CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 2: Instructions

Harvard SEAS - Fall 2021 *Sept. 21, 2021*

These instructions are mostly the same as those for our previous Active Learning exercise on September 7, except for the following: (a) please sit next to and pair with someone different than you did on 9/7, and (b) we are now explicitly allowing the Sender to draw some a diagram or two (only) to get the explanation started. The bulk of the explanation should still be oral, with the Receiver figuring out how to translate the algorithm and proof steps into the diagrams. We also recommend reading through the reflection quotes in the Sept. 14 lecture notes for inspiration on how to be most effective in this exercise.

One of the skills we want you to develop in CS120 is to be able to manage the complexity of mathematical proofs by thinking about them at different levels of abstraction, ranging from a high-level intuitive proof outline to being able to work out the low-level formal details. Importantly, these are not two separate skills—rather you need to learn to be able to *move between* these different levels of thinking about proofs, e.g., extracting the high-level intuition from a formal proof, and turning a high-level proof strategy into a detailed formal proof. The reason this skill is particularly valuable for you as computer scientists is that software and hardware systems can be extremely complex to design and understand; to manage that complexity, we need to be able to work at a high level, but to complete an implementation or analysis, we also need to be able to fill in the technical details.

To develop this skill, we will do in-class active learning exercises where half of you (the "Senders") are tasked with reading and understanding a formal proof, and then explaining it a high level to a classmate (a "Receiver"), who is then tasked with writing it down and filling in the formal details. In addition, these exercise will also have the benefit of reinforcing important concepts related to the content of the course.

The structure of the exercise is as follows. *Before class,* both the Sender and Receiver should study the reading assigned to them. The Receiver reading is short and only contains the theorem statement and motivation. The Sender reading contains a proof a proof of the theorem and should be studied more carefully. If you are a Sender, you should try to extract both the main ideas (low-to-high level translation) and the technical details. You may create some bullet notes or a high-level summary for yourself to help you in your presentation during class. Avoid the temptation to write a fully detailed script or to memorize the proof; instead try to internalize the ideas so that you can reproduce the proof based on an intuitive understanding.

In class, the exercise will proceed as follows:

1. Senders and Receivers form pairs (or triples if we end up with imbalanced attendance), introduce themselves to each other, and find an open desk. You should pair with someone that you have not paired with before.

2. The Sender describe the theorem and proof to the Receiver through an oral, interactive dialogue. The Sender should avoid writing, but may draw a diagram or two to get the explanation started (but further manipulations of the diagram should be left to the Receiver). The Receiver should be capturing their understanding by writing pictures and notes. The Receiver should ask questions of the Sender during the interaction. After the high-level

understanding is transmitted, the dialogue should continue on to filling in as many formal details as possible.

3. Senders and Receivers can raise their hands if they are jointly stuck or have clarification questions, and a member of the teaching staff will come to help.

4. After the exercise is complete (or at the end of class), we will have you fill out a reflection survey, which will provide valuable feedback for us and will give you material for your Participation Portfolios.

CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 2: Reading for Receivers

Harvard SEAS - Fall 2021 *Sept. 21, 2021*

The goals of this exercise are:

- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them
- to practice reasoning about updates to dynamic data structures and binary search trees in particular

In the previous class (Thursday 9/16), we have seen that a variety of different operations (search, insert, min/max, predecessor/successor) can be performed on a binary search tree (BST) in time $O(h)$, where h is the height of the tree. Here you will see how *deletions* can be done in time $O(h)$:

Theorem 0.1. *Given a binary search tree T of height h and a key K stored in the tree, we can delete a matching item-key pair (I, K) from T in time $O(h)$. Deletion means that we produce a new binary search tree that contains all of the item-key pairs in T except for one less occurrence of key K.*

To prepare for the exercise, we recommend reviewing the lecture notes from Thursday 9/16 to make sure you are comfortable with BSTs and the simpler operations on them.

CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 2: Reading for Senders

Harvard SEAS - Fall 2021 *Sept. 21, 2021*

The goals of this exercise are:

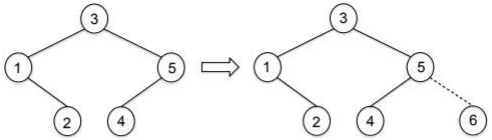
- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them
- to practice reasoning about updates to dynamic data structures and binary search trees in particular

In the previous class, we have seen that a variety of different operations (search, insert, min/max, predecessor/successor) can be performed on a binary search tree (BST) in time $O(h)$, where h is the height of the tree. Here you will see how *deletions* can be done in time $O(h)$:

Theorem 0.1. *Given a binary search tree T of height h and a key K stored in the tree, we can delete a matching item-key pair (I, K) from T in time $O(h)$. Deletion means that we produce a new binary search tree that contains all of the item-key pairs in T except for one less occurrence of key K.*

For the proof, we will have you read Roughgarden II, Section 11.3.8 (attached), which has a particularly good description of the deletion operation. It's important to note a few small differences between Roughgarden's treatment of BSTs and ours:

- Roughgarden assumes that all of the keys are distinct; feel free to assume the same during this exercise.
- Roughgarden's Predecessor query is a bit different than ours — it finds a (strict) predecessor of a key already in the tree, rather than a (weak) predecessor of a query key q. However, as you are reading, think about how the use of Roughgarden's Predecessor in the Delete operation can be replaced with a simple Max operation like we saw in class.



