|  |  |
| --- | --- |
| Ex. No.: 1 | **Implementation of Single Linked List** |

**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 <stdlib.h>**

**struct node {**

**int data;**

**struct node \*next;**

**};**

**void insertAtBeginning(int data);**

**void insertAfter(int data, int position);**

**void insertAtEnd(int data);**

**struct node\* findElement(int data);**

**struct node\* findNext(int data);**

**struct node\* findPrevious(int data);**

**int isLast(struct node \*n);**

**int isEmpty();**

**void deleteFromBeginning();**

**void deleteAfter(int position);**

**void deleteFromEnd();**

**void deleteList();**

**void displayList();**

**struct node \*head = NULL;**

**int main() {**

**int choice, data, position;**

**while (1) {**

**printf("\n1. Insert at Beginning\n");**

**printf("2. Insert After Position\n");**

**printf("3. Insert at End\n");**

**printf("4. Find Element\n");**

**printf("5. Find Next\n");**

**printf("6. Find Previous\n");**

**printf("7. Is Last\n");**

**printf("8. Is Empty\n");**

**printf("9. Delete from Beginning\n");**

**printf("10. Delete After Position\n");**

**printf("11. Delete from End\n");**

**printf("12. Delete List\n");**

**printf("13. Display List\n");**

**printf("14. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter data to insert at beginning: ");**

**scanf("%d", &data);**

**insertAtBeginning(data);**

**break;**

**case 2:**

**printf("Enter data to insert: ");**

**scanf("%d", &data);**

**printf("Enter position after which to insert: ");**

**scanf("%d", &position);**

**insertAfter(data, position);**

**break;**

**case 3:**

**printf("Enter data to insert at end: ");**

**scanf("%d", &data);**

**insertAtEnd(data);**

**break;**

**case 4:**

**printf("Enter data to find: ");**

**scanf("%d", &data);**

**struct node\* found = findElement(data);**

**if (found) {**

**printf("Element %d found.\n", data);**

**} else {**

**printf("Element %d not found.\n", data);**

**}**

**break;**

**case 5:**

**printf("Enter data to find next: ");**

**scanf("%d", &data);**

**struct node\* next = findNext(data);**

**if (next) {**

**printf("Next element is %d.\n", next->data);**

**} else {**

**printf("No next element found.\n");**

**}**

**break;**

**case 6:**

**printf("Enter data to find previous: ");**

**scanf("%d", &data);**

**struct node\* prev = findPrevious(data);**

**if (prev) {**

**printf("Previous element is %d.\n", prev->data);**

**} else {**

**printf("No previous element found.\n");**

**}**

**break;**

**case 7:**

**printf("Enter data to check if it's the last element: ");**

**scanf("%d", &data);**

**struct node\* element = findElement(data);**

**if (element && isLast(element)) {**

**printf("Element %d is the last element.\n", data);**

**} else {**

**printf("Element %d is not the last element.\n", data);**

**}**

**break;**

**case 8:**

**if (isEmpty()) {**

**printf("List is empty.\n");**

**} else {**

**printf("List is not empty.\n");**

**}**

**break;**

**case 9:**

**deleteFromBeginning();**

**break;**

**case 10:**

**printf("Enter position after which to delete: ");**

**scanf("%d", &position);**

**deleteAfter(position);**

**break;**

**case 11:**

**deleteFromEnd();**

**break;**

**case 12:**

**deleteList();**

**break;**

**case 13:**

**displayList();**

**break;**

**case 14:**

**exit(0);**

**default:**

**printf("Invalid choice!\n");**

**}**

**}**

**return 0;**

**}**

**void insertAtBeginning(int data) {**

**struct node \*newNode = (struct node \*)malloc(sizeof(struct node));**

**if (newNode == NULL) {**

**printf("Unable to allocate memory.\n");**

**exit(0);**

**}**

**newNode->data = data;**

**newNode->next = head;**

**head = newNode;**

**displayList();**

**}**

**void insertAfter(int data, int position) {**

**struct node \*newNode = (struct node \*)malloc(sizeof(struct node));**

**if (newNode == NULL) {**

**printf("Unable to allocate memory.\n");**

**exit(0);**

**}**

**newNode->data = data;**

**struct node \*temp = head;**

**for (int i = 1; temp != NULL && i < position; i++) {**

**temp = temp->next;**

**}**

**if (temp == NULL) {**

**printf("Position does not exist.\n");**

**return;**

**}**

**newNode->next = temp->next;**

**temp->next = newNode;**

**displayList();**

**}**

**void insertAtEnd(int data) {**

**struct node \*newNode = (struct node \*)malloc(sizeof(struct node));**

**if (newNode == NULL) {**

**printf("Unable to allocate memory.\n");**

**exit(0);**

**}**

**newNode->data = data;**

**newNode->next = NULL;**

**if (head == NULL) {**

**head = newNode;**

**} else {**

**struct node \*temp = head;**

**while (temp->next != NULL) {**

**temp = temp->next;**

**}**

**temp->next = newNode;**

**}**

**displayList();**

**}**

**struct node\* findElement(int data) {**

**struct node \*temp = head;**

**while (temp != NULL) {**

**if (temp->data == data) {**

**return temp;**

**}**

**temp = temp->next;**

**}**

**return NULL;**

**}**

**struct node\* findNext(int data) {**

**struct node \*temp = findElement(data);**

**if (temp != NULL && temp->next != NULL) {**

**return temp->next;**

**}**

**return NULL;**

**}**

**struct node\* findPrevious(int data) {**

**if (head == NULL || head->data == data) {**

**return NULL;**

**}**

**struct node \*temp = head;**

**while (temp->next != NULL && temp->next->data != data) {**

**temp = temp->next;**

**}**

**if (temp->next == NULL) {**

**return NULL;**

**}**

**return temp;**

**}**

**int isLast(struct node \*n) {**

**return n != NULL && n->next == NULL;**

**}**

**int isEmpty() {**

**return head == NULL;**

**}**

**void deleteFromBeginning() {**

**if (head == NULL) {**

**printf("List is empty.\n");**

**return;**

**}**

**struct node \*temp = head;**

**head = head->next;**

**free(temp);**

**displayList();**

**}**

**void deleteAfter(int position) {**

**if (head == NULL) {**

**printf("List is empty.\n");**

**return;**

**}**

**struct node \*temp = head;**

**for (int i = 1; temp != NULL && i < position; i++) {**

**temp = temp->next;**

**}**

**if (temp == NULL || temp->next == NULL) {**

**printf("Position does not exist or no element to delete.\n");**

**return;**

**}**

**struct node \*toDelete = temp->next;**

**temp->next = temp->next->next;**

**free(toDelete);**

**displayList();**

**}**

**void deleteFromEnd() {**

**if (head == NULL) {**

**printf("List 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;**

**}**

**displayList();**

**}**

**void deleteList() {**

**struct node \*temp;**

**while (head != NULL) {**

**temp = head;**

**head = head->next;**

**free(temp);**

**}**

**printf("List deleted.\n");**

**}**

**void displayList() {**

**struct node \*temp = head;**

**if (head == NULL) {**

**printf("List is empty.\n");**

**return;**

**}**

**printf("Data in the list:\n");**

**while (temp != NULL) {**

**printf("%d ", temp->data);**

**temp = temp->next;**

**}**

**printf("\n");**

**}**

|  |  |
