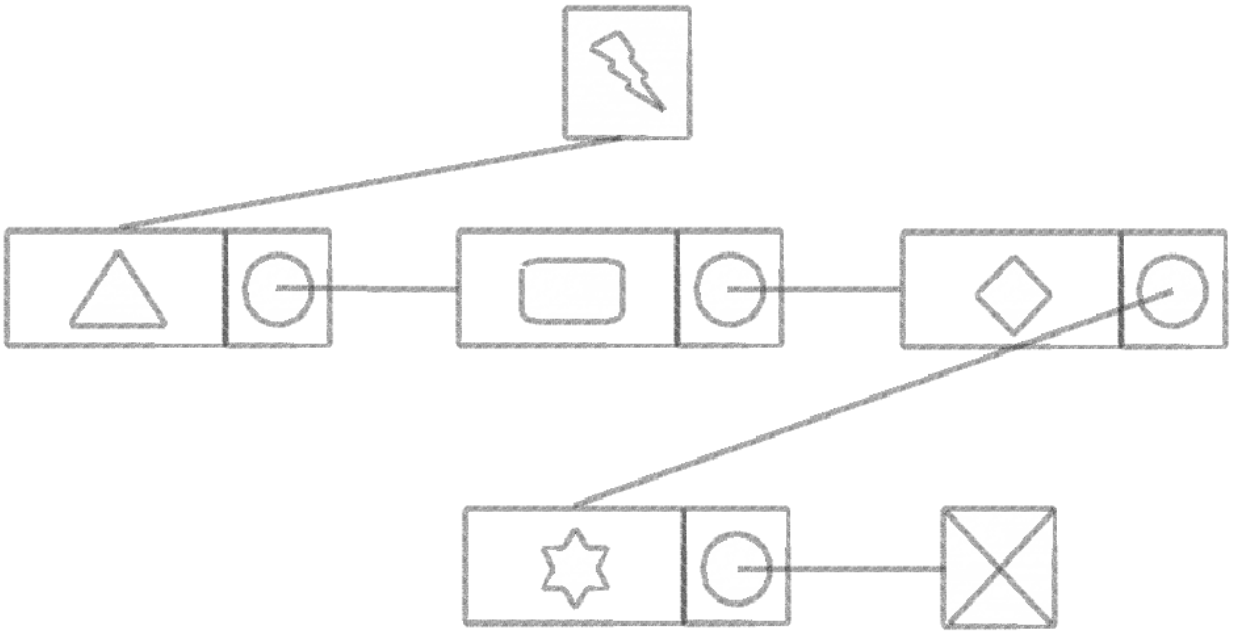
COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 9: APR 16

Lecturer: Dr. Andrew Hines *Scribes: Patrick Cormac English, Ciaran Barron*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*



Linked Lists

Contents

9.1 Outline

9.1.1 Learning Objectives

The learning objective for this class is to broadly understand the nature of the "Linked List" ("LL") data structure, its operations, and its specific suitability for practical application. This lecture review will cover the following areas:

- Linked Lists - General Overview
- The Linked List Structure - Contrast and Comparison with other linear data structures
- Applications of the Linked List Data Structure
- Operations on the Structure - Explanations and Time Complexity
- Extended forms of the Linked List (Skip lists, Multidimensional lists)

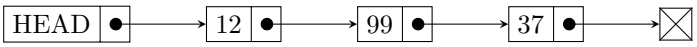
At the end of this review, the reader should have a fuller understanding of those characteristics of the Linked List that would, in a given scenario, benefit/disadvantage the application of a broader algorithm.

9.2 Why use a linked list?

A linked list is an abstract data type which links nodes sequentially using pointers. It provides an alternative to an Array based structure for situations where arrays are not suitable. Arrays have a number of limitations which are addressed by linked lists. The main ones being, arrays are fixed size and arrays are slow to insert/remove elements from. The structure of the linked list allows for quick and easy insertion/ removal of elements from the list and re-sizing of the list is possible even without a contiguous block of memory.

9.2.1 Structure

The basic structure of the linked list is shown in the diagram below.

