Contents

3.6 Balancing

An immutable binary tree can be balanced to minimise the time taken to access the data within it. A number of schemes for balancing binary trees are described in the literature. This section describes the implementation of a scheme for balancing the Canonical Binary Tree. The balancing process is independent of the ADT implemented by the Canonical Binary Tree.

The problem is to adapt a balancing scheme, implemented in an imperative programming language, so that it can be used to balance the immutable Canonical Binary Tree. Balancing reduces the average time required to access the data associated with a leaf. Balancing improves the performance of the priority queue, deque, map and vector implemented by the ADT by reducing the average access time of all of the Canonical Binary Tree operations.

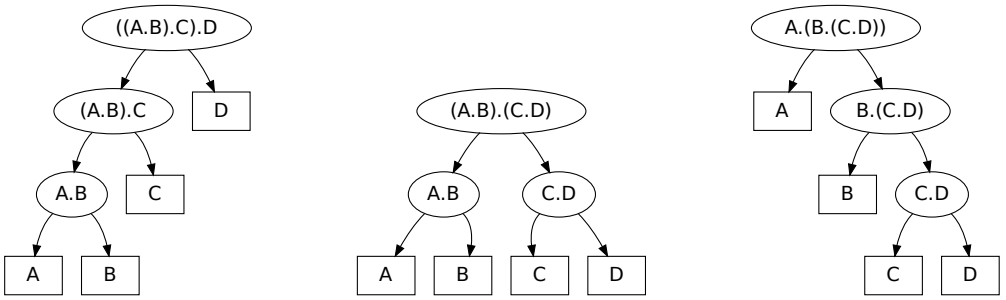
The main contribution of this section is the development of an Immutable Data Structure that separates the performance characteristics of the structure from the ADT to which it conforms. This section focuses on techniques for minimising the average access time and the amortised access time of Immutable Data Structures implemented in imperative programming languages.

3.6.1 Balancing schemes

The Canonical Binary Tree described in the previous section can be balanced independent of the ADT that it implements.

Figure 3.1 illustrates how the associativity property of the annotator function permit the topological transformations required to balance the tree regardless of the ADT that it implements.

A balancing scheme works by implementing a set of balancing invariants. When the invariants are compromised the tree is restructured so that its topology conforms to the invariants. Restructuring takes the form of topological transformations called rotations. A balancing scheme can be characterised by a set of invariants and a set of rotations. These are combined into a set of cases. Each case is characterised by a configuration of nodes that violates the invariants. These nodes are made to conform to the invariants by applying the rotations. Typically, balancing algorithms restructure the tree during a mutation, guaranteeing balancing invariants as post conditions to each mutating function. The post condition is enforced by applying the checks to each node altered by the



(a) (b) (c)

Figure 3.1: **The associativity property permits balancing.** The mean number of nodes that are traversed to reach a leaf of the balanced tree (b) is two, whereas for the unbalanced trees (a) and (c) it is 2.25. The associativity property of the annotator function '.' allows the topology of the tree to be modified, in order to reduce the average access time, without affecting the functionality of the ADT being implemented.

mutation.

Many tree balancing schemes are based on B-Trees [BM72]. In these schemes the invariants and balancing information are expressed in terms of binary B-Tree composed from the nodes of the binary tree. The B-Tree nodes are referred to as pseudo-nodes. Each B-Tree pseudo-node contains a number of binary tree nodes. The maximum number of children of a pseudo-node is referred to as the order of the B-tree. Nodes in the same pseudo-node are said to be joined by horizontal edges. Edges between pseudo-nodes are said to be vertical. Balancing information in the node indicates whether each edge is horizontal or vertical. Pseudo-nodes can be joined or split by changing the orientation of the edges. A balancing algorithm joins and splits the B-tree nodes so that the B-tree remains perfectly balanced. This places a limit on the imbalance of the underlying binary tree.

A red-black tree is a self balancing binary B-tree of order four. For every red-black tree there is at least one corresponding 2-4 B-Tree with elements in the same order. The invariants are described in terms of the colours red and black. The implementation of an ephemeral red-black tree is described in detail by Tarjan [GS78]. As a post condition to insertion each node on the path is examined to ensure that it complies with the invariants. Compromised invariants

are restored by a series of rotations. For each node on the path of an insertion there are five possible cases. There are six possible cases for each node during a deletion.

