Name: Sandhyarani Mudindi

Student ID: 005034219

Course Title: Algorithms and Data Structures (MSCS-532-A01)

Assignment number: 6

Table of Contents

Abstract

This report presents an in-depth analysis of selection algorithms and elementary data structures, focusing on their implementation, efficiency, and practical applications. The first section examines deterministic and randomized selection algorithms, particularly the Median of Medians and Quickselect methods. The deterministic Median of Medians algorithm ensures a worst-case linear time complexity by carefully choosing a pivot through recursive median selection. This approach guarantees balanced partitioning and avoids worst-case scenarios associated with poor pivot choices. In contrast, the Quickselect algorithm, which employs random pivot selection, achieves an expected linear time complexity but can degrade to quadratic time in the worst case. The second section explores fundamental data structures, including arrays, matrices, stacks, queues, linked lists, and rooted trees. Each data structure is evaluated based on its insertion, deletion, access, and traversal complexities. The report further discusses the trade-offs between arrays and linked lists for implementing stacks and queues, emphasizing efficiency in various scenarios.

Part 1: Implementation and Analysis of Selection Algorithms

Deterministic Selection (Median of Medians)

Approach:

- The deterministic selection algorithm works by dividing the array into small groups (commonly groups of 5 elements), finding the median of each group, and then recursively selecting the median of these medians. This median of medians is used as a pivot to partition the array. The key idea is that by choosing a “good” pivot, we can guarantee that the partition is reasonably balanced, which leads to a worst-case linear time complexity.
- Steps:

1. Base Case:
    - When the array has 5 or fewer elements, the algorithm simply sorts the array and returns the kth smallest element.
2. Divide into Groups:
    - The array is partitioned into sublists (each of size 5, except possibly the last one).
3. Find Medians:
    - Each sublist is sorted to find its median. Since each group is small, sorting is constant time per group.
4. Select Pivot:
    - The medians are collected, and the algorithm recursively finds the median of these medians. This median of medians is guaranteed to be “good enough” to ensure balanced partitions.
5. Partition the Array:

    The input array is partitioned into three parts.

   - lows: All elements less than the pivot.
   - pivots: All elements equal to the pivot.
   - highs: All elements greater than the pivot.
6. Recurse Accordingly:
   - Depending on the value of k relative to the sizes of these partitions, the algorithm recurses into the appropriate sublist.

Time complexity

The deterministic selection algorithm guarantees a worst-case linear time complexity by carefully choosing a pivot that ensures a good balance in the partitioning process. The most common implementation uses groups of 5 elements.

Step-by-Step Analysis:

1) Dividing into Groups:
   - The array of size n is divided into ⌈n/5⌉ groups, each containing at most 5 elements.
   - Sorting each group takes constant time per group (since 5 is a constant), resulting in a total cost of:
     - $(\frac{n}{5} 5) = O(5)$
2) Finding the Medians:
   - After sorting each group, the median of each group is selected. This again is O(n) since there are ⌈n/5⌉ groups and choosing the median from each sorted group is O (1) per group.
3) Recursive Call on Medians:
   - The algorithm then recursively finds the median of the medians. Let's denote the time for this recursive call as T(⌈n/5⌉).

4) Partitioning Around the Pivot:
    - Once the pivot is selected (the median of medians), the entire array is partitioned into three groups: elements less than the pivot, equal to the pivot, and greater than the pivot.
    - This partitioning takes O(n) time.
5) Balanced Partitioning:
    - Because of this guarantee, after partitioning, the worst-case size of the larger subproblem is at most 7n/10 (i.e., at most 70% of the original array).

Why It Achieves O(n) Worst-Case:

Recurrence Relation

The work done at each level of the recursion (grouping, finding medians, and partitioning) is O(n). The recurrence relation for the worst-case time T(n) can be expressed as:

    ◦ $\mathrm{T(n)} <= ([\frac{n}{5}]) + T(\frac{7n}{10}) + O(n)$

- $T([\frac{n}{5}])$ corresponds to the recursive call to find the median of medians.
- $T(\frac{7n}{10})$ corresponds to the worst-case recursive call on the larger partition.
- O(n) accounts for the grouping, finding medians, and partitioning.

Solution to the Recurrence:

- A detailed analysis of this recurrence shows that it resolves to:
    ◦ T(n)=O(n)

This means that no matter how the data is arranged, the algorithm will always finish in linear time relative to the input size. The guarantee comes from the fact that the pivot selection method always ensures a significant reduction in the problem size.

