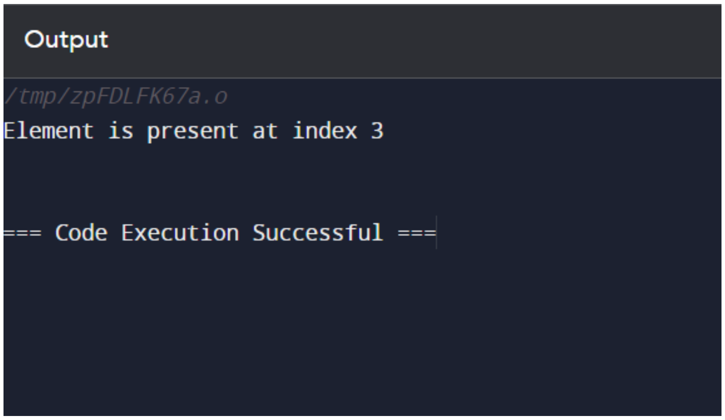
Exercise-1

Implement and analyze the following Algorithms using Divide and Conquer.

1.Binary Search with recursion.

```
#include <stdio.h>
int binarySearch(int arr[], int low, int high, int key) {
 if (high >= low) {
 int mid = low + (high - low) / 2;
 if (arr[mid] == key)
 return mid;
 if (arr[mid] > key)
 return binarySearch(arr, low, mid - 1, key);
 return binarySearch(arr, mid + 1, high, key);
 }
 return -1;
}
int main() {
 int arr[] = {2, 3, 4, 10, 40};
 int n = sizeof(arr) / sizeof(arr[0]);
 int key = 10;
 int result = binarySearch(arr, 0, n - 1, key);
 (result == -1) ? printf("Element is not present in array")
 : printf("Element is present at index %d\n", result);
 return 0;
}
```

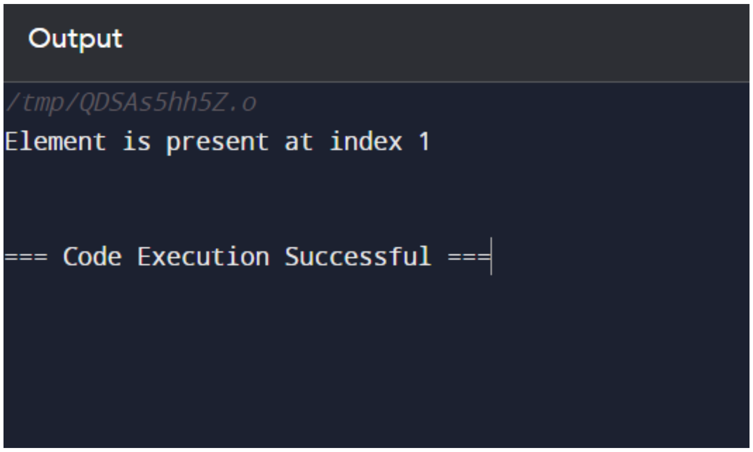
Output:



Binary Search without recursion.

```
#include <stdio.h>
int binarySearch(int arr[], int n, int key) {
 int low = 0, high = n - 1;
 while (low <= high) {
 int mid = low + (high - low) / 2;
 if (arr[mid] == key)
 return mid;
 else if (arr[mid] < key)
 low = mid + 1;
 else
 high = mid - 1;
 }
 return -1; // If element is not found
}
int main() {
 int arr[] = {2, 3, 4, 10, 40};
 int n = sizeof(arr) / sizeof(arr[0]);
 int key = 10;
 int result = binarySearch(arr, n, key);
 (result != -1)?printf("Element is present at index %d\n", result)
 :printf("Element is not present in array\n");
 return 0;
}
```

Output:



Merge Sort.

