Mahatma Gandhi Mission's

College of Engineering and Technology Kamothe Navimumbai

Department of Computer Engineering

Practical List

Subject: Data Structure Lab **Semester:** III

Sr. No	Name of the Experiment	Page No.
1	Implement Stack ADT using array.	
2	Convert an Infix expression to Postfix expression using stack ADT.	
3	Evaluate Postfix Expression using Stack ADT.	
4	Applications of Stack ADT.	
5	Implement Linear Queue ADT using array.	
6	Implement Circular Queue ADT using array.	
7	Implement Priority Queue ADT using array.	
8	Implement Singly Linked List ADT.	
9	Implement Circular Linked List ADT.	
10	Implement Binary Search Tree ADT using Linked List	
11	Implement Graph Traversal techniques: a) Breadth First Search b) Depth First Search	

EXPERIMENT NO: 1

Name of the Student:-_____

Roll No._____ Subject:-_____

Date of Practical Performed:-_____Staff Signature with Date

Marks

TITLE: Implement Stack ADT using array.

AIM: this module, we will be learn about:

- Gain the concept of stacks
- Understand the basic operations of stacks
- Practice the operations of stacks
- Test your conceptual understanding with a short quiz

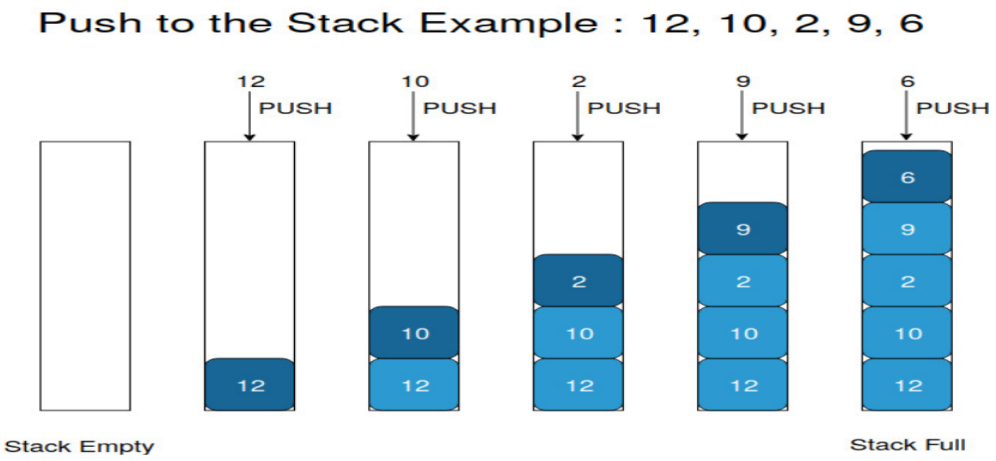
Software used: C and C++ Pogramming

- **Theory: I**magine a pile of books, with books stacked one over the other. From this pile of books, you can either put another book on top or remove a book from the top.
- The book which is at the bottom of the pile is the last one to be taken out, while the books at the top are removed first. Books can only be added to the top of the pile.
- Let the action of putting a book on the top be called as push and let the action of removing a book be called pop. A type of structure, similar to the example of the pile of books, can be represented as a data structure. Such a data structure is known as a stack.

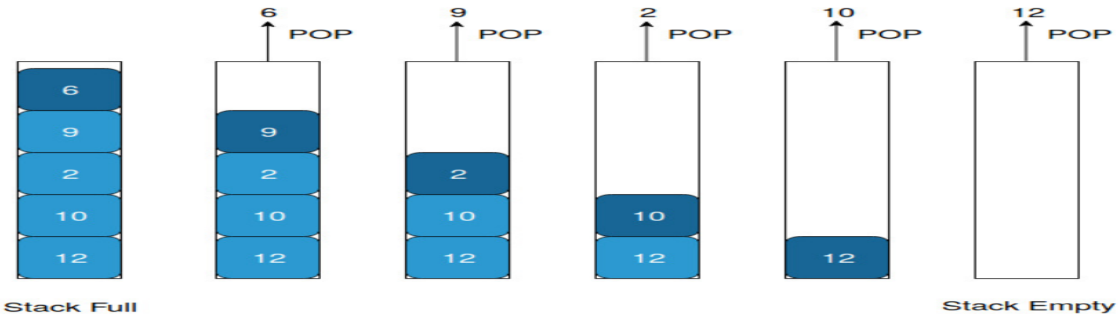
Stack Operations and Applications

Just like how we saw in the example of a stack of books, a stack data structure has two types of operations : push and pop. As we can see, a stack is an example of a last in, first out data structure (LIFO). That is, an element that is pushed last into a stack is the first to be popped out. Stacks have many applications. Lets explore a few of them. Reversing a word : Think about how you would reverse a word using a stack. First all the letters are pushed into the stack and then popped out one by one to get the reversed word. This would take linear time O(n). Undoing Changes in a Text Editor : A stack is also commonly used in text editors. Changes that the user makes are pushed into a stack. While undoing, they are popped out

Pushing into a Stack



Popping from a Stack



Program for Stack Data Structure using ARRAYS

```
/* C++ program to implement basic stack
operations */

#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack {
	int top;

public:
	int a[MAX]; // Maximum size of Stack

	Stack() { top = -1; }
	bool push(int x);
	int pop();
	int peek();
	bool isEmpty();
};

bool Stack::push(int x)
{
	if (top >= (MAX - 1)) {
		cout << "Stack Overflow";
		return false;
	}
	else {
		a[++top] = x;
		cout << x << " pushed into stack\n";
		return true;
	}
}

int Stack::pop()
{
	if (top < 0) {
		cout << "Stack Underflow";
		return 0;
	}
	else {
		int x = a[top--];
		return x;
	}
}
int Stack::peek()
{
	if (top < 0) {
```

```
                cout << "Stack is Empty";
                return 0;
        }
        else {
                int x = a[top];
                return x;
        }
}

bool Stack::isEmpty()
{
        return (top < 0);
}

// Driver program to test above functions
int main()
{
        class Stack s;
        s.push(10);
        s.push(20);
        s.push(30);
        cout << s.pop() << " Popped from stack\n";
        //print all elements in stack :
        cout<<"Elements present in stack : ";
        while(!s.isEmpty())
        {
                // print top element in stack
                cout<<s.peek()<<" ";
                // remove top element in stack
                s.pop();
        }

        return 0;
}
```

Conclusion: _____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Lab Assignments:

Q1. Describe Operations in stacks

Q2. Explain applications of stack

EXPERIMENT NO: 2

Name of the Student:-_____

Roll No._____ Subject:-_____

Date of Practical Performed:-_____Staff Signature with Date

Marks

TITLE: Convert an Infix expression to Postfix expression using stack ADT.