Space Complexity

Auxiliary Data Structures:

Grouping into Sublists:

- The algorithm divides the input array into sublists (groups) of 5 elements. In many implementations (especially those written in high-level languages like Python), this is done by creating new lists or slices, which requires additional memory proportional to the input size.

Partition Arrays:

- During the partitioning step, three separate lists are created for elements less than, equal to, and greater than the pivot. Each of these lists can, in the worst-case, contain almost all the elements. This leads to an O(n) space requirement for these auxiliary arrays.

Overall Space Complexity:

- Worst-Case: O(n) auxiliary space (primarily due to the creation of new lists for groups and partitions).
- Stack Space: O(logn) additional space for recursion, which is negligible compared to O(n) when new arrays are allocated.

Additional Overheads

Memory Allocation:

- Repeatedly allocating new arrays for partitions may lead to overhead from memory allocation, especially in high-level languages that manage memory automatically

Grouping Overhead:

- Sorting each small group of 5 elements introduces a small constant overhead, but since each group is small, this remains O(n) overall.

Randomized Selection (Quickselect)

Approach:

- The randomized selection algorithm, commonly known as Quickselect, chooses a random pivot and partitions the array based on this pivot. It then recurses into the subarray that contains the kth smallest element. Random pivot selection generally leads to a balanced partition on average, resulting in expected linear time performance.
- Steps:

1. Base Case:
   - When the array has one element, it is returned immediately as the kth smallest element.
2. Random Pivot Selection:
   - A pivot is randomly selected from the array. This randomness is key to achieving expected linear performance.
3. Partition the Array:

   The algorithm creates three lists:
   - lows: Elements less than the pivot.
   - pivots: Elements equal to the pivot.
   - highs: Elements greater than the pivot.
4. Recurse Accordingly:
   - Depending on the value of k compared to the sizes of lows and pivots, the algorithm recurses into the correct partition.

Time complexity

Randomized Quickselect chooses a pivot at random and then partitions the array around that pivot. While the worst-case time complexity is O (n^2), the expected time complexity is O(n).

Step-by-Step Analysis

1. Random Pivot Selection:
   - A pivot is chosen uniformly at random from the array. This operation is O (1).
2. Partitioning the Array:
   - The array is partitioned into elements less than, equal to, and greater than the pivot.
   - This partitioning requires a single pass through the array, costing O(n)
3. Recursive Call:
   - Only one partition (the one that contains the kth element) is chosen for the next recursive call.
   - After partitioning, only one of the partitions is recursively processed based on the rank k.

Why It Achieves O(n) Expected Time

Average Behavior:

- On average, the randomly chosen pivot will be "reasonably good"—that is, it will tend to split the array into two parts that are not extremely unbalanced. Although it is possible to get a very unbalanced split (e.g., one part having nearly all the elements), the probability of repeatedly choosing such poor pivots is very low.

Expected Recurrence:

- In expectation, if the pivot splits the array into two roughly equal parts, the recurrence for the expected time E[T(n)] becomes:
  - $E[T(n)] = E[T(\frac{n}{2})] + O(n)$

This recurrence also solves to:

- $E[T(n)] = O(n)$

- which resolves to O(n) expected time. While it can take O (n^2) in the worst-case, the probability of that occurring is very small.

In essence, the deterministic algorithm's structured approach ensures a linear time bound in every case, whereas the randomized algorithm relies on probability to achieve linear time on average, accepting the small risk of worst-case quadratic behavior.

Space Complexity

Auxiliary Data Structures:

- Partition Arrays: Like the deterministic algorithm, the randomized Quickselect implementation typically creates three lists for elements less than, equal to, and greater than the pivot. In the worst-case, this again results in O(n) space usage.

Overall Space Complexity:

- Expected/Typical Implementation: O(n) auxiliary space when using new lists for partitions.
- Optimized In-Place Version: O(1) auxiliary space (not counting the recursion stack) and O(logn) for recursion.

