OBJECT-ORIENTED LISTS WITH CONTEXTS

AREAS : OBJECT-ORIENTED PROGRAMMING / DATA STRUCTURES

C.T. Zahn
CSIS
Pace University
1 Martine Ave.
White Plains, NY 10606
Email: ZAHN@ PACE.EDU

J.A. Bergin
CSIS
Pace University
One Pace Plaza
New York, NY 10038
BERGINF@.EDU

ABSTRACT

An abstract data type is introduced for general handling of sequences of elements. This ADT features contexts which point between elements of a sequence and is implemented by doubly-linked circular lists with headers. DCLH employs the exclusive-or of two pointers to achieve two-way traversal at a cost of one access field per node. This ADT and its DCLH implementation are designed as a system of three object classes – Lists, Nodes and Contexts – and programmed in C++. The symmetry of DCLH with respect to forward and backward traversal allows an elegant and compact coding of the low-level routines. Mathematical properties of exclusive-or are crucial to the correctness of the implementation. This paper also illustrates the important object-orientied programming concept of linked abstractions. The DCLH Abstract Data Type is composed of three distince Data Abstractions: Lists, Nodes, and Contexts.

0. INTRODUCTION

An earlier paper [1] recommended the use of contexts pointing between elements of a sequence and showed how this idea fit nicely with an old exclusive-or method for achieving two-way traversal of linked-lists at a cost of one access field. To avoid many of the problems that arise in processing linked-lists an implementation was developed using two header nodes, exclusive-or access fields, and contexts consisting of a pair of pointers. This doubly-linked circular list with headers is abbreviated DCLH.

Contexts allow unambiguous insertion operations and any subsequence between two contexts can be processed as a sublist, deleted or copied etc. A context is clearly the way to identify where a list should be split in two. Its usefulness in iterators has been pointed out in [3].

The symmetry of DCLH allows two operations to be combined in many cases where a forward and a backward version are desired. For example, rather than operations Advance and Retreat to move a context we have Advance (direction) with a simple boolean parameter. The code for Advance(direction) is as compact as the one way Advance and does not use a two-way case analysis.

In Section 1. we present the abstract view of sequences with contexts and in Section 2. the details of the DCLH implementation. In Section 3. we review the algebraic properties of exclusive-or ( ) and present the invariant properties of the DCLH, highlighting the left-right symmetry.

Section 4. discusses the operations on a DCLH necessary for convenient use and Section 5. shows how the DCLH and its operations can be designed in an object-oriented fashion with three classes of objects. Section 6. introduces the context invalidation problem which will be solved in a separate paper. This section also discusses subtleties in the definition and implementation of Append and Split for lists. Finally, Section 7. gives details of selected parts of a C++ implementation and discusses various problems encountered.

1. SEQUENCES WITH CONTEXTS

Our abstract view of a sequence can be depicted as a sequence of symbols starting with H (for head), ending with T (for tail), with the sequence of values listed in order separated by periods. For example, the sequence (a, b, c) will be depicted as

H.a.b.c.T

in which the periods represent the four contexts where new elements could be inserted. The head and tail are not necessary but allow for a more reasonable looking empty sequence, H.T, as well as assuring that each context is between two "elements".

To indicate a context variable pointing into a sequence we use a caret symbol (s). The sequence (a,b,c) with context (b,c) singled out is depicted as

H.a.bs c.T

Any operation on sequences can be accomplished by manipulating contexts and making insertions and removals based on contexts.

2. DOUBLY-LINKED CIRCULAR LISTS WITH HEADERS

In the DCLH implementation of a sequence there is a node to represent each element of the sequence including the head and tail boundary elements. These nodes form a doubly-linked circular list structure shown by dashed lines in Figure 1. (The left and right dashed "tails" are connected to each other.) Each node contains a data field for the element value and a single access field set to the exclusive-or ( ) of the addresses of the previous and following nodes as shown. This idea is part of the lore of programming [2] and allows forward and backward traversal of the list so long as each reference into the list consists of a pair of adjacent node addresses. Such pairs define contexts, which are our preferred position pointers.

