The **AIM** of the project was to identify multiple data structures that solve the same problem, understand and learn the data structures and implement them to solve the problem and compare their results with brute-force and naïve algorithms that solve the same problem.

Problem:	Identifying **successor()** in a w-bit integer associate array
Data structures:	Van Emde Boas Tree, Fusion Tree

Literature

A w-bit integer associate array is a 1-dimensional data structure that store integers with bit size w. A naïve example would of such structure would be an array. Since the problem of finding a successor in an array is of time complexity *O(n)* where n being the number of elements in the array. Doing such search operations in large number would result in a very slow performance for some major applications like network routing, caching etc. This created a need for better data structures which could facilitate large number of similar operations in short time. A very well-known and widely used solution to this problem is binary search tree. A binary search tree is a data structure that stores integers in a tree structure adhering to some conditions which reduces the search operation to *O(ln(n))*.

In a binary tree,

- Any node can have only 2 children
- The value of the left child of the node is smaller to its value
- The value of right child of the node is larger to its value

Though this data-structure brought down the time-complexity of the problem exponentially, it still had some limitations.

- Insertion and deletion of data played a crucial role in the performance.
- Worst case time complexity of the algorithm is still *O(n)*

To overcome the above challenges, it was important to adopt balancing of the tree to make sure that the time complexity always stay at *O(ln(n)).* Tree data-structures like AVL trees, Red-Black trees were developed that enforces balancing which gives an average search time complexity of *O(ln(n))*.

However, as there were still scope to improve the performance of successor() or search() operation in associate arrays, there are various data-structures that were devolved, each with an application perspective. The two data-structures that were picked in this scenario were Van Emde Boas Trees and Fusion Trees. Both the data-structures to intelligently store w-bit integers to decrease the cost of search operation on them.

Van Emde Boas Trees

It's a recursive data structure which stores integers and provides a successor() operation with a time complexity of *O(ln(ln(u))* where u is the universe of this integer space, in other words, it is the maximum value of an integer that you can store in data-structure. This data structure is exponentially faster than Binary Search Trees.

Starting from a naïve data-structure, this data structure is a step by step improvement to achieve the time complexity of *O(ln(ln(u))* for insert() and successor().

Consider a vector of size u, which stores the information of the integer being present in the list or not, *1* if the integer is present and *0* if the integer is not.

With the *O(1)* time complexity for insert and *O(u)* for successor(),f let us look at a simple representation of that data structure with universe *u=25,* after inserting values = *[12,17,6,19]*.

0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0

Figure 1: Bit vector of size 25 with values [6,12,17,19]

The idea is to split the universe into multiple clusters, equally so we split it into √u structures each with a size of √u.

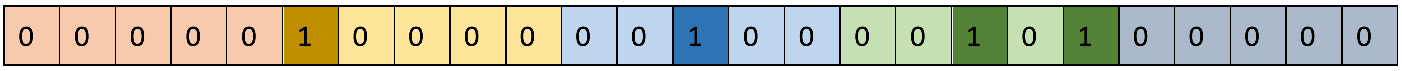

Figure 2: Bit vector of size 25 with values [6,12,17,19] broken down into 5 clusters, each with a size of 5

In Figure 2, we have broken down the vector into √u clusters marking them with different colors. Now, having a supporting vector of is size √u which stores the information if the cluster is empty or not will help us to bring down the search time logarithmically while looking for a successor, let us call that vector 'summary'.

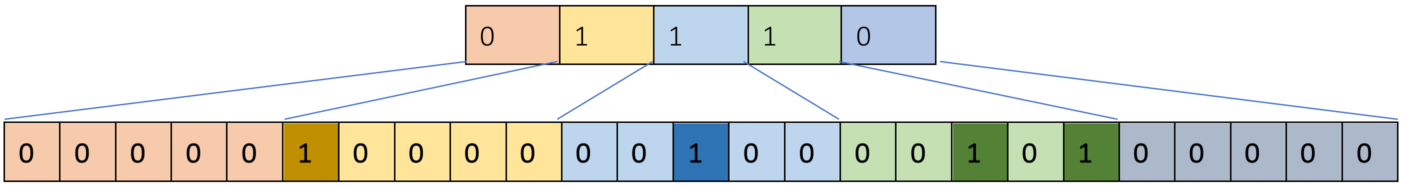

Figure 3: bit-vector of size 25 from figure 2 with its summary bit-vector

To bring down the time complexity of the successor() exponentially again, we store the values of minimum and maximum values of the cluster and skip the overhead of iterating all throughout the cluster once we identify the cluster number to which the query belongs to.

