Implementation and comparison of priority queue data structures

Jan Kukowski

Bachelor's thesis
Supervisor: dr hab. inż. Krzysztof Turowski

Jagiellonian University
Department of Theoretical Computer Science
Kraków 2025

Contents

Chapter 1

Introduction

1.1 Definitions

"Introduction to Algorithms" [Cor+22]

Chapter 2

Integer priority queues

2.1 Van Emde Boas tree

The van Emde Boas tree, first introduced in the 1975 paper *"Preserving Order in a Forest in Less Than Logarithmic Time"* [Emd75] by Dutch computer scientist Peter van Emde Boas, is a data structure for storing integer keys from a universe of fixed size M. Van Emde Boas trees support all associative array operations (Search, Insert, and Delete) as well as predecessor and successor queries with a time complexity of $O(\log \log M)$. This is achieved by leveraging a recursive decomposition of the universe into pieces of size $O(\sqrt{M})$.

In this chapter, we focus solely on the use of van Emde Boas trees as priority queues. When used in this context, van Emde Boas trees provide $O(\log \log M)$ time complexity for `push` and `pop` operations and $O(1)$ time complexity for `peek`, by augmenting the data structure to maintain the current minimum value after each operation.

Despite their attractive theoretical performance, van Emde Boas trees have a significant drawback in terms of memory consumption, as the data structure requires $O(M)$ space. This high memory requirement stems from the recursive structure, where subtrees are maintained even if sparsely populated. As a result, van Emde Boas trees are impractical for applications with large universes or limited memory, making them suitable only for scenarios where operations on small integer sets are required.

2.1.1 Structure and complexity

The core idea of the van Emde Boas tree is to recursively partition the universe of size M into $\lceil \sqrt{M} \rceil$ clusters of size $\lceil \sqrt{M} \rceil$. To achieve the desired time complexity for all operations, we need to ensure that each call performs only a constant amount of work plus a single recursive call on a subproblem of size $\lceil \sqrt{M} \rceil$.

Under this assumption, it is easy to prove the following lemma on the complexity of `push` and `pop` operations:

Lemma 2.1.1. The `push` and `pop` operations on a van Emde Boas tree of universe size M run in $O(\log \log M)$ time.

Proof. Each operation performs a constant amount of work plus a single recursive call on a subproblem of size $\lceil\sqrt{M}\rceil$. Thus, the recurrence is:

$$T(M) = T(\sqrt{M}) + O(1)$$

Let's substitute $M = 2^k$, so $\sqrt{M} = 2^{k/2}$, and define $S(k) = T(2^k)$. Then:

$$S(k) = S(k/2) + O(1)$$

This solves to:

$$S(k) = O(\log k)$$

Since $k = \log M$, we have $T(M) = O(\log \log M)$. □

Additionally, to ensure that each operation performs only a single recursive call, the structure is augmented by maintaining the current minimum and maximum values present in the tree at all times. This clearly yields $O(1)$ time complexity for `peek` operations.

We also maintain the cluster size, which is equal to $\lceil\sqrt{M}\rceil$, as well as the summary tree, which is itself a van Emde Boas tree with universe size equal to the number of clusters. The summary tree keeps track of which clusters are non-empty. Each cluster is a van Emde Boas tree with universe size $\lceil\sqrt{M}\rceil$, and is stored in the `clusters` array.

The class outline is as follows:

```
class VanEmdeBoasTree : public PriorityQueue<int> {
    int universeSize;
    int clusterSize;
    int minValue, maxValue;
    std::optional<std::unique_ptr<VanEmdeBoasTree>> summaryTree;
    std::vector<std::optional<std::unique_ptr<VanEmdeBoasTree>>> clusters;
};
```

The structure can be initialized by providing the universe size M:

```
VanEmdeBoasTree::VanEmdeBoasTree(int universeSize)
    : universeSize(universeSize),
      clusterSize(static_cast<int>(std::ceil(std::sqrt(universeSize)))),
      minValue(-1), maxValue(-1) {
    clusters.resize(clusterSize);
}
```

With the structure outline established, we now prove the space complexity of van Emde Boas trees:

Lemma 2.1.2. The space complexity of a van Emde Boas tree over a universe of size M is $O(M)$.

Proof. Each van Emde Boas tree contains one summary van Emde Boas tree of size $\lceil\sqrt{M}\rceil$ and $\lceil\sqrt{M}\rceil$ cluster van Emde Boas trees, each of size $\lceil\sqrt{M}\rceil$. We maintain $\lceil\sqrt{M}\rceil$ pointers to the cluster structures. In addition, there are also a constant number of integer fields, each requiring $O(\log M)$ bits.

Combining these, we get the recurrence:

$$S(M) = (1 + \sqrt{M})S(\sqrt{M}) + O(\sqrt{M})$$

The number of times we take the square root of M before reaching a value less than 2 is $\log \log M$. Unrolling the recurrence until we reach subproblems of size 2 gives:

$$S(M) \leq \left(\prod_{i=1}^{\log \log M} \left(M^{1/2^i} + 1 \right) \right) S(2) + \sum_{i=1}^{\log \log M} O\left(M^{1/2^i} \right) \left(M^{1/2^i} + 1 \right)$$

In the first term, the highest power of M is equal to the sum:

$$\sum_{i=1}^{\log \log M} \frac{1}{2^i}$$

which is a partial sum of a geometric series with sum 1. Thus, the entire term is $O(M)$.

In the second term, the largest subterm occurs at $i = 1$, and gives $O(M)$. For all $i > 1$, the exponent $2/2^i < 1$, so the corresponding terms are $o(M)$. Therefore, the entire second term is $O(M)$.

Combining both terms, we find:

$$S(M) = O(M) + O(M) = O(M)$$

□

In order to quickly locate an element within the tree, a set of simple helper functions is needed. Here, `high` denotes the index of the cluster containing an element, and `low` denotes the index of the element within that cluster. `index` is then used to combine both values.

```
int VanEmdeBoasTree::high(int x) const {
    return x / clusterSize;
}

int VanEmdeBoasTree::low(int x) const {
    return x % clusterSize;
}

int VanEmdeBoasTree::index(int high, int low) const {
    return high * clusterSize + low;
}
```

2.1.2 Push operation

We begin the implementation by handling important base cases. If the tree is empty, we initialize both `minValue` and `maxValue` with the inserted value x, and then return. Checking if the tree is empty is done in $O(1)$ by simply verifying that the `minValue` is initialized. Crucially, this means that we don't store the minimum recursively and that inserting into an empty tree takes constant time.

```
void VanEmdeBoasTree::insert(int x) {
    if (isEmpty()) {
        minValue = maxValue = x;
        return;
    }
```

We maintain the invariant that the minimum value is stored explicitly. If the new value is smaller, we swap it with the current minimum and continue inserting the larger of the two values.

```
    if (x < minValue) {
        std::swap(x, minValue);
    }
```

We then proceed to the main recursive case. For universes larger than 2, we decompose the integer x into two parts: its cluster index and its position within that cluster. If the corresponding cluster or the summary tree does not exist yet, we allocate them dynamically. If the target cluster is empty, we record its index in the summary tree and initialize its min/max directly. Otherwise, we recursively insert the position into the subcluster.

```
    if (universeSize > 2) {
        int clusterIndex = high(x);
        int position = low(x);

        if (!clusters[clusterIndex].has_value()) {
            clusters[clusterIndex] = std::make_unique<VanEmdeBoasTree>(clusterSize);
        }

        if (!summaryTree.has_value()) {
            summaryTree = std::make_unique<VanEmdeBoasTree>(clusterSize);
        }

        if (clusters[clusterIndex].value()->isEmpty()) {
            summaryTree.value()->insert(clusterIndex);
            clusters[clusterIndex].value()->minValue =
                clusters[clusterIndex].value()->maxValue = position;
        } else {
```

```
            clusters[clusterIndex].value()->insert(position);
        }
    }
```

Finally, we update the maximum value explicitly if the inserted value exceeds the current maximum.

```
    if (x > maxValue) {
        maxValue = x;
    }
}
```

In each possible path, we make only a single recursive call to insert. If the cluster we want to insert into is empty, we insert it recursively into the summary tree. If the cluster is not empty, we insert our value recursively into that cluster. All the remaining operations have constant running time, and therefore we achieve the desired time complexity for `push` operations.

