INDEX

NAME: Shraddha STD: III sem SEC: 3D ROLL NO: 1BM22EC232

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
Lab 1.	21/12/24	1. Swapping using pointers 2. Dynamic Memory Allocation 3. Stack implementation	1-4	
Lab 2.	28/12/24	1. Infix to postfix 2. Evaluation of postfix	5-7	
Lab 3.	28/12/24 11/1/24	1. Linear queue 2. Circular queue	7-10	
Lab 4.	11/1/24	1. Singly Linked List (create, insert, display)	11-13	
Lab 5.	18/1/24	1. Singly Linked List (delete and display)	13-16	
Lab 6.	25/1/24	1. Sort, Reverse and Concateration of Singly Linked list. 2. Singly Linked List stack and Queue Implementation	16-20	
Lab 7.	1/2/24	Create, insert and display doubly linked list.	21-22	
Lab 8.	15/2/24	BST. and traversal.	23-24	10

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
lab-9	22/2/24	BFS	25-26	
		DFS	27.	
				10
lab10	29/2/24	Hashing.	27-29.	

21/12/23

Date
Page 1

LAB-1

1. Write a program for swapping using pointers.

```
#include <stdio.h>
void swapping(int*x, int*y);
int main()
{
    int x,y;
    printf("Enter two numbers \n");
    scanf("%d%d", &x,&y);
    printf("The value of x and y before swapping is
            %d and %d\n", x,y);
    swapping(&x,&y);
    return 0;
}
void swapping(int*x, int*y)
{
    int temp = *x;
    *x = *y
    *y = temp;
    printf("The value of x and y after swapping
            is %d and %d", *x, *y);
}
```

2. WAP for Dynamic Memory Allocation (Program should include malloc, free, calloc, realloc).

```
# include <stdio.h>
# include <stdlib.h>
int main(){
    int *arr1, *arr2;
    int size;
    printf("Enter the size of the array:");
```

```
    scanf("%d", &size);
    arr1 = (int *) malloc (size * sizeof (int));
    if (arr1 == NULL) {
        printf("Memory allocation failed. \n");
        return 1;
    }
    printf ("Memory allocated successfully using malloc.
            \n");
    free (arr1);
    printf(" Memory freed using free. \n ");

    arr2 = (int *) alloc(size, sizeof (int));
    if (arr2 == NULL){
        printf (" Memory allocation failed.\n");
        return 1;
    }
    printf ("Memory allocated and initialized to
            zero using calloc.\n");
    int newSize;
    printf(" Enter the new size for the array: ");
    scanf ("%d", &newSize);

    arr2 = (int *) realloc (arr2, newSize * sizeof (int));
    if (arr2 == NULL){
        printf("Memory reallocation failed.\n");
        return 1;
    }
    printf ("Memory reallocated successfully using realloc.\n")
    free (arr2);
    printf("Memory freed after reallocation using
            free. \n");
    return 0;
}
```

O/P =>

```
Enter the size of the array: 3
Memory allocated succefully using malloc.
Memory freed using free.
Memory allocated and initialized to zero using calloc.

Enter the new size of the array: 6
Memory reallocated successfully using realloc.
Memory freed after reallocation using free.
```

3. WAP for stack implementation - (pop, push, display functions to be implemented)

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int top = -1;
int inp_array[SIZE];
void push();
void pop();
void show();
void main()
{
    int ch;
    while (1)
    {
        printf("Operations on the stack: \n");
        printf("1.Push the element \n2. Pop the element \n
                3.Show \n 4. End \n");
        printf("Enter the choice: \n");
        scanf("%d", &ch);

        switch(ch)
```

```
        {
        case 1:
                push();
                break();
        case 2:
                pop();
                break();
        case 3:
                show();
                break();
        case 4:
                exit(0);
        default:
            printf("Invalid choice /n");
        }
    }
}
void push()
{
    int x;
    if (top == SIZE -1)
    {
        printf("Overflow\n");
    }
    else
    {
        printf("Enter the element to be added in the
                stack : \n");
        scanf("%d", &x);
        top = top + 1;
        inp_array(top) = x;
    }
}
```

```
void pop()
{
    if (top == -1)
    {
        printf("Underflow\n");
    }
    else
    {
        printf("Popped element: %d\n", inp_array[top]);
        top= top-1;
    }
}
void show()
{
    if (top == -1)
    {
        printf("Underflow \n");
    }
    else
    {
        int i;
        printf("Elements in the stack are: \n");
        for (i=top; i>=0; --i);
            printf("%d\n", inp_array[i]);
    }
}
```

OUTPUT:

Operations on the stack:
1. Push the element
2. Pop the element
3. Show
4. End

Enter the choice:
1
Enter the element to be added in the stack:
20
Operations on the stack:
1. Push the element
2. Pop the element
3. Show
4. End
Enter the choice:
1.
Enter the element to be added in the stack:
24.
Operations on the stack:
1. Push the element
2. Pop the element
3. Show
4. End.
Enter the choice:
1
Enter the element to be added in the stack:
45
Operations on the stack:
1. Push the element
2. Pop the element
3. Show
4. End
Enter the choice:
2.
Popped element: 45
Operations on the stack
1. Push the element

2. Pop the element
3. Show
4. End
Enter the choice:
3
Elements in the stack are:
20
24
Operations on the stack:
1. Push the element
2. Pop the element
3. Show
4. End
Enter the choice:
4.

28/12/23

LAB-2

1. WAP for infix to postfix conversion.

