1.//BubbleSort

// Optimized implementation of Bubble sort

#include<iostream>

#include <stdio.h>

using namespace std;

void swap(int \*xp, int \*yp)

{

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

// An optimized version of Bubble Sort

void bubbleSort(int arr[], int n)

{

int i, j;

bool swapped;

for (i = 0; i < n-1; i++)

{

swapped = false;

for (j = 0; j < n-i-1; j++)

{

if (arr[j] > arr[j+1])

{

swap(&arr[j], &arr[j+1]);

swapped = true;

}

}

// IF no two elements were swapped by inner loop, then break

if (swapped == false)

break;

}

}

/\*Counting no of comparisons

void bubbleSort(int arr[], int n)

{

bool sorted = false;

for(int pass = 1; pass < n.size() && !sorted; pass++)

{

sorted = true;

int i;

for(i = 0; i < n.size() - pass; i++)

{

if(\*n[i + 1] < \*n[i])

{

swap(n, i, i + 1);

sorted = false;

}

}

CountbubbleSort += i;

}

cout<<"bubbleSort comparison is "<<CountbubbleSort<<endl;

}\*/

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

cout<<"[";

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

cout<<arr[i]<<"][";

}

// Driver program to test above functions

int main()

{

int arr[100];

int n;

cout<<"Enter the number of elements in array: ";

cin>>n;

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

cout<<"Enter the number-"<<i+1<<": ";

cin>>arr[i];

}

bubbleSort(arr, n);

cout<<"\n\nSorted array: \n";

printArray(arr, n);

return 0;

}

2.//Insertion Sort

// C++ program for insertion sort

#include<iostream>

#include <math.h>

#include <stdio.h>

using namespace std;

/\* Function to sort an array using insertion sort\*/

void insertionSort(int arr[], int n)

{

int i, key, j;

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

key = arr[i];

j = i - 1;

/\* Move elements of arr[0..i-1], that are

greater than key, to one position ahead

of their current position \*/

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

// A utility function to print an array of size n

void printArray(int arr[], int n)

{

int i;

cout<<"\n\nSorted array is:\n";

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

cout<<"["<<arr[i]<<"]";

cout<<"\n";

}

/\* Driver program to test insertion sort \*/

int main()

{

int arr[100];

int n;

cout<<"Enter numbers in array: ";

cin>>n;

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

cout<<"Enter element-"<<i+1<<": ";

cin>>arr[i];

}

insertionSort(arr, n);

printArray(arr, n);

return 0;

}

3.//Selection Sort

#include<iostream>

using namespace std;

//swap the content of a and b

void swapping(int &a, int &b){

int temp;

temp = a;

a = b;

b = temp;

}

void display(int \*array, int size){

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

cout << array[i] << " ";

cout << endl;

}

void selectionSort(int \*array, int size){

int i, j, imin;

for(i = 0; i<size-1; i++) {

imin = i; //get index of minimum data

for(j = i+1; j<size; j++)

if(array[j] < array[imin])

imin = j;

//placing in correct position

swap(array[i], array[imin]);

}

}

int main(){

int n;

cout << "Enter the number of elements: ";

cin >> n;

int arr[n]; //create an array with given number of elements

cout << "Enter elements:" << endl;

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

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

selectionSort(arr, n);

cout << "Array after Sorting: ";

display(arr, n);

}

4.//Merge Sort

/\* C program for Merge Sort \*/

#include<iostream>

#include<stdlib.h>

#include<stdio.h>

using namespace std;

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy the remaining elements of L[], if there

are any \*/

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

/\* Copy the remaining elements of R[], if there

are any \*/

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

/\* l is for left index and r is right index of the

sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

// Same as (l+r)/2, but avoids overflow for

// large l and h

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

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int arr[], int n)

{

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

cout<<"["<<arr[i]<<"]";

cout<<"\n";

}

/\* Driver program to test above functions \*/

int main()

{

int arr[100];

int n;

cout<<"Given array should have elements: ";

cin>>n;

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

cout<<"Provide element-"<<i+1<<": ";

cin>>arr[i];

}

mergeSort(arr, 0, n - 1);

printf("\nSorted array is \n");

printArray(arr, n);

return 0;

}

5.//Quick Sort

/\* C implementation QuickSort \*/

#include<iostream>

#include<stdio.h>

using namespace std;

// A utility function to swap two elements

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

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

{

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

/\* The main function that implements QuickSort

arr[] --> Array to be sorted,

low --> Starting index,

high --> Ending index \*/

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

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

// Separately sort elements before

// partition and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

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

cout<<"["<<arr[i]<<"]";

