Data Structures

Array, Linked Lists, Binary Tree, Map

Linked List vs Array

- **Arrays** store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows faster access to an element at a specific index.
- **Linked lists** are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element.

Major differences

- **Size**: Since data can only be stored in contiguous blocks of memory in an array, its size cannot be altered at runtime due to the risk of overwriting other data. However, in a linked list, each node points to the next one such that data can exist at scattered (non-contiguous) addresses; this allows for a dynamic size that can change at runtime.
- **Memory allocation**: For arrays at compile time and at runtime for linked lists. but, a dynamically allocated array also allocates memory at runtime.
- **Memory efficiency**: For the same number of elements, linked lists use more memory as a reference to the next node is also stored along with the data. However, size flexibility in linked lists may make them use less memory overall; this is useful when there is uncertainty about size or there are large variations in the size of data elements; memory equivalent to the upper limit on the size has to be allocated (even if not all of it is being used) while using arrays, whereas linked lists can increase their sizes step-by-step proportionately to the amount of data.

Major differences

- **Execution time**: Any element in an array can be directly accessed with its index; however in the case of a linked list, all the previous elements must be traversed to reach any element. Also, better cache locality in arrays (due to contiguous memory allocation) can significantly improve performance. As a result, some operations (such as modifying a certain element) are faster in arrays, while some others (such as inserting/deleting an element in the data) are faster in linked lists.

Following are the points in favor of Linked Lists.

- The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, the upper limit is rarely reached.
- Inserting a new element in an array of elements is expensive because room has to be created for the new elements and to create room existing elements have to be shifted.

So Linked list provides the following two advantages over arrays

- Dynamic size
- Ease of insertion/deletion

Linked lists have the following drawbacks:

- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.
- Extra memory space for a pointer is required with each element of the list.
- Arrays have better cache locality that can make a pretty big difference in performance.
- It takes a lot of time in traversing and changing the pointers.
- It will be confusing when we work with pointers

Array vs Binary Tree

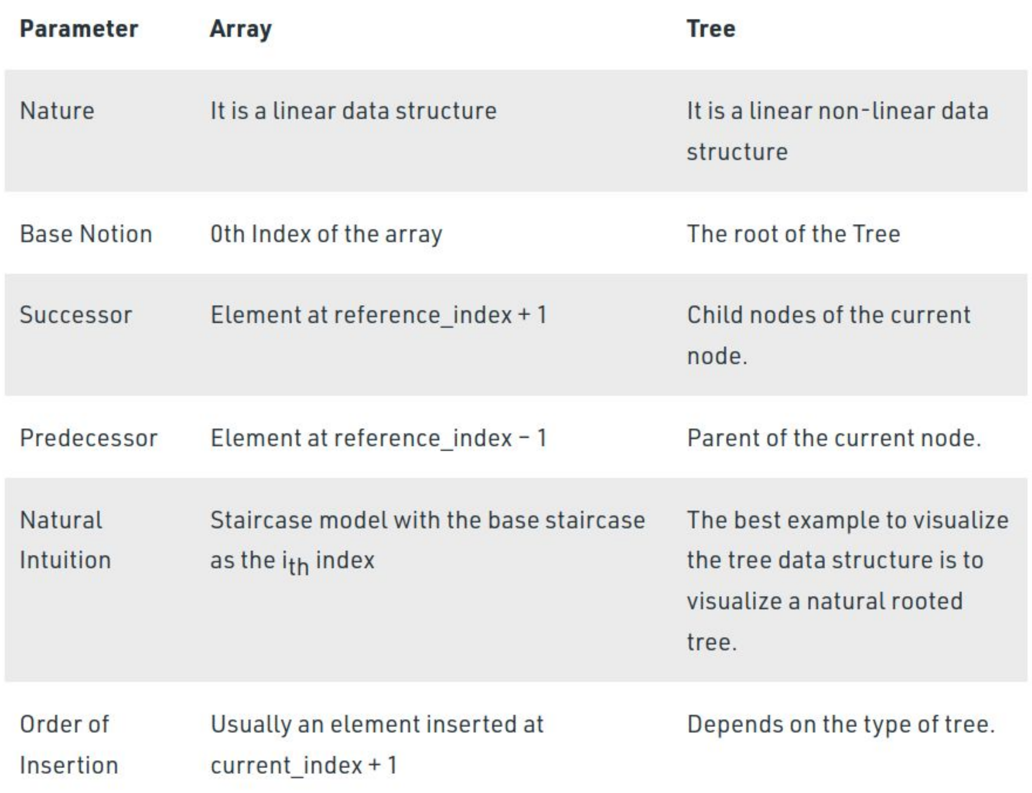
- The array is a linear data structure in which elements are stored at contiguous memory locations
- In array, we store elements of the same datatype together.
- It has index-based addressing as elements are stored at contiguous memory locations.
- Index starts from **0** and goes up to **(N – 1)** where **N** is the number of elements in the array.
- As the arrays allow random access of elements in **O(1)**. It makes accessing elements by position faster.
- The biggest disadvantage of an array is that its size cannot be increased.

