On the Implementation of Purely Functional Data Structures for the Linearisation case of Dynamic Trees

By:
Juan Carlos Sáenz-Carrasco

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

October 2019
Abstract

Dynamic trees, originally described by Sleator and Tarjan, have been studied in detail for non persistent structures providing $O(\log n)$ time for update and lookup operations as shown in theory and practice by Werneck.

However, there are two gaps in current theory. First, how the most common dynamic tree operations (link and cut) are computed over a purely functional data structure has not been studied in detail. Second, even in the imperative case, when checking whether two vertices $u$ and $v$ are connected (i.e. in the same component), it is taken for granted that the corresponding location indices (i.e. pointers, which are not allowed in purely functional programming) are known a priori and do not need to be computed, yet this is rarely the case in practice.

In this thesis we address these omissions by formally introducing two new data structures, FULL and TOP, which we use to represent trees in a functionally efficient manner. Based on a primitive version of finger trees – the de facto sequence data structure for the purely lazy-evaluation programming language Haskell – they are augmented with collection (i.e. set-based) data structures in order to manage efficiently $k$-ary trees for the so-called linearisation case of the dynamic trees problem. Different implementations are discussed, and their performance is measured.

Our results suggest that relative timings for our proposed structures perform sublinear time per operation once the forest is generated. Furthermore, FULL and TOP implementations show simplicity and preserve purity under a common interface.
Dedication

To my wife Tonita, for the unconditional support, fed with patience and love (and a teaspoon of sugar).

To my kids Sarah and Juan Pablo, for understanding my absence from home.

And special thanks to Mum and Dad who unfortunately passed away during my studies.
Acknowledgements

The present work could not be achievable without the support and encouragement from Mike Stannett. Equally important, I appreciate the advise from Georg Struth to pursue the goals of the high demands in the research community.

I definitely enjoyed the PhD project thanks to my colleagues at the Verification and Validation Lab and for sharing the lunch time with my colleagues at the Algorithms Group.

My studies would not have been possible without the funding from the Consejo Nacional de Ciencia y Tecnología, CONACYT (Mexican National Council for Science and Technology) under grant 411550, scholar 580617, CVU 214885.
Contents

1 Introduction ............................................. 13
  1.1 Problem Statement ................................. 14
  1.2 Motivation ........................................ 18
    1.2.1 Applications where dynamic trees operations take place ........................... 18
    1.2.2 Dynamic trees in Functional Programming ............................................. 19
  1.3 Benefits from Functional Programming ............. 19
  1.4 Source Language ..................................... 21
    1.4.1 Why Haskell? .................................... 21
  1.5 Terminology ......................................... 24
  1.6 Contributions ....................................... 24
  1.7 Structure of this Thesis ............................ 25

2 Related Work ............................................ 27
  2.1 Path Decomposition ................................. 28
    2.1.1 Purely Functional Implementation ....................... 29
  2.2 Tree Contraction .................................... 30
  2.3 Linearisation ....................................... 30
  2.4 Chapter notes ...................................... 33

3 Fundamentals .......................................... 34
  3.1 Forest and trees nomenclature ...................... 34
3.2 Input tree data structure ........................................ 36
3.3 Data.Set ........................................................... 37
3.4 2-3 trees ............................................................ 39
3.5 Monoids .............................................................. 41
    3.5.1 Monoidal annotation ............................. 42
3.6 Finger Trees ....................................................... 42
    3.6.1 Structure of FT ................................. 43
    3.6.2 Amounts of data stored in the FT data struc-
         ture .............................................. 46
    3.6.3 Operations in FT ................................. 50
    3.6.4 Accessing the endpoints of a FT ........... 53
    3.6.5 Inserting at the endpoints of a FT .......... 54
    3.6.6 Appending FTs ................................. 57
    3.6.7 Searching and splitting in FT .......... 59

4 Euler-Tour Trees Functionally, FUNETT ........................ 67
    4.1 Euler-tour trees by Henzinger and King .......... 67
        4.1.1 Representation of the input tree .......... 68
        4.1.2 Operations on ETT-HK .................... 68
    4.2 Euler-tour trees by Tarjan .......................... 70
        4.2.1 Representation of the input tree .......... 70
        4.2.2 Operations on ETT-T ....................... 70
    4.3 FUNETT ...................................................... 71
        4.3.1 Representation of the input tree .......... 72
        4.3.2 FUNETT data structure ...................... 72
        4.3.3 Operations on FUNETT ..................... 74
    4.4 Chapter Notes .............................................. 84

5 Indexless data structures ............................................ 85
    5.1 FULL dynamic trees ................................. 85
        5.1.1 FULL dynamic trees data types ........... 86
5.1.2 Full dynamic trees operations . . . . . . . . . . 88
5.1.3 Experimental analysis of Full dynamic trees 95
5.2 Top dynamic trees . . . . . . . . . . . . . . . . . . 126
  5.2.1 Top dynamic trees data types . . . . . . . . 127
  5.2.2 Top dynamic trees operations . . . . . . . . 129
  5.2.3 Top vs Full experimental results . . . . . 133

6 Conclusion 142
  6.1 Further Directions . . . . . . . . . . . . . . . . . . 143
List of Figures

1.1 Example of a forest .................................. 15
1.2 Example of pre-link ................................. 16
1.3 Example of link applied ............................ 16
1.4 Example of pre-cut ................................ 16
1.5 Example of cut applied ............................. 16

2.1 Input tree .............................................. 31
2.2 Input tree, turning edges into directed-edges ..... 31
2.3 ETT is built from input tree ....................... 31
2.4 ETT represented by a finger tree .................. 32

3.1 unit forest ............................................. 34
3.2 2-node forest ......................................... 35
3.3 10-node forest ....................................... 35
3.4 Input tree .............................................. 37
3.5 Balanced Search Tree or BST ...................... 39
3.6 Leafy 2-3 tree ....................................... 39
3.7 Nodal 2-3 tree ....................................... 40
3.8 Complete 2-3 tree .................................. 40
3.9 Finger tree definition, FT ......................... 43
3.10 Node type ............................................. 44
3.11 Digit (or affix) type ............................... 45
3.12 FT example .......................................... 46
3.13 Number of leaves, Empty bottom ................ 47
3.14 Number of leaves, Single bottom .......................... 48
3.15 Number of monoidal annotations, Empty bottom .... 49
3.16 Number of monoidal annotations, Single bottom .... 50
3.17 search an element ........................................... 63
3.18 split operation ............................................... 64
3.19 Ordered-set, initial example ................................. 66
3.20 Ordered-set, initial annotations ............................ 66
3.21 Ordered-set, final example ................................. 66

4.1 An input tree ................................................. 67
4.2 FUNETT, input tree .......................................... 73
4.3 FUNETT, initial example .................................... 74
4.4 Input tree example for cutTree ............................. 78
4.5 Trees result from cutTree ................................... 80
4.6 Tree result from linkTree ................................... 84

5.1 FUNETT, repeated edges .................................... 87
5.2 FUNETT, reduced sets ....................................... 87
5.3 Input forest example ......................................... 90
5.4 FULL link example .......................................... 92
5.5 FULL cut example ........................................... 94
5.6 Sample 1 of plotting ........................................ 96
5.7 Sample 2 of plotting ........................................ 96
5.8 Sample 3 of plotting ........................................ 97
5.9 Plotting unit FULL forest, means ......................... 98
5.10 Plotting unit FULL forest, medians ....................... 99
5.11 Plotting 2-node FULL forest, means .................... 100
5.12 Plotting 2-node FULL forest, medians .................. 101
5.13 Plotting 10-node FULL forest, means ................... 103
5.14 Plotting 10-node FULL forest, medians ................ 103
5.15 Plotting 300-node FULL forest, means ................. 104
5.29 Performance individual **link-cut** incrementing number of operations, *10-node* forest .............. 124
5.30 Performance individual **link-cut** incrementing number of operations, *300-node* forest .............. 125
5.31 Accumulators of a finger tree .................. 127
5.32 Top accumulator of a Top finger tree .......... 128
5.33 General view of a Top finger tree ............... 128
5.34 FullvsTop, forests construction, *300-node* .... 134
5.35 FullvsTop, forests construction, *unit* vs *2-node* vs *10-node* ........................................ 135
5.36 FullvsTop, connectivity *10-node* and *300-node* forests136
5.37 FullvsTop, **link** in *unit* and *2-node* forests .. 137
5.38 FullvsTop, **link** in *10-node* forests ............. 138
5.39 FullvsTop, **link** in *300-node* forests .......... 138
5.40 FullvsTop, **cut** in forests .................... 139
5.41 FullvsTop, individual **link-cut** in forests where input number of nodes ......................... 140
5.42 FullvsTop, individual **link-cut** in forests where input is number of operations .................. 141
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td><strong>Data.Set</strong> operations</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Finger tree operations</td>
<td>52</td>
</tr>
<tr>
<td>3.3</td>
<td>Accumulation of &lt;&gt; example</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary FT operations</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Summary of FT operations applied to FUNETT when set-union is the monoidal annotation</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>Bounds of FT operations applied to FUNETT for viewl, viewr, &lt; and &gt; cases</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>Bounds of FT operations applied to FUNETT for split and search cases</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>Bounds of cutTree operation</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Bounds of linkTree operation</td>
<td>81</td>
</tr>
<tr>
<td>4.7</td>
<td>Operations in ETT specifications</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>FULL dynamic trees operations</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>Tabular performance of <em>unit</em> FULL forest construction</td>
<td>99</td>
</tr>
<tr>
<td>5.3</td>
<td>Tabular performance of 2-node FULL forest construction</td>
<td>102</td>
</tr>
<tr>
<td>5.4</td>
<td>Tabular performance of 10-node and 300-node FULL forests construction</td>
<td>105</td>
</tr>
<tr>
<td>5.5</td>
<td>Tabular performance of all FULL forests construction</td>
<td>107</td>
</tr>
<tr>
<td>5.6</td>
<td>Amount of monoidal annotations in a FULL dynamic tree</td>
<td>109</td>
</tr>
</tbody>
</table>
5.7 Amount of monoidal annotations in a FULL dynamic tree via its affixes for a FULL forest. . . . . . 111
5.8 Tabular performance of all FULL forests connectivity 114
5.9 Tabular performance of FULL link . . . . . . . 117
5.10 Tabular performance of FULL cut . . . . . . . . 120
5.11 Tabular performance of all FULL linkcut . . . 122
5.12 Tabular performance of all FULL linkcut per operation . . . . . . . . . . . . . . . . . . . . . . . 123
5.13 Tabular values of performing link-cut . . . . 126
5.14 TOP dynamic trees operations . . . . . . . . . 133
Chapter 1

Introduction

The topic of this thesis is the purely functional programming approach to the handling of dynamic trees problem. Although dynamic trees problem attracted quite a lot of research in the last decades, since Sleator and Tarjan [1], there have been no insights towards the implementational side for the functional programming setting.

Okasaki [2, 3] pioneered research on efficient purely functional data structures just over two decades ago but his studies have not included the case of dynamic trees management, specifically for the linearisation case. Moreover, cases for lookups when an index is not provided have not been studied at all.

It is our main contribution that we implemented FULL and TOP, data structures to manage dynamic trees operations, thereby showing that this is feasible for the functional programming paradigm. We conducted an experimental study comparing our implementations in Haskell [4].

Before turning to the practical and implementational sides of the problem, however, we present the different approaches to it from the theoretical point of view. We show that the procedures provided by Henzinger and King [5, 6] and Tarjan [7] regarding
Euler-tour trees (ETTs) can be implemented declaratively, and we contribute an improvement for the basic structural cases, that is, the link and cut operations.

Both of our implementations, FULL and TOP, are built on top of finger trees, a purely functional data structure devised by Ross and Hinze [8]. We manage sequence operations for values stored at the leaves while look up operations are performed in the internal tree nodes (i.e. monoidal annotations) via binary search trees (i.e. BSTs). Sequence and BST data structures have been studied extensively in both the algorithms community and the data structures community for both functional and imperative settings, but not much work has been done for the cases where both structures coexist.

Our data types are capable of managing practically any binary search tree as the look up engine, providing it supports set operations (such as membership, insertion and union) for any polymorphic type on the leaves as long as it can be ordered. We show that our techniques are effective in practice by implementing and evaluating them.

1.1 Problem Statement

A dynamic tree allows three kinds of (basic) operations:

- Insert an edge.
- Delete an edge.
- Answer a question related to the maintained forest property.

The first two types of operations are called updates and the last one is a query. In the simplest case, this is a global question like
“Are vertex \( u \) and vertex \( v \) in the same tree?” or “Is vertex \( v \) on vertex \( u \)’s path towards the root?”, and the answer is just “True” or “False”. The purpose of a dynamic tree algorithm is to maintain a forest property faster than by recomputing it from scratch every time the forest changes. The term dynamic tree problem was coined by Sleator and Tarjan in [1]. The aims, implementational issues and data structure design by Sleator and Tarjan followed the imperative programming paradigm. We focus our attention in the aforementioned operations under the approach of purely functional programming considering a forest of fixed \( n \) number of vertices and consider only undirected edges through this document.

In figs. 1.1 to 1.5 we depict a small example from the above operations of inserting an edge, for which we shall call it link, deleting an edge, for which we shall call it cut and looking for an edge (query property) for which we shall call it connected. Let us start with link. So, having a forest \( f \) (Figure 1.1) we firstly locate the vertices (i.e. connected==False) where the new edge is about to be inserted (Figure 1.2) and then apply link to \( f \). When cutting, the edge is located (i.e. connected==True, Figure 1.4) within \( f \) and then cut is performed (Figure 1.5).

An update in the forest is local. If due to the application the forest changes globally, we can model this with several updates as in performing an unbound sequence of operations over such a
Figure 1.2: Identifying the vertices, `connected==False`, in $f$ for which `link` is about to be applied.

Figure 1.3: `link` is applied over $f$.

Figure 1.4: Identifying the vertices, `connected==True`, in $f$ for which `cut` is about to be performed.

Figure 1.5: `cut` is performed on $f$. 
forest. In the worst case, we could move from one forest to a totally different one as in a random forest generation. Therefore, it does not make sense to maintain the forest property of the new forest by means of data collected from the old forest faster than by recomputing it with a static forest algorithm. This scenario is suitable for designing and analysing persistent data structures.

Data structures which allow queries and insertion of edges, but not deletion of edges are called \textit{incremental} and \textit{decremental} otherwise [9]. In any case, we can refer to any of the above structures to be \textit{semi-dynamic}. If we want to distinguish between semi-dynamic data structures and data structures allowing both operations, then the latter are called fully \textit{dynamic data structures}. We provide implementation and experimental analysis for the semi and fully dynamic cases.

Note that the term \textit{forest property} is quite general. A forest property can be a predicate on the whole forest (e.g., testing membership), or a predicate on pairs of nodes (e.g., connectivity).

The forest property we will mainly deal with in this thesis is connectivity. Two vertices \(u\) and \(v\) are connected, if both vertices are members of the same component or tree. We want to be able to quickly answer each question of the type “Are vertices \(u\) and \(v\) connected in the current forest?”.

Each time an edge \(e = (u, v)\) is to be inserted, we ask the data structure whether \(u\) and \(v\) are already connected. If this is not the case, we decrease the forest tree-counter after inserting \(e\). If \(e\) is to be deleted, we delete it and we increase the forest tree-counter. The answer to the question whether the whole forest is connected is “True”, if and only if the tree-counter equals 1.
1.2 Motivation

Inserting and deleting edges are among the most fundamental and also most commonly encountered operations in trees, especially in the dynamic setting. This encourages simplicity and efficiency at the time of the computation so any application can use them. In this section, we motivate the approach of functional programming for these angles. This work forms part of a larger objective, that of the functional programming analysis on dynamic data structures.

1.2.1 Applications where dynamic trees operations take place

Since the definition of the dynamic trees problem data structure by Sleator and Tarjan [1], two major structural operations arise: link and cut, therefore the term Link-Cut trees for this data structure. Besides applications like UNION-SPLIT-FIND problems [10], dynamic trees computations are frequently needed in a wide spectrum of applications, to name a few:

- Flows on Networks; ([11], [12]) link and cut operations are used to maintain the residual capacities of edges and that of changing labels in the network.

- Rearrangement of Labelled Trees; recently applied to the problem of comparing trees representing the evolutionary histories of cancerous tumors. Bernardini et al. ([13]) analyse two updating operations: link-and-cut and permutation. The former is due to transform the topology of the input trees whereas the latter operation updates the labels without mutating its topology.
Geomorphology; Ophelders et al. [14] model the evolution of channel networks. Linking and cutting trees are used to model the dynamic behaviour of the growth and shrinking of areas in a river bed.

1.2.2 Dynamic trees in Functional Programming

Literature has shown a lot about updating edges in trees and graphs, see for instance the handbook for data structures regarding this topic in [9], but in practical terms relatively little work has been done for the functional programming, specifically for the dynamic setting. In the case of graph structures, Erwig [15] introduces a functional representation of graphs where a graph is defined by induction. Although an interface and some applications have been provided, none of those refer to the dynamic trees problem. For the case of trees, Kmett [16] defines a functional programming version (i.e. in Haskell) of that of the one defined by Sleator and Tarjan [1]; unfortunately Kmett’s work relies completely on monads and stateful computation making it difficult to reason about the operations. Also, the element of a forest is missing in Kmett’s work.

1.3 Benefits from Functional Programming

Amongst others, we highlight the features we shall put in place in our proposals in this thesis.

Programming perspective, an excerpt from [17]

- Using functions rather than loops and assignments to express algorithms.
• An algorithm expressed as a function is composed of other, more basic functions can be studied separately and revised in other algorithms.

• Functions that build trees can be studied separately from functions that consume trees.

Reasoning about programs, an excerpt from [17]

• Functional programming is a method of program construction that emphasises functions and their application rather than commands and their execution.

• Functional programming has a simple mathematical basis that support equational reasoning about the properties of the programs.

Function application, an excerpt from [18]

• In mathematics one visually writes $f(x)$ to express the application of function $f$ to the argument $x$. In Haskell, we write $f \ x$ to express the application of function $f$ to argument $x$. However, expressing $f(x)$ in Haskell is valid but unusual.

• Function application in Haskell allows us to reduce the number of brackets in an expression allowing a clarity and readability in the code, specially when expression as large.

Function composition, an excerpt from [18]

• Alike mathematics, two functions $f : Y \to Z$ and $g : X \to Z$ can be written as $f \cdot g$ and its application to an argument $x$ as $(f \cdot g)x = f(g \ x)$. The order of composition is from right
to left as functions are written to the left of the arguments to which they are applied.

- Grouping functions on the left, provided they can be composed, allows programmers to exploit higher function programming and reduce the lines of code in the program without sacrifice readability.

1.4 Source Language

All source code will be presented in Haskell [4], implemented using the Glasgow Haskell Compiler (GHC). However, the algorithms can all easily be translated into any other functional language supporting both strict and lazy evaluation.

Throughout this thesis, we assume that the reader is familiar with the basics of Haskell and Data Structures. In case of any problem, we refer to the introductory books by Bird [18] for general syntax and semantics, and Okasaki [19] for generalities on purely functional data structures.

As a check for accuracy in the examples throughout this dissertation, all the indented, typeset code is type-checked against our implementation every time the text is typeset. The code snippets throughout this dissertation are presented as illustrated here:

```haskell
function :: Type → Type → Type  -- function type signature
function x y = x + y             -- function definition
```

1.4.1 Why Haskell?

Haskell complies with the features of functional programming we have described earlier in this chapter, in particular the following
• Function application in Haskell allows us to reduce the number of brackets in an expression allowing a clarity and readability in the code, specially when expression as large.

• Grouping functions on the left, provided they can be composed, allows programmers to exploit higher function programming and reduce the lines of code in the program without sacrifice readability.

Another interesting feature is its purity, meaning there are no side-effects. So, we shall not worry about mutating accidentally the state of a variable or the entire program. That is, maintaining code might an advantage as well as looking for errors due to changing values to variables can be easily avoided.

Presentation of our specification and implementation is eased by the analysis and proofs of programs through equational reasoning. The following example, adapted from [20], shows the use of equational reasoning in proving by structural induction that an equation is valid for properties applied on a (binary) tree data structure.

Example (equational reasoning over trees)

Problem: Given the code below, prove that
\[ \text{sum(flatten t)} = \text{treesum t} \]
holds for all finite defined trees of type \text{Tree Int}.

Solution. Firstly, we define the data types and functions \text{sum}, \text{flatten} and \text{treesum}. Comments (text after two dashes) within rectangular or regular brackets act as equation labels.

\begin{verbatim}
data Tree a = Empty | Node (Tree a) a (Tree a)
\end{verbatim}
Proof The principle of structural induction tells us that if we want to prove that a property \( P \) holds for every finite defined tree \( t \) of type \( \text{Tree } a \), it is enough to prove that

- \( P(\text{Empty}) \) holds outright;
- \( P(\text{Node } l x r) \) holds whenever \( P(l) \) and \( P(r) \) both hold.

Part 1: Prove \( P(\text{Empty}) \) holds outright

\[
\begin{align*}
\text{sum ( flatten Empty )} & = \text{sum []} \quad \text{by [flat.1]} \\
& = 0 \quad \text{by [sum.1]} \\
& = \text{treesum Empty} \quad \text{by [tsum.1]}
\end{align*}
\]

Part 2: Prove \( P(\text{Node } l x r) \) holds if \( P(l) \) and \( P(r) \) both hold. As often happens, we need to prove some auxiliary results. In this case, we need to prove that, for any \( \text{Int} \) lists \( xs, ys, zs \) we have

\[
\text{sum ( xs ++ ys ++ zs )} = \text{sum xs} + \text{sum ys} + \text{sum zs} \quad \text{-- [lemma]}
\]

Taking this for granted (the proof again involves structural induction, this time over finite defined values of type \([a]\)), we have

\[
\begin{align*}
\text{sum (flatten (Node 1 x r))} & = \text{sum (flatten 1 ++ [x] ++ flatten r)} \quad \text{by [flat.2]} \\
& = \text{sum (flatten 1) + sum [x] + sum (flatten r)} \quad \text{by [lemma]} \\
& = \text{treesum 1 + sum [x] + sum (flatten r)} \quad \text{by [P(l)]} \\
& = \text{treesum 1 + sum [x] + treesum r} \quad \text{by [P(r)]} \\
& = \text{treesum 1 + (x + sum []) + treesum r} \quad \text{by [sum.2]} \\
& = \text{treesum 1 + (x + 0) + treesum r} \quad \text{by [sum.1]}
\end{align*}
\]
In terms of presence and usage, functional programming has gained presence in the language programming community, as in [21] and [22]. In particular, Haskell currently has some level of adoption in industry [23].

1.5 Terminology

Throughout the thesis, we will consider a forest $\mathcal{F}$ of undirected $k$-degree trees with $|\mathcal{V}| = n$ vertices, and $|\mathcal{E}| = e$ edges. We write $\log x$ as an abbreviation for $\max\{1, \log_2 x\}$ throughout this thesis, so $\log x$ is never smaller than 1.

The term operation is similarly overloaded, meaning both the functions supplied by an abstract data type and functions defined originally in relation to the dynamic trees problem. We reserve the term operation for the latter meaning, and use the term function for the former.

In Chapter 3 we provide a short summary of the data structures used in this thesis whilst in Chapter 4 basic terminology for the dynamic trees problem is presented.

1.6 Contributions

This work should be viewed as an exploration into the dynamic trees problem under the functional programming approach. In this section we list the main contributions of the thesis.

- We present, for the first time, a declarative functional implementation of the procedures for Euler-tour trees. We show
feasibility is possible under this approach.

- We make explicit the management of indices location per operation and per data structure. This has been taken for granted in the literature. By doing so, we make even clearer the specification given so far for the linearisation case of dynamic trees.

- We present FULL to deal with the data structures for the main update and query operations for the dynamic trees problem, specifically the linearisation case. We demonstrate experimentally that both data types allow algorithms to run in sublinear time for all the basic operations once the forest has been created, being this implementation the first appearance in the purely functional programming setting. This work has been presented at [25]

- We introduce TOP, an improved version of FULL by reducing the number of internal operations in its finger tree data structure. Performances show that, experimentally, TOP outperforms FULL in the vast majority of the cases from 1.2 up to 3 times. This work has been presented at [24].

- We make publicly available the source code for all of our implementations as well as the statistical data.

1.7 Structure of this Thesis

Chapter 2

We describe the current approaches that deal with the data structures that deal with the dynamic trees problem. We brief the work
done within the purely functional programming for each approach and provide some reasons for our choice amongst the approaches.

Chapter 3

A brief description of the functional data structures currently in the literature that play a basic role in our proposal, of which finger trees are the core data structure.

Chapter 4

We review Euler-tour trees, specifically the procedures defined by Henzinger and King [6] and Tarjan [7], and propose a functional and declarative implementation, that is, FUNETT.

Chapter 5

We devise a variety of data types in order to manage indexless structures, such as FULL and TOP, that shall support dynamic trees operations. We describe their design and implementation as a solution of ETT to solve the common dynamic tree operations under the purely functional programming approach.