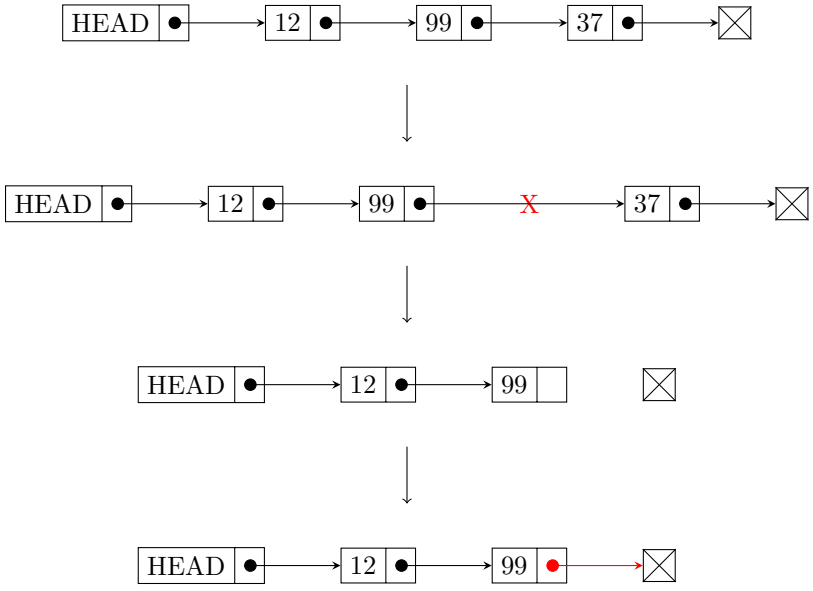


A linked list is made up of nodes. Each node stores an item of data and contains a reference to the next node in the list. In order to access any node in the list, you must start at the head element and follow each link in the structure until the desired element has been found. For example, to access element 2 in the above linked list, the following procedure must be followed:

1. Locate head of the list, a pointer to the head is generally what is stored in the linked-list variable in a script.
2. Once the head has been located, the link stored in the head must be followed to the next node (element 1), this link is generally a pointer to an address in memory where the next element is stored.
3. finally the pointer found in the first element of the list must be followed to arrive at the required node.

The end of a linked list is denoted by a null terminator. The last element of the list will point to a null value. This value tells the system that the last element has been reached and that there are no more nodes connected. If, for example, you wished to delete all elements from the list after element 1, then you would need to break the link between element 1 and 2 and replace it with a link to a null value. This is essentially just replacing the stored pointer with a new pointer to a null value.

Sample procedure to truncate a linked list:



9.2.2 Operations

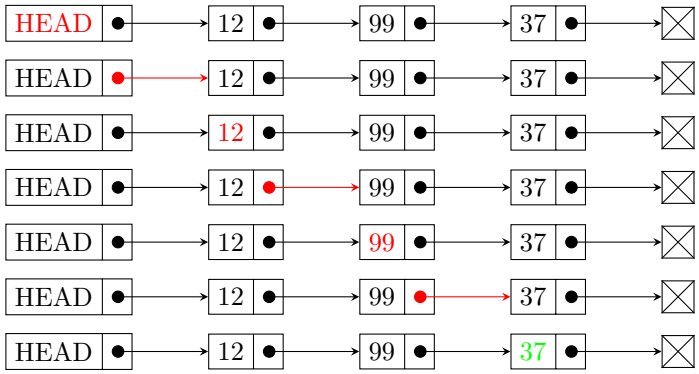
All operations on an element in a linked list, other than the head, will require the traversal of the list. As a linked list can only be accessed from the header and each node is found via a pointer on the previous node the only way to reach any node is to traverse the list. Once you can traverse the list and reach all elements there are a number of other operations which can be performed, a brief description of the main operations is provided below.

1. size, is_empty

   This operation finds the size of a linked list/ can also be used to check it is not empty.

2. get_elem_at_rank

   To find an element at a given rank, start with the first head of the list and move to the next pointer repeatedly until the required element is found. In the example below the 3rd node is found by traversing the list. The red marker indicated the current position in the list at each stage. The green marker represents completion of the search.

