RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

25 years OF ACADEMIC EXCELLENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Laboratory Manual

REGULATION 2023

CS23231 – DATA STRUCTURES

RAJALAKSHMI ENGINEERING COLLEGE

An Autonomous Institution, Affiliated to Anna University
Rajalakshmi Nagar, Thandalam – 602 105

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS23231 – DATA STRUCTURES (*Regulation 2023*)
LAB MANUAL

Name	M.HARISH
Register No.	2116-231501058
Year / Branch / Section:	1st Year / AIML / A
Semester	2nd Semester
Academic Year	2023-2024

LESSON PLAN

Course Code	Course Title (Laboratory Integrated Theory Course)	L	T	P	C
CS23231	Data Structures	1	0	6	4

LIST OF EXPERIMENTS	
Sl. No	**Name of the experiment**
Week 1	Implementation of Single Linked List (Insertion, Deletion and Display)
Week 2	Implementation of Doubly Linked List (Insertion, Deletion and Display)
Week 3	Applications of Singly Linked List (Polynomial Manipulation)
Week 4	Implementation of Stack using Array and Linked List implementation
Week 5	Applications of Stack (Infix to Postfix)
Week 6	Applications of Stack (Evaluating Arithmetic Expression)
Week 7	Implementation of Queue using Array and Linked List implementation
Week 8	Implementation of Binary Search Tree
Week 9	Performing Tree Traversal Techniques
Week 10	Implementation of AVL Tree
Week 11	Performing Topological Sorting
Week 12	Implementation of BFS, DFS
Week 13	Implementation of Prim's Algorithm
Week 14	Implementation of Dijkstra's Algorithm
Week 15	Program to perform Sorting
Week 16	Implementation of Open Addressing (Linear Probing and Quadratic Probing)
Week 17	Implementation of Rehashing

INDEX

S. No.	Name of the Experiment	Expt. Date	Page No	Faculty Sign
1	Implementation of Single Linked List (Insertion, Deletion and Display)	23/2/24	5	
2	Implementation of Doubly Linked List (Insertion, Deletion and Display)	1/3/24	10	
3	Applications of Singly Linked List (Polynomial Manipulation)	8/3/24	15	
4	Implementation of Stack using Array and Linked List implementation	15/3/24	18	
5	Applications of Stack (Infix to Postfix)	5/4/24	20	
6	Applications of Stack (Evaluating Arithmetic Expression)	12/4/24	22	
7	Implementation of Queue using Array and Linked List implementation	19/4/24	24	
8	Performing Tree Traversal Techniques	17/5/24	26	
9	Implementation of Binary Search Tree	17/5/24	28	
10	Implementation of AVL Tree	24/5/24	31	
11	Implementation of BFS, DFS	24/5/24	34	
12	Performing Topological Sorting	24/5/24	36	
13	Implementation of Prim's Algorithm	31/5/24	38	
14	Implementation of Dijkstra's Algorithm	31/5/24	40	
15	Program to perform Sorting	31/5/24	42	
16	Implementation of Collision Resolution Techniques	31/5/24	47	

Ex. No.: 1	Implementation of Single Linked List	Date: 23/2/24

Write a C program to implement the following operations on Singly Linked List.

- **(i)** **Insert a node in the beginning of a list.**
- **(ii)** **Insert a node after P**
- **(iii)** **Insert a node at the end of a list**
- **(iv)** **Find an element in a list**
- **(v)** **FindNext**
- **(vi)** **FindPrevious**
- **(vii)** **isLast**
- **(viii)** **isEmpty**
- **(ix)** **Delete a node in the beginning of a list.**
- **(x)** **Delete a node after P**
- **(xi)** **Delete a node at the end of a list**
- **(xii)** **Delete the List**

Algorithm:

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

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

struct node* head = NULL;

void insertfront(int ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = head;
        head = newnode;
    }
}

void insertend(int ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = NULL;
```

```
        if (head == NULL) {
            head = newnode;
        } else {
            struct node* t = head;
            while (t->next != NULL) {
                t = t->next;
            }
            t->next = newnode;
        }
    }
}

int listsize() {
    int count = 0;
    struct node* t = head;
    while (t != NULL) {
        count++;
        t = t->next;
    }
    return count;
}

void insertpos(int ele, int pos) {
    int ls = listsize();
    if ((head == NULL && pos != 1) || pos <= 0 || pos > ls + 1) {
        printf("\nInvalid position to insert a node\n");
        return;
    }

    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        if (pos == 1) {
            newnode->next = head;
            head = newnode;
        } else {
            struct node* temp = head;
            for (int count = 1; count < pos - 1; count++) {
                temp = temp->next;
            }
            newnode->next = temp->next;
            temp->next = newnode;
        }
    }
}

void findnext(int s) {
    struct node* temp = head;
    while (temp != NULL && temp->data != s) {
        temp = temp->next;
```

```
    }
    if (temp != NULL && temp->next != NULL) {
        printf("\nNext element of %d is %d\n", s, temp->next->data);
    } else {
        printf("\nNo next element for %d\n", s);
    }
}

void findprev(int s) {
    if (head == NULL || head->data == s) {
        printf("\nNo previous element for %d\n", s);
        return;
    }
    struct node* temp = head;
    while (temp->next != NULL && temp->next->data != s) {
        temp = temp->next;
    }
    if (temp->next != NULL) {
        printf("\nPrevious element of %d is %d\n", s, temp->data);
    } else {
        printf("\nElement %d not found\n", s);
    }
}

void find(int s) {
    struct node* temp = head;
    while (temp != NULL && temp->data != s) {
        temp = temp->next;
    }
    if (temp != NULL) {
        printf("\nElement %d is present in the list\n", s);
    } else {
        printf("\nElement %d is not present in the list\n", s);
    }
}

void isempty() {
    if (head == NULL) {
        printf("\nList is empty\n");
    } else {
        printf("\nList is not empty\n");
    }
}

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

```
    }

    void deleteAtEnd() {
        if (head == NULL) {
            printf("\nList is empty\n");
            return;
        }
        if (head->next == NULL) {
            free(head);
            head = NULL;
        } else {
            struct node* temp = head;
            while (temp->next->next != NULL) {
                temp = temp->next;
            }
            free(temp->next);
            temp->next = NULL;
        }
    }

    void delete(int ele) {
        if (head == NULL) {
            printf("\nList is empty\n");
            return;
        }
        if (head->data == ele) {
            struct node* temp = head;
            head = head->next;
            free(temp);
        } else {
            struct node* temp = head;
            while (temp->next != NULL && temp->next->data != ele) {
                temp = temp->next;
            }
            if (temp->next != NULL) {
                struct node* delNode = temp->next;
                temp->next = temp->next->next;
                free(delNode);
            } else {
                printf("\nElement %d not found\n", ele);
            }
        }
    }

    void display() {
        struct node* t = head;
        while (t != NULL) {
            printf("%d\t", t->data);
            t = t->next;
        }
```

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

int main() {
    insertfront(5);
    insertfront(10);
    insertfront(20);
    insertend(30);
    insertend(40);
    display();

    printf("\nAfter inserting 15 at the 2nd position\n");
    insertpos(15, 2);
    display();

    findnext(30);
    findprev(30);

    find(15);
    isempty();

    printf("\nAfter deleting the first element\n");
    deleteAtBeginning();
    display();

    printf("\nAfter deleting the last element\n");
    deleteAtEnd();
    display();

    printf("\nAfter deleting element 15\n");
    delete(15);
    display();

    return 0;
}
```

Ex. No.: 2	Implementation of Doubly Linked List	Date: 11/3/24

Write a C program to implement the following operations on Doubly Linked List.

(i) Insertion

(ii) Deletion

(iii) Search

(iv) Display

Algorithm:

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

struct node {
    int data;
    struct node* next;
    struct node* prev;
};

struct node* head = NULL;

void insertfront(int ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = head;
        newnode->prev = NULL;
        if (head != NULL) {
            head->prev = newnode;
        }
        head = newnode;
    }
}

void insertend(int ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = NULL;
        if (head == NULL) {
            newnode->prev = NULL;
            head = newnode;
        } else {
            struct node* t = head;
            while (t->next != NULL) {
```

```
                t = t->next;
            }
            t->next = newnode;
            newnode->prev = t;
        }
    }
}

int listsize() {
    int count = 0;
    struct node* t = head;
    while (t != NULL) {
        count++;
        t = t->next;
    }
    return count;
}

void insertpos(int ele, int pos) {
    int ls = listsize();
    if ((head == NULL && pos != 1) || pos <= 0 || pos > ls + 1) {
        printf("\nInvalid position to insert a node\n");
        return;
    }

    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        if (pos == 1) {
            newnode->next = head;
            newnode->prev = NULL;
            if (head != NULL) {
                head->prev = newnode;
            }
            head = newnode;
        } else {
            struct node* temp = head;
            for (int count = 1; count < pos - 1; count++) {
                temp = temp->next;
            }
            newnode->next = temp->next;
            newnode->prev = temp;
            if (temp->next != NULL) {
                temp->next->prev = newnode;
            }
            temp->next = newnode;
        }
    }
}
```

```c
void findnext(int s) {
    struct node* temp = head;
    while (temp != NULL && temp->data != s) {
        temp = temp->next;
    }
    if (temp != NULL && temp->next != NULL) {
        printf("\nNext element of %d is %d\n", s, temp->next->data);
    } else {
        printf("\nNo next element for %d\n", s);
    }
}

void findprev(int s) {
    struct node* temp = head;
    while (temp != NULL && temp->data != s) {
        temp = temp->next;
    }
    if (temp != NULL && temp->prev != NULL) {
        printf("\nPrevious element of %d is %d\n", s, temp->prev->data);
    } else {
        printf("\nNo previous element for %d\n", s);
    }
}

void find(int s) {
    struct node* temp = head;
    while (temp != NULL && temp->data != s) {
        temp = temp->next;
    }
    if (temp != NULL) {
        printf("\nElement %d is present in the list\n", s);
    } else {
        printf("\nElement %d is not present in the list\n", s);
    }
}

void isempty() {
    if (head == NULL) {
        printf("\nList is empty\n");
    } else {
        printf("\nList is not empty\n");
    }
}

void deleteAtBeginning() {
    if (head != NULL) {
        struct node* temp = head;
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
```

```
        }
        free(temp);
    }
}

void deleteAtEnd() {
    if (head == NULL) {
        printf("\nList is empty\n");
        return;
    }
    if (head->next == NULL) {
        free(head);
        head = NULL;
    } else {
        struct node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->prev->next = NULL;
        free(temp);
    }
}

void delete(int ele) {
    if (head == NULL) {
        printf("\nList is empty\n");
        return;
    }
    if (head->data == ele) {
        struct node* temp = head;
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
        free(temp);
    } else {
        struct node* temp = head;
        while (temp != NULL && temp->data != ele) {
            temp = temp->next;
        }
        if (temp != NULL) {
            if (temp->prev != NULL) {
                temp->prev->next = temp->next;
            }
            if (temp->next != NULL) {
                temp->next->prev = temp->prev;
            }
            free(temp);
        } else {
            printf("\nElement %d not found\n", ele);
```

```
        }
    }
}

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

int main() {
    insertfront(5);
    insertfront(10);
    insertfront(20);
    insertend(30);
    insertend(40);
    display();

    printf("\nAfter inserting 15 at the 2nd position\n");
    insertpos(15, 2);
    display();

    findnext(30);
    findprev(30);

    find(15);
    isempty();

    printf("\nAfter deleting the first element\n");
    deleteAtBeginning();
    display();

    printf("\nAfter deleting the last element\n");
    deleteAtEnd();
    display();

    printf("\nAfter deleting element 15\n");
    delete(15);
    display();

    return 0;
}
```

Ex. No.: 3	Polynomial Manipulation	Date: 8/3/24

Write a C program to implement the following operations on Singly Linked List.

(i) Polynomial Addition

(ii) Polynomial Subtraction

(iii) Polynomial Multiplication

Algorithm:

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

struct node {
    int coeff;
    int exp;
    struct node* next;
    struct node* prev;
};

struct node* head1 = NULL;
struct node* head2 = NULL;
struct node* headResult = NULL;

void insertTerm(struct node** head, int coeff, int exp) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->coeff = coeff;
        newnode->exp = exp;
        newnode->next = NULL;
        newnode->prev = NULL;
        if (*head == NULL) {
            *head = newnode;
        } else {
            struct node* t = *head;
            while (t->next != NULL) {
                t = t->next;
            }
            t->next = newnode;
            newnode->prev = t;
        }
    }
}

void display(struct node* head) {
    struct node* t = head;
    while (t != NULL) {
        if (t->coeff > 0 && t != head)
```

```
            printf("+ ");
        printf("%dx^%d ", t->coeff, t->exp);
        t = t->next;
    }
    printf("\n");
}

struct node* addPolynomials(struct node* head1, struct node* head2) {
    struct node* result = NULL;
    struct node* t1 = head1;
    struct node* t2 = head2;

    while (t1 != NULL && t2 != NULL) {
        if (t1->exp == t2->exp) {
            insertTerm(&result, t1->coeff + t2->coeff, t1->exp);
            t1 = t1->next;
            t2 = t2->next;
        } else if (t1->exp > t2->exp) {
            insertTerm(&result, t1->coeff, t1->exp);
            t1 = t1->next;
        } else {
            insertTerm(&result, t2->coeff, t2->exp);
            t2 = t2->next;
        }
    }

    while (t1 != NULL) {
        insertTerm(&result, t1->coeff, t1->exp);
        t1 = t1->next;
    }

    while (t2 != NULL) {
        insertTerm(&result, t2->coeff, t2->exp);
        t2 = t2->next;
    }

    return result;
}

struct node* multiplyPolynomials(struct node* head1, struct node* head2) {
    struct node* result = NULL;
    struct node* t1 = head1;
    struct node* t2 = head2;

    while (t1 != NULL) {
        t2 = head2;
        while (t2 != NULL) {
            insertTerm(&result, t1->coeff * t2->coeff, t1->exp + t2->exp);
            t2 = t2->next;
        }
```

```
        t1 = t1->next;
    }
    struct node* t = result;
    struct node* tPrev = NULL;

    while (t != NULL && t->next != NULL) {
        tPrev = t;
        struct node* tNext = t->next;
        while (tNext != NULL) {
            if (t->exp == tNext->exp) {
                t->coeff += tNext->coeff;
                tPrev->next = tNext->next;
                if (tNext->next != NULL) {
                    tNext->next->prev = tPrev;
                }
                free(tNext);
                tNext = tPrev->next;
            } else {
                tPrev = tNext;
                tNext = tNext->next;
            }
        }
        t = t->next;
    }

    return result;
}
int main() {
    insertTerm(&head1, 5, 2);
    insertTerm(&head1, 4, 1);
    insertTerm(&head1, 2, 0);

    insertTerm(&head2, 5, 1);
    insertTerm(&head2, 5, 0);

    printf("Polynomial 1: ");
    display(head1);

    printf("Polynomial 2: ");
    display(head2);

    headResult = addPolynomials(head1, head2);
    printf("\nAddition Result: ");
    display(headResult);

    headResult = multiplyPolynomials(head1, head2);
    printf("\nMultiplication Result: ");
    display(headResult);

    return 0;}
```

Ex. No.: 4	Implementation of Stack using Array and Linked List Implementation	Date: 15/3/24

Write a C program to implement a stack using Array and linked List implementation and execute the following operation on stack.

(i) Push an element into a stack

(ii) Pop an element from a stack

(iii) Return the Top most element from a stack

(iv) Display the elements in a stack

Algorithm:

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

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

struct node* top = NULL;

void push(int ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = top;
        top = newnode;
    }
}

int pop() {
    if (top == NULL) {
        printf("\nStack Underflow\n");
        return -1;
    } else {
        int popped = top->data;
        struct node* temp = top;
        top = top->next;
        free(temp);
        return popped;
    }
}

int peek() {
    if (top != NULL) {
        return top->data;
    } else {
```

```
        printf("\nStack is empty\n");
        return -1;
    }
}

int isEmpty() {
    return top == NULL;
}

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

int main() {
    push(10);
    push(20);
    push(30);
    display();

    printf("Top element is %d\n", peek());

    printf("Popped element is %d\n", pop());
    display();

    printf("Popped element is %d\n", pop());
    display();

    printf("Is stack empty? %s\n", isEmpty() ? "Yes" : "No");

    printf("Popped element is %d\n", pop());
    display();

    printf("Is stack empty? %s\n", isEmpty() ? "Yes" : "No");

    return 0;
}
```

Ex. No.: 5	Infix to Postfix Conversion	Date:5/4/24

Write a C program to perform infix to postfix conversion using stack.

Algorithm:

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

struct node {
    char data;
    struct node* next;
};

struct node* top = NULL;

void push(char ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = top;
        top = newnode;
    }
}

char pop() {
    if (top == NULL) {
        printf("\nStack Underflow\n");
        return -1;
    } else {
        char popped = top->data;
        struct node* temp = top;
        top = top->next;
        free(temp);
        return popped;
    }
}

char peek() {
    if (top != NULL) {
        return top->data;
    } else {
        return -1;
    }
}

int isEmpty() {
    return top == NULL;
}
```

```c
int precedence(char op) {
    switch (op) {
        case '+':
        case '-': return 1;
        case '*':
        case '/': return 2;
        case '^': return 3;
        default: return 0;
    }
}
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}
void infixToPostfix(char* infix, char* postfix) {
    int i = 0, j = 0;
    while (infix[i] != '\0') {
        if (isdigit(infix[i]) || isalpha(infix[i])) {
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            push(infix[i]);
        } else if (infix[i] == ')') {
            while (!isEmpty() && peek() != '(') {
                postfix[j++] = pop();
            }
            pop(); // Remove '(' from stack
        } else if (isOperator(infix[i])) {
            while (!isEmpty() && precedence(peek()) >= precedence(infix[i])) {
                postfix[j++] = pop();
            }
            push(infix[i]);
        }
        i++;
    }
    while (!isEmpty()) {
        postfix[j++] = pop();
    }
    postfix[j] = '\0';
}
int main() {
    char infix[100] = "a+b*(c^d-e)^(f+g*h)-i";
    char postfix[100];

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

    return 0;
}
```

Ex. No.: 6	Evaluating Arithmetic Expression	Date: 12/4/24

Write a C program to evaluate Arithmetic expression using stack.

Algorithm:

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

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

struct node* top = NULL;

void push(int ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = top;
        top = newnode;
    }
}

int pop() {
    if (top == NULL) {
        printf("\nStack Underflow\n");
        return -1;
    } else {
        int popped = top->data;
        struct node* temp = top;
        top = top->next;
        free(temp);
        return popped;
    }
}

int isEmpty() {
    return top == NULL;
}

int evaluatePostfix(char* expression) {
    int i = 0;
    while (expression[i] != '\0') {
        if (isdigit(expression[i])) {
```

```
            push(expression[i] - '0');
        } else {
            int val1 = pop();
            int val2 = pop();
            switch (expression[i]) {
                case '+': push(val2 + val1); break;
                case '-': push(val2 - val1); break;
                case '*': push(val2 * val1); break;
                case '/': push(val2 / val1); break;
            }
        }
        i++;
    }
    return pop();
}

int main() {
    char postfix[100] = "53+62/*35*+";
    printf("Postfix expression: %s\n", postfix);
    int result = evaluatePostfix(postfix);
    printf("Evaluation result: %d\n", result);

    return 0;
}
```

Ex. No.: 7	Implementation of Queue using Array and Linked List Implementation	Date: 19/ 4/24

Write a C program to implement a Queue using Array and linked List implementation and execute the following operation on stack.

(i) **Enqueue**

(ii) **Dequeue**

(iii) **Display the elements in a Queue**

Algorithm:

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

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

struct node* head = NULL;
struct node* tail = NULL;

void enqueue(int ele) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    if (newnode != NULL) {
        newnode->data = ele;
        newnode->next = NULL;
        if (tail == NULL) {
            head = tail = newnode;
        } else {
            tail->next = newnode;
            tail = newnode;
        }
    }
}

int dequeue() {
    if (head == NULL) {
        printf("\nQueue Underflow\n");
        return -1;
    } else {
        int dequeued = head->data;
        struct node* temp = head;
        head = head->next;
        if (head == NULL) {
            tail = NULL;
        }
        free(temp);
        return dequeued;
```

```
    }
}

int isEmpty() {
    return head == NULL;
}

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

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();

    printf("Dequeued element: %d\n", dequeue());
    display();

    printf("Dequeued element: %d\n", dequeue());
    display();

    printf("Is queue empty? %s\n", isEmpty() ? "Yes" : "No");

    enqueue(40);
    display();

    printf("Is queue empty? %s\n", isEmpty() ? "Yes" : "No");

    return 0;
}
```

Ex. No.: 8	Tree Traversal	Date: 17/5/24

Write a C program to implement a Binary tree and perform the following tree traversal operation.

(i) Inorder Traversal

(ii) Preorder Traversal

(iii) Postorder Traversal

Algorithm:

```
#include <stdio.h>
#include <malloc.h>
struct node {
    int data;
    struct node* left;
    struct node* right;
};
struct node* root = NULL;
struct node* newnode;

void insert(int ele) {
    newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = ele;
    newnode->left = newnode->right = NULL;

   if (root == NULL) {
        root = newnode;
    } else {
        struct node* current = root;
        struct node* parent = NULL;

        while (1) {
            parent = current;
            if (ele < parent->data) {
                current = current->left;
                if (current == NULL) {
                    parent->left = newnode;
                    return;
                }
            } else {
                current = current->right;
                if (current == NULL) {
                    parent->right = newnode;
                    return;
                }
            }
```

```c
        }
    }
}
void inorder(struct node* t) {
    if (root == NULL) return;
    if (t != NULL) {
        inorder(t->left);
        printf("%d ", t->data);
        inorder(t->right);
    }
}
void preorder(struct node* t) {
    if (root == NULL) return;
    if (t != NULL) {
        printf("%d ", t->data);
        preorder(t->left);
        preorder(t->right);
    }
}
void postorder(struct node* t) {
    if (root == NULL) return;
    if (t != NULL) {
        postorder(t->left);
        postorder(t->right);
        printf("%d ", t->data);
    }
}
int main() {
    insert(5);
    insert(3);
    insert(7);
    insert(2);
    insert(4);
    insert(6);
    insert(8);

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

    printf("Preorder Traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Ex. No.: 9	Implementation of Binary Search tree	Date: 17/5/24

Write a C program to implement a Binary Search Tree and perform the following operations.

