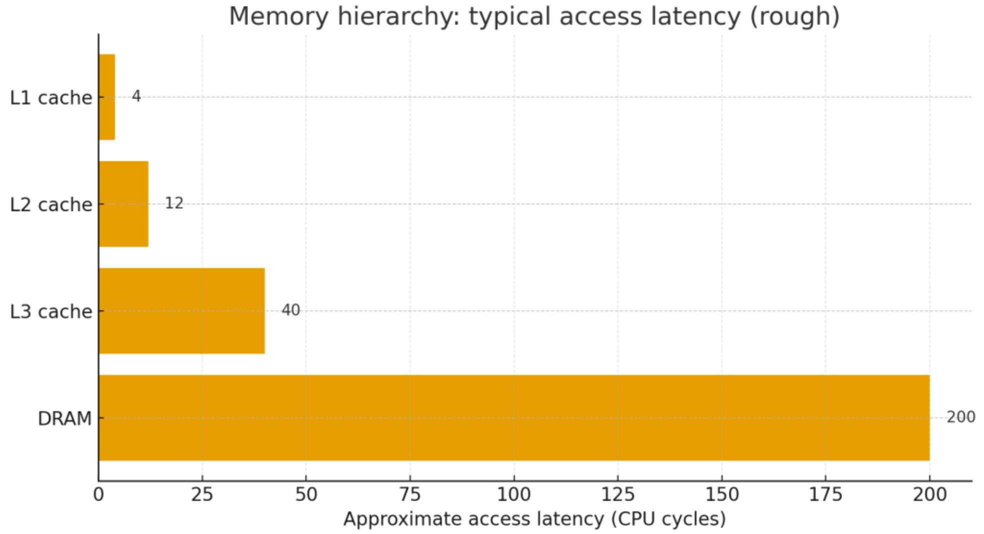
Lecture 4 – Linear and Hierarchical Collections: Arrays, Linked Lists, Stacks, Queues, and Heaps, Design & Performance Trade-offs

NURSULTAN KHAIMULDIN

N.KHAIMULDIN@ASTANAIT.EDU.KZ

Content

BIG-O ≠ SPEED
Arrays vs Linked Lists



Same $O(n)$ can differ by 10–100× in practice

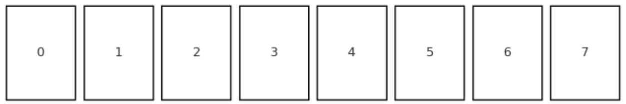
Memory hierarchy dominates: locality vs pointer-chasing

Branch prediction and control flow matter

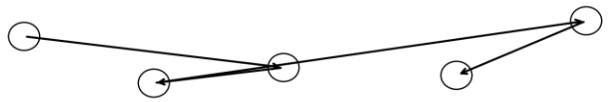
Allocation/GC overheads change constants

Design takeaway: choose data layout, not just operations

Array block (contiguous)



Linked nodes (pointer chasing)