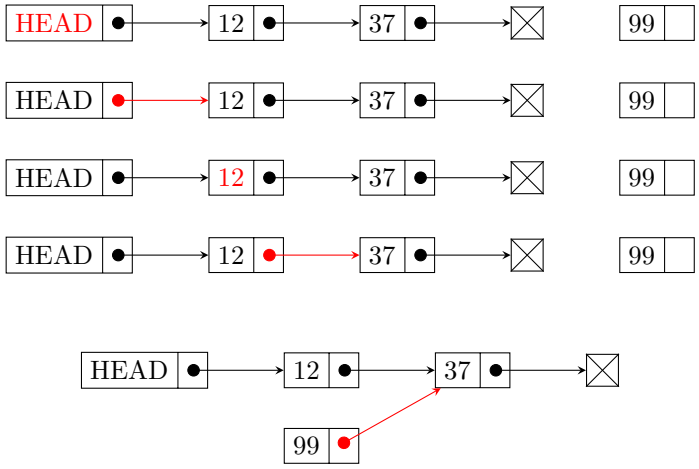


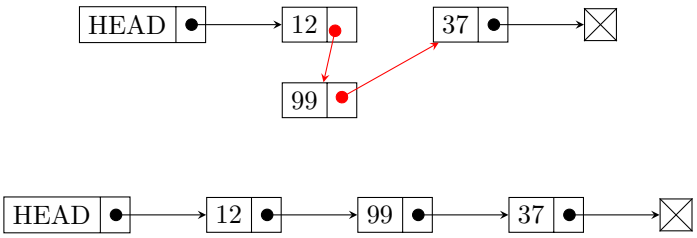
3. set_elem_at_rank

   To set the element at a given rank a similar procedure to the previous graphic should be followed, with an additional step. While the get_elem_at_rank returns the value stored in the list at the required rank the set element operation change that value.

4. insert_elem_at_rank

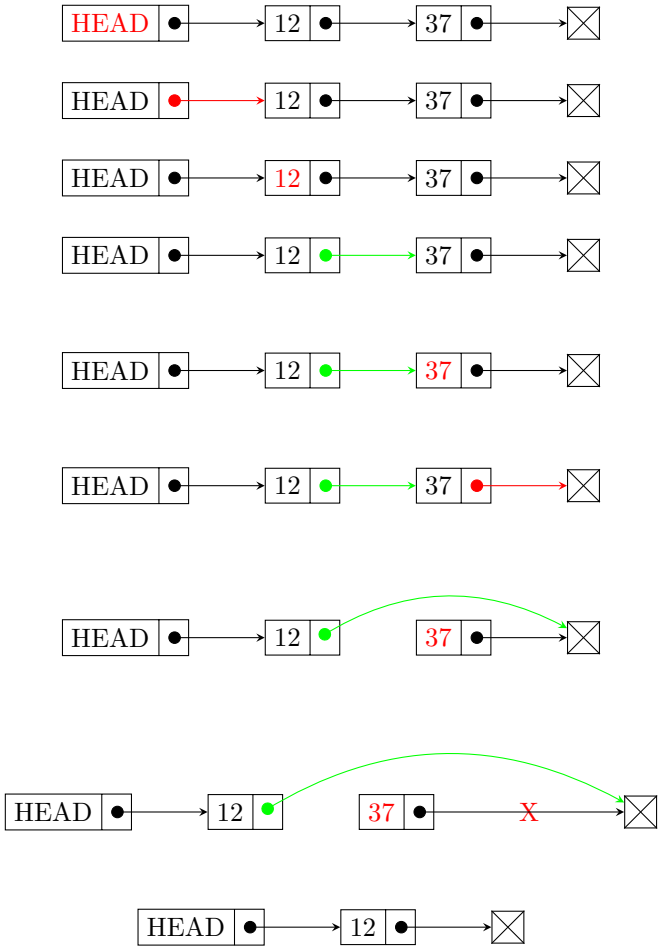
   The insert_elem_at_rank allows the user to include an additional node in the list between two nodes at a given rank. The operation locates the element which currently points to the element, X, at the given rank, then assigns the pointer value in the element to be inserted to point to element X, and sets the element which originally pointed to X to point to the new element. The illustration below shows an example of this operation, In the given example the goal is to insert a new element at rank 2. The element is shown on the right of the linked list.





5. remove_elem_at_rank

   The remove_elem_at_rank operation follows a similar procedure to the insert_elem_at_rank operation. Starting from the header of the list, the node pointing to the element at the rank to be deleted is found, elem_1. The address of elem_1 is stored and the next item of the list is found. The pointer in elem_1 is then replaced with the address stored in this element. An example is added below, green indicates a stored value. In this example element at rank 2 is deleted.



6. insert_after, insert_before

   Insert_after, and Insert_before are essentially the same as the insert operation with a minor change to the values stored during the process.

7. insert_first, insert_last

   The insert_first and insert_last operations are also similar to the insert at rank operations, but have some additional changes to be made.When the last element is changed the same process as insert_after can be used, but with the last element of the list. The insert last operation is particularly useful when populating a linked list in a loop.

9.2.2.1 Time complexity of operations

The table below shows the time complexity of various operations on a linked list compared to an array structure.[1]

Operation	Arrays	Linked Lists
size, is_empty	O(1)	O(1)
get_elem_at_rank	O(1)	O(n)
set_elem_at_rank	O(1)	O(n)
insert_elem_at_rank	O(n)	O(1)*
remove_elem_at_rank	O(n)	O(1)*
insert_first, insert_last	O(1)	O(1)
insert_after, insert_before	O(n)	O(1)*

The star beside the O(1) for insert_elem_at_rank, remove_elem_at_rank, and insert_after/before operations indicate that while this operation is itself O(1) to complete, it may take O(n) to complete. As this operation may require get_elem_at_rank operation to be executed prior to insert/remove, the complexity of the get_elem_at_rank operation must also be considered. The time complexity of get_elem_at_rank operation is is O(n). For this reason, the use of the insert_elem_at_rank operation may scale according to O(n).

The use of a linked list offers multiple advantages over the use of an array. As each node is only connected to the node ahead and behind it in the list, the size of the list does not need to be set prior to use. This is not the case with array based structures. This quality of being connected to only the preceding and subsequent elements of the list also allows for simple addition/deletion from the list. The pointer style connections between different nodes in the list means that the list can be fragmented in memory without losing any reduction in performance, this allows for increased memory performance over an array.

In general, the linked list structure outperforms the array structure when the data must be altered regularly but seldom read, while the Array structure outperforms the linked-list structure when the data is rarely edited but often read.

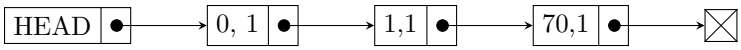
9.2.3 Applications

Linked lists can be applied in any scenario where data is written or read sequentially, or often. they are most often employed in a doubly linked list format but can be used as singly linked lists as well. One such example is in polynomial algebra:[2]

Linked lists can be used for representing numbers too large to store conventionally in a 64-bit number. If you wished to store the value of $2^{70} + 3$ then you would inevitably encounter an error of some sort by attempting to store it in 64 bits. The number can be converted to a polynomial by replacing the base with an algebraic variable and saving the coefficients in the corresponding element of the list. Applying this method to the above number, $2^70 + 3$, yields the following expression.

$$2^{70} + 3 \equiv x + x^2 + x^{70}$$

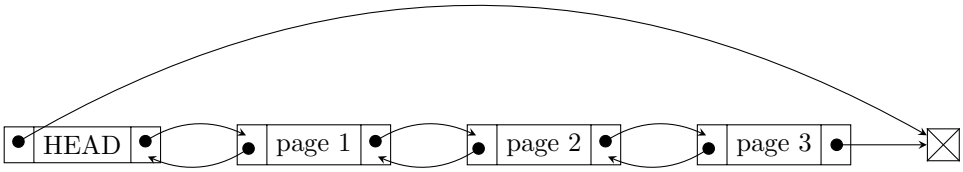
where x=2. A linked list view of this expression can be seen below. (Note: the left-most number is the index here, and the middle number is the data)



Then to convert this back to a number, you take the element index as the exponent of the base and the data as the coefficient, then add element-wise

$$1 * 2^0 + 1 * 2^1 + 1 * 2^{70}$$

Doubly linked lists, lists where each node has a bidirectional link (a pointer to both the preceding and subsequent nodes in the list), are used in a web browser to store the last page and next page information accessed when you hit the back/forward button.



The list is implement such that you can go back as far as the head (original page), when you click back the page is moved back along the list, and if you go to a new page the old link is cut and a new link is made and the previously stored forward information is lost. Doubly linked lists are generally used in any scenario where there is a forward or backward button.

9.3 Extended Forms

This section will expand upon the general concepts of the Linked List (pointers, chaining, "forward" movement) and describe how these can be expanded to more complex structures and applications.

9.3.1 Doubly Linked List

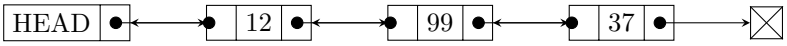
The doubly linked list is a variant of the Linked List format that includes an additional pointer (known as the "previous pointer"). This pointer links the current item to the item prior, allowing for "backward progression" through the list.

This ability to traverse the list in two directions has a number of advantages:

- The efficiency of a delete operation is increased in an intra-list context - As noted above, a delete operation in a Linked List requires the maintenance of a "previous" pointer, in order to maintain a record of that pointer for the purposes of "routing around" a deleted node. A doubly Linked List obviates the need for this additional pointer by permitting backward traversal.
- Similarly to the above, intra-list insert operations are also more efficient.
- Operations such as list-reversal become much simpler - simply change the "direction" of pointers, without having to reform the list structure.

