a

Fusion tree

November 5, 2023

Kartikey sahu "2022csb1087" ,
Prashant kumar "2022csb1202" ,
Pranav bhole "2022csb1103"

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Saumya Sarkar

Summary: A Fusion Trees is a static Data Structure that allows for predecessor queries in constant time on a set of constant size of numbers, but that requires significant large word sizes to be implemented in a computer that follows the standard word ram model. In this work, we provide the C++ code and reference to our implementation of fusion trees that performs predecessor queries with a constant number of calls .Our Fusion Tree is restriced to word size so we are implementing only for w=32 . as we are implementing in c++ if one can implement in python or similar programming language it can go upto a certain significant size but beyond that its depend on machine

1. Introduction

The concept of processing predecessor queries in constant time within a set of limited maximum size was initially introduced by Ajtai et al. in 1984. This idea later influenced the development of the Fusion Tree by Fredman & Willard in 1993. A Fusion Tree is a static data structure designed to provide constant-time answers to predecessor queries, but it's constrained to have a maximum size of O(wî/5) elements. Here, 'w' represents the word size of the underlying computer in the standard word RAM model . Given its size limitations, Fusion Trees are often used as nodes within a B-tree. When incorporated into a B-tree with a branching factor of wî/5 and n elements, Fusion Trees can significantly speed up predecessor query responses to O(logn/logw), reducing query times by a factor of log w.
In our work, we present a C++ implementation of the Fusion Tree that leverages a standard integer in C++.. For demonstration purposes, we offer a slower implementation of the integer using a standard C++ datatype, ensuring the correctness of our Fusion Tree's operation. However, one can designed the big integer class as a template or implement in python or other programming language where barrier of word size is somewhat reduce but after some point it goona restriced too , So it can be readily replaced with a faster implementation if future computer architectures allow for it. To facilitate the Fusion Tree's use as a B-tree node, we introduce an FusionTree class that stores essential constants defining the Fusion Tree's behavior. This includes user-provided initialization constants like the assumed word size of the computational environment and the maximum capacity of Fusion Trees. Additionally, the FusionTree class pre-computes crucial constants required for predecessor , Successor queries, w hich depend solely on the initialization constants and can be shared across all Fusion Trees within a B-tree.

2. C++ Reference

In this section of the report, we will provide documentation for our code, which can be found in our repository on https://github.com/pranavbhole123/CS201project. We will explain the public functions and operators needed for implementing fusion tree such as bitmanipulation .

3. A Brief Review of Fusion Trees

3.1. Sketching

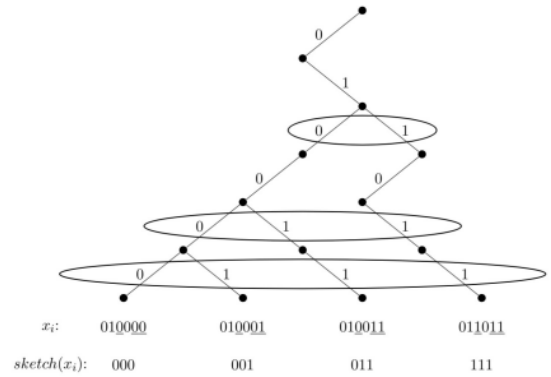

Figure 1: Figure 1 : sketching

We can represent a set of numbers as trie, or prefix tree. In such a tree, each number would be represented by a path down a binary tree in which we would turn left at level i if the i-th bit of the number is 0, and turn right if it is 1. Each level of this tree is associated with the position of a bit position in the numbers, and positions associated with levels of in which there is branching in the tree will be called important bits b0, b1, ..., br1. Then, sketch(x1) will simply be the the number formed by the concatenation of the bits of xi , in order, in positions b0, b1, ..., br1. An illustration of the trie representation of the elements in a fusion tree, their important bits and sketches can be seen in Figure 1.

