COMP 2402 - Study Session Questions & Answers

April 18, 2026

Disclaimer

- The material presented in these slides and during the study session will **not** influence what is on the assessment in one way or another — this is not the definitive content of the assessment.
- This is **not** a comprehensive study guide — it is intended to complement your studies.
- Please review the content from the professor and the textbook as well.
- Some of the answers might be incorrect / mistakes might exist=

Contents

Multiple Choice Questions

1 ArrayStack Growth Trigger (with Example)

Question

If we are currently about to *grow* the backing array a, what can you say about the number of `add()` and `remove()` operations (as a function of the current value of n) since the last time the `ArrayStack` was resized?

Choices

(a) At least $n/2$ `add()` operations have occurred since then

(b) At least $2n/3$ `add()` operations have occurred since then

(c) At least $2n/3$ `remove()` operations have occurred since then

(d) We cannot bound either the number of `add()` nor `remove()` operations.

(e) At least $n/2$ `remove()` operations have occurred since then

Explanation

Answer: (a) At least $n/2$ `add()` operations.

When we grow the array, it means the array is completely full. Let n be the current number of elements. Right after the last resize, the array had extra space — about half of it was empty. It takes roughly $n/2$ `add()` operations to fill that remaining space and trigger the next resize.

Example: Suppose the array capacity was 5 before the last resize. When we grew it, we doubled the capacity to 10. At that point, there were 5 elements in the array, leaving 5 empty spots. As we continue adding items:

$$\underbrace{[1, 2, 3, 4, 5, \_, \_, \_, \_, \_]}_{\text{after resize, half full}}$$

Each `add()` fills one of the empty slots. After 5 more adds:

$$\underbrace{[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}_{\text{full, trigger next growth}}$$

We see that $n/2 = 5$ `add()` operations were required to go from half full to full, causing the next growth.

Thus, at the moment of growing, at least $n/2$ `add()` operations have occurred since the last resize. We cannot say anything about `remove()` operations, because the array only grows when it becomes full.

2 ArrayStack Shrink Trigger (with Example)

Question

Recall that we shrink the backing array a when $3n \leq a$.length. If we are currently about to shrink the backing array a, what can you say about the number of `add()` and `remove()` operations since the last time the `ArrayStack` was resized?

Choices

(a) At least $n/2 - 1$ `add()` operations have occurred since then

(b) At least $2n/3$ `remove()` operations have occurred since then

(c) At least $2n/3$ `add()` operations have occurred since then

(d) At least $n/2 - 1$ `remove()` operations have occurred since then

(e) We cannot bound either the number of `add()` nor `remove()` operations.

Explanation

Answer: (d) At least $n/2 - 1$ `remove()` operations.

After any resize, the array is set so that its capacity is about twice the number of elements present, so it is roughly half full. Let the last-resize state be size n_0 and capacity L, so $n_0 \approx L/2$. We are about to shrink when $3n \leq L$, i.e., when the current size n has fallen to about $L/3$. The number of elements removed since the last resize is therefore at least

$$n_0 - n \;\approx\; \frac{L}{2} - \frac{L}{3} \;=\; \frac{L}{6}.$$

Because $3n \leq L$, we have $L \geq 3n$, hence

$$n_0 - n \;\geq\; \frac{3n}{6} \;=\; \frac{n}{2}.$$

Accounting for integer rounding and the fact we are at the *threshold* ($3n \leq L$ may hold by 1), the clean lower bound is $n/2 - 1$ removes.

Base-case picture: Suppose after the last resize the capacity became $L = 12$ and the size was $n_0 = 6$ (half full):

$$\underbrace{[1, 2, 3, 4, 5, 6, \_, \_, \_, \_, \_, \_]}_{\text{right after resize: } n_0=6,\ L=12}$$

We shrink when $3n \leq 12$, i.e., when $n \leq 4$. To reach $n = 4$ from $n_0 = 6$ requires $6 - 4 = 2$ `remove()`s. At that moment $n = 4$, and the bound $n/2 - 1 = 4/2 - 1 = 1$ is satisfied (indeed we removed $2 \geq 1$). In bigger arrays the gap widens, and the general lower bound becomes $n/2 - 1$.

3 ArrayStack Growth and Shrink Behaviour

Question

An `ArrayStack` stores n elements in an array a and:

- doubles the size of a when $n = a.\text{length}$, and
- halves the size of a when $3n \leq a.\text{length}$.

Right now, $n = 40$ and $a.\text{length} = 60$. You perform a long sequence of `add(x)` and `remove()` operations, but the size n never goes above 60 and never below 10.

Which of the following is the *best* upper bound on the total number of times the backing array can be resized during this sequence?

Choices

(a) $O(1)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n)$

(e) $O(n \log n)$

Explanation

The correct answer is **(d)**.

Every time we resize, we at least double or halve the array length. In the given range, $a.\text{length}$ can only take a small set of values (like $16, 32$, etc.), but the sequence of operations might keep moving n back and forth across the thresholds that trigger growth and shrink.

The key idea from amortized analysis in the textbook is:

- Each resize costs $O(a.\text{length})$, but
- There must be $\Omega(a.\text{length})$ pushes or pops between two consecutive resizes.

So the *amortized* cost per operation is $O(1)$, but over m operations we can still have up to $O(m)$ total work and therefore up to $O(m)$ resizes in the worst case.

Since the problem does not limit how many operations you do, only the range of n, the best safe upper bound on the number of resizes is *linear in the number of operations*, which in terms of n is written as $O(n)$.

Heuristic exam shortcut: how to eyeball this quickly

On an exam, you can get to **(d)** without doing full amortized analysis by using two quick sanity checks:

- There is *no* bound on how many operations you do; you can keep hovering around the grow/shrink thresholds forever. That already rules out tiny bounds like $O(1)$ or $O(\log n)$ or $O(\sqrt{n})$ — you could force many resizes by running a very long sequence of operations.

- Each operation can cause *at most one* resize (you can't grow or shrink twice in one `add` or `remove`). So if you do m operations, the number of resizes is at most m, i.e. *linear in the number of operations*. Among the options, the smallest "linear-ish" bound is $O(n)$.

So the fast way to think about it in exam conditions is:

"No global limit on operations" $\Rightarrow$ can have many resizes, but at most one per op $\Rightarrow$ linear bound $\Rightarrow O(n)$.

4 Circular Array for ArrayQueue and ArrayDeque

Question

An `ArrayQueue` uses a circular array of length a.length, with:

$$\texttt{j} = \text{index of the front element}, \quad n = \text{number of elements.}$$

The k-th element in the queue (where $0 \le k < n$) is stored at index $(\texttt{j} + k)$ mod a.length.

Suppose a.length $= 8$, $\texttt{j} = 6$, and $n = 5$. You now convert this `ArrayQueue` into an `ArrayDeque` by treating the same array as a deque with the *same* `j` and n. Which of the following statements is *always* true about how an `addFirst(x)` operation will behave in this situation (ignoring resizing)?

Choices

(a) It must shift all n elements one step to the right in the array.

(b) It can always insert at index $(\texttt{j} - 1)$ mod 8 in $O(1)$ time.

(c) It must move exactly the element at index $(\texttt{j} + n - 1)$ mod 8 to make room.

(d) It may insert at either end of the logical sequence, but the runtime is $\Theta(n)$.

(e) It can insert at index $(\texttt{j} - 1)$ mod 8 in $O(1)$ time *only if* that cell is currently empty.

Explanation

The correct answer is **(e)**.

In an `ArrayDeque`, we keep the n elements in a *contiguous block* of cells (wrapping around mod a.length), starting at `j`:

$$\text{occupied indices: } \texttt{j},\ \texttt{j}+1,\ \ldots,\ \texttt{j}+n-1 \pmod{a.\text{length}}.$$

For `addFirst(x)`:

- The "ideal" spot is the cell just before the front:

$$\text{target index} = (\texttt{j} - 1) \bmod a.\text{length}.$$

- If that cell is *empty*, we can just write x there and decrement `j`. This is a single write and a single index update $\Rightarrow O(1)$ time.
- If that cell is *not* empty (because the occupied block already touches it), then simply writing there would break the "contiguous block" invariant. In that case, the implementation may choose to shift a bunch of elements to make room, which can cost $\Theta(n)$ in the worst case.