AIM: Implement Infix expression to Postfix expression using stack ADT

Theory:

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.
Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.
Why postfix representation of the expression?
The compiler scans the expression either from left to right or from right to left.
Consider the below expression: a op1 b op2 c op3 d
If op1 = +, op2 = *, op3 = +
The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it. The result is then added to d after another scan.
The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
The corresponding expression in postfix form is abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

```
/* C++ implementation to convert
infix expression to postfix*/

#include<bits/stdc++.h>
using namespace std;

//Function to return precedence of operators
int prec(char c) {
    if(c == '^')
            return 3;
    else if(c == '/' || c=='*')
            return 2;
    else if(c == '+' || c == '-')
            return 1;
    else
```

```
        return -1;
}

// The main function to convert infix expression
//to postfix expression
void infixToPostfix(string s) {

    stack<char> st; //For stack operations, we are using C++ built in stack
    string result;

    for(int i = 0; i < s.length(); i++) {
        char c = s[i];

        // If the scanned character is
        // an operand, add it to output string.
        if((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += c;

        // If the scanned character is an
        // ‘(‘, push it to the stack.
        else if(c == '(')
            st.push('(');

        // If the scanned character is an ‘)’,
        // pop and to output string from the stack
        // until an ‘(‘ is encountered.
        else if(c == ')') {
            while(st.top() != '(')
            {
                result += st.top();
                st.pop();
            }
            st.pop();
        }

        //If an operator is scanned
        else {
            while(!st.empty() && prec(s[i]) <= prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
```

```
            }
        }

        // Pop all the remaining elements from the stack
        while(!st.empty()) {
            result += st.top();
            st.pop();
        }

        cout << result << endl;
    }

    //Driver program to test above functions
    int main() {
        string exp = "a+b*(c^d-e)^(f+g*h)-i";
        infixToPostfix(exp);
        return 0;
    }
```

Output
abcd^e-fgh*+^*+i-

Conclusion:

Q1. Explain Recursion in stack?

Q2. What is a linear data structure? Give two examples of linear data structures.

-

EXPERIMENT NO:3

Name of the Student:-_____

Roll No._____ Subject:-

Date of Practical Performed:-_____Staff Signature with Date

Marks

TITLE: Evaluate Postfix Expression using Stack ADT.

AIM: Implement and create and Evaluate Postfix Expression using Stack ADT.

.

Theory The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed infix to postfix conversion. In this experiment evaluation of postfix expressions is discussed.

Following is an algorithm for evaluation postfix expressions.

1) Create a stack to store operands (or values).
2) Scan the given expression and do the following for every scanned element.
    a) If the element is a number, push it into the stack
    b) If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack
3) When the expression is ended, the number in the stack is the final answer

Example:

Let the given expression be "2 3 1 * + 9 -". We scan all elements one by one.

1) Scan '2', it's a number, so push it to stack. Stack contains '2'
2) Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)
3) Scan '1', again a number, push it to stack, stack now contains '2 3 1'
4) Scan '*', it's an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result '3' to stack. The stack now becomes '2 3'.
5) Scan '+', it's an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. The stack now becomes '5'.
6) Scan '9', it's a number, we push it to the stack. The stack now becomes '5 9'.
7) Scan '-', it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result '-4' to the stack. The stack now becomes '-4'.
8) There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).

Code of experiment:

```
// C++ program to evaluate value of a postfix expression

#include <iostream>
#include <string.h>

using namespace std;

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

// Stack Operations
struct Stack* createStack ( unsigned capacity )
{

struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

	if (!stack) return NULL;

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

	if (!stack->array) return NULL;

	return stack;
}

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

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

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

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


// The main function that returns value of a given postfix expression

int evaluatePostfix(char* exp)
{
        // Create a stack of capacity equal to expression size

struct Stack* stack = createStack( strlen(exp));
        int i;

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

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

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

// Driver program to test above functions
int main()
{
        char exp[] = "231*+9-";
        cout<<"postfix evaluation: "<< evaluatePostfix(exp);
        return 0;}
```

OUTPUT:

postfix evaluation: -4

The time complexity of the evaluation algorithm is O(n) where n is a number of characters in the input expression.

There are the following limitations of the above implementation.
1) It supports only 4 binary operators '+', '*', '-' and '/'. It can be extended for more operators by adding more switch cases.
2) The allowed operands are only single-digit operands. The program can be extended for multiple digits by adding a separator-like space between all elements (operators and operands) of the given expression.

Conclusion:

Q1. Evaluate the following expression in postfix :623+-382/+*2^3+

Q2.Explain algorithm for postfix expression using stack

EXPERIMENT NO:4

Name of the Student:-_____

Roll No._____ Subject:-

Date of Practical Performed:-_____Staff Signature with Date

Marks

TITLE: Applications of Stack ADT.

AIM: Implement Iterative Tower of Hanoi

Theory : The Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape. The objective of the puzzle is to move all the disks from one pole (say 'source pole') to another pole (say 'destination pole') with the help of the third pole (say auxiliary pole).

