COS 226: midterm prep

Plan for today

- Data structure design
- Algorithm design
- Analysis of recursive algorithms

How do you feel about the midterm?

A. 😃
B. 😐
C. 🙁
D. 😬
E. 🤷‍♀️

Algorithm design [Fall 2018]

Given two integer arrays `a[]` and `b[]`, find an integer that appears in both arrays (or report that no such integer exists). Let m and n denote the lengths of `a[]` and `b[]`, respectively, and assume that $m \leq n$.

Here are the performance requirements:

- *Space:* the amount of extra space (besides `a[]` and `b[]`) must be constant. It is fine to modify `a[]` and `b[]`.
- *Time:* the order of growth of the running time must be $n \log m$ in the worst case.

Algorithm design [Fall 2018]: Thought process

Given two integer arrays `a[]` and `b[]`, find an integer that appears in both arrays (or report that no such integer exists). Let m and n denote the lengths of `a[]` and `b[]`, respectively, and assume that $m \le n$.

Here are the performance requirements:

- *Space:* the amount of extra space (besides `a[]` and `b[]`) must be constant. It is fine to modify `a[]` and `b[]`.
- *Time:* the order of growth of the running time must be $n \log m$ in the worst case.

- [Don't worry about performance]: search each item from `a[]` in `b[]`
- Hmm, it would be faster if `b[]` were sorted first.
- But is it OK to sort? — Must be in-place; worst case matters
  - Ah, question says it's fine to modify `a[]` and `b[]`.
- Wait, what if we swap the order of `a[]` and `b[]`? The question does mention $m \le n$, so it wants us to think carefully about the order.
- Let's consider both orders.
  - Sort the smaller array: $\boxed{m \log m} + n \log m$
  - Sort the larger array: $n \log n + \boxed{m \log n}$

(Sort: $n \log n$; Binary searches: $m \log n$)

Can be dropped in order-of-growth calculation since $m \le n$.

Data structure design [Spring 2015]

Design an efficient data type to store a *threaded set of strings*, which maintains a set of strings (no duplicates) and the order in which the strings were inserted, according to the following API:

`public class ThreadedSet`

	`ThreadedSet()`	*create an empty threaded set*
`void`	`add(String s)`	*add the string to the set (if it is not already in the set)*
`boolean`	`contains(String s)`	*is the string s in the set?*
`String`	`previousKey(String s)`	*the string added to the set immediately before s (null if s is the first string added; exception if s is not in set)*

Here is an example:

```
ThreadedSet set = new ThreadedSet();
set.add("aardvark");                // [ "aardvark" ]
set.add("bear");                    // [ "aardvark", "bear" ]
set.add("cat");                     // [ "aardvark", "bear", "cat" ]
set.add("bear");                    // [ "aardvark", "bear", "cat" ]
                                    // (adding a duplicate key has no effect)
set.previousKey("cat");             // "bear"
```

Data structure design [Spring 2015]: Thought process

Design an efficient data type to store a *threaded set of strings*, which maintains a set of strings (no duplicates) and the order in which the strings were inserted, according to the following API:

Hmm.. maybe stack or queue? Nevermind, let's just look at the API.

`public class ThreadedSet`

	`ThreadedSet()`	*create an empty threaded set*
`void`	`add(String s)`	*add the string to the set (if it is not already in the set)*
`boolean`	`contains(String s)`	*is the string s in the set?*
`String`	`previousKey(String s)`	*the string added to the set immediately before s (null if s is the first string added; exception if s is not in set)*

OK… can be any collection

Ah! Must be a dictionary

Right, so this is what they mean by order. We just need to map each string to previous

My mind doesn't even go to BST because it is dominated by the LLRB tree.

- So red-black tree or hash table? Either might work.
- Maybe it doesn't matter. Let's try to do it with an abstract symbol table.
- Adding a duplicate key has no effect.. confirms that symbol table is the right track
- Keep track of last added string in an instance variable.. null at the beginning
- Confirm that this satisfies the requirements
- Reading to the end, we notice:

Under reasonable technical assumptions, what is the order of growth of each of the methods as a function of the number of keys N in the data structure? Assume that the length of all strings is bounded by a constant.

