table entry, dispatch time is not necessarily constant. It is easy to see that the total memory requirement is nk for the tables, plus $O(w)$ memory for conflict resolution.

2.2 Subtyping Algorithms

(B)PE: *(Bit) Packed Encoding* [Krall et al. 1997a] SC was specialized into a subtyping test scheme called Packed Encoding (PE), by Vitek, Horspool and Krall. They also suggested packing several identifiers into the same byte, resulting in an encoding called Bit Packed Encoding (BPE).

NHE: *Near Optimal Hierarchical Encoding* [Krall et al. 1997b] *Bit-vector encoding* embeds the hierarchy in the lattice of subsets of $\{1, \ldots, \beta\}$. In this scheme, each type a is encoded as a vector vec_a of β bits. Relation $a \preceq b$ holds iff

$$\mathrm{vec}_b \wedge \mathrm{vec}_a = \mathrm{vec}_b \,. \tag{6}$$

The challenge in building a bit-vector encoding is in finding the minimal β for which such an embedding is possible. The problem is NP-hard [Habib and Nourine 1994], but several good heuristics were proposed. Currently, NHE due to Krall, Vitek and Hoorspool, is the best (in terms of smallest β) algorithm for bit vector encoding.

Bommel and Beck [2000] describe an incremental technique for updating a bit-vector encoding. Although no asymptotic results are given, and testing was limited to "randomly generated hierarchies", it appears from the authors description that the technique is useful for small hierarchies, with at most 300 types.

(C)PQE: *(Coalesced) PQ-Encoding* [Zibin and Gil 2001] PQE encoding, which uses PQ-trees [Booth and Leuker 1976] gives one of the best compression results of the subtyping matrix, while maintaining constant time for queries. The CPQE encoding is a variant of PQE which with the cost of one additional dereferencing, reduces the space consumption of PQE even further. Both techniques are not incremental since they require feeding whole program information into very sophisticated data structures.

Dynamic subtyping in SI Dietz [1982; 1987] suggested an asymptotically optimal solution to the dynamic subtyping problem, i.e., linear space requirements and constant time for queries and additions. The idea is to maintain the pre- and post-orders of the tree in an *ordered list* (see Definition 3.3 below). Subtyping tests are answered by using two list ORDER queries relying on the fact that $a \preceq b$ iff a occurs before b in the post-order and b occurs before a in the pre-order.

A different incremental algorithm for SI is Cohen's algorithm [Cohen 1991]. Let $l_t = |\mathrm{ancestors}(t)|$ denote the *level* of t, i.e., the distance of t from the root of the tree hierarchy. The algorithm associates with each type t an array of length l_t, storing the type-id of each $t' \succeq t$ in position $l_{t'}$. Cohen's algorithm gives *simple* and constant-time subtyping tests. The cost is that the space requirement might be $O(n^2)$ if the hierarchy is, for instance, a long chain. In practice, since the maximal number of ancestors is relatively small, the space requirement of Cohen's encoding is tolerable. Jalapeño [Alpern et al. 2001], IBM implementation of the JAVA virtual machine (JVM), uses Cohen's algorithm for subtyping tests where the supertype is a class.

3. A TOOL BOX OF LIST ALGORITHMS

Our dispatching and subtyping techniques are based on a representation of a type hierarchy as one or more linked lists, with the property that the descendants of a type are consecutive. New elements are inserted to the lists as new types are added to the hierarchy. This section

develops the machinery for the various list maintenance operations which shall be used as subroutines in the main algorithms. We will discuss and make precise notions such as list locations, sublists, intervals, and even dictionaries over lists.

We rely on a vanilla doubly-linked list representation, where list *elements* are represented as pointers to their respective memory locations. To make algorithms more uniform, the representation includes two stub elements, one at the beginning and another at the end of the list. (This representation eliminates the need to deal with special cases such as empty list and one-element list.) Lists are initially empty, and grow by insertions; no deletions are allowed.

DEFINITION 3.1. *A* location *is (any point between) two consecutive list elements.*

Thus, a location is any point in the list into which a new element can be inserted. Fig. 5 shows a list of five elements. As shown in the figure, there are six locations $\ell_1, \ldots, \ell_6$ in this list.

A location can be represented as a pointer to the element preceding it.

