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

The primary goal of this dissertation is to provide a thorough analysis and clean, efficient implementations of several lesser-known data structures that conform to a unified priority queue interface.

Chapter 1 introduces the fundamental functionality of priority queues and briefly surveys the most widely used and well-understood implementation strategies. Chapter 2 explores priority queue variants designed for scenarios where keys are drawn from a bounded universe (specifically, a finite set of unsigned integers), with performance characteristics that depend on the size of this universe. Chapter 3 presents alternative data structures to those discussed in the introductory chapter, focusing on designs that may offer simpler implementations or improved performance under specific conditions. Finally, Chapter 4 evaluates the practical efficiency of all implemented structures through benchmarking on large datasets.

1.1 Definitions

A priority queue is an abstract data structure that maintains a set of elements, each associated with a specific *priority*. It supports access to the element with the highest or lowest priority, depending on the variant. A *min*-priority queue provides access to the element with the lowest priority, while a *max*-priority queue provides access to the element with the highest priority.

The following definition, as given in *Introduction to Algorithms* [Cor+22], specifies the standard operations supported by a priority queue managing a collection S of elements:

- `insert(S, x)`: Insert element x into the priority queue.
- `maximum(S)` / `minimum(S)`: Return the element with the maximum or minimum key.
- `extractMax(S)` / `extractMin(S)`: Remove and return the element with the maximum or minimum key.
- `increaseKey(S, x, k)` / `decreaseKey(S, x, k)`: Modify the key of element x.

To simplify our analysis and avoid repetition, this work consistently assumes the use of a *min*-priority queue. Where necessary, clarifications are provided in the text to avoid ambiguity. Based on a slightly adapted version of the classical definition, we define a priority queue in terms of three fundamental *base operations*:

- `push(k)`: Inserts a key into the queue.
- `peek()`: Returns (without removing) the element with the lowest priority.
- `pop()`: Removes and returns the element with the lowest priority.

These core operations are sufficient for a wide range of applications and are required in all implemented priority queues.

Additionally, we identify two *optional operations* that, while not universally supported, are important in some use cases. These will be included in selected implementations:

- `decreaseKey(x, k)`: Decreases the key of element x to a new value k.
- `meld(q)`: Merges another priority queue q into the current one.

To formalize this interface, we define a generic C++ class template capturing the essential functionality of a priority queue:

```cpp
template <class Key, class Compare = std::less<Key>>
class PriorityQueue {
public:
     virtual Key pop() = 0;
     virtual Key peek() const = 0;
     virtual void push(const Key& key) = 0;
     virtual bool empty() const = 0;
};
```

This interface serves as the foundation for all implementations explored throughout this work.

In priority queue implementations that support optional operations, the operation signatures take the following form. Here, `PriorityQueue` denotes the specific implementation class, and `Node` represents the node type used in that implementation:

```cpp
void PriorityQueue::meld(PriorityQueue& other);
void PriorityQueue::decreaseKey(Node* node, Key newKey);
```

1.2 Overview

This section provides a brief historical overview of the development of priority queue data structures. We highlight several widely used and theoretically significant variants, setting the stage for the implementations and comparisons that follow. At the end of the section, we also provide a table summarizing the time complexities of priority queue operations on various implementations.

1.2.1 Classical approaches

The most straightforward, albeit inefficient, implementations of priority queues use either unsorted or sorted arrays (or lists). An unsorted array allows constant-time insertions, but locating and removing the element with the lowest priority requires scanning the entire collection, resulting in $O(n)$ time for both `pop` and `peek`. In contrast, a sorted array offers constant-time access to the minimum element, but insertion becomes costly, taking $O(n)$ time to maintain the sorted order.

The binary heap, introduced by J.W.J. Williams in 1964 [Wil64], is the most commonly used priority queue in both academic and practical settings. A binary heap (specifically, a *min*-heap in our context) is a complete binary tree that satisfies the heap-order property: each node is less than or equal to its children. Represented implicitly in an array, it supports both insertion and minimum deletion in $O(\log n)$ time and offers constant-time access to the minimum via `peek`.

Introduced by Fredman and Tarjan in 1987 [FT87], the Fibonacci heap consists of a forest of heap-ordered trees and is one of the earliest and best-known examples of amortized analysis. Most operations run in $O(1)$ amortized time, while `pop` takes $O(\log n)$ amortized time. Although asymptotically optimal in the amortized sense (as improving these bounds would violate known lower bounds for sorting), Fibonacci heaps are rarely used in practice due to their complex implementation and poor constant factors.

In contrast to amortized-efficient structures, worst-case optimal priority queues guarantee tight bounds on every operation. The two most notable examples are the *Brodal queue* and the *Strict Fibonacci heap*. These data structures support all operations except `pop` in $O(1)$ worst-case time; `pop` itself requires $O(\log n)$ worst-case time.

The Brodal queue, introduced by Gerth Stølting Brodal in 1996 [Bro96], was the first structure to achieve these bounds. The Strict Fibonacci Heap, proposed by Kaplan, Tarjan, and Zwick in 2002 [KTZ02], builds on ideas from the original Fibonacci heap while enforcing worst-case performance. Despite their optimal theoretical guarantees, both structures are known for their extreme implementation complexity and poor practical performance. As such, they are seldom used in practice and are not included in this work.

The following table summarizes the asymptotic time complexity of priority queue implementations mentioned above. A dash ($-$) indicates that a structure does not support the corresponding operation efficiently. An asterisk (*) denotes amortized complexity.

Structure	`push`	`pop`	`peek`	`decreaseKey`	`meld`
Unsorted array/list	$O(1)$	$O(n)$	$O(n)$	-	-
Sorted array/list	$O(n)$	$O(1)$	$O(1)$	-	-
Binary heap	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$	-
Fibonacci heap	$O(1)^*$	$O(\log n)^*$	$O(1)$	$O(1)^*$	$O(1)^*$
Brodal/Strict Fibonacci	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

Table 1.1: Time complexities of classical priority queue implementations.

1.2.2 Our implementations

We begin by describing several priority queue data structures that operate under a specific assumption: the key set consists of integers drawn from a bounded universe. We refer to these as *integer priority queues*. Their performance, both in time and space, depends on the size M of this universe. The table below summarizes the complexity of various integer priority queue implementations. Since none of the listed structures efficiently support `decreaseKey` or `meld`, we omit those operations from the table.

Structure	`push`	`pop`	`peek`	`space`
van Emde Boas tree	$O(\log\log M)$	$O(\log\log M)$	$O(1)$	$O(M)$
X-fast trie	$O(\log M)$	$O(\log M)$	$O(1)$	$O(n \log M)$
Y-fast trie	$O(\log\log M)$	$O(\log\log M)$	$O(1)$	$O(n)$

Table 1.2: Time and space complexities of integer priority queue implementations.

In a later chapter, we describe several *general-purpose priority queues*. These structures support arbitrary key sets and can be viewed as alternatives to classical implementations. Their complexities are summarized in Table 1.3.

Structure	`push`	`pop`	`peek`	`decreaseKey`	`meld`
Weak heap	$O(\log n)$	$O(\log n)$	$O(1)$	-	-
Skew heap	$O(\log n)^*$	$O(\log n)^*$	$O(1)$	$O(\log n)^*$	$O(\log n)^*$
Rank-pairing heap	$O(1)$	$O(\log n)^*$	$O(1)$	$O(1)^*$	$O(1)$

Table 1.3: Time complexities of implemented general-purpose priority queues.

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:

```cpp
template <class Key, class Compare = std::less<Key>>
class VanEmdeBoasTree : public PriorityQueue<Key, Compare> {
    const Key universeSize;
    const Key clusterSize;
    std::optional<Key> 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:

```cpp
VanEmdeBoasTree::VanEmdeBoasTree(Key universeSize)
    : universeSize(universeSize),
      clusterSize(static_cast<Key>(std::ceil(std::sqrt(universeSize)))),
      minValue(std::nullopt),
      maxValue(std::nullopt) {
    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:

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

We prove by induction on $u \geq u_0 = 2$. Define $c = S(u_0) + 1$ and $d = S(u_0) + 2$. The induction hypothesis is $S(u) \leq cu - d$.

Base case: For $u = 2$, by definition $S(2) = c \cdot 2 - d$.

Inductive step: Assume the hypothesis holds for all sizes $\leq u$. Then

$$S(u) = (1+\sqrt{u})S(\sqrt{u}) + \sqrt{u} \leq (1+\sqrt{u})(c\sqrt{u} - d) + \sqrt{u} = cu - d + (c - d + 1)\sqrt{u}.$$

By definiton, $c - d + 1 \leq 0$, and thus $S(u) \leq cu - d$. This completes the induction. □

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.

```
Key high(Key x) const {
    return x / clusterSize;
}

Key low(Key x) const {
    return x % clusterSize;
}

Key index(Key high, Key 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(Key x) {
    if (empty()) {
        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 minimum or maximum directly. Otherwise, we recursively insert the position into the subcluster.

```
    if (universeSize > 2) {
            Key clusterIndex = high(x), 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);
                auto &value = clusters[clusterIndex].value();
                value->minValue = 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 remove(Key x) {
if (empty()) {
    return;
}

if (*minValue == *maxValue) {
    minValue = maxValue = std::nullopt;
    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.

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

        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.

```cpp
    else {
        Key clusterIndex = high(x);
        Key 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.

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

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 bits of the key. 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 does not 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 does not 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 $\lceil \log_2 M \rceil$, which is the maximum length of a binary representation of an integer from the universe, 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, which can be achieved e.g. using any data structure that supports perfect hashing. □

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` do not 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. Given a key x, `predecessor` returns x if it exists in the structure; otherwise, it returns the greatest element less than x, or a null pointer if no such element exists.

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 first unmatched bit of the key is 1, we retrieve the largest key in the left subtree of the node 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] : nullptr);
}
```

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. 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 (short for Binary Search Tree) 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 [SA96] are a natural candidate, as they are well-known to support both operations in logarithmic expected time. Since each tree stores $O(\log M)$ elements, `split` and `merge` will take $O(\log \log M)$ expected 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 is 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 takes $O(\log\log M)$ 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 is 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. Before calling `remove`, the neighboring bucket we merge with has a size in the range $[(\log M + 1)/2,\ 2 \cdot \log M]$, due to the invariant holding previously. If the merged bucket stays within this range, the invariant still holds. If not, its size falls between $[2 \cdot \log M,\ 2 \cdot \log M + (\log M + 1)/2 - 1]$. In this case, we split it into two nearly equal parts to 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.

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 $2k + \mathrm{flip}[k]$ as its left child, and to the element at $2k + 1 - \mathrm{flip}[k]$ 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.

The central operation in our analysis is the `join` operation. Given a node j and its distinguished ancestor i, `join` restores the weak heap property between the subtrees rooted at i and j; that is, it ensures that the node j is not smaller than its distinguished ancestor i. 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 swap j with its ancestor i and then exchange the children of the lower root j. 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.

```cpp
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.

```cpp
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 Skew heap

The skew heap is a priority queue implemented as a binary tree, where all operations are based on a self-adjusting merge procedure. It was introduced by Robert Tarjan and Andrew Sleator in their 1986 paper *Self-adjusting heaps* [ST86]. Skew heaps are notable for their remarkably simple implementation and their ability to efficiently merge two heaps in $O(\log n)$ amortized time. What makes them particularly interesting from the analysis standpoint is that, despite having no structural constraints, their efficiency can be proven through a concise and elegant amortized analysis.

3.2.1 Structure and merge operation

The basic structure of the `SkewHeap` class is identical to that of a standard binary tree, so we omit its full definition here.

As noted earlier, the most important and complex operation on a skew heap is the `merge` operation. Given the roots of two skew heaps, denoted `a` and `b`, `merge` produces a new skew heap rooted at the smaller of the two (we continue to assume a *min*-priority queue). At the same time, it rearranges the subtrees to maintain the efficiency of subsequent operations.

Assuming that `a` is the smaller of the two roots, we recursively merge `a`'s right subtree with the heap rooted at `b`, and set the result as `a`'s new right subtree. Finally, we swap `a`'s left and right subtrees to complete the merge.

```
Node* SkewHeap::merge(Node* a, Node* b) {
    if (!a) return b;
    if (!b) return a;
    if (comp_(b->key, a->key)) {
        std::swap(a, b);
    }
    a->right = merge(a->right, b);
    std::swap(a->left, a->right);
    return a;
}
```

A more intuitive way to visualize how `merge` works is as follows: we first merge the rightmost paths of the two input heaps into a single right-leaning path with increasing keys. Along this path, each node retains its original left subtree unchanged. Once this combined path is formed, we simply swap the children at each node.

The running time of `merge` is proportional to the length of this merged path, as a constant amount of work is performed at each node. To guarantee that this operation is sufficiently efficient for our purposes, we will prove the following lemma using amortized analysis and the potential method:

Lemma 3.2.1. The `merge` operation has $O(\log n)$ amortized complexity.

Proof. Let $W(x)$ denote the size of the subtree rooted at node x, including x itself. We refer to a non-root node x as *heavy* if $W(x) > W(\text{parent}(x))/2$, and as *light* otherwise. Naturally, at most one child of any node can be heavy, since a heavy child's subtree accounts for more than half of the parent's subtree.

A key fact we will use extensively is that any downward path in a binary tree of n nodes contains at most $\lfloor \log n \rfloor$ light nodes. This follows because, for any path from node x to node y containing k light nodes, the subtree sizes satisfy $W(y) \leq W(x)/2^k$, as each light node contributes at most a halving of its parent's size. Therefore, $2^k \leq W(x)/W(y)$, which gives $k \leq \log(W(x)/W(y)) \leq \log n$.

We define the *potential* of our data structure as the number of heavy nodes that are right children. Initially, before any operations are performed, this potential is zero. It remains non-negative at all times and is bounded above by $n-1$ for a skew heap with n nodes.

Now consider performing a `merge` operation on two heaps h_1 and h_2, containing n_1 and n_2 elements, respectively, and let $n = n_1 + n_2$. The number of light nodes along the rightmost paths of h_1 and h_2 is bounded by $\lfloor \log n_1 \rfloor$ and $\lfloor \log n_2 \rfloor$, respectively, so the total number of light nodes on these paths is at most $2\lfloor \log n \rfloor$.

Let k_1 and k_2 denote the number of heavy nodes on the merged path originating from each of the two heaps. We can notice that all of those nodes become left children after the merge. Further, let k_3 be the number of nodes that become right heavy children as a result of the merge. Each node counted in k_3 corresponds to a light node on the merge path. Specifically, this corresponding node is its light sibling in the original heap. If the child of the last node on the merge path has no sibling, we include the root itself in the count. From our earlier derivations, it follows that $k_3 \leq \lfloor \log n \rfloor$.

Additionally, both roots of the original heaps participate in the merge path, contributing two more nodes to its length. With this setup in place, we can now bound the amortized time of `merge`. The total number of nodes on the merge path is at most

$$2 + 2\lfloor \log n \rfloor + k_1 + k_2.$$

The change in potential is given by

$$k_3 - (k_1 + k_2) \leq \lfloor \log n \rfloor - k_1 - k_2.$$

Thus, the amortized time of `merge` is bounded by

$$2 + 2\lfloor \log n \rfloor + k_1 + k_2 + \lfloor \log n \rfloor - k_1 - k_2 = 3\lfloor \log n \rfloor + 2 = O(\log n).$$

□

3.2.2 Push and pop operations

With the `merge` operation thoroughly understood, implementing `push` and `pop` becomes remarkably straightforward, requiring only a few lines of code. Their analysis is similarly simple, as each operation involves a single call to `merge` along with a constant amount of additional work.

```
void SkewHeap::push(const Key& key) override {
    Node* new_node = new Node(key);
    root = merge(root, new_node);
}
```

The `push` operation creates a new node and merges it with the `root`. Since it only requires a single call to `merge`, its amortized time complexity is $O(\log n)$.

```
SkewHeap::Key pop() override {
    if (empty()) {
        throw std::runtime_error("Priority queue is empty");
    }
    Key top = root->key;
    Node* old_root = root;
    root = merge(root->left, root->right);
    delete old_root;
    return top;
}
```

The `pop` operation first retrieves the value stored at the root, which will be returned. It then merges the root's left and right subtrees to form a single skew heap. As before, the single call to `merge` ensures an amortized time complexity of $O(\log n)$.

3.3 Rank-pairing heap

The *rank-pairing heap* is a priority queue data structure introduced by Bernhard Haeupler, Siddhartha Sen, and Robert Tarjan in their 2011 paper, *Rank-Pairing Heaps* [HST11]. Combining ideas from earlier structures such as binomial heaps, pairing heaps, and Fibonacci heaps, rank-pairing heaps were designed to offer a *simpler implementation* while matching the asymptotic time complexities of Fibonacci heaps.

The structure supports $O(1)$ worst-case time for `push`, `peek`, and `merge` operations, along with an impressive $O(1)$ *amortized* time for `decreaseKey` (which decreases the value stored at a given node, assuming direct access to it). The `pop` operation has an amortized complexity of $O(\log n)$.

A key concept underlying the data structure is the notion of *node ranks* (formally defined in a later section), which plays a central role in the analysis of its amortized time bounds. In this chapter, we focus specifically on *Type-2 rank-pairing heaps* as described in the original paper. Type-2 heaps slightly relax the rank rules used in Type-1 heaps, making them significantly easier to analyze while maintaining the same asymptotic time complexities.

3.3.1 Definitions

We first introduce several definitions to simplify the description and analysis of rank-pairing heaps.

A rank-pairing heap is represented as a collection of binary trees. Similar to the representation introduced for weak heaps (with child nodes swapped), each tree corresponds to a multi-way tree using the left-child, right-sibling convention.

In this representation, the left child of a node corresponds to its first child in the multi-way tree, and the right child corresponds to its next sibling. Since the root node of a multi-way tree has no siblings, its representation as a binary tree has an empty right subtree. Each tree is heap-ordered in the multi-way tree representation: the key of any node is smaller than the keys of all nodes in its left subtree.

The *right spine* of a node is the path starting at that node and following right-child pointers repeatedly. We also require a way to link two trees in $O(1)$ time while preserving the heap property of the multi-way representation. To link two trees rooted at nodes a and b, we first compare their keys. Assuming a has the smaller key (we again focus on *min*-priority queues), we detach the left subtree of a and make it the right subtree of b. We then attach the resulting tree rooted at b as the new left subtree of a.

Let $p(x)$ denote the parent of node x, and let $r(x)$ denote its rank. We say that the rank of a missing child is -1. For a non-root node x, we define its rank difference by $\Delta r(x) = r(p(x)) - r(x)$. A child node with rank difference i is called an i-child. A root with a left i-child is an i-node. A node whose two children are i- and j-children, is called an i, j-node. These definitions apply even if one or both children are missing, and they do not distinguish between the left and right children.

We define the *rank rule* as follows: every root is a 1-node, and every child is either a $1, 1$-node, a $1, 2$-node, or a $0, i$-node for some $i > 1$. A rank-pairing heap is a set of trees whose nodes obey this rule.

Let F_k denote the kth Fibonacci number, and let $\varphi = (1 + \sqrt{5})/2$ denote the golden ratio. We now prove a very useful lemma bounding the maximum rank of a node in a rank-pairing heap.

Lemma 3.3.1. *In a rank-pairing heap, every node of rank k has at least φ^k descendants, including itself. Therefore, $k \leq \log_\varphi n$.*

Proof. We use the inequalities $F_{k+3} - 1 \geq F_{k+2} \geq \varphi^k$, both of which can be proved by a simple induction. The proof proceeds by induction on the height of a node. A leaf has rank 0 and exactly one descendant (the leaf itself), so the base case holds: $1 = F_3 - 1 = 1$. Now assume the statement holds for all proper descendants of a node x of rank k. If x is a $0, i$-node, then its 0-child has at least $F_{k+3} - 1$ descendants by the induction hypothesis, and therefore so does x. If x is a $1, 1$-node or a $1, 2$-node, then by the induction hypothesis it has at least

$$(F_{k+1} - 1) + (F_{k+2} - 1) + 1 = F_{k+3} - 1$$

descendants, completing the induction. □

We will later use the potential method to analyze the amortized cost of both `pop` and `decreaseKey`. The potential of a tree is the sum of the potentials of its nodes. The potential of a node is the sum of its *base potential* and *extra potential*. The base potential of a node is the sum of the rank differences of its children, minus one. The extra potential of a node is one for a root, minus one for a $1,1$-node, and zero otherwise. In total, the potential of a node is zero for a $1,1$-node, one for a root, two for a $1,2$-node, and $i-1$ for a $0,i$-node.

3.3.2 Structure

With the necessary terminology in place, we can now outline the structure of the `RankPairingHeap` class. Each heap node is defined as follows:

```
struct RankPairingHeap::Node {
    Key key;
    int rank;
    Node* next;
    Node* left;
    Node* right;
    Node* parent;
}
```

By default, a node has rank 0 and is initialized with a key value. The roots of the individual trees in a rank-pairing heap form a circular linked list, where each node points to its successor via a `next` pointer. In addition, we define several helper functions to operate on nodes:

```
void RankPairingHeap::Node::getChildren(std::vector<Node*>& result) {
    result.push_back(this);
    if (left) left->getChildren(result);
    if (right) right->getChildren(result);
}

void RankPairingHeap::Node::linkLists(Node* other) {
    Node* nextNode = this->next;
    this->next = other->next;
    other->next = nextNode;
}
```

The `getChildren` function performs an in-order traversal of the subtree rooted at a given node and returns a list of all visited nodes. The `linkLists` function merges two circular linked lists of nodes into a single circular list.

The overall structure of the `RankPairingHeap` class is straightforward:

```cpp
class RankParingHeap : public PriorityQueue<Key, Compare> {
    Node* firstNode;
    unsigned size;
    Compare comp;
};
```

It maintains a single pointer, `firstNode`, which always refers to the node with the minimum key. This node also serves as the logical "start" of the circular list of tree roots.

3.3.3 Push and merge operations

The `push` operation inserts a new element into the rank-pairing heap by creating a node and appending it to the circular linked list that represents the root list.

```cpp
void RankPairingHeap::push(const Key& key) override {
    Node* newNode = new Node(key);
    size++;
    addNodeToRootList(newNode);
}
```

This insertion is managed by the helper function `addNodeToRootList`, which performs the necessary pointer updates to maintain the circular structure.

```cpp
void RankPairingHeap::addNodeToRootList(Node* node) {
    if (empty()) {
        firstNode = node;
        firstNode->next = firstNode;
        return;
    }

    node->next = firstNode->next;
    firstNode->next = node;

    if (comp(node->key, firstNode->key)) {
        firstNode = node;
    }
}
```

If the heap is empty, the `firstNode` pointer is set to the new node, and the node's `next` pointer is initialized to point to itself, forming a singleton circular list. Otherwise, the new node is inserted immediately after `firstNode` within the circular root list. Following insertion, the minimum pointer is updated if the new node's key is smaller. Each of these steps runs in constant time, so the overall time complexity of `push` is $O(1)$.

The `meld` operation combines two rank-pairing heaps:

```
void RankPairingHeap::meld(RankPairingHeap& other) {
    if (&other == this) return;
    if (other.empty()) return;

    if (empty()) {
        firstNode = other.firstNode;
        size = other.size;
        other.firstNode = nullptr;
        other.size = 0;
        return;
    }
```

The merging process begins by addressing edge cases. If the two heaps refer to the same object, the operation exits immediately. If one of the heaps is empty, the result is simply the other, non-empty heap.

```
    firstNode->next->linkLists(other.firstNode);

    if (comp(other.firstNode->key, firstNode->key)) {
        firstNode = other.firstNode;
    }

    size += other.size;
    other.firstNode = nullptr;
    other.size = 0;
}
```

In the general case, where both heaps contain nodes, their circular root lists are merged, and the `firstNode` pointer is updated to point to the node with the smaller key. As all steps involve only pointer manipulation, the total time complexity of `meld` remains $O(1)$.

3.3.4 Pop operation

The `pop` operation removes and returns the node with the smallest key from the heap.

To support this, the helper method `addRightSpine` traverses the right spine starting at a given node and adds each encountered node to the circular list of roots.

```
void RankPairingHeap::addRightSpine(Node* node) {
    while (node) {
        node->parent = nullptr;
        Node* right = node->right;
        node->right = nullptr;

        node->next = firstNode->next;
```

```
        firstNode->next = node;

        node = right;
    }
}
```

To maintain structural invariants, we must link all trees in the rank-pairing heap that share the same rank. This is achieved through the `consolidateBuckets` method.

```
std::vector<Node*> RankPairingHeap::consolidateBuckets(
    Node* root,
    unsigned nodeCount
) {
    std::vector<Node*> result;

    if (root->next == root) {
        return result;
    }

    constexpr double PHI = (1.0 + std::sqrt(5.0)) / 2.0;
    int maxRank = static_cast<int>(std::log(nodeCount) / std::log(PHI)) + 1;
    std::vector<Node*> buckets(maxRank, nullptr);
```

We begin by creating a list of empty buckets, with the size determined by the theoretical maximum rank established in Lemma 3.3.1.

```
    auto current = root->next;

    buckets[current->rank] = current;
    current = current->next;

    while (current != root) {
        auto next = current->next;
        auto rank = current->rank;

        if (buckets[rank]) {
            auto merged = linkNodes(current, buckets[rank], comp);
            buckets[rank] = nullptr;
            result.push_back(merged);
        } else {
            buckets[rank] = current;
        }

        current = next;
```

```
    }

    for (auto node : buckets) {
        if (node) result.push_back(node);
    }

    return result;
}
```

Next, we iterate over the root list, merging trees whenever two nodes share the same rank. Once consolidation is complete, we return a new set of roots, each with a distinct rank.

With this infrastructure in place, we can now implement `pop` for rank-pairing heaps:

```
Key pop() override {
    if (empty()) throw std::runtime_error("Heap is empty!");

    Node* minNode = firstNode;
    Key minKey = minNode->key;

    if (size == 1) {
        delete firstNode;
        firstNode = nullptr;
        size = 0;
        return minKey;
    }
```

If the heap contains only a single node, that node is deleted, `firstNode` is set to `nullptr`, and the corresponding key is returned immediately.

```
    addRightSpine(firstNode->left);

    std::vector<Node*> consolidatedRoots =
        consolidateBuckets(firstNode, size);

    firstNode = nullptr;
    size--;

    push(consolidatedRoots);

    delete minNode;
    return minKey;
}
```

If the heap contains only a single node, it is deleted, `firstNode` is set to `nullptr`, and its key is returned immediately.

In the general case, when multiple nodes are present, the algorithm extracts the children (in the multi-way representation) of the node being removed and adds them to the root list. It then consolidates trees with matching ranks, updates the root list, and finally deletes the removed node.

Lemma 3.3.2. The amortized running time of `pop` is $O(\log n)$.