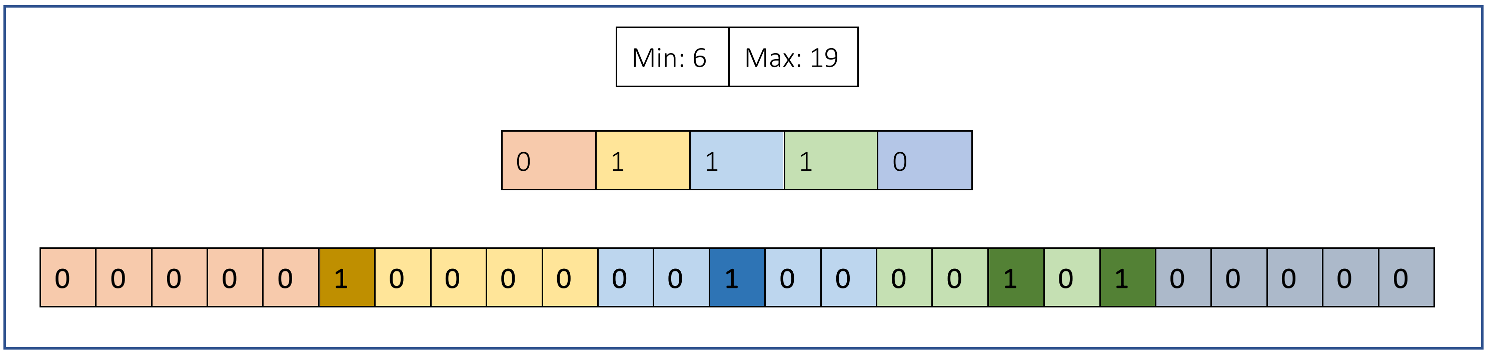

Figure 4: vEB (Van Emde Boas) structure of a single level

Recursively implementing structure in Figure 4 is Van Emde Boas Tree. Figure 5 gives you a better understanding of the data-structure.

My implementation of vEB data-structure:

```
class vEB {
  public:
    int universe;
    int min;
    int max;
    int num_clustures;
    vEB * summary;
    vector<vEB*> clustures;
}
```