Chapter 6

We give our conclusions, and suggest some topics for future research.
Chapter 2

Related Work

The literature covering the terms *dynamic trees*, *dynamic trees problem*, *dynamic trees operations* is vast, from Overmars’ work [26] on *dynamic data structures* to the classification for *dynamic trees* given by Demetrescu et al. in [9, Chapter 35]. The following quote by Werneck [27, pp 5-6] summarises the operations we shall study in this thesis

\[\ldots\text{We limit our discussion to dynamic trees. In particular, all data structures we discuss below [Path Decomposition, Tree Contraction, Euler Tours] can solve the dynamic connectivity problem for trees: they maintain a forest subject to edge insertions and deletions and support queries asking whether two vertices belong to the same tree or not}} \ldots\]

On the implementation side, Tarjan and Werneck [28] show that maintaining a forest under a sequence of insertions and deletions of edges can be done in $O(\log n)$ time per operation. The bound is amortised when using splay trees [29] and worst case when using red-black trees [30]. With respect to functional programming analysis, several efforts have been carried out [19, 31,
Standard implementations in Haskell are accessible at [33], although our own work requires a slight variant of the standard implementation (see Chapter 5).

2.1 Path Decomposition

Any two vertices in a tree define a unique path, namely the set of edges/vertices traversed while moving through the tree from one vertex to the other. The goal of path decomposition is to split the tree into a disjoint set of paths; each vertex should belong to exactly one path in the decomposition. Performing such a decomposition is straightforward, except where a vertex has two or more children, in which case we need to choose which of the children, if any, will be allocated to the same path as its parent. Such action implies to manage two or more trees either by a single structure, i.e. extended definition for a BST or a forest data structure. A well-known data structure that manages path decomposition is the one devised by Sleator and Tarjan [1], called link-cut trees or ST trees. These trees were devised originally to manage directed trees with fixed roots and labels stored at the edges. In this setting, all path-related queries refer to paths between some vertex and the root of its tree.

As explained in our problem statement in Section 1.1, we are interested in the link, cut and connected operations. However, connected is not defined explicitly in the original work by Sleator and Tarjan [1].

- \textit{link}(vertex }v\textit{, }w\textit{, real }x\textit{): Combine the trees containing }v\textit{ and }w\textit{ by adding the edge }\langle v, w \rangle\textit{ of cost }x\textit{, making }w\textit{ the parent of }v\textit{. This operation assumes that }v\textit{ and }w\textit{ are in different trees and }v\textit{ is a tree root.
• \textit{cut}(vertex v): Divide the tree containing vertex v into two trees by deleting the edge \((v, parent(v))\); return the cost of this edge. This operation assumes that \(v\) is not a tree root.

2.1.1 Purely Functional Implementation

To the best of our knowledge, there is not a formal study for path decomposition for the purely functional programming realm. However, there is an attempt through a Haskell implementation by Kmett [16] who claims the following operations run in \(O(\log n)\) each.

\begin{verbatim}
link :: (PrimMonad m, Monoid a) 
  ⇒ LinkCut a (PrimState m) 
  → LinkCut a (PrimState m) 
  → m ()

cut :: (PrimMonad m, Monoid a) 
  ⇒ LinkCut a (PrimState m) 
  → m ()

connected :: (PrimMonad m, Monoid a) 
  ⇒ LinkCut a (PrimState m) 
  → m ()
\end{verbatim}

Practically, all the operations in Kmett’s work rely on the \textit{ST} monad, which allows mutability on data. Even though the above code is considered pure, it is hard to reason around the expressions since they emulate pointers rather than mathematical equations. Take for instance, the definition for \textit{link}

\begin{verbatim}
link v w = st $ do
  access v
  access w
  set path v w
\end{verbatim}

An attempt to apply the equational reasoning we have seen in 1.4.1 over \textit{access} and \textit{set path} is not straightforward. In [18], Bird
considers that reasoning with monadic code, as the one above, is a topic of ongoing research.

2.2 Tree Contraction

The aim of this approach is to reduce the size of the trees in terms of their edges and vertices in the parallel setting. Three data structures have been studied in this context, topology trees [34], RC trees [35] and top trees [36].

Although purely functional programming and Haskell in particular are considered a naturally suitable paradigm for programming parallelism and concurrency [37, 38, 39], no analysis or implementation regarding the tree contraction approach have been fully studied.

Nevertheless, Morihata and Matsuzaki in [40] define the analysis and some data types towards the representation of tree contraction data structures in the functional programming setting. However, their work is limited to the analysis on functions that traverse and reduce the tree structure but the main dynamic tree operations are not studied.

2.3 Linearisation

Similar to the path decomposition approach, the linearisation approach is mostly studied through one structure, the Euler-tour tree, ETT. The term linearisation comes from the shape of the original data structure (i.e. a tree which is non linear) into a sequence seen as a line. That is, for every edge \((u, v)\) in the input tree, edge \((v, u)\) is generated and both edges form an ETT. In order to build an ETT sequence, we firstly take every vertex \(v\) and
replace it for the tuple (not an edge) \((v, v)\). By selecting an arbitrary vertex as root or leftmost element, depicted in Figure 2.1, we traverse the Euler-tour in any direction as in Figure 2.2 and add the selected edge to the sequence, see Figure 2.3. As soon as a vertex is discovered it is added to the sequence only once.

Having a ETT sequence, it is then manipulated through a more efficient structure such as BST. Once again, in case of amortised complexity analysis, splay trees are used and for the worst-case complexity AVL or red-black trees are considered. The term linearisation comes from the shape it takes when the Euler-tour is
disconnected between the last link and the head of such as tour.

Notice that an input tree can be represented by potentially many ETT depending on the selection of the vertex as root and the direction of the edges (clockwise or anticlockwise). An input tree \( t \) contains \( n \) vertices and \( n - 1 \) edges; an ETT is represented by \( n \) vertices and \( 2 \times (n - 1) \) edges. The sequence is comprised of tuples of vertices and edges altogether, known as elements. The size of an ETT is \( 3n - 2 \) elements. The performance we shall measure is based on \( n \), the number of elements per sequence.

It is our purpose in this thesis to transform an input tree into an ETT and that of managing its sequence through a finger tree (Figure 2.4), a data structure described in Chapter 3, to perform efficient dynamic tree operations.

![Figure 2.4: Finger tree \( Ft \) corresponding to ETT sequence from Figure 2.3](image)

The process of transforming input trees into ETT sequences is not unique for the linearisation approach. Another way, is to define a forest of singleton trees and from this point apply link and cut operations until desired sizes have been reached for the corresponding input trees representation. We further the discussion of this technique in Chapter 5. Definitions of the above link and cut with auxiliary operations are detailed in Chapter 4.
2.4 Chapter notes

We have seen three approaches to dealing with tree data structure under the sequence of dynamic operations. Path decomposition focuses on performing computation with values over the edges when forming paths; its analysis proceeds in a bottom-up fashion. Tree contraction analyses the tree structure within the parallel setting where values are stored over the vertices. Our interests in this thesis solely focus in the last approach, the linearisation case as it offers the following features

- No values (labels) over edges or vertices are required in order to perform link, cut and connected.

- Tree representation is a sequence, which is simpler to process in comparison to a collection of paths or condensed information in a contracted tree.
Chapter 3

Fundamentals

This chapter presents an overview of the fundamental data structures we use in our work. We commence with some nomenclature for forests and trees.

3.1 Forest and trees nomenclature

In this thesis we define a forest as a collection of fixed number of trees. A singleton-tree is a tree with no edges. A forest comprised of only singleton-trees is called unit-forest, depicted in Figure 3.1. We avoid a forest having just one singleton-tree as it is practically just a node or vertex.

\[
\begin{array}{cccccc}
1 & 2 & 3 & \ldots & k & \ldots & n \\
\text{T}_1 & \text{T}_2 & \text{T}_3 & \text{T}_k & \ldots & \text{T}_n
\end{array}
\]

Figure 3.1: A unit forest

A \( k \)-tree is a tree of degree \( k \), where the degree of a tree is the maximum number of edges of any vertex in that tree. The terms node and vertex are interchangeably in this thesis. An \( n \)-node
forest is a forest which practically all or all but one of its trees having $n$ vertices. 2-node forest is depicted in Figure 3.2 whereas 10-node forest is depicted in Figure 3.3.

The size of a forest (i.e. $\text{ForestSize}$) is the sum of the number of nodes (i.e. $\text{NumNodesForest}$) plus the number of edges living in that forest. Similarly, the size of a tree (i.e. $\text{TreeSize}$) is the sum of the nodes and edges defined for that tree $\text{NumNodesTree}$.

If a forest has only one tree and $\text{ForestSize} = \text{TreeSize}$,
where $\text{NumNodesForest} \geq 2$, is called one-tree forest.

3.2 Input tree data structure

The basic (rose) tree type, $\text{Tree } a$, is defined in Haskell’s Data.Tree package [41] as

```haskell
data Tree a = Node { rootLabel :: a, -- [roseTree] subForest :: Forest a }
```

where

```haskell
type Forest a = [Tree a]
```

This definition has both advantages and disadvantages for our own work. Since the definition of $\text{Tree } a$ does not constrain type $a$ to be ordered nor offer any kind of balancing, this structure is fairly inefficient because any querying and updating of an element requires us to traverse the entire structure to identify the corresponding place in the tree for the operation. As a result, inserting, deleting or looking up for an element in this kind of tree generally takes $O(n)$ time per operation for a tree containing $n$ elements. The main goal of this thesis is to find ways to represent such trees that allow for more efficient handling - accordingly, we sometimes refer to trees of this type as input trees, i.e. trees that are provided as inputs to our representation procedures.

Notice also that this definition does not allow us to generate empty instances of trees. Instead, the simplest tree comprises a single vertex with no edges (i.e. a singleton tree). On the other hand, the Data.Tree package includes useful functions to transform lists into trees and vice versa, as well as auxiliary operations such as pretty printing.

---

1The forest structure used in our own work is somewhat different, as we explain in Chapter 5.
Example

Consider the following `Tree Int` instance, `tree`. Although the definition implicitly defines a `root`, we will often find it useful to think of trees as rootless entities. Figure 3.4 shows both a rooted and a rootless representation of `tree`.

```
tree = Node 7
    [ Node 9
        [ Node 5 []
        , Node 2 []
        , Node 4 []
        , Node 3 []
        ]
```

Figure 3.4: An input tree of six vertices represented in both rooted and rootless form: hierarchical (left) and as a star (right).

3.3 Data.Set

An ubiquitous problem when dealing with algorithms and data structures is that of searching for an element. The simple yet powerful binary search tree, BST, model provides a rich family of solutions to this problem. In this thesis we shall focus our attention to the `Data.Set` data type which is a concrete instance for an efficient BST functional implementation [42] (enhanced in [43]). Research on set-like trees is vast; see, e.g. Derryberry [44].

The definition of a `set` in our work is defined following [45]. A `Set` is a BST defined either by an empty-value `Tip` or by a node `Bin` which stores a datum `a`, together with two subtrees (a
recursive call to Set each) and the Size of the tree rooted at that node:

```haskell
data Set a = Tip
  | Bin !Size !a !(Set a) !(Set a)
```

The type Size here is a synonym of the integer type Int, hence it is limited to handle tree sizes between $-2^{63}$ and $2^{63} - 1$. Each exclamation mark (e.g., !Size) is a bang annotation, which means that the type next to it will be evaluated to weak-head normal form (by pattern matching on it) or in a strict manner (see [45]).

Following [42] and [45] we list, in Table 3.1, the runtime performance for the Data.Set operations we shall use in the remaining of the thesis.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>member</td>
<td>testing membership</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>insert</td>
<td>inserting an element</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>union</td>
<td>disjoint union</td>
<td>$O(m \times (\log(n/m) + 1))$</td>
</tr>
</tbody>
</table>

Table 3.1: Data.Set operations, $n$ gives the number of vertices in the first (or only) tree operated upon; for those functions taking two trees as input, $m$ is the size of the second tree. We assume that $m \leq n$, otherwise trees are swapped.

**Example**

The BST corresponding to the input tree in Fig. 3.4 (and illustrated in Fig. 3.5) is defined by

```haskell
Bin  6  5 ( Bin  3  3 (Bin  1  2 Tip Tip) (Bin  1  4 Tip Tip) )
  ( Bin  2  7 Tip )
  ( Bin  1  9 Tip Tip )
```
Figure 3.5: The BST corresponding to the input tree in Fig. 3.4. Notice that the data elements (the numbers within the circles) have been sorted into ascending order. Numbers next to each circle give the size of the tree rooted at that node. Small circles correspond to instances of the Tip data constructor.

3.4 2-3 trees

A 2-3 tree is a tree all of whose internal (non-leaf) nodes hold either 2 or 3 subtrees. The arrangement of data stored in a 2-3 tree varies depending how its data type is defined. Let us start with the one where data is held only on the leaves of the tree (leafy tree), pictured in Figure 3.6.

```haskell
data TreeL a
  = LeafL a
  | Node2L (TreeL a) (TreeL a)
  | Node3L (TreeL a) (TreeL a) (TreeL a)
```

Figure 3.6: Leafy tree, a 2-3 tree with data only on the leaves

The case when data is stored only on the internal nodes of the tree (nodal tree) is defined below and visually represented in Figure 3.7.
The third case, when holding data in every node and leaf (complete 2-3 tree) is defined as follows

```haskell
data TreeB a b -- 2-3 trees holding data in both nodes and leaves
    = LeafB b
    | Node2B (TreeB a b) a (TreeB a b)
    | Node3B (TreeB a b) a (TreeB a b) a (TreeB a b)
```

Notice we define two type arguments \( a \) and \( b \) that may or may not be the same. For illustrative purposes we define two different type arguments in Figure 3.8.

Figure 3.8: A complete 2-3 tree (of type `TreeB Char Int`), defined with two different type arguments for values on the nodes and leaves
3.5 Monoids

A *monoid* is a set \( S \) along with a binary operation \( \star : S \to S \to S \) and a distinguished element \( \epsilon \in S \); subject to the axioms

\[
\epsilon \star x = x \star \epsilon = x \quad (3.1)
\]

monoidal identity

\[
x \star (y \star z) = (x \star y) \star z \quad (3.2)
\]

monoidal associativity

where \( x, y, z \in S \). We denote the above equations in Haskell as follows,

\[
\text{mempty} \ 'mappend' \ x = x \ 'mappend' \ \text{mempty} \quad -- \ [\text{mon.identity}]
\]

\[
x \ 'mappend' \ (y \ 'mappend' \ z) = (x \ 'mappend' \ y) \ 'mappend' \ z \quad -- \ [\text{mon.assoc}]
\]

where \text{mempty} is the \( \epsilon \), and \text{mappend} is the \( \star \) operation.

The Haskell implementation of monoids can be found in the \textbf{Monoid} type class within the \textbf{Data.Monoid} module [46]:

\[
\text{class} \ \text{Semigroup} \ a \Rightarrow \text{Monoid} \ a \ \\
\text{mempty} :: a \\
\text{mappend} :: a \to a \to a
\]

As shown, the \textbf{Monoid} class definition is constrained by \textbf{Semigroup}, which is an algebraic structure with no requirement for an identity element. It is just a set \( S \) with an associative binary operation represented by the \( <> \) symbol.

\[
\text{class} \ \text{Semigroup} \ a \ \\
\ (<> :: a) \to a \to a
\]

We shall use \( <> \) and \text{mappend} interchangably as they refer to the same binary operation. Both classes declare other methods, however they are not used in this thesis. Yorgey [47] presents an interesting collection of applications for monoids, specifically as the
means to design libraries in functional programming, in particular for Haskell.

3.5.1 Monoidal annotation

We use the term *monoidal annotation* to mean the result of performing the binary operation `mappend` in a monoid. Technically, a monoidal annotation is simply a type of data, as this represents the result of `mappend` or the application of `<>` or `⋆` functions.

\[ x \leftrightarrow y = z \]

where `z` is the monoidal annotation when applying `<>` over `x` and `y`.

3.6 Finger Trees

A finger tree, `Ft`, is a complete 2-3 tree which allocates the data at the leaves in such a way that the structure is always balanced. They are typically used as a general representation for sequences [8].

Before describing the finger tree data structure in detail, we note that the notation used in this Section differs slightly from that of Hinze and Paterson [8], as do some of the assumptions we make over such structures. The essential ideas, however, are the same and we shall explain the differences where they occur in Chapters 4, and 5. We assume, for `FT`s, that data is stored only on the leaf-nodes as the internal nodes are left to operational purposes, i.e. monoidal annotations. We distinguish the terms `sequence`, `finger tree` and `list` with the last being the linear representation of the first, and the `finger tree` its non-linear representation, i.e.
a Sequence is represented by a
\[
\begin{align*}
\text{List} & \quad \text{linear, simple, inefficient} \\
\text{FingerTree} & \quad \text{non-linear, complex, efficient}
\end{align*}
\]

When discussing lists we shall call an implementation inefficient if in practice all operations, except for insertion or deletion from the left, require traversing the entire structure in $\mathcal{O}(n)$ time per operation.

### 3.6.1 Structure of Ft

While finger trees are based on complete 2-3 trees, they are more general than the latter since they can perform different tasks due to the monoidal annotations within the internal nodes. A finger tree data structure is defined as follows.

\[
\begin{align*}
a \text{ Ft} \text{ is either} \\
\text{Empty} & \quad \leadsto \text{empty Ft} \\
\text{Single} & \quad \leadsto \text{containing a single element} \\
\text{Deep} & \quad \begin{cases}
\text{prefix} & \leadsto \text{of type Digit, defined below} \\
\text{middle} & \leadsto \text{of type Node, defined below} \\
\text{suffix} & \leadsto \text{of type Digit, defined below}
\end{cases}
\end{align*}
\]

At the top level of a Ft, level zero, the affixes store data, while from level one downwards, the affixes allocate further complete 2-3 trees (see Fig. 3.9).

![Diagram of Finger Tree](image)

**Figure 3.9:** The three data constructors in the Ft definition

Next, we explain how to implement a Ft.
The **Node** data type

Recall the two type arguments from complete 2-3 trees (e.g., `TreeB Char Int`) in Sect. 3.4. For finger trees we will write `v` for the internal node type (with `v` constrained to be an instance of `Monoid`). We will write `a` for the type argument for data on the leaves. So, the **Node** data type for **FT**s is defined by

```haskell
data Node v a = Node2 v a a | Node3 v a a a
```

In Fig. 3.10 below, we represent **Node** with a (blue) circle tagged with `N2` or `N3` labels for the cases of **Node2** and **Node3** data constructors respectively.

**Examples**

We provide two examples, a simple one with two element-values, `e_1` and `e_2`. The second is a recursive example defined by five element-values, `e_1 ... e_5`.

```haskell
nodeExample1 = Node2 z e_1 e_2
nodeExample2 = Node2 z (Node3 z e_1 e_2 e_3) (Node2 z e_4 e_5)
```

that is, `nodeExample1` has type `Node v a` whereas `nodeExample2` has type `Node v (Node v a)`, and value `z` is the monoidal annotation of data on leaves but abstracted away here for simplicity (see Fig. 3.10).

![Figure 3.10: Recursive and non-recursive examples of the Node type](image)

The **prefix and suffix** data types

The following data type, **Digit**, represents the affixes of a **FT**:

```haskell
data Digit a = One a | Two a a | Three a a a | Four a a a a
```

44
The monoidal annotation is not included in the \textit{Digit} type definition, it is performed later when data is inserted or updated. We represent the \textit{Digit} type with a white dialogue box and a Roman numeral for each data constructor.

\section*{Examples}
Consider these definitions (illustrated in Fig. 3.11):

\begin{verbatim}
digitExample1 = Four e₁ e₂ e₃ e₄
digitExample2 = Three (Node2 z e₁ e₂) (Node2 z e₃ e₄) (Node2 z e₅ e₆)
\end{verbatim}

In these examples \texttt{digitExample1} has type \textit{Digit} \textit{a} whereas \texttt{digitExample2} has type \textit{Digit} \textit{(Node v a)}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3_11.png}
\caption{Recursive and non-recursive examples of \textit{Digit} type}
\end{figure}

\subsection*{The \texttt{FingerTree} data type}

Putting all of above data types together, the following is the Haskell definition for the general case of a finger tree \texttt{Ft}.

\begin{verbatim}
data FingerTree v a = Empty
  | Single a
  | Deep
    v
    (Digit a) -- constructor for a FT
    (Digit a) -- monoidal annotation (referred as mon)
    (Digit a) -- prefix of FT (referred as pref)
    (FingerTree v (Node v a)) -- subtree of FT (referred as mid)
    (Digit a) -- suffix of FT (referred as suf)
\end{verbatim}

\subsection*{\texttt{Ft} Example}
The sequence $e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9$ within a \texttt{Ft} can be defined as follows and depicted as in Figure 3.12. We help out the example with auxiliary functions for the \texttt{Ft} affixes.
Deep z pref1 mid1 suf1

where

\[ \text{pref1} = \text{Three } e_1 e_2 e_3 \]
\[ \text{mid1} = \text{Deep z pref2 mid2 suf2} \quad \text{-- recursive FT call} \]
\[ \text{suf1} = \text{Two } e_8 e_9 \]
\[ \text{pref2} = \text{One } (\text{Node2 } z e_4 e_5) \]
\[ \text{mid2} = \text{Empty} \]
\[ \text{suf2} = \text{One } (\text{Node2 } z e_6 e_7) \]

Figure 3.12: \textbf{FingerTree} holding a sequence of nine elements \((e_1 \ldots e_9)\)

### 3.6.2 Amounts of data stored in the FT data structure

In order to have a clear picture for some of the operations over FTs, we need to know the size of such a data structure for the data hold on the leaves (elements of the sequence) and the internal nodes (monoidal annotations). Following the data constructors for the FT data type, depicted in Figure 3.13, we have that every affix contains up to four (Digit) elements, either leaves or subtrees (Node2 or Node3). Since we have two affixes per FT, we then have

\[
2 \sum_{k=0}^{h} (4 \times 3^k)
\]  

(3.3)

where \(h\) is the height for a FT of \(n\) leaf-nodes. The above formula is the case when the element at the bottom of the spine is the Empty data constructor.
For the case when the Single constructor is at the bottom, we simply add a unique subtree built from either Node3 or Node2 constructors, but we take the former for calculating the maximum number of elements. Since the above is the bottom of the spine, the number of nodes takes the height of the FT plus one more level. This is depicted in Figure 3.14

\[
2 \sum_{k=0}^{h} (4 \times 3^k) + 3^{(h+1)}
\]  

Figure 3.13: Maximum number of leaves in a FT, with Empty bottom
For the number of monoidal annotations in a FT, we have those are hold on the spine and on the Node subtrees. In particular, we interested only in the maximum amount, so we calculate only the ones on Node3. We depict this in Figure 3.15. Since we are taking into account all the internal nodes, we have

$$2 \sum_{i=1}^{h} \left( \sum_{j=1}^{i} 3^{j-1} \right) + h + 1$$

(3.5)

The lower bound of the summation is set to 1 since at level 0, we do not have any monoidal annotations stored in the FT. These are calculated at runtime only.
Figure 3.15: Maximum monoidal annotations in a FT, with Empty bottom

Similar to the amount of leaves in a FT, the above formula is for the case when the bottom of the spine is Empty. Otherwise, we add all the internal nodes of Single subtree at the bottom. This can be visualised in Figure 3.16.

$$2 \sum_{i=1}^{h} \left(4 \sum_{j=1}^{i} 3^{j-1}\right) + \sum_{k=1}^{h+1} 3^{k-1} + h + 2$$ (3.6)
Finally, the maximum number of $<>$s (i.e. monoidal binary operation) carried out in a FT occurs one time per Node2 subtrees and two times per Node3 subtrees. Furthermore, in the spine, for every non-empty subtree we have two more $<>$ operations.

$$
2 \sum_{i=1}^{h} \left( 4 \sum_{j=1}^{i} (2 \times 3^{j-1}) \right) + 2h
$$

(3.7)

And the corresponding Single bottom at the spine, we have

$$
2 \sum_{i=1}^{h} \left( 4 \sum_{j=1}^{i} (2 \times 3^{j-1}) \right) + \sum_{k=1}^{h+1} (2 \times 3^{k-1}) + 2h + 1
$$

(3.8)

### 3.6.3 Operations in FT

So far, we have defined data types that deploy abstract examples since we have not defined a monoidal annotation explicitly.
Prior to performing any operation over FTs it is necessary to define a function able to retrieve the monoidal annotation from every Ft data type. Devised by Hinze and Paterson in [8] and implemented in [48], this function is called measure and defined within a type class

```haskell
class (Monoid v) ⇒ Measured v a where
  measure :: a → v
```

This type class is called under different names in other instances of Ft, for instance, in Data.Sequence [49] it is defined as size

```haskell
class Sized a where
  size :: a → Int
```

In general, the Measured class instance is defined for every data constructor in FingerTree, Node and Digit

```haskell
instance (Monoid v) ⇒ Measured v (FingerTree v a) where
  measure Empty = mempty -- [mesEmpty]
  measure (Single x) = measure x -- [mesSingle]
  measure (Deep v _ _ _) = v -- [mesDeep]
```

For the Node data type, it is

```haskell
instance (Monoid v) ⇒ Measured v (Node v a) where
  measure (Node2 v _ _) = v -- [mesNode2]
  measure (Node3 v _ _ _) = v -- [mesNode3]
```

Since Digit data type does not have the type argument v, we map the function measure along its residents (i.e. arguments of its data constructors) through foldMap

```haskell
instance (Monoid v) ⇒ Measured v (Digit a) where
  measure = foldMap measure
```

where foldMap applies function f on behalf of measure as in the following instance

```haskell
instance Foldable Digit where
  foldMap f (One a) = f a -- [mesOne]
  foldMap f (Two a b) = f a <> f b -- [mesTwo]
  foldMap f (Three a b c) = f a <> f b <> f c -- [mesThree]
  foldMap f (Four a b c d) = f a <> f b <> f c <> f d -- [mesFour]
```

In order to deal with different combinations of values and data constructors, in [48] there are plenty of smart constructors. We show here just a sample for the case when given three values, function node3 returns a data constructor for Node type.

```haskell
node3 :: (Measured v a) ⇒ a → a → a → Node v a
node3 x y z = Node3 (measure x <> measure y <> measure z) x y z
```
Function \texttt{node3} builds up a complete 2-3 tree by calculating the monoidal annotation provided three arguments \(x, y\) and \(z\) of type \(a\) (type of data allocated on the leaves).

The library \texttt{Data.FingerTree} \cite{48} is the Haskell implementation of \cite{8}. It contains the data types, constructors, update and transformation function definitions for the general purpose finger tree data structure. Amongst all operations in \texttt{Data.FingerTree}, we limit our proposals \texttt{Full} and \texttt{Top} to the operations listed in Table 3.2.

Notice that each operation in such a table is polymorphic in the finger trees defined in \cite{48}. That is, any operation listed in Table 3.2 is valid when applied to FTs of atomic elements, i.e. \texttt{Int} as well as valid when applied to FTs of elements of type \((\texttt{Node} \ (<>)) (\texttt{Node} \ \texttt{Int}))\). Since \(<>\) has type \texttt{Monoid} \(v \Rightarrow v \rightarrow v \rightarrow v\), we represent the term \textit{monoidal annotation} with \(<>\).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{viewl}</td>
<td>view the first element of the tree</td>
<td>(O(1))</td>
</tr>
<tr>
<td>\texttt{viewr}</td>
<td>view the last element of the tree</td>
<td>(O(1))</td>
</tr>
<tr>
<td>(&lt;)</td>
<td>inserting from the left</td>
<td>(O(1))</td>
</tr>
<tr>
<td>(\triangleright)</td>
<td>inserting from the right</td>
<td>(O(1))</td>
</tr>
<tr>
<td>(\triangleright\triangleright)</td>
<td>appending two trees (concatenation)</td>
<td>(O(\log(\min(n,m))))</td>
</tr>
<tr>
<td>\texttt{split}</td>
<td>split a tree into two subtrees</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>\texttt{search}</td>
<td>looking for an element and perform a \texttt{split}</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

Table 3.2: Finger tree operations, taken from \cite{8}. In each case, \(n\) gives the number of vertices in the first (or only) tree operated upon; for those functions taking two trees as input, \(m\) is the number of vertices in the second tree. The result for \(\triangleright\triangleright\) assumes that \(m \leq n\) (if not, we can swap the order of the arguments before applying \(\triangleright\triangleright\)). All bounds are amortised.

Bounds in the above table are stated to be amortised by Hinze and Paterson in \cite{8}. This is because the output of a specific operation is defined on two or more rules. In general, when the input is short enough in length, such operation gets its fastest computation and when the input is large enough then the output is ‘amortised’ (i.e. divided) amongst the length of the input. We shall provide an example of the amortisation for each operation in Table 3.2. Since a \texttt{FT} is defined along with two type arguments, i.e. the one for the leaves and the one for the monoidal annotations, we shall describe the operation performances implicit on \(<>\), see Section 3.5.1, when such operation is not defined and explicit otherwise.
3.6.4 Accessing the endpoints of a FT

Functions \texttt{viewl} and \texttt{viewr} allow viewing and removing, in $O(1)$ time amortised, the endpoints of a FT. Let us start with the \texttt{viewl} definition. Analysis on \texttt{viewr} applies similarly.

\texttt{viewl :: (Measured v a) \Rightarrow FingerTree v a \rightarrow ViewL (FingerTree v) a}

\texttt{viewl Empty} = \texttt{EmptyL} -- [viewl.empty]
\texttt{viewl (Single x)} = \texttt{x :< Empty} -- [viewl.singleton]
\texttt{viewl (Deep _ (One x) m sf)} = \texttt{x :< rotL m sf} -- [viewl.recursiveCase]
\texttt{viewl (Deep _ pr m sf)}
\hspace{1em} = \texttt{lheadDigit pr :< deep (ltailDigit pr) m sf} -- [viewl.regularCase]

Rules \texttt{viewl.empty} and \texttt{viewl.singleton} are trivial, simply pattern match their input. Rule \texttt{viewl.regularCase} is performed by functions \texttt{lheadDigit}, \texttt{ltailDigit} and \texttt{deep}. The first two, run in \Theta(1) as both just pattern match their first argument as we can see below

\texttt{lheadDigit :: Digit a \rightarrow a}
\texttt{lheadDigit (One a)} = \texttt{a}
\texttt{lheadDigit (Two a _)} = \texttt{a}
\texttt{lheadDigit (Three a _ _)} = \texttt{a}
\texttt{lheadDigit (Four a _ _ _)} = \texttt{a}

Function \texttt{ltailDigit} is defined similarly in [48].

\texttt{deep :: (Measured v a) \Rightarrow Digit a \rightarrow FingerTree v (Node v a) \rightarrow Digit a \rightarrow FingerTree v a}
\texttt{deep pr mid sf} = \texttt{Deep ((measure pr \leftrightarrow measure m) \leftrightarrow measure sf) pr mid sf}

Performance for \texttt{deep} relies on the performance of $\leftrightarrow$ since \texttt{Deep} data constructor simply assembles the remaining parts of this function definition. For instance, in \texttt{Data.Sequence}, the operation $\leftrightarrow$ is the arithmetic addition yielding \texttt{deep} for \texttt{Data.Sequence} to run to in $O(1)$. Therefore, a FT for which it has not been defined its $\leftrightarrow$ operator, we simply state its performance as $O(<>)$.

For the rule \texttt{viewl.recursiveCase}, \texttt{rotL}, we have

\texttt{rotL :: (Measured v a) \Rightarrow FingerTree v (Node v a) \rightarrow Digit a \rightarrow FingerTree v a}
\texttt{rotL m sf} = \texttt{case viewl m of}
\hspace{1em} \texttt{EmptyL \rightarrow digitToTree sf}
\hspace{1em} \texttt{a :< m' \rightarrow Deep (measure m \leftrightarrow measure sf) (nodeToDigit a) m' sf}

Similar to \texttt{lheadDigit}, \texttt{nodeToDigit} perform in \Theta(1). Alike \texttt{deep}, function \texttt{digitToTree} depends upon $\leftrightarrow$ definition. Both, \texttt{nodeToDigit} and \texttt{digitToTree} are detailed in [48].
Complexity of viewl and viewr

We focus on viewl in this thesis, viewr can be analysed similarly. From the lazy evaluation, all the rules defining viewl shall return the left most element of a FT in \( \Theta(1) \) as the tail of such a FT remains unevaluated. Otherwise, the expressions evaluation turns on to strict. This is when the worst case, under the rule viewl.recursiveCase comes to play. Suppose we are required to return the left end of a FT along its tail. Suppose further that at every level down the tree there are only One data constructors down to the bottom of the spine being Empty its case. Having \( n \) total leaves in the FT, we have \( \mathcal{O}(\log n) \) cases to evaluate as viewl traverses the height of the FT. Therefore the total amount of time to perform such a case is \( \mathcal{O}(\log n) \times \mathcal{O}(<>). \) Once again, we leave the operation <> implicit as no monoidal annotation has been defined. Now, the total cost of the above performance is divided by the total number of elements evaluated as left-most for the viewl, that is, there are \( \mathcal{O}(\log n) \) elements on the far left yielding to \( \mathcal{O}(1) \times \mathcal{O}(<> \text{ per viewl operation).} \)

That is, getting the first element in a FT where its monoidal annotation is defined by Data.Sequence, which is the arithmetic addition, we have that each viewl takes \( \mathcal{O}(1) \), whereas defining Data.Set, seen in Section 3.3, as the monoidal annotation, the same operation shall take at most \( \mathcal{O}(\log n) \) time per viewl operation.

3.6.5 Inserting at the endpoints of a FT

These functions insert, in \( \mathcal{O}(1) \) amortised time, an element\(^2\) at the front (<) or at the rear (>) of a FT. We present the < case. Details of > can be found in \[48\]. In the following snippet, mon stands for the monoidal annotation, pref the prefix of FT, mid the subtree of current FT and suf the suffix of FT. Recall that measure is the function retrieving monoidal annotation values.

\[
\begin{align*}
\text{a} \triangleright \text{Empty} & = \text{Single a} & \text{-- trivial case} & [\triangleright .1] \\
\text{a} \triangleright \text{Single b} & = \text{deep (One a) Empty (One b)} & \text{-- balancing trivial case} & [\triangleright .2] \\
\text{a} \triangleright \text{Deep mon (Four b c d e) mid suf} & = \text{-- prefix of FT is full} & [\triangleright .3] \\
& \text{Deep} & \text{-- FT constructor, persistently} \\
& \text{(measure a <> mon)} & \text{-- updating monoidal annotation} \\
& \text{(Two a b)} & \text{-- a new prefix is built} \\
& \text{(node3 c d e <> mid)} & \text{-- new Node is created into mid} \\
& \text{suf} & \text{-- suffix is left intact} \\
\text{a} \triangleright \text{Deep mon pref mid suf} & = \text{--} & [\triangleright .4]
\end{align*}
\]

\(^2\)Polymorphic in FingerTree data type
Deep -- FT constructor, persistently
(measure a <|> mon) -- updating monoidal annotation
(consDigit a pref) -- new value inserted into prefix
mid -- middle part is left intact
suf -- suffix is left intact

consDigit :: a → Digit a → Digit a
consDigit a (One b) = Two a b -- [consDig.1]
consDigit a (Two b c) = Three a b c -- [consDig.2]
consDigit a (Three b c d) = Four a b c d -- [consDig.3]

Rules <|.1 and <|.2 are trivial running in $\Theta(1)$ as both simply pattern match the arguments. Similarly, function consDigit runs also in $\Theta(1)$ since it just patterns match its arguments. Rule <|.4 relies on the operator <|> and the Deep data constructor, so its performance is $O(|<>|)$. The remaining rule, <|.3, is the recursive case when inserting from the left into a FT.

**Complexity of <|**

It is stated in Table 3.2 that <| performs in $O(1)$ amortised. In order to get such performance it is assumed the binary operation <|> defined for the finger tree in matter, performs in constant time. We extend some analysis to determine the number of <|> operations applied within <|. Let us take an initial example of inserting six elements ($x_1...x_6$) into an Empty finger tree. By doing this, we shall enforce the four rules of <|.

$x_1 <| x_2 <| x_3 <| x_4 <| x_5 <| x_6 <| Empty$

= { by <|.1 }  -- [Op.1]
$x_1 <| x_2 <| x_3 <| x_4 <| x_5 <| (Single x_6)$
= { by <|.2 }  -- [Op.2]
$x_1 <| x_2 <| x_3 <| x_4 <| (\text{deep} \ (\text{One} \ x_5) \ \text{Empty} \ (\text{One} \ x_6))$
= { by deep function definition }  -- [Op.3]
$x_1 <| x_2 <| x_3 <| x_4 <| (\text{Deep} \ (<>) \ \text{One} \ x_5) \ \text{Empty} \ (\text{One} \ x_6)$
= { by <|.4 }  -- [Op.4]
$x_1 <| x_2 <| x_3 <| (\text{Deep} \ (<>) \ (\text{consDigit} \ x_4 \ (\text{One} \ x_5)) \ \text{Empty} \ (\text{One} \ x_6))$
= { by consDig.1 }  -- [Op.5]
$x_1 <| x_2 <| x_3 <| (\text{Deep} \ (<>) \ (\text{Two} \ x_4 \ x_5) \ \text{Empty} \ (\text{One} \ x_6))$
= { by <|.6 }  -- [Op.6]
$x_1 <| x_2 <| (\text{Deep} \ (<>) \ (\text{consDigit} \ x_3 \ (\text{Two} \ x_4 \ x_5)) \ \text{Empty} \ (\text{One} \ x_6))$
= { by consDig.2 }  -- [Op.7]
$x_1 <| x_2 <| (\text{Deep} \ (<>) \ (\text{Three} \ x_3 \ x_4 \ x_5) \ \text{Empty} \ (\text{One} \ x_6))$
= { by <|.4 }  -- [Op.8]
$x_1 <| (\text{Deep} \ (<>) \ (\text{consDigit} \ x_2 \ (\text{Three} \ x_3 \ x_4 \ x_5)) \ \text{Empty} \ (\text{One} \ x_6))$
= { by consDig.3 }  -- [Op.9]
x₁ < (Deep (<>₄) (Four x₂ x₃ x₄ x₅) Empty (One x₆))
= { by <₃ }  -- [Op.10]
Deep (<>₅) (Two x₁ x₂) (node₃ x₃ x₄ x₅ < Empty) (One x₆)
= { by node₃ function definition }  -- [Op.11]
Deep (<>₅) (Two x₁ x₂) (Node₃ (<>₆) x₃ x₄ x₅ < Empty) (One x₆)
= { by <₇₁ }  -- [Op.12]
Deep (<>₅) (Two x₁ x₂) (Single (Node₃ (<>₆) x₃ x₄ x₅)) (One x₆)

We have obtained six place holders for <>, where some of them represent just one appearance and other more than one. The following table details the accumulation for such operator.

<table>
<thead>
<tr>
<th>Op</th>
<th>xᵢ</th>
<th>&lt;&gt;ᵢ</th>
<th>&lt;&gt;ₜₐₜₜ</th>
<th>function</th>
<th>FT depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x₆</td>
<td>&lt;&gt;₀</td>
<td>Single</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>x₅</td>
<td>&lt;&gt;₁</td>
<td>deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>x₅</td>
<td>&lt;&gt;₁</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>x₄</td>
<td>&lt;&gt;₂</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>x₄</td>
<td>&lt;&gt;₂</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>x₃</td>
<td>&lt;&gt;₃</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>x₃</td>
<td>&lt;&gt;₃</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>x₂</td>
<td>&lt;&gt;₄</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>x₂</td>
<td>&lt;&gt;₄</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>x₁</td>
<td>&lt;&gt;₅</td>
<td>Deep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>x₁</td>
<td>&lt;&gt;₆</td>
<td>Node₃</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>x₁</td>
<td>&lt;&gt;₅₆</td>
<td>Empty</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: Operations involved in performing six insertions into an empty Ft

Following Table 4.7, we have eight <> operations at level 0 of the Ft depth. The ninth <> operator corresponds not only to Ft depth 1 but for a second call to < (by [Op.10–Op.12]). Since the < type is polymorphic, it computes any type of finger tree³ in at most eight <> operators per level in the Ft depth. That is, time complexity for < relies in the time complexity of the monoidal binary operation <>, which is 8 × O(<>) = O(<>).

Suppose x is a valid datum for t where t is a Ft of depth O(log n) which has all of its prefixes full, that is, every prefix is comprised of Four data constructors. Furthermore, assume that every Node is built only by Node₃ constructors. Then, defining ins as ins = x < t, it will perform in 8 × O(log n) × O(<>). Hence the amortised time per insertion is O(log n) ×

³FingerTree a, FingerTree (<> (Node a)), FingerTree (<> (Node (<> (Node (<> a))))), etc.
\(O(\langle\rangle)\) divided by total number of elements inserted. In the above case we have inserted a single element always from the left, that is \(O\log n\) elements, as we started from the top of the FT all the way down to the bottom of it. Alike \texttt{view1}, inserting an element from the left shall take \(O(\langle\rangle)\) per operation when monoidal annotation is not defined.

### 3.6.6 Appending FTs

Let \(ft1\text{=}Deep\ mon1\ pr1\ mid1\ sf1\) and \(ft2\text{=}Deep\ mon2\ pr2\ mid2\ sf2\) be two finger trees. The operator \(\triangleright\triangleright\), defined in [8] and implemented in [48], takes two finger trees, say \(ft1\) and \(ft2\), and appends them from the middle. In brief, \(\triangleright\triangleright\) performs

1. generates a \textit{new} subtree, \texttt{mid}, concatenating \texttt{mid1}, \texttt{sf1}, \texttt{pr2} and \texttt{mi2}

2. creates a \textit{new} finger tree \(ft = Deep\ (mon1 \langle\rangle mon2)\ pr1\ mid\ sf2\)

The appending process, numeral 1 above, is carried out by recursive calls of \textit{interleaved} functions \texttt{appendTree\_i} and \texttt{addDigits\_i}. The former inserts the elements \texttt{a} via \(\triangleright\) and \(\triangleright\). The latter function compares all combinations possible from \texttt{Digit} data constructor. Index \(i \in \{0 \ldots 4\}\) refers to a postfix in the name of such function in [48].

As a general definition for \texttt{appendTree\_i}, we have

\[
\text{appendTree\_i} :: (Measure v a) \Rightarrow \text{FingerTree v a} \rightarrow \alpha_i \rightarrow \text{FingerTree v a} \rightarrow \text{FingerTree v a}
\]

\[
\begin{align*}
\text{appendTree\_i} \text{ Empty} & \alpha_i \ x s = \alpha_i \triangleright \alpha_i \ x s & \text{-- [appendTree\_i.1]} \\
\text{appendTree\_i} \ x s & \alpha_i \text{ Empty} = x \triangleright \alpha_i \triangleright \ x s & \text{-- [appendTree\_i.2]} \\
\text{appendTree\_i} \ (\text{Single} \ x) & \alpha_i \ x s = x \triangleright \alpha_i \triangleright \ x s & \text{-- [appendTree\_i.3]} \\
\text{appendTree\_i} \ x s & \alpha_i \ (\text{Single} \ x) = x \triangleright \alpha_i \triangleright \ x & \text{-- [appendTree\_i.4]} \\
\text{appendTree\_i} \ (\text{Deep} \ mon1\ pr1\ mid1\ sf1) & \alpha_i \ (\text{Deep} \ mon2\ pr2\ mid2\ sf2) \\
& = \text{Deep} \ (\text{mon1} \langle\rangle \text{ measure}_i \ alpha_i \langle\rangle \text{ mon2}) \\
& \quad \text{pr1} \ (\text{addDigits}_i, \ text{mid1} \ sf1 \ alpha_i \ pr2 \ mid2) \ sf2 & \text{-- [appendTree\_i.5]}
\end{align*}
\]

where \(\alpha_0\) indicates zero elements, \(\alpha_1 = a\), \(\alpha_2 = a \ b\), \(\alpha_3 = a \ b \ c\), and \(\alpha_4 = a \ b \ c \ d\).

A general function definition for \texttt{addDigits\_i} can be

\[
\text{addDigits\_i} :: (Edges v a) \Rightarrow \text{FingerTree v (Node v a)} \rightarrow \text{Digit a} \rightarrow \alpha_i \rightarrow \text{Digit a} \\
\text{addDigits\_i} \text{ m1} \ (\text{One a}) \ alpha_i \ (\text{One f}) \ m2
\]
\[
\begin{align*}
\text{addDigits}_i(m_1 \cdot (\text{One } a) \cdot (\text{Two } f g) \cdot m_2) & = \text{appendTree}_i(mid_1(\text{node}_k), \ldots, mid_2) \quad \text{-- [addDigits}_i.1] \\
\text{addDigits}_i(mid_1(\text{Four } a b c d) \cdot \alpha_i(\text{Three } i j k) \cdot m_2) & = \text{appendTree}_i(mid_1(\text{node}_k), \ldots, mid_2) \quad \text{-- [addDigits}_i.2] \\
\ldots & \quad \text{-- [addDigits}_i.(3..14)] \\
\text{addDigits}_i(mid_1(\text{Four } a b c d) \cdot \alpha_i(\text{Four } i j k l) \cdot m_2) & = \text{appendTree}_i(mid_1(\text{node}_k), \ldots, mid_2) \quad \text{-- [addDigits}_i.16] \\
\end{align*}
\]

where \((\text{node}_k)\) can be either \((\text{node3 } a b c)\) or \((\text{node2 } a b)\)

**Complexity of \(\bowtie\)**

Since the actual implementation of \(\bowtie\) in [48] is defined throughout 106 lines of Haskell code \((5 \times \text{appendTree} + 16 \times \text{addDigits} + \bowtie)\), we just show a sample in this thesis.

\[
\begin{align*}
\text{pr1} & = (\text{One } x_1) \\
\text{mid1} & = \text{Empty} \\
\text{sf1} & = (\text{Four } x_2 \ x_3 \ x_4 \ x_5) \\
\text{pr2} & = (\text{Three } x_6 \ x_7 \ x_8) \\
\text{mid2} & = \text{Empty} \\
\text{sf2} & = (\text{One } x_9) \\
\text{Deep mon}_1 \text{ pr1 mid1 sf1 } \bowtie \text{ Deep mon}_2 \text{ pr2 mid2 sf2} & = \{ \text{ by } \bowtie \} \quad \text{-- [Op.1]} \\
\text{appendTree0} & = \{ \text{ by appendTree0 function definition } \} \quad \text{-- [Op.2]} \\
\text{Deep (mon}_1 \leftrightarrow \text{ mon}_2) \text{ pr1 (addDigits0 mid1 sf1 pr2 mid2) sf2} & = \{ \text{ by addDigits0.15 } \} \quad \text{-- [Op.3]} \\
\text{Deep } (\langle\rangle_1) (\text{One } x_1) & \quad \text{-- [Op.4]} \\
\text{Deep } (\langle\rangle_1) (\text{One } x_1) & \quad \text{-- [Op.5]} \\
\end{align*}
\]
Update operation is actually carried out by $\ll$ and $\llcorner$ throughout 106 function definitions but only 10 of them can be performed since it is one function call in 5 appendTree definitions and one call in 5 addDigits definitions. That is, given two finger trees of $n$ and $m$ number of elements at their leaves respectively, performance of $\llcorner \ll$ is $O(\log(\min(n,m))) \times O(\llcorner)$.

### 3.6.7 Searching and splitting in $Ft$

All $Ft$ operations we have seen so far do not take advantage of the monoidal annotations. They simply perform the corresponding $\llcorner$ per update. When looking up for a specific element in $Ft$ we use the value located at specific monoidal annotation as argument in a predicate. In this section we focus on function $\text{search}$ since it includes a $\text{split}$. A successful $\text{search}$ in [8] implemented in [48] is considered when given a predicate with monoidal annotations as arguments, turns from $\text{False}$ to $\text{True}$, splitting the input $Ft$ into three components: the left subtree, the element in matter and right subtree. In order to provide predictable results, [8] states that uniqueness of the split is guaranteed for monotonic predicates. So, the type signature for the predicate defines two monoidal annotations, the first one to evaluate to $\text{False}$ and the latter for testing $\text{True}$. To catch up all possible results from searching, the following defines the data type for such results.

```hs
data SearchResult v a
  = Position (FingerTree v a) a (FingerTree v a) -- success [srchR.1]
  | OnLeft    -- failed, predicate is True at both ends [srchR.2]
  | OnRight   -- failed, predicate is False at both ends [srchR.3]
  | Nowhere   -- failed, predicate is True at left end
               -- and False at the right end [srchR.4]
```

Since a $\text{FingerTree}$ consists of three data types, [48] defines one search function per data type. We show the initial definitions in each case as we discuss details on searching for our proposals $\text{FULL}$ and $\text{TOP}$.
Initial search on a FT

Searching in the top structure of a FT determines whether or not the element in matter is *somehow* in the structure. It does not perform a precise location for such element, it just asks: Is element \( x \) (within the predicate) in \( FT \)?

```haskell
search :: (Measure v a) ⇒
    (v → v → Bool) → FingerTree v a → SearchResult v a
search p t
| p_left && p_right = OnLeft -- [search.1]
| not p_left && p_right =
    case searchTree p mempty t mempty of
    Split l x r → Position l x r -- [search.2]
| not p_left && not p_right = OnRight -- [search.3]
| otherwise = Nowhere -- [search.4]
where
  p_left = p mempty vt
  p_right = p vt mempty
  vt = edges t
```

In [search.2] we evaluate the case when given predicate \( p \) evaluates from False to True. All other cases represent a failed search. Function searchTree returns the precise location by returning the split of FT,

```haskell
data Split t a = Split t a t
```

Searching in FingerTree

Function searchTree [48] determines if the searched element is either in the prefix, suffix or in the middle.

```haskell
searchTree :: (Measured v a) ⇒
    (v → v → Bool) → v → FingerTree v a → v →
    Split (FingerTree v a) a
searchTree _ _ Empty _ = error "searchTree invalid" -- [search.Error]
searchTree _ _ (Single x) _ = Split Empty x Empty -- [searchTree.1]
searchTree p mon1 (Deep _ pr mid sf) mon2
| p mon1 pr mon2sm =
  let Split l x r = searchDigit p mon1 pr mon2sm
  in Split (maybe Empty digitToTree l) x (deepL r mid sf) -- [Split.1]
| p mon1 pm mon2sf =
  let Split ml xs mr = searchTree p mon1pr mid mon2sf
  Split l x r = searchNode p
  (mon1pr <> measure ml) xs
  (measure mr <> mon2sf)
```

60
If the searched element is within the top affixes (i.e. level 0) or in `Single` implies no recursion on `searchTree`. On the other hand, a `searchTree` applied to a `Ft` of depth \(\geq 1\) ends up in the analysis of `searchNode` either because the element in the prefix, \([searchTree.2]\), in the suffix, \([searchTree.4]\), or at the end of the spine of `Ft`, via \([searchTree.1]\). Since `searchTree` was called from `search` after evaluating `True` \([search.2]\), any of the \([searchTree.1,2,3,4]\) shall return the searched element via `Split`, otherwise an error arises in \([search.Error]\). A `Split` is a constructor for two subtrees and searched element. It is helped out by other smart constructors: `deepL`, `deepR` and `digitToTree` [48] explained further in this section.

