CS 161 (Stanford, Winter 2026) Section 4

1 Warm-up: Binary Search Trees vs Heaps

For each of the following, choose the corresponding data structure. In this problem, "efficiently" means $O(\log n)$ time.

1. With this data structure you can efficiently find the element with key value 2025.

   (A) Red-black binary search trees (B) Heaps (C) Both (D) Neither

   Solution

   (A)

2. With this data structure you can efficiently find the smallest element.

   (A) Red-black binary search trees (B) Heaps (C) Both (D) Neither

   Solution

   (C)

3. With this data structure you can efficiently find the median element.

   (A) Red-black binary search trees (B) Heaps (C) Both (D) Neither

   Solution

   Trick question! Technically, for the data structures we saw in lecture, the answer is neither. But when constructing and maintaining BSTs it's easy for each node to store the number of descendants in its sub-tree. Then we can find the median by starting from the root and recursively going tothe child that contains the median.

4. This data structure is fast on average, but will be slow in the worst-case.

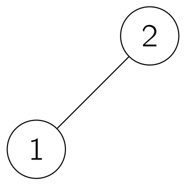
   (A) Red-black binary search trees (B) Heaps (C) Both (D) Neither

   Solution

   (D)

For each of the following, choose the corresponding data structure.

5.

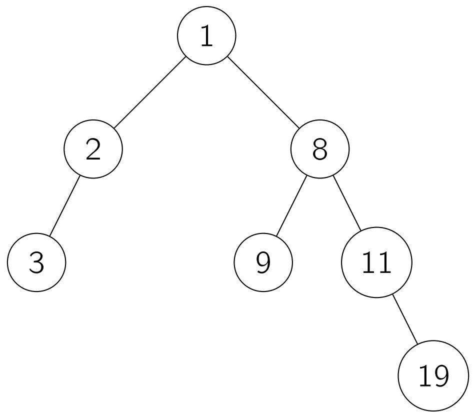


(A) Red-black binary search tree (B) Max-heap (C) Both (D) Neither

Solution

(C) The root is greater than all children and descendants so it satisfies the heap property, and the root is also greater than its left child (which you can color red), so it also is a red-black binary search tree.

6.

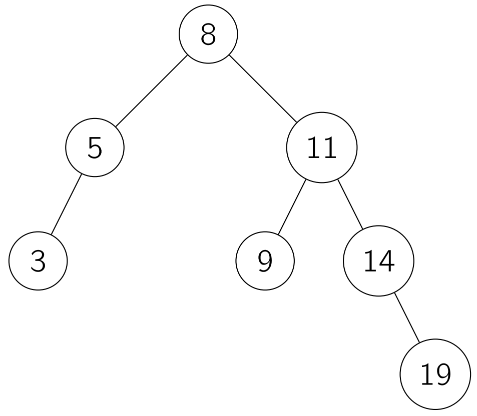


(A) Red-black binary search tree (B) Min-heap (C) Both (D) Neither

Solution

(D) This is almost a heap, but the tree is not complete! A heap must fill in all levels before adding nodes to the next level.

7.



(A) Red-black binary search trees (B) Heaps (C) Both (D) Neither

Solution

(A) Coloring 3, 11, 19 red and the rest black satisfies the definition.

2 Randomly Built BSTs

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\log n)$. A *randomly built binary search tree* with n nodes is one that arises from inserting the n keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely. Let $d(x, T)$ be the depth of node x in a binary tree T (The depth of the root is 0). Then, the average depth of a node in a binary tree T with n nodes is

$$\frac{1}{n}\sum_{x \in T} d(x, T).$$

1. Let the *total path length* $P(T)$ of a binary tree T be defined as the sum of the depths of all nodes in T, so the average depth of a node in T with n nodes is equal to $\frac{1}{n}P(T)$. Show that $P(T) = P(T_L) + P(T_R) + n - 1$, where T_L and T_R are the left and right subtrees of T, respectively.

