Data Structures

Stacks: the ADT, arrays and linked lists

The notes for the lesson “Stacks: the ADT, arrays and linked lists”: chapters 3.1 to 3.2, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 3.1 to 3.2

About these notes

These are the notes for the University free n easy video lesson **Stacks: the ADT, arrays and linked lists** on Data Structures: chapters 3.1 to 3.2, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

3.1 Stack ADT: push, pop and peek

A stack is a list where you may add and remove only at one end, called the **top**. Whatever went in last comes out first, so a stack is a **LIFO** (last in, first out) structure. It exists because many jobs naturally reverse order: undoing edits, returning from function calls, matching brackets, backing out of a maze. Every compiler, every browser's back button and every recursive program depends on one.

You need: the idea of an abstract data type (an earlier lesson, Abstract data types) and array indexing (an earlier lesson, One- and two-dimensional arrays).

The ADT: what a stack promises

An abstract data type lists the operations and what they do, not how they are stored. Some syllabi call this the "Stack ADT"; others simply say "stacks" under linear ADTs. The stack ADT has five operations on a stack S:

- **push(S, x)**: put item x on the top. The stack grows by one.
- **pop(S)**: remove the top item and return it. The stack shrinks by one.
- **peek(S)** (also called **top(S)**): return the top item without removing it.
- **isEmpty(S)**: true when the stack holds nothing.

- **isFull(S)**: true when a bounded stack has no room left (only meaningful for array stacks).

Two error states follow. **Underflow** is a pop or peek on an empty stack; **overflow** is a push on a full one. A correct implementation checks for both before touching the data. Every operation takes constant time, O(1), because only the top is ever touched; no shifting, no searching.

How the operations behave together

Think of a pile of plates in a hostel mess. New plates go on top and the next plate taken is the top one. If you push 10, then 20, then 30, the pile reads 10 at the bottom and 30 at the top. A pop removes 30 and returns it. Push 40 and the pile reads 10, 20, 40. Figure 3.1.1 shows exactly these three moments.

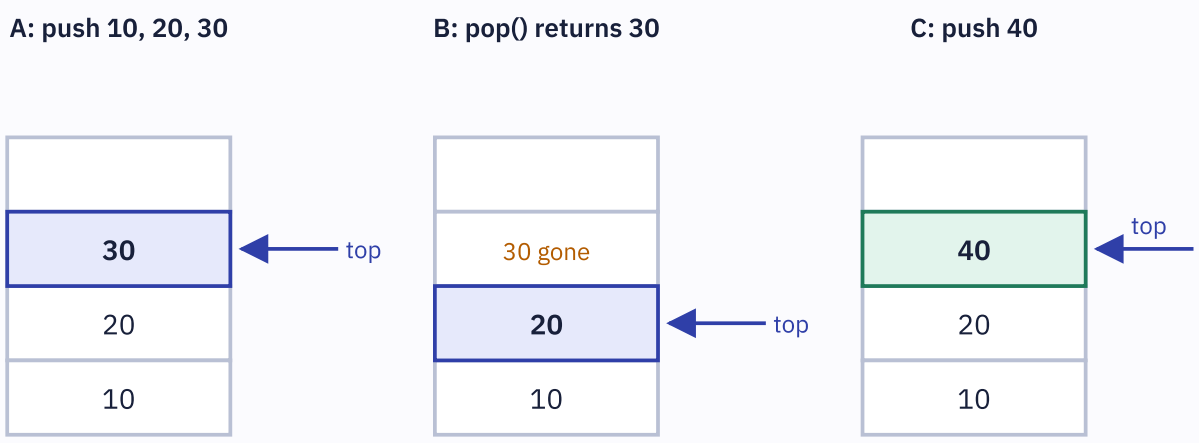


Figure 3.1.1 The same stack at three moments. Only the top cell changes; 10 and 20 are never touched. The top pointer moves up on push and down on pop, which is why every operation is O(1).

Where stacks are used

Some syllabi list the **function call stack** as an application, and it is the one you use without noticing. When a C program calls a function, the machine pushes an **activation record** (return address, parameters, local variables) on the run-time stack; when the function returns, that record is popped. Recursion is just many pushes before the first pop. For `fact(3)` the records for fact(3), fact(2), fact(1) are pushed in that order, then popped in reverse: fact(1) returns 1, fact(2) returns 2, fact(3) returns 6. Too deep a recursion overflows this stack, which is the "segmentation fault" you see with a missing base case.

Other uses, each worked in the chapters that follow: converting and evaluating arithmetic expressions (3.3, 3.4), checking balanced brackets (3.4), undo in editors, the back button in a

browser, and depth-first search of a graph (a later lesson, Breadth-first and depth-first traversal).

WORKED EXAMPLE

Question. A stack of capacity 5 starts empty. Perform the operations in order: push(7), push(3), pop(), push(9), peek(), push(2), push(5), pop(), pop(), isEmpty(). Show the stack contents (bottom to top), the value returned, and the index of the top after each operation (top = −1 when empty).

Solution. Track three things per step. The stack is written bottom → top.

STEP	OPERATION	STACK (BOTTOM → TOP)	RETURNS	TOP
0	start	(empty)	—	−1
1	push(7)	7	—	0
2	push(3)	7, 3	—	1
3	pop()	7	3	0
4	push(9)	7, 9	—	1
5	peek()	7, 9	9	1
6	push(2)	7, 9, 2	—	2
7	push(5)	7, 9, 2, 5	—	3
8	pop()	7, 9, 2	5	2
9	pop()	7, 9	2	1
10	isEmpty()	7, 9	false	1

Final answer: the stack holds 7, 9 with top = 1; the pops returned 3, 5, 2 in that order; peek returned 9 without changing anything. Notice that the stack never reached capacity 5, so no overflow occurred, and no pop was attempted on an empty stack, so no underflow.

EXAM TRAPS

- **peek is not pop.** peek returns the top but leaves the stack as it is. Students write the stack one item shorter after a peek and lose the rest of the trace.
- **Underflow vs overflow.** Pop on empty = underflow; push on full = overflow. The examiner's favourite one-mark question is "what happens when you pop an empty stack?"
- **Which end is the top?** In the trace, write the stack bottom → top and say so. Reversing it silently makes every pop look wrong.
- **Order of returned values.** Push 1, 2, 3 then pop three times returns 3, 2, 1. A stack *reverses* the input order; that is the whole point.
- **"Is every push/pop sequence possible?"** No. Pushing 1, 2, 3 in order can never produce the pop order 3, 1, 2: once 3 is popped, 2 is on top of 1 and must come out before 1.

CHECK YOURSELF

– REMEMBER **Name the five operations of the stack ADT and state the time complexity of each.** 2 marks

push, pop, peek (top), isEmpty, isFull. Each is O(1) because only the top element is touched.

– REMEMBER **What do underflow and overflow mean for a stack?** 2 marks

Underflow: pop or peek attempted on an empty stack. Overflow: push attempted on a full (bounded) stack.

– UNDERSTAND **Why is a stack called LIFO, and why does recursion need one?** 3 marks

Insertion and deletion both happen at the top, so the last item inserted is the first removed. Each recursive call must save where to resume and its local values; the calls finish in reverse order of starting, which is exactly LIFO, so the activation records are pushed and popped on a stack.

– APPLY **Starting empty: push(4), push(8), push(15), pop(), push(16), pop(), pop(), push(23). Give the final contents and the sequence of popped values.** 4 marks

Trace: 4 → 4,8 → 4,8,15 → pop 15 → 4,8,16 → pop 16 → pop 8 → 4,23. Final stack (bottom → top): 4, 23. Popped values in order: 15, 16, 8.

– APPLY **Items 1, 2, 3, 4 are pushed in that order, with pops allowed at any time. Can the pop sequence be 2, 4, 3, 1? Can it be 4, 1, 2, 3?** 4 marks

2, 4, 3, 1: yes. push 1, push 2, pop 2, push 3, push 4, pop 4, pop 3, pop 1. 4, 1, 2, 3: no. To pop 4 first, 1, 2, 3 must all be on the stack below it, so after 4 the next pop must be 3, not 1.

– ANALYZE **A text editor supports Undo with a stack of edits. What happens to Redo, and how would you support both?** 5 marks

Undo pops the last edit and reverses it. To allow Redo, push the undone edit onto a second stack (the redo stack); Redo pops from it and re-applies, pushing back onto the undo stack. Any new edit clears the redo stack, because the redo history is no longer valid. Two stacks give both operations in O(1).

BEYOND THE SYLLABUS

Not for the exam. In practice you rarely write a stack from scratch: C++ gives `std::stack` (an adaptor over a deque), Java gives `ArrayDeque` (preferred over the old synchronised `Stack` class), and Python uses a plain list with `append` and `pop`. The function call stack you met here is where buffer-overflow attacks live: writing past a local array can overwrite the saved return address, which is why compilers add stack canaries and operating systems randomise stack addresses. Search "stack canary", "ArrayDeque vs Stack Java" and "tail call optimisation" to see where this topic goes next.

3.2 Array and linked list implementation of stacks

The stack ADT of 3.1 can be stored two ways. An **array stack** keeps the items in a fixed array and remembers the index of the top; it is fast and compact but has a capacity. A **linked stack** keeps the items as nodes and makes the head of the list the top; it grows as long as memory lasts but pays for a pointer per item. Both give O(1) push, pop and peek. Exams ask you to write both in C and to say when each is the better choice.

You need: 3.1 for the operations; an earlier lesson (Singly linked lists: insertion, deletion and traversal) for nodes and `malloc`.

Array implementation

Keep an array `a[MAX]` and an integer `top`. The convention used in most Indian textbooks is **top = −1 for empty**, and `a[top]` is the top item. push first increments top, then stores; pop first reads, then decrements. isFull is `top == MAX − 1`. Some syllabi list this as "Push, Pop, Peek: array implementation"; others call the two operations "insertion and deletion".

```
#define MAX 4
int a[MAX];
int top = -1;

int isEmpty(void) { return top == -1; }
int isFull(void)  { return top == MAX - 1; }

void push(int x) {
    if (isFull()) { printf("Overflow\n"); return; }
    a[++top] = x;                /* increment, then store */
}
int pop(void) {
    if (isEmpty()) { printf("Underflow\n"); return -1; }
    return a[top--];             /* read, then decrement */
}
int peek(void) {
    if (isEmpty()) { printf("Underflow\n"); return -1; }
    return a[top];
}
```

This is C. The order inside push and pop matters: `a[++top]` moves to the new slot before writing; `a[top--]` reads the current slot and only then moves down. Nothing is erased on pop; the old value stays in memory but is now above the top and is treated as garbage.

Linked list implementation

Use a singly linked list and make the **head the top**. Pushing is insertion at the head and popping is deletion at the head, both O(1) with no traversal. If you made the tail the top, pop would need to walk the whole list to find the new tail, O(n), so never do that with a singly linked list. Some syllabi list this as "linked stacks".

```
struct node { int data; struct node *next; };
struct node *top = NULL;         /* empty stack */

void push(int x) {
    struct node *p = malloc(sizeof *p);
    if (p == NULL) { printf("Overflow (no memory)\n"); return; }
    p->data = x;
    p->next = top;               /* new node points to old top */
    top = p;                     /* new node becomes the top   */
}
int pop(void) {
    if (top == NULL) { printf("Underflow\n"); return -1; }
    struct node *p = top;
    int x = p->data;
    top = top->next;             /* unlink the first node */
```

```
    free(p);
    return x;
}
int peek(void) { return top ? top->data : -1; }
```

Two lines carry the whole logic. In push, `p->next = top` must come *before* `top = p`, or the old list is lost. In pop, save `top` in `p` before moving `top`, or you cannot free the node.

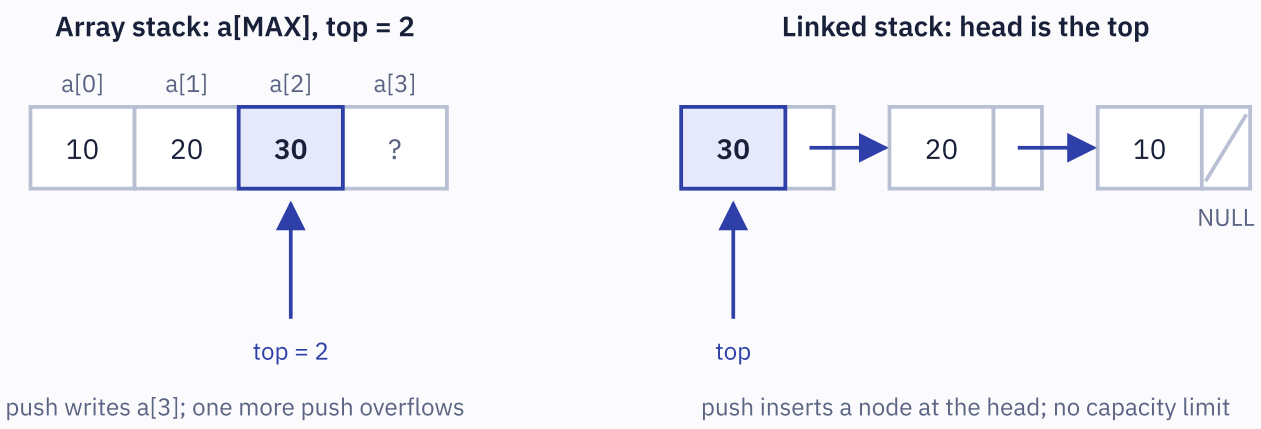


Figure 3.2.1 The same stack (10, 20, 30 with 30 on top) in both representations. In the array, top is an index and the cell above it is unused. In the linked version, top is a pointer to the first node, and the bottom item 10 is the last node, whose next is NULL.

Choosing between them

PROPERTY	ARRAY STACK	LINKED STACK
push / pop / peek	O(1)	O(1)
Capacity	Fixed at MAX; overflow possible	Limited only by heap memory
Memory per item	Just the item	Item + one pointer (8 bytes on 64-bit)
Unused memory	Empty slots above top are wasted	None; nodes exist only for stored items
Cost per operation	Index arithmetic; cache-friendly	malloc and free on every push and pop
Best when	Maximum size is known (expression evaluation, DFS on a known graph)	Size varies widely or is unknown

WORKED EXAMPLE

Question. Using the array implementation with MAX = 4, dry-run: push(5), push(8), push(2), push(9), push(4), pop(), pop(), peek(), pop(), pop(), pop(). Show top and the array after each step, and the output printed.

Solution. top starts at −1. isFull is true when top = 3.

STEP	CALL	CHECK	ACTION	A[0..3]	TOP	OUTPUT
1	push(5)	top=−1, not full	a[0]=5	5 _ _ _	0	
2	push(8)	not full	a[1]=8	5 8 _ _	1	
3	push(2)	not full	a[2]=2	5 8 2 _	2	
4	push(9)	not full	a[3]=9	5 8 2 9	3	
5	push(4)	top=3 = MAX−1, full	nothing	5 8 2 9	3	Overflow
6	pop()	not empty	return a[3]	5 8 2 (9)	2	9
7	pop()	not empty	return a[2]	5 8 (2) (9)	1	2
8	peek()	not empty	return a[1]	5 8 (2) (9)	1	8
9	pop()	not empty	return a[1]	5 (8) (2) (9)	0	8
10	pop()	not empty	return a[0]	(5) (8) (2) (9)	−1	5
11	pop()	top=−1, empty	nothing	(5) (8) (2) (9)	−1	Underflow

Values in brackets are still in memory but above top, so they are garbage. Printed output, in order: Overflow, 9, 2, 8, 8, 5, Underflow. The value 4 was never stored. With the linked implementation the same sequence would print 9, 2, 8, 8, 5, Underflow; the fifth push would succeed because there is no MAX.

EXAM TRAPS

- `a[++top]` **vs** `a[top++]`**.** With top = −1 for empty, push must pre-increment. Post-increment writes to a[−1] on the first push, which is undefined behaviour in C.
- **isFull test.** With top starting at −1, full is `top == MAX − 1`, not `top == MAX`. If your book starts top at 0 (pointing to the next free slot), full is `top == MAX`; state which convention you use.
- **Order of pointer assignments in linked push.** Writing `top = p` before `p->next = top` makes the node point to itself and loses the rest of the stack.
- **Linked stack never overflows?** It does when `malloc` returns NULL. Say "overflow only when memory is exhausted"; do not say "never".
- **Favourite question:** "Write C functions for push and pop using a linked list and explain why the head is used as the top." The answer is the O(1) argument above.

CHECK YOURSELF

– REMEMBER **In the array implementation with top = −1 for empty, write the conditions for isEmpty and isFull.** 2 marks

isEmpty: top == −1. isFull: top == MAX − 1.

– UNDERSTAND **Why is the head of the linked list chosen as the top of a linked stack?** 3 marks

Insertion and deletion at the head of a singly linked list take O(1): no traversal is needed. Using the tail as the top would make pop O(n), since the node before the tail must be found to become the new tail.

– UNDERSTAND **What does pop do to the memory of the popped element in each implementation?** 2 marks

Array: nothing; the value stays in the cell but is above top, so it is logically removed. Linked: the node is unlinked and freed, so the memory is returned to the heap.

– APPLY **Linked stack, starting empty: push(11), push(22), push(33), pop(), push(44). Draw the final list from top and state what pop returned.** 4 marks

After the three pushes: top → 33 → 22 → 11 → NULL. pop returns 33 and frees that node: top → 22 → 11 → NULL. push(44): top → 44 → 22 → 11 → NULL. pop returned 33.

– APPLY **An array stack of int with MAX = 1000 is used for a job that usually holds 10 items. How many bytes are reserved, and how much is typically wasted (4-byte int)? Would a linked stack be better?** 4 marks

Reserved: 1000 × 4 = 4000 bytes. Typically used: 10 × 4 = 40 bytes, so about 3960 bytes wasted. A linked stack with 10 nodes uses 10 × 16 = 160 bytes (4-byte int + 8-byte pointer, padded to 16 on a 64-bit machine) plus allocator overhead, so it is better on memory here, at the cost of a malloc per push.

– ANALYZE **A student's linked pop first moves top to the next node and only then calls free on top. What goes wrong?** 4 marks

After `top = top->next`, top already points to the second node, so `free(top)` frees the wrong node (the new top) and leaks the old one. The stack is then corrupt. Fix: save the old top in a temporary pointer, move top, then free the temporary.

BEYOND THE SYLLABUS

Not for the exam. Real libraries use a third design: a **dynamic array** that doubles its capacity when full (C++ `std::vector`, Java `ArrayList`, Python list). Push is then O(1) *amortised*: a rare O(n) copy is paid for by the many cheap pushes before it. It keeps the array's cache-friendliness and drops the fixed limit, which is why linked stacks are rare in production code. The Linux kernel and embedded systems still use fixed array stacks where allocation is forbidden. Search "amortized analysis dynamic array" and "lock-free stack ABA problem" to go further.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Stack (LIFO)	Insert and delete at one end, the top; push, pop, peek all O(1)	3.1
Underflow / overflow	pop or peek on empty / push (enqueue) on full	3.1, 3.5
Array stack	top = −1 empty; push: a[++top] = x; pop: return a[top−−]; full: top == MAX − 1	3.2
Linked stack	Head is the top; push = insert at head; pop = delete at head	3.2

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.

Analyze and evaluate

- **Show how a queue can be implemented using two stacks. Give the enqueue and dequeue algorithms and analyse their cost. Then show how a stack can be built from two queues. (12 marks)**

  Queue from stacks S1 (in) and S2 (out): enqueue pushes on S1, O(1). Dequeue: if S2 is empty, pop everything from S1 and push onto S2, then pop S2. Each element moves at most once from S1 to S2, so dequeue is O(1) amortised, O(n) worst case. Stack from queues Q1, Q2: push enqueues on Q1 (O(1)); pop dequeues all but the last element of Q1 into Q2, returns the last, and swaps the names, O(n). Spans 3.1, 3.2 and 3.5; marks for correct algorithms, a worked trace of three operations each, and the amortised argument.

- **Trace the run-time stack for the recursive C function fact(4). What is the maximum stack depth, and what happens for fact(100000)? Relate this to the array stack of chapter 3.2. (10 marks)**

  Calls push activation records fact(4), fact(3), fact(2), fact(1), fact(0); base case returns 1; records pop in reverse, multiplying: 1, 1, 2, 6, 24. Maximum depth is 5 records (n + 1). fact(100000) pushes 100001 records and can exceed the fixed run-time stack size, giving a stack overflow / segmentation fault: the run-time stack is an array stack with a fixed MAX, exactly the overflow of 3.2. Mention the iterative or tail-recursive alternative.

Glossary (terms used in this lesson)

Activation record	The data pushed on the run-time stack for one function call: return address, parameters, locals.
Balanced brackets	Every opener has a matching closer of the same type, properly nested.
Deque	Double-ended queue: insertion and deletion allowed at both ends.
LIFO	Last in, first out: the order of a stack.
Overflow	Insertion attempted on a full structure.

Peek (top)	Return the top item of a stack without removing it.
Pop	Remove and return the top item of a stack.
Push	Insert an item on the top of a stack.
Top	The single access end of a stack; in an array stack, the index of the last item.
Underflow	Deletion attempted on an empty structure.

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.