CS 130A Data Struc & Alg 1

Binary Search Trees, AVL Trees

This project requires you to implement operations on unbalanced binary search trees and AVL trees. You have to implement, analyze and compare the performances of a sequence of insertions and deletions on these search tree structures. You must write a test program to perform testing on a set of N distinct integers in the following ways:

- Insert N integers in increasing order, access them and delete them in the same order. (All inserts followed by accesses followed by deletes)
- Insert N integers in increasing order and delete them in the reverse order. (All inserts followed by accesses followed by deletes)
- Insert N integers in random order and delete them in random order. (All inserts followed by accesses followed by deletes)

The size of input N can be taken from 1000 to 10000.The run times must be measured and plotted with respect to the sizes to illustrate the difference. Also provide a short analysis of the runtimes(1 paragraph).

Grading Policy

- **Programming:** This is not a group assignment; turn in your own work. Implement all necessary operations (in order to perform the sequence of operations given above) on unbalanced binary search trees, AVL trees (80 points, 40 points for each).
- **Testing and report:** Test the trees you implemented and plot the run times with regard to different input sizes. Write a one paragraph analysis. Make a pdf file including the plot and the analysis and turn it in together with the code(20 points).

Implementation Details

You will need to provide the implementations for BST and AVL. You can assume that all the operations are performed on integers. You will need to implement the following operations on each of the trees :

- `insert i`: Insert the element in the tree and perform operations if necessary (For AVL tree, you will need to internally have rotate operations if tree becomes unbalanced at this step). Your program should print `Element inserted` or `Element already present`.
- `access i`: Lookup the element in the tree and your program should print `Element accessed` or `Element not found`.

- `delete i`: You need to delete the element in the tree and perform any operations if necessary. For a unbalanced BST, If the node to be deleted has 2 children, then take the strategy of replacing the data of this node with the smallest on the right subtree and than delete that node. Other scenarios are trivial (A node with 0 or 1 child to be deleted). AVL tree performs the deletion in the same way as a BST but will perform additional rebalancing if tree becomes unbalanced after deletion. Print the text `Element deleted` or `Element not found` (if element is not in the tree structure).

- `print`: You need to implement a print function which traverses the tree in pre order, in order and post order and prints out the tree elements. If the tree is empty, print `Empty tree` First print pre-order, then print in-order and finally print post-order.

  Note:
  Do not use recursive functions to implement the tree traversals while printing the tree elements.

  Extra Credit:
  Implement level order traversal or breadth first traversal as discussed which will be printed if input is "print bfs"

Example

The following are two test cases for both implementations.

Unbalanced BST Implementation

Sample Input

```
print
insert 50
insert 60
insert 40
insert 10
insert 20
access 10
print
insert 15
print
insert 40
insert 30
print
delete 40
print
insert 40
insert 70
print
delete 100
print
```

Sample Output

```
Empty tree
Element inserted
Element inserted
Element inserted
Element inserted
Element inserted
Element accessed
50 40 10 20 60
10 20 40 50 60
20 10 40 60 50
Element inserted
50 40 10 20 15 60
10 15 20 40 50 60
15 20 10 40 60 50
Element already present
Element inserted
50 40 10 20 15 30 60
10 15 20 30 40 50 60
15 30 20 10 40 60 50
Element deleted
50 10 20 15 30 60
10 15 20 30 50 60
15 30 20 10 60 50
Element inserted
Element inserted
50 10 20 15 30 40 60 70
10 15 20 30 40 50 60 70
15 40 30 20 10 70 60 50
Element not found
50 10 20 15 30 40 60 70
10 15 20 30 40 50 60 70
15 40 30 20 10 70 60 50
```

AVL Tree Implementation

Sample Input

```
print
insert 50
insert 60
insert 40
insert 10
insert 20
access 10
print
insert 15
print
```

```
insert 40
insert 30
print
delete 40
print
insert 40
insert 70
print
delete 100
print
```

Sample Output

```
Empty tree
Element inserted
Element inserted
Element inserted
Element inserted
Element inserted
Element accessed
50 20 10 40 60
10 20 40 50 60
10 40 20 60 50
Element inserted
20 10 15 50 40 60
10 15 20 40 50 60
15 10 40 60 50 20
Element already present
Element inserted
20 10 15 50 40 30 60
10 15 20 30 40 50 60
15 10 30 40 60 50 20
Element deleted
20 10 15 50 30 60
10 15 20 30 50 60
15 10 30 60 50 20
Element inserted
Element inserted
20 10 15 50 30 40 60 70
10 15 20 30 40 50 60 70
15 10 40 30 70 60 50 20
Element not found
20 10 15 50 30 40 60 70
10 15 20 30 40 50 60 70
15 10 40 30 70 60 50 20
```

General Issues

The TA/Readers should be able to run your unbalanced BST implementation as "`./bst`", AVL Tree implementation as "`./avl`" with input from stdin. Be sure to include a Makefile, and name the executable as described. The TA/Readers should be able to terminate your program by Ctrl-D (EOF for linux). When we test using files piped via cat, the EOF will be there.

The programs are tested automatically so make sure that you provide the output exactly as specified and shown above with no extra space lines to the output. The program should output to stdout. The input provided above is the subset of the input your program would be tested against. So be sure to check your program exhaustively for different cases.

This programming assignment is to be implemented in C++.

Submission

You need to submit all the files having your implementation, the Makefile and the experiment report pdf

```
turnin PA2@cs130a [list of files to be submitted]
```

Notice that when doing the turnin, just provide a list of files. Do not create a directory and place all the files in the directory and then turnin the directory