Written Problems

Problem Set 4
November 16, 2011
Graham Schmidt
schmidtg@gmail.com
Harvard ID: 30825489

1. Uninformed state-space search

a. State order for BFS

```
6 5 8    6 5 8    6 5 8    6 _ 8    6 5 8    6 5 8
7 _ 1    _ 7 1    7 1 _    7 5 1    7 4 1    7 _ 1
3 4 2    3 4 2    3 4 2    3 4 2    3 _ 2    3 4 2
```

b. State order for DFS

```
6 5 8    6 5 8    6 5 8    _ 5 8    6 5 8    6 5 8
7 _ 1    _ 7 1    7 _ 1    6 7 1    3 7 1    7 1 _
3 4 2    3 4 2    3 4 2    3 4 2    _ 4 2    3 4 2
```

c. State order for IDS

```
6 5 8    6 5 8    6 5 8    6 5 8    6 _ 8    6 5 8
7 _ 1    7 _ 1    _ 7 1    7 1 _    7 5 1    7 4 1
3 4 2    3 4 2    3 4 2    3 4 2    3 4 2    3 _ 2
```

2. Determining the depth of a node.

a. For a binary tree with n nodes, the time complexity is:

Best case:

O(1) – key we're looking for is at the root

Worst case (not balanced):

O(n) – the tree has only right or left nodes, and to find the depth of the last key, we would need to traverse through all nodes to find the last key. In this case the tree would essentially be a linked list.

Worst case (balanced):

O(n) – if the tree is balanced, this algorithm will still traverse all of the left branches first, and then all of the right branches. If our key to search for is the last item in the right branches, then we'll still have traversed through all of the nodes.

b. Revised algorithm for a binary search tree

```
public int depth(int key) {
    if (root == null) {
        throw new IllegalStateException(“the tree is empty”);
    return depthInTree(key, root);
}

private static int depthInTree(int key, Node root) {
    if (key == root.key)
        return 0;

    if (key < root.key) {  // Check current key
        if (root.left != null) {
            int depthInLeft = depthInTree(key, root.left);
            if (depthInLeft != -1)
                return depthInLeft + 1;
        }
    } else {
        if (root.right != null) {
            int depthInRight = depthInTree(key, root.right);
            if (depthInRight != -1)
                return depthInRight + 1;
        }
    }

    return -1;
}
```

c. For a binary search tree with n nodes, the time complexity is:

Best case:

O(1) – key we’re looking for is at the root

Worst case (not balanced):

O(n) – the tree has only right (or left) nodes due to the order in which the nodes were inserted. To find the depth of say the largest key, we would need to traverse through all the nodes (all right branches) to find the largest key. In this case the tree would essentially be a linked list.

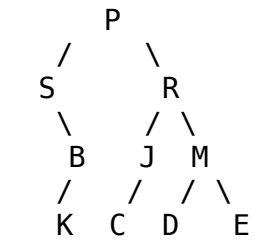
Worst case (balanced):

O(logn) – if the tree is balanced, this algorithm takes advantage of the fact it can ignore close to half of the nodes during each traversal. If the item we’re searching for is continually greater than each root node, we get to skip over half the values.

3. Tree traversal puzzles

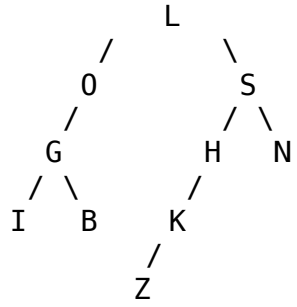
a.

