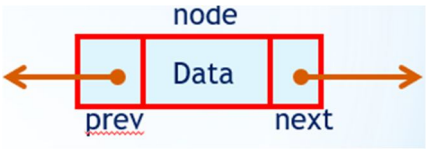
Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Defining a node of Double Linked List

Each node of doubly linked list (DLL) consists of three fields:

- **Item (or) Data**
- **Pointer of the next node in DLL**
- **Pointer of the previous node in DLL**

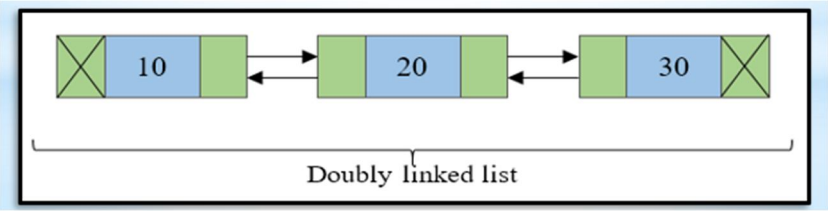


How to define a node of a doubly linked list (DLL)?

```
struct node
{
  int data;
  struct node *next; // Pointer to next node in DLL
  struct node *prev; // Pointer to previous node in DLL
};
```

- **Doubly linked list is a collection of nodes linked together in a sequential way.**
- **Doubly linked list is almost similar to singly linked list except it contains two address or reference fields, where one of the address field contains reference of the next node and other contains reference of the previous node.**
- **First and last node of a linked list contains a terminator generally a NULL value, that determines the start and end of the list.**
- **Doubly linked list is sometimes also referred as bi-directional linked list since it allows traversal of nodes in both direction.**

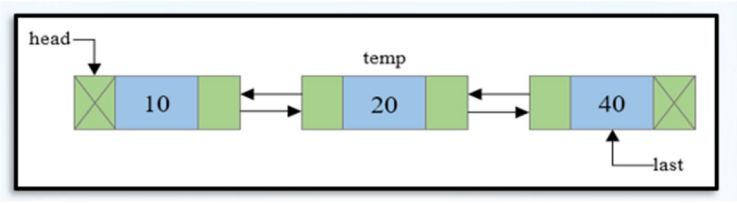
- **Since doubly linked list allows the traversal of nodes in both direction, we can keep track of both first and last nodes.**



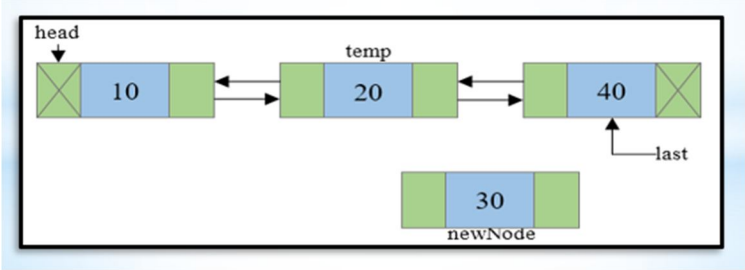
Insert at any Position

Steps to insert a new node at nth position in a Doubly linked list.

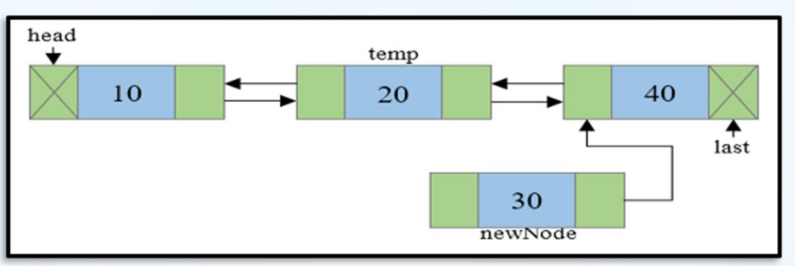
Step 1: Traverse to N-1 node in the list, where N is the position to insert. Say **temp** now points to N-1th node.



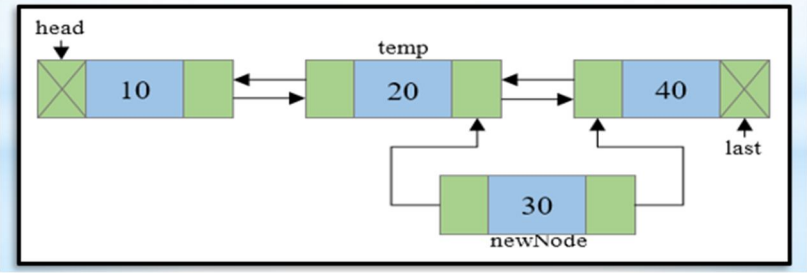
Step 2: Create a **newNode** that is to be inserted and assign some data to its data field.



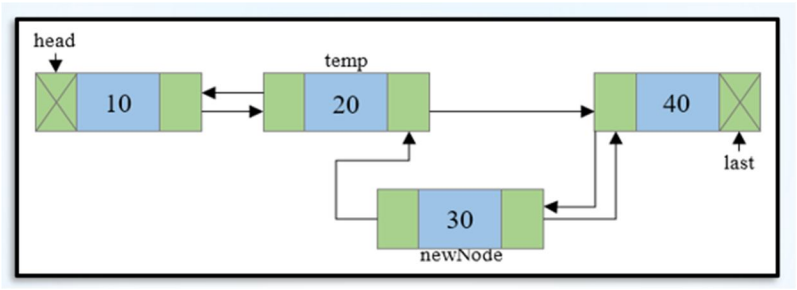
Step 3: Connect the next address field of **newNode** with the node pointed by next address field of **temp** node.



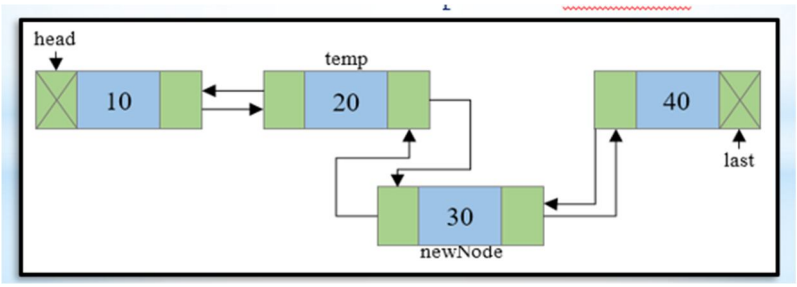
Step 4: Connect the previous address field of **newNode** with the **temp** node.



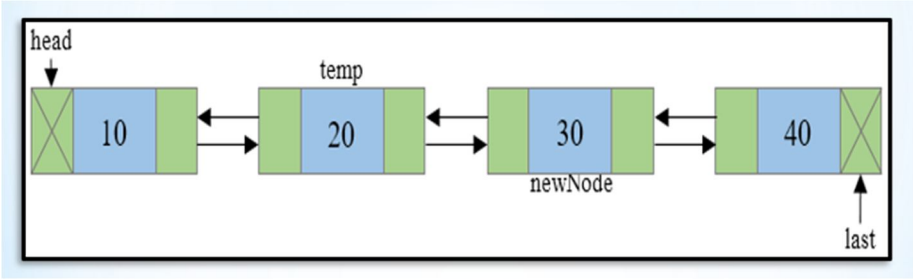
Step 5: Check if temp.next is not NULL then, connect the previous address field of node pointed by temp.next to newNode.



Step 6: Connect the next address field of temp node to newNode.



Step 7: Final doubly linked list looks like



A. Problem: Create a Double Linked list to store and print roll number, name and age of 3 students.

Description:

Assume the list with three nodes n1, n2 and n3 for 3 students.

Create a structure as a collection three variables – Name, Roll No & Age. Structure will also contain a pointer variable which will point to the next node. Pointer of the last will element will point to NULL.

Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Input and Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

B. Problem: Write a C program to create a Double Linked list of PAN numbers of 4 employees. Then insert the 5th employee's PAN number at the beginning of the list.
Description:

Create a list of 5 nodes containing integer numbers. Add another node at the beginning of the list.

A header pointer addresses the first node in the list. Each node points at the next node. While adding a node at 1st position, we need to make the header pointer addresses the new node and the new node points at the 1st node.

Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

C. Problem: Write a C program to create a Double Linked list of Aadhar number of 4 employees. Then insert the 5th employee's Aadhar number in the middle of the list.

Description:

Create a list of 5 nodes containing integer numbers. Add another node at the 3rd position.

