MIN-HEAP

#include<iostream>

#include<vector>

#include<queue>

using namespace std;

void Heapify(vector<int>& arr, int left, int right) {

int i = left;

while (2 \* i + 1 < right)

{

if (arr[i] > arr[2 \* i])

{

swap(arr[i], arr[2 \* i]);

i /= 2;

}

else if (arr[i] > arr[2 \* i + 1])

{

swap(arr[i], arr[2 \* i + 1]);

i = (i - 1) / 2;

}

else

{

++i;

}

}

}

int getMin(vector<int> heap)

{

return heap[0];

}

int extractMin(vector<int>& arr)

{

int min\_arr = arr[0];

swap(arr[0], arr[arr.size() - 1]);

arr.pop\_back();

Heapify(arr, 0, arr.size());

return min\_arr;

}

void insert(vector<int>& arr, int x)

{

arr.push\_back(x);

Heapify(arr, 0, arr.size());

}

bool is\_Heap(vector<int> arr)

{

int i = 0;

while (2 \* i + 1 < arr.size())

{

if (arr[i] > arr[2 \* i] || arr[i] > arr[2 \* i + 1])

{

return false;

}

else

{

++i;

}

}

return true;

}

int main()

{

vector<int> a;

int value;

cout << "Nhap gia tri :";

while (cin >> value)

{

cout << "Nhap gia tri :";

if (value == NULL)

{

break;

}

a.push\_back(value);

}

Heapify(a, 0, a.size());

cout << is\_Heap(a) << endl;

cout << getMin(a) << endl;

cout << extractMin(a) << endl;

cout << is\_Heap(a) << endl;

insert(a, 0);

cout << is\_Heap(a);

}

PRIORITY\_QUEUE

int N, Ai, K, Bi, sum = 0, keep;

vector<int>B;

priority\_queue<int> pq;//maxheap

//priority\_queue<int, vector<int>, greater<int>()> pq; //minhheap

cin >> N;

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

{

cin >> Ai;

pq.push(Ai);

}

cin >> K;

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

{

cin >> Bi;

B.push\_back(Bi);

}

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

{

if (B[i] == 1)

{

sum += pq.top();

pq.pop();

}

else if (B[i] == 2)

{

keep = pq.top();

pq.pop();

sum += pq.top();

pq.pop();

pq.push(keep);

}

}

cout << sum;

HASH-TABLE

#include<iostream>

#include<string>

#include<fstream>

#include<vector>

#include<sstream>

using namespace std;

struct CongTy

{

string ten\_Cong\_ty;

string ma\_So\_Thue;

string dia\_Chi;

};

CongTy convertstringToCongTy(string s)

{

CongTy congty;

stringstream ss;

ss << s;

getline(ss, congty.ten\_Cong\_ty, '|');

getline(ss, congty.ma\_So\_Thue, '|');

getline(ss, congty.dia\_Chi);

return congty;

}

vector<CongTy> ReadFile(string Filename)

{

ifstream in;

in.open(Filename);

vector<CongTy> comp;

if (!in.is\_open())

{

cout << "FILE NOT EXIST" << endl;

}

else

{

string temp;

while (!in.eof())

{

getline(in, temp);

comp.push\_back(convertstringToCongTy(temp));

}

}

return comp;

}

int main()

{

string filename = "mst.txt";

vector<CongTy> congty = ReadFile(filename);

}

BINARY-TREE

#include<iostream>

using namespace std;

struct Node

{

int key = 0;

Node\* left = NULL;

Node\* right = NULL;

};

typedef Node\* Tree;

Node\* CreateNode(int data)

{

Node\* p = new Node;

if (p == NULL)

{

cout << "NOT ENOUGH MEMORY";

}

else

{

p->key = data;

p->left = NULL;

p->right = NULL;

}

return p;

}

void CreateTree(Tree& root)

{

root = NULL;

}

void Insert(Tree& root, int value)

{

if (root == NULL)

{

Node\* p = CreateNode(value);

root = p;

}

else

{

if (root->key == value)

{

return;

}

if (root->key > value)

{

Insert(root->left, value);

}

else

{

Insert(root->right, value);

}

}

}

void NLR(Tree root)