Proof. Each new root created by disassembling the right spine, starting at the left child of the deleted node, has at least one unit of potential unless it was already a $1,1$-node. For each rank less than that of the deleted root, at most one $1,1$-node can become a new root, since each subsequent $1,1$-node has a strictly smaller rank, and the spine begins with a node of rank lower than the root's. By Lemma 3.3.1, there are at most $\log_\varphi n$ such $1,1$-nodes, so the increase in potential from disassembling the spine is bounded by $\log_\varphi n$.

Let h denote the number of trees produced after disassembly. The entire deletion of the minimum takes $O(h)$ time. If we scale time such that the processing of a single tree costs $1/2$ unit, the total time becomes $h/2 + O(1)$ units.

Each link operation during consolidation reduces the potential, as it transforms one of the roots into a $1,1$-node. Because the number of unique ranks is at most $\log_\varphi n+1$, at most that many trees do not participate in any link. Thus, there are at least $(h-(\log_\varphi n+1))/2$ links, each decreasing the potential.

Consequently, the change in potential is at most $O(\log n) - h/2$ units, while the actual work is bounded by $h/2+O(1)$ units, yielding an amortized time complexity of $O(\log n)$. □

3.3.5 DecreaseKey operation

The `decreaseKey` operation decreases the key of a given node, assuming direct access to that node is available.

To perform this operation correctly, we must be able to detach a node x from its parent, thereby promoting it to a new root. This functionality is provided by the helper function `detachFromParent`:

```
void RankPairingHeap::detachFromParent(Node* node, Node* parent) {
    if (parent->left == node) {
        parent->left = node->right;
    } else if (parent->right == node) {
        parent->right = node->right;
    }

    if (node->right) {
        node->right->parent = parent;
    }
```

```
    node->right = nullptr;
    node->parent = nullptr;
}
```

For x to become a new root, it must not have a right child. Detachment is performed by moving x's right subtree into x's current position in the parent's structure and severing the link from x to that subtree.

To restore the rank rule after decreasing a key, we apply the following procedure:

Rank-reduction: If a is a root, set $r(a) = r(\text{left}(a)) + 1$ and terminate. Otherwise, let b and c be the children of a, and compute:

$$k = \begin{cases} \max\{r(b), r(c)\} & \text{if } |r(b) - r(c)| > 1, \\ \max\{r(b), r(c)\} + 1 & \text{if } |r(b) - r(c)| \leq 1. \end{cases}$$

If $k \geq r(a)$, the rank-reduction process is complete. Otherwise, set $r(a) = k$ and continue the process recursively with $a = p(a)$.

This procedure is implemented precisely in the following method:

```
void RankPairingHeap::recalculateRank(Node* node) {
    while (node) {
        int oldRank = node->rank;

        if (node->parent == nullptr) {
            node->rank = getRank(node->left) + 1;
            break;
        }

        int leftRank = getRank(node->left);
        int rightRank = getRank(node->right);

        int newRank = (std::abs(leftRank - rightRank) > 1) ?
            std::max(leftRank, rightRank) :
            std::max(leftRank, rightRank) + 1;

        if (newRank >= oldRank) break;

        node->rank = newRank;
        node = node->parent;
    }
}
```

Lemma 3.3.3. The rank-reduction process restores the rank rule.

Proof. Decreasing the rank of a node a may cause only its parent $p(a)$ to violate the rank rule. If a violates the rule before a rank-reduction step, the first case of the procedure transforms it into a $0, i$-node, while the second case yields either a $1, 1$-node or a $1, 2$-node. In each case, the rule is restored at a. An inductive argument on the number of steps establishes the lemma. □

We are now ready to describe the `decreaseKey` operation. Suppose we wish to decrease the key at node x.

```
void RankPairingHeap::decreaseKey(Node* node, Key newKey) {
    node->key = newKey;
    Node* parent = node->parent;

    if (!parent) {
        if (comp(node->key, firstNode->key)) {
            firstNode = node;
        }
        return;
    }
```

If x is already a root, we simply check whether it now holds the minimum key and update `firstNode` accordingly. If x is not a root, we must promote it to one.

```
    detachFromParent(node, parent);

    push(node);

    recalculateRank(parent);
}
```

To do so, we first detach the subtree rooted at x from its parent. We then replace x's position in the tree with the subtree of its right child and insert x into the root list. If the new key at x is smaller than the current minimum, we update `firstNode` to point to it. Finally, we invoke the rank-reduction procedure at x's former parent to restore the rank rule.

We are now ready to state the key lemma establishing the amortized time complexity of `decreaseKey`, the most complex result in the analysis of rank-pairing heaps.

Lemma 3.3.4. The amortized running time of `decreaseKey` is $O(1)$.

Proof. Suppose we are decreasing the key at node x. If x is a root, the key decrease takes $O(1)$ actual time and does not change the potential. Suppose x is not a root. Let $u_0 = \text{left}(x)$, $u_1 = x$, and let $u_2, \ldots, u_k$ be such that u_j is the parent of u_{j-1}, where u_j for $2 \le j < k$ have their ranks decreased, and u_k is the final node in the rank-reduction process (either a root or a node that retains its rank). Let v_j, for $0 \le j < k$, be the sibling of u_j. Denote by r and r' the ranks before and after the key decrease.

The only nodes whose base potential changes as a result of the operation are $u_1, \dots, u_k$. We now try to bound this potential change. The expression $\Delta r(v_0) + \sum_{i=1}^{k-1} \Delta r(u_i)$ telescopes to $r(u_k) - r(v_0)$. Similarly, $\Delta r'(v_0) + \sum_{i=2}^{k-1} \Delta r'(u_i)$ (we don't include u_1 since it is a new root after the key decrease) telescopes to $r'(u_k) - r'(v_0)$. Note that $r'(u_k) - r'(v_0) \leq r(u_k) - r(v_0)$, because $r'(u_k) \leq r(u_k)$ and $r'(v_0) = r(v_0)$. Moreover, $\Delta r'(u_0) = 1 \leq \Delta r(u_0) + 1$, and $\Delta r'(v_j) \leq \Delta r(v_j) - 1$ for $0 \leq j \leq k-2$. In total, the sum of potentials of $u_1, \dots, u_k$ drops by at least $k-3$ and increases by at most 1, so it decreases by at least $k-2$ after decreasing the key.

We now bound the change in extra potentials of $u_1, \dots, u_k$. At most two of these nodes can be $1,1$-nodes before the key decrease, by the following argument: Let j be the minimum such that u_j is a $1,1$-node. Of course, it can decrease in rank by at most one to maintain non-negative rank differences. Hence, $u_{j'}$, for $j' > j$, can also decrease in rank by at most one by the rank rule. If $u_{j'}$ for $j' > j$ is a $1,1$-node, it becomes a $1,2$-node as a result of $u_{j'-1}$ decreasing in rank by one, so $u_{j'}$ does not itself decrease in rank, and the rank-reduction process ends at $u_{j'}$. All in all, the sum of extra potentials of $u_0, u_1, \dots, u_k$ increases by at most three as a result of the key decrease: at most two $1,1$-nodes become non-$1,1$-nodes, increasing the sum by two, and $u_1 = x$ becomes a root, increasing the sum by one.

We see that the key decrease operation causes the total potential to drop by at least $k-5$ units. If we scale the time of a single rank-reduction step to be 1 unit, the amortized time of the key decrease is $O(1)$. $\square$

Chapter 4

Implementation and benchmark

4.1 Implementation

All priority queues described in this work were implemented in C++. The full open-source code is available in the `koala-networkit` GitHub repository at: `https://github.com/krzysztof-turowski/koala-networkit/pull/37`. Each data structure is accompanied by unit tests to verify correctness for basic cases.

4.1.1 File structure

All priority queue implementations are header-only and organized as follows:

- `include/structures/PriorityQueue.hpp` – defines the common interface,
- `include/structures/priority_queue/VanEmdeBoasTree.hpp`,
- `include/structures/priority_queue/XFastTrie.hpp`,
- `include/structures/priority_queue/YFastTrie.hpp`,
- `include/structures/priority_queue/WeakHeap.hpp`,
- `include/structures/priority_queue/SkewHeap.hpp`,
- `include/structures/priority_queue/RankPairingHeap.hpp`.

4.2 Benchmark

All benchmarks were compiled using `g++` 13.3.0 and executed on an Intel Core i5 2.60 GHz CPU. One of the test sets uses data from the US Road System, available at `https://www.diag.uniroma1.it/~challenge9/download.shtml`.

4.2.1 Performance evaluation

To assess the performance of the various priority queues, several scenarios were considered:

- **Universe size scaling:** Time and memory usage were measured for different integer key universe sizes, using 1,000,000 randomized `push` and `pop` operations. This test was designed to evaluate how integer-based priority queues scale with increasing key space.
- **Queue size scaling:** To examine the effect of queue size, a fixed universe size of 2^{25} was used. For each test, a bulk of elements was inserted using `push` operations, followed by their removal using `pop`. Time and memory usage were measured separately for both phases.
- **String key performance:** (General-purpose queues only) This test assessed time usage when operating on long random string keys (1024 characters), with varying queue sizes. The goal was to evaluate the impact of costly key comparisons on overall performance.
- **Practical application:** (General-purpose queues only) Each queue was integrated into Dijkstra's shortest path algorithm [Dij59] and tested on real-world graphs from the 9th DIMACS Implementation Challenge. The following datasets of various sizes from the US Road System were used to compare real-world runtime performance:
  - **USA-road-d.NY.gr**: 264,346 nodes, 730,100 edges, 13.77 MB
  - **USA-road-d.FLA.gr**: 1,070,376 nodes, 2,687,902 edges, 53.07 MB
  - **USA-road-d.CAL.gr**: 1,890,815 nodes, 4,630,444 edges, 95.41 MB
  - **USA-road-d.W.gr**: 6,262,104 nodes, 15,119,284 edges, 325.84 MB
  - **USA-road-d.CTR.gr**: 14,081,816 nodes, 33,866,826 edges, 756.36 MB
  - **USA-road-d.USA.gr**: 23,947,347 nodes, 57,708,624 edges, 1322.82 MB

Below we provide results of the performance test scenarios mentioned. In all cases we also provide a baseline implementation from the C++ STL Library for comparison. The Figure 4.1 presents the common legend for all plots.

Figure 4.1: Legend assigning colors to priority queue implementations.

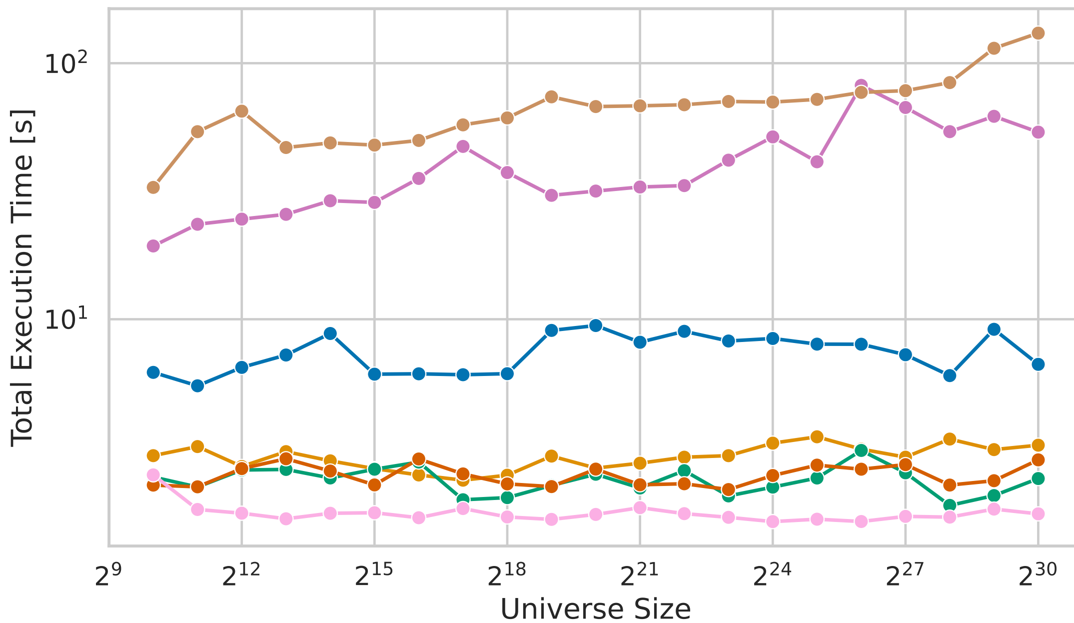

Figure 4.2: Total execution time vs. universe size.

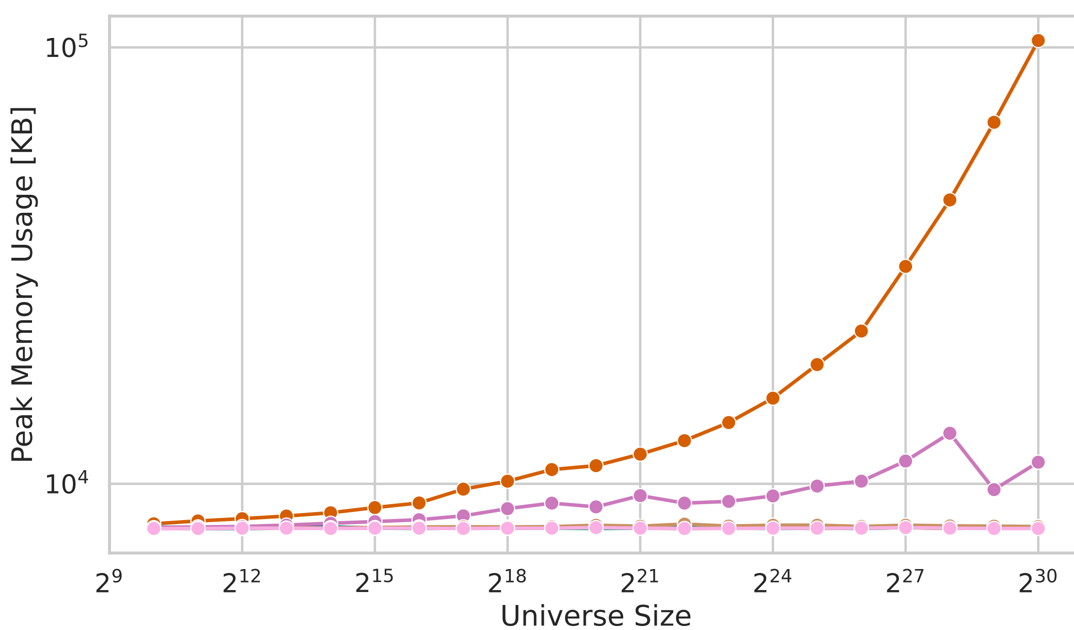

Figure 4.3: Memory usage vs. universe size.

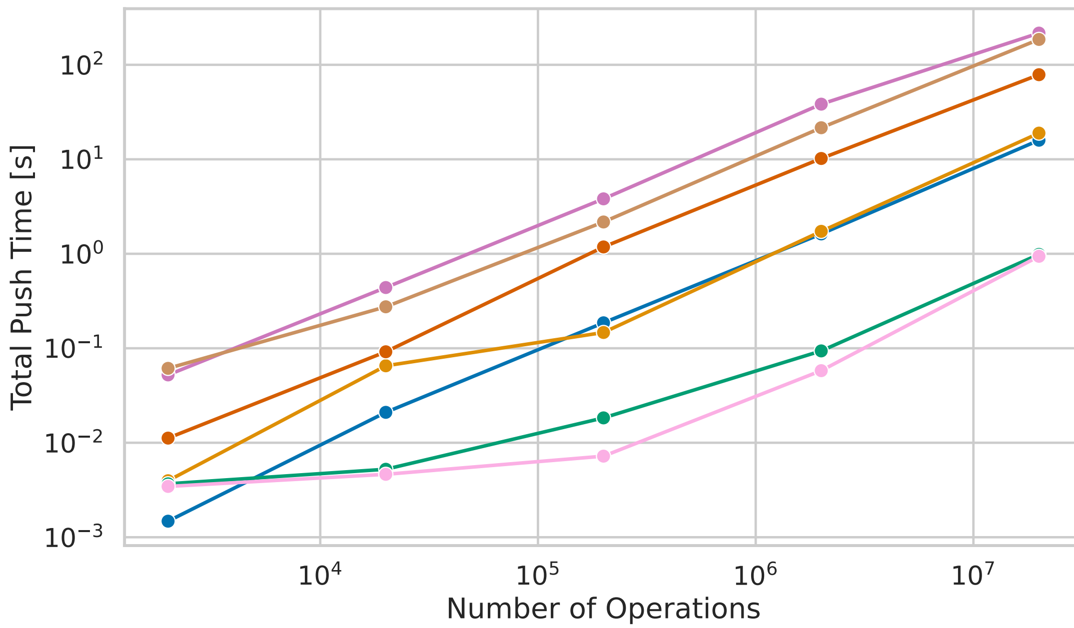

Figure 4.4: Push execution time vs. number of operations.

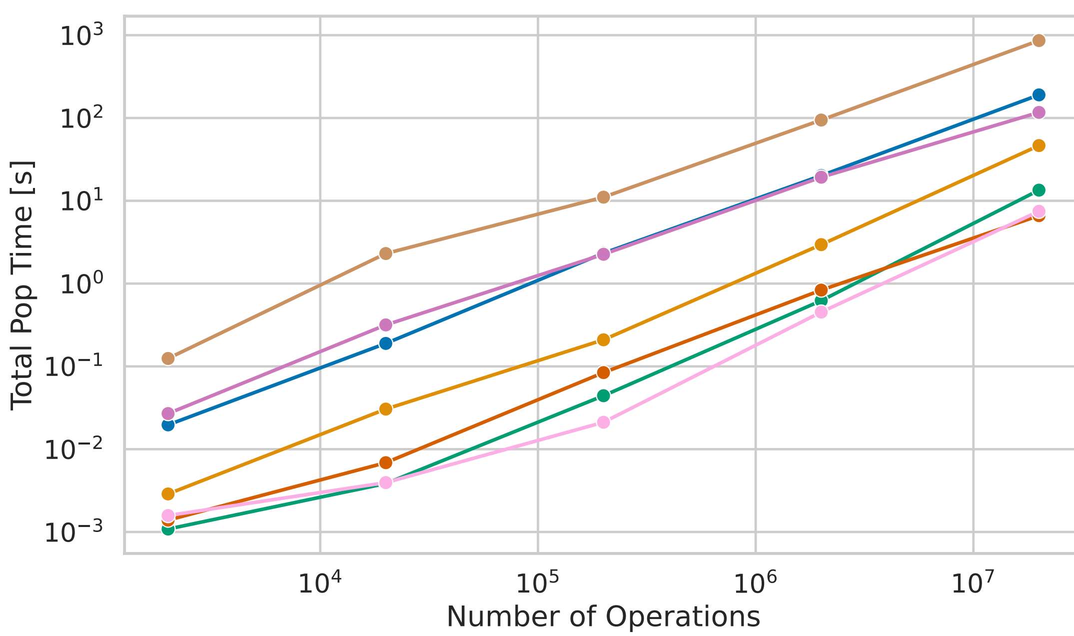

Figure 4.5: Pop execution time vs. number of operations.

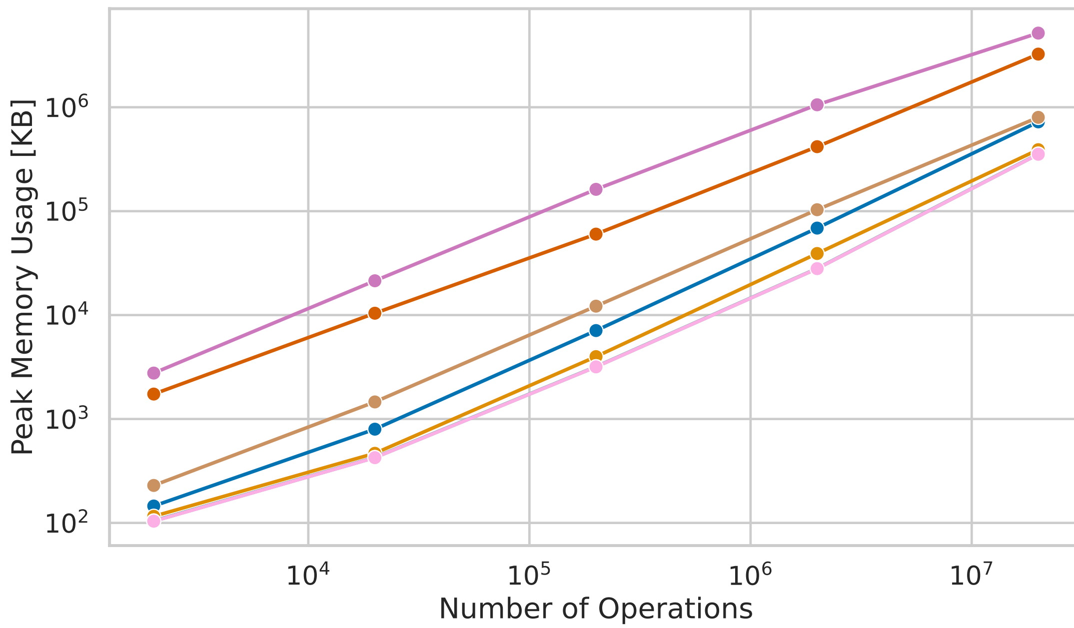

Figure 4.6: Memory usage vs. number of operations.

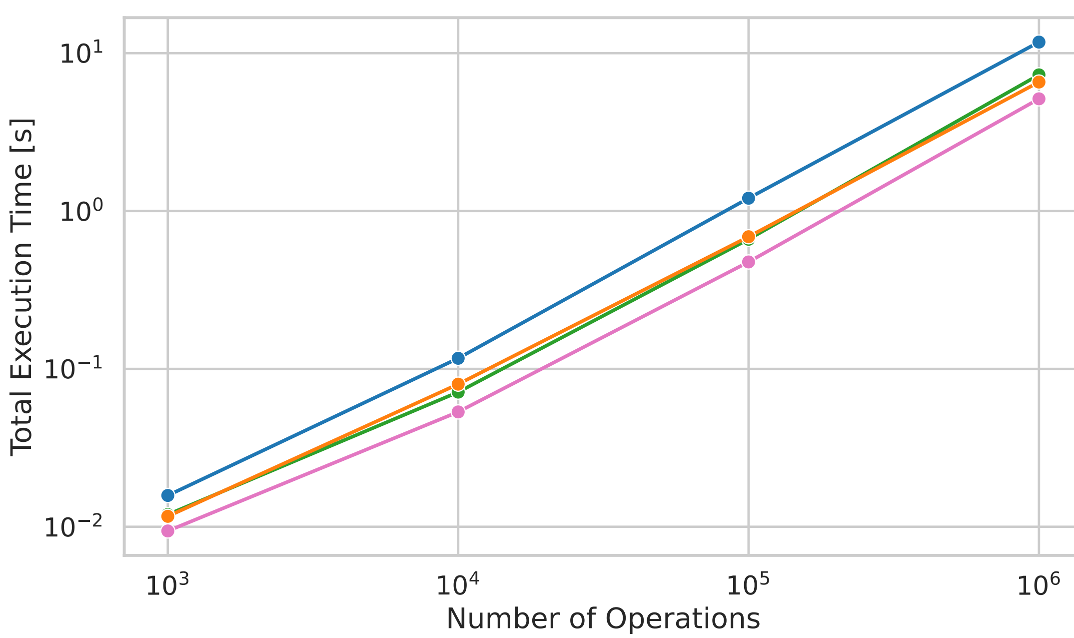

Figure 4.7: Total execution time vs number of operations on string keys.

File	Weak Heap	Skew Heap	R-p Heap	STL
USA-road-d.NY.gr	0.0578	0.0323	0.1410	0.0335
USA-road-d.FLA.gr	0.2369	0.1402	0.5291	0.1427
USA-road-d.CAL.gr	0.4981	0.2951	0.9647	0.3153
USA-road-d.W.gr	1.6977	1.0162	3.2455	1.1301
USA-road-d.CTR.gr	5.7985	4.5255	10.0441	4.3664
USA-road-d.USA.gr	7.0490	4.3979	13.3550	4.6924

Table 4.1: Benchmark results for general-purpose priority queues running Dijkstra's algorithm on USA road network graphs. Times are given in seconds. R-p heap denotes `RankPairingHeap`. STL denotes `std::priority_queue`.

4.2.2 Summary

The tests allow us to draw several interesting conclusions about the performance of different priority queue implementations. Perhaps the most notable, though expected, result is the very high memory usage of the van Emde Boas priority queue for large universe sizes, as illustrated in Figure 4.3. Aside from this, the van Emde Boas priority queue outperforms other integer priority queues, since the constant factors in the implementations of X-fast and Y-fast tries significantly reduce their practical efficiency. This is especially evident in Figure 4.2, where those priority queues perform the worst on large sets of random operations.

Among general-purpose priority queues, weak heaps emerge as a viable alternative to the standard binary heap, with performance closely matching the STL priority queue in many test cases. The skew heap, while slightly less efficient on artificial benchmarks, performs exceptionally well on real-world data, outperforming the STL priority queue on all but one test file when used in Dijkstra's algorithm, as shown in Table 4.1. Conversely, the rank-pairing heap is the least performant among general-purpose priority queues in many cases, likely due to the implementation complexity and the resulting constant-factor overhead.

Bibliography

[Bro96] Gerth Stølting Brodal. "Worst-case Efficient Priority Queues". In: *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1996), pp. 52–58.

[Cor+22] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. 4th ed. The MIT Press, 2022.

[Dij59] Edsger W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[Dut93] Ronald 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 (FOCS)*. 1975, pp. 75–84.

[FT87] Michael Fredman and Robert Tarjan. "Fibonacci heaps and Their Uses in Improved Network Optimization Algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[HST11] Bernhard Haeupler, Siddhartha Sen, and Robert Tarjan. "Rank-Pairing Heaps". In: *SIAM Journal on Computing* 40.6 (2011), pp. 1463–1485.

[KTZ02] Haim Kaplan, Robert Tarjan, and Uri Zwick. "Fibonacci Heaps Revisited". In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2002, pp. 1–10.

[SA96] Raimund Seidel and Cecilia Aragon. "Randomized Search Trees". In: *Algorithmica*. Vol. 16. 4–5. 1996, pp. 464–497.

[ST86] Daniel Sleator and Robert Tarjan. "Self-Adjusting Heaps". In: *SIAM Journal on Computing* 15.1 (1986), pp. 52–69.

[Wil64] J. W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348.

[Wil83] Dan Willard. "Log-logarithmic Worst-Case Range Queries Are Possible in Space $\Theta(N)$". In: *Information Processing Letters* 17.2 (1983), pp. 81–84.