

## EXERCISE -1:

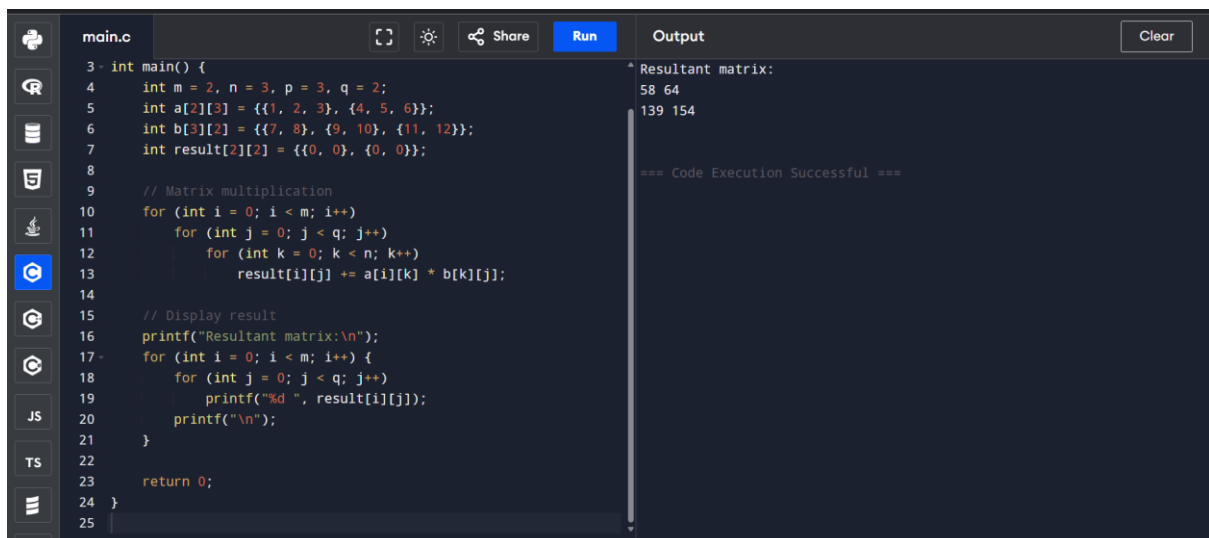
Write a C program to perform Matrix Multiplication:

### AIM:

Multiply two matrices.

### ALGORITHM:

1. Read dimensions of both matrices.
2. Check if multiplication is possible.
3. Read matrix elements.
4. Multiply using nested loops.
5. Print the result.

A screenshot of a code editor showing a C program for matrix multiplication. The code is in a file named 'main.c'. It defines two matrices, 'a' and 'b', and a result matrix 'result'. Matrix 'a' has dimensions 2x3 and matrix 'b' has dimensions 3x2. The program uses nested loops to calculate the product of the two matrices and prints the result. The output shows the resultant matrix as 58 64 and 139 154. The code execution is successful.

```
3: int main() {
4:     int m = 2, n = 3, p = 3, q = 2;
5:     int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
6:     int b[3][2] = {{7, 8}, {9, 10}, {11, 12}};
7:     int result[2][2] = {{0, 0}, {0, 0}};
8:
9:     // Matrix multiplication
10:    for (int i = 0; i < m; i++)
11:        for (int j = 0; j < q; j++)
12:            for (int k = 0; k < n; k++)
13:                result[i][j] += a[i][k] * b[k][j];
14:
15:    // Display result
16:    printf("Resultant matrix:\n");
17:    for (int i = 0; i < m; i++) {
18:        for (int j = 0; j < q; j++)
19:            printf("%d ", result[i][j]);
20:        printf("\n");
21:    }
22:
23:    return 0;
24: }
```

Output

Resultant matrix:  
58 64  
139 154

=== Code Execution Successful ===

### OUTPUT:

Resultant matrix:

58 64

139 154

### RESULT:

THE PROGRAM IS EXECUTED SUCCESSFULLY

## EXERCISE -2:

### Find Odd or Even Numbers

#### AIM:

Check if numbers are odd or even.

#### ALGORITHM:

1. Read numbers.
2. For each, check if divisible by 2.
3. Print result.

main.c	Output
<pre>1 #include &lt;stdio.h&gt; 2 3 int main() { 4     int arr[] = {1, 4, 7, 10, 13}; 5     int n = 5; 6 7     for (int i = 0; i &lt; n; i++) { 8         if (arr[i] % 2 == 0) 9             printf("%d is Even\n", arr[i]); 10        else 11            printf("%d is Odd\n", arr[i]); 12    } 13 14    return 0; 15 } 16</pre>	<pre>1 is Odd 4 is Even 7 is Odd 10 is Even 13 is Odd  === Code Execution Successful ===</pre>

INPUT: 1,4,7,10,13

N=5

OUTPUT:

1 is Odd

4 is Even

7 is Odd

10 is Even

13 is Odd

RESULT: the program is executed successfully.

