1.hashing:-

#include <stdlib.h>

#include <iostream>

using namespace std;

class HashTable{

private:

int size;

int \*H;

public:

HashTable(int size){

this->size = size;

H = (int \*)calloc(size, sizeof(int));

}

int hashF(int key){

return key % size;

}

void insert(int key){

int i = hashF(key);

H[i] = key;

display();

}

void dele(int key){

int i = search(key);

if (i != -1)

H[i] = 0;

else

cout << "Element not found can't be deleted" << endl;

display();

}

int search(int key){

int i = hashF(key);

if (H[i] == key)

return i;

return -1;

}

void display(){

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

{

cout << "[ " << i << " : " << H[i] << " ]" << endl;

}

}

};

int main(){

int size;

cout<<"Enter the size of Hash Table" <<endl;

cin >> size;

HashTable myTable(size);

cout << "1.Insert \n 2. Remove \n 3. Search \n 4. Any other to Quit" << endl;

while (1){

cout<< "Enter your choice : ";

int choice;

cin >> choice;

switch (choice)

{

case 1:

int ele;

cout<< "Enter an element to insert : ";

cin >> ele;

myTable.insert(ele);

break;

case 2:

cout<< "Enter an element to delete : ";

cin >> ele;

myTable.dele(ele);

break;

case 3:

cout<< "Enter an element to search : ";

cin >> ele;

if (myTable.search(ele) != -1)

cout << "Element Found" << endl;

else

cout << "Element Not Found" <<endl;

break;

case 4:

exit(0);

}

}

return 0;

}

2.linear probing:

#include<stdlib.h>

#include<iostream>

using namespace std;

class HashTable

{

private:

int size;

int \*H;

public:

HashTable(int size){

this->size = size;

H = (int \*)calloc(size,sizeof(int));

}

int hashF(int key){

return key % size;

}

void insert(int key){

int i = hashF(key);

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

int k = (i+j) % size;

if(H[k]==0){

H[k] = key;

break;

}

}

display();

}

void dele(int key){

int i = search(key);

if(i!=-1)

H[i]=0;

else

cout<<"Element not present in Hash Table, cannot be deleted" << endl;

}

int search(int key){

int i = hashF(key);

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

int k = (i+j) % size;

if(H[k]==key)

return k;

}

return -1;

}

void display(){

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

cout<< "[ "<< i <<" : "<< H[i] <<" ]"<<endl;

}

}

};

int main(int argc, char const \*argv[])

{

int size;

cout<<"Enter the size of Hash Table" <<endl;

cin>>size;

HashTable myTable(size);

cout << "1.Insert \n 2. Remove \n 3. Search \n 4. Any other to Quit" << endl;

while (1)

{

int choice;

cin >> choice;

switch (choice)

{

case 1:

int ele;

cin >> ele;

myTable.insert(ele);

break;

case 2:

cin >> ele;

myTable.dele(ele);

break;

case 3:

cin >> ele;

if (myTable.search(ele) != -1)

cout << "Found" << endl;

else

cout << "Not Found <<endl";

break;

default:

exit(0);

}

}

return 0;

}

3.quadratic probing:-

#include<stdlib.h>

#include<iostream>

using namespace std;

class HashTable

{

private:

int size;

int \*H;

public:

HashTable(int size){

this->size = size;

H = (int \*)calloc(size,sizeof(int));

}

int hashF(int key){

return key % size;

}

void insert(int key){

int i = hashF(key);

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

int k = (i+(j\*j)) % size;

if(H[k]==0){

H[k] = key;

break;

}

}

display();

}

void dele(int key){

int i = search(key);

if(i!=-1)

H[i]=0;

else

cout<<"Element not present in Hash Table, cannot be deleted" << endl;

}

int search(int key){

int i = hashF(key);

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

int k = (i+(j\*j)) % size;

if(H[k]==key)

return k;

}

return -1;

