IT UNIVERSITY OF COPENHAGEN

MSc Software Development

An Implementation of Theoretically Optimal Dynamic Rank, Select, and Predecessor Data Structure

Thesis Report

Hugo de Brito
hubr@itu.dk

1. September 2020

Supervisor
Holger Dell
hold@itu.dk

Acknowledgments

I want to start by paying special regards to my research supervisor Holger Dell. He has been a pillar in my learning, and his support and dedication have been decisive factors in the success of this project.

Martin Aumüller and Riko Jacob (from ITU's Algorithms group) deserve praise for never denying me their precious help when I needed to understand some important concepts.

My peers Weisi Li and Jonas Andersen, to whom I wish success and happiness, thank you for the support and company you offered me during this project.

My two Luísas, my mother and my grandmother, who have always been important references and provided me with the tools and opportunities that allowed me to pursue this journey, you mean the world to me.

Alex, my life partner, thank you for pushing me in to being the best version of myself all the time, but especially during this project.

Lastly, I would like to honor my dearest grandfather Augusto, who, unfortunately, has passed away amid the project. I dedicate this piece of work to him.

Abstract

This thesis investigates the theoretical data structure presented in the «Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search» paper, authored by Pătraşcu and Thorup. The authors claim that their proposal solves the dynamic predecessor problem optimally [Pătraşcu and Thorup, 2014]. Concretely, we follow the paper closely up to chapter 3.1 (inclusive), implementing the data structure and its algorithms, discussing and evaluating the running times. The data structure they propose denoted *Dynamic Fusion Node*, is made possible due to the smart use of techniques such as bitwise tricks, word-level parallelism, key compression, and wildcards (denoted "don't cares" in this report and also by Pătraşcu and Thorup), which for sets of size $n = w^{O(1)}$ and in the word-RAM model take $O(1)$ time. By using the *Dynamic Fusion Node* in the implementation of a k-ary tree, thus enabling the sets to be of arbitrary size, the running times are $O(\log_w n)$, proven by Fredman and Saks to be optimal [Fredman and Saks, 1989].

Using the [Pătraşcu and Thorup, 2014] paper as our primary reference, and after framing the topic within its context, we explore the essential tools and algorithms used by Pătraşcu and Thorup in their proposal. We will build, explore, and implement a library of relevant functions and algorithms used by the *Dynamic Fusion Node*. Based on the theoretical algorithms presented in [Pătraşcu and Thorup, 2014], the implementation is presented in iterative steps, starting from a naive way up to the insertion method while using all the algorithms and techniques described up to that point. We validate the implemented algorithms and data structures with correctness tests. Finally, we conclude the project, leaving some remarks and suggestions for future work, which include the necessary steps to make implementation $O(1)$, which at the moment is still bound by some subroutines that take more than constant time.

Up to the point where this project ends, Pătraşcu and Thorup's proposal seems sound, as we have been able to implement their data structure proposal using mostly $O(1)$ operations of the word RAM model.

The Java programming language is used throughout the project, and the resulting code is publicly available on the GitHub repository sited at `https://github.com/hugo-brito/DynamicIntegerSetsWithOptimalRankSelectPredecessorSearch` and subject to an MIT License.

Keywords: Algorithms, Dynamic Predecessor Problem, Integer Sets, Key Compression, Bitwise Operations, Most Significant Set Bit, Word-level Parallelism, Dynamic Fusion Node.

Contents

List of Tables

List of Figures

1 Introduction

Predecessor problem queries are everywhere. They can be found in several different applications, including database management systems (DBMS), internet routing, machine learning algorithms, big data, statistics, data compression, and spellchecking. They can be understood as a simpler version of the nearest neighbor problem [Bille, 2020]. Moreover, because they are at the basis of many systems, it is of great importance and relevance that the data structures that provide these queries are space and time-efficient.

This thesis investigates the theoretical data structure presented in the «Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search» paper, from 2014 and authored by Pătraşcu and Thorup. Concretely, we follow the paper closely up to chapter 3.1 (inclusive), implementing the data structure and its algorithms, discussing and evaluating the running times.

The authors claim that their proposal solves the dynamic predecessor problem optimally. Fusion Nodes compute static predecessor queries in $O(1)$ time [Fredman and Willard, 1993], but updates may entail recomputing the instance variables, making such operation to take polynomial time. It is stated that, for this reason, Fusion Trees only solve the static predecessor problem [Nelson and Liu, 2014]. Pătraşcu and Thorup address this by improving on how the keys in the set are compressed, allowing them to be computed and queried in $O(1)$ time for sets of size $n = w^{O(1)}$, e.g., the *Dynamic Fusion Node* [Pătraşcu and Thorup, 2014]. This is made possible due to the smart use of techniques such as bitwise tricks, word-level parallelism, key compression, and wildcards (denoted "don't cares" in this report and also by Pătraşcu and Thorup), which in the word RAM model take $O(1)$ time.

1.1 Context

The aforementioned paper presents a data structure called *Dynamic Fusion Node*. It is designed to maintain a dynamic set of integers, which improves on the fusion node authored by Fredman and Willard and published in [Fredman and Willard, 1993] while incorporating concepts from other data structures such as Patricia tries.

Like in the Fusion Tree by Fredman and Willard, the *Dynamic Fusion Node* is to be used in a k-ary tree implementation, thus solving the dynamic predecessor problem for sets of arbitrary size. When this goal is reached, Pătraşcu and Thorup claim that the running time for the dynamic rank and select is $O(\log_w n)$, proven by Fredman and Saks to be optimal [Fredman and Saks, 1989]. This is possible because all operations at the node

level take $O(1)$ time, making the overall running time bounded by the tree structure the node is to be implemented in.

This project can also be seen as a follow-up on the data structures discussed in the *Integer Data Structure* lectures from the «6.851: Advanced Data Structures»[1] and «CS 224: Advanced Algorithms»[2] courses from Harvard University and MIT, respectively. These courses' scribe notes have also been an important resource for the work we have developed in this project. In particular, Demaine's lecture 12 [Demaine et al., 2012b] and Nelson's lecture 2 [Nelson and Liu, 2014] of the aforementioned courses discuss the Fusion Trees, which, as previously mentioned, are data structures that establish many important concepts that the *Dynamic Fusion Tree* from Pătraşcu and Thorup builds upon. This is relevant because, in their lectures, they present the Fusion Tree as a static predecessor problem data structure, e.g., it can be efficiently queried but does not support efficient updates, whereas the present implementation is focused on supporting efficient updates as well.

1.2 Goals

Using the [Pătraşcu and Thorup, 2014] as the main literature resource, this project aims at:

- Exposing the data structure present in the aforementioned paper by implementing it as close as possible to its description.
- Reporting on the implementation, highlighting the algorithms and techniques that enable the data structure's functionality by means of illustrative examples.
- Evaluating the soundness of Pătraşcu and Thorup's claim, e.g., if it is possible to implement the data structure and its algorithms by using only $O(1)$ time operations in the word RAM model.
- Elaboration of a small survey on existing (dynamic) predecessor problem data structures, paving the way for a future benchmark between the present and the surveyed data structures.

1.3 Approach

We will be guided by the [Pătraşcu and Thorup, 2014] publication, using contributions from other authors whenever needed. The data structure will be implemented by taking incremental steps, which more or less correspond to the Chapters and Sections of our primary reference.

[1] The lectures notes, recordings, and other course materials are available through the link: `http://courses.csail.mit.edu/6.851/fall17/lectures/`

[2] The lectures notes, recordings, and other course materials are available through the link: `http://people.seas.harvard.edu/~minilek/cs224/fall14/index.html`

This project encompasses the present document and the algorithms and data structures here reported. All the code developed in the scope of this project has been made publicly available in a GitHub repository, accessible through the link `https://github.com/hugo-brito/DynamicIntegerSetsWithOptimalRankSelectPredecessorSearch` and subject to an MIT License. The programs were implemented using the Java programming language.

1.4 Scope

This project's scope is limited to:

- Implementation of helper functions that implement supporting algorithms for the *Dynamic Fusion Node* data structure.
- Implementation of the algorithms encompassed by the *Dynamic Fusion Node* as presented in [Pătraşcu and Thorup, 2014] but up to Chapter 3.1 (inclusive).

1.5 Report Structure

The present chapter provides the context, the approach taken, and establishes the scope of the project.

Chapter 2 provides the background needed to understand the problem at hand and some non-trivial techniques used by Pătraşcu and Thorup in their proposal. In Section 2.2, we will also see different ways to solve the predecessor problem, from the naive Array up until the Fusion Tree. The latter is also an essential premise for enabling the data structure of this paper.

Chapter 3 builds a library of relevant functions used for the *Dynamic Fusion Node*, which is also presented in the following sections of the chapter. Based on the theoretical algorithms presented in [Pătraşcu and Thorup, 2014], the implementation is presented in iterative steps, starting from a naive way up to the insertion method while using all the algorithms and techniques described up to that point.

We validate the implemented algorithms and data structures with correctness tests. These appear described in Chapter 4.

Chapter 5 concludes the project, leaving some remarks and suggestions for future work.

We have also included an appendix, A, where we mention some additional carried work that could be useful for future work.

1.6 Files

This project encompassed a set of files, which are handed in together with the present report in a `zip` archive. Once extracted, this `zip` archive includes the following:

Read-Me File The file named `README.md` and located in the root folder.

Written Report The file named `report.pdf` and located in the root folder.

Report Source Code Located in the `.\src\tex` folder.

Javadoc The `Javadoc` documentation can be found in the `.\src\javadoc` folder. For convenience, a shortcut was included in the root folder: `Javadoc.html`.

Implementation Source Code Located in the `.\src\main\java\integersets` folder.

Testing Classes Source Code Located in the `.\src\test\java` folder.

The remaining non-listed files include a `Gradle` build, which allows to run the testing classes for the implementations. To do so, open the terminal in the root folder and type `gradlew test`. Note that it might take several hours for the tests to conclude since the tests are currently configured to run on large instances.

2 Background

The [Pătraşcu and Thorup, 2014] paper presents a data structure for solving the dynamic predecessor problem and claims that its running times are optimal. In this chapter, a series of data structures, techniques, and other relevant aspects will be presented, laying the foundation for the implementation of the data structure presented by Pătraşcu and Thorup.

2.1 Basic Concepts

2.1.1 Word

A word consists of a w-bit integer. The bits which a word is comprised of are indexed from the least (right) to the most (left) significant bit, having the least significant bit index 0, whereas the most significant has index $w - 1$. The universe of all possible keys, denoted by $\mathcal{U}$, has size $u = 2^w$ thus we are bound to an universe $\mathcal{U} = \{0, 1, ..., 2^w - 1\}$, e.g., all combinations of w-bit words.

Since we will be implementing algorithms and data structures on a modern computer, we set $w = 64$. We might use different values of w for illustration purposes, but the implemented programs will be working with the value defined here unless stated otherwise.

2.1.2 The Predecessor Problem

Data structures that maintain a set S of (integer) keys and enable the following operations are said to solve the static predecessor problem [Beame and Fich, 1999]:

- member(x) returns `true` if $x \in S$ and `false` otherwise.
- predecessor(x) returns $\max\{y \in S \mid y < x\}$.
- successor(x) returns $\min\{y \in S \mid y \geq x\}$.

The predecessor problem can also be dynamic if the said data structure also allows [Beame and Fich, 1999]:

- insert(x) sets $S = S \cup \{x\}$.

- delete(x) sets $S = S \setminus \{x\}$.

In this context, the following operations might also be relevant:

- rank(x) returns $\#\{y \in S \mid y < x\}$.
- select(i) returns $y \in S$ with rank(y) $= i$, if any.

The data structure presented in [Pătraşcu and Thorup, 2014] implements all of the above. The publication also mentions the following invariants:

- predecessor(x) = select(rank($x - 1$))
- successor(x) = select(rank(x))

2.1.3 Models of Computation

In order to analyze and describe running times, computer scientists use models of computation. Each model states which operations have an associated cost and which ones do not. In the models used here, all the given operations either have one unit of cost or none.

Despite their theoretical relevance, when having a data structure and its operations measured against wall clock, one might be surprised with the results. This is because theoretical bounds have the potential to hide big constants, which are brought to light when wall clock measurements are performed. Nevertheless, we will enumerate some models of computation, as many of the data structures here presented have their running times described in terms of a given model, and therefore, it is of interest to provide this context.

The models are presented in descending order from strongest to the least strong. This means that a less restrictive model, such as the cell-probe model, is more suited to describe the theoretical lower bounds of a data structure than the weaker ones [Demaine et al., 2012a].

2.1.3.1 The Cell-Probe Model

In the cell-probe model, memory is divided into cells of size w, a parameter of the model. The only operations that come with an associated cost are reading or writing to memory, which are the memory accesses. Due to its simplicity, as stated in 2.1.3, it is widely used to prove lower bounds [Demaine et al., 2012a].

2.1.3.2 Trans-dichotomous RAM

In the trans-dichotomous RAM model, memory consists of an array of size S of w-bit words. Reading or writing to one of the memory cells costs $O(1)$. Additionally, memory

cells can be used as pointers to other cells, e.g., a single w-bit word can be used to access another cell. This implies that the word length w has to be large enough to be able to index all cells in the memory. Let the problem size be n:

$$w \geq \log_2(S) \implies w \geq \log_2(n) \tag{2.1}$$

Let us take a concrete example:

$$\frac{w = 4}{\begin{aligned} & w = 4 \geq \log_2(n) \\ \implies & 2^4 \geq n \\ \implies & 16 \geq n \end{aligned}}$$

We can see that if $w = 4$, then we can at most index 16 keys. This is sound because in binary representation of integers, with 4 bits, we can produce 16 different combinations of those bits, e.g., 2^4 combinations. This is exactly what expression 2.1 means: having a set word length defines the maximum number of keys we can index with a single word.

This model gets its name because it relates two dichotomies: problem size n; and the model of computation with words of size w [Demaine et al., 2012a].

2.1.3.3 Word RAM

Like the trans-dichotomous RAM model, the word RAM also operates with fixed size w-bit words. Additionally, the following operations have an associated cost [Nelson and Schorow, 2013]:

- Integer arithmetic (addition $+$, subtraction $-$, multiplication $\times$, division $\div$ and remainder of division (modulo) mod);
- Bitwise operations (negation $\neg$, and $\wedge$, or $\vee$, exclusive or $\oplus$);
- Bitwise shifting operations (right bit-shift $\gg$, left bit-shift $\ll$).

2.2 Predecessor Problem Data Structures

This section highlights some data structures that maintain integer sets and implement the (dynamic) predecessor problem, e.g., integer sets whose methods answer the queries mentioned in Section 2.1.2. The data structures are ordered by what they improve on from the previous, or by complexity because they add a new way of looking at the problem. We will also analyze, on a high level, their running times, and how the queries can be implemented. Note that we do not need to be concerned about the successor and predecessor queries for most of them, as they can be trivially implemented by using the definitions from Section 2.1.2.

2.2.1 Array

Perhaps the most naive way to maintain a set of integers is to implement it while having an underlying array. By ensuring that the array is always sorted after updates, we know that select queries are implemented by returning the key at the specified index, while rank queries are implemented by doing a binary search. These queries take $O(1)$ and $O(\log_2 n)$ time, respectively. Updating, e.g., inserting and deleting, would take $O(n)$ time because:

1. We would first need to find the rank of the key to be inserted (or deleted), i.

2. And then, all the keys whose rank is larger than i would have to be moved by one position in the array. For insertion, those keys would have to be moved to the right; for deletion, those keys would have to be moved to the left.

2.2.2 Red-Black BST

Using a self-balancing Binary Search Tree to implement the set, such as a red-black tree, could improve the running times quite substantially. This data structure guarantees $O(\log_2 n)$ for all of the dynamic predecessor problem queries. When searching, this guarantee is given by the height invariant of the tree, which is always $O(\log_2 n)$ [Cormen et al., 2009]. Updates consist of:

1. Searching, which takes $O(\log_2 n)$ time.

2. Once the rightful place of the key in the tree is found, a $O(1)$ number of operations take place in order to maintain the invariant of the tree.

Another advantage of this data structure is that, since the elements are sorted internally, select and rank queries can be trivially implemented by storing the size of each subtree at every node, which is then used to compute the result from those queries.

2.2.3 Binary Search Tries

The previous two data structures use comparison-based algorithms to order the keys internally, e.g., searching entails comparing the keys' full length. Radix algorithms take a different approach by examining portions of the key. Tries are radix-based data structures where the path from the root to a particular node is the prefix of the keys stored at the subtree rooted at that node. In this context, we will use them to store words (as defined in Section 2.1.1), but they can be used to store other types of data, for instance, strings, which can be of fixed or variable length. The chosen radix is related to the data we want to store in the Trie: for instance, when storing strings written with the English alphabet, the radix is 26, whereas in our case, base two integers, the radix is 2. Bitwise Tries can also denote Tries of radix two.

A Binary Search Trie is a Trie for storing integer words, where the keys are kept at leaf nodes. We use the bit values of a key to guide us when searching the following way: starting from the most significant bit of the key,

- If the bit value is 0, then we take the left child node.
- Otherwise, we take the right one.

We then consider the following less significant bit of the search key and repeat until the search ends. This happens when either we hit a leaf or a null link. A successful search is when the search key and the key at the leaf node were the same. An unsuccessful search is when either we end up on a null link or the search key, and the key at the leaf node are not the same. This last case happens when those keys share a prefix.

We can infer from the search algorithm's description that this data structure uses a considerable amount of space for storing paths consisting of prefixes shared among keys.

Since this tree is ordered and balanced, and even though it will have on average 44% more nodes than keys, searching and inserting take on average $O(\log_2 n)$ time for random and distinct keys, but the length of the keys bounds the worst-case running time, $O(w)$ [Sedgewick, 2002]. Rank and select queries are implemented with a similar approach as in the red-black tree: every node has the number of leaf nodes stored in its fields, which is used for these queries.

2.2.4 Patricia Tries

Patricia Tries, which name is an acronym for "practical algorithm to retrieve information coded in alphanumeric", also known as compressed Binary Tries, improve the space consumption in comparison with Binary Search Tries [Sedgewick, 2002]. They embody a different way to encode the same Trie abstraction as Binary Search Tries. They achieve this by allowing the internal nodes to store keys and by compressing the paths between nodes. Each node will have an associated branching bit stored, allowing us to skip the bits that belong to shared prefixes, fast-forwarding to the branching bits. The result is a Trie

with exactly as many nodes as there are keys, which also improves the worst running times of updating and querying from $O(w)$ to $O(\log_2 n)$ [Sedgewick, 2002].

2.2.5 van Emde Boas Trees

The van Emde Boas tree (vEB) data structure, despite its excessive space consumption ($O(u)$), introduces a big running time improvement, as all queries take $O(\log_2 w)$. If, as discussed in Section 2.1.1, $u = 2^w$ is the size of the universe of keys we can store in this data structure, then we can also express the running time as being $O(\log_2(\log_2 u))$. This means that queries are now sub-logarithmic, a substantial improvement from all the data structures we have seen so far. It uses the fact that the keys themselves can be used as memory addresses to access other keys [Nelson and Schorow, 2013].

The intuition behind this data structure is to superimpose a Binary Tree structure on a bit-vector. The bit values in this bit-vector will be 0 if the key at that index is not in the set, and 1 otherwise. To build the Binary Tree, we assume that each position of the bit-vector is a leaf. To compute the parent nodes of the leaf nodes, we take the bitwise $\vee$ of each consecutive pair of positions. We then do the same for the next level, taking the nodes that we just computed as the children and repeating until we reach the root. If the set contains at least one key, then the value at the root will be 1; otherwise, it will be 0. This rule also applies to each of the subtrees [Bille, 2020]. On this preliminary example:

- To find the minimum, starting at the root, we traverse the tree by always taking the left child if that node is 1. Otherwise, we take the right child.
- To find the maximum, we do the same as in finding the minimum, but giving preference to the right child.
- To find the successor, we would start at the leaf node of our query and go up the tree until we find a node whose right child we have not visited and whose value is 1. The successor of our query is the minimum of the subtree rooted at that node.
- Finding the predecessor will be similar to the successor: we find the maximum of the subtree whose root's left child we have not visited and had the value 1 when going up the tree from the query's leaf node.
- Searching is done in $O(1)$ time, as we have direct access to the bit-vector.
- To update, e.g., inserting and deleting, we have to update the bit value in the bit array and the nodes above that position.

