Programming with Indirect Pointers

Tibor Ásványi

Faculty of Informatics, Eötvös Loránd Univ.
Budapest, Pázmány Péter sétány 1/c, H-1117
asvanyi@inf.elte.hu

Abstract. Highly dynamic data collections are often represented by linked data structures, that is, linked lists, trees etc. Inserting into, and deleting from lists and trees are frequent actions while performing search-and-update operations on these data structures. In this paper we consider some non-recursive variants of these operations, and develop effective and simple implementations of them.
The indirect pointers will be used to implement and *simplify the operations* on the data structures, and *not* for the representation of the data structures. In this way, our approach basicly differs form that of other works.

1 Introduction

Using indirect pointers, that is pointers referring to other pointers in search-and-update operations of some linked data structures is a previously unpublished programming method of the author (there is no earlier paper on this topic by the author), who has been teaching it at the Eötvös Loránd University of Budapest for ten years now. During this period, this has become part of the programming practice of his best students, that is, it has become part of the programming folklore. This long practice has proved that we can considerably simplify many list and tree handling operations, if we use indirect pointers.

In order to understand the problem, let us suppose, that we have a simple linked list, and we want to insert a new node into the list, or we want to remove one of the nodes of the list. In both cases we need one program code to perform the update at the beginning of the list, and another for the general case. Similarly, updates at the root of a tree are special, compared to the updates at deeper levels.

In order to symplify the list operations, and to make them a bit more effective, traditionally sentinel nodes are proposed [1, 4, 7]. However, sentinel nodes need extra memory. This memory overhead may be significant, if we have many short lists in our application. (For example, consider hash tables, linked representation of sparse matrices and graphs.)

This paper shows that the sentinel nodes of the one-way lists are usually unnecessary. We get rid of the special cases using indirect pointers. And our method is often useful even for linked trees. (We have to mention that indirect pointers are supported by many widely used, high level programming languages. Consider C, C++, Pascal, Ada, Modula, and so on.)

1.1 The Organisation of This Paper

In this paper we enumerate seven small case studies of classical list and tree handling operations. In all cases we compare the traditional solutions with the solution using indirect pointers, in order to show the usefulness of the author's method.

We will show many example programs, each of them coded in C++ [5]. We will try to get a balance of good programming style, and of avoiding too specific C++ features. After all, we want to focus on the *handling* of the data structures.

We will rely on the following C++ definitions throughout this paper.

```
typedef struct lnode* list;

struct lnode{
    int data;
    lnode* link;
    lnode(){}
    lnode(int d) { data = d; }
    lnode( list L ) { link = L; }
    lnode( int d, lnode* next) { data = d; link = next; }
};
```

2 Building an Unsorted List

In this section we want to build a list from a sequence of data pieces. For simplicity, we get random numbers, and the elements of the list will contain them in the original order. (We suppose that function `int RandNat()` returns positive random integers.)

First we build a simple one-way list.

```
list build_list_0() { // 9 statements
    if( LENGTH>0 ) {
        list L = new lnode(randNat());
        lnode* p = L;
        for( int i = 1 ; i<LENGTH ; ++i ) {
            p->link = new lnode(randNat());
            p = p->link;
        }
        p->link = 0;
        return L;
    }
    else return 0;
}
```

The head of the list containing the first number is generated with a special code, the others in a loop. Also we need a conditional statement to handle the empty list.

Some sources [1] suggest that the head of the list should be a sentinel node, that is a node not used to store data. Using sentinel head we can simplify (and slightly speed up) some list operations [4, 7]. Using sentinel head, empty list is not a special case.

The sentinel head is temporary in the code below. Therefore the interface of this function is the same as that of the previous one. However, in order to simplify the subsequent list operations, too, it may be useful to generate a permanent sentinel head. (The change of the code is trivial.) In that case we have a permanent memory overhead compared to the first solution.

```
list build_list_h( ) { // 7 statements
    lnode H;  // H is a temporary sentinel head.
    lnode* p = &H;
    for( int i = 0 ; i<LENGTH ; ++i ) {
          p->link = new lnode(randNat());
          p = p->link;
    }
    p->link = 0;
    return H.link;
}
```

This simplification can be retained without memory overhead, if we use an indirect pointer `p` to book the end of the list.

```
// Indirect pointer  lnode** p  goes through list L.
// p = &L;      p = &(*p)->link;
// p ... p ...  p ...  p ...  p
// |     |      |      |      |
// V     V      V      V      V
// L->[d|l]->[d|l]->[d|l]->[d|\]

list build_list_i() { // 7 statements
    list L;
    lnode** p = &L;
    for( int i = 0 ; i<LENGTH ; ++i ) {
        *p = new lnode(randNat());
        p = &(*p)->link;
    }
    *p = 0;
    return L;
}
```

In the beginning, `p` refers to the pointer of the list. After this, `p` refers to the link field of the last node of the list. In both cases, we have to link the new end nodes to `*p` (as above).