Okasaki describes an immutable implementation of a red-black tree [Oka98] in the functional programming languages Haskell and ML. The implementation relies on repeated pattern matching. Each node on the path is compared with one of the cases and compromised invariants are fixed by applying rotations. Each case corresponds to a single program line in both functional programming languages making the implementation both brief and easy to follow.

The C++ STL contains an implementation of a red-black tree in an imperative programming language [Jos99]. The imperative implementation relies on explicit testing of each of the cases for each node on the path. Rotations are implemented by pointer manipulation. As a result the imperative implementation appears much more involved than the functional implementation. The high number of cases and the complexity of the rotations makes the implementation of red-black balancing in an imperative programming language both long winded and opaque.

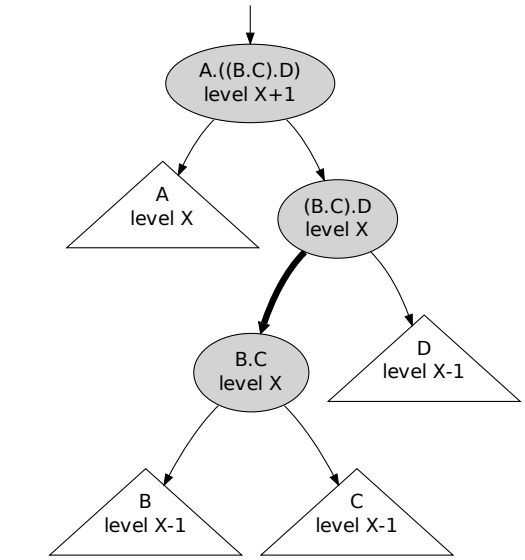
To apply a balancing scheme to the Canonical Binary Tree the rotations must be implemented using path copy. This promises to make the implementation even more unwieldy, so to ease the implementation effort a simpler balancing scheme than red-black is required.

3.6.2 Balancing the Canonical Binary Tree

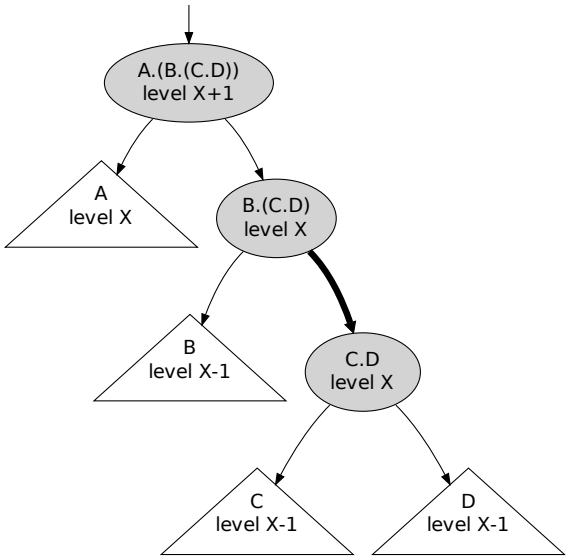
This section describes how the Canonical Binary Tree can be balanced using the AA-tree balancing scheme [And93]. The AA-tree has fewer cases and simpler rotations making it far simpler to implement than a red-black tree. The rotations are easily expressed in terms of path copy operations.

An AA-tree is a self balancing binary B-tree of order three. For every AA-tree there is at least one corresponding 2-3 B-Tree with elements in the same order. The balancing of ephemeral AA-trees is described in detail by Anderson [And93]. A 2-3 B-Tree consists of pseudo-nodes containing either one or two binary tree nodes. Two binary tree nodes joined by a horizontal edge are regarded as forming a pseudo-node. Restructuring operations implement rotations that maintain a perfectly balanced 2-3 B-tree. This places a limit on the imbalance of the underlying binary tree.

Each tree node maintains balancing information in the form of a level number.

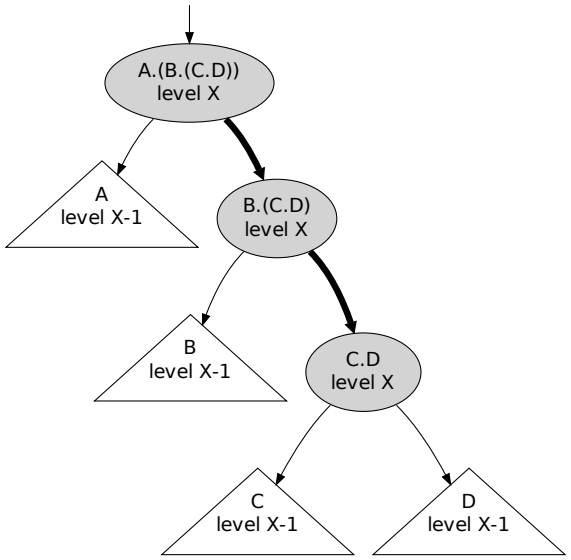


(a)

