Improved Symmetric Lists

Technical Report MIP-0409
October, 2004

Christian Bachmaier and Marcus Raitner

University of Passau,
94030 Passau, Germany
Fax: +49 851 509 3032
{bachmaier,raitner}@fmi.uni-passau.de

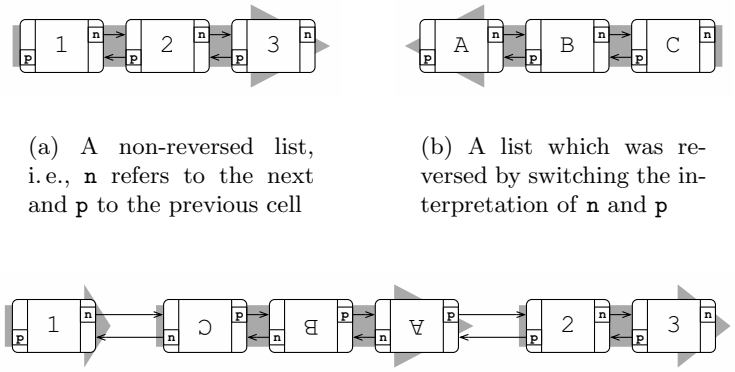
Abstract. We introduce a new data structure called *symlist* based on an idea of Tarjan [17]. A symlist is a doubly linked list without any directional information encoded into its cells. In a symlist the two pointers in each cell have no fixed meaning like previous or next in standard lists. Besides the common operations on doubly linked lists, symlists support the reversal of a list and the insertion of a (reversed) symlist into another one, both in constant time. This is an improvement over common implementations, e. g., the STL class `list`, where reversal needs linear time. A typical application of symlists is storing the children of a so called Q-node in a PQ-tree, a data structure used in linear time graph planarity testing algorithms. We show that a straightforward implementation of Tarjan's idea [17] leads to an anomaly when inserting a new element and provide a simple solution to avoid it. Finally, we present an implementation of symlists with iterators that is similar to the STL class `list` and its iterators.

1 Introduction

A doubly linked list consists of *cells* each containing a data field storing an *element* of the list and two pointers to its neighbours. In general, the interpretation of these pointers is hard-coded [1, 5, 10, 13]. Lists usually rely on the interpretation that the pointer named `prev` refers to the previous cell and the one named `next` to the next cell. Clearly, reversing such a list can either be done in linear time by interchanging the pointers in *all* cells or in constant time by globally reversing their *interpretation*. However, then the pointers are interpreted identically for *all* cells. This prevents inserting a reversed list into another in constant time, since then the pointers of the cells of the inserted list and of the original list have different meanings.

Example 1. Consider the two lists in Fig. 1(a) and Fig. 1(b). As the arrows in the background indicate, the list in Fig. 1(b) has been reversed, i. e., the pointers in cells `1`, `2`, and `3` on the one hand and cells `A`, `B`, and `C` on the other are interpreted

differently. If the list in Fig. 1(b) is inserted in the list in Fig 1(a) between cells 1 and 2 (in constant time), we obtain Fig. 1(c). Obviously, the adjacency pointers do not have the same meaning in all cells in this list. Of course, this can be fixed by interchanging the pointers in each cell of the inserted list, but this takes linear time.



(a) A non-reversed list, i. e., `n` refers to the next and `p` to the previous cell

(b) A list which was reversed by switching the interpretation of `n` and `p`

(c) The result of inserting the list in 1(b) into the one in 1(a). Not all pointers in this list have the same meaning, e. g., `n` in cell `1` refers to the next cell, but `n` in cell `B` refers to the previous one

Fig. 1. Example for the problem of inserting a reversed list into a non-reversed one in constant time with fixed interpretation for the pointers in the cells. In each cell `n` and `p` denote pointer `next` and `prev` respectively

1.1 The Concept

Following an idea of Tarjan [17], our data structure symlist solves this problem by ignoring the common direction information of the pointers in a cell. Symlists only preserve the invariant that the two pointers refer to the two adjacent cells without specifying directly or indirectly which one is the previous and which is the next. Although this is not common in data structures (cf. [1, 5, 10, 13]), it makes sense from a software engineering perspective because the directional information is no feature of the cell. It is needed only for traversing a list.

