Comparison Based Analysis of Advanced Tree Structures.

Monika
monikapanghal@gmail.com

Abstract— In this paper, the comparison is done among the most commonly used tree data structures i.e. B-trees and binary trees. This comparison is made among all the variants of these trees on the basis of various operations done over them like creation, insertion, deletion, and searching. The steps needed for these operations are counted and compared. This study can help us to find out easily that which tree data structure can be used in an application to make it efficient.

Keywords— Binary Search Tree, Self Balancing Tree**,** B-Tree, SplayTree, Scapegoat Tree, AA Tree, AVL Tree, B+ tree, Binary Heap.

1. INTRODUCTION

Now a day computer plays a very important role in human life. In almost every application computer needs to store and process the information. For this purpose certain criterion is needed using which the information can be stored and processed efficiently. That is why the term data structure came into existence. Data structure is in fact a generalized term that represents various methods that can be used to store and retrieve information. In this paper the focus is on the study of one such data structure called tree structure and various advances in this structure. Here also a comparison among the various tree structures is made to find out the best one suited for different applications.

1.1 Data structure

What actually computers do? A very simple answer to this question is that computer stores data, process it and reproduces it as information as and when required. Representation of data should be in a proper format so that accurate information can be produced at high speed.

In computer science, a data structure is a way of storing data in a computer so that it can be used efficiently. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structures are implemented in a programming language as data types and the references and operations they provide [1][2]). Based on their characteristics data structure can be classified as follows.

1. Base data structure
2. Linear data structure
3. Non-linear or hierarchical data structure

1.1.1 Base data structure

The data structure that can be used as a base to other large data structures is called base data structure. It can also be called data type in term of programming languages.

1.1.2 Linear data structure

Any data structure which organizes the data elements one after the other is a linear data structure. The elements of a linear data structure form a sequence or a linear list [3].

1.1.3 Non- linear or hierarchical data structure

A data structure is said to be non-linear or hierarchical if its elements doesn’t form a sequence or a linear list, instead the structure looks hierarchical [4]. We are concerned with this type of data structure as **Tree Structure** belongs to this type.
From all the above discussion we can draw the hierarchical structure of the data structure family as shown below.

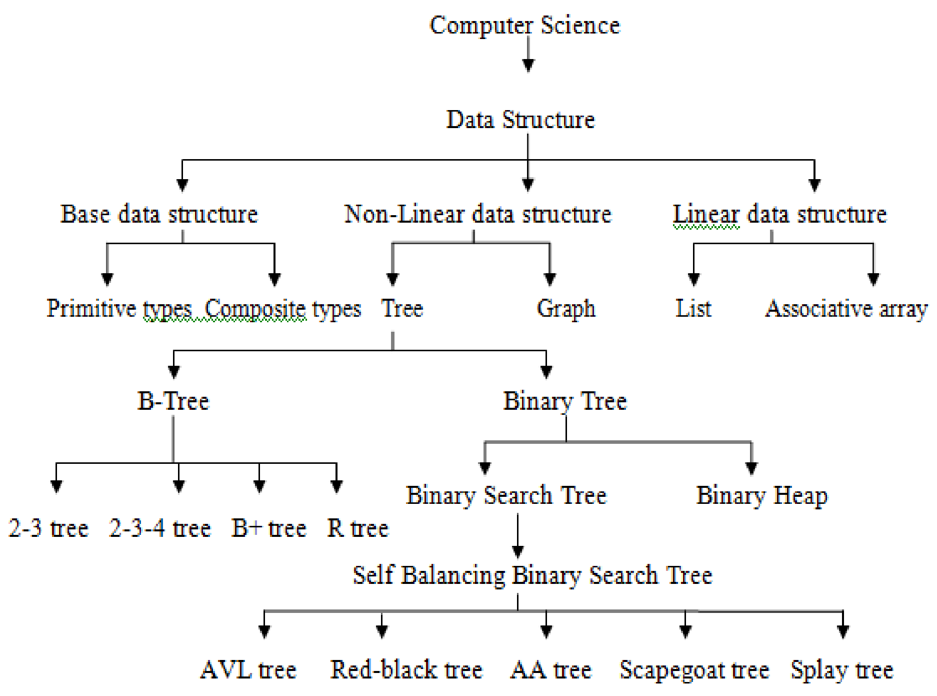


Fig. 1.1 The Hierarchical Structure Representing the Data Structure Family

2. Tree Data Structure

In computer science, a tree is a widely-used data structure that emulates a tree structure with a set of linked nodes [5]. The tree structure consists of nodes. The topmost node is called root node (node without parent), the nodes at the end of the tree are called leaf nodes (nodes without child), and other nodes are internal nodes.