However, the above come at the storage overhead of an additional pointer space for each node.

The following figure demonstrates a sample doubly Linked List.

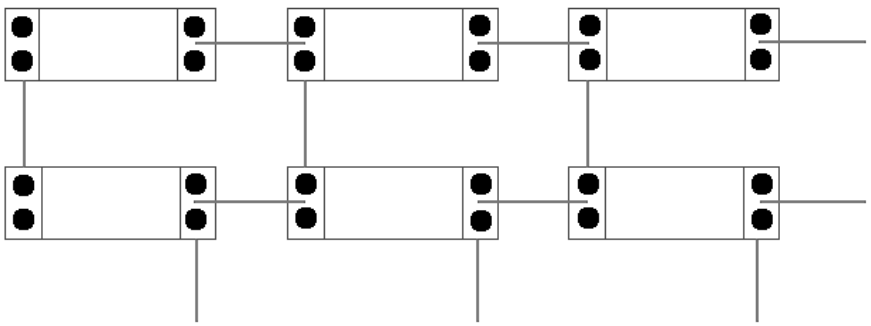


9.3.2 Multidimensional Linked List

The concept of additional links can be further expanded to allow for "dimensions" within a linked list. The most common way of visualising this is the "triply Linked List", an example of which may be seen at the bottom of this section.

The additional functionality/layers of traversal all require additional pointer space, in order to account for the additional axis. As seen below, a triply linked list requires 4 pointers ("Left, Right, Up, Down"). If one were to expand this concept to a fourth axis (i.e. a 3-dimensional list), each node would require 6 pointers, and so on.

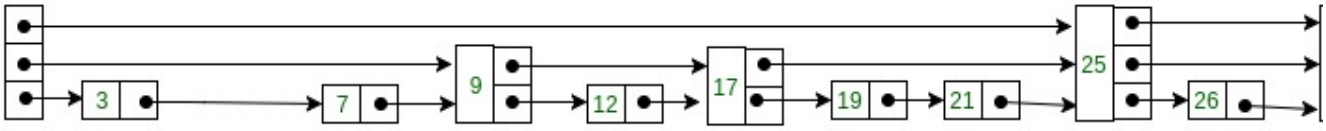
The linked list is not necessarily the most efficient structure for such an application, as graphs/multidimensional arrays are often more appropriate, given that overheads and maintenance complexity mount as dimensions increase.



9.3.3 Skip List

A skip list is an implementation of the Linked list that allows for more efficient searching, by allowing "jump traversal" though the list, rather than relying on incremental node traversal.[3]

The skip list operates by means of "node levels". At an abstract level, and as seen in the diagram below, different levels allow for different traversal gaps. The base level is a standard transition



The primary benefit of a skip list is retrieval time. Retrieval of items can be $O((n))$ on average within a Skip List context. The worst case time for a retrieval operation is essentially the number of "top level" nodes, and the number of nodes in a segment. If we take the example of the above table, we can see that attempting to search for 26 will take us to the first "level 3" node, and then to the first "level 1" node. Additional layering can reduce time complexity further, to the ideal average of $O(logn)$.

Skip lists can be complex, and it is possible to create an "inefficient" list from a space perspective. [1]. In addition, insertion is a more involved process. "Coin flips" are typically used to determine the "level" of insertion. ie. a heads and a tails might mean an insertion at levels 0 and 1, while two tails might mean an insertion at levels 0, 1, and 2.

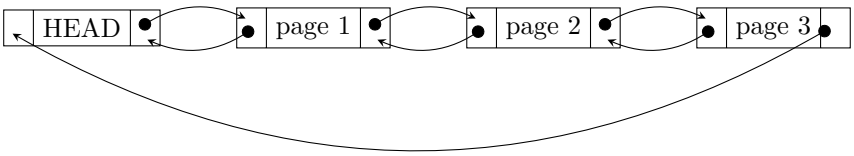
9.3.4 Circular list

A Circular Linked List is a variant of the Linked List where the "out" pointer of the last node, rather than pointing to "null", instead directs to the address of the first node. The diagram at the base of this section depicts a typical Circular List.[4]

There are two primary advantages to the Circular List format:

- The list can be traversed continuously, from any point, meaning that searches need not proceed from the "head element"

