DEPARTMENT OF ARCHITECTURE AND ENGINEERING

SOFTWARE ENGINEERING

DATA STRUCTURES PROJECT

# *VAN EMDE BOAS TREES*

WORKED BY:

ALVI HYSA

ENGJËLL ABAZAJ

INDRIT FERATI

# TABLE OF CONTENTS

# 1.Overview

A van Emde Boas tree, alternatively called van Emde Boas priority queue is a tree data structure that implements a map (associative array) with m-bit integer keys. It was created in 1975 by Peter van Emde Boas, a Dutch computer scientist and his team.

This data structure performs all operations in *O(log m)* time or in *O(log log M)* time, where M=2^m is the biggest element that can be stored in the tree. It is important to not confuse this parameter M with the total number of elements stored in the tree, which is what is usually used to measure the performance of other tree data structures.

It is also important to note that vEB trees are generally not space efficient. In order to store 32-bit integers for example, it requires 2 to the power of 32 bits of storage. However, there are similar data structures which have the same time efficiency and an even better space efficiency of O(n) (n being the total number of stored elements). But vEB trees can be modified to require only O(n log M) space, which is quite useful.

# 2. Historical Context And Development

The creation of van Emde Boas trees dates back to 1975. The main inventor of this data structure was the Dutch computer scientist Peter van Emde Boas. He worked together with a team in order to produce this algorithm, which included notable contributors such as Kasper de Jonge, Frank Kaashoek and Martin H. Overmars.

**2.1 Origin and Contributors**

Peter van Emde Boas was an important name in the field of theoretical computer science during the middle of the 1970s. His research was mainly focused on efficient algorithms and data structures, which eventually resulted in the development of this particular tree. With the development of this data structure, the team was attempting to develop a tree that could efficiently handle dynamic order statistics operations such as insertion, deletion and successor and predecessor queries on an integer set.

**2.2 Development Timeline**

**1975:** The initial concept of the data structure was introduced. There was a need for an algorithm that was able to perform operations in O(log log M) time, where M represents the biggest element that can be stores in the tree, as previously stated.

**1980s:** The van Emde Boas tree gained a lot of recognition in the academic community. During this time, research was conducted to optimize the space complexity of the algorithm and various improvements and modifications were suggested to improve the practical applicability.

**1990s:** Other related data structures were created, such as y-fast tries and x-fast tries. These structures managed to achieve similar time efficiencies while improving space efficiency, making the van Emde Boas tree data structure more practical.

**2000s to Present:** Research has focused on the implementation of the trees in various programming environments and their applications in different fields, including computational geometry and network routing. Apart from the practical application, a lot of attention has also been directed to the theoretical aspect of this algorithm in advanced computer science courses.

# 3. Detailed Complexity Analysis

**3.1 Universe Size (M=2^m)**

The universe size determines the range of keys that the tree is able to store. For an m-bit integer, M is 2^m.

**3.2 Hierarchical Decomposition**

The tree divides the universe recursively into smaller clusters, where each cluster deals with a subset of the universe, thus allowing for efficient navigation and updates. Clusters are recursively defined van Emde Boas trees that manage subsets of the universe. This way they contribute to the general efficiency of the operations.

The two functions high(x) and low(x) split the integer x into higher and lower bits. These bits are used to identify the relevant cluster and position inside that cluster.

**3.3 Summary Structure**

The summary structure records which clusters contain elements and this creates the opportunity for rapid identification of the next relevant cluster for predecessor and successor queries.

**3.4 Recursive Operations**

Operations such as successor, predecessor, insert and delete are executed recursively and they navigate through the summary and clusters. These operations usually have a time complexity of O(log log M). Maximum and minimum on the other side have constant time complexity, because the vEB tree maintains direct reference to the maximum and minimum elements.

**3.5 Practical Implications**

Van Emde Boas trees are especially useful for applications requiring fast order statistics operation on integer keys. However, the space complexity of this data structure causes limitations in practice and other more optimized variants are usually more desirable in real applications. Still, the theoretical importance of vEB trees cannot be denied.