Additional Overheads

Random Pivot Selection Overhead:

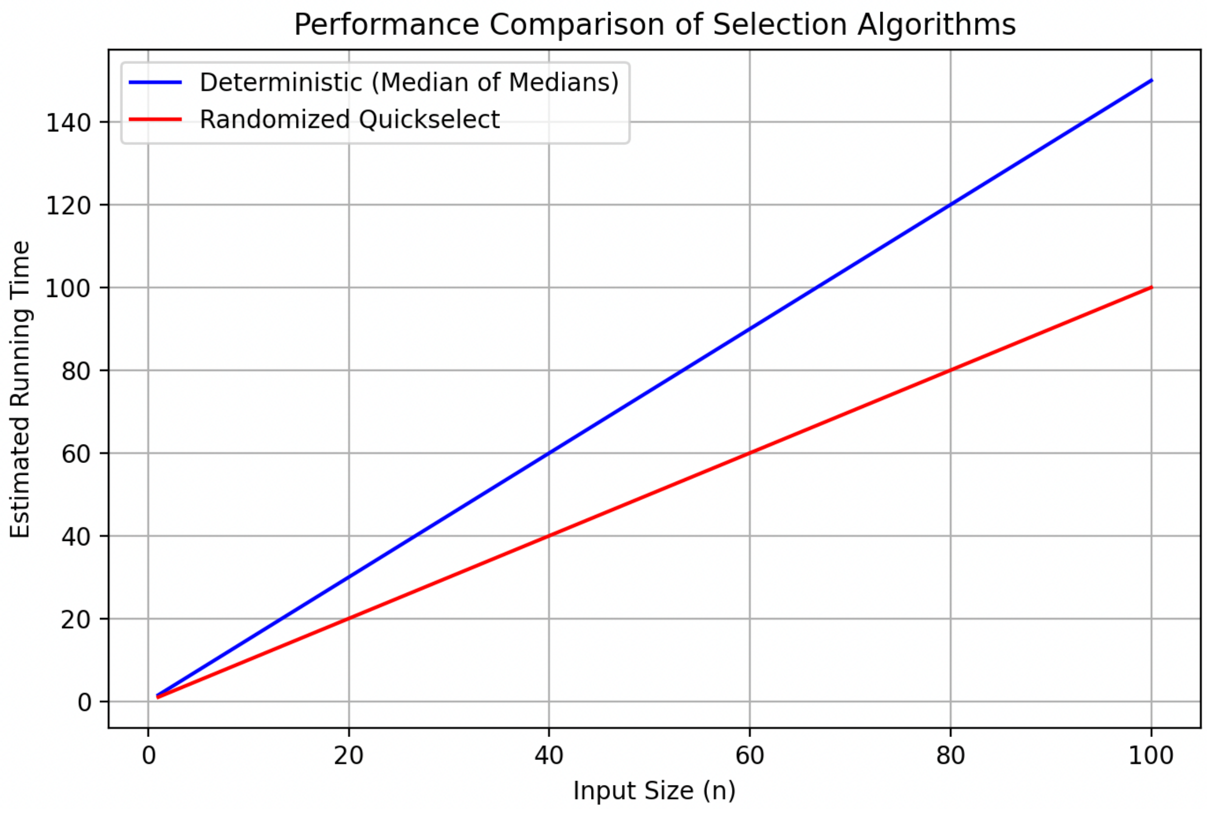
- Selecting a pivot at random is an O (1) operation. However, the randomness introduces non-deterministic behavior, which does not affect space complexity but can affect performance variability.

Memory Allocation:

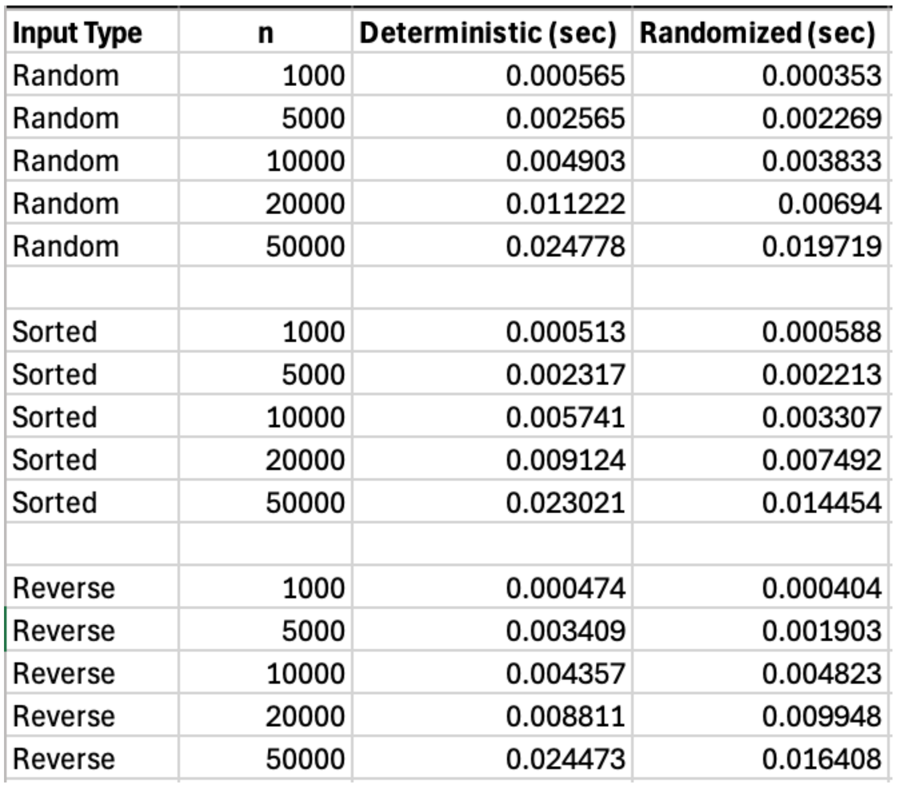
- Like the deterministic algorithm, creating new lists for each partition involves memory allocation overhead.

Comparison Table and Graphs:

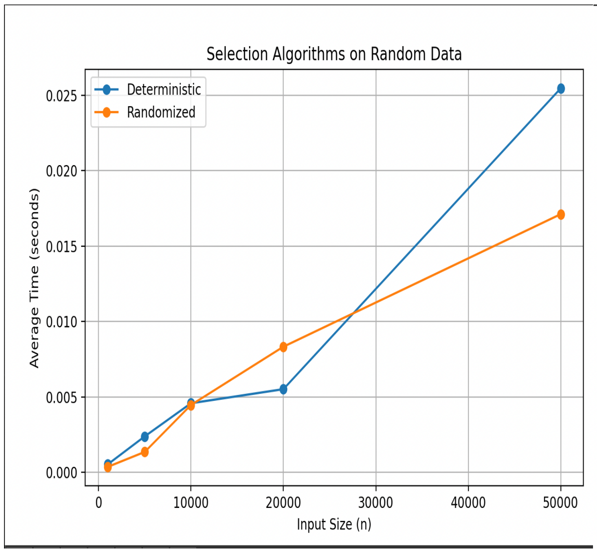
Feature	Median of Medians (Deterministic)	Randomized Quickselect
Time Complexity	O(n) (worst-case)	O(n) (average-case), O(n^2) (worst-case)
Space Complexity	O(log n) (typically)	O(log n) (average-case), O(n) (worst-case)
Pivot Selection	Deterministic (Median of Medians)	Randomized
Implementation	More complex	Simpler
Use Case	When guaranteed linear time is crucial	Generally preferred due to simplicity and good average-case performance

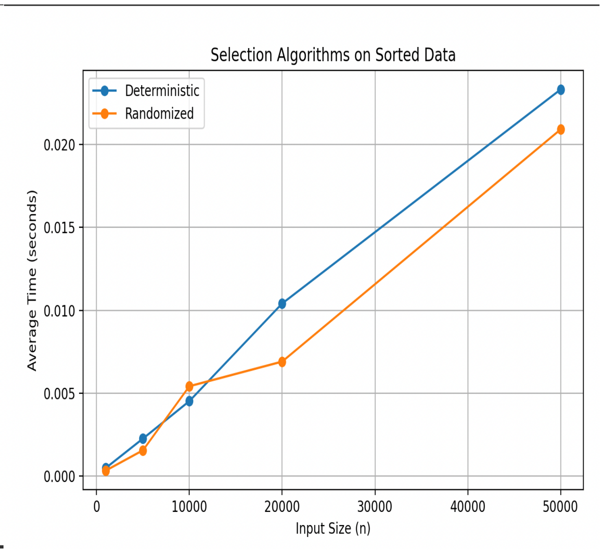


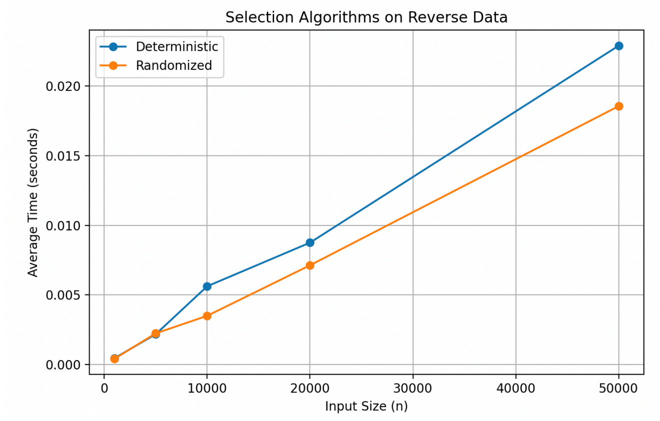
Empirical Analysis



Input Type	n	Deterministic (sec)	Randomized (sec)
Random	1000	0.000565	0.000353
Random	5000	0.002565	0.002269
Random	10000	0.004903	0.003833
Random	20000	0.011222	0.00694
Random	50000	0.024778	0.019719
Sorted	1000	0.000513	0.000588
Sorted	5000	0.002317	0.002213
Sorted	10000	0.005741	0.003307
Sorted	20000	0.009124	0.007492
Sorted	50000	0.023021	0.014454
Reverse	1000	0.000474	0.000404
Reverse	5000	0.003409	0.001903
Reverse	10000	0.004357	0.004823
Reverse	20000	0.008811	0.009948
Reverse	50000	0.024473	0.016408







- When dealing with random data, the Randomized Quickselect algorithm generally exhibits close-to-linear performance with relatively low constant factors, as the random pivot selection helps ensure balanced partitioning on average. In contrast, the Deterministic Median of Medians algorithm also maintains O(n) time complexity but incurs additional overhead due to its systematic grouping and recursive median selection, often leading to a higher constant factor. Empirical results confirm that both algorithms scale linearly with input size, though the randomized approach frequently appears faster for moderate input sizes; however, as the input size grows, the difference in execution time may diminish due to the increasing impact of constant factors.
- From a theoretical standpoint, the deterministic approach guarantees O(n) worst-case performance by systematically selecting pivots that eliminate a fixed fraction of elements at each step. In contrast, the randomized algorithm achieves expected O(n) time complexity but, in rare cases, may degrade to O (n^2) if the pivot choices are consistently poor. Despite this, such worst-case scenarios occur infrequently in practice.
- However, the randomized approach often outperforms its deterministic counterpart due to smaller constant factors, as the latter incurs additional costs from element grouping and recursive median computation.

Part 2: Elementary Data Structures Implementation and Discussion

Time Complexity

Arrays

Insertion:

- In our simple fixed-size array implementation, inserting a value at a specific index is an O (1) operation (provided the index is known) because it directly assigns a value.

Deletion:

- Our implementation simply sets an index to None, which is O(1).

Access:

- Random access by index is O(1) because arrays provide direct memory access.

Matrices

Insertion, Deletion, and Access:

- Each operation (accessing or updating a cell) in a matrix is O(1) since you compute the row and column indices and then access that element directly.

Stacks (Using Arrays)

- Push (Insertion at the end): O(1) on average (amortized O(1) when the underlying dynamic array occasionally resizes).
- Pop (Removal from the end): O(1) because it directly removes the last element.
- Peek: O(1) as it simply returns the last element.

Queues (Using Arrays)

- Enqueue (Insertion at the end): O(1) on average (amortized O(1) due to the same reasons as the stack push).

- Dequeue (Removal from the beginning): O(n) because removing the first element requires shifting all the remaining elements one position forward in a typical array-based implementation.
- Front (Access first element): O(1) as it returns the element at index 0.

Linked Lists

Insertion:

- If you insert at the beginning of a singly linked list, it's O(1).
- In our implementation, insertion is performed at the end, which requires traversal of the list, resulting in O(n) in the worst case.

Deletion:

- Finding the node to delete is O(n) in the worst case (since you might have to traverse the entire list), and the deletion itself is O(1) once the node is found.

Traversal:

- Traversal through the list is O(n) where n is the number of nodes.

Rooted Trees

Traversal:

- For a tree where each node is visited exactly once, the traversal is O(n), where n is the total number of nodes.