}

void display(){

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

cout<< "[ "<< i <<" : "<< H[i] <<" ]"<<endl;

}

}

};

int main(int argc, char const \*argv[])

{

int size;

cout<<"Enter the size of Hash Table" <<endl;

cin>>size;

HashTable myTable(size);

cout << "1.Insert \n 2. Remove \n 3. Search \n 4. Any other to Quit" << endl;

while (1)

{

int choice;

cin >> choice;

switch (choice)

{

case 1:

int ele;

cin >> ele;

myTable.insert(ele);

break;

case 2:

cin >> ele;

myTable.dele(ele);

break;

case 3:

cin >> ele;

if (myTable.search(ele) != -1)

cout << "Found" << endl;

else

cout << "Not Found <<endl";

break;

default:

exit(0);

}

}

return 0;

}

4.double hashing:-

#include <iostream>

#include<cmath>

using namespace std;

class HashTable

{

private:

int size;

int \*H;

public:

HashTable(int size)

{

this->size = size;

H = (int \*)calloc(size, sizeof(int));

}

int hashFunction1(int key)

{

return key % size;

}

bool isPrime(int x)

{

for (int i = 2; i < (int)sqrt(x); i++)

{

if (x % i == 0)

return false;

}

return true;

}

int nearestPrime(int n)

{

int p = 1;

for (int i = 2; i < n; i++)

{

if (isPrime(i))

p = i;

}

return p;

}

int hashFunction2(int key)

{

int mPrime = nearestPrime(key);

return (mPrime - (key % mPrime));

}

void insert(int key)

{

int i = hashFunction1(key);

for (int j = 0; j < size; j++)

{

int k = (i + (j \* hashFunction2(key))) % size;

if (H[k] == 0)

{

H[k] = key;

break;

}

}

display();

}

void dele(int key)

{

int i = search(key);

if (i != -1)

H[i] = 0;

else

cout << "Element not present in Hash Table, cannot be deleted" << endl;

}

int search(int key)

{

int i = hashFunction1(key);

for (int j = 0; j < size; j++)

{

int k = (i + (j \* hashFunction2(key))) % size;

if (H[k] == key)

return k;

}

return -1;

}

void display()

{

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

{

cout << "[ " << i << " : " << H[i] << " ]" << endl;

}

}

};

int main()

{

int size;

cout << "Enter the size of Hash Table" << endl;

cin >> size;

HashTable myTable(size);

cout << "1.Insert \n 2. Remove \n 3. Search \n 4. Any other to Quit" << endl;

while (1)

{

int choice;

cin >> choice;

switch (choice)

{

case 1:

int ele;

cin >> ele;

myTable.insert(ele);

break;

case 2:

cin >> ele;

myTable.dele(ele);

break;

case 3:

cin >> ele;

if (myTable.search(ele) != -1)

cout << "Found" << endl;

else

cout << "Not Found <<endl";

break;

default:

exit(0);

}

}

return 0;

}

5.separate chaining:-

#include <stdlib.h>

#include <iostream>

using namespace std;

struct node

{

struct node \*prev;

int data;

struct node \*next;

};

typedef struct node Node;

class DoubleLL

{

private:

Node \*head;

Node \*tail;

int size;

public:

DoubleLL()

{

head = tail = NULL;

size = 0;

}

void display()

{

Node \*temp = head;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

void insert(int val)

{

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

newNode->data = val;

newNode->prev = newNode->next = NULL;

if (head == NULL)

{

head = tail = newNode;

}

else

{

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

size++;

}

// This function will return the address of the element in list or else NULL

Node \*search(int value)

{

Node \*temp = head;

for (size\_t i = 0; i < size; i++)

{

if (temp->data == value)

return temp;

temp = temp->next;

}

return NULL;

}

void del(int value)

{

Node \*t = search(value);

//cout << t << endl;

if (t != NULL)

{

Node \*prevNode = t->prev;

Node \*nextNode = t->next;

//Node \* nextNode = t->next->next;

if (nextNode != NULL)

nextNode->prev = prevNode;

else{

tail = prevNode;

}

if (prevNode != NULL)

prevNode->next = nextNode;

else{

head = nextNode;

}

free(t);

}

}

};

void display(DoubleLL list[],int size){

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

cout<< i << "-->";

list[i].display();

}

}

int main(int argc, char const \*argv[])

{

int size;

cout<<"Enter the size of the HashTable"<<endl;

cin >> size;

DoubleLL list[size];

cout << "1.Insert \n 2. Remove \n 3. Search \n 4. Any other to Quit" << endl;

while (1)

{

int choice;

cout<<"Choose from menu: ";

cin >> choice;

int element;

cout<<"Enter the Key"<<endl;

cin >> element;

int hashIndex = element % size;

switch (choice)

{

case 1:

list[hashIndex].insert(element);

display(list,size);

break;

case 2:

list[hashIndex].del(element);

display(list,size);

break;

case 3:

if (list[hashIndex].search(element) != NULL)

cout << "Found" << endl;

else

cout << "Not Found <<endl";

break;

default:

exit(0);

}

}

return 0;

}

6.heap operation:-

#define MAX 10

#include <iostream>

using namespace std;

class BinaryHeapADT

{

public:

void heapSort(int a[MAX], int n);

void buildHeap(int a[MAX],int,int);

};

void BinaryHeapADT::heapSort(int a[MAX], int n)

{

int i;

int temp,k,z=0;

for(i=(n/2);i>0;i--)

buildHeap(a,i,n);

for(i=n;i>=1;i--)

{

temp = a[1];

a[1] = a[i];

a[i] = temp;

buildHeap(a,1,i-1);

}

}

void BinaryHeapADT::buildHeap(int a[MAX],int root,int bottom)

{

int temp,maxc;

int done = 0;

while((root\*2 <= bottom) && (!done))

{

if(root\*2==bottom)

maxc = root\*2;

else

if(a[root\*2] > a[root\*2+1])

maxc = root\*2;

else

maxc = root\*2+1;

if(a[root] < a[maxc])

{

temp = a[root];

a[root] = a[maxc];

a[maxc] = temp;

root = maxc;

}

else

done = 1;

}

}

int main()

{

int a[7]={0,33,66,22,88,55,44};

BinaryHeapADT b;

b.heapSort(a,7);

for(int i=1;i<=7;i++)

cout<<a[i]<<" ";

return 0;

}

7.binary search non rec:-

#define MAX 20

#include<stdlib.h>

#include<iostream>

using namespace std;

struct Node

{

int data;

struct Node \*left,\*right;

};

class BST\_NR

{

public:

Node \* insert(Node \*,int);

void inorder\_nr(Node \*);

void preorder\_nr(Node \*);

void postorder\_nr(Node \*);

};

Node \* BST\_NR :: insert(Node \*t,int x)

{

if(t==NULL)

{

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

t->data = x;

t->left= t->right = NULL;

return t;

}

else

if(x < t->data)

{

t->left = insert(t->left,x);

}

else

if(x > t->data)

t->right= insert(t->right,x);

else

cout<<"\n Data already exists...!";

return t;

}

void BST\_NR :: inorder\_nr(Node \*t)

{

Node \*stk[MAX];

Node \*temp;

int top = 0;

temp = t; // Assign root (t) to temp

while(1)

{

while(temp!=NULL)

{

stk[++top] = temp;

temp = temp->left;

}

if(top>0)

{

temp = stk[top--];

cout<<temp->data<<" ";

temp = temp->right;

}

else

return;

}

}

void BST\_NR :: preorder\_nr(Node \*t)

{

Node \*stk[MAX];

int top = 0;

Node \*temp;

temp = t; // assign root to temp

while(1)

{

while(temp!=NULL) // wil be continued till no further left node.

