INDEX

DS

NAME: S. Gajanana Nayak STD: CSE-3D SEC: ROLL NO:

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	21/12/23	~~Program~~ LAB 1 - Stack		
2	28/12/23	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 → LL operation & implement.		
→	1/2/24	Leetcode 2) → Reversal of LL		
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	HackerRank → Swap Nodes		

Program 1: (21/12/23)

write a program to swap two numbers using C

```
#include<stdio.h>
void swap(int *, int *);
void main()
{
  int a, b;
  printf("Enter two numbers:\n");
  scanf("%d %d", &a, &b);
  printf("Before swapping, values of a and b are: %d %d", a, b);
  swap(&a, &b);
}
void swap(int *p, int *q)
{
    int temp;
    temp = *b;
    *p = *q;
    *q = temp;
    printf("After swapping, values of a and b are: %d %d", *p, *q)
}
```

Output:

```
Enter two numbers:
4 5
Before swapping, values of a and b are: 4 5
After swapping, values of a and b are: 5 4
```

(21/12/23)

Program 2;

write a program for dynamic memory allocation

```
#include<stdio.h>
void malloc(int);
void calloc(int);
void ~~realloc(int)~~;

Void main()
{
    int n;
    printf("Enter the value of n:\n");
```

```
    scanf("%d", &n);
    MallocEx(n);
    CallocEx(n);
}
void MallocEx(int n)
{
    int *ptr;
    int i;
    int arr[n];
    ptr = (int*) malloc(n * sizeof(int));
    for(i=0; i<n; i++)
    {
        ptr[i] = i+1;
    }
    printf("Malloc dynamic memory allocation \n");
    printf("The elements of the array are: \n");
    for(i=0; i<n; i++)
    {
        printf("%d", ptr[i]);
    }
    printf("\n");
    free(ptr);
}

void CallocEx(int n)
{
    int *ptr;
    int i;
    int arr[n];
    ptr = (int*) calloc(n, sizeof(int));
    for(i=0; i<n; i++)
    {
        ptr[i] = i+1;
    }
    printf("calloc dynamic memory allocation \n");
    printf("The elements of the array are: \n");
    for(i=0; i<n; i++)
    {
        printf("%d", ptr[i]);
    }
```

```
printf("\n");
printf("Realloc dynamic memory allocation\n");
printf("The elements of the array are: \n");
n=10;
ptr=(int *)realloc(ptr, n*sizeof(int));
for(i=5; i<n; i++)
{
  ptr[i]=i+1;
}
for(i=0; i<n; i++)
{
  printf("%d", ptr[i]);
}
free(ptr);
}
```

Output:

Enter the value of n:
5
Malloc dynamic memory allocation
The elements of the array are:
1 2 3 4 5
Calloc dynamic memory allocation
The elements of the array are:
1 2 3 4 5
Realloc dynamic memory allocation
The elements of the array are:
1 2 3 4 5 6 7 8 9 10

(21/12/23)

Program 3:
write a program for stack implementation

```
#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\n 2. 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:\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
  {
    printf ("Elements in the stack are:\n");
    for (int 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:
4
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:
5
Operations on the stack:
1. Push the element
2. Pop the element
3. Show
4. End
Enter the choice
2
Popped element: 5
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:
4
```

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

NLP
29/12/2023

Infix to Postfix (LAB-3)

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

#define MAX 100
char stack[MAX];
int top = -1;

void push(char);
char pop();
int precedence(char);
void infixtopostfix(char infix[], char postfix[]);

void push(char x)
{
    if (top == MAX-1)
    {
        printf("overflow!\n");
        exit(EXIT_FAILURE);
    }
    else
    {
        top++;
        stack[top] = x;
    }
}
char pop()
{
    if (top == -1)
    {
        printf("underflow\n");
    }
    else
    {
        char popped = stack[top];
        top--;
```

