UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Jennifer Veismann

The Spine: An efficient two-level dynamic B-Tree with fast selection

Bachelor's Thesis (9 ECTS)

Supervisor: Bruno Rucy Carneiro Alves de Lima

Tartu 2024

The Spine: An efficient two-level dynamic B-Tree with fast

Abstract:

This thesis compares the efficiency of the Rust programming language standard library B-Tree implementation against a simplified alternative, the “Spine”. As the work is highly context-dependent, the thesis also provides an overview of the Rust programming language. As the “Spine” is a two-level B-Tree indexed by a Fenwick tree, an overview of the basic operations of the Fenwick tree is also provided.

The comparison of the different operations of the two data structures reveals that the "Spine" is significantly faster in the select operation than the regular B-Tree, especially for larger data sizes. In the case of other operations, depending on the dataset size, there was not as big a difference as with the select operation.

Keywords:

B-Trees, B-Tree insert, B-Tree delete, B-Tree search, Fenwick Tree, The Spine: B-Trees indexed by a Fenwick Tree, Rust programming language, select operation.

CERCS: P160, Statistics, operation research, programming, actuarial mathematics

Spine: tõhus kahetasandiline dünaamiline B-puu kiire valikuga

Lühikokkuvõte:

See lõputöö võrdleb Rust programmeerimiskeele standardteegis B-Puu rakendamise efektiivsust lihtsustatud alternatiivi, “Spine”-ga. Kuna lõputöö on väga kontekstipõhine siis antakse ülevaate uuest programmeerimiskeelest Rust. Kuna “Spine” on idekseeritud B-Puu Fenwick-Puu abil, siis antakse ülevaade ka selle põhilistest operatsioonidest, mida “Spine” kasutab.

Kahe andmestruktuuri erinevate operatsioonide võrdlusest tuleb välja, et “Spine” on valimise operatsioonis oluliselt kiirem, kui seda on B-Puu eriti suuremate andmestruktuuridema. Teiste operatsioonide puhul olenes andmestiku suurusest, kuid niivõrd suurt erinevust kui otsingu operatsiooniga ei tekkinud.

Võtmesõnad:

B-Puu, B-Puu lisamis operatsioon, B-Puu kustutamis operatsioon, B-Puu otsimise operatsioon, Fenwick-Puu, Spine: Fenwick-Puu abil indekseeritud B-Puu, Rust programmeerimiskeel.

CERCS: P160 Statistika, operatsioonanalüüs, programmeerimine, finants- ja kindlustusmatemaatika

Table of Contents

1 Introduction

Data structures play a significant role in the efficiency of algorithms, especially for database management and information retrieval systems. Among these structures, the B-Tree stands out thanks to its wide application in systems that require efficient read-and-write operations. As data sizes grow, there is always a need for faster and better structures to handle those datasets more efficiently.

This thesis introduces "The Spine," a two-level dynamic B-Tree indexed by a Fenwick Tree. The motivation for introducing this new structure is that it serves as a simple order-statistic B-tree with potentially better cache locality. This structure is particularly beneficial when working with dynamic data sets where time-sensitive insertion, deletion, and search operations are needed [1].

The purpose of The Spine is to leverage the efficient operations of the Fenwick Tree to maintain an order-statistic index efficiently.

The thesis aims to provide an overview and evaluation of "The Spine," an alternative to the Rust standard library B-Tree. Rust's ownership model and memory safety features are crucial in building reliable data structures that manually manage memory [3]. This approach ensures that Rust's performance characteristics and ability to prevent common programming errors like data races and buffer overflows benefit "The Spine" [3]. The design of "The Spine," with less indirection than the standard library B-Tree, aligns well with Rust.

This thesis will explore "The Spine", detailing its design and the motivation behind its simple architecture. It will test the performance of this new data structure against traditional B-Trees through a series of benchmarks that measure performance time for select, insert and delete operations across different dataset sizes.

2 Related Works

The thesis has similarities with prior works that focus on developing robust, efficient data structures that improve data indexing, select, search, and retrieval operations.

Fenwick Tree

Both this thesis and the previous work by Cushman [5] explore the efficient use of Fenwick Trees in combination with other data structures to improve data access performance. Both works share the same goal of using Fenwick Trees for their logarithmic update and query times, helping to optimise select operations.

Efficient data structures

Efficient data structures are crucial for optimising selection, ranking, and data manipulation tasks across different computing environments. Multiple works, including this thesis, focus on developing indexing techniques to address these challenges. Graefe's work [6] highlights modern techniques to optimise B-Trees, especially in database systems. The Spine's design is similar to Graefe's approach to modular, space-efficient indexing systems. While Lokeshwar's work [7] analyses different structures to balance search speed and memory efficiency, this thesis integrates Fenwick Trees with B-Trees to create "The Spine", which optimises rank and selection performance.

Panchal & Kumar's work [8] and this thesis focus on optimising structures for efficient operations. Similarly, the work on high-dimensional data structures proposes an optimised R-train+ structure to enhance spatial and high-dimensional indexing. The limitations of traditional indexing structures, like B-Trees and R-Trees, which often need more depth and complexity when managing high-dimensional data, are addressed in both works.

Both works address data structures' performance in search operations. While Lokeshwar's work analyses different structures to balance search speed and memory efficiency, this thesis integrates Fenwick Trees with B-Trees to create "The Spine," which optimises rank and selection performance.

3 Rust basics

Rust is a statically typed, compiled programming language renowned for its focus on safety, concurrency, and performance. Developed with an advanced type system, Rust ensures memory safety through its unique borrow checker, which prevents data races and other common vulnerabilities found in systems programming [9].

This chapter explores the basics of Rust programming language. It is essential to understand Rust basics as they are used throughout the thesis. The variables, basic types, functions, ownership, and how control flows are examined. It's important to note that this chapter is based on the teachings from the Rust Programming book [10].

3.1 Basic Types

In this subsection, Rust's common programming concepts will be discussed. In Figure 1. the variables are mostly immutable. Once one variable has a name, the value is unchangeable. The compiler ensures it will remain the same once the value is declared. Even though variables are immutable by default, they can be mutable by adding ‘mut’ in front of the variable name.

Constants are always immutable. Unlike with variables, users cannot use “mut” with constants. Constants can be declared in any scope, which makes coding easier in cases where some values are needed in many parts of the code. Peculiar to constants is that they can be set only to a constant expression, not a result of a value. In Rust, the naming convention for constants is to use all uppercase with underscores between words.

All the variable types must be specified. Rust has scalar types: integers, floating-points, numbers, Booleans, and characters. The variable signing goes slightly differently than in some other languages; for example, an integer is a number without a fractional component. Each integer value can be signed or unsigned and has a specific size. Every variable that has been signed can store numbers from -(2n - 1) to 2n - 1 – 1 inclusive, where the letter n is the number of bits that the variable uses.

```
let x = 5; // value has been declared, so it is now unchangeable
let mut y = 6;
y = 7; // value can be changed because the "mut" keyword
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3; // making a constant
```

Figure 1. Variable naming example [11].

3.2 Functions

The keyword “fn” must be followed by a function name and a set of parentheses to declare a new function. This language uses a snake case for function variable names. To call out a function, Rust does not care where it is called, only that it is defined somewhere in the scope where the caller sees it. The language is expression-based and does not have the same distinctions as other languages. The function can be used as a statement or as an expression. The difference is that statements do not return a value, unlike expressions. Expressions comprise most of the code and can be part of statements. The most common expressions include a function and a new scope block between curly brackets. In Rust, there is no naming return value. The final expression in the block is the returned value. Most of the functions return the last expression implicitly.

```
fn main() {
    println!("Hello, world!");
    another_function();
}
```

Figure 2. Function example [12].

If statements must be bool, unlike in some languages, Rust will not try to convert types to Boolean, which are not Boolean. In both the “if” and “else” blocks, the return value must be the same type, or there will be an error because variables must have a single type. There are three kinds of loops: loop, while, and for. The keyword loop tells us to run the code forever or until it explicitly demands to stop. There is a way to specify a loop with a label optionally, and when the loop uses break or continue with the label, the keywords only apply to this labelled loop.

3.3 Ownership

Ownership allows Rust to guarantee memory safety without relying on a garbage collector[1], making it crucial to comprehend how ownership functions. Whether a value is on the heap or the stack is essential, which makes Rust different from many other programming languages. Both stack and heap are parts of memory that are in the code, but they are structured differently. Stack always has a fixed size, unlike the Heap. A heap needs a certain amount of

[1] https://www.techtarget.com/searchstorage/definition/garbage-collection

space in memory, so the memory allocator can find a spot in a heap with enough space and mark it as being in use.

Pushing to the heap is slower than pushing to the stack because the stack does not need to search for a place to store data. Accessing data in a heap is slower than with the stack.

What is important to remember about ownership is that each value has an owner. There is only one owner at a time, and the value is dropped when the owner goes out of scope. The memory returning is different from that of other languages. Many programming languages have a garbage collector that keeps and cleans up memory if it is unused, but in Rust, the user must identify when to free memory. If the user frees memory twice, there will be a bug. There is a way to drop the value and free the memory by itself, and it happens when the variable that owns it goes out of scope. In some cases, when the value is copied (works with strings), the first value will be deleted.

There is a way to refer to some value without taking ownership of it. The user must use "&" before calling it out. The syntax "&" lets the user create new references without owning them, which means the value is not dropped when the reference stops being used. It is important to note that while referring to some values, they are still immutable unless the keyword "mut" is used. A data race is akin to a race condition which occurs when multiple pointers access the same data simultaneously, and at least one of these pointers is involved in writing the data.

```
let s1 = String::from("hello");
let len = calculate_length(&s1);// syntax &s1 reference to s1 without
owning it
```

Figure 3. Ownership example [13].

Rust's uniqueness is in its robust memory safety features, and one peculiar aspect is the concept of string slices. Unlike other languages, this one provides a powerful mechanism for referencing parts of a string without owning the entire string. This is achieved through string slices, denoted by the type "&str". An example of the usefulness of string slices is when the user needs to extract the first word from a string. By using a string slice ("&str") instead of just an index, Rust ensures that the reference is still valid and connected to the original string data. This method prevents using outdated indexes, which can lead to subtle bugs.

String slices are not limited to strings but extend to other collections like arrays. The syntax for array slices, such as "&[i32]", mirrors string slices, providing a consistent and intuitive

way to reference portions of various data structures. This uniformity in handling slices contributes to the capability to enforce memory safety at compile time, reducing the likelihood of runtime errors

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
assert_eq!(slice, &[2, 3]);
```

Figure 4. Slices example [14].

3.4 Structs

Rust has merged some principles from object-oriented programming, such as how classes are defined in other object-oriented languages. It uses the 'struct' keyword to outline its entire structure. Inside a “struct” are variables known as fields, each given a specific name. This detailed naming enhances the code's readability by clarifying what each field is for, so there is no need to rely on the order of the fields. This careful naming of fields in structures helps to explain the purpose of each piece of data, leading to code that speaks for itself.

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Figure 5. A user struct definition [15].

Thanks to dot notation, accessing and modifying data is straightforward when working with structs. Users can pinpoint and alter specific pieces of data within a structure without hassle.

```
fn main() {
    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

Figure 6. Using struct update syntax to set a new email value for a User instance but to use the rest of the values from user1 [15].

Methods use the keyword “fn”, name, parameters, and return a value like functions. Still, methods are associated with a struct, Enum, or trait object, and their first parameter is always self, representing the instance of the type on which the method is called. The implementation of methods using struct is done using the “impl” block, and with that, this block is associated with all contained functions and methods of the type specified after the simple keyword (Figure 7).

```
struct Rectangle {
    width: u32,
    height: u32,
}
impl Rectangle {
    fn area(&self) -> u32 {
        Self.width * self.height
    }
}
```

Figure 7. Defining an area method on the Rectangle struct [16].

3.5 Enum

Enumerations represent sets of types. All possible type variants are enumerated in Enum, and an Enum value can only be one of its variants. To use an Enum, the keyword “enum” must be followed by the name of the Enum and the set of variants in curly braces like in Figure 8. The data is directly attached to each Enum variant, so there is no need for extra structure. Enum’s advantage is that each variant can have different types and amounts of data.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
```

```
}
```

Figure 8. An enum where each variant stores different amounts and types of values [17].

Rust's handling of null values differs from many other programming languages, as it lacks the traditional null feature, often leading to errors, vulnerabilities, and system crashes. Instead, it applies the “Option<T>” Enum to show the common scenario where a value could be present or absent. Unlike null, “Option<T>” forces developers to explicitly handle the case when a value might be absent, reducing the risk of assuming a not-null value incorrectly. The compiler ensures that “Option<T>” and the underlying type “T” are distinct, preventing operations that mix the two types and promoting code safety.

The “Option<T>” enum is beneficial and included in the prelude, so there is no need to import it into the project. Its variants can be used, “Some” and “None”, directly without prefixing them with “Option::” (Figure 9). Despite being readily accessible, “Option<T>” is still a regular enum, and “Some(T)” and “None” are just its different forms.

```
enum Option<T> {
    None,
    Some(T),
}
```

Figure 9. Example of “Option<T>” with variants [17].

Rust features a control flow mechanism known as “match”, which lets the developers compare a value against a series of patterns and execute corresponding code based on the first matching pattern. Unlike traditional conditional expressions, “match” is not restricted to Boolean conditions. One notable application of match is with Enums. Enum variants serve as patterns, and the compiler ensures that all possible cases are handled.

The importance of “match” is a crucial safety feature, and the compiler enforces that all potential cases are covered, reducing the likelihood of bugs (Figure 10). This is seen in scenarios like handling “Option<T>”, where forgetting to account for the none case triggers a compile-time error.

```
fn plus_one(x: Option<i32>) -> Option<i32> {
```

```
match x {
            None => None,//Checks if x matches None
            Some(i) => Some(i + 1),//checks if Some(x) matches Some(i)
        }
    }
let five = Some(5);//takes value 5
let six = plus_one(five);takes value 6
let none = plus_one(None);//takes value none
```

Figure 10. A function that uses a match expression on an Option<i32> [18].

4 B-Trees: The standard

One of the more widely used data structures is the B-tree. Database engines commonly use B-trees, including prominent systems such as MySQL[2] and SQLite[3]. B-trees were mentioned for the first time in July 1970, when the research work Organization and Maintenance of Large Ordered Indices was published on B-trees; the authors of this work were Rudolf Bayer and Edward M. McCreight [19]. Although log-structured merge trees[4] have developed significantly and are widely used, B-trees remain the dominant indexing structure, serving as a standard implementation in almost all relational databases and finding applications in various non-relational databases [20].

This chapter discusses using Rust code to examine the B-tree search, insert, and remove operations. All the code used in this chapter is taken from the Rust documentation [21].

4.1 Core Operations

Rust's standard library B-Tree implementation will be introduced for an equal ground of comparison. Looking into Rust's standard library B-Tree implementation.

```
1    pub fn search_node<Q: ?Sized>(self, key: &Q) -> SearchResult<BorrowType, K, V, Type,
2  Type>
3    where
4        Q: Ord,
5        K: Borrow<Q>,
6    {
7        match unsafe { self.find_key_index(key, 0) } {
8            IndexResult::KV(idx) => Found(unsafe { Handle::new_kv(self, idx) }),
9            IndexResult::Edge(idx) => GoDown(unsafe { Handle::new_edge(self, idx) }),
10        }
11    }
```

Figure 11. Search_node function example [22].

---

2 https://www.oracle.com/in/mysql/what-is-mysql/#:~:text=MySQL%20is%20a%20relational%20database%20management%20system,-Databases%20are%20the&text=A%20relational%20database%20stores%20data,physical%20files%20optimized%20for%20speed.

3 https://www.simplilearn.com/tutorials/sql-tutorial/what-is-sqlite#:~:text=SQLite%20is%20used%20to%20develop,some%20data%20within%20an%20application.

4 https://www.scylladb.com/glossary/log-structured-merge-tree/#:~:text=A%20log%2Dstructured%20merge%2Dtree,memory%20and%20disk%2Dbased%20structures.

The search logic for B-Trees in Rust is happening inside two functions. The function "search_node" (Figure 11) is designed to look for a given key in a B-Tree node and return the search result. The returned result can be "Found" if the key is found (line 8), or it can return "GoDown" (line 9), which provides direction on where to proceed because the key was not found.