**Searching in `Digit`**

Looking for an element in an affix is performed by pattern matching the data constructors for `Digit`. Once a data constructor is chosen, evaluating the predicate against the monoidal annotations provided by the elements in `Digit`, a `Split` of the `Ft` is returned.

\[
\text{searchDigit} :: (\text{Edges } v \ a) \Rightarrow
\quad (v \rightarrow v \rightarrow \text{Bool}) \rightarrow v \rightarrow \text{Digit } a \rightarrow v \rightarrow
\quad \text{Split (Maybe (Digit a)) } a
\text{searchDigit} p \ \text{mon} \ _1 \ (\text{One } a) \ \text{mon} \ _2 = \text{Split Nothing a Nothing} \quad \text{-- [searchDig.1]}
\text{searchDigit} p \ \text{mon} \ _1 \ (\text{Two } a \ b) \ \text{mon} \ _2
\quad | p \ \text{mon} \ _a \ \text{mon} \ _b \ =\text{Split Nothing a (Just (One b))} \quad \text{-- [searchDig.2]}
\quad | \text{otherwise} \ = \text{Split (Just (One a)) b Nothing} \quad \text{-- [searchDig.3]}
\text{where}
\quad \text{mon} \ _a = \text{measure a}
\quad \text{mon} \ _b = \text{measure b} \leftrightarrow \text{mon} \ _2
\]

... 
\text{searchDigit} p \ \text{mon} \ _1 \ (\text{Three } a \ b \ c) \ \text{mon} \ _2

... 
\text{searchDigit} p \ \text{mon} \ _1 \ (\text{Four } a \ b \ c \ d) \ \text{mon} \ _2

61
Searching in **Node**

The ultimate frontier for searching an element in a FT with depth $\geq 1$ is defined in `searchNode`. Like `searchDigit`, this function also patterns match on its data constructors and splits up the given FT via the monoidal annotations provided as arguments in `Node2` or `Node3`.

```haskell
searchNode :: (Edges v a) ⇒
    (v → v → Bool) → v → Node v a → v →
    Split (Maybe (Digit a)) a
searchNode p mon
  1 (Node2 _ a b) mon
  2 |
| p mon a mon b = Split Nothing a (Just (One b)) -- [searchNode.1]
| otherwise = Split (Just (One a)) b Nothing -- [searchNode.2]
where
  mon a = mon 1 <> measure a
  mon b = measure b <> mon 2
... 
searchNode p mon
  1 (Node3 _ a b c) mon
```

**Complexity of search**

Function `search` performs two implicit operations, a look up and a split. Let $t$ be a FT of $n$ elements and depth $O(\log n)$, where $n > 0$. We are interested in `search` for $x$, an element located at the deepest prefix. So, predicate in [search.2] evaluates `True` and `searchTree` is called $O(\log n)$ times while performing up to four $<>$ operators when comparing the predicate $p$, that is, looking up for an element takes $O(\log n) \times O(<>)$. In Figure 3.17 we depict the look up for $x$.
Once element \( x \) has been found, \textit{Split} function performs a bottom-up computation to construct two subtrees. Function \textit{searchTree} calls for \([\text{Split}_{1,2,3}]\) which in turn calls smart constructors \textit{deepL}, \textit{digitToTree} and \textit{deepR}. These constructors \textit{glue} the corresponding affixes and middle subtrees through \textit{rotL}, \textit{deep} and \textit{rotR} helper functions. We present here just the functions processing the left hand side structures. Provided two affixes and a subtree (\textit{mid}), \textit{deepL} builds up a FT. In the absence of a prefix (passed as \textit{Nothing}), \textit{deepL} builds the FT from the \textit{mid} and the suffix; in case \textit{mid} is \textit{Empty}, then FT is made of the suffix solely. The last two cases are performed by \textit{rotL} and \textit{digitToTree}.

\begin{align*}
deepL & \colon (\text{Measured } v \ a) \Rightarrow \text{Maybe } (\text{Digit } a) \Rightarrow \\
& \quad \text{FingerTree } v \ (\text{Node } v \ a) \Rightarrow \text{Digit } a \Rightarrow \text{FingerTree } v \ a \\
\text{deepL Nothing mid sf} & = \text{rotL mid sf} \\
\text{deepL (Just pr) mid sf} & = \text{deep pr mid sf}
\end{align*}

\begin{align*}
\text{rotL} & \colon (\text{Measured } v \ a) \Rightarrow \\
& \quad \text{FingerTree } v \ (\text{Node } v \ a) \Rightarrow \text{Digit } a \Rightarrow \text{FingerTree } v \ a \\
\text{rotL mid sf} & = \text{case viewL mid of} \\
& \quad \text{EmptyL} \rightarrow \text{digitToTree sf} \\
& \quad a :< \text{mid'} \rightarrow \text{Deep (measure mid <> measure sf)} \\
& \quad \ (\text{nodeToDigit a) mid'} \ \text{sf}
\end{align*}

\begin{align*}
\text{digitToTree} & \colon (\text{Measured } v \ a) \Rightarrow \text{Digit } a \Rightarrow \text{FingerTree } v \ a \\
\text{digitToTree } (\text{One } a) & = \text{Single } a \\
\text{digitToTree } (\text{Two } a \ b) & = \text{deep } (\text{One } a) \ \text{Empty } (\text{One } b) \\
\text{digitToTree } (\text{Three } a \ b \ c) & = \text{deep } (\text{Two } a \ b) \ \text{Empty } (\text{One } c) \\
\text{digitToTree } (\text{Four } a \ b \ c \ d) & = \text{deep } (\text{Two } a \ b) \ \text{Empty } (\text{Two } c \ d)
\end{align*}
The following illustration summarises the calls to the above smart constructors and helper functions.

```
\begin{align*}
\text{a Split calls} & \quad \left\{ \\
\text{digitToTree} & \leadsto \text{deep} \\
\text{deepL} & \quad \text{deep} \\
\text{rotL} & \quad \{ \text{Deep}(\langle\rangle) \text{pr mid sf} \\
\text{digitToTree} & \leadsto \text{deep} \}
\end{align*}
```

Since building up the subtrees is bottom up, it takes $O(\log n)$ to get level zero. Then, taking the amount of $\langle\rangle$ operators into account, being `deep` the largest with two⁴, the runtime for `Split` takes is $2 \times O(\log n) \times O(\langle\rangle)$ per subtree. Figure 3.18 illustrates the subtrees construction via `Split`.

![Figure 3.18: Search as a Split of a Ft into two subtrees and an element](image)

**Sequence as BST, an application of a finger tree**

We follow the implementation provided by Hinze and Paterson in [8]. For illustrative purposes, we show only the structural definitions and incremental case, referring the reader to Section 4.7 of [8] for the `deletion` and `$\uplus$` operations.

We define our sequence to be a `Ft` holding element `Elem` of any type `a` with its monoidal annotation being of type `Key a`.

---

⁴see function definition of `deep` in 3.6.4
newtype OrdSeq a = OrdSeq ( FingerTree (Key a) (Elem a) )
emptyOrdSeq  = OrdSeq empty -- constructor OrdSeq followed by an empty FT
newtype Elem a = Elem a deriving (Eq, Ord)
data Key a = NoKey | Key a deriving (Eq, Ord)

By defining NoKey the identity element and Key being the last (maximum) element selected so far, we can instantiate data type Key as a monoid

instance Monoid (Key a) where
  mempty       = NoKey -- the identity element
  mappend k NoKey = k
  mappend _ k    = k -- 2nd arg. is the maximum [<>.OrdFT]
                    -- guaranteed by the update operations

Also we need to define how a single element is measured

instance Measured (Key a) (Elem a) where
  measure (Elem x) = Key x -- [measure.Elem]

Finally, the insertion operation is split in two functions, ins and insert. The former, our contribution to show explicitly all three cases when inserting an element into a sequence. The latter, defined originally by Hinze and Paterson [8], performs the case when the input is neither the maximum nor the minimum element to be in the sequence.

ins :: (Ord a) ⇒ a → OrdSeq a → OrdSeq a
ins x os@(OrdSeq xs)
  | Key x ≥ measure xs = OrdSeq ( xs ⊢ (Elem x) )
  | Elem x ≤ leftmost = OrdSeq ( (Elem x) < xs )
  | otherwise = insert x os
where
  (leftmost :< _) = viewl xs

insert :: (Ord a) ⇒ a → OrdSeq a → OrdSeq a
insert x (OrdSeq xs) = OrdSeq ( left ∞ (Elem x < right) )
where (left, right) = split (≥ Key x) xs

In the following figures we show the insertion of ten elements, Elem 1 ... Elem 10 not necessarily in order, into an empty FT. We start picturing the data at the leaves in Figure3.19
Figure 3.19: Ordered-set via $\text{Ft}$, data ($\text{Elem}$ type) at the leaves

By applying rule $[\text{measure.Elem}]$ we obtain the initial monoidal annotations ($\text{Keys}$), as seen in Figure 3.20

Figure 3.20: Ordered set via $\text{Ft}$, initial monoidal annotations

Finally, in Figure 3.21 it is shown the application of the monoidal binary operation $[<.\text{OrdFT}]$ all over the tree.

Figure 3.21: Ordered set via $\text{Ft}$, data and monoidal annotations in the entire structure
Chapter 4

Euler-Tour Trees Functionally, FunETT

In this chapter we discuss the specifications regarding the construction and manipulation, link and cut, of Euler-tour trees. Specifically, we analyse the work done by Henzinger and King [5], Ett-HK, and Tarjan [7], Ett-T. Then, we describe FunETT, our purely functional programming proposal implemented in Haskell and show its performance. Finally, we summarise this chapter with a brief comparison between the tree specifications. For practical purposes, we refer to the following example as the input (arbitrary) tree for the three specifications.

4.1 Euler-tour trees by Henzinger and King

We follow the specification for the representation of the Ett data structure and its operations from [5].
4.1.1 Representation of the input tree

Henzinger and King encode the input tree of $n$ vertices using a sequence of $2n - 1$ symbols generated by the following procedure called ET, adapted from [5].

Root the tree at an arbitrary vertex
Call ET(root)

ET(x)
visit x;
for each child c of x do
  ET(c);
  visit x.

In the above procedure, every visit $i$ to a vertex $v$ is stored as $o_{v_i}$, the $i$th occurrence of $v$, into the sequence ET. An instance of this procedure to the input tree in Figure 4.1 results in $ET(c) = o_7^1 o_9^1 o_5^1 o_9^2 o_2^1 o_9^3 o_4^1 o_9^4 o_3^1 o_9^5 o_{T2}^1$.

Such representation does not offer uniqueness for the vertices nor the edges allocated in ET.

4.1.2 Operations on ETT-HK

Cutting a tree is referred as deletion of an edge, defined by Henzinger and King in [5] as

To delete edge $\{a, b\}$ from $T$: Let $T_1$ and $T_2$ be the two trees that result, where $a \in T_1$ and $b \in T_2$. Let $o_{a_1}$, $o_{b_1}$, $o_{b_2}$ represent the occurrences encountered in the two traversals of $\{a, b\}$. If $o_{a_1} < o_{b_1}$ and $o_{b_1} < o_{b_2}$, then $o_{a_1} < o_{b_1} < o_{b_2} < o_{a_2}$. Thus, $ET(T_2)$ is given by the interval of $ET(T)$ $o_{b_1}, \ldots, o_{b_2}$ and $ET(T_1)$ is given by splicing out of $ET(T)$ the sequence $o_{b_1}, \ldots, o_{a_2}$.

We point out the following features from ETT-HK deletion operation

1. A conditional via the operator $<$ between occurrences ensures that edge $\{a, b\}$ is in the sequence ET, otherwise no deletion is performed.

2. Computing $ET(T_2)$ requires two implicit look up operations, one for $o_{b_1}$ and one for $o_{b_2}$. Then, an implicit split is performed when “...is given by the interval...” is stated.
3. Computing $\text{ET}(T_1)$ requires two *implicit* look up operations, one for $o_1$ and one for $o_2$. Then, an *explicit* split and append are performed when splicing out the above interval.

4. An interesting point is the notion of immutability when sequences for $T_1$ and $T_2$ are split up from $T$.

Prior to link two trees in Ett-HK, a rerooting operation specified by Henzinger and King in [5], is defined as follows,

**To change the root of $T$ from $r$ to $s$:** Let $o_s$ denote any occurrence of $s$. Splice out the first part of the sequence ending with the occurrence before $o_s$, remove its first occurrence ($o_r$), and tack the first part on to the end of the sequence, which now begins with $o_s$. Add a new occurrence $o_s$ to the end.

The features we have found

1. One look up is performed for $o_s$.
2. One split is carried out when splicing out the $\text{ET}(T_s)$
3. One deletion for $o_r$
4. One append by tacking the first part onto the last part of the split
5. One insertion when adding $o_s$ at the tail of the new sequence

Linking two trees in Ett-HK are referred as *joining* two rooted trees

**To join two rooted trees $T$ and $T'$ by edge $e$:** Let $e = \{a, b\}$ with $a \in T$ and $b \in T'$. Given any occurrences $o_a$ and $o_b$, reroot $T'$ at $b$, create a new occurrence $o_{a_n}$ and splice the sequence $\text{ET}(T')o_{a_n}$ into $\text{ET}(T)$ immediately after $o_a$. Henzinger and King [5]

Finally, the features from Ett-HK joining operation

1. Occurrences $o_a$ and $o_b$ require two look up operations
2. Extra operations are performed when calling *reroot*
3. When creating \( o_{a_n} \), all occurrences of \( o_a \) should be counted, that is, looking for \( n - 1 \) occurrences of \( a \) might take linear time in the size of the sequence. We take this operation as simple look up.

4. One insertion when placing \( o_{a_n} \) at the tail of \( \text{ET}(T') \).

5. Two append and one split operations are performed when splicing out \( \text{ET}(T')o_{a_n} \) into \( \text{ET}(T) \).

6. Immutability is not preserved for \( \text{ET}(T) \) in above operation.

7. Tree \( T \) is not rerooted at \( a \) during the join operation.

### 4.2 Euler-tour trees by Tarjan

We follow the specification for the representation of the Ett data structure and its operations from [7].

#### 4.2.1 Representation of the input tree

A list \( L \) representing an arbitrary tree \( T \) is formed by

- For every edge \( \{v, w\} \in T \) there are two (directed) edges \((v, w)\) and \((w, v)\) in \( L \).
- For every vertex \( v \in T \), there is a unique pair \((v, v)\) in \( L \).

Since \( T \) is a data structure for a tree, there is a unique edge between two vertices. Since the edges placed in \( L \) are directed, the uniqueness in edges is preserved. In the above representation, \( L \) has \( 3n - 2 \) pairs, each pair having either an edge or a vertex.

The input tree from Figure 4.1 is represented in Ett-HK as list \( L = [(7, 7), (7, 9), (9, 9), (9, 5), (5, 5), (5, 9), (9, 2), (2, 2), (2, 9), (9, 4), (4, 4), (4, 9), (9, 3), (3, 3), (3, 9), (9, 7)] \).

#### 4.2.2 Operations on Ett-T

The terms \textit{catenate} and \textit{append} are interchangeable. Tarjan in [7] defines the link operation as follows,
...suppose \( \text{link}(\{v, w\}) \) is selected. Let \( T_1 \) and \( T_2 \) be the trees containing \( v \) and \( w \) respectively, and let \( L_1 \) and \( L_2 \) be the lists representing \( T_1 \) and \( T_2 \). We split \( L_1 \) just after \((v, v)\), into lists \( L_1^1, L_1^2 \), and we split \( L_2 \) just after \((w, w)\) into \( L_2^1, L_2^2 \). Then we form the list representing the combined tree by catenating the six lists \( L_1^1, L_1^2, [\{v, w\}], L_2^1, L_2^2, [(w, v)] \) in order. Thus linking takes two splits and five catenations; two of the latter are the special case of catenation with singleton lists ...

Notice that list \( L \) in its original order consists of \( L_1^1 L_1^2 \), achieving the re-root operation by simply appending the swapped lists as in \( L_1^2 L_1^1 \), provided that the split was done at the specific vertex. Additionally, the second tree represented by \( L_2 \) has also been rerooted as \( L_2^2 L_2^1 \). This is a particular difference with respect to \text{link} in Ett-HK where only the second (\( T' \)) tree is rerooted.

...perform \( \text{cut}(\{v, w\}) \). Let \( T \) be the tree containing \( \{v, w\} \), represented by list \( L \). We split \( L \) before and after \((v, w)\) and \((w, v)\), into \( L^1, [(v, w)], L^2, [(w, v)], L^3 \) (or symmetrically \( L^1, [(w, v)], L^2, [(v, w)], L^3 \)). The lists representing the two trees formed by the cut are \( L^2 \) and the list formed by catenating \( L^1 \) and \( L^3 \). Thus cutting takes four splits (of which two are the special case of splitting off one element) and one catenation ...

4.3 FUNETT

We introduce FUNETT data structure in this section considering the features analysed from both Ett-HK and Ett-T. In particular, like Ett-HK we define an explicit operation for rerooting trees. Like Ett-T, we consider the list-like representation of the input tree. Unlike the Ett-HK and Ett-T, we define our proposal to be

1. Immutable, purely functional

2. Explicit, defining operations search (look up and split), append, and insert altogether the specification
4.3.1 Representation of the input tree

In order to represent the input tree by ETT, we define the function \( rt2et \), short for rose tree to euler tree sequence. Recall the input tree is managed by \([\text{roseTree}]\) in Section 3.2.

\[
\begin{align*}
\text{rt2et} & : (\text{Eq a}) \Rightarrow \text{Tree a} \rightarrow [(a,a)] \\
\text{rt2et} (\text{Node x ts}) &= \text{case ts of} \\
& \quad \text{[]} \quad \rightarrow [(x,x)] \quad \text{-- singleton case [rt2et.1]} \\
& \quad (t’:ts’) \rightarrow \text{root ++} \quad \text{-- tree length > 1 [rt2et.2]} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \text{concat ( map (λt→pref t ++ rt2et t ++ suff t) ts ) -- [rt2et.2]} \\
& \text{where} \\
& \quad \text{pref v} = [(x,\text{rootLabel v})] \quad \text{-- [rt2et.3]} \\
& \quad \text{suff v} = [(\text{rootLabel v},x)] \quad \text{-- [rt2et.4]} \\
& \quad \text{root} = [(x,x)]
\end{align*}
\]

The well-formed ETT sequence is preserved by [rt2et.1] when input tree is a singleton tree, otherwise by [rt2et.2] recursively satisfying the \( \ldots (x,y)(\text{rt2et tree})(y,x) \ldots \) case.

Similar to procedure ET in ETT-HK, \( \text{rt2et} \) traverses the input tree once, returning a Euler-tour sequence in \( O(n) \) where \( n \) is the number of elements in the input tree. Then, we collect the output from \( \text{rt2et} \) and pass it as argument to any of the following helper functions to get a Ft.

- \( \text{foldr} \ (\triangleleft) \text{Empty} \): application of \( \triangleleft \) to the sequence starting from \( \text{Empty} \)
- \( \text{fromList} \): helper function from [48]

4.3.2 FunETT data structure

In Chapter 3 we showed the benefits and features for dealing with sequences through finger trees.

Both ETT-HK and ETT-T assume that the locations for splitting (i.e. cutting) and appending (i.e. linking) the sequences or lists are given apriori, that is, no computation for looking up edges or vertices is performed or expressed in the specification. In our proposal, we make explicit the mechanism for looking up an element within any sequence avoiding extra arguments to be passed onto \( \text{link} \) and \( \text{cut} \). Since finger trees allow monoidal annotations on internal nodes, we take advantage on those annotations by defining the monoid as a BST (i.e. set-like tree), that is, an efficient data structure for look ups and updates. Then, FunETT is a Ft with monoidal annotation \( \text{Data.Set} \) [45] and measurement definition as follows
type FunETT a = FingerTree (Set (a,a)) (a,a)

instance (Ord a) ⇒ Measured (Set (a,a)) (a,a) where
  measure (x,y) = insert (x, y) empty -- [measure.TreeEF]

Set insertion in rule measure.TreeEF above is performed only when an
operation asks for the monoidal annotation of the atomic value in FunETT. That is, a datum on the leaves generates a singleton set of itself at runtime.

The empty-set as the identity for the union operation forms a monoid
over Data.Set, predefined in [45] as

instance Monoid (Set a) where
  mempty = empty -- the empty set
  x <> y = union x y -- union of disjoint sets x and y

From the above, we have a set per internal node in FunETT, with the
largest (just the one at the top of Ft) containing $3n - 2$ pair-elements and
the smallest being the singleton, generated on demand.

Example of FunETT

The following input tree is the same tree from Figure 4.1, but presented here
as star-shaped. We selected the top node to be the root, although this is
arbitrary.

![Diagram of tree](image)

Figure 4.2: Input tree (top left) is Euler-tour numbered (top right). ETT (bottom) is generated by rt2et

By placing the ETT above and inserting it to an Empty Ft element wise,
we have the result in Figure 4.3 below. The dark circles and arrows do not
belong to the data structure, they are placed below the leaves for illustrative purposes only.

![Euler-tour tree representation](image)

Figure 4.3: FUNETT for the Euler-tour tree representation from input tree in Figure 4.2

### 4.3.3 Operations on FUNETT

Recalling the performance of operations over a FT from Chapter 3, in particular the ones analysed through the $<>$ operator, we show a summary in Table 4.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{viewl}$, $\text{viewr}$, $&lt;&gt;$, $\triangleleft$</td>
<td>$\mathcal{O}(&lt;&gt;)$</td>
</tr>
<tr>
<td>$\bowtie$, $\text{split}$, $\text{search}$</td>
<td>$\mathcal{O}(\log n) \times \mathcal{O}(&lt;&gt;)$</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of operations bounds over a generic FT

Now, having defined Data.Set as the monoidal annotation for FUNETT, we have that the set union is the monoidal binary operation, therefore $\mathcal{O}(<>)$ is $\mathcal{O}(m \times (\log(n/m) + 1))$, where $m + n$ is the size of the largest set in the FT and that $m \leq n$. Furthermore, all bounds in Data.Set are worst case, that is, non amortised. In Table 4.2 we show the above substitution.
<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>viewl, viewr, ⊲, ⊳</td>
<td>$O(m \times (\log(n/m) + 1))$</td>
</tr>
<tr>
<td>split, search</td>
<td>$O(\log n) \times O(m \times (\log(n/m) + 1))$</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of $\text{Ft}$ operations applied to $\text{FUNETT}$ when set-union is the monoidal annotation

For the remaining of this thesis, we consider the following equation when $m < n$,

$$O(m \times (\log(n/m) + 1)) = O(m \times \log(n/m))$$

and the following equation when $m = n$,

$$O(m \times (\log(n/m) + 1)) = O(n)$$

Taking the above upper limit bounds, we consider the following analysis in order to get the bounds for cases where experimental results might fall into. Let us consider the base for the logarithmic cases as $b = 10$. Now, running five case analyses for each bound in Table 4.2, that is, $m = \{1, 10, 100, 1000, 10000\}$ and $n = 10000$. Recall that $m$ and $n$ represent the sizes of the sets allocated in the internal nodes of a particular $\text{Ft}$ prior to the application of $\text{mappend}$ or the monoidal binary operation $\star$. We always consider $m \geq 1, n \geq 1$ and $m \leq n$, otherwise $m$ and $n$ are swapped.

