Static Header as Sentinel

Massimo Ancona† Walter Cazzola†

† DISI-University of Genova, Via Dodecaneso 32, 16100 Genova, Italy
e-mails: ancona@dsi.unige.it

‡ DISI-University of Milano, Via Comelico 39-41, 20135 Milano, Italy
e-mails: cazzola@dsi.unimi.it

September 28, 2008

Abstract

Writing code to handle dynamic data structures might seem to be an easy task, but write an efficient, readable and maintainable code is not such a simple task.

In this short note we investigate some problems in developing code for handling dynamic data structures, and we propose techniques to overcome them. We take into account the interesting method proposed by Qiu in SIGPLAN Notices [3].

Many authors have addressed the problem of handling dynamic data structures by suggesting several clever tricks to simplify dynamic data structures management [2, 4] and to avoid the non-uniform behavior of empty data structures, and access to the first and the last element. In particular, the use of data sentinels and dummy headers makes the implementation more orthogonal and easy to maintain. The cost is a slightly increased size in space which is negligible when compared with the benefits produced.

Keywords: Programming Methodologies.

1 The Problem

Several books and papers address the problem of managing dynamic data structures [2, 4]. When we write code to handle dynamic data structures like lists, queues, trees and so on, we must be handle two related problems:

1. Not all items of the structure have the same properties, e.g. the tree root has not parent item, leaves have no children;

2. Structures are potentially unbounded, but de-facto limited, i.e. each structure always has a terminal item.

Such problems are simple, but hide several traps. In order to avoid such traps algorithms become redundant and include special cases resulting difficult to read, to maintain and often inefficient.

The use of sentinels and dummy headers is recommended by most books introducing elementary data structures and algorithms [2, 4]. The purpose of a sentinel and of a dummy header is to simplify termination and starting tests in a repetition command (usually a while statement) used for visiting the data structure. The presence of an initial dummy header and of a terminal sentinel makes the management of all data elements uniform, because each of them has a predecessor and a successor. Recently Qiu [3] improved the use of data sentinels by proposing a method offering the advantage provided by a dummy header without paying the associated extra cost.

In this paper, we go ahead in that direction by generalizing the approach of Qiu and proposing a method that uses both a dummy header and a terminal sentinel without paying extra costs or, at least, minimizing the extra space required for them depending on the implementation language.
used.

We demonstrate our method for the case of linked lists implemented in C language, but the method can be easily extended to other languages and other dynamic data structures.

In the rest of the paper, we will use the following definition of linked list.

```c
typedef struct node node, *list;
struct node
  int key;
  list next;
```

As running examples we use two simple operations: the first is a key search in a list and the second is a key insertion in an ordered list.

```c
list general_search( list l, int k )
  
  for( ; l != NULL && l->key != k ; l = l->next )
  
  return ((l != NULL)?l:NULL);
```

In this case it could be noted that the problem 2 requires a test performed for all item of the list (l != NULL). Such test is executed, in the worst case, for all items in the list.

Moreover, depending on whether the language/compiler performs a conditional evaluation of boolean expression (i.e. it short circuits boolean expressions or not), the evaluation of condition (l->key != k) may produce a runtime error.

```c
list general_ordered_insert( list l, int k )
  
  for( 12 = NULL, 11 = 1;
       11 != NULL & & 11->key < k;
       11 = 11->next );
  
  if (11 != NULL & & 11->key == k ) return 11;
  
  l3 = (list)malloc(sizeof(node));
  l3->key = k;
  l3->next = 12;
  if (12 = NULL) {
    // inserts the key as the first item
  } else {
    l2 = l1;
    l1 = l3;
    return l1;
  }
```

On the other hand, the above code points out the problem of handling special items (problem 1), like, for example, the first one. First and last items in general require special attention and additional code in order to be correctly handled. The additional code increases code dimension, while reducing readability and maintainability.

2 Qiu’s Solution

In [3] Qiu presented static header, a new technique to retain the advantages of using a dummy header to overcome the drawbacks due to problem 2 without extra space.

The idea is to declare the dummy item local to the function; in this way it is automatically discarded at function termination.

The benefit achieved by Qiu method is shown by the following insertion function (see for comparison the related example of the previous section).

```c
list qiu’s_ordered_insert( list l, int k )
  
  node header; list l1, l2;
  
  for(l2 = &header; l != NULL & & l->key < k; l2 = l, l = l->next);
  
  if (l == NULL & & l->key == k ) {
    l1 = (node *)malloc(sizeof(node));
    l1->key = k;
    l1->next = l;
    return header.next;
  }
```

62
3 Our Extension

The basic idea beyond our proposal is to extend dynamic data with both a dummy header and a terminal sentinel integrated into a single item. In this way we add just one item to the structure that can be managed like the static header of Qiu. To this aim we use a different list model: a circular list. In a circular list the last item is linked to the first one and no NULL pointer is used.

Our idea is to merge the dummy header with the sentinel into a single item and to consider the last item to act as entry element of the list.

Our list structure is sketched in figure 1. We use the following data type to encapsulate our list model:

```
list:
  init : key_type (→ list
  first, next : list (→ list
  key : list (→ key_type
```

For our purposes only the implementation of the init operation is relevant.

```
list init( int k ) {
  new = (node *)malloc(sizeof(node));
  new(key) = k;
  new(next) = new;
  return new;
}
```

The init operation is used to create a new list; it sets all references, according to our list model.

An interesting side-effect of our model is the capability of adding a new item both at the head and at the tail of the list with the same operation.

The only difference is about which item is the new root item after insertion. In the latter case is the new one, while in the former case nothing is changed.

Furthermore such an insert operation has a constant time complexity \( \Theta(1) \).

3.1 The Search Operation

Search operation takes advantage of our model when a static sentinel is inserted between the last and first item of the list, like in the figure below.

Such an item has the advantage of the static header of Qiu and that of a sentinel that eliminates any test on NULL pointers.

Thus, the algorithm shows a uniform behavior on all items, while the sentinel guarantees a simple and safe termination test.

```
list search( list l, int k ) {
  node sentinel;  /* automatic variable */
  list ll;
  /* initialize & insert the sentinel */
  sentinel.key = k;
  sentinel.next = l(→ next);
  l(→ next) = &sentinel;
  /* search for the key */
  for( ll = sentinel.next; ll(→ key) != k; ll = ll(→ next) );
  /* to avoid the dangling reference */
  l(→ next) = sentinel.next;
}
```

Figure 1: Our Model of a List

Figure 2: lab
Note that the sentinel has the same structure of all other items of the list. The sentinel is a name local to the search function body which is automatically discarded at function exit (see [1]). The algorithm does not take care about not finding the key in the list.

There is the risk of introducing a dangling reference when the sentinel is discarded; thus, it is mandatory to re-link the last item of the list to the first one before leaving the function.

3.2 The Ordered Insert Operation

Similar benefits are also obtained when operation insert is implemented for an ordered list.

The difference with Qiu’s proposal is due to the absence of the test on the list termination.

The algorithm starts by inserting a new item: such an item plays the role of list header (it is placed at the first position) and that of a sentinel (it is used as trailer item terminating the search when the key is not present in the list). Then the algorithm looks for a place where to insert the new item. When such a place is found, the new item is moved to the right position.

It is obvious that the sentinel cannot be defined as a local (automatic) variable.

```c
/* return the right item */
return ((l1 != NULL) ? l1: NULL);
```

```c
if (l1(-> key == k) free(header);
else {
    header(-> next = l1;
    l2(-> next = header;
}
return l;
```

When the searched item already exists in the list we have to free the memory used by the header. This is a negligible drawback because there exist very efficient ways to handle dynamic memory allocation and because this operation is not performed often. With a slight redundant implementation, it is possible to maintain an automatic variable for implementing the header with a method very close to that of Qiu.

4 Conclusion

Our approach has the same advantages of that of Qiu. It supports management of list items that is independent of item position, i.e. the first and the last items are handled in the same way, thus reducing the number of tests performed for each item. The number of operations performed by our implementation of search and of ordered insert is, in the worst case, equal to n tests performed in the main loop, while Qiu’s algorithm requires 2n tests, where n is the number of items in the list. Thus, our method has two advantages: it permits to write a more readable and more efficient code.

References