```
    return popped;
  }
}
int precedence(char symbol)
{
  if(symbol=='^')
      return 3;
  else if(symbol=='*' || symbol=='/')
      return 2;
  else if(symbol=='+' || symbol=='-')
      return 1;
  else
      return -1;
}
void infix to postfix(char infix[], char postfix[])
{
  int i=0, j=0;
  char symbol, temp;
  push('#');
  while((symbol=infix[i++])!='\0')
  {
    if(symbol=='(')
        push(symbol);
    else if(isalnum(symbol))
        postfix[j++]=symbol;
    else if(symbol==')')
    {
      while(stack[top]!='(')
      {
        postfix[j++]=pop();
      }
      temp=pop();
    }
```

```
        else
        {
            while (precedence(stack[top]) >= precedence(symbol))
            {
                postfix[j++] = pop();
            }
            push(symbol);
        }
    }
    while(stack[top] != '#')
    {
        postfix[j++] = pop();
    }
    postfix[j] = '\0';
}

int main()
{
    char infix[MAX], postfix[MAX];
    printf("Enter a valid parenthesized infix exp:\n");
    scanf("%s", infix);
    infix to postfix(infix, postfix);
    printf("The postfix exp is: %s\n", postfix);
    return 0;
}
```

Output:

Enter a valid parenthesized infix exp:

a * b + c * d - e

The postfix exp is: ab * cd * + e -

ALP
28/12/2022

Postfix evaluation (LAB-3)

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

#define max 20

int stack[20];
int top=-1;

void push (int a)
{
  stack[++top]=a;
}
int pop()
{
  return stack[top--];
}

void main()
{
  char postfix[max];
  printf("Enter postfix expression: ");
  scanf("%s", postfix);
  int res=0, a, b;
  for(int i=0; i<strlen(postfix); i++)
  {
    if(isalnum(postfix[i]))
        push(postfix[i]-'0');
    else
    {
        b= pop();
        a= pop();
```

```
switch (postfix[i])
{
case '+': push(a+b);
          break;
case '-': push(a-b);
          break;
case '*': push(a*b);
          break;
case '/': push(a/b);
          break;
case '^': push(a^b);
          break;
    }
  }
}
res = pop()
printf("%s = %d", postfix, res);
}
```

Output:

Enter postfix expression : 23@*31*+5-

23*31*+5- = 4

Linear Queue (LAB-3)

```
#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");
    ;
  }
  printf("Queue:");
  for(int i = front; i <= rear; i++)
  {
    printf("%d", queue[i]);
    else { front++;
  }
}
```

```
void display() {
if (front == -1) {
    printf("Queue Empty");
    return;
}
printf("Queue:");
for(int i=front; i<=rear;
    i++)
  {
    printf("%d",queue[i]);
  }
}
```

```
void main()
{
 int choice;
 int a;
 while(1)
 {
  printf("\n1. Insert \n2. Delete \n3. Display \n 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 : 7

Queue : 7

1. Insert
2. Delete
3. Display
Choice: 1
Enter an element: 8
Queue: 7 8

1. Insert
2. Delete
3. Display
Choice: 2
Queue: 8

1. Insert
2. Delete
3. Display
Choice: 3
Queue: 8

Circular Queue (LAB-3)

```
#include <stdio.h>
#define size 5
int queue[size];
int front = -1;
int rear = -1;

void enqueue (int a)
{
  if ((front == rear + 1) || (front == 0 && rear == size - 1))
  {
    printf ("Queue overflow \n");
    return;
  }
```

Circular Queue (LAB-7)

```
#include <stdio.h>
#define size 5
int queue[size];
int front = -1;
int rear = -1;

void enqueue (int a)
{
   if ((front == rear + 1) || (front == 0 && rear == size - 1))
   {
      printf ("Queue overflow \n");
      return;
   }
```

```
else
{
    if (front == -1)
        front = 0;
    rear = (rear+1) % size;
    queue[rear] = a;
}
}
void dequeue()
{
  if (front == -1)
  { printf("Queue Empty\n");
  }
  else
  {
    int a = queue[front];
    if (front == rear)
    {
      front = -1;
      rear = -1;
    }
    else
    {
      front = (front+1) % size;
    }
    printf("Deleted element = %d\n", a);
    return (a);
  }
}
void display()
{
    if (front == -1)
    {
```