Inorder traversal: SKBPCJRDME
Preorder traversal: PSBKRJCMDE

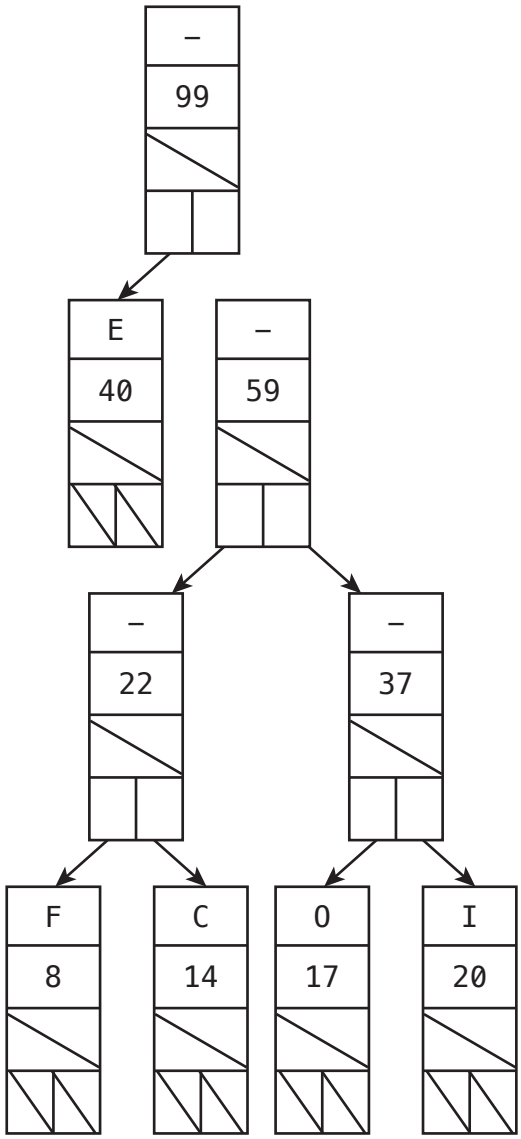


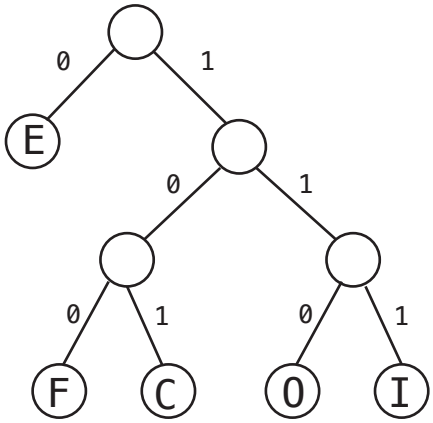
b.

Postorder traversal: IBGOZKHNSL
Preorder traversal: LOGIBSHKZN



4. a) Huffman encoding



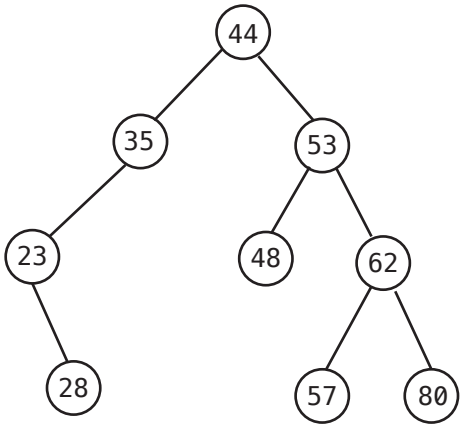


b) Office: 1101001001111010

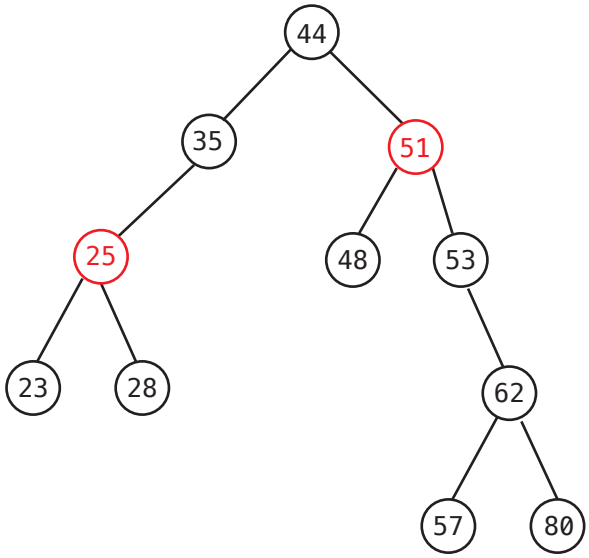
5. a) Preorder: 44 35 23 28 53 48 62 57 80