2.1.3 Pop operation

Once again, the function begins by handling several base cases. If the tree is empty, there is nothing to remove. If the tree contains a single element, we reset both `minValue` and `maxValue`. Here, we assume that the tree contains the removed element—this is reasonable to assume since we will only ever be removing the current minimum value. If there are only two elements in the tree, we set both `minValue` and `maxValue` to the other element.

```
void VanEmdeBoasTree::remove(int x) {
    if (isEmpty()) {
        return;
    }

    if (minValue == maxValue) {
        minValue = maxValue = -1;
        return;
    }

    if (universeSize <= 2) {
        if (x == 0) {
            minValue = 1;
        } else {
            minValue = 0;
        }
        maxValue = minValue;
        return;
    }
```

What follows is the main recursive case. We split it based on whether the element to be removed is equal to the minimum or maximum value.

If the element to remove is the minimum, we find the new minimum. This is done by querying the summary tree for the first non-empty cluster and removing the minimum value in this cluster. After removing this value, we also remove the cluster index from the summary tree if the cluster becomes empty.

```
if (x == minValue) {
    int firstCluster = summaryTree.value()->minValue;
    int firstClusterMin = clusters[firstCluster].value()->minValue;

    minValue = index(firstCluster, firstClusterMin);
    clusters[firstCluster].value()->remove(firstClusterMin);

    if (clusters[firstCluster].value()->isEmpty()) {
        summaryTree.value()->remove(firstCluster);
    }
}
```

If x is not the current minimum, we remove it from its cluster. Again, we check if the cluster becomes empty and update the summary.

```
else {
    int clusterIndex = high(x);
    int position = low(x);

    clusters[clusterIndex].value()->remove(position);

    if (clusters[clusterIndex].value()->isEmpty()) {
        summaryTree.value()->remove(clusterIndex);
    }
}
```

Finally, if the element to remove was the maximum, we compute the new maximum value. If the summary tree is not empty, we find the last non-empty cluster and retrieve its maximum.

```
if (x == maxValue) {
    if (summaryTree.value()->isEmpty()) {
        maxValue = minValue;
    } else {
        int lastCluster = summaryTree.value()->maxValue;
        maxValue = index(lastCluster, clusters[lastCluster].value()->maxValue);
    }
```

```
    }
}
```

It might not be immediately clear why this operation achieves the desired complexity, since in some cases we make two recursive calls to `remove`. Here, we use the fact that removing from a tree that contains a single element takes constant time. We only make the second recursive call in the case where the first one was on a tree containing one element, and therefore only one of the calls will require non-constant time. Therefore, both `push` and `pop` achieve $O(\log \log M)$ time complexity.

2.2 X-fast trie

"Log-logarithmic worst-case range queries are possible in space $\Theta(N)$*"* [Wil83]

2.3 Y-fast trie

"Log-logarithmic worst-case range queries are possible in space $\Theta(N)$*"* [Wil83]

2.4 Fusion tree

"BLASTING Through the Information Theoretic Barrier with FUSION TREES" [FW90]

Chapter 3

General purpose priority queues

3.1 Weak heap

"Weak-heap sort" [Dut93]

3.2 Brodal queue

"Worst-case efficient priority queues" [Bro96]

Chapter 4

Implementation and benchmark

4.1 Implementation and benchmark

Bibliography

[Bro96] Gerth Stølting Brodal. "Worst-case efficient priority queues". In: *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*. 1996, pp. 52–58.

[Cor+22] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2022.

[Dut93] Ronald D. Dutton. "Weak-heap sort". In: *BIT* 33.3 (1993), pp. 372–381.

[Emd75] Peter van Emde Boas. "Preserving order in a forest in less than logarithmic time". In: *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*. 1975, pp. 75–84.

[FW90] M. L. Fredman and D. E. Willard. "BLASTING Through the Information Theoretic Barrier with FUSION TREES". In: *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing (STOC '90)*. New York, NY: ACM, 1990, pp. 1–7.

[Wil83] Dan E. Willard. "Log-logarithmic worst-case range queries are possible in space Θ(N)". In: *Information Processing Letters* 17.2 (1983), pp. 81–84.