X-Fast Trie

November 7, 2022

Abhishek Jaiswal (2021CSB1061) ,
Akanksh Caimi (2021CSB1064) ,
Devanshu Dhawan (2021CSB1082)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Ms. Akanksha

Summary: In this project we implement a data-structure known as X-Fast Trie.It brings down the search time of BST from O(logN) to O(1) and also the search time of predecessor and successor from O(logN) to O(log(logN)).

---

1. Introduction

1.1. What is a Trie?

A trie is a prefix tree for storing strings. Here, we will store integers in the form of bitwise strings.

1.2. X-Fast Trie

X-fast trie is a data structure used to store integers from a bounded domain. It is a bitwise trie, i.e. a binary tree where each subtree stores values having binary representations with common prefix. Each level is stored as an array of Hash Tables. Further, descendent pointers are used to connect the upper levels to the final level, elements of which are weaved together in a doubly linked list to find successor without travelling back up.

2. Operations

X-fast tries support the operations of an ordered associative array. This includes the usual associative array operations, along with two more order operations, Successor and Predecessor: [2, 3]

2.1. Find(k)

Searching the value associated with a key k that is in the data structure can be done in constant time by looking up k in LSS[0], which is a hash table on all the leaves.

2.2. Successor and Predecessor

To find the successor or predecessor of a key k, we first find A_k, the lowest ancestor of k. This is the node in the trie that has the longest common prefix with k. To find A_k, we perform a binary search on the levels. We start at level h/2, where h is the height of the trie. On each level, we query the corresponding hash table in the level-search structure with the prefix of k of the right length. If a node with that prefix does not exist, we know that A_k must be at a higher level and we restrict our search to those. If a node with that prefix does exist, A_k can not be at a higher level, so we restrict our search to the current and lower levels.

Once we find the lowest ancestor of k, we know that it has leaves in one of its subtrees (otherwise it wouldn't be in the trie) and k should be in the other subtree. Therefore the descendant pointer points to the successor or the predecessor of k. Depending on which one we are looking for, we might have to take one step in the linked list to the next or previous leaf.

Since the trie has height O(log U), the binary search for the lowest ancestor takes O(log log U) time. After that, the successor or predecessor can be found in constant time, so the total query time is O(log log U).

2.3. Insert

To insert a key-value pair (k, v), we first find the predecessor and successor of k. Then we create a new leaf for k, insert it in the linked list of leaves between the successor and predecessor, and give it a pointer to v. Next, we walk from the root to the new leaf, creating the necessary nodes on the way down, inserting them into the respective hash tables and updating descendant pointers where necessary.
Since we have to walk down the entire height of the trie, this process takes O(log U) time.

2.4. Delete

To delete a key k, we find its leaf using the hash table on the leaves. We remove it from the linked list, but remember which were the successor and predecessor. Then we walk from the leaf to the root of the trie, removing all nodes whose subtree only contained k and updating the descendant pointers where necessary. Descendant pointers that used to point to k will now point to either the successor or predecessor of k, depending on which subtree is missing.
Like insertion, this takes O(logU) time, as we have to walk through every level of the trie.

3. Figures, Tables and Algorithms

3.1. Figures

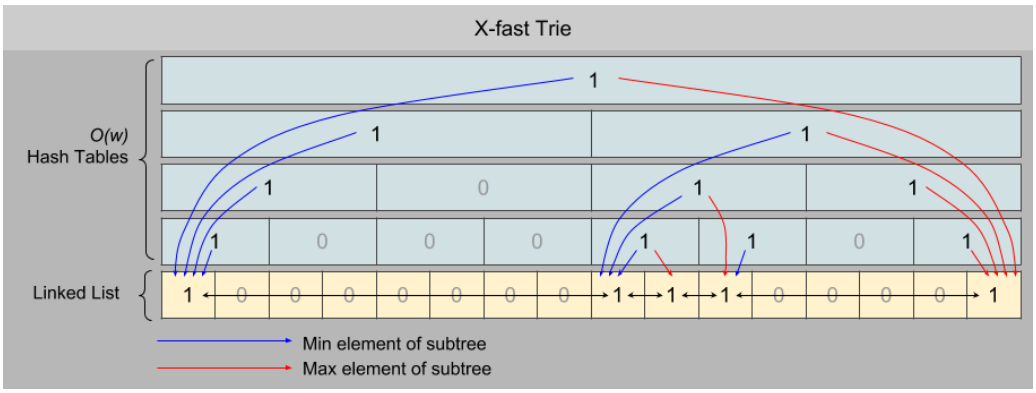

Figure 1: Basic Structure of X-Fast Trie

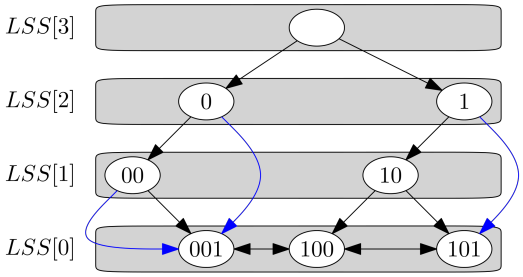

Figure 2: Hash Map used in X-Fast Trie

3.2. Algorithms

Pseudo-algorithms for operations discussed above are described below:

Algorithm 1 Find(k)

Since the leaf nodes are saved as a hash table, this can be done directly in O(1) time.

```
1: Find(XFast,k)
2: if XFast[W].find(k)! = XFast[W].end() then
3:    return XFast[W][k]
4: end if
5: return nullpointer
```

Algorithm 2 Find_Successor(k),Find_Predecesssor(k)

1. Binary search the levels, each of which is a hash table.
2. Set low = 0 and high = h, where h = height of the tree.
3. Repeatedly set mid = (low + high) / 2, and check if hash table has contains the prefix of k. If it does, that means target node is between levels mid and high(inclusive). So set low = mid. Otherwise, target node is between low and mid, so set high = mid.
4. The node reached will have longest common prefix with k. Since k is not in trie, either left or right child pointer is empty. So descendant pointer is used to directly reach the leaf node.
5. After reaching leaf node, double linked list needs to be traversed forward or backward just once to get wanted node, or to determine that no such element exists.

Algorithm 3 Insert(k,v)

1. Find the predecessor and successor of k.
2. Create a new node representing k. Have the node store value v.
3. Add the node between predecessor and successor in doubly linked list.
4. Start and root and traverse down to the leaf while creating missing internal nodes.

Algorithm 4 Delete(k)

1. Find the leaf node with key k using hash table.
2. Delete the node and connect its successor and predecessor in doubly-linked lists.
3. Traverse upwards till root and delete all internal nodes that had only k inside its subtrees. Update descendant pointers to point to either its successor or predecessor appropriately.

4. Comparison

4.1. Comparison with Binary Search Tree (BST)

Below is the comparison of X-Fast Trie with Binary Search tree (BST) using different parameters.

	X-Fast Trie	Binary Search Tree
Search	O(1)	O(log N)
Space	O(NlogU)	O(N)
Predecessor	O(log logU)	O(logN)
Successor	O(log logU)	O(logN)
Insert	O(logU)	O(logN)
Delete	O(logU)	O(logN)

All the above time complexities are in amortised time.

4.2. Comparison with Van-Emde Boas Tree and Y-Fast Trie

[1] Below is the comparison of X-Fast Trie with Van-Emde Boas Tree and Y-Fast Trie using different parameters.

	X-Fast Trie	Van-Emde Boas Tree [4]	Y-Fast Trie [5]
Search	O(1)	O(log logU)	O(log logU)
Space	O(NlogU)	O(U)	O(N)
Predecessor	O(log logU)	O(log logU)	O(log logU)
Successor	O(log logU)	O(log logU)	O(log logU)
Insert	O(logU)	O(log logU)	O(log logU)
Delete	O(logU)	O(log logU)	O(log logU)

5. Applications of X-Fast Trie

Comparing search time asymptotics for X-fast tries and search trees, we see that X-fast shines when log[N] grows faster than log logU, which comes down to the criteria of N » logU.

The following are practical examples of X-fast trie:

1.Consider you are developing a flight searching website. For the given date-time of expected departure we return the list of all the upcoming flights. Flights are updated and changed frequently. Date-time minutes can make up all the possible 'U' values, and the flights to a specific direction make up n set.

2. Internet router needs to redirect the IP packets to other routers with the IP closest to the requested one.

3.Trackless bit-Torrent peer-to-peer networks look up content by hash and the nodes with IDs closest to this hash (by some metric) are assigned with tracking that content.

X-fast trie can get useful to lookup content in huge networks with large number of nodes (so that n » logU).

6. Conclusions

X-Fast trie assists in handling heavy data efficiently. It is implemented using a trie. Integers are stored in a hash map and the bits in the bottom level are connected using a linked list.

Basic Operations like Successor and Predecessor have time complexity as low as O(log log U). This is used in search engines for finding the nearest results with lightning speeds.

When X-Fast trie is put against other data structures (BST, VEBT, Y-Fast Trie), it is observed that X-Fast Trie provides results with minimal spaces and time complexity.

7. Bibliography and citations

Acknowledgements

We appreciate Dr. Anil Shukla for his guidance and supervision which has provided a lot of resources needed to complete our project. We want to thank Ms. Akanksha who helped us with this project, without her support and suggestions it would not have been possible.
Our friends and families constantly encouraged us throughout the process when we felt discouraged or became frustrated because they knew how much work went into this venture hence we extend our gratitude to them too!

References

[1] Erik Demaine. Integer: models, predecessor problem, van Emde Boas, x-fast and y-fast trees, indirection.

[2] Standford University. x-fast trie basic operations.

[3] Wikipedia. Operatins of X-Fast Trie.

[4] Wikipedia. Van-emde boas tree.

[5] Wikipedia. Y-FAST TRIE.