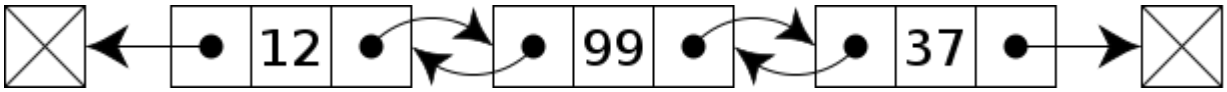
Doubly linked list

In computer science, a **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list. If there is only one sentinel node, then the list is circularly linked via the sentinel node. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.



A doubly linked list whose nodes contain three fields: the link to the previous node, an integer value, and the link to the next node.

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Contents

Nomenclature and implementation

The first and last nodes of a doubly linked list are immediately accessible (i.e., accessible without traversal, and usually called *head* and *tail*) and therefore allow traversal of the list from the beginning or end of the list, respectively: e.g., traversing the list from beginning to end, or from end to beginning, in a search of the list for a node with specific data value. Any node of a doubly linked list, once obtained, can be used to begin a new traversal of the list, in either direction (towards beginning or end), from the given node.

The link fields of a doubly linked list node are often called **next** and **previous** or **forward** and **backward**. The references stored in the link fields are usually implemented as pointers, but (as in any linked data structure) they may also be address offsets or indices into an array where the nodes live.

Basic algorithms

Consider the following basic algorithms written in Ada:

Open doubly linked lists

```
record DoublyLinkedNode {
    next // A reference to the next node
    prev // A reference to the previous node
    data // Data or a reference to data
}
```

```
record DoublyLinkedList {
     DoublyLinkedNode firstNode   // points to first node of list
     DoublyLinkedNode lastNode    // points to last node of list
}
```

Traversing the list

Traversal of a doubly linked list can be in either direction. In fact, the direction of traversal can change many times, if desired. **Traversal** is often called **iteration**, but that choice of terminology is unfortunate, for **iteration** has well-defined semantics (e.g., in mathematics) which are not analogous to **traversal**.

Forwards

```
node  := list.firstNode
while node ≠ null
    <do something with node.data>
    node  := node.next
```

Backwards

```
node  := list.lastNode
while node ≠ null
    <do something with node.data>
    node  := node.prev
```

Inserting a node

These symmetric functions insert a node either after or before a given node:

```
function insertAfter(List list, Node node, Node newNode)
    newNode.prev  := node
```

```
    if node.next == null
        newNode.next  := null -- (not always necessary)
        list.lastNode  := newNode
    else
        newNode.next  := node.next
        node.next.prev  := newNode
    node.next  := newNode
```

```
function insertBefore(List list, Node node, Node newNode)
    newNode.next  := node
    if node.prev == null
        newNode.prev  := null -- (not always necessary)
        list.firstNode  := newNode
    else
        newNode.prev  := node.prev
        node.prev.next  := newNode
    node.prev  := newNode
```

We also need a function to insert a node at the beginning of a possibly empty list:

```
function insertBeginning(List list, Node newNode)
    if list.firstNode == null
        list.firstNode  := newNode
        list.lastNode   := newNode
        newNode.prev  := null
        newNode.next  := null
    else
        insertBefore(list, list.firstNode, newNode)
```

A symmetric function inserts at the end:

```
function insertEnd(List list, Node newNode)
     if list.lastNode == null
         insertBeginning(list, newNode)
     else
         insertAfter(list, list.lastNode, newNode)
```

Removing a node

Removal of a node is easier than insertion, but requires special handling if the node to be removed is the *firstNode* or *lastNode*:

```
function remove(List list, Node node)
    if node.prev == null
        list.firstNode  := node.next
    else
        node.prev.next  := node.next
    if node.next == null
        list.lastNode  := node.prev
    else
        node.next.prev  := node.prev
```

One subtle consequence of the above procedure is that deleting the last node of a list sets both *firstNode* and *lastNode* to *null*, and so it handles removing the last node from a one-element list correctly. Notice that we also don't need separate "removeBefore" or "removeAfter" methods, because in a doubly linked list we can just use "remove(node.prev)" or "remove(node.next)" where these are valid. This also assumes that the node being removed is guaranteed to exist. If the node does not exist in this list, then some error handling would be required.

Circular doubly linked lists

Traversing the list

Assuming that *someNode* is some node in a non-empty list, this code traverses through that list starting with *someNode* (any node will do):

Forwards

```
node  := someNode
do
    do something with node.value
    node  := node.next
while node ≠ someNode
```

Backwards

```
node  := someNode
do
    do something with node.value
    node  := node.prev
while node ≠ someNode
```

Notice the postponing of the test to the end of the loop. This is important for the case where the list contains only the single node *someNode*.

Inserting a node

This simple function inserts a node into a doubly linked circularly linked list after a given element:

```
function insertAfter(Node node, Node newNode)
    newNode.next  := node.next
    newNode.prev  := node
    node.next.prev  := newNode
    node.next       := newNode
```

To do an "insertBefore", we can simply "insertAfter(node.prev, newNode)".

Inserting an element in a possibly empty list requires a special function:

```
function insertEnd(List list, Node node)
    if list.lastNode == null
        node.prev := node
        node.next := node
    else
        insertAfter(list.lastNode, node)
    list.lastNode := node
```

To insert at the beginning we simply "insertAfter(list.lastNode, node)".

Finally, removing a node must deal with the case where the list empties:

```
function remove(List list, Node node);
    if node.next == node
        list.lastNode := null
    else
        node.next.prev := node.prev
        node.prev.next := node.next
        if node == list.lastNode
            list.lastNode := node.prev;
    destroy node
```

Deleting a node

As in doubly linked lists, "removeAfter" and "removeBefore" can be implemented with "remove(list, node.prev)" and "remove(list, node.next)".

Advanced concepts

Asymmetric doubly linked list

An asymmetric doubly linked list is somewhere between the singly linked list and the regular doubly linked list. It shares some features with the singly linked list (single-direction traversal) and others from the doubly linked list (ease of modification)

It is a list where each node's *previous* link points not to the previous node, but to the link to itself. While this makes little difference between nodes (it just points to an offset within the previous node), it changes the head of the list: It allows the first node to modify the *firstNode* link easily.[1][2]

As long as a node is in a list, its *previous* link is never null.

Inserting a node

To insert a node before another, we change the link that pointed to the old node, using the *prev* link; then set the new node's *next* link to point to the old node, and change that node's *prev* link accordingly.

```
function insertBefore(Node node, Node newNode)
    if node.prev == null
        error "The node is not in a list"
    newNode.prev  := node.prev
    atAddress(newNode.prev)  := newNode
    newNode.next  := node
    node.prev = addressOf(newNode.next)
```

```
function insertAfter(Node node, Node newNode)
    newNode.next  := node.next
    if newNode.next != null
        newNode.next.prev = addressOf(newNode.next)
    node.next  := newNode
    newNode.prev  := addressOf(node.next)
```

Deleting a node

To remove a node, we simply modify the link pointed by *prev*, regardless of whether the node was the first one of the list.

```
function remove(Node node)
    atAddress(node.prev)  := node.next
    if node.next != null
        node.next.prev = node.prev
    destroy node
```

See also

- XOR linked list

- SLIP (programming language)

References

1. http://www.codeofhonor.com/blog/avoiding-game-crashes-related-to-linked-lists
2. https://github.com/webcoyote/coho/blob/master/Base/List.h