**CSE 211 Homework**

**Due: 25.4.2017**

* 1. Draw the binary tree that gave the following results when traversed:

**Inorder:** ABDCEFG

**Preorder:** CBADEFG

**Postorder:** ADBGFEC

* 1. Create a Binary Search Tree (BST) and insert the following numbers in this order:

9 10 8 1 2 26 35 30 4 11 7 21

* 1. Delete 26 and 10 and show the resulting tree.

1. Assume that we are working on the following graph:
2. Show the adjacency matrix representation of the graph.
3. Print the nodes using BFS starting from node 1. (Use the smallest number first when picking a node)
4. Print the nodes using DFS starting from node 1. . (Use the smallest number first when picking a node)
5. If the adjacency matrix A given as the input, write a function that finds the total number of edges in the graph.
6. What is the complexity of your function at (d) ?
7. What would be the complexity of function at (d) if you use adjacency lists instead of adjacency matrix?
8. The following array contains random numbers.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Value | 12 | 28 | 14 | 41 | 18 | 67 | 52 | 25 | 10 | 11 | 4 | 19 | 37 | 7 | 22 |

1. Convert this array into a min. heap (using Build-Heap (Heapify) function ) and show the final array in the following table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Value |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

1. Apply Delete-Min function to the heap in part (a). This function will delete the minimum item in the heap while keeping the array as a heap. Draw the final layout of array in the following table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Value |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

1. Insert 19 into the heap in part (b) and draw the final layout of array in the following table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Value |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

1. When a heap is traversed preorder the following output is generated:

21 19 24 18 62 10 46 25 18

Draw the tree that represents the given heap.

1. Suppose we have the following BST implementation
   1. Write a function that finds the total number of nodes that has two children in the tree.
   2. Write a function that returns the sum of minimum and maximum numbers in the given BST.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* genBST.h \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// generic binary search tree

#include <queue>

#include <stack>

#ifndef BINARY\_SEARCH\_TREE

#define BINARY\_SEARCH\_TREE

template<class T>

class Stack : public stack<T> {

public:

T pop() {

T tmp = top();

stack<T>::pop();

return tmp;

}

};

template<class T>

class Queue : public queue<T> {

public:

T dequeue() {

T tmp = front();

queue<T>::pop();

return tmp;

}

void enqueue(const T& el) {

push(el);

}

};

template<class T> class BST;

template<class T>

class BSTNode {

public:

BSTNode() {

left = right = 0;

}

BSTNode(const T& e, BSTNode<T> \*l = 0, BSTNode<T> \*r = 0) {

el = e; left = l; right = r;

}

T el;

BSTNode<T> \*left, \*right;

};

template<class T>

class BST {

public:

BST() {

root = 0;

}

~BST() {

clear();

}

void clear() {

clear(root);

root = 0;

}

bool isEmpty() const {

return root == 0;

}

void preorder() {

preorder(root);

}

void inorder() {

inorder(root);

}

void postorder() {

postorder(root);

}

void insert(const T&);

void recursiveInsert(const T& el) {

recursiveInsert(root,el);

}

T\* search(const T& el) const {

return search(root,el);

}

T\* recursiveSearch(const T& el) const {

return recursiveSearch(root,el);

}

void deleteByCopying(BSTNode<T>\*&);

void findAndDeleteByCopying(const T&);

void deleteByMerging(BSTNode<T>\*&);

void findAndDeleteByMerging(const T&);

void iterativePreorder();

void iterativeInorder();

void iterativePostorder();

void breadthFirst();

void MorrisPreorder();

void MorrisInorder();

void MorrisPostorder();

void balance(T\*,int,int);

protected:

BSTNode<T>\* root;

void clear(BSTNode<T>\*);

void recursiveInsert(BSTNode<T>\*&, const T&);

T\* search(BSTNode<T>\*, const T&) const;

T\* recursiveSearch(BSTNode<T>\*, const T&) const;

void preorder(BSTNode<T>\*);

void inorder(BSTNode<T>\*);

void postorder(BSTNode<T>\*);

virtual void visit(BSTNode<T>\* p) {

cout << p->el << ' ';

}

};

template<class T>

void BST<T>::clear(BSTNode<T> \*p) {

if (p != 0) {

clear(p->left);

clear(p->right);

delete p;

}

}

template<class T>

void BST<T>::insert(const T& el) {

BSTNode<T> \*p = root, \*prev = 0;

while (p != 0) { // find a place for inserting new node;

prev = p;

if (el < p->el)

p = p->left;

else p = p->right;