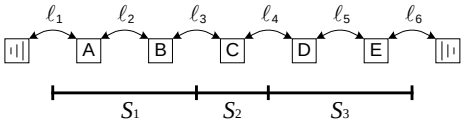

Fig. 5. A list, its locations, and a sublist partitioning

DEFINITION 3.2. *A* sublist *is a non-empty set of consecutive, non-stub list elements.*

The list in Fig. 5 is partitioned into three sublists, S_1, S_2 and S_3:

$$\begin{aligned} S_1 &= \{\mathsf{A}, \mathsf{B}\}, \\ S_2 &= \{\mathsf{C}\}, \\ S_3 &= \{\mathsf{D}, \mathsf{E}\}. \end{aligned} \tag{7}$$

A sublist can be represented as pointers to its first and last element.

3.1 Ordered lists

One of the most basic list operations is the order query, in which it is asked whether one given element precedes another.

Order queries play an important role in the encoding of a type hierarchy as a collection of lists. If the lists are such that the descendants of each type are consecutive, i.e., a sublist, then order queries with the end points of this sublist answer the question of whether one type is descendant of another.

An order query can be answered by a slow process of list traversal. The answer can be given in constant time if each element carried its ordinal number, but then insertions to the list become slow.

DEFINITION 3.3. *The* ordered list maintenance problem *is to maintain a list supporting two kinds of operations:* INSERT *transactions and* ORDER *queries of the following sort: "Given two elements in the list, determine which one precedes the other." An* ordered list *is a data structure supporting the above operations.*

This problem received some attention in the algorithmic literature [Dietz 1982; Tsakalidis 1984]. In a paper entitled "Two Algorithms for Maintaining Order in a List", Dietz and Sleator [1987] give the best algorithm for this problem, achieving $O(1)$ worst-case time per operation. However, the authors comment that their other algorithm "is probably the best algorithm to use in practice", even though it is theoretically inferior, since its amortized insertion time is $O(\log n)$ (comparisons are still $O(1)$ though, with much smaller hidden constants). Our empirical findings in implementing this algorithm concur with this observation.

This other algorithm is based on a technique known as *self-adjustment*. In a nutshell, each list element a is assigned a label $\mathrm{pos}(a)$ which is an integer in the range $[1, \ldots, N]$, $N \geq 4n^2$, where n is an upper bound on the number of elements in the list. An insertion of an element c between a and b is simple if $\mathrm{pos}(a) + 2 \leq \mathrm{pos}(b)$, in which case

$$\mathrm{pos}(c) \leftarrow \left\lfloor \frac{\mathrm{pos}(a) + \mathrm{pos}(b)}{2} \right\rfloor .$$

Otherwise, the algorithm re-adjusts the labels in a sublist containing a and b with a *sufficiently uneven distribution* of density of labels; special provisions are made to deal with the case that the list ends before such a sublist is found. (Specifically, the list is considered cyclic and the ordinal of each element a is obtained by a mod-N subtraction of the label of the first list element from $\mathrm{pos}(a)$.) Dietz and Sleator's use a *potential function* analysis method to show that with a suitable definition of a "sufficiently uneven distribution" condition, the amortized cost of redistribution is $O(\log n)$.

3.2 Dictionaries over lists

A *dictionary* over a list is a data structure which associates auxiliary data with all list elements. This data structure must (obviously) support Lookup queries which return the data associated with a given element. The data associated with an existing element never changes, but the dictionary should be updated whenever new elements are inserted to the list.

Many dictionaries can be kept over the same list. (Later we shall see that the dispatch table of each message is realized as a distinct dictionary, mapping receiver types to dispatching results on this message.) This is why the costs involved in the list representation are not counted as part of the space and time complexity measures of the dictionary.

Clearly, if the data stored in the dictionary is entirely unrelated to the list structure, then the most efficient dictionary implementation is that of standard data structures theory. We assume that the list can be partitioned into p (disjoint) sublists, such that the dictionary associates the same data with all elements in any sublist. (Such a partitioning where $p = 3$ was depicted in Fig. 5.)

A more compact dictionary representation using $O(p)$ space is achieved by (i) managing the base list as an ordered list, and (ii) storing in the dictionary only the first element in each sublist. The dictionary is implemented as a balanced binary search tree (BBST), where list elements are used as the comparison values in the tree nodes. A tree search is then conducted using constant time ORDER queries to find the sublist that contains any given element. Here and henceforth all lists are assumed to be ordered.

The following definition is pertinent.

DEFINITION 3.4. *The* interior *of a sublist is the set of all locations between any two of its elements. Its* boundary *includes the location prior to its first element, and the location*

following its last element.

