# An Efficient List Implementation

Arne Andersson
Uppsala University
and
Stefan Nilsson
Royal Institute of Technology

We present an efficient implementation of the Java List interface, supporting all basic operations in $O(\log n)$ time on a list of size $n$. Our experiments confirm that the data structure is competitive also in practice.

Categories and Subject Descriptors: F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Data Structures

Additional Key Words and Phrases: List, index tree, label tree, general balanced tree

## 1. INTRODUCTION

In this article we present an implementation of the Java `List` data structure that supports all basic operations in $O(\log n)$ time on a list of size $n$. A `List` in Java is defined as an interface and two standard implementations are provided: `ArrayList` and `LinkedList`. Unfortunately, both of these have poor running times for some basic operations, such as insertion and deletion.

The Java `List` interface offers a rich set of operations, but we will restrict our attention to a small subset, since all other operations can be efficiently implemented on top of these basic operations.

—`size()` Returns the number of elements in the list.

—`Object get(int i)` Returns the element at the specified position in the list.

—`void add(int i, Object o)` Inserts the specified element at the specified position in the list.

—`Object remove(int i)` Removes the element at the specified position in the list.

—`int indexOf(Object o)` Returns the index in the list of the first occurrence of the specified element, or $-1$ if the list does not contain this element.

Name: Arne Andersson
Affiliation: Computing Science Department, Information Technology, Uppsala University
Address: SE-751 05 Uppsala, Sweden
Name: Stefan Nilsson
Affiliation: Department of Numerical Analysis and Computing Science, Royal Institute of Technology
Address: SE-100 44 Stockholm, Sweden

| operation | array | linked list | index tree | label tree |
|---|---|---|---|---|
| `int size()` | 1 | 1 | 1 | 1 |
| `Object get(int i)` | 1 | $n$ | $\log n$ | $\log n$ |
| `void add(int i, Object o)` | $n$ | $n$ | $\log n$ | $\log n$ |
| `Object remove(int i)` | $n$ | $n$ | $\log n$ | $\log n$ |
| `int indexOf(Object o)` | $n$ | $n$ | $n$ | $\log n$ |

Fig. 1. Time bounds for basic list operations.

The problem is somewhat similar to *list indexing*. In list indexing, we have only one copy of each element, and a reference to the nearest neighbor is given at insertion. Dietz [Dietz 1989] gave a solution in which all operations run in $O(\log n/\log\log n)$ amortized time. This is optimal due to a lower bound of Fredman and Saks [Fredman and Saks 1989]. Our goal is not to achieve a solution with optimal asymptotic bounds, but a simple and practical implementation.

The standard Java `ArrayList` implementation consists of a growable array. Hence, `get()` operates in constant time, while `add()`, `remove()`, and `indexOf()` have linear time complexity. `LinkedList` is a doubly linked list and hence all basic access operations, including `get()`, have linear worst-case time.

We offer two new implementations. The first one, `IndexTreeList`, consists of an index tree [Cormen et al. 1990] implemented as a general balanced tree [Andersson 1999]. `get()`, `add()`, and `remove()` run in logarithmic time, while `indexOf()` has linear time complexity. The second data structure, `LabelTreeList`, is an extension of the first that achieves logarithmic amortized time for all basic operations at the expense of using more memory. A summary of these running times is shown in Figure 1.

In Section 2 we present the index tree data structure and discuss how to implement it efficiently with a general balanced tree. In Section 3 we show how to achieve logarithmic time for all operations with a special node labeling technique. In Section 4 we present experimental data to support our claim that the algorithm is practically useful.

## 2. AN INDEX TREE IMPLEMENTATION USING A GENERAL BALANCED TREE

Using an index tree to represent a list is a well known technique [Cormen et al. 1990]. An index tree is a binary search tree augmented with an addition field in each node that indicates the size of the subtree rooted at this node. Given this information it is possible to compute the index of a node during a top-down traversal from the root to the node.

To achieve logarithmic time search and update operations on this we may use any balanced binary tree. We have chosen the general balanced tree, since it is fast and rebalancing is made by partial rebuilding . This turns out to be an important feature when we extend the data structure.