Subtle hint that hashing is preferable… constant string length means lookups take constant amortized time

New this semester: no hint about hash tables

Expect this statement in *every* design problem, regardless of whether or not hash tables make sense:

"You may make any standard technical assumptions that we have seen in this course."

Algorithm design: anagrams [Spring 2016]

Treat string as char array; sort both; test if equal

Call two strings equivalent if one is an anagram of the other. An equivalence class is a set in which any pair is equivalent.

Equivalent

Example: ["aaa", "aab", "aba"] has two equivalence classes.

Given an array of strings, design an algorithm to find the number of equivalence classes among them.

Performance requirement: worst-case order of growth running time $NM (\log N + \log M))$ where N is the length of the array and M is the max length of the strings.

Understanding the question:

- How do you test if two strings are equivalent to each other?
- What is the relationship between equivalence and duplicates?

First sort each string (char array). Then:

- Method 1: insert into symbol table (LLRB tree); query the size
- Method 2: sort the array of strings; then traverse array, count # of key changes

Performance (method 1):

- $N (M \log M) + N (\log N) M$

Char compares per string compare

Sorting each string

String compares per insert

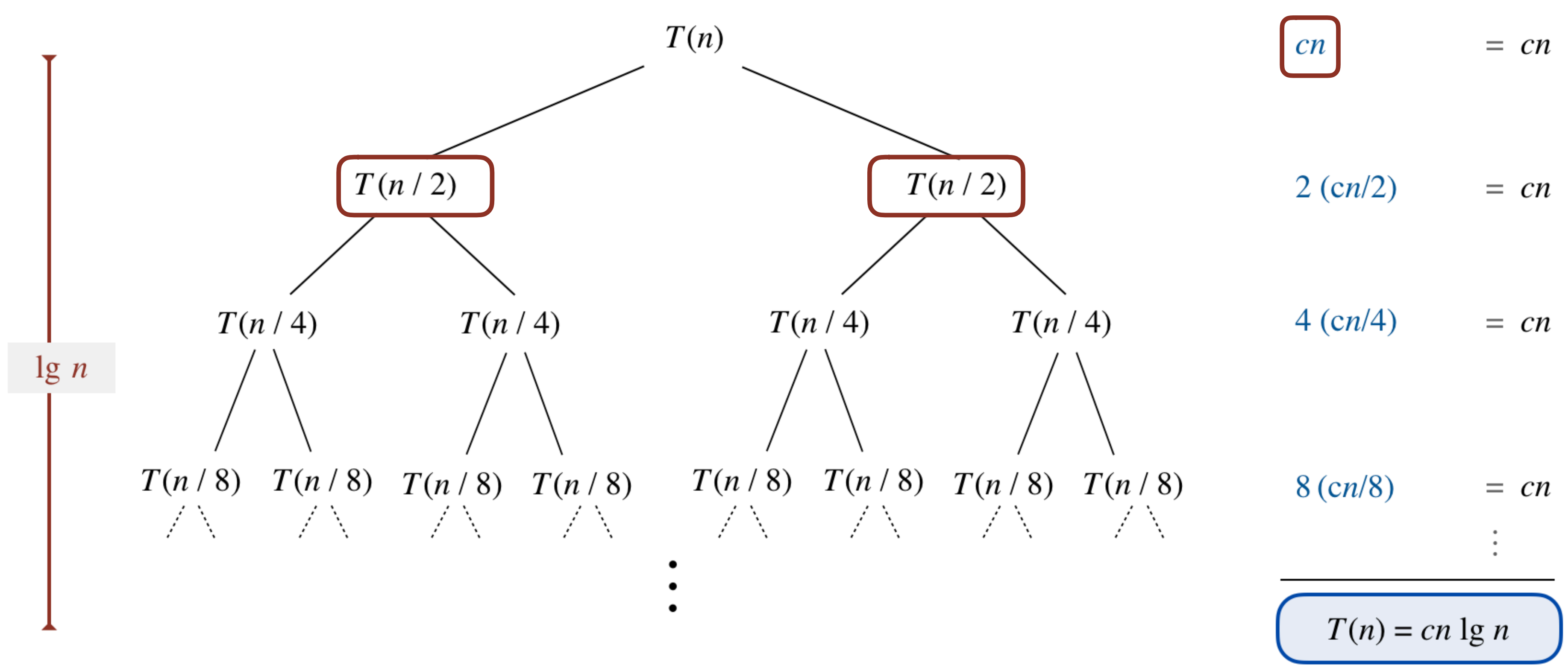
Analysis of recursive algorithms [Fall 2015; modified here]

