Data Structures and Algorithms

Iñaki Lakunza

March 19, 2024

Contents

1 Introduction

What is a Data Structure? A data structure (DS) is a way of organizing data so that it can be used effectively. It is just a way of organizing data, so that it can then be accessed and updated easily.

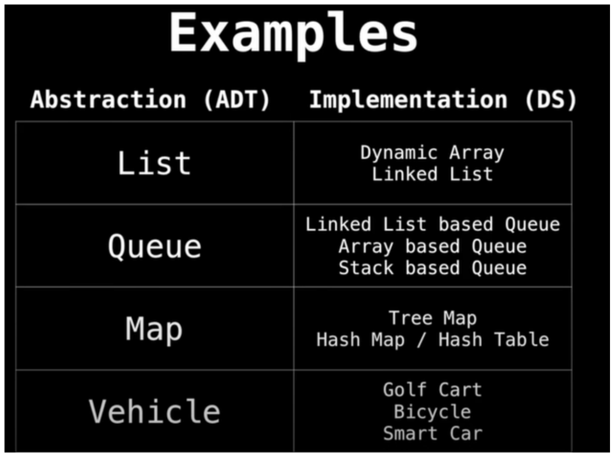
Why Data Structures? They are essential ingredients in creating fast and powerful algorithms. They help to manage and organize data in a very natural way. And, they make the code cleaner and easier to understand. Data Structures can make a difference between having an okay product and an outstanding one.

1.1 Abstract Data Type

An **Abstract data type (ADT)** is an abstraction of a data structure which PROVIDES ONLY THE INTERFACE TO WHICH A DATA STRUCTURE MUST ADHERE TO. The interface does not give any specific details about how something should be implemented or in what programming language.

As an example, we suppose that our abstract data type is for a mode of transportation, to get from pint A to point B. There are different modes of transportation, like walking, or going by train. These specific modes of transportation would be analogous to the data structures themselves. We want to get from one place to another, that is our abstract data type. And, how did we do that? That is our data structure.

These are some data type examples:



Examples

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 Bicycle Smart Car

Figure 1:

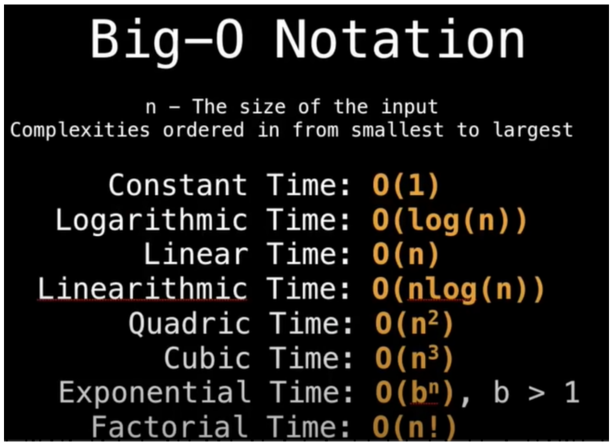
As we can see, lists can be implemented in two ways, we can have a dynamic array, or a linked list. And the same for the rest, we can implement any abstraction in very different ways. **Abstract data types only defines how a data structure should behave, and what methods it should have, but not the details surrounding how those methods are implemented**.

1.2 Computational Complexity

We must take into account the **time** and the **space** needed by our algorithm.

Big-O Notation gives an upper bound of the complexity in the **worst** case, helping to quantify performance as the input size becomes **arbitrarily large**.

We care when our input becomes large, so because of that reason we will be ignoring constants, for example.



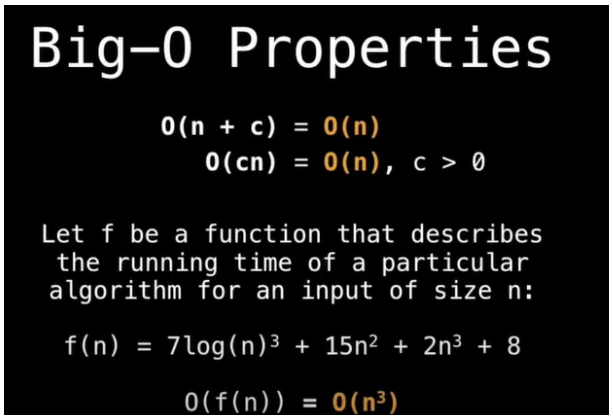
Big-O Notation

n - The size of the input
Complexities ordered in from smallest to largest

Constant Time: $O(1)$
Logarithmic Time: $O(\log(n))$
Linear Time: $O(n)$
Linearithmic Time: $O(n\log(n))$
Quadric Time: $O(n^2)$
Cubic Time: $O(n^3)$
Exponential Time: $O(b^n)$, $b > 1$
Factorial Time: $O(n!)$

Figure 2:

n will usually be the size of the input coming in to our algorithm.



Big-O Properties

$O(n + c) = O(n)$
$O(cn) = O(n)$, $c > 0$

Let f be a function that describes the running time of a particular algorithm for an input of size n:

$f(n) = 7\log(n)^3 + 15n^2 + 2n^3 + 8$

$O(f(n)) = O(n^3)$

Figure 3:

O really cares when our notation becomes very big, that is why remove constant values, when adding or when multiplying. (But when the constant is very very large in practice we should consider it).

For example the two following blocks run in constant time, because they do not depend on the input size n:

```
a := 1
b := 1
c := a + 5*b
```

```
i := 0
While i<11 Do
    i = i + 1
```

In the second case, we are doing a loop, but the loop does not depend on the input size, so it needs a constant time.

