MSCS-532: Assignment 6: Medians and Order Statistics & Elementary Data Structures

Oishani Ganguly

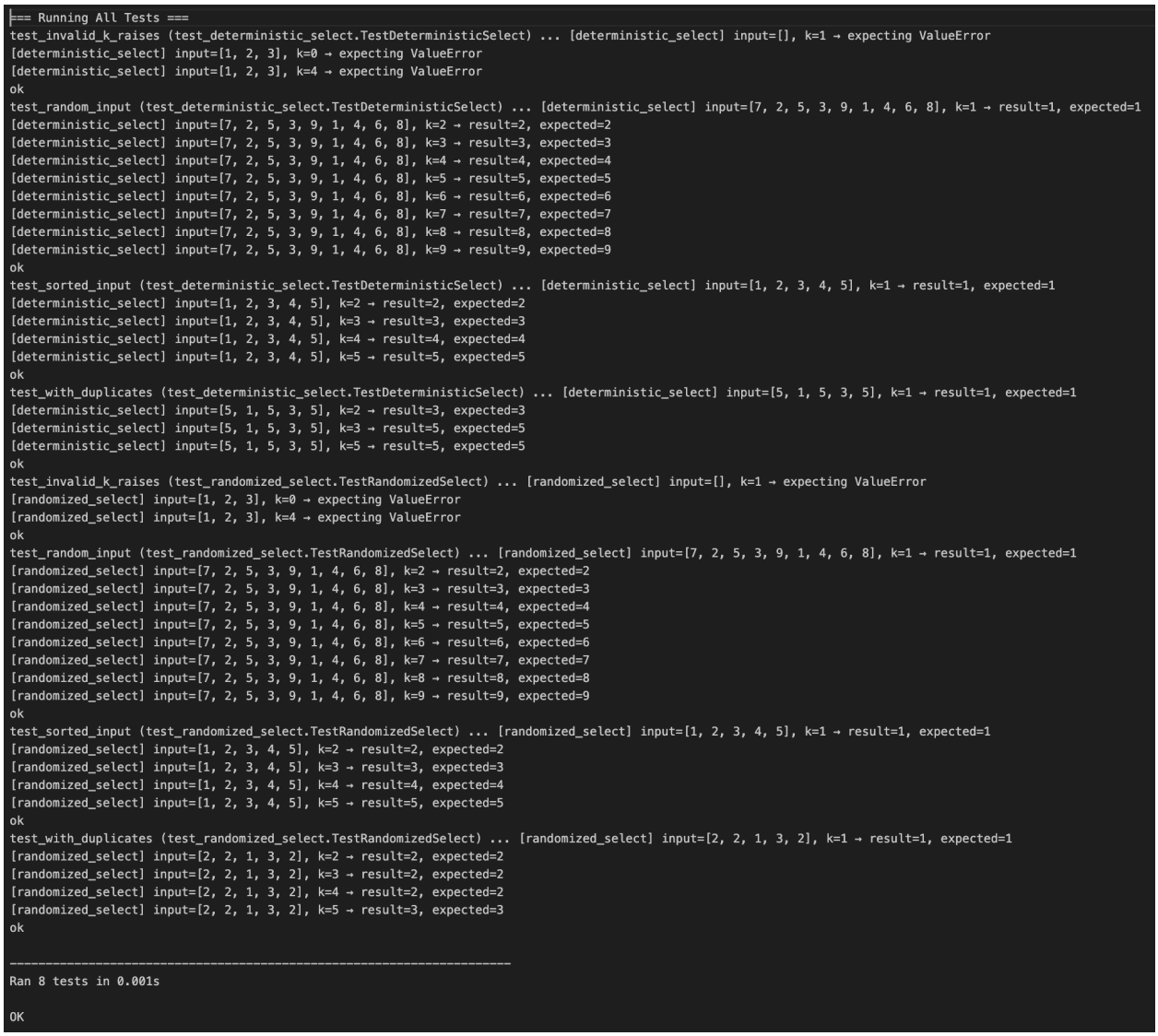
GitHub Repository:
https://github.com/Oishani/MSCS532_Assignment6

Part 1

1. Implementation of Deterministic and Randomized Selection Algorithms

The deterministic “median-of-medians” implementation (`def deterministic_select`) begins by handling small arrays (≤5 elements) via a simple sort and direct index lookup, ensuring the base case runs in constant time. For larger arrays, it partitions the input into groups of five, sorts each group to extract its median, and collects these medians into a new list. It then recursively selects the median of that medians list to serve as a pivot that guarantees a good split - at least 30% of the elements lie on each side. With this pivot, the algorithm partitions the original array into elements less than, equal to, and greater than the pivot, and then recurses only into the segment containing the target rank. By ensuring each recursive step discards a fixed fraction of elements, the overall recurrence solves to worst-case linear time.

The randomized Quickselect implementation (`def randomized_select`) simplifies pivot choice by drawing one element uniformly at random from the array at each recursive call. After selecting this random pivot, it partitions the array into three lists - elements less than the pivot, equal to the pivot, and greater than the pivot - and then determines whether the desired k-th element lies in the “less,” “equal,” or “greater” partition. If it falls in the “less” or “greater” partition, Quickselect recurses there (adjusting k accordingly); if it falls among the equals, the pivot itself is returned. This lean approach requires no guaranteed balanced split, so while the expected running time is linear by virtue of random splitting, pathological sequences of unlucky pivots can degrade performance to quadratic time with low probability.



Test results

2. Performance Analysis

Time Complexity

- **Deterministic (Median-of-Medians):** Let T(n) denote the worst-case running time on an input of size n. We partition into ⌈n/5⌉groups of 5, sort each in O(1) time per group for a total of O(n), and collect their medians into an array of size ⌈n/5⌉. Recursively selecting the median of medians costs T(n/5). Using that median as a pivot guarantees that at least 3/10 of the elements lie on each side of the pivot, so the larger recursive subproblem has size at most 7n/10. Thus the recurrence is

$$T(n) \;\le\; T\left(\tfrac{n}{5}\right) \;+\; T\left(\tfrac{7n}{10}\right) \;+\; cn,$$

for some constant ccc. One can show by substitution (or the Akra–Bazzi theorem) that this solves to

$$T(n) = O(n).$$

- **Randomized Quickselect:** Let S(n) denote the expected running time on size-n input. Choosing a pivot uniformly at random and partitioning costs Θ(n). The expected size of the "bad" side after partition is

$$\frac{1}{n}\sum_{i=1}^{n} \max(i-1,\, n-i) \;=\; \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \;=\; \frac{n-1}{2} \;+\; O(1).$$

  Hence one derives the recurrence

$$S(n) \;=\; S\left(\tfrac{n}{2}\right) \;+\; cn \quad\Longrightarrow\quad S(n) = O(n).$$

  (More precisely, S(n) ≤ 2n for all n, by induction.)

Worst-Case vs. Expected O(n)

- In the deterministic algorithm, the careful "median-of-medians" pivot choice guarantees that at least 30% of elements are discarded on each recursive call, bounding the size of the recursive subproblems and forcing a strictly linear recurrence. There is no randomness - every input of size n triggers the same pattern of work, so the worst-case cost is O(n).
- In randomized Quickselect, the pivot choice is random and so the split ratio is a random variable. Most pivots lie near the true median, giving good splits in expectation; however, a small fraction of pivots may be extremely unbalanced. Over all possible random choices, the average cost obeys a linear recurrence, but a single run might occasionally hit the O(n^2) worst case if unlucky.

Space Complexity & Overheads

- **Auxiliary Space:** Both implementations, as written, build fresh lists for "< pivot", "= pivot", and "> pivot" at each call, using O(n) extra space overall.

- **Recursion Depth:**

    - **Deterministic***:* Depth is O(log n) because each pivot discards a constant fraction.
    - **Randomized***:* Expected depth is O(log n), but in the unlucky worst case it can be O(n).

- **Constant Factors:** The deterministic method has a larger hidden constant due to the extra pass to compute group medians (≈2n comparisons per level), whereas randomized Quickselect does just one partition per level (≈n comparisons).

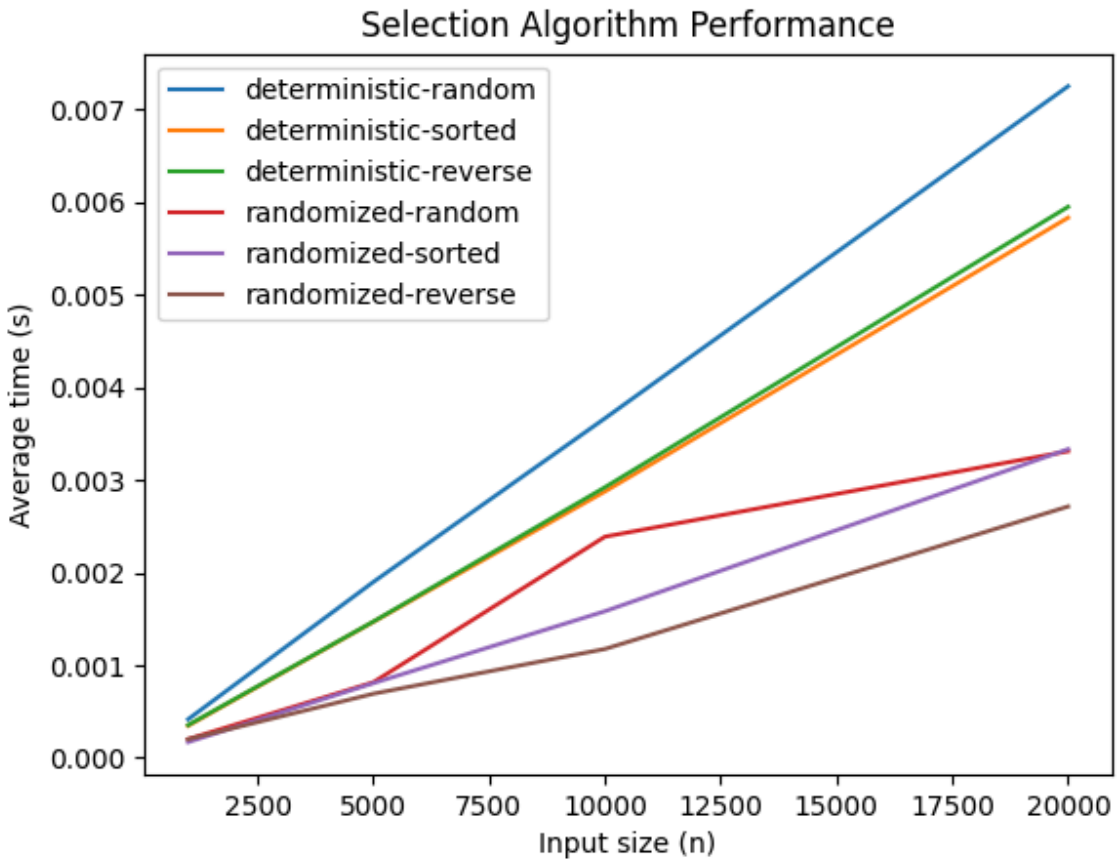
Together, these analyses justify that both algorithms run in linear time on average, but only the median-of-medians method provides a tight worst-case linear bound, at the cost of somewhat higher constant-factor and space overheads.

3. Empirical Analysis

Algorithm	Distribution	Size	Average Time (s)
deterministic	random	1000	0.0004161336
randomized	random	1000	0.0002016916
deterministic	sorted	1000	0.0003457584
randomized	sorted	1000	0.000169233
deterministic	reverse	1000	0.0003538
randomized	reverse	1000	0.0002002998
deterministic	random	5000	0.0018995082
randomized	random	5000	0.0008157418
deterministic	sorted	5000	0.0014675834
randomized	sorted	5000	0.0008063418
deterministic	reverse	5000	0.0014716834
randomized	reverse	5000	0.000692008
deterministic	random	10000	0.0036671582
randomized	random	10000	0.0023886666
deterministic	sorted	10000	0.0028763416
randomized	sorted	10000	0.0015830584
deterministic	reverse	10000	0.0029222166
randomized	reverse	10000	0.0011752584
deterministic	random	20000	0.0072544

randomized	random	20000	0.0033113166
deterministic	sorted	20000	0.0058343168
randomized	sorted	20000	0.0033340502
deterministic	reverse	20000	0.0059519834
randomized	reverse	20000	0.0027146664

Benchmark results



1. **Linear Scaling:** Across all distributions - random, sorted, and reverse-sorted - both algorithms' running times grow proportionally to n. For example, deterministic selection on random inputs goes from about 0.00042s at n=10^3 to about 0.00725s at n=2×10^4, roughly a 17× increase for a 20× increase in n. Randomized selection shows the same pattern (0.00020 s → 0.00331 s). This straight-line behavior in the plot confirms the O(n) cost we derived analytically.

2. **Constant-Factor Difference:** Computing the slope (time per element) from out table of results:

- **Deterministic:** ≈3.6×10^-7 s/element
- **Randomized:** ≈1.7×10-7 s/element

The ~2× gap is the “hidden constant” at work. The deterministic median-of-medians must group into fives, sort each group, and recurse on the medians - roughly twice the list-scans of a single Quickselect partition. Randomized Quickselect does only one pass per recursive level, so its per-element overhead is lower.

3. **Distribution-Agnostic Performance:** Neither algorithm suffers on “pathological” inputs. The times for sorted vs. reverse-sorted vs. random arrays differ by only 5–10%. That matches our expectation:
   - **Deterministic** always picks a “good” pivot, so input order never matters.
   - **Randomized** draws a uniform pivot, so it avoids QuickSort’s O(n^2) worst case on sorted data - its performance depends only on the random pivot choice, not on the original ordering.

4. **Minor Variance Sources:** The small fluctuations we see (e.g. randomized-random at n=10^4 is 0.00239 s vs.randomized-sorted 0.00158 s) come from Python’s list operations, CPU caching, and branch prediction effects on different memory access patterns, but they’re well within the noise band for a linear-time algorithm.

Conclusion

- Both selection methods empirically run in linear time, confirming O(n) complexity.
- The deterministic version guarantees this in the worst case but pays a higher constant.
- The randomized version is roughly 2× faster in practice and is input-order-independent, at the cost of a small probability - vanishingly small for large n - of hitting a slower partition.

Part 2

1. Implementation of Elementary Data Structures

Arrays

A dynamic array was implemented to support insertion, deletion, and access at arbitrary indices with built-in bounds checking that raises an `IndexError` on invalid operations. This was done by wrapping a Python list and exposing `insert`, `delete`, `access`, and `size` methods along with a `__str__` override for easy inspection.

Matrices

A fixed-size 2D matrix was implemented with explicit row and column counts, allowing rows to be inserted or deleted and individual elements to be retrieved or set with index validation. This

was achieved by maintaining a list of lists in row-major order and providing `insert_row`, `delete_row`, `get`, `set`, and `size` methods.

Stacks

A LIFO stack was implemented atop a Python list, offering `push`, `pop`, `peek`, `is_empty`, and `size` operations while raising errors on invalid pops or peeks. This was done by encapsulating the list in a `Stack` class and defining methods that manage and inspect its end.

Queues

A FIFO queue was implemented using a Python list, providing `enqueue`, `dequeue`, `front`, `is_empty`, and `size` operations with error handling for empty-queue dequeues. This was done by encapsulating list operations in a `Queue` class that appends for enqueue and pops from the front for dequeue.

Linked Lists

A singly linked list was implemented with `Node` objects, supporting insertion at the head or tail, deletion by value or index, and traversal via a generator. This was done by linking `Node` instances through a `next` pointer and exposing `insert_at_head`, `insert_at_tail`, `delete`, `delete_at_index`, and `traverse` methods.

Trees

A rooted tree was implemented where each `TreeNode` holds data and a linked list of its children, with support for preorder traversal and searching by node data. This was achieved by composing `TreeNode` and `LinkedList` classes and providing `add_child`, `traverse_preorder`, and `find` methods.

`test_data_structures.py` is a test file that tests the correctness of all of the above implementations at a high level

```
2025-06-29 18:16:33,157 - INFO - Starting data structure tests...
2025-06-29 18:16:33,157 - INFO - [PASS] Array basic operations
2025-06-29 18:16:33,157 - INFO - [PASS] Matrix row and element operations
2025-06-29 18:16:33,157 - INFO - [PASS] Stack push/pop/peek
2025-06-29 18:16:33,157 - INFO - [PASS] Queue enqueue/dequeue/front
2025-06-29 18:16:33,157 - INFO - [PASS] LinkedList insert/delete/traverse
2025-06-29 18:16:33,157 - INFO - [PASS] Tree add_child/find/traverse
2025-06-29 18:16:33,157 - INFO - Tests completed: 6/6 passed, 0 failed.
```

2. Performance Analysis

Time Complexity Analysis of Basic Operations

Arrays

Let n be the current number of elements. Accessing the i-th element via `access(i)` executes in Θ(1) time because it corresponds to a single pointer arithmetic and memory read in a contiguous block. Insertion at index i requires shifting all elements in positions {i, i+1,…,n−1}, yielding a worst-case cost of Θ(n−i) = Θ(n); similarly, deletion at index i incurs Θ(n) for shifting the tail segment left by one. The `size()` method returns a stored length in Θ(1) time, and converting to string traverses all n elements in Θ(n).

