



ASTANA IT  
UNIVERSITY

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

NURSULTAN KHAIMULDIN

[N.KHAIMULDIN@ASTANA.IT.EDU.KZ](mailto:N.KHAIMULDIN@ASTANA.IT.EDU.KZ)

# Content



1. BIG-O  $\neq$  SPEED
2. Arrays vs Linked Lists
3. Stacks & Queues: Design Options
4. Amortized Analysis
5. Heaps At A Glance
6. Priority Queue In Java
7. JVM-specific costs

# 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

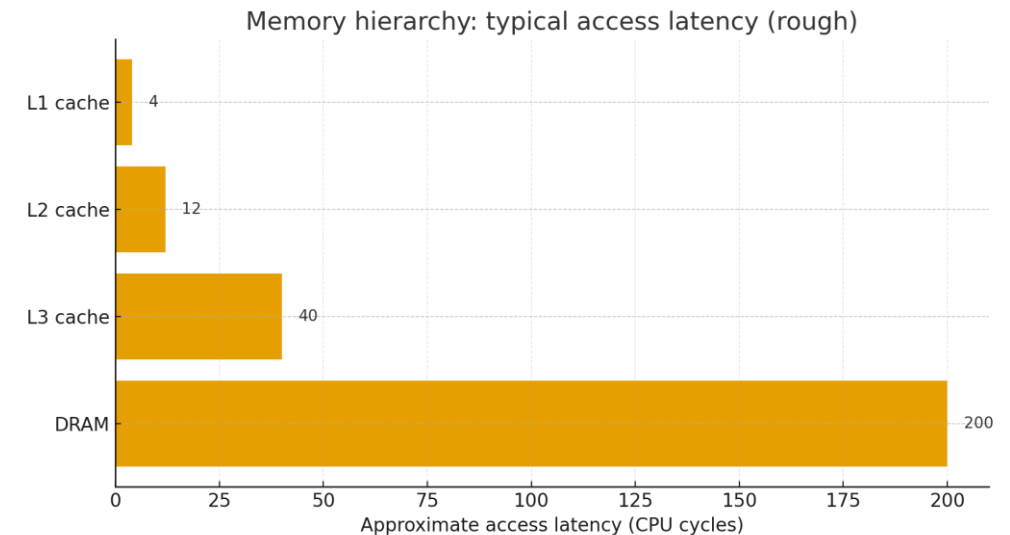
Design takeaway: choose data layout, not just operations

**Arrays:** contiguous memory, predictable access, prefetch-friendly

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

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

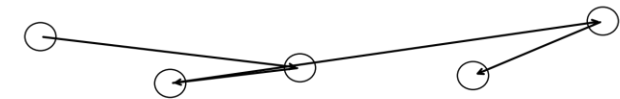
Iteration: both  $O(n)$ , arrays are usually much faster on JVM



Array block (contiguous)



Linked nodes (pointer chasing)

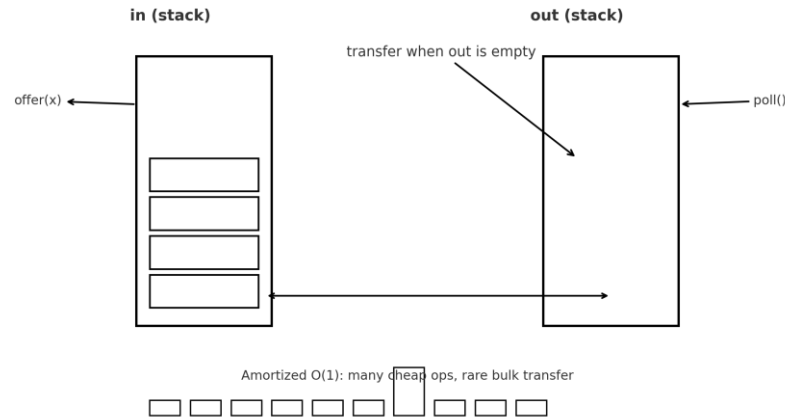
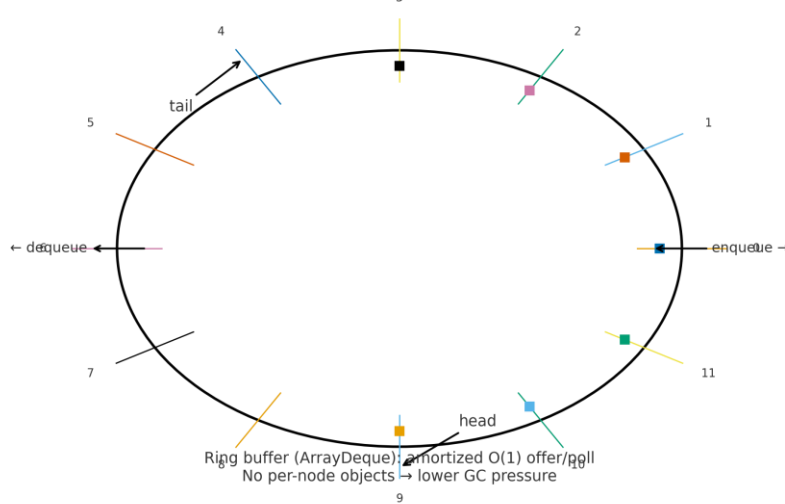


