INDEX

1BM22CS229

NAME: S. Rakhal STD.: III SEC.: C D ROLL NO.: SUB.: DS using C

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	21/12/2023	Lab 1 - Stack.		
2	28/12/2023	Lab 2 - Infix to postfix		
3	11/1/24	Lab 3 [→ linear queue → circular queue		
4	11/1/24	Lab 4 → Singly Linked List		
5	18/1/24	Lab 5 → Singly LL (deletion)		
→	18/1/24	Leetcode 1) minstack		
6	25/1/24	LAB 6 → L.L operations		
→	1/2/24	Leetcode 2) Reversal of part		
7	1/2/24	Lab 7) → Doubly Linked List		
→	1/2/24	Leetcode 3) Split LL in parts		
8	15/2/24	Lab 8) → Binary search tree		
→	15/2/24	Leetcode 4) Rotate list		
9	22/2/24	Lab 9) → BFS & DFS		
10	29/2/24	Lab 10) → Hashing		
→	29/2/24	Hacker Rank → snowpro dy		

Signed
In time.
5/3/24.

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks

Classmate
Date 28/12/23
Page

Lab programs week 3

Q1) write a program to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (Plus), - (minus), * (multiply), / (divide) and ^ (power).

Input:

```
#define
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_SIZE 100

int isOperator (char ch){
        return (ch == + || ch == '-' || ch == '*' || ch =
}

int getprecedence (char ch) {
        if (ch == '+' || ch == '-' ){
        return 1;
        }
        else if ( ch == '*' || ch == '/' ){
        return 2;
        }
        else if (ch == '^' ){
        return 3;
        }
        else {
          return 0;
```

P.T.O

```
void infixtopostfix( char infix[], char postfix[])
{
 char stack [Max_SIZE];
 int top=-1;

 int i,j;
 for (i=0; j=0; infix[i] != '\0';++i){
      char ch = infix[i];

 if (isalnum(ch)){
     postfix[j++] = ch;
 } elseif ( ch == '(' ) {
     stack [++top] = ch
 } elseif( ch == ')'){
     while (top >=0 && stack[top] != '('){
         postfix [j++] = stack[top--];}

 }
  if ( top>=0 && stack[top] == '('){
                 top--;
            }
       elseif (is operator (ch)){
            while ( top>=0 && getprecede
       (stack [top]) >= get precidence(ch))
                 postfix [j++] = stack[to
            }
            stack [++top] = ch;
       }
  }
  while (top>=0) {
      postfix [j++] = stack [top--];
  }
 postfix[j] = '\0';
```

```
int main () {

char infix [MAX_SIZE];
char postfix [MAX_SIZE];

printf ("Enter the infix expression: ");
scanf ("%s", infix);

infixtoPostfix (infix, postfix);

printf ("postfix expression: %s\n", postfix);

return 0;

}
```

OP:-

Enter infix Expression: a+b*(c^d-e)^(f+g*h)-i";

~~#Function call~~

~~infix to Postfix (exp);~~

Postfix Expression:

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

2) Evaluation of postfix :

```
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 100

int is operand(char ch){
    return isdigit(ch);
}

int top=-1;

void push(int item){
    if(top >= MAX_SIZE-1){
        printf("Stack overflow\n");
        exit(EXIT_FAILURE);
    }
    top++;
    stack[top] = item;
}

int pop(){
    if(top<0){
        printf("Stack underflow");
        return -1;
    }
    int item = stack[top];
    top--;
    return item;
}
int is_operator(char symbol){
    if(symbol=='+'||symbol=='-'||symbol=='*'||
    symbol=='/'){
        return 1;
    }
    return 0;
}
```

```
int evaluate(char expression[]){
    int i=0;
    char symbol = expression[i];
    int operand1, operand2, result;

    while(symbol != '\0'){
        if(symbol >= '0' && symbol <= '9'){
            int num = symbol - '0';
            push(num);
        }
        else if(is_operator(symbol)){
        op2 = pop();
        op1 = POP();
        switch(symbol){
            case '+': result = OP1 + OP2; break;
            case '-': result = OP1 - OP2; break;
            case '*': result = OP1 * OP2; break;
            case '/': result = OP1 / OP2; break;
        }
        push(result);
        }

        i++;
        symbol = expression[i];
    }
    result = pop();
    return result;

}

int main(){
    char expression[] = "5 6 7 + * 1 -";
    int result = evaluate(expression);
printf("Result = %d \n", result);
return 0;
}
```