```
        printf("Queue Empty\n");
        return;
    }
    else
    {
        int i;
        printf("\n Front = %d", front);
        printf("\n Items = ");
        for(i=front; i!=rear; i=(i+1)%size)
        {
            printf("%d", queue[i]);
        }
        printf("%d", queue[i]);
        printf("\n Rear = %d", rear);
    }
}
void main()
{
    int choice;
    int a;
    while(1)
    {
        printf("\n 1. Insert \n 2. Delete \n 3. Display \n Choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1: printf("Enter an element: ");
                    scanf("%d", &a);
                    enqueue(a);
                    display();
                    break;
            case 2: dequeue();
                    display();
                    break;
            case 3: display();
                    break;
        }
    }
```

Output:

1. Insert
2. Delete
3. Display
Choice: 1
Enter an element: 2

Front = 0
Items = 2
Rear = 0

1. Insert
2. Delete
3. Display
Choice: 1
Enter an element: 3

Front = 0
Items = 23
Rear = 1

1. Insert
2. Delete
3. Display
Choice: 1
Enter an element: 4

Front = 0
Items = 234
Rear = 2

1. Insert
2. Delete
3. Display
Choice: 2
Deleted element = 2

Front = 1
Items = 34
Rear = 2

LAB - 4 (Insertion of element at every pos in a Linked List)

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

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

Struct Node *createNode (int value)
{
  struct Node *newNode = (struct Node*) malloc (sizeof (struct Node));
  if (newNode == NULL)
  {
    printf ("Memory allocation failed \n");
    exit (1);
  }
  newNode → data = value;
  newNode → next = NULL;
  return newNode;
};

void displayList (struct Node *head)
{
  if (head == NULL)
  {
    printf ("List is empty \n");
    return;
  }
  struct Node * temp = head;
  printf ("Linked list : \n");
  while (temp != NULL)
  {
    printf ("%d → ", temp → data);
    temp = temp → next;
  }
```

```
        printf("NULL\n");
    }
struct Node *insertatbeginning(struct Node *head, int value)
{
    struct Node *newNode = createNode(value);
    newNode->next = head;
    return newNode;
};
void insertatend(struct Node *head, int value)
{
    struct Node *newNode = createNode(value);
    struct Node *temp = head;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = newNode;
}
void insertatposition(struct Node *head, int position, int value
{
    struct Node *newNode = createNode(value);
    struct Node *temp = head;
    int count = 1;
    while (temp != NULL && count < position - 1)
    {
        temp = temp->next;
        count++;
    }
    if (temp == NULL)
    {
        printf("Position out of range\n");
        free(newNode);
    return;
```

```
    newNode->next = temp->next;
    temp->next = newNode;
}

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

    head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);

    printf("Initial:\n");
    displayList(head);

    head = insertAtBeginning(head, 0);
    printf("After insertion at the beginning:\n");
    displayList(head);

    insertAtPosition(head, 3, 10);
    printf("After insertion at position 3:\n");
    displayList(head);

    ~~insertAtPosition(2)~~
    insertAtEnd(head, 20);
    printf("After insertion at the end:\n");
    displayList(head);

    return 0;
}
```

Output:

Initial:

Linked List:

1 → 2 → 3 → NULL

After insertion at the beginning:

Linked List:

0 → 1 → 2 → 3 → NULL

After insertion at position 3:

Linked list:

0 → 1 → 10 → 2 → 3 → NULL

After insertion at the end:

Linked list:

0 → 1 → 10 → 2 → 3 → 20 → NULL

Executed
11/1/24

LAB-5 (Deletion of elements in a Linked List) (18/1/24).