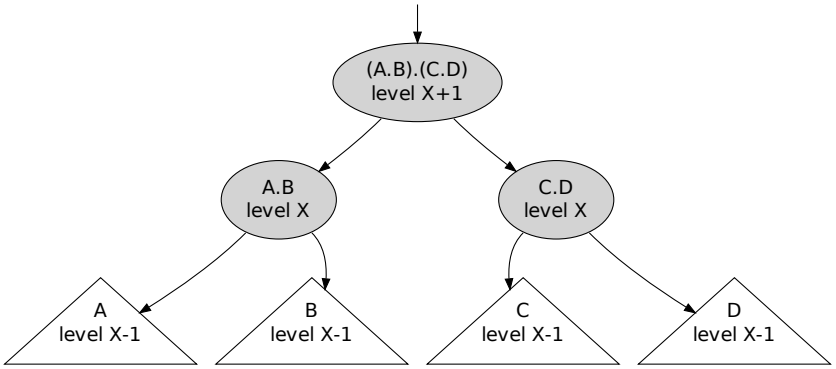


(b)

Figure 3.2: **A skew balancing rotation** corrects violations of the invariant that only right edges are horizontal. Horizontal edges linking the nodes within a pseudo-node are shown in bold. The level number adjustment is also indicated. (a) A subtree that violates the invariant because it has a horizontal left edge. (b) A right rotation transforms the horizontal left edge into a horizontal right edge.



(a)



(b)

Figure 3.3: **Split rotation** corrects violations of the invariant that the largest pseudo-node has three children. Horizontal edges linking the nodes within a pseudo-node are shown in bold. The level number adjustment is also indicated.
(a) A subtree that violates the invariant because it forms a pseudo-node with four children.
(b) A left rotation is performed to transform a pseudo-node with four children into three pseudo-nodes with two children each. The split balancing rotation reduces the size of a pseudo-node by elevating the middle node.