Clearly, a traversal is more complicated for a symlist than for an ordinary doubly linked list. In our implementation, we use iterators that store the directional information along with the cell they refer to. Whenever the iterator is moved in either direction both its position and its direction must be updated (see Sect. 2 for a detailed description). Although the basic idea of symlists is

already described in [17], an anomaly which a straight-forward implementation of this idea will have (cf. Sect. 3.1) is not mentioned.

A symlist is related to the *quad-edge* data structure [7] that is used for efficiently representing embeddings of graphs in two-dimensional manifolds. It consists of quad-edges and each node contains four references to adjacent quad-edges. For iterating through this data structure, two flags are stored along with the current quad-edge. These are used to determine the next quad-edge in the iteration. Depending on the value of these flags, either the graph, its dual, or its mirror image are traversed. If we use two of the four pointers of a quad-edge and one of its flags only, iterating through this data structure is similar to the iteration through a symlist. However, the quad-edge data structure was designed for that particular purpose only and is not meant to be generic. It is used for iterating through the represented graph and does not support reversals and insertions.

Sleator and Tarjan [16] describe the efficient management of *dynamic paths* via *biased trees*. There every internal node v of the tree has a flag reversed(v). The *reversal state* of v is an exclusive or of the *reversed flags* from v to the root. This indicates whether the path segment descendant to v is reversed or not and can be computed within logarithmic time. We touch on this work because a list can be seen as a path containing data information in its nodes. Of course, an insertion of a new element in the path, an operation not mentioned in [16], has to update this tree structure which is not possible in constant time.

1.2 Applications

Symlists can be used for any doubly linked list; in particular, when in addition to the standard operations, both reversing the list and inserting a list into another has to be done in constant time.

Within the *PQ-tree* data structure [2], we have these requirements for the lists storing the children of a tree node. A PQ-tree is used for the efficient storage and manipulation of permutations of a set S. PQ-trees are rooted trees with leaves and two types of inner nodes, namely P- and Q-nodes. The leaves correspond to the elements of S, and the possible permutations are encoded by the combination of the two types of inner nodes. The children of a P-node can be permuted arbitrarily, whereas the children of a Q-node are ordered, and this order can only be reversed.

The most important operation on PQ-trees is `reduce`. It restricts the encoded set of permutations such that all elements of a given set $S' \subseteq S$ are consecutive in all remaining permutations. This operation may reverse the order of the children of some Q-nodes and insert the children of a Q-node somewhere in between the children of another Q-node. In order to achieve the complexity of `reduce` proved in [2], both reversing a list of children and inserting a list of children into another must be done in constant time. This complexity of `reduce` is crucial for the linear time graph planarity testing and embedding algorithm [4].

In some implementations of PQ-trees no separate list data structure is used for storing the children of a Q-node. The references to the adjacent siblings are

stored in each child [2, 11, 12]. In these implementations the adjacency pointers are treated as an unordered set. Hence, they have no fixed meaning either and reversing and inserting a reversed list can be done in constant time. However, from a software engineering perspective, it is better to have a separate and reusable data structure with a well-defined interface that provides appropriate methods for accessing and manipulating the children of a Q-node.

There is another linear time planarity test of Boyer and Myrvold [3] which does not use PQ-trees but also needs an efficient data structure for flipping biconnected components, see [3, Section 2]. They call this data structure *doubly linked cycle with no sense of clockwise*. This is exactly the paradigm of symlists.

