Data Structures

Resumo

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)

Arrays

Key-Terms

Array

A linear collection of data values that are accessible at numbered indices, starting at index 0.

The following are an array's standard operations and their corresponding time complexities:

- **Accessing a value at a given index**: O(1)
- **Updating a value at a given index**: O(1)
- **Inserting a value at the beginning**: O(n)
- **Inserting a value in the middle**: O(n)
- **Inserting a value at the end**:
  - amortized O(1) when dealing with a **dynamic array**
  - O(n) when dealing with a **static array**
- **Removing a value at the beginning**: O(n)
- **Removing a value in the middle**: O(n)
- **Removing a value at the end**: O(1)
- **Copying the array**: O(n)

A static array is an implementation of an array that allocates a fixed amount of memory to be used for storing the array's values. Appending values to the array therefore involves copying the entire array and allocating new memory for it, accounting for the extra space needed for the newly appended value. This is a linear-time operation.

A dynamic array is an implementation of an array that preemptively allocates double the amount of memory needed to store the array's values. Appending values to the array is a constant-time operation until the allocated memory is filled up, at which point the array is copied and double the memory is once again allocated for it. This implementation leads to an amortized constant-time insertion-at-end operation.

A lot of popular programming languages like JavaScript and Python implement arrays as dynamic arrays.

Linked List

Singly Linked List

A data structure that consists of nodes, each with some value and a pointer to the next node in the linked list. A linked list node's value and next node are typically stored in `value` and `next` properties, respectively.

The first node in a linked list is referred to as the **head** of the linked list, while the last node in a linked list, whose `next` property points to the `null` value, is known as the **tail** of the linked list.

Below is a visual representation of a singly linked list whose nodes hold integer values:

```
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> null
```

A singly linked list typically exposes its head to its user for easy access. While finding a node in a singly linked list involves traversing through all of the nodes leading up to the node in question (as opposed to instant access with an array), adding or removing nodes simply involves overwriting `next` pointers (assuming that you have access to the node right before the node that you're adding or removing).

The following are a singly linked list's standard operations and their corresponding time complexities:

- **Accessing the head**: O(1)
- **Accessing the tail**: O(n)
- **Accessing a middle node**: O(n)
- **Inserting / Removing the head**: O(1)
- **Inserting / Removing the tail**: O(n) to access + O(1)
- **Inserting / Removing a middle node**: O(n) to access + O(1)
- **Searching for a value**: O(n)

Doubly Linked List

Similar to a **singly linked list**, except that each node in a doubly linked list also has a pointer to the previous node in the linked list. The previous node is typically stored in a `prev` property.

Just as the `next` property of a doubly linked list's **tail** points to the `null` value, so too does the `prev` property of a doubly linked list's **head**.

Below is a visual representation of a doubly linked list whose nodes hold integer values:

```
null <- 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 -> null
```

While a doubly linked list typically exposes both its head and tail to its user, as opposed to just its head in the case of a singly linked list, it otherwise behaves very similarly to a singly linked list.

The following are a doubly linked list's standard operations and their corresponding time complexities:

- **Accessing the head**: O(1)
- **Accessing the tail**: O(1)
- **Accessing a middle node**: O(n)
- **Inserting / Removing the head**: O(1)
- **Inserting / Removing the tail**: O(1)
- **Inserting / Removing a middle node**: O(n) to access + O(1)
- **Searching for a value**: O(n)

Circular Linked List

A linked list that has no clear **head** or **tail**, because its "tail" points to its "head," effectively forming a closed circle.

A circular linked list can be either a **singly circular linked list** or a **doubly circular linked list**.

Hash Tables

Hash Table

A data structure that provides fast insertion, deletion, and lookup of key/value pairs.

Under the hood, a hash table uses a **dynamic array** of **linked lists** to efficiently store key/value pairs. When inserting a key/value pair, a hash function first maps the key, which is typically a string (or any data that can be hashed, depending on the implementation of the hash table), to an integer value and, by extension, to an index in the underlying dynamic array. Then, the value associated with the key is added to the linked list stored at that index in the dynamic array, and a reference to the key is also stored with the value.

Hash tables rely on highly optimized hash functions to minimize the number of **collisions** that occur when storing values: cases where two keys map to the same index.

Below is an example of what a hash table might look like under the hood:

```
[
  0: (value1, key1) -> null
  1: (value2, key2) -> (value3, key3) -> (value4, key4)
  2: (value5, key5) -> null
  3: (value6, key6) -> null
  4: null
  5: (value7, key7) -> (value8, key8)
  6: (value9, key9) -> null
]
```

In the hash table above, the keys **key2**, **key3**, and **key4** collided by all being hashed to **1**, and the keys **key7** and **key8** collided by both being hashed to **5**.