All the mentioned operations, except for searching, take $O(\log_2 u)$ time.

To improve this and achieve the sub-logarithmic running times, we superimpose a tree of varying degrees on the bit-vector. The value of the root node will be the summary of the whole range. Its children will cover $u^{1/2}$, and its grandchildren $u^{1/4}$. If at each level we store the minimum, the maximum, and a summary of the vEB trees below, we can compute queries in $O(\sqrt{w})$ time plus a constant number of operations, $O(1)$. Thus we

have the following recurrence relation: $\mathrm{T}(u) = \mathrm{T}(\sqrt{u}) + O(1)$, resulting in a running time for the whole tree of $O(\log_2(\log_2 u))$ [Demaine et al., 2012a].

As mentioned, van Emde Boas trees have the space complexity drawback: $O(u)$. This is due to the bit-vectors used, which in total allocate space for all the possible keys in the universe, $\mathcal{U}$.

2.2.6 Fusion Trees

The Fusion Trees description we summarize here is based on the [Nelson and Liu, 2014] and [Demaine et al., 2012b] lectures.

Fusion Trees are understood as k-ary trees, e.g., each node stores roughly k keys. We define $k = \Theta(w^{1/5})$. With this definition, the height of the tree will be $O(\log_w n)$. This is also the running time for querying in a Fusion Tree. So the problem we have to solve is to ensure that searching within nodes takes only $O(1)$ time. This way, the overall data structure's query time will be the same as the bound of the tree's height.

In the word RAM model, and having set $k = \Theta(w^{1/5})$, each node will require $k \cdot w = w^{6/5}$ bits to store all the keys. This means that, without any additional tools and algorithms, we cannot process each node in $O(1)$ time.

Fusion Nodes only use algorithms based on word RAM operations, listed in Section 2.1.3.3, such that we can query a Fusion Node with $O(1)$ number of operations. The first challenge is to fit $w^{6/5}$ bits in a word, such that we can use this result to query the node.

2.2.6.1 Pre-processing Phase

Since this is a static data structure, we use the pre-processing phase to build the k-ary tree with the given set of keys, disregarding its running time. Upon its construction, each node will contain at most k keys.

Bitwise Binary Tree Abstraction We note that, for any given node, when looking at the keys' values in binary and building a Bitwise Trie with the keys at that node, we need only to know at most $k-1$ bit indices in order to differentiate its keys. These are the branching bits in the Bitwise Trie built with those keys.

Let x be a key in the Fusion Node, a *sketching* function, sketch(x) is defined by keeping only the branching bits of x, as we have just defined.

We note that the sketches of the keys have the same order as the keys. Note also that if the *sketching* needs at most $k-1$ bits per key in the node, then in total, we need $k \cdot (k-1) = O(w^{2/5})$ bits to store all the bits of the key sketches; thus we can fit that in a single word, which is kept at each node.

Because it is difficult to compute perfect *sketches* (defined in the paragraph above) in $O(1)$ time using only word RAM operations, an *approximate sketch* is used instead. The key difference between the *perfect sketches* and *approximate sketches* is the amount of bits used to store the *sketches* of the keys: *approximate sketches* use at most $(k-1)^4 = O(w^{4/5})$ instead of the $O(w^{2/5})$ bits of the *perfect sketches*. By using *approximate sketches*, we can define constants and masks to be kept at each node that will help us define a function that will return us the *approximate sketch* of a query in $O(1)$ time. This is also the reason for the choice of the value $k = w^{1/5}$.

2.2.6.2 Querying

At this stage, we know that we have pre-computed all the *sketches* of the keys at every node.

To query in a Fusion Node, we need to be able to find the rank of a *sketch* within a word containing the *sketches* the keys in that node. To do so, we use the Rank Lemma 1 algorithm, thoroughly described in Section 2.3.6.

There is a scenario where we might query a key that does not belong in the node, thus computing the *sketch* of such key might produce a result that does not reflect the order of the keys. This happens when the queried key branches at a node whose corresponding bit was not considered in the *sketching* function.

Assume that we wish to query a key q among the set of keys in a Fusion Node, $\{x_0, x_1, \ldots, x_i\}$. Computing the rank of sketch(q) among the *sketches* of the keys at that node will give us other two keys in the node, x_i and x_{i+1}, such that sketch(x_i) $<$ sketch(q) $<$ sketch(x_{i+1}). As mentioned, this does not mean that $x_i < q < x_{i+1}$, but by computing the index of the bit where q branched from those keys, y, we can learn where q lies among all the other keys in the node. We achieve this by computing a mask, e, that:

- If q branched to the right in y, e.g., bit y was 1 in q, then $e = y011\ldots1$;
- Otherwise if q branched to the left, then $e = y100\ldots1$.

A claim states that finding where sketch(e) fits among the *sketches* of the keys at that node is the same as where q fits among the keys in the node, thus solving the problem [Nelson and Liu, 2014].

Learning the most significant set bit of a word in $O(1)$ time will also play an important role when querying. An algorithm that achieves this has been thoroughly described in Section 2.3.4.3. Let msb(z) denote the function that returns the most significant set bit of a word, z. After finding one of the x_i's (described above), we can easily find the branching bit, y, the expression:

$$y = \text{msb}(x_i \oplus q)$$

By knowing where query q fits among the keys in the node, the remaining operations

are trivially implemented, thus completing the description of Fusion Trees with $O(\log_w n)$ query time.

2.2.7 Summary

Table 2.1 is a summary of relevant data structures that incrementally lead to the data structure presented in this project. The running times are given in the word RAM model.

Data Structure	Remark	Running time Update	Query	Space
(Naïve) Array		$O(n)$	$O(\log_2 n)$	$O(n)$
Red-Black BST [Sedgewick and Wayne, 2011]		$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$
Binary Search Trie [Sedgewick, 2002]	Average	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$
	Worst	$O(w)$	$O(w)$	
Patricia Trie [Sedgewick, 2002]		$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$
van Emde Boas Tree [Nelson and Schorow, 2013]		$O(\log_2 w)$	$O(\log_2 w)$	$O(u)$
Fusion Tree [Demaine et al., 2012a]	Static	N/A	$O(\log_w n)$	$O(n)$
Dynamic Fusion Tree [Pătraşcu and Thorup, 2014]		$O(\log_w n)$	$O(\log_w n)$	$O(n)$

Table 2.1: Data structures used to solve the predecessor problem and their respective theoretical running times

2.3 Summary of Techniques used in the Present Implementation

All of the algorithms presented in this section have been implemented as static functions in the `Util` class, present in the repository.

2.3.1 Bit Operations

Let A be a w-bit length word. This section features a set of functions that operate on A's bits. Since these operations only use bitwise shifting and masking, all the algorithms presented in this section take $O(1)$ time.

2.3.1.1 Is the Bit Set

This operation aims at extracting the bit at position d in A or, in other words, getting the value of the bit at position d. Let us denote this operation by $\text{bit}(d, A)$. We have:

$$\text{bit}(d, A) = (A \gg d) \wedge 1$$

Example:

$$\begin{aligned} w &= 16 \\ d &= 8 \\ A &= 0101\ 1101\ 1100\ 0111_2 \\ \text{bit}(8, A) &= (0101\ 1101\ 1100\ 0111_2 \gg 8) \wedge 1 \\ \hline \text{bit}(8, A) &= 0000\ 0000\ 0101\ 1101_2 \\ &\wedge\ 0000\ 0000\ 0000\ 0001_2 \\ \hline \text{bit}(8, A) &= 1 \end{aligned}$$

2.3.1.2 Set Bit

This operation sets the bit at position d in A to 1. Let us denote this operation by $\text{setBit}(d, A)$. We have:

$$\text{setBit}(d, A) = A \vee (1 \ll d)$$

Example:

$$\begin{aligned} w &= 16 \\ d &= 4 \\ A &= 0101\ 1101\ 110\underline{0}\ 0111_2 \\ \text{setBit}(4, A) &= 0101\ 1101\ 110\underline{0}\ 0111_2 \vee (1 \ll 4) \\ \hline \text{setBit}(4, A) &= 0101\ 1101\ 110\underline{0}\ 0111_2 \\ &\vee\ 0000\ 0000\ 000\underline{1}\ 0000_2 \\ \hline \text{setBit}(4, A) &= 0101\ 1101\ 110\underline{1}\ 0111_2 \end{aligned}$$

Note that, if the bit a index d is alredy set, then A is returned unchanged.

2.3.1.3 Delete Bit

This operation sets the bit at position d in A to 0. Let us denote this operation by $\text{deleteBit}(d, A)$. We have:

$$\text{deleteBit}(d, A) = A \wedge \neg(1 \ll d)$$

Example:

$$\begin{aligned} w &= 16 \\ d &= 6 \\ A &= 0101\ 1101\ 1\underline{1}00\ 0111_2 \\ \text{deleteBit}(6, A) &= 0101\ 1101\ 1\underline{1}00\ 0111_2 \wedge \neg(1 \ll 6) \\ \text{deleteBit}(6, A) &= 0101\ 1101\ 1\underline{1}00\ 0111_2 \\ &\wedge\ \neg(0000\ 0000\ 0\underline{1}00\ 0000_2) \\ \hline \text{deleteBit}(6, A) &= 0101\ 1101\ 1\underline{1}00\ 0111_2 \\ &\wedge\ 1111\ 1111\ 1\underline{0}11\ 1111_2 \\ \hline \text{deleteBit}(6, A) &= 0101\ 1101\ 1\underline{0}00\ 0111_2 \end{aligned}$$

Note that, if the bit a index d is alredy 0, then A is returned unchanged.

2.3.2 Masks

Pătrașcu and Thorup refer constants that we will denote by masks, which are useful for bitwise operations. In this section, we will see how we can compute them.

- The mask $0^{w-j}1_2^j$ consists of a word with the $w-j$ most significant bits set to 0, and the least j significant bits set to 1. It is useful, for instance, to mask the first j least significant bits of a word and it is computed with the expression:

$$0^{w-j}1_2^j = (1 \ll j) - 1$$

Example:

$$\begin{aligned} w &= 16 \\ j &= 4 \end{aligned}$$

$$\begin{aligned} 0^{16-4}1_2^4 &= (1 \ll 4) - 1 \\ &= (0000\ 0000\ 0000\ 0001_2 \ll 4) - 1 \\ &= 0000\ 0000\ 0001\ 0000_2 - 1 \\ &= \underbrace{0000\ 0000\ 0000}_{w-j}\ \underbrace{1111}_{j}{}_2 \end{aligned}$$

Note that when $j = w$, then this mask is simply -1, because in the two's complement, the binary representation of -1 is 1^w.

- The mask $1^{w-j}0_2^j$ consists of a word with $w-j$ significant bits set to 1, and the least j significant bits set to 0. For instance, it is useful to mask the first $w-j$ most significant bits of a word. It is easily computed by negating the result from the previous expression:

$$1^{w-j}0_2^j = \neg((1 \ll j) - 1)$$

Example:

$$\begin{aligned} w &= 16 \\ j &= 4 \end{aligned}$$

$$\begin{aligned} 1^{16-4}0_2^4 &= \neg((1 \ll 4) - 1) \\ &= \neg((0000\ 0000\ 0000\ 0001_2 \ll 4) - 1) \\ &= \neg(0000\ 0000\ 0001\ 0000_2 - 1) \\ &= \neg 0000\ 0000\ 0000\ 1111_2 \\ &= \underbrace{1111\ 1111\ 1111}_{w-j}\ \underbrace{0000}_{j}{}_2 \end{aligned}$$

Note that when $j = w$, then this mask is simply 0.

2.3.3 Fields of Words

We follow the definitions from [Pătraşcu and Thorup, 2014] in regards to viewing words as sets of fields of some length $f \leq w$. Let A be a w-bit length word, then if A is comprised of fields, analogously to bit indexing of a word, its least significant field is the rightmost one, denoted $A\langle 0 \rangle_f$; its most significant field is the leftmost one, denoted $A\langle \lfloor w/f \rfloor \rangle_f$, and so on. Note that the functions presented in this section consist of simple bitwise shifting and masking with the expressions defined in 2.3.2. Regarding running times, all these algorithms take $O(1)$ time.

2.3.3.1 Field Retrieval

This operation consists of retrieving field $A\langle i \rangle_f$ and it is denoted by $\text{getField}(i, f, A)$. It is defined by:

$$\text{getField}(i, f, A) = (A \gg (i \times f)) \wedge ((1 \ll f) - 1)$$

Example:

$$\begin{aligned} i &= 1 \\ f &= 4 \\ A &= 0101\ 1101\ \underline{1100}\ 0111_2 \\ \text{getField}(1, 4, A) &= (0101\ 1101\ \underline{1100}\ 0111_2 \gg \underbrace{(1 \times 4)}_{4}) \wedge (\underbrace{(1 \ll 4) - 1}_{1111_2}) \end{aligned}$$

$$\begin{aligned} \text{getField}(1, 4, A) &= 0000\ 0101\ 1101\ \underline{1100}_2 \\ &\wedge\ 0000\ 0000\ 0000\ 1111_2 \end{aligned}$$

$$\text{getField}(1, 4, A) = 0000\ 0000\ 0000\ \underline{1100}_2$$

A range of fields can be retrieved in a single operation. We denote by $\text{getFields}(i, j, f, A)$ the operation consisting of the retrieval of the fields $\{A\langle i \rangle_f, \ldots, A\langle j-1 \rangle_f\}$, which is defined by:

$$\text{getFields}(i, j, f, A) = (A \gg (i \times f)) \wedge ((1 \ll ((j - i) \times f)) - 1)$$

Example:

$$\begin{aligned}
i &= 1 \\
j &= 3 \\
f &= 4 \\
A &= 0101\ \underline{1101}\ \underline{1100}\ 0111_2 \\
\text{getFields}(1,2,4,A) &= (0101\ \underline{1101}\ \underline{1100}\ 0111_2 \gg \underbrace{(1 \times 4)}_{4}) \wedge (\underbrace{(1 \ll ((3-1) \times 4)) - 1}_{1111\ 1111_2}) \\
\hline
\text{getFields}(1,2,4,A) &= 0000\ 0101\ \underline{1101}\ \underline{1100}_2 \\
&\wedge\ 0000\ 0000\ 1111\ 1111_2 \\
\hline
\text{getFields}(1,2,4,A) &= 0000\ 0000\ \underline{1101}\ \underline{1100}_2
\end{aligned}$$

We can also specify a lower field and retrieve all the fields from that position up to the end of the word. This is denoted by $\text{getFields}(i, f, A)$ and it is defined by:

$$\text{getFields}(i, f, A) = A \gg (i \times f)$$

Example:

$$\begin{aligned}
i &= 2 \\
f &= 4 \\
A &= \underline{0101}\ \underline{1101}\ 1100\ 0111_2 \\
\text{getFields}(2,4,A) &= \underline{0101}\ \underline{1101}\ 1100\ 0111_2 \gg \underbrace{(2 \times 4)}_{8} \\
\hline
\text{getFields}(1,2,4,A) &= 0000\ 0000\ \underline{0101}\ \underline{1101}_2
\end{aligned}$$

2.3.3.2 Field Assignment

Conversely, it is also possible to assign a value to a particular field. To do so, a mask m is required, and it is computed as a function of i (the position of the field to be set) and f (the length of the fields in A). Thus we have:

$$m = ((1 \ll f) - 1) \ll (i \times f)$$

Setting field y in A, denoted by $\text{setField}(i, y, f, A)$, is defined by:

$$\text{setField}(i, y, f, A) = \underbrace{(A \wedge \neg m)}_{\text{(a) Reset field}} \vee \underbrace{(y \ll (i \times f) \wedge m)}_{\text{(b) Set field}}$$

Example:

$$
\begin{aligned}
i &= 1 \\
y &= 1001_2 \\
f &= 4 \\
m = (\underbrace{(1 \ll 4) - 1}_{1111_2}) \ll (\underbrace{1 \times 4}_{4}) &= 0000\ 0000\ 1111\ 0000_2 \\
A &= 0101\ 1101\ \underline{1100}\ 0111_2
\end{aligned}
$$

$$
\begin{aligned}
\text{(a)} \quad A \wedge \neg m &= 0101\ 1101\ \underline{1100}\ 0111_2 \\
&\wedge\ 1111\ 1111\ 0000\ 1111_2 \\
\text{(a)} \quad &= 0101\ 1101\ 0000\ 0111_2
\end{aligned}
$$

$$
\begin{aligned}
y \ll (1 \times 4) &= 0000\ 0000\ 0000\ \underline{1001}_2 \ll 4 \\
&= 0000\ 0000\ \underline{1001}\ 0000_2 \\
\text{(b)} \quad (y \ll 4) \wedge m &= 0000\ 0000\ \underline{1001}\ 0000_2 \\
&\wedge\ 0000\ 0000\ 1111\ 0000_2 \\
\text{(b)} \quad &= 0000\ 0000\ \underline{1001}\ 0000_2
\end{aligned}
$$

$$
\begin{aligned}
\text{setField}(1, y, 4, A) = \text{(a)} \vee \text{(b)} &= 0101\ 1101\ \underline{0000}\ 0111_2 \\
&\vee\ 0000\ 0000\ \underline{1001}\ 0000_2
\end{aligned}
$$

$$
\text{setField}(1, y, 4, A) = 0101\ 1101\ \underline{1001}\ 0111_2
$$

2.3.4 Most Significant Set Bit

Learning about the most significant set bit of a word will be an important operation in the implementation. For this reason, we will look at different ways to achieve this result. Since it is to be used as a subroutine in certain operations, it has the potential to become a bottleneck if not implemented carefully. We denote the most significant set bit of x by $\text{msb}(x)$.

2.3.4.1 Naive

The simplest way to achieve the intended outcome is to loop through the word until the first non-zero bit is found, returning the word length minus the number of iterations that it took to find that bit.

This approach takes $O(w)$ time.

2.3.4.2 Lookup

In this approach, a lookup table containing the most significant bit answers for all combinations of 8 bits is pre-computed.

When a query comes, it is iterated in fields of 8 bits, starting from the most significant field. If that field is not 0, then the most significant bit of the queried word lies in that block. In this case, the answer will be the most significant set bit of that field plus all the bits in the non-iterated fields. Since the answer to any combination of 8 bits is pre-computed, we know the answer in constant time.

Otherwise, if the most significant field is 0, the algorithm looks at the second most significant field and does the same operation as described in the previous paragraph. It will iterate the fields until a non-zero is found, returning the answer which is given by the lookup table and the position of the field where the first non-zero bit was found.

This approach takes $O(1)$ time after the lookup table has been computed but, not only the lookup table takes time to compute, it also uses some space.

2.3.4.3 Constant Time with Parallel Comparison

This operation here described is based on the materials from the lectures whose scribe notes are [Nelson and Liu, 2014] and [Demaine et al., 2012b]. It comprises four main steps. Let x be the query for the most significant set bit:

1. We divide x in $\sqrt{w}$ fields of $\sqrt{w}$ bits. The goal is to *summarize* the fields in x such that if a field is not empty, then its summary is 1, and it is 0 otherwise.

2. We *compress* the summary such that it fits in a single $\sqrt{w}$-bit field. A bitwise shift operation is also carried in order to compute a word containing the resulting summary on its least significant field.

3. Then we do *parallel comparison* of the summary word to find the first non-empty field of x because it will be that field that contains the most significant bit of x.

4. We use the same technique as in step 3, but now on the first non-empty field of x. We end with simple arithmetic to return the most significant set bit.

This approach takes $O(1)$ time and requires a small lookup table of $\sqrt{w}/2$ size, which can easily be explicitly stored together with the algorithm.

We now run a small example where every step is illustrated. For simplicity, let $w = 16$. This implies that we have $\sqrt{16} = 4$ fields of 4 bits each. Let us also assume that our query is $x = 0101\ 0000\ 1000\ 1101_2$.

2.3.4.3.1 Step 1 — Summarize the query fields The goal of this step is to compute a summary word of w size, whose leading bit of each of its $\sqrt{w}$-bit fields is a summary of each $\sqrt{w}$-bit field of the query word x. If in a given field, the leading bit in the summary word is 1, then that field in x was not empty (one or more bits were set), and vice-versa.

