VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

PRIYANSHU KUMAR (1BM22CS210)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Dec 2023- March 2024

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

This is to certify that the Lab work entitled **"DATA STRUCTURES"** carried out by **Priyanshu Kumar (1BM22CS210)**, who is a bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of Data structures Lab - **(23CS3PCDST) work** prescribed for the said degree.

Prof. Sneha S Bagalkot
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course outcomes:

CO1	Apply the concept of linear and nonlinear data structures.
CO2	Analyze data structure operations for a given problem
CO3	Design and develop solutions using the operations of linear and nonlinear data structure for a given specification.
CO4	Conduct practical experiments for demonstrating the operations of different data structures.

LAB 1

Program 1: Swapping Using Pointers

```c
#include <stdio.h>

void main() {
   int a, b;

   printf("Name: Priyanshu Kumar USN: 1BM22CS210\n");

   printf("Enter values of a & b:\n");
   scanf("%d %d", &a, &b);

   printf("Before swapping:\n");
   printf("a = %d & b = %d\n", a, b);

   swap(&a, &b);
   printf("\nAfter swapping:\n");
   printf("a = %d & b = %d\n", a, b);
}

void swap(int *p, int *q) {
   int temp = *p;
   *p = *q;
   *q = temp;
}
```

```
Name: Priyanshu Kumar USN: 1BM22CS210
Enter values of a & b:
3 5
Before swapping:
a = 3 & b = 5

After swapping:
a = 5 & b = 3
```

Program 2: Dynamic Memory Allocation

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

int main() {
   int *arr1;
   int size1 = 5;
   arr1 = (int*)malloc(size1 * sizeof(int));
   int i;
```

```c
if (arr1 == NULL) {
    printf("Memory allocation failed.\n");
    return 1;
}

printf("Using malloc:\n");
for (i = 0; i < size1; i++) {
    arr1[i] = i;
    printf("%d ", arr1[i]);
}
printf("\n");
free(arr1);

int size2 = 5;
int *arr2;

arr2 = (int*)calloc(size2, sizeof(int));

if (arr2 == NULL) {
    printf("Memory allocation failed.\n");
    return 1;
}

printf("\nUsing calloc:\n");
for (i = 0; i < size2; i++) {
    arr2[i] = i * 3;
    printf("%d ", arr2[i]);
}
printf("\n");


int new_size = 10;
arr2 = (int*)realloc(arr2, new_size * sizeof(int));

if (arr2 == NULL) {
    printf("Memory reallocation failed.\n");
    return 1;
}

printf("\nUsing realloc:\n");
for (i = 0; i < new_size; i++) {
    arr2[i] = i * 4;
    printf("%d ", arr2[i]);
}
```

```
    printf("\n");

    free(arr2);

    return 0;
}
```

```
Using malloc:
0 1 2 3 4

Using calloc:
0 3 6 9 12

Using realloc:
0 4 8 12 16 20 24 28 32 36
```

Program 3: Stack Implementation

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

int stack[10];
int count = -1;

void push(int);
void pop();
void display();

int main() {
    for (;;) {
        printf("\n1. Push value to stack\n");
        printf("2. Pop value from stack\n");
        printf("3. Display values\n");
        printf("4. Exit\n\n");

        int t;
        printf("Enter your choice: ");
        scanf("%d", &t);
        printf("\n");

        if (t == 1) {
            int v;
            printf("Enter value to push: ");
            scanf("%d", &v);
            push(v);
        }
        else if (t == 2) {
```

```c
            pop();

        }
        else if (t == 3) {
            display();
        }
        else {
            exit(0);
        }
    }
}

void push(int e) {
    if (count < 10) {
        stack[++count] = e;
    } else {
        printf("Stack is full!\n");
    }
}

void pop() {
    if (count >= 0) {
        stack[count--] = 0;
    } else {
        printf("Stack is empty!\n");
    }
}

void display() {
    if (count == -1)
        return;
    int i;
    printf("The values are (top to bottom):\n");
    for (int i = count; i >= 0; i--) {
        printf("%d\n", stack[i]);
    }
    printf("\n");
}
```

```
1. Push value to stack
2. Pop value from stack
3. Display values
4. Exit

Enter your choice: 1
Enter value to push: 10
Enter your choice: 1
Enter value to push: 15
Enter your choice: 1
Enter value to push: 30
Enter your choice: 3
The values are (top to bottom):
30
15
10

Enter your choice: 2
Enter your choice: 3
The values are (top to bottom):
15
10
```

