Adam Hayen

CS-203 Final Exam
Adam Hayen

1 (a)

The input size of this algorithm is the number of elements in int array A.

Best and worst case analyses are needed because the basic operation count depends on more than just the input size. The algorithm can finish with one basic operation if the first elements compared are not equal.

The basic operation is the comparison of two elements inside the nested for loop. This operation is the only one inside the nested for loop and can happen many times, so it is a good indicator of the efficiency of this algorithm.

1 (b)

The best-case scenario is when the first two elements that are compared are not equivalent. In this case, the algorithm returns false and is therefore completed.

$$C_{best}(n) = \sum_{i=0}^{1}[(\sum_{j=i+1}^{1} 1)] = 1 \in \Theta(1)$$

The worst case scenario is when the input array is symmetric, so the algorithm completes all possible loops.

$$C_{worst}(n) = \sum_{i=0}^{n-2}[(\sum_{j=i+1}^{n-1} 1)] = \frac{n(n-1)}{2} = \frac{(n^2 - n)}{2} \in \Theta(n^2)$$

So, in the general case scenario, the count can be classified in the following classes of growth:

$$C(n) \in \Omega(1)$$

$$C(n) \in O(n^2)$$

1 (c)

The algorithm shown below is a decrease-by-one recursive method because it decreases the amount of work by one with each recursive call. The basic operation of this algorithm is the comparison of elements in the array.

```
public static boolean isSymmetric( int[][] A, int i, int j) {
      //Base case, checks if at end
      if(i == (A.length - 1) && j == (A.length - 1)) {
            return true; //Went all the way through without asymmetry
      }
      if (A[i][j] != A[j][i]) { //Compares elements
          return false;
      } else if ( i == j ) { //Resets row, increases column
          return isSymmetric(A, 0, j + 1);
      } else { //Iterate through row, keeps same column
          return isSymmetric(A, i + 1, j );
      }
}
```

The best case scenario for this algorithm is the same as the iterative solution. If the first element comparison is asymmetric the algorithm will complete with only one operation, so C_{best} = $\Theta(1)$.

Similarly to the sample algorithm, the worst case scenario for this algorithm is when the input array is symmetric. The algorithm can perform at most n^2 comparisons, so efficiency class for this algorithm is C_{worst}=$\Theta(n^2)$. Again, the general case can be classified in the following classes of growth:

$$C(n) \in \Omega(1)$$

$$C(n) \in O(n^2)$$

2 (a)

i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9	i = 10
151	**177**	**122**	**188**	**136**	**114**			**195**	**163**	**140**

2 (b)

i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9	i = 10	i = 11
		163	140	188			122				195

i = 12	i = 13	i = 14	i = 15	i = 16	i = 17	i = 18	i = 19	i = 20	i = 21	i = 22
	151			177					136	114

2 (c)

The efficiencies, load factors, and collisions of the two hash tables are calculated below:

Part A (original):

Size$_{max}$ = 11:

$a = \frac{9}{11} = 0.82$

Collisions: 9

Efficiency:

Successful search: $\frac{1}{2}[1 + 1/(1 - 0.82)] = 3.28$

Unsuccessful search: $\frac{1}{2}[1 + 1/(1 - 0.82)^2] = 15.93$

Part B (new/resized):

Size$_{max}$ = 23:

$a = \frac{9}{23} = 0.39$

Collisions: 1

Efficiency:

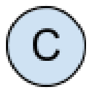
Successful search: $\frac{-1[ln(1-0.39)]}{0.39} = 1.27$

Unsuccessful search: $\frac{1}{(1-0.39)} = 1.64$

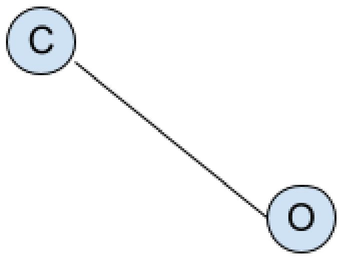
The resized hash table with quadratic probing is more time efficient than the original hash table. However, the new hash table is not very space efficient because, at 0.39, its load factor is far below the recommended max load factor of ~0.75. A new capacity of 17 would be better, unless the program expects many new items in the future.The original hash table needed to be resized because the load factor of 0.82 was over the recommended limit. This load is apparent in its efficiency calculation, where on average it takes over 3 probes to get a successful search and there are 9 collisions total. The decision to use quadratic probing on the new table was a good one because it is generally more efficient than linear probing.

3 (a)

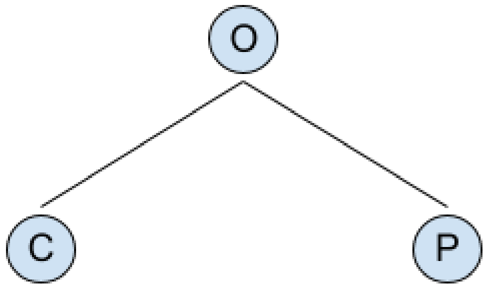
1:



2:

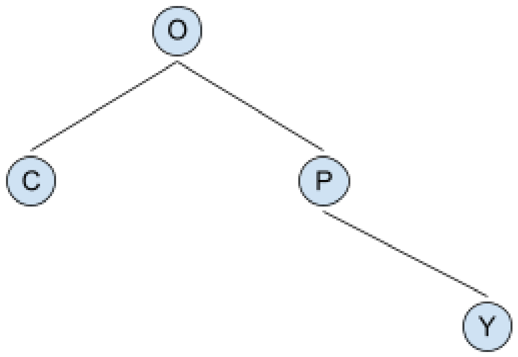


3:



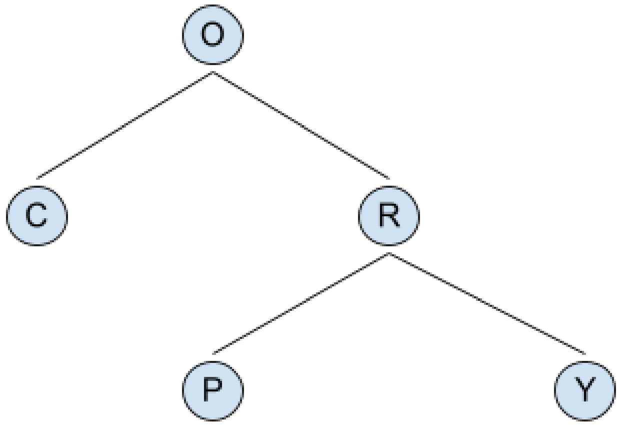
Left rotation of C around O

4:



Normal BST insertion

5:

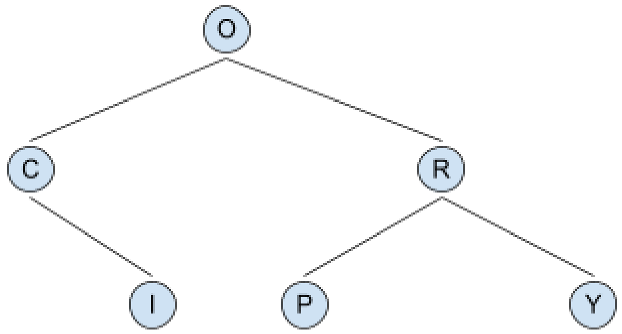


Right left rotate:

Rotate Y right around R

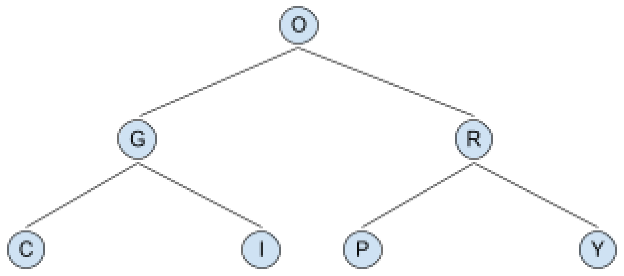
Rotate P left around R

6:



Normal BST insertion

7:

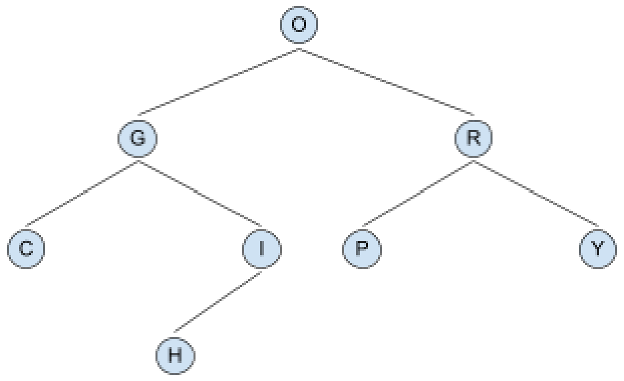


Right left rotate:

Rotate I right around G

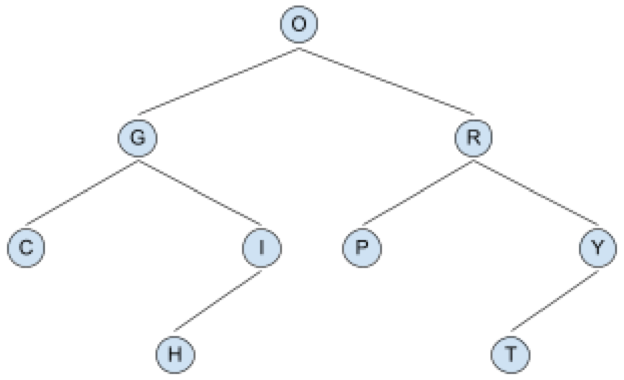
Rotate C left around G

8:



Normal BST insertion

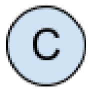
9:



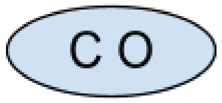
Normal BST insertion

3 (b)

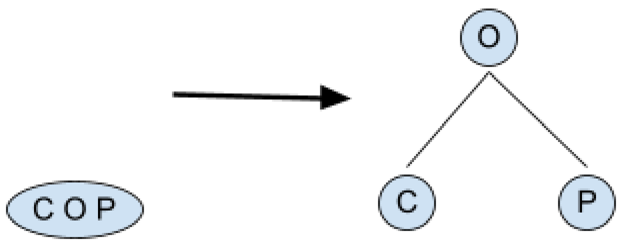
1:



2:

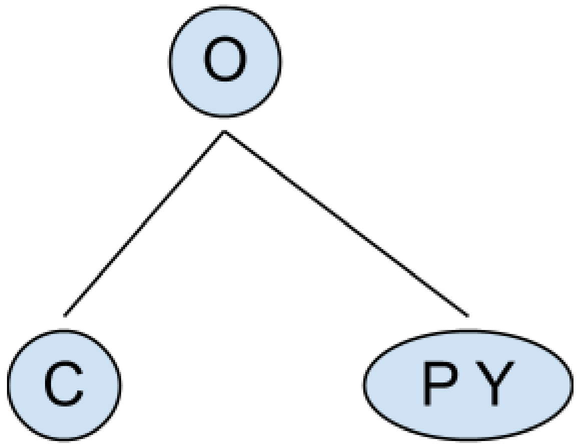


3:

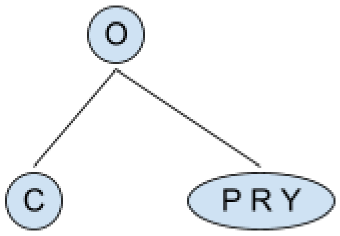


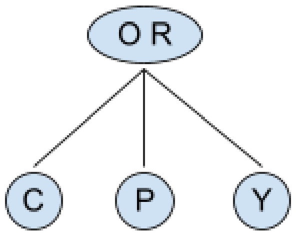
Temporary 4-node. O becomes parent, C and P become separate 2-nodes.

4:



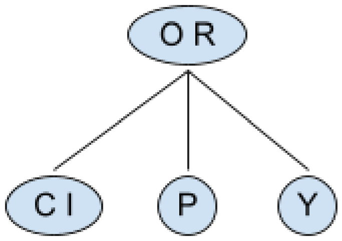
5:



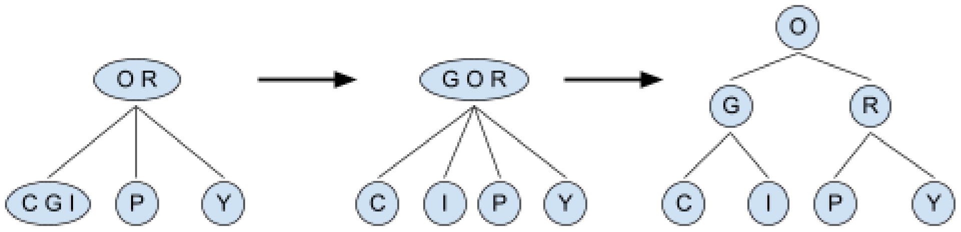


Temporary 4-node. R is moved to parent, P and Y become separate 2-nodes.

6:

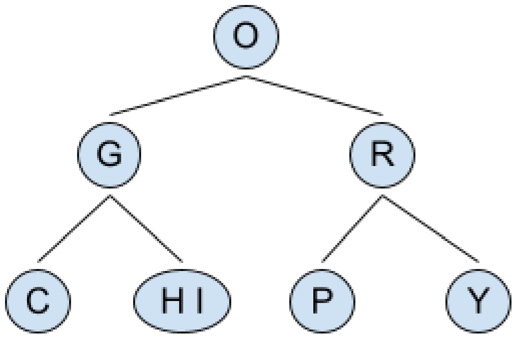


7:



Two temporary 4-nodes

8:



9: Final 2-3 tree