On the other hand, the following run in **linear** time: $O(n)$

```
i := 0
While i<n Do
    i = i+1
# f(n) = n
# O(f(n)) = O(n)


i := 0
While i<n Do
    i = i+3

# f(n) = n/3
# O(f(n)) = O(n)
```

In the second case, we are incrementing by 3, so we will end up ending the loop 3 times faster, so we will end up doing $n/3$ iterations, but, since we do not care about constants, the time complexity is n.

Next, both of the following examples run in **quadratic** time. The first one may be obvious since n work done n times is $n * n = O(n^2)$.

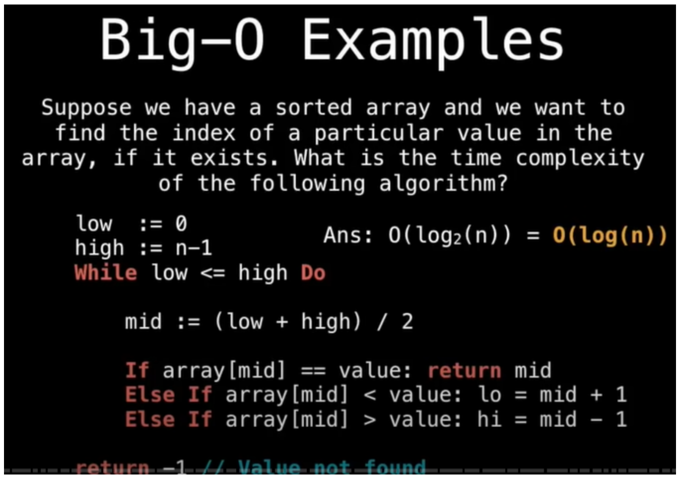
```
For (i := 0; i<n; i = i+1)
    For (j := 0; j<n; j = j+1)

# f(n) = n*n = n^2 , O(f(n)) = O(n^2)



For (i := 0; i<n; i = i+1)
    For (j := 0; j<n; j = j+1)
    # We have replaced in the second For the 0 with i
```

The first block is obvious, but, in the second case, focusing just in the second loop, since i goes from $[0, n)$, the amount of looping is directly determined by what i is. Remark that if $i = 0$, we do n work, if $i = 1$, we do $n - 1$ work, if $i = 2$, we do $n - 2$ work, etc... So, we end up having the following: $(n) + (n - 1) + (n - 2) + (n - 3) + ... + 3 + 2 + 1$ This turns out to be $n(n + 1)/2$, which can be considered as n^2, because we do not care about constants. So, $O(n^2)$.

Now, here is a more complex example, where we do a search in a binary tree:

Here is another example:

```
i := 0
While i<n Do
```



Big-O Examples

Suppose we have a sorted array and we want to find the index of a particular value in the array, if it exists. What is the time complexity of the following algorithm?

```
low  := 0
high := n-1
While low <= high Do

    mid := (low + high) / 2

    If array[mid] == value: return mid
    Else If array[mid] < value: lo = mid + 1
    Else If array[mid] > value: hi = mid - 1

return -1 // Value not found
```

Ans: $O(\log_2(n)) = O(\log(n))$

Figure 4:

```
    j=0
    While j<3*n Do
        j = j+1
    j=0
    While j<2*n Do
        j = j+1
    i = i+1

# f(n) = n*(3n + 2n) = 5n^2
# O(f(n)) = O(n^2)
```

We have two inner loops, so, we add the time of the loops that are the same, and multiply different level loops.

Another example:

```
i := 0
While i<3*n Do
    j := 10
    While j<=50 Do
        j = j+1
    j=0
    While j< n*n*n Do
        j = j+2
    i = i+1

# f(n) = 3n + (40 + n^3 /2) = 3n/40 + 3n^4 /2
# O(f(n)) = O(n^4)
```

We have i going from 0 to $3 * n$ in the outside. So we have to multiply that with what it is going on in the inside. Inside, j goes from 10 to 50, so that does 40 loops exactly every loop. So that is a constant 40 amount of loop. In the second loop, j is less than n^3, but $j = j + 2$ so it is accelerated, so in the inside we are going to get $(40 + n^3/2)$, and we have to multiply that by $3n$.

2 Static and Dynamic Arrays

2.1 Introduction

Arrays are probably the most used Data Structure, probably because it forms the fundamental building block of all data structures. With arrays and pointers, we could be able to construct any data structure.

Static Arrays

A static array is a **fixed length** container containing n elements **indexable** from the range $[0, n-1]$.

By being indexable, we mean that each slot or index in the array can be referenced with a number.

On the other hand, static arrays are given **contiguous chunks of memory**, meaning that our chunk of memory will not have holes and gaps, it will be contiguous, all addresses will be adjacent in our static array.

When and where is a static array used?

- Storing and accessing sequential data
- Temporarily storing objects
- Used by IO routines as buffers
- Lookup tables and inverse lookup tables. This way be can retrieve the data easily from a table of information.
- Can be used to return multiple values from a function. This is useful in programming languages where just a single return value is allowed in functions.
- Used in dynamic programming to cache answers to subproblems.

Complexity The access time for static and dynamic arrays is constant because of a property that arrays are indexable.

