PANDIT DEENDAYAL ENERGY UNIVERSITY

Formerly Pandit Deendayal Petroleum University (PDPU)

Laboratory Manual
24CS203P: Data Structures Lab

COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF TECHNOLOGY

Name of Student:

Roll No:

Branch:

Sem./Year:

Academic Year:

Pandit Deendayal Energy University

Raisan, Gandhinagar – 380 007, Gujarat, India

Computer Science and Engineering

Certificate

This is to certify that

Mr./Ms. ___________________________________ Roll no.___________

of 3rd semester degree course in Computer Science and Engineering has satisfactorily completed his/her term work in Data Structures Lab (24CS203P) subject during the semester from____________ to ____________ at School of Technology, PDEU.

Date of Submission:

Signature:

Faculty In-charge Head of Department

Index

Name:
Roll No:

Sr. No.	Experiment Title	Pages		Date of Completion	Marks (out of 10)	Sign.
		From	To			
1	Revision of Arrays					
2	Revision of Structures					
3	Revision of Pointers					
4	Stack					
5	Application of Stack					
6	Queue					
7	Linked List					
8	Trees					
9	Graphs					
10	Hashing					

LABORATORY MANUAL

FOR

DATA STRUCTURES (24CS203P)

Odd Semester

Academic Year: 2025-2026

PREPARED BY

DR. ADITYA SHASTRI

ASSISTANT PROFESSOR

Department of Computer Science & Engineering

School of Technology

PANDIT DEENDAYAL ENERGY UNIVERSITY, GANDHINAGAR

DATA STRUCTURES LAB

Course Outcomes (COs):

After completion of this course the students will be able to

1. Recall the differences between primitive and non-primitive datatypes.
2. Apply searching and sorting algorithms on linear data structures.
3. Implement search, insert, and delete operations on data structures.
4. Implement hash tables and collision resolution techniques.
5. Evaluate the complexity of algorithms.
6. Design application using linear and non-linear data structures.

List of Experiments:

Sr. No.	Experiments Performed	Mapped CO	Mapped PO
1	Study and Implement Arrays Data Structures and their Applications.	CO - 1, 2	PO - 1, 2, 3, 5, 8, 12
2	Study and Implement Structures in C.	CO – 1, 2	PO - 1, 2, 8
3	Study and Implement Pointers in C.	CO – 1, 2	PO - 1, 2, 8
4	Study and Implement Linked List data structure.	CO - 1, 2, 3	PO - 1, 2, 3, 5, 8, 12
5	Study and Implement Stack using Array/ Linked List.	CO - 1, 2, 3	PO - 1, 2, 3, 5, 8, 12
6	Study and Implement Queue using Array/ Linked List.	CO - 1, 2, 3	PO - 1, 2, 3, 5, 8, 12
7	Implement program to convert Infix into Postfix notation and evaluate the postfix notation.	CO - 3, 5	PO - 1, 2, 3, 4, 5, 8, 12
8	Study and Implement Tree Data structure and perform pre-order, in-order and post-order traversal.	CO - 3, 5	PO - 1, 2, 3, 4, 5, 8, 12
9	Study and Implement Graph Data structure and perform Breadth First Search and Depth First Search.	CO - 3, 5	PO - 1, 2, 3, 4, 5, 8, 12
10	Study and Implement Hash Function and Tables.	CO - 4	PO - 1, 2, 3, 4, 5, 8, 12
11	Mini Project (For Advanced Learners)	CO - 1, 2, 3, 4, 5, 6	PO - 1, 2, 3, 5, 8, 9, 10, 11, 12

Pre-requisites:

1. Basic knowledge of Computer Programming Fundamentals: Understanding of Arrays, Strings, Structure, Pointers and other fundamentals.
2. LAB: C Programming

List of Experiments

Experiment No. 1 [Revision of Arrays]

Programs covering basics of Arrays: searching, sorting, minimum and maximum elements etc.

1. Write a program in C to perform linear and binary search.

2. Write a program in C to perform bubble sort, insertion sort and selection sort. Take the array size and array elements from user.

3. Write a program in C that obtains the minimum and maximum element from the array. Modify this program to give the second largest and second smallest element of the array.

Experiment No. 2 [Revision of Structures]

Programs covering structure definition, array of structure, nested structures etc.

1. Create a structure Student in C with student name, student roll number and student address as its data members. Create the variable of type student and print the values.

2. Modify the above program to implement arrays of structure. Create an array of 5 students and print their values.

3. Create a structure Organization with organization name and organization ID as its data members. Next, create another structure Employee that is nested in structure Organization with employee ID, employee salary and employee name as its data members. Write a program in such a way that there are two organizations and each of these contains two employees.

Experiment No. 3 [Revision of Pointers]

Programs covering use of pointers with arrays, structures, functions etc.

1. Write a program in C to implement arrays of pointers and pointers to arrays.

2. Write a program in C to implement pointers to structures.

3. Write a program in C to perform swapping of two numbers by passing addresses of the variables to the functions.

Experiment No. 4 [Linked List]

1. Write a program that takes ***two sorted lists*** as inputs and merge them into one sorted list.
   For example, if the first linked list ***A*** is 5 =>10 =>15, and the other linked list ***B*** is 2 => 3 => 20, then output should be 2 => 3 => 5 => 10 => 15 => 20.

2. Write a program to ***insert a new node*** into the linked list. A node can be added into the linked list using three ways: [Write code for all the three ways.]
   a. At the front of the list
   b. After a given node
   c. At the end of the list.

3. Write a program to ***delete a node*** from the linked list. A node can be deleted from the linked list using three ways: [Write code for all the three ways.]
    a. Delete from the beginning
    b. Delete from the end
    c. Delete from the middle.

4. Implement the ***circular linked list*** and perform the operation of traversal on it. In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again.

5. Implement the ***doubly linked list*** and perform the deletion and/ or insertion operation on it. Again, you can perform insertion deletion according to the three ways as given above. Implement all of them according to availability of time.

Experiment No. 5 [Stack Application]

1. Implement a stack using an array and using a linked list.

2. Given a stack, sort it using recursion. Use of any loop constructs like `while`, `for`, etc. is not allowed. We can only use the following functions on Stack S:

a. isEmpty(S):	Tests whether stack is empty or not.
b. push(S):	Adds new element to the stack.
c. pop(S):	Removes top element from the stack.
d. top(S):	Returns value of the top element.

Experiment No. 6 [Stack Applications]

1. Convert the given infix expression into postfix expression using stack.
   Example- Input: $a + b * (c\^{}d - e)\^{}(f + g * h) - i$
   Output: $abcd\^{}e - fgh * +\^{} * +i -$

2. Write a program to evaluate the following given postfix expressions:
    a. $2\ 3\ 1\ *\ +9\ -$ Output: -4
    b. $2\ 2\ +\ 2\ /\ 5\ *\ 7\ +$ Output: 17

3. Given an expression, write a program to examine whether the pairs and the orders of "{", "}", "(", ")", "[", "]" are correct in the expression or not.
   Example: Input: exp = "[( )]{ }{[( )( )]( )}" Output: Balanced
   Input: exp = "[( ])" Output: Not Balanced

Experiment No. 7 [Queue]

1. Implement various functionalities of Queue using Arrays. For example: insertion, deletion, front element, rear element etc.