28/12/2023

OP

P. No. 57

3A 9) Queues Implementation

Input :-

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 5

void enqueue (int);
void dequeue ();
void display();

int items[SIZE], front = -1, rear = -1;

int main () {

    dequeue ();

    enqueue (1);
    enqueue (2);
    enqueue (3);
    enqueue (4);
    enqueue (5);
    enqueue (6);

    display ();

    dequeue ();

    display ();
    return 0;
```

```
void enqueue (int value){
    if (rear == SIZE-1)
        printf ("\n Queue is Full! ");
    else {
        if (front == -1)
            front = 0;
        rear++;
        items [rear] = value;
        printf ("\n Inserted -> %d", value);
    }
}

void dequeue (){
    if (front == -1)
        printf ("\n Queue is empty!");
    else {
        printf ("\n Deleted : %d", items[front]);
        front++;
        if (front > rear)
            front = rear = -1;
    }
}

// Function to print the queue
void display (){
    if (rear == -1)
        printf ("\n Queue is Empty!");
    else {
        int i;
        printf ("\n Queue elements are: \n");
        for (i = front; i <= rear; i++){
            printf ("%d ", items[i]);
        }
        printf ("\n");
}
```

OUTPUT:-

Week-3

8 Q) circular queue implementation.

Input:-

```
#include<stdio.h>
#define SIZE 5

int items[SIZE];
int front=-1, rear=-1;

int isfull(){
    if((front==rear+1)||front==0 && rear==
SIZE-1))
    return 1;
    return 0;
}

int isempty(){
    if(front==-1) return 1;
    return 0;
}

void enqueue(int element){
    if(isFull())
        printf("\n Queue is full!! \n");
    else {
        if(front==-1) front=0;
        rear=(rear+1)%SIZE;
        items[rear]=element;
        printf("\n Inserted -> %d", element);
    }
}
```

```
int dequeue(){
    int element;
    if(isEmpty()){
        printf("\n Queue is empty!!\n");
        return(-1);
    } else {
        element = items[front];
        if(front == rear){
            front = -1;
            rear = -1;
        }
        else {
            front = (front+1)%SIZE;
        }
        printf("\n Deleted element -> %d\n", element);
        return(element);
    }
}

void display(){
    int i;
    if(isEmpty())
        printf("\n Empty Queue\n");
    else {
        printf("\n Front -> %d", front);
        printf("\n Items -> ");
        for(i=front; i!=rear; i=(i+1)%SIZE){
            printf("%d", items[i]);
        }
        printf("%d", items[i]);
        printf("\nRear -> %d\n", rear);
    }
}
```

```
int main(){

    dequeue();

    enqueue(1);
    enqueue(2);
    enqueue(3);
    enqueue(4);
    enqueue(5);

    enqueue(6);

    display();
    dequeue();
    display();

