Applications & Implementation of Zip Tree and Bloom Filter

November 7, 2022

Patel Het Rasikkumar (2021CSB1119) ,
Vavdiya Harsh (2021CSB1139) ,
Ayush Sahu (2021CSB1077)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Vinay Sir

Summary: We introduce the zip tree, a form of randomized binary search tree that integrates previous ideas into one practical, performant, and pleasant-to-implement package. A zip tree is a binary search tree in which each node has a numeric rank and the tree is (max)-heap-ordered with respect to ranks, with rank ties broken in favor of smaller keys. Zip trees are essentially treaps (Seidel and Aragon 1996), except that ranks are drawn from a geometric distribution instead of a uniform distribution, and we allow rank ties. These changes enable us to use fewer random bits per node. We perform insertions and deletions by unmerging and merging paths ("unzipping" and "zipping") rather than by doing rotations, which avoids some pointer changes and improves efficiency. The methods of zipping and unzipping take inspiration from previous top-down approaches to insertion and deletion (Stephenson 1980; Martínez and Roura 1998; Sprugnoli 1980). From a theoretical standpoint, this work provides two main results. First, zip trees require only O(loglogn) bits (with high probability) to represent the largest rank in an n-node binary search tree; previous data structures require O(logn) bits for the largest rank. Second, zip trees are naturally isomorphic to skip lists (Pugh 1990), and simplify the mapping of (Dean and Jones 2007) between skip lists and binary search trees.
Bloom Filter is a probabilistic data structure which saves memory space and time efficiently, but the trade-off remains as false positives. It tells us if the value is definitely not in the input stream or maybe in the stream. Since Standard Bloom Filters do not support deleting elements various variants of Bloom Filters have been introduced. Due to the positives of Bloom Filters like compact summarization of streaming data, it has gained importance in applications that use higher volumes of data like in network traffic management, database management and cloud security. In this paper we implement a Bloom Filter to test membership of URLs and provide a warning to malicious websites or access to kid friendly websites. By creating a second Bloom Filter with maximized size we cross verify the query results of the first Bloom Filter to declare with absolute certainty of no false positive result.

1. Introduction

At a high level, a zip tree is a

- randomized balanced binary search tree,
- that is a way of encoding a skiplist as a BST, and
- that uses a pair of operations called zipping and unzipping rather than tree rotations.

The first bullet point - that zip trees are randomized, balanced BSTs - gives a feel for what a zip tree achieves at a high level. It's a type of balanced binary search tree that, like treaps and unlike red/black trees, uses randomization to balance the tree. In that sense, a zip tree isn't guaranteed to be a balanced tree, but rather has a very high probability of being balanced.

The second bullet point - that zip trees are encodings of skiplists - shows where zip trees come from and why, intuitively, they're balanced. You can think of a zip tree as a way of taking the randomized skiplist data structure, which supports all major operations in expected time O(log n), and representing it as a binary search tree. This provides the intuition for where zip trees come from and why we'd expect them to be so fast.

The third bullet point - zip trees use zipping and unzipping rather than tree rotations - accounts for the name of the zip tree and what it feels like to code one up. Zip trees differ from other types of balanced trees (say, red/black trees or AVL trees) in that nodes are moved around the tree not through rotations, but through a pair of operations that convert a larger chain of nodes into two smaller chains or vice-versa.

2. Zip Tree

A zip tree is a binary search tree in which each node has a numeric rank and the tree is (max)-heap- ordered with respect to ranks, with ties broken in favor of smaller keys: the parent of a node has rank greater than that of its left child and no less than that of its right child. We choose the rank of a node randomly when the node is inserted into the tree. We choose node ranks independently from a geometric distribution with mean 1: the rank of a node is non-negative integer k with probability

$$1/(2^{(k+1)})$$

.

One can view a zip tree as a treap but with a different choice of ranks and with different insertion and deletion algorithms.Our choice of ranks reduces the number of bits needed to represent them from O(logn) to log(log(n))+O(1).

One can view a zip tree as a compact representation of a skip list. There is a natural isomorphism between zip trees and skip lists. Given a zip tree, the isomorphic skip list contains item e in the level-k sublist if and only if e has rank at least k in the zip tree. Given a skip list, the isomorphic zip tree contains item e with rank k if and only if e is in the level-k sublist but not in the level-(k + 1) sublist.

The structure of a zip tree is uniquely determined by the keys and ranks of its nodes.