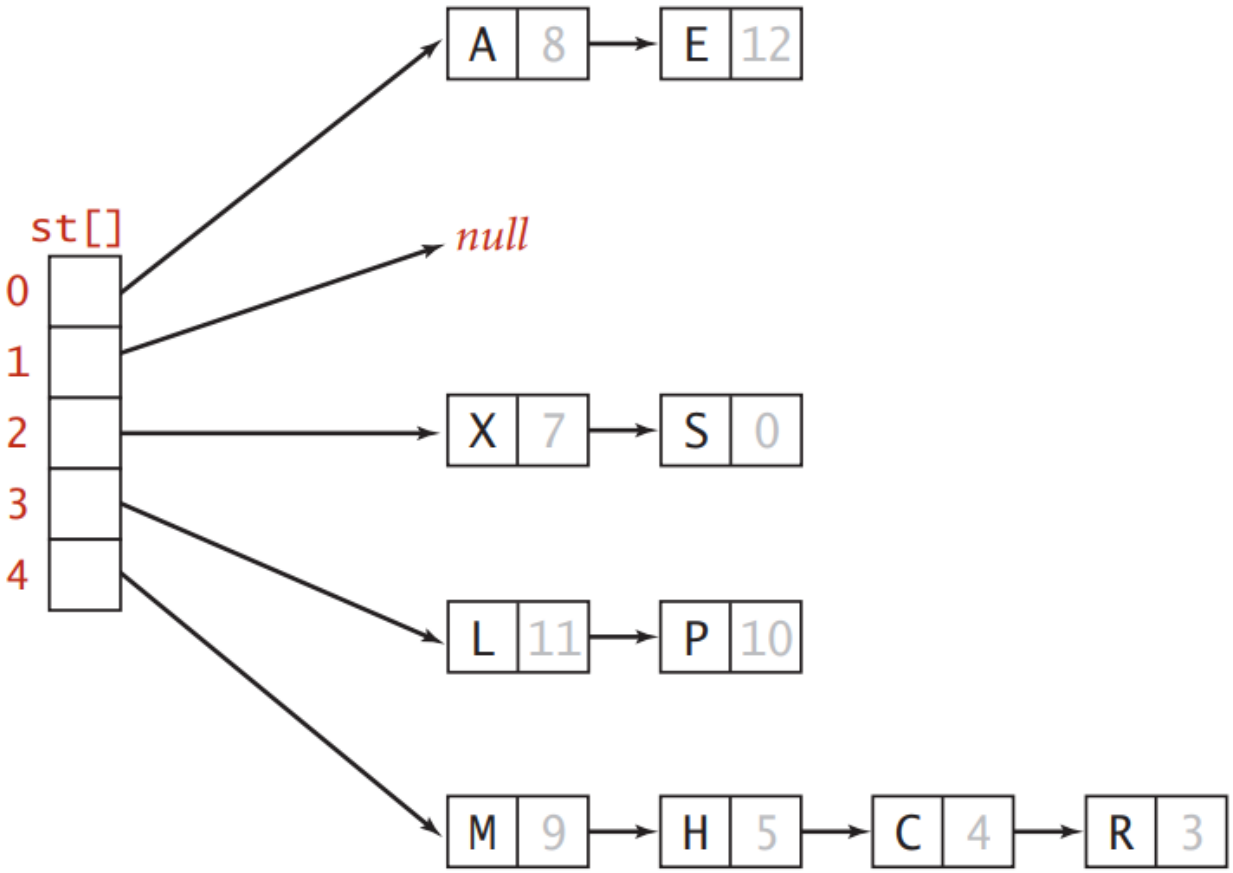
- The above characteristic also allows for the implementation of other structures, such as a queue, or a Fibonacci Heap, allowing for additional flexibility.[2]



[1] http://ticki.github.io/blog/skip-lists-done-right/
[2] https://www.geeksforgeeks.org/circular-linked-list//

9.3.5 Separate Chaining Symbol Table

One final, and somewhat more specialised variant of a linked list arises from its hybridisation into the Separate Chaining Symbol Table ("SCST"). More information about this entity can be obtained in lecture series 12, but a diagram of the structure can be found below. The SCST is used as a means of facilitating the implementation of Hash Tables, and involves subsuming individual linked lists within specific hash outputs in a hash table, allowing for their use without "perfect" hashing.



References

[1] "Lecture notes, set 9. comp20230: data structure and algorithms.",

[2] *Linked lists, csmsu*, https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html.

[3] http://ticki.github.io/blog/skip-lists-done-right/2.

[4] https://www.geeksforgeeks.org/circular-linked-list//.