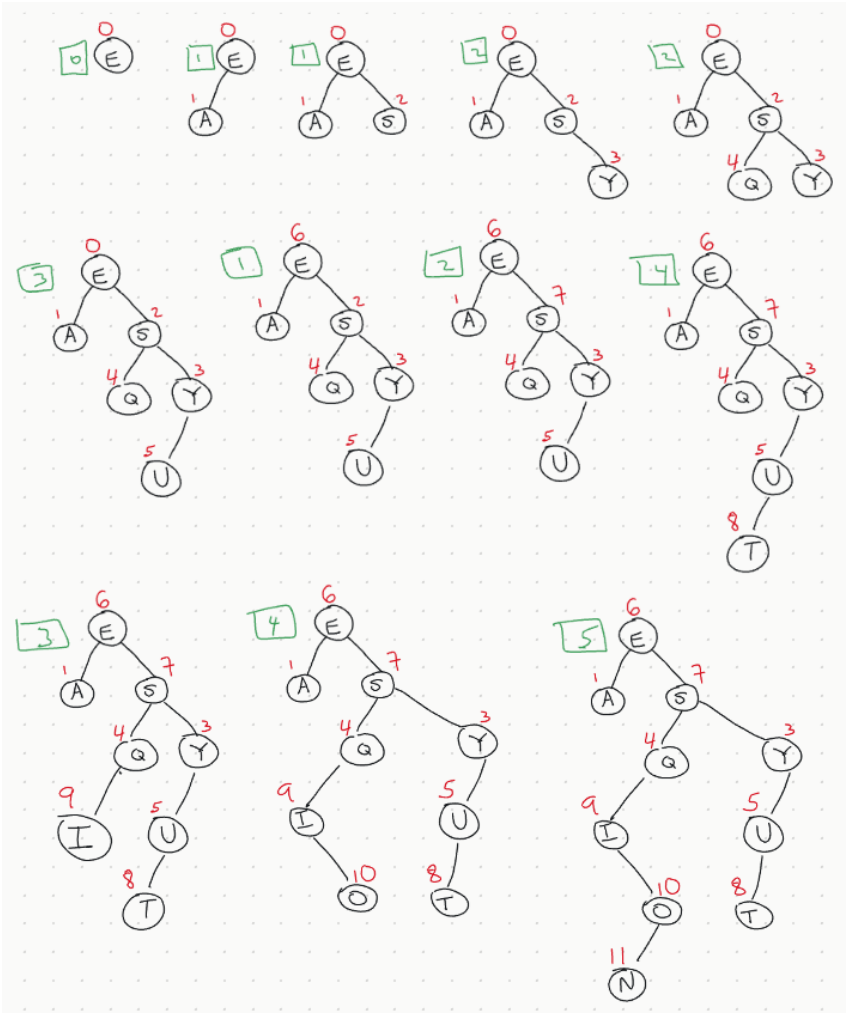

Figure 1: Exercise 1: Sequence of trees formed when inserting the keys `E A S Y Q U E S T I O N` into a binary search tree.

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.

