New Combinatorial Properties and Algorithms for AVL Trees

Mahdi Amani

Under Supervision of
Linda Pagli, Anna Bernasconi

This dissertation is submitted to the University of Pisa
For the degree of Doctor of Philosophy

October 2015
Abstract

In this thesis, new properties of AVL trees and a new partitioning of binary search trees named core partitioning scheme are discussed, this scheme is applied to three binary search trees namely AVL trees, weight-balanced trees, and plain binary search trees.

We introduce the core partitioning scheme, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees called cores. Using this technique we achieve the same theoretical efficiency of modern cache-oblivious data structures by using classic data structures such as weight-balanced trees or height balanced trees (e.g. AVL trees). We preserve the original topology and algorithms of the given balanced search tree using a simple post-processing with guaranteed performance to completely rebuild the changed cores (possibly all of them) after each update. Using our core partitioning scheme, we simultaneously achieve good memory allocation, space-efficient representation, and cache obliviousness. We also apply this scheme to arbitrary binary search trees which can be unbalanced and we produce a new data structure, called Cache-Oblivious General Balanced Tree (COG-tree).

Using our scheme, searching a key requires $O(\log_B n)$ block transfers and $O(\log n)$ comparisons in the external memory model and the cache-oblivious model. These complexities are theoretically efficient. Interestingly, the core partition for weight-balanced trees and COG-tree can be maintained with amortized $O(\log_B n)$ block transfers per update whereas maintaining the core partition for AVL trees requires more than a poly-logarithmic amortized cost.

Studying the properties of these trees also lead us to some other new properties of AVL trees and trees with bounded degree, namely, we present gap terminology and we prove Tarjan et al.’s conjecture on the number of rotations in a sequence of deletions and insertions.

Keywords: AVL trees, Weight-balanced tree, External memory model, Cache-oblivious model, Core partitioning scheme, COG-tree, Hat partitioning scheme, Gap terminology, AVL rotation.
Dedication

“Thesis topics come and go, but the musical styling of journey are forever.”

I may not be a religious man, but it is hard to escape the feeling that, underneath all the problems you face in your routine life, there is something of the divine in the solutions you find. I saw God’s hand showing me the way when I was stuck and lost.

This dissertation is dedicated to him and to my wife who shared her patience and support through the hard life of a young researcher.

To the compassionate one and to Eli
Acknowledgments

This work could not have been achieved without the help, support, and encouragement of many people. My heartfelt thanks is owed to many people who made this journey possible. My special thanks and appreciation to Prof. Linda Pagli, Prof. Anna Bernasconi, and Prof. Roberto Grossi not only for their help and encouragement throughout my studies, but also for their compassion and empathetic understanding and touching my life during these years that I had the great honor to work with them. I am grateful as well to Prof. Pierpaolo Degano for coordinating and all the support that he has given me throughout my academic life in Pisa.

I thank the members of my dissertation committee for their contribution and their good-natured support, I also thank the other members of department of computer science, university of Pisa, for all good memories, help and all the things they taught me. I also thank Robert Tarjan for his time, support and encouragement to prove their conjecture on the number of rotations in AVL trees.

Last but not least, I sincerely and wholeheartedly express my gratitude to all my family members specially my parents and my wonderful wife for their patience and support.
Contents

1 Introduction 1
  1.1 Memory Hierarchy and Memory Models .......................... 1
  1.2 Main Results .............................................. 2
  1.3 Other Results on Properties of AVL Trees ..................... 5
  1.4 Thesis Organization and Overview ............................. 7

2 Background 9
  2.1 Trees and Binary Search Trees .............................. 9
    2.1.1 Binary Search Trees ................................... 10
    2.1.2 Tree Representation .................................... 20
    2.1.3 Tree Traversal .......................................... 20
  2.2 External Memory & Cache-Oblivious Memory Models ............ 21
    2.2.1 External Memory Model ................................. 22
    2.2.2 Cache-Oblivious Memory Model .......................... 23
  2.3 Data Structures for External Memory & Cache-Oblivious Memory Models 24
  2.4 Exhaustive Generation of Trees with Bounded Degree ........ 28
    2.4.1 Generation Preliminaries ............................... 29
    2.4.2 Trees with Bounded Degree ............................. 34
  2.5 Summary .................................................. 35

3 Core Partitioning Scheme 37
  3.1 Core Partitioning Preliminaries .............................. 37
  3.2 Core Partitioning Scheme ................................... 39
    3.2.1 Core Partitioning ....................................... 39
    3.2.2 Maintaining the Core Partition .......................... 42
  3.3 Weight-Balanced Trees ..................................... 44
    3.3.1 Cores in Weight-Balanced Trees ......................... 44
    3.3.2 Amortized Analysis for Repartitioning .................. 46
    3.3.3 External-Memory Search Trees .......................... 47
    3.3.4 Cache-Oblivious Search Trees .......................... 48
    3.3.5 Space-Efficient Search Trees ........................... 49
  3.4 AVL Trees ................................................ 50
    3.4.1 Cores in AVL Trees ..................................... 51
    3.4.2 Amortized Analysis for Repartitioning .................. 54
  3.5 Summary .................................................. 56

4 Core Partitioning Directly on Plain Binary Trees 57
  4.1 Preliminaries and Notation .................................. 58
  4.2 Definition of COG-Tree ..................................... 60
  4.3 Maintaining a COG-Tree ..................................... 63
    4.3.1 Deletions .............................................. 63
    4.3.2 Insertions ............................................. 64
CONTENTS

4.3.3 Amortized Analysis ........................................... 66
4.3.4 External-Memory Search Trees ............................. 68
4.3.5 Cache-Oblivious Search Trees .............................. 68
4.4 Summary ......................................................... 69

5 Hat Partitioning Scheme ........................................ 71
  5.1 Random Model ................................................ 72
  5.2 Hat Partitioning Scheme ..................................... 73
  5.3 AVL Trees as External-Memory Search Trees .............. 74
  5.4 Maintaining the Hat Partitioning Scheme .................. 75
  5.5 Average Time Complexity ................................... 76
  5.5.1 Improved Hat Partitioning Scheme ...................... 85
  5.6 Summary ....................................................... 85

6 Other Properties of AVL Trees .................................. 87
  6.1 GAP Terminology .............................................. 87
  6.1.1 General Properties of Gaps ............................... 88
  6.1.2 Gaps in Insertions and Deletions ......................... 91
  6.2 Rank Terminology ............................................ 96
  6.3 Amortized Rotation Cost of AVL Trees ..................... 99
  6.3.1 Expensive AVL Trees ................................... 100
  6.4 Summary ....................................................... 102

7 Generation of Trees with Bounded Degree .................... 103
  7.1 The Encoding Schema ....................................... 104
  7.2 The Generation Algorithm .................................. 106
  7.3 Ranking and Unranking Algorithms ......................... 110
  7.4 Summary ....................................................... 116

8 Conclusions and Discussion .................................... 117

Bibliography .................................................... 121
## List of Figures

2.1 An embedding of an ordered rooted tree in the plane on a set of 11 nodes (4 internal nodes and 7 leaves) with root labeled by \( x \). .................................................. 10

2.2 An example of AVL trees. .................................................. 14

2.3 Right rotation at node \( x \). Triangles denote subtrees. The inverse operation is a left rotation at \( y \). .................................................. 14

2.4 \( BB[\alpha] \)-tree for \( \alpha = 2/7 \) while it is not for \( \alpha = 1/3 \). ................................. 16

2.5 An example of Randomized Search Tree, in this picture the numbers are priorities and the alphabets are the keys. ................................. 18

2.6 An inorder traversal algorithm for binary trees. .......................................................... 21

2.7 A postorder traversal algorithm for binary trees. .......................................................... 21

2.8 A preorder traversal algorithm for binary trees. .......................................................... 21

2.9 The external memory model. .......................................................... 23

2.10 Left: \( C_3H_8 \) propane, middle and right: \( C_4H_{10} \) butanes. ................................. 35

2.11 A \( T^\Delta \) tree with 12 nodes. .......................................................... 35

3.1 Decomposition of a binary search tree into its cores. .......................................................... 40

3.2 Left: when \( w \) is higher than \( v \), so \( u = w \). Right: when \( w \) is lower than \( v \), so \( u = v \). ................................. 43

4.1 The path \( v_1, v_2, \ldots, v_{2h_{\min}(v)+1} \), from the new inserted leaf \( f \) to \( v \), and node \( v_{2h_{\min}(v)-1} \) whose first \( h_{\min}(v) - 1 \) levels are full. .......................................................... 67

5.1 a) Insertion of new node \( f \) with absorption. b) Insertion of new node \( f \) with height increase. .......................................................... 77

5.2 Critical path before and after insertion of \( f \) in single rotation (top) and double rotation (bottom). .......................................................... 78

5.3 hat rescan analysis while inserting a new node \( f \). .......................................................... 84

5.4 Improved hat partitioning scheme. .......................................................... 86

6.1 Gaps before and after absorption(a) and height increase(b). .......................................................... 92

6.2 Gaps before and after single rotation. .......................................................... 93

6.3 Gaps before and after double rotation. .......................................................... 93

6.4 A sequence of \( n \) insertion into an empty tree with \( \text{"whole-tree-epochs"} \) and \( \text{"height increases"} \). .......................................................... 95

6.5 Gaps before and after deletions in case of no children of deleted node(a) or one child(b). .......................................................... 96

6.6 Rebalancing cases after insertion. Numbers next to edges are rank differences. Rank differences of unmarked edges do not change. The promote step may repeat. All cases have mirror images. .......................................................... 96

6.7 Rebalancing cases after deletion. Numbers next to edges are rank differences. Rank differences of unmarked edges do not change. Each case except the first single rotation case may repeat. All cases have mirror images. .......................................................... 98
6.8 Recursive definition of $E$. Numbers on edges are rank differences. The two trees shown are in $E$ if $A$ and $C$ are in $E$, $B$ is an AVL tree, and $A$, $B$, and $C$ have the same rank. .................................................. 100
6.9 Deletion and insertion of the shallow leaf in a type-$L$ tree of rank $2$. .............. 101
6.10 Deletion and insertion of the shallow leaf in a type-$L$ tree of rank $k + 2$ .......... 101

7.1 An example of a tree $T \in T_{n}^{\Delta}$ (for $\Delta \geq 4$). Its codeword is “slstrmsrtmsnr”. .. 105
7.2 a) The first $T_{n}^{\Delta}$ tree in A-order. b) The last $T_{n}^{\Delta}$ tree in A-order. .......... 105
7.3 $T^{\Delta}$ trees encoded by $C = sx$ and $C = slx_{1}mx_{2}\ldots mx_{j-1}rx_{j}$. ............. 106
7.4 Algorithm for generating the successor codeword for $T_{n}^{\Delta}$ trees in A-order. ...... 108
7.5 Algorithm for updating the children. .................................................. 108
7.6 Algorithm for updating the neighbors. .................................................. 109
7.7 $T_{n}^{\Delta}$ tree whose first subtree has exactly $m$ nodes and its root has maximum degree $d$. 111
7.8 Ranking algorithm for $T_{n}^{\Delta}$ trees. .............................................. 112
7.9 Unranking algorithm for $T_{n}^{\Delta}$ trees. .............................................. 114
Chapter 1

Introduction

Trees are one of the most important basic and simple data structures for organizing information in computer science, and have found many applications such as database [76, 57], pattern recognition [57], decision table programming [57], analysis of algorithms [57], string matching [57], switching theory [103], image processing [92, 100], and even in the theoretical design of circuits required for VLSI [103]. Trees are also widely used for showing the organization of real world data such family trees, taxonomies, and modeling of the connections between neurons of the brain in computational neuroscience [22, 23].

Many balanced search trees have been designed for their usage in main memory, with optimal asymptotical complexity in terms of CPU time and number of performed comparisons, such as AVL trees [1], red-black trees [15], weight-balanced trees [74], and 2-3 trees [46], just to name the pioneering ones. Unfortunately these data structures perform poorly when cache performance is taken into account and large data sets are stored in external memory.

1.1 Memory Hierarchy and Memory Models

In this thesis, we adopt external memory model [2] and cache-oblivious model [41, 82] to evaluate I/O complexities. The memory hierarchies of modern computers are composed of several levels of memories, that starting from the caches, have increasing access time and capacity. The design of data structures and algorithms must now take care of this situation and try to efficiently amortize the cost of memory accesses by transferring blocks of contiguous data from one level to another. The CPU have access to a relatively small but fast pool of solid-state storage space, the main memory; it could also communicate with other, slower but potentially larger storage spaces, the
external memory. The memory hierarchies of modern computers are composed of several levels of memories start from the caches. Caches have very small access time and capacity comparing to main memory and external memory. From cache to main memory, then to external memory, access time and capacity increase significantly.

In external memory model [2], the computer has access to a large external memory in which all of the data resides. This memory is divided into memory blocks each containing $B$ words, and $B$ is known. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are free; they take no time at all and that is because of the fact that external memory is so much slower than random access memory [71]. We assume that each external memory access (called an I/O operation or just I/O) transmits one page of $B$ elements.

Traditional databases are designed to reduce the number of disk accesses, since accessing data on the disk is orders of magnitude more expensive than accessing data in main memory. With data sets becoming resident in main memory, the new performance bottleneck is the latency in accessing data from the main memory [51]. Therefore, we also adopt the cache-oblivious model [41, 82] to evaluate the I/O complexity, here called cache complexity. The cache-oblivious model is a simple and elegant model introduced in [41, 82] which allows to consider only a two-level hierarchy, but proves results for a hierarchy composed of an unknown number of levels. In this model, memory has blocks of size $B$, where $B$ is an unknown parameter and a cache-oblivious algorithm is completely unaware of the value of $B$ used by the underlying system.

1.2 Main Results

We propose a general method to store the nodes of balanced search trees and obtain provably good space-efficient external-memory/cache-oblivious data structures. It hinges on the decomposition of a balanced search tree into a set of disjoint cores: a core is a complete balanced binary tree that appears as a portion of the balanced tree. A core of height $h$ has $2^h - 1$ nodes when the height of a node is the number of nodes on the longest simple downward path from that node to a leaf [57]. Our method is not invasive, as it does not change the original algorithms. It just requires a post-processing procedure after each update to maintain the cores. The nodes of a core are stored in a chunk of consecutive memory cells. Hence, the core partition adds a memory layout for the nodes of a balanced tree but does not interfere with the original algorithms for the tree.
The benefits of our technique are that we can reuse the vast knowledge on balanced search trees to provide a repertoire of space-efficient external-memory and cache-oblivious data structures that are competitive with modern data structures that are purposely designed for these models. This technique opens a number of possibilities that are known for modern search data structures but unknown for several previous balanced trees.

- I/O efficiency and cache-obliviousness can be achieved.
- Dynamic memory management can be handled by allocating a common contiguous memory chunk for all the keys of each core.
- The pointers requires a total of $O(n)$ bits of storage rather than $\Omega(n \log n)$ for a tree of $n$ nodes.
- Search operation can be performed in $O(\log_B n)$ I/Os.
- Efficient dynamization of static data structures can be also obtained.

We emphasize that the above features just require the original algorithms described for the given balanced tree, thus offering simultaneously many features that have been introduced later on different search trees. What we add is the maintenance of our structure for the nodes, and the algorithmic challenge is how to maintain it efficiently. When performing the updates, we proceed as usual, except that we perform a post-processing: we take the topmost core that should be changed because of the update, and recompute the partition from it in a greedy fashion.

We call this technique, \textit{core partitioning scheme}, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees (cores). Using this technique we achieve the same theoretical efficiency of modern cache-oblivious data structures by using the classic structures such as weight-balanced trees or height balanced trees such as AVL trees or original binary search trees. We show that these “classic data structures” can be dynamized as efficient as very modern ones in external memory model and cache-oblivious memory model. Using our core partitioning scheme, we show how to store balanced trees such as weight-balanced trees and height-balanced trees (AVLs), so that they simultaneously achieve good memory allocation, space-efficient representation, and cache obliviousness. When performing updates, we show that weight-balanced trees can be maintained with a logarithmic cost, while AVL trees require super poly-logarithmic cost by a lower bound on the subtree size of the rotated nodes. In these two case studies, searching a key using core partitioning scheme requires $O(\log_B n)$ block transfers and $O(\log n)$ comparisons.
in the external memory model. The cost becomes $O(\log_B n + \log(\frac{\log(B+1)}{\log(\log n + 1)}))$ block transfers and $O(\log n)$ comparisons in the cache-oblivious model. Later we show that also in cache-oblivious model searching a key can be decreased to $O(\log_B n)$ block transfers. These complexities are theoretically efficient.

The notion of core partition introduced above shows how to obtain cache-efficient versions of classical balanced binary search trees such as AVL trees and weight-balanced trees. A natural question is whether the core partition can be applied also to arbitrary binary search trees which can be unbalanced. We give a positive answer to this question by presenting a data structure, called Cache-Oblivious General Balanced Tree (COG-tree).

A binary tree is typically kept balanced by storing at each node some information on the structure of the tree and checking at each update that some constraints on the structure of the tree are maintained. This information must be dynamically updated after insertions and deletions. A different approach let the tree assume any shape as long as its height is logarithmic. In this way there is no need of storing and checking the balance information, but it is sufficient to check whether the maximal possible height has been exceeded. Trees of this kind, called General Balanced Trees, introduced by [7] and later rediscovered by [43] under the name of scapegoat trees, can be efficiently maintained and require as additional space only that for the pointers. They are restructured with an operation, called partial rebuilding that transforms a subtree of the tree in a perfectly balanced tree. The operation is expensive having a cost proportional to the number of nodes of the subtree, but performed rarely hence has a low amortized cost.

The COG-tree of $n$ nodes has an improved cache complexity of $O(\log_B n)$ amortized block transfers and $O(\log n)$ amortized time for updates, and $O(\log_B n)$ block transfers and $O(\log n)$ time for searches. Same as before, the $O(\log_B n)$ amortized block transfers for update is theoretically efficient. The space occupancy is $O(n)$ extra bits besides the space needed to store the keys alone.

By applying the core partitioning scheme on AVL trees, and showing that it can not be maintained in amortized logarithmic time, another question would be whether the core partitioning scheme idea can be applied to AVL trees with constant or logarithmic average time? We give a positive answer to this question too by showing a simple version of core partitioning to maintain the nodes of an AVL tree in external memory model. For $n$ keys and block size $B$, the searching cost is $O(\log_B n)$ block transfers in the worst case, while the updating cost is equal to the searching cost plus $O(1)$ expected block transfers (for restructuring the tree) assuming that AVL trees of the same height are uniformly distributed. The analysis is based on the fact that the expected cost
1.3. OTHER RESULTS ON PROPERTIES OF AVL TREES

is constant under the assumption that rebalancing a node has a cost proportional to its subtree size. As before, we preserve the original topology and the insertion algorithms of the 1962 paper introducing AVL trees. We only require a greedy post-processing for keeping the nodes in the blocks of the external memory.

We adopt the external memory model [2] to evaluate the I/O complexity, using $B$ to denote the block size of the data transfers between main and external memory. The I/O complexity accounts for the number of block transfers performed during the computation. We show that in an AVL tree of height $H$, the topmost $\lceil H/2 \rceil$ levels form a complete binary tree. To apply the core partitioning we can choose any $h^* \leq \lceil H/2 \rceil$. As a result, for $n \geq B$, the AVL tree can be decomposed with $h^* = \log \sqrt{B}$ into cores with a constant size ($B$) we call them here hats, obtaining a hat partitioning scheme. The maintenance algorithm is using the basic idea of core partitioning.

Comparing our partitioning problem to some previous work, we observe that the randomized search trees in [93] have an expected cost of $O(\log n)$ for a rotation, when the cost of the rotation is proportional to the subtree size of the rotated node. If we apply this analysis to our partitioning scheme the expected cost is $O(\log n)$ instead of the $O(1)$ cost that we propose in this paper. We should observe that the former bound is for a randomized algorithm while the latter is the average-case analysis of our deterministic scheme. With our hat partitioning scheme the original AVL trees can actually meet space and external memory efficiency in a unified way.

Mehlhorn and Tsakalidis in [70] showed that the total rebalancing cost for a sequence of $n$ arbitrary insertions is at most $2.618n$. For random insertions the bound is improved to $2.26n$. We also prove that this cost is constant in average.

1.3 Other Results on Properties of AVL Trees

Studying the properties of these trees also lead us to some other new properties of AVL trees and trees with bounded degree, namely, we present gap terminology and we prove Tarjan et al. ’s conjecture on the number of rotations in a sequence of deletions and insertions and finally, we generate trees with bounded degree in an specified ordering (A-order).

Gaps in AVL trees are special tree edges such that the height difference between the subtrees, rooted at their two endpoints, is equal to 2. Using gaps we prove the Basic-Theorem that allows to express the size of a given AVL tree in terms of the heights of the gaps. The Basic-Theorem can represent any AVL tree (and its subtrees) with a series of powers of 2 (of the heights of the gaps) instead of classic representation of AVL trees. The Basic-Theorem and its corollaries are
interesting to characterize the tree size of any AVL tree with a very simple and useful formula. They describe the precise relationship between the size of the tree and the heights of the nodes, also the subtree sizes and the heights of the gaps, and finally they independently describe the relationship between the heights of the nodes and the heights of the gaps. We will also investigate how gaps change (they disappear or are created) in an AVL tree during a sequence of insertions and deletions.

As we know, an insertion in an $n$-node AVL tree takes at most two rotations, but a deletion in an $n$-node AVL tree can require $\Theta(\log n)$. A natural question is whether deletions can take many rotations not only in the worst case but in the amortized case as well. A sequence of $n$ successive deletions in an $n$-node tree takes $O(n)$ rotations [102], but what happens when insertions are intermixed with deletions?

Heaupler, Sen, and Tarjan [50] conjectured that alternating insertions and deletions in an $n$-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations, but they provided no construction to justify their claim. Using Rank terminology presented by Haeupler, Sen, and Tarjan [50], we provide such a construction which causes each deletion to do $\Omega(\log n)$ rotations: we show that, for infinitely many $n$, there is a set $E$ of expensive $n$-node AVL trees with the property that, given any tree in $E$, deleting a certain leaf and then reinserting it produces a tree in $E$, with the deletion having performed $\Theta(\log n)$ rotations. One can do an arbitrary number of such expensive deletion-insertion pairs. The difficulty in obtaining such a construction is that in general the tree produced by an expensive deletion-insertion pair is not the original tree. Indeed, if the trees in $E$ have even height $k$, $2^{k/2}$ deletion-insertion pairs are required to reproduce the original tree.

Finally the last result in this thesis is the generation of trees with bounded degree in A-order. Exhaustive generation of certain combinatorial objects has always been of great interest for computer scientists. Designing algorithms to generate combinatorial objects has long fascinated mathematicians and computer scientists as well. Some of the earlier works on the interplay between mathematics and computer science have been devoted to combinatorial algorithms. Because of its many applications in science and engineering, the subject continues to receive much attention.

Studying combinatorial properties of restricted graphs or graphs with configurations has also many applications in various fields such as machine learning and chemoinformatics. Studying combinatorial properties of restricted trees and outerplanar graphs (e.g. ordered trees with bounded degree) can be used for many purposes including virtual exploration of chemical universe, reconstruction of molecular structures from their signatures, and the inference of structures of chemical
compounds [119, 95, 42, 48, 98, 52, 13]. Therefore, in Chapter 7, we will study the generation of unlabeled ordered trees whose nodes have maximum degree $\Delta$. For the sake of simplicity, we denote such a tree by $T^\Delta$ tree, we also use $T_n^\Delta$ to denote the class of $T^\Delta$ trees with $n$ nodes.

Typically, trees are encoded as strings over a given alphabet and then these strings (called codeword) are generated [79]. Any generating algorithm is characterized by the ordering it imposes on the set of objects being generated and by its complexity. The most well-known orderings on trees are A-order and B-order [117]. Besides the generation algorithm for trees, ranking and unranking algorithms are also important in the concept of tree generation [88, 36, 117]. Given a specific order on the set of trees, the rank of a tree (or corresponding sequence) is its position in the exhaustive generated list, and the ranking algorithm computes the rank of a given tree (or corresponding sequence) in this order. The reverse operation of ranking is called unranking; it generates the tree (or sequence) corresponding to a given rank. For this class of trees, besides an efficient algorithm of generation in A-order we present an encoding over 4 letters and size $n$ with two efficient ranking and unranking algorithms. The generation algorithm has $O(n)$ time complexity in worst case and $O(1)$ in average case. The ranking and unranking algorithms have $O(n)$ and $O(n \log n)$ time complexity, respectively. The presented ranking and unranking algorithms use a precomputed table of size $O(n^2)$ (assuming $\Delta$ is constant).

1.4 Thesis Organization and Overview

In summary, this assertion is organized as follows. Some preliminaries on binary search trees, external memory model, cache-oblivious model, important search tree data structures, and the concept of exhaustive generation of trees with bounded degree are presented in Chapter 2. In Chapter 3, we propose a general method to store the nodes of balanced search trees (the core partitioning scheme). Then the core partitioning scheme is applied directly to plain binary search trees in Chapter 4, while in Chapter 5, we study the constant average time of core partitioning on AVL trees. In Chapter 6, we present some new features and properties of AVL trees including the proof of Heaupler, Sen, and Tarjan [50] conjecture that alternating insertions and deletions in an $n$-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. Chapter 7 is dedicated to generation of trees with bounded degree which is a byproduct of our research. Finally, some concluding remarks and suggestions for further research are given in Chapter 8.
Chapter 2

Background

In this chapter, we study some basic concepts of important binary search trees [32, 57, 110, 29, 91], external memory model [2] and cache-oblivious model [82], modern data structures for these two models [12, 19, 20, 21, 26, 75], and the concept of exhaustive generation of trees with bounded degree [80, 97, 117].

2.1 Trees and Binary Search Trees

Trees are one of the most important basic and simple data structures for organizing information in computer science. Trees have many applications including database generation, decision table programming, analysis of algorithms, string matching [57, 91], switching theory, theoretical VLSI circuit design [103], image processing [92, 100], and maintaining data [76]. Trees are also widely used for showing the organization of real world data such family/genealogy trees [33], taxonomies, and modeling of the connections between dendrites of the brain in computational neuroscience [23]. Also in image processing, particular cases of t-ary trees, quadtrees and octrees, are used for the hierarchical representation of 2 and 3 dimensional images, respectively [92].

There are many notions for trees as well as various notations concerning graphs. We suppose the reader is familiar with basic concept of graph, trees and algorithms. In this section, some definitions and properties of several kinds of trees are presented.

A rooted tree is a tree in which one of the nodes is distinguished from the others. The distinguished node is called the root of the tree. We often refer to a node of a rooted tree as a node of the tree. In a rooted tree, degree of a node is defined as the number of its children and a leaf is a node of degree 0. An internal node is a node of degree at least 1. A labeled tree is a tree in which
Each node is given a unique label. The nodes of a labeled tree on \( n \) nodes are typically given the labels \( 1, 2, \ldots, n \).

Consider a node \( x \) in a rooted tree \( T \) with root \( r \). Any node \( y \) on the unique path from \( r \) to \( x \) is called an ancestor of \( x \). If \( y \) is an ancestor of \( x \), then \( x \) is a descendant of \( y \). If \( y \) is an ancestor of \( x \) and \( x \neq y \), then \( y \) is a proper ancestor of \( x \) and \( x \) is a proper descendant of \( y \). The subtree rooted at \( x \) is the tree consisting of the descendants of \( x \), rooted at \( x \). The length of the path from the root \( r \) to a node \( x \) plus one is the level (depth) of \( x \) in \( T \). The height of a node in a tree is the number of nodes on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root. The height of a tree is also equal to the largest level of nodes in the tree.

An ordered tree or plane tree is a rooted tree for which an ordering is specified for the children of each node. This is called a “plane tree” because an ordering of the children is equivalent to an embedding of the tree in the plane, with the root at the top and the children of each node lower than that node. Given an embedding of a rooted tree in the plane, if one fixes a direction of children, say left to right, then an embedding gives an ordering of the children. Conversely, given an ordered tree, and conventionally drawing the root at the top, then the child nodes in an ordered tree can be drawn left-to-right, yielding an essentially unique planar embedding. Figure 2.1 shows an embedding of an ordered rooted tree in the plane with root labeled by ‘\( x \)’, in this figure, the node with gray color are the internal ones and the rest are the leaves.

2.1.1 Binary Search Trees

There are a lot of different classes of trees presented in literature, in this section, we list and introduce some which are important.
t-ary Trees and Binary Trees:

A *t*-regular tree is a rooted tree in which each node has *t* children. 2-regular trees are also called *binary trees*, while 3-regular trees are sometimes called *ternary trees*. To construct a *t*-regular tree from a rooted tree, to every node which has *q* < *t* children, *t* − *q* special nodes are added as its children. These special nodes are called *null pointers* (*null nodes*). Clearly, the constructed tree is not unique. A *t*-ary tree is an ordered *t*-regular tree, in which every internal node has exactly *t* ordered children. An *n*-node *t*-ary tree *T* is a *t*-ary tree with *n* nodes, *i.e.*, |*T*| = *n*. Clearly, an *n*-node *t*-ary tree has (*t* − 1)*n* + 1 null pointers. The *t*-ary tree that contains no nodes is called an empty tree or null tree. Also, a *t*-ary tree *T* can be defined recursively as being ‘a null pointer’ or a node together with a sequence *T*₁, *T*₂, . . . , *T*ₜ of *t*-ary trees. *T*ᵢ is called a subtree of *T*. So sometime a tree *T* is shown as *T* = *T*₁, *T*₂, . . . , *T*ₜ.

It is well known that binary trees with *n* internal nodes are counted by the *n*<sup>th</sup> Catalan number [99]:

$$C_n = \frac{1}{n+1} \binom{2n}{n},$$

and it is also known that the number of *t*-ary trees with *n* internal nodes is [99, 45]:

$$\frac{1}{tn+1} \binom{tn+1}{n}.$$

A complete *t*-ary tree is a *t*-ary tree in which all leaves have the same level. For *t* = 2, the *t*-ary trees are called binary trees, where each node has a left and a right child. Also, a binary tree is best described recursively. A binary tree *T* is a structure defined on a finite set of nodes that either:

- contains no node, or

- is composed of three disjoint sets of nodes: a root node, a set of nodes called left subtree of *T*, and a set of nodes called right subtree of *T*, and their roots are called left child and right child of the root, respectively. Both subtrees are themselves binary trees.

A binary tree is not simply an ordered tree in which each node has degree at most 2. For example, in a binary tree, if a node has just one child, the position of the child, whether it is the left child or the right child, matters. In an ordered tree, there is no distinguishing a solo child as being left or right. Sometimes a binary tree *T* is shown as *T* = *T*ᴸ *T*ᴿ, in which *T*ᴸ is the left subtree and *T*ᴿ
is the right subtree of $T$. Clearly, a complete balanced binary tree of height $h$ has $2^h - 1$ nodes.

*Binary search tree (BST)* based on binary tree structure, is basically a data structure on binary trees where each node has a comparable *key* (and an associated *value*) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node’s left subtree and smaller than the keys in all nodes in that node’s right subtree. This data structure is one of the most common data structure who guarantees to restore data in a sorted way. The size of a BST is only limited by the amount of free memory in the operating system. As mentioned, since the main advantage of binary search trees is that it remains ordered, it provides quicker search times than many other data structures. The common properties of binary search trees are as follows.

- The left subtree of a node contains only nodes with keys less than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtrees are binary search trees.
- Each node can have up to two successor nodes.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records. The major advantage of binary search trees over other data structures is that the related *sorting algorithms* and *search algorithms*. The other advantages are:

- BST is mostly fast in insertion and deletion etc (depends on how much balanced the tree is).
- Very efficient and its implementation is easier than other data structures.
- Stores keys in the nodes in a way that searching, insertion and deletion can be done efficiently.
- Nodes in tree are dynamic in nature.

Some of their disadvantages are as follows.

- The shape of the binary search tree totally depends on the order of insertions, and it can be very *unbalanced*, so that the search operation has worst case time complexity $O(n)$.
- The keys in the binary search tree may be long and the run time may increase.
After a long intermixed sequence of random insertion and deletion, the expected height of the tree approaches the square root of the number of keys which grows much faster than \(\log n\).

In the decades, researchers have introduced many interesting binary search trees and other tree data structures to keep the tree as balanced as possible, so the main operations such as insertion and search have the worst case cost \(O(\log n)\). In the following we study the most important ones.

**AVL Trees**

*Height-balanced binary trees (hb-trees)* have the property that, for every node, the heights of the left and right subtrees differ at most by an integer value \(\Delta\) [40, 67]. AVL trees, the original type of balanced binary search trees were introduced over 50 years ago [1] but still are remarkable for their efficiency. AVL trees are the first family of hb-trees which appeared in the literature, for which \(\Delta = 1\). Since the invention of AVL trees in 1962, a wide variety of ways to balanced binary search trees have been proposed. They are mostly based on some particular *rebalancing* algorithms executed after an insertion or deletion to maintain the tree balanced.

An AVL tree (“Adelson-Velskii and Landis’ tree”, named after the inventors) is a *self-balancing* binary search tree. It was the first data structure of this kind to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Search, insertion, and deletion all take \(O(\log n)\) time in both the “average” and “worst cases”, where \(n\) is the number of nodes in the tree prior to the operation. An example of AVL tree is given in Figure 2.2. Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications (insertions and deletions) are followed by some more operations called *tree rotations*, which help to restore the height balance of the subtrees. Figure 2.3 illustrates a rotation.

