A comparison between functional and object oriented programming approaches in JavaScript

Kim Svensson Sand   Tord Eliasson

April 20, 2017

Contents

1 Abstract

1.1 Keywords

2 List of figures

3 List of tables

4 List of symbols

5 Abbreviations

ES6 = ECMAScript 2015

FP = Functional Programming

OOP = Object-Oriented Programming

6 Introduction

We will compare two different programming paradigms, Functional programming and Object oriented programming, both written in JavaScript. We will compare these by writing four different algorithms in each paradigm and then compare runtime, memory usage, time of development and lines of code. The algorithms we will implement is a binary tree, shellsort, tower of hanoi and dijkstra's algorithm.

7 Background and Related work

Object-oriented or imperative languages are dominating with languages such as C, C++, Java and C#. However, functional languages seem to be of interest in the industry for handling big data and concurrency with languages such as Scala, Erlang or Haskell [1, 2, 3]. Some are also arguing that a more functional approach will give you less code, more readable code, code that is easier to maintain and easier to test, and that learning it will give you better experience as a programmer [1, 4].

Here we will describe the functional programming paradigm, FP, and also the object oriented programming paradigm, OOP.

7.1 Functional programming

Functional programming, FP, is a programming paradigm that at its core is based on lambda-calculus [5]. Programs are constructed using functions and by avoiding changing the state. By not modifying the state side effects are avoided. Computation is done by changing the environment, rewriting the functions, rather than changing variables. Multiple functions can be composed into larger and more complex functions, and should be reduced to its simplest state following mathematical rules. The following are concepts used in FP.

Lazy evaluation - Lazy evaluation is when a value is not calculated until it is needed [6].

Static type checking - Pure functional languages usually have static type checking [5]. This means that variables are of a certain type, for example int or char. In such languages it is not allowed to use functions with the wrong types. So if there is for example a function taking an int as parameter it is illegal to call that function with a String as its parameter. In dynamic type checked-languages this would be legal, since variables are not specified as types, and might cause an unexpected error. For example in JavaScript:

```
//JavaScript has dynamic type checking
function double(nr) {
return nr * 2;
}

var word = "string";
double(word); //Will return NaN but still continue running
```

A similar program in golang [7], that is a statically type checked language:

```
func main() {
  var word string = "string"
  fmt.Printf("%d", double(word)) //Error: cannot use word (type string) as type int in argument to double
}

func double(nr int) int {
  return nr * 2
}
```

This will give an error when compiling.

There are however pure functional languages with dynamic type checking, such as Clean [8]. There are also languages that are mostly functional with dynamic type checking, such as Common Lisp [9] or Scheme [10] [11].

Side effects - Side effects are for example changing a variable or any interaction outside the function [1]. This is avoided in functional programming since it may result in incorrect and unexpected behaviour.

Pure functions - A pure functions always returns the same result, given the same input, and does not have side effects [1]. See the following example:

```
var sum = 0;

//Impure
function add(a, b) {
  sum = a + b;
}

//Pure
function add(a, b) {
  var tmp = a + b;
  return tmp;
}
```

Higher order functions - Higher order functions are an important concept in functional programming [6]. First class functions mean that functions are treated as values, which means that they can be stored in variables, stored in arrays or created if needed. A higher order function is a first class function that either takes a function as a parameter, returns a function as a result or both. This makes it possible to compose larger and more complex functions.

```
function add(a, b) {
  return a + b;
}

//Functions can be placed in variables.
var thisFunc = add;

//And also put in other variables.
var sameFunc = thisFunc;
sameFunc(1, 2); //Will give the output 3.

//Functions can also be used as parameters or return
    values.
function applyFunc(f, a, b) {
  return f(a, b);
}

applyFunc(function(a, b) {
```

```
  return a * b;
}, 3, 2); //Will give output of 6

//Returns a function that returns a string
function getStringCreator(category, unit) {
  return function(value) {
    return category + ': ' + value + unit;
  }
}

var weightStringCreator = getStringCreator('Weight', 'kg');
weightStringCreator(5); //Outputs "Weight: 5kg"
```

Recursion - Recursive functions are functions that call themselves and are used as loops [6]. It is important in functional programming since it can hide mutable state and also implement laziness. In functional programming recursion is used rather than loops.

```
//Adds all numbers from start to end with a loop
function iterativeAdd(start, end) {
  var sum = 0;
  while(start <= end) {
    sum += start;
    start++;
  }

  return sum;
}

//Adds all numbers from start to end recursively
function recursiveAdd(start, end) {
  if (start == end) {
    return end;
  }
  else {
    return start + recursiveAdd(start + 1, end);
  }
}
iterativeAdd(1, 6); //Outputs 21
recursiveAdd(1, 6); //Also outputs 21
```

Currying - Currying means a function can be called with fewer arguments than it expects and it will return a function that takes the remaining arguments [1].

```
//Returns a new function that takes the remaining argument or the sum of a and b if both are provided.
function addWithCurrying(a, b) {
  if(b) {
```

```
    return a + b;
  }
  else {
    return function(b) {
      return a + b;
    }
  }
}

var curryAdd = addWithCurrying(4);
curryAdd(6); //Outputs 10
addWithCurrying(4, 6); //Also outputs 10
```

Immutable data structures - In functional programming mutations, that are side effects, are avoided [6]. Hidden side effects can result in a chains of unpredicted behaviour in large systems. Instead of mutating the data itself a local copy is mutated and returned as a result, as seen in the pure functions example.

7.2 Object-oriented programming

Object-oriented programming is a programming paradigm which is built around "objects". These objects may contain data, in form of fields, often referred as attributes and code in form of procedures, often referred to as methods [26].

These objects are created by the programmer to represent something with the help of its attributes and methods. What kind of variables and functions an object should contain is defined in a class, which works like a blueprint for the object. For example, a class can represent an employee. An employee has the attributes assignment and salary. The employee also has a method for doing work. Then the employee has to work somewhere, so we create another object for a company where our employee can work. The company has the attributes income and number of employees. It also has methods to hire employees and fire employees. Since there are more employees who works at this company, we can add more employees by creating new objects of the information in employee class.

By building objects together like this, you can build programs by the object oriented paradigm.

Class - A class is a model for a set of objects, which the object oriented paradigm is built around[35]. The class establishes what the object will contain, for example variables and functions, and signatures and visibility of these. To create a object of any kind, a class must be present.

```
//Creates a class Animal with the properties name and age and a function for logging the properties to the screen
class Animal {
  constructor (name, age) {
    this.name = name;
    this.age = age;
  }

  logAnimal() {
    console.log('Name:␣' + this.name + '\nAge:␣' + this.age);
  }
}

var animal = new Animal('Buster', '9');
animal.logAnimal();
//Outputs Name: Buster
//Age: 9
```

Object - An object is a capsule that contains the variables and functions established in the class. While objects created from the same class contains the same variables and functions, since the information it contains is handled specifically for every object, the information in the class can vary much. All data and functions can be accessible from outside the object.

Inheritance - When an object acquires all properties and functionality of another object. [57] it is called inheritance. This provides code reusability.

```
//Based upon the Animal class with an added property race
//The original logAnimal() is overidden so the new property is also logged to the screen.
class Dog extends Animal {
  constructor(name, age, race) {
    super(name, age);
    this.race = race;
  }

  logAnimal() {
    super.logAnimal();
    console.log('Race:␣' + this.race);
  }
}

var dog = new Dog('Buster', '9', 'Shitzu');
dog.logAnimal();
//Outputs Name: Buster
//Age: 9
//Race: Shitzu
```

Encapsulation - Encapsulation can be used to refer to two different things [35, 57]. A mechanism to restrict direct access to some object components and the language construct that facilitates the bundling of data with methods. The access part is done by making the different parts of an object public or private and the bundling is made with objects.

Polymorphism - When a task is used in different ways it is called polymorphism. [35, 57]This is achieved with the help of overloading and overriding.

Overloading - Refers to creating a function with the same name as another function, often very similar, with either different types of variables in the parameters or different number of parameters[57].

Overriding - Refers to overwrite a function written in a superclass to make it do something else than first intended, without changing the superclass. [57]

7.3 JavaScript

JavaScript is a prototype based with first class functions. This makes it a multi-paradigm language with support for Object-oriented, imperative and functional paradigms. Since we are comparing FP and OOP we will ignore the imperative paradigm. How does the support of FP and OOP look in JavaScript, how many of the listed concepts above is supported?

7.3.1 Functional programming

JavaScript is not a functional language, but it is possible to write functional code with it [1]. There are also libraries that makes functional programming in JavaScript easier, such as Underscore.js [5]. However, we will use ECMA 2015, ES6, that already provide many of the functions provided by Underscore.js. This is also to use the same environment for our different implementations in this experiment.

In JavaScript it is possible to treat functions as any other variable, pass them as function parameters or store them in arrays, so called first class functions [1]. There are also higher order functions such as map(), filter() and reduce() that might replace loops [2]:

map() - Calls a provided function on every element in an array and returns an array with the outputs [27].

filter() - Returns a new array with all the elements that passes a test in a provided function.

reduce() - Reduces a array to a single value.

However there is no automatic currying or immutable data structures in JavaScript. JavaScript is also dynamically type checked. All of these can be added to JavaScript with libraries.

7.3.2 Object oriented programming

JavaScript has always had support for OOP. But with ES6, OOP starts to look more like classical OOP languages like for example Java [30].

JavaScript do not fully support encapsulation, since you can't make variables and functions private or public. Otherwise there is full support for OOP in JavaScript.

As a workaround to this, many programmers add an underscore to the variable or function before the name of it. This underscore symbolizes that it should be handled as a private instead of a public.

7.4 Related work

In "Curse of the excluded middle" [3] Erik Meijer argues with multiple examples that the industry should not focus on a combination between functional and objected oriented methods to counter handling big data with concurrency and parallelism. He concludes that it is not good enough to avoid side effects in imperative or object oriented languages. It is also not good enough to try to ignore side effects in pure functional languages. Instead he thinks that people should either accept side effects or think more seriously about using the functional paradigm.

In "Functional at scale" [4] by Marius Eriksen he is explaining why Twitter uses methods from functional programming to handle concurrent events that arises in large distributed systems in cloud environments. In functional programming it is possible build complex parts out of simple building blocks, thus making systems more modular. He concludes that the functional paradigm has multiple tools for handling the complexity present in modern software.

Eriksson and Ärleryd are looking at how to use functional practises when developing front end applications by taking inspiration from Elm in their master's thesis [11]. They have researched each practise in Elm to see if it is possible to use these practises in JavaScript together with tools and libraries. Their conclusion was that it is possible to replicate functional practises from Elm in

JavaScript, but that they prefer working with Elm. In JavaScript multiple libraries had to be used to use the same practises. They also concluded that even though functional programming is not widely used within the industry, functional practises can still be used in all projects.

In "Improving Testability and Reuse by Transitioning to Functional Programming" [12], Benton and Radziwill state that functional programming is better suited for test driven development (TDD) and concludes that a shift toward the functional paradigm benefits reuse and testability of cloud-based applications.

Alic, Omanovic and Giedrimas has made a comparative analysis of functional and object-oriented programming languages[15]. They have compared four languages, C#, F#, Java and Haskell based on performance, runtime and memory usage. Their conclusion is that Java is the fastest while Haskell uses much less memory, and that programming paradigms should be combined to increase execution efficiency.

Dobre and Xhafa writes in "Parallel Programming Paradigms and Frameworks in Big Data Era" that we now are in a big data era [47]. They also review different frameworks, programming paradigms and more in a big data perspective. Around paradigms they state, "functional programming is actually considered today to be the most prominent programming paradigm, as it allows actually more flexibility in defining big data distributed processing workflows."

In "Comparing programming paradigms: an evaluation of functional and object-oriented programs" R. Harrison, et al. compares the quality of code in functional and object oriented programming [51]. To compare these, they use C++ and SML. While their discussion states that they would probably use OOP, since C++ is a better language, the standard list functions were of great help, the debugging was better and reusability was much higher in SML.

Guido Salvaneschi, et al. have conducted an empirical evaluation of the impact of Reactive Programming (RP), with functional programming concepts, on program comprehension [48]. Their experiment involved 127 subjects and the results suggests that RP is better for program comprehension when compared with OOP. They conclude that with RP the subjects produced more correct code without the need of more time, and also that the comprehension of RP programs is less tied to programming skills.

In "Using Functional Programming within an Industrial Product Group: Perspectives and Perceptions" [49] David Scott et al. presents a case-study of using FP in the multiparadigm language OCaml in a large product development team. They found that the team's project was a success even though there were some drawbacks to using OCaml, such as lack of tool support. The engineers believed that OCaml enabled them to be more productive than if they would have used one of the mainstream languages, such as C++ or Python.

8 Method

In this experiment we will implement four different algorithms. We have chosen algorithms that are well known, so that our focus will be on the different implementations, rather than on how the algorithms work.

8.1 Algorithms

We will implement tree search algorithms, the shellsort algorithm, the tower of hanoi-algorithm and Dijkstra's algorithm. In the search tree we will implement tree traversal, which is a recursive algorithm, as is the tower of hanoi algorithm. These algorithms fit well for FP since it uses recursion. The implementation of the binary tree structure can however be implemented by using classes and objects, that is used in OOP. Shellsort uses state and iteration which is used in OOP and avoided in FP. Dijkstra's algorithm also uses state and iteration, and the graph can also be implemented using classes and objects.

We chose algorithms that use both recursion and iteration to not give advantage to any paradigm. When implementing these algorithms and data structures we can also make use of OOP methods, such as classes and objects. The algorithms' purpose are also different to avoid for example implementing four different sorting algorithms.

8.1.1 Tree search algorithms

A binary tree is a tree consisting of nodes, where a node can have a maximum of two children [38]. These children can be described as the left and the right subtree and the parent is the root. The property that differs a binary search tree from a standard binary tree, is the order of the nodes. In a binary tree the order can be undecided, but in a search tree the nodes are stored in an order based on some property. For example in our binary search tree the left subtree of a root will contain smaller numbers than the root, and the right subtree will contain larger numbers, see example in figure 1. Using tree traversal it is possible to find certain nodes or get a list of sorted objects. Our binary tree will contain random numbers and the functions:

findNode(comparable, rootNode) - Finds a specific key in the tree.

inOrderTraversal(root) - Returns an array of all numbers in the tree, sorted smallest to biggest.

insert(comparable, rootNode) - Inserts number into the tree. Returns true or false depending on success. If comparable is already in the tree, false should be returned. Note that in the functional implementation this function will return the resulting tree instead of mutating the tree and returning true or false.

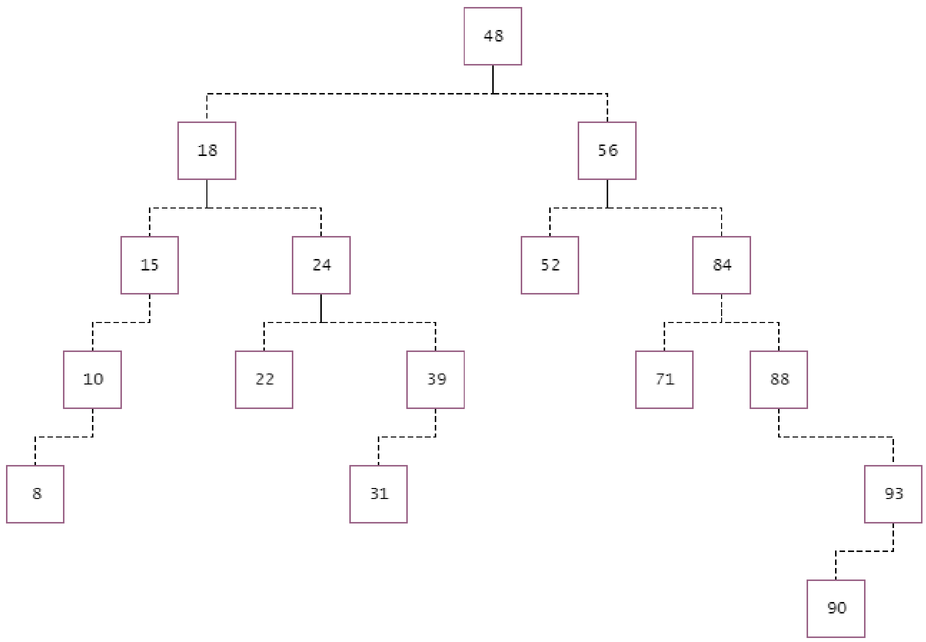

Figure 1: Binary tree example

Algorithm descriptions:

algorithm findNode(comparable, root)

```
if root is undefined then
  return null
else if comparable is equal to comparable in root then
  return root
else if comparable is larger than comparable in root then
  return findNode(comparable, rightSubtree in root)
else if comparable is smaller than comparable in root then then
  return findNode(comparable, leftSubtree in root)
end if
```

algorithm inOrderTraversal(root)

```
set array to empty array
if comparable in root is undefined then
  return array
else then
  add inOrderTraversal(leftSubtree in root) to end of array
  add comparable in root at end of array
  add inOrderTraversal(rightSubtree in root) at end of array
  return array
end if
```

Running this in the tree in figure 1 would return the array [8, 10, 15, 18, 24, 22, 31, 39, 48, 52, 56, 71, 84, 88, 90, 93].

algorithm insert(comparable, root)

```
if comparable is equal to comparable in root then
    return false
else if comparable is larger than comparable in root then
  if rightSubtree in root is undefined
    create newNode
    set comparable in newNode to comparable
    set rightSubtree of root to newNode
    return true
  else then
      return insert(comparable, rightSubtree in root)
  end if
else if comparable is smaller than comparable in root then
  if leftSubtree in root is undefined then
create newNode
    set comparable in newNode to comparable
    set leftSubtree in root to newNode
    return true
  else then
    return insert(comparable, leftSubtree in root)
  end if
end if
```

8.1.2 Shellsort algorithm

The shellsort algorithm is named after its creator Donald Shell [30]. It was one of the first algorithms to break the quadratic time barrier. The algorithm sorts by sorting items using insertion sort with a gap. For each run the gap is decreased until the gap is 1 and the items are sorted. How the gap is decreased is decided with a gap sequence. Different gap sequences gives shellsort a different worst-case running time.

We will use one of Sedgewick's gap sequences that has one of the fastest worst-case running times $O(n^3/4)$.

The sequence is

$\{1, 5, 19, 41, 109\}$, where the terms are of the form

$4^k - 3 * 2^k + 1$, or

$9 * 4^k - 9 * 2^k + 1$

In our implementation the sequence is put in an array and not calculated during execution, since we do not want to get different results between our implementation because of the calculation of the gap sequence.

This algorithm will sort an array filled with randomized numbers. Our implementation uses this algorithm in a function:

shellsort(array) - Takes an unsorted array as an input and returns a sorted copy of the array.

Algorithm description:

algorithm algorithm shellsort(array of size n)

```
set sortedArray to array
set gapSequence to Sedgewick's␣gap␣sequence.
set␣currentGapIndex␣to␣0
set␣currentGap␣to␣the␣largest␣gap␣i␣gapSequence␣where␣gap␣is␣smaller␣than␣n␣divided␣by␣2
set␣currentGapIndex␣to␣the␣index␣of␣currentGap␣in␣gapSequence
while␣currentGap␣is␣larger␣than␣0␣do
␣␣for␣i␣=␣currentGap␣to␣n
␣␣␣␣set␣currentValue␣to␣array[i]
␣␣␣␣set␣currentIndex␣to␣i
␣␣␣␣while␣currentIndex␣-␣currentGap␣is␣larger␣or␣equal␣to␣0␣and␣sortedArray[currentIndex␣-␣currentGap]␣is␣larger␣than␣currentValue␣do
␣␣␣␣␣␣set␣sortedArray[currentIndex]␣to␣sortedArray[currentIndex␣-␣currentGap]
␣␣␣␣␣␣set␣currentIndex␣to␣currentIndex␣-␣currentGap
␣␣␣␣end␣while
␣␣␣␣set␣sortedArray[currentIndex]␣to␣currentValue
␣␣end␣for
␣␣set␣currentGapIndex␣to␣currentGapIndex␣-␣1
```

```
␣␣set␣currentGap␣to␣gapSequence[currentGapIndex]
end␣while
return␣sortedArray
```

8.1.3 The Tower of Hanoi algorithm

The Tower of Hanoi is a game invented by mathematician douard Lucas in 1883 [39]. The game consists of three pegs and a number of disks stacked in decreasing order on one of the pegs, see figure 2. The goal is to move the tower from one peg to another by moving one disk at a the time to one of the other pegs. A disk can not be placed on a peg on top a smaller disk.

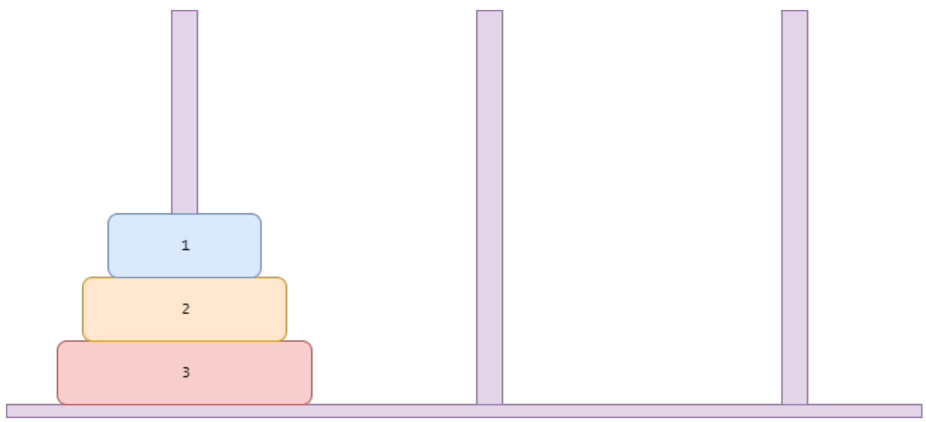

Figure 2: Tower of hanoi image

This algorithm will move a tower from one peg to another in the function:

hanoi(tower, start, dest, aux) - Moves tower from start to dest following the rules of the tower of hanoi problem. Returns the start, dest and aux pegs with the repositioned tower.

Algorithm description:

algorithm hanoi(tower, start, dest, aux)

```
(*Pseudo code based on that tower is the number of the
    largest disk, where 1 is the smallest disk in the
    tower.*)
if tower is equal to 1
  move tower from start to dest
else
```

```
  hanoi(tower - 1, start, aux, dest)
  move tower from start to dest
  hanoi(tower - 1, aux, dest, start)
end if
```

8.1.4 Dijkstra's algorithm

Dijkstra's algorithm is an algorithm for finding the shortest path in a graph consisting of a number of nodes connected by edges [30], where the weight of the edges is known, see figure 3. The algorithm will find the shortest path from the start node to the end node in a graph. It is initiated by:

1. setting the distance to the start node to 0.
2. setting the distance to all other nodes to infinity.
3. mark all nodes as unvisited.

The algorithm will then find the shortest path using the following steps:

1. Set current node to the node with the smallest distance that has not already been visited.
2. For all neighbors to current node that has not already been visited, check if their distance is smaller than the distance of current node + the distance to the neighbor. If so, update the distance of the neighbor.
3. Mark current node as visited.
4. Repeat until all nodes have been visited or the end node has been visited.

See the result in figure 4.

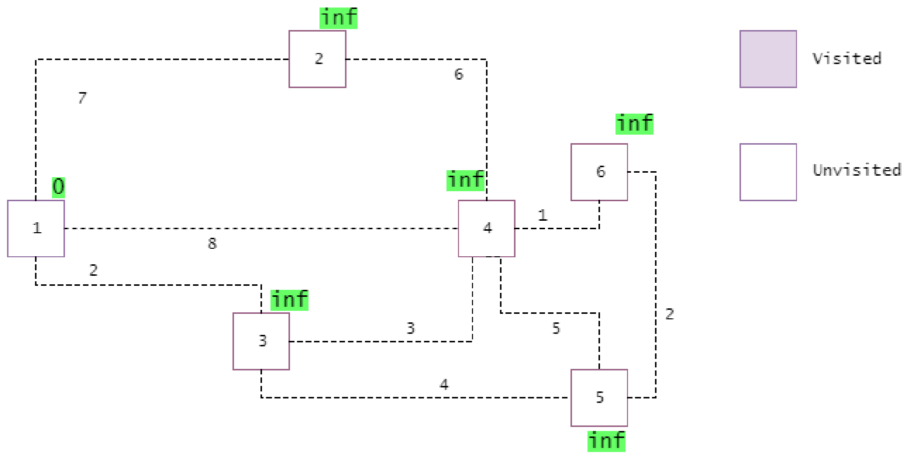

Figure 3: After initiation of Dijkstra's algorithm

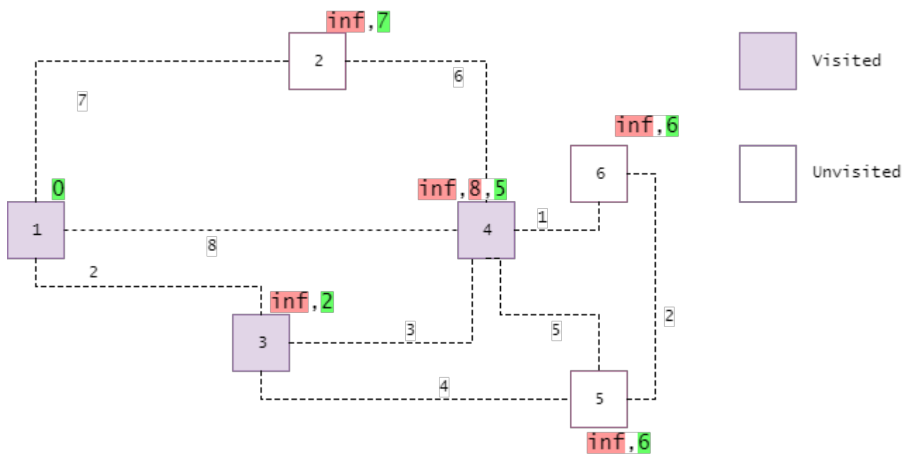

Figure 4: After three nodes have been visited with Dijkstra's algorithm

This algorithm will be used in a function:

dijkstras(graph, startNode, endNode) - That takes graph, startNode and endNode and returns an array with the shortest path from startNode to destNode.

Algorithm description:

algorithm dijkstras(graph, startNode, endNode)

```
for each node in graph
  set dist[node] to infinity
  set path[node] to undefined
end for
set dist[startNode] to 0
set unvisitedNodes to nodes in graph
while unvisitedNodes is not empty or endNode is not in unvisitedNodes do
  set current to node where dist[node] is smallest and node is in unvisitedNodes
  remove current from unvisitedNodes
    for each neighbor of current where neighbor is in unvisited do
        set temp to dist[current] + weight of edge in graph, where edge is from current to neighbor
    if temp is smaller than dist[neighbor] then
      set dist[neighbor] to temp
      set path[neighbor] to path[current] + current)
    end if
  end for
end while
return path[endNode]
```

8.2 Environment

We will implement the algorithms using Atom, a text editor, and compile and run our tests an1d our code with Node.js. To manage our dependencies we are using npm. For automation we are using Grunt.

(Add image describing the build of our files)

Since node.js does not support ES6-modules we are using babel to transpile our ES6-modules to node-modules that are supported. We are doing this since we have prior experience with ES6 modules and not with node-modules. Since node.js support almost everything else in ES6 we will only be transpiling the modules, leaving our code in an ES6 standard.

To measure memory and run time we are using node.js process, that is a global and therefore always available within node.js applications [50]. We are using the following functions:

process.hrtime(time) - Returns the current high resolution time in a [seconds, nanoseconds] tuple Array. If the optional time-parameter, an earlier hrtime, is used, it will return the difference between that time and the current time.

process.memoryUsage() - Returns an object describing the memory usage of the Node.js process measured in bytes.

8.3 Testing

Our code will be tested through code reviews and unit testing.

8.3.1 Code reviews

We will review each other's implementations and our implementations will also be reviewed by a third party. The reviews should help us to find bugs and also to confirm that we have used FP or OOP methods.

8.3.2 Unit testing

Unit testing will be done using the JavaScript libraries Mocha [45] for writing tests, Chai [46] for evaluating expressions, and Karma [44] for automated tests.

Our implementations will have to pass the tests in appendix A to be accepted as done.

8.4 Implementation

We will proceed from a template with the following structure:

```
|---dist
|   \---<Files generated from babel>
|---src
|   |---js
|        |---<function and class files>
|   \---index.js
|---es6-test
|   \---<Test files>
|---test
|   \---<Test files generated from babel>
| Gruntfile.js
| package.json
```

To run tests use the command:

npm test - Will transpile the test files in the es6-test-folder, place the generated files in the test-folder and run the tests.

To run the algorithm use the command:

npm start - Will transpile the JavaScript-files in src-folder, place the generated files in the dist-folder and run index.js in the dist-folder.

For an implementation of an algorithm to be considered done the following has to be implemented and provided:

- Tests for the algorithm that are defined in appendix A placed in (projectName)/es6-test
- A complete implementation of the algorithm that has passed the tests placed in (projectName)/src/js
- A measurement implementation that must run the algorithm and measure time and memory usage using node.js process.
- A timelog describing how long the implementation took.

8.4.1 Guidelines for writing the different paradigm

Functional programming guidelines:

- Do not use classes
- Treat functions as variables
- Compose functions to build programs
- Use already provided pure functions when possible
- Write pure functions
- Do not change variables
- Use recursion instead of iteration

Object-Oriented Programming guidelines:

- Use classes and objects to build programs
- Use inheritance when possible
- Use encapsulation
- Each class should have only one job
- Prefer iteration over recursion

References

[1] drboolean, "Gitbook, proffesor frisby's mostly adequate guide to functional programming," 2017.

[2] M. Eriksen, "Functional at scale," *Queue*, vol. 14, pp. 60:39–60:55, Aug. 2016.

[3] N. Eriksson and C. Ärleryd, "Elmulating javascript," Master's thesis, Linkping University, Human-Centered systems, 2016.

[4] E. Meijer, "The curse of the excluded middle," *Commun. ACM*, vol. 57, pp. 50–55, June 2014.

[5] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science, London: Springer, 2010.

[6] M. Fogus, *Functional JavaScript*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., first ed., may 2013.

[7] "The go programming language."

[8] "Clean," dec 2016.

[9] "Common lisp," may 2015.

[10] "Scheme reports," jan 2015.

[11] "Comparison of functional programming languages," dec 2016.

A Appendix

A.1 Test cases

ID	T1
Algorithm	Binary search tree algorithms
Function	insert(comparable, rootNode)
Description	Inserted items should be added at the correct place in the tree. If number already exists in the binary search tree, it should not be inserted.
Preconditions	See figure 5
ID	T1.1
Input	9, node(13)
Expected values	FP: None, OOP: See figure 5
Expected output	FP: See figure 5, OOP: false
ID	T1.2
Input	8, node(13)
Expected values	FP: None, OOP: See figure 6
Expected output	FP: See figure 6, OOP: true
ID	T1.3
Input	1, node(13)
Expected values	FP: None, OOP: See figure 7
Expected output	FP: See figure 7, OOP: true
ID	T1.2
Input	33, node(13)
Expected values	FP: None, OOP: See figure 8
Expected output	FP: See figure 8, OOP: true

ID	T2
Algorithm	Binary search tree algorithms
Function	findNode(comparable, rootNode)
Description	findNode should return the node containing comparable. If comparable is not in the tree undefined should be returned.
Preconditions	See figure 5
ID	T2.1
Input	5, node(13)
Expected values	None
Expected output	undefined
ID	T2.2
Input	13, node(13)
Expected values	None
Expected output	node(13)
ID	T2.3
Input	2, node(13)
Expected values	None
Expected output	node(2)
ID	T2.4
Input	32, node(13)
Expected values	None
Expected output	node(32)
ID	T2.5
Input	20, node(13)
Expected values	None
Expected output	node(20)

ID	T3
Algorithm	Binary search tree algorithms
Function	inOrderTraversal()
Description	Should return a sorted array of the elements in the tree. If the tree is it should return an empty array.
ID	T3.1
Preconditions	See figure 5
Input	FP: node(13), OOP: none
Expected values	None
Expected output	[2, 3, 6, 7, 9, 13, 16, 20, 24, 32]
ID	T3.2
Preconditions	Empty tree
Input	FP: none, OOP: none
Expected values	None
Expected output	[]

ID	T4
Algorithm	Shellsort
Function	shellsort(array)
Description	If an array of numbers is input a sorted array should be returned. If an empty array is input an empty array should be returned.
Preconditions	None
ID	T4.1
Input	[]
Expected values	None
Expected output	[]
ID	T4.2
Input	[9, 8, 1, 15, 3, 4, 11, 2, 7, 6]
Expected values	None
Expected output	[9, 8, 1, 15, 3, 4, 11, 2, 7, 6]

ID	T5
Algorithm	The Tower of Hanoi
Function	hanoi(tower, start, dest, aux)
Description	Should return start, dest and aux pegs with moved tower. If tower has zero discs it should return empty start, dest and aux. The algorithm should take $2^n - 1$ moves.
Preconditions	None
ID	T5.1
Input	tower size 8, peg1 with tower, peg3 empty, peg2 empty
Expected values	FP: none, OOP: nrOfMoves=255
Expected output	peg1 empty, peg2 empty, peg3 with tower
ID	T5.2
Input	tower size 0, peg1 empty, peg3 empty, peg2 empty
Expected values	FP: none, OOP: nrOfMoves=0
Expected output	peg1 empty, peg2 empty, peg3 empty

ID	T6
Algorithm	Dijksta's algorithm
Function	dijkstras(graph, startNode, endNode)
Description	Should return the shortest path from startNode to endNode. If startNode is same as endNode it should return empty path. If graph is empty it should return empy path.
Preconditions	None
ID	T6.1
Input	See figure 9, node1, node6
Expected values	None
Expected output	See figure 10
ID	T6.2
Input	See figure 9, node1, node4
Expected values	None
Expected output	See figure 11
ID	T6.3
Input	See figure 9, node2, node5
Expected values	None
Expected output	See figure 12
ID	T6.4
Input	See figure 9, node1, node1
Expected values	None
Expected output	empty path
ID	T6.5
Input	nodes=[], edges[], noNode, noNode
Expected values	None
Expected output	empty path

A.2 Test case images

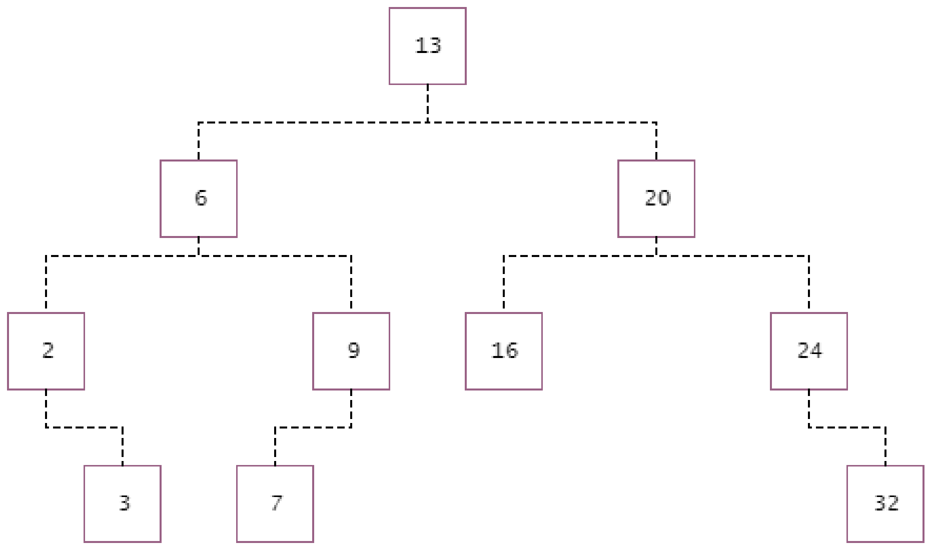

Figure 5: Input tree

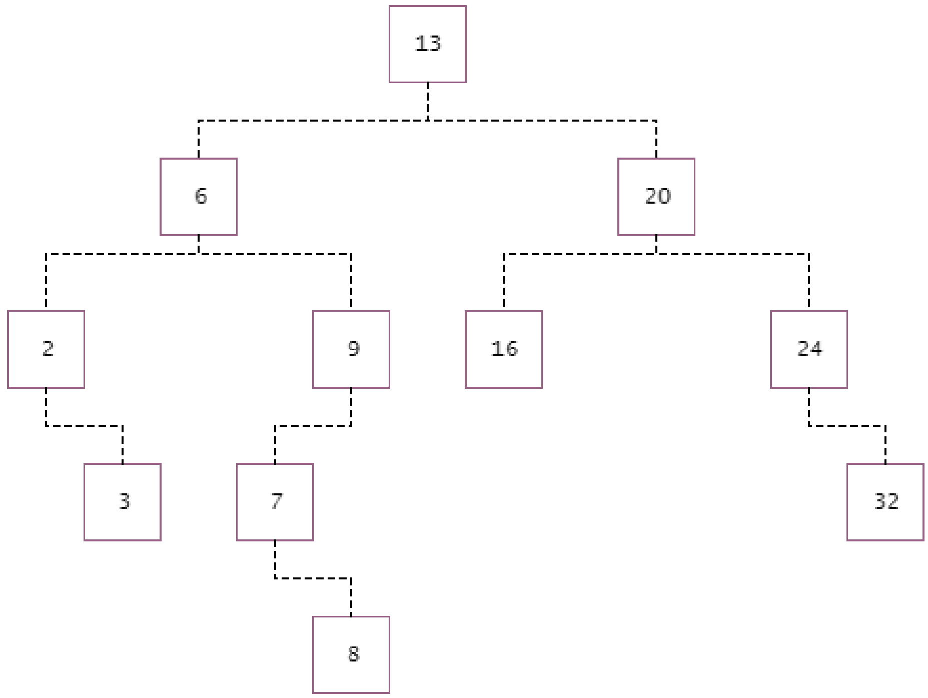

Figure 6: Output tree after inserting 8

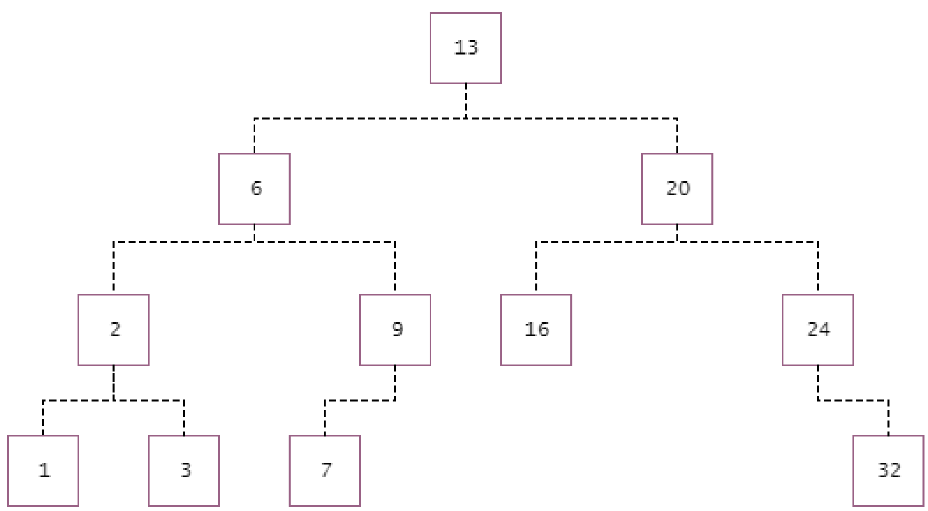

Figure 7: Output tree after inserting 1

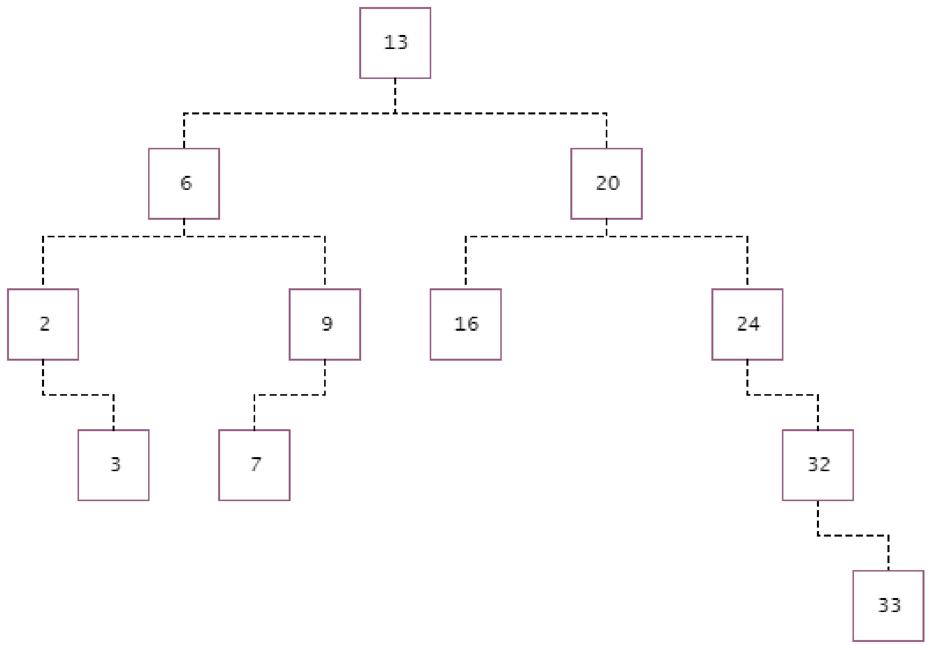

Figure 8: Output tree after inserting 33

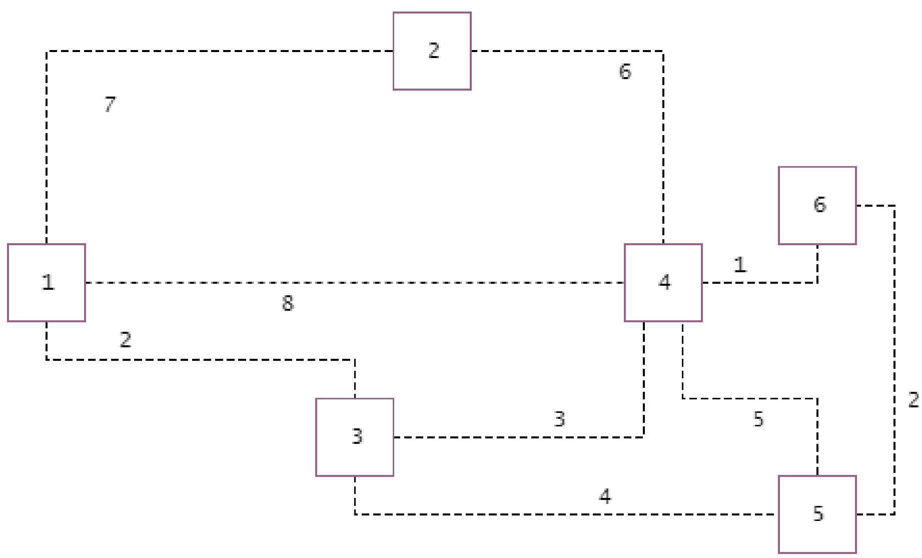

Figure 9: Input graph

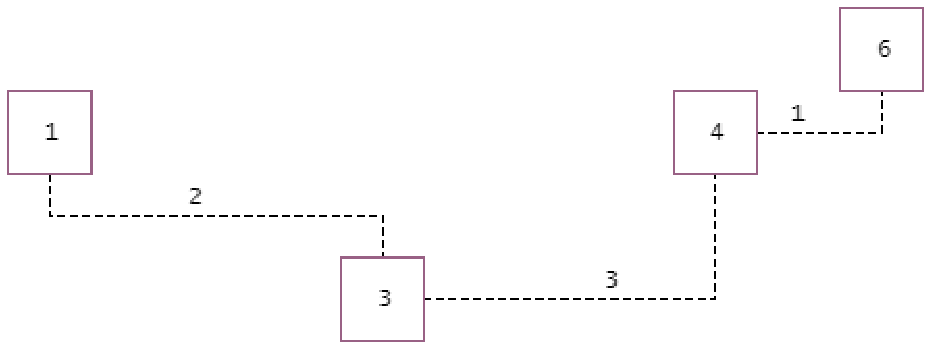

Figure 10: Output graph 1

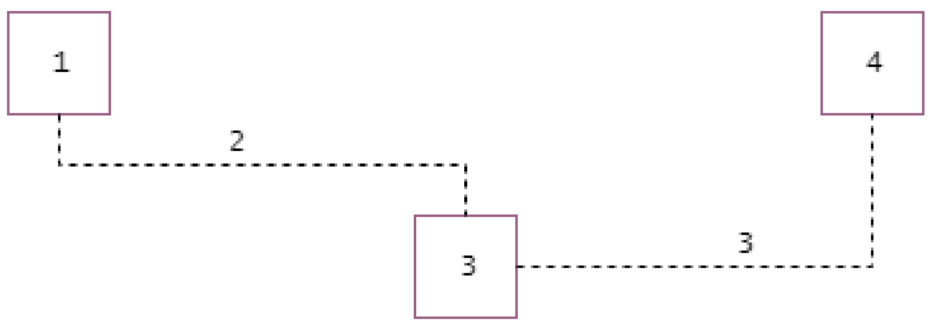

Figure 11: Output graph 2

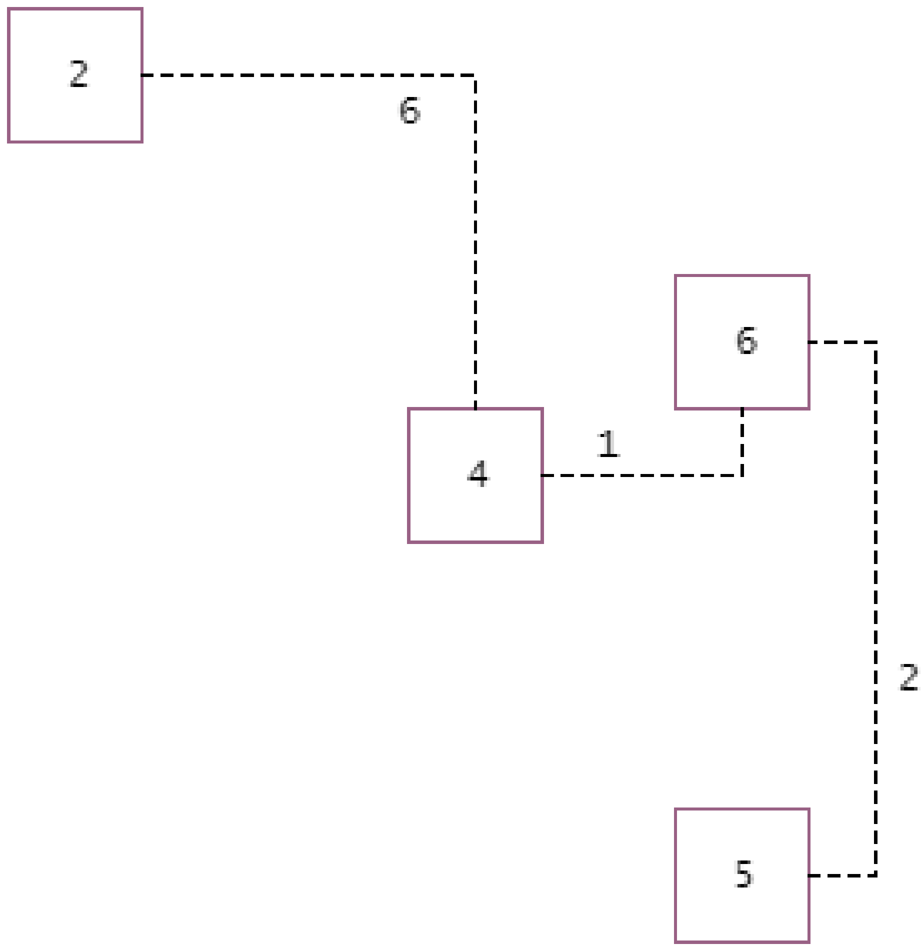

Figure 12: Output graph 3