INSERT

1. Start at the root node.
2. Repeatedly traverse left and right child pointers, as appropriate (left if k is smaller than the current node's key, right if it's larger), until a null pointer is encountered.
3. Replace the null pointer with one to the new object. Set the new node's parent pointer to its parent, and its child pointers to null.

The operation preserves the search tree property because it places the new object where it should have been.[12] The running time is the same as for SEARCH, which is $O(height)$.

11.3.8 Implementing DELETE in $O(height)$ Time

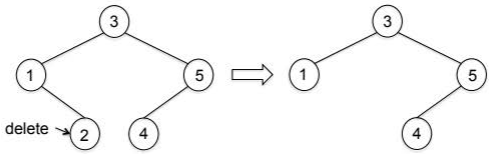
In most data structures, the DELETE operation is the toughest one to get right. Search trees are no exception.

> DELETE: for a key k, delete the object with key k from the search tree, or report that no such object exists.

The main challenge is to repair a tree after a node removal so that the search tree property is restored.

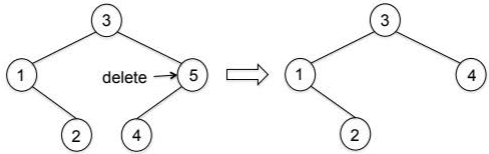
[12] More formally, let x denote the newly inserted object and consider an existing object y. If x is not a member of the subtree rooted at y, then it cannot interfere with the search tree property at y. If it is a member of the subtree rooted at y, then y was one of the nodes visited during the unsuccessful search for x. The keys of x and y were explicitly compared in this search, with x placed in y's left subtree if and only if its key is smaller than y's.

The first step is to invoke SEARCH to locate the object x with key k. (If there is no such object, the DELETE operation halts and reports this fact.) There are three cases, depending on whether x has 0, 1, or 2 children. If x is a leaf, it can be deleted without harm. For example, if we delete the node with key 2 from our favorite search tree:



For every remaining node y, the nodes in y's subtrees are the same as before, except possibly with x removed; the search tree property continues to hold.

When x has one child y, we can splice it out. Deleting x leaves y without a parent and x's old parent z without one of its children. The obvious fix is to let y assume x's previous position (as z's child).[13] For example, if we delete the node with key 5 from our favorite search tree:



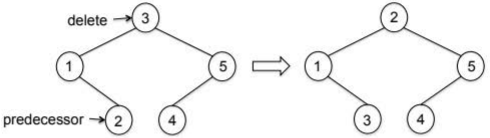
By the same reasoning as in the first case, the search tree property is preserved.

The hard case is when x has two children. Deleting x leaves *two* nodes without a parent, and it's not clear where to put them. In our running example, it's not obvious how to repair the tree after deleting its root.

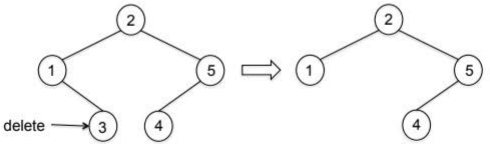
[13] Insert your favorite nerdy Shakespeare joke here...

The key trick is to reduce the hard case to one of the easy ones. First, use the PREDECESSOR operation to compute the predecessor y of x.[14] Because x has two children, its predecessor is the object in its (non-empty!) left subtree with the maximum key (see Section 11.3.5). Because the object with the maximum key is computed by following right child pointers as long as possible (see Section 11.3.4), y cannot have a right child; it might or might not have a left child.

Here's a crazy idea: *Swap* x and y! In our running example, with the root node acting as x:



This crazy idea looks like a bad one, as we've now violated the search tree property (with the node with key 3 in the left subtree of the node with key 2). But every violation of the search tree property involves the node x, which we're going to delete anyway.[15] Because x now occupies y's previous position, it no longer has a right child. Deleting x from its new position falls into one of the two easy cases: We delete it if it also has no left child, and splice it out if it does have a left child. Either way, with x out of the picture, the search tree property is restored. Back to our running example:



[14] The successor also works fine, if you prefer.

[15] For every node z other than y, the only possible new node in z's subtree is x. Meanwhile, y, as x's immediate predecessor in the sorted ordering of all keys, has a key larger than those of all the other nodes in x's old left subtree and less than those of all the nodes in x's old right subtree. Thus, the search tree condition holds for y in its new position, except with respect to x.

DELETE

1. Use SEARCH to locate the object x with key k. (If no such object exists, halt.)
2. If x has no children, delete x by setting the appropriate child pointer of x's parent to null. (If x was the root, the new tree is empty.)
3. If x has one child, splice x out by rewiring the appropriate child pointer of x's parent to x's child, and the parent pointer of x's child to x's parent. (If x was the root, its child becomes the new root.)
4. Otherwise, swap x with the object in its left subtree that has the biggest key, and delete x from its new position (where it has at most one child).

The operation performs a constant amount of work in addition to one SEARCH and one PREDECESSOR operation, so it runs in $O(height)$ time.

11.3.9 Augmented Search Trees for SELECT

Finally, the SELECT operation:

> SELECT: given a number i, between 1 and the number of objects, return a pointer to the object in the data structure with the ith-smallest key.

To get SELECT to run quickly, we'll *augment* the search tree by having each node keep track of information *about the structure of the tree itself*, and not just about an object.[16] Search trees can be augmented in many ways; here, we'll store at each node x an integer $size(x)$ indicating the number of nodes in the subtree rooted at x (including x itself). In our running example

[16] This idea can also be used to implement the RANK operation in $O(height)$ time (see Problem 11.4).