(We use the same name conventions for the functions in the whole paper: `func_0()` use neither sentinel head nor indirect pointers, `func_h()` use sentinel head, and `func_i()` use indirect pointers.)

3 Deleting from an Unsorted List

In this section we suppose that we have a one-way list of integers, and we want to delete the first occurence of a given number. If there is no such node, we do not modify the list.

First we consider a classical solution for simple one-way lists. Next our list has a sentinel head: the code becomes simpler. Last we put the advantages of the previous solutions together: we use an indirect pointer.

```
void remove_0( list& L, int d ) { // 10 statements
    lnode* pp;  lnode* p = L;
    while( p != 0 &&  p->data != d ) {
        pp = p;  p = p->link;
    }
    if( p != 0 ) {
        if( p == L )  L = L->link;
        else pp->link = p->link;
        delete p;
    }
}
```

```
void remove_h( list L, int d ) { // 8 statements
    lnode* pp = L;  lnode* p = L->link;
    while( p != 0 &&  p->data != d ) {
        pp = p;  p = p->link;
    }
    if( p != 0 ) {
        pp->link = p->link;  delete p;
    }
}
```

```
void remove_i( list& L, int d ) { // 7 statements
    lnode** p = &L;
    while( *p != 0 &&  (*p)->data != d ) {
        p = &(*p)->link;
    }
    if( *p != 0 ) {
        lnode* q = *p;  *p = q->link;  delete q;
    }
}
```

4 Inserting into a Sorted List

In this section we suppose that we have a strictly sorted one-way list of integers, and we want to insert a given number, if it is not already on the list. If it is, we do not modify the list.

First we consider a classical solution for simple one-way lists.

```
void sorted_insert_0( list& L, int d ) { // 9 statements
    lnode* pp;  lnode* p = L;
    while( p != 0 && p->data < d ) {
        pp = p;  p = p->link;
    }
    if( p == 0 || p->data > d )
        if( p == L )  L = new lnode(d,L);
        else pp->link = new lnode(d,p);
}
```

We can give a simpler solution, if the list has a sentinel head:

```
void sorted_insert_h( list L, int d ) { // 7 statements
    lnode* pp = L;
    lnode* p = L->link;
    while( p != 0 && p->data < d ) {
        pp = p;  p = p->link;
    }
    if( p == 0 || p->data > d )
        pp->link = new lnode(d,p);
}
```

But the solution with an indirect pointer is the simplest by far:

```
void sorted_insert_i( list& L, int d ) { // 5 statements
    lnode** p = &L;
    while( *p != 0 && (*p)->data < d ) {
        p = &(*p)->link;
    }
    if( *p == 0 || (*p)->data > d )  *p = new lnode(d,*p);
}
```

5 Merging Sorted Lists

In this case study we suppose that we have two strictly increasing, simple one-way lists, and we want to compute their sorted union in the first one, while the second one is destroyed, and its pointer becomes undefined. Let us consider the three solutions in the usual order.

```
void sorted_union_0( list& L, lnode* q ) { // 21 statements
    if( L == 0 )  L = q;
    else{
        lnode* pp;  lnode* p = L;
        while( p != 0 && q != 0 ) {
            if( p->data > q->data ) {
                lnode* r = q;  q = q->link;
                r->link = p;
                if( p==L )  pp = L = r;
                else{
                    pp->link = r;  pp = r;
                }
            }
            else{ // ( p->data <= q->data )
                if( p->data == q->data ) {
                   lnode* r = q;  q = q->link;  delete r;
                }
                pp = p;  p = p->link;
            }
        }
        if( q != 0 )  pp->link = q;
    }
}
```

In the next solution we use a temporary sentinel head, in order to decrease the number of conditionals performed during the run of the program.

```
void sorted_union_h( list& L, lnode* q ) { // 19 statements
    lnode H(L);  // H is a temporary sentinel head.
    lnode* pp = &H;  lnode* p = L;
    while( p != 0 && q != 0 ) {
        if( p->data > q->data ) {
            lnode* r = q;  q = q->link;
            r->link = p;  pp->link = r;
            pp = r;
        }
        else{ // ( p->data <= q->data )
            if( p->data == q->data ) {
                lnode* r = q;  q = q->link;  delete r;
            }
            pp = p;  p = p->link;
        }
    }
    if( q != 0 )  pp->link = q;
    L = H.link;
}
```

