שאלות נכון \ לא נכון

(1)

Under the simple uniform hashing assumption, the probability that three specific data elements (say 1, 2 and 3) hash to the same slot (i.e., $h(1) = h(2) = h(3)$) is $1/m^3$, where m is a number of buckets.

Solution: False. The above formula only describes the probability of collision in a *fixed* bucket (say bucket number 1). The correct answer is $1/m^2$.

(2)

Given an array of n integers, each belonging to $\{-1, 0, 1\}$, we can sort the array in $O(n)$ time in the worst case.

Solution: True. Use counting sort, e.g., after adding 1 to all numbers.

(3)

The following array is a max heap: $[10, 3, 5, 1, 4, 2]$.

Solution: False. The element 3 is smaller than its child 4, violating the max-heap property.

(4)

RADIX SORT does not work correctly (i.e., does not produce the correct output) if we sort each individual digit using INSERTION SORT instead of COUNTING SORT.

Solution: False. INSERTION SORT (as presented in class) is a stable sort, so RADIX SORT remains correct. The change can worsen running time, though.

(5)

Could a binary search tree be built using $o(n \lg n)$ comparisons in the comparison model? Explain why or why not.

Solution: No, or else we could sort in $o(n \lg n)$ time by building a BST in $o(n \lg n)$ time and then doing an in-order tree walk in $O(n)$ time.

(6)

A hash table guarantees constant lookup time.
Explain:

Solution: **False.** It only has *expected* constant lookup time; if $\Theta(n)$ elements collide, then lookup may take $\Theta(n)$ time in the worst case (assuming chaining).

(7)

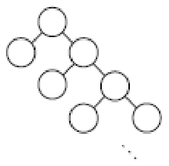
A non-uniform hash function is expected to produce worse performance for a hash table than a uniform hash function.
Explain:

Solution: **True.** A non-uniform hash function is more likely to result in collisions, which leads to slower lookup times.

(8)

If every node in a binary tree has either 0 or 2 children, then the height of the tree is $\Theta(\lg n)$.
Explain:

Solution: **False.** One counterexample is a tree like the one shown here, extending down and to the right. It has $\Theta(n)$ height.



(9)

A heap A has each key randomly increased or decreased by 1. The random choices are independent. We can restore the heap property on A in linear time.
Explain:

Solution: **True.** Simply call BUILD-HEAP at the root, which runs in $\Theta(n)$ time.

(10)

If you know the numbers stored in a BST and you know the structure of the tree, you can determine the value stored in each node.

Solution: **True.** You can do an inorder walk of the tree, which would order the nodes from smallest key to largest key. You can then match them with the values.

(11)

In max-heaps, the operations insert, max-heapify, find-max, and find-min all take $O(\log n)$ time.

Solution: **False.** The minimum can be any of the nodes without children. There are $n/2$ such nodes, so it would take $\Theta(n)$ time to find it in the worst case.

(12)

When you double the size of a hash table, you can keep using the same hash function.

Solution: **False.** If you double the size of the table, you need to change the hash function so that it maps keys to {0, 2m-1} instead of mapping them to {0, m-1}. However, some people answered True, but explained that this would be inefficient. This answer also received full credit.

(13)

We can sort 7 numbers with 10 comparisons.

Solution: **False.** To sort 7 numbers, the binary tree must have $7! = 5040$ leaves. The number of leaves of a complete binary tree of height 10 is $2^{10} = 1024$. This is not enough.

(14)

Merge sort can be implemented to be stable.

Solution: **True.** Whether it is stable or not depends on which element is chosen next in case there is a tie during the merge step. If the element from the left list (the list of elements that came earlier in the original array) is always chosen, then the merge sort is stable.

(15)

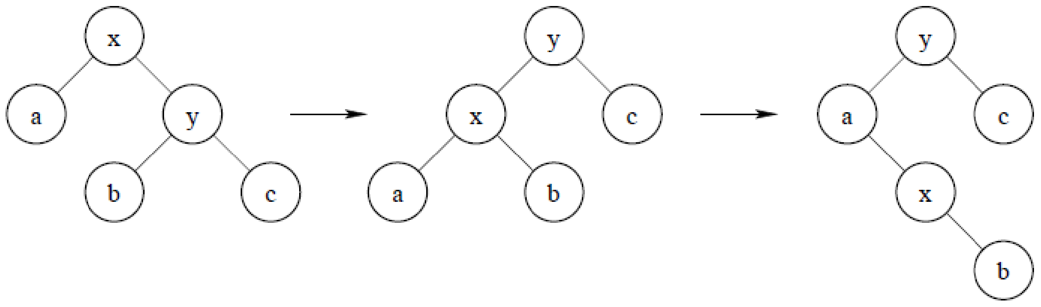
A $\Theta(n^2)$ algorithm always takes longer to run than a $\Theta(\log n)$ algorithm.

Solution: **False.** The constant of the $\Theta(\log n)$ algorithm could be a lot higher than the constant of the $\Theta(n^2)$ algorithm, so for small n, the $\Theta(\log n)$ algorithm could take longer to run.

(16)

Performing a left rotation on a node and then a right rotation on the *same* node will not change the underlying tree structure.
Explain:

Solution: False. The figure below shows what happens in this case. To undo the left rotation on x, we must do a right rotation on y.



(17)

While inserting an element into a BST, we will pass the element's predecessor and successor (if they exist).
Explain:

Solution: True. The predecessor of a node is either the maximum element of its left subtree, or one of its ancestors. A newly-inserted node has no descendants, so the predecessor must be one of its ancestors, and hence the predecessor is on the path that was traversed during the insertion procedure. (A similar argument holds for the successor.)

(18)

For a hash table using open addressing, if we maintain $m = \Theta(n)$, then we can expect a good search and insert runtime.
Explain:

Solution: False. If the hash table is nearly full, for example if $n = m - 1$, then the runtimes will take $\frac{1}{1-\alpha} = O(n)$.

(19)

Given two heaps with n elements each, it is possible to construct a single heap comprising all $2n$ elements in $O(n)$ time.

Solution: TRUE. Simply traverse each heap and read off all $2n$ elements into a new array. Then, make the array into a heap in $O(n)$ time by calling MAX-HEAPIFY for $i = 2n$ down to 1.

(20)

Building a heap with n elements can always be done in $O(n \log n)$ time.

Solution: TRUE. In fact, we can build a heap in $O(n)$ time by putting the elements in an array and then calling MAX-HEAPIFY for $i = n$ down to 1.

(21)

Given a hash table of size n with n elements, using chaining, the minimum element can always be found in $O(1)$ time.

Solution: FALSE. We will need to scan the entire table; this takes $\Omega(n)$ time.

(22)

Running merge sort on an array of size n which is already correctly sorted takes $O(n)$ time.

Solution: FALSE. The merge sort algorithm presented in class always divides and merges the array $O(\log n)$ times, so the running time is always $O(n \log n)$.

(23)

We can always find the maximum in a min-heap in $O(\log n)$ time.

Solution: FALSE. The *maximum* element in a min-heap can be anywhere in the bottom level of the heap. There are up to $n/2$ elements in the bottom level, so finding the maximum can take up to $O(n)$ time.

(24)

In a heap of depth d, there must be at least 2^d elements. (Assume the depth of the first element (or root) is zero).

Solution: TRUE. The minimum number of elements in a heap of depth d is one more than the maximum number of elements in a heap of depth $(d-1)$. Since a level at depth d in a binary heap can have up to 2^d elements, the number of elements at depth $(d-1)$ is $\sum_{i=0}^{d-1} 2^i = 2^d - 1$. So the minimum number of elements in a heap of depth d is $(2^d - 1) + 1 = 2^d$.

(25)

Inserting an element into a binary search tree of size n always takes $O(\log n)$ time.

Solution: FALSE. Inserting an element into a binary search tree takes $O(h)$ time, where h is the height of the tree. If the tree is not balanced, h may be much larger than $\log n$ (as large as $n - 1$).

(26)

Any two (possibly unbalanced) BSTs containing n elements each can be merged into a single balanced BST in $O(n)$ time.

Solution: TRUE. Use in-order traversal of the two BSTs to create two sorted lists of length n in $O(n)$ time, merge them into a single sorted list of length $2n$ in $O(n)$ time, and then create a balanced BST from the sorted lists in $O(n)$ time.

(27)

The height of any binary search tree with n nodes is $O(\log n)$.
Explain:

Solution: False. In the best case, the height of a BST is $O(\log n)$ if it is balanced. In the worst case, however, it can be $\Theta(n)$.

(28)

The depths of any two leaves in a max heap differ by at most 1.
Explain:

Solution: True. A heap is derived from an array and new levels to a heap are only added once the leaf level is already full. As a result, a heap's leaves are only found in the bottom two levels of the heap and thus the maximum difference between any two leaves' depths is 1.

A common mistake was pointing out that a heap could be arbitrarily shaped as long as the heap property (parent greater than its children in the case of a max-heap) was maintained. This heap is not a valid heap, as there would be gaps if we tried to express it in array form, heap operations would no longer have $O(\log n)$ running time, and heap sort would fail when using this heap.

(29)

For any constants $x, y > 1$, we have $n^x = O(y^n)$.
Explain:

Solution: True. Exponential growth always dominates polynomial growth. To show more rigorously, we want to show that $\frac{n^x}{y^n} = 0$ as n goes to infinity. For large enough n, we have

$$0 < \frac{\log n}{n} < \frac{\log y}{x+1}$$
$$(x+1)\log n < n \log y$$
$$n^{x+1} < y^x$$
$$\frac{n^x}{y^n} < \frac{1}{n}$$

Since $\frac{1}{n} = 0$ as n goes to infinity, this shows that $\frac{n^x}{y^n} = 0$ for the same limit and thus $n^x = O(y^n)$. This proof was not necessary for full credit.

(30)

Instead of using counting sort to sort digits in the radix sort algorithm, we can use any valid sorting algorithm and radix sort will still sort correctly.

Solution: False. Need stable sort.

(31)

A set of n integers whose values are in the range $[0, n^8)$ can be sorted in $O(n)$ time.
Explain:

Solution: True. Use radix sort with a radix of size n. Then each invocation of counting sort takes $O(n + n) = O(n)$ time. Each element has 8 "digits", so the total time for radix sort is $O(8n) = O(n)$.

(32)

There exists a comparison sort of 5 numbers that uses at most 6 comparisons in the worst case.
True **False**
Explain:

Solution: **False.** The number of leaves of a decision tree which sorts 5 numbers is 5! and the height of the tree is at least $\lg(5!)$. Since $5! = 120$, $2^6 = 64$, and $2^7 = 128$, we have $6 < \lg(5!) < 7$. Thus at least 7 comparisons are required.

(33)

Heapsort can be used as the auxiliary sorting routine in radix sort, because it operates in place.
True **False**
Explain:

Solution: **False.** The auxiliary sorting routine in radix sort needs to be stable, meaning that numbers with the same value appear in the output array in the same order as they do appear in the input array. Heapsort is not stable. It does operate in place, meaning that only a constant number of elements of the input array are ever stored outside the array.

(34)

The sequence $\langle 20, 15, 18, 7, 9, 5, 12, 3, 6, 2 \rangle$ is a max-heap.
True **False**
Explain:

Solution: **True.** For every node with 1-based index $i > 1$, the node with index $\lfloor \frac{i}{2} \rfloor$ is larger.