The properties of exclusive-or which make this possible are presented in Section 3. Both head and tail are necessary for an elegant and symmetric version of DCLH. With a single boundary node one cannot distinguish between an empty list and one with a single element. It is also very important that exclusive-or ( ) obey the "distinctness" property

X =Y if and only if X Y = 0

since the empty list H.T has access (H) = address (T) address (T) = 0 and similarly access (T) = 0 . On the other hand,

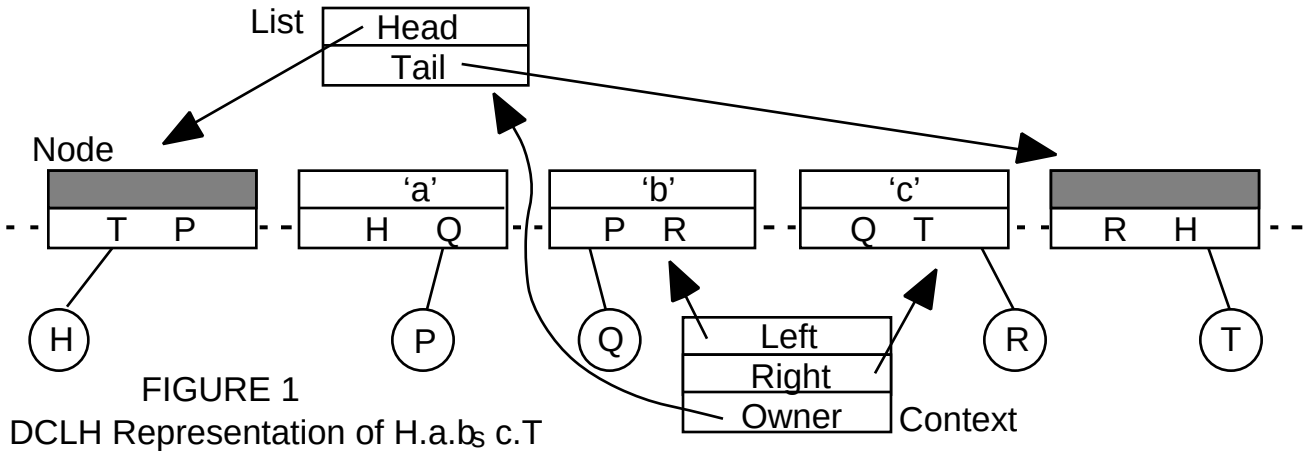

FIGURE 1
DCLH Representation of H.a.b$_S$ c.T

any list with one or more elements will have access (H) ≠ 0 and access (T) ≠ 0, because a node's two neighbors must be distinct in this case!

As depicted in Figure 1, The DCLH data structure is completed by giving each list pointer fields to both tail and head nodes (what amounts to a standard context) and by giving each context an owner field to bind it permanently to a specific list. The owner field is needed to detect illegal attempts to move a context past a boundary in the list. Note that the standard (tail to head) context is illegal as a user context but allows easy traversal to either the first or last context in the list. This is a consequence of having two boundary elements and making the list circular.

3. INVARIANT PROPERTIES AND SYMMETRY

The following algebraic properties of exclusive-or ( ) are crucial for the DCLH implementation of sequences:

A (B C) = (A B) C	associative
A B = B A	commutative
A 0 = A	identity
A A = 0	self inverse
A B = 0 if and only if A=B	distinctness
(A X) X = A	cancellation
A New = (A Old) (Old New)	substitution

The cancellation law depends on the earlier laws and is basic to the use of the access field in nodes. The substitution law is used by several algorithms to replace an access field (A Old) by (A New) when one neighbor of a context has changed. The law we have called "distinctness" is crucial to the detection of an empty list as mentioned above.

The main invariant for a DCLH list promises that if
(H = a_0, a_1,... a_k,...a_n , a_{n+1} = T) is an n element list then Access (a_j) = address (a_{j-1}) address (a_{j+1}) where index arithmetic is modulo n+2.

Immediate Corollaries are

address (a_{j-1}) = address (a_{j+1}) access (a_j)

address (a_{j+1}) = address (a_{j-1}) access (a_j)

These formulas allow forward and backward traversal of any DCLH lists.

Whenever the list L is non-empty, access (head (L)) $\neq$ 0, so we can get to the first or last element by

address (first (L)) = tail (L) access (head (L))

address (last (L)) = head (L) access (tail (L))

The key context invariant promises that if context C has owner List (H = a_0, a_1, a_2, ..., a_n,a_{n+1} = T) then for some k such that 0 $\leq$ k $\leq$ n, left (C) = address (a_k) and right (C) = address (a_{k+1}).

Reversing a list L requires only swap (head (L), Tail (L)).

The uncompromised left, right symmetry of this data structure will be exploited in the final low-level algorithms. This will require simply that the context pairs (tail, head) and (left, right) be 2-element arrays indexed by a 0-1 direction value and the addition of a direction parameter to several operations. For example, instead of two procedures Advance and Retreat to move a context we shall have a single procedure *advance (direction)* which *does not* involve a 2-case analysis!

We note for completeness that the techniques described here will fail in the presence of a generational (or similar) garbage collector that moves objects. This is due to the fact that the access values are computations based on pointers and not pointers themselves.

4. OPERATIONS FOR THE DCLH

To make effective use of DCLH lists requires the following operations (described abstractly):

Create a List or Context.
Make a list Empty.
Ask if a List Is Empty?
Place a Context at First or Last position in a List.
Move a Context forward or backward.
Ask if a Context is At End? or At Beginning? of a List.
Insert value at a Context in a List.
Remove the value before or after a Context.
Return the Value before or after a Context.
Set the Value before or after a Context.

These are probably a minimal set of operations in some reasonable sense. For example, one could place a context at the end of a list by placing it first and moving it forward until it is at the end. In the strict sense one of the two placement operations is redundant, but common sense dictates both be implemented directly. The symmetry of DCLH allows both operations to be folded into a single compact procedure.

Other operations that deal directly with Lists include the following:

Insert a value at First or Last position in a List.
Return First or Last value in a List.
Remove First or Last value in a List.
Find First or Last value in a List equal to a given value
Remove All values from a List equal to a given value.
Reverse a List.
Append two Lists, creating a longer List.
Split a List into two Lists at a Context.

These operations allow full access at both ends of a sequence (a so-called dequeue). They can all be implemented using contexts and the operations on contexts described above. Append and Split involve some subtle problems that are discussed in the following sections.

5. OBJECTS, CLASSES AND METHODS

It seems natural to implement the DCLH using three classes of objects – Lists, Contexts, and Nodes. The nodes are the containers for values and have an access and value field. All three object types are represented in Figure 1. Nodes are used exclusively to build Lists and properly belong to a unique list. Sharing is not allowed, although the user program can use pointer indirection to achieve sharing of node values among two or more lists. The creation of a context always links it to a unique list – a binding that is unbreakable.

It is convenient to think of the DCLH as having a type parameter governing the data type of the values in the sequences so that a form of parametric polymorphism may be achieved as in functional languages like ML [5]. This can be done if each class is parametrized by the data type.

Most DCLH operations fall obviously into one of these classes. List creation, emptying, and the empty query belong to the List class as do Insert First, First, and Remove First, and analogous operations at the Last position. Remove All, Reverse, and Append also seem appropriate to lists but Split is ambiguous.

All the operations in Section 4 that involve a context naturally belong to the Context class. Also operations that find values are Context operations, since they need to tell us "where" the value is found.

One operation that is conceptually attractive but somewhat awkward in the object-oriented paradigm is to Remove the subsequence between two contexts. The method is applied to one of the two contexts implicitly, while the other context must be supplied as a parameter.

Symmetric operations use a direction parameter with values *forward* and *backward* to determine in which direction the operation will be applied. All such operations have a default value of *forward*. For example, *removeFirst()* or *removeFirst(forward)* removes the first element of a List, while *removeFirst(backward)* removes the last element.

6. CONTEXT INVALIDATION, APPEND, AND SPLIT

The main invariant which assures internal consistency of the DCLH list can be maintained by careful implementation of all operations that actively modify a list. These are list creation and clear which must initialize to the empty list (access values in head and tail are both NULL), insert, and remove element. The correctness of these operations depends on the validity of any context that may be used so the context invariant must be assured for every context. An operation that directly manipulates a context can be checked for maintenance of the invariant, but a problem arises since List modifications may invalidate one or more contexts that are not directly involved in the operation.

The insertion of a new value node at context C = (P,R) will leave C as a valid context before or after the inserted node, but any other list context that is a clone of the original C will now be incorrect since the left and right fields no longer represent neighboring nodes; P and R are now separated by the new node.

Removal of the node Q will invalidate any other context referring to Q since the context invariant requires its left and right fields to refer to neighboring nodes of the list, certainly not any deleted ones.

The implementation described in this paper makes it the programmer's responsibility to avoid the use of invalidated contexts. In a separate paper [6] we shall describe an automatic implementation of context validity. This involves each list keeping a list of its "owned" contexts and patching any damaged contexts on the spot. There are several subtleties and a nasty danger of infinite recursion, both in definition and operation – a list contains a list of the contexts into itself!

Correctness concerns are simplified by the design choice to implement most List operations via Context operations. In particular, Insert First which is a list method uses the context method Insert. As a result all modifications to a list occur through the two context operations Insert and Remove. If these two preserve the DCLH invariant, then all other operations do likewise.

Append is an operation that concatenates two existing lists L1 and L2. This is a very efficient operation in DCLH as was shown in [1]. If we choose to enlarge L1 to the new list and leave L2 empty, there is still the problem of invalidation of contexts originally belonging to L2.

