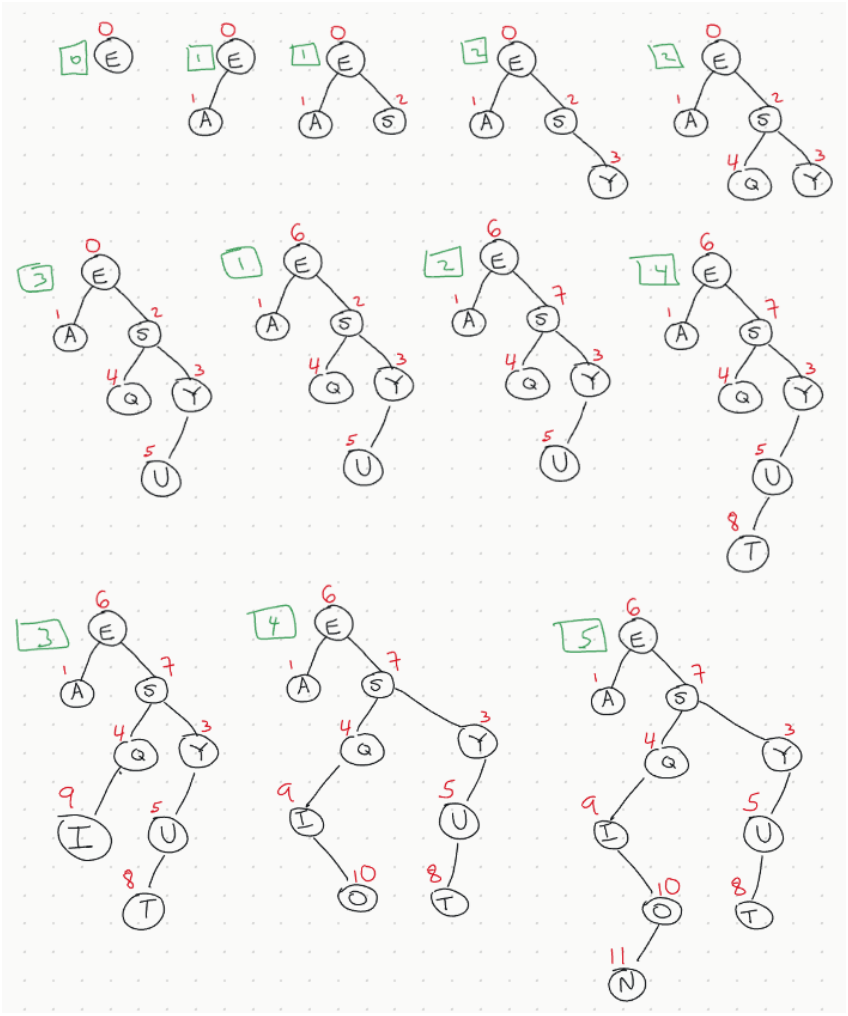

Figure 1: Exercise 1: Sequence of trees formed when inserting the keys `E A S Y Q U E S T I O N` into a binary search tree.

Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
December 1st, 2024

3.2: Binary Search Trees

Exercise 1. Draw the BST that results when you insert the keys `E A S Y Q U E S T I O N`, in that order (associating the value `i` with the `i`th key, as per the convention in the text) into an initially empty tree. How many compares are needed to build the tree?

Solution. The sequence of binary search trees is shown in Figure 1. The boxes next two the root show the number of compares necessary to complete the insertion at that step. The sequence of compare counts is: 0, 1, 1, 2, 2, 3, 1, 2, 4, 3, 4, 5. Adding gives a total of 28 compares.

Exercise 2. Inserting the keys in the order `A X C S E R H` into an initially empty BST gives a worst-case tree where every node has one null link, except one at the bottom,

which has two null links. Give five other orderings of these keys that produce worst-case trees.

Solution. We can insert them in the following ways:

(i) `A X C S E R H` (given).

(ii) `X S R H E C A`.

(iii) `A C E H R S X`.

(iv) `X A S C R E H`.

(v) `X A S R H E C`.

(vi) `A X C E H R S`.

See Figure 2.

Exercise 3. Give five orderings of the keys `A X C S E R H` that, when inserted into an initially empty BST, produce the *best-case* tree.

