N-Dimensional Array
Ziggurat Indices

Copywrite 2022 Adligo Inc
Contact info@adligo.com
Author Scott Morgan
Dates 2022-02-01 – 2022-02-20
Version v0.2

Table of Contents

Abstract Summary

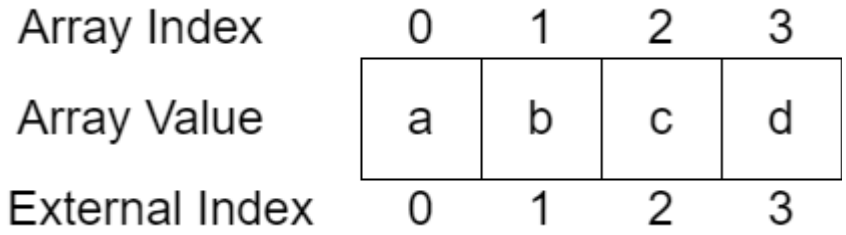
This paper explores the space time complexity of the use of a unbounded universe of items referenced in arrays to solve the problem of indexing. The basic structure involves starting with a simple array of slots for items. Then in order to grow the data structure past the initial array length an additional array is created with it's zero slot pointing to the original array,

When this recursive growth structure is combined with bit masking and shifting techniques similar to those in use in Van Emde Boas trees interesting efficiencies can be achieved in the classic space time complexity trade offs.

Basic Ziggurat Structure

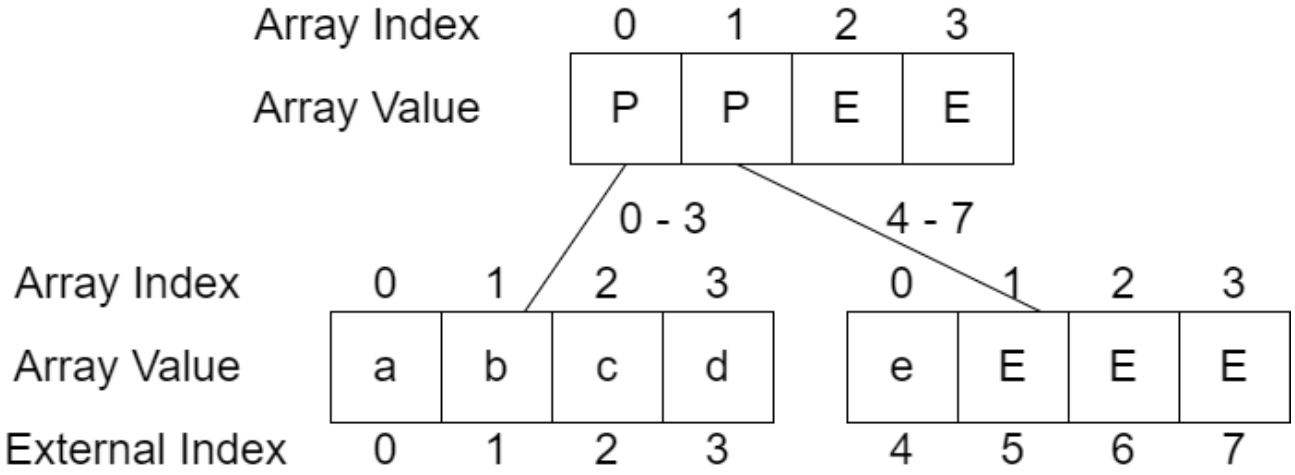
This basic structure starts with an array with a size of **2 ^ E = B. Where E is the exponent of 2 used to identify the base B.** This is interesting because unlike most algorithms it allows for a configurable base (instead of simply using base 2), for example in this diagram the base is four.

Figure 1:



Then once you need the ability to add a 5th item to the data structure a new array is created, which references the old array as one of it's elements. I have called this a ziggurat because of it's flat top shape which is different from a pyramid. Although this has been more commonly called a tree, the most common upside down whiteboard representation of CS trees look more like a pyramid. The following Figure 2 diagram illustrates this next stage in the data structure.