(i) Insert

(ii) Delete

(iii) Search

(iv) Display

Algorithm:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node* left;
    struct node* right;
};
struct node* createNode(int data) {
    struct node* newNode = (struct node*)malloc(sizeof(struct node));
    if (newNode != NULL) {
        newNode->data = data;
        newNode->left = NULL;
        newNode->right = NULL;
    }
    return newNode;
}
struct node* insert(struct node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}
struct node* search(struct node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    }
    if (data < root->data) {
        return search(root->left, data);
    }
    return search(root->right, data);
}
```

```
struct node* findMin(struct node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}
struct node* deleteNode(struct node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }
        struct node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
void inorder(struct node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d\t", root->data);
        inorder(root->right);
    }
}
void preorder(struct node* root) {
    if (root != NULL) {
        printf("%d\t", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(struct node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d\t", root->data);
    }
```

```
}
int main() {
    struct node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

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

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

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

    int key = 40;
    if (search(root, key) != NULL) {
        printf("Element %d found in the BST\n", key);
    } else {
        printf("Element %d not found in the BST\n", key);
    }
    printf("Deleting 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal after deletion: ");
    inorder(root);
    printf("\n");
    printf("Deleting 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal after deletion: ");
    inorder(root);
    printf("\n");

    printf("Deleting 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal after deletion: ");
    inorder(root);
    printf("\n");

    return 0;
}
```

Ex. No.: 10	Implementation of AVL Tree	Date: 24/5/24

Write a function in C program to insert a new node with a given value into an AVL tree. Ensure that the tree remains balanced after insertion by performing rotations if necessary. Repeat the above operation to delete a node from AVL tree.

Algorithm:

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

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

struct node* root = NULL;
struct node* newnode;

int height(struct node* N) {
    if (N == NULL) return 0;
    return N->height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

struct node* rightRotate(struct node* y) {
    struct node* x = y->left;
    struct node* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

struct node* leftRotate(struct node* x) {
    struct node* y = x->right;
    struct node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
```

```
}

int getBalance(struct node* N) {
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int ele) {
    if (node == NULL) {
        newnode = (struct node*)malloc(sizeof(struct node));
        newnode->data = ele;
        newnode->left = newnode->right = NULL;
        newnode->height = 1;
        return newnode;
    }
    if (ele < node->data) {
        node->left = insert(node->left, ele);
    } else if (ele > node->data) {
        node->right = insert(node->right, ele);
    } else {
        return node;
    }
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);

    if (balance > 1 && ele < node->left->data) return rightRotate(node);
    if (balance < -1 && ele > node->right->data) return leftRotate(node);
    if (balance > 1 && ele > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && ele < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

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

void preorder(struct node* t) {
    if (root == NULL) return;
    if (t != NULL) {
```

```
        printf("%d ", t->data);
        preorder(t->left);
        preorder(t->right);
    }
}

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

int main() {
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

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

    printf("Preorder Traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Ex. No.: 11	Graph Traversal	Date: 24/5/24

Write a C program to create a graph and perform a Breadth First Search.

Algorithm:

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

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

struct adj_list {
    struct node* head;
};

struct graph {
    int num_vertices;
    struct adj_list* adj_lists;
    int* visited;
};

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

struct graph* create_graph(int n) {
    struct graph* graph = (struct graph*)malloc(sizeof(struct graph));
    graph->num_vertices = n;
    graph->adj_lists = (struct adj_list*)malloc(n * sizeof(struct adj_list));
    graph->visited = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        graph->adj_lists[i].head = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void add_edge(struct graph* graph, int src, int dest) {
    struct node* new_node1 = new_node(dest);
    new_node1->next = graph->adj_lists[src].head;
    graph->adj_lists[src].head = new_node1;

    struct node* new_node2 = new_node(src);
```

```
    new_node2->next = graph->adj_lists[dest].head;
    graph->adj_lists[dest].head = new_node2;
}

void bfs(struct graph* graph, int v) {
    int queue[1000];
    int front = -1;
    int rear = -1;
    graph->visited[v] = 1;
    queue[++rear] = v;
    while (front != rear) {
        int current_vertex = queue[++front];
        printf("%d ", current_vertex);
        struct node* temp = graph->adj_lists[current_vertex].head;
        while (temp != NULL) {
            int adj_vertex = temp->vertex;
            if (graph->visited[adj_vertex] == 0) {
                graph->visited[adj_vertex] = 1;
                queue[++rear] = adj_vertex;
            }
            temp = temp->next;
        }
    }
}

int main() {
    struct graph* graph = create_graph(6);
    add_edge(graph, 0, 1);
    add_edge(graph, 0, 2);
    add_edge(graph, 1, 3);
    add_edge(graph, 1, 4);
    add_edge(graph, 2, 4);
    add_edge(graph, 3, 4);
    add_edge(graph, 3, 5);
    add_edge(graph, 4, 5);
    printf("BFS traversal starting from vertex 0: ");
    bfs(graph, 0);
    return 0;
}
```

Ex. No.: 11	Graph Traversal	Date: 24/5/24

Write a C program to create a graph and perform a Depth First Search.

