Data Structures

Data structures and ADTs

The notes for the lesson “Data structures and ADTs”: chapters 1.1 to 1.2, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

`2 chapters` `Chapters 1.1 to 1.2`

About these notes

These are the notes for the University free n easy video lesson **Data structures and ADTs** on Data Structures: chapters 1.1 to 1.2, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

1.1 What a data structure is; linear vs non-linear

A data structure is a way of arranging data in memory together with the operations that use that arrangement. The arrangement is the whole point: the same 60 marks stored in an array, a linked list or a tree are the same data, but finding the 40th mark costs one address calculation in the array, 40 pointer hops in the list and about six comparisons in a balanced tree. Choosing a structure is choosing which operations will be cheap. Some syllabi list this chapter as "Classification of Data Structures: Primitive and Non-Primitive, Linear and Non-Linear"; others call it "Introduction to data structures".

Data, information and data types

Data is a raw value: 72, 'A', 3.14, "Chennai". Information is data with a meaning attached: 72 is a mark out of 100, 'A' is a grade. A program never stores a bare value; it stores a value of a **data type**, and a data type is a set of allowed values plus the operations allowed on them. In C, `int` is the whole numbers that fit in 4 bytes with +, −, ×, ÷ and comparisons; `char` is one byte holding a character code; `float` and `double` approximate real numbers; a pointer holds an address.

These built-in types are the **primitive** data types. Everything built out of them is **non-primitive**: an array (many values of one type side by side), a structure or record (values of different types under one name), a string, a file, and every structure in this book. Primitive/non-primitive is the

first classification examiners ask for; linear/non-linear is the second, and it applies only to the non-primitive side.

What a data structure is

Formally, a data structure is a collection of data values, the relationships among them, and the operations that can be applied to them. In practice it has two halves: a **storage structure** (how the values sit in memory: one block, scattered nodes joined by pointers, a table of slots) and a set of **operations** (insert, delete, search, traverse, sort) whose cost depends on that storage. An array keeps its elements in one contiguous block, so element i is at address base + i × size and is reached in one step; but inserting at the front means shifting every element right. A linked list scatters its nodes and joins them with pointers, so inserting at the front is one pointer change, but reaching element i means walking i pointers.

So the design question is always the same: which operations does this problem do most, and which structure makes those cheap? A railway reservation system searches by PNR millions of times a day; an editor's undo history only adds and removes at one end; a road map must answer "what is next to this junction?". Each gets a different structure. Another lesson gives the language to ask the question; later lessons give the structures that answer it.

Linear and non-linear structures

A structure is **linear** when its elements form a single sequence: every element has at most one predecessor and at most one successor, so the whole structure can be traversed in one pass, first to last, without ever choosing a direction. Arrays, linked lists, stacks and queues are linear. A stack and a queue are linear with restricted access (one end only, or in at one end and out at the other), which is why a later lesson treats them as ADTs built on arrays or lists.

A structure is **non-linear** when an element can be related to many others. A tree is a hierarchy: one root, and every other node has one parent and any number of children, like folders inside folders. A graph is a network: any node may be joined to any other, like cities joined by roads, cycles allowed. Traversing a non-linear structure forces a choice at each step (which child first? which neighbour next?), so a later lesson teaches tree traversals and a later lesson teaches breadth-first and depth-first search.

Three other classifications cut across this line: **contiguous vs linked** storage (array vs linked list; a tree can be stored either way), **static vs dynamic** size (fixed at compile time vs grown at run time, chapter 1.3) and **homogeneous vs heterogeneous** contents (an array holds one type, a record holds several).

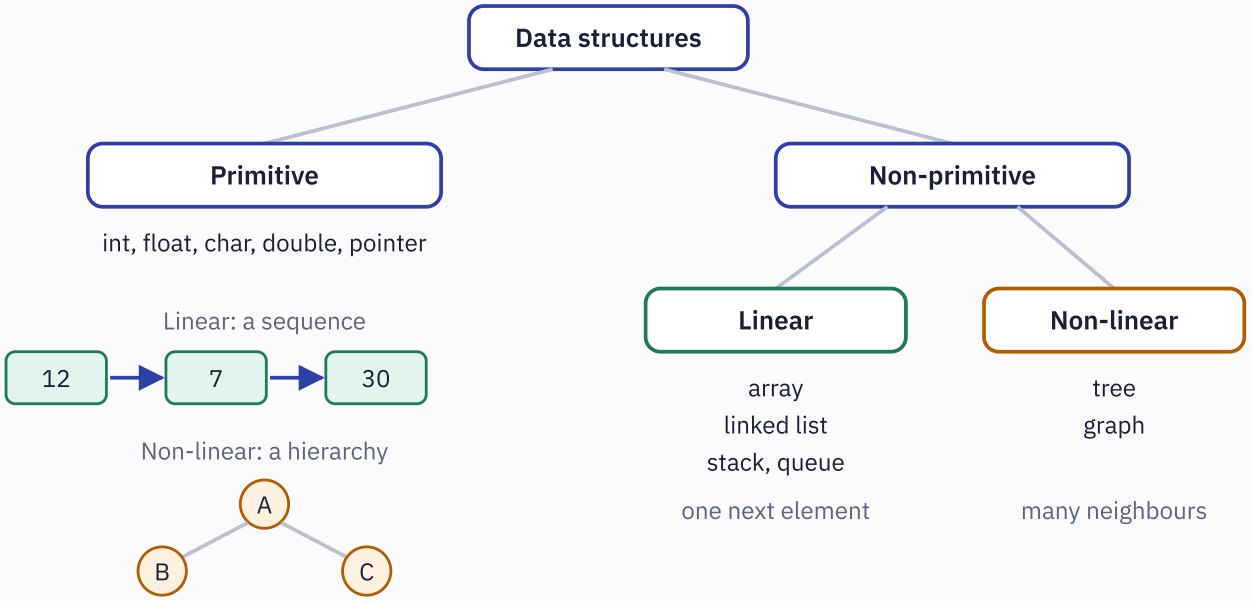


Figure 1.1.1 The two classifications examiners ask for. Primitive types are the building blocks; non-primitive structures split into linear (every element has one next element, as in 12 → 7 → 30) and non-linear (an element can have many neighbours, as A has children B and C). Notice that stack and queue sit under linear: they are sequences with restricted access.

WORKED EXAMPLE

Question. (a) For each situation, say whether a linear or a non-linear structure fits and name the structure: (1) the undo history of a text editor; (2) print jobs waiting at a hostel printer; (3) the folders and files on a laptop; (4) the Chennai Metro map with its interchange stations; (5) the marks of 60 students in roll-number order. (b) The marks in (5) are stored as a C array of `int` starting at address 5000. Where is the mark of roll number 41, and how many steps does reaching it take?

Solution. (a) Ask what each situation does most and in what order.

SITUATION	DOMINANT OPERATIONS	CLASS	STRUCTURE
Undo history	add the latest action, take back the latest action	linear	stack (last in, first out)
Print jobs	add at the back, serve from the front	linear	queue (first in, first out)
Folders and files	one folder holds many items, each item has one parent	non-linear	tree
Metro map	a station joins several others, loops exist	non-linear	graph
60 marks by roll number	read or update mark i directly	linear	array

(b) Roll number 41 is index 40 when roll numbers start at 1 and indices at 0. With `sizeof(int)` = 4 bytes, address = base + index × size = 5000 + 40 × 4 = 5000 + 160 = **5160**. The address takes one multiplication and one addition whatever the index, so reaching any mark costs a constant number of steps: the "direct access" that makes arrays the right choice when items are looked up by position. A linked list would walk 40 links for the same lookup.

EXAM TRAPS

- "Linear" does not mean "stored in one block". A linked list is linear (one next element) but its nodes are scattered; an array is both linear and contiguous. Keep linear/non-linear (the shape of the relationships) separate from contiguous/linked (the storage).
- A stack and a queue are linear, however they are drawn. They restrict where you may insert and delete; they do not change the one-after-another shape.
- A skewed tree in which every node has one child looks like a list but is still a tree (non-linear): the relationship is parent-to-children, not previous-to-next.
- Examiners' favourite: "Define data structure. Classify data structures with examples." Give the definition (data + relationships + operations), then the tree of Figure 1.1.1 with two examples per leaf.

CHECK YOURSELF

– REMEMBER **Define a data structure in one sentence and name the two halves every definition contains.** 2 marks

A data structure is an organisation of data values in memory, with the relationships among them, together with the operations applied to them. The two halves are the storage structure (how values are placed) and the operations (insert, delete, search, traverse) whose cost depends on that placement.

– REMEMBER **List four linear and two non-linear data structures.** 2 marks

Linear: array, linked list, stack, queue. Non-linear: tree, graph.

– UNDERSTAND **Why is a linked list called linear even though its nodes are not next to each other in memory?** 3 marks

Linear describes the relationships, not the addresses: each node has exactly one successor (its next pointer) and at most one predecessor, so the nodes form one sequence traversed in a single pass. Where the nodes physically sit is the contiguous-vs-linked question, a separate classification.

– UNDERSTAND **Distinguish a data type from a data structure with one example each.** 3 marks

A data type is a set of values and the operations on them, fixed by the language: `int` with +, −, ×, ÷. A data structure organises many values of some type and adds operations whose cost depends on the organisation: an array of `int` with direct access by index, or a linked list of `int` with cheap insertion at the front.

– APPLY **An array of 200 `double` values (8 bytes each) begins at address 10000. Give the address of index 125 and the number of steps to reach it.** 3 marks

Address = 10000 + 125 × 8 = 10000 + 1000 = 11000. One multiplication and one addition: a constant number of steps regardless of the index.

– ANALYZE **A student claims a binary tree in which every node has only a right child should be reclassified as linear because it "is just a list". Evaluate the claim.** 4 marks

The claim confuses one instance's shape with the structure's definition. Classification follows the defining relationship: a tree relates parent to children (possibly many), so a degenerate instance with one child per node is still a tree with tree operations. It does behave like a list in cost (a search walks every node), which is why a later lesson introduces balanced (AVL) trees to prevent such shapes. Classification: still non-linear; the cost observation: correct.

BEYOND THE SYLLABUS

Not for the exam. In practice you rarely write these structures from scratch: C++ gives you `std::vector` and `std::map`, Java the Collections framework, Python `list`, `dict` and `set`; the skill is knowing which one your workload needs. Performance work goes one level further: a CPU fetches memory in cache lines of 64 bytes, so a contiguous array is often faster than a linked list even where the list "should" win, and game and database engineers design layouts around this ("data-oriented design"). Functional languages add persistent structures that keep every old version after an update. Search for "cache-friendly data structures" and "persistent data structures".

1.2 Abstract data types (ADTs)

An abstract data type is a data type described by what its operations do, not by how its values are stored. A List ADT promises get, insert, remove and size with stated effects; whether the list lives in an array or in linked nodes is an implementation decision that the user of the list never sees. The idea exists so that the "what" can be fixed, taught and tested once, while the "how" is swapped freely for speed or memory. Most syllabi list this as "Abstract Data Types (ADTs)"; one syllabus words it as "introduction to ADTs and linear ADTs, for example sets, lists, vectors and sequences", and some tutorials ask you to "design simple ADTs", which is the worked example below.

You need: the definition of a data type and the linear/non-linear classification from chapter 1.1.

Specification, not implementation

Every data type is two things: a set of values and a set of operations. An **abstract** data type writes down only the part a user can observe: the operations, their inputs and outputs, and the rules that connect them. It says nothing about memory. The built-in `int` is already an ADT in this sense: you use +, −, × and comparisons and never think about the two's-complement bits underneath. The ADT idea applies the same discipline to the structures you build yourself.

A good specification has four parts. The **domain**: what the values are (a sequence of integers; a set of strings). The **operations** with their **signatures**: name, arguments, result type. The **preconditions**: what must be true before an operation may be called (get(i) needs $0 \le i < \text{size}$). The **postconditions**: what is true afterwards (after insert(i, x), size has grown by one and get(i) returns x). Think of a UPI app: pay(to, amount) has a precondition (enough balance) and a postcondition (both balances changed); you never see the bank's tables. That boundary, interface on one side and implementation on the other, is what the word "abstract" means here.

The ADTs this course is built from

Almost every structure in later lessons is one of five ADTs. The table gives the operations by their usual names; later chapters implement them.

ADT	VALUES	CORE OPERATIONS	IMPLEMENTED IN
List (sequence)	ordered items $a_0 \dots a_{(n-1)}$	get(i), set(i, x), insert(i, x), remove(i), size()	A later lesson: arrays, linked lists
Stack	a list with access at one end	push(x), pop(), top(), isEmpty()	A later lesson
Queue	a list with insert at the rear, remove at the front	enqueue(x), dequeue(), front(), isEmpty()	A later lesson
Set	distinct items, no order	add(x), remove(x), contains(x), size()	A later lesson (search trees), a later lesson (hashing)
Map (dictionary)	key → value pairs	put(k, v), get(k), remove(k)	A later lesson, a later lesson

Notice what the table does not say: nothing about pointers, indices or memory. A Stack is defined by the rule "pop returns the item pushed most recently", and both an array with a top index and a chain of nodes satisfy that rule.

One interface, many implementations

The payoff of an ADT is that the same client code runs on different implementations. Client code written against the List ADT calls `insert(0, x)`; it does not know that an array implementation shifts n elements to make room while a linked implementation changes one pointer. If the array version turns out too slow for a workload full of front insertions, the implementation is replaced and the client does not change by a character. Standard libraries work this way: Java's `List` is an interface with `ArrayList` and `LinkedList` behind it. Chapter 1.9 shows the mechanism in code: a class packages an ADT's operations with a hidden representation.

Choosing the implementation is an analysis question, which is why chapters 1.5 to 1.8 follow. The ADT fixes the meaning; the data structure fixes the cost.