2. Implement various functionalities of Queue using Linked Lists. Again, you can implement operation given above.

3. Implement Priority Queue, where every element has a priority associated with it. Perform operations like Insertion and Deletion in a priority queue.

4. Implement Double Ended Queue that supports following operation:
    a. insertFront(): Adds an item at the front of Deque.
    b. insertLast(): Adds an item at the rear of Deque.
    c. deleteFront(): Deletes an item from the front of Deque.
    d. deleteLast(): Deletes an item from the rear of Deque.

5. Implement Double Ended Queue that supports following operation:
    a. getFront(): Gets the front item from the queue.
    b. getRear(): Gets the last item from queue.
    c. isEmpty(): Checks whether Deque is empty or not.
    d. isFull(): Checks whether Deque is full or not.

Experiment No. 8 [Trees]

1. Write a program to insert an element, delete an element and search an element in the Binary Tree.

2. Study and implement the Binary Tree and perform following three types of traversals in a binary tree:
    a. Pre-order Traversal
    b. In-order Traversal
    c. Post-order Traversal

3. Given a preorder traversal sequence of a Binary Search Tree, construct the corresponding Binary Search Tree.

Experiment No. 9 [Graphs]

1. For a given graph $G = (V, E)$, study and implement the Breadth First Search (or traversal) i.e., BFS. Also, perform complexity analysis of this algorithm in-terms of time and space.

2. For a given graph $G = (V, E)$, study and implement the Depth First Search (or traversal) i.e., DFS. Also, perform complexity analysis of this algorithm in-terms of time and space.

3. Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. Perform same task for undirected graph as well.

4. Implement Minimum Spanning Tree (MST) using the greedy Kruskal's algorithm. A MST or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Experiment No. 10 [Hashing]

1. Given a limited range array containing both positive and non-positive numbers, i.e., elements are in the range from -MAX to +MAX. Our task is to search if some number is present in the array or not in O(1) time.

2. Given two arrays: *A* and *B*. Find whether *B* is a subset of *A* or not using Hashing. Both the arrays are not in sorted order. It may be assumed that elements in both arrays are distinct.

Experiment No. 11, 12, 13 [Mini Project and Problem Solving]

These laboratory sessions are dedicated for doing a mini project and doubt solving sessions. Students are expected to do a mini project by forming a group of 5 on the concepts they have learned in data structures. They are free to select any mini project as per their choice. If they are not able to decide, they can select any one from the following:

1. **Mini Voting System:**
   An online voting system is a software platform that enables organizations to conduct votes and elections securely. A high-quality online voting system strikes a balance between ballot security, convenience, and the overall needs of a voting event. By collecting the input of your group in a systematic and verifiable manner, online voting tools and online election voting systems assist you in making crucial decisions. These decisions are frequently taken on a yearly basis – either during an event (such as your organization's AGM) or at a specific time of the year. Alternatively, you may conduct regular polls among your colleagues (e.g., anonymous student feedback surveys).

   With this voting system, users can enter their preferences and the total votes and leading candidate can be calculated. It's a straightforward Data structure project that's simple to grasp. Small-scale election efforts can benefit from this.

2. **Library Management System:**
   Library management is a project that manages and preserves electronic book data based on the demands of students. Both students and library administrators can use the system to keep track of all the books available in the library. It allows both the administrator and the student to look for the desired book. You can include the functionalities like searching book, issue books, view available books and more using various data structures.

3. **Electricity Bill Calculator:**
   The Electricity Cost Calculator project is an application-based micro project that predicts the following month's electricity bill based on the appliances or loads used. Visual studio code was used to write the code for this project. This project employs a multi-file and multi-platform strategy (Linux and Windows). People who do not have a technical understanding of calculating power bills can use this program to forecast their electricity bills for the coming months; however, an electricity bill calculator must have the following features:

A. All loads’ power rating
B. Unit consumed per day
C. Units consumed per month, and
D. Total load calculation

4. **Movie Ticket Booking System:**
   The project’s goal is to inform a consumer about the MOVIE TICKET BOOKING SYSTEM so that they can order tickets. The project should have the goal of making the process as simple and quick as possible. The user can book tickets, cancel tickets, and view all booking records using the system. This project’s major purpose is to supply various forms of client facilities as well as excellent customer service. It should meet nearly all the conditions for reserving a ticket.

5. **Bus Reservation System:**
   This mini project is built on the concept of booking bus tickets in advance. The user can check the bus schedule, book tickets, cancel reservations, and check the bus status board using this system. When purchasing tickets, the user must first enter the bus number, after which the system will display the entire number of bus seats along with the passengers’ names, and the user must then enter the number of tickets, seat number, and person’s name.
   This project will require use of arrays, if-else logic, loop statements, and various functions like login( ), cancel( ), etc. for its implementation.

Experiment 1 [Revision of Arrays]

Perform Linear and Binary Search on Array.

Linear Search

Assume that item is in an array in random order and we have to find an item. Then the only way to search for a target item is, to begin with, the first position and compare it to the target. If the item is at the same, we will return the position of the current item. Otherwise, we will move to the next position. If we arrive at the last position of an array and still cannot find the target, we return -1. This is called the Linear search or Sequential search.

```
int search(int array[], int n, int x)
{

    // Going through array sequencially
    for (int i = 0; i < n; i++)
        if (array[i] == x)
            return i;
    return -1;
}
```

Binary Search

In a binary search, however, cut down your search to half as soon as you find the middle of a sorted list. The middle element is looked at to check if it is greater than or less than the value to be searched. Accordingly, a search is done to either half of the given list.

```
int binarySearch(int array[], int x, int low, int high)
{
    // Repeat until the pointers low and high meet each
    // other
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (array[mid] == x)
            return mid;

        if (array[mid] < x)
            low = mid + 1;

        else
            high = mid - 1;
    }

    return -1;
}
```

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

Questions on Arrays for Students to Answer:

1. What do you mean by an Array?
2. How to create an Array?
3. How arrays are stored in memory?
4. Why array is a part of linear data structures?
5. What are the advantages and disadvantages of Array?

Experiment 2 [Revision of Structures]

Study and implementation of Structures in C.

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name. Before you can create structure variables, you need to define its data type. To define a struct, the "***struct***" keyword is used.

Syntax of struct:

```
struct structureName
{
   datatype member1;
   datatype member2;
   datatype member3;
   -----
};
```

For example:

```
struct address
{
   char name[50];
   char street[100];
   char city[50];
   int pin;
};
```

Declare structure variables: A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
struct Point
{
   int x, y;
} p1;  //The variable p1 is declared with 'Point'


//A variable declaration like basic data types
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1; //The variable p1 is declared like a normal variable
}
```

Initialize structure members: Structure members cannot be initialized with declaration. The reason for this is, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created. Structure members can be initialized using curly braces ‘{}’. For example, following is a valid initialization.

```c
struct Point
{
   int x, y;
};

int main()
{
   // A valid initialization. member x gets value 0 and y
   // gets value 1.  The order of declaration is followed.
   struct Point p1 = {0, 1};
}
```

Access elements of Structures: Structure members are accessed using dot (.) operator.

```c
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1 = {0, 1};

   // Accessing members of point p1
   p1.x = 20;
   printf("x = %d, y = %d", p1.x, p1.y);

   return 0;
}
```

Questions on Structures for students to Answer:

1. What do you mean by structures?
2. What are the various applications of structures?
3. How structures are different than classes in C++ and Java?
4. Can we have the nested structures and how it is performed?
5. How can we allocate the memory dynamically to structures?

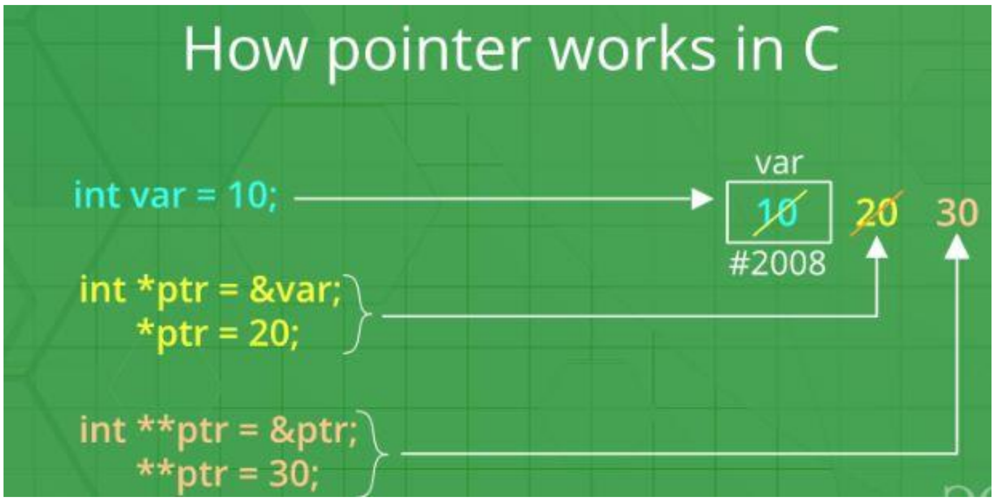
Experiment 3 [Revision of Pointers]

Study and implementation of Pointers in C.

Pointers store address of variables or a memory location.

Syntax:

```
datatype *var_name;

// An example pointer "ptr" that holds
// address of an integer variable or holds
// address of a memory whose value(s) can
// be accessed as integer values through "ptr"

int *ptr;
```



Using a pointer: To use pointers in C, we must understand below two operators.

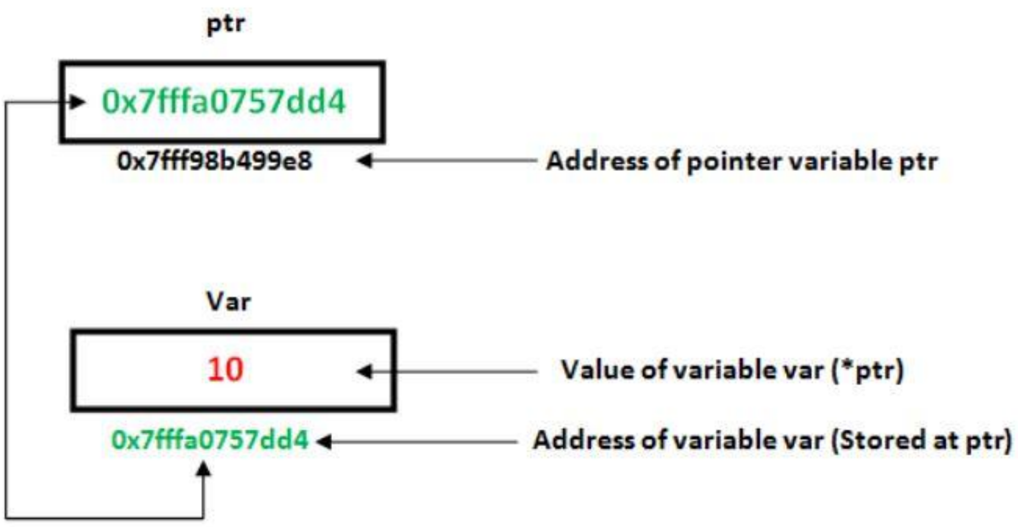
1. To access address of a variable to a pointer, we use the unary operator & (ampersand) that returns the address of that variable. For example, the entity $\&\boldsymbol{x}$ gives us address of variable x.
2. One more operator is unary * (Asterisk) which is used for two things: To declare a pointer variable: When a pointer variable is declared in C/C++, there must be a * before its name.

To access the value stored in the address we use the unary operator (*) that returns the value of the variable located at the address specified by its operand. This is also called Dereferencing.

```c
int main()
{
    // A normal integer variable
    int Var = 10;

    // A pointer variable that holds address of var.
    int *ptr = &Var;

    // This line prints value at address stored in ptr.
    // Value stored is value of variable "var"
    printf("Value of Var = %d\n", *ptr);

    // The output of this line may be different in different
    // runs even on same machine.
    printf("Address of Var = %p\n", ptr);

    // We can also use ptr as lvalue (Left hand
    // side of assignment)
    *ptr = 20; // Value at address is now 20

    // This prints 20
    printf("After doing *ptr = 20, *ptr is %d\n", *ptr);

    return 0;
}
```

Output:

