

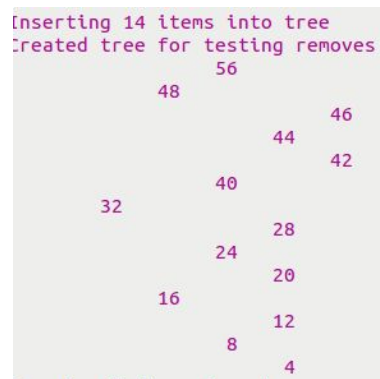
## Testing Unit Drivers and Explanations& Performance Evaluation:

6 examples of Unit Drivers that were tested with various conditions.

1.

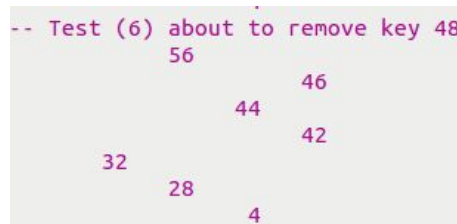
```
if (UnitNumber == 0) /* enabled with -u flag */
{
    // example test to remove leaves, 12 and 20, then internal nodes
    // 8, 24, 40 with one child, then 16, 48 with two children
    const int ins[] = {32,16,8,24,4,12,20,28,48,40,56,44,42,46};
    const int del[] = {12,20,8,24,40,16,48};
    unitDriver(ins, sizeof ins / sizeof(int),
              del, sizeof del / sizeof(int));
}
```

This unit driver is a simple add and delete of nodes and leaves. The program correctly created the binary search tree and prints it out in the terminal. At the end the provided del[] numbers are removed from the binary search tree and configured in the correct order.



beginning:

end:

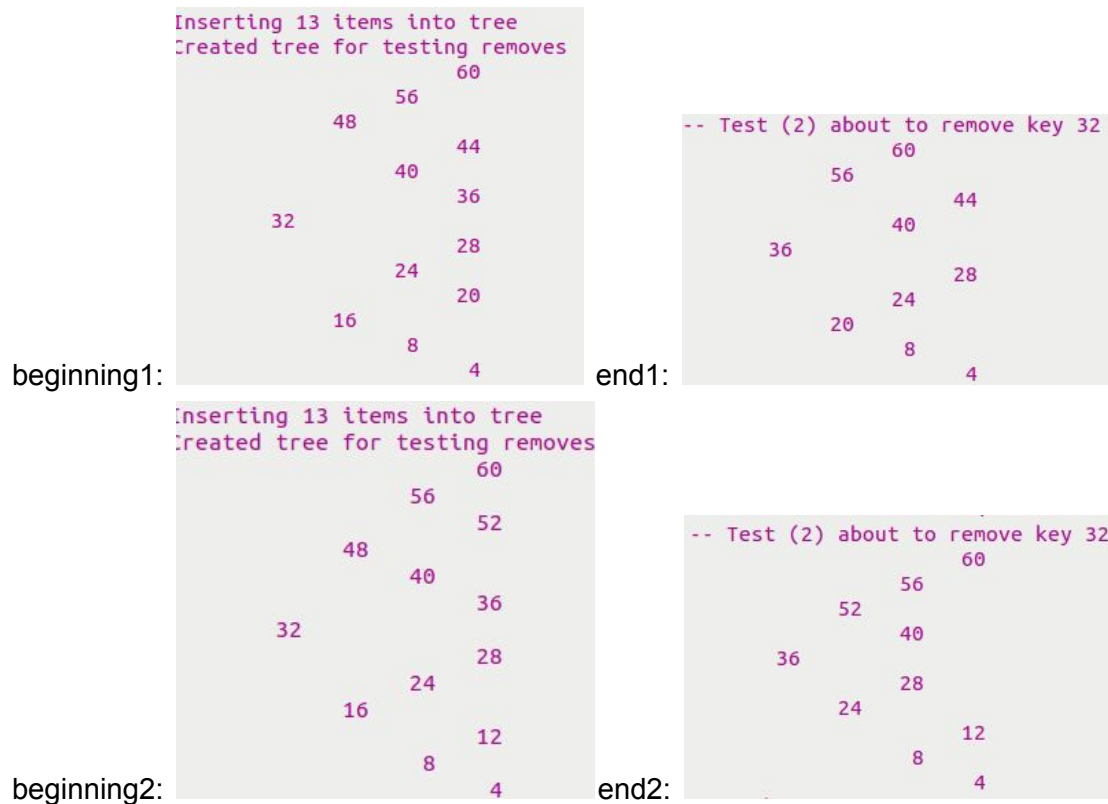


2.

```
if (UnitNumber == 1)
{
    // example tests: (48) is missing its right-left child and
    //                  (16) is missing its left-right child
    const int ins1[] = {32,16,48,8,24,40,56,4,20,28,36,44,60};
    const int del1[] = {16,48,32};
    unitDriver(ins1, sizeof ins1 / sizeof(int),
              del1, sizeof del1 / sizeof(int));

    // example tests: (16) is missing its right-left child and
    //                  (48) is missing its left-right child
    const int ins1b[] = {32,16,48,8,24,40,56,4,12,28,36,52,60};
    const int del1b[] = {16,48,32};
    unitDriver(ins1b, sizeof ins1b / sizeof(int),
              del1b, sizeof del1b / sizeof(int));
}
```

This unit driver tests if the program can successfully make two binary trees when asked and tests if a few specific nodes are missing their right or left child.



From the results of this unit driver you can see that the program successfully created two binary search trees in which both commands and actions were done successfully when inserting and then deleting. Everything is in order.

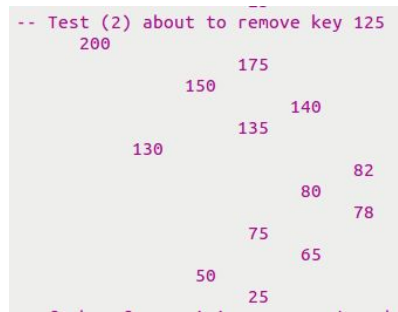
3.

```

if (UnitNumber == 2)
{
    // example deletion with many children
    const int ins[] = {200,100,50,150,25,75,125,175,65,85,135,80,130,140,78,82};
    const int del[] = {100,85,125};
    unitDriver(ins, sizeof ins / sizeof(int),
              del, sizeof del / sizeof(int));
}

```

This example just makes use of many children in the binary search tree and sees if the output is as expected.



beginning:

end:

This program is successfully able to handle a large number of inputs in the binary search tree and remove all that was asked and still be in order.

4.

```
if (UnitNumber == 3)
{
    // check replace for duplicate key
    const int ins[] = {10, 10};
    const int del[] = {10};
    unitDriver(ins, sizeof ins / sizeof(int),
              del, sizeof del / sizeof(int));
}
```

This example tests the case where you have multiple tens as input, in which only one should remain, but this 10 is also removed. In this case 10 is the root node.

```
===== Unit Driver =====

Inserting 2 items into tree
Created tree for testing removes
10
Removing 1 items from tree
-- Test (0) about to remove key 10
```

beginning and end:

The picture above show that the program was successful in adding just one 10 with the last 10's data\_ptr and then removing without any problems.

5.

```
if (UnitNumber == 5)
{
    // root
    const int ins5[] = {100, 50, 125, 25, 75, 65, 60, 70, 110, 120, 115, 122};
    const int del5[] = {100};
    unitDriver(ins5, sizeof ins5 / sizeof(int),
              del5, sizeof del5 / sizeof(int));
}
```

This example inputs a large amount of nodes into the binary and attempts to remove just the root.

Inserting 12 items into tree  
Created tree for testing removes

```

      125
     /  \
    120  122
   /  \  /  \
  110 75 115
 /  \
100  65
   /  \
  50  70
   /  \
  25  60

```

Removing 1 items from tree  
-- Test (0) about to remove key 100

```

      125
     /  \
    120  122
   /  \  /  \
  110 75 115
     /  \
    50  70
    /  \
   25  60

```

beginning:

end:

The program is successfully able to input all the data into the BST and then just remove the root and sort in correct order.

6.

```

if (UnitNumber ==6){
    //left of parent
    const int ins6[] = {200, 100, 50, 125, 25, 75, 65, 60, 70, 110, 120, 115, 122};
    const int del6[] = {100};
    unitDriver(ins6, sizeof ins6 / sizeof(int),
              del6, sizeof del6 / sizeof(int));
}

```

This example inputs a large amount of nodes and attempts to take out one which have predecessor and successors that are far away.

Inserting 13 items into tree  
Created tree for testing removes

```

      200
     /  \
    125  122
   /  \  /  \
  120 110 115
 /  \
100  75
   /  \
  50  70
   /  \
  25  65
     \
      60

```

Removing 1 items from tree  
-- Test (0) about to remove key 100

```

      200
     /  \
    125  122
   /  \  /  \
  120 110 115
     /  \
    50  70
    /  \
   25  65
      \
       60

```

beginning:

end:

The program is able to successfully remove the 100 node even though the successor and predecessor were far away. The BST is still in order.

All of these unit drivers were executed without any memory leaks using valgrind. There were many other examples and testing cases that were executed but these were just some of the few.



Running the driver with an optimal driver gives the result of: (./lab5 -o)

```
----- Access driver -----
Access trials: 50000
Levels for tree: 16
Build optimal tree with size=65535
After access exercise, time=30.914, tree size=65535
  Expect successful search=1.00049, measured=35.9572, trials=24839
  Expect unsuccessful search=4.00043, measured=40.9986, trials=25161
----- End of access driver -----
```

Running the driver with a randomly generated tree gives the result of: (./lab5 -r)

```
----- Access driver -----
Access trials: 50000
Levels for tree: 16
Build random tree with size=65535
After access exercise, time=32.087, tree size=65535
  Expect successful search=1.00122, measured=49.3284, trials=25037
  Expect unsuccessful search=4.00116, measured=54.3999, trials=24963
----- End of access driver -----
```

Running the driver with a poor order for inserting keys gives the result of: (./lab5 -p)

```
----- Access driver -----
Access trials: 50000
Levels for tree: 16
Build poor tree with size=65535
After access exercise, time=98.098, tree size=65535
  Expect successful search=1.00803, measured=329.703, trials=24840
  Expect unsuccessful search=4.00797, measured=336.124, trials=25160
----- End of access driver -----
```

Running the equilibrium drivers gives the result of: (./lab5 -e)

```
----- Equilibrium test driver -----
Trials in equilibrium: 50000
Levels in initial tree: 16
Initial random tree size=65535
Expect successful search for initial tree=1.00122
Expect unsuccessful search for initial tree=4.00116
After exercise, time=54.37, new tree size=65449
successful searches during exercise=84.1671, trials=24998
unsuccessful searches during exercise=90.9784, trials=25002
Validating tree...passed
After access experiment, time=27.002, tree size=65449
Expect successful search=1.00116, measured=49.2587, trials=25046
Expect unsuccessful search=4.0011, measured=53.2613, trials=24954
----- End of equilibrium test -----
```

Standish explains about the expected values for the successful and unsuccessful searches. The measured values for each successful and unsuccessful search for the drivers were a lot higher than expected. This could be due to the implementation and design of my program however the program still successfully does its job. For the optimal and random trees as generated by the drivers, the optimal driver is closer to standish's expected value and is a better driver naturally since this is the "optimal" driver. The time it took to execute was also quicker than the random driver. For a worst case tree I expected the measured searches for successful and unsuccessful to be very high and the time to be much longer than a best case tree. My implementation successfully supports the claim that the successful search time has a complexity class of  $O(\log n)$  because if you count the nodes on each level starting with the root, and if each level has the max number of nodes, then you would be adding by multiples of 2. (i.e  $2^0 + 2^1 + 2^2 + \dots = h$ .) now if you solve with respect to  $n$  you get  $n = O(\log h)$ .