Table 1. Typical operations on lists and iterators and their time complexity

	Operation	symlist	ordinary list
List	`insert`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`splice`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`erase`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`reverse`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
	`size`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	`empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Iterator	`++`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`--`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Common implementations of doubly linked lists encode the direction of the list into their cell by using pointers with a uniform meaning for the whole list [1, 5, 10, 13]. As illustrated by Example 1, these implementations cannot provide constant time methods for both reversing a list and inserting a reversed list into another. Hence, they cannot be used in a PQ-tree within the desired time bounds. Our data structure symlist fills this gap. As shown in Table 1, the time complexity of all methods on lists are identical except for reversing a list that can be done in constant time on a symlist.

2 Implementation

Our implementation is based on the interface of the STL class `list` [14] containing methods like `empty`, `insert`, `splice`, `erase`, and `reverse`, see Fig. 2. The interface is enriched with some new methods `attach_sublist`, `detach_sublist`, `blind_insert`, and `blind_erase` (see Sect. 3.2).

A symlist consists of cells containing two adjacency pointers, `p[0]` and `p[1]`. As suggested in [10] and used in STL, our implementation is a cyclic list with an additional cell between the first and the last cell, see Fig. 3. It is called the *end cell* because this is the cell where the end iterator, `symlist::end()`, refers to.

Fig. 2. The interface of our implementation in UML notation

The end cell is stored in our list class along with a flag `dir` $\in \{0, 1\}$ indicating which of the two pointers refers to the first cell of the list. In addition to its cell object `cell`, each iterator stores its current direction as a flag `dir` $\in \{0, 1\}$ indicating that `p[dir]` in `cell` leads to the next cell.

Algorithm 1 explains the effect of the `++`-operator applied to an arbitrary iterator `it` in detail (the `--`-operator operates analogously).

Algorithm 1: Advancing to the next cell in iterator's direction

```
operator++()
begin
    tmp ⟵ it.cell                    // make a copy of the current cell
    it.cell ⟵ tmp.p[it.dir]          // move to the next cell
    if it.cell.p[0] = tmp then
        it.dir ⟵ 1                   // p[0] points back, hence new dir is 1
    else
        it.dir ⟵ 0                   // p[1] points back, hence new dir is 0
    end
end
```

Insertion of a new cell `z` before an arbitrary iterator `it` referring to cell `x` works as follows: With `--it` we determine the previous cell `y` of `x` in the sense of `it`'s local direction. After updating the appropriate pointers of `x`, `y`, and `z`, a new iterator pointing to `z` and having the same direction as `it` is returned.

For an iteration we need begin and end iterators. As already mentioned, the end iterator points to the end cell, and its direction flag is set to the direction flag of the list. In order to get a begin iterator, i. e., `symlist::begin()`, we internally call `++` on the end iterator, which gives an iterator pointing to the first element in the list and heading in the direction of the list.

Now reversing a symlist can be done in constant time by flipping the direction flag of the list. This avoids the problems illustrated in Example 1 and avoids using linear time for reversing. Furthermore, an iterator can easily be reversed by flipping its direction flag. As in STL, all existing iterators, except the ones referring to erased cells, stay valid and consistent to their possibly new adjacency throughout the lifetime of a symlist and all update operations on it. This also holds if the blind operations from Sect. 3.2 are used.

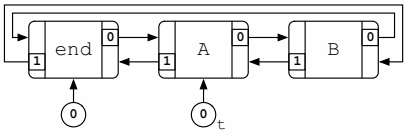
3 Extensions

3.1 Losing Information

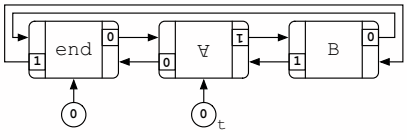
In the basic implementation as outlined above, the information stored in an iterator together with the direction flag of the list may be insufficient. This is clarified by the following example.

Example 2. Consider the symlists in Fig. 3(a) and Fig. 3(b). These are almost identical. The only difference is that `p[0]` and `p[1]` in cell `A` are switched. Therefore the iterator `t` in Fig. 3(a) has the same direction as the end iterator, whereas `t` in Fig. 3(b) has the opposite direction. Hence, inserting a new cell before `t` in Fig. 3(a) results in a new cell between `end` and `A`. On the other hand, in Fig. 3(b) this insertion results in a new cell between `A` and `B`.