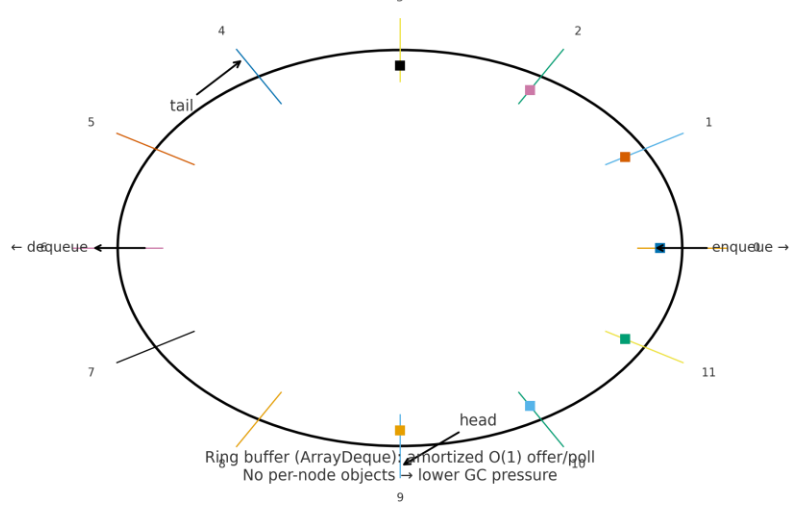


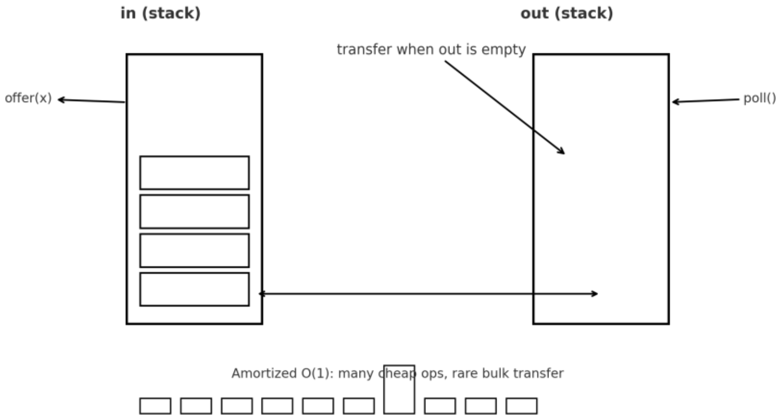
Arrays: contiguous memory, predictable access, prefetch-friendly

Linked lists: per-node objects → poor locality, more branches

ArrayList implements RandomAccess; LinkedList.get(i) is linear

Iteration: both $O(n)$, arrays are usually much faster on JVM





Stacks & Queues: Design Options

Prefer ArrayDeque for stack/queue (ring buffer; amortized

LinkedList queue: extra allocations, GC pressure

Two-stack queue: rare bulk transfer; amortized $O(1)$ per operation

```java
import java.util.ArrayDeque;
import java.util.Deque;

final class TwoStackQueue<E> {
    private final Deque<E> in = new ArrayDeque<>(), out = new ArrayDeque<>();
    void offer(E x) { in.push(x); }
    E poll() {
        if (out.isEmpty()) while (!in.isEmpty()) out.push(in.pop());
        return out.isEmpty() ? null : out.pop();
    }
}
```

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<Integer> q = new ArrayDeque<>();
        q.offer(10); q.offer(20);
        System.out.println(q.poll());

