Algorithm Design and Software Libraries: Recent Developments in the LEDA Project

Kurt Mehlhorn Stefan Näher *

Abstract

LEDA (Library of Efficient Data Types and Algorithms) is an ongoing project which aims to build a library of the efficient data structures and algorithms used in combinatorial computing [12]. We discuss three recent aspects of the project: The cost of flexibility, implementation parameters, and augmented trees.

Keyword Codes: D.2.0; D.1.5; F.2.2
Keywords: Software Engineering, General; object-oriented Programming; Nonnumerical Algorithms and Problems

1 Introduction

We will treat the topic on the basis of a concrete example: the LEDA (Library of Efficient Data Types and Algorithms) software library. The authors started the LEDA project in 1989 as an attempt to narrow the gap between algorithm research, teaching and implementation. The project aims to build a library of the efficient data structures and algorithms used in combinatorial computing. The rationale of the project is given in [12] and briefly reviewed in section 2. LEDA is freely available by anonymous ftp for educational and research use since the spring of 1991 (server: ftp.cs.uni-sb.de, directory: /pub/LEDA).

The design of a library rises a large number of issues ranging from software engineering to algorithm design. In this paper we discuss three recent but unrelated aspects of the project to illustrate the wide span of problems encountered.

LEDA is designed to be a general purpose tool and therefore programs based on it tend to be less efficient than special implementations. In section 3 we discuss the major source of inefficiency in LEDA and show that this overhead is relatively small (never more than a factor of three in time and two in space).

*Max-Planck-Institut für Informatik, Saarbrücken, Germany; this research was partially supported by ESPRIT II Basic Research Actions Program, under contract No. 3075 (project ALCOM).

Elegance and ease of use is a major concern in library design. For many data types in combinatorial computing there are many possible implementations. In the current version of LEDA there is no elegant mechanism for choosing between different implementations of the same data type. In section 4 we describe such a mechanism based on multiple inheritance. It will be available in the next version of LEDA.

Augmented trees are abundant in computational geometry. Examples of augmented trees are range trees [3, 21], segment trees [4], interval trees [8], and dynamization of order decomposable searching problems [17]. Asymptotically efficient realizations of augmented trees are usually based on weight-balanced trees [16]; in the case of range and segment trees implementations are available in LEDA. Skip lists [18, 19] were recently proposed as an alternative to balanced trees for one dimensional dictionaries. We argue in section 5 that skip lists can also serve as the basis for augmented tree schemes. They lead to simpler and more efficient programs. Related but weaker results were also obtained by [10].

2 LEDA, a Brief Review

LEDA is a growing library of efficient data types and algorithms used in combinatorial computing. The main features are:

- LEDA provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. This collection includes most of the data types and algorithms described in the text books of the area (e.g. [1, 11, 6]), i.e., stacks, queues, lists, sets, dictionaries, ordered sequences, partitions, priority queues, directed, undirected, and planar graphs, lines, points, planes and basic algorithms in graph and network theory and computational geometry.

- LEDA gives a precise and readable specification for each of the data types and algorithms mentioned above. The specifications are short (typically, not more than a page), general (so as to allow several implementations), and abstract (so as to hide all details of the implementation). For many efficient data structures access by position is important. In LEDA, we use the concept of an item to cast positions into an abstract form, cf. Figure 1.

- LEDA contains efficient implementations for each of the data types, e.g., Fibonacci heaps and redistributive heaps for priority queues, red-black trees and dynamic perfect hashing for dictionaries, ...

- LEDA contains a convenient data type graph that allows to write programs for graph problems in a form close to the typical text book presentation. It offers the standard iterations such as "for all nodes v of a graph G do" (written $forall\_nodes(v,G)$) or "for all neighbors w of v do" (written $forall\_adj\_nodes(w,v)$), it allows to add and delete vertices and edges and it offers arrays and matrices indexed by nodes and edges,..., cf. Figure 2. A long term goal is to realize the equation "Algorithm + LEDA = program".