}

// Driver program to test above functions

int main()

{

int arr[100];

int n;

cout<<"Numbers in array should be: ";

cin>>n;

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

cout<<"Enter element-"<<i+1<<": ";

cin>>arr[i];

}

quickSort(arr, 0, n-1);

cout<<"\n\nSorted array:\n";

printArray(arr, n);

return 0;

}

6.//Heap Sort

// C++ program for implementation of Heap Sort

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

void heapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i) {

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// main function to do heap sort

void heapSort(int arr[], int n)

{

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// One by one extract an element from heap

for (int i = n - 1; i >= 0; i--) {

// Move current root to end

swap(arr[0], arr[i]);

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

/\* A utility function to print array of size n \*/

void printArray(int arr[], int n)

{

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

cout<<"["<<arr[i] <<"]";

cout << "\n";

}

// Driver program

int main()

{

int arr[100];

int n;

cout<<"Array should have the number in: ";

cin>>n;

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

cout<<"Provide element-"<<i+1<<": ";

cin>>arr[i];

}

heapSort(arr, n);

cout << "\nSorted array is \n";

printArray(arr, n);

}

7.//Radix Sort

// C++ implementation of Radix Sort

#include<iostream>

using namespace std;

// A utility function to get maximum value in arr[]

int getMax(int arr[], int n)

{

int mx = arr[0];

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

if (arr[i] > mx)

mx = arr[i];

return mx;

}

// A function to do counting sort of arr[] according to

// the digit represented by exp.

void countSort(int arr[], int n, int exp)

{

int output[n]; // output array

int i, count[10] = {0};

// Store count of occurrences in count[]

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

count[ (arr[i]/exp)%10 ]++;

// Change count[i] so that count[i] now contains actual

// position of this digit in output[]

for (i = 1; i < 10; i++)

count[i] += count[i - 1];

// Build the output array

for (i = n - 1; i >= 0; i--)

{

output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];

count[ (arr[i]/exp)%10 ]--;

}

// Copy the output array to arr[], so that arr[] now

// contains sorted numbers according to current digit

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

arr[i] = output[i];

}

// The main function to that sorts arr[] of size n using

// Radix Sort

void radixsort(int arr[], int n)

{

// Find the maximum number to know number of digits

int m = getMax(arr, n);

// Do counting sort for every digit. Note that instead

// of passing digit number, exp is passed. exp is 10^i

// where i is current digit number

for (int exp = 1; m/exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

// A utility function to print an array

void print(int arr[], int n)

{

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

cout<<"["<<arr[i]<<"]";

cout<<"\n";

}

// Driver program to test above functions

int main()

{

int arr[100];

int n;

cout<<"Array should encounter the numbers: ";

cin>>n;

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

cout<<"Counter element-"<<i+1<<": ";

cin>>arr[i];

}

radixsort(arr, n);

cout<<"\nSorted array is:\n";

print(arr, n);

return 0;

}

8.//DFT

// C++ program to print DFS traversal from

// a given vertex in a given graph

#include<iostream>

#include<list>

using namespace std;

// Graph class represents a directed graph

// using adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing

// adjacency lists

list<int> \*adj;

// A recursive function used by DFS

void DFSUtil(int v, bool visited[]);

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to vs list.

}

void Graph::DFSUtil(int v, bool visited[])

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFSUtil(\*i, visited);

}

// DFS traversal of the vertices reachable from v.

// It uses recursive DFSUtil()

void Graph::DFS(int v)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Call the recursive helper function

// to print DFS traversal

DFSUtil(v, visited);

}

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Depth First Traversal\n"

" (starting from vertex 2) \n";

g.DFS(2);

return 0;

}

9.//BFT

// Program to print BFS traversal from a given

// source vertex. BFS(int s) traverses vertices

// reachable from s.

#include<iostream>

#include<list>

using namespace std;

// This class represents a directed graph using

// adjacency list representation

class Graph

{

int V; // No. of vertices

// Pointer to an array containing adjacency

// lists

list<int> \*adj;

public:

Graph(int V); // Constructor

// function to add an edge to graph

void addEdge(int v, int w);

// prints BFS traversal from a given source s

void BFS(int s);

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to vs list.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool \*visited = new bool[V];

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

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent

// vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i)

{

if (!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

// Driver program to test methods of graph class

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal\n"

<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}Tutorialspoint

Search your favorite tutorials...

