Data Structures Notes

By Agnes Wangechi

Hello Stranger! *These are notes made during the my #100daysofCode Challenge. Start date: 21st November 2022 (in-progress)*
The notes were written using Notion app and exported into a pdf.

*If you can't **explain it simply**, you don't **understand it well** enough.*

~ Albert Einstein

1. *Introduction*

A `data structure (DS)` is a way of organizing data so that it can be used efficiently.

Data structures are helpful in:

1. Creating fast and powerful algorithms.
2. Managing and organizing data.
3. Making code cleaner and easier to understand.

2. *Abstract Data Type vs Data Structure*

Abstract data types are `analogous to modes of transportation` e.g. walking, train, biking, plane. `They are just ways of getting from one point to another.` When you choose a ***`specific mode of transportation like biking then that is considered a data structure.`***

An `abstract data type (ADT)` is referred to as an abstraction of a data structure which provides only the interface to which a data structure must adhere to. The interface does not give specific details about how something should be implemented or in what programming language.

Abstraction (ADT)	Implementation (DS)
List	Dynamic Array, Linked List
Queue	Linked list based queue, Array based queue, Stack based queue,
Map	Tree Map, Hash Map / Hash Table
Vehicle	Golf cart, Smart car, Bicycle

---

3. *Computational Complexity Analysis*

This helps us understand the performance that the data structures are providing. i.e knowing the time and space an algorithm will need.

An algorithm that takes so much time to run even if it uses very little space is not effective. The same is when an algorithm uses all the bytes but runs fast is still not efficient.

This is how Big-O Notation was created.

3.1. *Big-O Notation (Come back to this)*

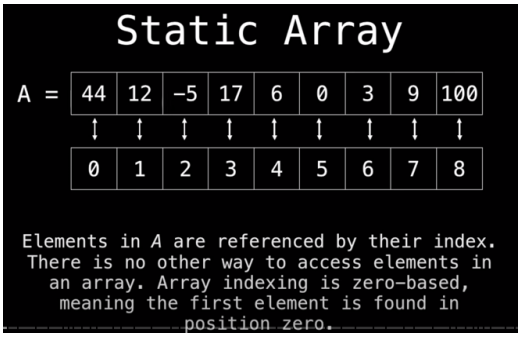
Big-O Notation gives an upper bound of the complexity in the worst case, helping to quantify performance as input size becomes arbitrary large. `In other words we are always looking for the worst case scenario of an algorithm when using Big-O in terms of time and space.`

Lets say you are looking for a number 8 in an unordered list of unique numbers, in terms of time, Big-O Notation tells us the time it will take to find 8, in the worst case; when 8 is the last in the list.

4. *Static and Dynamic Arrays*

4.1. Static Arrays

Static Arrays are arrays where the `size/length is specified when the array is created/allocated.` Because the length is fixed, this type of array is also know as fixed-length arrays / fixed arrays.



4.1.2. Application of Static arrays

1. Temporary store objects
2. Lookup tables and inverse lookup tables due to its indexing.
3. Can be used to return multiple values from a table
4. Storing and accessing sequential data
5. Used by Input/Output routines as buffers

4.1.3. How static Arrays are implemented in different languages

Language	Defined Values	Fixed-Length with Undefined or Default Values
C++	`int values[] = {0, 1, 2};`	`int values[3];`
C#	`int[] values = {0, 1, 2};`	`int[] values = = new int[3];`
Java	`int[] values = {0, 1, 2};`	`int[] values = = new int[3];`
JavaScript	`var values = [0, 1, 2];`	`var values = new Array(3);`
Python	`values = [0, 1, 2]`	`values = [None] * 3`
Swift	`var values:[Int] = [0, 1, 2]`	`var values: [Int] = [Int](repeating: 0, count: 3)`

4.1.4. Complexity of Static Arrays

	Static Arrays	*Explanation*
Accessing	O(1)	It is constant because of indexing. You can find a particular value in an array.
Searching	O(n)	It takes linear time since you have to look through all the values in the array.
Insertion	N/A	Due to its fixed nature, no new values can be added to the array
Appending	N/A	Due to its fixed

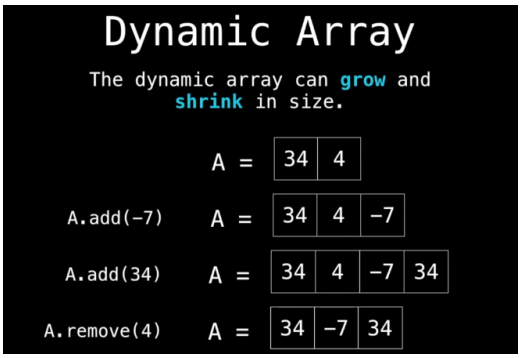
6. Used in dynamic programming to cache answers to sub-problems

4.2. Dynamic Arrays

This is an array that allows individual elements to be added and / or removed during runtime.

4.2.1. Operation in Dynamic Arrays

This example below is in the Java language.



A dynamic array can be implemented using a static array. The following steps explains how this takes place.

1. Create a static array with an initial capacity
2. Add elements to the static array

		nature, no new values can be appended to the array
Deletion	N/A	Due to its fixed nature, no values can be deleted from the array

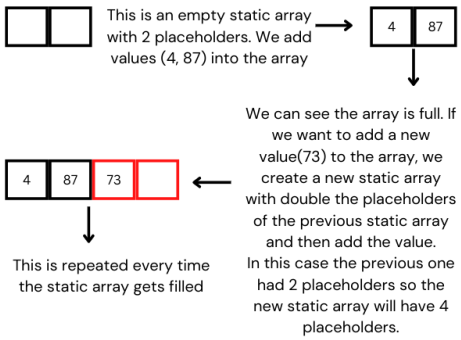
4.2.2. How Dynamic Arrays are implemented in different languages

Language	Class	Add	Remove
C++	`#include <list>` `std::list`	`insert`	`erase`
C#	`System.Collections.Generic.List`	`Add`	`Remove`
Java	`java.util.ArrayList`	`add`	`remove`
JavaScript	`Array`	`push` , `splice`	`pop` , `splice`
Python	`List`	`append`	`remove`
Swift	`Array`	`append`	`remove`

4.2.3. Complexity of Dynamic Arrays

	Dynamic Arrays	*Explanation*
Accessing	O(1)	It is constant because of indexing. You can find a particular value in an array.
Searching	O(n)	It takes linear time since you have to look through all the values in the array.
Insertion	O(n)	It is linear because you have to recopy and shift all the elements to

3. If a new element is to be added but the static array is full, a new static array with double the capacity is created and copy the original elements in.



		the right when adding to an array
Appending	O(1)	It takes constant time because it does not take time to add a value at the end of an array.
Deletion	O(n)	It is linear because you have to recopy and shift all the elements when deleting from an array.

5. *Linked Lists*

5.1. Introduction

A linked list is a sequential list of nodes that hold data that point to other nodes containing data.

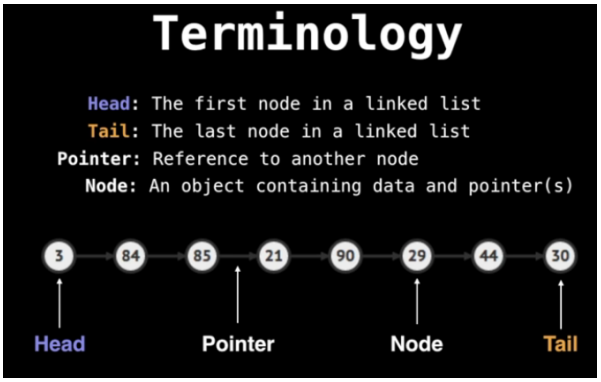
It should be noted that the last node always has a null reference show the end of the list.

Pointers are also known as references.

Nodes are implemented as structs and classes

5.1.1 Applications of Linked Lists

1. Used in Stack, List, Queue implementation due to their great time complexity during adding and removing elements.
2. Making of circular lists; where the last node pointer points to the first node.An example is representing corners of a polygon.
3. Can be used to model real world objects such as trains.
4. Used in separate chaining which is present in certain hash table implementations to deal with hashing collisions.



5. Used in implementation of adjacent lists for graphs.

5.2. Singly linked lists vs Doubly linked lists

Singly linked lists only hold a reference to the next node while doubly linked lists hold a reference to both the next node and the previous node. In implementation of both, a reference is always maintained to the head and tail of the lists for easier additions and removals from both ends of the lists.

Singly linked lists

Pros	**Cons**
Uses less memory	Cannot easily access previous elements
Easier to implement	

Doubly linked list

5.2.1 Complexity of Linked Lists

	Singly linked list	*Explanation*
Search	O(n)	It is linear because we have to go through all the elements.
Insert at head	O(1)	It takes constant time because there is always a pointer at the head of a list.
Insert at tail	O(1)	It takes constant time because there is always a pointer at the tail of a list.
Remove at head	O(1)	It takes constant time because there is always a pointer at the head of a list.
Remove at tail	O(n)	Even is we have a reference to the tail, we can only remove it once and we wont know the new tail. So to know the new

Pros	Cons
Can be traversed backwards; can access the list from the tail.	Takes 2x the memory
Because there is access of all the nodes, adding/removing a node is done in constant time.	

		tail, we need to to go through all the elements till the end hence takes linear time.
Remove at middle of list	O(n)	It is linear because we have to go through all the elements until the middle is reached.

	Doubly linked list	*Explanation*
Search	O(n)	It is linear because we have to go through all the elements.
Insert at head	O(1)	It takes constant time because there is always a pointer at the head of a list.
Insert at tail	O(1)	It takes constant time because there is always a pointer at the tail of a list.
Remove at head	O(1)	It takes constant time because there is always a pointer at the head of a list.
Remove at tail	O(1)	There is a always a reference to the previous node of the tail node hence the new tail is easily known.
Remove at middle of list	O(n)	It is linear because we have to go through all the elements until the middle is reached.

5.3. Implementation and Deletion of Linked lists (Create Visualizations on Canva)

This explains how both the singly and doubly linked lists are implemented.

1. Singly linked list
2. Doubly linked list

---

6. *Stacks*