# Stacks & Queues: Design Options

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



```
1 import java.util.ArrayDeque;
2 import java.util.Deque;
3
4 final class TwoStackQueue<E> { no usages
5     private final Deque<E> in = new ArrayDeque<>(), out = new ArrayDeque<>(); 3 usages
6     void offer(E x) { in.push(x); } no usages
7     @ E poll() { no usages
8         if (out.isEmpty()) while (!in.isEmpty()) out.push(in.pop());
9         return out.isEmpty() ? null : out.pop();
10    }
11 }
```

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

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

        Deque<Integer> st = new ArrayDeque<>();
        st.push(e: 1); st.push(e: 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 $\times 2$ ):

- Most push =  **$O(1)$** ; occasional **resize =  $O(n)$**  (bulk copy).
- Total copies after  $N$  pushes  $\leq 2N$  (each element moves a constant # of times).
- **Average per push  $\approx (2N \text{ moves})/N = O(1)$**   $\rightarrow$  fast in practice.

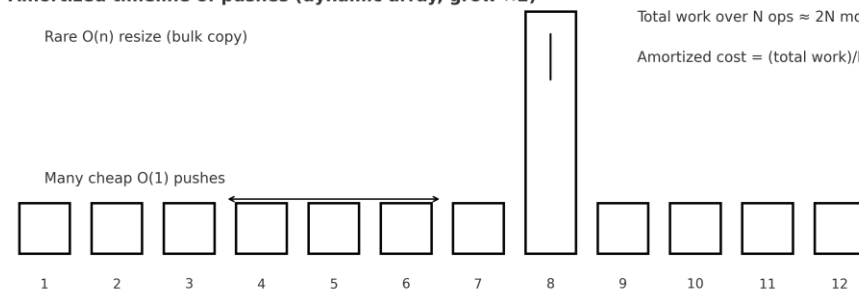
## Two-stack queue (in/out):

- $\text{offer}(x) \rightarrow$  push to **in**;  $\text{poll}() \rightarrow$  pop from **out** (if empty: transfer all from **in** to **out** once).
- Each element is moved **at most once** from **in**  $\rightarrow$  **out**.
- Over  $N$  dequeues: total moves  $\leq N \Rightarrow O(N)$  total  $\Rightarrow O(1)$  amortized per operation.

Amortized timeline of pushes (dynamic array, grow  $\times 2$ )

Rare  $O(n)$  resize (bulk copy)

