INDEX

NAME: Navya STD.: III sem SEC.: D ROLL NO.: 175 SUB.: DS observation book.

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	07.12.23	Week 0	1	10
2	21.12.23	Week 1	7	10
3.	28.12.23	Week 2	13	10
4.	11.01.2024	Week 3	23	10
5	18.01.2024	Week 4	33	10
6.	18.01.2024	Leet code [minstack]	31	5
7.	18.01.2024	Leet code [reverse]	40.	10
8.	25.01.2024	Week - 5	43.	10
9.	25.01.2024	Leetcode [linked list]	48.	10
10.	1.02.24	Week-6	52.	10
11	1.02.24	Leet code[Singly Linked List]	58.	
12.	15-02-24	Week - 7	60.	10
13.	15-02-24	leetcode	61	
14.	22-02-24	Week - 8	62	
15.	22-02-24	leetcode	65	
16.	29-02-24	Week-9	68	10

LAB PROGRAM - 01

1. Output:

Enter 1 to create account
Enter 2 to withdraw amount
Enter 3 to deposit an amount
Enter 4 for balance inquiry
Enter 0 to exit.

Enter an option:
1
Enter the id number of applicant:
2468
Enter the initial amount to deposit:
5000
Enter pin number:
1234

Account details recorded successfully
Your account number is 3247

Enter 1 to create account
Enter 2 to withdraw amount
Enter 3 to deposit an amount
Enter 4 for balance inquiry
Enter 0 to exit

Enter an option:
2
Enter the account number:
3247
Enter the amount withdraw:
8000

Enter your pin number:
1234
Please collect your cash.

Enter 1 to create account
Enter 2 to withdraw amount
Enter 3 to deposit an amount
Enter 4 four balance inquiry
Enter 0 to exit

Enter an option:
3
Enter account number:
3947
Enter the amount to deposit:
1000
Deposited Successfully

Enter 1 to create account
Enter 2 to withdraw amount
Enter 3 to deposit an amount
Enter 4 four balance inquiry
Enter 0 to exit

Enter an option:
4
Enter the account number:
3947
The balance is 5000
Enter 1 to create account
Enter 2 to withdraw amount
Enter 3 to deposit an amount
Enter 4 four balance inquiry
Enter 0 to exit

Enter an option:
0
Exiting....

2. Output:

Enter the number of strings:
3
Enter the strings:
Basvanagudi
hebbal
bmsce
The lexicographically sorted strings are:
Basvanagudi
bmsce
hebbal.

3. Output:

Enter the number of elements in the array:
4
Enter the elements of the array:
a[0] = 45
a[1] = 64
a[2] = 72
a[3] = 81
Enter the element to check:
45
The element 45 is present in the array.

4. Output:

Enter the larger string:
BMSCE123456
Enter the substring:
E123
The substring is found in the larger string.

5. Output:

Enter the number of elements in the array:
6
a[0] = 1
a[1] = 2
a[2] = 1
a[3] = 3
a[4] = 2
a[5] = 1

Enter the number whose index of the last occurrence has to be found:
1
The index of the last occurrence of the number is 5

6. Output:

Enter the number of elements in the array:
5
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
Enter the number to be found:
4
The number is found at position 4.

7. Output:

Enter the number of elements in the array:
4
a[0] = 10
a[1] = 20
a[2] = 30
a[3] = 40
Enter the number to find:
30
The number is found at position 3.

8. Output:

Enter the number of elements in the array:
4
Enter the elements of the array:
a[0] = 45
a[1] = 69
a[2] = 2
a[3] = 94

The minimum element in the array:
2
The maximum element in the array:
96.

LAB-02

1.
```
#include <stdio.h>
void swapNumbers(int *a, int *b);
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main()
{
    int n1, n2;
    printf("Enter the first number: \n");
    scanf("%d", &n1);
    printf("Enter the second number: \n");
    scanf("%d", &n2);
    printf("Before swapping: \n");
    printf("First number: %d\n", n1);
    printf("Second number: %d\n", n2);
    swapNumbers(&n1, &n2);
    printf("\n After swapping: \n");
    printf("First number: %d\n", n1);
    printf("Second number: %d\n", n2);
    return 0;
}
```

Output:

Enter the first number:
24
Enter the second number
62

Before swapping:
first number: 24
second number: 62

After swapping:
first number : 62
second number : 24

2.
```
#include <stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr1, *ptr2;
    int n;
    printf("Enter
    printf("Memory allocation through
     malloc\n");
     int *ptr = (int*)malloc(5*(sizeof(int)));
    if(ptr == NULL){
      printf("Memory allocation failed");
    }
    for(int i=0;i<5;i++){
        ptr[i]=i;
        printf("%d\t",ptr[i]);
    }
    printf("\n Memory allocation through
        calloc\n");
    int *ptr1 = (int*) calloc(4,(sizeof(int)));
    if(ptr1==NULL){
      printf("Memory allocation failed");
    }
     for(int i=0; i<4; i++){
          ptr1[i]=i*2;
          printf("%d\t",ptr1[i]);}
```

```
printf("\n Memory allocation through
  realloc \n");
int *ptr2 = (int*) realloc (ptr1, 6 * (sizeof(int)));
if (ptr2 == NULL) {
   printf("Memory re-allocation failed");
  free (ptr1);
}
for (int i=4 ; i<6 ; i++); {
   ptr2[i] = i*2;
 }
for (int i=0; i<6 ; i++) {
   printf ("%d \t", ptr2[i]); }
 free (ptr2);
  printf(" \n Memory after free \n");
   for (int i=0; i<6; i++) {
     printf ("%d \t", ptr2[i]); }
   printf("\n Memory allocation through
     calloc \n");
   ptr2 = (int*) calloc (4, (sizeof (int)));
     if (ptr1 == NULL) {
        printf (" Memory allocation failed");
   }
     for (int i=0 ; i<4; i++) {
        ptr2[i] = i*2;
         printf ("%d \t", ptr2[i]); }
     return 0;
   }
```

p.t:0

Output :

```
Memory allocation through malloc
0 1 2 3 4
Memory allocation through calloc
0 2 4 6
Memory allocation through realloc
0 2 4 6 8 10
Memory after free
11670576 0 11665744 0 8 10
Memory allocation through calloc
0 2 4 6
```

```
#include <stdio.h>
#include <stdlib.h>
#define max 100
int top = -1;
int stack [max];
void push (int a);
int pop ();
void display ();
int main()
{
    int arr [100], size;
    printf(" Enter the array size : ");
    scanf(" %d ", &size);
    printf(" Enter values of stack : \n ");
    for (int i=0 ; i < size ; i++) {
        scanf (" %d ", &arr [i]);
        push (arr [i]);
    }
```

```
printf("Stack before popping:\n");
display();
for(int i=size-1; i>=0; i--)
{
    pop();
}
printf("Stack after popping:\n");
display();
return 0;
}
void push(int a){
    if(top==max-1){
        printf("stack overflow\n");
    return;
}
    top=top+1;
    stack[top]=a;
}
    int pop(){
    if(top==-1){
        printf("Stack overflow\n");
        return -1;
    }
    top--;
    return stack[top];
}
    void display(){
    if(top==-1){
        printf("Stack is empty\n");
        return;
}
    printf("Stack elements:\n");
```

p.t.o

```
for(int i=0; i<=top; i++){
    printf("%d\t", stack[i]);
}
    printf("\n");
}
```