Figure 2:

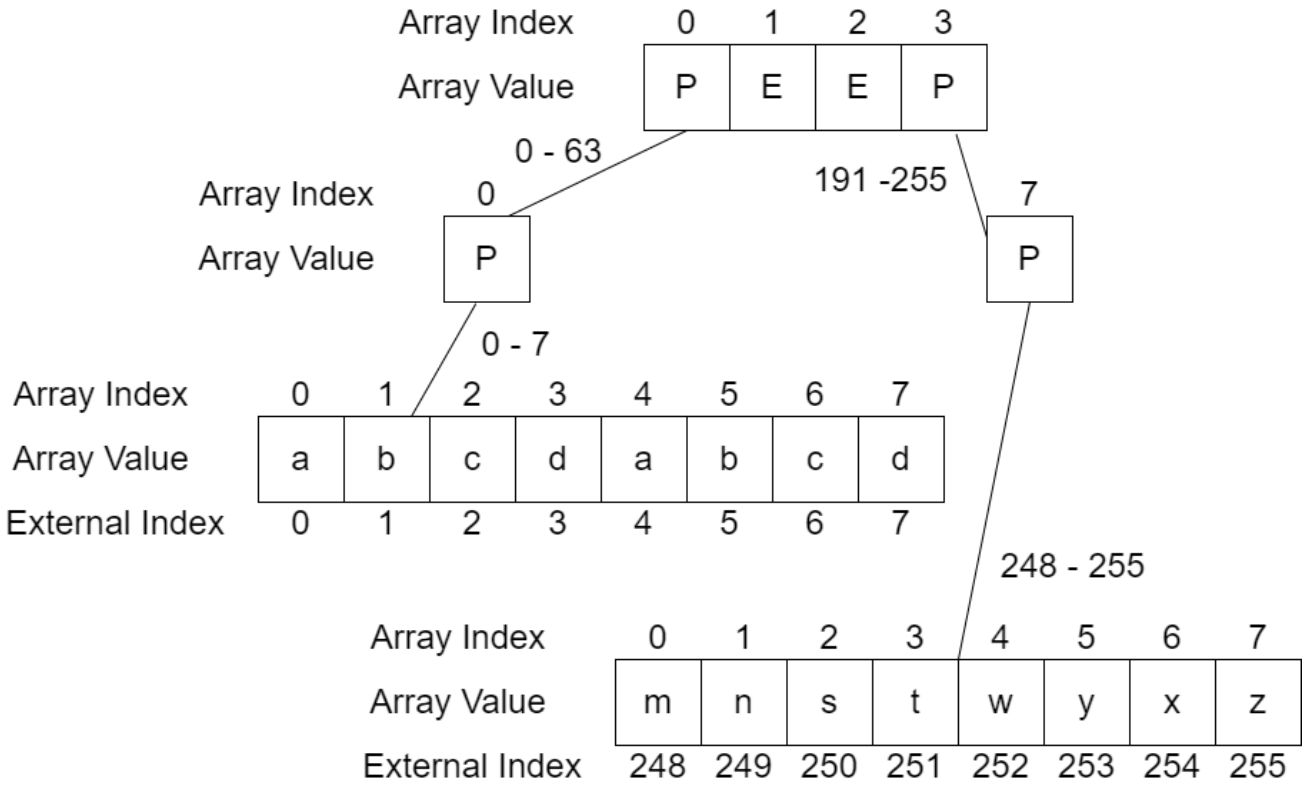


Note in the above diagram the capital P represents a present reference and the capital E represents a empty reference.

Space Complexity / Cost of a Ziggurat Index

The space cost is complex to calculate with something like Asymptotic Analysis for a number of reasons. With the simple structure it is definitely greater than O(n) noting the empty slots and slots used for present references. In addition having a variable base, which also plays into the typical base 2 of Asymptotic Analysis causes issues. Finally the space usage is quite different for sparse and dense usage. Noting these challenges I believe the space to be between O(n^2) for sparse usage and O(log n) for dense usage.