Solution.

(i) `H C S A E R X`.

(ii) `H S C A E R X`.

(iii) `H S C E A X R`.

(iv) `H S R X C A E`.

(v) `H S X R C A E`.

These all result in the same tree, depicted in Figure 3.

Exercise 4. Suppose that a certain BST has keys that are integers between `1` and `10`, and we search for `5`. Which sequence below *cannot* be the sequence of keys examined?

(a) `10, 9, 8, 7, 6, 5`

(b) `4, 10, 8, 6, 5`

(c) `1, 10, 2, 9, 3, 8, 4, 7, 6, 5`

(d) `2, 7, 3, 8, 4, 5`

(e) `1, 2, 10, 4, 8, 5`

Solution.

(a) This is a valid sequence. For example, this could be a tree where the keys were inserted in descending order.

(b) This is a valid sequence.

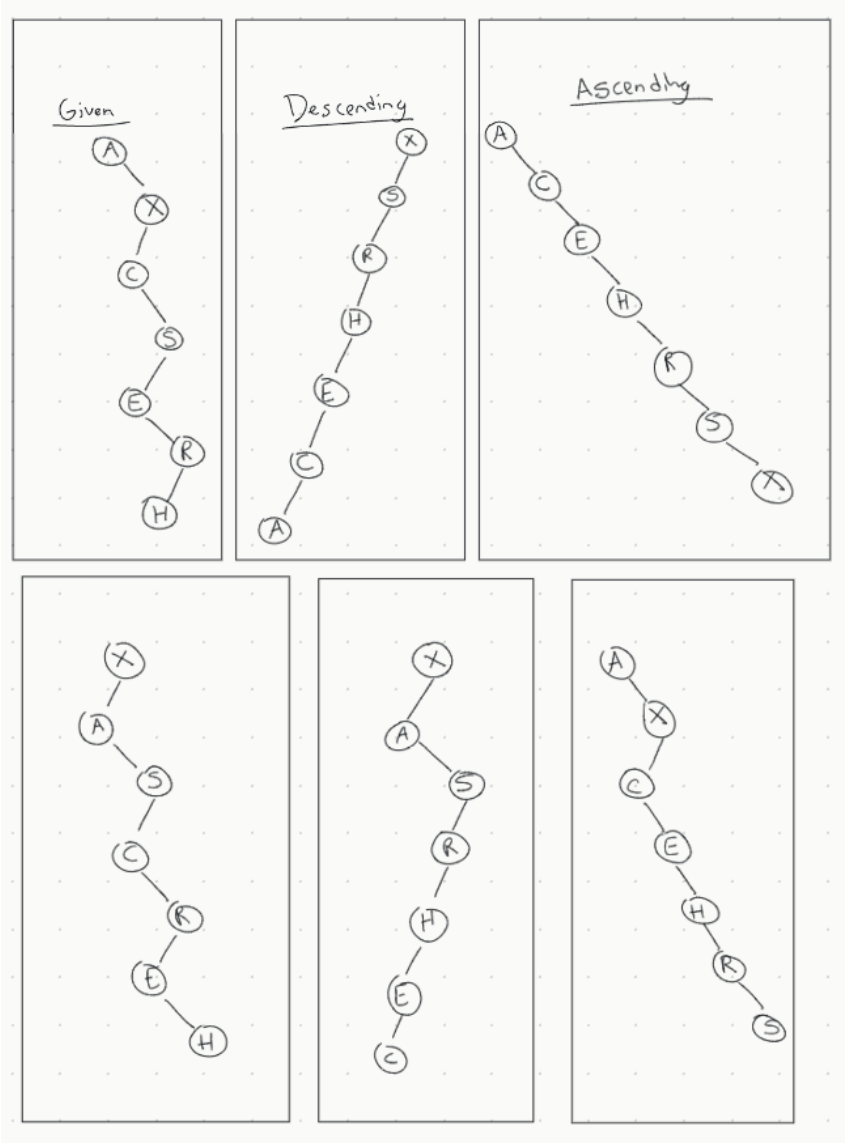

Figure 2: Exercise 2: Worst-case binary search trees created by inserting the keys `A X C S E R H` into an initially empty tree in a particular order

