LECTURE 2

Classification, Operations, and Abstract Data Types

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

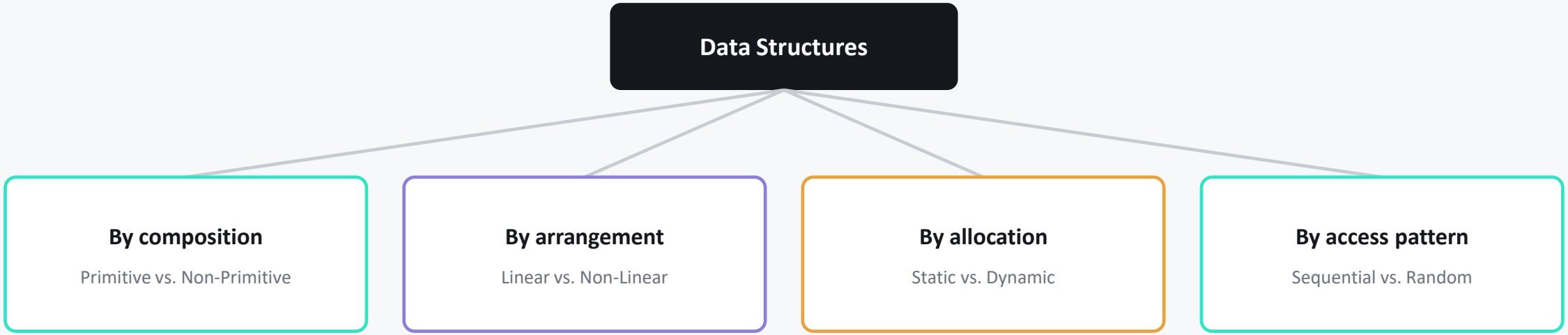
AGENDA

Learning Objectives

1. Six criteria used to classify any data structure
2. The standard operations every data structure exposes in some form
3. What an Abstract Data Type (ADT) is, and why interface and implementation are separate
4. The SAME ADT, implemented two different ways, proving that separation is real
5. Abstraction and encapsulation, and how C++ actually enforces them
6. The trade-offs behind every data structure design choice

THE LANDSCAPE

Classifying Any Data Structure



A std::vector and a linked list are both LINEAR, but one is static-ish (contiguous, resizes in chunks) and the other is genuinely DYNAMIC (grows one node at a time). Every data structure sits somewhere on ALL of these axes at once — there's no single “type.” A fifth axis, ephemeral vs. persistent, is covered separately next.

THE LANDSCAPE

Composition and Arrangement

Primitive

- Built-in types the language gives for free
- int, char, float, bool
- Each holds exactly one value

Non-Primitive

- Built FROM primitives
- array, linked list, stack, queue, tree, graph
- Combine many values into one organized whole

Linear

- Elements form a sequence
- Each element has one predecessor, one successor
- array, linked list, stack, queue

Non-Linear

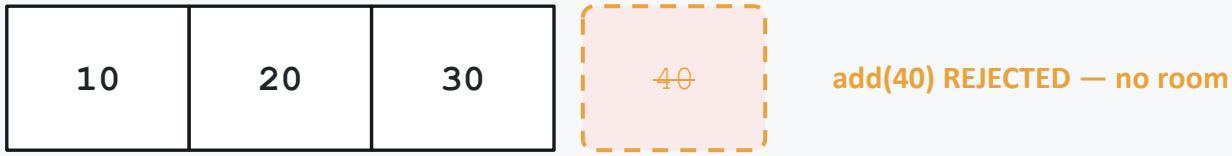
- An element can connect to MORE than one other
- A tree node can have several children
- A graph vertex can have many neighbors

THE LANDSCAPE

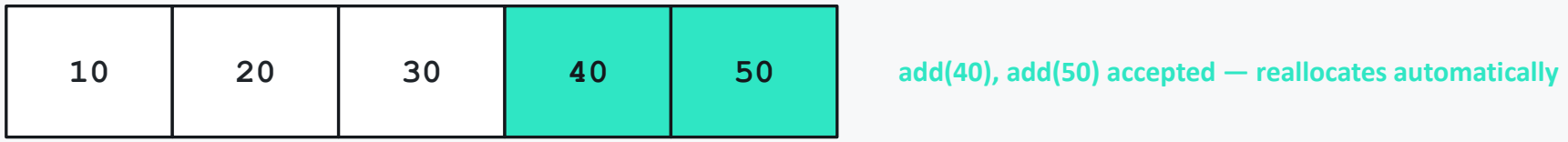
Static vs. Dynamic Structures

A static structure has a fixed size decided at creation. A dynamic structure grows and shrinks while the program runs.

