

CSc545 - fall 2018 - Homework #1.

Due: Sep 22 2018 11:59pm

Jordan L. Siah

September 22, 2018

Instructions.

1. Solution may **not** be submitted by students in pairs.
2. You may submit a pdf of the homework, either printed or hand-written and scanned, as long as it is **easily** readable.
3. If your solution is illegible not clearly written, it might not be graded.
4. Unless otherwise stated, you should prove the correctness of your answer. A correct answer without justification may be worth less.
5. If you have discussed any problems with other students, mention their names clearly on the homework. These discussions are not forbidden and are actually **encouraged**. However, you must write your whole solution yourself.
6. Unless otherwise specified, all questions have the same weight.
7. You may refer to data structures or their properties studied in class without having to repeat details, and may reference theorems we have studied without proof. If your answer requires only modifications to one of the algorithms, it is enough to mention the required modifications, and the effect (if any) on the running time and on other operations that the algorithm performs.
8. In general, a complete solution should contain the following parts:
 - (a) A high level description of the data structures (if needed). *E.g. We use a binary balanced search tree. Each node contains, a key and pointers to its children. We augment the tree so each node also contains a field...*
 - (b) A clear description of the main ideas of the algorithm. You may include pseudocode in your solution, but this may not be necessary if your description is clear.
 - (c) Proof of correctness (e.g. show that your algorithm always terminates with the desired output).
 - (d) A claim about the running time, and a proof showing this claim.

1. Consider an infinite sequence of functions f_i . It is known that for every i , it is true that $f_i(n) = O(n)$. We define a new function $g(n) = \sum_{i=1}^n f_i(n)$. Is it true that $g(n) = O(n^2)$?

Answer: We can rewrite this as a summation series:

$$g(n) = f_1(n) + f_2(n) + f_3(n) + \dots + f_n(n)$$

And we know that $f_i(n) = O(n)$,

so we can replace every $f_i(n)$ with $O(n)$,

$$\text{so we get } g(n) = \sum_{i=1}^n O(n)$$

and from the arithmetic series defined for insertion sort in the slides, it would be valid to say that $g(n) = O(n^2)$

Insertion sort analysis

Worst case: Input reverse sorted.

$T(n) = 2c + 3c + 4c + \dots + c(n-1) = cn(n-1)/2$

$T(n) = \sum_{j=2}^n \Theta(j) = \Theta(n^2)$ [arithmetic series]

2. Assume that we insert the keys $1, 2 \dots n$ into a binary search tree T . The algorithm for inserting a key k_i is simple. Initially the tree is empty. Then for $i = 1, 2 \dots n$, to insert key k_i we perform a search for k_i starting at the root of T , and once reaching a NULL pointer, we insert k_i as a new leaf. For simplicity, assume $k_i = i$.

We do **not** balance the tree after each insertion.

As usual $h(T)$, the height of T denote the number of edges from the root of T to the leaf that is the most remote from the root.

(a) warm-up

- i. show that the height of T is $\Theta(n)$.

Answer: Since it is given that the keys are inserted in increasing order from $1 - n$, this means that for every insert, the current key we are trying to insert is greater than the previous key that was inserted, therefore, every key inserted would forcefully end up being the right node of the previous key inserted and our tree height would grow by 1 for every insert, turning our tree into what is essentially a list. Therefore, the tree height of T is $\Theta(n)$.

- ii. Show that the time to perform all n insertions is $\Theta(n^2)$

Answer: If we imagine the tree as a list again as in the previous answer, we can also assume the properties of a list when inserting into this tree given that every $k_i = i$ for $i = 1, 2 \dots n$

This means that the insert operation in this tree takes $\Theta(n)$ since we are always inserting at the end of our imagined list.

We can therefore write a function $g(n)$ that represents all insert operations in this tree and write it as the summation:

$$g(n) = \sum_{k_i=1}^n \Theta(n) \text{ which is the same as } g(n) = \Theta(n^2)$$

- iii. next we insert the same keys into T but we could pick different permutations. Show that there is an order for which the time to perform all insertions is $O(n \log n)$

Answer: First, we need to use the assumption that the keys are given in the order from $1, 2, \dots, n$ (A sorted order) to our advantage.

