Quarter 1 Final, Data Structures

Jay R Bolton

December 13, 2011

1. *(a)*

```
abcde
-f-g-
--h--
-i-j-
klmno
compact = [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o]
(each row concatenated)
```

(b) This is a generalized version. The data structure of "list" holds the compacted 1d x-matrix along with the size (max rows or max columns) of the matrix (`list.size`). The matrix is assumed always square. This could perhaps be simpler with a different compaction scheme, but for some reason I only considered the compaction scheme of concatenating each row.

```
In imperative pseudocode:

get(row, col, list)
  if(row == 1) # we're in first row
    return index(list, col)
  else if(row == list.rows) # we're in last row
    return index(list, 3*list.size - 5 + col)
  else if(row == col) # we're on right sloping diagonal
    return index(list, row + col + size - 4)
  else if(row+col == size) # we're on left sloping diagonal
    if(row < col)
      return index(list, row + (size+1 - col) + size - 3)
    else
      return index(list, row + (size+1 - col) + size - 5)
  else
    error("Invalid index")


In pseudo-Haskell:

get row col (XMatrix list size) =
 | row == 1        = list !! col
 | row == size     = list !! (3*size - 5 + col)
 | row == col      = list !! (row + col + size - 4)
 | row+col == size = let index = row + (size+1 - col) + size
                      in if (row < col) then list !! (index-3)
                         else list !! (index-5)
 | otherwise       = error "Invalid index"
```

2.

```
    A
   / \
  /   \
 /     \
B       D
 \     / \
  C   E   G
     /   / \
    F   H   J
       /
      I
```

```
The algorithm implemented in Haskell:

Tree a = Node a (Tree a, Tree a) | Nil
rtrav [] [] = Nil
rtrav (p:ps) in = let i = find p in
                      lin = take i in
                      rin = drop (i+1) in
                      lpre = take (length lin) ps
                      rpre = drop (length lin) ps
                   in Tree p ((rtrav lpre lin),(rtrav rpre rin))
```

3. *(a)*

Index	**Key**
0	38
1	14
2	11
3	42
4	
5	
6	
7	7
8	73
9	8
10	22
11	34
12	25

(b) Deletion of 73:

Index	**Key**
0	38
1	14
2	
3	42
4	
5	
6	
7	7
8	8
9	22
10	34
11	11
12	25

Deletion of 22:

Index	**Key**
0	38
1	14
2	
3	42
4	
5	
6	
7	7
8	8
9	34
10	
11	11
12	25

4. *(a)*

```
int computeS(TreeNode t) {
  if(t->left == NULL && t->right == NULL) /* we're at a leaf */
    return 0; /* base case */
  else if(t->left == NULL) /* branches right */
    return 1 + computeS(t->right);
  else if(t->right == null) /* branches left */
    return 1 + computeS(t->left);
  else /* two children */
    return 1 + min(computeS(t->left), computeS(t->right));
}
/*
I think this could be shorter.....
Another version:
*/
int computeS(TreeNode t) {
  if(t == NULL) return -1;
  else return 1 + min1(computeS(t->left), computeS(t->right));
}
int min1(int x, int y) {
  if(x == -1) return y;
  else if (y == -1) return x;
  else return x < y ? x : y;
}
/* This second version's base case is NULL trees and uses a strange
minimum function that favors the integer that is not negative one, so
for branches it chooses the non-null child and leaves become 0. */
```

(b) My first version is $\mathcal{O}(N)$ because it will visit every node in the tree.

The second version is also $\mathcal{O}(N)$ but is slightly worse if we consider constants because it has to visit the NULL children.

5. *(a)*

```
I'll do positive = more left
        negative = more right
       +1
      /  \
     /    \
    /      \
   +1        0
  /  \      /  \
 -1   +1   +1   0
 / \  /    /   / \
0  +1 0   0   0   0
   /
  0
```

Yes, it's an AVL search tree because all balance factors are less than |2|.

(b) Insert 13:

```
       30
      /   \
     /     \
    /       \
   25        50
  /  \      /  \
 10     27  40  60
 / \  /    /   / \
5  15 26  35   55  70
   /
  12
   \
    13
```

We find as we percolate up recursively that 15 is +2. So we do an LR rotation

```
       30
      /  \
     /    \
    /      \
   25       50
  /  \      /  \
 10     27  40  60
 / \   /   /   / \
5  13  26  35  55  70
   / \
  12  15
```

We continue to percolate and find that all is balanced. Then we insert 14

```
       30
      /  \
     /    \
    /      \
   25       50
  /  \      /  \
 10     27  40  60
 / \    /   /   / \
5  13   26  35  55  70
   / \
  12  15
     /
    14
```

We find that 10 is -2 We do a RR rotation

```
         30
        /  \
       /    \
      /      \
     25       50
    /  \      /  \
   13     27  40  60
   / \    /   /   / \
 10  15   26  35  55  70
 /\   /
5  12 14
```

All is balanced. We then insert 36.

```
         30
        /  \
       /    \
      /      \
     25       50
    /  \      /  \
   13     27  40  60
   / \    /   /   / \
 10  15   26  35  55  70
 /\   /       \
5  12 14       36
```

40 is imbalanced We do an LR

```
         30
        /  \
       /    \
      /      \
     25       50
    /  \      /  \
   13     27  36   60
   / \    /   / \  / \
 10  15   26 35 40 55  70
 /\   /
5  12 14
```

(6)
```
insert 10,9,8

    [8,9,10]
```

```
   insert 7,6

       [9]
       / \
     [8]  [10]

       [9]
       / \
   [6,7,8] [10]

   insert 5,4

       [7,9]
      /  |  \
   [6]  [8] [10]

       [7,9]
      /  |  \
[4,5,6] [8] [10]

    insert 3,2

      [5,7,9]
     /  | |  \
  [4] [6] [8] [10]

       [5,7,9]
     /  |   |  \
[2,3,4] [6] [8] [10]

    insert 1

        [7]
        /  \
    [3,5]   [9]
   /  | \   /  \
[2] [4] [6] [8] [10]

          [7]
        /    \
    [3,5]     [9]
   /   |  \    /  \
[1,2] [4] [6] [8] [10]
```

(b)

```
  () = black
  [] = red

          (7)
        /    \
    (3)      (9)
    /  \      /  \
 (1)   [5]  (8) (10)
   \   |  \
  [2] (4) (6)
```

(c) Delete 10,9,8,7 from the 2-3-4 tree.

```
delete 10

          [7]
        /    \
    [3,5]     [9]
   /   |  \    /  \
[1,2] [4] [6] [8] []

          [7]
        /    \
    [3,5]      []
   /   |  \    /  \
[1,2] [4] [6] [8,9] []

          [5]
        /     \
    [3]        [7]
   /   |  \    /
[1,2] [4] [6] [8,9]

delete 9

          [5]
        /     \
    [3]        [7]
   /   |  \    /
[1,2] [4] [6] [8]

delete 8

          [5]
        /     \
    [3]        [7]
   /   |  \    /
[1,2] [4] [6] []

          [5]
        /     \
    [3]        []
   /   |  \    /
[1,2] [4] [6] [7]

          []
        /
    [3,5]
   /   |  \
[1,2] [4] [6,7]

    [3,5]
   /   |  \
[1,2] [4] [6,7]
```

```
delete 7

    [3,5]
   /   |  \
[1,2] [4] [6]
```

(d) Couldn't find a very clear explanation of this. Since RB trees seem like a different representation of the same underlying concept in 2-3-4 trees, I'll just emulate the steps above, but with the red-black format.

```
delete 10
          (7)
        /    \
    (3)       (9)
    /  \      /  \
 (1)    [5]  (8) (10)
   \    |  \
  [2] (4) (6)

          (7)
        /    \
    (3)       (9)
    /  \      /  \
 (1)    [5]  (8) ()
   \    |  \
  [2] (4) (6)

          (7)
        /    \
    (3)       ()
    /  \      /
 (1)    [5]  (9)
   \    |  \    /
  [2] (4) (6) [8]

          (5)
        /    \
    (3)       (7)
    /  \      /
 (1)    [5]  (9)
   \    |  \    /
  [2] (4) (6) [8]

delete 9

          (5)
        /    \
    (3)       (7)
    /  \      /
 (1)    [5]  ()
   \    |  \    /
  [2] (4) (6) [8]

          (5)
        /    \
    (3)       (7)
    /  \      /
 (1)   [5]  (8)
   \   |  \
  [2] (4) (6)

delete 8

          (5)
        /    \
    (3)      (7)
    /  \     /
 (1)   [5]  ()
   \   |  \
  [2] (4) (6)

          (5)
        /    \
    (3)      ()
    /  \     /
 (1)   [5]  (7)
   \   |  \
  [2] (4) (6)

    (3)
    /  \
 (1)   [5]
   \   |  \
  [2] (4) (6)
            \
            [7]

delete 7

    (3)
    /  \
 (1)   [5]
   \   |  \
  [2] (4) (6)
```

7.

```
 1-15 binary tree

        8
       / \
      /   \
     /     \
    4        12
   /  \     /   \
  2    6   10   14
 / \  / \  / \   / \
1   3 5  7 9 11 13 15

 search for 15
 zag-zag

        8
       / \
       /   \
      /     \
     4       15
    /  \     /
   2    6   14
  / \  / \  /
 1   3 5  7 12
           /  \
         10   13
        /  \
       9   11

  zag

       15
      /
     /
     8
   /   \
  4      14
 /  \   /
2    6  12
/\  / \ / \
1 3 5 7 10 13
       / \
      9  11

or...
zag

         8
        / \
       /   \
      /     \
     4        12
    /  \     /   \
   2    6   10   15
  / \  / \  / \   /
 1   3 5  7 9 11 14
                 /
               13

zag-zag

         15
        /
      12
      / \
     8   14
    / \   /
   4  10 13
  /\  / \
 2 6  9 11
/\ /\
13 5 7
```

8.

```
int equal(List a, List b) {
  Iterator it_a = iterator(a);
  Iterator it_b = iterator(b);
  while(hasNext(it_a) && hasNext(it_b)) {
    if(equals(it_a, it_b)) {
      next(it_a); next(it_b);
    }
    else return 0;
  }
  if(hasNext(it_b) && !hasNext(it_a)) return 0;
  else if(hasNext(it_a) && !hasNext(it_b)) return 0;
  else return 1;
}
/* This could be a bit simplified if the lists store a length field */
```

The time complexity of this function is $\mathcal{O}(N)$ where N is the length of the input lists (particularly the shorter of the two)

I arrived at this by noting that the expressions before and after the loop are constant time. The expressions inside the loop are constant time. The loop itself executes N times, where N is the length of the shorter of the two lists (it breaks as soon as we reach the end of either). This function is faster when the lists are unequal earlier on and is slowest for equal lists or for lists with equal prefixes.