CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 4: Dynamic Data Structures

Harvard SEAS - Fall 2021 *Sept. 14, 2021*

1 Announcements

Recommended Reading:

- CS50 Week 5: https://cs50.harvard.edu/college/2020/fall/notes/5/
- Roughgarden II Sections 11.0–11.3.2
- CLRS Chapter 10, Sections 12.0–12.1
- Salil OH today 4-5 on Zoom
- Panopto recording is higher quality than zoom recording
- PS1 due tomorrow
- Likely an active learning exercise Thursday

2 Reflections and Feedback

Selections from Active-Learning Reflection:

- "There was a big "aha" moment when I used an example to describe how counting sort worked."
- "It would be really helpful if for the sender there was a video sent out with a visual intuition for the algorithm/proof, or additional resources other than the paper sent."
- "I think that my preparation before class (taking the time to understand, walk through, and explain the proof to myself multiple times) enabled me to explain the proof clearly and methodically."
- "I think I communicated well, establishing more of a dialogue than a lecture"
- "Next time I prepare to be a sender, I can take thorough notes at the start (since I find that helpful), but I should also have an abbreviated version with higher-level cues/reminders and then try to communicate my conceptual understanding. - I should understand every detail of the proof."
- "I was ready to just into the explanation, but my partner first started by asking: "What are we trying to prove?" to start getting a foothold."

- "I would like to communicate more effectively about when something should be written down (eg is technical / specific enough for putting pen to paper) versus just for a high-level understanding"
- "My partner was unclear on the idea of the universe of [U] and how this differs from the universe for comparison-based sorting algorithms."
- "We weren't confident about the exact formality required of certain parts of the proof so we asked Salil."
- "My partner ran into some challenges conceptualizing the proof in practice, which is understandable since my explanation was quite abstract."
- "The idea of turning the count array to an array of linked lists was the hardest part to communicate today. We overcame these challenges by making the receiver draw out what they thought each array should look like, and I would tell them which parts were misunderstood, what was correct."
- "My partner did great! She took clear, concise notes, spoke up immediately when something was unclear, and asked effective questions about important details (like why the runtime of certain parts of the computation were what they were). This pushed me to make sure I *really* understood the proof."
- Challenges with groups of 3.
- "Yes, he asked specific questions on minute steps/leaps I may have been hand-waving accidentially, which allowed me to think very specifically and refine my own understanding (or at least formalize it to the level of verbal communication), which in turn helped him gain the understanding of those low-level steps as well."
- "My partner did an amazing job at creating visuals!"
- "I confirmed my understanding at each step of the explanation and repeated my interpretation of what was explained back to the sender to ensure I was on the same page. I participated actively in the explanation and helped come up with analogies/connections with what we had learned to help us both understand the material better. Continuous feedback was key."
- "I asked different questions and drew diagrams of what I thought they were explaining to me. They would give me feedback on my diagrams and would tell me if I was understanding it right or wrong."
- "It was difficult to take notes and pay full attention to the sender, so I gave up on it early on. It's hard to know what to write down. Next time, I'll ask to have a quick high level overview of the topic before the detailed explanation so that I know what's important to note."
- "I can improve by learning how to analyze the time complexity of algorithms better maybe by reading the CLRS book."
- "I can improve by reviewing previous definitions, which would help my understanding a little more when writing my interpretation of the proof."

- "I think I should spend a longer amount of time trying to think of a possible solution for myself before caving into the explanation."
- "I could definitely be a better listener, as well as being more thoughtful in what I ask. This involves being more focused."
- "Since this was our first group exercise, it was up in the air on how best to execute."
- "I think we might have found it a little challenging to not give everything away and give me time to consider the next steps."
- "Confusion regarding item-key pairs vs. key-value pairs which we were more used to, the motivation behind needing a linked list."
- "My partner was fantastic. They were especially keen on breaking down the steps of the algorithm and continuing their explanation only when I had fully understood the process of one step."
- "Explained the concepts very well to me with illustrations and answered my questions to the best of their ability. Gave the big picture which helped a lot."
- "My partner did not prepare for the explanation very well... The expectation to come prepared for class is now much more clear between the group which will help moving forward."

