X-Fast and Y-Fast Tries

Christopher Liao, Yunze Lian

Introduction

In algorithms and data structures, it is a common theme to invent specialized algorithms that can in theory outperform the optimal algorithm, given some assumptions about the data. For example, in class we learned sorting algorithms which have linear run time on integers of a certain size. *X-Fast* and *Y-Fast Tries* apply the same principle to *ordered associative arrays*.

An *ordered associative array* is defined as a data structure which supports the following operations:

- *Insert*
- *Delete*
- *Successor*: The smallest number in the array that is larger than the input.
- *Predecessor*: The largest number in the array that is smaller than the input.
- *Lookup*: Is the input in the array?

A balanced Binary Search Tree (BST) is an example of such a data structure that performs all of the above operations in logarithmic time. However, when the data is assumed to be integers of a certain size, we can speed up these operations to *log(w)*, where *w* is the number of bits in the integer. An example application where this type of data structure might be useful is storing data by timestamp. Consider a system which logs events by timestamp, which is a 64 bit integer. A user wants to quickly retrieve events surrounding a certain timestamp. A Y-fast Trie may be useful for this application.

This report is organized as follows. We first explain how the X-fast trie works. We then explain how the Y-fast trie improves the X-fast trie. We conclude with some results from our implementation of X-fast trie, Y-fast trie, and BST.

Notation

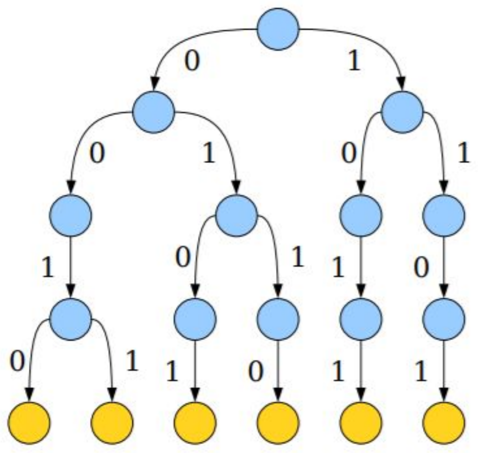
We shall use the following notation throughout the report:

M	Largest integer in the data (constant in practice)
N	Number of elements in the data structure
w	Number of bits in the integer

Note that *w = log(M)*. We always use base 2 log.

X-fast trie

A trie is a *prefix tree*, commonly used to store words in a dictionary. If we consider integers as just strings of bits, we can store integers in a *bitwise trie.* Figure 1 illustrates a bitwise trie for 4-bit integers.



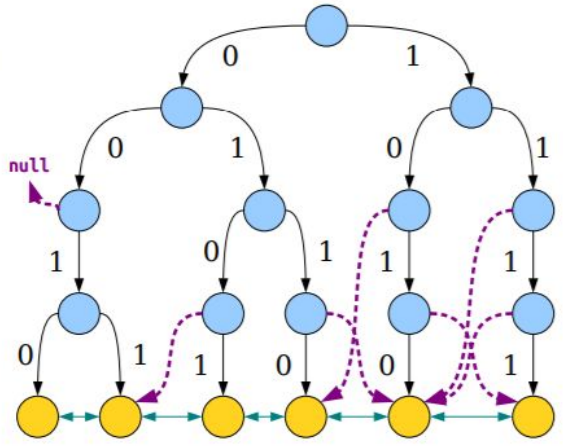


Figure 1 (Left). An example of a naive bitwise trie storing 6 4-bit binary integers (10, 101, 110, 1011, and 1101). Figure from [1]
Figure 2 (Right). An example of an X-Fast Trie with *w*=4. The green arrows doubly link the data to speed-up in-order walk. The purple arrows link internal nodes to either successor or predecessor leaf node. These purple arrows speed-up successor and predecessor operations when the input is not a leaf node. In addition, each level's prefixes are stored in a hash table. In this example, there are four levels. Starting from the level below the root (level 1), the hash tables store the following information. Level 1: [0, 1]. Level 2: [00, 01, 10, 11]. Level 3: [001, 010, 011, 101, 111]. Level 4: [0010, 0101, 0110, 1010, 1111]. Figure from [1]
Important: Level 4 (leaf nodes) is the only level with data. The other levels store auxiliary information.

