Data Structure, Quiz 1 — Solution

A. Single Choice Questions — Answers & Explanations

(1-1) / (1-2) Answer: **(B)** `3 2`

Key concepts: class variable (Python) / static member variable (C++)

- `count` / `static int count` is a **class-level (shared) variable** — it is shared across all instances and incremented with each constructor call.
- `self.id` / `id` is an **instance variable** — assigned individually at construction time and never changes afterward.

Object created	`count` after	Instance `id`
`c1`	1	1
`c2`	2	2
`c3`	3	3

- `Counter.count` = **3** (final shared value)
- `c2.id` = **2** (captured at the moment c2 was created)
- Output: `3 2`

(A) is wrong because `count` starts at 0 but is incremented before `id` is assigned.
(C) is wrong because `c2.id` was set when `count` was 2, not 3.
(D) is wrong because three objects are created, so `count` ends at 3.

(2) Answer: **(C)**

Key concept: correctness + time complexity of anagram detection

- **(A)** Brute-force: correct, but $O(n^2)$ — not optimal.
- **(B)** Sort-then-compare: correct, but $O(n \log n)$ — better, still not optimal.
- **(C)** Frequency count with 26-element array: **correct AND** $O(n)$ — optimal choice.
- **(D)** Set comparison: **NOT correct** — a `set` discards duplicate characters. For example, `"aab"` and `"abb"` both produce the set `{'a', 'b'}`, so they would be mistakenly identified as anagrams.

The trap in this question is option (D): its $O(n)$ complexity looks attractive, but it fails on strings with repeated characters.

(3) Answer: **(B)**

Key concept: time-space tradeoff

- **(A) False** — using more memory does not *always* make an algorithm faster; it depends on the situation.
- **(B) True** — this is the classic tradeoff: storing precomputed results (e.g., a 26-element character-count array in anagram detection) can reduce time from $O(n^2)$ to $O(n)$, at the cost of extra space.
- **(C) False** — time and space complexities are independent; e.g., merge sort is $O(n \log n)$ time and $O(n)$ space.
- **(D) False** — code length has no direct relationship to algorithmic complexity.

(4) Answer: **(C)**

Key concept: list construction efficiency (from lecture benchmark)

- **(A)** Preallocate + fill by index: $O(n)$ total — efficient.
- **(B)** Repeated `append` / `push_back`: amortized $O(1)$ per operation → $O(n)$ total — efficient.
- **(C)** Repeated **concatenation**: each step copies all existing elements to a new array.
  - Step 1: copy 1 element; Step 2: copy 2 elements; ...; Step n: copy n elements.
  - Total: $1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$ — **worst**.
- **(D)** Built-in generation: $O(n)$, often with additional C-level optimizations — fastest in practice.

The key insight is that concatenation creates a brand-new array at each step, causing quadratic total work.

(5) Answer: **(B)**

Key concept: compact array vs referential array

- **(A) False** — compact arrays require a **fixed, uniform element type** (e.g., all `int32`). Mixed types require a referential structure.
- **(B) True** — a compact array stores raw data values side by side in memory, with no per-element pointer overhead. A referential array (Python list) stores a pointer per element **plus** the pointed-to object, consuming extra memory.
- **(C) False** — compact arrays are typically fixed-size; dynamic resizing is handled at a higher level and is not inherently faster.
- **(D) False** — any array (compact or referential) that supports dynamic resizing has over-allocation overhead.

(6) Answer: **(B)** $O(n)$

Key concept: singly linked list traversal

Without a `tail` reference, the only way to reach the last node is to start at `head` and follow `.next` pointers one by one until reaching a node whose `.next` is `None` / `NULL`. With n nodes, this requires $n - 1$ pointer traversals → $O(n)$.

This motivates maintaining a `tail` pointer: it makes tail-append $O(1)$ at the cost of extra bookkeeping during other operations.

(7) Answer: **(C) The first node (head) of the list**

Key concept: circularly linked list structure

- In a **singly linked list**, the last node's `next` = `None` / `NULL` — this signals the end of the list.
- In a **circularly linked list**, the last node's `next` is redirected to point back to the **head** node, forming a closed loop.

This means traversal can begin at any node and will eventually visit every node in the list without encountering a `None` terminator.