Split is an operation that divides a list L into two separate lists L1 and L2 at a context C. If we leave the prefix list L1 as the new value of L and deliver L2 as a function return or through a variable parameter there is still a problem with the original contexts. In principle each original context of L, except C and its clones, can be assigned to L1 or L2 in a perfectly natural way. The clones of C can be assigned at the end of L1 or beginning of L2. Append and Split are not implemented in the current version of the software.

7. IMPLEMENTATION OF DCLH IN C++

The implementation defines three C++ template classes, parametrized by type <data>. The classes all have the usual suite of constructors and overloaded operators. The user may therefore pass lists and contexts by value or by reference, and may freely assign one list to another, etc. We show excerpts from the class definitions below.

The ListNode class has private constructors, making direct manipulation of nodes impossible except by Lists and Contexts, which are ListNode friends.

```
template <class data>  class ListNode {
public:
    data at()const{return _value;};
    void atPut(const data val){_value = val;};
private:
    data _value;
    ListNode<data> *_access;
        // This will not be a valid address but the XOR  of the two addresses of the left and right
        //  neighbors.  DO NOT DEREFERENCE the access value.
```

```
    ListNode(data val, ListNode<data> *left,
            ListNode<data> *right;
    ListNode();
    ListNode(const ListNode<data> & N);
    ~ListNode();
    const ListNode<data> & operator=
        (const ListNode<data>& N);
friend class List<data>;
friend class Context<data>;
};
```

The system will generate only pointers to ListNode objects, so the default constructor will not actually be used. Within the ListNode constructor and elsewhere, access values of nodes are produced and manipulated by a function ExclOr, which is machine dependent. On machines in which a long is similar to a pointer we may use:

```
template <class data>
ListNode<data> *ExclOr(ListNode<data> *a,
                       ListNode<data> *b)
{   return (ListNode<data>*)((long)a^(long)b);
}
```

The direction type is an enumeration, though we could use a boolean value. It is logically the subscript type of the head nodes in Lists.

```
    enum direction{backward=0, forward=1};
```

When a List is created we require the user to supply a valid value of the data type. This value is not stored (logically) but will be returned if the user tries to get a value from an empty list. The dummy value is actually stored in the head nodes. The implementation uses an array _head of two ListNode pointers. _head[0] is the "tail" and _head[1] is the "head" of the list, though the full symmetry makes the names head and tail redundant.

```
template <class data> class List {
public:
    List(data dummy); // empty list
    List(List& L);
    ~List();
    List & operator= (List& L);
    void insertFirst(const data val, direction d = forward);
        // Insert val at end "d" of the list.
    data first(direction d = forward)const;  // Returns data
        // in first location, or dummy for empty list.
    data removeFirst(direction d = forward);
        // Removes and returns data at beginning(end) of
        // list.  Returns dummy and is otherwise
        // a no-op for an empty list.
    void reverse(void);
    bool empty()const; // TRUE for an empty list.
    void clear();  // Removes all data from the list,
        // leaving it empty.
    void removeAll(data val);
        // Removes all copies of val from this list.
    long length(void);
private:
    ListNode<data> *_head[2];
    long _length;
    ListNode<data> *newNode(data val,
        ListNode<data> *left, ListNode<data> *right);
        // Returns an initialized ListNode*
friend class Context<data>;
};
```

Many of the List operations are implemented via Contexts. InsertFirst is typical.

```
template <class data>  void List<data>::
insertFirst (const data val, direction d = forward)
{    Context<data> c(*this,d);
     c.insert(val, d);
}
```

The interesting operations are provided by Contexts, which are also implemented with a two element array, as well as an owner (List) reference. Using a reference variable rather than a pointer guarantees that the value is always initialized during construction and that it is never changed.

```
template <class data> class Context {
public:
     Context(List<data> & L, direction d = forward);
          // Create a context at the "d" end of a List L.
     Context(const Context<data>& C,
                    direction d = forward);
     ~Context();
     const Context<data> & operator=
                    ( const  Context<data>& C);
     void insert(const data val, direction d = forward);
          // Creates a new location at this position; i.e.
          // between the two referenced values.
          // Leaves the location after the new value in the
          // d direction.
     void toFirst(direction d = forward);
          // move to the position before the first value.
     data next(direction d = forward) const;
          // Returns data in next position.
     void setNext(const data val,
                    direction d = forward) const;
          // Set the data in the following position if any.
     data removeNext(direction d = forward);
          // Returns the data removed from the list.
          // Returns dummy if nothing can be removed.
     bool atEnd(direction d = forward)const;
          // TRUE if after the last element.
     void advance(direction d = forward);
          // No-op if at end.
     bool findNext(data val, direction d = forward);
          // Moves forward until the data can be found in
          //  the next location. Does not wrap around the
          // end.  Does not move if data is immediately
          // "forward" in direction d.  Does not move if data
          // cannot be found in direction d.
          // Requires operator== from the <data> type.
     bool iterate(data &v, direction d = forward);
          // If not after last item then set v to the next item,
          // advance,  and return true. Otherwise do not set
          // v, leave position  unchanged and return false.
private:
     List<data> & _owner;
     ListNode<data> *_nbr[2];
friend class List<data>;
friend ostream&
     operator<< (ostream& outp , List<data>& val);
};
```

