| **CS1200: Intro. to Algorithms and their Limitations** | Prof. Anurag Anshu |
|---|---|
| Lecture 5: Data Structures | |
| *Harvard SEAS - Spring 2026* | *2026-02-08* |

# 1 Announcements

- SRE 2 on Wednesday.

# 2 Recommended Reading

- Hesterberg–Vadhan, Chapter 3, Section 4.1–4.5
- CLRS Chapter 12.0–12.3
- Roughgarden II, Sec. 11.2–11.4
- CS50 Week 5: https://cs50.harvard.edu/x/2022/weeks/5/

# 3 Static data strutures

**Q:** Suppose we have already verified that an instance of Interval Scheduling–Decision has no conflicts, and another interval $[a^*, b^*]$ is given to us (e.g. another listener tries to buy some airtime). Do we need to spend time $O(n \log n)$ again to decide whether or not we can fit that interval into the schedule?

**A:**

The sorted array in the above solution is an example of a (static) *data structure*: a way of encoding our input data that allows us to answer certain queries about the data efficiently. As with computational problems, we'll want to distinguish the problem that data structures are supposed to solve from the way in which we solve them. Let's begin by formalizing the former:

**Definition 3.1.** A *static abstract data type* is a quadruple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$ where:

- $\mathcal{I}$ is a (typically-infinite) set of possible inputs $x$, and $\mathcal{O}$ is a (sometimes-infinite) set of possible outputs $y$.
- $\mathcal{Q}$ is
- for every $x \in \mathcal{I}$ and $q \in \mathcal{Q}$,

Given such an abstract data type, we want to design efficient algorithms that preprocess the input $x$ into an encoding that allows us to quickly answer queries $q$ that come later. For example, to be able to determine whether a new interval conflicts with one of the original ones, it suffices to solve the following data-structure problem.

**Input:** An array of key-value pairs $x = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, with each $K_i \in \mathbb{R}$

**Queries:**

- `search`$(K)$ for $K \in \mathbb{R}$: output some $(K_i, V_i)$ such that $K_i = K$.
- `predecessor`$(K)$ for $K \in \mathbb{R}$: output some $(K_i, V_i)$ such that $K_i = \max\{K_j : K_j < K\}$.
- `successor`$(K)$ for $K \in \mathbb{R}$: output some $(K_i, V_i)$ such that $K_i = \min\{K_j : K_j > K\}$.

**Abstract Data Type** Static Predecessors & Successors

A data structure is now illustrated in the following figure:

Sometimes (e.g. in the study of programming languages) data structures are referred to as *implementations* of an abstract data type.

Our goal is for Eval to run as fast as possible. (As we'll see in examples below, sometimes there are multiple types of queries, in which case we often separately measure the running time of each type.) Secondarily, we would like Preprocess to also be reasonably efficient and to minimize the memory usage of the data structure Preprocess$(x)$.

Note that there is no pre-specified problem that the algorithms Preprocess and Eval are required to solve individually; we only care that *together* they correctly answer queries. Thus, a big part of the creativity in designing data structures is figuring out what the form of Preprocess$(x)$ should be. Our first example (from the discussion above and encapsulated in the theorem below) takes it to be a sorted array, but we will see other possibilities in subsequent sections (like binary search trees and hash tables).

**Theorem 3.2.** Static Predecessors & Successors *has a data structure in which:*

- Eval$(x', (\texttt{search}, K))$, Eval$(x', (\texttt{predecessor}, K))$, Eval$(x', (\texttt{successor}, K))$ *all take time*
- Preprocess$(x)$ *takes time*
- Preprocess$(x)$ *has size*

Predecessors+Successors have many applications. They enable one to perform a RangeSelect—selecting all of the elements of a dataset whose keys fall within a given range. This is a fundamental operation across many applications and systems, including relational databases (e.g. a university

database selecting all CS alumni who graduated in the 1990s), NoSQL data stores (e.g. selecting all users of a social network within a given age range), and ML systems (e.g. filtering intermediate results during neural network training sessions).

**Other Queries.** `search`, `predecessor`, and `successor` are only a few of the kinds of queries that are useful to ask on a dataset of key-value pairs. Some additional examples are:

1. `min`$(K)$: return some $(K_i, V_i)$ such that $K_i = \min\{K_j : j \in [n]\}$.
2. `rank`$(K)$: return the *number* of pairs $(K_i, V_i) \in S$ such that $K_i < K$.

# 4 Dynamic Data Structures

In many applications, it is often the case that we do not get all of our data at once, but rather it comes in through incremental updates over time. This gives rise to *dynamic* data structures:

**Definition 4.1.** A *dynamic abstract data type* 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 $\epsilon$ is the empty input (e.g. a length 0 array), since the inputs can usually be constructed through a sequence of updates. For example, consider following dynamic version of Predecessors+Successors:

**Updates:**

- `insert`$(K, V)$ for $K \in \mathbb{R}$:
- `delete`$(K)$ for $K \in \mathbb{R}$:

**Queries:**

- `search`$(K)$ for $K \in \mathbb{R}$:
- `predecessor`$(K)$ for $K \in \mathbb{R}$:
- `successor`$(K)$ for $K \in \mathbb{R}$:

**Abstract Data Type** Dynamic Predecessors & Successors()

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 data structure.

Dynamic abstract data types require us to come up with algorithms that also implement the updates in addition to preprocessing and queries:

**Definition 4.2.** Let $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, \mathcal{U}, f)$ be a dynamic abstract data type. A *(dynamic) data structure* for $\Pi$ is a triple of algorithms $(\text{Preprocess}, \text{EvalQ}, \text{EvalU})$ such that

Now, the primary goal in designing dynamic data structures is typically for EvalU and EvalQ to be extremely fast. Secondarily, we often also care about the space usage of the data structure and the time for preprocessing.

# 5 Binary Search Trees

Sorted arrays allow us to answer many queries in time $O(\log n)$ or $O(1)$, which is very fast, but the update times are $O(n)$, which is slower than we'd like. A data structure that will enable us to have $O(\log n)$ runtime for both queries and updates is a *binary search tree*.

## 5.1 Binary Search Tree Definition and Intro

The intuition for binary search trees is to have the encoding of the data mimic the recursive structure of binary search (so that we can still perform queries in time $O(\log n)$) but with more flexibility to add and remove elements than allowed by a sorted array.

**Definition 5.1.** A *binary search tree (BST)* is a recursive data structure. The *empty BST*, denoted $\emptyset$, 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: Thus, the worst-case time for an insertion in a BST is governed by the tree's *height*:

**Definition 5.2** (height of a tree)**.** The *height* $h$ of a BST is

## 5.2 Operations on Binary Search Trees

**Theorem 5.3.** *Given a binary search tree of height $h$, 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 imposes a constraint for 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:**

```
   Insert(T,(K,V)):
 0 if T = ∅ then
 1 |
 2 else
 3 |  Let v be the root of T, T_L its left-subtree, and T_R its right-subtree;
 4 |  if K ≤ K_v then
 5 |  |
 6 |  |
 7 |  |
 8 |  else
 9 |  |
10 |  |
11 |  |
```

**Algorithm 5.1:** `Insert()`

**Runtime:**

**Proof of Correctness:**

- **Search:**

  The algorithm is as follows:

  **Proof of correctness:** The 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 `Insert` and gives a runtime of $O(h)$.

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

- **Predecessor (and Successor):**
  The algorithm for next-smaller follows along related lines to search algorithm:

- **Deletions:** The algorithm for deletion is the subject of the Sender–Receiver Exercise in the next class.

□

## 6 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 the data structure produced by one such approach that is relatively easy to describe, called AVL trees, after mathematicians named Adelson-Velsky and Landis.

**Definition 6.1** (AVL Trees)**.** An *AVL Tree* is a binary search tree that is:

1.

2.

**Lemma 6.2.** *Every nonempty height-balanced tree with $n$ vertices has height at most $2\log_2 n$.*

*Proof.*

□

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:*

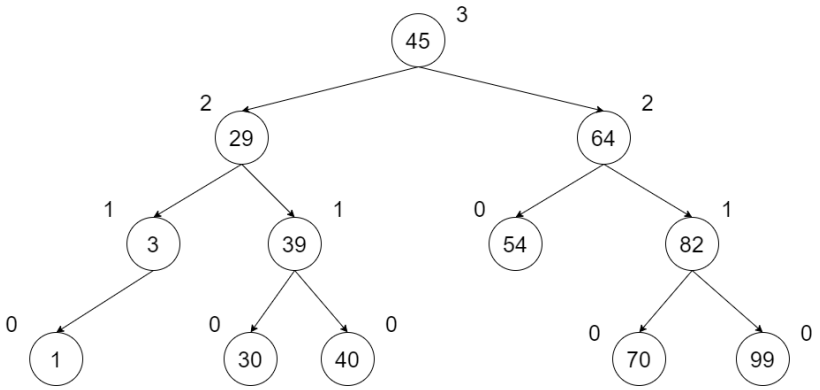
**Runtime of a rotation:**

**Theorem 6.3.** *We can insert/delete a key-value pair to/from an AVL Tree $T$ while preserving the AVL Tree properties in time $O(\log n)$.*

*Proof Sketch.* We informally describe the idea for insertion; for deletion, similar ideas can be applied to the BST deletion method from the Sender–Receiver Exercise.

□

**Example 6.4.** We depict the algorithm with an example, where the value attributes are chosen to be empty. Consider the following AVL tree:




We insert 65 by creating a new leaf $\ell$, resulting in the following tree. We update the heights of the corresponding vertices and check the height balance, which is violated because:

To fix this, we then use rotations. Let $y$ be the vertex of key 64, and $z$ be the vertex of key 82.

We apply two rotations to restore height-balance. The first one is `Right.rotate(z)`, which results in the following tree:

Next, we apply `Left.rotate(y)` to result in the following, height-balanced tree:

Notice that the height of the root (45) has changed from 4 to 3, so if this were a subtree of a larger tree where 45 had a sibling of height 5, then we would still violate height-balance and would need to keep working our way up the tree, using rotations to repair it.

Combining Theorem 6.3 with Theorem 5.3, we deduce:

**Theorem 6.5.** Dynamic Predecessors & Successors *has a data structure in which all queries and updates can be performed in time* $O(\log n)$*, where* $n$ *the current number of key-value pairs stored by the data structure.*

By Theorem 5.3, `min` and `max` queries can be supported by the same data structure in $O(\log n)$. Using size augmentation as in the next problem set , we can also support `select` and `rank` queries in time $O(\log n)$.

# 7 Implementing Data Structures

Object-oriented programming (like programming in Python) provides a convenient way to implement data structures that aligns with our formalization above. Specifically, we can represent data structures in Python by using Python `class`es, (like on Problem Set 0),which are similar to C `struct`s, but also allow us to attach *methods* that implement the Preprocess and Eval functions of a data-structure solution. The implementation details of these methods (as well as the private attributes) can be hidden from users of the `class`, creating an "abstraction barrier" that allows or even forces a user of the class to focus on the data-structure problem that it solves, without concern for the particular solution. (This is similar to how the notion of *reduction* $\Pi \leq \Gamma$ above abstracts away how the problem $\Gamma$ is solved when using it to solve $\Pi$.) See the textbook for sample code.