Using indirect pointers, the complexity of the program can be decreased dramatically.

```
void sorted_union_i( list& L, lnode* q ) { // 14 statements
    lnode** p = &L;
    while( *p != 0 && q != 0 ) {
        if( (*p)->data == q->data ) {
            lnode* r = q;  q = q->link;  delete r;
        }
        else if( (*p)->data > q->data ) {
            lnode* r = q;  q = q->link;
            r->link = *p;  *p = r;
        }
        p = &(*p)->link;
    }
    if( q != 0 )  *p = q;
}
```

6 The Difference of Two Sorted Lists

In this section we suppose that we have two strictly increasing, simple one-way lists, and we want to compute their sorted difference in the first one, while the second one is unchanged.

```
void sorted_difference_0( list& L, lnode* q ) { // 16 statements
    lnode* pp;  lnode* p = L;
    while( p != 0 && q != 0 ) {
        if( p->data > q->data )  q = q->link;
        else if( p->data < q->data ) {
            pp = p;  p = p->link;
        }
        else{ // ( p->data == q->data )
            if( p==L ) {
                L = L->link;  delete p;  p = L;
            }
            else{ // ( p != L )
                pp->link = p->link;  delete p;  p = pp->link;
            }
            q = q->link;
        }
    }
}
```

We can spare the last test in the loop, if we use a temporary sentinel head. In this way, we can get some speed up.

```
void sorted_difference_h( list& L, lnode* q ) { // 14 statements
    lnode H(L);  // H is a temporary sentinel head.
    lnode* pp = &H;  lnode* p = L;
    while( p != 0 && q != 0 ) {
        if( p->data > q->data )  q = q->link;
        else if( p->data < q->data ) {
            pp = p;  p = p->link;
        }
        else { // ( p->data == q->data )
            pp->link = p->link;  delete p;
            p = pp->link;  q = q->link;
        }
    }
    L = H.link;
}
```

Again, the complexity of the program can be decreased much more, if we use indirect pointers.

```
void sorted_difference_i( list& L, lnode* q ) { // 10 statements
    lnode** p = &L;
    while( *p != 0 && q != 0 ) {
        if( (*p)->data > q->data )  q = q->link;
        else if( (*p)->data < q->data )  p = &(*p)->link;
        else { // ( (*p)->data == q->data )
            lnode* r = *p;  *p = r->link;  delete r;
            q = q->link;
        }
    }
}
```

7 Inserting into a Binary Search-tree

In this section and in the next one we need the following types.

```
typedef struct tnode* tree; // binary tree

struct tnode{
    int data;
    tnode* left;  tnode* right;
    tnode( int d ) { data = d;  left = right = 0; }
};
```

We suppose that we have an unbalanced binary search tree of integers, with no duplicated elements. We want to insert a number, except if it is already contained. The traditional solution is something like this:

```
void sorted_insert_0( tree& t, int d ) { // 15 statements
    if( t == 0 )  t = new tnode(d);
    else{
        tnode* p = t;
        for(;;) {
            if( d < p->data )
                if( p->left )  p = p->left;
                else{ p->left = new tnode(d);  break; }
            else if( d > p->data )
                if( p->right )  p = p->right;
                else{ p->right = new tnode(d);  break; }
            else break;
        }
    }
}
```

Using indirect pointers, we can refer to the root pointer, and to the left and right links of the nodes.

```
// Indirect pointers  p, q, r, s  in tree  T
// (long pointer refer to the whole object,
//  short pointer refer to the designated field)
//
//                      p == &T
//                      |
//                      V
//                      T
// &(*p)->left == q     |    r == &(*p)->right;
//                |     |    |
//                V     V    V
//              [left|data|right]
//                           |    s == &(*r)->right;
//                           |    |
//                           V    V
//                    [left|data|right]
```

In this way we get a new solution. And its complexity is roughly the half of the complexity of the original.

```
void sorted_insert_i( tree& t, int d ) { // 8 statements
    tnode** p = &t;
    while( *p != 0 ) {
        if( d < (*p)->data )  p = &(*p)->left;
        else if( d > (*p)->data )  p = &(*p)->right;
        else return;
    }
    *p = new tnode(d);
}
```

8 Deleting from a Binary Search-tree

In our last problem we have un unbalanced binary search tree with no duplicated element. And we want to delete a given value from it.

