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

Lecture 4: Data Structures

Harvard SEAS - Fall 2022 *Sept. 13, 2022*

1 Announcements

2 Recommended Reading

- CLRS Chapter 10
- Roughgarden II, Sec. 10.0–10.1, 11.1
- CS50 Week 5: https://cs50.harvard.edu/x/2022/weeks/5/

3 Interval Scheduling

The lecture notes for the previous class introduced the IntervalScheduling-Decision computational problem, motivated by checking whether a collection of time reservations (e.g. radio airtime) have any conflicts:

Input	: A collection of intervals $[a_0, b_0], \ldots, [a_{n-1}, b_{n-1}]$, where each $a_i, b_i \in \mathbb{R}$ and $a_i \le b_i$
Output	: YES if the intervals are disjoint from each other (for all $i \ne j$, $[a_i, b_i] \cap [a_j, b_j] = \emptyset$) NO otherwise

Computational Problem IntervalScheduling-Decision

The notes show that IntervalScheduling-Decision can be solved efficiently by a reduction to *Sorting*:

Proposition 3.1. *IntervalScheduling-Decision on inputs that consist of n intervals can be reduced to Sorting an array of n key-value pairs in time $O(n)$. That is, IntervalScheduling-Decision $\le_{O(n),n}$ Sorting.*

As a corollary, IntervalScheduling-Decision can be solved in time $O(n \log n)$.

Q: Suppose we have already solved IntervalScheduling in this way, 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 we can fit that interval in?

4 Static Data Structure

The sorted array in the above solution is an example of a (static) data structure. Let's abstract what data structures are supposed to do.

Definition 4.1. A *static data structure problem* is a quadruple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$ where:

-
-
-

For such a data-structure problem, we want to design efficient algorithms that preprocess the input x into a data structure that allows for quickly answering 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 (K_i, V_i) such that $K_i = K$.
- Next-smaller(K) for $K \in \mathbb{R}$: output (K_i, V_i) such that $K_i = \max\{K_j : K_j < K\}$.

Data-Structure Problem Static Predecessors

To formalize these two types of queries using Definition 4.1, we can take

$$\mathcal{Q} =$$

Note that both types of queries may have no valid solution, or may have multiple solutions. (Why?) If we removed the Next-smaller queries and only kept Search queries, we would have the (static) *Dictionary* data structure problem, which we will study in a couple of weeks.

Using Static Predecessors+Successors for Interval Scheduling:

Remark. The terminology *predecessor* comes from the fact that when K is a key in the input array x, then Next-smaller(K) should return the *predecessor* of K. 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 K can be any real number) is more standard and more useful.

Predecessor Data Structures (and the equivalent Successor Data Structures) have many applications. They enable one to perform a "range select" — selecting all of the elements of a dataset whose keys fall within a given range. This is a fundamental operation across many application and systems, including relational databases (e.g. a university database selecting all CS alumni who graduated in the 1990's), 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).

Definition 4.2. For a (static) data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$, an *implementation* is a pair of algorithms $(\text{Preprocess}, \text{Eval})$ such that

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).

A solution to the Static Predecessor Problem:

- Preprocess(x):
- Eval(x', (`search`, K)):
- Eval(x', (`next-smaller`, K)):

5 Implementing 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, x: 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
        self.sortedarray = MergeSort(x)

      def search(self, K: float):
        left = 0
        right = len(self.sortedarray) - 1
        mid = 0

        while left <= right:

            mid = (left + right) // 2

            # If K is greater, ignore left half
            if self.sortedarray[mid][0] < K:
                left = mid + 1

            # If K is smaller, ignore right half
            elif self.sortedarray[mid][0] > K:
                right = mid - 1

            else:
                return self.sortedarray[i]

        # if no solution is found
        return None

      def nextsmaller(self, K: float):
        # ...

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

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

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

6 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 6.1. A *dynamic data structure problem* is a quintuple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{U}, \mathcal{Q}, f)$ where:

-
-
-
-

Often we take $\mathcal{I} = \emptyset$, 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 (i.e. data structure implementation).

Definition 6.2. For a dynamic data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, \mathcal{U}, f)$, an *implementation* 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 Section 5, with the update operations also implemented as methods.

7 In-class Exercise

Suppose we use a sorted array to implement a data structure storing a dynamically changing multiset S of key-value pairs with insertions and deletions. How efficiently can you perform each of the following operations (in the worst case), as a function of the current number n of elements of S? Possible answers are $O(1)$, $O(\log n)$, $O(n)$, and "other".

1. Insert(K, V)
2. Delete(K)
3. Min(): return the smallest key K in S.
4. Rank(K): return the *number* of pairs $(K', V') \in S$ such that $K' < K$.