3.2. Desketching

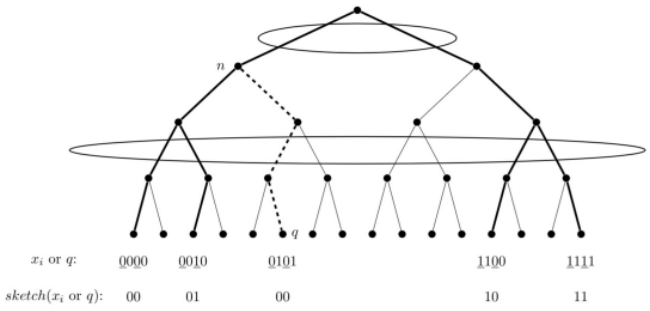

Figure 2: Figure 1 : Desketching

In Desketchifying, we assume we know the sketch neighbors xi and xi+1 of a number q in the fusion tree, and we want to find the predecessor of q in constant time. Let n be lowest common ancestor of both q and xi or q and xi+1 (whichever is higher) in the trie representation of the fusion tree. Then, n represents the highest node in the path defined by q that diverges from the tree. This means that the subtree to which q should belong is empty in the fusion tree, but there is some element in the other subtree. Notice that this node must be the LCA of q and one of its neighbors because either the predecessor of q will be in the non-empty subtree (if q takes a turn to the right) or the successor of q will be in the non-empty subtree (if q takes a turn to the left). An example of this can be seen in Figure 2. Let p be the prefix of q that is above node n and let y be the length of this prefix. If the y-th bit of q is 1 (q branches to the right) and we know its predecessor is in the left subtree of n, then we only need to find the rightmost element of this subtree. Notice that this can be done by finding the sketch predecessor of p011...1, which can be done in constant time with parallel comparison. Analogously, if the y-th bit of q is 0 and we know its successor is in the left subtree of n, we simply have to find the sketch predecessor of p100...0 in the fusion tree. This search will either find the predecessor or successor of q, and we must check in order to return the predecessor of q.

3.3. Parelled companion

Parallel comparison is a crucial technique to a fusion tree. The idea is that if we have an ordered list of numbers x1, x2, ..., xk and want to find the predecessor of q in constant time in that list. Padding all xi with zeros so that they have the same number of bits, we would define the number D1 to be the concatenation of the elements of the list with a 1 before each element, i.e., D1 = 1x11x2...1xk. Then we would define D2 is the concatenation of k repetitions of q with a 0 before each q, i.e., D2 = 0q0q...0q, again padding q with zeros so that it has the same number of bits as the elements of the list. If we calculate the difference D3 = D1 D2, in all the positions in which there was a 1 between some xi , this 1 will only be there in D3 if xi q. Since the xi are ordered, there will be some position j such that the 1's before all xi such that i < j will have become 0's in D3, but will have remained 1's for all i j. If we extract only these bits in the positions placed before the xi , the number of those bits that became 0's in D3 will be the position of the predecessor of q in the list. Therefore, we just have to extract these bits and count the number of 1's.

3.4. General overview

In order to answer a predecessor query in constant time, a fusion tree will keep all the data necessary for this operation in a single word. To fit all its elements in a single word, the fusion tree must keep a smaller representation of them that is able to preserve their order, which can be achieved with an operation called sketching. When we apply sketching to a set of k numbers, we can represent each element xi in just k 1 bits as sketch(xi) and still preserve their relative order, i.e., sketch(xi) < sketch(xj) xi < xj . A perfect sketch would represent the integers in k 1 bits, but we cannot compute a perfect sketch in constant time in a standard word RAM computing model. However, we can compute an approximated sketch, which has the same property of preserving order among elements, but that uses up to k 4 bits. Thus, if we limit the maximum size of a fusion tree to k = O(w pow(1/5)), the representation of the k approximated sketches will take space O(k •k4) = O(w1/5• w4/5) = O(w). Therefore, we can use approximated sketch instead of perfect sketch as our sketch function and all the representations of the elements in the fusion tree will still fit in a single word. When all the elements of a set can be fit in a single word, we can perform predecessor queries in this set in constant time using a technique called parallel comparison. Therefore, given a number q, we can use parallel comparison to find the predecessor of sketch(q) among the sketches of the elements in our set in constant time. However, the sketch predecessor of q, i.e., the largest xi in the set such that sketch(xi) sketch(q) is not necessarily the predecessor of q because sketching only preserves relative order among the elements of the set being sketched. However, if we have the sketch predecessor xi of q, we know that that sketch(xi) sketch(q) < sketch(xi+1), and with just these neighbors, xi and xi+1 we can find the predecessor of q in constant time using a technique called desketchifying . It happens that if we look at the trie representation of the set of numbers, the LCA of either q and xi or q and xi+1 is the root of the subtree that contains either the predecessor or the successor of q. Then, we only have to query these subtrees for either their leftmost or rightmost element, which can be easily done with parallel comparison. To find the LCA in a trie we only have to find the longest common prefix between the elements. In integers, finding the length of longest common prefix between x and y is the same as finding the most significant set bit of x y which can be done in constant time. In the following subsections, we will further detail the implementation of the aforementioned techniques, sketching, desketchifying, approximating sketch, parallel comparison, and most significant set bit.

3.5. Most significant set bit

To find the most significant set bit of an integer x of bit size w, we will first divide its bits in buckets of size w, and find the first bucket with a set bit. We can use a technique similar to what we did in (3.5) to find which buckets have set bits. Suppose w = 4. Let x 0 be x but with zeros in all bits in positions that are multiples of 4. Let F = 10001000...1000. If we calculate the difference t0 = F x0, the only 1's of F that will remain in t0 are those in buckets were there were no set bits in x0(excluding the first bit of each bucket). From t0, it is simple to calculate t, a number that has the first bit of each w bucket set only if x has a bit set in that bucket, we only need to remember to alsocheck the first bit of each bucket of x, since they were setto zero x0. If we can find the first set bit if t, we can find the bucket with the first set bit of x. Let us call the first bit of each bucket of t its important bits. Luckily, since we know the position of these bits, we know there is a bit mask m1 such that t · m is a perfect sketch of the important bits of t. Therefore, we have the bits of t in order and occupying only w bits. We can then use parallel comparison to find the predecessor of t · m among the list of powers of 2 (20, 21, ..., 2w) and this will define the first setbit of t · m, which will also define the first bucket with a set bit in x. To find the first set bit within that bucket, we only have to apply the same parallel comparison with powers of 2 again, now to the entire bucket we have just found.

3.6. Sorting in nlogn

Sorting in o(n lg n) This work detailed how to search a w-bit word in O(lg n /lg lg n) in a fusion tree data structure. It also describes how to sort n elements using B-tree. All elements inserted in a fusion tree result in a sorted set of elements. The paper given in references shows how to transform a static fusion tree in a dynamic one. A dynamic fusion tree is optimized to update keys in O(lg n/ lg lg n + lg(lg(n))) by update. The resulting sort complexity is

$$n\left(\log_B n + \frac{\lg n}{\lg\lg n} + \lg\lg n\right) = O\left(n\frac{\lg n}{\lg\lg n}\right).$$

(a) One logo.

4. ALGORITHM

article algorithm algpseudocode

Algorithm 1 InsertNormal(node, k)

```
    if node is a leaf node then
      i ← node.key_count
3:    while i ≥ 1 and k < node.keys[i − 1] do
        node.keys[i] ← node.keys[i − 1]
        i ← i − 1
6:    end while
      node.keys[i] ← k
      node.key_count ← node.key_count + 1
9:  else
      i ← node.key_count
      while  do
        i ≥ 1 and k < node.keys[i − 1] i ← i − 1
12:   end while
      if node.children[i].key_count == max_keys then
15:     splitChild(node, i)
        if k > node.keys[i] then
          i ← i + 1
18:     end if
      end if
      insertNormal(node.children[i], k)
21: end if
```

Algorithm 2 Insert a Key k into the Fusion Tree

```
Insert k if Root.keyCount == KeysMax then
TempNode ← CreateNewNodeWithKeysMax()
TempNode.isLeaf ← false
TempNode.keyCount ← 0
TempNode.children[0] ← Root
Root ← TempNode
SplitChild(TempNode, 0)
InsertNormal(TempNode, k)
else
InsertNormal(Root, k)
end if
```

Algorithm 4: predecessor(k , node)

```
function predecessor(k, node):
        if node.key_count == 0:
    if node is a leaf node:
        return -1
    else:
        return predecessor(k, node.children[0])
if node.keys[0] > k:
    if node is not a leaf node:
        return predecessor(k, node.children[0])
    else:
        return -1
if node.keys[node.key_count - 1] <= k:
    if node is a leaf node:
        return node.keys[node.key_count - 1]
    else:
        ret = predecessor(k, node.children[node.key_count])
```

Figure 3: Figure 1 : predecessor

.

```
            return max(ret, node.keys[node.key_count - 1])
    pos = parallelComp(node, k)
    if pos >= node.key_count:
        // Handle out of bounds case
    if pos == 0:
        pos += 1
            x = node.keys[pos]
pos = parallelComp(node, temp)
    if pos == 0:
        if node is a leaf node:
            return node.keys[pos]
        res = predecessor(k, node.children[1])
        if res == -1:
            return node.keys[pos]
```

Figure 4: predecessor part 2

.

```
    else:
        return res
if node is a leaf node:
    return node.keys[pos - 1]
else:
    res = predecessor(k, node.children[pos])
    if res == -1:
        return node.keys[pos - 1]
    else:
return res
```

Figure 5: predecessor part 3

.

Algorithm 3: Successor(k,node)

```
function successor(k, node):
    if node.key_count == 0:
        if node is a leaf node:
            return -1
        else:
            return successor(k, node.children[0])
    if node.keys[0] >= k:
        if node is not a leaf node:
            res = successor(k, node.children[0])
            if res == -1:
                return node.keys[0]
            else:
                return min(node.keys[0], res)
        else:
            return node.keys[0]
    if node.keys[node.key_count - 1] < k:
        if node is a leaf node:
            return -1
        else:
            return successor(k, node.children[node.key_count])
    pos = parallelComp(node, k)
    if pos >= node.key_count:
        // Handle out of bounds case
    if pos == 0:
        pos += 1
```

Figure 6: succesor part 1

.

```
x = max(node.keys[pos - 1], node.keys[pos])
common_prefix = 0
i = w
while i >= 0 and ((x & (1 << i)) == (k & (1 << i)):
    common_prefix |= x & (1 << i)
    i -= 1
if i == -1:
    return x
temp = common_prefix | (1 << i)
pos = parallelComp(node, temp)
if node is a leaf node:
    return node.keys[pos]
else:
    res = successor(k, node.children[pos])
    if res == -1:
        return node.keys[pos]
    else:
        return res
```

Figure 7: Succesor part 2

.

5. Some further useful suggestions

Theorems have to be formatted as follows:

Theorem 5.1. *A B-tree with degree B 4 and height h respect: h = O(log n/log B) A sequential search is made to search a key k in a B-tree node. Such search takes O(B) and it is repeated in each B-tree level in the worst case. The result is an O(B log n/log B) overall time to search the key. As B is constant, the complexity is equivalent to O(lg n).*

Theorem 5.2. *Given a compressed trie with S = (s1, . . . , st), the numbers of relevant bit will be at most t1.*

Theorem 5.3. *The bit b = (x, s) is the new relevant bit in the compressed trie with S {x} elements.*

Theorem 5.4. *The most significant bits of x and s are equals. The first bit to diverge is b . Consider the branch between x and s in the trie with S {x}. If x[b] = 1, the x predecessor is the largest element in the b = 0 branch. If x[b] = 0, the x successor is the smallest element in the branch b = 1.*