The time complexity for search is \(O(\log n)\), and the time complexity for insertion is \(O(\log n)\) for searching the place where the key must be inserted, plus a constant number of rebalancing operations which take constant time if the tree is maintained by pointer-based data structures. For deletion, the time required is again \(O(\log n)\) for search, plus a maximum of \(O(\log n)\) rotations on the way back to the root, so the operation can be completed in \(O(\log n)\) time. An insertion in an \(n\)-node AVL tree takes at most two rotations, but a deletion in an \(n\)-node AVL tree can take \(\Theta(\log n)\). Heaupler, Sen, and Tarjan [50] conjectured that alternating insertions and deletions in an \(n\)-node AVL tree can cause each deletion to do \(\Omega(\log n)\) rotations, but they provided no
CHAPTER 2. BACKGROUND

Figure 2.2: An example of AVL trees.

Figure 2.3: Right rotation at node $x$. Triangles denote subtrees. The inverse operation is a left rotation at $y$.

construction to justify their claim, and this would be one of our challenges.

Besides AVL trees, many other interesting data structures, such as B-trees [31, 16, 32], redblack trees [15], weight-balanced trees [74], 2-3 trees [46], and $(a,b)$-trees [55] have been introduced, probably none of them reaching the same appeal. If we look at “vintage” AVL trees with today’s eyes, they are indeed pretty modern. The English translation [1] of the Russian paper by Adel’son-Vel’skiĭ and Landis is very close, except some terminology, to the way AVL trees are currently presented in classroom. The rebalancing operations after an insertion are extremely elegant. In the following we study other well known data structures.

B-tree, 2-3 Tree and $(a,b)$-trees

B-tree can be considered as a generalization of a binary search tree in which a node can have more than two children and the structure remains always balanced [31]. In B-trees, internal nodes can have any number of children within some pre-defined range. For example, 2-3 trees are B-trees which any internal node can have only 2 or 3 children [46] and $(a,b)$-tree is a B-tree where each node has at least $a$ and at most $b$ children and $a \leq b/2$ [55]. Because of some rebalancing operations
on B-trees, the structure will be always balanced and it guarantees logarithmic time complexity in both worst case and average case for search, insertion and deletion. The range of possible number of children will be optimized regarding to hardware specification to make it practically fast for systems that read and write large blocks of data. More information can be found in almost all text books of algorithms and data structures, for example [16, 32, 57].

**Red-Black Trees**

A red-black tree is another type of self-balancing binary search trees. This structure comes with an extra bit of storage per node which is its color (red or black). The self-balancing is provided by painting each node with one of two colors (these are typically called ‘red’ and ‘black’, hence the name of the trees) in such a way that the resulting painted tree satisfies certain properties that don’t allow it to become significantly unbalanced. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently. Formally a red-black tree is a binary search tree with the following properties [15, 46, 32]:

- A node is either red or black.
- The root is black.
- All leaves are black.
- Every red node must have two black child nodes.
- Every path from a given node to any of its descendant leaves contains the same number of black nodes.

These properties guarantees that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that a red-black tree of \( n \) internal nodes has height at most \( 2 \log(n + 1) \) [32]. The rebalancing operations for this tree after an insertion or deletion takes \( O(\log n) \) time in worst case [32]. Considering the fact that each black node may have 0, 1 or 2 red children, a red-black tree can be expressed as a B-tree of order 4, where each node can contain between 1 to 3 values and (accordingly) between 2 to 4 child pointers. In such a B-tree, each node will contain only one value matching the value in a black node of the red-black tree, with an optional value before and/or after it in the same node, both matching an equivalent red node of the red-black tree.
Weight-Balanced Trees

Like other self-balancing trees, a weight-balanced tree (WBT) stores information pertaining to balance in their nodes and perform rotations to restore balance when it is disturbed by insertion or deletion operations. Specifically, each node stores the size of the subtree rooted at the node, and the sizes of left and right subtrees are kept within some factor of each other. The number of elements in a tree is equal to the size of its root, and the size information is exactly the information needed to implement the operations of an order statistic tree. Weight-balanced trees are also called trees of bounded balance, or BB[$\alpha$]-trees [74].

For a binary tree, the weight is the number of null pointers (null nodes), which is equivalent to the number of nodes (i.e., the size) plus one. The weight of a node $u$ is denoted by $w(u)$ and its balance $\beta(u) = w(u.l)/w(u)$ is the ratio between the weight of $u$’s left child and $u$’s weight (note that $w(null) = 1$ by definition of weight) [74].

For a parameter $\alpha$, where $0 < \alpha \leq 1$, a weight-balanced tree (a.k.a. BB[$\alpha$]-tree) is a binary search tree where each node $u$ satisfies $\alpha \leq \beta(u) \leq 1 - \alpha$, which is equivalent to say that $\alpha \cdot w(u) \leq w(u.l)$, $w(u.l) \leq (1 - \alpha) \cdot w(u)$ for each node $u$ and its two children $u.l$ and $u.r$.

For example, the tree shown in Figure 2.4 is a BB[$\alpha$]-tree for $\alpha = 2/7$ while it is not for $\alpha = 1/3$.

As observed by the inventors Nievergelt and Reingold [74], a node of weight 3 should have one child of weight 1, so they assume that $0 < \alpha \leq 1/3$. Moreover, Blum and Mehlhorn [25] show that rebalancing a BB[$\alpha$]-tree with rotations can be done when $2/11 < \alpha \leq 1 - \sqrt{2}/2 = 0.2928\ldots$. When $\alpha$ is strictly inside this interval, they show that there exists $\delta > 0$ depending on $\alpha$ such that an unbalanced node $u$ has balance factor $(1 + \delta)\alpha \leq \beta(u) \leq 1 - (1 + \delta)\alpha$ after its balance is restored using rotations. Overmars [77, Sect.4.2] shows that rebalancing can be also done with partial rebuilding, and this only requires $0 < \alpha < 1/2$ and obtains a value of $\beta(u)$ close to 1/2.
after restoring the balance of $u$.

**Rank-Balanced Trees**

*Rank-balanced trees* are an extension of AVL trees, where each node $x$ has an integer rank $r(x)$ which is proportional to its height. If $x$ is a node with parent $p(x)$, the rank difference of $x$ is $r(p(x)) - r(x)$. A node is called an $i$-child if its rank difference is $i$, and an $i,j$-node if its children have rank differences $i$ and $j$. The initial rank rule is that every node is a 1,1-node or a 1,2-node. This rule gives exactly the AVL trees. If no deletions occurs, a rank-balanced tree remains an AVL tree; with deletions, 2,2-nodes will be allowed. The rank and hence the height of a rank-balanced tree is at most $2 \log n$. Considering an initially empty tree and a sequence of $m$ insertions and $d$ deletions ($n = m - d$), it has been shown that the height of the resulting rank-balanced tree is at most $\log_{\phi} m$ and the total number of rebalancing steps is at most $3m + 6d$, which means $O(1)$ amortized rebalancing steps per insertion or deletion. Rank-balanced trees can be rebalanced bottom-up after an insertion or deletion using at most two rotations worst-case [49, 50].

**Relaxed AVL Trees (ravls)**

*Relaxed AVL trees (ravls)* are a special class of rank-balanced trees in which the rank difference from child to parent can be non-constant [63, 94]. In this relaxation of AVL trees, rebalancing is done after insertions but not after deletions, yet access time remains logarithmic in the number of insertions. The structure maintains insertion and deletion as follows.

To insert a new item into such a tree, first do a search on its key. When the search reaches a missing node, replace this node with the new item. For deletion, first find the item to be deleted (by doing a binary search). If neither child of the item is missing, find either the next item or the previous item, by walking down through left (right) children of the right (left) child of the item, until reaching a node with a missing left (right) child. Then swap the item with the item found. Now the item to be deleted is either a leaf or has one missing child. In the former case, replace it by a missing node; in the latter case, replace it by its non-missing child. If each node has pointers to its children, an access, insertion, or deletion takes $O(h + 1)$ time in the worst case, where $h = \log_{\phi} m$ is the height of the tree and $\phi$ and $m$ are the golden ratio and the number of insertions, respectively. This structure needs $O(\log \log m)$ bits of balance information per node, or $O(\log \log n)$ with periodic rebuilding, where $n$ is the number of nodes. An insertion takes up to
two rotations and constant amortized time.

**Randomized Search Trees**

Randomized search tree is a data structure for a set $X$ of pairs of key and priority. Randomized search trees are based on a tree called treap which is a rooted binary tree of $X$ that is arranged in inorder with respect to the keys and in heaporder with respect to the priorities. Inorder means that the keys are sorted with respect to inorder traversal and heaporder means that the priorities are sorted as a heap (or for any node $v$ the priority of $v$ is greater than priorities of all its ascendants) [93, 10]. In Figure 2.5 an example of treap is shown.

Randomized search trees have an expected cost of $O(\log n)$ for a rotation, when the cost of the rotation is proportional to the subtree size of the rotated node.

**Splay Trees**

The splay tree presented by Sleator and Tarjan [96] does not require any balance information stored in the nodes. However, the height of a splay tree is not guaranteed to be $O(\log n)$. The logarithmic cost for searching in a splay tree is amortized not worst case. The splay tree is a self-adjusting form of binary search trees. On an $n$-node splay tree, all the standard search tree operations have an amortized time bound of $O(\log n)$ per operation. In splay trees a simple heuristic restructuring function called splaying is applied whenever the tree is accessed. To splay a tree at a node $v$, repeat the following splaying step until $v$ is the root of the tree.
• Case 1 (zig): If \( p(v) \) is the tree root, rotate the edge joining \( v \) with \( p(v) \) (This case terminates splaying the tree).

• Case 2 (zig-zig): If \( p(v) \) is not the root and \( v \) and \( p(v) \) are both left or both right children, rotate the edge joining \( p(v) \) with its grandparent \( p(p(v)) \) and then rotate the edge joining \( v \) with \( p(v) \).

• Case 3 (zig-zag): If \( p(v) \) is not the root and \( v \) is a left child and \( p(v) \) a right child, or vice versa, rotate the edge joining \( v \) with \( p(v) \) and then rotate the edge joining \( v \) with the new \( p(v) \).

Splaying, is similar to move-to-root in that it does rotations bottom-up along the access path and moves the accessed item all the way to the root. But it differs in that it does the rotations in pairs, in an order that depends on the structure of the access path.

**Scapegoat Trees**

*Scapegoat trees* presented by Galperin and Rivest in [44] are binary search trees that unlike most balanced-tree schemes, do not require keeping extra data (e.g. ‘colors’, ‘weights’, ‘rank’, etc.) in the tree nodes. Each node in the tree contains only a key value and pointers to its two children. Associated with the root of the whole tree are the only two extra values needed by the scapegoat scheme: the number of nodes in the whole tree, and the maximum number of nodes in the tree since the tree was last completely rebuilt. In a scapegoat tree a typical rebalancing operation begins at a leaf, and successively examines higher ancestors until a node (the scapegoat) is found that is so unbalanced that the entire subtree rooted at the scapegoat can be rebuilt at zero cost, in an amortized sense. Scapegoat trees provides worst-case \( O(\log n) \) lookup time, and \( O(\log n) \) amortized insertion and deletion time.

**General Balanced Trees**

Anderson’s *general balanced trees* [7] are maintained by partial rebuilding, this idea is similar to the technique that we explain in Chapter 4. For general balanced trees, in order to achieve efficient maintenance of a balanced binary search tree, no shape restriction other than a logarithmic height is required. The obtained class of trees, general balanced trees, may be maintained at a logarithmic amortized cost with no balance information stored in the nodes. Thus, whenever amortized bounds are sufficient, there is no need for sophisticated balance criteria. The maintenance algorithms use
partial rebuilding. The main idea in maintaining a general balanced tree is to let the tree take any shape as long as its height does not exceed \( c \log |T| \) for some constant \( c > 1 \). When this criterion is violated, the height can be decreased by partial rebuilding at a low amortized cost. Anderson in [7] proved that the amortized cost incurred by general balanced trees is lower than what has been shown for weight-balanced trees. In general balanced trees, no rebalancing is performed. Scapegoat trees and general balanced trees restructure using partial rebuilding that transforms a subtree of the tree in a perfectly balanced tree. The operation is expensive having a cost proportional to the number of nodes of the subtree, but performed rarely hence has a low amortized cost.

### 2.1.2 Tree Representation

To represent a \( t \)-ary tree in a computer, the most common but non efficient way is *linked representation*. In this representation, each internal node of tree will have \( t + 1 \) fields: one *data* and \( t \) children fields. Data field is used for holding data (or label) of a node and \( i^{th} \) child field points to \( i^{th} \) subtree of node. In addition a pointer is used to point to a root of a tree. For binary trees, we have two pointer fields in each node, called left child and right child fields. In this representation, no memory is needed for null pointers (null nodes) and all pointers to empty trees are *null*.

The other method of tree representation is when a tree is represented by integer or alphabet sequences. This operation is called *tree encoding*. Basically, the uniqueness of encoding, the length of the encoding, and the capability of constructing the tree from its representation, which is called *decoding*, are essential considerations in the design of the tree encoding schema [68].

### 2.1.3 Tree Traversal

There are many operations that may be performed on trees. One notion that arises frequently is the idea of *traversing* a tree or *visit* each node in a tree exactly once. A full traversal produces a linear order for the information in a tree. Here, first we define the traversal operations for binary trees and then extend some of them to \( t \)-ary trees.

In a binary tree, if we assume that \( L, V, \) and \( R \) stand for moving left, visiting the node, and moving right, respectively, and if we adopt the convention that we traverse left before right, then the only three traversals will be: \( LVR, LRV \), and \( VLR \). To these traversal types we assign the names *inorder*, *postorder*, and *preorder* respectively. The earliest algorithms represented for tree traversals which mainly use stacks, can be easily written in recursive form. Recursive algorithms for inorder, preorder, and postorder traversals are similar, only the position of visiting the nodes
Procedure InOrder(Current: TreePtr)
begin
  if (Current ≠ NULL) then begin
    InOrder(Current.LeftChild);
    Visit(Current.Data);
    InOrder(Current.RightChild);
  end;
end;

Figure 2.6: An inorder traversal algorithm for binary trees.

Procedure PostOrder(Current: TreePtr)
begin
  if (Current ≠ NULL) then begin
    PostOrder(Current.LeftChild);
    PostOrder(Current.RightChild);
    Visit(Current.Data);
  end;
end;

Figure 2.7: A postorder traversal algorithm for binary trees.

Procedure PreOrder(Current: TreePtr)
begin
  if (Current ≠ NULL) then begin
    Visit(Current.Data);
    PreOrder(Current.LeftChild);
    PreOrder(Current.RightChild);
  end;
end;

Figure 2.8: A preorder traversal algorithm for binary trees.

differ due to the corresponding traversal. The inorder, postorder, and preorder traversal algorithms are presented in Figures 2.6, 2.7 and 2.8, respectively. By using inorder traversal in binary search trees, we are able to list the keys ordered (sorted).

The preorder and postorder traversal can be extended and used for any class of trees, e.g., for t-ary trees in preorder traversal, at first, we visit data field of a node and then traverse the t subtrees of this node one by one. The same procedure can be applied to AVL trees, trees with bounded degree, etc.

2.2 External Memory & Cache-Oblivious Memory Models

In this thesis, we adopt both external memory model [2] and cache-oblivious model [41, 82] to evaluate I/O complexities. The basic computer systems use a memory hierarchy, the CPU has access to a relatively small but fast pool of solid-state storage space, the main memory; it could
also communicate with other, slower but potentially larger storage spaces, the *external memory*. The memory hierarchies of modern computers are composed of several levels of memories starting from the *caches*. Caches have very small access time and capacity comparing to main memory and external memory. From cache to main memory, then to external memory, access time and capacity increases significantly.

Since accessing data in main memory is *expensive* relative to the processor speeds, modern processors make use of processor caches. A processor *cache* is a block of low-latency memory that sits between the processor and main memory, and stores the contents of the most recently accessed memory addresses. Latency in retrieving data from the cache is one to two orders of magnitude smaller than the latency in retrieving data from the main memory [41, 82, 51]. In the following we study external memory model and cache-oblivious model which describe different layers of memory hierarchy.

### 2.2.1 External Memory Model

Accessing an item from external storage is extremely slow. In 2013 as mentioned in [71], the average access time of hard disks was 160000 times slower than random access memories (RAMs) and the average access time of solid state drives was 2500 times slower than random access memory (RAM). These speeds are fairly typical; accessing a random byte from RAM is thousands of times faster than accessing a random byte from a hard disk or solid-state drive. Access time, however, does not tell the whole story. When we access a byte from a hard disk or solid state disk, an entire block of the disk is read.

This is the idea behind the external memory model of computation, illustrated schematically in Figure 2.9. In this model, the computer has access to a large external memory in which all of the data resides. This memory is divided into memory blocks each containing $B$ words. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are free; they take no time at all. The fact that internal memory computations are free may seem a bit strange, but it simply emphasizes the fact that external memory is so much slower than RAM [71]. We assume that each external memory access (called an I/O operation or just I/O) transmits one page of $B$ elements. We measure the efficiency of an algorithm in terms of the number of I/Os it performs and the number of disk blocks it uses.

External memory data structures have been developed for a wide range of applications, includ-
2.2. EXTERNAL MEMORY & CACHE-OBLIVIOUS MEMORY MODELS

Figure 2.9: The external memory model.

2.2.2 Cache-Oblivious Memory Model

The memory hierarchies of modern computers are composed of several levels of memories, that starting from the caches, have increasing access time and capacity. Most of today’s processor architectures use a hierarchical memory system: a number of caches are placed between the processor and the main memory. Caching has become an increasingly important factor in the practical performance of main-memory data structures. Processor speeds have increased faster than memory speeds, and many applications that previously needed to read data from disk can now fit all of the necessary data in main memory. The relative importance of caching will likely increase in the future [51, 86, 90]. The cache-oblivious model introduced by [41, 82] allows to consider only a two-level hierarchy, but proves results for a hierarchy composed of an unknown number of levels. Cache-oblivious model helps to evaluate the I/O complexity, here called cache complexity and still expressed as number of block transfers of size $B$. Note that $B$ is now an unknown parameter for the block size and a cache-oblivious algorithm is completely unaware of the value of $B$ used by the underlying system.

This model is composed of two parts: the ideal-cache model and cache-oblivious algorithms. The ideal-cache model has two levels of memory: cache and main memory. The cache contains $M$ locations partitioned into blocks of $B$ contiguous locations each. The main memory can be arbitrarily large. The processing unit can address the locations of the main memory but only the data in cache can be used. If the data needed by the computation is not in cache, a cache fault (cache misses) is caused and the corresponding block is transferred from the main memory. The
number of processor cache faults has a critical impact on the performance of the system. The goal is to improve performance by reducing the number of processor cache faults that are incurred during a search operation [51]. When the cache is full, an optimal off-line replacing strategy is used to replace a block with the new one. The cache is fully associative: each block from main memory can be stored anywhere in the cache. An algorithm operating in the ideal-cache model cannot directly manage the transfers of blocks.

There are two types of cache-conscious algorithms; namely, cache-sensitive (or cache-aware) model, where the parameters of the caches are assumed to be known to the implementation (similar to external memory model) and in contrast, cache-oblivious algorithms that attempt to optimize themselves to an unknown memory hierarchy.

In this thesis, we focus on “cache-oblivious model”; because the cache-sensitive model is covered by our results in external memory model. An algorithm is cache-oblivious if it cannot explicitly use the parameters that are specific to the given memory hierarchy. If the algorithm operates in the ideal-cache model, it cannot be defined in terms of parameters $B$ and $M$. Being cache-oblivious is an algorithm’s strength: since the cache complexity analysis holds for any value of $B$ and $M$, it holds for any level of a more general, multi-level memory hierarchy, as shown in [41]. The cache-oblivious model can be seen as a “successor” of the RAM model, a successor that incorporates a lot of the new architectural aspects which characterize the real world computing systems in a more refined way.

### 2.3 Data Structures for External Memory & Cache-Oblivious Memory Models

Here we study more recent data structures designed for and working with external memory model or cache-oblivious memory model. These data structures are normally more complicated than static data structures but work more efficiently with real world computers.

A range query is a common database operation that retrieves all records where some value is between an upper and lower boundary. A diagonal corner query is a two sided range query whose corner must lie on the line $x = y$ and whose query region is the quarter plane above and to the left of the corner. Kanellakis et al. [56] developed the metablock tree for answering diagonal corner queries in optimal $O(\log_B N + T/B)$ I/Os using optimal $O(N/B)$ blocks of external memory. Here $T$ denotes the number of points reported. The structure supports insertions only
in \( O(\log B N + (\log_B^2 N)/B) \) I/Os amortized. A simpler static structure with the same bounds was described by Ramaswamy in [84].

The Arge and Vitter weight-balanced B-tree [12] presented in 2003, works in external memory, weight-balanced B-tree is an optimal external memory data structure for answering stabbing queries on a set of dynamically maintained intervals. Given a set of intervals, a stabbing query with a point \( p \) asks for all intervals containing \( p \). Part of the structure uses a weight-balancing technique for efficient worst-case manipulation of balanced trees. The structure uses \( O(N/B) \) disk blocks to maintain a set of \( N \) intervals such that insertions and deletions can be performed in \( O(\log_B N) \) I/Os and such that stabbing queries can be answered in \( O(\log_B N + T/B) \) I/Os, where \( T \) denotes the number of points reported.

The buffer tree presented in [11] is a well-known example of a general technique for external memory that supports batched operations I/O efficiently. The main idea in the technique is to perform operations on an external (high fan-out) tree data structure in a lazy manner using main-memory-sized buffers associated with internal nodes of the tree. As an example, imagine we are working on a height \( O(\log_m n) \) search tree structure with elements stored in the leaves, that is, a structure with fan-out \( \Theta(m) \) internal nodes and \( N \) elements stored in sorted order in \( n \) leaves with \( \Theta(B) \) elements each, then assign buffers of size \( \Theta(m) \) blocks to each of the \( O(n/m) \) internal nodes of the structure. When we want to insert a new element, we do not search down the tree for the relevant leaf right away. Instead, we wait until we have collected a block of insertions (or other operations), and then we insert this block into the buffer of the root. When a buffer “runs full” the elements in the buffer are “pushed” one level down to buffers on the next level (this is named buffer-emptying process). Deletions or other and perhaps more complicated updates, as well as queries, are basically performed in the same way. Note that as a result of the laziness, we can have several insertions and deletions of the same element in the tree at the same time, and we therefore “time stamp” elements when they are inserted in the root buffer. The laziness also means that queries are batched since a query result may be generated (and reported) lazily by several buffer-emptying processes. This technique does not seem to smoothly extend to the cache-oblivious model.

Typical cache optimization techniques include clustering, compression and coloring [30, 85]. Clustering tries to pack, in a cache block, data structure elements that are likely to be accessed successively. Compression tries to remove irrelevant data and thus increases cache block utilization by being able to put more useful elements in a cache block. This includes key compression, struc-
ture encodings such as pointer elimination and fluff extraction. Caches have finite associativity, which means that only a limited number of concurrently accessed data elements can map to the same cache line without causing conflict. Coloring maps contemporaneously-accessed elements to non-conflicting regions of the cache [85]. Ladner [62, 61] considered the effects of caches on sorting algorithms and improved performance by restructuring these algorithms to exploit caches. In addition, they constructed a cache-conscious heap structure that clustered and aligned heap elements to cache blocks.

*T-Trees* have been proposed as a better index structure in main memory database systems. A T-Tree is a balanced binary tree with many elements in a node. Elements in a node contain adjacent key values and are stored in order. Its aim is to balance the space overhead with searching time and cache behavior is not considered [64]. T-Trees put more keys in each node and give the impression of being cache conscious. But if we think of it carefully, we can observe that for most of the T-Tree nodes, only two end keys are actually used for comparison. This means that the utilization of each node is low. Since the number of key comparisons is still the same, T-Trees do not provide any better cache behavior than binary search [85].

The simplest cache-sensitive variant of the B-tree is an ordinary *B+-trees* where the node size is chosen to match the size of a cache block [85]. In B+-trees, in each internal node we store keys and child pointers, but the record pointers are stored on leaf nodes only. Multiple keys are used to search within a node. If we fit each node in a cache line, this means that a cache load can satisfy more than one comparison. So each cache line has a better utilization ratio. Although B+-trees were designed for disk-based database systems, they actually have a much better cache behavior than T-trees [85]. *Hash indices* can also benefit from cache optimization. The most common hashing method is the chained bucket hashing [57].

A more advanced version of B+-tree called the Cache-Sensitive B+-tree or CSB+-tree [86] additionally removes pointers from internal nodes by storing the children of a node consecutively in memory. The CSB+-tree has been further optimized using a variety of techniques, such as prefetching, storing only partial keys in nodes, and choosing the node size more carefully [90].

A cache-oblivious layout scheme for fixed-topology trees has been introduced in [3] but it is an open problem to extend it to dynamic trees. Based on this scheme, a new indexing technique called *Cache-Sensitive Search Trees (CSS-trees)* was presented in [90]. The main idea of this technique is to store a directory structure on top of a sorted array. The directory represents a balanced search tree stored itself as an array. Nodes in this search tree are designed to have size matching
the cache-line size of the machine. Therefore, it performs a top-down layout of balanced trees that is apparently close to our core partitioning described in Chapter 3. However, as the partition described in [90] works for any dynamic tree, the resulting blocks are not necessarily cores, and pointers are used internally, because of fixing the so-called broken nodes (and indeed the complete balanced binary trees are never mentioned and exploited in that paper). The authors of [90] report some experimental study to show improvements over traditional trees in practice, but no analysis with provably logarithmic bounds is given for the updates.

van Emde Boas (vEB) layout [18, 20, 26] can compactly stores without using any pointer an array of a constant number of power of two elements. Given a search tree, where each node has $O(1)$ children, vEB layout describes a mapping from the nodes of the tree to their positions in the memory. Assuming the search tree has height $\Theta(\log n)$, this structure performs search operation in $\Theta(\log_B n)$ I/O transfers, which is optimal to within a constant factor. The basic idea of vEB layout is as follows. Suppose the tree has height $h$ which is a power of two. Conceptually split the tree at the middle level of edges, between nodes of height $h/2$ and $h/2+1$. This breaks the tree into the top recursive subtree $A$ of height $h/2$, and several bottom recursive subtrees $B_1, B_2, \ldots, B_k$ of height $h/2$. In particular for complete balanced binary trees (which all internal nodes have 2 children), then the recursive subtrees have size $\sqrt{n+1} - 1$, and $k = \sqrt{n+1}$. The layout of the tree is obtained by recursively laying out each recursive subtree, and combining these layouts in the order $A, B_1, B_2, \ldots, B_k$.

Anderson and Lai in [8] presented a fast updating of well-balanced trees. Their structure maintains binary search trees with an optimal and near-optimal number of incomplete levels. For a binary search tree with one incomplete level, the amortized insertion cost is $O(\log^3 n)$, for a tree with 2 incomplete levels, the amortized insertion cost is $O(\log^2 n)$. Finally, the amount of restructuring work is decreased to $O(\log n)$ by increasing the number of incomplete levels to 4. This yields an improved amortized bound on the dictionary problem.

Binary Trees of Small Height [26] were presented by Brodal et al., this data structure makes use of the ‘fast updating of well-balanced trees’ [8] implemented by an implicit version of the van Emde Boas layout [20, 26, 82]. For a tree of $n$ nodes and block size $B$, this structure requires $(1 + \epsilon)n$ space and performs search operation in the worst case $O(\log_B n)$ block transfers and updates in $O(\log_B^2 n/\epsilon B)$ amortized number of block transfers. This structure allows also efficient range queries in $O(\log_B n + k/B)$ block transfers in the worst case, where $k$ is the output size.

The most efficient data structure for search trees on cache-oblivious model is presented by
Bender et al. in [19], they presented a cache-oblivious data structure called the exponential structures for dynamic searching. An exponential tree is a tree of $O(\log \log n)$ levels where the degrees of nodes descending from the root level decrease doubly exponentially, e.g. as in the series $n^{1/2}, n^{1/4}, n^{1/8}, \ldots, 2$. In the exponential structure, internal nodes may have many children and they are called fat nodes. The layer of a fat node is the number of fat nodes below (i.e., leave fat nodes have level 0). The number of keys stored in a layer $i$ fat node, $i \geq 1$, is in the range $[2^{2^i - 2^i - 1}, 2 \times 2^{2^i}]$, except for the topmost fat node, where the range is given by $[2 \times 2^{2^k - 1}, 2 \times 2^{2^k}]$, and $k$ is the layer of the tree. Each layer 0 fat node contains a single item. Loosely speaking, the volumes of the fat nodes square at each successive layer. When updating, if a layer $i$ fat node $V$ acquires $2 \times 2^{2^i}$, it splits as evenly as possible into two subtrees $V_1$ and $V_2$ in time $O(|V|)$. When $V$ splits, this adds one to the number of its parent’s children. This is accommodated by completely rebuilding the parent. In splitting layer $i$ fat node $V$ into subtrees $V_1$ and $V_2$, besides creating $V_1$ and $V_2$, they copy all of $V$’s descendant fat nodes into either the portion of the array being used for $V_1$ and its descendants (or that being used for $V_2$ and its descendants). Thus they achieved the search time $O(\log_B n)$ I/Os but they increased the space significantly (to $O(n \log^2 n)$).

Then by using buckets of size $\Theta(\log^2 n)$, implemented as two layers of records of size in the range $[\log n, 2 \log n)$, they obtained the space of $O(n)$ words, we think their space can be also reduced to the optimum theoretical result which is $O(n)$ bits.

### 2.4 Exhaustive Generation of Trees with Bounded Degree

The last result in this thesis is the generation of trees with bounded degree in A-order. Therefore, in this section, the basic consideration of tree generation and the concept of encoding are discussed; Finally we introduce the class of trees with bounded degree.

Exhaustive generation of certain combinatorial objects has always been of great interest for computer scientists [80, 97, 117]. In general, generation of combinatorial structure problem consists in constructing all possible combinatorial structures of a particular kind in a certain order [60]. For example, a list of all trees with a given number of nodes $n$, may be used to test, analyze the complexity, prove the correctness of an algorithm, or data compression in data communication.

Designing algorithms to generate combinatorial objects has long fascinated mathematicians and computer scientists as well. Some of the earlier works on the interplay between mathematics and computer science have been devoted to combinatorial algorithms. Because of its many applications in science and engineering, the subject continues to receive much attention. In general
term, this branch of computer science can be defined as follows. Given a combinatorial object, design an efficient algorithm for generating all instances of the object. These combinatorial objects could be anything like graphs, trees, parentheses strings, permutations, combinations, partitions, derangements, and minimum spanning trees.

Because of the importance of trees, it is natural to study their properties, and as a result of the existence of numerous applications of trees, algorithms for the generation of trees have been extensively studied, and many ingenious generating algorithms for performing this task have been discovered [1, 37, 58, 60, 66, 78, 81, 106, 88, 89, 101, 104, 105, 36, 115, 118].

2.4.1 Generation Preliminaries

In most of trees generation algorithms, a tree is represented by integer or alphabet sequences, and then all possible sequences of this representation are generated. This operation is called tree encoding. Basically, the uniqueness of encoding, the length of the encoding, and the capability of constructing the tree from its representation, which is called decoding, are essential considerations in the design of the tree encoding schema [79]. By choosing a suitable codeword to represent the trees, we can design an efficient generation algorithm for these codewords.

It is particularly impressive to note the variation of representations of trees that are possible, such as the bit strings [83, 117], the weight sequences [80], the P-sequences [79], the \(\ell\)-sequences [79], the Ballot sequences [87], the Z-sequences [117], etc. In all cases, a 1-1 correspondence is established between the set of trees and the set of certain integer or alphabet sequences; then the set of trees is generated by generating the set of corresponding integer sequences.

A-order and B-order

Any generating algorithm is characterized by the ordering it imposes on the set of objects being generated and by its complexity. The most well-known orderings on trees are A-order and B-order [117]. The A-order definition uses global information concerning the tree nodes and appear to be a natural ordering of trees, whereas the B-order definition uses local information. Trees are prominently generated in local order, though natural order and other less useful orders have been addressed to a lesser extent. Up to the present time, the well known tree generating algorithms have utilized B-order, or some other ones, and only a few of them have used A-order. This is perhaps not so surprising if one notes that the generation of trees in A-order is indeed a very difficult task. Here we illustrate these orderings. Let \(\prec_A\) and \(\prec_B\) denote the A-order and B-order
orderings, respectively. Let $T_n$ be an arbitrary class of trees of size $n$. For $T, T' \in T_n$, the most commonly used linear orderings of trees may be defined as follows [105, 104, 117].

**Definition 1** Let $T$ and $T'$ be two trees in $T_n$, $T_i$ and $T'_i$ show the $i$th subtrees of $T$ and $T'$, respectively, and $k = \max\{\deg(T), \deg(T')\}$, we say that $T$ is less than $T'$ in $A$-order ($T \prec_A T'$), iff

- $|T| < |T'|$, or
- $|T| = |T'|$ and for some $1 \leq i \leq k$, $T_j =_A T'_j$ for all $j = 1, 2, \ldots, i - 1$ and $T_i \prec_A T'_i$.

where $|T|$ (size of $T$) is usually defined as the number of nodes in the tree $T$ and $\deg(T)$ is defined as the degree of the root of the tree $T$.

$A$-order is considered to be the most natural ordering on $T_n$. From the above definition, it is obvious that the natural order takes the size of a tree into account and hence a global knowledge of trees is compared. This is precisely what makes the generation of most of trees in the natural ordering non-trivial.

**Definition 2** Let $T$ and $T'$ be two trees in $T_n$ and $k = \max\{\deg(T), \deg(T')\}$, we say that $T$ is less than $T'$ in $B$-order ($T \prec_B T'$), iff

- $\deg(T) < \deg(T')$, or
- $\deg(T) = \deg(T')$ and for some $1 \leq i \leq k$, $T_j =_B T'_j$ for all $j = 1, 2, \ldots, i - 1$, and $T_i \prec_B T'_i$.

$B$-order is referred to as local order, because in this ordering, we compare the characters of the concurrent nodes (whether they are internal nodes or leaves). In other words, it takes a local view of trees being compared, and the task is easier. This explains why the generation of some trees such as binary trees or $t$-ary trees in a local ordering is popular. One of the advantages for listing trees in the natural order is that trees of small sizes are listed prior to trees of larger sizes. However, no such an advantage is observed in the local order. Furthermore, let $T, T' \in T_n$; it is possible to have $T \prec_A T'$ and at the same time $T' \prec_B T$. Hence, in general, the natural order and the local order list trees in slightly different orders.

**Tree Encoding**

It is well understood that algorithms for generating trees directly (linked form) are complicated and inefficient due to the need of changing the shape of tree [97]. It is indeed easier to manipulate
2.4. EXHAUSTIVE GENERATION OF TREES WITH BOUNDED DEGREE

an alphabet sequence which represent a class of trees, and process alphabet sequences instead of that class of trees as explained in [68]. In this way, trees are encoded as strings over a given alphabet and then these strings (called codeword) are generated. By choosing a suitable codeword to represent the trees, we can design an efficient generation algorithm for these codewords. Here, we explain the primaries of tree encoding on an arbitrary class of trees of size $n$, named $T_n$.

In general, an alphabet sequence can be defined as follows. Let $\mathbb{S}$ be the set of possible strings on an alphabet set $\sum = \{\delta_1, \delta_2, \ldots, \delta_r\}$, i.e., $\mathbb{S} = \{s|s \in \sum^*\}$, and $\mathbb{S}_n$ is the subset of $\mathbb{S}$ with all strings of length $n$, i.e., $\mathbb{S}_n = \{s|s \in \mathbb{S} \text{ and } |s| = n\}$. If string $A$ belongs to $\mathbb{S}_n$, then $A$ is shown as $A = (a_1, a_2, \ldots, a_n)$, such that each $a_i \in \sum$.

For defining an alphabet sequence corresponding to a tree $T \in T_n$, first an alphabet set $\sum$ (letter or integer) is considered and each node of the tree is labeled with an element of $\sum$ with regard to a specific rule (notice that we speak about labeling only for notational convenience; it is naturally possible to distinguish internal node from external ones without having any label), then the tree is traversed with one traversal procedure (preorder, inorder, or postorder) and each node label is listed in this traversal. The resulting sequence is the corresponding sequence of tree. The length of the sequence is $\ell$ and it is usually a factor of the number $n$ of the nodes of $T$. This function is called tree encoding and the sequence generated by it is called codeword, code sequence, tree sequence, or simply encoding. Let $\sum$ and $\mathbb{S}_n$ be defined as above, then the encoding function is a bijection

$$\text{encoding} : T_n \rightarrow \mathbb{S}_n.$$  

The inverse function of encoding is called decoding, and by employing it, we can obtain a tree $T \in T_n$ corresponding to each code sequence. This function is also a bijection:

$$\text{decoding} : \mathbb{S}_n \rightarrow T_n.$$  

A tree sequence $A \in \mathbb{S}_n$ will be called feasible if there is a tree $T \in T_n$ such that $A = \text{encoding}(T)$.

In fact, in encoding or decoding processes, we established a 1-1 correspondence between $T_n$ trees and tree sequences. Once the correspondence is established, an algorithm can be presented to generate all tree sequences. It should be noted that we can also define an ordering for the set of code sequences $\mathbb{S}_n$. Two such ordering are lexicographic ordering and minimal change ordering [117, 88].
For two strings $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, with $A$ and $B \in S_n$, the lexicographic ordering (lexicographical order) $\prec_{\text{lex}}$ or $\prec_{\ell}$ on $S_n$ is defined for $A$ and $B$ by the following relation:

$$A \prec_{\text{lex}} B \iff \exists j \ (1 \leq j \leq n) \text{ such that } a_1 = b_1, a_2 = b_2, \ldots, a_{j-1} = b_{j-1} \text{ and } a_j < b_j.$$ 

### Ranking and Unranking Algorithms

Besides the generation algorithm for trees, ranking and unranking algorithms are also important in the concept of tree generation [88, 36, 117]. Let us consider an arbitrary class of trees of size $n$ ($n$ nodes) showed by $T_n$, the elements of this set can be listed based on any defined ordering such as $A$-order or $B$-order. By having $T_n$ and an ordering (e.g. A-order or B-order), the “position” of tree $T$ in $T_n$ is called rank, the rank function determines the rank of $T$; the inverse operation of ranking is unranking, for a position $r$, the unrank function gives the tree $T$ corresponding to this position.

Recall that, the rank function determines the rank of a given tree (i.e., the position of the tree) with respect to the ordering $\prec$. In other words, the rank of a tree is the number of trees that precede this tree in the order $\prec$. Therefore, the rank function will be a bijection:

$$\text{rank} : (T_n, \prec) \rightarrow \{1, 2, \ldots, |T_n|\},$$

and for a tree $T_i \in T_n$, we have:

$$\text{rank}(T_i) = i.$$

A rank function defines a total ordering on the elements of $T_n$, by the following relation:

$$\forall T_i, T_j \in T_n, \ T_i \prec T_j \iff \text{rank}(T_i) < \text{rank}(T_j),$$

Conversely, there is a unique rank function associated with any total ordering defined on $T_n$.

If $\text{rank}$ is a ranking function defined on $T_n$, then there is a unique unranking function associated with the function rank. The function unrank is also a bijection:

$$\text{unrank} : \{1, 2, \ldots, |T_n|\} \rightarrow (T_n, \prec),$$

and for any $i \in \{1, 2, \ldots, |T_n|\}$, we have:
unrank(i) = Ti.

Unrank is the inverse function of the function rank, meaning that if T ∈ Tn:

rank(T) = i ⇔ unrank(i) = T.

Efficient ranking and unranking functions have several potential uses. We mention some of them now. One application is the generation of a “random” tree from the set Tn. This can be done easily by generating a random integer i ∈ {1, 2, . . . , |Tn|}, and then unranking on i. This algorithm ensures that every element of Tn is chosen with equal probability of \( \frac{1}{|T_n|} \), assuming that the random number generator being used is unbiased.

Another use of ranking and unranking is in storing trees in the computer. Instead of storing a tree, which could be complicated, an alternative would be to simply store its rank, which of course is just an integer. If the tree is needed at any time, then it can be recovered by using the unranking algorithm. Also, for example, in traditional tree compression algorithm for encoding the tree to code sequence and decoding the code sequence back to a tree, the ranking and unranking algorithms can be used.

It is particularly impressive to note the variation of representations of trees that are possible, such as the bit strings [83, 117], the weight sequences [80], the P-sequences [79], the \( \ell \)-sequences [79], the Ballot sequences [87], the Z-sequences [117], and etc. In all cases, a 1-1 correspondence is established between the set of trees and the set of certain integer or alphabet sequences; then the set of trees is generated by generating the set of corresponding integer sequences.

Many papers have been published earlier in the literature for generating different classes of trees. For example we can mention the generation of binary trees in [79, 105, 114], k-ary trees in [88, 37, 59, 115, 58, 54, 69, 36], rooted trees in [72, 24, 111], trees with \( n \) nodes and \( m \) leaves in [78], neuronal trees in [81, 104], and AVL trees in [66]. On the other hand, many papers have thoroughly investigated basic combinatorial features of chemical trees [98, 48, 47, 35, 28, 65, 112].

More related to our work, in [52] a coding for chemical trees without the generation algorithm, and in [13] the enumeration of chemical trees and in [42, 95] the enumeration of tree-like chemical graphs have been presented. Hendrickson and Parks in [53] investigated the enumeration and the generation of carbon skeletons which can have cycles and are not necessarily trees. The work most related to our research is an algorithm for the generation of certain classes of trees such as chemical
trees in [14] with no ranking or unranking algorithm. In that paper, all chemical trees with \( n \) nodes are generated from the complete set of chemical trees with \( n-1 \) nodes, the redundant generations are possible and they needed to minimize the possible redundancy.

The problem of enumeration of ordered trees were also studied in [116] and the generation of different ordered trees (with no bounds on the degrees of the nodes) were studied in [117]. In [119], a generation algorithm with constant average delay time but with no ranking or unranking algorithms was given for all unrooted trees of \( n \) nodes and a diameter at least \( d \) such that the degree of each vertex with distance \( k \) from the center of the tree is bounded by a given function. In [113] all unrooted unlabeled trees have been generated in constant average time with no ranking or unranking algorithms. Nakano and Uno in [73] gave an algorithm to generate all rooted unordered trees with exactly \( n \) nodes and diameter \( d \) in constant delay time. Up to now, to our knowledge, no efficient generation, ranking or unranking algorithms are known for either ‘chemical trees’ or ‘ordered trees with bounded degree’.

### 2.4.2 Trees with Bounded Degree

Studying combinatorial properties of restricted graphs or graphs with configurations has many applications in various fields such as machine learning and chemoinformatics. Studying combinatorial properties of restricted trees and outerplanar graphs (e.g. ordered trees with bounded degree) can be used for many purposes including virtual exploration of chemical universe, reconstruction of molecular structures from their signatures, and the inference of structures of chemical compounds [119, 95, 42, 48, 98, 52, 13].

In Chapter 7, we study the generation, ranking and unranking of unlabeled ordered trees whose nodes have maximum degree \( \Delta \), denoted by \( T^\Delta \) trees, we also use \( T^\Delta_n \) to denote the class of \( T^\Delta \) trees with \( n \) nodes. Chemical trees are the most similar trees to \( T^\Delta \) trees. Chemical trees are the graph representations of alkanes, or more precisely, the carbon atom skeleton of the molecules of alkanes [98, 48, 47, 35, 28, 65].

The alkane molecular family is partitioned into classes of homologous molecules, that is molecules with the same numbers of carbonium and hydrogen atoms; the \( n^{th} \) class of alkane molecular family is characterized by the formula \( C_n H_{2n+2} \), \( n = 1, 2, ... \) [13] with the same numbers of carbonium and hydrogen atoms. They are usually represented by indicating the carbonium atoms and their links, omitting to represent hydrogen atoms [13], therefore, all the nodes would have the same label; carbon (i.e., the tree is unlabeled), as shown in Figure 2.10 for \( n = 3 \) and \( n = 4 \). A chemical
2.5 Summery

In this chapter, some basic concepts of important binary search trees [32, 57, 110, 29, 91], external memory model [2] and cache-oblivious model [82], modern data structures for these two models [12, 19, 20, 21, 26, 75], and the concept of exhaustive generation of trees and trees with bounded degree [80, 97, 117] were presented.
Chapter 3

Core Partitioning Scheme

We propose a general method to store the nodes of balanced search trees and obtain provably good space-efficient external-memory/cache-oblivious data structures. It hinges on the decomposition of a balanced search tree into a set of disjoint cores: a core is a complete balanced binary tree (of height $h$ and with $2^h - 1$ nodes) that appears as a portion of the balanced tree. Our method is not invasive, as it does not change the original algorithms. It just requires an efficient post-processing after each update to maintain the cores. The nodes of a core are stored in a chunk of consecutive memory cells. Hence, the core partition adds a memory layout for the nodes of a balanced tree but does not interfere with the original algorithms for the tree.

In this chapter, in Section 3.1, we introduce the basic idea of cores in binary search trees and its preliminaries, we define the core partition in Section 3.2, then we show that it can be applied to weight-balanced trees in Section 3.3. After that, we discuss how to obtain space-efficient external-memory/cache-oblivious search trees. We then prove that the core partition can be applied to AVL trees in Section 3.4, and prove a lower bound on the amortized cost of updates.

3.1 Core Partitioning Preliminaries

For a parameter $h^*$ that depends on the chosen type of balanced tree, our recursive scheme requires that the first $h^*$ levels of nodes in the given balanced tree are full, thus they form a core. It conceptually removes these nodes and applies recursively this process to the resulting bottom subtrees. The recursion ends when the subtree size is below a threshold $r^*$ to be specified. As a result, the given balanced tree is decomposed into cores, which are central to our findings. We obtain a core partition when the cores found along any root-to-leaf path of the balanced tree are
of exponentially decreasing size, with \(O(1)\) number of them being of size smaller than \(r^*\).

As a case study, we consider AVL trees [1] and weight-balanced trees [74]. We can obtain a space-efficient external-memory/cache-oblivious version of them with little programming effort. We prove that both types of balanced trees of \(n\) nodes admit a core partition that takes \(O(n/B)\) blocks of memory of size \(B\), using the external-memory/cache-oblivious models [2, 82]. Searching a key requires \(O(\log B n)\) block transfers and \(O(\log n)\) comparisons in the external memory model. The cost becomes \(O(\log B n + \log(\frac{\log(B+1)}{\log(\log n+1)}))\) block transfers (which can be reduced to \(O(\log B n)\) block transfers) and \(O(\log n)\) comparisons in the cache-oblivious model. Interestingly, the core partition for weight-balanced trees can be maintained with amortized \(O(\log B n)\) block transfers per update whereas maintaining the core partition for AVL trees requires more than a poly-logarithmic amortized cost.

We think that this hidden property holds also for other balanced trees, thus making our method of independent interest. This is an example of the benefit of our technique as we can reuse the vast knowledge on balanced search trees to provide a repertoire of space-efficient external-memory and cache-oblivious data structures that are competitive with modern data structures that are purposely designed for these models (e.g. [19, 20, 21, 26]). This opens a number of possibilities that are known for modern search data structures but unknown for several previous balanced trees.

- I/O efficiency and cache-obliviousness can be achieved for a tree of \(n\) nodes, as explained in Section 3.3.4.
- Dynamic memory management can be easily handled by allocating a common contiguous memory chunk for all the keys of each core, since each core contains a number of keys that is a power of two (minus one). This alleviates memory fragmentation.
- The pointers requires a total of \(O(n)\) bits of storage rather than \(\Omega(n \log n)\) for a tree of \(n\) nodes. For example, each core can be internally stored using an implicit heap layout.
- Efficient dynamization of static data structures can be obtained by storing a static data structure inside each core. The data structure is completely rebuilt when the core changes.

We emphasize that the above features just require the original algorithms described for the given balanced tree, thus offering simultaneously many features that have been introduced on different search trees. What we add is the maintenance of the core partition of the nodes, and the algorithmic challenge is how to maintain it efficiently. When performing the updates, we proceed as
usual, except that we perform a post-processing: we take the topmost core that should be changed because of the update, and recompute the partition from it in a greedy fashion.

When comparing our results to previous work, we observe that it is folklore to prove that cores can be found in some data structures as mentioned in Chapter 2 but they never has been used before in the literature to make classic data structures efficient in external-memory/cache-oblivious models. We think that the contribution of our work is to show how to exploit the core partition to turn some existing balanced search trees into competitive external-memory/cache-oblivious data structures that have guaranteed bounds, using a general technique.

We adopt the external memory model [2] introduced in Chapter 2 to evaluate the I/O complexity, where $B$ is the block size of the data transfers between main and external memory, and the I/O complexity accounts for the number of block transfers performed during the computation. We also adopt the cache-oblivious model [82] presented in Chapter 2 to evaluate the I/O complexity, here called cache complexity and still expressed as the number of block transfers of size $B$. Note that $B$ is now an unknown parameter for the block size and a cache-oblivious algorithm is completely unaware of the value of $B$ used by the underlying system: this is a strength as it can thus show good performances on a multilevel memory hierarchy without knowing the cache size or the size of the block transfer.

### 3.2 Core Partitioning Scheme

For an arbitrary binary tree with $n$ nodes, the level of a node is the number of nodes above including itself (the root is on level 1), we say that level $i$ in $T$ is full, where $1 \leq i \leq H$, if it contains all the $2^{i-1}$ nodes, we adopt the standard terminology [57], where the height $H$ is the number of levels of nodes (so $H = 1$ for the tree with a single node), and the size is the number of nodes.

#### 3.2.1 Core Partitioning

We say that a binary tree has a core of height $h^*$, if its topmost $h^*$ levels form a complete balanced binary tree. We are interested in the families of binary search trees for which each subtree has a core of guaranteed height.

Consider a binary search tree $T$ with $n$ nodes and any two given integer parameters $h^* \geq 1$ and $r^* \geq 1$, such that each nonempty subtree of $T$ of size larger than $r^*$ has a core of size $h^*$, where $h^*$ is a function of the subtree size or height and $r^*$ is a function of $n$ or $B$. The recursive scheme
Figure 3.1: Decomposition of a binary search tree into its cores.

consists of the following greedy steps.

1. Conceptually remove the topmost core of height $h^*$ (made up of the topmost $h^*$ levels), which is a complete balanced binary tree of $2^{h^*} - 1$ nodes.

2. Recursively perform the core removal of the bottom subtrees thus obtained, where each of the bottom subtrees can potentially have different height or size.

3. Stop the recursion for a (possibly empty) subtree when its size is $\leq r^*$.

The case for $h^* = 1$ and $r^* = 1$ returns the trivial partition of the tree $T$ into its individual nodes and is of little interest. But other choices of $h^*$ and $r^*$ are more interesting to investigate: a binary search tree $T$ can be seen as conceptually decomposed into a collection of complete balanced binary trees, i.e., the cores, where each core is the top tree that is obtained from the recursive scheme applied to its subtree, plus the subtrees of size $\leq r^*$. Two cores are linked together if and only if there is one node in one of the two cores that is linked to a node in the other core, where one of the two nodes is the root of the core and the other is a leaf of the other core. Figure 3.1 illustrate core partitioning on a small binary search tree.

In the following, when we consider any root-to-leaf path, we let $C_1, C_2, \ldots, C_{t-1}, B_t$ be the subtrees thus traversed for $n > 1$: here $C_1$ is the core containing the root of the tree, $B_t$ is the (possibly empty) subtree of size $\leq r^*$ at the end of the path, and $C_2, \ldots, C_{t-1}$ are the cores traversed when going from $C_1$ downward to $B_t$. We say that core $C_i$ is at level $i$ to indicate that the path from the root of the tree to any descendant of $C_i$ (nodes in $C_i$ included) must traverse $C_1, C_2, \ldots, C_i$. We also denote by $h_i^*$ the height of $C_i$, namely, $|C_i| = 2^{h_i^*} - 1$.

**Definition 3 (core partition)** We say that our recursive scheme with parameters $h^*$ and $r^*$ is a (successful) core partition if both conditions below are satisfied.

- [continued in the next page]
3.2. CORE PARTITIONING SCHEME

1. There exists a positive constant $\gamma < 1$ such that for any sequence $C_1, C_2, \ldots, C_{t-1}, B_t$ traversed by a root-to-leaf path, the cores are of exponentially decreasing size in $\gamma$, namely, there is an integer constant $c \geq 1$ such that $h^*_i \leq \gamma h^*_i - c$ for $c < i \leq t - 1$.

2. For any sequence $C_1, C_2, \ldots, C_{t-1}, B_t$ traversed along a root-to-leaf path, there are $O(1)$ cores $C_i$ of small size $|C_i| < r^*$.

Our definition resembles what happens in van Emde Boas trees [20, 107, 108] and exponential trees [9, 19] in that the cores found along a root-to-leaf path have exponentially decreasing size, except a constant number of them.

Lemma 1 For any binary tree of size $n$, a core partition with parameters $h^*$ and $r^*$ correctly terminates producing subtrees of size $\leq r^*$ and $O(n/r^*)$ cores, with $O\left(\log\left(\log(n+1) \over \log(r^*+1)\right)\right)$ cores traversed in any root-to-leaf path.

Proof: Since $h^* \geq 1$ by definition of our recursive scheme, the latter eventually terminates when the subtree size is $\leq r^*$. Also, there are overall $O(n/r^*)$ cores generated by the scheme as we prove next. We observe that there exist at most $n/r^* + 1$ cores $C$ satisfying the condition $|C| \geq r^*$ since the sum of their sizes cannot exceed $n$. Hence we have to show that there exist $O(n/r^*)$ cores $C$ of size $|C| < r^*$. Consider the cores $C_1, C_2, \ldots, C_{t-1}$ traversed in any root-to-leaf path where there are some $C_i$’s such that $|C_i| < r^*$. Recall that each such $C_i$ is obtained by a subtree of size $\geq r^* + 1$ by our recursive scheme. Moreover, there can be only $O(1)$ such $C_i$’s by Definition 3.2. Overall, the number of these cores is $O(n/r^*)$ because there are $O(n/r^*)$ corresponding subtrees of size $\geq r^* + 1$ that are disjoint and, by Definition 3.2, $O(n/r^*)$ subtrees that are not disjoint.

As for $t$, observe that $h^*_i \leq \gamma^{i-kc} \log(n+1)$ holds when using the disequality of Definition 3.1 by induction to prove that $h^*_i \leq \gamma^k h^*_{i-kc}$, where $k$ is the largest integer such that $i - kc \geq 1$, and the fact that $h^*_{i-kc} \leq \log(n+1)$ as $|C_{i-kc}| = 2^{h^*_{i-kc}} - 1 \leq n$. Let $j$ be the largest $i$ such that $|C_i| \geq r^*$. Observe that $h^*_j \geq \log(r^* + 1)$, and that $t = j + O(1)$ by Definition 3.2. Hence $\gamma^{j-kc} \log(n+1) \geq \log(r^* + 1)$, which implies that $j = O(\log(\log(n+1)/\log(r^* + 1)))$, and so does $t$. \hfill \Box

Corollary 1 For $r^* \geq \log n$, the total space needed for all the pointers of the core partition is $O(n)$ bits.

\footnote{We will show that $\gamma = 2/3$ and $c = 2$ for AVL trees, and $\gamma = (\log_2/\alpha(1 - \alpha) + 1)$ and $c = 1$ for weight-balanced $BB[\alpha]$.}
Proof: The total number of cores is $O(n/r^*)$ as proved above, therefore, the total space needed for the pointers (pointing to the cores) will be upper bounded by: $O(n/r^*) \times \log n = O(n)$ bits. □

As mentioned, the fact the all levels are full in a core $C$ can be exploited by the implicit van Emde Boas (vEB) layout [20, 26, 82], which compactly stores $C$ pointerless as an array of $|C|$ elements. These elements are the keys in the nodes of $C$, so that it takes $O(\log_B |C|)$ block transfers to implicitly traverse $C$ during a search path [20]. Since the cores in that path are of nonincreasing exponential size, the total cost is $O(\log_B)$ block transfers.

### 3.2.2 Maintaining the Core Partition

A natural question is how to handle updates, namely, insertions and deletions. We need a basic primitive to maintain the cores, which uses the following simple observation. Given a node $z$ that is the root of the topmost core that changes size or content, observe that locally applying the core partition scheme to $z$ and its descendants does not change the global core partition obtained from the root of the whole tree. We exploit this locality to update the core partition of a binary search tree by means of the following greedy algorithm for a node $u$.

- **repartition**($u$):
  1. rebuild the core $C$ that contains $u$, and
  2. find $v_1, \ldots, v_k$, the topmost descendants of $u$ that are not in $C$, and locally recompute the core partition for each node $v_i$ if it is needed ($i = 1, \ldots, k$).

We run the above primitive as a post-processing step after each insertion. We proceed as usual by inserting a new node (typically a leaf) $f$ and finding its ancestor $v$ (if it exists) that has to be restructured. We also find the topmost ancestor $w$ (if it exists) of $f$ such that $w$ is the root of a core that changes size because of the insertion of $f$. If neither $v$ nor $w$ exist, return; otherwise, let $u$ be the topmost between $v$ and $w$ (as shown in Figure 3.2), and perform **repartition**($u$).

As for deletions, if the physical deletion is actually made, we proceed as in the insertion, locating the topmost node $u$ and performing **repartition**($u$). Another possibility is to avoid to use **repartition**. We simply mark the searched key as logically deleted, and remove that mark if the key is inserted again. We periodically rebuild the tree when the number of these marked keys is a constant fraction of the total number of keys. We will use both approaches.

When analyzing the cost of **repartition**($u$), we will focus on the three main events listed below.
3.2. CORE PARTITIONING SCHEME

$u = \bullet w$

$u = \bullet v$

$\bullet f$

$\bullet f$

Figure 3.2: Left: when $w$ is higher than $v$, so $u = w$. Right: when $w$ is lower than $v$, so $u = v$.

- **core resize**: if $w$ exists and $u = w$, this accounts for (1) and (2) in $\text{repartition}(u)$, with a cost proportional to the size of the full subtree rooted at the core’s root.

- **core rescan**: if $v$ exists and $u = v$, this accounts for (1) in $\text{repartition}(u)$, with a cost proportional to the size of the core containing the critical node $v$.

- **subtree rescan**: if $v$ exists and $u = v$, this accounts for (2) in $\text{repartition}(u)$, with a cost proportional to the size of the subtree rooted at the critical node $v$.

Note that **core resize** only occurs when $u = w$ (Figure 3.2: Left) while **core rescan** and **subtree rescan** occur when $u = v$ (Figure 3.2: Right). Also the case $u = w = v$ is feasible and all the three events happen in this case: however **core resize** is chosen as representative since its cost dominates that of **core rescan** plus **subtree rescan**. If logical deletions are performed, there can be actually a fourth event. It happens because of rebuilding the binary tree when the number of the deleted keys is a constant fraction of the total number of keys, whose amortized complexity can be analyzed in a traditional way, and thus it is not discussed here (or put into another way, the inserted keys pay also for their possible deletion in the future).

**Lemma 2** When $\text{repartition}(u)$ is applied to a node $u$, let $g$ be the size of the core containing $u$ (if **core rescan** occurs in case $u = v$) and $s$ be either the size of the subtree rooted at $u$ (if **subtree rescan** occurs in case $u = v$) or the size of the subtree rooted at the core’s root (if **core resize** occurs in case $u = w$). Then, the cost is $O(g + s)$ time and $O((g + s)/\min\{r^*, B\})$ block transfers, where $B$ is the block size.

*Proof*: An $O(g + s)$-time algorithm can rebuild the core partition as required by $\text{repartition}(u)$. Moreover, the number of cores is $O(s/r^*)$ by Lemma 1, and scanning them takes so many block transfers plus $O(g/B + s/B)$. \qed
3.3 Weight-Balanced Trees

Recall from the definition of weight-balanced trees given in Chapter 2, for a binary tree, the weight is the number of null nodes (null pointers), which is equivalently the number of nodes (i.e., the size) plus one. The weight of a node \( u \) is denoted by \( w(u) \) and its balance \( \beta(u) = w(u.l)/w(u) \) is the ratio between the weight of \( u \)'s left child and \( u \)'s weight (note that \( w(null) = 1 \) by definition of weight), \( u.l \) and \( u.r \) denote the left child and the right child of \( u \), respectively.

For a parameter \( \alpha \), where \( 0 < \alpha \leq 1 \), a weight-balanced tree (a.k.a. BB\([\alpha] \)-tree) is a binary search tree where each node \( u \) satisfies \( \alpha \leq \beta(u) \leq 1 - \alpha \), which is equivalent to say that \( \alpha \cdot w(u) \leq w(u.l), w(u.l) \leq (1 - \alpha) \cdot w(u) \) for each node \( u \) and its two children \( u.l \) and \( u.r \).

For example, the tree shown in Figure 2.4 is a BB\([\alpha] \)-tree for \( \alpha = 2/7 \) while it is not for \( \alpha = 1/3 \). As observed by the inventors Nievergelt and Reingold [74], a node of weight 3 should have one child of weight 1, so they assume that \( 0 < \alpha \leq 1/3 \). Moreover, Blum and Mehlhorn [25] show that rebalancing a BB\([\alpha] \)-tree with rotations can be done when \( 2/11 < \alpha \leq 1 - \sqrt{2}/2 = 0.2928 \ldots \). When \( \alpha \) is strictly inside this interval, they show that there exists \( \delta > 0 \) depending on \( \alpha \) such that an unbalanced node \( u \) has balance factor \( (1 + \delta)\alpha \leq \beta(u) \leq 1 - (1 + \delta)\alpha \) after its balance is restored using rotations. Overmars [77, Sect.4.2] shows that rebalancing can be also done with partial rebuilding, and this only requires \( 0 < \alpha < 1/2 \) and obtains a value of \( \beta(u) \) close to 1/2 after restoring the balance of \( u \). In both cases, the two properties are important in the amortized complexity for the following reason, as proved in [25, 77].

**Lemma 3** For weight-balanced trees, given a node \( u \), the number of updates between two consecutive rebalancing operations on \( u \) is \( \Omega(w(u)) \) [25, 77].

### 3.3.1 Cores in Weight-Balanced Trees

The height of a BB\([\alpha] \)-tree of size \( n \) is \( H \leq \log_{1/(1-\alpha)}(n+1) \). Indeed, its root \( r \) has weight \( w(r) = n+1 \) and the deepest leaf \( f \) has weight \( w(f) = 2 \). Along the path from \( r \) to \( f \), the weight of each node is at most \( 1 - \alpha \) times the weight of its parent. Hence, by a simple induction, we have that \( 2 = w(f) \leq w(r) \cdot (1 - \alpha)^{H-1} = (n+1) \cdot (1 - \alpha)^{H-1} \). Thus, \( H \leq \log_{1/(1-\alpha)}(n+1)/2 + 1 < \log_{1/(1-\alpha)}(n+1) \) as \( 1/(1 - \alpha) < 2 \). For a simplified notation, we ignore roundings when using \( \alpha \) and logarithms in this section. We use the following facts to obtain cores (Section 3.2.1).
Fact 1. For a $BB[\alpha]$-tree of $n$ nodes, the nodes on its topmost $\log_{1/\alpha}(n+1)$ levels form a complete balanced binary tree.

Proof: Consider a shortest path from the root $r$ of a $BB[\alpha]$-tree of $n$ nodes to a null (i.e., external) node $x$. Let $\ell$ be the number of nodes (including $x$ itself) along this path: since $w(r) = n + 1$ and $w(x) = 1$, we obtain that $x$ should have weight at least $(n+1)\alpha^{\ell-1}$ by a simple induction on $\ell$. Hence $(n+1)\alpha^{\ell-1} \leq w(x) = 1$. Thus the topmost $\ell - 1 \geq \log_{1/\alpha}(n+1)$ levels do not contain null nodes, and form a complete balanced binary tree.

Remark 1. As for the core partition and its dynamic maintenance, we set $h^* = \log_{2/\alpha}(n+1)$ and observe that it forms a core by Fact 1 as $h^* \leq \log_{1/\alpha}(n+1)$ (Our choice of $h^*$ will be clear when discussing the amortized analysis in Section 3.3.2). We also guarantee that $h^* \geq 1$, which means $n \geq 2/\alpha - 1$, by fixing $r^* \geq 2/\alpha - 1$. In this way, when $n \geq r^*$, a core of height $h^*$ surely exists by Fact 1, and when $n < r^*$ we stop the recursion on the subtree (see Section 3.2.1). The term $f(\alpha) = 2/\alpha - 1$ is a decreasing function for increasing $\alpha$, where $0 < \alpha < 1/2$, and it tends to $+\infty$ for $\alpha \to 0$. If we restrict to the range $2/11 < \alpha < 1/2$, we cover the interesting cases in the literature, and we have that $10 > f(\alpha) > 3$. Hence, it is always safe to choose $r^* \geq 10$ for $2/11 < \alpha < 1/2$.

We now show that we obtain a core partition (Definition 3) using the scheme described in Section 3.2.

Lemma 4. For any $BB[\alpha]$-tree of size $n$ with $2/11 < \alpha < 1/2$, the scheme of Section 3.2 with $h^* = \log_{2/\alpha}(n+1)$ and $r^* \geq 10$ successfully creates a core partition with $\gamma = (\log_{2/\alpha}(1-\alpha)+1) < 1$ and $c = 1$, where a core $C_i$ at level $i$ has size $|C_i| < (n+1)\frac{1}{1-2\gamma}$.

Proof: We prove that the conditions of Definition 3 are met, using the following notation. Consider a tree $T$ and its topmost core $C$ of height $h^*$. Also, consider a node $u$ in $T \setminus C$ such that $u$’s parent is in $C$. Let $T_u$ denote the subtree rooted at $u$ and $C_u$ be the topmost core of $T_u$, where we denote the height of $C_u$ by $h_u^*$ (Note that $C$ and $C_u$ are consecutive in any path from the root to $u$ or any of its descendants). We have that $\alpha^{h^*}(n+1) \leq |T_u| + 1 \leq (1-\alpha)^{h^*}(n+1)$ for the balance property of $BB[\alpha]$-trees.

The condition of Definition 3.1 is met as $h_u^* < \gamma h^*$ with $\gamma = (\log_{2/\alpha}(1-\alpha)+1) < 1$. Indeed, since $h_u^* = \log_{2/\alpha}(|T_u| + 1) \leq \log_{2/\alpha}((1-\alpha)^{h^*}(n+1))$, we can rewrite the latter disequality as $h_u^* \leq h^*\log_{2/\alpha}(1-\alpha) + \log_{2/\alpha}(n+1)$. Replacing the last addend by $h^*$, we obtain $h_u^* \leq h^*(\log_{2/\alpha}(1-\alpha) + 1)$, where $\gamma = (\log_{2/\alpha}(1-\alpha)+1) < 1$ as $1 - \alpha < 1 < 2/\alpha$. 