When an element t is inserted into the list at location ℓ, we must also update all the dictionaries over the list. To do so, we must search for ℓ in each such dictionary. If ℓ falls within S, an existing sublist of the dictionary, then in the general case, S must be replaced in the BBST by three new sublists: elements of S prior to ℓ, the new element t, and the elements of S following ℓ. In the case that the data associated with t happens to be the same as that of S then the replacement is not carried out.

On the other hand, if ℓ is found to be on the boundary of sublists S_1 and S_2, then we must check first whether the data associated with t is the same as that of S_1 or S_2. If this is the case, then either S_1 or S_2 is appropriately extended. Otherwise, a new sublist with t is inserted into the BBST.

There are three additional special cases, in which either S_1, or S_2 (or both) do not exist. These occur when ℓ happens to be the first or the last location in the list, or both (i.e., when the list is empty). Handling these cases should be obvious.

LEMMA 3.5. *A dictionary over an ordered list can be implemented in $O(p)$ space and $O(\log p)$ time for all operations.*

3.3 Non-exhaustive dictionaries

Consider now a variant to the problem described above, in which a dictionary over a list is allowed to associate data with only *some* of the list elements; a Lookup query for all other elements returns null. This variant captures the dispatching problem more accurately since there could be types which do not recognize a certain message.

The modification of a BBST managing an exhaustive dictionary to deal with a non-exhaustive dictionary is not very difficult. Instead of storing just sublists heads in the BBST leaves, each such leaf will store both the first and the last element of its respective sublist. If a search for an element t ends in a leaf whose sublist is S, but $t \notin S$, then the result of this search is null.

Suppose that there are m dictionaries over a list. A transaction to insert an element t into the list must also detail a set of m' dictionaries which associate data with t. Typically, $m' \ll m$. We would like to refrain from investing time to update the $m - m'$ dictionaries which do not associate any data with t. Again, this objective does not seem difficult: the intact $m - m'$ BBSTs store *pointers* to list elements, rather than absolute positions, and therefore need not be updated.

There is however a catch in this argument. Even if no data is associated with t in a certain dictionary, its insertion may split a sublist of this dictionary. We are compelled to make the following assumption:

> *Elements without associated data cannot be inserted in the interior of an existing sublist of some dictionary.*

With this assumption, the insertion must inspect m' dictionaries instead of m. Luckily, the assumption holds in our specific application.

3.4 Intervals and their intersection

In our main algorithms we shall use a heuristic for searching for an appropriate location to insert a type into a list of types. The notions of intervals and their intersections will play an important role in this heuristic and in other occasions in these algorithms.

DEFINITION 3.6. *An* interval *in the list is a set of consecutive locations. The* boundary *of an interval $I = \{\ell_1, \ldots, \ell_s\}$, denoted $\mathcal{B}(I)$, comprises its first and last locations, i.e., $\mathcal{B}(I) = \{\ell_1, \ell_s\}$. All other locations in the interval, i.e., $I \setminus \mathcal{B}(I)$, are the* interior.

Fig. 6 depicts six intervals over the ordered list of Fig. 5. For example, interval $I_3 = \{\ell_3, \ell_4, \ell_5, \ell_6\}$, in Fig. 6, has two interior locations: ℓ_4 and ℓ_5; the two other locations are the boundary $\mathcal{B}(I) = \{\ell_3, \ell_6\}$.

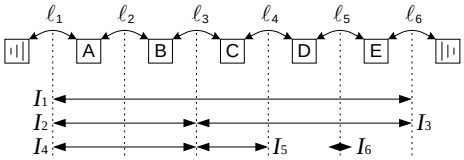

Fig. 6. An example of an ordered list and six intervals

A *proper* interval is an interval with two locations or more. For example, interval I_3 in Fig. 6 is proper.

A *degenerate* interval has only one location. The interior of such interval is empty, and the boundary is singleton. Interval I_6 in Fig. 6 is degenerate.

An *empty interval* has an empty boundary and an empty interior.

Note that there is a straightforward correspondence between intervals and sublists.

DEFINITION 3.7. *The interval of a sublist S includes the locations in the boundary and the interior of S. The sublist of an interval I includes all list elements between the first and last locations of I.*

Let S be a sublist, and let I be the interval of S. Then, not surprisingly, the sublist of I is S. The converse is almost true, except that the sublists of empty and degenerate intervals are undefined.