A header pointer addresses the first node in the list. Each node points at the next node. While adding a node at 3rd position, we need to make the 2nd node points at new node and pointer of new node will point at the 3rd node. In this way we can add an additional node in between middle of the list.

Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

D. Problem: Write a C program to create Double Linked list of storing heights of 10 students. Then (i) delete one student data from middle of the list, (ii) delete one node from the beginning of the remaining list, (iii) delete one node from the end of the remaining list.

Objective of Experiment:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

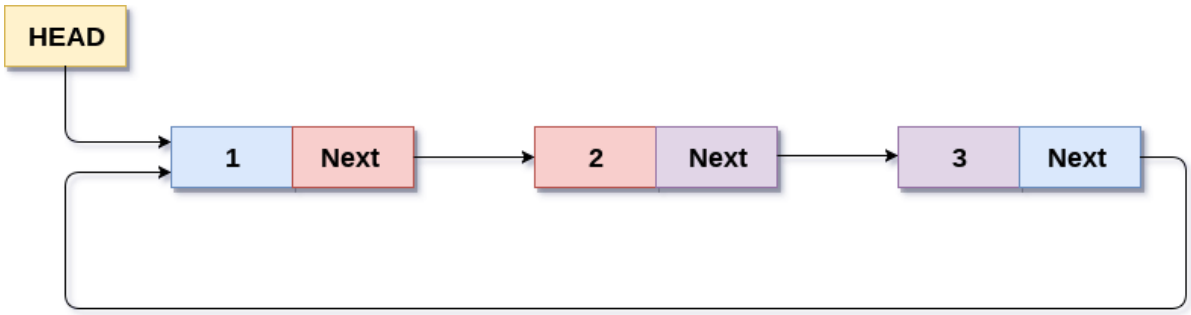
Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 6

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.

2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Problem: Write a C program to create Menu driven program to do following operation on Circular Linked list

1. Create
2. Insert at any position
3. Delete from any position
4. Search the element and delete
5. Reverse

Objective of Experiment:

Define Circular Linked List:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 7

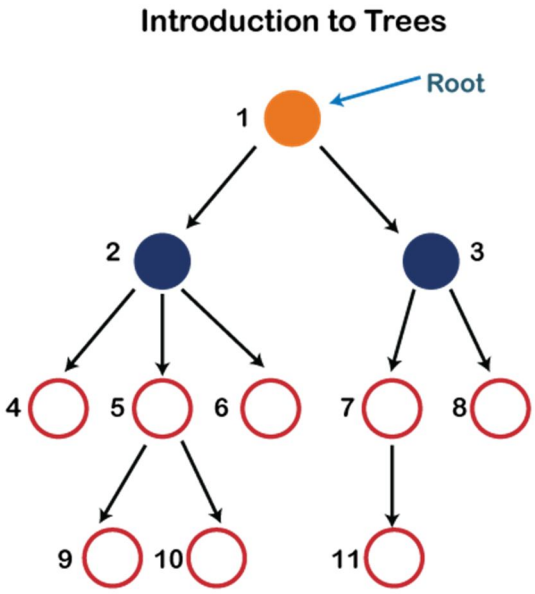
Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



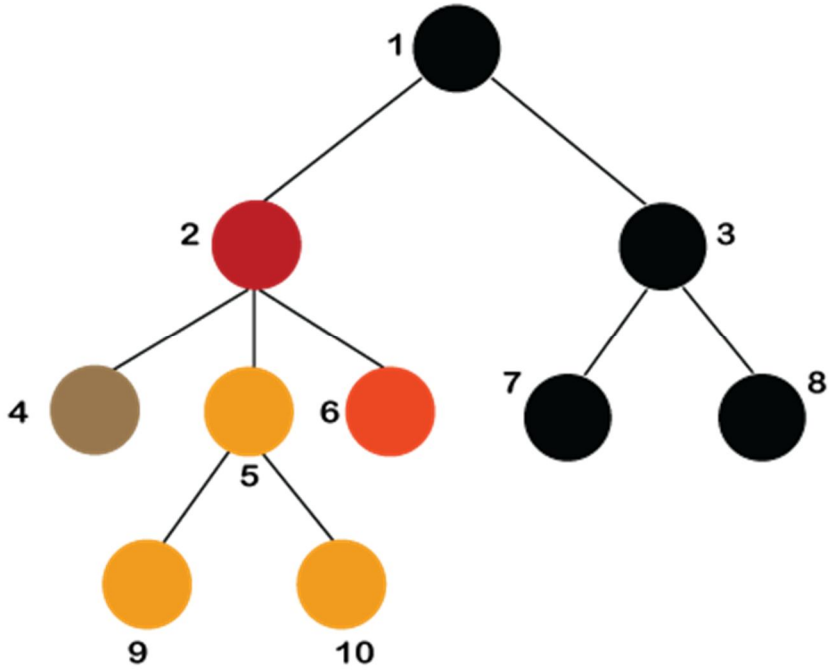
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various

