

Name: \_\_\_\_\_ SOLUTION KEY \_\_\_\_\_

VT ID # \_\_\_\_\_

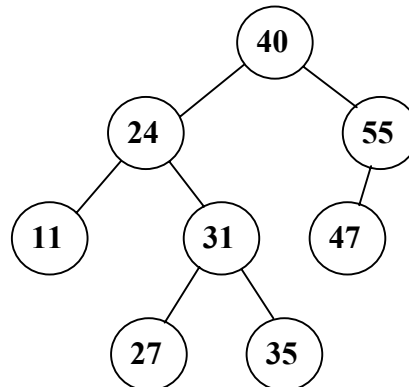
This homework assignment will be **handwritten and turned in in-class, at the start of class (2:00pm) on Thursday, October 28, 2004**. Write your answers on this document in the space provided for each question. Be sure to write your name on each page and keep the pages stapled together. No late assignments will be accepted.

For Problem 1, consider the following C++ function:

```
void mysterySort(T data[], int n)
{
    for (i = 0; i < n-1; i++)
    {
        for (j = n-1; j > i; --j)
        {
            if (data[j] < data[j-1])
            {
                T tmp = data[j];
                data[j] = data[j-1];
                data[j-1] = tmp;
            }
        }
    }
}
```

For Problems 2 – 6, consider Tree T shown below:

Tree T



1. [15 points] Fill-in the table below to indicate in Big- $\Theta$  notation the number of comparisons and the number of moves that `mysterySort` will make in the best, worst, and average case. **In order to receive full credit, in the space after the table, you must fully justify your answer for each cell in the table.**

	Best-case	Worst-Case	Average-Case
Number of Comparisons	$n^2$	$n^2$	$n^2$
Number of Moves	0	$n^2$	$n^2$

### Number of Comparisons

For the number of comparisons, consider how many times the statement below is performed:

```
if (data[j] < data[j-1])
```

In the `mysterySort` function, the best, worst, and average case are all the same in terms of the number of comparisons because neither the  $i$  or  $j$  loops depend on the ordering of the elements in the array to be sorted. Thus, the number of iterations through the  $i$  and  $j$  loops only depend on the number of elements in the array. The number of comparisons can be expressed as:

$$\text{number of comparisons} = \sum_{i=1}^{n-1} \sum_{j=i}^{n-2} 1 = \sum_{i=1}^{n-1} (n-2-i-1+1) = \sum_{i=1}^{n-1} (n-i-1)$$

Further simplification of the expression above shows that there is an  $n^2$  term that dominates the expression, and thus, the number of comparisons is  $\Theta(n^2)$  for the best, worst, and average case.

### Number of Moves

The only time data is moved in the `mysterySort` function is when the if statement shown below is true.

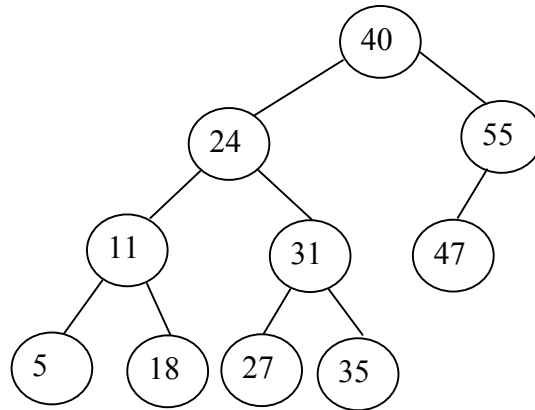
```
if (data[j] < data[j-1])
```

Thus, the best case occurs when the data is already sorted. In the best case, the statement above will never be true, and thus, zero moves will be made.

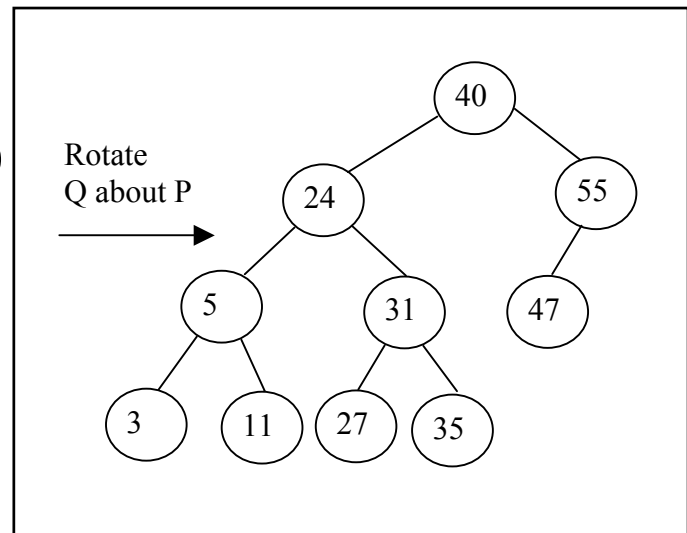
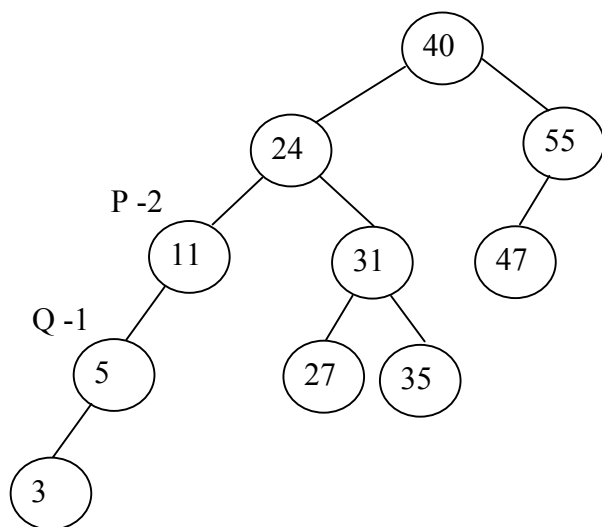
In the worst case, the if statement is true every time and thus the number of moves will be the same as the number of comparisons times three, which is still  $\Theta(n^2)$ .

For the average case, assume that the if statement is true only half the time. In this case, the number of moves will be  $3/2$  the number of comparisons, which is  $\Theta(n^2)$ .

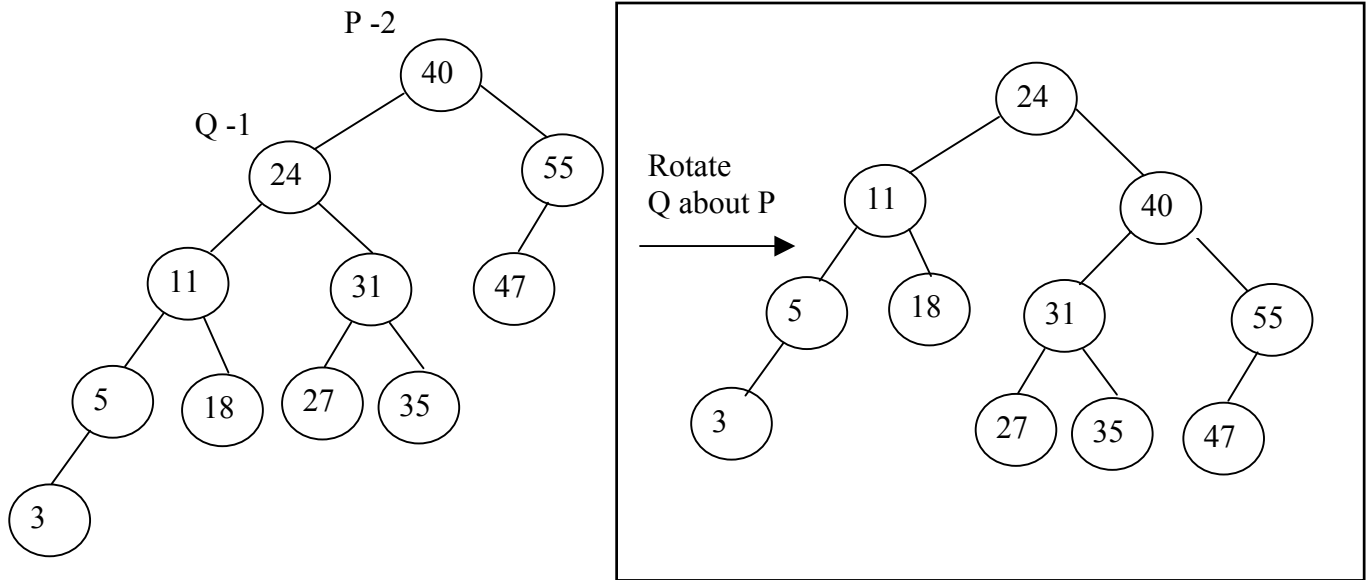
2. [5 points] Starting with Tree T as shown on the first page, draw the AVL tree that would result from inserting 5 and 18 (in that order) into Tree T.