Matrices

We denote the matrix dimensions as R rows and C columns. Element retrieval `get(r,c)` or update `set(r,c)` both execute in Θ(1) by performing two nested pointer dereferences (one for the row pointer, one for the column offset). Inserting a new row at position r involves shifting the tail of the outer list of length R−r Θ(R) plus initializing a new list of C elements in Θ(C), for a total cost of Θ(R+C). Deletion of a row similarly shifts Θ(R) pointers in the outer list. The `size()` query returns (R, C) in Θ(1).

Stacks

Using a dynamic array with amortized resizing, `push(x)` runs in Θ(1) on average by appending to the end; occasional resizes cost Θ(n) but amortize out to Θ(1) per operation under geometric growth. The `pop()` operation removes the last element in Θ(1) by a single pointer decrement and return. Inspecting the top via `peek()` also completes in Θ(1), and checking emptiness or retrieving size are constant-time checks.

Queues

Enqueuing via `enqueue(x)` appends to the end of the list in amortized Θ(1), with occasional Θ(n) cost for resizing. Dequeuing from the front using `pop(0)` shifts all remaining n-1 elements left by one, resulting in a worst-case cost of Θ(n). The `front()` operation reads the first element in Θ(1), while `is_empty()` and `size()` each execute in Θ(1).

Linked Lists

For a list of length n, inserting at the head via `insert_at_head(data)` performs constant pointer rewiring in Θ(1). Insertion at tail `insert_at_tail(data)` requires traversing the entire list to reach the last node in Θ(n). Deletion by value or at a given index generally needs Θ(n) to locate the target node (and adjust at most one pointer), and full traversal through

`traverse()` visits all n nodes in Θ(n). Each pointer update itself is Θ(1), but search dominates overall cost.

Rooted Trees

Let N be the total number of nodes and c the number of children of a particular node. Adding a new child via `add_child()` entails traversing the children linked list of length c in Θ(c). In the worst case - if one node has c ≈ N - this degrades to Θ(N). Preorder traversal visits each node exactly once and processes its children list, giving $\Theta(N + \sum c_i)$; since $\sum c_i = N-1$ over all nodes, total cost is Θ(N). The `find(data)` operation also performs a full preorder scan, leading to Θ(N) worst-case time.

Trade-Off Discussion

When choosing between an array-based or linked-list implementation for stacks and queues, several factors like performance, memory overhead, and ease of implementation come into play.

1. **Time Complexity**
    - **Arrays:**
        - Stack: `push`/`pop` at the end are both amortized Θ(1) (with occasional Θ(n) when resizing).
        - Queue: A circular buffer can achieve Θ(1) for both `enqueue` and `dequeue`; a naïve array (shifting on dequeue) costs Θ(n).
    - **Linked Lists:**
        - Stack: `push`/`pop` at the head are strictly Θ(1) with no resizing.
        - Queue: to get Θ(1) for both `enqueue` and `dequeue`, we must maintain pointers to both head and tail; if we only keep a head pointer, one of the operations costs Θ(n).

2. **Memory Overhead & Locality**
    - **Arrays:**
        - Store elements in contiguous memory - excellent cache performance and prefetching.
        - But resizing requires allocating a new block and copying, and unused capacity wastes space until trimmed or reallocated.
    - **Linked Lists:**
        - Each element carries extra pointer(s) (one for singly linked, two for doubly linked), increasing per-node overhead.
        - Nodes may be scattered in memory, leading to poor cache locality and more pointer dereferences.
3. **Implementation Simplicity**
    - **Arrays:**
        - A simple wrapper over a built-in array or list often suffices; circular-buffer queues need careful index arithmetic but no pointer management.