Solution

Let $r(T)$ denote the root of tree T. Note the depth of node x in T is equal to the length of the path from $r(T)$ to x. Hence, $P(T) = \sum_{x \in T} d(x, T)$. For each node x in T_L, the path from $r(T)$ to x consists of the edge $(r(T), r(T_L))$ and the path from $r(T_L)$ to x. The same reasoning applies for nodes x in T_R. Equivalently, we have

$$d(x, T) = \begin{cases} 0, & \text{if } x = r(T) \\ 1 + d(x, T_L), & \text{if } x \in T_L \\ 1 + d(x, T_R), & \text{if } x \in T_R \end{cases}$$

Then,

$$\begin{aligned}\sum_{x\in T} d(x,T) &= d(r(T),T) + \sum_{x\in T_L} d(x,T) + \sum_{x\in T_R} d(x,T)\\ &= 0 + \sum_{x\in T_L}[1 + d(x,T_L)] + \sum_{x\in T_R}[1 + d(x,T_R)]\\ &= |T_L| + |T_R| + \sum_{x\in T_L} d(x,T_L) + \sum_{x\in T_R} d(x,T_R)\\ &= n - 1 + P(T_L) + P(T_R).\end{aligned}$$

It follows that $P(T) = P(T_L) + P(T_R) + n - 1$.

2. Let $E(n)$ be the expected total path length of a randomly built binary search tree with n nodes. Show that $E(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(E(i) + E(n-i-1))$.

Hint: It may help to think about the n keys being {1, ..., n}, and then expanding out an expectation as a sum over all the different possible values for the root $r(T)$.

Solution

Let T be a randomly built binary search tree with n nodes. Without loss of generality, we assume the n keys are {1, ..., n}. By definition, $E(n) = \mathbb{E}_{\mathbb{T}}[P(T)]$. Then, $E(n) = \mathbb{E}_{\mathbb{T}}[P(T_L) + P(T_R) + n - 1] = n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)]$, where T_L and T_R are the left and right subtrees of T, respectively. Note

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n} \mathbb{E}_{\mathbb{T}}[P(T_L)|r(T) = i] \cdot Pr(r(T) = i).$$

Since each element is equally likely to be the root of T, $\Pr(r(T) = i) = \frac{1}{n}$ for all i. Conditioned on the event that element i is the root, T_L is a randomly built binary search tree on the first $i - 1$ elements. To see this, assume we picked element i to be the root. From the point of view of the left subtree, elements $1, \ldots, i - 1$ are inserted into the subtree in a random order, since these elements are inserted into T in a random order and subsequently go into T_L in the same relative order. Hence, $\mathbb{E}_{\mathbb{T}}[P(T_L)|r(T) = i] = E(i - 1)$. Putting these together, we get

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n} \frac{1}{n} E(i-1).$$

Similarly, we get $\mathbb{E}_{\mathbb{T}}[P(T_R)] = \sum_{i=1}^{n} \frac{1}{n} E(n-i)$. Then,

$$\begin{aligned} E(n) &= n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)] \\ &= n - 1 + \frac{1}{n}\sum_{i=1}^{n}[E(i-1) + E(n-i)] \\ &= n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}[E(i) + E(n-i-1)], \end{aligned}$$

where we changed the indexing of the sumamtion in the last equality.

3. Show that $E(n) = O(n \log n)$. You may cite a result previously proven in the context of other topics covered in class.

   Hint: It may help to use $E(n) = n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)]$

   Solution

   This is the same recurrence that appears in the analysis of Quicksort.

4. Design a sorting algorithm based on randomly building a binary search tree. Show that its (expected) running time is $O(n \log n)$. Assume that a random permutation of n keys can be generated in time $O(n)$.

   Solution

   The algorithm is 1) construct a randomly built binary search tree T by inserting given elements in a random order; and 2) do the inorder traversal on T to get a sorted list. Note step 2 can be done in $O(n)$ time. We argue that step 1 takes $O(n \log n)$ time in expectation. We observe that given the final state of tree T, we can compute the amount of work spent to construct T. To insert a node x at depth d, we traversed exactly the path from the root to the parent of x, at depth $d-1$, to insert it. Hence, we can upper bound the total work done to construct T by $O(P(T))$. From part (c), we know that $P(T) = O(n \log n)$ in expectation. It follows that Step 1 takes $O(n \log n)$ time in expectation. Overall, the algorithm runs in $O(n \log n)$ in expectation.