LAB 2

Program 1: Infix to Postfix Conversion

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

#define MAX_SIZE 100

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

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

void infixToPostfix(char infix[], char postfix[]) {
   char stack[MAX_SIZE];
   int top = -1;
   int i, j;

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

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

```
            stack[++top] = ch;
        }
    }

    while (top >= 0) {
        postfix[j++] = stack[top--];
    }
    postfix[j] = '\0';
}

int main() {
    char infix[MAX_SIZE];
    char postfix[MAX_SIZE];

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

    infixToPostfix(infix, postfix);

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

    return 0;
}
```

```
Enter the infix expression: 2+(4+2)*3
Postfix expression: 242+3*+
```

Program 2: Postfix Evaluation

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

#define MAX_SIZE 30

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

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

```c
int pop() {
    if (top < 0) {
        printf("Stack underflow\n");
        exit(1);
    }
    return stack[top--];
}

int evaluate(char expression[]) {
    int i = 0;
    char symbol;
    int op1, op2, result;

    while ((symbol = expression[i++]) != '\0') {
        if (isdigit(symbol)) {
            push(symbol - '0');
        } else if (symbol == '+' || symbol == '-' || symbol == '*' || symbol == '/')
{
            op2 = pop();
            op1 = pop();
            switch (symbol) {
                case '+':
                    result = op1 + op2;
                    break;
                case '-':
                    result = op1 - op2;
                    break;
                case '*':
                    result = op1 * op2;
                    break;
                case '/':
                    result = op1 / op2;
                    break;
            }
            push(result);
        }
    }
    return pop();
}

int main() {
    char expression[] = "567+*8-";
    int result = evaluate(expression);
    printf("Result = %d\n", result);
    return 0;
```

```
}
```

```
Result = 57
```

LAB 3

Program 1: Queue Implementation

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

#define MAX_SIZE 20

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

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

int main() {
    dequeue();
    enqueue(1);
    enqueue(3);
    enqueue(4);
    enqueue(5);
    enqueue(6);
    display();
    dequeue();
    display();
    return 0;
}

void enqueue(int value) {
    if (rear == MAX_SIZE - 1) {
        printf("Queue is full\n");
    } else {
        if (front == -1)
            front = 0;
        rear++;
        items[rear] = value;
    }
}

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

```c
            front = -1;
            rear = -1;
        } else {
            front++;
        }
    }
}

void display() {
    if (rear == -1) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements are: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", items[i]);
        }
        printf("\n");
    }
}
```

```
Queue is empty
Queue elements are: 1 3 4 5 6
Deleted: 1
Queue elements are: 3 4 5 6
```

Program 2: Circular Queue Implementation

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

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

int isFull() {
    if ((rear + 1) % SIZE == front)
        return 1;
    else
        return 0;
}

int isEmpty() {
    if (front == -1 && rear == -1)
        return 1;
    else
        return 0;
}
```

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

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

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

int main() {
    dequeue();
```

```
    enqueue(5);
    enqueue(7);
    enqueue(10);
    display();
    dequeue();
    display();
    return 0;
}
```

```
Queue is empty
Inserted 5
Inserted 7
Inserted 10
Front: 0
Items: 5 7 10
Rear: 2
Deleted: 5
Front: 1
Items: 7 10
Rear: 2
```

LAB 4

Program 1: Singly Linked List (Insert)

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

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

void insertAtBeginning(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

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

void insertAtEnd(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head_ref;
    new_node->data = new_data;
    new_node->next = NULL;
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}

void printList(struct Node* node) {
```

```c
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

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

    insertAtEnd(&head, 1);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 3);
    insertAtEnd(&head, 4);
    insertAfter(head->next, 5);

    printf("Linked List: ");
    printList(head);

    return 0;
}
```

```
Linked List: 3 2 5 1 4
```

Program 2: LeetCode - Min Stack

```c
#define MAX_SIZE 1000

typedef struct {
    int *stack;
    int *min_stack;
    int top;
} MinStack;

MinStack* minStackCreate() {
    MinStack* obj = (MinStack*)malloc(sizeof(MinStack));
    obj->stack = (int*)malloc(MAX_SIZE * sizeof(int));
    obj->min_stack = (int*)malloc(MAX_SIZE * sizeof(int));
    obj->top = -1;
    return obj;
}

void minStackPush(MinStack* obj, int val) {
    if(obj->top == -1){
        obj->top++;
        obj->stack[obj->top] = val;
```

```
        obj->min_stack[obj->top] = val;
    } else if (obj->top < MAX_SIZE-1) {
        obj->top++;
        obj->stack[obj->top] = val;
        if(obj->min_stack[obj->top-1] < val)
            obj->min_stack[obj->top] = obj->min_stack[obj->top-1];
        else
            obj->min_stack[obj->top] = val;
    }
}

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

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

int minStackGetMin(MinStack* obj) {
    return obj->min_stack[obj->top];
}

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

LAB 5

Program 1: Singly Linked List (Delete)

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

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

void insertAtBeginning(struct Node** head_ref, int new_data) {
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   new_node->data = new_data;
   new_node->next = *head_ref;
   *head_ref = new_node;
}

void insertAfter(struct Node* prev_node, int new_data) {
   if (prev_node == NULL) {
       printf("The previous node cannot be NULL\n");
       return;
   }
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   new_node->data = new_data;
   new_node->next = prev_node->next;
   prev_node->next = new_node;
}

void insertAtEnd(struct Node** head_ref, int new_data) {
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   struct Node* last = *head_ref;
   new_node->data = new_data;
   new_node->next = NULL;
   if (*head_ref == NULL) {
       *head_ref = new_node;
       return;
   }
   while (last->next != NULL)
       last = last->next;
   last->next = new_node;
}

void deleteNode(struct Node** head_ref, int key) {
   struct Node *temp = *head_ref, *prev;
```

```c
    if (temp != NULL && temp->data == key) {
        *head_ref = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL)
        return;
    prev->next = temp->next;
    free(temp);
}

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

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

    insertAtEnd(&head, 1);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 3);
    insertAtEnd(&head, 4);
    insertAfter(head->next, 5);

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

    printf("\nAfter deleting an element:\n");
    displayList(head);
    return 0;
}
```

```
Linked List: 3 2 5 1 4

After deleting an element:
3 2 5 1 4
```

Program 2: LeetCode - Reverse Linked List

```c
struct ListNode* createNode(int val) {
    struct ListNode* newNode = (struct ListNode*)malloc(sizeof(struct ListNode));
    newNode->val = val;
    newNode->next = NULL;
    return newNode;
}

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

    struct ListNode dummy;
    dummy.next = head;
    struct ListNode* prev = &dummy;

    for (int i = 1; i < left; ++i) {
        prev = prev->next;
    }

    struct ListNode* current = prev->next;
    struct ListNode* next = NULL;
    struct ListNode* tail = current;

    for (int i = left; i <= right; ++i) {
        struct ListNode* temp = current->next;
        current->next = next;
        next = current;
        current = temp;
    }
    prev->next = next;
    tail->next = current;
    return dummy.next;}
```

LAB 6

Program 1: Singly Linked List (Sorting, Reversing, Concatenation)

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

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

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

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

void sortList(struct Node* head) {
    struct Node *current, *nextNode;
    int temp;

    if (head == NULL) {
        return;
    }

    do {
        current = head;
        nextNode = current->next;

        while (nextNode != NULL) {
            if (current->data > nextNode->data) {
                temp = current->data;
                current->data = nextNode->data;
                nextNode->data = temp;
```

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

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

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

    return prev;
}

void concatenateLists(struct Node* list1, struct Node* list2) {
    while (list1->next != NULL) {
        list1 = list1->next;
    }
    list1->next = list2;
}

int main() {
    struct Node* head1 = createNode(3);
    head1->next = createNode(1);
    head1->next->next = createNode(5);

    printf("Original Linked List 1: ");
    displayList(head1);

    sortList(head1);
    printf("Sorted Linked List 1: ");
    displayList(head1);

    head1 = reverseList(head1);
    printf("Reversed Linked List 1: ");
    displayList(head1);
```