- **Algorithm 1** solves problems of size N by recursively dividing them into 2 sub-problems of size $N/2$ and combining the results in time c (where c is some constant).
- **Algorithm 2** solves problems of size N by solving one sub-problem of size $N/2$ and peforming some processing taking some constant time c.
- **Algorithm 3** solves problems of size N by solving two sub-problems of size $N/2$ and performing a *linear* amount (i.e., cN where c is some constant) of extra work.

For each algorithm

Running time for problems of size N

- complete the equation $T(N)$ = ____ * $T(N/2)$ + _____ (e.g. Algorithm 3: $T(N) = 2\,T(N/2) + cN$)
- Think of concrete algorithms that match the pattern (e.g. Algorithm 3: mergesort)
- Solve for $T(N)$ by picture (e.g. solution for Algorithm 3 shown below)



Analysis of recursive algorithms [Fall 2015]

- **Algorithm 1** solves problems of size N by recursively dividing them into 2 sub-problems of size $N/2$ and combining the results in time c (where c is some constant). $T(N) = 2\ T(N/2) + c \Rightarrow T(N) \sim cN$
- **Algorithm 2** solves problems of size N by solving one sub-problem of size $N/2$ and peforming some processing taking some constant time c. $T(N) = T(N/2) + c \Rightarrow T(N) \sim c \log N$
- **Algorithm 3** solves problems of size N by solving two sub-problems of size $N/2$ and performing a *linear* amount (i.e., cN where c is some constant) of extra work. $T(N) = 2\ T(N/2) + cN \Rightarrow T(N) \sim c\ N \log N$

Concrete examples:

- Algorithm 1: in-order/pre-order/post-order traversal of a complete binary tree.
- Algorithm 2: binary search.
- Algorithm 3: mergesort.

(Quicksort is similar but not the same; heapsort is not recursive.)

Subtlety: the answers above are a slight abuse of tilde notation. As opposed to order of growth, constant factors matter in tilde notation, so we must specify the base of the logarithm rather than simply write $\log N$ (which leaves the base unspecified).

Data structure + algorithm design [Fall 2012]

Given k sorted arrays with N total keys, is there a key that appears more than once?

Performance requirement: $N \log k$ worst case; Extra space proportional to k.

[Rules out a single big symbol table]

First attempt: check for duplicates among the k smallest elements; if none, remove them and repeat. Doesn't work.

How to fix? Remove only remove globally smallest element.