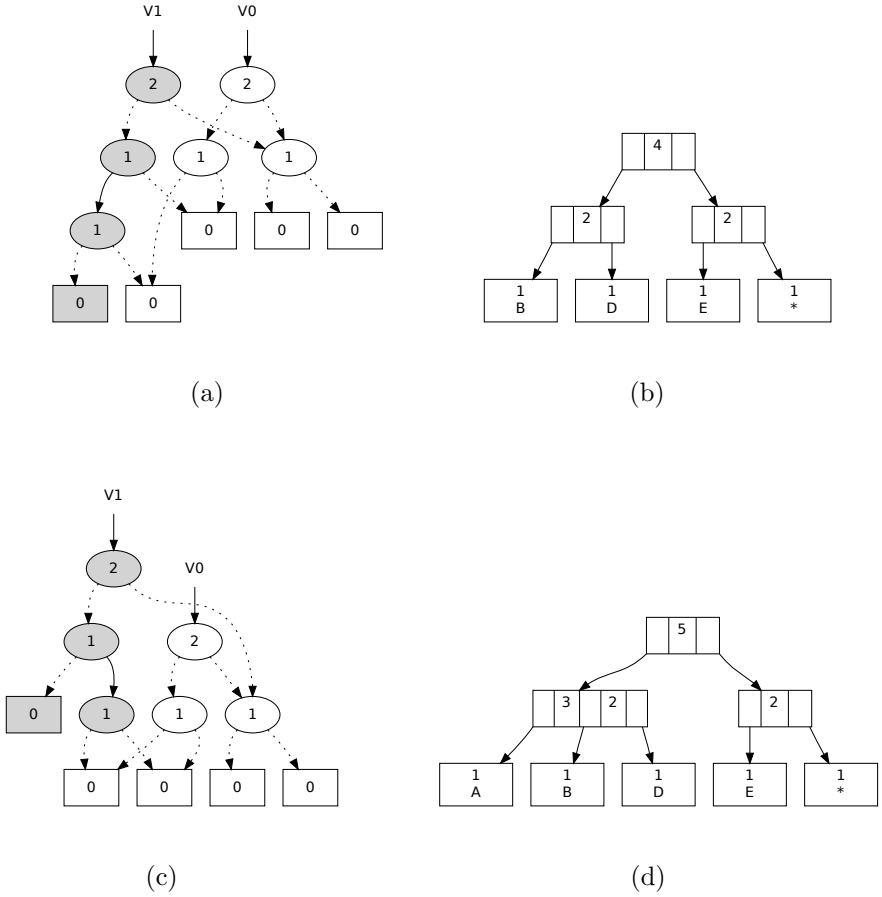


Figure 3.4: **Example of a skew balancing rotation acting on a vector.** (a) A Canonical Binary Tree implementing a vector. The level number of each node is shown. Vertical edges are dotted. Horizontal edges are bold. The insertion of an element, with a value of A, creates a new version which is shaded. Insertion without balance creates a horizontal left edge that violates the invariant that only right edges may be horizontal so a skew balancing rotation is performed.
(b) Version V0 of a vector viewed as a 2-3 B-tree.
(c) The Canonical Binary Tree implementing this vector after inserting the value A and performing a skew balancing rotation. The skew balancing rotation transforms a potential horizontal left edge into a horizontal right edge.
(d) Version V1 of the vector viewed as a 2-3 B-tree.

