1) write a C program to perform addition of all elements using calloc for a single matrix

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

int main() {
    int r, c;
    int i, j;
    int **a;  // Pointer to a pointer (2D array)
    int sum = 0;

    printf("Enter the number of rows: ");
    scanf("%d", &r);

    printf("Enter the number of columns: ");
    scanf("%d", &c);

    a = (int **)calloc(r, sizeof(int *));
    for (i = 0; i < r; i++) {
        a[i] = (int *)calloc(c, sizeof(int));
    }
    printf("Enter the elements of the matrix:\n");

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &a[i][j]);
        }
    }

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            sum += a[i][j];
        }
    }

    printf("Sum of all elements: %d\n", sum);

    for (i = 0; i < r; i++) {
        free(a[i]);
    }
    free(a);

    return 0;
}
```

----

2) Write a C program to perform addition of row wise elements for single matrix using calloc.

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

int main() {
    int r, c;
    int i, j;
    int **a;
    int *sum;

    printf("Enter the number of rows: ");
    scanf("%d", &r);

    printf("Enter the number of columns: ");
    scanf("%d", &c);

    // Allocate memory for the rows
    a = (int **)calloc(r, sizeof(int *));
    for (i = 0; i < r; i++) {
        a[i] = (int *)calloc(c, sizeof(int));
    }

    sum = (int *)calloc(r, sizeof(int));

    printf("Enter the elements of the matrix:\n");

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &a[i][j]);
        }
    }

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            sum[i] += a[i][j];
        }
    }

    printf("Row-wise sums:\n");
    for (i = 0; i < r; i++) {
        printf("Row %d: %d\n", i + 1, sum[i]);
    }

    // Free the allocated memory
    for (i = 0; i < r; i++) {
        free(a[i]);
    }
    free(a);
    free(sum);

    return 0;
}
```

----

3)Addition of column wise elements using malloc for single matrix.

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

int main() {
    int r, c;
    int i, j;
    int **a;
    int *sum;

    printf("Enter the number of rows: ");
    scanf("%d", &r);

    printf("Enter the number of columns: ");
    scanf("%d", &c);

    // Allocate memory for the matrix
    a = (int **)malloc(r * sizeof(int *));
    for (i = 0; i < r; i++) {
        a[i] = (int *)malloc(c * sizeof(int));
    }

    sum = (int *)calloc(c, sizeof(int));

    printf("Enter the elements of the matrix:\n");

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &a[i][j]);
        }
    }

    // Perform the addition of column-wise elements
    for (j = 0; j < c; j++) {
        for (i = 0; i < r; i++) {
            sum[j] += a[i][j];
        }
    }

    printf("Column-wise sums:\n");
    for (j = 0; j < c; j++) {
        printf("Column %d: %d\n", j + 1, sum[j]);
    }

    // Free the allocated memory
    for (i = 0; i < r; i++) {
        free(a[i]);
    }
    free(a);
```

```
    free(sum);

    return 0;
}
```

----

4)Addition of all elements using malloc for TWO matrix of same size:

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

int main() {
    int r, c;
    int i, j;
    int **a, **b;
    int **result;

    printf("Enter the number of rows: ");
    scanf("%d", &r);

    printf("Enter the number of columns: ");
    scanf("%d", &c);

    a = (int **)malloc(r * sizeof(int *));
    for (i = 0; i < r; i++) {
        a[i] = (int *)malloc(c * sizeof(int));
    }


    b = (int **)malloc(r * sizeof(int *));
    for (i = 0; i < r; i++) {
        b[i] = (int *)malloc(c * sizeof(int));
    }

    result = (int **)malloc(r * sizeof(int *));
    for (i = 0; i < r; i++) {
        result[i] = (int *)malloc(c * sizeof(int));
    }

    printf("Enter the elements of the first matrix:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &a[i][j]);
        }
    }

    printf("Enter the elements of the second matrix:\n");
```

```
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &b[i][j]);
        }
    }

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }
    }

    printf("Resultant matrix after addition:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    for (i = 0; i < r; i++) {
        free(a[i]);
        free(b[i]);
        free(result[i]);
    }
    free(a);
    free(b);
    free(result);

    return 0;
}
```

----

5) Addition of row wise elements for TWO matrix of same size using malloc:

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

int main() {
    int r, c;
    int i, j;
    int **a, **b;
    int *sumA, *sumB;

    printf("Enter the number of rows: ");
    scanf("%d", &r);

    printf("Enter the number of columns: ");
```

```
scanf("%d", &c);

// Allocate memory for the first matrix
a = (int **)malloc(r * sizeof(int *));
for (i = 0; i < r; i++) {
    a[i] = (int *)malloc(c * sizeof(int));
}

// Allocate memory for the second matrix
b = (int **)malloc(r * sizeof(int *));
for (i = 0; i < r; i++) {
    b[i] = (int *)malloc(c * sizeof(int));
}

// Allocate memory for storing row-wise sums of the matrices
sumA = (int *)malloc(r * sizeof(int));
sumB = (int *)malloc(r * sizeof(int));

printf("Enter the elements of the first matrix:\n");
for (i = 0; i < r; i++) {
    for (j = 0; j < c; j++) {
        scanf("%d", &a[i][j]);
    }
}

printf("Enter the elements of the second matrix:\n");
for (i = 0; i < r; i++) {
    for (j = 0; j < c; j++) {
        scanf("%d", &b[i][j]);
    }
}


for (i = 0; i < r; i++) {
    for (j = 0; j < c; j++) {
        sumA[i] += a[i][j];
        sumB[i] += b[i][j];
    }
}

printf("Row-wise sums for the first matrix:\n");
for (i = 0; i < r; i++) {
    printf("Row %d: %d\n", i + 1, sumA[i]);
}

printf("Row-wise sums for the second matrix:\n");
for (i = 0; i < r; i++) {
    printf("Row %d: %d\n", i + 1, sumB[i]);
}

for (i = 0; i < r; i++) {
    free(a[i]);
    free(b[i]);
}
free(a);
```

```
    free(b);
    free(sumA);
    free(sumB);

    return 0;
}
```

----

6) database of students (name, phone number, address, 10th marks, 12th marks, eligible for engineering or not):

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

struct Student {
    char name[20];
    char phoneNumber[12];
    char address[100];
    float tenthMarks;
    float twelfthMarks;
    int isEligibleForEngineering;
};

int main() {
    int n;

    printf("Enter the number of students: ");
    scanf("%d", &n);

//struct: calls the function, Student: acts as a datatype like int, students:is a variable
    struct Student* s = (struct Student*)malloc(n * sizeof(struct Student));

    // Input student details
    for (int i = 0; i < n; i++) {
        printf("\nEnter details for student %d:\n", i + 1);

        printf("Name: ");
        scanf(" \n%s", s[i].name);

        printf("Phone Number: ");
        scanf(" \n%s", s[i].phoneNumber);

        printf("Address: ");
        scanf(" \n %s", s[i].address);

        printf("10th Marks: ");
        scanf("%f", &(s[i].tenthMarks));
```

```
        printf("12th Marks: ");
        scanf("%f", &(s[i].twelfthMarks));

        printf("Eligible for Engineering (0 - No, 1 - Yes): ");
        scanf("%d", &(s[i].isEligibleForEngineering));
    }

    // Print student details
    printf("\nStudent Details:\n");
    for (int i = 0; i < n; i++) {
        printf("\nStudent %d:\n", i + 1);
        printf("Name: %s\n", s[i].name);
        printf("Phone Number: %s\n", s[i].phoneNumber);
        printf("Address: %s\n", s[i].address);
        printf("10th Marks: %.2f\n", s[i].tenthMarks);
        printf("12th Marks: %.2f\n", s[i].twelfthMarks);
        printf("Eligible for Engineering: %s\n", s[i].isEligibleForEngineering ? "Yes" : "No");
    }

    // Free allocated memory
    free(s);

    return 0;
}
```

----

7) Database of work of ISRO - (name of activity, scientist involved, advantage to society, minimum 10 such activities):

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

struct isro {
    char name[100];
    char scientist[100];
    char advantage[1000];
};

int main() {
    int n = 10; // Number of ISRO activities

    struct isro* activities = (struct isro*)malloc(n * sizeof(struct isro));

    // Input ISRO activity details
    for (int i = 0; i < n; i++) {
        printf("\nEnter details for ISRO activity %d:\n", i + 1);

        printf("Name of Activity: ");
```

```
        scanf(" \n%s", activities[i].name);

        printf("Scientist Involved: ");
        scanf(" \n%s", activities[i].scientist);

        printf("Advantage to Society: ");
        scanf(" \n%s", activities[i].advantage);
    }

    // Print ISRO activity details
    printf("\nISRO Activities:\n");
    for (int i = 0; i < n; i++) {
        printf("\nISRO Activity %d:\n", i + 1);
        printf("Name of Activity: %s\n", activities[i].name);
        printf("Scientist Involved: %s\n", activities[i].scientist);
        printf("Advantage to Society: %s\n", activities[i].advantage);
    }

    // Free allocated memory
    free(activities);

    return 0;
}
```

----

8) Addition of two Single Variable Polynomials, using the representation where the exponents are implicit:

```
#include <stdio.h>

struct poly
{
    int coeff;
    int expo;
};

int readpoly(struct poly p[10])
{
    int t;
    printf("Enter the total number of terms: ");
    scanf("%d", &t);

    printf("Enter the coefficients and exponents:\n");
    for (int i = 0; i < t; i++)
    {
        printf("Term %d: ", i + 1);
        scanf("%d%d", &(p[i].coeff), &(p[i].expo));
    }
```

```
    return t;
}

int addpoly(struct poly p1[10], struct poly p2[10], int t1, int t2, struct poly p3[10])
{
    int i = 0, j = 0, k = 0;

    while (i < t1 && j < t2)
    {
        if (p1[i].expo == p2[j].expo)
        {
            p3[k].coeff = p1[i].coeff + p2[j].coeff;
            p3[k].expo = p1[i].expo;
            i++;
            j++;
            k++;
        }
        else if (p1[i].expo > p2[j].expo)
        {
            p3[k].coeff = p1[i].coeff;
            p3[k].expo = p1[i].expo;
            i++;
            k++;
        }
        else
        {
            p3[k].coeff = p2[j].coeff;
            p3[k].expo = p2[j].expo;
            j++;
            k++;
        }
    }

    while (i < t1)
    {
        p3[k].coeff = p1[i].coeff;
        p3[k].expo = p1[i].expo;
        i++;
        k++;
    }

    while (j < t2)
    {
        p3[k].coeff = p2[j].coeff;
        p3[k].expo = p2[j].expo;
        j++;
        k++;
    }

    return k;
}

void displaypoly(struct poly p[10], int terms)
{
```

```
    for (int i = 0; i < terms; i++)
    {
        printf("%dx^%d ", p[i].coeff, p[i].expo);
        if (i < terms - 1)
            printf("+ ");
    }
    printf("\n");
}

int main()
{
    struct poly p1[10], p2[10], p3[10];
    int t1, t2, t3;

    printf("Enter the first polynomial:\n");
    t1 = readpoly(p1);
    printf("First Polynomial: ");
    displaypoly(p1, t1);

    printf("\nEnter the second polynomial:\n");
    t2 = readpoly(p2);
    printf("Second Polynomial: ");
    displaypoly(p2, t2);

    t3 = addpoly(p1, p2, t1, t2, p3);
    printf("\nResult: ");
    displaypoly(p3, t3);

    return 0;
}
```

----

9)Multiplication of two Single Variable Polynomials, using the representation where the exponents are implicit

```
#include <stdio.h>

struct poly
{
    int coeff;
    int expo;
};

int readpoly(struct poly p[10])
{
    int t;
```

```c
    printf("Enter the total number of terms: ");
    scanf("%d", &t);

    printf("Enter the coefficients and exponents:\n");
    for (int i = 0; i < t; i++)
    {
        printf("Term %d: ", i + 1);
        scanf("%d%d", &(p[i].coeff), &(p[i].expo));
    }

    return t;
}

int multiplypoly(struct poly p1[10], struct poly p2[10], int t1, int t2, struct poly p3[10])
{
    int k = 0;

    for (int i = 0; i < t1; i++)
    {
        for (int j = 0; j < t2; j++)
        {
            p3[k].coeff = p1[i].coeff * p2[j].coeff;
            p3[k].expo = p1[i].expo + p2[j].expo;
            k++;
        }
    }

    return k;
}

void displaypoly(struct poly p[10], int terms)
{
    for (int i = 0; i < terms; i++)
    {
        printf("%dx^%d ", p[i].coeff, p[i].expo);
        if (i < terms - 1)
            printf("+ ");
    }
    printf("\n");
}

int main()
{
    struct poly p1[10], p2[10], p3[100];
    int t1, t2, t3;

    printf("Enter the first polynomial:\n");
    t1 = readpoly(p1);
    printf("First Polynomial: ");
    displaypoly(p1, t1);

    printf("\nEnter the second polynomial:\n");
    t2 = readpoly(p2);
    printf("Second Polynomial: ");
    displaypoly(p2, t2);
```

```
    t3 = multiplypoly(p1, p2, t1, t2, p3);
    printf("\nResult: ");
    displaypoly(p3, t3);

    return 0;
}
```

----

10) Evaluation of Single Variable Polynomial, using the representation where the exponents are implicit

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

int main()
{
    int a[20];
    int n, x, sum = 0;

    printf("Enter the Value of X: ");
    scanf("%d", &x);
    printf("Enter the Degree of the polynomial: ");
    scanf("%d", &n);
    printf("Enter the Coefficients of the polynomial:\n");
    for (int i = 0; i <= n; i++)
    {
        printf("Coefficient of x^%d: ", i);
        scanf("%d", &a[i]);
    }

    for (int i = 0; i <= n; i++)
    {
        sum += a[i] * pow(x, i);
    }

    printf("Evaluation of the polynomial: %d\n", sum);
    return 0;
}
```

----

11)Addition of two Multivariable Polynomials with two variables, using the representation where the expon
ents are implicit

```
#include <stdio.h>

struct Term {
    int coeff;
    int vars[2];
};

void readPoly(struct Term poly[], int *numTerms) {
    printf("Enter the total number of terms: ");
    scanf("%d", numTerms);

    printf("Enter the coefficients and exponents (var1, var2) of each term:\n");
    for (int i = 0; i < *numTerms; i++) {
        printf("Term %d:\n", i + 1);
        printf("Coefficient: ");
        scanf("%d", &poly[i].coeff);
        printf("Exponent (var1): ");
        scanf("%d", &poly[i].vars[0]);
        printf("Exponent (var2): ");
        scanf("%d", &poly[i].vars[1]);
    }
}

void addPoly(const struct Term poly1[], int numTerms1, const struct Term poly2[], int numTerms2, struct
Term result[], int *numTermsResult) {
    int i = 0, j = 0, k = 0;

    while (i < numTerms1 && j < numTerms2) {
        if (poly1[i].vars[0] > poly2[j].vars[0] || (poly1[i].vars[0] == poly2[j].vars[0] && poly1[i].vars[1] > poly2[j].v
ars[1])) {
            result[k++] = poly1[i++];
        } else if (poly1[i].vars[0] < poly2[j].vars[0] || (poly1[i].vars[0] == poly2[j].vars[0] && poly1[i].vars[1] < po
ly2[j].vars[1])) {
            result[k++] = poly2[j++];
        } else {
            result[k].coeff = poly1[i].coeff + poly2[j].coeff;
            result[k].vars[0] = poly1[i].vars[0];
            result[k].vars[1] = poly1[i].vars[1];
            i++;
            j++;
            k++;
        }
    }

    while (i < numTerms1) {
        result[k++] = poly1[i++];
    }

    while (j < numTerms2) {
        result[k++] = poly2[j++];
    }
```

```
    *numTermsResult = k;
}

void displayPoly(const struct Term poly[], int numTerms) {
    for (int i = 0; i < numTerms; i++) {
        printf("%dx^%dy^%d ", poly[i].coeff, poly[i].vars[0], poly[i].vars[1]);
        if (i < numTerms - 1) {
            printf("+ ");
        }
    }
    printf("\n");
}

int main() {
    struct Term poly1[10], poly2[10], result[20];
    int numTerms1, numTerms2, numTermsResult;

    printf("Polynomial 1:\n");
    readPoly(poly1, &numTerms1);

    printf("\nPolynomial 2:\n");
    readPoly(poly2, &numTerms2);

    addPoly(poly1, numTerms1, poly2, numTerms2, result, &numTermsResult);

    printf("\nResult of addition:\n");
    displayPoly(result, numTermsResult);

    return 0;
}
```

----

12) Create magic square matrix of order 3×3 :

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

int main(){
    int i, j, n, **a, sumd1=0, sumd2=0, sumr=0, sumc=0, f=0;

    printf("Enter Number of order of Square Matrix: ");
    scanf("%d", &n);

    a=(int**)malloc(n* sizeof(int*));
    for(i=0;i<n;i++){
        a[i]=(int*)malloc(n* sizeof(int));
```

```
}

printf("Elements: ");
for(i=0;i<n;i++){
    for(j=0; j<n; j++){
        scanf("%d", &a[i][j]);
    }
}

printf("Matrix:\n ");
for(i=0;i<n;i++){
    for(j=0; j<n; j++){
        printf("%d \t", a[i][j]);
    }
    printf("\n");
}



for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        if(i==j){
            sumd1=sumd1+a[i][j];
        }
        if(i+j==n-1){
            sumd2=sumd2+a[i][j];
        }
    }
}
if(sumd1!=sumd2)
{
    f=1;
}
else
{
    for(i=0;i<n;i++){
        sumc=0;
        sumr=0;
        for(j=0;j<n;j++){
            sumr=sumr+a[i][j];
            sumc=sumc+a[i][j];
        }
        if(sumc!=sumd1){
            f=1;
        }
        else if(sumr!=sumd1){
            f=1;
        }
        else{
            f=0;
        }
    }
}
if(f==0){
    printf("Matrix is Magic Square Matrix ");
```

```
    }
    else{
    printf("Matrix is Not Magic Square Matrix");
}
    printf("\nSum of Diagonal 1: %d", sumd1);
    printf("\nSum of Diagonal 2: %d", sumd2);
    printf("\nSum of Rows : %d", sumr);
    printf("\nSum of Cols : %d", sumc);



    for(i=0;i<n;i++){
        free(a[i]);
    }

    free(a);

    return 0;
}
```

----

13) create a square matrix where only the rows and the column sum is up to the magic constant.(semi magic square).

```
#include <stdio.h>
int main() {
    int N;

    printf("Enter the order of the semi-magic square: ");
    scanf("%d", &N);

    int semiMagicSquare[N][N];
    int magicSum = (N * (N * N + 1)) / 2;

    int rowSum = 0, colSum = 0;

    // Generate semi-magic square
    for (int i = 0; i < N; i++) {
        rowSum = 0;
        colSum = 0;

        for (int j = 0; j < N; j++) {
            semiMagicSquare[i][j] = magicSum - rowSum; // Fill each element to maintain row sum
            rowSum += semiMagicSquare[i][j]; // Update row sum
            colSum += semiMagicSquare[j][i]; // Update column sum
        }
```

```
        semiMagicSquare[i][i] = magicSum - colSum; // Adjust diagonal element to maintain column sum
    }

    // Print semi-magic square
    printf("Semi-Magic Square:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", semiMagicSquare[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

----

14) Checking of Sparse Matrix:

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

int main(){
    int i, j, r, c, **a, count=0;

    printf("Enter Number of Rows: ");
    scanf("%d", &r);
    printf("Enter Number of Cols: ");
    scanf("%d", &c);

    a=(int**)malloc(r* sizeof(int*));
    for(i=0;i<r;i++){
        a[i]=(int*)malloc(c* sizeof(int));