3.3. WEIGHT-BALANCED TREES
The condition of Definition 3.2 holds as, for any sequence of cores $C_1, C_2, \ldots, C_{t-1}, B_t$ traversed by a root-to-leaf path, there are $O(1)$ cores $C_i$’s of size $|C_i| < r^*$. To see why, we first observe that $|C_i| \leq |C_{i-1}|$ for $2 \leq i \leq t-1$ by construction. Thus, let us consider $C_{t-1}$. If its size is $\geq r^*$, we have nothing else to prove. Otherwise, observe that the subtree $T_{t-1}$ of which $C_{t-1}$ is the topmost core, has size $|T_{t-1}| \geq r^* + 1$ by the recursive scheme, and the height of $C_{t-1}$ is $h^*_t = \log_{2/\alpha}(|T_{t-1}| + 1) \geq \log_{2/\alpha}(r^* + 2)$. Hence, $(r^* + 2)^{1/\log(2/\alpha)} - 1 \leq 2^{h^*_t} - 1 = |C_{t-1}| < r^*$. Since $h^*_{t-2} > 2^{-1} \cdot h^*_t$ by Definition 3.1, an immediate induction on $j = 1, 2, \ldots$ gives that $|C_{t-1-j}| \geq |C_{t-1}|/\gamma$ by transitivity. Finding the smallest $j$ such that $((r^* + 2)^{1/\log(2/\alpha)} - 1)^{-j} \geq r^*$ gives an upper bound of $j + 1$ on the maximum number of cores having size $< r^*$ in $C_1, C_2, \ldots, C_{t-1}, B_t$. But $j = O(\log_{2/\alpha}(2/\alpha)) = O(1)$, thus proving the condition. Finally, the claim on the size of the core at level $i$ easily follows from the above discussion. \hfill \Box

### 3.3.2 Amortized Analysis for Repartitioning

We show that the size of a core is smaller than the size of its bottom subtrees. This is important to amortize the cost of core rescan.

**Fact 2** Consider a core $C$ in a $BB[\alpha]$-tree $T$ with $2/11 < \alpha < 1/2$ and parameters $h^* = \log_{2/\alpha}(n+1)$ and $r^* \geq 10$. Let $z$ be a node in $T \setminus C$ such that $z$’s parent is in $C$, and let $T_z$ be the subtree rooted in $z$. Then, $|C| \leq |T_z|$.

**Proof:** Let $n$ be the size of $T$, which is rooted at the topmost node of the core $C$. Recalling that $|C| = 2^{h^*} - 1$, it suffices to prove that $2^{h^*} \leq |T_z| + 1$. Note that $z$ is $h^*$ levels below the root of $T$. Hence, we have that $w(z) = |T_z| + 1 \geq \alpha^{h^*}(n + 1)$ by definition of balance in $BB[\alpha]$ trees. We show that $2^{h^*} \leq \alpha^{h^*}(n + 1)$ to prove our claim. By taking the logarithms, we obtain $h^* \leq h^* \log \alpha + \log(n + 1)$, namely, $h^*(1 - \log \alpha) \leq \log(n + 1)$. By replacing $h^*$ with $\log_{2/\alpha}(n + 1)$, we obtain the disequality $\log_{2/\alpha}(n + 1) \cdot (1 - \log \alpha) \leq \log(n + 1)$, which is true since $\log 2/\alpha = 1 - \log \alpha$. \hfill \Box

Now we show how to amortize the cost of repartition($u$) stated in Lemma 2, and focus on the three main events listed in Section 3.2.2. Let $T_u$ be the subtree rooted at $u$, $C_u$ be the core containing $u$, and $T$ be the subtree having $C_u$ as topmost core (so $T_u \subseteq T$).

- **core resize:** Let $n_0$ be the size of $T$ when the last core resize occurred for $C_u$ and $n_1$ be the size of $T$ for the current core resize of $C_u$. Since the size changed, the height changed
3.3. WEIGHT-BALANCED TREES

by 1 (increased or decreased), so \(|\log_{2/\alpha}(n_0 + 1) - \log_{2/\alpha}(n_1 + 1)| \geq 1\). This implies that \(|n_0 - n_1| = \Omega(|T|)\), and thus so many fresh update operations below \(u\) can cover the cost.

- **core rescan**: By Fact 2, the size of \(C_u\) is upper bounded by that of a subtree of \(T_u\), and so \(|C_u| \leq |T_u|\), which means that this cost is absorbed by **subtree rescan**.

- **subtree rescan**: By Lemma 3, we can charge the \(O(|T_u|)\) cost to \(\Omega(w(u))\) fresh update operations below \(u\) as \(w(u) = |T_u| + 1\).

From the discussion above, for each update operation, we can charge \(O(\log n)\) credits for the running time and \(O((\log n) / \min\{r^*, B\})\) credits for the cache complexity, with \(O(1)\) credits (respectively, \(O(1 / \min\{r^*, B\})\) credits) to be used for each ancestor as illustrated above. Therefore, we have the following result.

**Lemma 5** For any \(BB[\alpha]-tree\) of size \(n\) with \(2/11 < \alpha < 1/2\), its core partition with parameters \(h^* = \log_{2/\alpha}(n + 1)\) and \(r^* \geq 10\) can be dynamically maintained with an amortized cost of \(O(\log n)\) time and \(O((\log n) / \min\{r^*, B\})\) block transfers per update operation.

3.3.3 External-Memory Search Trees

Setting \(r^* = \max\{\log n, B\}\), we obtain a \(B\)-tree-like data structure for external memory [17]. More precisely, the complete balanced binary tree represented by each core \(C_i\) can be stored in blocks of size of multiples of \(B\), so that it takes \(O(1 + h^*_i / \log B)\) I/Os to traverse \(C_i\) (e.g. see [109]). Moreover, the sibling subtrees of size at most \(r^*\) for which the recursion stops, are packed together in a greedy fashion from left to right.

**Theorem 1** Given a core partition for a \(BB[\alpha]-tree\) of size \(n\) with \(2/11 < \alpha < 1/2\) and parameters \(h^* = \log_{2/\alpha}(n + 1)\) and \(r^* = \max\{\log n, B\}\), where \(B \geq 10\) is the block transfer size for the external memory, we can store its nodes in contiguous portions of memory of size of multiples of \(B\), so that \(O(n/B)\) blocks are occupied and any search path from the root to a node requires \(O(\log B n)\) I/Os and \(O(\log n)\) comparisons.

**Proof**: Since there are \(O(\log \log n)\) different sizes of memory chunks, each of length a power of two, it is not difficult to keep the \(n\) nodes in \(O(n)\) contiguous memory space. This guarantees that \(O(n/B)\) blocks are occupied for any \(B\). As for the search cost, \(O(\log n)\) comparisons derive from the standard analysis of \(BB[\alpha]-trees\). Consider the cores \(C_1, C_2, \ldots, C_{t-1}\) traversed in a root-to-leaf path of the \(BB[\alpha]-tree\), and let \(B_i\) be the (possibly empty) subtree of size at most \(r^*\) at the end of
the path. We just need to follow external-memory references when moving from one core to another
core or to \(B_t\). Thus the I/O complexity is \(O(1 + h^*_t / \log B)\) per core \(C_i\) plus \(O(1 + (\log(r^*)) / \log B)\)
I/O to access \(B_t\). This gives a total I/O cost of \(O(t + \sum_{i=1}^{t-1} h^*_i / \log B + (\log(r^*)) / \log B)\), which
can be upper bounded by Lemma 4 as

\[
O \left( \log \log_B n + (1 / \log B) \sum_{i=1}^{t-1} \frac{\gamma_i - 1}{\log 2/\alpha} \log(n + 1) + \max\{1, \log \log n / \log B\} \right) = O(\log_B n)
\]

because \(\sum_{i=1}^{t-1} \frac{\gamma_i - 1}{\log 2/\alpha} = O(1)\).

### 3.3.4 Cache-Oblivious Search Trees

We fix \(r^* = \log n\) and employ the following memory layout of the nodes. We store the subtrees of
size at most \(r^*\) in a contiguous memory chunk that can be scanned. We then store the complete
balanced binary tree inside each core \(C_i\) in a contiguous memory chunk using the van Emde Boas
layout \([82, 20, 26]\), so that it takes \(O(1 + h^*_i / \log B)\) block transfers to traverse \(C_i\) during a search
path. This suffices to obtain cache-oblivious bounds.

**Theorem 2** Given a core partition for a \(BB[\alpha]\)-tree of size \(n \geq 1024\) with \(2/11 < \alpha < 1/2\) and
parameters \(h^* = \log_{2/\alpha}(n + 1)\) and \(r^* = \log n\), we can use the memory layout where subtrees
and cores are each stored in a contiguous memory chunk, each core using the van Emde Boas
layout, so that \(O(n/B)\) blocks are occupied and any search path from the root to a node requires
\(O(\log_B n + \log(\frac{\log(B+1)}{\log(\log n+1)})\) block transfers and \(O(\log n)\) comparisons.

**Proof:** Since there are \(O(\log\log n)\) different sizes of memory chunks, each of length a power of
two, it is not difficult to keep the \(n\) nodes in \(O(n)\) contiguous memory space. This guarantees
that \(O(n/B)\) blocks are occupied for any \(B\). The bound of \(O(\log n)\) comparisons derives from
the \(BB[\alpha]\)-trees. As for the cache complexity, consider the cores \(C_1, C_2, \ldots, C_{t-1}\) traversed in a
root-to-leaf path of the \(BB[\alpha]\)-tree, and the subtree \(B_t\) of size at most \(r^* = \log n\) at the end of
the path. Let \(C_k\) be the smallest core among \(C_1, C_2, \ldots, C_{t-1}\) (i.e., largest \(k\)) such that \(|C_k| \geq B\)
and \(1 \leq k \leq t - 1\). Then, the cache complexity of traversing \(C_1, C_2, \ldots, C_k\) is \(O(\log_B n)\) as we
just saw before. The extra cost is given by traversing \(C_{k+1}, \ldots, C_{t-1}, B_t\), namely \(O(t - k)\) block
transfers. Note that \(|C_{k+1}| < B\) by definition of \(k\) and, hence, \(|T_{k+1}| \leq B^{O(1)}\) by our choice
of \(h^*\). Using Lemma 1 on \(T_{k+1}\), which is clearly a \(BB[\alpha]\)-tree with \(\leq B^{O(1)}\) nodes, we obtain
\(t - k = O(\log(\log(|T_{k+1}| + 1) / \log(r^* + 1))) = O(\log \frac{\log(B+1)}{\log(\log n+1)})\).
3.3.5 Space-Efficient Search Trees

We fix \( r^* = \log n \) and store each core \( C_i \) using the *implicit* van Emde Boas layout for its keys into an array without requiring internal pointers [82, 20, 26]. We only keep the pointers from the node in the last level of \( C_i \) to the roots of the “children” cores. We also store the keys of each small subtree of size \( n_0 \leq r^* \) in an array of \( n_0 \) entries. The resulting arrays for the cores and the small subtrees are stored, in decreasing order of size one after the other, in two large segments \( C \) and \( S \) of adjacent memory cells. The arrays for the cores are kept in \( C \) while the arrays for the small subtrees are kept in \( S \). Note that the wasted space is minimal in this way, since we have to store, for each size, how many arrays are of that size. Also, by Corollary 1, just \( O(n/\log n) \) pointers have to be explicitly stored, each using \( O(\log n) \) bits.

The \( O(\log_B n) \) Block Transfer Solution for Search Operation.

In Theorem 2, the \( \log(\log(\log(B+1)\log \log n + 1)) \) term in the number of block transfers for search operation can be avoided by keeping any core \( C \) and all its descendants in a contiguous portion of memory. This may increase the space but we will see how to handle the space increase.

Let \( T \) be a subtree with core \( C \) and \( T_1, T_2, \ldots, T_k \) be the topmost subtrees below \( C \), and let level of \( C \) (denoted by \( \text{lev}(C) \)) be the number of ‘cores’ above \( C \) (note that the level of a core is different from the level of a node defined in Chapter 2) and layer of \( C \) (denoted by \( \text{lay}(c) \)) be the maximum level of all cores of the entire tree minus \( \text{lev}(C) \). To ensure that \( C \) and all descendant nodes fit in a contiguous portion of memory, we assign sufficient space for \( C \) and its subtrees in a recursive manner. For a core \( C \) at layer \( i \) with subtree size \( n \) we assign \( \Theta(3^i n) \) space to \( C \) and all its descendants (i.e., at each layer, the size of the tree triples). Therefore, since the maximum level of a core in a given tree is \( O(\log \log n) \), this will increase the total space to \( O(n3^i \log n) = O(n(\log_2 n)^{\log 3}) \) (we will see later how to decrease this space to \( O(n) \)). More precisely, the new partitioning will be as follows. For \( 1 \leq i \leq k \) let \( n_i = |T_i| \) be the number of nodes in \( T_i \), and \( SP(T) \) denotes the space needed to store \( T \) in this new partitioning. For a subtrees \( T' \) less than \( r^* \), we set \( SP(T') = 3^{\text{lay}(T')}|T'| \).

We allocate a memory of size \( C \) to store the core \( C \) plus the space needed for each \( T_i \) (\( 1 \leq i \leq k \)) in a recursive manner which is equal to \( \sum_{i=1}^{k} SP(T_i) \) plus a free space which is equal to \( 2 \sum_{i=1}^{k} SP(T_i) + |C|(3^{\text{lay}(C)} - 1) \), let us denote this free space by spare-space. It can be easily observed that this approach preserves the assumption that each subtree of layer \( i \) and size \( n \) occupies \( \Theta(3^i n) \) space. During a sequence of insertions, let space-over-flow (SOF) denotes the
event that $T$ can not fit anymore in the space allocated before, in this case we release that space and we relocate a new space of size three times of the previous space from the spare-space of its parent core; then, if its parent has a new SOF because of this new memory allocation, we repeat this procedure until we reach an ancestor who does not have SOF or we reach the root, leading to triple the size of the entire data structure.

Using induction we prove that whenever SOF happens on $T$, the number of nodes in $T (|T|)$ has been doubled since the last time (or the last rebalancing). Let us suppose this is true for any subtree $T_i$. Let $n'$ be the size of $T$ when $T$ had the last rebalancing. During the sequence of insertions, some of the $T_i$'s have had one or more SOF until the spare-space of $T$ finishes, in the last event of a $T_i$'s SOF, when there is no more enough space in the spare-space of $T$, the sum of $|T_i|$s in which $T_i$ had a SOF and relocated the space, is at least $n'$ (the size of $T$ when it had the last rebalancing), so we had at least $n'$ fresh insertions, this completes the induction. Therefore, when a new SOF occurs, we have enough credits to rebalance the entire tree $T$, release its current space, and allocate the space twice as before from its parent core (or doubling the entire tree space).

**Corollary 2** The search time in this new partitioning is $O(\log_B n)$ in the cache-oblivious model.

**Lemma 6** The space needed for this partitioning can be reduced to $\Theta(n)$.

**Proof:** To decrease the space to $O(n)$, we use buckets of size $\Theta((\log n)^{\log 3})$ similar to Bender and his team’s buckets in [19]. We guarantee that the leaf nodes of the tree are buckets of size in between $(\log n)^{\log 3}$ and $2((\log n)^{\log 3})$, the total space to store these buckets will be $O(\frac{n}{(\log n)^{\log 3}} \times (\log n)^{\log 3}) = O(n)$. On the other hand this bucketing reduces the number of nodes in the main tree to $\frac{n}{(\log n)^{\log 3}}$ which leads us to decrease the total space needed for partitioning to:

$$O\left(\frac{n}{(\log n)^{\log 3}} \times (\log \frac{n}{(\log n)^{\log 3}})^{\log 3}\right) = O(n).$$

\[\Box\]

### 3.4 AVL Trees

The results described in Subsection 3.3.3 can be also obtained using AVL trees within the same bounds, as the core partition can be applied to them. However, maintaining the core partition is more expensive as we show at the end of this section.
3.4. AVL TREES

3.4.1 Cores in AVL Trees

We exploit the following folklore to define the cores in AVL trees.

**Fact 3** Consider an AVL tree of height $H$. Then, the nodes on its topmost $\lceil H/2 \rceil$ levels form a complete balanced binary tree.

*Proof*: By induction on $H$. For $H = 1$ and $H = 2$, the property trivially holds as there is a single node in the topmost $\lceil H/2 \rceil$ levels. For the inductive step on $H \geq 3$, we assume that the property is true for any AVL tree of height $< H$, and we then use this assumption to prove the statement for height $H$. We consider two cases for the given AVL tree $T$.

Let $H$ be even. Consider the left and right subtrees of $T$. Their height is at least $H - 2$, thus by the induction hypothesis, they are complete at least in the topmost $\lceil (H - 2)/2 \rceil = H/2 - 1$ levels. Thus, considering the additional level of the root of $T$, we have that $T$ is complete at least in the topmost $H/2 = \lceil H/2 \rceil$ levels.

Let now $H$ be odd. The height of the left and right subtrees of $T$ is at least $H - 2$, and by the induction hypothesis, they are complete at least in the topmost $\lceil (H - 2)/2 \rceil = (H - 1)/2$ levels. Thus, considering the additional level of the root of $T$, we have that $T$ is complete at least in the topmost $(H - 1)/2 + 1 = (H + 1)/2 = \lceil H/2 \rceil$ levels. □

We fix $h^* = \lceil \log \sqrt{n} \rceil$, where $n$ is the size of the AVL (sub)tree, and thus the top tree has size $2^{h^*} - 1 < 2\sqrt{n}$. We also fix $r^* = 1$ for the sake of discussion, but other choices can be done. Note that the choice of $h^* = \lceil \log \sqrt{n} \rceil$ guarantees that the top tree is a core.

**Fact 4** For any AVL tree of height $H$ with $n > 1$ nodes, the topmost $\lceil \log \sqrt{n} \rceil$ levels of nodes form a complete balanced binary tree.

*Proof*: The claim immediately follows from Fact 3: we have $\lceil \log \sqrt{n} \rceil = \lceil \frac{1}{2} \log n \rceil \leq \lceil \frac{1}{2} \log (n+1) \rceil \leq \lceil H/2 \rceil$ as $H \geq \log (n+1)$.

□

Resembling what happens in Section 3.3, we want to prove that the recursive scheme provides a core partition as stated in Definition 3. However, here we fix $c = 2$, meaning that the cores are exponentially decreasing by taking *every other core* in the root-to-leaf path, as shown next. (This is not true for $c = 1$, as it can be checked when the left subtree of the root is a complete balanced binary tree of height $H - 1$ and the right subtree is a Fibonacci tree of height $H - 2$.)
Lemma 7 For any AVL tree with $n > 1$ node, the recursive scheme of Section 3.2 with $h^* = \lceil \log \sqrt{n} \rceil$ and $r^* \geq 1$ successfully creates a core partition with $\gamma = 2/3$ and $c = 2$, where a core $C_i$ at level $i$ has size $|C_i| < (2\sqrt{n})^{(\frac{3}{2})^i(1-1/2)}$.

The proof of Lemma 7 relies on the following properties of cores in the AVL tree (see Definition 3.1).

Lemma 8 Let $C_1, C_2, \ldots, C_{t-1}$ be the cores traversed in any root-to-leaf path for $t > 1$. Then, the following properties hold:

- $|C_2| \leq |C_1| < 2\sqrt{n}$;
- $|C_i| < |C_{i-2}|^{\frac{3}{2}}$ for $|C_{i-2}| > 1$ and $3 \leq i \leq t - 1$.

Proof: We know that $|C_1| < 2\sqrt{n}$ by our choice of $h^* = \lceil \log \sqrt{n} \rceil$ when $n > 1$; if $C_2$ exists, it cannot have more descendants than $C_1$, so it is $|C_2| \leq |C_1|$ by construction. In general, note that $|C_i| \leq |C_{i-1}|$ for $3 \leq i \leq t - 1$ where $3 \leq i \leq t - 1$: Let $T_i$ denote the subtree rooted at the topmost node of $C_i$. As $|T_i| < |T_{i-1}|$, it follows that $|C_i| = 2^{\log \sqrt{|T_i|}} - 1 \leq 2^{\log \sqrt{|T_{i-1}|}} - 1 = |C_{i-1}|$.

We now prove that $|C_i| < |C_{i-2}|^{\frac{3}{2}}$ for $|C_{i-2}| > 1$ and $3 \leq i \leq t - 1$. Let $h^*_i = \lceil \log \sqrt{|T_i|} \rceil$ denote the height of core $C_i$, and $H_i$ denote the height of subtree $T_i$. First suppose that

$$h^*_{i-1} \geq h^*_i + 1 \quad (3.1)$$

holds. (We will show later how to deal when (3.1) does not hold.) Also, observe that our choice of $h^*_i$ and (the proof of) Fact 4 imply that

$$H_i \geq 2h^*_i - 1 \quad (3.2)$$

Since $H_{i-2} = h^*_{i-2} + h^*_{i-1} + H_i$, we can use (3.1) and (3.2) to bound the height of $T_{i-2}$ as

$$H_{i-2} \geq h^*_{i-2} + 3h^*_i \quad (3.3)$$

We are ready to prove that $|C_i| < |C_{i-2}|^{\frac{3}{2}}$ for $|C_{i-2}| > 1$. By contradiction, suppose $|C_i| \geq |C_{i-2}|^{\frac{3}{2}}$. This is equivalent to say that $2^{\log \sqrt{|T_i|}} - 1 \geq (2^{\log \sqrt{|T_{i-2}|}} - 1)^{\frac{3}{2}}$. Since $(x - 1)^{\frac{3}{2}} \geq x^{\frac{3}{2}} - 1$ for $x \geq 1$, we obtain that $2^{\log \sqrt{|T_i|}} \geq (2^{\log \sqrt{|T_{i-2}|}})^{\frac{3}{2}}$ and so $\log \sqrt{|T_i|} \geq \frac{2}{3} \times \log \sqrt{|T_{i-2}|}$. That is,

$$h^*_i \geq \frac{2}{3} \times \left\lceil \log \sqrt{|T_{i-2}|} \right\rceil \quad (3.4)$$
Recalling from [57, p.460] that \( H_{i-2} \leq 1.4404 \times \log(|T_{i-2}| + 2) - 0.3277 \), we obtain from (3.3) and (3.4) that \( 1.4404 \times \log(|T_{i-2}| + 2) - 0.3277 \geq 3h_i^* - 3h^*_i \geq 3 \times \log \sqrt{|T_{i-2}|} \geq 3 \times \log \sqrt{|C_{i-2}|} \). But we have a contradiction for \(|T_{i-2}| \geq 8\), since the inequality \( 1.4404 \times \log(|T_{i-2}| + 2) - 0.3277 \geq \frac{3}{2} \times \log |T_{i-2}| \) does not hold in this case. Hence, we can conclude that \(|C_i| < |C_{i-2}| \frac{3}{4}\) for \(|T_{i-2}| \geq 8\). As for \(|T_{i-2}| < 8\), there are a small finite number of cases to examine, so we directly prove for them that \(|C_i| < |C_{i-2}| \frac{3}{4}\) when \(|C_{i-2}| > 1\): we inspect them case by case as at the end of this proof.

It remains to discuss when the inequality in (3.1) does not hold (whereas \( h_{i-1}^* \geq h_i^* \) is always true). Its purpose is to guarantee that (3.3) holds. However, since \( H_{i-2} = h_{i-2}^* + H_{i-1} \), we observe that \( H_{i-1} \geq 3h_i^* \) also implies that (3.3) holds, and so we are done. Consequently, it suffices to discuss the case when \( h_{i-1}^* = h_i^* \) and \( H_{i-1} \leq 3h_i^* - 1\). In the following, we use \( h \) as the shorthand for both \( h_i^* \) and \( h_{i-1}^* \), since they are equal. Also, observe that \( H_{i-1} = h_{i-1}^* + H_i = h + H_i \geq 3h - 1 \) by (3.2), thus implying that \( H_{i-1} = 3h - 1 \). This gives a precise scenario: both \( C_{i-1} \) and \( C_i \) are of height \( h \), the height of \( T_{i-1} \) is \( 3h - 1 \), and the height of \( T_i \) is \( 2h - 1 \) (so it extends by further \( h - 1 \) levels below \( C_i \)).

We prove that it not possible to have \( h > 3 \) in this scenario. For the given \( h = \lceil \log \sqrt{x} \rceil \), we observe by a simple induction that the feasible range of values for \( x \) are \( 2^{2(h-1)} < x \leq 2^{2h} \). Hence, \( 2^{2(h-1)} < |T_i| \leq |T_{i-1}| \leq 2^{2h} \). Also, the latter two trees cannot have less nodes than the Fibonacci trees of their same height, \( |T_i| \geq F_{2h+1} - 1 \) and \( |T_{i-1}| \geq F_{3h+1} - 1 \), formulated in terms of Fibonacci numbers (recalling that the Fibonacci tree of height \( k \) has \( F_{k+2} - 1 \) nodes [57, p.460]).

We are now ready to state a necessary condition that excludes the cases for \( h > 3 \). The quantity \( 2^{2h} - (F_{3h+1} - 1) \) represents an upper bound on the number of nodes that can be added to \(|T_{i-1}|\) without increasing its height \( 3h - 1 \). Then \( T_i \) cannot contain too many nodes, namely, \(|T_i| \leq (F_{2h+1} - 1) + [2^{2h} - (F_{3h+1} - 1)] \): starting from the minimal number \( F_{2h+1} - 1 \) of nodes for its height \( 2h - 1 \), we cannot add more than \( 2^{2h} - (F_{3h+1} - 1) \) nodes since \( T_i \) is a subtree of \( T_{i-1} \). Also, \( 2^{2(h-1)} < |T_i| \) as previously discussed. Putting all together, we obtain that \( 2^{2(h-1)} < |T_i| \leq (F_{2h+1} - 1) + 2^{2h} - (F_{3h+1} - 1) \), producing the necessary condition \( 2^{2(h-1)} < (F_{2h+1} - 1) + 2^{2h} - (F_{3h+1} - 1) \), which can be equivalently stated as

\[
\frac{3}{4} \times 4^h - F_{3h+1} + F_{2h+1} > 0
\]  

(3.5)

Note that the condition in (3.5) is satisfied only when \( h \leq 3 \). There are just a small finite number of cases for \( h \leq 3 \), so we directly prove for them that \(|C_i| < |C_{i-2}| \frac{3}{4}\) for \(|C_{i-2}| > 1\) as follows.
here by direct case inspection that \(|C_i| < |C_{i-2}|^{\frac{5}{3}}\) when \(|C_{i-2}| > 1\) and \(|T_{i-2}| < 8\). Note that \(|C_{i-2}|\) is a power of two minus 1, so the latter conditions and the choice of \(h_{i-2}^*\) imply that \(|C_{i-2}| = 3\). Given this, the only feasible choices are \(|T_{i-2}| \in [5 \ldots 7]\), and thus we have a very small number of feasible situations. Indeed, \(|T_{i-1}| \leq |T_{i-2}| - |C_{i-2}| \leq 4\). Hence, \(h_{i-1}^* = 1\) and \(|C_{i-1}| = 1\). This immediately implies that \(|C_i| \leq |C_{i-1}| = 1 < 3^{\frac{2}{3}} = |C_{i-2}|^{\frac{5}{3}}\), thus proving the first claim.

We also prove here by direct case inspection for \(1 \leq h \leq 3\) that \(|C_i| < |C_{i-2}|^{\frac{5}{3}}\) for \(|C_{i-2}| > 1\) when \(h = h_{i-1}^* = h\) and \(H_{i-1} = 3h - 1\). Recall that \(|C_{i-2}| > 1\) is equivalent to \(|C_{i-2}| \geq 3\) and so \(h_{i-2}^* \geq 2\), thus proving the second claim. Case \(h = 1\). Simply put, \(|C_i| = 1 < 3^{\frac{2}{3}} \leq |C_{i-2}|^{\frac{5}{3}}\).

Case \(h = 2\). Since \(H_{i-2} = h_{i-2}^* + H_{i-1} = h_{i-2}^* + 3h - 1 \geq 7\), the subtree \(T_{i-2}\) cannot have less nodes than those \((33)\) of the Fibonacci tree of height 7, so \(|T_{i-2}| \geq 33\). This implies that \(|C_{i-2}| = 2^{|\log \sqrt{|T_{i-2}|}| - 1} \geq 2^{|\log \sqrt{33}| - 1} = 7\). Thus, \(|C_i| = 2^h - 1 = 3 < 7^{\frac{2}{3}} \leq |C_{i-2}|^{\frac{5}{3}}\). Case \(h = 3\).

We first prove that \(h_{i-2}^* \geq 4\). Since \(h_{i-2}^* \geq h_{i-1}^* = h\), we have \(H_{i-2} = h_{i-2}^* + 3h - 1 \geq 4h - 1 = 11\).

The subtree \(T_{i-2}\) cannot have less nodes than those \((232)\) of the Fibonacci tree of height 11, so \(|T_{i-2}| \geq 232\) and \(h_{i-2}^* = |\log \sqrt{|T_{i-2}|}| \geq |\log \sqrt{232}| = 4\). Next, we give a better bound on the height of \(T_{i-2}\) as \(H_{i-2} = h_{i-2}^* + 3h - 1 \geq 12\). The Fibonacci tree of height 12 has 376 nodes, and so \(|C_{i-2}| = 2^{|\log \sqrt{|T_{i-2}|}| - 1} \geq 2^{|\log \sqrt{376}| - 1} = 31\). Thus, \(|C_i| = 2^h - 1 = 7 < 31^{\frac{2}{3}} \leq |C_{i-2}|^{\frac{5}{3}}\). □

We also need to prove that there are few small cores (see Definition 3.2). This follows the same path as we did at the end of the proof of Lemma 4, thus showing that AVL admits a core partition with \(\gamma = 2/3\) and \(c = 2\). Thus, let us consider \(C_{t-1}\) when its size is \(< r^*\), and observe that the subtree \(T_{t-1}\) of which \(C_{t-1}\) is the topmost core, has size \(|T_{t-1}| \geq r^* + 1\), and the height of \(C_{t-1}\) is \(h_{t-1}^* = [(1/2) \log_2 |T_{t-1}|] \geq (1/2) \log_2 (r^* + 1)\). Hence, \(\sqrt{r^* + 1} - 1 \leq |C_{t-1}| < r^*\). An immediate induction on \(j = 1, 2, \ldots\) gives that \(|C_{t-2-j}| \geq |C_{t-1}|^{(3/2)^j}\) by transitivity.

Setting \(j = 1\) we have that \((\sqrt{r^* + 1} - 1)^{(3/2)^j} \geq r^*\) gives an upper bound of \(j + 1 = 2\) on the maximum number of cores having size \(< r^*\) in \(C_1, C_2, \ldots, C_{t-1}, B_t\), thus proving the condition in Definition 3.2.

### 3.4.2 Amortized Analysis for Repartitioning

We prove that an amortized (poly)logarithmic cost cannot be achieved for maintaining a core partition of AVL trees, contrarily to the case of weight-balanced trees as discussed in Section 3.3.2. For any \(n \geq 2\), we can produce a sequence of \(n\) insertions into an initially empty AVL tree with \(\Omega(n)\) rotations. The cost of these operations is dominated by the repartition operations. In
particular, the total cost of the corresponding **subtree rescans** is a lower bound for the amortized cost of the sequence of \( n \) insertions. Thus we prove that the latter cost alone prevents from obtaining a poly-logarithmic amortized cost.

**Lemma 9** Given any AVL tree of height \( h \), its height can be increased by one with at most \( F_{h+2} \) insertions

**Proof**: By induction on \( h \), the base case is a unary node of height 1, and thus its height becomes 2 by a \( F_1 = 1 \) insertion that replaces one of the missing child by a leaf. For the inductive case, suppose that the height \( k < h \) of an AVL tree can be increased with at most \( F_{k+2} \) insertions. Let \( x \) be the root of the AVL tree of height \( h \), and observe that \( x \)'s children either have the same height or their heights differs by one. If \( x \) has two children of same height \( k = h - 1 \), we can increase the height of one of them by induction, and thus this increases the height of the AVL tree by one with \( F_{k+2} < F_{h+2} \) insertions. If \( x \) has one child \( y \) of height \( h - 1 \) and another one \( z \) of height \( h - 2 \), we first perform \( F_h \) insertions into the subtree rooted at \( z \) to increase its height by one and then perform \( F_{h+1} \) insertions into the subtree rooted at \( y \) to increase its height by one (using inductive hypothesis twice with \( k = h - 2 \) and \( k = h - 1 \)). These insertions are at most \( F_h + F_{h+1} = F_{h+2} \) in number, and increase the height of the AVL by one. \( \square \)

**Theorem 3** The amortization cost for **subtree rescans** is \( \Omega((\frac{2}{\phi})^{\log n}) \), where \( \phi = \frac{1+\sqrt{5}}{2} < 2 \) is the golden ratio, and thus **subtree rescans** for AVL trees cannot be amortized in poly-logarithmic time.

**Proof**: We provide a counterexample for a tree \( T \) of height \( h + 1 \) whose left subtree is a complete balanced binary tree of height \( h \), named \( B \), and the right subtree (right sibling of \( B \)) is an arbitrary AVL tree of height \( h \). Let \( P(B) \) and \( B^{\text{rib}} \) denote the parent of \( B \) (initially the root of the tree) and the right sibling of \( B \). We apply Lemma 9 “twice” to \( B^{\text{rib}} \) to increase its height by 2: first we increase its height from \( h \) to \( h + 1 \) by at most \( F_{h+2} \) insertions, then we increase its height from \( h + 1 \) to \( h + 2 \) by at most another \( F_{h+3} \) insertions. This makes its height \( h + 2 \), which in turn causes a rotation on the tree to make it balanced. Because of the rotation, \( P(B) \) and \( B^{\text{rib}} \) change and move one level below. By definition, now \( P(B) \) and \( B^{\text{rib}} \) denote to new parent and sibling of \( B \), thus, \( P(B) \) and \( B^{\text{rib}} \) will be again of height \( h + 1 \) and \( h \), respectively.

We repeatedly apply Lemma 9 “twice” to \( B^{\text{rib}} \) to increase its height by 2 (from \( h \) to \( h + 2 \)). Each time this height increases, it causes a rotation on \( P(B) \) and produces new \( P(B) \) and \( B^{\text{rib}} \) of height \( h + 1 \) and \( h \). In each rotation, \( B \) is involved in the **subtree rescans**, so the cost of the rotation is at least \( 2^h \). The number of insertions needed to generate this rotation at each iteration is at
most $F_{h+2} + F_{h+3} = F_{h+4}$. If we let $n' = 2^h$ and do $\frac{n'}{F_{h+4}}$ iterations, the total number of insertions is $n = O\left(\frac{n'}{F_{h+4}} \times F_{h+4}\right) = O(n')$. But the total cost of subtree rescan is $\Omega\left(n' \times \frac{n'}{F_{h+4}}\right) = \Omega\left(n(\frac{2}{\phi})^{\log n}\right)$ where $\phi$ is the golden ratio.

\section*{3.5 Summary}

In this chapter, we presented the core partitioning scheme, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees called cores, we preserve the original topology and algorithms of the given balanced search tree using a simple post-processing with guaranteed performance to completely rebuild the changed cores (possibly all of them) after each update. We introduced the necessarily properties for a core partitioning on a given binary search tree to be efficient in space and time, as case studies we applied core partitioning to weight-balanced trees and height-balanced trees (AVL trees). We had shown that they simultaneously achieve good memory allocation, space-efficient representation, and cache obliviousness. For AVL trees, the logarithmic amortization of insertion/deletion is impossible (i.e., AVL trees require super poly-logarithmic cost by a lower bound on the subtree size of the rotated nodes), while weight-balanced trees can be maintained with a logarithmic cost.
Chapter 4

Core Partitioning Directly on Plain Binary Trees

We introduced the notion of core partition in Chapter 3 to show how to obtain cache-efficient versions of classical balanced binary search trees such as AVL trees and weight-balanced trees. Looking at weight-balanced tree in Chapter 3 that are kept balanced using Overmars’ partial rebuilding [77, Sect.4.2] instead of rotations (see Section 3.3), we observe that a subtree is rescanned for two reasons.

- Its root $u$ is unbalanced and we use Overmars’ partial rebuilding.

- Its top core needs core resize and we have to maintain the core partition without changing the underlying topology.

This seems an interesting challenge to investigate: what if we use the core partition on plain binary search trees? In other words, what if we maintain the tree balanced just by using core partitioning without extra rebalancing operations such as rotations? An objection is that they do not have a core large enough. However, we can use an “aggressive” version of core resize, so that when we maintain the core partition, we also transform the subtree in a perfectly balanced tree as in Overmars’ partial rebuilding [77, Sect.4.2].

Now let us reformulate the challenge: take an empty plain binary search tree and, whenever core resize happens, transform the subtree in a perfectly balanced tree. We only operate the above, no rebalancing is performed.
It can be easily observed that we can get $O(\log^2 n)$ height using only the aggressive version of core resize. However, this is not so interesting, as the same bound can be obtained with the logarithmic rebuilding method using a logarithmic number of sorted arrays. Can we get $O(\log n)$? In this chapter we give a positive answer to this question.

We also adopt the cache-oblivious model [41, 82] explained in Chapter 2 to evaluate the I/O complexity. Note that $B$ is now an unknown parameter for the block size. We introduce a new data structure, called Cache-Oblivious General Balanced Tree (COG-tree) which guarantees logarithmic search time and logarithmic amortized insert time. We use cores as explained in Chapter 3, so we keep the first levels of each node of the tree in the form of a complete balanced binary tree. As we will see this property can be also maintained after insertion in a very simple way and in logarithmic amortized time for proper values of the number of full levels. The latter property allows us to efficiently use the new structure for cache-oblivious model.

Our proposal exhibits good performances that is a COG-tree of $n$ nodes requires $O(\log_B n)$ I/Os amortized for updates, and $O(\log_B n)$ I/Os worst case for searches. In addition, it can be laid out in $O(n)$ space. These complexities are optimal theoretically, and our structure compares optimally with respect to the previous ones. It obtains the same optimal results with respect to the one of Bender et al. [19], by using very simple, basic, and well known data structure.

In this chapter, in Section 4.1, we study some preliminaries of Cache-Oblivious General Balanced Trees (COG-trees), we define COG-trees in Section 4.2, then we show how to maintain COG-trees in cache-oblivious and external memory models in Section 4.3.

### 4.1 Preliminaries and Notation

As mentioned in Chapter 2, most binary search trees require storing data (e.g. ‘colors’, ‘weights’, ‘rank’, etc.) on the nodes of the tree and checking at each update that some constraints on the structure of the tree are maintained. This information must be dynamically updated after insertions and deletions. A different approach let the tree assume any shape as long as its height is logarithmic. In this way there is no need of storing and checking the balance information, but it is sufficient to check whether the maximal possible height has been exceeded. Trees of this kind, called General Balanced Trees, introduced by [7] and later rediscovered by [43] under the name of scapegoat trees, can be efficiently maintained and require as additional space only that for the pointers. They are restructured with an operation, called partial rebuilding, that transforms a subtree of the tree in a perfectly balanced tree. The operation is expensive having
4.1. PRELIMINARIES AND NOTATION

a cost proportional to the number of nodes of the subtree, but performed rarely hence has a low amortized cost.

Our Cache-Oblivious General Balanced Tree (COG-tree) data structure is a smooth extension of general balanced trees [7, 43] and guarantees the same logarithmic search time and logarithmic amortized insert time in the comparison model as the General Balanced Trees. Our structure, uses cores as presented in Chapter 3 (keeps the first levels of each node of the tree in the form of a complete balanced binary search tree) and we maintain this property after insertion in a very simple way and in logarithmic amortized time for proper values of the number of full levels. The latter property allows us to efficiently use the new structure for cache-oblivious model.

For a given node $u$, we use $T_u$ to denote the subtree of $T$ rooted at $u$, $s(u) = |T_u|$ the subtree size, and $h(u)$ the height of $T_u$. A perfectly balanced tree of $n$ nodes satisfies the property that its height is the minimal possible, namely, $H = \lceil \log_2(n + 1) \rceil$.

In the core partitioning introduced in Chapter 3, we say that $T$ has a core of height $h^*$, if its topmost $h^*$ levels are full. Suppose that each nonempty subtree of $T$ of size larger than $r^*$ has a core of height $h^*$, where $h^*$ is a function of the subtree size and $r^*$ is a function of $n$. In Chapter 3, in Figure 3.1 an example of core partitioning was given. We defined that a (successful) core partition if it satisfies the conditions below.

1. Any root-to-leaf path in $T$ traverses cores of exponentially nonincreasing size.

2. Only a constant number of the above cores are of small size $< r^*$.

The fact the all levels are full in a core $C$ can be exploited by the implicit van Emde Boas (vEB) layout [20, 26, 82], which compactly stores $C$ pointerless as an array of $|C|$ elements. These elements are the keys in the nodes of $C$, so that it takes $O(|C|)$ block transfers to implicitly traverse $C$ during a search path [20]. Since the cores in that path are of nonincreasing exponential size, the total cost is $O(\log_B |C|)$ block transfers.

**Fact 5** Consider a node $v$ in a core $C$, and let $m$ be the number of all the nodes descending from $v$ (including itself) that are inside $C$. Then, the inorder traversal of these nodes in $C$ requires $O(m/B + 1)$ block transfers.

As a result, we can produce the ordered sequence of the $k$ keys in the above nodes in $O(k/B)$ block transfers. Vice versa, given $k$ keys in sorted order, we can place them in their positions of $C$ in $O(k/B)$ block transfers (supposing they fulfill the search property of binary search trees).
4.2 Definition of COG-Tree

We begin by considering whether also unbalanced binary search trees can benefit of the core partition. In this section, we explore how to employ the core partition to make these trees balanced and cache-oblivious.

For a node $u$ in a binary search tree $T$, let $h_{\text{min}}(u) = \lceil \log_2(s(u) + 1) \rceil$ denote the minimal height of a binary search tree storing $s(u)$ keys. We can enforce cores in $T$ by requiring the following condition to be maintained by partial rebuilding.

{**fullness invariant**} The first $\max\{1, h_{\text{min}}(u) - 1\}$ levels of each node $u$ in $T$ are full.

Unfortunately, the condition alone does not guarantee to obtain a core partition. In particular, the height can be more than logarithmic, as we observe next.

For a given positive integer $j$, let $Q_j$ be the tree constructed as follows. $Q_j$ is built starting from a complete balanced binary tree of height $j$, where we replace one of the leaves with another complete balanced binary tree of height $j - 1$; again one leaf of this second tree is replaced with another complete balanced binary tree of height $j - 2$, and so on till we reach a binary tree of height 2.

**Lemma 10** The height of $Q_j$ is $\Theta(\log^2 |Q_j|)$.

**Proof:** The overall size of this tree is given by $|Q_j| = (2^j - 1) + (2^{j-1} - 2) + (2^{j-2} - 2) + \ldots + (2^2 - 2) = 2^{j+1} - 2j - 1$. On the other hand, the overall height of $Q_j$ is given by $h = j + (j - 2) + (j - 3) + \ldots + 1 = j(j - 1)/2 + 1 = \Theta(j^2) = \Theta(\log^2 |Q_j|)$. \hfill \Box

We can enforce to have a logarithmic height in $T$ by requiring the following condition to be maintained by partial rebuilding.

{**height invariant**} $h_{\text{min}}(u) \leq h(u) \leq 2h_{\text{min}}(u)$ for each node $u$ in $T$.

Note that the above condition has been employed several times in different forms, including for General Balanced Trees. However, it is not sufficient alone to get a core partition as shown next. Indeed the height invariant does not necessarily imply the presence of cores of sufficiently large height, as the following counterexamples show. A first counterexample is simply given by a tree composed of a root with a complete balanced binary tree as left subtree and null as right subtree. In this case it is easy to observe that the height invariant is satisfied, but there is no core involving the root. As a more general counterexample, we construct a tree with the following structure. We
start from a complete balanced binary tree of height 3, and we randomly remove one leaf. Let \( P_1 \) denote this tree. Starting from \( P_1 \), we recursively construct a tree \( P_{i+1} \) substituting each of the three leaves of \( P_1 \) with one subtree \( P_i \).

**Lemma 11** \( P_i \) satisfies the height invariant, but it cannot be partitioned into big cores.

**Proof**: By construction, it is obvious that, for any \( i \), \( P_i \) does not contain cores with height greater than three. Now, consider any node \( u \) in \( P_i \). To show that the height invariant is satisfied, we need to prove that the height \( h(u) \) of the subtree \( T_u \) rooted at \( u \) is less or equal to \( 2h_{\min}(u) \).

Observe that \( u \) is either the root or one of the children of the root of a \( P_{\lceil h(u)/2 \rceil} \) tree. In both cases, \( |T_u| \geq |P_{\lceil h(u)/2 \rceil}-1| + 1 \). Thus \( h_{\min}(u) \geq \log |P_{\lceil h(u)/2 \rceil}-1| \). On the other hand, it is easy to observe that \( |P_i| = 3|P_{i-1}| + 3 \), and therefore \( |P_i| = 3^i + 3^i+1 - 1 \).

Hence,

\[
\begin{align*}
h_{\min}(u) & \geq \log |P_{\lceil h(u)/2 \rceil}-1| \\
& \geq \log(3^{\lceil h(u)/2 \rceil}-1 + 3^{\lceil h(u)/2 \rceil} - 1) \\
& \geq h(u)/2.
\end{align*}
\]

Therefore, \( h(u) \leq 2h_{\min}(u) \). \( \square \)

We conclude that both fullness invariant and height invariant are necessarily for COG-tree to be efficient. In the following we study more features of COG-tree.

**Lemma 12** For any node \( v \) in a COG-tree \( T \), \( 2^{h_{\min}(v)-1} - 1 \leq s(v) \leq 2^{h_{\min}(v)+1} - 1 \).

**Proof**: Recall that by definition of \( h_{\min}(v) \) we have \( h_{\min}(v) = \lceil \log(s(v) + 1) \rceil \). Therefore, we get

\[
\begin{align*}
\log(s(v) + 1) - 1 \leq h_{\min}(v) \leq \log(s(v) + 1) + 1 \\
h_{\min}(v) - 1 \leq \log(s(v) + 1) \leq h_{\min}(v) + 1 \\
2^{h_{\min}(v)-1} \leq s(v) + 1 \leq 2^{h_{\min}(v)+1} \\
2^{h_{\min}(v)-1} - 1 \leq s(v) \leq 2^{h_{\min}(v)+1} - 1.
\end{align*}
\]

\( \square \)

In the following lemma we show that in average in a search path just 4 cores will be visited (average path length is 4 cores).
Lemma 13 In any tree $T$ with fullness invariant (e.g. COG-trees), the average number of cores traversed to visit a node (search a node) is less than or equal to 4.

Proof: We use the assumption that for a random search operation, the probability of searching for each node of $T$ is uniformly distributed. Let $n = |T|$, suppose that we perform $n$ search operations to visit all nodes of $T$, let $|E(T)|$ denotes the total number of cores traversed during these search operations divided by $n$ (i.e., the average number of cores traversed to visit a node).

We use induction on $n$ to prove that $|E(T)| \leq 4$, for small values of $n$, it is immediate. Assume that for any tree with fullness invariant and size less than $n$, the lemma holds. Let $C$ denotes the topmost core of $T$ (the core with $h_{\text{min}}(v) - 1$ topmost levels of $T$), since $|C| = 2^{h_{\text{min}}(v)} - 1$, by Lemma 12, we have that $|C| \geq \frac{n}{4}$.

Now let $T_1, T_2, \ldots, T_{2^{h_{\text{min}}(v)}}$ denote the $2^{h_{\text{min}}(v)}$ subtrees below $C$ and $n_i = |T_i|$ for $1 \leq i \leq 2^{h_{\text{min}}(v)}$ (also $\sum_{i=1}^{h_{\text{min}}(v)} n_i = n - |C|$). By induction hypothesis, we have $E(T_i) \leq 4$ for $1 \leq i \leq 2^{h_{\text{min}}(v)}$. On the other hand, for all $|C|$ nodes in the core $C$, we just traverse one core to visit them, and for the all nodes in $|T_i|$ (1 \leq i \leq 2^{h_{\text{min}}(v)}) in average, we traverse less than or equal to (4+1) cores (by induction hypothesis and also the fact that they are all below core $C$). Therefore:

$$E(T) = \frac{|C| \times 1 + \left(\sum_{i=1}^{h_{\text{min}}(v)} n_i\right) (4 + 1)}{n} \leq \frac{|C| \times 1 + (n - |C|) (4 + 1)}{n} \leq \frac{-4|C| + 5n}{n} = 5 - \frac{4|C|}{n}$$

Finally since $|C| \geq \frac{n}{4}$, $\frac{4|C|}{n} \geq 1$. Thereby $E(T) \leq 4$ and the lemma holds. $\square$

Fortunately, both the fullness and height invariants guarantee the existence of a core partition.

Lemma 14 If a binary tree $T$ satisfies both the fullness and height invariants, it also admits a core partition with $r^* \geq 1$ and $h^* = \max\{1, h_{\text{min}}(u) - 1\}$ for the core root $u$.

Proof: We prove that condition 1 of core partition in Section 4.1 holds. Consider a core $C$ and its root $u$. By the fullness and height invariants, we have that $|C| = 2^{h^*} - 1$ and $h_{\text{min}}(u) \leq h(u) \leq 2h_{\text{min}}(u)$. Consider now any two cores $C'$ and $C''$ such that $C'$ is a child of $C$ and $C''$ is a child of $C'$. Observe that $|C'| \geq |C''|$ where $|C'| = 2^{h'} - 1$ and $|C''| = 2^{h''} - 1$ for some
4.3. MAINTAINING A COG-TREE

$h', h''$. Their total height is upper bounded by the difference between $h(u)$ and the core height, namely, $h' + h'' \leq h(u) - h^* \leq 2h_{\min}(u) - h^* \leq h_{\min}(u)$. Hence, $h'' \leq 1/2h_{\min}(u)$ and thus $|C''| = O(\sqrt{|C|})$. It is immediate to see that also condition 2 holds as every two other cores the size decrease exponentially.

We call a binary search tree $T$ of $n$ nodes a Cache-Oblivious General Balanced Tree (shortly, COG-tree) when it satisfies both the fullness and height invariants: here we choose a core partition with $h^* = \max\{1, h_{\min} - 1\}$ and $r^* = \log n$.

The implementation of a COG-tree $T$ exploits its core partition and it is an easy programming task. Given a core $C$ of $T$, we observe that it contains $|C| = 2h^* - 1$ keys for some $h^*$, and $|C| + 1$ external pointers to its “children” cores. We can thus allocate an array of size $2|C| = 2h^* + 1$ entries, and fill it with the entries from $C$ using the implicit vEB layout (see Section 4.1), so no internal pointers among $C$’s nodes are stored. Also, the bottom subtrees of size $< r^* = \log n$ are stored sequentially.

**Corollary 3** For $r^* \geq \log n$, The total space occupancy is $O(n)$ bits besides the space needed for storing the $n$ keys, as there are $O(n/\log n)$ external pointers (pointing to the cores).

Because of the implicit vEB layout, same as explained in Chapter 3, searching a key requires $O(\log_B n + \log(\log(B+1)/\log(\log n+1)))$ block transfers and $O(\log n)$ CPU time in the cache-oblivious model. We later improve COG-trees to have a searching time of $O(\log_B n)$ block transfers instead of $O(\log_B n + \log(\log(B+1)/\log(\log n+1)))$, in the same manner we did in Chapter 3. It should be mentioned that in the external memory model, this cost reduces to $O(\log_B n)$ by setting $r^* = \max\{\log n, B\}$ without any other improvement.

### 4.3 Maintaining a COG-Tree

We show in this section that insertions and deletions can be supported each in $O(\log_B n)$ amortized block transfers (excluding the search cost) and $O(\log n)$ amortized time.

#### 4.3.1 Deletions

Similar to Chapter 3, for the deletion, we simply mark the searched key as logically deleted, and remove that mark if the key is inserted again. We periodically rebuild the entire tree when the number of these marked keys is a constant fraction of the total number of keys. This amortized complexity can be analyzed in a traditional way, and thus it is not discussed here.
4.3.2 Insertions

We focus therefore on the insertions. When a new key is inserted in $T$, a leaf $f$ is created. After that, we have to maintain both the fullness and height invariants described in Section 4.2, as well as the core partitioning.

Recall that for any node $u$, $h_{\min}(u) = \lceil \log(s(u) + 1) \rceil$. When $f$ has been inserted, any ancestor $z$ of $f$ whose value $h_{\min}(z)$ increases by one (we call this event minimum-height-increase of $z$) could violate the fullness invariant. To preserve both fullness or height invariants we then proceed as follows. We take the topmost ancestor $u$ of $f$ which violates the height invariant or has a minimum-height-increase, and apply the operation called balance($u$) which

(a) replaces $T_u$ by a perfectly balanced tree $T'_u$ of height $h_{\min}(u)$ storing the same set of keys, and

(b) updates the core partition.

Task (b) is performed as follows. Let $C$ be the core containing $u$: we replace the entries for $C \cap T_u$ in the array storing $C$ with the topmost $|C \cap T_u|$ entries from $T'_u$; observing that the number of these entries is a power of 2 minus 1, they correspond to the topmost full levels, say the first $t$ levels of $T'_u$. The remaining entries in $T'_u$, which are on levels greater than $t$, are stored in cores using a simple greedy top-down approach.

Operation balance($u$) can be performed in $O(s(u))$ time using a variation of the algorithm in [7], as we have now to rebuild also the cores, and the observation that the keys in $C \setminus T_u$ do not change. However, the resulting algorithm is not efficient in terms of cache complexity. Here we show how to perform balance($u$) in the cache-oblivious model.

**Lemma 15** Operation balance($u$) can be performed in $O(s(u))$ time and $O(s(v)/B + s(v)/\log n)$ block transfers in the cache-oblivious model.

**Proof:** Suppose that we want to perform an inorder traversal of the subtree $T_u$. We can easily see that it takes $O(s(u))$ time. Here we discuss its cache complexity. Let $C$ be the core containing $u$, and observe that the inorder traversal of $C \cap T_u$ requires a linear number of blocks, $O(|C \cap T_u|/B + 1)$, by Fact 5. For the rest of the cores in $T_u$, we use a bottom-up induction on the cores traversed by the inorder traversal. Let $C'$ be one of the cores below $C$ that are traversed in $T_u$, and let $d(C')$ be the number of “children” cores of $C'$. By Fact 5, it takes $O(|C'|/B + 1)$ block transfers. We should also add $d(C')$ block transfers that are needed to access its children. Hence,
the overall cache complexity of the inorder traversal is

$$O(|C \cap T_u|/B + \sum_{C' \in T_u} (|C'|/B + d(C'))) = O(s(v)/B + \sum_{C' \in T_u} d(C'))$$

where $\sum_{C' \in T_u} d(C') = O(s(v)/\log n)$ as there are so many cores (and bottom subtrees) in $T_u$. As a result, we can produce the sorted sequence of keys in $T_u$ with $O(s(v)/B + s(v)/\log n)$ block transfers. After that, it is a standard computation to build the perfectly balanced tree $T'_u$ with $O(s(v)/B)$ block transfers, thus completing task (a). As for task (b), we can observe that the cache complexity follows the same route as that for the inorder traversal, thus giving a total cost of $O(s(v)/B + s(v)/\log n)$ block transfers.

As stated in the next lemma, the rebalancing operation described above preserves both invariants in the whole COG-tree.

**Lemma 16** Operation balance($u$) preserves both height and fullness invariants.

**Proof**: The height invariant is always preserved as the operation balance($u$) executed on a node $u$ does not increase $h(u)$.

Now for a given node $v$, let $e(v)$ denote the topmost null pointer (empty node) in $T_v$ and let $\ell_v$ be the length of the path between $v$ and $e(v)$. When operation balance($u$) is performed, $T_u$ simply becomes balanced by filling some null pointers (empty nodes) closer to node $u$ with other nodes further from $u$. This means in general that balance($u$) does not decrease $\ell_z$ for any ancestor $z$ of $u$.

Suppose by contradiction that after performing balance($u$), there is an ancestor $z$ of $u$ that violates the fullness invariant. Before the balance($u$) operation, $T_z$ was full up to $h_{\min}(z) - 1$ level, so $\ell_z$ was at least equal to $h_{\min}(z) - 1$. On the other hand the fact that, after balance($u$), $z$ violates the fullness invariant, implies that $\ell_z \leq h_{\min}(z) - 2$. This means that balance($u$) caused a decrease in $\ell_z$, which is in contradiction with the previous result.

Finally, note that by Lemma 12, checking whether the fullness invariant has been violated or not on a node $v$ has cost $\Theta(s(v))$; therefore in case of a minimum-height-increase event, rebuilding the subtree is not an inefficient procedure. The amortized cost of this procedure will be discussed in the following section.
4.3.3 Amortized Analysis

We are now ready for the amortized analysis, where we need the following properties on the fullness and height invariants.

**Fact 6** For any node \( v \) in a COG-tree, \( \lfloor h(v)/2 \rfloor \leq h_{\text{min}}(v) \). Therefore, at least the topmost \( \max\{1, \lfloor h(v)/2 \rfloor - 1\} \) levels of \( T_v \) are full by the fullness invariant.

Consider an insertion of a new leaf \( f \) in a COG-tree \( T \), and suppose that a minimum-height-increase happens in an ancestor of \( f \). Let \( v \) be the topmost such an ancestor. Let \( n = s(v) \) be the size of the subtree \( T_v \) rooted at \( v \) and let \( T'_v \) denote the subtree rooted at \( v \) after the last rebalancing operation on \( v \) due to a previous minimum-height-increase. Finally, let \( n' = |T'_v| \) and \( h'_{\text{min}}(v) = \lceil \log(n'+1) \rceil \).

**Lemma 17** After the last balance\((v)\) operation, \( \Omega(n) \) new keys are inserted as descendants of \( v \), i.e., \( n - n' = \Omega(n) \).

**Proof:** By definition of minimum-height-increase, \( h_{\text{min}}(v) = h'_{\text{min}}(v) + 1 \) implying that \( \lceil \log(n+1) \rceil = \lceil \log(n'+1) \rceil + 1 \). Since in both cases we just perform a balance operation due to a minimum-height-increase, \( n \) and \( n' \) are powers of 2. Thus we have that \( \log n + 1 = \log n' + 2 \), implying \( n = 2n' \), and finally \( n - n' = \Omega(n) \). \( \square \)

Suppose that the height invariant is violated in an ancestor of the new inserted leaf \( f \), and let \( v \) be the topmost such an ancestor. Let \( n = s(v) \) be the size of \( T_v \), where its height is now \( h(v) = 2h_{\text{min}}(v) + 1 \) (it cannot be \( h(v) < h_{\text{min}}(v) \) as we are inserting \( f \)). Consider the last time \( v \) was involved in a rebalancing operation on \( v \) or on an ancestor of \( v \). We denote by \( T'_v \) the subtree rooted at \( v \) soon after that operation, and by \( n' \) its size \( (n' = |T'_v|) \). Recall that by definition of rebalancing operation, the height of \( T'_v \) is exactly \( h'_{\text{min}}(v) \).

**Lemma 18** \( h_{\text{min}}(v) = h'_{\text{min}}(v) \).

**Proof:** First observe that it cannot be \( h_{\text{min}}(v) < h'_{\text{min}}(v) \) since we are only inserting new keys in the tree. Suppose by contradiction that \( h_{\text{min}}(v) > h'_{\text{min}}(v) \). This means that a minimum-height-increase has occurred on \( v \) in between, implying a balance\((v)\) operation, in contradiction with the fact that the aforementioned operation is the last one involving \( v \). \( \square \)

**Lemma 19** After the last time \( v \) is involved in a balance\((v)\) operation, \( \Omega(n) \) new keys are inserted as descendants of \( v \), i.e., \( n - n' = \Omega(n) \).
4.3. MAINTAINING A COG-TREE

\[ v = v_{2h_{\min}(v) + 1} \]

\[ v_{2h_{\min}(v)} \]

\[ v_{2h_{\min}(v) - 1} \]

\[ v_{2h_{\min}(v) - 2} \]

\[ v = v_{2h_{\min}(v) + 1} \]

\[ v_{2h_{\min}(v)} \]

\[ v_{2h_{\min}(v) - 1} \]

\[ v_{2h_{\min}(v) - 2} \]

\[ h_{\min}(v) \]

\[ 2h_{\min}(v) + 1 \]

\[ v_3 \]

\[ v_2 \]

\[ x = v_1 \]

Figure 4.1: The path \( v_1, v_2, \ldots, v_{2h_{\min}(v) + 1} \), from the new inserted leaf \( f \) to \( v \), and node \( v_{2h_{\min}(v) - 1} \) whose first \( h_{\min}(v) - 1 \) levels are full.

Proof: Just recall that the height of \( T_v \) is \( 2h_{\min}(v) + 1 \). Now consider the path \( v_1, v_2, \ldots, v_\ell \), from the new inserted leaf \( v_1 = f \) to \( v_\ell = v \). This path is the longest path from \( v \) to one of its descendant leaves, otherwise inserting the new leaf \( f \) wouldn’t violate the height invariant. Moreover, the height difference between the subtrees rooted at any two consecutive nodes in the path \( v_1, v_2, \ldots, v_\ell \) (child and parent) is 1. Therefore, \( h(v_i) = i \), for \( 1 \leq i \leq \ell \), and \( \ell = 2h_{\min}(v) + 1 \). Observe that by Fact 6, for the subtree rooted at \( v_i \), at least the topmost \( \max\{1, \lfloor h(v_i)/2 \rfloor - 1\} = \max\{1, i/2 - 1\} \) levels form a complete balanced binary tree.

As shown in Figure 4.1, consider the tree rooted at \( v_{2h_{\min}(v) - 2} \), which is complete at least up to the level \( h_{\min}(v) - 2 \). Observe that this level contains \( 2^{h_{\min}(v) - 3} \) nodes. Since it corresponds to level \( h_{\min}(v) + 1 \) of \( T_v \) \( (v_{2h_{\min}(v) - 2} \) is three levels below \( v \) and at least up to the level \( h_{\min}(v) - 2 \) is a complete balanced binary tree), it is not part of \( T'(v) \) and therefore contains only new nodes (inserted after the last rebalancing involving \( v \)). Therefore, \( n = n' \geq 2^{h_{\min}(v) - 3} \).

On the other hand, \( n' \leq 2^{h_{\min}(v')} - 1 = 2^{h_{\min}(v) - 1} \), so \( n - n' \geq 1/8 n' \), also \( n = \Theta(n') \) (since \( h_{\min}(v) = h_{\min}'(v) \)). Consequently \( n - n' = \Omega(n) \). \( \square \)

Both Lemmas 17 and 19 imply that the balance(\( u \)) cost in Lemma 15 can be spread among \( O(s(u)) \) fresh insertions. As a result, the amortized cost is \( O(1) \) time and \( O(1/B + 1/log n) \) block transfers per new entry in the subtree \( T_u \). Since each new entry is an inserted leaf \( f \) at some time, and \( f \) is involved as a fresh entry in \( O(log n) \) ancestors, we can charge \( f \) with \( O(log n) \) time and \( O((log n)/B + 1) = O(log_B n) \) block transfers. Hence, the amortized cost of restructuring the COG-tree is \( O(log n) \) time and \( O((log n)/B + 1) = O(log_B n) \) block transfers. This cost does not account for searching the place where to insert a new leaf.
4.3.4 External-Memory Search Trees

In a similar manner to Chapter 3, by setting \( r^* = \max\{\log n, B\} \), we obtain a \( B \)-tree-like data structure for external memory [17]. More precisely, the complete balanced binary tree represented by each core \( C_i \) can be stored in contiguous portions of memory of size of multiples of \( B \), so that it takes \( O(1 + h^*_i / \log B) \) I/Os to traverse \( C_i \) (e.g. see [109]).

**Theorem 4** A COG-tree of \( n \) nodes with \( r^* = \max\{\log n, B\} \), has \( O(n/r^*) \) cores, and it can be stored in \( O(n) \) bits space besides the space needed to store the keys alone. In the external memory model with block size \( B \), \( O(n/B) \) blocks are occupied and any search path from the root to a node requires \( O(\log_B n) \) I/Os and \( O(\log n) \) comparisons.

**Proof**: The proof is similar to the proof of Theorem 1. \( \square \)

4.3.5 Cache-Oblivious Search Trees

By fixing \( r^* = \log n \), in a similar manner to Chapter 3, we can store the subtrees of size at most \( r^* \) in a contiguous memory chunk that can be scanned. We then store the complete balanced binary tree inside each core \( C_i \) in a contiguous memory chunk using the van Emde Boas layout [82, 20, 26], so that it takes \( O(1 + h^*_i / \log B) \) block transfers to traverse \( C_i \) during a search path. This suffices to obtain cache-oblivious bounds.

**Theorem 5** A COG-tree of \( n \) nodes can be stored in \( O(n) \) bits besides the space needed to store the keys alone. In the cache-oblivious model with cache size \( M \) and transfer block size \( B \), each search operation requires \( O(\log n) \) time and \( O(\log_B n + \log(\log(\log B + 1) / \log(\log n + 1))) \) block transfers, and each update operation has a restructuring cost of \( O(\log n) \) amortized time and \( O(\log_B n) \) amortized block transfers.

In Theorem 5, search operation is not efficient enough, more precisely, the \( \log(\log(\log B + 1) / \log(\log n + 1)) \) term in the number of block transfers of search operation is not theoretically efficient, therefore in the same manner we presented in Chapter 3, it can be avoided by keeping any core \( C \) and all its descendants in a contiguous portion of memory. By applying those techniques we obtain the \( O(\log_B n) \) block transfer cost for search operation and achieve the linear space.
4.4 Summary

In this chapter, we applied the core partitioning scheme introduced in Chapter 3 to arbitrary binary search trees which can be ‘unbalanced’. We then introduced a new data structure called Cache-Oblivious General Balanced Tree (COG-tree). The COG-tree of $n$ nodes has an improved cache complexity of $O(\log_B n)$ amortized block transfers and $O(\log n)$ amortized time for updates. Search operation takes $O(\log_B n)$ block transfers and $O(\log n)$ comparisons. The space occupancy is $O(n)$ extra bits besides the space needed to store the keys alone.
CHAPTER 4. CORE PARTITIONING DIRECTLY ON PLAIN BINARY TREES
Chapter 5

Hat Partitioning Scheme

As we proved in Chapter 3, that core partitioning on AVL trees can not be maintained in amortized logarithmic time. Another question would be whether the core partition idea can be applied to AVL trees with constant or logarithmic average time? We show a simple version of core partitioning to maintain the nodes of an AVL tree in external memory model. For $n$ keys and block size $B$, the searching cost is $O(\log_B n)$ block transfers in the worst case, while the updating cost is equal to the searching cost plus $O(1)$ expected block transfers (for restructuring the tree) assuming that AVL trees of the same height are uniformly distributed. The analysis is based on the fact that the expected cost is constant under the assumption that rebalancing a node has a cost proportional to its subtree size. We again preserve the original topology and the insertion algorithms of the 1962 paper introducing AVL trees (while deletions are weak and performed logically): we only require a greedy post-processing for keeping the nodes in the blocks of the external memory. We adopt the external memory model [2] as explained in Chapter 2 to evaluate the I/O complexity, using $B$ to denote the block size of the data transfers between main and external memory. The I/O complexity accounts for the number of block transfers performed during the computation.

As stated in Chapter 3, there is a simple property that for an AVL tree of height $H$, the topmost $\lceil H/2 \rceil$ levels form a complete binary tree. To apply the core partitioning we can choose any $h^* \leq \lceil H/2 \rceil$. As a result, for $n \geq B$, the AVL tree can be decomposed with $h^* = \log \sqrt{B}$ into cores with a constant size ($B$), we call them here hats, obtaining a hat partitioning scheme.

The maintenance algorithm is using the basic idea of core partitioning; when performing the insertions, we proceed as usual for AVL trees by inserting a new leaf and traversing the critical path, except that we perform a post-processing by recomputing the partition into hats. When
performing deletions, we mark the searched key as logically deleted, and periodically rebuild when a constant fraction of the updates are deletions. We prove that the hat partitioning can be maintained in constant average time per update using our probability model. Our analysis is based on the fact that the expected cost is constant under the assumption that rebalancing a node has a cost proportional to its subtree size. Mehlhorn and Tsakalidis in [70] showed that the total rebalancing cost for a sequence of $n$ arbitrary insertions is at most $2.618n$. For random insertions the bound is improved to $2.26n$. In this work we improve that result by showing that this cost is constant in average.

Comparing our partitioning problem to similar previous work, we observe that the randomized search trees in [93] have an expected cost of $O(\log n)$ for a rotation, when the cost of the rotation is proportional to the subtree size of the rotated node. If we apply this analysis to our partitioning scheme the expected cost is $O(\log n)$ instead of the $O(1)$ that we propose in this work. We should observe that the former bound is for a randomized algorithm while the latter is the average-case analysis of our deterministic scheme. What characterizes our approach is also that it is very simple and practical with constant sizes of cores, and non-invasive for the original algorithms. With our hat partitioning scheme the original AVL trees can actually meet space and external memory efficiency in a unified way.

In the rest of the chapter, in Section 5.1, we present the random model adopted in this work. In Section 5.2, hat partitioning scheme is defined, then we study hat partitioning in external memory model in Section 5.3, and in Section 5.4, we show how to maintain the structure during a sequence of insertions and deletions. Finally, average time complexity of maintaining the structure is discussed in Section 5.5.

### 5.1 Random Model

To compute the expected cost for maintenance of the structure, we needed to choose a probability model. Among prevalent probability models for search trees, besides the random permutation model, the uniform model has been used in many papers such as in [39, 27, 34, 38]. The uniform model assumes that search trees of the same size are uniformly distributed. In this work, we adopt a similar uniform model where is assumed that AVL trees of the same height are uniformly distributed.
5.2 Hat Partitioning Scheme

For an arbitrary AVL tree with \( n \) nodes, we adopt the standard terminology [57] where the height
\( H = O(\log n) \) is the number of levels of nodes (so \( H = 1 \) for the tree with a single node). Recall
the following fact from Chapter 3.

**Fact 7** Consider an AVL tree of height \( H \). Then, the nodes on its topmost \( \lceil H/2 \rceil \) levels form a
complete balanced binary tree.

**(Hat partitioning scheme.** Hat partitioning scheme is briefly, core partitioning scheme when
we choose a constant core size \( B \). Or more formally it is defined as follows. Given a tree \( T \) of \( |T| \)
nodes and an integer \( B \) that is a power of 2, where \( B \) is the external memory block size, the hat
partitioning scheme consists of the following steps.

- If \( |T| \geq B \), conceptually remove the top subtree of height \( \log \sqrt{B} = \frac{1}{2} \log B \) (made up of the
topmost \( \frac{1}{2} \log B \) levels), which is a complete balanced binary tree of \( \sqrt{B} - 1 \) nodes (called
hat). Observe that since \( |T| \geq B \), \( \frac{1}{2} \log B \leq \lceil H/2 \rceil \).

- Recursively perform the hat partitioning of the bottom subtrees thus obtained, where each
of the bottom subtrees can potentially have different height or size (e.g., this happens for a
Fibonacci tree).

- If \( |T| < B \), it means that we have reached a subtree with a small enough size so that we can
stop the recursion (we call this final subtree a bottom subtree).

**Decomposition into hats.** As a result of our hat partitioning scheme, an AVL tree can be
seen as conceptually decomposed into a collection of complete balanced binary trees called hats,
where each hat is the top complete balanced binary tree that is obtained from the recursive hat
partitioning scheme applied to its subtree. Same as before, two hats are linked together if and only
if there is one node in one of the two hats that is linked to a node in the other hat, where one of
the two nodes is the root of the hat and the other is a leaf of the other hat.

Consider any root-to-leaf path and let \( C_1, C_2, \ldots, C_{t-1}, B_t \) be the subtrees thus traversed when
\( n > 1 \): here \( C_1 \) is the hat containing the root of the AVL tree, \( B_t \) is the (possibly empty) bottom
subtree of size at most \( B - 1 \) at the end of the path, and \( C_2, \ldots, C_{t-1} \) are the hats traversed when
going from \( C_1 \) downward to \( B_t \). We say that hat \( C_i \) is at level \( i \) to indicate that the path from the
root of the AVL tree to any descendant of \( C_i \) (nodes in \( C_i \) included) must traverse \( C_1, C_2, \ldots, C_i \).
**Theorem 6** Given a hat partitioning for an AVL tree of \(n\) nodes with block size \(B\), we can store its nodes using \(O(n/B)\) blocks. Each search operation requires \(O(\log n)\) comparisons and \(O(\log_B n)\) I/O operations in the worst case, and each update (insertion/deletion) operation has a cost that is equal to the searching cost plus \(O(1)\) expected block transfers in our model.

### 5.3 AVL Trees as External-Memory Search Trees

In the decomposition described in Section 5.2 each hat has a constant size of \(\sqrt{B} - 1\) so every \(\sqrt{B} + 1\) hats take exactly \(B - 1\) space and can be easily allocated in a block of size \(B\) instead of allocating memory to individual nodes. The hats are packed together in a greedy fashion from left to right into the blocks of size \(B\): namely, while the next hat can be stored in the current block, pack it in the block; otherwise, open a new block. The same procedure is used to store the bottom subtrees of size at most \(B - 1\) for which the recursion stops. It takes \(O(1)\) I/Os to traverse each hat or bottom subtree.

**Lemma 20** For an AVL tree of size \(n \geq B\), represented using the hat partitioning scheme in \(O(n/B)\) blocks of external memory, the search operation costs \(O(\log n)\) comparisons and \(O(\log_B n)\) I/O transfers.

**Proof:** In every search path from the root to a single leaf, using hat partitioning, the number of required comparisons do not change with respect to standard AVL trees, since we keep the main AVL tree structure and hat partitioning is just a layout over external memory. Hence, the number of comparisons for the search operation is \(O(\log n)\), as derived from the standard analysis of AVL trees.

To upper bound the number of I/O transfers, recall that in a path from the root to a leaf we first traverse some hats then we reach the bottom subtree containing the leaf. Each hat has size \(\sqrt{B} - 1\) and height \(\log(\sqrt{B}) = \frac{1}{2} \log B\), and needs at most one I/O transfer to be traversed (this happens when all hats in the search path are stored in different memory blocks) and the bottom subtree also needs just one I/O transfer. Since the height of an AVL tree is \(\Theta(\log n)\), the maximum number of I/O transfers for a search operation will be \(\frac{\Theta(\log n)}{\Theta(\log_B n)} + 1 = \Theta(\log_B n)\). \(\square\)
5.4 Maintaining the Hat Partitioning Scheme

Maintaining the hat partitioning scheme is the same as maintaining core partitioning scheme explained in Chapter 3.

Let \( v \) be a node of an AVL tree, and let \( \text{key}(v) \) and \( b(v) \) be its key and balance factor, respectively. A node is balanced (or unbalanced) if its balance factor is 0 (or \( \pm 1 \)). According to the standard insertion algorithm for AVLs as described in [57], after the insertion of the new node three different situations can occur, namely:

- absorption;
- rotation at the critical node (single or double);
- height increase.

Using the standard definition of AVL trees [57], the size of a tree is defined as the number of nodes (ignoring null pointers). While performing an insertion of a new node \( f \) into an AVL tree \( T \), a new leaf is created with balance factor 0 and its critical node will be defined as the topmost closest ancestor whose balance factor is \( \pm 1 \), or the root if all the ancestors have balance factor 0.

More formally, let \( v_0 \), \( v_1, \ldots, v_k \) be the search path of \( \text{key}(f) \) (so \( v_0 \) will be the null pointer) and \( i \) be the maximum index such that \( b(v_i) = b(v_{i-1}) = \ldots = b(v_0) = 0 \). Then \( v_i, \ldots, v_0 \) is called critical path and the node \( v_{i+1} \) (if any) will be the critical node, if there is no such a \( v_{i+1} \) then we consider the root as critical node (this case happens only in height increase).

Finally, \( l_f \) denotes the length of this critical path, thus \( l_f = i \) [70].

Given a node \( z \) that is the root of the hat containing the critical node or the root of a new born hat (this happens when the size of the subtree rooted in \( z \) reaches \( B \)). The operator \text{repartition}(u)\) and its three main events were defined in Chapter 3, recall that we have:

- \text{repartition}(u): (1) define a new hat or rebuild the hat \( C \) that contains \( u \), and (2) find \( u_1, \ldots, u_{\sqrt{B}} \), the topmost descendants of \( u \) that are not in \( C \), and locally recompute the hat partitioning for each node \( u_i \) if needed (\( i = 1, \ldots, \sqrt{B} \)).

We proceed as usual for AVL trees by inserting a new leaf \( f \) and traversing the critical path towards the root, possibly performing a (single or double) rotation. After that, we know the critical node \( v_{i+1} \) (if it exists). We also find the topmost (closest to the root) ancestor \( w \) (if it exists) of \( f \) such that \( w \) is the root of a new hat born because of the insertion of \( f \). If neither \( v_{i+1} \) nor \( w \) exist, return; otherwise, let \( u \) be the topmost between \( v_{i+1} \) and \( w \), and perform \text{repartition}(u).
Recall from Chapter 3, that the three main events of \texttt{repartition}(u) for HAT partitioning are the following.

- **subtree rescan**: if the critical node \( v_{i+1} \) exists and \( u = v_{i+1} \), this accounts for (2) in \texttt{repartition}(u), with a cost proportional to the size of the subtree rooted at the critical node \( v_{i+1} \).

- **hat rescan**: if the critical node \( v_{i+1} \) exists and \( u = v_{i+1} \), this accounts for (1) in \texttt{repartition}(u), with a cost proportional to the size \( \sqrt{B} - 1 \) of the hat.

- **hat birth**: if \( w \) exists and \( u = w \), this accounts for (1) in \texttt{repartition}(u), with a cost proportional to the size \( \sqrt{B} - 1 \) of the new hat rooted at the ancestor \( w \).

Note that whenever \( u = w = v_{i+1} \), the only operation needed is the third one. Also observe that the cost of **subtree rescan** dominates that of **hat rescan** and **hat birth**. Now, let us assume that the value \( B \) that defines the upper bound on the bottom subtrees’ sizes corresponds to the block size of the data transfers between main and external memory.

**Lemma 21** When \texttt{repartition}(u) is applied to a node \( u \), let \( s \) be the size of the subtree rooted at \( u \) (if **subtree rescan** occurs), while \( \sqrt{B} - 1 \) is the size of the hat in both operations **hat rescan** and **hat birth**. Then, the cost is \( O(s) \) time and \( O(s/\sqrt{B}) \) block transfers, where \( B \) is the block size.

**Proof**: An \( O(s) \)-time algorithm can rebuild the hat partitioning as required by \texttt{repartition}(u). Recall that \( s \geq B \), otherwise the partition scheme is not computed. Moreover, the number of hats is \( O(s/\sqrt{B}) \), and scanning them takes at most so many block transfers. Moreover, the bottom subtrees are packed together in a greedy fashion into the blocks of size \( B \), so that scanning them all requires at most \( O(s/B) = O(s/\sqrt{B}) \) block transfers. \( \square \)

### 5.5 Average Time Complexity

In this section, we will show that the hat partitioning can be efficiently maintained in constant average time per update by simply repartitioning a portion of the tree after each update. Search, insertion and deletion are the main operations we need to support in our hat partitioned AVL tree structure. As we have observed earlier, the search operation is fast enough \((O(\log n))\) comparisons), but deletions are sometimes too expensive to make the hat partitioned AVL a dynamic data structure. We limit the operation set to search and insertion and we use logical deletion as before.
5.5. AVERAGE TIME COMPLEXITY

We also adopt a probability model which assumes that AVL trees of the same height are uniformly distributed. Let us consider the insertion operation of a new leaf \( f \) in a hat partitioned AVL tree \( T \). Recall from Section 5.4, after an insertion operation the primitive \( \text{repartition}(u) \) must be run, where \( u \) is the topmost node between the critical node and the topmost ancestor of the new inserted leaf \( f \) that is the root of a hat that changes its content because of a rebalancing operation, or the root of a born new hat. We need some preliminary lemmas.

**Lemma 22** In the worst case, the cost of subtree rescan is \( O(2^{l_f}) \), where \( l_f \) is the length of the critical path.

**Proof:** After insertion of a new node in the correct position, as illustrated on Figures 5.1 and 5.2, the height of the critical node subtree is \( l_f + 1 \) in cases of height increase and rotation and \( l_f + 2 \) in case of absorption. Hence the total number of nodes in the critical node subtree can not be more than \( 2^{l_f+2} - 1 \), therefore, the cost of rescan and rebuilding the entire critical node subtree is \( O(2^{l_f+2}) = O(2^{l_f}) \). \( \square \)

Let us now find an upper bound for the probability of executing a subtree rescan on a subtree of height \( l_f \). Let \( S_h \) be the number of AVL trees of height \( h \), and let \( P^b_h \) and \( P^{\pm1}_h \) be the probabilities that an arbitrary AVL tree of height \( h \) is balanced or unbalanced at the root, respectively. We have:

![Figure 5.1: a) Insertion of new node \( f \) with absorption. b) Insertion of new node \( f \) with height increase.](image)
Figure 5.2: Critical path before and after insertion of $f$ in single rotation (top) and double rotation (bottom).
5.5. AVERAGE TIME COMPLEXITY

Lemma 23 \( S_h = S_{h-1} S_{h-1} + 2 S_{h-1} S_{h-2}, \) \( S_0 = S_1 = 1, \) \( S_2 = 3. \)

**Proof:** An AVL tree of height 0 is an empty tree, so \( S_0 = 1. \) The other initial conditions can be verified by direct inspection. For \( h > 2, \) observe that the subtrees of the root can be both of height \( h - 1, \) or the left subtree has height \( h - 1 \) and the right one \( h - 2, \) or vice versa. \( \square \)

Lemma 24 There exist two positive constants \( \epsilon \leq 0.5 \) and \( \theta \leq 0.25 \) such that, for any \( h \geq 4: \)

\[
2^{2^{h-1}} \leq S_h \leq \epsilon 2^{2^{h-1+\theta}}.
\]

**Proof:** By induction on \( h. \)

**Base:** for \( h = 4 \) and \( h = 5, \) by Lemma 23 we have \( S_4 = 315 \) and \( S_5 = 108675, \) so \( 2^{2^3} \leq S_4 \leq \epsilon 2^{2^3+\theta} \) and \( 2^{2^4} \leq S_5 \leq \epsilon 2^{2^4+\theta} \) for \( \epsilon = 0.5 \) and \( \theta = 0.25. \)

**Induction step:** We assume \( 2^{2^{k-1}} \leq S_k \leq \epsilon 2^{2^{k-1+\theta}} \) for any \( 4 \leq k \leq h. \) By Lemma 23 we have \( S_{h+1} = S_h S_{h-1} + 2 S_h S_{h-1}. \) Thus

\[
2^{2^{h-1}} 2^{2^{h-1}} + 2^{2^{h-1}} 2^{2^{h-2}} \leq S_{h+1} \leq \epsilon^2 2^{2^{h-1+\theta}} 2^{2^{h-1+\theta}} + 2 \epsilon^2 2^{2^{h-1+\theta}} 2^{2^{h-2+\theta}},
\]

\[
2^{2^h} + 2^{2^{h-1}} 2^{2^{h-2}} \leq S_{h+1} \leq \epsilon^2 2^{2^{h+\theta}} + \epsilon^2 2^{(2^{h+\theta} - 2^{h+\theta} + 1)},
\]

\[
2^{2^h} \leq S_{h+1} \leq 2 \epsilon^2 2^{2^{h+\theta}}.
\]

Since \( \epsilon \leq 0.5, \) \( \epsilon^2 \leq \frac{1}{2} \epsilon, \) and therefore we have \( 2^{2^h} \leq S_{h+1} \leq \epsilon 2^{2^{h+\theta}}. \) \( \square \)

**Corollary 4** There exists \( \lambda \geq 0.43 \) such that for any \( h \geq 4, \) the probability \( P_h^{\pm 1} \) that an AVL tree of height \( h \) is unbalanced at root is upper bounded by:

\[
P_h^{\pm 1} \leq \frac{\epsilon}{2^{h-3}}.
\]

**Proof:** Consider an AVL tree of height \( h. \) The probability that the root is unbalanced is given by the ratio between the number of all AVL trees of height \( h \) unbalanced at the root and the number of all AVL trees of height \( h. \) Hence

\[
P_h^{\pm 1} = \frac{2S_{h-2} S_{h-1}}{S_h} \leq \frac{2 \epsilon^2 2^{2^{h-3+\theta}} 2^{2^{h-2+\theta}}}{2^{2h-1}} \leq \frac{\epsilon}{2^{2h-3(2^1+\theta) - 3}}.
\]

Letting \( \lambda = 2^2 - 2^{1+\theta} - 2^\theta, \) since \( \theta \leq 0.25 \) the thesis immediately follows. \( \square \)
Now let $v$ be a node in an AVL tree. Define $E^0_v (E^\pm_0)$ as the event that $v$ is balanced (unbalanced). Consider a critical path $v_1, v_2, ..., v_t, v_0$ of length $l_f = t$, where each $v_i$ has height $i$ ($v_0$ is the null pointer corresponding to the position of the new inserted key). Let $P(l_f = t)_{\text{abs/rot}}$ denotes the probability that $l_f = t$ in the cases of rotation or absorption in an AVL tree of height $> t$. In the following lemmas we evaluate $P(l_f = t)_{\text{abs/rot}}$:

$$P(l_f = t)_{\text{abs/rot}} = P(E^0_v) \cdot P(E^0_{v_1} | E^0_v) \cdot P(E^0_{v_2} | \{E^0_{v_1} \cap E^0_v\}) \cdot P(E^0_{v_{t+1}} | \{E^0_{v_t} \cap ... \cap E^0_v\}).$$

**Lemma 25** Let $v$ be the root of an AVL tree $T$ of height $h$ and $w$ be the left/right child of $v$. Then $P(E^0_v | E^0_v) = P(E^0_w)$, hence $E^0_v$ and $E^0_w$ are independent events.

**Proof:** Without loss of generality we assume that $w$ is the left child of $v$. We have $P(E^0_v) = \frac{S_{h-1}S_{h-2}}{S_h}$, $P(E^0_w) = \frac{S_{h-2}S_{h-3}}{S_{h-1}}$, and $P(E^0_v \cap E^0_w) = \frac{S_{h-2}S_{h-3}}{S_h}$, therefore $P(E^0_v | E^0_w) = P(E^0_v \cap E^0_w) / P(E^0_w) = P(E^0_v)$, which means that $E^0_v$ and $E^0_w$ are independent events. 

**Corollary 5**

1. For $1 \leq i \leq t$, $P(E^0_{v_i} | \{E^0_{v_{i-1}} \cap E^0_{v_{i-2}} \cap ... \cap E^0_v\}) = P(E^0_{v_i}) = P^0_i$.

2. In case of absorption or rotation, for the critical node $v_{t+1}$ we have:

$$P(E^\pm_{v_{t+1}} | \{E^0_{v_t} \cap E^0_{v_{t-1}} \cap ... \cap E^0_v\}) \leq \frac{1}{2}(P^\pm_{t+1} + P^\pm_{t+2}) < P^\pm_{t+1}.$$

**Proof:** The first case follows immediately from Lemma 25 and from the fact that for any node $v_i$ on the critical path, $1 \leq i \leq t$, the fact that $h(v_i) = i$ implies $P(E^0_v) = P^0_i$ (by definitions of $E^0_v$ and $P^0_h$).

In the second case, since $v_{t+1}$ is unbalanced and $v_t$ has height $t$, the height of $v_{t+1}$ can be $t+1$ or $t+2$, hence

$$P(E^\pm_{v_{t+1}} \cap \{E^0_{v_t} \cap ... \cap E^0_v\}) = \frac{(S_0 \cdot S_0 \cdot S_1 \cdot ... \cdot S_{t-1}) \cdot S_t - 1}{S_{t+1}} + \frac{(S_0 \cdot S_0 \cdot S_1 \cdot ... \cdot S_{t-1}) \cdot S_{t+1}}{S_{t+2}}$$

$$= (S_0 \cdot S_0 \cdot S_1 \cdot ... \cdot S_{t-1}) \cdot \left( \frac{S_t}{S_{t+1}} + \frac{S_{t+1}}{S_{t+2}} \right).$$
On the other hand:
\[
P(E^0_i \cap \ldots \cap E^0_v \cap E^0_v) = P^0_1 \cdot P^0_{i-1} \cdot P^0_v = \frac{S_{i-1} \cdot S_{i-1} \cdot S_{i-2} \cdot \ldots \cdot S_0 \cdot S_0}{S_i}
\]
and
\[
P(E^\pm_1 \mid \{E^0_i \cap \ldots \cap E^0_v\}) = P(E^\pm_1 \cap E^0_v \cap \ldots \cap E^0_v) / P(E^0_v \cap \ldots \cap E^0_v).
\]
Therefore
\[
P(E^\pm_1 \mid \{E^0_i \cap \ldots \cap E^0_v\}) = \frac{S_{i-1} S_i}{S_{t+1}} + \frac{S_{t+1} S_i}{S_{t+2}} = \frac{1}{2}(P^\pm_{t+1} + P_{t+1}).
\]
Since \(P^\pm_{t+1}\) is a decreasing function, we have \(P(E^\pm_1 \mid \{E^0_i \cap \ldots \cap E^0_v\}) < P^\pm_{t+1}.\)

**Corollary 6** The probability that the critical path \(l_f\) has length equal to \(t\) in the case of absorption or rotation, is
\[
P(l_f = t)_{abs/ro} < P^\pm_{t+1} \cdot \prod_{i=0}^{t} P^0_i.
\]

**Proof:** The thesis follows since
\[
P(l_f = t)_{abs/ro} = P(E^0_v) \cdot P(E^0_v \mid E^0_v) \cdot P(E^0_v \mid \{E^0_i \cap E^0_v\}) \cdot \ldots
\]
\[
\ldots \cdot P(E^\pm_1 \mid \{E^0_i \cap \ldots \cap E^0_v\})
\]
\[
= P(E^0_v) \cdot P(E^0_v) \cdot \ldots \cdot P(E^0_v) \cdot P(E^\pm_1 \mid \{E^0_v \cap E^0_{v-1} \cap \ldots \cap E^0_v\})
\]
\[
< P^0_v \cdot P^0_1 \cdot P^0_2 \cdot \ldots \cdot P^0_t \cdot P^\pm_{t+1} = P^\pm_{t+1} \cdot \prod_{i=0}^{t} P^0_i
\]
\[
< P^\pm_{t+1} \cdot \prod_{i=0}^{t} P^0_i.
\]

Let \(E[size_h]\) denotes the expected value of the number of nodes (size) in an AVL tree of height \(h\). In other word:
\[
E[size_h] = \frac{\sum T \in AVL\ tree\ of\ height\ h \cdot |T|}{S_h}.
\]

**Lemma 26** \(E[size_0] = 0, E[size_1] = 1\) and for \(h > 1\), there are two constants \(\delta_1 > 1.44\) and \(\delta_2 < 2\) such that \(\delta_1 E[size_{h-1}] < E[size_h] < \delta_2 E[size_{h-1}].\)
Proof: For \( h = 2 \) and \( h = 3 \) the thesis can be verified by direct inspection. For bigger values, let us consider an AVL tree \( T \) of height \( h \geq 4 \); the root of \( T \) is either balanced with probability \( P_0^h \) or unbalanced with probability \( P_1^h \). So the expected value is given by

\[
E[\text{size}_h] = P_0^h \cdot 2 \cdot E[\text{size}_{h-1}] + P_1^h \cdot (E[\text{size}_{h-1}] + E[\text{size}_{h-2}]).
\]

On the other hand, \( P_0^h = 1 - P_1^h \) and, using Corollary 4, it can be easily verified that, for \( h \geq 4 \), \( P_1^h < 0.28 \), so that \( P_0^h > 0.72 \). Therefore

\[
0.72 \cdot 2E[\text{size}_{h-1}] < E[\text{size}_h] < 2E[\text{size}_{h-1}] + P_1^h (E[\text{size}_{h-1}] + E[\text{size}_{h-2}]).
\]

This means that \( E[\text{size}_h] \) is geometrically increasing. \( \Box \)

**Theorem 7** For any insertion operation in a hat partitioned AVL tree of \( n \) nodes, the average time complexity \( T_a(n) \) for subtree rescan (repartitioning of the critical node subtree) is constant.

**Proof:** Let \( T'_a(n) \) denote the average time complexity in case of absorption or rotation and \( T''_a(n) \) denotes average time complexity in the case of height increase (the entire tree height increase).

Observe that \( T_a(n) \leq T'_a(n) + T''_a(n) \); we will show both values \( T'_a(n) \) and \( T''_a(n) \) are constants.

Recall from [57] that the height of an AVL tree with \( n \) nodes is at most \( 1.44 \log n \); on the other hand by Lemma 22, the cost of the insertion of a new node \( f \) is \( O(2^t) \), therefore

\[
T'_a(n) \leq \sum_{t=1}^{1.44 \log n} P(l_f = t)_{\text{abs/ro}} O(2^t).
\]

Then, by Corollary 4 and 6, we have

\[
T'_a(n) \leq \sum_{t=1}^{1.44 \log n} \left( \frac{P_1^{t+1} \cdot \prod_{i=0}^{t} P_i^0}{2^t} \right) \cdot O(2^t).
\]

Hence, there exists a constant \( c \) such that:

\[
T'_a(n) \leq \sum_{t=1}^{1.44 \log n} \left( \frac{c}{2^t} \right) \cdot O(2^t).
\]
Letting \( a(t) = \frac{c}{2^{t-1} \cdot 2^{\lambda}} \), we get \( \frac{a(t+1)}{a(t)} = \frac{2^{t-2} \cdot 2^{\lambda}}{2^{t-1} \cdot 2^{\lambda}} = \frac{1}{2} \). For large values of \( t \), \( \frac{a(t+1)}{a(t)} \ll 1 \), therefore \( T_a'(n) \) is a convergent series and \( T_a''(n) = O(1) \).

Let us now evaluate \( T_a''(n) \). By definition, \( T_a''(n) \) is the average time complexity of subtree rescan of the whole AVL tree in case of height increase of the entire tree. To upper bound \( T_a''(n) \), we consider the insertion a sequence of \( n \) keys in an initially empty tree and we compute the average time complexity considering only the height increase events.

Let \( H \) denote the height of the tree after \( n \) insertions. Since \( H < 1.44 \log n \), the number of height increase events is at most \( 1.44 \log n \). In case of height increase, if \( h - 1 \) and \( h \) denote the height of the entire tree before and after this event, then the expected value of the size of the AVL tree right after the height increase is upper bounded by \( E[\text{size}_h] \) (by definition). Therefore, for these \( n \) insertions, \( T_a''(n) \leq \sum_{x=1}^{H(n)} E[\text{size}_x] \) and since \( E[\text{size}_h] \) is geometrically increasing we have:

\[
T_a''(n) = \frac{\Theta(E[\text{size}_H])}{n} = \Theta(n) = O(1).
\]

Finally since \( T_a'(n) = O(1) \) and \( T_a''(n) = O(1) \), \( T_a(n) = O(1) \). \( \square \)

**Theorem 8** For any insertion operation in a hat partitioned AVL tree of \( n \) nodes, the average time complexity for hat rescan (repartitioning of the hat) is constant.

**Proof:** We will prove that the expected value of the cost of hat rescan is of the same order as the average cost of subtree rescan for each insertion. For this reason, consider an operation where the critical node is on the \( \ell \)th level of a hat \( C \) of a subtree \( T_z \) rooted in a node \( z \). As shown in Figure 5.3, subtrees rooted at the \( \ell \)th level of \( C \) have size \( n_1, n_2, \ldots, n_j \); let \( X_i \) denotes the event that the root of the \( i \)th such subtree is the critical node. If \( m \) is the size of \( T_z \), \( \sum_{i=1}^{j} n_i \leq m \). Without loss of generality let us assume \( m = 2^{2k} \) for some natural number \( k \).

For any operation with the critical node on this level, hat rescan will cost \( \sqrt{B} - 1 \), while the expected value of the cost of subtree rescan on this level is given by \( \sum_{i=1}^{j} P(X_i) n_i \), where \( P(X_i) \) is the probability of the event \( X_i \).

To find a lower bound for \( P(X_i) \), we assume as usual that the probability of inserting the new node into a subtree is proportional to the number of leaves of the subtree divided by the total number of leaves of \( T_z \), note that this does not interfere with our model of AVL trees of same height are uniformly distributed. Since in any AVL tree the number of leaves is within two
constant fractions of the number of nodes, we have that

\[ c_1 \frac{n_i}{m} \leq P(X_i) \leq c_2 \frac{n_i}{m}, \]

for two constant values \( c_1 \) and \( c_2 \). Therefore the expected value of the cost of \textit{subtree rescan} on the \( \ell \)th level of \( C \) will be

\[
\sum_{i=1}^{j} P(X_i) n_i \geq \sum_{i=1}^{j} c_1 \frac{n_i}{m} n_i \geq c_1 \frac{m}{j} \sum_{i=1}^{j} (n_i)^2.
\]

Since \( \sum_{i=1}^{j} (n_i)^2 \) is minimum when all the terms \( n_i \) are equal, we get

\[
\sum_{i=1}^{j} P(X_i) n_i \geq c_1 \frac{m}{j} \sum_{i=1}^{j} (n_i)^2 \geq c_1 \frac{m}{j} (\frac{m}{j})^2 \geq c_1 \frac{m}{j},
\]

and since \( j \leq \sqrt{B} \), we finally have

\[
\sum_{i=1}^{j} P(X_i) n_i \geq c_1 \frac{m}{j} \geq 2c_1 \frac{m}{\sqrt{B}} \geq 2c_1 \sqrt{B};
\]

since \( m \geq B \). This means that the expected value of the cost of \textit{subtree rescan} on each level of the hat \( C \) is greater than or equal to a constant factor of \( (\sqrt{B} - 1) \), where \( \sqrt{B} - 1 \) represents the cost of \textit{hat rescan}. Therefore, the average cost of \textit{hat rescan} can be covered by the average cost of \textit{subtree rescan}, which is constant.

\[ \Box \]

Finally observe that for any hat, the \textit{hat birth} operation occurs just once when the hat is created.
and it costs $O(\sqrt{B})$. Indeed, once a hat has been created, it will never disappear from the structure (ignoring the entire tree rebalancing caused by deletions). Therefore, the simplest way to compute the average cost of this case is to sum all hat birth operations costs during building an AVL tree of size $n$ (by inserting $n$ keys to an empty tree) and divide it by $n$, the result will be in fact amortized cost of hat birth which as we know is an upper bound of the average cost. Now starting from an empty tree, after a sequence of $n$ random insertions, the final tree has at most $O(n/\sqrt{B})$ hats, and each of them had a hat birth cost of $O(\sqrt{B})$, so the total cost of hat birth for $n$ insertions will be $O(\sqrt{B} \cdot n/\sqrt{B}) = O(n)$ which is constant for any single insertion.

Let us finally analyze the average number of I/O operations for the external memory data structure.

**Theorem 9** The average number of I/O operations required for an insertion of a new key into the hat partitioning scheme of an AVL tree is constant.

**Proof:** Since the operations of subtree rescan, hat rescan, and hat birth require constant time in average, their cost in terms of I/O operations is $O(1/\sqrt{B} + 1)$, since in the worst case we need to access one block for each hat plus one more access for the bottom subtree of size $O(B)$. The overall cost is then constant. □

### 5.5.1 Improved Hat Partitioning Scheme

We can improve the hat partitioning scheme in practice by observing the following property. Given a hat $C$ with root $z$, let $v_1, v_2, \ldots, v_{\sqrt{B}}$ be the topmost descendants of $z$ that are not in $C$. If each $v_i$ is also the root of a hat, the hat $C$ and all hats, rooted in $v_1, v_2, \ldots, v_{\sqrt{B}}$, form a complete balanced binary tree of height $\log B$ and size $B - 1$. Thus, we can merge them together in a big hat, called $B$-hat (see Figure 5.4). This improved scheme will result in an slightly more efficient data structure for external memory, as the hats can be nicely packed together into an external memory block, thus guaranteeing a reduced number of I/Os, still within the same asymptotic bound given for the simple hat scheme.

### 5.6 Summary

We proved in Chapter 3, that core partitioning on AVL trees can not be maintained in amortized logarithmic time. In this chapter, we showed that core partition idea can be applied to AVL trees
with constant average time on a random memory model assuming that AVL trees of the same height are uniformly distributed. On this given core partitioning, all the cores had constant size of $B$, and they were called hats, obtaining a hat partitioning scheme. For $n$ keys and block size $B$, in external memory model, the searching cost is $O(\log_B n)$ block transfers in the worst case, while the updating cost is equal to the searching cost plus $O(1)$ expected block transfers (for restructuring the tree).
Chapter 6

Other Properties of AVL Trees

In this chapter, we present some new features and properties of AVL trees. In Section 6.1, we first present our new Gap Terminology [4], where gaps are special tree edges such that the height difference between the subtrees, rooted at their two endpoints, is equal to 2. Using this terminology we then present the Basic-Theorem that can represent any AVL tree (and its subtrees) with a series of powers of 2 (of the heights of the gaps) instead of classic representation of AVL trees. Basic-Theorem characterizes the tree size of any AVL tree with a very simple formula. We also investigate that how gaps change during a sequence of insertions and deletions. In Section 6.2 and 6.3, we answer to the question whether deletions can take many rotations not only in the worst case, but also in the amortized case as well, when insertions are intermixed with deletions. Heaupler, Sen, and Tarjan [50] conjectured that alternating insertions and deletions in an $n$-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. We first study Rank terminology for AVL trees presented by Heaupler, Sen, and Tarjan in that paper. Then we provide a construction which causes each deletion to do $\Omega(\log n)$ rotations [5].

6.1 GAP Terminology

Let $T$ be an AVL tree of size $n$ and let $v$ be a given node.

- $|T|$ and $V(T)$ denote the size of tree $T$ and its set of nodes, respectively.
- $T_v$ denotes the subtree rooted in $v$,
- $p(v)$ and $\text{child}(v)$ denote parent and child of $v$, respectively, and $v.r$ and $v.l$ denote the right and left child of $v$, respectively,
• $h(v)$ denotes the height of $T_v$,

• $\text{lev}(v)$, denotes the number of nodes above $v$ plus one (i.e., the number of ancestors of $v$ including $v$).

**Definition 4** The balance factor of $v$ is the difference in heights of its two subtrees ($h_R - h_L$). The balance factor (bf) of the nodes of an AVL tree may take one of the values -1, 0, +1.

**Definition 5** For any pair of nodes $v$ and $w$, $w$ is a child of $v$, then the edge between $v$ and $w$ is called a **gap** iff the height difference between $v$ and $w$ is equal to 2. If there is a gap $g$ between $v$ and $w$ then we say $v$ has a gap child, and $v$ and $w$ are called parent and child of this gap.

For gap $g$ by $p(g)$, child($g$), and $h(g)$ we denote the parent, the child, and the height of $g$ respectively, where we define $h(g) = h(\text{child}(g))$. We also use $\text{GAP}(T)$ to denote the set of all the gaps in a tree $T$.

**Corollary 7** For any given node $v$ in an AVL tree $T$:

- There is at most one gap child for $v$ either between $v$ and $v.l$ or $v$ and $v.r$.

- Leaves have no gap children.

For node $v$, let $\text{key}(v)$ and $b(v)$ be its key and balance factor, respectively. A node is **balanced** (or **unbalanced**) if its balance factor is 0 (or $\pm 1$).

**Lemma 27** In a given AVL tree $T$ of size $n$ there are at most $\frac{n-1}{2}$ gaps, this bound is tight and can happen on trees similar to Fibonacci trees.

**Proof**: In any AVL tree of size $n$ there are $\frac{n+1}{2}$ leaves, which by Corollary 7, they can not have any gaps, and there are $\frac{n-1}{2}$ internal nodes, which can have at most one gap, therefore, any AVL tree of size $n$ has at most $\frac{n-1}{2}$ gaps. This bound is also tight for any fully **unbalanced** AVL tree (e.g., Fibonacci trees) where all internal nodes are unbalanced. □

According to the standard algorithms of AVL trees described in [57], two main operations which can change AVL tree’s structure are deletion and insertion. We are going to study gap-properties for an AVL tree, in general and during these these two operations.

### 6.1.1 General Properties of Gaps

The following theorem expresses the size of a given AVL tree with height $H$ in terms of the powers of 2 of the heights of the gaps.
Theorem 10 Basic-Theorem:

$$|T| = n = 2^H - 1 - \sum_{g \in \text{GAP}(T)} 2^{h(g)}.$$

Proof:

By induction on $H$. For $H = 0$ (empty tree) and $H = 1$ (just one node), the theorem trivially holds. For the inductive step on $H \geq 2$, we assume that the theorem holds for any AVL tree of height $h < H$, then we use this assumption to prove the statement for height $H$. Let $T_l$ and $T_r$ denote the left and right subtree of the root of the given AVL tree $T$, respectively. We consider two cases for $T$.

First suppose that the height of $T_l$ and $T_r$ are equal to $H - 1$. Then, $\text{GAP}(T) = \text{GAP}(T_l) \cup \text{GAP}(T_r)$ as the two edges between the root of $T$ and the roots of $T_l$ and $T_r$ are not gaps, and the theorem easily follows using the induction hypothesis,

$$|T| = |T_l| + |T_r| + 1 = 2^{H-1} - 1 - \sum_{g \in \text{GAP}(T_l)} 2^{h(g)} + 2^{H-1} - 1 - \sum_{g \in \text{GAP}(T_r)} 2^{h(g)} + 1$$

$$= 2^H - 1 - \sum_{g \in \text{GAP}(T)} 2^{h(g)}.$$

Now suppose that two subtrees have different heights $H - 1$ and $H - 2$. Then the set of gaps of $T$ contains all gaps in $T_l$ and $T_r$, plus the new gap $g'$ given by the edge between the root of $T$ and the root of the subtree of height $H - 2$. Therefore, using the induction hypothesis and the fact that $h(g') = H - 2$, we have:

$$|T| = |T_l| + |T_r| + 1 = 2^{H-1} + 2^{H-2} - 1 - \sum_{g \in \text{GAP}(T_l)} 2^{h(g)} - \sum_{g \in \text{GAP}(T_r)} 2^{h(g)}$$

$$= 2^H - 2^{H-2} - 1 - \sum_{g \in \text{GAP}(T_l)} 2^{h(g)} - \sum_{g \in \text{GAP}(T_r)} 2^{h(g)}$$

$$= 2^H - 1 - 2^{h(g')} - \sum_{g \in \text{GAP}(T_l)} 2^{h(g)} - \sum_{g \in \text{GAP}(T_r)} 2^{h(g)} = 2^H - 1 - \sum_{g \in \text{GAP}(T)} 2^{h(g)}.$$

To show how powerful is this theorem, in a given AVL tree, the following corollary can describe the precise relationship between size ($n$) and the heights of the nodes, subtree sizes and the heights of the gaps.

Corollary 8 For a gap $g$ let us define $\text{lev}(g)$ as the number of nodes above $g$ in the path from
g to the root (i.e., the number of node-ancestors of g), note that the level of a gap is equal to the level of its parent node, then:

$$\sum_{u \in V(T)} (2^{h(u)} - |T_u|) - \sum_{g \in GAP(T)} lev(g)2^{h(g)} = n.$$ 

**Proof:** By Theorem 10 (Basic-Theorem) we know that for any node u, $2^{h(u)} - \sum_{g \in GAP(T_u)} 2^{h(g)} - |T_u| = 1$. Therefore, by summing up of this formula over all nodes we have:

$$\sum_{u \in V(T)} 1 = n = \sum_{u \in V(T)} \{2^{h(u)} - |T_u| - \sum_{g \in GAP(T_u)} 2^{h(g)}\}.$$ 

On the other hand for any gap g, for any ancestor u of g, $g \in GAP(T_u)$ and vice versa. Therefore, there are exactly $lev(g)$ nodes u which $g \in GAP(T_u)$. So we can claim that:

$$\sum_{u \in V(T)} \sum_{g \in GAP(T_u)} 2^{h(g)} = \sum_{g \in GAP(T)} lev(g)2^{h(g)},$$ 

Therefore:

$$n = \sum_{u \in V(T)} (2^{h(u)} - |T_u|) - \sum_{u \in V(T)} \sum_{g \in GAP(T_u)} 2^{h(g)},$$

$$n = \sum_{u \in V(T)} (2^{h(u)} - |T_u|) - \sum_{g \in GAP(T)} lev(g)2^{h(g)}. $$

\[\square\]

**Lemma 28** For any given AVL tree T with size n and height H:

1. 

$$\sum_{u \in V(T)} (|T_u|) \leq nH,$$

2. 

$$\sum_{u \in V(T)} (|T_u|) = \Theta(n \log n).$$

**Proof:** For each level, the total sum of subtree sizes of nodes embedded on that level of T, can not be bigger than n. Since we have at most H levels on T, $\sum_{u \in V(T)} (|T_u|) \leq nH$. On the other hand we can also show that the sum of subtree sizes of nodes embedded on for example the first $H/4$ levels of T is $\Theta(n)$ (by an induction using the fact that there are only a few nodes on
6.1. GAP TERMINOLOGY

the very first levels), then the total sum of subtree sizes of nodes in the first $H/4$ levels will be $\Theta(nH) = \Theta(n \log n)$. □

Corollary 9 The powers of 2 of the heights of nodes and gaps are related by the following upper bounds.

1. $\sum_{u \in V(T)} (2^{h(u)}) - \sum_{g \in GAP(T)} \text{lev}(g)2^{h(g)} \leq nH$,

2. $\sum_{u \in V(T)} (2^{h(u)}) - \sum_{g \in GAP(T)} \text{lev}(g)2^{h(g)} = \Theta(n \log n)$.

Proof: Immediately by using Corollary 8 and Lemma 28. □

6.1.2 Gaps in Insertions and Deletions

In this section, we study how gaps change during deletion and insertion operations of AVL trees.

According to the standard insertion algorithm of AVL trees described in [57], after the insertion of a new node, three different situations can occur, namely:

1. absorption;

2. rotation at the critical node (single or double);

3. height increase.

For the insertion of a new node $v$ into an AVL tree $T$, let $v_k$ denote the root of $T$, $v_k, v_{k-1}, \ldots, v_0$ be the insertion path of $key(v)$ and $i$ be the maximum index such that $b(v_i) = b(v_{i-1}) = \ldots = b(v_0) = 0$, recall that $b(v)$ is the balance factor of node $v$. Hence, $v_{i+1}$ (if any) will be called the critical node, and $v_i, \ldots, v_0$ will be the critical path. The length of the critical path is $l_v = i$. In case of height increase of $T$, $i = k$ and there exists no critical node.

First we consider insertion, as we know there are three possibilities (absorption, rotation and height increase), when one of these cases occurs, gaps inside the subtree of the critical node will change, some will disappear and some will be created. The following lemma studies this property.

Lemma 29 In an insertion of a new node $v$ with critical path of length $l_v$ and critical node $u$:
92 CHAPTER 6. OTHER PROPERTIES OF AVL TREES

• In case of a “height increase” (increasing the height of the whole tree), a sequence of gaps with heights from 0 to $l_v$ will be created as shown in Figure 6.1.b).

• In case of “absorption” one gap of height $l_v$ will disappear (will be removed) and a sequence of gaps with heights from 0 to $l_v - 1$ will be created, see Figure 6.1.a).

• In case of “rotation” (single or double) one gap of height $l_v - 1$ will disappear (will be removed) and a sequence of gaps with heights from 0 to $l_v - 2$ will be created, see Figure 6.2, and 6.3.

Proof: As illustrated in Figure 6.1.a), 6.2 and 6.3, in cases of absorption and rotation, there should be always a gap $g$ before performing insertion whose parent is $u$; obviously this gap disappears after absorption/rotation. The only case remaining is height increase; in this case, as illustrated in Figure 6.1.b), there is no critical node and there is no gap to disappear. In all cases the sequences of created gaps are shown in Figure 6.1, 6.2, and 6.3.

Definition 6 For a given gap $g$, ‘consuming’ $g$ means that this gap has disappeared from the tree, either by a rotation or an absorption, and a sequence of gaps as mentioned before has been ‘generated’.

Lemma 30 A gap $g$ or a sequence of gaps can be generated either by height increase or by consuming a gap above, as shown in Figure 6.1, 6.2, and 6.3.

Proof: Notice that except the case of height increase (the whole tree height increase) for generating a gap or a sequence of gaps, one gap above should be consumed.
6.1. GAP TERMINOLOGY

Figure 6.2: Gaps before and after single rotation.

Figure 6.3: Gaps before and after double rotation.
Now by ignoring deletions, starting from an initially empty tree and inserting an arbitrary sequence of \( n \) keys, we will observe another interesting property of gaps. Let us consider time \( t_i \) as \( i^{th} \) insertion operation and by \( T^{t=i} \) we show our AVL tree on time \( i \).

**Epoch.** Consider the lifetime of an arbitrary gap \( g \), this gap should be generated in time \( t_{a_1} \) (we call it \( \text{birth}(g) \)) and it may has been consumed (if it has been consumed during \( n \) insertions) in time \( t_{a_2} > t_{a_1} \) (we call it \( \text{death}(g) \)). For any position of a gap in a tree, this position may have seen multiple gaps generated and consumed during a sequence of \( n \) insertions, let us split the whole time \((t = 1 \text{ to } t = n)\) by the times this gap has been consumed and we call each one an **Epoch of the gap**. We define also Epoch of the whole tree with a similar definition, considering the gaps in the first level.

We call **Epoch-\( i \)** the set of insertion operations that occur from the time when the \((i-1)^{th}\) consuming of gap in the first level occurs to the next time a gap in the first level has been consumed. Let us define the size of epoch-\( i \) \((n_i)\) and the height of epoch-\( i \) \((H_i)\) as the size and the height of AVL tree in the end of epoch-\( i \), respectively. Let \( m_i \) denotes the number of insertions during epoch-\( i \), so \( m_i = n_i - n_{i-1} \).

**Lemma 31** Inside each epoch-\( i \), there is **just one** height increase as shown in Figure 6.4. Equivalently that means \( H_i = i \).

**Proof:** By definition of epoch-\( i \), at time of the end of epoch-\( i \), a gap in the first level of tree has been consumed. Similarly at the end of epoch-(\( i-1 \)), another gap in the first level has been consumed, so somewhere inside epoch-\( i \) a new gap in the first level must be generated. By Lemma 30, since this is a first level gap, there is no gap above and this gap must have been generated by a height increase.

On the other hand, to show there can not be more than one height increase inside epoch-\( i \), observe that by contradiction, having two height increases inside epoch-\( i \) means that after the first height increase, the first level gap should been consumed, \( i.e., \) a new epoch has been started inside epoch-\( i ! \)

Let us define a gap \( g \) as an ancestor of a gap \( g' \) iff \( p(g) \) is an ancestor of \( p(g') \). Now we define a relationship between a consumed gap and one of its ancestors, for any gap \( g' \) we say this gap has been **adopted** by one of its ancestor gaps \( g \) if \( g \) is the first ancestor of \( g' \) which has been consumed after “\( g' \) has been consumed” \((\text{death}(g) > \text{death}(g'))\). For a gap \( g \) let \( \text{adopted}(g) \) be the set of nodes adopted by \( g \).
6.1. GAP TERMINOLOGY

Figure 6.4: A sequence of \( n \) insertion into an empty tree with “whole-tree-epochs” and “height increases”.

**Corollary 10** Any gap \( g \) which has been consumed once, will be adopted by exactly one ancestor gap except for the gaps in the first level of the tree.

*Proof:* By definition of adoption. \( \square \)

As we know, unlike insertion, deletion of a node can violate AVL tree condition at every level in the AVL tree. According to the standard insertion algorithm of AVL trees described in [57], after deletion of a node \( v \), three different situation based on the number of children of \( v \) can occur, namely:

1. \( v \) has 0 children: \( v \) will be deleted and nodes’ heights in the path from \( v \) to root may change and they may need to rebalance.

2. \( v \) has 1 child: \( v \) will be deleted, its child should be connected to its parent and nodes’ heights in the path from \( v \) to root may change and they may need to rebalance.

3. \( v \) has 2 children: we should find \( v \)’s successor and replace it with \( v \) and remove the successor, therefore, nodes’ heights in the path from successor to root may change and they may need to rebalance.

For the deletion of a node \( v \) from an AVL tree \( T \), let \( v = v_0, v_1, \ldots, v_t = u \) be the maximum path made of gaps starting from \( v = v_0 \) going upward (this path can be empty). As it has been shown in Figure 6.5, in both cases of “0 child” or “1 child”, after deletion, all gaps of this path will be consumed (disappeared) and the edge \((u, p(u))\) can become a gap or not, depending on the balance factor of \( p(u) \) before deletion (if \( b(p(u)) = 0 \) then \((u, p(u))\) will be a gap after deletion). In case of “1 child”, another gap will be generated as the child of \( v \). In the case of “2 children” since we replace \( v \) with its successor and we remove the successor, we will have similar cases of “0 child” or “1 child” but for the successor. So there is no need to study this case separately.
Figure 6.5: Gaps before and after deletions in case of no children of deleted node(a) or one child(b).

6.2 Rank Terminology

We now introduce and use the rank-balance framework of Haeupler, Sen, and Tarjan [50]. It is equivalent to the original definition of AVL trees [1] and it is easier to work with for proving the conjecture on the number of rotations of AVL trees in a sequence of insertions and deletions. A node in a binary tree is binary, unary, or a leaf if it has two, one, or no children, respectively. A unary node or leaf has one or two null pointers, respectively. A ranked binary tree is a binary tree in which each node \( x \) has a non-negative integer rank \( r_x \). By convention, a null pointer has rank \(-1\). The rank of a ranked binary tree is the rank of its root. Recall that we denote the parent of a node \( x \) by \( p(x) \). The rank difference of a child \( x \) is \( p(x).r - x.r \). A child of rank difference \( i \) is an \( i \)-child; a node whose children have rank differences \( i \) and \( j \) with \( i \leq j \) is an \( i,j \) node.

An AVL tree is a ranked binary tree satisfying the following rank rule: every node is 1,1 or 1,2. Since null pointers have rank \(-1\), every leaf in an AVL tree is 1,1 and has rank 0, and every unary node is 1,2 and has rank 1. Also, every node has at least one child of rank difference 1, we see that the rank of a node in an AVL tree equals its height.

Insertions and deletions in AVL trees can violate the rank rule. We restore the rank rule by changing the ranks of certain nodes and doing rotations, which change the tree structure locally.
while preserving the symmetric order of nodes.

Recall that AVL trees grow by leaf insertions and shrink by deletions of leaves and unary nodes. To add a leaf to an AVL tree, replace a null pointer by the new leaf and give the new leaf a rank of 0. If the parent of the new leaf was itself a leaf, it is now a 0,1 (unary) node, violating the rank rule. In this case, rebalance the tree by repeatedly applying the appropriate case in Figure 6.6 until the rank rule holds.

![Rebalancing cases after insertion](image)

Figure 6.6: Rebalancing cases after insertion. Numbers next to edges are rank differences. Rank differences of unmarked edges do not change. The promote step may repeat. All cases have mirror images.

A promotion (Figure 6.6a) increases the rank of a node \( x \) in Figure 6.6a by 1. We call the node whose rank increases the promoted node. Each promotion either creates a new violation at the parent of the promoted node or restores the rank rule and terminates rebalancing. Each single or double rotation (Figures 6.6b and 6.6c, respectively) restores the rank rule and terminates rebalancing.

To delete a leaf in an AVL tree, remove and replace it by a null pointer; to delete a unary node, replace it by its only child (initially changing no ranks). Such a deletion can violate the rank
rule by producing a 2,2 or 1,3 node. In this case, rebalance the tree by applying the appropriate case in Figure 6.7 until there is no violation. Each application of a case in Figure 6.7 either restores the rank rule or creates a new violation at the parent of the previously violating node. Whereas each rotation case in insertion terminates rebalancing, the rotation cases in deletion can be non-terminating.

Figure 6.7: Rebalancing cases after deletion. Numbers next to edges are rank differences. Rank differences of unmarked edges do not change. Each case except the first single rotation case may repeat. All cases have mirror images.
6.3 Amortized Rotation Cost of AVL Trees

In order to obtain an initial tree in our expensive set $E$, we must build it from an empty tree. Thus the first step in our construction is to show that any $n$-node AVL tree can be built from an empty tree by doing $n$ insertions. Although this result is easy to prove, we have not seen it in the literature.

**Theorem 11** Any $n$-node AVL tree can be built from an empty tree by doing $T$ insertions, each of which does only promotions.

**Proof:** Let $T$ be a non-empty AVL tree. The truncation $\overline{T}$ of $T$ is obtained by deleting all the leaves of $T$ and decreasing the rank of each remaining node by 1. We prove by induction on the rank $k$ of $T$ that we can convert its truncation $\overline{T}$ into $T$ by inserting the leaves deleted from $T$ to form $\overline{T}$, in an order such that each insertion does only promotions. The theorem then follows by induction on the height of the desired tree.

The empty tree can be converted into the one-node AVL tree by doing a single insertion. Thus the result holds for $k = 0$. Suppose $k > 0$ and the result holds for any rank less than $k$. Let $T$ be an AVL tree of rank $k$. Tree $T$ consists of a root $x$ and left and right subtrees $L$ and $R$, both of which are AVL trees. The truncation $\overline{T}$ of $T$ consists of root $x$, now of rank $k - 1$, and left and right subtrees $\overline{L}$ and $\overline{R}$. Both $L$ and $R$ have rank $k - 1$ or $k - 2$, and at least one of them has rank $k - 1$. Suppose $R$ has rank $k - 1$. By the induction hypothesis, $\overline{L}$ can be converted into $L$ and $\overline{R}$ can be converted into $R$ by inserting leaves, each insertion doing only promotions. Out of these insertions into either $\overline{L}$ or $\overline{R}$, exactly one of them will increase the rank of the root by 1.

In the right subtree of $\overline{T}$, do the sequence of insertions that converts $\overline{L}$ into $L$. Then, in the left subtree of the resulting tree, do the sequence of insertions that converts $\overline{R}$ into $R$. If $L$ has rank $k - 1$, then the insertion into $\overline{L}$ that increases the root rank by 1 will, when done in $\overline{T}$, also increase the root rank of $T$ by 1, from $k - 1$ to $k$, increasing the rank difference of the right child of the root from 1 to 2 but having no other effect on the right subtree of the root. Thus, after all the insertions into the left subtree, the tree consists of root $x$, now of rank $k$, left subtree $L$, and right subtree $R$ of rank $k - 2$. The subsequent insertions into the right subtree will convert it into $R$ without affecting the rest of the tree, producing $T$ as the final tree.

If on the other hand $L$ has rank $k - 2$, then the insertions into the left subtree of $\overline{T}$ will convert the left subtree into $L$, in the process increasing the rank of the root of the left subtree from $k - 3$ to $k - 2$ but having no effect on the root or the right subtree. The subsequent insertions will
convert the right subtree into $R$. Among these insertions, the one that increases the rank of the root of the right subtree from $k - 2$ to $k - 1$ will also increase the rank of $x$ from $k - 1$ to $k$, thereby converting the root of the left subtree from a 1-child to a 2-child but having no other effect on the left subtree. Thus the final tree is $T$. The argument is symmetric if $R$ has rank $k - 2$. □

6.3.1 Expensive AVL Trees

Our expensive trees have even rank. We define the set $E$ of expensive trees recursively. Set $E$ is the smallest set containing the one-node tree of rank 0 and such that if $A$, $B$, and $C$ are AVL trees of rank $k$ such that $A$ and $C$ are in $E$, then the two trees of rank $k + 2$ shown in Figure 6.8 are in $E$. The tree of type $L$ in Figure 6.8 contains a root $x$ of rank $k + 2$ and a left child $y$ of the root of rank $k + 1$, and has $A$, $B$, and $C$ as the left and right subtrees of $y$ and the right subtree of $x$, respectively. The tree of type $R$ in Figure 6.8 is similar except that $x$ is the right child of $y$ and $A$, $B$, and $C$ are the left subtree of $y$ and the left and right subtrees of $x$, respectively.

If $T$ is a tree in $E$, its shallow leaf is the leaf $z$ such that all nodes on the path from $z$ to the root, except the root itself, are 2-children. A straightforward proof by induction shows that the shallow leaf exists and is unique.

**Theorem 12** If $T$ is a tree in $E$ of rank $k$, deletion of its shallow leaf takes $k/2$ single rotations and produces a tree of rank $k - 1$. Reinsertion of the deleted leaf takes $k$ promotions and produces a tree of rank $k$ that is in $E$.

**Proof:** We prove the theorem by induction on $k$. In the one-node tree of rank 0, the shallow leaf is the only node. Its deletion takes no rotations and produces the empty tree; its reinsertion takes no promotions and reproduces the original tree. For $k = 2$, there is exactly one tree in $E$ of type $L$ and one of type $R$. As shown in Figure 6.9, rebalancing after deletion of the shallow leaf in the type-$L$ tree takes one rotation and produces a tree of rank 1, and reinsertion takes two promotions.
6.3. AMORTIZED ROTATION COST OF AVL TREES

Figure 6.9: Deletion and insertion of the shallow leaf in a type-L tree of rank 2.

Figure 6.10: Deletion and insertion of the shallow leaf in a type-L tree of rank $k + 2$

and produces the type-R tree. Symmetrically, deletion of the shallow leaf in the type-R tree takes one rotation and produces a tree of rank 1, and reinsertion takes one promotion and produces the type-L tree.

Suppose that the theorem is true for $k$. Let $T$ be a tree of rank $k + 2$ and type $L$ in $E$ (the argument is symmetric for a tree of type R). Let $x$ be the root, $y$ the left child of $x$, and $A$, $B$, and $C$ the left and right subtrees of $y$ and the right subtree of $x$, respectively (see the first tree in Figure 6.10). The shallow leaf of $C$ is the shallow leaf of $T$. By the induction hypothesis, its deletion in $C$ does $k/2$ rotations and converts $C$ into a tree $C'$ of rank $k - 1$. In $T$, deletion of the shallow leaf converts the right subtree of $x$ into $C'$, making the root of $C'$ a 3-child (see the second tree in Figure 6.10). This causes one more single rotation, for a total of $k/2 + 1$, and produces the tree $T'$ (shown as the third tree in Figure 6.10), of rank $k + 1$, with 1,1 root $y$ whose right child $x$ is also 1,1. By the induction hypothesis, reinsertion of the deleted leaf into $C'$ does $k$ promotions and converts $C'$ into a tree $C''$ in $E$ of rank $k$. In $T'$, the same reinsertion converts the right subtree of $T'$ into $C''$, making $x$ 0,1. This causes $x$ and then $y$ to be promoted, for a total of $k + 2$ promotions, and produces the tree $T''$ in Figure 6.10, which is a tree in $E$ of type $R$. □

Corollary 11 The proof of Theorem 12 implies that if one starts with a tree $T$ in $E$ of even rank
$k$ and does $2^{k/2}$ deletion-reinsertion pairs, the final tree will be $T$.

**Corollary 12** For infinitely many $n$, there is a sequence of $3n$ intermixed insertions and deletions on an initially empty AVL tree that takes $\Theta(n \log n)$ rotations.

**Proof**: Let $T$ be any tree in $E$. If $T$ has $n$ nodes, its height is $\Theta(\log n)$ since it is an AVL tree [1]. Apply Theorem 11 to build $T$ in $n$ insertions. Then repeat the following pair of operations $n$ times: delete the shallow leaf; reinsert the deleted leaf. By Theorem 12, the total number of rotations will be $\Theta(n \log n)$. \qed

### 6.4 Summary

In this chapter, we presented two terminology to study AVL trees; Gap terminology and Rank terminology, with the first terminology we proved a new set of theorems and lemmas for finding the subtree size of any node of an AVL tree with respect to the heights and the structure of Gaps, and with the second terminology we proved Heaupler, Sen, and Tarjan's conjecture that alternating insertions and deletions in an $n$-node AVL tree can cause each deletion to do $\Omega(\log n)$ rotations. To do this, we provided a construction which causes each deletion to do $\Omega(\log n)$ rotations: we show that, for infinitely many $n$, there is a set $E$ of expensive $n$-node AVL trees with the property that, given any tree in $E$, deleting a certain leaf and then reinserting it produces a tree in $E$, with the deletion having done $\Theta(\log n)$ rotations. In general, the tree produced by an expensive deletion-insertion pair is not the original tree. Indeed, if the trees in $E$ have even height $k$, $2^{k/2}$ deletion-insertion pairs are required to reproduce the original tree.
Chapter 7

Generation of Trees with Bounded Degree

Recall from Chapter 2, studying combinatorial properties of restricted graphs or graphs with configurations has many applications in various fields such as machine learning and chemoinformatics. Here we study unlabeled ordered trees whose nodes have maximum degree $\Delta$, we present a new encoding and generation algorithm with respective ranking and unranking algorithms for them [6].

A labeled tree is a tree in where to each node is given a unique label. A rooted tree is a tree in which one of the nodes is distinguished from the others as the root. An ordered tree or plane tree is a rooted tree for which an ordering is specified for the children of each node.

We denote unlabeled ordered trees whose nodes have maximum degree $\Delta$ by $T_\Delta$ trees, we also use $T_\Delta^n$ to denote the class of $T_\Delta$ trees with $n$ nodes. Formally, a $T_\Delta$ tree $T$ is defined as a finite set of nodes such that $T$ has a root $r$, and if $T$ has more than one node, $r$ is connected to $j \leq \Delta$ subtrees $T_1, T_2, \ldots, T_j$, each one of them is also recursively a $T_\Delta$ tree, by $T_\Delta^n$ we represent the class of $T_\Delta$ trees with $n$ nodes. An example of a $T_\Delta$ tree is shown in Figure 2.11.

As mentioned in Chapter 2, although many papers have been published earlier in the literature for generating different classes of trees, few of them were related to trees with bounded degree, and to our knowledge, no ‘efficient’ generation, ranking or unranking algorithms are known for ordered trees with bounded degree.

In Section 7.1, we present a new encoding for $T_\Delta^n$ trees. The size of our encoding is $n$ while the alphabet size is always 4. We also present a new generation algorithm with constant average time, and $O(n)$ worst case time in Section 7.2. In this algorithm, the trees are generated in A-order.
Ranking and unranking algorithms are also designed in Section 7.3 with $O(n)$ and $O(n \log n)$ time complexities, respectively. The presented ranking and unranking algorithms need a precomputation of size and time $O(n^2)$ (assuming $\Delta$ is constant).

### 7.1 The Encoding Schema

As mentioned earlier, in most of the tree generation algorithms, a tree is represented by an integer or alphabet sequence called *codeword*, hence all possible sequences of this representation are generated. In general, on any class of trees, we can define a variety of ordering for the set of trees. Classical orderings on trees are *A-order* and *B-order* which are defined as follows [105, 104, 117].

**Definition 7** Let $T$ and $T'$ be two trees in $T^\Delta$ and $k = \max\{\deg(T), \deg(T')\}$, we say that $T$ is less than $T'$ in *A-order* ($T \prec_A T'$), iff

- $|T| < |T'|$, or
- $|T| = |T'|$ and for some $1 \leq i \leq k$, $T_j =_A T'_j$ for all $j = 1, 2, \ldots, i - 1$ and $T_i \prec_A T'_i$;

where $|T|$ is the number of nodes in $T$ and $\deg(T)$ is the degree of the root of $T$.

**Definition 8** Let $T$ and $T'$ be two trees in $T^\Delta$ and $k = \max\{\deg(T), \deg(T')\}$, we say that $T$ is less than $T'$ in *B-order* ($T \prec_B T'$), iff

- $\deg(T) < \deg(T')$, or
- $\deg(T) = \deg(T')$ and for some $1 \leq i \leq k$, $T_j =_B T'_j$ for all $j = 1, 2, \ldots, i - 1$ and $T_i \prec_B T'_i$.

Our generation algorithm, given in the Section 7.2, produces the sequences corresponding to $T_n^\Delta$ trees in *A-order*. For a given tree $T \in T_n^\Delta$, the *generation algorithm* generates all the successor trees of $T$ in $T_n^\Delta$, the position of tree $T$ in $T_n^\Delta$ is called *rank*, the *rank function* determines the rank of $T$; the inverse operation of ranking is *unranking*. These functions can be easily employed in any random generation of $T_n^\Delta$ trees.

The main point in generating trees is to choose a suitable encoding to represent them, and generate their corresponding codewords. Regarding the properties of $T_n^\Delta$, we present our new encoding. For any $T_n^\Delta$ tree $T$, the encoding over 4 letters $\{s, \ell, m, r\}$ is defined as follows. The root of $T$ is labeled by $s$, and for any internal node, if it has only one child, that child is labeled by $s$, otherwise the leftmost child is labeled by $\ell$, and the rightmost child is labeled by $r$, and the children between the leftmost and the rightmost children (if exist) are all labeled by $m$. Nodes are
7.1. THE ENCODING SCHEMA

Figure 7.1: An example of a tree $T \in T_n^\Delta$ (for $\Delta \geq 4$). Its codeword is “s$l$mr$s$r$msr”.

Figure 7.2: a) The first $T_n^\Delta$ tree in A-order. b) The last $T_n^\Delta$ tree in A-order.

labeled in the same way for any internal node in each level recursively, and by a pre-order traversal of $T$, the codeword will be obtained. This labeling is illustrated in Figure 7.1. Note that the 4-letters alphabet codeword corresponding to the first and last $T_n^\Delta$ trees in A-order are respectively “$s\ell m^{\Delta-2}r\ell m^{\Delta-2}r\ldots \ell m^{(n \mod \Delta)-2}r$” and “$s^n$” which are shown in Figure 7.2-a and Figure 7.2-b.