# 4.Supported Operations

**4.1 Minimum():** This operation finds and returns the minimum key in the tree. (O(1))

```
int VanEmdeBoasTree::Minimum() {
    return min;
}
```

**4.2 Maximum():** This operation finds and returns the maximum key in the tree. (O(1))

```
int VanEmdeBoasTree::Maximum() {
    return max;
}
```

**4.3 Successor(x):** Given a key x, this operation finds the smallest key that is greater than x. (O(log log M))

```
int VanEmdeBoasTree::Successor(int x) {
    if (u == 2) {
        if (x == 0 && max == 1) {
            return 1;
        } else {
            return -1;
        }
    } else if (min != -1 && x < min) {
        return min;
    } else {
        int maxLow = cluster[high(x)]->Maximum();
        if (maxLow != -1 && low(x) < maxLow) {
            int offset = cluster[high(x)]->Successor(low(x));
            return index(high(x), offset);
        } else {
            int succCluster = summary->Successor(high(x));
            if (succCluster == -1) {
                return -1;
            } else {
                int offset = cluster[succCluster]->Minimum();
                return index(succCluster, offset);
            }
        }
    }
}
```

**4.4 Predecessor(x):** Given a key x, this operation finds the largest key that is smaller than x. (O(log log M))

```
int VanEmdeBoasTree::Predecessor(int x) {
    if (u == 2) {
        if (x == 1 && min == 0) {
            return 0;
        } else {
            return -1;
        }
    } else if (max != -1 && x > max) {
        return max;
    } else {
        int minLow = cluster[high(x)]->Minimum();
        if (minLow != -1 && low(x) > minLow) {
            int offset = cluster[high(x)]->Predecessor(low(x));
            return index(high(x), offset);
        } else {
            int predCluster = summary->Predecessor(high(x));
            if (predCluster == -1) {
                if (min != -1 && x > min) {
                    return min;
                } else {
                    return -1;
                }
            } else {
                int offset = cluster[predCluster]->Maximum();
                return index(predCluster, offset);
            }
        }
    }
}
```

**4.5 Insert(x):** This operation inserts an element with key x into the tree. (O(log log M))

```
void VanEmdeBoasTree::Insert(int x) {
    if (min == -1) {
        min = max = x;
    } else {
        if (x < min) {
            int temp = min;
            min = x;
            x = temp;
        }
        if (u > 2) {
            if (cluster[high(x)]->Minimum() == -1) {
                summary->Insert(high(x));
                cluster[high(x)]->min = low(x);
                cluster[high(x)]->max = low(x);
            } else {
                cluster[high(x)]->Insert(low(x));
            }
        }
        if (x > max) {
            max = x;
        }
    }
}
```

**4.6 Delete(x):** This operation removes an element with key x from the tree. (O(log log M))

```
void VanEmdeBoasTree::Delete(int x) {
    if (min == max) {
        min = max = -1;
    } else if (u == 2) {
        if (x == 0) {
            min = 1;
        } else {
            min = 0;
        }
        max = min;
    } else {
        if (x == min) {
            int firstCluster = summary->Minimum();
            x = index(firstCluster, cluster[firstCluster]->Minimum());
            min = x;
        }
        cluster[high(x)]->Delete(low(x));
        if (cluster[high(x)]->Minimum() == -1) {
            summary->Delete(high(x));
            if (x == max) {
                int summaryMax = summary->Maximum();
                if (summaryMax == -1) {
                    max = min;
                } else {
                    max = index(summaryMax, cluster[summaryMax]->Maximum());
                }
            }
        } else if (x == max) {
            max = index(high(x), cluster[high(x)]->Maximum());
        }
    }
}
```

**4.7 Member(x):** This operation checks if a key x is present in the tree. (O(log log M))

```
bool VanEmdeBoasTree::Member(int x) {
    if (x == min || x == max) {
        return true;
    } else if (u == 2) {
        return false;
    } else {
        return cluster[high(x)]->Member(low(x));
    }}
```

# 5.Variants And Extensions

**5.1 X-Fast Tries**

X-fast tires are another variant designed to optimize space usage. They achieve fast operations through hash tables and a trie structure. The universe is represented as a trie, with each level of the trie corresponding to a bit in the element's binary representation. Hash tables are used to store pointers to the nodes in the trie, allowing for quick access and navigation.

Query operations (successor, predecessor, membership) are handled still in O(log log M) time. X-fast tries to use O(n log M) space, where n is the number of elements, making them more space-efficient than traditional vEB trees but less than y-fast tries.

**5.2 Y-Fast Tries**

These trees are an optimized version of vEB trees that address space efficiency while preserving fast query times. They use a combination of x-fast tries and balanced binary search trees.

Y-fast tries consist of a top-level x-fast trie, which provides a compressed representation of the universe. The x-fast trie points to bottom-level BSTs, which store elements of smaller clusters. Operations such as insert, delete and query are handled in O(log log M) time. This algorithm tries to achieve O(n) space complexity, where n is the number of elements stored.

**5.3 Fusion Trees**

Fusion trees are designed to handle integer keys more efficiently by using word-level parallelism and advanced bit manipulation techniques.

Fusion trees use a combination of bit packing and parallel comparison to achieve high efficiency. They store multiple keys in a single machine word, allowing for rapid parallel comparisons.

Operations such as insert, delete and query can be performed in O(log_a n) time, where a is the word size of the machine. These trees use O(n) space.

# 6.Comparison With Other Data Structures

### 6.1 Binary Search Trees

vEB trees have faster query times O(log log M) in comparison with O(log n). However, vEB trees have a higher space and implementation complexity.

### 6.2 Hash Tables

vEB trees maintain order statistics, which hash tables do not provide. Higher space complexity and slower average-case performance for membership queries. O(1) for hash tables and O(log log M) for vEB trees.

### 6.3 Balanced Trees (ex. AVL)

vEB trees provide faster query times for large universe sizes, but they also have higher space and implementation complexity, just like with BSTs.

# 7.Advantages And Applications

## 7.1 Advantages

**Efficiency:** These trees appear to be appropriate for large-scale applications, especially when a large set of integers is present, due to the logarithmic time complexity that this data structure provides.

**Dynamic Sets:** vEB trees are quite appropriate for priority queues and databases, because they effectively maintain dynamic sets.

**Successor and Predecessor Searches:** These query operations are quite useful, especially in databases and graph algorithms where they are required multiple times. Their near-optimal time is very valuable in these scenarios.

## 7.2 Applications

**Databases:** These trees provide a great management of integer keys in database systems, where fast insertion, deletion and search are essential.

**Graph Algorithms:** Multiple graph algorithms make use of vEB trees, especially algorithms that include minimum and maximum tracking, such as dynamic graph problems.

**Cryptography:** The structure of these trees and their ordered nature can be useful in certain cryptographic applications.

## 8.Conclusions

Van Emde Boas(vEB) trees represent a significant advancement in data structure design, particularly for applications involving integer keys and requiring fast dynamic order statistics operations. The primary strength of vEB trees lies in their O(log log M) time complexity for a wide range of operations, including insertion, deletion, membership testing and finding predecessors and successors. This efficiency makes them highly suitable for scenarios where rapid access and updates to integer keys are crucial.

However, the usability of vEB trees comes with trade-offs. The high space complexity of O(M), where M is the universe size, can be a significant limitation, particularly for large universes with sparse elements. This space complexity problems are somewhat improved by variants like y-fast tries and x-fast tries, which offer more practical space complexities while maintaining similar time efficiencies. Despite these optimizations, the implementation complexity and maintenance overhead of vEB trees remain challenges that need careful consideration.

Van Emde Boas trees and their variant remain quite important in advanced data structure theory, providing a robust framework for efficient integer key operations. By balancing the trade-offs between speed and space and understanding the specific needs of their applications, software engineers can utilize these structures to achieve optimal performance in a variety of computational tasks.

## 9.References

https://en.wikipedia.org/wiki/Van_Emde_Boas_tree

https://www.youtube.com/watch?v=hmReJCupbNU

https://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/14/Slides14.pdf

https://courses.csail.mit.edu/6.851/spring12/scribe/L11.pdf

https://courses.csail.mit.edu/6.851/spring12/lectures/L11.pdf