Priority Queues

An instance Q of the data type *priority_queue* is a collection of items (*pq_item*). Every item contains a key from a type K and an information from a type I. K is called the key type of Q and I is called the information type of Q. The number of items in Q is called the size of Q. If Q has size zero it is called the empty priority queue. We use $< k, i >$ to denote the *pq_item* with key k and information i. I must be linearly ordered.

1. Creation of a priority queue

priority_queue(K, I) Q;

creates an instance Q of type *priority_queue*(K, I) and initializes it to the empty queue.

2. Operations on a priority_queue Q

K	Q.key(*pq_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in Q.
I	Q.inf(*pq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in Q.
pq_item	Q.insert(K k, I i)	adds a new item $< k, i >$ to Q and returns it.
pq_item	Q.find_min ()	returns an item with minimal information (nil if Q is empty)
void	Q.del_item(*pq_item it*)	removes the item *it* from Q. *Precondition*: *it* is an item in Q.
K	Q.del_min()	removes an item with minimal information from Q and returns its key. *Precondition*: Q is not empty.
pq_item	Q.decrease_inf(*pq_item it*, I i)	makes i the new information of item *it* *Precondition*: *it* is an item in Q and i is not larger than $inf(it)$.
void	Q.clear()	makes Q the empty priority queue
bool	Q.empty()	returns true, if Q is empty, false otherwise
int	Q.size()	returns the size of Q.

Priority queues are implemented by Fibonacci Heaps. Operations insert, del_item, del_min take time $O(\log n)$, find_min, decrease_inf, key, inf, empty take time $O(1)$ and clear takes time $O(n)$, where n is the size of the queue. The space requirement is $O(n)$.

Figure 1: The specification of priority queues: A priority queue is viewed as a set of containers called *pq_items*. Each container holds a key-information pair. An insertion takes a pair (k, i), creates a new container to hold the pair, and returns the name of the container, i.e., the *pq_item* $< k, i >$. The pair can be accessed through the item (in operations key, inf, del_item, and decrease_inf).

We refer the reader to the article [12] and the manual [14] for more details. LEDA is freely available for educational and research use since the spring of 1991.

3 The Cost of Flexibility

Most of the data types in LEDA are parameterized. For example, $priority\_queue(K, I)$ is the data type of priority queues storing pairs of keys and information from $K \times I$. Parameterized data types are powerful and flexible, since they can be customized by providing specific types for the formal type parameters. Figure 2 shows an application of the data type $priority\_queue(node, int)$ in Dijkstra's shortest paths algorithm. On the other hand this flexibility has its cost.

One of the basic design decisions in the LEDA project was that all implementations of data types and algorithms are supplied as pre-compiled object code modules that only have to be linked with application programs but never recompiled. However, the implementation of a parameterized data type depends on the actual type parameters. Each data structure for $priority\_queue(K, I)$, for instance, requires a linear order on the information type I, and needs to know how to compare two values of type I. So there has to be some kind of flow of information from the user-defined actual parameterized data type, e.g., $priority\_queue(node, int)$, back to the implementation, e.g., Fibonacci heaps. In LEDA, we achieve this by using the object-oriented programming techniques *inheritance* and *dynamic binding*.

Implementations, i.e., data structures, for parameterized data types are C++ classes that leave all routines depending on the actual parameter types undefined by declaring them as virtual functions. In the priority_queue example, the Fibonacci heap data structure (class f_heap) uses a virtual function cmp_inf to compare informations. Any actual priority queue type $priority\_queue(K, I)$ is derived from f_heap and redefines the cmp_inf function according to the linear order of I. Similar virtual functions are used to copy or destroy objects of types unknown to the implementation. More details can be found in [15].