3 More Sorting!

We are given an unsorted array A with n numbers between 1 and M where M is a large but constant positive integer. We want to find if there exist two elements of the array that are within T of one another.

1. Design a simple algorithm that solves this in $O(n^2)$.

2. Design a simple algorithm that solves this in $O(nlogn)$.
3. How could you solve this in $O(n)$? (Hint: modify bucket sort.)

Solution

1. Compare all pairs of numbers to see if any are within T of each other.
2. Sort the array, then compare only adjacent elements to see if they are within T of each other. (After sorting, if any two elements are within T, some adjacent pair must be within T.)
3. Use buckets of size T: bucket i holds elements in the range $[iT, (i+1)T)$. Since M is constant, we have $O(M/T)$ buckets, and placing all n elements takes $O(n)$ time.
   If any bucket contains ≥ 2 elements, they are within T of each other, so return true. Otherwise, each bucket has at most one element. Elements in buckets that differ by 2 or more are more than T apart, so we only need to check elements in adjacent buckets. There are at most $n-1$ such adjacent pairs to check, so this is $O(n)$.

4 Finding Min and Max: A Comparison Lower Bound

In the **comparison model**, an algorithm can only access the input elements through pairwise comparisons. A **comparison** takes two elements a and b and asks "is $a < b$?", receiving a yes/no answer. Given a list of n *distinct* values, we want to find both the minimum and maximum elements.

1. Show that $n-1$ comparisons are necessary and sufficient to find just the minimum of n distinct elements.

   Solution

   Sufficiency: We can find the minimum in $n-1$ comparisons as follows: compare elements 1 and 2, keep the smaller one as the "current minimum." Then compare the current minimum with element 3, updating if element 3 is smaller. Continue through all n elements. This uses exactly $n-1$ comparisons.
   Necessity: Consider the "potential minimum" status of each element. Initially, all n elements are potential minimums (none have lost a comparison). At the end, exactly one element can be identified as the minimum. Each comparison eliminates at most one element from being a potential minimum (the larger element in the comparison). Therefore, to eliminate $n-1$ elements from contention, we need at least $n-1$ comparisons.

2. Describe a simple algorithm that finds both the minimum and maximum using at most $2n-3$ comparisons.

Solution

First find the minimum using $n-1$ comparisons (as in part (a)). Then, among the remaining $n-1$ elements, find the maximum using $n-2$ comparisons. Total: $(n-1)+(n-2) = 2n-3$ comparisons.

3. Describe an algorithm that finds both the minimum and maximum using at most $\lceil 3n/2 \rceil - 2$ comparisons.

Solution

Algorithm:

(a) **Pairing phase:** Group the elements into $\lfloor n/2 \rfloor$ pairs. For each pair, compare them. This uses $\lfloor n/2 \rfloor$ comparisons. After this phase:
- The "small" elements (the smaller element from each pair) form a set S of size $\lfloor n/2 \rfloor$.
- The "large" elements (the larger element from each pair) form a set L of size $\lfloor n/2 \rfloor$.
- If n is odd, one unpaired element goes into both S and L.

(b) **Find minimum:** The global minimum must be in S (it cannot have been the larger element in any comparison). Scan through S to find its minimum using $|S|-1$ comparisons.

(c) **Find maximum:** The global maximum must be in L (it cannot have been the smaller element in any comparison). Scan through L to find its maximum using $|L|-1$ comparisons.

Comparison count:
- If n is even: $|S| = |L| = n/2$. Total comparisons:
$$\frac{n}{2} + \left(\frac{n}{2}-1\right) + \left(\frac{n}{2}-1\right) = \frac{3n}{2} - 2.$$
- If n is odd: $|S| = |L| = \lceil n/2 \rceil = (n+1)/2$. Pairing uses $(n-1)/2$ comparisons. Total:
$$\frac{n-1}{2} + \left(\frac{n+1}{2}-1\right) + \left(\frac{n+1}{2}-1\right) = \frac{3(n-1)}{2} = \left\lceil \frac{3n}{2} \right\rceil - 2.$$

In both cases, the algorithm uses exactly $\lceil 3n/2 \rceil - 2$ comparisons.