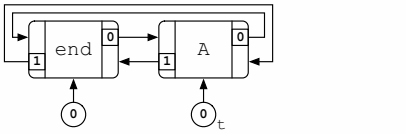
If we delete cell `B` in both Fig. 3(a) and 3(b), we get the configurations shown in Fig. 3(c) and Fig. 3(d), respectively. Clearly, inserting a new cell before `t` in Fig. 3(c) should give a list beginning with the new cell, and inserting it in Fig. 3(d) should give a list with the new cell at the end. But the two configurations in Fig. 3(c) and Fig. 3(d) are indistinguishable, i. e., both pointers in each cell refer to the other cell, the direction flag of the list is `0`, and `t` has direction `0`. Thus it is impossible to determine what "insertion before `t`" really means. Both positions could be correct.



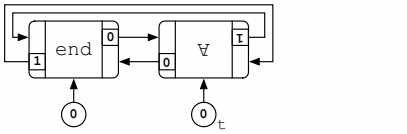
(a) Here inserting before `t` results in a new cell between `end` and `A`



(b) In this configuration inserting before `t` results in a new cell between `A` and `B`



(c) Result of deleting cell `B` in Fig. 3(a). Inserting the new cell should behave as in Fig. 3(a), i. e., should result in a list with the new cell at the beginning



(d) The result of deleting cell `B` in Fig. 3(b). This is identical to Fig. 3(c), but inserting a new cell should behave as in Fig. 3(b), i. e., with the new cell at the end

Fig. 3. Example where the information stored in an iterator is not sufficient to determine where to insert a new cell. The `0` and `1` in the cells denote pointers `p[0]` and `p[1]` respectively. The circles pointing to a cell are iterators. The number in an iterator is its direction flag. The iterator pointing to the end cell is the end iterator, i. e., its direction flag is the direction of the list

Obviously, this problem occurs only on singleton lists as in Fig 3(c). Through the iterator pointing to the only data cell and its direction flag, it is known which pointer of this cell is affected by an insertion. But it is impossible to determine which pointer in the end cell needs to be updated (cf. Fig. 3(c) and 3(d)). If the direction is encoded into the cells using pointers `prev` and `next`, this cannot occur because we know that the corresponding pointer for `prev` in the data cell is `next` in the end cell, for instance.

This problem can and must be resolved to prevent errors. As a particular application consider for example level planarity testing, which has been investigated in graph drawing, see [8, 9]. Like the ordinary graph planarity testing algorithm [2], it uses PQ-Trees. The children of all nodes could be stored in symlists, and in the template matching algorithm it can happen that a Q-node temporarily has only one child.

Since this anomaly occurs only on singleton lists, we simply can keep a *hidden* cell to avoid it. In our implementation this additional cell is always a neighbour of the end cell. This enforces some modifications in the methods of `symlist` to hide the existence of the additional cell. Beside the `insert` and `erase` methods, the `++` and `--` operations on iterators must be changed to ignore the invisible cell. To achieve this efficiently, our extension is to mark each cell according to its type (data, hidden, or end) when it is created.

Exactly the same problem occurs with Tarjan's reversible lists, as described in [17], but on lists of size two and not on singleton lists. This is due to the fact that there is no extra end cell and the global direction of the list is stored within the last data cell, the so called *tail*. Moreover, Tarjan does not describe what previous or next means to an arbitrary external pointer on a cell. In a symlist every iterator has a local direction, for a simple pointer this is impossible. But for some applications this feature is very useful, e. g. for (level) planarity testing.

3.2 Blind Operations

Besides the standard methods on doubly linked lists, we need the more specialised methods `attach_sublist` and `detach_sublist` for maintaining the children of a Q-node. If a planarity testing algorithm uses PQ-trees, sometimes a so called *pseudo Q-node*, e. g. in [2, p. 355], must be created, which temporarily contains a sequence of children out of the middle of another Q-node. This parent Q-node may not be known at the moment, because due to time complexity restrictions inner children of a Q-node do not know their parent in general. Hence, because only the iterators to the relevant children are known, we need a constant time operation `attach_sublist` which sets the adjacency pointers of that pseudo list's end cell `e` to the first and the last data cell, `w` and `v`, of the sublist. The adjacencies of `w` and `v` are adapted accordingly. Moreover, pointers to the previous and next cells from that sequence in the original list are stored in the pseudo list in order to know where the sublist must be inserted in the original list by `detach_sublist` later in the algorithm.

Another useful feature is the ability to insert or erase an element of a symlist if its position is given by an external iterator, but not the symlist object itself.

Therefore, we introduce two static methods `blind_insert` and `blind_erase`, both running in constant time. The use of such methods prohibits maintaining a size counter as a data member of `symlist` which is updated with each operation affecting the size.

4 Conclusion

We have completed and extended an idea of Tarjan [17] for usage in practice. A symlist is a variant of a doubly linked list which does not store any directional information in its cells. This makes it possible both to reverse a symlist and to insert a reversed symlist into another in constant time without affecting the time complexity of the other operations. Usual implementations of doubly linked lists, for instance the STL class `list`, require linear time for reversing [14, 15].

We have introduced new methods for attaching and detaching sublists or insertions and deletions that modify the list without knowing the list object itself. These are very important in the implementation of PQ-trees.

Our implementation of a symlist is similar to the STL class `list`. It is available as part of the GTL library [6]. Furthermore we have enhanced our implementation in order to use it as part of efficient level planarity testing on graphs. These extensions are used in a prototypic implementation of that test which is based on the technique of [8, 9].

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*, pages 204–205. Addison Wesley, 1983.

[2] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computers and System Sciences*, 12:335–379, 1976.

[3] John M. Boyer and Wendy Myrvold. Stop minding your P's and Q's: A simplified $\mathcal{O}(n)$ planar embedding algorithm. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*, pages 140–146, 1999.

[4] S. Even. *Graph Algorithms*, chapter 7–8, pages 148–191. Computer Science Press, 1979.

[5] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*, chapter 5.2. John Wiley & Sons, second edition, 2001.

[6] GTL. Graph Template Library. `http://www.infosun.fmi.uni-passau.de/GTL/`.

[7] L. J. Guibas and J. Stolfi. Primitives for the manipulations of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[8] L. S. Heath and S. V. Pemmeraju. Recognizing level-planar dags in linear time. In F. J. Brandenburg, editor, *Proc. 3rd International Symposium, GD '95*, number 1027 in LNCS, pages 300–311. Springer, 1995.

[9] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. In S. H. Whitesides, editor, *Proc. 6th International Symposium, GD '98*, number 1547 in LNCS, pages 193–204. Springer, 1998.

[10] D. S. Knuth. *The Art of Computer Programming*, volume 1, pages 280–281. Addison Wesley Longman, 3 edition, 1997.

[11] S. Leipert. PQ-trees – an implementation as template class in C++. Technical Report 97.259, Institut für Informatik Universität zu Köln, 1997.

[12] S. Leipert. *Level Planarity Testing and Embedding in Linear Time*. Dissertation, Mathematisch-Naturwissenschaftliche Fakultät der Universität zu Köln, 1998.

[13] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*, chapter 3.2. Cambridge University Press, 1999.

[14] D. R. Musser and A. Saini. *The STL Tutorial and Reference Guide*. Addison Wesley, 1996.

[15] Silicon Graphics, Inc. STL. Standard Template Library. `http://www.sgi.com/tech/stl/`.

[16] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(1):362–391, 1983.

[17] R. E. Tarjan. *Data Structures and Network Algorithms*, chapter 1.3, page 11. CBMS-NSF Regional Conferences Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983.