We start with the first operations (first row from Table 4.2),

<table>
<thead>
<tr>
<th>$m$</th>
<th>$n$</th>
<th>viewl, viewr, ⊲, ⊳</th>
<th>result</th>
<th>yields to bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10,000</td>
<td>1 ($\log \frac{10,000}{1}$)</td>
<td>4</td>
<td>$\Omega(\log n)$</td>
</tr>
<tr>
<td>10</td>
<td>10,000</td>
<td>10 ($\log \frac{10,000}{10}$)</td>
<td>30</td>
<td>$\Theta(m \log \frac{n}{m})$</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>100 ($\log \frac{10,000}{100}$)</td>
<td>200</td>
<td>$\Theta(m \log \frac{n}{m})$</td>
</tr>
<tr>
<td>1,000</td>
<td>10,000</td>
<td>1,000 ($\log \frac{10,000}{1,000}$)</td>
<td>1,000</td>
<td>$\Theta(m \log \frac{n}{m})$</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>$O(n)$</td>
<td>10,000</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Table 4.3: Bounds of $\text{Ft}$ operations applied to $\text{FUNETT}$ for viewl, viewr, ⊲ and ⊳ cases

Similarly, when applying the same analysis to the second set of operations from Table 4.2, we have,
<table>
<thead>
<tr>
<th>$m$</th>
<th>$n$</th>
<th>result</th>
<th>yields to bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10,000</td>
<td>$\log 10,000 \times 1 \left(\log \frac{10,000}{1}\right)$</td>
<td>$\Omega(\log^2 n)$</td>
</tr>
<tr>
<td>10</td>
<td>10,000</td>
<td>$\log 10,000 \times 10 \left(\log \frac{10,000}{10}\right)$</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>$\log 10,000 \times 100 \left(\log \frac{10,000}{100}\right)$</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>1,000</td>
<td>10,000</td>
<td>$\log 10,000 \times 1,000 \left(\log \frac{10,000}{1,000}\right)$</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>$\log 10,000 \times \mathcal{O}(n)$</td>
<td>$\mathcal{O}(n \log n)$</td>
</tr>
</tbody>
</table>

Table 4.4: Bounds of $F_t$ operations applied to FUNETT for $\bowtie$, split, and search cases

We extend specifications that of ETT-HK for reroot and ETT-T specification for link and cut. The addendum is mostly to include the explicit search, explicit split, and explicit append (via $\bowtie$). Considering further forest operations, we rename the update operations to linkTree and cutTree respectively.

**Cutting a FUNETT**

Let $v$ and $w$ be two vertices and tree be a FUNETT holding a well-formed ETT sequence. Then, following cut in ETT-T, operations search and $\bowtie$ from Chapter 3 and member (testing membership for sets) operation from [45], we have

1. cutTree :: Ord a \Rightarrow a \rightarrow a \rightarrow FunETT a \rightarrow Maybe (FunETT a, FunETT a)
2. cutTree v w tree = if v == w then Nothing else
3. case search pred tree of -- FUNETT: explicit search for $(v, w)$
   4. Position left _ right \rightarrow -- $(v, w)$ found
      -- ETT-T: split $L$ before and after $(v, w)$ so far
      -- $(v, w)$ is discarded by the wildcard _ (1st edge deletion)
   5. case (search pred left) of
      -- FUNETT: explicit search for $(w, v)$ on left subtree
   6. Position leftL _ rightL \rightarrow
      -- FUNETT: $(w, v)$ is on the left subtree
      -- ETT-T: split $L$ before and after $(v, w)$ and $(w, v)$
      -- $(w, v)$ is discarded by the wildcard _ (second edge deletion)
   7. Just (rightL, leftL $\bowtie$ right)
      -- ETT-T: $L^1[(w, v)]L^2[(v, w)]L^3$ (symmetrical case)
Performance of cutTree

The above Haskell script does not have recursive calls on itself, so its traversal takes a single pass. We state only those lines of code (LOC) containing runtimes other than $O(1)$. So, following the bounds stated in Table 4.4 we have

<table>
<thead>
<tr>
<th>LOC</th>
<th>expression</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>case (search pred tree) of</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>5</td>
<td>case (search pred left) of</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>7</td>
<td>Just (rightL, leftL \Join right)</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>9</td>
<td>case (search pred right) of</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>11</td>
<td>Just (leftR, left \Join rightR)</td>
<td>$\Theta(m \log n \log \frac{n}{m})$</td>
</tr>
<tr>
<td>17, 18</td>
<td>(member (v',w')) tree'</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Table 4.5: Bounds of cutTree operation

That is, operation cutTree takes $\Theta(m \log n \log \frac{n}{m})$ time, following the bounds from Table 4.5 above, where $m$ and $n$ are the sizes of the sets (monoidal annotations) involved. However, two exceptions might show up.
Firstly, the case of one of the sets involved in cutTree for all the LOCs is of size one, then the lower bound per operation for cutTree is $\Omega(\log^2 n)$. On the other hand, $O(n \log n)$ is the upper bound per cutTree operation, provided that all sets involved are of the same size. That is, $\Omega(\log^2 n) \leq \Theta(\text{cutTree}) \leq O(n \log n)$ where $n$ is the total number of nodes for the trees in the cutTree operation. These bounds are not longer amortised as the monoidal annotations bounds, via $<>$ operator, are worst case.

A cutTree example

Let tree be a valid FunEtt finger tree representing the input tree in Figure 4.4. Assuming that $\ll$, $\gg$, search and $\gg\ll$ operations are correct by [8], we present the following test-by-hand examples of cutTree application over tree. Recall that a finger tree holds a sequence of data (i.e. pairs for FunEtt) at its leaves. So, if we visually “hide” the internal nodes and branches, tree can be seen as a sequence, rooted at $(1,1)$

$$
tree = [(1,1),(1,2),(2,2),(2,1),(1,3),(3,3),(3,5),(5,5),(5,3),\ldots,(3,1),(1,4),(4,4),(4,1)]
$$

Performing cutTree on tree with edge $(1,3)$ we have

2. $\text{cutTree } 1 \ 3 \ \text{tree} = \text{if } 1 == 3 \ \text{then Nothing else}$
3. case (search pred tree) of
4. Position left _ right → -- underscore represents $(1,3)$

cutTree performs a case analysis, for which there are two cases: the search is successful or unsuccessful. Since search is correct by [8], the pair $(1,3)$ is successfully found at position 4 (0-base index) in tree and omitted for further processing as it is not part of the result. Position data constructor above holds two subtrees,

$$
\text{left} = [(1,1),(1,2),(2,2),(2,1)]
$$
and
\[
\text{right} = [(3,3),(3,5),(5,5),(5,3),(3,1),(1,4),(4,4),(4,1)]
\]
cutTree looks for the mirrored edge, (3,1), in the left subtree.

5. case (search pred2 left) of
6. Position leftL _ rightL \rightarrow -- unsuccessful: (3,1) \notin left subtree
7. Just (rightL, leftL \bowtie right) -- not performed

Since (3,1) is not in the left subtree, the alternative is to look it up in the right subtree (line 9.)

8. _ \rightarrow -- second case for (search pred2 left)
9. case (search pred2 right) of
10. Position leftR _ rightR \rightarrow -- (3,1) found in subtree right
11. Just (leftR, left \bowtie rightR) -- two Frs are returned
12. _ \rightarrow error "ETT malformed" --(1,3) found but not (3,1)
13. _ \rightarrow case (search pred2 tree) of -- 2nd case for (search pred tree)
14. Position _ _ _ \rightarrow error "ETT malformed" --(3,1) found but not (1,3)
15. _ \rightarrow Nothing -- neither (1,3) nor (3,1) \notin tree
16. where
17. pred tree _ = (member (1,3)) tree
18. pred2 tree _ = (member (3,1)) tree

So, it takes \( \mathcal{O}(18) \) lines of non recursive code to perform cutTree. Since finger tree operations are correct by [8], we show for this example that cutTree is also correct and always terminates as it returns either

- **Nothing** for the case input vertices are equal, not the case for this example.
- **Nothing** for the case when neither (1,3) nor (3,1) are in finger tree \( \text{tree} \).
- **error** for the case when pair (1,3) is in finger tree \( \text{tree} \) but (3,1) is not.
- **error** for the case when pair (3,1) is in finger tree \( \text{tree} \) but (1,3) is not.
- **Just** (rightL, leftL \bowtie right), not the case for this example.
- **Just** (leftR, left \bowtie rightR), the result for this example which is

```plaintext
Just( [(3,3),(3,5),(5,5),(5,3)]
    , [(1,1),(1,2),(2,2),(2,1),(1,4),(4,4),(4,1)] )
```

```
|-------------left---------|\infty|--------rightR-----|
```
Figure 4.5: Resulting trees after \texttt{cutTree} operation is applied to input tree in Figure 4.4.

representing the trees in Figure 4.5.

**Linking two FUNETTs**

We follow the Ett-HK approach regarding the reroot operation on one tree, but in our case it shall be on the first tree argument, that is, linking two trees, say \( t_v \) and \( t_w \), means that edge \((v, w)\) connects the rerooted tree \( t_v \) at \( v \) into the no (necessarily) rerooted tree \( t_w \) since it was originally rooted at some vertex \( x \).

Let \( v \) and \( w \) be two vertices, \( tv \) and \( tw \) be two FUNETT holding well-formed Ett sequences. Then, following \textit{link} in Ett-T, operations \texttt{search}, \texttt{split}, \texttt{\textlangle}, \texttt{\textrangle}, \texttt{\texttriangledown} from Chapter 3 and \texttt{member} (testing membership for sets) operation from \cite{[45]}, we have

1. \( \texttt{linkTree} :: \text{Ord } a \Rightarrow a \rightarrow \text{FunETT } a \rightarrow a \rightarrow \text{FunETT } a \rightarrow \text{Maybe } (\text{FunETT } a) \)
2. \( \texttt{linkTree } v \texttt{tv }w \texttt{tw }= \texttt{if } v = = w \texttt{then Nothing else} \)
3. \( \texttt{case } (\texttt{pairIn } (v,v) \texttt{tv }, \texttt{pairIn } (w,w) \texttt{tv }) \texttt{of} \)
4. \( \texttt{ (True, _ ) } \rightarrow \texttt{Nothing } -- v \in tw \)
5. \( \texttt{ ( _, True) } \rightarrow \texttt{Nothing } -- w \in tv \)
6. \( \texttt{ (False,False) } \rightarrow \)
7. \( \texttt{case } (\texttt{pairIn } (v,v) \texttt{tv }, \texttt{pairIn } (w,w) \texttt{tw }) \texttt{of} \)
8. \( \texttt{ (False, _ ) } \rightarrow \texttt{Nothing} \)
9. \( \texttt{ ( _, False) } \rightarrow \texttt{Nothing} \)
10. \( \texttt{ (True , True ) } \rightarrow \texttt{Just } $ \)
11. \( \texttt{let tv’ } = \texttt{reroot tv v} \)
12. \( \texttt{Position left _ right } = \texttt{search pred tw} \)
13. \( \texttt{in } (\texttt{left } \texttt{\textlangle} (w,w) \texttt{\textrangle } (w,v)) \texttt{\texttriangledown tv’ } \texttt{\texttriangledown ((v,w) } < \texttt{right) } \)
   -- in \texttt{FunETT} we append \( L^2_1, [(v,v)], [(v,w)] , t'_v , [(w,v)], L^2_2 \)
   -- in \texttt{Ett-T} we append \( L^2_1, L^1_1, [(v,w)], L^2_2, L^1_2, [(w,v)] \)
14. \( \texttt{where} \)
15. \( \texttt{pred tree _ } = \texttt{(member } (w,w)) \texttt{tree }-- \texttt{is } (w,w) \in \texttt{tree} ? \)
Function \texttt{pairIn} test membership for both vertices \(v\) and \(w\) rather than \texttt{split} or \texttt{search} on/for them.

1. \texttt{pairIn :: (Ord} \(a\) \texttt{)} \(\Rightarrow\) \(a\) \(\rightarrow\) FunETT \(a\) \(\rightarrow\) Bool
2. \texttt{pairIn} \(v\) \(\texttt{pair tree} = \texttt{case} \ \texttt{(search} \ \texttt{pred} \ \texttt{tree}) \ \texttt{of}
3. \ \ \ Position \_ \_ \_ \texttt{=} \texttt{True}
4. \_ \_ \_ \texttt{=} \texttt{False}
5. \texttt{where}
6. \texttt{pred tree'} \_ = (\texttt{member} \ \texttt{pair}) \ \texttt{tree'}

Finally, changing the root on a FunETT is as follows

1. \texttt{reroot :: Ord} \(a\) \(\Rightarrow\) FunETT \(a\) \(\rightarrow\) \(a\) \(\rightarrow\) FunETT \(a\)
2. \texttt{reroot} \(v\) \texttt{vertex} =
3. \texttt{let} (\texttt{left}, \texttt{right}) = \texttt{split} \ \texttt{pred} \ \texttt{tree}
4. \texttt{in} \texttt{right} \texttt{▷ left} \texttt{-- as in ETT-T when swapping } \texttt{L}_1^1, \texttt{L}_1^2 \texttt{to} \texttt{L}_1^2, \texttt{L}_1^1
5. \texttt{where}
6. \texttt{pred tree'} \_ = (\texttt{member} (\texttt{vertex}, \texttt{vertex})) \ \texttt{tree'}

**Performance of \texttt{linkTree}**

Alike \texttt{cutTree}, operation \texttt{linkTree} does not have recursive calls on itself, so its traversal takes a single pass. We state only those lines of code where runtimes differ from \(O(1)\). So, following the bounds stated in Table 4.4 we have

<table>
<thead>
<tr>
<th>LOC</th>
<th>function</th>
<th>expression</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>\texttt{pairIn}</td>
<td>\texttt{case (search pred tree) of}</td>
<td>(\Theta(m \log n \log \frac{n}{m}))</td>
</tr>
<tr>
<td>6</td>
<td>\texttt{pairIn}</td>
<td>\texttt{(member pair) tree'}</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>3</td>
<td>\texttt{reroot}</td>
<td>\texttt{split pred tree}</td>
<td>(\Theta(m \log n \log \frac{n}{m}))</td>
</tr>
<tr>
<td>4</td>
<td>\texttt{reroot}</td>
<td>\texttt{right ▷ left}</td>
<td>(\Theta(m \log n \log \frac{n}{m}))</td>
</tr>
<tr>
<td>6</td>
<td>\texttt{reroot}</td>
<td>\texttt{(member (vertex,vertex)) tree'}</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>12</td>
<td>\texttt{linkTree}</td>
<td>\texttt{search pred tw}</td>
<td>(\Theta(m \log n \log \frac{n}{m}))</td>
</tr>
<tr>
<td>13</td>
<td>\texttt{linkTree}</td>
<td>\texttt{((left ▷ (v,v)) ▷ (v’,w’)) ▷ tv’ ▷ ((v’,w’) ▷ right)}</td>
<td>(\Theta(m \log n \log \frac{n}{m}))</td>
</tr>
<tr>
<td>15</td>
<td>\texttt{linkTree}</td>
<td>\texttt{(member (v,v)) tree}</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

Table 4.6: Bounds of \texttt{linkTree} operation
That is, operation linkTree takes $\Theta(m \log n \log \frac{n}{m})$ time, following the bounds from Table 4.6 above, where $m$ and $n$ are the sizes of the sets (monoidal annotations) involved. However, two exceptions might arise. Firstly, the case of one of the sets involved in linkTree for all the LOCs is of size one, then the lower bound per linkTree operation is $\Omega(\log^2 n)$. Secondly, $\mathcal{O}(n \log n)$ is the upper bound for each linkTree operation, provided that all sets involved are of the same size. That is, $\Omega(\log^2 n) \leq \Theta(\text{linkTree}) \leq \mathcal{O}(n \log n)$ where $n$ is the total number of nodes for the trees in the linkTree operation. These bounds are not longer amortised as the monoidal annotations bounds, via $<>$ operator, are worst case.

A linkTree example

Let $tv$ and $tw$ be a valid FunEtt finger trees representing the input trees in Figure 4.5. Assuming that $<$, $>$, search and $\triangleright\triangleright$ operations are correct by [8], we present the following test-by-hand examples of linkTree application over $tv$ and $tw$. So, if we visually “hide” the internal nodes and branches, $tv$ and $tw$ can be seen as the following sequences, rooted at $(3,3)$ and $(1,1)$ respectively.

$$tv = [(3,3),(3,5),(5,5),(5,3)]$$
$$tw = [(1,1),(1,2),(2,2),(2,1),(1,4),(4,4),(4,1)]$$

Failed case of linkTree

Let $v=2$ and $w=4$, then linkTree over $tv$ and $tw$ is shown below.

\begin{verbatim}
2. linkTree 2 tv 4 tw = if 2 == 4 then Nothing else
3. case (pairIn (2,2) tw, pairIn (4,4) tv) of
4.   (True, _ ) → Nothing -- (2,2) ∈ tw
... Case analysis in line (4.) above, returns Nothing as $v=2 \in tw$. This is confirmed as True
by the left pairIn (2,2) tw which performs the following snippet.
2. pairIn (2,2) tw = case (search pred tw) of
3.   Position _ _ _ → True
4.   _ _ → False
5. where
6.   pred tree’ _ = (member (2,2)) tree’
\end{verbatim}

Successful case of linkTree

Let $v=3$ and $w=1$, then linkTree over $tv$ and $tw$ is shown below.
2. linkTree 3 tv 1 tw = if 3 == 1 then Nothing else
3. case (pairIn (3,3) tw, pairIn (1,1) tv) of
4. (True, _ ) → Nothing
5. (_ , True) → Nothing
6. (False, False) →
7. case (pairIn (3,3) tv, pairIn (1,1) tw) of
8. (False, _ ) → Nothing
9. (_ , False) → Nothing
10. (True , True ) → Just $
11. let tv' = reroot tv 3
12. Position left _ right = search pred tw
13. in ((left ⊿ (1,1)) ⊿ (1,3)) ⊿ tv' ⊿ ((3,1) ⊿ right)
14. where
15. pred tree _ = (member (1,1)) tree

After testing all the constraints against pairIn (lines 3-10), subtrees tv' (line 11), left and right (line 12) assemble the finger tree (line 13) to be return by linkTree. tv' is the resulting tree after applying reroot to tv and to vertex 3. in line 11. Following snippet describes this process.

2. reroot tv 3 =
3. let (left,right) = split pred tv
4. in right ⊿◁ left
5. where
6. pred tree' = (member (3,3)) tree'

Trees left=[]> and right=[(3,3),(3,5),(5,5),(5,3)] are merged (≿) and returned by reroot after applying split to predicate pred and tree tv. That is, tv' == tv as tv is already rooted at node 3.

So, it takes $O(15)$ lines of non recursive code to perform linkTree. Since finger tree operations are correct by [8], we show for this example that linkTree is also correct and always terminates as it returns either

- **Nothing**, for the case input vertices are equal, not the case for this example.
- **Nothing**, for the cases when none of the input vertices are in their corresponding trees.
- $$((left▷(1,1))▷(1,3))≿tv'≿ ((3,1)◁ right)$$, the result for this example which is

```
Just( [(1,1), (1,3), (3,3),(3,5),(5,5),(5,3), (3,1)
[]▷ (1,1)▷ (1,3)▷ --------tv'--------▷ ((3,1)◁
, (1,2),(2,2),(2,1),(1,4),(4,4),(4,1)]
----------right----------)
```

representing the tree in Figure 4.6.

83
Figure 4.6: Resulting tree after linkTree operation is applied to input trees in Figure 4.5.

4.4 Chapter Notes

This chapter contains original work and is based on the author’s paper presented at [25]. Some insights we contribute to the specifications for linking and cutting trees that of Henzinger and King [5] and that of Tarjan [7] are listed as follows,

- Pointing to the vertices as arguments to both link and cut operations are now defined explicitly in the specification. We achieve this through search by extending the analysis of [8] in Chapter 3 and its implementation in FunEtt data structure.

- The specification is actually the implementation into the Haskell programming language. This is, our belief, that FunEtt is the first declarative and purely functional programming approach to deal with the dynamic trees problem main operations altogether with its data structure. Furthermore, the present work is extended in order to include the forest structure in Chapter 5 based on FunEtt.

- A comparison between the operations used in each specification is provided below

<table>
<thead>
<tr>
<th>Operation, description</th>
<th>Ett-HK</th>
<th>Ett-T</th>
<th>FunEtt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space for ETT representation</td>
<td>$2n - 1$</td>
<td>$3n - 2$</td>
<td>$3n - 2$</td>
</tr>
<tr>
<td>cut look ups</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>cut splits</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>cut append</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>link look ups</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>link splits</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>link append</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Total operations in specification</td>
<td>16</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 4.7: Specifications of the dynamic tree operations link and cut
Chapter 5

Indexless data structures

Specifications Ett-HK and Ett-T, described in Chapter 4, do not mention nor describe a forest data structure, which is the container for the trees managed by operations link and cut. In order to avoid collisions with operations names, we shall call the above operations as linkTree and cutTree respectively.

In this chapter we extend FunEtt in order to manage link and cut operations over a forest structure, specifically two new alternatives are defined. We present FULL dynamic trees in Section 5.1, a Ft with all of its monoidal annotations storing sets. Then, in Section 5.2 we adjust the original Ft data structure, devised by Hinze and Paterson in [8] in order to reduce the allocation of monoidal annotations in the spine of the Ft. We call this structure Top dynamic trees. Additionally, we conclude with a comparison between FULL and Top dynamic trees. We describe the main functions and data types for both data structures leaving the helper functions and smart constructors accessible at [50].

5.1 FULL dynamic trees

We present a data structure for dealing with the case when for every monoidal annotation in a Ft we store a set in the form of a binary search tree (BST), specifically the one described in Section 3.3. Commencing with the data types in Section 5.1.1 we describe the monoidal annotation, leaves, trees and forest data structures. Then, we move towards the operations over the above data types in Section 5.1.2, in particular, connected, cut and link altogether
their runtime bounds. Thirdly, we present the results of the experimental analysis carried out on Full dynamic trees in Section 5.1.3.

5.1.1 Full dynamic trees data types

Edges and vertices can be managed under the same BST data structure for the linearisation case of dynamic trees, as implicitly stated in both ETT-HK and ETT-T specifications. However, we devise the storage of edges and vertices separately, that is, one BST for edges and one BST for vertices. Furthermore, we split the BST for vertices when such values are integers.

```haskell
data MultiSet a = MultiSet {getEvens :: S.Set a, 
getOdds :: S.Set a, 
getEdges :: S.Set (a,a)}
```

Recall that type `Set` is imported from `Data.Set` and prefixed here with `S`. in order to avoid conflicts with namespaces between `Data.FingerTree` and `Data.Set`. Although splitting the monoidal annotation into three different BSTs does not reduce the complexity stated in Table 3.1, we are actually cutting the height of the `S.Set` s by a factor of \(n\). From the total number of pairs, \(3n-2\), representing the ETT sequence, \(n\) is the number of vertices which is cut even further by isolating the vertices into odds and evens, provided the type for `S.Set` is `Int`, otherwise we define just one `S.Set` for vertices.

We define the type of the Ft leaves as follows,

```haskell
newtype MSet a = MSet (a,a)
```

and the tree as

```haskell
type TreeMSet a = FingerTree (MultiSet a) (MSet a)
```

where `FingerTree` is the data type defined at `Data.FingerTree` and previously explained in Section 3.6. Finally, our forest data type is another Ft having trees of type `TreeMSet a` as leaves and augmented with two integers, `NumNodes` and `ForestSize`. These numeric values are useful when asking whether the forest is one-tree forest (i.e. no more edges can be added) or is a unit forest (i.e. no edges at all).

```haskell
data ForestMSet a =
    ForestMSet NumNodes ForestSize (FingerTree (MultiSet a)(TreeMSet a))
```

Reduction of the size of the sets in Full data structure is possible by relaxing the uniqueness of the edges in the leaves of Ft. We store the edge \(\{v,w\}\) from the input tree as \((\min v w, \max v w)\) in the Full trees and
forests, one per traversal. As an example, Figure 5.1 highlights in thick borders the two (repeated) edges in ETT.

![Figure 5.1: Full Ft with repeated edges identified with thicker border for data on the leaves and underlined and coloured for edges on sets](image)

Recall that the total number of edges is $2n - 2$ out of $3n - 2$ total pairs in the ETT. Since we are about to store just $n$, half of number of edges, we are reducing the current total number of pairs for up to $1/3$ in the Full Ft.

Figure 5.2 illustrates the reduction of edges in comparison to the example in Figure 5.1. Notice, however, that the size of the ETT sequence remains as $3n - 2$.

![Figure 5.2: Full Ft with repeated edges not longer stored in the corresponding sets, however, they remain in the ETT sequence](image)

In order to define the monoidal annotation information of a Full dynamic tree, the following Measurement instance is stated.

```haskell
instance (Integral a, Ord a) ⇒ Measured (MultiSet a) (MSet a) where
```
\[ \text{measure (MSet } p) = \text{whichSet } p \]

\[
\text{whichSet } p@0(x,y)
\begin{align*}
| x &= y \&\& \text{even } x & = \text{MultiSet } (S.\text{insert } x S.\text{empty}) S.\text{empty} S.\text{empty} \\
| x &= y \&\& \text{odd } x & = \text{MultiSet } S.\text{empty} (S.\text{insert } x S.\text{empty}) S.\text{empty} \\
| \text{otherwise} & = \text{MultiSet } S.\text{empty} S.\text{empty} (S.\text{insert } p S.\text{empty})
\end{align*}
\]

That is, given a pair \( p \) (i.e. a vertex or an edge) wrapped in the leaf data constructor \texttt{MSet}, function \texttt{whichSet} returns the corresponding initial monoidal annotation which is then lifted up within \texttt{FT} by the \(<>\) operator. Notice that type argument \( a \) of the leaf \texttt{MSet} is constrained to be numeric by type class \texttt{Integral}.

### 5.1.2 Full dynamic trees operations

The root of a tree, \texttt{rootMSet}, receives a \texttt{FT} as input and returns its left-most vertex (i.e. the first element in the pair) from such a \texttt{FT}. In case an empty tree is passed by, \texttt{Nothing} is return.

\[
\begin{align*}
\text{rootMSet} &: \texttt{Integral } a \Rightarrow \texttt{TreeMSet } a \rightarrow \texttt{Maybe } a \\
\text{rootMSet } \texttt{tree} &= \texttt{case viewl } \texttt{tree of} \\
\texttt{EmptyL} & \rightarrow \texttt{Nothing} \\
\texttt{MSet } x :< \_ & \rightarrow \texttt{Just } (\texttt{fst } x)
\end{align*}
\]

Since \texttt{viewl} is the same as the one stated in Section 4.3.3, its runtime is \( \Omega(\log n) \leq \Theta(\text{rootMSet}) \leq O(n) \) where \( n \) is the total number of vertices of the tree where \texttt{viewl} takes place. Notice that \texttt{rootMSet} is defined to be applied to trees only.

Auxiliary functions \texttt{searchMSet} and \texttt{nodeInMSet} test whether a vertex is in a given \texttt{FULL} forest. The former relies on the monoidal annotation allocated at the top of the spine of the \texttt{FT} provided.

\[
\begin{align*}
\text{searchMSet} &: \texttt{(Integral } a, \texttt{Measured } (\texttt{MultiSet } a) b) \Rightarrow \texttt{(a,a)} \rightarrow \texttt{FingerTree } (\texttt{MultiSet } a) b \rightarrow \texttt{SearchResult } (\texttt{MultiSet } a) b \\
\text{searchMSet } p@0(x,y) \texttt{ftree}
\begin{align*}
| x &= y \&\& \text{even } x & = \texttt{let predicate } setx _ \\
| \quad = (S.\text{member } x) (\texttt{getEvens } setx) \texttt{in search predicate } ftree \\
| x &= y \&\& \text{odd } x & = \texttt{let predicate } setx _ \\
| \quad = (S.\text{member } x) (\texttt{getOdds } setx) \texttt{in search predicate } ftree \\
| \quad \text{otherwise} & = \texttt{let predicate } setx _ \\
| \quad \quad = (S.\text{member } p) (\texttt{getEdges } setx) \texttt{in search predicate } ftree
\end{align*}
\]

Function \texttt{search} within \texttt{searchMSet} is the same function described in Section 3.6.7. Therefore, performance for \texttt{searchMSet} is \( \Omega(\log^2 n) \leq \Theta(}
\texttt{searchMSet} \leq O(n \log n) \text{ where } n \text{ is the total number of vertices of the forest where } \texttt{searchMSet} \text{ takes place.}

Result of the following function \texttt{nodeInMSet} is either \texttt{Nothing} (i.e. vertex not in the forest) or the pair (tree, root of the tree).

\begin{verbatim}
nodeInMSet :: Integral a \Rightarrow a \rightarrow ForestMSet a \rightarrow Maybe (TreeMSet a, a)
nodeInMSet v (ForestMSet _ _ ft) =
  case (searchMSet (v,v) ft) of
    Position _ tree _ \rightarrow
      case (rootMSet tree) of
        Nothing \rightarrow
          Nothing -- empty tree, no root
        Just rootT \rightarrow
          Just (tree, rootT)
    _ \rightarrow
      Nothing -- vertex v not in forest
\end{verbatim}

Taking into account both \texttt{searchMSet} and \texttt{rootMSet} runtimes described above, we get that \(\Omega(\log^2 n) \leq \Theta(\texttt{nodeInMSet}) \leq O(n \log n)\) where \(n\) is the total number of vertices of the forest where \texttt{nodeInMSet} is applied to.

\textbf{connected operation in \textsc{full} dynamic trees}

Testing connectivity in \textsc{full} dynamic trees is via operation \texttt{connectedMSet}, taking two vertices and a forest and returning the pair comprising a boolean and the trees altogether their roots. Connectivity is not simply looking for the edge comprising the vertices but for the vertices living under the same tree, that said, we compare the roots of the trees for the input vertices being tested.

\begin{verbatim}
connectedMSet :: Integral a \Rightarrow a \rightarrow a \rightarrow ForestMSet a
  \rightarrow (Bool, Maybe (TreeMSet a, a, TreeMSet a, a))
connectedMSet x y f =
  case (nodeInMSet x f, nodeInMSet y f) of
    (Nothing, _ ) \rightarrow (False, Nothing)
    (_ , Nothing) \rightarrow (False, Nothing)
    (Just (tx,rx), Just (ty,ry)) \rightarrow
      if rx == ry then (True, Just(tx,rx,tx,rx))
              else (False, Just(tx,rx,ty,ry))
\end{verbatim}

Following performance from \texttt{nodeInMSet}, we have that \texttt{connectedMSet} lower and upper bounds are \(\Omega(\log^2 n)\) and \(O(n \log n)\) respectively, where \(n\) is the total number of vertices in the forest where \texttt{connectedMSet} is applied to.
A connected example

Let $forest$ be the collection of $t_1$, $t_7$, $t_8$ and $t_9$ input trees, depicted in Figure 5.3. Following is the $forest$ monoidal annotation.

\[
\text{MultiSet} \{ \text{getEvens} = \text{fromList} \{2,4,6,8\}, \\
\text{getOdds} = \text{fromList} \{1,3,5,7,9\}, \\
\text{getEdges} = \text{fromList} \{(1,2),(1,3),(1,4),(2,1),(3,1),(3,6), \\
(3,8),(4,1),(5,6),(6,3),(6,5),(6,7), \\
(7,6),(8,3),(8,9),(9,8)\} \}
\]

Let vertex 7 be searched via function $\text{nodeInMSet}$.

\[
\text{nodeInMSet } 7 \ (\text{ForestMSet } \_ \_ \ forest) = \\
\text{case } (\text{searchMSet } \{7,7\} \ forest) \text{ of} \\
\quad \text{Position } \_ \ \text{tree } \_ \rightarrow \\
\quad \text{case } (\text{rootMSet } \text{tree}) \text{ of} \\
\quad \quad \text{Nothing } \rightarrow \text{Nothing} \\
\quad \quad \text{Just } \text{rootT } \rightarrow \text{Just } (\text{tree}, \text{rootT}) \\
\quad \_ \rightarrow \text{Nothing}
\]

We assume function $\text{searchMSet}$ as correct as it relies on function $\text{search}$ from [8]. Since 7 is a member of $forest$ monoidal annotation, line (7) above matches the case analysis and that returns the pair $(\text{tree}, \text{rootT})$, which for the case of vertex 7 is $(t_7,7)$. For testing connectivity, we try vertices 7 and 8 within $forest$.

\[
\text{connectedMSet } 7 \ 8 \ forest = \\
\text{case } (\text{nodeInMSet } 7 \ forest, \ \text{nodeInMSet } 8 \ forest) \text{ of} \\
\quad \text{Nothing } \_ \rightarrow (\text{False}, \text{Nothing}) \\
\quad \_ \text{Nothing } \rightarrow (\text{False}, \text{Nothing}) \\
\quad \text{Just } (t_7,7) \text{ Nothing } \rightarrow (\text{False}, \text{Nothing}) \\
\quad \_ \text{Just } (t_8,8) \rightarrow \\
\quad \quad \text{if } 7 == 8 \text{ then } (\text{True}, \ \text{Just}(tx,rx,tx,rx)) \\
\quad \quad \text{else } (\text{False}, \ \text{Just}(t_7,7,t_8,8))
\]

After both $\text{nodeInMSet}$ functions are evaluated (line 2 above), the pair $\text{Just } (t_7,7), \ \text{Just } (t_8,8)$ matches the case analysis. Since roots 7 and
8 are different, \texttt{connectedMSet} returns the pair \((\text{False}, \text{Just}(t_7,7,t_8,8))\). Assume now we are testing connectivity for vertices 2 and 4. All of the above is satisfied except the last bit where \texttt{connectedMSet} returns the pair \((\text{True}, \text{Just}(t_1,1,t_1,1))\).

\textbf{link operation in FULL dynamic trees}

Update dynamic operations for \texttt{FULL} are described through \texttt{link} and \texttt{cut}. When any of the latter operations fail, we return the forest provided as input, otherwise a new version of the previous forest is returned.

\texttt{link} :: Integral a \Rightarrow a \rightarrow a \rightarrow \text{ForestMSet} a \rightarrow \text{ForestMSet} a

\texttt{link} \ x \  y \ \text{forest}@((\text{ForestMSet} \ n\text{nodes} \ \text{size} \ ft)) =

\begin{align*}
\text{case} \ \text{connectedMSet} \ x \ y \ \text{forest} \ \text{of}
& \quad \text{(False, Just(tx,rx,ty,ry))} \rightarrow \text{linkAll} \ (\text{linkTreeMSet} \ x \ tx \ y \ ty) \\
& \quad \quad \rightarrow \text{forest} \quad \text{-- x and y are currently connected in forest}
\end{align*}

\text{where}

\begin{align*}
\text{linkAll} \ \text{tree} & = \text{ForestMSet} \ n\text{nodes} \ \text{(size} + 1) \ \text{(tree} \ \triangleleft \ (l\text{forest} \triangleright \! \! \! \! \! \! \text{rforest})) \\
\text{Position} \ l\text{forest'} \ _ \ r\text{forest'} & = \text{searchMSet} \ (x,x) \ ft \quad \text{-- tx is left behind} \\
\text{Position} \ l\text{forest} \ _ \ r\text{forest} & = \text{searchMSet} \ (y,y) \ (l\text{forest'} \triangleright \! \! \! \! \! \text{rforest'}) \quad \text{--ty is left behind}
\end{align*}

A new tree \((\text{tree})\) is built from \texttt{linkTreeMSet} with input trees \((tx \text{ and } ty)\) provided by \texttt{connectedMSet} once this function tested positive for \(x \text{ and } y\) being connected at \texttt{forest}. The new forest is built via function \texttt{linkAll} which has left behind trees \(tx \text{ and } ty\). New tree is inserted with \(\triangleleft\) to the new subforest \((l\text{forest} \triangleright \! \! \! \! \! \text{rforest})\). Finally, the size of the forest is updated when \((\text{size} + 1)\).

Unless the above \texttt{forest} is \texttt{one-tree} forest, runtime complexity for \texttt{link} is determined by local and global monoidal annotations. That is, \texttt{linkTreeMSet} runs in \(\Theta(m \log n \log \frac{n^2}{m})\) where \(m\) and \(n\) are the sizes of the sets evaluated on \(tx \text{ and } ty\) trees, whereas the sets evaluated in \texttt{connectedMSet}, \texttt{searchMSet}, \(\triangleleft\), and \(\triangleright\) are regarded to the entire \texttt{forest}. Now, since \texttt{link} has a constant number of operations, the runtime for \texttt{link} is \(\Omega(\log^2 n) \leq \Theta(\text{link}) \leq \mathcal{O}(n \log n)\), where \(n\) is the total amount of nodes in the forest, not at the trees when applying \texttt{linkTreeMSet}.

\textbf{A link example}

Let \texttt{forest} be the collection of \(t_1, t_7, t_8\) and \(t_9\) input trees, depicted in Figure 5.3. Assume we try to \texttt{link} \(2 \ 4\) in \texttt{forest}. 

91
2. \textbf{link} 2 4 \textbf{forest@}(\textbf{ForestMSet} \textit{nnodes} \textit{size} \textit{ft}) =
3. \hspace{1em} \text{case connectedMSet 2 4 \textit{forest} of}
4. \hspace{1em} (\text{False, Just}(\textit{tx},\textit{rx},\textit{ty},\textit{ry})) \rightarrow \textbf{linkAll} (\textbf{linkTreeMSet} \times \textit{tx} \textit{ty})
5. \hspace{1em} _ \rightarrow \textit{forest}

Case analysis, in line 5, returns _, which is the lazy-evaluated form for (\text{True, Just}(\textit{t1},1,\textit{t1},1)). It is not necessary to express such a form completely as the \textbf{link} is failed due both vertices coexist under the same tree.

Now, we try \textbf{link} 8 7 in \textit{forest}.

2. \textbf{link} 8 7 \textbf{forest@}(\textbf{ForestMSet} \textit{nnodes} \textit{size} \textit{ft}) =
3. \hspace{1em} \text{case connectedMSet 8 7 \textit{forest} of}
4. \hspace{1em} (\text{False, Just}(\textit{t8},8,\textit{t7},7)) \rightarrow \textbf{linkAll} (\textbf{linkTreeMSet} 8 \textit{t8} 7 \textit{t8})

Since vertices 7 and 8 are members of different trees, line 4 in snippet above matches the case analysis and \textbf{linkAll} function is called after performing \textbf{linkTreeMSet}. We showed up in Section 4.3.3 that \textbf{linkTree} returns a valid finger tree, called here \textit{tree}, provided input trees and vertices are also valid.

Then, \textbf{linkAll} assembles the output for \textbf{link} as follows.

7. \textbf{linkAll} \textit{tree} = \textbf{ForestMSet} \textit{nnodes} (\textit{size} + 1) (\textit{tree} \leftarrow (\textit{lforest} \Join \textit{rforest}))

By using \textbf{searchMSet}, \textit{t7} and \textit{t8} are discarded from \textit{forest} yielding \textit{lforest} to be \textit{t9} and \textit{rforest} to be \textit{t1}. Finally, a new version of \textit{forest} is built (line 7) preserving the same number of nodes (recall those are fixed), incrementing its size by one and pushing \textit{tree} (i.e. linked \textit{t8} and \textit{t7}) into the merged \textit{t9} and \textit{t1}. The overall process is illustrated in Figure 5.4.

\textbf{cut operation in Full dynamic trees}

When deleting an edge from a forest, there is no need for the edge to be tested by \textbf{connectedMSet}, instead, we define \textbf{edgeInMSet} as the helper function for
cut.

dgeInMSet edge (ForestMSet _ _ ft) =
  case (searchEdgeMSet edge ft) of
  Position left tree right → Just (tree,left,right)
  _ → Nothing -- edge not in forest

Similar to nodeInMSet, bounds for edgeInMSet is \(\Omega(\log^2 n) \leq \Theta(\text{edgeInMSet}) \leq O(n \log n)\), where \(n\) is the total amount of nodes in the forest.

Alike link, a failed computation for cut returns the input (intact) forest, otherwise a new forest with size decreased by one is returned as the result.

cutMSet :: Integral a ⇒ a → a → ForestMSet a → ForestMSet a

cutMSet x y forest@(ForestMSet nnodes size ft) =
  case edgeInMSet (x,y) forest of
  Nothing → forest -- edge not in forest
  Just (tree,ltFor,rtFor) → buildForest (cutTreeMSet x y tree) ltFor rtFor

where
buildForest (leftTree,rightTree) lFor rFor
= ForestMSet nnodes (size - 1)
  (leftTree ◁ rightTree ◁ (lFor ⊲◁ rFor))

After the tree is found by edgeInMSet, provided the vertices \(x\) and \(y\), a pair of subforests (lFor,rFor) are also returned, in particular the ones not containing tree. Then, cutTreeMSet splits tree into two trees (leftTree, rightTree) after deleting edge \((x,y)\). The new forest to return by cut is built up by inserting the trees leftTree and rightTree into the merged subforests (lFor ⊲◁ rFor).

Performance for cut is calculated under the same context of link, that is, local and global monoidal annotations. That said, we have \(\Omega(\log^2 n) \leq \Theta(\text{cut}) \leq O(n \log n)\), where \(n\) is the total amount of nodes in the forest.

A cut example

Let forest be the collection of \(t_1, t_7\) and \(t_9\) input trees, depicted in Figure 5.4. Assume we try to cut 2 4 in forest. Although there is a path between vertices 2 and 4, there is not an edge between them. So, cut 2 4 forest returns forest (same as input) as cut is unsuccessful. This is shown below.

2. cutMSet 2 4 forest@(ForestMSet nnodes size ft) =
3. case edgeInMSet (2,4) forest of
4. Nothing → forest
We assume search correct by [8], then edgeInMSet evaluates to Nothing in the case analysis above. Now, applying cut 7 6 into the forest from Figure 5.4, we have the following snippet.

2. \text{cutMSet} 7 6 \text{forest}@(\text{ForestMSet} \text{nnodes} \text{size} \text{ft}) =
3. \text{case edgeInMSet (7,6) forest of}
4.  \text{Nothing} \rightarrow \text{forest}
5.  \text{Just (t7,ltFor,rtFor)} \rightarrow \text{buildForest (cutTreeMSet 7 6 t7) ltFor rtFor}
7.  \text{where}
8.  \text{buildForest (leftTree,rightTree) ltFor rtFor}
9.  = \text{ForestMSet} \text{nnodes} \text{(size - 1)}
10. \quad (\text{leftTree} \bowtie \text{rightTree} \bowtie (\text{ltFor} \bowtie \text{rtFor}))

\text{Just (t7,ltFor,rtFor)} matches edgeInMSet case analysis in lines 3 and 5 above. \text{ltFor} and \text{rtFor} are the resulting subforests after the search for edge (7,6) (and its mirrored (6,7)) within edgeInMSet. As we showed up in Section 4.3.3, calling (cutTreeMSet 7 6 t7), line 6) returns a valid pairs of trees as long as the vertices and forest provided are also valid. The valid pairs of trees in this example are \text{leftTree} and \text{rightTree}. Finally, the two trees and two sub forests are assembled, in line 10, as a new version of forest, pictured in Figure 5.5, altogether with the same number of nodes as before and decreasing its forest size by one.

**Summary of FULL operations performance**

Based in the previous analyses per FULL dynamic tree operations and from the performance stated at Table 4.2, we have in Table 5.1 the summary of bounds per operation, worst case (non amortised).
Table 5.1: Performance of the FULL dynamic tree operations, where \( n \) is the number of nodes in the FULL forest.

5.1.3 Experimental analysis of FULL dynamic trees

Our aim is to benchmark FULL dynamic trees described in this chapter, Section 5.1, when implemented in Haskell for the construction of a forest, its updates and its queries. We initially describe the experimental setup followed by the description for each benchmark operation.

Experimental Setup

All benchmarks were performed on a dedicated machine on a 2.2 GHz Intel Core i7 MacBook Pro with 16 GB 1600 MHz DDR3 running macOS High Sierra version 10.13.1 (17B1003). We imported the following libraries into our code from the online package repository Hackage: `Data.FingerTree` [48], code for finger trees, and `Data.Set` [45] for conventional sets. We used the R programming environment [51] for plotting. The time consumed per function was taken from the machine internal clock via `Data.Time.Clock` library [52].

The tree structure, forest structure, update and query operations upon the structures were implemented by the author in Haskell and compiled with `ghc` version 8.0.1 with optimisation `-O2`. Full source code and all graphs with the numerical data are available on the author’s repository in GitHub [50].

The running time of a given computation was determined by the mean and the median of thirty executions. In figs. 5.6 to 5.8 we show a sample of plottings for different input data (initial, intermediate and final). The size of input data is displayed on the \( x \) axis. The input elements, pairs of edges and vertices, are of type `Int`, that is, `((Int,Int))` where only positive values evaluated. For all implementations of data types we have seen, types are polymorphic which must be an instance of `Ord`, and some of the operations constrained to `Measured`.

<table>
<thead>
<tr>
<th>Operation</th>
<th>best case</th>
<th>worst case</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>( \Omega(\log n) )</td>
<td>( \mathcal{O}(n) )</td>
<td>trees</td>
</tr>
<tr>
<td>connected</td>
<td>( \Omega(\log^2 n) )</td>
<td>( \mathcal{O}(n \log n) )</td>
<td>forest</td>
</tr>
<tr>
<td>cut</td>
<td>( \Omega(\log^2 n) )</td>
<td>( \mathcal{O}(n \log n) )</td>
<td>forest</td>
</tr>
<tr>
<td>link</td>
<td>( \Omega(\log^2 n) )</td>
<td>( \mathcal{O}(n \log n) )</td>
<td>forest</td>
</tr>
</tbody>
</table>
Figure 5.6: Sample of plotting for an initial running

Figure 5.7: Sample of plotting for an intermediate running
In order to distinguish the results between the plottings from the means to those from the medians we shall show the sampling for means every 10 points and every 100 points for medians. We tried ascending order to fully test the data structure behaviour. The minimum plotting point is 11 and the maximum is 3011. All query and update dynamic operations in one way or another use auxiliary functions such as random lists of pairs. In all cases, these auxiliary functions are left out from the performance. The interested reader can find the source code for those auxiliary functions in module RndDynTs in [50].

Finally, in order to make a forest data structure available for the dynamic operations we focus our implementations to be strict evaluated. That is, when asking for connectivity or performing either link or cut, the host forest is fully evaluated and not just partially. The main reason for that is that both cut and link call edgeIn and connected respectively. So, if laziness is the expression evaluator, an update operation needs to wait for the look up to assembly all the sets in the forest and then find (or not) the pair in matter. This, adds an unwanted cost to the update operation.

**Full forests construction**

All query and update dynamic operations rely on a Full forest created in advanced. In this section we shall show the performance when building Full
forests under different tree sizes, specifically, for unit, 2-node, 10-node and 300-node forests.

**unit forest construction**

Since a unit forest contains only singleton trees, we practically insert Single \((x, x)\) trees straightforward into an Empty Full forest. The following snippet is illustrative only, details can be found in RndDynTs.

\[
\text{forest} = \text{foldr} (\triangleleft) \text{emptyForestMSet} (\text{map} (\lambda x \rightarrow \text{Single} (\text{MSet}(x, x))) \text{nodes})
\]

where nodes is a list of random pair values. Generation of nodes is not taken in the performance for Full forest creation.

The performance for unit Full forest creation is shown in figs. 5.9 and 5.10.

![Performance of Creating (Unit) Full Forests](image)

**Figure 5.9:** Performance of unit Full forest measured by means.
Figure 5.10: Performance of unit FULL forest measured by medians

Experimental results of unit forest construction

At each point in the curve, every 10 for Figure 5.9 and every 100 for Figure 5.10, a FULL forest is created from Empty. The runtime cost is due the operator, then result is in between $\Omega(\log n)$ and $O(n)$ per point as expected, where $n$ is the number of vertices in the forest. An excerpt of tabular values for Figure 5.10 is presented in Table 5.2.

<table>
<thead>
<tr>
<th>number of nodes</th>
<th>unit runtime in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0.012</td>
</tr>
<tr>
<td>111</td>
<td>0.148</td>
</tr>
<tr>
<td>211</td>
<td>0.391</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>17.531</td>
</tr>
<tr>
<td>2,911</td>
<td>18.059</td>
</tr>
<tr>
<td>3,011</td>
<td>18.216</td>
</tr>
</tbody>
</table>

Table 5.2: An excerpt of tabular values for Figure 5.10.
**2-node forest construction**

Similar to the *unit* forest, the *2-node* forest is constructed by just two insertion into and empty tree and then into an empty forest. Performance for *2-node* FULL forest creation is shown in figs. 5.11 and 5.12.

![Performance of Creating (2-node) Full Forests](image)

Figure 5.11: Performance of *2-node* FULL forest measured by means
Figure 5.12: Performance of 2-node Full forest measured by medians

**Experimental results of 2-node forest construction**

Performing the insertion of 2-node trees into Full forests involves two times the $\triangleleft$ operator, leading to almost twice of the running time with respect to its counterpart unit Full forests. That is, at 3011 nodes, running time for unit is (18.382 milliseconds) vs (33.797 milliseconds) for 2-node. Performance for this construction of Full forests is at least $\Omega(\log n)$ and at most $O(n)$, where $n$ is the number of nodes in the Full forest. An excerpt of tabular values for Figure 5.12 is shown in Table 5.3.
Table 5.3: An excerpt of tabular values for Figure 5.12.

<table>
<thead>
<tr>
<th>number of nodes</th>
<th>2-node runtime in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0.068</td>
</tr>
<tr>
<td>111</td>
<td>0.626</td>
</tr>
<tr>
<td>211</td>
<td>1.213</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>30.747</td>
</tr>
<tr>
<td>2,911</td>
<td>32.139</td>
</tr>
<tr>
<td>3,011</td>
<td>33.621</td>
</tr>
</tbody>
</table>

**10-node and 300-node forest construction**

The factor of constant growth in 10-node and 300-node FULL forests with respect to unit and 2-node is not necessarily proportional. That is, having 10 and 300 \( \triangleleft \) operations per tree respectively, does not increment the growth to 30 times the former w.r.t. the latter. This is because the larger the trees the less tree-inhabitants per forest, therefore the height of the forest decreases and the number of monoidal annotations in its affixes also reduces. We can see the performance in figs. 5.13 to 5.16.
Figure 5.13: Performance of 10-node Full forest measured by means

Figure 5.14: Performance of 10-node Full forest measured by medians
Figure 5.15: Performance of 300-node FULL forest measured by means

Figure 5.16: Performance of 300-node FULL forest measured by medians
Experimental results of 10-node and 300-node Full forest construction

The runtime at 3011 nodes between 10-node (figs. 5.13 and 5.14) and 300-node (figs. 5.15 and 5.16) forests is not around 30 times as expected (79.405 milliseconds vs 843.699 milliseconds). Recall that a Full forest is a finger tree having finger trees (i.e. Full trees) at its leaves. Then, constructing a Full forest out of 3011 nodes yields to a space to allocate 11 300-node trees (10 trees of 300 nodes each plus 1 tree of 11 nodes) or 302 10-node trees (301 trees of 10 nodes each plus 1 tree of 1 node). In one hand, the \(\triangleleft\) operation is executed more times in constructing a 300-node forest than in a 10-node forest. On the other hand, the 300-node forest has smaller height w.r.t. the 10-node forest and less amount of monoidal annotations in its affixes. Performance for both Full forest constructions is at least \(\Omega(\log n)\) and at most \(O(n)\), \(n\) being the number of nodes in the Full forest, with a constant factor above 10 between 10-node and 300-node Full forests. The bumpy (outliers) behaviour of the curve in figs. 5.13 and 5.15 is explained in the next section as it is more evident. An excerpt of tabular values for figs. 5.14 and 5.16 is shown in Table 5.4.

<table>
<thead>
<tr>
<th>number of nodes</th>
<th>10-node runtime in milliseconds</th>
<th>300-node runtime in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0.250</td>
<td>NA</td>
</tr>
<tr>
<td>111</td>
<td>2.343</td>
<td>NA</td>
</tr>
<tr>
<td>211</td>
<td>4.520</td>
<td>NA</td>
</tr>
<tr>
<td>311</td>
<td>6.776</td>
<td>84.895</td>
</tr>
<tr>
<td>411</td>
<td>8.981</td>
<td>91.979</td>
</tr>
<tr>
<td>511</td>
<td>11.608</td>
<td>125.420</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>73.647</td>
<td>780.995</td>
</tr>
<tr>
<td>2,911</td>
<td>76.194</td>
<td>809.462</td>
</tr>
<tr>
<td>3,011</td>
<td>79.092</td>
<td>849.095</td>
</tr>
</tbody>
</table>

Table 5.4: An excerpt of tabular values for figs. 5.14 and 5.16.

Full forests construction, a summary

In Figure 5.17 we present all the above Full forests construction performances in a single chart so it can be visualised their differences.
Performance of Creating Full Forests

![Graph showing performance of different numbers of nodes in Full forests construction](image)

Figure 5.17: Performance of 300-node, 10-node, 2-node, and unit Full forests construction, measured by medians.

**Experimental results of the Full forests construction**

By putting all Full forests construction performances together in the same chart, we can appreciate their growth differences. unit forest being the fastest and 300-node forest the slowest. This is expected as the more insertions (#) per forest the more time consuming. Nevertheless there is a significant gap between the slowest and fastest performances, the runtime growth for each Full forest construction remains within the lower bound of \( \Omega(\log n) \) and the upper bound of \( O(n) \), where \( n \) is the number of nodes in the Full forest. An excerpt of tabular values for Figure 5.17 is shown in Table 5.5.
<table>
<thead>
<tr>
<th>number of nodes</th>
<th>unit runtime in milliseconds</th>
<th>2-node runtime in milliseconds</th>
<th>10-node runtime in milliseconds</th>
<th>300-node runtime in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0.012</td>
<td>0.068</td>
<td>0.250</td>
<td>NA</td>
</tr>
<tr>
<td>111</td>
<td>0.148</td>
<td>0.626</td>
<td>2.343</td>
<td>NA</td>
</tr>
<tr>
<td>211</td>
<td>0.391</td>
<td>1.213</td>
<td>4.520</td>
<td>NA</td>
</tr>
<tr>
<td>311</td>
<td>0.651</td>
<td>1.857</td>
<td>6.776</td>
<td>84.895</td>
</tr>
<tr>
<td>411</td>
<td>0.876</td>
<td>2.427</td>
<td>8.981</td>
<td>91.979</td>
</tr>
<tr>
<td>511</td>
<td>1.155</td>
<td>3.228</td>
<td>11.608</td>
<td>125.420</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>17.531</td>
<td>30.747</td>
<td>73.647</td>
<td>780.995</td>
</tr>
<tr>
<td>2,911</td>
<td>18.059</td>
<td>32.139</td>
<td>76.194</td>
<td>809.462</td>
</tr>
<tr>
<td>3,011</td>
<td>18.216</td>
<td>33.621</td>
<td>79.092</td>
<td>849.095</td>
</tr>
</tbody>
</table>

Table 5.5: An excerpt of tabular values for Figure 5.17.

**connectivity in Full forests**

In order to benchmark connectivity, we not simply look for the edge being in a specific Full forest but to apply the function `connectedMSet` to such a forest. In this way, we are considering the following:

- Vertices $x$ and $y$ belong to the Full forest, by performing `nodeInMSet`.
- Roots of the trees for vertices $x$ and $y$ are equal, therefore there is connectivity or
- Roots of the trees for vertices $x$ and $y$ are different, therefore there is no path between $x$ and $y$.

We run `connectedMSet` over five different Full forests,

1. **10-node** forest under 500 runs,
2. **10-node** forest under 1,000 runs,
3. **300-node** forest under 300 runs,
4. **300-node** forest under 500 runs, and
5. **300-node** forest under 1,000 runs

107
Figure 5.18: Performance of connectedMSet over a 10-node Full forest, measured by means.

**Connectivity in 10-node Full forests**

We commence with connectedMSet ==False test for a 10-node forest after a thousand runs, showing its performance in Figure 5.18.

**Experimental results of Connectivity in 10-node Full forests**

We approach the runtime bounds when applying connectedMSet to a Full forest. So, given the following from Section 5.1.2, \( \Omega(\log^2 n) \leq \Theta(\text{connectedMSet}) \leq O(n \log n) \), where \( n \) is the number of nodes in the forest, the curve in Figure 5.18 is traced within the lower and upper bounds, being 1.131 milliseconds for its performance at 11 nodes and 857.321 milliseconds the performance at 3011 nodes.

**Outliers**

Recall that function measure described in (3.5) and (3.6), in Chapter 3, applies the \( \mathcal{O}(<>\rangle) \) at runtime (on-demand), that is, the set union is applied only at the Digit data constructors as the monoidal annotations for the Node data...
constructor were calculated when inserting (« and ») and when appending («▷»). Similarly, the monoidal annotations at the spine are calculated also in advanced.

In order to determine the reasons for which performance of connectedMSet has outliers, we show the affixes of the finger tree behind Full forest. Then, for every Digit data constructor we have a different number of set-unions to be calculated. So, for One there is no set union computation, for Two there is just one computation, for Three there are two computations and finally for Four there are three computations.

Then, in Table 5.6 we show the most notorious cases before, on and after when the outliers occur and referring them to the number of set union computations in the affixes. For practicality, we represent with Roman numeral the Digit data constructor, as I for One, II for Two and so on. The following acronyms are used in such a table: nnodes stands for “number of nodes (x-axis)”, and nops stands for “number of set union operations in the affixes”.

<table>
<thead>
<tr>
<th>row</th>
<th>nnodes</th>
<th>runtime</th>
<th>△</th>
<th>forest</th>
<th>nops: prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>631</td>
<td>180.268</td>
<td>0.678</td>
<td>[III,IV,IV],Empty,[I,I,I]</td>
<td>3+3+3 = 9</td>
</tr>
<tr>
<td>2</td>
<td>641</td>
<td>188.810</td>
<td>8.542</td>
<td>[IV,IV,IV],Empty,[I,I,I]</td>
<td>3+3+3 = 9</td>
</tr>
<tr>
<td>3</td>
<td>651</td>
<td>141.160</td>
<td>-47.650</td>
<td>[II,II,II],Single,[I,I,I]</td>
<td>1+1+1 = 3</td>
</tr>
<tr>
<td>4</td>
<td>661</td>
<td>147.557</td>
<td>6.397</td>
<td>[III,II,II],Single,[I,I,I]</td>
<td>2+1+1 = 4</td>
</tr>
<tr>
<td>5</td>
<td>671</td>
<td>156.129</td>
<td>8.571</td>
<td>[IV,II,II],Single,[I,I,I]</td>
<td>3+1+1 = 5</td>
</tr>
<tr>
<td>6</td>
<td>1981</td>
<td>653.749</td>
<td>8.775</td>
<td>[III,IV,IV,IV],Empty,[I,I,I,I]</td>
<td>2+3+3+3 = 11</td>
</tr>
<tr>
<td>7</td>
<td>1991</td>
<td>672.368</td>
<td>18.618</td>
<td>[IV,IV,IV,IV],Empty,[I,I,I,I]</td>
<td>3+3+3+3 = 12</td>
</tr>
<tr>
<td>8</td>
<td>2001</td>
<td>489.392</td>
<td>-182.975</td>
<td>[II,II,II,II],Single,[I,I,I,I]</td>
<td>1+1+1+1 = 4</td>
</tr>
<tr>
<td>10</td>
<td>2021</td>
<td>506.374</td>
<td>2.615</td>
<td>[IV,II,II,II],Single,[I,I,I,I]</td>
<td>3+1+1+1 = 6</td>
</tr>
<tr>
<td>11</td>
<td>2791</td>
<td>864.816</td>
<td>8.924</td>
<td>[III,IV,IV,IV],Single,[I,I,I,I]</td>
<td>2+3+3+3 = 11</td>
</tr>
<tr>
<td>12</td>
<td>2801</td>
<td>877.551</td>
<td>47.633</td>
<td>[IV,IV,IV,IV],Single,[I,I,I,I]</td>
<td>3+3+3+3 = 12</td>
</tr>
<tr>
<td>13</td>
<td>2811</td>
<td>674.967</td>
<td>-202.583</td>
<td>[II,II,II,II,II],Empty,[I,I,I,I,I]</td>
<td>1+1+1+1+0 = 4</td>
</tr>
<tr>
<td>14</td>
<td>2821</td>
<td>722.600</td>
<td>12.734</td>
<td>[III,II,II,II,II],Empty,[I,I,I,I,I]</td>
<td>2+1+1+1+0 = 5</td>
</tr>
<tr>
<td>15</td>
<td>2831</td>
<td>731.525</td>
<td>25.534</td>
<td>[IV,II,II,II,II],Empty,[I,I,I,I,I]</td>
<td>3+1+1+1+0 = 6</td>
</tr>
</tbody>
</table>

Table 5.6: Amount of monoidal annotations in a Full dynamic tree via its affixes

Column forest from Table 5.6 indicates the affixes at height $i$ of the forest in matter in the format $[p_0,p_1,\ldots],bt,[s_0,s_1,\ldots]$, where $p_i$ is the prefix at height $i$, $s_i$ is the suffix at height $i$, $bt$ is the bottom of the spine (Empty or Single) and $i \in \{0,1,2,\ldots\}$. So, in row 1 we have [Three,Four,Four], Empty, [One,One,One]. Following the differences, △, between runtimes from Table 5.6 we notice that the negative values match the outliers in the curve from Figure 5.18. Furthermore, the larger the number of nodes the higher
the bump in such a curve. This is because the size of the sets at deeper height in the finger tree behind the Full forest is larger, yielding to set union to process larger sets. So, the number of monoidal annotations for the prefix (3.8, taking away the suffix and the spine) is

\[ \sum_{i=1}^{h} \left( 4 \sum_{j=1}^{i} (2 \times 3^{j-1}) \right) \]

Let us take for instance rows 12 and 13 from Table 5.6: just at height 3, row 12 contains 108 set union operations whereas at the same height, row 13 has 36 set union operations.

Now, in Figure 5.19 we present the performances of the 10-node Full forests for both 500 and 1,000 runs when sampling is 100 medians.

![Performance of Connectivity on (10-node) Full Forests](Figure 5.19: Performance of connectedMSet over a 10-node Full forest, multiple runs.

Regarding the 300-node Full forests, we present in Figure 5.20 the case when connectedMSet == False and the function is executed 1,000 times, every execution with a different (random) pair of nodes.
Figure 5.20: Performance of connectedMSet over a 300-node Full forest.

Experimental results of testing connectivity in a 300-node Full forest

We follow the outliers analysis from Figure 5.18. This time, the stairs behaviour of the curve in Figure 5.20 is due to the application of connectedMSet to an apparently repeated tree within the Full forest. That is, every step shape in the curve is comprised with practically the same tree as is has 300 nodes. So, for instance, the first step in the curve (nodes 301 upto 601) has a singleton 300-node tree. Then, the second step has two 300-node trees, and so forth. Table 5.7 show the “first” outlier, between nodes 1491 and 1501.

<table>
<thead>
<tr>
<th>row</th>
<th>modes</th>
<th>runtime</th>
<th>Δ</th>
<th>forest</th>
<th>nops: prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1481</td>
<td>166.530</td>
<td>2.1390</td>
<td>[IV], Empty, [I]</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1491</td>
<td>165.958</td>
<td>-0.571</td>
<td>[IV], Empty, [I]</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1501</td>
<td>127.873</td>
<td>-38.085</td>
<td>[I,I], Empty, [I,I]</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1511</td>
<td>131.063</td>
<td>3.190</td>
<td>[I,I], Empty, [I,I]</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1521</td>
<td>127.134</td>
<td>-3.929</td>
<td>[I,I], Empty, [I,I]</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.7: Amount of monoidal annotations in a Full dynamic tree via its affixes for a Full forest.
The negative value of $\Delta$ in row 3 from Table 5.7 indicates the outlier between number of nodes (x-axis) 1491 and 1501. The other negative values (rows 2 and 5) are off the computation of connectedMSet. Figure 5.21 shows the outlier and the other negative $\Delta$s.

![Performance of case analysis for outliers in Connectivity 300-node Full forest](image)

**Figure 5.21:** Performance of connectedMSet over a 300-node FULL forest, showing one of the outliers.

Finally, in Figure 5.22 the performance for 10-node and 300-nodes FULL forest testing connectivity is shown.
Performance of Connectivity on Full Forests

Figure 5.22: Performance of `connectedMSet` over a 10-node and 300-node Full forest, for different number of runs per forest.

**Experimental results of connectivity between 10-node and 300-node Full forests**

Figure 5.22 shows the gaps between the different n-node Full forests when testing connectivity. 300-node forests outperform 10-node forests when running `connectedMSet`. Such a difference relies on the number of monoidal annotations performed by the `<>` operation, that is, a 10-node forest has more trees in its leaves than a 300-node forest. Having more leaves per forest implies having more monoidal annotations through the data structure, as the height is larger. However, for all the cases plotted, performance lies within the bounds that of `connectedMSet`, which are $\Omega(\log^2 n)$ the lower bound and $O(n \log n)$ the upper one, where $n$ is the number of nodes in the forest.

An excerpt of tabular values for Figure 5.22 is shown in Table 5.8 where acronyms 10-n stands for 10-node, 300-n stands for 300-nodes. The integer next each acronym is the number of runs.
**linking trees in Full forests**

For this experiment, we apply only `link` operations over a Full forest until the maximum number of edges in the forest is reached, that is, from unit to one-tree forests, from 2-node to one-tree forests, from 10-node to one-tree forests and from 300-node to one-tree forests.

For all of the cases, the input vertices to the `link` operation are provided by a random list which is not taken into account in the performance. Such a list is defined with the aim that for every pair in the list, the corresponding `link` is successful. So, consuming the list of pairs implies that the initial Full forest becomes a Full one-tree forest.

In figs. 5.23 and 5.24 the above process is shown.

![Table 5.8: An excerpt of tabular values for Figure 5.22.](image_url)
Figure 5.23: Performance of link operation over a unit, 2-node, 10-node and 300-node Full forests, sampling every 10 means.
Figure 5.24: Performance of link operation over a unit, 2-node, 10-node and 300-node Full forests, sampling every 100 medians.

**Experimental analysis for Full link**

Plottings from figs. 5.23 and 5.24 when link is applied over Full forests outputs different performances as shown below. Since every forest is set to reach the maximum number or pairs in the ETT, the maximum number of monoidal annotations is also reached, then the bound for link is $O(n \log n)$. Then, for every point plotted, $n$ nodes are computed, hence the total performance in each curve is $O(n \times n(\log n)) = O(n^2 \log n)$. However, there are constant factors between the $n$-node forests.

- **unit** performs the link operation on up to 3001 trees.
- **2-node** performs the link operation on up to 1500 trees, taking half of the time taken by the unit forest.
- **10-node** performs the link operation on up to 300 trees, which is about 10 times faster than the unit forest.
- **300-node** performs the link operation on up to 10 trees, which is about 300 times faster than unit forest.
In this experiment the number of link applications is limited to the number of leaves per forest. For instance, just 11 link operations on a 300-node forest or up to 300 link operations on a 10-node forest were applied. Furthermore, measuring the performance for a specific link operation is not accurate as each plotting point, such a operation is applied to a dynamic growth in the size of the host forest. For instance, in the unit forest case, the first link is applied to an edge-empty forest, whereas by the end of the sequence of links, the forest is practically edge-full. On the other hand, the height of the host forest for the former case is $O(\log n)$, i.e. 3011 leaves, and the height for the latter case is zero, since it is the Single finger tree. We shall see in Section 5.1.3 how this is sorted out.

An excerpt of tabular values for Figure 5.24 is shown in Table 5.9.

<table>
<thead>
<tr>
<th>number of nodes</th>
<th>unit runtime (millisec.)</th>
<th>2-n runtime (millisec.)</th>
<th>10-n runtime (millisec.)</th>
<th>300-n runtime (millisec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0.206</td>
<td>0.103</td>
<td>0.026</td>
<td>NA</td>
</tr>
<tr>
<td>111</td>
<td>10.671</td>
<td>4.803</td>
<td>0.873</td>
<td>NA</td>
</tr>
<tr>
<td>211</td>
<td>37.087</td>
<td>18.081</td>
<td>3.179</td>
<td>NA</td>
</tr>
<tr>
<td>311</td>
<td>84.722</td>
<td>39.884</td>
<td>6.417</td>
<td>0.083</td>
</tr>
<tr>
<td>411</td>
<td>146.215</td>
<td>70.997</td>
<td>12.166</td>
<td>0.087</td>
</tr>
<tr>
<td>511</td>
<td>230.657</td>
<td>104.620</td>
<td>17.973</td>
<td>0.133</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>9,123.865</td>
<td>4,299.726</td>
<td>740.431</td>
<td>13.165</td>
</tr>
<tr>
<td>2,911</td>
<td>9,929.812</td>
<td>4,714.005</td>
<td>846.462</td>
<td>14.165</td>
</tr>
<tr>
<td>3,011</td>
<td>10,483.036</td>
<td>5,076.541</td>
<td>898.199</td>
<td>16.021</td>
</tr>
</tbody>
</table>

Table 5.9: An excerpt of tabular values for Figure 5.24.

cutting trees in Full forests

Sort of opposite to link operation, in this experiment we apply the cut operation to Full forests in order to reduce the forest size from one-tree downto unit, from 300-node downto unit, from 10-node downto unit, and from 2-node downto unit.

For all of the cases, the input vertices to the cut operation are provided by a random list which is not taken into account in the performance. Such a list is defined in advance in such a way that every pair in the list allows the application of cut successful. So, consuming the list of pairs implies that the initial Full forest becomes a Full unit forest.
In figs. 5.26 and 5.40 we show the results for the experiment described above.

Figure 5.25: Performance of cut operation over a one-tree, 2-node, 10-node and 300-node Full forests, sampling every 10 means.
Figure 5.26: Performance of cut operation over a one-tree, 2-node, 10-node and 300-node Full forests, sampling every 100 medians.

Experimental analysis for cut

We commence the analysis comparing the curves 300-node vs one-tree from figs. 5.26 and 5.40. The size of the set in the top monoidal annotation of the FT for one-tree forest has the maximum number of pairs for the ETT, that is, $3n - 2$. The size of the set in the top monoidal annotation for 300-node forest is $3n - 2 - 10$ as it has just 10 edges away to be a one-tree. Although the forest size of one-tree forest is slightly larger than the 300-node forest, the performance of the latter is slightly slower. This is because the data structure of the host finger tree is actually larger as it holds 10 subtrees 300-node each (plus one small of eleven nodes). On the other hand, the finger tree data structure for the one-tree forest is Single. Recall, from tables 5.6 and 5.7, that the analysis on the affixes shows up that having larger affixes slows down the performance. This the case between one-tree and 300-node forests when applying cut. Regarding 2-node curve, the host finger tree has half of the $n$ nodes as leaves making the structure the largest in between the participants in this experiment. Nevertheless, for all curves depicted in figs. 5.26 and 5.40 the bounds for cutting trees in FULL forests are delimited.
by

- Cost of $\Omega(\log^2 n) \leq (\text{cut}) \leq \mathcal{O}(n \log n)$, where $n$ is the number of nodes in the Full forest.

- The curve applies $n$ nodes per plotting point resulting in the performance of $\mathcal{O}(n^2 \log n)$, $n$ being the number of nodes in the Full forest.

Again, this experiment poses some drawbacks. The number of cut operations is limited to the number of edges in the host forest. The measurement of the performance of a specific cut operation is not accurate as the forest size decreases during the sequence of cuts progresses. In the following section we show our proposal to measure the performance per cut operation. An excerpt of tabular values for Figure 5.26 is presented in Table 5.10.

<table>
<thead>
<tr>
<th>number of nodes</th>
<th>one-tree runtime</th>
<th>2-n runtime</th>
<th>10-n runtime</th>
<th>300-n runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>millsec.</td>
<td>millsec.</td>
<td>millsec.</td>
<td>millsec.</td>
</tr>
<tr>
<td>11</td>
<td>0.178</td>
<td>0.063</td>
<td>0.142</td>
<td>NA</td>
</tr>
<tr>
<td>111</td>
<td>6.171</td>
<td>1.398</td>
<td>4.153</td>
<td>NA</td>
</tr>
<tr>
<td>211</td>
<td>17.064</td>
<td>3.856</td>
<td>13.055</td>
<td>NA</td>
</tr>
<tr>
<td>311</td>
<td>34.801</td>
<td>7.393</td>
<td>28.049</td>
<td>35.889</td>
</tr>
<tr>
<td>411</td>
<td>61.111</td>
<td>11.939</td>
<td>47.084</td>
<td>61.739</td>
</tr>
<tr>
<td>511</td>
<td>92.828</td>
<td>17.224</td>
<td>69.754</td>
<td>94.237</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>3,187.991</td>
<td>445.086</td>
<td>2,303.065</td>
<td>3,262.757</td>
</tr>
<tr>
<td>2,911</td>
<td>3,329.041</td>
<td>481.762</td>
<td>2,395.523</td>
<td>3,463.268</td>
</tr>
<tr>
<td>3,011</td>
<td>3,660.431</td>
<td>510.081</td>
<td>2,584.284</td>
<td>3,823.798</td>
</tr>
</tbody>
</table>

Table 5.10: An excerpt of tabular values for Figure 5.26.

**linking and cutting trees in Full forests**

The aim of this experiment attempts to measure the performance of each dynamic update on Full forests. In order to do so, we look after the size of the forest after every operation. So, after each link operation we apply a cut operation and vice versa. Furthermore, we care also for the tree sizes, that is, the random list of vertices for linking and cutting is build in such a way that

- the tree size of the resulting tree after linking is not greater than twice the $n$-node forest. For instance, in a 300-node forest, the size of the resulting tree is not greater than 600 nodes.
• the tree size of any of the two resulting trees after cutting is not smaller than the quarter of the $n$-node forest.

Additionally, we run the $n$-node forest with different sizes for the input random list. The idea is to show the proportion between the results after such runnings. The random list of vertices is generated prior to the application of the dynamic updates and its runtime is not taken into the account of the performance when linking and cutting. At each plotting point we execute $i$ times the random list of vertices for link-cut over a 10-node forest and $j$ times for a 300-node forest, where $i \in \{500, 1000\}$ and $j \in \{300, 500, 1000\}$.

Plotting in Figure 5.27 shows the performance of the experiment described above.

![Performance of Linking & Cutting Full Forests](image)

Figure 5.27: Performance of link and cut operations over 10-node and 300-node Full forests, sampling every 100 medians.

**Experimental results for link-cut**

The growth for all 300-node curves in Figure 5.27 outrun the 10-node curves. While the size of the Full forest remains practically the same after each operation, the number of monoidal annotations per forest varies. Recall the
total number monoidal annotations in a \( F_t \) is

\[
2 \sum_{i=1}^{h} \left( 4 \sum_{j=1}^{i} 3^{j-1} \right) + h + 1
\]

Then, being \( h = 1 \) the height for a 300-node forest and \( h = 3 \) the height for a 10-node forest, we have that former holds up to 10 monoidal annotations and the latter 130. Even though the amount of monoidal annotations is larger in a 300-node tree in comparison with a 10-node tree, the monoidal annotations at the forest finger tree take more time to be computed as all the leaves are evaluated at certain point when linking or cutting. An excerpt of tabular values for Figure 5.27 is presented in Table 5.11.