Output:

```
Enter array size: 5
Enter values of stack:
1
2
3
4
5
Stack before popping:
Stack elements:
1 2 3 4 5
Stack after popping:
Stack is empty
```

Sneha
26/12/23

28.12.23

Week-2
Lab-03

1)
```
#include<stdio.h>
#include<stdlib.h>
#define MAX_SIZE 100
char stack[MAX_SIZE];
int top = -1;
void push(char item)
{
    if (top == MAX_SIZE-1) {
    printf("stack overflow\n");
    exit(EXIT_FAILURE);
    }
    stack[++top] = item;
}
char pop() {
        if (top == -1) {
            printf("stack underflow\n");
            exit(exit_failure);
            }
        return stack[top--];
}
int precedence(char operator) {
    switch(operator) {
        case '^':
            return 3;
        case '*':
        case '/':
            return 2;
        case '+':
        case '-':
            return 1;
        default:
```

```
    return 0;
}
}
void infixTopostfix(char infix[]) {
    char postfix[MAX_SIZE];
    int i, j;
    i=j=0;
    while (infix[i] != '\0') {
        char symbol = infix[i];
        if ((symbol >= 'a' && symbol
        <= 'z') || (symbol >= 'A' && symbol
        <= 'Z')) {
            postfix[j++] = symbol;
        } else if (symbol == '(') {
            push(symbol);
        } else if (symbol == ')') {
            while (top != -1 && stack[top] !=
            '(') {
                postfix[j++] = pop();
            }
            if (top == -1) {
            printf("Invalid Expression:
            Mismatched ~~paren~~ parentheses\n");
            exit(EXIT_FAILURE);
            }
            pop();
        } else {
            while (top != -1 && precedence
            (stack[top]) >= precedence
            (symbol)) {
                postfix[j++] = pop();
            }
            push(symbol);
        }
```

```
        i++;
    }
    while (top != -1) {
        if (stack[top] == '(') {
            printf("invalid expression: mismatched
            parentheses");
            exit(EXIT_FAILURE);
        }
        postfix[j++] = pop();
    }
    postfix[j] = '\0';
    printf("postfix expression: %s\n", postfix);
}
int main() {
    char infix[MAX_SIZE];
    printf("Enter a valid parenthesized
    infix expression:");
    fgets(infix, MAX_SIZE, stdin);
    infixToPostfix(infix);
    return 0;
}
```

Output:

```
Infix: A*B+C*D-E
AB* + C*D - E
AB* + CD* - E
AB*CD* + - E
AB*CD*+E -
Postfix:
AB*CD* +E -
```

(2) Postfix evaluation:

```
#include <stdio.h>
int stack [20];
int top = -1;
void push (int x)
{
  stack [++top] = x;
}
int pop ()
{
  return stack [top--];
}
int main ()
{
  char exp [20];
  char *e;
  int n1, n2, n3, num;
  printf (" Enter the expression :");
  scanf (" %s", exp);
  e = exp;
while (*e != '\0')
{
  if (isdigit (*e))
  {
    num = *e - 48;
    push (num);
  }
  else
  {
    n1 = pop ();
    n2 = pop ();
```

```
switch (*e)
{
case '+';
{
  n3 = n1 + n2;
  break;
}
case '-';
{
  n3 = n1 - n2;
  break;
}
case '*';
{
  n3 = n1 * n2;
  break;
}
case '/';
{
  n3 = n2 / n1;
  break;
}
e++;
}
printf("\n the result of expression
%s = %d \n", exp, pop());
```

Output:

Enter expression : 12 * 34 * +5 ↵
the result of expression 12* 34* +5 = 9

3) Linear Queue Implementation

```
#include <stdio.h>
#define SIZE 5
int front = -1, rear = -1;
int queue[SIZE];
int main()
{
    int c, elem, cl=0;
    while (cl==0)
    {
      printf("\n Enter 1 to insert into
        the queue \n");
      printf("\n Enter 2 to delete from
        the queue\n");
      printf(" Enter 3 to display \n");
      printf(" Enter 0 to exit \n");
      printf(" Enter your choice: \n");
      scanf ("%d", &c);
      switch (c)
      {
        case 0: printf (" Exiting... \n") ;
          cl=1; break;
        case 1:
          printf(" Enter the element to be
            inserted :\n");
          scanf ("%d", &elem);
          enqueue (elem);
          break;
        case 2: dequeue ();
              break;
        case 3: display_q ();
              break;
```

```
            default : printf ("Invalid choice \n");
                      break;
            }
        }
    return 0;
}
void enqueue (int elem)
{
    if (rear == SIZE - 1)
    {
        printf ("Queue is full, Cannot insert \n");
        exit (0);
    }
        queue [++rear] = elem;
        printf ("Element %d inserted successfully
        \n", elem);
    }

void dequeue ()
{
    if ( front == -1 || front > rear)
    {
        printf ("Queue is empty. Cannot delete \n")
        exit (0);
    }
    printf ("The element deleted is: %d\n ",
    queue [front]);
    front ++;
}
```

```
Void display_q()
{
 if (front == -1)
  {
    printf("Queue is empty \n");
     exit(0);
  }
   printf("The elements of the queue are: \n");
    for(int i = front ; i<=rear; i++)
      printf("%d", queue[i]);
}
```

Output:

```
Enter 1 to insert into the queue
Enter 2 to delete from the queue
Enter 3 to display
Enter 0 to exit

Enter your choice:
1
Enter the element to be inserted:
1
Element 1 inserted successfully
Enter your choice:
1
Enter the element to be inserted:
2
Element 2 inserted successfully
Enter your choice:
3
The elements of the queue are:
12
```