Counting Sort

Sorting AlgorithmAlgorithmsData Structure

Follow Answer 4043

1 Answer

Samual Sam

Samual Sam

Answered on 6th Jul, 2018

Counting sort is a stable sorting technique, which is used to sort objects according to the keys that are small numbers. It counts the number of keys whose key values are same. This sorting technique is effective when the difference between different keys are not so big, otherwise, it can increase the space complexity.

The complexity of counting Sort Technique

Time Complexity: O(n+r)

Space Complexity: O(n+r)

Input and Output

Input:

A list of unsorted data: 2 5 6 2 3 10 3 6 7 8

Output:

Array before Sorting: 2 5 6 2 3 10 3 6 7 8

Array after Sorting: 2 2 3 3 5 6 6 7 8 10

Algorithm

counting sort(array, size)

Input: An array of data, and the total number in the array

Output: The sorted Array

Begin

max = get maximum element from array.

define count array of size [max+1]

for i := 0 to max do

count[i] = 0 //set all elements in the count array to 0

done

for i := 1 to size do

increase count of each number which have found in the array

done

for i := 1 to max do

count[i] = count[i] + count[i+1] //find cumulative frequency

done

for i := size to 1 decrease by 1 do

store the number in the output array

decrease count[i]

done

return the output array

End

Source Code (C++)

#include<iostream>

#include<algorithm>

using namespace std;

void display(int \*array, int size) {

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

cout << array[i] << " ";

cout << endl;

}

int getMax(int array[], int size) {

int max = array[1];

for(int i = 2; i<=size; i++) {

if(array[i] > max)

max = array[i];

}

return max; //the max element from the array

}

void countSort(int \*array, int size) {

int output[size+1];

int max = getMax(array, size);

int count[max+1]; //create count array (max+1 number of elements)

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

count[i] = 0; //initialize count array to all zero

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

count[array[i]]++; //increase number count in count array.

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

count[i] += count[i-1]; //find cumulative frequency

for(int i = size; i>=1; i--) {

output[count[array[i]]] = array[i];

count[array[i]] -= 1; //decrease count for same numbers

}

for(int i = 1; i<=size; i++) {

array[i] = output[i]; //store output array to main array

}

}

int main() {

int n;

cout << "Enter the number of elements: ";

cin >> n;

int arr[n+1]; //create an array with given number of elements

cout << "Enter elements:" << endl;

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

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

countSort(arr, n);

cout << "Array after Sorting: ";

display(arr, n);

}

U = U ∪ {v}

Prim's Algorithm Implementation in C++

The program below implements Prim's algorithm in C++. Although adjacency matrix representation of graph is used, this algorithm can also be implemented using Adjacency List to improve its efficiency.

#include <iostream>

#include <cstring>

using namespace std;

#define INF 9999999

// number of vertices in grapj

#define V 5

// create a 2d array of size 5x5

//for adjacency matrix to represent graph

int G[V][V] = {

{0, 9, 75, 0, 0},

{9, 0, 95, 19, 42},

{75, 95, 0, 51, 66},

{0, 19, 51, 0, 31},

{0, 42, 66, 31, 0}

};

int main () {

int no\_edge; // number of edge

// create a array to track selected vertex

// selected will become true otherwise false

int selected[V];

// set selected false initially

memset (selected, false, sizeof (selected));

// set number of edge to 0

no\_edge = 0;

// the number of egde in minimum spanning tree will be

// always less than (V -1), where V is number of vertices in

//graph

// choose 0th vertex and make it true

selected[0] = true;

int x; // row number

int y; // col number

// print for edge and weight

cout << "Edge" << " : " << "Weight";

cout << endl;

while (no\_edge < V - 1) {

//For every vertex in the set S, find the all adjacent vertices

// , calculate the distance from the vertex selected at step 1.