(8) Answer: **(B) Stack**

Key concept: LIFO for base conversion

When converting, e.g., decimal 13 to binary:

- 13 ÷ 2 = 6 remainder **1** (LSB)
- 6 ÷ 2 = 3 remainder **0**
- 3 ÷ 2 = 1 remainder **1**
- 1 ÷ 2 = 0 remainder **1** (MSB)

Remainders are generated in **reverse** order (LSB first, MSB last). Pushing them onto a stack and then popping reverses the sequence → MSB is output first. Result: $\mathbf{1101_2}$.

- **(A)** A queue would output in the original order (LSB first) — incorrect.
- **(C)** A deque *could* work, but a simple stack is the natural choice.

(9) Answer: **(A) 7**

Postfix evaluation using a stack, scanning left to right:

Token	Action	Stack
`3`	push	`[3]`
`4`	push	`[3, 4]`
`+`	pop 4, 3 → 3+4=7, push	`[7]`
`2`	push	`[7, 2]`
`*`	pop 2, 7 → 7×2=14, push	`[14]`
`7`	push	`[14, 7]`

Token	Action	Stack
`-`	pop 7, 14 → 14−7=7, push	`[7]`

Result = **7**

(10) Answer: **(B)**

- **Unordered linked list** `add`: typically inserts at the `head` in $O(1)$ — no traversal needed since order does not matter.
- **Ordered linked list** `add`: must traverse the list from `head` to find the insertion position that maintains sorted order → $O(n)$ in the worst case.

(A) False — ordered list does NOT insert at head (that would break sorted order).
(C) False — inserting at tail maintains order only by coincidence.
(D) False — linked lists do not support random access, so binary search requires $O(n)$ traversal anyway; there is no benefit.

(11) Answer: **(B)**

- **(A)** Undo/Redo → **Stack** (LIFO: the last action performed is the first to be undone).
- **(B)** Printer spooler → **Queue** (FIFO: jobs are processed in arrival order — first come, first served). ✓
- **(C)** Postfix expression evaluation → **Stack** (operands are pushed; operators pop operands).
- **(D)** Browser back button → **Stack** (the most recently visited page is returned to first).

(12) Answer: **(B)**

Key concept: shallow copy behavior with nested mutable objects

A **shallow copy** (e.g., `copy.copy(a)` in Python, or copying a `vector<vector<int>>` by value in C++) creates a **new outer container**, but the elements inside are **copied by reference** (shared).

```python
import copy
a = [[1, 2], [3, 4]]
b = copy.copy(a)    # new outer list; inner lists are SHARED
b[0][0] = 99        # modifies the SHARED inner list → a[0] is also [99, 2]
b[1] = [5, 6]       # replaces b's reference; a[1] is unaffected → still [3, 4]
print(a)            # [[99, 2], [3, 4]]
```

- **(A)** describes a **deep copy** (`copy.deepcopy`).
- **(C)** describes a simple **assignment** (`b = a`) — an alias.
- **(D)** is the reverse of the truth.

B. Short-answer Questions — Reference Answers

(13) Reference Answers

(a) Method overriding
A child class provides its own implementation of a method that already exists in the parent class with the same name (and same signature). When the method is called on a child-class object, the child's version executes instead of the parent's. This is the primary mechanism for achieving runtime polymorphism in OOP (via `override` in C++ or simply redefining the method in Python).

(b) Polymorphism
The ability for the same interface (method name or operator) to behave differently depending on the actual type of the object at runtime. In OOP, it is most commonly achieved through **method overriding** (a subclass redefines a parent method) and **operator overloading** (custom classes define behavior for operators like `+`, `<`). The caller does not need to know the exact type of the object.

(c) Circularly linked list
A linked list in which the `next` pointer of the last node points back to the first node (head) instead of `None` / `NULL`, forming a closed loop. Any node can serve as a starting point to traverse all nodes in the list. Commonly used in scenarios requiring repeated cycling through elements (e.g., round-robin scheduling).

(d) Dictionary of Keys (DOK) format
A sparse matrix representation that uses a hash table (dictionary/ `dict`) where each key is a `(row, col)` tuple identifying the position of a non-zero element, and the value is the element itself. Elements not present in the dictionary are implicitly zero. Supports average-case $O(1)$ lookup and update. Well-suited for incrementally building a sparse matrix.

(14-1) / (14-2) Reference Answer: $O(n^2)$

First segment — triple nested loop with constant innermost:

$$n \times n \times 10 = 10n^2 = O(n^2)$$

The constant factor `10` is dropped in Big-O notation.

Second segment — outer `for` with inner `while j *= 2` up to `j <= i` :

- For a given `i` , the while loop runs $\lfloor \log_2 i \rfloor + 1$ times.
- Total: $\sum_{i=1}^{n} (\lfloor \log_2 i \rfloor + 1) \approx n \log n = O(n \log n)$

Overall: $O(n^2) + O(n \log n) = \boxed{O(n^2)}$

> The key insight here is recognizing that the constant `10` in the innermost loop has **no effect** on Big-O, and that summing $\lfloor \log_2 i \rfloor$ over $i = 1$ to n gives $O(n \log n)$, which is dominated by $O(n^2)$.

(15) Reference Answer

Infix expression: `(A + B) * (C - D) / E`

(a) Postfix expression: `A B + C D - * E /`

Step-by-step conversion:

Token	Action	Stack (bottom→top)	Output so far
`(`	push	`(`	
`A`	output	`(`	`A`
`+`	push	`( +`	`A`
`B`	output	`( +`	`A B`
`)`	pop until `(` → output `+`	empty	`A B +`
`*`	push	`*`	`A B +`
`(`	push	`* (`	`A B +`
`C`	output	`* (`	`A B + C`
`-`	push	`* ( -`	`A B + C`
`D`	output	`* ( -`	`A B + C D`
`)`	pop until `(` → output `-`	`*`	`A B + C D -`
`/`	`/` and `*` have equal precedence → pop `*` then push `/`	`/`	`A B + C D - *`
`E`	output	`/`	`A B + C D - * E`
end	pop all → output `/`	empty	`A B + C D - * E /`

(b) Maximum symbols in stack: 3 — occurring when the stack contains `* ( -` (after scanning `C -`).

(16) Reference Answer

Step-by-step trace:

Operation	Stack contents (bottom → top)	Notes
`push(3)`	`3`	
`push(7)`	`3, 7`	

Operation	Stack contents (bottom → top)	Notes
`push(1)`	`3, 7, 1`	
`pop()`	`3, 7`	removes 1 (top)
`push(4)`	`3, 7, 4`	
`push(9)`	`3, 7, 4, 9`	
`pop()`	`3, 7, 4`	removes 9 (top)
`push(2)`	`3, 7, 4, 2`	

(a) Final contents (bottom → top): `3, 7, 4, 2`

(b) `S.peek()` **returns** `2` — the element at the top of the stack without removing it.

(17) Reference Answer

Assume row-major order: $\text{addr}(B[i][j]) = \text{base} + (i \times C + j) \times S$

From the three given addresses:

$$\text{base} + (2C+1)S = 36 \quad \cdots(1)$$

$$\text{base} + (3C+4)S = 54 \quad \cdots(2)$$

$$\text{base} + 3S = 16 \quad \cdots(3)$$

$(2) - (1)$: $(C+3)S = 18 \quad \cdots(4)$

$(1) - (3)$: $(2C-2)S = 20 \quad \cdots(5)$

From (4): $S = \dfrac{18}{C+3}$. Substituting into (5):

$$(2C-2) \cdot \frac{18}{C+3} = 20 \implies 36C - 36 = 20C + 60 \implies 16C = 96 \implies \boxed{C = 6}$$

$$S = \frac{18}{9} = \boxed{2} \qquad (\text{element size} = 2)$$

$$\text{base} = 16 - 3 \times 2 = \boxed{10}$$

Verification — column-major would require: $\text{addr}(B[i][j]) = \text{base} + (j \times R + i) \times S$

$(1) - (3)$: $(R + 2 - 3R)S = (2 - 2R)S = 20$. Since $R \geq 1$, we get $2 - 2R \leq 0$, which would require $S \leq 0$ — **impossible**. Therefore the order is confirmed as **row-major**.

$$\text{addr}(B[4][2]) = 10 + (4 \times 6 + 2) \times 2 = 10 + 26 \times 2 = 10 + 52 = \boxed{62}$$

Summary: Row-major, 6 columns, element size = 2, $B[4][2]$ is at address **62**.