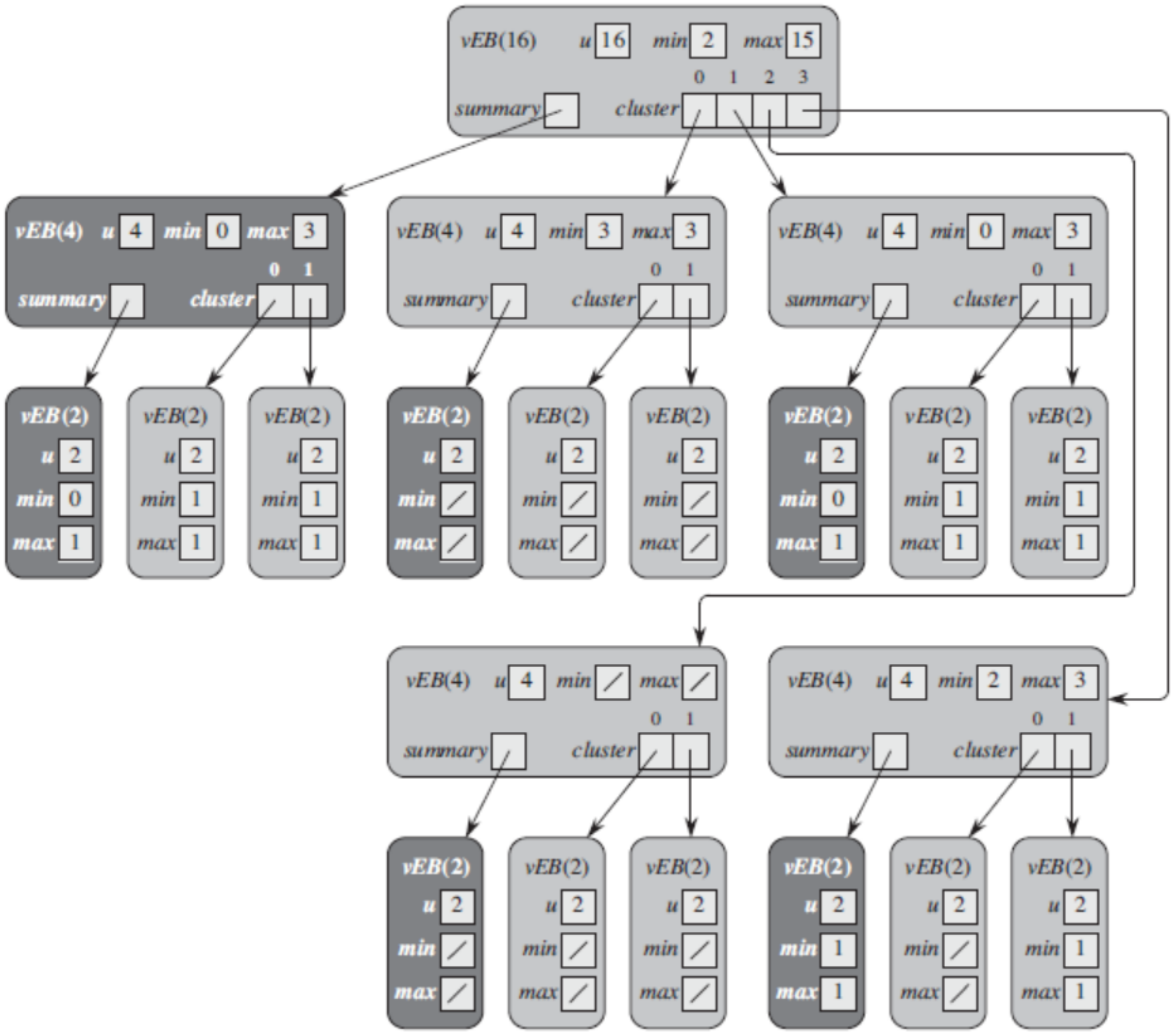

Figure 5: vEB
[https://cs.stackexchange.com/questions/88970/how-to-read-off-the-set-represented-by-a-van-emde-boas-tree]

Representation of a number in vEB

Consider a number ‘k’ in a vEB with universe ‘u’ which belongs to cluster ‘c’ and is at an index of ‘i’ where index starts from 0,1,2,… in that cluster, then we can represent the number as follows.

$$k = c*\sqrt{u} + i$$
$$c = high(k)$$
$$i = low(k)$$

Insert *O(ln(ln(u))*

The general idea of the insert is to not actually insert the value into the bit vector, but modify the summary vector, min and max value such that a successor() query can be addressed.

Algorithm

- Recursively find the cluster the element belongs to and change the bit of that cluster to 1
- Compare the element with it min and max and update them accordingly if they are non-empty and populate them if they are empty.
- If the element is smaller than the culture, update the min and insert the previous min of the cluster

Successor *O(ln(ln(u))*

The idea behind the successor is to find the cluster to which element belongs to recursively and return the max of the structure if the element is smaller or min of the next cluster if the element is larger

Algorithm

- Return min if min is greater than the element
- Return -1 if the element is greater than max
- Find the cluster that the element belongs to recursively and return the max of the cluster if the element is smaller than max
- If the element is greater than max, return the min of the next cluster.

Before we move on to fusion trees, let’s try and understand B-Trees.

B-Tree

It is a self-balancing tree data structure with a branching factor of w, that means every node inside that tree can have a maximum of up to ‘w‘ children and a maximum of up to ‘w-1’ values. There are certain conditions that B-Tree follow,

- All the values in every node are sorted in ascending order
- Keeps the height of the tree balanced using recursive algorithm
- Node with ‘v’ values always has ‘v+1’ children if it is not a leaf
- All leaves are in the same level

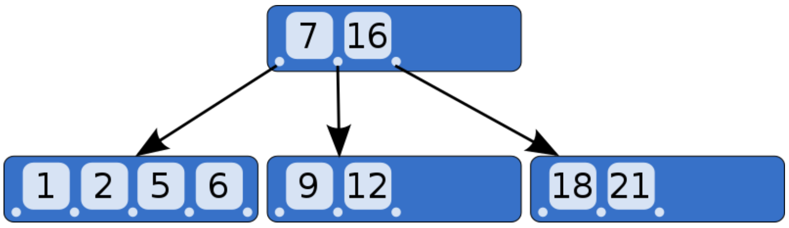

Figure 6: B-Tree with branching factor of 5

Insert

An insert in a B-Tree follows the idea of finding the right leaf recursively to insert the element, and insert into the leaf the element at its right index with the sort condition in place, and if the leaf overflows, i.e. if the number of values in the leaf is greater than the threshold, split the leaf into two halves at its median and add the median to the parent of the leaf, making the two halves children of the parent and perform this operation recursively until there is no overflow.