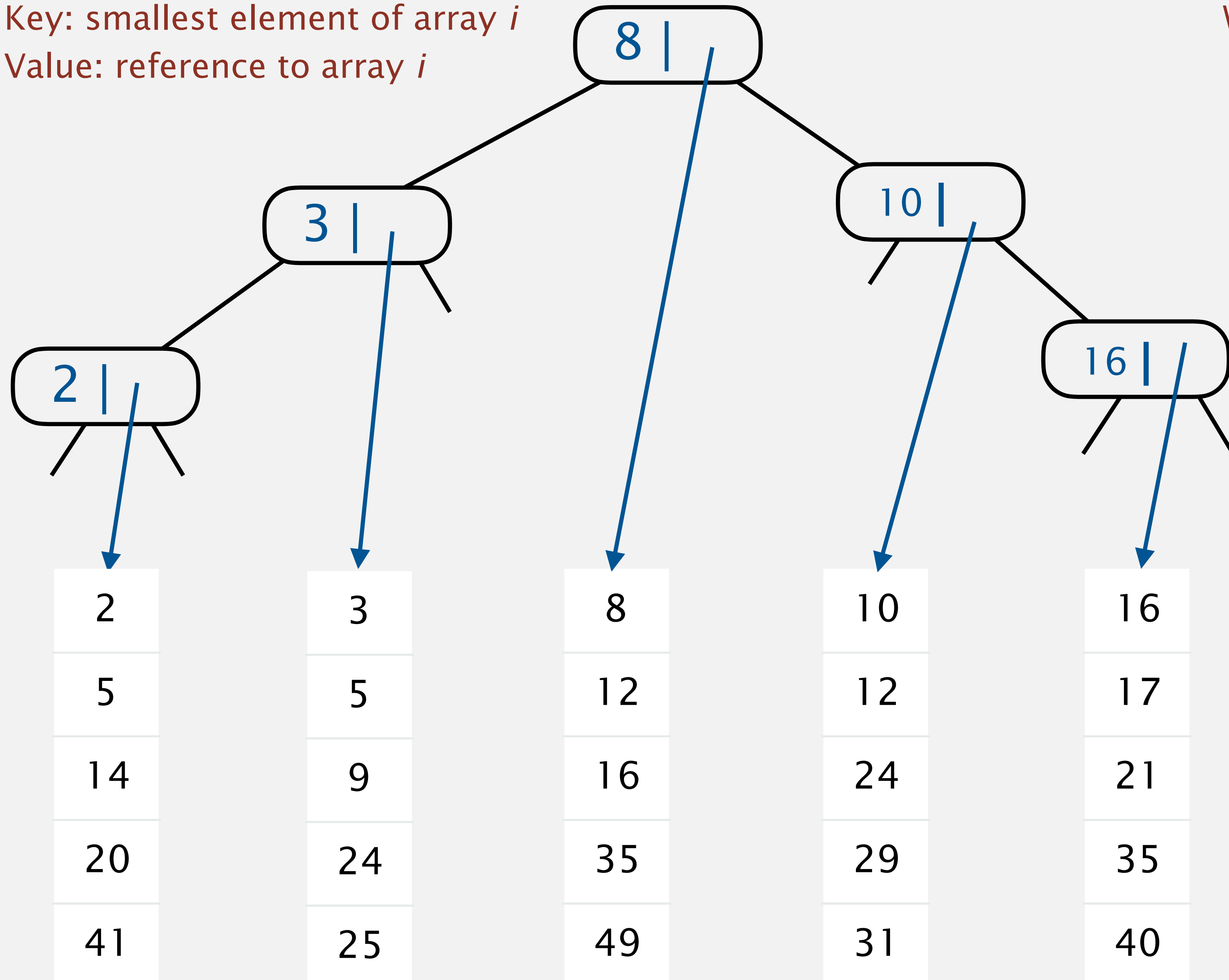
How to check if it appears among the other k-1 elements? Use a symbol table.

2	3	8	10	16
5	5	12	12	17
14	9	16	24	21
20	24	35	29	35
41	25	49	31	40

Main idea: search tree containing the smallest element from each array

Key: smallest element of array *i*

Value: reference to array *i*



While tree not empty:

Delete the smallest element from tree...

... and from corresponding array

Is next element from that array in the tree?