| --- | --- |
| Ex. No.: 2 | **Implementation of Doubly Linked List** |

**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 <stdlib.h>**

**struct node {**

**int data;**

**struct node \*prev;**

**struct node \*next;**

**};**

**struct node \*head = NULL;**

**void insertAtEnd(int data);**

**void deleteFromEnd();**

**struct node\* search(int data);**

**void displayList();**

**int main() {**

**int choice, data;**

**while (1) {**

**printf("\n1. Insert at End\n");**

**printf("2. Delete from End\n");**

**printf("3. Search\n");**

**printf("4. Display List\n");**

**printf("5. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter data to insert at end: ");**

**scanf("%d", &data);**

**insertAtEnd(data);**

**break;**

**case 2:**

**deleteFromEnd();**

**break;**

**case 3:**

**printf("Enter data to search: ");**

**scanf("%d", &data);**

**struct node\* found = search(data);**

**if (found) {**

**printf("Element %d found.\n", data);**

**} else {**

**printf("Element %d not found.\n", data);**

**}**

**break;**

**case 4:**

**displayList();**

**break;**

**case 5:**

**exit(0);**

**default:**

**printf("Invalid choice!\n");**

**}**

**}**

**return 0;**

**}**

**void insertAtEnd(int data) {**

**struct node \*newNode = (struct node \*)malloc(sizeof(struct node));**

**if (newNode == NULL) {**

**printf("Unable to allocate memory.\n");**

**exit(0);**

**}**

**newNode->data = data;**

**newNode->next = NULL;**

**if (head == NULL) {**

**newNode->prev = NULL;**

**head = newNode;**

**} else {**

**struct node \*temp = head;**

**while (temp->next != NULL) {**

**temp = temp->next;**

**}**

**temp->next = newNode;**

**newNode->prev = temp;**

**}**

**displayList();**

**}**

**void deleteFromEnd() {**

**if (head == NULL) {**

**printf("List is empty.\n");**

**return;**

**}**

**struct node \*temp = head;**

**while (temp->next != NULL) {**

**temp = temp->next;**

**}**

**if (temp->prev != NULL) {**

**temp->prev->next = NULL;**

**} else {**

**head = NULL;**

**}**

**free(temp);**

**displayList();**

**}**

**struct node\* search(int data) {**

**struct node \*temp = head;**

**while (temp != NULL) {**

**if (temp->data == data) {**

**return temp;**

**}**

**temp = temp->next;**

**}**

**return NULL;**

**}**

**void displayList() {**

**struct node \*temp;**

**if (head == NULL) {**

**printf("List is empty.\n");**

**return;**

**}**

**printf("Data in the list:\n");**

**temp = head;**

**while (temp != NULL) {**

**printf("%d ", temp->data);**

**temp = temp->next;**

**}**

**printf("\n");**

**}**

|  |  |
| --- | --- |
| Exp No.:3 | **Polynomial Manipulation** |

**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 <stdlib.h>**

**struct Node {**

**int coeff;**

**int exp;**

**struct Node\* next;**

**};**

**void insertNode(struct Node\*\* head, int coeff, int exp);**

**void displayPolynomial(struct Node\* head);**

**struct Node\* addPolynomials(struct Node\* poly1, struct Node\* poly2);**

**struct Node\* subtractPolynomials(struct Node\* poly1, struct Node\* poly2);**

**struct Node\* multiplyPolynomials(struct Node\* poly1, struct Node\* poly2);**

**void addTerm(struct Node\*\* head, int coeff, int exp);**

**int main() {**

**struct Node \*poly1 = NULL, \*poly2 = NULL, \*result = NULL;**

**int choice, coeff, exp;**

**printf("Enter the first polynomial:\n");**

**while (1) {**

**printf("Enter coefficient and exponent (enter -1 -1 to end): ");**

**scanf("%d %d", &coeff, &exp);**

**if (coeff == -1 && exp == -1) break;**

**insertNode(&poly1, coeff, exp);**

**}**

**printf("Enter the second polynomial:\n");**

**while (1) {**

**printf("Enter coefficient and exponent (enter -1 -1 to end): ");**

**scanf("%d %d", &coeff, &exp);**

**if (coeff == -1 && exp == -1) break;**

**insertNode(&poly2, coeff, exp);**

**}**

**while (1) {**

**printf("\n1. Polynomial Addition\n");**

**printf("2. Polynomial Subtraction\n");**

**printf("3. Polynomial Multiplication\n");**

**printf("4. Display Polynomials\n");**

**printf("5. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**result = addPolynomials(poly1, poly2);**

**printf("Resultant Polynomial (Addition): ");**

**displayPolynomial(result);**

**break;**

**case 2:**

**result = subtractPolynomials(poly1, poly2);**

**printf("Resultant Polynomial (Subtraction): ");**

**displayPolynomial(result);**

**break;**

**case 3:**

**result = multiplyPolynomials(poly1, poly2);**

**printf("Resultant Polynomial (Multiplication): ");**

**displayPolynomial(result);**

**break;**

**case 4:**

**printf("First Polynomial: ");**

**displayPolynomial(poly1);**

**printf("Second Polynomial: ");**

**displayPolynomial(poly2);**

**break;**

**case 5:**

**exit(0);**

**default:**

**printf("Invalid choice!\n");**

**}**

**}**

**return 0;**

**}**

**void insertNode(struct Node\*\* head, int coeff, int exp) {**

**struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));**

**newNode->coeff = coeff;**

**newNode->exp = exp;**

**newNode->next = NULL;**

**if (\*head == NULL || (\*head)->exp < exp) {**

**newNode->next = \*head;**

**\*head = newNode;**

**} else {**

**struct Node\* temp = \*head;**

**while (temp->next != NULL && temp->next->exp >= exp) {**

**temp = temp->next;**

**}**

**if (temp->exp == exp) {**

**temp->coeff += coeff;**

**free(newNode);**

**} else {**

**newNode->next = temp->next;**

**temp->next = newNode;**

**}**

**}**

**}**

**void displayPolynomial(struct Node\* head) {**

**if (head == NULL) {**

**printf("0\n");**

**return;**

**}**

**while (head != NULL) {**

**printf("%dx^%d", head->coeff, head->exp);**

**head = head->next;**

**if (head != NULL) {**

**printf(" + ");**

**}**

**}**

**printf("\n");**

**}**

**struct Node\* addPolynomials(struct Node\* poly1, struct Node\* poly2) {**

**struct Node\* result = NULL;**

**while (poly1 != NULL && poly2 != NULL) {**

**if (poly1->exp > poly2->exp) {**

**insertNode(&result, poly1->coeff, poly1->exp);**

**poly1 = poly1->next;**

**} else if (poly1->exp < poly2->exp) {**

**insertNode(&result, poly2->coeff, poly2->exp);**

**poly2 = poly2->next;**

**} else {**

**int sumCoeff = poly1->coeff + poly2->coeff;**

**if (sumCoeff != 0) {**

**insertNode(&result, sumCoeff, poly1->exp);**

**}**

**poly1 = poly1->next;**

**poly2 = poly2->next;**

**}**

**}**

**while (poly1 != NULL) {**

**insertNode(&result, poly1->coeff, poly1->exp);**

**poly1 = poly1->next;**

**}**

**while (poly2 != NULL) {**

**insertNode(&result, poly2->coeff, poly2->exp);**

**poly2 = poly2->next;**

**}**

**return result;**

**}**

**struct Node\* subtractPolynomials(struct Node\* poly1, struct Node\* poly2) {**

**struct Node\* result = NULL;**

**while (poly1 != NULL && poly2 != NULL) {**

**if (poly1->exp > poly2->exp) {**

**insertNode(&result, poly1->coeff, poly1->exp);**

**poly1 = poly1->next;**

**} else if (poly1->exp < poly2->exp) {**

**insertNode(&result, -poly2->coeff, poly2->exp);**

**poly2 = poly2->next;**

**} else {**

**int diffCoeff = poly1->coeff - poly2->coeff;**

**if (diffCoeff != 0) {**

**insertNode(&result, diffCoeff, poly1->exp);**

**}**

**poly1 = poly1->next;**

**poly2 = poly2->next;**

**}**

**}**

**while (poly1 != NULL) {**

**insertNode(&result, poly1->coeff, poly1->exp);**

**poly1 = poly1->next;**

**}**

**while (poly2 != NULL) {**

**insertNode(&result, -poly2->coeff, poly2->exp);**

**poly2 = poly2->next;**

**}**

**return result;**

**}**

**void addTerm(struct Node\*\* head, int coeff, int exp) {**

**struct Node\* temp = \*head;**

**struct Node\* prev = NULL;**

**while (temp != NULL && temp->exp > exp) {**

**prev = temp;**

**temp = temp->next;**

**}**

**if (temp != NULL && temp->exp == exp) {**

**temp->coeff += coeff;**

**if (temp->coeff == 0) {**

**if (prev != NULL) {**

**prev->next = temp->next;**

**} else {**

**\*head = temp->next;**

**}**

**free(temp);**

**}**

**} else {**

**struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));**

**newNode->coeff = coeff;**

**newNode->exp = exp;**

**newNode->next = temp;**

**if (prev != NULL) {**

**prev->next = newNode;**

**} else {**

**\*head = newNode;**

**}**

**}**

**}**

**struct Node\* multiplyPolynomials(struct Node\* poly1, struct Node\* poly2) {**

**struct Node\* result = NULL;**

**if (poly1 == NULL || poly2 == NULL) {**

**return NULL;**

**}**

**for (struct Node\* p1 = poly1; p1 != NULL; p1 = p1->next) {**

**for (struct Node\* p2 = poly2; p2 != NULL; p2 = p2->next) {**

**int coeff = p1->coeff \* p2->coeff;**

**int exp = p1->exp + p2->exp;**

**addTerm(&result, coeff, exp);**

**}**

**}**

**return result;**

**}**

|  |  |
| --- | --- |
| Exp No.: 4 | **Implementation of Stack using Array and Linked List Implementation** |

**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:**

**Stack using array implementation:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX 100**

**typedef struct {**

**int data[MAX];**

**int top;**

**} Stack;**

**void initStack(Stack\* s) {**

**s->top = -1;**

**}**

**int isFull(Stack\* s) {**

**return s->top == MAX - 1;**

**}**

**int isEmpty(Stack\* s) {**

**return s->top == -1;**

**}**

**void push(Stack\* s, int item) {**

**if (isFull(s)) {**

**printf("Stack is full.\n");**

**return;**

**}**

**s->data[++s->top] = item;**

**}**

**int pop(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return -1;**

**}**

**return s->data[s->top--];**

**}**

**int top(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return -1;**

**}**

**return s->data[s->top];**

**}**

**void display(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return;**

**}**

**for (int i = s->top; i >= 0; i--) {**

**printf("%d ", s->data[i]);**

**}**

**printf("\n");**

**}**

**int main() {**

**Stack s;**

**int choice, value;**

**initStack(&s);**

**while (1) {**

**printf("\n1. Push\n2. Pop\n3. Top\n4. Display\n5. Exit\nEnter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter value to push: ");**

**scanf("%d", &value);**

**push(&s, value);**

**break;**

**case 2:**

**printf("Popped value: %d\n", pop(&s));**

**break;**

**case 3:**

**printf("Top value: %d\n", top(&s));**

**break;**

**case 4:**

**display(&s);**

**break;**

**case 5:**

**exit(0);**

**default:**

**printf("Invalid choice.\n");**

**}**

**}**

**return 0;**

**}**

**Stack using linked list implementation:**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

**typedef struct {**

**struct Node\* top;**

**} Stack;**

**void initStack(Stack\* s) {**

**s->top = NULL;**

**}**

**int isEmpty(Stack\* s) {**

**return s->top == NULL;**

**}**

**void push(Stack\* s, int item) {**

**struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));**

**if (newNode == NULL) {**

**printf("Stack is full.\n");**

**return;**

**}**

**newNode->data = item;**

**newNode->next = s->top;**

**s->top = newNode;**

**}**

**int pop(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return -1;**

**}**

**struct Node\* temp = s->top;**

**int poppedValue = temp->data;**

**s->top = s->top->next;**

**free(temp);**

**return poppedValue;**

**}**

**int top(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return -1;**

**}**

**return s->top->data;**

**}**

**void display(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return;**

**}**

**struct Node\* temp = s->top;**

**while (temp != NULL) {**

**printf("%d ", temp->data);**

**temp = temp->next;**

**}**

**printf("\n");**

**}**

**int main() {**

**Stack s;**

**int choice, value;**

**initStack(&s);**

**while (1) {**

**printf("\n1. Push\n2. Pop\n3. Top\n4. Display\n5. Exit\nEnter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter value to push: ");**

