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

The X-fast trie is a data structure introduced by Dan Willard in his 1983 paper *"Log-logarithmic worst-case range queries are possible in space* $\Theta(N)$*"* [Wil83]. For a universe of size M, it supports search, predecessor, and successor queries in $O(\log \log M)$ time, while insertions and deletions require $O(\log M)$ time. The data structure uses $O(n \log M)$ space, where n is the number of stored elements. X-fast tries were developed as a foundation for the more advanced Y-fast tries, which incorporate X-fast tries as components.

The X-fast trie builds on the structure of a standard trie, enhancing it with binary search across trie levels to improve query efficiency. Although the `XFastTrie` class implements the predefined `PriorityQueue` interface, it must also support public predecessor queries and arbitrary key removal, as these operations are essential for integration into the Y-fast trie.

2.2.1 Structure and complexity

The core idea is to store all elements in a binary trie of height $\lceil \log_2 M \rceil$, where the keys are treated as binary numbers, padded with zeros to ensure uniform length. Internal nodes with a missing child preserve jump pointers: if a node has no left child, it stores a pointer to the minimum leaf in its right subtree; if it has no right child, it stores a pointer to the maximum leaf in its left subtree. Only the leaves contain actual keys, and these are connected into a doubly linked list to enable constant-time access to predecessor and successor. Each node in the trie corresponds to a prefix of some stored key, and every level of the trie is augmented with a hash table that maps prefixes to their corresponding nodes.

The simplified class outline looks like this:

```
class XFastTrie : public PriorityQueue<Key, Compare> {
    Key universeSize;
    int bitWidth;
    size_t size = 0;
    std::shared_ptr<Node> root;
    std::shared_ptr<Node> dummy;
    std::vector<std::unordered_map<Key, std::shared_ptr<Node>>> prefixLevels;
};
```

The `dummy` is a sentinel node, which demarcates the beginning and end of our doubly linked list of leaves. The element right after the sentinel will be the minimum element, while the element right before it will be the maximum.

The trie consists of nodes structured as follows:

```
struct XFastTrie::Node {
    std::shared_ptr<Node> children[2];
    std::shared_ptr<Node> parent;
    std::shared_ptr<Node> jump;
    std::shared_ptr<Node> linkedNodes[2];
    std::optional<Key> key;
};
```

Lemma 2.2.1. The space complexity of a X-fast trie storing n keys over a universe of size M is $O(n \log M)$.

Proof. Each node contains a constant number of pointers to other nodes and a single key. Therefore, the space requirement for a single node is constant. For each key, we maintain $O(\log M)$ nodes. Therefore, the space requirement for the entire bitwise trie part of the structure, including the doubly linked list of leaves, is $O(n \log M)$. Moreover, each node is also pointed to in the `prefixLevels` structure, which therefore requires an additional $O(n \log M)$ pointers to nodes. Finally, all remaining variables require constant space. □

We also define some helper constants to clarify referencing the left and right child (in the case of the `children` array) and the previous or next key (in the case of the `linkedNodes` array).

```
static constexpr int LEFT = 0;
static constexpr int RIGHT = 1;
static constexpr int PREV = 0;
static constexpr int NEXT = 1;
```

The `XFastTrie` is initialized by providing the universe size M:

```
XFastTrie::XFastTrie(Key universeSize)
    : universeSize(universeSize),
      bitWidth(calculateBitWidth(universeSize)),
      root(std::make_shared<Node>()),
      dummy(std::make_shared<Node>()),
      prefixLevels(bitWidth + 1)
{
    dummy->linkedNodes[PREV] = dummy;
    dummy->linkedNodes[NEXT] = dummy;
}
```

2.2.2 Push operation

The `push` operation begins by traversing down the trie along the path corresponding to the `key` to be inserted. If the entire path already exists, we return `false`, indicating that the key is already present.

```
bool XFastTrie::insert(Key key) {
    auto node = root;
    int level = 0;

    for (; level < bitWidth; ++level) {
        int bit = getBit(key, level);
        if (!node->children[bit]) break;
        node = node->children[bit];
    }

    if (level == bitWidth) {
        return false;
    }
```

If the key is not already present, a series of helper methods is invoked. First, the predecessor and successor of the new key are located. Next, the remaining path down the trie is constructed based on the key's bits. The new leaf is then linked between its predecessor and successor. Finally, jump pointers are updated by traversing up the trie, and the hash tables at each level are modified accordingly.

```
    auto [predecessor, successor] = getInsertNeighbors(node, key, level);

    node->jump = nullptr;
    auto insertedNode = createPath(node, key, level);

    linkNeighbors(insertedNode, predecessor, successor);
    updateJumpPointers(insertedNode, key);
    updatePrefixLevels(key);

    return true;
}
```

We can use the jump pointers directly to return the predecessor and successor of a key, given the lowest node matching a prefix of the key. If this node has no right child, the predecessor of the key will be the largest leaf in the node's left subtree, pointed to by the jump pointer; the successor will then be the predecessor's successor, or the minimum value in the trie if the predecessor doesn't exist. Similarly, if the node has no left child, the successor of the key will be the smallest leaf in the node's right subtree, pointed to by the

jump pointer; the predecessor will then be the successor's predecessor, or the maximum value in the trie if the successor doesn't exist.

```
std::pair<std::shared_ptr<Node>, std::shared_ptr<Node>> XFastTrie::getInsertNeighbors
(std::shared_ptr<Node> node, Key key, int level) const {
    int bit = getBit(key, level);
    if (bit == RIGHT) {
        auto pred = node->jump;
        auto succ = pred ? pred->linkedNodes[NEXT] : dummy->linkedNodes[NEXT];
        return {pred, succ};
    } else {
        auto succ = node->jump;
        auto pred = succ ? succ->linkedNodes[PREV] : dummy->linkedNodes[PREV];
        return {pred, succ};
    }
}
```

The following utility functions complete the insertion by adding any missing nodes to the trie, linking the new leaf into the doubly linked list, and updating the hash tables at each level by traversing the full path to the leaf.

```
std::shared_ptr<Node> XFastTrie::createPath
(std::shared_ptr<Node> node, Key key, int startLevel) {
    for (int level = startLevel; level < bitWidth; ++level) {
        int bit = getBit(key, level);
        node->children[bit] = std::make_shared<Node>();
        node->children[bit]->parent = node;
        node = node->children[bit];
    }
    node->key = key;
    return node;
}

void XFastTrie::linkNeighbors
(std::shared_ptr<Node> node, std::shared_ptr<Node> pred, std::shared_ptr<Node> succ) {
    node->linkedNodes[PREV] = pred;
    node->linkedNodes[NEXT] = succ;
    if (pred) pred->linkedNodes[NEXT] = node;
    if (succ) succ->linkedNodes[PREV] = node;
}

void XFastTrie::updatePrefixLevels(Key key) {
    auto node = root;
    for (int i = 0; i <= bitWidth; ++i) {
```

```
        prefixLevels[i][getPrefix(key, i)] = node;
        if (i < bitWidth) {
            int bit = getBit(key, i);
            node = node->children[bit];
        }
    }
}
```

We also need to update the jump pointers along the entire path. We do this by going from the leaf upward. At each node traversed, we make sure to maintain the invariant: if a node has no left child, it stores a pointer to the minimum leaf in its right subtree; if it has no right child, it stores a pointer to the maximum leaf in its left subtree.

```
void XFastTrie::updateJumpPointers(std::shared_ptr<Node> node, Key key) {
    auto parent = node->parent;
    while (parent) {
        if ((parent->children[LEFT] == nullptr &&
            (!parent->jump || parent->jump->key > key)) ||
            (parent->children[RIGHT] == nullptr &&
            (!parent->jump || parent->jump->key < key))) {
            parent->jump = node;
        }
        parent = parent->parent;
    }
}
```

Lemma 2.2.2. The average time complexity of the `push` operation on X-fast tries is $O(\log M)$

Proof. The complexity is dominated by a constant number of traversals over a path down the trie. Each such traversal requires $O(\log M)$ time, as the height of the trie is $\lceil \log_2 M \rceil$ and at each level we perform a constant amount of work. This is under the assumption that operations on the `prefixLevels` structure have average $O(1)$ time complexity. □

2.2.3 Pop operation

Like the `push` operation, the `pop` operation starts by traversing the trie along the path corresponding to the `key` to be removed. If the full path cannot be matched, it returns `false`, indicating the element is not present. Subsequently, several helper methods are invoked to remove the element from the linked list of leaves and to clean up both the trie path and the hash tables.

```
bool XFastTrie::remove(Key key) {
    auto node = root;
```

```
    for (int level = 0; level < bitWidth; ++level) {
        int bit = getBit(key, level);
        if (!node->children[bit]) return false;
        node = node->children[bit];
    }

    cleanupPath(node, key);
    unlinkNode(node);

    return true;
}
```

First, we remove the now unused nodes from both the trie and the hash tables at each level. If a node now has no children, we also clear its jump pointer.

```
void XFastTrie::cleanupPath(std::shared_ptr<Node> node, Key key) {
    auto parent = node->parent;
    int level = bitWidth - 1;

    for (; level >= 0; --level) {
        int bit = getBit(key, level);

        parent->children[bit] = nullptr;
        prefixLevels[level + 1].erase(getPrefix(key, level + 1));
        if (!parent->children[1 - bit]) parent->jump = nullptr;

        if (parent->children[1 - bit]) break;
        parent = parent->parent;
    }
}
```

Next, we update the jump pointers for the parent of the removed node and all ancestors above it that previously pointed to this node. If such an ancestor lacks a left child, its jump pointer is updated to the node's successor, which is now the smallest key in the ancestor's right subtree. Conversely, if the ancestor lacks a right child, we update its jump pointer to the node's predecessor, now the largest key in the ancestor's left subtree.

```
if(parent)
    parent->jump = node;

for (; level >= 0; --level) {
    if (parent->jump == node) {
        if(!parent->children[LEFT]) {
            parent->jump = node->linkedNodes[NEXT];
```

```
        } else if(!parent->children[RIGHT]) {
            parent->jump = node->linkedNodes[PREV];
        }
    }
    parent = parent->parent;
}
```

Finally, we remove the element from the linked list of leaves.

```
void XFastTrie::unlinkNode(std::shared_ptr<Node> node) {
    if (node->linkedNodes[PREV]) {
        node->linkedNodes[PREV]->linkedNodes[NEXT] = node->linkedNodes[NEXT];
    }
    if (node->linkedNodes[NEXT]) {
        node->linkedNodes[NEXT]->linkedNodes[PREV] = node->linkedNodes[PREV];
    }
}
```

The average time complexity of the `pop` operation on X-fast tries is $O(\log M)$. We omit the analysis as it is closely analogous to the analysis for `push`.

2.2.4 Predecessor operation

While `pop` and `push` don't offer any advantage in terms of time complexity over standard binary tries, they maintain a structure that now allows us to elegantly perform `predecessor` queries in $O(\log \log M)$ worst-case time.

Predecessors are found by a binary search over the levels of the trie to identify the longest matching prefix of the query key.

```
std::shared_ptr<Node> XFastTrie::findPredecessorNode(Key key) const {
    int low = 0, high = bitWidth + 1;
    auto node = root;

    while (high - low > 1) {
        int mid = (low + high) / 2;
        auto it = prefixLevels[mid].find(getPrefix(key, mid));
        if (it == prefixLevels[mid].end()) {
            high = mid;
        } else {
            node = it->second;
            low = mid;
        }
    }
```

After locating this prefix, the structure leverages jump pointers and predecessor/successor pointers from the linked list to identify the predecessor or successor of the corresponding node. If the key's first unmatched bit is 1, we retrieve the largest key in the node's left subtree via the jump pointer. Otherwise, we find the smallest key in the right subtree using jump pointers and then move one position backward in the linked list.

```
    if (low == bitWidth && node->key.has_value()) {
        return node;
    }

    int dir = getBit(key, low);
    return (dir == RIGHT) ? node->jump : (node->jump ? node->jump->linkedNodes[PREV] : null
}
```

Lemma 2.2.3. The worst-case time complexity of the `predecessor` operation on X-fast tries is $O(\log \log M)$.

Proof. The complexity is determined by the complexity of the binary search on the levels of the trie, as all other operations require constant time. This binary search looks for a value in a list of $O(\log M)$ levels, and therefore it takes $O(\log \log M)$ time. □

2.3 Y-fast trie

The Y-fast trie is a data structure introduced by Dan Willard in his 1983 paper, *"Log-logarithmic worst-case range queries are possible in space* $\Theta(N)$*"* [Wil83], which also presented the X-fast trie. It supports a dynamic set of integer keys from a universe of size M, allowing all standard associative array operations in expected $O(\log \log M)$ time. Notably, it achieves this performance using only $O(n)$ space, where n is the number of stored elements, which is a significant improvement over van Emde Boas trees.

2.3.1 Structure and complexity

The Y-fast trie builds upon the X-fast trie by partitioning the key set into disjoint subsets of elements, each stored in a balanced binary search tree such as a treap. A representative from each subset is stored in an X-fast trie in order to quickly locate the balanced binary search tree where the queried element is located.

The class outline looks like this:

```
class YFastTrie : public PriorityQueue<Key, Compare> {
    Key universeSize;
    int bitWidth;
    size_t size = 0;
    std::optional<Key> minimum;
    XFastTrie<Key, Compare> representatives;
```

```
    std::map<Key, std::shared_ptr<Koala::Treap<Key>>> buckets;
};
```

And like previously discussed structures, in can be initialized by providing the universe size M:

```
YFastTrie::YFastTrie(Key universeSize)
    : universeSize(universeSize),
      bitWidth(calculateBitWidth(universeSize)),
      representatives(universeSize),
      minimum(std::nullopt) {}
```

Assuming there are at least $\frac{\log M}{2}$ elements stored in the trie, we maintain a crucial invariant across all operations: each balanced binary search tree contains $\Theta(\log M)$ elements. Thus, there will be $O(n/\log M)$ representatives stored in the X-fast trie. We achieve this by splitting the BST during insertions if its size would exceed $2 \cdot \log M$ after the insertion. Similarly, during deletions, if the size of a BST drops below $\frac{\log M}{2}$, we merge it with one of the other BSTs.

Importantly, to preserve this invariant without compromising the overall time complexity of the structure, we need to use balanced binary search trees that support efficient `split` and `merge` operations. Treaps are a natural candidate, as they are well-known to support both operations in logarithmic time. Since each tree stores $O(\log M)$ elements, `split` and `merge` will take $O(\log \log M)$ time, which is sufficient for our needs.

With the invariant in mind, we now establish the upper bound for the memory required by an Y-fast trie.

Lemma 2.3.1. The space complexity of an Y-fast trie storing n elements is $O(n)$

Proof. As previously discussed, an X-fast trie over a universe of size M, storing n elements, requires $O(n \log M)$ space. In the Y-fast trie, only $O(n/\log M)$ representatives are stored in the X-fast trie, resulting in a space requirement of $O((n/\log M)\log M) = O(n)$ for the representatives trie.

Each element is also stored in a corresponding balanced binary search tree. Since each element appears exactly once, the total space required by all such trees is $O(n)$.

Aside from this, the data structure maintains only a constant number of variables. Therefore, the overall space complexity of the Y-fast trie is $O(n)+O(n)+O(1) = O(n)$. □

In addition to the standard operations supported by a treap, we use a helper function called `splitInHalf` to divide a bucket into two approximately equal halves. This function works by locating the middle element of the treap and then performing a standard split at that point:

```
void YFastTrie::splitBucket(std::shared_ptr<Koala::Treap<Key>>& bucket) {
    auto left = std::make_shared<Koala::Treap<Key>>();
    auto right = std::make_shared<Koala::Treap<Key>>();
```

```
    bucket->splitInHalf(*left, *right);

    Key newRepresentative = right->kth(1);
    representatives.push(newRepresentative);
    buckets[newRepresentative] = right;

    Key oldRepresentative = left->kth(1);
    buckets[oldRepresentative] = left;
}
```

2.3.2 Push operation

To insert a new key x into a Y-fast trie, we must first identify the balanced binary search tree that should contain it. Each BST corresponds to a disjoint subset of elements, and is represented in the X-fast trie by its minimum element. Therefore, to find the correct subset for x, we perform a predecessor query in the X-fast trie. The result is the largest representative $r \leq x$, which is the minimum element of the BST that should contain x. If no such representative exists (x is smaller than all current representatives), then x belongs to the first BST. If no BSTs exist yet, we initialize the first bucket. As described in the chapter on X-fast tries, the predecessor query takes expected $O(\log \log M)$ time. Then, we proceed to insert the element into the selected bucket in $O(\log \log M)$ time.

```
void YFastTrie::insert(const Key& key) {
    auto representative = findRepresentative(key);
    std::shared_ptr<Koala::Treap<Key>> bucket;

    if (!representative.has_value()) {
        if (!empty()) {
            Key rep = representatives.peek();
            bucket = buckets[rep];
            representative = rep;
        } else {
            bucket = std::make_shared<Koala::Treap<Key>>();
            representatives.push(key);
            buckets[key] = bucket;
            representative = key;
        }
    } else {
        bucket = buckets[*representative];
    }

    bucket->insert(key);
    ++size;
```

The minimum value is updated if necessary. Next, we determine whether there's a need to update the representative. If so, we remove the old value from `representatives` and `buckets` keyset and insert the new value. Again, each of these operations $O(\log \log M)$ time.

Finally, if the insertion causes the bucket to exceed its allowed size, we split it into two smaller buckets. This maintains the invariant that each bucket contains $\Theta(\log M)$ elements.

```
    if (!minimum.has_value() || key < *minimum) {
        minimum = key;
    }

    if (bucket->kth(1) != *representative) {
        Key newRepresentative = bucket->kth(1);
        representatives.remove(*representative);
        representatives.push(newRepresentative);
        buckets.erase(*representative);
        buckets[newRepresentative] = bucket;
        representative = newRepresentative;
    }

    if (bucket->size() > 2 * bitWidth) {
        splitBucket(bucket);
    }
}
```

Overall, the entire insertion process runs in expected $O(\log \log M)$ time, including locating the appropriate subset, updating the BST, and maintaining the structure's invariants through splits and representative updates.

2.3.3 Pop operation

To delete a key x from a Y-fast trie, we begin by locating the balanced binary search tree that contains it. As with insertion, this is done via a predecessor query in the X-fast trie, which returns the largest representative $r \leq x$.

```
void YFastTrie::remove(const Key& key) {
    auto representative = findRepresentative(key);
    if (!representative.has_value()) {
        return;
    }

    auto it = buckets.find(*representative);

    auto& bucket = it->second;
```

We then proceed to remove `key` from the bucket found. If the bucket becomes empty as a result, we remove it entirely. If the element removed was the representative, we update `representatives` and `buckets` with the smallest element remaining in the bucket. Likewise, if the deleted element was the `minimum`, we update the minimum by querying the X-fast trie, which can be done in constant time.

```
if (bucket->contains(key)) {
    bucket->erase(key);
    --size;

    if (bucket->size() == 0) {
        representatives.remove(*representative);
        buckets.erase(*representative);
    } else if (key == *representative) {
        Key newRepresentative = bucket->kth(1);
        representatives.remove(*representative);
        representatives.push(newRepresentative);
        buckets[newRepresentative] = bucket;
        buckets.erase(*representative);
        representative = newRepresentative;
    }

    if (minimum.has_value() && key == *minimum) {
        if (!representatives.empty())
            minimum = representatives.peek();
        else
            minimum = std::nullopt;
    }
}
```

After deletion, we check whether the size of the BST has fallen below $\frac{\log M}{2}$. If it has, we attempt to merge it with a neighboring BST. This involves locating an adjacent bucket, merging the two trees, and updating the set of representatives to reflect the change. First, we try to merge our bucket with its successor in `representatives`.

```
if (bucket->size() > 0 && bucket->size() < (bitWidth + 1) / 2) {
    Key currentRep = bucket->kth(1);
    auto it = buckets.find(currentRep);
    if (it != buckets.end()) {
        auto nextIt = std::next(it);
        if (nextIt != buckets.end()) {
            auto nextRepresentative = nextIt->first;
            it->second->mergeFrom(*nextIt->second);
            representatives.remove(nextRepresentative);
```

```
                buckets.erase(nextRepresentative);
                if (it->second->size() > 2 * bitWidth) {
                    splitBucket(it->second);
                }
            }
```

If no successor is available, we attempt the merge with the predecessor instead. It's important to note that the BST resulting from the merge may itself break the size invariant. If that happens, we must split the merged tree to rebalance the sizes and restore the invariant.

```
            else if (it != buckets.begin()) {
                auto prevIt = std::prev(it);
                prevIt->second->mergeFrom(*it->second);
                representatives.remove(currentRep);
                buckets.erase(currentRep);
                if (prevIt->second->size() > 2 * bitWidth) {
                    splitBucket(prevIt->second);
                }
            }
        }
    }
}
```

The merge operation requires $O(\log \log M)$ time, as does the rebalancing split if needed. Additionally, removing an outdated representative from the X-fast trie takes expected $O(\log \log M)$ time.

Altogether, `pop` has an expected time complexity of $O(\log \log M)$. This includes locating the appropriate BST, removing the element, and restoring the invariant through merging and updating representatives.

2.4 Fusion tree

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

Chapter 3

General purpose priority queues

3.1 Weak heap

The weak heap is a data structure related to binary and binomial heaps, introduced by Ronald Dutton in his 1993 paper *"Weak-heap sort"* [Dut93] as an efficient priority queue for sorting. Its primary advantage is the reduced number of comparisons required for each operation. It supports both `push` and `pop` operations in $O(\log n)$ worst-case time, using at most $\lceil \log n \rceil$ element comparisons.

3.1.1 Structure and complexity

Weak heaps loosen the structural requirements of standard binary heaps. In weak heap ordering (assuming a `min` priority queue, which we will adopt throughout this chapter), each element is required to be smaller than every element in its right subtree, while its relation to elements in the left subtree is unrestricted. As a consequence, to guarantee that the smallest element is at the root, the root must have no left child.

The array-based representation of weak heaps closely resembles that of standard binary heaps, with one key distinction: each element is associated with a corresponding `flip` bit. When this bit is set to 1, it indicates that the children of the corresponding node are swapped.

The class outline of a weak heap is extremely simple and is shown below:

```
class WeakHeap : public PriorityQueue<Key, Compare> {
    std::vector<Key> data;
    std::vector<bool> flip;
    Compare comp;
};
```

For any element at index k, we refer to the element at $2k + \text{flip}[k]$ as its left child, and to the element at $2k + 1 - \text{flip}[k]$ as its right child.

For the purpose of analysis, it is convenient to interpret the weak heap as a multi-way tree satisfying the standard heap property, represented as a binary tree using the right-child left-sibling convention. In this view, the right child of a node corresponds to its first

child in the multi-way tree, while the left child corresponds to its next sibling. We will therefore refer to the right child as the first child of a node, and to the left child as its next sibling.

We next define the distinguished ancestor of a node. In the multi-way tree representation, this simply corresponds to the direct parent of the node. In the binary tree representation, it is the parent of the first node on the path from the given node to the root that is a right child.

Put simply, the distinguished ancestor is the lowest ancestor of a node that, assuming the weak heap invariant holds everywhere, is guaranteed to store a smaller element. Because the distinguished ancestor plays a central role in weak heap operations, we define a dedicated method to compute it:

```
std::size_t WeakHeap::distinguishedAncestor(std::size_t index) const {
    while ((index % 2) == flip[index / 2]) {
        index = index / 2;
    }
    return index / 2;
}
```

The average distance d from a node to its distinguished ancestor is approximately 2. This follows from the observation that the distance is at least 1, and in half of the cases, the search proceeds one additional level up the tree. This results in $d = 1 + d/2$.

The central operation in our analysis is the `join` operation. Given a node j and its distinguished ancestor i, the `join` restores the weak heap property between the subtrees rooted at i and j. Crucially, this is the only operation that performs a direct comparison between two elements. We will always invoke the following method with a node and its distinguished ancestor as arguments, assuming that the weak heap property holds everywhere else except possibly between i and j:

```
bool WeakHeap::join(std::size_t parent, std::size_t child) {
    if (comp(data[child], data[parent])) {
        std::swap(data[parent], data[child]);
        flip[child] = !flip[child];
        return false;
    }
    return true;
}
```

If the weak heap property holds between i and j, no action is required. Otherwise, we first exchange the children of the lower root j and then swap j with its ancestor i. Exchanging the roots restores the weak heap property (in the multi-way tree representation) between i and j. The children exchange is necessary to ensure that the losing (larger) root preserves its original subtree after being moved. Specifically, the lower root's left child (or next sibling) becomes the former higher root's right child (or first child), preserving

its role as a child of the higher root. The lower root's right child (or first child) becomes the former higher root's left child. This does not violate the weak heap property, since it imposes no restrictions on left children.

Similar to the sift operations in a standard binary heap, `siftUp` restores the weak heap property along the entire path from j up to the root.

```
void WeakHeap::siftUp(std::size_t start) {
    std::size_t current = start;
    while (current != 0) {
        std::size_t ancestor = distinguishedAncestor(current);
        if (join(ancestor, current)) {
            break;
        }
        current = ancestor;
    }
}
```

The `siftDown` operation restores the weak heap property along the entire path starting at the leftmost descendant of the right child of j. This behavior is easier to visualize in the multi-way tree representation, where the right child of j and all nodes reached by repeatedly following left children are direct children of j in the multi-way tree.

```
void WeakHeap::siftDown(std::size_t start) {
    std::size_t size = data.size();
    std::size_t descendant = 2 * start + 1 - flip[start];
    while (2 * descendant + flip[descendant] < size) {
        descendant = 2 * descendant + flip[descendant];
    }
    while (descendant != start) {
        join(start, descendant);
        descendant = descendant / 2;
    }
}
```

Both of the above operations run in $O(\log n)$ average time, taking into account the constant average time required to find the distinguished ancestor. More importantly, they both require at most $\lceil \log n \rceil$ direct element comparisons. The height of the heap is at most $\lceil \log n \rceil + 1$, since, if we ignore the flips, it is represented as a complete binary tree. At most one comparison is performed at each level of the heap, except for the level containing the initial argument j.

3.1.2 Push and pop operations

With all of the helper functions in place, the implementation of `push` and `pop` is straightforward and closely resembles that of a typical binary heap. The `push` operation first inserts

the element at the end of the array representing the heap and initializes its corresponding flip bit. We must also reset the parent's flip bit, as it may have an arbitrary value from earlier `join` calls, if the new node will be the parent's only child. Finally, we sift the element up the heap, restoring the weak heap property at successive distinguished ancestors if it has been violated.

```
void WeakHeap::push(const Key& key) override {
    std::size_t index = data.size();
    data.push_back(key);
    flip.push_back(0);

    if ((index % 2 == 0) && index > 0) {
        flip[index / 2] = 0;
    }

    siftUp(index);
}
```

Much like in a standard binary heap, the `pop` operation begins by removing the root from the array representation and replacing it with the last element in the heap. The element is then sifted down to restore the weak heap property. As explained earlier, this involves comparing the new root to all elements along the leftmost path in its right subtree. These nodes are the only possible candidates for the new root, since, by the weak heap property, the elements in their respective right subtrees are guaranteed to be larger.

```
Key WeakHeap::pop() override {
    if (empty()) {
        throw std::runtime_error("Priority queue is empty");
    }
    Key minimum = data[0];
    std::size_t last = data.size() - 1;
    data[0] = data[last];
    data.pop_back();
    flip.pop_back();
    if (last > 1) {
        siftDown(0);
    }
    return minimum;
}
```

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.