1. We start by defining F. F is a w-bit word where the most significant position of every field is set to 1 and every other position is set to 0. In this particular example $F = 1000\ 1000\ 1000\ 1000_2$

2. In a local variable t_1, we store information about the leading bits of each field of x. This is done with $x \wedge F$.

$$\begin{aligned} x &= \underline{0}101\ \underline{0}000\ \underline{1}000\ \underline{1}101_2 \\ F &= 1000\ 1000\ 1000\ 1000_2 \\ \hline t_1 = x \wedge F &= \underline{0}000\ \underline{0}000\ \underline{1}000\ \underline{1}000_2 \end{aligned}$$

3. In another local variable t_2, we store x after setting the leading bits of each field to 0. This is done with $x \oplus t_1$.

$$\begin{aligned} x &= 0\underline{101}\ 0\underline{000}\ 1\underline{000}\ 1\underline{101}_2 \\ t_1 &= 0000\ 0000\ 1000\ 1000_2 \\ \hline t_2 = x \oplus t_1 &= 0\underline{101}\ 0\underline{000}\ 0\underline{000}\ 0\underline{101}_2 \end{aligned}$$

4. We subtract t_2 from F and save it to a local variable t_3. Since the leading bit of every field of F is 1, after subtracting t_2 from F, what remains is the information about if that field was empty (all zeros) or not. This information is given by the bit that remains at the most significant position of each resulting field. In other words,

if in any given field of the resulting word, the most significant bit is 1, then the corresponding field in x was empty; otherwise, it was not empty.

Since we only care about what remains of the most significant position of each field, in the example below, the remaining noise has been replaced with ?.

$$F = 1000\ 1000\ 1000\ 1000_2$$
$$t_2 = 0\underline{101}\ 0\underline{000}\ 0\underline{000}\ 0\underline{101}_2$$
$$t_3 = F - t_2 = \underline{0}???\ \underline{1}000\ \underline{1}000\ \underline{0}???_2$$

5. This step consists of clearing the noise from t_3 since we care only about knowing which fields in x were empty or not. To do so, we use $(\neg t_3) \wedge F$.

$$t_3 = \underline{0}???\ \underline{1}000\ \underline{1}000\ \underline{0}???_2$$
$$t_4 = \neg t_3 = \underline{1}???\ \underline{0}111\ \underline{0}111\ \underline{1}???_2$$
$$F = 1000\ 1000\ 1000\ 1000_2$$
$$t_5 = t_4 \wedge F = \underline{1}000\ \underline{0}000\ \underline{0}000\ \underline{1}000_2$$

6. The value calculated in step 2 for t_1 holds information about the leading bits of each field, whereas t_5 from step 5 contains the information about the non-leading bits. By merging both words, the resulting word will hold information about the whole word x. We achieve this with $t_1 \vee t_5$.

$$t_1 = \underline{0}000\ \underline{0}000\ \underline{1}000\ \underline{1}000_2$$
$$t_5 = \underline{1}000\ \underline{0}000\ \underline{0}000\ \underline{1}000_2$$
$$t_6 = t_1 \vee t_5 = \underline{1}000\ \underline{0}000\ \underline{1}000\ \underline{1}000_2$$

In fact, once F is defined, steps 2 to 5 can be computed all at once with the expression:

$$t_6 = (x \wedge F) \vee ((\neg(F - (x \oplus (x \wedge F)))) \wedge F) \tag{2.2}$$

2.3.4.3.2 Step 2 — Summary compression The goal of this step is to compress the summary word down to a single field.

1. The first step consists of shifting the leading bits of each field to the least significant position of each field. This is done with $t_6 \gg (\sqrt{w} - 1)$.

$$t_6 = \underline{1}000\ \underline{0}000\ \underline{1}000\ \underline{1}000_2$$
$$t_7 = t_6 \gg \underbrace{(\sqrt{w} - 1)}_{3} = 000\underline{1}\ 000\underline{0}\ 000\underline{1}\ 000\underline{1}_2$$

2. We wish now to copy the least significant bit of each field to a single field, keeping their relative order. Similarly to 2.3.4.3.1.1, we need to find a word C, which, when multiplied by t_7, will produce the result we are looking for. Let C be also partitioned in fields indexed $\{f_0, ..., f_{\sqrt{w}-1}\}$ where $f_{\sqrt{w}-1}$ is the most significant field. At every field f_i, the only set bit is $b_{\sqrt{w}-1-i}$). In our example $C = 0001\ 0010\ 0100\ 1000_2$.

3. Multiplying C with t_7 produces a result t_8 with all the important bits in their relative order in the most significant field. There will be some additional noise in the least significant fields, which we can easily clear by $t_8 \gg ((\sqrt{w}) \cdot (\sqrt{w} - 1))$.

$$t_7 = \underline{0001}\ \underline{0000}\ \underline{0001}\ \underline{0001}_2$$
$$C = 0001\ 0010\ 0100\ 1000_2$$
$$t_8 = t_7 \times C = \underline{1011}\ ????\ ????\ ????_2$$
$$x_s = t_8 \gg \underbrace{((\sqrt{w}) \cdot (\sqrt{w} - 1))}_{(16-4)} = 0000\ 0000\ 0000\ \underline{1011}_2$$

2.3.4.3.3 Step 3 — First non-empty field with parallel comparison At this stage, the goal is to compute the first non-empty field of x. To do so, we need to do a parallel comparison between x_s (the summary of x's fields) and the first $\sqrt{w}$ powers of two.

The parallel comparison consists of:

1. Taking a query x of length $l < w$, making as many copies of it as there are other vectors we wish to compare it with while padding these copies with 0. The resulting vector is stored in a word.

2. The vectors we wish to compare x with are stored in word A with each of their leading bits padded with 1.

3. We take the difference between A and the copies of x, checking how many of the padding bits remain 1 in A after this operation. The result stands for the number of vectors in A that were strictly smaller than x.

Since the vector we want to compare x_s with consists of $\sqrt{w}$ powers of two, we know that we will $\sqrt{w} \cdot (\sqrt{w} + 1)$ bits to store all such vectors. This extra bit per vector is due to the padding bit mentioned earlier. Since $\sqrt{w} \cdot (\sqrt{w} + 1) > w$, we know also that it might be necessary to do this operation in two iterations. Since a word does not hold enough bits to perform the comparison with a single word, we will split the vectors in two words: one representing the higher $\sqrt{w}/2$ powers of two, and another for the lower ones.

If the power of two vectors are sorted, this parallel comparison will be a monotone function, meaning that once the transition on the leading bit is found, the result is found. This also implies that if the result is found in the higher powers of two, then it is unnecessary to look for it in the lower end.

1. We define two bit-vectors, hi and lo, with $hi > lo$. These vectors are comprised of the concatenation of the $\sqrt{w}$ powers of two in sorted in descending order and padded with 1. From each power of two, we have to subtract 1 for this operation to work, because if we do not and the query is an exact power of two, the corresponding padding 1 will not be borrowed. Thus, in our example we have:

$$hi = 1 \ \underbrace{0111}_{2^3-1=7} \ 1 \ \underbrace{0011}_{2^2-1=3}{}_2$$
$$lo = 1 \ \underbrace{0001}_{2^1-1=1} \ 1 \ \underbrace{0000}_{2^0-1=0}{}_2$$

2. We define another vector t_9 consisting of $\sqrt{w}/2$ concatenated copies of x_s where each of the copies is prefixed by 0. This vector is achieved by multiplying x_s with a word M consisting of $\sqrt{w}/2$ fields of size $\sqrt{w}+1$ where the least significant bit of each field is set to 1. Note that, because we have set the size of each field of M to have an additional bit, this corresponds to prefix a 0 to each $\sqrt{w}$ field:

$$x_s = 0000\ 0000\ 0000\ \underline{1011}_2$$
$$M = 0\ 0001\ 0\ 0001_2$$
$$\hline$$
$$t_9 = x_s \times M = 0\ \underline{1011}\ 0\ \underline{1011}_2$$

3. To find the first non-empty field of x, we take the differences between t_9 and hi and lo respectively. The answer will lie in the first field whose leading bit resulted in a 0 after the operation.

$$hi = \underline{1}\ \underline{1}000\ \underline{1}\ \underline{0}100_2$$
$$lo = \underline{1}\ \underline{0}010\ \underline{1}\ \underline{0}001_2$$
$$t_9 = 0\ 1011\ 0\ 1011_2$$
$$\hline$$
$$t_{10} = hi - t_9 = \underline{0}\ ????\ \underline{0}\ ????_2$$
$$t_{11} = lo - t_9 = \underline{0}\ ????\ \underline{0}\ ????_2$$

4. Looking at t_{10}, we note that the first non-empty field of x is the most significant one because the leading bit of the first field resulting from the difference between hi and t_9 is now 0. Nevertheless, we will do the same operations on both t_{10} and t_{11} and concatenate the results, so we end up with a single result for the whole query. Note that in the previous step, the actual values within each field besides the leading bit are irrelevant, so we do some masking and shifting, similar to what was done in 2.3.4.3.1 and 2.3.4.3.2.

   a) The first step is to clear all the irrelevant bits in the fields and turn the leading

bits from 0 to 1. We choose a mask for this specific purpose.

$$t_{10} = \underline{0}\ ????\ \underline{0}\ ????_2$$
$$t_{11} = \underline{0}\ ????\ \underline{0}\ ????_2$$
$$Mask_1 = 1\ 0000\ 1\ 0000_2$$

$$t_{12} = (t_{10} \wedge Mask_1) \oplus Mask_1 = \underline{1}\ 0000\ \underline{1}\ 0000_2$$
$$t_{13} = (t_{11} \wedge Mask_1) \oplus Mask_1 = \underline{1}\ 0000\ \underline{1}\ 0000_2$$

b) Secondly, we wish to summarize the results in a single field. To achieve so, we first multiply the vectors by an integer V, which construction follows what we have done for 2.3.4.3.2.2, and that result will put all the important bits consecutive in the most significant field.

$$t_{14} = t_{12} \gg \sqrt{w} = 0\ 000\underline{1}\ 0\ 000\underline{1}_2$$
$$t_{15} = t_{13} \gg \sqrt{w} = 0\ 000\underline{1}\ 0\ 000\underline{1}_2$$

$$V = \ 01000\ 10000_2$$
$$t_{16} = t_{14} \times V = ???????\ \underline{11}00?\ ?????_2$$
$$t_{17} = t_{15} \times V = ???????\ \underline{11}00?\ ?????_2$$

c) After the multiplication, we have to remove the noise and move the result to the least significant field. We achieve this by defining another mask for this purpose, followed by the necessary bitwise shifts. The final step will be to merge the results of the higher and the lower powers of two: $t_{hi} \vee t_{lo}$.

$$Mask_2 = (1 \ll (\sqrt{w}/2)) - 1 = 11_2$$
$$t_{18} = t_{16} \wedge (Mask_2 \ll 2 \cdot \sqrt{w}) = \ \underline{11}\ 0000\ 0000_2$$
$$t_{19} = t_{17} \wedge (Mask_2 \ll 2 \cdot \sqrt{w}) = \ \underline{11}\ 0000\ 0000_2$$

$$t_{hi} = t_{18} \gg (\sqrt{w} + \sqrt{w}/2) = 00\ 0000\ \underline{11}00_2$$
$$t_{lo} = t_{19} \gg (\sqrt{w} \cdot \sqrt{w}/2) = 00\ 0000\ 00\underline{11}_2$$

$$t_q = t_{hi} \vee t_{lo} = \underline{1111}_2$$

5. Note that t_q can only be one of $\sqrt{w}$ possibilities: 0001_2, 0011_2, 0111_2, or 1111_2. For this reason, we implement a lookup table of size $\sqrt{w}$ for these values, which we return at the end of the method. We can even halve the size of the lookup table by

applying the following trick: instead of merging t_{hi} and t_{lo} into a single field, shift t_{hi} all the way to the right. In this scenario, t_{hi} can either be 11_2, 01_2 or 0. The values of the lookup table, in this case, will be only 11_2 and 01_2, so if t_{hi} takes one of those values, we return the value in the lookup table plus a constant corresponding to the least significant positions ($\sqrt{w}/2$). Should it be 0, then we do the lookup for t_{lo} and return the corresponding value in the lookup table.

In our example, since $t_q = 1111_2$ the result of this parallel comparison would be 3 (three), meaning, the method should now look at $x\langle 3\rangle_4$ (the most significant field).

2.3.4.3.4 Step 4 — Final result After learning in which field lies the most significant bit, we now run the same method as in 2.3.4.3.3 but now with the actual field.

1. We start by extracting the field from our query x. Let i be the result returned from 2.3.4.3.3, e.g. the index of the field. We extract f_i by shifting $(\sqrt{w} - 1 - i)$ fields in x to the right and bitwise $\wedge$ the result with a mask for this purpose.

$$x = \underline{0101}\ 0000\ 1000\ 1101_2$$
$$Mask_3 = (1 \ll \sqrt{w}) - 1 = 0000\ 0000\ 0000\ 1111_2$$

$$t_{20} = (x \gg \underbrace{(\sqrt{w} \cdot (\sqrt{w} - 1 - i))}_{4\times(4-1-0)}) \wedge Mask_3 = 0000\ 0000\ 0000\ \underline{0101}_2$$

2. We run another parallel comparison with 0101_2. Let d be the result of computing the parallel comparison of the first non-empty field of x, then in our example $d = 2$.

3. We can now compute the overall most significant bit of x. The final result is given the expression:

$$\text{msb}(x) = d + i \cdot \sqrt{w}$$

In our example, this will evaluate to:

$$\text{msb}(0101\ 0000\ 1000\ 1101_2) = 2 + 3 \cdot \sqrt{16} = 2 + 3 \times 4 = 14$$

2.3.5 Least Significant Set Bit

Let $\text{lsb}(x)$ denote the least significant set bit of x. According to Pătrașcu and Thorup, after computing $\text{msb}(x)$ we can in $O(1)$ time compute $\text{lsb}(x)$ with the expression:

$$\text{lsb}(x) = \text{msb}((x-1) \oplus x)$$

Example:

$$\begin{aligned} x &= 0101\ 1101\ 1100\ 0\underline{1}00_2 \\ x-1 &= 0101\ 1101\ 1100\ 0\underline{0}11_2 \\ (x-1) \oplus x &= 0000\ 0000\ 0000\ 0\underline{1}00_2 \end{aligned}$$

$$\text{lsb}(x) = \text{msb}((x-1) \oplus x) = \text{msb}(0000\ 0000\ 0000\ 0\underline{1}00_2) = 2$$

2.3.6 Rank Lemma 1

Another important operation in the context of the implementation of the data structure presented in [Pătraşcu and Thorup, 2014] is to compute the rank of a word. For this purpose, some subroutines are required, and the algorithm described in this section is one of them. This algorithm consists of the implementation of Lemma 1 by Fredman and Willard, cited by Pătraşcu and Thorup. It reads:

Lemma 1 *Let $m \cdot b \leq w$. If we are given a b-bit number x and a word A with m b-bit numbers stored in sorted order, that is, $A\langle 0\rangle_b < A\langle 1\rangle_b < \cdots < A\langle m-1\rangle_b$, then in constant time, we can find the rank of x in A, denoted rank(x,A).*

An algorithm that implements $\text{rank}(x, A)$ works as the following:

1. Computing how many fields in A have 0 as their leading bit.
2. Computing the leading bit of x.
3. If the leading bit of x is 0, then:
   a) We bitwise shift and mask A as needed, such that the fields whose leading bit are 1 are no longer present in A.
   b) We compute a word consisting of as many concatenated copies of x as there are fields left in A.
   c) We do parallel comparison[1] between (the shifted/masked) A and (copies of) x by setting the leading bit of each of the remaining fields of A to 1 and computing the difference between those words.
   d) The result of $\text{rank}(x, A)$ is given by the number of fields whose leading bit is now 0 because if the leading bit (that has been set to 1 in the previous step) is borrowed in the subtraction, then x is larger than the key stored at that field.

   Otherwise, if the leading bit of x is 1, then:

   a) We bitwise shift and/or mask A such that the fields whose leading bit is 0 are no longer present in A, storing the number of fields removed from A on a local variable.
   b) We set the leading bit of x to 0 and compute a word consisting of as many concatenated copies of x as there are fields left in A.
   c) We do parallel comparison between the remaining fields in A and the word we computed just before.

[1] See Section 2.3.4.3.3 to learn more about *parallel comparison*.

d) The result of $\text{rank}(x, A)$ is given by the number of fields whose leading bit was 0 just before A was shifted/masked plus the number of fields whose leading bit is 0 after the parallel comparison.

This operation takes $O(1)$ time, and we will run an example of this algorithm, explaining its intricacies.

2.3.6.1 Parameters

Since this algorithm branches depending on the leading bit of the query x, we will run the example with two queries x_1 and x_2 such that we explore both branches of the algorithm. Let A be the concatenation of the keys of $S = \{0101_2, 0110_2, 1100_2, 1110_2\}$. Note that since the keys in A are sorted, if key $y_i < y_j$, then y_i will be present in A on a less significant position than y_j.

$$\begin{aligned} A &= \underbrace{1110}_{A\langle 3\rangle_4}\ \underbrace{1100}_{A\langle 2\rangle_4}\ \underbrace{0110}_{A\langle 1\rangle_4}\ \underbrace{0101}_{A\langle 0\rangle_4}{}_2 \\ x_1 &= 1100_2 \\ x_2 &= 0111_2 \\ b &= 4 \\ m &= 4 \end{aligned}$$

2.3.6.2 Step 1 — Computing which fields have zero as their leading bit

Since the keys in A are sorted, we know that finding the position the least significant field whose first bit is 1 in A will tell us how many fields there are with 0 at their leading bit.

1. We start by finding M. M consists of a word with m fields of b-bits, where each field is filled with zeroes excluding the least significant bit, which is set to 1. In this particular example $M = 0001\ 0001\ 0001\ 0001_2$.

2. We mask the non-leading bits of each field of A. This is achieved with:

$$A \wedge (M \ll (b-1))$$

Thus we have:

$$\begin{aligned} M &= 0001\ 0001\ 0001\ 0001_2 \\ t_1 = M \ll (b-1) &= 1000\ 1000\ 1000\ 1000_2 \\ A &= \underline{1}110\ \underline{1}100\ \underline{0}110\ \underline{0}101_2 \\ \hline t_2 = A \wedge t_1 &= \underline{1}000\ \underline{1}000\ \underline{0}000\ \underline{0}000_2 \end{aligned}$$

3. Computing $\text{lsb}(t_2)$ will give us the index of leading bit of the least significant field whose leading bit is 1. Dividing the previous result by the field size, b, gives us the position of the first least significant field whose leading bit is 0. In this particular example:

$$b = 4$$
$$\text{lsb}(t_2) = \text{lsb}(1000\ \underline{1}000\ 0000\ 0000_2) = 11$$
$$t_3 = \frac{\text{lsb}(t_2)}{b} = \frac{11}{4} = 2$$

We know now that the first 2 least significant fields of A have 0 as their leading bit.

2.3.6.3 Step 2 — Computing the leading bit of the query

In order to extract the value of the leading bit of our queries x_1 and x_2, we resort to the algorithm of section 2.3.1. The leading bit of the query will be at position $b-1$, thus we have for x_1:

$$x_1 = \underline{1}100_2$$
$$\text{bit}(b-1, x_1) = \text{bit}(3, x_1) = 1$$

And for x_2:

$$x_2 = \underline{0}111_2$$
$$\text{bit}(b-1, x_2) = \text{bit}(3, x_2) = 0$$

2.3.6.4 Step 3 — Computing rank with parallel comparison

We will now run the algorithm for each of its branches. If the leading bit of the query is 1, then we run the first branch; otherwise, the second.