    }

    printf("Elements: ");
    for(i=0;i<r;i++){
        for(j=0; j<c; j++){
            scanf("%d", &a[i][j]);
        }
    }

    printf("Matrix:\n ");
    for(i=0;i<r;i++){
        for(j=0; j<c; j++){
```

```
            printf("%d \t", a[i][j]);
        }
        printf("\n");
    }

    for(i=0;i<r;i++){
        for(j=0;j<c;j++){
            if(a[i][j]==0){
                count++;
            }
        }
    }
    if((r*c)/2<count)
        printf("Given Matrix is a Sparse Matrix ");
    else
    printf("Given Matrix is not a Sparse Matrix ");

    for(i=0;i<r;i++){
         free(a[i]);
    }

    free(a);

    return 0;
}
```

----

15) Sparse Transpose:

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

int main() {
    int i, j, r, c, **a, count = 0, **b;

    printf("Enter Number of Rows: ");
    scanf("%d", &r);
    printf("Enter Number of Cols: ");
    scanf("%d", &c);

    a = (int**)malloc(r * sizeof(int*));
    for (i = 0; i < r; i++) {
        a[i] = (int*)malloc(c * sizeof(int));
    }

    b = (int**)malloc(r * sizeof(int*));
    for (i = 0; i < r; i++) {
        b[i] = (int*)malloc(c * sizeof(int));
```

```
    }

    printf("Elements: ");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            scanf("%d", &a[i][j]);
        }
    }

    printf("Matrix:\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            printf("%d \t", a[i][j]);
        }
        printf("\n");
    }

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            if (a[i][j] == 0) {
                count++;
            }
        }
    }

    if ((r * c) / 2 < count) {
        printf("Given Matrix is a Sparse Matrix\n");
    } else {
        printf("Given Matrix is not a Sparse Matrix\n");
    }

    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            b[i][j] = a[j][i];
        }
    }

    printf("Transpose:\n");
    for (i = 0; i < c; i++) {
        for (j = 0; j < r; j++) {
            printf("%d \t", b[i][j]);
        }
        printf("\n");
    }

    for (i = 0; i < r; i++) {
        free(a[i]);
        free(b[i]);
    }

    free(a);
    free(b);

    return 0;
}
```

----

16) Perform Selection Sort on numeric data entered by the user:

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

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int *a = (int *)malloc(n * sizeof(int));

    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }

    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min])
                min = j;
        }

        if (min != i) {
            int temp = a[i];
            a[i] = a[min];
            a[min] = temp;
        }
    }

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }

    free(a);

    return 0;
}
```

----

17) Perform Bubble Sort on the Numeric data entered by the user:

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n;

    printf("Enter the Number of Elements: ");
    scanf("%d", &n);

    int *a = (int*) malloc (n * sizeof(int));
    printf("Enter the Numbers to be Sorted: \n");
    for(int i = 0; i < n; i++){
        scanf("%d", &a[i]);

    }

    for(int i = 0; i < n; i++){
        for(int j = 0; j < (n-1); j++){
            if(a[j] > a[j+1]){
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
    printf("Sorted Elements: \n");
    for(int i = 0; i < n; i++){
        printf("%d ", a[i]);
    }

    free(a);
    return 0;
}
```

----

18) Perform Insertion Sort on the Numeric data entered by the user:

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int n;
```

```
    printf("Enter The Number of Arrays: ");
    scanf("%d", &n);

    int *a = (int*)malloc(n* sizeof(int));
    printf("Enter the Elements: ");
    for(int i=0; i<n; i++)
    {
        scanf("%d", &a[i]);
    }

    for(int i=1; i<n; i++){
        int key= a[i];
        int j=i-1;

        while(j>=0 && a[j]>key){
            a[j+1]=a[j];
            j = j-1;
        }
        a[j+1]=key;
    }

    printf("Sorted Elements:");
    for(int i=0;i<n;i++){
        printf("%d", a[i]);
    }
    free(a);
}
```

----

19) Perform Merge Sort on the Numeric data entered by the user:

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

void merge(int arr[], int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int size = rightEnd - leftStart + 1;
    int temp[size];

    int i = leftStart, j = rightStart, k = 0;

    while (i <= leftEnd && j <= rightEnd) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
```

```c
    }

    while (i <= leftEnd)
        temp[k++] = arr[i++];

    while (j <= rightEnd)
        temp[k++] = arr[j++];

    for (i = leftStart, k = 0; i <= rightEnd; i++, k++)
        arr[i] = temp[k];
}

void mergeSort(int arr[], int start, int end) {
    if (start >= end)
        return;

    int mid = (start + end) / 2;
    mergeSort(arr, start, mid);
    mergeSort(arr, mid + 1, end);
    merge(arr, start, mid, mid + 1, end);
}

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

----

20) Perform Quick Sort on the Numeric data entered by the user:

```c
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to partition the array and return the pivot index
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;
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
```

```c
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

----

21) Perform Radix Sort on the Numeric data entered by the user:

```c
#include <stdio.h>

int findMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

void countingSort(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};

    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

// Function to perform Radix Sort
void radixSort(int arr[], int n) {
```

```
    int max = findMax(arr, n);

    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    radixSort(arr, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

----

22) Perform Linear search on the Numeric data entered by the user.

```
#include <stdio.h>

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements:\n");
```

```c
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int key;
    printf("Enter the key to search: ");
    scanf("%d", &key);

    int found = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("Key found at index %d\n", i);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Key not found\n");
    }

    return 0;
}
```

----

23) Perform Binary search on the Numeric data entered by the user.

```c
#include <stdio.h>
int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements in sorted order:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int key;
    printf("Enter the key to search: ");
    scanf("%d", &key);
```

```
    int index = binarySearch(arr, 0, n - 1, key);

    if (index != -1) {
        printf("Key found at index %d\n", index);
    } else {
        printf("Key not found\n");
    }

    return 0;
}
int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid;  // Key found, return the index
        } else if (arr[mid] < key) {
            low = mid + 1;  // Key is in the right half
        } else {
            high = mid - 1;  // Key is in the left half
        }
    }

    return -1;  // Key not found
}
```

----

24) Perform Fibonacci search on the Numeric data entered by the user:

```
#include <stdio.h>

// Function to perform Fibonacci search
int fibonacciSearch(int arr[], int n, int key) {
    int fib2 = 0;  // (k-2)th Fibonacci number
    int fib1 = 1;  // (k-1)th Fibonacci number
    int fib = fib1 + fib2;  // kth Fibonacci number

    // Find the smallest Fibonacci number greater than or equal to the array size
    while (fib < n) {
        fib2 = fib1;
        fib1 = fib;
        fib = fib1 + fib2;
```