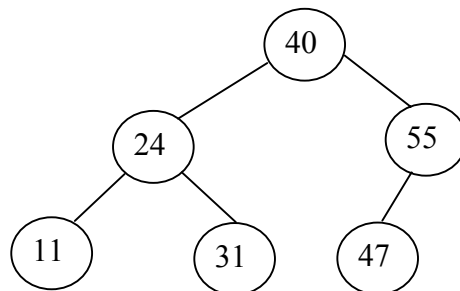
3. [5 points] Starting with Tree T as shown on the first page, draw the AVL tree that would result from inserting 5 and 3 (in that order) into Tree T.



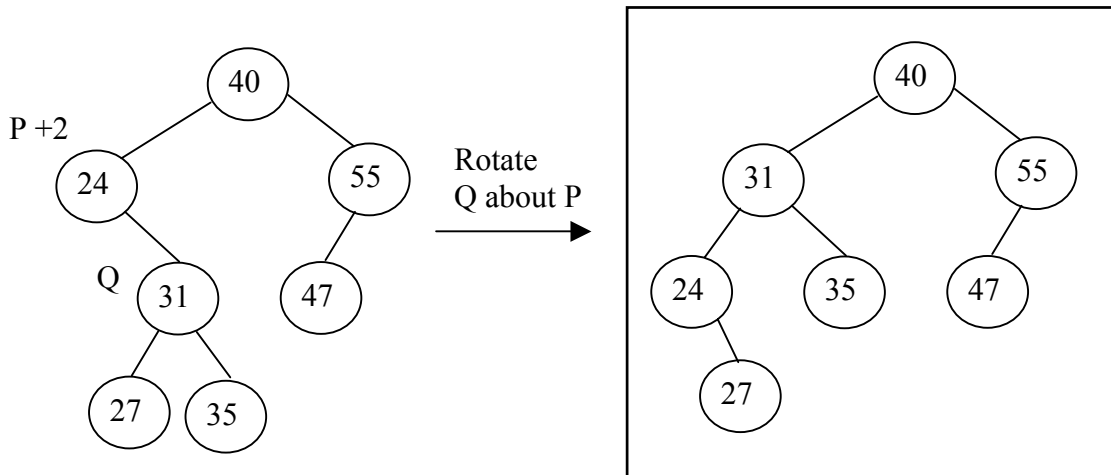
4. [5 points] Starting with Tree T as shown on the first page, draw the AVL tree that would result from inserting 5, 18, and 3 (in that order) into Tree T.



5. [5 points] Starting with Tree T as shown on the first page, draw the AVL tree that would result from deleting 27 and 35 (in that order) from Tree T.



6. [5 points] Starting with Tree T as shown on the first page, draw the AVL tree that would result from deleting 11 from Tree T.



7. [10 points] Consider calling the BuildHeap function shown on slide 4 of the class notes on Heaps with the following array as input: [ 36, 24, 21, 2, 18, 3, 7, 12, 10, 41, 25 ]. Show the array at the end of each iteration of the for loop in the BuildHeap function. (Hint: It may help to draw the heap on scratch paper).