4. Prove that any comparison-based algorithm that finds both the minimum and maximum of n distinct elements must use at least $\lceil 3n/2 \rceil - 2$ comparisons in the worst case.

Solution

We use an **adversary argument** based on tracking the information state.
Status tracking: For each element x, define:

- x is a **potential minimum** if x has never lost a comparison (never been shown to be larger than another element).
- x is a **potential maximum** if x has never won a comparison (never been shown to be smaller than another element).

Initial state: All n elements are both potential minimums and potential maximums.

Final state requirement: For the algorithm to correctly identify the min and max, at termination:

- Exactly 1 element can be a potential minimum (the true minimum).
- Exactly 1 element can be a potential maximum (the true maximum).

Thus, the algorithm must eliminate:

- $n - 1$ elements from being potential minimums.
- $n - 1$ elements from being potential maximums.

Define the **status count** as the total number of "potential minimum" and "potential maximum" statuses remaining. Initially, the status count is $2n$. At termination, it must be exactly 2.

Effect of comparisons: Consider a comparison between elements x and y. Let the outcome be $x < y$.

- x loses: if x was a potential maximum, it is no longer (its potential-max status is eliminated).
- y wins: if y was a potential minimum, it is no longer (its potential-min status is eliminated).

Key observation: A comparison can eliminate at most 2 statuses, and it eliminates 2 statuses *only if* both elements involved had never been compared before (both are "fresh", meaning x was still a potential max AND y was still a potential min, or vice versa). If at least one element has already lost or won a comparison, the comparison eliminates at most 1 status.

Counting argument: Let k be the total number of comparisons. Partition these into:

- f = number of comparisons where both elements are fresh (neither has been compared before).
- $k - f$ = number of comparisons where at least one element is not fresh.

Each comparison of the first type eliminates at most 2 statuses. Each comparison of the second type eliminates at most 1 status. Therefore:

$$\text{Total statuses eliminated} \leq 2f + (k - f) = f + k.$$

We need to eliminate $2n - 2$ statuses, so:

$$2n - 2 \leq f + k.$$

Bounding f: How many fresh-fresh comparisons can there be? Initially, n elements are fresh. Each fresh-fresh comparison involves 2 fresh elements and makes both of them non-fresh. Therefore:

$$f \leq \left\lfloor \frac{n}{2} \right\rfloor.$$

Final bound: Substituting:

$$2n - 2 \leq \left\lfloor \frac{n}{2} \right\rfloor + k,$$

$$k \geq 2n - 2 - \left\lfloor \frac{n}{2} \right\rfloor.$$

Case n **even:** $\lfloor n/2 \rfloor = n/2$, so:

$$k \geq 2n - 2 - \frac{n}{2} = \frac{3n}{2} - 2.$$

Case n **odd:** $\lfloor n/2 \rfloor = (n-1)/2$, so:

$$k \geq 2n - 2 - \frac{n-1}{2} = \frac{4n - 4 - n + 1}{2} = \frac{3n-3}{2} = \frac{3(n-1)}{2}.$$

In both cases: $k \geq \lceil 3n/2 \rceil - 2$.
Conclusion: Any comparison-based algorithm requires at least $\lceil 3n/2 \rceil - 2$ comparisons in the worst case.

5. Conclude that the exact worst-case complexity of finding both min and max is $\lceil 3n/2 \rceil - 2$ comparisons.

Solution

From part (c), we have an algorithm that uses exactly $\lceil 3n/2 \rceil - 2$ comparisons. From part (d), we have proven that any algorithm requires at least $\lceil 3n/2 \rceil - 2$ comparisons in the worst case.
Therefore, the **exact worst-case comparison complexity** of finding both the minimum and maximum of n distinct elements is:

$$\lceil \frac{3n}{2} \rceil - 2$$

This is a rare and beautiful result in algorithm analysis: we have determined the *exact* number of comparisons needed, not just the asymptotic complexity. The pair-and-compare algorithm is **optimal**.