The puzzle has the following two rules:

1. You can't place a larger disk onto a smaller disk
2. Only one disk can be moved at a time

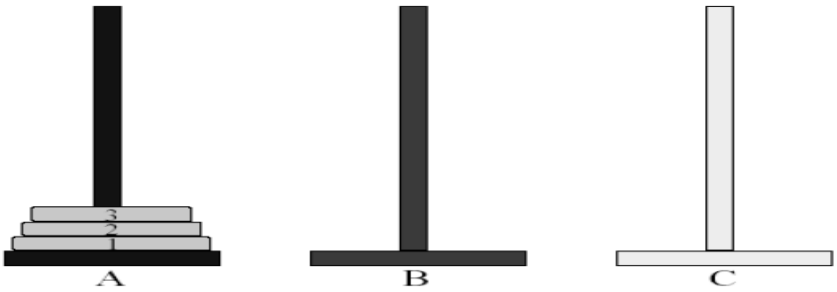
Iterative Algorithm:

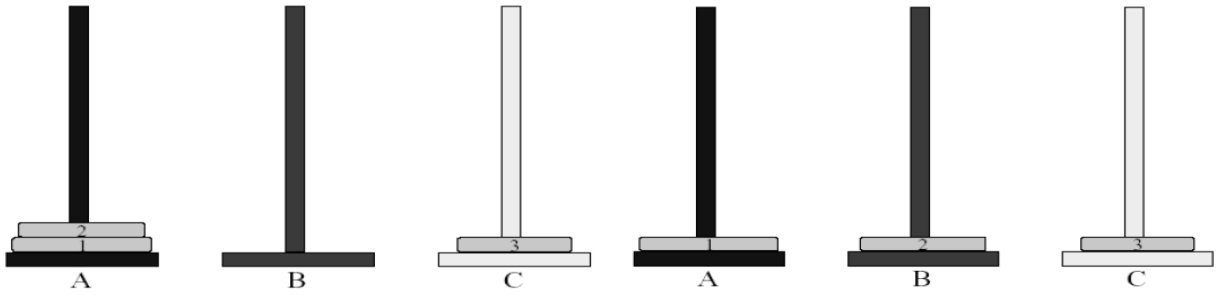
1. Calculate the total number of moves required i.e. "pow(2, n)
   - 1" here n is number of disks.
2. If number of disks (i.e. n) is even then interchange destination
   pole and auxiliary pole.
3. for i = 1 to total number of moves:
   if i%3 == 1:
   legal movement of top disk between source pole and
   destination pole
   if i%3 == 2:
   legal movement top disk between source pole and
   auxiliary pole
   if i%3 == 0:
   legal movement top disk between auxiliary pole
   and destination pole

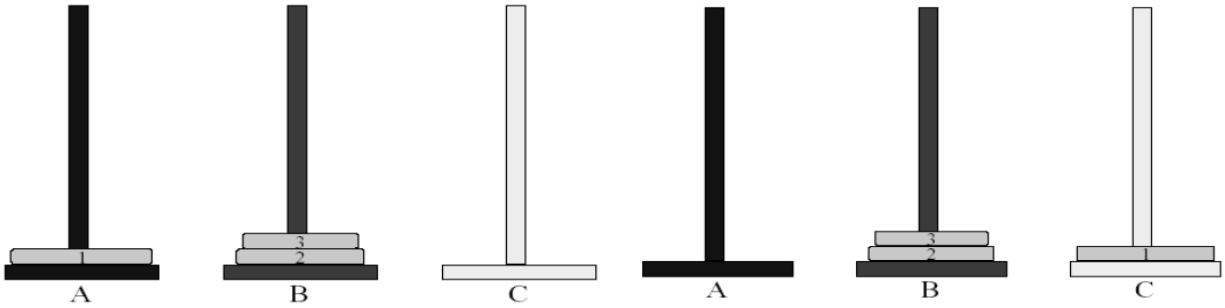
Example:

Let us understand with a simple example with 3 disks:

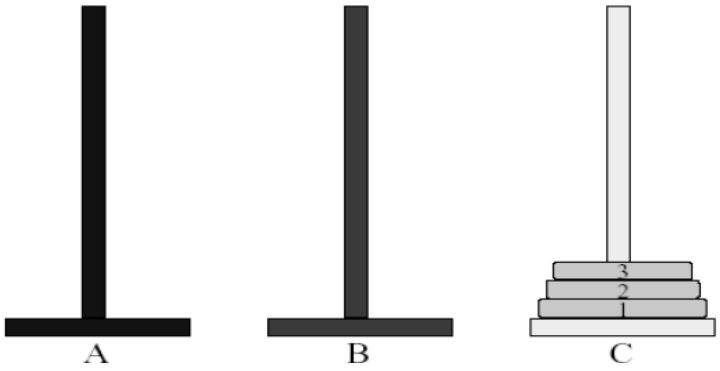
So, total number of moves required = 7







.