	Array[]										
	0	1	2	3	4	5	6	7	8	9	10
start	36	24	21	2	18	3	7	12	10	41	25
Idx = 4	36	24	21	2	41	3	7	12	10	18	25
Idx = 3	36	24	21	12	41	3	7	2	10	18	25
Idx = 2	36	24	21	12	41	3	7	2	10	18	25
Idx = 1	36	41	21	12	25	3	7	2	10	18	24
Idx = 0	41	36	21	12	25	3	7	2	10	18	24

8. [10 points] Using the implementation of quicksort shown on slides 40 and 41 of the class notes on Sorting, illustrate the operation of quicksort on the array [ 36, 24, 21, 2, 18, 3, 7, 12, 10, 41, 25 ]. Note that this implementation of quicksort chooses the middle-most element as the pivot value. To illustrate the operation of quicksort on the array, show the array at the end of each call to Partition. **You may not need to use all the rows in the table provided below, show as many rows in the table below as are needed.**

Here is the table as required for the assignment:

Pivot	LastPreceder	Array[]										
		0	1	2	3	4	5	6	7	8	9	10
start	start	36	24	21	2	18	3	7	12	10	41	25
3	1	2	3	21	24	18	36	7	12	10	41	25
7	2	2	3	7	24	18	36	21	12	10	41	25
21	6	2	3	7	10	18	12	21	36	24	41	25
18	5	2	3	7	12	10	18	21	36	24	41	25
12	4	2	3	7	10	12	18	21	36	24	41	25
24	7	2	3	7	10	12	18	21	24	36	41	25
41	10	2	3	7	10	12	18	21	24	25	36	41
25	8	2	3	7	10	12	18	21	24	25	36	41

These next two parts were not required in the assignment, but may be of interest in understanding how quicksort works.

Below is a more detailed trace:

	Pivot	LastPreceder	Idx	Array[]											
				0	1	2	3	4	5	6	7	8	9	10	
	start	start	start	36	24	21	2	18	3	7	12	10	41	25	
<b>Partition(0,10)</b>				36	24	21	2	18	3	7	12	10	41	25	
swap 0 & 5	3	0		3	24	21	2	18	36	7	12	10	41	25	
		1	3	3	2	21	24	18	36	7	12	10	41	25	
swap 0 & 1				2	3	21	24	18	36	7	12	10	41	25	
<b>Partition(0,0)</b>				2											
<b>Partition(2,10)</b>						21	24	18	36	7	12	10	41	25	
swap 2 & 6	7	2				7	24	18	36	21	12	10	41	25	
	7	2	10			7	24	18	36	21	12	10	41	25	
swap 2 & 2						7	24	18	36	21	12	10	41	25	
<b>Partition(2,1)</b>						7									
<b>Partition(3,10)</b>								24	18	36	21	12	10	41	25
swap 3 & 6	21	3				21	18	36	24	12	10	41	25		
	21	4	4			21	18	36	24	12	10	41	25		
	21	5	7			21	18	12	24	36	10	41	25		
	21	6	8			21	18	12	10	36	24	41	25		
swap 3 & 6						10	18	12	21	36	24	41	25		

				Array[]										
	Pivot	LastPreceder	Idx	0	1	2	3	4	5	6	7	8	9	10
<b>Partition(3,5)</b>							10	18	12					
swap 3 & 4	18	3					18	10	12					
	18	4	4				18	10	12					
	18	5	5				18	10	12					
swap 3 & 5							12	10	18					
<b>Partition(3,4)</b>							12	10						
swap 3 & 3	12	3					12	10						
	12	4	4				12	10						
swap 3 & 4							10	12						
<b>Partition(3,3)</b>							10							
<b>Partition(5,4)</b>								12						
<b>Partition(6,5)</b>									18					
<b>Partition(7,10)</b>											36	24	41	25
swap 7 & 8	24	7									24	36	41	25
	24	7	10								24	36	41	25
swap 7 & 7											24	36	41	25
<b>Partition(7,6)</b>											24			
<b>Partition(8,10)</b>												36	41	25
swap 8 & 9	41	8										41	36	25
	41	9	9									41	36	25
	41	10	10									41	36	25
swap 8 & 10												25	36	41
<b>Partition(8,9)</b>												25	36	
swap 8 & 8	25	8										25	36	
	25	8	9									25	36	
swap 8 & 8												25	36	
<b>Partition(8,7)</b>												25		
<b>Partition(9,9)</b>													36	
<b>Partition(11,10)</b>														41
FINAL SORTED				2	3	7	10	12	18	21	24	25	36	41

Below is a tree showing the calls to Partition – shown in the tree as P(Lo,Hi):

