Vrije Universiteit Amsterdam



Bachelor Thesis

**Verified Binomial Heaps in Lean 4**

**Author:** Jasper Abbes 2621491

*1st supervisor:* Jasmin Blanchette

*daily supervisor:* Jannis Limperg *2nd reader:* supervisor name

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

July 10, 2019 [fix]

Contents

[1. Introduction 3](#_Toc128924669)

[2. Background 4](#_Toc128924670)

[2.1 Binomial Heap 5](#_Toc128924671)

[2.2 Lean 4 6](#_Toc128924672)

[3. Definition of Data Structure 7](#_Toc128924673)

[4. Basic Operations 10](#_Toc128924674)

[5. Merging 11](#_Toc128924675)

[5.1 Combine 11](#_Toc128924676)

[5.2 MergeNodes 12](#_Toc128924677)

[6. Finding the Minimum 15](#_Toc128924678)

[7. Deletion 17](#_Toc128924679)

[8. Related Work 19](#_Toc128924680)

[9. Discussion and Conclusion 20](#_Toc128924681)

[References 21](#_Toc128924682)

[fix: check entire text on if it needs more paragraphs]

[fix: don’t forget to mention the error Jannis mentioned(email)] as an example of what can be found

[fix: in every section about proofs mention where full proof can be found along with the difficulty]

[fix: check if I use present tense everywhere]

[fix: maybe add : before mentioning pieces of code]

# 1. Introduction

[fix: find sources, maybe add some more on binomial heap]

Fix: instead of naming full section in intro use (Section 2)

Fix: maybe add explanation of the difference in using theorem and lemma.

Fix: maybe rephrase parts about theorem provers in the introduction and use different source

Data structures are a major and crucial part of software, they provide a way of saving data in a structured manner. Without data structures software development would become practically impossible. Data structures allow for the storage of data and they can be modified by using operations on them.[4] There is a great variety of data structures, from relatively simple ones such as a list, to more complex ones such as a binomial heap. The implementation details of a data structure can slightly vary per programming language. However, the properties of the data structure should always be met and consistent for any programming language. Data structures are the building blocks of software and getting the implementation right without an error can be challenging. Since data structures are such an important part of software, it is crucial to know if the implementation of these data structures is correct, this can be achieved by formally verifying the data structure. The verification of a data structure can lead to the detection of a bug in the implementation. Once a bug in the implementation is detected, it can be corrected by the community and subsequently a verification of the new implementation should be conducted.

Lean 4 is a functional programming language that can also be used as a theorem prover to perform formal verification [2]. Theorem provers can be split into two main categories, interactive and automated, both have some advantages and disadvantages [2]. Interactive theorem provers require every step to be fully verified and check if the given proof is indeed correct, while automated theorem provers tend to focus more on efficiency by helping the user find the proof with automated features.[2] The Lean 4 theorem prover tries to combine the best of both categories. With this research I want to verify that the binomial heap data structure in Lean 4 is correctly implemented. To establish this I used the Lean 4 language itself as a theorem prover. Along with the previously mentioned goal of this research I will also give my verdict on if the Lean 4 language is suitable to verify the implementation of data structures.

In the remainder of this paper there will be code snippets and figures with blocks of code. To make a distinction between normal text and the pieces of code, the pieces of code will have the following style: piece of code

To verify the binomial heap implementation I started by translating the properties of the binomial heap into the Lean 4 language. This was done by creating predicates that capture the properties of a binomial heap. After developing the predicates, it was essential to ensure that they were correct. The predicates are the centre of this research, a mistake in them would make the proofs that are conducted with them useless. Finally I had to proof that every instance of a binomial heap in Lean 4 has the type of these predicates. I created two main predicates, IsHeap and IsMinHeap, these predicates reference other predicates but this will be further explained in section 3. The properties that these predicates embody are not important at this stage so they will also be explained later. In Lean 4 there are multiple operations defined on the binomial heap, the operations that I will consider in this research are, empty, singleton, findMin, combine, merge and deleteMin. To verify the binomial heap implementation, I have to proof that if you have a binomial heap that has the IsHeap and IsMinHeap types and you perform one of the mentioned operations the output heap also has the IsHeap and IsMinHeap types. To achieve this I translated these problem statements into theorems in Lean 4. For each of the previously mentioned operations we thus get a theorem for IsHeap and IsMinHeap, only for deleteMin these two theorems were combined into one. For the findMin and deleteMin operations it was also necessary to proof that the operations actually return the minimum value from the binomial heap. For the findMin and deleteMin operations we thus have an extra proof. The resulting theorems had to be proven using the Lean 4 language itself, once this has been done we can say that the binomial heap implementation in Lean 4 is formally verified.

Before we continue, it is necessary to know that there are some limitations to the research conducted. With this research I only verify that every instance of a binomial heap in the Lean 4 language is indeed a correct binomial heap. To verify that the language also gives the correct binomial heap as an output we would need some additional membership predicate(s). These membership predicate(s) would need to make sure that every element in the binomial is the correct element given the input binomial heap. However, the benefits of creating these predicates along with the additional proofs would not outweigh the additional time it would take, so for the scope of this project I decided to not include this. If a full formal verification of the binomial heap in Lean 4 is necessary further research should be conducted. After this introduction I will share some background information on both the binomial heap data structure and the Lean 4 language. In the following section, Definition of data structure, I will explain the binomial heap data structure further with all the implementation details along with the created predicates. In the three sections that follow, Basic operations, Merging and Deletion, I will explain the details of the operations themselves along with some challenging or interesting parts of the proofs that belong to those operations. The section Basic operations has the same contents as the others but, as the name suggests, from multiple operations of which the proofs were significantly smaller than the other operations. After this I will share some related work in this area of research and to conclude this paper I will shed some more light on the limitations and give my conclusion.

[fix: in introduction mention where the implementation along with the proofs can be found]

# 2. Background

## 2.1 Binomial Heap

The binomial heap data structure is a data structure that is a heap consisting of binomial trees and can function as a priority queue [1]. A priority queue is the collective name for data structures that assign a certain priority value to each element in the data structure. Each element in the priority queue is served based on the priority value that is assigned to it [4]. Binomial heaps follow the heap property which has two variations: the minimal heap property and the maximal heap property. When the minimal heap property is implemented, the value of each node in a tree should be greater than its parent. The maximal heap property is the exact opposite, it guarantees that the value of each node in a tree is less than its parent. In the case of Lean 4 the minimal heap property is implemented and in the rest of this paper I will only discuss binomial heaps with the minimal heap property. The minimal heap property will be further explained in section 3.

The binomial heap was invented/developed by Vuillemin in 1978 as a result of the search for a data structure that has an efficient worst case time complexity for all of the priority queue operations. The priority queue operations include, insert, delete minimum and merge, the insert operation is not explicitly implemented in Lean 4. However, the insert operation is a combination of the singleton operation and the merge operation. If an element needs to be inserted, singleton can be called with the desired element and then merge can be called to merge the singleton with the original heap. In this paper an efficient time complexity will be considered to be at most O(log *n*) where n is the number of elements in the data structure. Use of relatively simple data structures like arrays, all fail to meet these set time requirements for one or more of the priority queue operations. There are various other data structures besides the binomial heap that were designed to meet the efficiency requirements for the standard priority queue operations, the majority of these data structures use a variant of heap-ordered trees. The challenge to meet the efficiency requirements mostly lies in the operations where the entire structure of these trees is altered, for example the merge operation. For the merge operation in particular it is impossible to achieve an efficient worst time complexity while using complete heap-ordered trees. However, altering the structure of complete trees would mean that they would become less balanced and it would lead to the trees losing some of the tree structure which is exactly what makes them useful. Binomial trees are not complete trees and thus less balanced. However, they are balanced enough to still capitalize on the useful tree structure. The structure and properties of the binomial trees will be explained in section 3. The use of binomial trees to form binomial heaps gives us a data structure that can perform all the priority queue operations in an efficient worst time of at most log n. Binomial heaps are a good well rounded option. However, depending on the goals there might be better data structures available because the performance of a data structure depends on the purpose [4]. One additional advantage of binomial heaps over other data structures is that each binomial tree in the heap can be viewed as a bit in a binary number. The number of elements in a binomial tree is always a power of 2, and since you can only have one tree of each of the power of twos in a binomial heap, this comparison is very useful for especially the merge operation. The number of binomial trees in a binomial heap always follows the number of 1 bits in the binary representation of the number of elements in the binomial heap. If you have a binomial heap with for example ten elements the binary number is 1010, the place of the 1’s also determines the number of elements in the particular trees. A binomial heap with ten elements thus always has one binomial tree with eight elements and one with two elements.

As with other data structures, implementing the binomial heap raises questions about implementation details. These details can be implemented to the developer’s likings. These choices do not necessarily influence the performance of the data structure or the way the data structure can be used. However, some of them could have an influence. Some of the choices made by the Lean 4 developers have a significant impact on this research, for example, the minimal heap property that was implemented. The choice for the minimal-heap property instead of the maximum heap property influenced the inductive predicates that were developed to be able to formally prove the data structure is correctly implemented.

## 2.2 Lean 4

As already mentioned before, I will discuss the verification of the binomial heap data structure in the Lean 4 language. In this section I will give a slight introduction into the Lean 4 language and where it originated from. A detailed description on the ins and outs of the Lean 4 language along with examples can be found in the Lean 4 manual. The Lean 4 manual can be found at [3]. In 2013, the lean project was started by Leonardo de Moura from Microsoft Research [6]. At this moment Lean 4 is the newest release of the Lean project. At the moment of this research, Lean 4 is not a stable language yet, meaning that the Language is updated frequently. Lean 4 is the successor of Lean 3 and it has many new features and improvements in comparison to Lean 3 [5]. An example of an improvement that was made is that with Lean 4 new extensions can be made to the language without changing the Lean source code. The Lean project is an open source project which allows the community to contribute to the language.

An important part of the Lean 4 language is the mathlib library which was started in Lean 3 by the Lean community [6]. The mathlib library contains mathematical components such as predefined lemmas/definitions. The mathlib library also is a major part of this research as multiple predefined lemmas were used, explicitly or implicitly by using the simplifier for example.

The Lean 4 language has two functionalities, it can be used as a functional programming language as well as an interactive theorem prover [2]. With a theorem prover objects can be defined using the theorem prover as a language. Ones an object is defined the properties that belong to that object can be translated into the language in the form of predicates. The predicates can thereafter be used to proof certain mathematical statements involving the defined object [6]. With a theorem prover we can thus not only verify mathematical statements but also other systems that can be translated into mathematical terms. Theorem provers can usually be divided into two categories: interactive and automated. Interactive theorem provers focus more on the fact that every step inside the proof is validated which (almost) guarantees that the proofs are sound. On the other hand, automated theorem provers try to optimize the process of proving theorems by having automated functionalities. Lean 4 is still primarily an interactive theorem prover. However, the stated aim of Lean 4 is to try and close the gap between the two categories by adding automated features [2]. New forms of automation can also be added to Lean 4 using the Lean 4 language itself [5].

The Lean 4 language depends on two major concepts: definitions and types. Definitions in Lean introduce new objects into the program [3]. A definition in lean has the following form:

def name (parameter : type) : returntype := body

Definitions in Lean can be used to declare constants as well as functions. Even theorems are a type of definition in Lean. Theorems are defined as definitions that have an output type of Prop [5].

In simple type theory every object has a type and new types can be derived from existing types, this is a powerful feature of type theory. Lean 4 uses dependent type theory which extends simple type theory by ensuring that types like Bool and Nat are seen as objects and thus have a type as well. As a consequence there is an infinite type hierarchy in Lean 4. The dependent type theory is called dependent because the type of an object can vary based on the type of its parameters [2].

# 3. Definition of Data Structure

[fix: split into more paragraphs especially the part about the predicates(one paragraph per predicate)]

In section 2.1, I have already briefly introduced the binomial heap data structure and the reasoning behind its development. In this section I will further explain the data structure along with all the implementation details and the inductive predicates that I created to capture the properties of a binomial heap.

A binomial heap is a heap consisting of binomial trees, I will thus start by explaining the properties of a binomial tree. The formal description is not very clear so I will describe this data structure in my own words. A binomial tree is a tree where the value of the parent node is always smaller than the value of its children, this is the case when the minimal-heap property is implemented. The minimal-heap property thus guarantees that the root node of the tree is the smallest node in the entire tree. The fact that in each binomial tree, the root node is the minimum value, is essential to the efficiency of some of the operations that can be performed on a binomial heap. The findMin operation for example only goes through all the root nodes in the heap and returns the lowest value. The other option would be the maximum-heap property which is the exact opposite and would guarantee that the root node has the biggest value in the tree. Each binomial tree has a rank, which is equal to the depth of the tree plus one. In Lean 4 the lowest rank a tree can have is one (singleton), this is the reason why the rank is not just the depth of the tree. I have seen other implementation of the binomial heap where the lowest rank is zero, which would make more sense. However, In this case, Lean 4 has a function defined on the binomial heap which returns the first and thus the lowest rank in the heap, this function returns zero when the heap is empty. The logical explanation is thus that to make a distinction between an empty heap and a singleton the ranks start from one instead of zero.

Since each node in the tree can be seen as a sub-tree, each node actually has its own rank. Additionally, the children (if any) of each tree should have the following ranks:

1, 1 + 1, 1 + 2, ..... , m-1

here m is the rank of the parent node. It is important to note that there can’t be any missing ranks in this order, this means that there has to be a child for each of these ranks.

A binomial heap consists of these binomial trees. For each rank ranging over the natural numbers, there can at most be one tree in the same heap. “At most”, because there does not have to be a tree for every rank. The order in which the trees occur in the heap is always increasing. The first tree in the list is thus the smallest, in other words, if [t₁, t₂,..., tₙ] is the list of trees then tᵢ < tᵢ₊₁ must hold for every value of i. These properties of the binomial heap ensure that the number of trees in a heap is at most log n and that the height is also at most log(*n*), where n is the number of elements in the binomial heap [1]. These are the properties that make binomial heaps useful and ensure that the priority queue operations are all efficient. The unique structure of the binomial trees makes the join operation relatively fast and also gives rise to the comparison to binary numbers as mentioned earlier.

When I started this project, the implementation of binomial heaps in the Lean 4 standard library used a list of heaps to represent the children of a node, this can be confusing as it makes the data structure more complex without adding any functionality. The more natural implementation would be to implement the children as one heap; this is because the children can be seen as trees themselves. In the original implementation it would mean that the children are a list of heaps but these heaps would all consist of exactly one tree. It thus makes more sense to implement the children as a single heap where each tree in this heap corresponds to a child. This change lead to some of the operations being simplified which also made it easier to subsequently formalize these operations

Figure . IsHeap predicates

mutual

inductive IsBinTree : BinTree α → Prop where

| mk: IsRankedTree 1 a.rank a.children.nodes → IsBinTree a

inductive IsRankedTree : Nat → Nat → List (BinTree α) → Prop where

| nil : IsRankedTree n n []

| cons : t.rank = n → IsRankedTree (n + 1) m ts → IsBinTree t → IsRankedTree n m (t::ts)

end

inductive IsHeapForest' : Nat → List (BinTree α) → Prop where

| nil : IsHeapForest' rank []

| cons : rank < t.rank → IsBinTree t → IsHeapForest' t.rank ts → IsHeapForest' rank (t::ts)

abbrev IsHeapForest : List (BinTree α) → Prop := IsHeapForest' 0

def IsHeap (h : Heap α): Prop :=

IsHeapForest h.nodes

Now let us look at the predicates that were created to capture the properties of the binomial heap. We start off with IsHeap and the predicates that IsHeap directly or indirectly references; they can be seen in Figure 1. IsHeap and its corresponding predicates together capture two properties of the binomial heap. The first property is the fact that the trees in the binomial heap are in increasing order, including that there can be at most one tree per rank. The other property is that the children of a binomial tree have the correct ranks and that these ranks are in the correct order. IsHeap takes as input a heap and the output is a proposition. The purpose of the IsHeap predicate is unfolding the list of trees from the heap and it calls IsHeapForest with that list.

IsHeapForest ts is an abbreviation for IsHeapForest’ 0 ts, where ts is the list of trees. The zero in this abbreviation is the lower bound for the ranks a tree can have, as mentioned before each tree should have a rank of at least one. IsHeapForest’ is the predicate that embodies the property that there can only be one tree per rank. Moreover, it also makes sure that the order in which these ranks occur is always increasing. IsHeapForest’ has two constructors, nil and cons. The nil constructor applies if the list of trees is empty and the cons constructor applies when the list is non-empty. The first argument of the cons constructor checks that the rank of the current tree is bigger than the rank of the previous tree. The zero used in the abbreviation makes sure that the first tree in the list is always at least one. The second argument of the cons constructor checks if for each tree in the list IsBinTree holds; IsBinTree is the predicate which we will discuss after we are finished with IsHeapForest’. The third argument is a inductive reference to IsHeapForest’ with the rank of the current first tree in the list and the current list of trees minus the first tree. As a result the rank of each child in the list is compared to the rank of the next tree in the list.

IsBinTree embodies the property that the ranks of the children of the tree that IsBinTree is called on, always starts with one and ends with the rank of the tree minus one. This is achieved by calling IsRankedTree with a fixed lower bound of one and the upper bound being the rank of the tree. IsBinTree has only one constructor which is mk. The constructor also only has one argument which is IsRankedTree 1 t.rank t.children.nodes where t is the binomial tree that was given to IsBinTree.

This brings us to the next predicate which is IsRankedTree. IsRankedTree is the predicate that checks if the ranks of the children of a tree are in the correct order and that they are all present. IsRankedTree has two constructors, nil and cons. The nil constructor again applies when the list of trees is empty, additionally to the list being empty the two other arguments need to be equal. The two natural numbers are the lower bound and the upper bound for the ranks of the children. If there are no children the lower bound set by IsBinTree and the rank of the tree will both be one and for that reason equal. The first argument to the cons constructor checks if the rank of the current child is exactly one higher than the previous child. The second argument checks that IsRankedTree holds for the next tree in the list. As mentioned earlier each node is a binomial tree; the third argument checks if this is indeed the case by referencing IsBinTree for each child. Because IsRankedTree refers to IsBinTree but also the other way around, IsRankedTree and IsBinTree are mutually defined.

We will now look at the IsMinHeap predicate along with the corresponding predicates. IsMinHeap and its corresponding predicates can be seen in Figure 2. The IsMinHeap predicate along with the corresponding predicates embody the minimal-heap property of the binomial heap, so they makes sure that every child has a bigger value than its parent. IsMinHeap has two constructors, nil and cons. The nil constructor again applies when the heap is empty. If the heap is empty, IsMinHeap will always hold. The cons constructor of IsMinHeap takes two arguments. The first argument is that IsSearchTree le n must hold, where n is the first tree in the list. le is a parameter that describes the less or equal relation between two values. In the case of a binomial heap, the values can be integers, letters or anything else which can be totally ordered. Because we do not know what kind of elements to expect, we need to get this relation through the parameter le. IsSearchTree is the predicate that will be discussed next. The second argument is that IsMinHeap le ns must hold, where ns is the current list of trees minus the first tree. What happens is that IsSearchTree le n must hold for each n in the list with trees.

IsSearchTree only has one constructor, this constructor is mk and it also has only one argument. The argument is IsMinTree le a.val a.children.nodes with a being the tree on which IsSearchTree was called. a.val is the value of the parent node of the tree, all the values in the remainder of the binomial heap have to be smaller than this value. a.children.nodes is the list with children of a.

This brings us to the next predicate, IsMinTree, IsMinTree is the actual predicate which embodies the minimal-heap property. IsMinTree has two constructors nil and cons. nil applies when the list with trees is empty in which case there were no children. The cons constructor takes three arguments, the first argument is le val n.val. With le being the less or equal relationship, val thus needs to be less or equal to n.val where val is the value of the parent and n.val the value of the first child in the list. The second argument is IsMinTree le val ns where ns is the list of children minus the previous first child. The effect that this second argument has, is that for every child the first argument must hold. The final argument is IsSearchTree le n with n being the first child in the list, this argument is necessary because the minimal-heap property must also hold for all the sub-trees. The IsSearchTree and IsMinTree predicates both reference each other which is why they are mutually defined.

The inductive predicates that are displayed are the center of this research and it is essential that they correctly embody the properties of a binomial heap.

mutual

inductive IsSearchTree (le : α → α → Bool) : BinTree α → Prop where

| mk : IsMinTree le a.val a.children.nodes → IsSearchTree le a

inductive IsMinTree (le : α → α → Bool) : α → List (BinTree α) → Prop where

| nil : IsMinTree le val []

| cons : le val n.val → IsMinTree le val ns → IsSearchTree le n → IsMinTree le val (n::ns)

end

inductive IsMinHeap (le : α → α → Bool) : Heap α → Prop where

| nil : IsMinHeap le (heap [])

| cons : IsSearchTree le n → IsMinHeap le (heap ns) → IsMinHeap le (heap (n::ns))

Figure 2. IsMinHeap predicates

# 4. Basic Operations

In the sections 4 up to and including 6, the different operations along with their proofs will be discussed. In this section the operations empty and singleton are discussed. The empty and singleton operations are not complicated operations and their proofs were also short and fairly straightforward.

We will start by looking at the empty operation:

def empty : Heap α

The empty operation, as the name already suggests, returns an empty heap. To verify that this operation was implemented correctly I created two theorems:

theorem IsHeap\_empty : IsHeap (@empty α)

theorem IsMinHeap\_empty : IsMinHeap le (@empty α)

The proofs for both of these theorems are trivial because an empty heap is by definition a correct binomial heap, so we can use the constructor tactic to finish both of these. The only difficulty with these theorems was that α needed to be given explicitly by using the @-modifier because otherwise Lean 4 could not know from the context what value to fill in.

The next operation we will look at is the singleton operation:

def singleton (a : α) : Heap α

The singleton operation takes one value a of type α and creates a binomial heap containing only the input value. To verify that this operation was implemented correctly I also created two theorems:

theorem singleton\_IsHeap : IsHeap (singleton a)

theorem singleton\_IsMinHeap : IsMinHeap le (singleton a)

The proofs of both of these theorems were again trivial because a heap containing only one element is by definition a correct binomial heap. For both proofs the constructor tactic was used which left some sub-goals that were trivial to solve.

# 5. Merging

In this section the combine and mergeNodes operation will be discussed along with the corresponding proofs. The combine operation is an operation that only involves binomial trees and is an important part of the mergeNodes operation. Because the combine operation cannot be used on binomial heaps and is only used as part of the mergeNodes operation, it will be discussed in a sub-section of the Merging section.

## 5.1 Combine

We will first look at the combine operation:

def combine (le : α → α → Bool) (n₁ n₂ : BinTree α) : BinTree α

The combine operation as already mentioned operates on binomial trees instead of binomial heaps. The combine operation takes two binomial trees of exactly the same rank and merges them into one binomial tree with a rank that is one higher than the ranks of the input trees. Because the input and output of the operation involves binomial trees, the theorems I proved to verify this operation use the predicates involving binomial trees. The two predicates that a binomial tree must satisfy are: IsBinTree and IsSearchTree. Both of the theorems created state that if the input trees satisfy the predicate then the output heap also satisfies to the predicate.

theorem combine\_trees\_IsBinTree (le : α → α → Bool) (a b : BinTree α) :

IsBinTree a → IsBinTree b → a.rank = b.rank → IsBinTree (combine le a b)

theorem combine\_trees\_IsSearchTree (a b : BinTree α) :

IsSearchTree le a → IsSearchTree le b → IsSearchTree le (combine le a b)

The proofs to these theorems were very similar to each other, they have the same structure and for both of them one lemma was created. The proofs for the combine operation were already a bit more challenging than the basic operations proofs but still relatively straightforward, that is why I will not spend too much time discussing them. For both of the proofs we unfold combine and split on the If/else statement that is present in the operation. The two goals that follow are mostly solved by applying the lemmas created, IsRankedTree\_append and IsMinTree\_append respectively. Both of these lemmas are solved by doing induction on the list with trees. In the combine\_trees\_IsSearchTree proof there was a small obstacle. One of the goals was le a.val b.val = true with one of the hypothesis being ¬le b.val a.val = true. As explained earlier le is a parameter that describes the less or equal relation. However, this relationship is not defined for this proof so the hypothesis not\_le\_le was created so that we are able to close this goal using the hypothesis.

## 5.2 MergeNodes

In this section the mergeNodes operation will be discussed along with its proofs. The actual operation is called merge but merge calls the mergeNodes operation with the lists with trees that are extracted from the heaps so the focus will be on the mergeNodes operation.

def mergeNodes (le : α → α → Bool) : List (BinTree α) → List (BinTree α) → List (BinTree α)

The mergeNodes operation merges two binomial heaps into one binomial heap or to be more accurate, it merges two lists of binomial trees that are extracted from binomial heaps. The mergeNodes operation can be seen in Figure 3. The mergeNodes operation uses pattern matching on the two input heaps to split on three cases, two of these cases are trivial. The first case is where the first input heap is empty, the output of the operation will then be the second input heap. The second case is when the second input heap is empty, the output will then be the first input heap. The final case of the pattern matching is the interesting case where both of the input heaps are non-empty. I will give a brief explanation on what happens in this case. We start by looking at the ranks of the first tree of both heaps, when they are not equal, the tree with the lowest rank will be placed in the output heap. With a recursive call to the mergeNodes operation with as arguments to the call both of the original input heaps minus the tree that got placed in the output heap. If the ranks of the first tree of each input heap are equal, they will be combined into one tree using the combine operation because there can only be one tree with a specific rank in a binomial heap. The combined tree has a rank of the original trees plus one. We now compare the rank of the combined tree to the ranks of the next trees in the binomial heaps. If the rank of the combined tree is not equal to either of these trees, the combined tree will be placed in the output heap and a recursive call is made to the merge operation with the original input heaps minus the first two trees that were merged. The same is done when both of the next trees in the input heaps have the same rank as the combined tree. When one of the ranks of the trees in the input heaps is the same as the combined tree we make a recursive call to merge with the original input heaps minus the first trees and with the combined tree prepended into the heap not containing a tree with that rank. The theorems that were created to verify the mergeNodes operation are:

theorem IsHeap\_merge (hxs : IsHeapForest' rx xs) (hys : IsHeapForest' ry ys) : IsHeapForest' (min rx ry) (mergeNodes le xs ys)

theorem IsMinHeap\_merge : IsMinHeap le (heap hx) → IsMinHeap le (heap hy) → IsMinHeap le (heap (mergeNodes le hx hy))

The IsHeap\_merge theorem was the most challenging and time consuming theorem to proof, not only because it was more difficult than for example the deletion theorems, but probably also because when I started this proof I still did not have much experience with the Lean 4 language. The strategy for these proofs was to follow the same structure as the mergeNodes operation has. In both of the merge proofs we thus use pattern matching on both of the lists with trees to split the proof into three cases. The first two cases are for the most part trivial. For these two cases in the IsHeap\_merge proof, I created three lemmas that will be used multiple times throughout the rest of the proofs. The first two lemmas empty\_heap\_merge\_left and empty\_heap\_merge\_right proof that if you merge an empty binomial heap with a non-empty one the resulting binomial heap is the same as the non-empty input heap. The third lemma IsHeapForest'\_weaken proofs that if you have IsHeapForest' n xs and m ≤ n then you have IsHeapForest' m xs. The third case of the IsHeap\_merge proof is where it gets interesting. I started by unfolding mergeNodes, which leaves us with multiple if/else statements which we then split by using the split tactic multiple times. After splitting multiple times we are left with multiple sub-goals. The length and difficulty of these sub-goals varies, some of them are fairly trivial while others are more complicated. The full proof is too lengthy to cover it all so I will discuss some interesting and challenging cases. The min function is an important part of this proof as it is stated in the theorem itself. While working on the proof min a b could be unfolded to if x ≤ y then x else y . However, after an update of Lean 4 unfolding min stopped working and I had to create a lemma which could be used to unfold min manually. This is a good example of how Lean 4 not being stable yet influenced this research. In the same sub-goal, an inductive application of the IsHeap\_merge theorem itself can be found. The sub-goal here is: IsHeapForest' (min h₁.rank (h₂.rank - 1)) (mergeNodes le t₁ (h₂ :: t₂))

This sub-goal is of the same form as the original goal after doing the pattern-matching. However, instead of having (h₁ :: t₁) as the left argument to mergeNodes, we now only have t₁. Only having t₁ means that the argument to mergeNodes got smaller, which allows for an inductive call to IsHeap\_merge. The two sub-goals that follow from this call were relatively straightforward. Throughout the rest of this proof there were multiple similar inductive calls to IsHeap\_merge. When I proved this theorem, Lean could not figure out that t₁ was the argument that should go at the place of xs in the theorem, so I had to give it explicitly. After checking the proof for errors I found out that this was no longer necessary because of an update to Lean. This was an example of how the Lean 4 language is constantly improving. Throughout all of the proofs the rename\_i tactic, which can be used to give inaccessible hypothesis a name, was only used when there was no other option. The rename\_i tactic can be considered bad practice as it becomes error-prone quickly. The split tactic left some inaccessible hypothesis that needed to be accessed and the rename\_i tactic was the only reasonable solution to this problem. A little further in the proof we see the first of many references to the rank\_combine lemma. The lemma is not complicated. However, it was definitely a useful one considering the amount of references to it. The lemma states that when you have two binomial trees of the same rank and you use the combine operation on them, the output tree will have a rank of the input trees plus one. A little further down, I used the theorem itself to get an induction hypothesis that looks somewhat like the current sub-goal. The current goal is IsHeapForest' (h₁.rank + 1) (mergeNodes le t₁ t₂) and the induction hypothesis we got is IsHeapForest' (min h₁.rank h₂.rank) (mergeNodes le t₁ t₂) . We first use the min\_hRank\_mergeNodes lemma to obtain the fact that min (hRank t₁) (hRank t₂) ≤ hRank (mergeNodes le t₁ t₂). For a better understanding of what this means I will shortly explain what the hRank operation does. hRank returns the rank of the first tree of the list with trees it receives. The min\_hRank\_mergeNodes lemma was a lengthy proof on its own but with many similar challenges to the IsHeap\_merge theorem, for that reason I will not go into the details of this lemma. After that, by\_cases was used to split on the case where t₁ and t₂ were both empty or not. The empty case was trivial but the non-empty case on the other hand was lengthy and challenging. The approach to the non-empty case was using the IsHeapForest'\_strengthen lemma which states that if you have IsHeapForest' rx ts and ry < hRank ts then IsHeapForest' ry ts holds. To finish the goal we used the induction hypothesis as the first argument which leaves us with one sub-goal: h₁.rank + 1 < hRank (mergeNodes le t₁ t₂) . Splitting on the min of min (hRank t₁) (hRank t₂) ≤ hRank (mergeNodes le t₁ t₂)we get two sub-goals that were solved using mostly predefined lemmas. The exact structure of the sub-goal previously described, also appears later on in the proof. The structure is the same. However, the sub-goals are slightly different. Repeating structures in the proofs like these occurred multiple times. In the next sub-goal we encounter another inductive call to the IsHeap\_merge theorem which leaves us two sub-goals. In the first goal nothing special happens but in the second sub-goal we encounter the IsHeapForest’\_weaken lemma which is the opposite of the IsHeapForest’\_strengthen lemma and is used multiple times throughout the proofs. The lemma states that if you have IsHeapForest' n xs and m ≤ n then IsHeapForest' m xs holds. As described earlier the overall structure is similar to the one we have already seen, but the sub-goals have some new lemmas which we will discuss. We have some hypothesis of the form (!h₁.rank + 1 == hRank []) = false , in this form the hypothesis cannot be used so it needs to be rewritten. Earlier in the proof we saw similar hypothesis. However those had true instead of false. The hypothesis with true can be rewritten using a predefined lemma but after an extensive search I could not find a lemma for the hypothesis using false. I created a lemma Bool.not\_eq\_false' which states ∀ (x : Bool), (!x) = false ↔ x = true , this allows us to use the hypothesis. In this sub-goal we also find a new lemma: hRank\_mergeNodes\_cons. This lemma states that if you have IsHeapForest' r (u :: y) , IsHeapForest' r (c :: z) and u.rank = c.rank then u.rank + 1 ≤ hRank (mergeNodes le (u :: y) (c :: z)) holds. In other words, if two lists of binomial trees are merged and the first trees in those lists have the same rank, then the rank of the first tree in the merged list has a rank of the original first trees plus one. This lemma was one of the more challenging lemmas to proof but it had similar challenges to IsHeap\_merge so I will not go into detail.

As discussed earlier the IsMinHeap\_merge proof also follows the structure of the mergeNodes function. The goals in this theorem were significantly less challenging than those in the IsHeap\_merge theorem. In this theorem it is visible that there is a repeating structure with all sub-goals being fairly straightforward to proof.

In these proofs and in fact in all the proofs, you might have noticed that in order to split on different cases, I switch back and forth between the cases tactic and the match tactic. The reason for this is that the cases tactic is more convenient in the cases where there is no naming necessary or where only the arguments to the constructor need naming. The match tactic has the advantage that in the cons cases you can name the otherwise inaccessible parts of the list. Using the match tactic thus leads to less uses of rename\_i .

Figure . mergeNodes

def mergeNodes (le : α → α → Bool) : List (BinTree α) → List (BinTree α) → List (BinTree α)

| [], h => h

| h, [] => h

| f@(h₁ :: t₁), s@(h₂ :: t₂) =>

if h₁.rank < h₂.rank then h₁ :: mergeNodes le t₁ s

else if h₂.rank < h₁.rank then h₂ :: mergeNodes le t₂ f

else

let merged := combine le h₁ h₂

let r := merged.rank

if r != hRank t₁ then

if r != hRank t₂ then merged :: mergeNodes le t₁ t₂ else mergeNodes le (merged :: t₁) t₂

else

if r != hRank t₂ then mergeNodes le t₁ (merged :: t₂) else merged :: mergeNodes le t₁ t₂

termination\_by \_ h₁ h₂ => h₁.length + h₂.length

decreasing\_by simp\_wf; simp\_arith [\*]

# 6. Finding the Minimum

In this section the findMin operation will be discussed along with the corresponding proofs. The findMin operation has three corresponding proofs because besides the proofs for IsBinTree and IsSearchTree , it is also needed to proof that the value of the returned tree is indeed the minimum value. The findMin operation:

def findMin (le : α → α → Bool) : List (BinTree α) → Nat → BinTree α × Nat → BinTree α × Nat

The findMin operation takes as input: the less than or equal relation, the list of trees from the binomial heap that still need to be assessed, the natural number that denotes the place of the current tree that is assessed and a tuple containing the tree with the current minimal value and the natural number denoting its place in the list. The findMin operation has two cases. The first case is when the list with trees is empty. In this case all the trees (if any) have been assessed and the tuple contains the tree with the minimal value and its id, this tuple is then returned. The second case is when the list with trees is non-empty. In this case, the value of the first tree in the list is compared to the current smallest value. If the value is not smaller than the one in the input tuple, findMin is recursively called with: the current list of trees minus the first tree, the current id plus one and the current input tuple. If the value of the first tree in the list is smaller than the value of the tree in the tuple, findMin is also recursively called. However, the input is now the same except for the tuple. The tuple is now the first tree from the list with the input id. These cases guarantee that from the trees that have already been assessed, the tuple always contains the tree with the minimum value. The two the theorems that proof that the findMin operation returns a correct binomial tree:

theorem IsBinTree\_findMin : ∀ hs idx h' idx', IsHeapForest' r hs → IsBinTree h' → IsBinTree (findMin le hs idx (h', idx')).fst

theorem IsSearchTree\_findMin : ∀ hs idx h' idx', IsMinHeap le (.heap hs) → IsSearchTree le h' → IsSearchTree le (findMin le hs idx (h', idx')).fst

Both of the proofs are fairly trivial since the findMin operation doesn’t alter any of the trees, meaning that if the input list was correct, the output tree should be correct too. The proofs both follow the structure of the operation, therefore there are two cases that follow from the pattern matching. The first case is trivial and in the second case an inductive reference to the theorem itself closes the goal.

To proof that the findMin operation actually returns the minimum, the following theorem was created:

theorem findMin\_is\_minimum : ∀ hs idx h' idx', le (findMin le hs idx (h', idx')).fst.val h'.val ∧ ∀ x ∈ hs, le (findMin le hs idx (h', idx')).fst.val x.val

The theorem states that given findMin le hs idx (h', idx'), h' and the value of each tree in hs should be no less than the value findMin returns. The idea to this proof is to split the proof into two parts, one for the left side of the ∧ and one for the right side. For both of these parts a lemma was created. For both of these lemmas I followed the structure of findMin, using pattern matching I thus obtained two cases. The findMin\_is\_minimum\_head lemma proofs the first part of the ∧. For this proof some additional hypothesis about the le relation needed to be defined, in particular the transitivity and reflexivity of the le relation. The first case of this proof is trivial. In the second case split is used to split on the two cases of findMin. The first goal is closed by an inductive reference to the theorem. In the second case we need a different approach because an inductive reference is not possible since the tuple in this goal contains t instead of h’. The solution to this problem is to create an induction hypothesis where the h’ gets replaced by t. This leaves us with a trivial goal. The findMin\_is\_minimum\_tail lemma proofs the second part of the theorem given the findMin\_is\_minimum\_head lemma. The first case of this lemma is trivial. In the second case, I used cases on t₂ ∈ t :: ts which results in two goals, one where t₂ is equal to t and one where t₂ is an element of ts. In both of the cases I split on findMin in both the goal and one of the hypothesis. The goals that follow were mostly solved by a reference to the findMin\_is\_minimum\_head lemma or an inductive call to the theorem. With the lemmas proved, the findMin\_is\_minimum theorem is just a matter of applying the lemmas correctly.

# 7. Deletion

In this section the deleteMin operation will be discussed along with its proof. For the deleteMin operation the proof for IsHeap and IsMinHeap was combined into one, as previously mentioned. The deleteMin operation:

def deleteMin (le : α → α → Bool) : Heap α → Option (α × Heap α)

The full deleteMin operation can be seen in Figure 4. The deleteMin operation thus takes a binomial heap as input and outputs an Option (α × Heap α). The option is necessary because if the input heap is empty the output should be none since we do not have a minimum nor an output heap. The deleteMin operation searches for the minimum in the heap and deletes it from the heap. The resulting binomial heap and the deleted minimum are the output in the form of an optional tuple. The deleteMin operation uses pattern matching to separate three different cases. The first case is when the input binomial heap is empty. When the input heap is empty the function simply returns none as there is no minimum. The second case is when there is only one tree in the heap. In this case, by definition of the binomial tree, the minimum is the root node of the tree. The output binomial heap without the deleted minimum, is the heap containing the children of the root node which was removed. The third case is where it gets interesting. In the third case we start of by calling findMin with the input heap. The findMin operation as discussed earlier, returns the tree which has the minimum value in the heap along with the id of the tree. The returned values are stored in separate variables. The tree containing the minimum value is subsequently deleted from the original heap, using the id to locate it. The resulting heap is then merged with the children of the minimum node, using the merge operation. The resulting heap along with the minimum value are then returned. The theorem created to verify the deleteMin operation:

theorem deleteMin\_non\_empty (h₁ : IsHeap xs) (h₂ : IsMinHeap le xs) : deleteMin le xs = some (y, ys) → IsHeap ys ∧ IsMinHeap le ys

This theorem assumes that the heap is non-empty. For the case where deleteMin returns none, I created a theorem, deleteMin\_empty\_IsHeap, that says that IsEmpty should be true for that heap. The proof for this is trivial and I will not discuss this further. The strategy for the deleteMin\_non\_empty proof was again to follow the structure of the operation itself. First we do pattern matching on the input heap so we get the same three cases as in the operation. The first case is the empty heap case which contradicts the assumption that the heap is non-empty so we can close this goal by contradiction. In the second goal the main challenge was to proof that the children of a node that satisfies the IsHeap and IsMinHeap predicates, also satisfy these predicates. Two lemmas were created to help solve this problem: children\_IsHeap and children\_IsMinHeap. For both of these lemmas I first had to find the correct order of cases that needed to be done. At this point it becomes clear that some form of induction will be necessary so I created an extra lemma for each proof, IsHeapForest'\_of\_IsRankedTree and IsMinHeap\_of\_IsMinTree respectively. These extra lemmas have a similar structure so I will only discuss the IsHeapForest'\_of\_IsRankedTree lemma. The lemma says if IsRankedTree r s nodes then IsHeapForest' (r - 1) nodes and r ≠ 0. Using pattern matching on IsRankedTree r s nodes, two cases are obtained, one per constructor. The nil case is trivial so we move on to the cons case. The goal is IsHeapForest' (r - 1) (t :: ts) , so we start by using the constructor tactic to obtain three sub-goals (one per argument to the constructor). The first two are trivial goals so we will move on to the third. The goal here is IsHeapForest' t.rank ts, notice that this is almost the same goal as the goal of the lemma itself. To make the inductive call to the lemma itself it is needed to replace t.rank with (r + 1) – 1, so the goals have the same pattern. Thereafter, the inductive call to the lemma itself can be made and this finishes the proof of the lemma. children\_IsHeap and children\_IsMinHeap both apply the lemmas previously mentioned after which they are finished. The application of these lemmas also closes the goals of the second goal of the deleteMin\_non\_empty theorem. In the third goal we have the following hypothesis: deleteMin le (Heap.heap (h :: hs)) = some (y, ys)

By unfolding the deleteMin and subsequently splitting on this hypothesis we get three new sub-goals, one for each case that we also encountered in the pattern matching at the beginning of this theorem. The first of these goals can be closed by contradiction since one of the hypothesis is: none = some (y, ys)which is a contradicting statement. In the second goal we encounter children\_IsHeap and children\_IsMinHeap again but besides those applications nothing interesting happens. In the third case findMin is unfolded and thereafter split is used to split into the two cases that findMin has. The first case can again be closed by contradiction since that is the empty case of the findMin operation and we are in the non-empty case of this proof. In the second case I use split again to split on the if/else statement of the findMin operation, which leaves us with two goals. Both of these goals follow the same structure and the approach to these proofs had many similarities, therefore I will only cover the overall structure of these cases ones. The main idea to these goals is to use And.intro to split the goal into a goal for IsHeap and a goal for IsMinHeap. The first of these goals is: IsHeap (Heap.heap (mergeNodes le ts₃ ts₄)) . This goal looks similar to the goal of the IsHeap\_merge theorem, the idea is thus to create hypotheses so that we can use IsHeap\_merge to close this goal. The IsHeap\_merge theorem uses the IsHeapForest’ predicate instead of IsHeap but this will not form an issue. To be able to use the IsHeap\_merge theorem, we need to establish the fact that ts₃ and ts₄ need to satisfy IsHeap. With have, we create two new subgoals, one for IsHeap (.heap ts₃) and one for IsHeap (.heap ts₄). In the first goal we split on the three cases of List.eraseIdx. In the first two of these cases, nothing new happens. In the third case the IsHeap\_delete\_BinTree lemma is used. The IsHeap\_delete\_BinTree lemma states that if IsHeap (.heap (a :: b)) then IsHeap (.heap (a :: List.eraseIdx b id)) . In the IsHeap\_delete\_BinTree proof a reference to the IsHeapForest'\_eraseIdx lemma is made. The IsHeapForest'\_eraseIdx lemma establishes the fact that if IsHeapForest' r a then IsHeapForest' r (List.eraseIdx a id). To proof this lemma pattern matching on IsHeapForest' r a is used to split on the two constructors. The first goal is straightforward and in the second goal an inductive reference to the lemma itself was made to finish the proof of this lemma. The IsHeap (.heap ts₄)goal only contains one new concept that I want to discuss. The min\_rank\_IsBinTree lemma states that if IsBinTree t then 0 < t.rank, this follows from the definition of the binomial tree. However, this lemma was not trivial to solve. The first challenge I encountered in this proof was the fact that I could not use cases on the following hypothesis: IsRankedTree 1 0 (Heap.nodes children) . The nodes operation should unwrap the list with trees from the heap but in this case when using the cases tactic, Lean could not solve the equation. To solve the problem I use generalize to create a place holder ts for Heap.nodes children. This lemma uses the rank\_zero\_IsRankedTree lemma to close the goal. The rank\_zero\_IsRankedTree lemma uses an inductive reference to itself to finish the proof. Now that IsHeap (.heap ts₃) and IsHeap (.heap ts₄) have been established, all that is left is altering the goal and hypothesis to fit the IsHeap\_merge theorem. The second goal of the And.intro has the same structure but now for IsMinHeap instead of IsHeap, for that reason I will now only discuss new or challenging facts for the rest of this proof. The IsMinHeap\_delete\_BinTree is the first new lemma that can be seen. This lemma is the IsMinHeap version of the IsHeap\_delete\_BinTree lemma, it has the same functionality as well as structure. The second part of this proof is almost identical to the first half, for that reason I will not discuss it any further.

The deleteMin operation should return the minimum value as mentioned earlier. To check if deleteMin indeed returns the minimum value I created the additional theorem:

theorem deleteMin\_non\_empty\_minimum : deleteMin le (.heap xs) = some (y, ys) → ∀ x ∈ xs, le y x.val

Proving that deleteMin returns the minimum in the first two cases is fairly trivial and easy. In the third case deleteMin uses findMin to find the minimum and I have already proven that findMin returns the minimum. Proving the third case is mostly a case of applying the findMin theorems and for that reason is not interesting.

The usage of the IsHeap\_merge and other theorems in this section also highlights the importance of the order in which the theorems were solved.

def deleteMin (le : α → α → Bool) : Heap α → Option (α × Heap α)

| heap [] => none

| heap [h] => some (h.val, h.children)

| heap hhs@(h::hs) =>

let (min, minIdx) := findMin le hs 1 (h, 0)

let rest := hhs.eraseIdx minIdx

let tail := merge le (heap rest) min.children

some (min.val, tail)

Figure 4. deleteMin

# 8. Related Work

In this section I will shed some light on related papers. The papers that I discuss here face similar challenges in regard to either the data structure that has to be verified or the theorem prover used to accomplish this.

Maybe: This paper discusses the verification of the binomial heap data structure implementation from the Java programming language. The verification language that was used is viper… First the binomial heap program is translated into the viper intermediate language before verifying it using the back-end verification system….

# 9. Discussion and Conclusion

With this research I have tried to verify the binomial heap implementation in the Lean 4 programming language, using Lean 4 itself as a theorem prover. Verifying a data structure like the binomial heap entails verifying that the data structure as well as the operations defined on that data structure are correctly implemented. Besides the verification of the binomial heap I also tried to make an assessment on how suitable Lean 4 is to verify a data structure like the binomial heap.

The main aim was to verify the binomial heap data structure so I will start by elaborating the results from that objective. I have verified the following operations of the binomial heap: empty, singleton, combine, merge, findMin and deleteMin. For each of these operation two theorems were created, except for findMin and deleteMin, for those an additional theorem was created. All these theorems were successfully proved, meaning that the binomial heap implementation in Lean 4 is indeed correct. With these findings we can now say the binomial heap in Lean 4 is verified. Since this research started the Lean 4 implementation of the binomial heap has changed significantly. The predicates and theorems do not align with the newer versions, meaning that the proofs in this research follow an older implementation that can be found here:[fix: url]. A note must be made that if an error was missed either in one of the predicate or in one of the theorem statements, the proofs would become invalid. Since I cannot rule this out with a hundred percent certainty, it is not guaranteed that the implementation is correct. One other limitation to this research is that although I established the fact that after each operation performed on a binomial heap, the resulting binomial heap has the correct properties, I did not proof that the resulting binomial heap had the correct elements. To establish this fact membership predicates could be made to embody the property that the correct elements of the input heap are present. However, this would not contribute much since it is for the most part fairly trivial that the operations return the correct elements. The advantages would thus not outweigh the time it would take to create this. The current version of the binomial heap implementation has been verified by a member of the Lean community and the membership predicates were also left out, indicating that it is a less important part of the verification.

I will now give my verdict on if Lean 4 is a suitable theorem prover for the verification of a data structure. I think Lean 4 is well suited for the verification of data structures. However, at this moment of time there are some disadvantages to using this language. At the moment of conducting this research, there is not a stable version of Lean 4. This means that the language was constantly changing and code can become obsolete quickly. The language being new also meant that there is not much information on the language readily available, making the learning process harder. After a rough start to the proofs and getting familiar with the language, the language became natural and pleasant to use. With this language I was able to translate all the properties of the binomial heap data structure into predicates without a problem. This indicates that the language is likely to be expressive enough to translate the properties of most data structures into predicates. Once I really got a grip of the language, the proving process became natural and enjoyable. The interactive part of the language ensures that all the steps that are made have to be proven. The feedback that the system gives for each step, immediately tells the user if it was correct or not, which helps the user obtaining a sound proof. The automated parts such as its extendable simplifier, simp, help speed up the proving process. The fact that you can move freely between tactic-mode and writing proof-terms is a nice feature of Lean 4. Although I mainly used tactic-mode because of the convenience it brings. The mathlib library is one of the main advantages of Lean 4. The mathlib library contained a lot of helpful predefined lemmas which I otherwise would have had to proof myself. My overall experience using the Lean 4 language was definitely positive. My verdict on if the language is suitable is by no means a confirmation that it is indeed suitable because my experience with theorem provers is still relatively limited.

[fix: is one of these a good conclusion?]

Although it is not fair to draw a definite conclusion to both of the questions because of the limitations to this research that were previously mentioned, this research might help further research in the right direction.

Although a definite answer to both of the asked questions cannot be given, I am fairly certain that the discussed implementation of the binomial heap is correct. The only fair answer that can be given on the question if the Lean 4 language is suitable to verify data structures is, based on this research there are slim to none complaints about the language itself which indicates that the language is indeed suitable for the verification of data structures.

# References

1. Algorithms, 3rd Edition, in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, and Searching. Robert Sedgewick, Addison-Wesley 2002. §9.7
2. Lean 4 tutorial
3. Lean 4 manual
4. Introduction to algorithms, third edition, Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein
5. The Lean 4 Theorem Prover and Programming Language, Leonardo de Moura, Sebastian Ullrich
6. <https://leanprover-community.github.io/index.html> [fix: if this should be reference]