

Topic 10: Data Types and Structures

Heaps and priority queues, the hidden cost of growing arrays, and choosing the right structure.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 10.2 Arrays, 10.4 Abstract data types. Try the challenge questions on the last pages before you read the answers.

Priority queues and the binary heap

A queue serves the item that arrived first. A **priority queue** serves the item with the highest priority: an operating-system scheduler, hospital triage, the open set in Dijkstra's algorithm. A **binary heap** implements it efficiently inside an ordinary array.

In a **min-heap**, every parent is smaller than or equal to its children, so the smallest item is always at the root. Using 1-based indexing, the children of position i are at positions $2i$ and $2i+1$.

- **Insert:** add at the end, then **sift up** (swap with parent while smaller than it).
- **Remove-min:** take the root, move the last item to the root, then **sift down** (swap with the smaller child while larger than it).
- Both take at most the tree's height, about $\log_2 n$ steps.

```
Insert 5    [5]
Insert 3    [5,3]  -> 3 < parent 5, swap -> [3,5]
Insert 8    [3,5,8]
Insert 1    [3,5,8,1] -> 1 < parent 5, swap -> [3,1,8,5]
                1 < parent 3, swap -> [1,3,8,5]
```

Dynamic arrays and amortised cost

Arrays have fixed size, yet Python lists and Java ArrayLists grow. They allocate spare capacity; when full they allocate an array **twice as big** and copy everything across. A single append can be expensive (copy n items), but it is rare. Appending items to a list that starts with capacity 1 copies 1, 2, 4, 8... items at each doubling, so after n appends the total copying is less than $2n$: on average **$O(1)$ per append**. This averaged cost is called **amortised** cost.

Compare a **linked list**: inserting at the front is always $O(1)$, but there is no direct access by index and the nodes are scattered in memory, which the cache (topic 4 extension) dislikes. In practice dynamic arrays win more often than the theory suggests.

Choosing the right structure

Need	Good choice	Why
Undo history, bracket matching, recursion	Stack	Last in, first out
Print jobs, breadth-first search	Queue	First in, first out
Always process the most urgent item next	Priority queue (heap)	Smallest/largest in $O(\log n)$
Fast lookup by position; rarely insert in the middle	Array	Direct indexing, cache friendly
Frequent insert/delete once you hold a node	Linked list	Only pointers change
Sorted data searched often	Sorted array + binary search	$O(\log n)$ search

Challenge questions

1. Insert 7, 4, 9, 2 in that order into an empty min-heap. Show the array after each insert.
2. Perform remove-min on the heap [2,4,9,7].
3. A dynamic array starts with capacity 1 and doubles when full. How many element copies happen in total while appending 16 items?

Worked answers

Only look once you have had a proper go at each question.

1. $[7] \rightarrow [7,4]$ then $4 < 7$ so swap $\rightarrow [4,7] \rightarrow [4,7,9] \rightarrow$ add 2 at position 4: parent is position 2 (7), swap $\rightarrow [4,2,9,7]$; parent is position 1 (4), swap $\rightarrow [2,4,9,7]$.
2. Remove the root 2 and move the last item (7) to the root: $[7,4,9]$. Sift down: the smaller child is $4 < 7$, swap $\rightarrow [4,7,9]$. The returned minimum was 2.
3. Doubling occurs when adding items 2, 3, 5 and 9, copying $1 + 2 + 4 + 8 = 15$ elements: fewer than the 16 items stored, so roughly one copy per append.