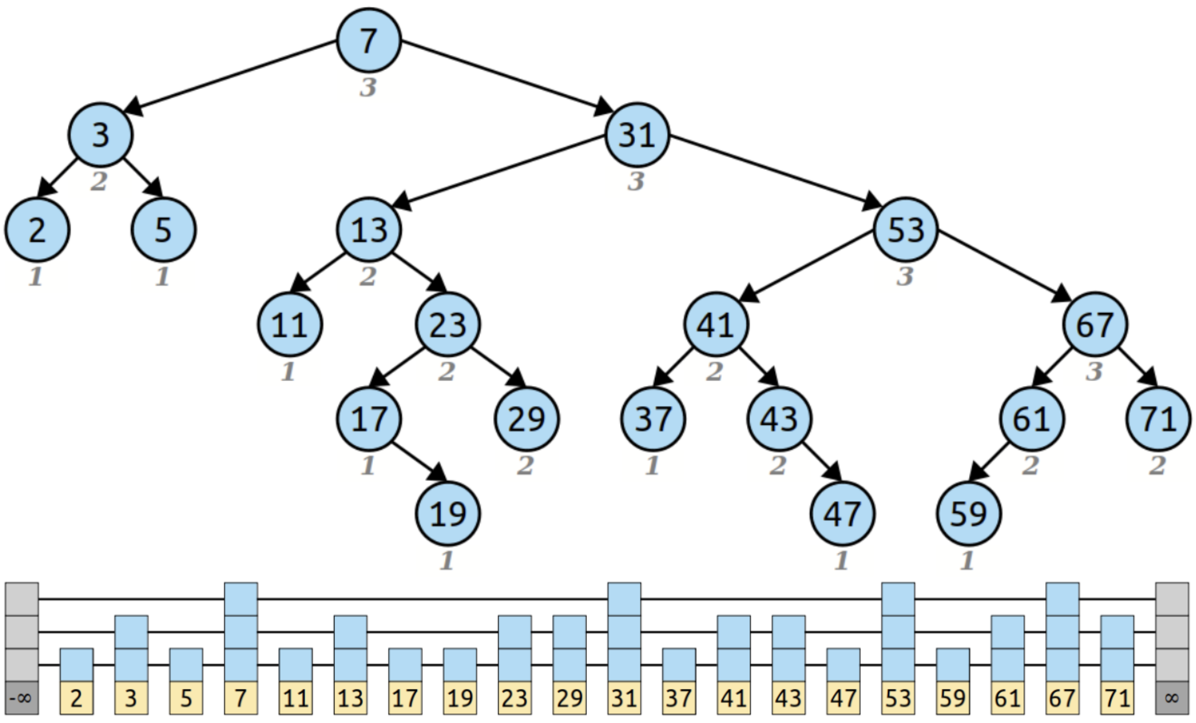
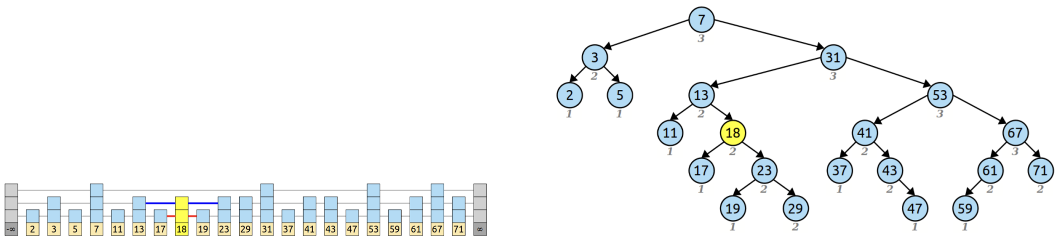


Figure 1: Representation Of Zip Tree Using Skip List.

3. Figures, Tables and Algorithms

3.1. Figures



(a) Skip list insertion (b) Zip tree insertion

Figure 2: Insertion:Unzipping

Complexity

	Average Complexity	Worstcase Complexity
Search	O(n)	O(n)
Insertion	O(1)	O(log n)
Delation	O(1)	O(log n)

Table 1: Time Complexity Analysis Of Zip Tree

3.2. Algorithms

Algorithm 1 Recursive Versions of Insertion and Deletion

```
insert(x, root):
if $root = null$ then
    x.left ← x.right ← null; x.rank ← RandomRank; return x
end if
if $x.key < root.key$ then
    if insert(x, root.left) = x then
        if x.rank < root.rank then
            root.left ← x
        else
            root.left ← x.right; x.right ← root; return x
        end if
    end if
else
    if insert(x, root->right) = x then
        if x.rank <= root.rank then
            root.right ← x
        else
            root.right ← x.left; x.left ← root; return x
        end if
    end if
end if
return root
```