SmallArrayBag — static, capacity fixed at 3



GrowableBag — dynamic, no fixed ceiling

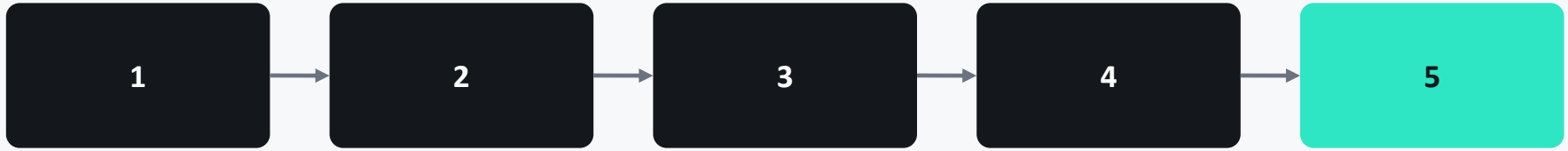


Same 5 add() calls through the same ADT interface — completely different behavior underneath. A fixed-capacity structure is simple with zero bookkeeping, but hits a hard wall; a dynamic one avoids the wall but pays a small ongoing reallocation cost.

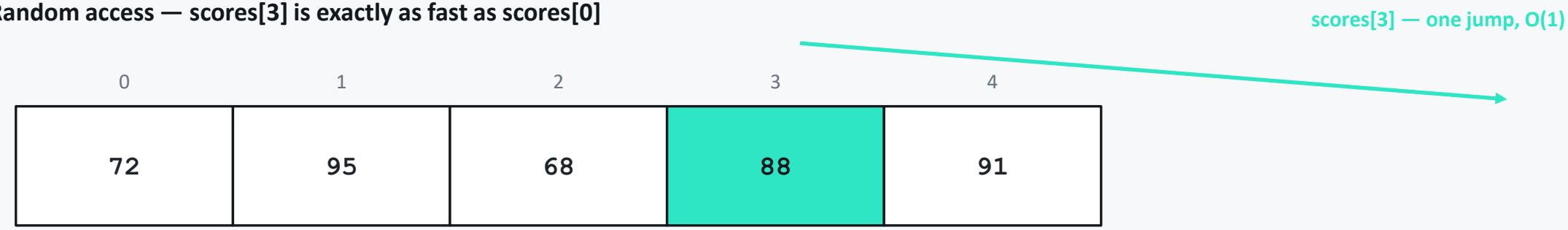
THE LANDSCAPE

Sequential vs. Random Access

Sequential access — reaching element 5 means visiting 1–4 first



Random access — scores[3] is exactly as fast as scores[0]

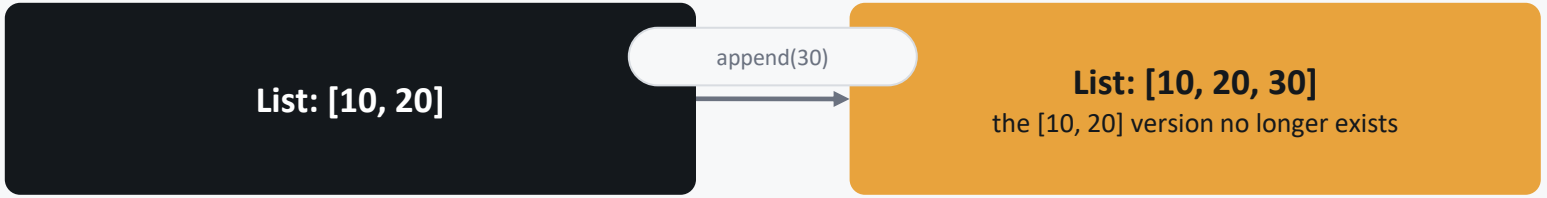


A linked list only supports sequential access — the only way to node 5 is through nodes 1–4. An array supports random access — the address formula (Lecture 4) computes any index directly.