So:

- It is *not* true that `addFirst` is always $O(1)$ (so (b) is too optimistic).
- It is *also* not true that we must always shift everything or always be $\Theta(n)$ (so (a) and (d) are too pessimistic).

The only statement that is always correct is that we get $O(1)$ time exactly in the "easy" case when the cell at $(\texttt{j} - 1)$ mod 8 is empty — which is exactly what option **(e)** says.

5 DualArrayDeque with Two ArrayStacks

Question

Recall that a `DualArrayDeque` implements the `List` interface using two `ArrayStacks`:

```
public class DualArrayDeque<T> extends AbstractList<T> {
  ArrayStack<T> front;   // front: stack whose TOP is the left end of the deque
  ArrayStack<T> back;    // back : stack whose TOP is the right end of the deque
  ...
}
```

Suppose our deque is $[A, P, G, T, Y]$ where `front` $= [A, P]$ (with A at the *top*) and `back` $= [G, T, Y]$ (with Y at the *top*). We denote the split with $||$, i.e., initially

$$[A, P \,||\, G, T, Y].$$

What do `front` and `back` look like after the following operations?

`Deque.remove(0);  Deque.add(2, E);  Deque.add(1, L).`

Choices

(a) $[P, L, \,||\, G, E, T, Y]$

(b) $[A, L, G \,||\, E, T, Y]$

(c) $[A, L \,||\, G, E, T, Y]$

(d) $[P, L, G \,||\, E, T, Y]$

Explanation

Answer: (d) $[P, L, G \,||\, E, T, Y]$**.**

Picture trace (indexing is 0-based):

Start: $[A, P \,||\, G, T, Y]$
`remove(0)` : delete $A \Rightarrow [P \,||\, G, T, Y] \;=\; [P, G, T, Y]$
`add(2, E)` : insert E at position $2 \Rightarrow [P, G, E, T, Y]$
`add(1, L)` : insert L at position $1 \Rightarrow [P, L, G, E, T, Y]$.

After the final operation the deque has $n = 6$ items. `DualArrayDeque` rebalances so that `front` and `back` hold about half each. Thus we split after 3 items:

$$\texttt{front} = [P, L, G], \qquad \texttt{back} = [E, T, Y],$$

i.e.,

$$[P, L, G \,||\, E, T, Y].$$

6 DualArrayDeque Balancing Cost

Question

A `DualArrayDeque` stores a list using two `ArrayStack`s: `front` and `back`. The `balance()` method is called whenever

$$3 \cdot \texttt{front.size()} < \texttt{back.size()} \quad \text{or} \quad 3 \cdot \texttt{back.size()} < \texttt{front.size()}.$$

During `balance()`, all n elements are copied into new arrays so that the sizes of `front` and `back` differ by at most 1.

Suppose you perform m operations (each one is a `get`, `set`, `add`, or `remove`) starting from an empty `DualArrayDeque`. Which of the following is the tightest asymptotic bound on the *total* time spent inside `balance()` over all m operations?

Choices

(a) $O(1)$

(b) $O(\log m)$

(c) $O(m^{1/2})$

(d) $O(m)$

(e) $O(m \log m)$

Explanation

The correct answer is **(d)**.

Each call to `balance()` costs $O(n)$ time, where n is the current number of elements, because it copies all elements into new arrays. However, the key observation (from the textbook) is that:

- After we rebalance, both `front` and `back` are within a factor of 3 of each other.
- It takes $\Omega(n)$ `add`/`remove` operations to make one side three times bigger than the other again.

This means that between two expensive `balance()` calls, we do at least $\Omega(n)$ cheap operations. By the standard amortized argument, the average extra cost per operation is $O(1)$, but the *total* cost over m operations is still at most $O(m)$.

So the total time spent inside `balance()` is $O(m)$, which is option (d).

7 RootishArrayStack `get(13)` Operation

Question

In a `RootishArrayStack`, a call to `get(13)` will return which element?

Choices

(a) `blocks.get(0)[13]`

(b) `blocks.get(13)[0]`

(c) `blocks.get(4)[3]`

(d) `blocks.get(3)[4]`

(e) `blocks.get(5)[4]`

Explanation

Answer: (c) `blocks.get(4)[3]`

A `RootishArrayStack` stores elements in multiple blocks where block b contains $b+1$ elements:

$$\text{Block 0: } 1, \quad \text{Block 1: } 2, \quad \text{Block 2: } 3, \quad \text{Block 3: } 4, \quad \text{Block 4: } 5, \text{ and so on.}$$

The total number of elements up through block b is the triangular number:

$$T_b = \frac{(b+1)(b+2)}{2}.$$

To find which block contains index i, we find the smallest b such that $T_b > i$.

For $i = 13$:

$$T_3 = \frac{(3+1)(3+2)}{2} = 10,$$
$$T_4 = \frac{(4+1)(4+2)}{2} = 15.$$

Since $10 \leq 13 < 15$, index 13 lies in block 4. The position within that block is the offset:

$$j = 13 - T_3 = 13 - 10 = 3.$$

Thus, the element is located at:

$$\boxed{\texttt{blocks.get(4)[3]}}.$$

Example visualization:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Block	0	1	1	2	2	2	3	3	3	3	4	4	4	4	4
Offset	0	0	1	0	1	2	0	1	2	3	0	1	2	3	4

As shown, index 13 indeed maps to `block 4, offset 3`.

8 RootishArrayStack Space Overhead

Question

A `RootishArrayStack` stores n elements in blocks of sizes $1, 2, 3, \ldots$. The blocks are full except possibly the last one.

Which of the following is the correct asymptotic bound on the *wasted* space (unused array cells) in terms of n?

Choices

(a) $\Theta(1)$

(b) $\Theta(\log n)$

(c) $\Theta(\sqrt{n})$

(d) $\Theta(n)$

(e) $\Theta(n \log n)$

Explanation

The correct answer is **(c)**.

If we have r blocks, their total capacity is

$$1 + 2 + \cdots + r = \frac{r(r+1)}{2}.$$

The data structure keeps just enough blocks so that this capacity is at least n. This means r is about $\sqrt{2n}$. More precisely, $r = \Theta(\sqrt{n})$.

Only the last block can be partially empty, so the number of unused cells is at most the size of one block, which is $r = \Theta(\sqrt{n})$. On the other hand, in the worst case the last block might be almost empty, so we can also lower bound the waste by a constant fraction of r.

Therefore, the wasted space is $\Theta(\sqrt{n})$, which is option (c).

9 LinkedList vs ArrayList: Insertion at Back

Question

```
public static void insertAtBack(List<Integer> l, int n) {
    for (int i = 0; i < n; i++) {
        l.add(new Integer(i));
    }
}
```

The above method is:

Choices

(a) much faster when `l` is an `ArrayList`

(b) much faster when `l` is a `LinkedList`

(c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

Explanation

Answer: (c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`.

The key operation here is `l.add(x)`, which adds a new element at the *end* of the list. - For an `ArrayList`, this is an amortized $O(1)$ operation because it appends to the next available position and only occasionally resizes. - For Java's `LinkedList`, which is implemented as a **doubly linked list**, insertion at the end is also $O(1)$ since it maintains a reference to the tail node.

Therefore, each call to `add()` runs in roughly constant time for both list types, and the entire loop of n iterations runs in $O(n)$ time either way.

Important note: If `LinkedList` were implemented as a **singly linked list**, appending would require traversing the entire list to reach the end—making each addition $O(n)$ and the total loop $O(n^2)$. The answer is only (c) because Java's `LinkedList` maintains both head and tail references as a doubly linked list.

Base-case example: For $n = 5$, both list types go through:

$$[\,] \;\to\; [0] \;\to\; [0,1] \;\to\; [0,1,2] \;\to\; [0,1,2,3] \;\to\; [0,1,2,3,4]$$

Each addition happens in constant time. Hence, the performance is similar for both implementations in Java.