```
Value of Var = 10

Address of Var = 0x7fffa057dd4

After doing *ptr = 20, *ptr is 20
```


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

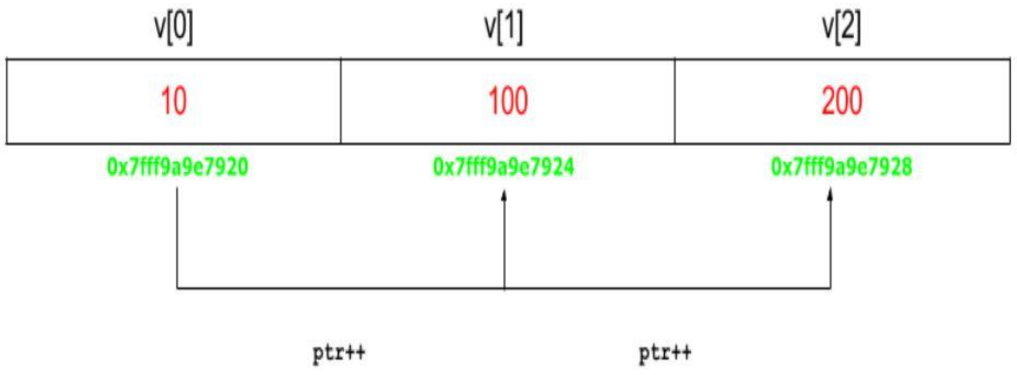
Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers. A pointer may be:

- incremented (+ +)
- decremented (- -)
- an integer may be added to a pointer (+ or + =)
- an integer may be subtracted from a pointer (– or - =)

Pointer arithmetic is meaningless unless performed on an array.

Note: Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between these two addresses.



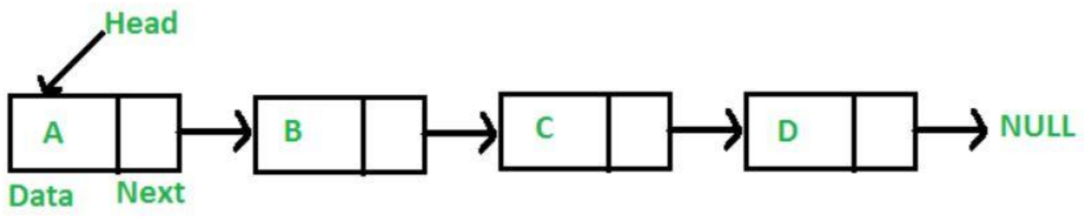
Questions on Pointers for students to Answer:

1. What do you mean by Pointers?
2. Why pointers are used?
3. What are the advantages of pointers?
4. Why pointers are removed from other programming languages like C++ and Java?
5. How can we define a pointer that can point to another structure and how to access the elements of that structure using this pointer?

Experiment 4 [Linked List]

Study and implement Linked List Data Structure.

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes. Here, each node stores the data and the address of the next node.



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list. Arrays can be used to store linear data of similar types, but arrays have the following limitations:

1. **The size of the arrays is fixed:** So, we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2. **Insertion of a new element/ Deletion of a existing element in an array of elements is expensive:** The room has to be created for the new elements and to create room existing elements have to be shifted but in Linked list if we have the head node then we can traverse to any node through it and insert new node at the required position.

Drawback of Linked List:

1. **Random access is not allowed.** We have to access elements sequentially starting from the first node (head node). So, we cannot do a binary search with linked lists efficiently with its default implementation.
2. Extra memory space for a pointer is required with each element of the list.
3. **Not cache friendly.** Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

In C, we can represent a node using structures. Below is an example of a linked list node with integer data.

```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```

Program for a simple linked list with three nodes:

```
struct Node {
    int data;
    struct Node* next;
};

int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

// Three blocks have been allocated dynamically. We have pointers to
these three blocks as head, second and third
       head           second           third
        |                |                |
    +---+-----+     +----+----+     +----+----+
    | #  | #  |     | #  | #  |     |  # |  # |
    +---+-----+     +----+----+     +----+----+

   # Represents any random value. Data is random because we haven't
assigned anything yet

    head->data = 1; // assign data in first node
    head->next = second; // Link first node with the second node

    second->data = 2; // assign data to second node
    second->next = third; // Link second node with the third node

    third->data = 3; // assign data to third node
    third->next = NULL;

     We have the linked list ready.
            head
             |
        +---+---+     +---+---+       +----+------+
        | 1  | o----->|  2  | o-----> |  3 | NULL |
        +---+---+     +---+---+       +----+------+

   Note that only head is sufficient to represent the whole list.  We
can traverse the complete list by following next pointers.

    return 0;
}
```

Types of Linked Lists:

1. Simple Linked List – In this type of linked list, one can move or traverse the linked list in only one direction
2. Doubly Linked List – In this type of linked list, one can move or traverse the linked list in both directions (Forward and Backward)
3. Circular Linked List – In this type of linked list, the last node of the linked list contains the link of the first/head node of the linked list in its next pointer and the first/head node contains the link of the last node of the linked list in its prev pointer

Basic operations on Linked Lists:

1. Deletion
2. Insertion
3. Search
4. Display

Time Complexity:

Time Complexity	Worst Case	Average Case
Search	$O(n)$	$O(n)$
Insert	$O(1)$	$O(1)$
Deletion	$O(1)$	$O(1)$

Auxiliary Space: $O(N)$

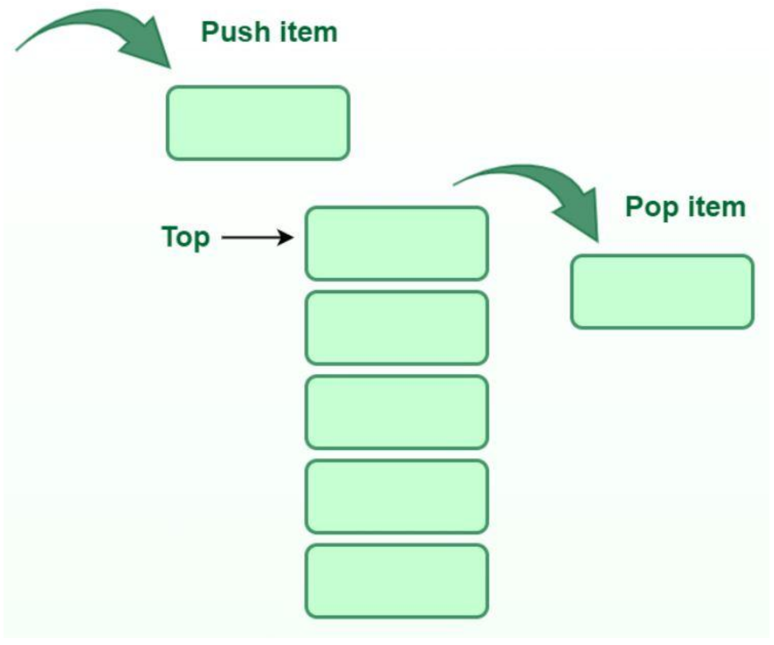
Questions on Linked List for students to Answer:

1. Why Linked List is a linear data structures even though it is not stored in a continuous memory location?
2. What are the advantages of Linked List over Arrays?
3. What are the disadvantages of Linked List?
4. What are the different types of Linked List and what advantages they have over each other?
5. Why the insertion and deletion complexities for Linked List are $O(1)$?

Experiment 5 [Stack]

Study and implement Stack as a linear data structure.

It is a linear data structure that follows a particular order in which the operations are performed. Last In First Out (LIFO) is the strategy in which the element that is inserted last will come out first. For example, you can take a pile of plates kept on top of each other. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.



Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations provided to us.

1. *push*() - Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.
2. *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.
3. *top*() - returns the top element of the stack.
4. *isEmpty*() - returns true is stack is empty, else false
5. *size*() - returns the size of stack

Algorithm for push:

```
begin
 if stack is full
    return
 endif
else
 increment top
 stack[top] assign value
end else
end procedure
```

Algorithm for pop:

```
begin
 if stack is empty
    return
 endif
else
 store value of stack[top]
 decrement top
 return value
end else
end procedure
```

Algorithm for Top:

```
begin
  return stack[top]
end procedure
```

Algorithm for isEmpty:

```
begin
 if top < 1
    return true
 else
    return false
end procedure
```

Time Complexities for various operations:

$push()$	$O(1)$
$pop()$	$O(1)$
$isEmpty()$	$O(1)$
$size()$	$O(1)$

Implementation of stack using arrays:

```
// A structure to represent a stack
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
    return stack->top == stack->capacity - 1;
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}
```

```
// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
Top element is: 20
Elements present in stack: 20 10
```

Questions on Stack for students to Answer:

1. How are stacks different from arrays?
2. What is the difference between the implementation of stack using array and linked list?
3. What are the various applications where stacks are used?
4. How stacks are used in recursive function calls?
5. Compare stack with queue.

Experiment 6 [Stack Application]

Study the conversion of infix to postfix conversion and evaluate the postfix notation.

Infix expression: The expression of the form $a\ operator\ b$ i.e., $(a\ +\ b)$. When an operator is in-between every pair of operands.

Postfix expression: The expression of the form $a\ b\ operator$ i.e., $(ab+)$. When an operator is followed by every pair of operands.

For examples:

Input: $A + B * C + D$

Output: $ABC * +D +$

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

Output: $AB + CDE/* -F +$

Steps to convert Infix to Postfix:

1. Initialize the Stack.
2. Scan the operator from left to right in the infix expression.
3. If the leftmost character is an operand, set it as the current output to the Postfix string.
4. And if the scanned character is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.
5. If the scanned operator has higher precedence than the existing precedence operator in the Stack or if the Stack is empty, put it on the Stack.
6. If the scanned operator has lower precedence than the existing operator in the Stack, pop all the Stack operators. After that, push the scanned operator into the Stack.
7. If the scanned character is a left bracket '(', push it into the Stack.
8. If we encountered right bracket ')', pop the Stack and print all output string character until '(' is encountered and discard both the bracket.
9. Repeat all steps from 2 to 8 until the infix expression is scanned.
10. Print the Stack output.
11. Pop and output all characters, including the operator, from the Stack until it is not empty.

Algorithm to evaluate Postfix:

1. Add ')' at the end of postfix expression as a sentinel.
2. Scan the expression from left to right.
3. If an operand is encountered, push it onto the stack.
4. If an operator $\oplus$ is encountered, the a = pop first element from stack and b = pop second element from the stack, perform $(b\ \oplus\ a)$ and push the result back in stack.
5. Repeat steps 3 and 4 until ')' is reached and print the top value of stack.

C Program to convert infix to postfix:

```c
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

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

    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;

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

    return stack;
}

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

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

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

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

```c
int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z')
           || (ch >= 'A' && ch <= 'Z');
}

// A utility function to return precedence of a given operator
// Higher returned value means higher precedence
int Prec(char ch)
{
    switch (ch) {
    case '+':
    case '-':
        return 1;

    case '*':
    case '/':
        return 2;

    case '^':
        return 3;
    }
    return -1;
}

// The main function that converts given infix to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k;

    struct Stack* stack = createStack(strlen(exp));
    if (!stack) // See if stack was created successfully
        return -1;

    for (i = 0, k = -1; exp[i]; ++i) {

        // If the scanned character is
        // an operand, add it to output.
        if (isOperand(exp[i]))
            exp[++k] = exp[i];

        // If the scanned character is an
        // '(', push it to the stack.
        else if (exp[i] == '(')
            push(stack, exp[i]);

        // If the scanned character is an ')',
        // pop and output from the stack
        // until an '(' is encountered.
        else if (exp[i] == ')') {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1; // invalid expression
            else
                pop(stack);
        }

        else // an operator is encountered
        {
            while (!isEmpty(stack)
                   && Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }
    }

    // pop all the operators from the stack
    while (!isEmpty(stack))
        exp[++k] = pop(stack);

    exp[++k] = '\0';
    printf("%s", exp);
}

int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";

      // Function call
    infixToPostfix(exp);
    return 0;
}
```

Output: $abcd\hat{}e - fgh * +\hat{} * +i -$

C Program to evaluate the postfix notation:

```
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};
```

```c
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

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

    if (!stack->array) return NULL;

    return stack;
}

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

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

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

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


int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

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

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

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

int main()
{
    char exp[] = "231*+9-";
    printf ("postfix evaluation: %d", evaluatePostfix(exp));
    return 0;
}
```

Output: postfix evaluation: -4

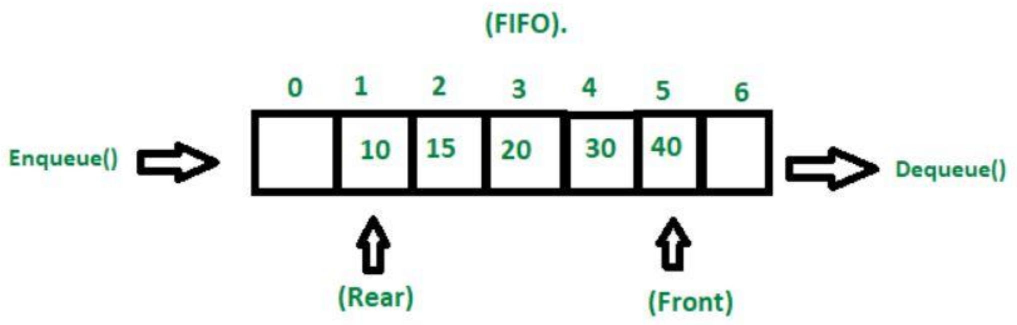
Experiment 7 [Queue]

Study and implement Queue as a linear data structure.

A queue is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order. We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.

FIFO Principle of Queue:

1. A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e., First come first serve).
2. Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue (sometimes, head of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the rear (or the tail) of the queue. See the below figure.



Characteristics of Queue:

1. Queue can handle multiple data.
2. We can access both ends.
3. They are fast and flexible.

Types of Queues:

1. Input Restricted Queue
2. Output Restricted Queue
3. Double-Ended Queue (Deque)
4. Circular Queue
5. Priority Queue

Array Representation:

Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

1. Queue: the name of the array storing queue elements.
2. Front: the index where the first element is stored in the array representing the queue.
3. Rear: the index where the last element is stored in an array representing the queue.