{

if (root != NULL)

{

cout << root->key<<" ";

NLR(root->left);

NLR(root->right);

}

}

void LNR(Tree root)

{

if (root != NULL)

{

LNR(root->left);

cout << root->key<< " ";

LNR(root->right);

}

}

void LRN(Tree root)

{

if (root != NULL)

{

LRN(root->left);

LRN(root->right);

cout << root->key<< " " ;

}

}

int height(Tree root)

{

if (root == NULL)

return 0;

else

{

int leftheight = height(root->left);

int rightheight = height(root->right);

if (leftheight > rightheight)

{

return (leftheight + 1);

}

else

{

return (rightheight + 1);

}

}

}

void CurrentLevel(Tree root, int level)

{

if (root == NULL)

return;

if (level == 1)

cout << root->key<< " ";

else if (level > 1) {

CurrentLevel(root->left, level - 1);

CurrentLevel(root->right, level - 1);

}

}

void LevelOrder(Tree root)

{

int h = height(root);

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

{

CurrentLevel(root, i);

}

}

int CountNode(Tree tree)

{

if (tree == NULL)

return 0;

else

return 1 + CountNode(tree->left) + CountNode(tree->right);

}

Node\* search(Tree root,int x)

{

if (root!=NULL)

{

if (root->key == x)

{

return root;

}

if (root->key > x)

{

return search(root->left, x);

}

else

{

return search(root->right, x);

}

}

return NULL;

}

void remove\_rightTree(Tree& root, Tree& righttree)

{

if (root->left != NULL)

{

remove\_rightTree(righttree, root->left);

}

else

{

righttree->key = root->key;

righttree = root;

root = root->right;

}

}

void remove\_leftTree(Tree& root, Tree& lefttree)

{

if (root->right != NULL)

{

remove\_leftTree(lefttree, root->right);

}

else

{

lefttree->key = root->key;

lefttree = root;

root = root->left;

}

}

void remove(Tree root, int x)

{

if (root != NULL)

{

if (root->key > x)

{

remove(root->left, x);

}

else if (root->key < x)

{

remove(root->right, x);

}

else

{

Node\* p = root;

if (!root->left)

{

root = root->right;

}

else if (!root->right)

{

root = root->left;

}

else

{

remove\_leftTree(p, root->right);

remove\_rightTree(p, root->left);

}

delete p;

}

}

else

{

cout << "Khong tim thay nut can xoa";

}

}

void removeTree(Tree& root)

{

if (root != NULL)

{

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

root = NULL;

else

{

removeTree(root->left);

removeTree(root->right);

}

}

}

int find\_Hight\_Node(Tree root, int x,int& height)

{

if (root == NULL)

{

return -1;

}

int leftHeight = find\_Hight\_Node(root->left, x, height);

int rightHeight= find\_Hight\_Node(root->right, x, height);

int temp = max(leftHeight, rightHeight) + 1;

if (root->key == x)

height = temp;

return temp;

}

int HeightNode(Node\* root, int x)

{

int h = -1;

int maxHeight = find\_Hight\_Node(root, x, h);

return h;

}

bool isBST(Tree root)

{

if (root == NULL)

return true;

if (root->left != NULL && root->left->key >= root->key)

return false;

if (root->right != NULL && root->right->key <= root->key)

return false;

if (!isBST(root->left) || !isBST(root->right))

return false;

return true;

}

int main()

{

int luachon = 0;

Tree tree;

CreateTree(tree);

while (true)

{

cout << "-------------MENU-------------" << endl;

cout << "1:Them 1 node vao cay" << endl;

cout << "2:Duyet cay theo NLR" << endl;

cout << "3:Duyet cay theo LNR" << endl;

cout << "4:Duyet cay theo LRN" << endl;

cout << "5:Duyet cay theo level" << endl;

cout << "6:Tim chieu cao cua cay" << endl;

cout << "7:Dem so nut trong cay" << endl;

cout << "8:Tim 1 node trong cay" << endl;

cout << "9:Xoa 1 node trong cay" << endl;

cout << "10:Xoa toan bo cay" << endl;

cout << "11:Tinh chieu cao cua 1 node" << endl;

cout << "12:Kiem tra xem co phai cay nhi phan" << endl;

cout << "0:Thoat" << endl;

cout << "Nhap lua chon:"; cin >> luachon;

switch (luachon)