```
#include <stdio.h>
#include <ctype.h>
#define SIZE 50
char stack[SIZE];
int top = -1;
push(char ele)
{
    stack[++top] = ele;
}
char pop()
{
    return(stack[top--]);
}
int pr(char symbol)
{
    if (symbol = '^')
    {
        return(3);
    }
    else if (symbol == '*' || symbol == '/')
    {
        return(2);
    }
    else if (symbol == '+' || symbol == '-')
    {
        return(1);
    }
    else {
        return(0);
    }
```

```
}
void main ()
{
    char infix [50], postfix [50], ch, ele;
    int i=0, k=0;
    printf ("Enter the infix expression:");
    scanf ("%s", infix);
    push('#');
    while ((ch=infix [i++]) != '\0')
    {
        if (ch == '(') push (ch);
        else
            if (isalnum (ch)) postfix [k++] = ch;
            else
                if (ch==')')
                {
                    while (stack [top] != '(')
                        postfix [k++] = pop();
                    ele = pop();
                }
                else
                {
                    while (pr(stack [top]) >= pr(ch))
                        postfix [k++] = pop();
                    push(ch);
                }
    }
    while (stack [top] != '#')
        postfix [k++] = pop();
    postfix [k] = '\0';
    printf ("\n Postfix expression = %s \n", postfix);
}
```

OUTPUT:

```
Enter the infix expression: A*B+C*D-E
Postfix expression = AB*CD*
```

2. WAP for evaluation of Postfix Expression.

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

#define MAX_SIZE 100

struct stack {
    int top;
    int items[MAX_SIZE];
};

void initialize(struct stack *s);
void push(struct stack *s, int value);
int pop(struct stack *s);
int isOperator(char ch);
int compute(char operator, int operand1, int operand2);
int evaluatePostfix(char postfixExpression[]);

int main(){
    char postfixExpression[MAX_SIZE];
    printf("Enter a postfix expression:");
    gets(postfixExpression);

    printf("Result: %d\n", evaluatePostfix
                (postfixExpression));
    return 0;
}
```

```
void initialize(struct stack *s){
    s->top = -1;
}
void push(struct stack *s, int value){
    if (s->top == MAX-SIZE -1){
        printf("Stack overflow\n");
        exit(EXIT_FAILURE);
    }
    s->items[++(s->top)] = value;
}
int pop(struct stack *s){
    if (s->top = -1){
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    }
    return s->items[(s->top)--];
}
int isOperator(char ch){
    return (ch = '+'||ch = '-'||ch = '*'||ch = '/');
}
int compute(char operator, int operand1, int operand2){
    switch(operator){
    case '+':
        return operand1+operand2;
    case '-':
        return operand1 - operand2;
    case '*':
        return operand1 * operand2;
    case '/':
        return operand1 & operand2
        if (operand2 !=0){
            return operand1/operand2;
```

```
        } else {
            printf("Error: Division by zero \n");
            exit(EXIT_FAILURE);
        }
    default:
        printf("Error: Invalid operator \n");
        exit(EXIT_FAILURE);
    }
}
int evaluatePostfix(char postfixExpression[]){
    struct stack s;
    initialise(&s);

    for(int i=0; postfixExpression[i]!='\0'; i++){
        if (isdigit(postfixExpression[i])){
            push(&s, postfixExpression[i] - '0');
        } else if (isoperator(postfixExpression[i])) {
            int op2 = pop(&s);
            int op1 = pop(&s);
            int result = compute(postfixExpression[i],
                                 op1, op2);
            push(&s, result);
        }
    }
    return pop(&s);
}
```