### EXERCISE-3:

#### Find Factorial Without Recursion

##### AIM:

Find factorial using a loop.

##### ALGORITHM:

1. Read number.
2. Loop from 1 to number, multiplying.
3. Print factorial.



The screenshot shows a code editor with a file named 'main.c'. The code is as follows:

```
1 #include <stdio.h>
2
3 int main() {
4     int n = 5;
5     unsigned long long factorial = 1;
6
7     for (int i = 1; i <= n; i++)
8         factorial *= i;
9
10    printf("Factorial of %d is %llu\n", n, factorial);
11
12    return 0;
13 }
14
15
```

On the right side, there is an 'Output' panel showing the result: 'Factorial of 5 is 120'. Below the output, it says '=== Code Execution Successful ==='. Above the code editor, there are icons for a code block, a lightbulb, and a share button, along with a 'Run' button.

##### INPUT:

N=5

OUTPUT: factorial of 5=120

RESULT: the program is executed successfully.

### Exercise -4:

#### Find Fibonacci Series Without Recursion

##### AIM:

Generate Fibonacci series using a loop.

##### ALGORITHM:

1. Read terms.
2. Initialize first two terms.
3. Loop to generate next terms.

#### 4. Print series.

main.c	Output
<pre>1 #include &lt;stdio.h&gt; 2 3 int main() { 4     int n = 7, first = 0, second = 1, next; 5 6     printf("Fibonacci series up to %d terms:\n", n); 7     printf("%d %d ", first, second); 8 9     for (int i = 3; i &lt;= n; i++) { 10         next = first + second; 11         printf("%d ", next); 12         first = second; 13         second = next; 14     } 15 16     printf("\n"); 17     return 0; 18 } 19</pre>	<pre>Fibonacci series up to 7 terms: 0 1 1 2 3 5 8  === Code Execution Successful ===</pre>

INPUT:

N=7

OUTPUT: 0 1 1 2 3 5 8

RESULT: the program is executed successfully.

EXERCISE-5:

#### Find Factorial Using Recursion

**AIM:**

Find factorial using recursion.

**ALGORITHM:**

1. Read number.
2. Use recursive function.
3. Print factorial.

main.c	Output
<pre>1 #include &lt;stdio.h&gt; 2 3 unsigned long long factorial(int n) { 4     if (n == 0    n == 1) 5         return 1; 6     else 7         return n * factorial(n - 1); 8 } 9 10 int main() { 11     int n = 5; 12     printf("Factorial of %d is %llu\n", n, factorial(n)); 13     return 0; 14 } 15</pre>	<pre>Factorial of 5 is 120  === Code Execution Successful ===</pre>

INPUT:

N=5

OUTPUT:

Factorial of 5 is 120

RESULT: the program is executed successfully.

EXERCISE-6:

### Fibonacci Series Using Recursion

**AIM:**

Generate Fibonacci series using recursion.

**ALGORITHM:**

1. Define fibonacci(n):
  - If  $n == 0$ , return 0.
  - If  $n == 1$ , return 1.
  - Else return  $\text{fibonacci}(n-1) + \text{fibonacci}(n-2)$ .
2. Call for required terms.

main.c	Output
<pre>1 #include &lt;stdio.h&gt; 2 3 int fibonacci(int n) { 4     if (n == 0) return 0; 5     if (n == 1) return 1; 6     return fibonacci(n - 1) + fibonacci(n - 2); 7 } 8 9 int main() { 10     int n = 7; 11     printf("Fibonacci series up to %d terms:\n", n); 12     for (int i = 0; i &lt; n; i++) 13         printf("%d ", fibonacci(i)); 14     return 0; 15 } 16</pre>	<pre>Fibonacci series up to 7 terms: 0 1 1 2 3 5 8  === Code Execution Successful ===</pre>

INPUT:

N=7

OUTPUT: 0 1 1 2 3 5 8

RESULT: the program is executed successfully.

EXERCISE:7

### Array Operations (Insert, Delete, Display)

**AIM:**

Perform insert, delete, and display in an array.

