Arrays (dynamic array/list)

Given `n = arr.length`,

- Add or remove element at the end: $O(1)$ amortized
- Add or remove element from arbitrary index: $O(n)$
- Access or modify element at arbitrary index: $O(1)$
- Check if element exists: $O(n)$
- Two pointers: $O(n \cdot k)$, where k is the work done at each iteration, includes sliding window
- Building a prefix sum: $O(n)$
- Finding the sum of a subarray given a prefix sum: $O(1)$

Strings (immutable)

Given `n = s.length`,

- Add or remove character: $O(n)$
- Access element at arbitrary index: $O(1)$
- Concatenation between two strings: $O(n+m)$, where m is the length of the other string
- Create substring: $O(m)$, where m is the length of the substring
- Two pointers: $O(n \cdot k)$, where k is the work done at each iteration, includes sliding window
- Building a string from joining an array, string builder, etc.: $O(n)$

Linked Lists

Given n as the number of nodes in the linked list,

- Add or remove element given pointer before add/removal location: $O(1)$
- Add or remove element given pointer at add/removal location: $O(1)$ if doubly linked
- Add or remove element at arbitrary position without pointer: $O(n)$
- Access element at arbitrary position without pointer: $O(n)$
- Check if element exists: $O(n)$
- Reverse between position `i` and `j`: $O(j-i)$
- Detect a cycle: $O(n)$ using fast-slow pointers or hash map

Hash table/dictionary

Given `n = dic.length`,

- Add or remove key-value pair: $O(1)$
- Check if key exists: $O(1)$
- Check if value exists: $O(n)$
- Access or modify value associated with key: $O(1)$
- Iterate over all keys, values, or both: $O(n)$

Note: the $O(1)$ operations are constant relative to `n`. In reality, the hashing algorithm might be expensive. For example, if your keys are strings, then it will cost $O(m)$ where m is the length of the string. The operations only take constant time relative to the size of the hash map.

Set

Given `n = set.length`,

- Add or remove element: $O(1)$
- Check if element exists: $O(1)$

The above note applies here as well.

Stack

Stack operations are dependent on their implementation. A stack is only required to support pop and push. If implemented with a dynamic array:

Given `n = stack.length`,

- Push element: $O(1)$
- Pop element: $O(1)$
- Peek (see element at top of stack): $O(1)$
- Access or modify element at arbitrary index: $O(1)$
- Check if element exists: $O(n)$

Queue

Queue operations are dependent on their implementation. A queue is only required to support dequeue and enqueue. If implemented with a doubly linked list:

Given `n = queue.length`,

- Enqueue element: $O(1)$
- Dequeue element: $O(1)$
- Peek (see element at front of queue): $O(1)$
- Access or modify element at arbitrary index: $O(n)$
- Check if element exists: $O(n)$

Note: most programming languages implement queues in a more sophisticated manner than a simple doubly linked list. Depending on implementation, accessing elements by index may be faster than $O(n)$, or $O(n)$ but with a significant constant divisor.

Binary tree problems (DFS/BFS)

Given n as the number of nodes in the tree,

Most algorithms will run in $O(n \cdot k)$ time, where k is the work done at each node, usually $O(1)$. This is just a general rule and not always the case. We are assuming here that BFS is implemented with an efficient queue.

Binary search tree

Given n as the number of nodes in the tree,

- Add or remove element: $O(n)$ worst case, $O(\log n)$ average case
- Check if element exists: $O(n)$ worst case, $O(\log n)$ average case

The average case is when the tree is well balanced - each depth is close to full. The worst case is when the tree is just a straight line.

Heap/Priority Queue

Given `n = heap.length` and talking about min heaps,

- Add an element: $O(\log n)$
- Delete the minimum element: $O(\log n)$
- Find the minimum element: $O(1)$
- Check if element exists: $O(n)$

Binary search

Binary search runs in $O(\log n)$ in the worst case, where n is the size of your initial search space.

Miscellaneous

- Sorting: $(n \cdot \log n)$, where n is the size of the data being sorted
- DFS and BFS on a graph: $O(n \cdot k + e)$, where n is the number of nodes, e is the number of edges, if each node is handled in $O(1)$ other than iterating over edges
- DFS and BFS space complexity: typically $O(n)$, but if it's in a graph, might be $O(n+e)$ to store the graph
- Dynamic programming time complexity: $O(n \cdot k)$, where n is the number of states and k is the work done at each state
- Dynamic programming space complexity: $O(n)$, where n is the number of states