Data Structure, Exam A — Solution

A. Single Choice Questions (5% each, 60%)

(1) Singly linked list append — time complexity (no tail pointer)

(B) $O(n)$ — must traverse the entire list to find the last node

With only a `head` pointer, there is no direct reference to the last node. To append, the algorithm must follow `next` pointers from `head` until reaching the node whose `next` is `None`, then link the new node. This requires visiting all n existing nodes $\rightarrow O(n)$.

Contrast: If a `tail` pointer is also maintained, append becomes $O(1)$ — simply set `tail.next = new_node; tail = new_node`.

(2-1) Operator overloading (`__add__`) code trace — Python

(B) `5/6`

`f1 + f2` invokes `Fraction.__add__`: `new_num = 1×3 + 1×2 = 5`, `new_den = 2×3 = 6` → `Fraction(5, 6)` → `__repr__` returns `"5/6"`.

(2-2) Operator overloading (`operator+`) code trace — C++

(B) `5/6`

`operator+` computes `new_num = 1×3 + 1×2 = 5`, `new_den = 2×3 = 6`. The `operator<<` overload outputs `5/6`.

Same result as Python — operator overloading semantics are equivalent in both languages for this example.

(3) ADT — which statement is TRUE?

(C)

An ADT separates **interface** from **implementation**:

- It specifies **what** operations exist (`push`, `pop`, `peek`) and **what** they guarantee (LIFO order).
- It says **nothing** about internal storage (array vs linked list vs deque).

Why the others are wrong:

- **(A)** — ADT deliberately does **not** specify implementation; that is its defining feature.
- **(B)** — Both a list-based Stack and a linked-list-based Stack are the **same** Stack ADT, because the ADT is defined by behavior, not by storage.
- **(D)** — ADT and data structure are **distinct** concepts. An ADT is the abstract specification; a data structure is a concrete implementation of that specification.

(4-1) Python list vs dict `in` operator

(A) $O(n)$ for list, $O(1)$ for dict

A Python `list` requires linear scan; a `dict` uses a hash table with $O(1)$ average key lookup.

(4-2) C++ vector vs unordered_map search

(A) $O(n)$ for vector, $O(1)$ for unordered_map

`std::find` on a vector is a linear scan; `std::unordered_map::count` uses a hash table.

(5) Invalid postfix expression

(C) `* A B + C`

Reading left to right as postfix: the first token is `*`, which is an operator needing two operands — but the stack is empty. This is invalid. (Note: `* A B + C` is a valid **prefix** expression: `*(A, B) + C` is not well-formed prefix either... the point is that postfix evaluation fails immediately at the `*` token.)

(6) Stack contents after converting 53 to binary

(C) `1, 0, 1, 0, 1, 1` **(bottom to top)**

Division steps (push remainder each time):

Division	Quotient	Remainder (pushed)
53 ÷ 2	26	1
26 ÷ 2	13	0
13 ÷ 2	6	1
6 ÷ 2	3	0
3 ÷ 2	1	1
1 ÷ 2	0	1

Stack (bottom to top): **1, 0, 1, 0, 1, 1**. Popping gives `110101₂ = 53` ✓

(7) Doubly linked list deletion with direct node reference

(A) $O(1)$

In a doubly linked list, each node has both `next` and `prev` pointers. Given a reference to the node, we can reach its predecessor via `node.prev` directly, relink in $O(1)$ — no traversal needed. (Compare: singly linked list requires $O(n)$ traversal to find the predecessor.)

(8) Prefix expression evaluation — `* + 2 3 - 7 4`

(B) 15

Parse recursively (or right-to-left with a stack):

- `+ 2 3` → $2 + 3 = 5$
- `- 7 4` → $7 - 4 = 3$
- `* 5 3` → $5 \times 3 = 15$

(9-1) Python class variable shared across instances — `Counter`

(A) `3` / `3`

`Counter.count` is a **class variable** — one copy shared by the class and all its instances.