6. Conclusions

This concludes our report. We have presented a documentation for our code, a brief review of the fusion tree data structure and a more detailed explanation of our implementation of the most important aspects of the fusion tree . This work aimed to describe the fusion tree data structured and the O(n lg n l/g lg n) sorting algorithm. The challenge was to understand many theorems and non trivial concepts and prepare a material for this Project also the word size is big problem while implementing this data structure This work also reveals some pitfalls in the use of lower bounds. For instance, if a generic problem needs at least f(n) operations, the real lower bound is Ω(f(n)/ lg n) because the widely accepted computational models are able to process lg n bit in O(1). An opportune future work would be to implement the fusion tree sorting algorithm and compare it with traditional algorithms. Another relevant aspect is the possibility of multiple operations in O(1) and the removal of irrelevant bits. Such possibilities present theoretical and practical consequences. In the theoretical field, the question is which problems could have their complexity decreased with multiple operations in O(1). In applied computing, the use of multiple operations inside a single word and the removal of irrelevant bits can accelerate traditional algorithms

7. Bibliography and citations

References

[1] Understanding fusion trees.

[2] Author Name. Title of the article. Journal Name, *Year.*

[3] D. Willard. Log-logarithmic worst case range queries are possible in space o(n). Inform. Process. Lett., *pages 81–84, 1983.*

[4] D. Willard. New trie data structures which support very fast search operations. J. Comput. System Sci., *28:379–394, 1984.*

According to [3], log-logarithmic worst case range queries are possible in space O(N) in [4].
Some helpful information can be found on Stack Overflow [1].
Some educational resources for Fusion Trees are available at [2].

Acknowledgements

We want to acknowledge and thank Prof. Erik Demaine for his incredible lectures and materials that allowed for the development of this project , Dr. Anil Shukla and Our Mentor Soumya Sarkar for his guidance . We wish to acknowledge Dr. Anil Shukla for his invaluable contributions and guidance throughout the project. His expertise, encouragement, and dedication to excellence have been instrumental in shaping our work and ensuring its quality. We are deeply grateful to our mentor, Soumya Sarkar, for his unwavering support, patience, and guidance. His mentorship has been pivotal in honing our skills, problem-solving abilities, and overall project management. His belief in our potential and commitment to our growth have made a profound impact. Their collective wisdom, encouragement, and mentorship have enriched our learning experiences and significantly contributed to the success of this project. We are honored to have had the privilege of working with such distinguished individuals, and we look forward to carrying their lessons and inspiration with us into our future endeavors."

