perplexity

Comprehensive Time Complexity Tables for Data Structures, Containers, and Algorithms

Based on extensive research, I have compiled comprehensive time complexity tables covering all major data structures, containers, and algorithms. This analysis includes **best case**, **average case**, and **worst case** scenarios along with **space complexity** for each operation.

Data Structures Operations Complexity

Data Structure	Access/Get	Search	Insert	Delete	Space Complexity
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Dynamic Array	O(1)	O(n)	O(1)*	O(n)	O(n)
Stack	O(1)	O(n)	O(1)	O(1)	O(n)
Queue	O(1)	O(n)	O(1)	O(1)	O(n)
Singly Linked List	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly Linked List	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table (Average)	O(1)	O(1)	O(1)	O(1)	O(n)
Hash Table (Worst)	O(n)	O(n)	O(n)	O(n)	O(n)
Binary Search Tree (Average)	O(log n)	O(log n)	O(log n) (n)		
Binary Search Tree (Worst)	O(n)	O(n)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Red-Black Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
B-Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Heap (Binary)	N/A	O(n)	O(log n)	O(log n)	O(n)
Skip List (Average)	O(log n)	O(log n)	O(log n)	O(log n)	O(n log n)
Skip List (Worst)	O(n)	O(n)	O(n)	O(n)	O(n log n)
Trie	O(m)	O(m)	O(m)	O(m)	O(ALPHABET_SIZE × average_key_length × N)

Note: Dynamic Array insertion is O(1) amortized, but O(n) in worst case due to resizing[1][2][3][4]

Sorting Algorithms Complexity

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort (Average)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$
Radix Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$
Bucket Sort (Average)	$O(n + k)$	$O(n)$	$O(n^2)$	$O(n + k)$
Shell Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
Tim Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Where n = number of elements, k = range of input values[5][6][7]

Searching Algorithms Complexity

Algorithm	Time Complexity	Space Complexity	Requirements
Linear Search	$O(n)$	$O(1)$	None
Binary Search	$O(\log n)$	$O(1)$	Sorted array
Jump Search	$O(\sqrt{n})$	$O(1)$	Sorted array
Interpolation Search (Average)	$O(\log \log n)$	$O(1)$	Uniformly distributed sorted array
Interpolation Search (Worst)	$O(n)$	$O(1)$	Uniformly distributed sorted array
Exponential Search	$O(\log n)$	$O(\log n)$	Sorted array
Fibonacci Search	$O(\log n)$	$O(1)$	Sorted array
Ternary Search	$O(\log n)$	$O(1)$	Sorted array

Where n = number of elements[8][9][10]

Graph Algorithms Complexity

Algorithm	Time Complexity	Space Complexity	Use Case
Breadth-First Search (BFS)	O(V + E)	O(V)	Shortest path (unweighted)
Depth-First Search (DFS)	O(V + E)	O(V)	Traversal, cycle detection
Dijkstra's Algorithm	O((V + E) log V)	O(V + E)	Shortest path (non-negative weights)
Bellman-Ford Algorithm	O(V × E)	O(V)	Shortest path (negative weights)
Floyd-Warshall Algorithm	$O(V^3)$	$O(V^2)$	All-pairs shortest path
Kruskal's Algorithm	O(E log V)	O(V)	Minimum spanning tree
Prim's Algorithm	O((V + E) log V)	O(V + E)	Minimum spanning tree
Topological Sort	O(V + E)	O(V)	Dependency resolution
Tarjan's SCC	O(V + E)	O(V)	Strongly connected components
*A Search**	O(b^d)	O(b^d)	Pathfinding with heuristic

Where V = vertices, E = edges, b = branching factor, d = depth[11][12][13]

String Matching Algorithms Complexity

Algorithm	Time Complexity	Space Complexity	Preprocessing
Naive String Matching	O(nm)	O(1)	None
Knuth-Morris-Pratt (KMP)	O(n + m)	O(m)	O(m)
Boyer-Moore	O(nm) worst, O(n/m) average	O(σ)	O(m + σ)
Rabin-Karp	O(nm) worst, O(n + m) average	O(1)	O(m)
Z Algorithm	O(n + m)	O(n + m)	O(m)
Manacher's Algorithm	O(n)	O(n)	None
Aho-Corasick	O(n + m + z)	O(m × σ)	O(m × σ)
Suffix Array Construction	O(n log n)	O(n)	O(n log n)

Where n = text length, m = pattern length, σ = alphabet size, z = number of matches[14][15][16][17]

Dynamic Programming Problems Complexity

Problem	Time Complexity	Space Complexity
Fibonacci (Naive)	O(2^n)	O(n)
Fibonacci (DP)	O(n)	O(n)
Longest Common Subsequence	O(mn)	O(mn)
Edit Distance (Levenshtein)	O(mn)	O(mn)

Problem	Time Complexity	Space Complexity
0/1 Knapsack	O(nW)	O(nW)
Coin Change	O(nW)	O(nW)
Longest Increasing Subsequence	O(n log n)	O(n)
Matrix Chain Multiplication	$O(n^3)$	$O(n^2)$
Optimal Binary Search Tree	$O(n^3)$	$O(n^2)$
Palindrome Partitioning	$O(n^3)$	$O(n^2)$

Where n, m = input sizes, W = weight/capacity constraint[18][19][20][21][22]

Specialized Data Structures

Union-Find (Disjoint Set Union) Complexity

Implementation	Union Time	Find Time	Space Complexity
Quick Find	O(n)	O(1)	O(n)
Quick Union	O(n)	O(n)	O(n)
Union by Rank	O(log n)	O(log n)	O(n)
Path Compression	O(n)	O(n) worst, O(1) amortized	O(n)
Union by Rank + Path Compression	$O(\alpha(n))$	$O(\alpha(n))$	O(n)

Where $\alpha(n)$ is the inverse Ackermann function, effectively constant for practical purposes[23][24][25][26][27]

Trie Data Structure Complexity

Operation	Time Complexity	Space Complexity
Insert	O(m)	O(m)
Search	O(m)	O(1)
Delete	O(m)	O(1)
Build Trie	O(N × avgL)	O(ALPHABET_SIZE × avgL × N)

Where m = key length, N = number of keys, avgL = average key length[28][29][30][31][32]

Key Insights and Notes

Big O Notation Hierarchy

The complexity classes in order of efficiency (best to worst):

- **O(1)** - Constant time[33][34]
- **O(log n)** - Logarithmic time[33][34]
- **O(n)** - Linear time[33][34]
- **O(n log n)** - Linearithmic time[33][34]
- **O(n^2)** - Quadratic time[33][34]
- **O(2^n)** - Exponential time[33][34]
- **O(n!)** - Factorial time[33][34]

Space Complexity Considerations

Space complexity includes both **auxiliary space** (extra memory used by algorithm) and **input space** (memory for storing input)[35][36]. Many algorithms can be optimized to use less space at the cost of time complexity.

Amortized Analysis

Some operations like dynamic array insertion have **amortized complexity** that differs from worst-case complexity. Amortized analysis provides the average performance over a sequence of operations[37][38].

Hash Table Performance

Hash table performance heavily depends on the **load factor** and **hash function quality**. With proper implementation, hash tables achieve O(1) average case for all operations[39][40][41][42].

This comprehensive analysis covers the fundamental time and space complexities you need to understand for algorithm design, interview preparation, and system optimization. The notation O(n), O(log n), etc. represents the **upper bound** of growth rate as input size increases[43][44][45].

Comprehensive Time Complexity Tables for C++ STL Containers and Operations

Based on extensive research of C++ STL containers, I have compiled detailed time complexity tables covering all major container types and their operations. This analysis focuses specifically on C++ STL containers with their underlying data structures and complexity guarantees.

Sequence Containers

Container	Access	Search	Insert (Position)	Insert (End)	Insert (Front)	Delete	Space Complexity
std::array	O(1)	O(n)	N/A	N/A	N/A	N/A	O(n)
std::vector	O(1)	O(n)	O(n)	O(1) amortized	O(n)	O(n)	O(n)
std::deque	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)
std::list	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(n)
std::forward_list	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)	O(n)

Key Details for Sequence Containers:

std::vector[1] [2] [41** with contiguous memory

- `push_back()` is O(1) amortized due to capacity doubling
- Random access via `operator[]` and `at()` in O(1)
- Insert and erase operations require shifting elements

std::deque[1] [3] [4]:

- Implemented as **segmented array** (double-ended queue)
- Efficient insertion/deletion at both ends
- Random access in O(1) but slightly slower than vector

std::list[1] [2] [5]:

- **Doubly linked list** implementation
- No random access (requires traversal)
- Efficient splice operations in O(1)
- Higher memory overhead due to node pointers

std::forward_list[6] [7] [8]:

- **Singly linked list** with forward-only traversal
- More memory efficient than `std::list`
- No `size()` function (would be O(n))
- Optimal for insertion-heavy scenarios with minimal memory

std::array[9] [10] [11]:

- **Fixed-size array** wrapper over C-style arrays
- No overhead compared to raw arrays
- Size must be known at compile time
- All operations except access are not applicable

Associative Containers (Ordered)

Container	Insert	Search/Find	Delete	Count	Lower/Upper Bound	Space Complexity
std::set	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
std::multiset	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
std::map	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
std::multimap	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

Implementation Details:

All ordered associative containers use **Red-Black Trees** [12] [13] [14]:

- Self-balancing binary search trees
- Maintain O(log n) height guarantee
- Elements stored in sorted order
- `multiset` and `multimap` allow duplicate keys using **Threaded Red Black Tree** [12]

Unordered Associative Containers (Hash-based)

Container	Insert	Search/Find	Delete	Space Complexity	Load Factor Impact
std::unordered_set	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(n)	Critical
std::unordered_multiset	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(n)	Critical
std::unordered_map	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(n)	Critical
std::unordered_multimap	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(1) avg, O(n) worst	O(n)	Critical

Hash Container Considerations:

Performance Dependency [15] [16] [17]:

- **Hash function quality** critically affects performance
- **Load factor** management prevents worst-case O(n) behavior
- Worst-case occurs when many elements hash to same bucket
- Average case assumes uniform distribution

When to Choose Hash vs Ordered[18] [19]:

- Use `unordered_map` when you need O(1) operations and don't require sorting
- Use `map` when you need guaranteed O(log n) and sorted iteration
- For small datasets, `map` might outperform due to lower overhead

Container Adapters

Container Adapter	Underlying Default	Push	Pop	Top/Front	Back	Space Complexity
std::stack	`std::deque`	O(1)	O(1)	O(1)	N/A	O(n)
std::queue	`std::deque`	O(1)	O(1)	O(1)	O(1)	O(n)
std::priority_queue	`std::vector` (heap)	O(log n)	O(log n)	O(1)	N/A	O(n)

Container Adapter Details:

std::stack[20] [21] [22]:

- LIFO (Last In, First Out) interface
- Can use `vector`, `deque`, or `list` as underlying container
- Default `deque` provides O(1) for all operations

std::queue[20] [23] [24]:

- FIFO (First In, First Out) interface
- Default `deque` supports efficient front and back operations
- Can implement with two stacks for O(1) amortized operations

std::priority_queue[1] [25] [26] [27]:

- Implemented as **max-heap** by default
- Built on `vector` with heap operations
- Construction from range: O(n)
- Element-by-element construction: O(n log n)

String Container

Operation	Time Complexity	Notes
Index Access []	O(1)	Random access like vector
Concatenation +	O(m+n)	Creates new string
Append +=	O(1) amortized	May cause reallocation
Insert/Erase	O(n)	Requires shifting characters

Operation	Time Complexity	Notes
Find/Search	O(n×m) worst, O(n) average	Depends on algorithm
Size/Length	O(1)	Stored as member variable

std::string Implementation[28] [29]:

- Acts like `std::vector<char>` since C++11
- Contiguous memory storage
- Small String Optimization (SSO) in many implementations
- Dynamic resizing with amortized O(1) append

Utility Containers

std::pair and std::tuple

Operation	Time Complexity	Space Complexity
Construction	O(1)	O(1)
Access (first/second)	O(1)	O(1)
Copy/Assignment	O(1)	O(1)
Comparison	O(1)	O(1)

std::pair[30] provides simple two-element container with constant-time operations.

std::tuple[31] [32] generalizes pair to n elements with same O(1) access complexity.

Specialized Containers

Container	Access	Operations	Space Efficiency
std::bitset	O(1)	Bitwise ops: O(n)	8x more efficient than bool array
std::valarray	O(1)	Math ops: O(n)	Optimized for numeric computation

std::bitset[33] [34] [35]:

- Fixed-size bit array
- Bitwise operations process multiple bits simultaneously
- Space efficient: 1 bit per boolean vs 8 bits for `bool`
- Not a full STL container (no iterators)

std::valarray[36] [37]:

- Vector-like container optimized for mathematical operations
- Built-in mathematical functions (sin, cos, etc.)
- Slice operations for subarray access

- Not a full STL container

Performance Guidelines and Best Practices

Choosing the Right Container:

For Sequential Access[38]:

- **std::vector**: Default choice for dynamic arrays
- **std::array**: When size is known at compile time
- **std::deque**: When you need efficient front insertion

For Associative Storage[18]:

- **std::map**: When you need sorted keys and guaranteed O(log n)
- **std::unordered_map**: When you need fastest average access and don't require sorting

For Stack/Queue Operations[22]:

- **std::stack**: For LIFO operations
- **std::queue**: For FIFO operations
- **std::priority_queue**: When elements have priority ordering

Memory Layout Impact:

Cache Performance[3] [39]:

- Contiguous containers (`vector`, `array`, `string`) have better cache locality
- Linked containers (`list`, `forward_list`) have poor cache performance
- Choose contiguous storage when possible for better performance

Amortized Complexity:

Many operations marked as O(1) are **amortized constant time**[4] [40]:

- `vector::push_back()` is O(1) amortized but O(n) worst case
- Hash table operations depend on load factor management
- Understanding amortized analysis is crucial for performance prediction

This comprehensive analysis provides the foundation for making informed decisions about C++ STL container selection based on specific performance requirements and use case constraints.

⁂

1. https://www.geeksforgeeks.org/cpp/analysis-of-time-and-space-complexity-of-stl-containers/
2. https://alyssaq.github.io/stl-complexities/
3. https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html

4. https://stackoverflow.com/questions/28266382/time-complexity-of-removing-items-in-vectors-and-deque/28266883
5. https://codefinity.com/courses/v2/212d3d3e-af15-4df9-bb13-5cbbb8114954/58324ed0-9644-4e88-ba8c-e93d15b8697a/7272fc45-26bf-4235-9167-fb4f46e618f6
6. https://www.geeksforgeeks.org/cpp/forward-list-c-set-1-introduction-important-functions/
7. https://www.thejat.in/learn/forward-list-in-stl
8. https://stackoverflow.com/questions/21786787/why-using-forward-list-with-char-is-much-more-optimize-than-using-it-with-long-l
9. https://embeddedartistry.com/blog/2017/06/28/an-introduction-to-stdarray/
10. https://www.modernescpp.com/index.php/std-array-dynamic-memory-no-thanks/
11. https://www.intel.com/content/www/us/en/docs/sycl/introduction/latest/05-array.html
12. https://www.geeksforgeeks.org/cpp/internal-data-structures-and-time-complexity-table-of-all-the-cpp-stl-containers/
13. https://stackoverflow.com/questions/222658/multiset-map-and-hash-map-complexity
14. https://stackoverflow.com/questions/222658/multiset-map-and-hash-map-complexity/68866247
15. https://stackoverflow.com/questions/15470948/c-unordered-map-complexity
16. https://timsong-cpp.github.io/cppwp/n4140/unord.req
17. https://dev.to/kevinrawal/tle-tales-when-unorderedmap-bites-you-5h7c
18. https://stackoverflow.com/questions/13799593/how-to-choose-between-map-and-unordered-map
19. https://www.simplilearn.com/tutorials/cpp-tutorial/unordered-map-cpp
20. https://stackoverflow.com/questions/63179545/time-complexity-of-queue-in-c
21. https://stackoverflow.com/questions/41316337/time-complexity-for-operations-in-implementation-of-stack-and-queue
22. https://www.geeksforgeeks.org/cpp/container-adapter-in-cpp/
23. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-analysis-of-queue-operations/
24. https://stackoverflow.com/questions/69931635/implement-queue-using-2-stacks-with-constant-time-complexity/69932432
25. https://stackoverflow.com/questions/67306334/time-complexity-of-priority-queue-in-c
26. https://www.geeksforgeeks.org/cpp/priority-queue-in-cpp-stl/
27. https://dev.to/dinhluanbmt/c-about-priorityqueuemin-max-heap-cap
28. https://runestone.academy/ns/books/published/cppds2/algorithm-analysis_analysis-of-string-operators.html
29. https://runestone.academy/ns/books/published/cppds/AlgorithmAnalysis/StringAnalysis.html
30. https://www.baeldung.com/java-pairs
31. https://pythonwife.com/tuples-in-python/
32. https://stackoverflow.com/questions/60568734/what-is-the-time-complexity-of-checking-membership-in-tuples/60569771
33. https://www.youtube.com/watch?v=jqJ5s077OKo
34. https://stackoverflow.com/questions/44520123/c-bitset-logical-operations-in-olog-n
35. https://www.geeksforgeeks.org/cpp/cpp-bitset-and-its-application/

36. https://www.linuxtopia.org/online_books/programming_books/c++_practical_programming/c++_practical_programming_202.html
37. https://stdcxx.apache.org/doc/stdlibug/22-7.html
38. https://www.modernescpp.com/index.php/c-core-guidelines-std-array-and-std-vector-are-your-friends/
39. https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.wp
40. https://yourbasic.org/algorithms/time-complexity-arrays/