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](#_Toc132272910)

[2. Background 4](#_Toc132272911)

[2.1 Binomial Heaps 4](#_Toc132272912)

[2.2 Lean 4 6](#_Toc132272913)

[3. Definition of the Data Structure 7](#_Toc132272914)

[4. Basic Operations 11](#_Toc132272915)

[5. Merging 11](#_Toc132272916)

[5.1 Merging Binomial Trees 12](#_Toc132272917)

[5.2 Merging Binomial Heaps 13](#_Toc132272918)

[6. Finding the Minimum 16](#_Toc132272919)

[7. Deletion 18](#_Toc132272920)

[8. Related Work 21](#_Toc132272921)

[9. Discussion and Conclusion 23](#_Toc132272922)

[References 25](#_Toc132272923)

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

# 1. Introduction

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. There is a great variety of data structures, from relatively simple ones such as lists, to more complex ones such as binomial heaps. The implementation details of a data structure can vary slightly per programming language. However, the invariants of the data structure should always be maintained by the implementation. Getting the implementation right without an error can be challenging. Because of the possibility that there is a bug in the implementation, it can be helpful to rule this possibility out. This can be achieved by formally verifying the data structure. The verification of a data structure ensures that any bugs that are present in the implementation are detected. If a bug in the implementation is detected, it can be corrected 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 (De Moura & Ullrich, 2021). Theorem provers can be split into two main categories, interactive and automated. Both of these categories have their advantages and disadvantages. Interactive theorem provers requires user interaction through almost every step of the proof. An interactive theorem prover can almost guarantee that the proof is indeed correct because it requires a specified proof. In contrast, automated theorem provers tend to focus more on efficiency by giving the users a full proof without any user input. Automated theorem provers leave more room for errors than interactive theorem provers. Lean 4 is an interactive theorem prover and so requires manual, explicit proofs by default, but it facilitates the development of proof automation tools. In this thesis I partially verify that the binomial heap data structure in Lean 4 is correctly implemented. To establish this I use the Lean 4 language itself as a theorem prover. In addition to performing the verification itself, I also investigate to what degree the Lean 4 language is suitable to verify the implementation of data structures.

To verify the binomial heap implementation I start by translating the invariants of the binomial heap into the Lean 4 language. This can be done by creating predicates that capture the invariants of a binomial heap. The predicates are the centre of this research since a mistake in them would make the proofs that are conducted with them useless. The predicates were thus carefully developed. I then prove that all the operations defined on the binomial heap maintain the invariants. Specifically I introduce two main predicates, IsHeap and IsMinHeap. I then consider the following operations: empty, singleton, findMin, combine, merge and deleteMin. To verify the binomial heap implementation, I prove that if a binomial heap has the IsHeap and IsMinHeap properties and one of the mentioned operations is performed on them, the output heap also has the IsHeap and IsMinHeap properties. 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. For the deleteMin operation these two theorems were combined into one. Additionally, I prove that the findMin and deleteMin operations actually return the minimum value from the binomial heap. In the case of deleteMin this returned value is also the value that is deleted from the heap.

The verification remains partial because I only verify that every binomial heap constructed by the mentioned operations 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 theorems. The theorems would need to state that the binomial heaps constructed by the operations contain the correct elements, given the input binomial heaps. These theorems could be formulated in several ways. An example could be that an extra predicate is developed for this. However, 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 desired further research is necessary.

In Section 2, I provide some background information on both the binomial heap data structure and the Lean 4 language. In Section 3, I explain the definition of the binomial heap data structure along with the created predicates that capture the invariants. In Section 4, I discuss some of the straigthforward operations together with their proofs. In Section 5, the operations that involve merging are discussed with the corresponding proofs. In Section 6, I consider the findMin operation along with its proofs. The final operation deleteMin, is discussed in Section 7 together with its proofs. After this I discuss some related verification projects, discuss the limitations of this thesis and give my conclusion.

The full binomial heap implementation along with the proofs can be found at: [fix]

# 2. Background

## 2.1 Binomial Heaps

The binomial heap is a data structure that is a heap consisting of binomial trees and can function as a priority queue (Sedgewick, 2002). 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 handled based on the priority value that is assigned to it (Cormen et al., 2009). In the case of a min-heap, this means that the element with the lowest priority value will be removed first. Binomial heaps maintain the heap property, which has two variations: the min-heap property and the max-heap property. When the min-heap property is implemented, the value of each node in a tree should be greater than its parent. The max-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 min-heap property is implemented and in the rest of this paper I will only discuss binomial heaps with the min-heap property. The min-heap property will be further explained in Section 3.

The binomial heap was introduced by Vuillemin in 1978 as a result of the search for a data structure that has a good worst case time complexity for all the priority queue operations (Sedgewick, 2002). These operations include insert, deleteMin 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 a good time complexity will be considered to be at most O(log *n*), where *n* is the number of elements in the data structure. Naïve implementations of priority queues, e.g. arrays, 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. Some of them indeed meet these requirements. 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 currently seems impossible to achieve an efficient worst time complexity while using complete heap-ordered trees (Sedgewick, 2002). 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 of binomial trees can be seen in Figure 1 where an example of a binomial heap is displayed. As can be seen in the figure, each tree has a rank. This rank is equal to the depth of the tree plus one. A binomial heap can have at most one tree per rank. Combining two trees of the same rank can be done in constant time because of the structure of these trees. To combine two trees of the same rank, the tree with highest value in the root node is placed as the rightmost child of the other root. This operation is the reason why the binomial heap has an efficient merge operation. The structure and properties of the binomial trees will be further 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 (Cormen et al., 2009).

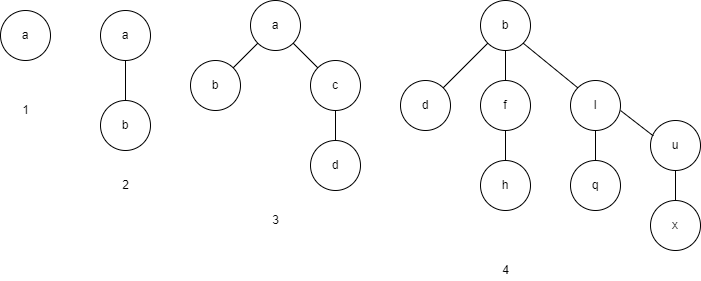


Figure 1. Binomial heap containing trees with ranks 1, 2, 3 and 4.

A nice property of the binomial heap 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 two. Since you can only have one tree of each of the power of twos in a binomial heap, inserting an element into the heap corresponds to the addition of one in binary addition. 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 liking. 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. Two of the choices made by the Lean 4 developers have a significant impact on this research. First, the min-heap property that was implemented. The choice for the min-heap property instead of the max-heap property influenced the inductive predicates that were developed to be able to formally prove the data structure is correctly implemented. Second, the children of a tree are in ascending order instead of descending order. This does not influence the performance of the data structure. However, it is interesting to see the differences between implementations.

## 2.2 Lean 4

I verified the binomial heap implementation in the Lean 4 language. In this section I give a brief introduction to the Lean 4 language and where it originated from. A detailed description of Lean 4 along with examples can be found in the Lean 4 manual.[[1]](#footnote-1) A tutorial on theorem proving in Lean 4 is also available.[[2]](#footnote-2)

In 2013, the Lean project was started by Leonardo de Moura. At this moment Lean 4 is the newest release of the Lean project. 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 (De Moura & Ullrich, 2021). The Lean project is an open source project, which allows the community to contribute to the language.

The most important Lean development is the mathlib library which was started in Lean 3 by the Lean community (The mathlib Community, 2020). It contains mathematical components such as predefined lemmas and definitions. The mathlib library is also a part of this research, as multiple predefined lemmas were used.

The Lean 4 language has two functions. It can be used as a functional programming language as well as an interactive theorem prover. In Lean’s functional programming language we can define data types and functions operating on them. Lean’s theorem proving component then allows us to verify these functions. To verify these functions, the correct invariants of the data type need to be translated into the language in the form of predicates. The predicates can thereafter be used to formulate theorems involving the data type. With a theorem prover we can thus not only verify pure theoretical 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 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. New forms of automation can also be added to Lean 4 using the Lean 4 language itself (De Moura & Ullrich, 2021).

The Lean 4 language revolves around two major concepts, definitions and types. Definitions in Lean introduce new objects into the program. 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 (De Moura & Ullrich, 2021). The Prop type stands for proposition; this means that objects of that type are statements that can only be true or false.

Besides universes and dependent function types, every type in the Lean library is an inductive type. Dependent function types are functions for which the type depends on a parameter to the function. For each inductive type there is a defined list of constructors. These constructors define the only possible ways in which an object of the inductive type can be constructed. The predicates developed for this thesis are also inductive types.

In the remainder of this thesis, recursion and induction are recurring themes. Some of the operations considered in this verification are defined recursively. This means that the operation makes a call to the operation itself. Induction is a common way of proving statements about recursively defined objects (including inductive types). As a result, the majority of the proofs in this thesis use some form of induction to prove the theorems. Proof by induction always starts by proving the base case which is usually for *n* = 0 or *n* = 1. Thereafter, it is assumed that the case for *n* = *k* holds, where *k* is an arbitrary number. Given this assumption, a proof for *n* = *k* + 1 is needed to conclude the proof. The notion of induction can be generalized to data types other than the natural numbers. For example, in this formalization induction is performed on lists. Lists in Lean 4 are inductively defined. In this case, the base case is for when the list is empty. The other proof is for when the list is non-empty. In the case of lists, the *n* from the explanation given about induction on the natural numbers, is the amount of elements in the list.

In Lean 4 there are two ways of constructing proofs, proof terms and tactic proofs. To use tactics, you can switch to tactic mode by using the by term. In Lean 4 you can switch back and forth between these two modes. The tactic mode was the only mode used for this thesis. While proof terms actually describe every step of the proof, tactics are instructions that tell Lean how to construct certain steps of the proof.

# 3. Definition of the Data Structure

In Section 2.1, I have already briefly introduced the binomial heap data structure and the reasoning behind its development. In this section I 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. In a binomial tree, the value of the parent node is always smaller than the value of its children. This is the case when the min-heap property is implemented. The min-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 has 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.

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. This is the reason why the rank is not just the depth of the tree. Such a tree can be created by the singleton operation. Other implementations of binomial heaps choose to use zero as the lowest rank. However, Lean 4 has a function defined on the binomial heap which returns the first and thus the lowest rank in the heap and which returns zero when the heap is empty.

Since each node in the tree can be seen as a subtree, each node actually has its own rank. This is relevant since the children (if any) of a node with rank *m* should have the ranks

1, 2, 3, ..... , *m* – 1.

It is important to note that there cannot be any missing ranks in this order, so there has to be a child for each of these ranks.

A binomial heap is a forest of binomial trees. For each rank, there can be at most one tree in a heap. However, there does not have to be a tree for every rank. The order in which the trees occur in the heap is always increasing by rank. The first tree in the list thus has the lowest rank. In other words, if [t₁, t₂,..., tₙ] is the list of trees, then tᵢ.rank < tᵢ₊₁.rank must hold for every *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 (Sedgewick, 2002). These are the properties that make binomial heaps useful and ensure that the priority queue operations are all efficient.

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. 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. The more natural implementation is to make the children collectively as one heap. 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. The Lean 4 definition eventually took over this simplification. However, the implementation has since been changed again. Now each node only links to one child and its sibling.

Figure 2. IsHeap predicates

mutual

inductive IsBinTree : BinTree α → Prop where

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

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 define the binomial heap, starting with the lower-level predicates. The predicates that define the binomial tree are thus discussed first since they are referenced by the higher-level predicates.

We start off with IsHeap and the predicates that IsHeap directly or indirectly references; they can be seen in Figure 2. The IsRankedTree predicate is the lowest-level predicate that is indirectly referenced by the IsHeap predicate, so that is our starting point. IsRankedTree ensures that 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 applies when the list of trees is empty. The two natural numbers are the lower bound and the upper bound for the ranks of the children. If there are no children (empty list), the lower bound set by IsBinTree and the rank of the tree are both one and hence equal. In addition to the list being empty, the two other arguments thus need to be equal. The first argument of the cons constructor ensures that the rank of the current child is exactly one higher than the previous child. As a consequence, this guarantees that the ranks of the children (if any) ascend from one to the rank minus one. The second argument ensures that IsRankedTree holds for the next tree in the list. As mentioned earlier, each node is a binomial tree; the third argument ensures that this is indeed the case by requiring IsBinTree for each child. Because IsRankedTree refers to IsBinTree but also the other way around, IsRankedTree and IsBinTree are mutually defined.

IsBinTree embodies the property that the ranks of the children of the tree that IsBinTree is called on always start with one and end 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. If a predicate has only one constructor, it can usually be written as a definition instead of an inductive type. However, IsBinTree is part of a mutual definition which prohibits this. 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.

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 ensures that the ranks are in ascending order. 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 ensures that the rank of the current tree is bigger than the rank of the previous tree. The zero used in the abbreviation ensures that the rank of the first tree in the list is at least one. The second argument of the cons constructor ensures that for each tree in the list IsBinTree holds. The third argument is a recursive reference to IsHeapForest’ with the rank of the first tree in the list as the first argument and the list of trees minus the first tree as the second argument. As a result the rank of each child in the list is compared to the rank of the next tree in the list.

The IsHeap predicate, is in contrast to the other predicates, defined as a definition instead of an inductive type. IsHeap takes as input a heap and the output is a proposition. The purpose of the IsHeap predicate is to extract the list of trees from the heap and it calls IsHeapForest with that list.

We will now consider the IsMinHeap predicate along with the corresponding predicates. IsMinHeap and its corresponding predicates can be seen in Figure 3. The IsMinHeap predicate embodies the min-heap property of the binomial heap: it ensures that every child has a bigger value than its parent. The le relation is a parameter to the IsMinHeap predicate that describes the less than 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.

Again we will start with the lowest-level predicate, IsMinTree. IsMinTree is the actual predicate which embodies the min-heap property. IsMinTree has two constructors, nil and cons. The nil constructor applies when the list of children given by IsSearchTree is empty. In this case IsMinTree holds trivially. The cons constructor takes three arguments, the first of which has type le val t.val. With le being the less than or equal relation, val thus needs to be less than or equal to t.val where val is the value of the parent and t.val the value of the first child in the list. The second argument is IsMinTree le val ts where ts is the list of children minus the previous first child. The effect of this second argument is that for every child the first argument must hold. The final argument is IsSearchTree le t with t being the first child in the list. This argument is necessary because the min-heap property must also hold for all the subtrees.

Figure 3. IsMinHeap predicates

mutual

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

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

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

| nil : IsMinTree le val []

| cons : le val t.val → IsMinTree le val ts → IsSearchTree le t → IsMinTree le val (t::ts)

end

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

| nil : IsMinHeap le (heap [])

| cons : IsSearchTree le t → IsMinHeap le (heap ts) → IsMinHeap le (heap (t::ts))

The IsSearchTree predicate will be discussed next. The IsSearchTree and IsMinTree predicates reference each other, they are mutually defined. IsSearchTree le t ensures that the tree t, has the min-heap property. IsSearchTree only has one constructor, mk, with one argument. The argument to this constructor is IsMinTree le t.val t.children.nodes with t being the tree on which IsSearchTree was called. Here t.val is the value of the parent node of the tree. All the values in the remainder of the tree have to be smaller than this value. The list with children of t is represented by t.children.nodes .

IsMinHeap has two constructors, nil and cons. The nil constructor again applies when the heap is empty. In this case, IsMinHeap holds trivially. The cons constructor of IsMinHeap takes two arguments. The first argument ensures that IsSearchTree le t holds, where t is the first tree in the list. The second argument ensures that IsMinHeap le ts must hold, where ns is the current list of trees minus the first tree. As a consequence, IsSearchTree le t must hold for each t in the list with trees from the binomial heap. This thus guarantees that each tree in the heap has the min-heap property.

# 4. Basic Operations

In the Sections 4 to 6, I discuss the different operations and their correctness proofs. 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 suggests, returns an empty heap. To verify that this operation was implemented correctly, I provide a proof of the following 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 is that the type α needs to be given explicitly by using the @ modifier because otherwise Lean 4 cannot know from the context what type to fill in.

We next consider 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 proved two theorems:

theorem singleton\_IsHeap : IsHeap (singleton a)

theorem singleton\_IsMinHeap : IsMinHeap le (singleton a)

The proofs of both of these theorems are again trivial because a heap containing only one element is by definition a correct binomial heap. For both proofs I use the constructor tactic, leaving some trivial subgoals.

# 5. Merging

In this section, the combine and mergeNodes operation are discussed along with the corresponding proofs. The combine operation is an important part of the mergeNodes operation. The combine operation merges two binomial trees, whereas the mergeNodes operation merges two binomial heaps.

## 5.1 Merging Binomial Trees

We first consider the combine operation:

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

As mentioned, the combine operation 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 greater than the ranks of the input trees. Because the inputs and output of the operation are binomial trees, the theorems I prove to verify this operation use the predicates for binomial trees. The two predicates that a binomial tree must satisfy are IsBinTree and IsSearchTree. We thus want to prove that if the input trees satisfy the predicates, then the output heap also satisfy the predicates:

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

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

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

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

The proofs of 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. 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 IsRankedTree\_append and IsMinTree\_append lemmas. The IsRankedTree\_append lemma has type

theorem IsRankedTree\_append (rt : IsRankedTree n m xs) (ha: IsBinTree a) (hrank: a.rank = m) :

IsRankedTree n (m + 1) (xs ++ [a])

The IsMinTree\_append lemma has type

theorem IsMinTree\_append (h : IsMinTree le m xs) (ha : IsSearchTree le a) (hba: le m a.val = true) :

IsMinTree le m (xs ++ [a])

Both of these lemmas are solved by doing induction on the list with trees. In the combine\_IsSearchTree proof there is a small obstacle. One of the goals is le a.val b.val = true with one of the hypotheses being ¬le b.val a.val = true. As explained earlier, le is a parameter that describes the less than or equal relation. Different types of elements can be ordered in more than one way. However, the le relation always has some properties that must hold. The not\_le\_le hypothesis describes such a property and is required to close the remaining goal.

not\_le\_le : ∀ x y, ¬ le x y → le y x

## 5.2 Merging Binomial Heaps

In this section the mergeNodes operation is 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)

Figure 4. 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 [\*]

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 4. It uses pattern matching on the two input heaps to split on three cases, two of which are trivial. The first case is when the first input heap is empty. The output of the operation is then the second input heap. The second case is when the second input heap is empty. The output is then the first input heap. The final case of the pattern matching is the interesting case where both of the input heaps are non-empty. 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 is placed in the output heap. The mergeNodes operation is recursively called. The arguments to this call are both of the lists of trees from the input. However, the tree that is placed in the output heap is removed from the list that originally contained this tree. If the ranks of the first tree of each input heap are equal, they are 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 is placed in the output heap and a recursive call is made to the mergeNodes 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 mergeNodes 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.

To verify the mergeNodes operation, I prove that the operation returns a correct binomial heap given that both of the input binomial heaps are of the correct form. This is translated into the following theorems:

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 prove. The strategy for both of the mergeNodes 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. Following the definition of the mergeNodes operation, the operation simply returns one of the input heaps (could be empty) in both of these cases. Given the assumptions made by the theorem that the input heaps are correct, the proofs of these cases speak for themselves. The third and final case of the IsHeap\_merge theorem is by far the most lengthy and challenging. This case contains many subgoals created by splitting on the if-else statements that are present in the mergeNodes operation. The length and difficulty of these goals varies. Some of the goals have similar structures to each other so I will only discuss one of them. After splitting, the goal in the first case is IsHeapForest' (min rx ry) (h₁ :: mergeNodes le t₁ (h₂ :: t₂)). Applying constructor leaves us with three subgoals, of which the first two are trivial. However, here we do encounter the Nat.min\_eq lemma. The min function is an important part of this proof as it is stated in the theorem itself. While working on the proof min x y 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 Nat.min\_eq , which could be used to unfold min manually. This is a good example of how Lean 4 not being stable yet influenced this research. The third goal of the constructor is IsHeapForest' h₁.rank (mergeNodes le t₁ (h₂ :: t₂)). The idea to solving this goal is to alter this goal to allow for an inductive reference to the IsHeap\_merge theorem itself. After altering, the goal that we are left with is the following:

IsHeapForest' (min h₁.rank (h₂.rank - 1)) (mergeNodes le t₁ (h₂ :: t₂))

This 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. This allows us to make an inductive call to IsHeap\_merge to close this goal. When I first 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.

Repeated use of the split tactic, which performs case analysis on the conditions of if-else statements, leaves us with multiple new goals. In the first case created by split, the goal is IsHeapForest' (min rx ry) (combine le h₁ h₂ :: mergeNodes le t₁ t₂). Two important hypotheses we have for this goal are IsHeapForest' rx (h₁ :: t₁) and IsHeapForest' ry (h₂ :: t₂). We start by using constructor to obtain three new goals. The first two of these new goals are relatively easy and contain nothing interesting besides the following lemma:

theorem rank\_combine : t₁.rank = t₂.rank → (combine le t₁ t₂).rank = t₁.rank + 1

Although this is a relatively simple lemma, it is a very useful one. It establishes the fact that the rank of two equally ranked trees combined, is the rank of the two input trees plus one. In the third goal, using rank\_combine we alter the goal to IsHeapForest' (h₁.rank + 1) (mergeNodes le t₁ t₂). We use the theorem itself to obtain the following induction hypothesis IsHeapForest' (min h₁.rank h₂.rank) (mergeNodes le t₁ t₂). This induction hypothesis looks already similar to the goal. We use by\_cases to split on the case where t₁ and t₂ are both empty or not. The empty case is trivial but the non-empty case is lengthy and challenging. In the non-empty case we apply the following lemma:

theorem IsHeapForest'\_strengthen : IsHeapForest' rx ts → ry < hRank ts → IsHeapForest' ry ts

Here the hRank is an operation that returns the rank of the first tree in the list. As the first argument we use the mentioned induction hypothesis which leaves us with h₁.rank + 1 < hRank(mergeNodes le t₁ t₂)as the goal. Using the lemma

theorem min\_hRank\_mergeNodes (ht₁ : IsHeapForest' r xs) (ht₂ : IsHeapForest' r ys) : min (hRank xs) (hRank ys) ≤ hRank (mergeNodes le xs ys)

we obtain min (hRank t₁) (hRank t₂) ≤ hRank (mergeNodes le t₁ t₂). Splitting on the min of this hypothesis we obtain two goals which were solved mostly by using predefined lemmas. The exact structure of the subgoal previously described also appears later on in the proof. However, the subgoals are slightly different. The next two goals that were created by previous splits have very similar goals and structures. The proofs to these goals were not trivial. However, they contained no new interesting concepts besides the fact that 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 have similar hypotheses. 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. The proof of the remaining case also has a structure that we have already discussed. However, in this subgoal we do find a new lemma that was needed to finish the goal:

theorem hRank\_mergeNodes\_cons (ht₁ : IsHeapForest' r (u :: y)) (ht₂ : IsHeapForest' r (c :: z)) : u.rank = c.rank → u.rank + 1 ≤ hRank (mergeNodes le (u :: y) (c :: z))

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 prove 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 subgoals being fairly straightforward to prove.

In these proofs, and in fact in all of 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 explicit arguments to the constructor need naming. The match tactic has the advantage that in the cons cases you can name the otherwise inaccessible implicit arguments. Using the match tactic thus leads to less uses of rename\_i , which is a tactic that can be used to name inaccessible hypotheses.

# 6. Finding the Minimum

In this section the findMin operation is discussed along with the corresponding proofs. The findMin operation has three corresponding proofs because besides the proofs for IsBinTree and IsSearchTree , I also prove that the value of the returned tree is indeed the minimum value. In this version of the binomial heap implementation, the findMin operation is only intended to be used as part of the deleteMin operation. The findMin operation therefore has a type that is relatively hard to comprehend.

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

The findMin operation can be seen in Figure 5. It 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 have been processed and the tuple contains the tree with the minimal value and its position in the list. This tuple is subsequently 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 position in the list 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 its position in the list. These cases guarantee that from the trees that have already been assessed, the tuple always contains the tree with the minimum value.

