1. (a) Given a number in the range 1 to $100^{100}$, how many base 2 digits are needed to represent that number? How many base 100 digits?

(b) Exactly how many rounds of bucket/pigeonhole sort will radix sort perform to sort a list of numbers in the range 1 to $100^{100}$ using base 2? Using base 100?

(c) If the numbers in a list of length $n$ are in the range 1 to $n^{100}$, how many base $n$ digits are needed to represent the number?

(d) Exactly how many rounds of bucket/pigeonhole sort will radix sort perform to sort a list of numbers in the range 1 to $n^{100}$ using base $n$?

(e) Given your answer to part (d), what is the runtime of radix sort on a list of numbers in a polynomial range (1 to $n^k$ for some constant $k$)?

(a) Writing a number $x$ in base $b$ requires $\lceil \log_b x \rceil + 1$ digits. So $100^{100}$ requires 665 digits in base 2 and 101 digits in base 100.

(b) Radix sort performs one round of bucket/pigeonhole sort per digit in the input. So it will take 665 and 101 rounds respectively.

(c) Using the same formula from part (a), it will take 101 digits.

(d) Similarly 101 rounds of bucket sort will be performed.

(e) Choosing to use base $n$, radix sort will perform $k + 1$ rounds of bucket sort and each round of bucket sort takes $O(n)$ time in that case. The total runtime for radix sort is $(k + 1)O(n) = O(n)$ because $k$ is assumed to be a constant.

2. Draw an AVL tree of height 5 with the maximum possible number of nodes. Draw an AVL tree of height 5 with the minimum possible number of nodes.

Maximum number of nodes is a perfectly balanced tree:
Minimum number of nodes can be constructed recursively:

3. Draw the binary search tree obtained by inserting the numbers 1 through 7 in sorted order. Draw the AVL tree obtained by doing the same sequence of insertions.

   Binary search tree insertion:

   The AVL tree insertion actually gives a perfectly balanced tree when you insert in order:

4. Consider inserting the values (5, 2), (3, 6), (0, 2), (4, 1), (2, 0), (1, 2), (1, 0), and (3, 1) into two hash tables of size 5. For one hash table use the hash function \( h(x, y) = 2x + y \). For the other use the hash function \( h(x, y) = x^2 + xy + y^2 \). Which hash function performs better? (Assume no resizing takes place)
The second hash function spread the elements out more evenly. So it performed better in this case.

Extra credit: On the last homework you were asked to build a rooted binary tree that will have a given traversal. Now given a list, build a balanced rooted binary tree such that the pre-order traversal of the tree is the given list. Do the same for the in-order and post-order traversals. Full credit will be awarded for a linear time solution.

A solution for the in-order version of the problem:

build_balanced(L):
    if L.size() == 1:
        return new Node(L[0])
    pivot = L.size()/2
    Node n = new Node(L[pivot])
    n->left = build_balanced(L[0:pivot])
    n->right = build_balanced(L[pivot+1:L.size()])
    return n

For the other traversals change the indexes appropriately i.e. for pre-order set pivot = L[0] and recurse on L[1, L.size()/2] and L[L.size()/2 : L.size()] and for post-order set pivot = L[L.size() - 1] and recurse on L[0 : L.size()/2] and L[L.size()/2, L.size() - 1].

A note on the extra credit: A solution to the extra credit is a useful tool. When the input is a sorted list and you build a tree with that list as its in-order traversal, the tree you constructed is a binary search tree. So if you have the sorted order of a list of elements, you can build a balanced binary search tree in $O(n)$ time. Attempting to build the binary search tree by just inserting the elements in some order (not necessarily the sorted order), may not give a balanced tree and can be shown to take $\Omega(n \log n)$ time no matter what order of insertion is chosen.