10 DLL Insertion: Add node u *before* node p

Question

Consider the correctness of the following two methods that add a node u before the node p in a doubly linked list `DLList` (assume standard fields `prev` and `next` and that p is an existing node):

```
protected Node add(Node u, Node p) {
    u.next = p;
    u.prev = p.prev;
    u.next.prev = u;
    u.prev.next = u;
    n++;
    return u;
}

protected Node add(Node u, Node p) {
    u.next = p;
    u.next.prev = u;
    u.prev = p.prev;
    u.prev.next = u;
    n++;
    return u;
}
```

Which statement is correct?

Choices

(a) The first method is correct

(b) The second method is correct

(c) Neither method is correct

(d) Both methods are correct

(e) Both (c) and (d)

Explanation

Answer: (a) The first method is correct.

We want to splice u between p.`prev` and p:

$$\cdots \longleftrightarrow p.\texttt{prev} \longleftrightarrow p \longleftrightarrow \cdots \quad \Longrightarrow \quad \cdots \longleftrightarrow p.\texttt{prev} \longleftrightarrow \boxed{u} \longleftrightarrow p \longleftrightarrow \cdots$$

Why method 1 works. It first remembers the old predecessor (`u.prev = p.prev`) and sets `u.next = p`. Then it fixes the two external pointers that must point to u:

$$p.\texttt{prev} \leftarrow u \quad (\texttt{u.next.prev = u}), \qquad (p.\texttt{prev})\text{'s next} \leftarrow u \quad (\texttt{u.prev.next = u}).$$

No information is overwritten before it is used, so the splice is correct (assuming the usual sentinel or that p.`prev` exists).

Why method 2 is wrong (order-of-updates bug). Line 2 sets `u.next = p`, line 3 immediately overwrites `p.prev` with `u` (`u.next.prev = u`). Now line 4 does `u.prev = p.prev`, but **after** line 3 we have `p.prev = u`, so this sets `u.prev = u` (self-loop). Line 5 then executes `u.prev.next = u`, which becomes `u.next = u`. The list is corrupted.

Tiny picture example. Start with just two nodes $[X] \leftrightarrow [P]$ (so $p.\texttt{prev} = X$). Method 2 performs:

```
u.next = P
P.prev = u   (overwrites X)
u.prev = P.prev = u
u.prev.next = u ⇒ u.next = u
```

yielding a broken list. Hence only the first method is correct.

11 Linked List vs Array for Mixed Operations

Question

You need a structure that stores n integers and must support:

- $O(1)$ time to insert or delete *at the front*, and
- $O(1)$ time to access the i-th element by index, for any $0 \leq i < n$.

Which of the following data structures, as implemented in the textbook, satisfies *both* requirements?

Choices

(a) Singly linked list with a pointer to the head

(b) Doubly linked list with pointers to head and tail

(c) `ArrayStack`

(d) None of the above

(e) Both (b) and (c)

Explanation

The correct answer is **(d)**.

- A singly or doubly linked list can insert or delete at the front in $O(1)$ time, but accessing the i-th element by index takes $O(i)$ time because we must walk through the list node by node.
- An `ArrayStack` supports $O(1)$ access by index, but inserting or removing at the front (index 0) requires shifting elements and takes $O(n)$ time. **Note**: This assumes that we are using the back of the array as the top of the stack.

So none of the listed textbook structures gives both $O(1)$ random access and $O(1)$ front insert/delete. The correct answer is "none of the above".

12 SkiplistSSet vs SkiplistList: Why does one need a stack?

Question

Recall that the `SkiplistSSet` (the Skiplist implementation of an `SSet`) `add(x)` method uses a stack to keep track of the nodes where the search path drops from some list L_r into L_{r-1}. In contrast, the `SkiplistList` `add(i,x)` method does not use such a stack.

Why does the `SkiplistSSet` need a stack but not the `SkiplistList`?

Choices

(a) The stack allows us to keep the elements in sorted order, which is required for a Sorted Set (`SSet`), but not for a List.

(b) The stack allows us to only add element x if it is not already in the Sorted Set (`SSet`), but not for a List.

(c) Both (a) and (b).

(d) Neither (a) nor (b).

Explanation

Answer: (b) The stack allows us to only add x if it is not already in the `SSet`, but not for a `List`.

In a `SkiplistSSet`, `add(x)` must behave like a set insertion: it should *not* insert a duplicate key. We first search for where x belongs; while descending, we push onto a stack each node just before the downward step. At the bottom level we can check whether the next key equals x:

$$\text{if } \texttt{next.key} = x \;\Rightarrow\; \text{duplicate found, abort (no insertion);}$$

otherwise we use the stored predecessors from the stack to update forward pointers at each level and insert x exactly once. The stack is crucial here because the search path must be retraced to perform level-by-level updates only if the duplicate check passes.

In contrast, `SkiplistList`'s `add(i,x)` inserts by *index*; lists allow duplicates and there is no "already present?" check. Once the position by index is reached, we can link in x along that single path without needing to remember predecessors at higher levels, so no stack is required.

Tiny picture example (duplicate vs. unique):

$$\text{SSet contains } [5, 8, 12, 20].$$

Insert 8: descend while pushing predecessors $[5, 5, 5]$ onto the stack; at the bottom we see $\texttt{next.key} = 8$, so we *abort*—no links are changed.

Insert 11: same descent; at the bottom $\texttt{next.key} = 12 \neq 11$, so we pop the stack to splice 11 after 8 at each level. For a `List`, both insertions would proceed at the requested index without any duplicate check or stack.

13 Skiplist Node Height Distribution

Question

In the randomized `Skiplist` from the textbook, each node independently chooses its height by flipping a fair coin until the first tails appears. The height is the number of heads in a row plus one (so every node has height at least 1).

What is the probability that a given node has height *exactly* k (for $k \geq 1$)?

Choices

(a) 2^{-k}

(b) $2^{-(k-1)}$

(c) $2^{-k} - 2^{-(k+1)}$

(d) $2^{-(k+1)}$

(e) $2^{-(k-1)} - 2^{-k}$

Explanation

The correct answer is **(a)**.

To have height exactly k, the coin flips must look like:

$$\underbrace{H, H, \ldots, H}_{k-1 \text{ heads}}, T.$$

That is, we get $k-1$ heads and then a tails. The probability is

$$\Pr(\text{height } = k) = \left(\frac{1}{2}\right)^{k-1} \cdot \left(\frac{1}{2}\right) = \left(\frac{1}{2}\right)^{k} = 2^{-k}.$$

This is exactly option (a).

A nice check is that the total probability over all $k \geq 1$ is

$$\sum_{k=1}^{\infty} 2^{-k} = 1,$$

so the distribution is valid.

14 Deterministic vs Randomized SkipList Facts

Question

Consider the "fixed" (deterministic) version of the `SkipList`. Which of the following facts, which are true for the fixed `SkipList`, are still true about the randomized `SkipList`?

Choices

(a) The runtime of `get(i)`, `set(i,x)`, and `find(x)` is O(length of search path to index i or element x).

(b) The amount of space required by the `SkipList` is: (space for dummy node) + $|L_0| + |L_1| + \cdots + |L_h|$, where h is the height of the `SkipList`.

(c) Every node will have height $\leq O(\log n)$, where height is the number of lists L_r that contain the node (or equivalently, the length of its `next` array).

(d) $|L_r| \approx \frac{1}{2}|L_{r-1}|$ (in expectation) for all r.

(e) None remain true.

Explanation

Answer: (a), (b), and (d).

In both the deterministic and randomized versions of the `SkipList`, the key performance and space relationships remain valid:

(a) The runtime of `get(i)`, `set(i,x)`, and `find(x)` is still proportional to the search-path length. For the randomized version, the expected length of this path is $O(\log n)$, so the expected runtime remains $O(\log n)$.

(b) The space cost is still determined by the total number of pointers across all levels, i.e. (dummy node) $+ |L_0| + |L_1| + \cdots + |L_h|$. Randomization changes the level sizes in expectation but not this overall additive structure.

(d) Randomization causes the number of nodes per level to *halve on average*, since each node is promoted to the next level with probability $\frac{1}{2}$. Thus $|L_r| \approx \frac{1}{2}|L_{r-1}|$ holds in expectation.