}

if (root == 0) // tree is empty;

root = new BSTNode<T>(el);

else if (el < prev->el)

prev->left = new BSTNode<T>(el);

else prev->right = new BSTNode<T>(el);

}

template<class T>

void BST<T>::recursiveInsert(BSTNode<T>\*& p, const T& el) {

if (p == 0)

p = new BSTNode<T>(el);

else if (el < p->el)

recursiveInsert(p->left, el);

else recursiveInsert(p->right,el);

}

template<class T>

T\* BST<T>::search(BSTNode<T>\* p, const T& el) const {

while (p != 0)

if (el == p->el)

return &p->el;

else if (el < p->el)

p = p->left;

else p = p->right;

return 0;

}

template<class T>

T\* BST<T>::recursiveSearch(BSTNode<T>\* p, const T& el) const {

if (p != 0)

if (el == p->el)

return &p->el;

else if (el < p->el)

return recursiveSearch(p->left,el);

else return recursiveSearch(p->right,el);

else return 0;

}

template<class T>

void BST<T>::inorder(BSTNode<T> \*p) {

if (p != 0) {

inorder(p->left);

visit(p);

inorder(p->right);

}

}

template<class T>

void BST<T>::preorder(BSTNode<T> \*p) {

if (p != 0) {

visit(p);

preorder(p->left);

preorder(p->right);

}

}

template<class T>

void BST<T>::postorder(BSTNode<T>\* p) {

if (p != 0) {

postorder(p->left);

postorder(p->right);

visit(p);

}

}

template<class T>

void BST<T>::deleteByCopying(BSTNode<T>\*& node) {

BSTNode<T> \*previous, \*tmp = node;

if (node->right == 0) // node has no right child;

node = node->left;

else if (node->left == 0) // node has no left child;

node = node->right;

else {

tmp = node->left // node has both children;

previous = node; // 1.

while (tmp->right != 0) { // 2.

previous = tmp;

tmp = tmp->right;

}

node->el = tmp->el; // 3.

if (previous == node)

previous->left = tmp->left;

else previous->right = tmp->left; // 4.

}

delete tmp; // 5.

}

// findAndDeleteByCopying() searches the tree to locate the node containing

// el. If the node is located, the function DeleteByCopying() is called.

template<class T>

void BST<T>::findAndDeleteByCopying(const T& el) {

BSTNode<T> \*p = root, \*prev = 0;

while (p != 0 && !(p->el == el)) {

prev = p;

if (el < p->el)

p = p->left;

else p = p->right;

}

if (p != 0 && p->el == el)

if (p == root)

deleteByCopying(root);

else if (prev->left == p)

deleteByCopying(prev->left);

else deleteByCopying(prev->right);

else if (root != 0)

cout << "el " << el << " is not in the tree\n";

else cout << "the tree is empty\n";

}

template<class T>

void BST<T>::deleteByMerging(BSTNode<T>\*& node) {

BSTNode<T> \*tmp = node;

if (node != 0) {

if (!node->right) // node has no right child: its left

node = node->left; // child (if any) is attached to its parent;

else if (node->left == 0) // node has no left child: its right

node = node->right; // child is attached to its parent;

else { // be ready for merging subtrees;

tmp = node->left; // 1. move left

while (tmp->right != 0)// 2. and then right as far as possible;

tmp = tmp->right;

tmp->right = // 3. establish the link between the

node->right; // the rightmost node of the left

// subtree and the right subtree;

tmp = node; // 4.

node = node->left; // 5.

}

delete tmp; // 6.

}

}

template<class T>

void BST<T>::findAndDeleteByMerging(const T& el) {

BSTNode<T> \*node = root, \*prev = 0;

while (node != 0) {

if (node->el == el)

break;

prev = node;

if (el < node->el)

node = node->left;

else node = node->right;

}

if (node != 0 && node->el == el)

if (node == root)

deleteByMerging(root);

else if (prev->left == node)

deleteByMerging(prev->left);

else deleteByMerging(prev->right);

else if (root != 0)

cout << "el " << el << " is not in the tree\n";

else cout << "the tree is empty\n";

}

template<class T>

void BST<T>::iterativePreorder() {

Stack<BSTNode<T>\*> travStack;

BSTNode<T> \*p = root;

if (p != 0) {

travStack.push(p);

while (!travStack.empty()) {

p = travStack.pop();

visit(p);

if (p->right != 0)

travStack.push(p->right);

if (p->left != 0) // left child pushed after right

travStack.push(p->left); // to be on the top of the stack;