```
1   unsafe fn find_key_index<Q: ?Sized>(&self, key: &Q, start_index: usize) ->
2  IndexResult
3    where
4        Q: Ord,
5        K: Borrow<Q>,
6    {
7        let node = self.reborrow();
8        let keys = node.keys();
9        debug_assert!(start_index <= keys.len());
10        for (offset, k) in unsafe { keys.get_unchecked(start_index..)
11  }.iter().enumerate() {
12            match key.cmp(k.borrow()) {
13                Ordering::Greater => {}
14                Ordering::Equal => return IndexResult::KV(start_index + offset),
15                Ordering::Less => return IndexResult::Edge(start_index + offset),
16            }
17        }
18        IndexResult::Edge(keys.len())
19    }
```

Figure 12. Find_key index function [22].

The main search with the node is happening inside the "find_key_index" function (Figure 12). This function goes over the keys of the node, starting from a given index (start_index). It locates the position of the key or determines where the key should belong if it's not present. Keys in the node are compared with the searched key. The mentioned method does the following three checks: if a key is greater than the searched key, the search continues to the following key (line 13), if a key is equal to the searched key, then the key is found, and the function returns the key index with its value (line 14), if the current key is greater then the searched key it indicates that searched key does not exist in the node and should be in a previous position, "IndexResult::Edge" is returned with the index where the search key belongs(line 15). If the search key exceeds all keys in the node, the function "find_key_index" returns an index pointing beyond the last key, indicating the search should

continue in the rightmost child node. The search mechanism is important in B-trees; it is the basis for insertion and deletion operations [22].

```
fn insert<A: Allocator + Clone>(
        mut self,
        key: K,
        val: V,
        edge: Root<K, V>,
        alloc: A,
    ) -> Option<SplitResult<'a, K, V, marker::Internal>> {
        assert!(edge.height == self.node.height - 1);

         if self.node.len() < CAPACITY {
            self.insert_fit(key, val, edge);
            None
         } else {
            let (middle_kv_idx, insertion) = splitpoint(self.idx);
            let middle = unsafe { Handle::new_kv(self.node, middle_kv_idx) };
            let mut result = middle.split(alloc);
            let mut insertion_edge = match insertion {
                LeftOrRight::Left(insert_idx) => unsafe {
                    Handle::new_edge(result.left.reborrow_mut(), insert_idx)
                },
                LeftOrRight::Right(insert_idx) => unsafe {
                    Handle::new_edge(result.right.borrow_mut(), insert_idx)
                },
            };
            insertion_edge.insert_fit(key, val, edge);
            Some(result)
        }
    }
}
```

Figure 13. Insert function [23].

The function “insert” (Figure 13) is used to insert key-value pairs into nodes of a B-tree structure. The following operations are fundamental to how B-trees manages data efficiently, balancing between multiple levels of nodes to maintain sorted order and quick access properties.

```
pub unsafe fn push_with_handle<'b>(
        &mut self,
        key: K,
```

```
        val: V,
    ) -> Handle<NodeRef<marker::Mut<'b>, K, V, marker::Leaf>, marker::KV> {
        let len = self.len_mut();
        let idx = usize::from(*len);
        assert!(idx < CAPACITY);
        *len += 1;
         unsafe {
             self.key_area_mut(idx).write(key);
             self.val_area_mut(idx).write(val);
             Handle::new_kv(
                 NodeRef { height: self.height, node: self.node, _marker: PhantomData },
                 idx,
             )
         }
```

Figure 14. Push_with_handle function [23].

The method “push_with_handle” (Figure 14) directly adds a new key-value pair to a leaf node. First, it checks the number of keys and ensures the capacity to add one more, and then the new key value is added to the node’s memory. Since this insertion is directly at the leaf level without traversing the tree, it has an O(1) time complexity, assuming space is available in the node.

```
fn insert_fit(&mut self, key: K, val: V, edge: Root<K, V>) {
        debug_assert!(self.node.len() < CAPACITY);
       debug_assert!(edge.height == self.node.height - 1);
       let new_len = self.node.len() + 1;

        unsafe {
            slice_insert(self.node.key_area_mut(..new_len), self.idx, key);
            slice_insert(self.node.val_area_mut(..new_len), self.idx, val);
            slice_insert(self.node.edge_area_mut(..new_len + 1), self.idx + 1, edge.node);
             *self.node.len_mut() = new_len as u16;
             self.node.correct_childrens_parent_links(self.idx + 1..new_len + 1);
         }
```

Figure 15. Insert_fit function [23].

The method “insert_fit” (Figure 15) handles insertion when there is not enough space in the node. It inserts a new key-value pair and adjusts the pointers to child nodes (edges). It involves shifting existing keys, values, and edges to make space for the new elements. The method performs multiple element shifts and has O(n) time complexity, where n is the number of elements in the node. In the method “insert” in Figure 13, if the node has reached capacity, and adding another key-value pair requires splitting the node. The node is split into

two, and the new key-value pair is added to the appropriate new node based on the split. Split logic determines the split point and manages the redistribution of keys and values, which is crucial. The split affects the current node and potentially requires adjustments (like adding new keys or splitting further) up the tree. The node split time complexity is O(n) due to the need to redistribute keys and values between the two new nodes. However, if the splitting cascades up the tree (a split causing a parent node to also split), the complexity can become O(log m), where m is the total number of keys in the tree due to the tree's height affecting the operations.

```
pub fn remove_kv_tracking<F: FnOnce(), A: Allocator + Clone>(
        self,
       handle_emptied_internal_root: F,
       alloc: A,
   ) -> ((K, V), Handle<NodeRef<marker::Mut<'a>, K, V, marker::Leaf>, marker::Edge>) {
       match self.force() {
           Leaf(node) => node.remove_leaf_kv(handle_emptied_internal_root, alloc),
           Internal(node) => node.remove_internal_kv(handle_emptied_internal_root,
 alloc),
       }
   }
```

Figure 16. Remove_kv_tracking function [24].

The main logic of B-Tree key-value pair removal is in the “remove_kv_tracking” function (Figure 16). This function removes a key-value pair from the tree. Depending on whether the current node is a leaf or an internal node, it directs the removal to the following function, “remove_leaf_kv” (Figure 17) or “remove_internal_kv” (Figure 18). Function “remove_kv_tracking” returns the removed key-value pair.

```
fn remove_leaf_kv<F: FnOnce(), A: Allocator + Clone>(
       self,
       handle_emptied_internal_root: F,
       alloc: A,
   ) -> ((K, V), Handle<NodeRef<marker::Mut<'a>, K, V, marker::Leaf>, marker::Edge>) {
       let (old_kv, mut pos) = self.remove();
       let len = pos.reborrow().into_node().len();
       if len < MIN_LEN {
           let idx = pos.idx();
            let new_pos = match pos.into_node().forget_type().choose_parent_kv() {
               Ok(Left(left_parent_kv)) => {
                   debug_assert!(left_parent_kv.right_child_len() == MIN_LEN - 1);
```

```
13                  if left_parent_kv.can_merge() {
14                      left_parent_kv.merge_tracking_child_edge(Right(idx),
15  alloc.clone())
16                  } else {
17                      debug_assert!(left_parent_kv.left_child_len() > MIN_LEN);
18                      left_parent_kv.steal_left(idx)
19                  }
20              }
21              Ok(Right(right_parent_kv)) => {
22                  debug_assert!(right_parent_kv.left_child_len() == MIN_LEN - 1);
23                  if right_parent_kv.can_merge() {
24                      right_parent_kv.merge_tracking_child_edge(Left(idx),
25  alloc.clone())
26                  } else {
27                      debug_assert!(right_parent_kv.right_child_len() > MIN_LEN);
28                      right_parent_kv.steal_right(idx)
29                  }
30              }
31              Err(pos) => unsafe { Handle::new_edge(pos, idx) },
32          };
33
34          pos = unsafe { new_pos.cast_to_leaf_unchecked() };
35
36          if let Ok(parent) = unsafe { pos.reborrow_mut() }.into_node().ascend() {
37              if
38  !parent.into_node().forget_type().fix_node_and_affected_ancestors(alloc) {
39                  handle_emptied_internal_root();
40              }
41          }
42      }
43      (old_kv, pos)
44  }
```

Figure 17. Remove_leaf_kv function [24].

The function "remove_leaf_kv" (Figure 17) removes a key-value pair from a leaf node. At first, the key value is removed, and then the node's length is checked to see if it is below the minimum allowed length (line 8). Because the B-Tree has to be balanced, and if node length is below minimum, it tries to rebalance the tree. Two ways to rebalance the tree are merging the node with a sibling node or stealing key-value pairs from a sibling node. After either the steal or merge operation, the parent node might need adjusting, and in that case, the higher level of three has the same operations (stealing, merging) recursively.

