Implementation of Single Linked List

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

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

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

// Structure Declaration for the nodes
struct node {
    int data;
    struct node *next;
};

// To create a linked list
struct node *create_linked_list (struct node *head) {
    struct node *new_node, *ptr;
    int num;
    char ch;
    ch = 'Y';
    while(ch=='Y')   {
        printf("Enter the element: ");
        scanf("%d", &num);
        new_node = malloc(sizeof(struct node));
        new_node->data=num;
        if(head==NULL) {
            new_node->next = NULL;
            head = new_node;
        }
        else {
            ptr=head;
            while(ptr->next!=NULL)
```

```
                ptr=ptr->next;
            ptr->next = new_node;
            new_node->next=NULL;
        }
        printf ("Do you want to add more elements? (Y/N) : ");
        scanf ("%s", &ch);
    }
    return head;
}

// To display each element of the list
void display (struct node *head) {
    struct node *temp = head;
    while (temp != NULL) {
        printf ("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// To insert a node at the beginning of the list
struct node * Insert_Beginning (struct node *head, int x) {
    struct node *new_node;
    new_node = malloc(sizeof(struct node));
    new_node->data = x;
    new_node->next = head;
    head = new_node;
    return head;
}

// To insert a node after the specified data
struct node * Insert_After (struct node *head, int x, int y) {
    struct node *new_node, *ptr, *ptr2;
    new_node = malloc (sizeof(struct node));
    new_node->data = y;
    ptr = head;
    while (ptr->data!=x) {
        ptr = ptr->next;
    }
    ptr2 = ptr->next;
    ptr->next = new_node;
    new_node->next = ptr2;
    return head;
}

// To insert a node at the end of the list
struct node * Insert_End (struct node *head, int x) {
    struct node *new_node, *ptr;
    new_node = malloc (sizeof(struct node));
    new_node->data = x;
    ptr = head;
```

```
    while (ptr->next!=NULL) {
        ptr = ptr->next;
    }
    ptr->next = new_node;
    new_node->next = NULL;
    return head;
}

// To find a node
struct node *Find (struct node *head, int x) {
    struct node *pos;
    pos = head->next;
    while (pos->data!=x) {
        pos = pos->next;
    }
    return pos;
}

// To find the next node
struct node *Find_Next (struct node *head, int x) {
    struct node *pos;
    pos = head->next;
    while (pos->data!=x) {
        pos = pos->next;
    }
    return pos->next;
}

// To find the previous node
struct node *Find_Previous (struct node *head, int x) {
    struct node *pos, *pos2;
    pos = head->next;
    while (pos->data!=x) {
        pos2 = pos;
        pos = pos->next;
    }
    return pos2;
}

// To check if it is the last node
int Is_Last (struct node *pos) {
    if (pos->next==NULL)
        return 1;
    else
        return 0;
}

// To check if it is an empty list
int Is_Empty (struct node *head) {
    if (head->next==NULL)
        return 1;
```

```
    else
        return 0;
}

// To delete the first node of the linked list
struct node * Delete_Beginning (struct node *head) {
    head = head->next;
    return head;
}

// To delete the node after the specified data
struct node * Delete_After (struct node *head, int x) {
    struct node *ptr;
    ptr = head;
    while (ptr->data!=x) {
        ptr = ptr->next;
    }
    ptr->next = ptr->next->next;
    return head;
}

// To delete the last node of the linked list
struct node * Delete_End (struct node *head) {
    struct node *ptr, *ptr2;
    ptr = head;
    while (ptr->next!=NULL) {
        ptr2 = ptr;
        ptr = ptr->next;
    }
    ptr2->next = NULL;
    return head;
}

void Delete_List (struct node *head) {
    struct node *temp;
    while (head!=NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {

    int ch;
    int x, y;
    int z=1;

    // Creating a linked list
    struct node *linked_list;
    struct node *temp;
```

```
    linked_list = create_linked_list ( linked_list );
    display ( linked_list );

    printf ("Menu: \n1. Insert a node in the beginning of the
list \n2. Insert a node after a particular node \n3. Insert a
node at the end of the list \n4. Find a node in a list \n5.
Find the next node \n6. Find the previous node \n7. Check if
it is the last node \n8. Check if the list is empty \n9.
Delete a node in the beginning of the list \n10. Delete a node
after a particular node \n11. Delete the node at the end of
the list \n12. Delete the list \n13. Quit \n");

    do {
        printf ("Choose an operation: ");
        scanf ("%d", &ch);
        switch (ch) {
            case 1:
                printf ("Enter the element to be inserted: ");
                scanf ("%d", &x);
                linked_list = Insert_Beginning (linked_list,
x);
                printf ("Resulting list: ");
                display (linked_list);
                break;
            case 2:
                printf ("Insert after: ");
                scanf ("%d", &x);
                printf ("Enter the element to be inserted: ");
                scanf ("%d", &y);
                linked_list = Insert_After (linked_list, x,
y);
                printf ("Resulting list: ");
                display (linked_list);
                break;
            case 3:
                printf ("Enter the element to be inserted: ");
                scanf ("%d", &x);
                linked_list = Insert_End (linked_list, x);
                printf ("Resulting list: ");
                display (linked_list);
                break;
            case 4:
                printf ("Enter the element to find: ");
                scanf ("%d", &x);
                temp = Find(linked_list, x);
                printf ("Required data: %d \n", temp->data);
                break;
            case 5:
                printf ("Enter the element to find after: ");
                scanf ("%d", &x);
                temp = Find_Next(linked_list, x);
```

```
                printf ("Required data: %d \n", temp->data);
                break;
            case 6:
                printf ("Enter the element to find before: ");
                scanf ("%d", &x);
                temp = Find_Previous(linked_list, x);
                printf ("Required data: %d \n", temp->data);
                break;
            case 7:
                printf ("Enter element to check if it is last:
");
                scanf ("%d", &x);
                if (Is_Last(Find(linked_list, x))==1)
                    printf ("Last Node \n");
                else
                    printf ("Not the last node \n");
                break;
            case 8:
                if (Is_Empty(linked_list)==1)
                    printf ("Empty \n");
                else
                    printf ("Not empty \n");
                break;
            case 9:
                linked_list = Delete_Beginning (linked_list);
                printf ("Resulting list: ");
                display (linked_list);
                break;
            case 10:
                printf ("Enter the element to be delete after:
");
                scanf ("%d", &x);
                linked_list = Delete_After (linked_list, x);
                printf ("Resulting list: ");
                display (linked_list);
                break;
            case 11:
                linked_list = Delete_End (linked_list);
                printf ("Resulting list: ");
                display (linked_list);
                break;
            case 12:
                Delete_List (linked_list);
                printf ("Successfully deleted \n");
                break;
            case 13:
                z = 0;
                break;
        }
    } while (z == 1);
```

```
    return 0;
}
```

Output

```
Enter the element: 10
Do you want to add more elements? (Y/N) : Y
Enter the element: 20
Do you want to add more elements? (Y/N) : Y
Enter the element: 30
Do you want to add more elements? (Y/N) : Y
Enter the element: 40
Do you want to add more elements? (Y/N) : N
10 20 30 40
Menu:
1. Insert a node in the beginning of the list
2. Insert a node after a particular node
3. Insert a node at the end of the list
4. Find a node in a list
5. Find the next node
6. Find the previous node
7. Check if it is the last node
8. Check if the list is empty
9. Delete a node in the beginning of the list
10. Delete a node after a particular node
11. Delete the node at the end of the list
12. Delete the list
13. Quit
Choose an operation: 1
Enter the element to be inserted: 50
Resulting list: 50 10 20 30 40
Choose an operation: 2
```

```
Insert after: 20
Enter the element to be inserted: 60
Resulting list: 50 10 20 60 30 40
Choose an operation: 3
Enter the element to be inserted: 70
Resulting list: 50 10 20 60 30 40 70
Choose an operation: 4
Enter the element to find: 60
Required data: 60
Choose an operation: 5
Enter the element to find after: 60
Required data: 30
Choose an operation: 6
Enter the element to find before: 60
Required data: 20
Choose an operation: 7
Enter element to check if it is last: 70
Last Node
Choose an operation: 8
Not empty
Choose an operation: 9
Resulting list: 10 20 60 30 40 70
Choose an operation: 10
Enter the element to be delete after: 60
Resulting list: 10 20 60 40 70
Choose an operation: 11
Resulting list: 10 20 60 40
Choose an operation: 12
Successfully deleted
Choose an operation: 13
```

Implementation of Doubly Linked List

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

i. Insertion
ii. Deletion
iii. Search
iv. Display

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

// Structure Declaration for the nodes
struct node {
    struct node *prev;
    int data;
    struct node *next;
};


// To create a linked list
struct node *Create_Linked_List (struct node *head) {
    struct node *new_node, *ptr;
    int num;
    char ch;
    ch = 'Y';
    while(ch=='Y') {
        printf("Enter the element: ");
        scanf("%d", &num);
        new_node = malloc(sizeof(struct node));
        new_node->data=num;
        if(head==NULL) {
            new_node->prev = NULL;
            new_node->next = NULL;
            head = new_node;
        }
        else {
            ptr=head;
            while(ptr->next!=NULL)
                ptr=ptr->next;
            ptr->next = new_node;
            new_node->prev = ptr;
            new_node->next=NULL;
        }
        printf ("Do you want to add more elements? (Y/N) : ");
        scanf ("%s", &ch);
    }
    return head;
```

```
}

// To check if it is an empty list
int Is_Empty (struct node *head) {
    if (head->next==NULL)
        return 1;
    else
        return 0;
}


// To check if it is the last node
int Is_Last (struct node *pos) {
    if (pos->next==NULL)
        return 1;
    else
        return 0;
}

// To find a node
struct node *Find (struct node *head, int x) {
    struct node *pos;
    pos = head->next;
    while (pos->data!=x)
        pos = pos->next;
    return pos;
}

// To find the previous node
struct node *Find_Previous (struct node *head, int x) {
    struct node *pos;
    pos = head->next;
    while (pos->data!=x)
        pos = pos->next;
    pos = pos->prev;
    return pos;
}

// To find the next node
struct node *Find_Next (struct node *head, int x) {
    struct node *pos;
    pos = head->next;
    while (pos->data!=x)
        pos = pos->next;
    return pos->next;
}

// To display each element of the list
void Display (struct node *head) {
    struct node *temp = head;
    while (temp != NULL) {
```

```
        printf ("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// To insert a node at the beginning of the list
struct node * Insert_Beginning (struct node *head, int x) {
    struct node *new_node;
    new_node = malloc(sizeof(struct node));
    new_node->data = x;
    new_node->prev = NULL;
    new_node->next = head;
    head->prev = new_node;
    head = new_node;
    return head;
}

// To insert a node at the end of the list
struct node * Insert_End (struct node *head, int x) {
    struct node *new_node, *ptr;
    new_node = malloc (sizeof(struct node));
    new_node->data = x;
    ptr = head;
    while (ptr->next!=NULL)
        ptr = ptr->next;
    ptr->next = new_node;
    new_node->prev = ptr;
    new_node->next = NULL;
    return head;
}

// To insert a node after the specified data
struct node * Insert_After (struct node *head, int x, int y) {
    struct node *new_node, *ptr, *ptr2;
    new_node = malloc (sizeof(struct node));
    new_node->data = y;
    ptr = head;
    while (ptr->data!=x)
        ptr = ptr->next;
    ptr2 = ptr->next;
    ptr->next = new_node;
    new_node->prev = ptr;
    new_node->next = ptr2;
    ptr2->prev = new_node;
    return head;
}

// To delete the first node of the linked list
struct node * Delete_Beginning (struct node *head) {
    struct node *ptr;
```

```
    ptr = head;
    head = head->next;
    head->prev = NULL;
    free (ptr);
    return head;
}

// To delete the last node of the linked list
struct node * Delete_End (struct node *head) {
    struct node *ptr, *ptr2;
    ptr = head;
    while (ptr->next!=NULL)
        ptr = ptr->next;
    ptr2 = ptr->prev;
    ptr2->next = NULL;
    free (ptr);
    return head;
}

// To delete the node after the specified data
struct node * Delete_Node (struct node *head, int x) {
    struct node *ptr, *ptr2, *ptr3;
    ptr = head;
    while (ptr->data!=x)
        ptr = ptr->next;
    ptr->prev->next = ptr->next;
    ptr->next->prev = ptr->prev;
    free (ptr);
    return head;
}

// Delete the list
void Delete_List (struct node *head) {
    struct node *temp;
    while (head!=NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {

    int ch, x, y, z;
    struct node *temp;

    // Creating a linked list
    struct node *linked_list;
    linked_list = Create_Linked_List ( linked_list );
    Display ( linked_list );
```

```
    // Menu
    printf ("Menu: \n1. Insert at beginning \n2. Insert at End
\n3. Insert after\n4. Delete at beginning \n5. Delete at end
\n6. Delete node \n7. Find node \n8. Find previous node \n9.
Find next node \n10. Empty list? \n11. Last element? \n12.
Delete list \n13. Quit \n");

    z=1;
    do {
        printf ("Choose an operation: ");
        scanf ("%d", &ch);
        switch (ch) {
            case 1:
                printf ("Enter the element to be inserted: ");
                scanf ("%d", &x);
                linked_list = Insert_Beginning (linked_list,
x);
                printf ("Resulting list: ");
                Display (linked_list);
                break;
            case 2:
                printf ("Enter the element to be inserted: ");
                scanf ("%d", &x);
                linked_list = Insert_End (linked_list, x);
                printf ("Resulting list: ");
                Display (linked_list);
                break;
            case 3:
                printf ("Insert after: ");
                scanf ("%d", &x);
                printf ("Enter the element to be inserted: ");
                scanf ("%d", &y);
                linked_list = Insert_After (linked_list, x,
y);
                printf ("Resulting list: ");
                Display (linked_list);
                break;
            case 4:
                linked_list = Delete_Beginning (linked_list);
                printf ("Resulting list: ");
                Display (linked_list);
                break;
            case 5:
                linked_list = Delete_End (linked_list);
                printf ("Resulting list: ");
                Display (linked_list);
                break;
            case 6:
                printf ("Enter the element to delete: ");
                scanf ("%d", &x);
                linked_list = Delete_Node (linked_list, x);
```

```
                printf ("Resulting list: ");
                Display (linked_list);
                break;
            case 7:
                printf ("Enter the element to find: ");
                scanf ("%d", &x);
                temp = Find(linked_list, x);
                printf ("Required data: %d \n", temp->data);
                break;
            case 8:
                printf ("Enter the element to find before: ");
                scanf ("%d", &x);
                temp = Find_Previous(linked_list, x);
                printf ("Required data: %d \n", temp->data);
                break;
            case 9:
                printf ("Enter the element to find after: ");
                scanf ("%d", &x);
                temp = Find_Next(linked_list, x);
                printf ("Required data: %d \n", temp->data);
                break;
            case 10:
                if (Is_Empty(linked_list)==1)
                    printf ("Empty \n");
                else
                    printf ("Not empty \n");
                break;
            case 11:
                printf ("Enter element to check if it is last:
");
                scanf ("%d", &x);
                if (Is_Last(Find(linked_list, x))==1)
                    printf ("Last Node \n");
                else
                    printf ("Not the last node \n");
                break;
            case 12:
                Delete_List (linked_list);
                printf ("Successfully deleted \n");
                break;
            case 13:
                z = 0;
                break;
        }
    } while (z == 1);

    return 0;
}
```

Output

```
Enter the element: 10
Do you want to add more elements? (Y/N) : Y
Enter the element: 20
Do you want to add more elements? (Y/N) : Y
Enter the element: 30
Do you want to add more elements? (Y/N) : Y
Enter the element: 40
Do you want to add more elements? (Y/N) : N
10 20 30 40
Menu:
1. Insert at beginning
2. Insert at End
3. Insert after
4. Delete at beginning
5. Delete at end
6. Delete node
7. Find node
8. Find previous node
9. Find next node
10. Empty list?
11. Last element?
12. Delete list
13. Quit
Choose an operation: 1
Enter the element to be inserted: 50
Resulting list: 50 10 20 30 40
Choose an operation: 2
Enter the element to be inserted: 60
Resulting list: 50 10 20 30 40 60
Choose an operation: 3
```

```
Insert after: 20
Enter the element to be inserted: 70
Resulting list: 50 10 20 70 30 40 60
Choose an operation: 4
Resulting list: 10 20 70 30 40 60
Choose an operation: 5
Resulting list: 10 20 70 30 40
Choose an operation: 6
Enter the element to delete: 20
Resulting list: 10 70 30 40
Choose an operation: 7
Enter the element to find: 70
Required data: 70
Choose an operation: 8
Enter the element to find before: 30
30
Required data: 70
Choose an operation: 9
Enter the element to find after: 10
Required data: 70
Choose an operation: 10
Not empty
Choose an operation: 11
Enter element to check if it is last: 40
Last Node
Choose an operation: 12
Successfully deleted
Choose an operation: 13
```

Polynomial Manipulation

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

i. Polynomial Addition
ii. Polynomial Subtraction
iii. Polynomial Multiplication

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

struct Node {
    int coeff;
    int exp;
    struct Node * next;
}* poly1 = NULL,*poly2=NULL,*result=NULL;

void create1() {
    struct Node * t, * last = NULL;
    int num, i;

    printf("Enter number of terms: ");
    scanf("%d", & num);
    printf("Enter each term with coeff and exp:\n");

    for (i = 0; i < num; i++) {
        t = (struct Node * ) malloc(sizeof(struct Node));
        scanf("%d%d", & t -> coeff, & t -> exp);
        t -> next = NULL;
        if (poly1 == NULL) {
            poly1 = last = t;
        } else {
```

```
            last -> next = t;
            last = t;
        }
    }
}

void create2() {
    struct Node * t, * last = NULL;
    int num, i;

    printf("Enter number of terms: ");
    scanf("%d", & num);
    printf("Enter each term with coeff and exp:\n");

    for (i = 0; i < num; i++) {
        t = (struct Node * ) malloc(sizeof(struct Node));
        scanf("%d%d", & t -> coeff, & t -> exp);
        t -> next = NULL;
        if (poly2 == NULL) {
            poly2 = last = t;
        } else {
            last -> next = t;
            last = t;
        }
    }
}


void Display(struct Node * p) {
    printf("(%dx^%d) ", p -> coeff, p -> exp);
    p = p -> next;
```

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

void normalize(){
    struct Node *ptr=result;
    while(ptr){
        struct Node *temp=ptr;
        while(temp->next){
            struct Node*erase;
            if(ptr->exp==temp->next->exp){
                ptr->coeff=ptr->coeff+temp->next->coeff;
                erase=temp->next;
                temp->next=temp->next->next;
                free(erase);
            }
            temp=temp->next;
        }
        ptr=ptr->next;
    }
}

void add(struct Node * p1, struct Node * p2) {
    struct Node * t, * last = NULL;
    int num1,num2;
    int p;
    if(p1->exp>p2->exp)
        p=p1->exp;
    else
```

```
    p=p2->exp;

while (p) {
    p--;
    if(p1!=NULL && p2!= NULL) {
        t = (struct Node * ) malloc(sizeof(struct Node));
        if(p1->exp==p2->exp) {
            num1=p1->exp;
            num2=p1->coeff+p2->coeff;
            p1 = p1 -> next;
            p2 = p2 -> next;
        }
        else if(p1->exp>p2->exp) {
            num1=p1->exp;
            num2=p1->coeff;
            p1 = p1 -> next;
        }
        else if(p1->exp<p2->exp) {
            num1=p2->exp;
            num2=p2->coeff;
            p2 = p2 -> next;
        }
        t->exp=num1;
        t->coeff=num2;

        t -> next = NULL;
        if (result == NULL) {
            result = last = t;
        } else {
            last -> next = t;
            last = t;
        }
```

```
}
if(p1!=NULL && p2==NULL) {
    t = (struct Node * ) malloc(sizeof(struct Node));
    num1=p1->exp;
    num2=p1->coeff;
    p1 = p1 -> next;
    t->exp=num1;
    t->coeff=num2;

    t -> next = NULL;
    if (result == NULL) {
        result = last = t;
    } else {
        last -> next = t;
        last = t;
    }
}
if(p1==NULL && p2!=NULL) {
    t = (struct Node * ) malloc(sizeof(struct Node));
    num1=p2->exp;
    num2=p2->coeff;
    p2 = p2 -> next;
    t->exp=num1;
    t->coeff=num2;

    t -> next = NULL;
    if (result == NULL) {
        result = last = t;
    } else {
        last -> next = t;
        last = t;
    }
```

```
        }
    }
    normalize(result);
    Display(result);
    result=NULL;
}

void sub(struct Node * p1, struct Node * p2) {
    struct Node * t, * last = NULL;
    int num1,num2;
    int p;
    if(p1->exp>p2->exp)
        p=p1->exp;
    else
        p=p2->exp;

    while (p) {
        p--;
        if(p1!=NULL && p2!= NULL) {
            t = (struct Node * ) malloc(sizeof(struct Node));
            if(p1->exp==p2->exp) {
                num1=p1->exp;
                num2=p1->coeff-p2->coeff;
                p1 = p1 -> next;
                p2 = p2 -> next;
            }
            else if(p1->exp>p2->exp) {
                num1=p1->exp;
                num2= - p1->coeff;
                p1 = p1 -> next;
            }
            else if(p1->exp<p2->exp) {
```

```
            num1=p2->exp;
            num2= - p2->coeff;
            p2 = p2 -> next;
        }
        t->exp=num1;
        t->coeff=num2;

        t -> next = NULL;
        if (result == NULL) {
            result = last = t;
        } else {
            last -> next = t;
            last = t;
        }
    }
    if(p1!=NULL && p2==NULL) {
        t = (struct Node * ) malloc(sizeof(struct Node));
        num1=p1->exp;
        num2=p1->coeff;
        p1 = p1 -> next;
        t->exp=num1;
        t->coeff=num2;

        t -> next = NULL;
        if (result == NULL) {
            result = last = t;
        } else {
            last -> next = t;
            last = t;
        }
    }
    if(p1==NULL && p2!=NULL) {
```

```
            t = (struct Node * ) malloc(sizeof(struct Node));
            num1=p2->exp;
            num2= - p2->coeff;
            p2 = p2 -> next;
            t->exp=num1;
            t->coeff=num2;

            t -> next = NULL;
            if (result == NULL) {
                result = last = t;
            } else {
                last -> next = t;
                last = t;
            }
        }
    }
    normalize(result);
    Display(result);
    result=NULL;
}

void mul(struct Node*p1,struct Node*p2) {
    int i=1;
    struct Node *t, *last=NULL;
    while(p1) {
        struct Node*temp=p2;
        while(temp) {
            t=malloc(sizeof(struct Node));
            t->coeff=(temp->coeff) * (p1->coeff);
            t->exp=temp->exp+p1->exp;
            if (result == NULL) {
                result = last = t;
```

```
            } else {
                last -> next = t;
                last = t;
            }
            temp=temp->next;
        }
        p1=p1->next;
    }
    normalize(result);
    Display(result);
    result=NULL;
}

int main() {
    int x;
    printf("enter first polynomial:\n");
    create1();
    Display(poly1);
    create2();
    Display(poly2);
    printf("\nADD: ");
    add(poly1,poly2);
    printf("\nSUB: ");
    sub(poly1,poly2);
    printf("\nMUL: ");
    mul(poly1,poly2);
    return 0;
}
```

Output

```
enter first polynomial:
Enter number of terms: 3
```

```
Enter each term with coeff and exp:
5 3
4 2
2 0
(5x^3) + (4x^2) + (2x^0)
Enter number of terms: 3
Enter each term with coeff and exp:
3 2
1 1
6 0
(3x^2) + (1x^1) + (6x^0)
ADD: (5x^3) + (7x^2) + (1x^1) + (8x^0)
SUB: (5x^3) + (1x^2) + (-1x^1) + (-4x^0)
MUL: (15x^5) + (20x^4) + (23x^3) + (22x^2) + (2x^1) + (12x^0)
```

Implementation of Stack

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

i. Push an element into a stack
ii. Pop an element from a stack
iii. Return the Top most element from a stack
iv. Display the elements in a stack

Array Implementation

```
#include <stdio.h>

#define MAX 10
int stack[MAX], top=-1;

// To check if the stack is full
int Is_Full () {
    if (top == MAX-1)
        return 1;
    else
        return 0;
}

// To check if the stack is empty
int Is_Empty () {
    if (top == -1)
        return 1;
    else
        return 0;
}

// To push an element into the stack
void Push (int x) {
    if (Is_Full())
        printf ("Stack overflow \n");
    else {
        top++;
        stack[top] = x;
    }
}

// To pop an element out of the stack
void Pop () {
    if (Is_Empty()) {
        printf ("Stack underflow \n");
    }
    else {
```

```
        printf ("%d \n", stack[top]);
        top--;
    }
}

// To display the element on the top of the stack
void Top () {
    if (Is_Empty())
        printf ("Stack underflow \n");
    else
        printf ("%d \n", stack[top]);
}

// To display the stack
void Display () {
    if (Is_Empty())
        printf ("Stack underflow \n");
    else {
        for (int i=top; i>=0; i--)
            printf ("%d ", stack[i]);
        printf ("\n");
    }
}

int main() {

    int x=1, y, z;

    // Menu
    printf ("Menu: \n1. Push \n2. Pop \n3. Top \n4. Display \n5.
Quit \n");

    while ( x==1 ) {
        printf ("Choose an operation: ");
        scanf ("%d", &y);
        switch (y) {
            case 1:
                printf ("Element to push: ");
                scanf ("%d", &z);
                Push (z);
                printf ("Stack: ");
                Display ();
                break;
            case 2:
                Pop ();
                printf ("Stack: ");
                Display ();
                break;
            case 3:
                Top ();
                break;
            case 4:
                Display ();
                break;
            case 5:
                x=0;
```

```
                break;
            }
        }

        return 0;
}
```

Output

```
Menu:

1. Push

2. Pop

3. Top

4. Display

5. Quit

Choose an operation: 1

Element to push: 10

Stack: 10

Choose an operation: 1

Element to push: 20

Stack: 20 10

Choose an operation: 1

Element to push: 30

Stack: 30 20 10

Choose an operation: 1

Element to push: 40

Stack: 40 30 20 10

Choose an operation: 1

Element to push: 50

Stack: 50 40 30 20 10

Choose an operation: 2
```

```
50

Stack: 40 30 20 10

Choose an operation: 2

40

Stack: 30 20 10

Choose an operation: 3

30

Choose an operation: 4

30 20 10

Choose an operation: 5
```

Linked List Implementation

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

// Structure for node
struct node {
    int data;
    struct node *next;
} *stack = NULL;

// To check if the stack is empty
int Is_Empty () {
    if (stack == NULL)
        return 1;
    else
        return 0;
}

// To push an element into the stack
void Push (int x) {
    struct node *new_node;
    new_node = malloc (sizeof(struct node));
    new_node->data = x;
    if (Is_Empty())
        new_node->next = NULL;
    else
        new_node->next = stack;
    stack = new_node;
}

// To pop an element out of the stack
void Pop () {
```

```
    if (Is_Empty()) {
        printf ("Stack underflow \n");
    }
    else {
        struct node *temp;
        temp = stack;
        stack = stack->next;
        printf ("%d \n", temp->data);
        free (temp);
    }
}

// To display the element on the top of the stack
void Top () {
    if (Is_Empty())
        printf ("Stack underflow \n");
    else
        printf ("%d \n", stack->data);
}

// To display the stack
void Display () {
    if (Is_Empty())
        printf ("Stack underflow \n");
    else {
        struct node *temp;
        temp = stack;
        while (temp!=NULL) {
            printf ("%d ", temp->data);
            temp = temp->next;
        }
        printf ("\n");
    }
}

int main() {

    int x=1, y, z;

    // Menu
    printf ("Menu: \n1. Push \n2. Pop \n3. Top \n4. Display \n5.
Quit \n");

    while ( x==1 ) {
        printf ("Choose an operation: ");
        scanf ("%d", &y);
        switch (y) {
            case 1:
                printf ("Element to push: ");
                scanf ("%d", &z);
                Push (z);
                printf ("Stack: ");
                Display ();
                break;
            case 2:
                Pop ();
```

```
                printf ("Stack: ");
                Display ();
                break;
            case 3:
                Top ();
                break;
            case 4:
                Display ();
                break;
            case 5:
                x=0;
                break;
        }
    }

    return 0;
}
```

Output

```
Menu:

1. Push

2. Pop

3. Top

4. Display

5. Quit

Choose an operation: 1

Element to push: 10

Stack: 10

Choose an operation: 1

Element to push: 20

Stack: 20 10

Choose an operation: 1

Element to push: 30

Stack: 30 20 10

Choose an operation: 1
```

```
Element to push: 40

Stack: 40 30 20 10

Choose an operation: 1

Element to push: 50

Stack: 50 40 30 20 10

Choose an operation: 2

50

Stack: 40 30 20 10

Choose an operation: 2

40

Stack: 30 20 10

Choose an operation: 3

30

Choose an operation: 4

30 20 10

Choose an operation: 5
```

Infix to Postfix Conversion

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

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

#define MAX 20

int stack[MAX], top = -1;
char expr[MAX];

void Push (char x) {
    top++;
    stack[top] = x;
}

char Pop () {
    char e;
    e = stack[top];
    top--;
    return e;
}

char Top () {
    return stack[top];
}

int Priority(char sym) {
    int p=0;
    if (sym == '(')
        p=0;
    else if (sym=='+' || sym=='-')
        p=1;
    else if (sym=='*'||sym=='/'||sym=='%')
        p=2;
    else if (sym=='^')
        p=3;
    return p;
}

int main() {

    printf("Enter the infix expression: ");
    scanf ("%s", expr);
    for (int i=0; i<strlen(expr); i++) {
        if(expr[i]>='a' && expr[i]<='z')
            printf ("%c",expr[i]);
        else if (expr[i]=='(')
            Push (expr[i]);
        else if (expr[i]==')') {
            while (Top()!='(')
```

```
                printf ("%c", Pop());
            Pop();
        }
        else {
            while (Priority(expr[i])<=Priority(Top())&&top!=-1)
                printf ("%c", Pop());
            Push(expr[i]);
        }
    }
    for (int i=top; i>=0; i--)
        printf("%c", Pop());

    return 0;
}
```

Output

```
Enter the infix expression: A+B-C^D*E%F/G

+A-B^C*D%E/FG
```

Evaluation of Arithmetic Expression

Write a C program to evaluate Arithmetic expression using stack.

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

#define MAX 20

int stack[MAX], top = -1;
char expr[MAX];

void Push (int x) {
    top++;
    stack[top] = x;
}

int Pop() {
    int x;
    x = stack[top];
    top--;
    return x;
}

int main() {

    int a, b, c, x;
    printf ("Postfix Expression: ");
    scanf ("%s", expr);
    for (int i=0; i<strlen(expr); i++) {
        if (expr[i]=='+'||expr[i]=='-'||expr[i]=='*'||expr[i]=='/')
{
            b = Pop();
            a = Pop();
            if (expr[i]=='+') {
                c = a+b;
                Push (c);
            }
            else if (expr[i]=='-') {
                c = a-b;
                Push (c);
            }
            else if (expr[i]=='*') {
                c = a*b;
                Push (c);
            }
            else {
                c = a/b;
                Push (c);
            }
        }
        else {
```

```
            printf ("Enter the value of %c: ", expr[i]);
            scanf ("%d", &x);
            Push (x);
        }
    }
    printf ("Result: %d", Pop());

    return 0;
}
```

Output

```
Postfix Expression: ABC+*D*

Enter the value of A: 23

Enter the value of B: 4

Enter the value of C: 10

Enter the value of D: 55

Result: 1540
```

Implementation of Queue

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

i. Enqueue
ii. Dequeue
iii. Display the elements in a Queue

Array Implementation

```
#include <stdio.h>
#define MAX 5

int Queue[MAX], front = -1, rear = -1;

int Is_Full() {
    if (rear == MAX-1)
        return 1;
    else
        return 0;
}

int Is_Empty () {
    if (front == -1)
        return 1;
    else
        return 0;
}

void Enqueue (int x) {
    if (Is_Full())
        printf ("Queue Overflow");
```

```
    else {
        rear++;
        Queue[rear]=x;
        if (front == -1)
            front++;
    }
}

int Dequeue () {
    if (Is_Empty())
        printf ("Queue Underflow \n");
    else {
        printf ("%d \n", Queue[front]);
        if (front == rear)
            front = rear = -1;
        else
            front++;
    }
}

void Display() {
    if (Is_Empty())
        printf ("Queue Underflow \n");
    else {
        for (int i=front; i<=rear; i++)
            printf ("%d ", Queue[i]);
        printf ("\n");
    }
}

int main() {
```

```
    int x, y=1;

    // Menu
    printf ("1. Enqueue 2. Dequeue 3. Display 4. Exit \n");

    while (y==1) {
        printf ("Operation: ");
        scanf ("%d", &x);
        switch (x) {
            case 1:
                printf ("Enter the element: ");
                scanf ("%d", &x);
                Enqueue(x);
                break;
            case 2:
                Dequeue ();
                break;
            case 3:
                Display ();
                break;
            case 4:
                y = 0;
        }
    }

    return 0;
}
```

Output

```
1. Enqueue 2. Dequeue 3. Display 4. Exit
Operation: 1
Enter the element: 10
```

```
Operation: 1
Enter the element: 20
Operation: 1
Enter the element: 30
Operation: 1
Enter the element: 40
Operation: 1
Enter the element: 50
Operation: 2
10
Operation: 3
20 30 40 50
Operation: 4
```

Linked List Implementation

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

struct node {
    int data;
    struct node *next;
} *front = NULL, *rear = NULL;
typedef struct node Queue;

int Is_Empty () {
    if (front == NULL)
        return 1;
    else
        return 0;
}

void Enqueue (int x) {
```

```
    Queue *new_node = malloc(sizeof(Queue));
    new_node->data = x;
    new_node->next = NULL;
    if (rear == NULL)
        front = rear = new_node;
    else {
        rear->next = new_node;
        rear = new_node;
    }
}

int Dequeue () {
    if (Is_Empty())
        printf ("Queue Underflow \n");
    else {
        int x;
        x = front->data;
        if (front == rear)
            front = rear = NULL;
        else
            front = front->next;
        printf ("%d \n", x);
    }
}

void Display() {
    if (Is_Empty())
        printf ("Queue Underflow \n");
    else {
        Queue *ptr;
        ptr = front;
        while (ptr!=NULL) {
```

```
            printf ("%d ", ptr->data);
            ptr = ptr->next;
        }
        printf ("\n");
    }
}

int main() {

    int x, y=1;

    // Menu
    printf ("1. Enqueue 2. Dequeue 3. Display 4. Exit \n");

    while (y==1) {
        printf ("Operation: ");
        scanf ("%d", &x);
        switch (x) {
            case 1:
                printf ("Enter the element: ");
                scanf ("%d", &x);
                Enqueue (x);
                break;
            case 2:
                Dequeue ();
                break;
            case 3:
                Display ();
                break;
            case 4:
                y = 0;
        }
```

```
    }

    return 0;
}
```

Output

```
1. Enqueue 2. Dequeue 3. Display 4. Exit
Operation: 1
Enter the element: 10
Operation: 1
Enter the element: 20
Operation: 1
Enter the element: 30
Operation: 1
Enter the element: 40
Operation: 1
Enter the element: 50
Operation: 2
10
Operation: 3
20 30 40 50
Operation: 4
```

Tree Traversal

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

i. Inorder Traversal
ii. Preorder Traversal
iii. Postorder Traversal

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

struct node {
    struct node *left;
    int element;
    struct node *right;
};
typedef struct node Node;

Node *Insert(Node *Tree, int e) {
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL) {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    }
    else if (e < Tree->element)
        Tree->left = Insert(Tree->left, e);
    else if (e > Tree->element)
        Tree->right = Insert(Tree->right, e);
    return Tree;
}
```

```
void Inorder(Node *Tree) {
    if (Tree != NULL) {
        Inorder(Tree->left);
        printf("%d\t", Tree->element);
        Inorder(Tree->right);
    }
}

void Preorder(Node *Tree) {
    if (Tree != NULL) {
        printf("%d\t", Tree->element);
        Preorder(Tree->left);
        Preorder(Tree->right);
    }
}

void Postorder(Node *Tree)  {
    if (Tree != NULL) {
        Postorder(Tree->left);
        Postorder(Tree->right);
        printf("%d\t", Tree->element);
    }
}

int main() {
    Node *Tree = NULL;
    int n, i, e, ch;
    printf("Enter number of nodes in the tree : ");
    scanf("%d", &n);
    printf("Enter the elements :\n");
    for (i = 1; i <= n; i++) {
        scanf("%d", &e);
```

```
        Tree = Insert(Tree, e);
    }
    do {
        printf("1. Inorder \n2. Preorder \n3. Postorder \n4.
Exit\n");
        printf("Enter your choice : ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                Inorder(Tree);
                printf("\n");
                break;
            case 2:
                Preorder(Tree);
                printf("\n");
                break;
            case 3:
                Postorder(Tree);
                printf("\n");
                break;
        }
    } while (ch <= 3);
    return 0;
}
```

Output

```
Enter number of nodes in the tree : 5
Enter the elements :
20 40 50 30 10
1. Inorder
2. Preorder
3. Postorder
```

```
4. Exit
Enter your choice : 1
10    20    30    40    50
1. Inorder
2. Preorder
3. Postorder
4. Exit
Enter your choice : 2
20    10    40    30    50
1. Inorder
2. Preorder
3. Postorder
4. Exit
Enter your choice : 3
10    30    50    40    20
1. Inorder
2. Preorder
3. Postorder
4. Exit
Enter your choice : 4
```

Implementation of Binary Search Tree

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

i. Insert
ii. Delete
iii. Search
iv. Display

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

struct node {
    struct node *left;
    int element;
    struct node *right;
};
typedef struct node Node;

Node *Insert(Node *Tree, int e) {
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL) {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    }
    else if (e < Tree->element)
        Tree->left = Insert(Tree->left, e);
    else if (e > Tree->element)
        Tree->right = Insert(Tree->right, e);
    return Tree;
}
```

```
Node *FindMin(Node *Tree)
{
if (Tree != NULL) {
    if (Tree->left == NULL)
        return Tree;
    else
        FindMin(Tree->left);
    }
}


Node *Delete(Node *Tree, int e) {
    Node *TempNode = malloc(sizeof(Node));
    if (e < Tree->element)
        Tree->left = Delete(Tree->left, e);
    else if (e > Tree->element)
        Tree->right = Delete(Tree->right, e);
    else if (Tree->left && Tree->right) {
        TempNode = FindMin(Tree->right);
        Tree->element = TempNode->element;
        Tree->right = Delete(Tree->right, Tree->element);
    }
    else    {
        TempNode = Tree;
        if(Tree->left == NULL)
            Tree = Tree->right;
        else if (Tree->right == NULL)
            Tree = Tree->left;
        free(TempNode);
    }
    return Tree;
```

```
}

void Find(Node *Tree, int e) {
    if (Tree == NULL)
        printf("Element is not found...!\n");
    else if (e < Tree->element)
        Find(Tree->left, e);
    else if (e > Tree->element)
        Find(Tree->right, e);
    else
        printf("Element is found...!\n");
}

void Display(Node *Tree) {
    if (Tree != NULL) {
        Display(Tree->left);
        printf("%d\t", Tree->element);
        Display(Tree->right);
    }
    printf ("\n");
}

int main() {
    int x, ch;
    Node *tree = NULL;
    printf ("MENU: 1.Insert 2.Delete 3.Search 4.Display \n");
    do {
        printf ("Enter your choice: ");
        scanf ("%d", &ch);
        switch (ch) {
            case 1:
                printf ("Enter the element: ");
```

```
                scanf ("%d", &x);
                tree = Insert (tree, x);
                break;
            case 2:
                printf ("Enter the element: ");
                scanf ("%d", &x);
                tree = Delete (tree, x);
                break;
            case 3:
                printf ("Enter the element: ");
                scanf ("%d", &x);
                Find (tree, x);
                break;
            case 4:
                Display (tree);
                break;
        }
    } while (ch<5);
    return 0;
}
```

Output

```
MENU: 1.Insert 2.Delete 3.Search 4.Display
Enter your choice: 1
Enter the element: 10
Enter your choice: 1
Enter the element: 20
Enter your choice: 1
Enter the element: 30
Enter your choice: 1
Enter the element: 40
Enter your choice: 1
```

```
Enter the element: 50
Enter your choice: 4
10    20    30    40    50
Enter your choice: 2
Enter the element: 30
Enter your choice: 4
10    20    40    50
Enter your choice: 3
Enter the element: 20
Element is found...!
Enter your choice: 5
```

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.`

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

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

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

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

struct node * Min_Node (struct node *root) {
    struct node * current = root;
    while (current->left!=NULL)
        current = current->left;
    return current;
```

```
}

int Balance_Factor (struct node *x) {
    if (x==NULL)
        return 0;
    else
        return (Height(x->left) - Height(x->right));
}

struct node * Right_Rotate (struct node *x) {
    struct node *a = x->left;
    struct node *b = a->right;
    a->right = x;
    x->left = b;
    a->height = Max (Height(a->right), Height(a->left))+1;
    x->height = Max (Height(x->right), Height(x->left))+1;
    return a;
}

struct node * Left_Rotate (struct node *x) {
    struct node *a = x->right;
    struct node *b = a->left;
    a->left = x;
    x->right = b;
    a->height = Max (Height(a->right), Height(a->left))+1;
    x->height = Max (Height(x->right), Height(x->left))+1;
    return a;
}

struct node *Insert (struct node *root, int x) {

    // First Node
```

```
if (root==NULL) {
    struct node *node = malloc(sizeof(struct node));
    node->data = x;
    node->left = NULL;
    node ->right = NULL;
    node ->height = 1;
    return node;
}

if (x < root->data)
    root->left = Insert (root->left, x);
else if (x > root->data)
    root->right = Insert (root->right, x);

root->height = 1+Max(Height(root->left), Height(root->right));

// LL Rotation
if (Balance_Factor(root)>1 && x<root->left->data)
    return Right_Rotate (root);

// RR Rotation
if (Balance_Factor(root)<-1 && x>root->right->data)
    return Left_Rotate (root);

// LR Rotation
if (Balance_Factor(root)>1 && x>root->left->data) {
    root->left = Left_Rotate(root->left);
    return Right_Rotate (root);
}

// RL Rotation
if (Balance_Factor(root)<-1 && x<root->left->data) {
```

```
        root->left = Right_Rotate(root->left);
        return Left_Rotate (root);
    }

    return root;

}

struct node *Delete (struct node *root, int x) {

    if (root==NULL)
        return root;

    if (x<root->data)
        root->left = Delete (root->left, x);
    else if (x>root->data)
        root->right = Delete (root->right, x);

    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 = Min_Node (root->right);
            root->data = temp->data;
```

```
            root->right = Delete(root->right, temp->data);
        }
    }

    if (root == NULL)
    return root;

    root->height = 1 + Max(Height(root->left), Height(root->right));

    // LL Rotation
    if (Balance_Factor(root)>1 && x<root->left->data)
        return Right_Rotate (root);

    // RR Rotation
    if (Balance_Factor(root)<-1 && x>root->right->data)
        return Left_Rotate (root);

    // LR Rotation
    if (Balance_Factor(root)>1 && x>root->left->data) {
        root->left = Left_Rotate(root->left);
        return Right_Rotate (root);
    }

    // RL Rotation
    if (Balance_Factor(root)<-1 && x<root->left->data) {
        root->left = Right_Rotate(root->left);
        return Left_Rotate (root);
    }

  return root;
}
```

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

int main(){
    printf ("MENU: 1.Insert 2.Delete 3.Display 4.Exit\n");
    int x, ch;
    struct node *root=NULL;
    do {
        printf ("Enter your choice: ");
        scanf ("%d", &ch);
        switch (ch) {
            case 1:
                printf ("Enter the element to be inserted: ");
                scanf ("%d", &x);
                root = Insert (root, x);
                break;
            case 2:
                printf ("Enter the element to be deleted: ");
                scanf ("%d", &x);
                root = Delete (root, x);
                break;
            case 3:
                Preorder(root);
                break;
        }
    } while (ch<=3);
    return 0;
```

```
}
```

Output

```
MENU: 1.Insert 2.Delete 3.Display 4.Exit
Enter your choice: 1
Enter the element to be inserted: 10
Enter your choice: 1
Enter the element to be inserted: 20
Enter your choice: 1
Enter the element to be inserted: 50
Enter your choice: 1
Enter the element to be inserted: 30
Enter your choice: 2
Enter the element to be deleted: 20
Enter your choice: 3
30 10 50
Enter your choice: 4
```

Implementation of BFS and DFS

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

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

#define SIZE 40

struct queue {
    int items[SIZE];
    int front;
    int rear;
};

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

struct node* createNode(int);
struct Graph {
    int numVertices;
    struct node** adjLists;
    int* visited;
};

// Creating a node
struct node* createNode(int v) {
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
```

```
    newNode->next = NULL;
    return newNode;
}

// Creating a graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));
    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

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

// Create a queue
struct queue* createQueue() {
```

```
    struct queue* q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Check if the queue is empty
int isEmpty(struct queue* q) {
    if (q->rear == -1)
        return 1;
    else
        return 0;
}

// Adding elements into queue
void enqueue(struct queue* q, int value) {
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

// Removing elements from queue
int dequeue(struct queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty");
        item = -1;
```

```
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
    return item;
}

// Print the queue
void printQueue(struct queue* q) {
    int i = q->front;
    if (isEmpty(q)) {
        printf("Queue is empty");
    } else {
        printf("\nQueue contains \n");
        for (i = q->front; i < q->rear + 1; i++) {
            printf("%d ", q->items[i]);
        }
    }
}

void DFS(struct Graph* graph, int vertex) {
    struct node* adjList = graph->adjLists[vertex];
    struct node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while (temp != NULL) {
```

```
        int connectedVertex = temp->vertex;
        if (graph->visited[connectedVertex] == 0) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

void BFS(struct Graph* graph, int startVertex) {
    struct queue* q = createQueue();
    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while (!isEmpty(q)) {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);
        struct node* temp = graph->adjLists[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;
            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

int main() {
    struct Graph* graph = createGraph(6);
```

```
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    printf("BFS:\n");
    BFS(graph, 0);

    // Reset visited array for DFS
    for (int i = 0; i < graph->numVertices; i++) {
        graph->visited[i] = 0;
    }

    printf("\nDFS:\n");
    DFS(graph, 0);

    return 0;
}
```

Output

```
BFS:
Queue contains
0
Visited 0
Queue contains
1 2
Visited 1
Queue contains
2 4 3
```

```
Visited 2
Queue contains
4 3
Visited 4
Queue contains
3
Visited 3

DFS:
Visited 0
Visited 2
Visited 4
Visited 3
Visited 1
```

Topological Sorting

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

```
#include <stdio.h>
#define SIZE 10
#define MAX 10

int G[SIZE][SIZE], i, j, k;
int front, rear;
int n, edges;

int b[SIZE], Q[SIZE], indegree[SIZE];

int create() {
    front = -1;
    rear = -1;
    for (i = 0; i < MAX; i++) { // Initialising the graph
        for (j = 0; j < MAX; j++) {
            G[i][j] = 0;
        }
    }
    for (i = 0; i < MAX; i++) {
        indegree[i] = -99;
    }
    n = 5;
    edges = 7;
    G[0][2] = 1;
    G[0][3] = 1;
    G[1][0] = 1;
    G[1][3] = 1;
    G[2][4] = 1;
```

```
    G[3][2] = 1;
    G[3][4] = 1;
    return n;
}

void Display(int n) {
    int V1, V2;
    for (V1 = 0; V1 < n; V1++) {
        for (V2 = 0; V2 < n; V2++) {
            printf("%d", G[V1][V2]);
        }
        printf("\n");
    }
}

void Insert_Q(int vertex, int n) {
    if (rear == n) {
        printf("Queue Overflow\n");
    } else {
        if (front == -1) { // Empty Queue condition
            front = 0;
        }
        rear = rear + 1;
        Q[rear] = vertex; // Inserting node into the Q
    }
}

int Delete_Q() {
    int item;
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return -1;
```

```
    } else {
        item = Q[front];
        front = front + 1;
        return item;
    }
}

int Compute_Indeg(int node, int n) {
    int v1, indeg_count = 0;
    for (v1 = 0; v1 < n; v1++) {
        if (G[v1][node] == 1) { // Checking for incoming edge
            indeg_count++;
        }
    }
    return indeg_count;
}

void Topo_ordering(int n) {
    j = 0;
    for (i = 0; i < n; i++) {
        indegree[i] = Compute_Indeg(i, n);
        if (indegree[i] == 0) {
            Insert_Q(i, n);
        }
    }

    while (front <= rear) {
        k = Delete_Q();
        b[j++] = k;
        for (i = 0; i < n; i++) {
            if (G[k][i] == 1) {
                G[k][i] = 0;
```

```
                indegree[i] = indegree[i] - 1;
                if (indegree[i] == 0) {
                    Insert_Q(i, n);
                }
            }
        }
    }
    printf("\nThe result after topological sorting is ...");
    for (i = 0; i < n; i++) {
        printf("%d ", b[i]);
    }
    printf("\n");
}

int main() {
    n = create();
    printf("The adjacency matrix is : \n");
    Display(n);
    Topo_ordering(n);
    return 0;
}
```

Output

```
The adjacency matrix is :
00110
10010
00001
00101
00000
The result after topological sorting is ...1 0 3 2 4
```

Implementation of Prim's Algorithm

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

```
#include <stdio.h>
#define SIZE 20
#define INFINITY 32767

/* This function finds the minimal spanning tree by Prim's Algorithm
*/
void Prim(int G[][SIZE], int nodes) {
    int tree[SIZE], i, j, k;
    int min_dist, v1, v2, total = 0;

    // Initialize the selected vertices list
    for (i = 0; i < nodes; i++)
        tree[i] = 0;

    printf("\n\nThe Minimal Spanning Tree Is:\n");
    tree[0] = 1;

    for (k = 1; k <= nodes - 1; k++) {
        min_dist = INFINITY; // Initially assign minimum dist as
infinity

        for (i = 0; i <= nodes - 1; i++) {
            for (j = 0; j <= nodes - 1; j++) {
                if (G[i][j] && ((tree[i] && !tree[j]) || (!tree[i]
&& tree[j]))) {
                    if (G[i][j] < min_dist) {
                        min_dist = G[i][j];
                        v1 = i;
```

```
                        v2 = j;
                    }
                }
            }
        }

        printf("\nEdge (%d %d) and weight = %d", v1, v2, min_dist);
        tree[v1] = tree[v2] = 1;
        total = total + min_dist;
    }

    printf("\n\n\tTotal Path Length Is = %d", total);
}

void main() {
    int G[SIZE][SIZE], nodes;
    int v1, v2, length, i, j, n;

    printf("\n\tPrim's Algorithm\n");
    printf("\nEnter Number of Nodes in The Graph: ");
    scanf("%d", &nodes);

    printf("\nEnter Number of Edges in The Graph: ");
    scanf("%d", &n);

    // Initialize the graph
    for (i = 0; i < nodes; i++)
        for (j = 0; j < nodes; j++)
            G[i][j] = 0;

    // Entering weighted graph
    printf("\nEnter edges and weights\n");
```

```
    for (i = 0; i < n; i++) {
        printf("\nEnter Edge by V1 and V2: [Read the graph from
starting node 0]");
        scanf("%d %d", &v1, &v2);

        printf("\nEnter corresponding weight: ");
        scanf("%d", &length);

        G[v1][v2] = G[v2][v1] = length;
    }

    printf("\n\t");
    Prim(G, nodes);
}
```

Output

```
  Prim's Algorithm

    Enter Number of Nodes in The Graph: 4

    Enter Number of Edges in The Graph: 5

    Enter edges and weights:

    Enter Edge by V1 and V2: [Read the graph from starting node 0]
    0 1
    Enter corresponding weight:
    10

    Enter Edge by V1 and V2: [Read the graph from starting node 0]
    0 2
    Enter corresponding weight:
```

```
6

Enter Edge by V1 and V2: [Read the graph from starting node 0]
0 3
Enter corresponding weight:
5

Enter Edge by V1 and V2: [Read the graph from starting node 0]
1 3
Enter corresponding weight:
15

Enter Edge by V1 and V2: [Read the graph from starting node 0]
2 3
Enter corresponding weight:
4


The Minimal Spanning Tree Is:

Edge (0 3) and weight = 5
Edge (2 3) and weight = 4
Edge (0 1) and weight = 10

    Total Path Length Is = 19
```

Implementation of Dijkstra's Algorithm

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

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

#define MAX_VERTICES 100

// Function to find the vertex with the minimum distance value
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;
}

// Function to print the constructed distance array
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]);
    }
}

// Function to implement Dijkstra's algorithm for a given graph and source vertex
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int vertices) {
    int dist[MAX_VERTICES]; // The output array dist[i] holds the shortest distance from src to i
    int sptSet[MAX_VERTICES]; // sptSet[i] will be true if vertex i is included in the shortest path tree or the shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and sptSet[] as false
    for (int i = 0; i < vertices; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }

    // Distance from source vertex to itself is always 0
    dist[src] = 0;

    // Find the shortest path for all vertices
    for (int count = 0; count < vertices - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not yet processed.
```

```
        // u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet, vertices);

        // Mark the picked vertex as processed
        sptSet[u] = 1;

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < vertices; v++) {
            // Update dist[v] only if it is not in the sptSet, there is an edge from u to v,
            // and the total weight of path from src to v through u is smaller than the current value of dist[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];
            }
        }
    }

    // Print the constructed distance array
    printSolution(dist, vertices);
}

int main() {
    int vertices;
```

```
    // Input the number of vertices
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

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

    int graph[MAX_VERTICES][MAX_VERTICES];

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

    int source;

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

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

    // Perform Dijkstra's algorithm
    dijkstra(graph, source, vertices);

    return 0;
}
```

Output

```
Input the number of vertices: 3
Input the adjacency matrix for the graph (use INT_MAX for infinity):
2 0 5
1 0 6
0 8 0
Input the source vertex: 1
Vertex   Distance from Source
0   1
1   0
2   6
```

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.

i. Quick Sort
ii. Merge Sort

Quick Sort

```
#include <stdio.h>


void QuickSort(int a[], int left, int right) {
    int i, j, temp, pivot;
    if (left < right) {
        pivot = left;
        i = left + 1;
        j = right;
        while (i < j) {
            while (a[i] < a[pivot])
                i++;
            while (a[j] > a[pivot])
                j--;
            if (i < j) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
        temp = a[pivot];
        a[pivot] = a[j];
        a[j] = temp;
        QuickSort(a, left, j - 1);
```

```
        QuickSort(a, j + 1, right);
    }
}

int main() {
    int i, n, a[10];
    printf("Enter the limit: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    QuickSort(a, 0, n - 1);
    printf("The sorted elements are: ");
    for (i = 0; i < n; i++)
        printf("%d\t", a[i]);
    return 0;
}
```

Output

```
Enter the limit : 5
Enter the elements : 50 30 10 20 40
The sorted elements are : 10    20    30    40    50
```

Merge Sort

```
#include <stdio.h>

void MergeSort(int arr[], int left, int right) {
    int center;
    if (left < right) {
        center = (left + right) / 2;
        MergeSort(arr, left, center);
```

```
        MergeSort(arr, center + 1, right);
        Merge(arr, left, center, right);
    }
}

void Merge(int arr[], int left, int center, int right) {
    int a[20], b[20], n1 = center - left + 1, n2 = right - center;
    for (int i = 0; i < n1; i++)
        a[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        b[j] = arr[center + 1 + j];
    int aptr = 0, bptr = 0, cptr = left;
    while (aptr < n1 && bptr < n2)
        arr[cptr++] = (a[aptr] <= b[bptr]) ? a[aptr++] : b[bptr++];
    while (aptr < n1)
        arr[cptr++] = a[aptr++];
    while (bptr < n2)
        arr[cptr++] = b[bptr++];
}

int main() {
    int i, n, arr[20];
    printf("Enter the limit: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    MergeSort(arr, 0, n - 1);
    printf("The sorted elements are: ");
    for (i = 0; i < n; i++)
        printf("%d\t", arr[i]);
    return 0;
```

```
}
```

Output

```
Enter the limit: 5
Enter the elements: 30 50 10 20 40
The sorted elements are: 10  20   30    40    50
```