CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Sender–Receiver Exercise 2: Reading for Senders

Harvard SEAS - Fall 2025 *2025-09-18*

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

We have already established most of Theorem 4.8 in the textbook, showing that a variety of different operations can be performed on BSTs in time $O(h)$, including `insert` updates. Here we will complete the proof, by showing that `delete` updates can also be done in time $O(h)$.

Theorem .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.*

0.1 The Proof

In many data structures, deletion is the trickiest operation, because naively deleting creates "holes" in the encoded data which we need to patch up to maintain the required structure and invariants (such as the BST Property).

Let T be a BST and suppose we receive a query to delete a key K. The first step, of course, is to follow Search(T,K)(Algorithm **??**) to see if K is actually in the tree. If so, we will need to know not just a corresponding value V, but actually obtain a vertex v of T that has $K_v = K$, as well as the parent p of v (if v is not the root). (So we won't use Search(T,K) as a black-box, but rather use a modification of its code.)

Now we break into cases depending on how many children v has.

Case 0: v has 0 children. In this case, we can just remove v and delete p's pointer to v. (If v is the root, we return the empty tree.) See Figure 1 for an illustration. Note that this maintains the BST property, since we haven't changed the ancestor-descendant relations between any other pair of vertices in the tree. Also, the new tree holds exactly the same multiset of key-value pairs as the original tree, minus one copy of a pair with key K (namely (K_v, V_v))).

Case 1: v has 1 child. Without loss of generality, say that v has a left-child v_L, but not a right-child. Now when we delete v, we can simply slide v_L up to v's place, by replacing p's pointer to v with a pointer to v_L. (If v was the root of T, then v_L becomes the new root.) See Figure 2 for an illustration. The correctness of this case follows by the same arguments as in Case 0.

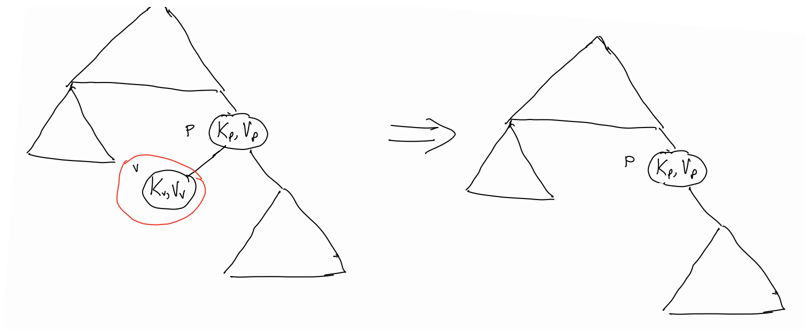

Figure 1: Deleting vertex v when it has 0 children. Objects to be deleted are circled in red.

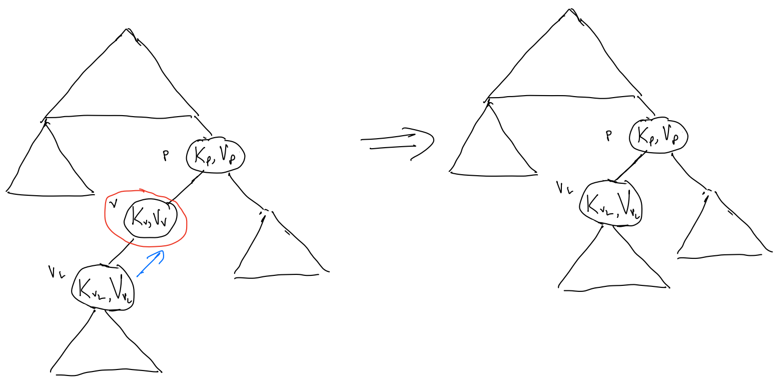

Figure 2: Deleting vertex v when it has 1 child. Objects to be deleted are circled in red, and objects to be moved are indicated with blue arrows.

Case 2: v has 2 children. This is the tricky case, because we cannot slide either child up into v's place, because neither child necessarily has a free child-pointer that can be used to point to v's other child. Instead we will move the key-value pair from a different vertex w, from potentially far away, into vertex v (overwriting (K_v, V_v)), and then delete vertex w from the tree. What properties do we want from w?

1. K_w should be at least as large as each of the keys in the subtree rooted at v's left-child, v_L, so that the resulting tree satisfies the BST Property.
2. K_w should be no larger than each of the keys in the subtree rooted at v's right-child, v_R, so that the resulting tree satisfies the BST Property.
3. w should have 0 or 1 children, so that we can delete it easily, as in Cases 1 and 2 above.

How can we find such a vertex w? Applying the `Maximum` operation to the subtree rooted at v_L! Then Item 1 will hold by the definition of `max`. Item 2 will hold by the BST property of the original tree T: since w is in the subtree under v_L, $K_w \leq K_v$, and K_v is no larger than each of the keys in the subtree under v_R. Item 3 holds by inspection of pseudocode for the `Maximum` operation: it keeps traversing right in the tree, and stops when it reaches a vertex w such that the right-subtree under w is empty. (See Algorithm **??** for case of the `Minimum` operation, which stops at a vertex whose left-subtree is empty.)

By deleting w and overwriting (K_v, V_v) with (K_w, V_w), the multiset of key-value pairs stored in the tree is indeed modified only by removing one copy of a key-value pair with key K, as desired. We have chosen w above to ensure that the resulting tree satisfies the BST Property. Thus, we have correctness in this case as well. See Figure 3 for an illustration.

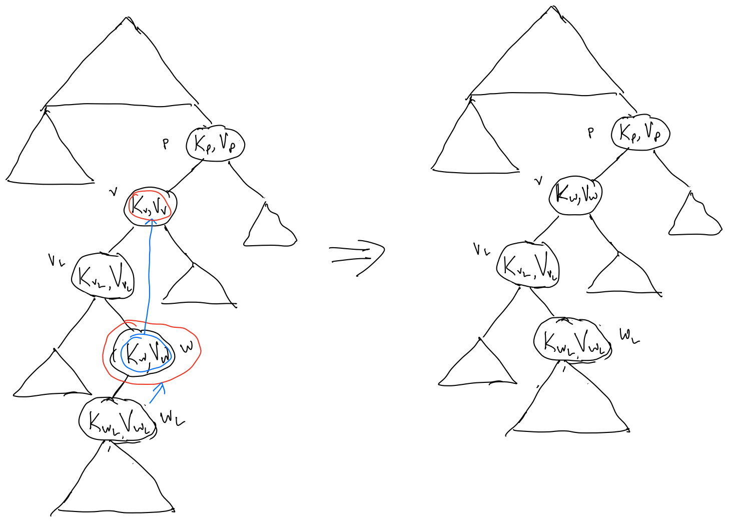

Figure 3: Deleting vertex v when it has 2 children and the maximum vertex in its left subtree w has 1 child. Objects to be deleted are circled in red, and objects to be moved are indicated with blue arrows.

Putting it all together, we obtain Algorithm .1:

```
   Delete($T$,$K$):
0  Run a modified version of Search($T$,$K$) to obtain a vertex $v$ in $T$ such that $K_v = K$ and
     $v$'s parent $p$ (if $v$ is not the root);
1  if $v = \perp$ then return $\perp$;
2  if $v$ has 0 children then
3  |   Delete $v$ and $p$'s pointer to $v$;
4  else if $v$ has 1 child then
5  |   Delete $v$ and replace $p$'s pointer to $v$ with a pointer to the child;
6  if $v$ has 2 children then
7  |   Let $w$ be the right-most vertex under $v_L$ (computed as in the Maximum operation), and
   |     let $q$ be its parent;
8  |   Let $(K_v, V_v) = (K_w, V_w)$;
9  |   Delete $w$ as in the cases above, according to whether or not it has 0 or 1 children (either
   |     deleting $q$'s pointer to $w$ or replacing it with a pointer to $w$'s only child, respectively);
10 return the tree $T'$ modified as above;
```

Algorithm .1: `Delete()`

Finally, we need to verify that the runtime is $O(h)$. Algorithm .1 involves one execution of a (slight modification of) `Search(`T`,`K`)` and one execution of `Maximum` on a subtree of T, plus $O(1)$ additional operations. Thus the runtime is $O(h) + O(h) + O(1) = O(h)$, as desired.