

Depth-wise Hashing with Deep Hashing Structures

Edward Capriolo

Abstract

The objective of this research is to implement a variation on scatter storage that uses linked arrays as the underlying data structure, a deep table, as well as a multi table probing technique, called depth probing. The paper describes two classes that implement the structure. It also shows simple functionality easily implemented in this structure. Finally scenarios will be discussed in which these concepts will be beneficial.

A two dimensional representation of a Deep Table

To begin to understand how a data structure works it is often helpful to have a two dimensional representation of the structure in mind. A Deep Table could be viewed as a pyramid of arrays that grow constantly by adding new deeper levels to the bottom. The default implementation LinkHashSet constructs the first array with a size of three. When a collision forces the table to grow a new table with a size of the next prime number is created. All operations on this table start from the first level and may continue until the next Set pointer is null.

Determining Insert Location (hashing)

A standard use of a hash is to find an insert location in a table. An object in a structure that uses depth-wise probing can only be inserted at one location in one level. Hashing is done by passing an object into the hash function as well as the current table size. The position where the object should be inserted in that table is returned. If the position is already occupied the same operation is tried again on a lower level until an insertion point is found

The default implementation is the objects hash code modulus table size however this public function can be overridden assuming the one location per table contract is followed. See Appendix E for details

Shared Collision Domains

For a table with linear probing the collision domain is the next element. If a table is poorly optimized a collision chain could be very long resulting in poor insertion and look up performance. Some tables use double hashing and twin prime tables to space out collisions, but this technique still encounters problems with a saturated table. A third bucketing technique involves using a linked list or a second underlying structure to handle a collision.

A deep table is significantly different than all these structures. Objects with identical hash code can only be in one location in each level. If a collision occurs the hash code is recalculated and insertion is tried on the next level.

While this seems like a simple process it results in a remarkable phenomenon: Data always hashes to the same place on a same sized table but insertion order will alter the path data takes on route to an empty place.

This is named a collision path. It is similar to a mean square hash in that it produces a pseudo random hash location that is effected by its own value and other elements in the table. In short the way a data collides until it finds an insertion point is pseudo random, not the hash code itself.

Implementing the Deep Table and DeepHashing

The implementation of LinkHashSet is made to show that this structure can be used to implement the `java.util.Set` interface while DeepHashMap can respectively represent the `java.util.Map` interface.

Java implements HashSet by chaining calls to a HashMap. This implementation works the opposite way a HashMap is two mirrored HashSets, one set representing the key the other the value. All set and map functions in their respective interfaces are implemented although for this research focuses on the common add, contain check, and remove functions.

Add is implemented in this manner:

- 1) find insertion point hash
- 2) if insertion point does not have a user value insert and return true
- 3) if the insertion point is occupied call the equals function
- 4) if the data is equal do nothing and return false
- 5) advance to the next set and return to step 1

Boolean values are returned because all classes implementing set must have this contracted behavior.

Contains and remove follow this same looping process. The function proceeds until it encounters a null value in the table. A dummy variable exist which is equivalent to null is but in place on removals to preserve the collision chain.

The DeepHashTable is implemented with two sets, the key set and the value set, both being DeepHashSets. Elements are stored in a value set in the corresponding level and location of their key. Searching the value set involves an iterative search because the value set is a mirror of the key set.

Predictable operation time

Deep hash tables can be a part of soft real time operations in a way that other hash tables can not. The collision resolution noted in the Collision Domains section can be a key concern to developers. In a time sensation application having a constant operation time is critical. Based on the depth of a deep hash table, the maximum insertion or retrieval time can always be calculated. The time $T = \text{number of levels in the table} * \text{the hash}() \text{ time} * \text{the equal}() \text{ checking time}$. If a sentient program manages the table depth it is possible to guarantee an insertion or retrieval time.

Java Virtual Machine

When evaluating any structure in java it is important to note that the Java virtual Machine is threaded. Time slicing between threads is not guaranteed. At all times during the running of a java program the garbage collector is navigating and analyzing references to find objects that are eligible to be garbage collected. If it finds a large amount of memory to be cleared, it may begin to do so (see Appendix C for an example of this). It is also important to remember that declaring objects and arrays of objects may start memory thrashing and this is beyond programmer control.

Proposed Java API extension

Entire books are devoted to the collections API, surprisingly absent are functions for doing operations on finite sets. In order to preserve the collection API as well as the Set interface, a utility class that can logically handle set operations while delegating most of the implementation to the underlying class is needed. This

class will contain operations to do set operations Union and Intersection as well as several others. The prototypes for this class are found in Appendix A

Iterator Class

The iterator of this set is implemented by an inner class called TopDownIterator. It works by starting at level 0 element 0 traversing the array to arrays length returning all user added elements. It then will then follow next set pointer and repeat the process on the next level until it reaches a null next set.