    enqueue(7);
    display();
    enqueue(8);
    return 0;
}
```

NLP 17/1/24

Output:

week 4:-

PROG 4: Singly Linked List Creation (insert and d

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

struct Node {

    int data;
    int *next;
};

void insertAtBeginning (struct Node *, *head_ref, int
new_data) {

struct Node * new_Node = (struct Node*) malloc
(sizeof (struct Node));

new_Node -> data = new_data;
new_Node -> next = (* head_ref);

  (* head_ref) = new_Node;
}

void insertAfter (struct Node * prev_node, int new_
data) {
  if (prev_node == NULL) {
  printf(" the given previous node cannot be
  null");
  return;
  }
  struct Node* new_node = (struct Node *) malloc
  (sizeof (struct Node));
  new_node -> data = new_data;
  new_node -> next = prev_node -> next;
  prev_node -> next = new_node;
}
```

```
void insertAtEnd(struct Node** head_ref, int new_data) {

struct Node* new_node = (struct Node*) malloc(
sizeof(struct Node));
struct Node *last = *head_ref;

new_node -> data = new_data;
new_node -> next = NULL;

if (*head_ref == NULL) {
 *head_ref = new_node;
  return;
 }
while (last -> next != NULL) last = last -> next;
last -> next = new_node;
return;
}

void printList(struct Node *node) {
      while (node != NULL) {
      printf("%d ", node -> data);
       node = node -> next;


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

    insertAtEnt(&head, 1);
    insert At Beginning (&head, 2);
    insert At Beginning (&head, 3);
    inserte At End (&head, 4);
    insert After (head -> next, 5);
    printf("Linked list:");
    printList(head);
}
```

Output:

Linked list: 3 2 5 1 4

Week 5

PROG5- Deletion in a dingly linked list.

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

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

void deleteAtbeginning (struct node **head)
{   struct node *temp;
    temp = head;
    head = head->next; free (temp);
}

void deleteAtEnd (struct node **head)
{
    struct node *temp;
    struct node *prevtemp;
    temp = prevtemp = head;
    while (temp->Next != NULL)
    {   prev temp = temp;
        temp = temp->Next;
    }
    prev temp->Next = 'Null';
    free (temp);
}

void deleteAtpos.( struct node **head) {
{ int pos; i=1;
  printf ("Enter the position you want to delete);
  scanf ("%d", &pos);
  struct node *temp, *head;
  struct node *prev = NULL;
```

NLP 18/1/24

```
while (temp->next != NULL) {
    prev = temp;
    temp = temp->next;
}
if (prev == NULL){
    *head_ref = NULL;
} else {
    prev->next = NULL;
}
free (temp);
}

void push (struct Node** head, int new_data)
{
    struct Node* new_node = (struct Node*)ma
(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = head;

}

int main() {
    struct node* head = NULL;
    push (head, 3);
    push (head, 4);
    push (head, 5);
    delete at beg (head);
    delete at pos (head, 0);
    delete at end (head);
```

P.T.O

minstack Leetcode problem.

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

typedef struct {
    int *stack;
    int* minvalues;
    int capacity;
    int top;
} Minstack;

Minstack* minstackCreate () {
        Minstack * obj = (Minstack *) malloc (sizeof(
Minstack));
        obj -> capacity = 1000;
        obj -> stack = (int *) malloc (obj -> capacity
sizeof(int));
        obj -> minvalues = (int *) malloc ( obj -> capacity
sizeof(int));
        obj -> top = -1;
        return obj;
}

void minstackPush (MinStack *obj, int val) {
        obj -> top ++;
        if (obj -> top == obj -> capacity) {

            obj -> capacity *= 2;
            obj -> stack = (int *) realloc(obj
            obj -> capacity * sizeof(int));
            obj -> minvalues , obj -> capacity *si
            (int ));
        }
```

```
obj -> stack [obj -> top] = val;

    if (obj -> top == 0 || val < obj -> minValues
    [obj -> top - 1]) {
            obj -> minValues [obj -> top] = val;
    obj -> minValues [obj -> top - 1];
    } else {
        obj -> minValues [obj -> top] = obj -> min
    Values [obj -> top - 1];
    }
}

void minStackPop (MinStack * obj) {
        if (obj -> top >= 0) {
            obj -> top--;
    }
}

int minStackTop (MinStack * obj) {
    if (obj -> top >= 0) {
        return obj -> stack [obj -> top];
    }
    return -1;
}

int minStackGetMin (MinStack * obj) {
    if (obj -> top >= 0) {
        return obj -> minValues [obj -> top];
    }
    return -1;
}
void minStackFree (MinStack * obj) {
    free (obj -> stack);
    free (obj -> minValues);
    free (obj);
}
```

Week 6.

Q.1 → write a program to implement Single linked list with following operations: Sort the linked list, Reverse the linked list, concatenate two linked list

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

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

typedef struct Node Node;

Node * createNode (int data ){
    Node * newNode = (Node *)malloc (sizeof(Node));
    newNode -> data = data;
    newNode -> next = NULL;
    return newNode;
}

void insertAtEnd (Node **head, int data){
    Node * newNode = createNode (data);
    if (*head == NULL){
        *head = newNode;
        return;
    }
    Node * temp = *head;
    while (temp -> next != NULL ){
        temp = temp -> next;
    }
    temp -> next = temp -> next;
}
```

Implementing SLL

```
        temp -> next = newNode;
}

void display (Node *head){
        Node* temp = head;
        while (temp != NULL){
            printf ("%d -> ", temp -> data);
            temp = temp -> next;
        }
        printf ("NULL");
}

void sort (Node * head){
        if (head == NULL || head -> Next == NULL){
            return;
        }
        Node * current = head;
        while (current -> next != NULL){
            Node * nextNode = current -> next;
            while (nextNode != NULL){
                if (current -> data > nextNode
                -> data ){
                int temp = current -> data;
                current -> data = nextNode -> data;
                nextNode -> data = temp;
                }
                nextNode = nextNode -> next;
            }
            current = current -> next;
        }
}
```

```
Node* reverse(Node* head){
    Node* prev = NULL;
    Node* current = head;
    Node* nextNode;
    while(current != NULL){
        nextNode = current -> next
        current -> next = prev;
        prev = current;
        current = nextNode;
    }
    return prev;
}

Node* concatenate(Node* head1, Node* head2){
    if(head == NULL){
        return head;
    }
    Node* temp = head1;
    while(temp -> next != NULL){
        temp = temp -> next;
    }
    temp -> next = head2;
    return head1;

}

int main(){
    Node* list1 = NULL;
    Node* list2 = NULL;

    insertAtEnd(&list1, 3);
    insertAtEnd(&list1, 1);
    insertAtEnd(&list1, 5);
```

```
insertAtEnd(&list2, 2);
insertAtEnd(&list2, 4);
insertAtEnd(&list2, 6);

printf("original list 1:");
display(list1);

printf("original list 2:");
display(list2);

sort(list1);
printf("Sorted list 1:");
display(list1);

list1 = reverse(list1);
printf("Reversed list 1:");
display(list1);

Node* concatenatedlist = concatenate(list1,
list2);
printf("concatenated list:");
display(concatenatedList);

return 0;
}
```

OUTPUT:-

```
Original List 1:  3 -> 1 -> 5 -> NULL.
original list 2:  2 -> 4 -> 6 -> NULL
sorted list:  1 -> 3 -> 5 -> NULL
Reversed List:  5 -> 3 -> 1 -> NULL
concatenated List:  5 -> 3 -> 1 -> 2 -> 4 -> 6 -> NUL
```

P2) Stack with linked list.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
        int data; struct Node * next;
} node;

node * head = NULL
int count = 0;
void int( int data);
int add ();
void display ();

int main (){
        int data, choice;
        printf ( 1) Push in 2 pop in 3 Exit \n choice: ");
        scanf ("%d", & choice);
        while (choice 1-3) {
            if (choice == 1){
                printf ("Enter data ");
                scanf ("%d, & data);
                insert (data);
            } else if (voice == 2) {
                printf ("Integer popped : %d\n", delete
            }
            display ();
            printf ("Enter choice ");
            scanf ("%d", & choice");
    } return 0;
}
void insert (int data) {
        node * newnode = (node *) malloc (size of (node))
        new_node -> data = data;
        new_node -> next = head; head = new_node
        count ++ ; return;
}
```

```
int delete(){
    node * temp = head;
    head = head -> next;
    int t = temp -> data
    free (temp)
      count --;
      redisn +;

}
void display (){
    node * temp = head;
    printf ("stack ")
    while (temp != NULL){
        printf ("%d", temp->data),
          ten = temp -> next }
        printf("\n");

}
```