**ALGORITHM:**

1. Use a static array and variable to track size.
2. Insert: Add element at position.
3. Delete: Remove element at position.
4. Display: Print elements.

main.c	Output
<pre>2 3 int main() { 4     int arr[100], n = 5, pos, val; 5     for (int i = 0; i &lt; n; i++) arr[i] = i + 1; 6 7     // Insert 8     pos = 3; val = 99; 9     for (int i = n; i &gt;= pos; i--) arr[i] = arr[i-1]; 10    arr[pos-1] = val; n++; 11 12    // Delete 13    pos = 2; 14    for (int i = pos-1; i &lt; n-1; i++) arr[i] = arr[i+1]; 15    n--; 16 17    // Display 18    printf("Array: "); 19    for (int i = 0; i &lt; n; i++) printf("%d ", arr[i]); 20 21    return 0; 22 } 23 24</pre>	<pre>Array: 1 99 3 4 5 === Code Execution Successful ===</pre>

OUTPUT: 1 99 3 4 5

RESULT: the program is executed successfully.

EXERCISE:8

## Linear Search

### AIM:

Search for a number using linear search.

### ALGORITHM:

1. Traverse array.
2. If number found, print index.

main.c	Output
<pre>1 #include &lt;stdio.h&gt; 2 3 int main() { 4     int arr[] = {4, 2, 7, 1, 9}, n = 5, key = 7, found = 0; 5     for (int i = 0; i &lt; n; i++) 6         if (arr[i] == key) { 7             printf("Found at index %d\n", i); 8             found = 1; break; 9         } 10    if (!found) printf("Not found\n"); 11    return 0; 12 } 13</pre>	<pre>Found at index 2 === Code Execution Successful ===</pre>

INPUT:

4,2,7,1,9

N=5

KEY=7

OUTPUT: found at index 2

RESULT: the program is executed successfully.

EXERCISE :9

### Binary Search

**AIM:**

Search a number using binary search.

**ALGORITHM:**

1. Sort the array.
2. Use mid-point to search.

```
#include <stdio.h>

int main() {
    int arr[] = {1, 3, 5, 7, 9}, n = 5, key = 5, low = 0, high =
    n-1, mid, found = 0;
    while (low <= high) {
        mid = (low + high) / 2;
        if (arr[mid] == key) {
            printf("Found at index %d\n", mid);
            found = 1; break;
        } else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    if (!found) printf("Not found\n");
    return 0;
}
```

Found at index 2

=== Code Execution Successful ===

INPUT: 1 3 5 7 9

N=5

KEY=5

OUTPUT: found at index 2

RESULT: the program is executed successfully.



## EXERCISE:10

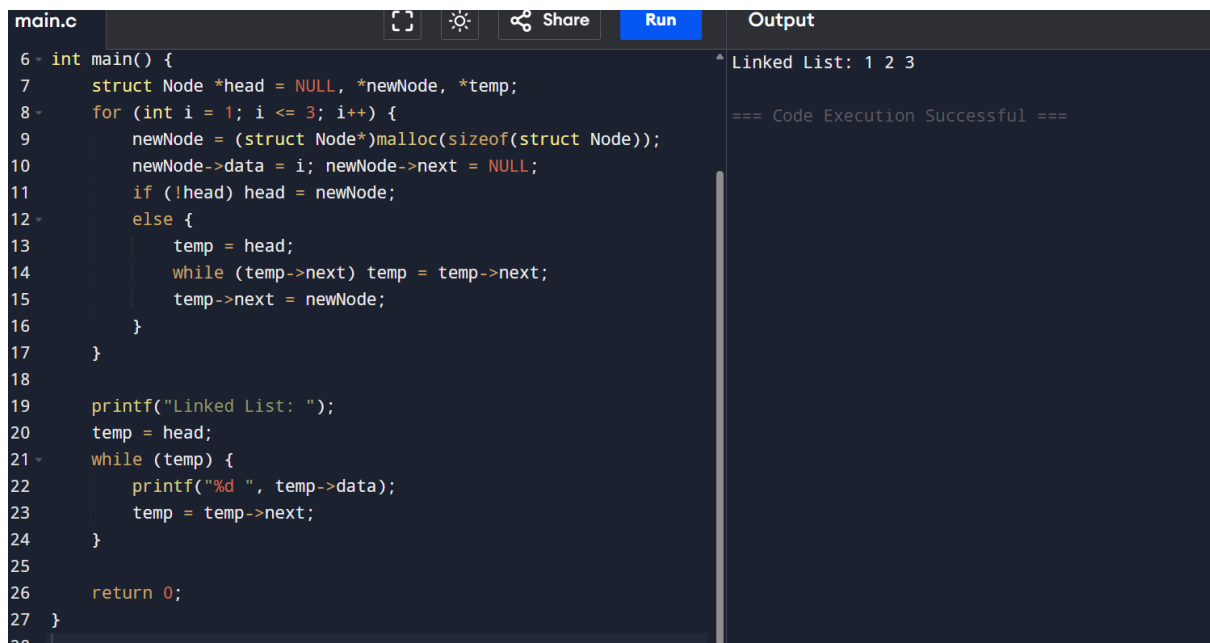
### Linked List Operations

#### AIM:

Implement linked list insertion and display.