This means that BEFORE we pick any keys to insert, we have an array of numbers going from $1, 2, \dots, n$

Using this fact about the numbers, we can perform a binary search style search on the keys to grab the middle element of each of the partitions of each binary search step.

This can be done because binary search splits the given array into 2 partitions using the middle element, yielding two partitions such that every element in the left partition is smaller than the middle element and every element in the right partition is bigger than the middle element.

This is important because if we were to choose such middle elements until all elements in the initially sorted array are chosen, then inserting these middle elements while keeping the order in which they were selected into the binary search tree would create a tree of height $\log(n)$ because every key chosen would have a left and a right child if there were at least 3 elements in each sub-partition array. By picking the middle element in each sub-partition array, we force all elements smaller than it to the left and all elements bigger than it to the right and repeat the process for every element. This should split the tree into 2 subtrees and we do this n times for all elements, therefore, each insertion takes maximum $\log(n)$ or $O(\log n)$ and we have n insertions, to perform, yielding a total of $O(n \log n)$ time to perform all insertions.

- iv. During the process of constructing the tree, we define random variable Y_{ij} to be 1 iff the keys i and j were compared to each other. (For every $1 \leq i, j \leq n$.) Otherwise $Y_{ij} = 0$. Show that the work to construct the tree is proportional to

$$\sum_{1 \leq i, j \leq n} Y_{ij}.$$

For example if j is the first key that was inserted, then it is in the root of T and it is compared to every other key inserted afterward.

Answer: Using the fact that the keys are inserted in an already sorted order, we would then know for certain that before every new insertion, we would hit every node currently in the binary search tree before inserting a new node, therefore every new node is compared to every other node smaller than it. This also means that each new key is always inserted as the right node of the previous key.

We can think of the above summation as everytime ANY key i is compared to any other key j , then $Y_{ij} = 1$, and since every key will be compared to each other by the property of the insertion order stated above, then we can say that the work to construct the tree is proportional to the summation since we can guarantee that every single key is compared to every other key before it and every other key after it.

This means that we can guarantee the claim that the amount of work required

to construct the tree is $= O(\text{number of comparisons})$ or $\theta(Y_{ij})$

- (b) Next assume that the keys are inserted in a random order, where each permutation is equally likely.

- i. What is the expected time

$$E \left(\sum_{1 \leq i, j \leq n} Y_{ij} \right) ?$$

Answer: Using the logic from the previous answer, we only change that the expected chance that key i is compared to key j is halved for every new level of the tree that we move to from the root, that is, the odds of i and j being in the same subtree assuming a random binary search tree permutation is $O(\log n)$.

- ii. Prove that the expected distance of a key j from the root is $O(\log n)$

Answer: A randomly built binary search tree has expected height $O(\log n)$, meaning that n items were inserted in a random order and therefore the search time for an item j is $O(h)$ where h is the height of the tree, and since the expected height of the tree is $O(\log n)$ for a random permutation of the n inputs, the maximum expected distance of j from the root is $O(\log n)$.

- iii. Prove that for large enough constant c , and large enough value of n , the probability that $h(T) \leq c \log_2 n$ is very close to 1. You could pick what is c and how to formalize 'close to 1'

3. Show the preference list of a set of n men and n women, for which more than one single stable pairing exists.

Answer: We can consider the following preference lists for 3 men and 3 women

$M_1 : W1, W2, W3$ — $W1 : M_1; M_2; M_3$

$M_2 : W1, W3, W2$ — $W2 : M_2, M_3, M_1$

$M_3 : W2, W3, W1$ — $W3 : M_3, M_1, M_2$

Since M_1 and $W1$ are at the top of each others' preference lists, they will be matched with each other in any stable pairing to prevent them from going rogue.

That leaves any combination of the rest of the two couples where Both M_2 and M_3 can be matched to either $W2$ or $W3$ without rogue couples.