Many cheap  $O(1)$  pushes



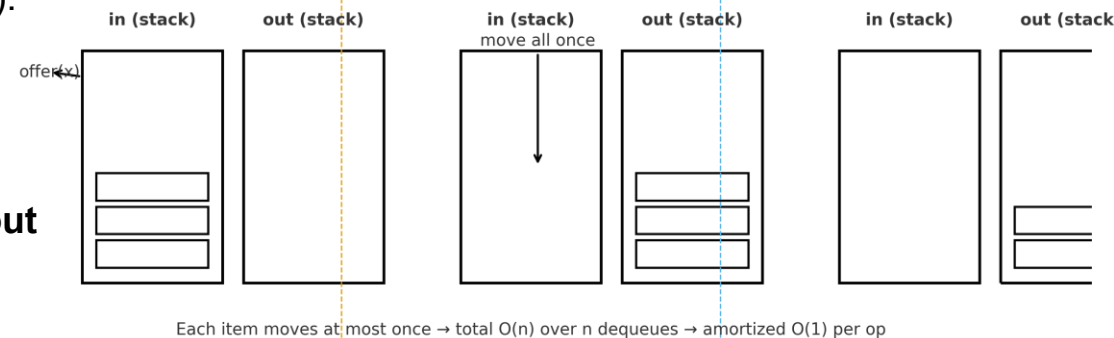
Total work over  $N$  ops  $\approx 2N$  moves

Amortized cost = (total work)/ $N \approx O(1)$

A) Enqueue

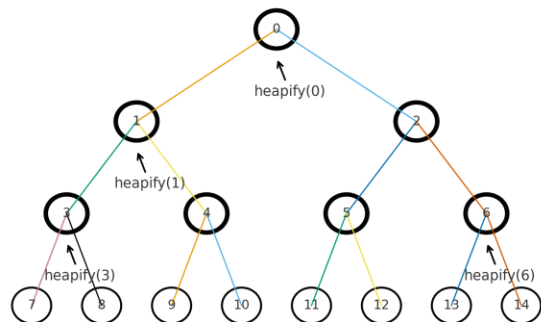
B) Transfer (when out is empty)

C) Dequeue



Each item moves at most once  $\rightarrow$  total  $O(n)$  over  $n$  dequeues  $\rightarrow$  amortized  $O(1)$  per op

## Build-heap from array in $O(n)$ : bottom-up heapify



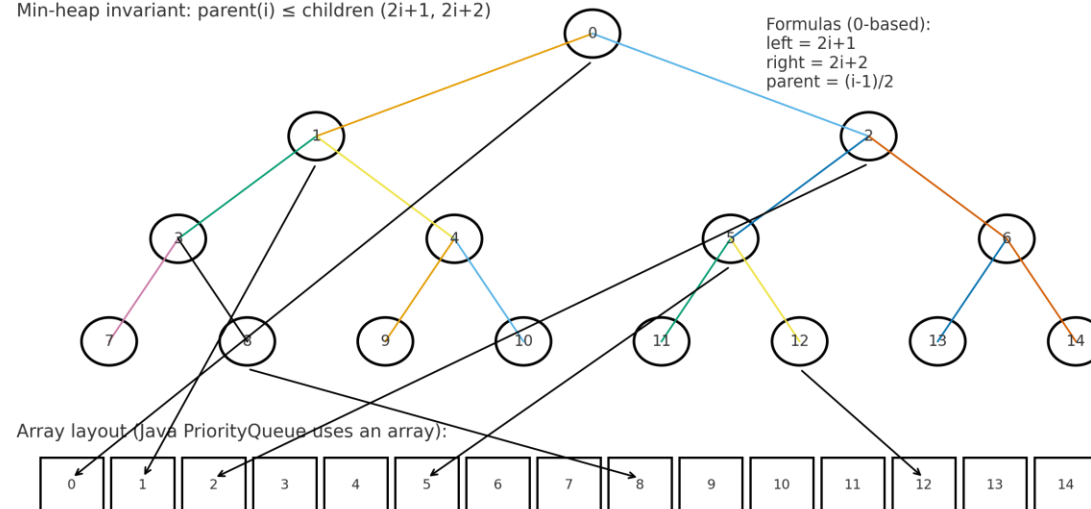
### Why $O(n)$ ? Sum of heights argument

- Only internal nodes need heapify:  
 $i = \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 2, \dots, 0$
- Nodes at height  $h \leq n / 2^{h+1}$
- Work per node  $\approx O(h)$  (sift-down depth)
- Total work  $\approx \sum_h (\text{nodes at } h) \cdot h$   
 $\leq n \cdot \sum_h (h / 2^{h+1}) = O(n)$

Bottom line: bottom-up heapify touches many shallow nodes and very few deep ones  $\rightarrow$  linear total work.

## Binary heap (array-backed, 0-based indices)

Min-heap invariant:  $\text{parent}(i) \leq \text{children}(2i+1, 2i+2)$



Formulas (0-based):  
 $\text{left} = 2i+1$   
 $\text{right} = 2i+2$   
 $\text{parent} = (i-1)/2$

Array layout (Java PriorityQueue uses an array):



Binary heap invariant; array layout (0-based):  $\text{left}=2i+1$ ,  $\text{right}=2i+2$ ,  $\text{parent}=(i-1)/2$

insert / deleteMin  $\rightarrow O(\log n)$ ; peek  $\rightarrow O(1)$

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

Heaps are partially ordered, not fully sorted

## Heaps are partially ordered, not fully sorted

Min-heap valid (parent  $\leq$  children):



Notice: array order is NOT sorted (e.g., 4 before 3).

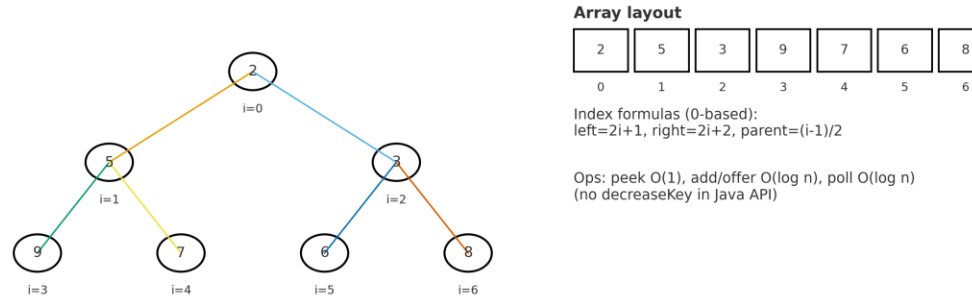
Heap guarantees only:  $a[0]$  is min; each parent  $\leq$  its children.

removeMin(): returns  $a[0]$ , then re-heapify to restore the invariant.

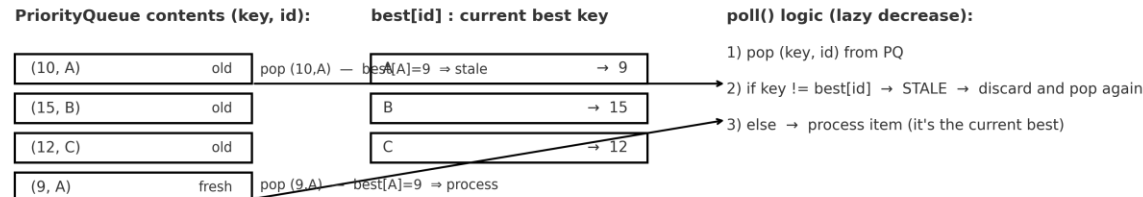
# Heaps at a glance

# Priority Queue In Java

PriorityQueue (Java) = binary min-heap over an array (0-based)



No decreaseKey in Java: reinsert with better key + skip stale on poll()



Use cases: Dijkstra/A\* (lazy decrease), schedulers, event queues. Reinsert on key improvement; skip stale entries on poll().

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  $\Rightarrow$  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  $\rightarrow$  current best key.

When you poll() (key,id), if key  $\neq$  best.get(id), it's stale  $\rightarrow$  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.

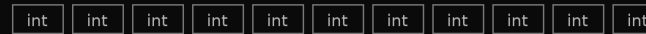
### Object headers & Reference

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.

## `int[]` vs `List<Integer>` on the JVM

`int[]` (one compact array of primitives)



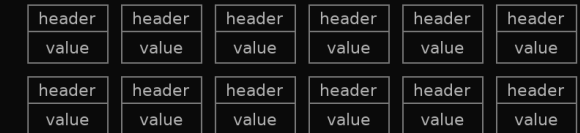
- Dense data
- Cache/prefetch friendly
- No boxing/headers

`List<Integer>` = `Object[]` of refs + many `Integer` objects

`Object[]` (references)



`Integer` objects (separate heap objects)



### Performance impact on the JVM:

- `List<Integer>`: many small objects → higher allocation rate & more GC work
- Extra indirection (ref → object) → worse spatial locality
- `int[]` keeps data dense → better cache behavior & branch predictability



Thank you

---