A naive bitwise trie (Figure 1) is not a useful data structure per se. Searching through this data structure takes *O(w) = O(logM)*, which is worse than BST's *O(logN)*. To make this structure useful, we have to add three more pieces of information (Illustrated in Figure 2):

1. Doubly link the leaves (where the data is stored). This makes it easy to iterate through the data in order. (Green arrows in Figure 2).
2. Draw "threaded pointers" (Purple arrows in Figure 2). If an *internal node* is missing the "0" pointer, instead of setting it to null, use it to point to the *leaf node* immediately to the left. If an internal node is missing the "1" pointer, use it to point to the leaf node immediately to the right. This makes the predecessor and successor operations faster. (If this is not clear, it will hopefully become clearer in the next section).
3. Store the *prefixes* at each level of the trie in a hash table. The hash table should have constant lookup time.

X-fast Trie Insert

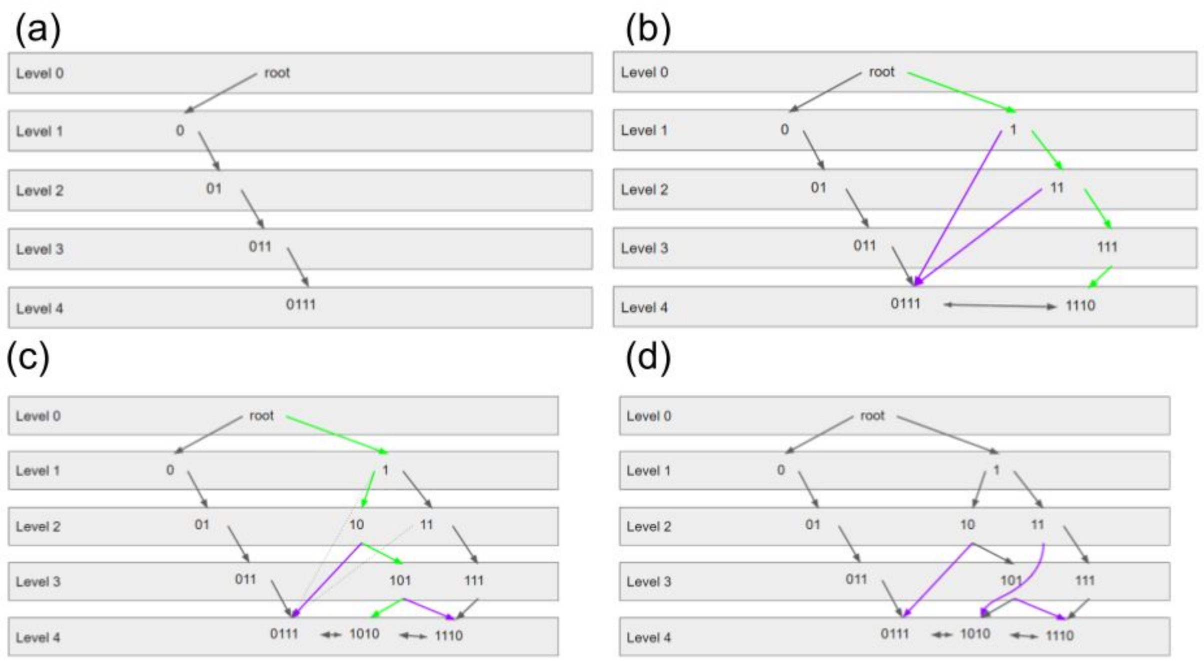


Figure 3. Starting from empty trie. **(a)** Insert 7 (0111). Pointers going to the left are "0" pointers"; pointers going to the right are "1" pointers. **(b)** Insert 14 (1110). **(c)** Insert 10 (1010). **(d)** Final X-fast Trie. Note that the grey boxes at each level indicate the prefixes stored in the hash table at each level.

Our implementation of the X-fast Trie Insert(x) is as follows:

1. Find p = Predecessor(x).
2. The successor of x is just s = p->right.
3. Put x between p and s. (p <-> x <-> s)
4. Insert x into the hash table at leaf level.
5. From root down to x, create any missing internal nodes (the prefix of any new node needs to be added to the appropriate hash table) and set pointers appropriately.
6. Let n_p be the lowest common ancestor of p and x.
7. From n_p down to p, any purple pointer that used to point to s should now point to x. (successor pointer update)
8. Let n_s be the lowest common ancestor of p and s.
9. From n_s down to s, any purple pointer that used to point to p should now point to x. (predecessor pointer update).

