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

Lecture 5: Dynamic Data Structures and Binary Search Trees

Harvard SEAS - Fall 2024 *2024-09-17*

1 Announcements

- PS2 to be posted tomorrow
- PS1 due tomorrow
- SRE this Thursday
- Anurag OH after class in SEC 3.323 (11:30-12:30).
- Gabe and Katherine OH on zoom, Wednesday.
- Late policy reminder: 8 late days total. 3 can be used on a single pset.

2 Recap/Loose-Ends from Lec 4

- Defs of a static data structure problem vs. a solution to such a problem. (cf. CS51: abstract data type vs. implementation.)
- Example: StaticPredecessors, which has many applications (enabling "range select" in relational databases, NoSQL data stores, ML systems).
- Solving StaticPredecessors using Sorted Arrays.
- Arrays vs. Linked Lists.

Q: How quickly can we implement other operations when using a sorted array to store a multiset S of key-value pairs?

1. `min`(): return the smallest key K in S.
2. `rank`(K): return the *number* of pairs $(K', V') \in S$ such that $K' < K$.
3. `insert`(K, V)
4. `delete`(K, V)

3 Dynamic Data Structures

As you might have been wondering for the Interval Scheduling Problem, it is often the case that we do not get all of our input data at once, but rather it comes in through incremental updates over time. This gives rise to *dynamic* data-structure problems.

Definition 3.1. A *dynamic data structure problem* is a quintuple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{U}, \mathcal{Q}, f)$ where:

- a set $\mathcal{I}$ of *inputs* (or *instances*)
- a set $\mathcal{U}$ of *updates*,
- a set $\mathcal{Q}$ of *queries*, and
- for every $x \in \mathcal{I}$, $u_0, u_1, \ldots, u_{n-1} \in \mathcal{U}$, and $q \in \mathcal{Q}$, a set $f(x, u_0, \ldots, u_{n-1}, q)$ of *valid answers*

Often we take $\mathcal{I} = \{\epsilon\}$, where ϵ is the empty input (e.g. a length 0 array), since the inputs can usually be constructed through a sequence of updates. For example:

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

A *multiset* is like a set but can contain more than one copy of an element. The multiset S appearing in the above definition is only used to define the functionality of the data structure, namely how queries should be answered. How this set is actually maintained is up to the particular solution.

To solve a dynamic data structure problem, we need to now come up with algorithms that also implement the updates.

The definition of what it means to implement a dynamic data structure with algorithms gets a bit cumbersome (and we don't expect you to remember it), but we include it here in the notes for completeness:

Definition 3.2. For a dynamic data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, \mathcal{U}, f)$, an *solution* is a triple of algorithms $(\text{Preprocess}, \text{EvalQ}, \text{EvalU})$ such that

Now, we would like both EvalU and EvalQ to be extremely fast. Dynamic data structures can also be conveniently implemented using `class`es in Python, similarly to as shown in Lecture 4, with the update operations also implemented as methods.

4 Binary Search Trees

4.1 Binary Search Tree Definition and Intro

Definition 4.1. A *binary search tree (BST)* is a recursive data structure. The *empty BST* has no vertices. Every nonempty BST has finitely many vertices, one of which is the root vertex r, and every vertex v has:

- a key K_v
- a value V_v
- a left subtree, which is either the empty BST or is rooted at a vertex denoted v_L, which we call the *left-child* of v.
- a right subtree, which is either the empty BST or is rooted at a vertex denoted v_R, which we call the *right-child* of v.

We require that the sets of vertices contained in the subtrees rooted at v_L and v_R are disjoint from each other and don't contain v. Furthemore, we require that every non-root vertex has exactly one parent, and the root vertex has no parents.

Crucially, we also require that the keys satisfy:

The BST Property:

The *multiset S* stored by a BST T is the multiset of key-value pairs occurring at vertices of T.

An example of a BST is:

Our original motivation to introduce the binary search tree data-structure was to reduce the cost of insertions and deletions in comparison to a sorted array. Inserting 35 in the BST example gives:

This motivates the definition of the tree's height, which captures the worst-case cost of insertion.

Definition 4.2 (Height of a tree)**.** The *height* h of a BST is

Theorem 4.3. *Given a binary search tree of height (equivalently, depth) h, all of the dynamic predecessors+successors operations (queries and updates), as well as min and max queries, can be performed in time $O(h)$.*

Since $O(h)$ only refers to sufficiently large h, note that we spend $O(1)$ time even when $h = -1$ or $h = 0$, even though $c \cdot h \leq 0$ in these cases.

Proof.