Algorithm 2 Iterative Versions of Insertion and Deletion

```
insert(x):
rank ← x.rank ← RandomRank
key ← x.key
cur ← root
while cur != null and (rank < cur.rank or (rank = cur.rank and key >cur.key)) do
   prev ← cur
   cur ← if key < cur.key then cur.left else cur.right
end while
if cur = root then
   root ← x
else if  key < prev.key  then
   prev.left ← x
else
   prev.right ← x
end if
if cur = null  then
   x.left ← x.right ← null; return
end if
if key < cur.key then
   . x.right ← cur
else
   x.left ← cur prev ← x
end if
while cur!=null do
   fix ← prev
   if cur.key < key then
      repeat
         prev ← cur; cur ← cur.right
      until cur = null or cur.key > key
   else
      repeat
         prev ← cur; cur ← cur.left
      until cur = null or cur.key < key
   end if
   if fix.key > key or (fix = x and prev.key > key) then
      fix.left ← cur
   else
      fix.right ← cur
   end if
end while
```

4. Theorem And Lemmas

Theorem 4.1. *The expected depth of a node in a ziptree is at most (3/2)log n+O(1).For c ≤1,the depth of a zip tree is O (c log n) with probability at least*

$$1 - 1/n^{(c)}$$

, where the constant inside the big "O" is independent of n and c.

Proof. One can prove this theorem using results from [6], but for completeness we prove it from scratch. The expected rank of the root is at most lg n + 3 by Theorem 3.2. By Lemmas 3.3 and 3.4, the expected number of ancestors of a node x, including x, is at most (3/2) lgn + O(1). The second half of the theorem follows from the high-probability bounds in Theorem 3.2 and Lemmas 3.3 and 3.4. the expected number of nodes visited during a search in a zip tree is at most (3/2)logn+O(1), and the search time is O (logn) with high probability.

Theorem 4.2. *The expected number of pointer changes during an unzip or zip is O (1). The prob- ability that an unzip or zip changes more than k + O(1) pointers is at most*

$$2/c^{(k)}$$

for some c > 1 independent of k.

Proof. The expected number of pointer changes is at most one plus the number of nodes on the unzipped path during an insertion or the two zipped paths during a deletion. For a given node x, these numbers are the same whether x is inserted or deleted. Thus, we need only consider the case of deletion. The probability that x has rank k is 1/2k+1. Given that x has rank k, the expected number of nodes on the two zipped paths is at most (3/2)k + 2 by Theorem 3.6. Summing over all possible values of k gives the first half of the theorem.

Choosing $\delta = 1$ in the second half of Theorem 3.6, we find that if x is a node of rank at most k/3, the number of parent changes during its insertion or deletion is more than k + O (1) with probability at most $2/e^{(k/18)}$. Thus, $c = e^{(k/18)}$ satisfies the second half of the theorem.

the expected time to unzip or zip is O (1), and the probability that an unzip or zip takes k steps is exponentially small in k.In some applications of search trees, each node contains a secondary data structure, and making any change to a subtree may require rebuilding the entire subtree, in time linear in the number of nodes.

Theorem 4.3. *The expected number of descendants of a node of rank k is at most* $[3(2^k)]$*-1. The expected number of descendants of an arbitrary node is at most (3/2)logn+2.*

Lemma 1 :
The structure of a zip tree is uniquely determined by the keys and ranks of its nodes.

Lemma 2 :
Let x be a node and let be the number of low ancestors of x. Then, the expected value of l is at most $k + 1$*, and for any*$d > 0, l \leq (1 + d)k + 1$ *with probability at least* $1 - e^{\frac{d^2 k}{2}}$

Lemma 3 : *Let x be a node and let h be the number of high ancestors of x. Then, the expected value of h is at most* $k/2$*, and for any d> 0,h* $\leq (1 + d)k/2$ *with probability at least*$1 - e^{-\frac{d^2 k}{2(2+d)}}$

Bloom Filter

5. Introduction

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. False positive means, it might tell that given username is already taken but actually it's not.
Interesting Properties of Bloom Filters

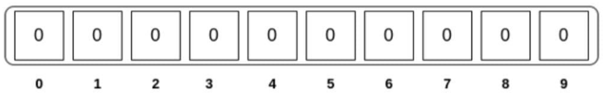
- Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
- Bloom filters never generate false negative result, i.e., telling you that a username doesn't exist when it actually exists.
- Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete "geeks" (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting "nerd" also Because bit at index 4 becomes 0 and bloom filter claims that "nerd" is not present.