Additionally, in the spirit of configuration if we compress the size of the arrays at or near the bottom (often called leaves in tree structures). We have the ability to trade time efficiency for space efficiency. The following Figure 3 diagram illustrates how this compression could occur.



Note the two 0-7 arrays of size 8 are on the same tier of the ziggurat, they were offset to make the diagram fit on the page better. Also note how the configurable nature of this data structure provides the ability for the user of the data structure to optimize a structure already in use with simple configuration.

Time Complexity / Cost of a Ziggurat Index

The main challenge of randomly accessing elements in this ziggurat index structure revolves around the inefficiencies of division algorithms. In order to overcome this challenge, bit shifting is employed to perform division like operations to identify the sub array. For example if you were looking for the 5th element with index 4 in Figure 2 you could divide 4/4 and get 1 which is the index of the 2nd element of the top of the and the remainder 0 is the index in the right sub array. Because the size of the arrays is always **2 ^ E where E is the exponent used to identify the base,** bit shifting of the number four represented in binary as;
100

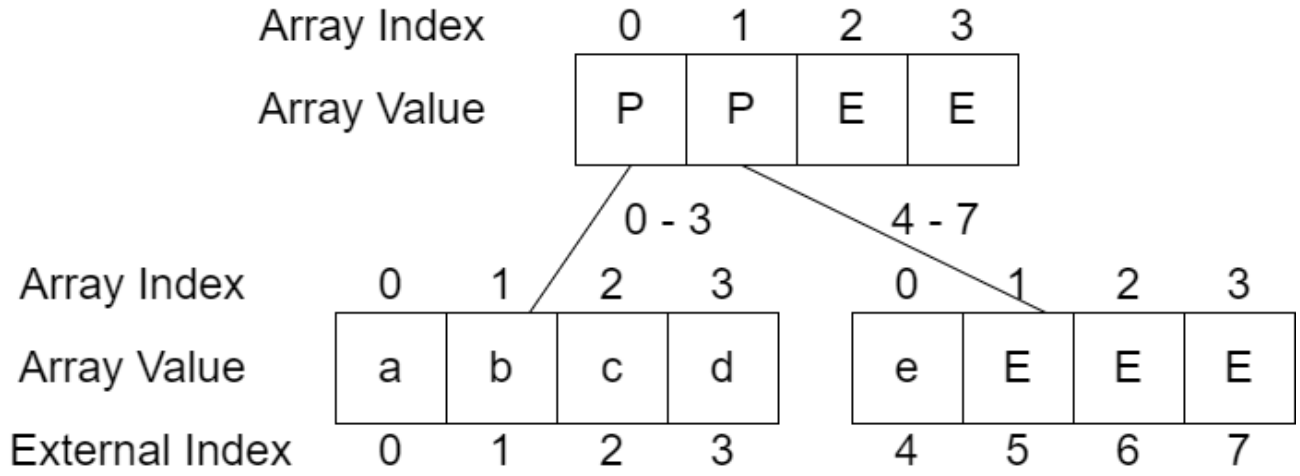
By 2 ^ **2** = 4
can shift the binary string 100 two slots to the right to find the binary string;
100 >> **2** = 1

In addition you can multiply of the quotient 1 by the exponent of the base (2) to shift the quotient two slots to the right;
1 << **2** = 100

Then you can subtract the multiple of the quotient from the initial number to get zero;
100 – 100 = 0

All of these operations complete in O(1) time, causing you to have 3x O(1) to identify the indexes of the arrays.

