Nolan Anderson

CS317: Algorithms — Homework Assignment #5

Due: See Canvas for Assignment Due Dates — (20 points)

NOTE: NO LATE ASSIGNMENTS WILL BE ACCEPTED because we often review them in class on the due date.

Please upload the document containing your answers. They can be handwritten and scanned, but they must be clearly legible to receive a grade on the assignment. PDF is the best format for canvas. You DO NOT Need to include this cover sheet in your upload. It is formatted for the grader to use for me, as needed.

Chapter 5 Work the following problems. Point values are provided for each problem.

Problem #	Points	Grader's Notes
Sec 5.2, #1	*2*	
Sec 5.2, #3	*2*	
Sec 5.2, #7b	*1*	
Sec 5.3, #2	*2*	*Explain what is wrong with the algorithm and correct it*
Sec. 5.5, #1	*3*	

Chapter 6: Work the following problems. Point values are provided for each problem.

Problem #	Points	Grader's Notes
Sec 6.1, #2	*3*	
Sec 6.1, #4	*1*	
Sec 6.3, #1	*3*	
Sec 6.3, #4	*3*	

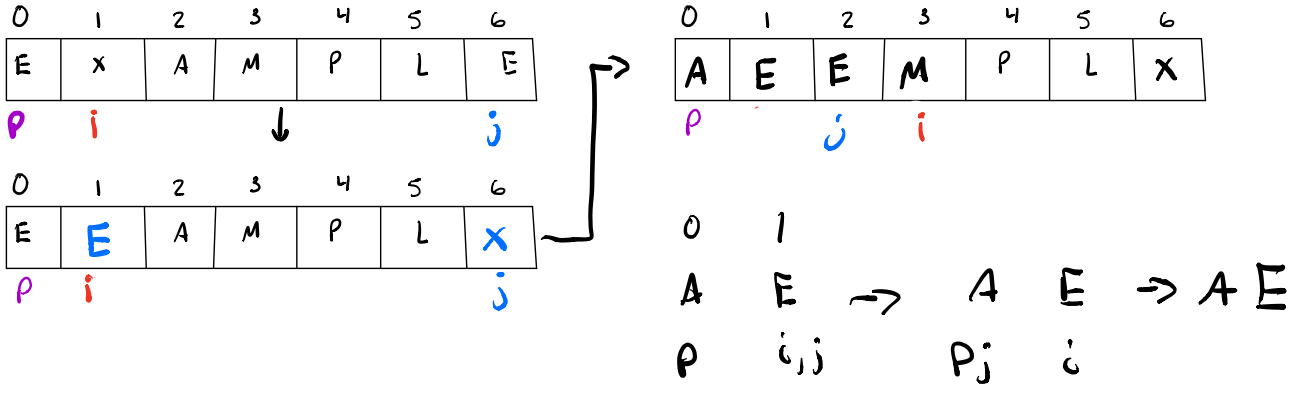
Other ungraded practice problems:

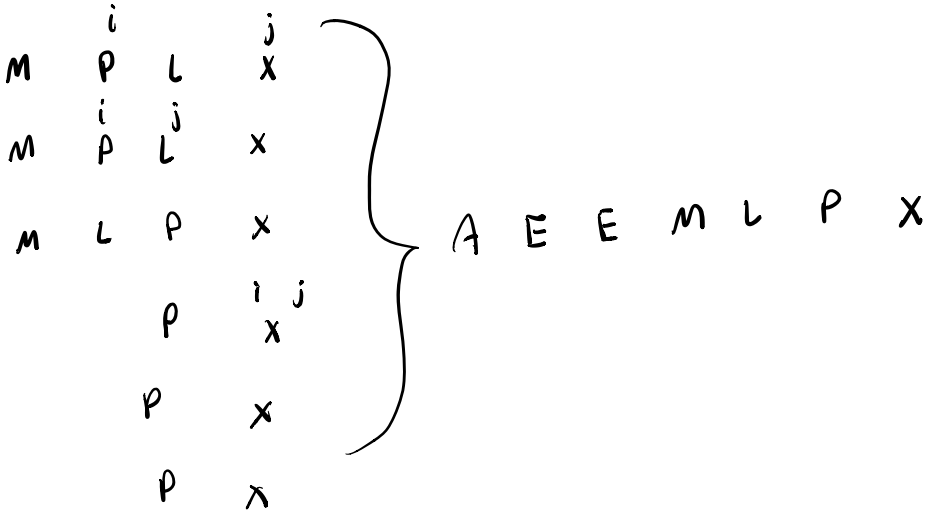
Sec 5.3: #5

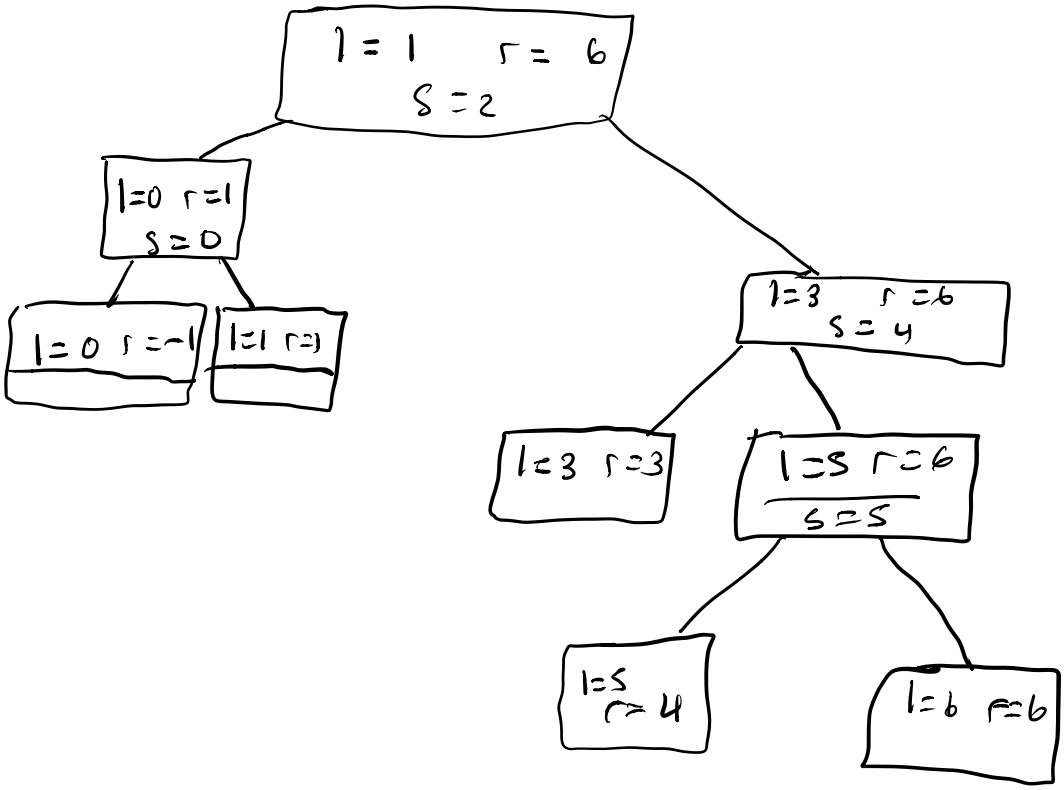
Sec 6.1: #3

5.2

1. Apply quicksort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made.




M P L X
M P L X
M L P X
P X
P X
P X
A E E M L P X


l = 1 r = 6
S = 2
l=0 r=1
S=0
l=0 r=-1
l=1 r=1
l=3 r=6
S = 4
l=3 r=3
l=5 r=6
S=5
l=5
r=4
l=6 r=6

3. Give an example showing that quicksort is not a stable sorting algorithm.

#1 is a prime example of this. The first E in the array ends up being placed after the second array.

A [4 9 1 6 7 5 4] ← here as well. The 4 in A[0] will actually end up after the 4 in A[6]

7b. True or false: For every $n > 1$, there are n-element arrays that are sorted faster by insertion sort than by quicksort?

Quicksort	Insertion Sort
$C_{best} = n \log n$	$C_{best} = O(n)$
$C_{worst} = n^2$	$C_{worst} = O(n^2)$

This is False because insertion sort is only faster for sorted array's, not every n-element array. In very few cases where insertion sort is faster.

5.3

2. The following algorithm seeks to compute the number of leaves in a binary tree.

ALGORITHM *LeafCounter(T)*

```
1 //Computes recursively the number of leaves in a binary tree
2 //Input: A binary tree T
3 //Output: The number of leaves in T
4 if T = ∅ return 0
5 else return LeafCounter(T_left) + LeafCounter(T_right)
```

Is this algorithm correct? If it is, prove it; if it is not, make an appropriate correction.

The issue here is that we're never checking for if the node is a leaf. In between lines 4 & 5, we should add

```
else if ( Tleft = null and Tright = null)
    return 1
```

5.5

1. a. For the one-dimensional version of the closest-pair problem, i.e., for the problem of finding two closest numbers among a given set of n real numbers, design an algorithm that is directly based on the divide-and-conquer technique and determine its efficiency class.

b. Is it a good algorithm for this problem?

Algorithm ClosestPair(p[l...r])
// Subarray p[l..r] of p[0..n-1]
// sort in increasing order
// Outputs distance between closest pairs.
if l=r
    return ∞ // because leftmost and rightmost is smortest distance.
else if r-l=1
    return p[r]-p[l] // difference is 1, subtract two values.
else return min { ClosestPair(p[l... L(l+r)/2]),
ClosestPair(p[(L l+r/2)+1...r]),
p[L(l+r)/2 +1]+1] - p[L l+r/2] }

↑ divide into two smaller parts etc....

Time Complexity:

$T(n) = 2T(n/2) + O(1)$

Master theorem: $a=2$ $b=2$ $d=0$

$a > b^d$

$T(n) = \Theta(n)$, added w/ mergesort:

$\Theta(n \log n)$

b) You could just as easily sort the array and then find the distance of the closest pairs which would be O(n log n) similar but a much simpler algorithm.

6.1

2. Let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ be two sets of numbers. Consider the problem of finding their intersection, i.e., the set C of all the numbers that are in both A and B.

a. Design a brute-force algorithm for solving this problem and determine its efficiency class.

b. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

```
BF Intersection (A, B)
    C ← A ∩ B          // list to contain elements from a and b.
    While(ai != null)  // loop through whole list
        if (ai = bj)   // if they are equal...
            C ← ai     // C gets the value
            remove bj  // remove bj, no longer needed
        ai = ai + 1    // go to next element
        bj = bj + 1    // go to next element
```

→ efficiency class: if there are no sets w/ common values, number of comparisons will be O(n x m)

b). PB Intersection (A, B)

```
C ← A + B          // add two lists into new list C.
MergeSort (C)      // sort the new Combined list.
while ( Ci != null) // while not at end of list
    if (Ci = Ci+1) // if Ci = next element
        D ← Ci     // D list gets the element
    Ci = Ci + 2    // increment to next value
return D           // return list of intersections.
```

O((n+m) log (n+m)) + O (n+m) ← mergesort efficience + rest.

4. Estimate how many searches will be needed to justify time spent on presorting an array of 10^3 elements if sorting is done by mergesort and searching is done by binary search. (You may assume that all searches are for elements known to be in the array.) What about an array of 10^6 elements?

10^3 elements by mergesort presorting.

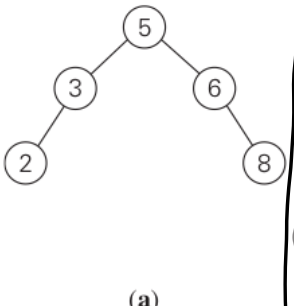
$n \log n + k \log_2 n \leq kn/2$

$$k \geq \frac{n \log n}{n/2 - \log n}$$

10^3: $k = 21$

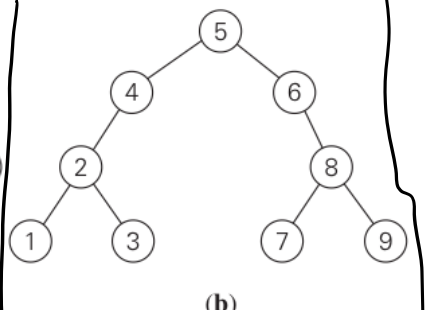
10^6: $k = 40$

1. Which of the following binary trees are AVL trees?



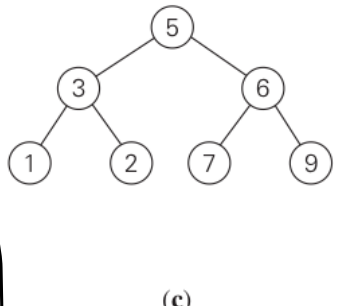
(a)

This is an AVL tree, it is balanced.



(b)

Two additional nodes 4 & 6. makes this unbalanced and not an AVL tree.



(c)

not a subtree, 7 is left of 6.

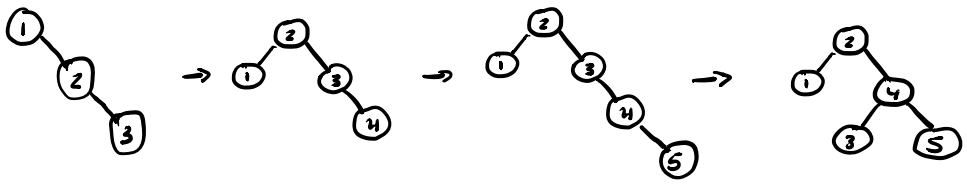
4. For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree.

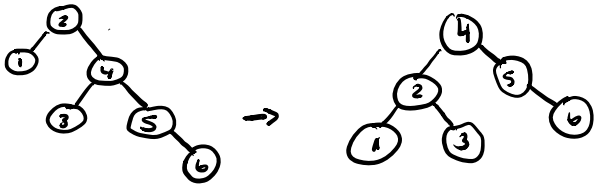
a. 1, 2, 3, 4, 5, 6

b. 6, 5, 4, 3, 2, 1

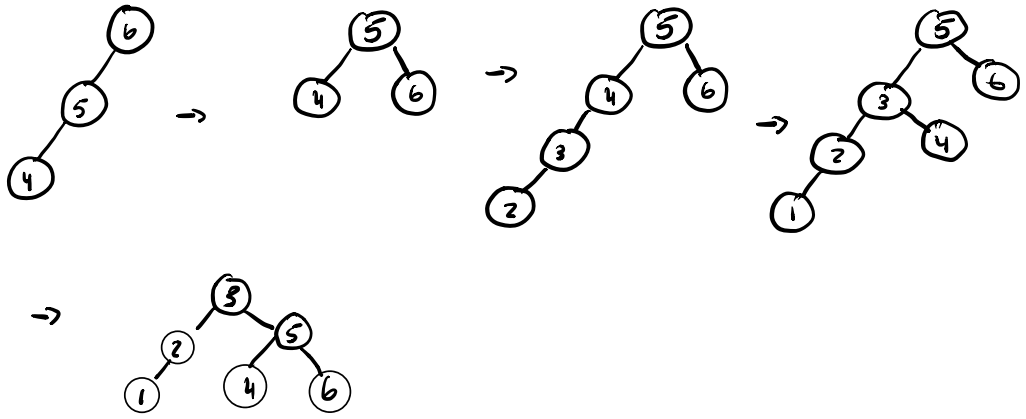
c. 3, 6, 5, 1, 2, 4

a)





b. 6, 5, 4, 3, 2, 1



c. 3, 6, 5, 1, 2, 4