UNIT 2 · LINEAR DATA STRUCTURES

9. Linear Structures: Applications

CSC211 · Algorithms and Data Structures

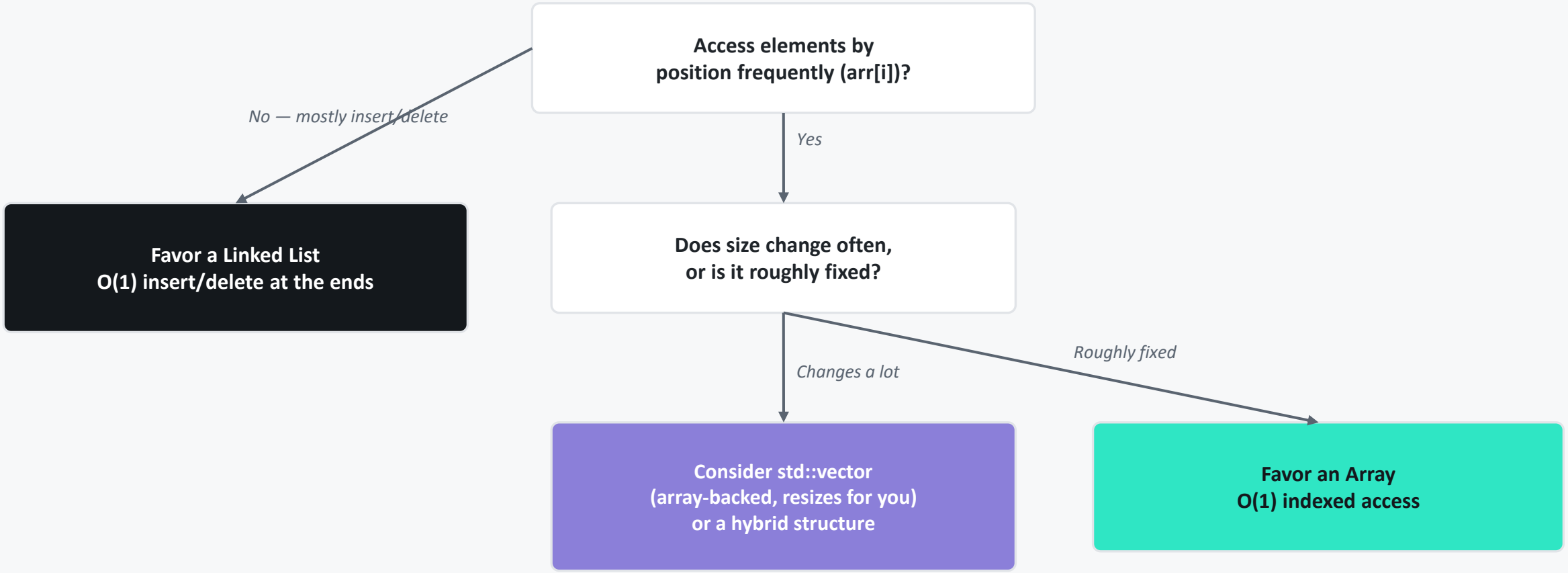
COMSATS University Islamabad, Lahore Campus

Agenda

1. A decision framework: array vs. linked list
2. The waiting-list problem, solved and timed both ways
3. The Josephus problem: a circular linked list in action
4. Combining structures — and real-world applications

Choosing an Appropriate Linear Structure

The decision almost always comes down to which operation your application performs most often.



Priority → Better Fit

Your priority	Better fit
Fast lookup by position, size roughly known	Array
Frequent insertion/deletion, especially at the front	Linked List
Frequent insertion/deletion AND occasional indexed access	std::vector or a later structure (tree, hash table)
Memory is extremely tight, every byte counts	Array — no per-element pointer overhead

“Dynamic” data — size unknown ahead of time, or constantly changing — is where linked lists earn their keep: notification lists, open-document lists, active-enemy lists.

Same Problem, Solved Both Ways

Maintain a waiting list: serve strictly in arrival order — remove from the front, add at the end.

array-based

```
class ArrayWaitingList {
  int data[10000];
  int frontIndex = 0, count = 0;
public:
  void arrive(int id) {
    data[frontIndex + count] = id;
    count++;
  }
  int serve() {
    int id = data[frontIndex];
    frontIndex++;  // just move marker
    count--;
    return id;
  }
};
```

linked-list-based

```
class LinkedWaitingList {
  Node *head=nullptr, *tail=nullptr;
public:
  void arrive(int id) {
    Node* n = new Node(id);
    if (!tail) { head=tail=n; return; }
    tail->next = n; tail = n;
  }
  int serve() {
    int id = head->data;
    Node* old = head;
    head = head->next;
    delete old;
    return id;
  }
};
```

Both O(1) per Operation — But Not Equal in Practice

5000 arrivals + 5000 serves. arrive()/serve() only ever touch the ends — the cheap access pattern for both structures.