Each `Counter()` call triggers `__init__`, which executes `Counter.count += 1`. After three instantiations (`a`, `b`, `c`), `Counter.count = 3`.

`a.count` looks up `count` on instance `a` first — no instance attribute found — falls through to the class attribute `Counter.count` → also `3`.

Key: Writing `Counter.count += 1` (not `self.count += 1`) ensures the class variable is modified, not a new instance attribute created on `self`.

(9-2) C++ static member — `Counter`

(A) `3` / `3`

`Counter::count` is a **static member variable** — one copy shared across all instances, initialized to `0` outside the class.

Each `Counter()` constructor call executes `count++`. After three objects (`a`, `b`, `c`), `Counter::count = 3`.

`a.count` accesses the same static variable (C++ allows instance access syntax for static members) → also `3`.

Same behaviour as Python (9-1): Both Python class variables and C++ static members provide one shared copy per class. The difference is syntax — Python uses `Counter.count` (or `self.count` for read, carefully), while C++ uses `Counter::count`.

(10) Deque-based palindrome checker

(A) Remove from front and rear simultaneously; palindrome if all pairs match

After loading all characters into the deque (rear), repeatedly remove one from the front and one from the rear and compare. If they always match and the deque ends empty (even length) or with one character (odd length), the string is a palindrome.

(11) Queue operations trace — final contents

(A) `[7, 1]`

Operation-by-operation trace (Queue: front → rear):

Operation	Queue (front → rear)
Initial	`[]`
`enqueue(5)`	`[5]`
`enqueue(3)`	`[5, 3]`
`dequeue()` → returns `5`	`[3]`
`enqueue(7)`	`[3, 7]`
`enqueue(1)`	`[3, 7, 1]`
`dequeue()` → returns `3`	`[7, 1]`

Queue is FIFO — the first element enqueued is the first dequeued.

(12) Big O notation — which statement is FALSE?

(B)

Big O notation **ignores constant factors**. Both n^2 and $1000n^2$ are $O(n^2)$, but Algorithm B performs **1000 times more steps** than Algorithm A for every n.

n	n^2 **steps (A)**	$1000n^2$ **steps (B)**
100	10,000	10,000,000
1,000	1,000,000	1,000,000,000

Big O tells us how an algorithm **scales** as n grows — not the exact step count. Two algorithms in the same Big O class can differ enormously in practice.

Why the others are correct (and not the answer):

- **(A)** — Constant factors and lower-order terms are dropped in Big O.
- **(C)** — n^2 steps → $O(n^2)$ by definition.
- **(D)** — Big O describes scaling behavior, not exact step count.

B. Short-answer questions (8% each, 40%)

(13) Term definitions

(a) Method Overloading: Defining multiple methods in the same class with the same name but different parameter lists (type or number). The correct version is chosen at call time based on the arguments. Python does not support traditional overloading but simulates it using default arguments or `*args`.

(b) Row-Major Order: A 2D array storage convention where elements are stored row by row in contiguous memory. For `A[i][j]` in a matrix with c columns: `1D index = i × c + j`. Common in C/C++ and NumPy (default).

(c) Deque (Double-ended Queue): A linear data structure that allows $O(1)$ insertion and removal from both the front and the rear. It generalises both Stack (LIFO) and Queue (FIFO).

(d) Infix Expression: An arithmetic notation where operators are placed **between** their operands (e.g., `A + B`). Requires parentheses and operator precedence rules for unambiguous evaluation.

(14-1) Time complexity analysis (Python)

Segment 1 — `while i <= n, i *= 3`:

- i takes values $1, 3, 9, \ldots, 3^k$ where $3^k \leq n$, so $k \leq \log_3 n$.
- Inner loop runs i steps for each i.
- Total $= 1 + 3 + 9 + \cdots + 3^{\lfloor \log_3 n \rfloor} = \frac{3^{\lfloor \log_3 n \rfloor + 1} - 1}{2} \approx \frac{3n}{2} = O(n)$.

Segment 2 — `for i in range(n): for j in range(1, n+1, 2)`:

- Outer loop: n iterations.
- Inner loop: $j = 1, 3, 5, \ldots \rightarrow$ about $\frac{n}{2}$ iterations.
- Total $= n \times \frac{n}{2} = O(n^2)$.

Overall: $O(n) + O(n^2) = O(n^2)$

(14-2) Time complexity analysis (C++)

Same analysis as (14-1):

- `while (i <= n) { ...; i *= 3; }` → geometric series with ratio 3 → $O(n)$
- `for (i=0; i<n; i++) for (j=1; j<=n; j+=2)` → $n \times \frac{n}{2} \rightarrow O(n^2)$

Overall: $O(n^2)$

(15) Infix to Postfix conversion — `A * B - (C + D) / E`

(a) Step-by-step trace:

Token	Operator Stack (bottom→top)	Output
A	[]	A
`*`	[*]	A
B	[*]	A B
`-`	[-]	A B * (pop * since same or higher prec than -)
`(`	[-, (]	A B *
C	[-, (]	A B * C
`+`	[-, (, +]	A B * C
D	[-, (, +]	A B * C D
`)`	[-]	A B * C D + (pop + and discard `(`)
`/`	[-, /]	A B * C D + (/ has higher prec than -)
E	[-, /]	A B * C D + E

Token	Operator Stack (bottom→top)	Output
end	[]	A B * C D + E / - (pop /, then -)

(b) Final postfix expression: `A B * C D + E / -`

(c) Maximum stack depth: 3 symbols (after pushing `-`, `(`, `+` sequentially).

(16) Complexity comparison table

Operation	Unordered Linked List	Dynamic Array (Python list)
Insert at beginning	$O(1)$	$O(n)$ (shift all elements)
Delete a specific element (value given)	$O(n)$ (traverse to find + $O(1)$ relink)	$O(n)$ (find + shift)
Access by index	$O(n)$ (must traverse)	$O(1)$
Search for an element	$O(n)$	$O(n)$

Preferred scenario: An Unordered Linked List is preferred when the application frequently **inserts or deletes at the head/front** of the sequence and **rarely needs index-based access** (e.g., a browser history list, a simple undo stack, or a real-time event log where new events are prepended).

(17) 2D array memory addressing — column-major

Let `addr(Grade[i][j]) = base + α·i + β·j`.

From the given data:

1. `base + 2α + 3β = 252`
2. `base + 5α + 1β = 226`
3. `base + 0α + 2β = 232`

(1) − (3): `2α + β = 20` ...(4)

(2) − (3): `5α − β = −6` ...(5)

(4) + (5): `7α = 14` → **α = 2** (element size)

From (4): **β = 20 − 4 = 16** (column stride)

From (3): **base = 232 − 32 = 200**

(a) Base address = 200

(b) Element size = α = 2 bytes

(c) Number of rows = β / α = 16 / 2 = 8 students (column stride = rows × element size)

(d) `Grade[7][5] = 200 + 2×7 + 16×5 = 200 + 14 + 80 =` **294**

Bonus: Sparse Matrix — Solution

(a) COO format:

```
row_indices = [0, 2, 2, 3]
col_indices = [2, 0, 3, 1]
values      = [3, 7, 2, 5]
```

(b) DOK format: Uses a Python **dictionary** where each key is a `(row, col)` tuple and the value is the non-zero element:

```
{(0,2): 3, (2,0): 7, (2,3): 2, (3,1): 5}
```

(c) Memory comparison:

- Dense 4×4: $16 \times 4 =$ **64 bytes**
- COO (3 arrays × 4 elements × 4 bytes): $3 \times 4 \times 4 =$ **48 bytes** → saves 25%

(d) General condition: COO is more memory-efficient when $\mathrm{NNZ} < \frac{n \times m}{3}$, because COO stores $3 \times \mathrm{NNZ}$ values while the dense format stores $n \times m$ values.