This gives us the following 2 stable pairings:

$P1 = (M_1, W1), (M_2, W3), (M_3, W2)$

$P2 = (M_1, W1), (M_2, W2), (M_3, W3)$

4. Suggest an $O(n^2)$ -time algorithm that determines if, given the preference lists of all the men and women, there exists more than one stable matching.

Answer: We can run the stable-marriage algorithm defined in lecture many times with all of the different orders of males and if the algorithm succeeds more than once, then we know there exists more than one stable matching.

5. Suggest a modification of the SkipList structure, such that in addition to the operations $\text{insert}(x)$, $\text{find}(x)$ and $\text{delete}(x)$, you could also answer the operation $\text{avg}(x_1, x_2)$ which reports the average of all keys stores in the skip list whose value is $\geq x_1$ but $\leq x_2$. The expected time for each operation should be $O(\log n)$.

Hint: Start by storing at each node v of the SL another field, called $size[v]$, containing the number of keys in the SL between v and the next node at the same level as v . Note that maintaining the values of these fields might imply extra work while performing other operations.

Answer: The key to solving this problem lies in adding extra operations to the $insert(x)$ operation that won't add to the total time complexity.

Firstly, for every insert, we need to store in that node the field v as suggested in the hint, and this operation would have a run time of $O(1)$ because all we'd need to do is before we insert any node, we check if the current node's next pointer is greater than the node we want to insert, if it is, then we increment its v count by 1, and if not, we go right if possible and down if not. Once a node is inserted, if it is to be bubbled up to the next level up, then we simply check the previous node on the next level up and its next node, then subtract 1 from the previous one's v and add that to the current one's v , this is also a $O(1)$ operation.

We also need to add a sum value to each node after insertion to keep track of the sum of every number less than the node we are trying to insert at all times. First we find the spot in which we want to insert the new node, this is just the $find(x)$ operation, and in the case that we should insert between two nodes, we simply take the previous node's sum and add it to the new node's sum value and iterate through the rest of the nodes in the skip list and update the sums accordingly. If we were deleting instead, then at this step we would subtract the value of the node to delete from the rest of the nodes in the skiplist after it, this still keeps $delete(x)$ an $O(\log n)$ operation. This entire operation is $O(c \log(n))$ where c represents how many nodes from the current node to the end we need to update the sum values of.

Once all of the insertions are done, to find the $avg(x_1, x_2)$, we simply do:

$$\frac{find(x_2).get(sum) - find(x_1).get(sum)}{find(x_2).get(v) - find(x_1).get(v) + 1}$$

Every operation here is a $O(\log(n))$ operation, therefore total runtime of this skiplist modification would be $O(\log n)$ for all of the operations.

6. (a) Let d be a fixed positive integer. Next consider a perfect SkipList constructed as follows: In order to create the i th level L_i of the SkipList, we scan the keys of level L_{i-1} , and promote to L_i every d 'th key. So for example, the perfect SkipList discussed in class uses the value $d = 2$. The case $d = 3$ implies that every third key is promoted, and so on.

- i. What exactly is the worst case running time of $find(x)$, as a function of both d and n ?

Answer: In a perfect skiplist where we control which d 'th key is promoted, the height should be capped at and never exceed $\log_d n$.

This means that the worst case is we have to find an element that is bigger than all elements in the list, but isn't in the list. This is the worst case possible because it would

require the traversal of every level of the skiplist, therefore, this runtime is just $O(\log_d n)$

- ii. Which value(s) of d will do you think would lead to good performances, and which are poor choices? Why?

Answer: Smaller values of $d > 1$ will lead to better performances because

the smaller the value is, the more levels that will be created in the skip list, and the more levels there are, the more $\log(n)$ performance we get because we can always skip between even more elements. Bigger values would theoretically be better if we were doing a lot of searches on the end points of the list, but if we are talking in a general sense, bigger values would be bad because there would be less levels theoretically, and if we do a search for a value that is in the middle of the list, that would be costly as we would get something closer to $O(n)$ performance. With a skiplist, we want to go down a level as little often as possible, and stay as high as we can ideally, and as d increases, so does the chance of having to go down a level to find what we are looking for since there are less nodes on the higher levels than the lower ones.

- (b) In this question, consider a SkipList \mathcal{L} created by inserting a set S of n keys into an (initially empty) SkipList. As seen in class, if a key x appears in level i , the probability that it also appears in level $i + 1$ is $\frac{1}{2}$.

Assume that we re-create a SkipList by inserting the same keys, in the same order, but this time this probability p is 0.01. (if a key x appears in level i , the probability that it also appears in level $i + 1$ is 0.01.) Will the expected time to perform $\text{find}(x)$ operation increase or decrease, compared to the expected time for the same operation in the original SkipList \mathcal{L} ?

Answer: The expected time to perform the $\text{find}(x)$ operation would increase because since the chance of a node in the skiplist being promoted has decreased from $\frac{50}{100}$ to $\frac{1}{100}$, so did the chance of a new level in the skiplist being created. And explained in 6A above, the less levels we have, the slower the $\text{find}(x)$ operation on the skiplist is going to be. This means we would be getting closer to $O(n)$ performance from $O(\log n)$.

7. What is the expected value of cnt after the following function is executed ?

```

cnt = 0; M =  $-\infty$  ;
for i = 1 to n {
     $x_i = \text{rand}()$  ; Comment: returns a random float number of between 0 and 1.
    if ( $x_i > M$ ) then {
        M =  $x_i$ ;
        cnt ++;
    }
}

```

Hint: Define the random variable Y_i which is 1 if

$$x_i > \max\{x_1, x_2 \dots x_{i-1}\}$$

and $Y_i = 0$ otherwise. What is $E(Y_1 + Y_2 + \dots + Y_n)$.

Answer: If we define the variable Y_i which is 1 if $x_i > \max\{x_1, x_2 \dots x_{i-1}\}$, then on the first iteration of the loop, this variable is guaranteed to be 1 since anything is bigger than negative infinity, so count is at least one from the beginning. This means that Y_i is 1.

For the next iterations of the loop, we're going to choose some number between 0 and 1,

and whatever number we choose, we divide all numbers between 0 and 1 into 2 partitions using the random number between 0 and 1 that we got as reference point. One side of the partition represents $Y_i = 0$ or $Y_i = 1$.

If we assume that we get a nice even spread of random numbers, then each side that the random number lands on should be roughly $\log(n)$, because the odds of getting something higher than what we got last time is proportional to how many divisions (which side) in we are. In an arithmetic sequence, assuming the nice even spread of numbers, this would look something like:

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \dots + \frac{1}{n} = \log(n)$$

Therefore, the expected value of cnt is $\log(n)$.

8. Create a SkipList for a set of n keys $S = \{k_1 \dots k_n\}$. The keys are known in advance, and are sorted. The height of the Skiplist is ≤ 3 .

The SkipList should be designed such that

$$\max_{k_i \in S} T(k_i)$$

is as small as possible, where $T(k_i)$ is defined as the time for searching k_i , where the search is starting, as usual in the upper leftmost element.

Answer: Since we are given a maximum height for the skiplist, we can use this fact to create something closer to a perfect skiplist to minimize the maximum search time for an element. This means that we can *bubbleup* every 2 elements in the skiplist to end up with exactly (+or - 1) half of all of the elements on the second level of the list, and a quarter of the elements on the 3rd level. Since we know the height and we can force bubble up, this property will hold strongly and the elements that get bubbled up will be evenly spaced out, therefore yielding a hard $O(\log n)$ maximum search time for the skiplist with any n .

9. Let L be a perfect SkipList constructed on a set of n keys, and let $X = \{k_1 \dots k_m\}$ be another set of keys. Assume $m < n$. Suggest an $O(m \log(\frac{n}{m}))$ time algorithm for computing which key(s) of X appear in L . You could assume that X is sorted.

You could **not** assume that the keys of X are equally spaced in L . Yet understanding this special case could shed some intuition.

Answer: For this one we can search for the largest key in X first, that way when we search for other keys smaller than this key in L , we know to never go past that previous largest most recent key we just searched for, thereby reducing the search time by m , the number of keys in X .