1st

Array-based finishes first — contiguous memory, no new/delete overhead

2x+

Linked-based takes at least 2x longer, same machine

O(1)

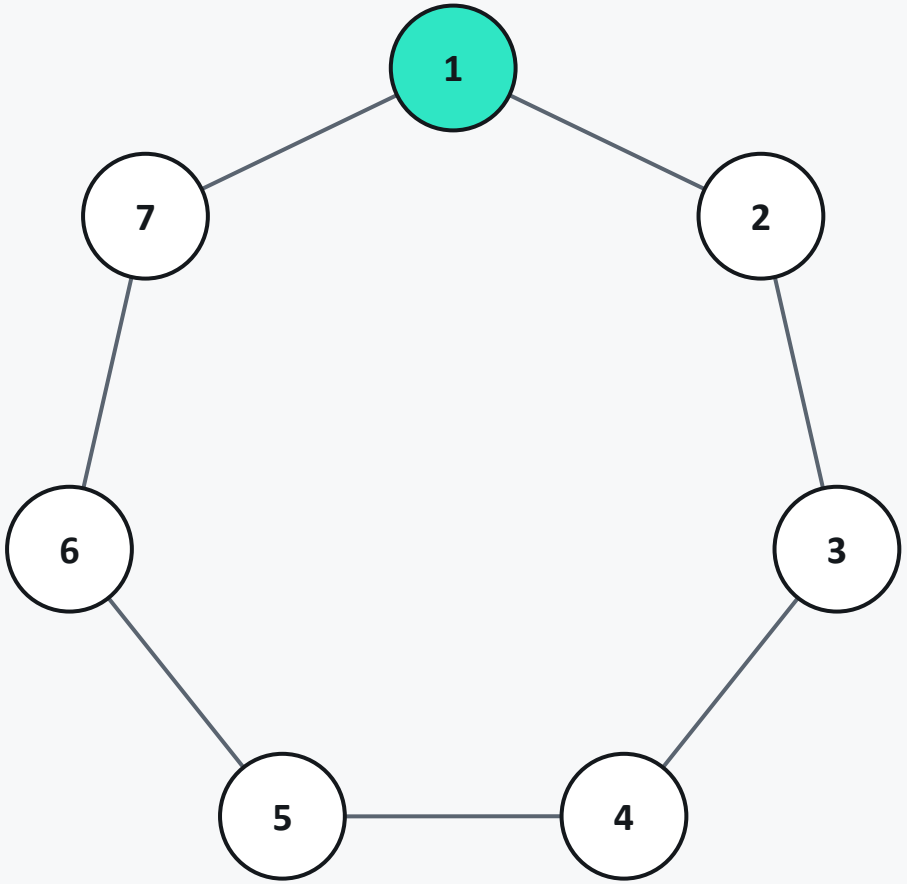
Big-O identical for both — the gap is constant-factor, not complexity

Report a verdict or ratio, never a raw timing number.

Raw microsecond counts don't reproduce across machines or even across runs on the same machine (CPU scheduling noise, cache warmup, background load). If a middle-removal were needed too, the array would need O(n) shifting while the linked list — given a pointer — would not: the "right" answer depends entirely on which operations the application performs.

The Josephus Problem: A Circular List's Strength

n people stand in a circle, numbered 1..n. Starting from person 1, count around and eliminate every k-th person. Who survives?



last node's next pointer wraps back to node 1 — no “end” to fall off

Same shape shows up in:

- Round-robin CPU scheduling
- Multiplayer turn rotation
- A “repeat” playlist skipping removed tracks

Josephus: Splice Before Delete

josephus.cpp — core loop

```
current = last;   // start counting from person 1
while (current->next != current) {
    for (int step = 1; step < k; step++)
        current = current->next;          // walk k-1 steps

    Node* toEliminate = current->next;
    current->next = toEliminate->next;  // splice out BEFORE delete
    delete toEliminate;
    current = current->next;              // continue from next survivor
}
```

`current->next != current` is the loop condition: once one node is left, that node's own next points back to itself — mirroring how head == nullptr signals an empty singly linked list.

Tracing the Elimination: n=7, k=3

Start

```
Circle: 1 2 3 4 5 6 7
```

Eliminate 4

```
Circle: 1 2 3 5 6 7
```

Eliminate 1

```
Circle: 2 3 5 6 7
```

Eliminate 6

```
Circle: 2 3 5 7
```

Eliminate 5

```
Circle: 2 3 7
```

Eliminate 7

```
Circle: 2 3
```

Survivor: 2

```
Circle: 2
```

Each elimination is O(1) once positioned — splice the node out, no shifting.

Why a Circular List, Not a Circular Array?

	Circular array (shifting)	Circular linked list
Counting k steps	O(k)	O(k)
Removing counted person	O(n) — shift everything after	O(1) — splice out
Total for n-1 eliminations	$O(n \cdot k + n^2)$	$O(n \cdot k)$

The outer loop runs n-1 times; each iteration walks k-1 steps before eliminating — $O(n \cdot k)$ total either way. The difference is entirely in the elimination step itself: splice (O(1)) vs. shift (O(n)).

Combined Linear Structure Operations

Text editor undo history

Array for the visible document (fast rendering, indexed access) + a linked list/stack of past edits (cheap push/pop of recent changes).

Music player

Array to display the playlist (jump to song #7 instantly) + a doubly linked list's prev/next for skip forward/back playback.

Real problems often need more than one structure working together — not a single winner.

Real-World Applications

Application	Structure used	Why
Undo/redo in an editor	Stack (linked list)	Most recent change undone first — LIFO
Print job queue	Queue (linked list)	Jobs print in submission order — FIFO
Autocomplete suggestions	Array (sorted)	Frequent lookups, rarely modified after load
Infinite-scroll feed	Linked list	Constantly appending; never jumps to post #500
Image pixel data	2D array	Every pixel needs instant O(1) (row, col) access

Key Takeaways

- Choosing between an array and a linked list is a trade-off, not a search for one objectively “best” structure — it depends on the operations the application performs most.
- Arrays win on indexed access with predictable size; linked lists win on frequent insertion/deletion, especially at the ends, with unpredictable size.
- Equal Big-O doesn't mean equal speed: arrays are typically faster in practice thanks to cache-friendly contiguous memory. Report a verdict or ratio, never a raw timing number.
- A circular linked list is the natural fit for cycling through a shrinking set of participants without a fixed end — the Josephus problem, round-robin scheduling, turn rotation.
- Real applications frequently combine multiple structures, each handling the part it's naturally good at.