Common operations that can be performed on any tree data structure are: Enumerating all the items, searching for an item, adding a new item at a certain position on the tree, deleting an item, removing a whole section of a tree (called pruning), adding a whole section to a tree (called grafting), finding the root for any node

Common uses: manipulate hierarchical data, make information easy to search, manipulate sorted lists of data

2.1 Classification

Tree data structure can be classified into various categories, out of which the most commonly used are:

1. B-Tree data structure
2. Binary tree data structure

2.1.1 B-Tree data structure- In computer science, a B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. It is most commonly used in databases and file systems [6]. In B-trees, internal nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes change. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation [7]. A B-tree is kept balanced by requiring that all external nodes are at the same depth. B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes. This usually occurs when most nodes are in secondary storage such as hard drives. By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often and efficiency increases [8]. A B-Tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties:

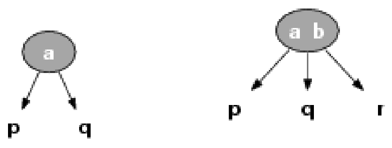
1. Every node has <= m children.
2. Every node (except root and leaves) has >= m/2 children.
3. The root has at least 2 children.
4. All leaves appear in the same level, and carry no information.
5. A non-leaf node with k children contains k – 1 keys

The B-Tree is actually a tree where the lower and the upper node limits (‘a’ and ‘b’) are given. Following are the various ways of implementing B-Tree.

a) 2-3 Tree
b) 2-3-4 or 2-4 tree
c) B+ tree

a) 2-3 Tree

2-3 tree in computer science is a B-tree that can contain only 2-nodes (a node with 1 field and 2 children) and 3-nodes (a node with 2 fields and 3 children). The leaf node is an exception to this. It has no children and two data element [9].



a 2-node a 3-node

Fig.2.1 Showing a 2-node and 3-node

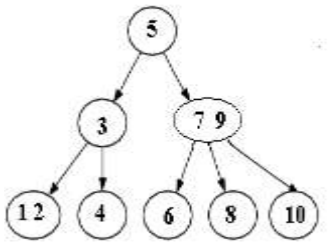


Fig. 2.2 An example of 2-3 tree.

Properties

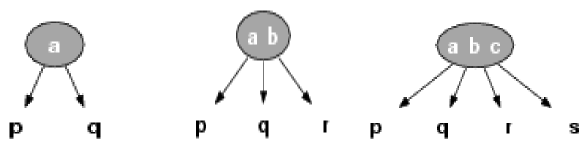
- Every non-leaf node has 2 or 3 children
- All leaves are at the same level (the bottom level)
- All data is kept in sorted order
- Every non-leaf node will contain 1 or 2 fields.

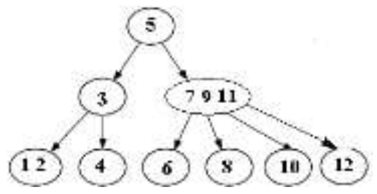
b) 2-3-4 Tree

A 2-3-4 tree in computer science is a B-tree of order 4. Like B-trees in general, 2-3-4 trees are a kind of self-balancing data structure that can be used as a dictionary. They can search, insert and delete in O(log *n*) time, where *n* is the number of elements in the tree. 2-3-4 trees are relatively difficult to implement in most programming languages because of the large number of special cases involved in operations on the tree. [9][10].

Properties - 2-3-4 trees store data as individual items called elements. These are grouped into nodes. Each node is either

- a 2-node, i.e. it contains 1 element and has 2 children, or
- a 3-node, i.e. it contains 2 elements and has 3 children, or
- a 4-node, i.e. it contains 3 elements and has 4 children.





a 2-node a 3-node a 4-node

Fig.2.3 Showing a 2-node, a 3-node and a 4-node **Fig. 2.4** An example of 2-3-4 tree.

c) B+ Tree- The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context. Given a storage system with a block size of *b*, a B+ tree which stores a number of keys equal to a multiple of *b* will be very efficient when compared to a binary search tree.

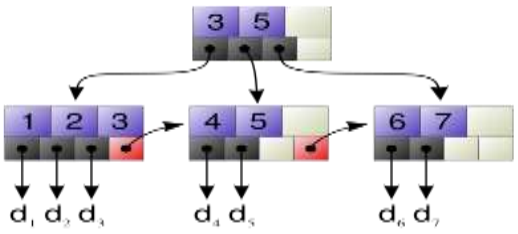


Fig. 2.5 A simple B+ tree example linking the keys 1-7 to data values d_1-d_7.