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

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

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

Smitha
18/1/24

```
void deleteFirst(struct Node **head){
    if(*head != NULL){
        struct Node *temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}

void deleteElement(struct Node **head, int value){
    struct Node *current = *head;
    *head = (*head)->next;
    free(temp);

    struct Node *current = *head;
    struct Node *prev = NULL;

    while(current != NULL && current->data != value){
        prev = current;
        current = current->next;
    }
    if(current == NULL){
        return;
    }
    if(prev == NULL){
        *head = current->next;
    }
    else{
        prev->next = current->next;
    }
    free(current);
}
```

```
void deleteLast(struct Node **head) {
    if (*head == NULL) {
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;

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

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

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

```
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);

    printf("Initial Linked List: ");
    displayList(head);

    deleteFirst(&head);
    printf("\nAfter deleting the first element: ");
    displayList(head);

    deleteElement(&head, 2);
    printf("\nAfter deleting specified element(2): ");
    displayList(head);

    deleteLast(&head);
    printf("\nAfter deleting the last element: ");
    displayList(head);

    return 0;
}
```

Output :

Initial Linked List : 1 → 2 → 3 → NULL

After deleting the first element : 2 → 3 → NULL

After deleting specified element(2) : 3 → NULL

After deleting the last element : NULL

Leetcode pgm

Lab 11/1/24 - pgm completed on 19/1/24

(Minstack Problem) - 18/1/24 - NP 19/1/24

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

typedef struct
{
    int value;
    int main;
} stackNode;

typedef struct
{
  stackNode *array;
  int capacity;
  int top;
} minstack;

MinStack * minStackCreate()
{
  MinStack * stack = (MinStack *) malloc (sizeof (MinStack));
  Stack → capacity = 10;
  Stack → array = (StackNode *) malloc (stack → capacity *
                  sizeof (StackNode));

  Stack → top = -1;
  return stack;
}
```

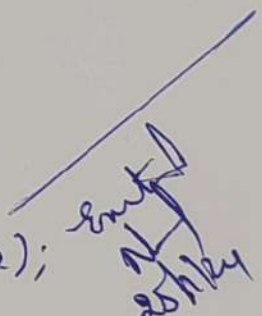
```
void minStackPush (MinStack * obj , int val)
{
    if (obj -> top == obj -> capacity - 1)
    {
        obj -> capacity *= 2;
        obj -> array = (StackNode *) realloc (obj -> array,
                        obj -> capacity * sizeof (StackNode));
    }
    StackNode newNode;
    newNode.value = val;
    newNode.min = (obj -> top == -1) ? val : (val < obj ->
                    array [obj -> top].min) ? val : obj ->
                    array [obj -> top].min;

    obj -> array [++(obj -> top)] = newNode;
}

void minStackPop (MinStack * obj)
{
    if (obj -> top != -1)
    {
        obj -> top --;
    }
}

int minStackTop (MinStack * obj)
{
    if (obj -> top != -1)
    {
        return obj -> array [obj -> top].value;
    }
    return -1;
}
```

```
int minStackGetMin (MinStack * obj)
{
    if (obj->top != -1)
    {
        return obj->array[obj->top].min;
    }
    return -1;
}

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

LAB-6: (25/1/24)

a) Linked List operations:

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

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

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

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

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 *reverseList (Node *head)
{
  Node *prev = NULL;
  Node *current = head;
  Node *nextNode = NULL;

  while (current != NULL)
  {
    nextNode = current -> next;
    current -> next = prev;
```

```
        prev = current;
        current = nextNode;
    }
    return prev;
}

Node *concatenateLists (Node *list1, Node *list2)
{
    if (list1 == NULL)
        return list2;

    Node *temp = list1;
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    temp->next = list2;

    return list1;
}

int main()
{
    Node *list1 = createNode(3);
    list1->next = createNode(1);
    list1->next->next = createNode(4);

    Node *list2 = createNode(2);
    list2->next = createNode(5);

    printf("Original list 1: ");
    displayList(list1);

    printf("Original list 2: ");
    displayList(list2);
```

```
list1 = sortList(list1);
printf("Sorted List1:");
displayList(list1);

list1 = reverseList(list1);
printf("Reversed List1:");
displayList(list1);

Node *concatenated = concatenatedLists(list1, list2);
printf("Concatenated List:");
displayList(concatenated);

return 0;
}
```

output:

original list 1: 3 → 1 → 4 → NULL
original list 2: 2 → 5 → NULL
Sorted list 1 : 1 → 3 → 4 → NULL
Reversed list 1 : 4 → 3 → 1 → NULL
Concatenated List : 4 → 3 → 1 → 2 → 5 → NULL

b) Implement single LL for stack operations (25/1/24)