{

cout<<temp->data<<" ";

stk[++top] = temp;

temp = temp->left;

}

if(top>0)

{

temp = stk[top--];

if(temp->right!=NULL)

temp = temp->right;

else

temp = NULL;

}

else

return;

}

}

void BST\_NR :: postorder\_nr(Node \*t)

{

Node \*temp;

Node \*stk1[MAX], \*stk2[MAX];

int top1, top2;

top1 = top2 = 0;

temp = t; //Assign root to temp;

stk1[++top1] = temp;

while(top1>0)

{

temp = stk1[top1--];

stk2[++top2] = temp;

if(temp->left!=NULL)

stk1[++top1] = temp->left;

if(temp->right!=NULL)

stk1[++top1] = temp->right;

}

while(top2>0)

{

temp = stk2[top2--];

cout<<temp->data<<" ";

}

}

int main()

{

BST\_NR obj;

int a[8]= { 4,5,7,6,3,2,9,1};

Node \*root = NULL;

for(int i=0;i<8;i++)

root = obj.insert(root,a[i]);

cout<<"\n The non recursive Inorder is...\n\n";

obj.inorder\_nr(root);

cout<<"\n\n The non recursive Preorder is...\n\n";

obj.preorder\_nr(root);

cout<<"\n\n The non recursive Postorder is...\n\n";

obj.postorder\_nr(root);

return 0;

}

8.avl tree:-

#include<stdlib.h>

#include <iostream>

using namespace std;

struct Node

{

int data;

int height;

struct Node \*left, \*right;

};

class AVLTreeADT

{

public:

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

struct Node \* remove(struct Node \*, int);

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

struct Node \* LL(struct Node \*);

struct Node \* RR(struct Node \*);

struct Node \* LR(struct Node \*);

struct Node \* RL(struct Node \*);

int height(struct Node \*);

void printTree(struct Node \*,int);

void inorder(struct Node \*);

void preorder(struct Node \*);

void postorder(struct Node \*);

int big(int,int);

struct Node \* findMin(struct Node \*);

struct Node \* findMax(struct Node \*);

};

int AVLTreeADT::big(int x,int y)

{

return (x>y)?x:y;

}

struct Node \* AVLTreeADT::insert(struct Node \*root, int value)

{

if(root==NULL)

{

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

root->data=value;

root->height=0;

root->left=NULL;

root->right=NULL;

return root;

}

else

if(root->data<value)

{

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

if((height(root->right) - height(root->left))>=2)

{

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

root = RR(root);

else

root = RL(root);

}

}

else

if(root->data>value)

{

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

if((height(root->left) - height(root->right))>=2)

{

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

root = LL(root);

else

root = LR(root);

}

}

else

{

cout<<"\n Data already exists..discard it !";

return root;

}

root->height = 1+big(height(root->left),height(root->right));

return root;

}

struct Node \* AVLTreeADT::LL(struct Node \*t)

{

struct Node \*temp;

temp = t->left;

t->left = temp->right;

temp->right = t;

t->height = 1+big(height(t->left),height(t->right));

temp->height = 1+big(height(temp->left),height(temp->right));

return temp;

}

struct Node \* AVLTreeADT::RR(struct Node \*t)

{

struct Node \*temp;

temp = t->right;

t->right = temp->left;

temp->left = t;

t->height = 1+big(height(t->left),height(t->right));

temp->height = 1+big(height(temp->left),height(temp->right));

return temp;

}

struct Node \* AVLTreeADT::LR(struct Node \*t)

{

t->left = RR(t->left);

return LL(t);

}

struct Node \* AVLTreeADT::RL(struct Node \*t)

{

t->right = LL(t->right);

return RR(t);

}

int AVLTreeADT::height(struct Node \*t)

{

if(t==NULL)

return -1;

else

if(t->left == NULL && t->right == NULL)

return 0;

else

if(t->left == NULL)

return 1+height(t->right);

else

if(t->right == NULL)

return 1+height(t->left);

else

return 1+big(height(t->left),height(t->right));