applications.

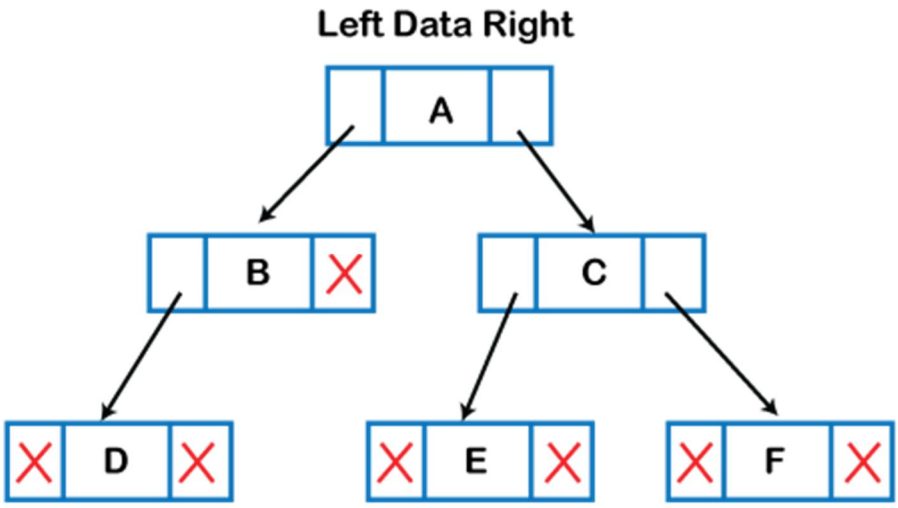


- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

```
1. struct node
2. {
3.   int data;
4. struct node *left;
5. struct node *right;
6. }
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Problem: Write a menu driven program to

i. Create a binary search tree

ii. Traverse the tree in inorder, preorder and postorder

iii. Search the tree for a given node
iv. Search an Element and Delete from the Tree.

Objective of Experiment:

Define Tree and its types:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

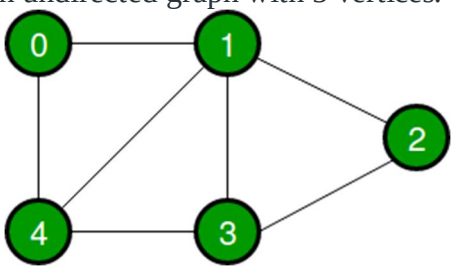
Experiment 8

Aim: Write a program to insert and delete nodes in a graph using adjacency list.

Objective: Implement different graph algorithms and its applications.
Outcome: Students will be able to understand and implement the operations of graph algorithms with the adjacency list representation.
Theory:
Graph is a data structure that consists of the following two components:

- A finite set of vertices also called nodes.
- A finite set of ordered pairs of the form (u, v) called edge. The pair is ordered because (u, v) is not the same as (v, u) in the case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Following is an example of an undirected graph with 5 vertices.



Example of undirected graph with 5 vertices

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale. See this for more applications of graphs.
In computer science, a graph is a data structure that is used to represent connections or relationships between objects. A graph consists of a set of vertices (also known as nodes) and a set of edges (also known as arcs) that connect the vertices. The vertices can represent anything from cities in a map to web pages in a network, and the edges can represent the relationships between them, such as roads or links.
A graph can be visualized as a collection of points (vertices) connected by lines (edges), where each vertex represents a point of interest and each edge represents a connection between two points. The edges can be directed or undirected, meaning they can either have a specific direction or be bidirectional. For example, a map of a city may have directed edges that represent the direction of one-way streets, while a social network may have undirected edges that represent friendships between individuals.
Representations of Graphs:
The following two are the most commonly used representations of a graph.

- Adjacency Matrix
- Adjacency List

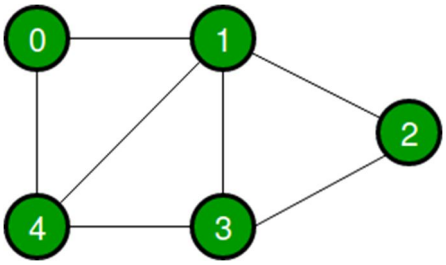
Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. The adjacency matrix for an undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

We follow the below pattern to use the adjacency matrix in code:

- In the case of an undirected graph, we need to show that there is an edge from vertex i to vertex j and vice versa. In code, we assign adj[i][j] = 1 and adj[j][i] = 1.
- In the case of a directed graph, if there is an edge from vertex i to vertex j then we just assign adj[i][j]=1.

See the undirected graph shown below:



Example of undirected graph with 5 vertices

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency matrix representation

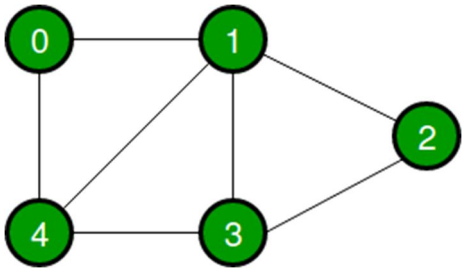
Advantages of Adjacency Matrix:

- Representation is easier to implement and follow.
- Removing an edge takes O(1) time.
- Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

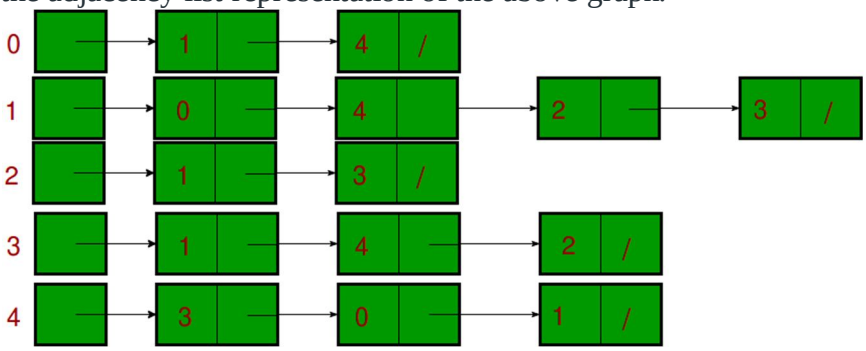
Disadvantages of Adjacency Matrix:

- Consumes more space O(V2). Even if the graph is sparse(contains less number of edges), it consumes the same space.
- Adding a vertex takes O(V2) time. Computing all neighbors of a vertex takes O(V) time (Not efficient).

Adjacency List:
An array of linked lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex.
This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.
Consider the following graph:



Example of undirected graph with 5 vertices
Following is the adjacency list representation of the above graph.



Adjacency List representation of the above graph
Advantages of Adjacency List:

- Saves space. Space taken is O(|V|+|E|). In the worst case, there can be a C(V, 2) number of edges in a graph thus consuming O(V2) space.
- Adding a vertex is easier.
- Computing all neighbors of a vertex takes optimal time.

Disadvantages of Adjacency List:
Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).
Test Cases:

Conclusion: Thus we have studied and implemented the operations of graph algorithms with the adjacency list representation.

Problem: Write a program to insert and delete nodes in graph using adjacency list.

Objective of Experiment:

Define Graph:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

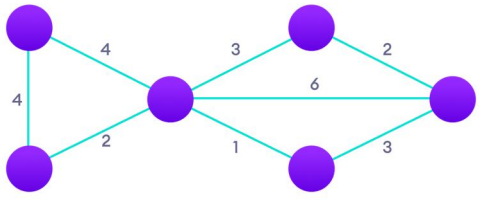
Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 9

Aim: Write a program to implement Dijkstra's shortest path algorithm for a given directed Graph.
Objective: Implement different graph algorithms and its applications.
Outcome: Students will be able to implement and analyze the shortest path algorithm
Theory:
Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.
How Dijkstra's Algorithm works
Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

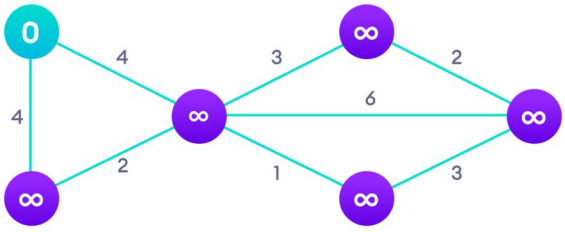
Fig: Each subpath is the shortest path

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.
Example of Dijkstra's algorithm
It is easier to start with an example and then think about the algorithm.



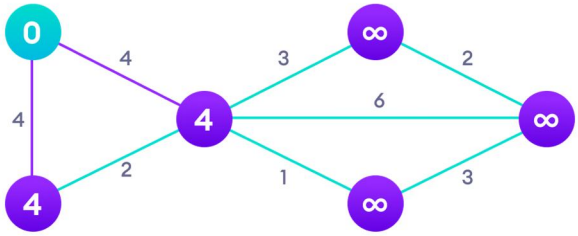
Step: 1

Start with a weighted graph



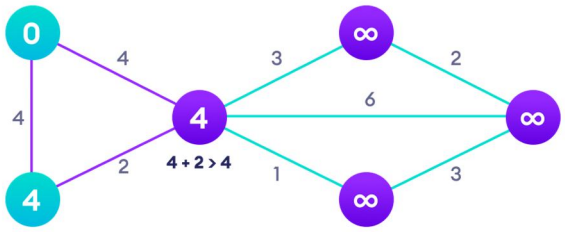
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



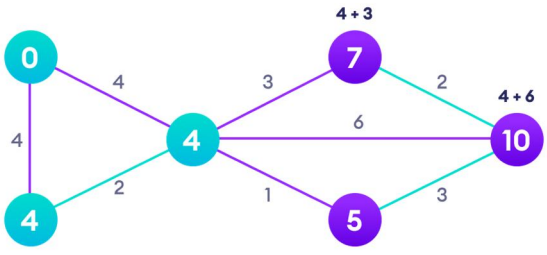
Step: 3

Go to each vertex and update its path length



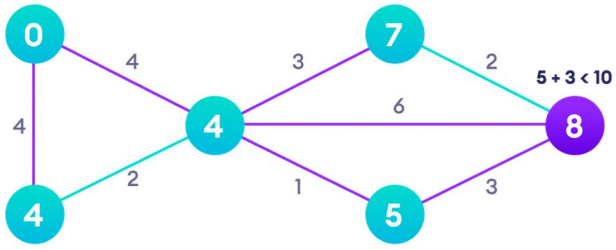
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



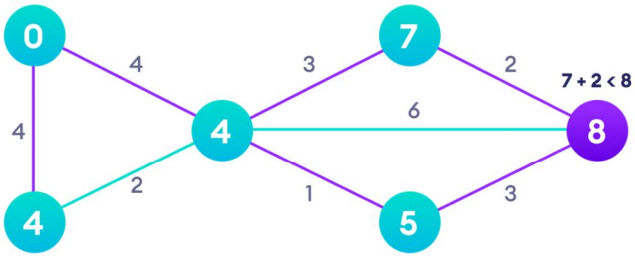
Step: 5

Avoid updating path lengths of already visited vertices



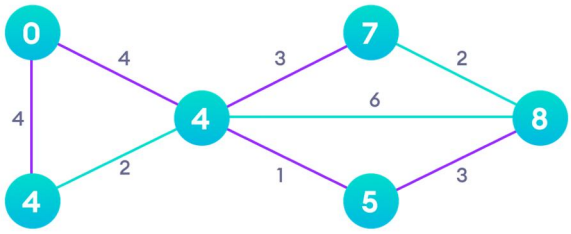
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Dijkstra's Algorithm

1. Create cost matrix C[ ][ ] from adjacency matrix adj[ ][ ]. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinite.
2. Array visited[ ] is initialized to zero.