```
fn remove_internal_kv<F: FnOnce(), A: Allocator + Clone>(
        self,
        handle_emptied_internal_root: F,
        alloc: A,
    ) -> ((K, V), Handle<NodeRef<marker::Mut<'a>, K, V, marker::Leaf>, marker::Edge>) {
        let left_leaf_kv = self.left_edge().descend().last_leaf_edge().left_kv();
        let left_leaf_kv = unsafe { left_leaf_kv.ok().unwrap_unchecked() };
        let (left_kv, left_hole) =
left_leaf_kv.remove_leaf_kv(handle_emptied_internal_root, alloc);

         let mut internal = unsafe { left_hole.next_kv().ok().unwrap_unchecked() };
         let old_kv = internal.replace_kv(left_kv.0, left_kv.1);
         let pos = internal.next_leaf_edge();
         (old_kv, pos)
    }
```

Figure 18. Remove_internal_kv function example [24].

In the function “remove_internal_kv” (Figure 18), the code finds the closest leaf node to the key value and uses it to replace the key-value pair in the internal node. This maintains the sorted order of the tree. This operation may cause the need for the tree to rebalance if the leaf from which a key value was taken falls below the minimum allowed length.

The complexity of B-tree operations is generally O(log n), where n is the number of elements in the tree. This is because the tree's height grows logarithmically with the number of elements. The removal operation, in particular, involves searching for the key-value pair to remove (O(log n)), followed by possible rebalancing steps. Each rebalancing step is a constant-time operation (merge, steal, or adjust pointers). Still, in the worst case, removal might require rebalancing up to the root, maintaining the O(log n) complexity.

4.1.1 Cache-friendliness in B-Tree

A cache-friendly data structure optimises data access times by using the CPU cache effectively. This cache is a small, high-speed memory unit temporarily holding frequently accessed data for quick access [26]. B-tree is cache-friendly since most are just arrays, contiguous regions of memory. The structure and access patterns are designed to minimise cache misses, enhancing performance [27].

B-Trees are cache-friendly through their structural organisation. Unlike binary trees, which include, among other things, nodes linking with each other through pointers and often happen

in a disjointed manner, B-Tree elements group themselves into nodes that are essentially pointers to arrays [26]. These all lead to a significant improvement in the cache performance due to structural alignment [28]. The B-Tree node is array-like, storing elements contiguously in memory, optimising cache line utilisation and ensuring faster access speeds than other storage structures [27]. This guarantees access speeds faster than any other forms for the sake of this storage structure [28]. The stored value is also contiguous, so the waste of the cache line is reduced, speeding up the cache access [26]. As such, B-trees provide an efficient solution to database indexing and other applications requiring high-speed data retrieval, which is essential since they exploit modern processor cache architectures for overall system performance increment [25].

4.2 What is necessary to implement rank and select (order-statistic B-Tree)

In an array containing duplicates, the rank of an element represents the average position of all its occurrences in a sorted order. For example, given the array [10,12,15,12,10,25,12], the element 10 appears twice. If we sort this array ([10,10,12,12,12,15,25]), element 10 would be in the first and second position. Therefore, its rank is the mean of these two positions, calculated as $\frac{1+2}{2} = 1.5$. In the same way, the rank of the number 12, which appears three times, would be $\frac{3+4+5}{3} = 4.0$ [29]. The select operation retrieves an element based on this ranking system. Rank 1.5 returns 10 in the same sorted array used in the rank example, as this element corresponds to that average rank.

An order-statistic B-tree is an extended traditional B-tree where the positions of some element can be retrieved as if it were ordered. Order-statistic B-tree is used for efficient searching, insertion, and deletion operations and rank and select operations [30]. Initially, the B-tree nodes must be modified to include relevant information to get an ordered statistic B-tree from a traditional B-tree. This involves augmenting each node with necessary data, which keeps track of the size of the subtree rooted at the node. After that, each node in the tree stores keys and additional information to facilitate ranking and selecting operations. The tree must maintain size information (number of elements) in each node to implement rank and select operations in an order-statistic B-tree [31].

Each node in the tree stores keys and additional information that facilitates ranking and selecting operations. Maintaining each node's size information (number of elements) is

crucial to efficiently implementing rank and select operations in an Order-statistic B-tree [32]. A B-tree's standard insertion and deletion operations are adapted to ensure that subtree sizes are updated whenever nodes are split or merged, which is critical for maintaining accurate rank information [30].

To determine the rank of a given element “x”, the algorithm starts at the root and recursively moves towards the leaf nodes, summing up the counts of elements less than x based on the traversed path. If x matches the key at a node, the rank is determined by adding the count of the left subtree to the rank count collected from the lower leaf nodes [31]. This is how it operates in a regular B-tree without additional data.

The procedure starts at the root to find the element with a given rank k (select operation). It compares k with the counts of the left subtree to decide whether to go left, stay at the current node, or move right. This decision process involves adjusting k to account for the skipped nodes, ensuring that the selection proceeds efficiently to find the k element [32].

5 Fenwick Trees: An unlikely ally.

Fenwick Trees, or binary indexed trees (BIT), are the key to developing the efficient order-statistic B-Tree referred to here as "The Spine" [1].

5.1 What is it used for?

Fenwick Tree is an efficient dynamic data structure. As a flat array of values, this tree can efficiently update, compute and manipulate prefix sums of an array of numbers. The Fenwick Tree accomplishes updates and queries in logarithmic time [33]. This efficiency is achieved through a clever structure that maintains partial sums across an array, allowing updates and prefix sum calculations to be carried out in O(log n) time [2]. It is essential because Fenwick Trees indexing allows for efficient order-statistic B-Trees (The Spine).

5.2 Core Operations

This subsection describes all core operations from the Fenwick Tree implemented in Rust.

The following method shows how to set up a Fenwick Tree using a list of numbers. When given a list, it rearranges the numbers into a structure that makes calculating cumulative sums faster. This setup is done once; after that, the Fenwick Tree is ready for the following operations.

```rust
impl<T> FromIterator<T> for FenwickTree<T>
 where
  T: Copy + AddAssign,
{

  fn from_iter<I>(iter: I) -> Self
  where
      I: IntoIterator<Item = T>,
  {
       let mut inner: Vec<T> = iter.into_iter().collect();
       let n = inner.len();

       for i in 0..n {
           let parent = i | (i + 1);
           if parent < n {
               let child = inner[i];
               inner[parent] += child;
           }
       }

       FenwickTree { inner }
   }
}
```

Figure 19. From_iter function[2].

Function “from_iter” (Figure 19) takes iterator “iter” as input and converts it into Fenwick tree. There are two. Primary operations there: collecting elements and initial construction. Collecting elements means that all elements are collected from the iterator into a vector “inner”. The next core operation was initial construction, which means that after the first operation, it takes each element in this vector, calculates the “parent” index in the tree, and adds the current element's value to the value at this parent index.

For example, to create a Fenwick Tree from an array of lengths [1,6,3,9,2]. After the construction, the internal representation of the tree is “vec! [1,7,3,19,2]”. The code starts by initialising “inner” as a direct copy of the input sequence. So initially, the inner will be “inner = [1,6,3,9,2]”, and the length n of the “inner” is 5. The Fenwick Tree construction involves iterating through each inner element and updating future elements based on the current index using bitwise operations. Now, going through the for loop first, when “i = 0”, the calculated parent is (“parent = 0 | (0 + 1) = 1”) 1, and since it is less than “n = 5”, the

“inner” will be updated “inner[1]” and becomes [1,7,3,9,2], it is updated in line “inner[1] = inner[1]+inner[0] = 6+1 = 7”. Next in the for loop, the “i = 1” so the parent will be 3 and the updated inner will be 16 (“inner[3] = inner[3]+inner[1] = 9+7 = 16”), after that the “inner” will be [1,7,3,16,2]. After the “i = 2,” the parent will be three and “inner” updates to [1,7,3,19,2] (“inner[3] = inner[3]+inner[2] = 16+3 = 19”). Next, the “i = 3’ and the calculated parent will be 7, which is greater than “n = 5”, so no further update is made. The same goes with “i = 4” the parent is 5, which is not less than “n = 5” so again no update is made.

The time complexity for creating the Fenwick tree from an iterator of “n” elements is O(n). The function iterates over all input elements (“n”) only once.

```
impl<T> FenwickTree<T> {
  pub fn prefix_sum(&self, index: usize, mut sum: T) -> T
  where
      T: Copy + AddAssign,
  {
      assert!(index < self.inner.len() + 1);

      let mut current_idx = index;

       while current_idx > 0 {
           sum += self.inner[current_idx - 1];
           current_idx &= current_idx - 1
       }

       sum
   }
```

Figure 20. Perfix_sum function [2].

The method “prefix_sum” (Figure 20) calculates the sum of elements from the start of the array up to the previously specified index. Starting from the given index, the method iteratively adds the value at “current_idx – 1” to the sum, it then updates “current_idx” using a bitwise operation “current_idx &= current_idx – 1”, effectively moving to the following relevant index for summing, according to the Fenwick Tree's structure.

The time complexity of the method “prefix_sum” method is “O(log n)”, where n is the length of the array. This is because Fenwick Tree skips over sections of the array that don’t need to be individually added to the sum, jumping back through the array in logarithmically decreasing steps. That is because each index is a sum of elements, and each step in the

operation jumps back through these elements. The number of jumps to return to the array's start is comparable to the number of bits in the index. Since each step potentially halves the distance to the beginning (by skipping ranges already summarised), the number of steps is logarithmic relative to the array's length. The space complexity for this method is “O(1)” since it only uses a certain amount of extra space regardless of the input size.

```
pub fn add_at(&mut self, index: usize, diff: T)
   where
       T: Copy + AddAssign,
   {
       let mut current_idx = index;

       while let Some(value) = self.inner.get_mut(current_idx) {
           *value += diff;
           current_idx |= current_idx + 1;
        }
    }
```

Figure 21. Add_at function [2].

The ”add_at” (Figure 21) method updates the Fenwick tree by adding a given difference (“diff”) to the value at a specified index. This update is propagated through the tree structure to maintain the logic of sums. The method starts by setting a working variable “current_idx” to the specified index. Then, the method enters a loop that continues until “current_idx” is at a valid position within the “inner” vector. After the loop, the value at “current_idx” is increased by “diff”, and all the other relevant positions are updated in the tree structure. The line “current_idx |= current_idx + 1;” is crucial, as it ensures that all necessary parent nodes are updated so that the tree can quickly calculate prefix sums. This bitwise operation is key in navigating through the tree efficiently. Modifying “current_idx” to point to the next significant position facilitates the propagation of changes to the necessary parent nodes within the tree. Consequently, this increases operational efficiency and accurately reflects cumulative updates across the structure. The time complexity of the method “add_at” is “O(log n)”, where “n” represents the number of elements in the array. This is because, in the worst case, updating a value requires propagating the change up through a series of parent nodes, with the depth of the tree updates comparable to the logarithm of the number of elements. The space complexity remains “O(1)”, indicating that the updates require constant additional space within the existing inner vector representing the Fenwick Tree.

```rust
pub fn index_of(&self, mut prefix_sum: T) -> usize
    where
        T: Copy + Ord + SubAssign,
    {
        let mut index = 0;
        let mut probe = most_significant_bit(self.inner.len()) * 2;

        while probe > 0 {
            let lsb = least_significant_bit(probe);
             let half_lsb = lsb / 2;
             let other_half_lsb = lsb - half_ls;

             if let Some(value) = self.inner.get(probe - 1) {
                 if *value < prefix_sum {
                     index = probe;
                     prefix_sum -= *value;

                     probe += half_lsb;

                     if half_lsb > 0 {
                         continue;
                     }
                 }
             }

             if lsb % 2 > 0 {
                 break;
             }

             probe -= other_half_lsb;
         }

         index
     }
 }
```

Figure 22. Index_of function[2].

The following method, “index_of” (Figure 22), aims to find the index in the original array that compares to the given sum in Fenwick. The method starts with an index set to 0, which shows the starting point for the search. The function starts by creating a variable named “probe”, which is used to navigate through the Fenwick Tree to find the index that matches

the given sum. It starts at the point at the highest possible power of two less than the array's length. The ”probe“ is like a pointer moving to different positions in the tree to check values.

The method enters a loop (runs as long as “probe” is bigger than 0) that adjusts the “probe” value to navigate through the tree. It calculates the “lsb” (least significant bit) of “probe” and divides it into two halves (“half_lsb” and “other_half_lsb”). In case the probe point is less than “prefix sum”, it adjusts “index” to the current probe, substracts the value at the probe from “prefix_sum”, and increases the probe to potentially skip ahead more efficiently. In another case, if the value at the probe is less than “prefix_sum” or the “lsb” is odd, the loop reduces the probe by “other_half_lsb” to improve the search towards smaller indices. Next, it checks if “half_lsb” is zero in that case, the loop continues without further adjustments in “probe”, but in case “lsb % 2 > 0”, the loop breaks immediately to ensure the function does not run into an infinite loop.

Time complexity is O(log n), which indicates that the function gets quicker at finding the index as the array size grows larger. It doesn’t need to check every element; it just jumps through them. The data complexity is O(1), and the function doesn’t need extra space that grows with the size of the array. It uses a few variables to track where and what it's looking for, no matter how big the array is.

6 The Spine: B-Trees indexed by a Fenwick Tree

The Rust standard library's BTreeSet[5] and BTreeMap[6] are widely used due to their robustness and reliable performance characteristics [30]. However, these structures have limitations, particularly regarding operations highly dependent on the order of elements, such as order-statistic operations. In fact, there is no support altogether for rank and select operations.

6.1 Motivation

The development of indexset, a two-level dynamic order-statistic b-tree implemented purely in Rust, addresses these shortcomings by performing in specific scenarios crucial for real-time applications. Inspired by the indexmap[7] project and Python's sorted container"[8], indexset aims to maintain the sorted order of elements while providing efficient lookups by value and position.

By leveraging the Fenwick tree and restricting the B-Tree to have only two levels, indexset simplifies its structure while enhancing efficiency. This design retains logarithmic insertions, searches, deletions, and adds constant timerank and selects operations. These features ensure that indexset can handle high-performance requirements with possibly lower complexity than traditional B-Trees [1].

[5] https://doc.rust-lang.org/std/collections/struct.BTreeSet.html
[6] https://doc.rust-lang.org/std/collections/struct.BTreeMap.html
[7] https://crates.io/crates/indexmap
[8] https://github.com/grantjenks/python-sortedcontainers

6.2 Core Operations