```
// Creating an empty queue

using namespace std;
// A structure to represent a queue
class Queue {
public:
    int front, rear, size;
    unsigned cap;
    int* arr;
};

// Function to create a queue of given capacity
// It initializes size of queue as 0
Queue* createQueue(unsigned cap)
{
    Queue* queue = new Queue();
    queue->cap = cap;
    queue->front = queue->size = 0;

    queue->rear = cap - 1;
    queue->arr = new int[(queue->cap * sizeof(int))];
    return queue;
}
```

Linked List Representation:

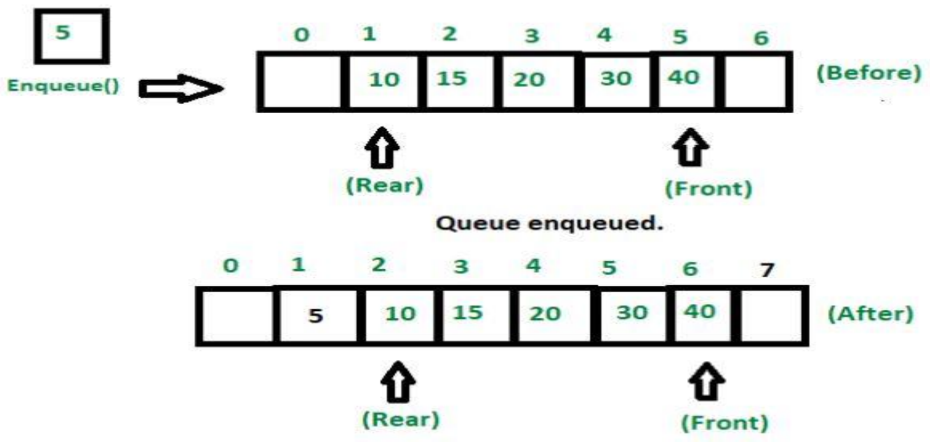
A queue can also be represented in Linked-lists, Pointers, and Structures.

```
struct QNode
{
    int data;
    QNode* next;
    QNode(int d)
    {
        data = d;
        next = NULL;
    }
};

struct Queue
{
    QNode *front, *rear;
    Queue()
    {
        front = rear = NULL;
    }
};
```

***Enqueue*()**: Adds (or stores) an element to the end of the queue.

The following steps should be taken to enqueue (insert) data into a queue:

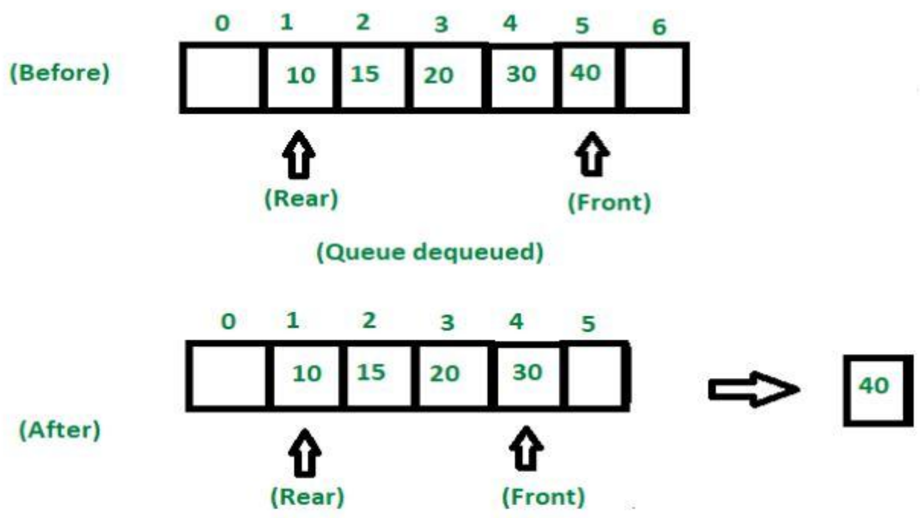
- Step 1: Check if the queue is full.
- Step 2: If the queue is full, return overflow error and exit.
- Step 3: If the queue is not full, increment the rear pointer to point to the next empty space.
- Step 4: Add the data element to the queue location, where the rear is pointing.
- Step 5: return success.



***Dequeue*()**: Removes (or access) the first element from the queue.

The following steps are taken to perform the dequeue operation:

- Step 1: Check if the queue is empty.
- Step 2: If the queue is empty, return the underflow error and exit.
- Step 3: If the queue is not empty, access the data where the front is pointing.
- Step 4: Increment the front pointer to point to the next available data element.
- Step 5: The Return success.



```cpp
// Implementation of queue(enqueue, dequeue).
#include <bits/stdc++.h>
using namespace std;

class Queue {
public:
    int front, rear, size;
    unsigned cap;
    int* arr;
};

Queue* createQueue(unsigned cap)
{
    Queue* queue = new Queue();
    queue->cap = cap;
    queue->front = queue->size = 0;

    queue->rear = cap - 1;
    queue->arr = new int[(queue->cap * sizeof(int))];
    return queue;
}

int isFull(Queue* queue)
{
    return (queue->size == queue->cap);
}

int isempty(Queue* queue) { return (queue->size == 0); }
// Function to add an item to the queue. It changes rear and size.
void enqueue(Queue* queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->cap;
    queue->arr[queue->rear] = item;
    queue->size = queue->size + 1;
    cout << item << " enqueued to queue\n";
}
// Function to remove an item from queue. It changes front and size.
int dequeue(Queue* queue)
{
    if (isempty(queue))
        return INT_MIN;
    int item = queue->arr[queue->front];
    queue->front = (queue->front + 1) % queue->cap;
    queue->size = queue->size - 1;
    return item;
}
```

```
int front(Queue* queue)
{
    if (isempty(queue))
        return INT_MIN;
    return queue->arr[queue->front];
}
int rear(Queue* queue)
{
    if (isempty(queue))
        return INT_MIN;
    return queue->arr[queue->rear];
}

int main()
{
    Queue* queue = createQueue(1000);
    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);
    cout << dequeue(queue);
    cout << " dequeued from queue\n";
    cout << "Front item is " << front(queue) << endl;
    cout << "Rear item is " << rear(queue);
    return 0;
}
```

Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue
Front item is 20
Rear item is 40
```

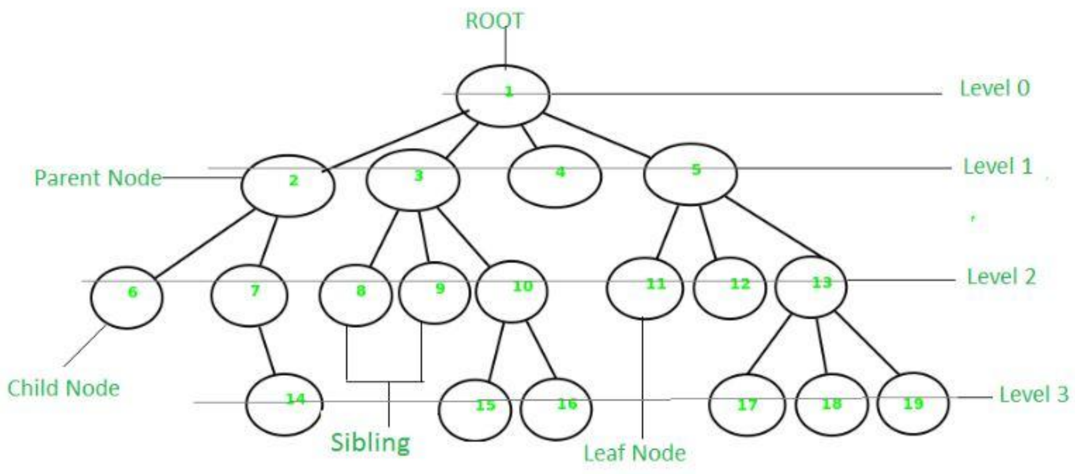
Questions on Queue for students to Answer:

1. What data structure can be used to implement a priority queue?
2. Queues are used for what purpose?
3. In data structures, what is a double-ended queue?
4. What is better, a stack or a queue?
5. Can you represent a queue using two stacks? If no, why? If yes. How?