According to a non-recursive variant of a classical solution [6], first we try to find the node containing this value (together with its parent node). If this node (containing the value to be deleted) has at most one child node, it can be deleted similarly to a node of a simple one-way list. If it has two children, we find the leftmost element of its right subtree, and copy the content of that node into it. After this, we delete that leftmost node.

```
void remove_data_0( tree& t, int d ) { // 31 statements
    tnode* p = t;  tnode* q;
    while( p != 0 && p->data != d ) {
        if( d < p->data ) { q = p;  p = p->left; }
        else{ // ( d > p->data )
             q = p;  p = p->right;
        }
    }
    if( p != 0 ) {
        if( p->left == 0 )
            if( p == t )  t = p->right;
            else if( p == q->left )  q->left = p->right;
            else q->right = p->right;
        else if( p->right == 0 )
            if( p == t )  t = p->left;
            else if( p == q->left )  q->left = p->left;
            else q->right = p->left;
        else{ // (p->left != 0 && p->right != 0 )
            tnode* r = p->right;
            while( r->left ) {
                q = r;  r = r->left;
            }
            p->data = r->data;
            if( r == p->right )  p->right = r->right;
            else q->left = r->right;
            p = r;
        }
        delete p;
    }
}
```

In the code above, we have to distinguish the the root node from other nodes, and the left children from the right children, too. Both distinctions become unnecessary, if we use indirect pointers, because such a pointer refers to the link pointing to the actual node.

```
void remove_data_i( tree& t, int d ) { // 18 statements
    tnode** p = &t;
    while( *p != 0 && (*p)->data != d ) {
        if( d < (*p)->data )  p = &(*p)->left;
        else p = &(*p)->right;
    }
    tnode* q = *p;
    if( q != 0 ) {
        if( q->left == 0 )  *p = q->right;
        else if( q->right == 0 )  *p = q->left;
        else{ // (q->left != 0 && q->right != 0 )
            tnode** r = &q->right;
            while( (*r)->left != 0 )  r = &(*r)->left;
            q->data = (*r)->data;
            q = *r;  *r = q->right;
        }
        delete q;
    }
}
```

Clearly, this last problem is the most complicated. And, if we use indirect pointers, the gain is substantial.

9 Benchmarks

For the tests the author used *Linux*, and the *g++* compiler together with the *gprof* program.

The "self seconds" data of the different functions is listed below (calls = the number of invocations of the different functions, size = the average size of the input lists/trees, Ord.sol = ordinary solution, Sent.head = solution with sentinel head, Ind.point. = solution with indirect pointers, speed(O/I) = the overhead of the ordinary solution compared to the solution with indirect pointers).

It is easy to see that the ordinary solutions have about 10% run-time overhead compared to the ones with indirect pointers.

	calls	size	Ord.sol.	Sent.head	Ind.point.	speed(O/I)
Build list	20000	10000	0.96	1.05	0.87	110%
Remove node	100000	20916	34.60	32.04	30.96	112%
Sorted insert	100000	15453	12.17	11.84	10.55	115%
Sorted union	5000	10000	3.87	3.51	3.40	114%
Sorted diff.	5000	10000	3.81	3.61	3.54	108%
b.s.tree ins.	1500000	473778	2.58	–	2.40	108%
b.s.tree del.	1500000	648186	2.76	–	2.61	106%

The complexities of the C++ implementations (number of statements) are collected into the following table. Our average gain in program complexity is more than 60%. It is much more significant than the reduction of run-time.

	Ordinary sol.	Sent.head	Ind.point.	size(O/I)
Build list	9	7	7	129%
Remove node	10	8	7	143%
Sorted insert	9	7	5	180%
Sorted union	21	19	14	150%
Sorted diff.	16	14	10	167%
b.s.tree ins.	15	–	8	188%
b.s.tree del.	31	–	18	172%

10 Conclusions

In this paper we used indirect pointers in order to simplify search-and-update operations of simple one-way linked lists and trees.

In the case of two-way lists and trees, the author does not see any possibility of taking advantage of indirect pointers. He thinks, they serve to help us, when we want to update one-way lists and trees.

We investigated seven classical problems and we found that the corresponding programs are more effective and much simpler, if we use indirect pointers. Compared to the use of sentinel nodes, the gain is more, and we do not need extra memory. The simplification of the programs is more significant, if the problem is more complicated.

These results are in harmony with the authors experiences while using this programming method in many other programs, and teaching it during the last decade at the Eötvös Loránd University in Budapest.

References

1. A. V. Aho, J. E. Hopcroft, J. D. Ullman: *Data Structures and Algorithms.* Addison-Wesley, 1983.
2. D. E. Knuth: *Fundamental Algorithms, The Art of Computer Programming.* Addison-Wesley, 1997.
3. D. E. Knuth: *Sorting and Searching, The Art of Computer Programming.* Addison-Wesley, 1997.
4. T. H. Cormen, C. E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms.* The MIT Press/McGraw Hill, Cambridge/Boston, 2003.
5. B. Stroustrup: *The C++ Programming Language* (Third Edititon). Addison-Wesley, 2002.
6. N. Wirth: *Algoritms + Data Structures = Programs.* Prentice-Hall Inc. 1982.
7. `http://en.wikipedia.org/wiki/Linked_list`