Algorithm

- Find the leaf into which the value should be inserted recursively and insert the value into the leaf at the right index
- If there is an overflow, call a split_child function on its parent with the child’s index
- In the split_child function
    - split the child into two halves at its median by copying their children if they are not the leaf
    - insert the median into self
    - add the two halves as your children
    - check for overflow and call split_child on your parent if there is one

Fusion Tree

The key idea of a Fusion Tree is to implement a w-bit associate array on top of a B-Tree with a branching factor of ‘w’. B-Tree gives a search time complexity of *$O(\log_w n)$* with an overhead of *$O(w)$* which is necessary for searching through the sorted array and choosing which child to go to. If we can do this operation in *$O(1)$* time, then we can achieve a successor() operation with a complexity of *$O(\log_w n)$.* This is exactly what fusion tree does. With a concept called sketching and parallel_comparision, Fusion Tree compares an element with all the values in a node and decides which child to go to in constant time.

Sketching

This concept helps to distinguish, ‘n’ numbers, each of w bits, with less number of bits by considering the branches at which the bit-tree separates. Let’s consider the below example.

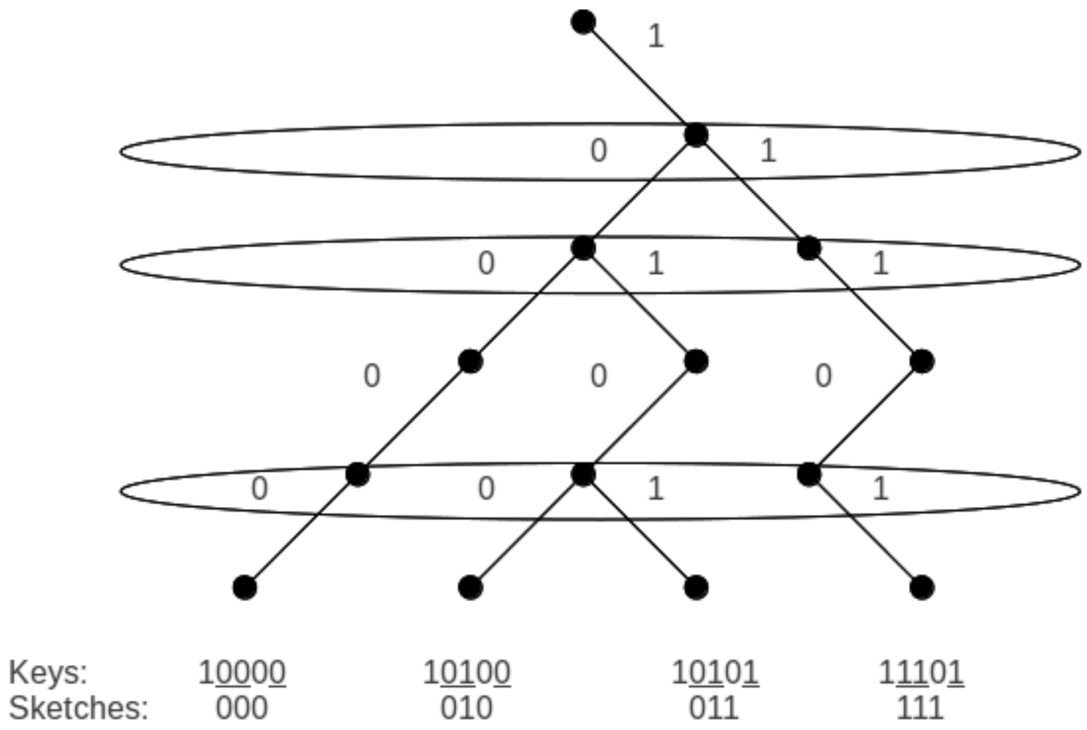

Figure 7: Sketching

In Figure 7, values, [16(10000), 20(10100), 21(10101), 29(11101)] can be distinguished by using just their 1st, 3rd and 4th bits of their binary representation. Hence the bit sequence that is used to distinguish a set of keys is called a sketch. In this example, sketch(16) is 000, sketch(20) is 010 and so on.

Now, there is a challenge here, if we consider the input 22 for the successor, if we take the sketch(22) with the same sketching values of the above values in Figure 7, we get a conflict, cause the sketch(18) would be 010 which is also the sketch of 20. To resolve this conflict, we find the longest common prefix of the value(22) with the value with which there is a conflict and the next value and suppose the longest common prefix is of length y

- If the y+1 th bit of the element is 1, then the successor of the element would be the value next to the element
- IF the y+1 th bit of the element is 0, then the successor of the element is the successor of the new element which is the smallest next largest element, i.e <y-bits>1000… of size w-bit

There is a way to find the longest common prefix of two numbers in *O(1)* time.

Now coming to parallele comparision, consider a node with w-1 values, [v1, v2, v3, v4, …, vw] and their respective sketch values are [sketch(v1), sketch(v2),….., sketch(vw)]. To parallel compare an element with this node, we calculate the sketch(element) and multiply it with

sketch(element) (000..1 000…1 000…1 …………. repeated w times) -- equation 1*

sketch(v1) 1sketch(v2) 1sketch(v3) ………..1sketch(vw) -- equation 2

equation 2 – equation 1 = 0/1….. 0/1……. 0/1…. repeated w time -- equation 3

This is because, every value less than the element will borrow the 1 on its right to make the subtraction possible and every value greater than the elemenet will not borrow that 1 at the msb of each sketch.

if we AND equation 3 with 1000… 1000…. 1000… repreated w times

we get 0/1 0000….. 0/1 0000….. 0/1 0000….. 0/1 0000….. repreated w times -- equation 4

The MSBs of each sketch-size value in equation 4, will be of format 0000…111…. where the first one in that value is the value that is greater than the input element and we proceed in that direction.

If we observe, we need to precompute the sketches of each value in every node beforehand to perform successor() operations, hence this is not a dynamic implementation of a associate array but a static one.

Insert

The insert in this data-structure follows the same algorithm as the one in the B-Tree

Successor

The successor operation uses the above explained concepts like sketching, parallel compare which work in *O(1)* time and searches for the element in the B-Tree and returns its successor accordingly.

Algorithm

- Find the sketch of the element
- Find the child to go to using parallel_compare
- Recurse until leaf and return the value greater than the element
- If you are a leaf and the max value in you is greater than or less than element, return null
- If you are not a leaf, and your child returned null, return the value greater than the child you choose if that is a valid case, or return null if that is an invalid case.

Comparison of Van Emde Boas Tree and Fusion Tree

Theory

Van Emde Boas Tree	Fusion Tree
Successor with O(ln(ln(u)) where u is the universe of the integers	Successor with $O(\log_W n)$ where n is the number of elements in the data structure and w is the bit size of the elements.
Dynamic	Static
Performs best if the number of values in the data structure is high	Performs best if the w is high
	Becomes a 2-3 tree with worst performance with w = 2

Runtime

Universe = 50K

n	Number of search ops	vEB Successor	Fusion Tree Successor
5k	1M	0.127	6.39
50k	1M	0.17	13.16
500k	1M	0.17	10.11
5M	1M	…	10.27
5k	10M	…	10.514
50k	10M	…	132.503
500k	10M	…	101.463
5M	10M	…	101.619

If you observe, since the universe is fixed to 50k, there is no difference in vEB successor with change in n or number of search ops. However, as n increases, fusion tree takes more time to perform search operations.

In another experiment,

Universe = 5M

n= 5M

for 1M search operations, vEB took 0.8 seconds.

Execution Information:

Van Emde Boas Tree:

Build Project: make all

Insert: <vEB_object>.insert(int value);

Successor: <vEB_object>.successor(int value;

Fusion Trees:

Build Project: make all

Insert: <fusion_object>.insert(int value)

PS, Please call <fusion_object>.initialize() after all inserts are completed as it’s a static implementation

Successor: <fusion_object>.successor(int value)

Constraints:

No testing was done for inserting duplicate values.

References

Ocw.mit.edu. (2015). *Divide and Conquer: van Emde Boas Trees*. [online] Available at: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-notes/MIT6_046JS15_lec04.pdf [Accessed 14 Dec. 2019].

Huynh, K., Klerman, S. and Shyu, E. (2012). *Fusion Trees*. [online] Courses.csail.mit.edu. Available at: https://courses.csail.mit.edu/6.851/spring12/scribe/lec12.pdf [Accessed 14 Dec. 2019].

GitHub. (2019). *melink14/veb-fusion*. [online] Available at: https://github.com/melink14/veb-fusion [Accessed 14 Dec. 2019].

Youtube.com. (2019). *YouTube*. [online] Available at: https://www.youtube.com/watch?v=xSGorVW8j6Q [Accessed 14 Dec. 2019].

Youtube.com. (2019). *YouTube*. [online] Available at: https://www.youtube.com/watch?v=hmReJCupbNU [Accessed 14 Dec. 2019].