Some Problem Set 0 Feedback (60 responses):

- Median time was 8-12 hours, but 15% of the class spent 20+ hours (you are not alone - do not get discouraged! we will work on reducing this)
- Bulk of time and highest variance: figuring out how to solve the theory problems
- Approx 50% of respondents disagreed (somewhat or strongly) that cs20 provided sufficient skills for the theory problems. We miscalibrated!
- Approx 60% of respondents agreed (somewhat or strongly) that cs50 provided sufficient skills for the programming problems.
- 38% of the students did not attend a section 0.
- "Would love to see the answers to the section problem"
- Did not like breakout rooms in Zoom OH
- Office hours hectic, especially in dining hall during mealtime
- Reluctance of TFs to "give away the answer"
- Prefer one platform for course materials
- Readings don't completely sync with lecture. Readings are meant as supplementary and/or to help review material from past courses (like linked lists, arrays, queues, stacks) — generally not required (except when we specifically say otherwise), and will often use different notation than class.

3 Recap and Loose Ends

Using Static Predecessors+Successors for Interval Scheduling:
Suppose we have a list $A = \{[a_0, b_0], \ldots, [a_{n-1}, b_{n-1}]\}$ of start and end times for intervals. We can initialize the StaticPredecessor data structure $D = \text{Preprocess}(A)$ on this list, with the keys equal to the start times and items equal to the interval. Then to check if a new interval $[a, b]$ "fits", we can run $D.\text{Eval}(a)$ to find the interval $[a_p, b_p]$ with the closest starting time preceding a. Then if $b_p < a$, we have that the start time doesn't overlap with the prior assigned block. To confirm that we don't overlap the *next* assigned block, we can query $D.\text{Eval}(b)$ and see if we obtain $[a_p, b_p]$. If we did, the predecessor to the end time is the prior start time, so there is no conflict. We could also implement successor queries in our static data structure.

Remark. CLRS only defines the predecessor problem where the query is a key of already in the set. However, the more general formulation we have given (where q can be any real number) is more standard and more useful.

Implementing Static Data Structures. As you saw on Problem Set 0, we can represent data structures in Python by using Python `class`es similarly to C `struct`s. However, we can 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 a user of the class, creating an "abstraction barrier" that allows the user to focus only on the data-structure problem that it solves, without concern for the particular solution. (This is the notion of an "abstract data type" that some of you may have seen in CS51.) For example:

```
class StaticPredecessor:
      def __init__(self,A: list): # preprocessing method
        # Python's built-in sorting doesn't take keys as explicit inputs,
        # but asks the user to specify a function that defines the keys
        def extractkey(pair): return pair[1]
        self.sortedarray=sorted(A,key=extractkey)

      def eval(q: float):
        # put code for binary search here
        # ...
        return pair

    MyDS = StaticPredecessor([("a",5),("b",3),("c",7),("d",2)])  # runs __init

    MyDS.eval(4) # should return ("b",3)
```

Note: Despite the name, a Python `list` is implemented as an array rather than as a linked list.

4 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 4.1. A *dynamic data structure problem* Π is given by

- 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 $\mathcal{I} = \emptyset$, since the inputs can usually be constructed through a sequence of updates. For example:

Updates: Insert an item-key pair (I, K) with $K \in \mathbb{R}$ **Queries :** Given a threshold $q \in \mathbb{R}$, return (I_i, K_i) such that $K_i = \max\{K_j : K_j \leq q, j \in [n]\}$, where $(I_0, K_0), (I_1, K_1), \ldots, (I_{n-1}, K_{n-1})$ is the sequence of all prior insertions, or $\perp$ if $q < \min\{K_j : j \in [n]\}$

Data-Structure Problem Dynamic Predecessors

In addition to insertions, we may also include other updates, like *deletions*, to the definition of our dynamic data structure problem.

Definition 4.2. For a dynamic data structure problem $\Pi = (\mathcal{I}, \mathcal{Q}, \mathcal{U}, f)$, a *solution* is a triple of algorithms $(\text{Preprocess}, \text{EvalQ}, \text{EvalU})$ such that for all $x \in \mathcal{I}$, $u_0, u_1, \ldots, u_{n-1} \in \mathcal{U}$ and $q \in \mathcal{Q}$, we have

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

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, since methods are allowed to modify the stored attributes.

4.1 Binary Search Trees

Definition 4.3. A *binary search tree (BST)* is a recursive data structure where every node v has:

- an item I_v
- a key K_v
- a pointer to a left child v_L (or `None`)
- a pointer to a right child v_R (or `None`)

Of course, we require that the parents of v_L and v_R (if they exist) are v. 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.

Example: