DATA STRUCTURE

REG NO 230901104 EEE-B

WEEK 1 (SINGLE LINKED LIST)

```
#include <stdio.h>
int n = 0;
void Insert(int a[], int p, int e);
void Delete(int a[], int p);
void Search(int a[], int e);
void Traverse(int a[]);
void Sort(int a[]);
int main()
{
int a[5], ch, e, p;
printf("1.Insert \n2.Delete \n3.Search");
printf("\n4.Traverse \n5.Sort \n6.Exit\n");
do
 {
 printf("\nEnter your choice : ");
 scanf("%d", &ch);
 switch(ch)
 {
 case 1:
 printf("Enter the position : ");
 scanf("%d", &p);
 printf("Enter the element : ");
 scanf("%d", &e);
 Insert(a, p, e);
 break;
 case 2:
 printf("Enter the position : ");
 scanf("%d", &p);
 Delete(a, p);
 break;
```

```
 case 3:
 printf("Enter the element : ");
 scanf("%d", &e);
 Search(a, e);
 break;
 case 4:
 printf("The elements are : ");
 Traverse(a);
 break;
 case 5:
 Sort(a);
 break;
 }
 } while(ch <= 5);
return 0;
}
void Insert(int a[], int p, int e)
{
int i;
for(i = n; i >= p; i--)
 a[i + 1] = a[i];
 a[p] = e;
 n = n + 1;
}
void Delete(int a[], int p)
{
int i;
for(i = p; i < n; i++)
 a[i] = a[i + 1];
 n = n - 1;
}
void Search(int a[], int e)
{
int i, flag = 0;
for(i = 0; i < n; i++)
 {
 if(e == a[i])
 {
```

```
 flag = 1;
 break;
 }
 }
 if(flag == 1)
 printf("Successful. Element %d is at location %d", e, i);
else
 printf("Unsuccessful.");
}
void Traverse(int a[])

{
int i;
for(i = 0; i < n; i++)
 printf("%d\t", a[i]);
}
void Sort(int a[])
{
int i, j, t;
for(i = 0; i < n-1; i++)
 {
 for(j = i + 1; j < n; j++)
 {
 if(a[i] > a[j])
 {
 t = a[i];
 a[i] = a[j];
 a[j] = t;
 }
 }
 }
}
```

OUTPUT

1.Insert

2.Delete

3.Search

4.Traverse

5.Sort

6.Exit

Enter your choice : 1

Enter the position : 0

Enter the element : 7

Enter your choice : 4

The elements are : 7

Enter your choice : 1

Enter the position : 0

Enter the element : 14

Enter your choice : 4

The elements are : 14 7

Enter your choice : 1

Enter the position : 1

Enter the element : 21

Enter your choice : 4

The elements are : 14 21 7

Enter your choice : 2

Enter the position : 1

Enter your choice : 4

The elements are : 14 7

Enter your choice : 3

Enter the element : 7

Successful. Element 7 is at location 1

WEEK 2(doubly linked list)

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
struct node *Prev;
int Element;
struct node *Next;
};
typedef struct node Node;
int IsEmpty(Node *List);
int IsLast(Node *Position);
Node *Find(Node *List, int x);
void InsertBeg(Node *List, int e);
void InsertLast(Node *List, int e);
void InsertMid(Node *List, int p, int e);
void DeleteBeg(Node *List);
void DeleteEnd(Node *List);
void DeleteMid(Node *List, int e);
void Traverse(Node *List);
int main()
{
Node *List = malloc(sizeof(Node));
 List->Prev = NULL;
 List->Next = NULL;
Node *Position;
int ch, e, p;
printf("1.Insert Beg \n2.Insert Middle \n3.Insert End");
printf("\n4.Delete Beg \n5.Delete Middle \n6.Delete End");
printf("\n7.Find \n8.Traverse \n9.Exit\n");
do
 {
 printf("Enter your choice : ");
 scanf("%d", &ch);
 switch(ch)
 {
 case 1:
```

```
printf("Enter the element : ");
scanf("%d", &e);
InsertBeg(List, e);
break;
case 2:
printf("Enter the position element : ");
scanf("%d", &p);
printf("Enter the element : ");
scanf("%d", &e);
InsertMid(List, p, e);
break;
case 3:
printf("Enter the element : ");
scanf("%d", &e);
InsertLast(List, e);
break;
case 4:
DeleteBeg(List);
break;
case 5:
printf("Enter the element : ");
scanf("%d", &e);
DeleteMid(List, e);
break;
case 6:
DeleteEnd(List);
break;
case 7:
printf("Enter the element : ");
scanf("%d", &e);
Position = Find(List, e);
if(Position != NULL)
printf("Element found...!\n");
else
printf("Element not found...!\n");
break;
case 8:
Traverse(List);
```

```
 break;
 }
 } while(ch <= 8);
return 0;
}
int IsEmpty(Node *List)
{
if(List->Next == NULL)
 return 1;
else
 return 0;
}
int IsLast(Node *Position)
{
if(Position->Next == NULL)
 return 1;
else
 return 0;
}
Node *Find(Node *List, int x)
{
Node *Position;
 Position = List->Next;
while(Position != NULL && Position->Element != x)
 Position = Position->Next;
return Position;
}
void InsertBeg(Node *List, int e)
{
Node *NewNode = malloc(sizeof(Node));
 NewNode->Element = e;
if(IsEmpty(List))
 NewNode->Next = NULL;
else
 {
 NewNode->Next = List->Next;
 NewNode->Next->Prev = NewNode;
 }
```

```
 NewNode->Prev = List;
 List->Next = NewNode;
}
void InsertLast(Node *List, int e)
{
Node *NewNode = malloc(sizeof(Node));
Node *Position;
 NewNode->Element = e;
 NewNode->Next = NULL;
if(IsEmpty(List))
 {
 NewNode->Prev = List;
 List->Next = NewNode;
 }
else
 {
 Position = List;
 while(Position->Next != NULL)
 Position = Position->Next;
 Position->Next = NewNode;
 NewNode->Prev = Position;
 }
}
void InsertMid(Node *List, int p, int e)
{
Node *NewNode = malloc(sizeof(Node));
Node *Position;
 Position = Find(List, p);
 NewNode->Element = e;
 NewNode->Next = Position->Next;
 Position->Next->Prev = NewNode;
 Position->Next = NewNode;
 NewNode->Prev = Position;
}
void DeleteBeg(Node *List)
{
if(!IsEmpty(List))
 {
```

```
 Node *TempNode;
 TempNode = List->Next;
 List->Next = TempNode->Next;
 if(List->Next != NULL)
 TempNode->Next->Prev = List;
 printf("The deleted item is %d\n", TempNode->Element);
 free(TempNode);
 }
else
 printf("List is empty...!\n");
}
void DeleteEnd(Node *List)
{
if(!IsEmpty(List))
 {
 Node *Position;
 Node *TempNode;
 Position = List;
 while(Position->Next != NULL)
 Position = Position->Next;
 TempNode = Position;
 Position->Prev->Next = NULL;
 printf("The deleted item is %d\n", TempNode->Element);
 free(TempNode);
 }
else
 printf("List is empty...!\n");
}
void DeleteMid(Node *List, int e)
{
if(!IsEmpty(List))
 {
 Node *Position;
 Node *TempNode;
 Position = Find(List, e);
 if(!IsLast(Position))
 {
 TempNode = Position;
```

```
 Position->Prev->Next = Position->Next;
 Position->Next->Prev = Position->Prev;
 printf("The deleted item is %d\n", TempNode->Element);
 free(TempNode);
 }
 }
else
 printf("List is empty...!\n");
}
void Traverse(Node *List)
{
if(!IsEmpty(List))
 {
 Node *Position;
 Position = List;
 while(Position->Next != NULL)
 {
 Position = Position->Next;
 printf("%d\t", Position->Element);
 }
 printf("\n");
 }
else
 printf("List is empty...!\n");
}
```

Output

```
Enter your choice :
1.Insert Beg
2.Insert Middle
3.Insert End
4.Delete Beg
5.Delete Middle
6.Delete End
7.Find
8.Traverse
9.Exit
Enter your choice : 1
Enter the element : 40
```

Enter your choice : 1

Enter the element : 30

Enter your choice : 1

Enter the element : 20

Enter your choice : 1

Enter the element : 10

Enter your choice : 8

10 20 30 40

Enter your choice : 7

Enter the element : 30

Element found...!

Enter your choice : 1

Enter the element : 5

Enter your choice : 8

5 10 20 30 40

Enter your choice : 3

Enter the element : 45

Enter your choice : 8

5 10 20 30 40 45

Enter your choice : 2

Enter the position element : 20

Enter the element : 25

Enter your choice : 8

5 10 20 25 30 40 45

Enter your choice : 4

The deleted item is 5

Enter your choice : 8

10 20 25 30 40 45

Enter your choice : 6

The deleted item is 45

Enter your choice : 8

10 20 25 30 40

Enter your choice : 5

Enter the element : 30

The deleted item is 30

Enter your choice : 8

10 20 25 40

Enter your choice : 9

Week 3

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

struct Term {
 int coefficient;
 int exponent;
 struct Term *next;
};
typedef struct Term Term;

Term *createTerm(int coeff, int exp) {
 Term *newTerm = (Term *)malloc(sizeof(Term));
 if (newTerm == NULL) {
 printf("Memory allocation failed\n");
 exit(1);
 }
 newTerm->coefficient = coeff;
 newTerm->exponent = exp;
 newTerm->next = NULL;
 return newTerm;
}

void insertTerm(Term **poly, int coeff, int exp) {
 Term *newTerm = createTerm(coeff, exp);
 if (*poly == NULL) {
 *poly = newTerm;
 } else {
 Term *temp = *poly;
 while (temp->next != NULL) {
 temp = temp->next;
 }
 temp->next = newTerm;
 }
}

void displayPolynomial(Term *poly) {
 if (poly == NULL) {
```

```
 printf("Polynomial is empty\n");
 } else {
 while (poly != NULL) {
 printf("(%dx^%d) ", poly->coefficient, poly->exponent);
 poly = poly->next;
 if (poly != NULL) {
 printf("+ ");
 }
 }
 printf("\n");
 }
}

Term *addPolynomials(Term *poly1, Term *poly2) {
 Term *result = NULL;
 while (poly1 != NULL && poly2 != NULL) {
 if (poly1->exponent > poly2->exponent) {
 insertTerm(&result, poly1->coefficient, poly1->exponent);
 poly1 = poly1->next;
} else if (poly1->exponent < poly2->exponent) {
 insertTerm(&result, poly2->coefficient, poly2->exponent);
 poly2 = poly2->next;
 } else {
 insertTerm(&result, poly1->coefficient + poly2->coefficient, poly1->exponent);
 poly1 = poly1->next;
 poly2 = poly2->next;
 }
 }
 while (poly1 != NULL) {
 insertTerm(&result, poly1->coefficient, poly1->exponent);
 poly1 = poly1->next;
 }
 while (poly2 != NULL) {
 insertTerm(&result, poly2->coefficient, poly2->exponent);
 poly2 = poly2->next;
 }
 return result;
}
```

```
Term *subtractPolynomials(Term *poly1, Term *poly2) {
 Term *result = NULL;
 while (poly1 != NULL && poly2 != NULL) {
 if (poly1->exponent > poly2->exponent) {
 insertTerm(&result, poly1->coefficient, poly1->exponent);
 poly1 = poly1->next;
 } else if (poly1->exponent < poly2->exponent) {
 insertTerm(&result, -poly2->coefficient, poly2->exponent);
 poly2 = poly2->next;
 } else {
 insertTerm(&result, poly1->coefficient - poly2->coefficient, poly1->exponent);
 poly1 = poly1->next;
 poly2 = poly2->next;
 }
 }
 while (poly1 != NULL) {
 insertTerm(&result, poly1->coefficient, poly1->exponent);
 poly1 = poly1->next;
 }
 while (poly2 != NULL) {
 insertTerm(&result, -poly2->coefficient, poly2->exponent);
 poly2 = poly2->next;
 }
 return result;
}

Term *multiplyPolynomials(Term *poly1, Term *poly2) {
 Term *result = NULL;
 Term *temp1 = poly1;
 while (temp1 != NULL) {
 Term *temp2 = poly2;
 while (temp2 != NULL) {
 insertTerm(&result, temp1->coefficient * temp2->coefficient, temp1->exponent + temp2->exponent);
temp2 = temp2->next;
 }
 temp1 = temp1->next;
 }
```

```
 return result;
}

int main() {
 Term *poly1 = NULL;
 Term *poly2 = NULL;

 insertTerm(&poly1, 5, 2);
 insertTerm(&poly1, -3, 1);
 insertTerm(&poly1, 2, 0);

 insertTerm(&poly2, 4, 3);
 insertTerm(&poly2, 2, 1);
 printf("Polynomial 1: ");
 displayPolynomial(poly1);
 printf("Polynomial 2: ");
 displayPolynomial(poly2);
 Term *sum = addPolynomials(poly1, poly2);
 printf("Sum: ");
 displayPolynomial(sum);
 Term *difference = subtractPolynomials(poly1, poly2);
 printf("Difference: ");
 displayPolynomial(difference);
 Term *product = multiplyPolynomials(poly1, poly2);
 printf("Product: ");
 displayPolynomial(product);
 return 0;
}
```

