CSC211 · Algorithms and Data Structures

18. Midterm Review

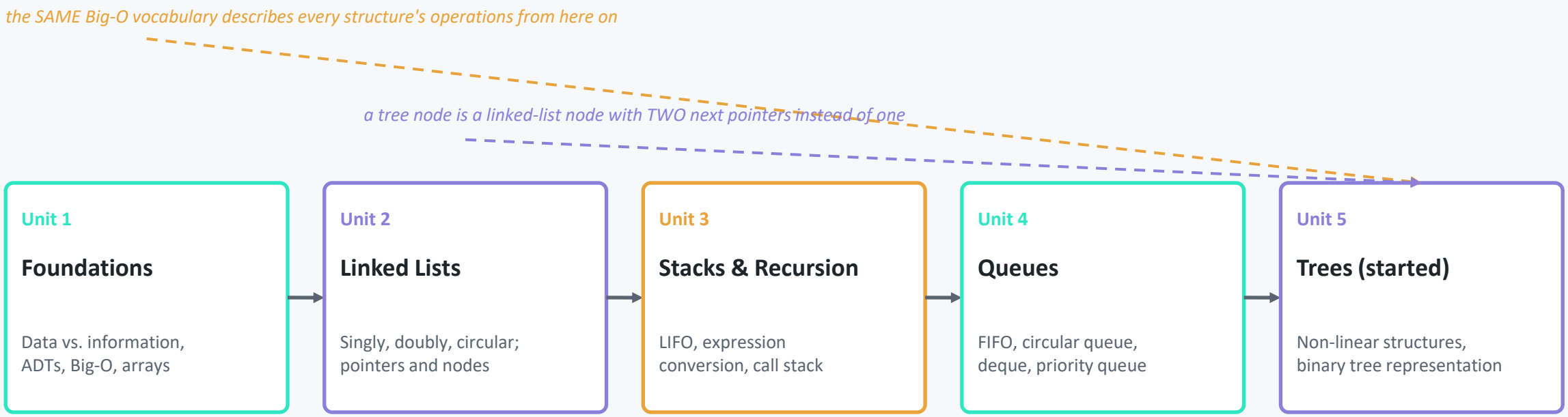
A checkpoint: everything from Unit 1 (foundations) through the start of Unit 5 (trees)

COMSATS University Islamabad, Lahore Campus

What This Review Covers

1. The concept map: how every unit builds on the one before it
2. A compact comparison of every structure covered so far
3. The full complexity cheat sheet, Units 1–5
4. Five self-tests: trace the code, predict the output, then verify
5. Questions to test yourself before the exam

Concept Map: Every Unit Builds on the Last



Two threads worth tracing before the exam: recursion's call stack (Unit 3) is the hidden mechanism behind every depth-first tree traversal (Unit 5); the queue (Unit 4) reappears verbatim as the mechanism behind level-order insertion and traversal.

Neither idea was re-taught in Unit 5 — both were simply reused.

Zooming In: Core Operations Per Unit

Unit 1	Unit 2	Unit 3	Unit 4	Unit 5
Big-O/θ/Ω: best, avg, worst case	Singly LL: O(1) front, O(n) else	Stack (LIFO): O(1) push/pop at TOP	Queue (FIFO): O(1) enqueue/dequeue	Binary tree: up to 2 children/node
Array: O(1) access, O(n) middle insert	Doubly LL: O(1) front/end, walks both ways	Recursion: the call stack IS a stack	Circular queue: reuses freed slots	Traversal: pre/in/post (recursive), level-order (queue)
	Circular LL: no null tail			

Recursion's call stack → drives depth-first traversal. · Queue → drives level-order insertion and traversal.

