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

Lecture 5: Binary Search Trees

Harvard SEAS - Fall 2022 *2022-09-15*

1 Announcements

- PS2 posted
- PS1 due: feedback at https://tinyurl.com/cs120ps1feedback
- Active Learning next Tuesday
- Reminder: OH not just for psets!
- Section/OH changes: see Ed

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

At the end of last lecture, we saw an implementation of this data structure in which the times for insertions, deletions, search queries, and next-smaller queries were: $O(n)$, $O(n)$, $O(\log n)$, and $O(\log n)$, respectively. Today 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:

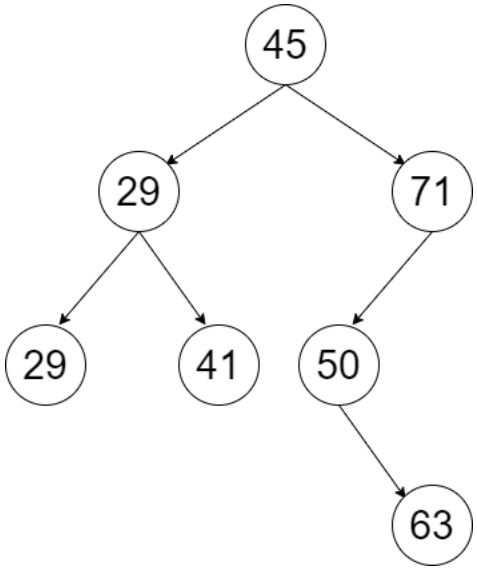
- a key K_v
- a value V_v
- a pointer to a left child v_L, which is another binary tree (possibly `None`, the empty BST)
- a pointer to a right child v_R, which is another binary tree (possibly `None`, the empty BST)

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

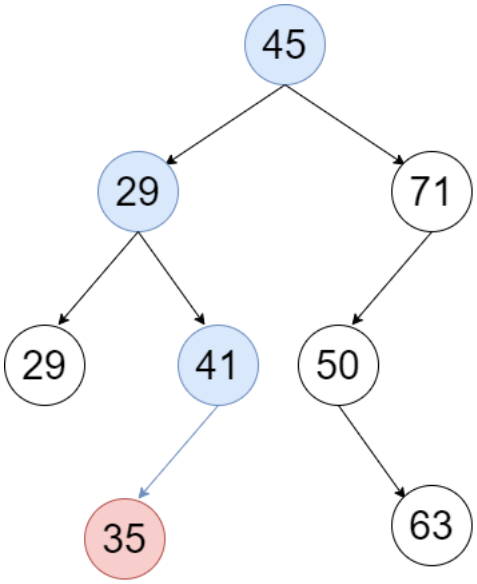
> If v has a left-child v_L, then the keys of v_L and all its descendants are no larger than K_v, and similarly, if v has a right-child, then the keys of v_R and all of its descendants are no smaller than K_v.

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:**

```
Insert(T,(K, V))
if T = ∅ then
    Let T = new BST with key K, value V, no children
    return T
Let v be the root of T.
if K ≤ K_v then
    T.left = Insert(T_L,(K, V))
    return T
else
    T.right = Insert(T_R,(K, V))
    return T
```

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:

The definition of a dynamic data structure problem doesn't actually impose any requirements on this operation: this is part of the preprocessing step, and the definition of correctness of a data structure implementation only requires that for all $x \in \mathcal{I}$, $u_0, u_1, \ldots, u_{m-1} \in \mathcal{U}$ and $q \in \mathcal{Q}$, if $f(x, u_0, u_1, \ldots, u_{m-1}) \neq \emptyset$, we have:

$$\text{EvalQ}(\text{EvalU}(\cdots \text{EvalU}(\text{EvalU}(\text{Preprocess}(x), u_0), u_1), \ldots, u_{m-1}), q) \in f(x, u_0, u_1, \ldots, u_{m-1}, q),$$

and $\perp$ otherwise—these are only conditions on the output of *queries*. Nevertheless, there's something we should prove about the insert operation, which will make proving correctness of queries easier: We should prove that the insert operation respects the BST property. Alternately, we can call this a requirement of the BST implementation, which the insert operation must satisfy to be a valid update.

Our proof of correctness uses induction on the height of the tree. Specifically, we prove the following statement by induction on h:

> (*) for every BST T of height at most h, `Insert`$(T, (K, V))$ returns a tree satisfying the BST property whose key-value pairs are $(K, V) \cup \bigcup_{(K_i, V_i) \in T} (K_i, V_i)$.

Our base case is $h = -1$, i.e. an empty tree. In this case, we return a tree containing exactly (K, V), where no vertex has any children, so the BST property is automatically satisfied.

For the induction step, assume that (*) is true for trees T of height at most h, and we'll prove it for trees of height at most $h+1$. Let T be an arbitrary tree of height $h+1 \geq 0$, so T has a root v. Given a search key K, we split into casework. If $K \leq K_v$, then T_L is a tree of height at most h, so by induction `Insert`$(T_L, (K, V))$ is a tree containing the key-value pairs in T_L plus (K, V) such that every vertex in it satisfies the BST property. Also, v itself satisfies the BST property: its right child hasn't changed, so it still satisfies the BST property. Every key K_L in its left child is either a key in T_L, which was no larger than K_v because T satisfied the BST property, or is K itself, which is at most K_v by the assumption of this case.

The other case, where $K > K_v$, is similar.

In-class Exercise : Write pseudocode to implement Search, and prove its correctness.

- **Search:**

```
1 Search(T,K)
2 Let v be the root of T.
3 if K_v = K then
4 |  return (K_v, V_v)
5 if K > K_v and T_R ≠ ∅ then
6 |  return Search(T_R,K)
7 if K ≤ K_v and T_L ≠ ∅ then
8 |  return Search(T_L,K)
9 return ⊥
```

Our proof of correctness uses induction on the height of the tree. Specifically, we prove the following statement by induction on h:

> (*) for every BST T of height at most h, if T contains a vertex with key K, then `Search`(T, K) returns a pair (K_w, V_w) such that $K_w = K$, and otherwise `Search`(T, K) returns $\perp$.

Our base case is height 0, i.e. a tree whose only vertex is the root v. In this case, T contains K if and only if $K = K_v$, which is exactly the condition under which `Search`(T, K) returns (K_v, V_v). Otherwise, `Search`(T, K) returns $\perp$, since $T_R = T_L = \emptyset$. So `Search`(T, K) is correct.

For the induction step, assume that (*) is true for trees T of height at most h, and we'll prove it for trees of height at most $h+1$. Given an arbitrary tree T of height $h+1$ and a search key K, we split into casework. Letting v be the root of T, if $K_v = K$ we are successful because we return (K_v, V_v). If $K_v < K$ then by the BST property K is in T if and only if K is in the right subtree T_R (which is of height at most h), so the recursive call `Search`(T_R, K) succeeds by induction, and the left call is similar.

For the runtime, note that we do a constant amount of work at each level and recurse at most h times on a tree of height h. To formally analyze this, let $T(h)$ be the maximum amount of work done by a search call on a tree of height at most h. Then for $h > 0$, we have

$$T(h) = T(h-1) + c,$$

for a constant c. Unrolling, we get

$$T(h) = T(h-1) + c = (T(h-2) + c) + c = \cdots = T(0) + h \cdot c = O(h).$$

- **Minimum (and Maximum):**
For Minimum, we always go left until there is no left child.

In-class Exercise : Write pseudocode to implement Next-smaller.

- **Next-smaller:**
  Given a query q, we again use a search type strategy. If $K_v = q$, we have found the next-smaller and can return (K_v, V_v). If $K_v > q$, return Next-smaller on the left subtree T_L (if it exists - if not, return $\perp$). Finally where $K_v < q$, call Next-smaller on the right subtree T_R (if it exists). If we receive $\perp$, return (K_v, V_v) and otherwise return the result of the call on T_R.

- **Deletions:** next class active learning exercise

□

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:

- Every vertex has an additional attribute containing its *height* in the tree (length of the longest path from the vertex to a leaf). ("data structure augmentation")
- Every pair of siblings in the tree have heights differing by at most 1 (where we define the height of the an empty subtree to be -1). ("height-balanced")

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

Proof.

Let $n(h) =$ min number of vertices in an AVL tree of height $\geq h$. Then, by the second AVL property, for $h \geq 2$, $n(h) \geq n(h-1) + n(h-2) + 1$. If we drop some constants, then

$$n(h) \geq n(h-1) + n(h-2) + 1 \geq 2n(h-2).$$

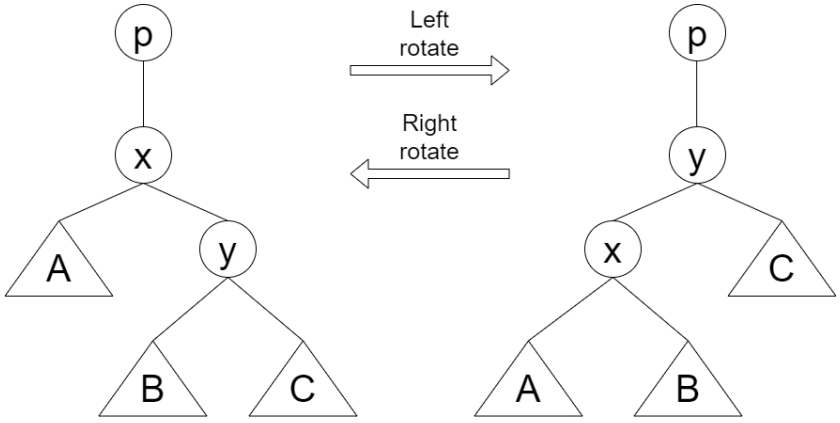
Now we are at a good place for unenrolling the expression:

