Elissa Ito October 14, 2022

# 6.1210 Problem Set 3

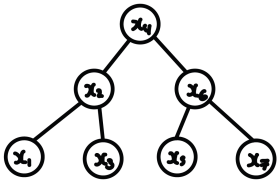
## Problem 1 *(Collaborators: None)*

(a) The INSERT method described in class performs as follows given root r and element to be inserted x, starting with the node at r:

1) checks if k is less than or equal to node
   i) if it is, checks if node has a left child
      a) if it does, repeats from step 1 with the left child as the node
      b) if it does not, insert x as the left child
   ii) if it is, checks if node has a right child
      a) if it does, repeats from step 1 with the right child as the node
      b) if it does not, insert x as the right child

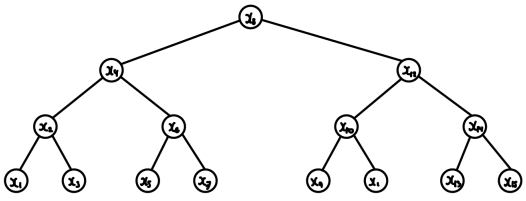
Thus to perform n inserts of elements $x_1 < x_2 < ... < x_n$, one can simply insert in order of $x_1, x_2, ...x_n$. This will cause $x_1$ to be the root, $x_2$ its right child, $x_3$ a BST of height $n-1$. Each $i$th insertion is done at depth $i-1$, thus the total time for the n insertions will be $\sum_{i=0}^{n-1} i = (n-1)^2/2$ which is $O(n^2)$.

(b) The 7 elements $x_1 < x_2 < ... < x_7$ should be added in the order of $x_4, x_2, x_6, x_1, x_3, x_5, x_7$,



yielding the BST: which has a height of 2.

(c) Given 15 elements $x_1 < x_2 < ... < x_{15}$, only 1 BST of height 3 can be made, since to fit all 15 elements in 4 levels $x_8$ has to be the node with $x_4$ as its left child and $x_{18}$ as its right child, and so on as depicted.



(d) Given 7 elements $x_1 < x_2 < ... < x_7$, only 1 binary tree is possible as shown in part b). However, there are 80 possible orders of insertion.

1) The first insertion must be $x_4$

2) The second insertion can either be $x_2$ or $x_6$.

i) If we inserted $x_2$, then we can choose to insert between $x_6, x_1, x_3$.

a) If we choose $x_6$, then the remaining 4 insertions have $P(4,4) = 4! = 24$ possible orders.

b) If we choose $x_1$, the successive choice has to be $x_6$ or $x_3$.
i) If we choose $x_6$, the remaining three elements can be inserted in $P(3,3) = 3! = 6$ possible orders.
ii) If we choose $x_3$, $x_6$ has to be inserted next, then the remaining two elements can be inserted in $P(2,2) = 2! = 2$ possible orders.
This yields 8 possible orders when $x_1$ is inserted third.

c) Similarly, if we insert $x_3$ third, there will be 8 possible orders.

Inserting $x_2$ second yields 40 possible orders of insertion.

ii) Similarly, if we insert $x_6$ second, that would yield 40 possible orders of insertion.

# Problem 2 *(Collaborators: None)*

## Part 2(a)

Alice should use the priority queue implementation of a heap, where each node has the properties dog $d$ and score $s_d$, and the dogs are heap sorted by score. A heap has operations build in $O(n)$, insert in $O(logn)_{am}$, delete-max in $O(logn)_{am}$. Thus build can be used to construct the data structure, insert can be used to insert dogs and their scores, and delete-max can be used to remove the highest-scoring dog and its score each meeting the runtime criteria.
A basic algorithm will be to:

1. Store the first node of the array at index i=1, which should be the largest key = highest score.
2. delete-max() the array (max-heapify-down() assumed to be part of delete-max()).
3. Repeat step 1 and 2 k times.
4. Insert the k stored nodes. (max-heapify-up() assumed to be part of insert()).

Returned the stored nodes; they are the k highest scoring dogs.
Step 1 takes $O(1)$, step 2 takes $O(logn)_{am}$. Step 3 repeats the three steps k times, so overall step 1-3 takes $O(klogn)$. Inserting k times takes $O(klogn)$. The overall algorithm thus takes $O(klogn)$. The algorithm is correct because we know each function works from lecture/recitation, thus the combined steps should work as well.

## Part 2(b)

Keep an AVL Tree of the $1_{st} - k_{th}$ highest scoring dogs (to be called Tops).
Keep a Heap of the $(k+1)_{st}$-last highest scoring dogs (to be called Bottoms).

1. build()
   (a) Build a priority queue implementation of an Heap with all n dogs, Bottoms.
   (b) Find and store the max element (aka the first element) in Bottoms.
   (c) Delete that max element in Bottoms.
   (d) Repeat steps (b) and (c) k times.

   (e) Build a set implementation of an AVL tree with the k stored dogs, Tops.

2. insert(x)
   (a) Find min $m$ of Tops.
   (b) If score of $x$ is smaller than or equal to $m$ (i.e. smaller than or equal to the kth highest score):
   i. Insert $x$ into Bottoms.
   (c) If score of $x$ is greater than $m$ (i.e. greater than the kth highest score):
      i. Delete $m$ from Tops.
      ii. Add $m$ to Bottoms.
      iii. Add $x$ to Tops.

3. del-max()
   (a) Find max $m$ of Tops.
   (b) Delete $m$ from Tops.
   (c) Find and store max $f$ (aka first element) of Bottoms.
   (d) Delete $f$ from Bottoms.
   (e) Insert $f$ into Tops.

4. display-highest-k()
   i. Return Tops

Algorithm Analysis:

1. build()
   Correctness: We know all steps work independently from lecture. When run in succession, steps (b)-(d) successfully remove the top k highest scoring dogs from Bottoms, and places them into Tops.
   Runtime: Step (a) takes $O(n)$. Step (b) takes $O(1)$. Step (c) takes $O(logn)_{am}$. Step (d) performs step (b) and (c) k times, thus steps (b)-(d) take $O(klogn)$. Step (e) takes $O(klogk)$. The overall algorithm takes $O(n)+O(klogn)$ which is smaller than $O(nlogk)$ since $k \leq n$.

2. insert()
   Correctness: We know all steps work from lecture. If the score to be inserted is smaller than the kth highest score, it should be added to the heap which stores the k+1th to last place scores. If it is larger than the kth highest score, it should be placed in the AVL tree which stores the top to kth highest scores, and the lowest score in the AVL tree should be placed in the heap - which is what the algorithm does.

Runtime: Step a) is $O(logk)$. Step b) is $O(1)$. Step bi) is $O(logn)_{am}$. Step c) is $O(1)$. Step ci) is $O(logk)$. Step cii) is $O(logn)_{am}$. Step ciii) is $O(logk)$. The whole operation is $O(logn)$.

3. del-max()
   Correctness: To delete the maximum score, the algorithm should delete the max in the AVL tree which stores the 1st-kth highest scores, then remove the top score in the k+1th-last scores stored in heaps and insert that into the AVL tree, which it does.
   Runtime: Tops is always size k (has k elements). Step a) thus takes $O(k)$. Step b) takes $O(logk)$. Step c) takes $O(1)$. Step d) takes $O(logn)$. Step e) takes $O(logk)$. The overall operation thus takes $O(logn)$.

4. display-highest-k()
   Correctness: The AVL tree always stores the top k scoring dogs, thus returning the AVL tree is correct.
   Runtime: $O(1)$ since we are just returning a pointer to the tree.

# Problem 3 *(Collaborators: None)*

## Part 3(a)

The data structure will be a set implementation of an AVL Tree, with each node being a movie with properties representing its date ($s_m$), id ($m$), and rank ($r_m$), as well the subtree properties: date ($s_r$), id ($r$) and rank ($r_r$) of the highest ranking movie that starts on or after $s_m$. The keys to the AVL tree are the node's (movie's) start dates ($s_m$). The augmentation will be maintained as follows:

1. If a node is a leaf, the highest ranking movie that starts on or after its stream date will be itself. i.e. $s_r = s_m, r = m, r_r = r_m$.
2. If 1. is false, if the node has a left child whose stream date is the same as its own, retrieve the $s_{rl}, rl, r_{rl}$ values i.e. the subtree properties of the left child which correspond to the highest ranking movie that starts on or after the left child's stream date.
3. If 1. is false, and the node has a right child, retrieve the $s_{rr}, rr, r_{rr}$ values i.e. the subtree properties of the right child which correspond to the highest ranking movie that starts on or after the right child's stream date.
4. Find the maximum ranking among $r_m, r_{rl}, r_{rr}$ (if they exist) and overwrite the corresponding id, date, and rankings to this node's highest ranking id, date, and ranking properties.

Step 2 and 3 can each be computed in $O(1)$ since retrieving a property of a node and binary comparisons are both $O(1)$. Since the subtree properties can be computed from augmentations of its children in $O(1)$ time, the three subtree properties can be maintained without changing dynamic operation costs of an AVL tree.

1. insert($m, s, r$): Inserts movie with ID $m$, starting date $s$ and ranking $r$.
   Algorithm: as shown in recitation 8, with added properties self.highestRankId, self.highestRankDate, self.highestRankRank. Initially, these can be set to the node's own id, rank, and date but when the maintain function is run (as described above), the node will store the accurate information about the highest ranking movie which starts on or after the node's date s.
   Runtime: Since the requirements for maintaining subtree properties without changing dynamic operation costs are met, the runtime is the same as a typical AVL Tree insertion: $O(logn)$

2. delete($m$): Deletes movie with ID $m$.
   Algorithm: Algorithm is the exact same as shown in recitation 8.
   Runtime: Runtime is the same as a typical AVL Tree deletion: $O(logn)$

3. earliest($s$): Outputs the movie that can be streamed earliest on or after the given date $s$.
   Algorithm: Return find(self,s), if nothing is returned return findnext(self,s).
   Runtime: Since both functions are $O(logn)$, $O(logn) + O(logn) = O(logn)$.

4. highest($s$): Outputs the movie that has the highest ranking and which starts on or after the given date $s$.
   Algorithm: Find the movie that can be streamed earliest on or after the given date using earliest($s$), then return the highest ranking id property of that node. If earliest($s$) returns none, return none.
   Runtime: Runtime is the earliest($s$) and finding the property of a node: $O(logn) + O(1) = O(logn)$.

## Part 3(b)

Code submitted to alg.mit.edu