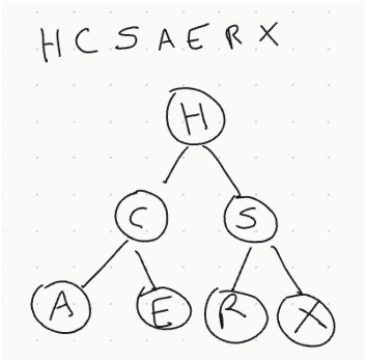

Figure 3: Exercise 3: Best-case binary search trees created by inserting the keys `A X C S E R H` into an initially empty tree in a particular order

(c) This is a valid sequence.

(d) This sequence cannot happen. The sequence of compares suggests the following:

(i) 2 is the root, and 5 is larger, so we go right.

(ii) 5 is smaller than 7, so we go left.

(iii) 5 is smaller than 3, so we go right.

(iv) 5 is compared against 8.

This last step is impossible because since 8 is larger than 7, but it appears on the subtree rooted at its left child, consisting of smaller keys, a contradiction.

(e) This is a valid sequence.

Exercise 5. Suppose that we have an estimate ahead of time of how often search keys are to be accessed in a BST, and the freedom to insert them in any order that we desire. Should the keys be inserted into the tree in increasing order, decreasing order of likely frequency of access, or some other order? Explain your answer.

Solution. Increasing order is not desirable because it leads to a worst-case tree where every internal node hade has 1 null link and the leaf node has 2 null links (in essence, a linked list).

Decreasing order of likely frequency of access biases towards the most frequency accessed nodes. However, if that order corresponds to the increasing or decreasing order of the keys, for example, then we may again have a worst-case tree.

The safest thing to do is to insert them in an order that yields a balanced tree. Suppose that there are n keys, and suppose we place them in sorted order, $k_0, k_1, \ldots, k_{n-1}$, so that $k_{i-1} \leq k_i$ for all $1 \leq i \leq n$. Let `lo = 0`, and `hi = n - 1` Then we can insert them as follows:

(i) If `lo > hi`, stop.

(ii) Compute the middle key at index `mid = lo + (hi - lo) / 2`. This key k_{mid} is the root.

(iii) Set the left child of k_{mid} to the tree resulting from (recursively) applying the algorithm to the subset of keys $k_0, \ldots, k_{mid-1}$.

(iv) Set the right child of k_{mid} to the tree resulting from (recursively) applying the algorithm to the subset of keys $k_{mid+1}, \ldots, k_{hi}$.

Notice this is effectively placing the partitioning the array like quicksort.

Exercise 6. Add to `BST` a method `height()` that computes the height of the tree. Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and a method like `size()` that adds a field to each node in the tree (and takes linear space and constant time per query).

Solution. See `com.segarciat.algs4.ch3.sec2.ex06.BST`.

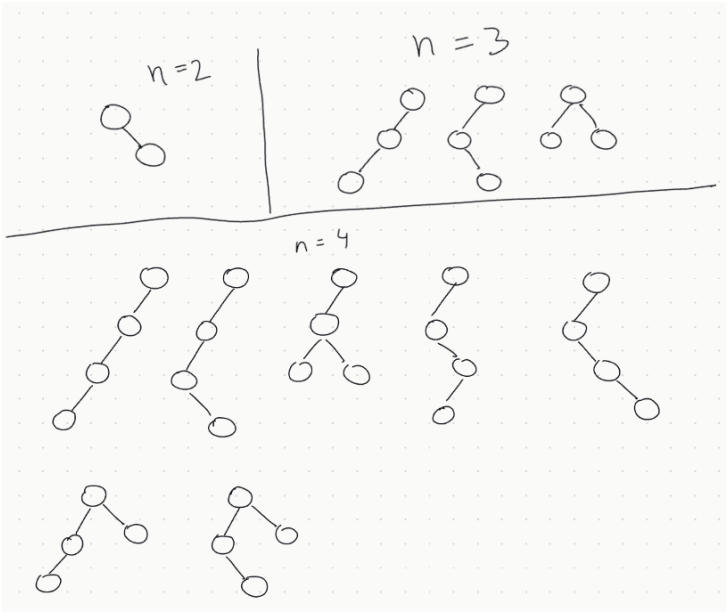

Figure 4: Exercise 9: All binary-search tree shapes for $n = 2, 3, 4$ vertices.