```
1  fn search<T: PartialOrd>(haystack: &[T], needle: &T, iterations: usize) -> Result<usize,
2  usize> {
3    let mut left = 0;
4    let mut right = haystack.len();
5    for _ in 0..iterations {
6        if left >= right {
7            break;
8        }
9
10     let mid = left + (right - left) / 2;
11
12       let mid_value = unsafe { haystack.get_unchecked(mid) };
13
14       if mid_value < needle {
15           left = mid + 1;
16       } else if mid_value > needle {
17           right = mid;
18       } else {
19           return Ok(mid);
20       }
21   }
22
23   Err(left)
24 }
```

Figure 23. Search function example[34].

Figure 23 shows how to perform a binary search on a sorted set (“haystack”) to find the index of the searched element (“needle”). The function signatures are: “T: PartialOrd“, which ensures that the elements in the array (“haystack”) are comparable. “Iterations: usize“ limits the number of iterations the loop can perform, providing a way to limit the search effort. This function returns “Result<usize, usize>“, where “Ok(usize)” is the index of the found element, and “Err(usize)” is the index where the element could be inserted. Variable initialisation in this function are “left = 0” (line 3) and “right=haystack.len()” (line 4). They initialise the bounds of the search, starting from the beginning (“left”) to the end (“right”) of the array. Next comes looping through the search process. The loop runs a maximum of “iterations” times, adjusting the search bounds in each iteration based on comparisons: break condition checks “if left >= right” it stops the loop if the search does not find the needle, middle index calculation “let mid = left + (right-left) / 2” calculates the middle index of the current search

bounds, avoiding potential overflow, element retrieval “let mid_value = unsafe { haystack.get_unchecked(mid) };” takes the value at the mid index without bounds checking, assuming the calculation is always within the valid range. In comparison and bound adjustment, if the middle value is less than the needle, the value ”left” is updated to “mid +1”, narrowing the search to the upper half. If the middle value exceeds the needle, the value “right” will be updated to “mid”, narrowing the search to the lower half. The “Ok(mid)” returns if the middle value equals the needle. The “left” value will be where the needle could be inserted to maintain order in the array.

The time complexity for this function is “O(min(log n, iterations))” because the search is limited by the “iterations” parameter. If the “iteration” is less than “log n”, the search might finish early without going through the entire range that would be checked in a standard binary search. The space complexity is “O(1)” because the search algorithm operates in place and uses a constant amount of space regardless of the input size.

This function is a controlled variation of binary search, allowing the user to define how many attempts should be made to find the target, potentially optimising performance for cases where an approximate location is acceptable or limiting computation in resource-constrained environments.

Two example cases illustrate the functionality and limitations a predefined iteration count imposes. The first example demonstrates a successful search within the given limits, where the array [1, 3, 5, 7, 9] is searched for the needle 5 with an iteration limit of 3. The algorithm quickly locates the needle at index two during the first iteration, effectively demonstrating the efficiency of a binary search when the target is within the immediate middle of the array, returning an output of “Ok(2)”.

The second example highlights the scenario where the needle is not within the specified iteration constraints. In this case, the array remains [1, 3, 5, 7, 9], but the target is now 8, with a tighter constraint of only two iterations. During the first iteration, the midpoint value of 5 is assessed and determined to be less than the needle, prompting an update of the search range to the upper half of the array. In the subsequent iteration, the new midpoint value of 9 is evaluated and found to be greater than the needle, narrowing the range to the interval where the needle would theoretically reside if it were present. The iteration limit is then reached without locating the target, showcasing the limitations of the search algorithm when restricted by a fixed number of iterations and resulting in an expected output of “Err(4)”, which

indicates where the needle could be inserted while maintaining the sorted order. These examples demonstrate the operational dynamics and potential constraints of employing iteration-limited binary search within sorted arrays.

```
pub fn insert(&mut self, value: T) -> bool {
    let node_idx = self.locate_node(&value);
    if self.inner[node_idx].len() == DEFAULT_INNER_SIZE {
        let new_node = self.inner[node_idx].halve();
        // Get the minimum
        let new_node_min = new_node.inner[0].clone();
        // Insert the new node
        self.inner.insert(node_idx + 1, new_node);
        let insert_node_idx = if value < new_node_min {
            node_idx
         } else {
           node_idx + 1
          };
          if self.inner[insert_node_idx].insert(value) {
              // Reconstruct the index after the new node insert.
              self.index = FenwickTree::from_iter(self.inner.iter().map(|node|
node.len()));
              self.len += 1;
              true
          } else {
              false
          }
      } else if self.inner[node_idx].insert(value) {
          self.index.add_at(node_idx, 1);
          self.len += 1;
          true
      } else {
          false
      }
  }
```

Figure 24. Insert function[34].

The insert function adds a value to a set, as seen in Figure 24. The method begins by identifying the node index (“node_idx”), where the value should be placed with the “locate_node” function. After the node index place is found, the function checks if the node at “node_idx” has reached capacity (“DEFAULT_INNER_SIZE”). In case the node is full, it is split in two, where “self.inner[node_idx].halve()” splits the node and returns the new node created from the split.

After the node is split, the correct node for inserting a new value is determined based on a comparison with the new node's smallest value (“new_node_min”). Next, the value is inserted into the appropriate node (either a new or original one). After the last step, the Fenwick Tree needs to be updated to reflect the change in node sizes. If the node is not split, the value is inserted directly into the identified node. Fenwick Tree is adjusted at “node_idx” to increment the count, reflecting the addition. The method returns “true” if the insertion was successful (value was not previously in the node) and “false” otherwise (value was already present and thus not added again).

In the inserting method, the first example is a scenario in which an element is inserted into a node without splitting a node. The data structure consists of two nodes: Node 0 contains elements [1, 3, 5], and Node 1 contains elements [7, 9]. The inserting value is 4. The algorithm for the “locate_node” function determines that four fits best in Node 0, positioned between 3 and 5. No splitting is required as Node 0 contains only three elements, which is well below the “DEFAULT_INNER_SIZE” of 5, so no splitting is required. The value four is successfully inserted into Node 0, and the updated node now contains values [1, 3, 4, 5]. The Fenwick Tree is updated to reflect the new total element count in Node 0, and “self.len” is incremented, indicating the successful addition of a new element.

```
1     fn delete<Q>(&mut self, value: &Q) -> (Option<T>, bool)
2    where
3        T: Borrow<Q>,
4        Q: Ord + ?Sized,
5    {
6        let mut removed = false;
7        let mut removal = None;
8        let (node_idx, position_within_node) = self.locate_value(value);
9        if let Some(candidate_node) = self.inner.get(node_idx) {
10           if let Some(candidate_value) = candidate_node.get(position_within_node) {
11               if value == candidate_value.borrow() {
12                   removal = Some(self.delete_at(node_idx, position_within_node));
13                   removed = true;
14               }
15           }
16       }
17
18       (removal, removed)
19   }
```

Figure 25. Delete function[34].

The delete function is designed to remove a specified element from a complex data structure (Figure 25). There is “value:&q” in the function parameters, which refers to the value to be deleted, allowing the function to handle potentially large or complex types efficiently without ownership transfer. It returns “Option<T>“ and “bool”. The first holds the removed element if found and removed, and the second indicates whether it was successful. There is initialisation “removed”, which indicates whether the deletion has occurred. The “removal” initialised to “None”, potentially to hold the removed element if found.

The deletion process begins with the “locate_value” function, which determines the exact location of the element within the data structure, precisely the node's index and the element's position within that node. This is needed to access the element directly without a complete data structure traversal.

Following a successful location, the function nested conditional checks if statements to verify the presence of the node and the specific element within that node. These checks ensure that the function can safely proceed with the deletion without runtime errors due to accessing non-existent elements. If the element at the found position matches the provided value (“value == candidate_value.borrow()”), the “delete_at” function is called to execute the actual removal of the element from the node.

