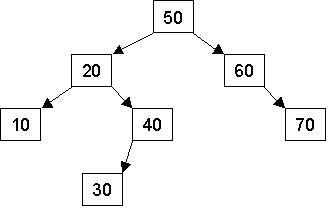
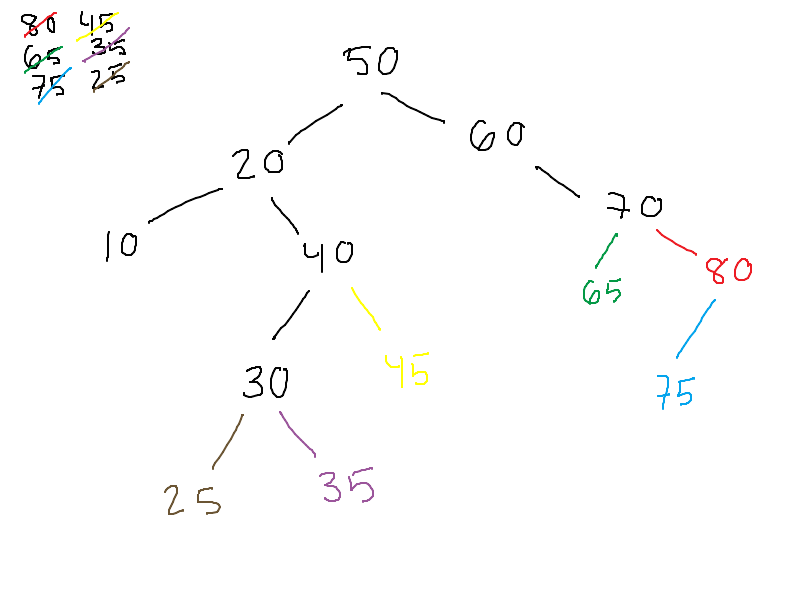
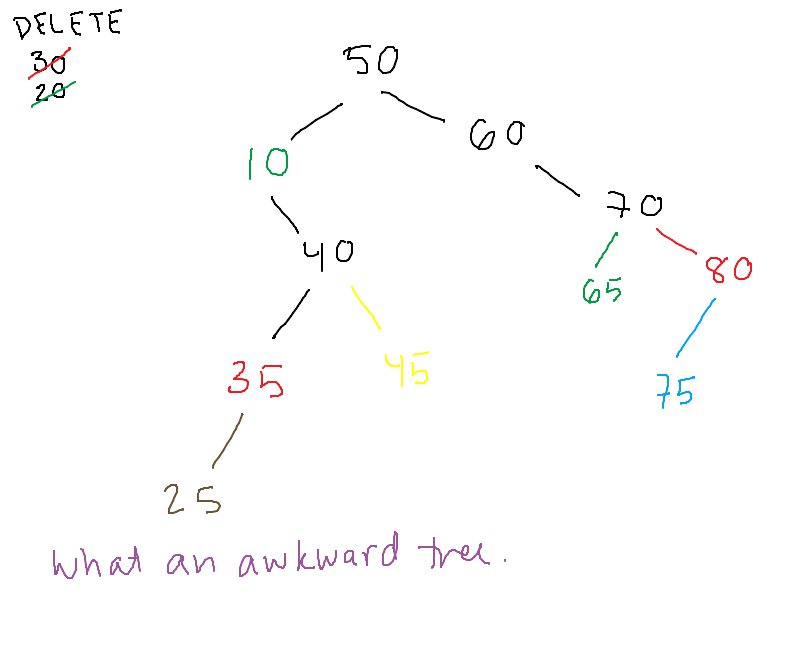
hw.doc

1.



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.



1. After inserting the nodes mentioned in part a, what is the resulting BST after you delete the node 30, then the node 20? (Again, just use a simple deletion algorithm with no balancing. If you have an option of making a choice, any correct choice is acceptable.) 
2. 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: LNR

10, 25, 35, 40, 45, 50, 60, 65, 70, 75, 80

Pre-order: NLR

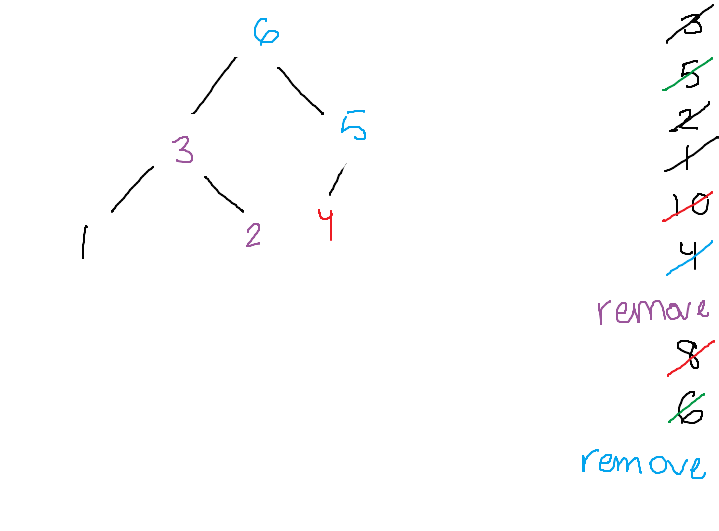
50, 10, 40, 35, 25, 45, 60, 70, 65, 80, 75

Post-order: LRN

25, 35, 45, 40, 10, 65, 75, 80, 70, 60, 50

2.

a. Show heap



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

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

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 5 | 3 | 4 | 1 | 2 |

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.

class weirdNode {

public:

Node(int value)

: m\_parent(nullptr), m\_Lchild(nullptr), m\_Rchild(nullptr), m\_value(value)

{ }

Node(int value, Node\* parent, Node\* Lchild, Node\* Rchild)

: m\_value(value), m\_parent(parent), m\_Lchild(Lchild),

m\_Rchild(Rchild)

{ }

void setParent(Node\* thing) { m\_parent = thing; }

Node\* getParent(Node\* thing) { return m\_parent; }

void setLchild(Node\* thing) { m\_Lchild = thing; }

Node\* getLchild(Node\* thing) { return m\_Lchild; }

void setRchild(Node\* thing) { m\_Rchild = thing; }

Node\* getRchild(Node\* thing) { return m\_Rchild; }

~Node() { }

private:

int m\_value;

Node\* m\_parent;

Node\* m\_Lchild;

Node\* m\_Rchild;

}; // semi colooooon

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. 471-473).

insertInorder(Node\* subTreePtr, Node\* newNodePtr, Node\* parentPtr)

{

if (subTreePtr == nullptr)

return newNodePtr;

else if (subTreePtr->getItem() > newNodePtr->getItem())

{

Node\* tempPtr = insertInorder(subTreePtr->getLchild(), newNodePtr, subTreePtr);

subTreePtr->setLchild(tempPtr);

tempPtr->setParent(subTreePtr);

}

else

{

Node\* temp = insertInorder(subTreePtr->getRchild(), newNodePtr, subTreePtr);

subTreePtr->setRchild(temp);

temp->setParent(subTreePtr);

}

}