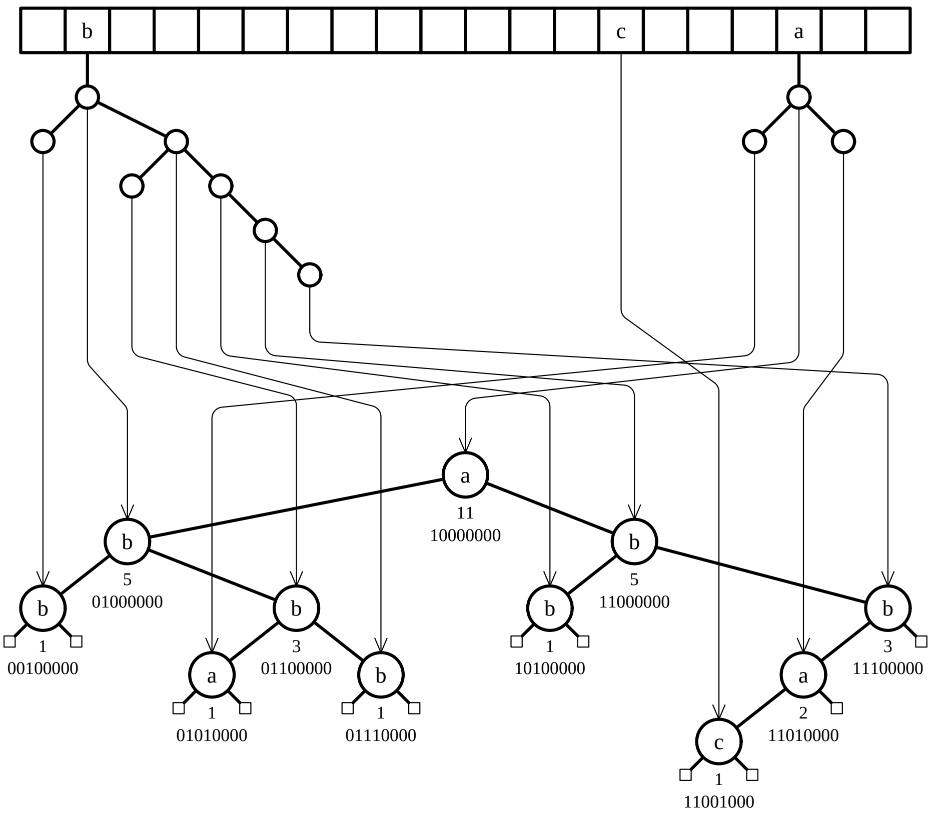

Fig. 2. The label tree data structure.

## 2.1 General Balanced Trees

The general balanced tree [Andersson 1996; Andersson 1999] is a plain vanilla binary search tree; no stored balance information is needed in the nodes and the tree is allowed to take any shape as long as its height is at most a constant times $\log n$. Still, it can be maintained efficiently at $O(\log n)$ amortized cost per update.

For the sake of completeness we give a short overview of GB-trees. Following the notation in [Andersson 1999], we do not distinguish between nodes and subtrees; the subtree $v$ is the subtree rooted at the node $v$ and $T$ is the entire tree (or the root of the tree). Each internal node contains one element and all leaves are empty. The number of edges on the path from the root $T$ to the node $v$ is the *depth* of $v$. The largest number of edges from a node $v$ to a leaf is the *height* of $v$, denoted $h(v)$. The number of leaves in $v$ is denoted $|v|$. (A tree containing $n$ elements has $n + 1$ leaves.)

A general balanced tree, or GB-tree, or GB($c$)-tree, is based on a relaxed balance criterion; essentially it can take any shape as long as its height is logarithmic. For some constant $c$, we require that

$$h(T) \leq \lceil c \log |T| \rceil$$

The basic philosophy for maintaining GB-trees is

*Do nothing until the tree becomes too high.*

During insertion only the path down to the inserted node may become too long. Balance will be restored by rebuilding an appropriate subtree, located by a depth-first search along the insertion path. This strategy does not hold for deletion, but the skewness caused by deletions can be handled by rebuilding the entire tree periodically. There is no need to store any balance information in the nodes; only two global integers, telling the size of $T$ and the number of deletions made since the last time $T$ was globally rebuilt, are needed.

Updates are performed in the following way:

—**Insertion** If the depth of the new leaf exceeds $\lceil c \log |T| \rceil$, we traverse the path bottom-up. At the lowest node $v$, $h(v) > \lceil c \log |v| \rceil$, we make a partial rebuilding. (There will always be a node $v$ satisfying $h(v) > \lceil c \log |v| \rceil$, since $h(T) > c \log |T|$.)

—**Deletion** When the number of deletions made since the last global rebuilding is larger than $d|T|$, $d$ is a constant, we make a global rebuilding.