As a consequence of the above described method every compare operation performed by the implementation results in a C++ function call. This can slow down computation if I is a simple type like int with an efficient built-in compare operation. But fortunately, as experiments show, the resulting overhead is not too large. For example, in the case of Dijkstra's algorithm a special integer Fibonacci heap implementation decreases the running time by at most a factor of 3. For space, the overhead induced by the library is at most a factor of two. Furthermore, in the next version of LEDA (cf. section 4) users can easily provide their own favorite implementations or derive special more efficient implementations from the LEDA data structure sources.

```
(1)  #include <LEDA/graph.h>
(2)  #include <LEDA/prio.h>
(3)  declare2(priority_queue,node,int)
(4)  declare(node_array,pq_item)
(5)  void DIJKSTRA(graph& G, node s, edge_array(int)& cost,
(6)                      node_array(int)& dist, node_array(edge)& pred )
(7)  { priority_queue(node,int) PQ;
(8)    node_array(pq_item) I(G, nil);
(9)    int c;
(10)   node u, v;
(11)   edge e;
(12)   forall_nodes(v, G)
(13)   { pred[v] = nil;
(14)     dist[v] = infinity;
(15)     I[v] = PQ.insert(v, dist[v]);
(16)   }
(17)   dist[s] = 0;
(18)   PQ.decrease_inf(I[s], 0);
(19)   while (!PQ.empty())
(20)   { u = PQ.del_min()
(21)     forall_adj_edges(e, u)
(22)     { v = G.target(e);
(23)       c = dist[u] + cost[e];
(24)       if ( c < dist[v])
(25)       { dist[v] = c;
(26)         pred[v] = e;
(27)         PQ.decrease_inf(I[v], c);
(28)       }
(29)     }
(30)   }
(31) }
```

Figure 2: Dijkstra's shortest path algorithm: The algorithm takes a graph $G = (V, E)$, a source node s and a function $cost : E \longrightarrow \mathbb{N}$ (given as an $edge\_array(int)$, i.e., an array indexed by edges and holding integers) and returns functions $dist : V \longrightarrow \mathbb{N}$ and $pred : V \longrightarrow E$ ($dist[v]$ is the length of the shortest path from s to v and $pred[v]$ is the last edge on a shortest path to v.) It uses internally a priority queue PQ from nodes to integers (the type $priority\_queue(node, int)$ is declared in line (3) and the priority queue PQ is created and initialized to the empty queue in line (7)) and a node array I of pq_items (the type $node\_array(pq\_item)$ is declared in line (4), and the array I is defined to have domain V and initial values nil in line (8)). The node array I stores for each node $v \in V$ the corresponding item in the priority queue (line (15)).

4 Implementation Parameters

An abstract data type (ADT) specifies only the functional properties of a data type and hides the implementation from the user. Figure 1 gives the LEDA specification of the abstract data type priority queue and Figure 2 gives an algorithm using it. There are many implementations of priority queues: Fibbonacci heaps, binary heaps, red-black trees, sorted or unsorted linear lists, ... Which implementation should be included in a library?

If one provides only one implementation, then this implementation should clearly be the "best" possible. This was the direction taken in versions 1 and 2 of LEDA. In the case of priority queues, we included Fibbonacci heaps because they are asymptotically as efficient as any other implementation. But, of course, only asymptotically. Also, there are better implementations for special cases, e.g., for integer keys or for shortest path computations on dense graphs. For other data types, e.g., range trees, there are implementations with vastly differing performance parameters (time-space tradeoff) and so, there is not even an asymptotically best implementation. All of this implies that providing only one implementation for each data type is not satisfactory.

So, one has to provide many and allow for the possibility of adding more. What properties should a mechanism for choosing between different implementations have ?

1. There should be a simple syntax for choosing between different implementations. In the next version of LEDA, the declaration
   PRIORITY_QUEUE <K,I,**rb_tree**> *PQ*
   creates an empty priority queue with key type K and information type I and selects red-black trees as the implementation variant. The declaration
   priority_queue<K,I> *PQ*
   selects the default implementation.

