A New Algorithm for a Chained Hash Table with a Partition Driven Binary Search Tree

Joon Sung Park
Professor Michael Biro

Friday, December 11, 2015

1. Introduction

A conventional hash table exploits the property of an array, or an array-like data structure that allows constant time lookup when given an index to the memory location. This is done by assigning a relatively unique index to each of the elements stored in the data structure, and later using those indexes to access elements. A well-designed and implemented hash table supports near-constant time search and other regular operations, and makes it an ideal choice for implementing dictionary-like data structures or associative arrays, among many others.

However, such performance comes with a catch: as the elements in the hash table gets clustered, that is, as the elements' indexes become less and less unique, the performance of a hash table quickly degenerates. A common solution for maintaining a relatively well-distributed hash table without heavy clustering has been to resize the hash table. By increasing the total number of indexes available, the elements in the hash table can be assigned much more unique indexes and the clustering can be undone. Not only well accepted, this resizing of the hash table has also been an unavoidable solution to many of the hash table implementations when dealing with a dynamically increasing set of values.

But for many of the existing hash table algorithms, this process of resizing is very expensive and time consuming. Often times, indexes of every present element have to be recalculated, or "rehashed," and copied over to a newly created table during a resizing process.

As the need to resize arises at a relatively unpredictable moment, the overall performance of hash table becomes unpredictable as well. For example, an insert operation could take constant time at one moment but linear time in the next, as the table may go through the resizing process before it inserts. This makes a hash table a bad choice for backing up real-time applications that cannot afford to pause and resize.

This paper presents a new hash table algorithm that utilizes separate chaining with a form of binary search tree. Ultimately, the presented algorithm suggests a new way for designing hash tables, with a potential for allowing rapid resizing by avoiding the rehashing process.

2. Hash Table and Its Terminology

In computer systems, an array is a continuation of memory slots with each of the slots containing some value. And when given an index of an array, retrieving the value from that index of the array can be done in constant time. As mentioned in the introduction, a hash table utilizes this aspect of an array. At the core of a hash table is an array of "buckets" with each array slot representing a bucket (In the figure above, there are 16 buckets in all, with each bucket's array

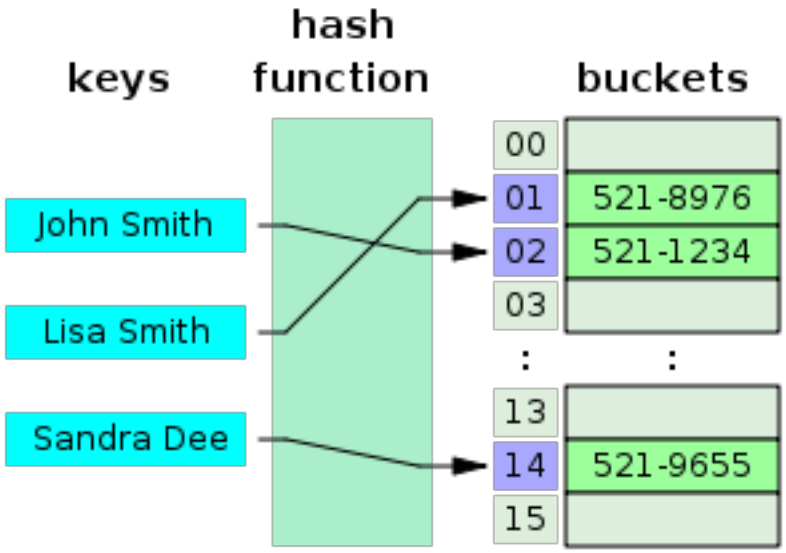

Figure 1, A Typical Hash Table Layout

Taken from https://en.wikipedia.org/wiki/Hash_table

index represented with a number in between 0 and 15). When a hash table is given a key for the insert operation to store new data, a "hash function" translates the key into one of the index numbers for buckets. Then the given key and some values related to the key are stored in the bucket designated by the calculated index.