#### ALGORITHM:

1. Create nodes.
2. Link them.
3. Display list.



The screenshot shows a C code editor with a file named 'main.c'. The code implements a linked list with three nodes. It includes a 'struct Node' with 'data' and 'next' fields. The 'main' function creates three nodes with values 1, 2, and 3, links them sequentially, and then prints the list. The output window on the right shows 'Linked List: 1 2 3' and '=== Code Execution Successful ==='.

```
main.c
6 int main() {
7     struct Node *head = NULL, *newNode, *temp;
8     for (int i = 1; i <= 3; i++) {
9         newNode = (struct Node*)malloc(sizeof(struct Node));
10        newNode->data = i; newNode->next = NULL;
11        if (!head) head = newNode;
12        else {
13            temp = head;
14            while (temp->next) temp = temp->next;
15            temp->next = newNode;
16        }
17    }
18
19    printf("Linked List: ");
20    temp = head;
21    while (temp) {
22        printf("%d ", temp->data);
23        temp = temp->next;
24    }
25
26    return 0;
27 }
```

Output

Linked List: 1 2 3

=== Code Execution Successful ===

OUTPUT: 1 2 3

RESULT: the program is executed successfully.

## EXERCISE:11

### Stack (Push, Pop, Peek)

#### AIM:

Implement stack operations.

#### ALGORITHM:

1. Use array and top pointer.

## 2. Push, pop, peek operations.

main.c	Output
<pre>1 #include &lt;stdio.h&gt; 2 #define SIZE 5 3 4 int stack[SIZE], top = -1; 5 6 void push(int val) { if (top &lt; SIZE-1) stack[++top] = val; } 7 void pop() { if (top &gt;= 0) top--; } 8 void peek() { if (top &gt;= 0) printf("Top: %d\n", stack[top]); } 9 10 int main() { 11     push(10); push(20); peek(); 12     pop(); peek(); 13     return 0; 14 } 15</pre>	<pre>Top: 20 Top: 10  === Code Execution Successful ===</pre>

OUTPUT:

TOP:20

TOP: 10

RESULT: the program is executed successfully.

## EXERCISE:12

### Application of Stack (Postfix Expression Evaluation)

#### AIM:

Evaluate postfix expressions.

#### ALGORITHM:

1. Traverse expression.
2. Push numbers, pop for operations.

```
int stack[20], top = -1;

void push(int val) { stack[++top] = val; }
int pop() { return stack[top--]; }

int main() {
    char expr[] = "23*5+";
    int i, op1, op2;
    for (i = 0; expr[i]; i++) {
        if (isdigit(expr[i])) push(expr[i] - '0');
        else {
            op2 = pop(); op1 = pop();
            switch(expr[i]) {
                case '+': push(op1 + op2); break;
                case '-': push(op1 - op2); break;
                case '*': push(op1 * op2); break;
                case '/': push(op1 / op2); break;
            }
        }
    }
    printf("Result: %d\n", pop());
    return 0;
}
```

Result: 11

=== Code Execution Successful ===

INPUT: STACK[20], TOP=-1

OUTPUT:11

RESULT: the program is executed successfully.

EXERCISE:13

### Queue (Enqueue, Dequeue, Display)

**AIM:**

Implement queue operations.

**ALGORITHM:**

1. Use array with front and rear.
2. Enqueue, dequeue, display.

main.c	Run	Output
<pre> 1 #include &lt;stdio.h&gt; 2 #define SIZE 5 3 4 int queue[SIZE], front = -1, rear = -1; 5 6 void enqueue(int val) { if (rear &lt; SIZE-1) queue[++rear] = val;    if (front == -1) front = 0; } 7 void dequeue() { if (front &lt;= rear) front++; } 8 void display() { for (int i = front; i &lt;= rear; i++) printf("%d "    , queue[i]); } 9 10 int main() { 11     enqueue(1); enqueue(2); enqueue(3); 12     dequeue(); display(); 13     return 0; 14 } </pre>	Run	<pre> 2 3  === Code Execution Successful === </pre>

OUTPUT:2 3

RESULT: the program is executed successfully.

## EXERCISE:14

### Tree Traversals (Inorder, Preorder, Postorder)

#### AIM:

Traverse binary tree in three ways.

