Problem Set 3

All parts are due March 17th, 2016 at 11:59PM.

Name: Arturo Chavez-Gehrig

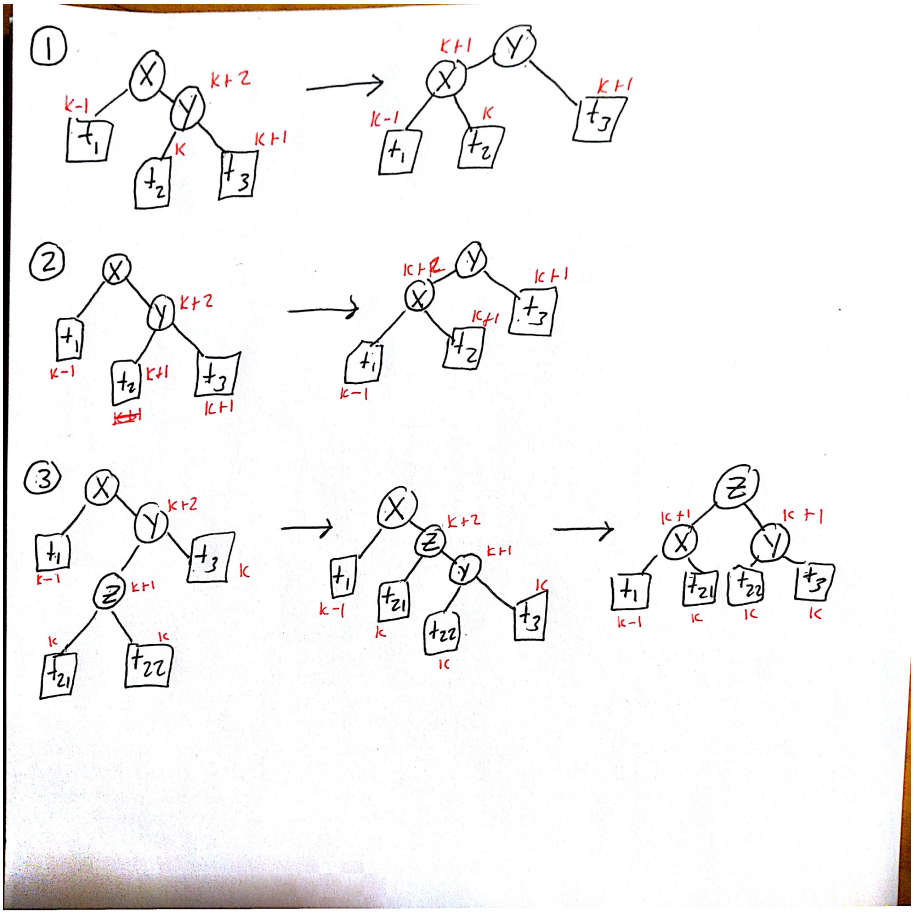
Collaborators:

Part A

Problem 3-1.

(a) In order to insert a new element into an AVL-2 tree, we will first add the new element as a leaf to the tree. Then, we will check that each node leading to the new leaf satisfies the AVL-2 property that the heights of each node's subtrees differ by at most 2. We will re-balance the tree using the same left shift and right shift operations as were used in AVL tree, moving from leaf up to the root, doing this at most h times. We are taking advantage of the property that a descendant subtree will remain balanced as we re-balance upwards towards the root of the tree.

