LESSON 66

Stacks, Queues, and Linked Lists

80 minutes (2 periods) | Advanced Algorithms and Data Structures | Outcome 2.7 | Outcome 2.6 | Outcome 2.9 | Outcome 2.3 | Outcome 1.22 | Outcome 2.20 | Outcome 1.7

WHAT THIS LESSON DOES

In this lesson, you'll explore essential data structures in computer science. Learn the concepts, operations, and real-world applications of these structures, then implement them in Python using VS Code. By the end, you'll create a simple project to apply your skills.

CURRICULUM OUTCOMES

Outcome	What the specification asks for
2.7	implement algorithms using a programming language to solve a range of problems
2.6	construct algorithms using appropriate sequences, selections/conditionals, loops and operators to solve a range of problems, to fulfil a specific requirement
2.9	assemble existing algorithms or create new ones that use functions (including recursive), procedures, and modules
2.3	implement modular design to develop hardware or software modules that perform a specific function
1.22	read, write, test, and modify computer programs
2.20	identify and fix/debug warnings and errors in computer code and modify as required
1.7	develop algorithms to implement chosen solutions

WHAT STUDENTS SHOULD BE ABLE TO DO

- Understand the fundamental concepts and principles of stacks, queues, and linked lists as key data structures.
- Master the core operations associated with stacks, queues, and linked lists, such as push/pop, enqueue/dequeue, and insert/delete.
- Implement these data structures effectively in Python using appropriate techniques and tools.
- Apply stacks, queues, and linked lists to solve practical problems, such as undo systems or task management.
- Recognise real-world use cases where these data structures provide efficient solutions.

BEFORE THE LESSON

- Confirm VS Code with the Python extension launches and runs a script on the machines students will use.
- Make a DataStructures folder the plan of the lesson expects, since several files (undo_system, my_queue, print_queue, task_manager) live in it.
- Test that collections.deque imports and runs, as the queue implementation relies on it.

Stacks, Queues, and Linked Lists

HOW THE LESSON RUNS

- Introduction
- Understanding Stacks
- Implementing a Stack
- Stack Use Case: Undo System
- Understanding Queues
- Implementing a Queue
- Queue Use Case
- Understanding Linked Lists
- Implementing a Linked List
- Challenge
- Wrap Up

TEACHING MOVES

- 2. Understanding Stacks. Use the stack of plates physically if you can. Say LIFO out loud and have students name the last item they would remove. Anchor push, pop and peek before any code appears.
- 3. Implementing a Stack. Point out that push maps to list append and pop to list pop from the end, and stress why both are fast. Have students predict output before running.
- 4. Stack Use Case: Undo System. Draw the link explicitly: each edit is pushed, undo pops the most recent. Ask what happens if you undo on an empty history before they code the guard.
- 5. Understanding Queues. Contrast directly with the stack just built. First in line served first. Ask students which real systems must be fair, then let them supply their own examples.
- 6. Implementing a Queue. Explain why deque beats a plain list here: popping from the front of a list is slow. This efficiency point is exam-relevant, so make it explicit.
- 7. Queue Use Case. Have them trace three jobs enqueued then dequeued and predict the order printed before running. The output confirms FIFO in a way words do not.
- 8. Understanding Linked Lists. Draw nodes as boxes with arrows on the board. Emphasise that each node holds data plus a reference to the next, and that the last points to None.
- 9. Implementing a Linked List. Build the Node class first and confirm students see it as a separate small class. Walk through traversal following .next until None so the chain is concrete.
- 10. Challenge. Before coding, have students state which structure does which job: queue for pending, stack for undo, linked list for the log. Correct wrong pairings before they type.
- 11. Wrap Up. Ask each student to name one problem best solved by each structure. Use their answers to check the LIFO, FIFO and dynamic-storage distinctions actually landed.

WHAT TO WATCH FOR

- Students treat stacks and queues as interchangeable because both add and remove items. Force a side-by-side trace of the same input through both. Seeing the reversed order from a stack against the preserved order from a queue makes LIFO versus FIFO undeniable.
- Students think a linked list node stores the next node itself rather than a reference to it. On the board, redraw a node and show the arrow pointing at a separate box. Reassigning .next changes the arrow, not the node, which is the whole idea.
- Students call pop or dequeue on an empty structure and are surprised by the crash. Have them run it deliberately, read the error, then add the is_empty guard themselves. The failure teaches why the check exists.

DIFFERENTIATION

- Emerging: Give the stack analogy of plates and have them narrate push and pop aloud before touching code. Pair them with a partner and let them complete only the stack section fully rather than all three.
- Developing: Have them explain in their own words why deque is used for the queue but a list suffices for the stack. Ask them to add a peek method to a structure that lacks one, reasoning from the pattern they have seen.
- Proficient: Push them to compare the time cost of inserting at the front of a Python list versus a linked list, Higher Level territory. Have them extend the TaskManager challenge with a redo feature and document the design as evidence usable in an Applied Learning Task.