2. An application uses the abstract priority_queue data type. It gets an additional parameter of this type which allows to choose between different implementations. The application can be compiled without knowledge of the implementation and can later be used with priority queue arguments with different implementations. In LEDA 3.0., the definition of "procedure DIJKSTRA" reads:

   ```
   void DIJKSTRA(graph& G, node s,
                 edge_array<int>& cost,
                 node_array<int>& dist,
                 node_array<edge>& pred,
                 priority_queue<node,int>& PQ)
   ```

   { the body is as in Figure 2 with the declaration
   "priority_queue <node,int> *PQ*" removed }

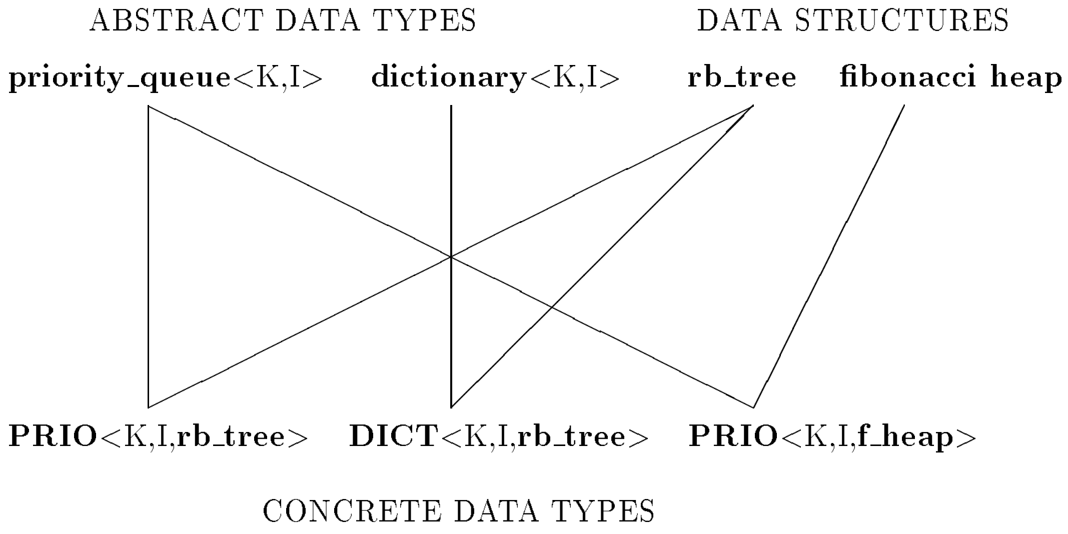

Figure 3: Multiple inheritance combines abstract data types and data structures to concrete data types

In the context of the declarations

priority_queue<node, int> FH_Q;
PRIORITY_QUEUE<node, int, rb_tree> RB_Q;
PRIORITY_QUEUE<node, int, my_impl> MY_Q;

the calls

DIJKSTRA(... , FH_Q);
DIJKSTRA(... , RB_Q);
DIJKSTRA(... , MY_Q);

are now possible.

The realization of the implementation parameter mechanism makes use of multiple inheritance, cf. Figure 3. Every concrete data type, say priority queues with the rb_tree implementation, is derived from its abstract data type and the data structure used to implement it. In the abstract data type class, all functions are virtual, i.e., have unspecified implementations. In the data structure class the details of the implementation are given and the classes in the bottom line of Figure 3 are used to match the abstract functions with the concrete implementations. We refer the reader to [15] for details.

5 Augmented Trees

This section deals with augmented trees. In the first part, we discuss requirements which a balanced tree scheme underlying an augmented tree must fulfill and in the second part we show that skip lists are a particularly elegant and efficient way to satisfy them.