        Deque<Integer> st = new ArrayDeque<>();
        st.push(1); st.push(2);
        System.out.println(st.pop());
    }
}
```

Amortized Analysis

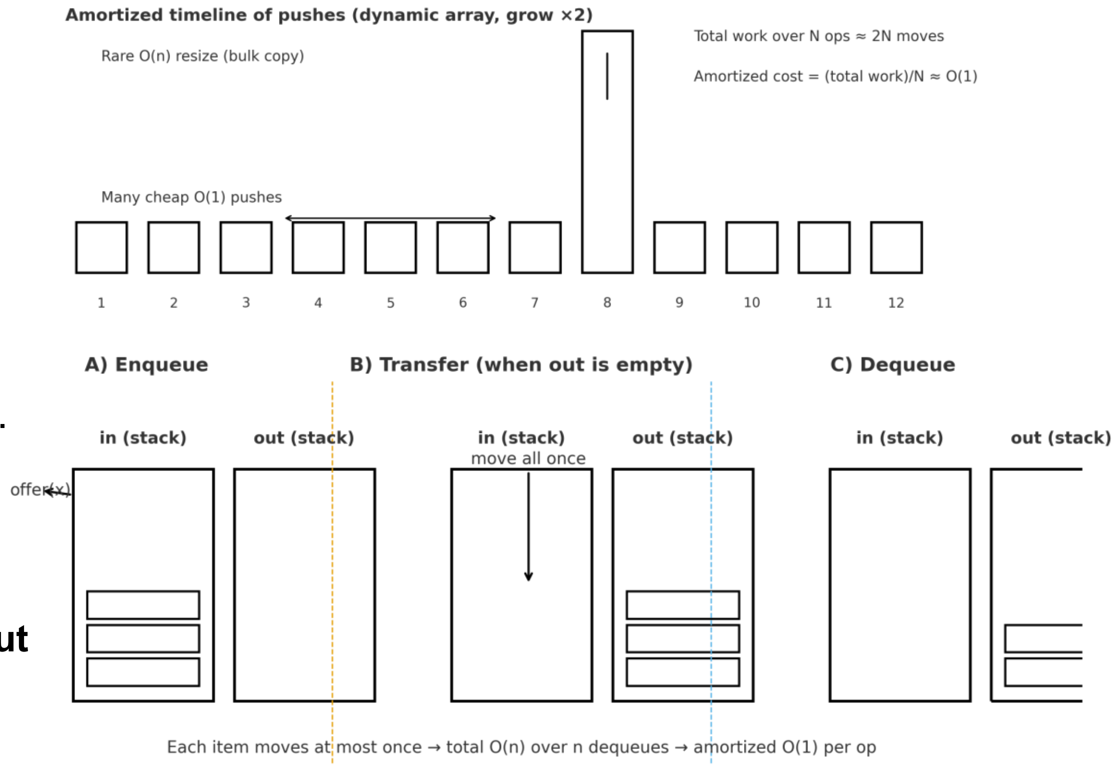
Amortized cost = **(total cost of a long sequence) / (# of operations)**.
We “spread” the rare expensive steps across many cheap ones.

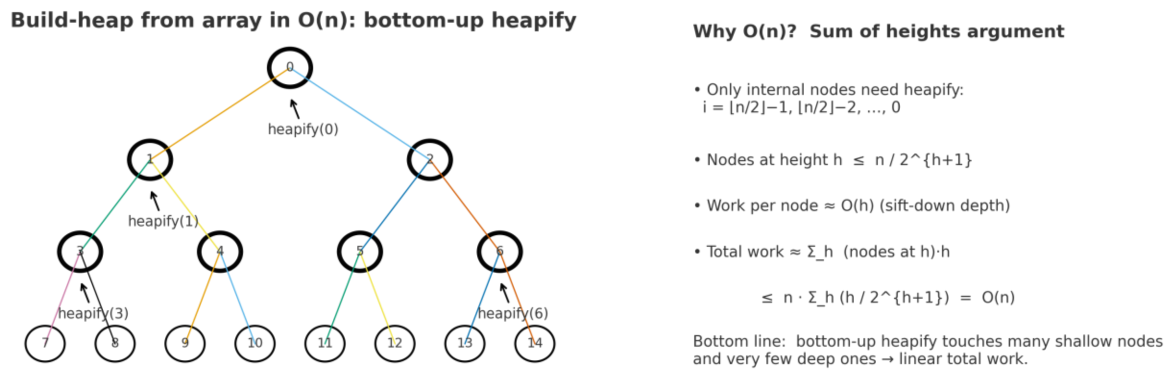
Dynamic array (grow ×2):

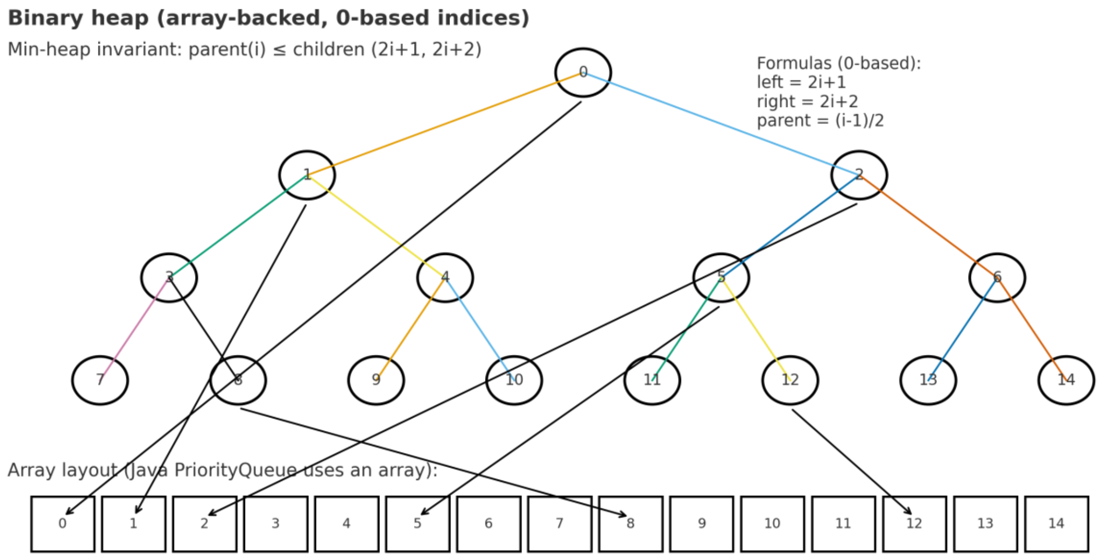
- Most push = **O(1)**; occasional **resize = O(n)** (bulk copy).
- Total copies after N pushes ≤ **2N** (each element moves a constant # of times).
- **Average per push ≈ (2N moves)/N = O(1)** → fast in practice.

Two-stack queue (in/out):

- offer(x) → push to **in**; poll() → pop from **out** (if empty: transfer all from **in** to **out** once).
- Each element is moved **at most once** from **in** → **out**.
- Over N dequeues: total moves ≤ **N** ⇒ **O(N)** total ⇒ **O(1)** amortized per operation.