In Fig. 6, for example, the sublist of the interval I_3 is $\{\mathsf{C}, \mathsf{D}, \mathsf{E}\}$; the interval of the sublist $\{\mathsf{A}, \mathsf{B}\}$ is I_4.

DEFINITION 3.8. *Given a set of intervals $\mathcal{I}$, their* intersection, *denoted $\Lambda(\mathcal{I})$, is the set of locations which belong to all intervals in $\mathcal{I}$.*

Observe that $\Lambda(\mathcal{I})$ is also an interval, and it might be degenerate, and even empty.

Algorithmically, $\Lambda(\mathcal{I})$ is computed by finding the largest first location of the intervals in $\mathcal{I}$, and the smallest last location of these intervals in $O(|\mathcal{I}|)$ time. Comparisons are carried out using constant time ORDER queries.

In Fig. 6, for example,

$$\Lambda(\{I_1, I_3, I_5\}) = \{\ell_3, \ell_4\}.$$

Also, $\Lambda(\{I_1, I_2, I_3, I_4, I_5\})$ is the degenerate interval $\{\ell_3\}$, and $\Lambda(\{I_1, I_2, I_3, I_4, I_5, I_6\})$, i.e., the intersection of the set of all intervals in the figure, is the empty interval $\emptyset$.

3.5 Segment partitioning

Let $\mathcal{I}$ be a set of non-empty intervals. In the above we were interested in the intersection of *all* members of $\mathcal{I}$. In considering intersections of subsets of $\mathcal{I}$, we find that $\mathcal{I}$ imposes

a partitioning of the list elements into segments. More precisely, given a list element t, let $\mathcal{I}(t)$ denote the set of intervals which include t, i.e.,

$$\mathcal{I}(t) = \{I \in \mathcal{I} \mid t \text{ is contained in the sublist of } I\}.$$

In Fig. 6, for instance, we have that $\mathcal{I}(\mathsf{A}) = \{I_1, I_2, I_4\}$.

DEFINITION 3.9. *Given a set of intervals $\mathcal{I}$, a* segment *is a maximal sublist S such that $\mathcal{I}(t)$ is the same for all $t \in S$. The* segment partitioning *of $\mathcal{I}$ is the set of all such segments.*

The segment partitioning of the intervals in Fig. 6 is nothing else than the sublist partitioning depicted in Fig. 5. The set $\{\mathsf{A}, \mathsf{B}\}$, for instance, is a segment since all of its elements t have an equal $\mathcal{I}(t) = \{I_1, I_2, I_4\}$, and for no other adjacent element t' we have $\mathcal{I}(t') = \mathcal{I}(t)$.

DEFINITION 3.10. *Let $\mathcal{I}$ be a (multi-) set of intervals. Then, $\mathcal{L}_\mathcal{I}$, the set of all boundary locations of intervals in $\mathcal{I}$ is defined by*

$$\mathcal{L}_\mathcal{I} = \bigcup_{I \in \mathcal{I}} \mathcal{B}(I).$$

Note that multiple occurrences of intervals are allowed in $\mathcal{I}$. However, locations cannot occur more than once in the set $\mathcal{L}_\mathcal{I}$.

The following lemma bounds the size of a segment partitioning of a set of intervals.

LEMMA 3.11. *A set of intervals $\mathcal{I}$ partitions the list into at most $2|\mathcal{I}| + 1$ segments.*

PROOF. Let us traverse the list while considering changes to the set $\mathcal{I}(t)$, where t is the current element. Let t and t' be two consecutive list elements, and let ℓ be the location between them. Then, $\mathcal{I}(t) \neq \mathcal{I}(t')$ only when ℓ is a boundary of some interval in $\mathcal{I}$, i.e., $\ell \in \mathcal{L}_\mathcal{I}$. Since $|\mathcal{L}_\mathcal{I}| \leq 2|\mathcal{I}|$, the number of segments is at most $2|\mathcal{I}| + 1$. □

3.6 Partitioning dictionaries

As mentioned above, a dictionary over a list is used to represent the dispatching table of a single message. Our implementation of such a dictionary (Section 3.2) relied on the fact that the list can be partitioned into p sublists, where the dispatching result is the same in each sublist.

We now turn to describing an alternative implementation of the dispatching table of a single message. This implementation relies on the fact that in the case of a dispatching table, the partitioning into segments is imposed by a collection of intervals. On certain inputs, the new implementation is superior to plain dictionaries over a list.