First, we show how symmetry is exploited. To move a Context to one end of the list or the other does not require a case analysis. We simply need to consider that 1-d represents the “opposite” direction, or the “opposite end” of the List, or

the “other” pointer in a Context. (Henceforth we shall omit the template heads and the Context<data>:: specifiers from the Context member function definitions. These definitions will appear inline in the Context class definition in any case.)

```
void toFirst(direction d = forward)
{     _nbr[1-d] = _owner._head[d];
      _nbr[d] =  ExclOr(_owner._head[1-d],
                        _owner._head[d]->_access);
}
```

Similarly, advance will move either direction without case analysis.

```
void advance(direction d = forward)
{     if(! atEnd( d)){
            ListNode<data> *temp = _nbr[1-d];
            _nbr[1-d] = _nbr[d];
            _nbr[d] = ExclOr(temp, _nbr[d]->_access);
      }
}
```

The most important operation, of course, is insert. It must create a new node between the two it refers to, give it an access value, as well as recomputing access values of the two nodes it originally referenced. It must also move its “other” neighbor pointer to point to the newly created node so that it remains a valid Context.

```
void insert(const data val, direction d = forward)
{     ListNode<data> *aNode =
            new ListNode<data>(val, _nbr[0],_nbr[1]);
      _nbr[0]->_access =
      ExclOr(_nbr[0]->_access, ExclOr(_nbr[1], aNode));
      _nbr[1]->_access =
       ExclOr(_nbr[1]->_access, ExclOr(_nbr[0], aNode));
      _nbr[1-d] = aNode;
      _owner._length++;
}
```

Note that the Context performing the insert is correctly updated, but that another context referencing the same two original nodes will be left pointing to two nodes that are not adjacent. If we try to advance that Context it will become hopelessly lost since the address computations within ExclOr depend fundamentally on adjacency. This problem of invalidating contexts upon insesrt or removal from a list is not unique to this system. The usual singly linked list, with pointers to nodes representing positions suffers from the same defect, though perhaps only on removal.

RemoveNext requires a bit more work, since we also need to delete a node. This operation also returns the data that is removed. We can therefore both retrieve and remove an item if we need to. Of course the returned value may be ignored and discarded, as C++ permits calling any function as a proper procedure.

```
data removeNext(direction d = forward)
{    data result = _nbr[d]->at();
      if(! atEnd(d)){
          ListNode<data> *n =
              ExclOr(_nbr[1-d], _nbr[d]->_access);
          _nbr[1-d]->_access =
              ExclOr(_nbr[1-d]->_access, ExclOr(_nbr[d], n));
          n->_access =
              ExclOr(n->_access, ExclOr(_nbr[d],_nbr[1-d]));
          ListNode<data> *temp = _nbr[d];
          _nbr[d] = n;
          delete temp;
          _owner._length--;
      }
     return result;
}
```

The software described in this paper is available from the authors in a more complete form than described here.

8. CONCLUSION

The design and implementation of lists presented here is significant to educators in two ways. First, it gives an implementation that is symmetric, efficient, and that exploits an important technique; namely the properties of XOR. Second, the design succeeds as a good example of partitioning of functionality between classes that together provide a complete solution to the needs of list users. Many other designs, notably those that attempt to use a single class for list, with a notion of “current position” fail in that they do not provide a means of implementing algorithms requiring multiple simultaneous positions. Finally the design provides a means of exploring the problem of context invalidation.

REFERENCES

[1] "A General Implementation of Sequence" [Zahn, 1989]
[2] *The Art of Computer Programming* [Knuth, 1968]
[3] *C++ Strategies and Tactics* [Murry, 1993]
[4] *Data Abstraction: The Object-Oriented Approach Using C++* [Bergin, 1994]
[5] *Programming in ML* [Ullman, 1994]
[6] "Solving the Context Invalidity Problem in DCLH Lists" [Bergin & Zahn, 1995]