Now, we prove the validity of this encoding for $T_n^\Delta$ trees (one-to-one correspondence).

**Definition 9** Suppose that $\{s, \ell, m, r\}^*$ is the set of all sequences with alphabet of $s, m, \ell, r$ and let $A$ be a proper subset of $\{s, \ell, m, r\}^*$, then we call the set $A$ a CodeSet$^\Delta$ iff $A$ satisfies the following properties:

1. $\epsilon \in A$ ( $\epsilon$ is a string of length 0),

2. $\forall x \in A : sx \in A$,

3. $\forall x_1, x_2, \ldots, x_i \in A$, and $2 \leq i \leq \Delta$: $\ell x_1 m x_2 m x_3 \ldots m x_{i-1} r x_i \in A$.

Now we show that a valid codeword is obtained by the concatenation of the character $s$ and each element of CodeSet$^\Delta$. 
Theorem 13 Let $A$ be the “CodeSet$_\Delta$” and $\delta$ be a string such that $\delta \in A$ and $C$ be a codeword obtained by the concatenation of the character $s$ and $\delta$ (we show it by $s\delta$). There is a one-to-one correspondence between $C$ and a unique $T^\Delta$ tree.

Proof: It can be proved by induction on the length of $C$. Initially for a codeword of length equal to 1, the proof is trivial. Assume that any codeword obtained in the above manner with length less than $n$ encodes a unique $T^\Delta$ tree. For a given codeword with length $n$, because of that concatenation of $s$ and $\delta$, we have:

1. $C = sx$, such that $x \in A$, or

2. $C = s\ell x_1 m x_2 \ldots m x_{j-1} r x_j$, such that $x_i \in A$, $\forall 1 \leq i \leq j \leq \Delta$.

For the first case by induction hypothesis, $x$ is a valid codeword of a $T^\Delta$ tree $T$; therefore, $sx$ is another codeword corresponding to a $T^\Delta$ tree by adding a new root to the top of $T$. This tree is shown in Figure 7.3-a. For the second case, by induction hypothesis and that concatenation of $s$ and $\delta$, each $sx_i$ for $1 \leq i \leq j$ is a valid codeword for a $T^\Delta$ tree, therefore with replacement of ‘$s$ with $\ell$ in $sx_1$’ and ‘$s$ with $m$ in $sx_i$ for $2 \leq i \leq j - 1$’ and finally ‘$s$ with $r$ in $sx_j$’ we can produce $\ell x_1, m x_2, \ldots m x_{j-1}, r x_j$ codewords. Now they all are subtrees of a $T^\Delta$ tree whose codeword is $C = s\ell x_1 m x_2 \ldots m x_{j-1} r x_j$ (add a new root and connect it to each one of them). This tree is shown in Figure 7.3-b.

For a $T^\Delta_n$ tree, this encoding needs only 4 alphabet letters and has length $n$. This encoding is simple and powerful, so it can be used for many other applications besides the generation algorithm. In the next section, we use it to generate $T^\Delta$ trees in A-order.

7.2 The Generation Algorithm

In this section, we present an algorithm that generates the successor sequence of a given codeword of a $T^\Delta_n$ tree in A-order. For generating the successor of a given codeword $C$ corresponding to a
7.2. THE GENERATION ALGORITHM

$T_\Delta^n$ tree $T$, the codeword $C$ is scanned from right to left. Scanning the codeword $C$ from right to left, corresponds to a reverse pre-order traversal of $T$. First we describe how this algorithm works directly on $T$, then we present the algorithm for generating the successor of $C$. For generating the successor of a given $T_\Delta^n$ tree $T$ we traverse the tree in reverse pre-order as follows.

1. Let $v$ be the last node of $T$ in pre-order traversal.

2. If $v$ doesn’t have any brothers, then
   - repeat \{ $v = \text{parent of } v.$ \}
   until $v$ has at least one brother or $v$ be the root of tree $T$.
   - If $v = \text{root}$, then the tree is the last tree in A-order and there is no successor.

3. If $v$ has at least one brother (obviously it has to be a left brother), delete one node from the subtree of $v$ and insert this node into its left brother’s subtree, then rebuild both subtrees (each one as a first tree with corresponding nodes in A-order).

The pseudo-code of this algorithm for codewords corresponding to $T_\Delta^n$ trees is presented in Figure 7.4. In this algorithm, $C$ is a global array of characters holding the codeword (the algorithm generates the successor sequence of this codeword), $n$ shows the size of the codeword (the number of nodes of the tree corresponded to $C$), $STsize$ is a variable contains the size of the subtree rooted by node corresponded to $C[i]$ and $SNum$ holds the number of consecutive visited $s$ characters. This algorithm also calls two functions $updateChildren(i, ChNum)$ presented in Figure 7.5, and $updateBrothers(i, ChNum)$ presented in Figure 7.6. The procedure $updateChildren(i, ChNum)$ regenerates the codeword corresponding to the children of an updated node and the procedure $updateBrothers(i, ChNum)$ also regenerates the codeword corresponding to the brothers of a node with regard to the maximum degree $\Delta$ for each node. In these algorithms, $C$ is a global array of characters holding the codeword, $i$ is the position of the current node in the array $C$, $ChNum$ is the number of children/brothers of $C[i]$ to regenerate the corresponding codeword and $NChild$ is a global array which $NChild[i]$ holds the number of left brothers of node corresponding to $C[i]$ plus one.

**Theorem 14** The algorithm Next presented in Figure 7.4 has a worst case time complexity of $O(n)$ and an average time complexity of $O(1)$.

**Proof:** Worst case time complexity of this algorithm is $O(n)$ because the sequence is scanned just once. For computing the average time, it should be noted that during the scanning process, every
Function $AOrder-Next(n : integer)$;
var $i, Current, STsize, SNum : integer; finished, RDeleted : boolean;
begin
  $Current := n$; $STSize := 0$; $RDeleted := false$; $finished := false$;
  while ($C[Current] = ’s’$ & ($Current ≥ 1$)) do
    $STSize += 1$; $Current -= 1$;
  if ($Current = 0$) then return (’no successor’);
  while (not $finished$) do
    $STSize += 1$;
    switch $C[Current]$ of
      case ’r’:
        $i := Current - 1$; $SNum := 0$;
        while ($C[i] = ’s’$) do
          $SNum := SNum + 1$; $i -= 1$;
        if ($C[i] = ’r’$) then begin
          updateBrothers ($Current$, $STSize$);
          $Current := i$; $STSize := SNum$;
        end;
        if ($C[i] = ’m’$) or ($C[i] = ’ℓ’$) then begin
          if ($STSize = 1$) then $RDeleted := true$;
          if ($STSize > 1$) then begin $STSize -= 1$; updateBrothers($Current + 1$, $STSize$);
            $Current := i$; $STSize := SNum + 1$;
          end;
        end;
      case ’m’:
        if ($RDeleted = true$) then $C[Current] := ’r’$;
        updateChildren ($Current + 1$, $STSize - 1$); $finished := true$;
      case ’ℓ’:
        if ($RDeleted = true$) then $C[Current] := ’ℓ’$;
        updateChildren ($Current + 1$, $STSize - 1$); $finished := true$;
    end;
end;

Figure 7.4: Algorithm for generating the successor codeword for $T_n^Δ$ trees in A-order.

procedure $updateChildren(i, ChNum : integer)$;
begin
  while ($ChNum > 0$) do begin
    if ($ChNum = 1$) then begin
      $C[i] := ’s’$; $NChild[i] := 1$; $i += 1$; $ChNum -= 1$;
    end;
    if ($ChNum > 1$) then begin
      $C[i] := ’ℓ’$; $NChild[i] := 1$; $i += 1$; $ChNum -= 1$;
      while ($NChild[i] < (Δ - 1)$ & ($ChNum > 1$)) do begin
        $C[i] := ’m’$; $NChild[i] := NChild[i] + 1$; $i += 1$; $ChNum -= 1$;
      end;
      $C[i] := ’r’$; $NChild[i] := NChild[i] - 1$; $i += 1$; $ChNum -= 1$;
    end;
  end;
end;

Figure 7.5: Algorithm for updating the children.
Procedure updateBrothers(i, ChNum: integer);
begin
    if ChNum = 1 then begin
        C[i] := 'r'; NChild[i] := NChild[i - 1]; ChNum --;
    end;
    if ChNum > 1 then begin
        C[i] := 'm'; ChNum --; i ++;
        while ((NChild[i] < (∆ - 1)) & (ChNum > 1)) do begin
            C[i] := 'm'; NChild[i] := NChild[i - 1] + 1; i ++; ChNum --;
        end;
        C[i] := 'r'; NChild[i] := NChild[i - 1] + 1;
        i ++; ChNum --; updateChildren(i, ChNum);
    end;
end;

Figure 7.6: Algorithm for updating the neighbors.

time we visit the characters s, m or ℓ, the algorithm will terminate, so we define $S_{i}^{n,∆}$ as the number of codewords of $T_{n}^{∆}$ trees whose the last character s, m or ℓ has distance $i$ from the end, and $S^{n,∆}$ as the total number of $T_{n}^{∆}$ trees. Obviously we have:

$$S^{n,∆} = \sum_{i=1}^{n} S_{i}^{n,∆}. \quad (7.1)$$

We define $H_{n}$ as the average time of generating all codewords of $T_{n}^{∆}$ trees,

$$H_{n} \leq (k/S^{n,∆}) \sum_{i=1}^{n} i S_{i}^{n,∆},$$

$$\leq (k/S^{n,∆}) \sum_{j=1}^{n} \sum_{i=j}^{n} S_{i}^{n,∆}.$$

Where $k$ is a constant value. On the other hand, consider that for $S_{j}^{n+1,∆}$ we have two cases, in the first case, the last character s, m or ℓ is a leaf and in the second one, it is not. Therefore, $S_{j}^{n+1,∆}$ is greater than or equal to just the first case, and in that case by removing the node corresponding to the ‘last character s, m or ℓ of the codeword’, the remaining tree will have a corresponding codeword belongs to exactly one of $S_{k}^{n,∆}$ cases, for $j \leq k \leq n$. By substituting $k$ and $i$ we have:

$$S_{j}^{n+1,∆} \geq \sum_{i=j}^{n} S_{i}^{n,∆}.$$

Therefore, for $H_{n}$ we have:

$$H_{n} \leq (k/S^{n,∆}) \sum_{j=1}^{n} S_{j}^{n+1,∆},$$

then by using Equation (1),

$$H_{n} \leq k S^{n+1,∆}/S^{n,∆}.$$
Finally from [117] we know that the total number of ordered trees is growing same as Catalan number, while $T_n^\Delta$ is a subset of ordered trees can not grow faster than that, this guarantees that for large enough values of $n$, $S^{n+1,\Delta}/S^{n,\Delta} = O(1)$. Therefore, $H_n \leq kO(1) = O(1)$.

\[ \square \]

It should be mentioned that this constant average time complexity is without considering the input or the output time.

### 7.3 Ranking and Unranking Algorithms

By designing a generation algorithm in a specific order, the ranking of algorithm is desired. In this section, ranking and unranking algorithms for these trees in A-order will be given. Ranking and unranking algorithms usually use a precomputed table of the number of a subclass of given trees with some specified properties to achieve efficient time complexities; these precomputations will be done only once and stored in a table for further use. Let $S^{n,\Delta}$ be the number of $T_n^\Delta$ trees, $S^{n,\Delta}_{m,d}$ be the number of $T_n^\Delta$ trees whose first subtree has exactly $m$ nodes and its root has maximum degree $d$, and $D^{n,\Delta}_{m,d}$ be the number of $T_n^\Delta$ trees whose first subtree has at most $m$ nodes and its root has maximum degree $d$.

**Theorem 15**

\[
\begin{align*}
D^{n,\Delta}_{m,d} &= \sum_{i=1}^{m} S^{n,\Delta}_{i,d}, \\
S^{n,\Delta} &= \sum_{i=1}^{n-1} S^{n,\Delta}_{i}. 
\end{align*}
\]

**Proof**: The proof is trivial. \[ \square \]

**Theorem 16**

\[
S^{n,\Delta}_{m,d} = S^{m+1,\Delta}_{m,1} \times \sum_{i=1}^{n-m-1} (S^{n-m,\Delta}_{i,d-1}).
\]

**Proof**: Let $T$ be a $T_n^\Delta$ tree whose first subtree has exactly $m$ nodes and its root has maximum degree $d$; by the definition and as shown in Figure 7.7 the number of the possible cases for the first subtree is $S^{m+1,\Delta}_{m,1}$ and the number of cases for the other parts of the tree is: $\sum_{i=1}^{n-m-1} (S^{n-m,\Delta}_{i,d-1})$. So:

\[
S^{n,\Delta}_{m,d} = S^{m+1,\Delta}_{m,1} \times \sum_{i=1}^{n-m-1} (S^{n-m,\Delta}_{i,d-1}).
\]

\[ \square \]
7.3. RANKING AND UNRANKING ALGORITHMS

Figure 7.7: $T_n^\Delta$ tree whose first subtree has exactly $m$ nodes and its root has maximum degree $d$.

Now, let $T$ be a $T_n^\Delta$ tree whose subtrees are defined by $T_1, T_2, \ldots, T_k$ and for $1 \leq i \leq k \leq \Delta$ : $|T_i| = n_i$ and $\sum_{i=1}^{k} n_i = n - 1$. For computing the rank of $T$, we have to enumerate the number of trees generated before $T$. Let $Rank(T, n)$ be the rank of $T$. The number of $T^\Delta$ trees whose first subtree is smaller than $T_1$ is equal to:

$$\sum_{i=1}^{n_1} S_i^{n_1, \Delta} + (\text{Rank}(T_1, n_1) - 1) \times \sum_{i=1}^{n_1} S_i^{n_1 - 1, \Delta},$$

and the number of $T^\Delta$ trees whose first subtree is equal to $T_1$ but the second subtree is smaller than $T_2$ is equal to:

$$\sum_{i=1}^{n_2} S_i^{n_1 - 1, \Delta} + (\text{Rank}(T_2, n_2) - 1) \times \sum_{i=1}^{n_1} S_i^{n_1 - n_2, \Delta}. $$

Similarly, the number of $T^\Delta$ trees whose first $(j-1)$ subtrees are equal to $T_1, T_2, \ldots, T_{j-1}$ and the $j^{th}$ subtree is smaller than $T_j$ is equal to:

$$\sum_{i=1}^{n_j} S_i^{n_{j-1}, \Delta} + (\text{Rank}(T_j, n_j) - 1) \times \sum_{i=1}^{n_{j-1}} S_i^{n_{j-1} - n_j, \Delta}. $$

Therefore, regarding enumerations explained above, for given tree $T \in T_n^\Delta$ whose subtrees are defined by $T_1, T_2, \ldots, T_k$, we can write:

$$\text{Rank}(T, 1) = 1,$$

$$\text{Rank}(T, n) = 1 + \sum_{j=1}^{k} \left( \sum_{i=1}^{n_j-1} S_i^{n_j-1, \Delta} \right) +$$

$$+ (\text{Rank}(T_j, n_j) - 1) \sum_{i=1}^{n_j} S_i^{n_j - n_j, \Delta}. $$
Function $\text{Rank}(\text{Beg}: \text{integer}; \text{var} \text{ Fin}: \text{integer})$

Var $R, \text{Point, PointFin, } j, \text{Nodes, } n: \text{integer};$

begin
  $n := N[\text{Beg}];$
  if ($n = 1$) then begin
    $\text{Fin} := \text{Beg}; \text{return}(1) \text{ end;}$
  else begin
    $\text{Point} := \text{Beg} + 1;$ $R := 0;$ $\text{Nodes} := 0;$ $j := 1;$
    while ($\text{Nodes} < n$) do begin
      $R := R + D[n - \text{Nodes}, N[\text{Point} - 1, \Delta - j + 1] +$
        $(\text{Rank}(\text{Point}, \text{PointFin}) - 1) \times$
        $D[(n - \text{Nodes} - N[\text{Point}], (n - \text{Nodes} - N[\text{Point}]), \Delta - j];$
      $\text{Nodes} := \text{Nodes} + N[\text{Point}]; j := j + 1;$
      $\text{Point} := \text{PointFin} + 1;$
    end;
    $\text{Fin} := \text{Point} - 1;$
    $\text{return}(R + 1);$  
  end;
end

Figure 7.8: Ranking algorithm for $T_n^\Delta$ trees.

Hence, from Theorem 15, by using $D_{m,d}^n = \sum_{i=1}^{m} S_{i,d}^n$, we have:

$$
\text{Rank}(T, 1) = 1,$$
$$
\text{Rank}(T, n) = 1 + \sum_{j=1}^{k} D_{(n - \sum_{\ell=1}^{j} n_{\ell}, (\Delta - j + 1)}^{(n - \sum_{\ell=1}^{j} n_{\ell}, (\Delta - j + 1)} +$

$$
+ (\text{Rank}(T_{j}, n_{j}) - 1)D_{(n - \sum_{\ell=1}^{j} n_{\ell}, (\Delta - j + 1)}^{(n - \sum_{\ell=1}^{j} n_{\ell}, (\Delta - j + 1)}).
$$

To achieve the most efficient time for ranking and unranking algorithms, we need to precompute $D_{m,d}^n$ and store it for further use. Assuming $\Delta$ is constant, to store $D_{m,d}^n$ values, a 3-dimensional table denoted by $D[n, m, d]$ is enough, this table will have a size of $O(n \times n \times \Delta) = O(n^2)$ and can be computed using Theorems 15 and 16 with time complexity of $O(n \times n \times \Delta) = O(n^2)$.

To compute the rank of a codeword stored in array $C$, we also need an auxiliary array $N[i]$ which keeps the number of nodes in the subtree whose root is labeled by $C[i]$ and corresponds to $n_i$ in the above formula. This array can be computed by a pre-order traversal or a level first search (DFS) algorithm just before we call the ranking algorithm.

The pseudo-code for ranking algorithm is given in Figure 7.8. In this algorithm, $\text{Beg}$ is the variable that shows the positions of the first character in the array $C$ whose rank is being computed ($\text{Beg}$ is initially set to 1), and $\text{Fin}$ is the variable that returns the position of the last character of $C$. 
Now the time complexity of this algorithm is discussed. Obviously computing the array \( N[i] \) takes \( O(n) \). Hence we discuss the complexity of ranking algorithm which was given in Figure 7.8.

**Theorem 17** The ranking algorithm has the time complexity of \( O(n) \).

**Proof:** Let \( T \) be a \( T_n^\Delta \) tree whose subtrees are defined by \( T_1, T_2, \ldots, T_k \) and for \( 1 \leq i \leq k \leq \Delta : \)

\[ |T_i| = n_i \text{ and } \sum_{i=1}^k n_i = n - 1, \]

and let \( T(n) \) be the time complexity of ranking algorithm, then we can write:

\[ T(n) = T(n_1) + T(n_2) + \ldots + T(n_k) + \alpha k, \]

where \( \alpha \) is a constant and \( \alpha k \) is the time complexity of the non-recursive parts of the algorithm. By using induction, we prove that if \( \beta \) is a value greater than \( \alpha \) then \( T(n) \leq \beta n \). We have \( T(1) \leq \beta \).

We assume \( T(m) \leq \beta(m - 1) \) for each \( m < n \), therefore:

\[ T(n) \leq \beta(n_1 - 1) + \beta(n_2 - 1) + \ldots + \beta(n_k - 1) + \alpha k, \]

\[ T(n) \leq \beta(n_1 + \ldots + n_k - k) + \alpha k, \]

\[ T(n) \leq \beta n - \beta k + \alpha k, \]

\[ T(n) \leq \beta n. \]

So the induction is complete and we have \( T(n) \leq \beta n = O(n) \). \( \square \)

Before giving the description of the unranking algorithm we need to define two new operators.

- If \( a \) and \( b \) are integer numbers then \( a \div b^+ \) is defined as follows:
  - If \( b \nmid a \) then \( a \div b^+ \) is equal to \( (a \div b) \).
  - If \( b \mid a \) then \( a \div b^+ \) is equal to \( (a \div b) - 1 \).

- If \( a \) and \( b \) are integer numbers then \( a \mod b^+ \) is defined as follows:
  - If \( b \nmid a \) then \( a \mod b^+ \) is equal to \( (a \mod b) \).
  - If \( b \mid a \) then \( a \mod b^+ \) is equal to \( b \).

For unranking algorithm, we need the values of \( S_n^\Delta \), these values can be stored in an array of size \( n \), denoted by \( S[n] \) (we assume \( \Delta \) is constant). The unranking algorithm is the reverse approach of the ranking algorithm, the unranking algorithm is given in Figure 7.9. In this algorithm, the rank \( R \) is the input, \( Beg \) is a variable showing the position of the first character in the global array \( C \) and initially is set to 1. The generated codeword will be stored in array \( C \). The variable \( n \) is
Function UnRank (R, Beg, n: integer; Root: char);
var Point, i, t, ChildNum: integer;
begin
  if ( (n = 0) or (R = 0) ) then return(Beg - 1)
  else begin
    if (n = 1) then begin
      C[Beg] := Root; return(Beg);
    end
    else begin
      C[Beg] := Root; Point := Beg + 1;
      Root := 'ℓ'; ChildNum := 0;
      while (n > 0) do begin
        ChildNum := ChildNum + 1;
        find the smallest i that D[n, i, ∆ - ChildNum + 1] ≥ R;
        R := R - D[n, i - 1, ∆ - ChildNum + 1];
        if (n - i) = 1 then
          if (ChildNum = 1) then Root := 'ℓ';
          else Root := 'r';
        t := S[n];
        Point := UnRank( (div(R, t)) + 1, Point, i, Root ) + 1;
        R := mod(R, t);
        n := n - i; Root := 'm';
      end;
      return(Point - 1);
    end
  end
end

Figure 7.9: Unranking algorithm for $T_n^\Delta$ trees.

the number of nodes and Root stores the character corresponding to the node we consider for the unranking procedure. For the next character we have two possibilities. If the root is $r$ or $s$ then the next character, if exists, will be $ℓ$ or $s$ (based on the number of root’s children). If the root is $m$ or $ℓ$, we have again two possible cases: if all the nodes of the current tree are not produced then the next character is $m$ otherwise the next character will be $r$.

**Theorem 18** The time complexity of the unranking algorithm is $O(n \log n)$.

**Proof:** Let $T$ be a $T_n^\Delta$ tree whose subtrees are defined by $T_1$, $T_2$, ..., $T_k$ and for $1 \leq i \leq k \leq \Delta$ : $|T_i| = n_i$ and $\sum_{i=1}^{k} n_i = n - 1$, and let $T(n)$ be the time complexity of the unranking algorithm. With regards to the unranking algorithm, the time complexity of finding $j$ such that $D[n, j, ∆ - ChildNum + 1] ≥ R$ for each $T_i$ of $T$ is $O(\log n_i)$, therefore, we have:

$$T(n) = O(\log n_1 + \log n_2 + \ldots + \log n_k) + T(n_1) + T(n_2) + \ldots + T(n_k).$$

We want to prove that $T(n) = O(n \log n)$. In order to obtain an upper bound for $T(n)$ we do
as follows. First we prove this assumption for \( k = 2 \) then we generalize it. For \( k = 2 \) we have 
\[
T(n) = O(\log(n_1) + \log(n_2)) + T(n_1) + T(n_2).
\]
Let \( n_1 = x \) then we can write the above formula as 
\[
T(n) = T(x) + T(n - x) + O(\log(x) + \log(n - x)) = T(x) + T(n - x) + C' \log(n).
\]

For proving that \( T(n) = O(n \log(n)) \) we use an induction on \( n \). We assume 
\[
T(m) \leq C_m \log(m)
\]
for all \( m \leq n \), thus in \( T(n) \) we can substitute 
\[
T(n) \leq C \times x \log(x) + C \times (n - x) \log(n - x) + C' \log(n).
\]

Let \( f(x) = C \times x \log(x) + C \times (n - x) \log(n - x) \), now the maximum value of \( f(x) \) with respect to \( x \) and by considering \( n \) as a constant value can be obtained by evaluating the derivation of \( f(x) \) which is 
\[
f'(x) = C \times \log(x) - C \times \log(n - x).
\]
Thus if \( f'(x) = 0 \) we get \( x = (n - 1)/2 \) and by computing \( f(1) \), \( f(n - 2) \) and \( f((n - 1)/2) \) we have: 
\[
\begin{align*}
 f(1) &= f(n - 2) = C \times (n - 2) \log(n - 2), \\
 f((n - 1)/2) &= 2C \times ((n - 1)/2) \times \log((n - 1)/2) < C \times (n - 2) \log(n - 2).
\end{align*}
\]
so the maximum value of \( f(x) \) is equal to \( C \times (n - 2) \log(n - 2) \) and therefore 
\[
T(n) \leq C \times (n - 2) \log(n - 2) + C' \times \log(n).
\]

It is enough to assume \( C = C' \), then 
\[
T(n) \leq C \times (n - 2) \log(n) + C \times \log(n) \leq C \times n \log(n).
\]

Now, for generalizing the above proof and proving \( T(n) = O(n \log n) \), we should find the maximum of the function \( f(n_1, n_2, \ldots, n_k) = \prod_{i=1}^{k} n_i \). By the Lagrange method we prove that the maximum value of \( f(n_1, n_2, \ldots, n_k) \) is equal to \( (\frac{n}{k})^k \). Then 
\[
\frac{df}{dk} = (\frac{n}{k})^{k-1} (\log(\frac{n}{k})) = 0,
\]
\[
\log(\frac{n}{k}) = 1,
\]
\[
\frac{n}{k} = e \Rightarrow k = \frac{n}{e},
\]
so the maximum value of \( f(n_1, n_2, \ldots, n_k) \) is equal to \( e^\frac{n}{e} \). We know that: 
\[
T(n) = O(\log n_1 + \log n_2 + \ldots + \log n_k) + T(n_1) + T(n_2) + \ldots + T(n_k),
\]
so
CHAPTER 7. GENERATION OF TREES WITH BOUNDED DEGREE

\[ T(n) = O(\log(\prod_{i=1}^{k} n_i) + \sum_{i=1}^{k} T(n_i)), \]
\[ T(n) < O(\log(n^{\frac{k}{2}})) + \sum_{i=1}^{k} T(n_i), \]
\[ T(n) < O(\frac{n}{\epsilon} \log e) = O(n) + \sum_{i=1}^{k} T(n_i). \]

Finally, by using induction, we assume that for any \( m < n \) we have \( T(m) < \beta m \log m \), therefore:

\[ T(n) = O(n) + \sum_{i=1}^{k} T(n_i), \]
\[ T(n) < O(n) + \sum_{i=1}^{k} \beta O(n_i \log n_i), \]
\[ T(n) < O(n) + \beta \log(\prod_{i=1}^{k} (n_i^{n_i})), \]
\[ T(n) < O(n) + O(\log(n^n)), \]
\[ T(n) = O(n \log n). \]

Hence, the proof is complete. \( \square \)

7.4 Summary

In this chapter, we studied the problem of generation, ranking and unranking of ordered trees of size \( n \) and maximum degree \( \Delta \); we presented an efficient algorithm for the generation of these trees in A-order with an encoding over 4 letters and size \( n \). Also two efficient ranking and unranking algorithms were designed for this encoding. The generation algorithm has \( O(n) \) time complexity in worst case and \( O(1) \) in average case. The ranking and unranking algorithms have \( O(n) \) and \( O(n \log n) \) time complexity, respectively. The presented ranking and unranking algorithms use a precomputed table of size \( O(n^2) \) (assuming \( \Delta \) is constant). For the future works, generating this class of trees in “B-order” and “minimal change ordering” and finding some explicit relations for counting them, are major unresolved problems.
Chapter 8

Conclusions and Discussion

Trees are one of the most important basic and simple data structures for organizing information in computer science. A great amount of research has been done on developing new data structures for organizing data. The memory hierarchies of modern computers are composed of several levels of memories, starting from the caches. Caches have very small access time and capacity comparing to main memory and external memory. From cache to main memory, then to external memory, access time and capacity increases significantly. Two main memory models to evaluate the I/O complexity are external memory model [2] and cache-oblivious model [41, 82].

In external memory model, accessing an item from external storage is extremely slow. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are considered of taking no time at all and this is because the external memory is so much slower than the random access memory [71]. We assume that each external memory access (called an I/O operation or just I/O) transmits one page of \( B \) elements.

Cache-oblivious model [41, 82] allows to consider only a two-level hierarchy, while proving results for a hierarchy composed of an unknown number of levels. In this model, memory has blocks of size \( B \) words, which \( B \) is an unknown parameter and a cache-oblivious algorithm is completely unaware of the value of \( B \) used by the underlying system.

We introduced the core partitioning scheme, which maintains a balanced search tree as a dynamic collection of complete balanced binary trees called cores. Using this technique we achieve the same theoretical efficiency of modern cache-oblivious data structures by using the classic structures such as weight-balanced trees or height balanced trees such as AVL trees or original binary search trees. We show that these “classic data structures” can be dynamized in as efficient as very
modern ones in cache-oblivious memory model. We preserve the original topology and algorithms of the given balanced search tree using a simple post-processing with guaranteed performance to completely rebuild the changed cores (possibly all of them) after each update. Using our core partitioning scheme, we show how to store balanced trees such as weight-balanced trees and height-balanced trees (AVLs), so that they simultaneously achieve good memory allocation, space-efficient representation, and cache obliviousness. When performing updates, we show that weight-balanced trees can be maintained with a logarithmic cost, while AVL trees require super poly-logarithmic cost by a lower bound on the subtree size of the rotated nodes.

The notion of core partition shows how to obtain cache-efficient versions of classical balanced binary search trees such as AVL trees and weight-balanced trees. A natural question is whether the core partition can be also applied to arbitrary binary search trees which can be unbalanced. We give a positive answer to this question: the resulting data structure, called Cache-Oblivious General Balanced Tree (COG-tree), can be seen as a smooth extension of Anderson’s General Balanced Tree to the cache-oblivious model with transfer block size $B$. The COG-trees and the core partitioning on weight-balanced trees of $n$ nodes have an improved cache complexity of $O(\log_B n)$ amortized block transfers and $O(\log n)$ amortized time for updates, and $O(\log_B n)$ block transfers and $O(\log n)$ time for search. The $O(\log_B n)$ amortized block transfers for update is theoretically efficient. The space occupancy is $O(n)$ extra bits besides the space needed to store the keys alone.

By applying the core partitioning scheme on AVL trees, and showing that it can not be maintained in amortized logarithmic time, another question would be whether the core partitioning scheme can be applied to AVL trees with constant or logarithmic average time? We show a simple version of core partitioning to maintain the nodes of an AVL tree in external memory model. For $n$ keys and block size $B$, the searching cost is $O(\log_B n)$ block transfers in the worst case, while the updating cost is equal to the searching cost plus $O(1)$ expected block transfers (for restructuring the tree) assuming that AVL trees of the same height are uniformly distributed. The analysis is based on the fact that the expected cost is constant under the assumption that rebalancing a node has a cost proportional to its subtree size.

We also improve the rebalancing cost of AVL trees in Mehlhorn and Tsakalidis [70]. They showed that the total rebalancing cost for a sequence of $n$ arbitrary insertions is at most $2.618n$. We prove this cost is constant in average.

We also introduced the gaps in AVL trees. Gaps are special tree edges such that the height difference between the subtrees, rooted at their two endpoints, is equal to 2. We showed how to
express the size of a given AVL tree in terms of the heights of the gaps. Using that, the size of any AVL tree can be characterized with a very simple and useful formula and we can describe the precise relationship between ‘the size and the heights of the nodes’ and ‘the subtree sizes and the heights of the gaps’, we can also independently describe the relationship between the heights of the nodes and the heights of the gaps. We have also studied gaps’ behavior in an AVL tree during a sequence of insertions and deletions.

As known, an insertion in an \( n \)-node AVL tree takes at most two rotations, but a deletion in an \( n \)-node AVL tree can take \( \Theta(\log n) \). A natural question is whether deletions can take many rotations not only in the worst case but in the amortized case as well? Heaupler, Sen, and Tarjan’s [50] conjectured that alternating insertions and deletions in an \( n \)-node AVL tree can cause each deletion to do \( \Omega(\log n) \) rotations. We proved that conjecture is true by providing a construction which causes each deletion to do \( \Omega(\log n) \) rotations: we showed that, for infinitely many \( n \), there is a set \( E \) of \textit{expensive} \( n \)-node AVL trees with the property that, given any tree in \( E \), deleting a certain leaf and then reinserting it produces a tree in \( E \), with the deletion having done \( \Theta(\log n) \) rotations. One can do an arbitrary number of such expensive deletion-insertion pairs. The difficulty in obtaining such a construction is that in general the tree produced by an expensive deletion-insertion pair is not the original tree. Indeed, if the trees in \( E \) have even height \( k \), \( 2^{k/2} \) deletion-insertion pairs are required to reproduce the original tree.

Finally as a byproduct of our research, we introduced a new encoding over an alphabet of size 4 for representing unlabeled ordered trees with maximum degree \( \Delta \). We use this encoding for generating these trees in \( \Lambda \)-order with \( O(1) \) average time and \( O(n) \) worst case time complexity. Due to the given encoding, both ranking and unranking algorithms are also designed taking \( O(n) \) and \( O(n \log n) \) time complexities (with a precomputation of size and time \( O(n^2) \)).

For the future works, the main problem would be applying core partitioning scheme on the remaining binary search trees such as red-black trees or 2-3 trees, investigating that if we can obtain efficient results in cache-oblivious/external memory model.
CHAPTER 8. CONCLUSIONS AND DISCUSSION
Bibliography