Insertion/Deletion:

- These operations depend on the specific tree implementation and where the insertion or deletion occurs. For example, if you need to search for the right place to insert/delete, it might take O(n) in an unbalanced tree.

Trade-offs Between Arrays and Linked Lists for Stacks and Queues

Stacks

Array-based Implementation:

Pros:

- Simple implementation with O (1) push and pop when using the end of the array.
- Good memory locality, which can lead to performance improvements due to caching.

Cons:

- Dynamic arrays may occasionally require resizing, which is an O(n) operation, but this cost is amortized.

Linked List-based Implementation:

Pros:

- O (1) push and pop operations can be achieved easily when operating at the head of the list.
- Does not require contiguous memory allocation.

Cons:

- Each element requires extra memory for the pointer/reference.
- Generally poorer locality of reference compared to arrays, which might reduce performance in some contexts.

Queues

Array-based Implementation:

Pros:

- Enqueue operation (insertion at the end) is O (1).

Cons:

- Dequeue (removal from the front) is O(n) in a simple array-based approach because it requires shifting elements. Can be optimized by implementing a circular buffer, but that adds complexity.

Linked List-based Implementation:

Pros:

- With pointers to both the head and tail, both enqueue and dequeue operations can be performed in O(1) time.
- More flexible for an unbounded queue since nodes can be added without worrying about contiguous memory allocation.

Cons:

- Increased memory overhead per element due to storing pointers.
- As with stacks, linked lists have poorer cache locality compared to arrays.

Efficiency Comparisons in Specific Scenarios

- Frequent insertions and deletions at arbitrary positions: Linked lists are generally more efficient than arrays due to the O(n) cost of shifting elements in arrays.
- Frequent access by index: Arrays are significantly faster (O (1)) than linked lists (O(n)).
- Implementing a LIFO (Last-In, First-Out) data structure: Stacks can be efficiently implemented with both arrays and linked lists (O (1) for push and pop).
- Implementing a FIFO (First-In, First-Out) data structure: Queues are efficiently implemented with linked lists (O (1) for enqueue and dequeue). Arrays have O(n) dequeue unless a circular buffer approach is used.
- Searching for an element: If the data is sorted, arrays allow for binary search (O (log n)). Unsorted arrays and linked lists require linear search (O(n)).

Practical Applications of These Data Structures in Real-World Scenarios

Arrays and Matrices:

- Image Processing: Images are represented as matrices of pixels. Operations like filtering, edge detection, and image transformations rely heavily on matrix manipulations.
- Scientific Computing: Numerical computations, simulations, and data analysis often involve large arrays and matrices. Libraries like NumPy in Python are built upon efficient array implementations.
- Game Development: Game boards, character positions, and other game data are often stored in arrays or matrices.

Stacks:

- Browser History: Web browsers use a stack to keep track of visited pages. The "back" button pops the last visited page from the stack.
- Expression Evaluation: Compilers and interpreters use stacks to evaluate arithmetic expressions (e.g., converting infix to postfix notation).

Queues:

- Task Scheduling: Operating systems use queues to manage processes waiting for CPU time or I/O operations.
- Print Queues: Printers use queues to manage print jobs. Documents are enqueued and printed in the order they were received.

Linked Lists:

- Dynamic Memory Allocation: Some memory allocation systems use linked lists to keep track of free memory blocks.

- Implementing Stacks and Queues (especially Queues): As discussed earlier, linked lists are efficient for implementing queues.

Rooted Trees:

- File Systems: File systems use tree structures to organize files and directories.
- Organization Charts: Companies use tree structures to represent the hierarchy of employees and departments.
- Decision Trees: In machine learning, decision trees are used for classification and regression tasks.

Scenarios Where One Data Structure Is Preferred:

- Memory usage is a primary concern: Arrays can be more memory-efficient if the size is relatively fixed and known in advance. Linked lists have pointer overhead.
- Ease of implementation: Arrays are often simpler to implement for basic use cases.
- Searching for an element: If the data is sorted, arrays enable binary search (O(log n)). If frequent searches are required, hash tables (not in the original prompt, but important) are often the best choice for average-case O (1) lookup.
- Frequent access by index: Arrays are the clear winner due to O (1) access time. Use arrays for lookups when the index is known.
- Frequent insertions/deletions in the middle of the sequence: Linked lists are preferred as they avoid the O(n) element shifting required by arrays. Use linked lists when you expect lots of modifications to the data.