Exercise 7. Add to `BST` a recursive method `avgCompares()` that computes the average number of compares reuired by a random search hit in a given BST (the internal path length of the tree divided by its size, plus one). Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and a method like `size()` that adds a field to each node in the tree (and takes linear space and constant time per query).

Solution. See `com.segarciat.algs4.ch3.sec2.ex07.BST`.

Exercise 8. Write a static method `optCompares()` that takes an integer argument `n` and computes the number of compares required by a random search hit in an optional (perfectly balanced) BST with `n` nodes, where all the null links are on the same level if the number of links is a power of 2 or one of two levels otherwise.

Solution. See `com.segarciat.algs4.ch3.sec2.ex08.BSTOptCompares`.

Exercise 9. Draw all the different BST shapes that can result when `n` keys are inserted into an initially empty tree, for `n = 2, 3, 4, 5`, and `6`.

Solution. See Figure 4 for `n = 2, 3, 4`. I considered the reflections on the y-axis to be of the same shape, so I chose not to draw them.

Exercise 10. Write a test client for `BST` that tests the implementations of `min()`, `max()`, `floor()`, `ceiling()`, `select()`, `rank()`, `delete()`, `deleteMin()`, `deleteMax()`, and `keys()` that are given in the text. Start with the standard indexing client given on page 370. Add code to take additional command-line arguments, as appropriate.

Exercise 11. How many binary tree shapes of n nodes are there with height n? How many different ways are there to insert n different keys into an initially empty BST that result in a tree of height $n - 1$? (See **Exercise 3.2.2**)

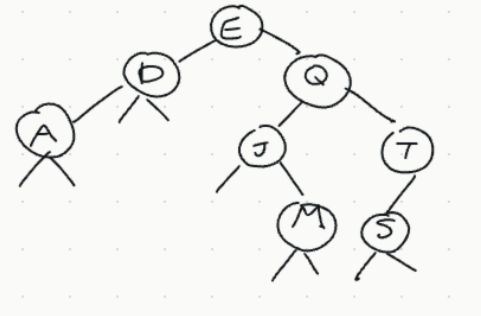

Figure 5: Exercise 15: Binary Search Tree

Solution. There are 0 binary tree shapes of n nodes with height n, because the height of a tree is the maximum depth among all of the nodes, and the smallest node is that of the root which is 0.

However, there are 2^{n-1} binary ways to insert n different keys into an initially empty BST that results in a tree of height $n-1$. A path from the root to the leaf in such a tree of height $n-1$ consists of a sequence of left and right turns. At each node, we make a decision of whether to go left or right. Since there are $n-1$ edges, and we choose to lean left or right at each step, we get 2^{n-1}.

Exercise 12. Develop a `BST` implementation that omits `rank()` and `select()` and does not use a count field in `Node`.

Solution. See `com.segarciat.algs4.ch3.sec2.ex11.BST`.

Exercise 13. Give nonrecursive implementations of `get()` and `put()` for `BST`. The implementation of `put()` is more complicated because of the need to save a pointer to the parent node to link in the new node at the bottom. Also, you need a separate pass to check whether the key is already in the table because of the need to update the counts. Since there are many more searches than inserts in performance-critical implementations, using this code for `get()` is justified; the corresponding change for `put()` might not be noticed.

Solution. See `com.segarciat.algs4.ch3.sec2.ex13.BSTNonRec`.

Exercise 14. Give nonrecursive implementations of `min()`, `max()`, `floor()`, `ceiling()`, `rank()`, `select()`, and `keys()`.

Solution. See `com.segarciat.algs4.ch3.sec2.ex14.BSTNonRec`.

Exercise 15. Give the sequences of nodes examined when the methods in `BST` are used to compute each of the following quantities for the tree drawn in Figure 5.

(a) `floor("Q")`

(b) `select(5)`

(c) `ceiling("Q")`

(d) `rank("J")`

(e) `size("D", "T")`

(f) `keys("D", "T")`

Solution.

(a) `"E"`, `"Q"`

(b) `"E"`, `"Q"`

(c) `"E"`, `"Q"`

(d) `"E"`, `"Q"`, `"J"`

(e) First the `contains(hi)` call leads to the following sequence: `"E"`, `"Q"`, `"T"`. Then the call to `rank(hi)` leads to the sequence `"E"`, `"Q"`, `"T"`. Then, the call `rank(lo)` leads to the sequence `"E", "D"`.

(f) `"E"`, `"D"`, `"Q"`, `"J"`, `"M"`, `"T"`

Exercise 16. Define the *external path length* of a tree to be the sum of the number of nodes on the paths from the root to all null links. Prove that the difference between the external and internal path length in any binary tree with n nodes is $2n$ (see **Proposition C**).

Solution.

Proof. Let E_n represent the external path length of a tree of size n, and I_n represent the internal path length of a tree of size n, where n is a positive integer. The proof is by induction on n.

For the base case, let $n = 1$. Since the root has one node, that node is the root at depth 0, and the internal node path length is 0. The root has two null links, and since the root is the only node on the path to them, it follows that the external path length is 2. Since $I_1 = 0$, $E_1 = 2$, and $E_1 - I_1 = 2 = 2 \cdot 1$, we see that the base case holds.

For the inductive case, suppose we have a tree of $n \geq 1$ nodes, and that $E_n - I_n = 2n$. Suppose that we add a new node, increasing the tree size to $n + 1$. If k is the depth of the new node, then the internal path length of the tree is now $E_{n+1} = I_n + k$.

When the new node was inserted, it replaced a null link, which contributed precisely k to the external path length; on the other hand, the new node contributed two new null links to the tree, and reaching each requires traversing through $k + 1$ nodes. Hence, the external path length of the new tree is $E_{n+1} = E_n - k + 2(k + 1) = E_n + k + 2$. Now see that

$$\begin{aligned} E_{n+1} - I_{n+1} &= (E_n + k + 2) - (I_n + k) \\ &= (E_n - I_n) + 2 \\ &= 2n + 2 \\ &= 2(n + 1), \end{aligned}$$

so the inductive step is asserted. By the principle of mathematical induction, the result now holds for all natural numbers. □

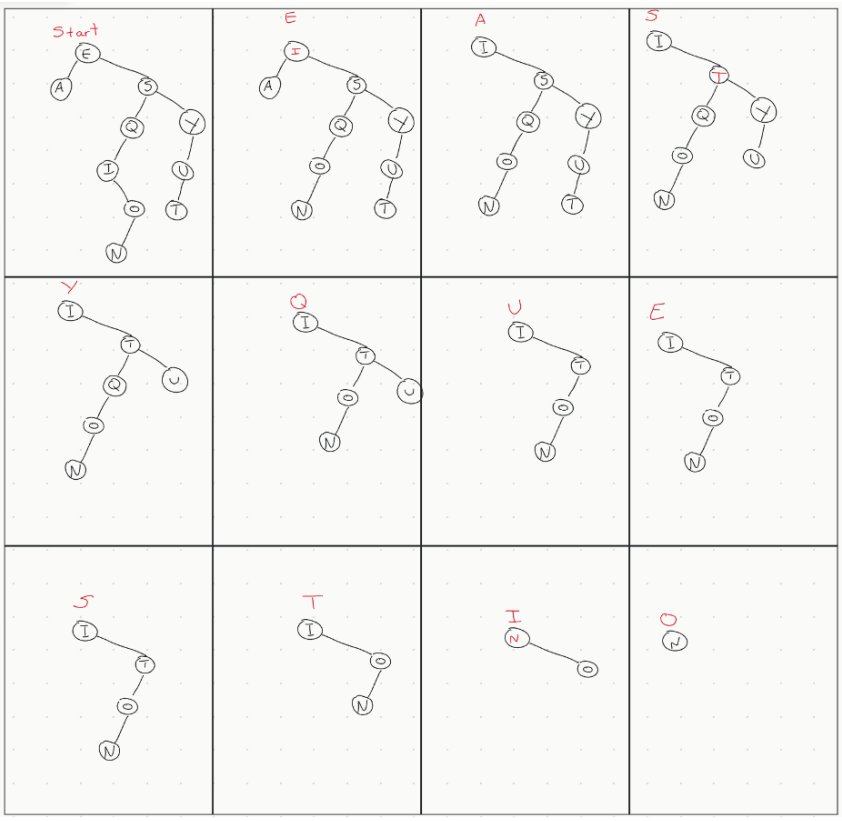

Figure 6: Exercise 17: Sequence of binary trees produced when deleting the keys from Exercise 1 in the order they were inserted.

