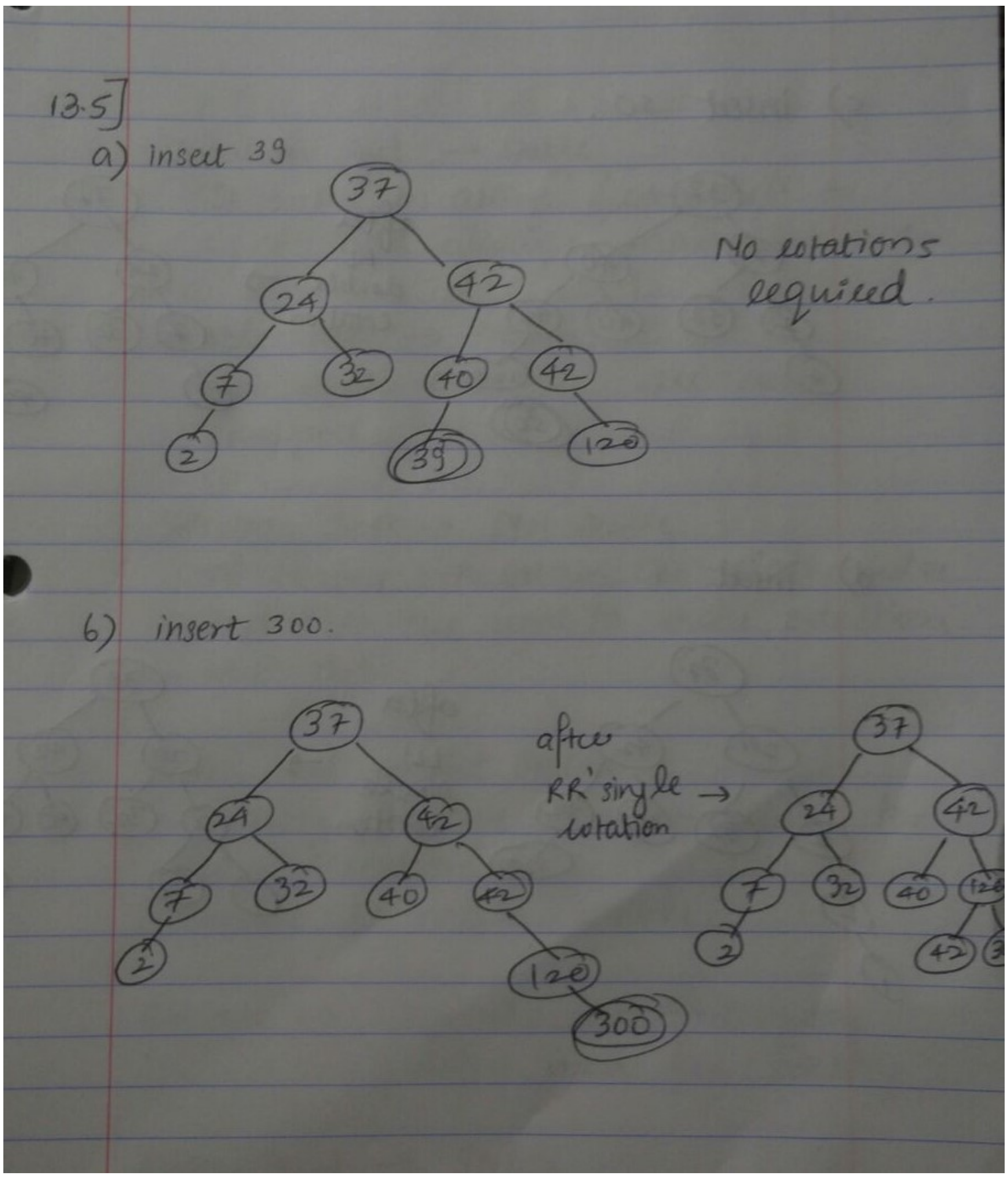
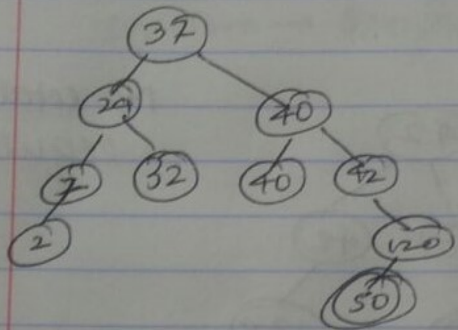
Homework 3
Mohit Galvankar mgalvank