- In this branch our query is $x_1 = 1100_2$:

  1. Since the leading bit of our query x_1 is 1, we know now that its rank in A is at least the number of fields in A whose leading bit is 0. So we proceed by removing those fields from A, and for this purpose we resort to the $\text{getFields}(i, f, A)$ method defined in Section 2.3.3.1. In our example:

$$A = \underline{1110}\ \underline{1100}\ 0110\ 0101_2$$
$$b = 4$$
$$t_3 = 2$$
$$A \coloneqq \text{getFields}(t_3, b, A) = 0000\ 0000\ \underline{1110}\ \underline{1100}_2$$

2. We apply the same principle to the M word by doing the very same operation as just before.

$$\begin{aligned} M &= \underline{0001}\ \underline{0001}\ 0001\ 0001_2 \\ b &= 4 \\ t_3 &= 2 \end{aligned}$$

$$M \coloneqq \text{getFields}(t_3, b, M) = 0000\ 0000\ \underline{0001}\ \underline{0001}_2$$

3. We set the leading bit of x_1 to 0 and multiply the result by M to produce a word containing concatenated copies of x_1.

$$\begin{aligned} x_1 &= 1100_2 \\ M &= 0000\ 0000\ \underline{0001}\ \underline{0001}_2 \end{aligned}$$

$$x_1' \coloneqq \text{deleteBit}(b-1, x_1) = 0100_2$$

$$x_1'' \coloneqq x_1 \times M = 0000\ 0000\ \underline{0100}\ \underline{0100}_2$$

4. The last ingredient needed for the parallel comparison is a mask comprised of a word whose leading bit of each field is 1. Shifting M by $b-1$ positions to the left will produce such result.

$$M = 0000\ 0000\ 000\underline{1}\ 000\underline{1}_2$$

$$M \coloneqq M \ll (b-1) = 0000\ 0000\ \underline{1}000\ \underline{1}000_2$$

5. We take the difference between A and x_1'' and mask out the bits at all positions except the leading bits of each field in the resulting word.

$$\begin{aligned} A &= 0000\ 0000\ 1\underline{110}\ 1\underline{100}_2 \\ x_1'' &= 0000\ 0000\ 0\underline{100}\ 0\underline{100}_2 \\ M &= 0000\ 0000\ \underline{1}000\ \underline{1}000_2 \end{aligned}$$

$$\begin{aligned} A - x_1'' &= 0000\ 0000\ \underline{1}???\ \underline{1}???_2 \\ d_1 = (A - x_1) \wedge M &= 0000\ 0000\ \underline{1}000\ \underline{1}000_2 \end{aligned}$$

6. To learn how many of the remaining fields in A are smaller than x_1, we compute

$\text{lsb}(d_1)$ and divide it by b.

$$\begin{aligned} d_1 &= 0000\ 0000\ 1000\ \underline{1}000_2 \\ \text{lsb}(d_1) &= 3 \\ b &= 4 \end{aligned}$$

$$t_4 = \frac{\text{lsb}(d_1)}{b} = \frac{3}{4} = 0$$

7. Computing $\text{rank}(x_1, A)$ consists of returning the number of fields in A that are strictly smaller than x_1. In this specific context, we know that all fields whose leading bit is 0 are smaller than x_1, and we have also just computed the number of fields whose leading bit is 1 but that are still smaller than x_1. So we just add those results, and we have:

$$\begin{aligned} \text{rank}(x_1, A) &= t_3 + t_4 = 2 + 0 \\ \text{rank}(x_1, A) &= 2 \end{aligned}$$

- In this branch our query is $x_2 = 0111_2$:

1. Since the leading bit of our query x_2 is 0, we can safely disregard the fields in A whose leading bit is 1. Similarly to what was done in the first branch, M suffers a similar change, and we will start with that by using the $\text{getFields}(i, j, f, A)$ function defined in Section 2.3.3.1. Thus we have:

$$\begin{aligned} M &= 0001\ 0001\ \underline{0001}\ \underline{0001}_2 \\ b &= 4 \\ t_3 &= 2 \end{aligned}$$

$$M \coloneqq \text{getFields}(0, t_3, b, M) = 0000\ 0000\ \underline{0001}\ \underline{0001}_2$$

2. We need to have as many concatenated copies of x_2 as there are fields in A whose leading bit is 0. So we multiply M by x_2 and we end with:

$$\begin{aligned} x_2 &= 0111_2 \\ M &= 0000\ 0000\ \underline{0001}\ \underline{0001}_2 \end{aligned}$$

$$x_2' \coloneqq x_2 \times M = 0000\ 0000\ \underline{0111}\ \underline{0111}_2$$

3. As before, we need a mask to perform the parallel comparison. So we use the same technique as in the other branch and shift M by $b-1$ positions to the left:

$$M = 0000\ 0000\ 000\underline{1}\ 000\underline{1}_2$$

$$M \coloneqq M \ll (b-1) = 0000\ 0000\ \underline{1}000\ \underline{1}000_2$$

4. Lastly, for the last ingredient of the parallel comparison, we discard the fields in A whose leading bit is 1, and set the leading bit of the remaining fields to 1. Thus we have:

$$\begin{aligned} A &= 1110\ 1100\ \underline{0110}\ \underline{0101}_2 \\ b &= 4 \\ t_3 &= 2 \\ M &= 0000\ 0000\ \underline{1}000\ \underline{1}000_2 \end{aligned}$$

$$A \coloneqq \text{getFields}(0, t_3, b, A) = 0000\ 0000\ \underline{0110}\ \underline{0101}_2$$

$$\begin{aligned} A \coloneqq M \vee A &= 0000\ 0000\ \underline{1}000\ \underline{1}000_2 \\ &\vee\ 0000\ 0000\ \underline{0}110\ \underline{0}101_2 \end{aligned}$$

$$A = 0000\ 0000\ \underline{1}110\ \underline{1}101_2$$

5. We take the difference between A and x_2' and mask out the bits at all positions except the leading bits of each field in the resulting word.

$$\begin{aligned} A &= 0000\ 0000\ 1\underline{110}\ 1\underline{101}_2 \\ x_2' &= 0000\ 0000\ \underline{0111}\ \underline{0111}_2 \\ M &= 0000\ 0000\ \underline{1}000\ \underline{1}000_2 \end{aligned}$$

$$\begin{aligned} A - x_2' &= 0000\ 0000\ \underline{0}???\ \underline{0}???_2 \\ d_2 = (A - x_2') \wedge M &= 0000\ 0000\ \underline{0}000\ \underline{0}000_2 \end{aligned}$$

6. Note that d_2 is 0. Since there are no set bits in d_2 then x_2 is larger than all x_2 must be larger than all the remaining fields in A, so we just return t_2.

$$\text{rank}(x_2, A) = t_3 = 2$$

3 Implementations

All the implementing classes described and discussed in this chapter can be found in the folder shown in figure 3.1.

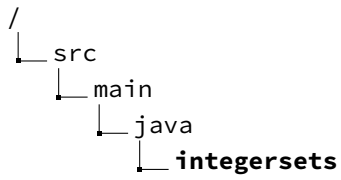

Figure 3.1: Location of the folder containing the implementations

Note that all the functions and implementations consider the integer keys as unsigned integers.

3.1 Utility Functions

This project features a class, `Util`, which implements static utility functions. In can be found in the path specified in figure 3.2.

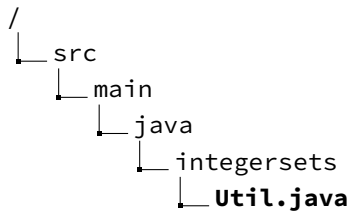

Figure 3.2: Location of the `Util.java` file in the folder structure

Below find listed the functions included in this class, together with a brief explanation. The implemented functions feature 32-bit versions, which, for brevity, were omitted. These differ solely on the input word type.

3.1.1 Bit Operations

The algorithms which these functions implement are described in Section 2.3.1. The functions included are:

- The method below returns the value of the bit at position d in A. It is enforced that $d \in [0, w[$.

```
1 public static int bit(final int d, final long A)
```

- The method below sets the bit at index d in A to 1 and returns A after the change. If the bit was already 1 or if the provided index $d \notin [0, w[$, then the function has no effect.

```
1 public static int setBit(final int d, int A)
```

- The method below sets the bit at index d to 0 and returns A after the change. If the bit was already 0 or if the provided index $d \notin [0, w[$, then the function has no effect.

```
1 public static int deleteBit(final int d, int A)
```

3.1.2 Fields of Words

The algorithms which these functions implement are described in Section 2.3.3. The functions included are:

- The method below returns $A\langle i \rangle_f$. It is enforced that $f \in [0, w]$ and $i \times f \in [0, w]$.

```
1 public static long getField(final int i, final int f, final long A)
```

- The method below returns $A\langle i, j \rangle_{g \times f}$. It is enforced that $f \in [0, w]$, $g < f$ and $i \times g + j \in [0, w]$.

```
1 public static int getField2d(final int i, final int j, final int g, final int
      f, final int A)
```

- The method below returns the range of fields $A\langle i \dots j \rangle_f$. It is enforced that $f \in [0, w]$, $i \times f \in [0, w]$, $j \times f \in [0, w]$, $i < j$, and $f \cdot (j - i) < w$.

```
1 public static long getFields(final int i, final int j, final int f, final long
       A)
```

- The method below returns the range of fields $A\langle i \cdots * \rangle_f$. It is enforced that $f \in [0, w]$ and $i \times f \in [0, w]$.

```
1 public static long getFields(final int i, final int f, final long A)
```

- The method below sets $A\langle i \rangle_f \coloneqq y$ and returns A after the change. It is enforced that $f \in [0, w]$ and $i \times f \in [0, w]$. Only the bits in $y\langle 0 \rangle_f$ are considered.

```
1 public static int setField(final int i, final int y, final int f, final long A
      )
```

3.1.3 String Representation of an Integer in Binary

- The method below returns a string representation of integer x in binary prefixed by `0b`, including leading zeros (and suffixed by `l` if it is a 64-bit integer).

```
1 public static String bin(final long x)
```

- The method below returns a string representation of integer x in binary prefixed by `0b`, including leading zeros, spaced by `_` every f bits counting from the least significant bit (and suffixed by `l` if it is a 64-bit integer).

```
1 public static String bin(final long x, final int f)
```

3.1.4 Helper Functions

- The method below implements a helper function for the `rankLemma1` method, returning the index of the transition from 0 to 1 in $field$, e.g., which powers of two are smaller than the input $field$.

```
1 private static int parallelComparison(final long field)
```

- This method is a helper function for the `parallelComparison` method, and simply returns the value associated with a given *pow* in a small lookup table.

```
1 private static int parallelLookup(final int pow)
```

- The method below is a helper method for the most significant set bit lookup functions, as described in [Anderson, 2005][1] and mentioned below. It populates a lookup table, allowing these functions to return the result quickly.

```
1 private static void generateLookupTable()
```

3.1.5 Most and Least Significant Set Bit

The algorithms which these functions implement are described in Sections 2.3.4 and 2.3.5. The functions included are:

- The method below returns the index of the most significant set bit of the target x. It is used as a subroutine in many other functions. The actual operation used to compute the result can be easily changed in the body of the method by altering the function that is called. The version featured with this report calls `msbConstant`, defined a few bullet points below.

[1] The following link redirects to the function that inspired this implementation: `https://graphics.stanford.edu/~seander/bithacks.html#IntegerLogLookup`

```
1 public static int msb(final long x)
```

- The method below returns the index of the least significant set bit of x.

```
1 public static int lsb(final long x)
```

- The method below returns the index of the most significant set bit of x by calling a Java standard library function and computing the result with an expression.

```
1 public static int msbLibrary(final long x)
```

- The method below implements a naive algorithm for computing the index of the most significant set bit of the target x, as described in [Anderson, 2005][2].

```
1 public static int msbObvious(long x)
```

- The method below implements a lookup algorithm for computing the index of the most significant set bit of the target x, as described in [Anderson, 2005][3]. For 64-bit integers, the function first splits the integer in two, calling the 32-bit method on the high half first, and then on the second in the first half is 0.

```
1 public static int msbLookupDistributedInput(final long x)
```

- The method below implements the algorithm from Section 2.3.4.3, which follows the lecture notes from [Demaine et al., 2012b] and [Nelson and Liu, 2014].

```
1 public static int msbConstant(long x)
```

3.1.6 Rank Lemma 1

The method below implements rank(x, A) from Section 2.3.6.

```
1 public static int rankLemma1(long x, long A, final int m, final int b)
```

The input parameters are:

- A query x, the integer to find the rank in A.
- A word A, containing keys with the same size as x, which must be sorted for the method to return a sound result.
- The number of keys in A, m.
- The length of each key in A, b, in bits.

[2]The following link redirects to the function that inspired this implementation: `https://graphics.stanford.edu/~seander/bithacks.html#IntegerLogObvious`

[3]The following link redirects to the function that inspired this implementation: `https://graphics.stanford.edu/~seander/bithacks.html#IntegerLogLookup`

3.1.7 Additional Utility Functions

- The method below takes a 64-bit integer x and returns a two-entry array, each position containing 32 of the 64 bits of x. The least significant bits of x will be at index 0, whereas the most significant bits will be at position 1.

```
1 public static int[] splitLong(final long x)
```

- The method below computes the reverse of the method just above. It takes a 32-bit integer array and combines its first two positions into a single 64-bit integer. Again, position 0 of the input array is used for the least significant bits of the resulting 64-bit integer, whereas position 1 will populate the remaining 32 most significant positions.

```
1 public static long mergeInts(final int[] x)
```

- The helper method below produces a word comprised of w/b fields of b bits in length, having each field its least significant bit set to 1. E.g., the resulting word will have the bits at index 0 and every b^{th} index set to 1. This function computes the result in $O(w/b)$ time.

```
1 public static long M(final int b, final int w)
```

- The helper method below returns a string representation of interpreting the matrix stored in the word A, with $\#rows$ rows and $\#columns$ columns.

```
1 public static String matrixToString(final int rows, final int columns, final
      long A)
```

- The helper method below returns an array containing n distinct `long` keys between 0 and *bound* (exclusive) produced with seed *seed* and sorted with an unsigned comparator.

```
1 public static long[] distinctBoundedSortedLongs(final int n, final long bound,
       final long seed)
```

- The helper method below returns an array containing n distinct `long` keys between 0 and *bound* (exclusive) produced with the default seed 42 and sorted with an unsigned comparator.

```
1 public static long[] distinctBoundedSortedLongs(final int n, final long bound)
```

- The helper method below returns an array containing n distinct `long` keys between 0 and the maximum unsigned value (exclusive) produced with the seed *seed* and sorted with an unsigned comparator.

```
1 public static long[] distinctSortedLongs(final int n, final long seed)
```

- The helper method below returns an array containing n distinct `long` keys between 0 and the maximum unsigned value (exclusive) produced with the default seed 42 and sorted with an unsigned comparator.

```
1 public static long[] distinctSortedLongs(final int n)
```

3.2 The RankSelectPredecessorUpdate Interface

Figure 3.3: Location of the `RankSelectPredecessorUpdate.java` file in the folder structure

As stated in Section 2.1.2, the data structure presented in [Pătraşcu and Thorup, 2014] solves the dynamic predecessor problem. For this reason, an interface denoted `RankSelectPredecessorUpdate` was implemented and it can be found in the path specified in figure 3.3. It features the following the method signatures and default methods:

- The insert(x) operation sets $S = S \cup \{x\}$ and it is to be implemented by a method with the signature:

```
1 void insert(long x);
```

- The delete(x) operation sets $S = S \setminus \{x\}$ and is to be implemented by a method with the signature:

```
1 void delete(long x);
```

- The member(x) operation returns `true` if $x \in S$, and `false` otherwise. It is implemented as a default method, making all implementing classes automatically inheriting the method:

```
1 default boolean member(final long x) {
2     if (isEmpty()) {
3         return false;
4     }
5     final Long res = successor(x);
6     return res != null && res == x;
7 }
```

- The predecessor(x) operation returns $\max\{y \in S \mid y < x\}$ and it is implemented as a default method, making all implementing classes automatically inheriting the method:

```
1 default Long predecessor(long x) {
2     return select(rank(x) - 1);
3 }
```

- The successor(x) operation returns $\min\{y \in S \mid y \geq x\}$ and it is implemented as a default method, making all implementing classes automatically inheriting the method:

```
1 default Long successor(long x) {
2     return select(rank(x));
3 }
```

- The rank(x) operation returns $\#\{y \in S \mid y < x\}$ and it is to be implemented by a method with the signature:

```
1 long rank(long x);
```

- The select(i) operation returns $y \in S$ with $rank(y) = i$ and it is to be implemented by a method with the signature:

```
1 Long select(long rank);
```

Additionally, the following method signatures/default method are included in the interface:

- A `size()` method that returns the current number of keys in the set:

```
1 long size();
```

- An `isEmpty()` default method that returns `true` if the set is empty and `false` otherwise:

```
1 default boolean isEmpty() {
2     return size() == 0;
3 }
```

- A `reset()` method that removes all current elements from the set:

```
1 void reset();
```

Note that some of the methods return a primitive type, whereas some others return a boxed type. This is because some queries are not mapped to any answer, and the boxed type provides the perfect way to model this kind of scenario: in this situation, `null` is returned.

3.3 Naive Implementation

Goal To implement a naive dynamic predecessor data structure with an array.

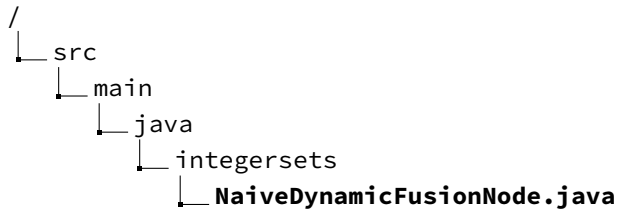

Figure 3.4: Location of the `NaiveDynamicFusionNode.java` file in the folder structure

We start with a naive implementation of name `NaiveDynamicFusionNode`, which follows closely the Array approach mentioned in Section 2.2.1. It holds a *key* array to store the keys, and a counter for the current number of keys in the data structure. The keys can be any 64-bit integer.

The basic idea behind this implementation is to store the keys in *key*, making their respective rank the same as their index in the array. In other words, *key* is always sorted.

Updates, e.g., insert and delete, take $O(n)$ time. This is because whenever a key is inserted, its rank i is found, and the key at that position as well as all following keys up to $n-1$ are updated. Since any given key index in *key* is its rank, all the keys with rank larger than the new key have to be moved one position to the right in *key* to make room for the new key and keeping rank consistent.

A rank query takes $O(\log_2 n)$ because a binary search is performed on *key*. Select is faster: $O(1)$, because we need only to access and return the key at position i in *key* to fulfill the query.

3.4 Dynamic Fusion Node with Binary Search for Rank

Goal To improve the time performance of the implementation presented in Section 3.3 by adding instance variables that enable indexing the keys by their rank. By doing so, updates and rank queries now take $O(\log_2 n)$ time. Select takes $O(1)$ time.

This implementation can be found in the file highlighted in figure 3.5.

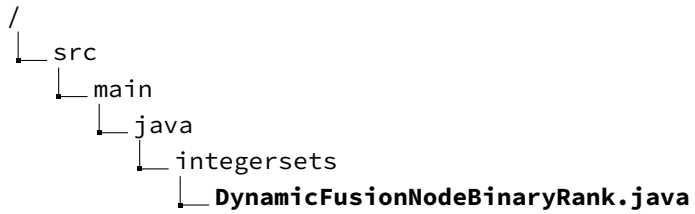

Figure 3.5: Location of the `DynamicFusionNodeBinaryRank.java` file in the folder structure

We take a step forward by improving the previous idea: this time, two additional words, *index* and $bKey$, are kept in the fields. The goal is to use the concepts described in the *Indexing* section of [Pătraşcu and Thorup, 2014] to implement this data structure. The running times will be $O(\log_2 n)$ for updating and querying, and this is because the rank operation resorts to binary search to produce the result.

An important parameter introduced in this implementation is k. It defines the capacity limit of the set. We wish to maximize this parameter, but we will see below that the word size, w, will be this parameter's constraining factor in this implementation.

This time, *key* is no longer sorted, but we still have access to the keys in their sorted order by using a simple device: we interpret *index* as an array of k fields of $\lceil \log_2 k \rceil$ bits in length. It indexes the keys in *key* by their rank the following way: Let i be the rank of a key in the set, then $index\langle i \rangle_{\lceil \log_2 k \rceil}$ will have the index in *key* of the key with rank i. In order to maximize the use of the number of bits in *index*, and knowing that our program is working with 64-bit, as stated in Section 2.1.1, we maximize k by solving the following inequation:

$$\begin{aligned} & k \cdot \log_2(k) \leq 64 \\ \iff & k \leq 16 \end{aligned} \tag{3.1}$$

Note that this is precisely what we have mentioned in Section 2.1.3.2 with expression 2.1. We maximized the capacity of the set by solving k in inequation 3.1. Thus we know now that if $k = 16$, then $\lceil \log_2(k) \rceil = 4$. So, each $\lceil \log_2(k) \rceil$-bit field of *index* stores the index of the key in *key* in their sorted order.

We interpret $bKey$ as an array where the values of the first k bits correspond to the positions in *key*. Like any other word, the bits in $bKey$ are indexed from 0 to $k-1$, and if the i^{th} bit is set to 1, then position i in *key* is free to store a key, and vice-versa. We initialize $bKey$ as -1, which represents the case where the set is empty.

3.4.1 Fields

The class holds the following fields:

- The class constants k and $\lceil \log_2(k) \rceil$:

```
1 private static final int k = 16;
2 private static final int ceilLgK = (int) Math.ceil(Math.log10(k)/Math.log10(2)
      );
```

- The array of keys, *key*:

```
1 private final long[] key = new long[k];
```

- The *index* word:

```
1 private long index;
```

- The map of the empty entries in *key*, $bKey$:

```
1 private int bKey;
```

  Note that only the first k bits are relevant for any given instance.

- An integer n, containing the current number of keys in the set:

```
1 private int n;
```

3.4.2 Helper Methods

- `firstEmptySlot()` returns the first available spot in *key* by computing $\text{lsb}(bKey)$. Since only the first k spots are valid results, a check is done to see if this result is within the range, returning -1 if not.

- `fillSlot(final int j)` sets position j in $bKey$ to not empty. This is done with a call to $\text{deleteBit}(j, bKey)$.

- `vacantSlot(int j)` sets the j^{th} position of $bKey$ to empty by calling $\text{setBit}(j, bKey)$.

- `getIndex(final long i)` returns the index in *key* of the key with rank i. This operation is done with a call to $\text{getField}(i, \lceil \log_2 k \rceil, index)$.

- The purpose of the overloaded method `updateIndex` is to maintain the correspondence between the rank of the keys in the set and their position in *key*. The version with `updateIndex(final int i)` signature removes rank i from *index*, whereas the `updateIndex(final int i, final int slot)` inserts in *index* at position i the index in *key* (here denoted by *slot*). Both versions make calls to the getFields methods

with the adequate parameters, merge the results with bitwise $\vee$ and write those back in *index*.

- `binaryRank(final long x)` returns the rank of x in the set, using `select(final long rank)` as a subroutine. As the name implies, the algorithm used in this implementation is binary search.

3.4.3 Implementation of the Interface Methods

- `insert(final long x)`:

  1. We call member(x) to know if the x is already in the set. If it is, then the method returns.
  2. We compare `size()` with k. This is because we can only insert if there is room for another key. We progress if there is room for x.
  3. The rank of x in the set is found with a call to rank(x) and stored in a local variable i.
  4. The first available spot in *key* is found with a call to `firstEmptySlot()` and stored in a local variable j.
  5. We store x in *key* at the position returned by `firstEmptySlot()`.
  6. We set the position taken x in *key* to not empty by calling `fillSlot(j)`.
  7. We update the *index* to reflect the new key's insertion with a call to `updateIndex(i, j)`.
  8. Lastly, we increment n by 1, updating the current total number of keys in the set.

- `delete(final long x)`:

  1. If the x is not in the set, the method does not progress. This is done with a call to member(x).
  2. The rank of the x in the set is found with a call to rank(x) and stored in a local variable i.
  3. $bKey$ is updated by making the spot taken by x in *key* empty. This is done with the call `vacantSlot(getIndex(i))`.
  4. We update the *index* to reflect the deletion of x with a call to `updateIndex(i)`.
  5. Lastly, we decrement n by 1, updating the current total number of keys in the set.

- `rank(final long x)` returns the rank of x in the set by calling the `binaryRank(final long x)` helper method.
- `select(final long rank)` starts by checking if the *rank* is within range, returning `null` if not. Then it calls `getIndex(rank)` and returns the key at that position in *key*.
- `size()` returns the value of n.
- `reset()`. Resetting the set is easily done by setting n to 0 and $bKey$ to -1 (because -1 has all the bits set to 1 in its two's complement binary representation).

3.4.4 Example

The *Indexing* technique combined with binary rank is best understood with an example, which we showcase in this section.

Assume that we have instantiated a `DynamicFusionNodeBinaryRank` and, for brevity, let $k = 8$. Then:

$$k = 8 \implies \lceil \log_2 k \rceil = 3$$

These will remain constant and are stored as static fields:

```
1 private static final int k = 8;
2 private static final int ceilLgK = (int) Math.ceil(Math.log10(k) / Math.log10(2));
```

Assume that at this point the above-mentioned instance contains the keys from expression 3.2, thus the mentioned instance maps to S.

$$S = \{10, 12, 42, -1337, -42\} \tag{3.2}$$

The order in which keys are inserted will influence the order in which they will appear in the instance variable *key*. But, as mentioned in Section 3.4, the instance variable *index* will index the key indices in *key* by their rank whereas the bit values of $bKey$ will specify if the corresponding position in *key* is empty. Thus a possible state for the instance is the following:

i	7	6	5	4	3	2	1	0
$index\langle i\rangle_3$	?	?	?	$\underbrace{000_2}_{0_{10}}$	$\underbrace{100_2}_{4_{10}}$	$\underbrace{110_2}_{6_{10}}$	$\underbrace{001_2}_{1_{10}}$	$\underbrace{011_2}_{3_{10}}$

(a) *index*

i	7	6	5	4	3	2	1	0
$bKey\langle i\rangle_1$	1	0	1	0	0	1	0	0

(b) *bKey*

i	7	6	5	4	3	2	1	0
$key[i]$	?	42	?	−1337	10	?	12	−42

(c) *key*

Figure 3.6: Set S represented as the *index*, *bKey* and *key* instance variables in a `DynamicFusionNodeBinaryRank` instance

An important note is that the values at indices in *key* marked with "?" might hold values, but these are not accessible since they are not index in *index*.

3.4.4.1 Querying

3.4.4.1.1 Select Let i be the rank of the key we wish to query. Then

$$\text{select}(i) = key[index\langle i\rangle_{\lceil \log_2 k\rceil}]$$

Assume we wish to know the key with rank 2, then:

$$\begin{aligned} i &= 2 \\ index\langle 2\rangle_3 &= 110_2 = 6 \\ \hline \text{select}(2) &= key[6] = 42 \end{aligned}$$

3.4.4.1.2 Rank As previously mentioned, at this point, the rank operation has been implemented with binary search using select as a subroutine. This works in $O(\log_2 n)$ time because select queries take $O(1)$ time, the select method allows us to access the keys as if they were stored in their respective sorted order and therefore, the running time is bound by the binary search algorithm itself.

3.4.4.2 Updating

3.4.4.2.1 Insert Let $x = -1000$ be the key we wish to insert and S the set we wish to insert it in, defined in expression 3.2 and figure 3.6. The insertion of x in S on this data structure is comprised of the following:

1. We check if x is already in the set, returning in such a case. This is not the case, so we proceed.

2. Then we check if the set has room for the new key by comparing `size()` with k. If it is full, then an exception is thrown. This is not the case, so we proceed.

3. We compute rank(x) and store it in a local variable i.

$$\frac{x = -1000}{i \coloneqq \text{rank}(-1000) = 4}$$

4. We find the first empty position in key with a call to `firstEmptySlot()`, storing the result in a local variable j.

$$j \coloneqq \texttt{firstEmptySlot()} = 2$$

5. We store x in key at position j. We can see key after this update on table 3.1.

i	7	6	5	4	3	2	1	0
$key[i]$	?	42	?	-1337	10	-1000	12	-42

Table 3.1: Instance variable key after setting $key[2] = -1000$

6. We mark position j as not empty, thus having table 3.2 as the resulting $bKey$. This is done with the call `fillSlot(j)`.

i	7	6	5	4	3	2	1	0
$bKey\langle i \rangle_1$	1	0	1	0	0	0	0	0

Table 3.2: Instance variable $bKey$ after marking position 2 as not empty

7. With i and j, we can now update $index$ such that the indices of the keys are once again sorted by the keys' ranks. This is done by calling `updateIndex(i, j)`. The resulting $index$ is shown in table 3.3.

i	7	6	5	4	3	2	1	0
$index\langle i \rangle_3$	?	?	$\underbrace{000_2}_{0_{10}}$	$\underbrace{010_2}_{2_{10}}$	$\underbrace{100_2}_{4_{10}}$	$\underbrace{110_2}_{6_{10}}$	$\underbrace{001_2}_{1_{10}}$	$\underbrace{011_2}_{3_{10}}$

(a) $index$

Table 3.3: Instance variable $index$ after the insertion of a key with rank 4 at position 2 in key

8. Lastly, we increment the instance variable n.

3.4.4.2.2 Delete We will now delete $x = -1000$, having as starting point the state the instance was right after the insertion described in 3.4.4.2.1. Deleting x from S on this data structure is comprised of the following:

1. We check if x is not in the set, returning in such a case. This is not the case, so we proceed.

2. We compute rank(x) and store it in a local variable i.

$$\frac{x = -1000}{i \coloneqq \text{rank}(-1000) = 4}$$

3. With a call to `getIndex(i)` we get the position in key where x is stored. We use that to mark that position as empty in $bKey$. This is done with `vacantSlot(getIndex(i))`. After this update, $bKey$ will be the same as in table 3.6c.

4. We also need to update $index$ by moving all the indices with rank larger than i, the rank of x, 1 position to the left. We do this by calling `updateIndex(i)`. This update will make $index$ be as shown in table 3.6a.

5. Lastly, we decrement the instance variable n.

3.5 Rank via Matching with "Don't Cares"

Goal Having the implementation from Section 3.4 as a starting point, the goal is to implement rank queries via matching with "don't cares". Rank will now take $O(k)$ time, and this is due to a subroutine, `match`, which at this stage is computed naively. Select queries keep their $O(1)$ time, whereas updates now take $O(k \cdot n)$ because the instance variables introduced in this implementation are maintained naively.

Static Fusion Trees have less than optimal update time because adding a new key entails recomputing sketches. Pătrașcu and Thorup address this by introducing "don't cares", which provides a simulation of a Patricia Trie at the node level. This is relevant because, differently to what happens at the Fusion Tree node from Fredman and Willard, inserting a new key in a Patricia Trie corresponds to adding a new branch node [Pătrașcu and Thorup, 2014].

In this incremental step, we keep the most of the implementation details of Section 3.4, plus we store the compressed keys with "don't cares" to enable the rank operation as described in the *Matching with don't cares* section of [Pătrașcu and Thorup, 2014].

The implementation discussed in this section can be found in the file highlighted in figure 3.5.

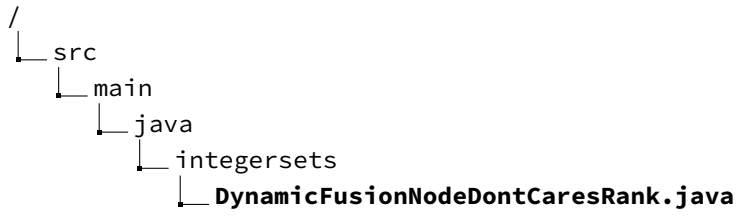

Figure 3.7: Location of the `DynamicFusionNodeDontCaresRank.java` file in the folder structure

Enabling rank via matching with "don't cares" entails the following:

- Compressing keys (or *sketching*) in the same style as Fusion Trees, from Fredman and Willard.

  In order to compress keys, we need a compressing key. Let $S = \{x_i, \ldots, x_j\}$. A naive way to compute a compressing key is to set the $\text{msb}(x_i \oplus x_j)$ bit by of all combinations of $\{x_i, \ldots, x_j\}$ keys in S in a word.

  We denote the compressed version of x by $\hat{x}$. A naive way to compress a key x is to iterate through the set bits of the compressing key, storing the bit values at those positions of x in $\hat{x}$.

- Encoding the information about "don't cares". We denote a compressed key with "don't cares" by $\hat{x}^?$. A compressed key with "don't cares" has length k and can be any combination of the characters $\{0, 1, ?\}$. Since the bits of a word in a real machine can only take the values $\{0, 1\}$, we need an additional construct to enable the third

character. This shows that a single word for storing the compressed keys with "don't cares" is not enough. So we keep two new instance variables, the words: *branch* and $free$.

These words are to be interpreted as two $k \times k$ bit matrices, and they are defined in the following way:

- We see each field $branch\langle i \rangle_k$ as a row in the matrix, and each row maps to a compressed key in the set with rank i. As its name implies, *branch* will contain the values of the branching bits. So each column j, e.g. $branch\langle i, j \rangle_{1 \times k}$, corresponds to a branching bit. We will store the compressed keys except for the "don't cares" positions, which are always stored with value 0. In other words, if a particular branching bit of a compressed key is not a "don't care", then the value of the bit at that position in *branch* (in its corresponding row) will be the same as of bit at that position in the corresponding compressed key; otherwise, it is 0.

- In regards to $free$, its rows and columns have the same correspondence as in *branch*. When seeing the keys in a matrix and ordered by their rank, we can see that the value of some of those positions does not influence the rank of the (compressed) keys. These positions will be "don't cares". We will encode this data in the following way: If a particular branching bit of a compressed key is a "don't care", the bit at the same index in $free$ (in its corresponding row) is 1, otherwise it is 0.

- A subroutine match(x) that uses all of the above and rank via Rank Lemma 1 (defined in Section 2.3.6 and implemented in Section 3.1.6).

  Let $\hat{x}^k$ be the compressed version of x copied k times and stored in a word. After compressing x and multiplying $\hat{x}$ with the mask M (which has been computed via the `Util` helper function and stored as a class variable), we achieve the intended result. Note that we have already used multiplication in a similar context in Section 2.3.6.2. Then match(x) is computed with the following expression [Pătraşcu and Thorup, 2014]:

$$\text{match}(x) = \underbrace{\text{rank}(\hat{x}, branch \vee (\hat{x}^k \wedge free))}_{\text{Rank Lemma 1}}$$

The implementation of Section 3.4 had the word *index* as the limiting factor of k because we had to be able to index all the positions of the set. This time, the limiting factor for k will be the *branch* and $free$ matrices: since those have k rows and k columns and a word in our machine is of length 64, we have:

$$\begin{aligned} k^2 &\leq 64 \\ \implies k &\leq \sqrt{64} \\ \iff k &\leq 8 \end{aligned}$$

This class shares many implementation details with the *Dynamic Fusion Node* with binary Rank from Section 3.4. For that reason, in the next few sections, we will see what is kept,

added, or altered compared with that implementation while expanding only on the last two. The highlighted changes enable the new operations of this class.

3.5.1 Fields

The following fields were kept unchanged and stand for the same as in 3.4.1:

```
1 private final long[] key = new long[k];
2 private long index;
3 private int bKey;
4 private int n;
```

The following fields were either altered or introduced:

- The parameter k has been altered. This is because, as explained in 3.5, *branch* and $free$ will limit further the capacity of the set:

```
1 private static final int k = 8;
```

- We will need a constant M to be used as mask in some of methods. We resort to the `M` function of the `Util` class, and store the result as a class variable.

```
1 private static final long M = Util.M(k, k * k);
```

- The operations introduced in this implementation require us to compress keys. To this extent, we store a word, $compressingKey$, which will retain the information regarding the bits to keep when compressing a key. If a bit is significant, then it will be set to 1 in $compressingKey$.

```
1 private long compressingKey;
```

- As explained in 3.5, the word *branch* is interpreted as a $k \times k$ bit matrix, $BRANCH$, that Pătraşcu and Thorup use to store the part of the data of compressed keys with "don't cares". We see each $branch\langle i \rangle_k$ as a row in the matrix, each row corresponding to the compressed version of the key with rank i in the set. Each column j, e.g., $branch\langle i, j \rangle_{k \times 1}$ corresponds to a branching bit of the compressed key with "don't cares" of rank i.

```
1 private long branch;
```

- The word $free$ is the $k \times k$ bit matrix $FREE$, which is used for storing the rest of the data of the compressed keys with "don't cares". Indexing of rows and columns are mapped in the same way as in *branch*.

```
1 private long free;
```

3.5.2 Helper Methods

The following helper methods suffered no changes and serve the same purpose as in the implementation from Section 3.4.2:

- `firstEmptySlot()`
- `fillSlot(final int j)`
- `vacantSlot(int j)`
- `getIndex(final long i)`
- The overloaded methods `updateIndex`

The following are helper methods introduced in this implementation:

- `updateCompressingKey()` updates the compressing key by looping through all the combinations of keys in the set and setting all the first branching bits between those keys in $compressingKey$. E.g., let x and y be a particular combination of two keys in the set. The method sets in $compressingKey$ the $\text{msb}(x \oplus y)$ of all combinations of keys in the set. This approach is naive and slow, with $O(n^2)$ time.
- `compress(final long x)` uses $compressingKey$ in order to compute the compressed version of the input key, x. It loops through the set bits of $compressingKey$, setting the bits $\hat{x}$ by reading actual bit values at those positions in x. Since this is done naively, it takes $O(k)$ time.
- `long compressedKeys()` returns a word containing all the compressed keys present in the set and ordered by rank. This is done by looping through all keys in the set and calling the `compress(x)` method on every key. This method is called after upon updating the set, and since all the compressed keys are recomputed, this takes $O(n)$ time.
- The `updateFree(final long compressedKeys)` method makes a call to the recursive `dontCares` method (explained below) which updates $free$ after the insertion or deletion of a key.
- The `dontCares` method has the following signature:

```
1 private long dontCares(long compressedKeys, long free, final int bit, final
      int lo, final int hi)
```

It uses the $compressedKeys$ word to compute $free$ recursively. The method with store the data about "don't cares" in $free$ as it iterates. The $compressedKeys$ parameter is not to be confused with $branch$, which is computed after $free$ is computed, and it is computed by the `compressedKeys()` method, described above, containing only the compressed keys. It is because we need to compute which positions are "don't cares" that we use the plain compressed keys with this method.

bit refers to the index of a column (in both $compressedKeys$ and $free$), whereas *hi* and *lo* refers to a range of rows (also in both $compressedKeys$ and $free$). $free$ is updated by following the steps:

1. Starting from the most significant column (given by the parameter *bit*), e.g., the most significant bit of the compressed key and a range of keys that includes all rows:
   - If all bits in that column are the same, then that position is a "don't care" for all keys. This is because, in this range, regardless of the value of the bit, the order of the keys is always the same.
   - If at least one bit differs in that column, then we care for that position in all keys. Again, the particular value for these rows and column contributes to the order of the compressed keys.
2. The method is called recursively:
   - If the bits were all the same for that range and in that column, then the method is called with the same range of rows, but now on the next lesser significant column.
   - Otherwise, we do two recursive calls: one for the range of rows whose bit was set to 0 and another for the keys that had the bit set to 1 in that column.
3. The recursive call chain reaches its end when the column index, *bit*, is -1.

Since the method iterates through all k columns and n rows (which correspond to compressed versions of keys in the set) of $free$, this algorithm takes $O(k \cdot n)$ time.

- After having $compressedKeys$ and $free$ computed, the `final long compressedKeys` method updates *branch* with the expression:

$$branch \coloneqq compressedKeys \wedge \neg free$$

This takes $O(1)$ time.

- The helper method `match(x)` has the signature:

```
1 private int match(final long x)
```

As mentioned in 3.5, it implements the operation described in the *Matching with don't cares* section of [Pătrașcu and Thorup, 2014]. Since it uses `compress(final long x)` as a subroutine, its running time is bounded by the running time of that operation.

- The `dontCaresRank(final long x)` returns the rank of x in the set, using `match(final long x)`, `select(final long rank)`, `msb(final int x)` as subroutines. The implementation follows the algorithm presented by Pătrașcu and Thorup

in the *Matching with don't cares* section of [Pătrașcu and Thorup, 2014]. Its running time is bounded by the running time of its slowest subroutine, `match(final long x)`, which in turn calls `compress(final long x)`, which takes $O(k)$ time.

3.5.3 Implementation of the Interface Methods

The following interface methods suffer no changes and serve the same purpose as the implementation from Section 3.4:

- `select(final long rank)`
- `size()`

The following interface methods suffer changes in relation to Section 3.4.3:

- `insert(final long x)` keeps all the steps described in Section 3.4.3 and adds the following right after those steps:
  1. The *compressingKey* is updated with a call to `updateCompressingKey()`.
  2. All the keys are compressed and the result is stored in a local variable with the statement:

```
1 final long compressedKeys = compressedKeys();
```

  3. *free* is updated with the call:

```
1 updateFree(compressedKeys);
```

  4. And the last step is to update *branch*, which is done with:

```
1 updateBranch(compressedKeys);
```

- Similarly to `insert`, `delete(final long x)` keeps all the steps from Section 3.4.2 and adds the following right after those steps:
  1. The *compressingKey* is updated with a call to `updateCompressingKey()`.
  2. All the keys are compressed and the result is stored in a local variable with the statement:

```
1 final long compressedKeys = compressedKeys();
```

  3. *free* is updated with the call:

```
1 updateFree(compressedKeys);
```

  4. And the last step is to update *branch*, which is done with:

```
1 updateBranch(compressedKeys);
```

- `rank(final long x)` returns the rank of x in the set by calling the `dontCaresRank(final long x)` helper method.
- `reset()`. Resetting the data structure now also entails resetting some of the instance variables. The additional operations are:
  - Setting $compressingKey$ to 0 because, in an empty set, there are no branching bits.
  - Setting $branch$ to 0 because, in an empty set, there are no compressed keys.
  - Setting $free$ to -1, because in an empty set, there are no keys and bits set in $free$ mean that we do not care for those positions of the compressed keys.

3.5.4 Example

This section is aimed at showcasing how the compressed keys with "don't cares" are stored in the data structure and what are the steps taken by the algorithm to return a rank(x) query using the algorithm implemented in the `dontCaresRank(final long x)` method.

Our example starts with a set containing the keys from table 3.4 and a rank query key $x = 1010\ 1010\ 1111_2$. For brevity, the selected set of keys will only have bits set between the indices 0 and 11. This way, we know that the 52 leading bits will have no influence when running the algorithm.

	11	10	9	8	7	6	5	4	3	2	1	0
7	1	0	1	1	0	0	0	0	1	1	1	0
6	1	0	1	0	1	1	0	1	0	1	1	0
5	1	0	1	0	1	1	0	0	1	1	1	1
4	1	0	1	0	1	1	0	0	1	1	0	0
3	1	0	0	1	0	1	0	1	0	0	0	1
2	0	1	1	0	1	0	1	1	1	1	1	1
1	0	1	1	0	0	0	1	0	0	1	1	0
0	0	0	0	1	0	1	0	0	1	0	1	0

Table 3.4: Keys present in the set stored in *key* in binary. The header row are the bit indices and the rank of the keys is the first column.

We can already see that rank(x) = 4.

3.5.4.1 Key Compression

The keys from table 3.4 will produce the following compressing key (with the 52 most significant bits omitted for brevity):

$$compressingKey = 1111\ 1001\ 0010_2$$

Having this compressing key implies that, when compressing keys, we keep only the bits at positions 1, 4, 7, 8, 9, 10, and 11. We can see these indices highlighted in table 3.5.

	11	10	9	8	7	6	5	4	3	2	1	0
7	1	0	1	1	0	0	0	0	1	1	1	0
6	1	0	1	0	1	1	0	1	0	1	1	0
5	1	0	1	0	1	1	0	0	1	1	1	1
4	1	0	1	0	1	1	0	0	1	1	0	0
3	1	0	0	1	0	1	0	1	0	0	0	1
2	0	1	1	0	1	0	1	1	1	1	1	1
1	0	1	1	0	0	0	1	0	0	1	1	0
0	0	0	0	1	0	1	0	0	1	0	1	0

Table 3.5: The highlighted columns correspond to the set bits of the compressing key. When compressing a key, we keep only the bits of the highlighted columns.

After compressing our keys with our $compressingKey$, we end up with the result of table 3.6.

	7	6	5	4	3	2	1	0
7	0	1	0	1	1	0	0	1
6	0	1	0	1	0	1	1	1
5	0	1	0	1	0	1	0	1
4	0	1	0	1	0	1	0	0
3	0	1	0	0	1	0	1	0
2	0	0	1	1	0	1	1	1
1	0	0	1	1	0	0	0	1
0	0	0	0	0	1	0	0	1

Table 3.6: Compressed Keys

3.5.4.2 Compressed Keys with "Don't Cares"

As previously mentioned, the particular value of some of the bits at some positions has no influence on the rank of the keys. These positions have been replaced with "?" on table 3.7. The operation of transforming compressed keys in compressed keys with "don't cares" corresponds to what the `dontCares` method described in 3.5.2 does.

	7	6	5	4	3	2	1	0
7	?	1	?	1	1	?	?	?
6	?	1	?	1	0	?	1	?
5	?	1	?	1	0	?	0	1
4	?	1	?	1	0	?	0	0
3	?	1	?	0	?	?	?	?
2	?	0	1	?	?	1	?	?
1	?	0	1	?	?	0	?	?
0	?	0	0	?	?	?	?	?

Table 3.7: Compressed Keys with "don't cares"

In the data structure, this data is stored in the two class variables, $branch$ and $free$, which follows the logic explained in 3.5. They are each a single w-bit word. If we lay each of those words in a $k \times k$ matrix, then each row will correspond to a compressed key. This is precisely what we see in the tables from figure 3.8.

	7	6	5	4	3	2	1	0
7	0	1	0	1	1	0	0	0
6	0	1	0	1	0	0	1	0
5	0	1	0	1	0	0	0	1
4	0	1	0	1	0	0	0	0
3	0	1	0	0	0	0	0	0
2	0	0	1	0	0	1	0	0
1	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0

(a) $branch$

	7	6	5	4	3	2	1	0
7	1	0	1	0	0	1	1	1
6	1	0	1	0	0	1	0	1
5	1	0	1	0	0	1	0	0
4	1	0	1	0	0	1	0	0
3	1	0	1	0	1	1	1	1
2	1	0	0	1	1	0	1	1
1	1	0	0	1	1	0	1	1
0	1	0	0	1	1	1	1	1

(b) $free$

Figure 3.8: Compressed keys with "don't cares" stored in the resulting words $branch$ and $free$ displayed in $k \times k$ matrices

3.5.4.3 Querying

Sections 3.5.4.1 and 3.5.4.2 have handled the contents of the set and the relevant class variables for this operation. We shall now see what unfolds as soon as we query rank($1010\ 1010\ 1111_2$).

1. The method starts by checking if the set is empty, returning 0 in this situation. It is not the case in the example, so we proceed.

2. A call to `match` with the query x is done: match($1010\ 1010\ 1111_2$). The operation entails compressing the query, computing the word $branch \vee (\hat{x}^k \wedge free)$ and computing the rank of $\hat{x}$ in that word via Rank Lemma 1.

a) We compress the query with compress(1010 1010 1111_2). Thus we have:

$$compressingKey = 1111\ 1001\ 0010_2$$
$$x = \underline{1010}\ \underline{1010}\ \underline{1111}_2$$

$$\hat{x} = \underbrace{\underline{0101}\ \underline{0101}_2}_{k}$$

b) We make k copies $\hat{x}$:

$$M = (0^{k-1}1)_2^k$$
$$\hat{x} = 0101\ 0101_2$$

$$\hat{x}^k = M \times \hat{x} = (0101\ 0101_2)^k$$

If we lay the resulting word in a matrix we have:

	7	6	5	4	3	2	1	0
7	0	1	0	1	0	1	0	1
6	0	1	0	1	0	1	0	1
5	0	1	0	1	0	1	0	1
4	0	1	0	1	0	1	0	1
3	0	1	0	1	0	1	0	1
2	0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	1
0	0	1	0	1	0	1	0	1

Table 3.8: k copies of $\hat{x}$ in a word laid in a $k \times k$ matrix

c) We compute $\hat{x}^k \wedge free$, which keeps only the "don't cares" bits in $\hat{x}^k$. By bitwise $\wedge$ tables 3.8 and 3.8b we have:

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0	1
5	0	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0	0
3	0	0	0	0	0	1	0	1
2	0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0	1
0	0	0	0	1	0	1	0	1

Table 3.9: $\hat{x}^k \wedge free$ in a word laid in a $k \times k$ matrix

d) We compute $branch \vee (\hat{x}^k \wedge free)$, which is done by bitwise $\vee$ tables 3.8a and

3.9. This operation will use the actual bits of $\hat{x}^k$ in the "don't cares" positions of all the compressed keys with "don't cares" in the set. Thus we end up with:

	7	6	5	4	3	2	1	0
7	0	1	0	1	1	1	0	1
6	0	1	0	1	0	1	1	1
5	0	1	0	1	0	1	0	1
4	0	1	0	1	0	1	0	0
3	0	1	0	0	0	1	0	1
2	0	0	1	1	0	1	0	1
1	0	0	1	1	0	0	0	1
0	0	0	0	1	0	1	0	1

Table 3.10: $branch \vee (\hat{x}^k \wedge free)$ in a word laid in a $k \times k$ matrix

e) Now, match returns $\text{rank}(\hat{x}, branch \vee (\hat{x}^k \wedge free))$ via Rank Lemma 1. In table 3.10, we can see that $\hat{x} = 0101\ 0101_2$ is larger than the keys up to row 4, meaning that its rank is 5 (the highlighted row). Thus we have:

$$\text{match}(x) = 5$$

This result is stored in a local variable, i.

3. We perform the query $\text{select}(i)$:

$$\frac{i = 5}{\text{select}(5) = 1010\ 1100\ 1111_2}$$

This is stored in a local variable, y.

4. We compare x with y using the library function `compareUnsigned`, storing the result in a local variable, *comp*.

$$\frac{\begin{array}{r} x = 1010\ 1010\ 1111_2 \\ y = 1010\ 1100\ 1111_2 \end{array}}{x < y \implies comp = -1}$$

If $comp = 0$ then $x = y$, which means that x is already present in the set, thus its rank will be i. In this case, the method would return i. In this example, this is not the case; thus, we proceed.

5. We find the branching bit between x and y and store it in a local variable, j. This

is done with $\text{msb}(x \oplus y)$:

$$\begin{aligned} x &= 1010\ 1010\ 1111_2 \\ y &= 1010\ 1100\ 1111_2 \\ \hline x \oplus y &= 0000\ 0\underline{1}10\ 0000_2 \\ \hline j &= \text{msb}(x \oplus y) = 6 \end{aligned}$$

6. The idea behind the current step is described in the *Desketchifying* chapter of [Demaine et al., 2012b]. Below, we also mention the abstraction of viewing the keys at the node as if they were in a Bitwise Binary Tree. This abstraction has been explained in Section 2.2.6.1.

   To find the actual rank of x, we need to use the match function once again. We know now that there is a key, y, that matched x but branched away from x in the 6^{th} bit. We have now two cases:

   - If $comp < 0$ then $x < y$, so we apply a mask to x that keeps only its $w - j$ most significant bits and sets the j least significant bits to 0. Let the value of x after this mask be x_m; when applying *match* to x_m, we go down the tree to the left as much as possible after the branching bit j because all the bits of x_m after j will be 0. The rank of x will be the $\text{match}(x_m)$, which Pătrașcu and Thorup denote by i_0 in [Pătrașcu and Thorup, 2014].

   - Otherwise, $x > y$. We apply a mask to x that sets its j least significant bits to 1. When calling match with x after this mask, we will go down the tree to the right as much as possible after the branching bit j because all the bits of x_m after j will be 1. The rank of x will be the $\text{match}(x_m) + 1$, which Pătrașcu and Thorup denote as $\text{match}(x_m) = i_1$ in [Pătrașcu and Thorup, 2014].

   The above-mentioned masks have been defined in Section 2.3.2.

   In our example, we have $x < y$, so we are in the first case. We apply a mask to x that keeps only the $w - j$ most significant bits of x.

$$\begin{aligned} j &= 6 \\ mask &= \neg((1 \ll j) - 1) \\ \hline mask &= \underbrace{1111\ 11}_{w-j}\underbrace{00\ 0000}_{j}{}_2 \\ x &= \underline{1010}\ \underline{10}10\ 1111_2 \\ \hline x_m &= x \wedge mask = \underline{1010}\ \underline{10}00\ 0000_2 \end{aligned}$$

   Finally we call match with x_m:

a) We compress the query with compress(1010 1000 0000_2). Thus we have:

$$\begin{aligned} compressingKey &= 1111\ 1001\ 0010_2 \\ x_m &= \underline{1010}\ \underline{1000}\ \underline{0000}_2 \\ \hline \hat{x}_m &= 0\underline{101}\ \underline{0100}_2 \end{aligned}$$

b) We make k copies $\hat{x}$:

$$\begin{aligned} M &= (0^{k-1}1)_2^k \\ \hat{x}_m &= 0101\ 0100_2 \\ \hline \hat{x}_m^k = M \times \hat{x}_m &= (0101\ 0100_2)^k \end{aligned}$$

If we lay the resulting word in a matrix we have:

	7	6	5	4	3	2	1	0
7	0	1	0	1	0	1	0	0
6	0	1	0	1	0	1	0	0
5	0	1	0	1	0	1	0	0
4	0	1	0	1	0	1	0	0
3	0	1	0	1	0	1	0	0
2	0	1	0	1	0	1	0	0
1	0	1	0	1	0	1	0	0
0	0	1	0	1	0	1	0	0

Table 3.11: k copies of $\hat{x}_m$ in a word laid in a $k \times k$ matrix

c) We compute $\hat{x}_m^k \wedge free$, which keeps only the "don't cares" bits in $\hat{x}^k$. By bitwise $\wedge$ the tables 3.11 and 3.8b we have:

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0	0
3	0	0	0	0	0	1	0	0
2	0	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0	0
0	0	0	0	1	0	1	0	0

Table 3.12: $\hat{x}_m^k \wedge free$ in a word laid in a $k \times k$ matrix

d) We compute $branch \vee (\hat{x}_m^k \wedge free)$, which is done which bitwise $\vee$ tables 3.8a and 3.12. This operation will use the actual bits of $\hat{x}^k$ in the "don't cares"

positions of all the compressed keys with "don't cares" in the set. Thus we end up with:

	7	6	5	4	3	2	1	0
7	0	1	0	1	1	1	0	0
6	0	1	0	1	0	1	1	0
5	0	1	0	1	0	1	0	1
4	0	1	0	1	0	1	0	0
3	0	1	0	0	0	1	0	0
2	0	0	1	1	0	1	0	0
1	0	0	1	1	0	0	0	0
0	0	0	0	1	0	1	0	0

Table 3.13: $branch \vee (\hat{x}_m^k \wedge free)$ in a word laid in a $k \times k$ matrix

e) Now, match returns $\text{rank}(\hat{x}_m, branch \vee (\hat{x}_m^k \wedge free))$ via Rank Lemma 1. In table 3.13, we can see that $\hat{x}_m = 0101\ 0100_2$ is larger than the keys up to row 3, meaning that its rank is 4 (the highlighted row). Thus we have:

$$\text{match}(x_m) = 4$$

7. We finish by returning the rank of x, which in this case:

$$\begin{aligned} i_0 &= \text{match}(x_m) = 4 \\ \implies \text{rank}(x) &= i_0 = 4 \end{aligned}$$

3.6 Inserting while Maintaining the Compressed Keys with "Don't Cares"

Goal Having the implementation from Section 3.5 as a starting point, the goal is to improve the insert method such that *branch* and $free$ are maintained using the algorithm described in [Pătrașcu and Thorup, 2014] for this effect. Rank queries keep their $O(k)$, and select queries keep their $O(1)$ time. Inserting is bound by the running time of one of its subroutines, which takes $O(k)$ time (the running time of this operation is addressed at a later section of [Pătrașcu and Thorup, 2014]). Apart from that, all the insert algorithm steps take $O(1)$ time. Delete is kept unaltered.

In this section, we will address how to maintain the compressed keys with "don't cares" when inserting a new key, as described in section *Inserting a key* of the [Pătrașcu and Thorup, 2014] paper. We take another incremental on the implementations from Sections 3.4 and 3.5, keeping most of their details with the exception of the `insert` method. To this extent, we also add some new helper methods. The resulting implementation can be found in the file specified in figure 3.9.

Figure 3.9: Location of the `DynamicFusionNodeDontCaresInsert.java` file in the folder structure

The algorithm for inserting a key x in the set while maintaining *branch* and $free$ entails:

- Rank via "don't cares", including its subroutines, as introduced and described in Section 3.5.
- Assessing if introducing a new key adds a new branching bit, e.g., a new column in *branch* and $free$.
- Computing the rank of new significant positions. In other words, if adding a new key implies adding a new column j in *branch* and $free$ because it is a new significant position, then we need to know how many set bits are there in the $compressingKey$ up to index j.
- Matrix operations in *branch* and $free$ such as:
  - Adding rows and columns.
  - Updating rows and columns.
  - Setting and deleting ranges of bits or particular bits in rows or columns.

- All of the operations described in the `insert (final long x)` method of Section 3.4.3.

3.6.1 Fields

All the class and instance variables remain unchanged in comparison with Section 3.5.1. They are:

```
1  private static final int k = 8;
2  private static final int ceilLgK = (int) Math.ceil(Math.log10(k) / Math.log10(2));
3  private static final long M = Util.M(k, k * k);
4  private final long[] key = new long[k];
5  private long index;
6  private int bKey;
7  private int n;
8  private long compressingKey;
9  private long branch;
10 private long free;
```

3.6.2 Helper Methods

The following helper methods, which have already been implemented in 3.4.2 and 3.5.2 are kept:

- `firstEmptySlot()`
- `fillSlot(final int j)`
- `vacantSlot(int j)`
- `getIndex(final long i)`
- The overloaded methods `updateIndex`
- `updateCompressingKey()`
- `compress(final long x)`
- `long compressedKeys()`
- `updateFree(final long compressedKeys)`
- `dontCares`
- `match(x)`
- `dontCaresRank(final long x)`

The following are helper methods introduced in this implementation:

- `matrixM(final int h)` returns a word which when interpreted as a $k \times k$ matrix has only column h set. Let $h = 5$, then `matrixM(5)` returns:

	7	6	5	4	3	2	1	0
7	0	0	1	0	0	0	0	0
6	0	0	1	0	0	0	0	0
5	0	0	1	0	0	0	0	0
4	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	0

Table 3.14: `matrixM(5)`

- `matrixMColumnRange(final int lo, final int hi)` returns a word which when interpreted as a $k \times k$ matrix will have the bits in the range of columns between lo (inclusive) and hi (inclusive) set. Let $lo = 3$ and $hi = 6$, then `matrixMColumnRange(3, 6)` returns:

	7	6	5	4	3	2	1	0
7	0	1	1	1	1	0	0	0
6	0	1	1	1	1	0	0	0
5	0	1	1	1	1	0	0	0
4	0	1	1	1	1	0	0	0
3	0	1	1	1	1	0	0	0
2	0	1	1	1	1	0	0	0
1	0	1	1	1	1	0	0	0
0	0	1	1	1	1	0	0	0

Table 3.15: `matrixMColumnRange(3, 6)`

- `matrixMRowRange(final int lo, final int hi)` returns a word which when interpreted as a $k \times k$ matrix will have the bits in the range of rows between lo (inclusive) and hi (inclusive) set. Let $lo = 3$ and $hi = 6$, then `matrixMRowRange(3, 6)` returns:

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Table 3.16: `matrixMRowRange(3, 6)`

- `insertAndInitializeColumn(final int h)` introduces a new column in *branch* and $free$, initializing it to its default bit value (0 in *branch* and 1 in $free$). Let *branch* and $free$ be the following:

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	1
2	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0

(a) *branch*

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	1	1	1	1	1	0	1	0
2	1	1	1	1	1	0	1	0
1	1	1	1	1	1	0	0	1
0	1	1	1	1	1	0	0	1

(b) $free$

Figure 3.10: Examples of *branch* and $free$

A call to `insertAndInitializeColumn(2)` will move all columns larger than 2 one position to the left and set values in the new column them to their respective default. In the tables from figure 3.11 we can see the new column highlighted.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0	1
2	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0

(a) *branch*

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	1	1	1	1	0	1	1	0
2	1	1	1	1	0	1	1	0
1	1	1	1	1	0	1	0	1
0	1	1	1	1	0	1	0	1

(b) $free$

Figure 3.11: *branch* and $free$ after insertion of column at position 2

- `insertRow(final int rank)` introduces a new row in *branch* and $free$ at position *rank*. All rows with rank larger than *rank* will occupy their respective next row. Calling `insertRow(2)` while having *branch* and $free$ as the tables from figure 3.10 as instance variables will result on the following:

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	1
3	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0

(a) *branch*

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	0	1	0
3	1	1	1	1	1	0	1	0
2	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	1
0	1	1	1	1	1	0	0	1

(b) $free$

Figure 3.12: *branch* and $free$ after insertion of row at position 2

3.6.3 Implementation of the Interface Methods

The following interface methods suffered no changes and serve the same purpose as the implementation from Section 3.5.3:

- `select(final long rank)`
- `size()`
- `delete(final long x)`[4]
- `rank(final long x)`
- `reset()`

Based on the description of *Inserting a key* of [Pătraşcu and Thorup, 2014], we implement `insert(final long x)` method the following way:

1. A local variable *rank* is initialized to 0.
2. If the set is not empty:
   a) We compute match(x) and store it in a local variable i.
   b) We query select(i) and and store it in a local variable y.

[4] At this stage, we still maintain the compressed keys with "don't cares" naively after removing a key from the set.

c) We use the unsigned comparator from the standard library, `compareUnsigned`, to assess if the keys, x and y, are equal, or if one is larger than the other. This is stored in a local variable *comp*.

d) If $comp = 0$, then $x = y$, which means that x is already present in the set. In this case, the method returns, introducing no changes to the contents of the data structure.

e) At this stage, we know that x is a new key, so before inserting it, we perform another check to see if the data structure has reached its capacity limit, k, throwing an exception in this case.

f) We find the branching bit between x and y, $\text{msb}(x \oplus y)$, and store it in a local variable, j.

g) We compute the rank of j in $compressingKey$ by masking all the bits at indices larger than j and calling the standard library function, `bitCount`, on the masked $compressingKey$. This is stored in a local variable, h. At this stage, this operation is done naively, as Pătraşcu and Thorup address this operation in a later section.

h) We compute i_0 and i_1, given by the calls $\text{match}(x \wedge \neg((1 \ll j) - 1))$ and $\text{match}(x \vee ((1 \ll j) - 1))$. These masks and queries have been explained in 3.5.4.3 and they will correspond to rows in *branch* and $free$, e.g. the rank of actual keys in the set.

i) Just like in 3.5.4.3, we compute the rank of x based on how its value compares with y, updating the local variable *rank* accordingly.

j) We compute and store locally `matrixM(h)`.

k) If j was not already a significant bit:

- We mark it as a significant bit in $compressingKey$.
- We update *branch* and $free$ with a call to `insertAndInitializeColumn(h)`.

l) We call `matrixMRowRange(i_0, i_1)` and intersect it with `matrixM(h)`. This will produce a mask that has column h in the range of rows between i_0 and i_1 set to 1, and every other rows and columns set to 0.

m) We use the mask from step 2l to mark column h in the range of rows between i_0 and i_1 as a "care" position. This entails setting those positions to 0 in $free$ and storing the bit value of position j in y in *branch*.

n) We insert a new row in *branch* and $free$ with a call to `insertRow(rank)`.

o) The local variable i, which stores the rank of y, is updated. This means that

if $x < y$, then after inserting x, the rank of y will have incremented. In such a case, we increment i to reflect that.

p) The last steps consist of updating row $rank$, which stores the $\hat{x}^?$, with the appropriate values. These are:

- In $branch$:
  - The values of the bits in positions between 0 and $h-1$ are set to 0:
$$branch\langle rank, 0 \dots h-1 \rangle_{k\times 1} \coloneqq 0^h$$
  - We read $x\langle j \rangle_1$ and store it at position h:
$$branch\langle rank, h \rangle_{k\times 1} \coloneqq x\langle j \rangle_1$$
  - We copy the bit values in positions between $h+1$ and $k-1$ from the $\hat{y}^?$ to the same positions:
$$branch\langle rank, h+1 \dots k-1 \rangle_{k\times 1} \coloneqq branch\langle i, h+1 \dots k-1 \rangle_{k\times 1}$$
- In $free$:
  - The values of the bits in positions between 0 and $h-1$ are set to 1:
$$free\langle rank, 0 \dots h-1 \rangle_{k\times 1} \coloneqq 1^h$$
  - We set position h to 0:
$$free\langle rank, h \rangle_{k\times 1} \coloneqq 0$$
  - We copy the bit values in positions between $h+1$ and $k-1$ from the $\hat{y}^?$ to the same positions:
$$free\langle rank, h+1 \dots k-1 \rangle_{k\times 1} \coloneqq free\langle i, h+1 \dots k-1 \rangle_{k\times 1}$$

3. We finish by computing the steps described in 3.4.3, effectively insert x in the set.

The overall running time of this algorithm is bound by the match subroutines (steps 2a and 2h) and finding the rank of the new branching bit (step 2g): these take $O(3 \cdot k) = O(k)$. Otherwise, all the remaining steps taken by this algorithm run in $O(1)$ time.

3.6.4 Example

We will now insert the key $x = 1001\ 0101\ 0001_2$ in a set containing the keys from table 3.17, showcasing all the steps. For brevity, the chosen keys only have set bits up to position 11 such that the 52 most significant bits do not influence any of the steps in the algorithm and can safely be omitted.

	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	0	1	0
1	0	1	1	0	1	0	1	1	1	1	1	1
2	1	0	1	0	1	1	0	0	1	1	0	0
3	1	0	1	0	1	1	0	1	0	1	1	0

Table 3.17: Binary representation of the keys present in the data structure. The table also shows their rank (on the first column) and the bit values at every index (on the first row) up to position 11

This set of keys will produce the following compressing key:

$$compressingKey = 1100\ 0001\ 0000_2$$

The compressed keys with "don't cares" stored in *branch* and $free$ are shown in the tables of figure 3.13.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	1
2	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0

(a) *branch*

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	1	1	1	1	1	0	1	0
2	1	1	1	1	1	0	1	0
1	1	1	1	1	1	0	0	1
0	1	1	1	1	1	0	0	1

(b) $free$

Figure 3.13: Resulting *branch* and $free$ after the compression of keys with "don't cares" from table 3.17

Inserting x in the given set is comprised of the following:

- We start by initializing a local variable $rank$, which will be used to store the rank of our query, x. Its initial value will be 0.
- If the set is not empty (which for the moment it is not), then we have to run the steps from Sections 3.6.4.1 to 3.6.4.2; otherwise we skip to 3.6.4.3. This is because when the set is empty, there is no need for updating the compressed keys with "don't cares".

3.6.4.1 Updating the Range of Keys whose Compressed Key match the Insertion Key

1. We start by computing some variables that are used for computing the rank of x along with i_0 and i_1, which specify a range of compressed keys with "don't cares" (rows in *branch* and $free$) which have to be updated upon the present insertion:

   a) We run match(x) and store it in a local variable i:

   $$x = 1001\ 0101\ 0001_2$$
   $$\text{---}$$
   $$i \coloneqq \text{match}(x) = 3$$

   b) We query select(i) and store it in a local variable y:

   $$i = 3$$
   $$\text{---}$$
   $$y \coloneqq \text{select}(i) = 1010\ 1101\ 0110_2$$

   c) We compare x and y with the `compareUnsigned` function from the standard library, storing the result in a local variable *comp*.

   $$x = 1001\ 0101\ 0001_2$$
   $$y = 1010\ 1101\ 0110_2$$
   $$\text{---}$$
   $$x < y \implies comp = -1$$

   Since $comp \neq 0$, then x is not in the set, so we proceed.

   d) We check also if the set has room for x with `size() < k`. In this case, it is not, so we proceed.

   e) To find the branching bit between x and y, we compute msb($x \oplus y$) and store it in a local variable, j.

   $$x = 10\underline{0}1\ 0101\ 0001_2$$
   $$y = 10\underline{1}0\ 1101\ 0110_2$$
   $$\text{---}$$
   $$x \oplus y = 00\underline{1}1\ 1000\ 0111_2$$
   $$\text{---}$$
   $$j \coloneqq \text{msb}(x \oplus y) = 9$$

   f) We need to find the rank of j in the *compressingKey* and store it in a local variable, h. This variable will specify the column where we will write the branching bit to. We mask the bits that are at higher positions than j and call

the standard library function, `bitCount`, on that result:

$$\begin{aligned} j &= 9 \\ compressingKey &= 1100\ \underline{0001}\ \underline{0000}_2 \\ mask &= (1 \ll j) - 1 = 0001\ 1111\ 1111_2 \\ \hline h &\coloneqq \texttt{bitCount}(compressingKey \wedge mask) = 1 \end{aligned}$$

g) We compute i_0 and i_1, which are computed by calling match$(x \wedge \neg((1 \ll j) - 1))$ and match$(x \vee ((1 \ll j) - 1))$:

$$\begin{aligned} j &= 9 \\ x &= 1001\ \underline{0101}\ \underline{0001}_2 \\ \hline \neg((1 \ll j) - 1) &= 1110\ 0000\ 0000_2 \\ x \wedge \neg((1 \ll j) - 1) &= 1000\ \underline{0000}\ \underline{0000}_2 \\ i_0 &\coloneqq \text{match}(x \wedge \neg((1 \ll j) - 1)) = 2 \\ \hline (1 \ll j) - 1 &= 0001\ 1111\ 1111_2 \\ x \vee ((1 \ll j) - 1) &= 1001\ \underline{1111}\ \underline{1111}_2 \\ i_1 &\coloneqq \text{match}(x \vee ((1 \ll j) - 1)) = 3 \end{aligned}$$

h) We compute rank(x). If $x < y$, then rank$(x) = i_0$; otherwise, it is $i_1 + 1$:

$$\begin{aligned} i_0 &= 2 \\ i_1 &= 3 \\ \hline comp = -1 &\implies \text{rank}(x) = i_0 \\ rank &\coloneqq 2 \end{aligned}$$

2. We compute the matrix mask needed for, among other things, adding a new column h in $branch$ and $free$, storing the result in a local variable $matrixM_h$. This results in table 3.18, where we can also see column h highlighted.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0
5	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0

Table 3.18: $matrixM_h$

3. We check if j was already a significant position:

$$\begin{aligned} j &= 9 \\ compressingKey &= 11\underline{0}0\ 0001\ 0000_2 \\ \hline \text{bit}(j, compressingKey) &= 0 \end{aligned}$$

Since it was not:

- We mark it as a significant position by setting bit j in $compressingKey$:

$$\begin{aligned} j &= 9 \\ compressingKey &= 11\underline{0}0\ 0001\ 0000_2 \\ \hline compressingKey \coloneqq \text{setBit}(j, compressingKey) &= 11\underline{1}0\ 0001\ 0000_2 \end{aligned}$$

- We update $branch$ and $free$ to include the new column h. The `insertAndInitializeColumn(h)` method is called thus altering $branch$ and $free$ as we see in the tables from figure 3.14.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0	1
2	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0

(a) $branch$

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	1	1	1	1	0	1	1	0
2	1	1	1	1	0	1	1	0
1	1	1	1	1	0	0	1	1
0	1	1	1	1	0	0	1	1

(b) $free$

Figure 3.14: Insertion of column h in $branch$ and $free$

4. We call the function `matrixMRowRange(`i_0`,` i_1`)` which returns a mask comprised of the range of rows $\{i_0, \ldots, i_1\}$. Bitwise $\wedge$ that mask with $matrixM_h$ computes a mask

where only the bits of rows $\{i_0, \ldots, i_1\}$ and column h are set. These are the bits that of the compressed keys that need to be updated due the insertion of the new key x. Thus the mask needed for this effect is the one from table 3.19.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Table 3.19: $matrixM_h^{i_0:i_1}$, which resulted from the intersection of the column matrix mask, $matrixM_h$, and the row matrix mask, $matrixM^{i_0:i_1}$

5. Now we need to update *branch* and $free$ accordingly. We know that we care for the highlighted positions in table 3.19 because i_0 and i_1 have matched x, so we set these positions to 0 in $free$. The value to store at those positions in *branch* has to be read from y:

$$y = 10\underline{1}0\ 1101\ 0110_2$$
$$j = 9$$
$$\text{bit}(j, y) = 1$$

We see that the bit value is 1, thus this will be the value we will update *branch* with. The result after this update can be seen in the tables from figure 3.15.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	1
2	0	0	0	0	1	0	1	0
1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0

(a) *branch*

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
3	1	1	1	1	0	1	0	0
2	1	1	1	1	0	1	0	0
1	1	1	1	1	0	0	1	1
0	1	1	1	1	0	0	1	1

(b) $free$

Figure 3.15: *branch* and $free$ after updating column h of the $i_0 : i_1$ compressed keys

3.6.4.2 Updating the Instance Variables with the new Compressed Key

We now have to insert a row for the new key in *branch* and $free$ and write its corresponding values. This entails the following:

1. We call `insertRow(rank)`, making row for $\hat{x}^?$ in row *rank* of *branch* and $free$. The tables from figure 3.16 show the result of this operation.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	1	1
3	0	0	0	0	1	0	1	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0

(a) *branch*

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	0	1	0	0
3	1	1	1	1	0	1	0	0
2	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	1	1
0	1	1	1	1	0	0	1	1

(b) $free$

Figure 3.16: Insertion of row *rank* in *branch* and $free$

2. Some of the values of $\hat{x}^?$ will be copied from $\hat{y}^?$. For this reason we have to update i, the rank of y, such that the needed values are copied from the right key.

$$\begin{aligned} i &= 3 \\ x &= 1001\ 0101\ 0001_2 \\ y &= 1010\ 1101\ 0110_2 \\ \hline x < y \implies i &:= 4 \end{aligned}$$

3. Lastly, we write $\hat{x}^?$ to *branch* and $free$.

   - Row *rank* of *branch* will be:

$$\begin{aligned} \textbf{(a)}\quad branch\langle rank, 0 \ldots h-1\rangle_{k\times 1} &:= 0^h \\ \textbf{(b)}\quad branch\langle rank, h\rangle_{k\times 1} &:= x\langle j\rangle_1 \\ \textbf{(c)}\quad branch\langle rank, h+1 \ldots k-1\rangle_{k\times 1} &:= branch\langle i, h+1 \ldots k-1\rangle_{k\times 1} \end{aligned}$$

$$branch\langle rank\rangle_k := \underbrace{000010}_{\textbf{(c)}}\underbrace{0}_{\textbf{(b)}}\underbrace{0}_{\textbf{(a)}}$$

   - Row *rank* of $free$ will be:

$$\begin{aligned} \textbf{(a)}\quad free\langle rank, 0 \ldots h-1\rangle_{k\times 1} &:= 1^h \\ \textbf{(b)}\quad free\langle rank, h\rangle_{k\times 1} &:= 0 \\ \textbf{(c)}\quad free\langle rank, h+1 \ldots k-1\rangle_{k\times 1} &:= free\langle i, h+1 \ldots k-1\rangle_{k\times 1} \end{aligned}$$

$$free\langle rank\rangle_k := \underbrace{111101}_{\textbf{(c)}}\underbrace{0}_{\textbf{(b)}}\underbrace{1}_{\textbf{(a)}}$$

We can see the updates described above in the tables from figure 3.17.

	7	6	5	4	3	2	1	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	1	1
3	0	0	0	0	1	0	1	0
2	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0

(a) *branch*

	7	6	5	4	3	2	1	0
7	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
4	1	1	1	1	0	1	0	0
3	1	1	1	1	0	1	0	0
2	1	1	1	1	0	1	0	1
1	1	1	1	1	0	0	1	1
0	1	1	1	1	0	0	1	1

(b) *free*

Figure 3.17: *branch* and $free$ after the insertion of $\hat{x}^?$

3.6.4.3 Storing the Key

With *branch* and $free$ now updated, we proceed with remaining insertion operations, which have been described in Section 3.4.3.

4 Validation

This chapter describes the tests that have been implemented and conducted in order to assert the soundness of the implementations discussed in Chapter 3. The implemented test classes can be found in the folder shown in figure 4.1.

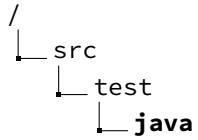

Figure 4.1: Location of the testing classes folder

Validation summary tables, which include the test results, together with the range or size of the test instances, have been included. The symbol ✓ in those tables signifies a passed test with the specified parameters.

4.1 `Util` Class Tests

All the tests concerning the `Util` class can be found in the `UtilTest.java` file, whose path is shown in figure 4.2.

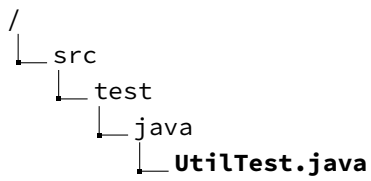

Figure 4.2: Location of the file containing the testing class of the `Util` class

4.1.1 msb Series Tests

The `Util` class contains many different implementations of the msb(x) operation. Since Java's standard library features the `numberOfLeadingZeros` function, we can easily test the different msb(x) implementations. Converting the result from the standard library function is done by subtracting its return value to $w - 1$.

Taking as an example the test on the `msbConstant()` implementation, for such a test to pass all instances in a given range have to assert that `Long.SIZE - 1 -`

`Long.numberOfLeadingZeros(i)` and `Util.msbConstant(i)` are the same. Tests on the other implementations differ only either on w (which can be either 32-bit `Integer` or 64-bit `Long`) and the function that is called, and each of them is named after the function they are testing, e.g. `msb32Obvious()` tests the 32-bit version the naive msb(x) algorithm.

4.1.1.1 Validation Results

Range lower bound	−100 000 000
Range upper bound	100 000 000
Function	Result
`msb32Obvious()`	✓
`msb32LookupDistributedOutput()`	✓
`msb32LookupDistributedInput()`	✓
`msb32Constant()`	✓

Table 4.1: Validation summary of the `msb32` functions

Range lower bound	−10 000 000 + `Integer.MIN_VALUE`
Range upper bound	10 000 000 + `Integer.MAX_VALUE`
Function	Result
`msb64Obvious()`	✓
`msb64LookupDistributedOutput()`	✓
`msb64LookupDistributedInput()`	✓
`msb64Constant()`	✓

Table 4.2: Validation summary of the `msb64` functions

4.1.2 `splitLong` and `mergeInts`

The `splitMerge()` method tests the `splitLong` and `mergeInts` methods by evaluating if the following property holds for a determined range of 64-bit keys: `key == mergeInts(splitLong(key))`.

4.1.2.1 Validation Results

Range lower bound	−100 000 000 000
Range upper bound	100 000 000 000
Function	Result
`splitMerge()`	✓

Table 4.3: Validation summary of the `splitLong` and `mergeInts` functions

4.1.3 Fields of Words Tests

The `getField` and `setField` methods are tested in the `setAndGetField32()` and `setAndGetField64()` methods (the latter differ solely on the size of the word where the fields are to be written to). A pass of the test consists of:

1. Varying b (being b the size of each field) between 1 and $w - 1$.

2. For each value b takes, the number of fields to be stored in A, m, is computed as a function of b. The goal is to fit as many fields as possible in A, having b and w as constraints. The number of fields to store in A is computed with $m = w/b$.

3. Then, m keys of b size are pseudo-randomly generated. Each time a key is generated, A is shifted to the left by b bits, and the key is stored both in an auxiliary array and in A with the `setField` function.

4. When all m keys have been generated and inserted, the auxiliary array is iterated, and at each iteration i, it asserts that the key at index i in the auxiliary array is the same as the one returned by the `getField` function.

4.1.3.1 Validation Results

Initial *seed*	42
#passes	100 000 000
Function	Result
`setAndGetField32()`	✓
`setAndGetField64()`	✓

Table 4.4: Validation summary of the setter and getter field functions

4.1.4 Test for Rank Lemma 1

The `rankLemma1` function is tested in a method with the same name. A pass in this test consists of:

1. Varying b (size in bits of each key). This range is specified in the fields of the `UtilTest` class by the variables `int loB` and `int hiB`.

2. For each value b takes, the number of keys to be stored in A, m, is computed by taking the minimum between 2^b and w/b. This is because the method requires that all keys stored in A to be distinct (and all combinations of keys of size b are given by 2^b), but also A can only store up to w/b keys.

3. We produce m distinct keys by using a pseudo-random generator and inserting them in a set. We stop once the size of the set is m.

4. An additional key x is generated. The rank of x will be computed and used later as one of the last checks of this test.

5. The keys are then copied to a list, where they are sorted in descending order.

6. The list is then iterated from the largest key to the smallest. Each iteration consists of inserting the current key in A. That key is also compared with x in order to compute the rank of x, which is determined by the index of the current key in the list.

7. The list is iterated once more in order to assess if the rank of each key in the list agrees with the rank computed with `rankLemma1` function.

8. Lastly, the rank of x, which has been computed before, is compared with its rank computed via the `rankLemma1` function.

4.1.4.1 Validation Results

Initial *seed*	42
#passes	100 000 000
`loB`	3
`hiB`	10
Function	Result
`rankLemma1()`	✓

Table 4.5: Validation summary of the Rank Lemma 1 function

4.2 Integer Data Structure tests

Each implementing class of the `RankSelectPredecessorUpdate` interface has a corresponding test class. The file names of those classes are the highlighted in figure 4.3.

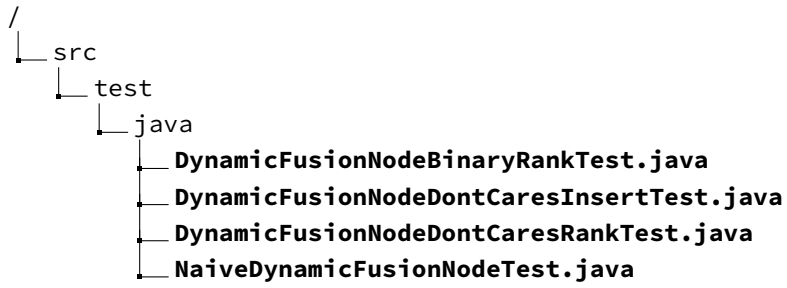

Figure 4.3: Location of the testing class files of the `RankSelectPredecessorUpdate` implementing classes

4.2.1 Testing Helper Class

In order to avoid duplicate code, a helper class, denoted `RankSelectPredecessorUpdateTest`, was implemented. It can be found in the same folder as the remaining test classes, as shown in figure 4.3.

Each of the implementation's testing classes creates an instance of the helper class and uses its methods to test its respective implementation.

The `RankSelectPredecessorUpdateTest` constructor takes the following parameters:

- `long seed`. This seed is used as a parameter for instantiating a pseudo-random generator from the Java standard library — `java.util.Random`. This instance will be later used to produce data for the test cases, such as seeds for passes (explained below), which in turn will produce keys to be used in the tests.

- `int passes`. Some tests can be executed in more than one pass. A pass in a particular test consists of generating data with its corresponding seed and running the test with that data When $passes > 1$, distinct $\#passes$ seeds are generated, resulting in different pseudo-random values, and the test is run `passes` number of times.

- `int numKeys`. This parameter defines the size of the data set to be generated. It is particularly important for testing instances of `DynamicFusionNode`, which size cannot exceed k.

4.2.2 `insertAndMemberSmallTest()`

The methods tested in this test are `insert` and `member`. A small set of predetermined keys is inserted in the set, and then `member` is called on the set with each of those keys.

4.2.3 `smallCorrectnessTest()`

An instance of a `RankSelectPredecessorUpdate` implementation is instantiated and keys are inserted such that after these insertions $S = \{10, 12, 42, -1337, -42\}$.

It is defined that:

- Select queries can range between 0 and $|S| - 1$. Should any query fall outside this range, then `null` is returned, meaning, no result.
- select$(0) = \min\{y \in S\}$.
- select$(|S| - 1) = \max\{y \in S\}$.
- Rank queries can be any w-bit integer, and their possible range of results $\in \left[0, |S|\right]$
- Any given predecessor query returns the largest key in the subset of keys that are **strictly** smaller than the query, otherwise `null`.
- Whereas a successor query returns the queried key if present, otherwise the smallest key in the subset of keys that are larger than the query if any (and in this case `null` is returned).

With the given set and the above-mentioned rules, a small test is performed. Table 4.6 shows the expressions to be evaluated, and all of them must evaluate to `true` for the test to pass.

Key	Member	Rank	Select	Predecessor	Successor
10	`true`	`rank(10) == 0`	`select(0) == 10`	`pred(10) == null`	`succ(10) == 10`
11	`false`	`rank(11) == 1`	`select(1) == 12`	`pred(11) == 10`	`succ(11) == 12`
12	`true`	`rank(12) == 1`	`select(1) == 12`	`pred(12) == 10`	`succ(12) == 12`
42	`true`	`rank(42) == 2`	`select(2) == 42`	`pred(42) == 12`	`succ(42) == 42`
−1337	`true`	`rank(-1337) == 3`	`select(3) == -1337`	`pred(-1337) == 42`	`succ(-1337) == -1337`
−1000	`false`	`rank(-1000) == 4`	`select(4) == -42`	`pred(-1000) == -1337`	`succ(-1000) == -42`
−42	`true`	`rank(-42) == 4`	`select(4) == -42`	`pred(-42) == -1337`	`succ(-42) == -42`
−1	`false`	`rank(-1) == 5`	`select(5) == null`	`pred(-1) == -42`	`succ(-1) == null`

Table 4.6: Small correctness tests

4.2.4 `insertThenMemberTest()`

The methods tested in this test are `insert` and `member`, and it can be executed in passes. It consists of:

1. Iterating through all the pseudo-randomly-generated keys and inserting them all in the instance to be tested.
2. Iterating in random order through all the keys and asserting $key \in S$.

4.2.5 `insertThenDeleteRangeOfKeysTest()`

The methods tested in this test are `insert` and `member`. It consists of iterating the whole range (which goes from 0 to `numKeys`), where each iteration `i` consists of:

1. Asserting that the key, `i` is not in the set by calling `member` on the set with the key.
2. Inserting the key `i` in the set.
3. Asserting that the key `i` is in the set.

Should the number of keys exceed 9, then every 1/10 of `numKeys` the data structure is reset, and all keys are removed.

4.2.6 `insertThenDeleteRandomKeysTest()`

The methods tested in this test are `insert` and `delete`, and it can be executed in passes. After inserting all the pseudo-randomly-generated keys in the set, the keys are iterated in random order. Each iteration consists of:

1. Asserting that $key \in S$.
2. Removing the key.
3. Asserting that $key \notin S$.

4.2.7 `deleteTest()`

This test aims to assert that only when deleting an existing key, the set's cardinality is altered. It tests the methods `delete` and `size()`, and it can be executed in passes. At each pass:

1. Pseudo-random keys are generated, and delete on the set is called with that key.
2. If the key was in the set, then the size must have decreased; otherwise, it must remain the same.

4.2.8 `sizeTest()`

The methods tested in this test are `insert`, `size` and `delete` and it can be executed in passes. Each pass consists of:

1. Iterating the pseudo-randomly-generated keys in random order. At each iteration, the key is inserted, and it is asserted that `size` has increased by 1.

2. After all the keys have been inserted, the keys are iterated in random order once more, and at each iteration, the key is removed, and it is asserted that `size` has decreased by 1.

4.2.9 `growingRankTest()`

This test aims at ensuring that the following property holds for all the keys in the set: the rank of keys in sorted order is a monotone increasing function. It can be executed in passes, and it works the following way:

- After inserting all the keys in the set, a copy of those keys is kept on a `TreeSet` (a set that keeps its values in sorted order) such that we can iterate them in their sorted order.
- A counter `i` is kept and initialized as 0. It is incremented at each iteration.
- The test then asserts that the key's rank is the same as the number of the current iteration. Note that we can assume that this condition must hold because the keys are iterated in sorted order.

4.2.10 `selectOfRankTest()`

This test aims at ensuring that the following property holds for all the keys in the set: if a key is present in the set, knowing its rank and them querying for select of that must return the key. It can be executed in passes, and it works by iterating through the pseudo-randomly-generated keys, and at each iteration, it is asserted that `key == select(rank(key))`.

4.2.11 `rankOfSelectTest()`

This test aims at ensuring that the following property holds for all the keys in the set: the rank of select a query is the query. It can be executed in passes, and it works by iterating from `0` to `numKeys`. At each iteration `i`, it is asserted that `i == rank(select(i))`.

4.2.12 Validation Results

This section summarizes the results from the tests described in Sections from 4.2.2 to 4.2.11 against the implementations described in Sections from 3.3 to 3.6. The results can be seen in table 4.8, where the correspondence between implementation and abbreviation is done in table 4.7. In table 4.8, the ✓symbol signifies a passed test. We can see that the highlighted implementations pass all tests, using as many keys as each implementation allows. Each test was repeated with a different set of keys $\#passes$ times.

Implementation	Described in Section	Abbreviation
`NaiveDynamicFusionNode`	3.3	**[A]**
`DynamicFusionNodeBinaryRank`	3.4	**[B]**
`DynamicFusionNodeDontCaresRank`	3.5	**[C]**
`DynamicFusionNodeDontCaresInsert`	3.6	**[D]**

Table 4.7: Correspondence table

Implementation	**[A]**	**[B]**	**[C]**	**[D]**
Initial *seed*	42	42	42	42
$\#passes$	10 000	100 000	100 000	100 000
$\#numKeys$	1 000	16	8	8
`insertAndMemberSmallTest()`	✓	✓	✓	✓
`smallCorrectnessTest()`	✓	✓	✓	✓
`insertThenMemberTest()`	✓	✓	✓	✓
`insertThenDeleteRangeOfKeysTest()`	✓	✓	✓	✓
`insertThenDeleteRandomKeysTest()`	✓	✓	✓	✓
`deleteTest()`	✓	✓	✓	✓
`sizeTest()`	✓	✓	✓	✓
`growingRankTest()`	✓	✓	✓	✓
`selectOfRankTest()`	✓	✓	✓	✓
`rankOfSelectTest()`	✓	✓	✓	✓

Table 4.8: Validation summary of the *Dynamic Fusion Node* implementations

5 Conclusion

We conclude this report by highlighting the contributions of this project:

- Within the scope defined in Section 1.4, we have explained the data structure presented in [Pătraşcu and Thorup, 2014], together with helper functions and algorithms. This has been complemented with illustrative examples.

- We have implemented and documented the data structure, making it publicly available for future work.

- We have implemented and included correctness tests in the repository. These attest to the correctness of the implementations we present and that were designed such that they can be used to assess the correctness of further implementations.

- We have listed other data structures that are of interest within the context of the dynamic predecessor problem, pointing to future work in the form of benchmarks.

5.1 Final Remarks

- There is a fine balance between space and time consumption when developing and implementing sub-logarithmic data structures and algorithms. For example, van Emde Boas trees is a very fast data structure, but with a big space consumption drawback. Fusion Trees improve on space consumption, but updates take excessive time. Dynamic Fusion Trees, introduced in this project, seem to find the perfect balance between space and time consumption.

- As mentioned in Section 2.1.3, theoretical bounds have the potential to hide big constants.

  This project has been about implementing algorithms that use a constant number of operations to compute the result, assuming that these are an improvement in comparison with logarithmic algorithms (such as binary search). We want to highlight that despite the fact that it is plausible to implement the *Dynamic Fusion Node* using only $O(1)$ operations of the word RAM model, the hidden constant might not be negligible.

  For instance, the `DynamicFusionNodeBinaryRank` implementation from Section 3.4 computes rank(x) with binary search. In this particular implementation, and because we set $w = 64$, allowing us to store at most $k = 16$ keys, we know that the worst-case

running time of a rank(x) query will be when the node is full, e.g., $n = k = 16$. Computing $\log_2 16 = 4$. We conclude that, when this implementation is full, it will take at most 4 iterations for rank to compute the answer.

The `DynamicFusionNodeDontCaresRank` implementation from Section 3.5 uses $O(1)$ operations to compute a rank(x) query. The constant number of operations used by this method is larger than the 4 iterations taken by the binary search rank algorithm from Section 3.4. The `match` subroutine alone, used by the rank with "don't cares" algorithm, uses 5 word RAM operations, and the whole rank algorithm needs to run this subroutine at least once; and if the key is not in the set, it needs to run it once again. That is already 10 operations, not counting with some other necessary subroutines such as the select query and finding the branching bit, j.

Our conclusion is that $w = 64$ is potentially a very small word size to take advantage of the running time of these algorithms.

5.2 Future Work

In this section, we leave some suggestions on how further work on this topic can be conducted. We have split these into three main categories:

1. Implementation, which covers specific data structures or algorithms.
2. Optimization, which entails improving the present code.
3. Benchmarking, which points to other data structures that either solve partially or totally the dynamic predecessor

5.2.1 Implementation

Delete methods The implementation featured in Section 3.6 implements the `delete` method naively, but all the ingredients needed to implement it while adhering to *Inserting a key* section of [Pătraşcu and Thorup, 2014] are already present in the implementation.

Key compression in constant time One of the bottlenecks of implementations from Sections 3.5 and 3.6 is how the compressed keys with "don't cares" are maintained. Specifically, the next iterative step in the implementation from Section 3.6 is to implement functions that compute the compressed keys in $O(1)$ time, as explained in Chapters [3.2 – 3.3] of [Pătraşcu and Thorup, 2014].

Dynamic Fusion Tree After enabling all the operations at the node level in $O(1)$ time with the previous steps' implementation, all that remains to complete the implementation

is to implement a k-ary tree using a *Dynamic Fusion Node* as its node. This is covered in Chapter 4 of [Pătrașcu and Thorup, 2014].

Non-recursive implementation Chapter 4 of [Pătrașcu and Thorup, 2014] mentions that once the dynamic fusion tree is implemented, the rank operation on a tree is given by a recursive function. Recursion in Java can be slow [Shirazi, 2003], and for this reason, a non-recursive alternative is preferred.

5.2.2 Benchmarking

msb functions It would be interesting to see how the different msb functions implemented in this project compare to each other in terms of time.

Dynamic Predecessor Problem Data Structures Once the *Dynamic Fusion Tree* is fully implemented, the next logical step would be to benchmark it with other data structures that solve, either partially or entirely, the dynamic predecessor problem. These would be the data structures mentioned in Section 2.2.

5.2.3 Optimization

Space Space consumption can be improved by, for instance, avoiding and combining some of the fields. When $k = 8$, $bKey$ would only use 8 of the 64 allocated bits, and $index$ would use $3 \times 8 = 24$ bits. So, in total, 32 bits for both words. This is an example of how space can be improved.

Simulate a longer word size One of the limitations of the present implementation is that the maximum integer size is 64 bits on a common modern machine. An interesting improvement would be to simulate larger word size w, which would increase k, allowing to store more keys at the node level. This would have to be carefully studied to keep every operation in $O(1)$ time.

Lookup `M` constant The `Util` class includes a function, `M`, which takes b and w as parameters and returns the word $(0^{b-1}1)^{(w/b)}$. It is a widely used function in this repository, and it has been implemented with a loop. With limited word size, this could easily be implemented as a lookup function, improving the running time up to $O(1)$.

Profiling This technique allows us to see which sections of the code the CPU uses most of the time, indicating possible code bottlenecks. By using it, the implementation can be fine-tuned up to the point where it can be competitive.

Bibliography

[Anderson, 2005] Anderson, S. (2005). Bit twiddling hacks. `https://graphics.stanford.edu/~seander/bithacks.html`. Accessed: 2020-08-17.

[Beame and Fich, 1999] Beame, P. and Fich, F. E. (1999). Optimal bounds for the predecessor problem. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 295–304.

[Bille, 2020] Bille, P. (2020). 02282 algorithms for massive data sets; lecture 2, spring 2020. `http://courses.compute.dtu.dk/02282/2020/predecessor/predecessor1x1.pdf`.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[Demaine et al., 2012a] Demaine, E., Christiano, P., Fingeret, S., and Rao, S. (2012a). 6.851: Advanced data structures; lecture 11—march 22nd, 2012.

[Demaine et al., 2012b] Demaine, E., Huynh, K., Klerman, S., and Shyu, E. (2012b). 6.851: Advanced data structures; lecture 12—april 3rd, 2012.

[Fredman and Saks, 1989] Fredman, M. and Saks, M. (1989). The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354.

[Fredman and Willard, 1993] Fredman, M. L. and Willard, D. E. (1993). Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436.

[Hentosh, 2016] Hentosh, J. (2016). Patriciaset.java. `https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/PatriciaSET.java.html`. Accessed: 2020-08-23.

[Nelson and Liu, 2014] Nelson, J. and Liu, D. (2014). Cs 224: Advanced algorithms; lecture 2—september 4th, 2014.

[Nelson and Schorow, 2013] Nelson, J. and Schorow, M. (2013). Cs 224: Advanced algorithms; lecture 1—september 2nd, 2013.

[Pătraşcu and Thorup, 2014] Pătraşcu, M. and Thorup, M. (2014). Dynamic integer sets with optimal rank, select, and predecessor search. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 166–175. IEEE.

[Sedgewick, 2002] Sedgewick, R. (2002). *Algorithms in Java, Parts 1-4*. Addison-Wesley Professional.

[Sedgewick and Wayne, 2011] Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-wesley professional.

[Shirazi, 2003] Shirazi, J. (2003). *Java performance tuning*. O'Reilly.

A Appendix

This appendix features some additional data structures that have been either entirely or partially implemented in the scope of the dynamic predecessor problem. They have been included together in the repository of the project, and the goal was to use them to benchmark them with the Dynamic Fusion Tree once this was fully implemented.

A.1 `BitsKey`

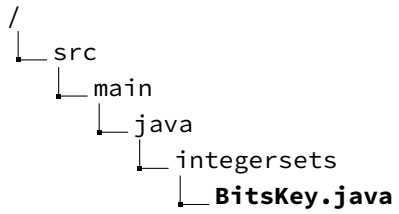

Figure A.1: Location of the `BitsKey.java` file in the folder structure

The `BitsKey` implementation consists of a simple helper class that stores a `long` value and offers some useful functions such as:

- `int bit(final int d)` returns the value of the d^{th} digit of the key stored in the instance.
- `int compareTo(final BitsKey that)` returns an integer that represents the how the value of the present instance of `BitsKey` compares with the value of `that BitsKey` instance. If it returns -1, then `that` is larger than itself; zero is they have the same value and 1 if itself is larger than `that`.
- `boolean equals(final BitsKey that)` returns a `boolean` value that is `true` if and only if the values of the two instances are the same; and `false` otherwise.

This implementation can be found in the file highlighted in figure A.1, and it is used as a helper class in the following implementations.

A.2 Additional Dynamic Predecessor Problem Data Structures

A.2.1 Binary Search Trie

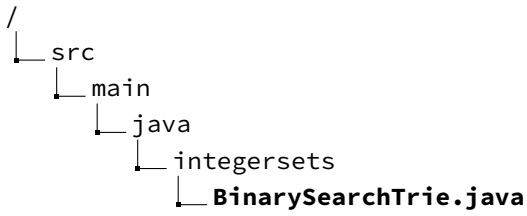

Figure A.2: Location of the `BinarySearchTrie.java` file in the folder structure

The `BinarySearchTrie.java` file, located in the path shown in figure A.2, features a fully functioning implementation of a Binary Search Trie as presented by Sedgewick in [Sedgewick, 2002]. This implementation implements the project's `RankSelectPredecessorUpdate` interface, which means that it has been expanded to answer dynamic predecessor problem queries. It has also been put through correctness tests as described in Section 4.2, passing all instances.

A.2.2 Patricia trie

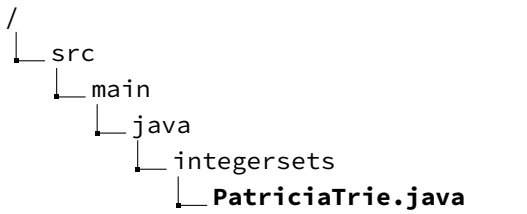

Figure A.3: Location of the `PatriciaTrie.java` file in the folder structure

The `PatriciaTrie.java` file, located in the path shown in figure A.3, features an implementation of a Patricia Trie as presented by Sedgewick in [Sedgewick, 2002], but it is missing the `rank` and `select` methods. It includes an overridden `member` method. In total, it supports insertion, deletion, and membership queries. For this reason, it does not pass all test instances from Section 4.2, but with the missing methods implemented, it is a valid data structure to be benchmarked with the other data structures mentioned in this paper.

A.2.3 Non-recursive Patricia Trie

Figure A.4: Location of the `NonRecursivePatriciaTrie.java` file in the folder structure

Recursion can be slow [Shirazi, 2003]. For this reason, we adapted a non-recursive implementation of a Patricia Trie from [Hentosh, 2016] to store instances of our custom class `BitsKey` at their nodes instead of `Strings`. This implementation is featured in the `NonRecursivePatriciaTrie.java` file, located in the path shown in figure A.4, and it lacks the `rank` and `select` methods in order to fully solve the dynamic predecessor problem. Like the recursive Patricia Trie implementation mentioned in Section A.2.2, it supports insertion, deletion, and membership queries, but the missing methods make some of the test instances from Section 4.2 fail. Implementing the missing methods makes this data structure another interesting instance to be benchmarked with the other data structures mentioned in this paper.