UNIVERSITY OF
BUCHAREST
VIRTUTE ET SAPIENTIA

Memory-efficient and easy data structure for dynamic sets

Lucian Bicsi

Coord. prof. Traian Serbanuta

Submitted for the degree of Bachelor in Computer Science

University of Bucharest

June 2018

UNIVERSITY OF BUCHAREST

Lucian Bicsi, Bachelor in Computer Science

Coord. prof. Traian Serbanuta

Memory-efficient and easy data structure for dynamic sets

Summary

In this thesis I will describe a trie-like data structure that implements the functionalities of a dynamic ordered set on integer data types, as well as an extension of the idea that allows us to handle (possibly long) strings.

The data structure will provide the basic set operations of find, insert, erase, as well as more complex operations of ordered sets (successor, predecessor). The data structure would also allow iterating through the underlying values in sorted order in linear time.

Contrary to other trie-like structures that handle integer data, the proposed data structure has a linear memory complexity, with the constant factor of 1.

The thesis will also include a reference to a GitHub page of the project that implements the data structure in C++ and provides means for testing and benchmarking against other similar data structures.

The thesis will tackle the advantages and disadvantages of using the proposed data structure, compared to other similar ones.

Contents

List of Figures

Chapter 1

Introduction

Keeping information about order in structured data is one of the oldest problems in computer science, and a problem that has numerous applications in design of algorithms, architecture of databases, search engines, cache-based optimization, and many more. One of the first references to using trees to handle ordered data dates to as long as close to sixty years ago [1].

The first self-balancing binary search **tree-based** data structure that guarantees a good worst-case complexity is the AVL tree [3] (named after the inventors, Adelson-Velsky and Landis), published in their 1962 paper "An algorithm for the organization of information" [1]. After that, other tree-based data structures were proposed, like splay trees, red-black trees, treaps, scapegoat trees, and so on. These tree-based data structures rely on the comparison model of computation, where performance is measured in memory accesses and comparison count.

Trie-based data structures, on the contrary, rely on the internal representation of the data (seen as a string of some alphabet Σ, usually binary). They operate on the RAM model of computation, and usually can be adapted to have a better performance, at the cost of a higher memory. The first sub-logarithmic data structure to operate on order-sensitive data was first described in 1977, in Peter van Emde Boas's publication "Preserving order in a forest in less than logarithmic time" [4]. The data structure achieves a worst-case complexity of $O(log(M))$ for a universe of integer words of size $O(2^M)$ (i.e., M-bit integers). However, the downside is that the van Emde Boas tree requires $O(2^M)$ memory, which makes it very impractical for regular usage. Recently, more memory-

[1] William C. Lynch's paper "More combinatorial properties of certain trees" [2], opens with: *"Douglas (1959), Windley (1960), and later Hibbard (1962) introduced binary search trees and gave their applications to sorting, searching, and file maintenance"*.

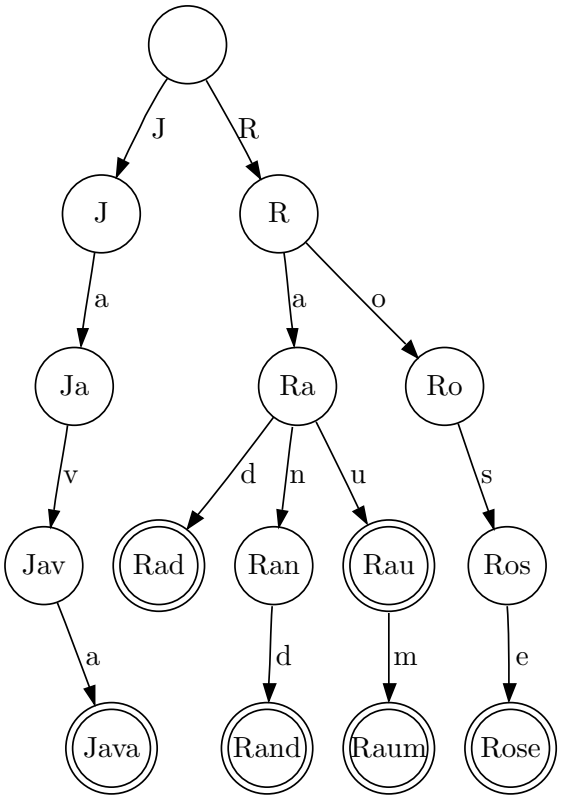


Figure 1.1: Trie for a set of words

efficient data structures (x-fast trees, y-fast trees, fusion trees) have been invented.

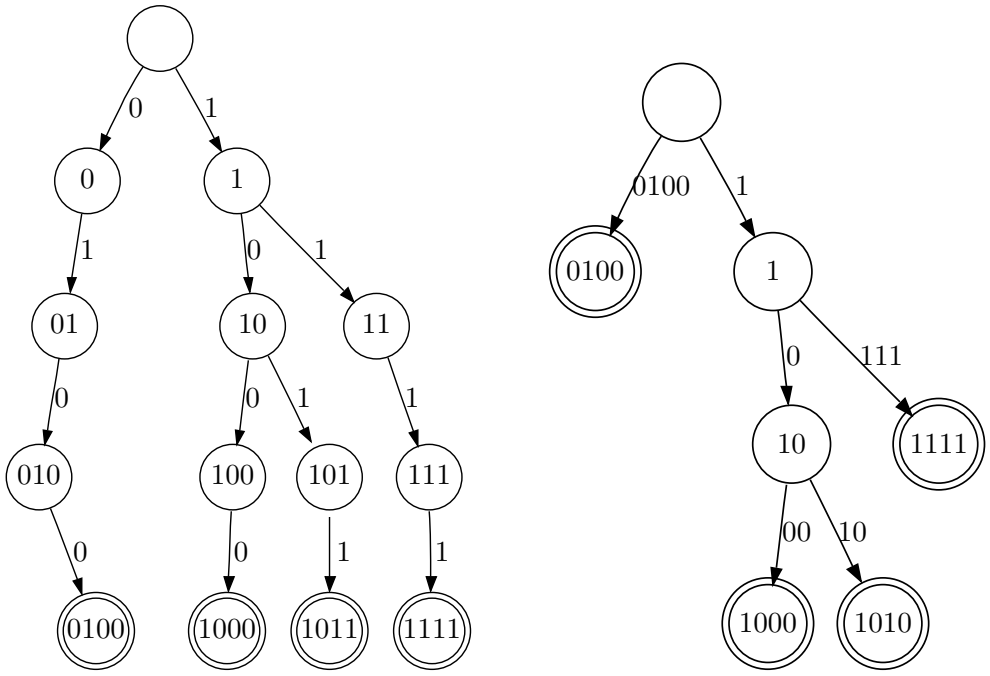
1.1 Problems with bitwise tries

Tries are data structures that operate on strings of arbitrary length. The trie data structure is very useful for exact string search, providing search suggestions, and organizing computer file systems. A representation of a trie on the set

$$S = \{\text{Java, Rad, Rand, Rau, Raum, Rose}\}$$

can be seen in Figure 1.1.

A **bitwise trie** on a set S of integers of M bits is a regular trie on the same set S, where we consider each integer as a binary string of length M. While the bitwise trie provides an easy and fast way to implement dynamic ordered set operations in $O(M)$ for M-bit words, we will see that it has a very big memory overhead. For example, consider the example of the bitwise trie on the set $S' = \{4[0100], 8[1000], 11[1011], 15[1111]\}$ of integers, illustrated in Figure 1.2 (a). Each integer in the set S' resides in a leaf node, while each intermediary node only contains the links to the two children and no extra information. One may notice that, for each integer in some set S of M-bit integers, there



(a) Regular bitwise trie (b) Its corresponding compressed trie

Figure 1.2: Bitwise trie for a set of 4-bit integers and its compression

will be at most M new nodes created. Therefore, the space required for this strategy is of order $O(Mn)$, for a set S of size n.

One possible improvement over the big space requirement is to consider doing **trie compression** on the bitwise trie, resulting in a data structure called **radix tree**. Compressing a tree relates to essentially keeping only intermediary nodes that have two or more children. This reduction can be proven to reduce the number of total nodes to $2n - 1$ for a set S of size n. However, this requires changing the child labels from characters (bits) to strings of characters, and is in particular hard to be implemented in a **dynamic** setup that would allow insertions and deletions. An example of compressing a trie is illustrated in Figure 1.2 (b).

Even considering **radix trees**, there is a significant memory overhead. For a set S of size n, $2n - 1$ trie nodes have to be stored, and each node has to have two pointers to its children, as well as **the full (string) representation** of the edge label. In conclusion, there will be at least two pointers and two integers stored per node [2].

[2]Note that, if a naive implementation of the radix tree is made, there can be as much as four integers and four child pointers per element in the set, which relates to a total of eight 64-bit words extra memory per element. This can be especially inefficient for algorithms handling a big amount of data or machines that have less memory to operate with.

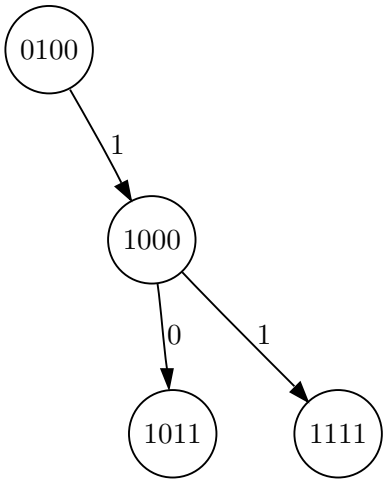


Figure 1.3: `PreTree` on a set of 4-bit integers.

1.2 Proposed data structure

The data structure described in this thesis is a **trie-based** data structure that can operate on integers, as well as a modification of it for strings of equal length. We will see that the main advantages of the proposed data structure are its low memory overhead, and its ease of implementation. The difference between the proposed data structure and a regular trie is that we decide to take advantages of the intermediary nodes, to store values inside them, not only inside leaves.

This will allow our data structure to allocate just as many nodes as there are elements in the set, which is an optimistic minimum for any data structure. Another advantage is that our data structure will not have to handle the complicated case of having string labels on edges, and having to split and merge nodes at insert/erase operations. Yet another advantage is that nodes within our trie will only store two pointers to their corresponding children, and no extra information is needed.

We decided to name the data structure `PreTree`, not only because of its simplicity and beauty, but also because we will see in the following chapter that it is tightly related to the **preorder traversal** of a rooted tree. An example of a `PreTree` on the same set of integers as above is illustrated in Figure 1.3.

Throughout the next chapter we will discuss what strategy we should choose for the elements that will reside in the intermediary nodes, as well as describe easy and straightforward implementations of the basic set operations (**find, insert, erase**), as well as (not-so-straightforward) implementations of the more complex set operations (**successor, predecessor**). This will, in essence, prove that our proposed data structure has the same computational power as any other ordered data structures provided in literature.

1.3 Preliminaries

In this section, we will define the terms and notions used throughout the thesis. One may safely skip this section if one is familiar with the general notions related to rooted trees, ordered set operations, and so on. We will relate to these definitions in the following chapters.

1.3.1 Graph theory

Definition 1.3.1. *An **undirected graph** G is a tuple (V, E) where V is a finite set, called the **set of vertices**, and $E \subseteq V \times V$ is a symmetric, anti-reflexive binary relation, called the **set of edges**.*

Definition 1.3.2. *Let $G = (V, E)$ an undirected graph, and $v \in V$. We will denote by $adj(v)$, called the **adjacent nodes** of v the set W of vertices such that $w \in W$ if and only if $(v, w) \in E$.*

Definition 1.3.3. *A **tree** is an undirected graph $G = (V, E)$ where, for any two vertices $u, v \in V$, $u \neq v$, there exists **exactly one** enumeration $w_1, w_2, ..., w_n$ of vertices (simple path) such that:*

- $w_1 = u$
- $w_n = v$
- $w_i \neq w_j$, *for all* $i \neq j$
- $(w_i, w_{i+1}) \in E$, *for all* $1 \leqslant i < n$

Definition 1.3.4. *A **rooted tree** $\mathcal{T}$ is a tuple (V, E, r), where (V, E) is a tree, and r is a special vertex, called the **root**.*

Definition 1.3.5. *Consider a rooted tree $\mathcal{T} = (V, E, r)$ and v a vertex, $v \neq r$. There is a unique simple path $r = w_1, w_2, ..., w_{n-1}, w_n = v$, and $n > 1$. The unique vertex w_{n-1} is called the **parent** of v in $\mathcal{T}$. We will denote this as $w_{n-1} = parent(v)$. Therefore,*

$$parent : V \setminus \{r\} \to V$$

is a well-defined function.

Definition 1.3.6. *Consider a rooted tree $\mathcal{T} = (V, E, r)$ and v a vertex. We will define the function $children(v) : V \to \mathcal{P}(V)$,[3] as follows:*

[3] For a set V, $\mathcal{P}(V)$ (sometimes denoted 2^V) is the set of all subsets $U \subseteq V$, including the empty set ϕ. This is also known as the **power set** of V. If V is finite, the size of its power set $\|\mathcal{P}(V)\|$ is equal to $2^{\|V\|}$.

- $children(r) = adj(r)$
- $children(v) = adj(v) \backslash \{parent(v)\}$, *for all* $v \neq r$

Definition 1.3.7. *A* ***labeled rooted tree*** *is a tuple* $\mathcal{T} = (V, E, \Sigma, r, l)$*, where* (V, E, r)*, is a* ***rooted tree****,* Σ *is a set called the* ***label set*** *(or alphabet), and*

$$l : E \to \Sigma,\ l(u, v) = l(v, u), \forall (u, v) \in E$$

is a function, called the ***edge label function****.*

Definition 1.3.8. *A labeled* ***tree node*** T *on an alphabet* Σ *is a compound structure defined recursively, based on the following axioms:*

- *The empty (leaf) node* T_ϕ *is a* ***tree node***
- *Any non-empty (intermediary) node* T *has* $\|\Sigma\|$ *tree nodes associated (one for each element on* Σ*), called the* ***children nodes*** *of* T*, noted* $children(T)$*.*

Remark. *From this point on, we will drop the* ***labeled*** *adjective from ; however, the reader should note that, if not mentioned explicitly, every time we mention a* ***rooted tree****, we refer to a* ***labeled rooted tree*** *instead.*

Lemma. *There is a isomorphism between* ***rooted trees*** *and* ***tree nodes****.*

We will not provide a proof for this lemma, leaving it as an exercise for the reader. However, the intuition behind the lemma is that a rooted tree $\mathcal{T} = (V, E, \Sigma, r, l)$ can be identified by a root node representing r, with its child nodes representing the corresponding children of r, and so on. Whenever a vertex $v \in V$ has no child with a corresponding edge label λ, we will attach the empty node T_ϕ as the corresponding child of v's node representation.

Remark. *Because of the bijection between rooted trees and tree nodes, we will use these two terms interchangeably throughout the thesis. In most cases, when we talk about rooted trees, we will in fact, refer to their tree node representations.*

Definition 1.3.9. *A binary rooted tree (tree node)* T *is a rooted tree where each non-empty (intermediate) node has* ***exactly two children****. We will denote these by* $T.left$*, and* $T.right$*, respectively. Alternatively, a binary rooted tree is a rooted tree on a binary alphabet* $\Sigma \cong \{0, 1\}$*.*

Throughout the following sections, all binary tree vertices (and binary tree nodes, consequently), will have keys associated with them. For a node T, we will refer to the key inside T as $T.key$.

Definition 1.3.10. *Consider a rooted binary tree T. A **pre-order traversal** (sometimes called node-left-right or NLR traversal) is an ordered sequence (list) of nodes, defined recursively:*

- $\textsc{PreOrder}(T_\phi) = [\,]$ *(empty list)*
- $\textsc{PreOrder}(T) = T + \textsc{PreOrder}(T.left) + \textsc{PreOrder}(T.right)$

*where $+$ denotes **list concatenation**.*

Remark. *The notion of **pre-order traversal** can be naturally extended to an arbitrary (n-ary) rooted tree, if there is an order defined on the label set Σ, by concatenating the result of applying pre-order traversal to all of the node's children in the proper order.*

1.3.2 Ordered sets

Definition 1.3.11. *Let U be a finite set of elements (not necessarily integers) called the **universe of keys**, $<$ a **total order** on U, and $S \subseteq U$. The tuple $\mathcal{S} = (S, <)$ (denoted, for simplicity, S) is called an ordered set of keys from U.*

Definition 1.3.12. *Let S be an ordered set of keys, and k a key (not necessarily in S). The **successor** of k is defined to be the smallest key k_+ with the following properties:*

- $k_+ \in S$
- $k < k_+$

Definition 1.3.13. *Let S be an ordered set of keys, and k a key (not necessarily in S. The **predecessor** of k is defined to be the greatest key k_- with the following properties:*

- $k_- \in S$
- $k_- < k$

Chapter 2

Bitwise trie on integers

As argued in the previous chapter, the key strategy that makes the `PreTree` data structure memory efficient is keeping information about one of the elements in the set inside the root node. In theory, there is no restriction as to what element we should choose to keep.

However, there are certain ideas that come to mind, each of them having advantages and disadvantages. During the working process for this thesis, we have considered two main heuristics, and examined their advantages and disadvantages. One of them would be keeping **minimum value** (maximum value) in the root of the tree, while the other would be keeping **any value**. However, we will only describe the former, as it turned out to be the more successful approach of the two.

2.1 The min invariant

Because we have the freedom of picking any value for the root of the tree (and its subtrees), one idea would be to keep the **minimum** (maximum) one from within the tree (subtree). One would argue that having this extra invariant will make the implementation harder, although we will see that the opposite turns out to be true in this case.

From this point on, we will refer to this property of the `PreTree` as the **min invariant**. We will use this property throughout our implementation, and we will make sure that any operation which mutates the data structure will preserve this key invariant.

Formally, the **min invariant** can be stated as such:

Definition 2.1.1. *A binary (n-ary) rooted tree T has the **min invariant** property if and only if one of the two conditions holds:*

- *It is the empty tree T_ϕ*

- *It is non-empty, and for all $T' \in children(T)$ we have that $T.key < T'.key$ and that T' has the **min invariant** property*

2.2 Motivation

We will see that keeping the minimum value in the root of the tree will provide us a lot of benefits at seemingly no extra complexity.

One key fact about keeping the minimum value in the root at all times is that the data structure will become **fully ordered**, and traversing the values in increasing order will be equivalent to a preorder traversal of the tree.

Another important fact about a `PreTree` is that the min invariant makes it so that it is not only a **search tree**, but also a **min heap**. The proof is an immediate consequence of the extra invariant kept. This means that, intuitively, we can expect the `insert` and `erase` operations to be similar, in some sense, to the pushing and popping operations of min heaps.

2.3 Preliminaries

In this section we will provide definitions to terms that will be used later along the chapter. Note that + denotes list concatenation.

Definition 2.3.1. *Let T be a `PreTree` and k a key. We will denote by* $\textsc{KeyPath}(T, k)$ *the following ordered list of nodes (defined recursively):*

- $\textsc{KeyPath}(T_\phi, k) = [\,]$ *(empty list)*
- $\textsc{KeyPath}(T, \overline{0k_2...k_n}) = T + \textsc{KeyPath}(T.left, \overline{k_2...k_n})$
- $\textsc{KeyPath}(T, \overline{1k_2...k_n}) = T + \textsc{KeyPath}(T.right, \overline{k_2...k_n})$

Informally, Definition 2.3.1 describes the path of nodes visited by descending down the tree T, going down the links dictated by the bits of k in order. This is very similar to the process of descending down a regular trie.

Definition 2.3.2. *Let T be a `PreTree`.*

1. *We will denote by* $\textsc{LeftPath}(T)$ *the following ordered list of nodes (defined recursively):*
   - $\textsc{LeftPath}(T_\phi) = [\,]$ *(empty list)*
   - $\textsc{LeftPath}(T) = T + \textsc{LeftPath}(T.left)$, ***if*** $T.left \neq T_\phi$

- LeftPath(T) = T + LeftPath($T.right$), ***otherwise***

2. *We will denote by* RightPath(T) *the following ordered list of nodes (defined recursively):*

   - RightPath(T_ϕ) = [ ] *(empty list)*
   - RightPath(T) = T + RightPath($T.right$), ***if*** $T.right \neq T_\phi$
   - RightPath(T) = T + RightPath($T.left$), ***otherwise***

Informally, Definition 2.3.2 describes the path of nodes visited by descending down the tree T, always preferring to go down the left link (in the case of LeftPath), or the right link (in the case of RightPath).