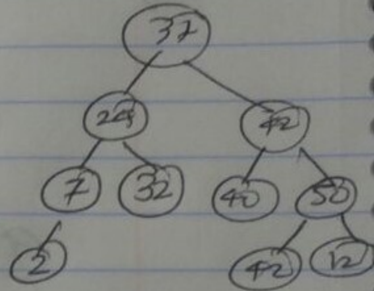
13.5 (a) Show the result (including appropriate rotations) of inserting the value 39 into the AVL tree on the left in Figure 13.4.
(b) Show the result (including appropriate rotations) of inserting the value 300 into the AVL tree on the left in Figure 13.4.
(c) Show the result (including appropriate rotations) of inserting the value 50 into the AVL tree on the left in Figure 13.4.
(d) Show the result (including appropriate rotations) of inserting the value 1 into the AVL tree on the left in Figure 13.4.



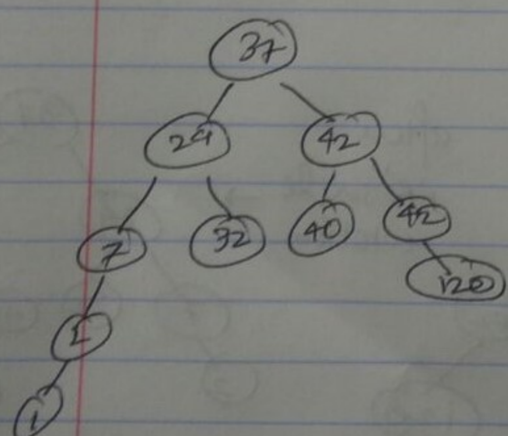
c) insert 50.



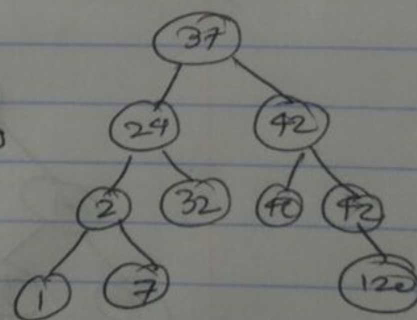
after RL' double rotation →



d) insert 1



after LL' single rotation →



7.1 Using induction, prove that Insertion Sort will always produce a sorted array.

7.1]

Insertion sort
if $n=0$, then return [ ]
else
insertionSort ($A_1, A_2, \cdots A_{n-1}$)
insert (A_n into $A_1, A_2, \cdots A_{n-1}$)

Base case
$P(0) \leftarrow$ input is empty array.
$P(0)$ is trivially true

We need to prove that $P(n-1) \rightarrow P(n)$.

Two operation :- Insertion sort
Insert.

InsertionSort is correct for $n-1$ ($P(n-1)$)
It suffices to prove that the insert subroutine is correct.

$Q(n)$: insert is correct for an array of n elements.
$Q(0)$: trivially true, a singleton is always sorted

Two subcase

1] $e > A_n$: e belongs at the end.

2] $e \leq A_n$: e is to the left of A_n.

In second case, we insert into the subarray $A_1 A_2 \dots A_{n-1}$.

This we can do by the inductive hypothesis.

The insert algorithm is correct by induction

$P(n) \rightarrow$ Insertion sort is correct for n inputs.

$P(0) \rightarrow$ trivially true (empty array)

$P(n-1) \rightarrow P(n)$ because of insert operations is correct.

Hence $P(n)$ for any $n \geq 0$ by mathematical induction.

7.6 Recall that a sorting algorithm is said to be stable if the original ordering for duplicate keys is preserved. Of the sorting algorithms Insertion Sort, Bubble Sort, Selection Sort, Shellsort, Mergesort, Quicksort, Heapsort, Binsort, and Radix Sort, which of these are stable, and which are not? For each one, describe either why it is or is not stable. If a minor change to the implementation would make it stable, describe the change.

7.6]

Insertion Sort → Stable
It sorts an array by moving from left to right and always maintains order.

Bubble Sort → Stable
Two equal elements are never swapped.

Selection Sort → Not stable.
It doesnt maintain the relative order.
Linked lists are used to make selection sort stable

Heap Sort → Not Stable.
Ordering of items is lost during the heap creation.
Cannot be made stable

Bin sort → Depends on underlying algo used for sorting.

Radix Sort → Stable.
Implements counting sort which is stable.

Shell sort → Not Stable.
Operations can change the order of elements with equal values.

Merge sort → Stable for most cases.
Stable if '<' condition used.
else if "<=" condition used → Not Stable

Quick sort → Not Stable.
Doesn't hold the order of elements.
Tracking of original array order can make it Stable

7.11 Modify Quicksort to find the smallest k values in an array of records. Your output should be the array modified so that the k smallest values are sorted in the first k positions of the array. Your algorithm should do the minimum amount of work necessary, that is, no more of the array than necessary should be sorted.

7.16 (a) Devise an algorithm to sort three numbers. It should make as few comparisons
as possible. How many comparisons and swaps are required
in the best, worst, and average cases?
(b) Devise an algorithm to sort five numbers. It should make as few comparisons
as possible. How many comparisons and swaps are required
in the best, worst, and average cases?
(c) Devise an algorithm to sort eight numbers. It should make as few comparisons
as possible. How many comparisons and swaps are required
in the best, worst, and average cases?

7.16

a) Let A, B, C be the elements.

Array ← [A, B, C] unsorted.
if (A < B && B < C)
    return array sorted.
else
    implement bubble sort.

Best case.

If the array is sorted. we can get the result ~~in constant time in O(n)~~ constant time. ~~time~~

Worst case.

If the array is not sorted and in reverse order we can get the result in $O(n^2)$ i.e., the complexity of bubble sort.

Average case.

If only one piece is out of order still we will have to loop through all the state. $O(n^2)$

b) Array ← [A, B, C, D, E] unsorted.

if (A>B && C>D)
    compare (A, C)

if (A>C)
    sort E into (A, C, D)
    sort B into (E, C, D)

else implement bubble sort.

Best case :-
We get a sorted array in 7 comparisons ie, constant time.

Worst case :-
$O(n^2)$
Same as previous Q 7.16 (a).

Average case
$O(n^2)$
Same as previous Q 7.16 (a)

c) When there are 8 elements,

construct a leonardo max-heap from array.

x ← last element index of array.

while(heap != empty)

remove max element from heap

place element at x.

move x to previous position.

rebalance max heap.

Best case :- O(n)

Worst - Average case :- O(nlogn).

7.5 Starting with the Java code for Quicksort given in this chapter, write a series of Quicksort implementations to test the following optimizations on a wide range of input data sizes. Try these optimizations in various combinations to try and develop the fastest possible Quicksort implementation that you can.

(a) Look at more values when selecting a pivot.

(b) Do not make a recursive call to **qsort** when the list size falls below a given threshold, and use Insertion Sort to complete the sorting process. Test various values for the threshold size.

(c) Eliminate recursion by using a stack and inline functions.

16.1 Solve Towers of Hanoi using a dynamic programming algorithm.