Properties

For a *b*-order B+ tree with *h* levels of index:

- The maximum number of records stored is $n = b^h$
- The minimum number of keys is $2(b / 2)^{h-1}$
- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations in the worst case
- Finding a record requires $O(\log_b n)$ operations in the worst case
- Removing a (previously located) record requires $O(\log_b n)$ operations in the worst case
- Performing a range query with *k* elements occurring within the range requires $O(\log_b n + k)$ operations in the worst case [11].

2.1.2 Binary tree data structure

In computer science, a binary tree is a tree data structure in which each node has at most two children. Typically the child nodes are called *left* and *right*. Binary trees are commonly used to implement binary search trees and binary heaps [12][13][14]).

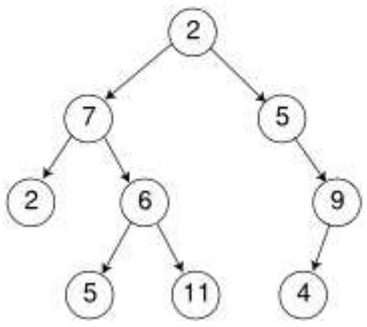


Fig 2.6 A simple binary tree of size 9 and height 3, with a root node whose value is 2.

The most commonly used binary trees are binary search tree and self balancing binary search trees. Following are the various ways of implementing Binary Tree.

i. Binary heap
ii. Binary search tree
iii. Self-balancing binary search trees
- AVL tree
- Red-black tree
- AA tree
- Scapegoat tree
- Splay tree

i. Binary heap

A Binary heap is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:

- The shape property: all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- The heap property: each node is greater than or equal to each of its children according to some comparison predicate which is fixed for the entire data structure.

ii. Binary search trees

In computer science, a binary search tree (BST) is a binary tree data structure which has the following properties:

- each node (item in the tree) has a value;
- a total order (linear order) is defined on these values;
- the left sub-tree of a node contains only values less than the node's value;
- the right sub-tree of a node contains only values greater than or equal to the node's value.

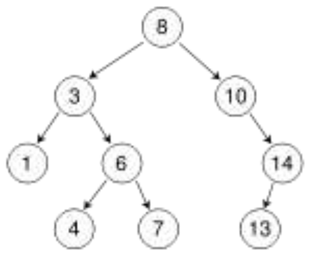


Fig. 2.7 A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multi sets, and associative arrays [15][16][17][18][19]).

iii. Self Balancing Binary Search Tree

In computer science, a self-balancing binary search tree or height-balanced binary search tree is a binary search tree that attempts to keep its height, or the number of levels of nodes beneath the root, as small as possible at all times, automatically. It is one of the most efficient ways of implementing associative arrays, sets, and other data structures [18][20][21]. Popular data structures implementing this type of tree include:

a) Red-black tree
b) AA tree
c) AVL tree
d) Splay tree
e) Scapegoat tree

a) Red-black tree

A red-black tree is a type of self-balancing binary search tree, a data structure used in computer science, typically used to implement associative arrays. It is complex, but has good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in O(log *n*) time, where *n* is the number of elements in the tree [22][23]. In red-black trees, the leaf nodes are not relevant and do not contain data. To save memory, sometimes a single *sentinel* node performs the role of all leaf nodes. All references from internal nodes to leaf nodes instead point to the sentinel node. Red-black trees, like all binary search trees, allow efficient in-order traversal of elements provided that there is a way to locate the parent of any node [24]. In addition to the ordinary requirements imposed on binary search trees, the following additional requirements of any valid red-black tree apply:

1. A node is either red or black.
2. The root is black.
3. All leaves are black. (The leaves are the null children.)
4. Both children of every red node are black.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes. (counting or not counting the null black nodes, it doesn't make a difference as long as you are consistent)

b) AA tree

An Arne Andersson tree (AA tree) in computer science is a red-black tree with one additional rule. Unlike red-black trees, RED nodes on an AA tree can only be added as a right sub-child. In other words, no RED node can be a left sub-child. The AA-Tree is considered as a simpler to code variant of the red-black tree and satisfies the following properties [25][26]:

1. Every node is colored red or black
2. The root node has to be black
3. Every leaf is a NIL node, and is colored black
4. If a node is red, then both its children are black
5. Every simple path from a node to a descendant leaf contains the same number of black nodes
6. Left children may not be red.

c) AVL Tree

AVL tree is named after G.M. Adelson-Velsky and E.M. Landis. An AVL tree is a special type of binary tree that is always "partially" balanced. An AVL tree is a binary tree in which the difference between the height of the right and left sub-trees (or the root node) is never more than one; therefore it is also called height-balanced. Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases. Additions and deletions may require the tree to be rebalanced by one or more tree rotations [27][28][29][30][31][32][33]).

d) Splay Tree

Splay Trees were invented by Sleator and Tarjan. A splay tree is a self-balancing binary search tree with the additional unusual property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log(n)) amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown [34][35][36][37][38][39]).

d) Scapegoat Tree

In computer science, a scapegoat tree is a self-balancing binary search tree, invented by Igal Galperin and Ronald L. Rivest. It provides worst-case O(log *n*) lookup time, and O(log *n*) amortized insertion and deletion time.