**scanf("%d", &value);**

**push(&s, value);**

**break;**

**case 2:**

**printf("Popped value: %d\n", pop(&s));**

**break;**

**case 3:**

**printf("Top value: %d\n", top(&s));**

**break;**

**case 4:**

**display(&s);**

**break;**

**case 5:**

**exit(0);**

**default:**

**printf("Invalid choice.\n");**

**}**

**}**

**return 0;**

**}**

|  |  |
| --- | --- |
| Exp No.: 5 | **Infix to Postfix Conversion** |

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

**Algorithm:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <ctype.h>**

**#define MAX 100**

**typedef struct {**

**char data[MAX];**

**int top;**

**} Stack;**

**void initStack(Stack\* s) {**

**s->top = -1;**

**}**

**int isFull(Stack\* s) {**

**return s->top == MAX - 1;**

**}**

**int isEmpty(Stack\* s) {**

**return s->top == -1;**

**}**

**void push(Stack\* s, char item) {**

**if (isFull(s)) {**

**printf("Stack is full.\n");**

**return;**

**}**

**s->data[++s->top] = item;**

**}**

**char pop(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return '\0';**

**}**

**return s->data[s->top--];**

**}**

**char peek(Stack\* s) {**

**if (isEmpty(s)) {**

**printf("Stack is empty.\n");**

**return '\0';**

**}**

**return s->data[s->top];**

**}**

**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) {**

**Stack s;**

**initStack(&s);**

**int k = 0;**

**for (int i = 0; infix[i] != '\0'; i++) {**

**if (isdigit(infix[i]) || isalpha(infix[i])) {**

**postfix[k++] = infix[i];**

**} else if (infix[i] == '(') {**

**push(&s, infix[i]);**

**} else if (infix[i] == ')') {**

**while (!isEmpty(&s) && peek(&s) != '(') {**

**postfix[k++] = pop(&s);**

**}**

**if (!isEmpty(&s) && peek(&s) != '(') {**

**printf("Invalid expression.\n");**

**return;**

**} else {**

**pop(&s);**

**}**

**} else if (isOperator(infix[i])) {**

**while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(infix[i])) {**

**postfix[k++] = pop(&s);**

**}**

**push(&s, infix[i]);**

**}**

**}**

**while (!isEmpty(&s)) {**

**postfix[k++] = pop(&s);**

**}**

**postfix[k] = '\0';**

**}**

**int main() {**

**char infix[MAX], postfix[MAX];**

**printf("Enter an infix expression: ");**

**scanf("%s", infix);**

**infixToPostfix(infix, postfix);**

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

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 6 | **Evaluating Arithmetic Expression** |

**Write a C program to evaluate Arithmetic expression using stack.**

**Algorithm:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <ctype.h>**

**// Stack structure**

**struct Stack {**

**int top;**

**unsigned capacity;**

**int\* array;**

**};**

**// Function to create a stack of given capacity.**

**struct Stack\* createStack(unsigned capacity) {**

**struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));**

**stack->capacity = capacity;**

**stack->top = -1;**

**stack->array = (int\*)malloc(stack->capacity \* sizeof(int));**

**return stack;**

**}**

**// Stack is empty when top is -1**

**int isEmpty(struct Stack\* stack) {**

**return stack->top == -1;**

**}**

**// Function to add an item to stack. It increases top by 1**

**void push(struct Stack\* stack, int item) {**

**stack->array[++stack->top] = item;**

**}**

**// Function to remove an item from stack. It decreases top by 1**

**int pop(struct Stack\* stack) {**

**if (!isEmpty(stack))**

**return stack->array[stack->top--];**

**return -1; // Underflow condition**

**}**

**// Function to return the top from stack without removing it**

**int peek(struct Stack\* stack) {**

**if (!isEmpty(stack))**

**return stack->array[stack->top];**

**return -1;**

**}**

**// Function to evaluate postfix expression**

**int evaluatePostfix(char\* expression) {**

**struct Stack\* stack = createStack(strlen(expression));**

**int i;**

**// Scan all characters one by one**

**for (i = 0; expression[i]; ++i) {**

**// If the scanned character is an operand (number here), push it to the stack.**

**if (isdigit(expression[i])) {**

**push(stack, expression[i] - '0');**

**} else {**

**// If the scanned character is an operator, pop two elements from stack, apply the operator and push the result back.**

**int val1 = pop(stack);**

**int val2 = pop(stack);**

**switch (expression[i]) {**

**case '+':**

**push(stack, val2 + val1);**

**break;**

**case '-':**

**push(stack, val2 - val1);**

**break;**

**case '\*':**

**push(stack, val2 \* val1);**

**break;**

**case '/':**

**push(stack, val2 / val1);**

**break;**

**}**

**}**

**}**

**return pop(stack);**

**}**

**int main() {**

**char expression[100];**

**printf("Enter a postfix expression: ");**

**scanf("%s", expression);**

**printf("Postfix evaluation: %d\n", evaluatePostfix(expression));**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 7 | **Implementation of Queue using Array and Linked List Implementation** |

**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:**

**Queue using array:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX 100**

**typedef struct {**

**int front, rear, size;**

**unsigned capacity;**

**int\* array;**

**} Queue;**

**Queue\* createQueue(unsigned capacity) {**

**Queue\* queue = (Queue\*) malloc(sizeof(Queue));**

**queue->capacity = capacity;**

**queue->front = queue->size = 0;**

**queue->rear = capacity - 1;**

**queue->array = (int\*) malloc(queue->capacity \* sizeof(int));**

**return queue;**

**}**

**int isFull(Queue\* queue) {**

**return (queue->size == queue->capacity);**

**}**

**int isEmpty(Queue\* queue) {**

**return (queue->size == 0);**

**}**

**void enqueue(Queue\* queue, int item) {**

**if (isFull(queue))**

**return;**

**queue->rear = (queue->rear + 1) % queue->capacity;**

**queue->array[queue->rear] = item;**

**queue->size = queue->size + 1;**

**printf("%d enqueued to queue\n", item);**

**}**

**int dequeue(Queue\* queue) {**

**if (isEmpty(queue))**

**return -1;**

**int item = queue->array[queue->front];**

**queue->front = (queue->front + 1) % queue->capacity;**

**queue->size = queue->size - 1;**

**return item;**

**}**

**void displayQueue(Queue\* queue) {**

**if (isEmpty(queue)) {**

**printf("Queue is empty\n");**

**return;**

**}**

**printf("Queue: ");**

**for (int i = queue->front; i != queue->rear; i = (i + 1) % queue->capacity)**

**printf("%d ", queue->array[i]);**

**printf("%d\n", queue->array[queue->rear]);**

**}**

**int main() {**

**Queue\* queue = createQueue(MAX);**

**enqueue(queue, 10);**

**enqueue(queue, 20);**

**enqueue(queue, 30);**

**enqueue(queue, 40);**

**displayQueue(queue);**

**printf("%d dequeued from queue\n", dequeue(queue));**

**printf("%d dequeued from queue\n", dequeue(queue));**

**displayQueue(queue);**

**return 0;**

**}**

**Queue using linked list:**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

**struct Queue {**

**struct Node \*front, \*rear;**

**};**

**struct Queue\* createQueue() {**

**struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));**

**queue->front = queue->rear = NULL;**

**return queue;**

**}**

**void enqueue(struct Queue\* queue, int item) {**

**struct Node\* temp = (struct Node\*)malloc(sizeof(struct Node));**

**temp->data = item;**

**temp->next = NULL;**

**if (queue->rear == NULL) {**

**queue->front = queue->rear = temp;**

**return;**

**}**

**queue->rear->next = temp;**

**queue->rear = temp;**

**printf("%d enqueued to queue\n", item);**

**}**

**int dequeue(struct Queue\* queue) {**

**if (queue->front == NULL)**

**return -1;**

**struct Node\* temp = queue->front;**

**queue->front = queue->front->next;**

**if (queue->front == NULL)**

**queue->rear = NULL;**

**int item = temp->data;**

**free(temp);**

**return item;**

**}**

**void displayQueue(struct Queue\* queue) {**

**struct Node\* temp = queue->front;**

**if (temp == NULL) {**

**printf("Queue is empty\n");**

**return;**

**}**

**printf("Queue: ");**

**while (temp != NULL) {**

**printf("%d ", temp->data);**

**temp = temp->next;**

**}**

**printf("\n");**

**}**

**int main() {**

**struct Queue\* queue = createQueue();**

**enqueue(queue, 10);**

**enqueue(queue, 20);**

**enqueue(queue, 30);**

**enqueue(queue, 40);**

**displayQueue(queue);**

**printf("%d dequeued from queue\n", dequeue(queue));**

**printf("%d dequeued from queue\n", dequeue(queue));**

**displayQueue(queue);**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 8 | **Tree Traversal** |

**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 <stdlib.h>**

**// Node structure**

**struct Node {**

**int data;**

**struct Node\* left;**

**struct Node\* right;**

**};**

**// Function to create a new node**

**struct Node\* createNode(int data) {**

**struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));**

**newNode->data = data;**

**newNode->left = newNode->right = NULL;**

**return newNode;**

**}**

**// Inorder traversal**

**void inorderTraversal(struct Node\* node) {**

**if (node == NULL)**

**return;**

**inorderTraversal(node->left);**

**printf("%d ", node->data);**

**inorderTraversal(node->right);**

**}**

**// Preorder traversal**

**void preorderTraversal(struct Node\* node) {**

**if (node == NULL)**

**return;**

**printf("%d ", node->data);**

**preorderTraversal(node->left);**

**preorderTraversal(node->right);**

**}**

**// Postorder traversal**

**void postorderTraversal(struct Node\* node) {**

**if (node == NULL)**

**return;**

**postorderTraversal(node->left);**

**postorderTraversal(node->right);**

**printf("%d ", node->data);**

**}**

**// Function to insert a node in the binary tree**

**struct Node\* insertNode(struct Node\* node, int data) {**

**if (node == NULL)**

**return createNode(data);**

**if (data < node->data)**

**node->left = insertNode(node->left, data);**

**else if (data > node->data)**

**node->right = insertNode(node->right, data);**

**return node;**

**}**

**int main() {**

**struct Node\* root = NULL;**

**int choice, value;**

**while (1) {**

**printf("\n1. Insert Node\n");**

**printf("2. Inorder Traversal\n");**

**printf("3. Preorder Traversal\n");**

**printf("4. Postorder Traversal\n");**

**printf("5. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter value to insert: ");**

**scanf("%d", &value);**

**root = insertNode(root, value);**

**break;**

**case 2:**

**printf("Inorder Traversal: ");**

**inorderTraversal(root);**

**printf("\n");**

**break;**

**case 3:**

**printf("Preorder Traversal: ");**

**preorderTraversal(root);**

**printf("\n");**

**break;**

**case 4:**

**printf("Postorder Traversal: ");**

**postorderTraversal(root);**

**printf("\n");**

**break;**

**case 5:**

**exit(0);**

**default:**

**printf("Invalid choice!\n");**

**}**

**}**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 9 | **Implementation of Binary Search tree** |

**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>**

**// Node structure**

**struct Node {**

**int data;**

**struct Node\* left;**

**struct Node\* right;**

**};**

**// Function to create a new node**

**struct Node\* createNode(int data) {**

**struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));**

**newNode->data = data;**

**newNode->left = newNode->right = NULL;**

**return newNode;**

**}**

**// Function to insert a node in the BST**

**struct Node\* insertNode(struct Node\* node, int data) {**

**if (node == NULL)**

**return createNode(data);**

**if (data < node->data)**

**node->left = insertNode(node->left, data);**

**else if (data > node->data)**

**node->right = insertNode(node->right, data);**

**return node;**

**}**

**// Function to find the minimum value node in the BST**

**struct Node\* findMin(struct Node\* node) {**

**while (node && node->left != NULL)**

**node = node->left;**

**return node;**

**}**

**// Function to delete a node in the BST**

**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;**

**}**

**// Function to search for a value in the BST**

**struct Node\* searchNode(struct Node\* root, int data) {**

**if (root == NULL || root->data == data)**

**return root;**

**if (data < root->data)**

**return searchNode(root->left, data);**

**return searchNode(root->right, data);**

**}**

**// Function to display the BST (Inorder Traversal)**

**void inorderTraversal(struct Node\* node) {**

**if (node == NULL)**

**return;**

**inorderTraversal(node->left);**

**printf("%d ", node->data);**

**inorderTraversal(node->right);**

**}**

**int main() {**

**struct Node\* root = NULL;**

**int choice, value;**

**while (1) {**

**printf("\n1. Insert Node\n");**

**printf("2. Delete Node\n");**

**printf("3. Search Node\n");**

**printf("4. Display BST (Inorder Traversal)\n");**

**printf("5. Exit\n");**

**printf("Enter your choice: ");**

**scanf("%d", &choice);**

**switch (choice) {**

**case 1:**

**printf("Enter value to insert: ");**

**scanf("%d", &value);**

**root = insertNode(root, value);**

**break;**

**case 2:**

**printf("Enter value to delete: ");**

**scanf("%d", &value);**

**root = deleteNode(root, value);**

**break;**

**case 3:**

**printf("Enter value to search: ");**

**scanf("%d", &value);**

**struct Node\* found = searchNode(root, value);**

**if (found != NULL)**

**printf("Value %d found in the BST.\n", value);**

**else**

**printf("Value %d not found in the BST.\n", value);**

**break;**

**case 4:**

**printf("BST Inorder Traversal: ");**

**inorderTraversal(root);**

**printf("\n");**

**break;**

**case 5:**

**exit(0);**

**default:**

**printf("Invalid choice!\n");**

**}**

**}**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 10 | **Implementation of AVL Tree** |

**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 <stdlib.h>**

**// Node structure**

**struct Node {**

**int data;**

**struct Node\* left;**

**struct Node\* right;**

**int height;**

**};**

**// Function to get the height of a node**

**int height(struct Node\* N) {**

**if (N == NULL)**

**return 0;**

**return N->height;**

**}**

**// Function to create a new node**

**struct Node\* createNode(int data) {**

**struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));**

**node->data = data;**

**node->left = node->right = NULL;**

**node->height = 1;**

**return node;**

**}**

**// Function to get the balance factor of a node**

**int getBalance(struct Node\* N) {**

**if (N == NULL)**

**return 0;**

**return height(N->left) - height(N->right);**

**}**

**// Right rotate**

**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;**

**}**

**// Left rotate**

**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;**

**}**

**// Function to insert a node into the AVL tree**

**struct Node\* insertNode(struct Node\* node, int data) {**

**if (node == NULL)**

**return createNode(data);**

**if (data < node->data)**

**node->left = insertNode(node->left, data);**

**else if (data > node->data)**

**node->right = insertNode(node->right, data);**

**else**

**return node;**

**node->height = 1 + max(height(node->left), height(node->right));**

**int balance = getBalance(node);**

**if (balance > 1 && data < node->left->data)**

**return rightRotate(node);**

**if (balance < -1 && data > node->right->data)**

**return leftRotate(node);**

**if (balance > 1 && data > node->left->data) {**

**node->left = leftRotate(node->left);**

**return rightRotate(node);**

**}**

**if (balance < -1 && data < node->right->data) {**

**node->right = rightRotate(node->right);**

**return leftRotate(node);**

**}**

**return node;**

**}**

**// Function to find the node with the minimum value in the AVL tree**

**struct Node\* findMin(struct Node\* node) {**

**struct Node\* current = node;**

**while (current->left != NULL)**

**current = current->left;**

**return current;**

**}**

**// Function to delete a node from the AVL tree**

**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) || (root->right == NULL)) {**

**struct Node\* temp = root->left ? root->left : root->right;**

**if (temp == NULL) {**

**temp = root;**

**root = NULL;**

**} else**

**\*root = \*temp;**

**free(temp);**

**} else {**

**struct Node\* temp = findMin(root->right);**

**root->data = temp->data;**

**root->right = deleteNode(root->right, temp->data);**

**}**

**}**

**if (root == NULL)**

**return root;**

**root->height = 1 + max(height(root->left), height(root->right));**

**int balance = getBalance(root);**

**if (balance > 1 && getBalance(root->left) >= 0)**

**return rightRotate(root);**

**if (balance > 1 && getBalance(root->left) < 0) {**

**root->left = leftRotate(root->left);**

**return rightRotate(root);**

**}**

**if (balance < -1 && getBalance(root->right) <= 0)**

**return leftRotate(root);**

**if (balance < -1 && getBalance(root->right) > 0) {**

**root->right = rightRotate(root->right);**

**return leftRotate(root);**

**}**

**return root;**

**}**

**// Function to perform inorder traversal of the AVL tree**

**void inorderTraversal(struct Node\* root) {**

**if (root != NULL) {**

**inorderTraversal(root->left);**

**printf("%d ", root->data);**

**inorderTraversal(root->right);**

**}**

**}**

**int max(int a, int b) {**

**return (a > b) ? a : b;**

**}**

**int main() {**

**struct Node\* root = NULL;**

**root = insertNode(root, 10);**

**root = insertNode(root, 20);**

**root = insertNode(root, 30);**

**root = insertNode(root, 40);**

**root = insertNode(root, 50);**

**root = insertNode(root, 25);**

**printf("Inorder traversal of the constructed AVL tree is:\n");**

**inorderTraversal(root);**

**root = deleteNode(root, 40);**

**printf("\nInorder traversal after deletion of 40:\n");**

**inorderTraversal(root);**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 11 | **Graph Traversal** |

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

**Algorithm:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX 100**

**// Adjacency list node**

**struct AdjListNode {**

**int dest;**

**struct AdjListNode\* next;**

**};**

**// Adjacency list**

**struct AdjList {**

**struct AdjListNode\* head;**

**};**

**// Graph structure**

**struct Graph {**

**int V;**

**struct AdjList\* array;**

**};**

**// Function to create a new adjacency list node**

**struct AdjListNode\* newAdjListNode(int dest) {**

**struct AdjListNode\* newNode = (struct AdjListNode\*)malloc(sizeof(struct AdjListNode));**

**newNode->dest = dest;**

**newNode->next = NULL;**

**return newNode;**

**}**

**// Function to create a graph of V vertices**

**struct Graph\* createGraph(int V) {**

**struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));**

**graph->V = V;**

**graph->array = (struct AdjList\*)malloc(V \* sizeof(struct AdjList));**

**for (int i = 0; i < V; ++i)**

**graph->array[i].head = NULL;**

**return graph;**

**}**

**// Function to add an edge to an undirected graph**

**void addEdge(struct Graph\* graph, int src, int dest) {**

**struct AdjListNode\* newNode = newAdjListNode(dest);**

**newNode->next = graph->array[src].head;**

**graph->array[src].head = newNode;**

**newNode = newAdjListNode(src);**

**newNode->next = graph->array[dest].head;**

**graph->array[dest].head = newNode;**

**}**

**// Function to print the adjacency list of a graph**

**void printGraph(struct Graph\* graph) {**

**for (int v = 0; v < graph->V; ++v) {**

**struct AdjListNode\* pCrawl = graph->array[v].head;**

**printf("\n Adjacency list of vertex %d\n head ", v);**

**while (pCrawl) {**

**printf("-> %d", pCrawl->dest);**

**pCrawl = pCrawl->next;**

**}**

**printf("\n");**

**}**

**}**

**// BFS function**

**void BFS(struct Graph\* graph, int startVertex) {**

**int visited[MAX] = {0};**

**int queue[MAX];**

**int front = -1;**

**int rear = -1;**

**visited[startVertex] = 1;**

**queue[++rear] = startVertex;**

**while (front != rear) {**

**int currentVertex = queue[++front];**

**printf("%d ", currentVertex);**

**struct AdjListNode\* temp = graph->array[currentVertex].head;**

**while (temp) {**

**int adjVertex = temp->dest;**

**if (!visited[adjVertex]) {**

**queue[++rear] = adjVertex;**

**visited[adjVertex] = 1;**

**}**

**temp = temp->next;**

**}**

**}**

**}**

**// DFS function**

**void DFSUtil(struct Graph\* graph, int v, int visited[]) {**

**visited[v] = 1;**

**printf("%d ", v);**

**struct AdjListNode\* temp = graph->array[v].head;**

**while (temp) {**

**int adjVertex = temp->dest;**

**if (!visited[adjVertex])**

**DFSUtil(graph, adjVertex, visited);**

**temp = temp->next;**

**}**

**}**

**void DFS(struct Graph\* graph, int startVertex) {**

**int visited[MAX] = {0};**

**DFSUtil(graph, startVertex, visited);**

**}**

**// Main function**

**int main() {**

**int V = 5;**

**struct Graph\* graph = createGraph(V);**

**addEdge(graph, 0, 1);**

**addEdge(graph, 0, 4);**

**addEdge(graph, 1, 2);**

**addEdge(graph, 1, 3);**

**addEdge(graph, 1, 4);**

**addEdge(graph, 2, 3);**

**addEdge(graph, 3, 4);**

**printf("Adjacency list of the graph:\n");**

**printGraph(graph);**

**printf("\nBFS starting from vertex 0:\n");**

**BFS(graph, 0);**

**printf("\nDFS starting from vertex 0:\n");**

**DFS(graph, 0);**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 12 | **Topological Sorting** |

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

**Algorithm:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX 100**

**// Adjacency list node**

**struct AdjListNode {**

**int dest;**

**struct AdjListNode\* next;**

**};**

**// Adjacency list**

**struct AdjList {**

**struct AdjListNode\* head;**

**};**

**// Graph structure**

**struct Graph {**

**int V;**

**struct AdjList\* array;**

**};**

**// Function to create a new adjacency list node**

**struct AdjListNode\* newAdjListNode(int dest) {**

**struct AdjListNode\* newNode = (struct AdjListNode\*)malloc(sizeof(struct AdjListNode));**

**newNode->dest = dest;**

**newNode->next = NULL;**

**return newNode;**

**}**

**// Function to create a graph of V vertices**

**struct Graph\* createGraph(int V) {**

**struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));**

**graph->V = V;**

**graph->array = (struct AdjList\*)malloc(V \* sizeof(struct AdjList));**

**for (int i = 0; i < V; ++i)**

**graph->array[i].head = NULL;**

**return graph;**

**}**

**// Function to add an edge to an undirected graph**

**void addEdge(struct Graph\* graph, int src, int dest) {**

**struct AdjListNode\* newNode = newAdjListNode(dest);**

**newNode->next = graph->array[src].head;**

**graph->array[src].head = newNode;**

**newNode = newAdjListNode(src);**

**newNode->next = graph->array[dest].head;**

**graph->array[dest].head = newNode;**

**}**

**// Function to print the adjacency list of a graph**

**void printGraph(struct Graph\* graph) {**

**for (int v = 0; v < graph->V; ++v) {**

**struct AdjListNode\* pCrawl = graph->array[v].head;**

**printf("\n Adjacency list of vertex %d\n head ", v);**

**while (pCrawl) {**

**printf("-> %d", pCrawl->dest);**

**pCrawl = pCrawl->next;**

**}**

**printf("\n");**

**}**

**}**

**// BFS function**

**void BFS(struct Graph\* graph, int startVertex) {**

**int visited[MAX] = {0};**

**int queue[MAX];**

**int front = -1;**

**int rear = -1;**

**visited[startVertex] = 1;**

**queue[++rear] = startVertex;**

**printf("BFS Order: ");**

**while (front != rear) {**

**int currentVertex = queue[++front];**

**printf("%d ", currentVertex);**

**struct AdjListNode\* temp = graph->array[currentVertex].head;**

**while (temp) {**

**int adjVertex = temp->dest;**

**if (!visited[adjVertex]) {**

**queue[++rear] = adjVertex;**

**visited[adjVertex] = 1;**

**}**

**temp = temp->next;**

**}**

**}**

**printf("\n");**

**}**

**// DFS function**

**void DFSUtil(struct Graph\* graph, int v, int visited[]) {**

**visited[v] = 1;**

**printf("%d ", v);**

**struct AdjListNode\* temp = graph->array[v].head;**

**while (temp) {**

**int adjVertex = temp->dest;**

**if (!visited[adjVertex])**

**DFSUtil(graph, adjVertex, visited);**

**temp = temp->next;**

**}**

**}**

**void DFS(struct Graph\* graph, int startVertex) {**

**int visited[MAX] = {0};**

**printf("DFS Order: ");**

**DFSUtil(graph, startVertex, visited);**

**printf("\n");**

**}**

**// Main function**

**int main() {**

**int V = 5;**

**struct Graph\* graph = createGraph(V);**

**addEdge(graph, 0, 1);**

**addEdge(graph, 0, 4);**

**addEdge(graph, 1, 2);**

**addEdge(graph, 1, 3);**

**addEdge(graph, 1, 4);**

**addEdge(graph, 2, 3);**

**addEdge(graph, 3, 4);**

**printf("Adjacency list of the graph:\n");**

**printGraph(graph);**

**BFS(graph, 0);**

**DFS(graph, 0);**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 13 | **Graph Traversal** |

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

**Algorithm:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <limits.h>**

**int minKey(int key[], int mstSet[], int V) {**

**int min = INT\_MAX, min\_index;**

**for (int v = 0; v < V; v++) {**

**if (mstSet[v] == 0 && key[v] < min) {**

**min = key[v];**

**min\_index = v;**

**}**

**}**

**return min\_index;**

**}**

**void printMST(int parent[], int graph[][V], int V) {**

**printf("Edge \tWeight\n");**

**for (int i = 1; i < V; i++)**

**printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);**

**}**

**void primMST(int graph[][V], int V) {**

**int parent[V];**

**int key[V];**

**int mstSet[V];**

**// Initialize all keys as INFINITE**

**for (int i = 0; i < V; i++) {**

**key[i] = INT\_MAX;**

**mstSet[i] = 0;**

**}**

**key[0] = 0;**

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

**for (int count = 0; count < V - 1; count++) {**

**int u = minKey(key, mstSet, V);**

**mstSet[u] = 1;**

**for (int v = 0; v < V; v++) {**

**if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {**

**parent[v] = u;**

**key[v] = graph[u][v];**

**}**

**}**

**}**

**printMST(parent, graph, V);**

**}**

**int main() {**

**int V;**

**printf("Enter the number of vertices in the graph: ");**

**scanf("%d", &V);**

**int graph[V][V];**

**printf("Enter the adjacency matrix for the graph (%d vertices x %d vertices):\n", V, V);**

**for (int i = 0; i < V; i++) {**

**for (int j = 0; j < V; j++) {**

**scanf("%d", &graph[i][j]);**

**}**

**}**

**// Print the solution**

**primMST(graph, V);**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 14 | **Graph Traversal** |

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

**Algorithm:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <limits.h>**

**int minDistance(int dist[], int sptSet[], int V) {**

**int min = INT\_MAX, min\_index;**

**for (int v = 0; v < V; v++)**

**if (sptSet[v] == 0 && dist[v] <= min)**

**min = dist[v], min\_index = v;**

**return min\_index;**

**}**

**void printSolution(int dist[], int V) {**

**printf("Vertex Distance from Source\n");**

**for (int i = 0; i < V; i++)**

**printf("%d \t\t %d\n", i, dist[i]);**

**}**

**void dijkstra(int graph[][V], int src, int V) {**

**int dist[V];**

**int sptSet[V];**

**for (int i = 0; i < V; i++)**

**dist[i] = INT\_MAX, sptSet[i] = 0;**

**dist[src] = 0;**

**for (int count = 0; count < V - 1; count++) {**

**int u = minDistance(dist, sptSet, V);**

**sptSet[u] = 1;**

**for (int v = 0; v < V; 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, V);**

**}**

**int main() {**

**int V;**

**printf("Enter the number of vertices in the graph: ");**

**scanf("%d", &V);**

**int graph[V][V];**

**printf("Enter the adjacency matrix for the graph (%d vertices x %d vertices):\n", V, V);**

**for (int i = 0; i < V; i++) {**

**for (int j = 0; j < V; j++) {**

**scanf("%d", &graph[i][j]);**

**}**

**}**

**dijkstra(graph, 0, V);**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 15 | **Sorting** |

**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**

**Algorithm:**

**#include <stdio.h>**

**#include <stdlib.h>**

**void quickSort(int arr[], int low, int high);**

**void mergeSort(int arr[], int left, int right);**

**void merge(int arr[], int left, int mid, int right);**

**void quickSort(int arr[], int low, int high) {**

**if (low < high) {**

**int pi = partition(arr, low, high);**

**quickSort(arr, low, pi - 1);**

**quickSort(arr, pi + 1, high);**

**}**

**}**

**int partition(int arr[], int low, int high) {**

**int pivot = arr[high];**

**int i = (low - 1);**

**for (int j = low; j <= high - 1; j++) {**

**if (arr[j] <= pivot) {**

**i++;**

**swap(&arr[i], &arr[j]);**

**}**

**}**

**swap(&arr[i + 1], &arr[high]);**

**return (i + 1);**

**}**

**void swap(int\* a, int\* b) {**

**int temp = \*a;**

**\*a = \*b;**

**\*b = temp;**

**}**

**void mergeSort(int arr[], int left, int right) {**

**if (left < right) {**

**int mid = left + (right - left) / 2;**

**mergeSort(arr, left, mid);**

**mergeSort(arr, mid + 1, right);**

**merge(arr, left, mid, right);**

**}**

**}**

**void merge(int arr[], int left, int mid, int right) {**

**int i, j, k;**

**int n1 = mid - left + 1;**

**int n2 = right - mid;**

**int L[n1], R[n2];**

**for (i = 0; i < n1; i++)**

**L[i] = arr[left + i];**

**for (j = 0; j < n2; j++)**

**R[j] = arr[mid + 1 + j];**

**i = 0;**

**j = 0;**

**k = left;**

**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++;**

**}**

**}**

**int main() {**

**int n;**

**printf("Enter the number of elements: ");**

**scanf("%d", &n);**

**int arr[n];**

**printf("Enter %d elements:\n", n);**

**for (int i = 0; i < n; i++) {**

**scanf("%d", &arr[i]);**

**}**

**quickSort(arr, 0, n - 1);**

**printf("\nSorted array using Quick Sort: ");**

**for (int i = 0; i < n; i++) {**

**printf("%d ", arr[i]);**

**}**

**mergeSort(arr, 0, n - 1);**

**printf("\nSorted array using Merge Sort: ");**

**for (int i = 0; i < n; i++) {**

**printf("%d ", arr[i]);**

**}**

**return 0;**

**}**

|  |  |
| --- | --- |
| Ex. No.: 16 | **Hashing** |

**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 <stdlib.h>**

**#include <string.h>**

**#define TABLE\_SIZE 10**

**struct HashEntry {**

**int key;**

**int data;**

**};**

**struct HashTable {**

**struct HashEntry \*\*table;**

**int size;**

**};**

**struct HashTable \*createHashTable(int size) {**

**struct HashTable \*hashTable = (struct HashTable \*)malloc(sizeof(struct HashTable));**

**if (hashTable == NULL) {**

**printf("Memory allocation failed!\n");**

**exit(EXIT\_FAILURE);**

**}**

**hashTable->size = size;**

**hashTable->table = (struct HashEntry \*\*)malloc(size \* sizeof(struct HashEntry \*));**

**if (hashTable->table == NULL) {**

**printf("Memory allocation failed!\n");**

**exit(EXIT\_FAILURE);**

**}**

**for (int i = 0; i < size; i++) {**

**hashTable->table[i] = NULL;**

**}**

**return hashTable;**

**}**

**int hashCode(struct HashTable \*hashTable, int key) {**

**return key % hashTable->size;**

**}**

**void insertOpenAddressing(struct HashTable \*hashTable, int key, int data) {**

**int index = hashCode(hashTable, key);**

**while (hashTable->table[index] != NULL) {**

**index = (index + 1) % hashTable->size;**

**}**

**struct HashEntry \*entry = (struct HashEntry \*)malloc(sizeof(struct HashEntry));**

**entry->key = key;**

**entry->data = data;**

**hashTable->table[index] = entry;**

**}**

**void insertClosedAddressing(struct HashTable \*hashTable, int key, int data) {**

**int index = hashCode(hashTable, key);**

**if (hashTable->table[index] == NULL) {**

**struct HashEntry \*entry = (struct HashEntry \*)malloc(sizeof(struct HashEntry));**

**entry->key = key;**

**entry->data = data;**

**hashTable->table[index] = entry;**

**} else {**

**while (hashTable->table[index] != NULL) {**

**index = (index + 1) % hashTable->size;**

**}**

**struct HashEntry \*entry = (struct HashEntry \*)malloc(sizeof(struct HashEntry));**

**entry->key = key;**

**entry->data = data;**

**hashTable->table[index] = entry;**

**}**

**}**

**void insertRehashing(struct HashTable \*hashTable, int key, int data) {**

**int index = hashCode(hashTable, key);**

**if (hashTable->table[index] == NULL) {**

**struct HashEntry \*entry = (struct HashEntry \*)malloc(sizeof(struct HashEntry));**

**entry->key = key;**

**entry->data = data;**

**hashTable->table[index] = entry;**

**} else {**

**int newIndex = (index + 1) % hashTable->size;**

**while (newIndex != index) {**

**if (hashTable->table[newIndex] == NULL) {**

**struct HashEntry \*entry = (struct HashEntry \*)malloc(sizeof(struct HashEntry));**

**entry->key = key;**

**entry->data = data;**

**hashTable->table[newIndex] = entry;**

**return;**

**}**

**newIndex = (newIndex + 1) % hashTable->size;**

**}**

**printf("Hash table is full. Rehashing required!\n");**

**}**

**}**

**void displayHashTable(struct HashTable \*hashTable) {**

**printf("Hash Table:\n");**

**for (int i = 0; i < hashTable->size; i++) {**

**if (hashTable->table[i] != NULL) {**

**printf("Index %d: Key=%d, Data=%d\n", i, hashTable->table[i]->key, hashTable->table[i]->data);**

**} else {**

**printf("Index %d: NULL\n", i);**

**}**

**}**

**}**

**void deleteHashTable(struct HashTable \*hashTable) {**

**for (int i = 0; i < hashTable->size; i++) {**

**if (hashTable->table[i] != NULL) {**

**free(hashTable->table[i]);**

**}**

**}**

**free(hashTable->table);**

**free(hashTable);**

**}**

**int main() {**

**struct HashTable \*hashTable = createHashTable(TABLE\_SIZE);**

**insertOpenAddressing(hashTable, 10, 20);**

**insertClosedAddressing(hashTable, 15, 25);**

**insertRehashing(hashTable, 20, 30);**

**displayHashTable(hashTable);**

**deleteHashTable(hashTable);**

**return 0;**