```
    struct Node* head2 = createNode(2);
    head2->next = createNode(4);
    head2->next->next = createNode(6);

    printf("Original Linked List 2: ");
    displayList(head2);

    concatenateLists(head1, head2);
    printf("Concatenated Linked List: ");
    displayList(head1);

    return 0;
}
```

```
Original Linked List 1: 3 -> 1 -> 5 -> NULL
Sorted Linked List 1: 1 -> 3 -> 5 -> NULL
Reversed Linked List 1: 5 -> 3 -> 1 -> NULL
Original Linked List 2: 2 -> 4 -> 6 -> NULL
Concatenated Linked List: 5 -> 3 -> 1 -> 2 -> 4 -> 6 -> NULL
```

Program 2: Stack & Queue Using Singly Linked List

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

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

struct LinkedList {
    struct Node* head;
};

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

struct LinkedList* initializeList() {
    struct LinkedList* newList = (struct LinkedList*)malloc(sizeof(struct
LinkedList));
    newList->head = NULL;
    return newList;
}
```

```c
void push(struct LinkedList* list, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = list->head;
    list->head = newNode;
}

int pop(struct LinkedList* list) {
    if (list->head == NULL) {
        printf("Stack/Queue is empty\n");
        return -1;
    }

    struct Node* temp = list->head;
    int poppedData = temp->data;
    list->head = temp->next;
    free(temp);
    return poppedData;
}

void enqueue(struct LinkedList* list, int data) {
    struct Node* newNode = createNode(data);
    if (list->head == NULL) {
        list->head = newNode;
        return;
    }

    struct Node* lastNode = list->head;
    while (lastNode->next != NULL) {
        lastNode = lastNode->next;
    }

    lastNode->next = newNode;
}

int dequeue(struct LinkedList* list) {
    return pop(list);
}

void display(struct LinkedList* list) {
    struct Node* current = list->head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
```

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

void freeList(struct LinkedList* list) {
    struct Node* current = list->head;
    struct Node* nextNode;

    while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
    }

    free(list);
}

int main() {
    struct LinkedList* linkedList = initializeList();

    push(linkedList, 1);
    push(linkedList, 2);
    push(linkedList, 3);
    printf("Stack: ");
    display(linkedList);
    printf("Popped from stack: %d\n", pop(linkedList));
    display(linkedList);

    enqueue(linkedList, 4);
    enqueue(linkedList, 5);
    enqueue(linkedList, 6);
    printf("\nQueue: ");
    display(linkedList);
    printf("Dequeued from queue: %d\n", dequeue(linkedList));
    display(linkedList);

    freeList(linkedList);

    return 0;
}
```

```
Stack: 3 2 1
Popped from stack: 3
2 1

Queue: 2 1 4 5 6
Dequeued from queue: 2
1 4 5 6
```

LAB 7

Program 1: Doubly Linked List Implementation

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

struct Node {
    int data;
    struct Node* prev;
    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(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

void insertToLeft(struct Node** head, struct Node* target, int value) {
    struct Node* newNode = createNode(value);

    if (target->prev != NULL) {
        target->prev->next = newNode;
        newNode->prev = target->prev;
    } else {
        *head = newNode;
    }

    newNode->next = target;
    target->prev = newNode;
}

void deleteNodeByValue(struct Node** head, int value) {
    struct Node* current = *head;

    while (current != NULL) {
        if (current->data == value) {
            if (current->prev != NULL) {
```

```c
                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 in the list.\n", value);
}

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

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

    head = createNode(1);
    struct Node* second = createNode(2);
    struct Node* third = createNode(3);

    head->next = second;
    second->prev = head;
    second->next = third;
    third->prev = second;

    displayList(head);

    insertToLeft(&head, second, 5);
    displayList(head);
```

```
    deleteNodeByValue(&head, 2);
    displayList(head);

    return 0;
}
```

```
Doubly Linked List: 1 2 3
Doubly Linked List: 1 5 2 3
Doubly Linked List: 1 5 3
```

Program 2: LeetCode - Split Linked List in Parts

```
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 width = length / k;
    int remainder = length % k;
    struct ListNode** result = (struct ListNode**)malloc(k * sizeof(struct
ListNode*));

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

    *returnSize = k;
    return result;}
```