Yes ==> duplicate found

No ==> add it to the tree and repeat

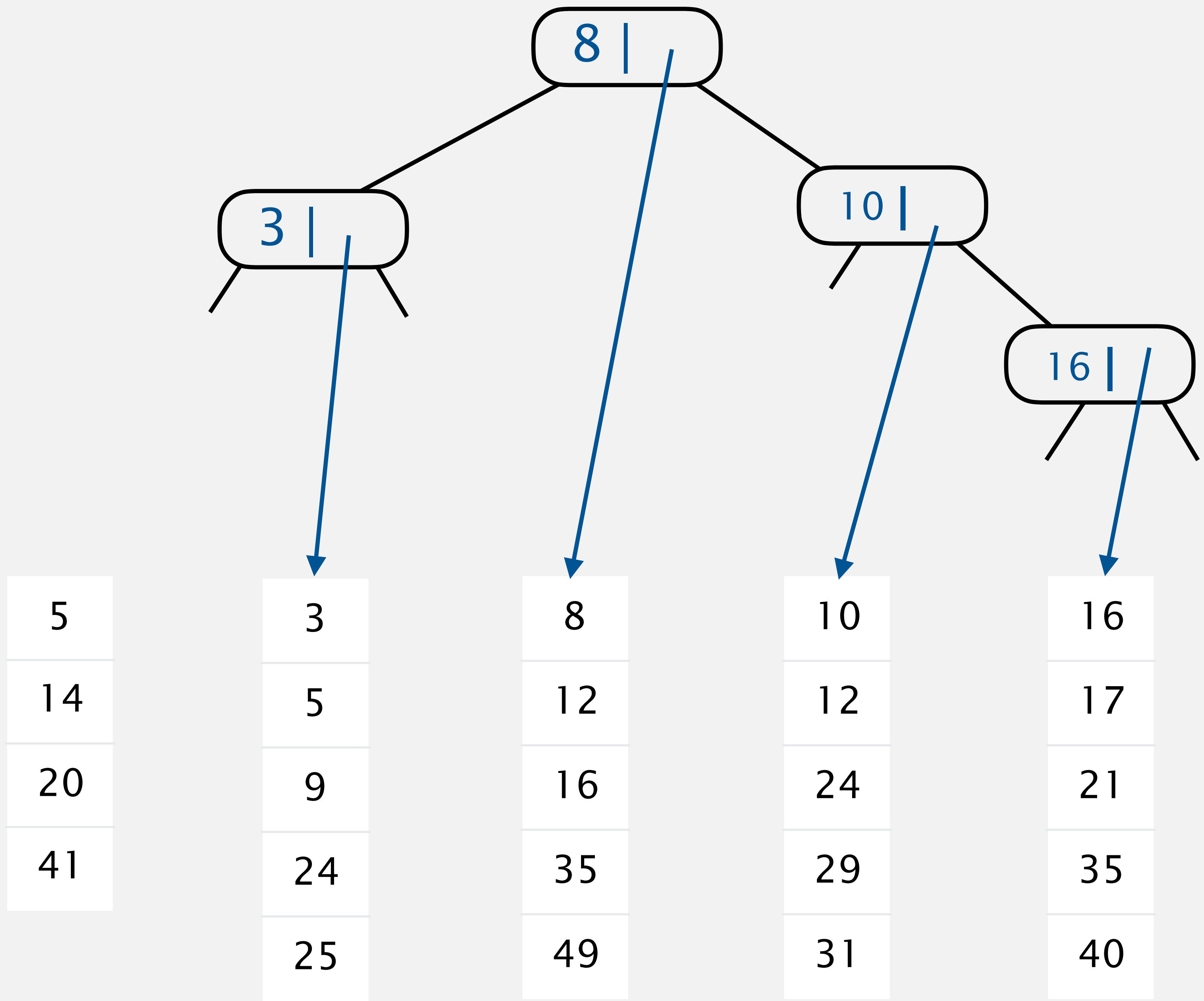
Notes

- This step is preceded by checking for duplicates within each sorted array
- BST shown for simplicity; for best performance, use LLRB tree instead
- Deletion from an ordered array is expensive; instead just keep track of how many elements have been "deleted"

Main idea: search tree containing the smallest element from each array

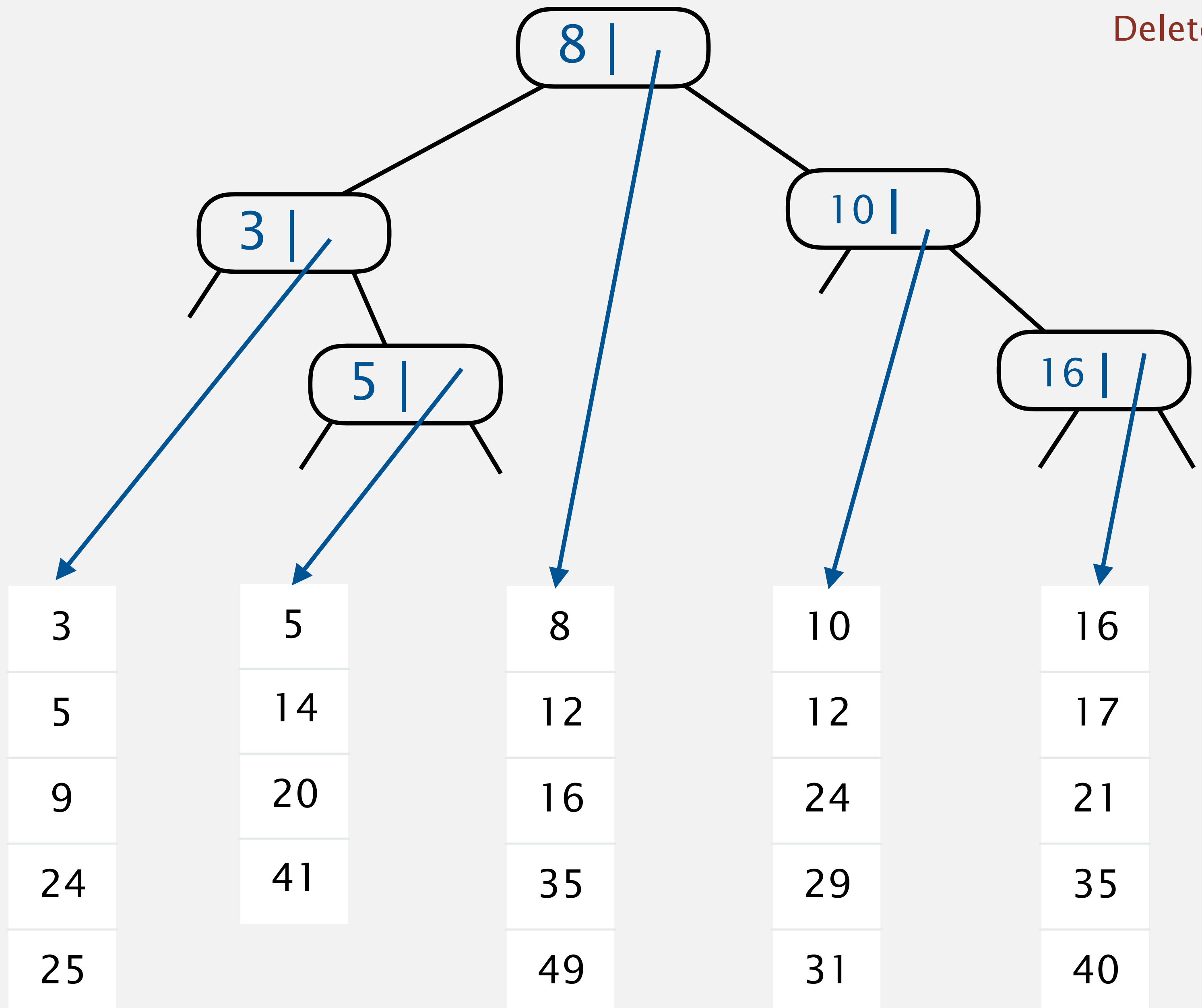
Is 5 in search tree?