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


void merge(int arr[], int l, int m, int r)
{
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[n1], R[n2];


        for (i = 0; i < n1; i++)
                L[i] = arr[l + i];
        for (j = 0; j < n2; j++)
                R[j] = arr[m + 1 + j];



        i = 0;



        j = 0;
```

```
	k = l;
	while (i < n1 && j < n2) {
		if (L[i] <= R[j]) {
			arr[k] = L[i];
			i++;
		}
		else {
			arr[k] = R[j];
			j++;
		}
		k++;
	}


	while (i < n1) {
		arr[k] = L[i];
		i++;
		k++;
	}


	while (j < n2) {
		arr[k] = R[j];
		j++;
		k++;
	}
}


void mergeSort(int arr[], int l, int r)
{
	if (l < r) {

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


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

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

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


void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Output:

```
Output

/tmp/pyt28gg4dS.o
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13


=== Code Execution Successful ===
```

Quick Sort

```
#include <stdio.h>

void swap(int* a, int* b)
{
        int temp = *a;
        *a = *b;
        *b = temp;
}

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

        int pivot = arr[low];
        int i = low;
        int j = high;

        while (i < j) {
```

```
        while (arr[i] <= pivot && i <= high - 1) {
            i++;
        }


        while (arr[j] > pivot && j >= low + 1) {
            j--;
        }
        if (i < j) {
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[low], &arr[j]);
    return j;
}

void quickSort(int arr[], int low, int high)
{
    if (low < high) {

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


        quickSort(arr, low, partitionIndex - 1);
        quickSort(arr, partitionIndex + 1, high);
    }
}
int main()
{
    int arr[] = { 19, 17, 15, 12, 16, 18, 4, 11, 13 };
    int n = sizeof(arr) / sizeof(arr[0]);

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

```
	}
	quickSort(arr, 0, n - 1);

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

Output:

```
Output
/tmp/nsHnZH6cua.o
Original array: 19 17 15 12 16 18 4 11 13
Sorted array: 4 11 12 13 15 16 17 18 19

=== Code Execution Successful ===
```

Exercise-2

Implement following Algorithms using Greedy Method

4. Minimum-cost spanning tree

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

int comparator(const void* p1, const void* p2)
{
	const int(*x)[3] = p1;
	const int(*y)[3] = p2;

	return (*x)[2] - (*y)[2];
}

void makeSet(int parent[], int rank[], int n)
```

```
{
	for (int i = 0; i < n; i++) {
		parent[i] = i;
		rank[i] = 0;
	}
}

int findParent(int parent[], int component)
{
	if (parent[component] == component)
		return component;

	return parent[component]
		= findParent(parent, parent[component]);
}

void unionSet(int u, int v, int parent[], int rank[], int n)
{
	u = findParent(parent, u);
	v = findParent(parent, v);

	if (rank[u] < rank[v]) {
		parent[u] = v;
	}
	else if (rank[u] > rank[v]) {
		parent[v] = u;
	}
	else {
		parent[v] = u;

		rank[u]++;
	}
}

void kruskalAlgo(int n, int edge[n][3])
{
	qsort(edge, n, sizeof(edge[0]), comparator);
```

```
	int parent[n];
	int rank[n];
	makeSet(parent, rank, n);

	int minCost = 0;

	printf(
		"Following are the edges in the constructed MST\n");
	for (int i = 0; i < n; i++) {
		int v1 = findParent(parent, edge[i][0]);
		int v2 = findParent(parent, edge[i][1]);
		int wt = edge[i][2];


		if (v1 != v2) {
			unionSet(v1, v2, parent, rank, n);
			minCost += wt;
			printf("%d -- %d == %d\n", edge[i][0],
				edge[i][1], wt);
		}
	}

	printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

int main()
{
	int edge[5][3] = { { 0, 1, 10 },
					{ 0, 2, 6 },
					{ 0, 3, 5 },
					{ 1, 3, 15 },
					{ 2, 3, 4 } };

	kruskalAlgo(5, edge);

	return 0;
```

```
}
```

Output:

```
Output

/tmp/ZsF5kFDNFV.o
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19


=== Code Execution Successful ===
```

5.Single Source Shortest Path (Dijkstra's);

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

#define V 9


int minDistance(int dist[], bool sptSet[])
{
	int min = INT_MAX, min_index;

	for (int v = 0; v < V; v++)
		if (sptSet[v] == false && dist[v] <= min)
			min = dist[v], min_index = v;

	return min_index;
```

```
}

void printSolution(int dist[])
{
	printf("Vertex \t\t Distance from Source\n");
	for (int i = 0; i < V; i++)
		printf("%d \t\t\t\t %d\n", i, dist[i]);
}


void dijkstra(int graph[V][V], int src)
{
	int dist[V];

	bool sptSet[V];


	for (int i = 0; i < V; i++)
		dist[i] = INT_MAX, sptSet[i] = false;

	dist[src] = 0;

	// Find shortest path for all vertices
	for (int count = 0; count < V - 1; count++) {
		// Pick the minimum distance vertex from the set of
		// vertices not yet processed. u is always equal to
		// src in the first iteration.
		int u = minDistance(dist, sptSet);

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

		// Update dist value of the adjacent vertices of the
		// picked vertex.
		for (int v = 0; v < V; v++)

			// Update dist[v] only if is not in sptSet,
```

```
                // there is an edge from u to v, and total
                // weight of path from src to v through u is
                // smaller than current value of dist[v]
                if (!sptSet[v] && graph[u][v]
                    && dist[u] != INT_MAX
                    && dist[u] + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
        }

        // print the constructed distance array
        printSolution(dist);
}

// driver's code
int main()
{
        /* Let us create the example graph discussed above */
        int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                            { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                            { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                            { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                            { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                            { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                            { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                            { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                            { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

        // Function call
        dijkstra(graph, 0);

        return 0;
}
```

Output:

```
Output
/tmp/aFbKnsrDSe.o
Vertex          Distance from Source
0                   0
1                   4
2                   12
3                   19
4                   21
5                   11
6                   9
7                   8
8                   14


=== Code Execution Successful ===
```

Exercise-3

Implement following Algorithms using Dynamic programing

6. Optimal binary search trees

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

int sum(int freq[], int i, int j);

int optCost(int freq[], int i, int j)
{
```

```
if (j < i)
        return 0;
if (j == i)
        return freq[i];

int fsum = sum(freq, i, j);
int min = INT_MAX;


for (int r = i; r <= j; ++r)
{
        int cost = optCost(freq, i, r-1) +
                                optCost(freq, r+1, j);
        if (cost < min)
                min = cost;
}

return min + fsum;
}

int optimalSearchTree(int keys[], int freq[], int n)
{

        return optCost(freq, 0, n-1);
}


int sum(int freq[], int i, int j)
{
        int s = 0;
        for (int k = i; k <=j; k++)
        s += freq[k];
        return s;
}

int main()
{
```

```
        int keys[] = {10, 12, 20};
        int freq[] = {34, 8, 50};
        int n = sizeof(keys)/sizeof(keys[0]);
        printf("Cost of Optimal BST is %d ",
                        optimalSearchTree(keys, freq, n));
        return 0;
}
```

Output:

```
Output
/tmp/3a1Y5qFA8I.o
Cost of Optimal BST is 142

=== Code Execution Successful ===
```

7. Traveling salesperson problem

```
#include <stdio.h>
#include <limits.h>
#define MAX 9999
int n = 4;
int distan[20][20] = {
  {0, 22, 26, 30},
  {30, 0, 45, 35},
  {25, 45, 0, 60},
```

```
  {30, 35, 40, 0}};
int DP[32][8];
int TSP(int mark, int position) {
  int completed_visit = (1 << n) - 1;
  if (mark == completed_visit) {
     return distan[position][0];
  }
  if (DP[mark][position] != -1) {
     return DP[mark][position];
  }
  int answer = MAX;
  for (int city = 0; city < n; city++) {
    if ((mark & (1 << city)) == 0) {
      int newAnswer = distan[position][city] + TSP(mark | (1 << city), city);
      answer = (answer < newAnswer) ? answer : newAnswer;
    }
  }
  return DP[mark][position] = answer;
}
int main() {
  for (int i = 0; i < (1 << n); i++) {
    for (int j = 0; j < n; j++) {
      DP[i][j] = -1;
    }
  }
  printf("Minimum Distance Travelled -> %d\n", TSP(1, 0));
  return 0;
}
```

Output:

```
Output
/tmp/HDy0jhAz76.o
Minimum Distance Travelled -> 122


=== Code Execution Successful ===
```

Exercise-4

Implement following Algorithms using Backtracking
8. N-Queens problem