LAB 8

Program 1: Binary Search Tree Implementation

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

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

struct TreeNode* createNode(int data) {
   struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}

struct TreeNode* insertNode(struct TreeNode* root, int data) {
   if (root == NULL) {
       root = createNode(data);
   } else if (data <= root->data) {
       root->left = insertNode(root->left, data);
   } else {
       root->right = insertNode(root->right, data);
   }
   return root;
}

void inorderTraversal(struct TreeNode* root) {
   if (root != NULL) {
       inorderTraversal(root->left);
       printf("%d ", root->data);
       inorderTraversal(root->right);
   }
}

void preorderTraversal(struct TreeNode* root) {
   if (root != NULL) {
       printf("%d ", root->data);
       preorderTraversal(root->left);
       preorderTraversal(root->right);
   }
```

```c
}

void postorderTraversal(struct TreeNode* root) {
   if (root != NULL) {
       postorderTraversal(root->left);
       postorderTraversal(root->right);
       printf("%d ", root->data);
   }
}

void display(struct TreeNode* root) {
   printf("Elements in the tree: ");
   inorderTraversal(root);
   printf("\n");
}

int main() {
   struct TreeNode* root = NULL;

   root = insertNode(root, 50);
   root = insertNode(root, 30);
   root = insertNode(root, 20);
   root = insertNode(root, 40);
   root = insertNode(root, 70);
   root = insertNode(root, 60);
   root = insertNode(root, 80);

   display(root);

   printf("Inorder traversal: ");
   inorderTraversal(root);
   printf("\n");

   printf("Preorder traversal: ");
   preorderTraversal(root);
   printf("\n");

   printf("Postorder traversal: ");
   postorderTraversal(root);
   printf("\n");

   return 0;
}
```

```
Elements in the tree: 20 30 40 50 60 70 80
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
```

Program 2: LeetCode - Rotate List

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

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

   int length = getLength(head);
   k = k % length;
   if (k == 0) {
       return head;
   }

   struct ListNode* tail = head;
   for (int i = 0; i < length - k - 1; i++) {
       tail = tail->next;
   }

   struct ListNode* newHead = tail->next;
   tail->next = NULL;

   struct ListNode* current = newHead;
   while (current->next != NULL) {
       current = current->next;
   }
   current->next = head;

   return newHead;
}
```

LAB 9

Program 1: Traversing a Graph Using BFS

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

#define MAX_NODES 100

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

struct Graph {
   int num_vertices;
   struct Node* adjacency_list[MAX_NODES];
};

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

struct Graph* createGraph(int num_vertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->num_vertices = num_vertices;

   for (int i = 0; i < num_vertices; ++i)
       graph->adjacency_list[i] = NULL;

   return graph;
}

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

   newNode = createNode(src);
   newNode->next = graph->adjacency_list[dest];
   graph->adjacency_list[dest] = newNode;
}
```

```c
void BFS(struct Graph* graph, int start) {
   int visited[MAX_NODES] = {0};
   int queue[MAX_NODES];
   int front = -1, rear = -1;

   queue[++rear] = start;
   visited[start] = 1;

   while (front < rear) {
       int current = queue[++front];
       printf("%d ", current);

       struct Node* temp = graph->adjacency_list[current];
       while (temp) {
           int adj_vertex = temp->data;
           if (!visited[adj_vertex]) {
               queue[++rear] = adj_vertex;
               visited[adj_vertex] = 1;
           }
           temp = temp->next;
       }
   }
}

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

   printf("BFS traversal starting from vertex 2: ");
   BFS(graph, 2);

   return 0;
}
```

```
BFS traversal starting from vertex 2: 2 3 0 1
```

Program 2: Traversing a Graph Using DFS

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

```c
#define MAX_NODES 100

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

struct Graph {
   int num_vertices;
   struct Node* adjacency_list[MAX_NODES];
};

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

struct Graph* createGraph(int num_vertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->num_vertices = num_vertices;

   for (int i = 0; i < num_vertices; ++i)
       graph->adjacency_list[i] = NULL;

   return graph;
}

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

   newNode = createNode(src);
   newNode->next = graph->adjacency_list[dest];
   graph->adjacency_list[dest] = newNode;
}

void DFS(struct Graph* graph, int vertex, int visited[]) {
   visited[vertex] = 1;
   struct Node* temp = graph->adjacency_list[vertex];
   while (temp) {
       int adj_vertex = temp->data;
```