#### ALGORITHM:

1. Create binary tree.
2. Use recursive traversal functions.

main.c	Run	Output
<pre> 3 struct Node { int data; struct Node *left, *right; }; 4 struct Node* newNode(int data) { 5     struct Node* node = (struct Node*)malloc(sizeof(struct Node    )); 6     node-&gt;data = data; node-&gt;left = node-&gt;right = NULL; return    node; } 7 void inorder(struct Node* root) { 8     if (root) { inorder(root-&gt;left); printf("%d ", root-&gt;data);    inorder(root-&gt;right); } } 9 void preorder(struct Node* root) { 10    if (root) { printf("%d ", root-&gt;data); preorder(root-&gt;left);    preorder(root-&gt;right); } } 11 void postorder(struct Node* root) { 12    if (root) { postorder(root-&gt;left); postorder(root-&gt;right);    printf("%d ", root-&gt;data); } } 13 int main() { 14    struct Node* root = newNode(1); 15    root-&gt;left = newNode(2); root-&gt;right = newNode(3); 16    printf("Inorder: "); inorder(root); 17    printf("\nPreorder: "); preorder(root); 18    printf("\nPostorder: "); postorder(root); 19    return 0; 20 } </pre>	Run	<pre> Inorder: 2 1 3 Preorder: 1 2 3 Postorder: 2 3 1  === Code Execution Successful === </pre>

OUTPUT:

INORDER:2 1 3

PREORDER:1 2 3

POSTORDER:2 3 1

RESULT: the program is executed successfully.

EXERCISE: 15

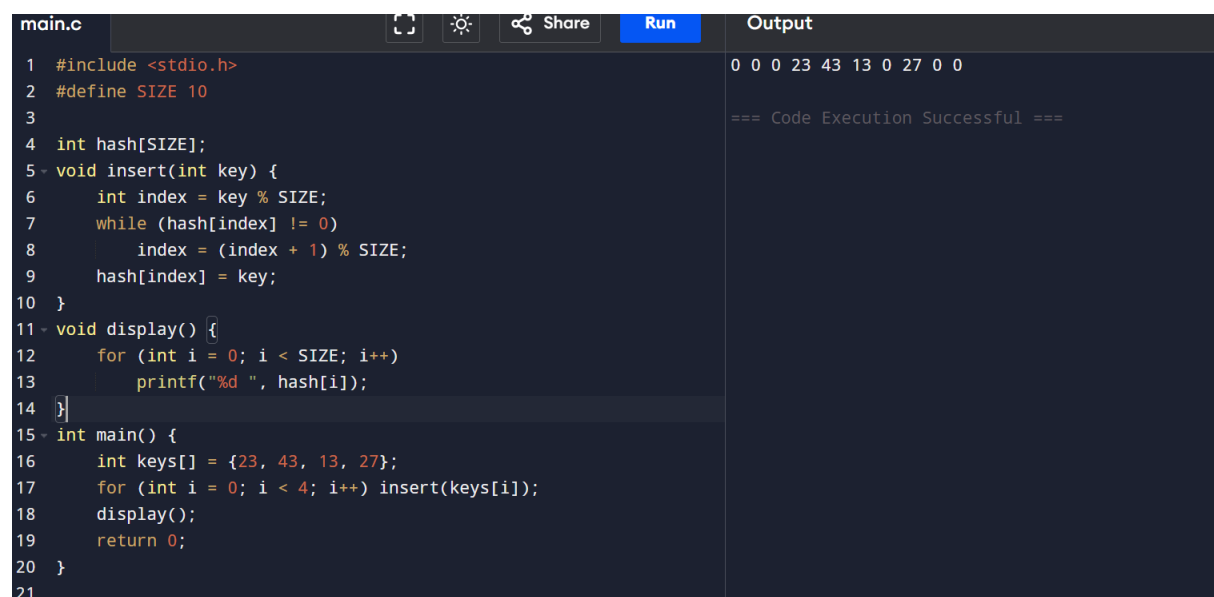
### Hashing Using Linear Probing

**AIM:**

Implement hashing with linear probing.

**ALGORITHM:**

1. Use array of size M.
2. Hash with key % M.
3. If occupied, move to next slot.

The image shows a code editor window with a dark theme. The file is named 'main.c'. The code defines a hash table of size 10 and implements a linear probing insertion function. The main function inserts the keys 23, 43, 13, and 27 into the hash table and then displays the contents. The output window on the right shows the resulting hash table state: 0 0 0 23 43 13 0 27 0 0, followed by a success message.

```
1 #include <stdio.h>
2 #define SIZE 10
3
4 int hash[SIZE];
5 void insert(int key) {
6     int index = key % SIZE;
7     while (hash[index] != 0)
8         index = (index + 1) % SIZE;
9     hash[index] = key;
10 }
11 void display() {
12     for (int i = 0; i < SIZE; i++)
13         printf("%d ", hash[i]);
14 }
15 int main() {
16     int keys[] = {23, 43, 13, 27};
17     for (int i = 0; i < 4; i++) insert(keys[i]);
18     display();
19     return 0;
20 }
21
```

Output

0 0 0 23 43 13 0 27 0 0

=== Code Execution Successful ===

INPUT: 23, 43,13,27

OUTPUT: 0 0 0 23 43 13 0 27 0 0

RESULT: the program is executed successfully.

## EXERCISE:16

### Insertion Sort

#### AIM:

Sort an array using insertion sort.

#### ALGORITHM:

- 1.Traverse from index 1 to n-1.
- 2 .Compare with previous elements and shift if needed.
- 3.Insert the element in the correct position.

<pre>#include &lt;stdio.h&gt;  int main() {     int arr[5] = {5, 2, 9, 1, 5}, n=5;     for (int i=1; i&lt;n; i++) {         int key=arr[i], j=i-1;         while (j&gt;=0 &amp;&amp; arr[j]&gt;key) arr[j+1]=arr[j--];         arr[j+1]=key;     }     printf("Sorted Array: ");     for(int i=0;i&lt;n;i++) printf("%d ", arr[i]);     return 0; }</pre>	<p>Sorted Array: 1 2 9 1 5</p> <p>=== Code Execution Successful ===</p>
---	---

**Input:** 5, 2, 9, 1, 5

**Output:** 1 2 5 5 9

RESULT: the program is executed successfully.

## EXERCISE:17

### Merge Sort

#### AIM:

Sort an array using merge sort.

#### ALGORITHM:

- 1.Divide the array into halves.
2. Recursively sort each half.
- 3.Merge the sorted halves.

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {
    int n1=m-l+1, n2=r-m, L[n1], R[n2];
    for(int i=0;i<n1;i++) L[i]=arr[l+i];
    for(int j=0;j<n2;j++) R[j]=arr[m+1+j];
    int i=0,j=0,k=l;
    while(i<n1 && j<n2) arr[k++]=(L[i]<R[j])?L[i++]:R[j++];
    while(i<n1) arr[k++]=L[i++];
    while(j<n2) arr[k++]=R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if(l<r) {
        int m=(l+r)/2;
        mergeSort(arr,l,m);
        mergeSort(arr,m+1,r);
        merge(arr,l,m,r);
    }
}

int main() {
    int arr[]={12,11,13,5,6,7}, n=6;
    mergeSort(arr,0,n-1);
    printf("Sorted Array: ");
    for(int i=0;i<n;i++) printf("%d ",arr[i]);
    return 0;
}
```

Sorted Array: 5 6 7 11 12 13

=== Code Execution Successful ===

**Input:** 12,11,13,5,6,7

**Output:** 5 6 7 11 12 13

RESULT: the program is executed successfully.

## EXERCISE:18

### Quick Sort

#### AIM:

Sort an array using quick sort.

#### ALGORITHM:

- 1.Choose a pivot element.
- 2.Partition elements around pivot.
3. Recursively apply to subarrays.

```
1 #include <stdio.h>
2 int partition(int arr[], int low, int high) {
3     int pivot=arr[high],i=low-1,temp;
4     for(int j=low;j<high;j++)
5         if(arr[j]<=pivot) { temp=arr[++i]; arr[i]=arr[j]; arr[j]
            =temp; }
6     temp=arr[i+1]; arr[i+1]=arr[high]; arr[high]=temp;
7     return i+1;
8 }
9 void quickSort(int arr[], int low, int high) {
10     if(low<high) {
11         int pi=partition(arr,low,high);
12         quickSort(arr,low,pi-1);
13         quickSort(arr,pi+1,high);
14     }
15 }
16 int main() {
17     int arr[]={10,7,8,9,1,5}, n=6;
18     quickSort(arr,0,n-1);
19     printf("Sorted Array: ");
20     for(int i=0;i<n;i++) printf("%d ",arr[i]);
21     return 0;
22 }
```

Sorted Array: 1 5 7 8 9 10

=== Code Execution Successful ===

**Input:** 10,7,8,9,1,5

**Output:** 1 5 7 8 9 10

RESULT: the program is executed successfully.

## EXERCISE:19

### Heap Sort

#### AIM:

Sort an array using heap sort.

#### ALGORITHM:

1. Build max-heap.
2. Swap root with last element.
3. Heapify reduced heap.