Exercise 17. Draw the sequence of BSTs that results when you delete the keys from the tree of **Exercise 3.2.1**, one by one, in the order they were inserted.

Solution. See Figure 6.

Exercise 18. Draw the sequence of BSTs that results when you delete the keys of **Exercise 3.2.1**, one by one, in alphabetical order.

Solution. See Figure 7.

Exercise 19. Draw the sequence of BSTs that results when you delete the keys from the tree of **Exercise 3.2.1**, one by one, by successively deleting the key at the root.

Solution. See Figure 8.

Exercise 20. Prove that the running time of the two-argument `keys()` in a BST is at most proportional to the tree height plus the number of keys in the range.

Solution.

Proof. Suppose that the two arguments of `keys()` are `lo` and `hi`. The recursive depth is dictated by the ceiling of `lo` or the floor of `hi`.

The algorithm descends into a left or right subtree effectively searching for `ceiling(lo)` and `floor(hi)`, taking time proportional to the tree height in the worst case. Once found, further recursive calls that lead to keys out of range are skipped, and the recursive calls

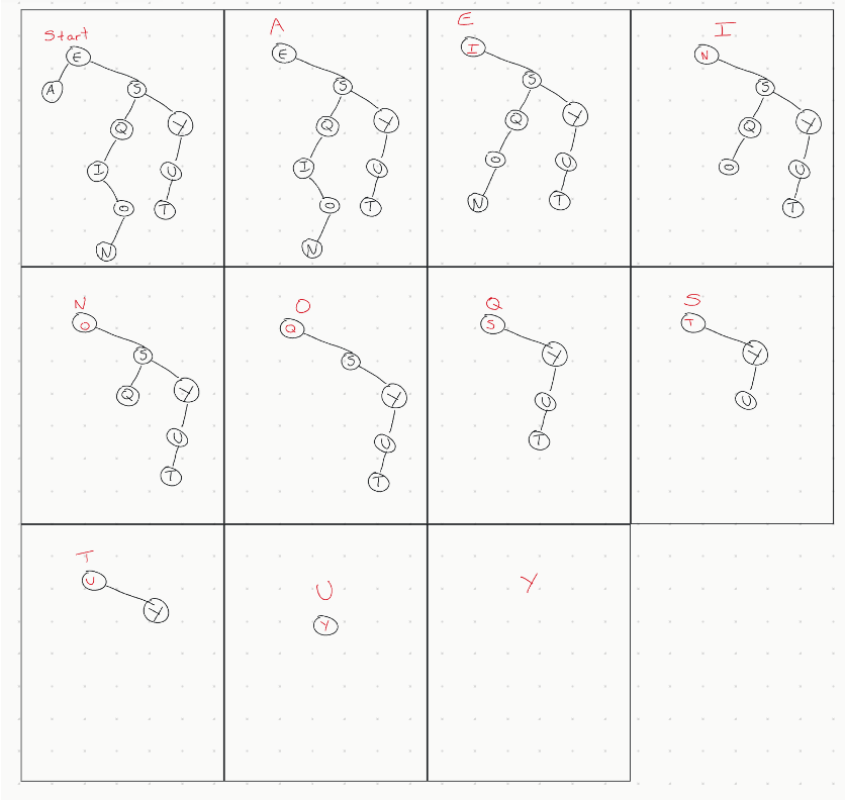

Figure 7: Exercise 18: Sequence of binary trees produced when deleting the keys from Exercise 1 in alphabetical order.