The following are a hash table's standard operations and their corresponding time complexities:

- **Inserting a key/value pair**: O(1) on average; O(n) in the worse case
- **Removing a key/value pair**: O(1) on average; O(n) in the worse case
- **Looking up a key**: O(1) on average; O(n) in the worse case

The worst-case linear-time operations occur when a hash table experiences a lot of collisions, leading to long linked lists internally, which take O(n) time to traverse.

However, in practice and especially in coding interviews, we typically assume that the hash functions employed by hash tables are so optimized that collisions are extremely rare and constant-time operations are all but guaranteed.

Stack

Stack

An array-like data structure whose elements follow the **LIFO** rule: **L**ast **I**n, **F**irst **O**ut.

A stack is often compared to a stack of books on a table: the last book that's placed on the stack of books is the first one that's taken off the stack.

The following are a stack's standard operations and their corresponding time complexities:

- **Pushing an element onto the stack**: O(1)
- **Popping an element off the stack**: O(1)
- **Peeking at the element on the top of the stack**: O(1)
- **Searching for an element in the stack**: O(n)

A stack is typically implemented with a **dynamic array** or with a **singly linked list**.

Queue

Queue

An array-like data structure whose elements follow the **FIFO** rule: **F**irst **I**n, **F**irst **O**ut.

A queue is often compared to a group of people standing in line to purchase items at a store: the first person to get in line is the first one to purchase items and to get out of the queue.

The following are a queue's standard operations and their corresponding time complexities:

- **Enqueuing an element into the queue**: O(1)
- **Dequeuing an element out of the queue**: O(1)
- **Peeking at the element at the front of the queue**: O(1)
- **Searching for an element in the queue**: O(n)

A queue is typically implemented with a **doubly linked list**.

String

String

One of the fundamental data types in Computer Science, strings are stored in **memory** as **arrays** of integers, where each character in a given string is mapped to an integer via some character-encoding standard like **ASCII**.

Strings behave much like normal arrays, with the main distinction being that, in most programming languages (C++ is a notable exception), strings are **immutable**, meaning that they can't be edited after creation. This also means that simple operations like appending a character to a string are more expensive than they might appear.

The canonical example of an operation that's deceptively expensive due to string immutability is the following:

```
string = "this is a string"
newString = ""

for character in string:
    newString += character
```

The operation above has a time complexity of **O(n^2)** where n is the length of `string`, because each addition of a character to `newString` creates an entirely new string and is itself an **O(n)** operation. Therefore, n O(n) operations are performed, leading to an O(n^2) time-complexity operation overall.

Graphs

Graph

A collection of nodes or values called **vertices** that might be related; relations between vertices are called **edges**.

Many things in life can be represented by graphs; for example, a social network can be represented by a graph whose vertices are users and whose edges are friendships between the users. Similarly, a city map can be represented by a graph whose vertices are locations in the city and whose edges are roads between the locations.

Graph Cycle

Simply put, a cycle occurs in a **graph** when three or more **vertices** in the graph are connected so as to form a closed loop.

Note that the definition of a graph cycle is sometimes broadened to include cycles of length two or one; in the context of coding interviews, when dealing with questions that involve graph cycles, it's important to clarify what exactly constitutes a cycle.

Acyclic Graph

A **graph** that has no **cycles**.

Cyclic Graph

A **graph** that has at least one **cycle**.

Directed Graph

A **graph** whose **edges** are directed, meaning that they can only be traversed in one direction, which is specified.

For example, a graph of airports and flights would likely be directed, since a flight specifically goes from one airport to another (i.e., it has a direction), without necessarily implying the presence of a flight in the opposite direction.

Undirected Graph

A **graph** whose **edges** are undirected, meaning that they can be traversed in both directions.

For example, a graph of friends would likely be undirected, since a friendship is, by nature, bidirectional.

Connected Graph

A **graph** is connected if for every pair of **vertices** in the graph, there's a path of one or more **edges** connecting the given vertices.

In the case of a **directed graph**, the graph is:

- **strongly connected** if there are bidirectional connections between the vertices of every pair of vertices (i.e., for every vertex-pair `(u, v)` you can reach `v` from `u` and `u` from `v`)
- **weakly connected** if there are connections (but not necessarily bidirectional ones) between the vertices of every pair of vertices

A graph that isn't connected is said to be **disconnected**.

Trees

Tree

A data structure that consists of nodes, each with some value and pointers to child-nodes, which recursively form **subtrees** in the tree.

The first node in a tree is referred to as the **root** of the tree, while the nodes at the bottom of a tree (the nodes with no child-nodes) are referred to as **leaf nodes** or simply **leaves**. The paths between the root of a tree and its leaves are called **branches**, and the **height** of a tree is the length of its longest branch. The **depth** of a tree node is its distance from its tree's root; this is also known as the node's **level** in the tree.