A *partitioning dictionary* maintains the segment partitioning of a multi-set of *proper* intervals $\mathcal{I}$ over a list. Lookup queries must return the segment which contains a given list element. Updates to the dictionary are either in an introduction of a new (proper) interval, or in an insertion of a new element into the list, with appropriate extension of neighboring intervals.

What happens to a segment partitioning when a new element is inserted into the list (specifically at location ℓ)? Note that after the insertion location ℓ ceases to exist; the insertion splits ℓ into two: location ℓ' before the inserted element and location ℓ'' after it.

Consider an interval I. If $\ell \in I \setminus \mathcal{B}(I)$, i.e., ℓ is in the *interior* of I, then surely after the insertion $\ell', \ell'' \in I$. Similarly, if $\ell \notin I$, then surely after the insertion $\ell', \ell'' \notin I$. However,

if ℓ is on the boundary of I ($\ell \in \mathcal{B}(I)$), then I may or may not be extended to contain the new element. In other words, after the insertion we are certain that one of ℓ' and ℓ'' will belong to I. Whether both locations belong to I must be specified by the Update transaction.

Let $\mathrm{neighbors}_{\mathcal{I}}(\ell) \subseteq \mathcal{I}$ be the multi-set of those intervals in $\mathcal{I}$ such that ℓ is on their boundary, i.e.,

$$\mathrm{neighbors}_{\mathcal{I}}(\ell) = \{I \in \mathcal{I} \mid \ell \in \mathcal{B}(I)\}.$$

(The subscript $\mathcal{I}$ is omitted whenever it is obvious from context.)

Consider again the list depicted in Fig. 6. There are in total four intervals whose boundary contain ℓ_3:

$$\mathrm{neighbors}(\ell_3) = \{I_2, I_3, I_4, I_5\}.$$

An insertion into location ℓ must also specify a set $N \subseteq \mathrm{neighbors}(\ell)$, such that every interval in N will contain the element inserted at ℓ. These intervals must be appropriately extended by the transaction.

As we shall see, we need to make the distinction between neighbors on the left and on the right of a location. For a location ℓ, let

$$\mathrm{left}(\ell) \subseteq \mathrm{neighbors}(\ell)$$

be the multi-set of intervals on the left of ℓ, i.e., those intervals whose last location is ℓ. Also, let

$$\mathrm{right}(\ell) = \mathrm{neighbors}(\ell) \setminus \mathrm{left}(\ell),$$

be the multi-set of intervals on the right of ℓ. In Fig. 6, for example, we have

$$\begin{aligned} \mathrm{left}(\ell_3) &= \{I_2, I_4\} \\ \mathrm{right}(\ell_3) &= \{I_3, I_5\} \end{aligned} \tag{8}$$

Suppose that an element is inserted at location ℓ_3. How should the intervals $\{I_1, \ldots, I_6\}$ and their segment partitioning be changed? Fig. 7 demonstrates the possible effects. As can be seen in the top of the figure, the insertion replaces ℓ_3 by ℓ_3' and ℓ_3''.

Fig. 7a shows the case that none of the neighboring intervals (I_2, I_3, I_4 and I_5) is extended. In this case, the insertion results in the creation of a new segment consisting of N. In Fig. 7b interval I_2 is extended, again resulting in the creation of the same new segment. As shown in Fig. 7c, this segment is also created if both interval I_2 (on the left of the insertion point) and I_5 (on the its right) are extended.

No segment is created if all the intervals on the left of insertion point (I_2 and I_4) are extended, while no intervals on the right are extended. In this case, shown in Fig. 7d, an existing segment $\{\mathsf{A}, \mathsf{B}\}$ is extended to include N. Fig. 7e shows the symmetrical case in which intervals I_3 and I_5 are extended. Again, no new segment is created, and segment $\{\mathsf{C}\}$ is extended to include N.

Formally, a *partitioning dictionary* maintains a multi-set of proper intervals $\mathcal{I}$ while supporting the following operations:

(1) Lookup(t) *where t is a non-stub list element.* The query returns the segment (as defined by the segment partitioning of $\mathcal{I}$) which contains t. This segment is returned as a unique segment identifier. Segment identifiers are assigned in order as new segments are created.