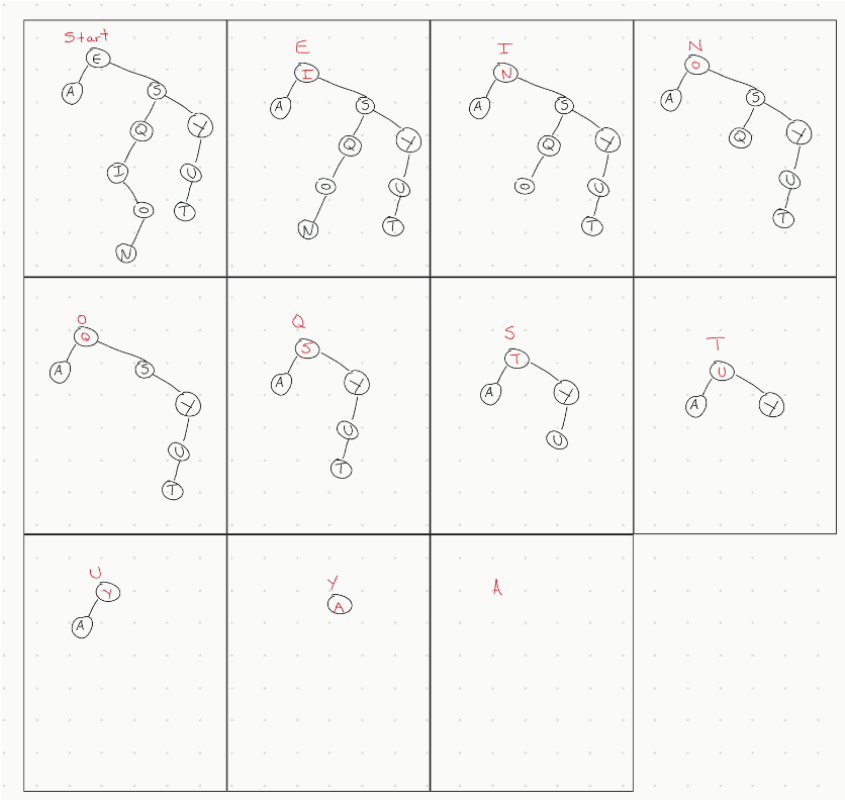

Figure 8: Exercise 19: Sequence of binary trees produced when successively deleting the roots from the final tree in Exercise 1 .

return to their caller to complete the inorder traversal. This means that the duration of the traversal corresponds directly to the number of keys in the range `lo..hi`.

Altogether, the running time of the two-argument `keys()` is bounded by the tree height plus the number of keys in the search range `lo..hi`. □

Exercise 21. Add a BST method `randomKey()` that returns a random key from the symbol table in time proportional to the tree height, in +the worst case.

Solution. We can easily implement this by using `select()`, passing as the rank `k` a random integer between 0 and `size() - 1`; see `com.segarciat.algs4.ch3.sec2.ex21.BST`.

Exercise 22. Prove that if a node in a BST has two children, its successor has no left child and its predecessor has no right child.

Solution.

Proof. Suppose `x` is a node in a BST with left child `y` and right child `z`. The successor is defined to be the next largest node in the binary tree. Since nodes with keys larger than the key of `x` are in the right subtree of `x`, this means the successor of `x` is in the subtree rooted at `z`.

Suppose the successor of `x` is called `z*` By definition of the successor, there are no nodes with keys between `x` and `z*`, and hence no nodes with keys smaller than the key of `z*` but larger than the key of `x`. This implies that the key of `z*` is the minimum of the keys of the nodes in the subtree rooted at `z`. Since `z*` has the minimum key of all nodes in the subtree rooted at `z`, it has no left child (otherwise, its left child would have the minimum key).

Similarly, the predecessor of `x` is the maximum node `y*` of the subtree rooted at its left child `y`. By a similar reasoning `y*` has no right child. □

Exercise 23. Is `delete()` commutative? (Does deleting `x`, then `y` give the same result as deleting `y`, the `x`)?

Exercise 24. Prove that no compare-based algorithm can build a BST using fewer than $\lg n! \sim n \lg n$ compares.

Solution.

Proof. Suppose that a compared-based algorithm was able to build a BST using fewer than $\lg n! \sim n \lg n$ compares. Since the keys in the BST are sorted, this implies that the algorithm can sort n items with fewer than $\lg n!$ compares, contradicting **Proposition I** in Section 2.2 of [SW11]. □

Exercise 25. *Perfect balance.* Write a program that inserts a set of keys into an initially empty BST such that the tree produced is equivalent to binary search, in the sense that the sequence of compares done in the search for any key in the BST is the same as the sequence of compares used by binary search for the same key.

