Data Structures and Algorithms
Part 1

Iñaki Lakunza

April 3, 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:

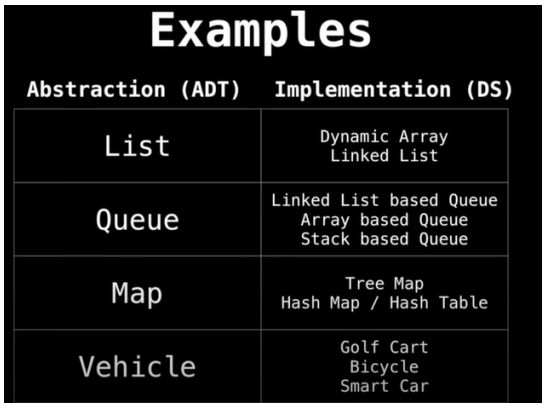


Figure 1:

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.

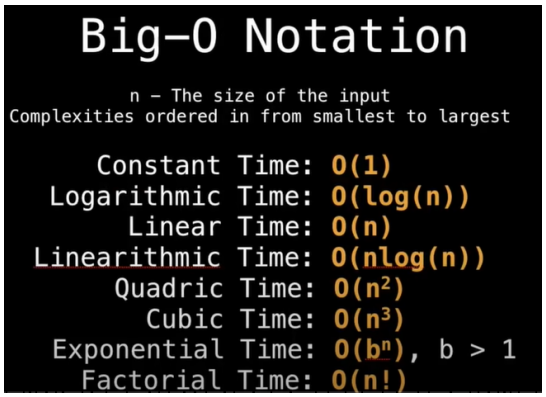


Figure 2:

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

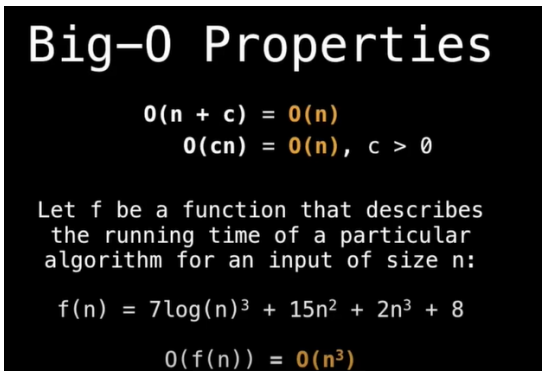


Figure 3:

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

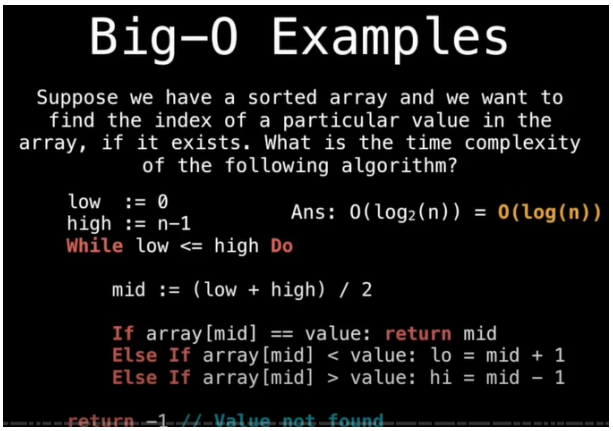


Figure 4:

```
    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

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:

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.

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**.

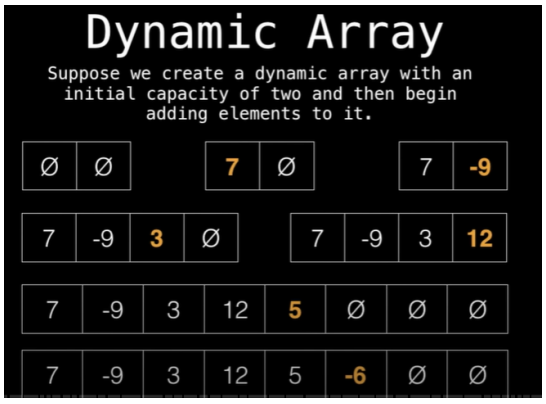


Figure 6:

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

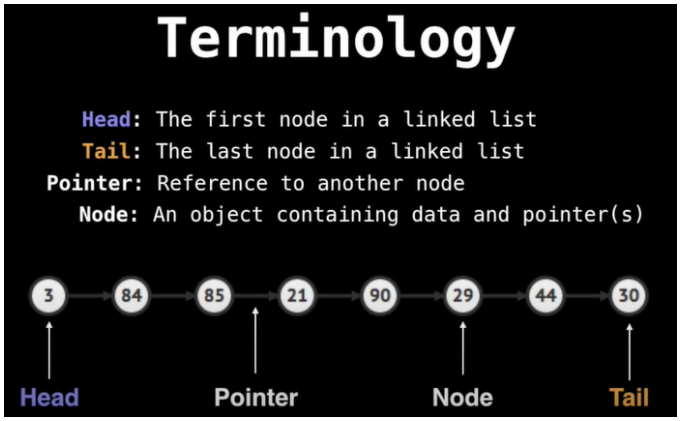


Figure 8:

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

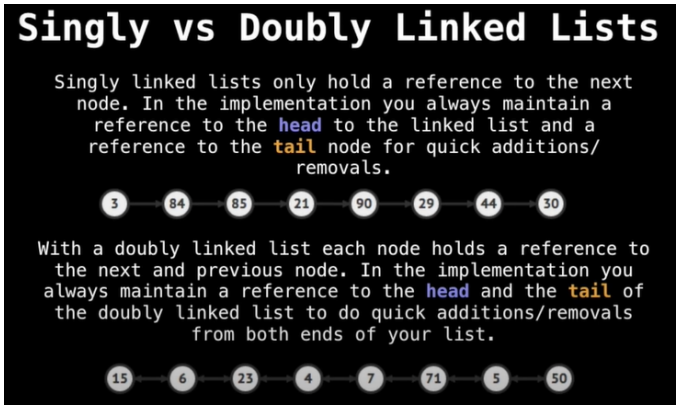


Figure 9:

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

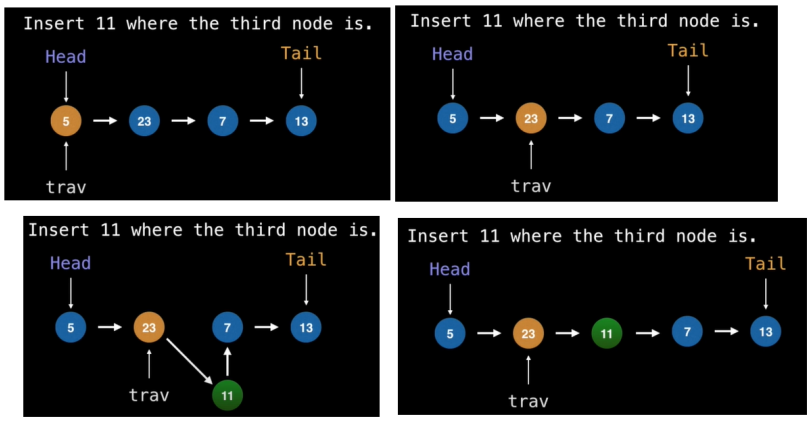


Figure 10:

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.

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

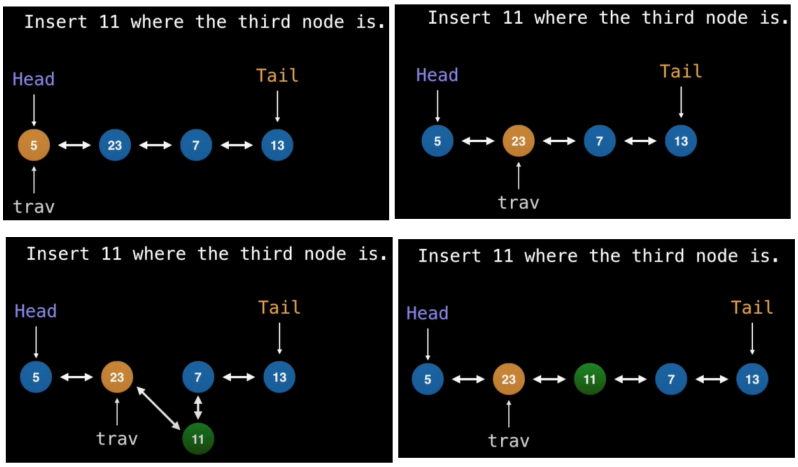


Figure 11:

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

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**.

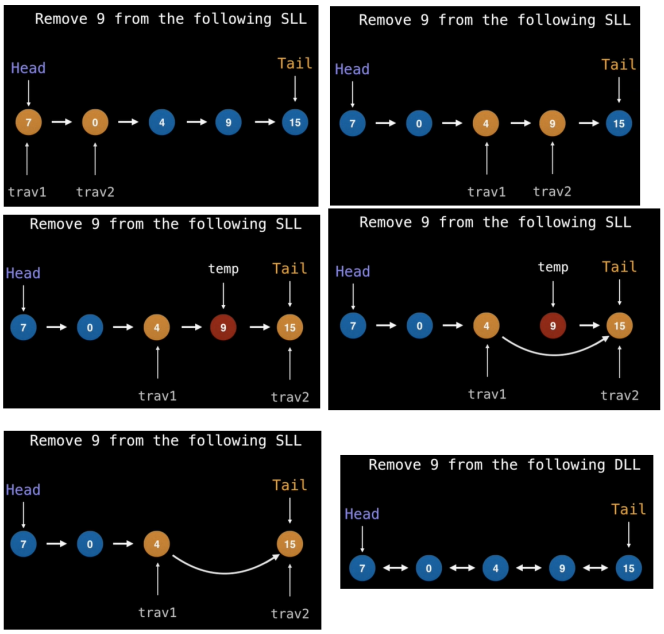


Figure 12:

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

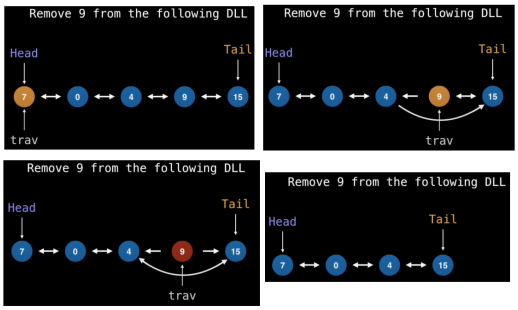


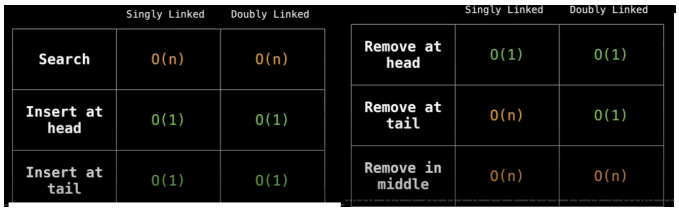
Figure 13:

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.

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.

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.

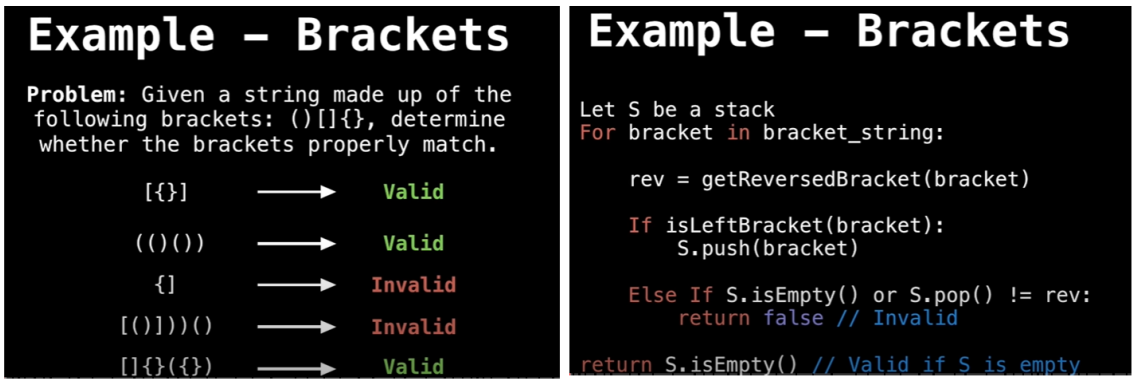


Figure 16:

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**.

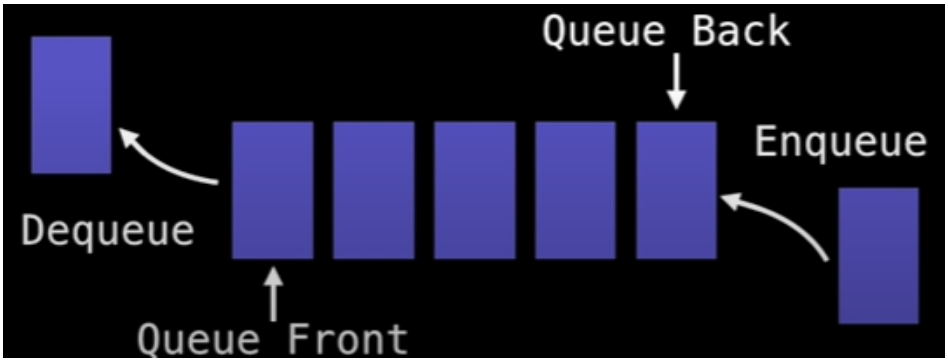


Figure 17:

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.

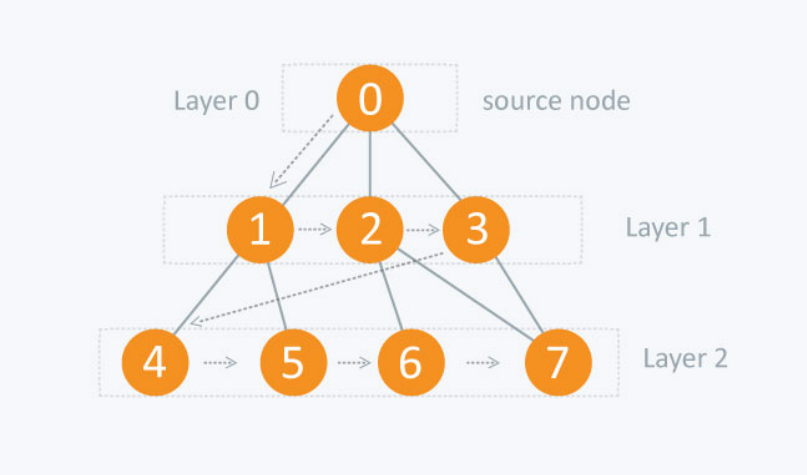


Figure 19:

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.

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.

```
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:

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.

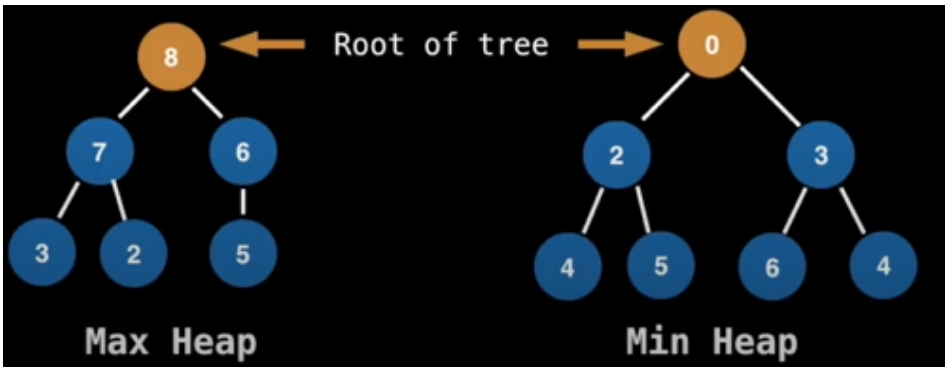


Figure 21:

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, 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.

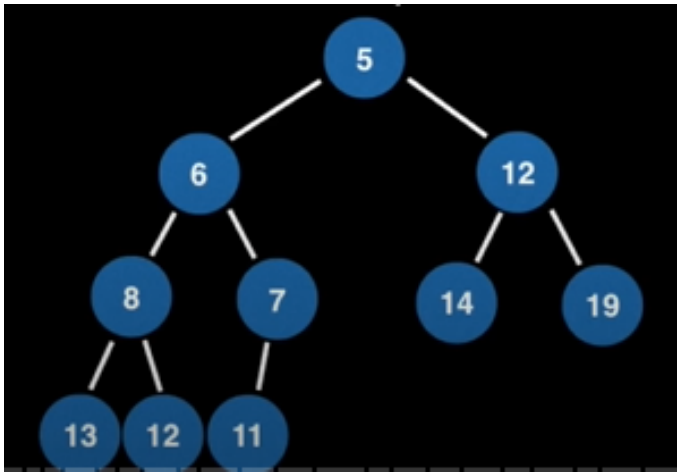


Figure 23:

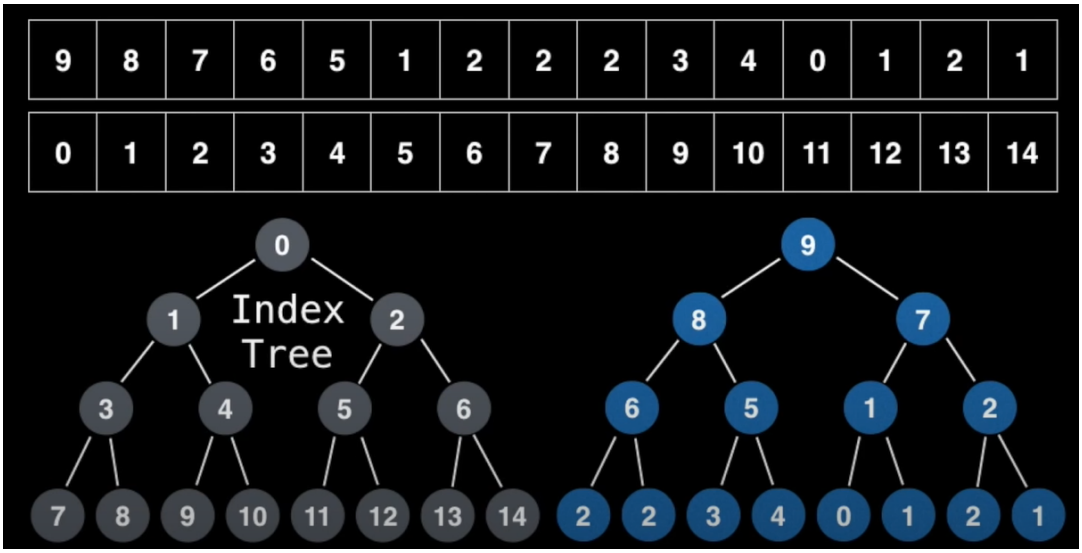


Figure 24:

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.

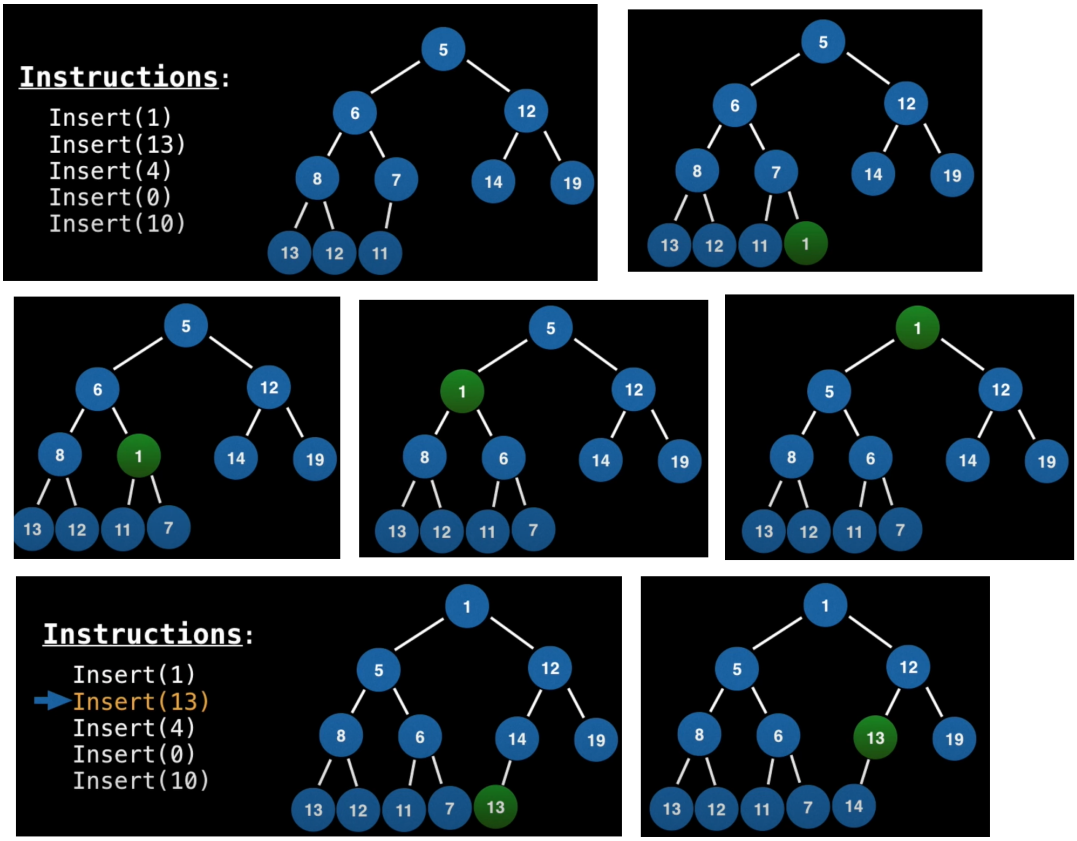


Figure 25:

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.

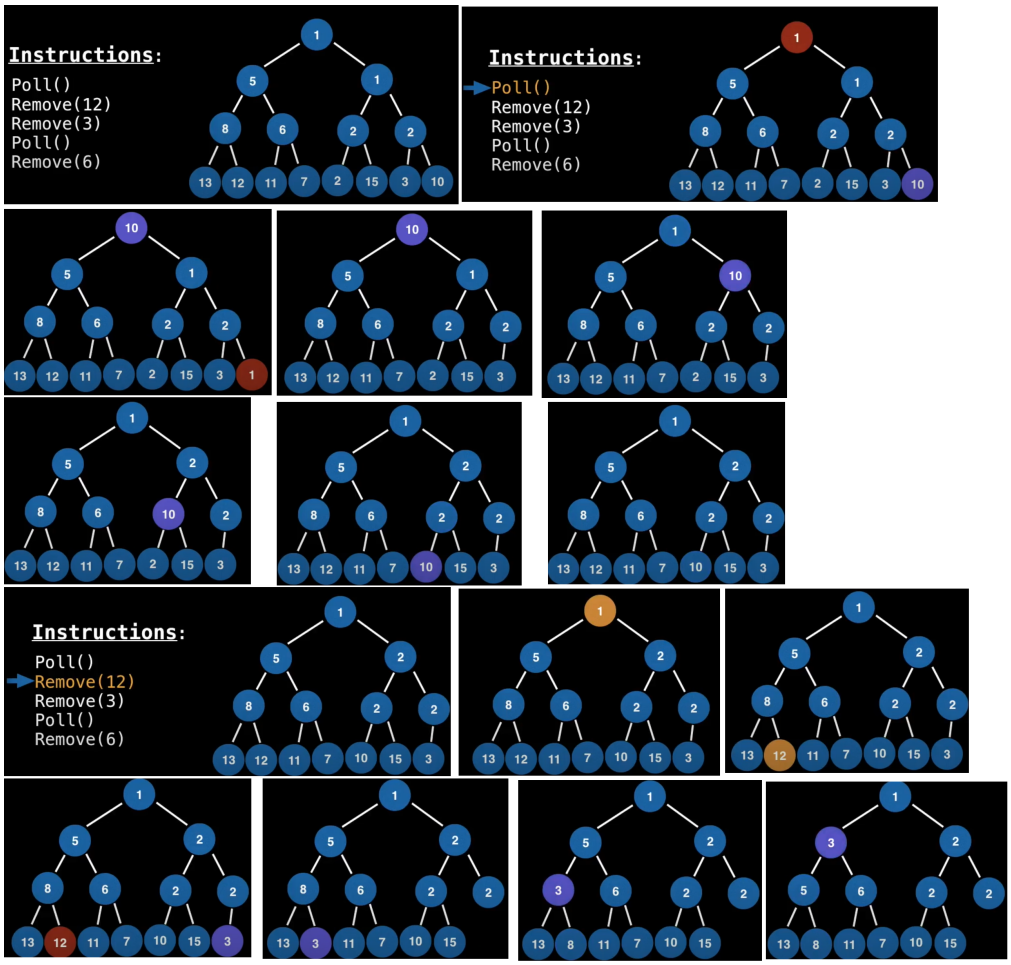


Figure 26:

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.

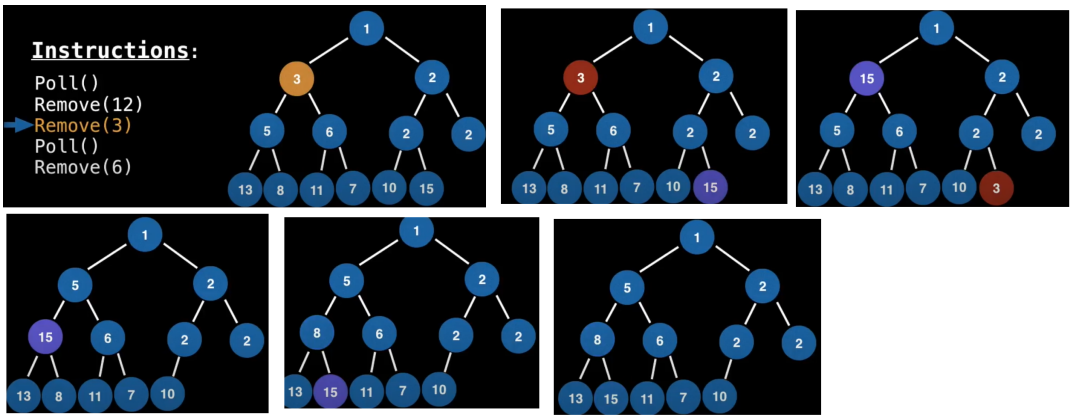


Figure 27:

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.

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.

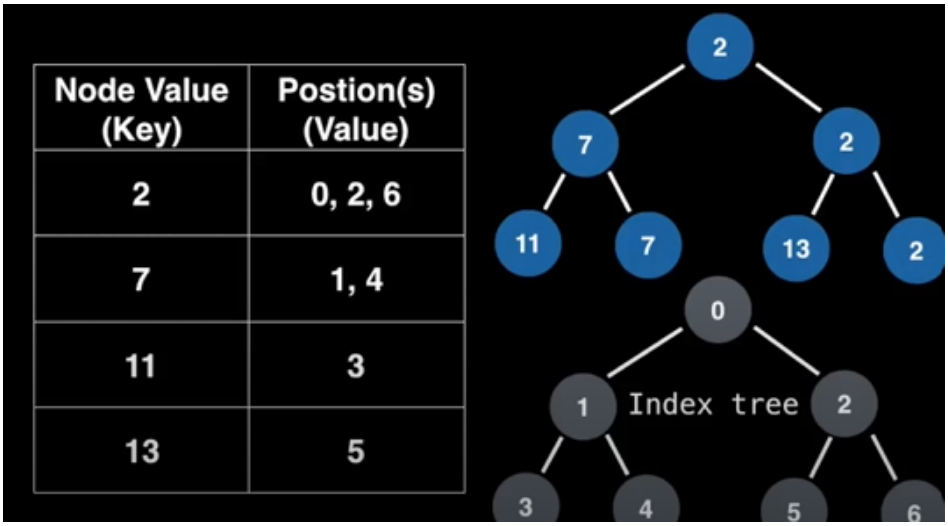


Figure 28:

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**.

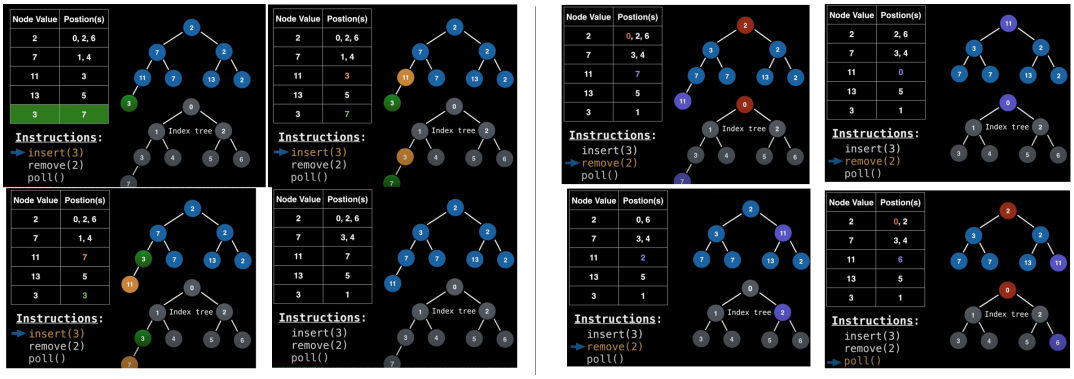


Figure 29:

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 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.

7 Union Find - Disjoint Set

7.1 Introduction

Union Find is a data structure that keeps track of elements which are split into one or more disjoint sets. It has two primary operations: **find** and **union**.

Find: Given an element, the union find will tell you what group that element belongs to.
Union: Merges two groups together.

Where is union find used?

- Kruskal's minimum spanning tree algorithm.
- Grid percolation.
- Network connectivity.
- Least common ancestor in trees.
- Image processing.

7.2 Complexity of Union Find

Construction	O(n)
Union	α(n)
Find	α(n)
Get component size	α(n)
Check if connected	α(n)
Count components	O(1)

Figure 30:

$\alpha(n)$ = **Amortized constant time**

The construction time of the union find algorithm is constant and Union, Find, Get component Size, Check if Connected and Count Components all take what is called **amortized constant time**, which means they take almost constant time but not quite constant time.

Finally, counting how many components we have takes a constant time.

7.3 Kruskal's Algorithm

Given a graph $G = (V, E)$ (where V is the number of vertices and E is the number of edges), we want to find a **Minimum Spanning Tree** in the graph (it may not be unique).

A minimum spanning tree is a subset of the edges which connect all vertices in the graph with the minimal total edge cost.

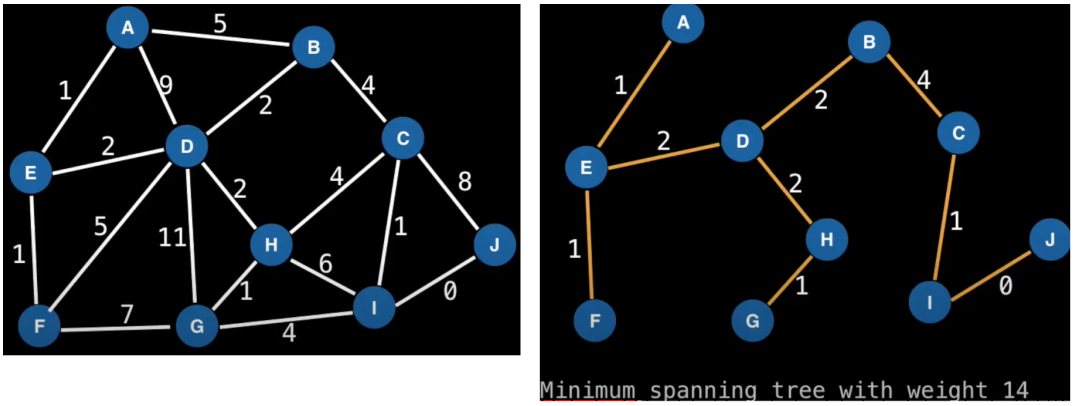


Figure 31:

In the image on the right we have a graph example, with some edges and some vertices. And a possible minimum spanning tree is the one on the right, which has total edge weight 14. Note that the minimum spanning tree is not necessarily unique, so if there is another minimum spanning tree it will also have a total edge weight of 14.

The pipeline is the following:

1. Sort edges by ascending edge weight.
2. Walk through the sorted edges and look at the two nodes the edge belongs to, if the nodes are already unified we don't include this edge, otherwise we include it and unify the nodes.
3. The algorithm terminates when every edge has been processed or all the vertices have been unified.

7.3.1 Kruskal's Algorithm: Example

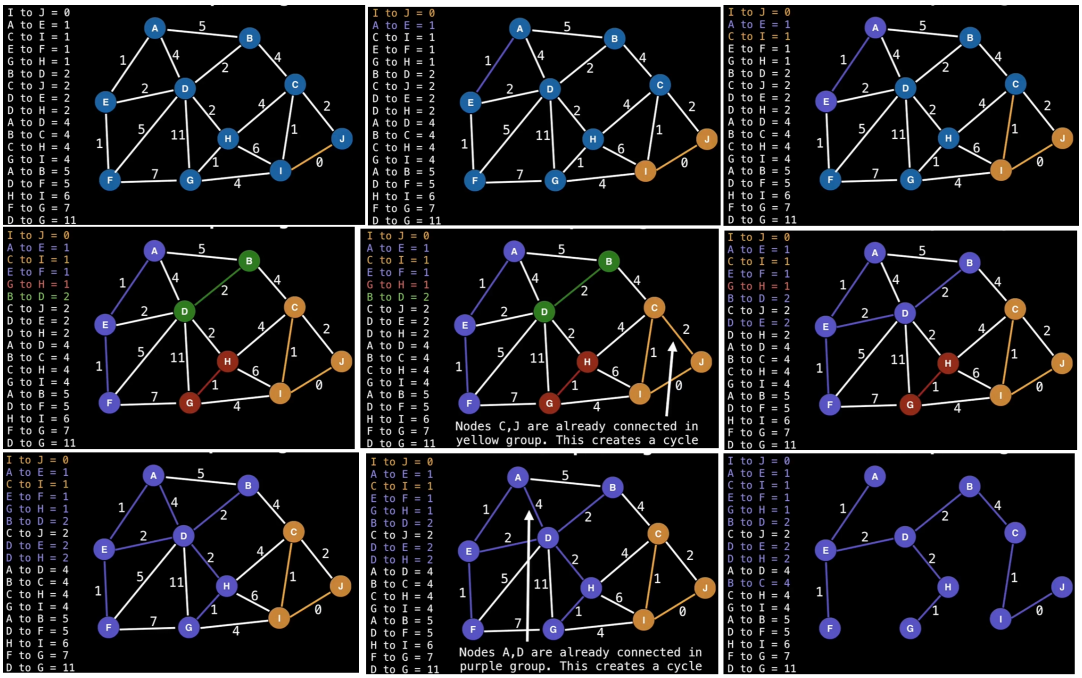


Figure 32:

We will start from the top, grouping I and J, making them orange. The next ones, A and E currently do not belong to any group, so we will unify them into a new group. Next is C to I: I belongs to group orange, but C does not have a group yet, so C can go to group orange, and we continue this way.

Now we are trying to connect C to J, but **notice that C and J already both belong to the same group, so we do not want to include that edge, since it is going to create a cycle**, so we will ignore it. To check if they belong to the same group, we will use the **find** operation in our union find to check what group they belong to. So this is where union find really comes into play.

We continue with the next edges, now we have D to E, we see that they belong to different groups, so we unite these two groups, it does not matter if we turn one purple or the other orange. This is where our **union operation** in our union find becomes useful, since we will use it to merge groups of different colors together very efficiently.

Then we continue with D to H, and do the same as we just did now. Then we have A to D, but they both already belong to the same group, so in order not to make a cycle we don't take into account this connection.

Then we have B to C, since they both belong to different groups we merge them into a larger group using **union**. And we see how we have already connected all nodes, using the minimum edge weight. So this way we have found the minimum spanning tree.

7.4 Union and Find Operations

We will first have to create our Union Find. We first have to construct a **bijection** (a mapping) between our objects and the integers in the range $[0, n)$.

Note: This step is not necessary in general, but it will allow us to construct an array-based union find.

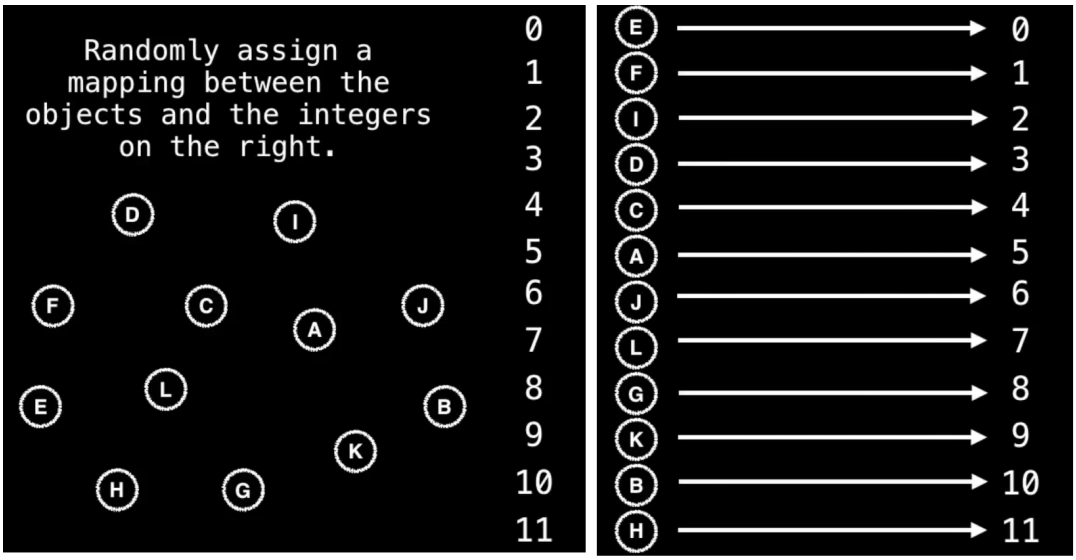


Figure 33:

If we have some random objects, and we want to assign a mapping to them, we can do it arbitrarily, as long as each element maps exactly to one number. The image on the right shows a possible random bijection.

And we want to store these mappings in a **Hashtable**, for instance, so we can do a lookup on them and determine what is everything mapped to.

Next we are going to construct an array and each index is going to have an associated object, this is possible through our mapping.

At the top part of the images in this example we can see the array we have just created, and in the center there is just a visual representation of what is going on.

In the upper left case, the value in the array for each position is currently the index which it is at. This is because originally, every node is a root node, meaning it maps to itself. But, as we will perform the instructions on the left, unifying groups together, or objects together into groups, we are going to be changing those values, mapping them to other values. And specifically, the way we are going to do it is by first, with some index i, in our array, index i's parent is going to be whatever index is at position i.

So, to begin with, we want to unify C and K, and we see that C has a root node of four and K has a root node of 9. So either C is going to become 9 or K is going to become 4. In this case we will make K become 4. Now we see how at index 9, which we know its K's position we are going to put a 4.

Next we are going to do something similar with F and E, and the same with A and J. And next we reach to the connection A to B. We see how A already is forming a group and B is a single node (a self-loop). So, in general we will be merging the smaller components into the larger ones. So we will be adding B to the group where A belongs.

We continue at it until we reach C and A, we see how each of them belongs to a group and that

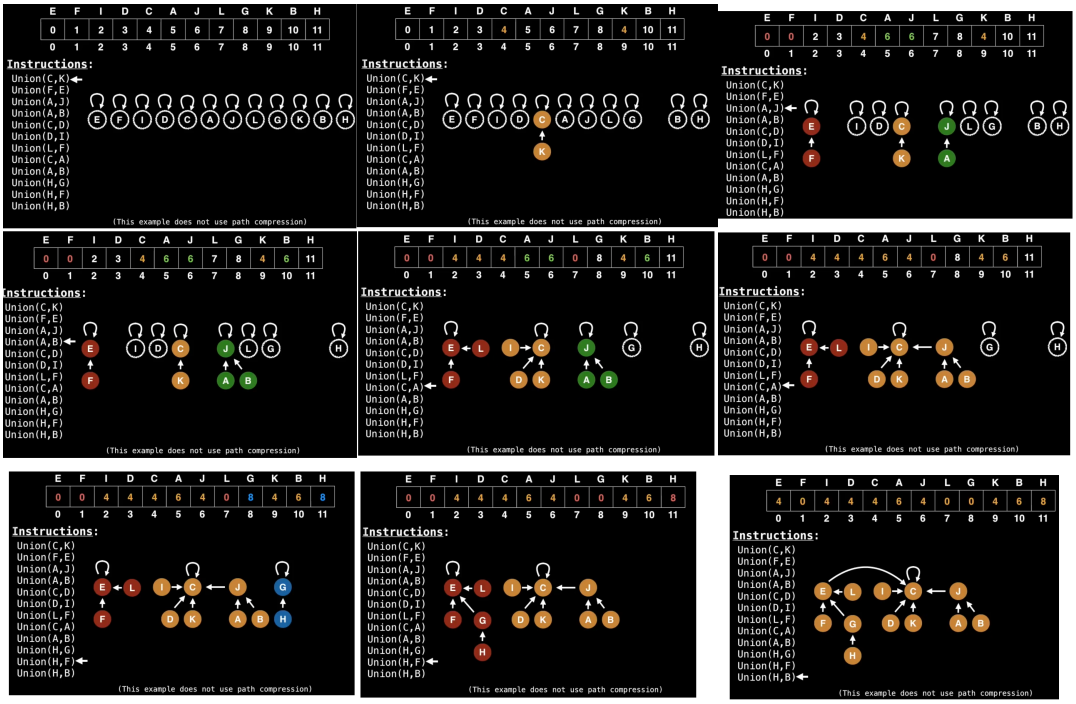


Figure 34:

both groups are different. We see how the orange group has 4 components while the green group has only 3 components, so we will be merging the smaller group into the larger group, so now J is going to be pointing at C (**see how instead of connecting C and A we point the smallest groups root node into the larger groups root node**).

Now we have A and B, and we see how they both belong to the same group, so we do not need to unify them, since they already belong to the same group.

We continue at it and we reach the connection H to F. We see that they belong to different groups so we merge the smallest one to the largest one (**see how we point the root of the smallest one to the root of the largest one, and not H to F**).

And finally we want to merge H and B, we see how they belong to different groups, so we merge the smallest with the largest (so E is pointing to C).

And this way we are done unifying the whole graph. Note that in this example we haven't used the **path compression technique**.

Summary

Find Operation: To find which component a particular element belongs, finding the root of that component by following the parent nodes, until a self loop is reached.

Union Operation: To unify two elements, finding which are the root nodes of each component and if the root nodes are different make one of the root nodes be the parent of the other.

Remarks

In the Union Find data structure we do not 'un-union' elements. In general, this would be very inefficient to do since we would have to update all the children of a node.

The number of components is equal to the number of root remaining. Also, remark that the number of root nodes never increases, they always remain the same or they decrease, since we are

unifying groups.

The implementation we have just seen does not support the almost constant time ($\alpha(n)$) time complexity, since checking if two nodes belong to the same group may take a large amount of hops. We will be achieving the wanted time complexity by using **path compression**.

7.5 Path Compression

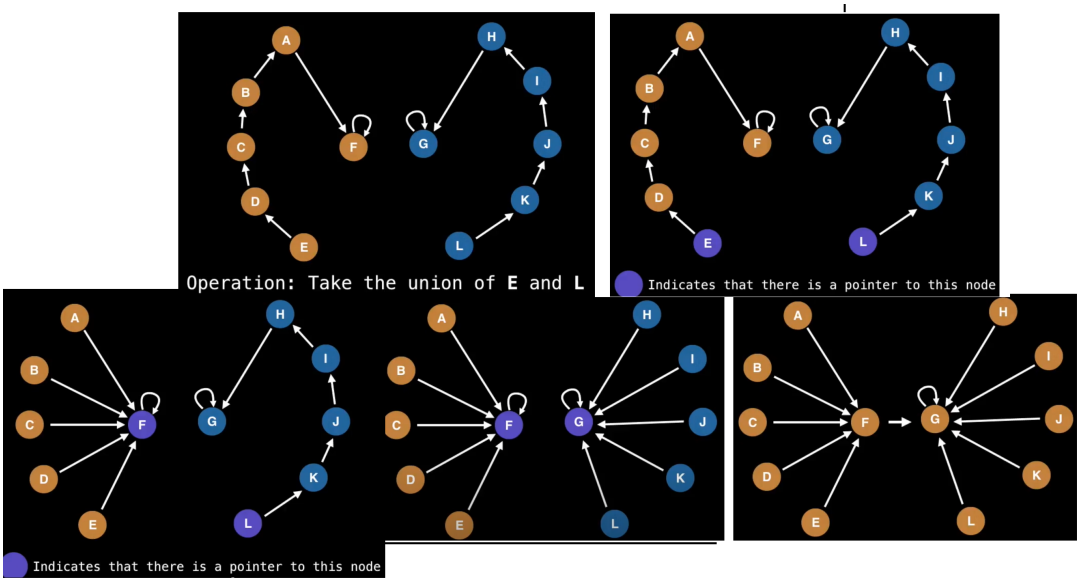


Figure 35:

Supposing we have this hypothetical union find, we will suppose we want to unify the node E and L. We will have pointers pointing at these nodes, and we would have to find their group's root node, which are F and G.

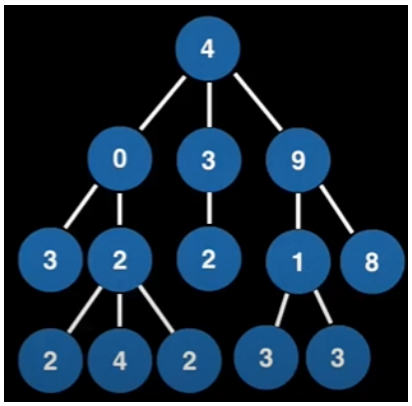
With path compression we will be finding these two root nodes, but apart from that we will be pointing the nodes in each group pointing at its root node. Once we have the reference to our parent node, we will be pointing the nodes in the group to the root node, so now everything along the path got compressed and now points to that root node. And doing so, every time we do a lookup, on either A, B, C, D or E in constant time, we will be able to find out what the parent or the root node is for that component, because we immediately point to it, we do not have to transverse a sequence of other nodes.

And we can do this since in union find we are always unifying things together and making them more and more compressed.

And once we have done this with both groups (in reality we would not have to do it now at once, since we would be doing it at each step), we can easily merge the two groups into a single group. This method is much more efficient that ending up forming large groups which take quite a large time to transverse.

8 Binary Trees and Binary Search Trees (BST)

8.1 Introduction



A **tree** is an **undirected graph** which satisfies any of the following conditions:

- An **acyclic connected** graph.
- A **connected graph** with N **nodes and** $N-1$ **edges**.
- A graph in which **any two vertices are connected by exactly one path**.

If we have a **rooted tree** then we will want to have a reference to the root node of our tree.

It does not always matter which node is selected to be the root node because any node can root the tree.

A **child** is a node extending from another node. A **parent** is the inverse of this. So, the root node does not have a parent node, although it may be useful to assign the parent of the root node to be itself. This is important for example in a filesystem tree, if we good to the root directory, if we use **cd ..** we want to still be in the root directory, so we have to make it point to itself.

On the other hand, a **leaf node** is a node with no children.

A **subtree** is a tree entirely contained within another. They are usually denoted using triangles. Note that subtrees may consist of a single node.

What is a Binary Tree (BT)?

A **binary tree** is a tree for which every node has at most two child nodes (it may have just one).

What is a Binary Search Tree (BST)?

A **binary search tree** is a binary tree that satisfies the **BST invariant**: LEFT SUBTREE HAS SMALLER ELEMENTS AND RIGHT SUBTREE HAS LARGER ELEMENTS.

Depending on the case we will allow or not allow a children node to have the same values as the parent node, we have to decide if we allow duplicates in our tree or not. BST operations allow for duplicate values, but most of the time we are only interested in having unique elements inside our tree.

Note that we do not necessarily have to have just numbers in our binary search tree, **it can be composed of any data that CAN BE ORDERED**.

When and where are Binary Trees used?

- Binary Search Trees (BSTs)
  - implementation of some map and set ADTs
  - Red Black Trees

  - AVL Trees
  - Splay Trees
  - etc ...

- Usef in the implementation of binary heaps.
- Syntax trees (used by compiler and calculators).
- Treap - a probabilistic DS (uses a randomized BST).

8.2 Complexity of BSTs

Operation	Average	Worst
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Remove	O(log(n))	O(n)
Search	O(log(n))	O(n)

Figure 36:

In the average case, when we are given some random data, the time complexity is going to be logarithmic for all possible operations. But, in the worst case, if our tree degenerates to being a line, then we can have some linear behavior going on.

So there are some trade-off, binary trees are easy to implement, and on average it is going to have a logarithmic behavior, but in the worst case we might be looking at some linear stuff.

8.3 Inserting elements into a Binary Search Tree (BST)

Binary Search Tree (BST) elements must be **comparable** so that we can order them inside the tree.

When inserting an element we want to compare its value to the value stored in the current node we're considering to decide on one of the following:

- Recurse down left subtree ($<$ case).
- Recurse down right subtree ($>$ case).
- Handle finding a duplicate value ($=$ case).
- Create a new node (found a null leaf).

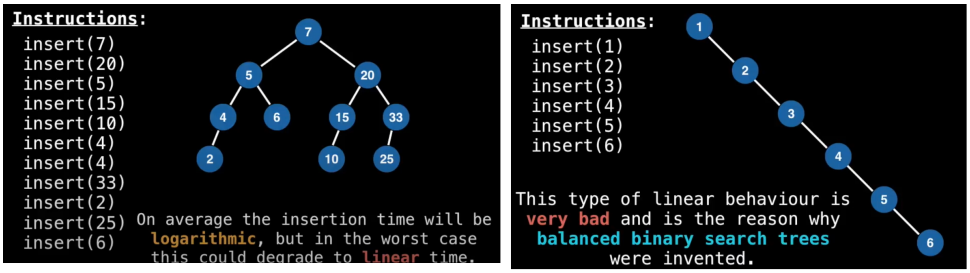


Figure 37:

When we encounter a value that is already in the tree. If our tree supports duplicate values then we add another node, otherwise we do nothing. Here are two examples, in the case of the left we do not accept duplicate values in the tree.

Because it may happen that we end up having a degenerate tree, as in the case on the right, some other trees, like **balanced binary trees** have been invented.

8.4 Removing elements from a Binary Search Tree (BST)

Removing elements from a Binary Search Tree (BST) can be seen as a two step process.

1. **Find** the element we wish to remove (if it exists).
2. **Replace** the node we want to remove with its successor (if any) to maintain the BST invariant.

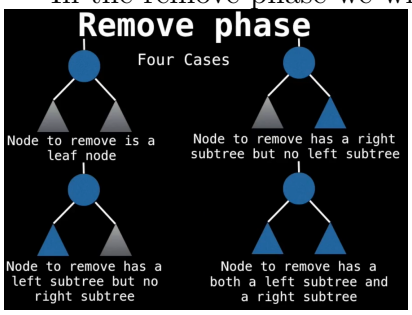
Recall the **BST invariant**: left subtree has smaller elements and right subtree has larger elements.

8.4.1 Find Phase

When searching our BST for a node with a particular value, one of the following four things will happen:

1. We hit a **null node** at which point we know the value does not exist within our BST.
2. Comparator value **equal to 0** (found it!).
3. Comparator value **less than 0** (the value, if it exists, is in the left subtree).
4. Comparator value **greater than 0** (the value, if it exists, is in the right subtree).

In the remove phase we will have one of the following four cases:



- Node to remove is a leaf node.
- Node to remove has a right subtree but no left subtree.
- Node to remove has a left subtree but no right subtree.
- Node to remove has both, a left subtree and a right subtree.

And so, we will have to act according to each case:

- **Case I, Node to remove is a leaf node:** If the node we wish to remove is a leaf node the we may do so without side effects.
- **Cases II & III, either the left/right child node is a subtree:** The successor of the node we are trying to remove in these cases will be the **root node of the left/right subtree**.

  It may be the case that we are removing the root node of the BST, in which case its immediate child becomes the new root.
- **Case IV, Node to remove has both, a left subtree and a right subtree:** We have to decide in which subtree will the successor of the node we are trying to remove be.

  We will be using BOTH. The successor can either be the largest value in the left subtree or the smallest value in the right subtree.

Here is a justification for why there could be more than one successor in the 4^{th} case:

The **largest value** in the **left subtree** satisfies the BST invariant since it:

1. Is larger than everything in left subtree. This follows immediately from the definition of being the largest.
2. Is smaller than everything in right subtree because it was found in the left subtree

On the other hand, the **smallest value** in the **right subtree** satisfies the BST invariant since it:

1. Is smaller than everything in right subtree. This follows immediately from the definition of being the smallest.
2. Is larger than everything found in left subtree because it was found in the right subtree.

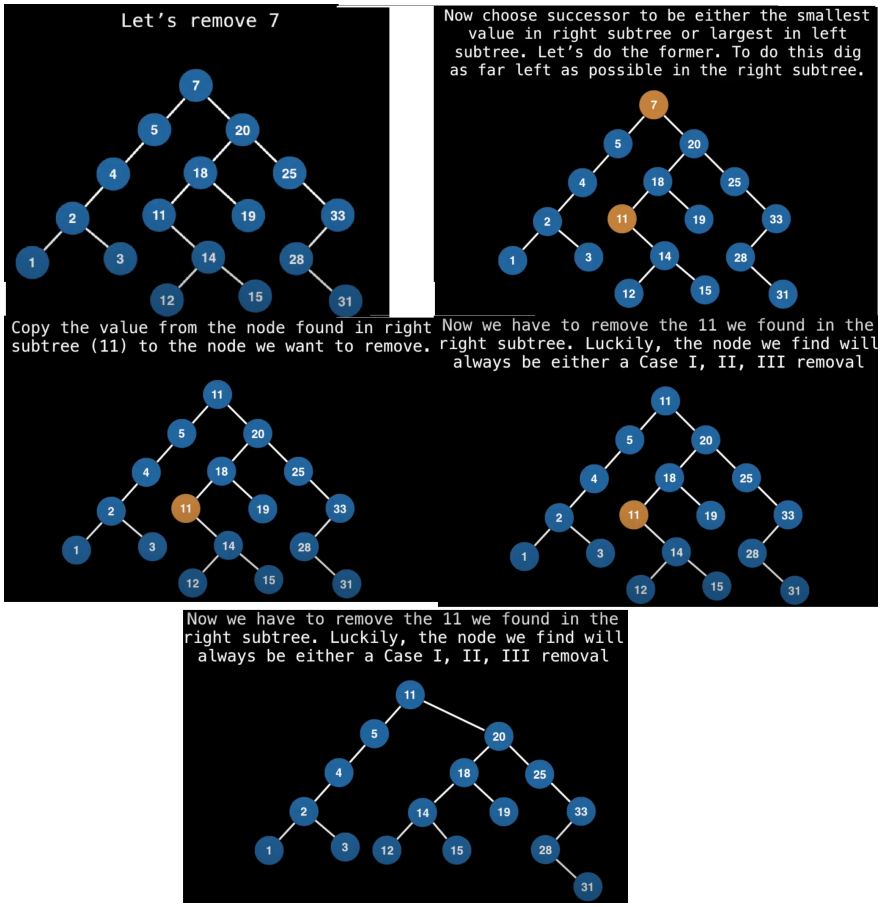


Figure 38:

In this example we want to remove 7, which belongs to the case IV, since it has both, a left and a right subtre. So first we have to find 7.

Now, we have to pick a successor in our left subtree or in our right subtree. In this example we will find **the smallest value in the right subtree**. Starting from the node we want to remove, we are going to go to its right subtree, to its right child, and then we are going to dig as far left as we can. And we will end when we have no more left childs, in this case on node 11. This will be the successor, since it is the smallest node in the right subtree.

Now we will copy the value from the node found in the right subtree (11) to the node we want to remove (7). Now we are going to remove the successor from its original position. **The node we find will always be either a case I, II or III**, so we can easily remove it.

So, we take it out and swap it with its unique subtree if it has.

8.5 Tree Traversals (Preorder, Inorder, Postorder & Level order)

Preorder, **Inorder** and **Posorder** traversals are naturally defined recursively:

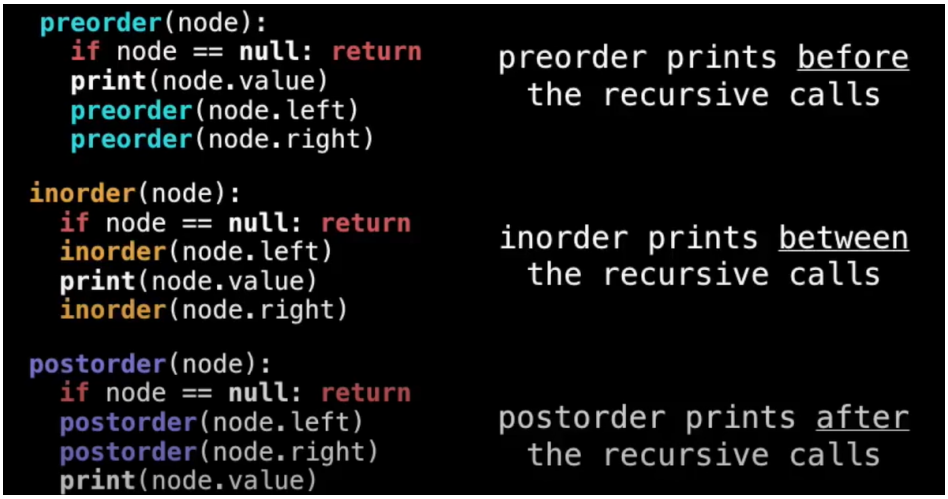


Figure 39:

Preorder prints **before** the recursive calls.
Inorder prints **between** the recursive calls.
Postorder prints **after** the recursive calls.

We can see that the only thing that is different between them is where the print statements is.

8.5.1 Preorder Traversal

In this case we print the value of the current node, and then traverse the left subtree followed by the right subtree.

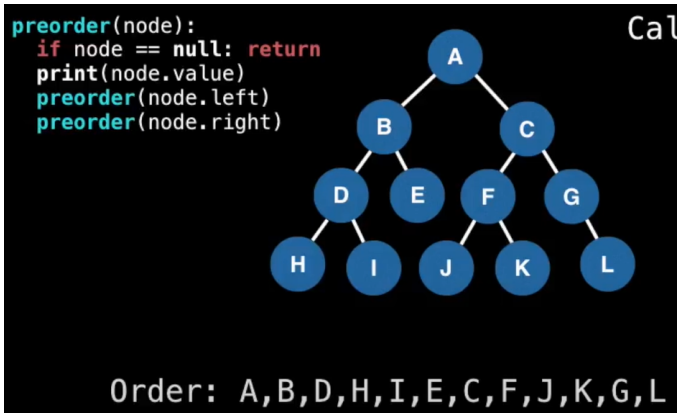


Figure 40:

We first have to print the value of the node we are at in and then tranverse the left subtree, and then the right subtree. So, since we start at the root, we will first print it, A, then we are going to go left, so print B, then D and then H, and once we have found a *None*, then we will go to the stack and go to the node which is on top of it, and throw it from the stack. In this case its D, and so

now we go to its right subtree and print it, which is I. Now if we had a subtree at its left we would go there, and then to the right subtree and so on.

We again go to the stack a go to B, and it sends us to E, then to A which sends us to C, and so on. So this way we get the following order:

A, B, D, H, I, E, C, F, J, K, G, L

8.5.2 Inorder Traversal

In this case we traverse the left subtree, then print the value of the node and continue traversing the right subtree. The example below is done with a BST.

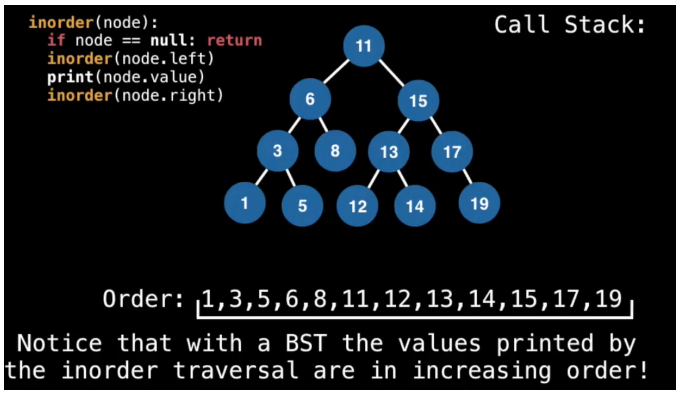


Figure 41:

We first have traverse the left subtree, and once we have done it then we print our value, so it this case the first value we print is 1. Then we recall to the stack and go to node 3, so we print it, and go to its right subtree. We traverse it (the subtree is just a node so we stay in 5), and then we print the value of the node we are at, so we print 5. Then we recall to our stack and go to 6 and print it, and go to its right subtree, ...

This way, the order that we will end up printing is the following:

1, 3, 5, 6, 8, 11, 12, 13, 14, 15, 17, 19

We see how we have printed all the values in order, that is why this traverse method is called **inorder**.

8.5.3 Postorder Traversal

In this case we traverse the left and right subtree and then we print the value we are at.

In this case, the last value we will be printing should be the root value, 11 in this example, since we traverse both the left and the right subtrees and then make the print.

So, we start at 11 and explore the left subtree, and once we find a *None* we explore its right subtree, and once we end up we print the value of the node we are at, so we print 1.

Then we recall to the stack and go to node 3, but don't print it since we haven't explored its right subtree yet. We explore it, and end up on node 5, where we can't go right anymore, so we print it. Then we recall to the stack and go to 3, and as we have already explored both its subtrees, we print it.

Then we recall the stack and go to 6, and analyze its right subtree, we end up analyzing it (we print 8), and once we have finished we go back to 6 and print it. Then we go back to 11 and analyze

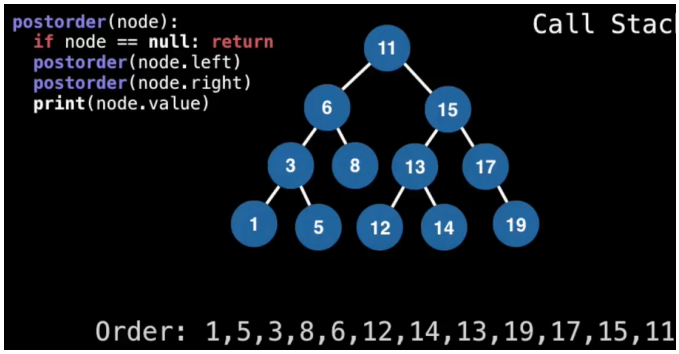


Figure 42:

its right subtree, and once we have finished (analyzing the values and printing the nexessary values) we go back to 11 and print it, and end with the loop. So the order would be the following:

1, 5, 3, 8, 6, 12, 14, 13, 19, 17, 15, 11

8.5.4 Level order traversal

In a level order traversal we want to print the nodes as they appear one layer at a time. So we want to do a **Breadth First Search**.

So, since we have to use a BFS, we will need to maintain a **Queue** of the nodes left to explore. We begin with the root inside of the queue and finish when the queue is empty.

At each iteration we add the left child and then the right child of the current node to our Queue.

So, in the case of the previous examples' BST, the order we would get using the Queue would be the following:

11, 6, 15, 3, 8, 13, 17, 1, 5, 12, 14, 19

We have to remember that when we use the queue we take the value at the top and insert the values at the bottom.

9 Hash Tables

9.1 Introduction

A **Hash Table (HT)** is a data structure that provides a mapping from keys to values using a technique called **hashing**.

We use **key-value mappings**, where a key must be unique and values can be repeated.

HTs are often used to track item frequencies. For instance, counting the number of times a word appears in a given context.

The key-value pairs we can place in a HT can be of any type not just strings and numbers, but also objects! However, the key needs to be **hashable**.

Hash Functions

A **hash function** $H(x)$ is a function that maps a key x to a whole number in a fixed range. For instance:

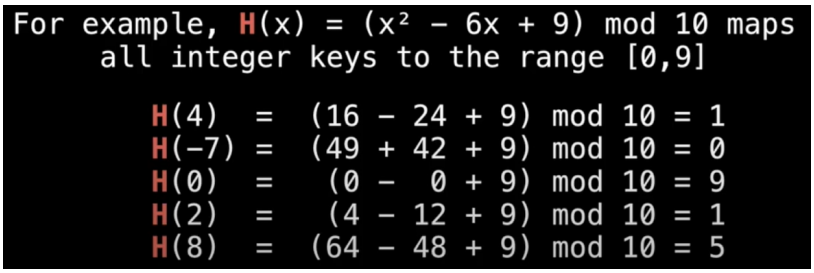


Figure 43:

In this example all values get mapped to be in a range between 0 and 9, both inclusive, no matter which integer value we use as x.

Notice that **the output is not unique**, we can get the same output using different inputs.

And we can also define hash functions for arbitrary objects such as strings, lists, tuples, multi data objects, etc... For instance:

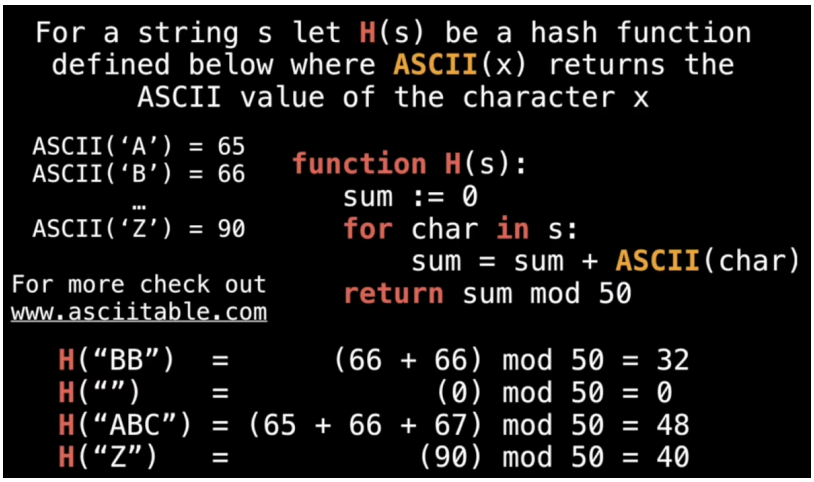


Figure 44:

Properties of Hash Functions

If $H(x) = H(y)$ then objects x and y MIGHT be equal, but if $H(x) \neq H(y)$ then x and y are certainly not equal.

We can use this property to speed up object comparisons. Instead of comparing x and y directly, a smarter approach is to first compare their hash values, and only if the hash values match we explicitly compare x and y.

Another property of hash functions is that they must be **deterministic**. This means that if $H(x) = y$ then $H(x)$ must always produce y and never another values. This may seen obvious, but it is critical to the functionality of a hash function.

Also, we try very hard to make hash functions **uniform**, to minimize the number of hash collisions.

A **hash collision** is when two objects x and y hash to the same value.

So, **what makes a key of type T HASHABLE?** Since we are going to use hash functions in the implementation of our hash table we need our hash functions to be deterministic. To enforce this behaviour, we demand that the **keys used in our hash table are immutable** data types. Hence, if a key of type T is immutable, and we have a hash function $H(k)$ defined for all keys k of type T, then we say a key of type T is hashable.

So, we need to use things like immutable strings, integers, but not things like sets or lists, or things that we can add or remove things from.

If we keep this condition and we have a hash function that is defined for all keys of type T, then we can say that that type T is hashable.

How does a hash table work?

Ideally, we would like to have a very fast insertion, lookup and removal time for the data we are placing within our hash table.

Remarkably, we can achieve all this in 0(1) time using a **hash function as a way to index into a hash table**. Note that the constant time behaviour attributed to hash tables is only true if we have a good **uniform hash function**.

How do we handle Hash Collisions?

We use one of many hash collision resolution techniques to handle this, the two most popular ones are **separate chaining** and **open addressing**.

Separate chaining deals with hash collisions by maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value.

Open addressing deals with hash collisions by finding another place within the hash table for the object to go by offsetting it from the position to which it hashed to.

9.2 Hash Table Complexity

On average, we can achieve constant time complexity. But it is extremely important to have a good **uniform hash function**. Otherwise we can get linear time in all operations.

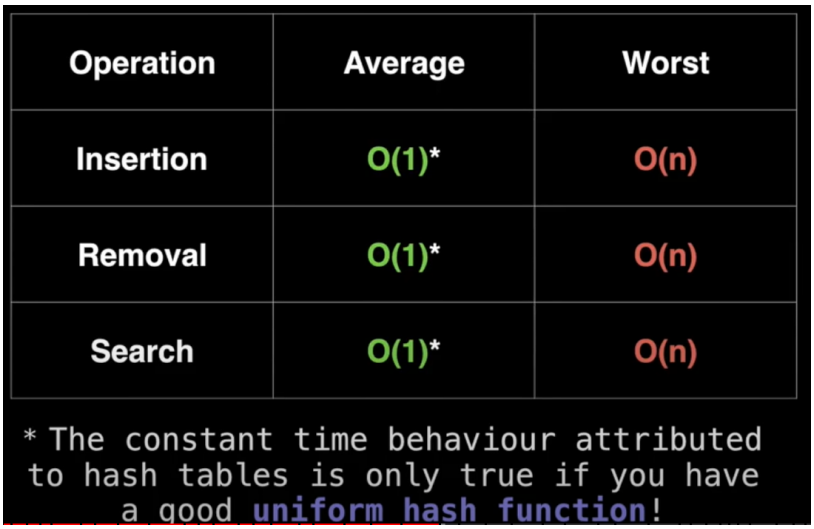


Figure 45:

9.3 Hash Table Separate Chaining

Separate chaining is one of many strategies to deal with hash collisions by maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value.

What separate chaining does is to maintain an auxiliary data structure to hold all the collisions. And so we can go back and look up inside that bucket or that data structure of values for the item we are looking for.

Note that the data structure used to cache the items which hashed to a particular value is not limited to a linked list. Some implementations use one or a mixture of: arrays, binary trees, self balancing trees and etc... .

So, **How do we maintain** $0(1)$ **insertion and lookup time complexity once our HT gets really full and we have long linked list chains?**
Once the HT contains a lot of elements we should create a new HT with a larger capacity and re-hash all the items inside the old HT and disperse them throughput the new HT at different locations.

How do we remove key-value pairs from the HT?
We apply the same procedure as doing a lookup for a key, but this time instead of returning the values associated with the key remove the node in the linked list data structure.

Can we use another data structure to model the bucket behaviour required for the separeate chaining method?
Yes, common data structures used instead of a linked list include: arrays, binary trees, self balancing trees, etc... . We can even go with a hybrid approach like using Java's HashMap. However, note that some of these are much more memory intensive and complex to implement than a simple linked list which is why they may be less popular.

9.4 Hash Table Open Addressing

9.4.1 Introduction

Quick recap on Hash Tables

The goal of the Hash Table (HT) is to construct a mapping from keys to values.

Keys must be hashable and we need a hash function that converts keys to whole numbers.

We use the hash function defined on our key set to index into an array (the hash table).

Hash functions are not perfect, therefore sometimes two keys k_1, k_2 ($k_1 \neq k_2$) hash to the same value. When this happens we have a hash collision.

Open addressing basics

When using open addressing as a collision resolution technique the **key-value pairs are stored in the table (array) itself** as opposed to a data structure like in separate chaining.

This means we need to care a great deal about the size of our hash table how many elements are currently in the table.

$$Load\,factor = \frac{items\,in\,table}{size\,of\,table}$$

Open addressing main idea

The 0(1) constant time behaviour attributed to hash tables assumes the load factor (α) is kept below a certain fixed value. This means once $\alpha > threshold$ we need to grow the table size (ideally exponentially, for instance double).

When we want to insert a key-value pair (k, v) into the hash table we hash the key and obtain an original position for where this key-value pair belongs.

If the position our key hashed to is occupied, we try another position in the hash table by offsetting the current position subject to a **probing sequence** $P(x)$. We keep doing this until an unoccupied slot is found.

There are an infinite amount of probing sequences we can come up with, here are a few:

- **Linear probing:**

  $P(x) = ax + b$ where a and b are constants.

- **Quadratic probing:**

  $P(x) = ax^2 + bx + c$ where a, b, c are constants.

- **Double hashing:**

  $P(k, x) = x \times H_2(k)$, where $H_2(k)$ is a secondary hash function.

- **Pseudo random number generator:**

  $P(k, x) = x \times RNG(H(k), x)$, where RNG is a random number generator function, seeded with $H(k)$.

A general insertion method for open addressing on a table of size N goes as follows:

```
x := 1
keyHash := H(k)
index := keyHash

while table[index] != null:
    index = (keyHash + P(k, x)) mod N
    x = x + 1

insert (k, v) at table[index]
```

Where $H(k)$ is the hash for the key k and $P(k, x)$ is the probing function.

Chaos with cycles

Most randomly selected probing sequences module N will produce a cycle shorter than the table size.

This becomes problematic when we are trying to insert a key-value pair and all the buckets on the cycle are occupied because we will get stuck in a infinite loop, as it can be seen below.

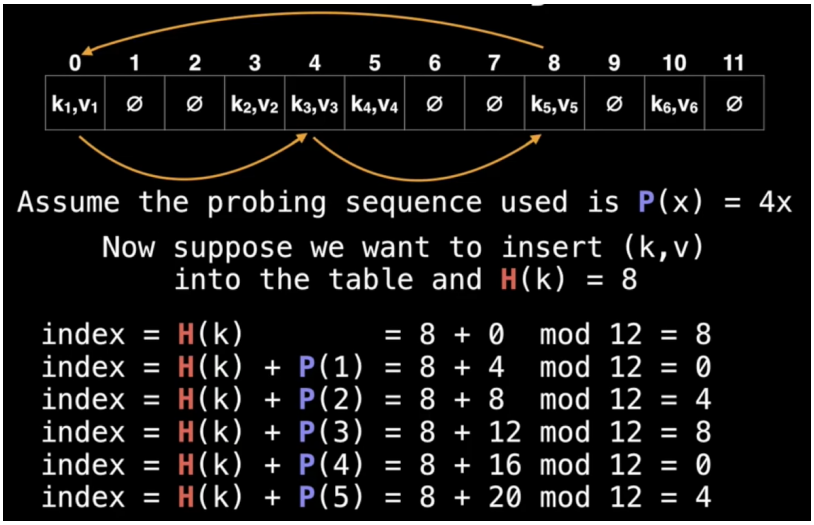


Figure 46:

So, in order to handle probing functions which produce cycles shorter than the table size, in general the consensus is that **we don't handle this issue**, instead we **avoid it altogether by restricting our domain of probing functions to those which produce a cycle of exactly length N**. Note that there are a few exceptions with special properties that can produce shorter cycles.

Techniques such as linear probing, quadratic probing and double hashing are all subject to the issue of causing cycles which is why the probing functions used with these methods are very specific.

Notice that open addressing is very sensitive to the used hashing function and probing function. This is not something we have to worry about if we are using separate chaining as a collision resolution method.

9.4.2 Hash table Linear Probing

Linear Probing is a probing method which probes according to a linear formula: $P(x) = ax + b$, where $a \neq 0$ and b are constants.

However, as we previously saw, not all linear functions are viable because they are unable to produce a cycle of order N. We need a way of handling this, because we may end up having the problem we have seen above.

Which values of the constant a in $P(x) = ax$ produce a full cycle module N?
This happens when a and N are **relatively prime**. Two number are relatively prime if their **Greatest Common Denominator (GCD)** is equal to one. Hence, when $GCD(a, N) = 1$, the probing function $P(x)$ is able to generate a complete cycle and we will always be able to find an empty bucket.

Inserting with LP
Instead of having to look to the GCD all the time, the most common choice of a linear probing function is $P(x) = 1x$, since **no matter the choice of N,** $GCD(N, 1) = 1$.

Once we have reached our threshold of occupancy of the Hash Table, before inserting any new element we have to grow our table. This is usually done in some exponential fashion such as doubling the table size. But, whatever we do, **we have to make sure that $GCD(N, a) = 1$ holds**.

9.4.3 Hash Table Quadratic Probing

Quadratic probing is a probing method which probes according to a quadratic formula: $P(x) = ax^2 + bx + c$, where a, b, c are constants and $a \neq 0$ (otherwise we have linear probing). Note that the constant c is obsolete.

However, as we previously saw, not all quadratic functions are viable because they are unable to produce a cycle of order N. We will need some way to handle this.

There are numerous ways of picking a quadratic probing function, but these are the three most popular approaches:

- Let $P(x) = x^2$, keep the table size a prime number > 3 and also keep $\alpha \leq 1/2$.
- Let $P(x) = (x^2 + x)/2$ and keep the table size a power of two.
- Let $P(x) = (-1^x) * x^2$ and keep the table size a prime N where $N = 3 \, mod \, 4$

9.4.4 Hash Table Double Hashing

Double Hashing is a probing method which probes according to a constant multiple of another hash function: $P(k, x) = x \times H_2(k)$, where $H_2(k)$ is a second hash function.

It is very important to remember that $H_2(k)$ must hash the same type of keys as $H_1(k)$.

Notice that double hashing reduces to linear probing, except that the constant is unknown until runtime.

As always, we have to avoid ending up on a infinite loop.

So, to fix the issue of cycles, we have to **pick the table size to be a prime number and also compute the value of δ**:

$\delta = H_2(k) \, mod \, N$

If $\delta = 0$, then we are guaranteed to be stuck in a cycle, so when this happens we have to set $\delta = 1$.

Notice that $1 \leq < N$ and $GCD(\delta, N) = 1$ since N is prime. Hence, with these conditions, we know that modulo N the sequence

$H_1(k), H_1(k) + 1\delta, H_1(k) + 2\delta, H_1(k) + 3\delta, H_1(k) + 4\delta + ...$ is certain to have order N.

How do we construct our secondary Hash Function?

Suppose the key k has type T. Whenever we want to use double hashing as a collision resolution method, we need to fabricate a new function $H_2(k)$ that knows how to hash keys of type T.

The keys we need to hash are always composed of the same fundamental building blocks. In particular: integers, strings, real numbers, fixed length vectors, etc..

There are many well known high quality hash functions for these fundamental data types. Hence, we can use and combine them to construct our function $H_2(k)$.

Frequently, the hash functions selected to compose $H_2(k)$ are picked from a pool of hash functions called **universal hash functions** which generally operate on one fundamental data type.

When resizing our hash table using linear probing, one strategy is to compute $2N$ and find **the next prime above this value**. So, for instance if we have $N = 7$, when doubling it we get 14, and the next prime number above it is 17, so we would use this value.

9.5 Open Addressing Removal

We can't remove the elements naively. For instance, imagine that we insert 3 elements, and all of them have the same hash value. We will have to probe once the second element and twice the third.

Then, if we remove naively the second element we will take it out and that position will get empty, so it will have a *null* value.

So, now if we try to find the third element, we will see that it is not in its hash location, so we probe for the first time, and we see that we get a *null*, so we must conclude that the key k_3 does not exist in the hash table otherwise we would have found it before reaching a *null* position.

The solution is to place a **unique marker** called **tombstone** instead of null to indicate that a (k, v) pair has been deleted and that the bucket should be skipped during search. And so, we will know that we should continue probing.

We have a lot of tombstones cluttering our HT, how do we get rid of them?

Tombstones count as filled slots in the HT, so they increase the load factor and will be removed when the hash table is resized. Additionally, when inserting a new (k, v) pair, we can replace buckets with tombstones with the new key-value pair.

An optimization we can do is to replace the earliest tombstone encountered with the value we did a lookup for. The next time we lookup the key it will be found much faster. This is called **lazy deletion**.