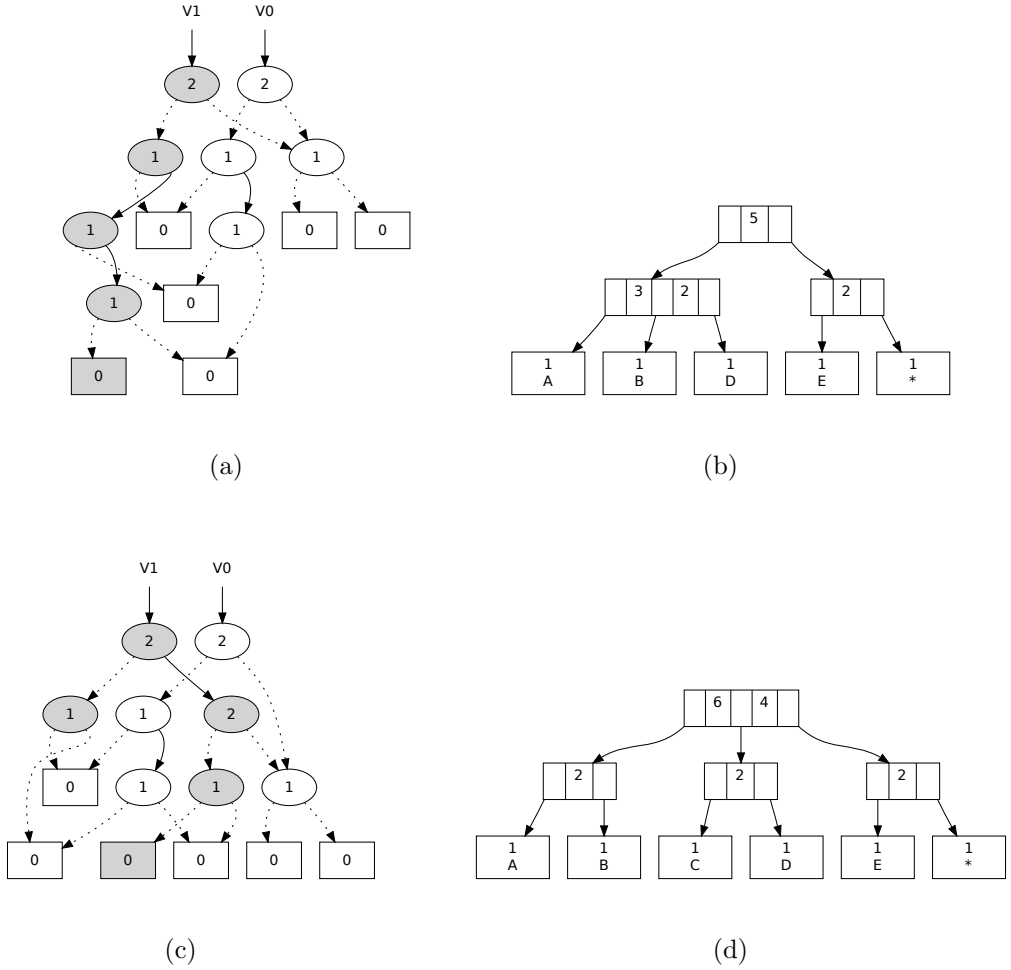


Figure 3.5: **Example of a split balancing rotation acting on a vector.**
(a) A Canonical Binary Tree implementing a vector. The level number of each node is shown. Vertical edges are dotted. Horizontal edges are bold. The insertion of an element, with a value of C, creates a new version which is shaded. Insertion without balance creates two adjacent horizontal right edges representing an overly full pseudo-node that violates the invariant that a pseudo node has at most three children.
(b) Version V0 of a vector viewed as a 2-3 B-tree.
(c) The Canonical Binary Tree implementing this vector after inserting the value C and performing a skew followed by a split balancing rotation. The split balancing rotation transforms the potential overly full pseudo-node by raising the level of the middle node. This creates a horizontal left edge which must be transformed into a horizontal right edge by a skew rotation.
(d) Version V1 of the vector viewed as a 2-3 B-tree.

This number corresponds to the level of the pseudo-node within the B-Tree. The root pseudo-node has the highest level number. Pseudo-nodes at the bottom of the tree are on level 1 and leaves are on level 0. A binary tree edge which connects nodes with equal level numbers is regarded as a horizontal edge and the nodes joined by a horizontal edge form a pseudo-node. A binary tree edge connecting nodes with different level numbers is regarded as a vertical edge and these edges connect pseudo-nodes to form the B-tree.

The AA-Tree maintains the following invariants:

- Only right edges are horizontal
- Pseudo-nodes have at most three children
- There are no breaks in the level numbers

The invariants can be implemented by examining three cases for insert operations and five cases for delete operations. Restructuring operations consisting of combinations of just two balancing rotations, called skew and split, are required to maintain the invariants. The skew and split balancing rotations change the topology of the Canonical Binary Tree. The associativity of the annotator function ensures that the annotations preserve the functionality of the ADT that it implements.

A 2-3 B-tree preserves the invariant that all right edges are horizontal. A horizontal left edge is transformed into a horizontal right edge by a right rotation called a skew.

Figure 3.2 illustrates a skew balancing rotation.

A 2-3 B-tree preserves the invariant that a pseudo-node has at most three children. A pseudo-node with four children contains two horizontal right edges. These are transformed into a left and a right vertical edge by a left rotation called a split. The split balancing rotation raises the level of the middle vertex.

Figure 3.3 illustrates a split balancing rotation.

Figure 3.4 illustrates an example of a skew balancing rotation acting on a vector.

Figure 3.5 illustrates an example of a split balancing rotation acting on a vector.

A tree can be balanced at any time. In our implementation balancing takes place during insert and delete operations. These operations create a new path

within the Canonical Binary Tree and this path can cause the balancing invariants to be compromised. Each node on the path is checked by comparing the configuration of neighbouring nodes to ensure the invariants are preserved. The process of checking the balancing invariants starts at the leaf and ends at root. The addresses of the vertices on the path created by the operation are maintained on a stack so that the path can be accessed in reverse order. An operation that would cause an invariant to be compromised is made compliant by restructuring the tree during the path copy.

3.6.3 Previous work

Tree balancing schemes reduce access time by ensuring that paths within a tree are of similar lengths so that the tree remains balanced. Guibas describes a commonly used balancing scheme for ephemeral trees called the red-black tree [GS78]. Balancing schemes are applicable to both ephemeral and Immutable Data Structures. Okasaki applies the red-black scheme to balance immutable binary trees implemented by a purely functional data structure [Oka98]. Okasaki also describes a purely functional implementation of the AVL tree balancing algorithm. However, the balancing of an immutable binary tree implemented in an imperative programming language has not previously been considered.

3.6.4 Utility functions

The Canonical Binary Tree provides an ADT-agnostic in-order traversal of the data structure that returns the values of the leaves. The first value returned is that of the left most leaf of the tree and the last value returned is that of the sentinel.

The Canonical Binary Tree also provides a full copy utility that creates a copy of the tree which shares leaves with the tree being copied and a naïve copy utility that creates an entirely new copy of the tree. The full copy utility can be used to both copy and compress a tree. The naïve copy can be used to both copy a tree and to convert between different ADTs.

3.6.5 Optimisation

To reduce the number of paths created by the balance process a copy of a path can be balanced in mutable storage and then copied to immutable storage. This

optimisation reduces the amount of storage consumed by the Immutable Data Structure at the cost of additional copying.