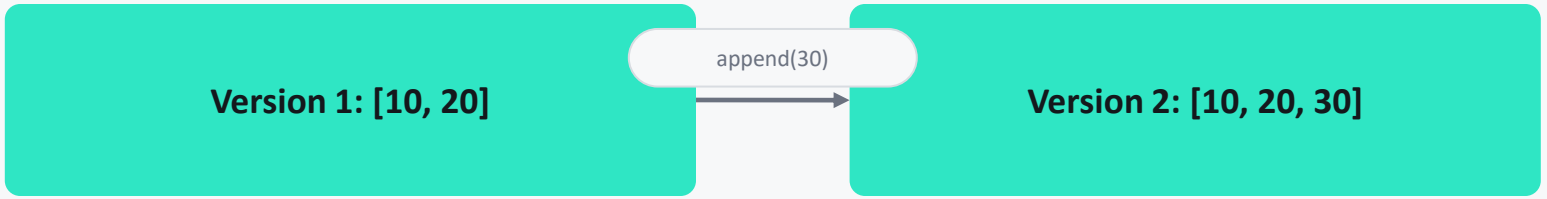
THE LANDSCAPE

Ephemeral vs. Persistent

Ephemeral — modifying overwrites the only copy



Persistent — modifying creates a new version, the old one survives



Version 1 is still reachable — nothing was overwritten

Almost every structure you build this course is ephemeral. Persistent structures preserve every previous version — the idea behind an editor's full undo history or Git's commit history.

WHAT YOU CAN DO

Operations on Data Structures

Operation	Meaning
Traversal	Visit every element, typically to process or display each one
Insertion	Add a new element
Deletion	Remove an existing element
Searching	Find whether (and where) a given value exists
Sorting	Rearrange elements into a defined order
Updating	Modify the value of an existing element
Merging	Combine two data structures of the same type into one

Every lecture from here on answers the same question for a new structure: how efficiently can it perform each of these, and which is it actually good at?

CORE CONCEPT

Abstract Data Type (ADT)

An ADT is a mathematical, logical description of a data structure — **what** operations it supports and what they do — with no mention of **how** it's built underneath.

Example: the Stack ADT

push
Add an item to the top.

pop
Remove the most recently pushed item.

peek
Look at the top item without removing it.

“A Stack supports push, pop, and peek, and pop always removes the most recently pushed item” is a COMPLETE ADT description. Nothing says whether it's backed by an array or a linked list — that's the entire point.

CORE CONCEPT

Interface and Implementation

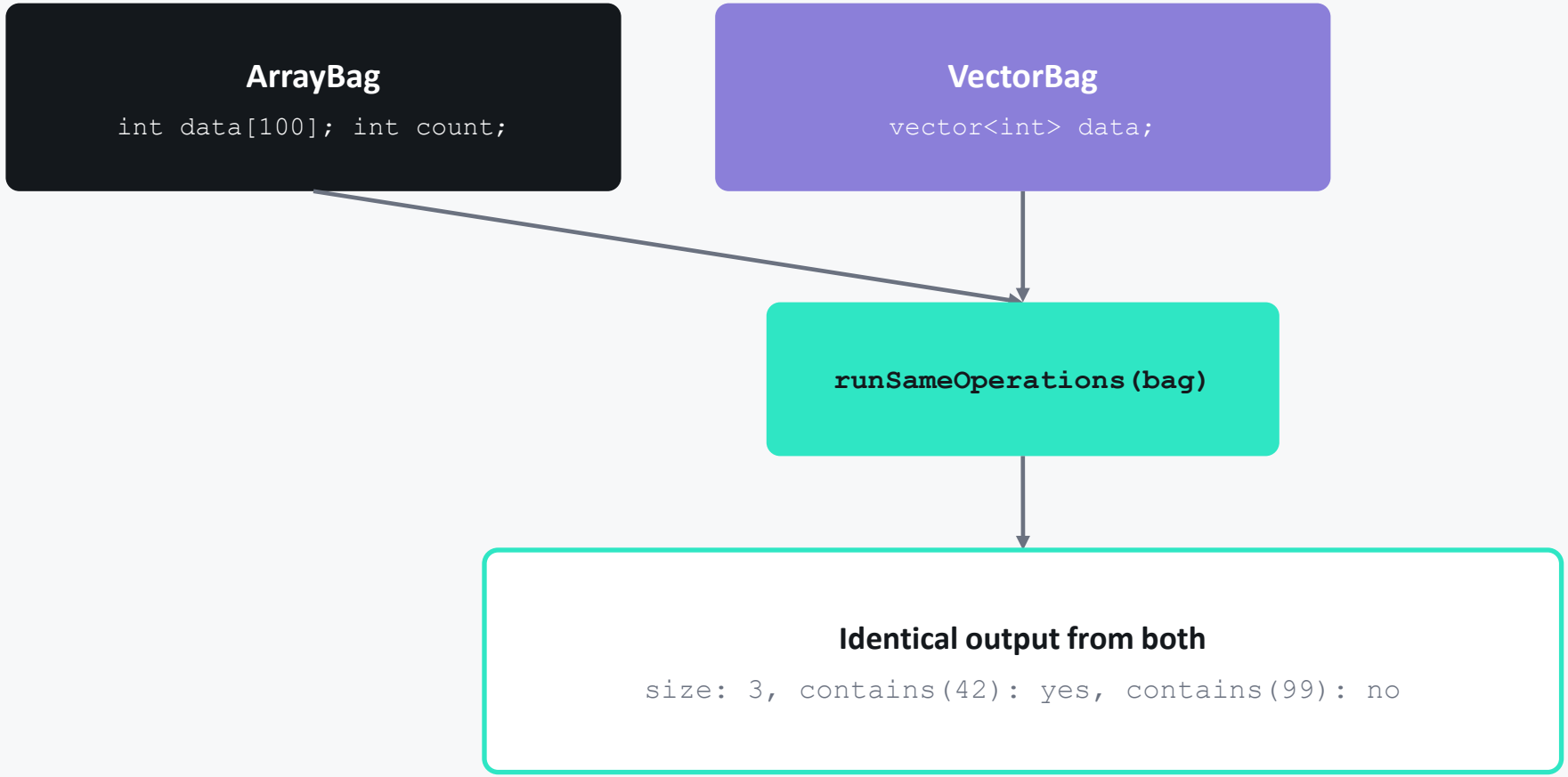
The interface is the ADT's public promise. The implementation is the actual code that makes it work — C++'s class keeps them visibly separate.

int_bag_demo.cpp (excerpt)

```
class IntBag {
public:
    void add(int value) { data[count++] = value; }
    bool contains(int value) const { /* scan data[0..count) */ }
    int size() const { return count; }
private:
    int data[100];   // implementation detail — callers never see this
    int count = 0;
};
```

PROOF, NOT JUST ASSERTION

One ADT, Two Implementations



runSameOperations never touches data or count directly — it genuinely cannot tell which implementation it's holding.

PROOF, NOT JUST ASSERTION

Same Five add() Calls, Different Outcomes

add(value)	SmallArrayBag (cap 3)	GrowableBag (dynamic)
add(10)	ok	ok
add(20)	ok	ok
add(30)	ok	ok
add(40)	REJECTED (full)	ok
add(50)	REJECTED (full)	ok
Final size	3	5

Both bags received the exact same five add() calls, through the exact same interface — and behaved completely differently, purely because of what's happening behind that interface.

CORE CONCEPT

Abstraction and Encapsulation

A

Abstraction

Exposing only what a user needs to know (add, contains, size) and hiding everything else.

E

Encapsulation

C++'s mechanism to enforce it: the private section physically prevents outside code from touching data or count.

encapsulation in action

```
// bag.data[0] = 5;
//    ^ ERROR: 'data' is private within this context

bag.add(5);    // OK — goes through the public interface
```

DESIGN THINKING

Every Design Is a Trade-off

T

Time vs. Space

A hash table answers “is this present?” almost instantly but spends extra memory. A plain array uses less memory but may check every element.

S

Simplicity vs. Efficiency

An unsorted array is trivial to insert into but slow to search. Keeping it sorted speeds up search but slows every insertion.

Criteria for selecting a structure

- What operations does the application perform most often?
- How much data will it realistically hold — does that size change at runtime?
- Is memory a hard constraint, or is speed the priority?
- How will the application's needs likely change as it grows?

A choice that's correct today can become wrong tomorrow — good engineers revisit their data structure choices as usage patterns become clear.

KEY TAKEAWAYS

Key Takeaways

- Every data structure can be classified along several independent axes at once: composition, arrangement, allocation, access pattern, and persistence.
- Every data structure supports some combination of seven core operations: traversal, insertion, deletion, searching, sorting, updating, merging.
- An Abstract Data Type describes WHAT a structure does, completely separate from HOW it's implemented.
- ArrayBag and VectorBag produced byte-for-byte identical output through the same interface — proof that interface and implementation are independent.
- Every design is a trade-off — usually time vs. space, or simplicity vs. efficiency — and the right choice can shift as an application's requirements evolve.