Data Structures

A data structure is a specific solution for organizing data that provides storage for items and capabilities for storing and retrieving them.

The data structure itself can hold store data statically or dynamically.

In a **static data structure**, the size of the structure is fixed. The content of the data structure can be modified but not without changing the memory space allocated to it. Such as an **array.**

In a **dynamic data structure**, the size of the structure is not fixed and can be modified during the operations performed on it.

Dynamic data structures are designed to facilitate change of data structures in the run time such as a **linked list**.

Static data structures provide easier access to elements with respect to dynamic data structures. Unlike static data structures, dynamic data structures are flexible.

Note: Not every data structure can be categorized as static or dynamic. A stack or queue can be implemented using an array or a linked list**.** 😕

Abstract Data Types

An (ADT) is the specification of a group of operations that make sense for a given data type. They define an interface (API) for working with variables holding data of a given type—hiding all details of how data is stored and operated in memory.

When our algorithms need to operate on data, we don’t directly instruct the computer’s memory to read and write. We use external data-handling modules that provide procedures defined in ADTs.

For example, to operate with variables that store lists, we need procedures for creating and deleting lists; procedures for accessing or removing the (n)th item; and a procedure for appending a new item to a list.

The definitions of these procedures (their names and what they do) are a List ADT. We can work with lists by exclusively relying on these procedures. That way, we never manipulate the computer’s memory directly.

There are different ways to structure data in memory, leading to different data-handling modules for a same data type. An example would be a stack or queue ADT. We can implement this ADT using an array or a linked list.

This means we can change the way the data is stored and manipulated by just using a different data-handling module.

It’s like cars: electric cars and gas-powered cars all have the same driving interface. Anyone who can drive a car can effortlessly switch to any other.

Data Structures

An ADT only describes how variables of a given data type are operated. It provides a list of operations, but doesn’t explain how data operations happen.

Conversely, how data structures describe data is to be organized and accessed in the computer’s memory. They provide ways for implementing ADTs in data-handling modules.

You can think about an ADT as the blueprint, while a **data structure** is the translation of those specifications into real code.

Common ADT:

- Stack and Queue
- List
- Tree
- Graph

Types of Structures

There are different ways to implement ADTs because there are different data structures. Selecting an ADT implementation that uses the best data structure according to your needs is essential for creating efficient computer programs.

The Hash Table

The Hash Table is a data structure that allows finding items in O(1) time. Searching for an item takes a constant amount of time, whether searching for 10 items or 10 million items.

Similar to an array, a hash table requires pre-allocating a big chunk of sequential memory to store data. But unlike an Array, items are not stored in an ordered sequence.

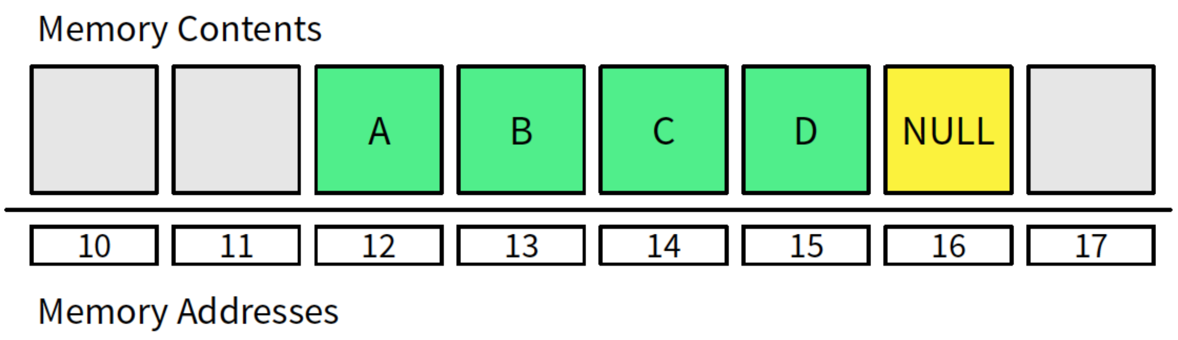
The position an item occupies is “magically” given by a hash function. That’s a special function that takes the data you want to store as input, and outputs a random looking number that is the memory position the item it will be stored at. This allows us to retrieve items instantly.