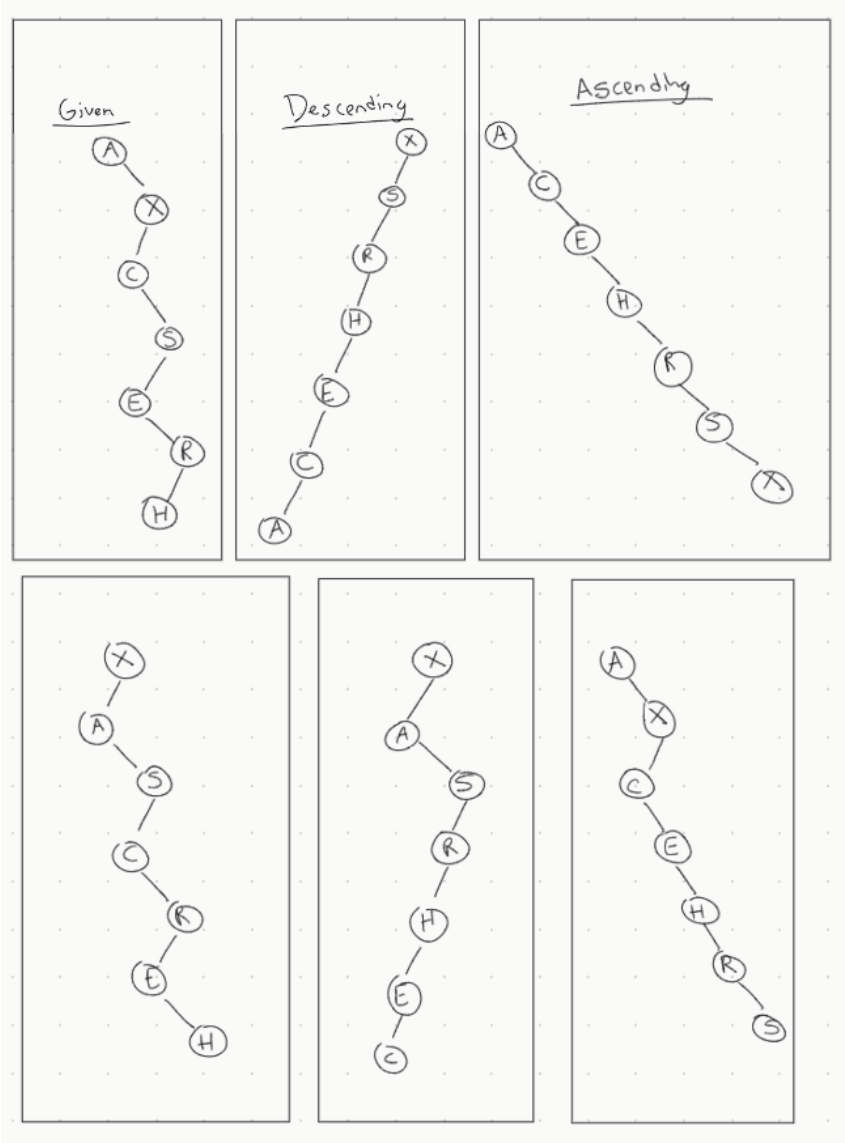

Figure 2: Exercise 2: Worst-case binary search trees created by inserting the keys `A X C S E R H` into an initially empty tree in a particular order

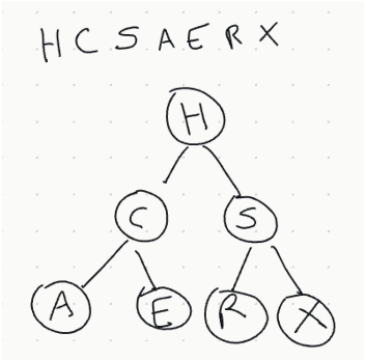

Figure 3: Exercise 3: Best-case binary search trees created by inserting the keys `A X C S E R H` into an initially empty tree in a particular order

(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`.

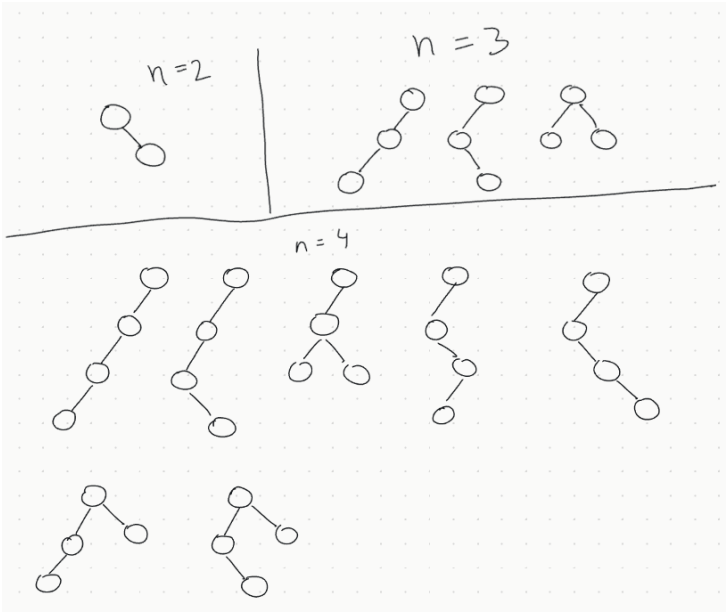

Figure 4: Exercise 9: All binary-search tree shapes for $n = 2, 3, 4$ vertices.

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**)

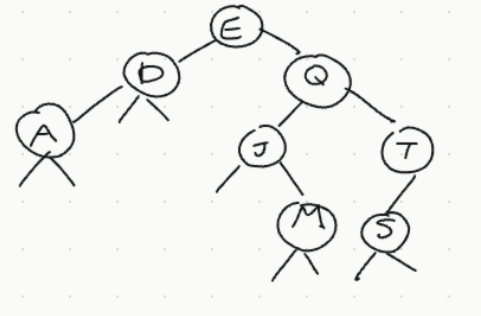

Figure 5: Exercise 15: Binary Search Tree

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"`

References

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