Steps 1-4 take *O(1)* time. The rest of the steps take *O(logM)*. Total time is *O(logM)*. Note that the constant in front of the logM is quite large. We have to walk the height of the trie three times in total (to p, x, and s).

We do not implement the delete operation. The delete operation takes the same amount of time and is just as messy to implement.

X-fast Trie Predecessor

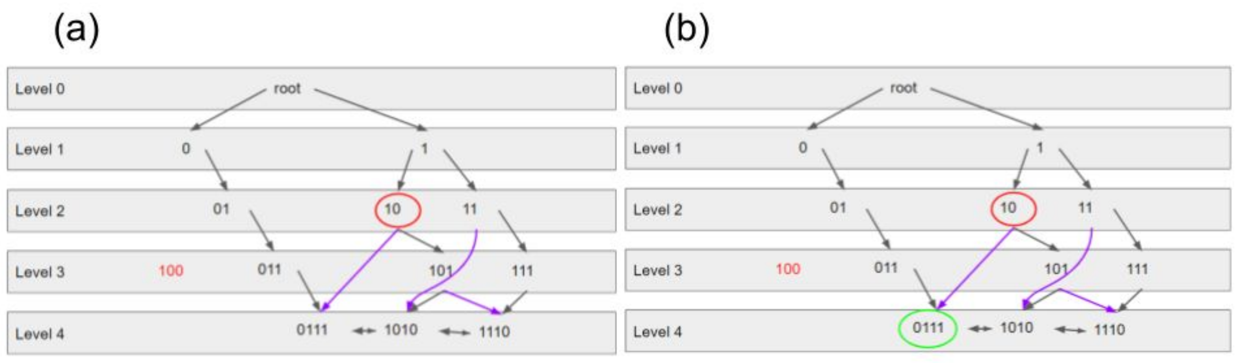


Figure 3. Illustration of predecessor operation on X-fast trie. The trie stores three numbers 7(0111), 10(1010), and 14(1110). We want to find predecessor of 9 (1001). Perform bisection search across the levels to find the longest prefix of 9 that is in the trie. **(a)** Start at level 2. The 2-bit prefix of 9 (1001) is "10". This is in the hash table at level 2. Now we look between levels 2 and 4. **(b)** Look at level 3. The 3-bit prefix of 9 is 100. This is not in the hash table at level 3. We conclude that the *longest prefix of 9 that is part of the trie* is 10, so we follow the predecessor pointer (purple) from node "10" to "0111". This is the answer, 7 is the predecessor of 9.

Figure 3 illustrates the predecessor operation on an X-fast trie. The steps are as follows:

1. Perform bisection search across the levels in the X-fast trie to find the *longest prefix of the input that is part of the trie*.
2. Follow the predecessor pointer from the node found in step 1. This yields the answer.

Step 1 performs bisection search across *O(logM)* levels. Each search is constant, because it is just a hash table lookup. Therefore, the total runtime is *O(loglogM)*. The successor operation uses the same idea and runs in the same amount of time.

X-fast Trie Lookup

X-fast trie lookup takes constant time. Simply determine if the input integer is part of the hash table at the leaf level.

Y-Fast Trie

The X-Fast Trie speeds up successor and predecessor operations, but the insert and delete operations are slow, because every insert and delete operation must iterate through the height of the trie multiple times. We can improve the insertion and deletion operations by using a two-tier data structure called the Y-fast Trie.

The Y-fast Trie uses an X-Fast Trie and multiple BSTs. Data is grouped into small BSTs; *the maximum value in each BST is organized in an X-Fast Trie*. The size of each BST is *O(logM)*, so there is a total of *O(N/logM)* BSTs. Deletion and insertion into *each individual BST* is inexpensive, but deletion and insertion into the *X-fast trie* is expensive. If we allow the size of the BSTs to fluctuate, most of the deletions and insertions happen at the BST level without needing to modify the X-fast Trie.