(35)

To sort n integers in the range from 1 to n^2, a good implementation of radix sort is asymptotically faster in the worst case than any comparison sort.
True **False**
Explain:

Solution: **True.** Split each number into two digits, each between 0 and n. Then make two passes of counting sort, each taking $\theta(n)$ time. Any comparison sort will take $\Omega(n \log n)$ time.

(36)

Call an ordered pair (x_1, y_1) of numbers ***lexically less than*** an ordered pair (x_2, y_2) if either (i) $x_1 < x_2$ or (ii) $x_1 = x_2$ and $y_1 < y_2$. Then a set of ordered pairs can be sorted lexically by two passes of a sorting algorithm that only compares individual numbers.
True **False**
Explain:

Solution: **True.** The idea is similar to RADIX-SORT though it is not exactly the same. First we sort on the y_i digit of each pair, (x_i, y_i). Next we sort on the x_i digit using a stable sort.

(37)

Any in-place sorting algorithm can be used as the auxiliary sort in radix sort.

True **False**

Explain:

Solution: **False.** The auxiliary algorithm has to be *stable*. In-place means the algorithm only uses constant additional memory.

(38)

Every sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case to sort n elements.

True **False**

Explain:

Solution: **False.** Counting sort, radix sort, and bucket sort all sort in fewer than $O(n \lg n)$ comparisons.
The lower bound only applies to comparison sorts.

(39)

The running time of Radix sort is effectively independent of whether the input is already sorted.
Explain:

Solution: **True.** All input orderings give the worst-case running time; the running time doesn't depend on the order of the inputs in any significant way.

(40)

There exists an algorithm to build a binary search tree from an unsorted list in $O(n)$ time.
Explain:

Solution: False. Because an in-order walk can create a sorted list from a BST in $O(n)$ time, the existence of such an algorithm would violate the $\Omega(n \lg n)$ lower bound on comparison-based sorts.

(41)

There exists an algorithm to build a binary heap from an unsorted list in $O(n)$ time.
Explain:

Solution: True. The standard BUILD-HEAP algorithm runs in $O(n)$ time.

(42)

In the worst case, merge sort runs in $O(n^2)$ time.
Explain:

Solution: True. Merge sort runs in $O(n \lg n)$ time which is $O(n^2)$.

(43)

If the load factor of a hash table is less than 1, then there are no collisions.
Explain:

Solution: **False.** The table contains fewer elements than it has slots, but that doesn't prevent elements from hashing to the same slot.

(44)

Using hashing, we can create a sorting algorithm similar to COUNTING-SORT that sorts a set of n (unrestricted) integers in linear time. The algorithm works the same as COUNTING-SORT, except it uses the hash of each integer as the index into the counting sort table.
Explain:

Solution: **False.** Counting sort requires that the elements in its table are ordered according their indices into the table. Hashing the integers breaks this property

(45)

There exists a comparison-based algorithm to construct a BST from an unordered list of n elements in $O(n)$ time.
Explain:

Solution: **False.** Because we can create a sorted list from a BST in $O(n)$ time via an in-order walk, this would violate the $\Omega(n \lg n)$ lower bound on comparison-based sorting.

(46)

A max-heap can support both the INCREASE-KEY and DECREASE-KEY operations in $\Theta(\lg n)$ time.
Explain:

Solution: True.
INCREASE-KEY: Change the node's key, then swap it with its parent until the heap invariant is restored.
DECREASE-KEY: Change the node's key, then MAX-HEAPIFY on the changed node.

(47)

For all asymptotically positive $f(n)$, $f(n) + o(f(n)) = \Theta(f(n))$.

Solution: True. Clearly, $f(n) + o(f(n))$ is $\Omega(f(n))$. Let $g(n) \in o(f(n))$. For any $c > 0$, $g(n) \leq c(f(n))$ for all $n \geq n_0$ for some n_0. Hence, $g(n) = O(f(n))$, whence $f(n) + o(f(n)) = O(f(n))$. Thus, $f(n) + o(f(n)) = \Theta(f(n))$.

(48)

The worst-case running time and expected running time are equal to within constant factors for any randomized algorithm.

Solution: False. Randomized quicksort has worst-case running time of $\Theta(n^2)$ and expected running time of $\Theta(n \lg n)$.

(49)

There exists a data structure to maintain a dynamic set with operations Insert(x,S), Delete(x,S), and Member?(x,S) that has an expected running time of $O(1)$ per operation.

Answer: `True. Use a hash table.`

(50)

The array

$$20 \quad 15 \quad 18 \quad 7 \quad 9 \quad 5 \quad 12 \quad 3 \quad 6 \quad 2$$

forms a max-heap.

Solution: **True.**

- $A[1] = 20$ has children $A[2] = 15 \le 20$ and $A[3] = 18 \le 20$.
- $A[2] = 15$ has children $A[4] = 7 \le 15$ and $A[5] = 9 \le 15$.
- $A[3] = 18$ has children $A[6] = 5 \le 18$ and $A[7] = 12 \le 18$.
- $A[4] = 7$ has children $A[8] = 3 \le 7$ and $A[9] = 6 \le 7$.
- $A[5] = 9$ has child $A[10] = 2$.
- $A[6], \ldots, A[10]$ have no children.

(51)

Suppose that a hash table with collisions resolved by chaining contains n items and has a load factor of $\alpha = 1/\lg n$. Assuming simple uniform hashing, the expected time to search for an item in the table is $O(1/\lg n)$.

Solution: **False.** The expected time to search for an item in the table is $O(1 + \alpha) = O(1 + 1/\lg n) = O(1)$. At least a constant running time $O(1)$ is needed to search for an item; subconstant running time $O(1/\lg n)$ is not possible.

(52)

Let A_1, A_2, and A_3 be three sorted arrays of n real numbers (all distinct). In the comparison model, constructing a balanced binary search tree of the set $A_1 \cup A_2 \cup A_3$ requires $\Omega(n \lg n)$ time.

Solution: **False.** First, merge the three arrays, A_1, A_2, and A_3 in $O(n)$ time. Second, construct a balanced binary search tree from the merged array: the median of the array is the root; recursively build the left subtree from the first half of the array and the right subtree from the second half of the array. The resulting running time is $T(n) = 2T(n/2) + O(1) = O(n)$.

(53)

Given an unsorted array $A[1 \,.\,.\, n]$ of n integers, building a max-heap out of the elements of A can be performed asymptotically faster than building a red-black tree out of the elements of A.

Solution: **True.** Building a heap takes $O(n)$ time, as described in CLRS. On the other hand, building a red-black tree takes $\Omega(n \lg n)$ time, since it is possible to produce a sorted list of elements from a red-black tree in $O(n)$ time by doing an in-order tree walk (and sorting requires $\Omega(n \lg n)$ time in a comparison model).

(54)

The hash function $h(k) = k \mod m$ is guaranteed to have no collisions if and only if m is a prime number.

Solution: False. If keys are larger than m then collisions could well occur.

(55)

When creating a new heap out of an array A, we do not need to call MAX-HEAPIFY on the last $\lceil n/2 \rceil$ elements, i.e., $A[\lceil n/2 \rceil : n]$.

Solution: True

(56)

Suppose you have designed a hash table with load factor $\alpha = 0.25$. Then, open addressing with linear probing would ensure that a search operation would take expected constant time.

Solution: False. Cluster sizes can be as large as $\Theta(\log n)$.

(57)

Suppose you have an array $A[1..n]$ of n elements in arbitrary order. Does the following alternate implementation of build-max-heap work? In other words, does it correctly build a max heap from the given elements $A[1..n]$? Why or why not?

```
build-max-heap($A$):
    for $i$ from 1 to $n/2$:
        max-heapify($A, i$)
```

This algorithm calls heapify starting at the root and working its way down the tree, instead of the other way around.

Solution: No. Starting with $[1, 2, 3, 4]$ we would get $[2, 4, 3, 1]$, which is not a max-heap.