5.1 Requirements

Augmented trees are abundant in computational geometry. Examples of augmented trees are range trees [3, 21], segment trees [4], interval trees [8, 7], and dynamization of order decomposable searching problems [17]. An account of these data structures can also be found in sections VII.2.2, VIII.5.1.3., VIII.5.1.1., and VII.1.3 of [11]. We will use 2-dimensional range trees as our running example.

Let $S \subseteq \mathbb{R}^2$ be a finite set of points in the plane and let $n = |S|$ be the cardinality of S. We use (x, y) to denote points in the plane and assume for the moment (this uniqueness assumption will later be dropped) that the x-coordinates of the points in S are pairwise distinct. A range tree for S consists of a primary tree T, which is simply a search tree for the x-coordinates of the points in T and a secondary tree T_v for each node v of T. The secondary tree T_v stores all points (x, y) in S in sorted order of y-coordinate such that v lies on the search path to x. A query is a box $[x_1, x_2] \times [y_1, y_2]$ and is supposed to return all points $(x, y) \in S$ such that $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. It can be answered as follows: Construct the set of nodes v of T such that the parent of v lies on either the search path to either x_1 or x_2 and such that v lies between the two paths and locate in T_v the set of points (x, y) such that $y_1 \leq y \leq y_2$. An insertion of a new point (x, y) is performed as follows: First insert x into the new primary tree and then insert (x, y) into the appropriate set of secondary trees. Deletions are processed analogously. Higher-dimensional versions of range trees also exist: In the d-dimensional version for storing points in d-dimensional space there is a d-fold hierarchy of trees. In principle, every (balanced) tree scheme can be used for primary and secondary trees. However, for the efficiency of augmented trees it is crucial that the primary tree scheme has the geometric size decrease and the strong rebalancing cost bound properties and supports weighted sets of data. We next discuss the three requirements.

Let $p_0, p_1, p_2, \ldots$ be a root to leaf path in the primary tree and let $size(p_i)$ be the number of leaves of the subtree rooted at p_i. The **geometric size decrease property** holds if there is a constant $a < 1$ such that $size(p_i) \leq a^i \cdot size(p_0) = a^i \cdot n$ for all $i \geq 0$. This property is useful in the following context: Operations (queries and updates) on augmented trees usually induce operations on the secondary structures along a path of the primary tree. If $C(m)$ bounds the cost of an operation on the secondary structure of size m then total cost is $C(n) + C(an) + C(a^2n) + \ldots$. For many functions C, this sums to $O(C(n))$. The geometric size decrease property holds for weight-balanced trees [16] [11, section III.5.1.] and their derivatives, and for randomized search trees [2]; it does not hold for any kind of height balanced tree scheme. (Consider for example a $(2, 3)$-tree of height $3h$ where a subtree T of height $2h$ consists solely for nodes of degree 3 and all nodes outside the subtree have degree 2. This tree has at most $2^{3h} + 3^{2h} \leq 2 \cdot 3^{2h}$ leaves of which more than 50% lie in the subtree T.)

Most kinds of balanced tree schemes are rebalanced by local changes, e.g., rotations or node splits, after an insertion or deletion. A local change in the primary tree T of an augmented tree scheme usually induces major changes in the secondary structures of the nodes involved in the local change. For example, a local change at a node v of the primary tree of a range tree requires to move approximately $size(v)$ elements between secondary trees. Let us therefore assume for simplicity that the cost of a local change at v is proportional to $size(v)$. A tree scheme is said to have a **strong bound for the rebalancing frequency** if the amortized rebalancing cost is logarithmic even if the cost of a local change is proportional to the size of the subtrees involved. Weight-balanced trees [21, 5] and randomized search trees [2] have strong bounds but no height-balanced scheme has. Also, at least for weight-balanced trees the known bounds involve large constant factors. For example, [21] states that every additional dimension in a range tree multiplies the bound for the update cost by $100 \log n$, i.e., the update cost in a d-dimensional tree is $(100 \log n)^{d-1}(c \log n)$, where $c \log n$ is the update cost in a 1-dimensional tree.

The previous two paragraphs made essential use of the uniqueness assumption, i.e., that no two points in S share an x-coordinate. Only this assumption implies that the size of the secondary tree T_v associated without node v is bounded by $size(v)$ and hence that a strong bound on rebalancing frequency implies a bound on the update cost of an augmented tree scheme. Consider for example a range tree for a set S, where $\frac{n}{2}$ points have the same x-coordinate, say x_0, and the other $\frac{n}{2}$ points have pairwise distinct x-coordinates. Then already the parent of the leaf x_0 has an auxiliary tree of size $\frac{n}{2}$. Also, insertion of a few points in the vicinity of x_0 will cause a local change at the parent of x_0 and this local change has cost $\Omega(n)$. Thus an amortized update cost $\Omega(n)$ results. The uniqueness assumption is a severe restriction for some applications, e.g., range-, segment-, and interval trees. (To be fair, we mention that frequently there is some transformation of the problem which introduces uniqueness; e.g., in the range searching problem we may use a primary tree for the pairs (x, y) ordered lexicographically, leave the secondary trees as described above, and replace a query box $[x_1, x_2] \times [y_1, y_2]$ by $[(x_1, -\infty), (x_2, +\infty)] \times [y_1, y_2]$. It is not clear however, whether such a transformation exists for all applications of augmented trees). Dropping the uniqueness assumption leads to weighted trees. In a weighted tree each leaf has a positive integer weight associated with it and the size of an internal node is the sum of the weights of its leaf descendants. (In the application to range trees the weight of leaf x is the number of distinct y with $(x, y) \in S$ and the insertion or deletion of a point (x, y) changes the weight of x by ± 1.) With this modified definition of size, the size of a node bounds the size of the secondary structure at the node and hence a strong bound on rebalancing frequency implies a bound on amortized update cost. D-trees [11, section III.6.2.] are a weighted tree scheme with a strong bound on the rebalancing cost and we have recently shown that a variant of pseudo-balanced trees [9] has strong bounds, too. For randomized search trees [2] the question is open and other weighted tree schemes, most notably splay trees [20] don't have it. However, D-trees a difficult to implement.

5.2 Skip Lists as a Basis for Augmented Trees

Skip lists [18] are a probabilistic alternative to balanced trees. In a skip list, each element is represented by a tower. A tower has a nonnegative height h and contains a key and $h+1$ **forward** pointers to towers. The i-th pointer in the tower points to the next tower of height i or more. Besides the towers representing the elements there are two extra towers called *header* and *stop*. They have equal height (called maxheight) and height larger than the height of any other tower. The height of the tower for an element is chosen randomly when the element is inserted. The height is h with probability $p(1-p)^h$ for all $h \geq 0$; here p is a parameter between 0 and 1 ($p = \frac{1}{2}$ and $p = \frac{3}{4}$ work well in practice). In [18, 19] it is shown that the search time in skip lists is logarithmic and that skip lists lead to very simple and efficient algorithms for other tree operations, e.g., insertions, deletions, finger searching, merging, joining, and splitting. We will now sketch that skip lists are also an alternative to weight-balanced and randomized search trees as a basis for augmented trees; the details can be found in [13].

For the sequel it will be convenient to associate a tree with a skip list. This tree can be obtained as follows:

1. Delete the tower *stop*.
2. Replace every tower v of height h by a linear list $v_h, v_{h-1}, \ldots, v_0$ of nodes and establish a so-called down link from v_i to v_{i-1} for all i with $h \geq i \geq 1$.
3. Create a forward link from v_i to w_i if the i-th forward pointer of tower v points to w, w is not the *stop* tower, and i is equal to the height of tower w.