So, after all these destination poles contains all the in order of size.
After observing above iterations, we can think that after a disk other than the smallest disk is moved, the next disk to be moved must be the smallest disk because it is the top disk resting on the spare pole and there are no other choices to move a disk.
Above figure Source Pole =A=S; Auxillary pole= B=A; Destination pole= C= D;
Program code:

```
// C++ Program for Iterative Tower of Hanoi
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

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

// function to create a stack of given capacity.
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;
}

// 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 show the movement of disks
void moveDisk(char fromPeg, char toPeg, int disk)
{
	cout <<"Move the disk " << disk <<" from " << fromPeg << " to "<< toPeg << endl;
}

// Function to implement legal movement between
// two poles
void moveDisksBetweenTwoPoles(struct Stack *src,
			struct Stack *dest, char s, char d)
{
	int pole1TopDisk = pop(src);
	int pole2TopDisk = pop(dest);

	// When pole 1 is empty
	if (pole1TopDisk == INT_MIN)
	{
		push(src, pole2TopDisk);
		moveDisk(d, s, pole2TopDisk);
	}

	// When pole2 pole is empty
	else if (pole2TopDisk == INT_MIN)
	{
		push(dest, pole1TopDisk);
		moveDisk(s, d, pole1TopDisk);
	}

	// When top disk of pole1 > top disk of pole2
	else if (pole1TopDisk > pole2TopDisk)
	{
		push(src, pole1TopDisk);
		push(src, pole2TopDisk);
```

```
            moveDisk(d, s, pole2TopDisk);
        }

        // When top disk of pole1 < top disk of pole2
        else
        {
            push(dest, pole2TopDisk);
            push(dest, pole1TopDisk);
            moveDisk(s, d, pole1TopDisk);
        }
}

//Function to implement TOH puzzle
void tohIterative(int num_of_disks, struct Stack *src, struct Stack *aux, struct Stack *dest)
{
    int i, total_num_of_moves;
    char s = 'S', d = 'D', a = 'A';

    //If number of disks is even, then interchange
    //destination pole and auxiliary pole
    if (num_of_disks % 2 == 0)
    {
        char temp = d;
        d = a;
        a = temp;
    }
    total_num_of_moves = pow(2, num_of_disks) - 1;

    //Larger disks will be pushed first
    for (i = num_of_disks; i >= 1; i--)
        push(src, i);

    for (i = 1; i <= total_num_of_moves; i++)
    {
        if (i % 3 == 1)
        moveDisksBetweenTwoPoles(src, dest, s, d);

        else if (i % 3 == 2)
        moveDisksBetweenTwoPoles(src, aux, s, a);

        else if (i % 3 == 0)
        moveDisksBetweenTwoPoles(aux, dest, a, d);
    }
}

// Driver Program
int main()
{

    // Input: number of disks
    unsigned num_of_disks = 3;

    struct Stack *src, *dest, *aux;
```

```
    // Create three stacks of size 'num_of_disks'
    // to hold the disks
    src = createStack(num_of_disks);
    aux = createStack(num_of_disks);
    dest = createStack(num_of_disks);

    tohIterative(num_of_disks, src, aux, dest);
    return 0;
}
```

Output:

Move the disk 1 from 'S' to 'D'

Move the disk 2 from 'S' to 'A'

Move the disk 1 from 'D' to 'A'

Move the disk 3 from 'S' to 'D'

Move the disk 1 from 'A' to 'S'

Move the disk 2 from 'A' to 'D'

Move the disk 1 from 'S' to 'D'

CONCLUSION:

Q1.Match the left and right parentheses in a character string

(a*(b+c)+d)

-

Q2. Parenthesis Matching, How do we implement this using a stack?

EXPERIMENT NO:5

Name of the Student:-_____

Roll No._____ Subject:-_____

Date of Practical Performed:-_____Staff Signature with Date

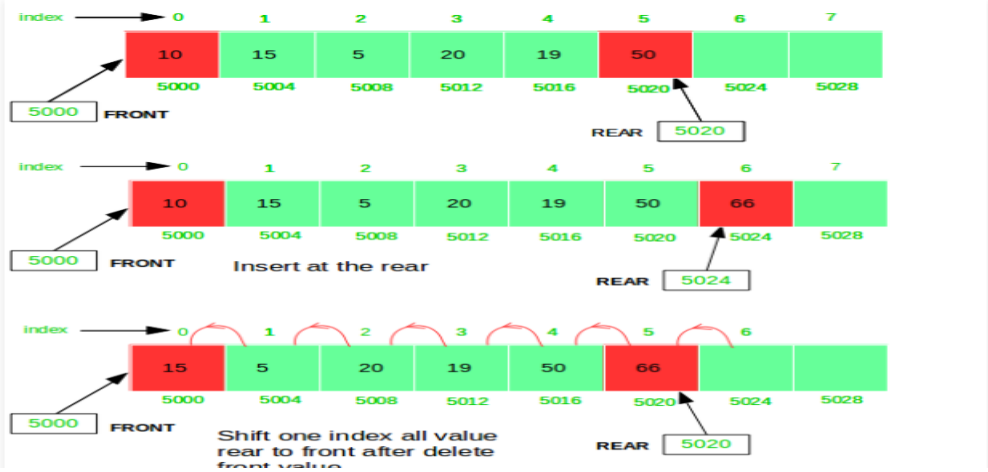
Marks

TITLE: Implement Linear Queue ADT using array.

Aim : To Implement Linear Queue ADT using array

Theory: In queue, insertion and deletion happen at the opposite ends, so implementation is not as simple as stack.
To implement a queue using array, create an array *arr* of size *n* and take two variables *front* and *rear* both of which will be initialized to *0* which means the queue is currently empty. Element *rear* is the index upto which the elements are stored in the array and *front* is the index of the first element of the array. Now, some of the implementation of queue operations are as follows:

1. **Enqueue:** Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If *rear* < *n* which indicates that the array is not full then store the element at *arr[rear]* and increment *rear* by *1* but if *rear* == *n* then it is said to be an Overflow condition as the array is full.
2. **Dequeue:** Removal of an element from the queue. An element can only be deleted when there is at least an element to delete i.e. *rear* > *0*. Now, element at *arr[front]* can be deleted but all the remaining elements have to shifted to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.
3. **Front:** Get the front element from the queue i.e. *arr[front]* if queue is not empty.
4. **Display:** Print all element of the queue. If the queue is non-empty, traverse and print all the elements from index *front* to *rear*.