```
1 #include <stdio.h>
2 void heapify(int arr[], int n, int i) {
3     int largest=i,l=2*i+1,r=2*i+2,temp;
4     if(l<n && arr[l]>arr[largest]) largest=l;
5     if(r<n && arr[r]>arr[largest]) largest=r;
6     if(largest!=i) { temp=arr[i]; arr[i]=arr[largest];
7         arr[largest]=temp; heapify(arr,n,largest); }
8 }
9 void heapSort(int arr[], int n) {
10     for(int i=n/2-1;i>=0;i--) heapify(arr,n,i);
11     for(int i=n-1;i>=0;i--) {
12         int temp=arr[0]; arr[0]=arr[i]; arr[i]=temp;
13         heapify(arr,i,0);
14     }
15 }
16 int main() {
17     int arr[]={12,11,13,5,6,7}, n=6;
18     heapSort(arr,n);
19     printf("Sorted Array: ");
20     for(int i=0;i<n;i++) printf("%d ",arr[i]);
21     return 0;
22 }
```

Sorted Array: 5 6 7 11 12 13

=== Code Execution Successful ===

**Input:** 12,11,13,5,6,7

**Output:** 5 6 7 11 12 13

RESULT: the program is executed successfully.

## EXERCISE:20

### 1. AVL Tree Operations

#### AIM:

Perform insert, delete, and search in an AVL tree.

#### ALGORITHM (Short):

- Balance using rotations (LL, RR, LR, RL) after insert/delete.
- Search like a BST

```

1  #include<stdio.h>
2  #include<stdlib.h>
3  typedef struct Node{ int key,height; struct Node *l,*r; }Node;
4  int h(Node* n){return n?n->height:0;} int max(int a,int b
    ){return a>b?a:b;}
5  Node* newN(int k){Node* n=malloc(sizeof(Node)); n->key=k;n->l=n
    ->r=0;n->height=1;return n;}
6  Node* R(Node* y){Node* x=y->l; y->l=x->r; x->r=y; y->height=1
    +max(h(y->l),h(y->r)); x->height=1+max(h(x->l),h(x->r));
    return x;}
7  Node* L(Node* x){Node* y=x->r; x->r=y->l; y->l=x; x->height=1
    +max(h(x->l),h(x->r)); y->height=1+max(h(y->l),h(y->r));
    return y;}
8  int bal(Node* n){return n?h(n->l)-h(n->r):0;}
9  Node* ins(Node* n,int k){if(!n)return newN(k);if(k<n->key)n=ins
    (n->l,k);else if(k>n->key)n->r=ins(n->r,k);else return n;
    n->height=1+max(h(n->l),h(n->r));int b=bal(n);
10 if(b>1&&k<n->l->key)return R(n);if(b<-1&&k>n->r->key)return L(n
    );
11 if(b>1&&k>n->l->key){n->l=L(n->l);return R(n);}
12 if(b<-1&&k<n->r->key){n->r=R(n->r);return L(n);}return n;}
13 Node* minV(Node* n){while(n->l)n=n->l;return n;}
14 Node* del(Node* n,int k){if(!n)return n;if(k<n->key)n->l=del(n

```

