A comparison of functional and object-oriented programming paradigms in JavaScript

Kim Svensson Sand and Tord Eliasson

June 6, 2017

Faculty of Computing

Blekinge Institute of Technology

SE-371 79 Karlskrona Sweden

Abstract

There are multiple programming paradigms that have their own set rules for how code should be written. Programming languages utilize one or multiple of these paradigms. In this thesis, we will compare object-oriented programming, that is the most used today with languages such as C++ and Java, and functional programming. Functional programming was introduced in the 1950's but suffered from performance issues, and has not been used much except for in the academic world. However, for its ability to handle concurrency and big data, functional programming is of interest in the industry again with languages such as Scala. In functional programming side effects, any interaction outside of the function, are avoided as well as changing and saving state.

To compare these paradigms we have chosen four different algorithms, which both of us have implemented twice, once according to object-oriented programming and once according to functional programming. All algorithms were implemented JavaScript. JavaScript is a multiparadigm language that supports both functional and object-oriented programming. For all implementations, we have measured development time, lines of code, execution time and memory usage. Our results show that object-oriented programming gave us better performance, but functional programming resulted in less code and a shorter development time.

Keywords

Functional programming, Object-oriented programming, Comparison, Programming paradigms, JavaScript

Contents

Abbreviations

ES6 = ECMAScript 2015

FP = Functional Programming

OOP = Object-Oriented Programming

RP = Reactive Programming

1 Introduction

Programming paradigms is a way to classify a certain way of programming [1]. Each paradigm has its own set of rules for how the code should be written. Some programming languages make it possible to write code according to one paradigm, while others make it possible to write according to multiple paradigms.

In the 1950's and 60's, multiple high-level languages were developed that today's programming languages are based upon [2]. These were imperative languages such as Fortran and Algol, functional languages such as Lisp and object-oriented languages such as Simula. Originally Lisp had performance issues and has not been used a lot commercially, but versions of it are still used today. However, it is mostly used an academic context [2, 3]. In the 70's, C was introduced and quickly established itself thanks to its ability to access low-level functionality, its compact syntax and its effective compiler. In the 80's, the object-oriented language C++ was developed. It was based on C and and took a lot of inspiration from Simula. Java, that is based on C++ was introduced in the 90's. Since imperative languages gained the upper hand in the 60's and 70's, new languages were based on those and continued being the most used.

Object-oriented or imperative languages are dominating today, with languages such as C, C++, Java and C#. However, functional languages seem to be of interest in the industry again for handling big data and concurrency, with languages such as Scala, Erlang and Haskell [4, 5, 6, 3]. Since functional programming (FP) tries to avoid state, it becomes less complex to run asynchronous and therefore easier to run concurrently. For example, Eriksen [5] explains that they use functional approaches at twitter to handle the complexity in cloud environments. FP is also well suited for test driven development (TDD) and could increase testability for cloud based applications [7].

There is a performance difference, where functional languages tend to be slower, but there are positive aspects that could compensate for this, such as memory usage and less code [8, 3]. Some are also arguing that a more functional approach will give you more readable code, code that is easier to maintain and easier to test, and that learning it will give you a better experience as a programmer [4, 9]. Studies suggest that developers that use functional approaches are more productive, and that the use of functional approaches give better program comprehension [10, 11].

We will compare two programming paradigms, object-oriented programming (OOP) and FP. To compare these, we have chosen four different algorithms which both of us will implement twice, once according to the object-oriented paradigm and once according to the functional paradigm. The different algorithms we will implement are binary search tree algorithms, the Shellsort algorithm, the Tower of Hanoi algorithm and Dijkstra's algorithm. The comparisons

we will make between the algorithms are execution time, memory usage, development time and lines of code. All algorithms will be written in JavaScript, since this is a well known programming language, we both had prior experience with and it supports both the functional and object-oriented paradigm.

We chose to do an experiment with the two of us, since we decided that the scope would be too large if we included more people. It was a good way for us to collect measurements to see the difference between the two paradigms.

1.1 Research Questions

In this thesis we will address the following research questions:

- **RQ1** - Will functional versus object-oriented approaches in JavaScript have an impact on performance, such as execution time and memory usage?
- **RQ2** - Can programmers with an object-oriented background decrease development time and code length by using practises from functional programming?

When comparing the different paradigms, it would be interesting to see how the execution time and memory usage differs. Because we are writing both paradigms in the same language, we are able to compare just the paradigms, without having to compare languages and compilers.

We both have studied at Blekinge Institute of Technology, where we only learned OOP. Therefore, it would be interesting to see how our programming will change by learning FP. We want to see if we can be better programmers and work more effectively using FP approaches. This can give teachers an indication about introducing FP, to give students a chance to use different paradigms depending on their needs, instead of always using OOP.

2 Background and Related work

Here we will describe the FP paradigm and also the OOP paradigm. We will also describe JavaScript and its support for functional and object-oriented methods. The related works described are articles that further explain the positive aspects of FP or studies similar to this one.

2.1 Functional programming

FP is a programming paradigm that at its core is based on lambda-calculus [2]. 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 [8].

Static type checking - Pure functional languages usually have static type checking [2]. This means that variables are of a certain type, for example int or char. In such languages it is not allowed to use functions or variables 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 dynamically type checked-languages this would be legal at compile time, since variables are not specified as types, but would cause an unexpected error at runtime. For example in JavaScript:

```
1 //JavaScript has dynamic type checking
2 function double(nr) {
3   return nr * 2;
4 }
5
6 var word = "string";
7 double(word); //Will return NaN but still continue
    running
```

A similar program in golang [12], that is a statically type checked language:

```
1 func main() {
2   var word string = "string"
3   fmt.Printf("%d", double(word)) //Error: cannot use
      word (type string) as type int in argument to double
4 }
```

```
5
6  func double(nr int) int {
7    return nr * 2
8  }
```

This will give an error when compiling.

There are however pure functional languages with dynamic type checking, such as Clean [13]. There are also languages that are mostly functional with dynamic type checking, such as Common Lisp [14] or Scheme [15] [16].

Side effects - Side effects are for example changing a variable or having some interaction outside of the function [4]. This is avoided in FP 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 [4]. See the following example in JavaScript:

```
1  var sum = 0;
2
3  //Impure
4  function add(a, b) {
5    sum = a + b;
6  }
7
8  //Pure
9  function add(a, b) {
10   var tmp = a + b;
11   return tmp;
12 }
```

Higher order functions - Higher order functions are an important concept in functional programming [8]. First class functions mean that functions are treated as values, which means that they can be stored in variables and 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. See the following example in JavaScript:

```
1  function add(a, b) {
2    return a + b;
3  }
4
5  //Functions can be placed in variables.
6  var thisFunc = add;
7
8  //And also put in other variables.
9  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 [8]. It is important in FP since it can hide mutable state and also implement laziness. In FP loops are avoided and recursion is used instead. Following is an example in JavaScript:

```
//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
```

```
22 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 [4]. Following is an example in JavaScript:

```
1  //Returns a new function that takes the remaining
       argument or the sum of a and b if both are provided.
2  function addWithCurrying(a, b) {
3    if(b) {
4      return a + b;
5    }
6    else {
7      return function(b) {
8        return a + b;
9      }
10   }
11 }
12
13 var curryAdd = addWithCurrying(4);
14 curryAdd(6); //Outputs 10
15 addWithCurrying(4, 6); //Also outputs 10
```

Immutable data structures - In functional programming mutations, that are side effects, are avoided [8]. 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. In languages with immutable data structures it is illegal to change the value of such variables.

2.2 Object-oriented programming

OOP is a programming paradigm which is built around "objects". These objects may contain data, in the form of fields often referred to as attributes, and code, in the form of procedures often referred to as methods [17].

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, see figure 1. An employee has the attributes name, 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 name, 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 employee class.

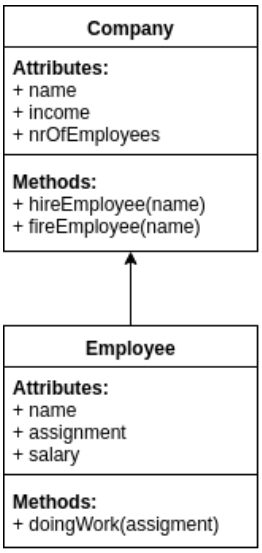

Figure 1: OOP class example

By creating classes and object like this, you can create programs according to the object-oriented paradigm.

The following are concepts used in OOP:

Class - A class is a model for a set of objects, which the object oriented paradigm is built around [2]. The class establishes what the object will contain, for example variables and functions, and signatures and visibility of these. To create an object of any kind a class must be present. In the JavaScript example below a class called Animal is implemented:

```
1 //Creates a class Animal with the properties name and
      age and a function for logging the properties to the
       screen
2 class Animal {
3   constructor (name, age) {
4     this.name = name;
5     this.age = age;
6   }
7
8   logAnimal() {
9     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 structure of variables and functions established in the class[2, 18]. While the structure looks the same in objects created with the same class, the containing information can vary.

Inheritance - When an object acquires all properties and functionality of another object it is called inheritance [18]. This provides code reusability. The following is an example in JavaScript:

```
//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. A mechanism to restrict direct access to some object components and the language construct that facilitates the bundling of data with methods [2, 18]. Access to variables and functions are established in the class by using the private and public keyword. The bundling is the object itself.

Polymorphism - When a method is used in different ways it is called polymorphism [2, 18]. 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 [18].

Overriding - Refers to overwriting a function written in a superclass to make it do something else than first intended, without changing the superclass [18].

2.3 JavaScript

JavaScript is a language with first class functions that is dynamically type checked. It is a multi-paradigm language with basic support for object-oriented, imperative and functional paradigm principles. There are multiple libraries to tweak JavaScript with certain functions or better support for certain paradigms, for example Underscore.js [19], for FP, or Flow [20], for static type checking. There are also a lot of languages available that are transpiled into JavaScript, for example TypeScript [21].

JavaScript can be run in most browsers and also in servers and is implemented to follow the ECMA standards [22]. It has a garbage collection system for handling memory [23]. The garbage collection removes objects or variables from memory that are unreachable by the program.

Objects in JavaScript are treated as references. When an object is initiated to a variable it is created and placed in memory and the variable is given a reference to it. This means that if another variable is assigned this object, it gets a copy of the reference to it, rather than a copy of the object itself. Arrays are objects in JavaScript and will be treated the same way.

```
1  //JavaScript Object handling
2  var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
3  var obj = {
4    text: 'Some text'
5  }
6
7  var arrayRef = array;
8  var objRef = obj;
9
10 // Gets references to original array and object.
11 console.log(arrayRef); // [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
12 console.log(objRef); // { text: 'Some text' }
13
14 // Change stuff
15 arrayRef.pop();
16 objRef.text = 'Some other text';
17
18 // Mutating new variables mutates original objects.
```

```
19 console.log(array); // [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
20 console.log(obj); // { text: 'Some other text' }
```

2.3.1 Functional programming

JavaScript is not a functional language, but it is possible to write functional code with it [4]. There are also libraries that make FP in JavaScript easier, such as Underscore.js [19] [8]. 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 [4]. There are also higher order functions such as map(), filter() and reduce() that might replace loops [24]:

map() - Calls a provided function on every element in an array and returns an array with the outputs [25].

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

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

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

There is a call stack limit in JavaScript that varies depending on the runtime environment. This limits the number of function calls that can be made, which also limits recursion. In ES6 there is tail call optimization, that makes it possible to make certain function calls without adding to the call stack, that would allow for better recursion. However this is currently not implemented for most servers and browsers [22].

2.3.2 Object-oriented programming

JavaScript has always had support for OOP. But with ES6, OOP in JavaScript starts to look more like classical OOP languages such as Java [26].

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 many programmers add an underscore to the variable or function name. This, however, does not actually make anything private. It just gives a hint to the programmer that it should be treated as such.

2.4 Related work

In "Curse of the excluded middle" [9] 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. 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" [5] by Marius Eriksen he is explaining why Twitter uses methods from FP to handle concurrent events that arises in large distributed systems in cloud environments. In the FP paradigm 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.

In "Improving Testability and Reuse by Transitioning to Functional Programming" [7], Benton and Radziwill state that FP 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.

Dobre and Xhafa writes in "Parallel Programming Paradigms and Frameworks in Big Data Era" that we now are in a big data era [27]. They also review different frameworks and programming paradigms in a big data perspective. Regarding 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" [27].

Eriksson and Ärleryd are looking at how to use functional practises, such as immutable data structures, pure functions and currying, when developing front end applications by taking inspiration from Elm [28] in their master's thesis [6]. 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 included to use the same practises. They also concluded that even though FP is not widely used within the industry, functional practises can still be used in all projects.

Alic, Omanovic and Giedrimas have made a comparative analysis of functional and OOP languages [3]. They have compared four languages, C#, F#, Java and Haskell based on execution time 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.

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 [29]. To compare these, they use C++ and SML. While they state in their discussion 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 FP concepts, on program comprehension [10]. Their experiment involved 127 subjects and the results suggest 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" [11], 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.

3 Method

In this experiment each person will implement four different algorithms, once with FP and once with OOP. There will be no cooperation between us and the algorithms will be implemented individually. 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. For each of these algorithms we will measure execution time, memory usage, development time and lines of code. We have not implemented these algorithms earlier, but before implementation we defined what and how the algorithms should be implemented. We also defined paradigm guidelines for us to follow during implementation and reviews, see section 3.6. This was to make the comparison as fair as possible, such as avoiding spending time on an algorithm for the first implementation, and then being more prepared for the second algorithm.

For each implementation we will measure execution time, memory usage, code length and the development time. The performance measurements, execution time and memory usage, will be used for answering RQ1. The development time and code length is measured for answering RQ2, which is interesting for seeing if we can implement more effectively using functional approaches. Code length is measured to find a connection between our development time and our way of implementing.

3.1 Algorithms

We will implement binary search tree algorithms, the Shellsort algorithm, the Tower of Hanoi algorithm and Dijkstra's algorithm. One of the binary search tree algorithms we will implement is in-order tree traversal, which is a recursive algorithm, as is the Tower of Hanoi algorithm. These algorithms fit well for FP since they use recursion. The implementation of the binary tree structure can however be implemented by using classes and objects, that are 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.

3.1.1 Binary search tree algorithms

A binary tree is a tree consisting of nodes, where a node can have a maximum of two children [30]. These children can be described as the left and the right subtree. 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 binary 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 the example in figure 2. Using tree traversal it is possible to find certain nodes or get a list of sorted objects. Our binary search 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.

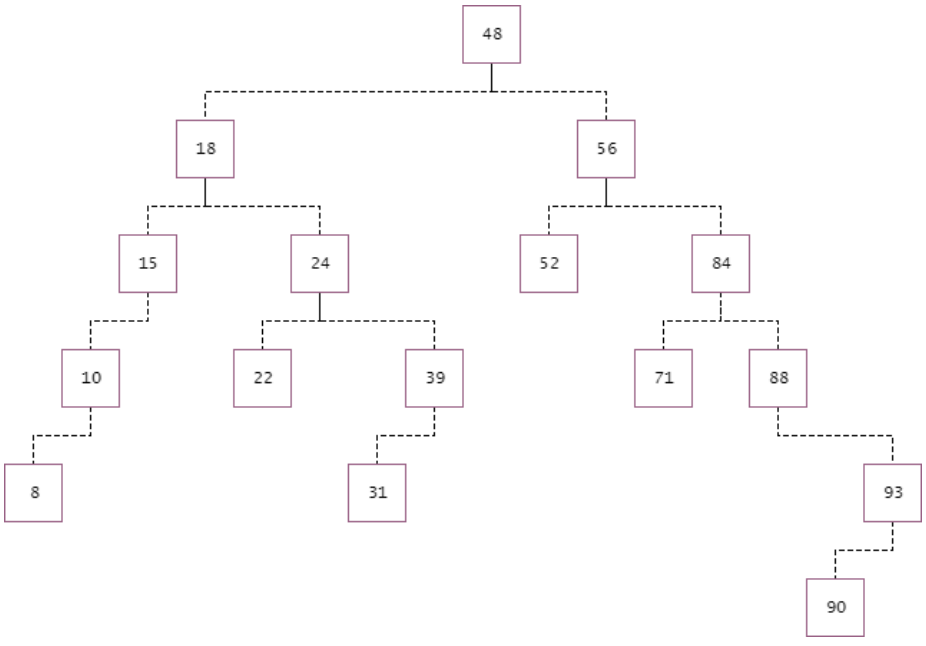

Figure 2: Binary tree example

The functions will utilize the following algorithms:

Algorithm descriptions in pseudocode:

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
  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 on the tree in figure 2 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 then
    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
```

3.1.2 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 give Shellsort one of the fastest worst-case running times, $O(n^3/4)$.

The sequence is

$\{1, 5, 19, 41, 109...\}$, where every other term is of the form

$9 * 4^k - 9 * 2^k + 1$, and every other term is of the form

$4^n - 3 * 2^n + 1$, where

$k = 0, 1, 2, 3...$ and $n = 2, 3, 4, 5....$

In our implementations the sequence is calculated during execution and included in our measurements.

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.

The function will utilize the following algorithm:

Algorithm description in pseudocode:

algorithm algorithm Shellsort(array of size n)

```
set sortedArray to array
set gapSequence to Sedgewicks 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
```

3.1.3 The Tower of Hanoi algorithm

The Tower of Hanoi is a game invented by mathematician Èdouard Lucas in 1883 [31]. The game consists of three pegs and a number of disks stacked in decreasing order on one of the pegs, see figure 3. 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.

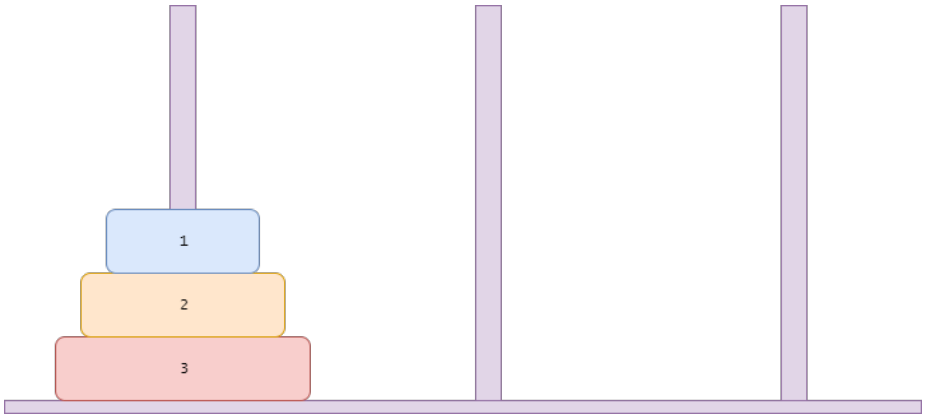

Figure 3: Tower of Hanoi image

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.

The function will utilize the following algorithm:

Algorithm description in pseudocode:

algorithm hanoi(tower, start, dest, aux)

```
1 (*Pseudo code based on that tower is the number of the
    largest disk, where 1 is the smallest disk in the
    tower.*)
2 if tower is equal to 1
3   move tower from start to dest
4 else
5   hanoi(tower - 1, start, aux, dest)
6   move tower from start to dest
7   hanoi(tower - 1, aux, dest, start)
8 end if
```

3.1.4 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 4. 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 5.

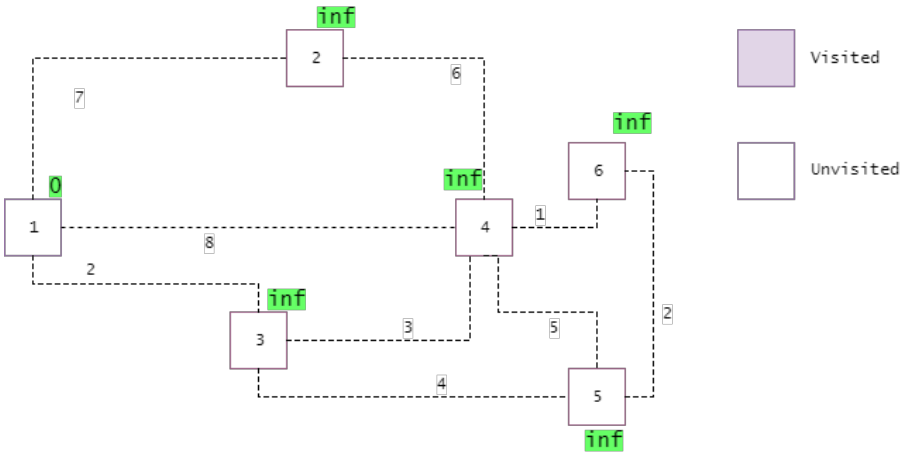

Figure 4: After initiation of Dijkstra's algorithm

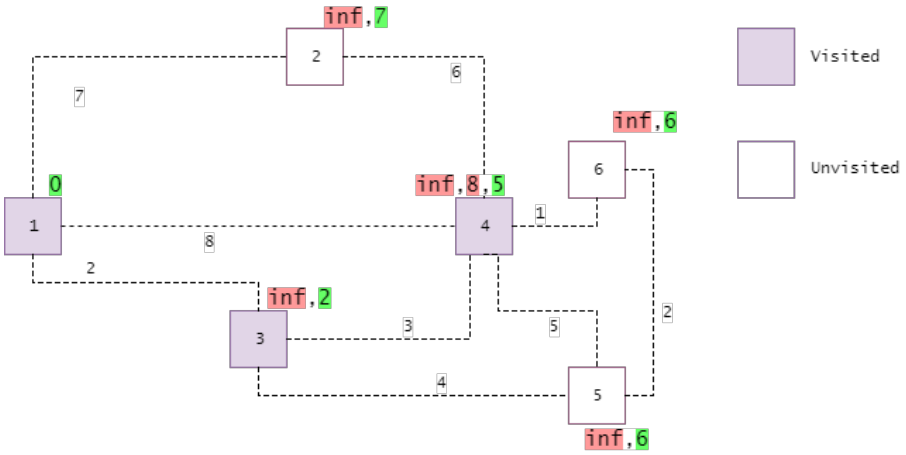

Figure 5: After three nodes have been visited with Dijkstra's algorithm

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.

The function will utilize the following algorithm:

Algorithm description in pseudocode:

algorithm dijkstras(graph, startNode, endNode)

```
1  for each node in graph
2    set dist[node] to infinity
3    set path[node] to undefined
4  end for
5  set dist[startNode] to 0
6  set unvisitedNodes to nodes in graph
7
8  while unvisitedNodes is not empty or endNode is not in
       unvisitedNodes do
9    set current to node where dist[node] is smallest and
       node is in unvisitedNodes
10   remove current from unvisitedNodes
11   for each neighbor of current where neighbor is in
       unvisited do
12     set temp to dist[current] + weight of edge in graph,
        where edge is from current to neighbor
13     if temp is smaller than dist[neighbor] then
```

```
14        set dist[neighbor] to temp
15        set path[neighbor] to path[current] + current
16      end if
17    end for
18  end while
19  return path[endNode]
```

3.2 Environment

We will implement the algorithms using Atom [32], a text editor, and compile and run our tests and our code with Node.js [33]. Tail call optimization is not available in Node.js, see 2.3.1. To manage our dependencies we are using npm [34] and for automation we are using Grunt [35]. We have been using github for managing and reviewing our code, see `https://github.com/KimSvenSand/ToKi`.

Since Node.js does not support ES6-modules we are using Babel [36] 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 supports 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 variable and therefore always available within Node.js applications [33]. 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. These times are relative to an arbitrary time in the past, and not related to the time of day and therefore not subject to clock drift.

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

For measuring development time, we will each have a timelog. In the timelog we will log the time when we start working, the time we stop working and additional comments that might help when analyzing our result. When the implementations are ready and reviewed, we will calculate how many hours we spent on each algorithm.

Since all our implementations have been implemented in Atom [32], we have used the Line Count package [37] for counting our lines of code. We are not counting blank lines or comments.

3.3 Testing

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

3.3.1 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 according to our guidelines, see 3.6. To have a third party review our code will decrease bias and will be a deciding factor as to if our code has to be improved. The time it takes to improve code will be added to the development time.

3.3.2 Unit testing

Unit testing will be done using the JavaScript libraries Mocha [38] for writing tests and Chai [39] for evaluating expressions. Our implementations will have to pass the tests in appendix A to be accepted as done. By defining tests beforehand, we will each include the same tests in our implementations and avoid spending time looking for small bugs, which might affect our results.

3.4 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 our tests we 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 execute our code we 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. The implementation must have a function for creating randomized data of a certain size according to table 1
- A timelog describing how long the implementation took.

3.5 Measurements and indata

We will measure execution time, memory usage, code length and the development time of each implementation. The measurements are done on the algorithms described earlier in this chapter and the algorithms will run on randomized data, described in table 1. We are not measuring the initiation of the randomized data, except for the binary search tree algorithms that include an insert-function. The algorithms are measured with three different sizes of data, described in table 1, to get more accurate results. We have chosen measurement sizes such as to avoid exceeding the call stack limit for our functional implementations, that use recursion.

Table 1: Table describing measurements to be implemented

Algorithm	Binary search tree algorithms
Description	Should measure insert(), inOrderTraversal() and findNode(), where findNode() is run three times with a random value between 1 and 10 * n as parameter.
Measurement data	A binary search tree with n nodes filled with randomised values between 1 and 10 * n.
Measurement sizes	n = 1000, n = 5000 and n = 10000.
Algorithm	Shellsort algorithm
Description	Should measure the Shellsort-function.
Measurement data	An array of length n filled with randomised values between 1 and 2 * n
Measurement sizes	n = 1000, n = 3000 and n = 6000.
Algorithm	Tower of Hanoi algorithm
Description	Should measure the hanoi-algorithm where a tower is moved from one peg to another.
Measurement data	A tower of n disks.
Measurement sizes	n = 10, n = 20 and n = 25.
Algorithm	Dijkstra's algorithm
Description	Should measure Dijkstra's algorithm from the first node in the randomised graph to the last node in the randomised graph.
Measurement data	A graph with n nodes where each node is connected to the last two nodes, if those nodes exist. The edges between the nodes should be of a randomised weight between 1 and 100.
Measurement sizes	n = 10, n = 15 and n = 20.

3.6 Guidelines for implementations and reviews

The following are guidelines for when the algorithms are implemented. They will also work as criteria for the implementations when reviewing, and has to be fulfilled for the implementation to be approved. The guidelines have been chosen with regards to literature describing programming approaches and what is supported by JavaScript. We have also chosen to prefer iteration for the object-oriented implementations, since we must use recursion for the functional implementations.

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

4 Results and analysis

Our implementations have been reviewed by each other and also by a lecturer at BTH with prior experience with FP. During reviews we focused on finding any deviation from the guidelines, see section 3.6, and finding larger bugs and mistakes. Small bugs were intended to be found using our tests, see appendix A, but a few bugs still slipped through and had to be fixed. They were found when we noticed some strange results that we could not explain. For some of our implementations we had made mistakes when implementing the index-files, that ran the measurements, and these were not included in our tests.

All of our code is available at `https://github.com/KimSvenSand/ToKi`

4.1 Development time

The total results for the development time were:

Kim OOP - 21 hours and 12 minutes.

Kim FP - 19 hours.

Tord OOP - 24 hours and 22 minutes.

Tord FP - 22 hours and 36 minutes.

In figure 6 we can see that both of us spent less time on the functional implementations in total. We both had to spend more time on the OOP solution when implementing the recursive algorithms, and more time on the FP solution when implementing the iterative algorithms. This is true except for Dijkstra's algorithm where Tord spent about as much time on both of the implementations.

Tord had to redo both of his Dijkstra's algorithm implementations, the functional Shellsort algorithm and the object-oriented Tower of Hanoi after the initial results were registered. He spent a lot of time finding and fixing the bug for his functional Dijkstra's algorithm, which explains the deviation from the expected pattern.

We can see that there was a 60% difference in development time between Kim's functional and object-oriented Tower of Hanoi algorithm. We can also see that she spent a lot more time than Tord did on his Tower of Hanoi implementations. This was because she struggled a lot with the iterative Tower of Hanoi algorithm and chose to use data structures where the algorithm had to be altered from the pseudocode in section 3.1.3. Another difference between Kim's and Tord's implementations was that Tord got stuck and searched for help online regarding the iterative Tower of Hanoi algorithm.

We hoped that we would see a clear advantage for FP, but the results differ between each algorithm. In total we spent less time on the functional implementations, but for some algorithms we both spent significantly less time on the object-oriented implementations. The results suggest that the major difference in development time is depending on the algorithm itself and how it is implemented, rather than on the paradigm that is used. However, both of us have had prior experience working with OOP and close to no experience working with FP. This means that FP has a larger advantage than what is apparent in our results.

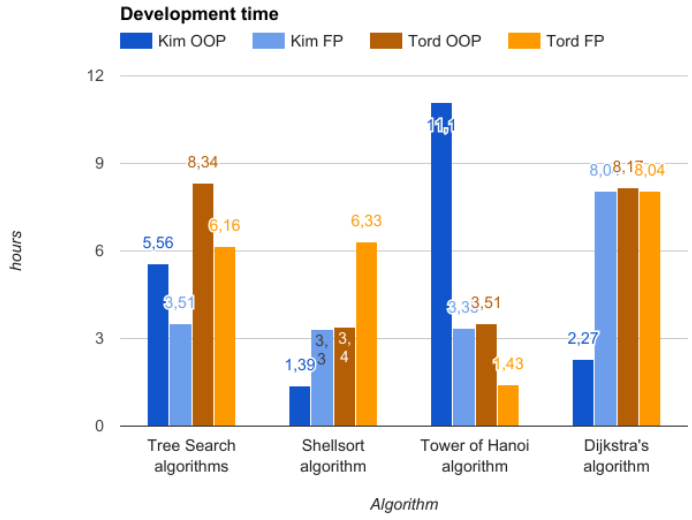

Figure 6: Development time for our implementations

4.2 Lines of code

The total results for the lines of code were:

Kim OOP - 930 lines of code.

Kim FP - 618 lines of code.

Tord OOP - 703 lines of code.

Tord FP - 530 lines of code.

For both Kim and Tord the functional implementations had less code in total and individually for all implementations except for the Shellsort algorithm, see

figure 7. For the Shellsort algorithm both OOP implementations had the least code.

We can see for Kim's Tower of Hanoi algorithm that there is a difference of 233 lines in code length between the OOP and the FP implementations. This is because of her choice to using classes to represent the Tower of Hanoi game, pegs and discs rather than just using arrays.

We expected that the functional implementations would produce less code than the object-oriented implementations. The functional implementations did, for all algorithms except for Shellsort, have less code because we did not implement classes. This gave us less code, but we both think that this does not increase readability. Classes add a layer of abstraction that we both found useful for writing our own code and understanding each other's.

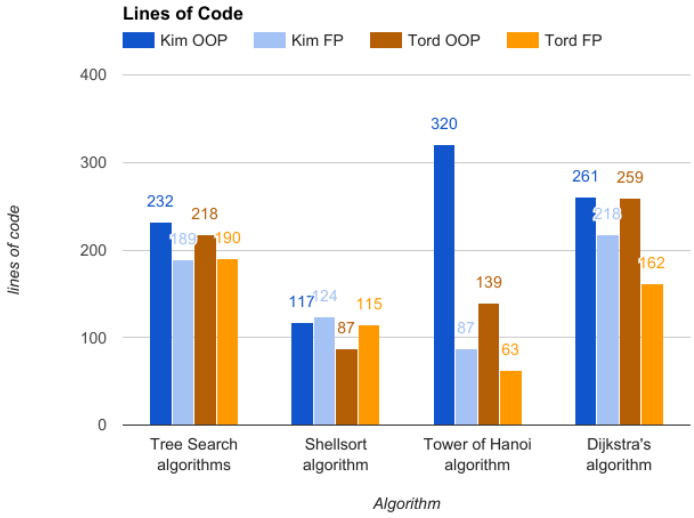

Figure 7: Lines of code in our implementations

4.3 Binary Search Tree Algorithms

In figure 8 and figure 9 we can see that for Kim the functional implementation has a better execution time, but still uses a lot more memory than the object-oriented implementation. Tord's object-oriented implementation performs a lot better than his functional, both for memory usage and execution time. Tord's functional implementation is a lot slower and uses a lot more memory than all the other implementations.

Kim's functional implementation is faster than her object-oriented, since her inOrderTraversal-function is unnecessarily complex and does not fully utilize the idea of the recursive algorithm seen in section 3.1.1. Her recursive algorithm uses more memory because we are handling our data structures as immutable. This means that we are copying our data structures recursively when initiating an array or a tree, which JavaScript handles by placing all these copies into memory. We suspect that this would have looked different in other languages that support immutable data structures.

The reason that Tords functional algorithm is so much slower is that he used arrays instead of native objects. The problem with arrays is not the array itself but how it is used for the algorithm. The child nodes are placed in the array at $n * 2 + 1$ or $n * 2 + 2$, where n is the parent's index. This made the array much bigger than necessary.

Overall our results suggest that it is possible to write good algorithms in both paradigms, but OOP has the upper hand since it is using significantly less memory than FP.

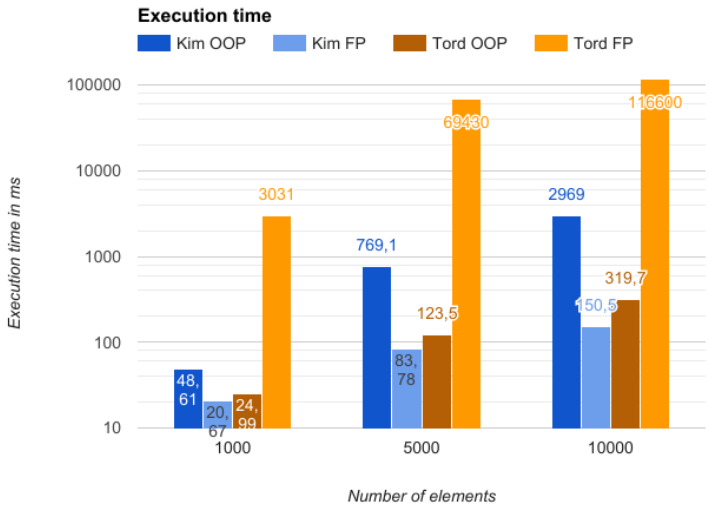

Figure 8: Execution time for binary search tree implementations.

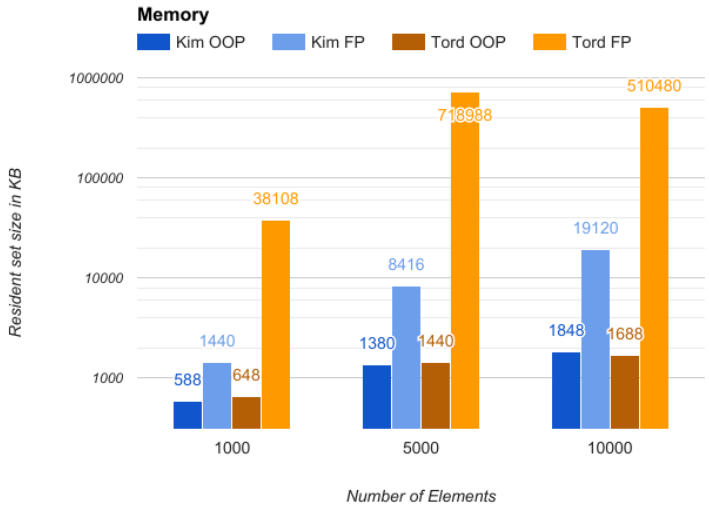

Figure 9: Memory usage for binary search tree implementations.

4.4 Shellsort

For Shellsort Kim's and Tord's results are following the same trend, see figure 10 and figure 11. Both of their OOP implementations performed better with faster execution times and lower memory usage. As seen in figure 6 and figure 7 both OOP implementations also had less code and a shorter development time than the FP implementations.

The reason that the functional and object-oriented implementations were very similar in code length is that there were not many classes to implement in the object-oriented solution. Working with and copying arrays recursively is also a lot slower, probably because of the high memory usage and JavaScript's garbage collection that has to keep collecting when the memory usage is too high.

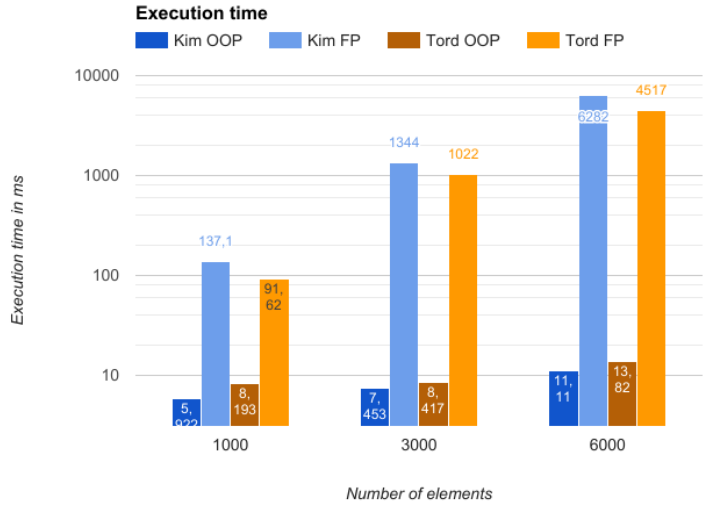

Figure 10: Execution time for the Shellsort implementations.

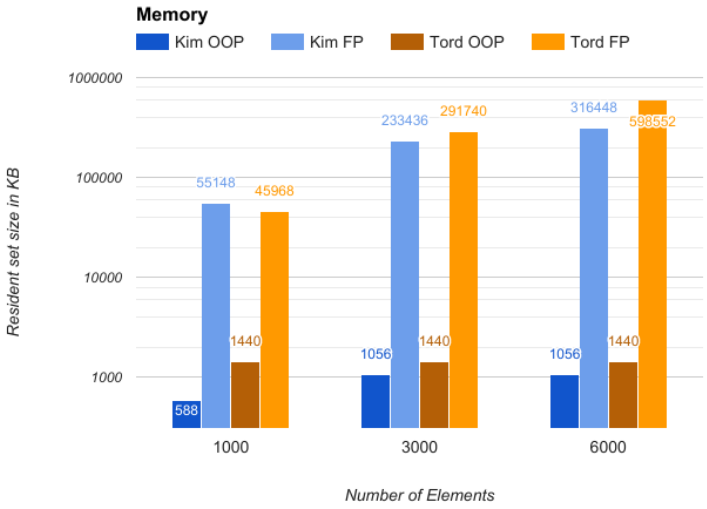

Figure 11: Memory usage for the Shellsort implementations.

4.5 Tower of Hanoi

In figure 12 and figure 13 we can see that Kim's functional implementation was both faster and used less memory than her object-oriented implementation. However, there was not as big of a difference in performance between the two compared to results for other implementations. Tord's object-oriented implementation was a lot faster and used a lot less memory than all the others, while his functional implementation was the slowest and used the most memory.

Tord had to change the object-oriented implementation of this algorithm after we had analysed the results. This was because during the first measurement runs this implementation ran out of memory with 25 elements and was significantly slower than it is currently. This resulted in that his implementation was improved after we analyzed the first results, which might have decreased the time he had to spend. On the other hand it shows us that it is easy for small mistakes to have a large impact on performance when handling larger data sizes. This was also the algorithm where Tord searched for help online, which might also have resulted in that it is more effective than the other implementations.

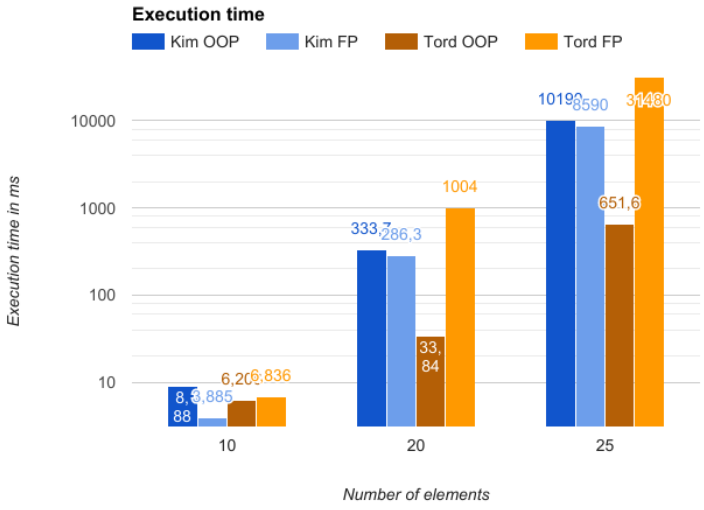

Figure 12: Execution time for the Tower of Hanoi implementations.

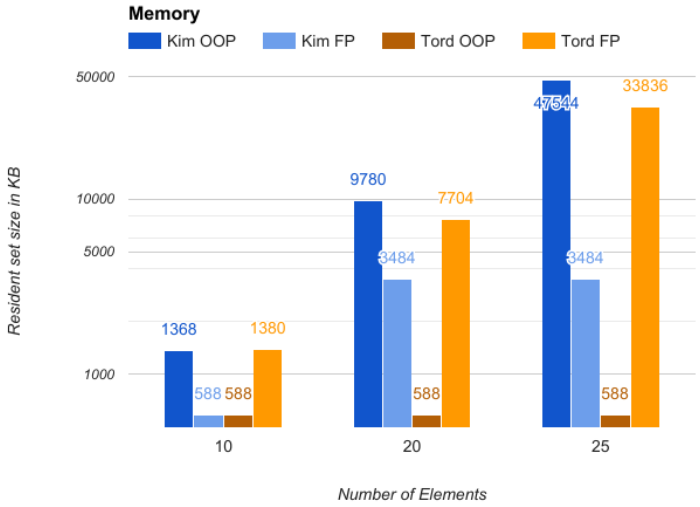

Figure 13: Memory usage for the Tower of Hanoi implementations.

4.6 Dijkstra's algorithm

Both Kim's and Tord's object-oriented implementations performed better than their functional implementations for Dijkstra's algorithm. Kim's functional implementation was slower than all the others, while Tord's functional implementation only performed slightly worse than his object-oriented implementation, see figure 14 and figure 15.

As expected the object-oriented implementations performed better for this algorithm. However, Kim's functional implementation is a lot slower and uses a lot more memory than the others. This is because she chose to represent edges as an object where the keys were the edges and the values were the weight. For example:

```
1 var edges = {
2   'node1-node2': 9
3   'node2-node3': 3
4 }
```

She used JavaScript's Object.keys() function, see [25], to get an array with the keys and then processed that to get the wanted edges. This adds a lot of copying and handling of objects and arrays, which in turn results in a higher memory usage and more garbage collecting, which leads to a slower execution time.

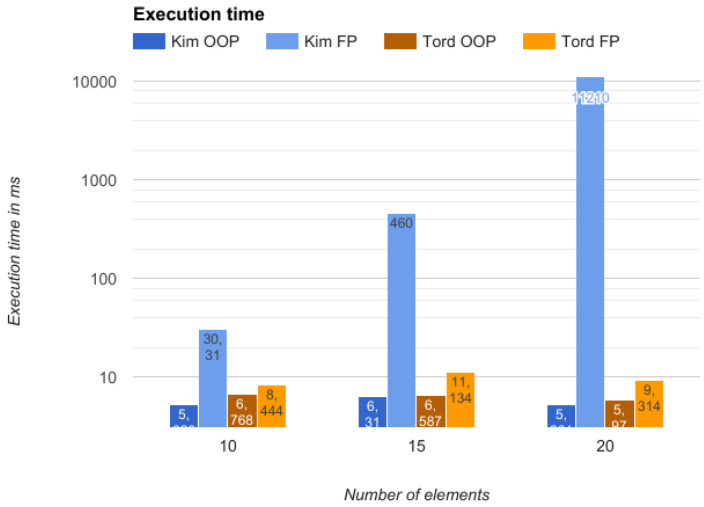

Figure 14: Execution time usage for Dijkstra's algorithm implementations.

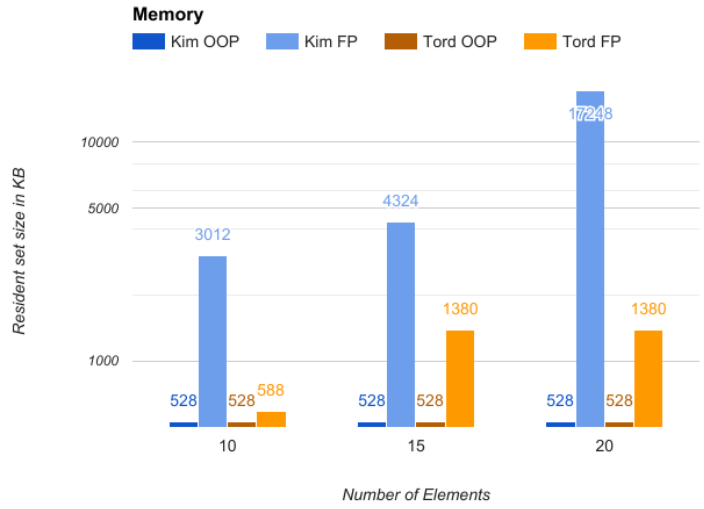

Figure 15: Memory usage for Dijkstra's algorithm implementations.

5 Discussion

Most of the implementations are faster when implemented in their expected way. For example, the binary search tree algorithms have a faster execution time when implemented recursively and the Shellsort algorithm has faster execution time when implemented iteratively. The development time results also follow this pattern. We spent more time when implementing, for example, Shellsort recursively or the Tower of Hanoi iteratively.

We expected the development time to be shorter for the object-oriented implementations, since we both have had prior experience with it and since we have nearly no prior experience with FP. The results show that this was not necessarily so. Neither of the paradigms seem significantly faster to implement, however the functional paradigm might have a slight upper hand since we both had no prior experience with it.

For the execution time results, we did expect the object-oriented implementations to be faster than the functional implementations, based on this study comparing functional and object-oriented languages [29] and our research about JavaScript, see section 2.3. Overall the functional implementations performed worse than the object-oriented implementations. The functional implementations were not always slower, but the slow functional implementations were slower than the slow object-oriented implementations.

Since we only had two different test groups with one person in each, it was difficult to analyze our results when algorithms were hard to compare. For example when an algorithm was much slower than all the others. This might not be because of the paradigm itself, but rather because the programmer has made a mistake. However, it can still give us some results as to if it is easier to make mistakes in one paradigm, but that would only be valid for that particular language. We think that it would be possible to draw more conclusions if there were more test subjects taking part in this experiment.

For this experiment it would have been better to have more people. However, we decided that our scope would be too big if we also incorporated other people. It would take time to find appropriate test subjects, and also to monitor them and make sure that we had our results on time. We also wanted to measure testability, but we could not find an appropriate measurement for this, so this was also not included in our scope. We focused on predefining as much as possible before starting the implementations. This was to avoid affecting our results with things such as different testing or different knowledge before implementing. We also had a third review our code for us, and the implementations and reviews were made individually.

Our results suggest that the paradigm methods themselves, such as using classes with methods or avoiding side effects, do not necessarily have a large impact on

the performance. In some cases, for example in our Shellsort implementation results in section 4.4, FP clearly performs worse than OOP. Shellsort requires a lot of hiding state and handling data structures as immutable, which is slow and memory consuming in JavaScript. Optimizing functional solutions is an option, but that would take additional time and would have to be considered when developing functional JavaScript. Perhaps using one of the functional libraries available solves this problem.

Overall we both appreciated trying out FP, but we don't think that we will use pure FP or functional languages in the future if we don't have to. We will, however, use a lot of the approaches that we have been using in this experiment. In our experience we found that we had less bugs when avoiding side effects and the functions were modular and easy to combine. We also liked using classes since there was a layer of abstraction on top of JavaScript's native types. We think that combining the two paradigms and using recursion or iteration when it fits best would benefit both the performance, development time and code readability. This conclusion is supported by the results of the comparative analysis by Alic et al. [3]. When using FP we spent less time in total, and when we used more time, it was because we spent time using recursion instead of iteration. If local loops are used instead, we could avoid the performance issues and still avoid side effects and bugs. We believe that if this is added to an object-oriented program structure, the code would be readable and development would be effective.

6 Conclusion

As we expected, based on our literature [3, 10], our object-oriented implementations performed better than our functional implementations, while our functional implementations had a shorter development time and less code. This was the case, even though we did not have any earlier experience with FP.

Based on our results, different paradigms are good for different things and one is not clearly better than the other. According to our research most languages perform well, and there are also a lot of multiparadigm languages out there. Our conclusion is that it is best to use a combination of the paradigms if possible. Especially since FP and OOP are not opposites. We think that object-oriented programmers would benefit from using methods from FP, even if they work in a more object-oriented language. To us it makes sense that a lot of the object-oriented languages, that are based on the imperative paradigm, are starting to implement support for functional approaches.

7 Future Work

In this experiment we have implemented the algorithms ourselves and compared our own work. It would be interesting to see a similar experiment with multiple test subjects that are not involved in the thesis work.

Some sources argue that FP should be good for handling data concurrently [9, 5, 27]. Therefore an experiment to compare different paradigms, when implementing concurrent and asynchronous algorithms, from both a performance, development and maintainability perspective would be interesting.

We also did these implementations in JavaScript. It would be interesting to see this kind of comparison in another language that better support full functional or full OOP approaches. Perhaps comparing other paradigms and a combination of paradigms as well.

References

[1] Programming paradigms. [Online]. Available: http://cs.lmu.edu/~ray/notes/paradigms/

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

[3] D. Alic, S. Omanovic, and V. Giedrimas, "Comparative analysis of functional and object-oriented programming," in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2016, pp. 667–672. [Online]. Available: http://ieeexplore.ieee.org.miman.bib.bth.se/document/7522224/citations

[4] drboolean. (2017) Gitbook, professor frisby's mostly adequate guide to functional programming. [Online]. Available: https://drboolean.gitbooks.io/mostly-adequate-guide/content/

[5] M. Eriksen, "Functional at scale," *Queue*, vol. 14, no. 4, pp. 60:39–60:55, Aug. 2016. [Online]. Available: http://doi.acm.org.miman.bib.bth.se/10.1145/2984629.3001119

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

[7] M. C. Benton and N. M. Radziwill, "Improving testability and reuse by transitioning to functional programming," *CoRR*, vol. abs/1606.06704, 2016. [Online]. Available: http://arxiv.org/abs/1606.06704

[8] M. Fogus, *Functional JavaScript*, 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., may 2013.

[9] E. Meijer, "The curse of the excluded middle," *Commun. ACM*, vol. 57, no. 6, pp. 50–55, Jun. 2014. [Online]. Available: http://doi.acm.org.miman.bib.bth.se/10.1145/2605176

[10] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini, "On the positive effect of reactive programming on software comprehension: An empirical study," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[11] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy, "Using functional programming within an industrial product group: Perspectives and perceptions," *SIGPLAN Not.*, vol. 45, no. 9, pp. 87–92, Sep. 2010. [Online]. Available: http://doi.acm.org.miman.bib.bth.se/10.1145/1932681.1863557

[12] Google. The go programming language. [Online]. Available: https://golang.org/

[13] (2016, dec) Clean. [Online]. Available: http://clean.cs.ru.nl/Clean

[14] The Common Lisp Foundation. (2015, may) Common lisp. [Online]. Available: https://common-lisp.net/

[15] (2015, jan) Scheme reports. [Online]. Available: http://www.scheme-reports.org/

[16] (2016, dec) Comparison of functional programming languages. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_functional_programming_languages

[17] E. Kindler, "Object-oriented simulation of simulating anticipatory systems," *International Journal of Computer Science*, pp. 163–171, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.8133

[18] J. Skansholm, *C++ direkt*, 3rd ed. Lund: Studentlitteratur AB, 2012.

[19] (2015, apr) Underscore.js. [Online]. Available: http://underscorejs.org/

[20] Facebook Inc. (2017) Flow: A static type checker for javascript. [Online]. Available: https://flow.org/

[21] Microsoft. (2016) Typescript - javascript that scales. [Online]. Available: https://www.typescriptlang.org/

[22] kangax and zloirok. (2017) Ecmascript 6 compability table. [Online]. Available: http://kangax.github.io/compat-table/es6/

[23] Mozilla Developer Network and individual contributors. (2017) Memory management - javascript — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

[24] M. Grady, "Functional programming using javascript and the html5 canvas element," *J. Comput. Sci. Coll.*, vol. 26, no. 2, pp. 97–105, Dec. 2010. [Online]. Available: http://dl.acm.org.miman.bib.bth.se/citation.cfm?id=1858583.1858597

[25] Mozilla Developer Network and individual contributors. (2017) Javascript reference - javascript — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

[26] R. S. Engelschall. (2016) Ecmascript 6: New features: Overview and comparison. [Online]. Available: http://es6-features.org/#ClassDefinition

[27] C. Dobre and F. Xhafa, "Parallel programming paradigms and frameworks in big data era," *International Journal of Parallel Programming*, vol. 42, no. 5, pp. 710–738, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10766-013-0272-7

[28] E. Czaplicki. (2017) Elm. [Online]. Available: http://elm-lang.org/

[29] R. Harrison, L. G. Smaraweera, M. R. Dobie, and P. H. Lewis, "Comparing programming paradigms: an evaluation of functional and object-oriented programs," *Software Engineering Journal*, vol. 11, no. 4, pp. 247–254, July 1996.

[30] M. A. Weiss, *Data Structures and Algorithm Analisys in C++*, 4th ed. Edinburgh Gate, Harlow, Essex CM20 2JE, England: Pearson, 2014.

[31] P. K. Stockmeyer. (1998, aug) Rules and practice of the game of the tower of hanoi. [Online]. Available: http://www.cs.wm.edu/~pkstoc/page_2.html

[32] (2017) Atom. [Online]. Available: https://atom.io/

[33] Node.js Foundation. (2017) Node.js. [Online]. Available: https://nodejs.org/en/

[34] npm Inc. (2017) npm. [Online]. Available: https://www.npmjs.com/

[35] (2016) Grunt: The javascript task runner. [Online]. Available: https://gruntjs.com/

[36] (2017) Babel: The compiler for writing next generation javascript. [Online]. Available: https://babeljs.io/

[37] (2016) line-count. [Online]. Available: https://atom.io/packages/line-count

[38] (2017, jan) Mocha - the fun, simple, flexible javascript test framework. [Online]. Available: https://mochajs.org/

[39] (2017, apr) Chai. [Online]. Available: http://chaijs.com/

A Appendix

In the following appendices, you will find our test cases and images describing the test data.

A.1 Test cases

All the images referenced in this appendix can be found in appendix A.2

Table 2: Binary search tree algorithms insert tests

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 16
ID	T1.1
Input	9, node(13)
Expected values	FP: None, OOP: See figure 16
Expected output	FP: See figure 16, OOP: false
ID	T1.2
Input	8, node(13)
Expected values	FP: None, OOP: See figure 17
Expected output	FP: See figure 17, OOP: true
ID	T1.3
Input	1, node(13)
Expected values	FP: None, OOP: See figure 18
Expected output	FP: See figure 18, OOP: true
ID	T1.2
Input	33, node(13)
Expected values	FP: None, OOP: See figure 19
Expected output	FP: See figure 19, OOP: true

Table 3: Binary search tree algorithms findNode tests

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 16
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)

Table 4: Binary search tree algorithms inOrderTraversal tests

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 16
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	[]

Table 5: Shellsort algorithm tests

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]

Table 6: Tower of Hanoi algorithm tests

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

Table 7: Dijkstra's algorithm tests

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 20, node1, node6
Expected values	None
Expected output	See figure 21
ID	T6.2
Input	See figure 20, node1, node4
Expected values	None
Expected output	See figure 22
ID	T6.3
Input	See figure 20, node2, node5
Expected values	None
Expected output	See figure 23
ID	T6.4
Input	See figure 20, 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

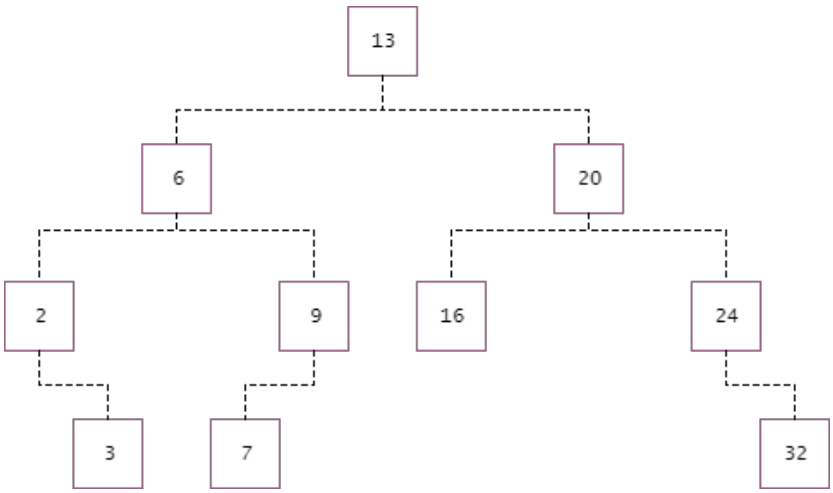

Figure 16: Input tree

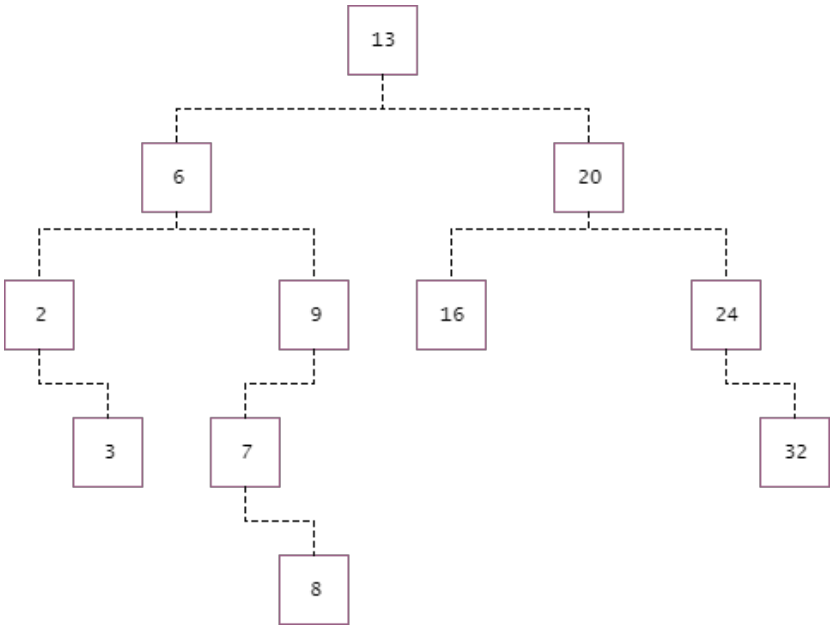

Figure 17: Output tree after inserting 8

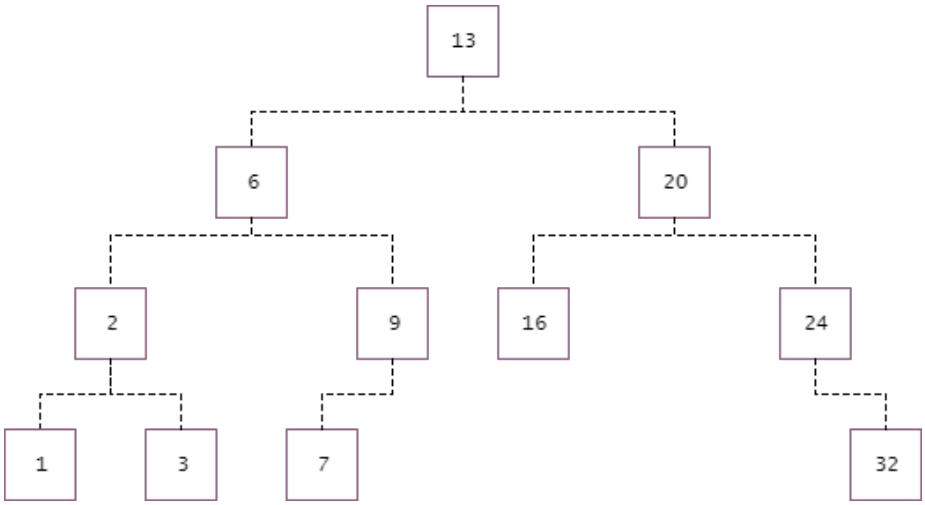

Figure 18: Output tree after inserting 1

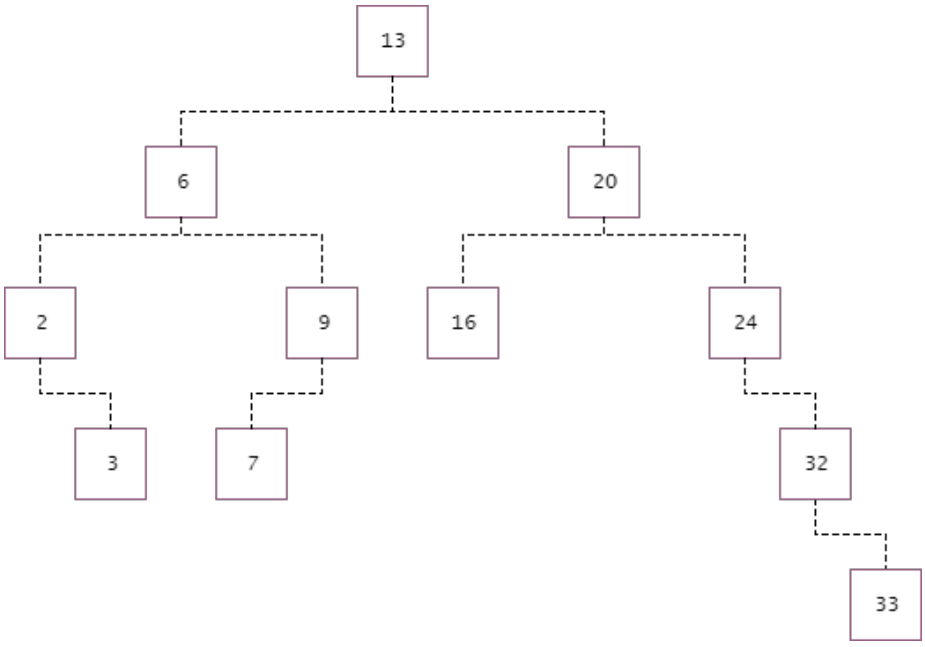

Figure 19: Output tree after inserting 33

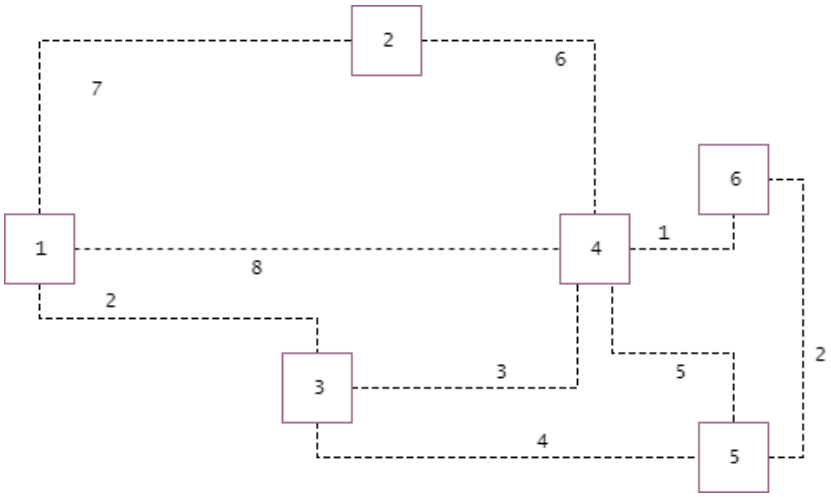

Figure 20: Input graph

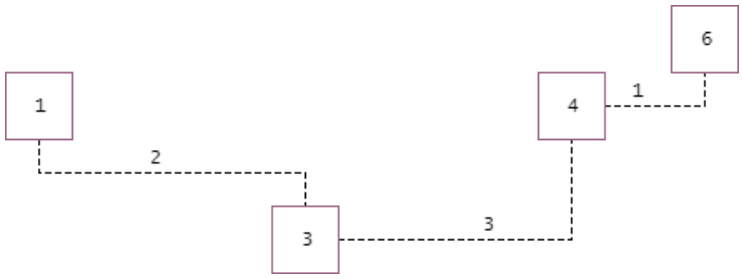

Figure 21: Output graph 1

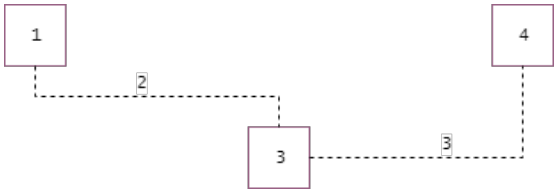

Figure 22: Output graph 2

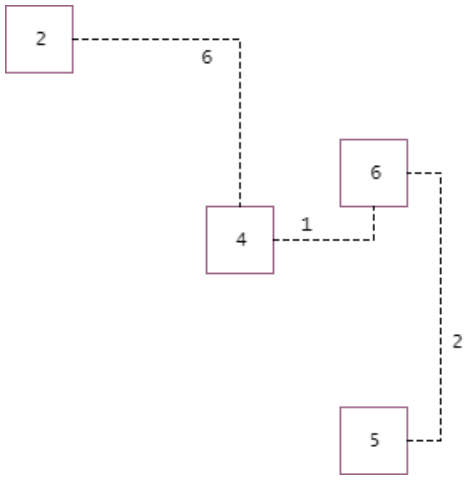

Figure 23: Output graph 3