MP5 Test Plan. Casey Hird

Unit Drivers Testing

Unit Driver 0 (-u 0)

This is the provided default test driver. It builds, displays, and then removes from a tree with a few levels, adding and then removing some random value of nodes. We can see from this in the test log that our insertion and removal functions work properly in general.

Unit Driver 1 (-u 1)

This unit driver is like the previous one. We build a small tree of around a dozen nodes, then remove a few of these. This is another generic function that further increases our confidence in our code when not exposed to extreme cases.

Unit Driver 2 (-u 2)

This unit driver tests our ability to build and delete from a slightly larger tree. We construct a tree with around 15 nodes, then demonstrate that we can remove these correctly.

Unit Driver 3 (-u 3)

This unit driver tests the ability of our insertion function to handle a duplicate key. This driver is very simple—there are 2 insertions each with a key of 10 followed by a single deletion of key 10. In a binary search tree, we cannot have repeated keys, so this demonstrates our the ability to recognize when a key has been duplicated, and to direct the insertion to add its data to the already existing node rather that attempting to form a new node.

Unit Driver 4 (-u 4)

This unit driver is designed to demonstrate how our binary search tree handles a tree in the worst possible shape. The nodes are inserted in order of their keys, resulting in a list, the worst case for a binary search tree. We demonstrate then that even with this worst case, we can correctly insert and remove nodes as we would in an optimized tree.

Unit Driver 5 (-u 5)

In this unit driver we simply test some more extreme possible input values to our program. So, we show that we can insert vary large positive values, very large negative values, and 0. This test simply demonstrates that our program can handle input data in a wide range.

Unit Driver 6 (-u 6)

This unit driver checks the ability of our data structure to handle several attempts at repeated insertion and deletion. We insert data with the same key several times and see that there is still only a single node in the tree with that key, so additional insertions simply update the data in that node. Similarly, we see that we can attempts to remove the same node multiple times. More broadly, after the first attempt to remove a single node (which does work) we are showing that our removal function is capable of handling attempts to remove nodes with keys that are not found in the tree.

Unit Driver 7 (-u 7)

We use this unit driver to demonstrate that our remove function is easily capable of deleting the root of the tree. We build a tree of 7 nodes, then we remove the root of the tree each time until there are none remaining. In doing this, we also show that we can remove from a tree with a single node, leaving behind an empty tree.

Unit Driver 8 (-u 8)

This unit driver tests our removal function, ensuring that we can remove leaves. To do this, we create a tree with several nodes, then remove them in an order such that each node is removed from the tree when it is a leaf.

Summary

We use these unit drivers to test specific functionality and boundary cases of our insertion and deletion functions. The output from each test, which includes a print out of the tree after each removal, can be seen in the test log.

Valgrind

Each of these unit test drivers was run along with valgrind so we could gauge the performance of our insertion, deletion, and access functions and how they maintained memory. The output from all these test cases is condensed below, where we can see that we have detected no memory leaks or other issues with memory.

Unit Driver 0:

```
==7464== Memcheck, a memory error detector

==7464== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.

==7464== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==7464== Command: ./lab5 -u 0

==7464== Parent PID: 7463

==7464==

==7464==

==7464== HEAP SUMMARY:

==7464==     in use at exit: 0 bytes in 0 blocks

==7464==   total heap usage: 30 allocs, 30 frees, 4,624 bytes allocated

==7464==

==7464== All heap blocks were freed -- no leaks are possible
```

```
==7464==
==7464== For counts of detected and suppressed errors, rerun with: -v
==7464== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 1:

```
==7465== Memcheck, a memory error detector
==7465== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7465== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7465== Command: ./lab5 -u 1
==7465== Parent PID: 7463
==7465==
==7465==
==7465== HEAP SUMMARY:
==7465==     in use at exit: 0 bytes in 0 blocks
==7465==   total heap usage: 55 allocs, 55 frees, 5,080 bytes allocated
==7465==
==7465== All heap blocks were freed -- no leaks are possible
==7465==
==7465== For counts of detected and suppressed errors, rerun with: -v
==7465== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 2:

```
==7466== Memcheck, a memory error detector
==7466== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7466== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7466== Command: ./lab5 -u 2
==7466== Parent PID: 7463
==7466==
==7466==
==7466== HEAP SUMMARY:
==7466==     in use at exit: 0 bytes in 0 blocks
==7466==   total heap usage: 34 allocs, 34 frees, 4,696 bytes allocated
==7466==
==7466== All heap blocks were freed -- no leaks are possible
==7466==
==7466== For counts of detected and suppressed errors, rerun with: -v
==7466== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 3:

```
==7467== Memcheck, a memory error detector
==7467== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7467== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7467== Command: ./lab5 -u 3
==7467== Parent PID: 7463
==7467==
==7467==
==7467== HEAP SUMMARY:
==7467==     in use at exit: 0 bytes in 0 blocks
==7467==   total heap usage: 5 allocs, 5 frees, 4,160 bytes allocated
==7467==
==7467== All heap blocks were freed -- no leaks are possible
==7467==
==7467== For counts of detected and suppressed errors, rerun with: -v
==7467== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 4:

```
==7468== Memcheck, a memory error detector
==7468== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
```

```
==7468== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7468== Command: ./lab5 -u 4
==7468== Parent PID: 7463
==7468==
==7468==
==7468== HEAP SUMMARY:
==7468==     in use at exit: 0 bytes in 0 blocks
==7468==   total heap usage: 18 allocs, 18 frees, 4,408 bytes allocated
==7468==
==7468== All heap blocks were freed -- no leaks are possible
==7468==
==7468== For counts of detected and suppressed errors, rerun with: -v
==7468== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 5:

```
==7469== Memcheck, a memory error detector
==7469== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7469== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7469== Command: ./lab5 -u 5
==7469== Parent PID: 7463
==7469==
==7469==
==7469== HEAP SUMMARY:
==7469==     in use at exit: 0 bytes in 0 blocks
==7469==   total heap usage: 12 allocs, 12 frees, 4,300 bytes allocated
==7469==
==7469== All heap blocks were freed -- no leaks are possible
==7469==
==7469== For counts of detected and suppressed errors, rerun with: -v
```

```
==7469== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 6:

```
==7470== Memcheck, a memory error detector
==7470== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7470== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7470== Command: ./lab5 -u 6
==7470== Parent PID: 7463
==7470==
==7470==
==7470== HEAP SUMMARY:
==7470==     in use at exit: 0 bytes in 0 blocks
==7470==   total heap usage: 21 allocs, 21 frees, 4,364 bytes allocated
==7470==
==7470== All heap blocks were freed -- no leaks are possible
==7470==
==7470== For counts of detected and suppressed errors, rerun with: -v
==7470== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 7:

```
==7471== Memcheck, a memory error detector
==7471== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7471== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7471== Command: ./lab5 -u 7
==7471== Parent PID: 7463
==7471==
==7471==
==7471== HEAP SUMMARY:
==7471==     in use at exit: 0 bytes in 0 blocks
```

```
==7471==   total heap usage: 16 allocs, 16 frees, 4,372 bytes allocated
==7471==
==7471== All heap blocks were freed -- no leaks are possible
==7471==
==7471== For counts of detected and suppressed errors, rerun with: -v
==7471== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Unit Driver 8:

```
==7472== Memcheck, a memory error detector
==7472== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7472== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7472== Command: ./lab5 -u 8
==7472== Parent PID: 7463
==7472==
==7472==
==7472== HEAP SUMMARY:
==7472==     in use at exit: 0 bytes in 0 blocks
==7472==   total heap usage: 16 allocs, 16 frees, 4,372 bytes allocated
==7472==
==7472== All heap blocks were freed -- no leaks are possible
==7472==
==7472== For counts of detected and suppressed errors, rerun with: -v
==7472== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Access Drivers Testing

The first section of our access drivers has no actual access testing but is instead just used to demonstrate proper functioning of insertion. Thus, our first three access drivers build 5 level trees and perform 0 accesses, the first tree built optimally, the second randomly, and the third poorly. We can see below the difference in the shapes of these trees, and we can see their shapes correspond to the optimization of the insertion used to build them.

Optimal Tree Output:

```
----- Access driver -----

Access trials: 0

  Levels for tree: 5

  Build optimal tree with size=31

                  62

              60

                  58

          56

                  54

              52

                  50

      48

                  46

              44

                  42

          40

                  38

              36

                  34

  32

                  30

              28

                  26

          24

                  22

              20
              18
     16
              14
           12
              10
         8
               6
            4
               2

              /---------------------32---------------------\
       /----------16----------\                   /----------48----------\
   /-----8----\         /----24----\         /----40----\         /----56----\
 /--4-\    /-12-\    /-20-\    /-28-\    /-36-\    /-44-\    /-52-\    /-60-\
 2    6   10   14   18   22   26   30   34   38   42   46   50   54   58   62
----- End of access driver -----

Random Tree Output:
----- Access driver -----
 Access trials: 0
 Levels for tree: 5
 Build random tree with size=31
           62
                 60
              58
                 56
        54
              52
                    50
```

```
                            48
                        46
                            44
                          42
                    40
                        38
                      36
                          34
                        32
                30
                    28
                  26
                        24
                      22
                        20
                            18
                          16
                            14
                    12
                        10
                      8
                        6
              4
                2

/--4-----------------------------------\
2                        /----30--------------------------------\
           /------------------26-\              /------------------54----------\
      /----12-------------\      28       /----40----------------\       /----62
```

```
     /--8-\         /-22-\        /----36-\        /-------52   /-58-\

     6   10      /----20   24        32-\    38    /----46----\     56   60

            /-16-\                  34       42-\    /-50

            14   18                           44   48

----- End of access driver -----


Poor Tree Output:

----- Access driver -----

 Access trials: 0

 Levels for tree: 5

 Build poor tree with size=31

        62

              60

                    58

                          56

                       54

                 52

           50

     48

           46

                 44

                       42

                          40

                    38

              36

        34

  32

                          30

                       28
```

```
                      26
                   24
                16
            20
         18
     16
         14
                12
                      10
                              8
                           6
                    4
             2

               /---------------------32---------------------\
             /-16-\                     /-----------------48-----------------\
 /---------------14   18-\                   34---------------\     /---------------62
 2------------\        20-\                   /------------46    50------------\
   /----------12         22-\                36----------\           /----------60
   4-------\              24-\                /-------44        52-------\
     /----10                26-\               38----\                /----58
     6-\                     28-\               /-42              54-\
       8                       30           40                    56
----- End of access driver -----
```

After demonstrating that we can build the desired shapes for each insertion method (optimized, random, and poor) we then ran access drivers with each of these along with different numbers of levels in the trees and 2 different insertion methods—standard binary search tree insertion, and AVL insertion. We can see the results of each of these in the test log, and they are summarized in the graphs shown below.

Optimal BST Insertion Access Times					
		Accesses			
		1000	10000	100000	1000000
	5	0.063	0.706	6.072	70.189
	10	0.106	1.078	10.417	104.07
	15	0.394	2.019	19.959	202.801
Levels	20	0.929	7.334	80.824	772.943

Optimal AVL Insertion Access Times					
		Accesses			
		1000	10000	100000	1000000
	5	0.063	0.651	5.928	59.046
	10	0.106	1.076	10.389	105.032
	15	0.243	1.989	20.887	203.43
Levels	20	0.931	7.303	74.702	786.092

Random BST Insertion Access Times					
		Accesses			
		1000	10000	100000	1000000
	5	0.06	0.61	5.732	56.387
	10	0.112	1.077	10.709	106.897
	15	0.24	2.375	23.918	242.582
Levels	20	1.031	9.482	100.533	1008.34

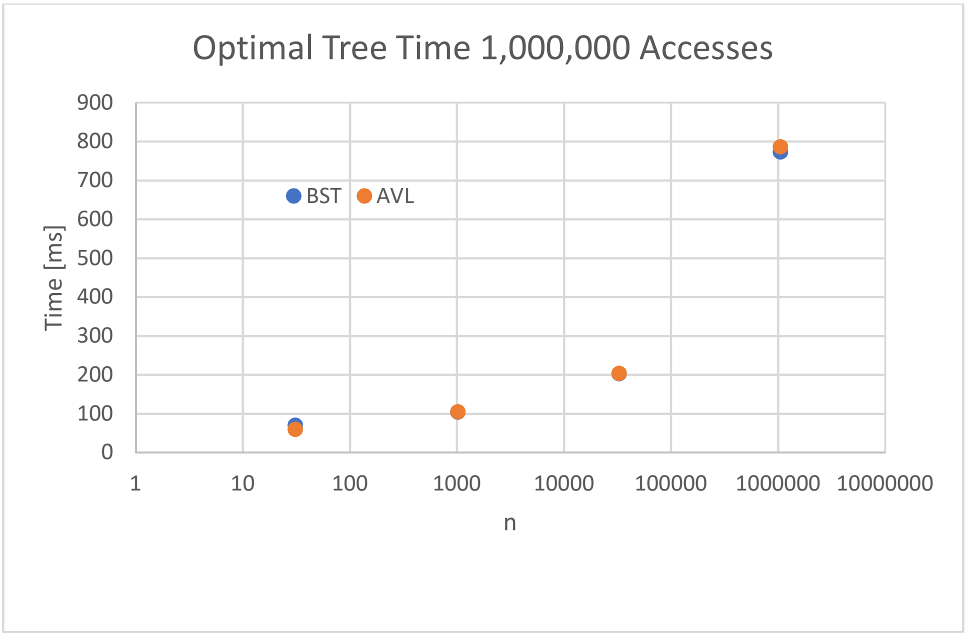
Random AVL Insertion Access Times					
		Accesses			
		1000	10000	100000	1000000
	5	0.061	0.606	6.119	58.529
	10	0.105	1.032	10.427	103.538
	15	0.209	2.083	21.25	212.916
Levels	20	0.844	8.577	86.407	856.622

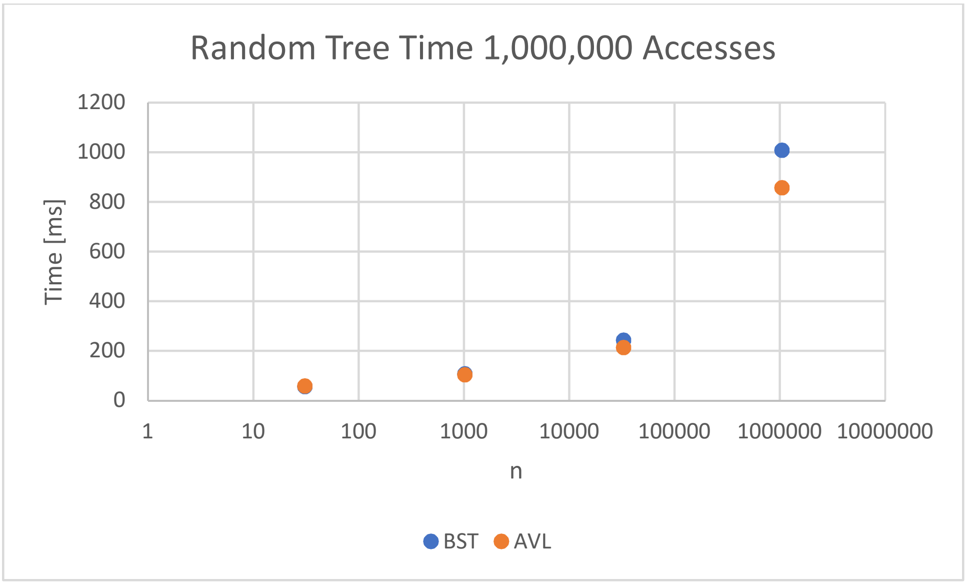
Poor BST Insertion Access Times					
		Accesses			
		1000	10000	100000	1000000
	5	0.075	0.758	7.424	61.094
	10	0.135	1.306	13.203	131.918
	15	0.711	6.489	64.535	664.33
Levels	20	3.868	40.182	359.708	3600

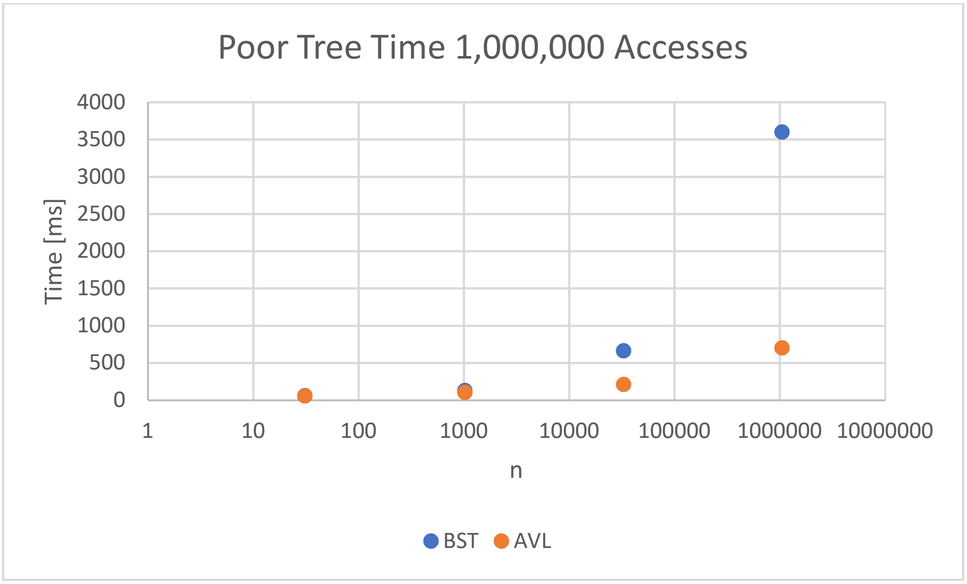
Poor AVL Insertion Access Times					
		Accesses			
		1000	10000	100000	1000000
	5	0.075	0.725	7.108	57.67
	10	0.106	1.089	10.42	103.292
	15	0.222	2.131	23.787	213.198
Levels	20	0.947	7.018	66.428	702.041

Performance Evaluation

We can analyze the performance of our binary search tree using the data given in the section above. We collect all of this into 3 plots showing the access times for trees of different sizes, one plot for each type of initial sorting (optimized, random, poor). These graphs are all seen below, each giving the time to perform 1,000,000 accesses on a binary search tree of size n.







We can see from the graphs above that we have roughly the results we expect. In the plots shown, the x axis is plotted on a logarithmic scale, so we expect that for O(logn) performance the plot would appear nearly linear. We see this most clearly in the optimized plot, which makes sense since, as an optimized tree, it will be nearest to the O(logn) optimal performance of a binary search tree. We do not have perfect linearity, so it is possible we have some time costing errors in our code, but we are very near optimal performance.

We also should note the difference between the plots of each sort type when insertions are simple BST and when they maintain the AVL property. In the optimized tree, we already maintain the AVL property, so there is no difference between the BST and AVL curves, as we can see in the graph. There is more of a pronounced difference, though, as we go to random insertion and then poor insertion. Observe that for the poor insertion, the AVL plot is nearly linear, as with the optimized plot, demonstrating that we have near optimal performance, while the BST plot increases much more quickly. This reflects what we would expect of a worst-case binary search tree—a BST in which every node has a single child, producing a list. For a list, since we must traverse an average of half of the elements in the list to insert a node, insertion is O(n). This is reflected in the poorly organized BST with simple BST insertion—the result of this is a data structure that is nearly a list, so the resulting insertion complexity for this will be near O(n).

In conclusion, we see that when we have a nearly optimized binary search tree, we have nearly optimized performance—insertion and access with O(logn) complexity. On the contrary, as we near a worst-case BST, which is equivalent to a list, we near the complexity O(n), that for a list.