Enter your choice :
2
~~Enter~~ the element deleted is 1

01.24

Week - 3

1. Circular Queue Implementation:

```
#include <stdio.h>
#define SIZE 5
int front = -1, rear = -1;
int CQ [SIZE];
int main()
{
    int c, c1=0;
    while( c1==0)
    {
     printf("\n Enter 1 to insert into the circular
      queue \n");
     printf(" Enter 2 to delete from circular
      queue \n");
     printf(" Enter 3 to display ~~elements of~~
       ~~the circular queue~~ .\n");
     printf(" Enter 0 to exit \n");
     printf (" Enter your choice: \n");
     scanf (" %d", &c);
     switch (c)
     {
        case 0: printf(" Exiting... \n") ;
                c1 = 1; break;
        case 1: enqueue() ; break;
        case 2: dequeue (); break;
        case 3: display_q(); break;
        default : printf (" invalid choice\n");
     }
    }
    return 0;
}
```

```
Void enqueue ()
{
   int elem ;
   if ( rear == SIZE-1)
   {
     printf(" Queue is full. Cannot insert\n")
     exit (0);
   }
   if (front == -1)
   {
      front =0;
      rear =0;
   }
   printf (" Enter the element to be inserted:
            \n");
   scanf ("%d", &elem);
   CQ [rear] = elem;
   printf (" element %d inserted successfully
      \n ", elem);
   rear =(rear+1) %SIZE;
}
Void dequeue ()
{
   if ( front == -1)
   {
      printf(" Queue is empty\n");
      exit (0);
   }
   printf (" The element deleted is %d \n",
            CQ [front]);
   front = (front +1) % SIZE ;
}
```

```
Void display_q()
{
  if (front == -1)
  {
    printf("Queue is empty \n");
    exit(0);
  }
  printf("The elements of the queue
    are: \n");
  for(int i = front; i<=rear; i++)
    printf("%d", CQ[i]);
}
```

Output:

Enter 1 to insert into the circular queue
Enter 2 to delete from circular queue
Enter 3 to display
Enter 0 to exit
Enter your choice:
1
Enter the element to be inserted:
1
Element 1 inserted successfully
Enter your choice:
1
Enter the element to be inserted
2
Element 2 inserted successfully
Enter your choice
3
The elements of the queue are
12

Enter your choice
2
The element deleted is 1.

2. Singly Linked list:

```
#include <stdio.h>
#include <stdlib.h>
Struct Node
{
    int data;
    Struct Node *next;
};
int main()
{
   Struct Node *head = NULL;
   int c=0, a, b;
   while (c1=3)
   {
    printf("\n Enter 1 to insert\n");
    printf("Enter 2 to display\n");
    printf(" Enter 3 to exit ");
    printf("Enter your choice:\n");
    scanf("%d", &c);
    switch (c)
   {
    case 1:
    printf("Enter the value to be inserted:\n");
    scanf("%d", &a);
    int c2;
    printf("Enter 1 to insert at front\n");
    printf(" Enter 2 to insert after a given node\n");
    printf(" Enter 3 to insert at end\n");
    scanf("%d", &c2);
```

```
switch (c2)
{
    case 1:
        insertFront(&head, a);
        break;
    case 2:
        insertEnd(&head, a);
        break;
    case 3:
        printf("Enter the value after which
            to insert \n");
        scanf("%d", &b);
        struct Node *temp = head;
        while (temp != NULL && temp->data != b)
            temp = temp->next;
        if (temp == NULL)
            printf("Element not found in the
            list \n");
        else
        {
            insertMiddle(temp, a);
        }
        break;
        default : printf("invalid choice : \n");
            break;
    }
    case 2:
        display(head); break;
        default: printf("Invalid choice \n");
            break;
    }
}
```

```
    return 0;
}

Void insertFront (struct Node ** head ,
                  int new_data)
{
    struct Node * new_node = (struct Node *)
            malloc (sizeof (struct Node));
  new_node -> data = new_data;
  new_node -> next = (*head);
   (*head) = new_node;
}

Void insertMiddle (struct Node * previous,
                   int new_data)
{
    if (previous == NULL)
    {
      printf ("The previous node entered
        cannot be NULL \n");
      return ;
    }
    struct Node * new_node = (struct Node*)
                malloc (sizeof (struct Node));
   new_node -> data = new_data;
   new_node -> next = previous -> next;
    previous -> next = new_node;
}
Void insertEnd (struct Node **head, int new_
                                    data)
{
  struct Node * new_node = (struct Node*)
                malloc (sizeof (struct Node));
```

```
struct Node *last = *head;
new_node -> data = new_data;
new_node -> next = NULL;
if (*head == NULL)
{
    *head = new_node;
    return;
}
while (last -> next != NULL)
    last = last -> next;
last -> next = new_node;
}

void display (struct Node *node)
{
  printf("The contents of the list
      are: \n");
  while (node != NULL)
  {
    printf(" %d", node -> data);
    node = node -> next;
  }
  printf("\n");
}
```

Output:

Enter 1 to insert:
Enter 2 to display:
Enter 3 to exit

Enter your choice
1
Enter the value to be inserted
1
Enter 1 to insert at the front
Enter 2 to insert after a given node
Enter 3 to insert at the end
Enter your choice
1
The contents of the list are:
1
Enter the value to be inserted:
2
:
Enter your choice:
1
The contents of the list are:
21
Enter the value to be inserted:
3
:
Enter your choice
2
Enter the value after which to insert:
2
The contents of the list are:
231

Leetcode Problem:

```
typedef struct {
        int array [20000];
        int min [20000];
        int top1;
        int top2;
    } MinStack;
MinStack * minStackCreate () {
    MinStack * obj = (MinStack*) malloc
            (sizeof (MinStack));
    obj -> top1 = -1;
    obj -> top2 = -1;
    return obj;
}

Void minStackPush (MinStack * obj, int val)
 {
    obj -> array [++obj -> top1] = val;
    if (obj -> top2 == -1) {
        obj -> min [++obj -> top2] = val;
        return;
    }
  int minTop = obj -> min [obj -> top2];
 if (minTop > val) {
    obj -> min [++obj -> top2] = val;
    return;
 }
 else {
```

```
        obj->min[++obj->top2] = minTop;
    }
}

void minStackPop(MinStack* obj) {
    obj->top1--;
    obj->top2--;
}

int minStackTop(MinStack* obj) {
    return obj->array[obj->top1];
}

int minStackGetMin(MinStack* obj) {
    return obj->min[obj->top2];
}

void minStackFree(MinStack* obj) {
    free(obj);
}
```