No. Insert 5

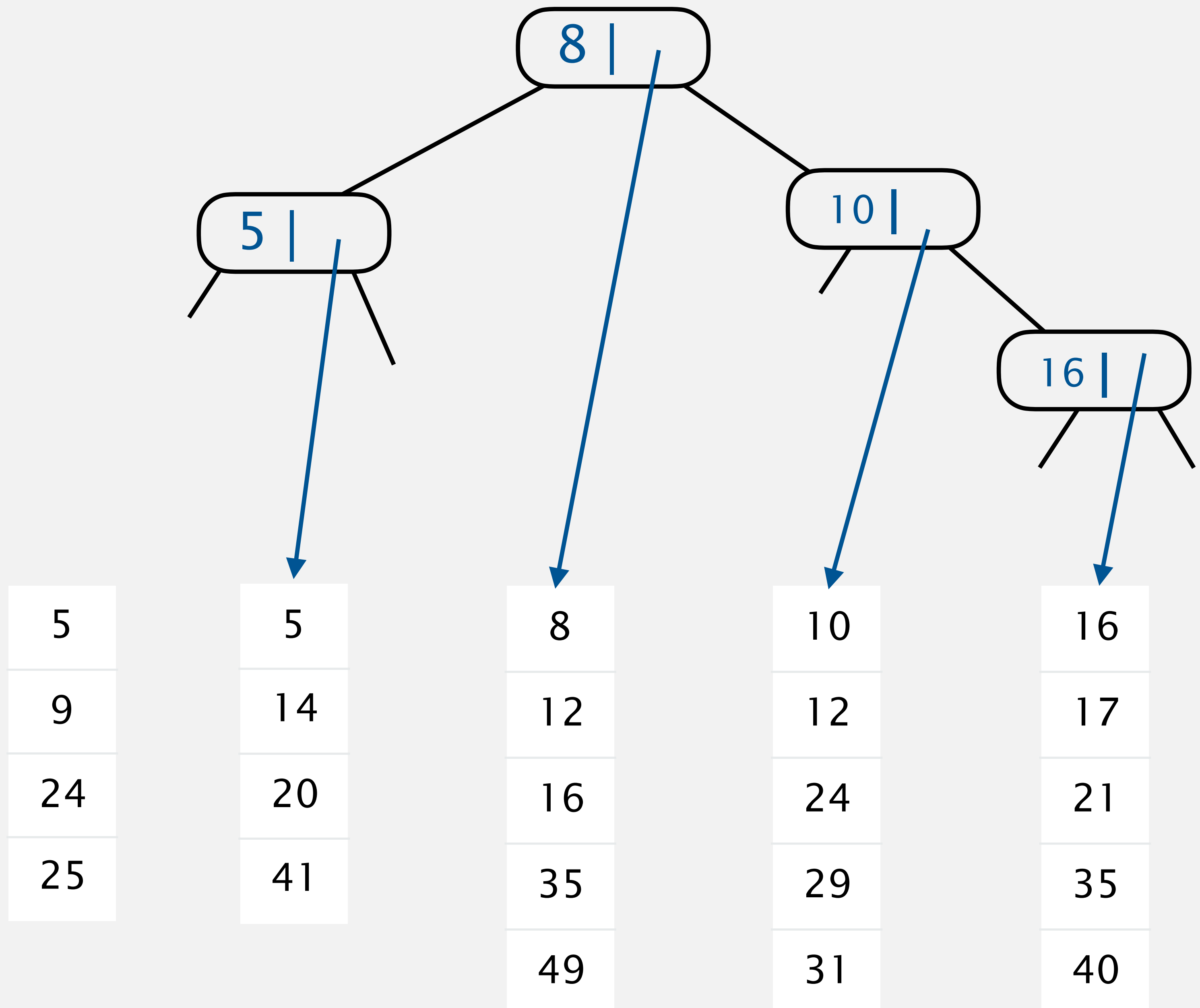


Main idea: search tree containing the smallest element from each array

Delete the smallest entry from search tree



Main idea: search tree containing the smallest element from each array



Is 5 in search tree?

Yes! Duplicate found

Some final tips

- All questions are eligible for partial credit
- Attempt the problems in order of difficulty
- Details matter — many opportunities for 1-point deductions
- Example of an easily missed detail: design solution that uses sorting — which sort algorithm? Does the question constrain your choices?
- Design question: get to a working but inefficient solution quickly; then iterate
- Commonly seen data structures/algorithms in past design questions:
  - Symbol table (either LLRB tree or hash table; review the differences)
  - Sorting arrays followed by binary search
  - Priority queues (a distant third)
- Don’t start writing as soon as you get the high-level idea. Take a minute to express your solution clearly. This matters.