To use the GB-tree as an index tree, we only need to add a field in each node that indicates the size of the subtree rooted at this node.

## 3. IMPLEMENTING THE `INDEXOF()` OPERATION EFFICIENTLY

Achieving logarithmic time also on the `indexOf()` operation is a bit more complicated. If the keys had been unique, a natural solution would have been to use a hash table, where each key has a pointer to its node in the index tree. Traversing the index tree upwards, assuming the presence of parent pointers, we can compute the index in logarithmic time.

However, the possibility of duplicate elements introduces complications. Now, we need to maintain the proper order among all duplicates, recall that `indexOf()` requires us to return the smallest index. And, in order to do this, we must be able to compare the indices of two elements. Using parent pointers, each such comparison would take logarithmic time.

Instead, we avoid parent pointers and use a list labeling technique. Each node in the index tree has a label; the ordering of the labels is consistent with the ordering of indices. Given two list entries $x_i$ and $x_j$, with index $i$ and $j$, $i < j$, the label stored in the node representing $x_i$ is smaller than the label in the $x_j$ node. Knowing the label of an element, we can view the index tree as a binary search tree and locate the proper node while we at the same time compute its index.

As the label of a node we use a binary string representing the path from the root of the index tree to this node. A zero bit indicates a left turn and a one indicates a right turn to. To achieve the correct ordering, we append the end marker "1000..." to the binary string.

The way a general balanced tree is maintained, by partial rebuilding, makes it particularly suitable for maintaining such labels; each time a subtree is rebuilt we only need to relabel the nodes involved in the rebuilding. From the label of the subtree's root, we can assign labels in the reconstructed subtree during a simple traversal.

We represent these labels with integers of type `long`. Since the height of the GB-tree is bounded by $c \log n$, 64 bits should be sufficient. For example, for $n = 2^{31}$ (32-bit signed integers are used for indexing in Java) and $c = 1.35$, a typical value for the rebalancing quotient of the GB-tree, the maximum number of bits is 43. To facilitate the comparison of labels we do not use the most significant bit of the long integer. Java's integer type is signed and the first bit is a sign bit. If the sign bit is 0, we may use standard integer comparison to compare labels.

In addition to the labels, we still need a hash table. In this case, all occurrences of the same element are stored together. Each hash table entry consists of an element in the list and a binary search tree, ordered by label (and hence by index). In our implementation we use a splay tree. Since our list labeling technique requires relabeling now and then, we do not store the labels explicitly in the splay trees. Instead, each node in the tree contains a pointer to the corresponding node in the index tree. Thus, even if the labels (and indices) change by updates, the splay trees can be maintained in proper order; the comparisons in the splay tree are always made by looking up the current labels in the index tree.

A complete picture of the data structure, including the GB-tree with labels and size information and the hash table with splay trees, is shown in Figure 2.

The adaptive behavior of splay trees seems suitable, since the Java List interface only offers operations that access the minimum and maximum element of the splay tree. In a scenario with multiple access to the minimum and maximum value without intermediate updates, all but the first access will be performed in constant time. Finally, a splay tree can be built in linear time when recreating the data structure from file, assuming that the list is stored sequentially according to index.

This data structure is rather space consuming. In addition to the index tree, each list entry requires a hash table entry and an entry in a splay tree. However, we have chosen a simpler representation for hash table entries containing only one pointer. This is beneficial when the list contains few repeated elements, which is not unusual in many applications. Hash table entries containing only one reference to the index tree are represented by a direct pointer to the index tree node. The benefits of this approach are twofold. Unique list entries will not need extra space for splay tree nodes. Also, the running time will decrease, since less memory allocation will take place.

## 4. EXPERIMENTS

Even though it is easy to give asymptotic time bounds for the list data structures disussed in this paper, it is still worthwhile to implement the algorithms and compare running times experimentally. In particular, we want to investigate how much the overhead of extra memory allocation and more complicated algorithms influences the behavior. It is also interesting to investigate the breakpoint where an index tree becomes more efficient than an array. Without experiment, this would be quite difficult to estimate: the array requires fewer memory allocations, the access operation is simpler, and Java has a special low level operation for array copy. Finally, we hope that the implementation will make the data structure more readily available and useful to practitioners.

## 4.1 method

The experiments were run on an Ultra 5 workstation using the SUN JDK 1.2 development kit. We have refrained from low-level optimizations to keep the code clean. Code is available from `http://www.nada.kth.se/~snilsson`

We performed the following four experiments.