Array vs Binary Tree

A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operations are as fast as in a linked list.

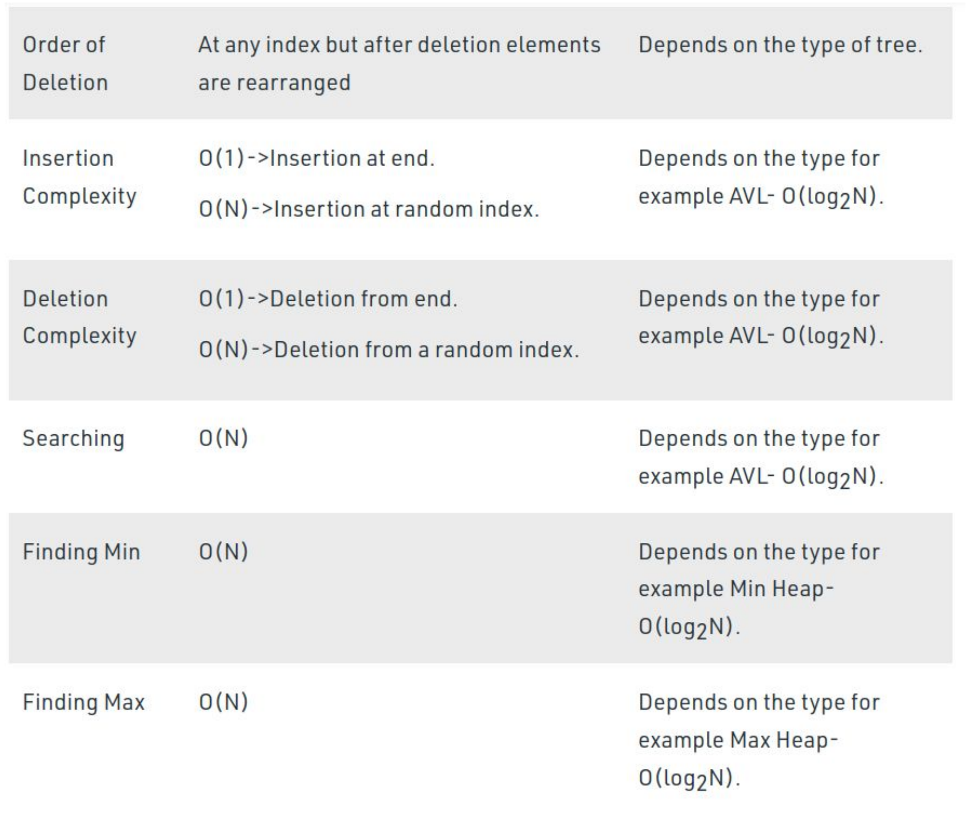
- A tree is a group of nodes starting from the root node.
- Every node has a specific parent and may or may not have multiple child nodes.
- Each node contains a value and references to the children.
- It’s a kind of graph data structure but does not have cycles and is fully connected.

Tabular difference between Array and Tree



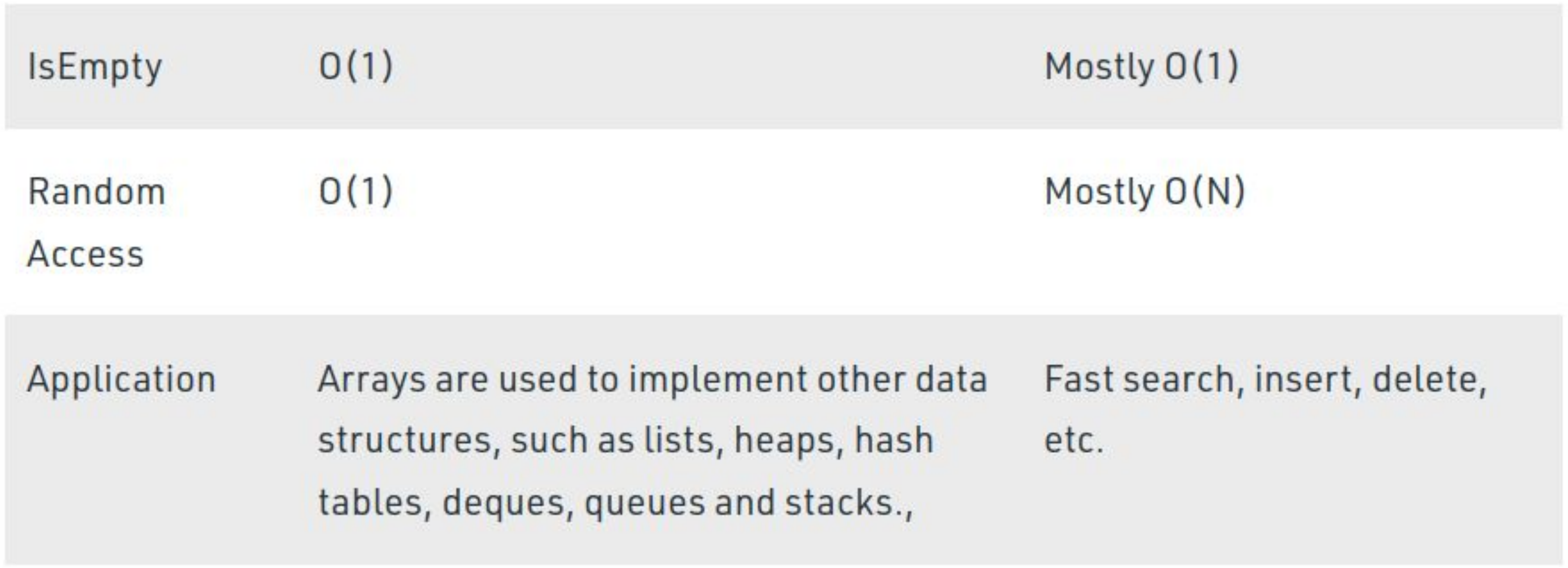
Parameter	Array	Tree
Nature	It is a linear data structure	It is a linear non-linear data structure
Base Notion	0th Index of the array	The root of the Tree
Successor	Element at reference_index + 1	Child nodes of the current node.
Predecessor	Element at reference_index - 1	Parent of the current node.
Natural Intuition	Staircase model with the base staircase as the i_{th} index	The best example to visualize the tree data structure is to visualize a natural rooted tree.
Order of Insertion	Usually an element inserted at current_index + 1	Depends on the type of tree.

Tabular difference between Array and Tree



Order of Deletion	At any index but after deletion elements are rearranged	Depends on the type of tree.
Insertion Complexity	O(1)->Insertion at end. O(N)->Insertion at random index.	Depends on the type for example AVL- O($\log_2 N$).
Deletion Complexity	O(1)->Deletion from end. O(N)->Deletion from a random index.	Depends on the type for example AVL- O($\log_2 N$).
Searching	O(N)	Depends on the type for example AVL- O($\log_2 N$).
Finding Min	O(N)	Depends on the type for example Min Heap- O($\log_2 N$).
Finding Max	O(N)	Depends on the type for example Max Heap- O($\log_2 N$).

Tabular difference between Array and Tree



IsEmpty	O(1)	Mostly O(1)
Random Access	O(1)	Mostly O(N)
Application	Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques, queues and stacks.,	Fast search, insert, delete, etc.

Properties of Sets data structure

Sets are data structures that hold collection of objects without duplicates. Main operation on a sets are:

- **add(element)** adds element to the set. This will not have any effect if element is already in the set
- **remove(element)** removes element of the set. This have no effect if elements is not present in the set.
- **contains(element)** returns **True** if element is in the set **False** otherwise
- **__iter__(element)** allows to iterate over elements of the set (in not specified order)

Main benefit of sets is the fact that time complexity of **add**, **remove**, **contains** are all constant time operations.

Map Data Structure

MAP is an abstract data structure (ADT)

- It stores key-value (k, v) pairs
- There cannot be duplicated keys

Maps are useful in situations where a key can be viewed as a unique identifier for the object

- the key is used to decide where to store the object in the structure. In other words, the key associated with an object can be viewed as the address for the object
- maps are sometimes called associative arrays

Note: Maps provide an alternative approach to searching

Map Data Structure

MAP ADT:

- size()
- isEmpty()
- get(k): -> this can be viewed as searching for key **k** (if M contains an entry with key k, return it, else return null)
- put(k, v): -> this can be viewed as inserting key **k**
  - If **M** does not have an entry with key **k,** add entry **(k, v)** and return **null**
  - else replace existing value of entry with **v** and return the old value
- remove(k): -> this can be viewed as deleting key **k** (remove entry (k, *) from M)

Map Example

(k,v) key=integer, value=letter

	M={}
• put(5,A)	M={(5,A)}
• put(7,B)	M={(5,A), (7,B)}
• put(2,C)	M={(5,A), (7,B), (2,C)}
• put(8,D)	M={(5,A), (7,B), (2,C), (8,D)}
• put(2,E)	M={(5,A), (7,B), (2,E), (8,D)}
• get(7)	return B
• get(4)	return null
• get(2)	return E
• remove(5)	M={(7,B), (2,E), (8,D)}
• remove(2)	M={(7,B), (8,D)}
• get(2)	return null

Map Implementations

- Arrays (Vector, ArrayList)
- Linked-List
- Binary search trees
- Hash tables