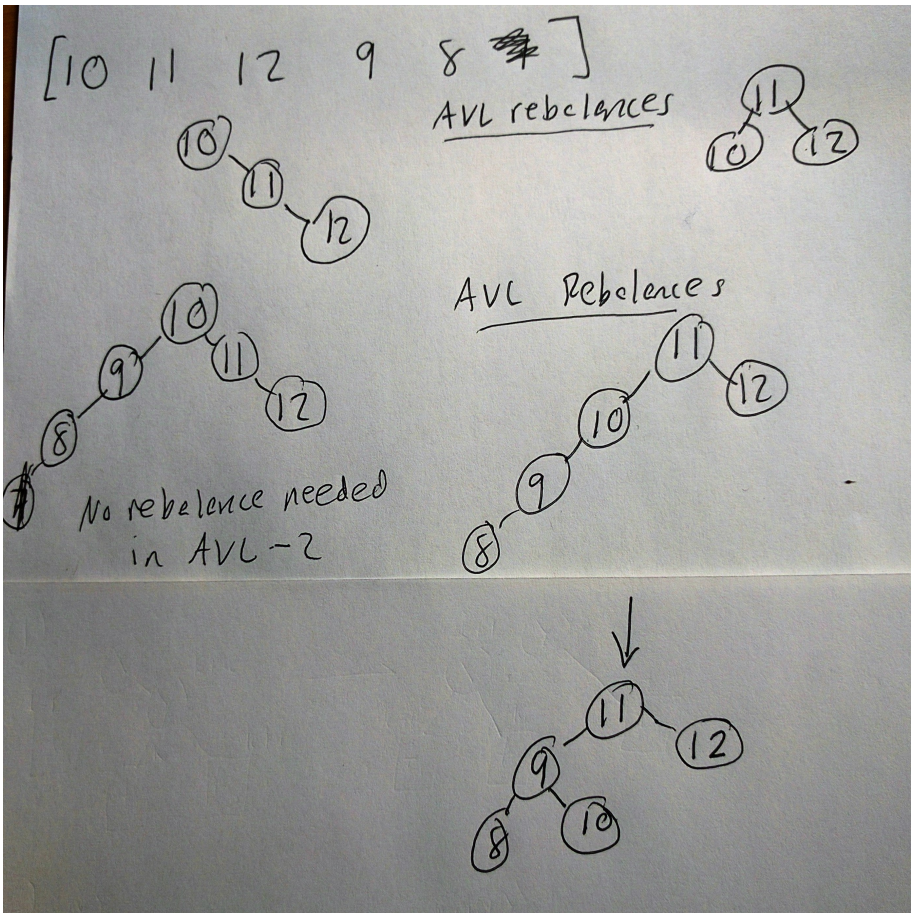
(b)



(1)

When a subtree needs to be re-balanced, it will fall into one of three cases shown in the picture above. In Case 1, x's right child is right heavy, leading to a right subtree that differs by 3 in height to the left. A left rotate allows the tree to be rebalanced as shown above, causing the tree height to decrease by 1. In Case 2, x's right child is balanced, but that right child has a height that is greater than 2 of the height of the left child. Again, left rotate resolves the imbalance, keeping the overall height of the subtree unchanged. In Case 3, y is left heavy. Now, a right rotation followed by a left rotation solves the imbalance, leading to a tree that has a reduced height by 1. All these cases can occur on the left side instead of the right, but a mirror image of the three cases listed above solve those imbalances. Therefore, all re-balances of an AVL-2 insert either reduce the height of the subtree by 1 or leave it unchanged.

(c)



(2)

In this sequence, we see that the AVL-2 never needs to be re-balanced, but the AVL does twice. This shows that since an AVL-2 is more lenient, it can allow the input to 'correct itself' before doing any action. The standard AVL does not allow such flexibility.

(d) In the least compact case the group of N elements forming a height h AVL-2 will satisfy the property.

$N_h = N_{h-1} + N_{h-3} + 1$

$N_{h-1} = N_{h-2} + N_{h-4} + 1$

$N_h = N_{h-2} + N_{h-4} + 1 + N_{h-3} + 1$

Now we know $N_h > 2N_{h-3} > 2^{h/3}$.

By transitivity $N_h > 2^{h/3}$.

Taking the log of both sides, we have $3logn > h$.

Therefore $h = O(logn)$.

(e) $n_h \geq 1 + n_{h-1} + n_{h-(c+1)}$ this is assuming the least compact case

$n_h \geq 2 * n_{h-(c+1)}$

$n_h \geq n^{\frac{h}{c+1}}$

$h = (c+1)logn = clogn = O(logn)$

Problem 3-2.

(a) The constraints of this array imply that A_i will be 'consecutive' to A_{i+1} because there is no overlap between the two subsequences. This is because every element in the subset of A_i is either greater than or less than every element in another subsequence A_j. Also, there are no duplicate elements, so there will be no intermixing of subsequences and we can sort the elements such that $A_i > A_j > A_k$ for some 3 subsets

(b) Usually, a subsequence of n can be arranged in $n!$ permutations, but because of the constraints outlined in part A, there are fewer options. We have m! ways to arrange the subsequences $(A_i, A_j, ...)$. There are k! ways to arrange a single subsequence A_i of k elements. We arrange all m subsequences of length k, so there $(k!)^m$ ways of arranging within all of the subsequences. So overall, we have $m! * (k!)^m$ possible permutations of the indices in A which can occur in the sorted outputs.

(c) Number of leaves is equal to 2^h and there are $m!*(k!)^m$ possible correct sorted outputs, therefore there must be at least that many leaves. We can use Stirling's approximation to simplify $m! * (k!)^m = [\sqrt{2 * \pi * m} * (\frac{m}{e})^m] * [\sqrt{2 * \pi * k} * (\frac{k}{e})^{km}]$

Taking the log on both sides, we get $h = m*log(\sqrt{2 * \pi * k}*\frac{m}{e})+km*log(\sqrt{2 * \pi * k}*\frac{k}{e})$

Therefore, in the 'worst case', $h = \Omega(mlogm + nlogk)$

(d) In order to sort the input sequence A most efficiently given the constraints above, we perform merge sort on each subsequence. This will run in $m * klogk = nlogk$. Then we do merge sort choosing the first element of each subset. This will run in $mlogm$ run time. Overall, we have a sorting algorithm with an asymptotic bound $O(nlogk + mlogm)$. This matches the lower bound we found in part c.

