Data Structures

Postfix, prefix and bracket matching

The notes for the lesson “Postfix, prefix and bracket matching”: chapters 3.3 to 3.4, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 3.3 to 3.4

About these notes

These are the notes for the University free n easy video lesson **Postfix, prefix and bracket matching** on Data Structures: chapters 3.3 to 3.4, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

3.3 Infix, prefix and postfix conversion

We write arithmetic in **infix** form, operator between operands: A + B. A machine prefers **postfix** (operator after both operands: A B +) or **prefix** (operator before: + A B), because those two forms need no brackets and no precedence rules to be read correctly. Converting infix to postfix is the standard first application of a stack, and the trace is a guaranteed exam question in most syllabi. Prefix is also called Polish notation and postfix reverse Polish notation (RPN).

> You need: 3.1 (push, pop, peek). Expression trees are covered in Unit 4, Expression trees; here we use the tree only as a picture.

Precedence, associativity and why brackets vanish

In infix, which operation happens first depends on **precedence** (^ highest, then * and /, then + and –) and, for equal precedence, on **associativity** (left to right for + – * /; right to left for ^). Brackets override both. In postfix and prefix the position of each operator already says which operands it takes, so neither precedence nor brackets are needed. Figure 3.3.1 shows why: all three forms are the same expression tree read in three orders.

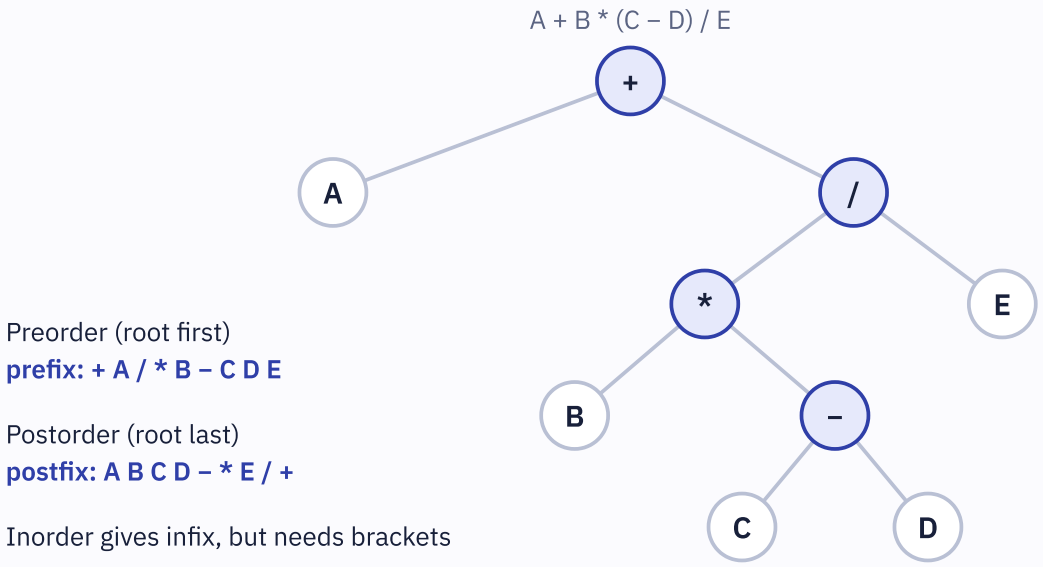


Figure 3.3.1 One expression, one tree, three readings. The operator nodes are shaded. Read root-left-right for prefix, left-right-root for postfix. The brackets of the infix form are only needed because inorder reading loses the tree shape.

Infix to postfix with a stack (the shunting-yard method)

Scan the infix string left to right. Operands go straight to the output; operators wait on a stack until an operator of lower precedence, a closing bracket or the end of input forces them out.

1. **Operand** (A, B, a number): append it to the output.
2. **(**: push it. It acts as a wall; nothing below it can be popped by precedence.
3. **)**: pop and output operators until a (is on top; pop that (and discard it. Brackets never reach the output.
4. **Operator** op: while the stack is not empty, the top is not (, and the top has *higher* precedence than op, or *equal* precedence and op is left-associative: pop the top to the output. Then push op.
5. **End of input**: pop everything to the output. If a (appears here, the brackets were unbalanced.

Each token is pushed and popped at most once, so the conversion is O(n) for n tokens. The only decision in the whole method is step 4, the comparison with the top of the stack.