8/1/24

Output:
All test cases passed

18.01.24

Week-04

1.
```
#include <stdio.h>
#include <stdlib.h>

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

int main()
{
    Struct Node* head = NULL;
    insertEnd (&head, 1);
    insertEnd (&head, 2);
    insertEnd (&head, 3);
    insertEnd (&head, 4);
    insertEnd (&head, 5);
    insertEnd (&head, 6);
    printf("InHal List :\n");
    display_q (head);
    int c, cl=0;
    while (cl==0)
    {
    printf(" Enter 1 to delete from the
        beginning \n");
    printf(" Enter 2 to delete from the
        end\n");
    printf ("Enter 3 to delete from a
      Specific position \n");
    printf (" Enter 4 to display \n");
    printf (" Enter your choice :\n");
    scanf ("%d", &c);
```

```
switch (c)
{
    case 0 : printf (" Exiting ... \n") ; c1=1; break;
    case 1 : delete_beg (&head); break;
    case 2 : delete_end (&head); break;
    case 3 : delete_mid (&head); break;
    case 4 : display_q (head); break;
    default : printf (" Invalid choice, Enter again
                                    \n"); break;
    }
}
  return 0;
}
Void delete_beg (struct Node ** head)
{
    if (*head == NULL)
    {
    printf (" List is empty ");
    }
    else
    {
        struct Node* ptr = *head;
        *head = (*head) → next;
        free (ptr);
        printf (" Node deleted from beginning \n");
    }
}
Void delete_end (struct Node** head)
{
    struct Node* ptr;
    struct Node* ptr1 = NULL;
    if (*head == NULL)
    {
        printf (" List is empty \n");
    }
```

```
else if ((*head) -> next ==NULL)
{
    free (*head);
    *head = NULL;
    printf(" Deleted the only node
     from the list \n");
}
else
{
    ptr =*head;
    while (ptr ->next != NULL)
    {
        ptr1 = ptr ;
        ptr = ptr -> next;
    }
    ptr1 -> next =NULL;
     free (ptr);
    printf(" Deleted the last node from
     the list \n");
}
}

Void delete_mid( struct Node** head)
{
    struct Node* ptr;
    struct Node* ptr1 = NULL;
    int loc;
    printf(" Enter the location of the
    node to be deleted :\n");
    scanf ("%d", &loc);
    ptr = *head;
    for ( int i =0 ; i<loc; i++)
```

```
    {
        ptr1 = ptr;
        ptr = ptr -> next;
        if (ptr == NULL)
        {
            printf(" less elements than required
              in the list \n");
            return;
        }
    }
    ptr1 -> next = ptr -> next;
    free (ptr);
    printf(" Node deleted from position %d \n", loc);
}

Void display_q (struct Node* head)
{
    struct Node* current = head;
    if (current == NULL)
    {
        printf(" The list is empty \n");
        return;
    }
    else
    {
        printf(" The contents of the list are: \n");
        while (current != NULL)
        {
            printf(" %d", current -> data);
            current = current -> next;
        }
        printf("\n");
    }
}
```

```
Struct Node* createLinkedList(int data)
{
    Struct Node* newNode = (struct Node*)
    malloc (sizeof(struct Node));
    newNode -> data = data;
    newNode -> next = NULL;
    return newNode;
}

Void insertEnd( struct Node** head, int data)
{
    Struct Node* newNode = createLinkedList(data)
    if (*head == NULL)
    {
      (*head = newNode;
    }
    else
    {
      Struct Node* current = *head;
      while (current -> next != NULL)
      {
        current = current -> next;
      }
      current -> next = newNode;
    }
}
```

Output:

Initial list:
The contents of the list are:
123456

Enter 1 to delete from beginning
Enter 2 to delete from end
Enter 3 to delete from specific position
Enter 4 to display
Enter 0 to exit

Enter your choice:
1
Node deleted from the beginning

Enter your choice:
4
The contents of the list are:
2456

Enter your choice:
2
Node deleted from the end

Enter your choice:
4
The contents of the list are:
245

Enter your choice:
3
Enter the location of the node to be deleted:

2
Node deleted from position 2

Enter your choice:
4
The contents of the list are:
25

18/1/24

Reverse Leetcode

```
Void append (struct ListNode** head, int val) {
    Struct ListNode *new_node = (struct
    ListNode *) malloc (sizeof (struct ListNode));
    new_node -> val = val;
    new_node -> next = NULL;
    struct ListNode *prev = *head;
    if (prev == NULL) {
        *head = new_node;
        return;
    }
    while (prev -> next) = NULL)
        prev = prev -> next;
    prev -> next = new node;
}
void mid (struct ListNode *prev, int
        val) {
    Struct ListNode* new_node =
    (Struct ListNode*) malloc (sizeof (
        Struct ListNode));
    new_node -> val = value;
    new_node -> next = (*head);
    *head = new_node;
}
Struct ListNode* reverseBetween
    (Struct ListNode* head, int left,
    int right)
{
```

```
struct ListNode* newhead = NULL;
int i = 1;
struct ListNode* cur = head;
struct ListNode* prev = newhead;
while (cur != NULL)
{
    if (left <= i && right >= i)
    {
        if (prev == NULL) {
            append (&newhead,
            cur->val);
            prev = newhead;
            cur = cur->next;
            i++;
            continue;
        } else if (left == 1) {
            push (&newhead, cur->val);
        }
    else
        mid (prev, cur->val);
    }
    else
    {
        append (&newhead, cur->val);
        prev = (prev == NULL) ? newhead:
            prev->next;
    }
    i++;
    cur = cur->next;
}
return newhead;
}
```

```
Void display (struct ListNode* head) {
    if (head == NULL) {
        printf("Linked list empty \n");
        return;
    }
    printf("Linked list :");
    while (head != NULL) {
        printf("%d", head->val);
        head = head->next;
    }
    printf("\n");
}
```

25/1/24

Week - 5

Q1. Implement Sort, Reverse & concatenation in a Singly Linked list.

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

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

void sort_list (struct Node *head)
{
    struct Node *p, *q;
    int temp;
    for( p=head ; p!=NULL; p=p->next)
    {
        for (q=p->next; q!=NULL; q=q->next
        {
            if(p->data > q->data)
            {
                temp = q->data;
                q->data = p->data;
                p->data = temp;
            }
        }
    }
}
```

```
Void reverse_list (struct Node *head)
{
    struct Node *cur, *pre, *next;
    cur = head;
    pre = NULL;
    while (cur != NULL)
    {
        next = cur -> next;
        cur -> next = pre;
        pre = cur;
        cur = next;
    }
    head = pre;
}

Void concat_list (struct Node *head1,
 struct Node *head2)
{
    struct Node *head3;
    struct Node * temp;
    if (head1 == NULL)
    {
        head3 = head2;
        display_list (head3);
    }
    else if (head2 == NULL)
    {
        head3 = head1;
        display_list (head3);
    }
    else
    {
```

```
    temp = head1;
    head3 = head1;
    while (temp -> next1 = NULL)
        temp = temp -> next;
    temp -> next = head2;
    display_list (head3);
  }
}

Void display_list (struct Node* head)
{
  struct Node* current = head;
  if ( current == NULL)
  {
    printf(" The list is empty :\n");
    while (current != NULL)
    {
      printf("%d", current -> data);
      current = current -> next;
    }
    printf("\n");
  }
}

Struct Node* createLinked list (int data)
{
  struct Node* newNode = (struct
    Node*) malloc (sizeof (struct Node));
  newNode -> data = data;
  newNode -> next = NULL;
  return newNode;
}
```

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

int main()
{
    struct Node *head = NULL;
    insertEnd(&head, 3);
    insertEnd(&head, 1);
    insertEnd(&head, 4);
    insertEnd(&head, 6);
    insertEnd(&head, 5);
    struct Node *head1 = NULL;
    insertEnd(&head1, 1);
    insertEnd(&head1, 3);
    insertEnd(&head1, 4);
    Insert End(&head1, 9);
    struct Node *head2 = NULL;
    insertEnd(&head2, 3);
    insert End(&head2, 1);
```

```
insertEnd(&head2, 4);
insertEnd(&head2, 2);
struct Node *head3 = NULL;
printf("Initial list:\n");
display_list(head);
int c, c1=0;
while(c1==0)
{
  printf("\n Enter 1 to sort the linked list\n");
  printf("Enter 2 to reverse the linked list\n");
  printf("Enter 3 to concatenate 2 linked list\n");
  printf("Enter 4 to display\n");
  printf("Enter 0 to exit\n");
  printf("Enter your choice:\n");
  scanf("%d", &c);
  switch(c)
  {
    case 0: printf("Exiting...\n");
      c1=1; break;
    case 1: sort_list(head);
      display_list(head); break;
    case 2: reverse_list(&head);
      display_list(head);
      break;
    case 3: printf("The 2 lists being concatenated are:\n");
      display_list(head1);
      printf("and\n");
      display_list(head2);
      printf("concatenated list:\n");
      concat_list(head1, head2); break;
```

```
case 4: display_list(head);
        break;
default: printf("invalid choice\n");
         break;
```

Output:

```
Initial list
The contents of the list are:
3 1 4 2 6 5

Enter 1 to sort the linked list
Enter 2 to reverse the linked list
Enter 3 to concatenate two linked list
Enter 4 to display
Enter 0 to exit

Enter your choice:
1
The contents of the list are
1 2 3 4 5 6

Enter your choice
2
The contents of the list are:
6 5 4 3 2 1

Enter your choice:
3
The contents of the list are:
1 3 4 2
and
The contents of the list are:
5 1 4 2
```

Concatenated list :
The contents of the list are
13423142.

2 Stack and Queue operations using singly linked list.

```
#include <stdio.h>
#include <stdlib.h>
Struct Node
{
    int data;
    Struct Node *next;
Void push_list (Struct Node **head,
                int new_data)
{
  Struct Node *new_node (Struct Node*)
                malloc (sizeof (Struct Node));
  new_node -> data = new_data;
  new_node -> next = NULL;
  Struct Node *last = *head;
  if ( *head == NULL)
      *head = new_node;
  else
  {
      while (last -> next != NULL)
          last = last -> next;
          last -> next = new_node;
  }
}
```

```
Void pop_listStack (struct Node *head)
{
    if (head == NULL)
    {
        printf("List is empty \n");
        return;
    }
    struct Node *last = *head;
    struct Node *pre;
    while (last -> next != NULL)
    {
        pre = last;
        last = last -> next;
    }
    free (last);
    pre -> next = NULL;
}
void enqueue_list (struct Node **head, int new_data)
{
    struct Node *new_node = (struct (Node*)
            malloc (sizeof (struct Node));
    new_node -> data = new_data
    new_node -> next = NULL;
    struct Node *last = *head;
    if (*head == NULL)
        *head = new_node;
    else
    {
        while (last -> next != NULL)
            last = last -> next;
        last -> next = new_node;
    }
}
```

```
void dequeue_list (struct Node **head)
{
    if (*head == NULL)
    {
        printf(" List is empty \n");
        return;
    }

Struct Node *temp = (*head)->next;
free (*head);
 *head = temp;
}

Void display_liststack (struct Node *head
{
    if (head == NULL)
    {
        printf (" list is empty \n");
        return;
    }
    printf ( " Stack: \n");
    while (head != NULL)
    {
        printf (" %d ", head->data
        head = head->next ;
    }
    printf (" \n");
}
```

```
void display_listQueue (struct Node *head)
{
  if (head == NULL)
  {
    printf(" list is empty \n");
    return;
  }
  printf(" Queue: \n");
  while (head1 = NULL)
  {
    printf ("%d", head -> data);
    head = head -> next;
  }
  printf ("\n");
}

int main()
{
  struct Node *head = NULL;
  struct Node *head1 = NULL;
  int c, c1=0, elem1, elem2;
  while (c1==0)
  {
    printf (" Enter 1 to push \n");
    printf (" Enter 2 to pop \n");
    printf (" Enter 3 to display \n");
    printf (" Enter 4 to enqueue \n");
    printf (" Enter 5 to dequeue \n");
    printf (" Enter 6 to display queue list");
    printf (" Enter 0 to exit ");
    printf (" Enter your choice \n");
    scanf ("%d", &c);
    switch (c) {
```

```
case 0: printf("Exiting...\n"); break;
case 1:
        printf("Enter the value to
          push:\n");
        scanf("%d", &elem1);
        push_list(&head, elem1);
        display_listStack(head);
        break;
case 2:
        pop_listStack(head);
        display_listStack(head);
        break;

case 3: display_listStack(head); break;

case 4: printf("Enter the value to enqueue
        scanf("%d", &elem2);
        enqueue_list(&head1, elem2);
        display_listQueue(head1);
        break;
case 5:
        dequeue_list(&head1);
        display_listQueue(head1);
        break;
case 6: display_listQueue(head1); break;
default: printf("Wrong choice\n");
        break;
    }
}
return 0;
}
```

Output:

Enter 1 to push
Enter 2 to pop
Enter 3 to display
Enter 4 to enqueue
Enter 5 to dequeue
Enter 6 to display Queue list
Enter 0 to exit