Priority Structure

The fastest operations on this table happen at level 0. Data that is deeper in the table will take longer to find. It follows that a deep table is a natural priority queue. A user of the table can plug into the fifoInsert() function, shown in Appendix D, and create a priority structure based deep table implementation.

Versatility

A deep table has many attributes that if carefully considered can modify its performance and functionality. Primarily, the starting table size in this implementation can be any prime number, in java sizes are restricted to powers of two. Special constructor can make any size. This structure segments data. In many situations it may be time consuming or impossible to allocate a contiguous block of memory large enough to handle table sizes, that problem is lessened significantly with this structure.

By selecting appropriate table size and depth factor a tradeoff can be made between time it takes to execute insertions and contains checking, as well as curtailing the amount of unused space in a table.

Benefits of using a Deep Table

When a standard hash table approaches the load factor it is typically rehashed. This is an asynchronous process in which every element is removed and replaced in a newer larger table. The reason that deep table structure has a far better insertion speed is that only enters this state when special options are supplied to the constructor and those conditions are exceeded.

When the preponderance of the operations is insertion and iteration a deep hash table will offer great performance. For example, a set operation may involve a

number of finite sets, and a number of set operations are executed. Using an array based set could result in a number of rehashes as the set grows, while a linked list implementation would make a contains check time consuming, still a third possibility is a Tree Set that will have slow insertion and binary contains checking. Any time data seems to indicate insertion, uncertain growth, use of iteration, contains checking without the need for ordering a deep Hashing structure may be your answer.

Drawbacks of the pure implementation

This implementation of this project demonstrates a pure model of deep tables and depth probing. As a result, there is no concept of load factor and no daemon table optimizing threads were added. More research into this topic may show that using combined approach may yield better results.

Conclusion

This research has shown that a new variation of scatter storage exists. Collision paths were found to be a way to distribute data across a table and, surprisingly, to be reminiscent of pseudo random numbers sequences. It is possible to implement deep tables and depth wise probing and to do so effectively. These structures have fast insertion performance and naturally lend themselves to many common computing applications. Built into this implementation is a number of factors such as table start size, depth factor, and table growth size that can customize performance making it a flexible structure for storing data.

Appendix A: Java API Extension

```
public java.util.Set union(java.util.Set s1, java.util.Set s2);
public java.util.Set intersection(java.util.Set s1, java.util.Set s2 );
public java.util.Set powerSet(java.util.Set s1);
public Boolean areDisjoint(java.util.Set s1, java.util.Set s2);
public Boolean existsIn(java.util.Set s1, java.lang.Object element);
public java.util.Set symetricDifference
(java.util.Set s1, java.util.Set s2);
public java.util.Set complementOfWithRespectTo
(java.util.Set s1, java.util.Set s2);
```

Appendix B: Special Terminology

Hashing: For years the word hashing was viewed with negative connotation hashing techniques were called scatter storage

Deep Table: A deep table can consist of infinite arrays chained together

Level: Is an array in a Deep Table, each level of a deep table can be considered a set

Depth Probing: A deep table resolves a collision by following the next pointer and trying the insertion at a deeper level, this concept is called depth probing

Depth Factor: When a table exceeds a depth supplied in a constructor a rehashing algorithm can be invoked.

Appendix C: The garbage collector doing behind the scenes work

```
java.util.HashSet hs = new java.util.HashSet();
java.util.HashSet ht = new java.util.HashSet();
```

```
Integer [] testData = new Integer[60000];
long startTime = System.currentTimeMillis();
for ( int i = 0 ; i < testData.size(); ++i){
    hs.add( new Integer(i) );
}
long stopTime = System.currentTimeMillis();
System.out.println("hs insertion took"+ (stopTime-startTime));
startTIme= System.currentTimeMillis();
```

```

for ( int i = 0 ; i < testData.size(); ++i){
    ht.add( new Integer(i) );
}
stopTime = System.currentTimeMillis();
System.out.println("ht insertion took"+ (stopTime-startTime));

```

Results:

hs insertion took 308

ht insertion took 540

Since there are no longer any references to hs after the print statement the java VM starts to try and garbage collect the structure resulting in a major time difference in the time of the identical process. To solve this manually call `System.gc()` or call a function of hs at the programs end to keep it from being collected while the second set is running.

Appendix D: creating a priority structure

```

net.nfinite.DeepHashSet priorityQueue = new net.nfinite.deepHashSet(40);
priorityQueue.blankAllocate(); // allocates 9 levels
Object veryImportant = new Object();
Object notImportant = new Object();
priorityQueue.fifoInsert( veryImportant, 0);
priorityQueue.fifoInsert( notImportant, 9);

```

Appendix E: Overriding the hash function

This is the default implementation

```

public int hash(object o, int tableSize){
    return Math.abs( o.hashCode / tableSize);
}

```

since hash is a public method it can be overridden through extension or an anonymous class.

```

java.util.Set s = new net.nfinite.HashSet(){
    public int hash(Object o, int table size) {
        // mean square, power hash, etc
    }
}

```