```
1  fn delete_at(&mut self, node_idx: usize, position_within_node: usize) -> T {
2        let removal = self.inner[node_idx].delete(position_within_node);
3        let mut decrease_length = false;
4       // check whether the node has to be deleted
5       if self.inner[node_idx].len() == 0 {
6           // delete it as long as it is not the last remaining node
7           if self.inner.len() > 1 {
8               self.inner.remove(node_idx);
9               self.len -= 1;
10               self.index = FenwickTree::from_iter(self.inner.iter().map(|node|
11  node.len()));
12            } else {
13                decrease_length = true;
14            }
15        } else {
16            decrease_length = true;
17        }
18
19        if decrease_length {
20            self.index.sub_at(node_idx, 1);
21            self.len -= 1;
```

```
22          }
23
24          removal
25      }
```

Figure 26. Delete_at function [34].

The “delete_at” function seen in Figure 26 removes an element from a specific node at a given position. This function is called once the delete function determines where the removed element is. The function “delete_at” first calls the removal operation with “self.inner[node_idx].delete(position_within_node);”. The “self.inner” refers to a collection of nodes, and “delete(position_within_node)” directly removes and returns the element at the specified position within the removed node.

Following the removal, the function checks if the node from where the element was removed is empty (“self.inner[node_idx].len() == 0”). If the node is empty, it checks whether it is the only node remaining in the collection. If there are other nodes (“self.inner.len() > 1”), the empty node is removed from the collection, and the overall length of the data structure (“self.len”) is decremented. The Fenwick Tree is reconstructed to have new lengths for the remaining nodes. This is done using “self.index = Fenwick-Tree::from_iter(self.inner.iter().map(|node| node.len()));”, which repeats over the lengths of all nodes to rebuild the index.

If the emptied node is the last in the collection, or if the node still contains other elements after the deletion, a flag “decrease_length” is set to “true”, prompting a decrement in the global index (“self.index.sub_at(node_idx, 1)”) and the overall length of the structure (“self.len -= 1”).

The time complexity of the “delete” and “delete_at” functions is in the worst case “O(n log n)” because these functions work together to locate, remove, and then adjust the structure of a potentially complex collection. The combined space complexity of these operations remains primarily O(n) due to the reconstruction of the Fenwick Tree.

Example of deleting an existing element from the data structure. The data structure is organised into nodes, each containing multiple elements arranged as follows: Node 0 holds [2, 4, 6], Node 1 holds [8, 10, 12], and Node 2 holds [14, 16, 18]. The task is to delete the value 10. the deletion starts with the “locate_value” function, which will find the value 10 in Node 1 at position 1. Next, the function “delete_at” with parameters “node_idx = 1” and “posi-tion_within_node = 1”. In this function, element ten is removed from its node, and then

the length of Node 1 is checked. Because the node still contains elements [8, 12], no significant restructuring of the data structure is required except for Fenwick Tree.

7 Evaluation

In Chapter 4, the B-Tree was discussed, and in Chapter 6, the Spine. In this chapter, those two will be compared using different data sizes. The analysis will focus on how operations behave under sequential and random access patterns. The tests are in small, medium, and large data sizes to determine when it is most beneficial to use the Spine algorithm.

Apple Macbook Pro with an M3 Pro Chip computer was used to run these benchmarks, and the version of Visual Studio was 1.89.0. The data used to test these operations consists of random sequential integers. The code uses these integers to benchmark operations insert, remove, select and rank.

7.1 Rank and select

The data represents nanosecond performance measurements for operations called "B-Tree select" and "Spine select" across different data sizes. The performance of Rank is the same as that of select operations, which are implemented the same way in the B-tree.

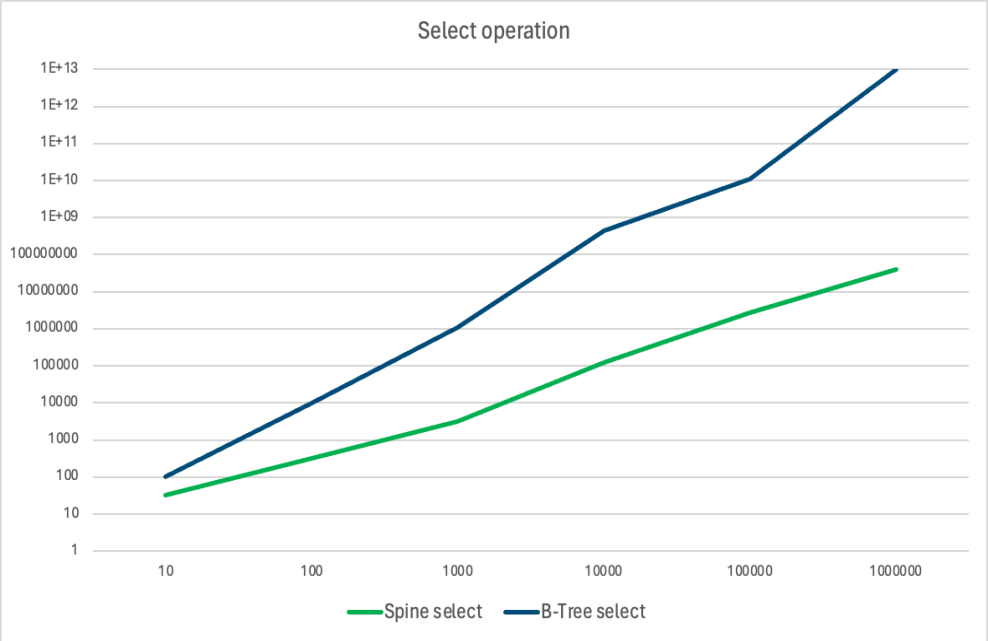

Figure 27. Rank and Select operations.

The time taken for both B-Tree and Spine select operations increases as the dataset size increases, as seen in Figure 27. With smaller inputs, the “Spine select” is anticipated to take less time to complete the select operation than the “B-Tree select”. This expectation is confirmed: With dataset size 10 to 100, the Spine select operation took an average of

172.2263 nanoseconds to complete the select operation, and the B-tree took an average of 5125.739 nanoseconds to complete the select. That indicates that for smaller data sets, the Spine select performs better.

For dataset sizes 1 000 to 10 000, "Spine select" is expected to outperform "B-Tree select". The results confirm this: Spine select with a medium-sized dataset takes, on average, 61917.84 nanoseconds and with the same datasets, the B-tree takes, on average, 218 229 630 nanoseconds, which is about 3525 times slower than the Spine. This indicates that the Spine implementation is more efficient for medium size datasets.

An even greater contrast is expected at the rank of 100 000, where the Spine select operation is anticipated to be faster than the B-tree. On average, the "Spine select" takes 2 746 960 nanoseconds, whereas the "B-Tree select" takes about 10 735 000 000 nanoseconds.

With a data size of 1 000 000, the "Spine select" time increases to 404 416 900 nanoseconds. However, the "B-Tree select" increases to an extremely high value, so within 48 hours, the function has not finished the select operation.

This corrected analysis concludes that "Spine select" consistently outperforms "B-Tree select" across all dataset sizes. As the dataset size increases, "Spine select" scales better, maintaining a lower performance time than "B-Tree select". This trend suggests that the "Spine" structure is best suited for select operations, especially as the data size grows.

7.2 Insert, delete

Figure 28 shows the measurements of the following patterns for the performance of insert and remove operations timed in nanoseconds on B-Trees and Spine data structures across various dataset sizes using sequential integers.

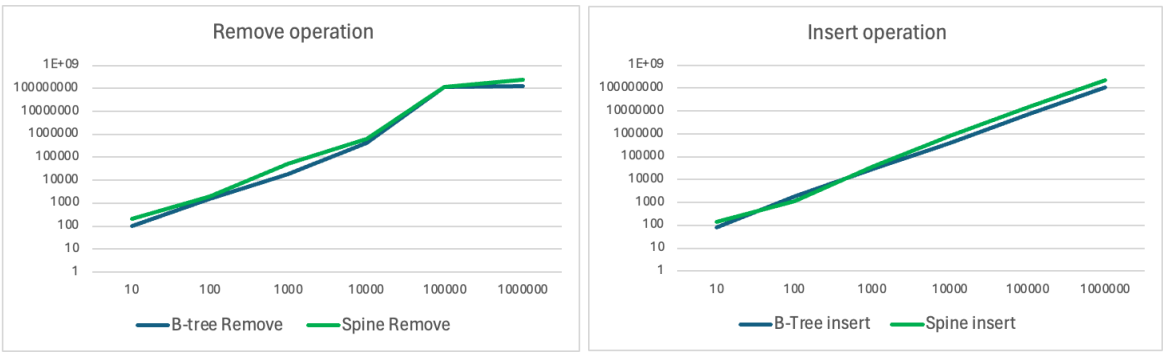

Figure 28. Delete and insert operations.

The insertion and removal operations time for B-Trees and with Spine increases with the size of the dataset. The insert operation times are expected to be similar for both data structures. The results show that the B-Tree and the Spine performance times are similar with small and medium dataset sizes. With bigger dataset sizes, the B-tree outperforms the Spine, which indicates that B-tree insertions are more efficient than the Spine with bigger dataset sizes.

Removal operation time comparison is expected to have similar values in all dataset sizes. As seen in Figure 28, with small and medium datasets, the B-Tree removal operation consistently outperforms the Spine by little. With bigger datasets, the performance of the Spine and B-tree are equal until the dataset size grows to 1 000 000, where the B-tree performs better again.

This data suggests that B-Trees are more efficient than Spine data structures for insertion and removal operations, especially with big data sizes.