```c
        if (!visited[adj_vertex])
            DFS(graph, adj_vertex, visited);
        temp = temp->next;
    }
}

int isConnected(struct Graph* graph) {
    int visited[MAX_NODES] = {0};
    DFS(graph, 0, visited);

    for (int i = 0; i < graph->num_vertices; ++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, 2);
    addEdge(graph, 3, 4);

    if (isConnected(graph))
        printf("The graph is connected.\n");
    else
        printf("The graph is not connected.\n");

    return 0;
}
```

```
The graph is not connected.
```

Program 3: HackerRank - Swap Node

```c
#include <assert.h>
#include <ctype.h>
#include <limits.h>
#include <math.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
char* readline();
char* ltrim(char*);
char* rtrim(char*);
char** split_string(char*);

int parse_int(char*);

// Node structure for the binary tree
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// Function to create a new Node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to swap subtrees at a given depth
void swapSubtrees(Node* root, int k, int depth) {
    if (root == NULL)
        return;

    if (depth % k == 0) {
        Node* temp = root->left;
        root->left = root->right;
        root->right = temp;
    }

    swapSubtrees(root->left, k, depth + 1);
    swapSubtrees(root->right, k, depth + 1);
}

// Function to perform inorder traversal of the binary tree
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);
```

```
}

// Function to swap nodes in a binary tree based on queries
int** swapNodes(int indexes_rows, int indexes_columns, int** indexes, int
queries_count, int* queries, int* result_rows, int* result_columns) {
   // Build the binary tree
   Node* root = createNode(1);
   Node* queue[indexes_rows];
   int front = -1, rear = -1;
   queue[++rear] = root;

   for (int i = 0; i < indexes_rows; i++) {
       Node* curr = queue[++front];
       int leftData = indexes[i][0];
       int rightData = indexes[i][1];

       if (leftData != -1) {
           curr->left = createNode(leftData);
           queue[++rear] = curr->left;
       }
       if (rightData != -1) {
           curr->right = createNode(rightData);
           queue[++rear] = curr->right;
       }
   }

   // Perform swapping based on queries
   int** resultArray = (int**)malloc(queries_count * sizeof(int*));
   *result_rows = queries_count;
   *result_columns = indexes_rows;

   int resultIndex = 0;
   for (int i = 0; i < queries_count; i++) {
       int k = queries[i];
       swapSubtrees(root, k, 1);

       // Traverse the tree in inorder and store the result
       int* result = (int*)malloc(indexes_rows * sizeof(int));
       int index = 0;
       inorderTraversal(root, &result, &index);
       resultArray[resultIndex++] = result;
   }

   return resultArray;
}
```

```
int main()
{
    FILE* fptr = fopen(getenv("OUTPUT_PATH"), "w");

    int n = parse_int(ltrim(rtrim(readline())));

    int** indexes = malloc(n * sizeof(int*));

    for (int i = 0; i < n; i++) {
        *(indexes + i) = malloc(2 * (sizeof(int)));

        char** indexes_item_temp = split_string(rtrim(readline()));

        for (int j = 0; j < 2; j++) {
            int indexes_item = parse_int(*(indexes_item_temp + j));

            *(*(indexes + i) + j) = indexes_item;
        }
    }

    int queries_count = parse_int(ltrim(rtrim(readline())));

    int* queries = malloc(queries_count * sizeof(int));

    for (int i = 0; i < queries_count; i++) {
        int queries_item = parse_int(ltrim(rtrim(readline())));

        *(queries + i) = queries_item;
    }

    int result_rows;
    int result_columns;
    int** result = swapNodes(n, 2, indexes, queries_count, queries, &result_rows,
&result_columns);

    for (int i = 0; i < result_rows; i++) {
        for (int j = 0; j < result_columns; j++) {
            fprintf(fptr, "%d", *(*(result + i) + j));

            if (j != result_columns - 1) {
                fprintf(fptr, " ");
            }
        }
```