OUTPUT
1) push
2) Pop
3) exit
coice 1
enter data.
linked list : }

Queues implementation using linked list:

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

typedef struct node ;
int val;
struct Node *val
}Node;
```

```
Node * head = NULL

void enqueue () {
    Node * ptr = (struct (Node *) malloc(sizeof
    int num;
    num = sc.next
    ptr -> val = num;
    if (head == val) {
        head = ptr;
            ptr = ptr -> next = NULL;
    }
    else {
        Node * ptr 2 = head;
    }   while ptr -> next != NULL) {
        ptr 2 -> ptr next;
        }
        ptr 2 -> next = ptr; }}

void display () {
        Node * ptr = head;
        while (ptr != NULL) {
            printf ("%d", ptr -> val);
                ptr = ptr -> next
            }
            printf ("\n")
    }

void dequeue () {
        Node * ptr = head;
    int num = ptr -> val;
    head = head -> next;
    free (ptr);
    printf ("Dequeue element : %d \n", num);
}
```

```
int main (){
    int choice;
printf("Enter the choice");
    while(1) {
        scanf ("%d", &choice);
        switch (choice) {
          case 1 : enqueue (); break;
          case 2 : dequeue (); break;
          case 3 : display, break;
          default : printf ("Exit");
                    return 0;
        }
printf ("Enter choice");
    }
```

O/P:

Enter choice 1
Enter element: 10
Enter choice 1
Enter element: 20
Enter choice 2
Enter element: 20
Enter choice 3
10.
Enter choice: 4
Existing program:

Week

Lab program.

9) WAP to implement doubly linked list with primitive operations
- Create a doubly linked list.
- Insert a new node to the left of the node.
- Delete the node based on a specific value
- display the contents of the list.

Input code:-

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

struct Node {
        int val;
        struct Node *prev;
        struct Node *next;
};

Node *head = NULL;

void insert (){
        int num, pos;
        printf ("Enter value")
        scanf("%d", &num);
        printf("Enter the node to insert left of: ");
        scanf("%d", &pos);
        Node *ptr = (Node*) malloc (sizeof(Node));
        ptr -> val = num;
        if( pos == 0){
          ptr->next = head.
          ptr -> prev = NULL;
          if ( head != NULL) {
                head -> prev = ptr;
          }
          head = ptr;
        }
```

```
Node * ptr1 = head;
if (pos != 0){
    for (int i = 0; i < pos; i++){
        ptr = ptr->next;
    }
    ptr->next = ptr1;
    ptr->prev = ptr1->prev;
    ptr1->prev->next = ptr;
    ptr1->prev = ptr;
}
}

void delete(){
    printf("Enter the value to delete");
    int loc = -1, len = -1, val;
    scanf("%d", &val);
    Node *ptr = head, *ptr2;
    while (ptr->next != NULL){
        len++;
        ptr = ptr->next;
    }
    ptr = head;
    if (loc == -1){
        printf("Delete element not in");
        return
    }
    if (loc == 0){
        printf("Deleted element %d \n", ptr->val);
        ptr = head;
        head = head->next;
        free(ptr);
        return;
    }
    if (loc == len){
```

```
ptr = head;
for (int j=0 ; j<loc ; j++){
ptr2 = ptr
ptr = ptr->next;
}
ptr2->next = ptr->next;
ptr->next->prevno = ptr2
free (ptr);
}

void main (){
int choice;
printf ("1. To insert val to left\n");
printf ("2) Delete");
printf ("3 To display list");
while (1) {
switch (choice){
case 1: insert (); break;
case 2: delete (); break
case 3: display (); break;
default: printf ("Existing");
return;
}
}
}
```

1.
1) To insert val to left
2) Delete
3) To display
Enter choice : 1
Enter node to insert : 0.
Enter val
Enter choice 1
enter val = 20
Enter no = 0.

