CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Lecture 5: Binary Search Trees

Harvard SEAS - Fall 2023 *2023-09-19*

1 Announcements

2 Binary Search Trees

2.1 Motivation

Our consideration of dynamic interval scheduling in the previous lecture led us to seek a dynamic data structure to solve the following problem:

Updates :

- Insert(K,V) for $K \in \mathbb{R}$: add one copy of (K, V) to the multiset S of key-value pairs being stored. (S is initially empty.)
- Delete(K) for $K \in \mathbb{R}$: delete one key-value pair of the form (K, V) from the multiset S (if there are any remaining).

Queries :

- Search(K) for $K \in \mathbb{R}$: output $(K', V') \in S$ such that $K' = K$.
- Next-smaller(K) for $K \in \mathbb{R}$: output $(K', V') \in S$ such that $K' = \max\{K'' : \exists V'' \ (K'', V'') \in S, \ K'' < K\}$.

Data-Structure Problem Dynamic Predecessors

From the last lecture and the lecture notes, we can infer an implementation of this data structure in which the times for insertions, deletions, search queries, and next-smaller queries are: $O(n)$, $O(n)$, $O(\log n)$, and $O(\log n)$, respectively. This data structure simply maintains a sorted array (we will discuss it briefly in this lecture). In this lecture, we'll look at another implementation, *binary trees*, in which some of those operations are faster.

2.2 Binary Search Tree Definition and Intro

Definition 2.1. A *binary search tree (BST)* is a recursive data structure. Every nonempty BST has a root vertex r, and every vertex v has:

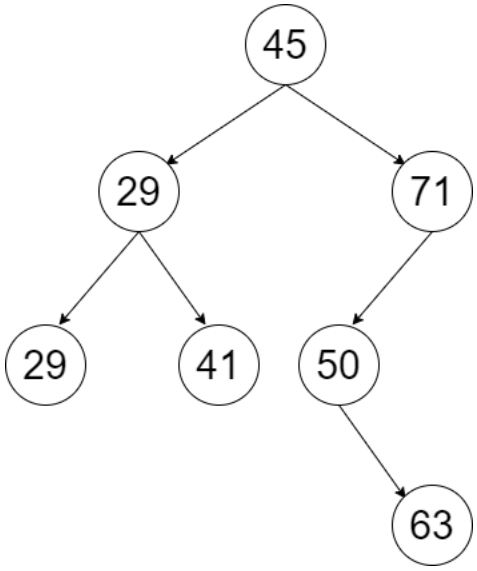
- a key K_v
- a value V_v
- a pointer to a left child v_L, which is another binary tree (possibly `None`, the empty BST)
- a pointer to a right child v_R, which is another binary tree (possibly `None`, the empty BST)

Crucially, we also require that the keys satisfy the *BST Property*:

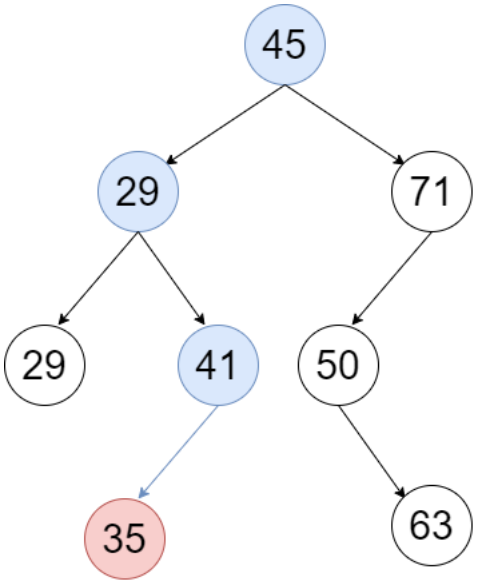
> If v has a left-child v_L, then the keys of v_L and all its descendants are no larger than K_v, and similarly, if v has a right-child, then the keys of v_R and all of its descendants are no smaller than K_v.

Note that the empty set satisfies all the properties above, and is a BST.

Example:



If we need to insert 35, then we color the vertices that we compare 35 to:



We remark that the description of binary search trees in Roughgarden and CLRS differs slightly from ours. First, instead of considering the data associated with a key to be a value stored at the same vertex, they treat the vertex itself as the data associated with a key. So, for example, to replace one data element with another, in our formulation, we could just rewrite the key and value attributes of the corresponding vertex, whereas the texts would require removing the vertex from the tree and rewiring pointers to the new vertex.

Relatedly, the texts (and our pset 0) also include a parent pointer at every vertex. It's not necessary for anything we are going to do today, but it can be useful in implementations, as you may have discovered in pset 0 (so feel free to include it if you prefer).

It is also worth noting that in practice, trees with fanout (the number of children, aka maximum outdegree) greater than 2 are often used for greater efficiency in terms of memory accesses. Specifically, in large data systems, the fanout is often chosen so that the size of each vertex is a multiple of the size of a "page" in memory or on disk.