b) Postorder: 28 23 35 48 57 80 62 53 44

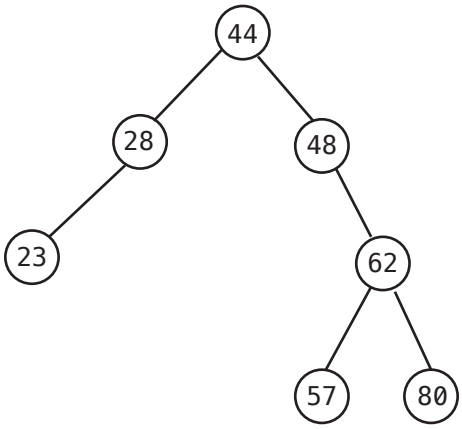
c) Before



After



d) Show tree after 53 and 35 are deleted



e) A tree is balanced if, for each node, the node's subtrees have the same height or have heights that differ by 1. The original tree is not balanced as the right subtree for node 35 has a height of 2, whereas it's left subtree has a height of 0.

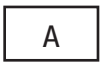
6. a) Insert keys A, D, G, B, F, C, H, I, E, J into an initially empty 2–3 tree.

Insert ‘A’

BEFORE

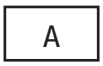
empty

AFTER



Insert ‘D’

BEFORE

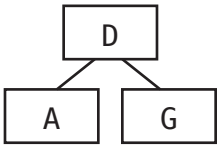


AFTER

Insert ‘G’

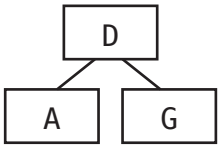
BEFORE

AFTER

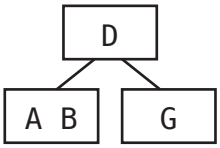


Insert ‘B’

BEFORE

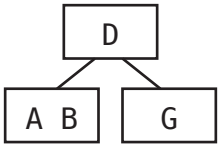


AFTER

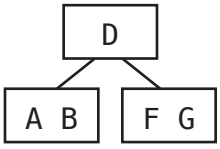


Insert ‘F’

BEFORE

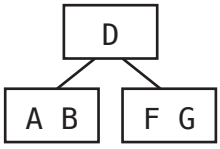


AFTER

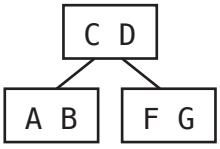


Insert ‘C’

BEFORE

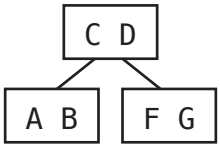


AFTER

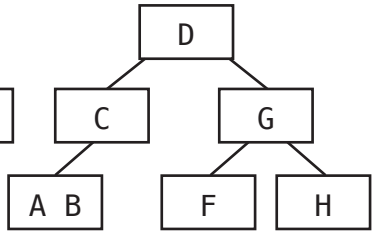


Insert ‘H’

BEFORE

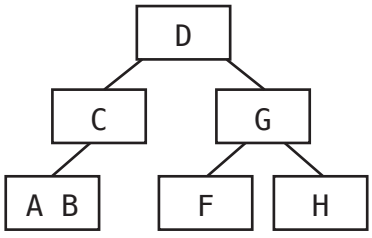


AFTER

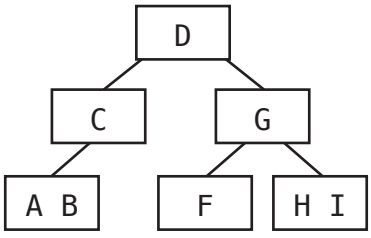


Insert ‘I’

