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 $\mathcal{O}(\log n)$ 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

List of Figures

List of Tables

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

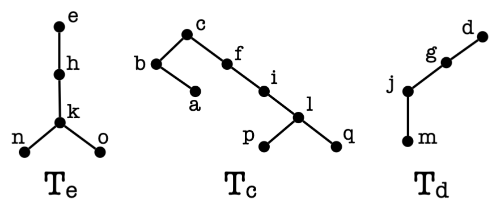

Figure 1.1: Example of a forest, called `f`.

“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 `cut`ting, 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

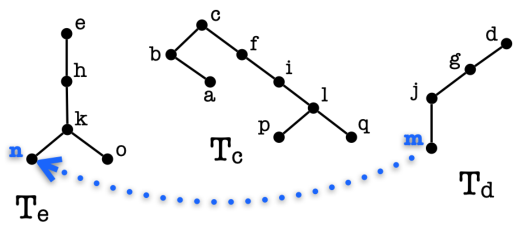

Figure 1.2: Identifying the vertices, `connected`==`False`, in `f` for which `link` is about to be applied.

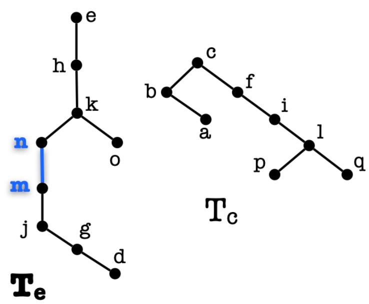

Figure 1.3: `link` is applied over `f`.

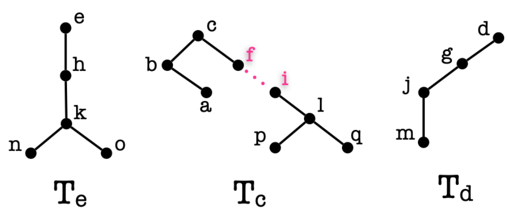

Figure 1.4: Identifying the vertices, `connected`==`True`, in `f` for which `cut` is about to be performed.

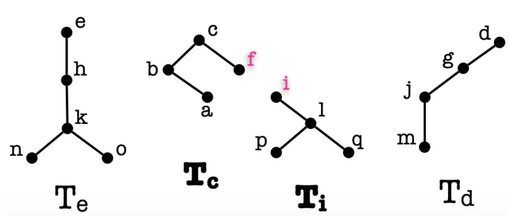

Figure 1.5: `cut` is performed on `f`.

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 *incremental* and *decremental* otherwise [9]. In any case, we can refer to any of the above structures to be *semi-dynamic*. If we want to distinguish between semi-dynamic data structures and data structures allowing both operations, then the latter are called fully *dynamic data structures*. We provide implementation and experimental analysis for the semi and fully dynamic cases.

Note that the term *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:

```
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

```
sum(flatten t) = treesum t
```

holds for all finite defined trees of type `Tree Int`.

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

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

```
flatten :: Tree a → List a
flatten Empty        = []                              -- [flat.1]
flatten (Node l x r) = flatten l ++ [x] ++ flatten r   -- [flat.2]

treesum :: Tree Int → Int
treesum Empty        = 0                           -- [tsum.1]
treesum (Node l x r) = treesum l + x + treesum r   -- [tsum.2]

sum :: [Int] → Int
sum []     = 0            -- [sum.1]
sum (x:xs) = x + sum xs   -- [sum.2]
```

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 `Tree a`, it is enough to prove that

- P(`Empty`) holds outright;
- P(`Node l x r`) holds whenever P(`l`) and P(`r`) both hold.

Part 1: Prove P(`Empty`) holds outright

```
sum ( flatten Empty )  =  sum []           by [flat.1]
                       =  0                by [sum.1]
                       =  treesum Empty    by [tsum.1]
```

Part 2: Prove P(`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 `Int` lists `xs`, `ys`, `zs` we have

```
sum ( xs ++ ys ++ zs )  =  sum xs + sum ys + sum zs    -- [lemma]
```

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

```
sum (flatten (Node l x r))
          =  sum (flatten l ++ [x] ++ flatten r)          by [flat.2]
          =  sum (flatten l) + sum [x] + sum (flatten r)  by [lemma]
          =  treesum l + sum [x] + sum (flatten r)        by [P(l)]
          =  treesum l + sum [x] + treesum r              by [P(r)]
          =  treesum l + (x + sum []) + treesum r         by [sum.2]
          =  treesum l + (x + 0) + treesum r              by [sum.1]
```

```
= treesum l + x + treesum r                    by [arithmetic]
= treesum (Node l x r)                         by [tsum.2]
```

■

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 $maximum\{1, \log_2 x\}$ 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

> ...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 ...

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 $\mathcal{O}(\log n)$ 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, 32]. 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].

- *link*(vertex v, w, real x): Combine the trees containing v and w by adding the edge (v, w) of cost x, making w the parent of v. This operation assumes that v and w are in different trees and v is a tree root.

- *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 $\mathcal{O}(\log n)$ each.

```haskell
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)
             → LinkCut a (PrimState m)
             → m ()
```

Practically, all the operations in Kmett's work rely on the `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 `link`

```haskell
link v w = st $ do
    access v
    access w
    set path v w
```

An attempt to apply the equational reasoning we have seen in 1.4.1 over `access` and `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

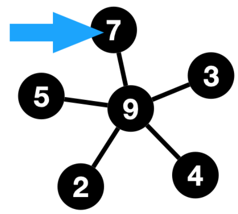

Figure 2.1: Input tree, with arbitrary node selected as its root

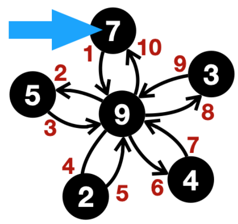

Figure 2.2: Input tree, non directed edges are turned into directed ones. Direction is selected arbitrary.

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

Figure 2.3: ETT sequence, built from the directed edges from input tree in Figure 2.2.

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.

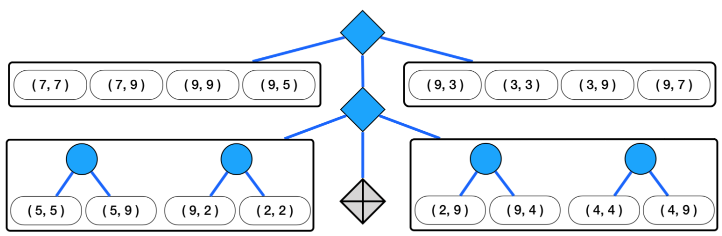

Figure 2.4: Finger tree FT corresponding to ETT sequence from Figure 2.3

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.

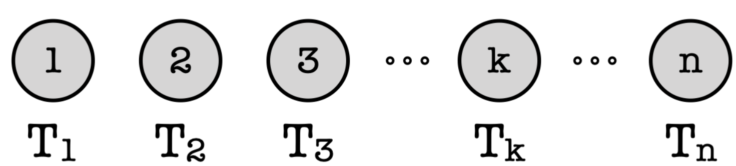

Figure 3.1: A *unit* forest

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.

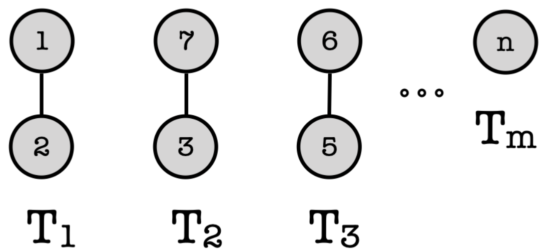

Figure 3.2: A *2-node* forest, all trees but T_m with 2 vertices each

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

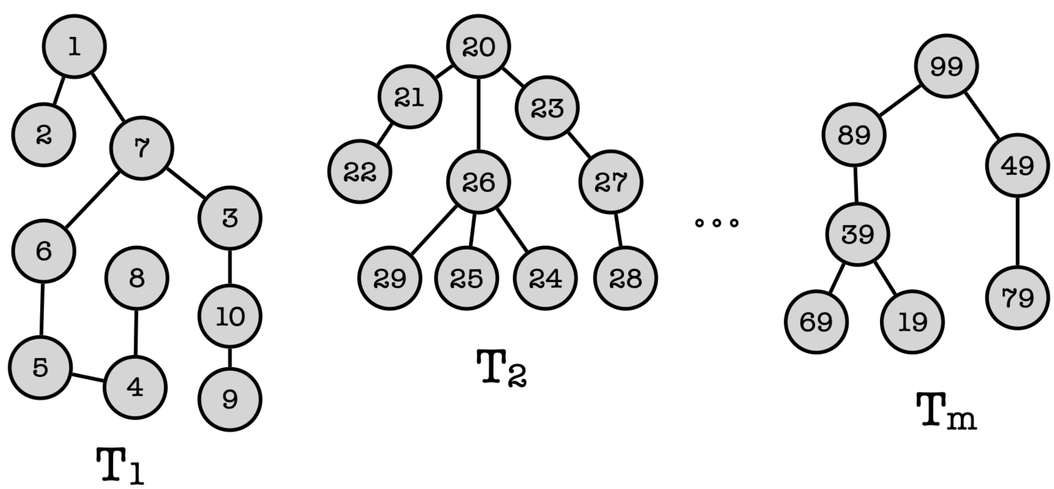

Figure 3.3: A *10-node* forest, all trees but T_m with 10 vertices each

If a forest has only one tree and `ForestSize` $=$ `TreeSize`, where NumNodesForest≥ 2, is called *one-tree* forest.

3.2 Input tree data structure

The basic (rose) tree type, Tree a, is defined in Haskell's Data.Tree package [41] as

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

where [1]

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

This definition has both advantages and disadvantages for our own work. Since the definition of 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 $\mathcal{O}(n)$ 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.

[1] The 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                          -- implicitly defined root vertex
         [ 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:

```
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.

Function	Description	Complexity
`member`	testing membership	$\mathcal{O}(\log n)$
`insert`	inserting an element	$\mathcal{O}(\log n)$
`union`	disjoint union	$\mathcal{O}(m \times (\log(n/m) + 1))$

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

```
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) )
```

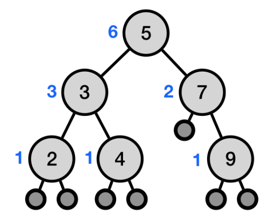

Figure 3.5: The BST corresponding to the input tree in Fig. 3.4. Notice that the data elements (the numbers within the circles) have been sorted into ascending order. Numbers next to each circle give the size of the tree rooted at that node. Small circles correspond to instances of the `Tip` data constructor.

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.

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

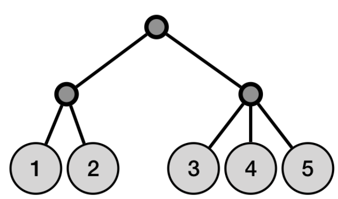

Figure 3.6: Leafy tree, a 2-3 tree with data only on the leaves

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

```haskell
data TreeN b
  = LeafN
  | Node2N (TreeN b) b (TreeN b)
  | Node3N (TreeN b) b (TreeN b) b (TreeN b)
```

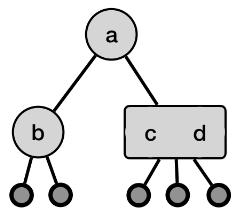

Figure 3.7: A non Leafy tree, a 2-3 tree with data only on internal nodes

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.

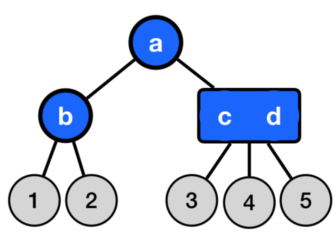

Figure 3.8: A complete 2-3 tree (of type `TreeB Char Int`), defined with two different type arguments for values on the nodes and leaves

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

$$\epsilon \star x = x \star \epsilon = x \tag{3.1}$$

monoidal indentity

$$x \star (y \star z) = (x \star y) \star z \tag{3.2}$$

monoidal associativity

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

```
mempty `mappend` x = x `mappend` mempty = x             -- [mon.identity]
x `mappend` (y `mappend` z)  = (x `mappend y) `mappend` z  -- [mon.assoc]
```

where `mempty` is the ϵ, and `mappend` is the $\star$ operation.

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

```
class Semigroup a ⇒ Monoid a where
  mempty  :: a
  mappend :: a → a → a
```

As shown, the `Monoid` class definition is constrained by `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.

```
class Semigroup a where
  (<>) :: a → a → a
```

We shall use `<>` and `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 $\star$ functions.

```
x <> 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.

$$\text{a } \texttt{Sequence} \text{ is represented by a } \begin{cases} \texttt{List} & \text{linear, simple, inefficient} \\ \texttt{FingerTree} & \text{non-linear, complex, efficient} \end{cases}$$

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.

$$\text{a } \texttt{FT} \text{ is either } \begin{cases} \texttt{Empty} & \rightsquigarrow \text{ empty } \texttt{FT} \\ \texttt{Single} & \rightsquigarrow \text{ containing a single element} \\ \texttt{Deep} & \begin{cases} \texttt{prefix} \rightsquigarrow \text{ of type } \texttt{Digit}\text{, defined below} \\ \texttt{middle} \rightsquigarrow \text{ of type } \texttt{Node}\text{, defined below} \\ \texttt{suffix} \rightsquigarrow \text{ of type } \texttt{Digit}\text{, defined below} \end{cases} \end{cases}$$

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).

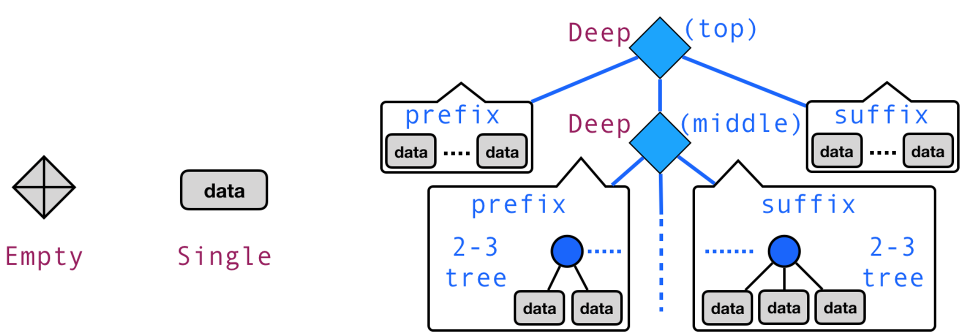

Figure 3.9: The three data constructors in the Ft definition

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

```
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, $e_1 \ldots e_5$.

```
nodeExample1 = Node2 z e₁ e₂
nodeExample2 = Node2 z (Node3 z e₁ e₂ e₃) (Node2 z e₄ e₅)
```

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).

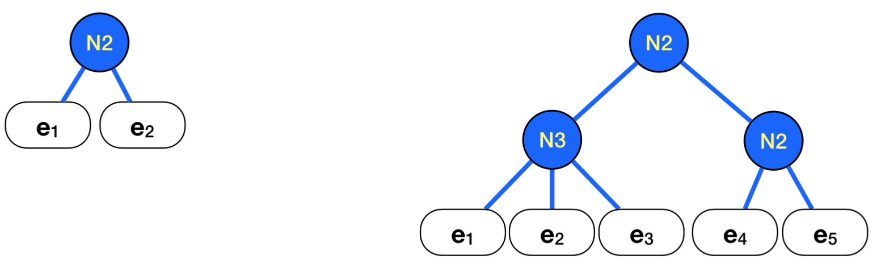

Figure 3.10: Recursive and non-recursive examples of the `Node` type

The prefix and suffix data types

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

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

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

Examples

Consider these definitions (illustrated in Fig. 3.11):

```
digitExample1 = Four e1 e2 e3 e4
digitExample2 = Three (Node2 z e1 e2) (Node2 z e3 e4) (Node2 z e5 e6)
```

In these examples `digitExample1` has type `Digit a` whereas `digitExample2` has type `Digit (Node v a)`.

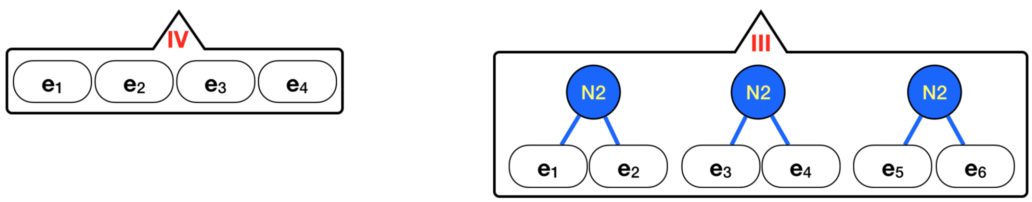

Figure 3.11: Recursive and non-recursive examples of `Digit` type

The `FingerTree` data type

Putting all of above data types together, the following is the Haskell definition for the general case of a finger tree FT .

```
data FingerTree v a
  = Empty
  | Single a
  | Deep                              -- constructor for a FT
        v                             -- 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)
```

FT Example

The sequence $e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9$ within a FT can be defined as follows and depicted as in Figure 3.12. We help out the example with auxiliary functions for the FT affixes.

```
Deep z pref1 mid1 suf1
 where
    pref1 = Three e1 e2 e3
    mid1  = Deep z pref2 mid2 suf2  -- recursive FT call
    suf1  = Two e8 e9

    pref2 = One (Node2 z e4 e5)
    mid2  = Empty
    suf2  = One (Node2 z e6 e7)
```

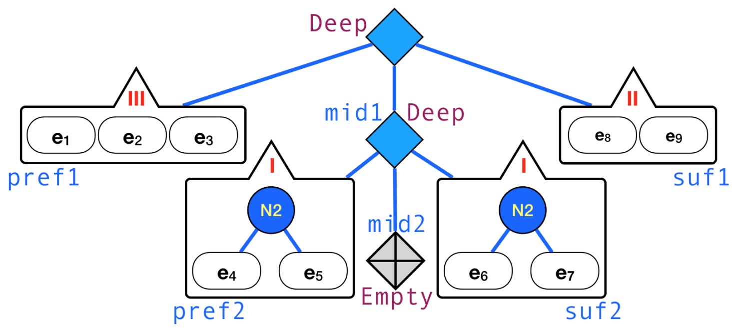

Figure 3.12: `FingerTree` holding a sequence of nine elements ($e_1 \ldots e_9$)

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 `FT`s, 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

$$2\sum_{k=0}^{h}(4 \times 3^k) \tag{3.3}$$

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.

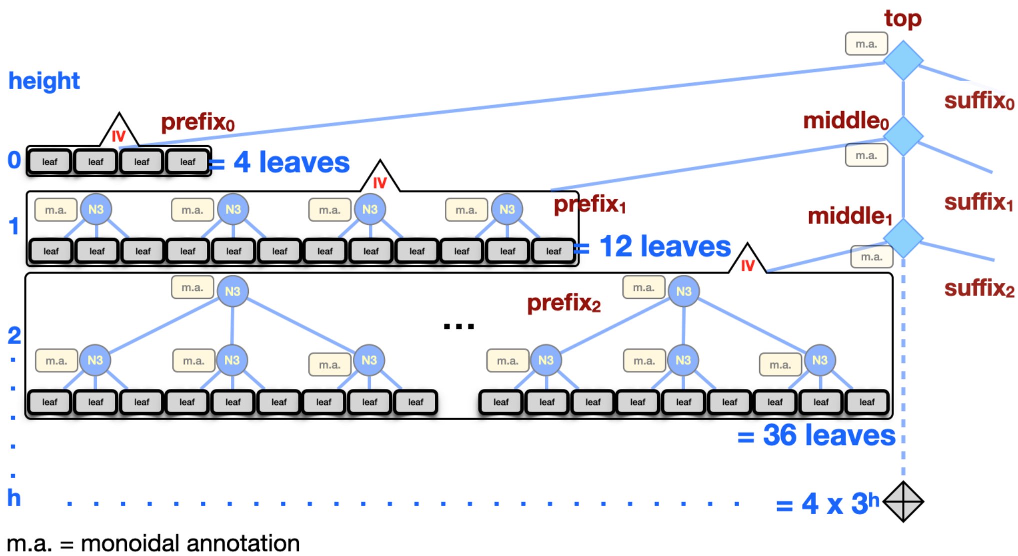

Figure 3.13: Maximum number of leaves in a `FT`, with `Empty` bottom

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

$$2\sum_{k=0}^{h}(4 \times 3^k) + 3^{(h+1)} \tag{3.4}$$

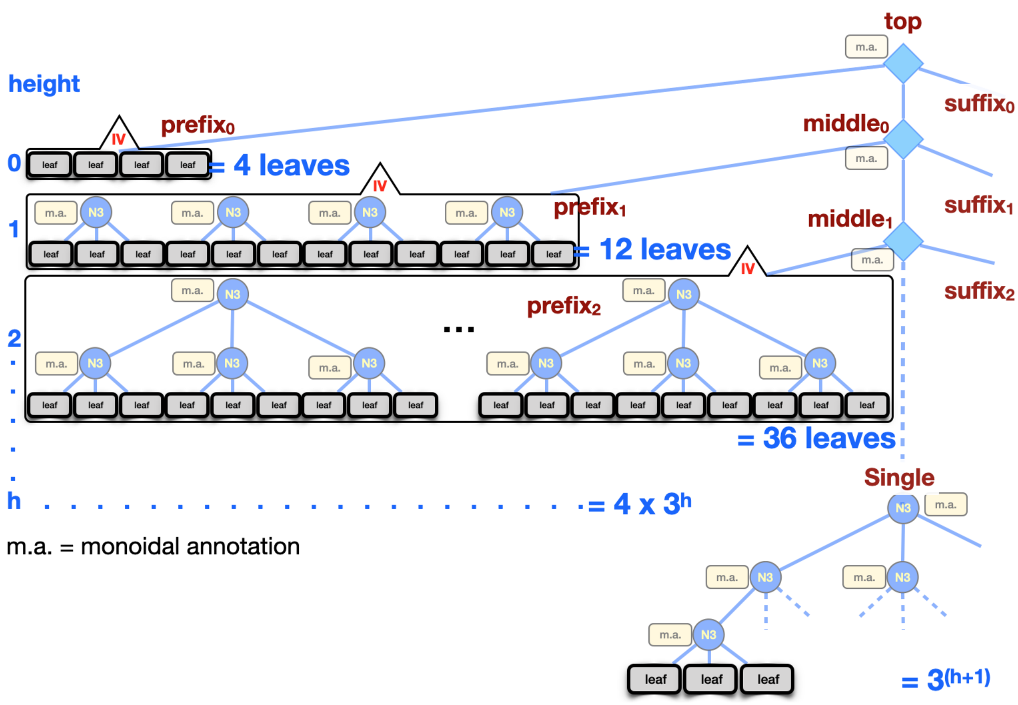

Figure 3.14: Maximum number of leaves in a FT, with bottom being Single

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

$$2\sum_{i=1}^{h}\left(4\sum_{j=1}^{i}3^{j-1}\right)+h+1 \tag{3.5}$$

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.

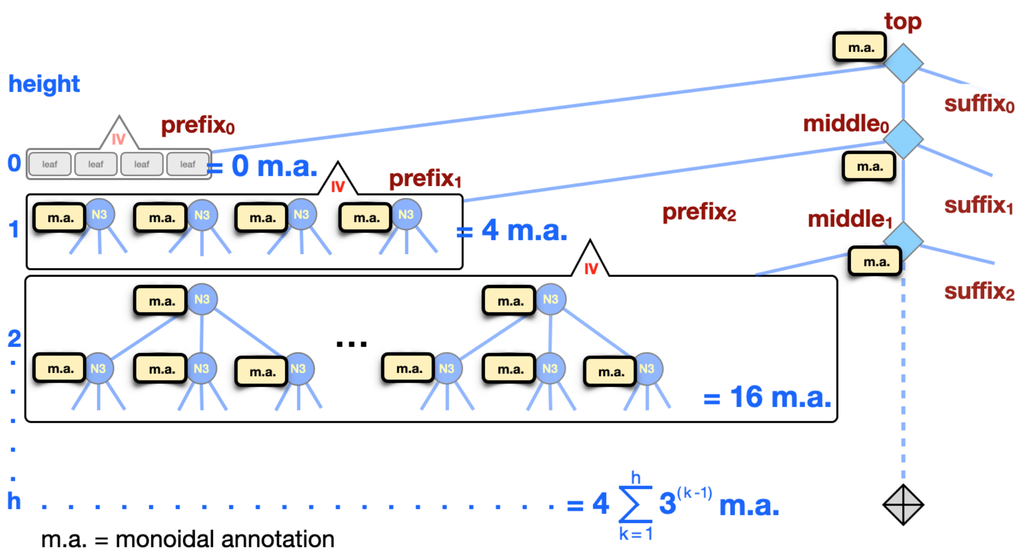

Figure 3.15: Maximum monoidal annotations in a FT, with Empty bottom

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.

$$2\sum_{i=1}^{h}\left(4\sum_{j=1}^{i}3^{j-1}\right)+\sum_{k=1}^{h+1}3^{k-1}+h+2 \tag{3.6}$$

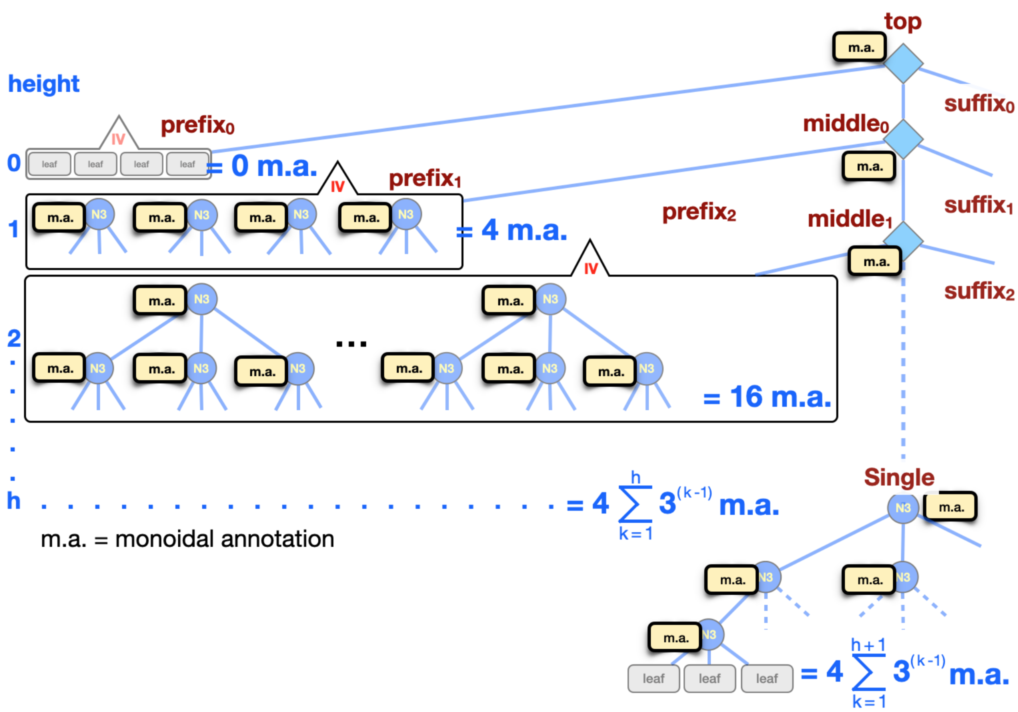

Figure 3.16: Maximum monoidal annotations in a FT, with Single bottom

Finally, the maximum number of <>s (i.e. monoidal binary operation) carried out in a FT occurs one time per Node2 subtrees and two time per Node3 subtrees. Furthermore, in the spine, for every non-empty subtree we have two more <> operations.

$$2\sum_{i=1}^{h}\left(4\sum_{j=1}^{i}(2\times 3^{j-1})\right)+2h \tag{3.7}$$

And the corresponding Single bottom at the spine, we have

$$2\sum_{i=1}^{h}\left(4\sum_{j=1}^{i}(2\times 3^{j-1})\right)+\sum_{k=1}^{h+1}(2\times 3^{k-1})+2h+1 \tag{3.8}$$

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

```
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`

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

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

```
instance (Measured v a) ⇒ 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

```
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`

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

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

```
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.

```
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 `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 `Data.FingerTree` [48] is the Haskell implementation of [8]. It contains the data types, constructors, update and transformation function definitions for the general purpose finger tree data structure. Amongst all operations in `Data.FingerTree`, we limit our proposals FULL and 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 [48]. That is, any operation listed in Table 3.2 is valid when applied to `FT`s of atomic elements, i.e. `Int` as well as valid when applied to `FT`s of elements of type (`Node (<>) (Node Int)`). Since `<>` has type `Monoid v ⇒ v → v → v`, we represent the term *monoidal annotation* with `<>`.

Function	Description	Complexity
`viewl`	view the first element of the tree	$\mathcal{O}(1)$
`viewr`	view the last element of the tree	$\mathcal{O}(1)$
$\triangleleft$	inserting from the left	$\mathcal{O}(1)$
$\triangleright$	inserting from the right	$\mathcal{O}(1)$
$\bowtie$	appending two trees (concatenation)	$\mathcal{O}(\log(\min(n, m)))$
`split`	split a tree into two subtrees	$\mathcal{O}(\log n)$
`search`	looking for an element and perform a `split`	$\mathcal{O}(\log n)$

Table 3.2: Finger tree operations, taken from [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 $\bowtie$ assumes that $m \leq n$ (if not, we can swap the order of the arguments before applying $\bowtie$). All bounds are amortised.

Bounds in the above table are stated to be amortised by Hinze and Paterson in [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 `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 viewl and viewr allow viewing and removing, in $\mathcal{O}(1)$ time amortised, the endpoints of a FT. Let us start with the viewl definition. Analysis on viewr applies similarly.

```
viewl :: (Measured v a) ⇒ FingerTree v a → ViewL (FingerTree v) a
viewl Empty                  =  EmptyL               -- [viewl.empty]
viewl (Single x)             =  x :< Empty           -- [viewl.singleton]
viewl (Deep _ (One x) m sf) =  x :< rotL m sf       -- [viewl.recursiveCase]
viewl (Deep _ pr m sf)
    =  lheadDigit pr :< deep (ltailDigit pr) m sf  -- [viewl.regularCase]
```

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

```
lheadDigit :: Digit a → a
lheadDigit (One a) = a
lheadDigit (Two a _) = a
lheadDigit (Three a _ _) = a
lheadDigit (Four a _ _ _) = a
```

Function ltailDigit is defined similarly in [48].

```
deep :: (Measured v a) ⇒
        Digit a → FingerTree v (Node v a) → Digit a → FingerTree v a
deep pr mid sf = Deep ((measure pr <> measure m) <> measure sf) pr mid sf
```

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

For the rule viewl.recursiveCase, rotL, we have

```
rotL :: (Measured v a) ⇒
       FingerTree v (Node v a) → Digit a → FingerTree v a
rotL m sf      =   case viewl m of
    EmptyL   →  digitToTree sf
    a :< m’  →  Deep (measure m <> measure sf) (nodeToDigit a) m’ sf
```

Similar to lheadDigit, nodeToDigit perform in $\Theta(1)$. Alike deep, function digitToTree depends upon <> definition. Both, nodeToDigit and digitToTree are detailed in [48].

Complexity of `viewl` and `viewr`

We focus on `viewl` in this thesis, `viewr` can be analised 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 downto 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 $\mathcal{O}(\log n) \times \mathcal{O}(\texttt{<>})$. 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 $\mathcal{O}(1) \times \mathcal{O}(\texttt{<>})$ 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 ($\triangleleft$) or at the rear ($\triangleright$) of a FT. We present the $\triangleleft$ case. Details of $\triangleright$ 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.

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

[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 $\mathcal{O}$(`<>`). 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 $\mathcal{O}(1)$ 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 ($x_1 \ldots x_6$) into an `Empty` finger tree. By doing this, we shall enforce the four rules of ◁.

```
x1 ◁ x2 ◁ x3 ◁ x4 ◁ x5 ◁ x6 ◁ Empty
= { by ◁.1 }                                                    -- [Op.1]
x1 ◁ x2 ◁ x3 ◁ x4 ◁ x5 ◁ (Single x6)
= { by ◁.2 }                                                    -- [Op.2]
x1 ◁ x2 ◁ x3 ◁ x4 ◁ (deep (One x5) Empty (One x6))
= { by deep function definition }                               -- [Op.3]
x1 ◁ x2 ◁ x3 ◁ x4 ◁ (Deep (<>1) (One x5) Empty (One x6))
= { by ◁.4 }                                                    -- [Op.4]
x1 ◁ x2 ◁ x3 ◁ (Deep (<>2) (consDigit x4 (One x5)) Empty (One x6))
= { by consDig.1 }                                              -- [Op.5]
x1 ◁ x2 ◁ x3 ◁ (Deep (<>2) (Two x4 x5) Empty (One x6))
= { by ◁.4 }                                                    -- [Op.6]
x1 ◁ x2 ◁ (Deep (<>3) (consDigit x3 (Two x4 x5)) Empty (One x6))
= { by consDig.2 }                                              -- [Op.7]
x1 ◁ x2 ◁ (Deep (<>3) (Three x3 x4 x5) Empty (One x6))
= { by ◁.4 }                                                    -- [Op.8]
x1 ◁ (Deep (<>4) (consDigit x2 (Three x3 x4 x5)) Empty (One x6))
= { 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 ◁.1 }                                                    -- [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.

Op	x_i	$<>_i$	$<>_{acc}$	function	FT depth
1	x_6	(N/A)	0	Single	0
2	x_5	$(<>_1)$	2	deep	0
3	x_5	$(<>_1)$	2		0
4	x_4	$(<>_2)$	3	Deep	0
5	x_4	$(<>_2)$	3		0
6	x_3	$(<>_3)$	4	Deep	0
7	x_3	$(<>_3)$	4		0
8	x_2	$(<>_4)$	5	Deep	0
9	x_2	$(<>_4)$	5		0
10	x_1	$(<>_5)$	6	Deep	0
11	x_1	$(<>_6)$	8	Node3	0
12	x_1	$(<>_{5,6})$	9	Empty	1

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 tree [3] 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 $8 \times \mathcal{O}(<>) = \mathcal{O}(<>)$.

Suppose `x` is a valid datum for `t` where `t` is a FT of depth $\mathcal{O}(\log n)$ 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 $8 \times \mathcal{O}(\log n) \times \mathcal{O}(<>)$. Hence the amortised time per insertion is $\mathcal{O}(\log n) \times$

[3] `FingerTree a`, `FingerTree (<>) (Node a)`, `FingerTree (<>) (Node (<>) (Node (<>) a))`, etc.

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

3.6.6 Appending FTs

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

1. generates a *new* subtree, `mid`, concatenating `mid1`, `sf1`, `pr2` and `mi2`
2. creates a *new* finger tree `ft = Deep (mon1 <> mon2) pr1 mid sf2`

The appending process, numeral 1 above, is carried out by recursive calls of *interleaved* functions `appendTree`$_i$ and `addDigits`$_i$. The former inserts the elements `a` via $\triangleleft$ and $\triangleright$. The latter function compares all combinations possible from `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 `appendTree`$_i$, we have

```
appendTree_i :: (Measure v a) ⇒
                   FingerTree v a → α_i →FingerTree v a →FingerTree v a
appendTree_i Empty        α_i xs          = α_i ◁_i  xs         -- [appendTree_i.1]
appendTree_i xs           α_i Empty       = xs  ▷_i α_i         -- [appendTree_i.2]
appendTree_i (Single x)  α_i xs          = x ◁ α_i ◁_i  xs     -- [appendTree_i.3]
appendTree_i xs           α_i (Single x) =xs  ▷_i α_i ▷  x     -- [appendTree_i.4]
appendTree_i (Deep mon1 pr1 mid1 sf1) α_i (Deep mon2 pr2 mid2 sf2)
    = Deep (mon1 <> measure_i α_i  <> mon2)
           pr1 (addDigits_i mid1 sf1 α_i pr2 mid2) sf2 -- [appendTree_i.5]
```

where α_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 `addDigits`$_i$ can be

```
addDigits_i :: (Edges v a) ⇒
              FingerTree v (Node v a) → Digit a → α_i →Digit a →
              FingerTree v (Node v a) → FingerTree v (Node v a)
addDigits_i m1 (One a) α_i (One f) m2
```

```
       = appendTree_i mid1 (node_k)_i ... mid2        -- [addDigits_i.1]
addDigits_i m1 (One a) α_i (Two f g) m2
       = appendTree_i mid1 (node_k)_i ... mid2        -- [addDigits_i.2]
...                                                   -- [addDigits_i.(3..14)]
addDigits_i mid1 (Four a b c d) α_i (Three i j k) mid2
       = appendTree_i mid1 (node_k)_i ... mid2        -- [addDigits_i.15]
addDigits_i mid1 (Four a b c d) α_i (Four i j k l) mid2
       = appendTree_i mid1 (node_k)_i ... mid2        -- [addDigits_i.16]
```

where (`node`$_k$) can be either (`node3 a b c`) or (`node2 a b`)

Complexity of ⋈

Since the actual implementation of ⋈ in [48] is defined throughout 106 lines of Haskell code (5×`appendTree` + 16×`addDigits` + ⋈), we just show a sample in this thesis.

```
pr1  = (One x1)
mid1 = Empty
sf1  = (Four x2 x3 x4 x5)
pr2  = (Three x6 x7 x8)
mid2 = Empty
sf2  = (One x9)
Deep mon1 pr1 mid1 sf1 ⋈ Deep mon2 pr2 mid2 sf2
= { by ⋈ }                                                        -- [Op.1]
appendTree0
= { by appendTree0 function definition }                          -- [Op.2]
Deep (mon1 <> mon2) pr1 (addDigits0 mid1 sf1 pr2 mid2) sf2
= { by addDigits0.15 }                                             -- [Op.3]
Deep (<>1) (One x1)
           ( appendTree3 mid1 (node3 x2 x3 x4)
                              (node2 x5 x6)
                              (node2 x7 x8) mid2 )
           (One x9)
= { by node3 and node2 function definitions }                      -- [Op.4]
Deep (<>1) (One x1)
           ( appendTree3 mid1 (Node3 (<>2) x2 x3 x4)
                              (Node2 (<>3) x5 x6)
                              (Node2 (<>4) x7 x8) mid2 )
           (One x9)
= { by appendTree3.1 }                                             -- [Op.5]
Deep (<>1) (One x1)
           ( (Node3 (<>2) x2 x3 x4) ◁
             (Node2 (<>3) x5 x6) ◁
             (Node2 (<>4) x7 x8) ◁  Empty )
```

```
           (One x₉)
= { by ◁.1, ◁.2, ◁.4 }                                        -- [Op.6]
Deep (<>₁) (One x₁)
           (Deep (<>₅)
                     Two (Node3 (<>₂) x₂ x₃ x₄) (Node2 (<>₃) x₅ x₆)
                     Empty
                     One (Node2 (<>₄) x₇ x₈)
           (One x₉)
```

Update operation is actually carried out by $\triangleleft$ and $\triangleright$ 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 $\bowtie$ is $\mathcal{O}(\log(min(n, m))) \times \mathcal{O}(\texttt{<>})$.

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 `<>` 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 `search` since it includes a `split`. A successful `search` in [8] implemented in [48] is considered when given a predicate with monoidal annotations as arguments, turns from `False` to `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 `False` and the latter for testing `True`. To catch up all possible results from searching, the following defines the data type for such results.

```
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 `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 Full and Top .

Initial search on a FT

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

```
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,

```
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.

```
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 mon₁pr mon₂sm =                                     -- [searchTree.2]
     let Split l x r     =  searchDigit p mon₁ pr mon₂sm
     in  Split (maybe Empty digitToTree l) x (deepL r mid sf) -- [Split.1]
  | p mon₁pm mon₂sf =                                     -- [searchTree.3]
     let Split ml xs mr = searchTree p mon₁pr mid mon₂sf
         Split l x r    = searchNode p
                           (mon₁pr <> measure ml) xs
                           (measure mr <> mon₂sf)
```

```
      in  Split (deepR pr ml l) x (deepL r mr sf)          -- [Split.2]
    | otherwise   =                                         -- [searchTree.4]
      let Split l x r    =  searchDigit p mon₁pm sf mon₂
      in  Split (deepR pr mid l) x (maybe Empty digitToTree r) -- [Split.3]
  where
    monmid = measure mid
    mon₁pr = mon₁ <> measure pr
    mon₁pm = mon₁pr <> monmid
    mon₂sm = monmid <> mon₂sf
    mon₂sf = measure sf <> mon₂
```

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 ≥ 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.

```
searchDigit :: (Edges v a) ⇒
               (v → v → Bool) → v → Digit a → v →
               Split (Maybe (Digit a)) a
searchDigit _ mon₁ (One a) mon₂ = Split Nothing a Nothing -- [searchDig.1]
searchDigit p mon₁ (Two a b) mon₂
  | p monₐ mon_b =Split Nothing a (Just (One b))          -- [searchDig.2]
  | otherwise   = Split (Just (One a)) b Nothing          -- [searchDig.3]
  where
    monₐ =  mon₁ <> measure a
    mon_b =  measure b <> mon₂
...
searchDigit p mon₁ (Three a b c) mon₂
...
searchDigit p mon₁ (Four a b c d) mon₂
```

...

Searching in Node

The ultimate frontier for searching an element in a Ft with depth ≥ 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.

```
searchNode :: (Edges v a) ⇒
              (v → v → Bool) → v → Node v a → v →
              Split (Maybe (Digit a)) a
searchNode p mon₁ (Node2 _ a b) mon₂
  | p monₐ mon_b = Split Nothing a (Just (One b))        -- [searchNode.1]
  | otherwise    = Split (Just (One a)) b Nothing        -- [searchNode.2]
  where
    monₐ = mon₁ <> measure a
    mon_b = measure b <> mon₂
...
searchNode p mon₁ (Node3 _ a b c) mon₂
```

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 $\mathcal{O}(\log n)$, 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 $\mathcal{O}(\log n)$ times while performing up to four <> operators when comparing the predicate p, that is, looking up for an element takes $\mathcal{O}(\log n) \times \mathcal{O}(<>)$. In Figure 3.17 we depict the look up for x

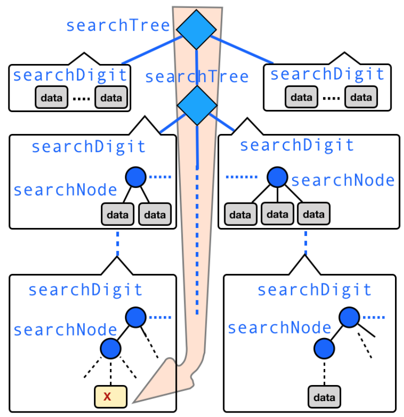

Figure 3.17: Search as a look up an element in a FT

Once element `x` has been found, `Split` function performs a bottom-up computation to construct two subtrees. Function `searchTree` calls for [$\texttt{Split}_{1,2,3}$] which in turn calls smart constructors `deepL`, `digitToTree` and `deepR`. These constructors *glue* the corresponding affixes and middle subtrees through `rotL`, `deep` and `rotR` helper functions. We present here just the functions processing the left hand side structures. Provided two affixes and a subtree (`mid`), `deepL` builds up a FT. In the absence of a prefix (passed as `Nothing`), `deepL` builds the FT from the `mid` and the suffix; in case `mid` is `Empty`, then FT is made of the suffix solely. The last two cases are performed by `rotL` and `digitToTree`.

```
deepL :: (Measured v a) ⇒ Maybe (Digit a) →
         FingerTree v (Node v a) → Digit a → FingerTree v a
deepL Nothing mid sf    = rotL mid sf
deepL (Just pr) mid sf = deep pr mid sf

rotL :: (Measured v a) ⇒
        FingerTree v (Node v a) → Digit a → FingerTree v a
rotL mid sf = case viewl mid of
    EmptyL   → digitToTree sf
    a :< mid' →Deep (measure mid <> measure sf)
                    (nodeToDigit a) mid' sf

digitToTree :: (Measured v a) ⇒ Digit a → FingerTree v a
digitToTree (One   a)       = Single a
digitToTree (Two   a b)     = deep (One a)   Empty (One b)
digitToTree (Three a b c)   = deep (Two a b) Empty (One c)
digitToTree (Four  a b c d) = deep (Two a b) Empty (Two c d)
```

The following illustration summarises the calls to the above smart constructors and helper functions.

$$\text{a } \texttt{Split} \text{ calls} \begin{cases} \texttt{digitToTree} & \rightsquigarrow \texttt{deep} \\ \texttt{deepL} & \begin{cases} \texttt{deep} & \\ \texttt{rotL} & \begin{cases} \texttt{Deep(<>)pr mid sf} & \\ \texttt{digitToTree} & \rightsquigarrow \texttt{deep} \end{cases} \end{cases} \end{cases}$$

Since building up the subtrees is bottom up, it takes $\mathcal{O}(\log n)$ to get level zero. Then, taking the amount of `<>` operators into account, being `deep` the largest with two[4], the runtime for `Split` takes is $2 \times \mathcal{O}(\log n) \times \mathcal{O}(\texttt{<>})$ per subtree. Figure 3.18 illustrates the subtrees construction via `Split`.

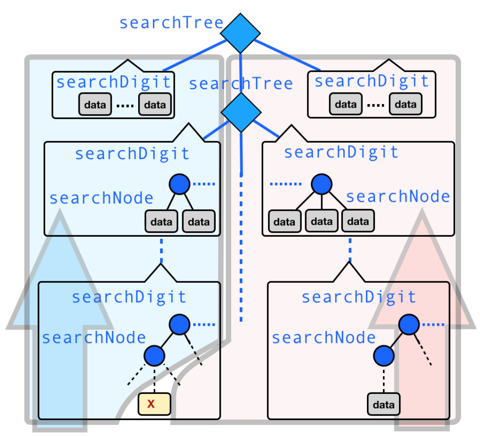

Figure 3.18: Search as a `Split` of a FT into two subtrees and an element

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 $\bowtie$ 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`.

[4] see 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

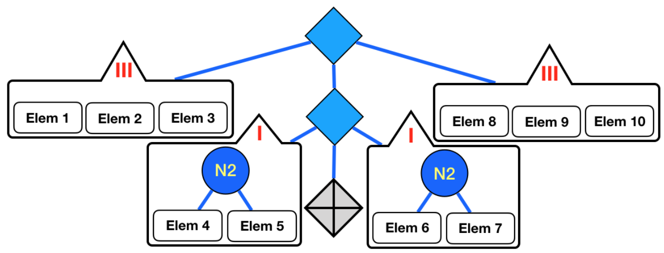

Figure 3.19: Ordered-set via FT, data (`Elem` type) at the leaves

By applying rule [`measure.Elem`] we obtain the initial monoidal annotations (`Key`s), as seen in Figure 3.20

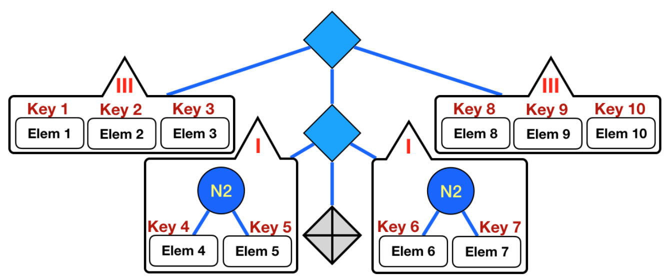

Figure 3.20: Ordered set via FT, initial monoidal annotations

Finally, in Figure 3.21 it is shown the application of the monoidal binary operation [<>.`OrdFT`] all over the tree.

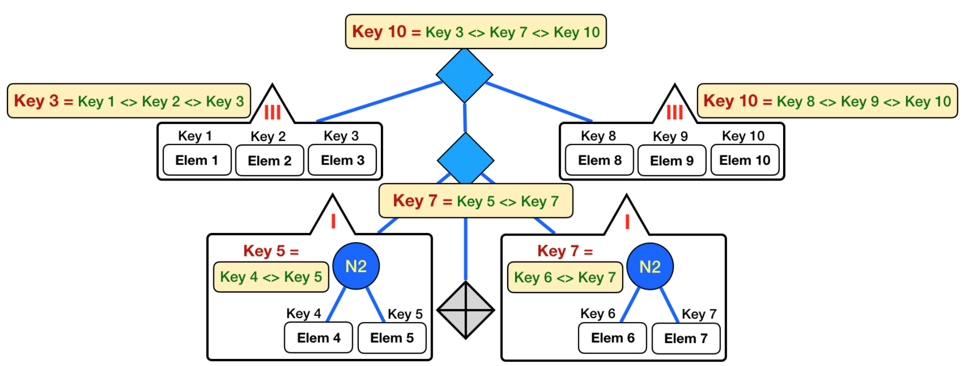

Figure 3.21: Ordered set via FT, data and monoidal annotations in the entire structure

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.

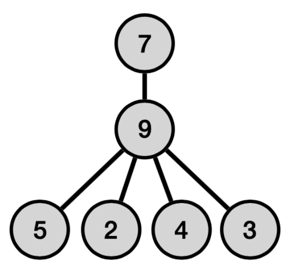

Figure 4.1: An input tree

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 ith occurrence of v, into the sequence `ET`. An instance of this procedure to the input tree in Figure 4.1 results in `ET(c) =` $o_{7_1} o_{9_1} o_{5_1} o_{9_2} o_{2_1} o_{9_3} o_{4_1} o_{9_4} o_{3_1} o_{9_5} o_{7_2}$. 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, $\mathtt{ET}(T_2)$ is given by the interval of $\mathtt{ET}(T)$ $o_{b_1}, \ldots, o_{b_2}$ and $\mathtt{ET}(T_1)$ is given by splicing out of $\mathtt{ET}(T)$ 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 $\mathtt{ET}(T_2)$ 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 $\texttt{ET}(T_1)$ requires two *implicit* look up operations, one for o_{b_1} and one for o_{a_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 $\texttt{ET}(T_s)$

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 $\texttt{ET}(T')o_{a_n}$ into $\texttt{ET}(T)$ 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 $\texttt{ET}(T')$.

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

6. Immutability is not preserved for $\texttt{ET}(T)$ 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 *catenate* and *append* are interchangeable. Tarjan in [7] defines the link operation as follows,

> ... suppose $link(\{v, w\})$ 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 $L_1^2, L_1^1, [(v, w)], L_2^2, L_2^1, [(w, v)]$ 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 reroot 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 `link` ing in ETT-HK where only the second (T') tree is rerooted.

> ... perform $cut(\{v, w\})$. 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 ... Tarjan [7]

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 `reroot`ing 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 `[roseTree]` in Section 3.2.

```
rt2et :: (Eq a) ⇒ Tree a → [(a,a)]
rt2et (Node x ts) = case ts of
  []        → [(x,x)]                          -- singleton case [rt2et.1]
  (t':ts') → root ++                           -- tree length > 1 [rt2et.2]
           concat ( map (λt→pref t ++ rt2et t ++ suff t) ts ) -- [rt2et.2]
    where
     pref v = [(x,rootLabel v)]     -- [rt2et.3]
     suff v = [(rootLabel v,x)]     -- [rt2et.4]
     root = [(x,x)]
```

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

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

- `foldr (◁) Empty`: application of $\triangleleft$ to the sequence starting from `Empty`
- `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 `link` and `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 `Data.Set` [45] and measurement definition as follows

```haskell
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 `insert`ion 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

```haskell
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.

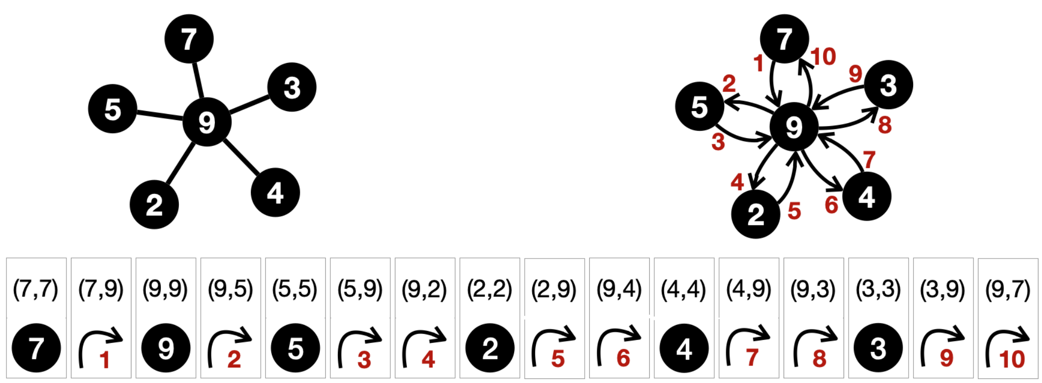

Figure 4.2: Input tree (top left) is Euler-tour numbered (top right). Ett (bottom) is generated by `rt2et`

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.

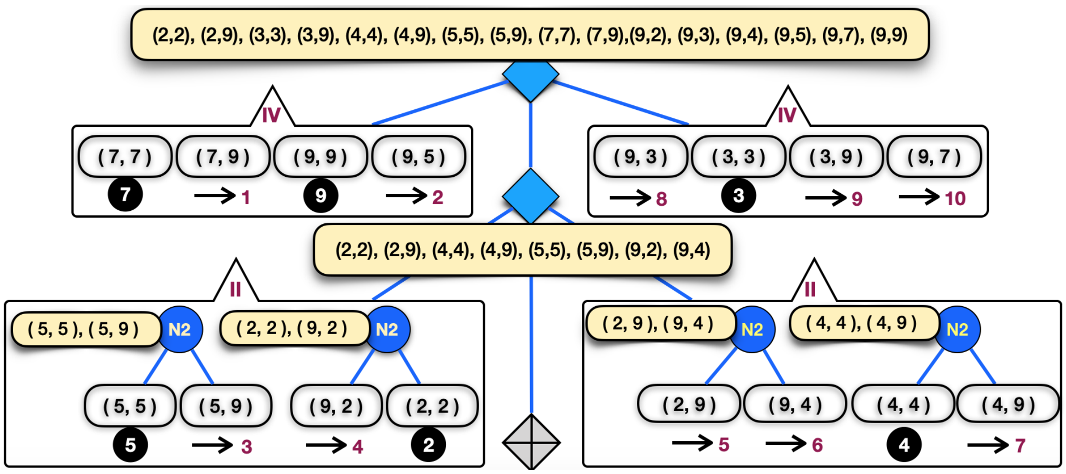

Figure 4.3: FunEtt for the Euler-tour tree representation from input tree in Figure 4.2

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.

Function	Complexity
`viewl`, `viewr`, $\lhd$, $\rhd$	$\mathcal{O}(\texttt{<>})$
$\bowtie$, `split`, `search`	$\mathcal{O}(\log n) \times \mathcal{O}(\texttt{<>})$

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 $\mathcal{O}(\texttt{<>})$ 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.

Function	Complexity
`viewl`, `viewr`, $\triangleleft$, $\triangleright$	$\mathcal{O}(m \times (\log(n/m) + 1))$
$\bowtie$, `split`, `search`	$\mathcal{O}(\log n) \times \mathcal{O}(m \times (\log(n/m) + 1))$

Table 4.2: Summary of FT operations applied to FunEtt when set-union is the monoidal annotation

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

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

and the following equation when $m = n$,

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

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 FT prior to the application of `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),

m	n	`viewl`, `viewr`, $\triangleleft$, $\triangleright$ $m\left(\log \frac{n}{m}\right)$	result	yields to bound
1	10,000	$1\left(\log \frac{10{,}000}{1}\right)$	4	$\Omega(\log n)$
10	10,000	$10\left(\log \frac{10{,}000}{10}\right)$	30	$\Theta(m \log \frac{n}{m})$
100	10,000	$100\left(\log \frac{10{,}000}{100}\right)$	200	$\Theta(m \log \frac{n}{m})$
1,000	10,000	$1,000\left(\log \frac{10{,}000}{1{,}000}\right)$	1,000	$\Theta(m \log \frac{n}{m})$
10,000	10,000	$\mathcal{O}(n)$	10,000	$\mathcal{O}(n)$

Table 4.3: Bounds of FT operations applied to FunEtt for `viewl`, `viewr`, $\triangleleft$ and $\triangleright$ cases

Similarly, when applying the same analysis to the second set of operations from Table 4.2, we have,

m	n	$\bowtie$, `split`, `search` $\log n \times m \left(\log \frac{n}{m}\right)$	result	yields to bound
1	10,000	$\log 10,000 \times 1 \left(\log \frac{10{,}000}{1}\right)$	16	$\Omega(\log^2 n)$
10	10,000	$\log 10,000 \times 10 \left(\log \frac{10{,}000}{10}\right)$	120	$\Theta(m \log n \log \frac{n}{m})$
100	10,000	$\log 10,000 \times 100 \left(\log \frac{10{,}000}{100}\right)$	800	$\Theta(m \log n \log \frac{n}{m})$
1,000	10,000	$\log 10,000 \times 1,000 \left(\log \frac{10{,}000}{1{,}000}\right)$	4,000	$\Theta(m \log n \log \frac{n}{m})$
10,000	10,000	$\log 10,000 \times \mathcal{O}(n)$	40,000	$\mathcal{O}(n \log n)$

Table 4.4: Bounds of FT 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 ⇒ a → a → FunETT a → 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 →     -- (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 pred₂ left ) of
     -- FUNETT: explicit search for (w,v) on left subtree
6.       Position leftL _ rightL →
        -- 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 ⋈ right)
            -- ETT-T : L¹[(w,v)]L²[(v,w)]L³ (symmetrical case)
```

```
                -- ETT-T result: L², L¹ and L³ are catenated
8.          _               → -- FUNETT: (w,v) is on the right subtree
9.             case (search pred₂ right) of
             -- FUNETT: explicit search for (w,v) on right subtree
10.                Position leftR _ rightR →
                -- ETT-T: split L before and after (v,w) and (w,v)
                -- (w,v) is discarded by the wildcard _ (2nd edge deletion)
11.                   Just (leftR, left ⋈ rightR)
                   -- ETT-T : L¹[(v,w)]L²[(w,v)]L³ (initial case)
                   -- ETT-T result: L², L¹ and L³ are catenated
12.                _ →  error "ETT malformed" -- (v,w) found but not (w,v)
13.     _ → case (search pred₂ tree) of -- 2nd case for (search pred tree)
14.     Position _ _ _ → error "ETT malformed" --(w,v) found but not (v,w)
15.     _              → Nothing  -- as neither (v,w) nor (w,v) ∉ tree
16.   where
17.    pred tree _  = (member (v,w)) tree
18.    pred₂ tree _ = (member (w,v)) tree
```

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 $\mathcal{O}(1)$. So, following the bounds stated in Table 4.4 we have

LOC	expression	runtime
3	`case (search pred tree) of`	$\Theta(m \log n \log \frac{n}{m})$
5	`case (search pred left) of`	$\Theta(m \log n \log \frac{n}{m})$
7	`Just (rightL, leftL ⋈ right)`	$\Theta(m \log n \log \frac{n}{m})$
9	`case (search pred right) of`	$\Theta(m \log n \log \frac{n}{m})$
11	`Just (leftR, left ⋈ rightR)`	$\Theta(m \log n \log \frac{n}{m})$
17,18	`(member (v',w')) tree'`	$\mathcal{O}(\log n)$

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.

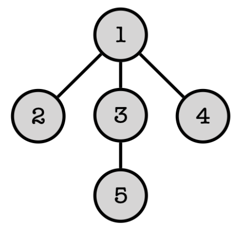

Figure 4.4: Input tree example for the `cutTree` operation

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, $\mathcal{O}(n \log n)$ is the upper bound per `cutTree` operation, provided that all sets involved are of the same size. That is, $\Omega(\log^2 n) \leq \Theta($ `cutTree` $) \leq \mathcal{O}(n \log n)$ 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 $\triangleleft$, $\triangleright$, `search` and $\bowtie$ 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)
       ,(3,1),(1,4),(4,4),(4,1)]
```

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

```
cutTree 1 3 tree = if 1 == 3 then Nothing else
  case (search pred tree) of
    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,

```
left = [(1,1),(1,2),(2,2),(2,1)]
```

and

```
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 → --unsuccessful: (3,1) ∉ left subtree
7.           Just (rightL, leftL ⋈ 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.         _ → -- second case for (search pred2 left)
9.           case (search pred2 right) of
10.           Position leftR _ rightR → -- (3,1) found in subtree right
11.             Just (leftR, left ⋈ rightR) -- two FTs are returned
12.           _ → error "ETT malformed" --(1,3) found but not (3,1)
13.     _ → case (search pred2 tree) of -- 2nd case for (search pred tree)
14.       Position _ _ _ → error "ETT malformed" --(3,1) found but not (1,3)
15.       _              → Nothing  -- neither (1,3) nor (3,1) ∉ 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 `tree`.
- `error` for the case when pair `(1,3)` is in finger tree `tree` but `(3,1)` is not.
- `error` for the case when pair `(3,1)` is in finger tree `tree` but `(1,3)` is not.
- `Just (rightL, leftL` ⋈ `right)`, not the case for this example.
- `Just (leftR, left` ⋈ `rightR)`, the result for this example which is

```
Just( [(3,3),(3,5),(5,5),(5,3)]
    , [(1,1),(1,2),(2,2),(2,1),(1,4),(4,4),(4,1)] )
      |----------left--------|⋈|------rightR----|
```

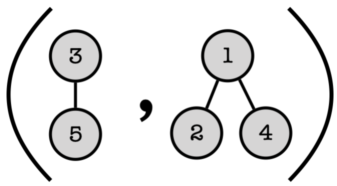

Figure 4.5: Resulting trees after `cutTree` operation is applied to input tree in Figure 4.4.

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, `link`ing 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 *link* in Ett-T, operations `search`, `split`, ◁, ▷, ⋈ from Chapter 3 and `member` (testing membership for sets) operation from [45], we have

```
linkTree :: Ord a ⇒ a → FunETT a → a → FunETT a → Maybe (FunETT a)
linkTree v tv w tw = if v == w then Nothing else
 case (pairIn (v,v) tw, pairIn (w,w) tv) of
   (True, _   ) → Nothing  -- v ∈ tw
   ( _ , True) → Nothing  -- w ∈ tv
   (False,False) →
   case (pairIn (v,v) tv, pairIn (w,w) tw) of
    (False, _    ) → Nothing
    (_    , False) → Nothing
    (True , True ) → Just $
     let tv'                   = reroot tv v
         Position left _ right = search pred tw
     in  ((left ▷ (w,w)) ▷ (w,v)) ⋈ tv' ⋈ ((v,w) ◁ right)
  -- in FunEtt we append L_2^1, [(v,v)], [(v,w)], t'_v, [(w,v)], L_2^2
  -- in Ett-T we append L_1^2, L_1^1, [(v,w)], L_2^2, L_2^1, [(w,v)]
 where
   pred tree _ = (member (w,w)) tree -- is (w,w) ∈ tree ?
```

Function pairIn test membership for both vertices v and w rather than split or search on/for them.

```
1. pairIn :: (Ord a) ⇒ a → FunETT a → Bool
2. pairIn pair tree = case (search pred tree) of
3.   Position _ _ _ → True
4.   _              → False
5.  where
6.    pred tree' _ = (member pair) tree'
```

Finally, changing the root on a FunEtt is as follows

```
1. reroot :: Ord a ⇒ FunETT a → a → FunETT a
2. reroot tree vertex =
3.  let (left,right) = split pred tree
4.  in  right ⋈ left  -- as in ETT-T when swapping L_1^1, L_1^2 to L_1^2, L_1^1
5.  where
6.      pred tree' = (member (vertex,vertex)) tree'
```

Performance of linkTree

Alike cutTree, operation 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 $\mathcal{O}(1)$. So, following the bounds stated in Table 4.4 we have

LOC	function	expression	runtime
2	pairIn	`case (search pred tree) of`	$\Theta(m \log n \log \frac{n}{m})$
6	pairIn	`(member pair) tree'`	$\mathcal{O}(\log n)$
3	reroot	`split pred tree`	$\Theta(m \log n \log \frac{n}{m})$
4	reroot	`right ⋈ left`	$\Theta(m \log n \log \frac{n}{m})$
6	reroot	`(member (vertex,vertex)) tree'`	$\mathcal{O}(\log n)$
12	linkTree	`search pred tw`	$\Theta(m \log n \log \frac{n}{m})$
13	linkTree	`((left ▷ (v,v)) ▷ (v',w'))` `⋈ tv' ⋈ ((v',w') ◁ right)`	$\Theta(m \log n \log \frac{n}{m})$
15	linkTree	`(member (v,v)) tree`	$\mathcal{O}(\log n)$

Table 4.6: Bounds of 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, $\Omega(\log^2 n) \leq \Theta($ 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 $\triangleleft$, $\triangleright$, search and $\bowtie$ 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

```
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.

```
pairIn (2,2) tw = case (search pred tw) of
  Position _ _ _ → True
  _              → False
 where
   pred tree' _ = (member (2,2)) tree'
```

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.

```
reroot tv 3 =
 let (left,right) = split pred tv
 in  right ⋈ left
 where
     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 $\mathcal{O}(15)$ 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.

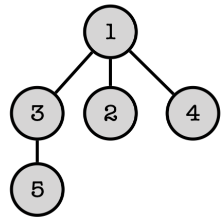

Figure 4.6: Resulting tree after `linkTree` operation is applied to input trees in Figure 4.5.

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 `link`ing and `cut`ting 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

Operation, description	Ett-HK	Ett-T	FunEtt
Space for Ett representation	$2n-1$	$3n-2$	$3n-2$
`cut` look ups	4	2	2
`cut` splits	2	2	2
`cut` append	1	1	1
`link` look ups	4	2	4
`link` splits	2	2	2
`link` append	3	5	3
Total operations in specification	16	14	14

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.

```
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 differents 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,

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

and the tree as

```
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).

```
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 .

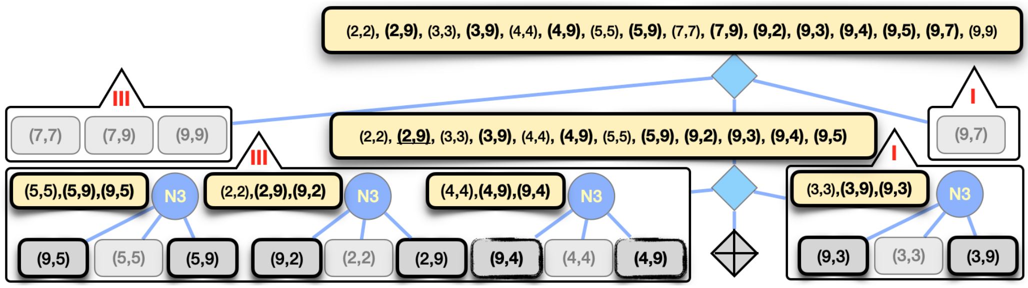

Figure 5.1: FULL FT with *repeated* edges identified with thicker border for data on the leaves and underlined and coloured for edges on sets

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 $3n - 2$.

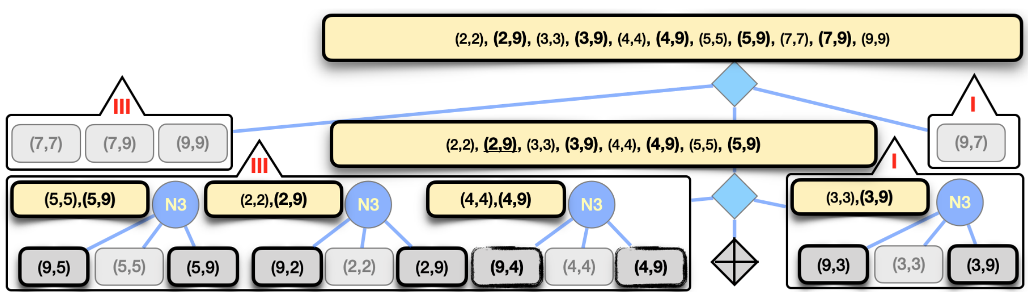

Figure 5.2: FULL FT with *repeated* edges not longer stored in the corresponding sets, however, they remain in the ETT sequence

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

```
instance (Integral a, Ord a) ⇒ Measured (MultiSet a) (MSet a) where
```

```
  measure (MSet p) = whichSet p

whichSet p@(x,y)
  | x==y && even x = MultiSet (S.insert x S.empty) S.empty S.empty
  | x==y && odd  x = MultiSet S.empty (S.insert x S.empty) S.empty
  | otherwise      = MultiSet S.empty S.empty (S.insert p S.empty)
```

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

5.1.2 Full dynamic trees operations

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

```
rootMSet :: Integral a => TreeMSet a -> Maybe a
rootMSet tree = case viewl tree of
  EmptyL      -> Nothing
  MSet x :< _ -> Just (fst x)
```

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

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

```
searchMSet :: (Integral a, Measured (MultiSet a) b) =>
              (a,a) -> FingerTree (MultiSet a) b -> SearchResult (MultiSet a) b
searchMSet p@(x,y) ftree
  | x==y && even x = let predicate setx _
  = (S.member x) (getEvens setx) in search predicate ftree
  | x==y && odd  x = let predicate setx _
  = (S.member x) (getOdds  setx) in search predicate ftree
  | otherwise      = let predicate setx _
  = (S.member p) (getEdges setx) in search predicate ftree
```

Function `search` within `searchMSet` is the same function described in Section 3.6.7. Therefore, performance for `searchMSet` is $\Omega(\log^2 n) \leq \Theta($

`searchMSet`) $\leq \mathcal{O}(n \log n)$ where n is the total number of vertices of the forest where `searchMSet` takes place.

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

```
nodeInMSet :: Integral a ⇒ a → ForestMSet a → Maybe (TreeMSet a, a)
nodeInMSet v (ForestMSet _ _ ft) =
  case (searchMSet (v,v) ft) of
    Position _ tree _ →
      case (rootMSet tree) of
        Nothing    → Nothing        -- empty tree, no root
        Just rootT → Just (tree, rootT)
    _              → Nothing        -- vertex v not in forest
```

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

`connected` operation in Full dynamic trees

Testing connectivity in Full dynamic trees is via operation `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.

```
connectedMSet :: Integral a ⇒ a → a → ForestMSet a
              → (Bool, Maybe (TreeMSet a, a, TreeMSet a, a) )
connectedMSet x y f =
  case (nodeInMSet x f, nodeInMSet y f) of
    (Nothing         , _            ) → (False, Nothing)
    (_               , Nothing      ) → (False, Nothing)
    (Just (tx,rx)    , Just (ty,ry)) →
     if rx == ry then (True,  Just(tx,rx,tx,rx))
                 else (False, Just(tx,rx,ty,ry))
```

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

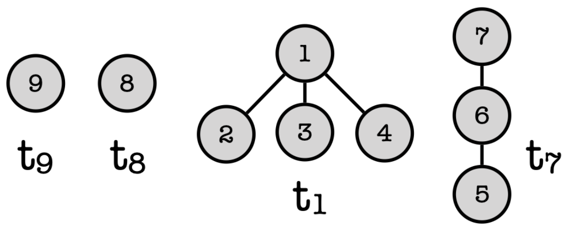

Figure 5.3: Input forest example with four input trees

A `connected` example

Let `forest` be the collection of `t1, t7, t8` and `t9` input trees, depicted in Figure 5.3. Following is the `forest` monoidal annotation.

```
MultiSet { getEvens = fromList [2,4,6,8]
         , getOdds  = fromList [1,3,5,7,9]
         , getEdges = 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 `nodeInMSet`.

```
2. nodeInMSet 7 (ForestMSet _ _ forest) =
3.   case (searchMSet (7,7) forest) of
4.     Position _ tree _ →
5.       case (rootMSet tree) of
6.         Nothing    → Nothing
7.         Just rootT → Just (tree, rootT)
8.     _              → Nothing
```

We assume function `searchMSet` as correct as it relies on function `search` from [8]. Since 7 is a member of `forest` monoidal annotation, line (7) above matches the case analysis and that returns the pair `(tree, rootT)`, which for the case of vertex 7 is `(t7,7)`. For testing connectivity, we try vertices 7 and 8 within `forest`.

```
2. connectedMSet 7 8 forest =
3.   case (nodeInMSet 7 forest, nodeInMSet 8 forest) of
4.     (Nothing        , _         ) → (False, Nothing)
5.     (_              , Nothing   ) → (False, Nothing)
6.     (Just (t7,7)    , Just (t8,8)) →
7.       if 7 == 8 then (True,  Just(tx,rx,tx,rx))
8.                 else (False, Just(t7,7,t8,8))
```

After both `nodeInMSet` functions are evaluated (line 2 above), the pair `Just (t7,7), Just (t8,8))` matches the case analysis. Since roots 7 and

8 are different, `connectedMSet` returns the pair `(False, Just(t7,7,t8,8)`. Assume now we are testing connectivity for vertices 2 and 4. All of the above is satisfied except the last bit where `connectedMSet` returns the pair `(True, Just(t1,1,t1,1)`.

`link` operation in FULL dynamic trees

Update dynamic operations for FULL are described through `link` and `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.

```
link :: Integral a ⇒ a → a → ForestMSet a → ForestMSet a
link x y forest@(ForestMSet nnodes size ft) =
      case connectedMSet x y forest of
        (False, Just(tx,rx,ty,ry)) → linkAll (linkTreeMSet x tx y ty)
        _           → forest -- x and y are currently connected in forest
 where
  linkAll tree = ForestMSet nnodes (size + 1) (tree ◁ (lforest ⋈ rforest))
  Position lforest' _ rforest'
       = searchMSet (x,x) ft                        -- tx is left behind
  Position lforest  _ rforest
       = searchMSet (y,y) (lforest' ⋈ rforest')   --ty is left behind
```

A new tree (`tree`) is built from `linkTreeMSet` with input trees (`tx` and `ty`) provided by `connectedMSet` once this function tested positive for `x` and `y` being connected at `forest`. The new forest is built via function `linkAll` which has left behind trees `tx` and `ty`. New tree is inserted with ◁ to the new subforest (`lforest ⋈ rforest`). Finally, the size of the forest is updated when `(size + 1)`.
Unless the above `forest` is *one-tree* forest, runtime complexity for `link` is determined by local and global monoidal annotations. That is, `linkTreeMSet` runs in $\Theta(m \log n \log \frac{n}{m})$ where m and n are the sizes of the sets evaluated on `tx` and `ty` trees, whereas the sets evaluated in `connectedMSet`, `searchMSet`, ◁, and ⋈ are regarded to the entire `forest`. Now, since `link` has a constant number of operations, the runtime for `link` is $\Omega(\log^2 n) \leq \Theta($ `link` $) \leq \mathcal{O}(n \log n)$, where n is the total amount of nodes in the forest, not at the trees when applying `linkTreeMSet`.

A `link` example

Let `forest` be the collection of `t1, t7, t8` and `t9` input trees, depicted in Figure 5.3. Assume we try to `link 2 4` in `forest`.

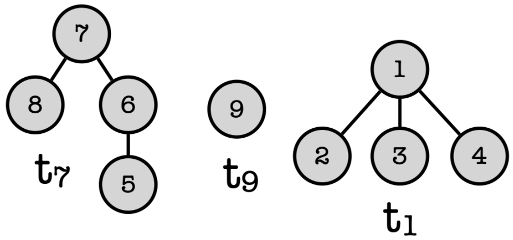

Figure 5.4: FULL `link` applied to vertices 8 and 7 and input forest from Figure 5.3.

```
2. link 2 4 forest@(ForestMSet nnodes size ft) =
3.       case connectedMSet 2 4 forest of
4.         (False, Just(tx,rx,ty,ry)) → linkAll (linkTreeMSet x tx y ty)
5.         _           → forest
```

Case analysis, in line 5, returns `_`, which is the lazy-evaluated form for `(True, Just(t1,1,t1,1))`. It is not necessary to express such a form completely as the `link` is failed due both vertices coexist under the same tree. Now, we try `link 8 7` in `forest`.

```
2. link 8 7 forest@(ForestMSet nnodes size ft) =
3.       case connectedMSet 8 7 forest of
4.         (False, Just(t8,8,t7,7)) → linkAll (linkTreeMSet 8 t8 7 t8)
```

Since vertices 7 and 8 are members of different trees, line 4 in snippet above matches the case analysis and `linkAll` function is called after performing `linkTreeMSet`. We showed up in Section 4.3.3 that `linkTree` returns a valid finger tree, called here `tree`, provided input trees and vertices are also valid. Then, `linkAll` assembles the output for `link` as follows.

```
7. linkAll tree = ForestMSet nnodes (size + 1) (tree ◁ (lforest ⋈ rforest))
```

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

`cut` operation in FULL dynamic trees

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

```
edgeInMSet 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 $\Omega(\log^2 n) \leq \Theta($ `edgeInMSet` $) \leq \mathcal{O}(n \log n)$, 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 $\Omega(\log^2 n) \leq \Theta($ `cut` $) \leq \mathcal{O}(n \log n)$, where n is the total amount of nodes in the forest.

A `cut` example

Let `forest` be the collection of `t1`, `t7` and `t9` 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
```

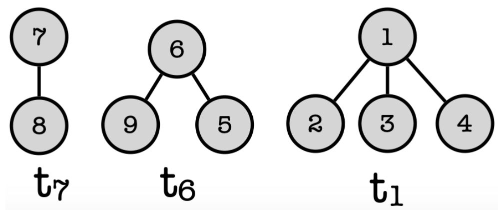

Figure 5.5: FULL `cut` applied to vertices 7 and 6 and input forest from Figure 5.4.

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. cutMSet 7 6 forest@(ForestMSet nnodes size ft) =
3.   case edgeInMSet (7,6) forest of
4.     Nothing    → forest
5.     Just (t7,ltFor,rtFor)
6.           → buildForest (cutTreeMSet 7 6 t7) ltFor rtFor
7.   where
8.     buildForest (leftTree,rightTree) lFor rFor
9.        = ForestMSet nnodes (size - 1)
10.                    (leftTree ◁ rightTree ◁ (lFor ⋈ rFor))
```

`Just (t7,ltFor,rtFor)` matches `edgeInMSet` case analysis in lines 3 and 5 above. `ltFor` and `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 `leftTree` and `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).

Operation	best case	worst case	context
`root`	$\Omega(\log n)$	$\mathcal{O}(n)$	trees
`connected`	$\Omega(\log^2 n)$	$\mathcal{O}(n \log n)$	forest
`cut`	$\Omega(\log^2 n)$	$\mathcal{O}(n \log n)$	forest
`link`	$\Omega(\log^2 n)$	$\mathcal{O}(n \log n)$	forest

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`.

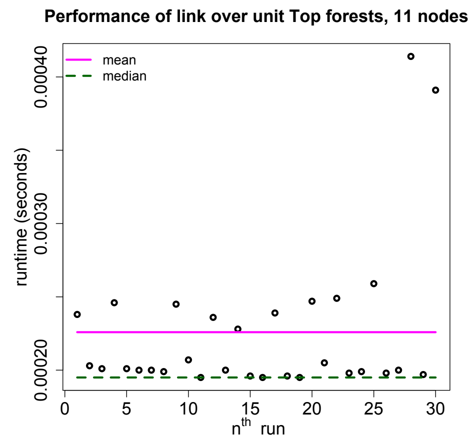

Figure 5.6: Sample of plotting for an initial running

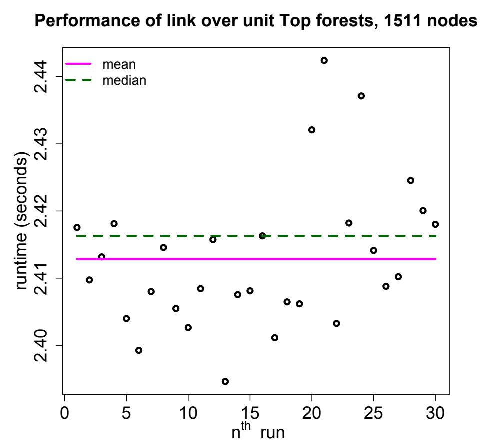

Figure 5.7: Sample of plotting for an intermediate running

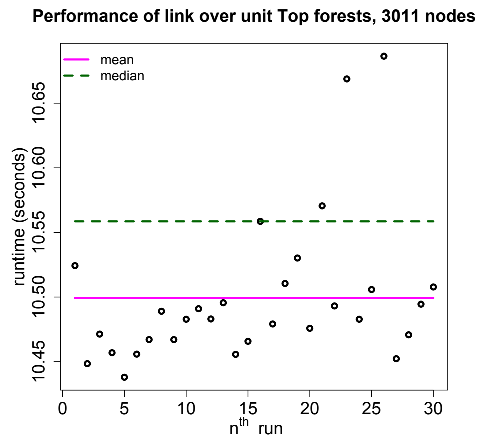

Figure 5.8: Sample of plotting for a final running

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, especifically, 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`.

```
forest = foldr (◁) emptyForestMSet (map (λx→Single (MSet(x,x))) 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.

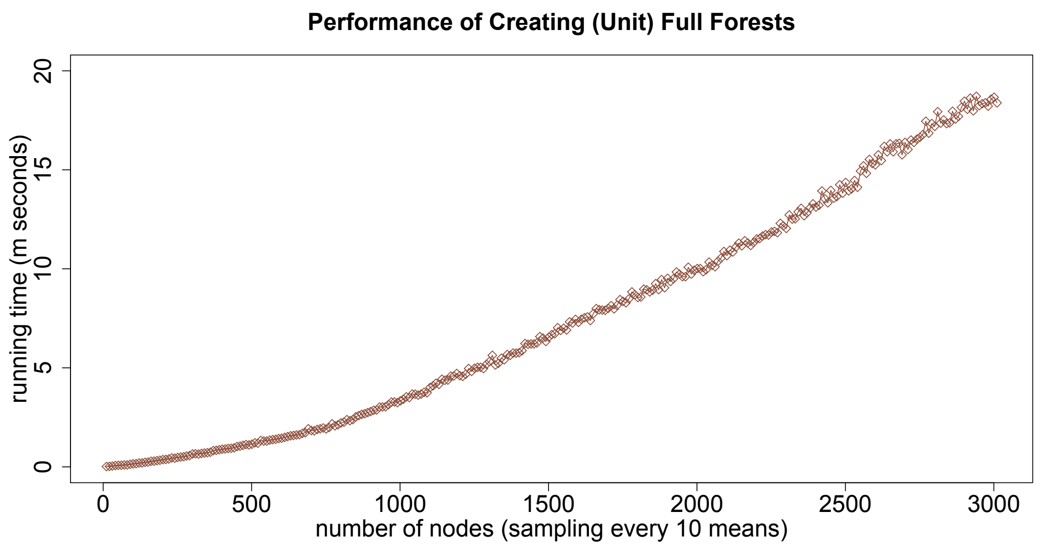

Figure 5.9: Performance of *unit* FULL forest measured by means

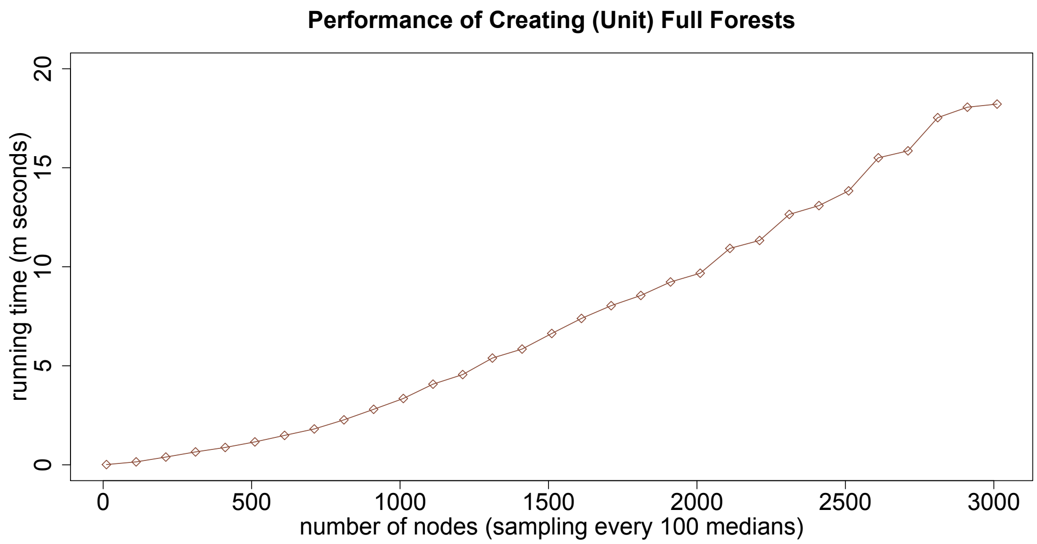

Figure 5.10: Performance of *unit* FULL forest measured by medians

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 $\triangleleft$ operator, then result is in between $\Omega(\log n)$ and $\mathcal{O}(n)$ 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.

number of nodes	*unit* runtime in milliseconds
11	0.012
111	0.148
211	0.391
...	...
2,811	17.531
2,911	18.059
3,011	18.216

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.

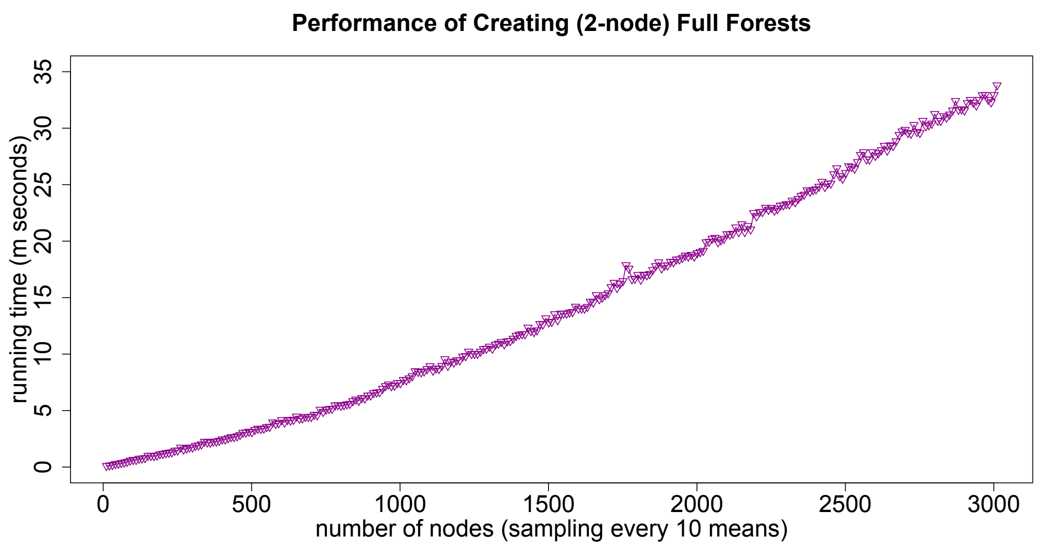

Figure 5.11: Performance of *2-node* FULL forest measured by means

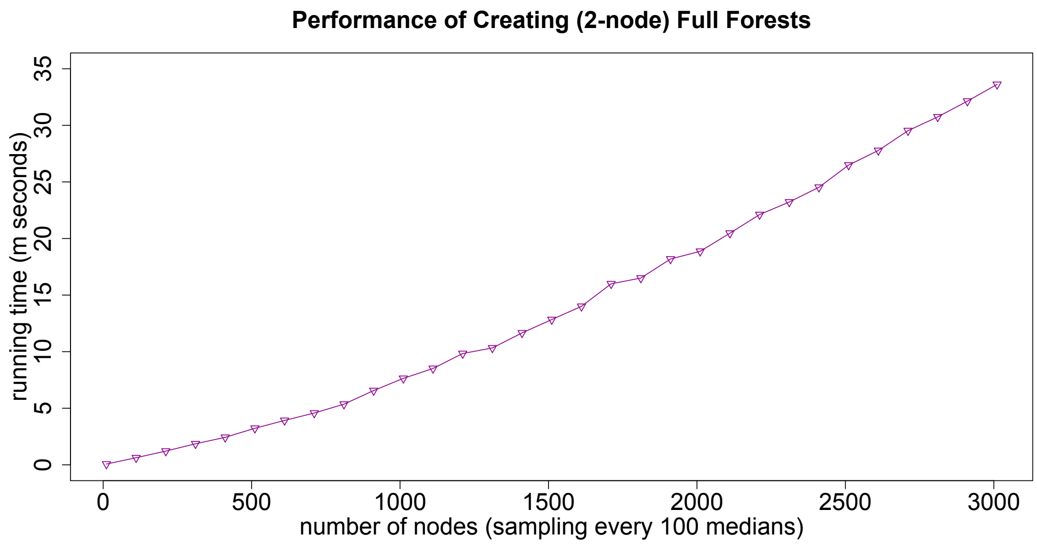

Figure 5.12: Performance of *2-node* Full forest measured by medians

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 $\mathcal{O}(n)$, 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.

number of nodes	*2-node* runtime in milliseconds
11	0.068
111	0.626
211	1.213
...	...
2,811	30.747
2,911	32.139
3,011	33.621

Table 5.3: An excerpt of tabular values for Figure 5.12.

***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.

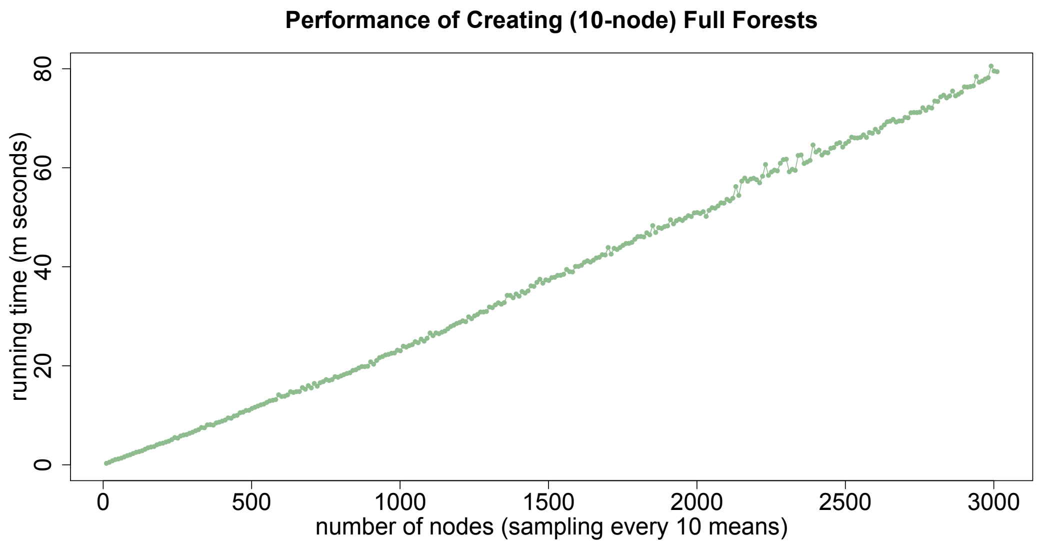

Figure 5.13: Performance of *10-node* Full forest measured by means

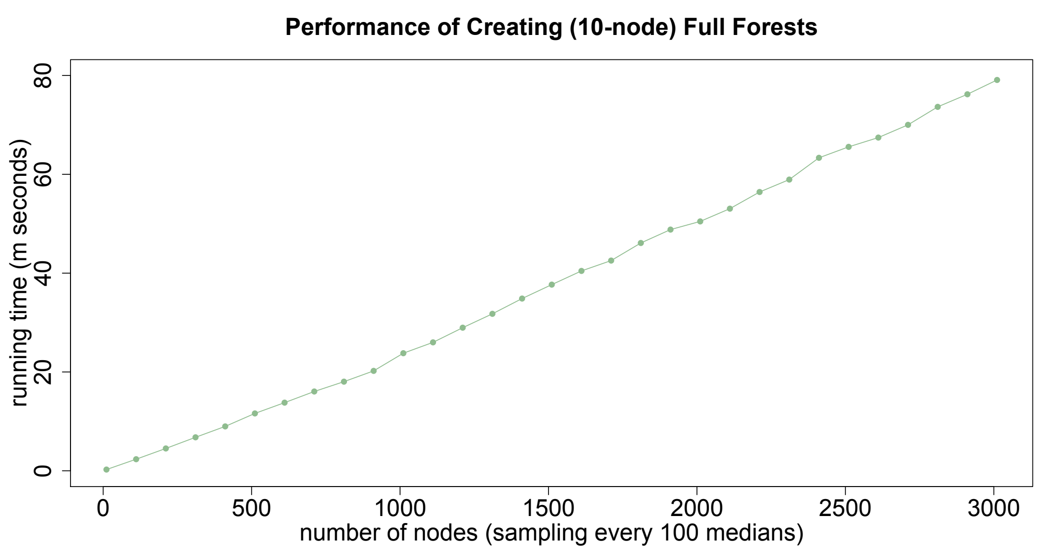

Figure 5.14: Performance of *10-node* Full forest measured by medians

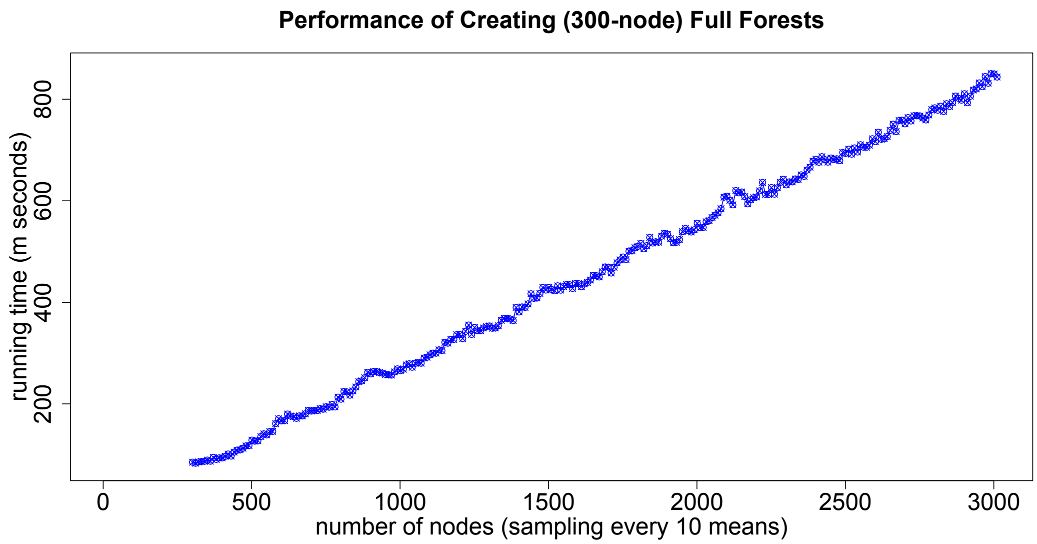

Figure 5.15: Performance of *300-node* FULL forest measured by means

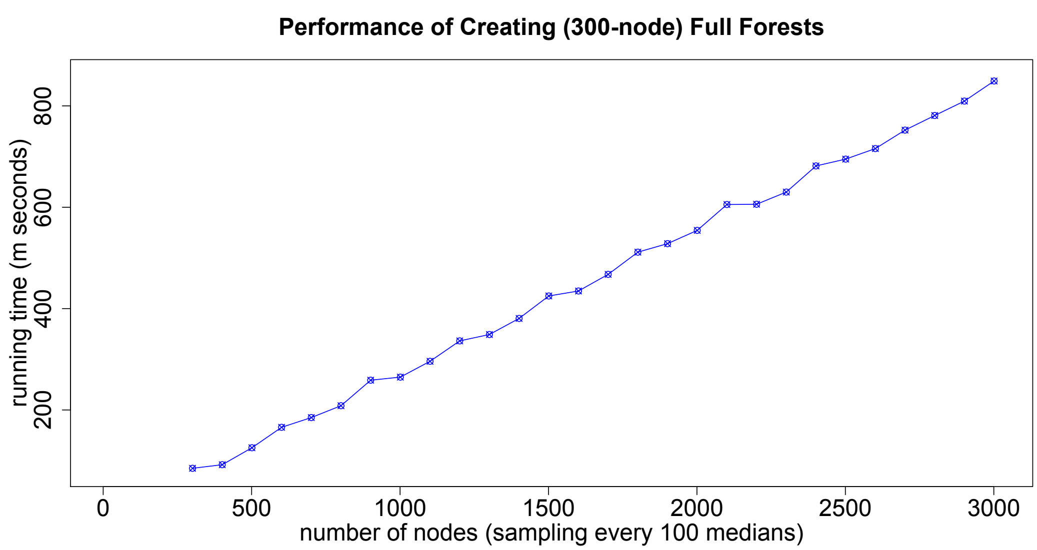

Figure 5.16: Performance of *300-node* FULL forest measured by medians

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 $\mathcal{O}(n)$, 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.

number of nodes	*10-node* runtime in milliseconds	*300-node* runtime in milliseconds
11	0.250	NA
111	2.343	NA
211	4.520	NA
311	6.776	84.895
411	8.981	91.979
511	11.608	125.420
...	...	...
2,811	73.647	780.995
2,911	76.194	809.462
3,011	79.092	849.095

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.

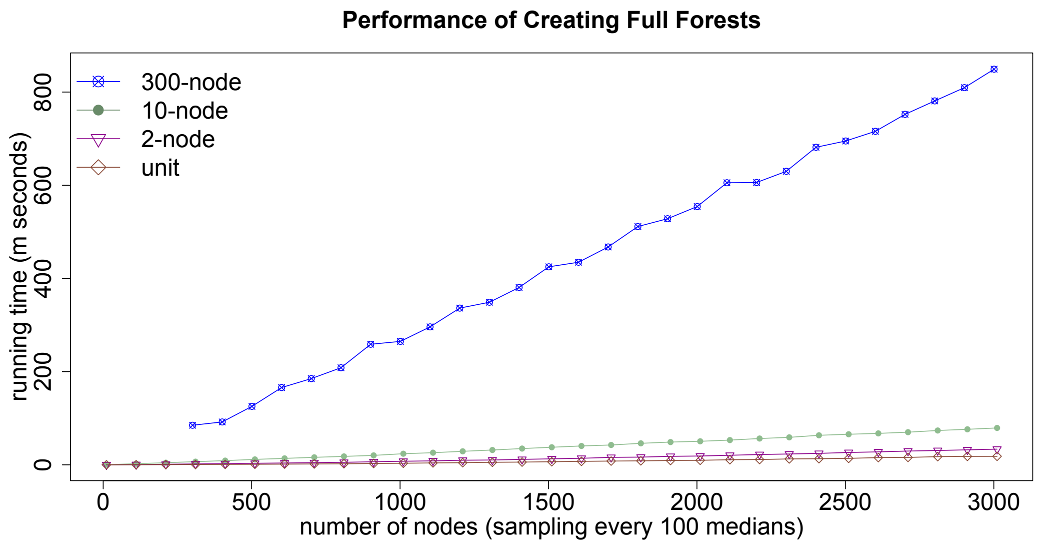

Figure 5.17: Performance of *300-node*, *10-node*, *2-node*, and *unit* FULL forests construction, measured by medians.

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 ($\triangleleft$) 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 $\mathcal{O}(n)$, 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.

number of nodes	*unit* runtime in milliseconds	*2-node* runtime in milliseconds	*10-node* runtime in milliseconds	*300-node* runtime in milliseconds
11	0.012	0.068	0.250	NA
111	0.148	0.626	2.343	NA
211	0.391	1.213	4.520	NA
311	0.651	1.857	6.776	84.895
411	0.876	2.427	8.981	91.979
511	1.155	3.228	11.608	125.420
...	...	...	...	...
2,811	17.531	30.747	73.647	780.995
2,911	18.059	32.139	76.194	809.462
3,011	18.216	33.621	79.092	849.095

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

`connect`ivity 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

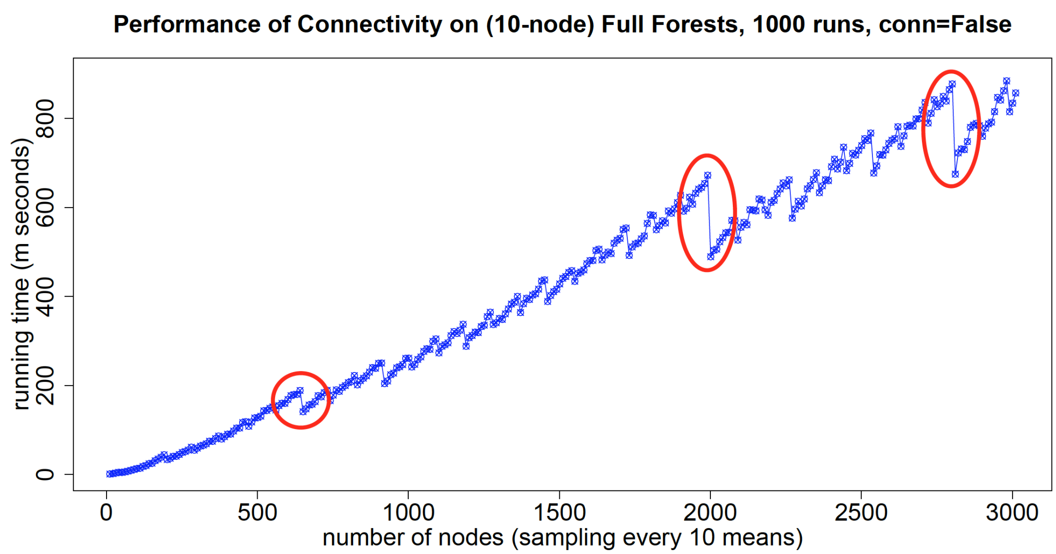

Figure 5.18: Performance of `connectedMSet` over a *10-node* FULL forest, measured by means.

`connect`ivity 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 `connect`ivity 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, $\Omega(log^2 n) \leq \Theta($ `connectMSet` $) \leq \mathcal{O}(n \log n)$, 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 $\mathcal{O}(<>)$ 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 ($\triangleleft$ and $\triangleright$) and when appending ($\bowtie$). 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` an 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".

row	nnodes	runtime	$\triangle$	forest	nops: prefix
1	631	180.268	0.678	**[III,IV,IV]**,Empty,[I,I,I]	2+3+3 = 8
2	641	188.810	8.542	**[IV,IV,IV]**,Empty,[I,I,I]	3+3+3 = 9
3	651	141.160	*-47.650*	**[II,II,II]**,Single,[I,I,I]	1+1+1 = 3
4	661	147.557	6.397	**[III,II,II]**,Single,[I,I,I]	2+1+1 = 4
5	671	156.129	8.571	**[IV,II,II]**,Single,[I,I,I]	3+1+1 = 5
6	1981	653.749	8.775	**[III,IV,IV,IV]**,Empty,[I,I,I,I]	2+3+3+3 = 11
7	1991	672.368	18.618	**[IV,IV,IV,IV]**,Empty,[I,I,I,I]	3+3+3+3 = 12
8	2001	489.392	*-182.975*	**[II,II,II,II]**,Single,[I,I,I,I]	1+1+1+1 = 4
9	2011	503.759	14.366	**[III,II,II,II]**,Single,[I,I,I,I]	2+1+1+1 = 5
10	2021	506.374	2.615	**[IV,II,II,II]**,Single,[I,I,I,I]	3+1+1+1 = 6
11	2791	864.816	8.924	**[III,IV,IV,IV]**,Single,[I,I,I,I]	2+3+3+3 = 11
12	2801	877.551	47.633	**[IV,IV,IV,IV]**,Single,[I,I,I,I]	3+3+3+3 = 12
13	2811	674.967	*-202.583*	**[II,II,II,II,I]**,Empty,[I,I,I,I,I]	1+1+1+1+0 = 4
14	2821	722.600	12.734	**[III,II,II,II,I]**,Empty,[I,I,I,I,I]	2+1+1+1+0 = 5
15	2831	731.525	25.534	**[IV,II,II,II,I]**,Empty,[I,I,I,I,I]	3+1+1+1+0 = 6

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, $\triangle$, 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 use 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.

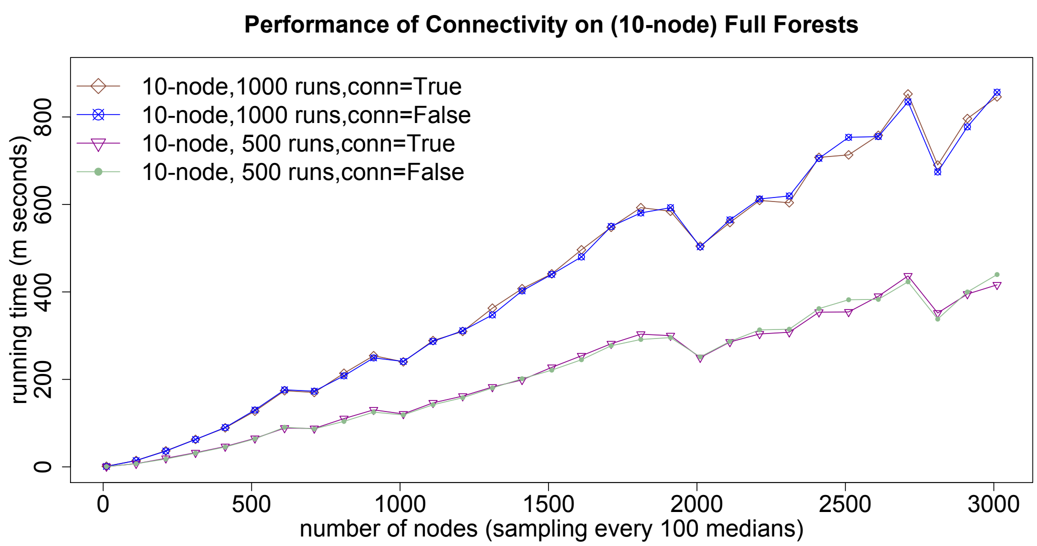

Figure 5.19: Performance of `connectedMSet` over a *10-node* FULL forest, multiple runs.

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.

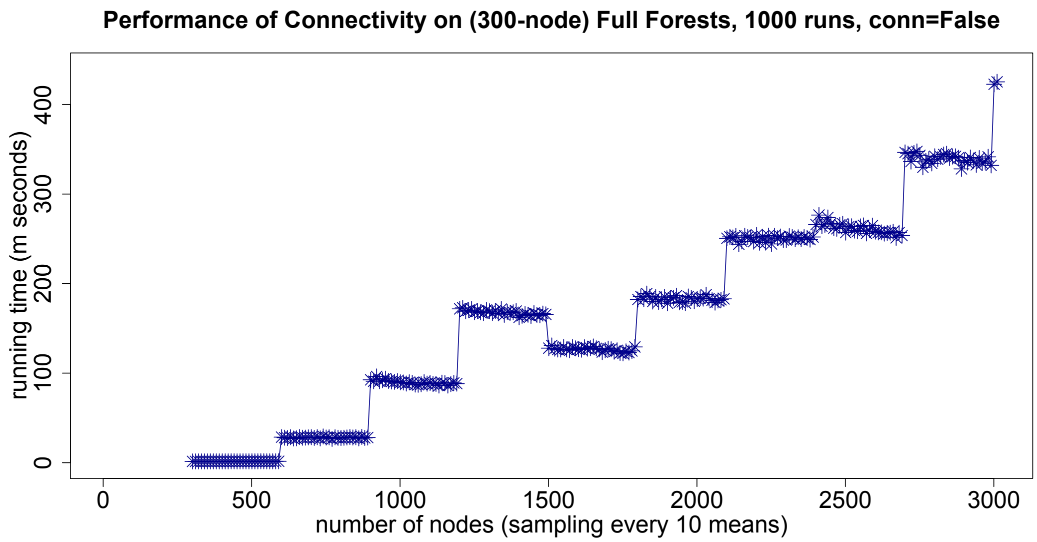

Figure 5.20: Performance of `connectedMSet` over a *300-node* FULL forest.

Experimental results of testing `connect`ivity 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.

row	nnodes	runtime	△	forest	nops: prefix
1	1481	166.530	2.1390	**[IV]**,Empty,[I]	3
2	1491	165.958	*-0.571*	**[IV]**,Empty,[I]	3
3	1501	127.873	*-38.085*	**[I,I]**,Empty,[I,I]	0
4	1511	131.063	3.190	**[I,I]**,Empty,[I,I]	0
5	1521	127.134	*-3.929*	**[I,I]**,Empty,[I,I]	0

Table 5.7: Amount of monoidal annotations in a FULL dynamic tree via its affixes for a FULL forest.

The negative value of $\triangle$ 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 $\triangle$s.

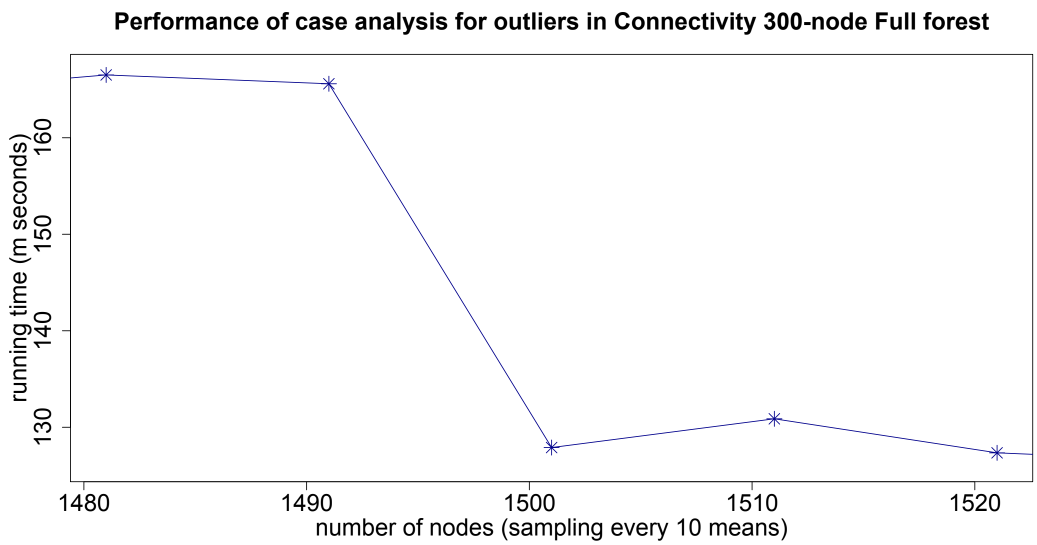

Figure 5.21: Performance of `connectedMSet` over a *300-node* FULL forest, showing one of the outliers.

Finally, in Figure 5.22 the performance for *10-node* and *300-nodes* FULL forest testing connectivity is shown.

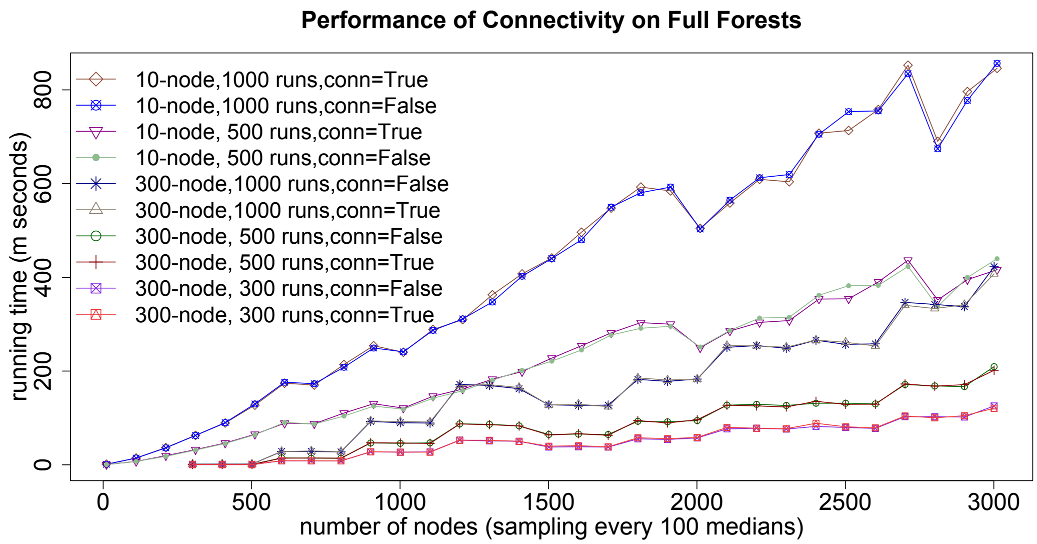

Figure 5.22: Performance of `connectedMSet` over a *10-node* and *300-node* Full forest, for different number of runs per forest.

Experimental results of `connect`ivity between *10-nod* 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 $\mathcal{O}(n \log n)$ 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.

num. nodes	*10-n* 500 millisec.	*10-n* 1K millisec.	*300-n* 300 millisec.	*300-n* 500 millisec.	*300-n* 1K millisec.
11	0.655	1.287	NA	NA	NA
111	7.185	14.536	NA	NA	NA
211	19.472	36.600	NA	NA	NA
311	32.345	62.802	0.499	0.81	1.533
411	46.450	89.199	0.460	0.712	1.454
511	65.059	127.219	0.472	0.757	1.522
...	...	...	...	...	...
2,811	351.718	690.024	100.168	167.822	333.293
2,911	395.071	796.342	104.855	170.994	341.553
3,011	416.171	845.860	120.629	201.683	407.499

Table 5.8: An excerpt of tabular values for Figure 5.22.

`link`ing 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.

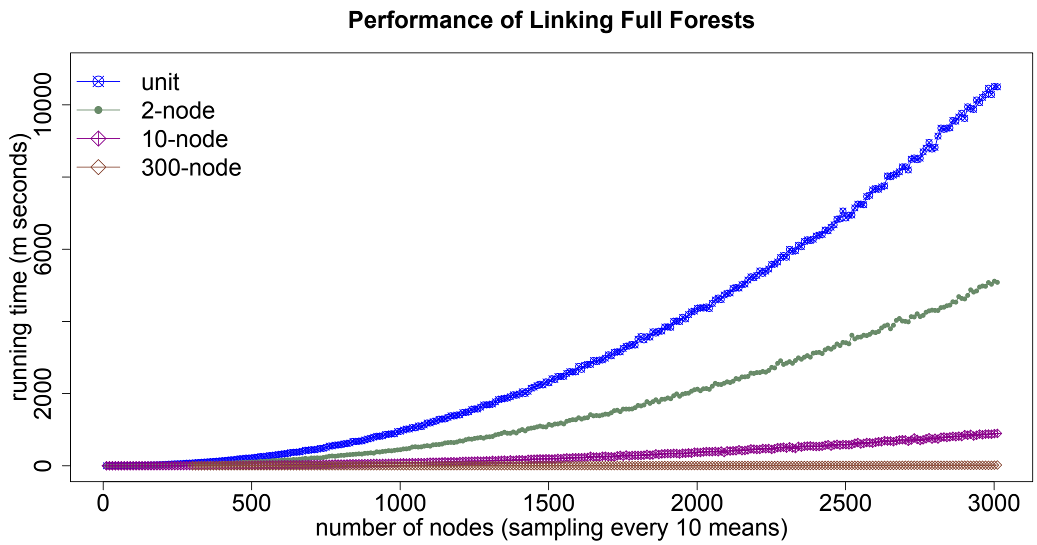

Figure 5.23: Performance of `link` operation over a *unit*, *2-node*, *10-node* and *300-node* FULL forests, sampling every 10 means.

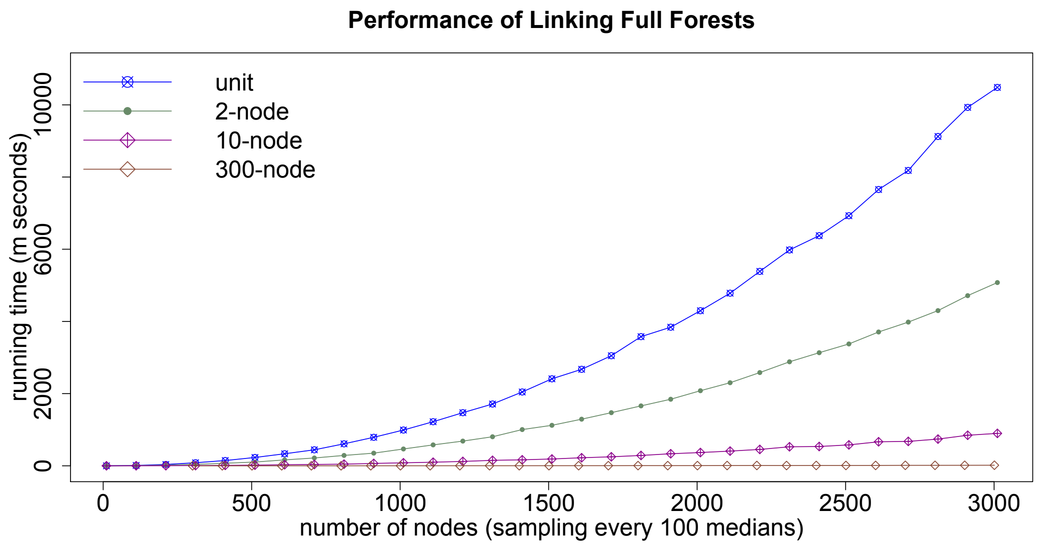

Figure 5.24: Performance of `link` operation over a *unit*, *2-node*, *10-node* and *300-node* FULL forests, sampling every 100 medians.

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 $\mathcal{O}(n \log n)$. Then, for every point plotted, n nodes are computed, hence the total performance in each curve is $\mathcal{O}(n \times n(\log n)) = \mathcal{O}(n^2 \log n)$. 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 `link`s, the forest is practically edge-full. On the other hand, the height of the host forest for the former case is $\mathcal{O}(\log n)$, 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.

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

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

`cut`ting 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.

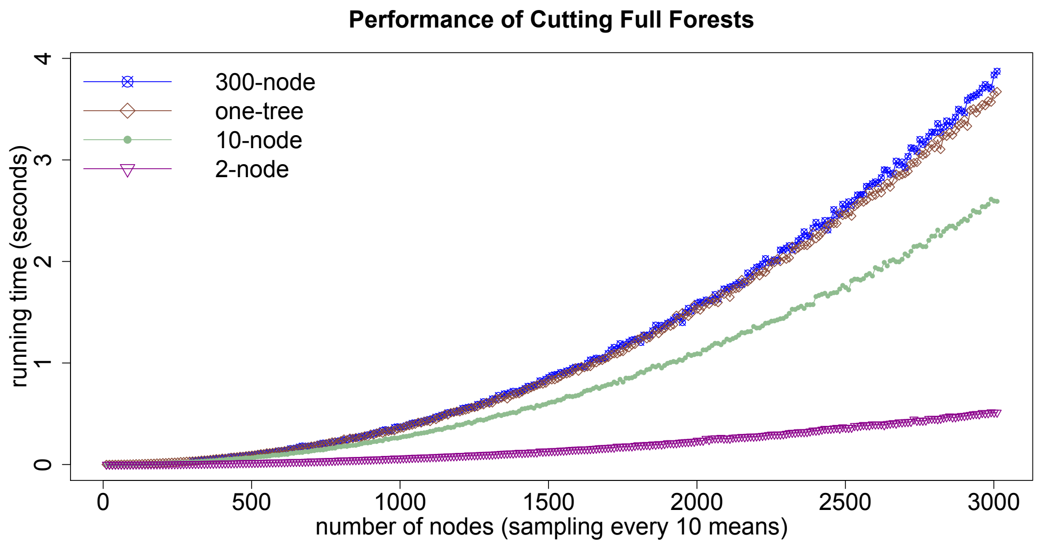

Figure 5.25: Performance of `cut` operation over a *one-tree*, *2-node*, *10-node* and *300-node* FULL forests, sampling every 10 means.

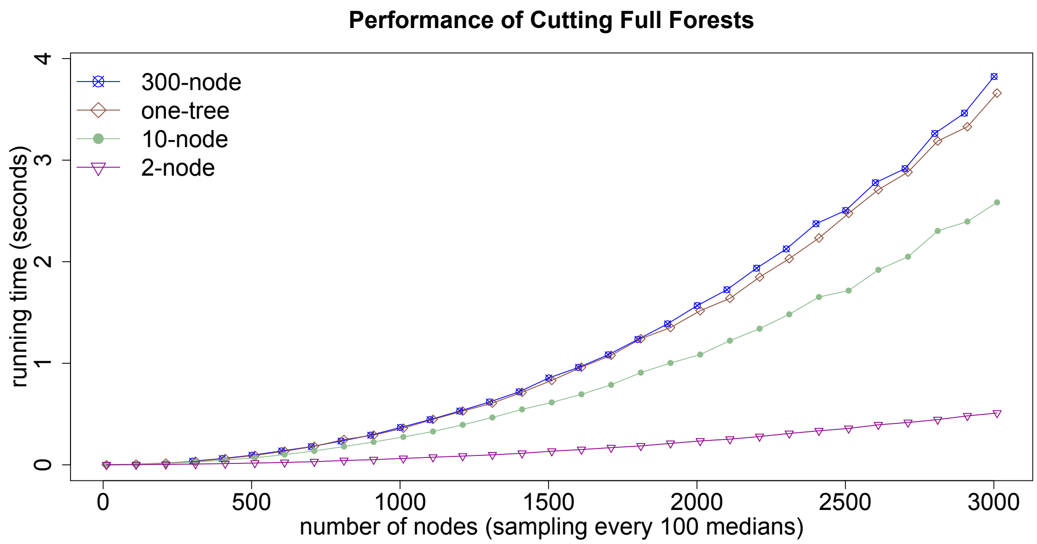

Figure 5.26: Performance of `cut` operation over a *one-tree*, *2-node*, *10-node* and *300-node* FULL forests, sampling every 100 medians.

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 $\Omega(\log^2 n) \leq ($ `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 `cut`s 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.

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

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

`link`ing and `cut`ting 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.

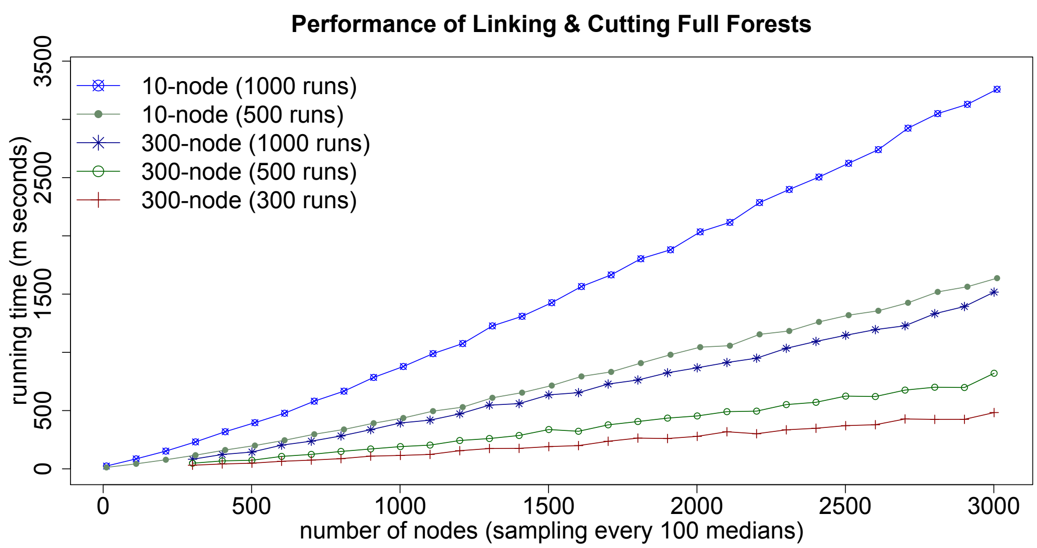

Figure 5.27: Performance of `link` and `cut` operations over *10-node* and *300-node* FULL forests, sampling every 100 medians.

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 FT 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.

num. nodes	*10-n* 500 millisec.	*10-n* 1K millisec.	*300-n* 300 millisec.	*300-n* 500 millisec.	*300-n* 1K millisec.
11	12.308	24.079	NA	NA	NA
111	42.962	85.524	NA	NA	NA
211	77.989	152.207	NA	NA	NA
311	117.056	231.254	30.927	48.005	82.320
411	161.479	318.589	41.609	68.053	122.978
511	199.922	396.193	48.347	73.593	144.137
...	...	...	...	...	...
2,811	1,518.818	3,049.764	424.455	700.562	1,331.922
2,911	1,563.308	3,129.308	424.914	698.088	1,393.413
3,011	1,636.775	3,258.576	483.003	820.563	1,517.262

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 or runnigs, 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.

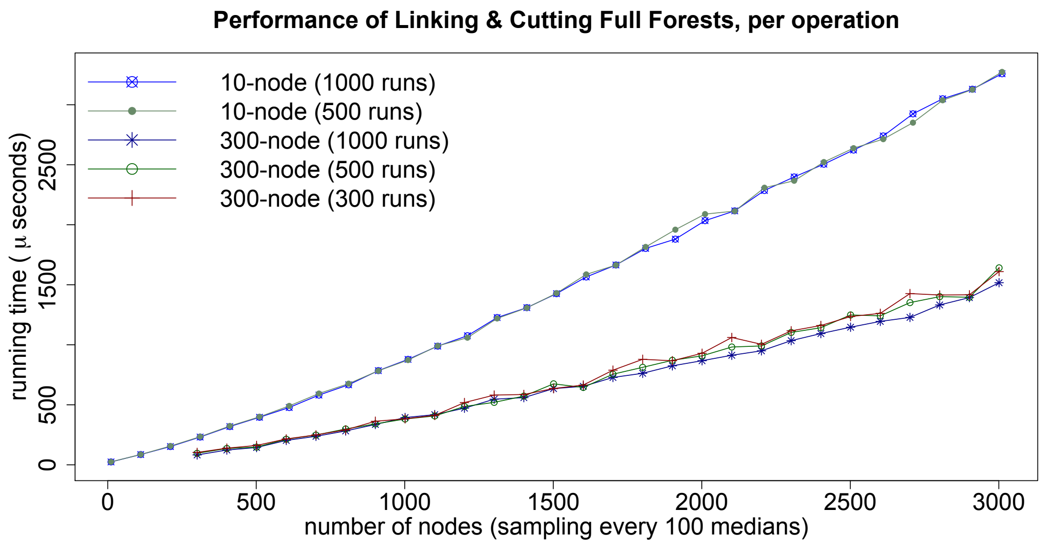

Figure 5.28: Performance of *individual* `link` and `cut` operations over *10-node* and *300-node* forests.

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

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

Table 5.12: An excerpt of tabular values for Figure 5.28.

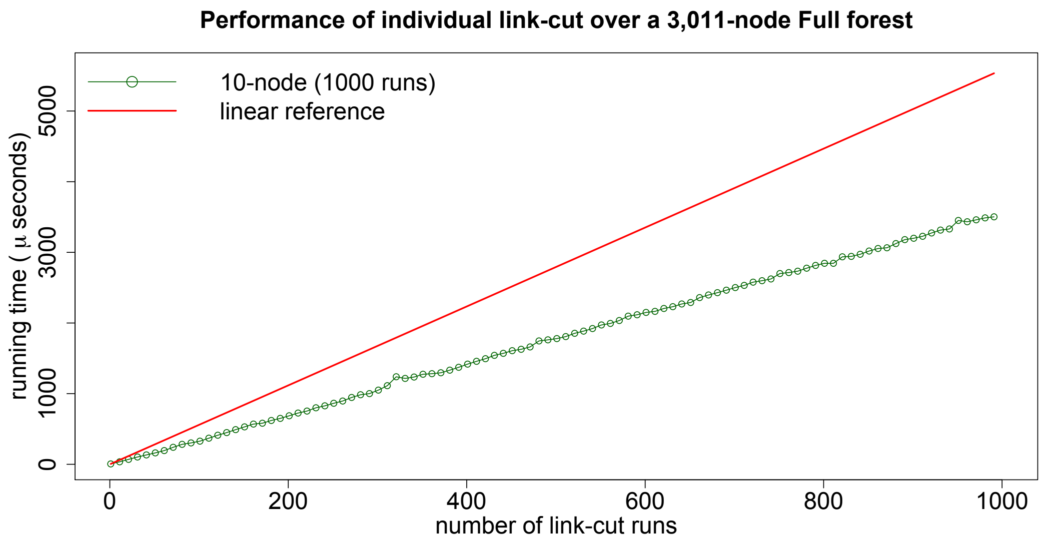

Figure 5.29: Performance of *individual* `link` and `cut` operations over a *10-node* forest having 3011 leaves.

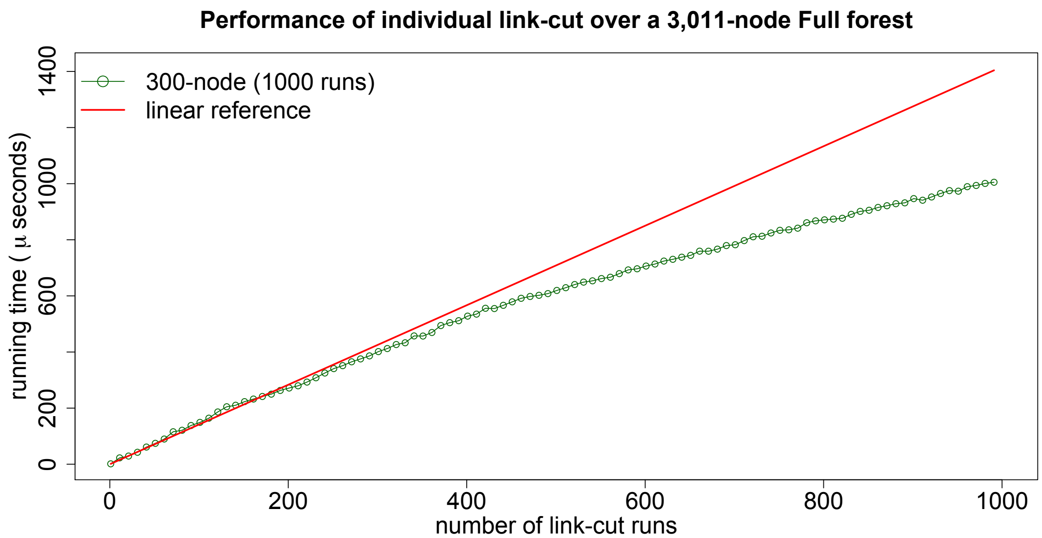

Figure 5.30: Performance of *individual* `link` and `cut` operations over a *300-node* forest having 3011 leaves.

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.

n^{th} input	runtime of the `link-cut` mean in μ seconds	runtime of 1^{st} mean $\times$ n^{th} input in μ seconds (linear reference)
1	5.584	5.584
11	35.406	61.424
21	69.757	117.264
971	3,460.134	5,422.064
981	3,487.708	5,477.904
991	3,502.422	5,533.744

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.

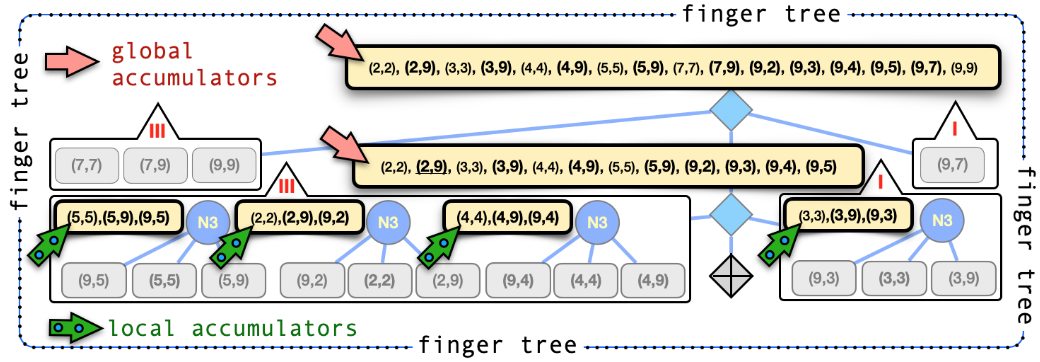

Figure 5.31: Accumulators identified in a FT data structure.

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.

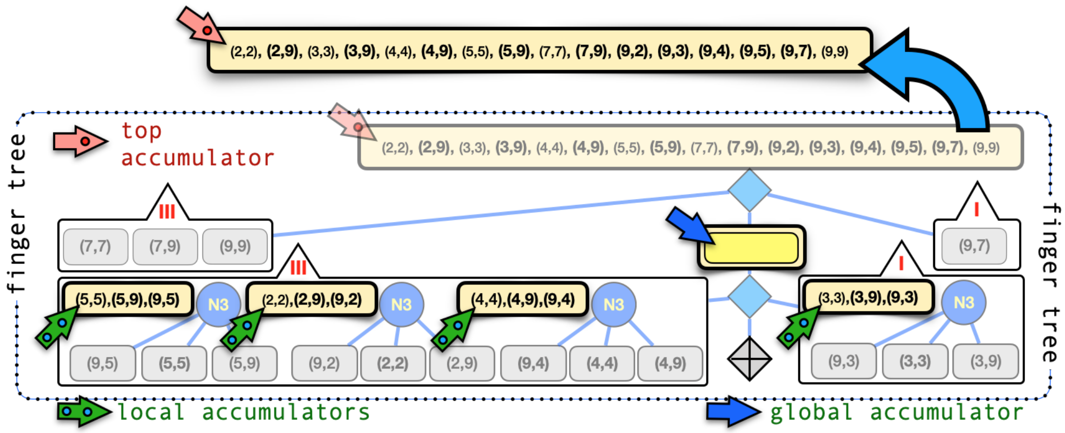

Figure 5.32: FT data structure for Top finger trees, top accumulator.

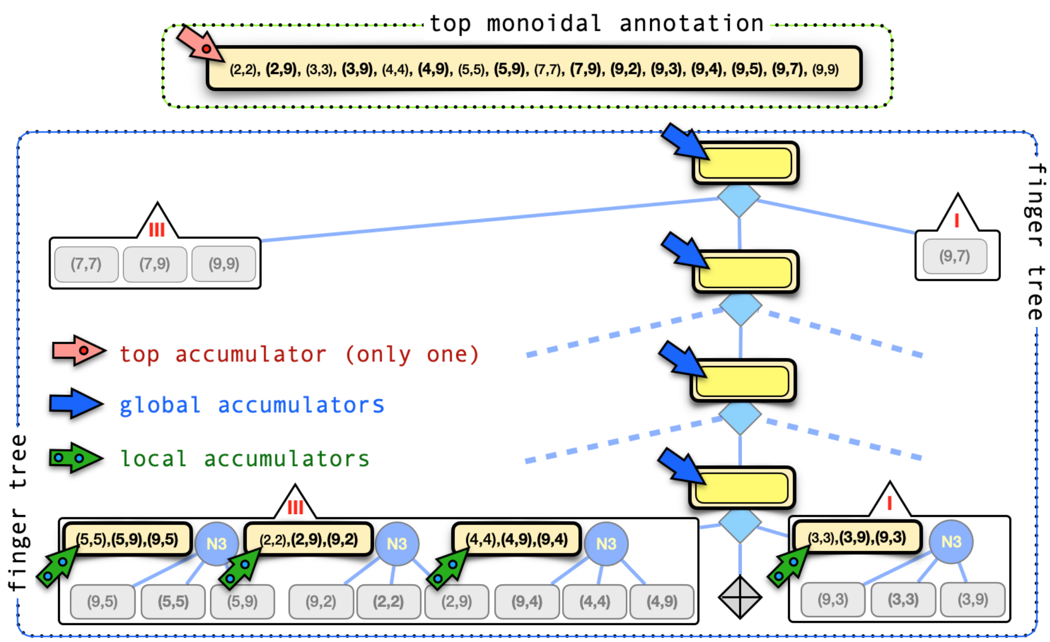

Figure 5.33: FT data structure for Top finger trees, general view.

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 evens 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 $\mathcal{O}(1)$ 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.

```
deep prefix middle suffix    -- FULL deep version
```

```
   = Deep (measure prefix <> measure middle prefix <> measure suffix)
       prefix middle suffix

deep prefix middle suffix     -- Top deep version
   = Deep mempty prefix middle suffix
```

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.

```
a ◁ Deep v (Four b c d e) m sf   =                    -- Full version
    Deep (measure a <> v) (Two a b) (node3 c d e ◁ m) sf
a ◁ Deep v pr m sf   =                                -- Full version
    Deep (measure a <> v) (consDigit a pr) m sf

a ◁ Deep v (Four b c d e) m sf   =                    -- Top version
    Deep mempty (Two a b) (node3 c d e ◁ m) sf
a ◁ Deep v pr m sf   =                                -- Top version
    Deep mempty (consDigit a pr) m sf
```

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 `<>`, which is $\Theta(m \log(\frac{n}{m}))$, 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 $\Theta(m \log n \log \frac{n}{m})$. Now, following the analysis done in Table 4.4 we get the bounds $\Omega(\log^2 n) \leq ($ `collectSetsMid` $) \leq \mathcal{O}(n \log n)$. 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 out 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`.

```
linkTree u tu v (TreeTop msetv tv) =
   let from = rerootTree tu u
     (Position (ls,left) _ (rs,right)) = searchMSet (v,v) msetv tv
     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 $\Omega(\log^2 n) \leq ($ `linkTree` $) \leq \mathcal{O}(n \log n)$, where n is the total number of nodes for the trees in the `linkTree` operation.

```
link x y forest@(ForestTop mset ft) =
```

```haskell
      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 (_,lf') _ (_,rf') = searchMSet (x,x) mset ft
    Position (_,lf)  _ (_,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 $\mathcal{O}(n \log n)$ 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.

```haskell
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))
      _                         →
        case (searchEdge pair' rs right) of
          Position (lsR,leftR) _ (rsR,rightR) →
            (TreeTop lsR leftR, TreeTop (mappend ls rsR) (left  ⋈ rightR))
          _               → 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 (leftTree ◁ ((lFor ⋈ rFor) ▷ rightTree))
```

Similarly, the lower and upper bounds for each operation are $\Omega(\log^2 n)$ and $\mathcal{O}(n \log n)$ 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.

Operation	name in Full	best case	worst case	context
`root`	`rootMSet`	$\Omega(\log n)$	$\mathcal{O}(n)$	trees
`connected`	`connectedMSet`	$\Omega(\log^2 n)$	$\mathcal{O}(n \log n)$	forest
`cut`	`cutMSet`	$\Omega(\log^2 n)$	$\mathcal{O}(n \log n)$	forest
`link`	`linkMSet`	$\Omega(\log^2 n)$	$\mathcal{O}(n \log n)$	forest

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.

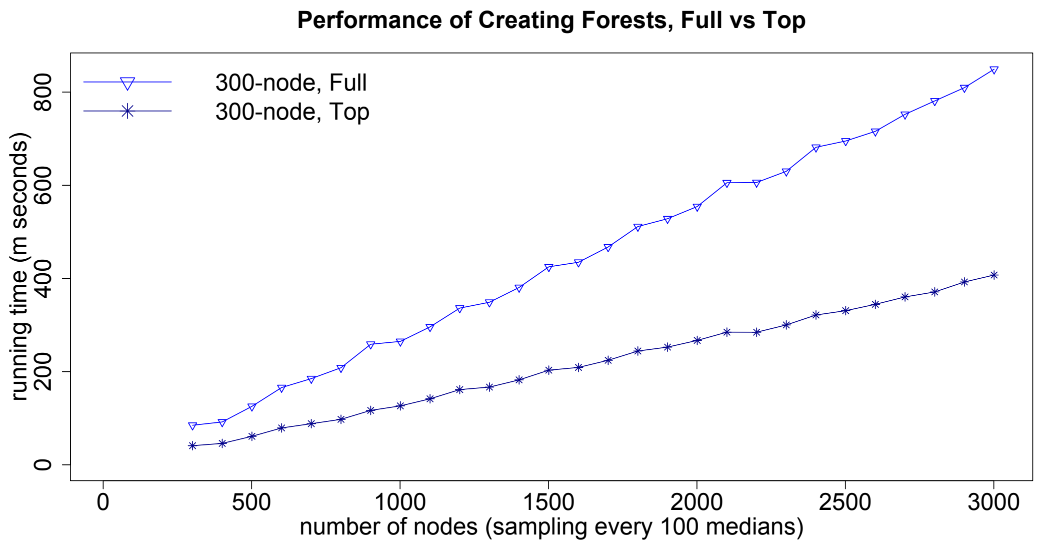

Figure 5.34: Performance of constructing *300-node* forests, Full vs Top.

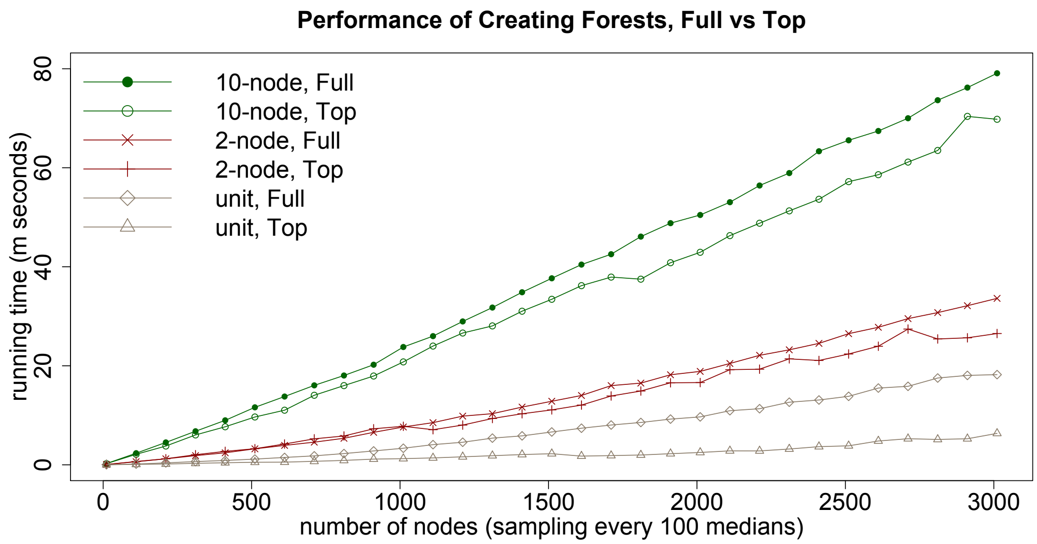

Figure 5.35: Performance of constructing *unit*, *2-node*, and *10-node* forests, Full vs Top.

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.

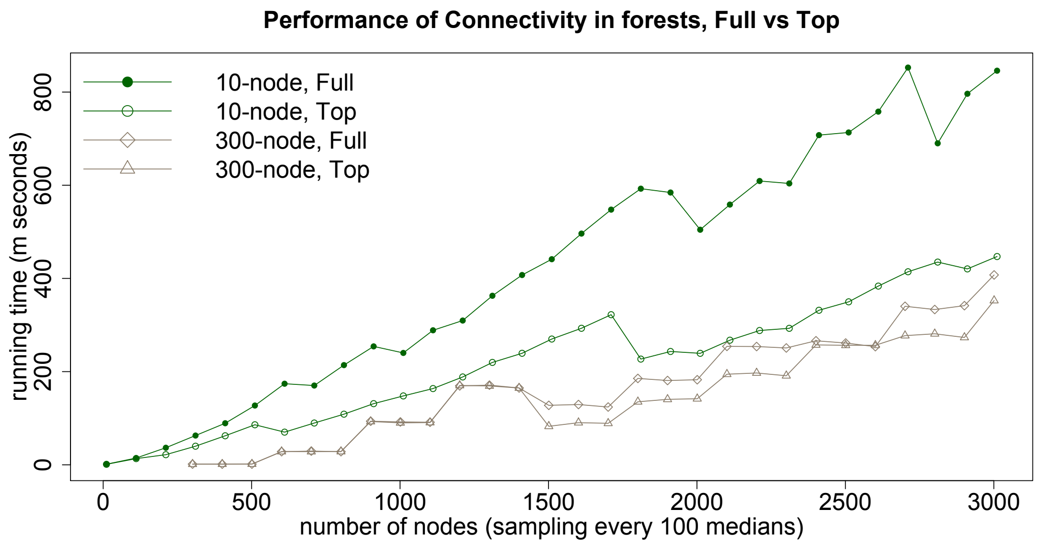

Figure 5.36: Performance of connectivity in *10-node* and *300-node* forests, Full vs Top.

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.

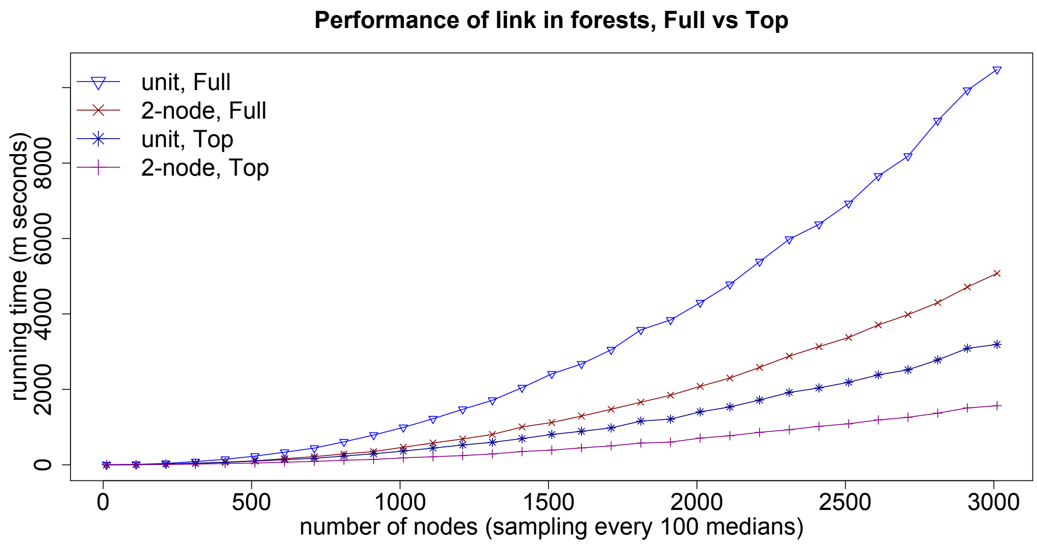

Figure 5.37: Performance of `link` operation in *unit* and *2-node* forests, Full vs Top.

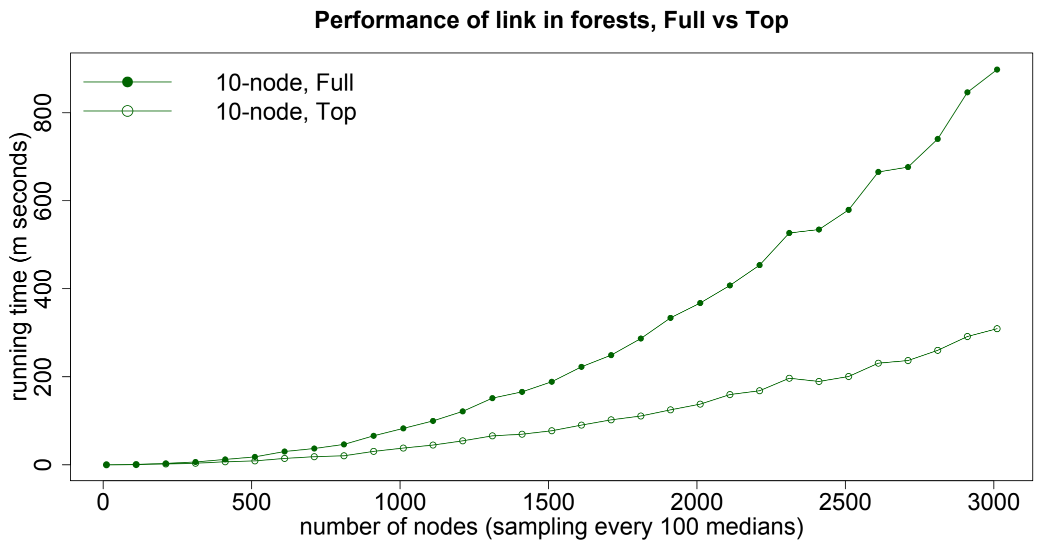

Figure 5.38: Performance of `link` operation in *10-node* forests, Full vs Top.

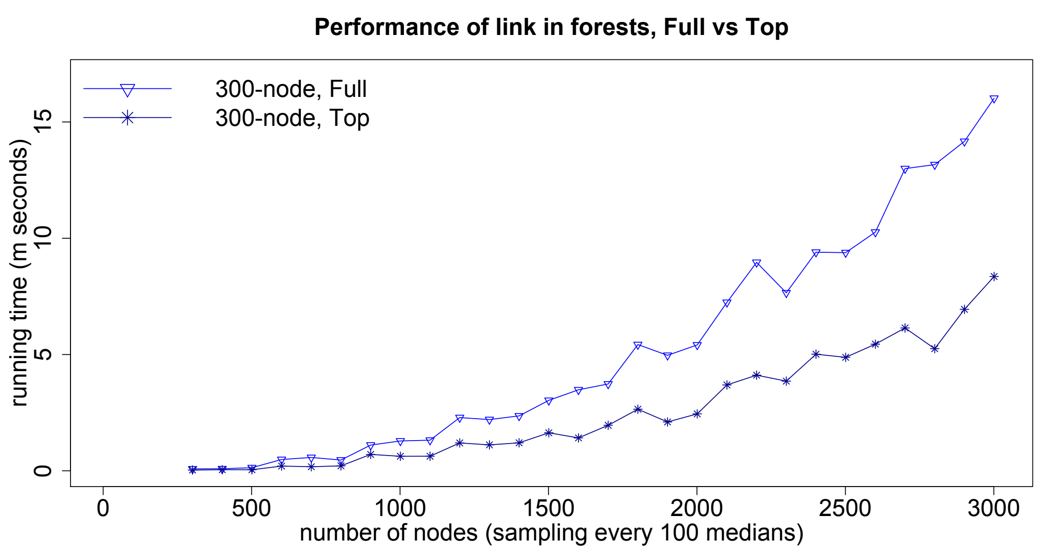

Figure 5.39: Performance of `link` operation in *300-node* forests, Full vs Top.

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.

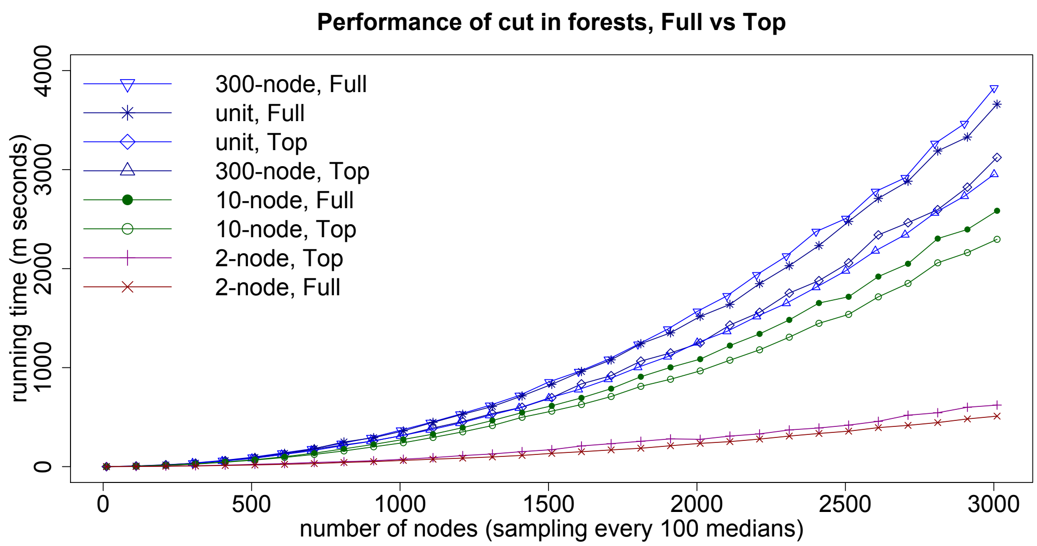

Figure 5.40: Performance of `cut` operation for *one-tree*, *300-node*, *10-node* and *2-node* forests, Full vs Top.

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.

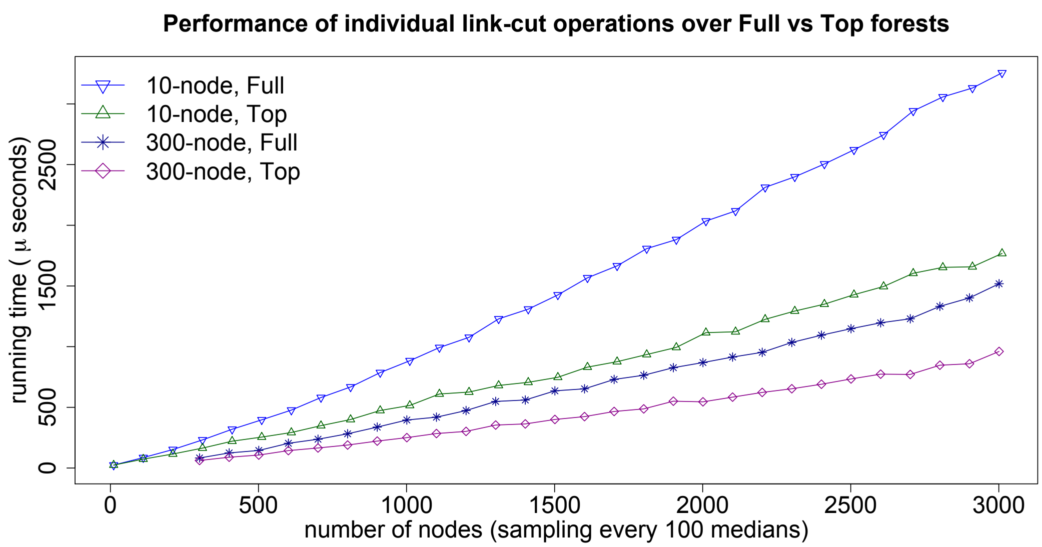

Figure 5.41: Performance of individual `link-cut` operations over *10-node* and *300-node* forests, FULL vs TOP.

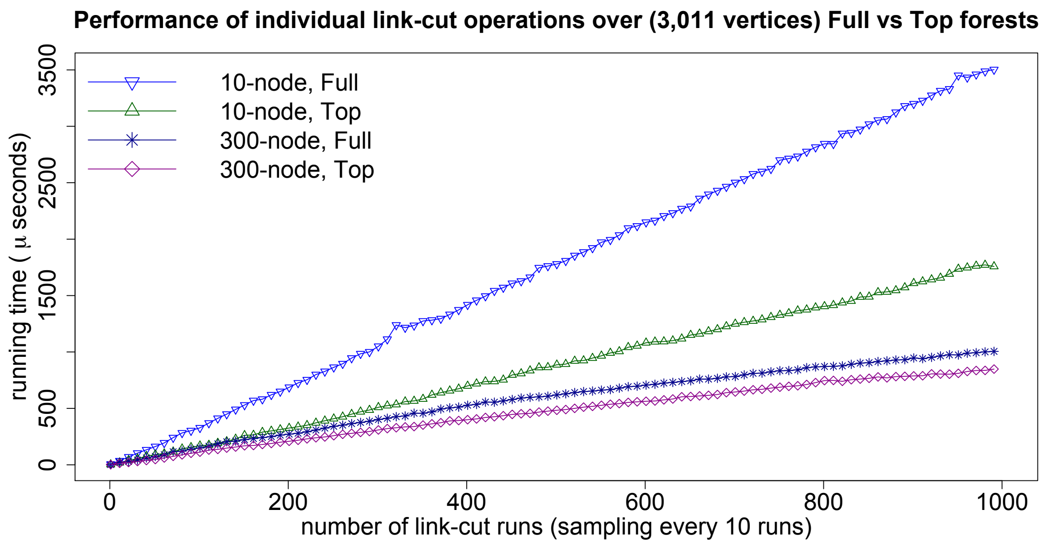

Figure 5.42: Performance of individual `link-cut` operations over (3,011 vertices) *10-node* and *300-node* forests, FULL vs TOP.

Experimental results of `link-cut` operations over *10-node* and *300-node* forests, FULL vs TOP

For all of the cases in figs. 5.41 and 5.42, performance of TOP shows a lower cost in time than FULL approach, being 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 `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 FUNETT, 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 FUNETT 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, $\Omega(\log^2 n) \leq ($ `dynOp` $) \leq \mathcal{O}(n \log n)$, where n is the number of vertices in the forest and that of `dynOp` $\in \{$ `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 $\Theta(\log n) \times \mathcal{O}(\texttt{<>})$ where `<>` 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 $\mathcal{O}(m \log(\frac{n}{m}))$ 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 $\Omega(\log n) \leq \Theta(\triangleleft) \leq \mathcal{O}(n \log n)$, where n is the number of vertices in the forest.

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

  - $\mathcal{O}(n \log n)$ 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 Chapter5.
  - 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 Chapter5.
  - 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 Chapter5.

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 advaced* 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

[1] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of computer and system sciences*, vol. 26, no. 3, pp. 362–391, 1983.

[2] C. Okasaki, "Purely functional data structures," Ph.D. dissertation, Carnegie Mellon University, 1996.

[3] ——, "`Data.Edison.Coll`," https://hackage.haskell.org/package/EdisonCore, 2018, [Online; accessed 10-Apr-2019; version 1.3.2.1].

[4] S. Marlow *et al.*, "Haskell 2010 language report," *Available online http://www.haskell.org/(May 2011)*, 2010.

[5] M. R. Henzinger and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," in *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, 1997, pp. 519–527.

[6] ——, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *Journal of the ACM (JACM)*, vol. 46, no. 4, pp. 502–516, 1999.

[7] R. E. Tarjan, "Dynamic trees as search trees via euler tours, applied to the network simplex algorithm," *Mathematical Programming*, vol. 78, no. 2, pp. 169–177, 1997.

[8] R. Hinze and R. Paterson, "Finger trees: a simple general-purpose data structure," *Journal of Functional Programming*, vol. 16, no. 02, pp. 197–217, 2006.

[9] C. Demetrescu, I. Finocchi, and G. Italiano, "Dynamic trees," *Handbook of Data Structures and Applications, Chapman & Hall/CRC Computer & Information Science Series*, 2004.

[10] L. Katherine, "Complexity of the union-split-find problems," 2007, massachusetts Institute of Technology, Master dissertation.

[11] R. E. Tarjan, *Data Structures and Network Algorithms.* Siam, 1983.

[12] D. Nanongkai, T. Saranurak, and S. Yingchareonthawornchai, "Breaking quadratic time for small vertex connectivity and an approximation scheme," *arXiv preprint arXiv:1904.04453*, 2019.

[13] G. Bernardini, P. Bonizzoni, G. Della Vedova, and M. Patterson, "A rearrangement distance for fully-labelled trees," *arXiv preprint arXiv:1904.01321*, 2019.

[14] T. Ophelders, W. Sonke, B. Speckmann, and K. Verbeek, "Kinetic volume-based persistence for 1d terrains," *35th European Workshop on Computational Geometry*, 2019.

[15] M. Erwig, "Inductive graphs and functional graph algorithms," *Journal of Functional Programming*, vol. 11, no. 05, pp. 467–492, 2001.

[16] E. Kmett, "`LinkCut Trees`," http://hackage.haskell.org/package/structs-0.1.1, 2017, [Online; accessed 10-Apr-2019; version 0.1.1].

[17] R. Bird and J. Gibbons, *Algorithm Design with Haskell.* Cambridge University Press, 2020.

[18] R. Bird, *Thinking functionally with Haskell.* Cambridge University Press, 2014.

[19] C. Okasaki, *Purely Functional Data Structures.* Cambridge University Press, 1999.

[20] M. Stannett, *COM2001: Advanced Programming Topics. A Practical Guide using Haskell and Java.* The University of Sheffield, 2013.

[21] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, no. 2, pp. 98–107, 1989.

[22] Z. Hu, J. Hughes, and M. Wang, "How functional programming mattered," *National Science Review*, vol. 2, no. 3, pp. 349–370, 2015.

[23] H. Community, "`Haskell in Industry`," https://wiki.haskell.org/Haskell_in_industry, At the time of writing this thesis, [Online; accessed 10-Apr-2019].

[24] J. C. Saenz-Carrasco and M. Stannett, "Fatfast: traversing fat branches fast in haskell," in *Symposium on Implementation and Application of Functional Languages.* ACM, 2019, presented.

[25] J. C. Saenz-Carrasco, "Funseqset: Towards a purely functional data structure for the linearisation case of dynamic trees problem," in *Workshop on Functional and (Constraint) Logic Programming*, vol. abs/1908.11105, 2019, presented. [Online]. Available: http://arxiv.org/abs/1908.11105

[26] M. H. Overmars, *The design of dynamic data structures.* Springer Science & Business Media, 1983, vol. 156.

[27] R. F. Werneck, "Design and analysis of data structures for dynamic trees," Ph.D. dissertation, Princeton University, 2006.

[28] R. E. Tarjan and R. F. Werneck, "Dynamic trees in practice," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 5, 2009.

[29] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, 1985.

[30] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *19th Annual Symposium on Foundations of Computer Science (SFCS 1978).* IEEE, 1978, pp. 8–21.

[31] S. Kahrs, "Red-black trees with types," *Journal of functional programming*, vol. 11, no. 4, pp. 425–432, 2001.

[32] R. Hinze, "Constructing red-black trees," in *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL*, vol. 99, 1999, pp. 89–99.

[33] G. Aumala *et al.*, "`Data.RedBlackTrees`," http://hackage.haskell.org/packages/search?terms=red+black+tree, 2017, [Online; accessed 10-Oct-2019].

[34] G. N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 781–798, 1985.

[35] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo, "Dynamizing static algorithms, with applications to dynamic trees and history independence," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2004, pp. 531–540.

[36] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup, "Minimizing diameters of dynamic trees," in *International Colloquium on Automata, Languages, and Programming.* Springer, 1997, pp. 270–280.

[37] S. Marlow, "Parallel and Concurrent Programming in Haskell," in *Central European Functional Programming School.* Springer, 2011, pp. 339–401.

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

[39] T. Harris, S. Marlow, and S. P. Jones, "Haskell on a shared-memory multiprocessor," in *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell.* ACM, 2005, pp. 49–61.

[40] A. Morihata and K. Matsuzaki, "Balanced trees inhabiting functional parallel programming," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 117–128, 2011.

[41] The University of Glasgow, "`Data.Tree`," http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Tree.html, 2002, [Online; accessed 10-Oct-2019; version 0.6.2.1].

[42] S. Adams, "Functional pearls efficient sets—a balancing act," *Journal of functional programming*, vol. 3, no. 4, pp. 553–561, 1993.

[43] M. Straka, "Adams' trees revisited," in *Trends in Functional Programming, 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers*, 2011, pp. 130–145. [Online]. Available: https://doi.org/10.1007/978-3-642-32037-8_9

[44] J. C. Derryberry, "Adaptive binary search trees," Ph.D. dissertation, Carnegie Mellon University, 2009.

[45] D. Leijen, "`Data.Set`," http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Set.html, 2002, [Online; accessed 10-Oct-2019; version 0.6.2.1].

[46] A. Gill, "`Data.Monoid`," http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Monoid.html, 2001, [Online; accessed 10-Oct-2019; version 4.12.0.0].

[47] B. A. Yorgey, "Monoids: theme and variations (functional pearl)," in *ACM SIGPLAN Notices*, vol. 47, no. 12. ACM, 2012, pp. 105–116.

[48] R. Paterson, "`Finger Tree`," http://hackage.haskell.org/package/fingertree-0.1.4.2, 2018, [Online; accessed 10-Oct-2019; version 0.1.4.2].

[49] ——, "`Data.Sequence`," http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Sequence.html, 2014, [Online; accessed 10-Oct-2019; version 0.6.2.1].

[50] J. C. Sáenz-Carrasco, "`Haskell and R source code`," http://github.com/jcsaenzcarrasco/thesis, 2019, [Online; accessed 10-Oct-2019].

[51] R. project, "`The R Project for Statistical Computing`," https://www.r-project.org/, 2019, [Online; accessed 10-Oct-2019].

[52] A. Yakeley, "`A time library`," http://hackage.haskell.org/package/time-1.9.3, 2019, [Online; accessed 10-Oct-2019; version 1.9.3].

Index