A binary search tree is said to be weight balanced if half the nodes are on the left of the root, and half on the right. An α-weight-balanced is therefore defined as meeting the following conditions:

$$\text{size(left)} <= \alpha*\text{size(node)} \quad \text{size(right)} <= \alpha*\text{size(node)}$$

A binary search tree that is α-weight-balanced must also be α-height-balanced, that is

$$\text{height(tree)} <= \log 1/\alpha(\text{NodeCount})$$

Scapegoat trees are not guaranteed to keep α-weight-balance at all times, however are always loosely α-height-balance in that

$$\text{height(scapegoat tree)} <= \log 1/\alpha(\text{NodeCount}) + 1$$

This makes scapegoat trees similar to red-black trees in that they both have restrictions on their height. They differ greatly though in their implementations [40][41][42].

3. Result

By counting the steps of various operations of B trees and binary trees structures we get the result shown in the table below.

Table 3.1 Average steps for operations of B-trees & Binary trees

	Average steps in Creation	Average steps in Insertion	Average steps in Deletion	Average steps in Searching
2-3 B Tree	2.6	5	6	1.6
2-3-4 B Tree	2.4	3	4	1.8
B+ Tree	4.2	4.6	3.4	3
Binary Heap	2.6	4.8	4	3
Binary Search Tree	2.2	3	3.6	2.2
AVL Tree	3	4.6	5	2.2
RB Tree	3.4	4.8	5.4	2.2
AA Tree	4.2	6.2	5.4	2.2
Scapegoat Tree	3	4.8	3.2	2.2
Splay Tree	3.4	6	6.2	5

3.1 Conclusion

From this table we can easily find out that:

i) 2-3 B tree is best for search-oriented applications like dictionary or directory.
ii) 2-3-4 tree is best for insertion oriented applications.
iii) Binary search tree is best where we frequently have to create new list of elements or insert new elements.
iv) Scapegoat tree is best for deletion-oriented applications.
v) We can also say that for the applications where all operations are of nearly equal importance we may use binary search tree or 2-3-4 B tree.

4 Future scope

The comparison made among different tree data structure is based on a single set of random data. The use of single random data can't show all the advantages or disadvantages of the tree structures depending upon some of their special characteristics.
In future it is possible to compare these tree structures after taking into consideration their various special characteristics, so that we get the complete data about them to find out their use in some particular applications. This comparison can be done easily by taking more than one set of data elements for all of them. These sets of data elements should explore all the properties of these tree data structures.

REFERENCES

[1] Paul E. Black, "data structure", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology, 2004.

[2] Bruno R. Preiss, Data Structures and Algorithms with Object-Oriented Design Patterns in C++, Java.

[3] Online definition at http://www.devmaster.net/wiki/Scene_graph.html

[4] ROCW (Rai Open Course Ware), **"**Introduction to data structure" at www.rocw.raifoundation.org/computing/BCA/Datastructure-alogrithm/lecture-notes/Lecture-01.pdf.

[5] Donald Knuth. The Art of Computer Programming: Fundamental Algorithms, Third Edition. Addison-Wesley, 1997. Section 2.3: Trees, pp.308–423.

[6] Jaideep Srivastava, C. V. Ramamoorthy: Efficient Algorithms for Maintenance of Large Database. ICDE 1988: 402-408

[7] Bayer R. and McCreight E., Organization of large ordered indexes. Acta Inf. 1 (1972), 173-189.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Chapter 18: B-Trees, pp.434–454.

[9] Michael T. Goodrich and Roberto Tamassia, "Data Structures and Algorithms in Java", 1998 at: http://www.cs.purdue.edu/homes/ayg/CS251_FALL_99/HTML/ hw8addnote .html.

[10] Walker Ellen, Hiram College, notes from: http://cs.hiram.edu /~walkerel/cs201 /234 tree.ppt.

[11] Silberschatz, Abraham, and Henry F. Korth, and S. Sundarshan. Database System Concepts. New York: McGraw-Hill, 2006.

