Dynamic Algorithms for Partially-Sorted Lists

Kostya Kamalah Ross

School of Computer and Mathematical Sciences

November 3, 2015

A thesis submitted to Auckland University of Technology in fulfilment of the requirements for the degree of Master of Philosophy.

Abstract

The dynamic partial sorting problem asks for an algorithm that maintains lists of numbers under the link, cut and change value operations, and queries the sorted sequence of the k least numbers in one of the lists. We examine naive solutions to the problem and demonstrate why they are not adequate. Then, we solve the problem in $O(k \cdot \log(n))$ time for queries and $O(\log(n))$ time for updates using the tournament tree data structure, where n is the number of elements in the lists. We then introduce a layered tournament tree data structure and solve the same problem in $O(\log^*_\varphi(n) \cdot k \cdot \log(k))$ time for queries and $O(\log^2(n))$ for updates, where φ is the golden ratio and $\log^*_\varphi(n)$ is the iterated logarithmic function with base φ. We then perform experiments to test the performance of both structures, and discover that the tournament tree has better practical performance than the layered tournament tree. We discuss the probable causes of this. Lastly, we suggest further work that can be undertaken in this area.

Contents

List of Figures

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Chapter 1

Introduction

Sorting is one of the oldest (and most widely-used) problems in computer science; sorted data is easier to manipulate, and allows a wider variety of efficient operations (such as searches) than unsorted data. Frequently, it is important to ensure that data remains sorted even under modification, insertion and removal of data, while not requiring expensive operations to re-sort it every time an update is made. This is the basis of various algorithms and data structures designed to maintain dynamic data in a sorted form.

A common use case for sorted data is the extraction of order statistics – this can range from a database query in SQL using the SORT ASCENDING and LIMIT commands, to Reddit's trending subreddits feed. In both of these, and many other cases, the order statistics are desired for relatively few items as compared to the total number of items in the collection (consider a 'Top 10' list, that could be drawn from thousands, or possibly millions of items). Additionally, both of these cases require operation on real-time data, which could change very rapidly. Both of these cases make the more traditional approach of sorting and then maintaining that data to be impractical.

In such cases, it is desirable to maintain collections information in a way that is amenable to performing the fastest queries possible, ideally with minimal influence from the number of items in the entire collection – only the size of the query. These collections should also be dynamic, allowing modification of existing information, insertion of new information and deletion of other information while maintaining the speed of queries. We refer to a structure that can perform this as a *partially-sorted list*.

We seek to examine the problem of implementing a partially-sorted list, analyze the performance of such implementations, and then test them in practice to see which gives better performance. We initially discuss some naive solutions and their deficiencies, and then present two different data structures that can implement a partially-sorted list – the tournament tree and the layered tournament tree. We analyze their perfor-

mance, and then describe experiments that were conducted to test their performance in practical terms. We determine that, while the layered tournament tree has better independence from the size of the collection being queried, in practice, the tournament tree has better performance.

1.1 Problem setup

This thesis seeks to present a solution to the following problem: maintain a dynamic data structure D representing a partially-sorted list of n pairwise-disjoint numbers, permitting the following operations:

- $\mathsf{psort}(D, k)$: Perform a partial sort operation on the numbers stored in D. This must return k numbers, or every number stored in D, whichever is smaller. The numbers returned must be in monotonically increasing order.
- $\mathsf{changeval}(D, i, n)$: Change the value at index i in D to n.
- $\mathsf{link}(D, D')$: Combine D and another partially-sorted list D' to form a single structure. The numbers stored in D' must sequentially follow those stored in D.
- $\mathsf{cut}(D, i)$: Separate D into two partially-sorted lists D_1, D_2 such that D_1 stores all numbers up to and including the number at index i, and D_2 stores the remaining numbers, preserving the same order as D.

We assume that in changeval and cut, we have a reference to the ith element available, allowing access to that element in $O(1)$.

Additionally, the psort operation should have an asymptotic time complexity that depends as little on the number of numbers stored in D as possible. We also aim to allow the update operations ($\mathsf{changeval}, \mathsf{link}, \mathsf{cut}$) to be performed in as close to $O(\log(n))$ as possible.

We refer to this as the *dynamic partial sorting problem.*

1.2 Contribution

The goal of this thesis is to design a solution to the dynamic partial sorting problem with good time complexities for all of the operations required. We describe two solutions based on dynamic trees. We also prove their correctness and analyze their time complexity.

We demonstrate that our first solution, based on a 'tournament tree' (or TT) data structure, is able to perform $\mathsf{psort}(D, k)$ in $O(k \cdot \log(m))$, and $\mathsf{changeval}(D, i, n)$ and

$\mathsf{cut}(D, i)$ in $O(\log(m))$, where m is the number of numbers stored. We also show that we can perform $\mathsf{link}(D, D')$ in $O(|\log(x) - \log(y)|)$, where x is the number of numbers stored in the larger of D, D', and y is the number of numbers stored in the smaller of D, D'.

We also describe an extension of the tournament tree, which we call the 'layered tournament tree' or LTT. We demonstrate that this solution is able to perform $\mathsf{psort}(D, k)$ in $O(\log^*_\varphi(m) \cdot k \cdot \log(k))$, and $\mathsf{changeval}(D, i, n)$ and $\mathsf{cut}(D, i)$ in $O(\log^2(m))$, where m is the number of numbers stored. We also show that we can perform $\mathsf{link}(D, D')$ in $O(\log^2(|x - y|)$, where x is the number of numbers stored in the larger of D, D' and y is the number of numbers stored in the smaller of D, D'.

Lastly, we demonstrate the performance of both structures experimentally by testing each of the psort, changeval, link and cut operations.

1.3 Related work

To the author's knowledge, there has not been work formally addressing the dynamic partial sorting problem. We describe related work dealing with the maintenance of sorted lists dynamically, theoretical work relating to partial sorting, and solutions to similar problems.

Dynamically maintaining a sorted list of numbers is a well-explored topic. Existing solutions rely on various self-balancing binary search trees, as described in Andersson, Fagerberg and Larsen [3]. None of these data structures are suitable for the dynamic partial sorting problem described in Section 1.1, as we require elements in the lists to preserve their orders while extracting order statistics from the lists. We present some examples of why this is the case in Chapter 2; in particular, we focus on the red-black tree and the splay tree as a source of ideas for our solution to the dynamic partial sorting problem.

The asymptotic time complexity of partial sorting static data has been studied extensively. The earliest work on this topic was the quickselect algorithm implementation by Hoare [12], which described a method of retrieving the kth smallest value from a static collection of numbers in a faster time complexity than sorting the collection. Floyd and Rivest [8] investigated possible lower bounds for this operation (which they term the selection problem). Additional work in this area was done by Huang and Tsai [14] and Kuba [17]. The problem was also generalized to sorting intervals, and work has been done to analyze the time complexity of such an operation [15].

Several data structures have been proposed to solve similar problems to the one described in Section 1.1. One such structure was proposed by Navarro and Paredes [18]. However, this structure is designed to be optimal in terms of space, rather than time,

and is also both amortized and online. Duch et al [7] presented another structure, but this attempts to solve a somewhat different problem (although the solution can still be used to perform the psort operation). The structure is also not dynamic, and depends heavily on the length of the input data.

A practical application of a similar problem to the one described in Section 1.1 was employed to solve problems in common-channel communication over single-hop wireless sensor networks by Bordim et al [5]. However, this concerns itself primarily with queries, rather than update operations, and is designed to be distributed, which our problem does not mention.

1.4 Organization

We begin by describing naive solutions to the dynamic partial sorting problem, as well as their limitations, in Chapter 2. In this chapter, we will also examine dynamic trees, and describe why they are particularly suited to solving the dynamic partial sorting problem. Lastly, we examine two examples of dynamic trees, which form the basis of the structures defined in this thesis.

In Chapter 3, we introduce the tournament tree data structure. We describe the structure, and then present algorithms that use that structure to solve the dynamic partial sorting problem. We prove the correctness of these algorithms, and analyze their time complexity. We also examine the deficiencies of the tournament tree as a solution, which we attempt to improve on in Chapter 4.

We attempt to improve on the tournament tree in Chapter 4 by defining an extension called the layered tournament tree. We describe this structure, and then define algorithms that allow us to solve the dynamic partial sorting problem with layered tournament trees. In the process, we also analyze the time complexity of these operations, and prove their correctness. We demonstrate that the layered tournament tree has better performance with respect to the psort operation, while still retaining acceptable performance on changeval, link and cut operations.

In Chapter 5, we attempt to experimentally demonstrate the performance of the tournament tree and layered tournament tree, both in isolation and in contrast with each other. We describe our implementation and the performance characteristics of the machine being used to perform the tests, and define how the experiments will be conducted and with what data. We then plot the results of the tests, and discuss the outcome.

Lastly, in Chapter 6, we summarize the thesis, discussing the outcomes of the research. We also consider future work that can be undertaken in relation to the dynamic partial sorting problem.

1.5 Preliminaries

We make some definitions that are used throughout this thesis. We consider $\mathbb{N} = \{0, 1, 2, \ldots\}$. We define a *list* L as a tuple $(a_1, a_2, \ldots, a_n)$ of pairwise-distinct integers.

1.5.1 Trees

We define a *tree* $T = (V, E)$ as a directed, labelled arborescence, as per Gross, Yellen and Zhang [9]. The labels for each $v \in V$ are integers.

The *size* of a tree T is $\mathsf{size}(T) = |V|$. We define the *root* of T as the node u with no parent. We define the *leaves* of T as all nodes which have no children. As this thesis will only consider *binary* trees, we will use 'tree' and 'binary tree' interchangeably to refer to binary trees.

We use $T(v)$ to denote the subtree rooted at the node v. A *path* is a set of nodes $\{u_0, u_1, \ldots, u_m\}$ such that $m \in \mathbb{N}$, u_o is a leaf and u_{i+1} is the parent of u_i for $0 \leq i < m$. We call m the *length* of the path. The *height* $h(T)$ of a tree T is the maximum length of every path in T.

Let u be a node in a tree T. The *layer* of u is the length of the path from u to the root of T. We define the *first* layer of T as layer number 0 (thus, the root), and the *last* layer as the layer whose number is the highest for T.

A tree $T = (V, E)$ is *balanced* if for every node $v \in V$, the height of its left subtree and its right subtree differ by no more than 1. We call T *full* if every node has exactly two children.

We define the concept of *tree rotation* in the usual way. See Figure 1.1 for an example of tree rotations.

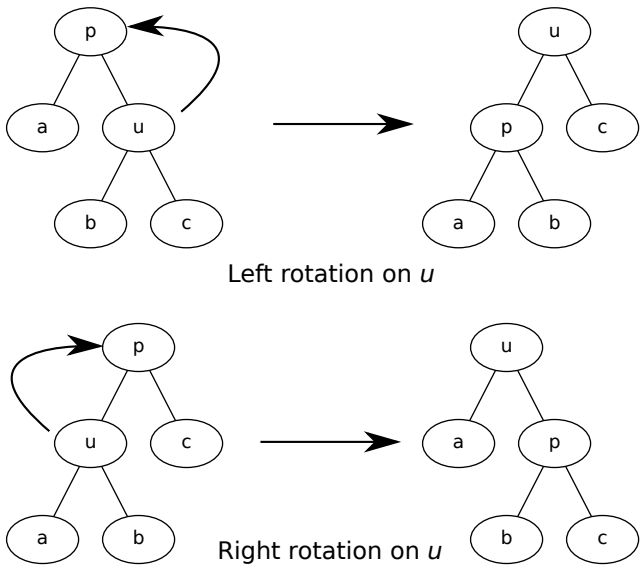

Figure 1.1: Tree rotation examples.

1.5.2 Asymptotic and Amortized Complexity

We give the following definition of a concept in asymptotic complexity, based on Cormen et al [6]:

Definition 1. Let f, g be functions. If there exist positive c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$, then we say that $f(n)$ is $O(g(n))$.

In this thesis, we will occasionally describe some operation `op` as being "a $O(f(n))$" operation, or "having time complexity of $O(f(n))$", or similar, for some function f. This is a shorthand for "the function representing the number of primitive operations to perform `op` is $O(f(n))$".

We use the following definition of amortized time complexity, based on Cormen et al [6]:

Definition 2. Let `op` be an operation. If there exists some function $T(n)$ such that a sequence of n calls to `op` has time complexity $T(n)$, we say that a single call to `op` has *amortized time complexity* $\frac{T(n)}{n}$.

1.5.3 Data structures

For the purposes of describing the different implementations of partially-sorted lists, we use several auxilliary data structures. We describe these here, along with any assumptions we make with regard to them.

Binary tree

Intuitively, our definition of the binary tree data structure is a representation of a directed, labelled arborescence [9]. Therefore, we will use the language developed in Subsection 1.5.1 with the binary tree data structure. More specifically, the concepts of roots, leaves, layers, tree balance, tree fullness and tree rotation carry forward to the binary tree data structure. We will also use the $T = (V, E)$ notation to describe a tree data structure, with V referring to the set of nodes in the tree data structure.

Unless specified otherwise, any reference to a 'tree data structure' in this thesis assumes a binary tree data structure.

A tree data structure is comprised of *nodes*, each of which stores references to other nodes or its integer *value*. We assume that these references are accessible in $O(1)$ time if we have a reference to the node itself.

More formally, each node u in a binary tree is a 4-tuple

$$(\mathsf{p}(u), \mathsf{left}(u), \mathsf{right}(u), \mathsf{val}(u))$$

such that

- $\mathsf{p}(u)$ is a reference to the parent of u, or nil if u is the root
- $\mathsf{left}(u)$ and $\mathsf{right}(u)$ are references to to the left and right child of u respectively, or nil if not present
- $\mathsf{val}(u)$ is a reference to the integer value of u

A binary tree is viewed as a reference to its root; thus, we will refer to a tree and its root interchangeably. We use $\mathsf{rotate\_left}(u)$ and $\mathsf{rotate\_right}(u)$ to denote a left rotation on u and a right rotation on u respectively.

Priority queue

A priority queue is a data structure to support priority-based retrieval of elements. We describe its operations below; in all cases, Q is a reference to a priority queue instance, x is an arbitrary element and i is an integer priority value.

- $\mathsf{insert}(Q, x, i)$: Adds x to Q with priority i, in time $O(\log(n))$.
- $\mathsf{delete\_min}(Q)$: Removes and returns the element in Q with the lowest priority, in time $O(\log(n))$.
- $\mathsf{empty}(Q)$: Returns true if Q does not contain any elements, and false otherwise, in time $O(1)$.

This corresponds to a min-heap as defined in Sahni [20], which also gives several possible implementations of such a structure. We present an example of a min-heap – the *binary heap*. We define the binary heap data structure as follows:

Definition 3. A *binary heap* B is a binary tree with the following additional properties:

1. Each node u has a field $\mathsf{data}(u)$, storing arbitrary data.
2. Let u be an internal node in B. The value of any child of u must be less than the value of u.
3. Every node in each layer, except the second-to-last, must have two children. The layers must be filled from left to right.

See Figure 1.2 for an example of a binary heap. We observe that, by Definition 3, the height of any binary heap B is $O(\log(n))$, where $n = |B|$.

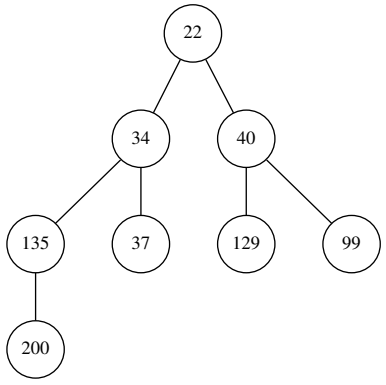

Figure 1.2: A binary heap. The nodes are labelled with their values.

A key operation for binary heaps is $\mathsf{percolate}(u)$, which is designed to correct any violations of property 2 as described in Definition 3. $\mathsf{percolate}(u)$ compares $\mathsf{val}(u)$ to $\mathsf{val}(\mathsf{p}(u))$. If $\mathsf{p}(u) \neq \mathsf{nil}$ and $\mathsf{val}(\mathsf{p}(u)) < \mathsf{val}(u)$, it exchanges the positions of the nodes and then calls itself recursively on u again; otherwise, it terminates. For an exact description, see Algorithm 1.

Algorithm 1 $\mathsf{percolate}(u)$

```
1: if u is not the root then
2:     if val(p(u)) < val(u) then
3:         Swap the positions of p(u) and u
4:         percolate(u)                    ▷ u will now be where p(u) used to be
```

We observe that by Definition 3, $\mathsf{percolate}(u)$ has time complexity $O(\log(n))$, where n is the size of the tree containing u.

We now briefly describe the implementation of the priority queue operations using the binary heap.To perform $\mathsf{insert}(B, x, i)$, we create a new node u such that $\mathsf{val}(u) = i$ and $\mathsf{data}(u) = x$. If B is empty, we set u to be its root; otherwise:

- If B contains a leftmost node v in its second-to-last layer which don't have two children, we insert u as a child of v;
- Otherwise, we insert u as the left child of the leftmost leaf of B.

We then call $\mathsf{percolate}(u)$ to correct any violations of the properties of the binary heap. To perform $\mathsf{delete\_min}(B)$, we set the value of its root to be $+\infty$, and then use a variant of the percolate operation to move the root node to a leaf position. We then simply remove the node and return it. To perform $\mathsf{empty}(B)$, it suffices to check if B has any nodes.

For the time complexity of each of the priority queue operations implemented using the binary heap, we give the following lemma. Its proof follows from Defintion 3 and Algorithm 1.

Lemma 1. *Let B be a binary heap whose size is n.* $\mathsf{insert}(B, x, i)$ *and* $\mathsf{delete\_min}(B)$ *are* $O(\log(n))$*, and* $\mathsf{empty}(Q)$ *is* $O(1)$*.*

Although Sahni [20] defines additional structures (such as the skew heap and the Fibonacci heap) which are able to achieve better performance, for the purpose of this thesis, the performance of the binary heap suffices to demonstrate the asymptotic time complexity of our data structures; thus, we will not discuss these alternative priority queue implementations.

Chapter 2

Dynamic Trees

In this chapter, we describe two naive solutions for the dynamic partial sorting problem, using dynamic arrays and linked lists. We demonstrate that these solutions have undesirable performance characteristics given the problem description given in Section 1.1. We then describe dynamic trees, and explain why they have desirable characteristics for the dynamic partial sorting problem. We describe two common examples of dynamic trees, which serve as the basis for the structures we present in Chapters 3 and 4.

2.1 Naive solutions

We present two naive solutions to the dynamic partial sorting problem, designed to illustrate the deficiencies of their performance relative the description of the problem. We will consider a solution based on dynamic arrays, as well as a solution based on linked lists. We will describe the operations, and analyze their performance.

2.1.1 Dynamic array

Our first naive solution is based on the *dynamic array* data structure, as described in Cormen, Leiserson, Rivest and Stein [6]. Intuitively, the data structure is based around an array whose size is increased in powers of two, which allows for amortized $O(1)$ insertion and deletion of elements from the back of the structure. Additionally, as the dynamic array is based on an array, it allows $O(1)$ element access and modification given an index.

We observe that, by Cormen et al [6], dynamic arrays can only be copied in linear time. We also observe that dynamic arrays can only be concatenated and split in amortized linear time.

Preliminaries

We will use A to refer to the dynamic array representation of the list $L = (a_1, a_2, \ldots, a_n)$, and A' to refer to the dynamic array representation of the list $L' = (a'_1, a'_2, \ldots, a'_m)$. We assume that any element of L is pairwise distinct from any element of L'. We will use $A[i]$ to refer to the element at index i in A. We will use $|A|$ to refer to the number of elements in A. We assume that we have access to any element in A, as well as $|A|$, in $O(1)$ time.

2.1.2 Quickselect

Our dynamic array-based solution will employ the quickselect algorithm, also known as Hoare's selection algorithm [12]. This algorithm is based on quicksort [13], and allows efficient location of the k-th smallest element in an unordered array. It also has the desirable property of partially-sorting the array, allowing us to more efficiently locate the elements larger than the kth smallest.

In order to describe quickselect, we first describe an important subroutine partition $(L, \ell, r, \mathsf{p\_ind})$, where $L = [a_1, a_2, \ldots, a_n]$ denotes an unsorted array. This subroutine arranges all elements whose indices range from ℓ to r inclusive as follows: all elements less than $a_{\mathsf{p\_ind}}$, then $a_{\mathsf{p\_ind}}$, then all elements greater than $a_{\mathsf{p\_ind}}$. It does this by first pivoting $a_{\mathsf{p\_ind}}$ to the index r, and then iterating over all elements between the indices ℓ and r, swapping smaller elements back toward the front while finding an insertion point for $a_{pivotind}$. Lastly, $a_{\mathsf{p\_ind}}$ is put into that insertion point, and its index is returned. For an exact description of partition, see Algorithm 2.

Algorithm 2 partition$(L, \ell, r, \mathsf{p_ind})$

```
1: pivot_value ← L[p_ind]
2: Swap L[p_ind] and L[r]          ▷ Move pivot element to the end of the working range
3: store_index ← ℓ
4: for i ∈ ℓ ... (r − 1) do
5:     if L[store_index] < pivot_value then
6:         Swap L[store_index] and L[i]
7:         store_index ← store_index + 1
8: Swap L[r] and L[store_index]
9: return store_index                  ▷ Move pivot element to its final place
```

We now describe quickselect(L, ℓ, r, k). We first check if $\ell = r$; if so, then there is only one element in the list, so we return it. Otherwise, we enter a loop and perform the following at each iteration:

1. Select a random index p_ind between ℓ and r inclusive.

2. Call partition$(L, \ell, r, \mathsf{p_ind})$ and store the return value as m.

3. If $k = m$, return the element of L at m; otherwise:

   (a) If $k < m$, set r to be $\mathsf{p\_ind} - 1$.

   (b) Else, set ℓ to be $\mathsf{p\_ind} + 1$.

For an exact description, see Algorithm 3.

Algorithm 3 quickselect(L, ℓ, r, k)

```
1: if ℓ = r then
2:     return L[ℓ]
3: while do
4:     p_ind ← a random index between ℓ and r inclusive
5:     p_ind ← partition(L, ℓ, r, p_ind)
6:     if k = p_ind then
7:         return L[k]
8:     else if k < p_ind then
9:         r ← p_ind − 1
10:    else
11:        ℓ ← p_ind + 1
```

We observe that by Algorithms 2 and 3, after finding the kth smallest element x in L, all elements after x in L are larger than x. We make use of this observation to perform the psort operation required by the dynamic partial sorting problem.

Considerations

Let n be the length of the input L to quickselect. The time complexity of quickselect is dependent on the elements chosen for p_ind, but is known to be $O(n)$ in the average case, based on similar arguments to the performance of quicksort [6].

Additionally, quickselect is an 'in-place' algorithm, as it changes L in the process of execution. This is not desirable in the case of the dynamic partial sorting problem, as it would cause the original order of the list to become destroyed in the process, complicating link and cut operations. This can be avoided by copying the structure before performing this operation, which is taken into account as part of the naive solution we present.

psort

To perform psort(A, k), we first copy A to create $c(A)$. We then call quickselect $(c(A), 1, |c(A)|, k)$ to retrieve the kth smallest element x. In order to retrieve the other

$k-1$ elements, we sort all elements in $c(A)$ to the right of x, then return them in order, followed by x.

This version of psort is correct by Algorithms 3 and 2. The time complexity of psort(A, k) is the time to copy A, plus the time to execute quickselect, followed by the time required to sort $k-1$ items. Assuming an asymptotically-optimal sorting algorithm, this gives a time complexity of $O(|A| + k\log(k))$.

changeval

To perform changeval(A, i, n), we simply modify the element at the given index to n. This requires $O(1)$ time by the description of a dynamic array.

link

Performing link(A, A') can only be done by removing elements from A' and inserting them into A, from the first to the last index of A'. This gives a time complexity of amortized $O(|A'|)$ to this operation.

cut

To perform cut(A, i), we first create a new dynamic array B, and then remove all elements after the element at index i from A and insert them into B. This gives an amortized $O(|A|)$ time complexity.

Theorem

We summarize this analysis in the following theorem, whose proof follows from Hoare [12] and Cormen et al [6].

Theorem 1. The dynamic array data structure solves the dynamic partial sorting problem, with the following asymptotic time complexity:

- psort(A, k): $O(|A| + k \cdot \log(k))$
- changeval(A, i, n): $O(1)$
- link(A, A'): $O(|A'|)$
- cut(A, i): $O(A)$

Limitations

The running time of the dynamic array-based implementation of psort is linear with respect to the number of numbers in the structure psort is being called on. Additionally, with the exception of changeval, the other update operations for the dynamic array are also linear with respect to the number of numbers in the structures being operated on. These make it unsuitable as a solution to the dynamic partial sorting problem, as it would be very inefficient.

2.1.3 Linked list

We now describe our second naive solution, based on linked lists. We assume that linked lists are doubly-linked, which means that we can access both the start and end nodes of the list in $O(1)$ time. Accessing an item in a linked list given an index requires linear time. This corresponds to the definition given in Cormen, Leiserson, Rivest and Stein [6].

Preliminaries

We will use M to refer to the linked-list representation of the list $L = (a_1, a_2, \ldots, a_n)$, and M' to refer to the linked-list representation of the list $L' = (a'_1, a'_2, \ldots, a'_m)$. We will use $|M|$ to refer to the number of elements in M.

psort

The same approach cannot be taken for linked lists as taken for dynamic arrays, as quickselect's performance relies on the ability to swap two elements in $O(1)$ time; thus, under the restrictions of linked lists, it would cause the time complexity of quickselect to become quadratic with respect to the size of the input structure. As this is worse than simply sorting the collection each time a psort is requested, this approach will not be considered.

Instead, we perform psort(M, k) using the *heapsort* approach [6]. This involves the following steps:

1. Create a new priority queue Q.
2. Insert all elements of M into Q.
3. Remove and return k elements from Q.

Based on Lemma 1, this operation runs in $O(|M| \cdot \log(|M|))$.

changeval

As by Section 1.1, for changeval(M, i, n) we assume that we have a reference to the element at index i in M, we can simply change its value to n. Like in the dynamic array solution, this is a $O(1)$ operation.

link

To perform link(M, M'), we only need to connect the last element of M with the first element of M'. As we have access to the last element of M in $O(1)$, this is a $O(1)$ operation.

cut

Performing cut(M, i) simply requires us to separate the linked list M into two lists by breaking the references that connect the element at index i to the subsequent element in M. As by Section 1.1, we have a reference to the element at index i, this can be performed in $O(1)$ time.

Theorem

We summarize this analysis in the following theorem, whose proof follows from Cormen et al [6].

Theorem 2. The linked list data structure solves the dynamic partial sorting problem, with the following asymptotic time complexity:

- psort(M, k): $O(|M| \cdot \log(|M|))$
- changeval(M, i, n): $O(1)$
- link(A, A'): $O(1)$
- cut(A, i): $O(1)$

Limitations

Although the linked list solution to the dynamic partial sorting problem has good performance for changeval, link and cut, its performance of psort is essentially the same as sorting the entire list. This makes it very dependent on the number of numbers stored, which makes it an unsuitable choice.

2.2 Dynamic trees for dynamic partial sorting

As can be seen from Section 2.1, the naive solutions presented for the dynamic partial sorting problem have poor performance. Thus, we consider an alternative method of supporting the dynamic partial sorting operations. We propose the use of *dynamic trees* for this purpose, as they have an inherent ability to store ordered data while enabling dynamic modification of the data they represent. We define dynamic trees, and also describe two common varieties – the *red-black tree* and the *splay tree*. Lastly, we describe the influence of these data structures on our solutions to the dynamic partial sorting problem.

2.2.1 Definition of dynamic tree

Intuitively, a dynamic tree data structure is a binary tree, which is designed to store ordered data. It also permits several update operations. Formally, we make the following definition:

Definition 4. A *dynamic tree* is a data structure D representing the ordered list of numbers $n_1, n_2, \ldots, n_m$. The structure also permits the following operations:

- $\mathsf{changeval}(D, i, x)$: Replaces the number at index i with x.
- $\mathsf{link}(D, D', x)$: Links together D and another dynamic tree D' by way of the new element x. This must be done in such a way thatthe order of both trees is maintained, with items in D' ordered after the items in D in the resulting new dynamic tree. The new item x must be ordered between the items in D and the items in D'.
- $\mathsf{cut}(D, i)$: Separates D into the new dynamic trees D_1, D_2, D_3, such that D_1 stores all numbers with indices below i, D_2 stores the number with index i, and D_3 stores all numbers with indices above i.

All of these operations must be $O(\log(m))$. We assume that in changeval and cut, we have a reference to the element representing the ith number in the list; thus, no searching is necessary.

This definition mirrors the definition of self-balancing binary trees given in Tarjan [22]. Such structures are often used to solve the dictionary problem [2], which requires search, delete and insert operations, with the additional constraint on the need to maintain some kind of order over the data being stored.

The earliest dynamic tree was the *AVL tree* [1]. This structure used tree rotations to ensure order was maintained, relying on a 'balance condition' to ensure that the time complexity of the operations remained logarithmically-bounded.

2.2.2 Red-black tree

The red-black tree was originally proposed by Rudolf Bayer [4], but was popularized by Guibas and Sedgewick [10]. We give a definition of the red-black tree below.

Definition 5. A *red-black tree* T of the sorted list $L = (a_1, a_2, \ldots, a_n)$ is a full binary tree. Each node u in T also has a field $\mathsf{key}(u)$, which is the index of $\mathsf{val}(u)$ in the list L. Every leaf node v in T has $\mathsf{val}(v) = \mathsf{nil}$.

For every internal node u, the following holds:

1. If $\mathsf{val}(\mathsf{left}(u)) \neq \mathsf{nil}$, then $\mathsf{key}(u) > \mathsf{key}(\mathsf{left}(u))$.
2. If $\mathsf{val}(\mathsf{right}(u)) \neq \mathsf{nil}$, then $\mathsf{key}(u) < \mathsf{key}(\mathsf{right}(u))$

Additionally, T has the following properties:

1. Every node u in T is either red or black.
2. The root of T is black.
3. All leaves in T are black.
4. Every red node has only black children.
5. For any internal node u, every path from u to any descendant leaf must contain the same number of black nodes.

The correctness of the following lemma follows from Definition 5.

Lemma 2. *Let R be the red-black tree of $L = (a_1, a_2, \ldots, a_n)$. $h(R) \leq 2 \cdot \lceil \log_2(n) \rceil$.*

We now describe how the red-black tree can be used as an implementation of a dynamic tree. Throughout, let R, R' be red-black trees.

changeval

By the problem description in Section 4, when performing $\mathsf{changeval}(R, i, x)$, we have a reference to the node u whose key is i. As we only change $\mathsf{val}(u)$, not $\mathsf{key}(u)$, we don't have to make any changes to the structure of R. Thus, we can just set $\mathsf{val}(u)$ to x, which can be done in $O(1)$ time.

fix_up for red-black trees

When performing link and cut operations on red-black trees, it is possible that the red-black tree properties will be violated. To correct this, we define a repair operation $\mathsf{fix\_up}(R, u)$, where u is a node in the red-black tree R. This operation is based on Cormen et al [6], and is described fully in Algorithm 4.

Algorithm 4 $\mathsf{fix\_up}(u)$

```
1: while p(u) is red do
2:     if p(u) = left(p(p(u))) then
3:         y ← right(p(p(u)))
4:         if y is red then
5:             Paint p(u) and y black
6:             Paint p(p(u)) red
7:             u ← p(p(u))
8:         else if u = right(p(u)) then
9:             u ← p(u)
10:            rotate_left(u)
11:        else
12:            Paint p(u) black
13:            Paint p(p(u)) red
14:            rotate_right(p(p(z)))
15:    else
16:        Do the same as the previous clause, except with right and left exchanged
17: Paint the root of R black
```

This operation guarantees the red-black tree properties after a link or cut operation, based on Cormen et al [6]. We observe that in Algorithm 4, the while-loop performs a number of iterations that is bounded by $h(R)$. By Lemma 2, this is $O(\log(n))$, where n is the number of numbers stored in R.

link

We can now describe the $\mathsf{link}(R, R', x)$ operation. We give only the case where $h(R) > h(R')$; the other case is symmetric. To perform $\mathsf{link}(R, R', x)$, we first create a new node u such that $\mathsf{val}(u) = x$, which we paint red. We then follow right child pointers from the root of R until we find a node v such that $h(v) = h(R')$. We then separate the subtree of v from the rest of R. If v is not black, we paint it black as part of this process. We then set $\mathsf{left}(u) = v$, $\mathsf{right}(u) = R'$, and reconnect u to R in the same position that v used to be in. We then call $\mathsf{fix\_up}(R, u)$ to ensure that the tree retains the red-black properties. For an exact description, see Algorithm 5.

Algorithm 5 $\mathsf{link}(R, R', x)$ $(h(R) > h(R')$ case)

```
1: Create a new red node u with val(u) = x
2: (r, r') ← the root of R, R' respectively
3: while h(r) > h(r') do
4:     r ← right(r)
5: Separate r from the rest of R
6: Paint r black
7: (left(u), right(u), p(u)) ← r, r', p(r)        ▷ Place u where r used to be
8: fix_up(R, u)
```

The correctness of this operation follows from the correctness of fix_up. The time complexity of this operation is based on the difference in height between R and R', due to the traversal to locate v and the subsequent call to fix_up. Thus, this operation is $O(|h(R) - h(R')|)$, which by Lemma 2 is $O(\log(n))$, where $n = \max\{h(R), h(R')\}$.

cut

We perform cut (R, i) using the approach taken by Tarjan [22]. By Definition 4, we assume that we have a reference to a node u such that $\mathsf{key}(u) = i$. We initialize the current node x, the previous node y, the left tree R_1 and the right tree R_2 to be the parent of u, u itself, the left subtree of u and the right subtree of u respectively. We then repeat the following until x is nil:

1. If y is the left child of x, simultaneously replace x, y, R_2 with $\mathsf{p}(x), x, \mathsf{link}(R_2, x, \mathsf{right}(x))$.

2. Otherwise, simultaneously replace x, y, R_1 with $\mathsf{p}(x), x, \mathsf{link}(\mathsf{left}(x), x, R_1)$.

We must also paint $\mathsf{right}(x)$ or $\mathsf{left}(x)$ respectively black prior to performing this operation. We then return R_1, x, R_2. For an exact description, see Algorithm 6. We will use u to refer to the node with key i in this description.

Algorithm 6 $\mathsf{cut}(R, u)$

```
1: (x, y, R1, R2) ← p(u), u, left(u), right(u)
2: while x ≠ nil do
3:     if y = left(x) then
4:         Paint right(x) black
5:         (x, y, R2) ← p(x), x, link(R2, x, right(x))
6:     else
7:         Paint left(x) black
8:         (x, y, R1) ← p(x), x, link(left(x), x, R1)
9: return R1, u, R2
```

The correctness of this operation follows from the description of the red-black tree and Algorithm 5. The time complexity of this operation is $O(\log(n))$ (where n is the number of values in R), as proved by Tarjan [22].

Theorem

We summarize this analysis in the following theorem. Its proof follows from Definition 5, Guibas and Sedgewick [10], and Tarjan [22].

