Vol. 4, No. 6 June 2013 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

The S-Linked List – A Variant Of The Linked List Data Structure

[1] Akinde Aderonke O., [2] Okolie Samuel O., [3] Kuyoro 'Shade O.
[1,2,3] Department Of Computer Science, Babcock University, Ilishan Remo, Ogun State, Nigeria.

ABSTRACT

Using doubly linked list in embedded systems can be expensive. Although doubly linked is efficient in terms of time complexity and highly flexible in itself, enabling easy traversal and update, but it often demands more memory space compared to other list data strucure – this is as a result of the space required for the extra pointers, doubling the amount needed for a singly linked list. In this paper, we introduce the S-linked list – a hybrid of the concept of the singly linked list and the circular linked list. The hybrid gives a data structure that is similar to the unrolled linked list, but rather than have an array in each node, we have a singly linked list. An analysis of the space complexity and assymptotic time complexity of the algorithm was carried out.

Keywords: *Linked lists, Algorithm, Time Complexity, Space Complexity*

1. INTRODUCTION

Embedded systems for the consumer market such as electronic toys, small mobile devices, amongst others need to be as cost-effective as possible [1]. Even if one-quarter of a dollar is saved, the fact that millions of such systems will be sold to the public is a great deal of cost saving for the company in question. Although most data structure algorithms are concerned about time complexity and efficiency. It is research-worthy to consider the possibility of maximizing the usage of the inexpensive but limited memory resource in embedded systems [2]. Generally, embedded systems often have limited resources - especially memory resource - and they do not enjoy the privilege of having secondary storage devices such as the CDROM found in PCs. Mobile devices such as mobile phones, Personal Digital Assistants (PDAs), smart phones etc. are a special category of embedded systems. The limitations of the mobile devices as well as some other embedded systems includes memory constraints, small size, amongst others[3].

As a result of the memory constraint, embedded system programming requires the use of dynamic memory allocation. When it comes to dynamic memory allocation, one of the preferred data structure is the linked lists data structure. A linked list data structure is known to be one which may change during execution since successive elements are connected by pointers. Linked lists are often associated with dynamic allocation of memory, but the use of linked lists in embedded systems may be very expensive.[3] This is as a result of the memory space required for keeping the pointers/reference to nodes in the list.

2. PROBLEM STATEMENT

In embedded systems, the use of doubly linked lists is very expensive. The doubly linked is highly flexible and efficient, but memory-demanding. The demand for memory is as a result of the space required for the extra pointers (which is double the amount needed for a singly linked list). [4]

A solution to the setback of doubly linked lists in embedded programming, is the use of unrolled linked lists. In an unrolled linked list, we have a doubly linked list with fewer nodes, where each node holds an array of data. However, this generally uses contiguous memory to hold nodes (because arrays make use of contiguous memory locations) so it becomes difficult to move nodes that are in the array. The doubly linked list is the easiest to traverse and update. The doubly linked list is also the most efficient in terms of time complexity but it is not cost-effective in terms of space complexity. [5]

3. METHODOLOGY

This paper proposes and introduces the S-linked list, which is a hybrid of the concept of the singly linked list and the circular linked list. The product of the amalgamation gives a data structure that looks somewhat like the unrolled linked list, but rather than have an array in each node, we have a singly linked list. An analysis of the space complexity and assymptotic time complexity of the algorithm was carried out.

4. AN OVERVIEW OF LINKED LIST DATA STRUCTURE

The linked list is often used as the basis for other containers including queue and stack containers [4]. The advantages of using a linked list is that: it does not waste unnecessary memory space as elements can be added to the list 'on the go'; it can be created just for the period of time it is needed; and it can grow or shrink during program execution because of its flexibility in efficiently inserting and deleting elements into the list. The linked list has a key advantage over the array data structure; data items need not be stored contiguously in memory or on disk and as such the elements of a linked list can be inserted or removed easily without reallocation or reorganization of the entire structure[4].