}

void AVLTreeADT::printTree(struct Node \*t,int level)

{

if(t!=NULL)

{

printTree(t->right, level+1);

for(int i=0;i<level;i++)

cout<<" ";

cout<<t->data<<"\n";

printTree(t->left, level+1);

}

}

void AVLTreeADT::inorder(struct Node \*t)

{

if(t!=NULL)

{

inorder(t->left);

cout<<t->data<<" ";

inorder(t->right);

}

}

struct Node \*AVLTreeADT :: remove(struct Node \*t, int k)

{

Node \*tmp;

if(t==NULL)

return t;

if(k < t->data){

t->left = remove(t->left,k);

if (height(t->right) - height(t->left) >= 2){

if (t->right->right)

t = RR(t);

else

t = RL(t);

}

}

else{

if(k > t->data){

t->right = remove(t->right,k);

if (height(t->left) - height(t->right) >= 2){

if (t->left->left)

t = LL(t);

else

t = LR(t);

}

}

else if (t->left && t->right){

tmp = findMin(t->right);

t->data = tmp->data;

t->right = remove(t->right,t->data);

tmp = NULL;

}

else{

tmp = t;

if (t->left == NULL)

t = t->right;

else if (t->right == NULL)

t = t->left;

delete tmp;

tmp = NULL;

}

}

//t->height = 1+big(height(t->left),height(t->right));

return t;

}

struct Node \* AVLTreeADT :: search(struct Node \*t, int data)

{

if(t==NULL)

return NULL;

else

if(t->left==NULL && t->right==NULL)

{

if(t->data == data)

return t;

return NULL;

}

else

if(t->data > data)

return search(t->left, data);

else

if(t->data < data)

return search(t->right, data);

else

return t;

}

void AVLTreeADT :: preorder(struct Node \*t)

{

if(t!=NULL)

{

cout<<t->data<<" ";

preorder(t->left);

preorder(t->right);

}

}

void AVLTreeADT :: postorder(struct Node \*t)

{

if(t!=NULL)

{

postorder(t->left);

postorder(t->right);

cout<<t->data<<" ";

}

}

struct Node \* AVLTreeADT :: findMin(struct Node \*t)

{

if(t==NULL)

return NULL;

else

if(t->left==NULL && t->right==NULL)

return t;

else

if(t->left == NULL)

return t;

else

return findMin(t->left);

}

struct Node \* AVLTreeADT :: findMax(struct Node \*t)

{

if(t==NULL)

return NULL;

else

if(t->left==NULL && t->right==NULL)

return t;

else

if(t->right == NULL)

return t;

else

return findMax(t->right);

}

int main()

{

struct Node \*root = NULL;

AVLTreeADT obj;

//int v;

int a[10] = {11,76,4,89,28,2,7,8,9,5};

for(int i=0;i<10;i++)

{

//cout<<"enter value";

//cin>>v;

root = obj.insert(root, a[i]);

obj.printTree(root,1);

cout<<"...................\n";

}

cout<<"\ninorder:\n";

obj.inorder(root);

cout<<"\npreorder:\n";

obj.preorder(root);

cout<<"\npostorder:\n";

obj.postorder(root);

cout<<"\nThe AVL tree structure is......\n";

obj.printTree(root,1);

struct Node \*temp = obj.search(root,89);

if(temp==NULL)

cout<<"element not found";

else

cout<<"element found";

temp = obj.findMin(root);

if(temp == NULL)

cout<<" No tree at all";

else

{

cout<<"\nThe min element in the AVL Tree : ";

cout<<temp->data<<"\n";

}

temp = obj.findMax(root);

if(temp == NULL)

cout<<" No tree at all";

else

{

cout<<"\nThe max element in the AVL Tree : ";

cout<<temp->data<<"\n";