```
#define N 4
#include <stdbool.h>
#include <stdio.h>

void printSolution(int board[N][N])
{
	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++) {
			if(board[i][j])
				printf("Q ");
			else
				printf(". ");
		}
		printf("\n");
	}
```

```
}


bool isSafe(int board[N][N], int row, int col)
{
	int i, j;

	for (i = 0; i < col; i++)
		if (board[row][i])
			return false;

	for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
		if (board[i][j])
			return false;

	for (i = row, j = col; j >= 0 && i < N; i++, j--)
		if (board[i][j])
			return false;

	return true;
}

bool solveNQUtil(int board[N][N], int col)
{
	if (col >= N)
		return true;


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


		if (isSafe(board, i, col)) {

			board[i][col] = 1;

			if (solveNQUtil(board, col + 1))
				return true;
```

```
                    board[i][col] = 0;
                }
        }


        return false;
}


bool solveNQ()
{
        int board[N][N] = { { 0, 0, 0, 0 },
                                                  { 0, 0, 0, 0 },
                                                  { 0, 0, 0, 0 },
                                                  { 0, 0, 0, 0 } };

        if (solveNQUtil(board, 0) == false) {
                printf("Solution does not exist");
                return false;
        }

        printSolution(board);
        return true;
}

int main()
{
        solveNQ();
        return 0;
}
```

Output:

```
Output

/tmp/lGaBoVJ75Q.o
. . Q .
Q . . .
. . . Q
. Q . .


=== Code Execution Successful ===
```

9. Graph Coloring problem

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

#define V 4

void printSolution(int color[]);

bool isSafe(int v, bool graph[V][V], int color[], int c)
{
	for (int i = 0; i < V; i++)
		if (graph[v][i] && c == color[i])
			return false;
	return true;
}


bool graphColoringUtil(bool graph[V][V], int m, int color[],
```

```
                                        int v)
{

    if (v == V)
        return true;


    for (int c = 1; c <= m; c++) {

        if (isSafe(v, graph, color, c)) {
            color[v] = c;


            if (graphColoringUtil(graph, m, color, v + 1)
                == true)
                return true;


            color[v] = 0;
        }
    }


    return false;
}


bool graphColoring(bool graph[V][V], int m)
{

    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    if (graphColoringUtil(graph, m, color, 0) == false) {
        printf("Solution does not exist");
        return false;
```

```
        }

        printSolution(color);
        return true;
}

void printSolution(int color[])
{
        printf("Solution Exists:"
                " Following are the assigned colors \n");
        for (int i = 0; i < V; i++)
                printf(" %d ", color[i]);
        printf("\n");
}

int main()
{

        bool graph[V][V] = {
                { 0, 1, 1, 1 },
                { 1, 0, 1, 0 },
                { 1, 1, 0, 1 },
                { 1, 0, 1, 0 },
        };
        int m = 3;


        graphColoring(graph, m);
        return 0;
}
```

Output:

```
Output

/tmp/P0fF0BI57n.o
Solution Exists: Following are the assigned colors
 1  2  3  2


=== Code Execution Successful ===
```

Exercise-5

Implement following Tree Operations

10. AVL Tree

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

struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

int getHeight(struct Node *n){
    if(n==NULL)
        return 0;
    return n->height;
}
```

```
struct Node *createNode(int key){
    struct Node* node = (struct Node *) malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

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

int getBalanceFactor(struct Node * n){
    if(n==NULL){
        return 0;
    }
    return getHeight(n->left) - getHeight(n->right);
}

struct Node* rightRotate(struct Node* y){
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return x;
}

struct Node* leftRotate(struct Node* x){
    struct Node* y = x->right;
    struct Node* T2 = y->left;
```

```
    y->left = x;
    x->right = T2;

    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return y;
}

struct Node *insert(struct Node* node, int key){
    if (node == NULL)
        return  createNode(key);

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int bf = getBalanceFactor(node);

        if(bf>1 && key < node->left->key){
            return rightRotate(node);
        }
        if(bf<-1 && key > node->right->key){
            return leftRotate(node);
        }
    if(bf>1 && key > node->left->key){
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
    if(bf<-1 && key < node->right->key){
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
    return node;
```