Output

```
Polynomial 1: (5x^2) + (-3x^1) + (2x^0)
Polynomial 2: (4x^3) + (2x^1)
Sum: (4x^3) + (5x^2) + (-1x^1) + (2x^0)
Difference: (-4x^3) + (5x^2) + (-5x^1) + (2x^0)
Product: (20x^5) + (10x^3) + (-12x^4) + (-6x^2) + (8x^3) + (4x^1)
```

Week 4

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

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

struct StackLL {
 struct Node* top;
};

struct StackArray {
 int* array;
 int top;
 int capacity;
};

struct StackLL* createStackLL() {
 struct StackLL* stack = (struct StackLL*)malloc(sizeof(struct StackLL));
 stack->top = NULL;
 return stack;
}

struct StackArray* createStackArray(int capacity) {
 struct StackArray* stack = (struct StackArray*)malloc(sizeof(struct StackArray));
 stack->capacity = capacity;
 stack->top = -1;
 stack->array = (int*)malloc(stack->capacity * sizeof(int));
 return stack;
}

int isEmptyLL(struct StackLL* stack) {
 return stack->top == NULL;
}

int isEmptyArray(struct StackArray* stack) {
```

```
 return stack->top == -1;
}

void pushLL(struct StackLL* stack, int data) {
 struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
 newNode->data = data;
 newNode->next = stack->top;
 stack->top = newNode;
}

void pushArray(struct StackArray* stack, int data) {
 if (stack->top == stack->capacity - 1) {
 printf("Stack Overflow\n");
 return;
 }
 stack->array[++stack->top] = data;
}

int popLL(struct StackLL* stack) {
 if (isEmptyLL(stack)) {
 printf("Stack Underflow\n");
 return -1;
 }
 struct Node* temp = stack->top;
 int data = temp->data;
 stack->top = stack->top->next;
 free(temp);
 return data;
}

int popArray(struct StackArray* stack) {
 if (isEmptyArray(stack)) {
 printf("Stack Underflow\n");
 return -1;
 }
 return stack->array[stack->top--];
}
```

```
int peekLL(struct StackLL* stack) {
 if (isEmptyLL(stack)) {
 printf("Stack is empty\n");
 return -1;
 }
 return stack->top->data;
}

int peekArray(struct StackArray* stack) {
 if (isEmptyArray(stack)) {
 printf("Stack is empty\n");
 return -1;
 }
 return stack->array[stack->top];
}

void displayLL(struct StackLL* stack) {
 if (isEmptyLL(stack)) {
 printf("Stack is empty\n");
 return;
 }
 struct Node* temp = stack->top;
 printf("Elements in stack: ");
 while (temp != NULL) {
 printf("%d ", temp->data);
 temp = temp->next;
 }
 printf("\n");
}

void displayArray(struct StackArray* stack) {
 if (isEmptyArray(stack)) {
 printf("Stack is empty\n");
 return;
 }
 printf("Elements in stack: ");
 for (int i = stack->top; i >= 0; i--) {
 printf("%d ", stack->array[i]);
```

```
 }
 printf("\n");
}
int main() {
 // Test linked list implementation
 struct StackLL* stackLL = createStackLL();
 pushLL(stackLL, 1);
 pushLL(stackLL, 2);
 pushLL(stackLL, 3);
 displayLL(stackLL);
 printf("Top element: %d\n", peekLL(stackLL));
 printf("Popped element: %d\n", popLL(stackLL));
 displayLL(stackLL);

 struct StackArray* stackArray = createStackArray(5);
 pushArray(stackArray, 4);
 pushArray(stackArray, 5);
 pushArray(stackArray, 6);
 displayArray(stackArray);
 printf("Top element: %d\n", peekArray(stackArray));
 printf("Popped element: %d\n", popArray(stackArray));
 displayArray(stackArray);
 return 0;
}
```

Output