Array vs. Linked List vs. Stack vs. Queue vs. Tree

Structure	Access	Search	Insert	Delete
Array	O(1)	O(n)	O(n) mid, O(1) end	O(n) mid, O(1) end
Linked list	O(n)	O(n)	O(1) front, O(n) else	O(1) front, O(n) else
Stack	O(1) top only	O(n)	O(1) push	O(1) pop
Queue	O(1) front/rear	O(n)	O(1) enqueue	O(1) dequeue
Binary tree	O(n)	O(n)	O(n)	O(n)

Every entry comes down to one question: *does reaching this position require walking past other elements first, or can you get there directly?*

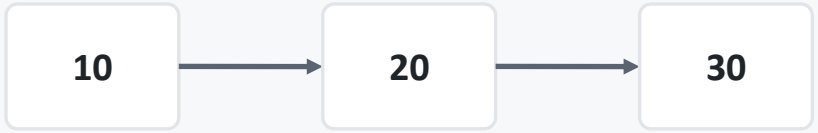
Complexity Cheat Sheet, Units 1–5

Structure	Access	Search	Insert	Delete
Array	O(1)	O(n)	O(n) mid, O(1) end (if space)	O(n) mid, O(1) end
Singly linked list	O(n)	O(n)	O(1) front, O(n) elsewhere	O(1) front, O(n) elsewhere
Doubly linked list	O(n)	O(n)	O(1) front/end, O(n) mid	O(1) front/end, O(n) mid
Circular linked list	O(n)	O(n)	O(1) at known node	O(1) at known node
Stack	O(1) top only	O(n)	O(1) push	O(1) pop
Queue	O(1) front/rear	O(n)	O(1) enqueue	O(1) dequeue
Circular queue	O(1) front/rear	O(n)	O(1), wraps via % capacity	O(1) dequeue
Binary tree (linked)	O(n)	O(n)	O(n) — find next open slot	O(n)
BST (preview, L20)	O(h)	O(h)	O(h)	O(h)

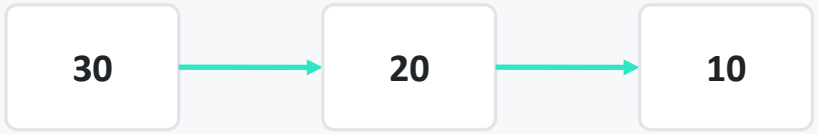
h = tree height. For a roughly balanced BST, $h \approx \log_2 n$ — O(log n). Nothing forces balance; a BST from sorted input degrades to h = n−1, i.e. O(n).

Self-Test: Singly Linked List Reversal (Unit 2)

Before: head → 10 → 20 → 30



After reversal: head → 30 → 20 → 10



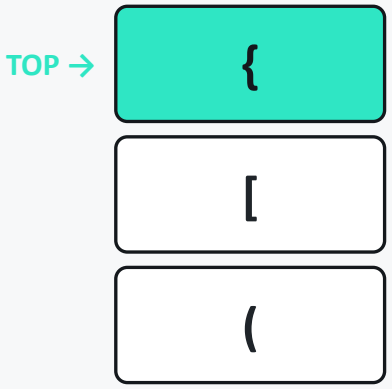
self_test.cpp — in-place reversal (Lecture 6)

```
Node* previous = nullptr;
Node* current = head;
while (current != nullptr) {
    Node* nextNode = current->next;
    current->next = previous;
    previous = current; current = nextNode;
}
head = previous;
```

30 -> 20 -> 10

real output — trace previous / current / nextNode one line at a time

Self-Test: Stack-Based Bracket Matching (Unit 3)



Stack after pushing ([{

```
expression = "([{}])"
```

trace_stack.cpp — core loop

```
for (char c : expression) {
    if (isOpener(c)) brackets.push(c);
    else {
        char opener = brackets.pop();
        if (!matches(opener, c)) balanced = false;
    }
}
```

([{ }]) → Balanced: yes

the most recently opened bracket must be the next one closed — exactly what pop() gives you

Self-Test: Circular Queue Wraparound (Unit 4)

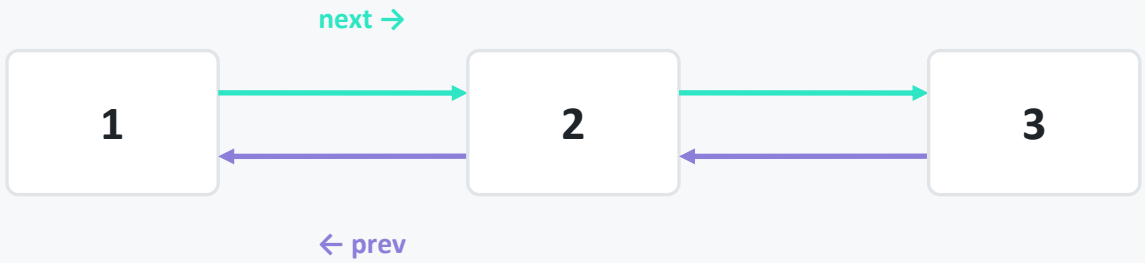
trace_circular_queue.cpp

```
void enqueue(int v) {
    rear = (rear + 1) % capacity;
    data[rear] = v; count++;
}
// after 5 enqueues + 3 dequeues,
// enqueue(6); enqueue(7); wraps to [0],[1]
```

4 5 6 7

dequeue order after the wrap — % capacity reuses slots dequeue() already freed

Self-Test: Doubly Linked List, Both Directions (Unit 2)



Direction	Sequence
Forward	1 2 3 (walk cur = cur->next from head)
Backward	3 2 1 (walk cur = cur->prev from tail)

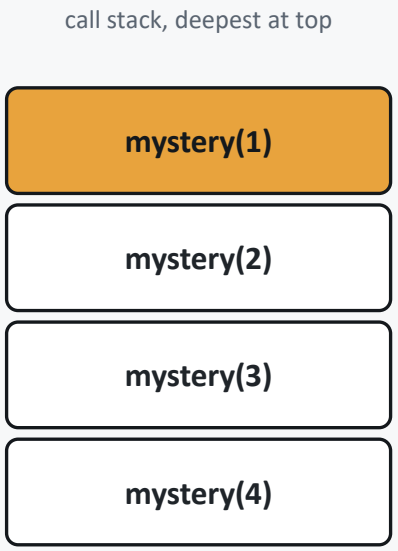
A doubly linked list's prev pointers do what a singly linked list cannot: walk backward from an arbitrary node without ever touching head.

Self-Test: Recursion's Call Order (Unit 3, previews Unit 5)

trace_recursion.cpp

```
int mystery(int n) {
    if (n <= 1) return 1;
    print("entering", n);
    int r = n * mystery(n - 1);
    print("leaving", n, r);
    return r;
}
```

```
entering mystery(4)
entering mystery(3)
entering mystery(2)
leaving mystery(2), returning 2
leaving mystery(3), returning 6
leaving mystery(4), returning 24
```



Every “entering” happens before the recursive call returns; every “leaving” happens after — the same mechanism Lecture 19 uses for pre-order (“entering”) and post-order (“leaving”) traversal.

Questions to Test Yourself

1. Worst-case vs. average-case complexity — which does this course default to?

2. Why is front-insertion O(1) for a linked list but O(n) for an array?

3. A stack and a queue both restrict access to a linear sequence — what's the one-sentence difference?

4. What problem does a circular queue solve that a plain array-based queue doesn't?

5. Full vs. complete vs. skewed binary tree — which shape matches a linked list?

6. Level-order insertion (L17) and level-order traversal (L19) both use a queue — same reason, or two reasons that happen to both need FIFO?

If any of these feel shaky, revisit that lecture before the exam — everything after this point builds directly on these foundations.

Key Takeaways

- Units 1–4 built every linear structure — array, linked list, stack, queue — each a different trade-off between fast access, insertion, and deletion.
- Big-O vocabulary from Unit 1 does not get re-taught, only re-applied — Unit 5 adds one refinement: complexity can depend on shape, not just size.
- Unit 5 has just begun the shift from linear to non-linear structures — a single node can now connect to more than one “next.”
- Two mechanisms get reused, not re-taught, in Unit 5: the queue drives level-order insertion and traversal; the call stack drives every depth-first traversal.
- If any self-test above felt shaky, revisit that lecture before the exam — everything after this point builds directly on these foundations.