The linked list data structure has several variations, which include the singly linked list, circular linked list, doubly linked list, etc. The singly linked list is the least flexible form of linked list as it only allows a uni-directional traversal of the list, since each node only holds a link or reference to the next node. On the other hand, the doubly linked list is known to be the most efficient (i.e. in terms of time complexity) when it comes to basic traversal, insertion, deletion operations. The nodes of a doubly-linked list can be traversed in both directions -

Vol. 4, No. 6 June 2013 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

forward and backward traversal - but methods that alter doubly-linked lists often require twice as much overhead; they occupy 50% more space than simple linear linked lists [6]. The circular linked-list is basically a hybrid of the singly and doubly-linked list, where the end node iterates forward to the beginning node and sometimes vice versa [7]. But, the circular linked list is not commonly used because of the problems with iterating through the list, although special list implementations can handle circular linked lists better than others. The unrolled linked list is a hybrid of the doubly linked list and an array. Each node of an unrolled linked list is doubly linked, but each node contains an array. To traverse an unrolled linked list, we start with the first node, linearly iterate through the data in the node's array, and then move on to the next node to continue the same process. This way, we have fewer nodes of data in one while reducing the amount of nodes altogether (therefore saving memory).

5. THE PROPOSED S-LINKED LIST

As aforementioned, the S-Linked introduced in this paper is a hybrid of the circular linked list and the singly linked list.

5.1 Creating the S-Linked List

Number of nodes = n
Skip factor = k
Therefore,
The resultant number of circular linked lists ≈ n/k

A singly linked list is initially created, where the head node points to the next node, and every other node points to the next node, while the last node point to NULL. (See Figure 1 below):

Where: number of node, n = 10

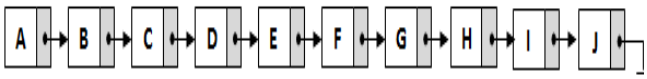


Fig 1: Initial singly linked

Starting from the last node, a backward pointer is used to point to a skip-node using the skip factor, k. A backward skip to a skip-node results into a circular linked list. (see Figure 2 below).

Where: skip factor, k = 3

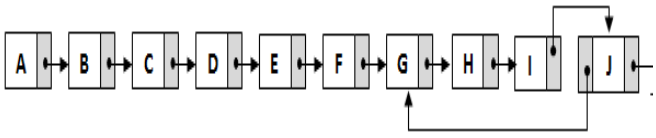


Fig 2: Backward skipping of the list

The skip-node then points to the next skip-node (see Figure 3). At the end of the loop that backwardly points to a skip node, we would have created n/k inherent circular list.

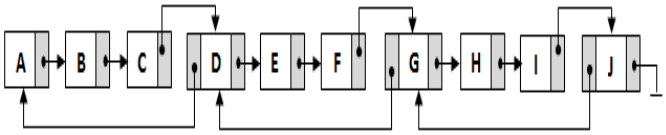


Fig 3: S-Linked list with ≈n/k ≈3 inherent circular lists

Number of nodes, n=10, Skip factor, k = 3.

Therefore, number of inherent circular list, n/k ≈ 10/3 ≈ 3 approximately. There are 3 inherent circular lists.

5.2 Deleting a Node from the S-Linked List

There are various instances of the delete operation, they are described below:

5.2.1 Deleting Head Node, Delete Head Node ( ):

a. Make backward pointer of the last node in the inherent circular list point to the next node after the head node. With this, the next node after the head node now becomes the head
b. Eliminate the next pointer of the deleted node

