The SC-Linked List – An Alternative Approach of the Linked List Data Structure

Prateek Kumar Agrawal
B. Tech Student
Department of Information Technology
Symbiosis Institute of Technology, Pune, India

Abstract— Utilizing doubly linked list rundown in inserted frameworks or embedded system can be costly. Albeit doubly connected is productive as far as time multifaceted nature and very adaptable in itself, empowering simple traversal and refresh, yet it frequently requests more memory space contrasted with other rundown information structure – this is because of the space required for the additional pointers, multiplying the sum required for a separately connected rundown. In this paper, we present the S-connected rundown – a half and half of the idea of the independently connected rundown and the round connected rundown. The half and half gives an information structure that is like the unrolled connected rundown, but instead than have a cluster in every hub, we have a separately connected rundown. An examination of the space unpredictability and asymptotic time multifaceted nature of the calculation was completed.

Keywords: Linked Lists, Algorithm, Time Complexity, Space Complexity

I. INTRODUCTION

Implanted frameworks for the purchaser market, for example, electronic toys, little cell phones, among others should be as financially savvy as conceivable [1]. Regardless of whether one-fourth of a dollar is spared, the way that a huge number of such frameworks will be sold to the open is a lot of cost putting something aside for the organization being referred to. Albeit most information structure calculations are worried about time intricacy and productivity. It is examined qualified to think about amplifying the utilization of the economical yet constrained memory asset in installed frameworks [2]. By and large, installed frameworks regularly have restricted assets - particularly memory asset - and they hate the benefit of having auxiliary stockpiling gadgets, for example, the CDROM found in PCs. Cell phones, for example, cell phones, Personal Digital Assistants (PDAs), PDAs and so forth are an uncommon class of inserted frameworks. The restrictions of the cell phones just as some other inserted frameworks incorporates memory requirements, little size, among others[3].

Because of the memory imperative, implanted framework programming requires the utilization of dynamic memory portion. With regards to dynamic memory distribution, one of the favored information structure is the connected records information structure. A connected rundown information structure is known to be one which may change amid execution since progressive components are associated by pointers. Connected records are regularly connected with dynamic portion of memory, yet the utilization of connected records in inserted frameworks might be very expensive.[3] This is because of the memory space required for keeping the pointers/reference to hubs in the rundown.

II. PROBLEM STATEMENT

In embedded systems, the use of doubly linked lists is excessive. 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]

III. METHODOLOGY

This paper proposes and introduces the SC-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 asymptotic time complexity of the algorithm was carried out.

IV. AN OVERVIEW OF LINKED LIST DATA STRUCTURE

In software engineering, a Linked rundown is a direct accumulation of information components, whose request isn't given by their physical position in memory. Rather, every component focuses to the following. It is an information structure comprising of a gathering of hubs which together speak to a grouping. In its most fundamental structure, every hub contains: information and a reference (at the end of the day, a connection) to the following hub in the arrangement. This structure takes into account effective inclusion or expulsion of components from any situation in the succession amid emphasis. Progressively mind boggling variations include extra connections, permitting increasingly productive inclusion or expulsion of hubs at subjective positions. A disadvantage of connected records is that get to time is straight (and hard to pipeline). Quicker access, for example, irregular access, isn't possible. Clusters have better reserve region contrasted with connected records.

Linked list / connected records are among the least complex and most normal information structures. They can be utilized to execute a few other normal conceptual information types, including records, stacks, lines, acquainted exhibits, and S-articulations, however it isn't exceptional to

actualize those information structures straightforwardly without utilizing a connected rundown as the premise.

The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while restructuring an array at run-time is a much more expensive operation. Linked lists allow insertion and removal of nodes at any point in the list, and allow doing so with a constant number of operations by keeping the link previous to the link being added or removed in memory during list traversal.

On the other hand, since simple linked lists by themselves do not allow random access to the data or any form of efficient indexing, many basic operations—such as obtaining the last node of the list, finding a node that contains a given datum, or locating the place where a new node should be inserted—may require iterating through most or all of the list elements. The advantages and disadvantages of using linked lists are given below. Linked list are dynamic, so the length of list can increase or decrease as necessary. Each node does not necessarily follow the previous one physically in the memory.

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 n.

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 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 list, 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).

V. THE PROPOSED SC-LINKED LIST

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

A. *Creating the SC-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

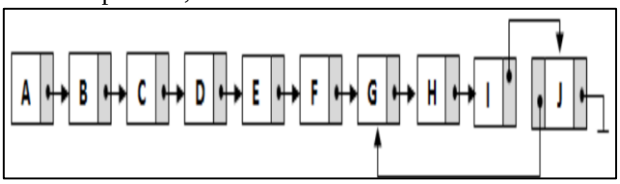


Fig. 2: Backward skipping of the list

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.

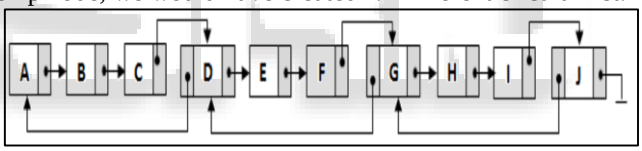


Fig. 3: SC-linked list with ≈n/k ≈3 inherent circular lists

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.

B. *Adding to the List*

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

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,
1) Make new node point to head node
2) Make backward pointer from inherent circular list's tail node point to new node
3) Eliminate previous backward pointer
4) New node now becomes the head node
b) Else,
1) Make new node point to head node
2) 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)

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)

1) Make last node in the inherent circular list to point to new node
2) Create backward pointer from the new node to point to head node of the inherent circular list
3) Eliminate previous backward pointer
4) Make new node point to NULL

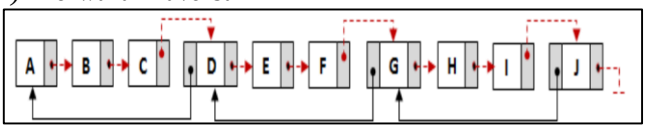
b) Else,

1) Make last node in the inherent circular list to point to new node
2) Create a backward pointer from the new node to itself. (with this, a new inherent circular list having one is created)

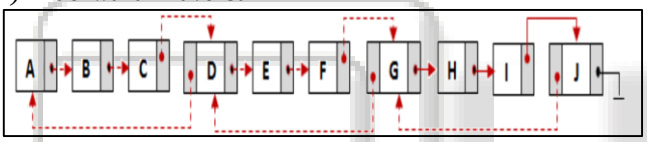
C. Traversing the List

The forward traversal of the SC-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 SC-linked list, the tail node of the current inherent circular list must be reached. (see Figure 12 – follow dashed lines)

1) Forward Traversal



2) Backward Traversal



D. Complexity Analysis of the SC-linked List

The time complexity analysis of the SC-linked list is stated in Table 1a below, a comparison of the time complexity of the SC-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 SC-linked list alongside that of the circularly, doubly and singly linked list:

1) Time Complexity of the SC-linked List

In the SC-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 linear time.

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

Table 1a: Time complexity of operations on the SC-linked list

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]

Table 1b: Comparison of SC-linked time complexity with other list data structure

2) Space Complexity of SC-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.

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

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

VI. Conclusion and Recommendations

The SC-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 SC-linked list, there is a need to balance the number of inherent circular lists in the SC-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 SC-linked list, the fewer the number of inherent circular lists, the better. It is better to have more inherent circular lists in the SC-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 SC-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 SC-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.mpiinf.mpg.de/~mehlhorn/ftp/Toolbox/Sequences.pdf, on: June 3, 2013

[7] Broadhurst M, 2010. Circular Linked List. Retrievedfrom:http://www.martinbroadhurst.com/articles/cir cular-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