- **Insertions:**
  The algorithm for insertion is:

```
Insert(T,(K,V))
if $T = None$ then
    
Let v be the root of T.
if $K \leq K_v$ then
    
else
    
```

Runtime: The runtime of this operation is $O(h)$ for the same reason as in Pset 0. More formally,

Proof of Correctness:

- **Search:**

  The algorithm is as follows:

  Proof of correctness: Our proof of correctness is similar in spirit to the previous case of Insertion - it uses induction on the height of the tree.

  Runtime: The analysis is similar to that of Insertion and gives a runtime of $O(h)$.

- **Minimum (or Maximum):**
  The algorithm is to move to the left child up until one reaches a vertex with no left child, and then output the key at that vertex.

- **Next-smaller (or Next-larger):**
  The algorithm for next-smaller follows along related lines to search algorithm:

- **Deletions:** The algorithm for deletion is the subject of the upcoming SRE.

$\square$

5 Balanced Binary Search Trees

So far we have seen that a variety of operations 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 BST property.

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

-
-

Lemma 5.2. *Every AVL Tree with n vertices has height (=depth) at most $2\log_2 n$.*

Proof.

$\square$

Thus, we can apply all of the operations on an AVL Tree in time $O(\log n)$. But recall that we need to maintain a dynamic data structure, and insertion or deletion operation 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 *rotation*.

Runtime of the rotation:

Theorem 5.3. *We can insert a new key-value pair into an AVL Tree T while preserving the AVL Tree property in time $O(\log n)$.*

Proof. (sketch)

The algorithm is informally described as follows:

□

Example: We depict the algorithm with an example, where the values attributes are chosen to be empty. Consider the AVL tree in Figure 1.

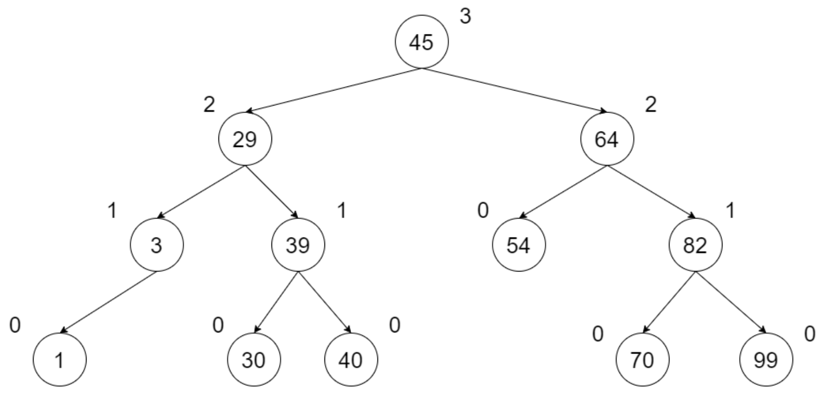

Figure 1: An example of AVL tree.

We insert 65 by creating a new leaf ℓ, resulting in the tree in Figure 2. We update the height of the corresponding vertices (rectangles in blue). Next we check the AVL property, which is violated since vertex 51 has height 0 while vertex 82 has height 2, so their difference is more than 1 (marked in red).

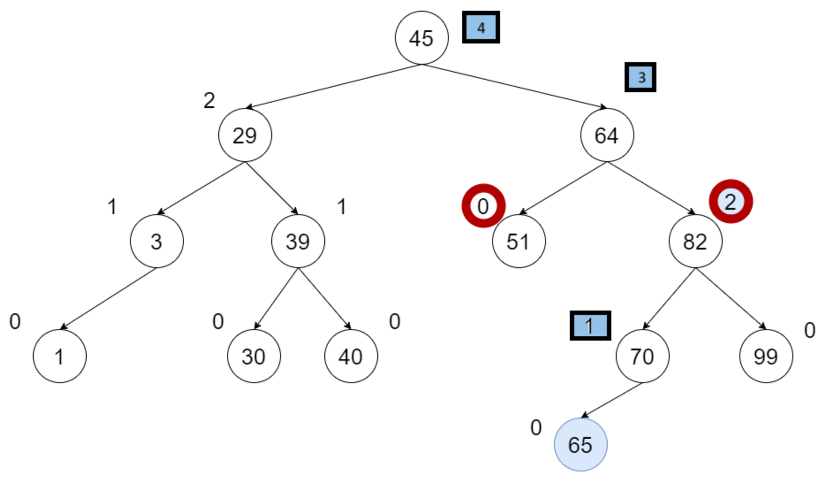

Figure 2: A new leaf is created with key 65. The height attributes are updated.

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