```c
    }

    int offset = -1;  // Offset from the first element

    while (fib > 1) {
        int i = (offset + fib2) < (n - 1) ? (offset + fib2) : (n - 1);

        if (arr[i] == key) {
            return i;  // Key found, return the index
        } else if (arr[i] < key) {
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;
            offset = i;
        } else {
            fib = fib2;
            fib1 = fib1 - fib2;
            fib2 = fib - fib1;
        }
    }

    if (fib1 == 1 && arr[offset + 1] == key) {
        return offset + 1;  // Key found, return the index
    }

    return -1;  // Key not found
}

int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements in sorted order:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int key;
    printf("Enter the key to search: ");
    scanf("%d", &key);

    int index = fibonacciSearch(arr, n, key);

    if (index != -1) {
        printf("Key found at index %d\n", index);
    } else {
        printf("Key not found\n");
    }

    return 0;
}
```

---

25) Perform Interpolation search on the Numeric data entered by the user:

```
#include <stdio.h>
int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements in sorted order:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int key;
    printf("Enter the key to search: ");
    scanf("%d", &key);

    int index = interpolationSearch(arr, n, key);

    if (index != -1) {
        printf("Key found at index %d\n", index);
    } else {
        printf("Key not found\n");
    }

    return 0;
}
int interpolationSearch(int arr[], int n, int key) {
    int low = 0;            // Lowest index of the search range
    int high = n - 1;       // Highest index of the search range

    while (low <= high && key >= arr[low] && key <= arr[high]) {
        if (low == high) {
            if (arr[low] == key) {
                return low;  // Key found, return the index
            }
            return -1;  // Key not found
        }

        // Perform interpolation formula to calculate the mid index
```

```
        int pos = low + (((double)(high - low) / (arr[high] - arr[low])) * (key - arr[low]));

        if (arr[pos] == key) {
            return pos;  // Key found, return the index
        }

        if (arr[pos] < key) {
            low = pos + 1;  // Search in the right half
        } else {
            high = pos - 1;  // Search in the left half
        }
    }

    return -1;  // Key not found
}
```

----

25)) Perform Exponential search on the Numeric data entered by the user.

```
#include <stdio.h>
int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements in sorted order:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int key;
    printf("Enter the key to search: ");
    scanf("%d", &key);

    int index = exponentialSearch(arr, n, key);

    if (index != -1) {
        printf("Key found at index %d\n", index);
    } else {
        printf("Key not found\n");
    }
```

```
    return 0;
}

int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid;  // Key found, return the index
        }

        if (arr[mid] < key) {
            low = mid + 1;  // Search in the right half
        } else {
            high = mid - 1;  // Search in the left half
        }
    }

    return -1;  // Key not found
}
int exponentialSearch(int arr[], int n, int key) {
    if (arr[0] == key) {
        return 0;  // Key found at the first element
    }

    int i = 1;
    while (i < n && arr[i] <= key) {
        i *= 2;  // Double the position for the next step
    }

    // Call binary search for the range from i/2 to min(i, n-1)
    return binarySearch(arr, i / 2, (i < n) ? i : n - 1, key);
}
```

----

26) Perform Ternary search on the Numeric data entered by the user:

```
#include <stdio.h>
int main() {
    int n;

    printf("Enter the number of elements: ");
    scanf("%d", &n);
```

```c
    int arr[n];

    printf("Enter the elements in sorted order:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int key;
    printf("Enter the key to search: ");
    scanf("%d", &key);

    int index = ternarySearch(arr, 0, n - 1, key);

    if (index != -1) {
        printf("Key found at index %d\n", index);
    } else {
        printf("Key not found\n");
    }

    return 0;
}
int ternarySearch(int arr[], int low, int high, int key) {
    if (low <= high) {
        int mid1 = low + (high - low) / 3;
        int mid2 = high - (high - low) / 3;

        if (arr[mid1] == key) {
            return mid1;  // Key found at mid1
        }

        if (arr[mid2] == key) {
            return mid2;  // Key found at mid2
        }

        if (key < arr[mid1]) {
            // Key lies in the left third
            return ternarySearch(arr, low, mid1 - 1, key);
        } else if (key > arr[mid2]) {
            // Key lies in the right third
            return ternarySearch(arr, mid2 + 1, high, key);
        } else {
            // Key lies in the middle third
            return ternarySearch(arr, mid1 + 1, mid2 - 1, key);
        }
    }

    return -1;  // Key not found
}
```

----

27) Write a C program to demonstrate hashing. Also calculate the space and time complexity.

```
#include <stdio.h>
#define SIZE 10
int hashFunction(int key) {
return key % SIZE;
}
int linearProbe(int hashArray[], int key) {
int index = hashFunction(key);
int i = 0;
while (hashArray[(index + i) % SIZE] != -1) {
i++;
}
return (index + i) % SIZE;
}
void insert(int hashArray[], int key) {
int index = hashFunction(key);
if (hashArray[index] == -1) {
hashArray[index] = key;
} else {
int newIndex = linearProbe(hashArray, key);
hashArray[newIndex] = key;
}
}
void display(int hashArray[]) {
for (int i = 0; i < SIZE; i++) {
if (hashArray[i] != -1) {
printf("Hash[%d] = %d\n", i, hashArray[i]);
}
}
}
int main() {
int hashArray[SIZE];
for (int i = 0; i < SIZE; i++) {
hashArray[i] = -1;
}
// Inserting elements into the hash table
insert(hashArray, 12);
insert(hashArray, 25);
insert(hashArray, 35);
insert(hashArray, 45);
insert(hashArray, 55);
// Displaying the hash table
display(hashArray);
return 0;
}
```

----

28) Write a C program to implement hash tables.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
typedef struct Node {
int key;
int value;
struct Node* next;
} Node;
typedef struct {
Node* head;
} LinkedList;
typedef struct {
LinkedList* array;
} HashTable;
HashTable* createHashTable() {
HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
hashTable->array = (LinkedList*)calloc(SIZE, sizeof(LinkedList));
return hashTable;
}
int hashFunction(int key) {
return key % SIZE;
}
Node* createNode(int key, int value) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->key = key;
newNode->value = value;
newNode->next = NULL;
return newNode;
}
void insert(HashTable* hashTable, int key, int value) {
int index = hashFunction(key);
Node* newNode = createNode(key, value);
if (hashTable->array[index].head == NULL) {
hashTable->array[index].head = newNode;
} else {
Node* current = hashTable->array[index].head;
while (current->next != NULL) {
current = current->next;
}
current->next = newNode;
}
}
int search(HashTable* hashTable, int key) {
int index = hashFunction(key);
Node* current = hashTable->array[index].head;
```

```
while (current != NULL) {
if (current->key == key) {
return current->value;
}
current = current->next;
}
return -1; // Key not found
}
void display(HashTable* hashTable) {
for (int i = 0; i < SIZE; i++) {
printf("Index %d: ", i);
Node* current = hashTable->array[i].head;
while (current != NULL) {
printf("(%d, %d) ", current->key, current->value);
current = current->next;
}
printf("\n");
}
}
void freeHashTable(HashTable* hashTable) {
for (int i = 0; i < SIZE; i++) {
Node* current = hashTable->array[i].head;
while (current != NULL) {
Node* temp = current;
current = current->next;
free(temp);
}
}
free(hashTable->array);
free(hashTable);
}
int main() {
HashTable* hashTable = createHashTable();
// Inserting elements into the hash table
insert(hashTable, 12, 45);
insert(hashTable, 25, 62);
insert(hashTable, 35, 78);
insert(hashTable, 45, 91);
insert(hashTable, 55, 34);
// Displaying the hash table
display(hashTable);
// Searching for a key in the hash table
int searchKey = 35;
int searchResult = search(hashTable, searchKey);
if (searchResult != -1) {
printf("Value for key %d: %d\n", searchKey, searchResult);
} else {
printf("Key %d not found in the hash table.\n", searchKey);
}
// Freeing the memory allocated for the hash table
freeHashTable(hashTable);
return 0;
}
```