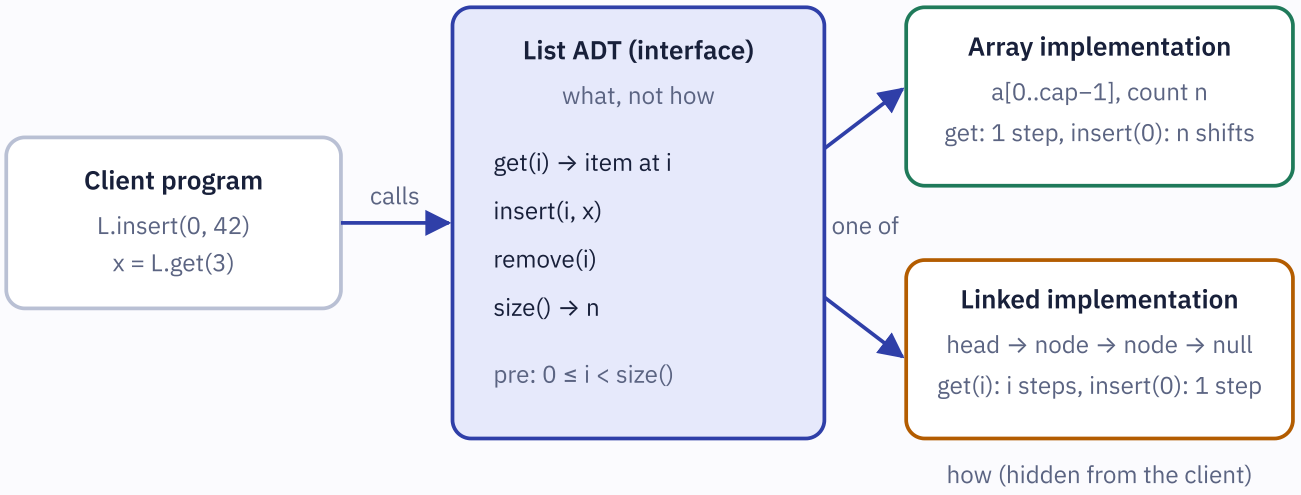


Figure 1.2.1 The wall an ADT builds. The client sees only the interface in the middle: operation names, results and preconditions. Either implementation on the right can sit behind it, and swapping one for the other changes the costs (bottom line of each box) but not a line of client code.

WORKED EXAMPLE

Question. (a) Write the specification of an IntList ADT with operations create, size, get, insert and remove. (b) Starting from an empty list, trace: insert(0, 5); insert(1, 9); insert(0, 2); get(1); remove(1); size(). (c) The list holds n = 1000 items. Compare the cost of insert(0, x) and of get(500) under an array implementation and a linked implementation.

Solution. (a) Specification (pseudocode; any clear notation is accepted):

```
ADT IntList
  Values:  a sequence a0, a1, ..., a(n-1) of integers, n >= 0
  create()      -> an empty list, n = 0
  size()        -> n
  get(i)        requires 0 <= i < n;  returns a(i); list unchanged
  insert(i, x)  requires 0 <= i <= n; afterwards a(i) = x, old a(i)..a(n-1)
                move to positions i+1..n, and n increases by 1
  remove(i)     requires 0 <= i < n;  returns a(i); a(i+1)..a(n-1) move
                to positions i..n-2, and n decreases by 1
```

(b) Trace, checking each precondition against the current n:

CALL	PRECONDITION	LIST AFTER	N	RETURNS
create()	—	[ ]	0	—
insert(0, 5)	0 ≤ 0 ≤ 0 ✓	[5]	1	—
insert(1, 9)	0 ≤ 1 ≤ 1 ✓	[5, 9]	2	—
insert(0, 2)	0 ≤ 0 ≤ 2 ✓	[2, 5, 9]	3	—
get(1)	0 ≤ 1 < 3 ✓	[2, 5, 9]	3	5
remove(1)	0 ≤ 1 < 3 ✓	[2, 9]	2	5
size()	—	[2, 9]	2	2

(c) Array: insert(0, x) must move a_0 … a_{999} one place right, 1000 moves, then write x: about 1001 steps. get(500) is one address calculation: about 1 step. Linked list: insert(0, x) creates a node and changes the head pointer: about 2 steps whatever n is. get(500) walks 500 links: about 500 steps. Same ADT, same trace in (b), opposite costs: the array wins by a factor of 500 on get and loses by a factor of 500 on insert at the front.

EXAM TRAPS

- ADT versus data structure: the ADT is the specification (operations and their meaning); the data structure is a concrete implementation (array, linked nodes). "A linked list is an ADT" is wrong; "List is an ADT, a linked list implements it" is right.
- "Abstract" does not mean vague. A specification must state every operation's inputs, outputs and preconditions; an answer that lists only operation names gets half marks.
- What happens when a precondition is broken (get(7) on a list of 3) is the implementation's business, usually an error; an ADT answer should state the precondition, not invent a return value like −1.
- An ADT is not the same as an "abstract class" in C++ or Java. The language feature is one way to express an interface; the ADT is the mathematical idea and exists in C too, where it is a header file plus functions.
- Examiners' favourite: "Define ADT. Specify the List (or Stack) ADT and give two implementations with their trade-offs." Use the four-part template, then the cost comparison of (c).

CHECK YOURSELF

– REMEMBER **Define an abstract data type and name the four parts of a good specification.** 2 marks

An ADT is a data type defined by its values and the behaviour of its operations, independent of any implementation. The four parts: domain of values, operation signatures, preconditions, postconditions.

– REMEMBER **Name the core operations of the Stack ADT and of the Map ADT.** 2 marks

Stack: push(x), pop(), top(), isEmpty(). Map: put(key, value), get(key), remove(key).

– UNDERSTAND **Why does using an ADT let an implementation be replaced without changing the programs that use it?** 3 marks

Client programs are written against the interface only: operation names, results and preconditions. Any implementation that satisfies the same specification is interchangeable, because nothing in the client refers to the representation (array index, pointer). Only the costs change.

– UNDERSTAND **In what sense is the built-in `int` type an ADT?** 2 marks

Programs use integers through operations (+, −, ×, comparisons) with known meaning and never touch the representation (two's-complement bits, byte order). The representation is hidden behind the operations, exactly the ADT discipline.

– APPLY **Specify a Set-of-strings ADT with add, remove, contains and size, including preconditions, then trace add("a"), add("b"), add("a"), contains("a"), remove("b"), size().** 4 marks

Values: a collection of distinct strings. add(x): afterwards x is in the set; if it already was, nothing changes. remove(x): requires x in the set (or specify "no effect if absent"); afterwards x is not in the set. contains(x) → true/false. size() → number of distinct strings. Trace: {a} → {a, b} → {a, b} (duplicate ignored) → contains("a") = true → {a} → size() = 1.

– ANALYZE **A team writes a Queue ADT but lets client code read the internal array index "front" directly "for speed". Analyse what is lost.** 4 marks

The interface boundary is broken: clients now depend on the representation (an array with a front index), so the queue can no longer be reimplemented as a linked structure or a circular buffer without rewriting every client. Bugs can also corrupt the queue's invariants, since clients can move "front" outside the operations that keep it consistent. Any speed gain is usually illusory, as a front() operation costs the same single read. Lost: substitutability, invariants, testability; the fix is to expose front() in the interface.

BEYOND THE SYLLABUS

Not for the exam. In industry the ADT idea appears as interfaces (Java, Go), abstract base classes and Protocols (Python), traits (Rust) and concepts (C++20), and tests written against the interface are what let teams replace an implementation safely. Formal methods go further: Z, Alloy and TLA+ write preconditions and postconditions precisely enough for a tool to check them, and "design by contract" checks them at run time. Do not confuse ADTs with the "algebraic data types" of Haskell and Rust, which build types from sums and products. Search for "design by contract" and "TLA+ specification".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Data structure	data values + the relationships among them + the operations on them; storage decides cost	1.1
Linear / non-linear	one predecessor and one successor (array, list, stack, queue) / many neighbours (tree, graph)	1.1
Array address	address of a[i] = base + i × element size (constant-time access)	1.1
ADT	values + operations specified by behaviour: domain, signatures, preconditions, postconditions	1.2

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.

Remember and understand

- **1. Define a data structure. Classify data structures as primitive and non-primitive, and the non-primitive ones as linear and non-linear, with two examples of each and one application for every example. (8 marks)**

  Definition: data values, their relationships, and the operations on them, with storage deciding cost. Tree of Figure 1.1.1: primitive (int, float, char, pointer); non-primitive linear (array: marks by roll number; linked list: playlist; stack: undo; queue: print jobs) and non-linear (tree: folder hierarchy; graph: road or metro map). State the one-predecessor-one-successor test for linear and the many-neighbours property for non-linear; one line distinguishing contiguous from linked storage earns the last mark.

- **2. What is an abstract data type? Write the specification of the List ADT and explain, with costs, how an array-based and a linked implementation differ while serving the same specification. (10 marks)**

  ADT = values + operations defined by behaviour, independent of representation; four parts of a specification. List spec as in chapter 1.2 (create, size, get, insert, remove with preconditions $0 \le i < n$ or $\le n$). Figure 1.2.1: the client uses only the interface. Costs for n items: array get(i) constant, insert(0, x) n shifts; linked get(i) up to i steps, insert at front constant. Conclude: same trace of operations, different costs; choice is an analysis question.

Analyze and evaluate

- **12. A phone-contacts app must look up a name very frequently, add a contact occasionally, and list all contacts in alphabetical order on demand. Choose a data structure class, specify the ADT the app needs, propose two implementations and compare their operation costs with Big-O. (16 marks; spans chapters 1.1, 1.2, 1.5, 1.6)**

  Operations: search(name) dominant, insert(name, number) occasional, listSorted(). ADT: a Map from name to number with those three operations and preconditions (search on absent name returns not-found). Implementation A: sorted array; search by binary search O(log n), insert O(n) shifts, listing O(n). Implementation B: unsorted linked list; search O(n), insert O(1) at front, listing needs a sort O(n log n). Since search dominates, A wins; mention that a later lesson's search trees and a later lesson's hashing improve insert while keeping search fast. Marks for naming linear vs non-linear choices, the ADT spec, the cost table and the justified recommendation.

- **15. A student stores a linked list's nodes in a static array of 10,000 node structures "to avoid malloc". Evaluate the design against dynamic allocation on memory use, flexibility, speed and failure behaviour, for lists of 50 and of 12,000 nodes. (10 marks; spans chapters 1.1 and 1.3)**

  Memory: the static pool reserves 10,000 nodes always (for 50 nodes almost all wasted); dynamic allocates exactly what is used. Flexibility: 12,000 nodes cannot fit the static pool at all, dynamic grows until the heap is exhausted. Speed: the static pool avoids allocator calls and may be faster and more cache-friendly, which is why real systems use pools for fixed workloads. Failure: static pool fails silently when full unless checked; malloc returns NULL. Verdict: a pool is a legitimate technique when the maximum is known; as a way to "avoid malloc" for unknown sizes it fails.

Glossary (terms used in this lesson)

Data structure	An organisation of data in memory together with the relationships among the values and the operations applied to them.
Linear structure	A structure whose elements form one sequence, each with at most one predecessor and one successor.
Non-linear structure	A structure in which an element may be related to many others, as in a tree or a graph.
Precondition	What must be true before an operation may be called; postcondition: what is true after it.
Heap	The region of memory from which malloc, calloc and realloc serve requests.

Sources and further reading

- Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd edition. Chapter 1 (basic concepts: algorithm specification, data abstraction, performance analysis) is the source of the step-count tables and the fixed-plus-variable space formula used here.
- Mark Allen Weiss, *Data Structures and Algorithm Analysis in C*, 2nd edition, Pearson. Chapter 2 (algorithm analysis) for the notations and the rules for loops; Chapter 3 for the List, Stack and Queue ADTs.
- Michael T. Goodrich, Roberto Tamassia and Michael H. Goldwasser, *Data Structures and Algorithms in Python*, Wiley. Chapters on object-oriented programming, algorithm analysis and recursion; the book behind the "ADTs and classes" wording of some syllabi.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, 4th edition, 2022. Chapters 2 to 4 for running times, asymptotic notation and recurrences.
- Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd edition, Addison-Wesley. Section 1.1 for the five characteristics of an algorithm and Euclid's algorithm.
- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd edition, Prentice Hall, for malloc, free and the storage model of C.
- Reema Thareja, *Data Structures Using C*: the introductory chapters match the classification and memory-allocation wording of several Indian syllabi.
- cppreference, realloc: what happens to the old block, when it moves, and the NULL-on-failure rule used in chapter 1.3.

- Python documentation, copy: shallow and deep copy operations: the definitions of shallow and deep copy quoted in chapter 1.9.
- Python documentation, sys: getrecursionlimit and setrecursionlimit: the interpreter's recursion depth limit mentioned in chapter 1.8.
- ACM / IEEE-CS / AAAI, Computer Science Curricula 2023 (CS2023), the current international curriculum referred to in the "Beyond the syllabus" boxes.

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.