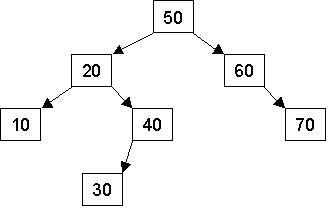
Jason Tieu

304047667

**Homework 5**

1. Consider the following binary search tree, ordered using the < relationship:



1. Using the simplest binary search tree (BST) insertion algorithm (no balancing), show the tree that results after inserting into the above tree the nodes 80, 65, 75, 45, 35 and 25 in that order.

50

/\

20 60

/\ \

10 40 70

/\ /\

30 45 65 80

/\

25 35

1. After inserting the nodes mentioned in part a, what is the resulting BST after you delete the node 30, then the node 20?

**Deleting 30 first:**

50

/\

20 60

/\ \

10 40 70

/\ /\

35 45 65 80

/

25

**After deleting 30, now delete 20:**

50

/\

10 60

\ \

40 70

/\ /\

35 45 65 80

/

25

1. After inserting the nodes mentioned in part a, what would be printed out by in-order, pre-order, and post-order traversals of the tree (assume your traversal function prints out the number at each node as it is visited)?

**In-order:** 10 -> 20 ->25 -> 30 -> 35 -> 40 -> 45 -> 50 -> 60 -> 65 -> 70 -> 80

**Pre-order:** 50 -> 20 -> 10 -> 40 -> 30 -> 25 -> 35 -> 45 -> 60 -> 70 -> 65 -> 80

**Post-order:** 10 -> 25 -> 35 -> 30 -> 45 -> 40 -> 20 -> 65 -> 80 -> 70 -> 60 -> 50

2. Consider the following operations on an initially empty heap h ordered by the < relationship. (This heap is a maxheap: the biggest item is at the top). The heap is represented as a binary tree:

1. h.insert(3);
2. h.insert(5);
3. h.insert(2);
4. h.insert(1);
5. h.insert(10);
6. h.insert(4);
7. int item;
8. h.remove(item); // Removes the biggest item from the heap, and puts it in item
9. h.insert(8);
10. h.insert(6);
11. h.remove(item);
12. Show the resulting heap (As in problem 1a, show the tree in some recognizable form.)

8

/\

3 6

/\ /\

1 2 4 5

1. Show how your heap from part a would be represented in an array.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Value | 8 | 3 | 6 | 1 | 2 | 4 | 5 |

1. Remove the top item from the heap and show the resulting array after the removal operation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
| Value | 6 | 3 | 5 | 1 | 2 | 4 |

3. In some binary search trees, each node has a left child pointer, a right child pointer and a parent pointer. The parent pointer of a node points to its parent (duh!), or is nullptr if the node is the root node. This problem will examine such trees.

1. Show a C++ structure/class definition for a binary tree node that has both child node pointers and a parent node pointer. Assume the data stored in each node is an int.

struct BinaryTreeNode

{

int data;

BinaryTreeNode\* parent;

BinaryTreeNode\* leftChild;

BinaryTreeNode\* rightChild;

BinaryTreeNode(int d)

{

data = d;

parent = leftChild = rightChild == nullptr;

}

};

1. Write pseudocode to insert a new node into a binary search tree with parent pointers. (Hint: You can find binary search tree insertion code on pp. 543-545 & 560-561 of the course textbook).

BinaryTreeNode\* FindNode(BinaryTreeNode\* p, int d)

{

//base cases

if p has no children

return p

if p's data equal to d

return p

if d is greater than p's data and p has a right child

return FindNode on p's right child

if d is less than p's data and p has a left child

return FindNode on p's left child

else

return p

}

void insertTreeNode(int new\_DATA)

{

if root of tree is null

set root to new node containing new\_DATA

return

set temp\_parent to FindNode(uses tree root and new\_DATA)

if FindNode does not return null

if key greater temp\_parent's key

set temp\_parent's right child to new node

set new node's parent to temp\_parent

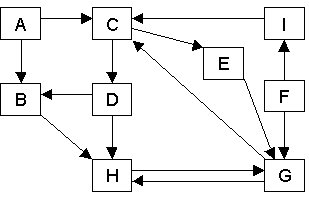
if key less than temp\_parent's key

set temp\_parent's left child to new node

set new node's parent to temp\_parent

}

4a. Show an adjacency matrix and an adjacency list for the following graph.



**Adjacency Matrix:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| To\from | A | B | C | D | E | F | G | H | I |
| A | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| C | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| F | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| G | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| H | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| I | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

**Adjacency List:**

A → B, C

B→ H

C→ D, E

D→ A, H

E→ G

F→ G, I

G→ C, H

H→ G

I→ C

4b. If you perform a depth-first traversal through this graph starting from vertex E, what vertices are visited, and in what order? There is more than one answer to this question, so list all the answers.

a) E → G → C → D → H

b) E → G → C → D → B → H

c) E → G → H