Why not (c)? In the randomized version, node heights follow a geometric distribution — unbounded in theory (though $O(\log n)$ in expectation). Therefore, (c) is not guaranteed to be true for every node, only in expectation.

Mini intuition: Think of the deterministic version as "perfectly balanced," while the randomized version is "balanced on average." Runtime (a), space (b), and expected-level sizes (d) remain valid, but per-node height bounds (c) do not hold deterministically.

15 Skiplist Expected Search Path Length

Question

In a randomized `Skiplist` with n elements, the height of each node is chosen as in the previous question. Let L be the length of the search path for a random key x (assuming all keys and coin flips are independent).

Which of the following is the correct asymptotic bound on $E(L)$?

Choices

(a) $E(L) = \Theta(1)$

(b) $E(L) = \Theta(\log n)$

(c) $E(L) = \Theta(\sqrt{n})$

(d) $E(L) = \Theta(n)$

(e) $E(L) = \Theta(\log^2 n)$

Explanation

The correct answer is **(b)**.

The analysis in the textbook shows that:

- At each level, the expected number of steps to the right is constant.
- The number of levels that contain any nodes at all is $O(\log n)$ with high probability.

Therefore, the expected search path length is the sum of $O(1)$ horizontal steps over $O(\log n)$ levels, giving $E(L) = O(\log n)$. A matching lower bound can also be shown, so

$$E(L) = \Theta(\log n),$$

which is option (b).

16 Iterative Tree Walk: Counting Leaves in `traverse2()`

Question

We use the iterative `traverse2()` (shown below) to walk a binary tree without recursion. Add a variable `numLeaves` to count the number of leaves in the tree rooted at `r`.

```
0  void traverse2() {
1    Node u = r, prev = nil, next;
2    while (u != nil) {
3      if (prev == u.parent) {          // from parent, go left if can
4        if (u.left != nil) next = u.left;
5        else if (u.right != nil) next = u.right;
6        else next = u.parent;
7      } else if (prev == u.left) {     // from left, go right if can
8        if (u.right != nil) next = u.right;
9        else next = u.parent;
10     } else {                          // from right, go back to parent
11       next = u.parent;
12     }
13     prev = u;
14     u = next;
15   }
16 }
```

Change 1: Insert `int numLeaves = 0;` between lines 1 and 2.
Change 2: Where should we increment `numLeaves`? Select from the answers below:

Choices

(a) add `numLeaves++;` into the **else** statement at line 6

(b) add `numLeaves++;` into the **else** statement at line 11

(c) add `numLeaves++;` into the **else** statements at lines 6 *and* 9

(d) add `numLeaves++;` into the **else** statement at line 9

(e) add `numLeaves++;` into the **else** statements at lines 6, 9, and 11

Explanation

Answer: (a) increment only at line 6.

A node is a leaf iff *both* children are `nil`. In this walk, we detect that case *exactly* when we arrive at node u from its parent (line 3) and discover that `u.left == nil` and `u.right == nil`; then the code takes the **else** at line 6 to go back to the parent. Therefore, counting a leaf should occur precisely at line 6:

```
6        else {                 // both children nil -> u is a leaf
           numLeaves++;
           next = u.parent;
         }
```

Lines 9 and 11 are not safe places to count:

- Line 9 fires when we came *from the left* and `u.right == nil`. Node u might still have a left child (so it is not a leaf).
- Line 11 fires when we came *from the right*; by then we have already visited u's right subtree—this says nothing about u being a leaf.

Mini picture examples.

- Single node tree: r has no children. We arrive from parent (conceptually `nil`), take line 6, and increment once $\Rightarrow$ `numLeaves` $= 1$.
- One-child tree: r has only a left child. We take line 4 (not a leaf), later line 9 when returning (still not a leaf); `numLeaves` remains 0—correct.

17 BinarySearchTree Insertion and Search Properties

Question

Consider the following statements about a Binary Search Tree (BST) that stores n distinct keys.

Choices

(a) The runtime of `add(x)`, `remove(x)`, and `find(x)` is $O(\log n)$ for all BSTs.

(b) The `find(x)` operation visits at most one node per level of the tree.

(c) If all keys are inserted into an initially empty BST in *sorted order*, the resulting tree will have height $O(\log n)$.

(d) The `add(x)` operation may cause previously inserted nodes to change their parent pointers.

(e) If an in-order traversal of a BST outputs the sequence $1, 2, 3, 4, 5$, then the BST contains exactly those five keys in sorted order.

Explanation

Correct answers: (b) and (e).

(b) is true because the `find(x)` method follows exactly one path from the root to a leaf—at most one node per level—either moving left or right depending on key comparisons.
(e) is also true since an in-order traversal of a BST always visits keys in ascending sorted order, meaning the keys stored in the tree are exactly the sequence output.

The other statements are false:

(a) Runtime $O(\log n)$ only holds for *balanced* BSTs (like AVL or Red–Black trees); a plain BST can degrade to $O(n)$.

(c) Inserting sorted input causes a completely skewed (chain-like) tree, not a balanced one.

(d) `add(x)` only changes pointers locally when inserting a new node—it does not restructure existing parent links unless it's a self-balancing BST.

Mini example:

$$\text{Insert } 1, 2, 3, 4, 5 \Rightarrow \text{tree becomes a chain: } 1 \to 2 \to 3 \to 4 \to 5,$$

so height $= n - 1$ and runtime is linear, not logarithmic. An in-order traversal still prints $1, 2, 3, 4, 5$ — verifying (e).

18 Binary Tree Height and Node Count

Question

Consider a binary tree T (not necessarily a BST) with height h (root has height 0). Let n be the number of nodes in T.

Which of the following statements is *always* true?

Choices

(a) $n \leq 2^h$

(b) $n \leq 2^{h+1} - 1$

(c) $n \geq 2^{h+1} - 1$

(d) $n \geq h$

(e) $n = 2^h - 1$

Explanation

The correct answer is **(b)**.

In any binary tree of height h, level i (counting the root as level 0) can contain at most 2^i nodes. So the total number of nodes is at most

$$1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1.$$

This inequality holds for *all* binary trees, even very unbalanced ones. Therefore, $n \leq 2^{h+1} - 1$ is always true.

The other options are either too weak or not always correct. For example, $n = 2^h - 1$ only holds for a perfect tree (every level completely full).

19 Treap Insertion and Priority

Question

A `Treap` stores keys that obey the Binary Search Tree property and random priorities that obey the heap property (smaller priority value closer to the root).

Suppose you insert a new key x with random priority $p(x)$. The BST insertion puts x as a leaf. Then you rotate x upwards until the heap property is restored.

Which of the following statements is *always* true about the final position of x?

Choices

(a) x will end up as the root if $p(x)$ is the smallest priority among all nodes.

(b) x will end up as the leftmost leaf if x is the smallest key.

(c) x cannot move above a node whose key is larger than x.

(d) x may rotate above a node with smaller key if its priority is larger.

(e) x's final depth depends only on its key, not its priority.

Explanation

The correct answer is **(a)**.

In a treap:

- The *BST* property controls the relative left/right positions of keys.
- The *heap* property (on priorities) controls which nodes can be above which.

After inserting x as a leaf (BST rule), we rotate it upward as long as its priority is smaller than its parent's priority. If $p(x)$ is the smallest priority in the entire tree, then x must end up above every node, so x will be the root.

Why (b) is wrong (with example).

(b) claims: "x will end up as the leftmost *leaf* if x is the smallest key."

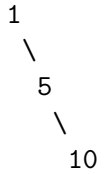
Let the existing treap (keys and priorities) be:

$$\text{key } 5 \text{ with } p(5) = 20 \quad \text{(root)}, \qquad \text{key } 10 \text{ with } p(10) = 30 \quad \text{(right child of 5)}.$$

Insert $x = 1$ with priority $p(1) = 5$ (the smallest priority):

- BST insertion puts 1 as the left child of 5.
- Since $p(1) < p(5)$, we rotate 1 up: 1 becomes the root, 5 its right child, and 10 stays as the right child of 5.

Final treap:



Here, $x = 1$ is the smallest key, but it is *not* a leaf (it has a right child). So "smallest key $\Rightarrow$ leftmost leaf" is false.

Why (c) is wrong (with example).

(c) claims: "x cannot move above a node whose key is larger than x."

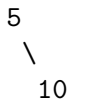
Start with a single-node treap:

$$\text{key } 10 \text{ with } p(10) = 50 \quad \text{(root)}.$$

Insert $x = 5$ with priority $p(5) = 1$ (very small):

- BST insertion: 5 goes to the left of 10.
- Since $p(5) < p(10)$, we rotate 5 up: 5 becomes the root, 10 becomes its right child.

Final treap:



Now $x = 5$ is *above* the node with larger key 10, contradicting (c). So (c) is false.

Why (d) is wrong (with example).

(d) claims: "x may rotate above a node with smaller key if its priority is larger."

But rotations only happen when the child's priority is *smaller* than the parent's priority (min-heap on priorities). If $p(x)$ is larger than its parent's priority, we *never* rotate it above that parent.

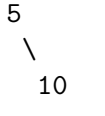
Example: start with

$$\text{key } 5 \text{ with } p(5) = 10 \quad \text{(root)}.$$

Insert $x = 10$ with priority $p(10) = 20$:

- BST insertion: 10 becomes the right child of 5.
- Here $p(10) > p(5)$, so the heap property is already satisfied; no rotations occur.

Final treap:



$x = 10$ has larger priority than the smaller key 5, but it *does not* rotate above it (and, in fact, the algorithm would never rotate in that direction). So (d) contradicts how treap rotations work.

Why (e) is wrong (with example).

(e) claims: "x's final depth depends only on its key, not its priority."

Take the same two keys but different priorities:

Case 1:

$$\text{Insert 5 with } p(5) = 10 \text{ (first)}, \quad \text{then 10 with } p(10) = 20.$$

BST: 5 root, then 10 right child. Priorities satisfy the heap rule ($10 < 20$), so no rotations. Final depth of key 10: depth 1 (child of root).

Case 2:

$$\text{Insert 5 with } p(5) = 20 \text{ (first)}, \quad \text{then 10 with } p(10) = 1.$$

BST: again 5 root, 10 right child. But now $p(10) < p(5)$, so we rotate 10 up: final treap has 10 as root, 5 as its left child. Final depth of key 10: depth 0 (the root).

Same key $x = 10$, but two different priority choices give two different depths. So the depth clearly depends on the *priority*, not just the key, and (e) is false.

Therefore, **(a)** is the only statement that is always true.

20 Scapegoat Tree Debugging Scenario

Question

You are debugging an implementation of a `ScapegoatTree` with balance parameter $\alpha = 2/3$. The tree supports inserts and deletes as described in the textbook:

- Keys are inserted like in a normal BST.
- After each insertion, if the depth of the new node x exceeds $\lfloor \log_{1/\alpha} q \rfloor$, you walk up towards the root to find a *scapegoat* node u on the path from the root to x with

  $$\max\{|u.\text{left}|, |u.\text{right}|\} > \alpha |u|$$

  and rebuild the subtree rooted at u into a perfectly balanced BST.
- A counter q tracks the total number of insertions so far. If after a deletion you have $n < \alpha q$, you rebuild the *entire* tree at the root and set $q := n$.

After running a long sequence of operations, your logging code shows:

(1) At some point, a single insertion took $\Theta(n)$ time.

(2) The value of q has been as large as 10^6.

(3) The root has been rebuilt at least once.

Which of the following statements are **guaranteed to be true**, no matter what particular sequence of operations you ran? (Select *all* that must hold.)

Choices

(a) The insertion in log entry (1) must have triggered a rebuild of a subtree of size $\Theta(n)$.

(b) After that expensive insertion in (1), the height of the tree was $O(\log n)$.

(c) Since q once reached 10^6, the root can be rebuilt at most $O(\log 10^6)$ times over the entire run.

(d) The total time spent in all subtree rebuilds up to this point is $O(q)$.

(e) If from now on you only perform insertions (no more deletions), the height of the tree will always remain $O(\log n)$.

Explanation

The correct answers are **(a)**, **(c)**, **(d)**, and **(e)**.

(a) True. A single insertion only becomes $\Theta(n)$ when it triggers a rebuild of a subtree of size $\Theta(n)$ (for example, when the scapegoat is the root, so you rebuild the whole tree). Normal BST insertion plus the path walk to find a scapegoat are only $O(\log n)$ when the height is under control; the $\Theta(n)$ cost comes from rebuilding a large subtree.

(b) False. Statement (1) says the *insertion* took $\Theta(n)$ time, but this is measured *during* the operation, not necessarily afterwards. The expensive part is the rebuild. After rebuilding, the subtree rooted at the scapegoat becomes perfectly balanced, so the height of that subtree is $O(\log |u|)$. However, this does not automatically tell us what the height of the *entire* tree is at that exact

moment (for example, if other parts of the tree were already unbalanced and did not violate the depth threshold yet). So we cannot guarantee that the whole tree's height is $O(\log n)$ immediately after the specific insertion in (1), just from that log entry alone.

(c) True. The root is rebuilt only when $n < \alpha q$ after a deletion, and then we set $q := n$. Each time this happens, q shrinks by at least a factor of $\alpha < 1$. Since q never increases during deletions and only increases by 1 per insertion, starting from some maximum value (here at most 10^6) it can only be reduced by a constant factor $O(\log q)$ times. Therefore, the root can be rebuilt at most $O(\log 10^6)$ times over the entire run.

(d) True. Every rebuild of a subtree of size k costs $O(k)$ time, and after rebuilding that subtree, it takes many operations before that subtree can become unbalanced enough to require another rebuild. The standard analysis in the textbook shows that if you "charge" the cost of a rebuild to the nodes in the rebuilt subtree, each node is charged only a constant total amount over all future rebuilds. Summing this over all nodes and all operations gives a total rebuild cost of $O(q)$ (and hence $O(m)$ where m is the number of operations so far).

(e) True. When there are only insertions (no deletions), the counter q is always equal to the current number of nodes n. By the way insertions are defined, whenever the depth of a newly inserted node exceeds $\lfloor \log_{1/\alpha} q \rfloor$, we find a scapegoat and rebuild its subtree, which restores a good height bound. This guarantees that, with only insertions, the height of the tree stays $O(\log n)$ at all times.

Summary. From the logs we can safely conclude:

- The "slow" insertion in (1) involved rebuilding a large subtree.
- The root can only be rebuilt a logarithmic number of times in terms of the maximum q.
- The total rebuild work so far is linear in the number of operations.
- As long as we only insert (no deletes), the tree's height stays logarithmic in n.

But we *cannot* conclude, just from (1), that the overall tree height right after that insertion was definitely $O(\log n)$.

21 Binary Heap `add(x)` vs `remove()` Cost

Question

A binary heap is stored in an array $a[0\,..\,n-1]$ using the usual parent/child index rules. Consider the following two operations:

- `add(x)`: place x at index n and bubble it *up*.
- `remove()`: remove the root, move the last element to index 0, and bubble it *down*.

Which statement best describes the worst-case cost of these operations?

Choices

(a) Both `add(x)` and `remove()` take $O(1)$ time.

(b) `add(x)` takes $O(\log n)$ time; `remove()` takes $O(1)$ time.

(c) `add(x)` takes $O(1)$ time; `remove()` takes $O(\log n)$ time.

(d) Both `add(x)` and `remove()` take $O(\log n)$ time.

(e) `add(x)` takes $O(\sqrt{n})$ time; `remove()` takes $O(\log n)$ time.

Explanation

The correct answer is **(d)**.

The heap is a complete binary tree of height $\Theta(\log n)$. In the worst case:

- `add(x)` may bubble x all the way from the last level up to the root, moving through $O(\log n)$ levels.
- `remove()` may bubble the replacement element from the root all the way down to a leaf, again moving through $O(\log n)$ levels.

Each step does a constant amount of work (a comparison and maybe a swap), so both operations have worst-case time $O(\log n)$.

22 Meldable Heap Shape Invariants

Question

You implement a `MeldableHeap` exactly as in the textbook: it stores elements in a binary tree that satisfies the *heap order* property, and `merge(h1,h2)` picks left/right child at random when recursing. You then run a long sequence of `add`, `remove`, and `merge` operations, starting from an empty heap.

Let n be the current number of elements in the heap. Which of the following structural properties is **guaranteed** to hold for the heap's underlying binary tree *after this sequence of operations*?

Choices

(a) The tree is always *complete* or *almost complete* (every level except possibly the last is full), like an array-based binary heap.

(b) At every node, the left subtree has at least as many nodes as the right subtree.

(c) The tree may look very unbalanced in the worst case, but its *expected* height is $O(\log n)$.

(d) An in-order traversal of the tree always lists the heap elements in sorted (non-decreasing) order.

(e) Every internal node has exactly two children (except possibly on the last level).

Explanation

The correct answer is **(c)**.

In a `MeldableHeap`, the **only** structural rule we enforce is the *heap order property*: every node's key is $\leq$ the keys in its children. We do *not* try to keep the tree complete, balanced by size, or "perfect-looking" in any other way.

The random left/right choices during `merge` make the merge process behave like a random walk down the tree. The analysis in the textbook shows that this random walk has **expected** length $O(\log n)$, which means: the expected height of the heap is $O(\log n)$. But the actual shape can be very lopsided for a particular run — only the expectation is logarithmic. That is exactly what option (c) says.

Why the other options are wrong:

- **(a) False:** A *binary heap* stored in an array keeps the tree complete, but a `MeldableHeap` does not. Its shape can be irregular; we never "fix" levels to be full.
- **(b) False:** We never compare subtree sizes, so there is no rule forcing the left subtree to be bigger than (or even comparable to) the right subtree.
- **(d) False:** In-order traversal gives sorted order only for a *Binary Search Tree*. A heap only orders nodes top-down (parent $\leq$ children), not left-to-right. In-order can be any weird order.
- **(e) False:** Internal nodes may have 0, 1, or 2 children. There is no requirement that every internal node has exactly two children.

So the "trick" is to remember that meldable heaps care about: heap order + randomized merge $\Rightarrow$ expected $O(\log n)$ height, but they do *not* enforce the neat, complete shape of array-based binary heaps.

23 2-4 Tree Height Upper Bound

Question

A 2–4 tree stores n keys in a rooted tree where every internal node has 2, 3, or 4 children, and all leaves are at the same depth.

Which of the following is a correct upper bound on the height h of a 2–4 tree in terms of n?

Choices

(a) $h \leq n$

(b) $h \leq 2\sqrt{n}$

(c) $h \leq \log_2 n$

(d) $h \leq \log_4 n$

(e) $h \leq \log_2(n+1) - 1$

Explanation

The correct answer is **(c)**.

In the worst case, each internal node has only 2 children. Then the tree behaves like a perfectly balanced binary tree. A full binary tree of height h has at least 2^h leaves and therefore at least 2^h keys. Thus,

$$n \geq 2^h \quad \Rightarrow \quad h \leq \log_2 n.$$

If some nodes have 3 or 4 children, the tree is even shorter, so this is a valid upper bound on the height.

Option (d), $\log_4 n$, is *too small* in the worst case (it assumes every internal node has 4 children), so it is not always an upper bound.

24 User Index with Red-Black Trees and 2–4 Trees

Question

You are implementing an in-memory index of usernames for a large website. The high-level design uses a 2–4 tree, but the actual code stores the data as a `RedBlackTree`.

On one shard, a single node of the conceptual 2–4 tree currently stores the three usernames

`"anna"`, `"maria"`, `"zoe"`

as a *4-node*:

`["anna" "maria" "zoe"]`

(all three keys in one 2–4-tree node).

In your red-black implementation, this single 2–4 node is represented by a small binary subtree.

Which red-black configuration (keys and colors) correctly represents this 4-node?

Choices

(a) A single **black** node storing `"maria"` with two **red** children storing `"anna"` (left) and `"zoe"` (right).

(b) A chain of three **black** nodes in a BST:

```
"anna"
    \
    "maria"
         \
         "zoe"
```

(c) A **red** root storing `"maria"` with two **black** children storing `"anna"` (left) and `"zoe"` (right).

(d) A single **black** node storing all three keys `"anna"`, `"maria"`, and `"zoe"` in an internal array.

(e) Any binary tree whose in-order traversal is `"anna"`, `"maria"`, `"zoe"`.

Explanation

The correct answer is **(a)**.

In the red-black / 2–4 correspondence:

- A 2-node (one key) ⇒ one **black** node in the red-black tree.
- A 3-node (two keys) ⇒ a black node with *one* red child.
- A 4-node (three keys) ⇒ a black node with *two* red children.

Here we have a 4-node containing three keys `"anna"` < `"maria"` < `"zoe"`. The corresponding red-black shape is:

black `"maria"` as the parent,

with a red left child `"anna"` and a red right child `"zoe"`, exactly as in option (a). The in-order traversal of that subtree is `"anna"`, `"maria"`, `"zoe"`, matching the 2–4-node's keys.

Why the other options are wrong:

- **(b)** Three black nodes in a chain correspond to *three separate 2-nodes* stacked in a tall path, not a single 4-node. The 2–4 model expects one multi-key node here, not three distinct ones.
- **(c)** A red root with black children violates the usual encoding: red links are meant to "glue" together keys *within* the same 2–4 node, and red nodes are never used as the root in the standard representation (we can always recolor so the root is black).
- **(d)** This is just a direct 2–4-tree node, not a binary `RedBlackTree` node. The whole point of the representation is that multi-key 2–4 nodes are simulated using *multiple* binary nodes, not a single node with an internal array of keys.
- **(e)** Many different binary tree shapes have in-order `"anna"`, `"maria"`, `"zoe"`, but most of them do *not* respect the red-black coloring rules that emulate a 4-node. The shape and coloring in (a) are the specific pattern that corresponds to a 4-node.

25 Graph Representation and BFS Runtime

Question

Let G be an undirected graph with n vertices and m edges. You want to run Breadth-First Search (BFS) from a single start vertex, using the textbook implementation.

Which representation and runtime pair is correct?

Choices

(a) Adjacency matrix; BFS runs in $O(n^3)$ time.

(b) Adjacency matrix; BFS runs in $O(m)$ time.

(c) Adjacency list; BFS runs in $O(n^2)$ time.

(d) Adjacency list; BFS runs in $O(n + m)$ time.

(e) Either representation; BFS always runs in $O(m)$ time.

Explanation

The correct answer is **(d)**.

With an adjacency list:

- We visit each vertex at most once.
- For each vertex, we scan through the list of its neighbors once.

Across the whole run, we do $O(n)$ work for visiting the vertices plus $O(m)$ work for scanning all adjacency lists, so the total running time is

$$O(n + m).$$

Now why the other options are wrong:

- **(a) Adjacency matrix;** $O(n^3)$

  This runtime for a BFS implementation using an adjacency matrix is $O(n^2)$, not $O(n^3)$.

- **(b) Adjacency matrix;** $O(m)$

  With an adjacency matrix, we do *not* touch only the m existing edges. For each vertex, we must check all n possible neighbors (all entries in that row), even if most entries are 0 (no edge). This gives $O(n^2)$ time, not $O(m)$, especially when $m \ll n^2$.

- **(c) Adjacency list;** $O(n^2)$

  With adjacency lists, we never scan all n possible neighbors per vertex. We only scan the actual edges that exist. That gives $O(n + m)$, not $O(n^2)$, so this overestimates the runtime.

- **(e) Either representation;** $O(m)$

  This ignores two things:

  – We must at least touch all reachable vertices, so there is an $O(n)$ term.

– With an adjacency matrix, we are forced to scan all n^2 entries in the worst case to discover neighbors, so BFS is $O(n^2)$ there, not $O(m)$.

So this statement is not true for both representations.

Therefore, the only correct representation–runtime pair is

$$\boxed{\text{Adjacency list; BFS runs in } O(n+m).}$$

26 Comparison Sorting Lower Bound

Question

You are given n distinct integers and you must sort them using only comparisons of the form "$x < y$?".

Which of the following statements is **the most correct** according to the decision-tree argument in the textbook?

Choices

(a) Any comparison-based sorting algorithm must perform at least n comparisons in the worst case.

(b) Any comparison-based sorting algorithm must perform at least $\Omega(n \log n)$ comparisons in the worst case.

(c) There exists a comparison-based sorting algorithm that runs in $O(n)$ time for all inputs.

(d) If the input is already sorted, every comparison-based algorithm will run in $O(n)$ time.

(e) Using a heap, we can sort in $O(n)$ time in the worst case.

Explanation

The correct answer is **(b)**.

The decision-tree argument says:

- Any comparison-based sorting algorithm can be seen as a binary decision tree where each internal node is a comparison.
- There are $n!$ possible permutations of n distinct elements, and the algorithm must end in a different leaf for each permutation.
- A binary tree with height h has at most 2^h leaves, so we must have $2^h \geq n!$.

Taking logs gives

$$h \geq \log_2(n!) = \Omega(n \log n),$$

so any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case. This is exactly what option (b) says.

Why the other options are not the "most correct":

- **(a)** "At least n comparisons" is technically true but too weak. The decision-tree lower bound shows we actually need *asymptotically more* than n comparisons in the worst case: $\Omega(n \log n)$. Since (b) is a strictly stronger (tighter) statement, (a) is not the best answer.
- **(c)** "There exists a comparison-based sorting algorithm that runs in $O(n)$ for *all* inputs" contradicts the lower bound. If an algorithm were $O(n)$ for all inputs, its worst-case number of comparisons would be $O(n)$, but the decision-tree argument proves any comparison-based algorithm needs $\Omega(n \log n)$ comparisons in the worst case. So (c) is false.

- **(d)** "If the input is already sorted, *every* comparison-based algorithm runs in $O(n)$" is too strong. Some algorithms (like a naive implementation of selection sort or heapsort) still do $\Theta(n \log n)$ comparisons even on already sorted input: they don't check or exploit that the input is pre-sorted. Only *some* algorithms (like insertion sort, certain mergesort variants, etc.) can be implemented so that they run in $O(n)$ on already sorted input. Since the statement quantifies over *every* algorithm, it is false.
- **(e)** "Using a heap, we can sort in $O(n)$ time in the worst case" is also false. Heapsort works by:

  $$\text{build-heap in } O(n) \text{ time} \quad + \quad n \text{ extract-min operations at } O(\log n) \text{ each,}$$

  for a total of $\Theta(n \log n)$ comparisons in the worst case. So using a heap does *not* beat the $\Omega(n \log n)$ barrier.

Therefore, (b) is the most correct statement that matches the decision-tree lower-bound argument.

27 Chained Hash Table Operations with Separate Chaining

Question

Consider a `ChainedHashTable` that uses an array $a[0 \ldots m-1]$ of singly linked lists (separate chaining). The hash function is

$$h(k) = k \bmod 5,$$

and `add(x)` always inserts x at the *front* of the list at index $h(x)$, while `remove(x)` deletes the *first* occurrence of x in that list (if it exists).

Suppose $m = 5$ and the current table contents are:

Index	List (front to back)
0	$10 \to 5$
1	6
2	empty
3	$3 \to 8$
4	empty

Now perform the following operations in order:

```
add(13); remove(5); add(20);
```

What are the final contents of the table (lists shown from front to back in each bucket)?

Choices

(a)

0	$10 \to 20$
1	6
2	empty
3	$3 \to 8 \to 13$
4	empty

(b)

0	$20 \to 10$
1	6
2	empty
3	$13 \to 3 \to 8$
4	empty

(c)

0	$10 \to 5 \to 20$
1	6
2	empty
3	$3 \to 8 \to 13$
4	empty

(d)

0	$20 \to 10 \to 5$
1	6
2	empty
3	$13 \to 8 \to 3$
4	empty

(e)

0	$20 \to 5 \to 10$
1	6
2	empty
3	$13 \to 3$
4	8

Explanation

The correct answer is **(b)**.

We track each operation step by step using the hash function and the definition of `add` and `remove`:

Initial table:

0	$10 \to 5$
1	6
2	empty
3	$3 \to 8$
4	empty

1) `add(13)`

Compute $h(13) = 13 \bmod 5 = 3$. We insert 13 at the *front* of the list at index 3:

$$3 : 13 \to 3 \to 8.$$

Now the table is:

0	$10 \to 5$
1	6
2	empty
3	$13 \to 3 \to 8$
4	empty

2) `remove(5)`

Compute $h(5) = 5 \bmod 5 = 0$. We only look in bucket 0 and delete the *first* occurrence of 5 in the list $10 \to 5$:

$$0 : 10 \to 5 \quad \Rightarrow \quad 0 : 10.$$

The table becomes:

0	10
1	6
2	empty
3	$13 \to 3 \to 8$
4	empty

3) `add(20)`

Compute $h(20) = 20 \bmod 5 = 0$. We insert 20 at the *front* of the list at index 0:

$$0 : 20 \to 10.$$

Final table:

0	$20 \to 10$
1	6
2	empty
3	$13 \to 3 \to 8$
4	empty

which matches option (b).

Why the other options are wrong:

- **(a)** Puts 13 at the *back* of bucket 3 and 20 at the *back* of bucket 0. Our definition says `add` inserts at the *front*, so this is incorrect.
- **(c)** Leaves 5 in bucket 0, but we explicitly called `remove(5)`, which should delete it from the list at index 0.
- **(d)** Keeps 5 and also reverses some list orders (e.g. at bucket 3), which does not match "insert at front / remove first occurrence."
- **(e)** Moves 8 into bucket 4, but the hash function is $h(k) = k \bmod 5$, so 8 always belongs in bucket 3 (since $8 \bmod 5 = 3$). Elements never move between buckets unless their key changes (which it does not).

This question is meant to test your understanding that in a chained hash table:

- The bucket index is determined solely by $h(x)$.
- `add(x)` affects only one list (at $h(x)$), usually at its front or back depending on the implementation.
- `remove(x)` also only searches and modifies that same list.

28 Linear Probing and Load Factor

Question

A `LinearHashTable` uses open addressing with linear probing. The table has size m and stores n keys, so the load factor is $\lambda = n/m < 1$. Assume the hash function behaves like a random function and keys are inserted using linear probing.

Which statement best describes the expected cost of a `find(x)` operation?

Choices

(a) Expected cost is $O(1)$ as long as $\lambda < 1$, independent of how close to 1 it is.

(b) Expected cost stays $O(1)$ only if λ is bounded away from 1 (for example, $\lambda \leq 1/2$).

(c) Expected cost is $\Theta(\lambda)$.

(d) Expected cost is $\Theta(1/(1-\lambda))$.

(e) Expected cost is $\Theta(\log(1/(1-\lambda)))$.

Explanation

The correct answer is **(d)**.

For linear probing with a random hash function, the textbook shows that the average number of probes for a search is proportional to

$$\frac{1}{1-\lambda}.$$

As λ gets close to 1, this value grows quickly, which reflects the clustering problem in linear probing.

This means:

- The expected cost is still constant when λ is bounded away from 1 (for example, $\lambda \leq 1/2$ gives a constant upper bound).
- But the correct asymptotic expression in terms of λ is $\Theta\big(1/(1-\lambda)\big)$, which is option (d).

Short Answer Questions

29 BST Construction from Insertion Sequence

Question

Draw the final **BinarySearchTree** (starting from an empty tree) after performing the insertions:

`add(18), add(7), add(20), add(13), add(16), add(14), add(19), add(5), add(2), add(12).`

Explanation

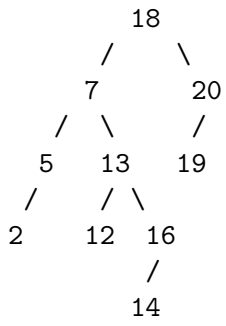
Model answer (full marks):

We insert each key using the BST rule: go left if the key is smaller, right if larger.

Insertion trace (key → path):

$$
\begin{aligned}
18:&\ \text{root}\\
7:&\ 7<18 \Rightarrow \text{left of } 18\\
20:&\ 20>18 \Rightarrow \text{right of } 18\\
13:&\ 13<18 \ \rightarrow\ 13>7 \Rightarrow \text{right of } 7\\
16:&\ 16<18 \ \rightarrow\ 16>7 \ \rightarrow\ 16>13 \Rightarrow \text{right of } 13\\
14:&\ 14<18 \ \rightarrow\ 14>7 \ \rightarrow\ 14>13 \ \rightarrow\ 14<16 \Rightarrow \text{left of } 16\\
19:&\ 19>18 \ \rightarrow\ 19<20 \Rightarrow \text{left of } 20\\
5:&\ 5<18 \ \rightarrow\ 5<7 \Rightarrow \text{left of } 7\\
2:&\ 2<18 \ \rightarrow\ 2<7 \ \rightarrow\ 2<5 \Rightarrow \text{left of } 5\\
12:&\ 12<18 \ \rightarrow\ 12>7 \ \rightarrow\ 12<13 \Rightarrow \text{left of } 13
\end{aligned}
$$

Final BST (ASCII diagram):



Checks (optional but recommended for students):

$$\text{In-order traversal} = (2, 5, 7, 12, 13, 14, 16, 18, 19, 20)$$

which is the sorted order of all inserted keys, confirming the structure is a valid BST.

30 Expected Number of Pointer Changes in `SkiplistSSet`

Question

Show that, during an `add(x)` or `remove(x)` operation, the expected number of pointers in a `SkiplistSSet` that get changed is constant.

Explanation

Full-mark explanation:

Every node in a `SkiplistSSet` has a random height h, where the probability of having height at least r is 2^{-r}. This means that:

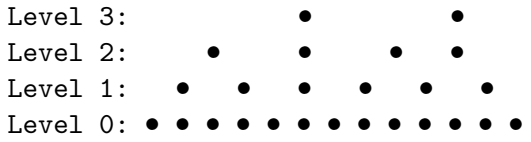
$$\mathbb{E}[h] = \sum_{r=1}^{\infty} 2^{-r} = 1.$$

Therefore, when a node is added or removed, the expected number of pointers that must be updated—one per level up to its height—is a constant, independent of n.

Intuition / Visualization (the way to actually "see" it):

Imagine that each node in the skiplist flips a coin to decide how many levels of "express lanes" it gets: - The first coin flip always succeeds, so every node appears at least in the bottom list (L_0). - Each additional level ($L_1, L_2, \dots$) requires another heads in a row. - So about half of the nodes make it to level 1, one-quarter to level 2, one-eighth to level 3, and so on.

Visually, if you drew your skiplist as stacked linked lists:



Notice how the number of nodes shrinks by about half at each higher level. This means that most nodes only appear in the bottom one or two levels, and only a few "lucky" tall ones reach high levels.

When we do an `add(x)` or `remove(x)`:
1. We first follow the search path to find where x belongs (this takes $O(\log n)$ time, but that's not what we're counting here).
2. Then we only adjust pointers for the small tower of nodes where x is inserted or removed. Each level of the tower requires two pointer updates — one before and one after the new node (or one before and one after deletion).
3. Because the expected tower height is just 1, that means on average we only change about 2–4 pointers total.

Analogy: Think of inserting a new "train station" into a railway system: - Every station connects to the main track (level 0). - Some stations also get an express line (level 1), a super-express line (level 2), etc. When you build a new station, you only connect it to a few of these tracks, and those connections are rare at higher levels. On average, you only need to hook it into a constant number of lines.

So, although searching in a skiplist can take $O(\log n)$ time, the **actual number of pointers changed during insertion or removal stays constant on average**:

$$\boxed{\text{Expected number of pointer changes} = O(1).}$$

31 Linked List Pointer Manipulation and Structure Understanding

Question

Suppose we have a singly linked list that initially stores the elements:

$$A \to B \to C \to D \to \texttt{null}.$$

You are given a pointer `p` that points to node `B`. Write the sequence of pointer updates required to insert a new node `X` *after* `B`, and then delete the node `C` from the list (assume you have a pointer to it). After both operations, draw or describe the final structure of the list.

Explanation

In a singly linked list, each node has one pointer, `next`, that points to the next node in the sequence. To insert `X` after `B`, we perform the following pointer updates:

```
X.next = B.next;  B.next = X;
```

This means that `X` now points to `C`, and `B` now points to `X`, giving:

$$A \to B \to X \to C \to D \to \texttt{null}.$$

To remove `C`, we bypass it by making the previous node's `next` pointer skip over `C`:

```
X.next = C.next;
```

This effectively removes `C` from the chain (though the node still exists in memory until garbage-collected or freed).

Final structure:

$$A \to B \to X \to D \to \texttt{null}.$$

Key ideas for you to remember:

- Always update pointers in the correct order—never lose track of the rest of the list.
- For insertion, link the new node to the rest of the list *before* changing the previous node's `next`.
- For deletion, skip over the target node by adjusting the `next` of the preceding node.

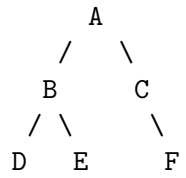
Optional diagram:

```
Initial:       A -> B -> C -> D -> null
After insert:  A -> B -> X -> C -> D -> null
After delete:  A -> B -> X -> D -> null
```

32 Binary Tree Traversal Order Interpretation

Question

Consider the following binary tree (not a Binary Search Tree):



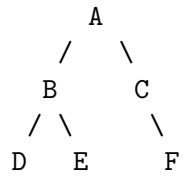
(a) Write the **in-order**, **pre-order**, and **post-order** traversal sequences for this tree.
(b) Briefly explain in your own words what distinguishes these three traversal orders conceptually.

Explanation

Full-mark explanation:

Traversal orders define the sequence in which nodes are visited during a recursive exploration of a binary tree.

Using the given tree:



We compute:

In-order (Left → Root → Right):

$$D, B, E, A, C, F$$

Pre-order (Root → Left → Right):

$$A, B, D, E, C, F$$

Post-order (Left → Right → Root):

$$D, E, B, F, C, A$$

Conceptual distinctions:

- **Pre-order** visits the root *first*, making it useful for copying or serializing the structure of a tree.
- **In-order** visits the root *between* its left and right subtrees — this yields sorted order in the special case of a BST, but here it just reflects "left-to-right" hierarchy.

- **Post-order** visits the root *last*, making it useful for deleting or evaluating trees bottom-up (e.g., in expression trees).

Mini visualization of traversal order:

- In-order: process children before parent (think "middle visit").
- Pre-order: visit the parent first, then the children.
- Post-order: visit both children, then the parent last.

Summary Table:

Order	Visit Sequence	Output for Tree Above
Pre-order	Root, Left, Right	A, B, D, E, C, F
In-order	Left, Root, Right	D, B, E, A, C, F
Post-order	Left, Right, Root	D, E, B, F, C, A

33 Comparing `ArrayStack` and `ArrayQueue` Behavior

Question

Suppose we start with two empty data structures that each store integers: an `ArrayStack` and an `ArrayQueue`. We perform the following operations in this order on *each* structure:

`add(1), add(2), add(3), remove(), add(4).`

(a) Write the final sequence of elements (from front/top to back) for each structure. (b) Briefly explain why they differ.

Explanation

1) `ArrayStack` (LIFO). The top is at the right; `add(x)` pushes, `remove()` pops.

`add(1)` : $[1]$
`add(2)` : $[1, 2]$
`add(3)` : $[1, 2, 3]$
`remove()` : pop $3 \Rightarrow [1, 2]$
`add(4)` : $[1, 2, 4]$

Final (top to bottom): $4, 2, 1$. As a left-to-right array view (front to back): $[1, 2, 4]$.

2) `ArrayQueue` (FIFO). The front is at the left; `add(x)` enqueues at back, `remove()` dequeues from front.

`add(1)` : $[1]$
`add(2)` : $[1, 2]$
`add(3)` : $[1, 2, 3]$
`remove()` : dequeue $1 \Rightarrow [2, 3]$
`add(4)` : $[2, 3, 4]$

Final (front to back): $[2, 3, 4]$.

Why they differ. A stack removes the most recent element (LIFO), so the `remove()` deletes 3. A queue removes the oldest element (FIFO), so the `remove()` deletes 1. After the final `add(4)`, the stack holds $[1, 2, 4]$ (top $= 4$) while the queue holds $[2, 3, 4]$ (front $= 2$).