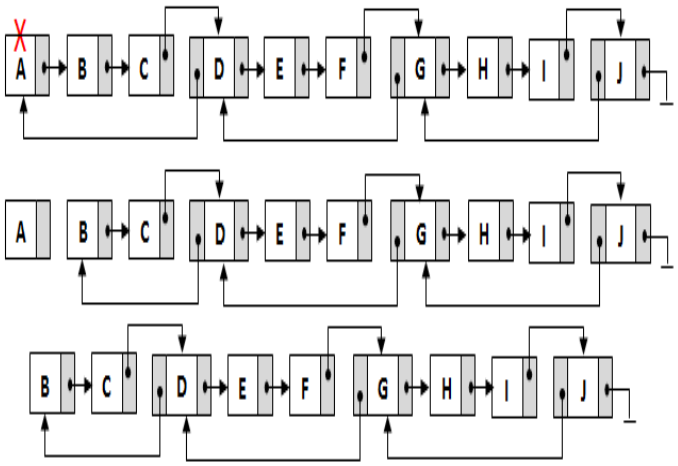


Fig 4: Deleting head node

5.2.2 Deleting Tail Node, Delete Tail Node ( ):

a. Create backward pointer from the node before the tail to point to the head node of the inherent circular list.
The prior node now becomes the tail node of the inherent circular list. (Points to null)
b. Eliminate the backward pointer of the previous tail node.

Vol. 4, No. 6 June 2013 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org

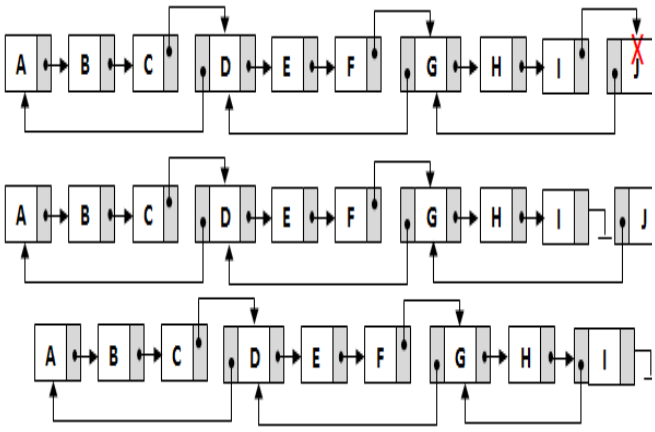


Fig 5: Deleting the tail node

5.2.3 Deleting a node that is not Head or Tail, Delete Node ( ):

a. Change the pointer of the previous node,
b. Make it (the previous node) point to the node after the node to be deleted. (See Figure 6)

To delete node E from the list,

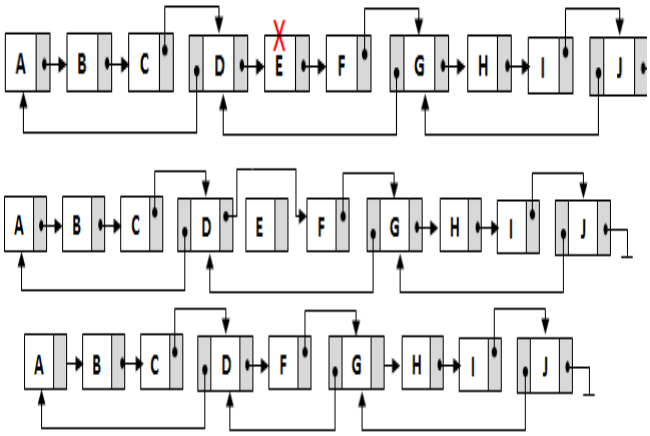


Fig 6: Deleting a node that is not head or tail,

5.3 Adding to the List

There are various instance of the add operation, they are described below:

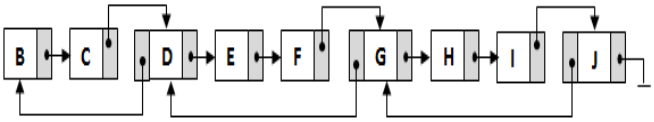
5.3.1 Adding Prior to the Head Node, Add Head Node ( ):

a. If number of nodes in the inherent circular list is less than k+1, (see Figure 7)

i. Make new node point to head node
ii. Make backward pointer from inherent circular list's tail node point to new node
iii. Eliminate previous backward pointer
iv. New node now becomes the head node

b. Else, (see Figure 8)

i. Make new node point to head node
ii. Create backward pointer from head node of the inherent circular list that points to new node (with this, a new inherent circular list is created)