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

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

```
typedef struct Node Node;
Node *createNode (int value)
{
  Node *newNode =(Node*)malloc(sizeof(Node));
  newNode->data = value;
  newNode->next = NULL;
  return newNode;
}
void displayList(Node *head)
{ while (head!=NULL)
  {
    printf("%d->", head->data);
    head = head->next;
  }
  printf("NULL\n");
}
typedef struct
{
  Node *top;
}LinkedList;
void push(LinkedList *stack, int value)
{
  Node *newNode = createNode(value);
  newNode->next = stack->top;
  stack->top = newNode;
}
int pop(LinkedList *stack)
{
  if(stack->top == NULL)
  {
```

Ended 25/4/24

```
        printf("Stack is empty\n");
        return -1;
    }
    int poppedValue = stack->top->data;
    Node *temp = stack->top;
    stack->top = stack->top->next;
    free(temp);
    return poppedValue;
}

int main()
{
    LinkedList stack;
    stack.top = NULL;

    printf("Stack operations:\n");
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    displayList(stack.top);
    printf("Popped from stack: %d\n", pop(&stack));
    printf("Popped from stack: %d\n", pop(&stack));
    displayList(stack.top);

    return 0;
}
```

Output:

Stack operations:

30 → 20 → 10 → NULL

Popped from stack: 30

Popped from stack: 20

10 → NULL

Implement single LL for queue operations

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
  int data;
  struct Node *next;
};
typedef struct Node Node;
Node *createNode(int value)
{
  Node * newNode = (Node*) malloc (sizeof (Node));
  newNode -> data = value;
  newNode -> next = NULL;
  return newNode;
}
void displayList (Node *head)
{
  while (head != NULL)
  {
    printf ("%d ->", head -> data);
    head = head -> next;
  }
  printf ("NULL\n");
}
typedef struct
{
  Node * front;
  Node * rear;
} LinkedList;

void enqueue (LinkedList *queue, int value)
{
  Node * newNode = createNode (value);
```

Emptied
25/11/24

```
    if (queue->front == NULL)
    {
        queue->front = newNode;
        queue->rear = newNode;
    }
    else
    {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}
int dequeue(LinkedList *queue)
{
    if (queue->front == NULL)
    {
        printf("Queue is empty\n");
        return -1;
    }
    int dequeuedvalue = queue->front->data;
    Node *temp = queue->front;
    queue->front = queue->front->next;
    free(temp);
    return dequeuedvalue;
}

int main()
{
    LinkedList queue;
    queue.front = NULL;
    queue.rear = NULL;

    printf("\n Queue operations:\n");
```

```
    enqueue (&queue, 40);
    enqueue (&queue, 50);
    enqueue (&queue, 60);
    displayList (queue.front);
    printf("Dequeued from queue: %d\n", dequeue(&queue));
    printf("Dequeued from queue: %d\n", dequeue(&queue));

    displayList (queue.front);

    return 0;
}
```

Output:

Queue operations:

40 → 50 → 60 → NULL

Popped from queue: 40

Popped from queue: 50

60 → NULL

LeetCode Problem (Reversal of Linked List)

18/1/24

```
struct ListNode* reverseBetween(struct ListNode* head,
int left, int right)
{
    if (head == NULL)
        return NULL;
    if (left == right)
        return head;
```

```
struct ListNode * prev = NULL;
struct ListNode * curr = head;

int index = 1;
while (index < left)
{
    prev = curr;
    curr = curr->next;
    index++;
}
struct ListNode * leftMinusOneNode = prev;

struct ListNode * leftNode = curr;
struct ListNode * next = NULL;

while (left <= right)
{
    next = curr->next;
    curr->next = prev;

    prev = curr;
    curr = next;
    left++;