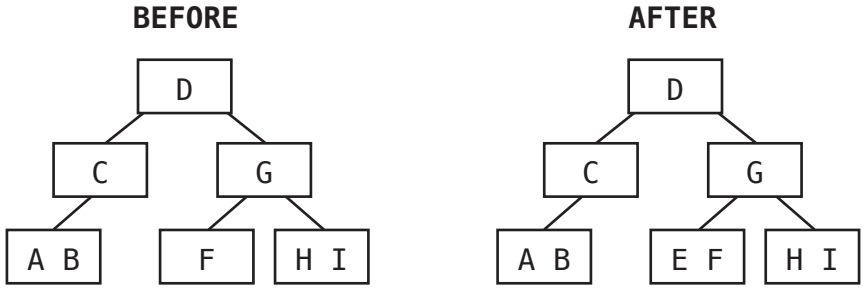
BEFORE



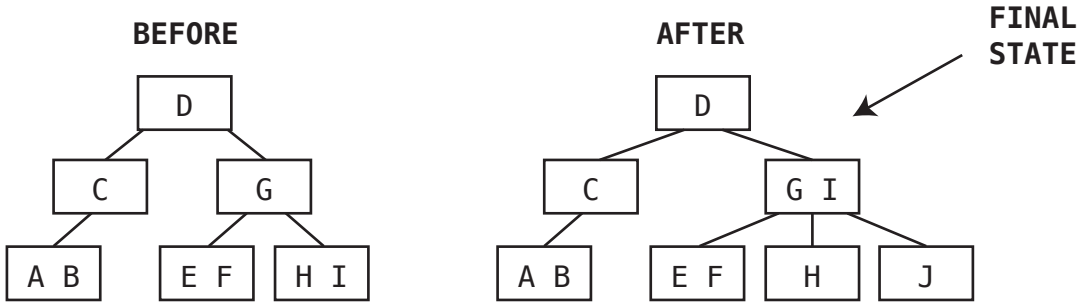
AFTER



Insert ‘E’



Insert ‘J’



6. b) Insert keys A, D, G, B, F, C, H, I, E, J into an initially empty B–tree of order 2.

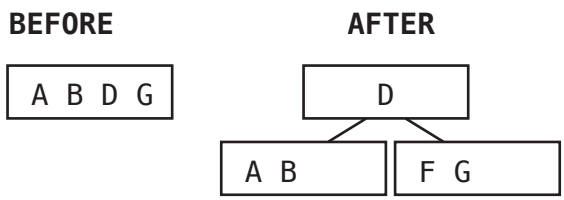
Insert ‘A’

Insert ‘D’

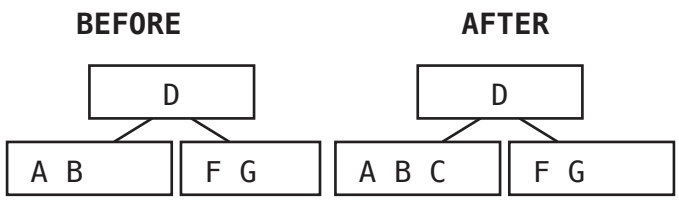
Insert ‘G’

Insert ‘B’

Insert ‘F’

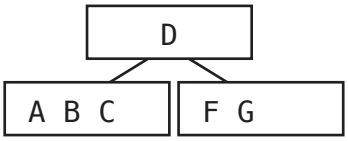


Insert ‘C’

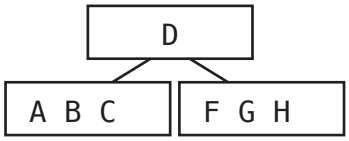


Insert ‘H’

BEFORE

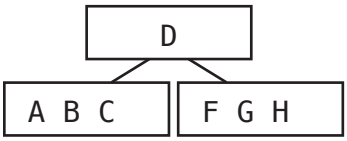


AFTER

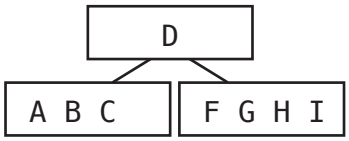


Insert ‘I’

BEFORE

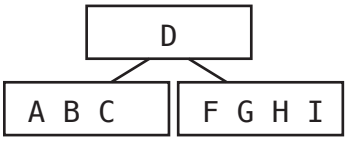


AFTER

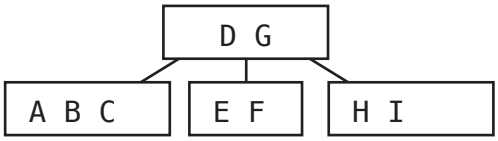


Insert ‘E’

BEFORE

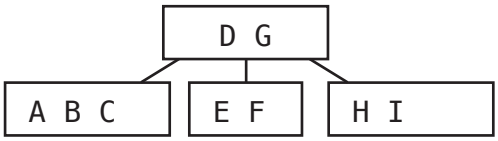


AFTER

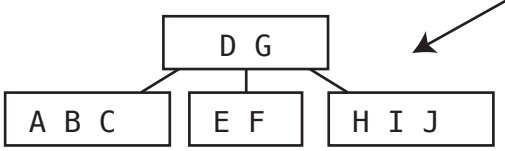


Insert ‘J’

BEFORE



AFTER



FINAL STATE