28/12/24

Date
Page 7

LAB-3

1. WAP to simulate the working of a queue of integers using an array. Provide the following operations: Insert, Delete, Display. The program should print appropriate messages for queue empty and queue overflow condition.

```
#include <stdio.h>
#define SIZE 30
int queue[size];
int front = -1;
int rear = -1;

void insert (int a)
{
    if (rear == size - 1)
    {
        printf ("Queue overflow\n");
        return;
    }
    else
    {
        if (front == -1)
            front = 0;
        queue [++rear] = a;
    }
}
void delete ()
{
    if (front == -1 || front > rear)
    {
        printf ("Queue Empty\n");
    }
```

```
    else
    {
        front++;
    }
}
void display()
{
    int i;
    if (front == -1);
    {
        print("Queue Empty /n");
        return;
    }
    printf("Queue:");
    for(i=front; i<=rear; i++)
    {
        printf("%d", queue[i]);
    }
}
void main()
{
    int choice;
    int a;
    while(1)
    {
        print("\n1. Insert \n2. Delete \n3 Display
              \n4. Choice:");
        scanf("%d", &choice);
        switch(choice)
        {
        case 1: printf("Enter an element:");
                scanf("%d", &a);
```

```
                insert(a);
                display();
                break;
            case 2: delete();
                display();
                break;
            case 3: display();
                break;
            }
        }
}
```

OUTPUT:

```
1. Insert
2. Delete
3. Display
Choice : 1
Enter an element: 24
Queue : 24
1 Insert
2 Delete
3. Display
Choice: 1
Enter an element: 57
Queue: 24  57
1. Insert
2. Delete
3. Display.
Choice: 1
Enter an element : 92.
Queue: 24  57  92.
```

11/1/24

1. Insert
2. Delete
3. Display.
Choice: 2.
Queue: 57 92.
1. Insert
2. Delete
3. Display
Choice: 3
Queue: 57 92

2. WAP to simulate the working of a circular queue of integers using an array. Provide the following operations: Insert, Delete and Display. The program should print appropriate messages for queue empty and queue overflow conditions.

```
#include <stdio.h>
# define pf printf
# define sf scanf
# define MAX 100
int cq[MAX];
int front=-1, rear=-1;

void enqueue(){
  int data;
  pf("\nEnter the element:");
  sf("%d", &data);

  if (front==-1 && rear==-1){
    front=rear=0;
    cq[rear]=data;
  }
```

```
    else if (rear+1)%MAX ==front)
        pf ("\n Queue Overflow");
    else {
        rear = (rear+1) % MAX;
        cq [rear]= data;
    }
}
void dequeue()
{
    if(front==-1 && rear==-1){
        pf ("\n Queue is empty");
    }
    else if (front==rear) {
        front= rear = -1;
    }
    else {
        front = (front+1)%MAX;
    }
}
void display()
{
    int i = front;
    pf ("\n The circular queue is :\n");
    while (i!=rear){
        pf ("\t%d", cq[i]);
        i = (i+1) % MAX;
    }
    pf ("\t%d", cq[i]);
}
int main()
{
    int ch;
```

```
while (1) {

pf("\n---- Enter ---- \n 1. Insertion \n 2. Deletion \n
        3. Display \t");
    sf("%d", &ch);
    switch (ch) {
      case 1 : enqueue();
               display();
               break;
      case 2 : dequeue();
               display();
               break;
      case 3 : display();
               return 0;
    }
  }
}
```

OUTPUT:

```
---- Enter ----
1. Insertion
2. Deletion
3. Display      1
Enter the element : 2
The circular queue is :
        2.
---- Enter ----
1. Insertion
2. Deletion
3. Display      1
Enter the element: 6
```

The circular queue is
2 6
---- Enter ----
1. Insertion
2. Deletion
3. Display 1.
Enter the element: 10
The circular queue is
2 6 10
---- Enter ----
1. Insertion
2 Deletion
3. Display 2.
~~Enter~~ 6
The circular queue is:
6 10
---- Enter ----
1. Insertion
2. Deletion
3. Display. 3.
The circular queue is
6 10

11/1/24

LAB-4

1. WAP to Implement Singly Linked List with following operations:

(a) Create a linked list.

(b) Insertion of a node at first position, at any position and at end of list.

(c) Display the contents of the linked list.

```
#include <stdio.h>
#include<stdlib.h>
#define pf printf
#define sf scanf

struct node{
    int data;
    struct node* next;
};
struct node
*head,*newnode,*temp,*last;

void push()
{
    newnode=(struct node*)malloc(sizeof(struct
                node));
    pf("Enter the data of the node to be inserted
        at the beginning:");
    sf("%d",&newnode->data);
    newnode->next=head;
    head=newnode;
    pf("\n--- The element %d has been inserted
        at beg---",newnode->data);
```

```
        pf("\n The linked list you have created is:\n");
        temp=head;
        while(temp != NULL)
        {
            pf("%d \t", temp -> data);
            temp = temp -> next;
        }
}
void append()
{
    newnode =(struct node*) malloc(sizeof(struct
                                    node));
    pf("Enter the data of the node 1:");
    sf("%d", & newnode -> data);
    newnode ->next= NULL;
    temp = head;
    while (temp -> next != NULL)
    {
        temp = temp -> next;
    }
    temp -> next = newnode;
    pf("\n The element %d has been inserted at
        the end:", newnode -> data);
    pf("The linked list you have created is:\n");
    temp = head;
    while(temp != NULL);
    {
        pf("%d \t", temp -> data);
        temp = temp -> next;
    }
}
```

```
void insert_pos()
{
    int pos, i=0;
    newnode = (struct node*) malloc (size of (struct
                                        node));
    pf ("Enter the data of the node:");
    sf ("%d", &newnode -> data);
    pf ("\n Enter the pos:");
    sf ("%d", &pos);
    temp = last = head;
    for (i=0; i<pos; i++)
    {
    last = temp;
    temp = temp -> next;
    }
    last -> next = newnode;
    newnode -> next = temp;

    pf ("The linked list you have created is:\n");
    temp = head;
    while (temp != NULL)
    {
        pf ("%d \t", temp -> data);
        temp = temp -> next;
    }
}
void display()
{
    pf ("The linked list you have: created is:\n");
    temp = head;
    while (temp != NULL)
    {
```

```
        pf ("%d\t", temp->data);
        temp = temp->next;
    }
}
int main()
{
    int ch;
    while (1){
        pf ("\n\n Enter your choice: \n 1 Insert
            at beginning\n2. Insert at end \n 3.
            Insert at a pos \n 4. Display:");
        sf ("%d ", &ch);
        switch (ch){
        case 1 :
                push();
                break;
        case 2:
                append();
                break;
        case 3:
                insert_pos();
                break;
        case 4:
                display();
                return 0;
        }
    }
}
```

OUTPUT:

1. Insert at beginning
2. Insert at end
3. Inert at a pos
4. Display: 2

Enter the data of the appending node: 6

The element 6 has been inserted at the end: The linked list you have created is:

2 4 6

Enter your choice:

1. Insert at beginning
2. Insert at End
3. Insert at pos
4. Display: 3

Enter the data of the node: 0

Enter the pos: 2

The linked list you have created is:

2 4 0 6

Enter your choice:

1. Insert at begin
2. Insert at end
3. Insert at pos
4. Display: 4

The linked list you have created is:

2 4 0 6.

18/1/24

Date
Page 13

LAB-5

1. WAP to Implement Singly Linked List with following operation.

(a) Create a linked list

(b) Deletion of first element, specified element, and last element in the list.

(c) Display the contents of the linked list.

```
#include <stdio.h>
#include <stdlib.h>
#define pf printf
#define sf scanf

struct node{
    int data;
    struct node*next;
};
struct node *head, *newcode , *temp, * temp1;

void create()
{
    newnode = (struct node*) malloc (sizeof(struct
                                        node));
    pf("Enter the data of the node to be
        inserted at the beginning:");
    sf("%d",&newnode -> data);
    newnode ->next= head;
    head = newnode;
    pf("\n---The element %d has been inserted
        at beg---", newnode ->data);
    pf("\nThe linked list you have created
```

```
        is :/n");
        temp = head;
        while (temp != NULL)
        {
            pf("%d\t", temp->data);
            temp = temp -> next;
        }
}
void delete_at_beg(){
        temp = head;
        if (head = NULL){
            pf("\n The linked list is empty");
        }
        else if (head -> next == NULL)
        {
            head = NULL;
            pf("\n The only node is deleted, the list
                is empty.");
        }
        else{
            head = temp->next;
            free(temp);
        }
}
void delete at end(){
    if (head == NULL)
    {
        pf("\n The linked list is empty");
    }
    else if (head->next == NULL)
    {
        head = NULL;
```

```
            pf("\nThe only node is deleted, the list
                is now empty.");
        }
        else{
            temp=head;
        while(temp->next != NULL):
        {
            temp1 = temp;
            temp = temp->next;
        }
        temp1->next =NULL;
        free(temp);
    }
}
void delete_at_pos()
{
    temp=head;
    int pos,i;
    if (head=NULL){
        pf("\nThe linked list is empty.");
    }
    else if(head->next==NULL)
    {
        head=NULL;
        pf("\nThe only node is deleted, the list is
            now empty");
    }
    else {
        pf("\nEnter the position of element to be
            deleted:");
        sf("%d",&pos);
```

```
        for(i=1; i<pos; i++)
        {
            temp1=temp;
            temp=temp->next;
            if(temp==NULL){
                pf("\nThere are less than %d
                    elements in the list", pos);
                return 0;
            }
        }
        temp1->next=temp->next;
        free(temp);
    }
}
void display()
{
    pf("The linked list you have created is:\n");
    temp=head;
    while(temp!=NULL)
    {
        pf("%d\t", temp->data);
        temp=temp->next;
    }
}
int main()
{
    int ch;
    while(1)
    {
        pf("\n----Menu----\n1. Create\n2. Delete
            at_beg\n3. Delete at end\n4. Delete
            at pos\n5. Display:");
```

```
        sf("%d", &ch);
    switch(ch)
    {
    case 1 : create();
             break;
    case 2: delete_at_beg();
             break;
    case 3 : delete_at_end();
             break;
    case 4: delete_at_pos();
             break;
    case 5: display();
             return 0;
     }
   }
}
```