A problem arises, however, when more than one key gets hashed into the same bucket, or as commonly called, "collide." A common remedy for collisions is "separate chaining." With separate chaining, each bucket is represented as a separate data structure, allowing it to represent more than one key and value pair. Of course this solution comes with a cost; the more elements represented in one bucket, the more work needed to locate the target element after hashing into a bucket. As the number of elements represented in a bucket increases, the runtime complexity of regular operations, such as lookup, starts to digress from constant time, resembling more of the chained data structure's complexity of respective operation.

So essentially, the strength of a hash table that grants near-constant time operations comes from the table's ability to maintain each bucket as light as possible. In part, this is achieved through a well-implemented hash function that evenly distributes the elements throughout the buckets. But more importantly, this is achieved through allocating more buckets to the hash table so there would be more room for unique indexes. This process of dynamically allocating more buckets to a given hash table is called "resizing." In many hash table algorithms, this resizing process takes place when the number of elements to the number of buckets ratio, which is called the "load factor," exceeds a certain threshold.

However, the resizing process is a costly one as mentioned above, and it may be worthwhile to note now why it is so. When calculating bucket indexes for each element, many of the hash functions include a modular operation that uses the number of buckets. So when the number of buckets changes during a resizing process, the previously calculated indexes are incorrect in the newly expanded hash table. Therefore, the bucket indexes of all existing elements and newly added elements have to be recalculated (rehashed) and copied over to the correct buckets in the newly created hash table. This is a linear time operation in relation to the number of elements, as the process has to manipulate every element present in the hash table.

The new hash table algorithm that will be presented in the following pages avoids the rehashing process entirely. This is done through separate chaining the hash table with a special form of a binary search tree.

3. Binary Search Trees

3.1 Conventional Binary Search Tree

Before describing the binary search tree that will be chained to the currently presented hash table, it would be useful to layout a more conventional binary search tree and consider its complexity. This section will not go over in depth the algorithm for a conventional binary search tree and its various operations. Rather, understanding the general shape of a binary search tree and its search operation's complexity would be sufficient for the scope of the current discourse. From this section on, the complexity will be formally discussed in big O notation.

An element in a binary tree has two children often denoted as left and right children. In a binary search tree, every element stores a unique key (and optional associated values) where the key of any element is greater than all keys stored in the left sub-tree, and smaller than all keys stored in the right sub-tree. For searching a specific key in a binary search tree, the operation starts from the root of the tree, comparing the root's key to the key that is being searched. If the root's key is equal to the key that is being searched, the operation returns true, designating that the key is found. If the root's key is smaller than the one that is being searched, the operation proceeds to search in the right sub-tree, and if the root's key is bigger than the one that is being searched, the operation proceeds to the left sub-tree (in these two cases, the recursive procedure takes place in the following step using the root of the sub-tree). If the tree finishes before the wanted key is found, the operation determines that the key does not exist in the current binary search tree and returns false.

Essentially, assuming the size of the right sub-tree and the left sub-tree are relatively similar, each step of the search operation roughly halves the scope of search. The complexity of this operation is commonly known to be O(log n) where n is the number of elements in the tree. This can be shown with a proof by strong induction.

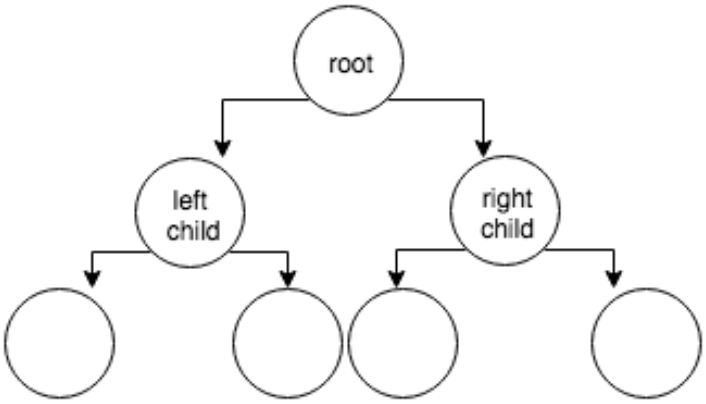

Figure 2, Binary Search Tree Layout

Let BS(n) be the total amount of time needed for a binary search and X be some constant representing the amount of time needed to run trivial steps of the algorithm. The current claim is that $BS(n) \leq X * \log n + BS(1)$. The base case for the strong induction is established when n = 1:

$$X * \log 1 + BS(1) = X * 0 + BS(1) = BS(1)$$

Then the inductive hypothesis would be for all k such that $k < n$,

$$BS(k) \leq X * \log k + BS(1)$$

Now, notice that when $n > 1$, there are three possibilities that need to be considered: the key being searched for is 1) equal to the current root's key, 2) less than the current root's key, 3) greater than the current root's key. In the first case where the key being searched for is equal to the current root's key, the only steps required are to return true and confirm that the key was found. These steps are trivial and can be simply represented as X. That is,

$$BS(n) \leq X \leq X * \log n + BS(1)$$

For the latter two cases, the steps are slightly more involved. When the operation fails to locate the desired key in the current's root, it will have to continue searching in the left or right sub-tree

that is about a half the size of the current tree. Therefore, in addition to some trivial steps, X, the operation will have to perform a search on half of the current root's children. This can be represented as,

$$BS(n) \leq X + BS(1/2 * n)$$

Note, $(1/2 * n) \leq n$. By inductive hypothesis, the above representations can be reduced to,

$$\begin{aligned} BS(n) &\leq X + (X * \log (1/2 * n) + BS(1)) \\ &\leq X + X * \log (1/2 * n) - X + BS(1) \\ &\leq X * \log n + BS(1) \end{aligned}$$

With this above proof that $BS(n) \leq X * \log n + BS(1)$, that the binary search takes O(log n) complexity is clear. While not discussed in this section, the complexity for other common operations on a binary search tree, such as insert and remove key, takes O(log n) complexity as well for essentially the same argument presented above. This is the case as these operations usually "search" for the place where a key can be inserted or removed, and take some trivial amounts of work to actually modify the tree.

3.2 Partition-Driven Binary Search Tree

Partition-driven binary search tree (PDBST) is similar to a regular binary search tree in that each element in the tree has two children, and maintains a certain order. However, a PDBST has a predetermined and unchangeable capacity that is given when the tree is created, and cannot contain more number of elements than the capacity. When determining whether a given child should be in the left or right sub-tree, PDBST, instead of using the parent key's value, uses the values that "partition" the capacity of the tree into some number of pieces, where "some number" is determined by 2^k, with k being the level of an element within the tree.

Consider a PDBST with the capacity of 100. The first element inserted will be the root of the tree (also note that the level of the root is 0). When inserting the second element, a conventional binary search tree would check if the new element's key is greater or smaller than the current root. However, PDBST will check if the new element's key is greater or smaller than

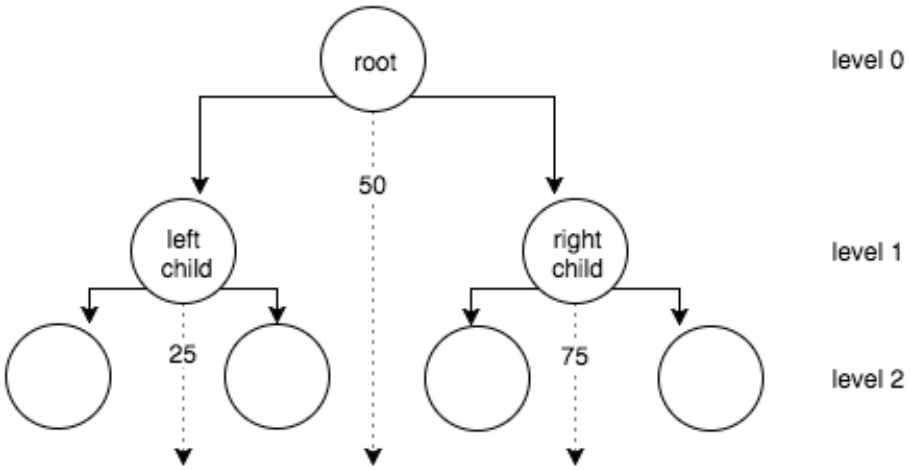

Figure 3, PDBST with Range 0 to 100 (partition values shown in dotted lines)