For node v_i let $set(v_i)$ be the set of keys associated with nodes w_0 reachable from v_i and let $size(v_i)$ be the cardinality of $set(v_i)$. We have:

Lemma 1 *Let $h \geq 0$ and let v be a tower of height h or more in a skip list for n elements. Then $E[size(v_h)] \leq \min n, 1/(1-p)^{h+1}$.*

Proof: Let $t \geq 0$ and observe that $size(v_h) > t$ if each of the $t-1$ towers following v has height at most h. Thus,

$$prob(size(v_h) \geq t) \leq (\sum_{i=0}^{h} p(1-p)^i)^{t-1}$$

where the inequality reflects the fact that there may be less than $t-1$ towers following v and hence

$$E(size(v_h)) = \sum_{t \geq 0} prob(size(v_h) \geq t) \leq 1/(1-p)^{h+1}$$

The bound $size(v_h) \leq n$ is obvious. ∎

Lemma 1 directly implies that skip lists have the geometric size decrease property. It also implies a strong bound on amortized update cost.

Lemma 2 *If the cost of creating a tower of height h for a new element x is $O(\sum\limits_{0\leq i\leq h} size(x_i))$ then the expected cost of an insertion is $O(\log n)$.*

Proof: For $h \geq 0$ define random variables X_h and S_h as follows: X_h is 1 if the tower for x has height h ore more and 0 otherwise, and S_h is 1 plus the number of towers of height h or less following x. Note that variables X_h and S_h are independent and that $S_h = size(x_h)$. The cost of inserting x is $O(\sum_h S_h \cdot X_h)$ and therefore the expected cost of inserting x is

$$
\begin{aligned}
O(E[\textstyle\sum_{h\geq 0} S_h \cdot X_h]) &= O(\textstyle\sum_{h\geq 0} E[S_h] \cdot E[X_h]) \\
&= O(\textstyle\sum_{h\geq 0} \min(n, \frac{1}{(1-p)^{h+1}}) \cdot (1-p)^h \\
&= O(\log n)
\end{aligned}
$$

∎

We will next discuss how range trees (under the uniqueness assumption) can be based on skip lists. With every tower v of height h associate an array $aux[-1..h]$ of secondary structures. For every i, $-1 \leq i \leq h$, $aux[i]$ is a secondary structure for the points (x, y) with $x \in set(v_i)$. Here $set(v_i)$ is defined as above for $i \geq 0$ and $set(v_{-1}) = \{$the key associated with $v\}$. [13] gives complete programs for queries, insertion, and deletions. These programs are simpler than for the weight-balanced tree base. [13] also gives an analysis based on Lemma 1 and 2. The analysis shows that each additional dimension in a range tree contributes the multiplicative factor $7 \log n$ to the update cost. Recall that the factor is $100 \log n$ for range trees based on weight-balanced trees, i.e., skip lists not only lead to simpler but also to more efficient data structures for orthogonal range searching. Note however, that $100 \log n$ is likely to overestimate the multiplicative factor for weight-balanced trees but that $7 \log n$ is the exact factor for skip lists.

Can one drop the uniqueness assumption? Yes, one can and in fact very little additional machinery is needed. The only change required is a different rule to determine the height of a tower. The height of a tower for an element of weight w is the sum of its guaranteed height g and its random height h, where $2^{g-1} \leq w \leq 2^{g+2}$ and $prob(h = t) = 2^{-(t+1)}$ for $t \geq 0$.

Lemma 3 *Consider a skip list for a set of n weighted elements and let W be the total weight. Then the expected maximal height of a tower is $O(1) + \log W$. In particular, if $W = n^{O(1)}$ then the expected maximal height is $O(\log n)$.*

Lemma 1 holds also for weighted skip lists and hence weighted skip lists also have the geometric size decrease property.