}
if (leftMinusOneNode == NULL)
        head = prev;

else
    leftMinusOneNode->next = prev;

leftNode->next = curr;

return head;
}
```

LAB - 7 : (Doubly Linked List) (11/2/24)

```
#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 *forget,
                       int data)

{
  struct Node *newNode = createNode(data);
  if (forget -> prev != NULL)
  {  forget -> prev -> next = newNode;
     newNode -> prev = forget -> 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

Leetcode Problem (Split Linked List in Parts) (1/2/24)

```
int getLength(struct ListNode* head)
{
  int length = 0;
  while (head != NULL)
  {
    length++;
    head = head->next;
  }
  return length;
}

struct ListNode** splitListToParts(struct ListNode* head,
                                   int k, int *returnSize)
{
  int length = getLength(head);
  int partsize = length / k;
  int remainder = length % k;
  struct ListNode** result = (struct ListNode**) malloc
                             (k * sizeof(struct ListNode*));
  *returnSize = k;
```

1/2/2024

```
for (int i = 0 ; i < k ; i++)
{
    int currentPartSize = partSize + (i < remainder ?
                                      1 : 0);
    if (currentPartSize == 0)
    {
        result[i] = NULL;
    }
    else
    {
        result[i] = head;
        for (int j = 0 ; j < currentPartSize - 1 ; j++)
        {
            head = head -> next;
        }
        struct ListNode* temp = head -> next;
        head -> next = NULL;
        head = temp;
    }
}
return result;
}
```

LAB-(8) :- Binary Search Tree

15/1/24

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

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

struct node *newNode(int data)
{
    struct node *node = (struct node*) malloc
                        (sizeof(struct node));

    node->data = data;
    node->left = node->right = NULL;
    return node;
}

struct node *insert(struct node *root, int data)
{
    if (root == NULL)
        return newNode(data);
    if (data <= root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);
    return root;
}
```

15/2/2024

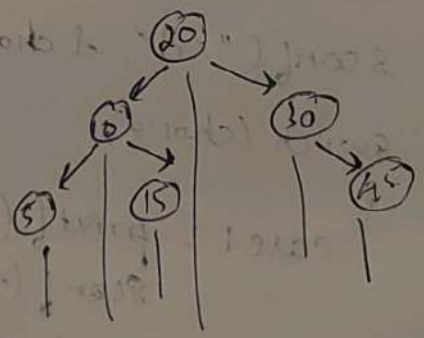
```
void inorder (struct node *temp)
{
    if (temp == NULL)
        return;
    inorder (temp -> left);
    printf ("%d", temp -> data);
    inorder (temp -> right);
}

void preorder (struct node * temp)
{
    if (temp == NULL)
        return;
    printf ("%d", temp -> data);
    preorder (temp -> left);
    preorder (temp -> right);
}

void postorder (struct node * temp)
{
    if (temp == NULL)
        return;
    postorder (temp -> left);
    postorder (temp -> right);
    printf ("%d", temp -> data);
}
```

```
int main()
{
  struct node * root = NULL;
  int data, choice;

  do
  {
    printf("Enter your choice: \n 1. Insert \n 2. Print
    Inorder \n 3. Print Preorder \n 4. Print Postorder \n
    5. Exit \n");
    scanf("%d", &choice);
    switch (choice)
    {
      case 1: printf("Enter value to be inserted: ");
              scanf("%d", &data);
              root = insert (root, data);
              break;
      case 2: printf("Inorder traversal of binary tree
                      is \n");
              inorder (root);
              printf("\n");
              break;
```

```
case 3: printf("Preorder traversal of binary tree is \n");
        preorder(root);
        printf("\n");
        break;
case 4: printf("Postorder traversal of binary tree is \n");
        postorder(root);
        printf("\n");
        break;

case 5 : printf("Exiting...");
         break;

default: printf("Invalid choice");
}

}while(choice!=5);
return 0;
}
```

output:

```
Enter your choice:
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit
1
Enter value to be inserted : 20
Enter your choice:
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit
```



1
Enter value to be inserted : 10
Enter your choice :
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit

1
Enter value to be inserted : 30
Enter your choice :
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit

1
Enter value to be inserted : 5
Enter your choice :
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit

1
Enter value to be inserted : 15
Enter your choice :
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit

1
Enter value to be inserted : 45

Enter your choice:
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit

2
Inorder traversal of binary tree is:
5 10 15 20 30 45

Enter your choice:
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit
3
Preorder traversal of binary tree is:
20 10 5 15 30 45

Enter your choice:
1. Insert
2. Print Inorder
3. Print Preorder
4. Print Postorder
5. Exit
4
Postorder traversal of binary tree is:
5 15 10 45 30 20

Leetcode → Rotate List

```
struct ListNode* rotateRight(struct ListNode *head, int k)
{
    if(head == NULL || head->next == NULL || k==0)
        return head;

    int len = 1;
    struct ListNode *tail = head;
    while (tail->next != NULL)
    {
        tail = tail->next;
        len++;
    }
    k = k % len;
    if(k==0)
        return head;
    struct ListNode *p = head;
    for(int i=0; i< len-k-1; i++)
    {
        p = p->next;
    }
    tail->next = head;
    head = p->next;
    p->next = NULL;
    return head;
}
```

LAB - (9) : (22/2/24)

BFS

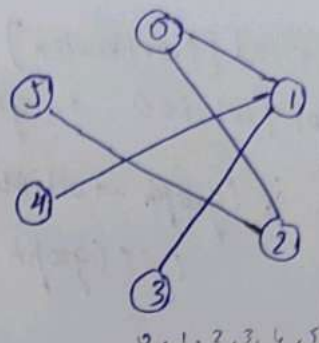
```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
struct Queue
{
   int items [MAX_SIZE];
   int front;
   int rear;
};
struct Queue * createQueue()
{
  struct Queue* queue = (struct Queue *) malloc (sizeof (struct Queue));
  queue → front = -1;
  queue → rear = -1;
  return queue;
}
int 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 ("\n Queue 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]==0)
        {
            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;
}
```

Output:

```
Enter the number of vertices: 6
Enter the number of edges: 5
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): 1 4
Enter edge 5 (source destination): 2 5
Enter the starting vertex for BFS: 0
Breadth First Search Traversal: 0 1 2 3 4 5
```



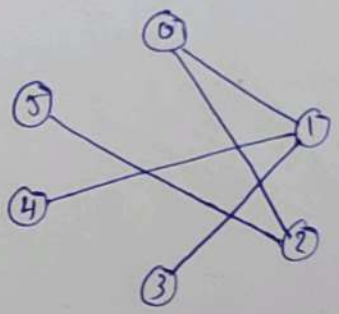
DFS

```
#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 = 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 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 && 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 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);
    }
    if(isConnected(graph))
        printf("The graph is connected\n");
    else
        printf("The graph is not connected\n");
    return 0;
}
```

Output:

```
Enter the number of vertices : 6
Enter the number of edges : 5
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): 1 4
Enter edge 5 (source destination): 2 5
The graph is connected
```



Sf:
22/2/24

LAB (10) : (Hashing) (29|2|24)

```
#include <stdio.h>
#define TABLE_SIZE 10
int hashFunction(int key)
{
  return key % TABLE_SIZE;
}
void insertValue(int hashTable[], int key)
{
  int i=0;
  int hkey = hashFunction(key);
  int index;
  do
  {
    index = (hkey + i) % TABLE_SIZE;
    if(hashTable[index] == -1)
    {
      hashTable[index] = key;
      printf("Inserted key %d at index %d\n", key, index);
      return;
    }
    i++;
  }while(i < TABLE_SIZE);
  printf("Unable to insert key %d. Hash table is full.\n", key);
}
```