```
}

void preOrder(struct Node *root)
{
    if(root != NULL)
    {

        printf(" %d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main(){
    struct Node * root = NULL;


    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 3);
    preOrder(root);
    return 0;
}
```

Output:

```
Output

/tmp/bPtOWdwINm.o
 4  2  1  3  5  6

=== Code Execution Successful ===
```

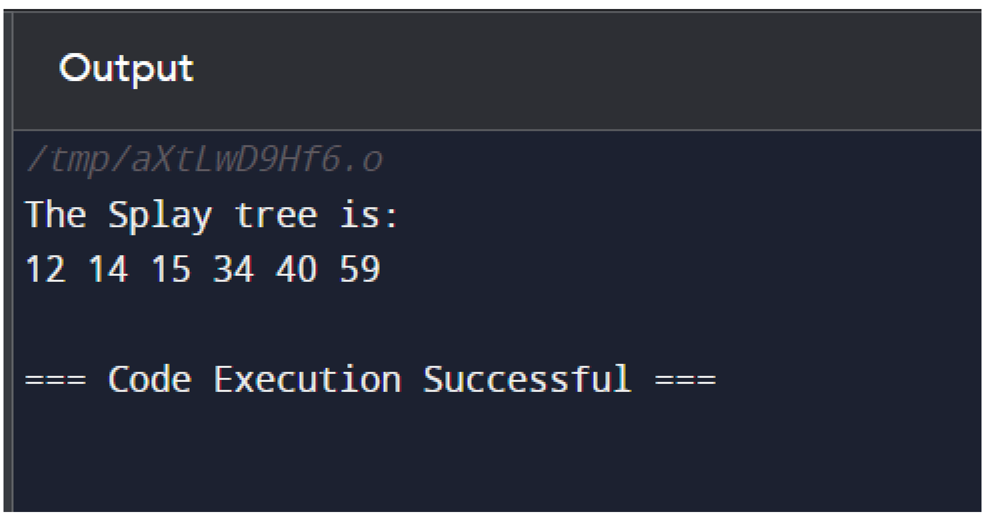
11. Splay Tree

```
#include <stdio.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *leftChild, *rightChild;
};
struct node* newNode(int data){
  struct node* Node = (struct node*)malloc(sizeof(struct node));
  Node->data = data;
  Node->leftChild = Node->rightChild = NULL;
  return (Node);
}
struct node* rightRotate(struct node *x){
  struct node *y = x->leftChild;
  x->leftChild = y->rightChild;
  y->rightChild = x;
  return y;
}
struct node* leftRotate(struct node *x){
  struct node *y = x->rightChild;
  x->rightChild = y->leftChild;
  y->leftChild = x;
  return y;
}
struct node* splay(struct node *root, int data){
  if (root == NULL || root->data == data)
    return root;
  if (root->data > data) {
    if (root->leftChild == NULL) return root;
    if (root->leftChild->data > data) {
      root->leftChild->leftChild = splay(root->leftChild->leftChild, data);
      root = rightRotate(root);
    } else if (root->leftChild->data < data) {
      root->leftChild->rightChild = splay(root->leftChild->rightChild, data);
```

```
      if (root->leftChild->rightChild != NULL)
        root->leftChild = leftRotate(root->leftChild);
    }
    return (root->leftChild == NULL)? root: rightRotate(root);
  } else {
    if (root->rightChild == NULL) return root;
    if (root->rightChild->data > data) {
      root->rightChild->leftChild = splay(root->rightChild->leftChild, data);
      if (root->rightChild->leftChild != NULL)
        root->rightChild = rightRotate(root->rightChild);
    } else if (root->rightChild->data < data) {
      root->rightChild->rightChild = splay(root->rightChild->rightChild, data);
      root = leftRotate(root);
    }
    return (root->rightChild == NULL)? root: leftRotate(root);
  }
}
struct node* insert(struct node *root, int k){
  if (root == NULL) return newNode(k);
  root = splay(root, k);
  if (root->data == k) return root;
  struct node *newnode = newNode(k);
  if (root->data > k) {
    newnode->rightChild = root;
    newnode->leftChild = root->leftChild;
    root->leftChild = NULL;
  } else {
    newnode->leftChild = root;
    newnode->rightChild = root->rightChild;
    root->rightChild = NULL;
  }
  return newnode;
}
void printTree(struct node *root){
  if (root == NULL)
    return;
  if (root != NULL) {
```