Experiment 8 [Trees]

Study and implement Tree as a non-linear data structure.

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the children"). This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



Syntax:

```
struct Node
{
   int data;
   struct Node *left_child;
   struct Node *right_child;
};
```

Basic Terminologies in Tree Data Structure:

1. **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.
2. **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.
3. **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
4. **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree.
5. **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {1, 2} are the ancestor nodes of the node {7}

6. **Descendant:** Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.
7. **Sibling:** Children of the same parent node are called siblings. {8, 9, 10} are called siblings.
8. **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
9. **Internal node:** A node with at least one child is called Internal Node.
10. **Neighbour of a Node:** Parent or child nodes of that node are called neighbours of that node.
11. **Subtree:** Any node of the tree along with its descendant.

Basic Operations on Tree:

1. Create – create a tree in data structure.
2. Insert − Inserts data in a tree.
3. Search − Searches specific data in a tree to check it is present or not.
4. Pre-order Traversal – perform Traveling a tree in a pre-order manner in data structure.
5. In-order Traversal – perform Traveling a tree in an in-order manner.
6. Post-order Traversal – perform Traveling a tree in a post-order manner.

CPP Program to demonstrate the Tree Data Structures:

```
using namespace std;
// Function to add an edge between vertices x and y
void addEdge(int x, int y, vector<vector<int> >& adj)
{
    adj[x].push_back(y);
    adj[y].push_back(x);
}
// Function to print the parent of each node
void printParents(int node, vector<vector<int> >& adj,
                  int parent)
{
    // current node is Root, thus, has no parent
    if (parent == 0)
        cout << node << "->Root" << endl;
    else
        cout << node << "->" << parent << endl;
    // Using DFS
    for (auto cur : adj[node])
        if (cur != parent)
            printParents(cur, adj, node);
}
// Function to print the children of each node
void printChildren(int Root, vector<vector<int> >& adj)
{
    // Queue for the BFS
    queue<int> q;
    // pushing the root
```

```cpp
    q.push(Root);
    // visit array to keep track of nodes that have been visited
    int vis[adj.size()] = { 0 };
    // BFS
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        vis[node] = 1;
        cout << node << "-> ";
        for (auto cur : adj[node])
            if (vis[cur] == 0) {
                cout << cur << " ";
                q.push(cur);
            }
        cout << endl;
    }
}
// Function to print the leaf nodes
void printLeafNodes(int Root, vector<vector<int> >& adj)
{
    // Leaf nodes have only one edge and are not the root
    for (int i = 1; i < adj.size(); i++)
        if (adj[i].size() == 1 && i != Root)
            cout << i << " ";
    cout << endl;
}
// Function to print the degrees of each node
void printDegrees(int Root, vector<vector<int> >& adj)
{
    for (int i = 1; i < adj.size(); i++) {
        cout << i << ": ";
        // Root has no parent, thus, its degree is equal to
        // the edges it is connected to
        if (i == Root)
            cout << adj[i].size() << endl;
        else
            cout << adj[i].size() - 1 << endl;
    }
}
int main()
{
    int N = 7, Root = 1;
    // Adjacency list to store the tree
    vector<vector<int> > adj(N + 1, vector<int>());
    addEdge(1, 2, adj);
    addEdge(1, 3, adj);
    addEdge(1, 4, adj);
    addEdge(2, 5, adj);
    addEdge(2, 6, adj);
    addEdge(4, 7, adj);
    // Printing the parents of each node
    cout << "The parents of each node are:" << endl;
    printParents(Root, adj, 0);

    // Printing the children of each node
    cout << "The children of each node are:" << endl;
    printChildren(Root, adj);

    // Printing the leaf nodes in the tree
    cout << "The leaf nodes of the tree are:" << endl;
    printLeafNodes(Root, adj);

    // Printing the degrees of each node
    cout << "The degrees of each node are:" << endl;
    printDegrees(Root, adj);

    return 0;
}
```

Output:

```
The parents of each node are:

1->Root
2->1
5->2
6->2
3->1
4->1
7->4
The children of each node are:
1-> 2 3 4
2-> 5 6
3->
4-> 7
5->
6->
7->
The leaf nodes of the tree are:
3 5 6 7
The degrees of each node are:
1: 3
2: 2
3: 0
4: 1
5: 0
6: 0
7: 0
```

Types of Tree data structures:

1. General Tree
2. Binary Tree
3. Balanced Tree
4. Binary Search Tree

Applications of Tree data structures:

1. Spanning Trees
2. Binary Search Trees
3. Storing Hierarchical data
4. Syntax Tree
5. Heap

Questions on Trees for students to Answer:

1. Why Tree is considered a non-linear data structure?
2. How it is different graph data structure?
3. What are the applications of this data structures apart from the ones that are mentioned above?
4. What are the complexities of various operations that are performed on trees?
5. What is the speciality of Heap tree?

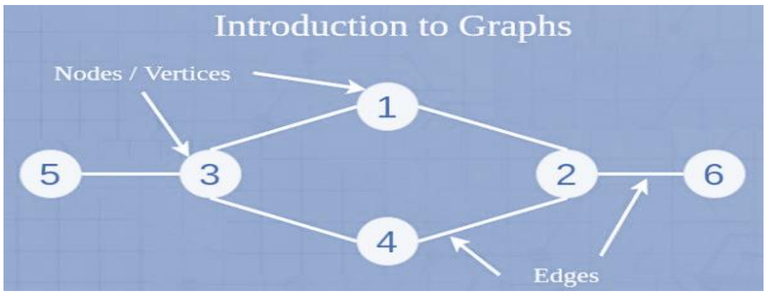
Experiment 9 [Graphs]

Study and implement Graph data structure and perform either BFS or DFS.

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally, a Graph is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by $G = (E, V)$.

1. **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labelled or unlabelled.
2. **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/ unlabelled.

Graphs are used to solve many real-life problems. 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, locale etc.



Breadth First Search (BFS) in Graph:

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories: Visited and Non-Visited. A Boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

Algorithm:

```
1. Declare a queue and insert the starting vertex.
2. Initialize a visited array and mark the starting vertex as
   visited.
3. Follow the below process till the queue becomes empty:
    a. Remove the first vertex of the queue.
    b. Mark that vertex as visited.
    c. Insert all the unvisited neighbours of the vertex into the
       queue.
```

CPP Program for BFS:

```
using namespace std;

class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency lists
    vector<list<int>> adj;
public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj.resize(V);
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    vector<bool> visited;
    visited.resize(V,false);

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (auto adjecent: adj[s])
        {
            if (!visited[adjecent])
            {
                visited[adjecent] = true;
                queue.push_back(adjecent);
            }
        }
    }
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Output:

```
Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1
```

Questions on Graphs for students to Answer:

1. What is a graph data structure?
2. Why it is non-linear in nature?
3. What are various applications of this data structure apart from the ones mentioned above?
4. Differentiate between a tree and a graph.
5. Do the complexity analysis of BFS and DFS.

Experiment 10 [Hash Tables]

Study the necessity of Hash function and Hash table.

Suppose we want to design a system for storing employee records with phone numbers (as keys). And we want the following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For arrays and linked lists, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in $O(\log n)$ time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order. With balanced binary search tree, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in $O(\log n)$ time.

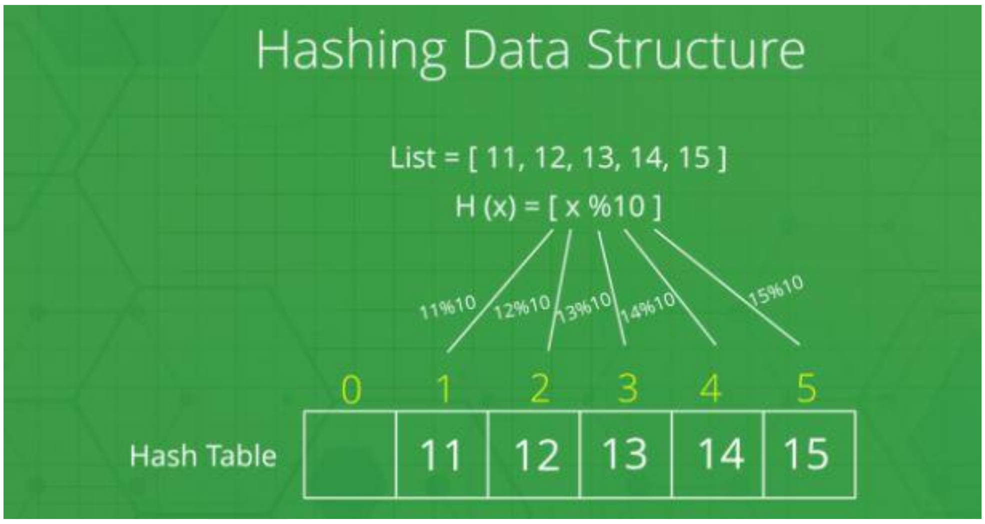
Another solution that one can think of is to use a direct access table where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in $O(1)$ time. For example, to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example, if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get $O(1)$ search time on average (under reasonable assumptions) and $O(n)$ in worst case. Now let us understand what hashing is.

Hashing: Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

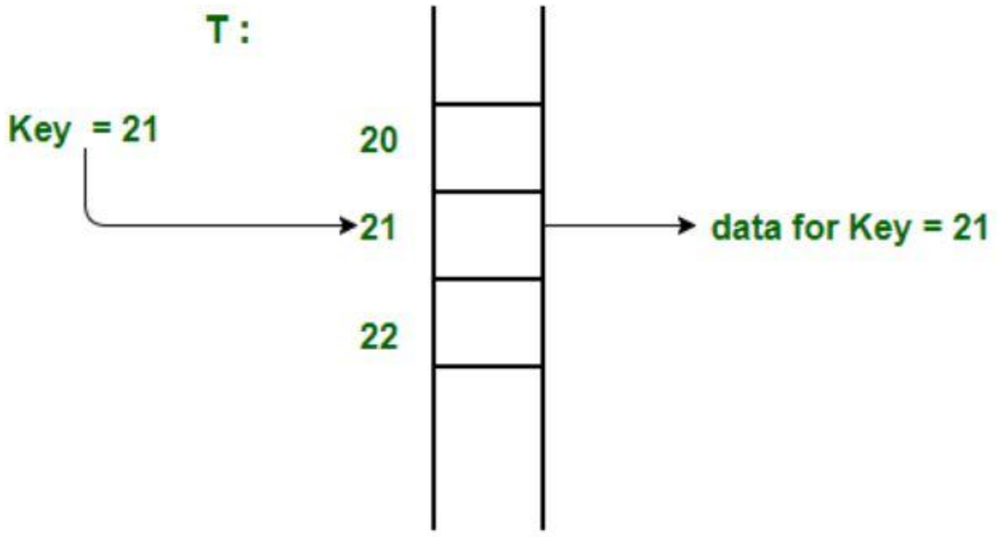
Let a hash function $H(x)$ maps the value x at the index $x\%10$ in an Array. For example, if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table, respectively.



Given a limited range array contains both positive and non-positive numbers, i.e., elements are in the range from -MAX to +MAX. Our task is to search if some number is present in the array or not in $O(1)$ time.

Since range is limited, we can use index mapping (or trivial hashing). We use values as the index in a big array. Therefore, we can search and insert elements in $O(1)$ time.

The idea is to use a 2D array of size $hash[MAX + 1][2]$



Algorithm:

```
Assign all the values of the hash matrix as 0.

Traverse the given array:

    If the element ele is non negative

assign hash[ele][0] as 1.

    Else take the absolute value of ele and

 assign hash[ele][1] as 1.
```

Implementation:

```
using namespace std;
#define MAX 1000
// Since array is global, it is initialized as 0.
bool has[MAX + 1][2];

// searching if X is Present in the given array or not.
bool search(int X)
{
    if (X >= 0) {
        if (has[X][0] == 1)
            return true;
        else
            return false;
    }

    // if X is negative take the absolute
    // value of X.
    X = abs(X);
    if (has[X][1] == 1)
        return true;

    return false;
}

void insert(int a[], int n)
{
    for (int i = 0; i < n; i++) {
        if (a[i] >= 0)
            has[a[i]][0] = 1;
       else
            has[abs(a[i])][1] = 1;
    }
}
```

```
int main()
{
    int a[] = { -1, 9, -5, -8, -5, -2 };
    int n = sizeof(a)/sizeof(a[0]);
    insert(a, n);
    int X = -5;
    if (search(X) == true)
       cout << "Present";
    else
       cout << "Not Present";
    return 0;
}
```

Output: Present

Questions on Hashing for students to Answer:

1. What is a hash function?
2. What is a hash table?
3. How use of hash function and table eases the task of searching?
4. Give examples where hashing is used?
5. What are the properties of a good hash function?

End of Lab Manual