(iii) Enter choice : 1
enter val : 30
enter node : 0.

(iv) Enter choice
20 → 30 → 10 → NULL

Enter choice : 2
Enter val : 20
Deleted element
Enter choice : 3
30 → 10 → NULL

Reverse linked list

```
struct listnode* reverseBetween( ) {

    struct listnode* l = head;
    struct listnode* r = head;
    int difference = right - left;
    if (left == right) return head;
    for(int i=0; i< left-1; i++){
        l = l->next;
    for (int i=0; i< right-1; i++){
        r = r->next;
    while (difference >=0){
        int temp = l->val;
        l->val = r->val;
        r->val = temp;
        l = l->next;
        r = l;
        for (int j=0, j< difference; j++)
            r = r->next;
        }
        different -= 2.
    }
    return head;
}
```

Week 8

15/02/2024

Tuesday

9) write a program
a) to construct a binary tree search
b) Traverse the tree using all the meathods i.e, inorder, preorder, postorder.
c) Display the elements in the tree.

Program

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

struct node {
    int key;
    struct node *left, *right;
};

struct node *newNode(int item) {
    struct node *temp = (struct node *)malloc(
sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

void inorder(struct node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d -> ", root->key);

        inorder(root->right);
    }
}
```

```
void preorder (struct node * root)
{
    if (root == NULL)
        return;
    printf ("%d →", root → key);

    preorder (root → left);

    preorder (root → right);

void postorder (struct node * root)
{
    if (root == NULL)
        return;
    post order (root → left);
    post order (root → right);

    printf ("%d → ", root → key);

}

struct node * insert (struct node *node, int key){

    if (node == NULL) return newNode (key);

    if (key < node → key)
        node → left = insert (node → left, key);
    else {
        node → right = insert (node → right, key);

    return node;

}
```

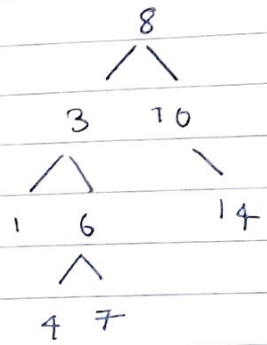
```
struct node * minvalueNode(struct node *node)
    struct node *current = node;

    while (current && current->left != NULL)
        current = current->left;

    return current;
}

int main(){

    struct node *root = NULL;
    root = insert(root, 8);
    root = insert(root, 3);
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 7);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 4);

    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");
    printf("preorder Traversal");
    preorder(root);
    printf("\n");
    printf("postorder Traversal");
    postorder(root);
    printf("\n");

}
```

OUTPUT:-

Inorder traversal: 1 → 3 → 4 → 6 → 7 → 8 → 10 → 14 →

preorder traversal: 8 → 3 → 1 → 6 → 4 → 7 → 10 → 14

postorder traversal: 1 → 4 → 7 → 6 → 3 → 14 → 10 → 8 →



1) Inorder Traversal:

- Start from the root (8).
- Traverse left: Inorder (3)
- Traverse left: Inorder (1).
- visit root: 1
- Traverse right: Inorder (3)
- visit root: 3
- Traverse right: Inorder (6)
- Traverse ~~right~~ left: Inorder (4)
- visit root: 4
- Traverse right: Inorder (6)
- visit root (6)
- Traverse right Inorder (7)
- visit root (7)
- Traverse left inorder (6)
- Traverse left inorder (3)
- Traverse right inorder (8)

visit root 8
Traverse Right inorder 10
visit root 10
Traverse Right inorder (14)
visit root 14

Preorder Traversal :

- Start from the root(8)
- visit root: 8
- Traversal left : preorder (3)
- visit root : 3
- Traversal left preorder (1)
- visit root : 1
- Traversal right : preorder (3)
- Traverse right : preorder (6)
  visit root : 6
  Traverse :

Breadth First Search

Input code:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue {
    int items[SIZE];
    int front;
    int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
void dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node {
    int vertex;
    struct node* next;
};

struct node* createNode(int);
struct Graph {
    int numVertices;
    struct node** adjLists;
    int* visited;
};
void printQueue(struct queue* q){
    int i = q->front;
```

```
    if(isEmpty(q)) {
        printf("Queue is empty");
    } else {
        printf("\nQueue contains \n");
        for(i = q->front; i<q->rear+1; i++){
            printf("%d", q->items[i]);
        }
    }
}

void bfs(struct Graph* graph, int startVertex) {
    struct queue* q = createQueue();
    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while(!isEmpty(q)) {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

        struct node* temp = graph->adjLists[currentVertex];
        while(temp) {
            int adjVertex = temp->vertex;

            if(graph->visited[adjVertex]==0){
            graph->visited[adjVertex]=1;
            enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

struct node* createNode(int v){
```

```
struct node* newNode = malloc (sizeof(struct node));
newNode->vertex = v;
newNode->next = NULL;
return newNode;
}

struct node* createNode(int v){
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// creating a graph

struct Graph* createGraph(int vertices){
    struct Graph* graph = malloc(sizeof(struct graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));
    int i;
    for(i=0; i<vertices; i++){
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest){
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    newNode = createNode(src);
```

```
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

struct queue *createQueue(){
    struct queue *q = malloc(sizeof(struct
    q->front = -1;
    q->rear = -1;
    return q;
}

int isEmpty(struct queue *q){
    if(q->rear == -1)
        return 1;
    else
        return 0;
}

void enqueue(struct queue *q, int value
    if(q->rear == SIZE-1)
        printf("\nQueue is Full!!");
    else{
        if(q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

int dequeue(struct queue *q){
    int item;
    if(isEmpty(q)){
        printf("Queue is empty");
        item = -1;
    } else {
```

```
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
        printf("Resetting queue");
        q->front = q->rear = -1;
    }
  }
  return item;
}

int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);
    return 0;
}
```

OUTPUT:

BFS Traversal starting from vertex 0:

Queue contains 0 Resetting Queue visited 0

Queue contains 2 1, visited 2

Queue contain 1 4 visited 1.

Queue contains 4 3 visited 4

Queue contains 3 Resetting queue visited;

