WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Français
Italiano
עברית
日本語
Русский
Српски / srpski
ไทย
中文
Edit links

XOR linked list

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(October 2009)*

An **XOR linked list** is a data structure used in computer programming. It takes advantage of the bitwise XOR operation to decrease storage requirements for doubly linked lists.

Contents [hide]

Description [edit]

An ordinary doubly linked list stores addresses of the previous and next list items in each list node, requiring two address fields:

```
...  A       B         C         D         E  ...
        ->  next ->  next  ->  next  ->
        <-  prev <-  prev  <-  prev  <-
```

An XOR linked list compresses the same information into *one* address field by storing the bitwise XOR (here denoted by ⊕) of the address for *previous* and the address for *next* in one field:

```
...  A        B         C         D         E  ...
       <->  A⊕C  <->  B⊕D  <->  C⊕E  <->
```

More formally:

```
 link(B) = addr(A)⊕addr(C), link(C) = addr(B)⊕addr(D), ...
```

When you traverse the list from left to right: supposing you are at C, you can take the address of the previous item, B, and XOR it with the value in the link field (B⊕D). You will then have the address for D and you can continue traversing the list. The same pattern applies in the other direction.

```
    i.e.  addr(D) = link(C) ⊕ addr(B)
    where
          link(C) = addr(B)⊕addr(D)
     so
          addr(D) = addr(B)⊕addr(D) ⊕ addr(B)

          addr(D) = addr(B)⊕addr(B) ⊕ addr(D)
    since
           X⊕X = 0
           => addr(D) = 0 ⊕ addr(D)
    since
```

```
             X⊕0 = x
             => addr(D) = addr(D)
      The XOR operation cancels addr(B) appearing twice in the equation and all we are
left with is the addr(D).
```

To start traversing the list in either direction from some point, you need the address of two consecutive items, not just one. If the addresses of the two consecutive items are reversed, you will end up traversing the list in the opposite direction.

Theory of operation [edit]

The key is the first operation, and the properties of XOR:

- X⊕X = 0
- X⊕0 = X
- X⊕Y = Y⊕X
- (X⊕Y)⊕Z = X⊕(Y⊕Z)

The R2 register always contains the XOR of the address of current item C with the address of the predecessor item P: C⊕P. The Link fields in the records contain the XOR of the left and right successor addresses, say L⊕R. XOR of R2 (C⊕P) with the current link field (L⊕R) yields C⊕P⊕L⊕R.

- If the predecessor was L, the P(=L) and L *cancel out* leaving C⊕R.
- If the predecessor had been R, the P(=R) and R cancel, leaving C⊕L.

In each case, the result is the XOR of the current address with the next address. XOR of this with the current address in R1 leaves the next address. R2 is left with the requisite XOR pair of the (now) current address and the predecessor.

Features [edit]

- Given only one list item, one cannot immediately obtain the addresses of the other elements of the list.
- Two XOR operations suffice to do the traversal from one item to the next, the same instructions sufficing in both cases. Consider a list with items `{…B C D…}` and with R1 and R2 being registers containing, respectively, the address of the current (say C) list item and a work register containing the XOR of the current address with the previous address (say C⊕D). Cast as System/360 instructions:

```
X  R2,Link    R2 <- C⊕D ⊕ B⊕D (i.e. B⊕C, "Link" being the link field
                                 in the current record, containing B⊕D)
XR R1,R2      R1 <- C ⊕ B⊕C    (i.e. B, voilà: the next record)
```

- End of list is signified by imagining a list item at address zero placed adjacent to an end point, as in `{0 A B C…}`. The link field at A would be 0⊕B. An additional instruction is needed in the above sequence after the two XOR operations to detect a zero result in developing the address of the current item,
- A list end point can be made reflective by making the link pointer be zero. A zero pointer is a *mirror*. (The XOR of the left and right neighbor addresses, being the same, is zero.)

Drawbacks [edit]

- General-purpose debugging tools cannot follow the XOR chain, making debugging more difficult; [1]
- The price for the decrease in memory usage is an increase in code complexity, making maintenance more expensive;
- Most garbage collection schemes do not work with data structures that do not contain literal pointers;
- XOR of pointers is not defined in some contexts (e.g., the C language), although many languages provide some kind of type conversion between pointers and integers;
- The pointers will be unreadable if one isn't traversing the list — for example, if the pointer to a list item was contained in another data structure;
- While traversing the list you need to remember the address of the previously accessed node in order to calculate the next node's address.
- XOR linked lists do not provide some of the important advantages of doubly-linked lists, such as the ability to delete a node from the list knowing only its address or the ability to insert a new node before or after an existing node when knowing only the address of the existing node.

Computer systems have increasingly cheap and plentiful memory, and storage overhead is not generally an overriding issue outside specialized embedded systems. Where it is still desirable to reduce the overhead of a linked list, unrolling provides a more practical approach (as well as other advantages, such as increasing cache performance and speeding random access).

Variations [edit]

The underlying principle of the XOR linked list can be applied to any reversible binary operation. Replacing XOR by addition or subtraction gives slightly different, but largely equivalent, formulations:

Addition linked list [edit]

```
 ...  A        B         C         D         E  ...
         <->  A+C  <->  B+D  <->  C+E  <->
```

This kind of list has exactly the same properties as the XOR linked list, except that a zero link field is not a "mirror". The address of the next node in the list is given by subtracting the previous node's address from the current node's link field.

Subtraction linked list [edit]

```
 ...  A        B         C         D         E  ...
         <->  C-A  <->  D-B  <->  E-C  <->
```

This kind of list differs from the "traditional" XOR linked list in that the instruction sequences needed to traverse the list forwards is different from the sequence needed to traverse the list in reverse. The address of the next node, going forwards, is given by *adding* the link field to the previous node's address; the address of the preceding node is given by *subtracting* the link field from the next node's address.

The subtraction linked list is also special in that the entire list can be relocated in memory without needing any patching of pointer values, since adding a constant offset to each address in the list will not require any changes to the values stored in the link fields. (See also Serialization.) This is an advantage over both XOR linked lists and traditional linked lists.

See also [edit]

- XOR swap algorithm

References [edit]

1. ^ http://www.iecc.com/gclist/GC-faq.html#GC,%20C,%20and%20C++

External links [edit]

- Example implementation in C++.
- Prokash Sinha, A Memory-Efficient Doubly Linked List // LinuxJournal, Dec 01, 2004

v · t · e	Data structures [hide]
Types	Collection · Container
Abstract	Associative array · Double-ended priority queue · Double-ended queue · List · Map · Multimap · Priority queue · Queue · Set (multiset) · Disjoint Sets · Stack
Arrays	Bit array · Circular buffer · Dynamic array · Hash table · Hashed array tree · Sparse array
Linked	Association list · Linked list · Skip list · Unrolled linked list · **XOR linked list**
Trees	B-tree · Binary search tree (AA · AVL · red-black · self-balancing · splay) · Heap (binary · binomial · Fibonacci) · R-tree (R* · R+ · Hilbert) · Trie (Hash tree)
Graphs	Binary decision diagram · Directed acyclic graph · Directed acyclic word graph
	List of data structures

Categories: Binary arithmetic | Linked lists