Infix to prefix

Reverse the infix string, swapping every (with). Run the postfix algorithm with one change in step 4: pop only while the top has *strictly higher* precedence (equal precedence does not pop, so that the final reversal restores left associativity). Reverse the result. The hand method is simpler

for short expressions: fully bracket the infix, then move each operator to the start of its own bracket pair (prefix) or the end (postfix), and delete the brackets.

WORKED EXAMPLE 1: INFIX TO POSTFIX

Question. Convert A + B * (C – D) / E to postfix, showing the stack and output after every token.

Solution. Precedence: * and / are 2, + and – are 1; all left-associative. Stack is written bottom → top.

TOKEN	ACTION	STACK	OUTPUT
A	operand → output		A
+	stack empty, push	+	A
B	operand → output	+	A B
*	top + has lower precedence, push	+ *	A B
(	push	+ * (	A B
C	operand → output	+ * (	A B C
–	top is (, push	+ * (–	A B C
D	operand → output	+ * (–	A B C D
)	pop – to output; pop and discard (	+ *	A B C D –
/	top * equal precedence, left-assoc → pop *; top + lower → push /	+ /	A B C D – *
E	operand → output	+ /	A B C D – * E
end	pop / then +		A B C D – * E / +

Answer: ABCD–*E/+. Check with the tree in Figure 3.3.1: it is the postorder reading.

WORKED EXAMPLE 2: INFIX TO PREFIX

Question. Convert the same expression A + B * (C – D) / E to prefix.

Solution. Reverse and swap brackets: E / (D – C) * B + A. Run the postfix algorithm with "equal precedence does not pop".

TOKEN	ACTION	STACK	OUTPUT
E	output		E
/	push	/	E
(	push	/ (	E
D	output	/ (	E D
–	top is (, push	/ (–	E D
C	output	/ (–	E D C
)	pop –; discard (	/	E D C –
*	top / is equal, do not pop; push	/ *	E D C –
B	output	/ *	E D C – B
+	top * higher → pop; top / higher → pop; push +	+	E D C – B * /
A	output	+	E D C – B * / A
end	pop +		E D C – B * / A +

Reverse the output: **+ A / * B – C D E**. Check by the hand method: fully bracketed, the expression is (A + ((B * (C – D)) / E)); moving each operator to the front of its bracket gives + A / * B – C D E, the preorder reading in Figure 3.3.1.

EXAM TRAPS

- **Equal precedence.** For postfix, an incoming * must pop a waiting * or / (left associativity). For A / B / C the second / must pop the first; pushing instead gives A B C / /, which is A / (B / C) and wrong. The answer is A B / C /.
- **^ is right-associative.** An incoming ^ does *not* pop a waiting ^. 2 ^ 3 ^ 2 is 2 3 2 ^ ^ (= 2^{9} = 512), not 2 3 ^ 2 ^ (= 64).
- **Brackets in the output.** Never. If a (or) appears in your postfix string, step 3 was done wrong.
- **Prefix by "just reverse the postfix".** Wrong: reversing A B + gives + B A, which swaps operands and breaks –, / and ^. Use the reverse-convert-reverse method or the bracket method.
- **Favourite question:** "Convert (A + B) * (C – D) ^ E to postfix with a stack trace", worth 8 marks for the table alone. Answer: A B + C D – E ^ *.

CHECK YOURSELF

– REMEMBER **Write A * B + C in infix, prefix and postfix.** 2 marks

Infix: A * B + C. Prefix: + * A B C. Postfix: A B * C +.

– REMEMBER **What is pushed on the stack during conversion, and what goes directly to the output?** 2 marks

Operators and opening brackets are pushed; operands go directly to the output. Closing brackets are never pushed; they trigger pops.

– UNDERSTAND **Why does postfix notation need no brackets?** 3 marks

In postfix every operator comes immediately after its two operands, so the grouping is fixed by position; the string is the postorder of the expression tree and the tree can be rebuilt uniquely from it. Brackets in infix exist only to override precedence, which postfix never consults.

– APPLY **Convert (A + B) * (C – D) ^ E to postfix with a stack trace.** 8 marks

(push; A out; + push; B out;) pop + → "A B +"; * push; (push; C out; – push; D out;) pop – → "A B + C D –"; ^ higher than * → push; E out → "A B + C D – E"; end: pop ^ then * → **A B + C D – E ^ ***.

– APPLY **Convert A – B – C and A ^ B ^ C to postfix and explain the difference.** 4 marks

A – B – C: left-associative, so the second – pops the first: A B – C –. A ^ B ^ C: right-associative, so the second ^ does not pop the first: A B C ^ ^.

– ANALYZE **A student converts A + B * C to prefix by reversing the postfix A B C * +. What do they get, and is it the same expression?** 4 marks

They get + * C B A, which is C * B + A. Numerically equal here only because + and * commute; for A – B * C the same trick gives – * C B A = C * B – A, a different expression. The correct prefix is + A * B C.

– ANALYZE **What is the maximum number of items on the operator stack while converting ((((A + B)))) and why is this a worst case?** 3 marks

Five: four (plus the +. Nested brackets each add a wall that cannot be popped until its) arrives, so the stack depth equals the nesting depth plus one; this is why the array stack for conversion must be sized for the deepest nesting, not the expression length.

BEYOND THE SYLLABUS

Not for the exam. Dijkstra published the shunting-yard algorithm in 1961 for the first ALGOL compiler, and it is still how calculators and simple interpreters parse. Modern compilers generalise it into **operator-precedence (Pratt) parsing**, which handles unary minus, function calls and user-defined operators. Postfix is the native language of stack machines: the Java Virtual Machine's bytecode, WebAssembly, PostScript printers and the Forth language all execute postfix directly. Search "Pratt parser", "reverse Polish HP calculator" and "WebAssembly stack machine".

3.4 Evaluating postfix expressions and parenthesis matching

Once an expression is in postfix, a single left-to-right scan with one stack computes its value: operands are pushed, and every operator pops two values, computes, and pushes the result. The same "push what is open, pop when it closes" idea checks whether brackets in a program are balanced. Both are short algorithms with long traces, and the trace is what the examiner marks.

You need: 3.1 (push, pop) and 3.3 (what postfix is).

Postfix evaluation

The stack holds *values*, not operators. Because every operator in postfix follows its two operands, by the time you meet the operator its operands are the two most recent values on the stack.

1. Scan the postfix string left to right, one token at a time.
2. Operand: push its value.
3. Operator: pop the top into **b** (right operand), pop again into **a** (left operand), compute **a op b**, push the result.
4. At the end the stack holds exactly one value: the answer. More than one value means an operator is missing; a pop on an empty stack means an operand is missing.

The order in step 3 is the only place to go wrong. For 8 2 – the stack is [8, 2]; the first pop gives b = 2, the second gives a = 8, and the result is a − b = 6, not 2 − 8. Each token is handled once, so evaluation is O(n). Here it is in C for single-digit operands:

```
int evalPostfix(const char *s) {
    int st[100], top = -1;
    for (int i = 0; s[i]; i++) {
        char c = s[i];
        if (c >= '0' && c <= '9') { st[++top] = c - '0'; continue; }
        int b = st[top--], a = st[top--];        /* right first, then left */
        switch (c) {
            case '+': st[++top] = a + b; break;
            case '-': st[++top] = a - b; break;
            case '*': st[++top] = a * b; break;
            case '/': st[++top] = a / b; break;
        }
    }
    return st[top];
}
/* evalPostfix("532*+84/-") returns 9 */
```

A **prefix** expression is evaluated the same way but scanning *right to left*; then the first pop is the left operand a and the second is b. Figure 3.4.1 traces the small expression used in the code.

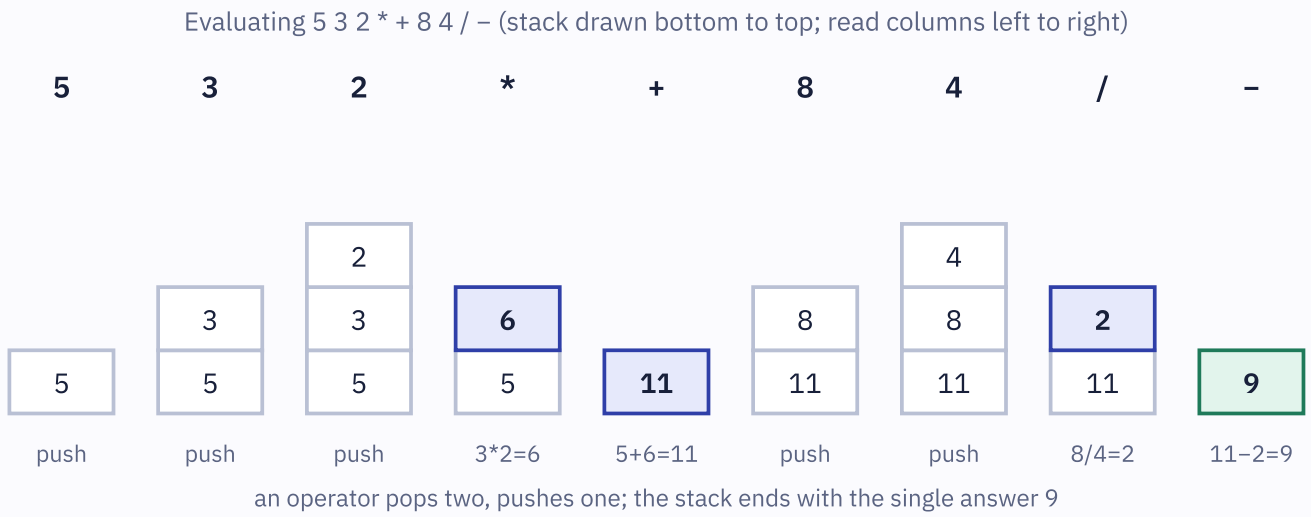


Figure 3.4.1 The value stack after each token of 5 3 2 * + 8 4 / –. Shaded cells are results of an operator. The stack never holds more than three values because the expression nests at most that deep.

Parenthesis matching

A string of brackets is **balanced** when every opening bracket has a matching closing bracket of the same type, in the right nesting order. Compilers do this check before anything else. The rule: an opening bracket is a promise, and the most recent promise must be kept first, which is LIFO.

1. Scan the string left to right. Ignore every character that is not a bracket.
2. Opening bracket ([{: push it.
3. Closing bracket)] }: if the stack is empty, report **unbalanced: extra closing**. Otherwise pop; if the popped bracket is not the matching opener, report **unbalanced: mismatch**.
4. At the end, if the stack is empty the string is balanced; if not, report **unbalanced: unclosed openers**, and the ones left on the stack are exactly the unclosed ones.

The check is O(n) time and, in the worst case of n opening brackets, O(n) extra space. A counter is enough for one bracket type; a stack is needed as soon as there are two types, because a counter cannot tell ([ )] from ([ ]).

WORKED EXAMPLE 1: POSTFIX EVALUATION

Question. Evaluate the postfix expression 6 2 3 + − 3 8 2 / + * 2 ^ 3 + showing the stack after each token.

Solution. Stack written bottom → top. For each operator, b is the first pop and a the second.

TOKEN	ACTION	STACK
6	push	6
2	push	6, 2
3	push	6, 2, 3
+	b=3, a=2, push 2+3=5	6, 5
−	b=5, a=6, push 6−5=1	1
3	push	1, 3
8	push	1, 3, 8
2	push	1, 3, 8, 2
/	b=2, a=8, push 8/2=4	1, 3, 4
+	b=4, a=3, push 3+4=7	1, 7
*	b=7, a=1, push 1×7=7	7
2	push	7, 2
^	b=2, a=7, push 7^2=49	49
3	push	49, 3
+	b=3, a=49, push 52	52

Answer: 52. The infix form is ((6 − (2 + 3)) * (3 + 8 / 2)) ^ 2 + 3 = $(1 \times 7)^2$ + 3 = 52, which confirms the trace.

WORKED EXAMPLE 2: PARENTHESIS MATCHING

Question. Check whether each string is balanced, showing the stack: (i) `{ [ ( ) ] ( ) } [` (ii) `( [ ) ]`.

Solution (i). { push → {; [push → { [; (push → { [(;) pop (matches → { [;] pop [matches → {; (push → { (;) pop (matches → {; } pop { matches → empty; [push → [. End of string with [on the stack: **unbalanced, one unclosed [**.

Solution (ii). (push → (; [push → ([;) pop gives [, which does not match): **unbalanced, mismatch at position 3**. Scanning stops there. A counter would have said "two open, two closed, fine", which is why the stack is needed.

EXAM TRAPS

- **Operand order.** First pop is the right operand. 8 2 – is 6; 8 2 / is 4; 2 3 ^ is 8. Swapping them is the most common lost mark in this chapter.
- **Prefix scans right to left.** And then the first pop is the *left* operand. Do not mix the two conventions in one answer.
- **Multi-digit numbers.** "12 3 +" is 15, not "1 2 3 +". In an exam write tokens separated by spaces; in code, read a whole number before pushing.
- **A non-empty stack at the end means unbalanced.** Students stop after the last character and forget step 4. "((" passes the scan but fails the final check.
- **Favourite question:** "Evaluate 2 3 4 * + 5 – using a stack" (answer 9) or "Write an algorithm to check balanced parentheses and trace it on { ([ ]) }" (balanced).

CHECK YOURSELF

– `REMEMBER` **What does the stack hold during postfix evaluation, and what happens when an operator is read?** `2 marks`

Operand values. On an operator, two values are popped (right operand first), the operation is applied, and the single result is pushed.

– `REMEMBER` **State the three ways a bracket string can fail the balance check.** `3 marks`

A closing bracket arrives when the stack is empty; a closing bracket does not match the popped opener; the stack is not empty when the string ends.

– UNDERSTAND **Why is a single counter not enough to check balance when there are two kinds of brackets?** 3 marks

A counter records only how many are open, not which kind or in what order. ([ )] has the counts right, but the) arrives while [is still the most recent open bracket. The stack keeps the opener types in order, so the mismatch is caught when) pops [.

– APPLY **Evaluate the postfix expression 2 3 4 * + 5 – with a stack trace.** 4 marks

2 → [2]; 3 → [2, 3]; 4 → [2, 3, 4]; * → 3×4 = 12 → [2, 12]; + → 2+12 = 14 → [14]; 5 → [14, 5]; – → 14−5 = 9 → [9]. Answer 9.

– APPLY **Evaluate the prefix expression – + 7 * 4 5 2 scanning right to left.** 4 marks

2 → [2]; 5 → [2, 5]; 4 → [2, 5, 4]; * → pop a=4, b=5, push 20 → [2, 20]; 7 → [2, 20, 7]; + → a=7, b=20, push 27 → [2, 27]; – → a=27, b=2, push 25 → [25]. Answer 25 (infix: 7 + 4×5 – 2).

– ANALYZE **A postfix string ends with two values still on the stack. What is wrong with the string, and can the algorithm tell where?** 4 marks

An operator is missing (there is one operand too many for the operators present). The algorithm cannot say where, because the extra operand could have been inserted at any point; it only detects the count mismatch at the end. A missing operand, by contrast, is detected at the exact operator that tries to pop an empty stack.

BEYOND THE SYLLABUS

Not for the exam. Postfix evaluation is how stack-based virtual machines run: the JVM's `iadd` pops two ints and pushes their sum, exactly step 3 above, and WebAssembly is a typed postfix language. Bracket matching is the simplest case of parsing a context-free language, and the stack is what makes a pushdown automaton stronger than a finite automaton; you will meet that again in Theory of Computation. Editors like VS Code run an incremental version of the matcher to colour bracket pairs as you type. Search "pushdown automaton", "JVM bytecode stack" and "Dyck language".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Infix / prefix / postfix	A + B / + A B / A B +; prefix and postfix need no brackets	3.3
Precedence	^ (right-assoc) > * / > + – (left-assoc); brackets override	3.3
Shunting-yard rule	Pop while top has higher precedence, or equal and operator is left-associative; then push	3.3
Prefix conversion	Reverse infix, swap brackets, convert (equal precedence does not pop), reverse result	3.3
Postfix evaluation	Operand → push; operator → b = pop, a = pop, push a op b; answer = last value	3.4
Prefix evaluation	Scan right to left; first pop is the left operand	3.4
Balanced brackets	Push openers; on a closer pop and match; balanced iff no failure and stack empty at the end	3.4

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

– **Compare infix, prefix and postfix notation. Why are prefix and postfix preferred by machines? Illustrate with A * (B + C). (8 marks)**

Definitions of the three forms with the example: infix A * (B + C), prefix * A + B C, postfix A B C + *. Machines prefer prefix/postfix because operator position fixes the grouping: no precedence table, no associativity rule, no brackets; a single scan with a stack evaluates them in O(n). Relate the three forms to preorder, inorder and postorder of the expression tree (Unit 4).

Apply

- **Convert the infix expression (A + B) * C – (D – E) * (F + G) to postfix and to prefix, showing the stack contents at every step. (12 marks)**

  Postfix trace ends with A B + C * D E – F G + * –. Prefix: reverse with brackets swapped: (G + F) * (E – D) – C * (B + A); convert with "equal precedence does not pop": G F + E D – * C B A + * –; reverse: – * + A B C * – D E + F G. Marks: stack table for postfix (6), correct prefix with method (6). Check both against the fully bracketed form ((A + B) * C) – ((D – E) * (F + G)).

- **Evaluate the postfix expression 5 6 2 + * 12 4 / – using a stack. Show the stack after every token. Then write a C function that evaluates a postfix string of single-digit operands. (12 marks)**

  5 → [5]; 6 → [5, 6]; 2 → [5, 6, 2]; + → [5, 8]; * → [40]; 12 → [40, 12]; 4 → [40, 12, 4]; / → [40, 3]; – → [37]. Answer 37. Code: array stack, loop over characters, push digits, on operator pop b then a and push a op b, return the remaining value (as in 3.4). Marks for the trace (5), operand order (2), code with the correct pop order and error checks (5).

- **Write an algorithm to check whether the brackets in an expression are balanced. Trace it on { ([ ]) (}) and on [( ) { }]. (10 marks)**

  Algorithm from 3.4: push openers; on a closer, report failure if the stack is empty or the popped opener does not match; at the end report failure if the stack is not empty. First string: { ([ ]) (→ stack { (; } pops (→ mismatch → unbalanced. Second string: every closer matches and the stack ends empty → balanced. Marks for the algorithm (4), each trace with stack contents (3 + 3).

Analyze and evaluate

- **Why does A ^ B ^ C convert to A B C ^ ^ while A – B – C converts to A B – C –? Show what goes wrong numerically if the rule is applied the other way for 2 ^ 3 ^ 2 and 10 – 5 – 2. (8 marks)**

  Associativity: ^ groups right to left, so the second ^ does not pop the first; – groups left to right, so it does. Wrong grouping: 2 3 ^ 2 ^ = 8^2 = 64 instead of 2^9 = 512; 10 5 2 – – = 10 – 3 = 7 instead of (10 – 5) – 2 = 3. Marks for the rule, the two traces and the numerical check.

Glossary (terms used in this lesson)

Associativity	Grouping order for operators of equal precedence: left to right for + – * /, right to left for ^.

Front	The deletion end of a queue; also the operation that returns that item without removing it.
Infix	Operator between its operands, as in A + B; needs precedence rules and brackets.
LIFO	Last in, first out: the order of a stack.
Polish notation	Another name for prefix notation; reverse Polish notation is postfix.
Pop	Remove and return the top item of a stack.
Postfix	Operator after its operands, as in A B +; evaluated left to right with a stack.
Precedence	Priority of operators: ^ above * and /, above + and –.
Prefix	Operator before its operands, as in + A B; evaluated right to left with a stack.
Push	Insert an item on the top of a stack.
Shunting-yard algorithm	Dijkstra's stack method for converting infix to postfix.
Top	The single access end of a stack; in an array stack, the index of the last item.

Sources and further reading

Textbooks

- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, 2nd edition. Chapter 3 (Lists, Stacks, and Queues) covers both ADTs, their array and linked implementations, and postfix conversion and evaluation.
- Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd edition. The chapter on stacks and queues includes the circular queue, expression evaluation and the multiple-stacks problem.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*. Chapter 10, Elementary data structures: stacks and queues as arrays, with pseudocode and the amortised two-stack queue in the exercises.
- Reema Thareja, *Data Structures Using C*. Widely prescribed; chapters on stacks, queues and their applications with many traced examples.
- Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *Data Structures and Algorithms*. Basic data types chapter: stacks, queues and mappings.

Open web resources (opened while writing)

- Wikipedia, "Shunting yard algorithm": en.wikipedia.org/wiki/Shunting_yard_algorithm. Dijkstra's 1961 method, with the precedence and associativity rules used in chapter 3.3.
- Python documentation, `collections.deque`: docs.python.org/3/library/collections.html. A production deque with O(1) appends and pops at both ends and an optional maxlen that behaves like a bounded circular buffer.
- cppreference, `std::stack`: en.cppreference.com/w/cpp/container/stack. The C++ stack is a container adaptor over a deque, which is why "stack built on a deque" appears in chapter 3.1.

Syllabi: the universal syllabus and a range of university course documents.

Built for the University free n easy YouTube channel from the universal syllabus of Data Structures (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.