Specification and Verification of Hash Table Data Structures with KeY

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.

Question

My central question is:

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 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. We also want to compare them with each other to find their strengths and weaknesses in the verification process. Further more we want to find an interface for hash tables, so when they appear as part of a Java program, the concept can change without any effect for KeY or the proof. Beyond that we want to see if there are any problems with KeY like bugs, unintuitive design or difficult segments of the verification process and find potential improvements like new features, expanded settings and better UI.

JML

The Java Modeling Language (JML) is a specification language for Java programs. It allows to define pre- and postconditions (hoard logic) for methods and is based on the idea of design by contract. It was created in 1999 and is continuously developed by the community.

KeY System
The KeY System is a formal verification tool for Java programs.
Proofs can be performed both automatically and interactively.
It uses JML specifications.
Java dynamic logic (JavaDL)

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 structure. 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.

Collision Resolution Strategies

Separate chaining
Each bucket of the hash table is one of the following data structures:
- Array
- Linked list
- List head cells: The first element is an entry.
- Self-balancing binary search tree
- dynamic array
- A second hash table

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. To do so a probing sequence is used.
- Linear probing: The next bucket is a fixed distance away, usually 1.
- 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.

Cuckoo hashing
This strategy uses two or more hash tables. If a new entry is added to one of the hash tables and the bucket is already filled, the new entry replaces the old one. The replaced entry is then placed in the next hash table. If the last hash table is reached then the first hash table is next. This process is repeated until an empty bucket is found.

The following strategies are variations on previous strategies and are less important for now.

Coalesced hashing
This strategy uses open addressing to place new entries in the hash table, but each entry with the same hash value is linked together. So if two entries with the same hash value are put into the hash table, then the first entry gets a pointer to the index of the second entry.

Hopscotch hashing
Each hash value has a neighborhood. Entries with the same hash value are always in the same neighborhood and if each entry in the neighborhood has already the same hash value the neighborhood’s size is increased. If an entry in the neighborhood does not have the same hash value, because it is am member of another one, then an empty space for the new entry is found and then swap with the first entry. If this first entry is then outside its neighborhood, then the process repeats.

Robin Hood hashing
This strategy uses open addressing, but if the entry in the current probing bucket is closer to its hash value bucket compared to the entry we want to place in the hash table, then we switch those two entries. We then use probing to find a bucket for the first entry with the same method. This is repeated until we find an empty bucket.

2-choice hashing
This strategy uses two hash tables and a new element gets placed in the hash table with few elements.

Comparison Part 1

Keys

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 and for the hash function, which we discuss later in this chapter. To compare to Keys we used the equals() method.
Here we run in or first problem. The verification of the put() and delete() methods, including some sub methods, run very long in KeY. Usually around 500.000 nodes and still not finishing and in some instances cause an OutOfMemoryError exception. With proofs of this size it was also difficult to see if the Specification was wrong or if it took just that long. Which is also the reason why I kept the interactive steps to a minimum. It is of course possible that interactive steps could decrease the length of the proof, but I was running out of time.
So we changed strategy, instead of using HashObject as Keys (and Objects as Values) we used int. With this change we were able to finish the verification and further decrease the size of the proofs. Even with these changes the original strategy did not work.
We also did try two other versions. The first uses int Keys, but Object Values. And the seconds used HashObject for Keys and Object for Values, but instead of the equals() method we used == to compare two Keys. We go into more details on how those different versions compare to each others and how the two different hash table concepts compare two each other.

Hash function

We first get a int associate with the Key. With the HashObject class we the value variable and with the int Keys we just use the key itself.
Then we check if this int is a negative number, and if so we make it positive. For this we created a method called 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.
Lastly we apply module with the size of the hash table to this value.
For Separate Chaining the result is the index of the chain (of elements) that can, but not necessarily does, contain the Key.
For Linear Probing the result is the start index from where we search for the Key. If we find a null the Key is not in the table.
The following is important for a hash function in the hash table. If the two elements are equal, then the hash value of the two elements is also equal. For the original strategy equal means equals() method and the rest it is ==.
Key1 == Key2   ==>   H(Key1) == H(Key2)

resize() method and pairs

It is not part of the implementation, because of time constraints. The pairs variable, which tracks the amount of Key-Value pairs in the table, is implemented but not specified. The pairs variable is primarily used to decide, if the size of the hash table needs to increase or decrease (resize()).

delete() method

Even with the int Keys and Values the delete() method was difficult to verify, so they got simplified for both hash table concepts. We go into more details in their respective chapters.

Separate Chaining

Before we go into more detail here is a short disclaimer. The different chains of this hash concepts usually have a dynamic size to avoid constant resizing when a lot of collisions happen. So data structures, like linked lists, are used and more static data structures, like arrays, are not very practical. Despite that we still used an array to keep it more simple and in retrospect this was necessary as we will see this chapter.

As mentioned in the previous chapter, we did simplify the delete() method in both hash table concepts. Originally 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. Now we instead just overwrite the key with null.

We are focusing on the version that has int Keys and Values.

Class invariants
The first one is strait forward, because if we have less then one chains we can’t use the hash table.
The second invariants is mostly strait forward. That both arrays are not null should be self explanatory. The arrays shouldn't be same array to avoid array aliasing.
The third invariants says that both the arrays keys and vals are int[] arrays. This is important because we overwrite arrays in those two arrays and KeY checks if keys or vals are subtypes.
The fourth invariants is important, since it would otherwise be possible that we have a Key or Value without a partner.
The fifth invariant is so no sub array is null.
The sixth invariants is the same as the fourth for the sub arrays and is for the same purpose.
The seventh and eighth invariants is to avoid aliasing in the sub arrays.
The ninth invariant is so one key can only appear a maximum of once in a chain.
//TODO: correct Bucket missing

Constructor

hash()

getIndex()
This method returns the index of the given key, if the key is in the hash table.
Otherwise it Returns -1.
The variable iHash is the hash value of the key and therefore the index of the chain

Specification

//I tried to split them in a useful way.
//I will ignore the constructor for now.

invariants

SCA:	9	
LP:	8	

requires

SCA:	13	
LP:	13	

ensures

SCA:	18	
LP:	15	

//might needs somthing more precise
loop_invariants

SCA:	2	
LP:	4 (2*2)	

assignable, but not (strictly_)nothig

SCA:	4	
LP:	3	

quantifiers:

SCA:	21	(1 exists, Rest forall)
LP:	29	(8 exists)

nested quantifiers:

SCA:	5	
LP:	4	

//Something that is specifily for our project.
//		The opposite would be: key != null, no aliasing ....
Unique / Special Properties.
SCA:	2
- Keys and Vals are equally long (2*)
- Each key is at most ones in the same chain

LP:
- Keys and Vals are equally long
- Each key is at most ones in the same chain
- If a key is not null, then the value is also not null. (get)
- between a key and it's hash value is no null value.
		And the variation in findEmpty
- At least one bucket contains a null. (or similar)

Separate Chaining

WithInt vs WithIntKeys

The constructor and the exceptional_behavior of the put() method are both faster in WithIntKeys
- exceptional: exceptions with null are easier to verify.
- constructor: ?

The put method needs about three times as long in WithIntKeys, but it is unclear why. The method increaseArraySize() has no significant changes in the statistics, but is called by the put method. This method is still the a major part of the put verification.

WithInt vs NoEquals

The exceptional_behavior of delete, get and put in NoEquals are significantly shorter. (See above)

The constructor is faster. (See above)

The delete and get methods are slower in the NoEquals variant, most likely because we now use objects as keys instead of ints.
Interestedly enough, while the put method is also a lot slower, it is still faster then the WhitIntKeys variant.

I wasn’t able to finish the increaseArraySize method for NoEquals.

WithInt vs WithEquals

The exceptional_behavior of delete, get and put are significantly shorter then before. (See above)

The hash method need longer, since now an equals is used in the specification.

The getIndex method needs a lot longer. Again most likely because of equals.

The get method needs less nodes, but has a lot more branches (equals uses one if) and symbolic execution. The amount of rules is roughly equal.

I wasn’t able to finish the increaseArraySize, put, delete methods and the constructor.

Linear Probing

WithInt vs WithIntKeys

exceptional_behavior of put and get in WithIntKeys are faster, but delete is slower.
Null and keyDummy?

Delete and put are faster.

overwritePair is slightly slower.

WithInt vs NoEquals

exceptional_behavior of delete, get and put in NoEquals have less nodes, but delete and get have more Branches and Symbolic Executions.
Less nodes is the change partly from iNull to null.
But we also check for keyDummy, so “==” with int seems to be easier then with Objects.

Normal of delete, put, findEmpty, getIndex and hash have more nodes, but especially the branches and Symbolic Executions did increase.

Normal of get has more Nodes and Symbolic Executions, but Branches did not change much.

Same with findEmpty, getIndex and hash.

I wasn’t able to finish the increaseArraySize method.

WithInt vs WithEquals

Every thing, except the constructor, has significantly increased. (Equals)

I wasn’t able to finish the overwritePair, put and delete methods.

SCA vs LP

WithInt

get and getIndex (The explanation should be in an earlier chapter)
The get method in LinearProbing has a smaller proof then the one of SeparateChainingArray, because LiniearProbing calls the hash method in the getIndex method while SeparateChainingArray already needs the hash value in the get method. Also two dimensional array might be a factor too. But the vast proof size increase of getIndex does come from the “between a key and it's hash value is no null value”.

Constructor
The constructor of SeparateChainingArray needs to initialize a two dimensional Array which results in a loop. And the loop_invariant (or maintaining) clause needs to establish all invariants for the arrays inside of keys and vals.

Hash
The hash method in SeparateChainingArray has surprisingly a bigger proof then LinearProbing, even though both method are similar. It because I don’t allow keyDummy. Be removing it the proof seems to be almost the same.

Exceptions
In LinearProbing an exception is thrown if the dummyKey is used and this makes the difference in proof size.

Put and delete
SeparateChainingArray has a bigger proof. The reason is the same as with get.