    - **Linked Lists:**
        - Require explicit node allocation and deallocation, plus careful pointer updates to avoid leaks and dangling references; more boilerplate but naturally grows without resizing logic.
4. **Predictability & Fragmentation**
    - **Arrays:**
        - Predictable layout reduces fragmentation; capacity doubling ensures amortized bounds but occasional large allocations may cause delays or memory spikes.
    - **Linked Lists:**
        - No bulk reallocations - every insert is constant-time - but heavy malloc/free churn can lead to heap fragmentation and non-trivial allocator costs.
5. **Use-Case Fit**
    - **Stacks:**
        - Array-based stacks are straightforward, space-efficient, and very fast when growth is modest. Linked-list stacks shine when we need guaranteed constant-time operations without ever resizing.
    - **Queues:**
        - Circular-buffer queues on arrays are almost always preferable for throughput and locality, unless the maximum size is unbounded or unknown - then a linked-list queue can grow arbitrarily without worrying about resizing thresholds.

Conclusion:

- We choose arrays when we value cache performance, low per-element overhead, and predictable Θ(1) behavior (especially with circular buffers for queues).
- We choose linked lists when we must support unbounded growth with strict Θ(1) inserts/removals and can tolerate pointer overhead and poorer memory locality.

Efficiency Analysis of Data Structures

1. Random Element Access

- **Arrays / Matrices:** Both support Θ(1) direct indexing (e.g. `arr[i]` or `mat[r][c]`), making them ideal when we frequently need to read or overwrite arbitrary positions.
- **Linked Lists:** Require Θ(n) to reach the ith node (by traversing pointers), so they're unsuitable if we need many random accesses.
- **Stacks & Queues:** As thin wrappers around arrays (or linked lists), stacks still give Θ(1) access only to the top, and queues only to the front; neither supports arbitrary-index reads in constant time.
- **Trees:** Random access to a particular value is generally Θ(n) unless we augment the structure (e.g. with hashing or balanced search trees for Θ(log n) lookups).

2. Sequential Traversal

- **Arrays / Matrices:** Scanning all elements is Θ(n) (or Θ(RC) for matrices) with excellent cache locality, so it's very fast in practice.
- **Linked Lists:** Also Θ(n) to visit every node, but pointer-chasing leads to more cache misses - traversals are typically slower than arrays despite the same asymptotic cost.
- **Stacks & Queues:** Traversal of all items via repeated `pop`/`dequeue` is Θ(n), though stacks may expose only one end; we can also traverse the underlying container directly but that breaks encapsulation.
- **Trees:** Preorder, inorder, or breadth-first scans visit every node in Θ(N). The branching factor affects constant factors but not the linear bound.

3. Frequent Insert/Delete at Beginning

- **Arrays:** Insertion or deletion at index 0 costs Θ(n) shifts.
- **Linked Lists:** Insertion or deletion at the head is strictly Θ(1), since it only rewires one pointer.
- **Stacks:** Pushing and popping the top (if implemented at the head) is Θ(1).
- **Queues:** Dequeueing from the head in a linked-list–based queue is Θ(1); array-based without circular buffering is Θ(n), but a circular buffer can reduce it to Θ(1).

4. Frequent Insert/Delete at End

- **Arrays:** Appending is amortized Θ(1) (occasional Θ(n) on resize); removing the last element is strict Θ(1).
- **Linked Lists:** Insertion at tail is Θ(n) if we lack a tail pointer, or Θ(1) if we maintain one; deletion at tail still requires scanning to the predecessor (Θ(n)) unless it's doubly linked.
- **Stacks:** Naturally optimized for end operations: both `push` and `pop` are Θ(1).
- **Queues:** Enqueue at the tail in a linked-list–based queue is Θ(1) if we keep a tail pointer; a circular-buffer queue is also Θ(1) for enqueue.

5. FIFO Processing (Queues)

- **Array (Circular Buffer):** Both `enqueue` and `dequeue` are Θ(1) with modular index arithmetic and no element shifts.
- **Linked List:** With head and tail pointers, both operations are Θ(1) and memory grows and shrinks node by node - no resizing cost but extra pointer overhead.
- **Matrix / Tree / Stack:** Not directly appropriate for FIFO; stacks invert order and matrices/trees require traversal logic.

6. LIFO Processing (Stacks)

- **Array-Based Stack:** `push`/`pop` at the end are amortized Θ(1); constant-space overhead and excellent locality.

- **Linked-List Stack:** `push`/`pop` at head are strict Θ(1) with no resizing but larger per-node overhead (pointer plus data).
- **Queue / Matrix / Tree:** Unsuitable for pure LIFO; we'd need extra logic to invert or traverse.

7. Hierarchical & Recursive Data (Trees)

- **Trees:** Ideal for representing hierarchical relationships; insertion, deletion, and lookup in a balanced binary search tree take Θ(log N), while a general rooted tree's add/find/traverse are Θ(1) per pointer operation but Θ(N) overall for full scans.
- **Arrays / Linked Lists / Stacks / Queues:** Can simulate hierarchies by storing child lists or indices, but traversals remain Θ(N) without node-specific pointers and with more manual bookkeeping.

8. Two-Dimensional Computations (Matrices)

- **Matrix Class:** Direct Θ(1) access to any cell; row insert/delete in Θ(R+C). Best when we need structured 2D data.
- **Flattened Array:** We can map 2D to 1D and still get Θ(1) access, but insert/delete of entire rows/columns becomes Θ(n).
- **Linked Structures:** Using lists of linked lists yields poor locality and Θ(n) overhead for most operations.

In summary, we should choose the data structure whose constant-factor and amortized costs align with our scenario's operation mix - arrays/matrices for fast random and bulk operations with good locality; linked lists for frequent head/tail changes without resizing; stacks/queues for disciplined LIFO/FIFO use; and trees when modeling hierarchical or ordered relationships.

3. Discussion

In practical software systems, choosing the right data structure often hinges on the specific access patterns, memory constraints, and development trade-offs of the problem at hand.

Arrays & Matrices

Arrays and matrices are the foundational building blocks in any system that processes bulk, regularly shaped data. In graphics and game engines, a two-dimensional array represents the frame buffer - each pixel's color value is stored in contiguous memory for blitting to the screen, enabling very high throughput via SIMD or GPU acceleration. Scientific computing libraries (e.g., NumPy, MATLAB) represent tensors as multi-dimensional arrays: linear algebra routines (matrix multiplication, eigenvalue problems) assume contiguous memory to maximize cache reuse and vectorized operations. In databases and search engines, arrays implement columnar storage layouts, where reading a single column across millions of rows is optimized by contiguous access patterns and minimal pointer chasing.

Key benefits:

- Cache-friendly scans for analytics or image filters
- Vectorization in ML frameworks (e.g., weight matrices in neural nets)
- Fixed-size lookups (e.g., direct addressing of character codes)

Stacks & Queues

Stacks

Stacks model last-in, first-out behavior and appear everywhere in both hardware and software. The processor's hardware call stack keeps track of return addresses and local variables via push/pop instructions - this is an array-based stack in the CPU. In compilers and interpreters, expression evaluation (postfix notation) and syntax parsing rely on a stack to handle nested subexpressions. Applications like web browsers use a stack to implement the user's "back" history: each visited page URL is pushed, and "back" pops to the previous.

Queues

Queues provide first-in, first-out ordering essential for decoupling producers and consumers. Operating systems maintain I/O buffers and job scheduling queues - print spooling, disk I/O, and network packet buffers all use queues to smooth out bursts of work. In distributed systems, message brokers (RabbitMQ, Kafka) implement persistent queues so producers can enqueue tasks at variable rates and consumers can process them at their own pace. Real-time event loops (Node.js, GUI frameworks) use a queue of callbacks to ensure events are handled in the order they arrive.

Key benefits:

- Deterministic ordering in task pipelines
- Low-latency push/pop for backtracking or streaming workloads
- Simple concurrency control when combined with locks or lock-free designs

Linked Lists

Linked lists shine in scenarios requiring frequent insertions or deletions at arbitrary points without reallocating large buffers. Memory allocators in OS kernels maintain a free-list of available blocks: when freeing memory, the block is simply linked into a free-list in $\Theta(1)$, avoiding expensive array resizes. Graph algorithms exploit adjacency lists (a map from node → linked list of neighbors) to store sparse graphs compactly and support $\Theta(1)$ insertion of new edges. LRU caches combine a hash table (for $\Theta(1)$ lookups) with a doubly linked list (to move recently used items to the front) so that eviction of the least-recently-used item is also $\Theta(1)$.