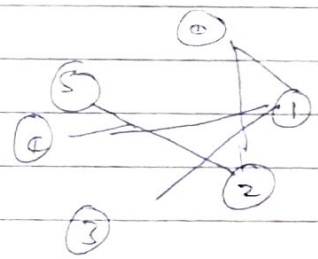
8) write a program to check whether the given graph is connected or not using DFS method.

```
# include <stdio.h>

#define MAX_VERTICS 100

void dfs(int graph[Max_vertices][MAX_VERT...
int vertex, int, vertices, int visited[MAX_VER...

int main(){

    int graph[MAX_VERTICES]
struct Graph
{ int vertices;
  int ** adjmatrix; };
struct Graph * Create Graph(int vertices)
{ struct Graph * graph (struct Graph *) malloc(
(struct graph)]; graph → vertices = vertices;
graph → adjMatrix = (int **) malloc (vertices * sizeof(...
for (int i=0; i< vertices; i++)
{ graph → adjMatrix [i] = (int *) malloc (vertices * sizeof(...
  for (int j = 0 ; j< vertices ; j++)
        graph → adjMatrix [i][j] = 0;
  }
  return graph;
}
void addEdge (struct Graph * graph, int src, int...
{
 graph → adjMatrix [src][dest] = 1;
 graph → adjMatrix [dest][src] = 1;
}
```

```
void DFS(struct Graph* graph, int startVertex, int visited[])
{ visited[startIndex] = 1;
for (int i = 0; i < graph->vertices; i++)
{ if graph->adjMatrix[startVertex][i] == 1 && visited[i] == 0) DFS(graph, i, visited);

}
int isConnected(struct Graph* graph)
{
int* visited = (int*)malloc(graph->vertices * sizeof(int));
for (int i = 0; i < graph->vertices; i++)
    visited[i] = 0;
DFS(graph, 0, visited);
for (int i = 0; i < graph->vertices; i++){
if(visited[i] == 0)
return 0;
}
return 1;
}
int main()
{ int vertices, edges, src, dest;
printf("Enter the no. of vertices: ");
scanf("%d", &vertices);
struct Graph* graph = createGraph(vertices);
printf("Enter the no of edges: ");
scanf("%d", &edges);
for(int i = 0; i < edges; i++)
{ printf("Enter edge %d (source destination): ", i+1);
scanf("%d %d", &src, &dest);
addEdge(graph, src, dest); }
if(isConnected(graph))
    printf("graph is connected\n");
else
    printf("The graph is not connected\n");
return 0;
}
```

OUTPUT:

```
enter the no. of vertices: 6
enter the no. of edges: 5
enter edge 1: 0 1
enter edge 2: 0 2
enter edge 3: 1 3
enter edge 4: 1 4
enter edge 5: 2 5
The graph is connected
```



Week-10

11.

(8) Implement hashing using linear probing

program:-

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE ]={NULL};

void insert()
{

int key, index, i, flag = 0, hkey;
printf("\n enter a value to insert into a ha
table\n");
scanf("%d", &key);
hkey = key % TABLE_SIZE;
for(i=0; i<TABLE_SIZE; i++)
{
    index = (hkey + i) % TABLE_SIZE;
    if (h[index] == NULL)
    {
        h[index] = key;
        break;
    }
}
if (i == TABLE_SIZE)
        printf("\n element cannot be inse

}

void search()
{
```

```
int key, index, i, flag=0, hkey;
printf("\n enter search element \n");
scanf("%d", &key);
hkey = key % TABLE-SIZE;
for(i=0; i<TABLE_SIZE, i++)
{
  index = (hkey+i)% TABLE- SIZE;
  if(h[index]==key)
  {
    printf("value is found at index %d", index);
    break;
  }
}
void display()
{
  int i;
  printf("\n elements in the hash table are \n");

  for(i=0; i<TABLE_SIZE; i++)
  printf("\n at index %d \t value = %d", i, h[i]);

}


main()
{
  int opt, i;
  while(1)
  {
    printf("\n Press 1. insert\t 2. Display\t 3 search
4. Exit \n");
    scanf("%d", &opt);
    switch(opt)
    {
      case 1:
        insert();
        break;
```

```
case 2:
    display();
    break;
case 3:
    search();
    break;
case 4: exit(0);
    }
}
}
```

OUTPUT:-

press 1. Insert 2. Display 3. Search 4. Exit
1
enter a value to insert into a hash table
12
press 1. Insert 2. Display 3 Search 4 Exit
1
enter a value to insert into a hash table
13
press 1. Insert 2. Display 3. search 4 Exit
2
elements in the hash table are.

at index 0 value = 00
at index 1 value = 00
at index 2 value = 12
at index 3 value = 13
at index 4 value = 00
at index 5 value = 0
at index 6 value = 0
at index 7 value = 0
at index 8 value = 0
at index 9 value = 0

Press 1. Insert 2 - Display 3. search 4. Exit
3

enter a search element .
12
value is ~~found~~ at index 2

29/2/24