----

29) Write a C program to generate a hash value using division hash function.

```
#include <stdio.h>
#define TABLE_SIZE 10
int hashFunction(int key) {
return key % TABLE_SIZE;
}
int main() {
int key;
printf("Enter a key: ");
scanf("%d", &key);
int hashValue = hashFunction(key);
printf("Hash value for key %d: %d\n", key, hashValue);
return 0;
}
```

----

30) Write a C program to generate a hash value using mid square hash function.

```
#include <stdio.h>
#include <math.h>
#define TABLE_SIZE 10
int hashFunction(int key) {
int square = key * key;
int numDigits = log10(square) + 1;
int numToRemove = (numDigits - TABLE_SIZE) / 2;
int quotient = square;
for (int i = 0; i < numToRemove; i++) {
quotient /= 10;
}
int hashValue = quotient % TABLE_SIZE;
return hashValue;
}
int main() {
int key;
printf("Enter a key: ");
scanf("%d", &key);
int hashValue = hashFunction(key);
printf("Hash value for key %d: %d\n", key, hashValue);
```

```
return 0;
}
```

----

31) Write a C program to generate a hash value using folding hash function.

```
#include <stdio.h>
#define TABLE_SIZE 10
int hashFunction(int key) {
int hashValue = 0;
while (key > 0) {
int lastTwoDigits = key % 100;
hashValue += lastTwoDigits;
key /= 100;
}
hashValue %= TABLE_SIZE;
return hashValue;
}
int main() {
int key;
printf("Enter a key: ");
scanf("%d", &key);
int hashValue = hashFunction(key);
printf("Hash value for key %d: %d\n", key, hashValue);
return 0;
}
```

----

32) Write a C program to generate a hash value using multiplication hash function.

```
#include <stdio.h>
#define TABLE_SIZE 10
int hashFunction(int key) {
int hashValue = 0;
while (key > 0) {
int lastTwoDigits = key % 100;
hashValue += lastTwoDigits;
key /= 100;
```

```
}
hashValue %= TABLE_SIZE;
return hashValue;
}
int main() {
int key;
printf("Enter a key: ");
scanf("%d", &key);
int hashValue = hashFunction(key);
printf("Hash value for key %d: %d\n", key, hashValue);
return 0;
}
```

----

33) Write a C program to implement hashing using linear probing as the collision resolution strategy.

```
#include <stdio.h>
#define TABLE_SIZE 10
int hashFunction(int key) {
return key % TABLE_SIZE;
}
void insert(int hashTable[], int key) {
int index = hashFunction(key);
// Linear probing until an empty slot is found
while (hashTable[index] != -1) {
index = (index + 1) % TABLE_SIZE;
}
hashTable[index] = key;
}
int search(int hashTable[], int key) {
int index = hashFunction(key);
// Linear probing until key is found or an empty slot is encountered
while (hashTable[index] != -1) {
if (hashTable[index] == key) {
return index; // Key found
}
index = (index + 1) % TABLE_SIZE;
}
return -1;
}
void display(int hashTable[]) {
for (int i = 0; i < TABLE_SIZE; i++) {
printf("Index %d: ", i);
if (hashTable[i] != -1) {
```

```
printf("%d", hashTable[i]);
}
printf("\n");
}
}
int main() {
int hashTable[TABLE_SIZE];
for (int i = 0; i < TABLE_SIZE; i++) {
hashTable[i] = -1;
}
insert(hashTable, 12);
insert(hashTable, 25);
insert(hashTable, 35);
insert(hashTable, 45);
insert(hashTable, 55);
display(hashTable);
int searchKey = 35;
int searchIndex = search(hashTable, searchKey);
if (searchIndex != -1) {
printf("Key %d found at index %d\n", searchKey, searchIndex);
} else {
printf("Key %d not found in the hash table.\n", searchKey);
}
return 0;
}
```

----

34)Write a C program to implement hashing using chaining with replacement as the collision resolution strategy.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
typedef struct Node {
int key;
int value;
struct Node* next;
} Node;
typedef struct {
Node* head;
} LinkedList;
typedef struct {
LinkedList* array;
} HashTable;
HashTable* createHashTable() {
HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
hashTable->array = (LinkedList*)calloc(TABLE_SIZE, sizeof(LinkedList));
```

```
return hashTable;
}
int hashFunction(int key) {
return key % TABLE_SIZE;
}
Node* createNode(int key, int value) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->key = key;
newNode->value = value;
newNode->next = NULL;
return newNode;
}
void insert(HashTable* hashTable, int key, int value) {
int index = hashFunction(key);
if (hashTable->array[index].head == NULL) {
// Empty slot, insert node as the head of the linked list
hashTable->array[index].head = createNode(key, value);
} else {
// Collision, replace the head node with the new node
Node* newNode = createNode(key, value);
newNode->next = hashTable->array[index].head;
hashTable->array[index].head = newNode;
}
}
int search(HashTable* hashTable, int key) {
int index = hashFunction(key);
Node* current = hashTable->array[index].head;
while (current != NULL) {
if (current->key == key) {
return current->value;
}
current = current->next;
}
return -1; // Key not found
}
void display(HashTable* hashTable) {
for (int i = 0; i < TABLE_SIZE; i++) {
printf("Index %d: ", i);
Node* current = hashTable->array[i].head;
while (current != NULL) {
printf("(%d, %d) ", current->key, current->value);
current = current->next;
}
printf("\n");
}
}
void freeHashTable(HashTable* hashTable) {
for (int i = 0; i < TABLE_SIZE; i++) {
Node* current = hashTable->array[i].head;
while (current != NULL) {
Node* temp = current;
current = current->next;
free(temp);
}
}
```