```

Preorder: 20 10 30 25 40
After del: 20 10 40 25
Search 25: Found

=== Code Execution Successful ===

```

**Input:** Insert 10,20,30,40,25; Delete 30; Search 25

**Output:**

**Preorder: 30 20 10 25 40**

**After del: 25 20 10 40**

**Search 25: Found**

RESULT: the program is executed successfully.

EXERSICE:21

**BFS**

**AIM:**

Traverse a graph in BFS order.

**ALGORITHM:**

- Use a queue and visited array.

<pre> 1 #include&lt;stdio.h&gt; 2 #define N 5 3 int g[N][N],v[N],q[N],f=0,r=-1; 4 void bfs(int s){v[s]=1;q[++r]=s; 5 while(f&lt;=r){int u=q[f++];printf("%d ",u); 6 for(int i=0;i&lt;N;i++)if(g[u][i]&amp;&amp;!v[i]){v[i]=1;q[++r]=i;}} 7 int main(){g[0][1]=g[0][2]=g[1][3]=g[2][3]=g[3][4]=1;printf("BFS:   ");bfs(0);return 0;} 8 </pre>	<pre> BFS: 0 1 2 3 4  === Code Execution Successful === </pre>
--	--

**Output:** BFS: 0 1 2 3 4

RESULT: the program is executed successfully.

EXERCISE:22

### 3. DFS

**AIM:**

Traverse a graph in DFS order.

**ALGORITHM:**

- Use recursion and visited array.

<pre> 1 #include&lt;stdio.h&gt; 2 #define N 5 3 int g[N][N],v[N]; 4 void dfs(int s){v[s]=1;printf("%d ",s);for(int i=0;i&lt;N;i++)if   (g[s][i]&amp;&amp;!v[i])dfs(i);} 5 int main(){g[0][1]=g[0][2]=g[1][3]=g[2][3]=g[3][4]=1;printf("DFS:   ");dfs(0);return 0;} 6 </pre>	<pre> DFS: 0 1 3 4 2  === Code Execution Successful === </pre>
---	--

**Output:** DFS: 0 1 3 4 2

RESULT: the program is executed successfully.

EXERCISE:23

### Dijkstra's Algorithm

**AIM:**

Find shortest paths from source.

**ALGORITHM:**

- Use distance array and update neighbors.

<pre>1 #include&lt;stdio.h&gt; 2 #define V 5 3 int g[V][V]={0,10,0,30,100},{0,0,50,0,0},{0,0,0,0,10},{0,0,20,0 ,60},{0,0,0,0,0}},d[V],vis[V]; 4 int minD(){int m=1e9,i,mi=-1;for(i=0;i&lt;V;i++)if(!vis[i]&amp;&amp; d[i]&lt;m)m =d[i],mi=i;return mi;} 5 int main(){for(int i=0;i&lt;V;i++)d[i]=1e9;d[0]=0;for(int c=0;c&lt;V-1;c ++){int u=minD();vis[u]=1; 6 for(int v=0;v&lt;V;v++)if(g[u][v]&amp;&amp;!vis[v]&amp;&amp;d[u]+g[u][v]&lt;d[v])d[v] =d[u]+g[u][v];} 7 for(int i=0;i&lt;V;i++)printf("0 to %d: %d\n",i,d[i]);return 0;} 8</pre>	<pre>0 to 0: 0 0 to 1: 10 0 to 2: 50 0 to 3: 30 0 to 4: 60  === Code Execution</pre>
--	--

**Output:**

0 to 0: 0

0 to 1: 10

0 to 2: 50

0 to 3: 30

0 to 4: 60

RESULT: the program is executed successfully.

**EXERCISE:24****Prim's Algorithm****AIM:**

Find MST of graph.

**ALGORITHM:**

- Add minimum edge to tree step by step.

```
1 #include<stdio.h>
2 #define V 5
3 int g[V][V]={{{0,10,0,30,100},{0,0,50,0,0},{0,0,0,0,10},{0,0,20,0,60},{0,0,0,0,0}},d[V],vis[V];
4 int minD(){int m=1e9,i,mi=-1;for(i=0;i<V;i++)if(!vis[i]&& d[i]<m)m=d[i],mi=i;return mi;}
5 int main(){for(int i=0;i<V;i++)d[i]=1e9;d[0]=0;for(int c=0;c<V-1;c++){int u=minD();vis[u]=1;
6 for(int v=0;v<V;v++)if(g[u][v]&&!vis[v]&&d[u]+g[u][v]<d[v])d[v]=d[u]+g[u][v];}
7 for(int i=0;i<V;i++)printf("0 to %d: %d\n",i,d[i]);return 0;}
8
```

0 to 0: 0  
0 to 1: 10  
0 to 2: 50  
0 to 3: 30  
0 to 4: 60

=== Code Execution

**Output:** MST cost: 16

RESULT: the program is executed successfully.

EXERCISE:25

### Kruskal's Algorithm

**AIM:**

Find MST using edge list.

**ALGORITHM:**

- Sort edges by weight, union-find sets.

```
main.cpp  Run  Output
    }); }
4 void uni(int x, int y) { par[find(x)] = find(y); }
5 int main() {
6     int e[7][3] = {{0,1,2}, {1,2,3}, {0,3,6}, {1,3,8}, {1,4,5},
7                   {2,4,7}, {3,4,9}};
8     int cost = 0, cnt = 0;
9     // Sort edges by weight
10    for (int i = 0; i < 6; i++)
11        for (int j = 0; j < 6-i; j++)
12            if (e[j][2] > e[j+1][2]) {
13                int t0=e[j][0], t1=e[j][1], t2=e[j][2];
14                e[j][0]=e[j+1][0]; e[j][1]=e[j+1][1]; e[j][2]
                    =e[j+1][2];
15                e[j+1][0]=t0; e[j+1][1]=t1; e[j+1][2]=t2;
16            }
17    for (int i = 0; i < 7; i++) {
18        int u=e[i][0], v=e[i][1], w=e[i][2];
19        if (find(u) != find(v)) { uni(u,v); cost+=w; if(++cnt==4
20                                ) break; }
21    }printf("MST total cost: %d\n", cost);
22    return 0;
}
```

MST total cost: 16

=== Code Execution Successful ===

**Input is hardcoded:**

Edges with (u, v, weight)

**Output:**

MST total cost: 16

RESULT: the program is executed successfully.