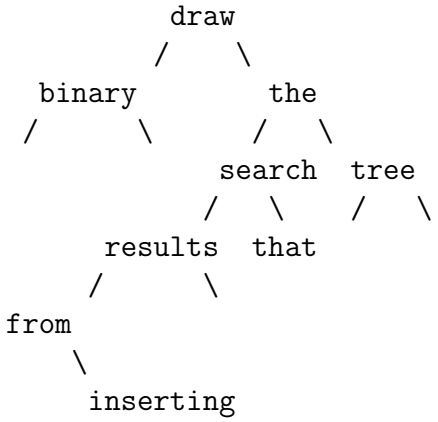
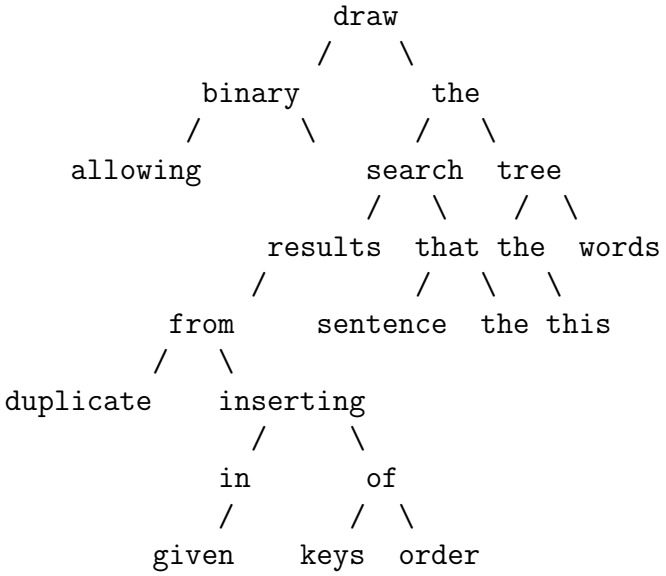
CSE250 "Virtual Assignment 7" Answer Key Spring 2022

(1) Draw the binary search tree that results from inserting the words of this sentence in the order given, allowing duplicate keys. Use alphabetical order of lowercased words with the lower words at left. Then show the results of deleting all three occurrences of the word "the", one at a time. (It is OK to use either the inorder successor or predecessor for deletion, and putting an equal key left or right, but please show each step separately on the relevant part of the tree—you do not have to re-draw the whole tree each time. A virtual 12 + 9 = 21 pts.)

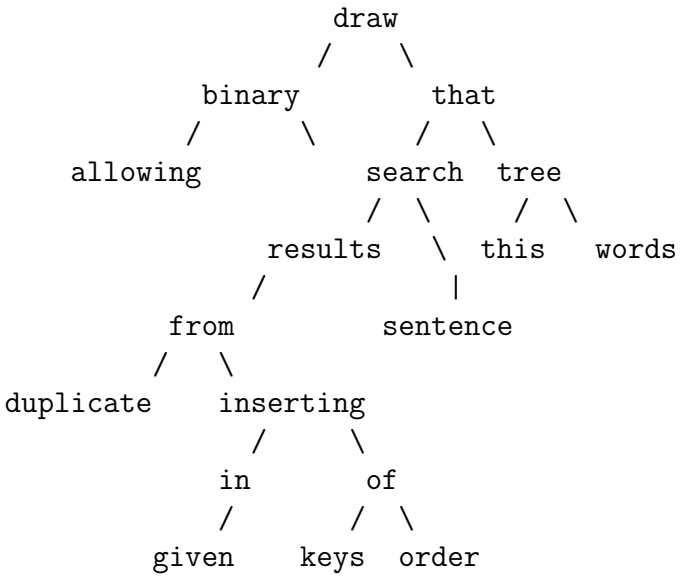
Answer: Up to the second occurrence of `the`, it must be:



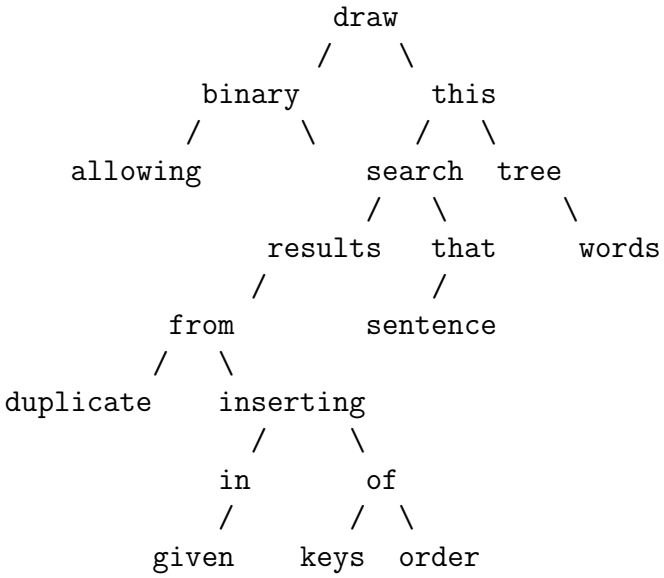
Now for the duplicate `the`, the left-leaning strategy puts it as the right child of `that`, while the right-leaning strategy makes it the left child of `tree`. Either is fine, and it isn't even necessary to be consistent—so long as the tree is legal, the 3 occurrences of `the` will be consecutive in the inorder transversal, that's fine. Let's do the latter first and then the former:



For the three deletions, it is interesting that whatever way you used to insert and whatever order you delete them in, the only difference is whether you use the inorder predecessor or the inorder successor as the "victim" when deleting `the` as the right child of `draw` (for the last time, if you use the leaf or elbow `the` as victim first). In the predecessor case, the tree is



In the other case, you get



(2) For each task below labeled 1.–8., say which of the following best describes its running time:

(a) Guaranteed $O(1)$ time.

(b) Amortized $O(1)$ time.

(c) Usually $O(1)$ time.

(d) Guaranteed $o(n)$ time.

(e) Usually $o(n)$ time.

(f) Guaranteed $O(n)$ time.

In all cases n denotes the number of items currently in the underlying data structure, and any other parameters are stated. Note that the answer "$o(n)$ time" can convey either $O(\log n)$ time or $O(\sqrt{n})$ time, or other *sub-linear* times. The variable `vec` stands for a vector, `dlist` for a doubly-linked list (not necessarily sorted), `ba` for a "`BALBOADLL`" data structure (assuming it has just been newly created or refreshed, no deletions), `arr` for an `ArrayBuffer` (or the `SortedArray` class in the ISR repository, same answers), `bst` for a BST—i.e. a basic binary search tree without balance guarantee, `itr` for an iterator of the appropriate kind, `comp` for a typical comparator function, and `item` for a typical item in the data structure. *Justifications* are not required, but might help for partial credit. The tasks (for a virtual $8 \times 3 = 24$ pts.) are:

1. For a BST iterator `itr`, the call `itr.next()` *Answer:* Amortized $O(1)$ time, which is stronger than saying Usually O(1) time.

2. `dlist.pushRear(item);` (equivalently, `dlist.insert(item,dlist.end)`) *Answer:* Always $O(1)$ time, guaranteed, which is part of the theoretical advantage of linked lists over vectors for adding/erasing data in already-known locations.

3. `dlist.remove(itr);` *Answer:* Guaranteed $O(1)$ time, since just a few links need to be spliced.

4. `arr.remove(itr);` where `itr` points in the middle of `arr` *Answer:* Guaranteed $O(n)$ time, and one can really not say better—this kind of surgery basically always "munges" the array so it has to be re-allocated.

5. For a `BALBOADLL` iterator `itr`, the call `itr.next()` *Answer:* Guaranteed $O(1)$ time. (Technically, this presumes that when an array becomes empty, its node is spliced out of the linked list, so that the iterator never has to waste time stepping thru empty nodes when it has gone past the ened of a previous node's array. In hash tables this becomes a real issue with empty buckets.)

6. `bst.find(item)` *Answer:* In a "scraggly tree" this can take n steps, so it is not guaranteed $o(n)$ time. But it is usually $O(\log n)$ time since "random data" makes fairly balanced trees, and since the $\log(n)$ function is a case of "$o(n)$," this is Usually $o(n)$ time.

7. Preorder traversal of a BST. *Answer:* The full transversal is guaranteed $O(n)$ time, regardless of how scraggly the tree is.

8. For a BALBOADLL object `ba` and index $j < n$, the call `ba(j)`. This supposes that indexing is simulated by running down the linked list and adding up the `length` of each constituent array until the sum goes over `j`, in which case the current array `arr` has the jth overall element. If `s` was the sum before adding the size of that array, so `s <= j`, then return `arr(j - s)`. *Answer:* If there are r arrays, each about $m \approx n/r$ elements, then you spend $O(r)$ time using the sums to find the array where the element would be. Then the indexed lookup in that array is $O(1)$ more time. Presuming the tradeoff is maintained so that $r = o(n)$, for instance $r \approx \sqrt{n}$, this is Guaranteed $o(n)$ time.

(3) Show how a binary search tree `bst` can implement indexing `bst(j)` in $O(h)$ time, where h is the height of the tree. In particular, if the tree is *balanced*—which I specify to mean $h \leq 2\log_2 n$ for a tree of n nodes overall—then this means indexing is in $O(\log n)$ time. The needed wrinkle is that each `Node` has an extra field `mySize: Int` which maintains the number of nodes in the subtree rooted at that node. So, for instance, the leaves are exactly those nodes with `mySize == 1`, and that might be a quicker check than both `left` and `right` being `endSentinel` (or being `null` in the text). And of course `root.mySize` equals n itself—this would supersede the BST class needing to maintain a `_size` field.

The particular case $j = (n-1)/2$ when n is odd means that the middle element of such a tree can be found in $O(\log n)$ time. Sketch your algorithm in Scala-like pseudocode and explain why it has the desired time. As usual, you may presume that reading and comparing fields of nodes are $O(1)$-time operations unto themselves. (A virtual 27 pts., making 72 on the virtual set.)

Answer: This is well described as a recursive method `bst(u,j)` to find the node of index j in the subtree rooted at u, presuming $0 \leq j <$ `u.mySize` with zero-based indexing. We can also suppose that the empty tree (or end sentinel) returns size 0. So take k to be the size of the left subtree. If $j = k$, return `u`. If $j < k$, then the left child v cannot be the end sentinel and node j must be in its subtree, so call `bst(v,k-j)`. Else, the node must be in the right subtree, so call `bst(u.right,j-k-1)`.

```
def bst(j,u) = {
   val ls = u.left.mySize
   if (j == ls) u else if (j < ls) bst(ls-j, u.left) else bst(j-ls-1, u.right)
}
```

Assuming j is correctly in range, the recursion must bottom out with a "$j = k$" case before it hits the end, and it descends by one level on each call. Hence it makes at most h recursive calls (after the initial call to `bst(j) = bst(root,j)`). Because the `mySize` fields is maintained rather than computed by traversal each time, the body of each call takes $O(1)$ time for the comparison and some basic operations, so the whole time is $O(h)$.