Heap

#include <iostream>

#include <vector>

using namespace std;

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

{

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(vector<int>& heap,int size, int i)

{

int largest = i;

int l = (2 \* i )+ 1;

int r = (2 \* i) + 2;

if (l < size && heap[l] > heap[largest])

largest = l;

if (r < size && heap[r] > heap[largest])

largest = r;

if (largest != i)

{

swap(&heap[i], &heap[largest]);

heapify(heap, size,largest);

}

}

void insert(vector<int>& heap, int element)

{

heap.push\_back(element);

int size = heap.size();

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

{

heapify(heap, size,i);

}

}

//for sorting heap and to make priority queue

void heapSort(vector<int>& heap) {

int size = heap.size();

for (int i = size / 2 - 1; i >= 0; i--) //heapifying heap if necessary

heapify(heap, size, i);

for (int i = size - 1; i >= 0; i--) { //sorting heap

swap(&heap[0], &heap[i]);

heapify(heap, i, 0);

}

int counter = 1;

for (int i = size - 1; i >= 0; i--) { //defining priorites

cout << "Priority " << counter << " -> " << heap[i] << "\n";

counter += 1;

}

}

void printArray(vector<int>& heap)

{

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

cout << heap[i] << " ";

cout << "\n";

}

int main()

{

vector<int> heap;

insert(heap, 10);

insert(heap, 20);

insert(heap, 15);

insert(heap,30);

insert(heap, 40);

cout << "Max-Heap array: ";

printArray(heap);

heapSort(heap);

}

Binary Search Tree

#include <iostream>

using namespace std;

struct bst\_node {

int data;

bst\_node\* left;

bst\_node\* right;

};

bst\_node\* createNode(int data) {

bst\_node\* root = new bst\_node();

root->data = data;

root->left = NULL;

root->right = NULL;

return root;

}

bst\_node\* insertNode(bst\_node\*root,int data) {

if (root == NULL) {

bst\_node\* root = createNode(data);

return root;

}

else if (root->data > data) { //element is small

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

return root;

}

else if(root->data < data) { //element is big

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

return root;

}

}

void preorderTraversal(bst\_node\* root) {

if (root == NULL) { //base case

return;

}

else {

cout << root->data << " ";

preorderTraversal(root->left);

preorderTraversal(root->right);

}

}

void inorderTraversal(bst\_node\* root) {

if (root == NULL) {

return;

}

else {

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

}

void postorderTraversal(bst\_node\* root) {

if (root == NULL) {

return;

}

else {

postorderTraversal(root->left);

postorderTraversal(root->right);

cout << root->data << " ";

}

}

bst\_node\* findMin(bst\_node\* root) {

bst\_node\* temp = root;

while (temp->left != NULL) {

temp = temp->left;

}

return temp;

}

bst\_node\* findMax(bst\_node\* root) {

bst\_node\* temp = root;

while (temp->right != NULL) {

temp = temp->right;

}

return temp;

}

bst\_node\* deleteNode(bst\_node\* root,int element) {

if (root == NULL) {

return root;

}

//finding the element

else if (root->data > element) {

deleteNode(root->left,element);

}

else if (root->data < element) {

deleteNode(root->right, element);

}

//deleting the element

else {

//case -- have one child

if (root->left == NULL) {

bst\_node\* temp = root->right;

delete root;

return temp;

}

else if (root->right == NULL) {

bst\_node\* temp = root->left;

delete root;

return temp;

}

//case -- have two children

bst\_node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->left, root->data);

}

return root;

}

void searchNode(bst\_node\* root,int element) {

if (root == NULL) {

return;

}

//checking the element

else if (root->data == element) {

cout << "Node found : " << root->data;

}

//finding element

else if (root->data > element) {

searchNode(root->left, element);

}

else if (root->data < element) {

searchNode(root->right, element);

}

}

Graphs

#include <iostream>

#include <vector>

#include <queue>

#include <string>

#include <limits>

using namespace std;

struct edge {

int src;

int dest;

edge(int s, int d) {

this->src = s;

this->dest = d;

}

};

void createGraph(vector<edge> graph[]) {

graph[0].push\_back(edge(0, 1));

graph[0].push\_back(edge(0, 2));

graph[1].push\_back(edge(1, 0));

graph[1].push\_back(edge(1, 3));

graph[2].push\_back(edge(2, 0));

graph[2].push\_back(edge(2, 4));

graph[3].push\_back(edge(3, 1));

graph[3].push\_back(edge(3, 4));

graph[3].push\_back(edge(3, 5));

graph[4].push\_back(edge(4, 2));

graph[4].push\_back(edge(4, 3));

graph[4].push\_back(edge(4, 5));

graph[5].push\_back(edge(5, 3));

graph[5].push\_back(edge(5, 4));

graph[5].push\_back(edge(5, 6));

graph[6].push\_back(edge(6, 5));

cout << "Graph was created successfully...";

}

void bfs(vector<edge> graph[]) {

queue<int> queue;

bool visited[7];

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

visited[i] = false;

queue.push(0);

while (!queue.empty()) {

int current = queue.front();

queue.pop();

if (visited[current] == false) {

cout << current << " ";

visited[current] = true;

for (int i = 0; i < size(graph[current]); i++) {

queue.push(graph[current][i].dest);

}

}

}

}

void dfs(vector<edge> graph[], int current, bool visited[]) {

cout << current << " ";

visited[current] = true;

for (int i = 0; i < graph[current].size(); i++) {

if (visited[graph[current][i].dest] == false)

dfs(graph, graph[current][i].dest, visited);

}

}

//by using dfs

void allPaths(vector<edge> graph[], int current, bool visited[], string path, int target) {

if (current == target) {

cout << path << "\n";

return;

}

else {

for (int i = 0; i < size(graph[current]); i++) {

if (visited[graph[current][i].dest] == false) {

visited[current] = true;

string e = to\_string(graph[current][i].dest);

allPaths(graph, graph[current][i].dest, visited, path + e, target);

visited[current] = false;

}

}

}

}

//all edges should be given weights to work with dijkstra’s algo

void dijkstra(vector<Edge> graph[],int src) {

priority\_queue<Pair> pq;

int distance[6];

for (int i = 0; i < 6; i++) {

if (i != src)

distance[i] = numeric\_limits<int>::max();

}

distance[src] = 0;

bool visited[6];

for (int i = 0; i < 6; i++)

visited[i] = false;

pq.push(Pair(src, 0));

while (pq.empty() == false) {

Pair current = pq.top();

pq.pop();

if (visited[current.node] == false) {

visited[current.node] = true;

for (int i = 0; i < graph[current.node].size(); i++) {

Edge e = graph[current.node][i];

int u = e.src;

int v = e.dest;

if (distance[u] + e.wt < distance[v]) {

distance[v] = distance[u] + e.wt;

pq.push(Pair(v, distance[v]));

}

}

}

}

cout << "Shortest paths from source to all vertices";

for (int i = 0; i < 6; i++) {

cout <<"Vertex "<<i <<" : "<< distance[i] << " ";

}

}