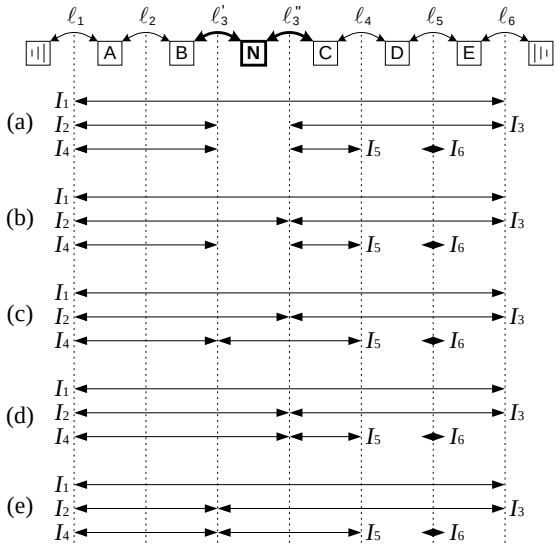

Fig. 7. The list of Fig. 6 after inserting an element N at location ℓ_3 with various settings of N, the list of neighboring intervals to extend: (a) $N = \emptyset$, (b) $N = \{I_2\}$, (c) $N = \{I_2, I_5\}$, (d) $N = \{I_2, I_4\}$, and (e) $N = \{I_3, I_5\}$,

(2) New(t) *where* t *is a non-stub list element.* Let I be the smallest interval containing t, i.e., $I = \{\ell', \ell''\}$, where location ℓ' precedes t and location ℓ'' follows it. Then, the transaction adds I to $\mathcal{I}$, and returns a new identifier for this interval.

(3) Update(ℓ, N) *where* ℓ *is a list location and* $N \subseteq \mathrm{neighbors}(\ell)$ *is a list of intervals identifiers (as returned by* New *transactions).* The transaction updates the dictionary when a new element is inserted into the list at location ℓ. The intervals in N, which are adjacent to ℓ, must be appropriately extended to include the new element.

As indicated by the proof of Lemma 3.11, the segment partitioning of $\mathcal{I}$ is uniquely defined by the set $\mathcal{L}_\mathcal{I}$. Our dictionary implementation is based on a representation of the set $\mathcal{L}_\mathcal{I}$ augmented with the first and last locations in the list. The records for the location in this augmented set are stored as leaves of a BBST, which supports order-based queries in logarithmic time. In addition, a hash table of pointers to these records makes it possible to access locations in constant time.

The data structure also includes a record for each interval in $\mathcal{I}$. Each interval record includes appropriate references to the records of the first and last locations of the interval.

The record of a location ℓ stores $\mathrm{left}(\ell)$ as a linked list of pointers to the records of the appropriate intervals. This linked list makes it possible to efficiently examine all intervals ending at ℓ. It is sometimes necessary to change the end point of an interval. To this end, we use back pointers from interval records into the linked list $\mathrm{left}(\ell)$. The record of an interval I, $I \in \mathrm{left}(\ell)$, includes a pointer $\mathrm{first}(I)$ to this node in the linked list $\mathrm{left}(\ell)$ which has a pointer to the record of I. It therefore requires only constant time to remove I from the list $\mathrm{left}(\ell)$ and place it in another list $\mathrm{left}(\ell')$.

There is also a similar list $\mathrm{right}(\ell)$ in the record of each location ℓ. The record of each interval I has also a similar back pointer $\mathrm{last}(I)$. Let ℓ be such that $I \in \mathrm{right}(\ell)$.

Then, $\mathrm{last}(I)$ is a pointer to the node in the linked list $\mathrm{right}(\ell)$ which stores the address of I.

Our implementation therefore supports changes to each of the boundary points of an interval in constant time.

A Lookup(t) query is answered by searching the BBST for the nearest locations on the right and on the left of t. These two locations define the segment of t.

Algorithm 1 describes the implementation of a New(t) transaction. The algorithm creates a record for the interval $\{\ell', \ell''\}$ and inserts these two locations into the BBST and hash-table if they did not appear there already. Note that checking for existence is done in constant time, while the time required for inserting locations is logarithmic. The updates of the lists $\mathrm{right}(\ell')$ and $\mathrm{left}(\ell'')$, and the setting of the pointers $\mathrm{first}(I)$ and $\mathrm{last}(I)$ are all done in constant time.

Algorithm 1 New(t), t is a non-stub list element

```
1: Let ℓ′ be the location prior to t
2: If ℓ′ is not found in the hash table then
3:     Create a new record for ℓ′
4:     Insert this record to the hash table and to the BBST
5: Let ℓ″ be the location following t
6: If ℓ″ is not found in the hash table then
7:     Create a new record for ℓ″
8:     Insert this record to the hash table and to the BBST
9: Let I be a new interval record
   // I will be the smallest interval containing t, i.e., {ℓ′, ℓ″}
10: Push the address of I to the linked list right(ℓ′)
11: Let first(I) be the top of the linked list right(ℓ′)
12: Push the address of I to the linked list left(ℓ″)
13: Let last(I) be the top of the linked list left(ℓ″)
14: return I
```

The implementation of an Update(ℓ, N) transaction is slightly more demanding since we need to examine the effect on $\mathcal{L}_{\mathcal{I}}$ of the split of ℓ into ℓ' and ℓ''.

If $\ell \notin \mathcal{L}_{\mathcal{I}}$, i.e., ℓ is not on the boundary of any interval in $\mathcal{I}$, then $N \subseteq \mathrm{neighbors}(\ell)$ is necessarily empty. After the insertion $\ell', \ell'' \notin \mathcal{L}_{\mathcal{I}}$, and the partitioning dictionary is unchanged. For instance, in our running example (Fig. 6), if an element is inserted at location ℓ_2, then the segment partitioning is (essentially) unchanged, since no new segment is created. Intervals I_1, I_2 and I_4 must of course be appropriately "stretched", and so must be the segment $\{\mathsf{A}, \mathsf{B}\}$, but our end-points based representation of segments means that no modifications to the data-structure are required.

Fig. 7 demonstrates the other, more difficult, case to consider, in which an element is inserted at a location which is on the boundary of one or more of the intervals in $\mathcal{I}$, i.e., $\ell \in \mathcal{L}_{\mathcal{I}}$. After the insertion then, either ℓ' or ℓ'' (or both) are in $\mathcal{L}_{\mathcal{I}}$. Fig. 7e is an example of the case $\ell' \in \mathcal{L}_{\mathcal{I}}$. The case $\ell'' \in \mathcal{L}_{\mathcal{I}}$ is shown in Fig. 7d. Fig. 7a, Fig. 7b, and Fig. 7c, are examples of a creation of a new singleton segment between ℓ' and ℓ'', i.e., both $\ell', \ell'' \in \mathcal{L}_{\mathcal{I}}$.

Algorithm 2 presents the implementation of an Update transaction. Lines 1–2 deal with the case that the location of the insertion is in the interior of one of the segments in the partitioning. No changes to the data structure are necessary in this case.

Algorithm 2 Update(ℓ, N), ℓ is a list location and $N \subseteq \mathrm{neighbors}(\ell)$ is a list of interval records.

```
1:  If ℓ is not found in the hash table then // ℓ ∉ L_I, hence N = ∅
2:      return
    // Create the two new locations
3:  Let ℓ' be a new location record; left(ℓ') ← left(ℓ); right(ℓ') ← ∅
4:  Let ℓ'' be a new location record; left(ℓ'') ← ∅; right(ℓ'') ← right(ℓ)
    // Extend neighboring intervals as specified by N
5:  For all I ∈ N do
6:      If I ended at ℓ then
7:          Transfer I from left(ℓ') to left(ℓ'')
8:          Let the last location of I be ℓ''
9:      else // I started at ℓ
10:         Transfer I from right(ℓ'') to right(ℓ')
11:         Let the first location of I be ℓ'
    // Remove ℓ from the BBST and the hash table; insert instead ℓ' or ℓ'' or both
12: Remove ℓ from the hash table
13: If left(ℓ') = ∅ and right(ℓ') = ∅ then // Case ℓ' ∉ L_I
14:     Replace ℓ by ℓ'' as a leaf of the BBST
15:     Insert ℓ'' into the hash table
16: else if left(ℓ'') = ∅ and right(ℓ'') = ∅ then // Case ℓ'' ∉ L_I
17:     Replace ℓ by ℓ' as a leaf of the BBST
18:     Insert ℓ' into the hash table
19: else // Case ℓ', ℓ'' ∈ L_I
20:     Remove ℓ from the BBST
21:     Insert ℓ' and ℓ'' into the BBST
22:     Insert ℓ' and ℓ'' into the hash table
```

Otherwise, we create records for the two new locations ℓ' and ℓ'' (lines 3–4), and split the intervals neighboring on ℓ between them: All intervals on the left of ℓ are assigned to the left of ℓ'; all intervals on the right of ℓ are assigned to the right of ℓ''. Note that the linked list representation of the left and right interval neighbors makes it possible to carry out the statements $\mathrm{left}(\ell') \leftarrow \mathrm{left}(\ell)$ and $\mathrm{right}(\ell'') \leftarrow \mathrm{right}(\ell)$ in constant time.