{

case 1:

{

int value,n;

cout << "Nhap so luong node can them vao cay:"; cin >> n;

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

{

cout << "Nhap Node :"; cin >> value;

Insert(tree, value);

}

break;

}

case 2:

{

NLR(tree);

system("pause");

break;

}

case 3:

{

LNR(tree);

system("pause");

break;

}

case 4:

{

LRN(tree);

system("pause");

break;

}

case 5:

{

LevelOrder(tree);

system("pause");

break;

}

case 6:

{

cout<< height(tree);

system("pause");

break;

}

case 7:

{

cout << CountNode(tree);

system("pause");

break;

}

case 8:

{

int x = 0;

cout << "Nhap Node can tim:"; cin >> x;

Node\* Search = search(tree, x);

cout << Search->key;

system("pause");

break;

}

case 9:

{

int x = 0;

cout << "Nhap Node can xoa:"; cin >> x;

remove(tree, x);

break;

}

case 10:

{

removeTree(tree);

break;

}

case 11:

{

int x;

cout << "Nhap Node can tinh chieu cao:"; cin >> x;

cout << HeightNode(tree, x);

system("pause");

break;

}

case 12:

{

if (isBST(tree))

cout << "La cay nhi phan tim kiem";

else

{

cout << "Khong la cay nhi phan tim kiem";

}

system("pause");

break;

}

case 0:

exit(0);

}

}

}

AVL-TREE

#include<iostream>

#include<vector>

#include<queue>

using namespace std;

struct Node {

int key=0;

int height=0;

Node\* left=NULL;

Node\* right=NULL;

};

Node\* CreateNode(int data)

{

Node\* p = new Node;

if (p == NULL)

{

cout << "NOT ENOUGH MEMORY";

}

p->key = data;

p->left = p->right = NULL;

p->height = 1;

return p;

}

int GetHeight(Node\* node)

{

if (node == NULL) {

return 0;

}

else return node->height;

}

Node\* RotateRight(Node\* y)

{

Node\* x = y->left;

Node\* t2 = x->right;

x->right = y;

y->left = t2;

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

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

return x;

}

Node\* RotateLeft(Node\* x)

{

Node\* y = x->right;

Node\* t2 = y->left;

y->left = x;

x->right = t2;

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

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

return y;

}

int getBalance(Node\* node)

{

if (node == NULL)

return 0;

return GetHeight(node->left) - GetHeight(node->right);

}

Node\* Insert(Node\* root, int data)

{

if (root==NULL)

{

return CreateNode(data);

}

else

{

if (data < root->key)

{

root->left = Insert(root->left, data);

}

else if (data > root->key)

{

root->right = Insert(root->right, data);

}

else

{

cout << "Co gia tri trung" << endl;

return NULL;

}

}

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

int balance = getBalance(root);

if (balance > 1)

{

if (getBalance(root->left) < 0)

{

root->left = RotateLeft(root->left);

}

root = RotateRight(root);

}

else if (balance < -1)

{

if (getBalance(root->right) > 0)

{

root->right = RotateRight(root);

}

root = RotateLeft(root);

}

return root;

}

bool isAVL(Node\* root)

{

if (!root)

{

return true;

}

int balance = getBalance(root);

if (balance > 1 || balance < -1)

return false;

else

return isAVL(root->left) && isAVL(root->right);

}

void LNR(Node\* root)

{

if (root==NULL) return;

LNR(root->left);

cout << root->key << "-" << GetHeight(root) << " ";

LNR(root->right);

return;

}

void NLR(Node\* root)

{

if (!root) return;

cout << root->key << "-" << GetHeight(root) << " ";

LNR(root->left);

LNR(root->right);

return;

}

void LRN(Node\* root)

{

if (!root) return;

LNR(root->left);

LNR(root->right);

cout << root->key << "-" << GetHeight(root) << " ";

return;

}

void levelOrder(Node\* root)

{

queue<Node\*>queue;

if (root!=NULL)

{

queue.push(root);