```
free(hashTable->array);
free(hashTable);
}
int main() {
HashTable* hashTable = createHashTable();
// Inserting elements into the hash table
insert(hashTable, 12, 45);
insert(hashTable, 25, 62);
insert(hashTable, 35, 78);
insert(hashTable, 45, 91);
insert(hashTable, 55, 34);
// Displaying the hash table
display(hashTable);
// Searching for a key in the hash table
int searchKey = 35;
int searchResult = search(hashTable, searchKey);
if (searchResult != -1) {
printf("Value for key %d: %d\n", searchKey, searchResult);
} else {
printf("Key %d not found in the hash table.\n", searchKey);
}
// Freeing the memory allocated for the hash table
freeHashTable(hashTable);
return 0;
}
```

----

35) Write a C program to implement hashing using chaining without replacement as the collision resolutio n strategy.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
typedef struct Node {
int key;
int value;
struct Node* next;
} Node;
typedef struct {
Node* head;
} LinkedList;
typedef struct {
LinkedList* array;
} HashTable;
```

```
HashTable* createHashTable() {
HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
hashTable->array = (LinkedList*)calloc(TABLE_SIZE, sizeof(LinkedList));
return hashTable;
}
int hashFunction(int key) {
return key % TABLE_SIZE;
}
Node* createNode(int key, int value) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->key = key;
newNode->value = value;
newNode->next = NULL;
return newNode;
}
void insert(HashTable* hashTable, int key, int value) {
int index = hashFunction(key);
if (hashTable->array[index].head == NULL) {
// Empty slot, insert node as the head of the linked list
hashTable->array[index].head = createNode(key, value);
} else {
// Collision, append the new node at the end of the linked list
Node* current = hashTable->array[index].head;
while (current->next != NULL) {
current = current->next;
}
current->next = createNode(key, value);
}
}
int search(HashTable* hashTable, int key) {
int index = hashFunction(key);
Node* current = hashTable->array[index].head;
while (current != NULL) {
if (current->key == key) {
return current->value;
}
current = current->next;
}
return -1; // Key not found
}
void display(HashTable* hashTable) {
for (int i = 0; i < TABLE_SIZE; i++) {
printf("Index %d: ", i);
Node* current = hashTable->array[i].head;
while (current != NULL) {
printf("(%d, %d) ", current->key, current->value);
current = current->next;
}
printf("\n");
}
}
void freeHashTable(HashTable* hashTable) {
for (int i = 0; i < TABLE_SIZE; i++) {
Node* current = hashTable->array[i].head;
while (current != NULL) {
```

```
Node* temp = current;
current = current->next;
free(temp);
}
}
free(hashTable->array);
free(hashTable);
}
int main() {
HashTable* hashTable = createHashTable();
// Inserting elements into the hash table
insert(hashTable, 12, 45);
insert(hashTable, 25, 62);
insert(hashTable, 35, 78);
insert(hashTable, 45, 91);
insert(hashTable, 55, 34);
// Displaying the hash table
display(hashTable);
// Searching for a key in the hash table
int searchKey = 35;
```

----

36) Write a C program to implement closed hashing.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
typedef struct {
int key;
int value;
} Entry;
typedef struct {
Entry* table;
int size;
} HashTable;

HashTable* createHashTable() {
HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
hashTable->table = (Entry*)calloc(TABLE_SIZE, sizeof(Entry));
hashTable->size = TABLE_SIZE;
return hashTable;

}
int hashFunction(int key) {
```

```
return key % TABLE_SIZE;
}

int probe(int index, int attempt, int size)
{
return (index + attempt) % size;
}

void insert(HashTable* hashTable, int key, int value) {
int index = hashFunction(key);
int attempt = 0;
while (hashTable->table[index].key != -1) {
attempt++;
index = probe(index, attempt, hashTable->size);
}
hashTable->table[index].key = key;
hashTable->table[index].value = value;
}
int search(HashTable* hashTable, int key) {
int index = hashFunction(key);
int attempt = 0;
while (hashTable->table[index].key != -1) {
if (hashTable->table[index].key == key) {
return hashTable->table[index].value;
}
attempt++;
index = probe(index, attempt, hashTable->size);
}
return -1; // Key not found
}
void display(HashTable* hashTable) {
for (int i = 0; i < hashTable->size; i++) {
if (hashTable->table[i].key != -1) {
printf("Index %d: (%d, %d)\n", i, hashTable->table[i].key, hashTable->table[i].value);
}
}
}
void freeHashTable(HashTable* hashTable) {
free(hashTable->table);
free(hashTable);
}
int main() {
HashTable* hashTable = createHashTable();


insert(hashTable, 12, 45);
insert(hashTable, 25, 62);
insert(hashTable, 35, 78);
insert(hashTable, 45, 91);
insert(hashTable, 55, 34);

display(hashTable);

int searchKey = 35;
int searchResult = search(hashTable, searchKey);
```

```
if (searchResult != -1) {
printf("Value for key %d: %d\n", searchKey, searchResult);
} else {
printf("Key %d not found in the hash table.\n", searchKey);
}

freeHashTable(hashTable);
return 0;
}
```

----

37) Write a C Program to implement a Hash Table with Linear Probing.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
typedef struct {
int key;
 int value;
} Entry;
typedef struct
{
 Entry* table;
 int size;
} HashTable;
HashTable* createHashTable()
{
 HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
hashTable->table = (Entry*)calloc(TABLE_SIZE, sizeof(Entry));
hashTable->size = TABLE_SIZE;
return hashTable;
}
int hashFunction(int key) {
return key % TABLE_SIZE;
}
int probe(int index, int attempt, int size)
{
return (index + attempt) % size;
}

void insert(HashTable* hashTable, int key, int value)
{ int index = hashFunction(key);
int attempt = 0;
while (hashTable->table[index].key != 0) {
attempt++; index = probe(index, attempt, hashTable->size);
```

```
}
hashTable->table[index].key = key;
hashTable->table[index].value = value;
}
int search(HashTable* hashTable, int key) {
int index = hashFunction(key);
int attempt = 0;
while (hashTable->table[index].key != 0) {
 if (hashTable->table[index].key == key) {
return hashTable->table[index].value;
} attempt++;
index = probe(index, attempt, hashTable->size);
}
return -1;
void display(HashTable* hashTable) {
for (int i = 0; i < hashTable->size; i++) {
if (hashTable->table[i].key != 0)
{ printf("Index %d: (%d, %d)\n", i, hashTable->table[i].key, hashTable->table[i].value);
 }
    }
}
void freeHashTable(HashTable* hashTable) {
free(hashTable->table);
free(hashTable);
}
int main() {
HashTable* hashTable = createHashTable();

insert(hashTable, 12, 45);
insert(hashTable, 25, 62);
insert(hashTable, 35, 78);
insert(hashTable, 45, 91);
insert(hashTable, 55, 34);
display(hashTable);
int searchKey = 35;
 int searchResult = search(hashTable, searchKey);
if (searchResult != -1) {
printf("Value for key %d: %d\n", searchKey, searchResult);
}
else {
 printf("Key %d not found in the hash table.\n", searchKey);
}
freeHashTable(hashTable);
return 0;
}
```