OUTPUT:

```
The linked list you have created is:
50      40      30      20      10
----Menu---
1. Create
2. Delete at beg
3. Delete at end
4. Delete at pos
5. Display:    2.

----Menu---
1. Create
2. Delete at beg
3. Delete at end
```

4. Delete at pos
5. Display : 3
---- Menu ----
1. Create
2. Delete at beg
3. Delete at end
4. Delete at pos
5. Display : 4
Enter the position of element to be deleted: 3

---- Menu ----
1. Create
2. Delete at beg
3. Delete at end
4. Delete at pos
5. Display : 5

The linked list you have created is:
40 30

25/1/24

Date
Page 16

LAB-6

- Sort the singly linked list. Reverse and concatenation.

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int data;
    struct Node* next;
};
type of $truct Node  Node;
Node* create Node( int value)
{
    Node* newNode = (Node*) malloc (size of (Node));
    newNode -> data = value;
    newNode -> next = NULL;
    return newNode;
}
void display list (Node * head)
{
    while (head != NULL)
    {
        pf ("%d->", head -> data);
        head = head -> next;
    }
    pf ("NULL");
}
Node * sortlist (Node * head)
{
    if (head == NULL || head -> next = NULL)
    {
        return head;
    }
```

```
    int swapped;
    Node * temp;
    Node * end = NULL;

    do
    {
        swapped = 0;
        temp = head;
        while (temp -> next != end)
        {
            if (temp -> data -> temp -> next -> data)
            {
                int tempData = temp -> data;
                temp -> data = temp -> next -> data;
                temp -> next -> data = tempData;
                swapped = 1;
            }
            temp = temp -> next;
        }
        end = temp;
    }
    while (swapped)
     return head;
}
Node * reverse List (Node * head)
{
    Node* prev = NULL;
    Node* current = head;
    Node* nextnode = NULL;
    while (current != NULL)
    {
        nextNode = current -> next;
```

```
        current->next = prev;
        prev = current;
        current = nextnod;
    }
    return prev;
}
Node * concatenation (Node * L1, Node* L2)
{
    if (L1 = NULL)
        return L2;
    Node * temp = L1;
    while (temp ->next != NULL)
    {
        temp = temp -> next;
    }
    temp->next = L2;
    return L1;
}
int main()
{
    Node * L1 = create Node (3);
    L1 -> next = create Node(1);
    L1 -> next -> next = createNode(4);

    Node* L2 = CreateNode (2);
    L2 -> next = creat Node (5);
    printf ("Original list1");
    displaylist (L1);
    printf ("Original list2");
    displaylist (L2);
```

```
L1 = sortlist (L1);
pf ("Sorted list 1:");
display list (L1);

L1 = sortList (L2);
pf ("Sorted list 2:");
displayList (L2);

L1 = reverselist (L1);
pf (" Reverse List1");
displayList (L1);

Node * concatenated = concatenated List (L1&, L2);
pf (" Concatenated list !");
display List (concat List);
return 0;
}
```

- Stack implementation of Single Linked List.

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node * next;
};
struct node * head, * newnode, * temp, * temp1;

void push()
{
    newnode = (struct node*) malloc (sizeof (struct node));
    pf ("Enter data to push");
    scanf ("%d", &newnode -> data);
    newnode -> next = head;
    head = newnode;
}
void display()
{
    pf ("The created stack is");
    temp = head;
    while (temp != NULL)
    {
        pf ("%d", temp -> data);
        temp = temp -> next;
    }
}
void pop ()
{
    if (head == NULL);
        printf ("Stack empty");
    else
```

```
    {   {
            head = head -> next;
        }
    }
}
int main()
{
    int ch;
    while(1)
    {
        pf(" 1. Push \n 2. Pop \n 3. Display \n. 4. Exit");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: push();
                display();
                break;
        case 2: pop();
                display();
                break;
        case 3: display();
                break;
        case 4: exit(0);
        }
    }
}
```

OUTPUT

1. Push
2. Pop
3. Display
4. Exit

```
Choice: 1
Enter data: 20

1. Push
2. Pop
3. Display
4. Exit
Choice: 1
Enter data: 25

The stack is 20 25.
```

- Queue implementation of Singly Linked List

```
# include <stdio.h>
# include <stdlib.h>
struct node{
    int data;
    struct node* next;
};
struct node *head, * newnode, * temp, *temp1;
void pushqueue()
{
    newnode = (struct node*) malloc (sizeof (struct node));
    pf("Enter data to push");
    scanf("%d", &newnode -> data);
    newnode -> next = head;
    head = new node;
}
void display()
{
    pf("The queue is:");
    temp = head;
```

```
        while (temp != NULL)
        {
            pf("%d", temp -> data);
            temp = temp -> next;
        }
}
void popqueue()
{
    if (head == NULL)
        pf("Queue empty");
    else
    {
        temp = temp1 = head;
        while (temp -> next != NULL)
            {
                temp1 = temp;
                temp = temp -> next;
            }
        temp1 -> next = NULL;
        free(temp);
    }
}
int main()
{
    int ch;
    while (1)
    {
        pf("1.Push \n 2. Pop \n 3. Display");
        scanf("%d", &ch);
        switch (ch)
        {
```

```
            case 1 : pushqueue();
                     display();
                     break;
            case 2 : popqueue();
                     display();
                     break;
            case 3 : display();
                     break;
                }
        }
}
```

OUTPUT :

```
1. Push
2. Pop
3 Display
Choice : 1
Enter element : 50

1. Push
2. Pop
3. Display
Choice : 1
Enter element : 60

The created list is 50 60.
```

1/2/24

LAB-7

- WAP to implement doubly linked list with primitive operations.

(a) Create a doubly linked list.
(b) Insert a new node to the left of the node.
(c) Delete the node based on a specific value.
Display the contents of the list.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct Node *prev;
    struct Node* next;
};
struct Node* createNode(int data)
{
    struct Node* newNode = (struct Node*) malloc
                           (sizeof(struct Node));
    if (newNode == NULL)
    {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
```

```
void insertNodeToLeft(struct Node* head, struct
        Node*target, int data)
{
    struct Node* newNode = createNode(data);
    if (target->prev != NULL)
    {
        target->prev->next = newNode;
        newNode->prev = target->prev;
    }
    else
    {
        head = newNode;
    }
    newNode->next = target;
    target->prev = newNode;
}

void deleteNode(struct Node* head, int value)
{
    struct Node* current = head;
    while (current != NULL)
    {
        if (current->data == value)
        {
            if (current->prev != NULL)
            {
                current->prev->next = current->next;
            }
            else
            {
                head = current->next;
            }
```

```
            if(current->next != NULL)
            {
                current->next->prev = current->prev;
            }
            free(current);
            return;
        }
        current = current->next;
    }
    printf("Node with value %d not found\n",
                                        value);
}
void displayList(struct Node* head)
{
    printf("Doubly Linked list:");
    while(head != NULL){
    {
        printf("%d<-> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
int main()
{
    struct node *head = NULL;
    head = createNode(1);
    head->next = createNode(2);
    head->next->prev = head;
    head->next->next = createNode(3);
    head->next->next->prev = head->next;
```

```
    displayList (head);
    insertNodeToLeft (head, head -> next, 10);
    printf ("After insertion \n");
    displayList (head);

    deleteNode (head, 2);
    printf ("After deletion: \n");
    displayList (head);

    return 0;
}
```

OUTPUT:

```
Doubly linked list : 1 <-> 2 <-> 3 <-> NULL
After insertion:
Doubly linked list : 1 <-> 10 <-> 2 <-> 3 <-> NULL
After deletion:
Doubly linked list: 1 <-> 10 <-> 3 <-> NULL
```

Sneha
1/2/24

LAB-8

WAP to construct a BST.
WAP to traverse the tree using all the methods i.e, pre-order, in-order and postorder.
WAP to display the elements in the tree.

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

typedef struct BST
{
   int data;
   struct BST* left;
   struct BST* right;
}node;

node *create(){
    node *temp;
    printf("Enter data \n");
    temp=(node*)malloc(sizeof(node));
    scanf("%d",&temp->data);
    temp->left=temp->right=NULL;
    return temp;
}
void inorder(node* root)
{
    if(root!=NULL)
    {
        inorder(root->left);
        printf("%d ",root->data);
        inorder(root->right);
    }
```

```
void preorder(node *root)
{
    if(root!=NULL)
    {
        printf("%d",root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(node* root)
{
    if(root!=NULL)
    {
        postorder(root->left);
        postorder(root->right);
        printf("%d",root->data);
    }
}
void insert(node *root, node *temp)
{
    if(temp->data < root->data)
    {
        if(root->left!=NULL)
            insert(root->left,temp);
        else
            root->left=temp;
    }
    else if(temp->data > root->data)
    {
        if(root->right!=NULL)
            insert(root->right,temp);
        else
            root->right=temp;
```

```
    else
    {
        printf("Ignoring duplicate data %d \n", temp->data);
    }
}
int main()
{
    char ch;
    int choice;
    node *root = NULL, *temp;
    do
    {
        temp = create();
        if (root == NULL)
            root = temp;
        else
            insert(root, temp);
        printf("\n Do you want to continue? (y/n): ");
        getchar();
        scanf("%c", &ch);
    }
    while (ch == 'y' | ch == 'Y');
    printf("Choose traversal type \n");
    printf("1. Preorder traversal \n 2. Postorder Traversal
        \n 3. Inorder traversal \n");
    printf("Enter your choice (1-3) \n");
    scanf("%d", &choice);

    switch (choice){
        case 1:
            printf("\n BST in preorder traversal \n");
            preorder(root);
            break;
```

```
        case 2:
            printf("\nBST in postorder traversal\n");
            postorder(root);
            break;
        case 3:
            printf("\nBST in inorder travelsal\n");
            inorder(root);
            break;
        default:
            printf("Invalid Choice");
    }
    return 0;
}
```

O/P:-

```
Enter data:
15
Do you want to continue?(y/n):y
Enter data:
5
Do you want to continue?(y/n):y
Enter data:
20
Do you want to continue?(y/n):y
Enter data:
37
Do you want to continue?(y/n):y
Enter data:
24
Do you want to continue?(y/n):y
Enter data:
```

Do you want to continue ? (y/n) : n

Choose traversal type

1. Preorder traversal

2. Postorder traversal

3. Inorder traversal

Enter your choice (1 - 3)

2.

BST in postorder traversal

3 5 24 37 20 15

Date
Page

LAB-9

1. WAP to traverse a graph using BFS method.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
struct queue {
    int items[MAX_SIZE];
    int front;
    int rear;
};
struct Queue* createQueue(){
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct
    queue->front = -1;
    queue->rear = -1;
    return queue;
}
int isEmpty(struct Queue* queue){
    if(queue->rear == -1)
        return 1;
    else
        return 0;
}
void enqueue(struct Queue* queue, int value){
    if(queue->rear == MAX_SIZE-1)
        printf("\nQueue is full!");
    else {
        if(queue->front == -1)
            queue->front = 0;
        queue->rear++;
        queue->items[queue->rear] = value;
    }
}
```

```
int dequeue(struct Queue* queue) {
    int item;
    if (isEmpty(queue)) {
        printf("Queue is empty");
        item = -1;
    }
    else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
    return item;
}
struct Graph {
    int vertices;
    int** adjMatrix;
};
struct Graph* createGraph(int vertices)
{
    struct Graph* graph = (struct graph*) malloc(sizeof(struct Graph));
    graph->vertices = vertices;
    graph->adjMatrix = (int*)malloc(vertices * sizeof(int));
    for (int i=0; i<vertices; i++) {
        graph->adjMatrix[i] = (int*) malloc(vertices * sizeof(int));
        for(int j=0; j<vertices; j++)
            graph->adjMatrix[i][j] = 0;
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest){
```

```
    graph->adjMatrix[src][dest] = 1;
    graph->adjMatrix[dest][src] = 1;
}
void BFS(struct Graph* graph, int startVertex){
    int visited[MAX_SIZE] = {0};
    struct Queue* queue = createQueue();
    visited[startVertex] = 1;
    enqueue(queue, startVertex);
    printf("Breadth First Search Traversal:");
    while(!isEmpty(queue)){
        int currentVertex = dequeue(queue);
        printf("%d", currentVertex);
        for(int i=0; i<graph->vertices; i++){
            if(graph->adjMatrix[currentVertex][i] == 1 && visited[i]
            {
                visited[i] = 1;
                enqueue(queue, i);
            }
        }
    }
    printf("\n");
}
int main(){
    int vertices, edges, src, dest;
    printf("Enter the number of vertices:");
    scanf("%d", &vertices);
    struct Graph* graph = createGraph(vertices);
    printf("Enter the number 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);
    }
```

```
    int startVertex;
    printf("Enter the starting vertex for BFS:");
    scanf("%d", &startVertex);
    BFS(graph, startVertex);
    return 0;
}
```

O/P =>