A tree is effectively a **graph** that's **connected**, **directed**, and **acyclic**, that has an explicit root node, and whose nodes all have a single **parent** (except for the root node, which effectively has no parent). Note that in most implementations of trees, tree nodes don't have a pointer to their parent, but they can if desired.

There are many types of trees and tree-like structures, including **binary trees**, **heaps**, and **tries**.

Binary Tree

A **tree** whose nodes have up to **two** child-nodes.

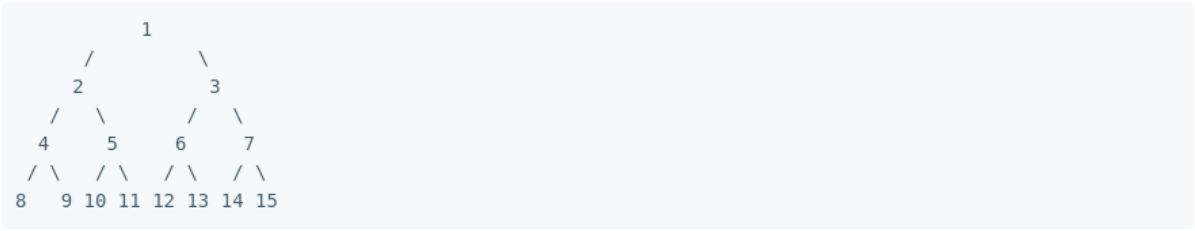
The structure of a binary tree is such that many of its operations have a logarithmic time complexity, making the binary tree an incredibly attractive and commonly used data structure.

K-ary Tree

A **tree** whose nodes have up to **k** child-nodes. A **binary tree** is a k-ary tree where **k == 2**.

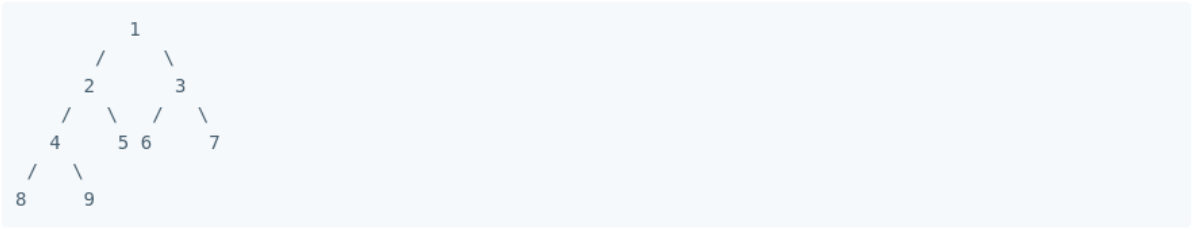
Perfect Binary Tree

A **binary tree** whose interior nodes all have two child-nodes and whose **leaf nodes** all have the same **depth**. Example:

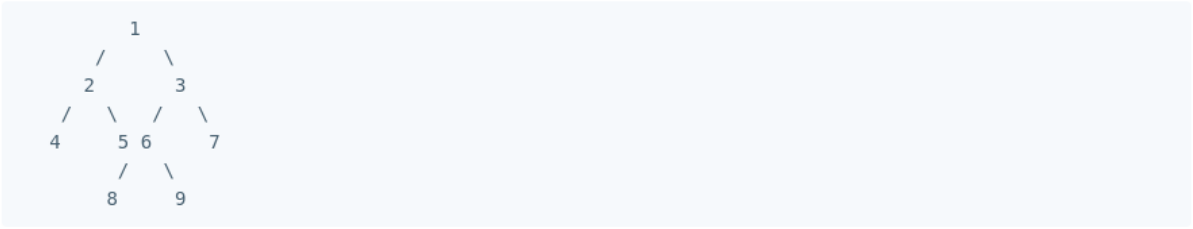


Complete Binary Tree

A **binary tree** that's *almost* **perfect**; its interior nodes all have two child-nodes, but its **leaf nodes** don't necessarily all have the same **depth**. Furthermore, the nodes in the last **level** of a complete binary tree are as far left as possible. Example:



Conversely, the following binary tree *isn't* complete, because the nodes in its last level aren't as far left as possible:



Balanced Binary Tree

A **binary tree** whose nodes all have left and right **subtrees** whose **heights** differ by no more than 1.

A balanced binary tree is such that the logarithmic time complexity of its operations is maintained.

For example, inserting a node at the bottom of the following *imbalanced* binary tree's left subtree would cleary not be a logarithmic-time operation, since it would involve traversing through most of the tree's nodes:

The following is an example of a balanced binary tree:

Full Binary Tree

A **binary tree** whose nodes all have either two child-nodes or zero child-nodes. Example: