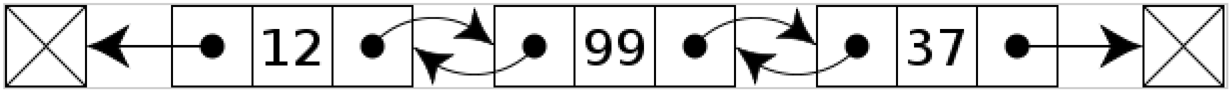
Doubly linked list

From Wikipedia, the free encyclopedia

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 two fields, called *links*, that are references to the previous and to the next node in the sequence of nodes. 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: an integer value, the link to the next node, and the link to the previous 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.

The concept is also the basis for the mnemonic link system memorization technique.

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 {
    prev // A reference to the previous node
    next // A reference to the next 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
         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
         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
```

//NODEPA 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)".

Double linked list implementation

The following program illustrates implementation of double linked list functionality in C programming language.

```
/*
Description:
Double linked list header file
License: GNU GPL v3

*/
#ifndef DOUBLELINKEDLIST_H
#define DOUBLELINKEDLIST_H

/* Codes for various errors */
#define NOERROR 0x0
#define MEMALLOCERROR 0x01
#define LISTEMPTY 0x03
#define NODENOTFOUND 0x4

/* True or false */
#define TRUE 0x1
#define FALSE 0x0

/* Double linked DoubleLinkedList definition */
 typedef struct DoubleLinkedList
{
    int number;
    struct DoubleLinkedList* pPrevious;
    struct DoubleLinkedList* pNext;
}DoubleLinkedList;

/* Get data for each node */
extern DoubleLinkedList* GetNodeData(DoubleLinkedList* pNode);

/*  Add a new node forward */
extern void AddNodeForward(void);

/* Add a new node in the reverse direction */
extern void AddNodeReverse(void);

/* Display nodes in forward direction */
extern void DisplayNodeForward(void);

/*Display nodes in reverse direction */
extern void DisplayNodeReverse(void);


/* Delete nodes in the DoubleLinkedList by searching for a node */
extern void DeleteNode(const int number);

/* Function to detect cycle in a DoubleLinkedList */
extern unsigned int DetectCycleinList(void);

/*Function to reverse nodes */
extern void ReverseNodes(void);

```

```c
/* function to display error message that DoubleLinkedList is empty */
void ErrorMessage(int Error);

/* Sort nodes */
extern void SortNodes(void);

#endif
```

```c
/* Double linked List functions */
/*****************************************************
Name: DoubledLinked.c
version: 0.1
Description:  Implementation of a DoubleLinkedList.
These functions provide functionality of a double linked  List.
Change history:
0.1 Initial version

License: GNU GPL v3

*****************************************************/
#include "DoubleLinkedList.h"
#include "stdlib.h"
#include "stdio.h"

/* Declare pHead */
DoubleLinkedList* pHead = NULL;

/* Variable for storing error status */
unsigned int Error = NOERROR;

DoubleLinkedList* GetNodeData(DoubleLinkedList* pNode)
{
    if(!(pNode))
    {
        Error = MEMALLOCERROR;
        return NULL;
    }
   else
   {
    printf("\nEnter a number: ");
    scanf("%d",&pNode->number);
    return pNode;
   }
}

/* Add a node forward */
void AddNodeForward(void)
{
    DoubleLinkedList* pNode = malloc(sizeof(DoubleLinkedList));
    pNode = GetNodeData(pNode);
    if(pNode)
    {
    DoubleLinkedList* pCurrent = pHead;
    if (pHead== NULL)
    {
        pNode->pNext= NULL;
        pNode->pPrevious= NULL;
        pHead=pNode;
    }
    else
    {
      while(pCurrent->pNext!=NULL)
      {
        pCurrent=pCurrent->pNext;
      }
      pCurrent->pNext= pNode;
      pNode->pNext= NULL;
      pNode->pPrevious= pCurrent;
    }
    }
    else
    {
        Error = MEMALLOCERROR;
    }

}

/* Function to add nodes in reverse direction,
Arguments; Node to be added.
Returns : Nothing
*/
void AddNodeReverse(void)
{
    DoubleLinkedList* pNode =  malloc(sizeof(DoubleLinkedList));
    pNode = GetNodeData(pNode);
    if(pNode)
    {
    DoubleLinkedList* pCurrent =  pHead;
    if (pHead==NULL)
    {
        pNode->pPrevious= NULL;
        pNode->pNext= NULL;
        pHead=pNode;
    }
    else
    {
     while(pCurrent->pPrevious != NULL )
     {
        pCurrent=pCurrent->pPrevious;
     }
    pNode->pPrevious= NULL;
    pNode->pNext= pCurrent;
    pCurrent->pPrevious= pNode;
    pHead=pNode;
    }
    }
    else
    {
        Error = MEMALLOCERROR;
    }


}

/* Display Double linked list data in forward direction */
void DisplayNodeForward(void)
{
    DoubleLinkedList* pCurrent =  pHead;
    if (pCurrent)
    {
     while(pCurrent != NULL )
      {
            printf("\nNumber in forward direction is %d ",pCurrent->number);
            pCurrent=pCurrent->pNext;
      }
    }
    else
    {
          Error = LISTEMPTY;
          ErrorMessage(Error);
    }
}

/* Display Double linked list data in Reverse direction  */
void DisplayNodeReverse(void)
{
    DoubleLinkedList* pCurrent =  pHead;
    if (pCurrent)
    {
      while(pCurrent->pNext != NULL)
      {
        pCurrent=pCurrent->pNext;
      }
      while(pCurrent)
      {
```