Algorithm:

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

int vis[100];

struct Graph {
    int V;
    int E;
    int** Adj;
};

struct Graph* adjMatrix() {
    struct Graph* G = (struct Graph*)malloc(sizeof(struct Graph));
    if (!G) {
        printf("Memory Error\n");
        return NULL;
    }
    G->V = 7;
    G->E = 7;
    G->Adj = (int**)malloc((G->V) * sizeof(int*));
    for (int k = 0; k < G->V; k++) {
        G->Adj[k] = (int*)malloc((G->V) * sizeof(int));
    }
    for (int u = 0; u < G->V; u++) {
        for (int v = 0; v < G->V; v++) {
            G->Adj[u][v] = 0;
        }
    }
    G->Adj[0][1] = G->Adj[1][0] = 1;
    G->Adj[0][2] = G->Adj[2][0] = 1;
    G->Adj[1][3] = G->Adj[3][1] = 1;
    G->Adj[1][4] = G->Adj[4][1] = 1;
    G->Adj[1][5] = G->Adj[5][1] = 1;
    G->Adj[1][6] = G->Adj[6][1] = 1;
    G->Adj[6][2] = G->Adj[2][6] = 1;
    return G;
}

void DFS(struct Graph* G, int u) {
    vis[u] = 1;
    printf("%d ", u);
    for (int v = 0; v < G->V; v++) {
```

```
            if (!vis[v] && G->Adj[u][v]) {
                DFS(G, v);
            }
        }
    }

    void DFStraversal(struct Graph* G) {
        for (int i = 0; i < 100; i++) {
            vis[i] = 0;
        }
        for (int i = 0; i < G->V; i++) {
            if (!vis[i]) {
                DFS(G, i);
            }
        }
    }

    void main() {
        struct Graph* G;
        G = adjMatrix();
        DFStraversal(G);
    }
```

Ex. No.: 12	Topological Sorting	Date: 24/5/24

Write a C program to create a graph and display the ordering of vertices.

Algorithm:

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

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

struct Graph {
    int numVertices;
    struct node** adjLists;
    int* visited;
};

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

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = (struct node**)malloc(vertices * sizeof(struct node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

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

void topologicalSortUtil(int v, struct Graph* graph, int* stack, int* stackIndex) {
    graph->visited[v] = 1;
```

```c
    struct node* adjList = graph->adjLists[v];
    struct node* temp = adjList;

    while (temp != NULL) {
        int connectedVertex = temp->vertex;
        if (!graph->visited[connectedVertex]) {
            topologicalSortUtil(connectedVertex, graph, stack, stackIndex);
        }
        temp = temp->next;
    }

    stack[(*stackIndex)++] = v;
}

void topologicalSort(struct Graph* graph) {
    int* stack = (int*)malloc(graph->numVertices * sizeof(int));
    int stackIndex = 0;

    for (int i = 0; i < graph->numVertices; i++) {
        if (graph->visited[i] == 0) {
            topologicalSortUtil(i, graph, stack, &stackIndex);
        }
    }

    for (int i = stackIndex - 1; i >= 0; i--) {
        printf("%d ", stack[i]);
    }
    free(stack);
}

int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 5, 2);
    addEdge(graph, 5, 0);
    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 1);

    printf("Topological Sort: ");
    topologicalSort(graph);
    printf("\n");

    return 0;
}
```

Ex. No.: 13	Graph Traversal	Date: 31/5/24

Write a C program to create a graph and find a minimum spanning tree using prims algorithm.

Algorithm:

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

#define MAX_VERTICES 100

int minKey(int key[], int mstSet[], int vertices) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < vertices; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void printMST(int parent[], int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; i++) {
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}

void primMST(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    int parent[MAX_VERTICES];
    int key[MAX_VERTICES];
    int mstSet[MAX_VERTICES];

    for (int i = 0; i < vertices; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < vertices - 1; count++) {
        int u = minKey(key, mstSet, vertices);

        mstSet[u] = 1;

        for (int v = 0; v < vertices; v++) {
```

```
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent, graph, vertices);
}

int main() {
    int vertices;
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES];

    printf("Input the adjacency matrix for the graph:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    primMST(graph, vertices);
    return 0;
}
```

Ex. No.: 14	Graph Traversal	Date: 31/5/24

Write a C program to create a graph and find the shortest path using Dijikstra's Algorithm.

Algorithm:

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

#define MAX_VERTICES 100

int minDistance(int dist[], int sptSet[], int vertices) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < vertices; v++) {
        if (!sptSet[v] && dist[v] < min) {
            min = dist[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void printSolution(int dist[], int vertices) {
    printf("Vertex \tDistance from Source\n");
    for (int i = 0; i < vertices; i++) {
        printf("%d \t%d\n", i, dist[i]);
    }
}

void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int vertices) {
    int dist[MAX_VERTICES];
    int sptSet[MAX_VERTICES];

    for (int i = 0; i < vertices; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }

    dist[src] = 0;

    for (int count = 0; count < vertices - 1; count++) {
        int u = minDistance(dist, sptSet, vertices);
        sptSet[u] = 1;

        for (int v = 0; v < vertices; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
```

```
            }
        }
    }

    printSolution(dist, vertices);
}

int main() {
    int vertices;
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES];

    printf("Input the adjacency matrix for the graph (use INT_MAX for
infinity):\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    int source;
    printf("Input the source vertex: ");
    scanf("%d", &source);

    if (source < 0 || source >= vertices) {
        printf("Invalid source vertex. Exiting...\n");
        return 1;
    }

    dijkstra(graph, source, vertices);
    return 0;
}
```

Ex. No.: 15	Sorting	Date: 31/5/24

Write a C program to take n numbers and sort the numbers in ascending order. Try to implement the same using following sorting techniques.