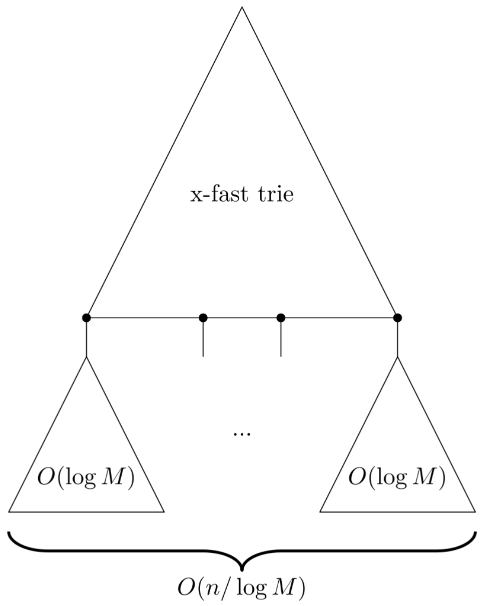


Figure 4. High level illustration of Y-fast trie. Figure from [5]

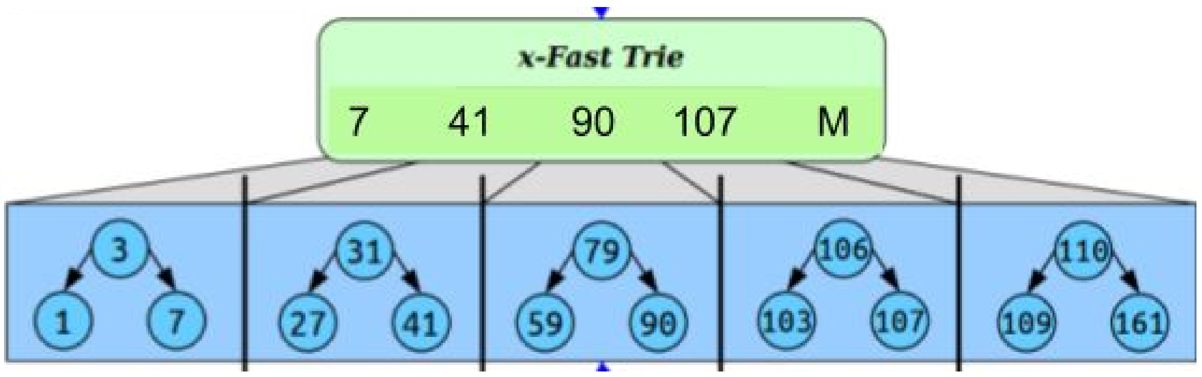


Figure 5. Example of Y-fast trie. All data is stored in BSTs. The data in the X-fast trie is auxiliary information. We label each BST using the maximum value in the BST. For example, the label of the left-most BST is 7, so 7 is stored in the X-fast Trie. (Except for the right most BST containing the largest integers; this BST is labeled with the maximum possible integer M). Figure from [1]. *We allow the size of BSTs to vary between logM / 2 and logM * 2.*

Y-Fast Trie Insertion

To insert x:

1. Find the successor of x in the x-Fast Trie (call this r). Let the BST corresponding to r be called T.
2. Insert x into T.
3. If size(T) >= logM*2, we need to split T into two trees. The first tree will consist of the smaller half of T. We add the maximum value of this new tree to the X-fast trie.

Amortized runtime analysis:

Finding the successor of x in the X-fast trie takes *O(loglogM)*. Inserting x into T takes *O(loglogM)* also, because each BST has *logM* elements and height *loglogM.* These first two steps happen for every insertion. Step 3 takes *O(logM)* time, because splitting the BST (size logM) requires logM time and inserting into X-fast trie requires logM time. Step 3 only happens every logM insertions (If this is not clear, consider a BST of logM elements; we add logM elements; now it has 2*logM elements; so we split, and end up with 2 BSTs with logM elements; we can then add up to logM elements to each of these BSTs before having to split again).

If we perform N insertions, every insertion takes loglogM time, but N/logM of them take additional logM time. The average time per insertion is then:

(N/logM * logM + N * loglogM) / N = 1 + loglogM

Therefore, the average insertion takes *O(loglogM).*

We can use the same type of analysis for deletion.

Y-Fast Trie Successor and Predecessor
The successor operation is straight forward. Simply find successor in X-fast Trie and then find successor in the corresponding BST. Both these operations take *O(loglogM)* time.

Y-Fast Trie Lookup
To lookup a value, find the successor in the X-fast Trie, then lookup in the corresponding BST. Lookup takes *O(loglogM)* time

Complexity Comparison

	BST	X-fast Tries	Y-fast Tries
Space	O(N)	O(N log M)	O(N)
Lookup	O(log N)	O(1)	O(log log M)
Insert/Delete	O(log N)	O(log M)	O(log log M)
Successor/Predecessor	O(log N)	O(log log M)	O(log log M)

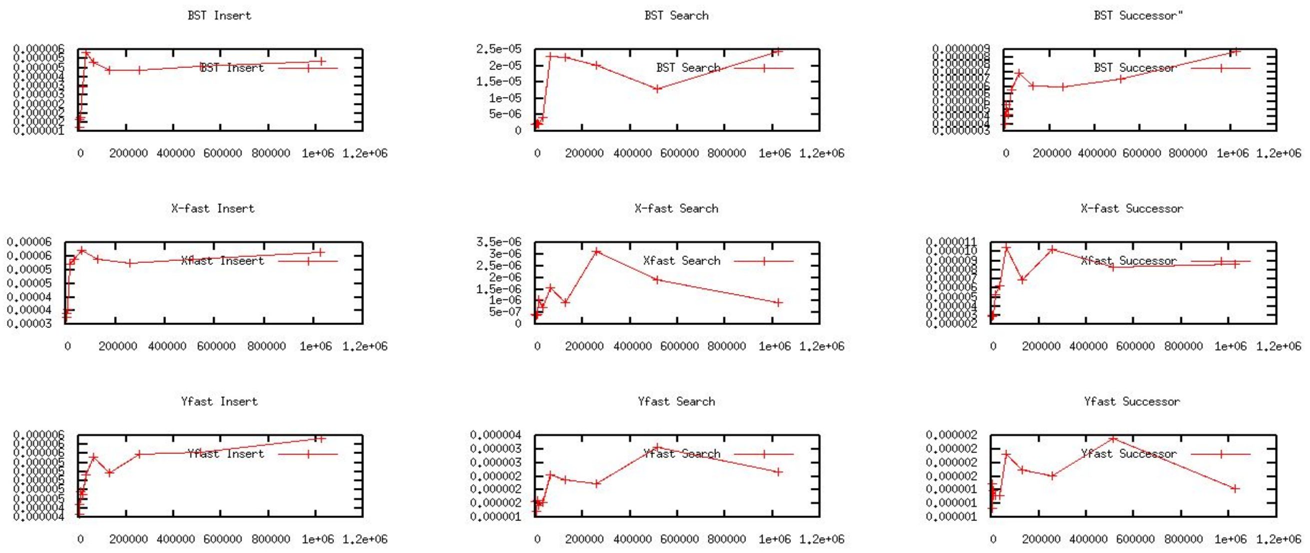


Figure 6. Comparison of run time for insert, successor, and lookup operations between the three data structures. The x axis is the number of integers in the data structure N. The y-axis is the execution time per operation in seconds. *Note: M=64, it is constant.* Therefore, we expect the time of the X-fast and Y-fast trie operations to be *constant* with respect to N. We expect BST operations to look like *log(N)*. In reality, it is hard to show this empirically. We found that despite X-fast and Y-fast operations to be constant w.r.t. N in theory, in practice they *increase* w.r.t. N, likely due to increasing expensive memory operations.

Implementation

We implement the insertion, successor and lookup operations for each of these data structures. We then plot the time against each other.

We use *std::map* for the balanced BST implementation. We use *std::unordered_map* for hash table implementation.

We found that our implementation of X-fast and Y-fast tries have much worse performance for each of these operations when compared to the BST. This is likely because large time constants and inefficient memory operations. We expect the performance could be improved by optimizing for memory access.

We do not implement the deletion operation, but it should take the same time as insertion. Note that we could tune the parameter which determines how much flexibility we allow in the size of child BSTs in a Y-fast Trie. Giving more flexibility makes the Y-fast Trie look more like a BST, and giving less flexibility makes it look more like an X-fast Trie.

Work Division

Christopher Implemented X-fast and Y-fast Tries and wrote the description of these algorithms. Yunze performed run time analysis, including the code to time the insertion and successor operations and plotting the graphs.

References

[1] http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/15/Slides15.pdf
[2] https://iq.opengenus.org/y-fast-trie/
[3] https://iq.opengenus.org/y-fast-trie/
[4] https://en.wikipedia.org/wiki/X-fast_trie
[5] https://en.wikipedia.org/wiki/Y-fast_trie