```c
        printf("\nNumber in Reverse direction is %d ",pCurrent->number);
        pCurrent=pCurrent->pPrevious;
      }
    }
    else
    {
       Error = LISTEMPTY;
       ErrorMessage(Error);
    }

}

/* Delete nodes in a double linked List */
void DeleteNode(const int SearchNumber)
{
    unsigned int Nodefound = FALSE;
    DoubleLinkedList* pCurrent =  pHead;
    if (pCurrent != NULL)
    {
    DoubleLinkedList* pNextNode =  pCurrent->pNext;
    DoubleLinkedList* pTemp = (DoubleLinkedList* ) NULL;

    if (pNextNode != NULL)
    {
    while((pNextNode != NULL) && (Nodefound==FALSE))
    {
      // If search entry is at the beginning
      if(pHead->number== SearchNumber)
       {
        pCurrent=pHead->pNext;
        pHead= pCurrent;
        pHead->pPrevious= NULL;
        Nodefound =TRUE;
       }
       /* if the search entry is somewhere in the DoubleLinkedList or at the end */
      else if(pNextNode->number == SearchNumber)
        {
           Nodefound = TRUE;
           pTemp = pNextNode->pNext;
           pCurrent->pNext = pTemp;

            /* if the node to be deleted is not NULL,,,
            then point pNextnode->pNext to the previous node
            which is pCurrent */
           if(pTemp)
           {
               pTemp->pPrevious= pCurrent;
           }
          free(pNextNode);
        }
      /* iterate through the Double Linked List until next node is NULL  */
      pNextNode=pNextNode->pNext;
      pCurrent=pCurrent->pNext;
    }

    }
    else if (pCurrent->number == SearchNumber)
    {
        /* add code to delete nodes allocated with other functions if
         the search entry is found.
         */
        Nodefound = TRUE;
        free(pCurrent);
        pCurrent= NULL;
        pHead = pCurrent;
    }

    }
    else if (pCurrent == NULL)
    {
        Error= LISTEMPTY;
        ErrorMessage(Error);
    }
    if (Nodefound == FALSE && pCurrent!= NULL)
    {
       Error = NODENOTFOUND;
       ErrorMessage(Error);
    }

}

/* Function to detect cycle in double linked List */
unsigned int DetectCycleinList(void)
{
    DoubleLinkedList* pCurrent = pHead;
    DoubleLinkedList* pFast = pCurrent;
    unsigned int cycle = FALSE;
    while( (cycle==FALSE) && pCurrent->pNext != NULL)
    {
        if(!(pFast = pFast->pNext))
        {
        cycle= FALSE;
        break;
        }
        else if (pFast == pCurrent)
        {
        cycle = TRUE;
         break;
        }
        else if (!(pFast = pFast->pNext))
        {
           cycle = FALSE;
           break;

        }
        else if(pFast == pCurrent)
        {
            cycle = TRUE;
            break;

        }
        pCurrent=pCurrent->pNext;
    }
    if(cycle)
    {
        printf("\nDouble Linked list is cyclic");
    }
    else
    {
        Error=LISTEMPTY;
        ErrorMessage(Error);
    }
    return cycle;
}

/*Function to reverse nodes in a double linked list */
void ReverseNodes(void)
{
 DoubleLinkedList *pCurrent= NULL, *pNextNode= NULL;
 pCurrent = pHead;
 if (pCurrent)
 {
  pHead =NULL;
  while (pCurrent != NULL)
   {
     pNextNode = pCurrent->pNext;
     pCurrent->pNext = pHead;
     pCurrent->pPrevious=pNextNode;
     pHead = pCurrent;
     pCurrent = pNextNode;
   }
 }
 else
 {
     Error= LISTEMPTY;
     ErrorMessage(Error);
 }
}

/* Function to display diagnostic errors */
```

```
void ErrorMessage(int Error)
 {
     switch(Error)
     {
         case LISTEMPTY:
         printf("\nError: Double linked list is empty!");
         break;

         case MEMALLOCERROR:
         printf("\nMemory allocation error ");
         break;

         case NODENOTFOUND:
         printf("\nThe searched node is not found ");
         break;

         default:
         printf("\nError code missing\n");
         break;
     }
 }
```

```
/* main.h header file */
#ifndef MAIN_H
#define MAIN_H

#include "DoubleLinkedList.h"

/* Error code */
extern unsigned int Error;

#endif
```

```
/****************************************************/
/****************************************************
Name: main.c
version: 0.1
Description:  Implementation of a double linked list

Change history:
0.1 Initial version
License: GNU GPL v3
****************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "main.h"

int main(void)
{
    int choice =0;
    int InputNumber=0;
    printf("\nThis program creates a double linked list");
    printf("\nYou can add nodes in forward and reverse directions");
    do
    {
        printf("\n1.Create Node Forward");
        printf("\n2.Create Node Reverse");
        printf("\n3.Delete Node");
        printf("\n4.Display Nodes in forward direction");
        printf("\n5.Display Nodes in reverse direction");
        printf("\n6.Reverse nodes");
        printf("\n7.Exit\n");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            AddNodeForward();
            break;

            case 2:
            AddNodeReverse();
            break;

            case 3:
            printf("\nEnter the node you want to delete: ");
            scanf("%d",&InputNumber);
            DeleteNode(InputNumber);
            break;

            case 4:
            printf("\nDisplaying node data in forward direction \n");
            DisplayNodeForward();
            break;

            case 5:
            printf("\nDisplaying node data in reverse direction\n");
            DisplayNodeReverse();
            break;

            case 6:
            ReverseNodes();
            break;

            case 7:
            printf("Exiting program");
            break;

            default:
            printf("\nIncorrect choice\n");
        }

    } while (choice !=7);
    return 0;
}
```

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