```
        printTree(root->leftChild);
        printf("%d ", root->data);
        printTree(root->rightChild);
    }
}
int main(){
    struct node* root = newNode(34);
    root->leftChild = newNode(15);
    root->rightChild = newNode(40);
    root->leftChild->leftChild = newNode(12);
    root->leftChild->leftChild->rightChild = newNode(14);
    root->rightChild->rightChild = newNode(59);
    printf("The Splay tree is: \n");
    printTree(root);
    return 0;
}
```

Output:



Exercise-6

Implement following Pattern Matching Algorithms.

12. KMP Algorithm

```
#include <stdio.h>
#include <string.h>
void computeLPS(char *pattern, int M, int *lps) {
 int len = 0;
 lps[0] = 0;
 int i = 1;
 while (i < M) {
 if (pattern[i] == pattern[len]) {
 len++;
 lps[i] = len;
 i++;
 } else {
 if (len != 0) {
 len = lps[len - 1];
 } else {
 lps[i] = 0;
 i++;
 }
 }
 }
}
void KMPSearch(char *text, char *pattern) {
 int N = strlen(text);
 int M = strlen(pattern);
 int lps[M];
 computeLPS(pattern, M, lps);
 int i = 0;
 int j = 0;
 while (i < N) {
 if (pattern[j] == text[i]) {
 i++;
 j++;
```

```
 }
 if (j == M) {
 printf("Pattern found at index %d\n", i - j);
 j = lps[j - 1];
 } else if (i < N && pattern[j] != text[i]) {
 if (j != 0)
 j = lps[j - 1];
 else
 i++;
 }
 }
}
int main() {
 char text[] = "ABABDABACCAABDCDDEEPUABABCABAB";
 char pattern[] = "DEEPU";
 printf("Text: %s\n", text);
 printf("Pattern: %s\n", pattern);
 printf("Pattern matching using KMP algorithm:\n");
 KMPSearch(text, pattern);
 return 0;
}
```

Output:

```
Output
/tmp/COoII2Bi7G.o
Text: ABABDABACCAABDCDDEEPUABABCABAB
Pattern: DEEPU
Pattern matching using KMP algorithm:
Pattern found at index 16


=== Code Execution Successful ===
```

13. RK Algorithm

```
#include <stdio.h>
#include <string.h>
#define d 256
#define q 101
void RabinKarpSearch(char *text, char *pattern) {
 int M = strlen(pattern);
 int N = strlen(text);
 int i, j;
 int p = 0;
 int t = 0;
 int h = 1;
 for (i = 0; i < M - 1; i++)
 h = (h * d) % q;
 for (i = 0; i < M; i++) {
 p = (d * p + pattern[i]) % q;
 t = (d * t + text[i]) % q;
 }
 for (i = 0; i <= N - M; i++) {
 if (p == t) {
 for (j = 0; j < M; j++) {
 if (text[i + j] != pattern[j])
 break;
 }
 if (j == M)
 printf("Pattern found at index %d\n", i);
 }
 if (i < N - M) {
 t = (d * (t - text[i] * h) + text[i + M]) % q;
 if (t < 0)
 t = (t + q);
 }
 }
}
int main() {
```

```
 char text[] = "ABAAABDCDLUFFYABAABAB";
 char pattern[] = "LUFFY";
 printf("Text: %s\n", text);
 printf("Pattern: %s\n", pattern);
 printf("Pattern matching using Rabin-Karp algorithm:\n");
 RabinKarpSearch(text, pattern);
 return 0;
}
```

Output:

```
Output

/tmp/GibZiPd67F.o
Text: ABAAABDCDLUFFYABAABAB
Pattern: LUFFY
Pattern matching using Rabin-Karp algorithm:
Pattern found at index 9


=== Code Execution Successful ===
```