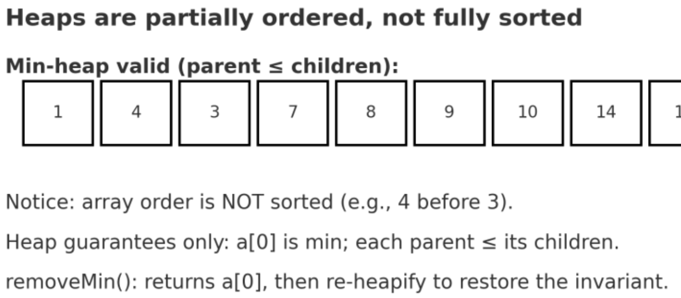


Binary heap invariant; array layout (0-based): left=2i+1, right=2i+2, parent=(i−1)/2

insert / deleteMin → O(log n); peek → O(1)

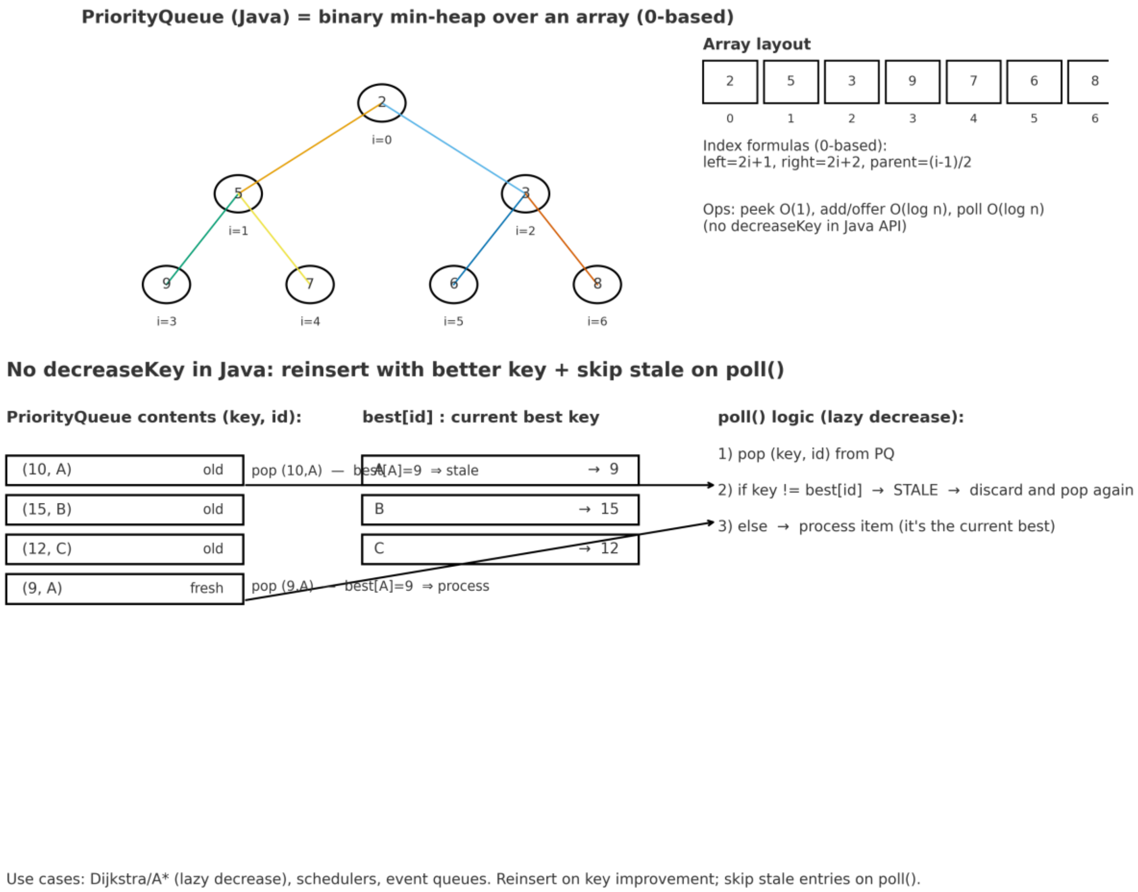
Build-heap from array → O(n) (bottom-up heapify), not $O(n \log n)$

Heaps are partially ordered, not fully sorted



Heaps at a glance

Priority Queue In Java



PriorityQueue is a binary min-heap stored in an array (0-based).
Index math: left = 2*i+1, right = 2*i+2, parent = (i-1)/2.
Costs: peek = O(1); add/offer, poll = O(log n) (walk height of the heap).

Tight layout ⇒ good cache locality; fewer objects than tree nodes.

Java API does not expose it (heap has no handles to find an item in O(log n)).
Reinsert an item with the improved key and skip stale entries when popping.

Lazy-decrease pattern (on poll()).
Keep a best map from id → current best key.
When you poll() (key,id), if key != best.get(id), it's stale → discard and pop again; else process.

JVM-specific costs

int[] vs List<Integer>

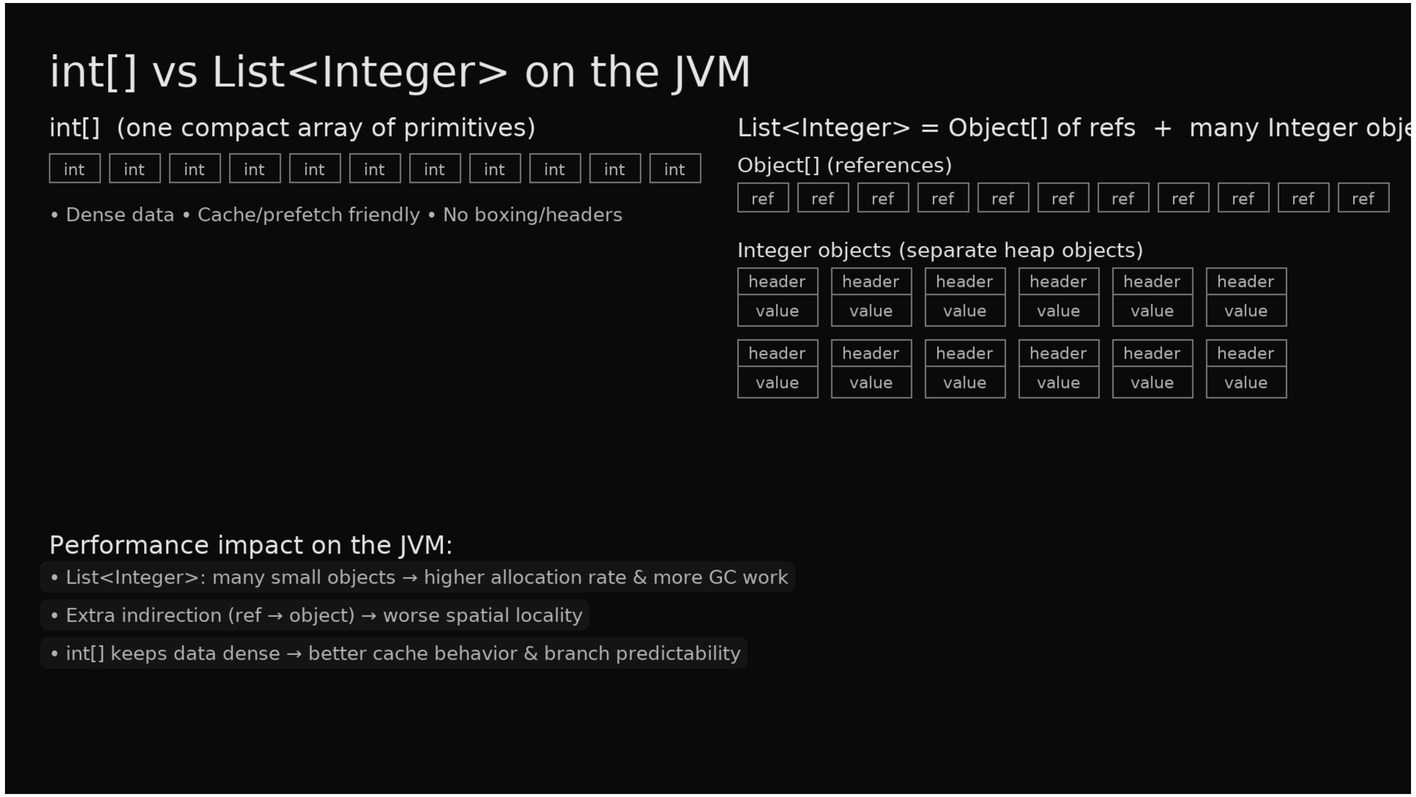
int[] stores primitives contiguously in a single array → excellent locality, no boxing, few objects.

List<Integer> stores an Object[] of references → each element is a separate Integer object (header + value [+ padding]) at another address. That means more objects, more allocations/GC, and extra indirection on every access.

Object headers & Reference

Each heap object carries a header and reference fields; this increases the per-element footprint compared to primitives and spreads data across memory.

Traversals that chase reference (linked nodes, boxed collections) produce unpredictable addresses and data-dependent branches → more cache/TLB misses and branch mispredictions. Tight loops over arrays are predictable and JIT-friendly.



Thank you