Theorem 2.2. *Given a binary search tree of height (equivalently, depth) h, all of the dynamic predecessors operations (queries and updates) can be performed in time $O(h)$.*

Proof.

- **Insertions:**

Runtime: The runtime of this operation is $O(h)$ for the same reason as in pset 0: we walk down the tree at most once, doing $O(1)$ work at each step.

Proof of Correctness:

- **Search:**

- **Minimum (and Maximum):**

- **Next-smaller:**

- **Deletions:** next SRE

□

3 Balanced Binary Search Trees

So far we have seen that a variety of operations (search, insertion, deletion, min, max, next-smaller, and next-larger) can be performed on Binary Search trees in time $O(h)$, where h is the height of the tree. Thus, we are now motivated to figure out how we can ensure that our binary search trees remain shallow (e.g. of height $O(\log n)$ where n is the number of items currently in the dataset). While doing so, we need to be sure that we retain the binary-search-tree property (the ordering of keys between a vertex and its left-descendants and its right-descendants).

There are several different approaches for retaining balance in binary search trees. We'll focus on one, called AVL trees, after mathematicians named Adelson-Velsky and Landis.

Definition 3.1 (AVL Trees)**.** An *AVL Tree* is a binary search tree in which:

-
-

Lemma 3.2. *Every AVL Tree with n vertices has height (=depth) at most $2\log_2 n$.*

Proof.

□

So we can apply all of the aforementioned operations on an AVL Tree in time $O(\log n)$. But an insertion or deletion can ruin the AVL property. To fix this, we need additional operations that allow us to move vertices around while preserving the binary search tree property. One important example is a *rotation*.

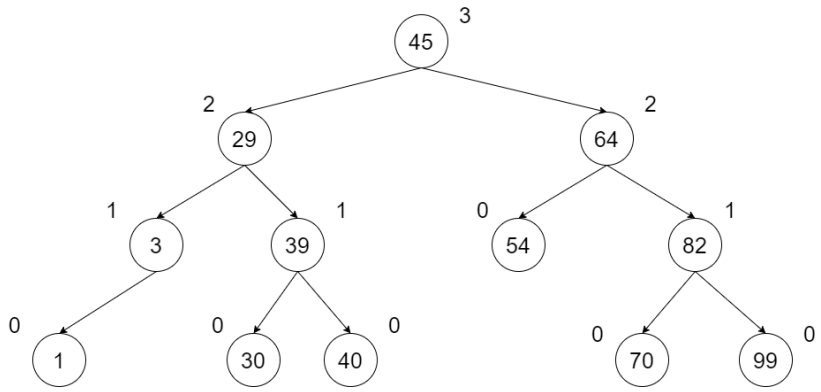
How much time does a rotation take?

Theorem 3.3. *We can insert a new key-value pair into an AVL Tree while preserving the AVL Tree property in time $O(\log n)$.*

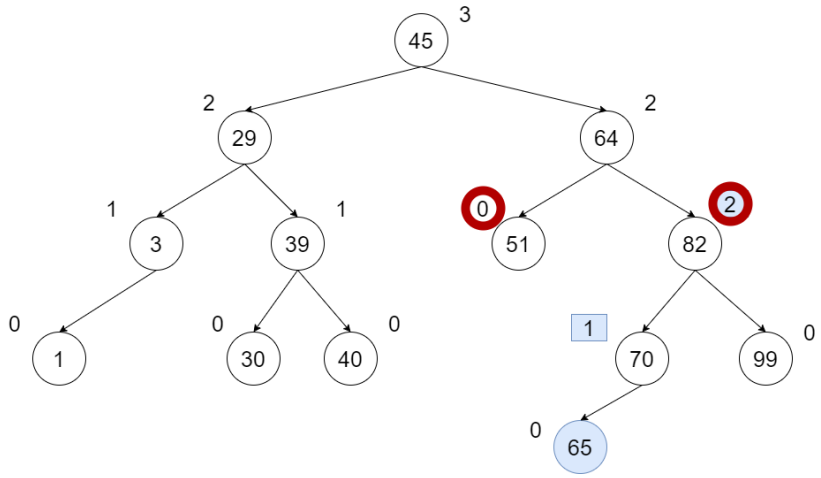
Proof. (sketch)

□

Let us see an example. The following is our original BST tree:



Now we want to insert $L = 65$. We update the height of the corresponding vertices (rectangles in blue), but when we reach the vertex 82 we see that we are now violating the AVL property: vertex 51 has height 0 while vertex 82 has height 2, so their difference is more than 1 (marked in red).



To fix this, we then use AVL rotations. Let y be the vertex of key 64, and z be the vertex of key 82, and to the above tree we apply the operation `Right.rotate(z)` followed by `Left.rotate(y)`.