Program Code:

```
// C++ program to implement a queue using an array
#include <bits/stdc++.h>
using namespace std;

struct Queue {
	int front, rear, capacity;
	int* queue;
	Queue(int c)
	{
		front = rear = 0;
		capacity = c;
		queue = new int;
	}

	~Queue() { delete[] queue; }

	// function to insert an element
	// at the rear of the queue
	void queueEnqueue(int data)
	{
		// check queue is full or not
		if (capacity == rear) {
			printf("\nQueue is full\n");
			return;
		}

		// insert element at the rear
		else {
			queue[rear] = data;
			rear++;
		}
		return;
	}

	// function to delete an element
	// from the front of the queue
	void queueDequeue()
	{
		// if queue is empty
		if (front == rear) {
			printf("\nQueue is empty\n");
			return;
		}

		// shift all the elements from index 2 till rear
		// to the left by one
		else {
			for (int i = 0; i < rear - 1; i++) {
				queue[i] = queue[i + 1];
```

```
                }

                // decrement rear
                rear--;
        }
        return;
    }

    // print queue elements
    void queueDisplay()
    {
        int i;
        if (front == rear) {
            printf("\nQueue is Empty\n");
            return;
        }

        // traverse front to rear and print elements
        for (i = front; i < rear; i++) {
            printf(" %d <-- ", queue[i]);
        }
        return;
    }

    // print front of queue
    void queueFront()
    {
        if (front == rear) {
            printf("\nQueue is Empty\n");
            return;
        }
        printf("\nFront Element is: %d", queue[front]);
        return;
    }
};

// Driver code
int main(void)
{
    // Create a queue of capacity 4
    Queue q(4);

    // print Queue elements
    q.queueDisplay();

    // inserting elements in the queue
    q.queueEnqueue(20);
    q.queueEnqueue(30);
    q.queueEnqueue(40);
    q.queueEnqueue(50);

    // print Queue elements
    q.queueDisplay();
```

```
        // insert element in the queue
        q.queueEnqueue(60);

        // print Queue elements
        q.queueDisplay();

        q.queueDequeue();
        q.queueDequeue();

        printf("\n\nafter two node deletion\n\n");

        // print Queue elements
        q.queueDisplay();

        // print front of the queue
        q.queueFront();

        return 0;
}
```

Output:

Queue is Empty

20 <-- 30 <-- 40 <-- 50 <--

Queue is full

20 <-- 30 <-- 40 <-- 50 <--

after two node deletion

40 <-- 50 <--

Front Element is: 40

Conclusion: _____

Q1.Explain operations used in Queue as a Data Structure.

Q2.Explain the types of Queue as Data structure

EXPERIMENT NO:6

Name of the Student:-_____

Roll No._____ Subject:-_____

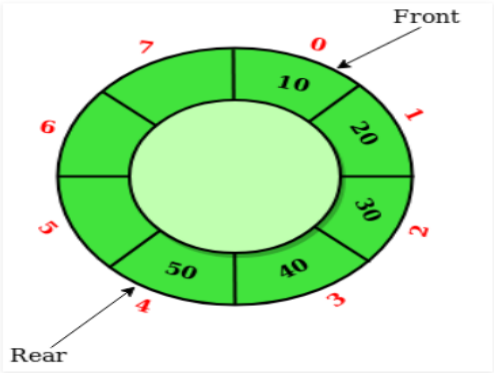
Date of Practical Performed:-_____Staff Signature with Date

Marks

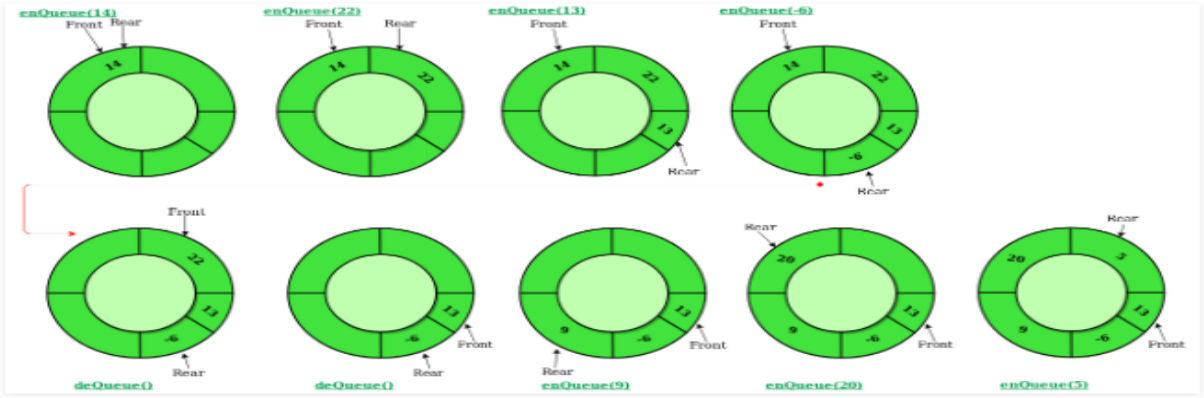
TITLE: Implement Circular Queue ADT using array.

Aim : To implement Circular Queue ADT using array.

Theory: Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.



Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
  1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
  1. Check whether queue is Empty means check (front==-1).
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Time Complexity: Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

Applications:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Program Code

```
// C or C++ program for insertion and
// deletion in Circular Queue
#include<bits/stdc++.h>
using namespace std;

class Queue
{
        // Initialize front and rear
        int rear, front;
```

```
        // Circular Queue
        int size;
        int *arr;

        Queue(int s)
        {
        front = rear = -1;
        size = s;
        arr = new int[s];
        }

        void enQueue(int value);
        int deQueue();
        void displayQueue();
};


/* Function to create Circular queue */
void Queue::enQueue(int value)
{
        if ((front == 0 && rear == size-1) ||
                        (rear == (front-1)%(size-1)))
        {
                printf("\nQueue is Full");
                return;
        }

        else if (front == -1) /* Insert First Element */
        {
                front = rear = 0;
                arr[rear] = value;
        }

        else if (rear == size-1 && front != 0)
        {
                rear = 0;
                arr[rear] = value;
        }

        else
        {
                rear++;
                arr[rear] = value;
        }
}

// Function to delete element from Circular Queue
int Queue::deQueue()
{
        if (front == -1)
        {
                printf("\nQueue is Empty");
```

```
		return INT_MIN;
	}

	int data = arr[front];
	arr[front] = -1;
	if (front == rear)
	{
		front = -1;
		rear = -1;
	}
	else if (front == size-1)
		front = 0;
	else
		front++;

	return data;
}

// Function displaying the elements
// of Circular Queue
void Queue::displayQueue()
{
	if (front == -1)
	{
		printf("\nQueue is Empty");
		return;
	}
	printf("\nElements in Circular Queue are: ");
	if (rear >= front)
	{
		for (int i = front; i <= rear; i++)
			printf("%d ",arr[i]);
	}
	else
	{
		for (int i = front; i < size; i++)
			printf("%d ", arr[i]);

		for (int i = 0; i <= rear; i++)
			printf("%d ", arr[i]);
	}
}

/* Driver of the program */
int main()
{
	Queue q(5);

	// Inserting elements in Circular Queue
	q.enQueue(14);
	q.enQueue(22);
	q.enQueue(13);
	q.enQueue(-6);
```

```
        // Display elements present in Circular Queue
        q.displayQueue();

        // Deleting elements from Circular Queue
        printf("\nDeleted value = %d", q.deQueue());
        printf("\nDeleted value = %d", q.deQueue());

        q.displayQueue();

        q.enQueue(9);
        q.enQueue(20);
        q.enQueue(5);

        q.displayQueue();

        q.enQueue(20);
        return 0;
}
```