}

while (queue.size() > 0)

{

int size = queue.size();

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

{

Node\* temp = queue.front();

queue.pop();

cout << temp->key << "-" << GetHeight(temp) << " ";

if (temp->left!=NULL)

{

queue.push(temp->left);

}

if (temp->right!=NULL)

{

queue.push(temp->right);

}

}

}

}

int countLeaves(Node\* root)

{

if (!root)

{

return 0;

}

else

{

return countLeaves(root->left) + countLeaves(root->right) + 1 ;

}

}

int countLesser(Node\* root, int x) {

if (!root)

{

return 0;

}

else

{

if (root->key < x)

return countLesser(root->left, x) + countLesser(root->right, x) + 1;

else if (root->key == x)

return countLeaves(root->left);

else

return countLesser(root->left, x);

}

}

Node\* minNode(Node\* node)

{

Node\* current = node;

while (current->left != NULL)

{

current = current->left;

}

return current;

}

Node\* deleteNode(Node\* root, int x)

{

if (root == NULL)

{

return root;

}

else if (root->key < x)

{

root->right = deleteNode(root->right, x);

}

else if (root->key > x)

{

root->left = deleteNode(root->left, x);

}

else

{

if (root->left && root->right)

{

Node\* temp = minNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

else if (root->right)

{

Node\* temp = root->right;

delete root;

return temp;

}

else if (root->left)

{

Node\* temp = root->left;

delete root;

return temp;

}

else

{

delete root;

return NULL;

}

}

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

int balance = getBalance(root);

if (balance > 1)

{

if (getBalance(root->left) < 0)

{

root->left = RotateLeft(root->left);

}

root = RotateRight(root);

}

else if (balance < -1)

{

if (getBalance(root->right) > 0)

{

root->right = RotateRight(root);

}

root = RotateLeft(root);

}

return root;

}

Node\* search(Node\* root, int x) {

if (root != NULL)

{

if (root->key == x)

{

return root;

}

if (root->key > x)

{

return search(root->left, x);

}

else

{

return search(root->right, x);

}

}

return NULL;

}

int countGreater(Node\* root, int x) {

if (!root) {

return 0;

}

else

{

if (root->key > x)

{

return 1 + countGreater(root->left, x) + countGreater(root->right, x);

}

else if (root->key == x)

{

return countLeaves(root->right);

}

else

{

return countGreater(root->right, x);

}

}

}

void remove\_rightTree(Node\*& root, Node\*& righttree)

{

if (root->left != NULL)

{

remove\_rightTree(righttree, root->left);

}

else

{

righttree->key = root->key;

righttree = root;

root = root->right;

}

}

void remove\_leftTree(Node\*& root, Node\*& lefttree)

{

if (root->right != NULL)

{

remove\_leftTree(lefttree, root->right);

}

else

{

lefttree->key = root->key;

lefttree = root;

root = root->left;

}

}

void remove(Node\*& root, int x)

{

if (root != NULL)

{

if (root->key > x)

{

remove(root->left, x);

}

else if (root->key < x)

{

remove(root->right, x);

}

else

{

Node\* p = root;

if (!root->left)

{

root = root->right;

}

else if (!root->right)

{

root = root->left;

}

else

{

remove\_leftTree(p, root->right);

remove\_rightTree(p, root->left);

}

delete p;

}

}

else

{

cout << "Khong tim thay nut can xoa";

}

}

void removeTree(Node\*& root)

{

if (root != NULL)

{

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

root = NULL;

else

{

removeTree(root->left);

removeTree(root->right);

}

}

}

int main()

{

Node\* root = NULL;

int n = 0, value = 0, luachon = 0;

while (true)

{

cout << "-------------MENU-------------" << endl;

cout << "1:Them 1 node vao cay" << endl;

cout << "2:Duyet cay theo NLR" << endl;

cout << "3:Duyet cay theo LNR" << endl;

cout << "4:Duyet cay theo LRN" << endl;

cout << "5:Duyet cay theo level" << endl;

cout << "6:Dem so luong nut trong cay nho hon 1 gia tri" << endl;

cout << "7:Dem so luong nut trong caylon hon 1 gia tri" << endl;

cout << "8:Dem la trong cay" << endl;

cout << "9:Xoa 1 node trong cay" << endl;

cout << "10:Xoa toan bo cay" << endl;

cout << "11:Kiem tra xem co phai cay AVL" << endl;

cout << "12:Tim 1 node trong cay" << endl;

cout << "0:Thoat" << endl;

cout << "Nhap lua chon:"; cin >> luachon;

switch (luachon)