A insertion/deletion or more generally a weight change by ± 1 is processed as follows. Change the weight w and if this violates the constraint $2^{g-1} \leq w \leq 2^{g+2}$ then set g to $\lfloor \log w \rfloor$ and construct a new tower of height $g+h$ where h is chosen randomly. Under the assumption of Lemma 2 this has cost $\sum_{0 \leq i \leq g+h} size(x_i)$ and therefore expected cost $O(2^g \log n)$. Amortizing this bound over the $\Omega(2^g)$ weight changes which did not require the creation of a new tower gives again an amortized expected update cost of $O(\log n)$, i.e., Lemma 2 stays true for weighted skip lists. [13] shows that this implies that skip lists can be used as an efficient means for orthogonal range searching even without the uniqueness assumption. In [13], we also show how other kinds of augmented tree schemes can be based on skip lists.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Company, 1983

[2] C. Aragon, R. Seidel, *Randomized Search Trees*, Proceedings of the 30th IEEE Symposium on Foundations of Computer Science , 1989, 540-545

[3] J.L. Bentley, *Decomposable Searching Problems*, Inf. Processing Letters, Vol.8, 1979, 244-251.

[4] J.L. Bentley, *Solutions to Klee's Rectangle Problems*, typescript, Dept. of Computer Science, Carnegie-Mellon University, 1977

[5] N. Blum, K. Mehlhorn, *On the Average Number of Rebalancing Operations in Weight-Balanced Trees*, Theoretical Computer Science 11, 1980, 303-320

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press/McGraw-Hill Book Company, 1990

[7] E.M. Mc Creight, *Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles*, Xerox Parc Report CSL-80-09, 1980

[8] H. Edelsbrunner, *Dynamic Rectangle Intersection Searching*, Report F47, Technical University Graz, Institut für Informationsverarbeitung, 1980

[9] J. Feigenbaum,R.E. Tarjan, *Two new kinds of biased search trees* The Bell System Technical Journal, Vol.62, No.16, Dec 83, p.3139-3158.

[10] E.N. Hanson, *The Interval Skip List: A data structure for finding all intervals that overlap a point*, 2nd WADS, LNCS Vol. 519, 1991, 153-164

[11] K. Mehlhorn, *Data Structures and Algorithms*
Vol. 1: Sorting and Searching
Vol. 2: Graph Algorithms and NP-Completeness
Vol. 3: Computational Geometry
Springer Publishing Company, 1984

[12] K. Mehlhorn, S. Näher, *LEDA, a Library of Efficient Data Types and Algorithms*, Communications of the ACM, to appear (preliminary Version in Proceedings of the 14th Symposium on Mathematical Foundations of Computer Science, LNCS Vol. 379, 88-106, 1989)

[13] K. Mehlhorn, S. Näher, R. Raman, *Skip Lists as a Basis for Augmented Trees*, in preparation

[14] S. Näher, *LEDA User Manual Version 2.0*, TR A17/90, Univ. des Saarlandes, Saarbrücken, 1990

[15] S. Näher, *Parameterized Data Types in LEDA*, in preparation

[16] I. Nievergelt, E.M. Reingold, *Binary Search Trees of Bounded Balance*, SIAM Journal of Computing, Vol. 2, 1973, 33-43

[17] M.H. Overmars, J. van Leeuwen, *Two General Methods for Dynamization of Decomposable Searching Problems*, Journal of Computing, Vol. 26, 1981, 155-166

[18] W. Pugh, *Skip Lists: A probabilistic Alternative to Balanced Trees*, Communications of the ACM, Vol. 33, No. 6, 1990, 668-676

[19] W. Pugh, *A Skip List Cookbook*, Technical Report, CS-TR-2286.1, University of Maryland, College Park,1990

[20] D.D. Sleator, R.E. Tarjan, *Self Adjusting Binary Search Trees*, Journal of the ACM, Vol. 32, No. 3, 1985, 652-686

[21] Willard, D.E., Lueker, G.S., *Adding Range Restriction Capability to Dynamic Data Structures* Journal of the ACM, Vol.32, No.3, 1985, 597-617.