A. Appendix

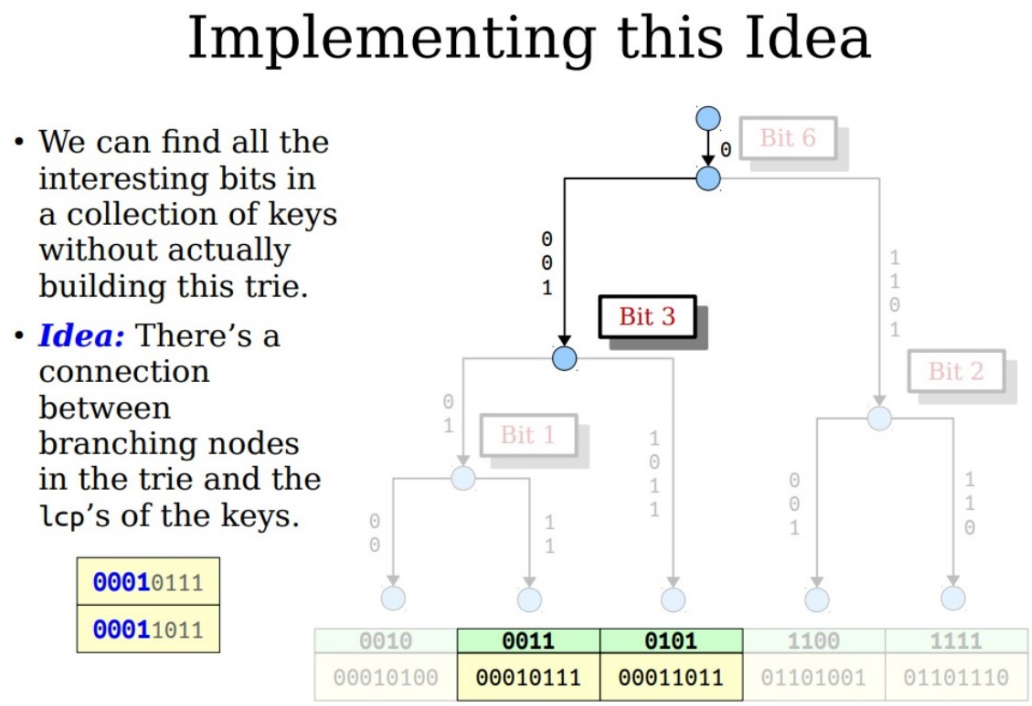

Figure 8: Figure 1 : appendix1

.

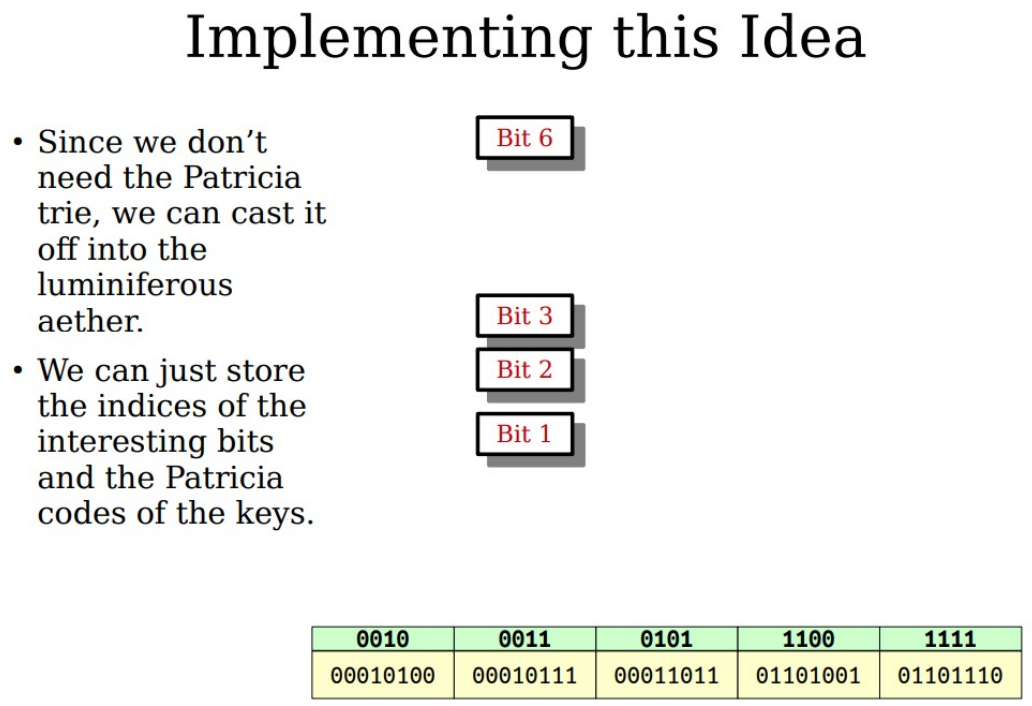

Figure 9: Figure 1 : appendix1

.

Implementing this Idea

- We've assumed up to this point that we can compute Patricia codes in time O(1).
- This is the last step we need to figure out!
- How do we do this?

0010	0011	0101	1100	1111
00010100	00010111	00011011	01101001	01101110

Figure 10: Figure 1 : appendix3

.