The next block of commands (lines 5–11) corrects this initial split of intervals as per the specification N of intervals to extend. If an interval in N was on the left of ℓ, then its extension to include the newly inserted element places it on the list $\mathrm{left}(\ell'')$ rather than $\mathrm{left}(\ell')$. Similarly, if the interval was on the right of ℓ, then the extension places it on the list $\mathrm{right}(\ell')$ rather than $\mathrm{right}(\ell'')$.

At the end of the block, the records of ℓ' and ℓ'' are ready to be inserted to the hash-table and to the BBST. At first we are certain to remove ℓ from the hash table (line 12). Updates to the hash table are constant time. However, since insertions and deletions from the BBST

are logarithmic time, we wish to avoid such operations if possible. Specifically, if the split is such that $\ell' \notin \mathcal{L}_\mathcal{I}$ or $\ell'' \notin \mathcal{L}_\mathcal{I}$, then the update of the BBST can be done in constant time.

Lines 13–15 deal with the case, shown in Fig. 7d, in which $\ell' \notin \mathcal{L}_\mathcal{I}$. If there are no intervals on the left or on the right of location ℓ', then it does not take place in the segment partitioning. No changes to the structure of the BBST are made in this case. All that is required to do is to replace the record in the leaf of ℓ with the record of ℓ''. We also need to insert ℓ'' into the hash table.

Lines 16–18 present the symmetrical case, shown in Fig. 7e, in which $\ell'' \notin \mathcal{L}_\mathcal{I}$. Note that if $\ell \in \mathcal{L}_\mathcal{I}$ then either $\ell' \in \mathcal{L}_\mathcal{I}$ or $\ell'' \in \mathcal{L}_\mathcal{I}$. In other words, it could not be that all of the lists $\mathrm{left}(\ell')$, $\mathrm{right}(\ell')$, $\mathrm{left}(\ell'')$, and $\mathrm{right}(\ell'')$ are empty (since at least one of $\mathrm{left}(\ell)$ or $\mathrm{right}(\ell)$ was not empty). Therefore, the cases dealt with in lines 13–15 and 16–18 are mutually exclusive.

There an important but subtle point to notice. It is legitimate to replace a leaf ℓ with a leaf ℓ' (line 14), or with a leaf ℓ'' (line 17). The BBST remains consistent since the relative order in the master list of ℓ is the same as ℓ' and ℓ''. However, if ℓ is stored in an internal node of the tree, then the BBST will be invalidated by the algorithm, since ℓ ceases to exist. To avoid this predicament, the locations in internal nodes of the BBST are represented as pointer to an appropriate leaf.

The general case of an insertion to the master list is handled by lines 19–22. In this case, ℓ must be replaced by both ℓ' and ℓ''. To do so the code removes the leaf ℓ from the BBST, and then inserts the two new leaves ℓ' and ℓ'' in logarithmic time.

Let $p = |\mathcal{I}|$; let q be the number of Update transaction; let N_i, $i = 1, \ldots, q$ be the multi-set of intervals which were extended when the i^{th} element was inserted.

LEMMA 3.12. *A partitioning dictionary over an ordered list can be implemented in $O(p)$ space, $O(\log p)$ time for a* Lookup *query, $O(\log p)$ time for a* New *transaction, and*

$$O(p \log p + \sum_{i=1}^{q} |N_i|)$$

time for all Update *transactions.*

Proof Lookups are implemented in logarithmic time as searches in the BBST. Similarly, New transactions are logarithmic time since they require to insert new leaves to the BBST.

The time complexity of Algorithm 2 is $O(|N_i|)$ for the loop in line 5. In addition, lines 20–21 execute in $O(\log p)$ time. However, these lines are executed only when a new segment is created. Lemma 3.11 guarantees that the number of such segments is linear in p. □

The simple implementation of a dictionary over a list (Lemma 3.5) gives a time complexity of $O(q \log p)$ for a series of q updates. Partitioning dictionaries are superior to these if the average size of N_i is asymptotically smaller than $\log p$.

4. INCREMENTAL ALGORITHMS FOR SINGLE INHERITANCE HIERARCHIES

This section describes incremental algorithms for the subtyping and dispatching problems in an SI hierarchy. For simplicity, we assume, without loss of generality, that the hierarchy is a tree rather than a forest.