```
Enter the number of vertices: 4
Enter the number of edges: 4
Enter edge 1 (source destination): 0 1
Enter edge 2 (source destination): 0 2
Enter edge 3 (source destination): 1 3
Enter edge 4 (source destination): 2 3
Enter the starting vertex for BFS: 2
Breadth First Traversal: 2 0 3 1
```

2. WAP to check whether given graph is connected or not using DFS method.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
struct Graph{
    int vertices;
    int** adjMatrix;
};
struct Graph* createGraph(int vertices){
    struct Graph*graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->vertices = vertice;
    graph->adjMatrix = (int*) malloc (vertices * sizeof(int));
    for (int i=0; i<vertices;i++){
        graph->adjMatrix[i] = (int*)malloc(vertices*sizeof(int));
        for (int j=0; j<vertices; j++)
            graph->adjMatrix[i][j] =0;
    }
    return graph;
}
void addEdge (struct Graph* graph, int src, int dest){
    graph->adjMatrix [src][dest] = 1;
    graph->adjMatrix [dest][src] = 1;
}
void DFS(struct Graph* graph,int startVertex, int visited[])
{
    visited [startvertex] = 1;
    for (int i=0; i< graph->vertices ; i++){
    {   if(graph->adjMatrix [startVertex][i] ==1 && visted[i]
        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;
    }
    return 1;
}
int main(){
    int vertices, edges, src, dest;
    printf ("Enter the number of vertices:");
    scanf("%d", &vertices);
    struct Graph* graph = createGraph(vertices);
    printf ("Enter the number 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);
    }
    if (isConnected (graph))
        printf("The graph is connected.\n");
    else
        printf("The graph is not connected \n");
    return 0;
}
```

O/P =>

```
Enter the number of vertices: 4
Enter the number of edges : 4
Enter edge 1 (source destination): 0  1
```

Enter edge 2 (source destination): 1 2
Enter edge 3 (source destination): 2 3
Enter edge 4 (source destination): 3 0
The graph is connected.

29/07/24

LAB-10

```
#define TABLE_SIZE 100
#define KEY_LENGTH 5
#define MAX_NAME_LENGTH 50
#define MAX_DESIGNATION_LENGTH 50
struct Employee {
    char key[KEY_LENGTH];
    char name[MAX_NAME_LENGTH];
    char designation[MAX_DESIGNATION_LENGTH];
    float salary;
};
struct HashTable{
    struct Employee* table[TABLE_SIZE];
}
int hash_function(const char *key, int n)
{
    int sum=0;
    for(int i=0; key[i]!='\0'; i++){
        sum += key[i];
    {
    return sum%n;
}
void insert(struct hashTable *ht, struct Employeee* emp){
    int index=hash_function(emp->key, TABLE_SIZE);
```

```
    while (ht -> Table[index] != NULL)
        index = (index + 1) % TABLE_SIZE;
    ht -> table [index] = emp;
}
struct Employee *search (struct hashtable *ht, const char *key){
    int index = hash_function (key, TABLESIZE);
    while (ht -> table [index] -> key, key) == 0{
        {
            return ht -> table [index];
        }
        index = (index + 1) % TABLESIZE;
    } }
    return NULL;
}
int main()
{
    struct HashTable ht;
    struct Employee * emp;
    char key[KEY_LENGTH];
    char filename[100];
    char line[100];
    FILE * file;

    for(int i=0; i< TABLESIZE; i++){
        ht.Table[i] = NULL;
    }
    printf("Enter filename:");
    scanf("%s", filename);
    file = fopen (filename, "r");
    if (file == NULL)
    {
        printf (" Error opening");
        return 1;
    }
```

```
while (fgets (line,sizeof (line), file)){
    emp = (struct Employee*) malloc(sizeof (struct Employee));
    insert (&ht, emp);
}
int choice;
int search key
    printf (" 1. search \n 2. Exit ");
    printf (" Enter choice ");
    scanf (" %. s", &choice);
    switch (choice)
    {
        case 1 : printf (" Enter key ");
                 scanf (" %.s", key);
                 emp = search (&ht, key);
                 if (emp != NULL)
                 {
                 printf (" Employee found with key %.s\n" employee);
                 }
                 else {
                     printf (" Not found ");
                 }
                 break;
        Case 2 : printf (" Exiting ");
                 break;
        default : printf (" Invalid choice: ");
    }
    while (choice != 2);
    return 0;
}.
```

Hackerrank

Swapping Subtree function

```
void SwapAtLevel (node *root, int k, level) {
    if (root == NULL)
        return;
    if (level % k == 0){
        node* temp = root->left;
        root->left = temp;
        }
    SwapAtLevel (root->left, k, level+1);
    SwapAtLevel (root->right, k, level+1); }
int** SwapNodes (int index_rows, int i)
columns, int** indexes, queries)
Node** node = (Node**) malloc (index+1) sizeof (node));
for (i = 0; i<indexes-row; i++){
    int leftindex = index [i] [0];
    int rightindex = index [i] [1];
    if (leftindex != -1) nodes (i+1) -> left = nodes[leftindex];
    if (rightindex != -1) nodes (i+1) -> right = nodes[right->root];
    for (i=1; i<=n; i++){
        if (s[i]). count ++;
        if (count == n)
            printf (" Connected");
        else    printf ("not connected");
        return 0;
}
```

O/P ⇒ Enter vertices 26
Enter matrix:

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

1 → 2
2 → 4
4 → 3
3 → 5
3 → 6
Connected

2/3/24