----

38) Write a program to insert a value in linear probing.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
typedef struct {
int key;
int value;
} Entry;
void insert(int table[], int key, int value)
{
int index = key % TABLE_SIZE;
while (table[index] != 0) {
 index = (index + 1) % TABLE_SIZE;
}
table[index] = value;
}
void display(int table[]) {
for (int i = 0; i < TABLE_SIZE; i++) {
if (table[i] != 0)
{
printf("Index %d: %d\n", i, table[i]);
 }
    }
}
int main() {
int hashTable[TABLE_SIZE] = {0};

insert(hashTable, 12, 45);
insert(hashTable, 25, 62);
insert(hashTable, 35, 78);
insert(hashTable, 45, 91);
insert(hashTable, 55, 34);
display(hashTable);
return 0;
 }
```

----

39) Write a program to search a value in linear probing;

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 10
typedef struct {
int key;
int value;
```

```
} Entry;
int search(int table[], int key)
{
int index = key % TABLE_SIZE;
while (table[index] != 0)
{
if (table[index] == key) {
return index;
}
index = (index + 1) % TABLE_SIZE;
}
return -1;
}
int main() {
int hashTable[TABLE_SIZE] = {0};
hashTable[2] = 45;
hashTable[5] = 62;
hashTable[8] = 78;
hashTable[1] = 91;
hashTable[4] = 34;
int searchKey = 78; int searchResult = search(hashTable, searchKey);
if (searchResult != -1) { printf("Value %d found at index %d\n", searchKey, searchResult); } else { printf("V
alue %d not found in the hash table.\n", searchKey);
 }

return 0;
}
```

----

40) Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Insert (1, 5): Assign the pair {1, 5} at the index (1%20 =1) in the Hash Table.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 20
typedef struct {
int key;
int value;
} Pair;
void insert(Pair table[], int key, int value) {
int index = key % TABLE_SIZE;
table[index].key = key;
table[index].value = value;
}
```

```
void display(Pair table[]) {
for (int i = 0; i < TABLE_SIZE; i++) {
if (table[i].key != 0)
{
printf("Index %d: (%d, %d)\n", i, table[i].key, table[i].value);
}
}
}
int main()
{
Pair hashTable[TABLE_SIZE] = {0};

insert(hashTable, 1, 5);
insert(hashTable, 2, 15);
insert(hashTable, 3, 20);
insert(hashTable, 4, 7);
display(hashTable);
return 0;
}
```

----

41) Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an arra y of capacity 20 is used as a Hash Table: Insert(2, 15): Assign the pair {2, 15} at the index (2%20 =2) in th e Hash Table.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 20
typedef struct {
int key;
int value;
} Pair;
void insert(Pair table[], int key, int value)
{
int index = key % TABLE_SIZE;
table[index].key = key;
table[index].value = value;
}
void display(Pair table[]) {
for (int i = 0; i < TABLE_SIZE; i++) {
if (table[i].key != 0)
{
printf("Index %d: (%d, %d)\n", i, table[i].key, table[i].value);
}
```

```
}
}
int main()
{
Pair hashTable[TABLE_SIZE] = {0};
insert(hashTable, 1, 5);
insert(hashTable, 2, 15);
insert(hashTable, 3, 20);
insert(hashTable, 4, 7);
display(hashTable);
return 0;
}
```

----

42) Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Insert(3, 20): Assign the pair {3, 20} at the index (3%20 =3) in the Hash Table.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 20
typedef struct {
int key;
int value;
} Pair;
void insert(Pair table[], int key, int value) {
int index = key % TABLE_SIZE; table[index].key = key;
table[index].value = value;
}
void display(Pair table[]) {
for (int i = 0; i < TABLE_SIZE; i++)
{
 if (table[i].key != 0)
{
printf("Index %d: (%d, %d)\n", i, table[i].key, table[i].value);
}
}
}
int main()
{
Pair hashTable[TABLE_SIZE] = {0};
insert(hashTable, 1, 5);
insert(hashTable, 2, 15);
insert(hashTable, 3, 20);
```

```
insert(hashTable, 4, 7);
display(hashTable);
return 0;
}
```

----

43) Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an arra y of capacity 20 is used as a Hash Table: Find(4): The key 4 is stored at the index (4%20 = 4). Therefore, print the 7 as it is the value of the key, 4, at index 4 of the Hash Table.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 20
typedef struct {
int key;
int value;
} Pair;
int find(Pair table[], int key) {
int index = key % TABLE_SIZE;
if (table[index].key == key) {
 return table[index].value;
}
else {
return -1;
}
 }
int main() {
 Pair hashTable[TABLE_SIZE] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}
};
int searchKey = 4; int searchResult = find(hashTable, searchKey);
if (searchResult != -1) {
printf("Value: %d\n", searchResult);
}
else
{ printf("Key %d not found in the hash table.\n", searchKey);
}
return 0;
}
```

----

44) Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Delete (4): The key 4 is stored at the index (4%20 = 4). After deleting Key 4, the Hash Table has keys {1, 2, 3}.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 20
typedef struct {
int key;
int value;
} Pair;
void deleteKey(Pair table[], int key)
{
int index = key % TABLE_SIZE;
if (table[index].key == key) {
table[index].key = 0;
table[index].value = 0;
printf("Key %d deleted from the hash table.\n", key);
}
else
{
printf("Key %d not found in the hash table.\n", key);
}
}
void display(Pair table[])
{
printf("Keys in the hash table: ");
for (int i = 0; i < TABLE_SIZE; i++)
{
if (table[i].key != 0)
{
printf("%d ", table[i].key);
}
}
printf("\n");
}
int main() {
Pair hashTable[TABLE_SIZE] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
deleteKey(hashTable, 4);
display(hashTable);
return 0;
}
```

----

45) Suppose the operations are performed on an array of pairs, {{1, 5}, {2, 15}, {3, 20}, {4, 7}}. And an array of capacity 20 is used as a Hash Table: Find(4): Print -1, as the key 4 does not exist in the Hash Table.

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 20
typedef struct {
int key;
int value;
} Pair;
int find(Pair table[], int key)
{
int index = key % TABLE_SIZE;
if (table[index].key == key) {
return table[index].value;
}
else
{
return -1;
}
 }
int main()
{
Pair hashTable[TABLE_SIZE] = {{1, 5}, {2, 15}, {3, 20}, {4, 7}};
int searchKey = 4;
int searchResult = find(hashTable, searchKey);
if (searchResult != -1)
{
printf("Value: %d\n", searchResult);
}
else {
printf("-1\n"); }
return 0;
}
```