Solution. During binary search, the middle key of the current range is compared at each step against the search key. See `com.segarciat.algs4.ch3.sec2.ex25.PerfectBalance`.

Exercise 27. *Memory usage.* Compare the memory usage of `BST` with the memory usage of `BinarySearchST` and `Sequential SearchST` for n key-value pairs, under the assumptions described in Section 1.4 (see **Exercise 3.1.21**). Do not count memory for the keys and values themselves, but do count references to them. Then draw a diagram that depicts the precise memory usage of a BST with `String` keys and `Integer` values (such as the ones built by `FrequencyCounter`), and then estimate the memory usage (in bytes) for the BST built when `FrequencyCounter` uses `BST` for *Tale of Two cities*.

Solution. In **Exercise 3.1.21**, I determined that the memory usage for `SequentialSearchST` for n key-value pairs would be $32 + 48n$ bytes plus the cost of the keys and values themselves. I also determined that the cost of `BinarySearchST` is between $72 + 16n$ and $72 + 64n$ bytes, plus the cost for the keys and values themselves.

For `BST`, first we consider the cost of each `Node`. This consists of 16 bytes of overhead, 8 bytes overhead for the reference to the enclosing class, 8 bytes for the reference to the `key` field, 8 bytes for the reference to the `val` field, `16` bytes total to account for the references to `left` and `right`, `4` bytes for the reference to `n`, and 4 bytes of padding. The total memory cost for a `Node` is thus 64 bytes. The `BST` class has 16 bytes of overhead, 8 bytes for the reference to `root`, and $64n$ bytes for all n nodes (key-value pairs), making the memory cost $24 + 64n$ bytes plus the cost of the n keys and values themselves.

An `Integer` takes up `24` bytes, so n integer values amounts to $24n$ bytes. The cost of a `String` according to Section 1.4 on page 202 of [SW11] is $56 + 2m$ where m is the total number of characters in the `String` object. If M is the length of the largest `String` key, then the cost is bounded by

$$24 + 64n + 24n + n \cdot (56 + 2M) = 24 + 144n + 2Mn$$

The longest word in *Tale of Two Cities* is `farmergeneralhowsoever`, which has 22 characters, and the shortest "word" has 1 character. There are $10,674$ distinct words in the text. Therefore, letting $n = 10674$ and $M = 22$, we see that the memory cost T is bounded by

$$24 + 144 \cdot 10674 + 2 \cdot 1 \cdot 10674 \leq T \leq 24 + 144 \cdot 10674 + 2 \cdot 22 \cdot 10674$$
$$1558428 \leq T \leq 2006736$$

This means it takes up at least 1.486 MiB and at most 1.913 MiB.

Exercise 30. *BST reconstruction.* Given the preorder (or postorder) traversal of a BST (not including null nodes), design an algorithm to reconstruct the BST.

Solution. For preorder, we add the keys in the same order; for postorder, we can add the keys to a stack and traverse the postorder in reverse.

See `com.segarciat.algs4.ch3.sec2.ex30.BSTReconstruction`.

Exercise 32. *Subtree count check.* Write a recursive method that takes a `Node` as argument and returns `true` if the subtree count field `n` is consistent in the data structure rooted at that node, `false` otherwise.

Solution. See `com.segarciat.algs4.ch3.sec2.ex32.BST`.

Exercise 33. *Select/rank check.* Write a method that checks, for all `i` from `0` to `size() - 1`, whether `i` is equal to `rank(select(i))` and, for all keys in the BST, whether `key` is equal to `select(rank(key))`.

Solution. See `com.segarciat.algs4.ch3.sec2.ex33.SelectRankCheck`.

Exercise 37. *Level-order traversal.* Write a method `printLevel()` that takes a `Node` as argument and prints the keys in the subtree rooted at that node in level order (in order of their distance from the root, with nodes on each level from left to right). *Hint*: Use a `Queue`.

Solution. See `com.segarciat.algs4.ch3.sec2.ex37.BST`.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.