These findings could guide decisions on which data structure to use based on the operation frequency and expected application dataset size. For frequent insertions in large datasets, B-Trees may be more suitable, whereas for applications with fewer removals or smaller datasets, the choice might be more balanced.

8 Conclusions

In conclusion, this thesis has examined and compared the performance of two data structures: the traditional B-Tree and the Spine, which integrates a Fenwick Tree for indexing. The results of performance tests indicate that the Spine structure significantly outperforms the B-Tree in select and rank operations, especially as the dataset size increases. This performance advantage is due to the efficient indexing provided by the Fenwick Tree, which allows the Spine to be better able to perform large-scale data select tasks with lower latency and better scalability.

The traditional B-Tree shows better performance in insert and delete operations over a different magnitude of dataset sizes. The B-Tree's structure allows for more efficient handling of these operations, making it more suitable for applications with frequent dataset updates. This efficiency, however, comes at the cost of slower select operations compared to the Spine.

These results show the need to choose data structures based on specific application requirements. Spine proves to be better for applications that need select efficiency, especially with large datasets. On the other hand, for scenarios where insertions and deletions are more critical, B-Trees have better performance, ensuring efficiency even as dataset sizes increase. This performance differentiation highlights the importance of choosing data structures with applications' operational demands and data scalability needs.

The Spine structure offers a compelling alternative for applications that demand high select efficiency and have large datasets that aren't updated often, especially when select and rank operations are crucial. Future research could explore further optimisation of the Spine structure to improve its insertion and deletion operations perfoneed to be. One waymore to do this is to develop algorithms that can efficiently handle batch insertions and deletions. Processing multiple updates in a batch may reduce the overhead associated with individual update operations.

9 References

[1] Lima, B.R.C.A. de (2024). brurucy/indexset. [online] GitHub. Available at: https://github.com/brurucy/indexset [Accessed 8 March 2024].

[2] Lima, B.R.C.A. de (2024). brurucy/ftree. [online] GitHub. Available at: https://github.com/brurucy/ftree [Accessed 8 March 2024].

[3] Jiang, R., Dong, P., Ding, Y., Wei, R., & Jiang, Z. (2023). Thetis: A Booster for Building Safer Systems Using the Rust Programming Language. Applied Sciences, 13(23)

[4] Bugden, W., & Alahmar, A. (2022). Rust: The programming language for safety and performance. arXiv preprint arXiv:2206.05503.

[5] Cushman, M. (2023). Scaled Fenwick Trees. IEEE.

[6] Graefe G. (2011). Modern B-Tree Techniques. Foundations and Trends® in Databases, 3(4)

[7] Lokeshwar, B., Zaid, M. M., Naveen, S., Venkatesh, J., & Sravya, L. (2022, December). Analysis of time and space complexity of array, linked list and linked array (hybrid) in linear search operation. In 2022 International Conference on Data Science, Agents & Artificial Intelligence (ICDSAAI) (Vol. 1). IEEE.

[8] Panchal, B., & Kumar, V. (2016, August). Improving data structure operations for high dimensional data. In 2016 1st India International Conference on Information Processing (IICIP). IEEE.

[9] Frez, K., Oyarzún, M., Inostrosa-Psijas, A., Moreno, F., & Wainer, G. (2023, December). RustSim: A Process-Oriented Simulation Framework for the Rust Language. In 2023 Winter Simulation Conference (WSC). IEEE.

[10] The Rust programming language - The Rust programming language. (n.d.). [online] Available at: https://doc.rust-lang.org/book/title-page.html [Accessed 5 February 2024.]

[11] Variables and Mutability - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html [Accessed 5 February 2024.]

[12] Functions - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch03-03-how-functions-work.html [Accessed 6 May 2024].

[13] References and Borrowing - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html [Accessed 7 February 2024].

[14] The Slice Type - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch04-03-slices.html [Accessed 7 February 2024].

[15] Defining and Instantiating Structs - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch05-01-defining-structs.html [Accessed 7 February 2024].

[16] Method Syntax - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch05-03-method-syntax.html [Accessed 7 February 2024].

[17] Defining an Enum - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html [Accessed 10 February 2024].

[18] The match Control Flow Construct - The Rust Programming Language. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/book/ch06-02-match.html [Accessed 10 February 2024].

[19] Bayer, R., & McCreight, E. (1970, November). Organization and maintenance of large ordered indices. In Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control.

[20] Kleppmann, M. (2017). Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. " O'Reilly Media, Inc.".

[21] BTreeMap in std::collections - Rust. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/std/collections/struct.BTreeMap.html [Accessed 9 March 2024].

[22] search.rs - source. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/src/alloc/collections/btree/search.rs.html [Accessed 9 March 2024].

[23] node.rs - source. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/src/alloc/collections/btree/node.rs.html [Accessed 9 March 2024].

[24] remove.rs - source. (n.d.). Doc.rust-Lang.org. [online] Available at: https://doc.rust-lang.org/src/alloc/collections/btree/remove.rs.html [Accessed 9 March 2024].

[25] Comer, D. (1979). Ubiquitous B-tree. ACM Computing Surveys (CSUR), 11(2).

[26] Emelyanov, P. "Xemul, & Corless, P. (2021, November 23). The Taming of the B-Trees. ScyllaDB. https://www.scylladb.com/2021/11/23/the-taming-of-the-b-trees/ [Accessed 6 May 2024].

[27] Graefe, G., & Larson, P. A. (2001, April). B-tree indexes and CPU caches. In Proceedings 17th International Conference on Data Engineering. IEEE.

[28] Rao, J., & Ross, K. A. (2000, May). Making B+-trees cache conscious in main memory. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data.

[29] GeeksforGeeks. (2018). Rank of all elements in an array. [online] Available at: https://www.geeksforgeeks.org/rank-elements-array/. [Accessed 8 May 2024]

[30] Zeh, N. (2014). Data Structures - Lecture Notes for CS 3110: Design and Analysis of Algorithms. [online] Available at: https://web.cs.dal.ca/~nzeh/Teaching/3110/Slides/ds-lecture-notes.pdf [Accessed 6 May 2024].

[31] Amani, M. (2016). New Combinatorial Properties and Algorithms for AVL Trees (Doctoral dissertation, University of Pisa, Italy).

[32] Introduction to Algorithms Problem Set 5 Solutions Problem 5-1. Skip Lists and B-trees. (2005). [online] Available at: https://ocw.tau.edu.ng/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/assignments/ps5sol.pdf. [Accessed 6 May 2024].

[33] Fenwick, P. M. (1994). A new data structure for cumulative frequency tables. Software: Practice and experience, 24(3).

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Jennifer Veismann,
(*author's name*)

1. grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

The Spine: An efficient two-level dynamic B-Tree with fast selection,
(*title of thesis*)

supervised by Bruno Rucy Carneiro Alves de Lima.
(*supervisor's name*)

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in points 1 and 2.

4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Jennifer Veismann
15/05/2024