Theorem 3. The red-black tree implements the dynamic tree, with the following asymptotic time complexity:

- $\mathsf{changeval}(R, i, x)$: $O(1)$
- $\mathsf{link}(R, R', x)$: $O(\log(\max\{h(R), h(R')\}))$
- $\mathsf{cut}(R, i)$: $O(\log(|R|))$

2.2.3 Splay tree

The splay tree, first introduced by Sleator and Tarjan [21], was designed to solve the dictionary problem, as well as perform modification operations, in amortized time. It also makes previously-accessed elements easy to access again. It also introduced the concept of representing a tree as a collection of paths [22].

Intuitively, a splay tree can be thought of as a collection of vertex-disjoint paths. To define the paths, the edges of the tree are partitioned into two kinds – a 'solid' edge, and a 'dashed' edge. At most one solid edge can enter any vertex; this partitions the tree into vertex-disjoint *solid paths*. See Figure 2.1 for an example of such a tree.

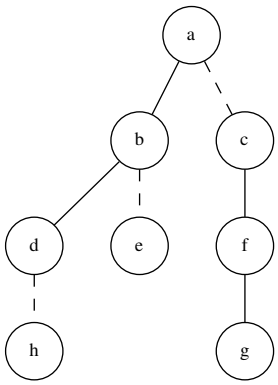

Figure 2.1: An example of a splay tree, separated into the paths (h), (e), (d, b, a) and (g, f, c).

Formally, we define splay trees as follows:

Definition 6. A *splay tree* S of a list $L = (a_1, a_2, \ldots, a_n)$ is a binary tree. Additionally, each u in S also has a field $\mathsf{key}(u)$, which is the index of $\mathsf{val}(u)$ in the list L.

For every internal node v, the following holds:

1. If $\mathsf{left}(v) \neq \mathsf{nil}$, then $\mathsf{key}(v) > \mathsf{key}(\mathsf{left}(v))$
2. If $\mathsf{right}(v) \neq \mathsf{nil}$, then $\mathsf{key}(v) < \mathsf{key}(\mathsf{right}(v))$
3. At most one edge connecting v to one of its children is a *solid* edge; all other edges connecting v to its children are *dashed* edges.

splay

In order to permit the performance bounds required by the definition of dynamic trees, it is necessary to define the *splaying* operation $\mathsf{splay}(u)$ for splay trees. This performs a sequence of rotations to make u the root of its tree. There are three cases:

1. If $\mathsf{p}(u)$ is the root, then we perform one rotation (either left or right, depending on whether u is a left or right child) to make u the root.
2. If $\mathsf{p}(u)$ is not the root, and u and $\mathsf{p}(u)$ are either both left children or both right children, we first rotate $\mathsf{p}(u)$, then rotate u. We demonstrate this with left rotations in Figure 2.2.
3. Otherwise, we first rotate u upwards, and then rotate u upwards again. We demonstrate this in Figure 2.3.

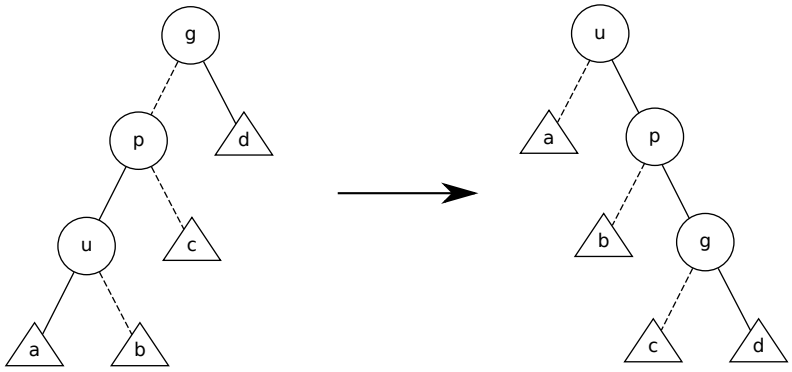

Figure 2.2: Case 2 of splay. The triangular nodes indicate arbitrary subtrees.

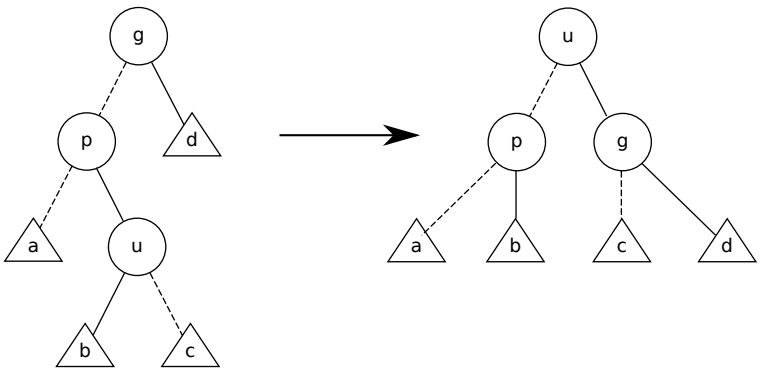

Figure 2.3: Case 3 of splay. The triangular nodes indicate arbitrary subtrees.

We repeat these steps until u becomes the root of its tree. Along the way, we also make any edge connecting u and $\mathsf{p}(u)$ solid (changing any other child edges to dashed as needed). For an exact description, see Algorithm 7. In the description, when we say to rotate a node u 'upwards', it means to perform $\mathsf{rotate\_left}(u)$ if u is a right child, and $\mathsf{rotate\_right}(u)$ if u is a left child.

Algorithm 7 $\mathsf{splay}(u)$

```
1:  if u is not the root then
2:      (p, g) ← p(u), p(p(u))
3:      Make the edge between u and p solid
4:      if g ≠ nil then
5:          if u, p are both left children or both right children then
6:              Rotate p upwards
7:              Rotate u upwards
8:          else
9:              Rotate u upwards
10:             Rotate p upwards
11:         splay(u)
12:     else
13:         Rotate u upwards
```

We give the following lemma based on Sleator and Tarjan [21] and Tarjan [22] without a proof.

Lemma 3. *Let S be a splay tree with n nodes. Let m denote the number of* splay *operations performed on S. As m approaches infinity, the height of S approaches $O(\log(n))$.*

We observe that Lemma 3 gives the splay operation an amortized time complexity of $O(\log(n))$. Splaying also preserves the splay tree properties defined previously.

changeval

The changeval(S, i, n) operation for splay trees is similar to the equivalently-named red-black tree operation. As by Definition 4, we have a reference to the element whose key is equal to i, we can simply change its value to n, requiring $O(1)$ time.

link and cut

To perform link(S, S', x), we first perform splay(u), where u is the node such that key(u) is the largest in S. We then create a new node u with val$(u) = x$. We set u as the right child of the root of S, and then set S' as the right child of u. This operation preserves the ordering property of the splay tree. By Lemma 3, this operation has amortized $\log(|S|)$ time complexity.

When we perform cut(S, i), by Definition 4, we have a reference to the node u whose key is i. We first use splay(u) to make u the root of S. We then separate the left and right subtrees of u, and return u's left subtree, u itself, and u's right subtree. Similarly to link, this operation has amortized $O(\log(|S|))$ time complexity.

The correctness of both of these operations is guaranteed by the definition of the splay operation and the key-based ordering property of the nodes of a splay tree given by Definition 4.

Theorem

We summarize this analysis in the following theorem. Its proof follows from Lemma 3, Sleator and Tarjan [21] and Tarjan [22].

Theorem 4. The splay tree implements the dynamic tree, with the following asymptotic time complexity:

- changeval(S, i, x): $O(1)$
- link(S, S', x): Amortized $O(\log(|S|))$
- cut(S, i): Amortized $O(\log(|S|))$

Representing trees as collections of paths

We can use the definition of the splay tree above to represent paths connected by solid edges as trees in their own right, as described in Tarjan [22]. This approach stores any splay tree S as a set of node-disjoint trees, each representing a path connected by solid edges in S. Thus, performing a splay operation would involve link and cut operations as solid and dashed edges changed.

Such an approach to representing trees has several advantages – it allows for faster searching, and allows parallel operations on the structure, as well as working better with external storage. These advantages are discussed in more detail by Tarjan [22].

2.2.4 Suitability for the dynamic partial sorting problem

The dynamic tree has several desirable properties that allow us to improve on the time complexity of the naive solutions to the dynamic partial sorting problem. In particular, the dynamic tree allows for modification of stored data, as well as linking and cutting of data structures, while maintaining some kind of order over the elements in all collections.

The red-black tree implementation of the dynamic tree serves as a direct inspiration for the tournament tree data structure described in Chapter 3. In particular, the maintenance of the height of a red-black tree while preserving order over elements by use of rotation is used directly by the tournament tree data structure.

The splay tree's capability to represent a tree as a set of modifiable paths serves as the basis for the layered tournament tree data structure described in Chapter 4. However, unlike the splay tree, the layered tournament tree continues the decomposition process recursively, decomposing each path-representing tree into a collection of paths as well, thus forming the 'layers' of the resulting structure.

In both cases, however, some modifications are needed to the ideas provided by these structures to allow efficient solutions to the query operation of the dynamic partial sorting problem. The approach taken is designed to allow maintaining the original (unsorted) order of a list while allowing querying of order statistics – see Chapters 3 and 4 for more details.

Chapter 3

Tournament Trees for Partial Sorting

In this chapter, we describe the tournament tree (or TT) data structure, and describe and analyze it as a method of solving the dynamic partial sorting problem. We also discuss its limitations in leadup to Chapter 4.

3.1 The tournament tree

The tournament tree data structure is inspired by the tournament sort algorithm, which uses the idea of a single-elimination tournament[16]. Formally, we define it as follows:

Definition 7. Let $L = (a_1, a_2, \ldots, a_n)$ be a list. A *tournament tree* $T = (V, E)$ of L is a balanced, full tree data structure that satisfies the following properties:

- T has exactly n leaves, whose values are $a_1, a_2, \ldots, a_n$ respectively.
- For every internal node $v \in V$, if $\mathsf{val}(\mathsf{left}(v)) = a_i$ and $\mathsf{val}(\mathsf{right}(v)) = a_j$, then $i < j$ and $\mathsf{val}(v) = \mathsf{min}\{a_i, a_j\}$.

We can use a TT to represent a list by storing the elements of the list as leaf nodes, in the same order as in the list. See Figure 3.1 for an example.

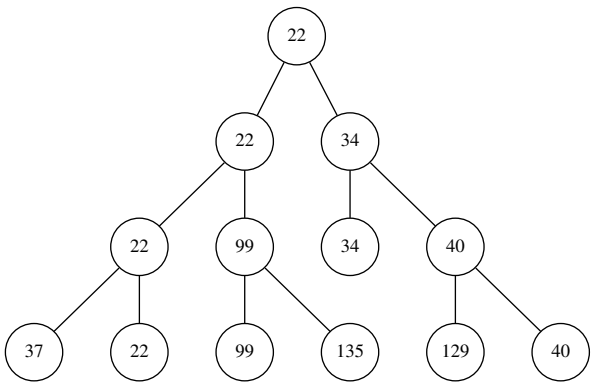

Figure 3.1: A TT of the list $L = (37, 22, 99, 135, 34, 129, 40)$.

We also define the following concepts.

Definition 8. Let $T = (V, E)$ be a TT. For any $u, v \in V$, we write $u \sim v$ if $\mathsf{val}(u) = \mathsf{val}(v)$.

As we use TTs as representations of lists of pairwise-distinct numbers, the equivalence relation $\sim$ partitions the nodes in a TT into disjoint paths.

Definition 9. The *principal path* $\mathsf{Path}(u)$ of a node u is the equivalence class $\{v \mid u \sim v\}$. The *value* of $\mathsf{Path}(u)$ is $\mathsf{val}(u)$.

Intuitively, we view $\mathsf{Path}(u)$ as a path that *originates* from a leaf in a TT, and extends upwards. We can view every node in $\mathsf{Path}(u)$ as 'gaining' its value from this leaf. Hence, we single out this leaf, and define the following.

Definition 10. Let P be a principal path. The *origin* of P is the leaf in P.

When we refer to 'a principal path' in a TT T, we mean $\mathsf{Path}(u)$ for some u in T. In addition, we define the following concept:

Definition 11. The *subordinate* $\mathsf{sub}(u)$ of u is a child of u that does not belong to the same principal path as u.

See Figure 3.1 for a visual representation of each of the preceding definitions.

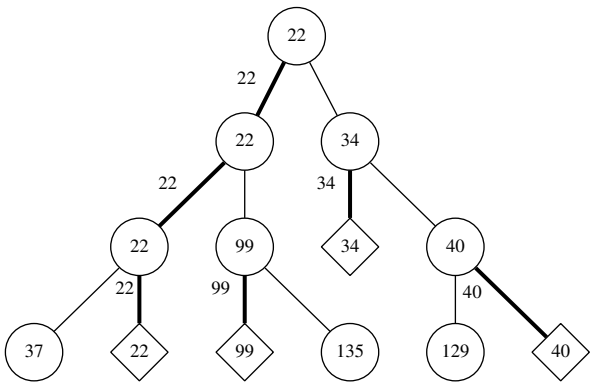

Figure 3.2: A TT of the list $L = (37, 22, 99, 135, 34, 129, 40)$. Principal path edges are bolded and labelled with the value of their principal path. The origins of principal paths are diamond-shaped.

Lastly, we observe that, as a TT is balanced, its height is logarithmic with respect to the number of its leaves. More specifically, we prove the following lemma:

Lemma 4. *If T is a TT with $n > 0$ leaves, then the height of T is not more than $\log_\varphi(n)$, where $\varphi = \frac{\sqrt{5}+1}{2}$ is the golden ratio.*

Proof. It suffices to show that the least number of leaves $f(h)$ in any TT with height $h \geq 0$ is φ^h. We make the following observation. Note that here, we use the fact that any TT is balanced and full.

$$f(h) \geq \begin{cases} 1 & \text{if } h = 0, \\ 2 & \text{if } h = 1, \\ f(h-1) + f(h-2) & \text{otherwise} \end{cases}$$

□

3.2 Dynamic partial sorting using TTs

We now describe an algorithm for solving the dynamic partial sorting problem using TTs. Throughout, we use $L = (a_1, a_2, \ldots, a_n)$ to represent a list, and $T = (V, E)$ as the TT representing that list as per 3.1.

3.2.1 psort

For any $L = (a_1, a_2, \ldots, a_n)$, we list its elements in monotonically increasing order as $x_1, x_2, \ldots, x_n$. By definition, the root of T has the smallest value; therefore, to find x_1, we only need to return the root. For finding the subsequence x_is, we make the following observation.

Observation 1. For any $1 \leq i < n$, let P_i denote the principal path in T with the value x_i. The number x_{i+1} is $\mathsf{val}(u)$, where u is a subordinate of some node in

$$P_1 \cup P_2 \cup \cdots \cup P_i$$

Hence, to compute the $(i+1)$th smallest number in L, we must examine all principal paths whose origins are $x_1, x_2, \ldots, x_i$, as well as the values of the subordinates of nodes on those paths.

Based on the above observation, we perform $\mathsf{psort}(T, k)$ as follows:

1. Output the root of T as well as its value.
2. Whenever we output a node u, we examine the subordinates of all nodes in $\mathsf{Path}(u)$.
3. Continue this process until we return $\mathsf{min}\{k, n\}$ items.

During this process, we use a priority queue to store the nodes examined so far. We describe the operation formally in Algorithm 8.

Algorithm 8 $\mathsf{psort}(T, k)$

```
1: u ← the root of T
2: Make a new priority queue Q
3: for k iterations do
4:     Output val(u)
5:     while u is not a leaf do
6:         insert(Q, sub(u), val(sub(u)))
7:         u ← the child of u with the same value as u
8:     if empty(Q) then
9:         break
10:    else
11:        u ← delete_min(Q)
```

Lemma 5. *Let T be a TT, $n = \mathsf{size}(T)$, and $k > 0$. Then, psort(T, k) is $O(k \cdot \log(n))$.*

Proof. By Lemma 4, every path of T is bounded by $\log_\varphi(n)$. This means that after the $\mathsf{psort}(T, k)$ operation outputs an element, it inserts at most $\lceil \log_\varphi(n) \rceil$ nodes into the priority queue. Hence, the size of the priority queue is bounded by $k \cdot \log_\varphi(n)$. Therefore, the time complexity of this operation is $O(k \cdot \log(n))$. □

3.2.2 Update operations and fix_up

Any update operation performed on a TT can potentially cause it to become unbalanced or have incorrect values for its internal nodes. Thus, we define a 'repair' operation which is used to correct an unbalanced TT and any internal node values which may require changing.

The operation walks the path from its initial node to the root of the tree. At each node u on this path, we must perform two tasks:

1. Check whether $T(u)$ is unbalanced; if it is, we perform a rotation to correct it, then resume the operation.

2. Correct $\mathsf{val}(u)$ to be the minimum of the values of its children.

For an exact description, see Algorithm 9.

Algorithm 9 $\mathsf{fix\_up}(u)$

```
1: (v, v') ← left(u), right(u)
2: if |h(T(v)) − h(T(v'))| > 1 then
3:     if h(T(v)) > h(T(v')) then
4:         rotate_right(v)          ▷ This will cause u to become the right child of v
5:     else
6:         rotate_left(v')          ▷ This will cause u to become the left child of v'
7:     fix_up(u)
8: val(u) ← min{val(v), val(v')}
9: if u is not the root then
10:    fix_up(p(u))
```

3.2.3 changeval

To perform $\mathsf{changeval}(T, i, n)$, we first change the value of the ith leaf u to n. This can make the values of every ancestor of u incorrect; thus, we call $\mathsf{fix\_up}(\mathsf{p}(u))$ to fix T. For an exact description, see Algorithm 10. Based on Section 1.1, the description will use u as a parameter to represent the reference we have to the ith leaf of T.