6. Working Of Bloom Filter

Suppose we want to compare two strings. Instead of storing the entire string, we compute the hash of the string and then compare the hashes. Computing hash and then comparing them takes O(1) time instead of the O(n) time it would take to compare the strings directly.

One drawback of this method is the limited range of the hash function. The hash function h1 ranges from 0 to r1-1. We cannot identify more than r1 strings uniquely with this method as at least two strings will have the same hash value. To compensate for this, we can use multiple hash functions. We use k hash functions, h1, h2, hk. Suppose we pass two strings through these hash functions and compute their hashes. If all the hashes are identical for both the strings, there is a very high probability that both the strings are the same. On the other hand, even if one hash does not match, we can definitely say that the strings are different.

Bloom filter is essentially an array that stores Os and 1s. In the image below, each cell represents a bit. The number below the bit is its index for a bloom filter of size 10.



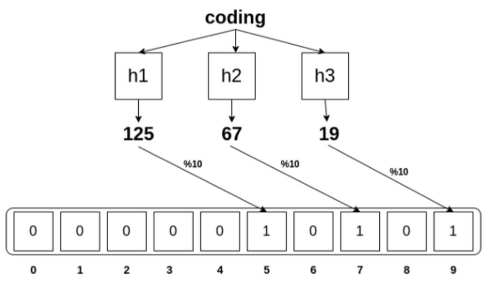
Empty Bloom Filter

7. Inserting Element In Bloom Filter

In order to add an element, we need to pass it through the k hash functions. Bits are set at the index of the hashes in the array. For example, we want to add the word "coding". After passing it through three hash functions, we get the following results.

- h1("coding") = 125
- h2("coding") = 67
- h3("coding") = 19

Suppose that the size of our bloom filter is m = 10. We need to take mod of 10 for each of these values so that the index is within the bounds of the bloom filter. Therefore, the indexes at 125%10 = 5, 67%10 = 7 and 19% 10 = 9 have to be set to 1.



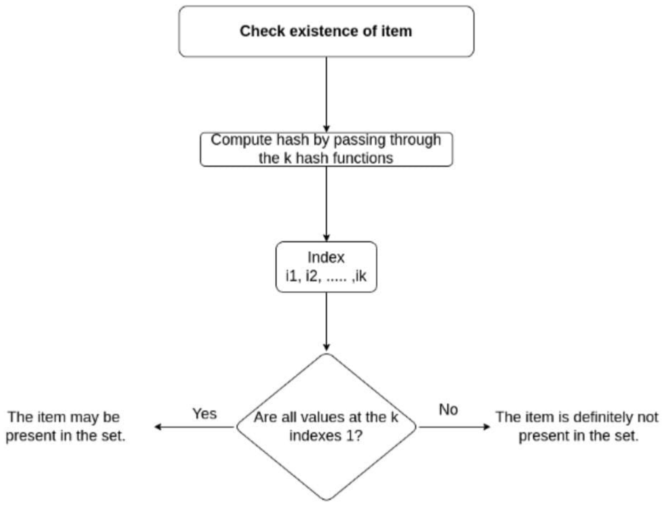
Inserting in bloom filter

Testing membership of an item in Bloom filter

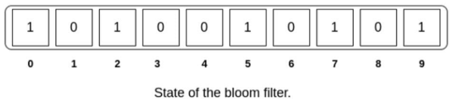
If we want to test the membership of an element, we need to pass it through the same hash functions. If the bits are already set for all these indexes, then this element might exist in the set. However, even if one index is not set, we are sure that this element is not present in the set.

Let's say we want to check the membership of "cat" in our set. Furthermore, we have already added two elements, "coding" and "music", to our set.

- Coding has the hash output (125, 67, 19) from the three hash functions, and as discussed above, the indexes (5, 7, 9) are set to 1.
- Music has the hash output (290, 145, 2) and the indexes (0, 2, 5) are set to


Check existence of item
Compute hash by passing through the k hash functions
Index i1, i2, ... ,ik
Are all values at the k indexes 1?
Yes
The item may be present in the set.
No
The item is definitely not present in the set.

(a)



(b)

We pass "cat" through the same hash functions and get the following results.

- h1("cat") = 233
- h2("cat") = 155
- h3("cat") = 9

So we check if the indexes 3, 5, 9 are all set to 1. As we can see, even though indexes 5 and 9 are set to 1, 3 is not. Thus we can conclude with 100% certainty that "cat" is not present in the set.
Now let's say we want to check the existence of "gaming" in our set. We pass it through the same hash functions and get the following results.

- h1("gaming") = 235
- h2("gaming") = 60
- h3("gaming") = 22

We check if the indexes (0, 2, 5) are all set to 1. We can see that all of these indexes are set to 1. However, we know that "gaming" is not present in the set. So this is a false positive.
Can we predict the probability of these false positives? Yes, and it is explained below.
Let n = number of elements, m = length of the bloom filter, k = number of hash functions.
Assume that the hash function selects each index with equal probability. The probability that a particular bit is not set to 1 by a specific hash function during the insertion of an element is 1-(1/m). If we pass this element through k hash functions, the probability that the bit is not set to 1 by any of the hash functions is (1 -(1/m))∧k.
After inserting inserted n elements, the probability that a particular bit is still 0 is (1 -(1/m))∧(kn). Therefore the probability that it is 1 is 1-(1 (1/m))∧(kn).
We want to test the membership of an element in the set.
Each of the k array positions computed by the hash functions is 1 with a probability 1 -(1- (1/m))∧(kn). The probability of all of them being 1, which would result in a false positive, is (1 -(1 (1/m))∧ k). This is also known as the error rate. As we can see, if m tends to infinity, the error rate tends to zero. Another way to avoid a high false-positive rate is to use multiple hash functions.

8. Application

Bloomfilter

1. Weak password detaction
2. Internet Cache Protocol
3. Safe browsing in Google Chrome
4. Wallet synchronization in Bitcoin
5. Hash based IP Traceback
6. Cyber security like virus scanning

9. Conclusions

Zip Tree
We introduce the zip tree a form of randomized binary search tree that integrates previous ideas into one practical, performant, and pleasant-to-implement package.We perform insertions and deletions by unmerging and merging paths (unzipping and zipping) rather than by doing rotations, which avoids some pointer changes and improves efficiency. The methods of zipping and unzipping take inspiration from previous top-down approaches to insertion and deletion

From a theoretical standpoint, this work provides two main results. First, zip trees require only O (log log n) bits (with high probability) to represent the largest rank in an n-node binary search tree; previous data structures require O (log n) bits for the largest rank. Second, zip trees are naturally isomorphic to skip lists, and simplify Dean and Jones' mapping between skip lists
Bloom Filter
A Bloom filter is a data structure designed to tell rapidly and memory efficiently whether an element is present in a set. The tradeoff is that it is probabilistic; it can result in False positives. Nevertheless, it can definitely tell if an element is not present. Bloom filters are space-efficient; they take up O(1)space, regardless of the number of items inserted as the bloom filter size remains the same. However, their accuracy decreases as more elements are added. Insert, and lookup operations are O(1) time.

10. Bibliography and citations

Acknowledgements

We wish to thank our instructor, Dr Anil Shukla and our Teaching Assistant Vinay Sir for their guidance and valuable inputs provided during the project.

References

[1] Bernhard Haeupler ·Siddhartha Sen· and Robert Tarjan. Rank-balanced trees. *ACM Transactions on Algorithms*, 11:1–26, 4 2015.

[2] Herman Chernof. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23:493–507, 4 1993.

[3] Brian Dean and Zachary Jones. Exploring the duality between skip lists and binary search trees. *In ACM Southeast Regional Conference (ACMSE)*, page 395–400, 2007.

[4] Robert Tarjan· Caleb Levy· and Stephen Timmel. Zip trees. *In Workshop on Algorithms and Data Structures (WADS).*, page 566–577, 2019.

[5] Conrado Martínez and Salvador Roura. Randomized binary search trees. *ACM*, 45:288–323, 2 1998.

[6] C. J. Stephenson. A method for constructing binary search trees by making insertions at the root. *International Journal of Computer & Information Sciences*, 9:15–29, 1 1980.

[7] Stephen Timmel. Zip trees: A new approach to concurrent binary search trees. *The Annals of Mathematical Statistics*, page 493–507, 2017.

Bloomfilter

1. https://en.wikipedia.org/wiki/Bloom_filter
2. https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff
3. https://www.quora.com/What-are-the-best-applications-of-Bloom-filters