```
int searchValue (int hashTable[], int key)
{
    int i=0;
    int hkey = hashFunction(key);
    int index;

    do
    {
        index = (hkey+i) % TABLE_SIZE;
        if (hashTable[index] == key)
        {
            printf("key %d found at index %d\n", key, index);
            return index;
        }
        i++;
    }while (i<TABLE_SIZE);
    printf("key %d not found in hash table\n", key);
    return -1;
}

int main()
{
    int hashTable[TABLE_SIZE];
    for (int i=0; i<TABLE_SIZE; i++)
    {
        hashTable[i] = -1;
    }
    insertValue (hashTable, 1234);
    insertValue (hashTable, 5678);
    insertValue (hashTable, 9876);

    searchValue (hashTable, 5678);
    searchValue (hashTable, 1111);

    return 0;
}
```

Output: inserted key 1234 at index 4
inserted key 5678 at index 8
inserted key 9876 at index 6
key 5678 not found in hash table
key 1111 not found in hash table

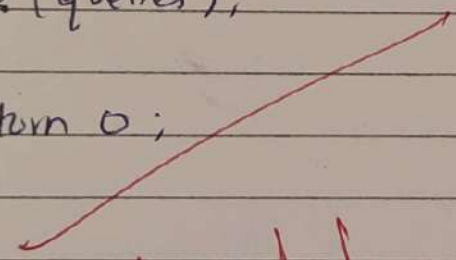
HACKERANK program (Swap Nodes)

```
#include<assert.h>
#include<stdbool.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct Node
{ int data;
  struct Node *left;
  struct Node *right;
}Node;

Node *createNode(int data)
{ Node *newNode = (Node*)malloc(sizeof(Node));
  newNode->data = data;
  newNode->left = NULL;
  newNode->right = NULL;
  return newNode;
}
void inorderTraversal(Node* root, int *result, int *index)
{ if(root==NULL) return;
  inorderTraversal(root->left, result, index);
  result[(*index)++] = root->data;
  inorderTraversal(root->right, result, index);
}
void swapAtLevel(Node *root, int k, int level)
{ if(root==NULL) return;
  if((level % k==0){
      Node *temp = root->left;
      root->left = root->right;
      root->right = temp;
  }
  swapAtLevel(root->left, k, level+1);
  swapAtLevel(root->right, k, level+1);
}
int **swapNodes(int indexes_rows, int index_columns, int **indexes,
int queries_count, int *queries, int *result_rows, int *result_columns)
{
  Node **nodes = (Node**) malloc((indexes_rows+1)*sizeof(Node*));
  for(int i=1; i<=indexes_rows; i++){
      nodes[i] = createNode(i);
  }
```

```
for(int i=0 ; i<indexes_rows ; i++) {
    int leftIndex = indexes[i][0];
    int rightIndex = indexes[i][1];
    if(leftIndex != -1) nodes[i+1] -> left = nodes[leftIndex];
    if(rightIndex != -1) nodes[i+1] -> right = nodes[rightIndex];
}
int ** result = (int **) malloc (queries_count * sizeof(int *));
*result_rows = queries_count;
*result_columns = indexes_rows;
for(int i=0 ; i<queries_count; i++) {
    swapAtLevel(nodes[1], queries[i], 1);
    int *traversalResult = (int *) malloc (indexes_rows * sizeof(int));
    int index = 0;
    inOrderTraversal(nodes[1], traversalResult, &index);
    result[i] = traversalResult;
}
free(nodes);
return result;
}
int main() {
int n;
scanf("%d", &n);
int ** indexes = malloc(n * sizeof(int*));
for(int i=0 ; i<n ; i++) {
    indexes[i] = malloc(2 * sizeof(int));
    scanf("%d %d", &indexes[i][0], &indexes[i][1]);
}
int queries_count;
scanf("%d", &queries_count);
int * queries = malloc(queries_count * sizeof(int));
for(int i=0; i<queries_count; i++) {
    scanf("%d", &queries[i]);
}
int resultRows;
int resultColumns;
int ** result = swapNodes(n, 2, indexes, queries_count, queries, &resultRows,
                          &resultColumns);
for(int i=0; i<resultRows; i++) {
    for(int j=0; j<resultColumns; j++) {
        printf("%d ", result[i][j]);
    }
    printf("\n");
    free(result[i]);
}
}
```

8/5
29/2/24

```
free(result);
for (int i=0; i<n; i++)
{
    free(indexes[i]);
}
free(indexes);
free(queries);

return 0;

}
```



29/12/24