{

case 1:

{

int value, n;

cout << "Nhap so luong node can them vao cay:"; cin >> n;

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

{

cout << "Nhap Node :"; cin >> value;

Insert(root, value);

}

break;

}

case 2:

{

NLR(root);

system("pause");

break;

}

case 3:

{

LNR(root);

system("pause");

break;

}

case 4:

{

LRN(root);

system("pause");

break;

}

case 5:

{

levelOrder(root);

system("pause");

break;

}

case 6:

{

int x;

cin >> x;

cout << countLesser(root, x) << endl;

system("pause");

break;

}

case 7:

{

int x;

cin >> x;

cout << countGreater(root, x) << endl;

system("pause");

break;

}

case 8:

{

cout << countLeaves << endl;

system("pause");

break;

}

case 9:

{

int x = 0;

cout << "Nhap Node can xoa:"; cin >> x;

remove(root, x);

break;

}

case 10:

{

removeTree(root);

break;

}

case 11:

{

if (isAVL(root))

cout << "La cay AVL";

else

{

cout << "Khong la cay AVL";

}

system("pause");

break;

}

case 12:

{

int x = 0;

cout << "Nhap Node can tim:"; cin >> x;

Node\* Search = search(root, x);

cout << Search->key;

system("pause");

break;

}

}

}

}

GRAPH

#include<iostream>

#include<fstream>

#include<vector>

#include<queue>

#define MAX 100

using namespace std;

struct Graph

{

int num\_vertices;

vector<vector<int>>adjacency\_matrix;

};

Graph Read\_File(string filename)

{

Graph G;

fstream f;

f.open(filename);

if (!f.is\_open()) {

cout << "ERROR\n";

}

else

{

int n;

f >> n;

G.num\_vertices = n;

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

{

vector<int>a;

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

{

int x;

f >> x;

a.push\_back(x);

}

G.adjacency\_matrix.push\_back(a);

}

}

f.close();

return G;

}

void Display(Graph G) {

for (int i = 0; i < G.num\_vertices; i++)

{

cout << i << " : ";

for (int j = 0; j < G.num\_vertices; j++)

{

if (G.adjacency\_matrix[i][j] > 0)

{

cout << j << " ";

}

}

cout << endl;

}

}

bool UndirectedGraph(Graph G) {

for (int i = 0; i < G.num\_vertices; i++)

{

for (int j = 0; j < G.num\_vertices; j++)

{

if (G.adjacency\_matrix[i][j] != G.adjacency\_matrix[j][i])

{

return false;

}

}

}

return true;

}

int Count\_Edges(Graph G)

{

int count = 0;

if (UndirectedGraph(G) == true)

{

for (int i = 0; i < G.num\_vertices; i++)

{

for (int j = i; j < G.num\_vertices; j++)

{

count += G.adjacency\_matrix[i][j];

}

}

}

else if (UndirectedGraph(G) == false)

{

for (int i = 0; i < G.num\_vertices; i++)

{

for (int j = 0; j < G.num\_vertices; j++)

{

count += G.adjacency\_matrix[i][j];

}

}

}

return count;

}

void List\_DegreeOfVertices(Graph G)

{

if (UndirectedGraph(G))

{

for (int i = 0; i < G.num\_vertices; i++)

{

int count = 0;

for (int j = 0; j < G.num\_vertices; j++)

{

count += G.adjacency\_matrix[i][j];

}

cout << "So bac dinh " << i << ": " << count << endl;

}

}

else if (UndirectedGraph(G) == false)

{

for (int i = 0; i < G.num\_vertices; i++)

{

int count\_out = 0;

for (int j = 0; j < G.num\_vertices; j++)

{

count\_out += G.adjacency\_matrix[i][j];