1. **Quick Sort**
2. **Merge Sort**

Code :

```
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[low];
    int i = low;
    int j = high;
    while (i < j) {
        while (arr[i] <= pivot && i <= high - 1) {
            i++;
        }
        while (arr[j] > pivot && j >= low + 1) {
            j--;
        }
        if (i < j) {
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[low], &arr[j]);
    return j;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int partitionIndex = partition(arr, low, high);
        quickSort(arr, low, partitionIndex - 1);
        quickSort(arr, partitionIndex + 1, high);
    }
}

int main() {
    int arr[] = { 19, 17, 15, 12, 16, 18, 4, 11, 13 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    quickSort(arr, 0, n - 1);

    printf("\nSorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

2.Merge Sort

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

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
```

```c
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main() {
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printf("Given array is \n");
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size - 1);
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Ex. No.: 16	Hashing	Date: 31/5/24

Write a C program to create a hash table and perform collision resolution using the following techniques.

(i) Open addressing

(ii) Closed Addressing

(iii) Rehashing

Algorithm:

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

typedef struct HashTable {
    int size;
    int count; // Number of elements in the table
    int* keys;
    int* values;
    bool* isOccupied; // Indicates if a slot is occupied
} HashTable;

HashTable* createTable(int size) {
    HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
    newTable->size = size;
    newTable->count = 0;
    newTable->keys = (int*)malloc(sizeof(int) * size);
    newTable->values = (int*)malloc(sizeof(int) * size);
    newTable->isOccupied = (bool*)malloc(sizeof(bool) * size);
    for (int i = 0; i < size; i++) {
        newTable->isOccupied[i] = false;
    }
    return newTable;
}

int hashFunction(int key, int size) {
    return key % size;
}

void rehash(HashTable* hashTable);

void insert(HashTable* hashTable, int key, int value) {
    if ((float)hashTable->count / hashTable->size >= 0.75) {
        rehash(hashTable);
    }
```

```c
    int hashIndex = hashFunction(key, hashTable->size);
    int originalIndex = hashIndex;
    int i = 1;

    while (hashTable->isOccupied[hashIndex]) {
        if (hashTable->keys[hashIndex] == key) {
            // If the key already exists, update the value
            hashTable->values[hashIndex] = value;
            return;
        }
        // Linear probing
        hashIndex = (originalIndex + i) % hashTable->size;
        i++;
    }

    hashTable->keys[hashIndex] = key;
    hashTable->values[hashIndex] = value;
    hashTable->isOccupied[hashIndex] = true;
    hashTable->count++;
}

void rehash(HashTable* hashTable) {
    int oldSize = hashTable->size;
    int* oldKeys = hashTable->keys;
    int* oldValues = hashTable->values;
    bool* oldIsOccupied = hashTable->isOccupied;

    int newSize = oldSize * 2;
    hashTable->keys = (int*)malloc(sizeof(int) * newSize);
    hashTable->values = (int*)malloc(sizeof(int) * newSize);
    hashTable->isOccupied = (bool*)malloc(sizeof(bool) * newSize);
    hashTable->size = newSize;
    hashTable->count = 0;

    for (int i = 0; i < newSize; i++) {
        hashTable->isOccupied[i] = false;
    }

    for (int i = 0; i < oldSize; i++) {
        if (oldIsOccupied[i]) {
            insert(hashTable, oldKeys[i], oldValues[i]);
        }
    }

    free(oldKeys);
    free(oldValues);
    free(oldIsOccupied);
}

int search(HashTable* hashTable, int key) {
```

```
    int hashIndex = hashFunction(key, hashTable->size);
    int originalIndex = hashIndex;
    int i = 1;

    while (hashTable->isOccupied[hashIndex]) {
        if (hashTable->keys[hashIndex] == key) {
            return hashTable->values[hashIndex];
        }
        // Linear probing
        hashIndex = (originalIndex + i) % hashTable->size;
        i++;
        if (hashIndex == originalIndex) {
            break; // We have circled back to the original index
        }
    }
    return -1; // Key not found
}

void delete(HashTable* hashTable, int key) {
    int hashIndex = hashFunction(key, hashTable->size);
    int originalIndex = hashIndex;
    int i = 1;

    while (hashTable->isOccupied[hashIndex]) {
        if (hashTable->keys[hashIndex] == key) {
            hashTable->isOccupied[hashIndex] = false;
            hashTable->count--;
            return;
        }
        // Linear probing
        hashIndex = (originalIndex + i) % hashTable->size;
        i++;
        if (hashIndex == originalIndex) {
            break; // We have circled back to the original index
        }
    }
}

void freeTable(HashTable* hashTable) {
    free(hashTable->keys);
    free(hashTable->values);
    free(hashTable->isOccupied);
    free(hashTable);
}

int main() {
    HashTable* hashTable = createTable(5);
    insert(hashTable, 1, 10);
    insert(hashTable, 2, 20);
    insert(hashTable, 3, 30);
```

```
    insert(hashTable, 4, 40);
    insert(hashTable, 5, 50);
    insert(hashTable, 6, 60); // This should trigger rehashing
    printf("Value for key 1: %d\n", search(hashTable, 1));
    printf("Value for key 2: %d\n", search(hashTable, 2));
    printf("Value for key 3: %d\n", search(hashTable, 3));
    printf("Value for key 4: %d\n", search(hashTable, 4));
    printf("Value for key 5: %d\n", search(hashTable, 5));
    printf("Value for key 6: %d\n", search(hashTable, 6));
    delete(hashTable, 3);
    printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));
    freeTable(hashTable);
    return 0;
}
```

2.CLOSED ADDRESSING

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

typedef struct Node {
    int key;
    int value;
    struct Node* next;
} Node;

typedef struct HashTable {
    int size;
    Node** table;
} HashTable;

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

HashTable* createTable(int size) {
    HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
    newTable->size = size;
    newTable->table = (Node**)malloc(sizeof(Node*) * size);
    for (int i = 0; i < size; i++) {
        newTable->table[i] = NULL;
    }
    return newTable;
}
```

```c
int hashFunction(int key, int size) {
    return key % size;
}

void insert(HashTable* hashTable, int key, int value) {
    int hashIndex = hashFunction(key, hashTable->size);
    Node* newNode = createNode(key, value);
    newNode->next = hashTable->table[hashIndex];
    hashTable->table[hashIndex] = newNode;
}

int search(HashTable* hashTable, int key) {
    int hashIndex = hashFunction(key, hashTable->size);
    Node* current = hashTable->table[hashIndex];
    while (current != NULL) {
        if (current->key == key) {
            return current->value;
        }
        current = current->next;
    }
    return -1;
}

void delete(HashTable* hashTable, int key) {
    int hashIndex = hashFunction(key, hashTable->size);
    Node* current = hashTable->table[hashIndex];
    Node* prev = NULL;
    while (current != NULL && current->key != key) {
        prev = current;
        current = current->next;
    }
    if (current == NULL) {
        return;
    }
    if (prev == NULL) {
        hashTable->table[hashIndex] = current->next;
    } else {
        prev->next = current->next;
    }
    free(current);
}

void freeTable(HashTable* hashTable) {
    for (int i = 0; i < hashTable->size; i++) {
        Node* current = hashTable->table[i];
        while (current != NULL) {
            Node* temp = current;
            current = current->next;
            free(temp);
```

```
        }
    }
    free(hashTable->table);
    free(hashTable);
}

int main() {
    HashTable* hashTable = createTable(10);
    insert(hashTable, 1, 10);
    insert(hashTable, 2, 20);
    insert(hashTable, 12, 30);
    printf("Value for key 1: %d\n", search(hashTable, 1));
    printf("Value for key 2: %d\n", search(hashTable, 2));
    printf("Value for key 12: %d\n", search(hashTable, 12));
    printf("Value for key 3: %d\n", search(hashTable, 3)); // Key not present
    delete(hashTable, 2);
    printf("Value for key 2 after deletion: %d\n", search(hashTable, 2));
    freeTable(hashTable);
    return 0;
}
```

C) REHASHING

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

typedef struct Node {
    int key;
    int value;
    struct Node* next;
} Node;

typedef struct HashTable {
    int size;
    int count; // Number of elements in the table
    Node** table;
} HashTable;

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

HashTable* createTable(int size) {
    HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
    newTable->size = size;
```

```
    newTable->count = 0;
    newTable->table = (Node**)malloc(sizeof(Node*) * size);
    for (int i = 0; i < size; i++) {
        newTable->table[i] = NULL;
    }
    return newTable;
}

int hashFunction(int key, int size) {
    return key % size;
}

void insert(HashTable* hashTable, int key, int value);
void rehash(HashTable* hashTable) {
    int oldSize = hashTable->size;
    Node** oldTable = hashTable->table;
    int newSize = oldSize * 2;
    hashTable->table = (Node**)malloc(sizeof(Node*) * newSize);
    hashTable->size = newSize;
    hashTable->count = 0;
    for (int i = 0; i < newSize; i++) {
        hashTable->table[i] = NULL;
    }
    for (int i = 0; i < oldSize; i++) {
        Node* current = oldTable[i];
        while (current != NULL) {
            insert(hashTable, current->key, current->value);
            Node* temp = current;
            current = current->next;
            free(temp);
        }
    }
    free(oldTable);
}

void insert(HashTable* hashTable, int key, int value) {
    if ((float)hashTable->count / hashTable->size >= 0.75) {
        rehash(hashTable);
    }
    int hashIndex = hashFunction(key, hashTable->size);
    Node* newNode = createNode(key, value);
    newNode->next = hashTable->table[hashIndex];
    hashTable->table[hashIndex] = newNode;
    hashTable->count++;
}

int search(HashTable* hashTable, int key) {
    int hashIndex = hashFunction(key, hashTable->size);
    Node* current = hashTable->table[hashIndex];
    while (current != NULL) {
```

```
        if (current->key == key) {
            return current->value;
        }
        current = current->next;
    }
    return -1;
}

void delete(HashTable* hashTable, int key) {
    int hashIndex = hashFunction(key, hashTable->size);
    Node* current = hashTable->table[hashIndex];
    Node* prev = NULL;
    while (current != NULL && current->key != key) {
        prev = current;
        current = current->next;
    }
    if (current == NULL) {
        return;
    }
    if (prev == NULL) {
        hashTable->table[hashIndex] = current->next;
    } else {
        prev->next = current->next;
    }
    free(current);
    hashTable->count--;
}

void freeTable(HashTable* hashTable) {
    for (int i = 0; i < hashTable->size; i++) {
        Node* current = hashTable->table[i];
        while (current != NULL) {
            Node* temp = current;
            current = current->next;
            free(temp);
        }
    }
    free(hashTable->table);
    free(hashTable);
}

int main() {
    HashTable* hashTable = createTable(5);
    insert(hashTable, 1, 10);
    insert(hashTable, 2, 20);
    insert(hashTable, 3, 30);
    insert(hashTable, 4, 40);
    insert(hashTable, 5, 50);
    insert(hashTable, 6, 60); // This should trigger rehashing
    printf("Value for key 1: %d\n", search(hashTable, 1));
```

```
    printf("Value for key 2: %d\n", search(hashTable, 2));
    printf("Value for key 3: %d\n", search(hashTable, 3));
    printf("Value for key 4: %d\n", search(hashTable, 4));
    printf("Value for key 5: %d\n", search(hashTable, 5));
    printf("Value for key 6: %d\n", search(hashTable, 6));
    delete(hashTable, 3);
    printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));
    freeTable(hashTable);
    return 0;
}
```

Rajalakshmi Engineering College

Rajalakshmi Nagar Thandalam, Chennai - 602 105.

Phone : +91-44-67181111, 67181112