—**add** Start with an empty list and insert $n$ elements into the list at random positions.

—**remove** Start with a list of size $n$ and make $n$ successive calls `remove(i)`, where $i$ is a random number such that $0 \leq i < m$, where $m$ is the size of the remaining list.

—**get** Start with a list of size $n$ and make $n$ successive calls `get(i)`, where $i$ is a random number such that $0 \leq i < n$.

—**indexOf** Start with a list of size $n$ and make $n$ successive calls `indexOf(elem)`, where `elem` is chosen at random from a data space of size $S$.

Both the data and the access order was precomputed and stored in arrays. The running times were obtained from the system clock and times are reported in milliseconds. Each experiment was performed five times and we report the minimum running time. The data to be stored in the list consisted of integers randomly drawn from an interval $0 \ldots S-1$. In fact, the numbers were represented as decimal character strings. For the first three experiments the nature of the data doesn't affect running times and we have used the value $S = 10000000$. The `indexOf()` method, however, is affected by the number of repeated elements in the list. In this case we performed the experiments with $S$ equal to 10, $1,000$, and $10,000,000$.

## 4.2 discussion

The results for **add** and **remove** are similar. Figure 3 shows the time to build a list by adding elements at random positions. As expected, the array and linked list implementations exhibit a quadratic performance and hence are feasible for small instances only. A closeup of this plot is shown in Figure 4. Notice that the breakpoint where it is faster to build an index tree than an array is low, around 1000 elements.

Figure 5 shows that the get operation behaves as expected. The linked list has linear access time, the index tree logarithmic, while the array has constant time access. For small instances the differences in running time are small. For large lists containing 100000 elements the access time is about 15 times slower for an index tree than an array. For small lists with only a few thousand elements the ratio is about 5.

The `indexOf` operation depends on the number of different keys stored in the list. If there are very few keys, and hence many repetitions, the data structures are evenly matched as can be seen in Figure 6. With fewer repetitions the picture is very different. Figure 7 shows statistics for a list consisting of elements from a data space of size $S = 1000$. As expected, the search algorithms based on linear search become unfeasibly slow for large lists. The label tree is the only feasible alternative.

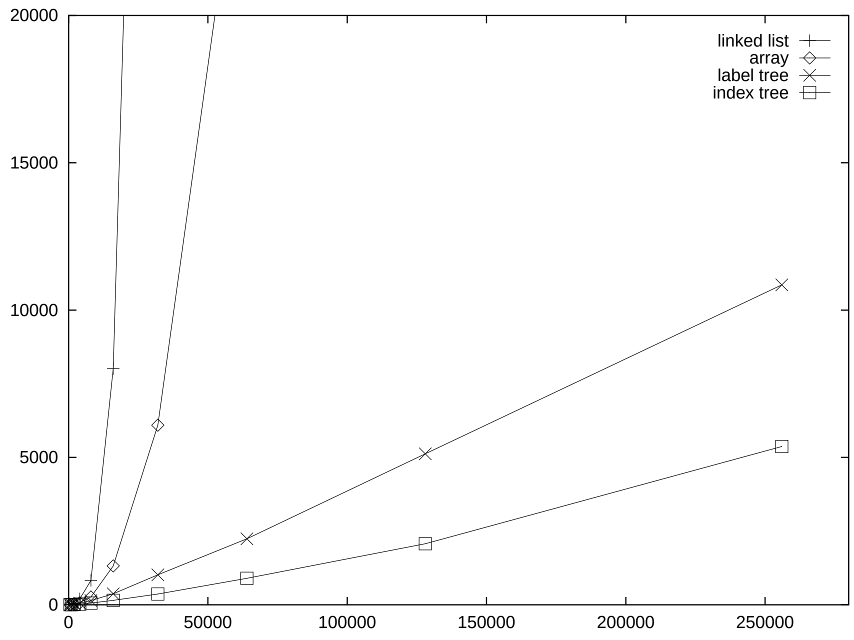

Fig. 3. **add** The time (ms) to build a list of size $n$ by adding elements at random positions.