}

cout << "So bac ra dinh " << i << ": " << count\_out << endl;

int count\_in = 0;

for (int j = 0; j < G.num\_vertices; j++)

{

count\_in += G.adjacency\_matrix[j][i];

}

cout << "So bac vao dinh " << i << ": " << count\_in << endl;

}

}

}

void DFS(Graph G, int start,bool check[] )

{

check[start] = false;

for (int i = 0; i < G.num\_vertices; i++)

{

if (G.adjacency\_matrix[start][i] == 1 && check[i])

{

DFS(G, i, check);

}

}

}

void BFS(Graph G, int start, bool check[])

{

queue<int>list;

for (int i = 0; i < G.num\_vertices; i++)

{

check[i] = false;

}

check[start] = true;

list.push(start);

while (!list.empty())

{

start = list.front();

list.pop();

cout << start << " ";

for (int i = 0; i < G.num\_vertices; i++)

{

if (!check[i])

{

check[i] = 1;

list.push(i);

}

}

}

}

bool ConnectedGraph(Graph G, bool check[])

{

for (int i = 0; i < G.num\_vertices; i++)

{

check[i] = 0;

}

for (int i = 0; i < G.num\_vertices; i++)

{

if (!check[i])

{

return true;

DFS(G, i, check);

}

}

return false;

}

int Count\_ConnectedGraph(Graph G, bool check[])

{

int count = 0;

for (int i = 0; i < G.num\_vertices; i++)

{

if (!check[i])

{

count++;

DFS(G, i, check);

}

}

return count;

}

int main()

{

Graph G;

G = Read\_File("graph.txt");

Display(G);

if (UndirectedGraph(G))

{

cout << "Do thi vo huong!" << endl;

}

else

{

cout << "Do thi co huong!" << endl;

}

cout << "So canh cua do thi: " << Count\_Edges(G) << endl;

List\_DegreeOfVertices(G);

bool check\_DFS[MAX];

cout << " DFS: " << endl;

DFS(G, 2, check\_DFS);

bool check\_BFS[MAX];

cout << " BFS: " << endl;

BFS(G, 2, check\_BFS);

bool check\_Connected[MAX];

if (ConnectedGraph(G, check\_Connected) == true)

{

cout << "Do thi co lien thong!" << endl;

}

else {

cout << "Do thi khong lien thong!" << endl;

}

bool check\_Connected1[MAX];

cout << "So thanh phan lien thong: " << Count\_ConnectedGraph(G, check\_Connected1);

system("pause");

}

SORT

vector<int> interchangesort(vector<int>&arr)

{

int size = arr.size();

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

{

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

{

if (arr[i] > arr[j])

{

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

}

}

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

int BinarySearch(vector<int>arr,int left,int right,int x)

{

if (right <= left)

return(x > arr[left]) ? (left + 1) : left;

int mid = (left + right) / 2;

if (arr[mid] == x)

return mid + 1 ;

else if (arr[mid] > x)

{

return BinarySearch(arr, left, mid - 1, x);

}

else return BinarySearch(arr, mid + 1, right, x);

}

vector<int> BinaryInsertionSort(vector<int>arr)

{

int key = 0, pos = 0, newpos = 0;

int size = arr.size();

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

{

key = arr[i];

pos = i - 1;

newpos = BinarySearch(arr, 0, pos, key);

while (pos >= newpos)

{

arr[pos + 1] = arr[pos];

pos--;

}

arr[pos + 1] = key;

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

vector<int> insertionSort(vector<int> arr)

{

int key = 0,pos = 0;

int size = arr.size();

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

{

key = arr[i];

pos = i - 1;

while (pos >= 0 && arr[pos] > key)

{

arr[pos + 1] = arr[pos];

pos--;

}

arr[pos + 1] = key;

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

vector<int> ShellSort(vector<int>arr)

{

int n = arr.size();

int interval = 0;

for (interval = n / 2; interval >= 1; interval /= 2)

{

for (int j = interval; j < n; j++)

{

for (int i = j - interval; i >= 0; i -= interval)

{

if (arr[i] > arr[i + interval])

{

swap(arr[i], arr[i + interval]);

}

else break;

}

}

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

vector<int> SelectionSort(vector<int>arr)

{

int size = arr.size();

int min;

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

{

min = i;

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

{

if (arr[j] < arr[min])

{

swap(arr[min], arr[j]);

min = j;

}

}

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

vector<int> BubbleSort(vector<int> arr)

{

size\_t size = arr.size();

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

{

for (size\_t j = size - 1; j > i; j--)

{

if (arr[j] < arr[static\_cast<\_\_int64>(j) - 1])

swap(arr[j], arr[static\_cast<\_\_int64>(j) - 1]);

}

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

vector<int> ShakerSort(vector<int>arr)

{

int size = arr.size();

int left = 0, right = size - 1;

while (right > left)

{

for (int i = left; i < right; i++)

{

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

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

}

right--;

for (int j = right; j > left; j--)

{

if (arr[j] < arr[j - 1])

swap(arr[j], arr[j - 1]);

}

left++;

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

void CountingSort(int \*arr, int n) {

int arr1[100] = {};

int count\_arr[100] = {};

int x = arr[0];

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

{

if (arr[i] > x)

x = arr[i];

}

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

{

count\_arr[i] = 0;

}

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

{

count\_arr[arr[i]]++;

}

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

{

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

}

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

{

arr1[count\_arr[arr[i]] - 1] = arr[i];

count\_arr[arr[i]]--;

}

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

{

arr[i] = arr1[i];

}

}

////////////////////////////////////////////////////////////////////////////////////

void sift(vector<int>&arr, int left, int right)// xây dựng heap

{

int i = left;

int j = i \* 2 + 1;

int x = arr[i];

while (j <= right)

{

if (j < right)

{

if (arr[j] <= arr[j + 1])

j++;

}

if (x >= arr[j]) break;

else

{

arr[i] = arr[j];

}

i = j;

j = i \* 2 + 1;

}

arr[i] = x;

}

void Heapivfy(vector<int>&arr,int n)// đưa phần tử vào heap

{

int left = (n / 2) - 2;

while (left >= 0)

{

sift(arr, left, n);

left--;

}

}

vector<int> HeapSort(vector<int>& arr, int n)//sắp xếp tăng dần

{

Heapivfy(arr, arr.size());

for (int j = n - 1; j >= 0; j--)

{

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

sift(arr, 0, j - 1);

}

return arr;

}

////////////////////////////////////////////////////////////////////////////////////

void Merge(int arr[], int left, int mid, int right)

{

int temp[7] = {};

int i = left, j = mid + 1, k = left;

while (i <= mid && j <= right)

{

if (arr[i] <= arr[j])

{

temp[k] = arr[i];

i++;

}

else

{

temp[k] = (arr[j]);

j++;

}

k++;

}

if (i > mid)

{

while (j <= right)

{

temp[k] = arr[j];

j++, k++;

}

}

else

{

while (i <= mid)

{

temp[k] = arr[i];

i++, k++;

}

}

for (int k = left; k <= right; k++)

{

arr[k] = temp[k];

}

}

void MergeSort(int arr[], int left, int right)

{

if (left < right)

{

int mid = (left + right) / 2;

MergeSort(arr, left, mid);

MergeSort(arr, mid + 1, right);

Merge(arr, left, mid, right);

}

}

////////////////////////////////////////////////////////////////////////////////////

int partition(int arr[], int left, int right)

{

int pivvot = arr[right];

int i = (left - 1);

for (int j = left; j <= right - 1; j++)

{

if (arr[j] < pivvot)

{

i++;

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

}

}

swap(arr[i + 1], arr[right]);

return (i + 1);

}

void quickSort(int arr[], int left, int right)

{

if (left < right)

{

int piv = partition(arr, left, right);

quickSort(arr, left, piv - 1);

quickSort(arr, piv + 1, right);

}

}

////////////////////////////////////////////////////////////////////////////////////

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;

}

void countSort(int \*arr, int n, int exp)

{

int \*output=new int[n];

int i, count[1000] = {};

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

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

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

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

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

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

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

}

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

arr[i] = output[i];

}

void RadixSort(int \*arr, int n)

{

int m = getMax(arr, n);

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

countSort(arr, n, exp);

}