```
        if (i != result_rows - 1) {
            fprintf(fptr, "\n");
        }
    }

    fprintf(fptr, "\n");

    fclose(fptr);

    return 0;
}

char* readline() {
    size_t alloc_length = 1024;
    size_t data_length = 0;

    char* data = malloc(alloc_length);

    while (true) {
        char* cursor = data + data_length;
        char* line = fgets(cursor, alloc_length - data_length, stdin);

        if (!line) {
            break;
        }

        data_length += strlen(cursor);

        if (data_length < alloc_length - 1 || data[data_length - 1] == '\n') {
            break;
        }

        alloc_length <<= 1;

        data = realloc(data, alloc_length);

        if (!data) {
            data = '\0';

            break;
        }
    }

    if (data[data_length - 1] == '\n') {
        data[data_length - 1] = '\0';
```

```
        data = realloc(data, data_length);

        if (!data) {
            data = '\0';
        }
    } else {
        data = realloc(data, data_length + 1);

        if (!data) {
            data = '\0';
        } else {
            data[data_length] = '\0';
        }
    }

    return data;
}

char* ltrim(char* str) {
    if (!str) {
        return '\0';
    }

    if (!*str) {
        return str;
    }

    while (*str != '\0' && isspace(*str)) {
        str++;
    }

    return str;
}

char* rtrim(char* str) {
    if (!str) {
        return '\0';
    }

    if (!*str) {
        return str;
    }

    char* end = str + strlen(str) - 1;
```

```c
    while (end >= str && isspace(*end)) {
        end--;
    }

    *(end + 1) = '\0';

    return str;
}

char** split_string(char* str) {
    char** splits = NULL;
    char* token = strtok(str, " ");

    int spaces = 0;

    while (token) {
        splits = realloc(splits, sizeof(char*) * ++spaces);

        if (!splits) {
            return splits;
        }

        splits[spaces - 1] = token;

        token = strtok(NULL, " ");
    }

    return splits;
}

int parse_int(char* str) {
    char* endptr;
    int value = strtol(str, &endptr, 10);

    if (endptr == str || *endptr != '\0') {
        exit(EXIT_FAILURE);
    }

    return value;
}
```

LAB 10

Program 1: Implementing Hash Table

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

#define MAX_EMPLOYEES 100
#define HT_SIZE 10

typedef struct {
   int key;
   char name[50];
} EmployeeRecord;

typedef struct {
   int key;
   int address;
} HashTableEntry;

HashTableEntry hashTable[HT_SIZE];

int hashFunction(int key) {
   return key % HT_SIZE;
}

void initializeHashTable() {
   for (int i = 0; i < HT_SIZE; i++) {
       hashTable[i].key = -1;
       hashTable[i].address = -1;
   }
}

void insertRecord(EmployeeRecord record) {
   int hashIndex = hashFunction(record.key);

   while (hashTable[hashIndex].key != -1) {
       hashIndex = (hashIndex + 1) % HT_SIZE;
   }

   hashTable[hashIndex].key = record.key;
   hashTable[hashIndex].address = hashIndex;
}
```

```c
int searchRecord(int key) {
    int hashIndex = hashFunction(key);

    while (hashTable[hashIndex].key != key && hashTable[hashIndex].key != -1) {
        hashIndex = (hashIndex + 1) % HT_SIZE;
    }

    if (hashTable[hashIndex].key == key) {
        return hashIndex;
    } else {
        return -1;
    }
}

void displayRecord(int index) {
    if (index != -1) {
        printf("Employee Record Found:\n");
        printf("Key: %d\n", hashTable[index].key);
        printf("Address: %d\n", hashTable[index].address);
    } else {
        printf("Employee Record Not Found.\n");
    }
}

int main() {
    int choice, key;
    EmployeeRecord records[MAX_EMPLOYEES];
    int numRecords = 0;

    initializeHashTable();

    do {
        printf("\nEmployee Record Management System\n");
        printf("1. Insert Record\n");
        printf("2. Search Record\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if (numRecords < MAX_EMPLOYEES) {
                    printf("Enter employee key: ");
                    scanf("%d", &records[numRecords].key);
                    printf("Enter employee name: ");
```

```c
                    scanf("%s", records[numRecords].name);
                    insertRecord(records[numRecords]);
                    numRecords++;
                    printf("Record inserted successfully.\n");
                } else {
                    printf("Maximum number of records reached.\n");
                }
                break;
            case 2:
                printf("Enter employee key to search: ");
                scanf("%d", &key);
                int index = searchRecord(key);
                displayRecord(index);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    return 0;
}
```

```
Employee Record Management System
1. Insert Record
2. Search Record
3. Exit
Enter your choice: 1
Enter employee key: 4444
Enter employee name: aman
Record inserted successfully.
```

```
Employee Record Management System
1. Insert Record
2. Search Record
3. Exit
Enter your choice: 2
Enter employee key to search: 4444
Employee Record Found:
Key: 4444
Address: 1
```