[12] Donald Knuth. The art of computer programming vol 1. Fundamental Algorithms, Third Edition. Addison-Wesley, 1997. Section 2.3, especially subsections 2.3.1–2.3.2 (pp.318–348).

[13] Kenneth A Berman, Jerome L Paul. Algorithms: Parallel, Sequential and Distributed. Course Technology, 2005. Chapter 4. (pp.113–166).

[14] Weiss Mark Allen, Data structures and problem solving using java, (3rd ed), Addison- Wesley, 2006. Section 18.2

[15] Kingsley Sage, University of Sussex, Lecture notes from http://www.informatics .sussex. ac.uk/users/khs20/dats/pdfs/Lecture5.pdf.

[16] Weiss Mark Allen, Data structures and problem solving using java, (3rd ed), Addison-Wesley, 2006. Chapter 19.

[17] Heger, Dominique A. (2004), "A Disquisition on The Performance Behavior of Binary Search Tree Data Structures", European Journal for the Informatics Professional **5** (5), http://www.upgrade-cepis.org/issues/2004/5/up5-5Mosaic.pdf

[18] Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001 Chapter 12: Binary search trees, pp.253–272. Section 15.5: Optimal binary search trees, pp.356–363.

[20] Sleator D.D., and Tarjan R.E., Self-adjusting binary trees. In Proceedings of the 15th Annual ACM Symposium on Theory of Computing (Boston, Mass., Apr. 25-27**)**. ACM, New York, 1983. Pages 235-245.

[21] Sleator D.D., and Tarjan R.E., A data structure for dynamic trees. J. Cmp. Syst. Sci. 26 (1983). Pages 362-391.

[22] Rudolf Bayer, Symmetric Binary B-Trees: Data Structures and Maintenance Algorithms, Acta Informatica, 1:290-306, 1972.

[23] Leo J. Guibas and Robert Sedgewick, A Dichromatic Framework for Balanced Trees, Proceedings of the 19th Annual Symposium on Foundations of Computer Science, pages 8-21. IEEE Computer Society, 1978.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Chapter 13: Red-Black Trees, pages 273–301.

[25] Andersson A., Balanced search trees made simple. In Proc. Workshop on Algorithms and Data Structures, pages 60--71. Springer Verlag, 1993.

[26] Online notes of Binary search trees from http://www.cs.usu.edu/~allan/DS/Notes /Ch19.pdf.

[27] Adelson-velskii G.M., and Landis E. M., An algorithm for the organization of information, Sov. Math. Dokl. 3 (1962). Pages 1259-1262**.**

[28] Knuth D.E., The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997. Pages 458–475 of section 6.2.3: Balanced Trees.

[29] Levitin Anany, The Design and Analysis of Algorithms, Addison Wesley 2003.

[30] Hart, Chris 1998, Doing a literature review: releasing the social science research imagination, Sage Publications, London.

[31] Robert Sedgewick, *Algorithms*, Addison-Wesley, 1983, page 199, chapter 15: Balanced Trees.

[32] Thomas A. Standish, *Data Structure Techniques*, Addison-Wesley Longman Publishing Co., 1980, section 3.7.3

[33] Gonnet Gaston H. & Baeza-Yates Ricardo , Handbook of Algorithms and Data Structures, Addison-Wesley, 1984, section 3.4.1.

[34] Sleator Daniel D. & Tarjan Robert E., "Self-Adjusting Binary Search Trees", Journal of the ACM **32** (3), 1985: pp. 652-686.

[35] Cole R., B. Mishra, J. Schmidt & A. Siegel, On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences. SIAM Journal on Computing 30, 2000: pages 1-43.

[36] Cole R., On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof. SIAM Journal on Computing 30, pages 44-85, 2000.

[37] Tarjan R. E., Sequential access in splay trees takes linear time. Combinatorica 5, pages 367-378, 1985.

[38] Amr Elmasry. On the sequential access theorem and deque conjecture for splay trees. Theor. Comput. Sci. 314(3), pages 459-466, 2004.

[39] Fredman M. L., Johnson D. S., McGeoch L. A., and Ostheimer G., “Data structures for traveling salesmen,” J. Algorithms, vol.18, pp.432–479, 1995.

[40] Galperin Igal & Rivest Ronald L., "Scapegoat trees", Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms. 1993: pp. 165-174.

[41] Galperin Igal, On Consulting a Set of Experts and Searching, Doctoral Degree Thesis, 1996.chapter 3.

[42] Andersson Arne, "General balanced trees", Journal of Algorithms 30, 1999: pp. 1-28.