Data Structures and Algorithms

Iñaki Lakunza

March 28, 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:

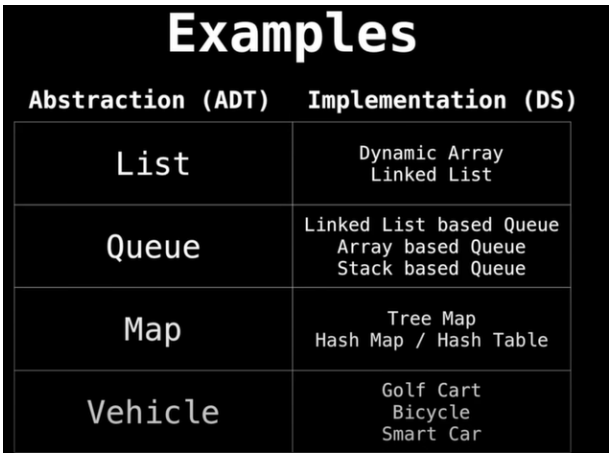


Figure 1:

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.

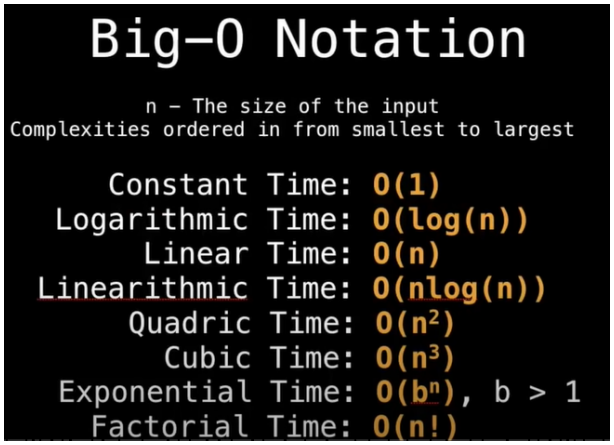


Figure 2:

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

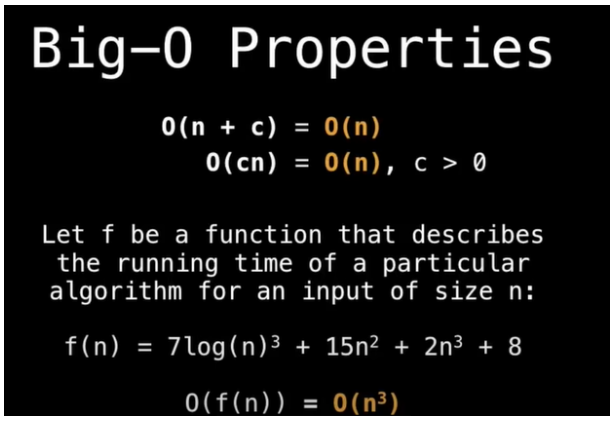


Figure 3:

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
```

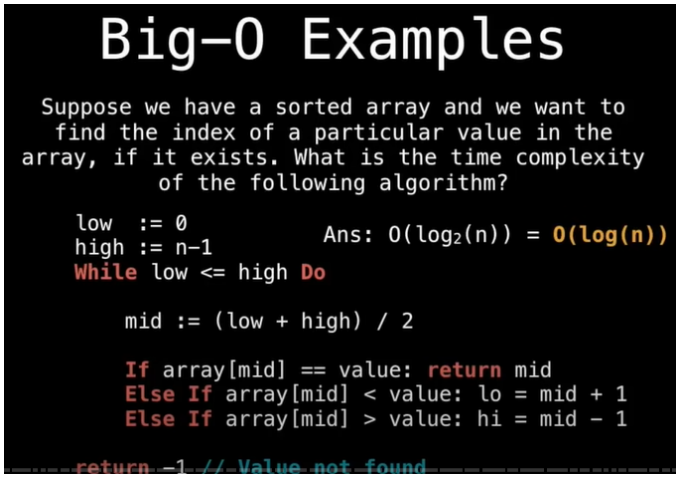


Figure 4:

```
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.

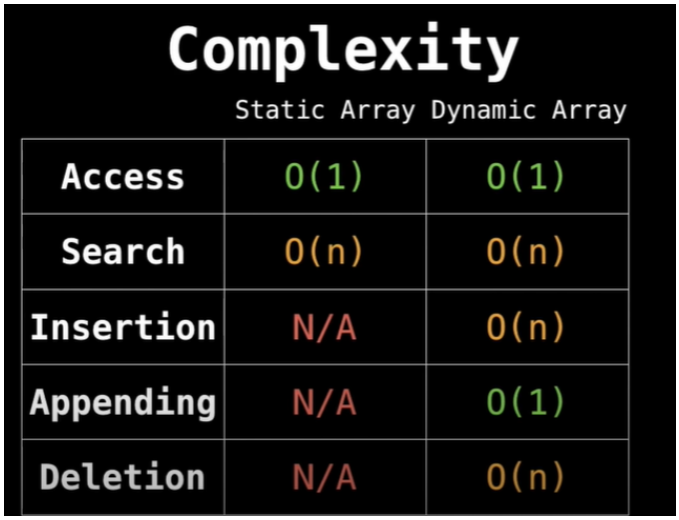


Figure 5:

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**.

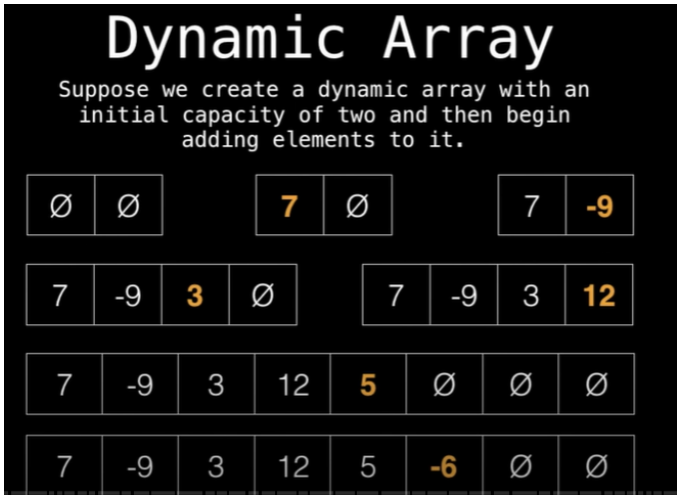


Figure 6:

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

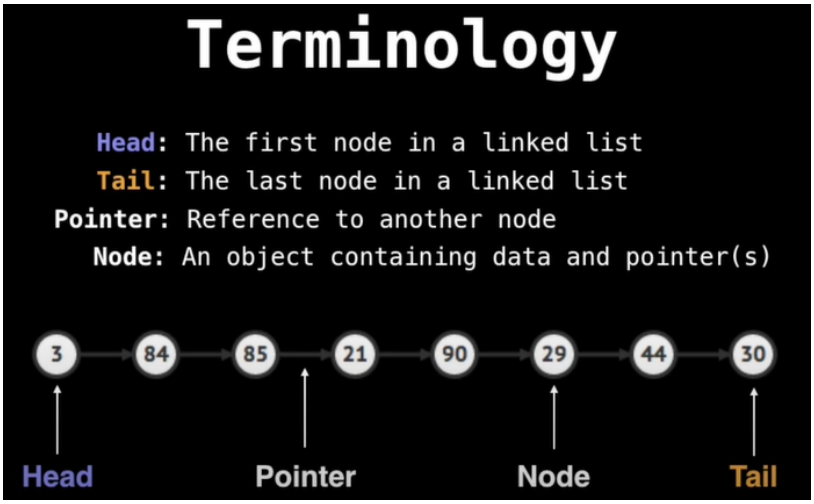


Figure 8:

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

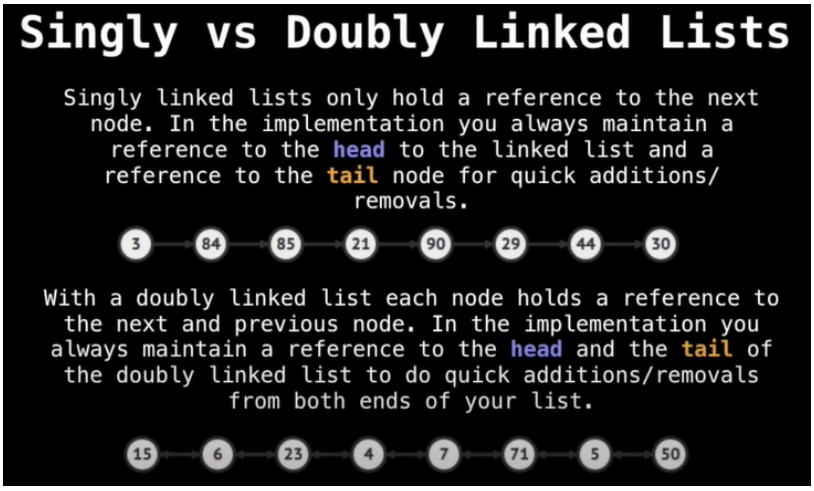


Figure 9:

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.

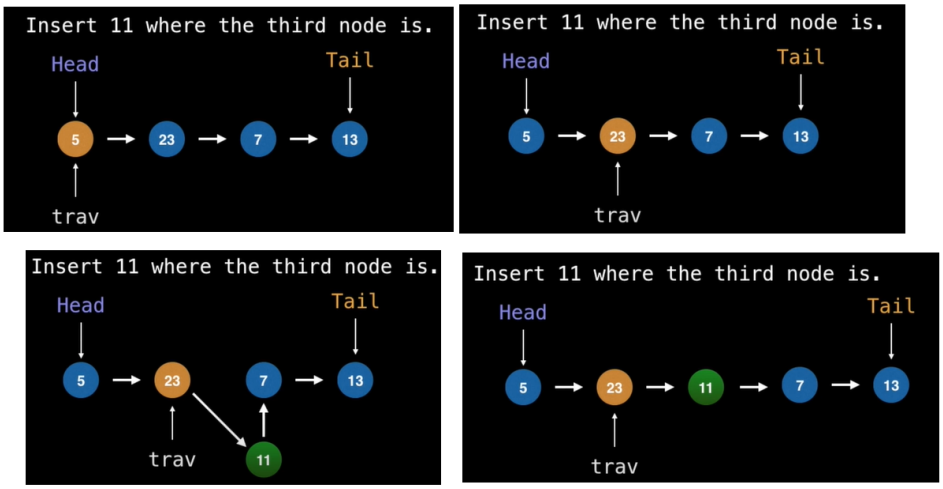


Figure 10:

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

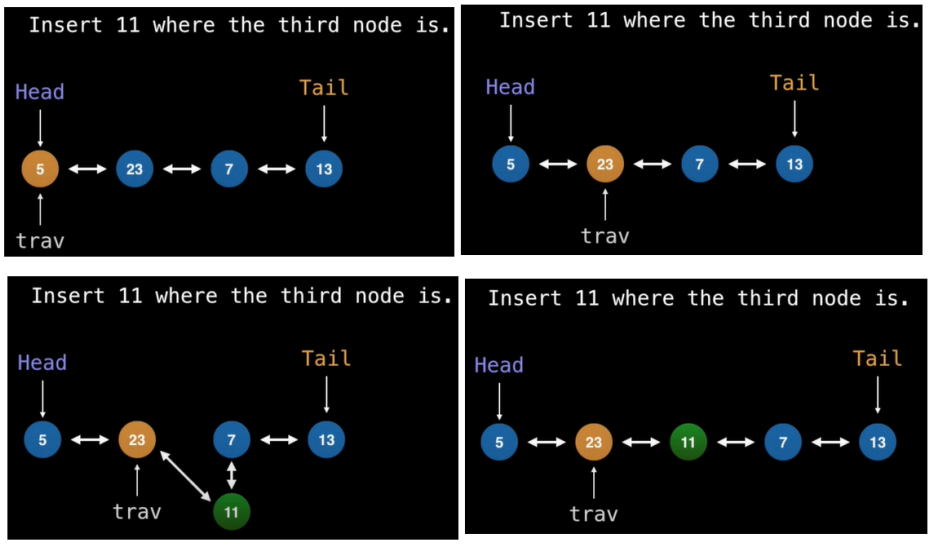


Figure 11:

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

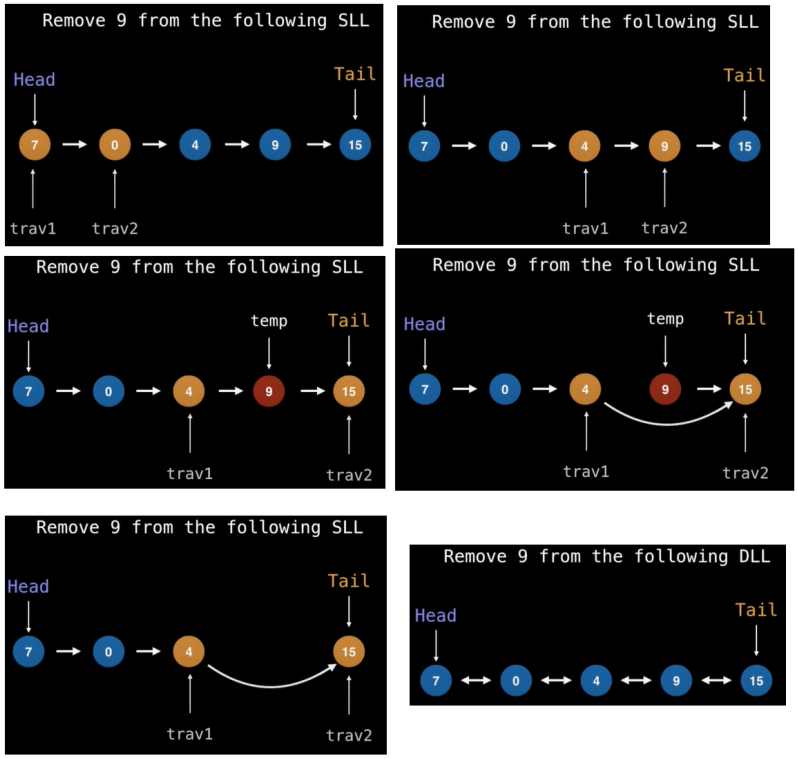


Figure 12:

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.

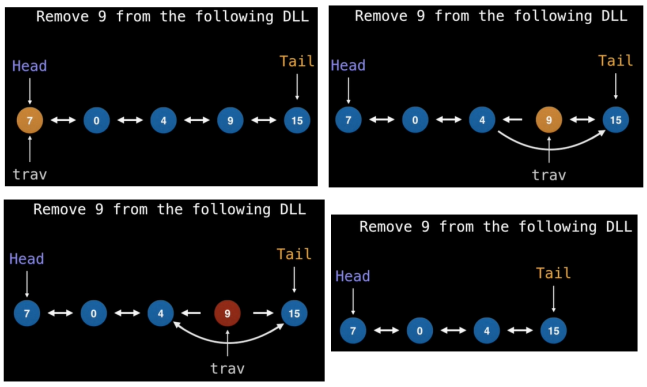


Figure 13:

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

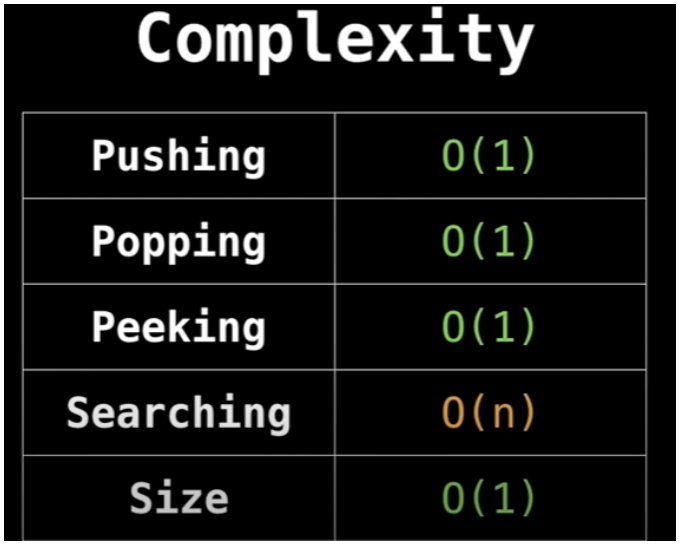
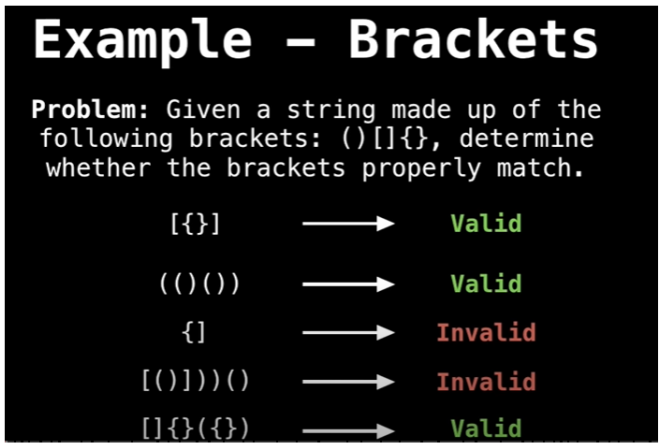


Figure 15:

(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.



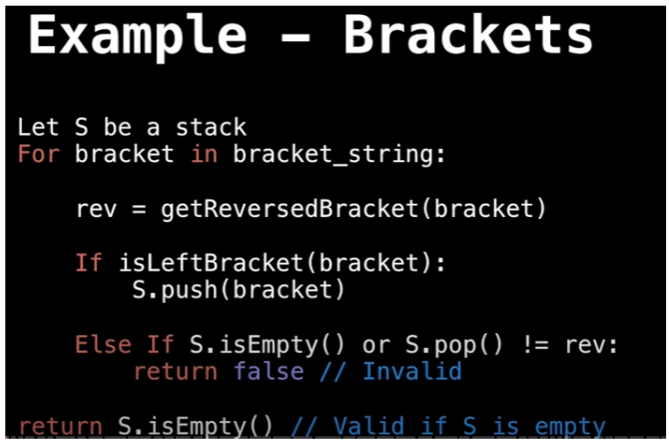


Figure 16:

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**.

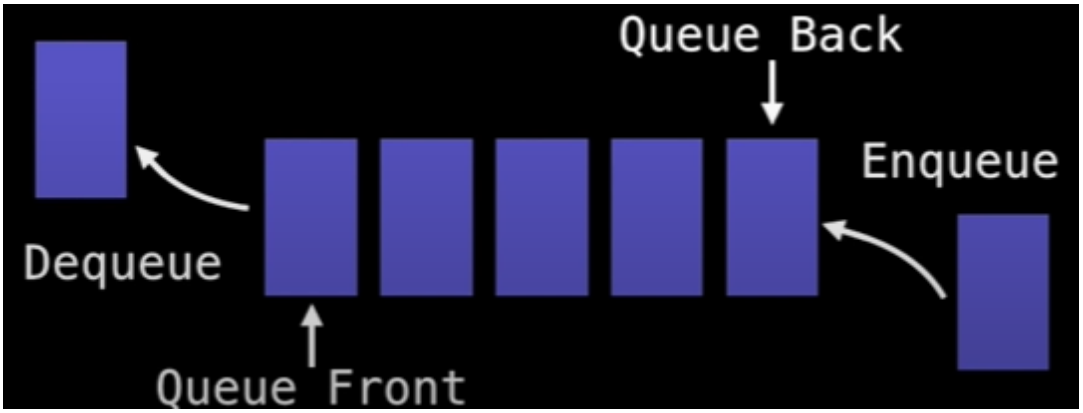


Figure 17:

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.

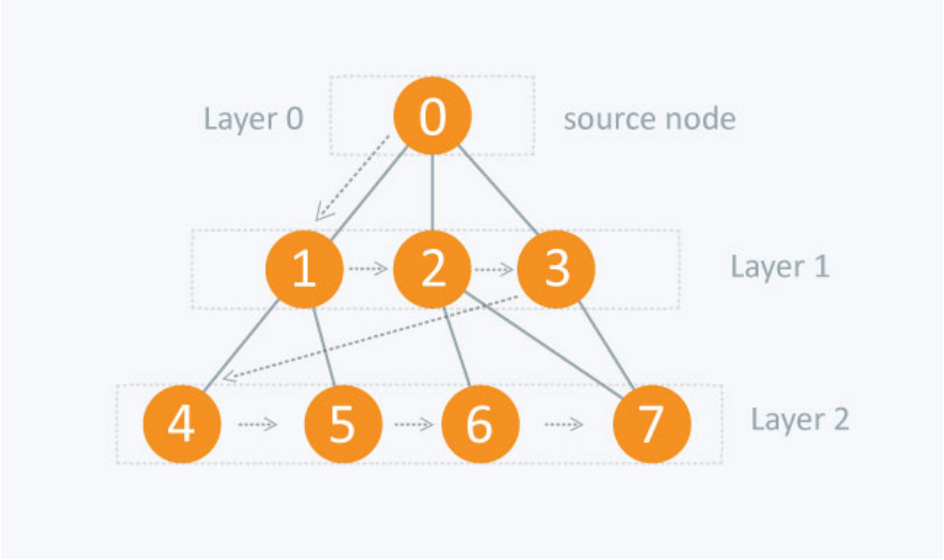


Figure 19:

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:

5.4 Queue Implementation Details

The most popular ways of implementing a queue are either using arrays, singly linked lists, or doubly linked lists. If we are using a static array, we have to make sure it is big enough.

In the case of a **Singly Linked list**, we are going to have a head pointer and a tail pointer. Initially they are both null, but, as we enqueue, we push the TAIL pointer forward. So we are adding a node and then getting the tail pointer to point to the next node.

Dequeue is a bit of the opposite: instead of pushing the tail forward we will be pushing the HEAD forward. We will move the head pointer to the next pointer, and then the element that was left over was the one we want to dequeue and return to the user. So, when we push the head pointer forward, after we have handled it, we have to set it to null, to be able to deallocate it from the memory to avoid memory leaks.

And, at the end, if we remove the whole queue, we will end up in the same position as in the beginning: the Head and the Tail will be pointing to Null.

6 Priority Queues

6.1 Introduction

A priority Queue is an Abstract Data Type (ADT) that operates similar to a normal queue, except that **each element has a certain priority**. The priority of the elements in the priority queue determine the order in which elements are removed from the PQ.

NOTE: Priority queues only support **comparable data**, meaning the data inserted into the priority queue must be able to be ordered in some way either from least to greatest or greatest to least. This is so that we are able to assign relative priorities to each element.

We will have two main operations: **pool**, which takes out the element which has the highest priority. And **add**, which will add an element to our queue, and the order that it will be assigned will depend on its priority, to its entry time.

The priority queue needs to know which is the next element that will be necessary to be removed. As humans, we can have some idea of how to do this depending on the needed case, but we have to put this idea in the computer. For doing this, we will use what is it known as **heap**.

6.2 Heap

What is a Heap?

A heap is a **tree** based Data Structure that satisfies the **heap invariant** (also known as **heap property**): If A is a parent node of B, then A is ordered with respect to B for all nodes A, B in the heap.

What this means is that the value of the parent node is always greater that or equal to the value of the child node for all nodes. Or the other way around: the value of the parent node is less than or equal to the value of the child node for all nodes.

This means we end up getting two types of heaps: **Max Heaps** and **Min Heaps**. Max heaps are the one where the parent node is always greater than its children and the min heap is the opposite.

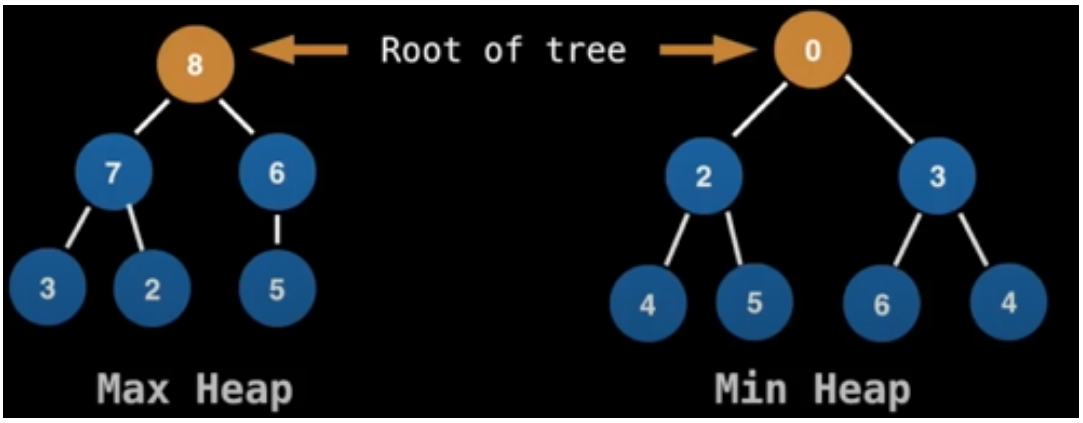


Figure 21:

Both heaps of the image are actually **binary heaps**, because every node has exactly two children. And the children cannot seek values that are not drawn in.

Heaps form the underlying canonical structure of the priority queues, so much so that priority queues are sometimes called heaps, although this isn't technically correct, since the priority queue is an abstract data type, meaning it can be implemented with other data structures also.

When and where is a Priority Queue used?

- Used in certain implementations of Dijkstra's Shortest Path algorithm.
- Anytime you need to dynamically fetch the 'next best' or 'next worst' element.
- Used in Huffman coding (which is often used for lossless data compression).
- Best First Search (BFS) algorithms such as A* use PQs to continuously grab the next most promising node.
- Used by Minimum Spanning Tree (MST) algorithms.

6.3 Complexity PQ with binary heap

Binary Heap construction	O(n)
Polling	O(log(n))
Peeking	O(1)
Adding	O(log(n))
Naive Removing	O(n)
Advanced removing with help from a hash table *	O(log(n))
Naive contains	O(n)
Contains check with help of a hash table *	O(1)

Figure 22:

There exists a method to construct a binary heap from an unordered array in linear time. Pooling or removing an element from the root of the heap takes logarithmic time, because you need to restore the heap invaraint, which can take up to a logarithmic time. Peeking or seeking the value at the top of our heap can take constant time. Adding an element to our heap take logarithmic time since we will possibly have to reshuffle the heap by bubbling up the value.

On the other hand, removing an element which is not at the root of our heap needs to do a linear scan for the element we want to remove and then remove in. The problem with this is that it can be

extremely slow in some situations, specially if we are removing a lot of elements. We should avoid this operation when possible. However, there exists another way of removing an element which is not at the top of the heap which requires lower time, and requires the use of a hash table.

Again, adding an element naively in the heap takes a linear time since we have to scan through all the elements. But we can also lower the required time by using a hash table.

The downside of using a hash table is that it requires an extra linear space factor and it does add some constant overhead because we are accessing our table a lot when doing swaps.

6.4 Turning Min PQs into Max PQs

Problem: Often the standard library of most programming languages only provide a min PQ, which sorts by smallest elements first, but sometimes we need a Max PQ.

Since elements in a priority queue are comparable, they implements some sort of **comparable interface** which we can simply **negate** to achieve a Max Heap.

We can negate our comparison operation, for instance <= would be turned to >=. Note that we turn it to >= and not >.

But, there exists another trick, which consists of **negating the numbers as we insert them into the PQ and negating them again when they are taken out**. This has the same effect as negating the comparator.

6.5 Adding elements to a Binary Heap

Ways of Implementing a Priority Queue

Priority queues are usually implemented with heaps since this gives them the best possible time complexity.

The priority Queueu (PQ) is an Abstract Data Type (ADT), hence heaps are not the only way to implement PQs. As an example, we could use an unsorted list, but this would not give us the best possible time complexity.

There are many types of heaps we could use to implement a priority queue: Bynary Heap, Fibonacci Heap, Binomial Heap, Pairing Heap, and so on. But for simplicity we will only see Binary Heaps.

A **binary heap** is a **binary tree** that supports the **heap invariant**. In a binary tree every node has exactly two children.

A **complete binary tree** is a tree in which at every level, except possibly the last, is completely filled and all the nodes are as far left as possible.

So, when we insert nodes, we will insert them at the bottom row. As far left to meet this complete binary tree property. Using this property is very important because it gives us an **insertion point**, no matter the heap looks like or what values are in it.

Binary Heap Representation

There is a canonical way of representing a binary tree, which is to use an array. Using an array is very convenient because when we are maintaining this complete tree property, the insertion position is just the last position in our array. However, this is not the only way we can represent the heap,

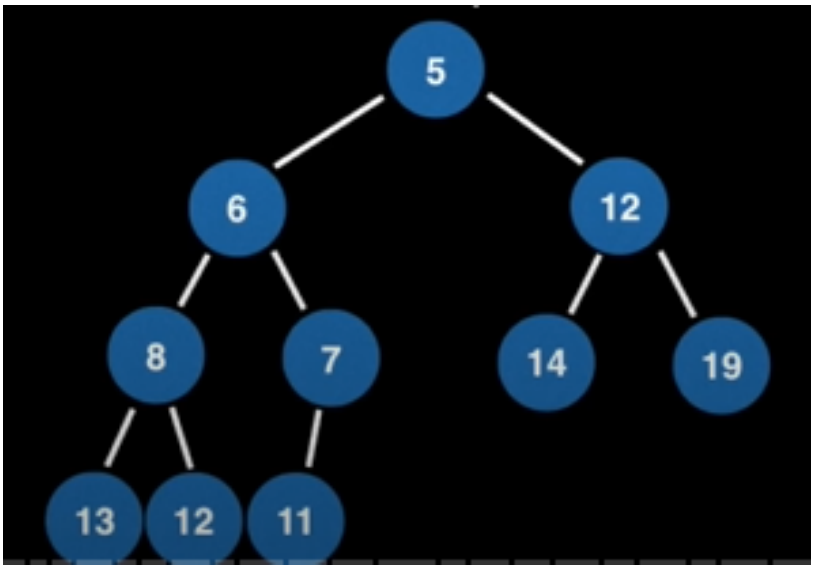


Figure 23:

we can also represent the heap using objects and pointers, recursively adding and removing nodes as needed. But the array construction is very elegant and also very very fast.

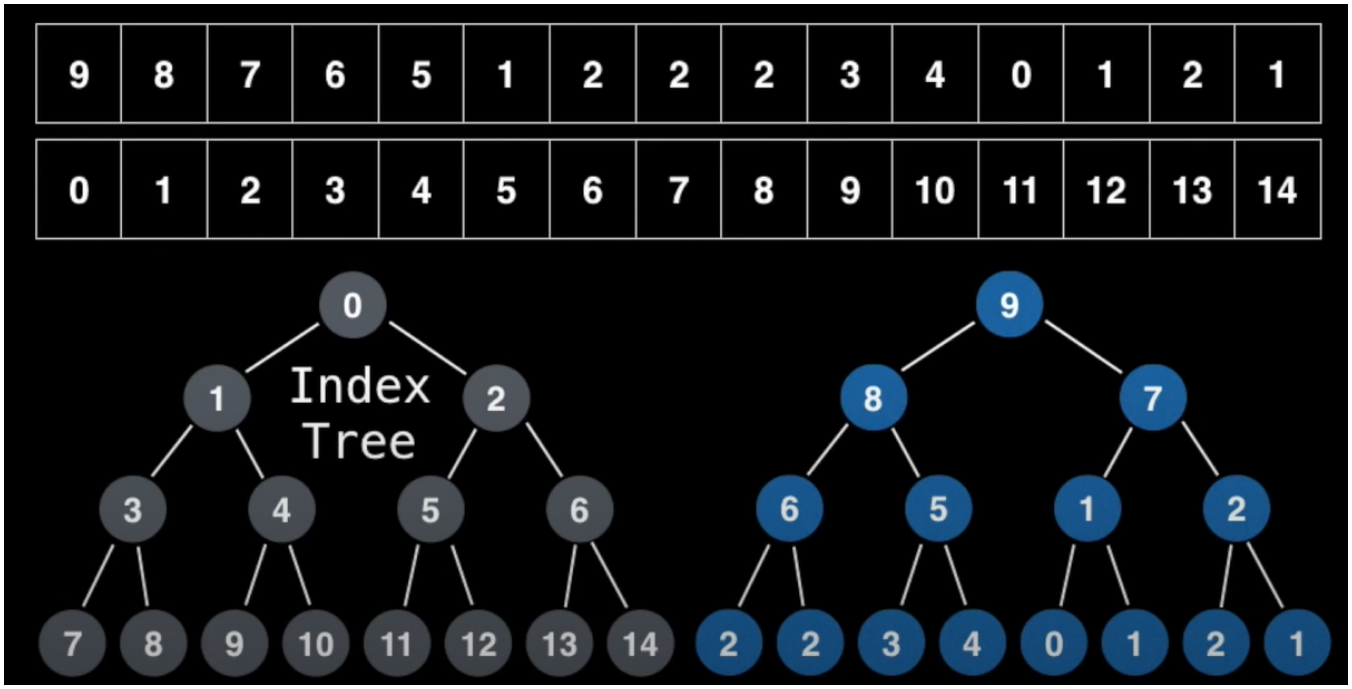


Figure 24:

The left tree shows the indexing of the tree, while the tree on the right shows the actual values of the tree. Remark that as we read elements in the array from left to right, it is as we are pacing through the heap one layer at a time.

Another interesting property of inserting in a binary heap using arrays is that we can easily access all the children and parent nodes. So, supposing i is the index of a parent node, then the

left child is going to be at index two times i plus one, and the right child of that node is going to be at two times i plus two (note that this is zero-based, and if we want it to be one based we jsut have to substract one to these values).

```
# Let i be the parent node index

# Left child index: 2i+1
# Right child index: 2i+1
#   (zero based)
```

So, using this technique, we have all the information we need to manipulate our array.

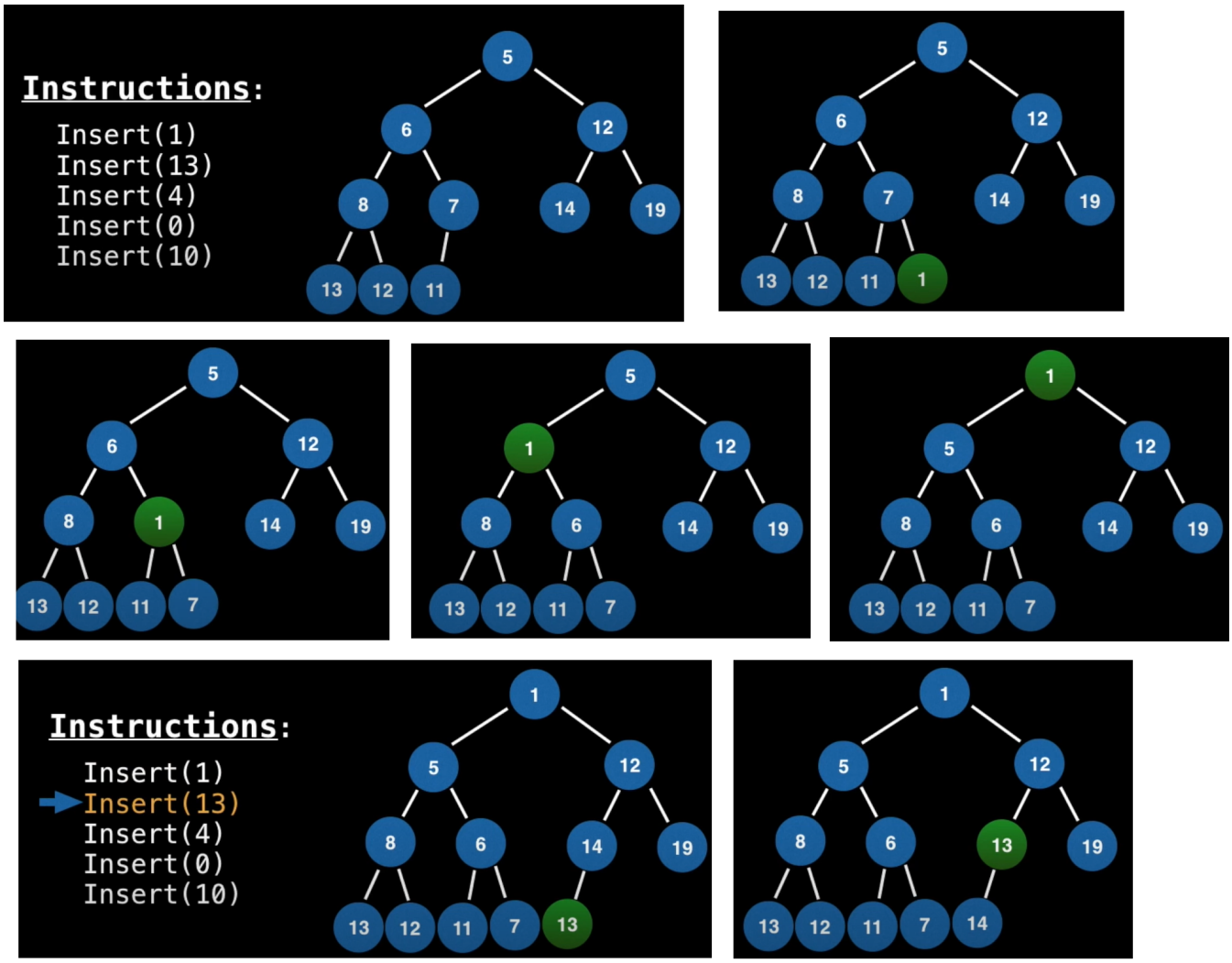


Figure 25:

So, for adding elements to a binary heap, we have to take into account that we must preserve the **Heap Representation**, so we will first insert the new element at the next position, in this case at the right of the node with value 11. This does not preserve the heap representation, so we will swap the new node with its parent node, in this case we will have to do it again, and then we will also have to do it again. We will have to do the swapping until the heap representation is preserved.

Next, we have to insert the node with value 13, we insert it at the next position, and since the heap representation is not preserved, we have to swap it with its parent value, and we see that now the representation is preserved, so we are okay.

And we will do this process with all values.

6.6 Removing elements from a Binary Heap

In general, with heaps, we always want to remove the root value because it's the node of interest, it's the node with the highest priority, the one with the highest or the lowest value.

When we remove the root we call it **pooling**. A special thing about removing the root is that we do not have to search for its index, because in an array implementation, it has position or index 0.

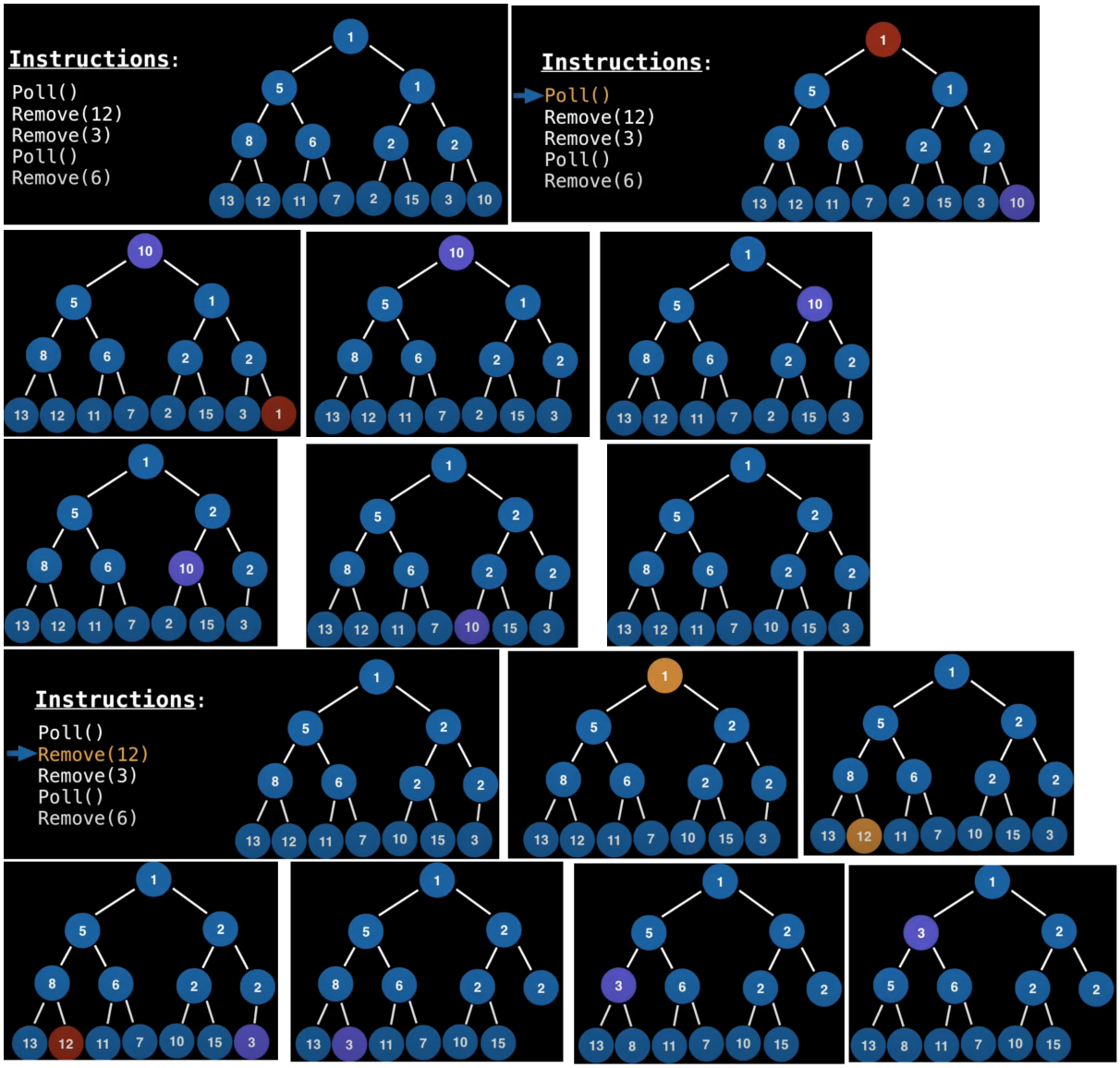


Figure 26:

When we pool, in red we have the node we want to remove and in purple the node we will swap

it with. We will **always swap it with the last node**, the one at the bottom right, the one at the end of our array, since we will always have its index.

So we begin by swapping them and then we get rid of the node we want to remove, which is now at the last position of our array. And now, we see how the heap representation is not preserved, so we have to make sure it does. So we have to do what it is known as **bubbling down**, when adding to the heap in the previous case we did **bubble up**.

So, we look at 10's children and see which has the lower value (because we have a Min Heap), and we swap our node with the smallest node (and **when we have a tie we have to select the LEFT node**). So, we keep bubbling down until the heap representation is preserved.

On the other hand, when we want to remove a chosen value, not the value at the root, we will first have to do a **linear search** for that node (**the most common thing is to do a Breadth First Search, since we have ordered our heap in levels, from left to right, so when iterating linearly through the array we would be doing a BFS**). We start at the root node and we linearly search through all nodes until we find the node we want to remove.

Once we have found it, we mark it as red since it is the node we want to remove, and the last node, the one at the last position in the array, will be marked as purple, since we will use it for swapping. We swap them and we remove the node we want to remove, which is now at the last position. Now we are violating the heap invariant, so we have to bubble up our node until the heap invariant is satisfied.

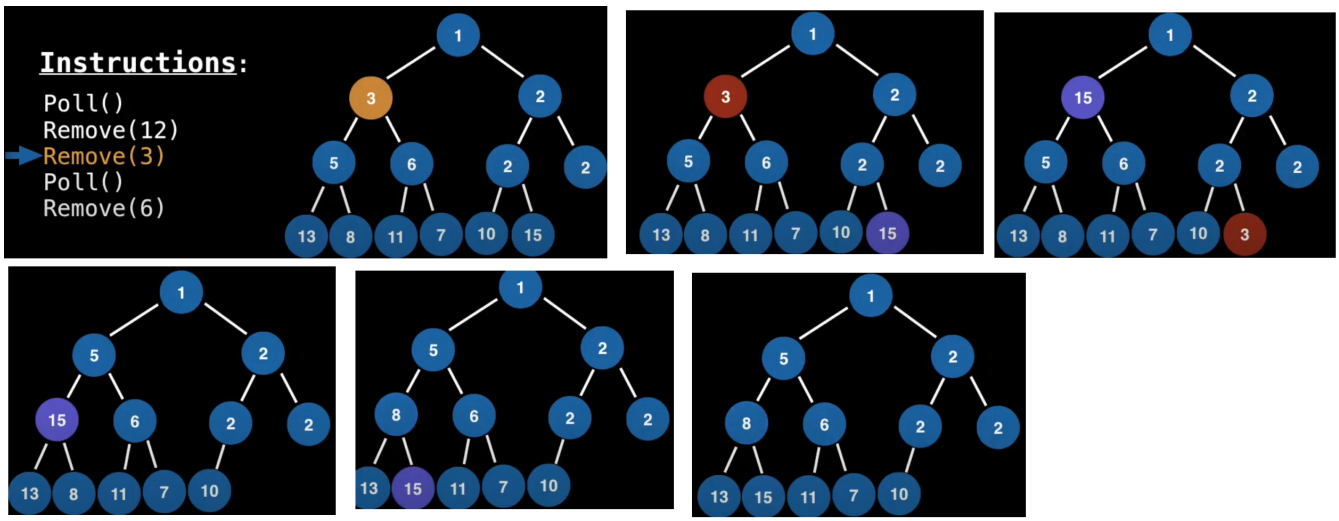


Figure 27:

Now we want to remove the node with the value 3, so, as earlier, we first have to do a linear search to find it. Then, we have to mark it to remove in red and select in purple the last node of the heap, the one which is in the last position in the array. We swap them and remove the node we want to remove, which is now at the last position. Now, the heap invariant is not preserved, so we have to make sure it is.

Now, **our node is not at the last level, and since the nodes above it do preserve the heap invariant, we have to BUBBLE DOWN the node**. So, we bubble it down until we assure that the heap invariant is satisfied.

From this example, we can conclude that **Polling takes a logarithmic time ($0(log(n))$) and Removing takes a linear time ($0(n)$)**.

6.7 Removing Elements From Binary Heap in $0(log(n))$

The inefficiency of the removal algorithm comes from the fact that we have to perform a linear search to find out where an element is indexed at. So, the solution comes from using a**Hastable** to find out where a node is indexed at.

A Hashtable provides a constant time lookup and update for a mapping from a key (the node value) to a value (the index).

Problem:

What if there are two or mode nodes with the same value? What problems would that cause?

Instead of mapping one value to one position, we will map one value to multiple positions. We can maintain a **Set** or **Tree Set** of indexes for which a particular node value (key) maps to.

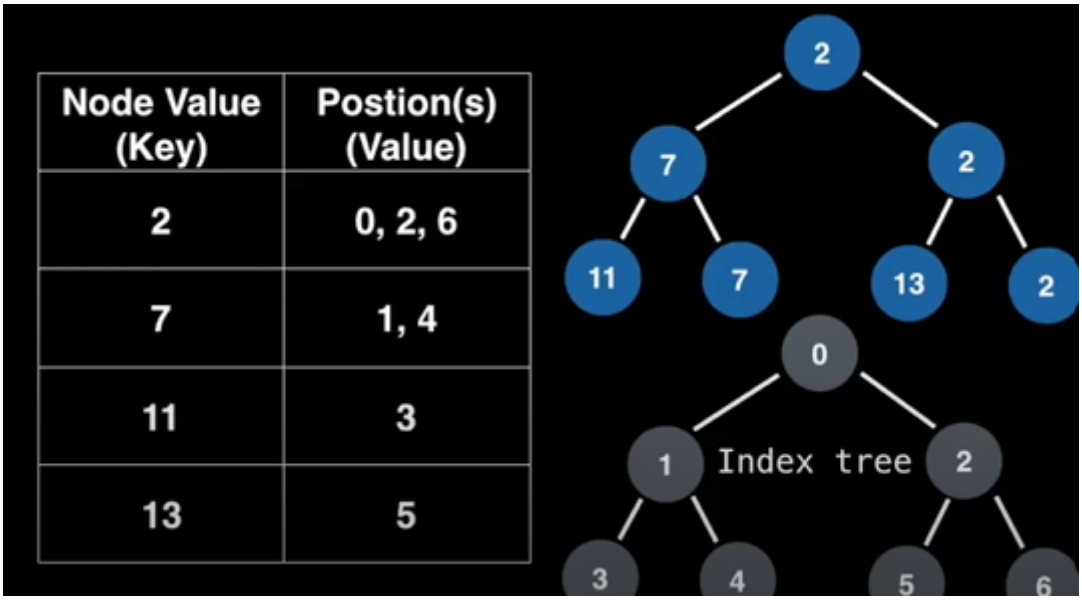


Figure 28:

The blue heap has repeated values, for instance the 2 is there three times and 7 is there twice. The three from below shows the index tree, which will help us determine the index position of the nodes in the tree. On the left we can see the Hashtable, we can see how different values can be found at multiple indexes. If the nodes move in the tree we have to take into account that in the Hashtable, tracking all the movements.

When we want to remove a repeated node in our heap, we have to pick which node to remove. But in reality it does not matter which one we choose. If we want to remove a 2, we have 3 different possible values, so it does not matter which one we remove. **It does not matter which one we remove AS LONG AS THE HEAP INVARIANT IS PRESERVED**.

First we have to insert 3, so we place it at the bottom of the heap in the insertion position, and we also keep track of it in the Hashtable. Now that we have inserted our new node, we have to make sure that the heap invariant is preserved, so we bubble it up until it is, swapping their

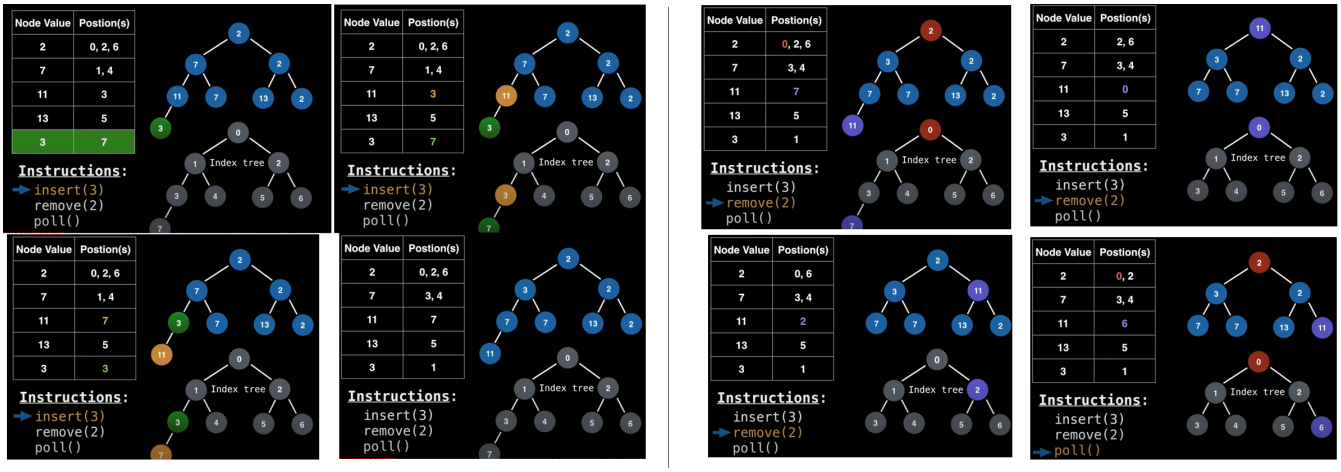


Figure 29:

positions in the Hashtable.

Now we have to remove 2 from the heap. It does not matter which 2 we remove if the heap invariant is satisfied. In this case if we remove the last two we will immediately satisfy the heap invariant, but we will replace the first one, which happens to be at the root.

So, for removing the 2 at the root, we mark it as red and select the node at the bottom of the heap (purple) and swap them, making sure that we swap their indexes in the Hashtable. And now we remove the node we want to remove, which is located at the bottom of the heap, at take it out from the Hashtable.

Now we need to satisfy the heap invariant, so we need to bubble down the node. We select the lowest children node (because we have a Min Heap) and make the swap, and repeat this process until the heap invariant is satisfied.