2.4 An example

Consider the following set S of 3-bit numbers:

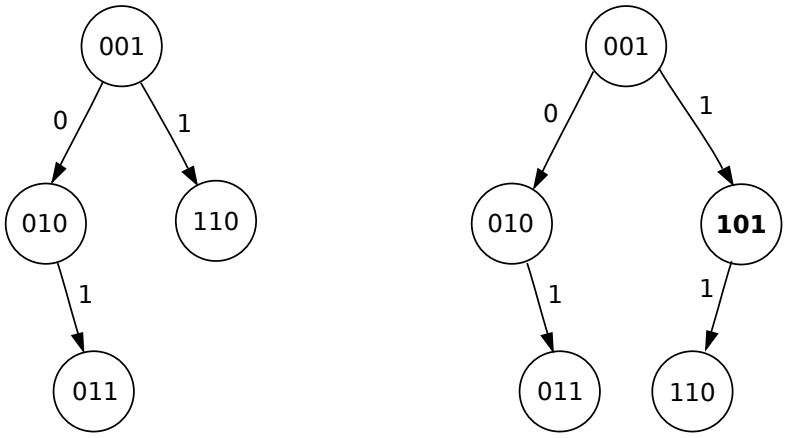
$$\{1[001], 2[010], 3[011], 6[110]\}$$

Let us try to build a `PreTree` T on the set S. Because of the min invariant, the root value of the tree T can only be value 1. After this, we are left with values $2, 3, 6$. Because 2 and 3 have the most significant bit equal to 0, they will form the left son of T, whereas 6 will form the right son of T. After inserting all the values, T will look like the left picture in Figure **??**.

Let us now consider an `insert` operation with the value 5. One may notice that some nodes on the path from root along the bits of number 5 are unchanged; however, once 5 is the minimum value in the sub-tree, then it should become the root, so the current root needs to be "demoted". In this case, the value 6 has to be put one level lower (Figure 2.1 - right picture).

2.5 Basic set operations

In this section we will describe the implementation of the basic set operations of **finding a key**, **inserting a key**, and **erasing a key**. We will see that every function can be implemented in time proportional to the height of the tree, so every such operation has complexity $O(M)$ for a key universe of size 2^M.



(a) Before (b) After

Figure 2.1: `PreTree` insert operation

2.5.1 Finding a key

The most important operation for a set is **finding** a key. The important observation in order to implement this operation is that the sought key can only lie on its corresponding **key path** (see Definition 2.3.1).

This leads us to the very straightforward algorithm, described in pseudo-code in Algorithm 1. Note that this approach makes a key comparison at each level of the tree, and that will lead to a sub-optimal complexity for keys that cannot be compared in $O(1)$ time. This will be optimized in Chapter 3.

2.5.2 Pushing and popping

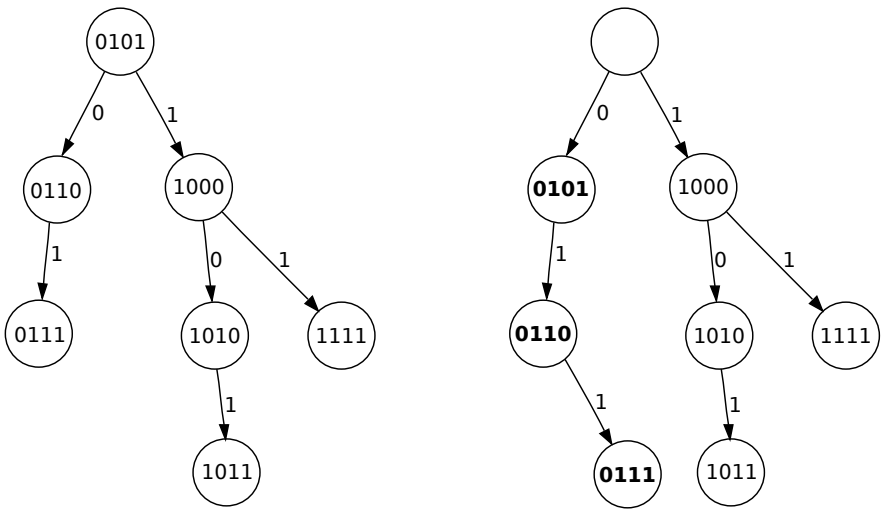
In this section we will define the operation of **popping** the root of a `PreTree` T, effectively erasing it from T and its counterpart of **pushing** the root's key down the tree, making the root available for insertion. These operations will be essential for implementing the set operations of insertion and deletion.

We can see that popping and pushing on T will work by **shifting** a root-leaf path either up or down. Moreover, in a pop operation, as the minimum value will always be on the "left-est" non-empty child, this means that we need no extra comparisons (compared to a regular min-heap, where we would need to first determine which node to pull from, and just shift the values up along the **left path** of T (see Definition 2.3.2).

However, an easier way to look at **pushing** and **popping** is in terms of recursive

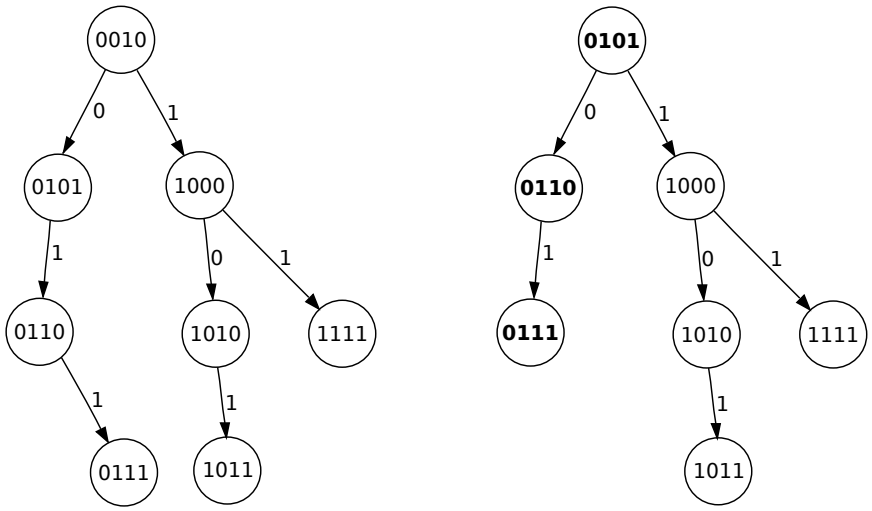
Algorithm 1 Finding a key k on a `PreTree` T of level l

```
1:  function FIND(T, k, l)
2:      if T = T_φ or T.key > k then                ▷ Base case 1: Unsuccessful
3:          return T_φ                               ▷ Key k is not in T
4:      end if
5:      if T.key = k then                           ▷ Base case 2: Successful
6:          return T                                 ▷ Found key k
7:      end if
8:      if EXTRACTBIT(k, l) = 0 then
9:          return FIND(T.left, k, l + 1)
10:     else
11:         return FIND(T.right, k, l + 1)
12:     end if
13: end function
```



(a) Before (b) After

Figure 2.2: `PreTree` push operation



(a) Before (b) After

Figure 2.3: `PreTree` pop operation

functions on tree nodes. In the next part we will present the algorithms for the two operations.

Pushing

The first thing to notice is that pushing a key onto an empty tree will result in a **singleton** tree of that key (a tree that has only one node, which contains the key to be pushed).

If the tree T is not empty, then the root of the tree after pushing will have to have the key to be pushed (by design). At this point, we can replace the old root's key with the key to be pushed, and then **recursively** push the old root's key into the correct sub-tree of T.

The function pseudo-code is described in Algorithm 2. Note that in the reassignments of keys, we can take advantages of the move semantics, in case our keys have a bigger size.

Popping

Popping is the inverse operation of pushing, so it is expected to work similarly. Indeed, there is an easy recursive implementation of it.

Once again, we will start with the base case. Because in this case the root will be deleted, we can argue that the base case is either **an empty tree** or **a singleton tree**. In this case, we will always return the empty tree.

Otherwise, we will have to remove the root's key from the tree T. However, because the tree has children, some other key from the sub-tree has to become the new root's key.

Algorithm 2 Pushing a key k on a `PreTree` T of level l

```
1:  function PUSH(T, k, l)
2:      if T = T_φ then                                  ▷ Base case: empty tree
3:          return SINGLETON(k)                  ▷ Return a tree composed of just the key k
4:      end if
5:      if EXTRACTBIT(T.key, l) = 0 then
6:          T.left ← PUSH(T.left, T.key, l + 1)          ▷ Push the root's key onto left son
7:      else
8:          T.right ← PUSH(T.right, T.key, l + 1)        ▷ Push the root's key onto right son
9:      end if
10:     T.key ← k                                        ▷ The new root's key becomes k
11:     return T
12: end function
```

Because of the minimum invariant, it is easy to see that the only candidate for the root's key is the leftmost non-empty child's key. In this case, we will replace the root's key with the child's key and recursively pop that child.

The function pseudo-code is described in Algorithm 3. As in the pushing case, the reassignments of keys can be implemented using move semantics, for performance.

2.5.3 Inserting and erasing

Once we have defined the **push** and **pop** operations, we will use them in order to implement insertions and deletions. Each of them will require descending into the right subtree, then applying a push or a pop operation to that subtree.

Inserting

As seen before in Figure **??**, in order to insert a new key k into a tree T, we will have to descent down the the key's unique path, until the key k becomes strictly less than the current sub-tree's root key.

One important note is that, during the descent, at the last step we might encounter that the root's key matches k. In this case, the key to be inserted already exists in T, and nothing happens.

The function pseudo-code is described in Algorithm 4. An important note is that the current version requires a comparison of inequality and equality at each level [see lines 2-7], which can be optimized to having only one comparison of words. (Chapter 3)

Algorithm 3 Popping a `PreTree` T

```
1: function POP(T)
2:     if T.left ≠ T_φ then                          ▷ Start by checking the left sub-tree
3:         T.key ← T.left.key                        ▷ Root's key becomes left child's key
4:         T.left ← POP(T.left)                      ▷ Recursively pop left child
5:         return T
6:     end if
7:     if T.right ≠ T_φ then                         ▷ The left sub-tree is empty at this point
8:         T.key ← T.right.key                       ▷ Root's key becomes right child's key
9:         T.right ← POP(T.right)                    ▷ Recursively pop right child
10:        return T
11:    end if
12:    return T_φ                                    ▷ T is either empty, or a leaf at this point
13: end function
```

Algorithm 4 Inserting a key k into a `PreTree` T of level l

```
1: function INSERT(T, k, l)
2:     if T = T_φ or T.key > k then                  ▷ Base case 1: Successful insertion
3:         return PUSH(T, k, l)                      ▷ We found where our key belongs in T
4:     end if
5:     if T.key = k then                             ▷ Base case 2: Unsuccessful insertion
6:         return T                                  ▷ The key is already in the tree, just return
7:     end if
8:     if EXTRACTBIT(k, l) = 0 then
9:         T.left ← INSERT(T.left, k, l + 1)         ▷ Insert key onto left son
10:    else
11:        T.right ← INSERT(T.right, k, l + 1)       ▷ Insert key onto right son
12:    end if
13:    return T
14: end function
```

Erasing

In order to erase a key k from a `PreTree` T, we will first descend down the key's path in the tree (as before, with insertion), and once we have found our key in T, we will pop it from the tree.

Again, we might find that the key k does not exist in T, in which case nothing should happen.

The function pseudocode is described in Algorithm 5. As you can see, the erase function has a strikingly similar structure with the insert function. This will be very helpful, as we can condense both operations in a more generic one, just by setting different callbacks (Algorithm 7).

Once again, the erase operation requires a large number of key comparisons, which might be inefficient for large keys; however, the same solution that solves insertions will solve erases as well. (Chapter 3)

Algorithm 5 Erasing a key k from a `PreTree` T of level l

```
1:  function ERASE(T, k, l)
2:      if T = T_φ or T.key > k then                ▷ Base case 1: Unsuccessful erase
3:          return T                  ▷ The key does not exist in the tree, just return
4:      end if
5:      if T.key = k then                            ▷ Base case 2: Successful erase
6:          return POP(T)                            ▷ Remove the key from the tree
7:      end if
8:      if EXTRACTBIT(k, l) = 0 then
9:          T.left ← ERASE(T.left, k, l + 1)              ▷ Erase key from left son
10:     else
11:         T.right ← ERASE(T.right, k, l + 1)           ▷ Erase key from right son
12:     end if
13:     return T
14: end function
```

2.5.4 A more generic approach

One may notice that the implementations for FIND (Algorithm 1), INSERT (Algorithm 4) and ERASE (Algorithm 5) have a lot of common structure. In this case, a wise idea is to build a more generic function which has callbacks for the base cases, and implement our

functionality as an interface to that function.

For more clarity, refer to the implementation in Algorithm 7.

This implementation is functionally equivalent to the algorithms described above; however, we will prefer this implementation, as not only it is shorter, but also because it will give us a clearer view when analyzing the algorithm's performance.

Algorithm 6 A generic helper function for `PreTree`'s basic functionality

```
1: function Apply(T, k, l, f_FOUND, f_NOT_FOUND)
2:     if T = T_φ or T.key > k then                    ▷ Base case 1: Not found
3:         return f_NOT_FOUND(T, l)                    ▷ Apply first callback
4:     end if
5:     if T.key = k then                               ▷ Base case 2: Found
6:         return f_FOUND(T, l)                        ▷ Apply second callback
7:     end if
8:     if ExtractBit(k, l) = 0 then
9:         T.left ← Apply(T.left, k, l + 1, f_FOUND, f_NOT_FOUND)
10:    else
11:        T.right ← Apply(T.right, k, l + 1, f_FOUND, f_NOT_FOUND)
12:    end if
13:    return T
14: end function
```

2.5.5 Traversing

Because the `PreTree` is essentially a preorder search tree, applying a function to its keys in increasing order is essentially equivalent to doing a **preorder traversal** of it. The pseudo-code is described in Algorithm 8.

Algorithm 8 Call a procedure f over all keys of T in increasing order

```
1: function ForEach(T, f)
2:     if T ≠ T_φ then
3:         call f(T.key)          ▷ Call f for root key (minimum value, by invariant)
4:         ForEach(T.left, f)                          ▷ Recurse into left child
5:         ForEach(T.right, f)                         ▷ Recurse into right child
6:     end if
7: end function
```

Algorithm 7 Basic set operations using the generic helper (6)

```
1: function FIND(T, k, l)
2:     ret ← T_φ
3:     f_FOUND ← (T', l') ⇒ (ret ← T', return T')
4:     f_NOT_FOUND ← (T', l') ⇒ (return T')
5:     APPLY(T, k, l, f_FOUND, f_NOT_FOUND)
6:     return ret
7: end function
8:
9: function INSERT(T, k, l)
10:    f_FOUND ← (T', l') ⇒ (return T')
11:    f_NOT_FOUND ← (T', l') ⇒ (return PUSH(T', k, l'))
12:    return APPLY(T, k, l, f_FOUND, f_NOT_FOUND)
13: end function
14:
15: function ERASE(T, k, l)
16:    f_FOUND ← (T', l') ⇒ (return POP(T'))
17:    f_NOT_FOUND ← (T', l') ⇒ (return T')
18:    return APPLY(T, k, l, f_FOUND, f_NOT_FOUND)
19: end function
```

2.6 Supporting predecessor, successor queries

In this section we will describe how the `PreTree` can support more complex search operations, as finding the **successor** and **predecessor** of a given key k (see Definitions 1.3.12, 1.3.13).

Unfortunately, supporting predecessor and successor queries on a `PreTree` is trickier than the previous operations. Another downside is that, because our data structure is **left-heavy** (meaning that lower values are given a higher priority against higher values), we should expect the need of an asymmetric approach to **successor** and **predecessor**..

2.6.1 Successor

Our main strategy is to prove that finding the successor of a given key k, just like in the **find** operation, can be reduced to checking against **a small set of candidate keys**.

Let us consider a key k and its path (Definition 2.3.1). The first thing to notice is that anything that is to the left of the key path contains keys that are strictly smaller than k. We will consider two distinct cases:

1. k is less than the last element in KeyPath(k)
2. k is not less than the last element in KeyPath(k)

In the first case, we claim that the successor of k lies inside KeyPath(k). Indeed, consider the following:

$$\textsc{KeyPath}(k) = [l_1, l_2, ..., l_n]$$

$$l_1 < l_2 < ... < l_i \leqslant k < l_{i+1} < ... < l_n$$

Because we are in the first case, we have that $0 \leqslant i < n$. Now suppose the successor of k is not l_{i+1}. Then the successor of k cannot be inside l_{i+1}'s subtree (due to the min invariant), and it cannot be in KeyPath(k). The only case left is that there is a **left link** $l_j \to l_{j+1}$ for some $j \leqslant i$ for which the successor lies inside l_j's **right link**'s subtree (i.e., the successor is l_j's right link, due to the **min invariant**). Then l_{i+1} is of form $\overline{b_1 b_2 ... b_{j-1} 0...}$ and successor of k is of form $\overline{b_1 b_2 ... b_{j-1} 1....}$. Therefore $k < l_{i+1} < \textsc{Succ}(k)$, contradiction.

In the second case, due to a similar argument as before, we find that the successor of k is, in fact, the **deepest** node for which $l_j \to l_{j+1}$ is a **left link** and there exists some **right link** $l_j \to l*$. In this case, $l* = \textsc{Succ}(k)$.

The pseudo-code for the Successor function is described in Algorithm 9.

Algorithm 9 Find successor of k in a `PreTree` T of level l

```
1: function SUCCESSOR(T, k, l)
2:     if T = T_φ then
3:         return T_φ                          ▷ No successor for an empty tree
4:     end if
5:     if T.key > k then
6:         return T                            ▷ Case 1: found successor in KEYPATH
7:     end if
8:     if EXTRACTBIT(k, l) = 0 then
9:         ret ← SUCCESSOR(T.left, k, l + 1)
10:        if ret = T_φ then                   ▷ If successor is not found down the KEYPATH
11:            return T.right                  ▷ Case 2: output right child
12:        else
13:            return ret
14:        end if
15:    else
16:        return SUCCESSOR(T.right, k, l + 1)
17:    end if
18: end function
```

2.6.2 Predecessor

Due to our choice of keeping a **min invariant** (as opposed to a max invariant), making our tree a preorder search tree (as opposed to a postorder search tree), finding the predecessor of a key is not as easy as finding its successor. However, we will adopt the same strategy of **reducing the number of candidate keys**.

Let us, again, consider the key path of k:

$$\textsc{KeyPath}(k) = [l_1, l_2, ..., l_n]$$

$$l_1 < l_2 < ... < l_i < k \leqslant l_{i+1} < ... < l_n, 0 \leqslant i \leqslant n$$

Naturally, the only case where k has no predecessor is when $i = 0$. Otherwise, we have no reason to consider candidates inside $\textsc{KeyPath}(k)$ other than l_i. Similarily as before, anything to the right of $\textsc{KeyPath}(k)$ is strictly larger than k, so it should not be considered. The only candidates that remain lie inside a subtree of $l*$ for which $l_j \rightarrow l*$ is a **left link** and $l_j \rightarrow l_{j+1}$ is a **right link**, and $j \leqslant i$. However, a stronger claim can be made, as before. It is enough to look at the left subtree of l_i, as any nodes lying on the left of the key path above l_i will be smaller than l_i, and l_i is a candidate solution.

Moreover, in this case, the last node in $\textsc{RightPath}(l*)$ (Definition 2.3.2) is the successor of k.

The algorithm for finding the predecessor is described in pseudo-code in Algorithm 11.

Algorithm 10 Find the maximum key inside T

```
1:  function RightMost(T)
2:      if T = T_φ then
3:          return T_φ
4:      end if
5:      if T.right ≠ T_φ then
6:          return RightMost(T.right)          ▷ Always prefer going right
7:      else
8:          return RightMost(T.left)
9:      end if
10: end function
```

Algorithm 11 Find predecessor of k in a `PreTree` T of level l

```
1: function PREDECESSOR(T, k, l)
2:     if T = T_φ or T.key ⩾ k then
3:         return T_φ                          ▷ Nothing in T's sub-tree is of interest
4:     end if
5:     ret ← T_φ
6:     if EXTRACTBIT(k, l) = 0 then
7:         ret ← PREDECESSOR(T.left, k, l + 1)
8:     else
9:         ret ← PREDECESSOR(T.right, k, l + 1)
10:        if ret = T_φ then                                   ▷ We found l_i
11:            ret ← RIGHTMOST(T.left)                 ▷ Try with the left sub-tree
12:        end if
13:    end if
14:    if ret = T_φ then
15:        return T                ▷ If everything failed, return the only candidate left
16:    else
17:        return ret
18:    end if
19: end function
```

Chapter 3

Extension to strings

In the previous chapter, we have discussed the implementation of the `PreTree` data structure as a **ordered set**. However, as the data structure is inherently a trie, a natural step forward is to analyze the possibility of extending its usage to arbitrarily long data types, namely strings.

In this chapter we will describe an extension for `PreTree`, which we will call `PreTrie`, that would allow us to handle bigger-size words like **strings**. During this chapter we will focus on the special case of **binary strings** (alphabet of size 2). However, the approach described can be adapted to handle strings of any alphabet size. [3.6]

We will also discuss the limitations of this extensions, when handling strings of varying length, and possible ideas of overcoming this issues.

3.1 Challenges

There are two main challenges that we have to overcome in order to support the data structure on strings.

Up until this point, we have assumed $O(1)$ complexity of comparing two keys. However, **this is not the case anymore**, as one would require $O(min(n, m))$ time to compare two strings of length n and m respectively.

The problem that one would face when naively extending the implementation described in the previous chapter to the case of strings is that some algorithms that require a large number of comparisons would run quadratically in the length of the string (e.g. Algorithm 6, which is the core of all the basic set operations)

Another assumption that will not hold during this chapter is that words have an equal length. We will see that this will pose serious problems to our complexity bounds, and we

will try to provide a solution in Section 3.5 of this chapter.

3.2 Assumptions

When discussing memory in the string case, there is an inevitable question as to whether or not we should take into account the strings themselves as extra memory. This was not an issue in the `PreTree` case, as integers were considered $O(1)$ memory. In this case, however, there are some assumptions that we make:

1. **The input is read-only**. We cannot modify the input. Any modification of the input would mean effectively using extra memory, therefore we will not allow that.

2. **The input is available quickly at all times**. The input strings must be able to be accessed by reference at all times in $O(1)$ complexity.

3. **The input strings are character arrays**. More so, the input string must have $O(1)$ random access to their characters (bits).

4. *The input is binary.* This is for the ease of explaining the following algorithms, and for reusability of the algorithms described in the last chapter; we will see that this is assumption can be withdrawn, as most times it is better to work with bigger alphabets, rather than binary.

Under these assumptions, we can safely argue that the `PreTrie` has the same memory constraints as the `PreTree`, therefore we get the same memory efficient data structure for longer keys.

3.3 The String Insertion Problem

Before describing a solution to our challenges, let us consider an easier preliminary problem, which we will refer to as the **String Insertion Problem** (SIP).

Problem. *Consider a list L of N distinct strings $s_1, ..., s_N$* ***sorted in lexicographic order*** *and a query word t. Find i $(0 \leqslant i \leqslant N)$ such that $s_1 < s_2 < ... < s_i < t \leqslant s_{i+1} < ... < s_N$. (String Insertion Problem)*

Remark. *As one might notice, the choice for the name of the problem is not coincidental: the problem requires finding the position where the new string t should be inserted into a list of sorted strings, in order to preserve their order.*

3.3.1 Solving the SIP problem

One naive approach of solving the SIP is comparing each string to the query string, which takes $O(\|t\|N)$ time. A better idea is to use the fact that the list of the strings is sorted, and binary search for the position, leading to a better time bound of $O(\|t\|log(N))$. In this section we will claim a different time bound, which will be more beneficial for the implementation of the `PreTrie`:

Lemma. *The String Insertion Problem can be solved in time* $O(N + \|t\|)$.

We will prove this lemma by describing an algorithm which solves the SIP, and then proving its time complexity.

3.3.2 Reduction to equal length

First, let us consider an extra character \$, and append \$ to t. After this, we will make all strings of length equal to t, either by appending \$ to the end of them (if shorter), or by truncating them. In the implementation of the algorithm, we will not actually do this, but only take this transformation into account while accessing the strings.

L = [ab, baac, baaca] t = baa

$L' \leftarrow$ [ab\$\$, baac, baac] $t' \leftarrow$ baa\$

Figure 3.1: Input before and after preprocessing

Note that, after this preprocessing step, some strings might become equal (see Figure 3.1); however, there will be at most one string in L' that equals t'.

3.3.3 Main idea

Let us will consider each prefix string $t_{1...i}$ of t in increasing length and compute the set S_i of indices of input strings that have $t_{1...i}$ as a prefix. Formally:

$$j \in S_i \Leftrightarrow t_{1...i} = L_{j,1...i}$$

It can be easily seen that, because the input words are sorted, the set of indices S_i is, in fact, a continuous range $[a_i, b_i]$ of indices (or the empty set). Moreover, we have that $[a_i, b_i] \subseteq [a_j, b_j]$, for all $i > j$.

The idea is to use the **two pointers technique** to effectively compute $[a_i, b_i], \forall i = \overline{1, \|t\|}$. If at any point during this process we find two consecutive strings for which we can

conclude that $L_o < t < L_{o+1}$, we immediately output o. Because all of the strings at any time are equal to t up to some prefix, this can be done in time $O(1)$.

If the algorithm doesn't halt for $\|t\|$ steps, then $a_{\|t\|} = b_{\|t\|} = o$ and $L_o = t$, so we terminate by outputting $o - 1$.

3.3.4 Algorithm

The remarkably simple algorithm that solves the SIP problem in the reduced case is described below in pseudo-code (Algorithm 12).

Algorithm 12 Solving the SIP problem (reduced version)

function SolveSIP(L, t)
  $a \leftarrow 1$; $b \leftarrow \|L\|$ ▷ Initialize with $a_0 = 1$, $b_0 = \|L\|$
  for $i = \overline{1, \|t\|}$ **do**
    while $a \leqslant b$ **and** $L_{a,i} < t_i$ **do**
      $a \leftarrow a + 1$ ▷ $a_i > a$, therefore increment a
    end while
    if $a > b$ **or** $L_{a,i} > t_i$ **then**
      return $a - 1$ ▷ Found solution $L_{a-1} < t < L_a$
    end if
    while $a \leqslant b$ **and** $L_{b,i} > t_i$ **do**
      $b \leftarrow b - 1$ ▷ $b_i < b$, therefore decrement b
    end while
    if $a > b$ **or** $L_{b,i} < t_i$ **then**
      return b ▷ Found solution $L_b < t < L_{b+1}$
    end if
  end for
  return $a - 1$ ▷ $a = b$ at this point, just return one of them
end function

3.3.5 Time complexity

The algorithm is iterative. The number of iterations inside the **for** loop is at most $\|t\|$.

Each iteration inside the **while** loops effectively decreases the difference between b and a with exactly 1. Since the initial difference between b and a is $\|L\| - 1$, and the algorithm halts when $a > b$, it means that the instructions inside the **while** loops can only be executed a maximum of $\|L\|$ times.

Therefore, the total complexity of the algorithm is $O(\|L\| + \|t\|)$. This concludes the proof for Lemma 3.3.1.

3.4 Implementing basic set operations

At this point, we are ready to provide an implementation for the basic set operations of **insertion**, **deletion**, and **finding**.

3.4.1 Improving the Apply algorithm

Consider the algorithm for Apply, described in Chapter 2 (Algorithm 6). The problem with this algorithm is that **it performs a comparison on each level** of the trie. This will be problematic in the case of long strings, as comparing two strings is computationally more expensive than comparing two integers. In other words, for a key k, the Apply algorithm runs in time $O(\|k\|^2)$, if keys are to be compared in $O(\|k\|)$.

However, the **key observation** is that due to the fact that the lexicographically smallest string will always lie higher in the tree (the **min invariant**), the Apply function does nothing more than solving the SIP problem for a key k on the sorted list of strings given by KeyPath(k) (Definition 2.3.1).

Moreover, it is easy to see that KeyPath(k) has size at most $\|k\|$. Therefore, just by adapting the linear-time solution to the SIP problem to our particular instance, we can implement a modified version of the Apply algorithm that runs in time $O(\|k\|)$ for a key k.

Algorithm

Algorithm 13 describes the improved Apply function, with complexity $O(k)$, in pseudo-code. Note that because we chose the root node of the `PreTrie` to be a sentinel, the result of the SIP problem will never be to the left of the first element.

Remark. *One might also consider implementing Algorithm 13 with no additional stack memory, if one decides to implement parent links, in addition to child links, in their implementation of the* `PreTrie`*. The strategy is, however, very similar and it will not be discussed in this paper.*

Algorithm 13 Improved version of the APPLY algorithm

function APPLY($T, k, f_{FOUND}, f_{NOT\_FOUND}$)
  $L \leftarrow$ KEYPATH(T, k) ▷ returns the list of nodes of k's path in T
  $a \leftarrow 1$; $b \leftarrow \|L\|$
  for $i = \overline{1, \|k\|}$ **do**
    while $a \leqslant b$ **and** $L_a.key_i < k_i$ **do**
      $a \leftarrow a + 1$
    end while
    if $a > b$ **or** $L_a.key_i > k_i$ **then**
      Replace link $L_{a-1} \rightarrow L_a$ with $L_{a-1} \rightarrow f_{NOT\_FOUND}(L_a, k, a)$
      return
    end if
    while $a \leqslant b$ **and** $L_b.key_i > k_i$ **do**
      $b \leftarrow b - 1$
    end while
    if $a > b$ **or** $L_b.key_i < k_i$ **then**
      Replace link $L_b \rightarrow L_{b+1}$ with $L_b \rightarrow f_{NOT\_FOUND}(L_{b+1}, k, b + 1)$
      return
    end if
  end for
  Replace link $T_a \rightarrow T_{a+1}$ with $T_a \rightarrow f_{FOUND}(T_{a+1}, k, a + 1)$
  return
end function

3.4.2 Pushing and popping

Fortunately, the implementations of the PUSH and POP algorithm described in Algorithms 2, 3 do not make any key comparisons in their original implementations, therefore they do not suffer from longer word sizes, therefore they can easily be adapted in the more general case.

3.5 Important limitation

Let us consider our set of strings S and the list of operations L described below.

$$S = \{\text{aa, aaa, aaaa, aaaaa, aaaaaa, ...}\}$$

$$L = [\textsc{Insert}(\text{a}), \textsc{Erase}(\text{a}), \textsc{Insert}(\text{a}), \textsc{Erase}(\text{a}), ...]$$

If the size of set S is n and the size of list L is m, then simulating all the operations inside list L would take $O(nm)$ time. That is because insering string a into T requires pushing all the other strings of T one level lower, and this can take up to $O(max_{s \in S} \|s\|) = O(n)$. However, in a regular trie, the guaranteed complexity of an insertion/erase of any string s is $O(\|s\|)$.

3.5.1 Fortunate cases

The worse complexity of $O(max_{s \in S} \|s\|)$ has no implications when all words have equal or similar sizes. One example of such case is using a `PreTrie` to operate on any form of hashes. However, if words can vary arbitrarily in length, the complexity guarantee is broken.

Another fortunate case is when few to no erase operations are being made. In this case, if there are no erase operations, there is a guaranteed $O(\|s\|)$ **amortized complexity** per operation regarding a key s.

This follows immediately from the fact that there will be no **pop** operations, and every recursive call of a **push** operation will increase the level of one of the keys in the tree by exactly 1. Because the level of keys cannot ever decrease, and the maximum level of a certain key is equal to the size of the key, it means that after any number of operations, **the total amount of times a push operation is called can never exceed** $\sum_{s \in S} \|s\|$.

3.5.2 Solving the drawback

The solution to solving the unfortunate case described above is simply to **never erase**. In this case, one can disregard popping completely, and instead focus on a different strategy:

lazy deletion.

However, one cannot simply modify the key of a node to a null value whenever a key gets deleted, as this would break the **min invariant**. The correct approach is to duplicate the minimum from the corresponding child to the deleted node, and update a flag inside the node that states that it is, in fact, an empty node instead of a regular node. Note that several nodes might be changed during this duplication, as a whole path of empty nodes has to be updated whenever the minimum value in that path gets deleted.

3.6 Handling bigger alphabet sizes

Up until now, we have focused our work on handling binary strings. However, the approach can be extended to handle strings of an alphabet Σ of any size (i.e., $\|\Sigma\| > 2$).

We will describe two main approaches in order to achieve this.

3.6.1 First approach

One could convert the string from an arbitrary-sized alphabet to a binary alphabet, by assigning a binary sequence to each letter (e.g., using the ASCII representation of characters). This would map each character $\sigma \in \Sigma$ to a binary sequence $s_1 s_2 ... s_\psi \in \{0,1\}^\psi$.

It can be shown that such a mapping always exists for $\psi = \lceil log(\|\Sigma\|) \rceil$.

An immediate consequence is that each converted string would have its size multiplied by a factor of $\psi = \lceil log(\|\Sigma\|) \rceil$. This means that every result up until this point will hold for a bigger alphabet, but the time complexity would be multiplied by a factor of ψ.

One may find this consequence alarming; however, most implementations of regular tries have the same complexity bound, as they are using a balanced binary search tree to store the child links in case of a bigger alphabet, which has the same worst-case factor.

Remark. *Obviously, explicitly converting strings into a binary representation would require allocating extra memory for the binary representations themselves. However, most programming languages allow bit manipulation of characters inside the strings (viewed as character arrays), so an implicit conversion can be made with no extra memory.*

3.6.2 Second approach

Instead of converting the alphabet to binary, one might choose to extend the notion of the `PreTrie` from a binary trie to a $\|\Sigma\|$-ary trie.

This would probably be the preferred option for most use cases, as it would provide better average-case time bounds, and also the freedom of choosing the underlying data structure for the child links. One may achieve the following **overhead factors**[1]:

- $O(\|\Sigma\|)$ memory, $O(1)$ time (random-access arrays)
- $O(1)$ memory, $O(\|\Sigma\|)$ time (linked lists)
- $O(1)$ memory, $O(log(\|\Sigma\|))$ time (balanced binary search trees)
- $O(1)$ memory, $O(1)$ average time (hash tables)

Remark. *Even though the time complexities would suggest that using hash tables would be the preferred option to store child links, note that some algorithms require finding the minimum child link, and that might require* $O(\|\Sigma\|)$ *worst-case time or* $O(log(\|\Sigma\|))$ *(using an additional min-heap). Another disadvantage of hash tables is that, contrary to the average time bound, they tend to have a big overhead on small alphabet sizes (in this case, the first strategy might be recommended).*

[1] By overhead factors, we mean that the complexity if any operation will have to be multiplied by the described complexity of the factor

Chapter 4

Evaluation & Results

In this chapter we will discuss the actual implementation in C++ of both the `PreTree` and `PreTrie` data structures, and show how different versions perform against the STL `std::set` container. The implementation, as well as the testing and benchmark utilities, are available and open-source on GitHub [1].

4.1 Description

The code provided on GitHub contains an implementation of different versions of the `PreTree` and `PreTrie` data structures. There is a recursive implementation of the `PreTree` data structure, as well as an iterative one. There are also different utilitary programs that handle **testing** and **benchmarking**.

4.1.1 Data structure implementations

We have provided implementations for a number of different implementations of the `PreTree` and `PreTrie` data structures, as well as other similar data structures, in order to compare their performance. Please refer to the GitHub repository in order to find the actual implementations.

Generic data structures

`StdSet` is a wrapper for the STL `std::set` data structure. It is used in order to compare our data structure to the current set container inside C++.

[1] The GitHub repository can be accessed via the following link: https://github.com/bicsi/PreTree

`StdUnorderedSet` is a wrapper for the STL `std::unordered_set` data structure. It is used in order to compare our data structure to the current unordered set container inside C++. Note that this is just for reference purposes, as `std::unordered_set` is **not** an ordered data structure (as the name would suggest), so it does not fit into the same category as the proposed data structure.

`Treap` is a generic recursive implementation of the split and merge **treap** data structure (without rotations). A treap is a binary search tree with an expected height of $O(log(n))$ for n keys. The keys do not have to be random.

Data structures on numeric data

`PreTree Rec` is the recursive implementation of the `PreTree` data structure, exactly as described throughout this paper.

`PreTree Iter` is the iterative implementation of the `PreTree` data structure. The operations are functionally identical to the `PreTree Rec` version, although the implementation requires extra care for details, and we decided not to describe it in the paper.

`PreTreeR Rec` uses a different strategy of keeping any value inside the root, instead of preserving the **min invariant** (Section 2.1). The operations of **find, insert, erase** provided to be easier to implement and faster; however, we found no trivial implementation of **predecessor, successor**, as of the current time, therefore this version does not provide a full ordered set implementation, so it does not fit into the same category as `PreTree`.

Data structures on string data

`PreTrie Iter` is the iterative implementation of the `PreTrie` data structure. It operates on an alphabet of size 26 (lowercase English alphabet), but can be configured to work with an arbitrary-sized alphabet. All the operations described in this paper were adapted to work with n-ary rooted trees instead of binary ones, for the implementation of the `PreTrie`.

`BasicTrie` is an iterative implementation of the trie data structure, which was argued before to have a bigger memory overhead.

Remark. *Note that there is no recursive implementation for the* `PreTrie` *data structure. The reason is that we have found the recursive version to be tougher to implement than the iterative one, due to the extra SIP (Section 3.3) step, and we expected a recursive version to perform generally worse than the iterative version; therefore, we decided not to include it in the benchmark altogether).*

4.1.2 Operation set generator

An operation set generator is needed for both testing and benchmarking. In order to have a large freedom of parameterizing an operation set, we need a generic random operation set generator, that can handle different configurations. Our generator can be adapted to generate different subsets of operations (insert, erase, find, predecessor, successor), a variable number of different values, and a variable number of operations. This would allow us to generate both a large number of operations on a small set of values (for testing purposes), and a large number of operations on a large set of values (for benchmarking purposes). It would also allow us to benchmark and test different parts of the implementation (i.e., a certain type of operation).

4.1.3 Testing

For testing the data structures, we preferred the strategy of **fuzzy testing**[2], comparing outputs to an adapted version of the STL `set` data structure provided by the C++11 standard, which is argued to be correct in this case. Fuzzy testing is a far better alternative to other strategies like unit testing in the case of data structures, as this method (if used correctly) will capture a lot more special cases that one may miss during the creation of unit tests, and in most cases, one can easily provide a sub-optimal but straightforward data structure to test against.

The implementations of the data structures have been fuzzy tested on batches of 100 tests, of $1000-10000$ random operations (insertions, erases, queries) on $100-1000$ different randomly-generated values. We had to generate the set of values that we would operate on during the test beforehand, because otherwise there would be a very slim chance of

[2]Contrary to other types of software testing, fuzzy testing involves testing the implementation against random sets of operations. This process involves creating a random operation generator, as well as two different implementations: one being the one to be tested, the other being a presumably correct (but most of the times sub-optimal) implementation. The outputs are then compared, and if equal, the test is passed.

insertion operations to fail, and erase operations to succeed, as this would essentially mean generating the same key twice, an event that has a very slim probability of occurrence.

4.1.4 Benchmarking

Benchmarking is one of the most important parts of the project. It involves measuring the performance of the proposed data structure, compared to other data structures, and analyzing the efficiency of different operations, hoping to find bottlenecks to optimize. We have implemented a generic benchmarking tool using C++ **template metaprogramming**, and made sure that every data structure would expose the same interface for handling the set operations. In this sense, we have built wrappers for the STL `set` and `unordered_set` data structures, that would allow answering predecessor and successor queries, using the provided `lower_bound` and `upper_bound` methods.

Benchmarking configuration

We have benchmarked the implementations under three different dataset configurations: **random operations of random values**, **insertions of random values in sorted order**, and **insertions of random values in reverse-sorted order**. These three dataset configurations will be used to benchmark both the integer and the string data structures.

In addition to that, we have an extra dataset configuration for testing integer data structures: **insertions of a permutation of numbers**. The motivation behind the last dataset is to test the worst-case scenario for the `PreTree` data structure (in this scenario, the tree would be least balanced).

The results have been obtained on an early 2015 model of a MacBook Pro laptop, having a 2,7 GHz Intel Core i5 processor, and 8 GB 1867 MHz DDR3 RAM. The tests have been run under the same configuration inside the Terminal, using the `-O2` optimization flag of the `clang` compiler [3].

4.2 Results

In the following section, we will provide graphs that will illustrate the performance of the proposed data structure in relation to the other similar data structures tested against.

For a brief description of the structures present in the graph legend, please refer to Subsection 4.1.1 of this chapter.

[3] The full compilation command was: `g++ -std=c++11 -Wall -Wextra -O2`, compiled with `clang-902.0.39.1` under Mac OS X High Sierra Version 10.13.4.

4.2.1 Integer data, random operations

The values are random 30-bit integers. The set of values used during the benchmark is approximately 10% of the number of operations (represented on the horizontal axis). The time in seconds is represented on the vertical axis. The results are illustrated in Figure 4.1.

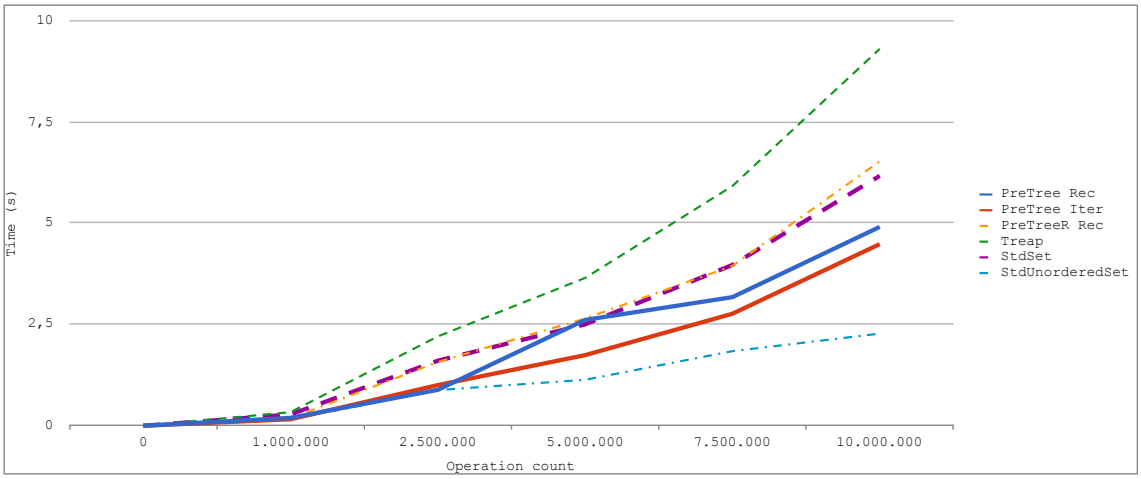


Figure 4.1: Benchmark results for integer data, random operations

4.2.2 Integer data, sorted insertions

The set of values consists of random 30-bit integers, and equals the number of operations. All the operations are insertions of values. The values are being inserted in increasing order. The results are illustrated in Figure 4.2.

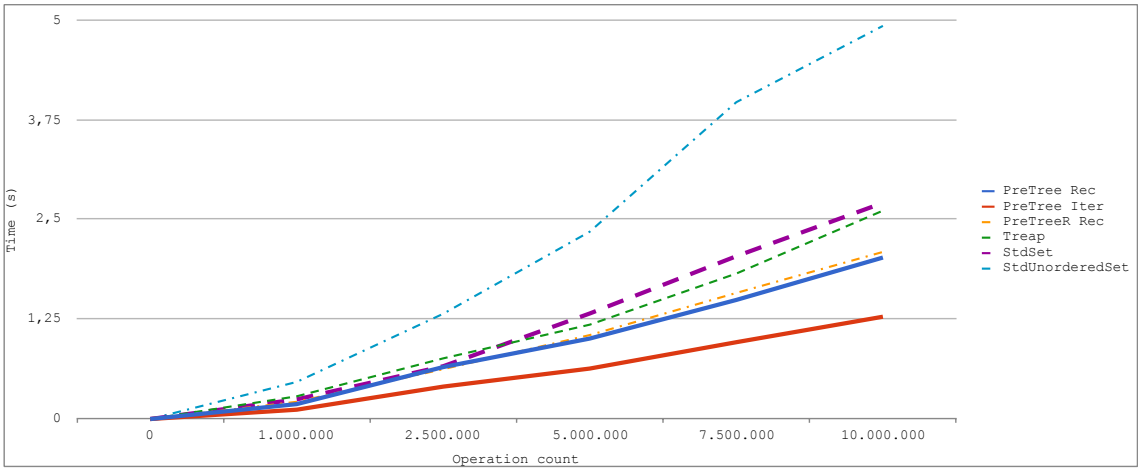


Figure 4.2: Benchmark results for integer data, sorted insertions

4.2.3 Integer data, reverse-sorted insertions

The set of values consists of random 30-bit integers, and equals the number of operations. All the operations are insertions of values. The values are being inserted in decreasing order. The results are illustrated in Figure 4.3.

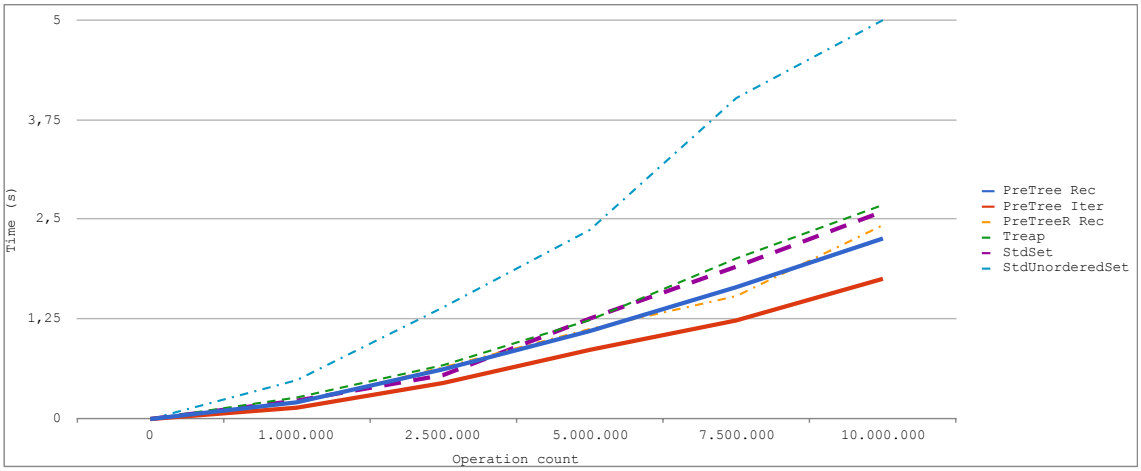


Figure 4.3: Benchmark results for integer data, reverse-sorted insertions

4.2.4 Integer data, random permutation

The set of values is a random permutation of $\{1, 2, ..., n\}$, where n is the total number of operations. All the operations are insertions of values. The values are being inserted in the order given by the permutation. The results are illustrated in Figure 4.4.

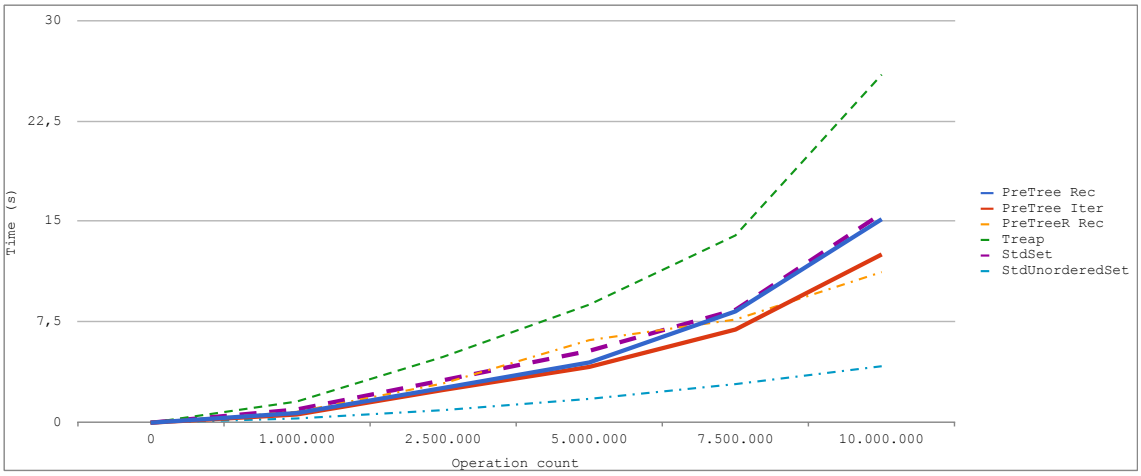


Figure 4.4: Benchmark results for integer data, random permutation

4.2.5 String data, random operations

The set of values used during the benchmark is approximately 10% of the number of operations. The values are random strings composed of 128 characters consisting of lowercase letters of the English alphabet. The results are illustrated in Figure 4.5.

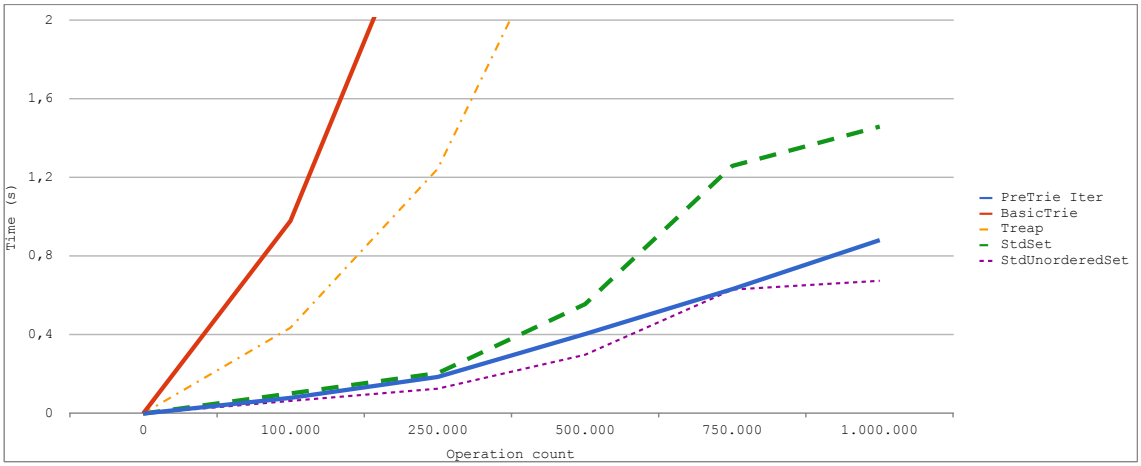


Figure 4.5: Benchmark results for string data, random operations

4.2.6 String data, sorted insertions

The values are random strings composed of 128 characters consisting of lowercase letters of the English alphabet, and its size is equal to the number of operations. All the operations are insertions of values. The values are being inserted in increasing order. The results are illustrated in Figure 4.6.

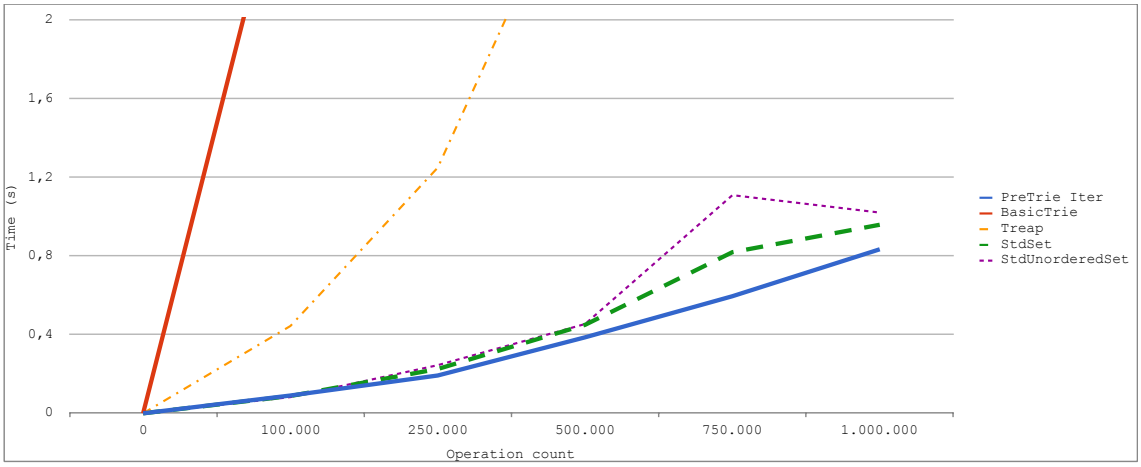


Figure 4.6: Benchmark results for string data, sorted insertions

4.2.7 String data, reverse-sorted insertions

The values are random strings composed of 128 characters consisting of lowercase letters of the English alphabet, and its size is equal to the number of operations. All the operations are insertions of values. The values are being inserted in decreasing order. The results are illustrated in Figure 4.7.

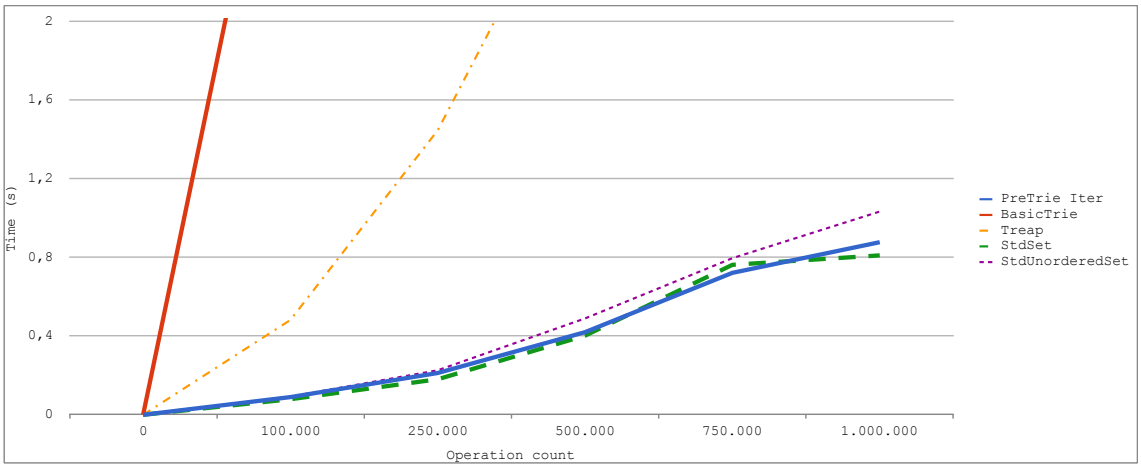


Figure 4.7: Benchmark results for string data, reverse-sorted insertions

4.3 Interpretation

The results show an impressive performance of the `PreTree` data structure compared to the other data structures included. As expected, in all of the benchmark cases, the iterative version performs slightly better than the recursive version. Even though `std::unordered_set` performs better than all the other data structures in some cases, it is not an ordered container, so it should only be taken as a reference point, and not as means of comparison.

In the first test cases, where the values are randomly generated, the `PreTree` data structure performs better, because the height of the tree is $O(log(n))$ for M-bit numbers. For a random permutation, however, the height increases to $O(M)$ for big enough n[4].

For the string datasets, we can see that the `PreTrie` data structure still manages to surpass `std::set` by a considerable margin. The margin increases as the data gets bigger[5].

The reason why a regular trie performs so bad for the sorted inserions or reverse-sorted

[4]It can be proven that, after the insertion of the first M values, the height of the tree will be equal to M.

[5]For 50 million operations acting on 5 million random 128-character strings, the execution times are: `PreTrie`: 17.276s; `StdSet`: 33.1935s; `StdUnorderedSet`: 13.065s. A regular trie or a treap times out or exceeds memory limit at this point.

insertions is that the memory requirement is very big: for $500,000$ strings, each having a length of 128, the memory required by a regular trie is approximately 1GB. At this point, memory allocation and de-allocation becomes a bottleneck for the data structure's performance.

4.4 Potential sources of errors

As with any experiment and benchmark, there are several sources of errors that can have an impact on the results. By making the reader aware of the sources of errors, we hope that we will rise the interest of analyzing them and finding solutions, which will, in turn, determine a better, more accurate insight.

The biggest source of errors is related to the **implementation** itself. Every data structure can be implemented in a variety of ways, leading to a big variety in performance. A sub-optimal implementation can lead to poor performance, while a super-optimized implementation (taking into account cache-friendliness, compiler optimization, minimizing copying, and so on) can give a significant boost to the data structure's performance. In our results, we have not used such optimizations for the `PreTree` data structure; however, we expect the implementation of `std::set` to have these aspects taken into consideration, in order to provide a container that is fast and reliable.

Another significant source of errors could arise from the fact that, inside the benchmark utility, the results are computed for each of the data structures **under the same process**, which might lead to big differences especially in cases like memory leaks, which might hinder performance. However, *our results are being stood by our confidence that these programming errors have not occurred* inside the implementation of the data structures.

Bugs in implementation could represent yet another source of errors. In this sense, we have built a testing utility to verify the correctness of the functions, but errors can still arise in case of bugs that break the guaranteed time complexity bound of our data structure.

Chapter 5

Conclusion

During this thesis we have described a data structure for handling dynamic set operation on both integers and strings, which is very similar to a trie, but has a very small memory overhead. The data structure proved to be very efficient in terms of both memory and time, proving that the idea is worth to be taken into account in a practical case.

5.1 Speed and performance

For a set S of n M-bit integers, the data structure provides a guaranteed $O(M)$ worst-case bound for each operation; moreover, if the set of numbers we operate with is random, then a better $O(log(n))$ average complexity is achieved. For a set S of binary strings of size M, the same $O(M)$ worst-case bound is achieved. Again, if the set of strings is random, better time bounds can be proven.

The proposed data structure manages to beat the standard `std::set` container provided in C++ Standard Template Language in both the integer case and the string case, by a significant factor.

The data structure is very memory-efficient, requiring only a constant number of pointers for each element inserted, regardless of its size. We can also argue that there is a time vs. memory trade-off: the more child pointers each tree node has, the faster the operations will perform; however, at the same time, more child pointers would mean a bigger memory overhead. Fortunately, one can freely choose the number of children in their implementation, in order to achieve their needs.

5.2 Ease of implementation

Ease of implementation is a factor that is overlooked most of the times, when designing a data structure; however, it is an important factor. One advantage of an easy to implement data structure is that of being easier to understand. The more complex a data structure becomes, the easier it is for a programmer to introduce mistakes (bugs) in the implementation. At the same time, if a data structure is simple enough in its original form, one can more easily improve it with further works of research.

We argue that the `PreTree` is one of the easiest to implement and to understand trie-like data structures that have constant extra memory per element and can support dynamic set operations (insertion, deletion). The next easiest alternative would be compressed tries, which, despite being easy to understand conceptually, are significantly harder to implement.

5.3 Final words

We are confident that the `PreTree` and `PreTrie` data structures, due to their balance between being time and memory efficient and easy to understand and implement, could become a viable option for designing data structures that should support dynamic set operations, like implementing caches or databases.

However, further research is definitely needed in order to check whether the implementations proposed in this paper can be improved, as well as to see what extra functionality one can implement using this model.

Bibliography

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962. 1

[2] William C. Lynch. More combinatorial properties of certain trees. *The Computer Journal*, 7(4):299–302, 1965. 1

[3] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983. 1

[4] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 75–84, Oct 1975. 1