}

cout<<"height : "<<obj.height(root)<<"\n";

struct Node \*tmp = obj.remove(root,5);

obj.inorder(root);

return 0;

}

9.prims:-

#include <iostream>

using namespace std;

int main()

{

int a, b, u, v, n, i, j, ne = 1, x;

int visited[10] = {0}, min, mincost = 0, cost[10][10];

cout << "\n Enter the number of nodes:" << endl;

cin >> n;

cout <<"\n Enter the adjacency matrix:\n";

for (i = 1; i <= n; i++)

for (j = 1; j <= n; j++)

cin >> cost[i][j];

cout <<"enter the starting vertex";

cin >> x;

visited[x] = 1;

cout << endl;

while (ne < n)

{

min = 999;

for (i = 1; i <= n; i++)

{

for (j = 1; j <= n; j++)

{

if (cost[i][j] && ((visited[i] && !visited[j]) || (visited[j] && !visited[i])))

{

if (cost[i][j] < min)

{

min = cost[i][j];

a = i;

b = j;

}

}

}

}

cout << ne++ << " edge (" << a << "," << b << " =" << min << " )" << endl;

mincost += min;

visited[b] = visited[a] = 1;

}

cout << "\n Minimun cost=" << mincost << endl;

return 0;

}

10.krushkals:-

#include <iostream>

using namespace std;

int find(int);

int uni(int, int);

int parent[9];

int main()

{

int i, j, k, a, b, u, v, n, ne = 1;

int min, mincost = 0, cost[9][9];

cout << "\n\n\tImplementation of Kruskal's algorithm\n\n";

cout << "\nEnter the no. of vertices\n";

cin >> n;

cout << "\nEnter the cost adjacency matrix\n";

for (i = 1; i <= n; i++)

{

for (j = 1; j <= n; j++)

{

cin >> cost[i][j];

if (cost[i][j] == 0)

cost[i][j] = 999;

}

}

cout << "\nThe edges of Minimum Cost Spanning Tree are\n\n";

while (ne < n)

{

for (i = 1, min = 999; i <= n; i++)

{

for (j = 1; j <= n; j++)

{

if (cost[i][j] < min)

{

min = cost[i][j];

a = u = i;

b = v = j;

}

}

}

u = find(u);

v = find(v);

if (uni(u, v))

{

cout << "\n"

<< ne++ << " edge (" << a << "," << b << " =" << min << " )" << endl;

mincost += min;

}

cost[a][b] = cost[b][a] = 999;

}

cout << "\n\tMinimum cost = " << mincost << endl;

return 0;

}

int find(int i)

{

while (parent[i])

i = parent[i];

return i;

}

int uni(int i, int j)

{

if (i != j)

{

parent[j] = i;

return 1;

}

return 0;

}

11.dijkstras:-

#include <iostream>

using namespace std;

#define infinity 999

void dij(int n, int v, int cost[10][10], int dist[])

{

int i, u, count, w, flag[10], min;

for (i = 1; i <= n; i++)

flag[i] = 0, dist[i] = cost[v][i];

count = 2;

while (count <= n)

{

min = 99;

for (w = 1; w <= n; w++)

if (dist[w] < min && !flag[w])

min = dist[w], u = w;

flag[u] = 1;

count++;

for (w = 1; w <= n; w++)

if ((dist[u] + cost[u][w] < dist[w]) && !flag[w])

dist[w] = dist[u] + cost[u][w];

}

}

int main()

{

int n, v, i, j, cost[10][10], dist[10];

cout <<"\n Enter the number of nodes:"<<endl;

cin >> n;

cout <<"\n Enter the cost matrix:"<<endl;

for (i = 1; i <= n; i++)

for (j = 1; j <= n; j++)

{

cin >> cost[i][j];

if (cost[i][j] == 0)

cost[i][j] = infinity;

}

cout <<" Enter the source vertex:"<<endl;

cin >> v;

dij(n, v, cost, dist);

cout <<"Shortest path:\n";

for (i = 1; i <= n; i++)

if (i != v)

cout << " " << v << "=>" << i << " cost= " << dist[i] << endl;

return 0;

}

12.bruteforce:-

#include<string.h>

#include<iostream>

using namespace std;

class PatternMatcher

{

public:

int bruteforce(char text[40], char pattern[10]);

};

int PatternMatcher :: bruteforce(char text[40], char pattern[10])

{

int m,n,i,j;

n = strlen(text);

m = strlen(pattern);

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

{

for(j=0;j<m;j++)

{

if(text[i+j]!=pattern[j])

break;

}

if(j==m)

return i;

}

return -1;

}

int main()

{

char text[40], pat[10];

PatternMatcher obj;

cout<<"Enter main text:";

cin>>text;

cout<<"Enter pattern/substring:";

cin>>pat;

int pos = obj.bruteforce(text,pat);

if(pos==-1)

cout<<"Sorry... No such pattern found";

else

cout<<"Pattern found at "<<pos<<" in the main string...";

return 0;

}

13.booyere:-

#include<string.h>

#include<iostream>

using namespace std;

int find(char pattern[ ], char ch)

{

int m, i; m = strlen(pattern);

for (i = m - 2; i >= 0; i--)

{

if (ch == pattern[i])

{

return i;

}

}

return -1;

}

int BoyerMoore(char text[ ], char pattern[ ])

{

int n, m, i, j, lastch;

n = strlen(text);

m = strlen(pattern);

i=m-1; j=m-1;

while (i < n) // not end of string S

if (pattern[j] == text[i])

if (j == 0) // first char of pattern

return i;

else

{

j--; i--; // go left

}

else // no match – find char in pattern

{

lastch = find(pattern, text[i]);

if (lastch == -1) // not found

i = i + m; // jump over

else

i = i + j-lastch; // align char

j = m - 1; // restart from right

}

return -1; // not matched

}

int main()

{

char text[] = "This is demo program";

char pattern[] = "gram";

int loc = BoyerMoore(text, pattern);

if(loc==-1)

cout<<"Sorry...! No pattern found..!";

else

cout<<"The pattern starts from location "<<loc<<" in main text..!";

return 0;

}

14.kmp:-

#include<iostream>

using namespace std;

// Calculating the failure function & place it in the prefix & suffix table

void findPrefix(char pattern[40], int m, int prefArray[40])

{

int length = 0;

prefArray[0] = 0;

for(int i = 1; i<m; i++)

{

if(pattern[i] == pattern[length])

{

length++;

prefArray[i] = length;

}

else {

if(length != 0)

{

length = prefArray[length - 1];

i--;

}

else

prefArray[i] = 0;

}

}

}

void kmp(char mainString[40], char pattern[20]) {

int n, m, i = 0, j = 0;

for(n=0;mainString[n]!='\0';n++); // to find length of mainstring

for(m=0;pattern[m]!='\0';m++); // to find the length of the pattern

int prefixArray[m]; // precomputed table with the values to be skipped

findPrefix(pattern, m, prefixArray); // preparing failure function & table

cout<<"\n The precomputed table is..\n ";

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

cout<<prefixArray[i]<<" ";

cout<<"\n";

while(i < n)

{

if(mainString[i] == pattern[j])

{

i++; j++;

}

if(j == m)

{

cout<<"Pattern found at "<<i-j<<"\n"; // Here i-j is the starting position of the pattern in the mainstring

return;

}

else

if((i < n) && (pattern[j] != mainString[i])) // if mismatch occurs at jth position

{

if(j != 0)

j = prefixArray[j-1];// here we consider the number of positions to be skipped

else

i++;

}

}

cout<<" Sorry...! Pattern not found...!";

}

int main() {

char str[50] = "aaaabbaaabaabaabbbaaaab";

char patt[20] = "aabaa";

kmp(str, patt);

}