CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Sender–Receiver Exercise 2: Reading for Senders

Harvard SEAS - Fall 2024 *2024-09-19*

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

1 The Result

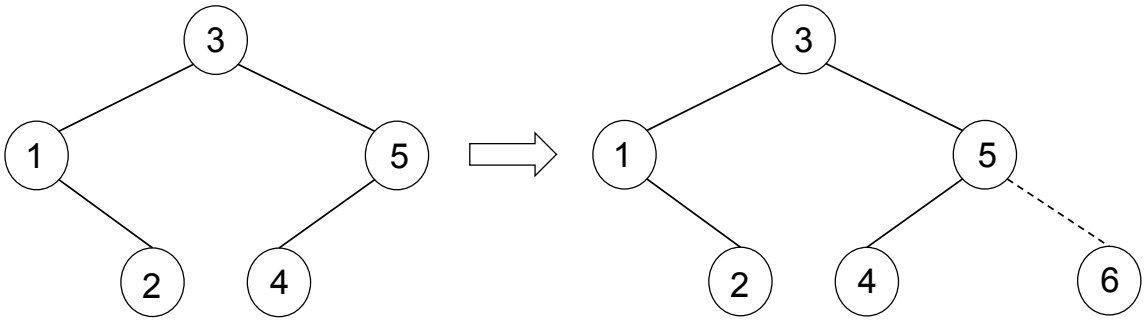
In the previous class (Tuesday 09-17), we saw that insert operations can be performed on a binary search tree (BST) in time $O(h)$, where h refers to the height of the tree. As an in-class exercise, some of you saw that a variety of different operations (search, min/max, next-smaller/next-larger) can also be performed in time $O(h)$; pseudocode for those operations is in the detailed lecture notes. Here you will see how *deletions* can be performed in time $O(h)$.

Theorem 1.1. *Given a binary search tree T of height h, and a key K stored in the tree, we can delete a matching key-value pair (K, V) from T in time $O(h)$. Deletion means that we produce a new binary search tree that contains all of the key-value pairs in T except for one less occurrence of a pair with key K.*

2 The Proof

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 our next-smaller query — it finds what's next-smaller than a key already in the tree, rather than what's next-smaller than an arbitrary query key K that's not necessarily in the tree.



What if there is already an object with key k in the tree? If you want to avoid duplicate keys, the insertion can be ignored. Otherwise, the search follows the left child of the existing object with key k, pushing onward until a null pointer is encountered.

INSERT

1. Start at the root node.
2. Repeatedly traverse left and right child pointers, as appropriate (left if k is at most the current node's key, right if it's bigger), 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.[13] The running time is the same as for SEARCH, which is $O(height)$.

11.3.8 Implementing DELETE in $O(height)$ Time

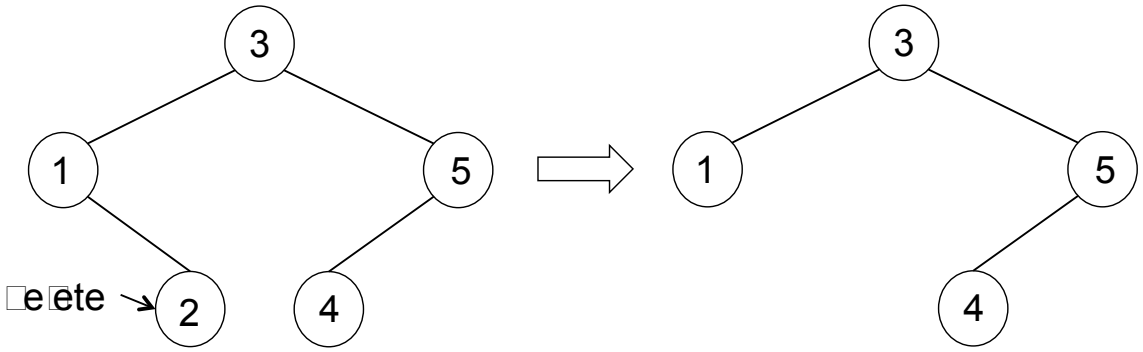
In most data structures, the DELETE operation is the toughest one to get right. Search trees are no exception.

> DELETE: for a key k, delete an object with key k from the search tree, if one exists.

[13] 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 no larger than y's.

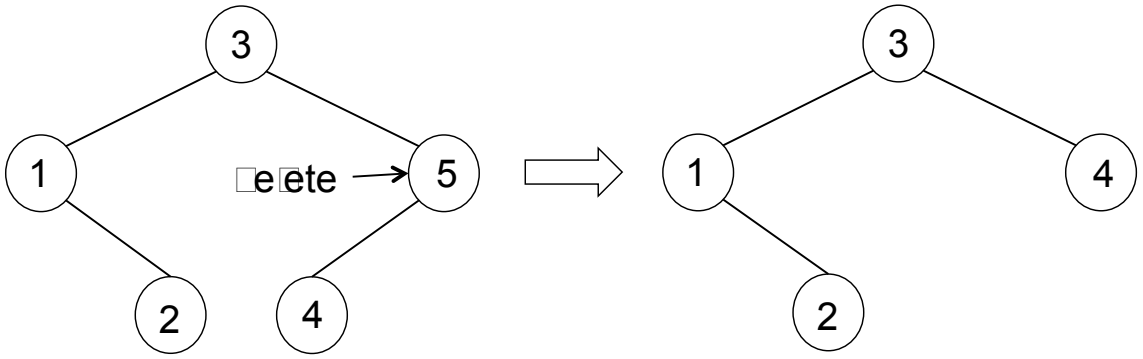
The main challenge is to repair a tree after a node removal so that the search tree property is restored.

The first step is to invoke SEARCH to locate an object x with key k. (If there is no such object, DELETE has nothing to do.) 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).[14] For example, if we delete the node with key 5 from our favorite search tree:



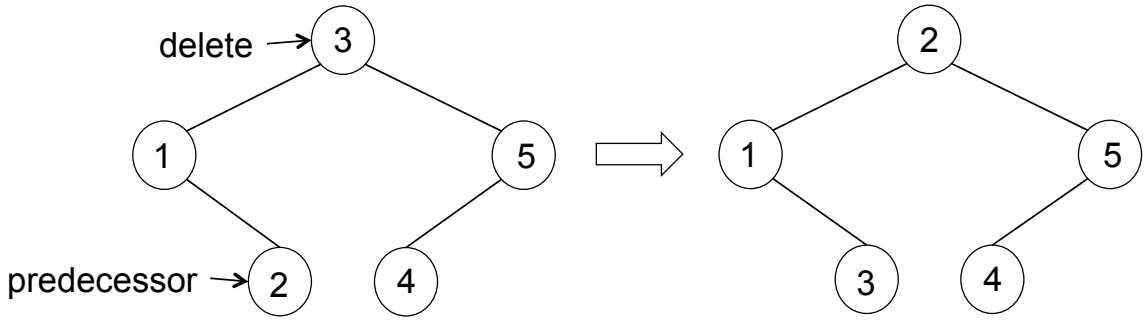
By the same reasoning as in the first case, the search property is preserved.

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.

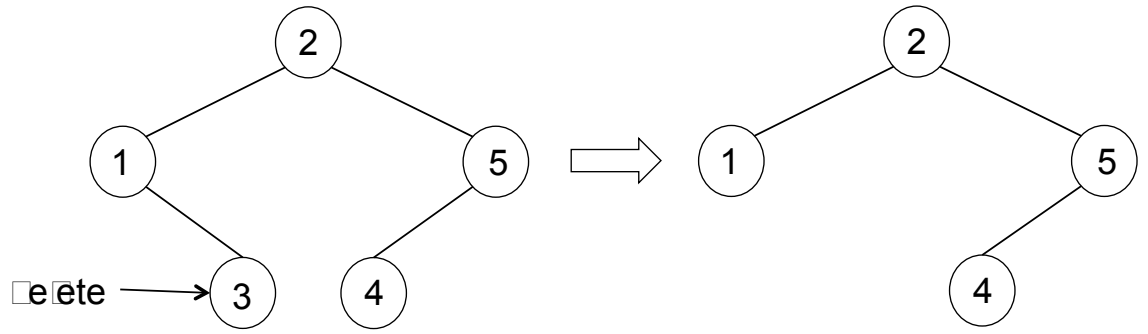
[14] Insert your favorite nerdy Shakespeare joke here...

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.[15] 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). Since the maximum 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.[16] 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:



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

[16] 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 in x's old left subtree and greater than those 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 an 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.[17] 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

[17] This idea can also be used to implement the RANK operation in $O(height)$ time (as you should check).