

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 | 11 |  |
| 3 | Applications of Singly Linked List (Polynomial Manipulation) | 8/3/24 | 17 |  |
| 4 | Implementation of Stack using Array and Linked List implementation | 15/3/24 | 21 |  |
| 5 | Applications of Stack (Infix to Postfix) | 5/4/24 | 24 |  |
| 6 | Applications of Stack (Evaluating Arithmetic Expression) | 12/4/24 | 27 |  |
| 7 | Implementation of Queue using Array and Linked List implementation | 19/4/24 | 29 |  |
| 8 | Performing Tree Traversal Techniques | 17/5/24 | 31 |  |
| 9 | Implementation of Binary Search Tree | 17/5/24 | 34 |  |
| 10 | Implementation of AVL Tree | 24/5/24 | 38 |  |
| 11 | Implementation of BFS, DFS | 24/5/24 | 41 |  |
| 12 | Performing Topological Sorting | 24/5/24 | 45 |  |
| 13 | Implementation of Prim’s Algorithm | 31/5/24 | 47 |  |
| 14 | Implementation of Dijkstra’s Algorithm | 31/5/24 | 49 |  |
| 15 | Program to perform Sorting | 31/5/24 | 51 |  |
| 16 | Implementation of Collision Resolution Techniques | 31/5/24 | 54 |  |

|  |  |  |
| --- | --- | --- |
| 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.**

1. **Insert a node in the beginning of a list.**
2. **Insert a node after P**
3. **Insert a node at the end of a list**
4. **Find an element in a list**
5. **FindNext**
6. **FindPrevious**
7. **isLast**
8. **isEmpty**
9. **Delete a node in the beginning of a list.**
10. **Delete a node after P**
11. **Delete a node at the end of a list**
12. **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;

    }

    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;

}

**OUTPUT**

A white sheet with black text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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.**

1. **Insertion**
2. **Deletion**
3. **Search**
4. **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;

        }

    }

}

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;

}

**OUTPUT**

A white paper with black text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Ex. No.: 3 | **Polynomial Manipulation** | Date: 8/3/24 |

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

1. **Polynomial Addition**
2. **Polynomial Subtraction**
3. **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;}

**OUTPUT**

A white background with black numbers

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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.**

1. **Push an element into a stack**
2. **Pop an element from a stack**
3. **Return the Top most element from a stack**
4. **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;

}

**OUTPUT**

A white paper with black text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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;

}

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;

}

**OUTPUT**

A black text on a white background

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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;

}

**OUTPUT**



|  |  |  |
| --- | --- | --- |
| 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.**

1. **Enqueue**
2. **Dequeue**
3. **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;

}

**OUTPUT**

A white background with black text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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.**

1. **Inorder Traversal**
2. **Preorder Traversal**
3. **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;

                }

            }

        }

    }

}

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;

}

**OUTPUT**



|  |  |  |
| --- | --- | --- |
| 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.**

1. **Insert**
2. **Delete**
3. **Search**
4. **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;

}

**OUTPUT**

A screenshot of a computer screen

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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;

}

**OUTPUT**

A white background with black numbers

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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);

}

**OUTPUT**



|  |  |  |
| --- | --- | --- |
| 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;

    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;

}

**OUTPUT**



|  |  |  |
| --- | --- | --- |
| 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 = 5;

    int graph[MAX\_VERTICES][MAX\_VERTICES] = {

        {0, 2, 0, 6, 0},

        {2, 0, 3, 8, 5},

        {0, 3, 0, 0, 7},

        {6, 8, 0, 0, 9},

        {0, 5, 7, 9, 0}

    };

    primMST(graph, vertices);

    return 0;

}

**OUTPUT**

A number of numbers and a weight

Description automatically generated with medium confidence

|  |  |  |
| --- | --- | --- |
| 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 = 5;

    int graph[MAX\_VERTICES][MAX\_VERTICES] = {

        {0, 10, INT\_MAX, 30, 100},

        {10, 0, 50, INT\_MAX, INT\_MAX},

        {INT\_MAX, 50, 0, 20, 10},

        {30, INT\_MAX, 20, 0, 60},

        {100, INT\_MAX, 10, 60, 0}

    };

    int source = 0;

    dijkstra(graph, source, vertices);

    return 0;

}

**OUTPUT**

A screen shot of a computer

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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;

}

**OUTPUT**

A number on a white background

Description automatically generated

**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];

        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;

}

**OUTPUT**

A number and text on a white background

Description automatically generated

|  |  |  |
| --- | --- | --- |
| 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.**

1. **Open addressing**
2. **Closed Addressing**
3. **Rehashing**

**Algorithm:**

#include <stdio.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

typedef struct HashTable {

    int size;

    int count;

    int\* keys;

    int\* values;

    bool\* isOccupied;

} 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);

    }

    int hashIndex = hashFunction(key, hashTable->size);

    int originalIndex = hashIndex;

    int i = 1;

    while (hashTable->isOccupied[hashIndex]) {

        if (hashTable->keys[hashIndex] == key) {

            hashTable->values[hashIndex] = value;

            return;

        }

        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];

        }

        hashIndex = (originalIndex + i) % hashTable->size;

        i++;

        if (hashIndex == originalIndex) {

            break;

        }

    }

    return -1;

}

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;

        }

        hashIndex = (originalIndex + i) % hashTable->size;

        i++;

        if (hashIndex == originalIndex) {

            break;

        }

    }

}

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);

    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;

}

**OUTPUT**

A screenshot of a computer program

Description automatically generated

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;

}

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));

    delete(hashTable, 2);

    printf("Value for key 2 after deletion: %d\n", search(hashTable, 2));

    freeTable(hashTable);

    return 0;

}

**OUTPUT**

A screenshot of a computer

Description automatically generated

**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;

}

**OUTPUT**

A screenshot of a computer

Description automatically generated



Rajalakshmi Engineering College

Rajalakshmi Nagar Thandalam, Chennai - 602 105.

Phone : +91-44-67181111, 67181112