Key benefits:

- Constant-time local updates when we have a node reference
- No bulk reallocations for unbounded or unpredictable growth
- Ease of splice operations (e.g., concatenating lists, moving sublists)

Trees

Trees capture hierarchical and ordered relationships that arrays and lists cannot. File systems (NTFS, ext4) organize directories and files as B-trees or B+ trees to keep disk accesses to Θ(log n) even as millions of entries accumulate. In-memory, search trees (red-black, AVL) underpin language runtime object maps and sorted containers (e.g., C++'s `std::map`), offering Θ(log n) insert, delete, and lookup. XML/JSON parsers build a DOM tree where each element node holds pointers to child nodes, enabling recursive traversal for rendering or transformation. Even priority queues are commonly implemented as binary heaps - complete binary trees stored in arrays - combining tree structure with array-backed efficiency.

Key benefits:

- Hierarchical modeling of nested data (file systems, document object models)
- Balanced variants guarantee logarithmic operations for large datasets
- Extensible traversals (preorder, inorder, postorder) to support a variety of algorithms

When to Choose Which Data Structure

1. Dense, Fixed-Size Collections: Arrays & Matrices

When we know our maximum size ahead of time or can tolerate occasional bulk reallocations, contiguous arrays or matrices are the go-to choice. Their elements occupy adjacent memory addresses, which yields minimal per-element overhead (just the raw data) and excellent CPU cache utilization. We use a simple array when we need extremely fast random reads/writes (Θ(1)), such as lookups in a symbol table or direct mapping of pixel values in an image buffer. For multi-dimensional data - graphics, scientific grids, or neural-network weight tensors - a matrix abstraction keeps our indexing logic clear while still delivering direct access to any (i, j) cell in constant time. The trade-off is occasionally paying Θ(n) to expand capacity and potentially over-allocating memory if our usage grows slowly.

2. Unbounded or Highly Dynamic Lists: Linked Lists

If our workload involves unpredictable growth or frequent insertions and deletions in the middle of a sequence, a singly or doubly linked list may outperform array-based approaches. In a linked list, inserting or removing a known node is a constant-time pointer update (Θ(1)), with no need to shift blocks of memory or reallocate a large buffer. This makes linked lists ideal for implementing freelists in custom allocators, adjacency lists in sparse-graph representations, or LRU caches where nodes must be quickly spliced to the front or back. Their downside is

increased memory usage (one or two extra pointers per element) and poorer locality - pointer dereferences can lead to cache misses.

3. Strict LIFO vs. FIFO Queues: Stacks & Circular Buffers

When we need a stack (last-in, first-out), an array-backed implementation is trivial: push and pop at the end are amortized Θ(1), and we enjoy perfect locality. We should use this for backtracking algorithms (DFS), call-stack emulation, or undo-history buffers. For a queue (first-in, first-out), a circular-buffer array yields true Θ(1) enqueue and dequeue by wrapping head/tail indices modulo capacity, avoiding any element shifts. This pattern is perfect for producer/consumer pipelines, task schedulers, and event loops. If we cannot predefine an upper bound on queue length, we should switch to a linked-list–based queue so we never hit a size limit, accepting a small per-node pointer cost.

4. Ordered, Searchable Data: Trees & Heaps

When our application demands sorted order, range queries, or priority-based retrieval, array and list structures fall short. A self-balancing binary search tree (e.g., AVL, red-black) guarantees Θ(log n) insertion, deletion, and lookup, making it suited for language runtimes' sorted containers or in-memory indexes. B-trees and B+ trees, which store keys in arrays within each node, shine in database and file-system contexts by minimizing disk I/O through high fan-out. For priority-based scheduling (e.g., job dispatchers), a binary heap - an array-representation of a complete binary tree - supports Θ(log n) inserts and Θ(log n) extracts while maintaining compactness and cache-friendly traversals.

5. Composite & Specialized Structures

Many real-world systems layer primitives to balance trade-offs:

- **Hash Maps with Chaining:** An array of buckets, each bucket a linked list. We get average-case Θ(1) lookups with graceful handling of collisions.
- **Skip Lists:** A randomized tower of linked lists that approximates balanced-tree performance (Θ(log n)) without complex rotations.
- **Deques (Double-Ended Queues):** A doubly linked list or circular buffer allowing Θ(1) inserts/removals at both ends - useful for caching algorithms (e.g., sliding-window rate limiters).

By understanding each primitive's cost model - memory footprint, pointer overhead, locality, and operation complexity - we can mix and match to tailor data structures that meet our application's unique demands.