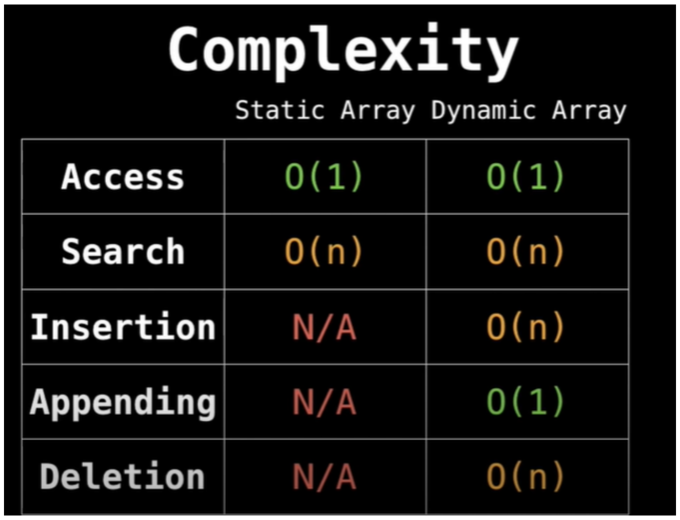
On the other hand, searching takes a linear time, since we have to transverse all the elements in the array, and, if the element we are looking for does not exist (worst case), we will have to analyze all elements in the array.

Inserting, appending and deleting in a static array does not make sense. This is because the static array is a fixed size container, it cannot grow larger or smaller.

But inserting in a dynamic array will cost a linear time, because we will potentially have to shift all the elements in the array to the right, and recopy all the elements into the new static array (**this is assuming we are implementing dynamic arrays using static arrays**).

Appending though is constant because when we append elements we just have to resize the internal static array containing all those elements. But this happens so rarely that appending becomes constant time.

Deletions are linear for the same reason that insertions are linear, because in the worst case we will have to shift all the elements in the array and potentially recopy the whole static array.



Complexity

	Static Array	Dynamic Array
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	N/A	O(n)
Appending	N/A	O(1)
Deletion	N/A	O(n)

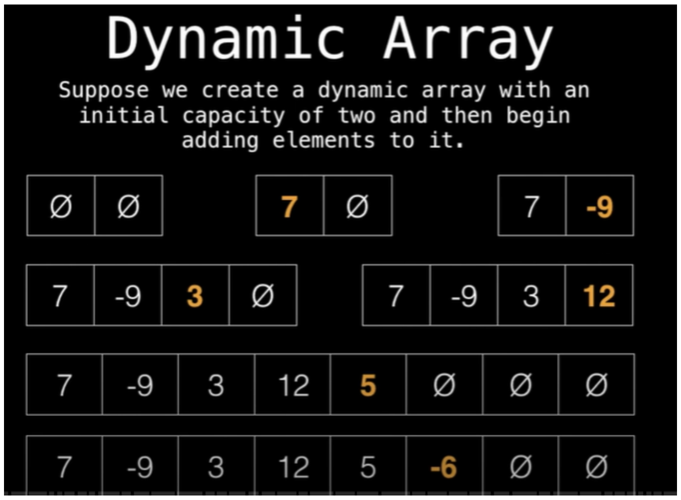
Figure 5:

2.2 Operations in Dynamic Arrays

As we know, dynamic arrays can grow and shrink in size as needed. So the dynamic arrays can do all similar **get** and **set** operations that static arrays can do, but, unlike the static array, it grows inside as dinamically as needed.

How can we implement a dynamic array?
One ways is to **use a static array** (this is not the only way):

1. Create a satic array with an initial capacity.
2. Add elements to the underlying static array, keeping track of the number of elements.
3. If adding another element exceeds the internal capacity of our static array, **create a new static array with twice the capacity and copy the original elements to it**.



Dynamic Array

Suppose we create a dynamic array with an initial capacity of two and then begin adding elements to it.

Figure 6:

3 Singly and Doubly Linked Lists

3.1 Introduction

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

Is is important to notice that **every node has a pointer to another node**.

And also notice that the last pointer points to null, meaning that there are no more nodes at this point. The last node always has a null reference.

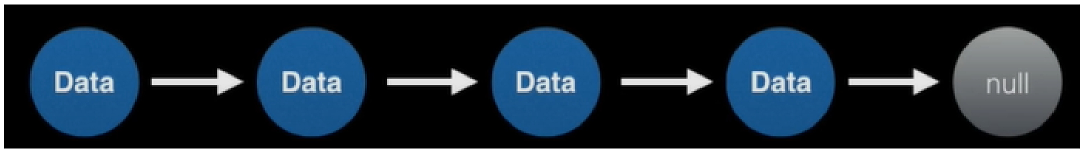
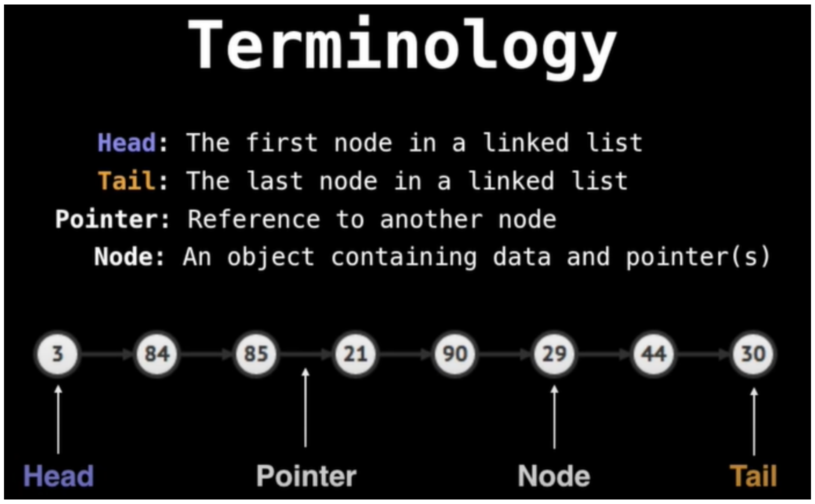


Figure 7:

Where are linked lists used?

