Karlsruhe Institute of Technology

Specification and Verification of Hash Table Data Structures with KeY

Bachelor's Thesis by

Christian Jung

at the KIT Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	Dr. Mattias Ulbrich
Second reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Alexander Sebastian Weigl, M. Sc.

07/01/2021 – 07/05/2021

Karlsruher Institut für Technologie
KIT-Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that the work presented in this thesis is entirely my own. I confirm that I specified all employed auxiliary resources and clearly acknowledged anything taken verbatim or with changes from other sources. I further declare that I prepared this thesis in accordance with the rules for safeguarding good scientific practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, 07/05/2021

. . . . . .

(Christian Jung)

Abstract

This thesis is an evaluation of the specification and verification of two different hash table data structures with KeY. Hash tables are widely used data structures, because of their speed. We want to lay the ground work for the specification and verification of other hash tables data structures and Java programs that use a hash table. It is also a way to get new insight into the strengths and weaknesses of the verification process in KeY. We take already existing implementations of these two hash tables data structures and modified them for the verification. Then we look at how they could be specified in Java Modelling Language (JML). After that we see how well those specification could be verified in KeY and what problems we encountered.

Zusammenfassung

Diese Abschlussarbeit ist eine Evaluation der Spezifikation und Verifikation von zwei Hashtabellen-Datenstrukturen mit KeY. Hashtabellen sind weit verbreitete Datenstrukturen, da sie sehr schnell sind. Wir wollen eine Grundlage für andere Arten von Hashtabellen-Datenstrukturen und Java Programmen mit Hashtabellen schaffen. Damit können wir auch neue Einsichten über den Verifikationsprozess in KeY erlangen. Dafür nehmen wir bereits erstellte Implementation dieser zwei Hashtabellen und passten diese für die Verifikation an. Diese haben wir dann mit der Java Modelling Language (JML) spezifizieren. An schließlich haben wir diese in KeY verifizieren und die Probleme betrachten, die dabei entstanden sind.

Contents

List of Tables

1 Introduction

A hash table is a dictionary data structure, that can insert, delete and search data in constant time. These effects are more apparent when the memory space is large. Hash tables are widely used for their advantages, so it is of great interest to assess their quality in KeY and create a foundation for Java programs that use them. So we ask the question:

How well can hash table concepts be specified in JML and verified in KeY?

A hash table concept consists of a data layout, a hash function and a collision resolution strategy. A more detailed description of what a hash table concept, JML and KeY is follows in this chapter. When we write verification process, this means the process of specification in JML and verification in KeY together.

First we want to know if the concepts can even be specified and verified. Then we want to see what parts of the verification process are difficult. We also want to find their strengths and weaknesses in the verification process and find what parts of the specification can be reused for other concepts. This can get us new insight into the strengths and weaknesses of the verification process with KeY.

In this thesis we look at two different hash table concepts and provide the following for both:

- Four different Java Implementations
- A JML Specification for all Implementations
- A KeY proof (.proof files) for all finished Contracts.
- An Assessment, the Statistics and a Recommendation.

This can be found in the GitHub repository for this thesis, Jung, 2021. The statistics and some parts of the implementation and specification can be found in the appendix A.

Structure of the thesis. We start by explaining our Java implementations of the two hash table concepts in Chapter 2 and specify them with JML in Chapter 3. Along the way we explain some problems we encountered with this and how we fixed them. Then we verify the implementations against their specifications using KeY and collect statistics from the resulting proofs in Chapter 4. With those statistics we highlight difficult steps of the verification process in Chapter 5. The conclusion is in Chapter 6.

Java Modeling Language and KeY. The Java Modeling Language (JML) is a specification language for Java programs. It allows to define pre- and postconditions for methods and is based on the idea of design by contract. It was created in 1999 and is continuously developed by the community. Further information can be found at the website of Leavens, 2017.

The KeY System is a formal verification tool for Java programs. It verifies the program against it's formal specification, which is written in JML or Java Dynamic Logic (JavaDL). Proofs can be performed both automatically and interactively. This allows the user to skip repetitive parts of the verification and guide the system through difficult sections. The visual representation of the proof tree helps finding those sections and flaws in the specification. For further information about KeY and JML look at Ahrendt et al., 2016.

Related work. In parallel to this thesis there is another thesis about hash tables in KeY. It is written by Martin de Boer and is about verifying Identity Hash Map. His implementation is from OpenJDK and uses Linear Probing. He wants to find potential bugs in this implementation, while we want to create a foundation for hash table concepts in KeY. Since our works have similarities we had meetings together to help improve each other's work. He writes his thesis at the Open University of the Netherlands and their website is ou.nl

There is also a thesis about sequential and concurrent hash tables in Iris, a separation logic framework. It was written by Clausen, 2017 and called "Verifying Hash tables in Iris". The thesis builds upon a paper by Pottier, 2016 named "Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic". The implementation is written in the OCaml programming language and using CFML. CFML is a tool for the interactive verification, using Coq and Separation Logic.

1.1 Hash Table Concepts

The data layout builds the foundation of the hash table and is usually an array. We call a single data location in the data layout a bucket and every bucket of a single data layout has the same properties. This bucket can contain data directly or an extra data structure, like a linked list.

The hash function calculates a fixed-size value for any data of arbitrary size. For a hash table the hash values are the indexes of the used data layout. In Java the method `hashCode` can calculate a hash value for any object. This, plus an extra function to limit the hash value to the indexes, is often the hash function.

A collision occurs when two different pieces of data have the same hash value. In practices this is usually unavoidable. A collision resolution strategy handles such an event, by finding a new location, either inside the same bucket or in another one. Some collision resolution strategies are explained below.

1.1.1 Separate Chaining

The buckets do not contain the elements directly, instead they contain a data structure like in Figure 1.1. Such a bucket are called a chain. When two elements have a collision, both elements are placed in the same chain. Since we do not know in advanced where this happens, dynamic data structures are recommended, so constant resizing of chains does not cause slowdown. Some possible data structures would be:

- Linked List

- Self-Balancing Binary Search Tree
- Dynamic Array
- A second Hash Table

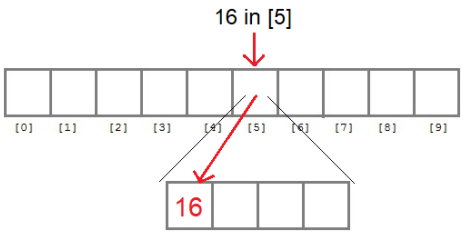

Figure 1.1: Separate Chaining. The element 16 is placed in chain [5].

1.1.2 Open Addressing

Each bucket of the hash table can contain no more then one entry. If a bucket already contains an entry, then an empty bucket needs to be found. We use the hash value as the start index and then a probing sequence to find the next index. We continue from the start of the array in a circular fashion. Some possible probing sequence are as following:

- Linear probing: The next bucket is a fixed distance away called interval.
- Quadratic probing: The distance to the next bucket increases in a quadratic fashion. (1, 4, 9, ...)
- Double hashing: The index of the next bucket is calculated by a second hash function.

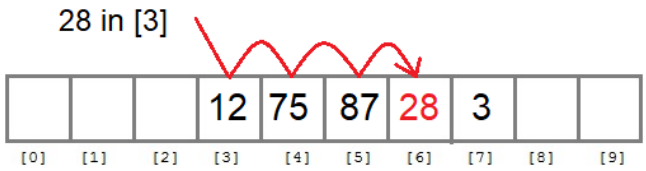

Figure 1.2: Linear probing with interval 1. The element 28 is supposed to go in bucket [3], but another element is already there. So the probing sequence is used to find an empty bucket. Buckets [4] and [5] are already full, so 28 is placed in [6].

Every time an element is placed, searched or deleted the hash table needs to search for the element and searching the whole hash table would be a linear run time. To avoid this, open addressing has an additional property. When searching for an element and an empty bucket is found, then the element is not in the hash table. This property has to be maintained by this strategy and must be considered when deleting an element in the hash table.

2 Implementation

In this chapter we explain our implementation. First we talk about the common behaviors between the two hash table concepts, then we look at Separate Chaining Array (SPA) and Linear Probing (LP) individually. In the last section we explain the changes we made to the implementation for the verification process.

We did not want to create a completely new implementation to avoid creating and solving our own problem. By using an implementation that is already used we can keep this evaluation more practical. Our implementation originates from the website of Sedgewick and Wayne, 2016. Before we even started we needed to change some aspects of this implementation. First KeY does not support Generics, so they were replaced by a class/type. The hash function was simplified and the methods `size`, `isEmpty` and `contains` got removed, as they were not necessary. Our implementation of Separate Chaining also does not use an extra class for the data layout. Instead we use two arrays, one for the keys and one for the values.

2.1 The Two Concepts

2.1.1 Common Behavior

The elements of the hash tables are pairs of keys and values. The data layout for both concepts are two arrays, but the bucket type is different for both. Two keys are considered equal based on the method of comparison, which we explain in the last section of this chapter.

The constructor receives an `int` as parameter, which is the length of the arrays and is set to 1 if it is smaller then 1. Then the variable for the array length, either `buckets` or `chains`, is set, the amount of elements is set to `0` and the data layout is initialized.

For both concepts the method `hash` receives a key. Then we get a `int` associate with the key and limit it to the range of indexes for the hash table with the module operator. Since in Java using the modulo operator on a negative number results in a negative number, we first need to guarantee that the `int` is not negative. For this we created the method `abs`. It returns the absolute value of a given number, but it is not strictly an absolute function. In Java there is no absolute value for `Integer.MIN_VALUE`, in this case `0` is returned. This method is almost identically for both concepts and the implementation (and specification) for SPA can be found in Appendix A.3.

Hash table implementations usually have a way to change the size of the hash table, if the amount of elements becomes to small or big. This could be done with a method and a `int` to track the amount of valid key-value pairs. Because of time constraints, we did not

include the method in our implementation. They are mentioned in this and later chapter, so from now on we call the method `resize` and the `int pairs`.

The `get` method returns the value associated with the given key in this hash table and returns a `null` if the key is not in the table. The `put` method inserts the given key-value pair into the hash table, overwriting the old value with the new value if the hash table already contains the given key. The `delete` method removes the given key from this hash table and therefore makes it's associated value inaccessible. This method does nothing if the hash table does not contain the key.

2.1.2 Separate Chaining Array

We used an array to keep the verification process more simple and in retrospect this was necessary as we will see in a later chapter. This hash table concept uses two two-dimensional arrays, one for the keys called `keys` and one for the values called `vals`. The `int chains` is the length of the first dimension of `keys` and `vals`. The elements of the first dimension are the chains and the elements of those, so the second dimension of the arrays, are the key-value pairs.

The method `hash` returns the index of the chain that can, but not necessarily does, contain the given key. This method is used in the `get`, `put` and `delete` method.

The parameters of the method `getIndex` is a key and its hash value. It returns the index for the chain of the given key, if the key is in this chain. Otherwise it returns `-1`. This method is also used by other methods to check if the hash table contains the key.

The method `increaseArraySize` receives a key-value pair and the hash value of the key as parameters. It then increases the size of the chain given by the hash value by one, both for `keys` and `vals`, and adds the key-value pair to this chain. The method `increaseAndCopy` receives an array and an element. It creates an array of the same type that is one space larger and copies all elements of the old array into the new one. It then adds the given element to the end of new array. This method is only used by the method `increaseArraySize`.

2.1.3 Linear Probing

This hash table concept has two arrays, one for the keys called `keys` and one for the values called `vals`. The `int buckets` is the length of `keys` and `vals`. It is necessary that the `keys` array always has at least one bucket that contains a `null`, which would normally be guaranteed by the `pairs` variable and the `resize` method. This implementation of Linear Probing uses an interval of 1.

The method `hash` returns the index from where we start the search for the key in `keys`. It is called by the methods `getIndex` and `findEmpty`.

The parameters of the method `getIndex` is a key. It returns the index of the given key, if the key is in the hash table. If we find a bucket with a `null`, then the key is not in the hash table and `-1` is returned. This method is also used by other methods to check if the hash table contains the key.

The `findEmpty` method works similar to the `getIndex` method, but it only stops when it finds a `null` and returns it's index. It is only called, when the key is not in the hash table.

Since there is at least one `null` in the hash table, the method always returns a valid index for the array keys. This index is the first one found by the probing sequence.

The method `overwritePair` inserts a key-value pair into the hash table. It also calls the `findEmpty` method to find a suitable position for the pair. It is called by the `put` method, when the given key is not already in the hash table. This simple method will be important in later chapters.

2.2 Model of Verification

In this section we will look at the changes we made to the original implementations to help the verification process. Let's start with some simple changes. First the methods `getIndex`, `increaseAndCopy`, `increaseArraySize`, `findEmpty` and `overwritePair` were introduced to make the proofs smaller and more manageable.

Both the `put` and `delete` method return a `int`, which is the position in the hash table that got changed by the method. This is not needed for normal use of the concepts and is just here to help the verification process.

The constructor for SPA uses a for-loop to initialize the second dimension of the arrays, all with the same length. Java does have a shortcut to do this in one command, but this still uses a loop internally. So it is necessary to have a contract for this loop for the verification.

The constructor for LP has an extra command that sets the first element of the `keys` array to `null`. This is the default state in Java and does not make any difference for the normal use of the class. But it does effect the verification of the constructor.

2.2.1 Keys and Variants

We started by using a new class for the keys called `HashObject`. This class has a `final int` variable called `value`, it is set by the constructor and is returned by the methods `getValue` and `hashCode`. These methods have no other effect and do the same. The `hashCode` method is a separate method so it can easily be changed, as it is used by the `hash` method of our hash table implementations. The `getValue` method is used in the `equals` method, seen Listing 2.1, of this class to compare itself to another object. Two keys are considered the same if they both are a `HashObject` and have the same `value`.

```
1 public final boolean equals(Object otherObject) {
2     if (!(otherObject instanceof HashObject)) return false;
3     return this.value == ((HashObject) otherObject).getValue();
4 }
```

Listing 2.1: equals method

Here we ran in a problem, as the verification of the `put` and `delete` methods, including some sub methods, caused difficulties with the verification in KeY.

So we changed the strategy, instead of using `HashObject` for keys and `Object` for values we used the `int` data type. To compare two keys we used the equality operator. With those changes we were able to finish the verification and could improve the specification.

After we finished the `int` variant of the concepts we added two additional variants to get a better idea where the problems occur. This resulted in four different variants and so

eight Java files in total. Those variants only effect the types for the keys and values and how we compare two keys. The four variants are as follows:

- WithInt (WI): Keys and values are `int` data types and it uses the equality operator to compare keys.
- WithIntKeys (WIK): Same as WI, but values are now of the `Object` class.
- NoEqulas (NE): Keys have the `HashObject` class, values the `Object` class and it uses the equality operator to compare keys.
- WithEquals (WE): Same as NE, but it uses the `equals` method to compare keys.

Our assumption was that WI is the simplest to verify, then WIK, then NE and WE is the hardest.

Our implementation of the hash table concepts use the `null` keyword in some circumstances and the `int` data type does not have this. So we use `0`, called `iNull` in the code, as our `null` for `int` and a key that has this value is invalid.

Also the method `increaseAndCopy` in SPA now has three different versions depending on the variant. One for the `int` data type, the `HashObject` class and the `Object` class.

2.2.2 Delete

Even with the WI variant the `delete` method was difficult to verify, so the methods got simplified for both hash table concepts.

Originally for Separate Chaining Array the size of the chain was decreased by one every time a key is removed from the table, to keep it more similar to a linked list. This was done by creating two new arrays and copying all not deleted key-value pairs into the new arrays. Now instead we overwrite the key with a `null`.

Normally when deleting a key-value pair from the hash table with Linear Probing the key and value is set to `null`. This new `null` could be between any key and its hash value, which would mean that the key cannot be found anymore. It is therefore necessary to reorganize the hash table after deleting a key-value pair. This would mean removing all affected key-value pairs, keeping in mind this could cause other pairs to be affected with the same problem, and then reinserting them with the `put` method. Instead of a `null`, we set a deleted key to a dummy key. This dummy key is `final`,is always the same and an exception is thrown if it is a parameter for `get`, `put` or `delete`. This makes this key inaccessible and so effectively deleting the previous key-value pair at this position. Our implementation cannot overwrite this dummy key, because allowing this caused problems with the verification of the `overwritePair` method.

2.2.3 Linear Probing

For the methods `findEmpty` and `getIndex`, when searching for the key, it is possible that the end of the array is reached and we continue the search from the beginning of the array. To do so it is common to use the modulo operator like in Listing 2.2.

```
for (i = hash(key); keys[i] != null; i = (i + 1) % buckets)
```

Listing 2.2: modulo loop

Here we instead use two different for-loops as seen in Listing 2.3. The first goes from the hash value to the end of the array and the other starts at 0 and ends before the hash value. This point is never reached, since the array always contains a `null`. This is done to simplify the specification, which will be explained in the next chapter.

```
for (int j = hash(key); j < buckets; j++) {...}
for (int k = 0; k < hash(key); k++) {...}
```

Listing 2.3: divided loops

Even thought it is not possible to reach end of the second loop, at the end of both methods is a return statement. This helps the verification of both methods.

3 Specification

In this chapter we go into detail on our specifications of the two hash concepts. First we talk about specifications that both hash concepts have in common. And then we look at the specifications unique to the two hash table concepts. The `normal_behavior` is the standard contract name and every method has this. The `get`, `put` and `delete` method for all 8 variants have also an `exceptional_behavior`, for when exceptions occur. We're focusing on the WI variants. but the changes between the variants are straightforward so they will be listed here. If the specification uses the type of a variable, then it is changed according to the variant. When two keys are compared the WE variant uses the `equals` method, instead of the equality operator.

3.1 Common Specifications

All keys and values as parameters for methods must be non-null for their `normal_behavior`. For objects this is the default, so there is no `requires` clause in the code in this case. This clause is the precondition of the contract and can be used in the verification. The `ensures` clause is the postcondition to the contract and the counterpart to the `requires` clause. The verifying the contract of a method the `ensures` clause must hold after the methods execution. If a method with a `requires` is called by another method, the verification has to proof that the conditions are met when the call happens. It then can use the `ensures` clauses of the called method to help it's verification.

We did specify and verify the constructor as a starting point for our proof. A completed proof for any contract of a method means the invariants hold before and after it's execution. But with the constructor we can show that they even hold from the very beginning. The contract itself requires that the given value, the length of the hash table, is at least 1. The constructor must ensure that `keys` and `vals` are new arrays, `pairs` is 0 and the length of the hash table is the given length.

3.1.1 Class Invariants

All class invariants are included in the Appendix, the ones for SPA are in A.1 and the ones for LP are in A.2. The arrays `keys` and `vals` are always non-null and the size is bigger then 0. If one of these was not true, the table could not contain any elements and first call of the `put` method would call the `resize` method to fix this. The arrays `keys` and `vals` are equally long, since keys and values always come in pairs. The arrays `keys` and `vals` are different arrays to avoid aliasing problems. The arrays `keys` and `vals` are not subtypes. This is important for verification, since otherwise the type of array could still be

a different subtype then an element that is written into the array. It uses the expression `\typeof(x) == \type(T)` , where x is the array and T the array type.

3.1.2 Hash Method

The `hash` method is `strictly_pure` and a `helper` method. A method with the `helper` modifier does ignore the class invariants. A method that is `pure` has no side effects, which allows it's use in contracts. It forces all `assignable` clauses of the contracts of this method to have the `\nothing` trait. A `strictly_pure` method is same, but `\nothing` is instead `\strictly_nothing`. The `assignable` clause describes what parts of the heap can be changed by the method. The trait `\nothing` means it can only create and effect it's own objects and `\strictly_nothing` also restricts this, so it can only create and change it's own primary data types.

The `hash` method also has a `accessible` clause, so it can only read the table length variable and the value of the key, if we use the NE or WE variant. The method uses an `ensures_free` clause that says, this method always has the same result when the given key is the same. But to use this clause we need to prove that this is true outside of KeY. Since a `int` key uses itself and the `value` variable of a `HashObject` is `final`, the `int` associated with the given key is always the same. The `abs` method and modulo operator are self-explanatory.

When we're using the `equals` method to compare two keys, it is important that two equal keys also have the same hash value. Different hash values would mean that the two keys have different chains (SPA) or start positions (LP) which could cause duplicate keys in the hash table. The variants that use the equality operator still use this specification, since helps the verification in some instances.

3.2 Separate Chaining Array

The specification for the WI variant has 9 class invariants, 14 `requires` clauses, 20 `ensures` clauses, 3 `loop_invariants`, 5 `assignable` clauses (not counting any with (`\strictly_`) `\nothing`), 27 quantifiers and 9 nested quantifiers. The other variants only have minor differences.

The constructor for SPA is more complex since it uses a for-loop. It uses a `loop_invariant` and the conditions set by it need to hold when entering, repeating or leaving the loop. The `loop_invariant` needs to guarantee that all invariants for the second dimension of keys and vals hold, so they can be proven at the end of the method. What those are is mentioned in the Class Invariants subsection below. The method must ensure that all arrays of the second dimension are new.

3.2.1 Class Invariants

All the class invariants in the common sections, except the bigger then 0, also apply to the second dimension of the keys and vals arrays. The no subtype specification has a special

scenario here. When the type of an array is the array of a primitive data type, e.g. `int[]`, it still needs to be specified, since arrays are still objects.

Each valid key can at most be once in the same chain as seen in Listing 3.1. This is limited to chains and not the whole hash table, because another invariant was supposed to limit each key to its correct bucket. Correct means that the first index in the two-dimensional array `keys` is the hash value for each key, since this is the result of the hash function.

```
(\forall int x; 0 <= x && x < chains;
    (\forall int y; 0 <= y && y < keys[x].length && keys[x][y] != iNull;
        (\forall int z; y < z && z < keys[x].length && keys[x][z] != iNull;
            keys[x][z] != keys[x][y])));
```

Listing 3.1: The "at most once" invariant for SPA

But using both invariants caused problems when verifying the method `increaseArraySize`. This problem even occur when the “correct bucket” invariant was just a `requires` clause. There are similar problems when using two other invariants. The first was "no key or value can be `null`" and the other was "if a key is not `null`, then the value is also not `null`". They all have in common that they look at every element in the two-dimensional array. The correct bucket invariant can be seen in Listing 3.2 as an example.

```
(\forall int x; 0 <= x && x < chains;
    (\forall int y; 0 <= y && y < keys[x].length;
        x == hash(keys[x][y])));
```

Listing 3.2: The "correct bucket" invariant

Since `null` keys and values are allowed, the clause “if a key is not `null`, then the value is also not `null`” is necessary for the `get` method. But as mentioned above this clause caused problems with other invariants, so we limited it to a `required` clause for the `get` method.

3.2.2 Private Methods

The method `getIndex` is `strictly_pure`. The method requires that the given hash value is a valid index for the array `keys`. What this method must ensure depends on the result, but is limited to the chain given by the hash value. If the result is `-1` the given key is not in the chain and if it is not `-1` the result is a valid index of the chain and is the position of the key. The `loop_invariant` guarantees that every position in the array until now does not include the given key. This is always true, since the method terminates if the key is found.

The method `increaseArraySize` requires that the given hash value is a valid index for the array `keys` and the given key is not an element in the chain given by the hash value. The latter part is done with the `getIndex` method. This method can only change the chains and the `pairs` variable. It must ensure that the new chains are still the correct type, their length increased by one, they contain all the old key-value pairs in the same order and the new key-value pair is the last element of the chains.

The method `increaseAndCopy` is a `helper` method and has `\nothing` as `assignable` trait. This method must ensure that it creates a new array that has the same type as the given array and the length of the array must be increase by one. The array must contain all old elements in the same order and the new element is the last element of the new array. The `loop_invariant` guarantees that every position in the new array until now contains all the

elements of the given array in the same order. The two variations, `increaseAndCopyKeys` and `increaseAndCopyVals`, have additional properties. Since the arrays can contain a `null` they need to be `nullable`, but the given and resulting array cannot be `null`.

3.2.3 Public Methods

The method `get` is a `pure` method. First we talk about the `normal_behavior`. The `assignable` clause has `\strictly_nothing`as trait. As mentioned above, this method requires, that every non-null key is paired with a non-null value. What this method must ensures depends on the result, but is limited to the chain of the hash value for the given key. If the result is a `null`, the key is not in the chain and if it is not `null`, the key is in the chain with the index of the hash value of this key and the result is at the same position in `vals`. The method is only `pure`, because an exception creates an object. For the exception to occur the given key must be `null`.

The `put` method can change the chains of the hash value for the given key, the entries of the same `vals` chain and the `pairs` variable. The method must ensure that this chain contains a key that is equal to the given key and the given value is at the same position in the `vals` chain. The first index for `keys` and `vals` is the hash value of the key and the second is the result of the method. It also must ensure that the rest of the pairs in this chain are unchanged and at the same position. For the exception to occur the given key or value must be `null`. The implementation and specification of this method for the WE variant can be found in Appendix A.4.

The `normal_behavior` of the `delete` method can change the entries of the `keys` chain for the given key and the `pair` variable. For this behavior the method must ensure that the key is not in the chain, which can already be true at the beginning. It also must ensure that the rest of the pairs in this chain are unchanged and at the same position. For the exception to occur the given key must be `null`. The implementation and specification of this method for the WE variant can be found in Appendix A.5.

3.3 Linear Probing

The specification for the WI variant has 8 class invariants, 14 `requires` clauses, 16 `ensures` clauses, 4 `loop_invariants`, 4 `assignable` clauses (not counting any with (`\strictly_`) `\nothing`), 29 quantifiers and 4 nested quantifiers. The other variants only have minor differences. All keys as parameters for methods cannot be the dummy key for their `normal_behavior`.

3.3.1 Class Invariants

Each key can at most be once in the hash table except the dummy key and `null`, as seen in Listing 3.3.

```
1 (\forall int y; 0 <= y && y < buckets && keys[y] != iNull && keys[y] != keyDummy;
2     (\forall int z; y < z && z < buckets && keys[z] != iNull && keys[z] != keyDummy;
```

```
3         keys[z] != keys[y]));
```

Listing 3.3: The "At most once" invariant for LP

If a key is not `null`, then the value is also not `null`. This is important for the `get` method, since it returns a `null` if the key is not in the table.

In the `keys` array between the index of a key and it's hash value is no `null`, as seen in Listing 3.4. This is an important property for Linear Probing, but also difficult to verify. This is split into two invariants to avoid using the modulo operator like we did with `getIndex` and `findEmpty`. The reason for this will be explained in the next subsection.

```
1 (\forall int x; 0 <= x && x < buckets && x >= hash(keys[x])
2     && keys[x] != iNull && keys[x] != keyDummy;
3         (\forall int y; hash(keys[x]) <= y && y <= x; keys[y] != iNull));
4
5 (\forall int x; 0 <= x && x < buckets && x < hash(keys[x])
6     && keys[x] != iNull && keys[x] != keyDummy;
7         (\forall int y; hash(keys[x]) <= y && y < buckets; keys[y] != iNull)
8         && (\forall int y; 0 <= y && y < x; keys[y] != iNull));
```

Listing 3.4: The "no null in between" invariant

At least one bucket contains a `null`. Normally when the hash table is almost full, the `resize` method is called to guarantee this.

3.3.2 Private Methods

The method `getIndex` is `strictly_pure`.What this method must ensures depends on the result. If the result is `-1` the given key is not in the `keys` array and if it is not `-1` the result is a valid index for the `keys` array and is the position of the key. The two `loop_invariant`guarantees that every position in the array until now did not include the given key. This is always true, since the method terminates if the key or a `null` bucket is found.

The method `findEmpty` is `strictly_pure`.This method must ensure that the result is a valid index for the `keys` array and it is the position of a `null`. It must also ensure that the returned index is the first index of the `keys` array containing a `null`, when starting the search from the hash value of the given key. This can be seen in Listing 3.5. This is necessary for the verification of the class invariants when proofing the `overwritePair` method.

```
1 (\result >= hash(key)) ==>
2     (\forall int y; hash(key) <= y && y < \result; keys[y] != iNull);
3
4 (\result < hash(key)) ==>
5     ((\forall int y; hash(key) <= y && y < buckets; keys[y] != iNull)
6     && (\forall int y; 0 <= y && y < \result; keys[y] != iNull));
```

Listing 3.5: The "first null" specification

As mentioned in the previous chapter the methods `getIndex` and `findEmpty` use two loops instead of one with the modulo operator. When using the modulo version we had two options. The first is to use a modulo in the specification. While this works, every

method that uses those two methods would need to change their specification two work with the modulo as well. This is not ideal as it would make the specification of every method more complex. So instead we choose a specification without the modulo operator. But this made the verification of `getIndex` and `findEmpty` harder in KeY, but splitting the loop in two fixed this.

The method `overwritePair` requires that the `keys` array does not contain the given key. For this the `getIndex` method is used. It also requires that at least two different buckets contain a `null`. So at least one bucket is still `null` after the methods execution and class invariants can be verified. This method can change the elements in the arrays `keys` and `vals` and change the `pairs` variable. The method must ensure that there is a key in the `keys` array that is equal to the given key. Also the given value must be at the same position in the `vals` array. One thing to note here is, that this clause does not use the methods result to check if the key-value pair is in the hash table. It instead uses the `\exists` quantifier, because it made verification easier. It also must ensure that the rest of the pairs in the hash table are unchanged and at the same position. The implementation and specification of this method for the WE variant can be found in Appendix A.6.

3.3.3 Public Methods

The method `get` is a `pure` method. First we talk about the `normal_behavior`. The `assignable` clause has `\strictly_nothing` as trait. What this method must ensures for this behavior depends on the result. If the result is `null`, the key is not in the keys array and if it is not `null`, the key is in `keys` and the result is at the same position in `vals`. The reason why the method is only `pure` is, that an exceptions creates an object. For the exception to occur the given key must be `null` or the dummy key.

The `normal_behavior` of the `put` method has the same specification as `overwritePair`, except it does not require that the `keys` array does not contain the given key. For the exception to occur the given key must be `null` or the dummy key or the given value must be `null`. The implementation and specification of this method for the WE variant can be found in Appendix A.7.

The `normal_behavior` of the `delete` method can change the elements in the array `keys` and the `pairs` variable. For this behavior the method must ensure that the key is not in the `keys` array, which can already be true at the beginning. It also must ensure that the rest of the pairs in the hash table are unchanged and at the same position. For the exception to occur the given key must be `null` or the dummy key.

4 Verification

In this chapter we look at the verification of the two hash concepts in KeY. We also explain the statistics we gained form the verification. All proofs that have statistics can be completed without interactive steps when using “Java verif. std.” as proof search strategy in KeY. The only Taclet option we changed was `methodExpansion` to `noRestriction`.

For the variants WE the `normal_behavior` of `increaseArraySize`, `overwritePair` and `put` and `delete` for both concepts could not be finished. This is also the case for the constructor for SPA-WE and the methods `increaseArraySize` and `overwritePair` for the NE variants. So they do not have any statistics. We want to briefly describe the reason why those proofs are unfinished. When trying to verify them they ran to about 500.000 rules applications and still were not finished. At this point some methods caused an `OutOfMemoryError` Exception for KeY, but even if not the KeY interface became unstable which made interactive steps problematic. At this point we decided to change strategy and introduced the variants. This was done to find problems in our specification and decrease the size of the proofs. We did neglect interactive proof steps for this and mainly used the visual representation of the proof tree in KeY to find problems.

4.1 Statistics

The Rules applications (RA) is the amount of rules applications used in the proof. It serves as a measure for the length of the proof. The Tables 4.1 and 4.2 show all of them and a "-" marks a unfinished proof. We’re focusing primarily on this statistic, but also include additional ones in the Appendix A.4. There are a total of nine tables. One for each variant and one that compares SPA-WI to LP-WI, by dividing the statistics of first through the second. For each implementation where every contract could be proven the tables also include a row called total. This shows the total amount for each statistic for all contracts combined, except for ATS here it is the average of all. The table that compares SPA-WI with LP-WI, also compares the methods `increaseArraySize` and `overwritePair` with each other. Both method do different things, but are used in the same way in their respective `put` method. Both are called when the given key is not in the hash table.

We have the following extra statistics. The Nodes are the amount of nodes in the proof tree. One node is usually a single rule applications, but multiple simplification steps in a row are combined into one node. Branches are the amount of leafs or goals for a proof tree. Symbolic execution steps (SES) is the amount of rules applications that are symbolic executions. Automode time (AMT) is the total amount of time KeY needed to calculate the proof and is in milliseconds. The average time per step (ATS) is AMT divided by Nodes. Those values can fluctuate, e.g. because of background task, so they should be considered with caution. We also want to mention here that a freshly opened KeY instance can be

slower then usually for a while. We did test this by opening KeY, loading the Java file, finish the proof, then reloading the file, finish the proof again and repeated this a few times. It seems the automode time did stabilize after a few attempts and we tried to give KeY some warm-up time before taking the statistics. But this is an additional reason why the time statistics should be considered with caution.

Table 4.1: Rules applications for Separate Chaining Array

Contract	WithInt	WithIntKeys	NoEquals	WithEquals
constructor	24096	17939	21623	-
delete - normal	32060	30730	46293	-
delete - exceptional	2560	2442	19	19
get - normal	15353	15824	19398	15650
get - exceptional	2814	2720	12	12
getIndex	3460	3497	3227	23011
hash - normal	1303	1303	1432	3648
hash - accessible	1115	1115	1237	1237
increaseAndCopy(Keys)	2398	2398	2490	2490
increaseAndCopy(Vals)	2398	2490	2490	2490
increaseArraySize	58964	60101	-	-
put - normal	82624	252500	224650	-
put - exceptional	5124	2464	28	28
total	231871	395523	-	-

Table 4.2: Rules applications for Linear Probing

Contract	WithInt	WithIntKeys	NoEquals	WithEquals
constructor	3577	3652	3793	3715
delete - normal	15290	13967	25701	-
delete - exceptional	16590	18026	16811	80939
findEmpty	9388	9409	18556	24007
get - normal	1160	1189	1561	3853
get - exceptional	19366	17248	18302	80621
getIndex	44216	44231	71120	252070
hash - normal	1061	1061	1456	7461
hash - accessible	921	931	1238	2434
overwritePair	385191	410942	-	-
put - normal	29632	26818	48215	-
put - exceptional	25488	18327	18966	83832
total	551880	565801	-	-

5 Evaluation

In this chapter we highlight the problems with the verification process for the two hash table concepts. We start by making a comparison between the two WI variants to display the strengths and weaknesses of both. We then look at both hash table concepts individually starting with the WI variant and work towards WE. Some methods have a second contract, either a `exceptional_behavior`or a `accessible` clause. If not mentioned otherwise, we always talk about the `normal_behavior`.

For the WE variant for both hash table concepts, the proof for the `put` and `delete` method could not be finished. The NE counterparts could be finished, so the reason for this problem is the equals method from the HashObject class. Since NE works, it is possible that the proof are just very long. But it is possible that interactive proofs steps are necessary to complete the proofs.

Separate Chaining Array vs Linear Probing. The constructor in SPA-WI has a much longer proof then the one in LP-WI. This is because the constructor for SPA has a loop and the `loop_invariant` needs to guarantee all the invariants for the chains.

The proof for `get` is longer in SPA-WI, but for `getIndex` it is shorter. One reason is that both concepts call the `hash` method in different places, but there is more to it. A big difference between the two `getIndex` methods is, that LP-WI stops it's search when it has found a `null`. Proofing that the key is not in the hash table when a `null` is found is difficult. The proof for the `put` and `delete` method is also longer in SPA-WI. So the location of the hash function seems to be important, but the two dimensional structure of SPA is also part of the reason.

The proof for the `hash` method is longer in SPA-WI and the only difference is that LP-WI does not allow the dummy key as parameter. The proof for the `exceptional_behavior` for `get`, `put` and `delete` is shorter in SPA-WI and the only difference is that LP-WI also throws an exception when the given key is the dummy key. The total amount of rules applications for SPA-WI is not even half the amount LP-WI has. This is because the `overwritePair` method has by far the longest proof of both WI variants and is more than 6 times longer then `increaseArraySize`.

Separate Chaining Array. For WI the proof for `put` and `increaseArraySize` together have more then half of the total rules applications. Both methods can change the heap, so it must be proven that the invariants still hold. The invariant "Each key is at most ones in the same chain." is a big contributor.

When switching from WI to WIK, so changing the value type from `int` to `Object`, the proof for the `put` method is 3 times as long. Surprisingly the length of the proof for `increaseArraySize` did not change much. This method is only called when the key is not in the chain of the key's hash value. So one might think the increase comes when

overwriting the value of an equal key. Those two things are divided by a branch in KeY, but the majority of the rules still were applied in the `increaseArraySize` branch. Surprisingly, when switching from WIK to NE, so changing the key type from `int` to `HashObject`, the length of the `put` proof did not increase further, in fact it even did decrease slightly. But `increaseArraySize` could not be finished anymore.

The length of the proof for the `exceptional_behavior` of the `get`, `put` and `delete` method becomes very small in NE and WE and also for put in WIK. The exception is thrown when a parameter is `null`. In the cases above the `null` keyword is used and in every other instance `iNull`. The proof for the constructor is shorter in WIK and NE then it is in WI. When switching from WIK to NE the proofs for `delete` and `get` did become longer. This would not be too surprising, but the one for `put` did become shorter. The proof for the constructor and `increaseArraySize` of WE have similar problems as the proofs for `put` and `delete`. When switching from NE to WE the length of the proof for `hash` and `getIndex` did increase by a lot. They both compare keys with each other, so the added complexity from `equals` is the cause. But the proof for the `get` method is shorter in WE and it uses both methods and `equals` in it's contract.

Linear Probing. For WI the proof for the `overwritePair` method has almost 70% of the total rules applications. This method inserts a new key-value pair in the hash table and therefore changes the heap. As we have already seen, this increases the length of the proof by a lot. But here we have an additional challenge. The invariant "between the index of a key and it's hash value is no null" is a big contributor. The proof for this method could not be finished with the NE and WE variant, so the change from `int` to HashObject causes problems.

The switch from WI to WIK seems to make almost no difference to the length of the proofs. From WIK to NE and NE to WE the length of most proofs did increase. So the change from `int` to `HashObject` and from the equality operator to the `equals` method does make the LP more difficult.

6 Conclusion and Future Work

The most difficult part of the verification of the hash table concepts is when the heap got changed. This is even true for the simplest variant WI. The `overwritePair` method form LP is the prime example, since it has by far the longest proof. Because LP has extra invariants for it's use of `null`, the invariants are a big part of this problem. On top of this we have the problem with the `equals` method. Even though it is not much more complex then the equality operator, it still made the verification a lot more difficult.

So what can be done from here on out? The `delete` method in there original state and the `resize` method for both concepts would be a start. One way to do the delete method for LP would be to use an alternative `resize` method that ignores the deleted key-value pair. The variant WE is also still open and a higher focus on interactive steps could be the solution.

Finally there are more options for Open Addressing and Separate Chaining, as mentioned in the introduction. We started this thesis with SPA and later did LP. The Specification of LP was a lot faster, then the one for SPA. This was mainly because many of their specifications are similar and only need minor changes. The main work was the methods that are not in SPA and the invariants for the special use of `null`. So at least the specification of new hash table concepts should not be to difficult.

Bibliography

Ahrendt, Wolfgang et al., eds. (2016). *Deductive Software Verification - The KeY Book: From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer. DOI: 10.1007/978-3-319-49812-6.

Clausen, Esben Glavind, ed. (2017). *Verifying Hash tables in Iris*. URL: https://iris-project.org/pdfs/2017-thesis-verifying-hash-tables-in-iris.pdf.

Jung, Christian, ed. (2021). *The GitHub Repository for this Thesis*. URL: https://github.com/ChristianJ225/HashTableWithKeY.

Leavens, Gary T., ed. (2017). *The Java Modeling Language (JML)*. URL: https://www.cs.ucf.edu/~leavens/JML/index.shtml.

Pottier, François, ed. (2016). *Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic*. URL: http://gallium.inria.fr/~fpottier/publis/fpottier-hashtable.pdf.

Sedgewick, Robert and Kevin Wayne, eds. (2016). *Algorithms, 4th Edition: Searching*. URL: https://algs4.cs.princeton.edu/30searching/.

A Appendix

A.1 Class Invariants

```
1  //The hash table has at least one chain.
2  instance invariant  chains > 0;
3
4  //The arrays keys and vals are nonnull and are two different arrays.
5  instance invariant  keys != null && vals != null && keys != vals;
6
7  //The arrays keys and vals are not suptypes.
8  instance invariant  \typeof(keys) == \type(HashObject[][])
9                      && \typeof(vals) == \type(Object[][])
10                     && (\forall int x; 0 <= x && x < chains;
11                         \typeof(vals[x]) == \type(Object[])
12                         && \typeof(keys[x]) == \type(HashObject[]));
13
14 //The arrays keys and vals are equally long.
15 instance invariant  chains == vals.length && chains == keys.length;
16
17 //Every element in keys or vals is nonnull.
18 instance invariant  (\forall int x; 0 <= x && x < chains;
19                         keys[x] != null && vals[x] != null);
20
21 //Each array inside of keys has an equally long partner array in vals at the same postion.
22 instance invariant  (\forall int x; 0 <= x && x < chains;
23                         keys[x].length == vals[x].length);
24
25 //No two different postion in keys or vals have a reference to the same subarray.
26 instance invariant  (\forall int x; 0 <= x && x < chains;
27                         (\forall int y; x < y && y < chains;
28                             keys[x] != keys[y] && vals[x] != vals[y]));
29
30 //No array in keys and vals have the same reference.
31 instance invariant  (\forall int x; 0 <= x && x < chains;
32                         (\forall int y; 0 <= y && y < chains;
33                             keys[x] != vals[y]));
34
35 //Each key is at most ones in the same chain.
36 instance invariant  (\forall int x; 0 <= x && x < chains;
37                         (\forall int y; 0 <= y && y < keys[x].length && keys[x][y] != null;
38                           (\forall int z; y < z && z < keys[x].length && keys[x][z] != null;
39                                 !keys[x][z].equals(keys[x][y]))));
```

Listing A.1: The Class Invariants of Separate Chaining Array With Equals

```
// The hash table has at least one bucket.
instance invariant  buckets > 0;

//The arrays keys and vals are nonnull and are two different arrays.
instance invariant  keys != null && vals != null && keys != vals;

// The arrays keys and vals are equally long.
instance invariant  buckets == keys.length && buckets == vals.length;

//The array vals is not a suptype.
instance invariant  \typeof(keys) == \type(HashObject[])
                    && \typeof(vals) == \type(Object[]);

//Each key is at most ones in the hash table.
instance invariant  (\forall int y; 0 <= y && y < buckets
                    && keys[y] != null && !keyDummy.equals(keys[y]);
                        (\forall int z; y < z && z < buckets
                        && keys[z] != null && !keyDummy.equals(keys[z]);
                            !keys[z].equals(keys[y])));

//If a key is not null, then the value is also not null.
//  This is important for get(), since it returns null if the key is not in the table.
instance invariant  (\forall int x; 0 <= x && x < buckets;
                            (keys[x] != null) ==> (vals[x] != null));

//The following two invariants guarantee that between a key and it's hash value
//is no null value. This is important for LinearProbing so it can stop searching
//for a Keys if it finds a null.
instance invariant  (\forall int x; 0 <= x && x < buckets && x >= hash(keys[x])
                    && keys[x] != null && !keyDummy.equals(keys[x]);
                        (\forall int y; hash(keys[x]) <= y && y <= x;
                            keys[y] != null));
instance invariant  (\forall int x; 0 <= x && x < buckets && x < hash(keys[x])
                    && keys[x] != null && !keyDummy.equals(keys[x]);
                        (\forall int y; hash(keys[x]) <= y && y < buckets; keys[y] != null)
                        && (\forall int y; 0 <= y && y < x; keys[y] != null));

//At least one bucket contains a null. When the hash table is almost full,
//the resize() method is called to guarantee this.
//Note: the resize() method is currently not used.
instance invariant  (\exists int x; 0 <= x && x < buckets; keys[x] == null);
```

Listing A.2: The Class Invariants of Linear Probing With Equals

A.2 Separate Chaining Array With Equals

```
/*@ public normal_behavior
  @   requires  chains > 0;
  @   ensures   (\forall HashObject ho; key.equals(ho) ==> (\result == hash(ho)))
  @             && 0 <= \result && \result < chains;
  @   ensures_free  \result == hash(key);
  @   assignable    \strictly_nothing;
```

```
  @   accessible    key.value, this.chains;
  @
  @ helper
  @*/
private /*@ strictly_pure @*/ int hash(HashObject key) {
    return abs(key.hashCode()) % chains;
}
```

Listing A.3: The hash Method of SPA-WE

```
/*@ public normal_behavior
  @   //The key-value pair is now in the hash table and at the same position.
  @   ensures   key.equals(keys[hash(key)][\result]) && vals[hash(key)][\result] == val;
  @
  @   //The method has no effect on hash table positions were the key isn't placed.
  @   ensures   (\forall int x; 0 <= x && x < \old(keys[hash(key)].length) && x != \result;
  @                keys[hash(key)][x] == \old(keys[hash(key)][x])
  @                && vals[hash(key)][x] == \old(vals[hash(key)][x]));
  @
  @   assignable    keys[hash(key)], vals[hash(key)], vals[hash(key)][*], pairs;
  @
  @ also
  @ exceptional_behavior
  @   requires  key == null || val == null;
  @   signals_only  IllegalArgumentException;
  @   signals   (IllegalArgumentException e) true;
  @*/
public int put(HashObject key, Object val) {
    if (key == null)
        throw new IllegalArgumentException("The first argument to get() is null");
    if (val == null)
        throw new IllegalArgumentException("The second argument to put() is null");
    int iHash = hash(key);
    int index = getIndex(iHash, key);

    if (index != -1) {
        vals[iHash][index] = val;
        return index;
    }
    increaseArraySize(iHash, key, val);
    return keys[iHash].length - 1;
}
```

Listing A.4: The put Method of SPA-WE

```
/*@ public normal_behavior
  @
  @   //The given key is not in the table. (This can already be true at the beginning)
  @   ensures   (\forall int x; 0 <= x && x < keys[hash(key)].length;
  @                !(key.equals(keys[hash(key)][x])));
  @
  @   //The method has no effect on hash table positions were the key wasn't placed.
  @   ensures   (\forall int x; 0 <= x && x < keys[hash(key)].length && x != \result;
  @                    keys[hash(key)][x] == \old(keys[hash(key)][x]));
```

```
  @
  @   assignable keys[hash(key)][*], pairs;
  @
  @ also
  @ exceptional_behavior
  @   requires key == null;
  @   signals_only IllegalArgumentException;
  @   signals (IllegalArgumentException e) true;
  @*/
public int delete(HashObject key) {
    if (key == null)
        throw new IllegalArgumentException("The argument to delete() is null");
    int iHash = hash(key);
    int index = getIndex(iHash, key);
    if (index != -1) {
        keys[iHash][index] = null;
        pairs--;
    }
    return index;
}
```

Listing A.5: The delete Method of SPA-WE

A.3 Linear Probing With Equals

```
/*@ public normal_behavior
  @   requires  !keyDummy.equals(key);
  @
  @   //The key is not in the hash table.
  @   requires  getIndex(key) == -1;
  @
  @   //There are at least two nulls in the hash table.
  @   //So the invarants hold after a null is replaced with a key.
  @   requires  (\exists int x; 0 <= x && x < buckets;
  @                (\exists int y; x < y && y < buckets;
  @                    keys[y] == null && keys[x] == null));
  @
  @   //The key-value pair is now in the hash table and at the same position.
  @   ensures   (\exists int x; 0 <= x && x < buckets;
  @                key.equals(keys[x]) && vals[x] == val);
  @
  @   //The method has no effect on hash table positions were the key isn't placed.
  @   ensures   (\forall int x; 0 <= x && x < buckets && x != \result;
  @                keys[x] == \old(keys[x]) && vals[x] == \old(vals[x]));
  @
  @   assignable    keys[*], vals[*], pairs;
  @*/
private int overwritePair(HashObject key, Object val) {
    int index = findEmpty(key);
    keys[index] = key;
    vals[index] = val;
    pairs++;
```

```
28      return index;
29  }
```

Listing A.6: The overwritePair Method of LP-WE

```
1  /*@ public normal_behavior
2    @   //The same as overwritePair, except
3    @   //"requires   getIndex(key) == -1;"
4    @   //doesn't appear hear.
5    @ also
6    @ exceptional_behavior
7    @   requires  key == null || keyDummy.equals(key) || val == null;
8    @   signals_only  IllegalArgumentException;
9    @   signals   (IllegalArgumentException e) true;
10   @*/
11 public int put(HashObject key, Object val) {
12      if (key == null || keyDummy.equals(key))
13          throw new IllegalArgumentException("The first argument to put() is invalid");
14      if (val == null)
15          throw new IllegalArgumentException("The second argument to put() is invalid");
16      int index = getIndex(key);
17      if (index != -1) {
18          vals[index] = val;
19          return index;
20      }
21      return overwritePair(key, val);
22  }
```

Listing A.7: The put method of LP-WE

A.4 Statistics Tables

Table A.1: Complete Statistics Table for SPA-WI vs LP-WI

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	7.59	6.11	1.87	16.86	2.23	6.74
delete - normal	2.27	1.83	0.65	2.13	0.94	2.1
delete - exceptional	0.17	0.09	0.06	0.15	0.9	0.15
get - normal	15.13	13.78	26.45	18.15	1.2	13.24
get - exceptional	0.16	0.1	0.05	0.16	0.96	0.15
getIndex	0.09	0.03	0.02	0.05	0.55	0.08
hash - normal	1.17	1.27	1.35	1.18	1.01	1.23
hash - accessible	1.15	1.27	1.31	1.37	1.19	1.21
put - normal	2.89	2.47	1.35	5.05	1.75	2.79
put - exceptional	0.22	0.11	0.08	0.19	0.84	0.2
iAS vs oP	0.15	0.08	0.03	0.14	0.95	0.15
total	0.44	0.29	0.13	0.6	1.12	0.42

Table A.2: Complete Statistics Table for SeparateChainingArrayWithInt

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	18650	165	293	114758	6.183	24096
delete - normal	26728	231	1064	70109	2.623	32060
delete - exceptional	2108	13	113	7265	3.448	2560
get - normal	11939	124	1005	22247	1.863	15353
get - exceptional	2322	19	117	8176	3.522	2814
getIndex	2718	11	75	3074	1.131	3460
hash - normal	839	14	170	659	0.786	1303
hash - accessible	686	14	152	478	0.697	1115
increaseAndCopy	1503	22	103	1625	1.081	2398
increaseArraySize	43072	285	1023	107033	2.485	58964
put - normal	64252	474	2788	316785	4.93	82624
put - exceptional	4220	25	224	15147	3.59	5124
total	179037	1397	7127	667356	2.69	231871

Table A.3: Complete Statistics Table for SeparateChainingArrayWithIntKeys

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	12527	98	290	65846	5.256	17939
delete - normal	25551	234	1064	69163	2.706	30730
delete - exceptional	1998	13	113	6869	3.439	2442
get - normal	12392	124	1005	25750	2.078	15824
get - exceptional	2235	19	117	8901	3.984	2720
getIndex	2749	11	75	3234	1.176	3497
hash - normal	839	14	170	679	0.81	1303
hash - accessible	686	14	152	492	0.718	1115
increaseAndCopyKeys	1503	22	103	1755	1.168	2398
increaseAndCopyVals	1537	24	102	1570	1.022	2490
increaseArraySize	44309	323	1044	125913	2.814	60101
put - normal	195547	1800	2999	855082	4.372	252500
put - exceptional	2019	13	113	7410	3.671	2464
total	303892	2709	7347	1172664	2.55	395523

Table A.4: Complete Statistics Table for SeparateChainingArrayNoEquals

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	16149	129	293	86083	5.33	21623
delete - normal	38997	428	1753	109410	2.805	46293
delete - exceptional	11	1	0	13	1.3	19
get - normal	14348	189	1786	27508	1.917	19398
get - exceptional	11	1	0	11	1.1	12
getIndex	2518	11	77	3009	1.195	3227
hash - normal	873	18	236	693	0.794	1432
hash - accessible	754	16	188	506	0.671	1237
increaseAndCopyKeys	1537	24	102	1484	0.966	2490
increaseAndCopyVals	1537	24	102	1687	1.098	2490
increaseArraySize	-	-	-	-	-	-
put - normal	184128	1693	5216	743981	4.04	224650
put - exceptional	19	1	0	44	2.444	28

Table A.5: Complete Statistics Table for SeparateChainingArrayWithEquals

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	-	-	-	-	-	-
delete - normal	-	-	-	-	-	-
delete - exceptional	11	1	0	12	1.2	19
get - normal	10451	166	1956	21332	2.041	15650
get - exceptional	11	1	0	19	1.9	12
getIndex	14185	366	3510	36560	2.577	23011
hash - normal	2279	42	579	1827	0.802	3648
hash - accessible	754	16	188	553	0.734	1237
increaseAndCopyKeys	1537	24	102	1522	0.99	2490
increaseAndCopyVals	1537	24	102	1531	0.996	2490
increaseArraySize	-	-	-	-	-	-
put - normal	-	-	-	-	-	-
put - exceptional	19	1	0	13	0.722	28

Table A.6: Complete Statistics Table for LinearProbingWithInt

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	2457	27	157	6807	2.771	3577
delete - normal	11796	126	1632	32954	2.793	15290
delete - exceptional	12465	141	1913	47762	3.831	16590
findEmpty	6161	83	889	6792	1.102	9388
get - normal	789	9	38	1226	1.555	1160
get - exceptional	14186	187	2269	52187	3.679	19366
getIndex	31298	407	4433	64562	2.062	44216
hash - normal	715	11	126	558	0.781	1061
hash - accessible	597	11	116	350	0.587	921
overwritePair	284469	3438	37917	746723	2.624	385191
put - normal	22267	192	2069	62688	2.815	29632
put - exceptional	19061	220	2956	81189	4.259	25488
total	406261	4852	54515	1103798	2.40	551880

Table A.7: Complete Statistics Table for LinearProbingWithIntKeys

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	2498	27	157	7456	2.985	3652
delete - normal	10671	111	1529	30268	2.836	13967
delete - exceptional	13090	159	2207	50527	3.86	18026
findEmpty	6184	83	889	6587	1.065	9409
get - normal	823	9	38	1250	1.52	1189
get - exceptional	12722	151	2037	48101	3.781	17248
getIndex	31369	402	4450	66931	2.133	44231
hash - normal	715	11	126	555	0.777	1061
hash - accessible	608	11	116	348	0.573	931
overwritePair	300545	3759	40886	970844	3.23	410942
put - normal	20221	150	1772	53965	2.668	26818
put - exceptional	13301	162	2242	59834	4.498	18327
total	412747	5035	56449	1296666	2.49	565801

Table A.8: Complete Statistics Table for LinearProbingNoEquals

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	2601	28	157	12914	4.966	3793
delete - normal	17518	324	4412	55649	3.176	25701
delete - exceptional	11190	206	2701	41045	3.668	16811
findEmpty	12897	180	1964	17383	1.347	18556
get - normal	1191	9	46	1597	1.342	1561
get - exceptional	12240	226	2873	42282	3.454	18302
getIndex	48580	794	9051	99851	2.055	71120
hash - normal	897	18	236	851	0.949	1456
hash - accessible	755	16	188	595	0.789	1238
overwritePair	-	-	-	-	-	-
put - normal	33518	496	6013	82948	2.474	48215
put - exceptional	12576	233	3001	41237	3.279	18966

Table A.9: Complete Statistics Table for LinearProbingWithEquals

Contract	Nodes	Branches	SES	AMT	ATS	RA
constructor	2540	28	157	12255	4.826	3715
delete - normal	-	-	-	-	-	-
delete - exceptional	50181	1063	12904	224256	4.469	80939
findEmpty	15815	255	2930	20298	1.283	24007
get - normal	2033	47	572	2878	1.416	3853
get - exceptional	49996	1062	12827	213161	4.263	80621
getIndex	162588	3073	35831	397987	2.447	252070
hash - normal	4546	88	1224	3981	0.875	7461
hash - accessible	1464	32	361	1445	0.987	2434
overwritePair	-	-	-	-	-	-
put - normal	-	-	-	-	-	-
put - exceptional	51619	1088	13120	234958	4.551	83832