To add the node A before the head node in the list above,

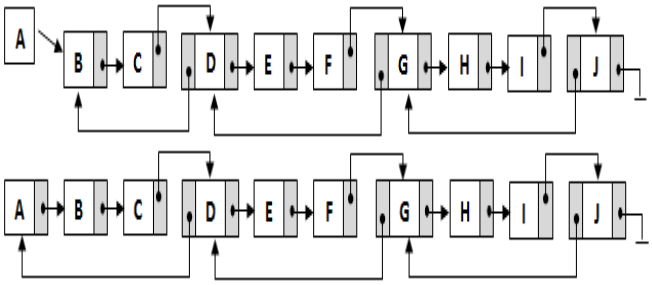


Fig 7: Adding prior to the head node – Scenario 1

To add a node Z before the head node,

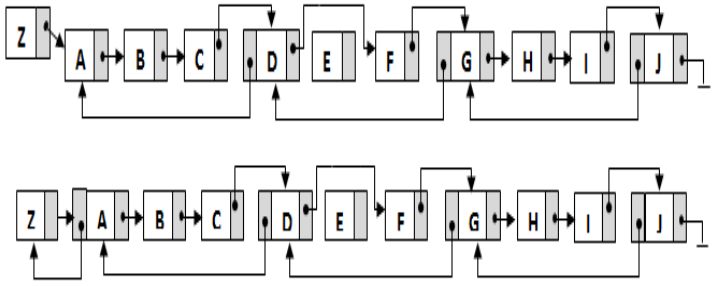


Fig 8: Adding prior to the head node – Scenario 2

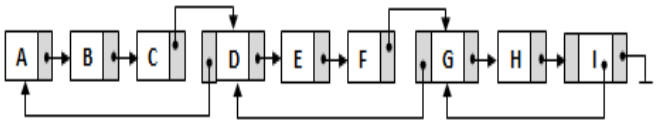
5.3.2 Adding after the Tail Node, Add Tail Node ( ):

a. If number of nodes in the inherent circular list is less than k+1, (see Figure 9)
i. Make last node in the inherent circular list to point to new node
ii. Create backward pointer from the new node to point to head node of the inherent circular list
iii. Eliminate previous backward pointer
iv. Make new node point to NULL

b. Else, (see Figure 10)
i. Make last node in the inherent circular list to point to new node
ii. Create a backward pointer from the new node to itself. (with this, a new inherent circular list having one is created)

Vol. 4, No. 6 June 2013 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

http://www.cisjournal.org



To insert the node J after the tail node in the list above,

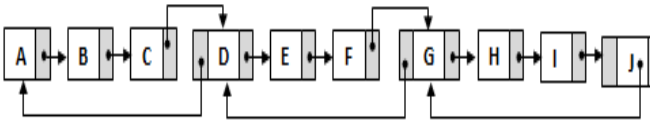
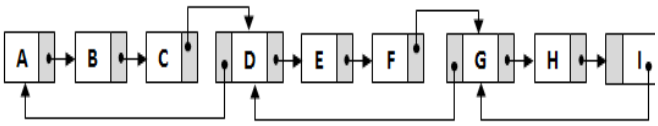


Fig 9: Adding after the tail node – Scenario 1



To insert node K after the tail node in the list above

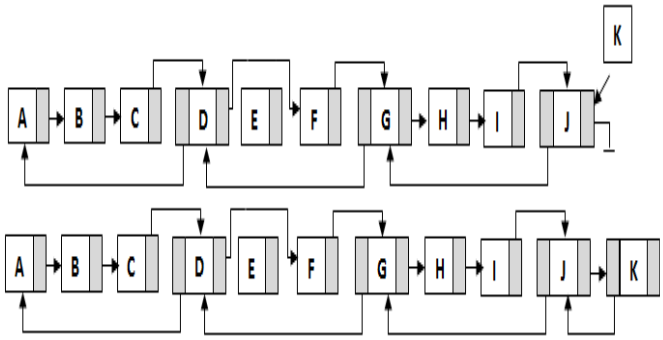


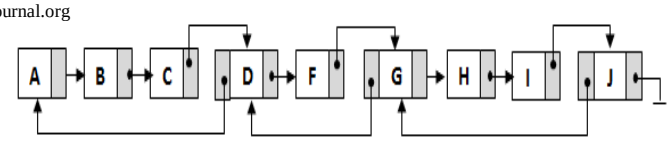
Fig 10: Adding after the tail node – Scenario 2

5.3.3 Adding a Node, Add Node ( ):

a. If the number of nodes in the inherent circular list is less than (k+1):
   i. Make new node point to the next node and
   ii. Make previous node point to the new node

b. Else if the number of nodes in the previous circular list is less than (k+1)
   i. Make the current tail node point to the new node
   ii. Create a backward pointer from the new node to the head node of the circular list. (i.e. new node becomes the tail node of the circular list)

c. Else,
   i. Make new node point to the next node of the inherent circular node
   ii. Create a backward pointer from the tail node of the circular node to the next node after the head node of the circular list (with this the node becomes the new head node of the inherent circular list).
   iii. Create a backward pointer from the head node to itself (with this we have created a new inherent circular list with only one node – the head node). (see Figure 11)



To add node E to the list above,

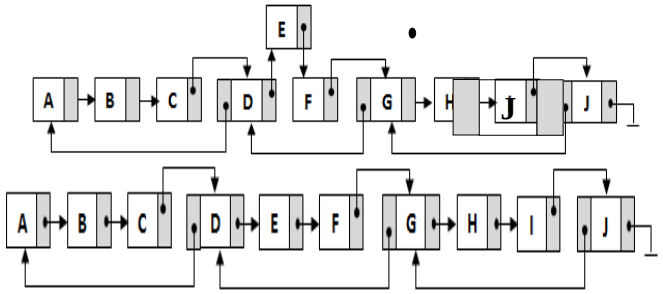
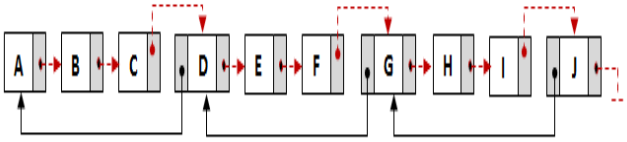


Fig 11: Adding a node

5.4 Traversing the List

The forward traversal of the S-Linked list is the same as that if the regular singly linked list. However, the backward traversal occurs in a different mode. To carry out a backward traversal in an S-Linked list, the tail node of the current inherent circular list must be reached. (see Figure 12 – follow dashed lines)

Forward Traversal



Backward Traversal

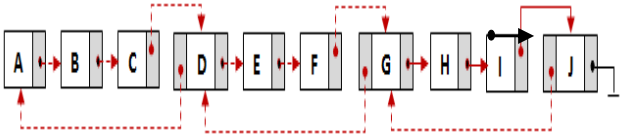


Fig 12: Forward and Backward traversal of the S-Linked list

5.5 Complexity Analysis of the S-Linked List

The time complexity analysis of the S-Linked list is stated in Table 1a below, a comparison of the time complexity of the S-Linked list is carried out against other list data structures – circularly, doubly and singly linked list – shown in Table 1b. Table 2 shows the details of the space complexity of the S-Linked list alongside that of the circularly, doubly and singly linked list:

5.5.1 Time Complexity of the S-Linked List

In the S-Linked list, deletion and addition of head and tail nodes take place in constant time (best case), while deletion and addition of nodes that are not tail or head takes place in logarithmic time (worst case). When a linear search is required, it may take place in constant time (best case scenario, for example when the sought node is the first node, or when the pointer address is known) or in linear time (worst case scenario, e.g. when the search requires that every node pointer is checked before the node is found). A forward traversal of the S-

Linked list is in linear time, while backward traversal is in linearithmic time.

Table 1a: Time complexity of operations on the S-Linked list

Operation	Time Complexity
Delete / Add first node	O(1)
Delete / Add tail node	O(1)
Delete / Add node	$O(\log_k n)$
Linear Search - best case	O(1)
Linear Search - worst case	O(n)
Forward Traversal	O(n)
Backward Traversal	$O(n\log_k n)$

Table 1b: Comparison of S-Linked time complexity with other list data structure

Table 1b: Comparison of S-Linked time complexity with other list data structure

Operation	Circular Linked List	Doubly Linked List	S-Linked List	Singly Linked List
Delete/Add first node	O(1)		O(1)	O(1)
Delete/Add tail node	O(n)		O(1)	O(n)
Delete/Add node	O(n)		$O(log_k n)$	O(n)
Linear Search best case	O(1)		O(1)	O(1)
Linear Search worst case	O(n)		O(n)	O(n)
Forward Traversal	O(n+1)	O(n)	O(n)	O(n)
Backward Traversal	O(2n)	O(n)	$O(nlog_k n)$	N/A[3]

5.5.2 Space Complexity of S-Linked List in Comparison with other Linked List Forms

A comparison of the space required for referencing nodes in given in Table 1 below, where n represents one word size.

Table 2: Comparison of space complexity of the S-Linked list with other linked list

List Type	Space Complexity
Singly Linked List	2n
Circular Linked List	2n
Doubly Linked List	3n-1
S-Linked List	2n + n/k

From the space complexity analysis shown in Table 2 above, we see a difference in the space complexities of the different forms of list data structure. It is clear that the doubly linked list has the highest space requirement especially in case where n is large. The space complexity of the S-Linked list is close to that of the singly linked and circular linked list (especially in cases where the skip factor k, is close fraction of n).

6. CONCLUSION AND RECOMMENDATIONS

The S-Linked list can be best applied in situations where backward traversals are not often needed. In such scenarios, the advantage of saving memory space is gained, and the difference in time required is somewhat marginal

For maximum performance of the S-Linked list, there is a need to balance the number of inherent circular lists in the S-Linked list with the number of nodes in each inherent circular list - which is the same thing as the skip factor, k. That is, n/k should be balanced with k such that, even though there are few inherent circular lists, the number of nodes that will be traversed in each inherent circular list should also be minimal, so as to achieve maximum throughput.

In order to best maximize the limited memory space resource in embedded systems using S-Linked list, the fewer the number of inherent circular lists, the better. It is better to have more inherent circular lists in the S-Linked list, than to have few inherent circular lists with numerous nodes in each inherent circular list. In other words, for optimum efficiency it is better to have $k << n/k$ than to have $n/k << k$.

Future work should empirically investigate how maximum efficiency can be realized with the S-Linked List – will the list be most efficient when k is very big/very small/average/a factor of n. This will help investigate and realize the scenario(s) in which the S-Linked list can be best applied.

REFERENCES

[1] Agrawal S.C., Singh S., Gautam A.K. & Singh M.K., 2012. Basic Concept of Embedded 'C': Review, International Journal of Computer Science and Informatics (IJCSI) ISSN (PRINT): 2231 – 5292, Volume-1, Issue-3. Retrieved from: http://interscience.in/IJCSI_Vol1Iss3/16.pdf, on: May 7, 2012.

[2] Mostafa E. & Jerry L. T., 2008, Maximal strips data structure to represent free space on partially reconfigurable FPGAs. ipdps, pp.1-8, 2008 IEEE International Symposium on Parallel and Distributed Processing,

[3] Narasimha Y.M, 2009, Introduction to embedded systems. Retrieved from: http://www.slideshare.net/yayavaram/introduction-to-embedded-systems-2614825, on: May 7, 2012.

[4] Shyram C, 2012, Multiply linked lists. Retrieved from: http://www.classle.net/projects/multiply-linked-lists, on: May 7, 2012.

[5] Sinha, P., 2004, A Memory Efficient Doubly-Linked List. Linux Journal. Retrieved from: http://www.linuxjournal.com/article/6828, on May 15, 2012.

[6] Mehlhorn K., 2005, Representing Sequences by Arrays and Linked Lists. Retrieved from: http://www.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/Sequences.pdf, on: June 3, 2013

[7] Broadhurst M, 2010. Circular Linked List. Retrieved from:http://www.martinbroadhurst.com/articles/circular-linked-list.html, on: June 3, 2013.

[8] Department of Computer Science, University of Western Otario, 2006. Course note on: Analysis of Algorithms, CS 1037a – Topic 13. Retrieved from: http://www.csd.uwo.ca/courses/CS1037a/notes/topic13_AnalysisOfAlgs.pdf, on: April 24, 2012