$$\begin{aligned} n(h) &\geq 2n(h-2) \\ &\geq 4n(h-4) \\ &\geq 8n(h-6) \\ &\vdots \\ &\geq n^{n/2}n(0) \\ &= 2^{h/2}. \end{aligned}$$

(The above is for the case when h is even; a similar analysis can be done when h is odd.) □

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*.

We first investigate **rotations on BSTs.**



On the left: all keys in A are $\leq K_x$, all keys in B are $\in [K_x, K_y]$, and all keys in C are $\geq K_y \implies$ BST property is preserved in the right.

From left to right we have the operation `Left.rotate(x)`. From right to left we have the operation `Right.rotate(y)`.

How much time does a rotation take?

Rotations only take $O(1)$ time! One only needs to change a constant number of pointers – there is no traversing or other expensive operations.

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)

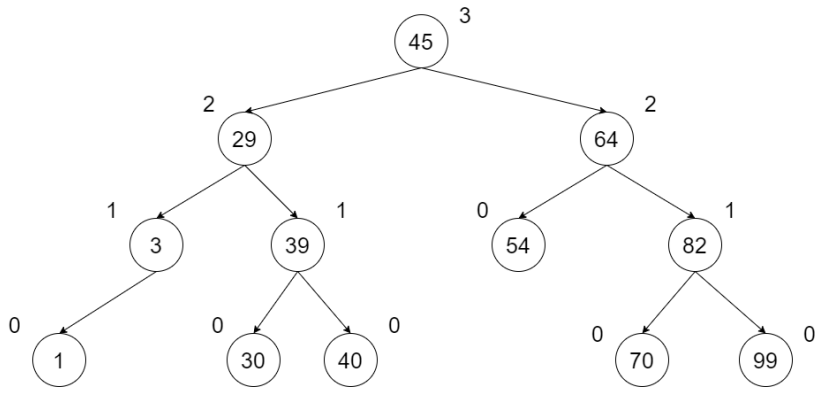
The algorithm runtime is as follows:

- Do an ordinary BST insert $\rightarrow$ adding a leaf L.
- Update heights working all the way from L to the root.
- When we see that we violate the AVL tree property we use rotations to fix it.

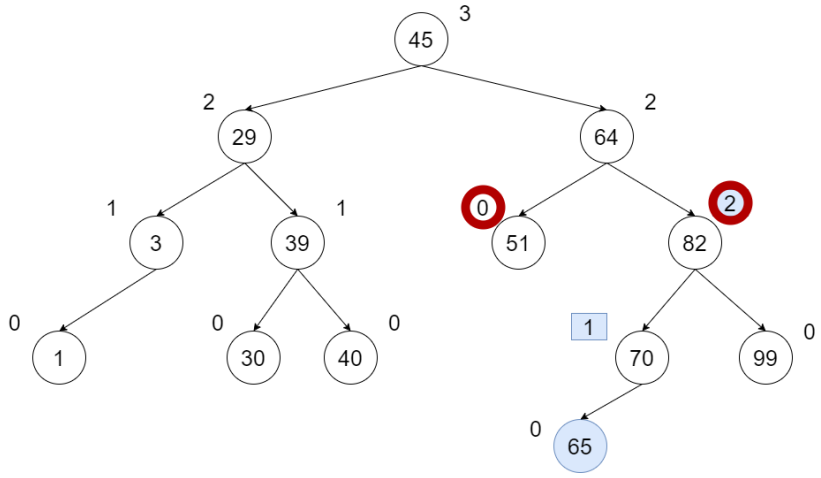
In total, this requires time $O(\text{height}) = O(\log n)$.

□

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)`.

In-class Exercise : Draw the tree above after `Right.rotate(z)` followed by `Left.rotate(y)`.

In-class Exercise : If we Insert 64.5, what rotations would fix the tree?

In-class Exercise (Harder) : What's the smallest sequence of insertions to the tree above (followed by rotations to fix the AVL property) for which the vertex of key 45 would no longer be the root?