the value 50, which partitions the current PDBST's capacity into 2^1 parts (note here that keys are assumed to be numeric values). If the new key is greater than 50, it will be inserted as the right child of the current root, and if it is smaller than 50, it will be inserted as the left child of the current root.

Imagine now that both the left and right children of the root have been filled, and the next value is to be inserted. Much like a conventional binary search tree, PDBST will first search for the place where the value should be inserted. It checks to see if the newly added element's key is greater or smaller than 50. If it is greater than 50, it moves on to the right child, and it is smaller, to the left child. Let's say, for example, the new key is smaller than 50. Instead of comparing the new key to the left child's key, PDBST compares the new key to the value 25 as the multiples of 25 partitions the capacity into $2^2 = 4$ parts. If the new key is smaller than 25, it will be inserted as the left child of the level 1 left child, and if greater than 25, it will be inserted as the right child of the level 1 left child.

Removing an element in a PDBST takes caution as not to disconnect the tree. The only elements that can be removed safely are the elements without any children. Therefore, to remove an element in a PDBST, the element that is to be removed should be swapped with one of its children that have no child of its own and then removed (children here can be one of the two direct children, or any other children down the tree). Notice that this remove operation is done preserving the property of PDBST.

Searching for a key in PDBST will essentially take the same form of operation as the one in a conventional binary search tree, except that it will be relying on the partitioning values instead of the keys of the roots when deciding whether to proceed left or right. The concept of halving the number of elements to be searched at each level remains unchanged. Therefore, the complexity for the search operation, as well as other regular operations described above, should be O(log n) by the same argument presented for the binary search tree's complexity.

4. Hash Table with Separate Chaining to PDBST

As stated in the introduction, the driving motivation for the current discourse was to present a hash table algorithm that avoids the rehashing process entirely during a resizing operation. While the hash table presented here is similar in structure with other chained hash tables, it answers to this motivation by taking a full advantage of the PDBST's properties.

4.1 The Table's Layout and the Hash Function

There is no doubt that if a hash function is "injective," that is, if it maps every input to a unique bucket in the hash table, it is perfect. Such a function would be able to directly locate any entries without additional searching, giving the constant time complexity for all its regular operations. However, to support such a function would require preemptively allocating bucket spaces for every possible input even when there is no guarantee that the most of the allocated buckets will be used. So unless the number of every possible input is considerably small or there is a guarantee that most of the allocated buckets will be used, supporting a perfect hash table is highly impractical. Of course, this is true for the currently presented algorithm as well.

But while allocating buckets for every possible input is impractical and bucket collisions unavoidable, it is possible to exploit the injective nature of a perfect hash function in order to avoid rehashing. The current algorithm starts by finding out the upper bound for the number of all possible inputs. For example, if the inputs were expected to be 5 alphabet characters, this would be 26^5. If the upper bound cannot be clearly defined, however, the value for the upper bound should be set to some arbitrarily large number that is guaranteed to be greater than all possible inputs. Larger bound will not degrade the performance of the current hash table

algorithm. Also, in the current algorithm, the hash table will not be able to contain more elements than what the upper bound dictates due to its close pairing with its chained data structure, PDBST. Therefore, it is absolutely critical that the upper bound set at the beginning is large enough to contain the number of all possible elements.

As a chained hash table, each bucket of the current hash table is represented as a PDBST. Now, recall that PDBST has a range; this is crucial to the current hash table algorithm. The range of PDBST is determined by the number of buckets present (hash table's capacity) and by the upper bound in such a way that the number of buckets partitions the estimated number of all possible inputs. For example, if the hash table's current capacity is 10 and there are 1000 possible inputs, the range of the first bucket should be 1 to 100, second 101 to 200, and so on, ending with the last bucket that ranges from 901 to 1000. When an input is hashed, unlike in a regular hash table, the input is hashed to a unique integer that is less than the upper bound given in the previous step, and then put into the PDBST whose range contains the hashed value (notice that this works essentially the same way as a perfect hash function with the injection between all possible inputs and unique integers, except no physical memory space is allocated to match the injection).

It should be apparent that the methods and complexity for the regular operations, such as search, insert, and remove follows closely with those of a regular hash table chained with binary search trees. For example, a search process in the currently presented hash table would take some constant time for hashing the entry, and then traveling the PDBST in the bucket if necessary. So long as each bucket is not concentrated with elements, the operation takes near constant time, and logarithmic time otherwise. However, this is not the case for the resizing operation of the hash table.

4.2 The Resizing Algorithm

Note again that to resize the hash table is to increase the number of buckets that are present in the table. With the current hash table algorithm, this can be done simply by dividing in half each PDBST representing the buckets. A property of PDBST with a range of x to y is that all elements to the left of the root is another PDBST with the range of x to $(x + y)/2$ and all elements to the right of the root is PDBST with the range of $(x + y)/2$ to y. Then, to increase the number of buckets to twice the current number, for each PDBST representing a bucket, the left sub-tree and

right sub-tree will each represent a bucket in the resized tree, and the root of the main tree will be inserted back into either the left or the right sub-tree.

This method maintains all the properties of the hash table prior to resizing, and can be done without any rehashing process. The complexity of this operation takes only logarithmic time at worst, and constant time at best in relation to the number of elements present in the hash table as the only non-trivial operation is the insertion of the root of PDBST back into one of its sub-trees. But this operation still takes linear time in relation to the number of bucket, as every bucket has to be checked to see if it's empty, and if not, be divided into two sub-trees.

5. Further Issues and Potentials

The current research started with the ultimate goal of achieving dynamic resizing of a conventional hash table with some slight modifications. The presented algorithm tried to tackle this goal by entirely avoiding the rehashing process during resizing with a separate chaining to PDBST whose properties are exploitable in the greater scope of the hash table. How much was gained, and how much was lost in this effort to avoid the rehashing process?

The obvious strength of the presented algorithm is the fact that it does not need to rehash during the resizing process. As the hash function gets more complicated and involved, this becomes even greater strength. Also, because most of the PDBST remains untouched during the resizing, and can be reused afterwards, the current algorithm is especially efficient when many elements are represented in each PDBST – that is, when the buckets are concentrated. However, in most real-life hash tables, the load factor is often below 1 (for example, Python's load factor is at 2/3 and Java at 0.75). And this means buckets are rarely concentrated enough to fully benefit from the avoiding rehashing process.

Therefore, it is uncertain at the moment how much performance improvement the current algorithm will bring in relation to the conventional hash table algorithm, and further testing would be required to fully understand its efficiency. However, as concluding the current discourse, it would be worth recognizing that an important potential for the current algorithm may lie in its extreme parallelizability. As stressed throughout, the most important aspect of the current algorithm is the fact that each bucket is a result from partitioning the entire hash table. And an important side-effect of this in the computational perspective is that operations on this

hash table can be performed on a multiple computing platforms (CPUs) with each platform making changes to its designated buckets simultaneously, without much need for synchronization. This could bring high performance boost to the current hash table's regular operations that are not possible in conventional hash tables. But of course, further testing would be necessary in this regard as well.

6. References

(1) Weiss, Mark Allen. *Data Structures and Algorithm Analysis in C.* Fourth ed. Print.

(2) Bryant, Randal E., and David R. Hallaron. *Computer Systems: A Programmer's Perspective.* 2nd ed. Boston: Prentice Hall, 2011. Print.