Implementation of Enhanced Singly Linked List Equipped with DLL Operations: An Approach towards Enormous Memory Saving

Devishree Naidu and Abhishek Prasad Jr., *Member, IACSIT*

***Abstract*—A linked list is a complex data structure, especially useful in systems or application programming. Linked list uses dynamic allocation of memory by which it allocates memory at the runtime alike of array so it is also called as dynamic data structure. As dynamic list is used for runtime allocation of memory, This feature is very useful in making operating system to keep track of running process that are alive or in sleep mode. This requires either a node to be inserted in between or to be removed. On other side doubly linked list that helps in moving forward or backward mode. We Proposed singly linked list itself to implement operation of doubly linked list as well as singly linked list. Our approach makes use of EX-OR Technique to acquire next node and previous node, To traverse in both the direction but the problem with doubly linked list is that, it requires large amount of memory to store addresses and therefore singly linked list can perform same task of traversing in both direction without increasing its memory requirement. This concept can be usefully applicable to the system or application programming where memory requirement is more viz MRU list , cache in Browser that allow to hit back.**

***Index Terms*—SLL, DLL, traversing, MRU, exclusive-OR**

I. Introduction

Exclusive OR is one of the operations which are performed between two binary numbers. It is also called XOR and it defined as

$$A \text{ XOR } B = \begin{cases} \text{If A and B are not equal } 0 \\ 0 \text{ otherwise} \end{cases}$$

Assume A=1010 and B=1110 then A XOR B the according to above rule A XOR B will be 0100. If we have (A XOR B) and B then we can get A in the equation and if we have (A XOR B) and A then we can get B by using the given formula.

$$A = B \text{ XOR } (A \text{ XOR } B) \quad \text{and}$$

$$B = A \text{ XOR } (A \text{ XOR } B)$$

Consider the list given in the "Fig. 1.a" each node contains the address of the next node in the list.

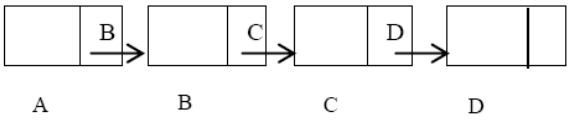

Fig. 1.a Simple list.

Assume that the base addresses of the nodes be A, B, C and D. Then the pointers of each node will contain the address of the next node of the list. Here we change the pointer's value of each node by the XOR of base addresses of its previous and next node. In the case of corner nodes ie first and last node there is no previous and next node so there we assume the previous as NULL and next as NULL [1].

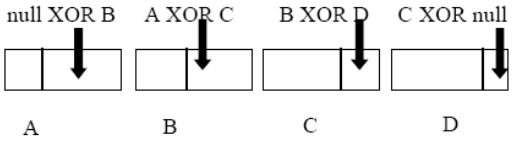

Fig. 1.b Enhanced SLL.

In "Fig. 1. b" a representation of linked list nodes with changed pointer values is shown. For the forward traversal of the list we XOR the pointer's value of the first node with NULL then it will give the address of the second node in the list. Again we XOR the address of the first node with the pointer's value of the second node it will give the address of the third node of the list. In this manner we can traverse the whole list in forward direction. Similarly for the backward traversal of list we XOR the pointer's value of the last node with the NULL it will give the address of the second last node of the list. Again we XOR the base address of the last node with the pointers of the second last node it will give the address of the third last node of the list [2], [3]. In this manner we can traverse the list in backward direction.

II. Operations

By doing this type of modification in the pointer's value of the each node we can perform all the operations of DLL on the SLL. All the basic operations such as forward traversal, backward traversal, insertion and deletion can be performed on it.

A. Forward Traversal Algorithm

We can traverse the list in forward direction by using the following algorithm.

- We have first and last pointers pointing to first and last node of the list.

Manuscript received June 15, 2013; revised November 5, 2013. Implementation Of Enhanced Singly Linked List Equipped With DLL Operations: An Approach Towards Enormous Memory Saving

Devishree Naidu is with Computer science and Engineering Department, Rashtrasant Tukdoji Maharaj Nagpur University, India (e-mail: devishree. naidu@ gmail.com).

Abhishek Prasad is with Wireless Research and Development, Marvell Technology Pvt. Ltd. Pune, India (e-mail: Prasad.abhishek86@gmail.com).

DOI: 10.7763/IJFCC.2014.V3.276

- When we XOR the base address of last node with the link of first node it will give the address of second node.

Now we have pointers which are pointing to the first node and the second node of the list. Now again by XOR we get the address of third node [4].

In this way we can traverse the whole list in forward direction.

We have developed a program which is in C language using Turbo C++ compiler which demonstrate the working of this concept on the machine. "Fig. 2. a" is the snapshot of our program which represents the traversal of singly linked list in forward direction.

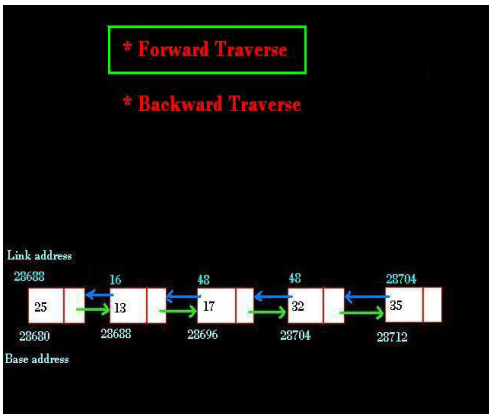

Fig. 2. a. Forward traversing.

B. Backward Traversal Algorithm

We can traverse the list in backward direction by using following algorithm.

- We have first and last pointers pointing to first and last node of the list.
- When we XOR the base address of first node with link of last node it will give the base address of second last node.
- Now we have pointers which are pointing to the last node and second last node of the list. Now again by XOR we get the address of third last node [4], [5].

In this way we can traverse the whole list in backward direction. One of the major beauties of this algorithm is, "there is only one function for the traversal of the list which can traverse the list in both the forward and backward direction". If we pass the address of the first node to the traverse function then it will traverse the list in forward direction and when we pass the address of the last node then it will traverse the list in backward direction.

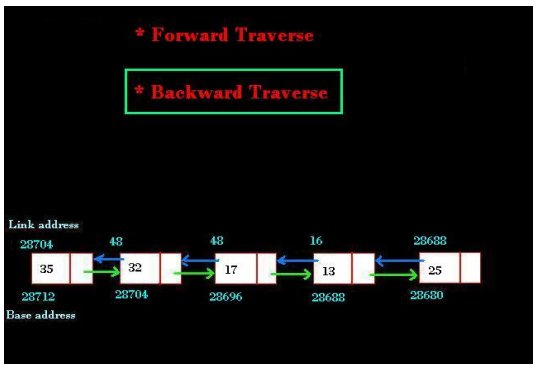

Fig. 2. b. Backward traversing.

The pseudo code for the traversing of list is as follows.

```
void traverse ( struct node * KEY )
    {
        struct node *temp;
        prev = NULL ;
       curr = KEY ;
       while( curr )
  {
     printf( curr - > data );
     temp = curr ;
     curr = ( curr -> link  ) XOR prev ;
     prev = temp;
         }
     }
```

If you call the above module with KEY equal to address of the first node then the whole list will be traversed in forward direction, and if module is called with KEY equal to address of the last node then the whole list will be traversed in backward direction.

C. Insertion Algorithm

We can insert a new node anywhere in the list without disturbing the links of the existing nodes. Only we have to change the links of the nodes which are previous and next with respect to the newly inserted node [6].

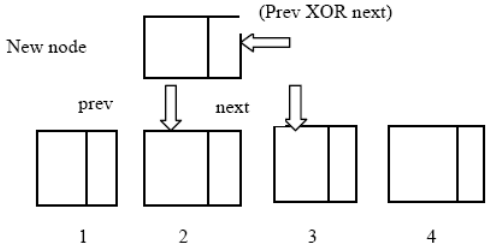

Fig. 3. a. Insertion of new node.

Suppose we have to insert a new node between node-2 and node-3 as shown in "Fig. 3. a" We use the following algorithm to insert a new node.

- We set the pointers prev and next which points to the node-2 and node-3.
- We store (prev XOR next) in the link part of new node.
- We change the link of node-2(prev) by the XOR of base address of node-1 (node previous to prev pointer) and new node.
- In the same way we change the link of node-3 by XOR of base address of node-4 (node next to next pointer) and new node.
- If we have to insert the node in beginning then we set prev=last and next=first and perform step-2, step-3 and step-4.
- If we have to insert the node in end then we set prev=last and next=first and perform step-2 ,step-3 , step-4.

In this way we can insert a new node anywhere in the list by using insertion module.

```
void insert_node ( int  position )
{
```

- Traverse the list and go to given position using traverse module.

```
    traverse ( KEY , position ) ;
```

- Get the new node.
  new = create_node (data) ;

new = prev XOR curr ;
prev - > link = ((prev -> link) XOR next) XOR new ;
curr - > link = ((next -> link) XOR prev) XOR new ;

}

D. Deletion Algorithm

We can delete any node of the linked list without disturbing the links of existing nodes. Only we have to change the links of previous and next node [7]. Consider the list given below and we have to delete the node-3.

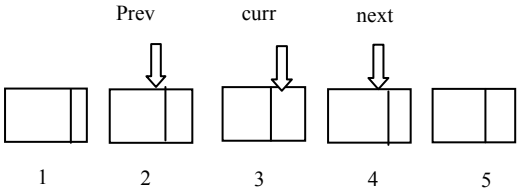

Fig. 3. b. Deletion of a Node.

As shown in "Fig. 3.b" we set three pointer prev, next, and curr which are pointing to previous node, next node, and the node which we have to be delete.

We use the following algorithm to delete a node from the linked list.

We change the link of node-2(prev) by XOR of base address of node-1(node previous to prev pointer) and node-4(next).

- We change the link of node-4(next) by XOR of base address of node-2(prev) and node-5(node next to next pointer).
- Free the pointer curr (node which we have to delete).
- In the case when we have to delete the first node then we take prev=(address of last node) and next= (address of second node) curr=(address of first node) and perform step-1 , step-2 and step-3.

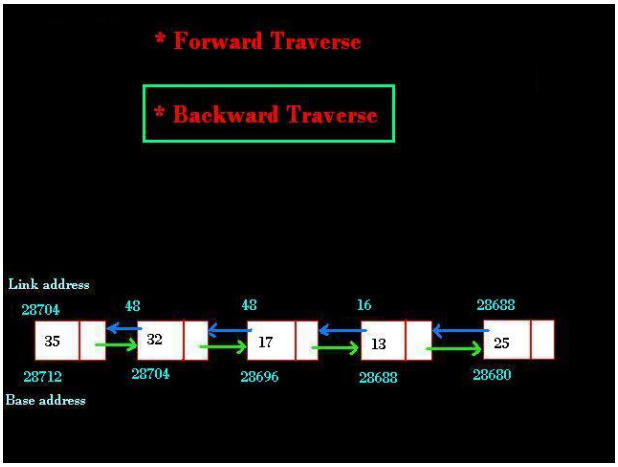

Fig. 3. c. Backward traversal.

- In the case when we have to delete the last node then we take prev=(address of second last node) , next=(address of first node) curr=(address of last node) and perform the step-1 ,step-2 and step-3.

We will use the following module to delete any node from our list. Here we will make use of traverse module in order to reach to the particular node that we want to delete.

void delete_node(int position)
{

- Traverse the list and go to given position using traverse module.
- traverse (KEY , position) ;

prev -> link = ((prev -> link) XOR curr) XOR next ;
next -> link = ((next -> link) XOR curr) XOR prev;

}

III. Simulation on Mobile Phone

This type of linked list is applied where there is a memory constrain. We can apply this list in many handheld devices where there is a serious memory constraint. Now a day's Mobile phone is one of the most common handheld devices. We can apply our concept for storing the contact information in mobile phones. In mobile phones there is a phonebook which contains the numbers of contacts which are stored in doubly linked list. These contacts are sorted on contact name. Assume the phonebook of a phone can store three information i.e. First names, Last name and contact number.

"Fig. 4. a" represents the phonebook of a mobile phone which contains the contact name and phone numbers of different persons. If we implement the phonebook through the double linked list then the graph between the numbers of contacts and memory requirement are given as in "Fig. 4.b" when we use the doubly linked list then we have to store the addresses of both the forward and backward nodes of the list hence more memory is required [8].

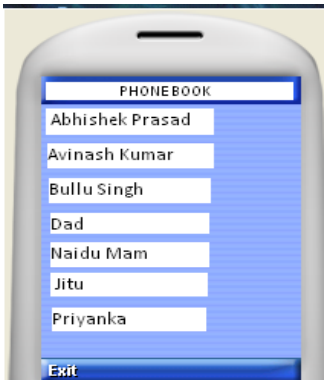

Fig. 4. a. DLL. operation on Phonebook.

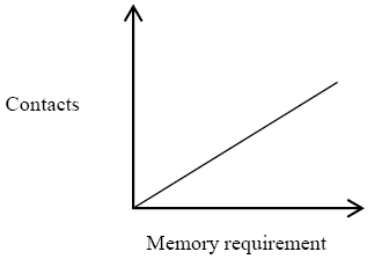

Fig. 4. b. Memory requirement for storing contacts.

When we use the enhanced linked list then the memory is being saved by considerable amount and the memory and contact graph results as in "Fig. 4.c".

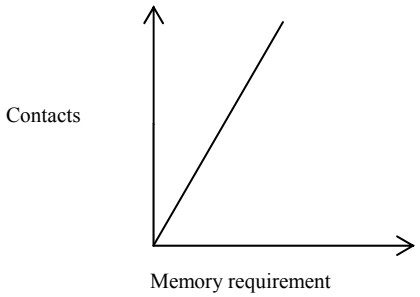

Fig. 4. c. Memory saving graph with S. L. L.

Hence we can save considerable amount of memory by using enhanced linked list [9], [10] and we can use the precious memory of handheld system in some useful work.

IV. Summary

Linked list is a dynamic data structure which is used to dynamically allocate the memory. It allocates the memory during the life time of the program. Linked list are of basically of two types called singly linked list and doubly linked list. Singly linked list can be traversed in only one direction. Unidirectional traversal is the drawback of the singly linked list which can be overcome by the doubly linked list. Doubly linked list stores the addresses of both the previous and next node so the traversals in both the directions are possible. We introduced a new concept of EX- OR in the singly linked list and make it possible to traverse in both the directions without increasing the memory requirement. Hence we can replace the doubly linked list by this enhanced singly linked list. This enhanced singly linked list is very useful in domains where higher priority is given to the memory. We showed a simulation with 'C' program and also with handheld device mobile phone. With this we can conclude that by saving a large amount of memory we are able to perform all the DLL operations on "SLL. itself" - removing its drawback. This makes a great impact on the use of resources that requires more memory for their operation and we are at short of limited memory so that it can be utilized further in performing some more useful task. This particular technique can also be very useful in operating system keeping track of processes for their efficient performance.

References

[1] W. Li, S. Mohanty, and K. Kavi. "A page-based hybrid (software hardware) dynamic memory allocator," *IEEE Computer Architecture Letters*, 2006.

[2] J. Meng and L. C. Paulson, "Translation higher-order problems to first-order clauses," *Successful Comp. Reasoning*, pp. 70-80,2006

[3] A. Rogers, M. C. Carlisle, J. H. R. Laboratories, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, pp. 233-263, 1995.

[4] W. Gloger. (2009). Dynamic Memory Allocator implementations in Linux System Libraries. [Online]. Available: http://www.dent.med.unimuenchen.de/wmglo/malloslides.html

[5] M. Barnett, R. D. Line, M. Fahndrich, and K. R. M. Leino, "Verification of object-oriented programs with invariants," *Journal of Object Technology*, vol. 3, no. 6, pp. 27-56, 2004

[6] H. Sahni and A. Freed, *Fundamentals of Data Structures in C* 2nd edition, ch. 4, pp 190-195.

[7] H. J. Boehm, "The Space Cost of Lazy Reference counting," in *Proc. 31st ACM Sigplan-Sigact Symp. Principles of Programming Languages*, 2004, pp. 210-219.

[8] M. B. Zorn. (2001). Debugging Tools for Dynamic Storage Allocation and Memory management. [Online]. Available: www.cs.colorado.edu/homes/zom/public_html/MallocDebug.html

[9] S. Kumar. "A Language for Programmable Devices. Technical report, Princeton University," Department of Computer Science, January 2002.

Devishree Naidu recevied her B. E in computer science. She recevied M. E specialization in wireless communication and computing from G. H. Raisoni College of Engineering, Rashtrasant Tukdoji Maharaj University (University of Nagpur) India in year 2009.

She has around Seven years of Teaching Experience, Currently working as a Assistant Professor at Department of Computer Science and Engineering, Shri Ramdeobaba College of Engineering and Management, Nagpur, INDIA. She has guided undergraduate and post graduate student for research project work in the field of mobile computing, and wireless sensor network. She is currently carrying her research work in sensor networks. Her previous research work was in Header compression Related to TCP/IP Protocol.