1. A given value is run through the hash function
2. The function will output the exact position the item should be stored in memory.
3. Fetch that memory position
4. If it was stored, you will find it there.

Note: In JavaScript a hash table is just an object.

The Array

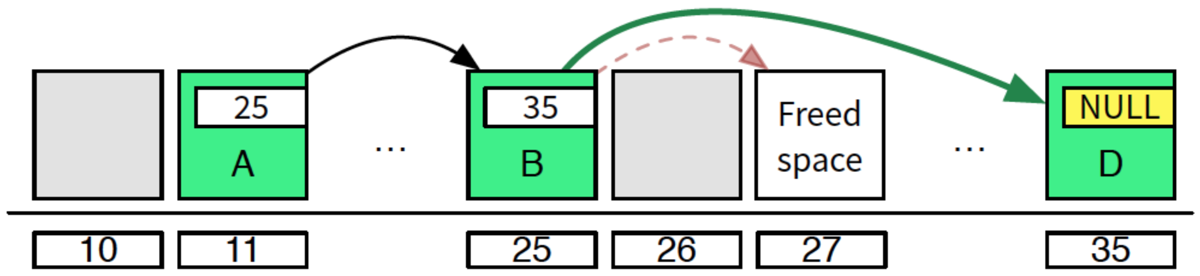
The array is the simplest way to store a bunch of items in computer memory. It consists in allocating a **sequential** space in the computer memory, and writing your items sequentially in that space, marking the end of the sequence with a special NULL token



The Linked List

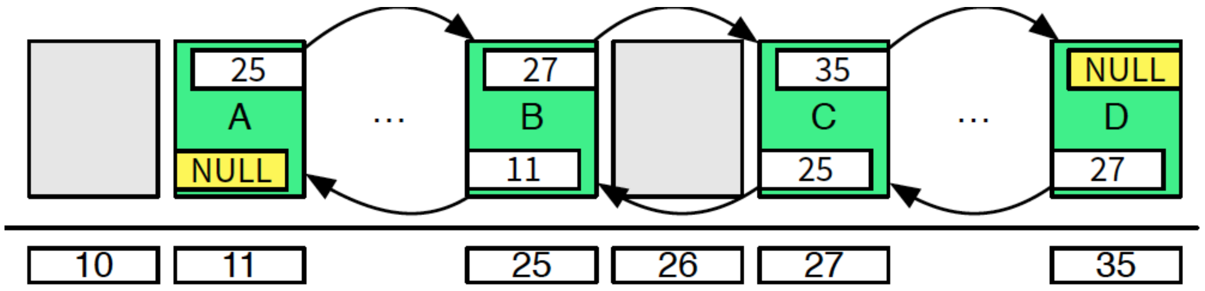
With Linked Lists, items are stored in a chain of cells that don't need to be at sequential memory addresses. Memory for each cell is allocated as needed.

Linked lists can be used to implement Stacks, Lists, and Queues. There's no problem growing the list: each cell can be kept at any part of the memory. It's also easy to insert items in the middle or delete any item by changing the cell pointers.



The Doubly Linked List

The Double Linked List is the Linked List with an extra: cells have two pointers: one to the cell that came before it, and other to the cell that comes after.



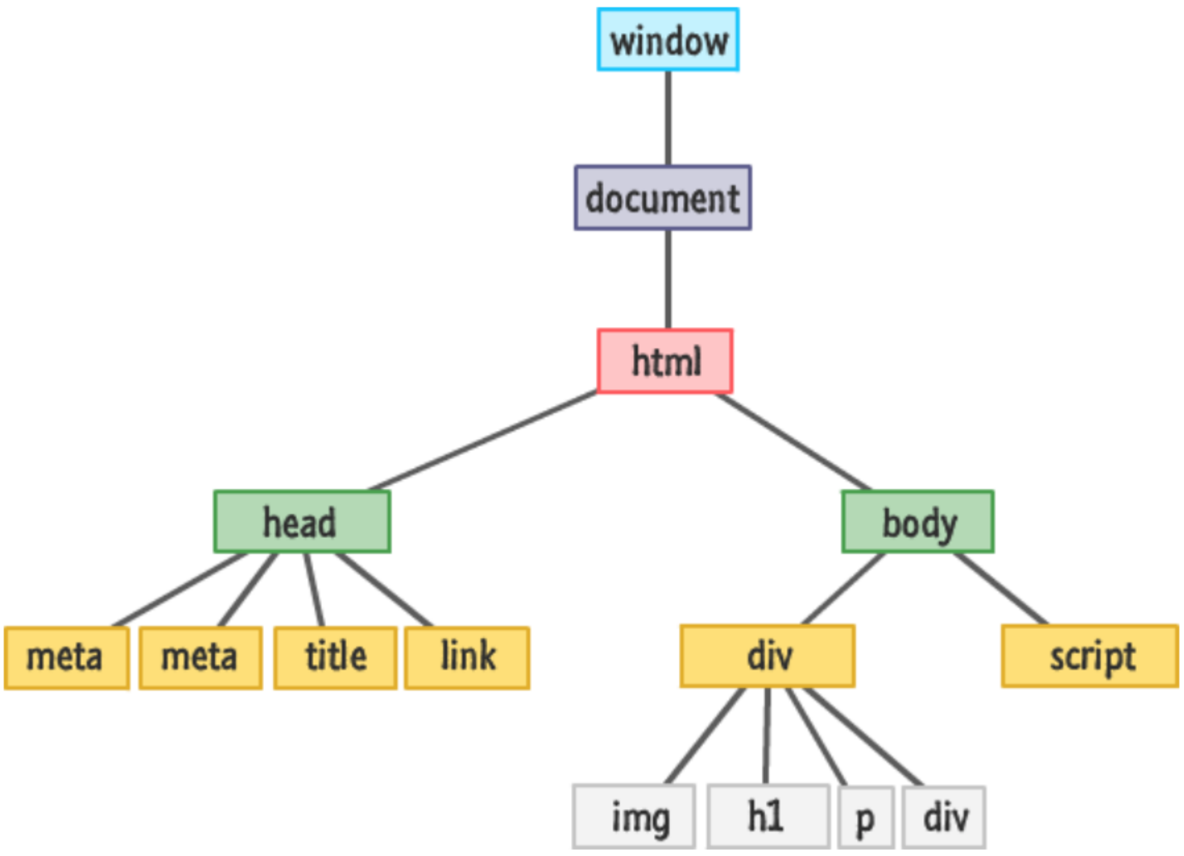
Note: Feature-rich programming languages (not JavaScript) often come with built-in implementations for List, Queue, Stack and other ADTs. These implementations often resort to a default data structure.

The Tree

Like the Linked List, the Tree employs memory cells that do not need to be contiguous in physical memory to store objects. Cells also have pointers to other cells. Unlike Linked

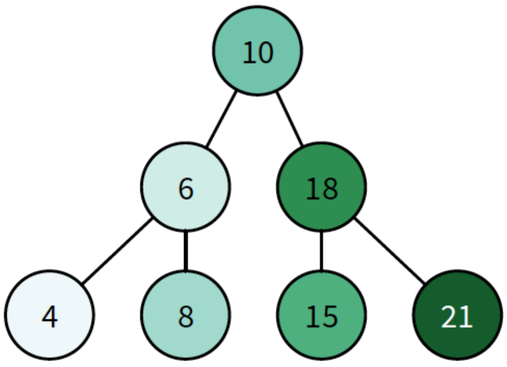
Lists, cells and their pointers are not arranged as a linear chain of cells, but as a tree-like structure.

In the Tree terminology, a cell is called a **node**, and a pointer from one cell to another is called an **edge**. The topmost node of a tree is the **Root Node**: the only node that doesn't have a parent. Apart Root from the Root Node, nodes in trees must have exactly one parent.



The Binary Search Tree

A Binary Search Tree is a special type of Tree that can be efficiently searched. Nodes in Binary Search Trees can have at most two children.



Nodes are positioned according to their value/key. Children nodes to the left of the parent must be **smaller** than the parent, children nodes to the right must be greater.