Figure 5. findMin

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

| [], \_, r => r

| h::hs, idx, (h', idx') => if le h'.val h.val then findMin le hs (idx+1) (h', idx') else findMin le hs (idx+1) (h, idx)

To prove that the findMin operation returns a correct binomial tree, I prove the following two theorems:

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 easy since the findMin operation doesn’t alter any of the trees. This means that if the input list with trees is 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 prove that the findMin operation actually returns the minimum, I prove the following theorem:

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'), the value findMin returns should be less than or equal to the value of h' and the value of each tree in hs. The idea of this proof is to split the proof into two parts, one for the left side of the conjunction and one for the right side. For both of these parts a lemma is created. For both of these lemmas I followed the structure of findMin, using pattern matching to obtain two cases. We will start with the findMin\_is\_minimum\_head lemma which proofs the left side of the conjunction of findMin\_is\_minimum:

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

For this proof we need to make additional assumptions about the properties of the le relation, it must be transitive and reflexive. The first case of the pattern matching in the findMin\_is\_minimum\_head 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 the goal is le (findMin le ts (idx + 1) (t, idx)).fst.val h'.val = true. Unlike the previous goal, 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. The goal can subsequently be closed by using the assumed properties of the le relation. The findMin\_is\_minimum\_tail lemma proves the second part of the theorem given the findMin\_is\_minimum\_head lemma:

theorem findMin\_is\_minimum\_tail : ∀ 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 first case of the pattern matching in this lemma is trivial. In the second case, I used cases on the hypothesis t₂ ∈ t :: ts which results in two goals. In the first goal t₂ is equal to t. In the second goal 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 these lemmas proved, the findMin\_is\_minimum theorem is just a matter of applying these lemmas correctly.

# 7. Deletion

In this section the deleteMin operation is discussed along with its proof. For the deleteMin operation the proof for IsHeap and IsMinHeap was combined into one. The deleteMin operation has type:

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

The full deleteMin operation can be seen in Figure 6. It 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. In this case 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. In the third case we start off 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 position of the tree in the list. These returned values are stored in two separate variables. The tree containing the minimum value is subsequently deleted from the original heap, using its position in the list that was returned by the findMin call. The resulting heap is then merged with the children of the minimum node, using the merge operation. The resulting heap and the minimum value are then returned.

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 6. deleteMin

To verify the deleteMin operation, we prove:

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

theorem deleteMin\_empty\_heap : deleteMin le xs = none → isEmpty xs

The deleteMin\_non\_empty theorem assumes that the heap is non-empty. For the case where deleteMin returns none, I created a theorem, deleteMin\_empty\_heap, that says that IsEmpty should be true for that heap. The proof for deleteMin\_empty\_heap is trivial. 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 to 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 is to prove that the children of a node that satisfies the IsHeap and IsMinHeap predicates also satisfy these predicates. The following lemmas solve this problem:

theorem children\_IsHeap : IsHeap (.heap [h]) → IsHeap h.children

theorem children\_IsMinHeap : IsMinHeap le (.heap [h]) → IsMinHeap le h.children

In the proofs for both of these lemmas it is necessary to use the cases tactic multiple times on different hypotheses and variables. At this point it becomes clear that induction will be necessary to finish the proof. To achieve this, I define an extra lemma for each of the proofs: IsHeapForest'\_of\_IsRankedTree and IsMinHeap\_of\_IsMinTree. These extra lemmas have a similar structure and function so I only discuss the IsHeapForest'\_of\_IsRankedTree lemma:

theorem IsHeapForest'\_of\_IsRankedTree (h₁ : r ≠ 0) :

IsRankedTree r s nodes → IsHeapForest' (r - 1) nodes

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 subgoals (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 goal fits the pattern of the theorem. Thereafter, the inductive call to the lemma itself can be made to finish the proof of this lemma. The children\_IsHeap and children\_IsMinHeap lemmas apply the IsHeapForest'\_of\_IsRankedTree and IsMinHeap\_of\_IsMinTree lemmas which concludes their proofs. The application of the children\_IsHeap and children\_IsMinHeap lemmas also finish 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 subgoals, 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 hypotheses 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 we use split again to split on the findMin operation. This leaves us with two goals. Both of these goals follow the same structure and the approach to these proofs has many similarities, therefore I will only cover the first case of this split . The main idea of the proof to this goal 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 satisfy IsHeap. With have, we create two new subgoals, one for IsHeap (.heap ts₃) and one for IsHeap (.heap ts₄). In the first of these goals we split on the three cases of List.eraseIdx. In the first two of these cases, nothing worth mentioning happens. After rewriting the third goal we create the following goal IsHeap (Heap.heap (a :: List.eraseIdx as n)). To solve this goal, we use the following lemma:

theorem IsHeap\_delete\_BinTree {id : Nat} : IsHeap (.heap (a :: b)) → IsHeap (.heap (a :: List.eraseIdx b id))

The only thing worth mentioning from the proof of the IsHeap\_delete\_BinTree lemma is the application of the following lemma:

theorem IsHeapForest'\_eraseIdx {id : ℕ} : IsHeapForest' r a → IsHeapForest' r (List.eraseIdx a id)

To prove this lemma, pattern matching on IsHeapForest' r a is used to split on the two constructors. The first goal is a straightforward proof. Besides an inductive reference to the lemma itself, the second goal contains no new interesting concepts. The IsHeap (.heap ts₄)goal only contains one new concept that I want to discuss, the following lemma:

theorem min\_rank\_IsBinTree : IsBinTree t → 0 < t.rank

This follows from the definition of the binomial tree. However, this lemma was not trivial to solve. The first challenge we encounter in this proof is the fact that we cannot use cases on the 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 cannot solve the equation. To solve the problem we use generalize to create a place-holder ts for Heap.nodes children. To finalize the proof of this lemma, the following lemma is applied:

theorem rank\_zero\_IsRankedTree : IsRankedTree (n + 1) 0 ts → False

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 created by the And.intro is IsMinHeap le (Heap.heap (mergeNodes le ts₃ ts₄)). The proof has the same structure as the proof for the previous goal, but now for IsMinHeap instead of IsHeap. For this reason I will now only shortly discuss one new lemma, IsMinHeap\_delete\_BinTree. This lemma is the IsMinHeap version of the IsHeap\_delete\_BinTree lemma and it has the same functionality as well as structure.

The deleteMin operation should return the minimum value as mentioned earlier. To check if deleteMin indeed returns the minimum value, we prove the following 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 easy. In the third case, deleteMin uses findMin to find the minimum and I have previously established that findMin returns the minimum. Proving the third case thus mostly involves applying the findMin theorems.

# 8. Related Work

In this section I discuss some related verification projects. The verifications that I discuss here face similar challenges in regard to the data structure that has to be verified and the theorem prover used to accomplish this.

In Meis et al. (2022) the binomial heap data structure is implemented and verified in the Isabelle proof assistant (Paulson et al., 2019). The binomial heap is defined in a similar way to the Lean 4 implementation. There are only two noticeable deviations; the first one being that the priority of a node is separated from the value of the node. In the Lean 4 implementation there is no separation between the two. The second deviation is that the lower bound of the rank of a binomial tree is zero instead of one. The predicates, or invariants as they are referred to in this formalization, are set up slightly differently to the predicates in this paper. In this formalization a binomial tree of rank *r* + 1 is defined to be composed of two binomial trees of rank *r* using the link (equivalent to combine) operation (except when *r* = 0). The tree invariant assumes that the link operation is correctly implemented. As a result, the predicates are less complex in comparison to the predicates in this thesis. The link operation guarantees (if correct) the min-heap property along with the property that the ranks of the children of a tree are correct. Using the link operation avoids having to explicitly define these properties. With the definition of the binomial tree in this paper, they basically combined the IsBinTree and IsSearchTree into one predicate. The rank-invar is similar to IsHeapForest’; They both ensure that the trees are in ascending order and that there can be at most one tree with a specific rank. The predicates in this paper are defined inductively while their predicates are defined recursively. The theorems are for the most part similar to the theorems defined in this paper. However, as opposed to this thesis, this formalization includes theorems that verify that after applying the operations the output heap contains the correct elements. There is no extra predicate for this but instead the tree-to-multiset operation was used to define these theorems.

In Appel (2022), the binomial heap data structure is verified using the Coq proof assistant (The Coq Development Team, 2022). The implementation that is verified has some significant differences in comparison to the Lean 4 implementation in this thesis. To start with, the binomial heap data structure uses the max-heap property as opposed to the min-heap property that is used in Lean 4. Additionally, the implementation views the trees in the binomial heap as power of two heaps instead of binomial trees. The right child of the root node in a power of two heap is always empty. The binomial trees and power of two heaps are equivalent. However, binomial trees follow the left-child right-sibling correspondence, so that they are represented in a slightly different way (Sedgewick, 2002). In the Lean 4 binomial heap implementation, the list containing the binomial trees does not leave empty spaces at the positions for which there is no tree with the corresponding rank, which in theory should be the case to make the comparison with binary numbers. The Coq implementation inserts a Leaf (empty tree) in those positions. In principle this does not affect the behavior of the data structure. However, the operations and predicates defined on the data structure have some differences because of this. Because this property is maintained there is no need to explicitly store the rank of a tree. The rank of a tree can be retrieved by looking at the index of the tree in the list. Although at first glance the predicates might seem to be quite different, the way in which they are set up is similar. The idea in both of these formalizations is to create some predicate that for every tree in the binomial heap the structure is correct and the min-heap property or max-heap property is satisfied. I decided to split the predicates into one that ensures the min-heap property and one that ensures the correct structure of the trees along with the binomial heap containing trees of ascending ranks. The Coq formalization combines everything into one predicate. The pow2heap' predicate basically has the same function as the IsRankedTree and the IsMinTree predicates combined, although, the way the structure of the trees is checked is different as a consequence of the trees having a different structure. The priq predicate is similar to the IsHeapForest abbreviation in that it sets the lower bound for the rank of the trees in the binomial heap. The priq' predicate is also similar to IsHeapForest’. However, because of the implementation differences, an additional check is needed to know if there is a tree or Leaf at each entry in the list

At the time of writing this thesis, the binomial heap implementation in Lean 4 has changed significantly compared to when I started this research. A partial verification of the new implementation is already available.[[3]](#footnote-3) This formalization, just like mine, does not cover the fact that the binomial heaps should contain the correct elements after applying these operations. The predicates used for this formalization have a different setup to the predicates in this thesis. The predicates do have some similarities to the verification in Coq that was previously discussed. However, because the implementation of the binomial heap differs, they are not exactly the same. The Lean 4 binomial heap formalization uses two predicates, Heap.WellFormed and HeapNode.WellFormed. These are both recursively defined, as opposed to the predicates in this thesis, which are inductive. Their Heap.WellFormed predicate is similar to IsHeapForest’, as they maintain the same property of the binomial heap. The HeapNode.WellFormed predicate is essentially the same as the [pow2heap'](https://softwarefoundations.cis.upenn.edu/vfa-current/Binom.html#BinomQueue.pow2heap') from the Coq formalization, although that one has some more cases because of the differences in the implementation. These predicates are comparable to each other, mainly because in both of these formalizations they view the binomial trees as power of two heaps.

# 9. Discussion and Conclusion

I have presented a partial verification of 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. I have verified the following operations of the binomial heap: empty, singleton, combine, merge, findMin and deleteMin. For each of these operations 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 part of the binomial heap implementation in Lean 4 that this verification covers is indeed correct. With these findings we can now say that the binomial heap in Lean 4 is partially 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.

A note must be made that if an error was made either in one of the predicates or in one of the theorem statements, the proofs would become invalid. Since I cannot rule this out with absolute certainty, it is thus not guaranteed that the implementation is correct. However, it is highly unlikely that this is the case since the error would be subtle enough for me to miss and the theorems would still be provable despite this error. 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 prove that the resulting binomial heap contains the correct elements. To establish this fact, membership predicates could be defined, enabling us to state the property that the correct elements of the input heap are present. However, the advantages would 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.

I will now briefly discuss whether I think Lean 4 is a suitable theorem prover for the verification of data structures. I think Lean 4 is well suited for the verification of data structures. However, currently there are some minor disadvantages to using this language. There is currently no 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 obtain a sound proof. Automation such as the extensible 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, though 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 contains a lot of helpful predefined lemmas which I otherwise would have had to prove myself. Besides the stated issues with the language being new, the only complaint I have about the Lean 4 language is that the performance of the system rapidly decreases as the proofs grow. I noticed this especially while I was working on the IsHeap\_merge proof, which is the largest proof in this development. Besides this issue, my overall experience using the Lean 4 language was definitely positive and there were no other major issues that I encountered.

Although a definite answer to both of the stated 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, from my experience there are few to no issues with the language itself, which indicates that the language is indeed suitable for the verification of data structures.

# References

Appel, A.W. (2022). Verified Functional Algorithms, Software Foundations, volume 3, Binomial Queues. Retrieved from <https://softwarefoundations.cis.upenn.edu/vfa-current/Binom.html>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms, third edition. MIT Press.

De Moura, L., & Ullrich, S. (2021). The Lean 4 Theorem Prover and Programming Language. Lecture Notes in Computer Science, 625–635. DOI: <https://doi.org/10.1007/978-3-030-79876-5_37>

Meis, R. Nielsen, F. Lammich, P. (2022). Binomial Heaps and Skew Binomial Heaps. Retrieved from [https://www.isa-afp.org/entries/Binomial-Heaps.html#](https://www.isa-afp.org/entries/Binomial-Heaps.html)

Paulson, L. C., Nipkow, T., & Wenzel, M. (2019). From LCF to Isabelle/HOL. Formal Aspects of Computing, volume 31, issue 6, 675–698. DOI: <https://dl.acm.org/doi/10.1007/s00165-019-00492-1>

Sedgewick, R. (2002). Algorithms in Java, Parts 1-4. Addison-Wesley Professional.

The Coq Development Team. (2022). The Coq Proof Assistant (8.16). Zenodo. DOI: <https://doi.org/10.5281/zenodo.7313584>

The mathlib Community. (2020). The lean mathematical library. CPP 2020: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, 367-381. DOI: <https://dl.acm.org/doi/10.1145/3372885.3373824>

1. https://leanprover.github.io/lean4/doc/whatIsLean.html [↑](#footnote-ref-1)
2. https://leanprover.github.io/theorem\_proving\_in\_lean4/title\_page.html [↑](#footnote-ref-2)
3. https://github.com/leanprover/std4/blob/5507f9d8409f93b984ce04eccf4914d534e6fca2/Std/Data/BinomialHeap.lean [↑](#footnote-ref-3)