Figure 2



Bit Masking Time Complexity Optimizations

Finally if we store the presence or lack of presence in each array as a binary string, technique from Van Embe Boas trees, for example;

Figure 2 Binary Array Strings;
Top Center; 1100
Bottom Left: 1111
Bottom Right: 1000

This allows us to do bit masking to identify for example if we are looking at the Bottom Right slot 0 and we want to stream the previous and next element in index order. We can go up to the top center node and discover with a bit mask or simple if statement that there are ones set to the left of the top center one slot. Then move to the Bottom Left mode and perform a bit mask style binary search, i.e.
Bottom left binary and top right half of bottom left
1111 & 11 = 11
Then half that
11 & 1 = 1

A similar pattern can be followed to identify that there are no elements greater than the bottom right slot zero. This can be achieved by masking the right of the bottom right again with a binary search like mask;
000 & 111 = 0
So after no items are found in the bottom right greater than the zero node, we move up the Ziggurat's tiers, and recurse. We then bit mask the top center node looking at the right two bits with the mask;
00 & 11 = 0
This results in the knowledge that nothing is after slot zero in the bottom right array.

Comparison of the Ziggurat with more common Data Structures

Time Complexity Comparison

Also note the following interesting properties of this data structure when compared with other common data structures. In particular note the high lights of green rows

Operation	Ziggurat Index	ArrayList	Hashtable
Add	O(1) to O(log log n)	O(1) to Amortized O(n^2)	O(1) to Amortized O(n^2)
Delete	O(1) to O(log n)	O(1)	O(1) to O(log n)
Get / Random Access	O(1) to O(log log n)	O(1)	O(1) to O(log n)
Imperative Loop / Random Access	O(n log n)	O(n)	O(n) to O(n log n)
Stream Operations	O(n)	O(n)	O(n) to O(n log n)
Update	O(1) to O(log log n)	O(1)	O(1) to O(log n)

Space Complexity Comparison

In the space chart I use the letter u to denote the universe (max capacity) of the collection, and n to indicate the items in the collection. In addition I use the letter p to indicate the space used by empty slots (pointers to null).

	Ziggurat Index	ArrayList	Hashtable
Space Consumption	O(u) to O(1)	O(u) to O(1)	O(u) to O(1)

Although not much can be determined by the Asymptotic Analysis of the space use by these data structures using the worst case scenario formulas. Some deeper rhetoric gives further insight.

Ziggurat Index

Similar to a simple array, Hashtable or Java's ArrayList you could create a Ziggurat Index and only put a single element in it, which would result in horrid use of space. However with a ziggurat index, once you have two or more dimensions of arrays you could only use something more similar to O(n log u) space.

Further if you turn on the leaf most compression of tiers at or near the leaf, a significant amount of space (i.e. less references / pointers to null) occur, gets reclaimed.

ArrayList

In many ways the array list provides a nice benchmark as it is backed by a single array. The main drawback of the amortized method is that it will at seemingly random time create a large amount of unused memory (i.e. Java's slots for pointers to null), which MAY never actually get used. In the worst case this is roughly ½ Integer.MAX_VALUE in java * 8 (64 bit java) or 8 billion bytes. I believe this worst case stems from the Amortization of measuring performance which is NOT needed by the Ziggurat Index data structure.

Hashtable

While the Hashtable is great tool, the obvious difference between a Ziggurat Index and a Hashtable is that a Zigguat Index does NOT require any sort of hash function. In addition a Ziggurat Index MAY be used as a hashtable.

Conclusions

A Ziggurat Index makes for a incredibly versatile data structure, allowing users to optimize for time or space after usage as necessary. Compression of leaf or near leaf nodes can easily be added or removed. The main constraint of Ziggurat Indices seems to be the max integer of the programming languages, for example in Java this is the BigInteger.

In addition a Ziggurat Index MAY be used to implement other advanced algorithms like Tries and Heaps.

Citations

Big Integer Max Size
https://stackoverflow.com/questions/12693273/is-there-an-upper-bound-to-biginteger#:~:text=The%20number%20is%20held%20in,2%20%5E%2032)%20%5E%20Integer.

Division Algorithms
https://en.wikipedia.org/wiki/Division_algorithm

Merkel Forrests
Still looking for the reference that gave me this idea, I think it's a MIT lecture.
https://www.youtube.com/watch?v=Bqs_LzBjQyk

Van Embe Boas trees
https://en.wikipedia.org/wiki/Van_Emde_Boas_tree