```
Elements in stack: 3 2 1
Top element: 3
Popped element: 3
Elements in stack: 2 1
Elements in stack: 6 5 4
Top element: 6
Popped element: 6
Elements in stack: 5 4
```

WEEK 5

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX_SIZE 100

struct Stack {
 int top;
 unsigned capacity;
 char *array;
};

struct Stack* createStack(unsigned capacity) {
 struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
 stack->capacity = capacity;
 stack->top = -1;
 stack->array = (char*) malloc(stack->capacity * sizeof(char));
 return stack;
}

int isFull(struct Stack* stack) {
 return stack->top == stack->capacity - 1;
}

int isEmpty(struct Stack* stack) {
 return stack->top == -1;
}

void push(struct Stack* stack, char item) {
 if (isFull(stack))
 return;
 stack->array[++stack->top] = item;
}

char pop(struct Stack* stack) {
 if (isEmpty(stack))
 return '\0';
```

```
 return stack->array[stack->top--];
}

int precedence(char op) {
 if (op == '+' || op == '-')
 return 1;
 else if (op == '*' || op == '/')
 return 2;
 else
 return -1;
}

void infixToPostfix(char* infix, char* postfix) {
 struct Stack* stack = createStack(strlen(infix));
 int i, j;
 for (i = 0, j = -1; infix[i]; ++i) {
if (isalnum(infix[i]))
 postfix[++j] = infix[i];
 else if (infix[i] == '(')
 push(stack, '(');
 else if (infix[i] == ')') {
 while (!isEmpty(stack) && stack->array[stack->top] != '('
 postfix[++j] = pop(stack);
 if (!isEmpty(stack) && stack->array[stack->top] != '('
 return;
 else
 pop(stack);
 } else {

while (!isEmpty(stack) && precedence(infix[i]) <= precedence(stack->array->top))



 postfix[++j] = pop(stack);
 push(stack, infix[i]);
 }
 }
 while (!isEmpty(stack))
 postfix[++j] = pop(stack);
```

```
 postfix[++j] = '\0';
}

int main() {
 char infix[MAX_SIZE];
 char postfix[MAX_SIZE];
 printf("Enter an infix expression: ");
 fgets(infix, MAX_SIZE, stdin);
 infix[strcspn(infix, "\n")] = 0;
 infixToPostfix(infix, postfix);
 printf("Postfix expression: %s\n", postfix);
 return 0;
}
```

Output

```
Enter the infixt expression:((a+b)*(c+d)*(e/f)*
Postfix expression is:ab+cd+*ef/*g^
```

WEEK 6

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX_SIZE 100
int stack[MAX_SIZE];
int top = -1;
void push(int item) {
if (top >= MAX_SIZE - 1) {
printf("Stack Overflow\n");
} else {
top++;
stack[top] = item;
}
}
int pop() {
if (top < 0) {
printf("Stack Underflow\n");
return -1;
} else {
return stack[top--];
}
}
int evaluateExpression(char* exp) {
int i, operand1, operand2, result;
for (i = 0; exp[i] != '\0'; i++) {
if (isdigit(exp[i])) {
push(exp[i] - '0');
} else {
operand2 = pop();
operand1 = pop();
switch (exp[i]) {
case '+':
push(operand1 + operand2);
break;
case '-':
```

```
push(operand1 - operand2);
break;
case '*':
push(operand1 * operand2);
break;
case '/':
push(operand1 / operand2);
break;
}
}
}
result = pop();
return result;
}
int main() {
char exp[MAX_SIZE];
printf("Enter the arithmetic expression: ");
scanf("%s", exp);
int result = evaluateExpression(exp);
printf("Result: %d\n", result);
return 0;
}
```

OUTPUT

```
Enter the arithmetic expression: 55
Result: 5
```

WEEK 7

```
int main()
{
   int ch; #include <stdio.h>
#include <stdlib.h>
struct Queue
{ int ele;
 struct Queue *next;};
typedef struct Queue q
q *r=NULL;
;
q *f=NULL;

void enqueue(int x)
{ q *newnode=malloc(sizeof(q));
 newnode->ele=x;
 if(f==NULL && r==NULL)
 { f=r=newnode;
   newnode->next=NULL;
   return;
 }
 r->next=newnode;
 r=newnode;
 newnode->next=NULL;}
 //f=newnode;}

void dequeue()
{  if(f==NULL && r==NULL)
  { printf("UNDERFLOW\n");
    return;}
  if(f==r)
  { printf("THE DELETED ELE IS %d\n",f->ele);
    f=r=NULL;
    return;}
   q *temp=f;
   printf("DELETED ELEMENT IS %d\n",temp->ele);
   f=f->next;
   free(temp);
```

```
}

void display()
{  q *temp=f;
   while(temp!=NULL)
   { printf("%d ",temp->ele);
     temp=temp->next;
   }
   printf("\n");
}

int main()
{
    int ch;
    printf("1 TO ENQUEUE\n2 TO DEQUEUE\n3 TO DISPLAY\n");
    do
    { printf("ENTER YOUR CHOICE ");
      scanf("%d",&ch);
      switch(ch)
      { case 1:
           int x;
           printf("ELEMENT TO BE ADDED");
           scanf("%d",&x);
           enqueue(x);
           break;
        case 2:
          dequeue();
          break;

       case 3:
          display();
          break;
       default:
         break;
     } } while(ch<=3);

    printf("THANK YOU");
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 100

int q[SIZE];
int f=-1,r=-1;

void enqueue(int x)
{ if(f==-1 && r==-1)
  { f++;
    r++;
    q[f]=x;
    return;
  }
  if(r==SIZE-1)
  { printf("OVERFLOW\n");
    return;}
  r++;
  q[r]=x;
}

void dequeue()
{ if(f==-1 && r==-1)
  { printf("UNDERFLOW\n");
    return;}
  if(f==r)
  { printf("THE DELETED ELE %d\n",q[f]);
    f=r=-1;
    return;}
  printf("The deleted element is %d\n",q[f]);
  f++;
}

void display()
{ for(int i=f;i<=r;i++)
  { printf("%d ",q[i]);
  }
  printf("\n");}
```

```
   printf("1 TO ENQUEUE\n2 TO DEQUEUE\n3 TO DISPLAY\n");
   do
   { printf("ENTER YOUR CHOICE ");
     scanf("%d",&ch);
     switch(ch)
     { case 1:
          int x;
          printf("ELEMENT TO BE ADDED");
          scanf("%d",&x);
          enqueue(x);
          break;
       case 2:
          dequeue();
          break;

       case 3:
          display();
          break;
       default:
         break;
     } } while(ch<=3);

   printf("THANK YOU");
}
```

OUTPUT:-

1 TO ENQUEUE

2 TO DEQUEUE

3 TO DISPLAY

ENTER YOUR CHOICE 1

ELEMENT TO BE ADDED20

ENTER YOUR CHOICE 1

ELEMENT TO BE ADDED30

ENTER YOUR CHOICE 1

ELEMENT TO BE ADDED40

ENTER YOUR CHOICE 3

20 30 40

ENTER YOUR CHOICE 2

DELETED ELEMENT IS 20

ENTER YOUR CHOICE 3

30 40

ENTER YOUR CHOICE 4

WEEK 8

```
#include <stdio.h>
#include <stdlib.h>
  struct Tree
  { int ele;
    struct Tree *left;
    struct Tree *right;};
  typedef struct Tree tree;
  //tree *root=NULL;

  tree *create(tree *root,int x)
  { if(root==NULL)
    { tree *newnode=malloc(sizeof(tree));
      newnode->ele=x;
      newnode->left=NULL;
      newnode->right=NULL;
      root=newnode;}
    else if(x<root->ele)
    { root->left=create(root->left,x);
    }
    else if(x>root->ele)
    { root->right=create(root->right,x);
    }
    return root;
  }

  void inorder(tree *root)
  { if(root!=NULL)
    { inorder(root->left);
      printf("%d ",root->ele);
      inorder(root->right);
    }
```

```
}
void preorder(tree *root)
{ if(root!=NULL)
  {
    printf("%d ",root->ele);
    preorder(root->left);
    preorder(root->right);
  }

}
void postorder(tree *root)
{ if(root!=NULL)
  {
    postorder(root->left);
    postorder(root->right);
    printf("%d ",root->ele);
  }

}

int main()
{   tree *root=NULL;
    int n,x;
    printf("ENTER NO OF ELEMENTS");
    scanf("%d",&n);
    printf("ENTER THE ELEMENTS ");
    for(int i=0;i<n;i++)
    {  scanf("%d",&x);
       root=create(root,x);
    }
    printf("INORDER TRAVERSAL IS ");
    inorder(root);
    printf("\nPOSTORDER TRAVERSAL IS ");
    postorder(root);
    printf("\nPREORDER TRAVERSAL IS ");
    preorder(root);
```

```
    return 0;
}
```

OUTPUT:-

ENTER NO OF ELEMENTS7

ENTER THE ELEMENTS 100 90 110 80 95 105 111

INORDER TRAVERSAL IS 80 90 95 100 105 110 111

POSTORDER TRAVERSAL IS 80 95 90 105 111 110 100

PREORDER TRAVERSAL IS 100 90 80 95 110 105 111

WEEK 9

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

struct Node {
    int data;
    struct Node *left, *right;
};

// Function to create a new BST node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) return createNode(data);

    if (data < root->data)

root->left = insertNode(root->left, data);
else if (data > root->data)
        root->right = insertNode(root->right, data);

    return root;
}
```

```
struct Node* findMin(struct Node* node) {
    struct Node* current = node;

    while (current && current->left != NULL)
        current = current->left;

    return current;
}

struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else {
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}

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

```
    if (root == NULL || root->data == data)
        return root;

    if (root->data < data)
        return searchNode(root->right, data);

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

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

int main() {
    struct Node* root = NULL;
    int choice, data,n;
    printf("Enter the no of elements to be inserted");
    scanf("%d",&n);
    printf("Enter elements");
    for(int i=0;i<n;i++)
    { scanf("%d",&data);
      root=insertNode(root, data);}
    while (1) {
        printf("\nBinary Search Tree Operations Menu\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```
            printf("Enter data to insert: ");
            scanf("%d", &data);
            root = insertNode(root, data);
            printf("%d inserted.\n", data);
            break;

        case 2:
            printf("Enter data to delete: ");
            scanf("%d", &data);
            root = deleteNode(root, data);
            printf("%d deleted.\n", data);
            break;

        case 3:
            printf("Enter data to search: ");
            scanf("%d", &data);
            struct Node* foundNode = searchNode(root, data);
            if (foundNode != NULL)
                printf("%d found in the tree.\n", data);
            else
                printf("%d not found in the tree.\n", data);
            break;

        case 4:
            printf("In-order display of the BST: ");
            inOrder(root);
            printf("\n");
            break;

        case 5:
            exit(0);
            break;

        default:
            printf("Invalid choice! Please try again.\n");
    }
}
```

```
    return 0;
}
```

OUTPUT:-

```
Enter the no of elements to be inserted6
Enter elements100 90 110 80 95 105

Binary Search Tree Operations Menu
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 4
In-order display of the BST: 80 90 95 100 105 110

Binary Search Tree Operations Menu
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 2
Enter data to delete: 90
90 deleted.

Binary Search Tree Operations Menu
1. Insert
2. Delete
3. Search
4. Display
5. Exit
Enter your choice: 4
In-order display of the BST: 80 95 100 105 110

Binary Search Tree Operations Menu
1. Insert
2. Delete
```

3. Search

4. Display

5. Exit

Enter your choice: 3

Enter data to search: 80

80 found in the tree.

Binary Search Tree Operations Menu

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 5

WEEK 10

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

struct Node
{
   int key;
   struct Node *left;
   struct Node *right;
   int height;
};
typedef struct Node node;
int height(node *n)
{
   if (n==NULL)
      return 0;
   return n->height;
}
node *findmin(node *tree)
{ if(tree==NULL)
  return NULL;
```

```
 else if(tree->left==NULL)
    return tree;
  else
    return findmin(tree->left);
}

int max(int a,int b)
{ return (a>b)?a:b;
}
node *rightrotate(node *y)
{ node *x=y->left;
  node *t2=x->right;

  x->right=y;
  y->left=t2;
  y->height=1+max(height(y->left),height(y->right));
  x->height=1+max(height(x->left),height(x->right));
  return x;
}

node *leftrotate(node *x)
{ node *y=x->right;
  node *t2=y->left;

  y->left=x;
  x->right=t2;
  x->height=1+max(height(x->left),height(x->right));
  y->height=1+max(height(y->left),height(y->right));
  return y;
}
int getbalance(struct Node *n)
{
   if (n == NULL)
       return 0;
   return height(n->left) - height(n->right);
}
```

```
node *insert(node *tree,int k)
{ if(tree==NULL)
  { node *newnode=malloc(sizeof(node));
    newnode->key=k;
    newnode->left=NULL;
    newnode->right=NULL;
    newnode->height=1;
    tree=newnode;
  }

  else if(k<tree->key)
       tree->left=insert(tree->left,k);
  else if(k>tree->key)
       tree->right=insert(tree->right,k);
  //else
     //return tree;
  tree->height=1+max(height(tree->left),height(tree->right));
  int bal=getbalance(tree);
  if(bal>1 && k<tree->left->key)
       return rightrotate(tree);
  if(bal<-1 && k>tree->right->key)
       return leftrotate(tree);
  if(bal>1 && k>tree->left->key)
  {    tree->left=leftrotate(tree->left);
       return rightrotate(tree);  }
  if(bal<-1 && k<tree->right->key)
  {    tree->right=rightrotate(tree->right);
       return leftrotate(tree);  }
  return tree;
}

node *delete(node *tree,int e)
{  node *temp=malloc(sizeof(node));
   if(e<tree->key)
      tree->left=delete(tree->left,e);
   else if(e>tree->key)
      tree->right=delete(tree->right,e);
   else if(tree->left && tree->right)
```

```
 { temp=findmin(tree->right);
   tree->key=temp->key;
   tree->right=delete(tree->right,temp->key);
 }
else
{ temp=tree;
  if(tree->left==NULL)
     tree=tree->right;
  else if(tree->right==NULL)
    tree=tree->left;
   free(temp);
}
if (tree == NULL)
    return tree;

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

int balance = getbalance(tree);

 if (balance > 1 &&
    getbalance(tree->left) >= 0)
    return rightrotate(tree);

 // Left Right Case
 if (balance > 1 &&
    getbalance(tree->left) < 0)
 {
    tree->left = leftrotate(tree->left);
    return rightrotate(tree);
 }

 // Right Right Case
 if (balance < -1 &&
    getbalance(tree->right) <= 0)
    return leftrotate(tree);

 // Right Left Case
```

```
    if (balance < -1 &&
        getbalance(tree->right) > 0)
    {
        tree->right = rightrotate(tree->right);
        return leftrotate(tree);
    }

    return tree;
}
void inorder(node *tree)
{ if(tree!=NULL)
  { //printf("%d ",tree->key);
    inorder(tree->left);
    printf("%d ",tree->key);
    inorder(tree->right);}
}

int main()
{
    node *tree=NULL;
    int n;
    printf("ENTER TOT NO OF ELEMENTS");
    scanf("%d",&n);
    int e;
    printf("ENETR ELEMENTS");
    for(int i=0;i<n;i++)
    {    scanf("%d",&e);
        tree=insert(tree,e);}
    //inorder(tree);
    printf("ENETR ELE TO BE DELETED");
    scanf("%d",&e);
    tree = delete(tree,e);
    inorder(tree);


    return 0;
}
```

OUTPUT:-

ENTER TOT NO OF ELEMENTS9

ENETR ELEMENTS9 5 10 0 6 11 -1 1 2

ENETR ELE TO BE DELETED10

-1 0 1 2 5 6 9 11

WEEK 11

```
#include <stdlib.h>

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

node *create(int data) {
    node *N = malloc(sizeof(node));
    N->data = data;
    N->next = NULL;
    return N;
}

struct Queue {
    int ele;
    struct Queue *next;
};
typedef struct Queue q;
q *f = NULL;
q *r = NULL;

void enqueue(int ele) {
    q *newnode = malloc(sizeof(q));
    newnode->ele = ele;
    newnode->next = NULL;
    if (f == NULL && r == NULL) {
        f = r = newnode;
```

```
        return;
    }
    r->next = newnode;
    r = newnode;
}

int dequeue() {
    if (f == NULL) {
        return -1; // Return -1 if the queue is empty
    }
    q *temp = f;
    f = f->next;
    int s = temp->ele;
    free(temp);
    if (f == NULL) {
        r = NULL; // Update rear pointer if the queue becomes empty
    }
    return s;
}

void addedge(node *adj[], int u, int v) {
    node *newnode = create(v);
    newnode->next = adj[u];
    adj[u] = newnode;
}

void bfs(node *adj[], int si, int v) {
    int visited[v];
    for (int i = 0; i < v; ++i) {
        visited[i] = 0;
    }

    enqueue(si);
    visited[si] = 1;

    while (f != NULL) {
        int u = dequeue();
        printf("%d ", u);
```

```
        node *temp = adj[u];
        while (temp != NULL) {
            int d = temp->data;
            if (!visited[d]) {
                visited[d] = 1;
                enqueue(d);
            }
            temp = temp->next;
        }
    }
    printf("\n");
}

int main() {
    int vertices = 5;

tices; ++i)
        adjList[i] = NULL;

    // Add edges to the graph
    addedge(adjList, 0, 1);
    addedge(adjList, 0, 2);
    addedge(adjList, 1, 3);
    addedge(adjList, 1, 4);
    addedge(adjList, 2, 4);

    printf("Breadth First Traversal starting from vertex 0: ");
    bfs(adjList, 0, vertices);

    return 0;}
```

OUTPUT FOR BFS:-

Breadth First Traversal starting from vertex 0: 0 2 1 4 3

CODE:-

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

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

node *create(int data) {
    node *N = malloc(sizeof(node));
    N->data = data;
    N->next = NULL;
    return N;
}

void addedge(node *adj[], int u, int v) {
    node *newnode = create(v);
    newnode->next = adj[u];
    adj[u] = newnode;
}

void dfsUtil(node *adj[], int v, int visited[]) {
    visited[v] = 1;
    printf("%d ", v);

    node *temp = adj[v];
    while (temp != NULL) {
        int d = temp->data;
        if (!visited[d]) {
            dfsUtil(adj, d, visited);
        }
        temp = temp->next;
    }
}

void dfs(node *adj[], int si, int vertices) {
    int visited[vertices];
```

```
    for (int i = 0; i < vertices; ++i) {
        visited[i] = 0;
    }
    dfsUtil(adj, si, visited);
}

int main() {
    int vertices = 5;

    node *adjList[vertices];
    for (int i = 0; i < vertices; ++i)
        adjList[i] = NULL;

    addedge(adjList, 0, 1);
    addedge(adjList, 0, 2);
    addedge(adjList, 1, 3);
    addedge(adjList, 1, 4);
    addedge(adjList, 2, 4);

    printf("Depth First Traversal starting from vertex 0: ");
    dfs(adjList, 0, vertices);

    return 0;
}
```

OUTPUT FOR DFS:-

Depth First Traversal starting from vertex 0: 0 2 4 1 3

WEEK 12

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

// Structure to represent a stack
struct Stack {
        int data;
```

```
        struct Stack* next;
};

struct Graph {
        int V; // No. of vertices
        struct List* adj;
};
struct List {
        int data;
        struct List* next;
};
struct Stack* createStackNode(int data)
{
        struct Stack* newNode
                = (struct Stack*)malloc(sizeof(struct Stack));
        newNode->data = data;
        newNode->next = NULL;
        return newNode;
}

struct List* createListNode(int data)
{
        struct List* newNode
                = (struct List*)malloc(sizeof(struct List));
        newNode->data = data;
        newNode->next = NULL;
        return newNode;
}

struct Graph* createGraph(int V)
{
        struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));

        graph->V = V;
        graph->adj
                = (struct List*)malloc(V * sizeof(struct List));
        for (int i = 0; i < V; ++i) {
                graph->adj[i].next = NULL;
```

```
    }
    return graph;
}

void addEdge(struct Graph* graph, int v, int w)
{
    struct List* newNode = createListNode(w);
    newNode->next = graph->adj[v].next;
    graph->adj[v].next = newNode;
}

void topologicalSortUtil(struct Graph* graph, int v,
                                                bool visited[],
                                                struct Stack** stack)
{
    visited[v] = true;

    struct List* current = graph->adj[v].next;
    while (current != NULL) {
        int adjacentVertex = current->data;
        if (!visited[adjacentVertex]) {
            topologicalSortUtil(graph, adjacentVertex,
                                                        visited, stack);
        }
        current = current->next;
    }


    struct Stack* newNode = createStackNode(v);
    newNode->next = *stack;
    *stack = newNode;
}

void topologicalSort(struct Graph* graph)
{
    struct Stack* stack = NULL;

    bool* visited = (bool*)malloc(graph->V * sizeof(bool));
```

```
	for (int i = 0; i < graph->V; ++i) {
		visited[i] = false;
	}


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

	}

	// Print contents of stack
	while (stack != NULL) {
		printf("%d ", stack->data);
		struct Stack* temp = stack;
		stack = stack->next;
		free(temp);
	}

	// Free allocated memory
	free(visited);
	free(graph->adj);
	free(graph);
}

int main()
{

	struct Graph* g = createGraph(6);
	addEdge(g, 5, 2);
	addEdge(g, 5, 0);
	addEdge(g, 4, 0);
	addEdge(g, 4, 1);
	addEdge(g, 2, 3);
	addEdge(g, 3, 1);

	printf("Topological Sorting Order: ");
	topologicalSort(g);
```

```
    return 0;
}
```

OUTPUT:-

Topological Sorting Order:5 4 2 3 1 0

WEEK 13

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 10
#define INF 999999
int graph[MAX_VERTICES][MAX_VERTICES];
int vertices;
void createGraph() {
int i, j;
printf("Enter the number of vertices: ");
scanf("%d", &vertices);
printf("Enter the adjacency matrix:\n");
for (i = 0; i < vertices; i++) {
for (j = 0; j < vertices; j++) {
scanf("%d", &graph[i][j]);
}
}
}
int findMinKey(int key[], bool mstSet[]) {
int min = INF, min_index;
for (int v = 0; v < vertices; v++) {
if (mstSet[v] == false && key[v] < min) {
min = key[v];
min_index = v;
}
}
return min_index;
}
void printMST(int parent[]) {
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 parent[vertices];
int key[vertices];
bool mstSet[vertices];
for (int i = 0; i < vertices; i++) {
key[i] = INF;
mstSet[i] = false;
}
key[0] = 0;
parent[0] = -1;
for (int count = 0; count < vertices - 1; count++) {
int u = findMinKey(key, mstSet);
mstSet[u] = true;
for (int v = 0; v < vertices; v++) {
if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
parent[v] = u;
key[v] = graph[u][v];
}
}
}
printMST(parent);
}
int main() {
createGraph();
primMST();
return 0;
}
```

OUTPUT

```
Enter the number of vertices: 2
Enter the adjacency matrix:
22
25
26
26
```

Edge Weight

0 - 1 26

WEEK 14

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_VERTICES 10
#define INF 999999
int graph[MAX_VERTICES][MAX_VERTICES];
int vertices;
void createGraph() {
int i, j;
printf("Enter the number of vertices: ");
scanf("%d", &vertices);
printf("Enter the adjacency matrix:\n");
for (i = 0; i < vertices; i++) {
for (j = 0; j < vertices; j++) {
scanf("%d", &graph[i][j]);
}
}
}
int minDistance(int dist[], bool sptSet[]) {
int min = INF, min_index;
for (int v = 0; v < vertices; v++) {
if (sptSet[v] == false && dist[v] <= min) {
min = dist[v];
min_index = v;
}
}
return min_index;
}
void printSolution(int dist[]) {
printf("Vertex \t Distance from Source\n");
for (int i = 0; i < vertices; i++) {
printf("%d \t %d\n", i, dist[i]);
}
}void dijkstra(int src) {
```

```
int dist[vertices];
bool sptSet[vertices];
for (int i = 0; i < vertices; i++) {
dist[i] = INF;
sptSet[i] = false;
}
dist[src] = 0;
for (int count = 0; count < vertices - 1; count++) {
int u = minDistance(dist, sptSet);
sptSet[u] = true;
for (int v = 0; v < vertices; v++) {
if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v])
{
dist[v] = dist[u] + graph[u][v];
}
}
}
printSolution(dist);
}
int main() {
createGraph();
int source;
printf("Enter the source vertex: ");
scanf("%d", &source);
dijkstra(source);
return 0;
}
```

OUTPUT

```
Enter the number of vertices: 2
Enter the adjacency matrix:
22
22
22
54
Enter the source vertex: 5
Vertex   Distance from Source
 0          999999
 1          999999
```

WEEK 15

```
#include <stdio.h>
#include <stdlib.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[high];
int i = (low - 1);
for (int j = low; j <= high - 1; j++) {
if (arr[j] < pivot) {
i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]);
return (i + 1);
}
void quickSort(int arr[], int low, int high) {
if (low < high) {
int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}
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);
}
}
int main() {
int n;
printf("Enter the number of elements: ");
scanf("%d", &n);
int arr[n];
printf("Enter %d elements:\n", n);
for (int i = 0; i < n; i++) {
scanf("%d", &arr[i]);
```

```
}
printf("\nSorting using Quick Sort:\n");
quickSort(arr, 0, n - 1);
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
printf("\n\nSorting using Merge Sort:\n");
mergeSort(arr, 0, n - 1);
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

OUTPUT

```
Enter the number of elements: 3
Enter 3 elements:
123
145
639

Sorting using Quick Sort:
123 145 639

Sorting using Merge Sort:
123 145 639
```

WEEK 16

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define TABLE_SIZE 10
typedef struct Node {
int data;
struct Node* next;
} Node;
Node* createNode(int data) {
Node* newNode = (Node*)malloc(sizeof(Node));
if (newNode == NULL) {
```

```
printf("Memory allocation failed!\n");
exit(1);
}
newNode->data = data;
newNode->next = NULL;
return newNode;
}
int hashFunction(int key) {
return key % TABLE_SIZE;
}
Node* insertOpenAddressing(Node* table[], int key) {
int index = hashFunction(key);
while (table[index] != NULL) {
index = (index + 1) % TABLE_SIZE;
}
table[index] = createNode(key);
return table[index];
}
void displayHashTable(Node* table[]) {
printf("Hash Table:\n");
for (int i = 0; i < TABLE_SIZE; i++) {
printf("%d: ", i);
Node* current = table[i];
while (current != NULL) {
printf("%d ", current->data);
current = current->next;
}
printf("\n");
}
}
Node* insertClosedAddressing(Node* table[], int key) {
int index = hashFunction(key);
if (table[index] == NULL) {
table[index] = createNode(key);
} else {
Node* newNode = createNode(key);
newNode->next = table[index];
table[index] = newNode;
```

```
}
return table[index];
}
int rehashFunction(int key, int attempt) {
// Double Hashing Technique
return (hashFunction(key) + attempt * (7 - (key % 7))) % TABLE_SIZE;
}
Node* insertRehashing(Node* table[], int key) {
int index = hashFunction(key);
int attempt = 0;
while (table[index] != NULL) {
attempt++;
index = rehashFunction(key, attempt);
}
table[index] = createNode(key);
return table[index];
}
int main() {
Node* openAddressingTable[TABLE_SIZE] = {NULL};
Node* closedAddressingTable[TABLE_SIZE] = {NULL};
Node* rehashingTable[TABLE_SIZE] = {NULL};
// Insert elements into hash tables
insertOpenAddressing(openAddressingTable, 10);
insertOpenAddressing(openAddressingTable, 20);
insertOpenAddressing(openAddressingTable, 5);
insertClosedAddressing(closedAddressingTable, 10);
insertClosedAddressing(closedAddressingTable, 20);
insertClosedAddressing(closedAddressingTable, 5);
insertRehashing(rehashingTable, 10);
insertRehashing(rehashingTable, 20);
insertRehashing(rehashingTable, 5);
// Display hash tables
displayHashTable(openAddressingTable);
displayHashTable(closedAddressingTable);
displayHashTable(rehashingTable);
return 0;
}
```

OUTPUT

Hash Table:

0: 10

1: 20

2:

3:

4:

5: 5

6:

7:

8:

9:

Hash Table:

0: 20 10

1:

2:

3:

4:

5: 5

6:

7:

8:

9:

Hash Table:

0: 10

1: 20

2:

3:

4:

5: 5

6:

7:

8:

9: