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
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 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. First we look at how they can be specified in Java Modelling Language (JML). We then use those specification to see how well they can be verified in KeY and what problems we encountered.

Zusammenfassung

Deutsche Zusammenfassung

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

A hash table is a dictionary data structure, that can insert, delete and search data in constant time.Those effects are more apparent when the memory space is large. But the worst-case time for insert, delete and search is linear.
A hash function is used to calculate the most likely location of an entry in the hash table. It is possible for two different entries to have the same hash value, this is known as a collision. A large number of collisions can lead to the worst-case time above.
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.

1.2 Problem Statement

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 is and how we assess them follows below. When I 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.

1.3 Contribution

We create a Java implementations of two hash table concepts (Chapter 3) and specify them with JML (Chapter 4). 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 (Chapter 5). With those statistics we highlight difficult sections of the verification process (Chapter 6). The conclusion will be Chapter 7.
For each hash table concept we will provide the following:

- A Java implementation
- A JML specification

- A KeY proof (.proof files)
- An assessment, the statistics and a recommendation.

2 Outline

2.1 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 usually the hash function.

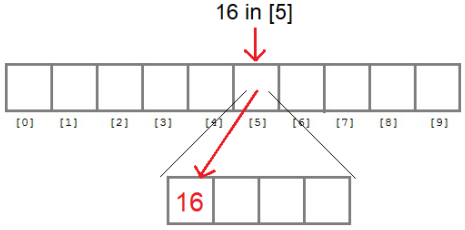
A collision occurs when two different pieces of data have the same hash value. In practices this is usually unavoidable. A collision resolution strategy handles such an event, by finding a new location, either inside the same bucket or in another one. Some collision resolution strategies are explained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

2.1.1 Separate chaining

The buckets don't contain the elements directly, instead they contain a data structure like in Figure 2.1. When two element have a collision, both elements are placed in the same bucket. Since we don't know in advanced where this happens dynamic data structures are recommended, so constant bucket resizing is doesn't cause slowdown. Some possible data structures would be:

- Linked list
- Self-balancing binary search tree
- dynamic array
- A second hash table

Figure 2.1: Separate chaining

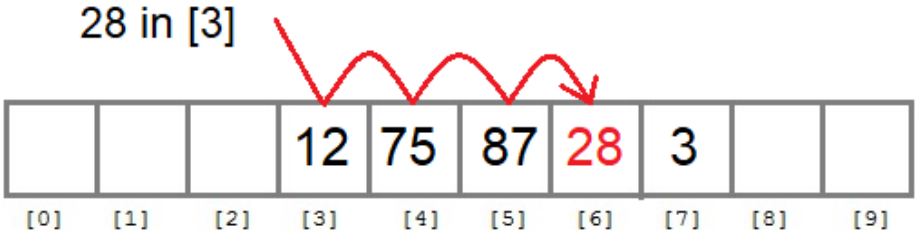


2.1.2 Open addressing

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 2.2: Linear probing with interval 1



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 runtime. 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.

2.2 Implementation

We didn't want creating a completely new implementation to avoid creating and solve our own problem. By using an implementation that is already used we can keep this evaluation more practical. Our implementation originates from a booksite.
Before we even started we did need to change some aspects of this implementation. First KeY doesn't support Generics, so they were replaced by a singel class/type. The hash function was simplified and the methods `size`, `isEmpty` and `contains` got removed, as they weren't necessary. Our implementation of Separate Chaining also doesn't use an extra class for the data structure and instead two arrays.

2.3 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.

The KeY System is a formal verification tool for Java programs. It verifies the program against it 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.

2.4 Related work

In parallel to my thesis there was 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 create a foundation hash table concepts in KeY. Since our works have similarity we had meetings together to help improve each other's work.

There was also a thesis about sequential and concurrent hash tables in Iris, a separation logic framework. It was written by Esben Glavind Clausen and called "Verifying Hash tables in Iris". The thesis builds upon a paper by François Pottier 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.

3 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.

3.1 Common behavior

The elements for our 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 the same based on the method of comparison, which will be explained later.
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 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.

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 didn't include it in our implementation. This still will be 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 its associated value inaccessible. This method does nothing if the hash table doesn't contain the key.

3.2 Separate Chaining Array

A short disclaimer before we start. 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 second dimension of those arrays are the chains. The variable chains is the length of the hash table, so the length of the first dimension of keys and vals.

The method `hash` returns the index for keys and vals 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 a key is in the hash table at all.
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 the hash table.
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`.