Output:

Elements in Circular Queue are: 14 22 13 -6

Deleted value = 14

Deleted value = 22

Elements in Circular Queue are: 13 -6

Elements in Circular Queue are: 13 -6 9 20 5

Queue is Full

Conclusion:

-

Q1.Explain Operations in Circular Queue.

Q2.Explain applications of circular queue.

EXPERIMENT NO: 7

Name of the Student:-_____

Roll No._____ Subject:-_____

Date of Practical Performed:-_____Staff Signature with Date

Marks

TITLE: Implement Priority Queue ADT using array.

AIM: To Implement Priority Queue ADT using array

Theory : Priority Queue is an extension of the Queue data structure where each element has a particular priority associated with it. It is based on the priority value, the elements from the queue are deleted.

- **enqueue():** This function is used to insert new data into the queue.
- **dequeue():** This function removes the element with the highest priority from the queue.
- **peek()/top():** This function is used to get the highest priority element in the queue without removing it from the queue.

Approach: The idea is to create a structure to store the value and priority of the element and then create an array of that structure to store elements. Below are the functionalities that are to be implemented:

- **enqueue():** It is used to insert the element at the end of the queue.
- **peek():**
    - Traverse across the priority queue and find the element with the highest priority and return its index.
    - In the case of multiple elements with the same priority, find the element with the highest value having the highest priority.
- **dequeue():**
    - Find the index with the highest priority using the **peek()** function let's call that position as **ind**, and then shift the position of all the elements after the position **ind** one position to the left.
    - Decrease the size by one.

Application of Priority Queue:

- For Scheduling Algorithms the CPU has to process certain tasks having priorities. The process of having higher priority gets executed first.
- In a time-sharing computer system, the process of waiting for the CPU time gets loaded in the priority queue.
- A Sorting-priority queue is used to sort heaps.

Program Code:

```
// C++ program for the above approach

#include <bits/stdc++.h>
using namespace std;

// Structure for the elements in the
// priority queue
struct item {
		int value;
		int priority;
};

// Store the element of a priority queue
item pr[100000];

// Pointer to the last index
int size = -1;

// Function to insert a new element
// into priority queue
void enqueue(int value, int priority)
{
		// Increase the size
		size++;

		// Insert the element
		pr[size].value = value;
		pr[size].priority = priority;
}

// Function to check the top element
int peek()
{
		int highestPriority = INT_MIN;
		int ind = -1;

		// Check for the element with
		// highest priority
		for (int i = 0; i <= size; i++) {

			// If priority is same choose
			// the element with the
			// highest value
			if (highestPriority== pr[i].priority&& ind > -1&& pr[ind].val < pr[i].value)
{
				highestPriority = pr[i].priority;
				ind = i;
			}
			else if (highestPriority < pr[i].priority)
{
```

```
                        highestPriority = pr[i].priority;
                        ind = i;
                }
        }

        // Return position of the element
        return ind;
}

// Function to remove the element with
// the highest priority
void dequeue()
{
        // Find the position of the element
        // with highest priority
        int ind = peek();

        // Shift the element one index before
        // from the position of the element
        // with highest priortity is found
        for (int i = ind; i < size; i++) {
                pr[i] = pr[i + 1];
        }

        // Decrease the size of the
        // priority queue by one
        size--;
}

// Driver Code
int main()
{
        // Function Call to insert elements
        // as per the priority
        enqueue(10, 2);
        enqueue(14, 4);
        enqueue(16, 4);
        enqueue(12, 3);

        // Stores the top element
        // at the moment
        int ind = peek();

        cout << pr[ind].value << endl;

        // Dequeue the top element
        dequeue();

        // Check the top element
        ind = peek();
        cout << pr[ind].value << endl;

        // Dequeue the top element
        dequeue();
```

```
        // Check the top element
        ind = peek();
        cout << pr[ind].value << endl;

        return 0;
}
```

Output:

16

12

Conclusion:

-

Q1.Explain operations of Priority Queue

Q2.Explain Applications of Priority Queue.

.

EXPERIMENT NO:8

Name of the Student:-_____

Roll No._____ Subject:-_____

Date of Practical Performed:-_____Staff Signature with Date

Marks

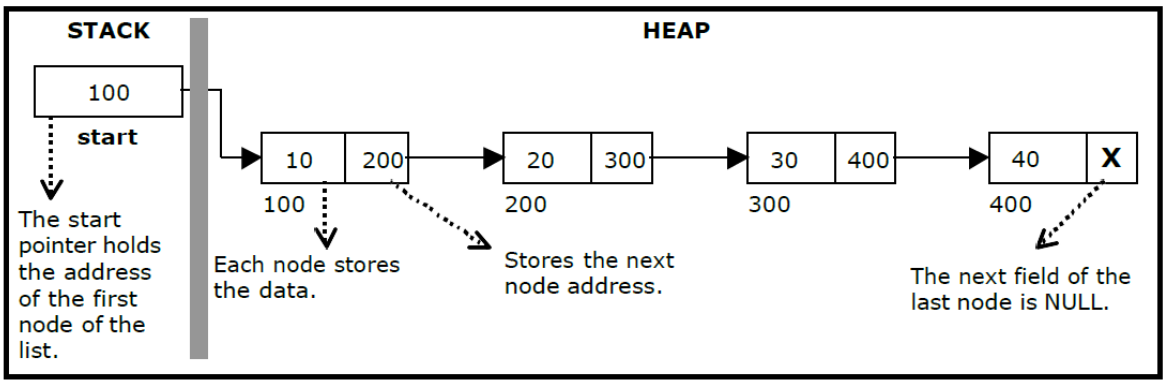
TITILE: Implement Singly Linked List ADT.

AIM: To Implement Singly Linked List ADT.

THEORY: A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the —start node



Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see figure):

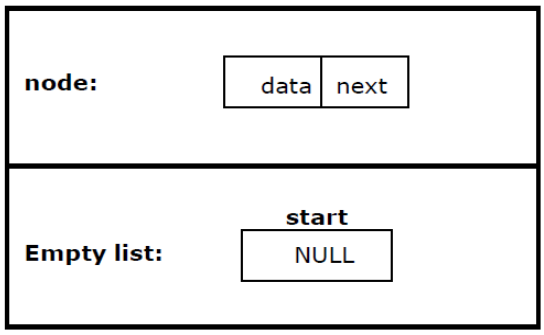
- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.

- Initialise the start pointer to be NULL.

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

typedef struct slinklist node;

node *start = NULL;
```



The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Program Code:

```
// A simple C++ program for traversal of a linked list
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;
};

// This function prints contents of linked list
// starting from the given node
void printList(Node* n)
{
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}

// Driver code
int main()
{
    Node* head = NULL;
```

```
    Node* second = NULL;
    Node* third = NULL;

    // allocate 3 nodes in the heap
    head = new Node();
    second = new Node();
    third = new Node();

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

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

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

    printList(head);

    return 0;
}
```

Output:

```
1  2  3
```

Conclusion:

Q1.Explain the types linked list

Q2.Explain the application of Linked list.

EXPERIMENT NO:9

Name of the Student:-_____

Roll No._____ Subject:-_____

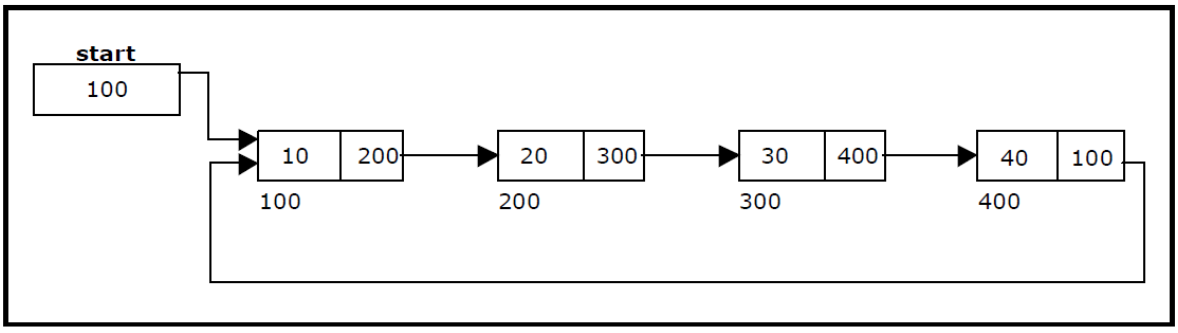
Date of Practical Performed:-_____Staff Signature with Date

Marks

TITLE: Implement Circular Linked List ADT.

Aim : To implement Implement Circular Linked List ADT.

Theory: It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.



The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Progarm code:

```
#include <iostream>
using namespace std;
struct Node {
  int data;
  struct Node *next;
};
struct Node* head = NULL;
void insert(int newdata) {
  struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *ptr = head;
  newnode->data = newdata;
  newnode->next = head;
  if (head!= NULL) {
    while (ptr->next != head)
    ptr = ptr->next;
    ptr->next = newnode;
  } else
  newnode->next = newnode;
  head = newnode;
}
void display() {
  struct Node* ptr;
  ptr = head;
  do {
    cout<<ptr->data <<" ";
    ptr = ptr->next;
  } while(ptr != head);
}
int main() {
  insert(3);
  insert(1);
  insert(7);
  insert(2);
  insert(9);
  cout<<"The circular linked list is: ";
  display();
```

```
  return 0;
}
```

Output

The circular linked list is: 9 2 7 1 3

Conclusion:

Q1.Write ADT for inserting a node at the beginning using circular linked list.

Q2.Write ADT for inserting node at the middle in circular linked list

---

EXPERIMENT NO:10

Name of the Student:-_____

Roll No._____ Subject:-_____

Date of Practical Performed:-_____Staff Signature with Date

Marks

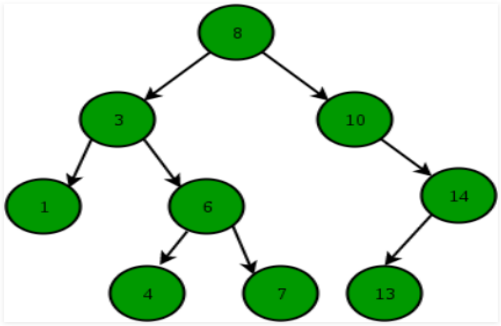
TITLE: Implement Binary Search Tree ADT using Linked List

AIM: To Implement Binary Search Tree ADT using Linked List.

Theory:

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

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes.



The above properties of Binary Search Tree provides an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search for a given key.

Illustration to insert 2 in below tree:

1. Start from the root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. After reaching the end, just insert that node at left(if less than current) else right.

Time Complexity: The worst-case time complexity of search and insert operations is O(h) where h is the height of the Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Program code:

```
// C++ program to demonstrate insertion
// in a BST recursively.
#include <iostream>
using namespace std;

class BST
{
	int data;
	BST *left, *right;

public:
	// Default constructor.
	BST();

	// Parameterized constructor.
	BST(int);

	// Insert function.
	BST* Insert(BST*, int);

	// Inorder traversal.
	void Inorder(BST*);
};

// Default Constructor definition.
BST ::BST()
	: data(0)
	, left(NULL)
	, right(NULL)
{
}

// Parameterized Constructor definition.
BST ::BST(int value)
{
	data = value;
	left = right = NULL;
}

// Insert function definition.
BST* BST ::Insert(BST* root, int value)
{
	if (!root)
	{
		// Insert the first node, if root is NULL.
		return new BST(value);
	}

	// Insert data.
	if (value > root->data)
	{
		// Insert right node data, if the 'value'
		// to be inserted is greater than 'root' node data.

		// Process right nodes.
		root->right = Insert(root->right, value);
	}
	else
	{
		// Insert left node data, if the 'value'
		// to be inserted is greater than 'root' node data.
```

```
                // Process left nodes.
                root->left = Insert(root->left, value);
        }

        // Return 'root' node, after insertion.
        return root;
}

// Inorder traversal function.
// This gives data in sorted order.
void BST ::Inorder(BST* root)
{
        if (!root) {
                return;
        }
        Inorder(root->left);
        cout << root->data << endl;
        Inorder(root->right);
}

// Driver code
int main()
{
        BST b, *root = NULL;
        root = b.Insert(root, 50);
        b.Insert(root, 30);
        b.Insert(root, 20);
        b.Insert(root, 40);
        b.Insert(root, 70);
        b.Insert(root, 60);
        b.Insert(root, 80);

        b.Inorder(root);
        return 0;
}
```

Output

```
20

30

40

50

60

70

80
```

Conclusion:

Q1.Explain the properties of binary search tree.

Q2.Explain how binary tree can be represented

EXPERIMENT NO:11

Name of the Student:-_____

Roll No._____ Subject:-_____

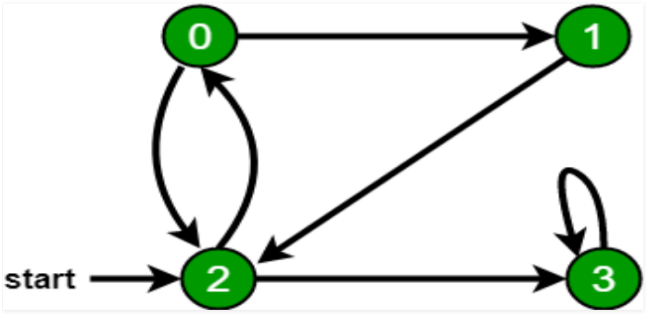
Date of Practical Performed:-_____Staff Signature with Date

Marks

TITLE: Implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search

AIM: To implement Graph Traversal techniques: a) Depth First Search b) Breadth First Search.

Theory: Breadth First Search for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.



Following are the implementations of simple Breadth-First Traversal from a given source.
The implementation uses an adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes and the queue of nodes needed for BFS traversal.

Theory: Depth First Traversal for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

- **Approach:** Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.
- **Algorithm:**
  1. Create a recursive function that takes the index of the node and a visited array.
  2. Mark the current node as visited and print the node.
  3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

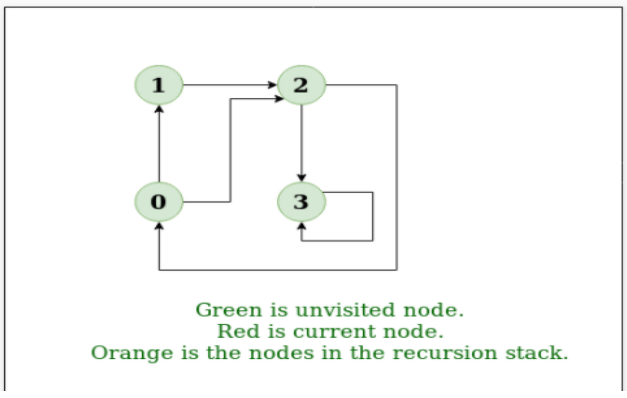
Input: *n = 4, e = 6*

0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3

Output: *DFS from vertex 1 : 1 2 0 3*

Explanation:

DFS Diagram:



Program code for BFS:

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
	int V; // No. of vertices

	// Pointer to an array containing adjacency
	// lists
	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 = new list<int>[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
	bool *visited = new bool[V];
	for(int i = 0; i < V; i++)
		visited[i] = 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);

	// 'i' will be used to get all adjacent
	// vertices of a vertex
	list<int>::iterator i;

	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 (i = adj[s].begin(); i != adj[s].end(); ++i)
                {
                        if (!visited[*i])
                        {
                                visited[*i] = true;
                                queue.push_back(*i);
                        }
                }
        }
}

// Driver program to test methods of graph class
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

Program code for BFS:

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include <bits/stdc++.h>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
public:
        map<int, bool> visited;
        map<int, list<int>> adj;

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

```
        // DFS traversal of the vertices
        // reachable from v
        void DFS(int v);
};

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

void Graph::DFS(int v)
{
        // Mark the current node as visited and
        // print it
        visited[v] = true;
        cout << v << " ";

        // Recur for all the vertices adjacent
        // to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (!visited[*i])
                        DFS(*i);
}

// Driver code
int main()
{
        // Create a graph given in the above diagram
        Graph g;
        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 Depth First Traversal"
                        " (starting from vertex 2) \n";
        g.DFS(2);

        return 0;
}
```

Output:

Following is Depth First Traversal (starting from vertex 2)

2 0 1 3

Conclusion:

Q1.Explain Properties of Breadth first search.

Q2.Explain Properties of Depth First Search.

-