(i) Enter your choice:
1
Enter the value to push:
1
Stack:
1

Enter your choice:
1
Enter the value to push:
2
Stack:
12

Enter your choice:
2
Stack:
1

Enter your choice
3
Stack:
1

Enter your choice:
4
Enter the value to enqueue:
2
Queue:
12

Enter your choice:
5
Queue:
2

Enter your choice:
6
Queue:
6

LeetCode [Singly linked list]

```
Void append (struct ListNode **head, int val)
{
    struct ListNode *new_node = (struct ListNode*)
                malloc (sizeof (struct ListNode));

    new_node -> val = val;
    new_node -> next = NULL;
    struct ListNode *prev = *head;
    if ( prev == NULL)
    {
        *head = new_node;
        return;
    }
    while (prev -> next != NULL)
        prev = prev -> next;
```

```
    prev->next = new_node;
}
Void mid (struct ListNode *prev, int val)
{
    struct ListNode *new_node = (struct ListNode*)
                malloc (sizeof (struct ListNode));
    new_node -> val = val;
    new_node -> next = prev -> next;
    prev -> next = new_node;
}

void push ( struct ListNode **head, int value)
{
    struct ListNode * new_node = (struct
                ListNode*)malloc (sizeof (
                struct ListNode));
    new_node -> val = value;
    new_node -> next = (*head);
    *head = new_node;
}

struct ListNode* reverseBetween (struct ListNode *he
                -ad, int left, int
                right)

{
    struct ListNode *newhead = NULL;
    int i = 1;
    struct ListNode *cur = head;
    struct ListNode *prev = newhead;
    while (cur != NULL)
    {
    if (left <= i && right >= i)
    {
```

```
if(prev == NULL)
{
    append(&newhead, cur->val);
    prev = newhead;
    cur = cur->next;
    i++;
    continue;
}
else if(left == 1)
    push(&newhead, cur->val);
else
{
    append(&newhead, cur->val);
    prev = (prev = NULL) ? newhead :
                prev->next;
}
    i++;
    cur = cur->next;
}
return newhead;
}

Void display(struct listNode *head){
    if(head == NULL)
    {
        printf("linked list is empty\n");
        return;
    }
    printf("linked list: ");
    while(head != NULL)
    {
        printf("%d", head->val);
        head = head->next;
    }
```

classmate
Date
Page

```
    printf("\n");
}
```

Output:

All tests are passed!

Sneha
30/1/24

Week-6

Q1. WAP to implement doubly link 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.

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

Struct Node
{
    int data;
    Struct Node *prev;
    Struct Node *next;
};

Struct Node *createNode (int value)
{
    Struct Node* new_node = (Struct Node*)
            malloc (sizeof (Struct Node));
    if (newNode == NULL)
    {
        printf(" Memory allocation failed. \n");
        exit(1);
    }
    new_node -> data = new_data; value;
    new_node -> prev = NULL;
    new_node -> next = NULL;
```

```
    return new_node;
}

void insertLeft(struct Node** head, int value,
                int targetValue)
{
    struct Node* newNode = createNode(value);
    struct Node* current = *head;
    while (current != NULL && current->data !=
                targetValue)
    {
        current = current->next;
    }
    if (current == NULL)
    {
        printf("Node with value %d not found.
                \n", targetValue);
        free(newNode);
        return;
    }
    if (current->prev != NULL)
    {
        current->prev->next = new_node;
        new_node->prev = current->prev;
    }
    else
    {
        *head = new_node;
    }
    new_node->next = current;
    current->prev = new_node;
}
```

```
Void deleteNode (struct Node** head, int value)
{
    struct Node* current = *head;
    while (current != NULL && current -> data !=
                                        value)
    {
        current = current -> next;
    }
    if (current == NULL)
    {
        printf("Node with value %d
            not found.\n", value);
        return;
    }
    if (current -> prev != NULL)
    {
        current -> prev -> next = current -> next;
    }
    else
    {
        *head = current -> next;
    }
    if (current -> next != NULL)
    {
        current -> next -> prev = current -> prev;
    }
Void displayList (struct node *head)
{
    struct Node *current = head;
    while (current != NULL)
    {
        printf("%d", current -> data);
        current = current -> next;
    }
}
```

```
int main()
{
    struct Node *head = NULL;
    int choice, value, targetValue;
    do
    {
    printf("Enter 1 to create doubly linked list\n");
    printf("Enter 2 to insert a new node to the
            left of the node.\n");
    printf("Enter 3 to delete a node based on
            a specific value.\n");
    printf("Enter 4 to display\n");
    printf("Enter 5 to Exit\n");
    printf("Enter your choice:\n");
    scanf("%d", &choice);
    switch (choice){
            case 1: if (head != NULL)
                {
                  printf("Doubly linked list
                  already created.\n"); }
                else {
                printf("Enter the value for the node:");
                scanf("%d", &value);
                head = createNode(value);
                printf("Doubly linked list created
                successfully.\n"); }
                break;
            Case 2: if (head == NULL) {
            printf("List is empty\n"); }
             else {
            printf("Enter the value to insert:");
            scanf("%d", &value);
            printf("Enter the value of the
            node to the left of which to insert:");
```

```
scanf("%d", &targetValue);
insertLeft(&head, value, targetValue);
} break;
Case 3: if (head == NULL) {
printf("List is empty"); }
else {
printf("Enter the value to delete:");
scanf("%d", &value);
deleteNode(&head, value); }
break;
Case 4: if (head == NULL) {
printf("List is empty.\n"); } {
else {
printf("The contents of list are:\n");
display list(head); } break;
Case 5: printf("Exiting...\n"); break;
default: printf("Invalid choice. Enter again");
break;
}
} while (choice != 5);
return 0;
}
```

Output:

Enter 1 to insert create a doubly linked list
Enter 2 to insert a node to the left of a node
Enter 3 delete a node based on specific value.
Enter 4 to display
Enter 5 to exit

Enter your choice:
1
Enter the value for the node: 3
Doubly linked list created successfully.

Enter your choice:
2
Enter the value to insert:
2
Enter the value of node to the left of which to insert:
3
Enter 4
Enter your choice:
3.
Enter the value to delete:
2.

Enter your choice:
4
The contents of the list are:
13.

Enter your choice:
5
Exiting...

Sneha
11/2/24

leetcode:

Singly Linked List:

```
Struct ListNode ** splitListToParts (struct
                    ListNode** head,
    int k, int * returnSize) {
  struct ListNode * current = head;
   int length = 0;
   while (current) {
        length ++;
```

```
current = current -> next ; }
int part_size = length /k
int extra_nodes = length % k
struct ListNode ** result =
    (struct ListNode**) malloc
    (k* sizeof ( struct ListNode *));
current = head;
for ( int i=0 ; i<k ; i++) {
    struct ListNode * part_head =
        current;
    int part_length = part_size +
        (i< extranodes ? 1:0 );
    for ( int j=0 ; j< part_length -1 &&
        current ; j++ ) {
    current -> next = NULL; }
    result [i] = part_head ;
    current = next_node ; }
    else {
        result [i] = NULL ; }
    }
*returnSize = k
return result ;
}
```

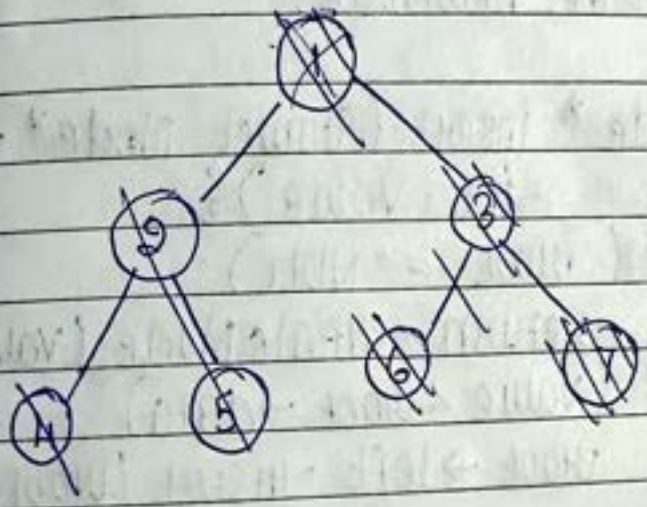
Output:

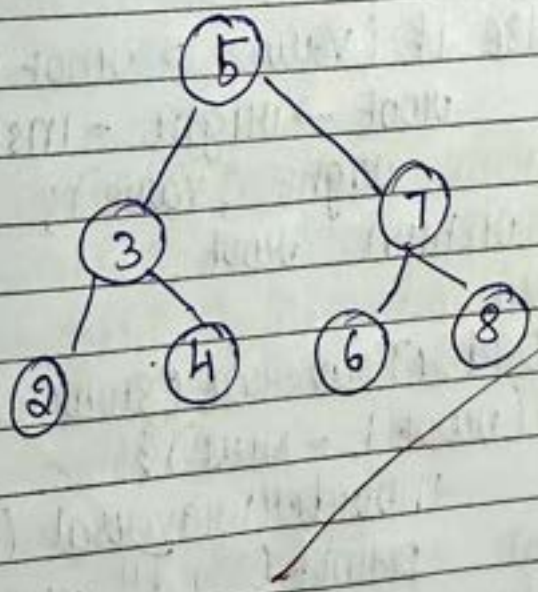
case 1
head = [1, 2, 3]
k = 5
output = [[1], [2], [3], [ ], [ ]]

Case 2 : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
K = 3
Output = [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]

Week - 07

1. a. To construct a binary search tree
   b. To traverse the tree using all the methods i.e, in-order, preorder and postorder
   c. To display the elements in the tree.





```
#Struct <stdio.h>
#Struct <stdlib.h>
Struct Node {
    int data;
    Struct Node * left;
    Struct Node * right;
};
Struct Node * createNode (int value) {
    Struct Node * newNode = (Struct Node *)
    malloc (sizeof( Struct Node));
    newNode -> data = value;
    newNode -> left = newNode -> right = NULL;
    return newNode;
}
Struct Node * insert (Struct Node * root,
                      int value ) {
    if ( root == NULL)
        return createNode (value);
    if (value < root -> data)
        root -> left = insert (root -> left,
                               value);
    else if (value > root -> data)
        root -> right = insert (root ->
                  right , value);
    return root;
}
Void inOrderTraversal (Struct Node * root) {
    if (root != NULL) {
        inOrderTraversal (root -> left);
        printf (" %d", root -> data);
        inOrderTraversal (root -> right);
    }
}
```

```
Void preOrderTraversal (struct Node *root) {
        if (root != NULL) {
            printf("%d", root->data);
            preOrderTraversal (root->left);
            preOrderTraversal (left->right);
        }
    }
Void postOrderTraversal (struct Node * root) {
1       if (root) = NULL) {
            postOrderTraversal (root->left);
            postOrderTraversal (root->right);
            printf("%d", root->data);
        }
    }
Void display (struct Node * root) {
        printf ("Elements in the tree: ");
        inOrderTraversal (root);
        printf ("\n");
    }
int main () {
    struct Node * root = NULL;
    int choice, value;
    do {
        printf(" 1. Insert element ");
        printf(" 2. display element \n");
        printf(" 3. In-order traversal \n");
        printf(" 4. Pre-order traversal \n");
        printf(" 5 Post-order Traversal \n");
        printf(" Exiting \n");
        printf(" Enter your choice: \n");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            printf(" Enter the element to insert:
                \n");
```

```
            scanf("%d", &value);
            root = insert(root, value);
            break;
        case 2: display(root);
            break;
        case 3: printf("In-order traversal: ");
            inOrderTraversal(root);
            printf("\n");
            break;
        case 4: printf("Pre-order traversal: ");
            PreOrderTraversal(root);
            printf("\n");
            break;
        case 5: printf("Post-order traversal: ");
            PostOrderTraversal(root);
            printf("\n");
            break;
        Case 0: printf("Exiting....");
            break;
        default: printf("Invalid choice");
        }
    } while (choice != 0);
    return 0;
}
```

Output:

1. Insert element
2. display element
3. In-order traversal
4. Pre-order traversal
5. Post-order traversal
0. Exit

Enter your choice: 1
Enter the element to insert: 2.

Enter your choice: 2
Elements in the tree: 2 3 4 5, 6 7 8

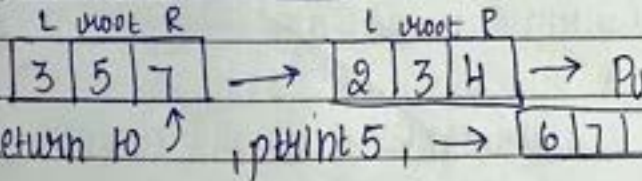
Enter your choice: 3
In-order traversal: 2 3 4 5 6 7 8

Enter your choice: 4
Pre-order traversal: 5 3 2 4 7 6 8

Enter your choice: 5
Post-order traversal: 2 4 3 6 8 7 5

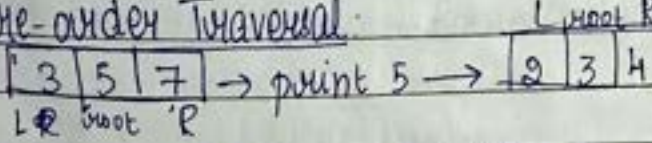
Tracing:

In-order Traversal:



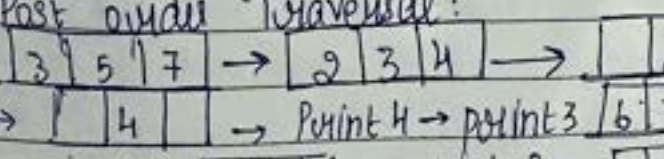
return to ↑, print 5, → print 2, 3, 4 → print 6, 7, 8

Pre-order Traversal:



→ print 5 → print 3 → print 2 → print 4 → print 7 → print 6 → print 8

Post order Traversal:



→ print 2 → print 4 → print 3 → print 6 → print 8 → print 7 → print 5

Leetcode

```
struct ListNode* rotateRight (struct
 ListNode** head , int k) {
 if (head ==NULL ||k ==0){
     return head ;}
 int n=0;
 struct ListNode* last =head;
 while (last -> next != NULL) {
         n++;
         last = last->next; }
 n++;
 last->next = head;
 int rotate = (n - (k%n));
 for (int i=0 ;i <rotate ;i++){
     head = head->next; }
 struct ListNode* ptr1 = head;
 while (ptr1->next != head){
     ptr1 = ptr1->next;
 }
 ptr1 ->next = NULL;
 return head; }
```

Week - 8

1. BFS and DFS method.

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_VERTICES 100
Struct Queue
{
    int front, rear, size;
    unsigned capacity;
    int* array;
};
Struct Queue* createQueue(unsigned capacity)
{
    Struct Queue* queue = (Struct Queue*) malloc(sizeof(Struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = (int*)malloc(queue->capacity* sizeof(int));
    return queue;
}
int isEmpty(Struct Queue* queue)
{
    return (queue->size == 0);
}
Void enqueue(Struct Queue* queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear+1)%queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size+1;
}
```

```
int dequeue (struct Queue * queue) {
        if (isEmpty (queue))
            return -1;
        int item = queue -> array [queue -> front];
        queue -> front = (queue -> front + 1) %
        queue -> capacity;
        queue -> size = queue -> size - 1;
        return item;
}
int isFull (struct Queue* queue)
{
    return (queue -> size == queue -> capacity);
}
Struct Graph {
        int numVertices;
        int** adjMatrix;
};
Struct Graph* createGraph (int numVertices) {
        struct Graph* graph = (struct
    Graph*) malloc (sizeof (struct Graph));
    graph -> numVertices = numVertices;
    graph -> adjMatrix = (int**) malloc
 (numVertices * sizeof (int*));
  for (int i=0 ; i < numVertices ; i++) {
        graph -> adjMatrix [i] = (int*) malloc
        (numVertices * sizeof (int)); }
    for (int i = 0 ; i < numVertices ; i++) {
        for (int j = 0 ; j < numVertices ; 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)
{
    struct Queue*
    queue = createQueue (Max_vertices);
    int visited [Max_vertices];
    for (int i=0 ; i < Max_vertices ; i++)
    {
        visited [i] = 0 ; }
    visited [startVertex] = 1 ;
    enqueue (queue , startVertex);
    while (!isEmpty (queue))
    {
        int currentVertex = dequeue (queue);
        printf("%d", currentVertex);
        for (int i=0 ; i<graph → numVertices ; i++)
        {
            if (graph → adjMatrix [currentVertex]
            [i] == 1 && ! visited [i])
            {
                visited [i] = 1;
                enqueue (queue , i); }
            }
        } free (queue);
    }
Void DFSUtil (struct Graph * graph , int vertex,
int visited []) {
        visited [vertex] = 1;
        printf ("%d" , vertex);
        for (int i=0 ; i<graph → numVertices;
                                    i++) {
```

```
        if (graph->adjMatrix[vertex][i] == 1 &&
            !visited[i]) {
            DFSUtil(graph, i,
                    visited);
        }
    }
}

int isConnected(struct Graph* graph)
{
    int visited[MAX_VERTICES];
    for (int i = 0; i < MAX_VERTICES; i++) {
        visited[i] = 0;
    }
    DFSUtil(graph, 0, visited);
    for (int i = 0; i < graph->numVertices;
         i++) {
        if (!visited[i]) {
            return 0;
        }
    }
    return 1;
}

int main()
{
    struct Graph* graph = createGraph(5);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    printf("BFS Traversal: ");
    BFS(graph, 0);
    if (isConnected(graph)) {
        printf("The graph is connected
                \n");
    }
    else {
        printf("The graph is not
                connected");
    }
    for (int i = 0; i < graph->numVertices;
         i++) {
        free(graph->adjMatrix[i]);
    }
    free(graph->adjMatrix);
    free(graph);
    return 0;
}
```

```
int tail = 0;
int head = 0;

void enqueue (struct node ** queue,
  struct node *root) {
      queue [tail] = root;
      tail ++;
    }
struct node * dequeue ( struct node ** queue)
    {
      struct node *temp = queue [ head ];
      head ++;
      return temp;
    }
int main (){
  int nodes_count, i, temp, h, tc_num,
  index, inc, temp1, temp2;
  scanf ("%d", &nodes_count);
  struct node *root_perm, *root_temp;
  struct node * q [ nodes_count ];
  for (i=0; i <nodes_count; i++) {
      q[i] = NULL;
    }
  i=0, index = 1;
  root_temp = root_perm = create_
  node (1);
  enqueue (q, root_temp);

while ( index <= 2 * nodes_count) {
      root temp = dequeue (q);
      scanf ("%d", &temp1);
      if (temp1 == -1) {
      } else {
        root_temp -> left = create_node
                    (temp1);
```

```
}
enqueue(q, root_temp->left);
if(temp2 == -1){
}
else{
root_temp->right = create_node(
enqueue(q, root_temp->right);
}
index = index + 2;
}
h = height(root_perm);
scanf("%d", &tc_num);
while(tc_num)
{
    uscanf("%d", &inc);
    temp = inc;
    swap_nodes(root_perm, inc, 1, h);
    inorder(root_perm);
    printf("\n");
    tc_num--;
}
return 0;
}
```

Output

All tests are passed.

29-02-24

Lab program-

1. Hash function H:K → L as H(K) = K mod m, and implement hashing technique to map a given key K to the address space L.

```
#include<stdio.h>
#include<stdlib.h>
#define table_size 10
int hashFunction(int key, int m){
        return key % m;
    }
void insert(int hashtable[], int key, int m){
        int i=0;
        int hkey = hashFunction(key,m);
        int index;
        do{
            index = (hkey+i) % m;
            if(hashtable[index]==key){
            printf("key %d found at inde
             %d\n", key, index);
            return;
          }
          i++;
        }while(i<m);
        printf("key id found Unable to
        insert key %d .Table is full.\n", key);
    }
Void search(int hashtable[], int key, int m){
        int i=0;
        int hkey = hashfunction(key,m);
        int index;
        do{
            index = (hkey+i)%m;
```

```
        if (hashtable[index] == key){
            printf("key %d found at index %d\n",
                key, index);
            return;
        }
        i++;
    } while (i<m);
    printf("key %d not found in the table", key);
}
int main(){
    int hashtable[table_size]
    int i;
    for (i = 0; i< table_size; i++){
        hashtable[i] = -1;
    }
insert(hashtable, 1234, table_size);
insert(hashtable, 5678, table_size);
insert(hashtable, 9012, table_size);
insert(hashtable, 2314, table_size);
search(hashtable, 5678, table_size);
search(hashtable, 9999, table_size);
return 0;
}
```

Output:

```
Inserted key 1234 at index 4
Inserted key 5678 at index 8
Inserted key 9012 at index 2
key 5678 found at index 8
key 9999 not found in the table
```