```
for(i=0;i<n;i++)
visited[i]=0;
```

3. If the vertex 0 is the source vertex then visited[0] is marked as 1.
4. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n-1 from the source vertex 0.

```
for(i=1;i<n;i++)
distance[i]=cost[0][i];
```

Initially, the distance of the source vertex is taken as 0. i.e. distance[0]=0;

5. for(i=1;i<n;i++)

Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark visited[w] as 1.
– Recalculate the shortest distance of remaining vertices from the source.
– Only, the vertices not marked as 1 in the array visited[ ] should be considered for recalculation of distance. i.e. for each vertex v
if(visited[v]==0)
distance[v]=min(distance[v],
distance[w]+cost[w][v])

Time Complexity: The program contains two nested loops each of which has a complexity of O(n). n is the number of vertices. So the complexity of the algorithm is O(n2).

Conclusion: Thus we have implemented and analyzed the Dijkstra's shortest path algorithm

Program: Write a program to implement Dijkstra's shortest path algorithm for a given directed Graph.

Objective of Experiment:

Define shortest path problem:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 10

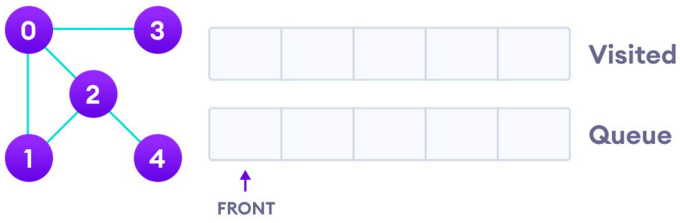
Aim: Write a program in C to implement Breadth First Search using linked representation of graph.
Objective: Implement different graph algorithms and its applications.
Outcome: Students will be able to implement Breadth First Search using a linked representation of graph.
Theory:
Breadth first search - Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.
A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

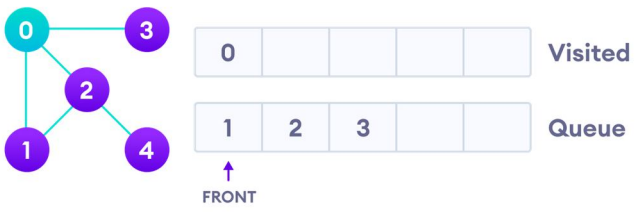
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

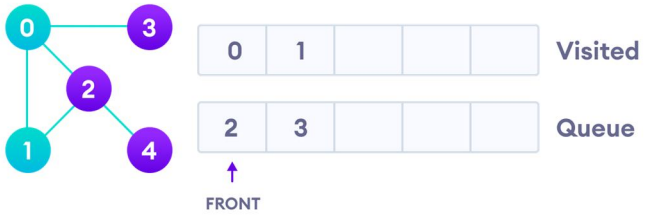
The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node
BFS example
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



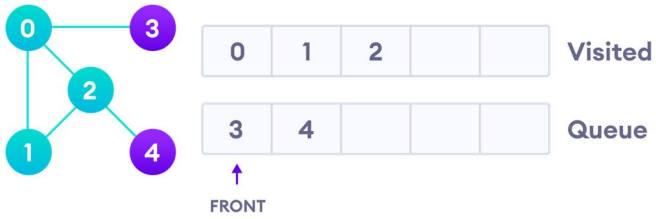
Undirected graph with 5 vertices
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



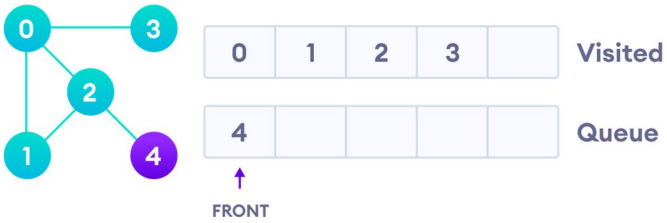
Visit start vertex and add its adjacent vertices to queue
Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



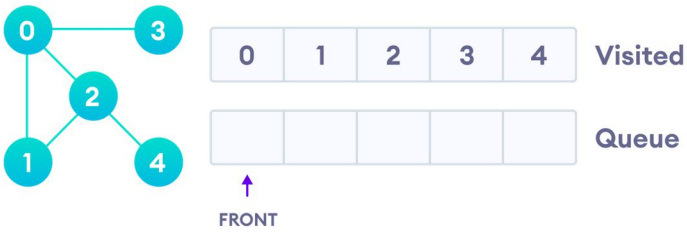
Visit the first neighbor of start node 0, which is 1
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



Visit 2 which was added to queue earlier to add its neighbors



4 remains in the queue
Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Visit last remaining item in the queue to check if it has unvisited neighbors
Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS Algorithm Complexity
The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
The space complexity of the algorithm is O(V).

BFS Algorithm Applications
1. To build index by search index
2. For GPS navigation

3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree

Conclusion: Thus we have implemented Breadth First Search using a linked representation of graph.

Problem: Write a program in C to implement Breadth First Search using linked representation of graph.

Objective of Experiment:

Define Graph and Application of BFS:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.

Experiment 11

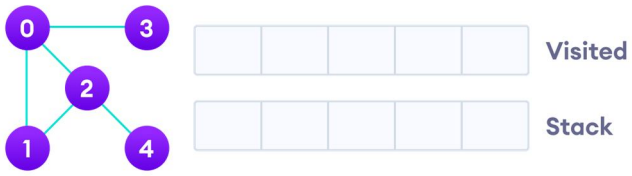
Aim: Write a program in C to implement Depth First Search using linked representation of graph.
Objective: Implement different graph algorithms and its applications.
Outcome: Students will be able to implement Depth First Search using a linked representation of graph.
Theory:
Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.
A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

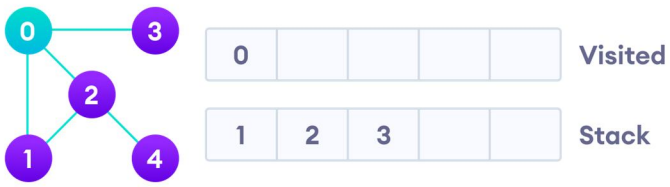
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

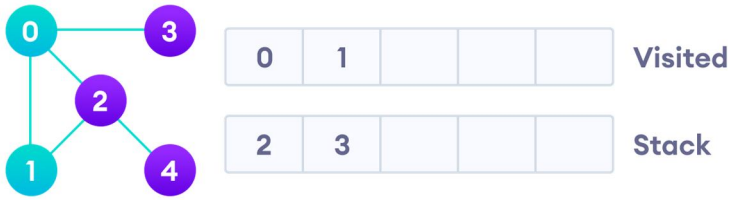
Depth First Search Example
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



Undirected graph with 5 vertices
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

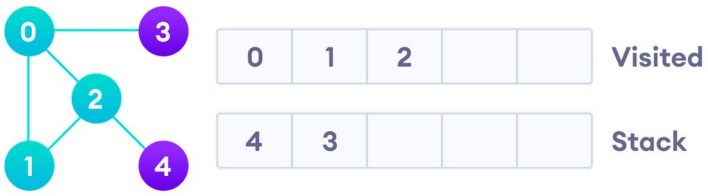


Visit the element and put it in the visited list
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

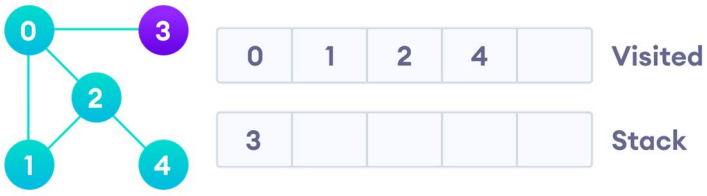


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



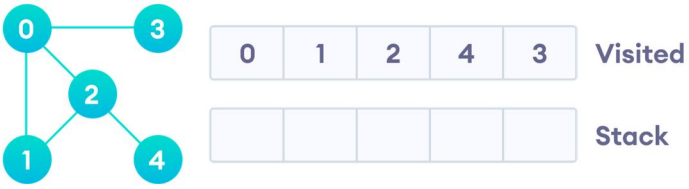
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit



it.

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

DFS Pseudocode

In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)
   u.visited = true
   for each v ∈ G.Adj[u]
      if v.visited == false
         DFS(G,v)

init() {
   For each u ∈ G
      u.visited = false
    For each u ∈ G
     DFS(G, u)
}
```

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
The space complexity of the algorithm is O(V).

Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

Conclusion: Thus we have implemented Depth First Search using a linked representation of graph.

Program: Write a program in C to implement Depth First Search using linked representation of graph.

Objective of Experiment:

Define Graph and Application of DFS:

Flow Chart/Pseudo Code/Algorithm:

Output with Edge Cases:

Source Code, with description and with Output Need to be Uploaded to the Google Classroom.