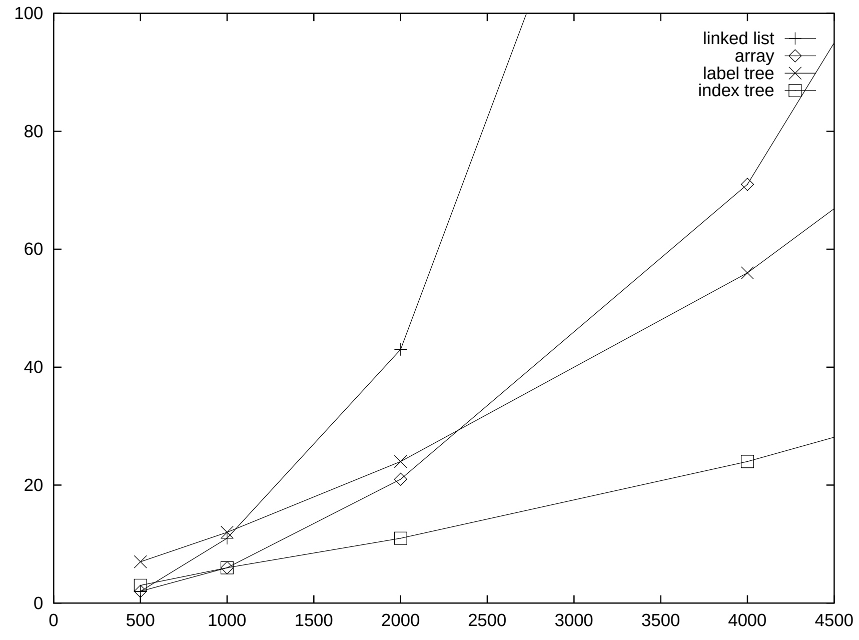

Fig. 4. Closeup of Figure 3.

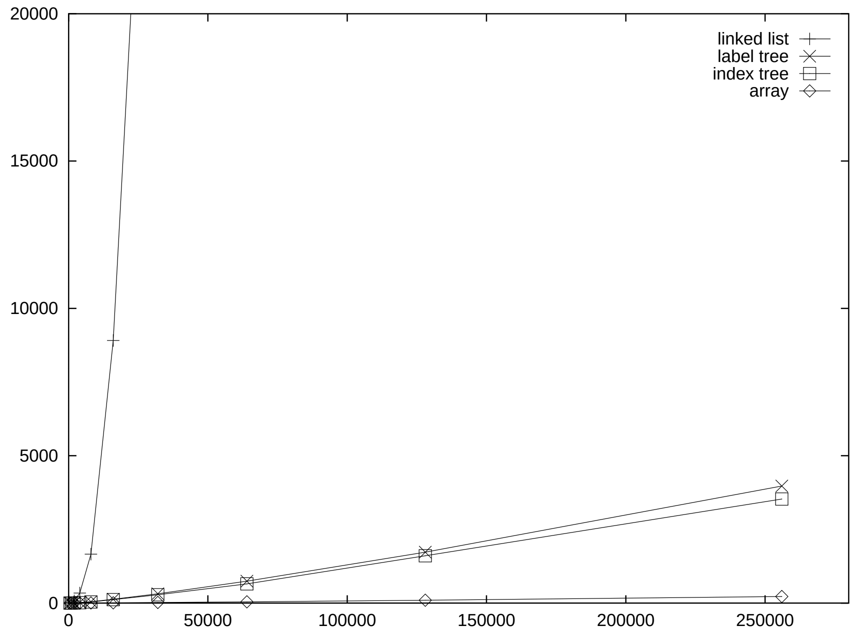

Fig. 5. **get** The time (ms) to access $n$ random positions in a list of size $n$.