- Used in many List, Queue & Stack implementations, because of their great time complexity for adding and removing elements.
- Great for creating circular lists, making the pointer in the last node point to the first node. Used to model repeating event cycles
- Can easily model real world objects such as trains.
- Used in separate chaining, which is present in certain Hash-table implementations to deal with hashing collisions.
- Often used in the implementation of adjacency lists for graphs..

Terminology



Terminology

Head: The first node in a linked list
Tail: The last node in a linked list
Pointer: Reference to another node
Node: An object containing data and pointer(s)

Figure 8:

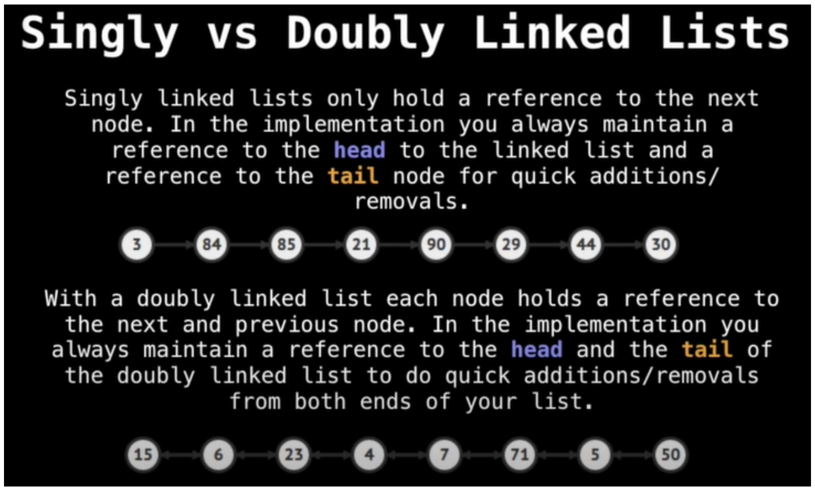
It is very important to **always maintain a reference to the head of the link lists**. This is because we need somewhere to start when transversing our list.

We also give a name to the last element of the linked list. This is called the **tail of the list**.

Then we have the nodes themselves, which contain pointers (pointers are also sometimes called references), and these pointers always point to the next node.

The nodes themselves are usually represented as structs, or classes, when implemented.

3.2 Singly vs Doubly Linked List



Singly vs Doubly Linked Lists

Singly linked lists only hold a reference to the next node. In the implementation you always maintain a reference to the head to the linked list and a reference to the tail node for quick additions/removals.

With a doubly linked list each node holds a reference to the next and previous node. In the implementation you always maintain a reference to the head and the tail of the doubly linked list to do quick additions/removals from both ends of your list.

Figure 9:

Singly linked lists only contain a pointer to the next node. While doubly linked lists also contain a pointer to the previous node, which can be quite useful sometimes. (We could also have tiply or quadruply linked lists).

The pros of using a **singly linked list** are the lower memory rquirement and the simpler implementation. But its cons are that the previous element cannot be accesed. On the other hand, **doubly linked lists** can be transversed backwards, but the bad part is that the memory requirement is twice larger than with singly linked lists.

3.3 Implementation details

3.3.1 Inserting in Singly Linked Lists

We want to insert 11 in the third position, where 7 currently is. **Always the first thing to do is to create a pointer which points to the head**.

Now, we will seek up to but not including the node we want to remove. So, we will seek ahead advancing our transversal pointer, moving it to the 23, as in the top right image.

And now, we are already where we need to insert the next node. So we will create the next node, the green node, and we will make 11's next pointer point to 7, meaning that the next node from 11 will be 7. And, the next step is to change 23's next node, its next pointer will be 11. **We can do this because we have access to the node 23 because we have a reference to it with our transversal pointer**. We will get what it is on the bottom left image.

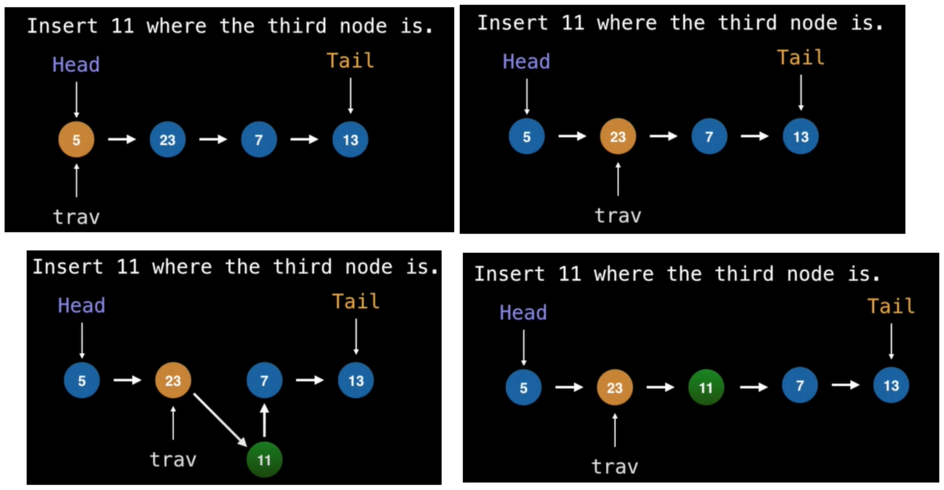


Figure 10:

And then we will be done, we will have the structure of the image on the bottom right (the bottom left and bottom right structures are the same).

3.3.2 Inserting in Doubly Linked List

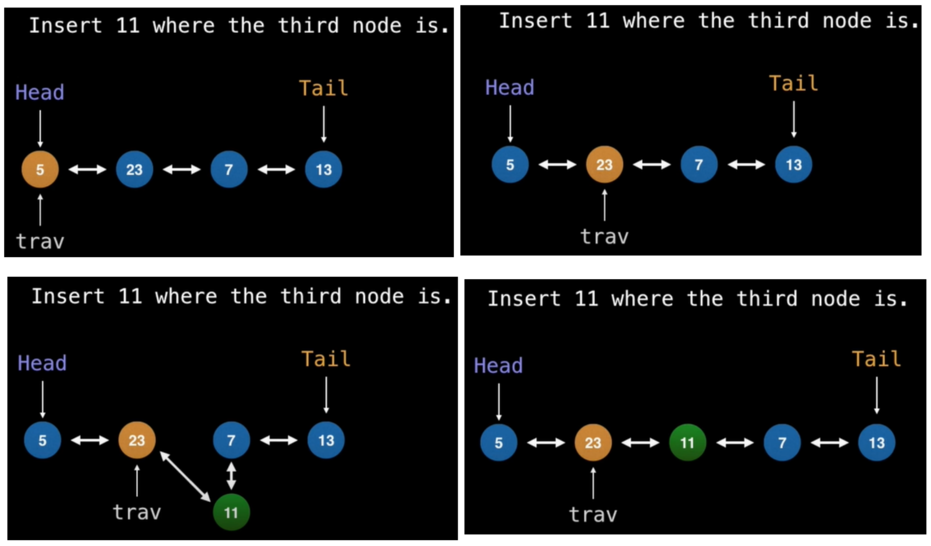


Figure 11:

This process is trickier than the previous, since now we have two pointers in each node, but the concept is exactly the same.

As always, we first make a pointer to our head, and create a transversal pointer, which will point to the head at first and then we will be advancing, **until we are just before to the insertion position**.

As said, we will move forward our transversal node until we reach the previous node where we want to make the insertion, as in the top right image.

We will then create the new node, which is 11, and point 11's next pointer to point to 7. Now, we will also point 11's previous pointer to 23, which we can do, since our transversal node is currently pointing to 23.

And now, we will make 7's previous pointer point to 11. And, the last step is to point 23's next pointer point to 11, and so, we will be getting the structure of the images on the bottom.

3.3.3 Removing in Singly Linked List

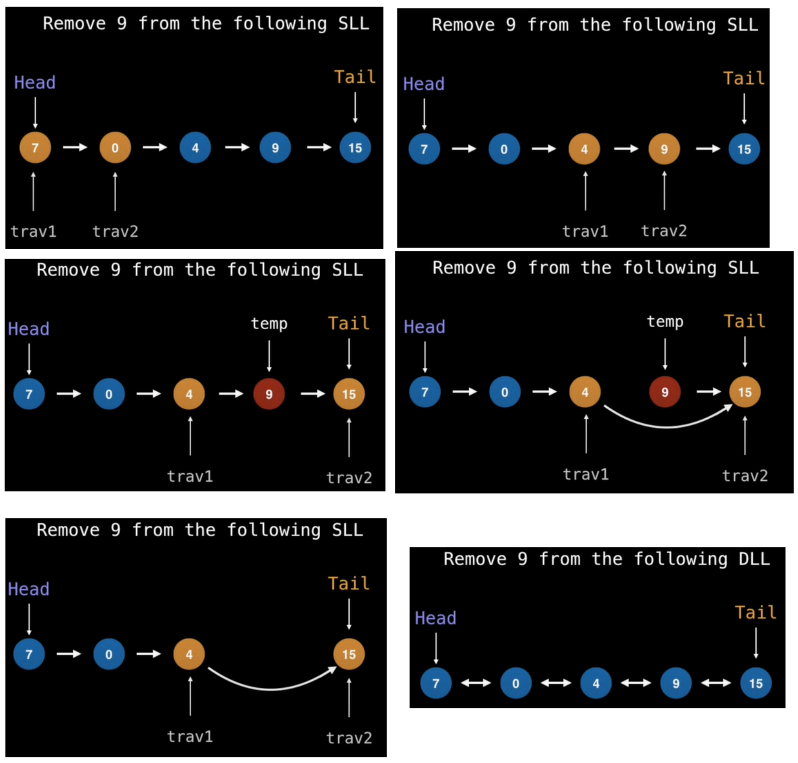

Remove 9 from the following SLL
Head
Tail
7
0
4
9
15
trav1
trav2
temp
Remove 9 from the following DLL

Figure 12:

Now, we want to remove the node with the value 9 from the singly linked list. The trick we will use will be the use of **two pointers**.

First, we will point our head pointer to the head node, as always. And then we will create two transversal pointers, the 1st one pointing to the head and the second one pointing to the head's next node, as in the top right image. Now, we will advance trav2 until we find the node we want to remove, also advancing trav1, so we will get the top right situation.

Now, we will create a temporal pointer to the node we wish to remove, so we can deallocate its memory later, and then we will advance trav2 to the next node. And so we will get the situation of the middle left, at this point, node 9 is ready to be removed.

Now, we will set trav1's next pointer equal to trav2. And now we will are able to remove our temporary pointer.

And so, now the temp has been deallocated. **It is important to make sure we clean up our memory, to avoid memory leaks, specially in languages like C and C++.**

This way we are in the situation of the bottom images.

3.3.4 Removing in Doubly Linked List

The idea is the same as earlier: we seek up to the node we wish to remove, but this time we only need one transversal pointer, because each node has a reference to the previous node.

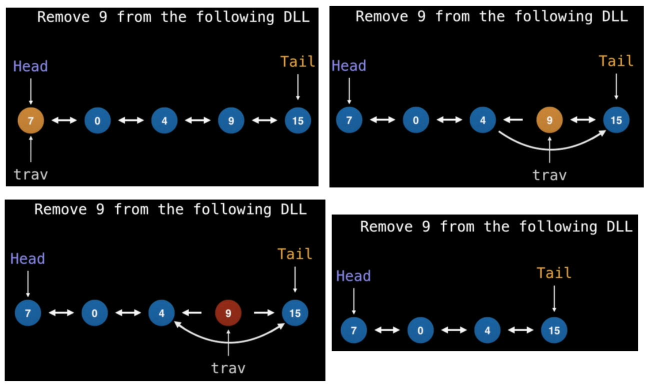
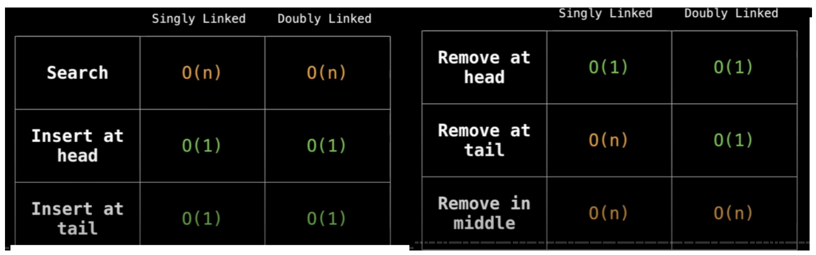


Figure 13:

So, we will start our transversal node from the head, and seek up the node we wish to remove. Now, we will set 4's next pointer equal to 15. We can do this because since we are in node 9, we have access to nodes 4 and 15, we can do ***trav.previous.next = 15***. And, similarly, we will point 15 to 4. Now, the node 9 is ready to be removed. And so, we will be able to remove it.

3.4 Complexity Analysis



	Singly Linked	Doubly Linked
Search	O(n)	O(n)
Insert at head	O(1)	O(1)
Insert at tail	O(1)	O(1)

	Singly Linked	Doubly Linked
Remove at head	O(1)	O(1)
Remove at tail	O(n)	O(1)
Remove in middle	O(n)	O(n)

Figure 14:

We see that in both cases, the search is linear because the element we are looking for is not in the linked list, we will have to transverse the whole list.

On the other hand, we see how in both cases the insertion on the head and on the tail requires constant time, because we always maintain a pointer to the head and to the tail.

To remove the head is also constant time in both cases, since we have a reference to it.

However, removing from the tail is different. It takes a linear time to remove from doubly linked lists, and constant for doubly linked lists. The thing is that we do have a reference to the tail in a singly linked list, **we can remove it, but only once, because we can't reset the value of what the tail is, so, we had to seek to the end of the linked list and find out what the new tail is equal to**. Doubly linked lists do not have this problem, since they have a pointer to the previous node, so, when removing the last node, we can easily move the pointer to the tail to the previous node.

Removing somewhere in the middle is also linear time, since in the worst case we would need to seek through n-1 elements, which is linear.

4 Stack

4.1 Introduction

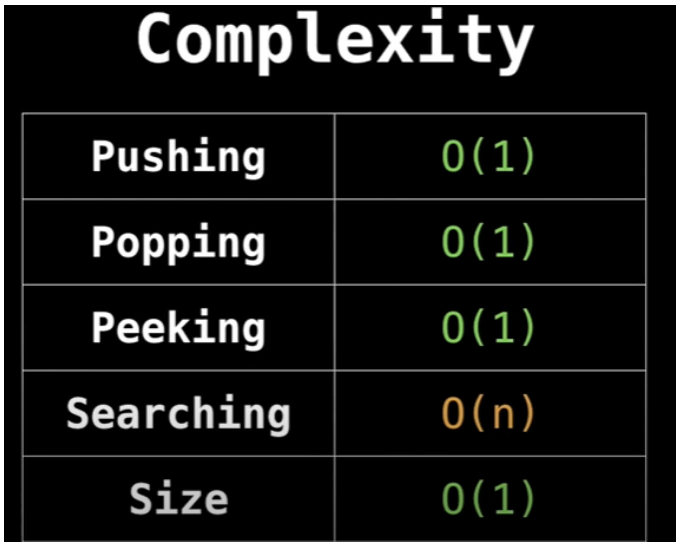
A stack is a one-ended linear data structure which models a real world stack by having two primary operations, **push** and **pop**.

There is always a pointer pointing to the top block of a stack. Block can always be added or taken out at the top of the stack. This behaviour is commonly known as **LIFO: Last In First Out**.

When and where is a Stack used?

- Use by undo mechanisms in text editors.
- Used in compiler syntax checking for matching brackets and braces.
- Can be used to model a pile of books or plates.
- Used behind the scenes to support recursion by keeping track of previous function calls.
- Can be used to do a **Depth First Search (DFS)** on a graph.

4.2 Complexity analysis



Complexity

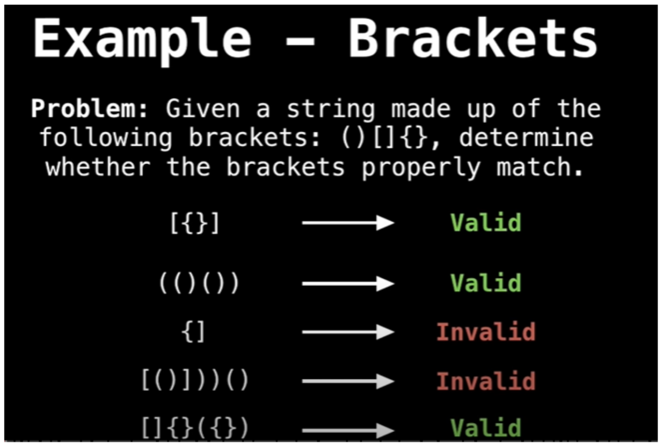
Pushing	O(1)
Popping	O(1)
Peeking	O(1)
Searching	O(n)
Size	O(1)

Figure 15:

(The table assumes we have implemented a stack using a Linked List!)

Pushing takes a constant time since we always have a reference at the top of the stack, and the same happens for pooping and peeking.

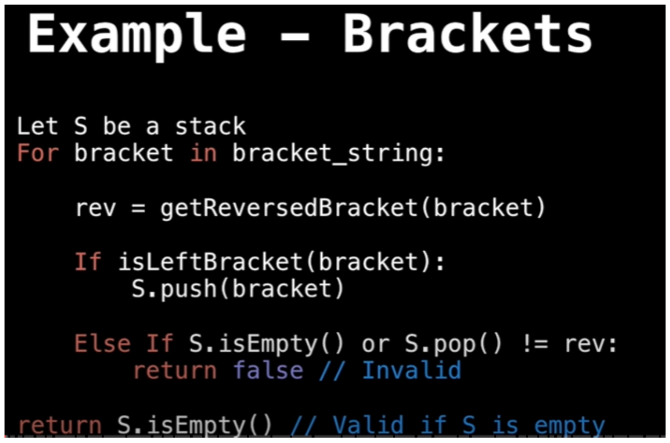
On the other hand, searching takes a linear time, since the element we are searching for is not necessarily at the top of the stack, so we might end up scanning all the elements in the stack.



Example – Brackets

Problem: Given a string made up of the following brackets: ()[]{}, determine whether the brackets properly match.

```
[{}]        ──→  Valid
(()())      ──→  Valid
{]          ──→  Invalid
[()]))()    ──→  Invalid
[]{}({})    ──→  Valid
```



Example – Brackets

```
Let S be a stack
For bracket in bracket_string:

    rev = getReversedBracket(bracket)

    If isLeftBracket(bracket):
        S.push(bracket)

    Else If S.isEmpty() or S.pop() != rev:
        return false // Invalid

return S.isEmpty() // Valid if S is empty
```

Figure 16:

4.3 Example: Brackets

We can use the stack to solve the stated problem. For every left bracket we can simply push those on the stack.

Once we encounter a right bracket we have to do two checks: We first need to look if the stack is empty, and, if it is not, we have to pop the top element of the stack, and see if the popped element is equal to the reversed current element.

And, once we have finished analyzing all the elements, we have to check if the stack is empty. This must be done in order to check that we have not left any elements in there.

The code implementation can be seen in the image at the right.

4.4 Stack implementation

Stacks are often implemented as either arrays, singly linked lists or even sometimes doubly linked lists. We will now see how to push nodes onto a stack with a singly linked list.

We need somewhere to start with to begin, so we will point the head pointer to a null node, this means that the stack is initially empty.

Then, the trick **for creating a stack with singly linked lists is to push the new elements BEFORE THE HEAD and not at the tail of the list**. This way we have pointers pointing in the correct direction when we need to pop elements off the stack.

When we want to pop elements, we have to move the pointer to the next element and **deallocate the last node**.

5 Queues

5.1 Introduction

A queue is a linear data structure which models real world queues by having two primary operations: **enqueue** and **dequeue**.

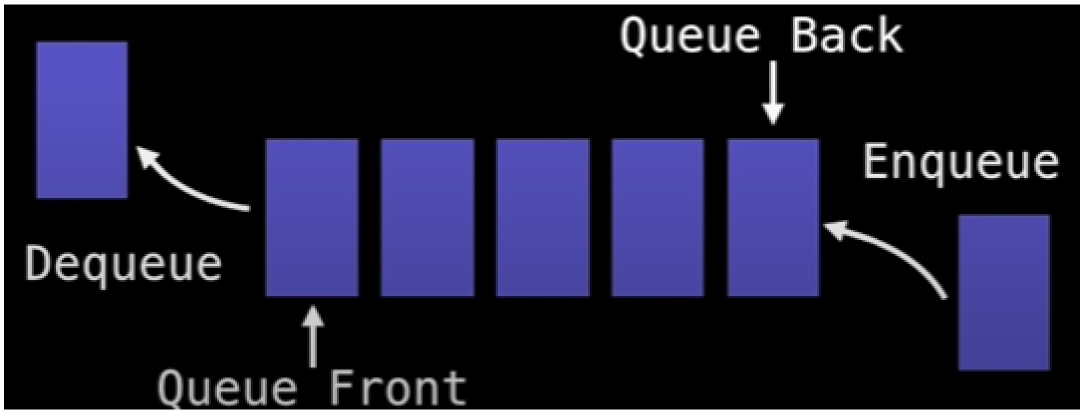


Figure 17:

All queues have a front end and a back end. We insert elements through the back of the queue, this is known as **enqueuing**, and we remove elements from the front of the queue, which is known as **dequeuing**.

Enqueuing can also be named as **adding**, or **offering**, and dequeuing can also be known as **polling** or **removing** (removing might be ambiguous, since we don't specify from where we are removing, front or back).

When and where is a Queue used?

- Any waiting line models a queue, for example a lineup at a movie theatre.
- Can be used to efficiently keep track of the x most recently added elements.
- Web server request management where you want first come first serve
- Bread first search (BFS) graph traversal.

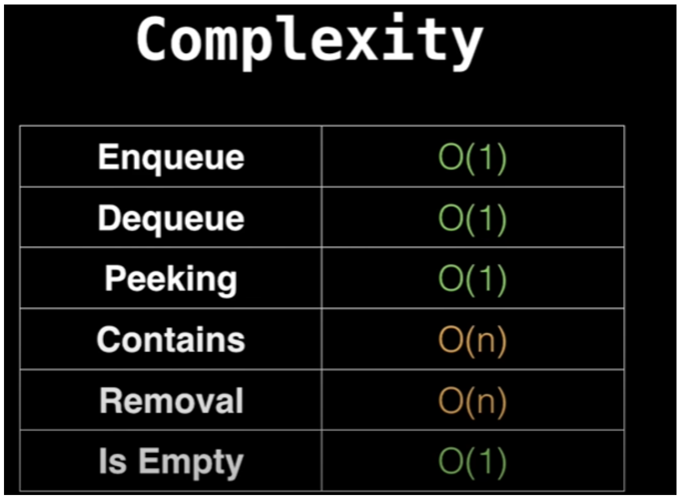
5.2 Complexity analysis

It is quite straightforward to see how enqueuing and dequeuing operations require constant time.

Peeking means to look at the value in the front of the queue, which also requires a constant time. However, looking if an element is withing the queue requires linear time, since we would potentially need to scan through all of the elements.

On the other hand, we have element removal, but not in the sense of dequeuing, but in removing and element in the middle of the queue. This also requires a linear time, since we have to transverse the queue.

And, finally, checking if the queue is empty only requires a constant time, since we always keep track of the queue front and the queue back.



Complexity

Enqueue	O(1)
Dequeue	O(1)
Peeking	O(1)
Contains	O(n)
Removal	O(n)
Is Empty	O(1)

Figure 18:

5.3 Queue example: Breadth First Search (BFS)

A Breadth First Search is an operation we can do on a graph to do a graph transversal. First by visiting all the neighbors of the starting node, and then visiting all the neighbors of the first node we visited, and then all the neighbors of the node we second visited and so on. And so expanding through all the neighbors as we go.

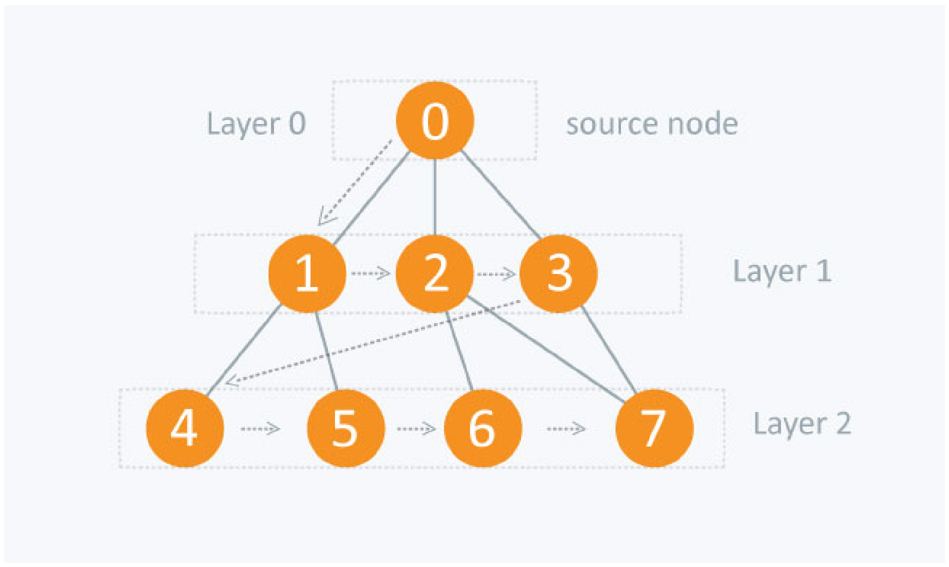
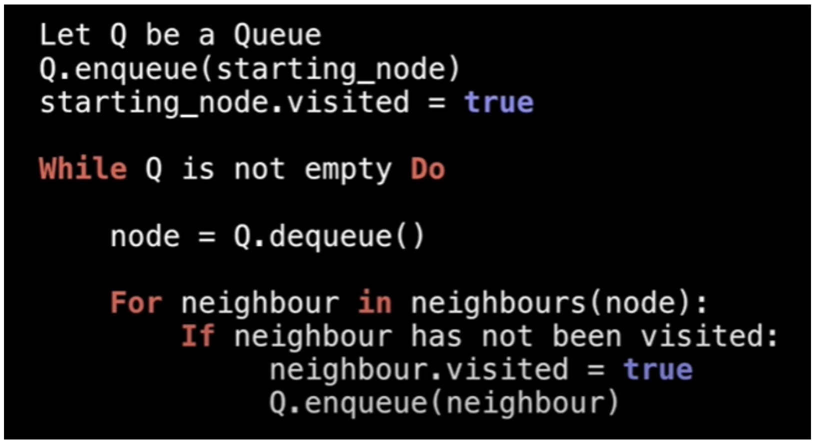


Figure 19:

The idea when using the BFS algorithm is the following, using a queue:

First we add the starting node to our queue and we mark it as visited. And while our queue is not empty, we pull an element from our queue (dequeue), and then, for every neighbor of this node, we just dequeued, if the neighbor has not been visited yet, we add it to the queue. So now we have a way of processing all the nodes in our graph in a breadth first search order



```
Let Q be a Queue
Q.enqueue(starting_node)
starting_node.visited = true

While Q is not empty Do

    node = Q.dequeue()

    For neighbour in neighbours(node):
        If neighbour has not been visited:
             neighbour.visited = true
             Q.enqueue(neighbour)
```

Figure 20: