**DATA STRUCTURE**

**EXPERIMENT** **1: IMPLEMENTATION OF SINGLE LINKED LIST**

**1.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:**

### (i) Insert a node at the beginning of a list

1. **Start**.
2. **Input**: The data to insert, data, and the head pointer, head.
3. **Create a new node** with the data data.
4. **Set** the next pointer of the new node to point to the current head.
5. **Update** head to point to the new node.
6. **End**.

### (ii) Insert a node after a given node PPP

1. **Start**.
2. **Input**: The data to insert, data, and the node after which to insert, P.
3. **Create a new node** with the data data.
4. **Set** the next pointer of the new node to point to the node that P points to.
5. **Set** the next pointer of P to point to the new node.
6. **End**.

### (iii) Insert a node at the end of a list

1. **Start**.
2. **Input**: The data to insert, data, and the head pointer, head.
3. **Create a new node** with the data data.
4. **If** head is NULL, set head to point to the new node and **end**.
5. **Otherwise**, traverse to the last node in the list.
6. **Set** the next pointer of the last node to point to the new node.
7. **End**.

### (iv) Find an element in a list

1. **Start**.
2. **Input**: The data to find, data, and the head pointer, head.
3. **Initialize** a pointer current to head.
4. **While** current is not NULL:
   * **If** current->data equals data, **return** current.
   * Move current to current->next.
5. **Return** NULL if the data is not found.
6. **End**.

### (v) Find the next node

1. **Start**.
2. **Input**: A pointer to the current node, node.
3. **Return** node->next.
4. **End**.

### (vi) Find the previous node

1. **Start**.
2. **Input**: The head pointer, head, and the node to find the previous of, node.
3. **Initialize** a pointer current to head.
4. **If** head is NULL or node is the first node, **return** NULL.
5. **While** current->next is not node:
   * Move current to current->next.
6. **Return** current.
7. **End**.

### (vii) Check if a node is the last node

1. **Start**.
2. **Input**: A pointer to the node, node.
3. **If** node->next is NULL, **return** true.
4. **Otherwise**, **return** false.
5. **End**.

### (viii) Check if the list is empty

1. **Start**.
2. **Input**: The head pointer, head.
3. **If** head is NULL, **return** true.
4. **Otherwise**, **return** false.
5. **End**.

### (ix) Delete a node at the beginning of a list

1. **Start**.
2. **Input**: The head pointer, head.
3. **If** head is NULL, **return**.
4. **Initialize** a pointer temp to head.
5. **Set** head to head->next.
6. **Free** the memory of temp.
7. **End**.

### (x) Delete a node after a given node PPP

1. **Start**.
2. **Input**: The node after which to delete, P.
3. **If** P is NULL or P->next is NULL, **return**.
4. **Initialize** a pointer temp to P->next.
5. **Set** P->next to temp->next.
6. **Free** the memory of temp.
7. **End**.

### (xi) Delete a node at the end of a list

1. **Start**.
2. **Input**: The head pointer, head.
3. **If** head is NULL, **return**.
4. **If** head->next is NULL, free head and set head to NULL, and **return**.
5. **Initialize** a pointer current to head.
6. **While** current->next->next is not NULL, move current to current->next.
7. **Free** the memory of current->next.
8. **Set** current->next to NULL.
9. **End**.

### (xii) Delete the entire list

1. **Start**.
2. **Input**: The head pointer, head.
3. **Initialize** a pointer current to head.
4. **While** current is not NULL:
   * **Initialize** a pointer temp to current.
   * Move current to current->next.
   * **Free** the memory of temp.
5. **Set** head to NULL.
6. **End**.

**PROGRAM:**

#include <stdio.h>

#include<stdlib.h>

#include<malloc.h>

void createfnode(int ele);

void insertfront(int ele);

void insertend(int ele);

void display();

struct node

{

int data;

struct node\* next;

};

struct node\* head = NULL;

struct node \*newnode;

void insertfront(int ele)

{

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

if(newnode!=NULL)

{ newnode->data=ele;

if(head!=NULL)

{

newnode->next=head;

head=newnode;

}

else

{

newnode->next=NULL;

head=newnode;

}

}

}

void insertend(int ele)

{

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

if(newnode!=NULL)

{

newnode->data=ele;

newnode->next=NULL;

if(head!=NULL)

{

struct node \*t;

t=head;

while(t->next!=NULL)

{

t=t->next;

}

newnode->next=NULL;

t->next=newnode;

}

else

{

head=newnode;

}

}

}

int listsize()

{

int c=0;

struct node \*t;

t=head;

while(t!=NULL)

{

c=c+1;

t=t->next;

}

printf("\n The size of the list is %d:\n",c);

return c;

}

void insertpos(int ele,int pos)

{

int ls=0;

ls=listsize();

if(head == NULL && (pos <= 0 || pos > 1))

{

printf("\nInvalid position to insert a node\n");

return;

}

// if the list is not empty and the position is out of range

if(head != NULL && (pos <= 0 || pos > ls))

{

printf("\nInvalid position to insert a node\n");

return;

}

struct node\* newnode = NULL;

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

if(newnode != NULL)

{

newnode->data=ele;

struct node\* temp = head;

//getting the position-1 node

int count = 1;

while(count < pos-1)

{

temp = temp -> next;

count += 1;

}

//if the position is 1 then insertion at the beginning

if(pos == 1)

{

newnode->next = head;

head = newnode;

}

else

{

newnode->next = temp->next;

temp->next = newnode;

}

}

}

void findnext(int s)

{

struct node \*temp;

temp=head;

if(temp==NULL&&temp->next==NULL)

{

printf("No next element ");

}

else

{

while(temp->data!=s)

{

temp=temp->next;

}

printf("\nNext Element of %d is %d\n",s,temp->next->data);

}

}

void findprev(int s)

{

struct node \*temp;

temp=head;

if(temp==NULL)

{

printf("List is empty ");

}

else

{

while(temp->next->data!=s)

{

temp=temp->next;

}

printf("\n The previous ele of %d is %d\n",s,temp->data);

}

}

void find(int s)

{

struct node \*temp;

temp=head;

if(head==NULL)

{

printf("\n List is empty");

}

else

{

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

{

temp=temp->next;

}

if(temp!=NULL && temp->data==s)

{

printf("\n Searching ele %d is present in the addr of %p",temp->data,temp);

}

else

{

printf("\n Searching elem %d is not present",s);

}

}

}

void isempty()

{

if(head==NULL)

{

printf("\nList is empty\n");

}

else

{

printf("\nList is not empty\n");

}

}

void deleteAtBeginning()

{

struct node \*t;

t=head;

head=t->next;

}

void deleteAtEnd()

{

struct node \*temp;

temp=head;

if(head==NULL)

{

printf("\n List is empty");

}

else

{

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

{

temp=temp->next;

}

temp->next=NULL;

}

}

void display()

{

struct node \*t;

t=head;

while(t!=NULL)

{

printf("%d\t",t->data);

t=t->next;

}

}

void delete(int ele)

{

struct node \*t;

t=head;

if(t->data==ele)

{

head=t->next;

}

else

{

while(t->next->data!=ele)

{

t=t->next;

}

t->next=t->next->next;

}

}

int main()

{

do

{

int ch,a,pos;

printf("\n Choose any one operation that you would like to perform\n");

printf("\n 1.Insert the element at the beginning");

printf("\n 2.Insert the element at the end");

printf("\n 3. To insert at the specified position");

printf("\n 4. To view list");

printf("\n 5.To view list size");

printf("\n 6.To delete first element");

printf("\n 7.To delete last element");

printf("\n 8.To find next element");

printf("\n 9. To find previous element");

printf("\n 10. To find search for an element");

printf("\n 11. To quit");

printf("\n Enter your choice\n");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("\n Insert an element to be inserted at the beginning\n");

scanf("%d",&a);

insertfront(a);

break;

case 2:

printf("\n Insert an element to be inserted at the End\n");

scanf("%d",&a);

insertend(a);

break;

case 3:

printf("\n Insert an element and the position to insert in the list\n");

scanf("%d%d",&a,&pos);

insertpos(a,pos);

break;

case 4:

display();

break;

case 5:

listsize();

break;

case 6:

printf("\n Delete an element to be in the beginning\n");

deleteAtBeginning();

break;

case 7:

printf("\n Delete an element to be at the end\n");

deleteAtEnd();

break;

case 8:

printf("\n enter the element to which you need to find next ele in the list\n");;

scanf("%d",&a);

findnext(a);

break;

case 9:

printf("\n enter the element to which you need to find prev ele in the list\n");;

scanf("%d",&a);

findprev(a);

break;

case 10:

printf("\n enter the element to find the address of it\n");;

scanf("%d",&a);

find(a);

break;

case 11:

printf("Ended");

exit(0);

default:

printf("Invalid option is chosen so the process is quit");

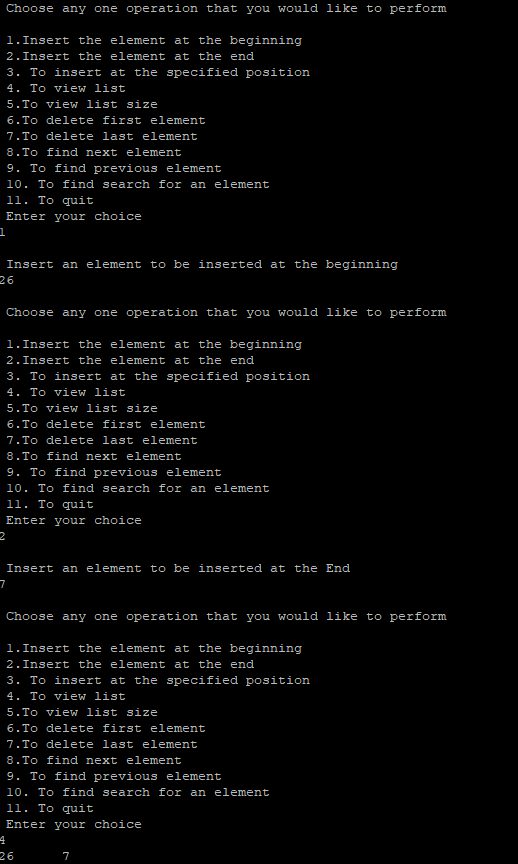
}

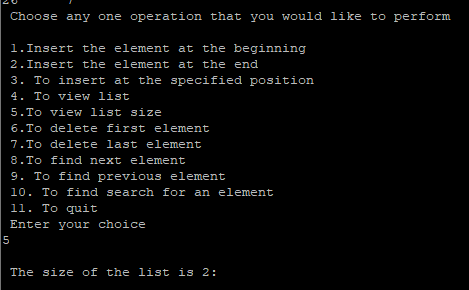
}while(1);

return 0;

}

**OUTPUT:**





**EXPERIMENT 2: 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**

**ALGORITHM:**

#### Insertion at the Beginning

1. **Start**.
2. **Input**: Data to insert, data, and head pointer, head.
3. **Create a new node** with data.
4. **Set** the new node's next to head.
5. **Set** the new node's prev to NULL.
6. **If** head is not NULL, set head->prev to the new node.
7. **Update** head to the new node.
8. **End**.

#### Deletion at the Beginning

1. **Start**.
2. **Input**: Head pointer, head.
3. **If** head is NULL, **return**.
4. **Initialize** temp to head.
5. **Set** head to head->next.
6. **If** head is not NULL, set head->prev to NULL.
7. **Free** temp.
8. **End**.

#### Display List

1. **Start**.
2. **Input**: Head pointer, head.
3. **Initialize** current to head.
4. **While** current is not NULL:
   * Print current->data.
   * Move current to current->next.
5. **End**.

**PROGRAM:**

#include<stdio.h>  
#include<stdlib.h>  
struct node  
{  
        struct node \*prev;  
        int data;  
        struct node \*next;  
};  
struct node \*createnode(int data)  
{  
        struct node \*newnode=malloc(sizeof(struct node));  
        newnode->prev=NULL;  
        newnode->data=data;  
        newnode->next=NULL;  
        return newnode;  
}  
struct node \*addToBeginning(struct node \*head,int data)  
{  
        struct node \*newnode=createnode(data);  
        if(head!=NULL)  
        {  
                head->prev=head;  
        }  
        newnode->next=head;  
        return newnode;  
}  
struct node \*addToEnd(struct node \*head,int data)  
{  
        struct node \*newnode=createnode(data);  
        if(head==NULL)  
        {  
                return newnode;  
        }  
        struct node \*temp=head;  
        while(temp->next!=NULL)  
        {  
                temp=temp->next;  
        }  
        temp->next=newnode;  
        newnode->prev=temp;  
        return head;  
}  
struct node \*addToMiddle(struct node \*head,int pos,int data)  
{  
        if(head==NULL||pos<=0)  
        {  
                printf("Invalid position\n");  
                return head;  
        }  
        struct node \*newnode=createnode(data);  
        struct node \*temp=head;  
        while(pos>1 && temp->next!=NULL)  
        {  
                temp=temp->next;  
                pos--;  
        }  
        newnode->next=temp->next;  
        newnode->prev=temp;  
        if(temp->next!=NULL)  
        {  
                temp->next->prev=newnode;  
        }

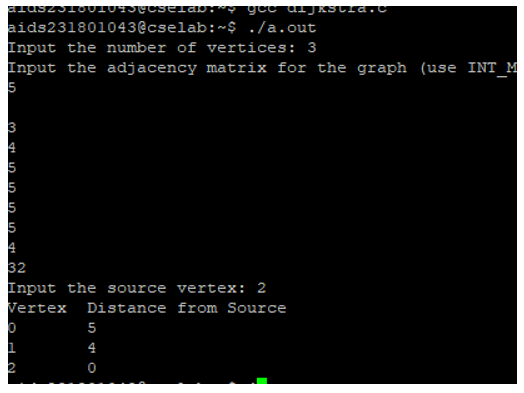
 temp->next=newnode;  
        return head;  
}  
struct node \*deleteFromBeginning(struct node \*head)  
{  
        if(head==NULL)  
        {  
                printf("List is empty\n");  
                return NULL;  
        }  
        struct node \*temp=head;  
        head=head->next;  
        if(head!=NULL)  
        {  
                head->prev=NULL;  
        }  
        free(temp);  
        return head;  
}  
struct node \*deleteFromEnd(struct node \*head)  
{  
        if(head==NULL)  
        {  
                printf("List is empty\n");  
                return NULL;  
        }  
        struct node \*temp=head;  
        while(temp->next!=NULL)  
        {  
                temp=temp->next;  
        }  
        if(temp->prev!=NULL)  
        {  
                temp->prev->next=NULL;  
        }  
        free(temp);  
        return head;  
}  
struct node \*deleteFromMiddle(struct node \*head,int pos)  
{  
        if(head==NULL)  
        {  
                printf("List is empty\n");  
                return NULL;  
        }  
        struct node \*temp=head;  
        while(pos>1 && temp->next!=NULL)  
        {  
                temp=temp->next;  
                pos--;  
        }  
        if(temp==head)  
        {  
                head=deleteFromBeginning(head);  
        }  
        else if(temp->next==NULL)  
        {  
                head=deleteFromEnd(head);  
        }  
        else

 {  
                temp->prev->next=temp->next;  
                temp->next->prev=temp->prev;  
                free(temp);  
        }  
        return head;  
}  
void printList(struct node \*head)  
{  
        struct node \*temp=head;  
        while(temp!=NULL)  
        {  
                printf("%d",temp->data);  
                temp=temp->next;  
        }  
        printf("NULL\n");  
}  
struct node \*findElement(struct node \*head,int key)  
{  
        struct node \*current=head;  
        while(current!=NULL)  
        {  
                if(current!=NULL)  
                {  
                        printf("Element %d found in the list\n",key);  
                        return current;  
                }  
                current=current->next;  
        }  
        printf("Element not found");  
        return NULL;  
}  
int main()

{  
        struct node \*head=NULL;  
        int choice,data,pos;  
        printf("\n1.Add to beginning");  
        printf("\n2.Add to end");  
        printf("\n3.Add to middle");  
        printf("\n4.Delete from beginning");  
        printf("\n5.Delete from end");  
        printf("\n6.Delete from middle");  
        printf("\n7.Search element");  
        printf("\n8.Dispaly");  
        printf("\n9.Exit");  
        while(1)  
        {  
                printf("\nEnter your choice: ");  
                scanf("%d",&choice);  
                switch(choice)  
                {  
                        case 1:  
                                {  
                                printf("Enter data: ");  
                                scanf("%d",&data);  
                                head=addToBeginning(head,data);  
                                break;  
                                }  
                        case 2:  
                                {

 {  
                                printf("Enter data: ");  
                                scanf("%d",&data);  
                                head=addToEnd(head,data);  
                                break;  
                                }  
                        case 3:  
                                {  
                                printf("Enter position: ");  
                                scanf("%d",&pos);  
                                printf("\nEnter data: ");  
                                scanf("%d",&data);  
                                head=addToMiddle(head,pos,data);  
                                break;  
                                }  
                        case 4:  
                                {  
                                head=deleteFromBeginning(head);  
                                break;  
                                }  
                        case 5:  
                                {  
                                head=deleteFromEnd(head);  
                                break;  
                                }  
                        case 6:  
                                {  
                                printf("Enter position: ");  
                                scanf("%d",&pos);  
                                head=deleteFromMiddle(head,pos);  
                                break;  
                                }  
                        case 7:  
                                {  
                                printf("Enter element: ");  
                                scanf("%d",&data);  
                                head=findElement(head,data);  
                                break;  
                                }  
                        case 8:  
                                {  
                                printf("List:");  
                                printList(head);  
                                break;  
                                }  
                        case 9:  
                                {  
                                exit(0);  
                                }  
                        default:  
                                {  
                                printf("Invalid choice\n");  
                                }  
                }  
                return 0;  
        }  
}

**OUTPUT:**

****

**EXPERIMENT 3: POLYNOMIAL ADDITION**

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

1. **Polynomial Addition**
2. **Polynomial Subtraction**
3. **Polynomial Multiplication**

**ALGORITHM:**

1. **Define Node Structure:**
   * Define a struct node with three members: int coeff, int expo, and struct node \*next.
2. **Insert Node in Polynomial:**
   * Create a new node with given coefficient and exponent.
   * If the list is empty or the exponent of the new node is greater than the head node, insert the new node at the beginning.
   * Otherwise, traverse the list to find the appropriate position based on the exponent and insert the new node.
3. **Create Polynomial:**
   * Read the number of terms in the polynomial.
   * For each term, read the coefficient and exponent and insert the node into the polynomial.
4. **Print Polynomial:**
   * Traverse the list and print each term in the form coeff\*x^expo.
   * Print a + between terms and a newline at the end.
5. **Add Two Polynomials:**
   * Initialize pointers for traversing both polynomials.
   * Compare the exponents of the current nodes from both polynomials:
     + If exponents are equal, add the coefficients and create a new node with the sum.
     + If the exponent of the first polynomial is greater, insert the node from the first polynomial.
     + If the exponent of the second polynomial is greater, insert the node from the second polynomial.
   * Continue until both polynomials are traversed.
   * If any polynomial has remaining terms, append them to the result.
6. **Main Function:**
   * Create two polynomial linked lists.
   * Call the function to create and read terms for both polynomials.
   * Call the function to add the two polynomials.
   * Print the result.

**PROGRAM:**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int coeff;

int expo;

struct node \*next;

};

struct node\* insert(struct node \*head,int co,int exp)

{

struct node \*temp;

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

newnode->coeff=co;

newnode->expo=exp;

newnode->next=NULL;

if(head==NULL || exp>head->expo)

{

newnode->next=head;

head=newnode;

}

else

{

temp=head;

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

temp=temp->next;

newnode->next=temp->next;

temp->next=newnode;

}

return head;

}

struct node\* create(struct node \*head)

{

int n,i;

int coeff;

int expo;

printf("Enter the no of terms:");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("Enter the coeefficient for term %d:",i+1);

scanf("%d",&coeff);

printf("Enter the exponent for term %d:",i+1);

scanf("%d",&expo);

head=insert(head,coeff,expo);

}

return head;

}

void print(struct node\* head)

{

if(head==NULL)

printf("No Polynomial");

else

{

struct node \*temp=head;

while(temp!=NULL)

{

printf("%dx^%d",temp->coeff,temp->expo);

temp=temp->next;

if(temp!=NULL)

printf("+");

else

printf("\n");

}

}

}

void polyAdd(struct node \*head1, struct node \*head2)

{

struct node \*ptr1=head1;

struct node \*ptr2=head2;

struct node \*head3=NULL;

while(ptr1!=NULL && ptr2!=NULL)

{

if(ptr1->expo == ptr2->expo)

{

head3=insert(head3,ptr1->coeff+ptr2->coeff,ptr1->expo);

ptr1=ptr1->next;

ptr2=ptr2->next;

}

else if(ptr1->expo > ptr2->expo)

{

head3=insert(head3,ptr1->coeff,ptr1->expo);

ptr1=ptr1->next;

}

else if(ptr1->expo < ptr2->expo)

{

head3=insert(head3,ptr2->coeff,ptr2->expo);

ptr2=ptr2->next;

}

}

while(ptr1!=NULL)

{

head3=insert(head3,ptr1->coeff,ptr1->expo);

ptr1=ptr1->next;

}

while(ptr2!=NULL)

{

head3=insert(head3,ptr2->coeff,ptr2->expo);

ptr2=ptr2->next;

}

printf("Added Polynomial is: ") ;

print(head3);

}

int main()

{

struct node \*head1=NULL;

struct node \*head2=NULL;

printf("Enter first polynomial\n");

head1=create(head1);

printf("Enter second polynomial\n");

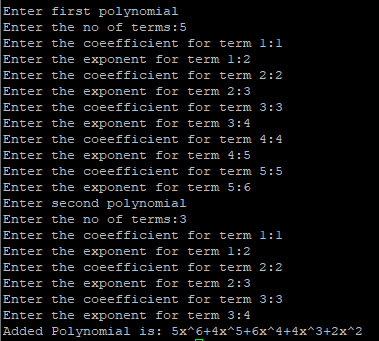
head2=create(head2);

polyAdd(head1,head2);

return 0;

}

**OUTPUT:**

****

**EXPERIMENT 4: IMPLEMENTATION OF STACK USING LINKED LIST AND ARRAY**

**4.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:**

#### Using Array

1. **Start**.
2. **Initialize** stack array and top to -1.
3. **Push Operation**:
   * **Input**: Element x.
   * Increment top.
   * Set stack[top] to x.
4. **Pop Operation**:
   * **If** top is -1, stack is empty.
   * Return stack[top] and decrement top.
5. **End**.

#### Using Linked List

1. **Start**.
2. **Initialize** head pointer head to NULL.
3. **Push Operation**:
   * **Create** a new node with data.
   * **Set** new node's next to head.
   * **Update** head to new node.
4. **Pop Operation**:
   * **If** head is NULL, stack is empty.
   * **Initialize** temp to head.
   * **Set** head to head->next.
   * **Free** temp.
5. **End**

**PROGRAM 1:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int Data;

struct Node \*next;

} \*top;

void popStack() {

struct Node \*temp, \*var = top;

if (var == top) {

top = top->next;

free(var);

} else {

printf("\nStack Empty");

}

}

void push(int value) {

struct Node \*temp;

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

temp->Data = value;

if (top == NULL) {

top = temp;

top->next = NULL;

} else {

temp->next = top;

top = temp;

}

}

void display() {

struct Node \*var = top;

if (var != NULL) {

printf("\nElements are as:\n");

while (var != NULL) {

printf("\t%d\n", var->Data);

var = var->next;

}

printf("\n");

} else {

printf("\nStack is Empty");

}

}

int main() {

int i = 0;

top = NULL;

printf(" \n1. Push to stack");

printf(" \n2. Pop from Stack");

printf(" \n3. Display data of Stack");

printf(" \n4. Exit\n");

while (1) {

printf(" \nChoose Option: ");

scanf("%d", &i);

switch (i) {

case 1: {

int value;

printf("\nEnter a value to push into Stack: ");

scanf("%d", &value);

push(value);

break;

}

case 2: {

popStack();

printf("\n The last element is popped");

break;

}

case 3: {

display();

break;

}

case 4: {

struct Node \*temp;

while (top != NULL) {

temp = top->next;

free(top);

top = temp;

}

return 0;

}

default: {

printf("\nwrong choice for operation");

}

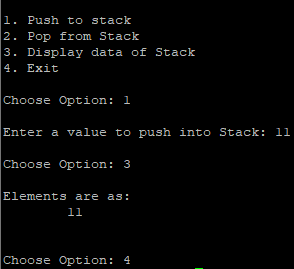
}

}

return 0;

}

**OUTPUT 1:**

****

**PROGRAM 2:**

Implementation of Stack using Array

#include<stdio.h>

int stack[100],choice,n,top,x,i;

void push(void);

void pop(void);

void display(void);

int main()

{

top=-1;

printf("\n Enter the size of STACK[MAX=100]:");

scanf("%d",&n);

printf("\n\t STACK OPERATIONS USING ARRAY");

printf("\n\t--------------------------------");

printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");

do

{

printf("\n Enter the Choice:");

scanf("%d",&choice);

switch(choice)

{

case 1:

{

push();

break;

}

case 2:

{

pop();

break;

}

case 3:

{

display();

break;

}

case 4:

{

printf("\n\t EXIT POINT ");

break;

}

default:

{

printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");

}

}

}

while(choice!=4);

return 0;

}

void push()

{

if(top>=n-1)

{

printf("\n\tSTACK is over flow");

}

else

{

printf(" Enter a value to be pushed:");

scanf("%d",&x);

top++;

stack[top]=x;

}

}

void pop()

{

if(top<=-1)

{

printf("\n\t Stack is under flow");

}

else

{

printf("\n\t The popped elements is %d",stack[top]);

top--;

}

}

void display()

{

if(top>=0)

{

printf("\n The elements in STACK \n");

for(i=top; i>=0; i--)

printf("\n%d",stack[i]);

printf("\n Press Next Choice");

}

else

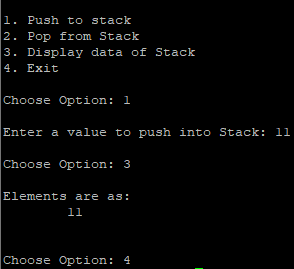
{

printf("\n The STACK is empty");

}

}

**Output 2:**

****

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

**ALGORITHM:**

Step 1: Start the program.

Step 2: Get the infix expression as input.

Step 3: Read the input from left to right.

Step 4: If the input is operand then place it in the postfix expression.

Step 5: Else if the input symbol is an operator then check for the operator type and

also the precedence, pop entries from the stack and place them in the

postfix expression until the lowest priority operator is encountered.

Step 6: ‘(‘symbol will be popped from stack only when we get a ‘)’ symbol.

Step 7: When the input is completely read then pop the elements in stack until it

becomes empty.

Step 8: Display the postfix expression.

Step 9: Stop the program.

**PROGRAM:**

#include<stdio.h>

#include<conio.h>

#include<alloc.h>

int top=0,st[20];

char inf[40],post[40];

void postfix();

void push(int);

char pop();

void main()

{

clrscr();

printf("Enter the infix expression:");

scanf("%s",inf);

postfix();

getch();

}

void postfix()

{int i,j=0;

for(i=0;inf[i]!=0;i++)

{switch(inf[i])

{

case '+':while(st[top]>=1)

post[j++]=pop();

push(1);

break;

case '-':while(st[top]>=1)

post[j++]=pop();

push(2);

break;

case '\*':while(st[top]>=3)

post[j++]=pop();

push(3);

break;

case '/':while(st[top]>=4)

post[j++]=pop();

push(4);

break;

case '^':

post[j++]=pop();

push(5);

break;

case '(':push(0);

break;

case ')':while(st[top]!=0)

post[j++]=pop();

top--;

break;

default:

post[j++]=inf[i];

}}

while(top>0)

post[j++]=pop();

printf("\nPostfix expression is =>\n\t\t%s",post);

}void push(int ele)

{

top++;

st[top]=ele;

}char pop()

{int el;

char e;

el=st[top];

top--;

switch(el)

{case 1:

e='+';

break;

case 2:

e='-';

break;

case 3:

e='\*';

break;

case 4:

e='/';

break;

case 5:

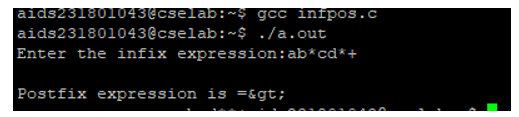
e='^';

break;

}return(e);

}

**OUTPUT**

****

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

**ALGORITHM:**

Step 1: Start the program.

Step 2: Read the postfix expression from left to right

Step 3: If the symbol read is an operand then push it onto the stack

Step 4: If the operator is read POP two operands and perform arithmetic

operations if operator is

+ then result=operand 1 + operand 2

- then result=operand 1 - operand 2

\* then result=operand 1 \* operand 2

/ then result=operand 1 / operand 2

Step 5: Push the result onto the stack

Step 6: Repeat steps 2-5 till the postfix expression is not over

Step 7: Stop the program.

**PROGRAM:**

#include <stdio.h>

#include <string.h>

int top = -1;

int stack[100];

void push (int data) {

stack[++top] = data;

}

int pop () {

int data;

if (top == -1)

return -1;

data = stack[top];

stack[top] = 0;

top--;

return (data);

}

int main()

{

char str[100];

int i, data = -1, operand1, operand2, result;

printf("Enter ur postfix expression:");

fgets(str, 100, stdin);

for (i = 0; i < strlen(str); i++)

{

if (isdigit(str[i]))

{

data = (data == -1) ? 0 : data;

data = (data \* 10) + (str[i] - 48);

continue;

}

if (data != -1)

{

push(data);

}

if (str[i] == '+' || str[i] == '-'|| str[i] == '\*' || str[i] == '/')

{

operand2 = pop();

operand1 = pop();

if (operand1 == -1 || operand2 == -1)

break;

switch (str[i])

{

case '+':

result = operand1 + operand2;

push(result);

break;

case '-':

result = operand1 - operand2;

push(result);

break;

case '\*':

result = operand1 \* operand2;

push(result);

break;

case '/':

result = operand1 / operand2;

push(result);

break;

}

}

data = -1;

}

if (top == 0)

printf("The answer is:%d\n", stack[top]);

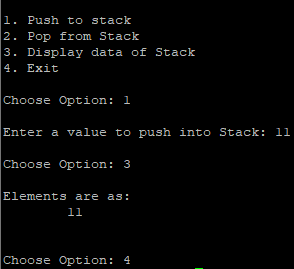
else

printf("u have given wrong postfix expression\n");

return 0;

}

**OUTPUT**

****

**7.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:**

Step 1: Start the program.

Step 2: For queue insertion operation, check for queue overflow

Step 3: If R>=N then print queue overflow else increment rear pointer and insert

the element.

Step 4: For queue deletion operation, check for underflow of the queue.

Step 5: If F=0 then print queue underflow else delete the element and increment

the front pointer

Step 6: Stop the program.

**PROGRAM:**

#include<stdio.h >

#include<conio.h >

#include<alloc.h >

struct queue

{

int data;

struct queue \*next;

};

struct queue \*addq(struct queue \*front);

struct queue \*delq(struct queue \*front);

void main()

{

struct queue \*front;

int reply,option,data;

clrscr();

front=NULL;

do

{

printf("\n1.addq");

printf("\n2.delq");

printf("\n3.exit");

printf("\nSelect the option");

scanf("%d",&option);

switch(option)

{

case 1 :

front=addq(front);

printf("\n The element is added into the queue");

break;

case 2 :

front=delq(front);

break;

case 3 : exit(0);

}

}while(1);

}

struct queue \*addq(struct queue \*front)

{

struct queue \*c,\*r;

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

if(c==NULL)

{

printf("Insufficient memory");

return(front);

}

printf("\nEnter data");

scanf("%d",&c->data);

c->next=NULL;

if(front==NULL)

{

front=c;

}

else

{

//insert new node after last node

r=front;

while(r->next!=NULL)

{

r=r->next;

}}

return(front);

}

struct queue \*delq(struct queue \*front)

{

struct queue \*c;

if(front==NULL)

{

printf("Queue is empty");

return(front);

}

printf("Deleted data:%d",front->data);

c=front;

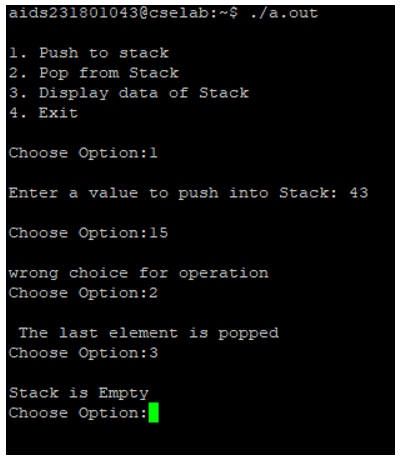
front=front->next;

free(c);

return(front);

}

**OUTPUT:**

****

**8.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:**

1. **Start**.

2. **Input**: Data to insert, data, and root pointer, root.

3. **If** root is NULL, create new node with data and set it as root.

4. **Else**, recursively insert data into left or right subtree based on comparison with root->data.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int element;

struct node\* left;

struct node\* right;

};

struct node\* createNode(int val)

{

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

Node->element = val;

Node->left = NULL;

Node->right = NULL;

return (Node);

}

void traversePreorder(struct node\* root)

{

if (root == NULL)

return;

printf(" %d ", root->element);

traversePreorder(root->left);

traversePreorder(root->right);

}

TREE TRAVERSAL

void traverseInorder(struct node\* root)

{

if (root == NULL)

return;

traverseInorder(root->left);

printf(" %d ", root->element);

traverseInorder(root->right);

}

void traversePostorder(struct node\* root)

{

if (root == NULL)

return;

traversePostorder(root->left);

traversePostorder(root->right);

printf(" %d ", root->element);

}

int main()

{

struct node\* root = createNode(36);

root->left = createNode(26);

root->right = createNode(46);

root->left->left = createNode(21);

TREE TRAVERSAL

root->left->right = createNode(31);

root->left->left->left = createNode(11);

root->left->left->right = createNode(24);

root->right->left = createNode(41);

root->right->right = createNode(56);

root->right->right->left = createNode(51);

root->right->right->right = createNode(66);

printf("\n The Preorder traversal of given binary tree is -\n");

traversePreorder(root);

printf("\n The Inorder traversal of given binary tree is -\n");

traverseInorder(root);

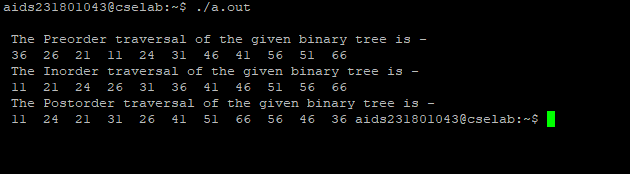
printf("\n The Postorder traversal of given binary tree is -\n");

traversePostorder(root);

return 0;

}

**OUTPUT:**



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

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

**ALGORITHM:**

1. **Start**.

2. **Input**: Root pointer, root.

3. **If** root is not NULL:

* Recursively perform inorder traversal on left subtree.
* Visit root->data.
* Recursively perform inorder traversal on right subtree.

4.**End**

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

struct BinaryTreeNode {

int key;

struct BinaryTreeNode \*left, \*right;

};

struct BinaryTreeNode\* newNodeCreate(int value) {

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

temp->key = value;

temp->left = temp->right = NULL;

return temp;

}

struct BinaryTreeNode\* searchNode(struct BinaryTreeNode\* root, int target) {

if (root == NULL || root->key == target) {

return root;

}

if (root->key < target) {

return searchNode(root->right, target);

}

return searchNode(root->left, target);

}

struct BinaryTreeNode\* insertNode(struct BinaryTreeNode\* node, int value) {

if (node == NULL) {

return newNodeCreate(value);

}

if (value < node->key) {

node->left = insertNode(node->left, value);

} else if (value > node->key) {

node->right = insertNode(node->right, value);

}

return node;

}

void postOrder(struct BinaryTreeNode\* root) {

if (root != NULL) {

postOrder(root->left);

postOrder(root->right);

printf(" %d ", root->key);

}

}

void inOrder(struct BinaryTreeNode\* root) {

if (root != NULL) {

inOrder(root->left);

printf(" %d ", root->key);

inOrder(root->right);

}

}

void preOrder(struct BinaryTreeNode\* root) {

if (root != NULL) {

printf(" %d ", root->key);

preOrder(root->left);

preOrder(root->right);

}

}

struct BinaryTreeNode\* findMin(struct BinaryTreeNode\* root) {

if (root == NULL) {

return NULL;

} else if (root->left != NULL) {

return findMin(root->left);

}

return root;

}

struct BinaryTreeNode\* delete (struct BinaryTreeNode\* root, int x) {

if (root == NULL) {

return NULL;

}

if (x > root->key) {

root->right = delete (root->right, x);

} else if (x < root->key) {

root->left = delete (root->left, x);

} else {

if (root->left == NULL && root->right == NULL) {

free(root);

return NULL;

} else if (root->left == NULL || root->right == NULL) {

struct BinaryTreeNode\* temp;

if (root->left == NULL) {

temp = root->right;

} else {

temp = root->left;

}

free(root);

return temp;

} else {

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

root->key = temp->key;

root->right = delete (root->right, temp->key);

}

}

return root;

}

int main() {

struct BinaryTreeNode\* root = NULL;

root = insertNode(root, 50);

insertNode(root, 30);

insertNode(root, 20);

insertNode(root, 40);

insertNode(root, 70);

insertNode(root, 60);

insertNode(root, 80);

if (searchNode(root, 60) != NULL) {

printf("60 found");

} else {

printf("60 not found");

}

printf("\n");

postOrder(root);

printf("\n");

preOrder(root);

printf("\n");

inOrder(root);

printf("\n");

struct BinaryTreeNode\* temp = delete (root, 70);

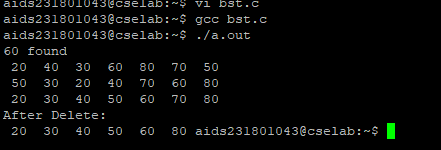
printf("After Delete: \n");

inOrder(root);

return 0;

}

**OUTPUT:**



**10.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:**

1. **Start**.
2. **Input**: Data to insert, data, and root pointer, root.
3. **Perform** BST insertion.
4. **Update** height of the current node.
5. **Get** balance factor.
6. **If** unbalanced, perform appropriate rotations (LL, LR, RR, RL).
7. **End**.

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node\* left;

struct node\* right;

int ht;

};

struct node\* root = NULL;

struct node\* create(int);

struct node\* insert(struct node\*, int);

struct node\* delete(struct node\*, int);

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

struct node\* rotate\_left(struct node\*);

struct node\* rotate\_right(struct node\*);

int balance\_factor(struct node\*);

int height(struct node\*);

void inorder(struct node\*);

void preorder(struct node\*);

void postorder(struct node\*);

struct node\* create(int data) {

struct node\* new\_node = (struct node\*) malloc(sizeof(struct node));

if (new\_node == NULL) {

printf("\nMemory can't be allocated\n");

return NULL;

}

new\_node->data = data;

new\_node->left = NULL;

new\_node->right = NULL;

new\_node->ht = 1; // New nodes are initially at height 1

return new\_node;

}

struct node\* rotate\_left(struct node\* root) {

struct node\* right\_child = root->right;

root->right = right\_child->left;

right\_child->left = root;

root->ht = height(root);

right\_child->ht = height(right\_child);

return right\_child;

}

struct node\* rotate\_right(struct node\* root) {

struct node\* left\_child = root->left;

root->left = left\_child->right;

left\_child->right = root;

root->ht = height(root);

left\_child->ht = height(left\_child);

return left\_child;

}

int balance\_factor(struct node\* root) {

if (root == NULL)

return 0;

int lh = root->left ? root->left->ht : 0;

int rh = root->right ? root->right->ht : 0;

return lh - rh;

}

int height(struct node\* root) {

if (root == NULL)

return 0;

int lh = root->left ? root->left->ht : 0;

int rh = root->right ? root->right->ht : 0;

return (lh > rh ? lh : rh) + 1;

}

struct node\* insert(struct node\* root, int data) {

if (root == NULL) {

struct node\* new\_node = create(data);

return new\_node;

} else if (data > root->data) {

// Insert the new node to the right

root->right = insert(root->right, data);

if (balance\_factor(root) == -2) {

if (data > root->right->data) {

root = rotate\_left(root);

} else {

root->right = rotate\_right(root->right);

root = rotate\_left(root);

}

}

} else {

root->left = insert(root->left, data);

if (balance\_factor(root) == 2) {

if (data < root->left->data) {

root = rotate\_right(root);

} else {

root->left = rotate\_left(root->left);

root = rotate\_right(root);

}

}

}

// Update the heights of the nodes

root->ht = height(root);

return root;

}

struct node\* delete(struct node\* root, int x) {

if (root == NULL) {

return NULL;

}

if (x > root->data) {

root->right = delete(root->right, x);

if (balance\_factor(root) == 2) {

if (balance\_factor(root->left) >= 0) {

root = rotate\_right(root);

} else {

root->left = rotate\_left(root->left);

root = rotate\_right(root);

}

}

} else if (x < root->data) {

root->left = delete(root->left, x);

if (balance\_factor(root) == -2) {

if (balance\_factor(root->right) <= 0) {

root = rotate\_left(root);

} else {

root->right = rotate\_right(root->right);

root = rotate\_left(root);

}

}

} else {

if (root->right != NULL) {

struct node\* temp = root->right;

while (temp->left != NULL)

temp = temp->left;

root->data = temp->data;

root->right = delete(root->right, temp->data);

if (balance\_factor(root) == 2) {

if (balance\_factor(root->left) >= 0) {

root = rotate\_right(root);

} else {

root->left = rotate\_left(root->left);

root = rotate\_right(root);

}

}

} else {

return root->left;

}

}

root->ht = height(root);

return root;

}

struct node\* search(struct node\* root, int key) {

if (root == NULL) {

return NULL;

}

if (root->data == key) {

return root;

}

if (key > root->data) {

return search(root->right, key);

} else {

return search(root->left, key);

}

}

void inorder(struct node\* root) {

if (root == NULL) {

return;

}

inorder(root->left);

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

inorder(root->right);

}

void preorder(struct node\* root) {

if (root == NULL) {

return;

}

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

preorder(root->left);

preorder(root->right);

}

void postorder(struct node\* root) {

if (root == NULL) {

return;

}

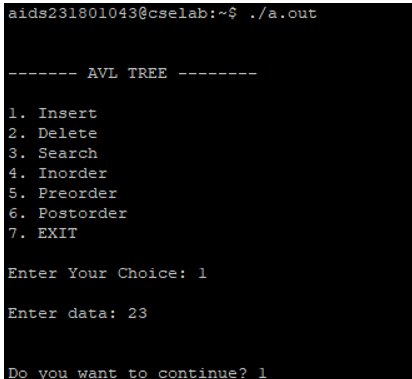
postorder(root->left);

postorder(root->right);

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

}

**OUTPUT:**

****

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

**AIM:**

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

**ALGORITHM:**

1. Define a structure Graph to represent the graph, containing the number of vertices (V), number of edges (E), and the adjacency matrix (Adj).
2. Implement a function adjMatrix() to create the graph using an adjacency matrix. Allocate memory for the graph structure and the adjacency matrix.
3. Initialize the adjacency matrix with zeros.
4. Set the edges in the adjacency matrix based on the connections between vertices.
5. Implement the DFS function (DFS) to perform depth-first traversal starting from a given vertex u.
6. Mark the vertex u as visited and print it.
7. For each adjacent vertex v of u that is not visited and is connected to u, recursively call DFS on v.
8. Implement DFStraversal function to perform DFS traversal on the entire graph.
9. Initialize an array vis to keep track of visited vertices.
10. Iterate over all vertices of the graph, and if a vertex is not visited, call DFS on it.
11. In the main function, create the graph using adjMatrix and perform DFS traversal using DFStraversal

**PROGRAM:**

#include <stdio.h>

#include <stdlib.h>

int vis[100];

struct Graph {

int V;

int E;

int\*\* Adj;

};

struct Graph\* adjMatrix()

{

struct Graph\* G = (struct Graph\*)

malloc(sizeof(struct Graph));

if (!G) {

printf("Memory Error\n");

return NULL;

}

G->V = 7;

G->E = 7;

G->Adj = (int\*\*)malloc((G->V) \* sizeof(int\*));

for (int k = 0; k < G->V; k++) {

G->Adj[k] = (int\*)malloc((G->V) \* sizeof(int));

}

for (int u = 0; u < G->V; u++) {

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

G->Adj[u][v] = 0;

}

}

G->Adj[0][1] = G->Adj[1][0] = 1;

G->Adj[0][2] = G->Adj[2][0] = 1;

G->Adj[1][3] = G->Adj[3][1] = 1;

G->Adj[1][4] = G->Adj[4][1] = 1;

G->Adj[1][5] = G->Adj[5][1] = 1;

G->Adj[1][6] = G->Adj[6][1] = 1;

G->Adj[6][2] = G->Adj[2][6] = 1;

return G;

}

void DFS(struct Graph\* G, int u)

{

vis[u] = 1;

printf("%d ", u);

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

if (!vis[v] && G->Adj[u][v]) {

DFS(G, v);

}

}

}

void DFStraversal(struct Graph\* G)

{

for (int i = 0; i < 100; i++) {

vis[i] = 0;

}

for (int i = 0; i < G->V; i++) {

if (!vis[i]) {

DFS(G, i);

}

}

}

void main()

{

struct Graph\* G;

G = adjMatrix();

DFStraversal(G);

}

**PROGRAM 2:**

#include <stdio.h>

#include <stdlib.h>

struct node {

int vertex;

struct node\* next;

};

struct adj\_list {

struct node\* head;

};

struct graph {

int num\_vertices;

struct adj\_list\* adj\_lists;

int\* visited;

};

struct node\* new\_node(int vertex) {

struct node\* new\_node = (struct node\*)malloc(sizeof(struct node));

new\_node->vertex = vertex;

new\_node->next = NULL;

return new\_node;

}

struct graph\* create\_graph(int n) {

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

graph->num\_vertices = n;

graph->adj\_lists = (struct adj\_list\*)malloc(n \* sizeof(struct adj\_list));

graph->visited = (int\*)malloc(n \* sizeof(int));

int i;

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

graph->adj\_lists[i].head = NULL;

graph->visited[i] = 0;

}

return graph;

}

void add\_edge(struct graph\* graph, int src, int dest) {

struct node\* new\_node1 = new\_node(dest);

new\_node1->next = graph->adj\_lists[src].head;

graph->adj\_lists[src].head = new\_node1;

struct node\* new\_node2 = new\_node(src);

new\_node2->next = graph->adj\_lists[dest].head;

graph->adj\_lists[dest].head = new\_node2;

}

void bfs(struct graph\* graph, int v) {

int queue[1000];

int front = -1;

int rear = -1;

graph->visited[v] = 1;

queue[++rear] = v;

while (front != rear) {

int current\_vertex = queue[++front];

printf("%d ", current\_vertex);

struct node\* temp = graph->adj\_lists[current\_vertex].head;

while (temp != NULL) {

int adj\_vertex = temp->vertex;

if (graph->visited[adj\_vertex] == 0) {

graph->visited[adj\_vertex] = 1;

queue[++rear] = adj\_vertex;

}

temp = temp->next;

}

}

}

int main() {

struct graph\* graph = create\_graph(6);

add\_edge(graph, 0, 1);

add\_edge(graph, 0, 2);

add\_edge(graph, 1, 3);

add\_edge(graph, 1, 4);

add\_edge(graph, 2, 4);

add\_edge(graph, 3, 4);

add\_edge(graph, 3, 5);

add\_edge(graph, 4,5);

printf("BFS traversal starting from vertex 0: ");

bfs(graph, 0);

return 0;

}

**OUTPUT:**



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

**algorithm.**

1. Define a function minKey to find the vertex with the minimum key value that is not yet included in the MST.
2. Define a function printMST to print the edges and their weights in the MST.
3. Define the primMST function to generate the MST:
4. Initialize arrays parent, key, and mstSet.
5. Set all elements of key to INT\_MAX, and all elements of mstSet to 0.
6. Make the key of the first vertex 0 and set its parent to -1.
7. Repeat for vertices - 1 times:
   1. Find the vertex u with the minimum key value from the set of vertices not yet included in the MST.
   2. Add u to the MST.
   3. Update the key value and parent index of all adjacent vertices of u.
8. Define the main function:
9. Input the number of vertices.
10. Input the adjacency matrix for the graph.
11. Call primMST to find and print the MST.

**PROGRAM:**

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minKey(int key[], int mstSet[], int vertices) {

int min = INT\_MAX, minIndex;

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

if (!mstSet[v] && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

void printMST(int parent[], int graph[MAX\_VERTICES][MAX\_VERTICES], int

vertices) {

printf("Edge \tWeight\n");

for (int i = 1; i < vertices; i++) {

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

}

}

void primMST(int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

int parent[MAX\_VERTICES];

int key[MAX\_VERTICES];

weight edge

int mstSet[MAX\_VERTICES];

for (int i = 0; i < vertices; i++) {

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0; // Make key 0 so that this vertex is picked as the

first vertex

parent[0] = -1;

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

int u = minKey(key, mstSet, vertices);

mstSet[u] = 1;

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

the key[v]

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

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

}

}

}

printMST(parent, graph, vertices);

}

int main() {

int vertices;

printf("Input the number of vertices: ");

scanf("%d", &vertices);

if (vertices <= 0 || vertices > MAX\_VERTICES) {

printf("Invalid number of vertices. Exiting...\n");

return 1;

}

int graph[MAX\_VERTICES][MAX\_VERTICES];

printf("Input the adjacency matrix for the graph:\n");

for (int i = 0; i < vertices; i++) {

for (int j = 0; j < vertices; j++) {

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

}

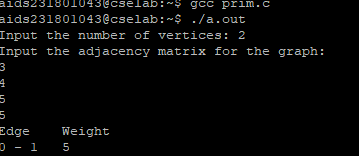
}

primMST(graph, vertices);

return 0;

}

**OUTPUT:**



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

**ALGORITHM:**

1. Initialize an array dist[] to store the shortest distance from the source vertex to each vertex, and an array sptSet[] to keep track of vertices included in the shortest path tree.
2. Initialize all distances as INT\_MAX and sptSet[] as 0.
3. Set the distance of the source vertex to itself as 0.
4. Repeat the following steps for vertices - 1 times: a. Pick the vertex u not yet included in sptSet[] with the minimum distance value. b. Mark u as processed. c. Update the distance value of all vertices adjacent to u which are not yet included in sptSet[].
5. Print the distances from the source vertex to all other vertices.

**PROGRAM:**

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minDistance(int dist[], int sptSet[], int vertices) {

int min = INT\_MAX, minIndex;

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

if (!sptSet[v] && dist[v] < min) {

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

void printSolution(int dist[], int vertices) {

printf("Vertex \tDistance from Source\n");

for (int i = 0; i < vertices; i++) {

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

}

}

void dijkstra(int graph[MAX\_VERTICES][MAX\_VERTICES], int src, int

vertices) {

int dist[MAX\_VERTICES]; // 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

for (int i = 0; i < vertices; i++) {

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

dist[src] = 0;

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

.

int u = minDistance(dist, sptSet, vertices);

sptSet[u] = 1;

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

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX &&

dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

printSolution(dist, vertices);

}

int main() {

int vertices;

printf("Input the number of vertices: ");

scanf("%d", &vertices);

if (vertices <= 0 || vertices > MAX\_VERTICES) {

printf("Invalid number of vertices. Exiting...\n");

return 1;

}

int graph[MAX\_VERTICES][MAX\_VERTICES];

printf("Input the adjacency matrix for the graph (use INT\_MAX for

infinity):\n");

for (int i = 0; i < vertices; i++) {

for (int j = 0; j < vertices; j++) {

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

}

}

int source;

printf("Input the source vertex: ");

scanf("%d", &source);

if (source < 0 || source >= vertices) {

printf("Invalid source vertex. Exiting...\n");

return 1;

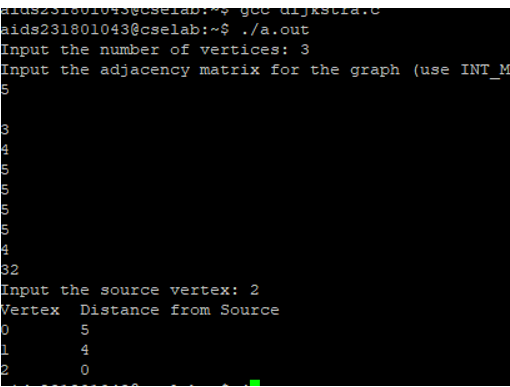
}

dijkstra(graph, source, vertices);

return 0;

}

**OUTPUT:**

****

**15. 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:**

1. **Start**.
2. **Input**: Array arr, low index low, high index high.
3. **If** low < high:
   * **Partition** the array:
     1. Choose a pivot element.
     2. Rearrange the elements such that all elements less than the pivot are on the left, all elements greater than the pivot are on the right.
     3. Place the pivot in its correct position and get the pivot index pi.
   * **Recursively** apply Quick Sort to the subarray before pi (arr[low..pi-1]).
   * **Recursively** apply Quick Sort to the subarray after pi (arr[pi+1..high]).
4. **End**.

### Merge Sort Algorithm

1. **Start**.
2. **Input**: Array arr, low index l, high index h.
3. **If** l < h:
   * **Find** the middle point m as l + (h - l) / 2.
   * **Recursively** apply Merge Sort to the left subarray (arr[l..m]).
   * **Recursively** apply Merge Sort to the right subarray (arr[m+1..h]).
   * **Merge** the two sorted subarrays:
     1. Create temporary arrays L and R.
     2. Copy data to L and R.
     3. Merge the temporary arrays back into arr[l..h].
4. **End**.

**Program 1:**

**#include <stdio.h>**

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

**{**

**int temp = \*a;**

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

**\*b = temp;**

**}**

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

**{**

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

**int i = low;**

**int j = high;**

**while (i < j) {**

**while (arr[i] <= pivot && i <= high - 1) {**

**i++;**

**}**

**while (arr[j] > pivot && j >= low + 1) {**

**j--;**

**}**

**if (i < j) {**

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

**}**

**}**

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

**return j;**

**}**

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

**{**

**if (low < high) {**

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

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

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

**}**

**}**

**int main()**

**{**

**int arr[] = { 19, 17, 15, 12, 16, 18, 4, 11, 13 };**

**int n = sizeof(arr) / sizeof(arr[0]);**

**printf("Original array: ");**

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

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

**}**

**quickSort(arr, 0, n - 1);**

**printf("\nSorted array: ");**

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

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

**}**

**return 0;**

**}**

**Program 2:**

**#include <stdio.h>**

**#include <stdlib.h>**

**void merge(int arr[], int l, int m, int r)**

**{**

**int i, j, k;**

**int n1 = m - l + 1;**

**int n2 = r - m;**

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

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

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

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

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

**i = 0;**

**j = 0;**

**k = l;**

**while (i < n1 && j < n2) {**

**if (L[i] <= R[j]) {**

**arr[k] = L[i];**

**i++;**

**}**

**else {**

**arr[k] = R[j];**

**j++;**

**}**

**k++;**

**}**

**while (i < n1) {**

**arr[k] = L[i];**

**i++;**

**k++;**

**}**

**while (j < n2) {**

**arr[k] = R[j];**

**j++;**

**k++;**

**}**

**}**

**void mergeSort(int arr[], int l, int r)**

**{**

**if (l < r) {**

**int m = l + (r - l) / 2;**

**mergeSort(arr, l, m);**

**mergeSort(arr, m + 1, r);**

**merge(arr, l, m, r);**

**}**

**}**

**void printArray(int A[], int size)**

**{**

**int i;**

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

**printf("%d ", A[i]);**

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

**}**

**int main()**

**{**

**int arr[] = { 12, 11, 13, 5, 6, 7 };**

**int arr\_size = sizeof(arr) / sizeof(arr[0]);**

**printf("Given array is \n");**

**printArray(arr, arr\_size);**

**mergeSort(arr, 0, arr\_size - 1);**

**printf("\nSorted array is \n");**

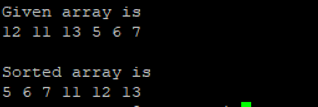
**printArray(arr, arr\_size);**

**return 0;**

**}**

**Output:**

****

****

**16: 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:**

### (i) Open Addressing (Linear Probing)

1. **Start**.
2. **Input**: Hash table HT, size m, key k.
3. **Compute** the initial hash index i = hash(k) % m.
4. **While** HT[i] is not empty and HT[i] does not contain k:
   * **Increment** i (i.e., i = (i + 1) % m).
5. **If** HT[i] is empty, **insert** k at HT[i].
6. **Else**, handle collision if necessary.
7. **End**.

### (ii) Closed Addressing (Chaining)

1. **Start**.
2. **Input**: Hash table HT (array of linked lists), key k.
3. **Compute** the hash index i = hash(k) % m.
4. **If** HT[i] is NULL:
   * **Create** a new linked list at HT[i].
5. **Insert** k at the beginning (or end) of the linked list HT[i].
6. **End**.

### (iii) Rehashing

1. **Start**.
2. **Input**: Hash table HT, size m, load factor threshold threshold.
3. **If** the load factor (number of elements / size of table) exceeds threshold:
   * **Create** a new hash table HT' with a larger size, typically 2\*m.
   * **For each** element k in HT:
     + **Compute** the new hash index i = hash(k) % (2\*m).
     + **Insert** k into HT' using open or closed addressing.
   * **Set** HT = HT' and update m.
4. **End**.

**Program 1:**

#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };

int b[10];

void merging(int low, int mid, int high) {

   int l1, l2, i;

   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {

      if(a[l1] <= a[l2])

         b[i] = a[l1++];

      else

         b[i] = a[l2++];

   }

   while(l1 <= mid)

      b[i++] = a[l1++];

   while(l2 <= high)

      b[i++] = a[l2++];

   for(i = low; i <= high; i++)

      a[i] = b[i];

}

void sort(int low, int high) {

   int mid;

   if(low < high) {

      mid = (low + high) / 2;

      sort(low, mid);

      sort(mid+1, high);

      merging(low, mid, high);

   } else {

      return;

   }

}

int main() {

   int i;

   printf("List before sorting\n");

   for(i = 0; i <= max; i++)

      printf("%d ", a[i]);

   sort(0, max);

   printf("\nList after sorting\n");

   for(i = 0; i <= max; i++)

      printf("%d ", a[i]);

}

**Program 2:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Node {

    int key;

    int value;

    struct Node\* next;

} Node;

typedef struct HashTable {

    int size;

    Node\*\* table;

} HashTable;

Node\* createNode(int key, int value) {

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

    newNode->key = key;

    newNode->value = value;

    newNode->next = NULL;

    return newNode;

}

HashTable\* createTable(int size) {

    HashTable\* newTable = (HashTable\*)malloc(sizeof(HashTable));

    newTable->size = size;

    newTable->table = (Node\*\*)malloc(sizeof(Node\*) \* size);

    for (int i = 0; i < size; i++) {

        newTable->table[i] = NULL;

    }

    return newTable;

}

int hashFunction(int key, int size) {

    return key % size;

}

void insert(HashTable\* hashTable, int key, int value) {

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

    Node\* newNode = createNode(key, value);

    newNode->next = hashTable->table[hashIndex];

    hashTable->table[hashIndex] = newNode;

}

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

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

    Node\* current = hashTable->table[hashIndex];

    while (current != NULL) {

        if (current->key == key) {

            return current->value;

        }

        current = current->next;

    }

    return -1;

}

void delete(HashTable\* hashTable, int key) {

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

    Node\* current = hashTable->table[hashIndex];

    Node\* prev = NULL;

    while (current != NULL && current->key != key) {

        prev = current;

        current = current->next;

    }

    if (current == NULL) {

        return;

    }

    if (prev == NULL) {

        hashTable->table[hashIndex] = current->next;

    } else {

        prev->next = current->next;

    }

    free(current);

}

void freeTable(HashTable\* hashTable) {

    for (int i = 0; i < hashTable->size; i++) {

        Node\* current = hashTable->table[i];

        while (current != NULL) {

            Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(hashTable->table);

    free(hashTable);

}

int main() {

    HashTable\* hashTable = createTable(10);

    insert(hashTable, 1, 10);

    insert(hashTable, 2, 20);

    insert(hashTable, 12, 30);

    printf("Value for key 1: %d\n", search(hashTable, 1));

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

    printf("Value for key 12: %d\n", search(hashTable, 12));

    printf("Value for key 3: %d\n", search(hashTable, 3)); // Key not present

    delete(hashTable, 2);

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

    freeTable(hashTable);

    return 0;

}

**Program 3:**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

    int key;

    int value;

    struct Node\* next;

} Node;

typedef struct HashTable {

    int size;

    int count;

    Node\*\* table;

} HashTable;

Node\* createNode(int key, int value) {

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

    newNode->key = key;

    newNode->value = value;

    newNode->next = NULL;

    return newNode;

}

HashTable\* createTable(int size) {

    HashTable\* newTable = (HashTable\*)malloc(sizeof(HashTable));

    newTable->size = size;

    newTable->count = 0;

    newTable->table = (Node\*\*)malloc(sizeof(Node\*) \* size);

    for (int i = 0; i < size; i++) {

        newTable->table[i] = NULL;

    }

    return newTable;

}

int hashFunction(int key, int size) {

    return key % size;

}

void insert(HashTable\* hashTable, int key, int value);

void rehash(HashTable\* hashTable) {

    int oldSize = hashTable->size;

    Node\*\* oldTable = hashTable->table;

    int newSize = oldSize \* 2;

    hashTable->table = (Node\*\*)malloc(sizeof(Node\*) \* newSize);

    hashTable->size = newSize;

    hashTable->count = 0;

    for (int i = 0; i < newSize; i++) {

        hashTable->table[i] = NULL;

    }

    for (int i = 0; i < oldSize; i++) {

        Node\* current = oldTable[i];

        while (current != NULL) {

            insert(hashTable, current->key, current->value);

            Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(oldTable);

}

void insert(HashTable\* hashTable, int key, int value) {

    if ((float)hashTable->count / hashTable->size >= 0.75) {

        rehash(hashTable);

    }

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

    Node\* newNode = createNode(key, value);

    newNode->next = hashTable->table[hashIndex];

    hashTable->table[hashIndex] = newNode;

    hashTable->count++;

}

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

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

    Node\* current = hashTable->table[hashIndex];

    while (current != NULL) {

        if (current->key == key) {

            return current->value;

        }

        current = current->next;

    }

    return -1;

}

void delete(HashTable\* hashTable, int key) {

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

    Node\* current = hashTable->table[hashIndex];

    Node\* prev = NULL;

    while (current != NULL && current->key != key) {

        prev = current;

        current = current->next;

    }

    if (current == NULL) {

        return;

    }

    if (prev == NULL) {

        hashTable->table[hashIndex] = current->next;

    } else {

        prev->next = current->next;

    }

    free(current);

    hashTable->count--;

}

void freeTable(HashTable\* hashTable) {

    for (int i = 0; i < hashTable->size; i++) {

        Node\* current = hashTable->table[i];

        while (current != NULL) {

            Node\* temp = current;

            current = current->next;

            free(temp);

        }

    }

    free(hashTable->table);

    free(hashTable);

}

int main() {

    HashTable\* hashTable = createTable(5);

    insert(hashTable, 1, 10);

    insert(hashTable, 2, 20);

    insert(hashTable, 3, 30);

    insert(hashTable, 4, 40);

    insert(hashTable, 5, 50);

    insert(hashTable, 6, 60); // This should trigger rehashing

    printf("Value for key 1: %d\n", search(hashTable, 1));

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

    printf("Value for key 3: %d\n", search(hashTable, 3));

    printf("Value for key 4: %d\n", search(hashTable, 4));

    printf("Value for key 5: %d\n", search(hashTable, 5));

    printf("Value for key 6: %d\n", search(hashTable, 6));

    delete(hashTable, 3);

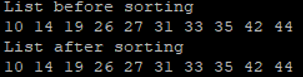
    printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));

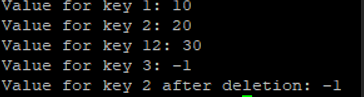
    freeTable(hashTable);

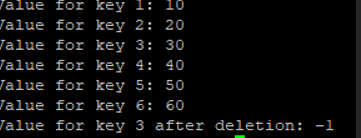
    return 0;

}

**Output:**





****