<table>
<thead>
<tr>
<th>num. nodes</th>
<th>10-n 500</th>
<th>10-n 1K</th>
<th>300-n 300</th>
<th>300-n 500</th>
<th>300-n 1K</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12.308</td>
<td>24.079</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>111</td>
<td>42.962</td>
<td>85.524</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>211</td>
<td>77.989</td>
<td>152.207</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>311</td>
<td>117.056</td>
<td>231.254</td>
<td>48.005</td>
<td>82.320</td>
<td></td>
</tr>
<tr>
<td>411</td>
<td>161.479</td>
<td>318.589</td>
<td>41.609</td>
<td>68.053</td>
<td>122.978</td>
</tr>
<tr>
<td>511</td>
<td>199.922</td>
<td>396.193</td>
<td>73.593</td>
<td>144.137</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>1,518.818</td>
<td>3,049.764</td>
<td>424.455</td>
<td>700.562</td>
<td>1,331.922</td>
</tr>
<tr>
<td>2,911</td>
<td>1,563.308</td>
<td>3,129.308</td>
<td>424.914</td>
<td>698.088</td>
<td>1,393.413</td>
</tr>
<tr>
<td>3,011</td>
<td>1,636.775</td>
<td>3,258.576</td>
<td>483.003</td>
<td>820.563</td>
<td>1,517.262</td>
</tr>
</tbody>
</table>

Table 5.11: An excerpt of tabular values for Figure 5.27.

**Cost per link-cut operation**

We run two experiments in order to see the performance of the link and cut operations individually. Firstly, we take the total amount of performance from previous experiment (link-cut over Full forests) and divided it by the number of runs, i.e. \( i \) times for 10-node and \( j \) times for 300-node forests. Secondly, we run incrementally each operation over an evolving 10-node forest with 3011 nodes. Since the forest data structure is purely functional (i.e. immutable), we get a different forest per application of the link and cut. In the second and third charts we get straight forward the performance per operation as we increment the number of operations rather than the number of the nodes. We appreciate both experiments in figs. 5.28 to 5.30.
Figure 5.28: Performance of individual link and cut operations over 10-node and 300-node forests.

An excerpt of tabular values for Figure 5.28 is presented in Table 5.12.

<table>
<thead>
<tr>
<th>num. nodes</th>
<th>10-n 500 µ sec.</th>
<th>10-n 1K µ sec.</th>
<th>300-n 300 µ sec.</th>
<th>300-n 500 µ sec.</th>
<th>300-n 1K µ sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>24.616</td>
<td>24.079</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>111</td>
<td>85.925</td>
<td>85.524</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>211</td>
<td>155.978</td>
<td>152.207</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>311</td>
<td>234.112</td>
<td>231.254</td>
<td>103.090</td>
<td>96.01</td>
<td>82.319</td>
</tr>
<tr>
<td>411</td>
<td>322.957</td>
<td>318.589</td>
<td>138.696</td>
<td>136.106</td>
<td>122.978</td>
</tr>
<tr>
<td>511</td>
<td>399.846</td>
<td>396.193</td>
<td>161.156</td>
<td>147.187</td>
<td>144.137</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,811</td>
<td>3,037.636</td>
<td>3,049.764</td>
<td>1,414.850</td>
<td>1,401.124</td>
<td>1,331.922</td>
</tr>
<tr>
<td>2,911</td>
<td>3,126.616</td>
<td>3,129.307</td>
<td>1,416.383</td>
<td>1,396.175</td>
<td>1,393.413</td>
</tr>
<tr>
<td>3,011</td>
<td>3,273.550</td>
<td>3,258.576</td>
<td>1,610.009</td>
<td>1,641.127</td>
<td>1,517.262</td>
</tr>
</tbody>
</table>

Table 5.12: An excerpt of tabular values for Figure 5.28.
Figure 5.29: Performance of individual link and cut operations over a 10-node forest having 3011 leaves.
Figure 5.30: Performance of individual link and cut operations over a 300-node forest having 3011 leaves.

**Experimental results per individual link-cut operation**

Figure 5.28 shows the performance per operation when the input is the number of nodes per forest, not the number of operations over a single forest. Curves in figs. 5.29 and 5.30 show the sublinear behaviour when performing link-cut individually over a specific n-node forest and specific forest size. The linear reference is given by taking the first plotting input and multiplying it by the number of runs the operation is applied. A sample of the plotted values in Figure 5.30 is shown in Table 5.13.
Table 5.13: Plotting values when performing link-cut over a 10-node forest with 3,011 leaves, Figure 5.30

### 5.2 Top dynamic trees

Our Full dynamic tree data structure, in Section 5.1, relies on the data types and operations those of the finger tree by Hinze and Paterson [8] with no augmentations, other than defining Data.Set as the monoidal annotation. In this, and the following sections we attempt to reduce the amount of monoidal annotations in such a finger tree. Our analysis start by identifying the scope of monoidal annotations there are in a finger tree. We depict this in Figure 5.31. The finger tree data structure itself is not altered. However, our new dynamic tree proposal, called Top, is also a finger tree wrapped with a slightly different data constructors than those in our Full proposal. We describe the appropriate data types in Section 5.2.1. Then, we define the changes to some of the finger tree functions that reflect the reduction in the amount of the monoidal annotations in Section 5.2.2.

<table>
<thead>
<tr>
<th>n\text{th} input</th>
<th>runtime of the link-cut mean in µ seconds</th>
<th>runtime of 1\text{st} mean × n\text{th} input in µ seconds (linear reference)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.584</td>
<td>5.584</td>
</tr>
<tr>
<td>11</td>
<td>35.406</td>
<td>61.424</td>
</tr>
<tr>
<td>21</td>
<td>69.757</td>
<td>117.264</td>
</tr>
<tr>
<td>971</td>
<td>3,460.134</td>
<td>5,422.064</td>
</tr>
<tr>
<td>981</td>
<td>3,487.708</td>
<td>5,477.904</td>
</tr>
<tr>
<td>991</td>
<td>3,502.422</td>
<td>5,533.744</td>
</tr>
</tbody>
</table>
5.2.1 Top dynamic trees data types

We notice that global accumulators (i.e. monoidal annotations in the spine of the Ft) are a repeated version of the sets in the corresponding affixes in the Ft. Even more, that repeated version implies that two (or one when at the bottom of the spine) set union operations (<>s) are performed from the largest subsets at height $k$ of the Ft. Avoiding the computation of <> at the spine of the Ft, on the other hand, loses information needed when computing a Ft as a leaf in the forest. Our proposal to overcome this is via two strategies:

Top$_1$ Searching in a Ft is limited to the affixes, then we devise a new function definition for search.

Top$_2$ We maintain a monoidal annotation outside the Ft data structure (for both trees and forest) which allocates the same information as the one monoidal annotation at the top of the original Ft. This is illustrated in figs. 5.32 and 5.33.

Figure 5.31: Accumulators identified in a Ft data structure.
Figure 5.32: $F_T$ data structure for Top finger trees, top accumulator.

Figure 5.33: $F_T$ data structure for Top finger trees, general view.

Strategy $Top_1$ is described in the following section as search is a finger
tree operation. Now, the data types for the TOP tree and forest are defined below. We define the monoidal annotation for TOP finger trees exactly the same as in FULL finger trees (see Section 5.1). The pairs of vertices and edges are stored in Leaf data constructor.

newtype Leaf a = Leaf (a,a) -- pair for edges and vertices

A TOP tree is a Ft of Leafs and monoidal annotation Data.Set. Additionally, it comprises an external (to the Ft) non-empty monoidal annotation.

data TreeTop a
 = TreeTop
   (MultiSet a)  -- top monoidal annotation
   (FingerTree (MultiSet a)(Leaf a)) -- finger tree of pairs

A TOP forest is a Ft of TreeTop trees and monoidal annotation Data.Set. Additionally, it comprises an external non-empty monoidal annotation. The number of nodes and the size of the TOP forest are left out its data type and is calculated by the functions nnodesForest and sizeForest respectively, defined below.

data ForestTop a
 = ForestTop
   (MultiSet a)  -- top monoidal annotation
   (FingerTree (MultiSet a)(TreeTop a)) -- finger tree of TreeTop trees

nnodesForest :: ForestTop a → Int
nnodesForest (ForestTop _ Empty) = 0
nnodesForest (ForestTop (MultiSet _ _ _ _ _ _ _ odds_ _) _ ) = S.size evens + S.size odds

sizeForest :: ForestTop a → Int
sizeForest (ForestTop _ Empty) = 0
sizeForest forest@(ForestTop (MultiSet _ _ edges _ _ ) _ )
   = S.size edges + nnodesForest forest

Notice that determining the size of the forest or its number of nodes takes \( O(1) \) as it just patterns match its TOP monoidal set.

5.2.2 TOP dynamic trees operations

We commence with the smart constructor deep that assembles a Ft by passing to it the affixes and a subtree. By pairing up one function from FULL and one from TOP we can spot their differences.

depth prefix middle suffix -- FULL deep version
Recalling the FingerTree data type

data FingerTree v a
  = Empty
  | Single a
  | Deep
    v             -- top global accumulator
    (Digit a)
    (FingerTree v (Node v a)) -- first v is global accumulator
    (Digit a)

It is the first and the second v (or monoidal annotations) from the above data
type that are replaced by mempty, which is the monoidal identity element (i.e.  
empty set). The above is done for the remaining of Top finger tree functions 
that compute global accumulators.

As a last comparison, we take the third and fourth rules from the ◁ operator. Again, we pair up the corresponding function definitions from 
FULL and TOP.

\[
\begin{align*}
a \triangleleft \text{Deep } v \ (\text{Four } b \ c \ d \ e) \ m \ sf &= & \text{FULL version} \\
  & = \text{Deep (measure } a \triangleleft v) \ (\text{Two } a \ b) \ (\text{node3 } c \ d \ e \triangleleft m) \ sf \\

a \triangleleft \text{Deep } v \ pr \ m \ sf &= & \text{FULL version} \\
  & = \text{Deep (measure } a \triangleleft v) \ (\text{consDigit } a \ pr) \ m \ sf
\end{align*}
\]

\[
\begin{align*}
a \triangleleft \text{Deep } v \ (\text{Four } b \ c \ d \ e) \ m \ sf &= & \text{TOP version} \\
  & = \text{Deep mempty} \ (\text{Two } a \ b) \ (\text{node3 } c \ d \ e \triangleleft m) \ sf \\

a \triangleleft \text{Deep } v \ pr \ m \ sf &= & \text{TOP version} \\
  & = \text{Deep mempty} \ (\text{consDigit } a \ pr) \ m \ sf
\end{align*}
\]

From the above snippet, we highlight smart constructor node3 which builds 
a 2-3 subtree for the prefix. As we are interested in the monoidal annotations 
for this subtree, we leave such constructor intact in Top finger tree function.

**searching in Top finger trees**

Since monoidal annotations from spine of the Top Ft are left out, we help 
out the “new” search function with the definition of an auxiliary function 
that collects the monoidal annotations from the affixes, on demand (i.e. at runtime).
collectSetsMid Empty = mempty
collectSetsMid (Single x) = measure x
collectSetsMid (Deep set prefix middle suffix)
  = (measure prefix) <> collectSetsMid middle <> (measure suffix)

Performance of collectSetsMid follows the performance of $\langle \rangle$, which is $\Theta(m \log(n/m))$, where $m$ and $n$ are the sizes of the sets evaluated during collectSetsMid. Since this function is recursive, it traverses the height of the finger tree in the worst case. Then, the overall performance is $\Theta(m \log n \log(n/m))$. Now, following the analysis done in Table 4.4 we get the bounds $\Omega(\log^2 n) \leq (\text{collectSetsMid}) \leq O(n \log n)$. In practice, however, the above does not pose any additional runtime as we shall see in the experimental results in Section 5.2.3.

Even though collectSetsMid helps out the search to find out a specific pair within the TOP finger tree, the former function poses a drawback when looking for the second edge when applying cut. Recall that sets in FULL store the two edges $(x,y)$ and $(y,x)$ as only one, that is $(\min(x,y), \max(x,y))$. Since the search in TOP is powered by the affixes only rather than the prefix-middle-suffix scheme, some edges in the ETT sequence can be lost in those unique-edge storage. In order to solve this out, we maintain the order of the edges in the sets for our TOP approach.

**link operation in TOP dynamic trees**

Dynamic update operations for TOP forests are pretty similar to those in FULL with the difference in updating the forest size and the storage of both edges in the ETT. Here are the snippets for linkTree and link.

```haskell
linkTree u tu v (TreeTop msetv tv) =
  let from = rerootTree tu u
      ft = ((left >>= Leaf (v,v)) >>= Leaf (v,u))
          >>= (ftTree from) >>= (Leaf (u,v) <$> right)
      mset = foldr buildMSet (mappend msetv (msetTree tu)) [(v,u),(u,v)]
   in TreeTop mset ft
```

Since we adding just two edges (highlighted bits) w.r.t. linkTree that of FULL trees, bound for this functions are $\Omega(\log^2 n) \leq (\text{linkTree}) \leq O(n \log n)$, where $n$ is the total number of nodes for the trees in the linkTree operation.

```haskell
link x y forest@(ForestTop mset ft) =
```
case connected x y forest of
    (False, Just(tx,rx,ty,ry)) → linkAll (linkTree x tx y ty)
    _ → forest

where
    linkAll tree = ForestTop msetn (tree ◁ (lf ◁ rf))
    msetn = foldr buildMSet mset [(x,y),(y,x)]
    Position (_,if') _ (_,rf') = searchMSet (x,x) mset ft
    Position (_,if) _ (_,rf) = searchMSet (y,y) mset (lf' ◁ rf')

By following the operation linkTree above, we get that lower and upper bounds for link for TOP forests are \(\Omega(\log^2 n)\) and \(O(n \log n)\) respectively, where \(n\) is the number of nodes in the TOP forest.

**cut operation in TOP dynamic trees**

Cutting trees by their own and within a TOP forest follow the same logic as link for TOP forests. Following are the corresponding snippets.

cutTree u v tree@(TreeTop mset ft) = case searchEdge pair mset ft of
    Position (ls,left) _ (rs,right) →
        case (searchEdge pair' ls left) of
            Position (lsL,leftL) _ (rsL,rightL) →
                (TreeTop rsL rightL, TreeTop (mappend lsL rs)(leftL ◁ right))
            _ → undefined -- error "Tree malformed"
    _ → undefined -- error "Tree malformed"

where
    pair = (u,v) ; pair' = (v,u)

cut x y forest@(ForestTop mset ft) =
    case edgeInForest (x,y) forest of
        Nothing → forest
        Just (tree,ltFor,rtFor) →
            buildForest (cutTree x y tree) (ftForest ltFor) (ftForest rtFor)

where
    msetn = foldr delPairMSet mset [(x,y),(y,x)]
    buildForest (leftTree, rightTree) lFor rFor = ForestTop msetn (lTree ◁ ((lFor ◁ rFor) ◁ rightTree))

Similarly, the lower and upper bounds for each operation are \(\Omega(\log^2 n)\) and \(O(n \log n)\) respectively. However, for cutTree, \(n\) is the size of the sets of the TOP tree trimmed and for cut \(n\) is the number of vertices in the TOP forest.
Summary of TOP operations performance

Based in the previous analyses per TOP dynamic tree operations and from the performance stated at Table 4.2, we have in Table 5.14 the summary of bounds per operation. Recall the bounds are not longer amortised as monoidal annotations comprise worst time bounds.

<table>
<thead>
<tr>
<th>Operation</th>
<th>name in FULL</th>
<th>best case</th>
<th>worst case</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>rootMSet</td>
<td>$\Omega(\log n)$</td>
<td>$O(n)$</td>
<td>trees</td>
</tr>
<tr>
<td>connected</td>
<td>connectedMSet</td>
<td>$\Omega(\log^2 n)$</td>
<td>$O(n \log n)$</td>
<td>forest</td>
</tr>
<tr>
<td>cut</td>
<td>cutMSet</td>
<td>$\Omega(\log^2 n)$</td>
<td>$O(n \log n)$</td>
<td>forest</td>
</tr>
<tr>
<td>link</td>
<td>linkMSet</td>
<td>$\Omega(\log^2 n)$</td>
<td>$O(n \log n)$</td>
<td>forest</td>
</tr>
</tbody>
</table>

Table 5.14: Performance of the TOP dynamic tree operations, where $n$ is the number of nodes in TOP forest.

5.2.3 TOP vs FULL experimental results

Runtime bounds for both FULL and TOP dynamic trees are equivalent, see tables 5.1 and 5.14. Rather than presenting the performance for each experiment in TOP dynamic trees, we compare the results of our two proposals under the same experimental setup described in Section 5.1.3.

Forest creation

For forests construction we present the results in two charts in such way the curves can be distinguishable, commencing with the 300-node forests in Figure 5.34 followed by remaining unit, 2-node, and 10-node cases in Figure 5.35.
Figure 5.34: Performance of constructing 300-node forests, Full vs Top.
Experimental results of construction Full vs Top forests

Top dynamic trees outperforms the Full counterpart on all the cases. In particular, Top 300-node forest runs in practically half of the runtime w.r.t. Full counterpart (in Figure 5.34).

Connectivity

In the look up operation of connected, we present the results for 10-node and 300-node cases in Figure 5.36 for both Full and Top dynamic trees.
Experimental results of Full vs Top connectivity

Top dynamic trees outperforms the Full counterpart on all the cases. In particular, Full 10-node forest runs in practically double of the runtime w.r.t. Top counterpart (in Figure 5.36).

Linking trees in forests

Due to the differences between all the link runtime performances, we present the results in three charts. unit and 2-node are shown in Figure 5.37, 10-node in Figure 5.38 and 300-node is presented in Figure 5.39.
Figure 5.37: Performance of link operation in unit and 2-node forests, Full vs Top.
Figure 5.38: Performance of link operation in 10-node forests, Full vs Top.

Figure 5.39: Performance of link operation in 300-node forests, Full vs Top.
Experimental results of `link` for **Full vs Top** forests

For each case of `link` in figs. 5.37 to 5.39 Top outperforms significantly Full. The trend is 3 to 1 faster, except for the 300-node case where the trend is 2 to 1.

**Cutting trees in forests**

In Figure 5.40 we illustrate all of the cases of `cut` operation for one-tree, 300-node, 10-node and 2-node forests, Full vs Top.

![Performance of cut in forests, Full vs Top](image)

Figure 5.40: Performance of `cut` operation for one-tree, 300-node, 10-node and 2-node forests, Full vs Top.

**Experimental results of `cut` for Full vs Top forests**

Except for the 2-node case, all performances of operation `cut` in Figure 5.40, the growth of the curve is faster for Full dynamic trees. The differences amongst each proposal vary from 1.12 to 1.29 Top being faster than Full, except for the 2-node case.
Linking and cutting trees in forests

We conduct two experiments here with the aim to measure individual performance of link and cut operations over 10-node and 300-node forests, for both FULL and TOP approaches. The input for the first experiment, in Figure 5.41, is regarded to the number of nodes. The input for the second experiment, in Figure 5.42, is the number of link-cut operations over a forest size of 3,011 nodes.

![Performance of individual link-cut operations over Full vs Top forests](image)

Figure 5.41: Performance of individual link-cut operations over 10-node and 300-node forests, FULL vs TOP.
Figure 5.42: Performance of individual \texttt{link-cut} operations over (3,011 vertices) \textit{10-node} and \textit{300-node} forests, \texttt{FULL} vs \texttt{TOP}.

\textbf{Experimental results of \texttt{link-cut} operations over \textit{10-node} and \textit{300-node} forests, \texttt{FULL} vs \texttt{TOP}}

For all of the cases in figs. 5.41 and 5.42, performance of \texttt{TOP} shows a lower cost in time than \texttt{FULL} approach, being \texttt{TOP} faster, from 1.2 up to 2.0 times (Figure 5.41) when input is the number of nodes and from 1.6 up to 1.8 times faster (Figure 5.42) when input is the number of \texttt{link-cut} per operations. Furthermore, the trend of the plotted curves in all cases is sublinear.
Chapter 6

Conclusion

In the analysis of the state of the art for dynamic trees problem implementation we realised there are some gaps in the literature. We focused in answering two of those gaps.

Firstly, we focused our attention to the feasibility in implementing, under the purely functional programming setting, the linearisation case of such a dynamic trees problem stated by Sleator and Tarjan [1] in Chapter 1. In our attempt to bridge such a gap we presented in Chapter 5 two data structures, named FULL and TOP.

Secondly, making explicit the location of the vertices involved in the most common operations over dynamic trees, as this has been taken for granted in current imperative programming implementations (pointer-based). Our analysis stepped onto the specifications by Henzinger and King [5] and Tarjan in [7] and as a result we devised FUENET, a purely functional programming specification in Chapter 4 and calculate the lower and upper bounds for the auxiliary and main functions for FULL and TOP.

Finally, we showed in practice, that our claim for the performance of the main dynamic operations connected, link and cut over FULL and TOP data structures is sublinear per operation.

The achievements in theory and practice for the aforementioned proposals, is due to the following contributions:

- We demonstrated the performance and implementability of FUENET in the purely functional programming Haskell. Practical performance was conducted by benchmarking experimental analysis on FULL dynamic trees and on TOP dynamic trees, in Chapter 5.
• Performance claimed in the Abstract is demonstrated in theory (in Chapter 4) and in practice (in Chapter 5). Specifically, $\Omega(\log^2 n) \leq (\text{dynOp}) \leq O(n \log n)$, where $n$ is the number of vertices in the forest and that of dynOp $\in \{\text{connected, cut, link}\}$.

• We showed (in chapters 4 and 5) that the definition of the monoidal annotation of a finger tree is crucial to its performance. We believe this is the first time this fact is explicitly stated, not assumed. For instance, in [8], Hinze and Paterson claimed that performance of the insertion operation from the left or $\triangleleft$ operator is $\Theta(\log n)$ amortised. We step on top of that and claimed that such a performance is $\Theta(\log n) \times O(\triangleright)$ where $\triangleright$ is the monoidal binary operation definition. Since the applications published in [8] defined only $\Theta(1)$ monoidal annotations such as the arithmetic addition or the max or min functions, the original claim holds. However, by defining the monoidal annotation as the set-union, the operator is then $O(m \log(n/m))$ worst case where $m$ and $n$ are the size of the sets evaluated in the set-union operation. Hence the overall performance for $\triangleleft$ is $\Omega(\log n) \leq \Theta(\triangleleft) \leq O(n \log n)$, where $n$ is the number of vertices in the forest.

• Let $n$ be the number of vertices in a forest. We showed experimentally that

- $O(n \log n)$ is the worst case per operation for our dynamic operations (Full and Top) cut and link, where $n$ is the number of vertices in the forest. This occur when the same operation is applied to a unit forest turning it onto a one-tree forest or from a one-tree forest downsizing it to a unit forest. This is depicted in figs. 5.37 and 5.40 in Chapter 5.

- Operations cut and link (in both Full and Top) perform sublinear per operation when applied to a forest which size (number of nodes + number of edges) never is its maximum ($3n - 2$) nor its minimum ($n$). This is illustrated in figs. 5.41 and 5.42 in Chapter 5.

- Outliers in the curves from our experiments where identified and analysed. The reason behind such behaviour is due to the monoidal operation of set-union when performed on demand, particularly on the affixes of the finger tree. This is depicted in figs. 5.18 and 5.20 in Chapter 5.

6.1 Further Directions

A lot of work remains to be carried out. In the implementational side, we highlight the following
1. A variation of the dynamic query (FULL and TOP) could be to ask for *the path* (since paths on trees are unique) that actually connects $u$ and $v$ if they are connected.

2. Design of evaluation strategies when parallelising the two nodeIn functions that comprise connected. Furthermore, parallelism for search in TOP dynamic trees could traverses both affixes at a time.

3. Experimental analysis can be conducted for the case when $<>$ operator over the affixes is changed from *on demand* to *in advance* approach. Since theoretical bounds should remain the same, experimental analysis could not only smooth the outliers in the performance of $<>$, but in a truly runtime reduction by a constant factor.

On the other hand, formal proofs and verification can be lead to at least

1. Proof of correctness for every data structure in this thesis

2. Proof of correctness and completeness for every operation upon the data structures

3. Analysis of the input sequence of dynamic tree operations when behaving as infinite list.
Bibliography


[38] ——, Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming. " O'Reilly Media, Inc.", 2013.


Index

binary search tree, 11, 32
  BST, 32
  example, 33
  in Haskell, 32
connectivity, 13
data structure
collection, 32
digit
  example, 40
dynamic tree, 11
  applications, 14
  in functional programming, 14
  operations, 11
  problem, 12
equational reasoning, 17
Euler-tour tree, 25
finger tree, 11, 37
  data type, 40
  example, 40
  operations, 45
  structure, 38
forest, 29
  n-node, 30
  one-tree, 31
  size, 30
  unit, 29
forest property, 13
function vs operation, 19

Haskell, 16
input tree
  example, 32
  in Haskell, 31
link cut trees, 15
measure, 46
monoid, 36
  in Haskell, 36
monoidal annotation, 11
node
  data type, 39
  example, 39
ordered sequence
  application, 59
semigroup, 36
sequence, 11
singleton tree, 31
tree
  2-3 tree, 34
  complete 2-3 tree, 35
  Euler-tour, 25
  leafy, 34
  nodal, 34
  singleton, 29
  size, 30