Algorithm 10 $\mathsf{changeval}(T, u, n)$

```
1: val(u) ← n
2: if u is not the root then
3:     fix_up(p(u))
```

Lemma 6. *Let T be a TT, and let $n = \mathsf{size}(T)$. Then, $\mathsf{changeval}(T, u)$ is $O(\log(n))$.*

Proof. By Lemma 4, fix_up must modify at most $\lceil \log_\varphi(n) \rceil + 1$ nodes. Each such modification consists of an assignment, a two-way comparison, and possibly a stopping check, each of which requires constant time. Additionally, as no nodes are added or removed, no rotations need to be performed by fix_up. Thus, we have at most $3 \cdot (\lceil \log_\varphi(n) \rceil + 1)$ constant-time operations, which makes $\mathsf{changeval}(T, u)$ a $O(\log(n))$ operation. □

3.2.4 link

Let $L' = (a'_1, a'_2, \ldots a'_m)$ denote a list of numbers such that every number in L' is different to any number in L, and let T' be the TT of L'. Without loss of generality, we assume that $h(T) > h(T')$; the other case is symmetric.

To perform $\mathsf{link}(T, T')$, we following right child pointers from the root of T until we reach a node u such that $h(T(u)) \leq h(T')$. We then cut the subtree $T(u)$ away from T, and replace it with a new node v. We set $\mathsf{left}(v)$ to be u, $\mathsf{right}(v)$ to be the root of T', and $\mathsf{val}(v)$ as the minimum of the values of its children. This change can cause the new tree to become unbalanced, and may also require us to modify the values of the nodes on the path from v to the root. To solve these problems, we call $\mathsf{fix\_up}(\mathsf{p}(v))$. See Algorithm 11 for an exact description.

Algorithm 11 $\mathsf{link}(T, T')$ $(h(T) > h(T')$ case)

1: $(u, u') \leftarrow$ the root of T, the root of T'
2: **while** $h(T(u)) > h(T')$ **do**
3: $\quad u \leftarrow \mathsf{right}(u)$
4: Create a new node v
5: $(\mathsf{right}(\mathsf{p}(u)), \mathsf{left}(v), \mathsf{right}(v)) \leftarrow v, u, u'$
6: $\mathsf{val}(v) \leftarrow \mathsf{min}\{\mathsf{val}(u), \mathsf{val}(u')\}$
7: $\mathsf{fix\_up}(p(v))$

Lemma 7. *Let T, T' be TTs, and let $m = |h(T) - h(T')|$. Then, $\mathsf{link}(T, T')$ is $O(m)$.*

Proof. By the description of link, exactly one subtree of the resulting tree could be unbalanced, by at most an additional height difference of 1. Thus, we observe that that $\mathsf{link}(T, T')$ performs at most one rotation and up to m-many changes to the values of nodes while walking the path to the root. Therefore, the $\mathsf{link}(T, T')$ operation is $O(m)$. □

3.2.5 cut

To perform $\mathsf{cut}(T, i)$, we need to split the TT T at the ith leaf u to form two TTs: one containing all leaves to the left of u (and u itself), the other containing the rest. For

this operation, we first walk the path from u to the root of T, deleting every edge on the path and incident to it. We also remove any internal nodes which have no children as part of this process. This breaks T into a collection of subtrees, the root of each of which was a child of a node on the path from u to the root of T. We then link (using the link procedure described in 3.2.4) the subtrees containing leaves to the left of u (and u itself) in T to form a TT T_1, and the rest of the subtrees into another TT T_2. For an exact description, see Algorithm 12; as in Subsection 3.2.3, we will use u to denote the reference to the ith leaf of T.

Algorithm 12 $\mathsf{cut}(T, u)$

```
1: (x, y) ← p(u), u
2: Create two empty TTs T_1, T_2
3: T_1 ← T(y)
4: while x ≠ nil do
5:     if y = left(x) then
6:         T_2 ← link(T_2, T(right(x)))
7:     else
8:         T_1 ← link(T(left(x)), T_1)
9:     (y, x) ← x, p(x)
```

Lemma 8. *Let T be a TT, u be a leaf in T, and $n = size(T)$. Then* $\mathsf{cut}(T, u)$ *is* $O(\log(n))$.

Proof. Let $P = \{u_0, u_1, \ldots, u_k\}$ be the path in T from $u_0 = u$ to the root of T, where $u_{i+1} = \mathsf{p}(u_i)$ for all $0 \leq i < k$. By Algorithm 12, the $\mathsf{cut}(T, u)$ operation separates T into a collection of TTs

$$\widehat{T}_1, \widehat{T}_2, \ldots, \widehat{T}_k$$

where each $\widehat{T}_i$ is either the left or the right subtree of u_i. As T is balanced, one could easily prove by induction on i that

$$h(\widehat{T}_i) \leq 2i - 1$$

The $\mathsf{cut}(T, u)$ operation then iteratively joins the trees $\widehat{T}_1, \ldots, \widehat{T}_k$ to form two trees T_1, T_2, such that T_1 contains all leaves to the left of (and including) u, and T_2 contains the other leaves. By Lemma 7 that time time required for any link operation is linear on the height difference between the two trees being linked. The total running time of

the sequence of link operations performed is therefore at most

$$2 \cdot \sum_{i \geq 1}^{k-1} \left(h\left(\widehat{T}_{i+1}\right) - h\left(\widehat{T}_i\right) \right) = 2\left(h\left(\widehat{T}_k\right) - h\left(\widehat{T}_1\right) \right)$$
$$\leq 2(2k-1)$$

The value of k is at most $h(T)$, which is bounded by $\log_\varphi(n)$ according to Lemma 4. Thus, the total time required for $\mathsf{cut}(T, u)$ is $O(\log(n))$. □

3.3 Limitations of the TT

Although the TT is a large improvement on the naive solutions presented in Chapter 2, it still has a major limitation in that its psort operation depends on both the query size and the number of elements stored. In practical applications, where n could be much larger than k, it is desirable to make the running time of *psort* independent of n.

Thus, in Chapter 4, we develop an additional data structure designed to solve the dynamic partial sorting problem while minimizing the influence of the size of the structure on the *psort* operation.

Chapter 4

Layered Tournament Trees

In this chapter, we present an extension of the tournament tree, designed to solve the dynamic partial sorting problem. This data structure is designed such that the running time of psort operations on it is (almost) independent of the number of numbers it stores. We call this structure the *layered tournament tree* (or LTT).

4.1 The LTT

We describe the layered tournament tree data structure. Throughout, let $L = (a_1, a_2, \ldots, a_n)$ be a list of pairwise-distinct numbers.

Intuitively, an LTT of L maintains a number of layers that extend downwards. Each layer consists of a number of TTs. The tree in the top layer is the TT of L; a tree in any lower layer stores a principal path in a tree in the layer above.

Formally, we make the following definitions:

Definition 12. Let T be the TT of L. Let $P = \{u_0, u_1, \ldots, u_k\}$ be a principal path in T, where u_0 is the origin of P, and $u_{i+1} = \mathsf{p}(u_i)$ for $0 \leq i < k$.

We define the *team* of P as the list of numbers

$$t = (\mathsf{val}(\mathsf{sub}(u_k)), \mathsf{val}(\mathsf{sub}(u_{k-1})), \ldots, \mathsf{val}(\mathsf{sub}(u_1)))$$

A *team* in T is a team of some principal path in T.

Note that only a principal path with more than one element has a team. We generally denote teams with a lower-case t.

Definition 13. We define a *layered TT* (*LTT*) of L as the set Γ_L of TTs that satisfies the following:

- If $L = (x)$, then $\Gamma_L = \{S\}$, where S consists of a single node u such that $\mathsf{val}(u) = x$.
- Otherwise, Γ_L contains a TT T of L, as well as an LTT Γ_t for each team t in T. More precisely,
$$\Gamma_L = \{T\} \cup \bigcup \{\Gamma_t \mid t \text{ is a team in } T\}$$

When the list L is clear from the context, we drop the subscript, weiting Γ_L simply as Γ.

Next, we give a precise definition of layers in an LTT Γ_L:

Definition 14. Let T be a TT in Γ_L. We say that

- T is in *layer* 0 of Γ_L if T is the TT of L; and
- T is in *layer* i of Γ_L, where $i > 0$, if T is a TT of a team t in a layer-$(i-1)$ tree in Γ_L.

If a team is in layer i of Γ, we call it a *layer-i team* in Γ; we call the tree of such a team a *layer-i tree* in Γ. The *layer number* of Γ is the maximum $i \geq 9$ such that a tree is in layer i of Γ.

Let P be a principal path in a layer-i tree of Γ, where $i \geq 0$ and the length of P is at least 1. By Definitions 12 and 14, Γ contains a TT T of the team of P in layer $(i+1)$. We call T the *team tree* of P. The *team tree* $\mathsf{Team}(u)$ of any node u is the team tree of the principal path containing u.

Recall that the origin of a principal path P s the leaf in P. We introduce the following notions:

- Suppose u is an internal node in a layer-i tree T in Γ. We define $\mathsf{down}(u)$ as the origin v of the principal path in $\mathsf{Team}(u)$ such that $\mathsf{val}(v) = \mathsf{val}(\mathsf{sub}(u))$.
- Suppose u is a leaf in a layer-i tree T in Γ, where $i > 0$. We define $\mathsf{up}(u)$ as the internal node v in a layer-$(i-1)$ tree such that $\mathsf{down}(v) = u$.

This finishes the description of the LTT data structure; see Figure 4.1 for an example of an LTT.

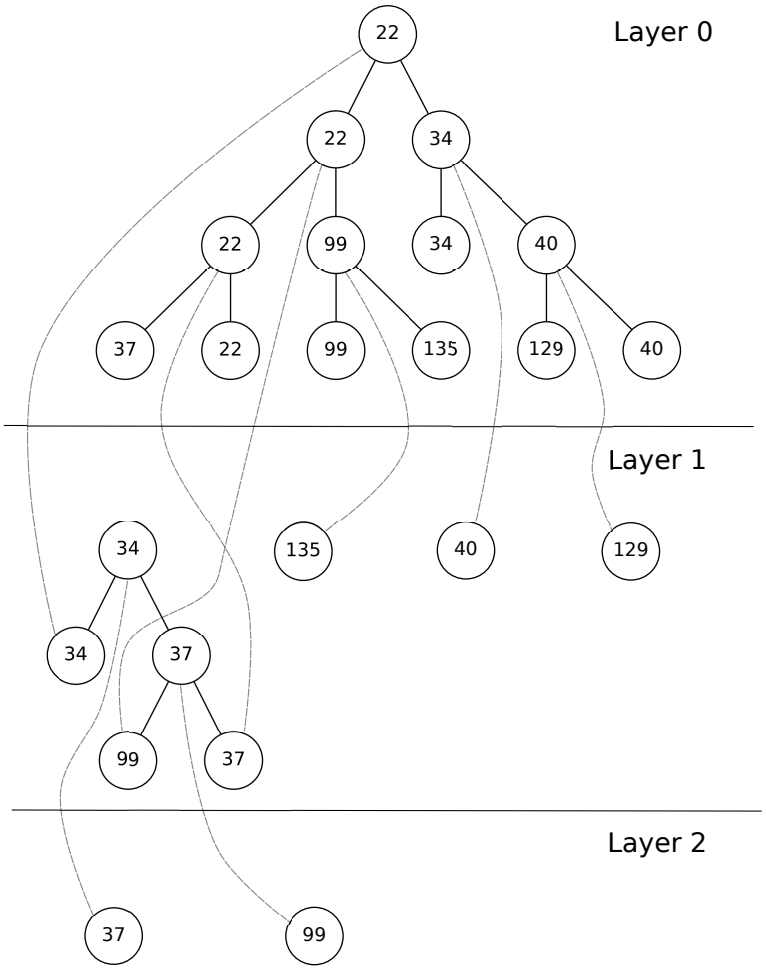

Figure 4.1: An LTT of the data in Figure 3.1. The up and down references are indicated by a dashed grey line. The layer number is 3.

4.2 Using an LTT for dynamic partial sorting

We now describe and analyze the psort, changeval, link and cut operations for the LTT. First, we describe the concept of iterated logarithm, which is necessary for the analysis of the time complexity of the LTT operations.

4.2.1 Iterated logarithm

The factors that determine the time complexity of the dynamic partial sorting operations are

1. The height of a layer-i tree in an LTT Γ for $i \geq 0$; and

2. The layer number of Γ.

To analyze the height of a layer-i tree in an LTT Γ for any $i \geq 0$, we recall the following function:

Definition 15. Let $b > 1$ be a real number. The *iterated logarithm with base* b $\log_b^*(n)$ of a number $n > b$ is the smallest $i \geq 0$ such that

$$\underbrace{\log_b \cdots \log_b}_{i}(n) \leq 1$$

It is known that the iterated logarithm function is defined for all $b \leq e^{\frac{1}{e}}$. The function $\log_b^n$ is known to be extremely slow-growing; for example, when b is the golden ratio φ, $\log_b^*(10^6) = 6$ and $\log_b^*(10^{10000}) = 7$.

More precisely, $\log_b^*(n)$ is the inverse of the power tower function with base b, defined as

$$b \uparrow\uparrow n = \underbrace{b^{b^{\cdot^{\cdot^{b}}}}}_{n}$$

Hence, we have the following lemma, which we state without a proof.

Lemma 9. *For any $b \geq e^{\frac{1}{e}}$, for all $i \geq 0$, we have*

$$\exists n' > 0 \forall n > n' \quad \log_b^*(n) \leq \underbrace{\log_b \cdots \log_b}_{i}(n)$$

Lemma 10. *For any $i \geq 1$, the size of any layer-i team is at most $\underbrace{\log_\varphi \cdots \log_\varphi}_{i}(n)$, where $n = |L|$. Furthermore, the layer number of the LTT of L is at most $\log_\varphi^*(n)$.*

Proof. By Lemma 4, the height of any TT is at most $\log_\varphi(m)$, where m is the number of leaves in the tree. The first statement of the lemma follows directly from the fact that the number of leaves in a layer-i tree is at most the height of a layer-$(i-1)$ tree in the LTT Γ. The second statement follows directly from the first statement. □

As an example, suppose that $|L| = 1,000,000$; then, the layer number of Γ_L is at most $\log_\varphi^*(10^6) \leq 6$.

4.2.2 psort

We describe the psort operation on the LTT data structure. As in Subsection 3.2.1, we use $x_1, x_2, \ldots, x_n$ to denote the numbers stored in the list L in monotonically increasing order.

The algorithm searches for, and outputs, each x_i iteratively by exploring the layer-0 TT T. The smallest number x_1 is the value of the root of T. If $k = 1$ or L contains only

one element, then the algorithm terminates. Otherwise, to find the second-smallest number x_2, let P be the principal path of the root of T. The number x_2 is the smallest number in the team of P.

Unlike in Algorithm 8, where we check through the subordinates of all nodes in P, in this algorithm we recursively apply the partial sort operation on the layer-1 team tree of P. In this way, the search continues in a lower layer.

To formally describe the psort operation on LTTs, we make the following definition:

Definition 16. Let L be a list of numbers with size n. An *iterator* of L is a data structure $\mathsf{It}(L)$ that supports an operation $\mathsf{next}(L)$ with the following property: In the ith call to $\mathsf{next}(L)$, the operation outputs x_i if $i \leq n$; otherwise, it outputs nil.

An iterator $\mathsf{It}(L)$ maintains a priority queue Q, which will contain nodes in T. The psort operation amounts to creating an iterator $\mathsf{It}(L)$ and calling $\mathsf{next}(L)$ a fixed number of times to obtain the partially-sorted numbers. To create an iterator for T, the algorithm simply creates an empty priority queue Q.

We will use u_i to denote the leaf with value x_i in the layer-0 tree of L for $1 \leq i \leq n$. For convenience, we consider the output of $\mathsf{next}(L)$ to be the leaf u_i rather than its value x_i.

We describe the $\mathsf{next}(L)$ operation by induction on the number of elements in L. When the operation $\mathsf{next}(L)$ is called the first time, we return the origin of $\mathsf{Path}(r)$, where r is the root of T. In subsequent calls to $\mathsf{next}(L)$, if L contains only one element, then the algorithm returns nil. Suppose L contains more than one element, and assume that we have defined iterators for lists with fewer elements than L.

Suppose $i \geq 1$ and we have made i calls to $\mathsf{next}(L)$, which outputs the nodes

$$u_1, u_2, \ldots u_i$$

Algorithm 13 implements the $\mathsf{next}(L)$ operation for the $(i+1)$th call.

Algorithm 13 $\mathsf{next}(L)$ (The $(i+1)$th call)

```
1: if Team(u_i) is not empty then
2:     Create an iterator It(Team(u_i))
3:     a ← next(Team(u_i))
4:     insert(Q, up(a), val(sub(a)))
5: if empty(Q) then
6:     Output nil
7: else
8:     x ← delete_min(Q)
9:     u_{i+1} ← the origin of Team(sub(x))
10:    b ← next(Team(x))
11:    if up(b) ≠ nil then
12:        insert(Q, up(b), val(sub(b)))
13:    Output u_{i+1}
```

Correctness of next

To show the correctness of Algorithm 13, we make the following definition:

Definition 17. Let v be a node in the TT T. The *superordinate* of v is a node $\mathsf{sup}(v)$ in T whose subordinate belongs to $\mathsf{Path}(v)$.

The *superordinate set* of a set U of nodes is

$$\mathsf{sup}(U) = \{\mathsf{sup}(v) \mid v \in U\}$$

For the next definition, we consider a set U of nodes in T.

Definition 18. A node v is an *U-candidate* if there is some $u \in U$ such that $v \in \mathsf{Path}(u)$ and for any $w \in \mathsf{Path}(u)$, $\mathsf{val}(\mathsf{sub}(w)) < \mathsf{val}(\mathsf{sub}(v))$ if and only if $w \in \sup(U)$. We denote the set of U-candidates as $\mathsf{Can}(U)$.

Lemma 11. *For every* $1 \le i < n$, $\mathsf{sup}(u_{i+1}) \in \mathsf{Can}(\{u_1, \ldots, u_i\})$.

Proof. We prove this lemma by induction on i. By the definition of the TT T, u_2 is the subordinate of a node $v \in \mathsf{Path}(u_1)$. Furthermore, $\mathsf{val}(u_2)$ is the smallest number in the team of $\mathsf{val}(u_1)$. Hence, $\sup(u_2) \in \mathsf{Can}(\{u_1\})$.

Suppose the statement holds for $i \ge 1$. Let $x = \mathsf{sup}(u_{i+1})$. Our goal is to show that $x \in \mathsf{Can}(\{u_1, \ldots, u_i\})$. For any node $v \in \mathsf{Path}(x)$, we have $\mathsf{val}(v) < \mathsf{val}(u_{i+1})$, as otherwise, v would not be on the same principal path as x. Hence, the head of $\mathsf{Path}(x)$ is u_j for some $1 \le j \le i$.

Let w be a node in $\mathsf{Path}(x)$. Suppose $\mathsf{val}(\mathsf{sub}(w)) < \mathsf{val}(\mathsf{sub}(x))$. Since $\mathsf{val}(\mathsf{sub}(x)) = \mathsf{val}(u_{i+1})$, $\mathsf{Team}(\mathsf{Path}(\mathsf{sub}(w)))$ would contain a number that has a smaller value than u_{i+1}. Therefore, w must be $\sup(u_j)$ for some $1 \leq j \leq i$. This means that $w \in \mathsf{sup}(\{u_1, \ldots, u_i\})$. Conversely, suppose $w \in \mathsf{sup}(\{u_1, \ldots, u_i\})$. Then, by choice of u_{i+1} we have $\mathsf{val}(\mathsf{sub}(w)) < \mathsf{val}(u_{i+1}) = \mathsf{val}(\mathsf{sub}(x))$. Thus, $x \in \mathsf{Can}(\{u_1, \ldots, u_i\})$. □

The next lemma implies the correctness of Algorithm 13.

Lemma 12. *For any $i \geq 1$, the ith call to $\mathsf{next}(L)$ returns the node u_i if $i \leq n$, and nil otherwise.*

Proof. We prove the lemma by induction on the number of calls to $\mathsf{next}(L)$. It is clear that in the first call to $\mathsf{next}(L)$, the algorithm returns the node u_1, which is the origin of the principal path that contains the root of the tree of L.

Consider the second call to $\mathsf{next}(L)$. If L contains only one number, then $\mathsf{Team}(u_1)$ does not exist and the priority queue Q is empty at line 5. If L contains more than one element, then $\mathsf{Team}(u_1)$ is defined. At line 5, Q will store the element $x = \mathsf{up}(a)$, where $a = \mathsf{next}(\mathsf{Team}(u_1))$ is the node with the smallest value in $\mathsf{Team}(u_1)$. By definition, $\mathsf{Can}(\{u_1\}) = \{x\}$.

For the inductive step, suppose we are calling $\mathsf{next}(L)$ for the $(i+1)$th time, where $i \geq 1$. We make the following inductive assumption: When the algorithm reaches line 5,

(I1) if L contains no more than i elements, then the priority queue Q is empty;

(I2) if L contains at least $i+1$ elements, then the priority queue Q contains exactly those nodes in $\mathsf{Can}(\{u_1, \ldots, u_i\})$.

If L contains no more than i elements, then by (I1) the algorithm returns nil and Q remains empty. Suppose instead that L contains at least $i+1$ elements. By (I2), when the algorithm reaches line 5, the priority queue Q contains exxactly those nodes in $\mathsf{Can}(\{u_1, \ldots, u_i\})$. Let x be the least element in Q. By Lemma 11, x is $\sup(u_{i+1})$. Thus, the algorithm would locate and return the node u_{i+1}.

We then need to verify that the $\mathsf{next}(L)$ operation preserves the inductive invariants (I1) and (I2). It is clear that (I1) holds at line of the $(i+2)$th call to $\mathsf{next}(L)$.

To verify (I2), let S, S' denote the sets of nodes stored in the priority queue Q at line 5 in the $(i+1)$th and the $(i+2)$th call to $\mathsf{next}(L)$ respectively. Let b be the leaf that has the next smallest value in $\mathsf{Team}(x)$ after x. After we finish the $(i+1)$th call to $\mathsf{next}(L)$, Q would store the set $S \setminus \{x\} \cup \{\mathsf{up}(b)\}$. In the $(i+2)$th call to $\mathsf{next}(L)$,

before reaching line 5, the algorithm would add the node $\mathsf{up}(a)$ to Q, where a has the least value in $\mathsf{Team}(u_{i+1})$. Therefore, we have

$$S' = S \setminus \{x\} \cup \{\mathsf{up}(a), \mathsf{up}(b)\} = \mathsf{Can}(\{u_1, \ldots, u_i, u_{i+1}\})$$

Hence, (I2) is preserved. □

As described above, the psort operation on the LTT of L amounts to creating an iterator of L and calling $\mathsf{next}(L)$ the required number of times. By Lemma 12, the operation outputs the desired numbers in monotonically increasing order.

Time complexity of psort

We now analyze the time complexity of the $\mathsf{psort}(T, k)$ operation. Throughout, let T be the LTT of the list $L = (a_1, a_2, \ldots, a_n)$.

Suppose t is a layer-i team in Γ_L. Any call to the $\mathsf{next}(t)$ operation may in turn trigger a sequence of calls to next on teams in lower layers. The algorithm maintains a priority queue for every team in which an iterator is created.

Each call to $\mathsf{next}(t)$ performs a fixed number of priority queue operations (such as insert and delete_min), at most two calls to the $\mathsf{next}(t')$ operation on some layer-$(i+1)$ team t', and a fixed number of other elementary operations. Among these operations, the first call to $\mathsf{next}(t')$ occurs immediately after the $(i+1)$-iterator of t' is created. This call to $\mathsf{next}(t')$ simply involves a pointer lookup, requiring constant time. Furthermore, by Lemma 4, the number of leaves of the team tree of t' is at most $\log_\varphi(m)$, where m is the number of elements in t.

Suppose we perform k calls to $\mathsf{next}(t)$ where $k \geq 1$. Note that for any team t' in layer $j > i$, the algorithm would make at most $k - 1$ calls to $\mathsf{next}(t')$. With every call to $\mathsf{next}(t')$, the number of elements stored in the priority queue increases by at most 2. Thus the number of elements stored in any priority queue is at most than $2k$. Therefore, the time for inserting an element to or deleting the minimum element from the priority queue is $O(\log(k))$.

Summing up the above costs over all k calls, the operations perform $O(k)$ number of priority queue operations, $k - 1$ calls to next on trees in a layer down, and other operations that take a total of $O(k)$ time. We use $\mu(k, m)$ to denote the time taken by k calls to $\mathsf{next}(t)$ where the team tree of t has m leaves. Thus, there is a constant $d > 0$ such that:

$$\mu(k, m) \leq \begin{cases} dk \log k + \mu(k - 1, \log_\varphi(m)) & \text{if } m > 1; \\ d & \text{otherwise.} \end{cases} \tag{4.1}$$

Lemma 13. *The* psort(T, k) *operation is* $O(\log^*_\varphi(n) \cdot k \cdot \log(k))$.

Proof. The psort(T, k) operation makes k calls to the next(L) operation. Thus, the running time of psort(T, k) is $\mu(k, n)$. By Equation 4.1 we get

$$\mu(k,n) \leq d\cdot k\cdot\log(k)+d\cdot(k-1)\cdot\log(k)+d\cdot(k-2)\cdot\log(k)+\cdots+d\cdot(k-s+1)\cdot\log(k)+d$$

where s is the layer number of Γ_L. By Lemma 10, $s \leq \log^*_\varphi(n)$. Thus, psort(T, k) is $O(\log^*_\varphi(n) \cdot k \cdot \log(k))$. □

4.2.3 changeval, link and cut

We now describe the changeval, link and cut operations on LTTs. Throughout, let T be the LTT representation of the list $L = (a_1, a_2, \ldots, a_n)$, and let T' be the LTT representation of the list $L' = (a'_1, a'_2, \ldots, a'_m)$. We assume that the elements of L are pairwise disjoint from elements in L'. Based on Section 1.1, we will use u as a parameter to represent a reference to the node which is the ith leaf in T in the changeval and cut operations.

We define each of the changeval(T, u, n), link(T, T') and cut(T, u) operations by induction on the maximum layer number of T, T'. If an LTT consists of only one layer, it contains only one node. Therefore, the cut and changeval operations performed on such an LTT are trivial. To perform link(T, T') where both T, T' consist only of one layer, we create a new node v and set left(v), right(v) to be the roots of T, T' respectively in the layer-0 tree, and then create a layer-1 tree with a single node whose value is the larger of the values of the children of v.

In subsequent sections, we describe and analyze the changeval(T, u, n), link(T, T') and cut(T, u) operations where T, T' have more than one layer. The inductive hypothesis assumes that correct implementations of link and cut on LTTs with fewer layers than T, T' exist.

The fix_up operation for LTTs

The changeval, link and cut operations can cause the LTT data structure to become broken, as the changes to one layer must be reflected correctly on all lower layers. Thus, we must apply other procedures to restore the LTT structure after such changes. We refer to this operation as fix_up(u), where u is a node in an LTT tree in some layer. Throughout this chapter, when we refer to fix_up, we mean the operation defined here, as opposed to the same operation defined in Chapter 3.

Intuitively, the fix_up(u) operation maintains the LTT structure on the path from u to the root of the tree once a change has occured on a child. It walks the path from u

to the root, and performs the following procedures in each step:

1. Separate u from its principal path from below, so that both $\mathsf{left}(u)$ and $\mathsf{right}(u)$ are detached from $\mathsf{Path}(u)$.
2. Link the smaller of $\mathsf{left}(u), \mathsf{right}(u)$ with $\mathsf{Path}(u)$.
3. Set $\mathsf{val}(u)$ as the minimum of the values of its children.
4. If u is not the root, repeat this process with $\mathsf{p}(u)$ instead of u.

To separate and link the principal path mentioned above, we use the cut and link operations on the team trees of the corresponding principal paths. Note that the above operaton may change the subordinate of u. This requires us to change the value of $\mathsf{down}(u)$ in the team tree $\mathsf{Team}(u)$, which can be performed by calling $\mathsf{changeval}(\mathsf{Team}(u), \mathsf{down}(u), \max\{\mathsf{left}(u), \mathsf{right}(u)\})$ recursively. Note that the team trees used as arguments to the link and cut operations, as well as the recursive call to changeval, have strictly fewer layers than the tree containing u. Thus, by the inductive hypothesis, these operations have been defined.

For an exact description, see Algorithm 14.

Algorithm 14 $\mathsf{fix\_up}(u)$

```
1: if u ≠ nil then
2:     if val(left(u)) < val(right(u)) then
3:         (z, z') ← left(u), right(u)
4:     else
5:         (z, z') ← right(u), left(u)
6:     val(u) ← val(z)
7:     (T1, T2) ← cut(Team(u), down(u))
8:     changeval(T1, down(u), val(z')) ▷ Change the value of down(u) in the layer below
9:     x ← link(T1, Team(z))
10:    fix_up(p(u))
```

We now analyze the correctness of the $\mathsf{fix\_up}(u)$ operation. More specifically, let v be an internal node in an LTT. We use the following invariants:

(J1) $\mathsf{val}(v) = \mathsf{min}\{\mathsf{val}(\mathsf{left}(v)), \mathsf{val}(\mathsf{right}(v))\}$

(J2) $\mathsf{val}(\mathsf{down}(v)) = \mathsf{val}(\mathsf{sub}(v))$

(J3) If v has a child v' that is an internal node, and $\mathsf{val}(v) = \mathsf{val}(v')$, then $\mathsf{down}(v), \mathsf{down}(v')$ belong to the same team tree $\mathsf{Team}(v)$, and $\mathsf{down}(v)$ is to the left of $\mathsf{down}(v')$ in $\mathsf{Team}(v)$.

Intuitively, these three invariants state that the LTT structure is maintained. Indeed, (J1) states that the value of v is assigned according to the TT property, (J2) states that $\mathsf{down}(v)$ has the correct value, and (J3) states that the team tree of $\mathsf{down}(v)$ is correctly maintained.

To demonstrate the correctness of fix_up, we make the following definition:

Definition 19. Let v be a node in the LTT of L. The *parent-down closure* of v is the minimal set $\mathsf{pd}(v)$ of nodes in the LTT that contains v, and for any node $w \in \mathsf{pd}(v)$,

1. $\mathsf{p}(w) \in \mathsf{pd}(v)$ if w is not the root of a tree; and
2. $\mathsf{down}(w) \in \mathsf{pd}(v)$ if w is not a leaf in a tree.

We observe that $\mathsf{fix\_up}(u)$ can only update the values, as well as link and separate team trees, for nodes in the set $\mathsf{pd}(u)$. Hence, inuitively, $\mathsf{pd}(u)$ denotes the 'region of operation' in the LTT containing u of $\mathsf{fix\_up}(u)$.

For the next lemma, recall that we assume by the inductive hypothesis that a correct implementation of link and cut can be called on LTTs with fewer layers than the LTT of u.

Lemma 14. *After running* $\mathsf{fix\_up}(u)$*, (J1) – (J3) hold for every node* $v \in \mathsf{pd}(u)$*.*

Proof. The proof proceeds by induction on the number of layers in the LTT T of L. The statement is clear for T with a single layer (which consists of only one node). Now suppose that T contains $m > 1$ layers. Take a node $v \in \mathsf{pd}(u)$ that is in layer 0 of T. Then, v will be the argument of some call to some recursive call of fix_up. During that call, (J1) holds after running Line 6, (J2) holds after running Line 8, and (J3) holds after running Line 9 for v.

Suppose that (J1) – (J3) hold for all nodes in $\mathsf{pd}(u)$ on some layer i, and $v \in pd(u)$ is an internal node in a layer-$(i+1)$ tree of the LTT. Then, by definition of $\mathsf{pd}(u)$, there is some leaf w in the subtree rooted at v such that $w = \mathsf{down}(w')$ for some $w' \in \mathsf{pd}(u)$. Let w be the rightmost leaf with this propery. The algorithm must have made a call to $\mathsf{changeval}(T_1, w, \mathsf{val}(z'))$ during its execution. In this call to changeval, the recursive calls visit v, and make (J1) – (J3) hold for v using Lines 6, 8 and 9 respectively. □

Tree rotations in an LTT

Before we can describe the link and cut operations for LTTs, we describe the tree rotation operation for LTTs, which is an important subroutine. Throughout this chapter, any reference to rotate_left and rotate_right refer to the augmented variants of these operations described here.

We describe the left rotation rotate_left(u); the right rotation operation is symmetric. To perform rotate_left(u), we first separate both u and $\mathsf{p}(u)$ from the rest of their principal paths from above and below. We then perform the left rotation of u as if for a normal binary tree. Lastly, the restore the principal path of $\mathsf{p}(u)$ by caling the fix_up($\mathsf{p}(u)$) operation. This will fix the principal paths we separated in this operation and preserve the structure of the LTT. For an exact description, see Algorithm 15.

Algorithm 15 rotate_left(u)

1: $y \leftarrow \mathsf{p}(u)$
2: **if** y is not the root **then**
3: cut($\mathsf{Team}(\mathsf{p}(y)), \mathsf{down}(\mathsf{p}(y))$) ▷ Separate y from above.
4: cut($\mathsf{Team}(y), \mathsf{down}(y)$) ▷ Separate y from below.
5: cut($\mathsf{Team}(u), \mathsf{down}(u)$) ▷ Separate u from below.
6: Perform a normal left tree rotation on u.
7: fix_up(y)

To demonstrate the correctness of the augmented tree rotation operation, we give the following lemma. It is a consequence of Lemma 14, and the proof is straightforward.

Lemma 15. *Let y be the parent of u. After running* rotate_left(u)*, (J1) – (J3) hold for every node $v \in \mathsf{pd}(y)$.*

The operations

We now describe the changeval, link and cut operations on LTTs in detail. Throughout this section, we use T to refer to an LTT of the list $L = (a_1, a_2, \ldots, a_n)$ and T' to refer to an LTT of the list $L' = (a'_1, a'_2, \ldots, a'_n)$. Additionally, based on Section 1.1, the description will use u as a parameter to represent the reference we have to the ith leaf of T.

To perform changeval(T, u, n), we simply change $\mathsf{val}(u)$ to n, then call fix_up($\mathsf{p}(u)$) to restore the LTT data structure. The correctness of this operation follows from Lemma 14.

We now describe the link(T, T') operation. For simplicity, we only describe the case when the height of the layer-0 tree of T is greater than the layer-0 tree of T'; the other case is symmetric. We first find a node u on the rightmost path in the layer-0 tree of T, such that $T(u)$ has the same high as the layer-0 tree of T'. We then create a new node v, making it a child of $\mathsf{p}(u)$, and set $T(u)$ as v's left subtree and the layer-0 tree of T' as v's right subtree. We then fix the principal paths by calling fix_up(v). This operation may leave the resulting layer-0 tree unbalanced; hence, we walk the path from v to the root, and if we find a node y on this path such that its subtrees are unbalanced, we call rotate_left to correct this.

This finishes the description of link for LTTs. Note that in this operation, all recursive calls to link and cut are made on LTTs with fewer layers than T, and are thus defined by the inductive hypothesis. For an exact description, see Algorithm 16.

Algorithm 16 $\mathsf{link}(T, T')$

1: $(r_1, r_2) \leftarrow$ the roots of the layer-0 trees of T, T' respectively
2: Follow right references from r_1 to find u such that $T(u)$ and r_2 have the same height
3: Create a new node v and the corresponding node $\mathsf{down}(v)$ in the layer below
4: $\mathsf{p}(v) \leftarrow \mathsf{p}(u)$
5: $(\mathsf{left}(v), \mathsf{right}(v)) \leftarrow u, r_2$
6: $\mathsf{fix\_up}(v)$
7: Follow p references from v; if we find an unbalanced node, call a rotation operation to correct it.

We perform the $\mathsf{cut}(\ell, u)$ operation in a similar way to Algorithm 12. The operation first calls changeval on u to assign it a value smaller than all numbers in T (we call it $-\infty$ for convenience). In this way, all nodes on the path from u to the root form a principal path. The operation then walks the path from u to the root, joining all subtrees to its left into a new tree and all subtrees to its right into another new tree. Finally it restores the value of u and joins u to the first new tree. We perform all the joining of trees using the link operation; see Algorithm 17 for an exact description.

Algorithm 17 $\mathsf{cut}(T, u)$

1: $a \leftarrow \mathsf{val}(u)$
2: $\mathsf{changeval}(\ell, u, -\infty)$
3: $(x, y) \leftarrow \mathsf{p}(u), u$
4: Create two empty TTs T_1, T_2
5: **while** $x \neq \mathsf{nil}$ **do**
6: **if** $y = \mathsf{left}(x)$ **then**
7: $T_2 \leftarrow \mathsf{link}(T_2, T(\mathsf{right}(x)))$
8: **else**
9: $T_1 \leftarrow \mathsf{link}(T(\mathsf{left}(x)), T_1)$
10: $y \leftarrow x$; $x \leftarrow \mathsf{p}(x)$
11: $\mathsf{val}(u) \leftarrow a$; $\mathsf{link}(T_1, u)$ ▷ Link T_1 with the restored u

We now examine the correctness of the changeval, link and cut operations. We make the following lemma:

Lemma 16. *Let T be the LTT of $L = (a_1, a_2, \ldots, a_n)$, T' be the LTT of $L' = (a'_1, a'_2, \ldots, a'_m)$, and u be a leaf node in the layer-0 tree of T. (J1) – (J3) hold after performing any of* $\mathsf{changeval}(T, u, x)$, $\mathsf{link}(T, T')$ *or* $\mathsf{cut}(T, u)$.

Proof. For the changeval(T, u, x) operation, as u is a leaf in the layer-0 tree of T, by Lemma 14, (J1) – (J3) still hold for every node in the LTT after the operation completes. For link(T, T'), by Line 6 in Algorithm 16, (J1) – (J3) are preserved for every node. If the operation must perform a rotation, by Lemma 15, (J1) – (J3) still hold for every node, thus making link correct. For cut(T, u), (J1) – (J3) hold by the correctness of changeval and link. □

Time complexity

We now analyze the time complexity of the update operations. For any list L with n elements, we define $s_i(n)$ as the maximum number of elements of a layer-i team in the LTT of L. It is clear that $s_0(n) = n$. By Lemma 10, for all $n > 0$ we have

$$\begin{aligned} &s_{\log^*_\varphi(n)}(n) = 1, \text{ and} \\ &\forall i \geq 0 : \ s_{i+1}(n) \leq \log_\varphi(s_i(n)) \end{aligned} \tag{4.2}$$

For convenience, we set $s_i(n) = 1$ for all $i > \log^*_\varphi(n)$.

We will express the complexity of the update operations using the variables $s_i(n)$.

Lemma 17. *For any $i \geq 0$, there is a constant $n_0 > 0$ such that for all $n > n_0$ we have*

$$\prod_{j \geq i+1} s_j(n) \leq s_i(n)$$

Proof. As $s_j(n) = 1$ for all $n > 0$ and $j \geq \log^*_\varphi(n)$, the statement is clear for $i \geq \log^*_\varphi(n) - 1$. The proof proceeds by induction on i. Fix $0 < i < \log^*_\varphi(n)$ and suppose there is n_0 such that the statement holds for all $n > n_0$. Then for all $n \geq n_0$ we have

$$\begin{aligned} \prod_{j \geq i} s_j(n) &= s_i(n) \cdot \prod_{j \geq i} s_j(n) \\ &\leq s_i^2(n) && \text{(by the ind. hyp.)} \\ &\leq \log^2_\varphi(s_{i-1}(n)) && \text{(by (4.2))} \end{aligned}$$

Take n' such that

$$\log^2_\varphi(s_{i-1}(n')) \leq s_{i-1}(n').$$

Then for all $n \geq \max\{n', n_0\}$

$$\prod_{j \geq i} s_j(n) \leq \log^2_\varphi(s_{i-1}(n)) \leq s_i(n).$$

□

Recall that the fix_up(u) operation calls itself recursively several times. We analyze the running time of each call separately. Without loss of generality, we assume in the next lemma that the list L contains no fewer elements than L'.

Lemma 18. *Let n be the number of elements in the list L, and let u be a node in the LTT T of L. Let T' be the LTT of L'. The following hold for the update operations:*

(a) Each call to fix_up(u) *runs in time $O(s_2^2(n))$.*

(b) The fix_up(u) *and* changeval(T, u, x) *operations run in time $O(s_1(n) \cdot s_2^2(n))$.*

(c) The link(T, T') *operation runs in time $O(d(\ell, \ell') \cdot s_2^2(n))$ where $d(T, T')$ is the height difference between the layer-0 trees of T and T'.*

(d) The cut(T, u) *operation runs in time $O(s_1(n) \cdot s_2^2(n))$.*

Proof. We prove the lemma by induction on the layer number of T. The statements are clear if T consists of a single layer. For the case when T has more than one layer, we prove each statement as follows:

(a) We use Time($n, 0$) to denote the maximal running time of each call to fix_up(u). It is clear that the number of calls is bounded by the length of the path from u to the root, which is at most $s_1(n)$. Hence the total running time of fix_up(u) is $s_1(n)$Time($n, 0$).

Note also that each recursive call of fix_up(u) may make a recursive call to fix_up on a team in the layer below, and this recursive call may trigger further recursive calls to fix_up on lower layers of the LTT. Thus for $0 \leq i \leq \log_\varphi^*(n)$ and any layer-i team t, we define Time(n, i) as the maximal running time of a recursive call in a recursive call fix_up(v) that is made within fix_up(u) on a layer below the layer containing u. Since the recursive call fix_up(v) consists of at most $s_{i+1}(n)$ recursive calls, the total running time of fix_up(v) is at most $s_{i+1}(n)$Time(n, i).

To prove (a), we prove by induction on i that Time(n, i) is $O(s_{i+2}^2(n))$ for all $0 \leq i \leq \log_\varphi^*(n)$.

It is clear that $\text{Time}\left(n, \log_\varphi^*(n)\right) = 1$. Now suppose t is a layer-i team where $i < \log_\varphi^*(n)$. Each recursive call made as part of fix_up(v) makes one call to cut and one call to link. Both of these subroutine calls are made on teams in the next layer down, which by the inductive hypothesis takes $O(s_{i+2}(n)s_{i+3}^2(n))$. The iteration also recursively calls fix_up on a team in the next layer down. By the above argument this takes $s_{i+2}(n)$Time($n, i+1$). Lastly, each call also performs a

fixed number of other elementary operations. Therefore we obtain the following expression for $0 \leq i < \log^*_\varphi(n)$:

$$\mathsf{Time}(n,i) \leq c_1 s_{i+2}(n) s^2_{i+3}(n) + s_{i+2}(n)\mathsf{Time}(n,i+1) + c_2$$

where $c_1, c_2 > 0$ are constants. For convenience we drop the parameter n in the above expression to get

$$\mathsf{Time}(i) \leq c_1 s_{i+2} s^2_{i+3} + s_{i+2}\mathsf{Time}(i+1) + c_2 \tag{4.3}$$

Applying telescoping on (4.3), we obtain

$$\begin{aligned}
\mathsf{Time}(0) &\leq c_1 s_2 s_3^2 + c_1 s_2 s_3 s_4^2 + \cdots + c_1 s_2 \cdots s_{\log^*_\varphi(n)} s_{\log^*_\varphi(n)+1} s^2_{\log^*_\varphi(n)+2} \\
&\quad + c_2 + c_2 s_2 + \cdots + c_2 s_2 \ldots s_{\log^*_\varphi(n)} \\
&\leq c_1 \sum_{i=1}^{\log^*_\varphi(n)} \left(s_{i+2} \prod_{j=2}^{i+2} s_j \right) + c_2 \sum_{i=2}^{\log^*_\varphi(n)} \prod_{j=2}^{i} s_j \\
&\leq c_1 \sum_{i=1}^{\log^*_\varphi(n)} s_2 s_3^2 s_{i+2} + c_2 \log^*_\varphi(n) s_2 s_3^2 \qquad \text{(by Lemma 17)} \\
&\leq c_1 \log^*_\varphi(n) s_2 s_3^3 + c_2 \log^*_\varphi(n) s_2 s_3^3
\end{aligned}$$

Hence the running time of a single call to $\mathsf{fix\_up}(u)$ is $O(\log^*_\varphi(n) s_2(n) s_3^3(n))$. By Lemma 9, $\log^*_\varphi(n)$ is $O(s_3(n))$ and thus $\mathsf{Time}(n,0)$ is $O(s_2(n)s_3^4(n))$, which by (4.2), is $O(s_2^2(n))$.

(b) This statement follows directly from (a) and the fact that the maximum number of iterations performed by the $\mathsf{fix\_up}(u)$ operation is $s_1(n)$.

(c) For the $\mathsf{link}(T, T')$ operation we use the following inductive hypothesis: Any calls to cut and $\mathsf{fix\_up}$ on teams at layer-1 of the LTT T takes time $c \cdot s_2(n) \cdot s_3^2(n)$ for some constant $c > 0$.

Let T_1 and T_2 be the top layer trees of T and T' respectively and $d(T_1, T_2)$ be the height difference between T_1 and T_2. Recall that the $\mathsf{link}(T, T')$ operation finds a node u on the rightmost path of T_1 such that $T(u)$ and T_2 have the same height and links $T(u)$ and T_2 to a new node below this node. Hence the $\mathsf{fix\_up}(v)$ operation in $\mathsf{link}(T, T')$ consists of $d(T_1, T_2)$ iterations. By (a), this call to $\mathsf{fix\_up}(v)$ takes time $c_1 \cdot d(\ell, \ell') \cdot s_2^2(n)$, where c_1 is a constant.

The $\mathsf{link}(T, T')$ operation also potentially makes a call to $\mathsf{rotate\_left}(y)$ which consists of three calls to cut and one call to $\mathsf{fix\_up}$ on teams at a lower layer. By the

inductive hypothesis, these subroutine calls to takes time $c_2 \cdot s_2(n) \cdot s_3^2(n)$ for some constant $c_2 > c$. The $\mathsf{link}(T, T')$ operation also performs $O(d(T_1, T_2))$ many other elementary operations. Therefore the running time of $\mathsf{link}(T, T')$ is at most

$$c_1 \cdot d(T_1, T_2) \cdot s_2^2(n) + c_2 \cdot s_2(n) \cdot s_3^2(n) + c_3 \cdot d(T_1, T_2).$$

Note that we may pick c to be bigger than $c_1 + c_3$ and therefore the above expression is at most

$$(c_1 + c_3) \cdot d(T_1, T_2) \cdot s_2^2(n) + c_2 \cdot s_2(n) \cdot s_3^2(n)$$

which is at most $c \cdot d(T_1, T_2) \cdot s_2^2(n)$ when n is sufficiently large. Therefore the running time for $\mathsf{link}(T, T')$ is $O(d(T_1, T_2) \cdot s_2^2(n))$.

(d) Let T_0 be the top-layer tree of T. The cut operation first makes a call to $\mathsf{changeval}(T_0, u, -\infty)$, which by (b) takes time $O\left(s_1(n) \cdot s_2^2(n)\right)$. It then walks the path from u to the root. Let $P = \{u_0, u_1, \ldots, u_m\}$ be the path in T_0 from $u_0 = u$ to the root of T_0 where $u_{i+1} = p(u_i)$ for all $0 \leq i < m$. It is clear that $m \leq s_1(n)$ and thus the traversal itself takes time $O(s_1(n))$.

By Alg. 17, the $\mathsf{cut}(T, u)$ operation separates T_0 into a collection of trees

$$\widehat{T}_1, \widehat{T}_2, \ldots, \widehat{T}_k$$

where each $\widehat{T}_i$ is either the left or the right subtree of u_i. As T_0 is balanced, one could easily prove by induction on i that

$$h\left(\widehat{T}_i\right) \leq 2i - 1.$$

The $\mathsf{cut}(T, u)$ operation then iteratively joins the trees $\widehat{T}_1, \widehat{T}_2, \ldots, \widehat{T}_k$ to form two trees T_1 and T_2. Let n_i be the number of leaves in the tree $\widehat{T}_i$. By (c) the total running time of the sequence of link operations performed is at most

$$\begin{aligned}
& 2 \sum_{i \geq 1}^{m-1} \left(h\left(\widehat{T}_{i+1}\right) - h\left(\widehat{T}_i\right)\right) \cdot s_2^2\left(n_{i+1}\right) \\
\leq\ & 2 \sum_{i \geq 1}^{m-1} \left(h\left(\widehat{T}_{i+1}\right) - h\left(\widehat{T}_i\right)\right) \cdot s_2^2(n) \\
\leq\ & 2 \left(h\left(\widehat{T}_m\right) - h\left(\widehat{T}_1\right)\right) \cdot s_2^2(n) \\
\leq\ & 2 s_1(n) s_2^2(n).
\end{aligned}$$

Therefore the overall running time of the $\mathsf{cut}(\ell, u)$ operation is $O\left(s_1(n) \cdot s_2^2(n)\right)$.

□

We make the following lemma, whose correctness follows from Lemmas 18 and 10.

Lemma 19. *Let T be the LTT of a list L such that $|L| = n$, and let T' be the LTT of a list L' such that $|L'| = m$. Also, let a be the absolute value of the difference in height between T and T''s layer-0 trees. Then,* $\mathsf{changeval}(T, u, x)$ *and* $\mathsf{cut}(T, u)$ *are both* $O(\log^2(n))$*, and* $\mathsf{link}(T, T')$ *is* $O(\log^2(a))$.

4.3 Conclusion

In this chapter, we described an improved data structure for solving the dynamic partial sorting problem. We have demonstrated that is has better performance with respect to the psort operation, having an asymptotic time complexity that is almost independent of the number of numbers stored in the data structure. The LTT also permits the changeval, link and cut operations in $O(\log^2(n))$ time complexity (where n is the number of numbers stored), which is still asymptotically sublinear. In Chapter 5, we will attempt to demonstrate the improved psort performance experimentally, as well as compare the performance of the TT and the LTT with respect to modification operations.

Chapter 5

Experimental Results

In this chapter, we aim to test the performance ascribed to the TT and the LTT in Chapters 3 and 4. We describe the experimental setup and the data used for testing, and then present and discuss the results, both individually and by comparison. Lastly, we determine which data structure solves the dynamic partial sorting problem best in practice.

5.1 Experimental Setup

Each data structure was implemented in Lua. The experiments were conducted using LuaJIT 2.0.4, patched for Lua 5.2 support. The hardware used to perform the experiments was a computer running Parabola GNU/Linux-libre 4.0.4-gnu-1 with 64-bit support. The testing machine has an Intel Core i3-4160 CPU, clocked at 3.6 GHz, with 8GB of RAM and 16GB of swap space. The data for each test was generated as follows: for a test of size n, a list of values of the form $(1, 2, \ldots, n)$ was created, and then shuffled randomly. To ensure sensible values and to avoid bias from LuaJIT, each experiment was repeated 10 times, and their results averaged.

To test psort, two experiments were conducted: the first generated a single tree of size 50,000, and then performed psort on it with k ranging from 1 to 50,000; the second experiment generated trees ranging in size from 200 to 50,000, and then performed psort on each tree, with k fixed at 200. This was designed to show how psort scales if either the size of the tree, or k, were kept constant.

To test changeval, trees whose size n ranged from 1 to 50,000 were generated. For each tree, a value $1 \leq x \leq n + 1$ was reserved (meaning it was not put into the tree). Then, a random leaf was selected and its value changed to x.

To test link, trees were generated whose size difference ranged from 0 (meaning the trees had identical heights) to 49,999 (meaning that one tree was a leaf). Then, the

two trees were linked.

To test cut, trees whose size n ranged from 1 to 50,000 were generated. For each tree, a random leaf was chosen, and the tree was then cut on that leaf.

5.2 TT

5.2.1 psort

The results of the experiments can be seen in Figures 5.1 and 5.2.

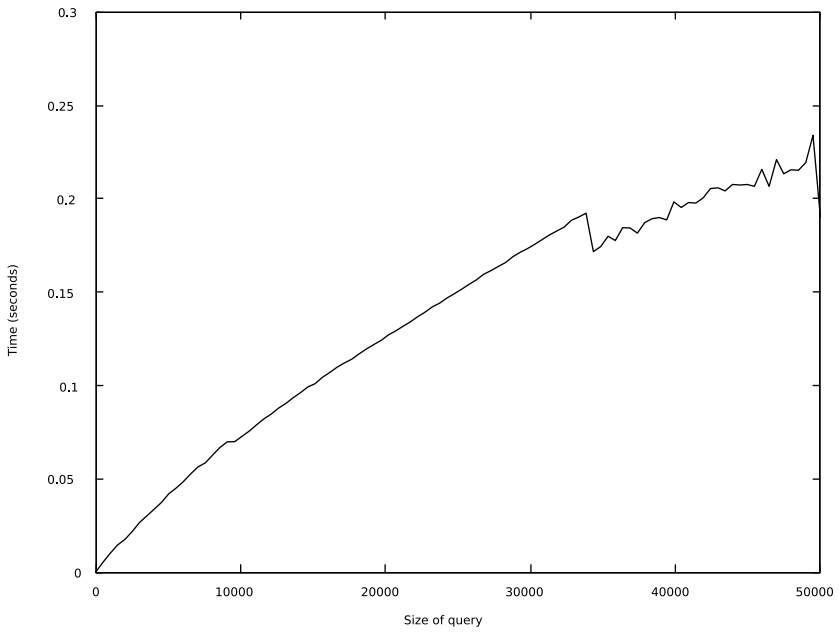

Figure 5.1: Performance of psort for the TT with a variable-sized query and a fixed-size tree.

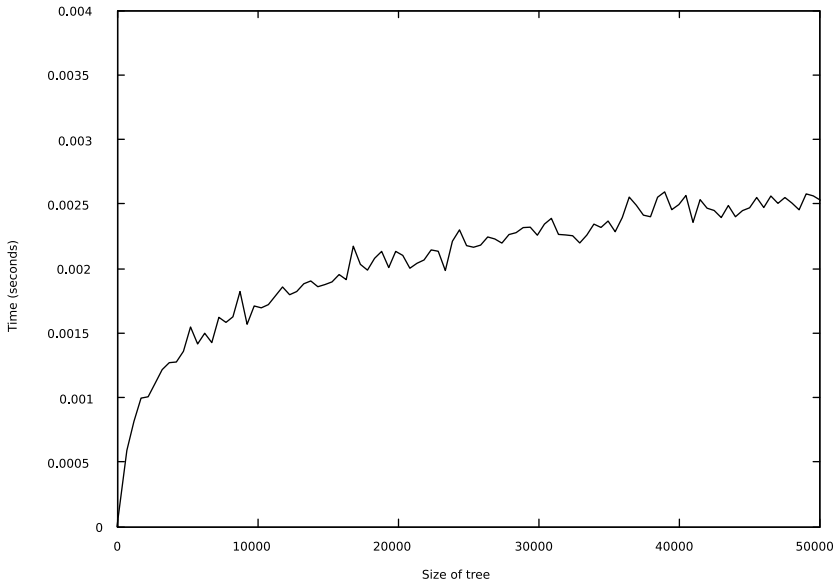

Figure 5.2: Performance of psort for the TT with a fixed-size query and a variable-size tree.

These results clearly demonstrate that the performance of psort in practice is in-line with the asymptotic time complexity as analyzed in Chapter 3.

5.2.2 changeval

The results of the experiment can be seen in Figure 5.3.

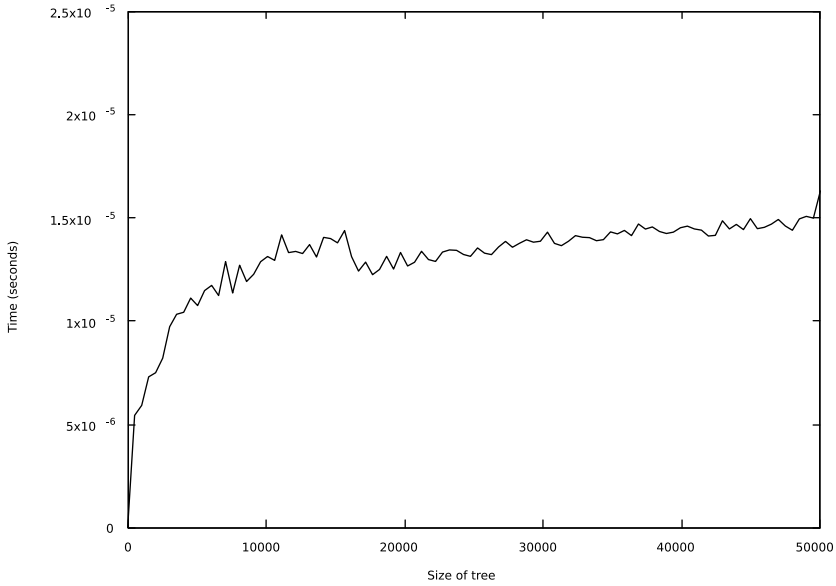

Figure 5.3: Performance of changeval for the TT.

These results demonstrate that the performance of changeval is indeed logarithmic with respect to the size of the tree being operated on if the position of the leaf being

modified initially is randomly-selected.

5.2.3 link

The results of the experiment can be seen in Figure 5.4.

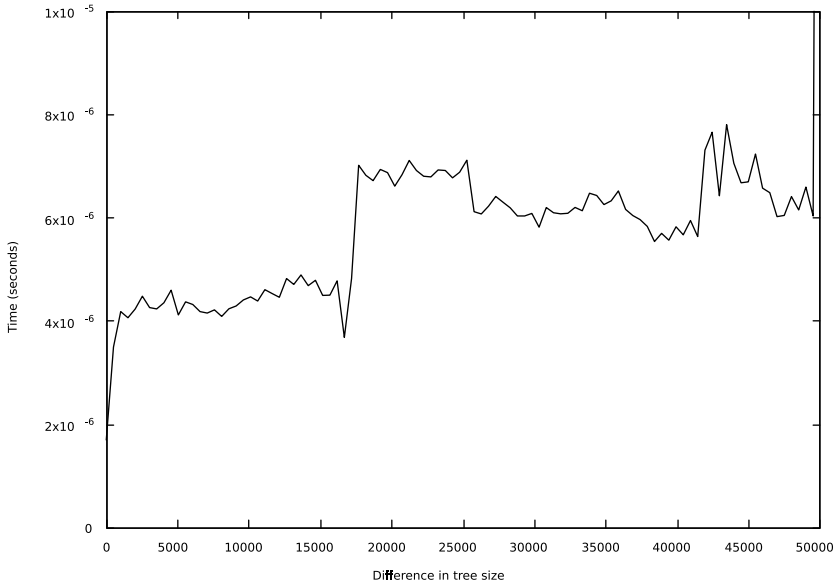

Figure 5.4: Performance of link for the TT.

Although the time consumed seems to vary somewhat, especially as the difference in tree sizes became larger, these results clearly indicate that the performance of link is indeed logarithmic with respect to the difference in size of the two trees being linked. This is consistent with our analysis in Subsection 3.2.4.

5.2.4 cut

The results of the experiment can be seen in Figure 5.5.

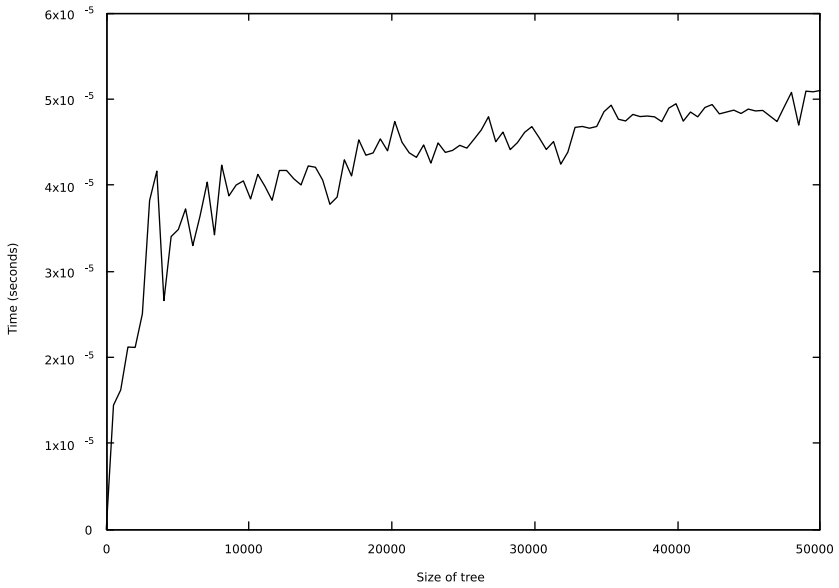

Figure 5.5: Performance of cut for the TT.

These results are consistent with our analysis in Subsection 3.2.5, indicating that the performance of cut is logarithmically-bounded by the size of the tree being separated.

5.3 LTT

5.3.1 psort

The results of the experiments can be seen in Figures 5.6 and 5.7.

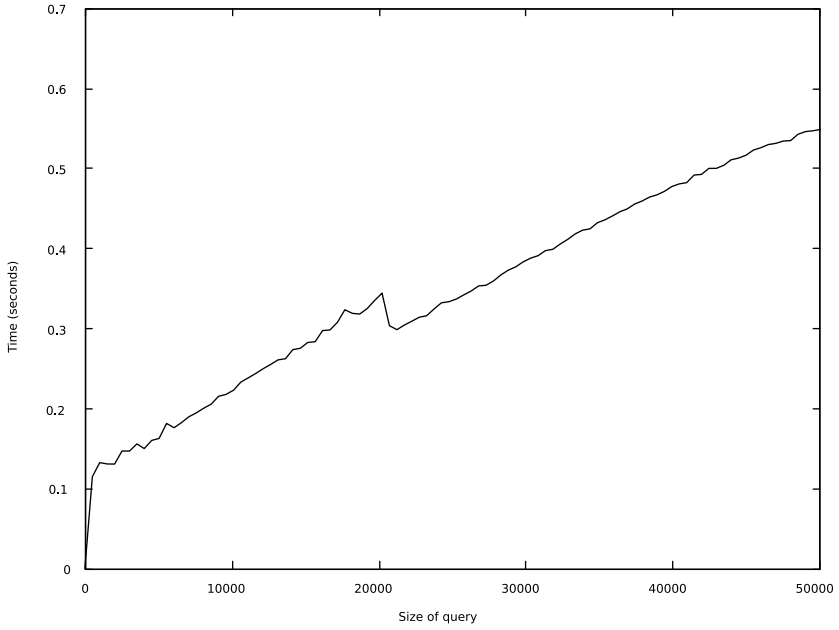

Figure 5.6: Performance of psort for the LTT with a variable-sized query and a fixed-size tree.

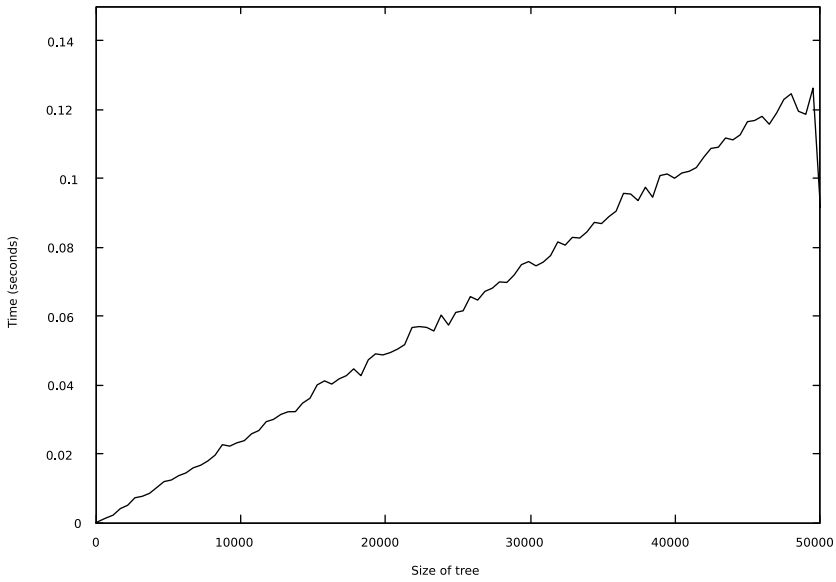

Figure 5.7: Performance of psort for the LTT with a fixed-size query and a variable-size tree.

These results are very different from what was expected after the analysis in Chapter 4. In both cases, the LTT appears to exhibit linear-time behaviour, which is quite surprising. We discuss the possible reasons for this in Section 5.9.

5.3.2 changeval

The results of the experiments can be seen in Figure 5.8.

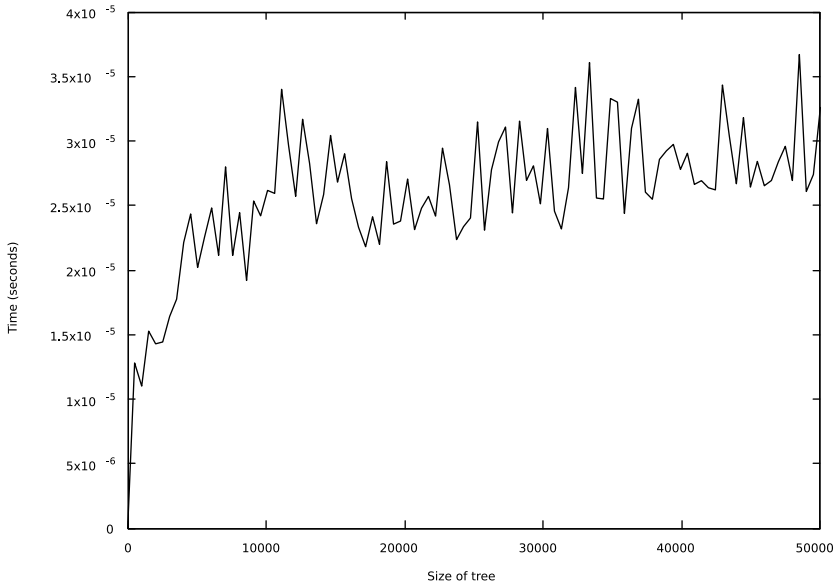

Figure 5.8: Performance of changeval for the LTT.

These results are consistent with the time complexity analysis for this operation

given by Lemma 18. The higher degree of variance than exhibited by the equivalent TT operation can be attributed to the larger variance in the amount of work required to repair lower layers, especially due to the invocation of link and cut operations, which even on TTs exhibited significant performance variance.

5.3.3 link

The results of the experiments can be seen in Figure 5.9.

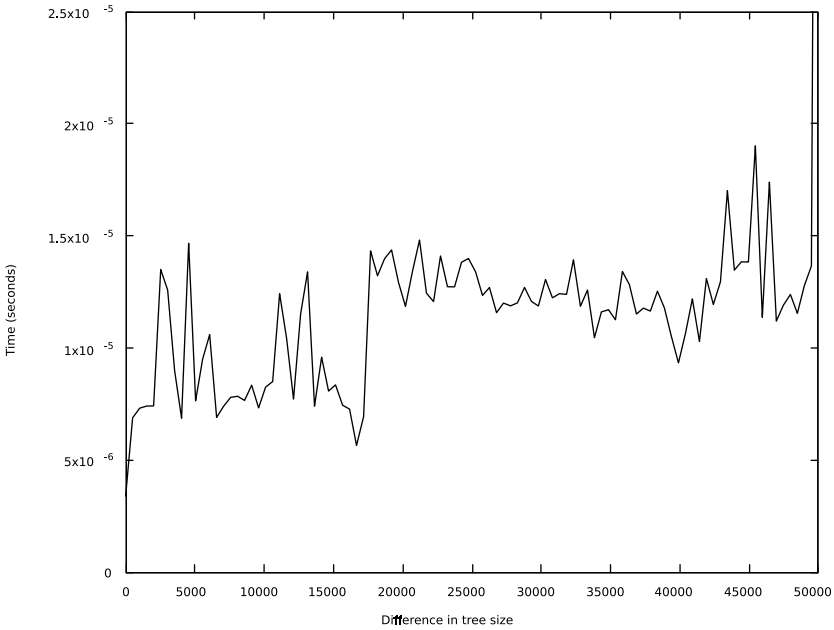

Figure 5.9: Performance of link for the LTT.

As with changeval, these results are within expected norms, despite exhibiting a higher degree of variance than the TT link.

5.3.4 cut

The results of the experiments can be seen in Figure 5.10.

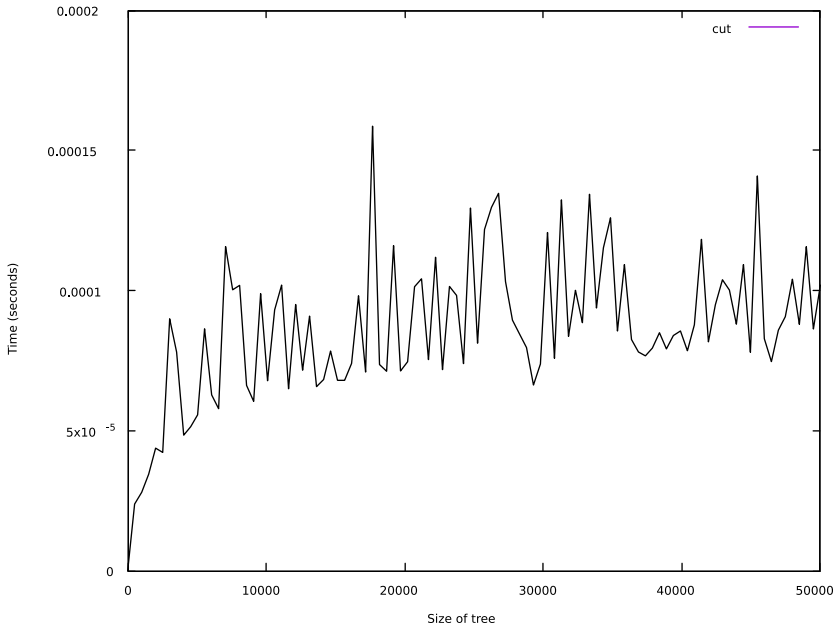

Figure 5.10: Performance of cut for the LTT.

This is consistent with the results for the LTT changeval and link operations, and is thus what was expected.

5.4 Comparison

We now indicate the performance difference between the TT and the LTT by graphing each of the experiments for both of the data structures on the same set of axes.

5.5 psort

The comparison of the two experiments on variable-sized queries and fixed-size trees is depicted on Figure 5.11; the comparison of the two experiments on fixed-size queries and variable-size trees is depicted on Figure 5.12.

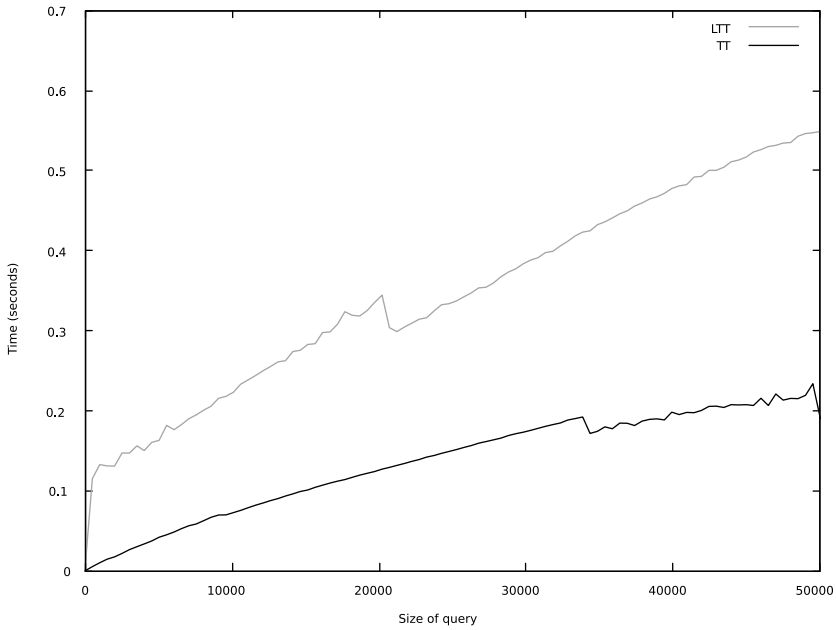

Figure 5.11: Comparison of the performance of psort for the LTT and TT with a variable-sized query and a fixed-size tree.

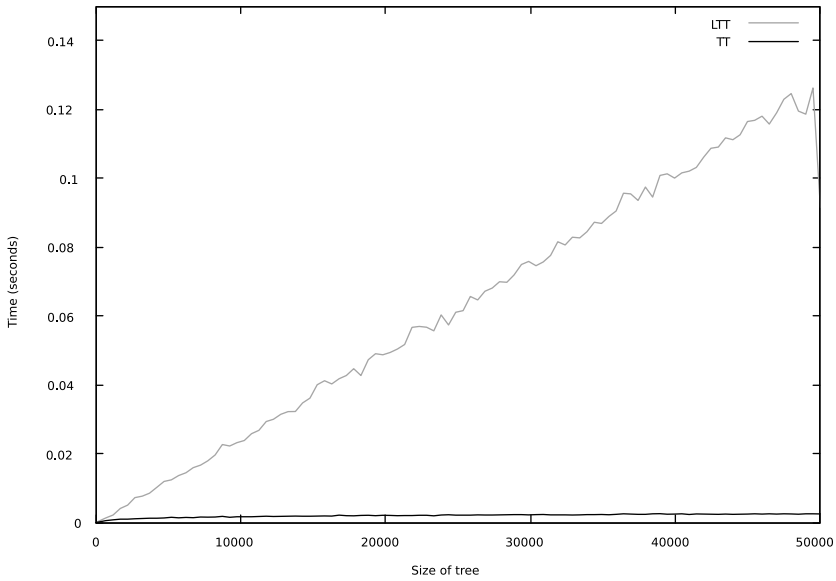

Figure 5.12: Comparison of the performance of psort for the LTT and TT with a fixed-size query and a variable-size tree.

This shows that, despite the theoretical analysis, the performance of the LTT is significantly worse than that of the TT, both in terms of actual values and in terms of growth. We consider the possible reasons for this in Section 5.9.

5.6 changeval

The performance of the LTT and TT versions of changeval is depicted on Figure 5.13.

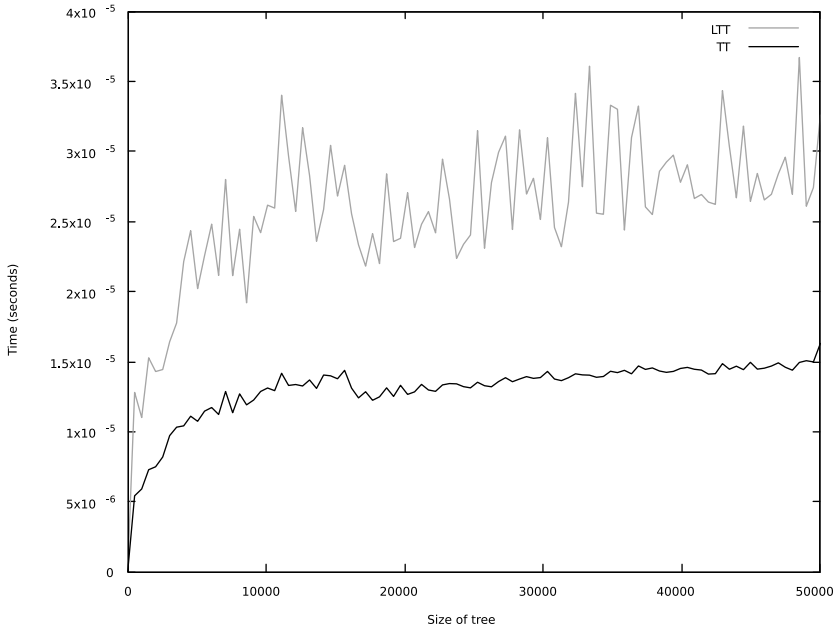

Figure 5.13: Comparison of the performance of changeval for the LTT and TT.

The differences in performance between the LTT and TT versions of changeval is within expected parameters based on the analysis given in Chapters 3 and 4.

5.7 link

The performance of the LTT and TT versions of link is depicted on Figure 5.14.

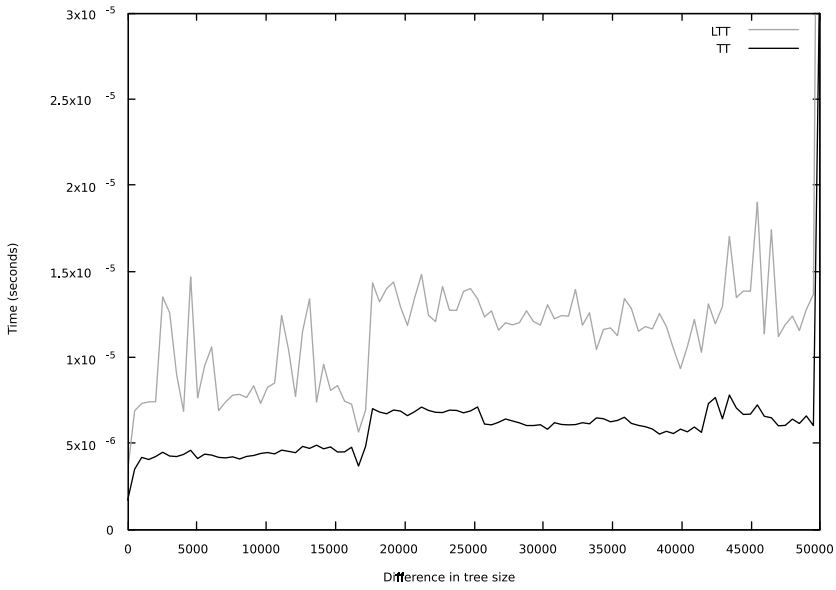

Figure 5.14: Comparison of the performance of link for the LTT and TT.

The differences in performance between the LTT and TT versions of link is within expected parameters based on the analysis given in Chapters 3 and 4.

5.8 cut

The performance of the LTT and TT versions of cut is depicted on Figure 5.15.

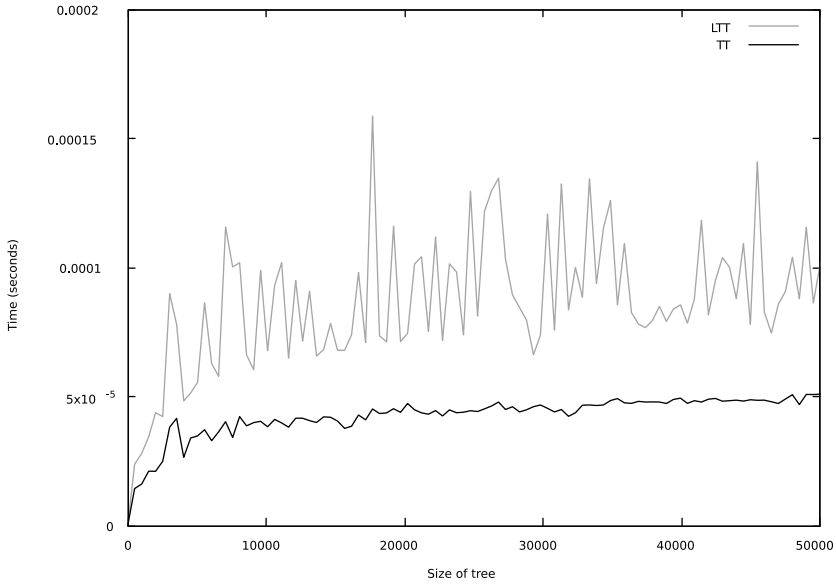

Figure 5.15: Comparison of the performance of cut for the LTT and TT.

The differences in performance between the LTT and TT versions of cut is within expected parameters based on the analysis given in Chapters 3 and 4.

5.9 Analysis and conclusion

Overall, the performance of the TT was exactly as expected. The LTT, however, performed unusually with respect to its query operation, supposedly its main benefit over the TT. In both cases, its behaviour appears to be roughly linear, which is most surprising in the second test, as the expected performance of psort for an LTT when k is kept constant should be roughly constant. Additionally, overall, the LTT *psort* performed worse than its TT counterpart by a significant margin.

Overall, the poor performance of the LTT psort operation overall can be explained by several means. Firstly, the theoretical analysis of the LTT psort operation's running time ignores considerable constant factors, which together likely contribute to a much larger running time than the analysis itself would suggest. Secondly, significant amounts of state must be kept in order to maintain the iterators used in the LTT psort operation, which is a likely contributor to the linear-seeming running time of both operations. Lastly, the relative performance of $O(\log^*_\varphi(n) \cdot k \cdot \log(k))$ versus $O(k \cdot \log(n))$ only becomes apparent when n is extremely large, while k is extremely small relative n. Given that the test data set n to at most 50,000, it is possible that no real performance gains can be seen at that scale. However, tests at the scale required to confirm this need very large amounts of memory, which make such testing impractical.

The linear performance of the LTT's psort when k is kept constant is more difficult to explain. Although the additional constant factors being ignored by the analysis would impact the time required, it should not cause a constant-time operation to become linear. Additionally, the state required by the iterators on a fixed-size query should only expand as rapidly as the layers do (and indeed, is the reason for the $\log^*_\varphi(n)$ factor in the analysis), which also fails to explain this performance. The sudden drop-off in time required towards the end suggests that once again, the performance may improve with larger test data, but the test sizes required to validate this claim are too large to be practical.

Overall, the experiments demonstrate that, for the given size of data, the TT has better practical performance characteristics than the LTT. In particular, the psort operation on TTs appears to perform much better in practice for TTs than LTTs in the range of data being tested. Otherwise, the performance characteristics appear consistent with the theoretical analysis provided by Chapters 3 and 4.

Chapter 6

Conclusion and Further Work

6.1 Thesis conclusion

This thesis aimed to solve the dynamic partial sorting problem by presenting a solution that would permit all of the operations described in Section 1.1. To this end, we initially discussed some naive solutions and explained why their performance characteristics were not desirable. To help provide an alternative, we discussed dynamic trees and why they are more suitable as a basis for a solution to this problem. We then presented two solutions to the dynamic partial sorting problem: the TT and the LTT.We described both structures and how they would support the operations required by the dynamic partial sorting problem, and analyzed the correctness and time complexity of these operations. Lastly, we experimentally tested both structures, and investigated the outcomes, both individually and in comparison to each other.

We can conclude that the performance characteristics of the TT as a solution to the dynamic partial sorting problem are superior to those of the LTT in practical terms. While we speculate that larger data sets may allow for better performance for the LTT, the experimental data could not be used to confirm this. While the theoretical analysis appears to indicate that we can implement a structure that disregards (or nearly disregards) the size of the data set when performing its queries, it appears that this does not necessarily hold up in practice. Thus, we believe that the TT is a better practical solution to the dynamic partial sorting problem on the data sets we measured.

6.2 Further work

We propose three possible directions for future work with the dynamic partial sorting problem. The first is the concept of optimizing data structures for dynamic partial sorting with a fixed size (or range of sizes) of queries; the second is the application of

parallelism to the algorithms used to implement the dynamic partial sorting operations; and the third is the optimization of the resulting structure for external memory use and persistence[11].

6.2.1 Optimized queries

This idea is similar in principle to *optimized binary search trees*, as presented in Cormen et al[6]. These data structures are designed to optimize query operations based on a history of queries, rearranging the structure to permit the most frequent queries to be performed the fastest. In the case of the dynamic partial sorting problem, we would seek to perform optimizations by determining an optimal query size, and then creating a data structure that could perform this query efficiently.

We can perform these optimizations either statically or dynamically. In the case of optimizing for query size, in the static case, we have a table of queries and the probability that a query will have that length (similarly to the optimal BST). We then determine an expected query length, and make a structure to perform queries of that length optimally. In the dynamic case, the structure keeps track of query probabilities, and dynamically rebuilds itself when the expected query length changes.

These are both suitable directions of research, as there is likely to be a fixed query size that is often needed (such as a database which is used to dynamically emit a 'Top 10' list, for example). The dynamic case is more complex, and more suited to this problem, due to the dynamic nature of the data.

6.2.2 Parallelism

Due to the size of the data that would be typical in an application of the dynamic partial sorting problem (such as databases performing an SQL query with LIMIT and ORDER restrictions), parallel implementations of the psort operation, as well as the update operations, are an interesting possible dimension for future work. Given that research on parallel implementations of dynamic tree operations has yielded interesting results[19], it may be possible to achieve significant speed-up in psort queries by using parallel techniques. Parallel construction and modification of data structures intended to solve the dynamic partial sorting problem are also worth considering, as the structures are supposed to support mutation as well as searching.

6.2.3 External memory use and persistence

As the dynamic partial sorting problem is meant to solved for large data structures (such as databases), another important class of optimizations to consider is the ability

for the data to be used with external memory. These would likely require a different implementation, as they are aimed at minimizing the amount of data present in working memory at any one time. If this is to be employed for databases, these optimizations are essential, as in-memory data storage quickly reaches a limit that can only be surpassed efficiently by good use of external storage.

Bibliography

[1] Adelson-Velsky, G., Landis, E.: An algorithm for the organization of information. Proceedings of the USSR Academy of Sciences, 146. 263–266. 1962.

[2] Anderson, A.: Optimal bounds on the dictionary problem. Proc. Symposium on Optimal Algorithms, Springer-Verlag. 106–114. 1989.

[3] Andersson, A., Fagerberg, R., Larsen, K.: Balanced Binary Search Trees. In: Mehta, D., Sahni, S., eds: Handbook of Data Structures and Applications. 182–205. 2002.

[4] Bayer, Rudolf.: Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica, 1 (4). 290–306. 1972

[5] Bordim, J., Nakano, K., Shen, H.: Sorting on Single-Channel Wireless Sensor Networks. In: Hsu, F., Ibarra, H., Saldaña, R., eds, Proc. of the International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'02). 133–138. 2002.

[6] Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, the MIT Press. 2002

[7] Duch, A., Jiménez, R., Martínez, C.: Selection by rank in k-dimensional binary search trees. Random Structures and Algorithms, appeared online 2012

[8] Floyd, R., Rivest, R.: Expected time bounds for selection. In: Communications of the ACM 18(3). 165–172. 1975.

[9] Gross, J., Yellen, J., Zhang, P.: Handbook of Graph Theory, Second Edition. CRC Press. 116. 2013.

[10] Guibas, L., Sedgewick, R.: A dichromatic framework for balanced trees. Proceedings of the 19th Annual Symposium on Foundations

[11] Haim, K.: Persistent Data Structures. In: Mehta, D., Sahni, S., eds: Handbook of Data Structures and Applications. 182–205. 2002. of Computer Science. IEEE. 1978.

[12] Hoare, C.: Find (Algorithm 65). Comm. ACM, 4:321–322. 1961.

[13] Hoare, C.: Quicksort. Computer Journal, 5:10–15, 1962.

[14] Huang, H., Tsai, T., Quickselect and the Dickman function. Combinatorics, Probability and Computing 11 (4). 353–371. 2000.

[15] Jiménez, R., Martínez, C.: Interval Sorting. Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP 2010), Part I. 238–248. 2010.

[16] Knuth, D.: The Art of Computer Programming, Sorting and Searching, Volume 3. 141–142. 1998.

[17] Kuba, M.: On Quickselect, partial sorting and Multiple Quickselect. Information Processing Letters 99 (5). 181–186. 2006.

[18] Navarro, G., Paredes, R.: On Sorting, Heaps and Minimum Spanning Trees. Algorithmica 57 (4). 585–620. 2010.

[19] Park, H., Park, K., Parallel algorithms for red-black trees. Theoretical Computer Science, 262 (1-2). 415–435. 2001.

[20] Sahni, S.: Leftist Trees. In: Mehta, D., Sahni, S., eds: Handbook of Data Structures and Applications. 101. 2002.

[21] Sleator, D., Tarjan, R.: Self-Adjusting Binary Search Trees. Jounral of the ACM, 32 (3). 625–686. 1985.

[22] Tarjan, R.: Data Structures and Network Algorithms. Bell Laboratories. 1983.