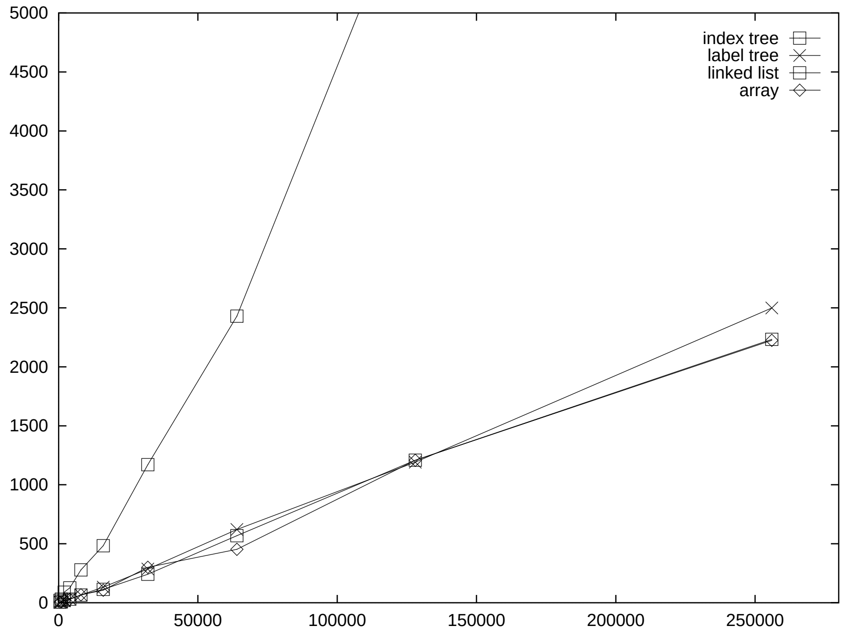

Fig. 6. **indexOf** The time (ms) to locate the minimum index of $n$ random elements in a list of size $n$. The size of the data space $S = 10$.

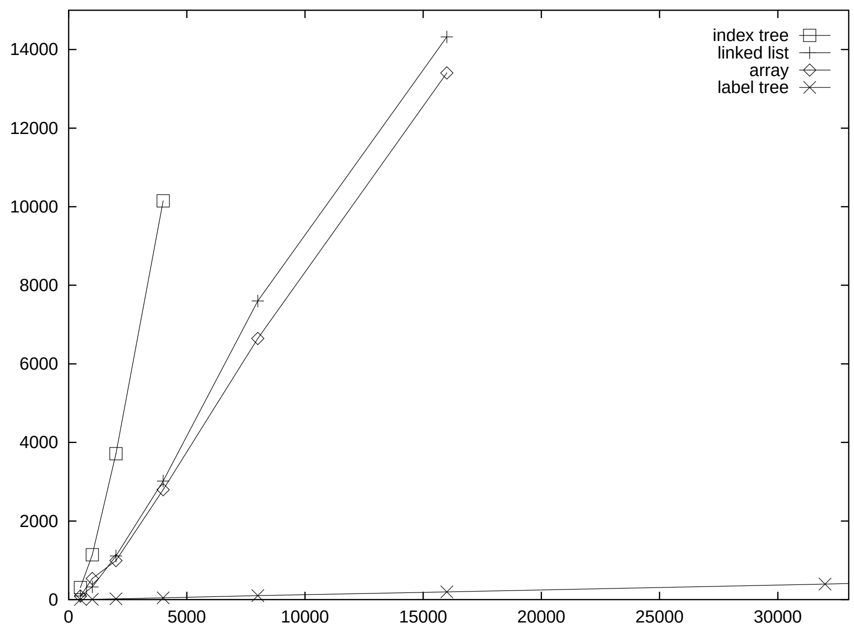

Fig. 7. **indexOf** The time (ms) to locate the minimum index of $n$ random elements in a list of size $n$. The size of the data space $S = 1000$.

## 5. CONCLUSIONS

Our main goal has been to provide a useful list data structure with good overall performance and no bottlenecks. We believe that the current implementations meet these design goals and we hope that the code will be useful in practice. As a general purpose list implementation, we recommend the label tree. It has no weak spots, all basic operations have logarithmic time complexity, and the space overhead, as compared to Java's standard list implementations, is moderate.

To turn the code into a complete general purpose library class, some additional routine work remains, such as implementing cloning, serialization, and more efficient iterators. It is possible to iterate through the data structure in linear time by maintaining a stack indicating the path from the root of the tree to the current node. Serialization, that is, storing the data structure on file, can be done in linear time. In the file we only need to store the list elements in sequential order, neither the trees, nor the hash table needs to be explicitly stored. The GB-tree is build in linear time using the rebalancing operation on the complete tree and each splay tree insertion takes constant time, since the indices are inserted in increasing order.

## REFERENCES

Andersson, A. 1996. Which flavor of balanced trees? Plain vanilla! In *DIMACS Implementation challenge* (1996).

Andersson, A. 1999. General balanced trees. *Journal of Algorithms 30*, 1–18.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms.* McGraw-Hill.

DIETZ, P. 1989. Optimal algorithms for list indexing and subset rank. In *Proc. 1989 Workshop on Algorithms and Data Structures* (1989), pp. 39–46. Springer-Verlag. LNCS 382.

FREDMAN, M. AND SAKS, M. 1989. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st ACM Symposium on the Theory of Computing* (1989), pp. 354–354.