Anderson describes a 2-3 B-tree in which the balancing information takes the form of a level number and the invariants act on this number. This formulation is known as an AA-tree. However, the level number is just an implementation convenience. The invariants of the tree can also be expressed in terms of just horizontal and vertical edges. Such a formulation results in a smaller node size as a single bit indicator can be used to indicate the orientation of an edge.

A node may contain information about the subtree it suspends because that subtree is immutable. This information can reduce the need to access the subtree during balancing operations. It is possible to record within the node itself whether or not a child of a node is a leaf. It is also possible to record whether or not the link to a child's left child is horizontal or vertical within the parent node. These optimisations reduce the number of nodes that must be accessed when testing the balancing invariants. Using these optimisations it is only necessary to access nodes on the path to balance the tree.

The size of a node can have a significant effect on performance as larger node sizes reduce the effectiveness of caching. In our implementation the data type of the annotation is supplied as a template parameter. The smallest data type that can accommodate the expected range of annotation values is chosen to minimise the node size. The nature of the references to a child node also affects the size of the node. Memory displacements or ordinal numbers may be used instead of pointers to reduce the node size when the memory to be used by the data structure is pre-allocated in a contiguous chunk.

A tree with multiple children per node improves access times by making the tree shallower but this optimisation comes at the cost of increased implementation complexity. A binary tree can be mapped onto a tree with larger nodes by making some of the parent-child relationships implicit. When paths are written to contiguous memory the relationship between a node and one of its children can be implied, because this child resides in a consecutive memory location, so it is only necessary that a node contain a reference to one of its children and a bit indicating whether that child is to the left or right. An immutable path is typically written into contiguous memory leaf first, so the implicit child of a node immediately precedes it in storage. By restricting the implicit parent-child relationship to nodes on a path it is possible to achieve the effect of a large

node with multiple children without significantly increasing the implementation complexity.

When implementing Immutable Data Structures using C or C++ it is tempting to allocate individual nodes by using the functions *malloc* or *new*, but we found that this leads to poor performance. The effect is particularly pronounced in a concurrent execution environment where memory allocation is typically serialised. We found that allocating nodes on a cache line boundary and fitting an integral number of nodes into a cache line improved performance. In our implementation we used the Threading Building Blocks scalable memory allocator to pre-allocate cache aligned contiguous chunks of storage for our data structures [Int09]. A discussion of memory allocation in Immutable Data Structures can be found on our website [Jar11].

3.6.6 Amortised access time

Amortised analysis is a method of analysing the performance of a function acting on a data structure. The idea is that costly operations occur rarely and that their affect on performance is offset by the occurrence of cheaper, more frequent operations. The analysis proceeds by establishing the worst case time bound of the function and how frequently this worst case occurs. The amortised time per operation is the worst case time bound on a series of m operations divided by m. Goodrich gives a detailed explanation of amortised time analysis for many common ephemeral data structures [GT09].

Amortised analysis has two distinct purposes. Firstly it guides data structure design and allows comparison between equivalent operations on different data structures. For example, the amortised time for the insertion of a value with associated priority into a priority queue ADT implemented by two different data structures can be compared. Secondly, it guides application development because it indicates the relative cost of different operations.

Published amortised access times for functions acting on a data structure are an important guide for developing high performance applications. However, access times are a characteristic of the data structure implementation rather than the ADT. If the application programmer needs to know the amortised access time for a series of operations then the details of the implementation of that data structure have not been completely hidden by abstraction. Using the results of amortised analysis as a guide to developing applications is orthogonal to the

principle of encapsulating the implementation details of a data structure.

The amortised time for all functions of the Canonical Binary Tree are identical as each function requires that a single leaf be accessed. The worst case time bound is $2log_2(n)$ as the longest path to a leaf in a balanced AA-tree is twice as long as the shortest [And93].

Bibliography

[And93] Arne Andersson. Balanced search trees made simple. In *WADS '93: Proceedings of the Third Workshop on Algorithms and Data Structures*, pages 60–71, London, UK, 1993. Springer-Verlag.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21. IEEE, 1978.

[GT09] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2009.

[Int09] Intel. Intel Threading Building Blocks: Programming for Current and Future Multicore Platforms. *IEEE/ACM International Symposium on Code Generation and Optimization*, July 2009.

[Jar11] Kim Jarvis. Transactional Data Structures. http://transactionalmemory.com, June 2011.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A tutorial and reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Oka98] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, New York, NY, USA, 1998.