Problem 3-3.

(a) We can use radix sort on the last names going from the last letter to the first letter, sorting with counting sort on each letter. Counting sort works well here because there are only 26 letters that can possibly be at each position and counting sort is stable (a requirement to be used in radix sort).

Counting sort runs in $\Theta(n+k)$, but in this case k is $O(n)$ since there exactly 26 'buckets' for the possible values so counting sort actually runs in $\Theta(n)$. Radix sort performs counting sort for each letter therefore maximum number of times is the maximum length of the last name (k_l). So radix sort runs in $O((n+2^r)k_1)$, but we set $2^r = \frac{b}{r}$ so $O(k_l * n)$.

(b) Same as the algorithm specified in part a, but now we start on the last letter of the first name. We sort each letter going from the last letter to the first letter of the last name. Counting sort is stable and very efficient for sorting by letter. Our algorithm will take $2 * k * n$ where k is the longest of any first or last name. So this sorting algorithm takes $O(kn)$.

(c) See analysis parts a and b showing run times of $O(kn)$.

Problem 3-4.

(a) Upper bound on expected time for search to complete can be represented by the probability of choosing a specific slot in the hash table times the length of the chain, in this unfortunate case, length n since all elements are hashed to the same location in the hash table, forming a linked list of length n. $O(E(\text{completed search}) = \frac{1}{m} * n + \frac{m-1}{m} = \frac{n}{m})$

(b) Parallel reasoning to part a, $O(E(\text{completed search}) = \frac{1}{m} * \frac{n}{2} + \frac{1}{m} * \frac{n}{2} + \frac{m-2}{m} = \frac{n}{m})$

(c) Let x be the length of the chain in each slot of the hash table and y be the number of empty locations in the hash table. So $O(E(\text{completed search}) = \frac{1}{m} * \sum x + \frac{y}{m} = \frac{n}{m})$

(d) Supposing that search is guaranteed to be successful, it is equally likely to be in any part of the linked list. Therefore, in expectation the element we are looking for is found in the middle of the linked list. Using this information, we conclude that $O(E(\text{completed search}) = \frac{n}{2m} + \frac{n-1}{m} = \frac{n}{m})$

(e) We define a random variable CH_i representing the number of inserted keys into the same location H_i as 1 if the key hashes to that slot and 0 otherwise. So $E(CH_i) = \sum_0^n E(H_i) = \sum_0^n P(H_i = 1) + 0 * P(H_i = 0)$.

We know $P(H_i = 1) = \frac{1}{m}$ due to uniform hashing, so $E(CH_i) = \frac{n}{m}$.

Then the expected time to complete search is $O(\frac{n}{m}) + O(1)$.

Problem 3-5.

(a) We are going to use a version of the Rabin-Karp algorithm for string searching. So we will use a hashing function taking the names of the authors corresponding to a number. We will use a dictionary and enter the hash number as the key and the author's name

as the value. We will perform a rolling hash through the elements of each scroll. If the window of k elements matches the entry in our dictionary corresponding to an author's name, we will save that substring and later loop through these candidates, matching them to the exact name of the author. Alternatively, if the k elements do not match any of the hashed author's name encodings, we will drop the leftmost character and add the rightmost character and repeat.

Using the Rabin-Karp recitation notes as a guide.

FindAuthor(list of authors):

AuthorDict={}

for author in list of authors:

AuthorDict[encode(author)=author]

return AuthorDict

Matching(AuthorDict,P(scroll))

$h_p = hash(P)$

if hash(P) matches the encoding for the author

if Verify(0,m)

print author

return

rolling hash remove the first letter

$h_p = h_p - e^{scroll(i-1)}$

multiply by 26 and add new letter to the end

$h_p = 26 * h_p + scroll[i - 1 + m]$

if $h_a == h_p$ and Verify

print author

return

else no match

return none

(b) For the pre-processing of the list of signatures by the authors, we meet of $O(mk)$ target bound because we encode m author's names of length k and hashing takes $O(1)$.

(c) Rolling hash takes run time $O(k + n - k - 1)$ which equals $O(n)$ as long as k is less than n. We are relying on the fact that dictionary search takes $O(1)$ for the hash table of signatures as well as text. We must add $O(k)$ because the length of author's signature takes linear time. Overall, searching for string matches takes $O(n + k)$.