}

}

}

template<class T>

void BST<T>::iterativeInorder() {

Stack<BSTNode<T>\*> travStack;

BSTNode<T> \*p = root;

while (p != 0) {

while (p != 0) { // stack the right child (if any)

if (p->right) // and the node itself when going

travStack.push(p->right); // to the left;

travStack.push(p);

p = p->left;

}

p = travStack.pop(); // pop a node with no left child

while (!travStack.empty() && p->right == 0) { // visit it and all nodes

visit(p); // with no right child;

p = travStack.pop();

}

visit(p); // visit also the first node with

if (!travStack.empty()) // a right child (if any);

p = travStack.pop();

else p = 0;

}

}

template<class T>

void BST<T>::iterativePostorder() {

Stack<BSTNode<T>\*> travStack;

BSTNode<T>\* p = root, \*q = root;

while (p != 0) {

for ( ; p->left != 0; p = p->left)

travStack.push(p);

while (p->right == 0 || p->right == q) {

visit(p);

q = p;

if (travStack.empty())

return;

p = travStack.pop();

}

travStack.push(p);

p = p->right;

}

}

template<class T>

void BST<T>::breadthFirst() {

Queue<BSTNode<T>\*> queue;

BSTNode<T> \*p = root;

if (p != 0) {

queue.enqueue(p);

while (!queue.empty()) {

p = queue.dequeue();

visit(p);

if (p->left != 0)

queue.enqueue(p->left);

if (p->right != 0)

queue.enqueue(p->right);

}

}

}

template<class T>

void BST<T>::MorrisInorder() {

BSTNode<T> \*p = root, \*tmp;

while (p != 0)

if (p->left == 0) {

visit(p);

p = p->right;

}

else {

tmp = p->left;

while (tmp->right != 0 &&// go to the rightmost node of

tmp->right != p) // the left subtree or

tmp = tmp->right; // to the temporary parent of p;

if (tmp->right == 0) { // if 'true' rightmost node was

tmp->right = p; // reached, make it a temporary

p = p->left; // parent of the current root,

}

else { // else a temporary parent has been

visit(p); // found; visit node p and then cut

tmp->right = 0; // the right pointer of the current

p = p->right; // parent, whereby it ceases to be

} // a parent;

}

}

template<class T>

void BST<T>::MorrisPreorder() {

BSTNode<T> \*p = root, \*tmp;

while (p != 0) {

if (p->left == 0) {

visit(p);

p = p->right;

}

else {

tmp = p->left;

while (tmp->right != 0 &&// go to the rightmost node of

tmp->right != p) // the left subtree or

tmp = tmp->right; // to the temporary parent of p;

if (tmp->right == 0) { // if 'true' rightmost node was

visit(p); // reached, visit the root and

tmp->right = p; // make the rightmost node a temporary

p = p->left; // parent of the current root,

}

else { // else a temporary parent has been

tmp->right = 0; // found; cut the right pointer of

p = p->right; // the current parent, whereby it ceases

} // to be a parent;

}

}

}

template<class T>

void BST<T>::MorrisPostorder() {

BSTNode<T> \*p = new BSTNode<T>(), \*tmp, \*q, \*r, \*s;

p->left = root;

while (p != 0)

if (p->left == 0)

p = p->right;

else {

tmp = p->left;

while (tmp->right != 0 &&// go to the rightmost node of

tmp->right != p) // the left subtree or

tmp = tmp->right; // to the temporary parent of p;

if (tmp->right == 0) { // if 'true' rightmost node was

tmp->right = p; // reached, make it a temporary

p = p->left; // parent of the current root,

}

else { // else a temporary parent has been found;

// process nodes between p->left (included) and p (excluded)

// extended to the right in modified tree in reverse order;

// the first loop descends this chain of nodes and reverses

// right pointers; the second loop goes back, visits nodes,

// and reverses right pointers again to restore the pointers

// to their original setting;

for (q = p->left, r = q->right, s = r->right;

r != p; q = r, r = s, s = s->right)

r->right = q;

for (s = q->right; q != p->left;

q->right = r, r = q, q = s, s = s->right)

visit(q);

visit(p->left); // visit node p->left and then cut

tmp->right = 0; // the right pointer of the current

p = p->right; // parent, whereby it ceases to be

} // a parent;

}

}

template<class T>

void BST<T>::balance (T data[], int first, int last) {

if (first <= last) {

int middle = (first + last)/2;

insert(data[middle]);

balance(data,first,middle-1);

balance(data,middle+1,last);

}

}

#endif