// if the vertex is already in the set S, discard it otherwise

//choose another vertex nearest to selected vertex at step 1.

int min = INF;

x = 0;

y = 0;

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

if (selected[i]) {

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

if (!selected[j] && G[i][j]) { // not in selected and there is an edge

if (min > G[i][j]) {

min = G[i][j];

x = i;

y = j;

}

}

}

}

}

cout << x << " - " << y << " : " << G[x][y];

cout << endl;

selected[y] = true;

no\_edge++;

}

return 0;

}

include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

#define edge pair<int,int>

class Graph {

private:

vector<pair<int, edge>> G; // graph

vector<pair<int, edge>> T; // mst

int \*parent;

int V; // number of vertices/nodes in graph

public:

Graph(int V);

void AddWeightedEdge(int u, int v, int w);

int find\_set(int i);

void union\_set(int u, int v);

void kruskal();

void print();

};

Graph::Graph(int V) {

parent = new int[V];

//i 0 1 2 3 4 5

//parent[i] 0 1 2 3 4 5

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

parent[i] = i;

G.clear();

T.clear();

}

void Graph::AddWeightedEdge(int u, int v, int w) {

G.push\_back(make\_pair(w, edge(u, v)));

}

int Graph::find\_set(int i) {

// If i is the parent of itself

if (i == parent[i])

return i;

else

// Else if i is not the parent of itself

// Then i is not the representative of his set,

// so we recursively call Find on its parent

return find\_set(parent[i]);

}

void Graph::union\_set(int u, int v) {

parent[u] = parent[v];

}

void Graph::kruskal() {

int i, uRep, vRep;

sort(G.begin(), G.end()); // increasing weight

for (i = 0; i < G.size(); i++) {

uRep = find\_set(G[i].second.first);

vRep = find\_set(G[i].second.second);

if (uRep != vRep) {

T.push\_back(G[i]); // add to tree

union\_set(uRep, vRep);

}

}

}

void Graph::print() {

cout << "Edge :" << " Weight" << endl;

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

cout << T[i].second.first << " - " << T[i].second.second << " : "

<< T[i].first;

cout << endl;

}

}

int main() {

Graph g(6);

g.AddWeightedEdge(0, 1, 4);

g.AddWeightedEdge(0, 2, 4);

g.AddWeightedEdge(1, 2, 2);

g.AddWeightedEdge(1, 0, 4);

g.AddWeightedEdge(2, 0, 4);

g.AddWeightedEdge(2, 1, 2);

g.AddWeightedEdge(2, 3, 3);

g.AddWeightedEdge(2, 5, 2);

g.AddWeightedEdge(2, 4, 4);

g.AddWeightedEdge(3, 2, 3);

g.AddWeightedEdge(3, 4, 3);

g.AddWeightedEdge(4, 2, 4);

g.AddWeightedEdge(4, 3, 3);

g.AddWeightedEdge(5, 2, 2);

g.AddWeightedEdge(5, 4, 3);

g.kruskal();

g.print();

return 0;

}

When we run the program, we get output as

Edge : Weight

1 - 2 : 2

2 - 5 : 2

2 - 3 : 3

3 - 4 : 3

0 - 1 : 4

Kruskal's vs Prim's Algorithm

Prim's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an edge, Prim's algorithm starts from a vertex and keeps adding lowest-weight edges which aren't in the tree, until all vertices have been covered.

Algorithms Programs

Programs List

Program 5: WAP to implement BFS in a graph represented via adjacency list.

Breadth First Search

#include<iostream>

#include<cstdlib>

using namespace std;

int cost[10][10],i,j,k,n,qu[10],front,rare,v,visit[10],visited[10];

int main()

{

int m;

cout <<"enterno of vertices";

cin >> n;

cout <<"ente no of edges";

cin >> m;

cout <<"\nEDGES \n";

for(k=1;k<=m;k++)

{

cin >>i>>j;

cost[i][j]=1;

}

cout <<"enter initial vertex";

cin >>v;

cout <<"Visitied vertices\n";

cout << v;

visited[v]=1;

k=1;

while(k<n)

{

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

{

if(cost[v][j]!=0 && visited[j]!=1 && visit[j]!=1)

{

visit[j]=1;

qu[rare++]=j;

}

}

v=qu[front++];

cout<<v << " ";

k++;

visit[v]=0; visited[v]=1;

}

return 0;

}

#include<iostream>

using namespace std;

struct node

{

int key;

node \*parent;

char color;

node \*left;

node \*right;

};

class RBtree

{

node \*root;

node \*q;

public :

RBtree()

{

q=NULL;

root=NULL;

}

void insert();

void insertfix(node \*);

void leftrotate(node \*);

void rightrotate(node \*);

void del();

node\* successor(node \*);

void delfix(node \*);

void disp();

void display( node \*);

void search();

};

void RBtree::insert()

{

int z,i=0;

cout<<"\nEnter key of the node to be inserted: ";

cin>>z;

node \*p,\*q;

node \*t=new node;

t->key=z;

t->left=NULL;

t->right=NULL;

t->color='r';

p=root;

q=NULL;

if(root==NULL)

{

root=t;

t->parent=NULL;

}

else

{

while(p!=NULL)

{

q=p;

if(p->key<t->key)

p=p->right;

else

p=p->left;

}

t->parent=q;

if(q->key<t->key)

q->right=t;

else

q->left=t;

}

insertfix(t);

}

void RBtree::insertfix(node \*t)

{

node \*u;

if(root==t)

{

t->color='b';

return;

}

while(t->parent!=NULL&&t->parent->color=='r')

{

node \*g=t->parent->parent;

if(g->left==t->parent)

{

if(g->right!=NULL)

{

u=g->right;

if(u->color=='r')

{

t->parent->color='b';

u->color='b';

g->color='r';

t=g;

}

}

else

{

if(t->parent->right==t)

{

t=t->parent;

leftrotate(t);

}

t->parent->color='b';

g->color='r';

rightrotate(g);

}

}

else

{

if(g->left!=NULL)

{

u=g->left;

if(u->color=='r')

{

t->parent->color='b';

u->color='b';

g->color='r';

t=g;

}

}

else

{

if(t->parent->left==t)

{

t=t->parent;

rightrotate(t);

}

t->parent->color='b';

g->color='r';

leftrotate(g);

}

}

root->color='b';

}

}

void RBtree::del()

{

if(root==NULL)

{

cout<<"\nEmpty Tree." ;

return ;

}

int x;

cout<<"\nEnter the key of the node to be deleted: ";

cin>>x;

node \*p;

p=root;

node \*y=NULL;

node \*q=NULL;

int found=0;

while(p!=NULL&&found==0)

{

if(p->key==x)

found=1;

if(found==0)

{

if(p->key<x)

p=p->right;

else

p=p->left;

}

}

if(found==0)

{

cout<<"\nElement Not Found.";

return ;

}

else

{

cout<<"\nDeleted Element: "<<p->key;

cout<<"\nColour: ";

if(p->color=='b')

cout<<"Black\n";

else

cout<<"Red\n";

if(p->parent!=NULL)

cout<<"\nParent: "<<p->parent->key;

else

cout<<"\nThere is no parent of the node. ";

if(p->right!=NULL)

cout<<"\nRight Child: "<<p->right->key;

else

cout<<"\nThere is no right child of the node. ";

if(p->left!=NULL)

cout<<"\nLeft Child: "<<p->left->key;

else

cout<<"\nThere is no left child of the node. ";

cout<<"\nNode Deleted.";

if(p->left==NULL||p->right==NULL)

y=p;

else

y=successor(p);

if(y->left!=NULL)

q=y->left;

else

{

if(y->right!=NULL)

q=y->right;

else

q=NULL;

}

if(q!=NULL)

q->parent=y->parent;

if(y->parent==NULL)

root=q;

else

{

if(y==y->parent->left)

y->parent->left=q;

else

y->parent->right=q;

}

if(y!=p)

{

p->color=y->color;

p->key=y->key;

}

if(y->color=='b')

delfix(q);

}

}

void RBtree::delfix(node \*p)

{

node \*s;

while(p!=root&&p->color=='b')

{

if(p->parent->left==p)

{

s=p->parent->right;

if(s->color=='r')

{

s->color='b';

p->parent->color='r';

leftrotate(p->parent);

s=p->parent->right;

}

if(s->right->color=='b'&&s->left->color=='b')

{

s->color='r';

p=p->parent;

}

else

{

if(s->right->color=='b')

{

s->left->color=='b';

s->color='r';

rightrotate(s);

s=p->parent->right;

}

s->color=p->parent->color;

p->parent->color='b';

s->right->color='b';

leftrotate(p->parent);

p=root;

}

}

else

{

s=p->parent->left;

if(s->color=='r')

{

s->color='b';

p->parent->color='r';

rightrotate(p->parent);

s=p->parent->left;

}

if(s->left->color=='b'&&s->right->color=='b')

{

s->color='r';

p=p->parent;

}

else

{

if(s->left->color=='b')

{

s->right->color='b';

s->color='r';

leftrotate(s);

s=p->parent->left;

}

s->color=p->parent->color;

p->parent->color='b';

s->left->color='b';

rightrotate(p->parent);

p=root;

}

}

p->color='b';

root->color='b';

}

}

void RBtree::leftrotate(node \*p)

{

if(p->right==NULL)

return ;

else

{

node \*y=p->right;

if(y->left!=NULL)

{

p->right=y->left;

y->left->parent=p;

}

else

p->right=NULL;

if(p->parent!=NULL)

y->parent=p->parent;

if(p->parent==NULL)

root=y;

else

{

if(p==p->parent->left)

p->parent->left=y;

else

p->parent->right=y;

}

y->left=p;

p->parent=y;

}

}

void RBtree::rightrotate(node \*p)

{

if(p->left==NULL)

return ;

else

{

node \*y=p->left;

if(y->right!=NULL)

{

p->left=y->right;

y->right->parent=p;

}

else

p->left=NULL;

if(p->parent!=NULL)

y->parent=p->parent;

if(p->parent==NULL)

root=y;

else

{

if(p==p->parent->left)

p->parent->left=y;

else

p->parent->right=y;

}

y->right=p;

p->parent=y;

}

}

node\* RBtree::successor(node \*p)

{

node \*y=NULL;

if(p->left!=NULL)

{

y=p->left;

while(y->right!=NULL)

y=y->right;

}

else

{

y=p->right;

while(y->left!=NULL)

y=y->left;

}

return y;

}

void RBtree::disp()

{

display(root);

}

void RBtree::display(node \*p)

{

if(root==NULL)

{

cout<<"\nEmpty Tree.";

return ;

}

if(p!=NULL)

{

cout<<"\n\t NODE: ";

cout<<"\n Key: "<<p->key;

cout<<"\n Colour: ";

if(p->color=='b')

cout<<"Black";

else

cout<<"Red";

if(p->parent!=NULL)

cout<<"\n Parent: "<<p->parent->key;

else

cout<<"\n There is no parent of the node. ";

if(p->right!=NULL)

cout<<"\n Right Child: "<<p->right->key;

else

cout<<"\n There is no right child of the node. ";

if(p->left!=NULL)

cout<<"\n Left Child: "<<p->left->key;

else

cout<<"\n There is no left child of the node. ";

cout<<endl;

if(p->left)

{

cout<<"\n\nLeft:\n";

display(p->left);

}

/\*else

cout<<"\nNo Left Child.\n";\*/

if(p->right)

{

cout<<"\n\nRight:\n";

display(p->right);

}

/\*else

cout<<"\nNo Right Child.\n"\*/

}

}

void RBtree::search()

{

if(root==NULL)

{

cout<<"\nEmpty Tree\n" ;

return ;

}

int x;

cout<<"\n Enter key of the node to be searched: ";

cin>>x;

node \*p=root;

int found=0;

while(p!=NULL&& found==0)

{

if(p->key==x)

found=1;

if(found==0)

{

if(p->key<x)

p=p->right;

else

p=p->left;

}

}

if(found==0)

cout<<"\nElement Not Found.";

else

{

cout<<"\n\t FOUND NODE: ";

cout<<"\n Key: "<<p->key;

cout<<"\n Colour: ";

if(p->color=='b')

cout<<"Black";

else

cout<<"Red";

if(p->parent!=NULL)

cout<<"\n Parent: "<<p->parent->key;

else

cout<<"\n There is no parent of the node. ";

if(p->right!=NULL)

cout<<"\n Right Child: "<<p->right->key;

else

cout<<"\n There is no right child of the node. ";

if(p->left!=NULL)

cout<<"\n Left Child: "<<p->left->key;

else

cout<<"\n There is no left child of the node. ";

cout<<endl;

}

}

int main()

{

int ch,y=0;

RBtree obj;

do

{

cout<<"\n\t RED BLACK TREE " ;

cout<<"\n 1. Insert in the tree ";

cout<<"\n 2. Delete a node from the tree";

cout<<"\n 3. Search for an element in the tree";

cout<<"\n 4. Display the tree ";

cout<<"\n 5. Exit " ;

cout<<"\nEnter Your Choice: ";

cin>>ch;

switch(ch)

{

case 1 : obj.insert();

cout<<"\nNode Inserted.\n";

break;

case 2 : obj.del();

break;

case 3 : obj.search();

break;

case 4 : obj.disp();

break;

case 5 : y=1;

break;

default : cout<<"\nEnter a Valid Choice.";

}

cout<<endl;

}while(y!=1);

return 1;

}

void Bucket\_Sort(int array[], int n)

{

int i, j;

int count[n];

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

{

count[i] = 0;

}

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

{

(count[array[i]])++;

}

for(i=0,j=0; i < n; i++)

{

for(; count[i]>0;(count[i])--)

{

array[j++] = i;

}

}

}