3.3 Linear Probing

This hash table concept has two arrays, one for the keys called keys and one for the values called vals. The variable buckets is the length of the hash table, so the length of keys and vals.
It is necessary that keys 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 has 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 method `getIndex` is a key. It returns a index for keys of the given key, if the key is in the hash table. If we find a null the key is not in the hash table and a -1 is returned. This method is also used by other methods to check if a keys is in the table at all.
The `findEmpty` method works similar to the getIndex method, but it only stops when it finds a null and returns it's index for keys. 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. This simple method will be important in later chapters.

3.4 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`

where introduced to split the proofs in smaller, more manageable proofs.
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. But 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 doesn't make any difference for normal use of the class. But it does effect the verification of the constructor.

3.4.1 Keys and Variants

We started by using a new Class for the keys called `HashObject`. This class has a final `int` variable called value and it is set by the constructor. This variable is used to compare `HashObjects` in the `equals` method of the class, seen Listing 3.1. Two keys are considered the same if they are both `HashObjects` and have the same value. The value is also used in the hash function.

Listing 3.1: equals method

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

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 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 intermediate 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 `Objects`.
- NoEqulas (NE): Keys has 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 is that WI is the simplest to verify, then WIK, then NE and finally WE is the hardest.

Our implementation of the hash table concepts use the `null` keyword in some circumstances and the `int` data type doesn't have this. So we use 0 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.

3.4.2 Delete

Even with the WI variant the `delete` method was difficult to verify, so they 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 we instead just 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 a key and its hash value, which would make this key not find able. 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, which isn't a valid key, effectively deleting the key. This key can't be overwrite by a valid key, because this change caused problems with the verification of the `overwritePair` method.

3.4.3 Linear Probing

For the methods `findEmpty` and `getIndex`, when searching for the key it is possible that the end of the array is reach 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 3.2

Listing 3.2: modulo loop

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

Here we instead use two different for-loops as seen in Listing 3.3. 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.

Listing 3.3: divided loops

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

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

4 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.

4.1 Common specifications

All keys and values as parameters for methods must be non-null for their normal_behavior. We're focusing on the WI variants, since the changes between the variants are straightforward. But they will be listed here. Per default an object is considered non-null in JML, so there is only a requires clause if the keys or values are an `int`. If the specification uses the type of a variable, then it is changed according to the variant. The same applies for the use of the `equals` method or the equality operator, when comparing two keys to one another.

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 the it's execution. But with the constructor we can show that they even hold from the very beginning. The contract it self requires that the given value, the length of the hash table, is at least 1. The constructor must ensure that the that keys and vals are new arrays, the amount of pairs is 0 and the length of the hash table is the given length.

4.1.1 Class invariants

The arrays keys and vals are always non-null and the size is bigger then 0. If one of these was true there would be no pair in the hash table and the first call of the `put` method would call the `resize` method.
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, but not when the array is of the type `int`. 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.

4.1.2 hash

The `hash` method is strictly_pure and a helper method. It also has a accessible clause only allowing access to the table length and the value of the key, if the use the NE or WE variant.
The method uses a ensures_free clause that says calling this method same key always has

the same result. But to use this clause we need to proof out side of KeY that this is true. Since a `int` key uses it self and the value variable of a `HashObject` cannot be changed after creating said object, the base value for the hash function is always the same if the given key is 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. If this wouldn't be the case two different keys that are equal could 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. This is because it helped the verification in some instances.

4.2 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. 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. The method must ensure that the second dimension consists of new arrays.

4.2.1 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 only be once in the same chain as seen in Listing 4.1. 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.

Listing 4.1: At most once invariant for SPA

```
(\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])));
```

But using both invariants caused problems when verifying the method `increaseArraySize`. This problem even occurred when the "correct bucket" invariant was just a required clause. I had similar problems when using two other invariants. The first was that no key or value

can be `null` and the other was that 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 4.2 as an example.

Listing 4.2: correct bucket invariant

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

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.

4.2.2 Private methods

The method `getIndex` is strictly_pure. It requires that the given hash value is a valid index for the array keys. What this method must ensures 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 up until now didn't 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 pair variable. It must ensure that the new chains are still the correct type, their length did increase by one, they still contains 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 still 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. loop_invariant guarantees that every position in the new array up until now contains all the 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 can't be `null`.

4.2.3 Public methods

The `get`, `put` and `delete` method have all two contracts, one for normal_behavior and one for exceptions.

The method `get` is a pure method. First we talk about the normal_behavior. The assignable clause is strictly_nothing here. 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 keys chain

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`.

The `put` method can change the chains of the hash value for the given key, the entries of the same vals chain and the pair 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 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 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`.

4.3 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.

4.3.1 Class invariants

Each key can at most be once in the in the hash table except the dummy key and null as seen in Listing 4.3.

Listing 4.3: At most once invariant for LP

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

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 4.4. This is an important property for Linear Probing, but also difficult to verify. This is split into two invariants to avoid using a modulo operator like we did with getIndex and findEmpty. The reason for this will be explained in the next subsection.

Listing 4.4: No null in between invariant

```
(\forall int x; 0 <= x && x < buckets && x >= hash(keys[x])
    && keys[x] != iNull && keys[x] != keyDummy;
```

```
        (\forall int y; hash(keys[x]) <= y && y <= x;
            keys[y] != iNull));

(\forall int x; 0 <= x && x < buckets && x < hash(keys[x])
    && keys[x] != iNull && keys[x] != keyDummy;
        (\forall int y; hash(keys[x]) <= y && y < buckets;
            keys[y] != iNull)
        && (\forall int y; 0 <= y && y < x;
                keys[y] != iNull));
```

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

4.3.2 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 up until now didn't 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 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 4.5. This is necessary for the verification of the class invariants when proofing the `overwritePair` method.

Listing 4.5: First null

```
(\result >= hash(key)) ==>
    (\forall int y; hash(key) <= y && y < \result;
        keys[y] != iNull);

(\result < hash(key)) ==>
    ((\forall int y; hash(key) <= y && y < buckets;
        keys[y] != iNull)
    && (\forall int y; 0 <= y && y < \result;
        keys[y] != iNull));
```

As mentioned in the previous chapter the methods `getIndex` and `findEmpty` use two loops instead of one with a modulo operator. When using the modulo version we had two options. The first was to use a modulo in the specification. While this worked, every method that used those two methods would need to change their specification two work with the modulo as well. This wasn't ideal as it would make the specification of every method more complex. So we 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 doesn't 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 method 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 doesn't use the methods result to check if the key-value pair is in the hash table. It instead uses an 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.

4.3.3 Public methods

The `get`, `put` and `delete` method have all two contracts, one for normal_behavior and one for exceptions.

The method `get` is a pure method. First we talk about the normal_behavior. The assignable clause is strictly_nothing here. 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 doesn't require that the keys array doesn't 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 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.

5 Verification

In this chapter we look at the verification of the two hash concepts ub KeY. We will also explain the statistics we gained form the verification.

5.1 KeY

All proofs that have statistics can be completed without interactive steps when using "Java verif. std." as proof search strategy. The only Taclet option we did change was methodExpansion to noRestriction.
For the variants WE the normal_behavior of `increaseArraySize`, `overwritePair` and `put` and `delete` for both concepts couldn't 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 don't 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 weren't 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.

5.2 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. We're focusing primarily on this statistic, but included additional ones.
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 avg. 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 did 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.
We did not see any similar fluctuations on the other types of statistics, in fact they all were always exactly the same even after repeated retries.

We have nine tables in total. One for each variant and one that compares SPA-WI to LP-WI, by dividing the statistics of first through the second. The gabs between the rows is so all contracts of one method are grouped together.
For each implementation where every contract could be proven the tables also include a row called total. This shows 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 isn't in the hash table.

6 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 strength 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.

6.1 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 `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 might be part of the reason.

The proof for the `hash` method is longer in SPA-WI and the only difference is that LP-WI doesn't 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`.

6.2 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` couldn't 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 our `int` null value.
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 wouldn't 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 most likely the cause. But the proof for the `get` method uses both methods and uses equals in it's contract and is shorter then before.

6.3 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 couldn't 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.

7 Conclusion and future Works

The most difficult part of the verification of 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 isn't much more complex then the equality operator, it still made the verification a lot more difficult.

So what to do from here on out? The `delete` method in there original state and the `resize` method would be a start. One way to do the delete method for LP would be to use a alternative `resize` method that ignores the deleted key-value pair.
The variant WE is also still open, but a better machine then mine or interactive steps might be enough.
Finally there are more options for Open Addressing and Separate Chaining, as mentioned in the Outline chapter. 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 were the methods that aren't in SPA and the invariants for the special use of `null`. So at least Specification of new hash table concepts shouldn't be to difficult.

A Appendix

A.1 First Appendix Section

Figure A.1: A figure

...