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

Lecture 9: Feb 21

Lecturer: Dr. Andrew Hines *Scribes: John McLoughlin, Toms Murphy*

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

Contents

Overleaf: https://www.overleaf.com/read/czrygrdzxwkt

9.1 Outline

This session introduces linked lists and shows how they can be created, traversed and manipulated. It also provided information on their Big-O as opposed to array-based structures and describes some of their use cases.

9.1.1 What is a Linked List?

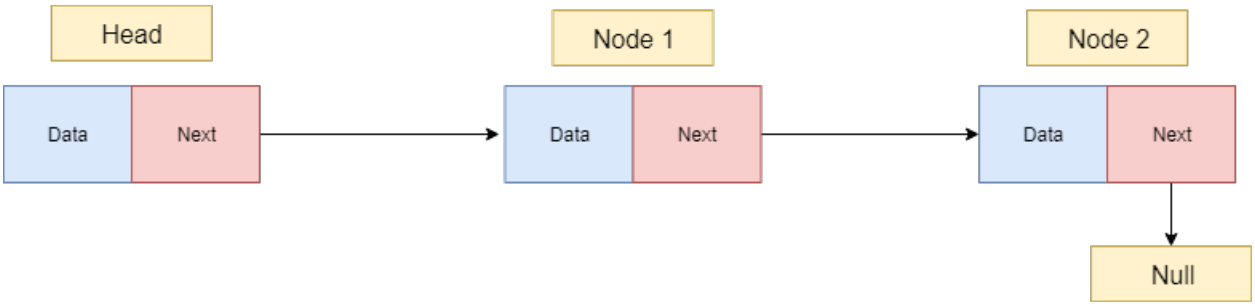


Figure 9.1: Example of a Linked List

A linked list is a collection of nodes each of which consists of two references, a reference to an object that is an element of the sequence and a reference to the next object in that sequence. This results in a linear sequence which can only be traversed in one direction. The first node is known as the head. Linked lists are used as an alternative to array-based sequences such as the list found in Python. The array is more centralized, occupying a single chunk in memory, whereas the linked list is distributed and spread out through the memory. As a result, the size of a linked list is not determined at its creation; it is free to expand and contract as needed as it uses space proportionate to its current number of elements. This introduces both advantages and disadvantages in terms of modifying or traversing a linked list.

Linked Lists	**Arrays**
Dynamic, scalable size which is efficient to alter	Fixed size which is expensive to alter
Inserting and deleting elements is efficient as there is no shifting required	Insertions and deletions usually require shifting and is therefore expensive
No random access, thus is not a suitable option for accessing elements by index as seen in sorting options	Random access supported, which is suitable for access by index
No memory wasted as memory is allocated as needed	If the array is full then there is no memory wasted, otherwise there is memory allocated to nothing
Slow sequential access due to elements not being contiguous in memory	Contiguous memory allocation means fast sequential access

Table 9.1: Linked Lists vs Arrays

9.1.2 Linked List Representation in Memory

Each node is a unique object which stores both a reference to its element and a reference to the next node in the sequence or, in the case of the tail, a reference to null. Another object represents the list as a whole and, at a minimum, this object must contain a reference to the head of the list. Without this reference the head would be lost and, consequentially, the entire list would be lost. The tail of the list isn't usually stored because it is locatable by traversing the list but it is sometimes stored for the sake of convenience. The length of the list may also be stored, once again this is for convenience. As with the tail, if the length of the list is not stored then the list must be traversed in order to find it.

9.1.3 Linked List Big-O

Operations	**Arrays**	**Linked List**
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)

Table 9.2: Linked Lists vs Arrays: Big-O

O(1) means that an operation will always execute in constant time regardless of the size of the input whereas O(n) means that the time an operation takes will scale linearly with the size of the input, ie: if you double the size of the input, the length of time to execute should also double. In the case of a linked list, traversal isn't counted twice so that if we are already at the rank we want to insert into then the insert is O(1) otherwise it will become O(n) since the list must be traversed in order to find the position at which we want to insert.

9.2 Basic Operations

9.2.1 Get and Next

Two basic methods are used to access elements in a linked list. If current refers to the current element of the list, current.get_element() returns the data object of the current node while current.get_next() returns the data object of the next node.

9.2.2 Traversing a Linked List

Traversal of a linked list begins at the head and proceeds to the tail, the path being determined by the pointers present in each node.

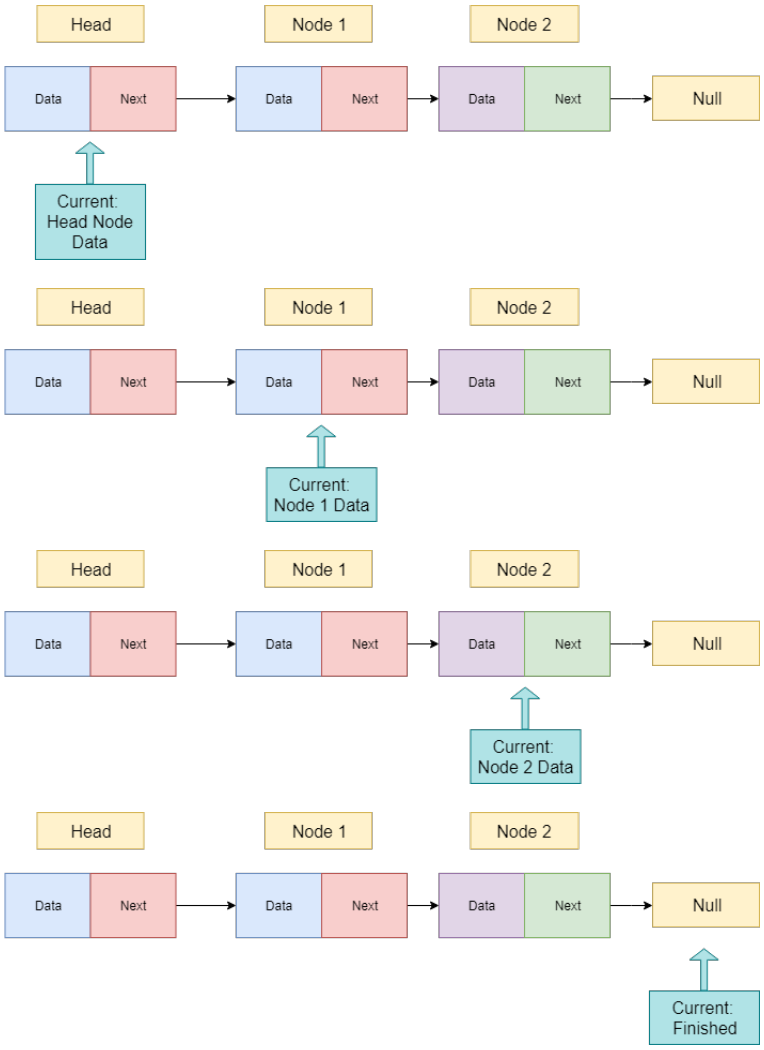


Figure 9.2: Traversing a Linked List

Traversal Pseudocode

```
Input: Linked list,headNode
Output: Every node has been traversed
    currentElement points at headNode
    while currentElement not equals None/null:
        print currentElement.get_element()
        currentElement points at currentElement.get_next()
```

9.2.3 Inserting into a Linked List

9.2.3.1 Inserting at the Head of a Linked List

Insertion at the head of a linked list begins with the creation of a new node containing a new data object and its pointer is pointed at the current head of the list. The list's head reference is then set to point at the new element.

Inserting At The Head Pseudocode

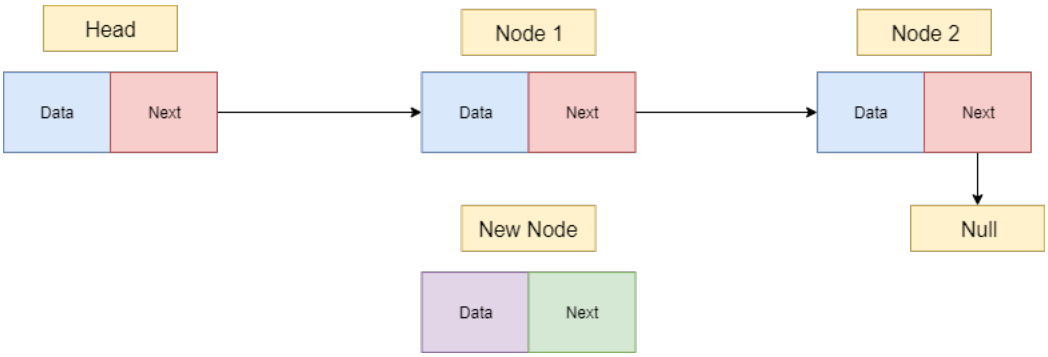
```
Input: Linked list, newElement, head
Output: newElement inserted at head
    newest = newElement
    newest.next points at headNode
    List.head points at newest
```

9.2.3.2 Inserting at the Tail

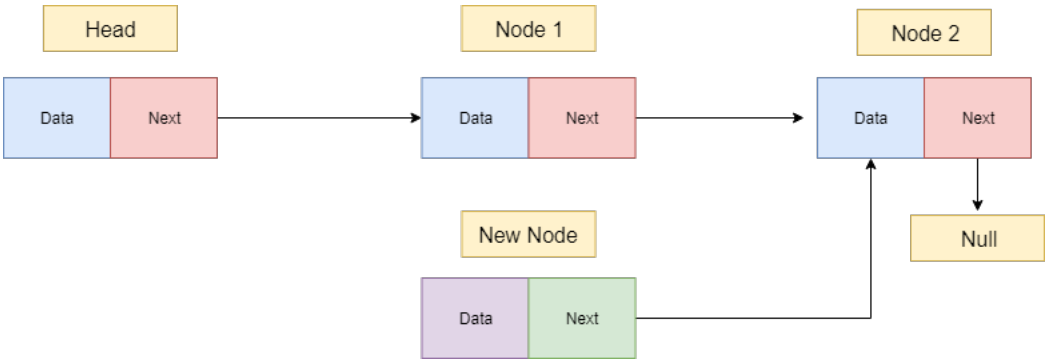
The ease of inserting at the tail of a linked list depends on whether or not a reference to the tail was stored. If this reference was stored then the list won't have to be traversed in order to locate the tail. Once the tail has been located, either through direct reference or by traversal, insertion begins by creating a new node. This node's pointer is pointed at null and then the pointer of the current tail is pointed at the new node. If a tail reference exists, this must also be updated to point at the new tail.

Inserting at the Tail

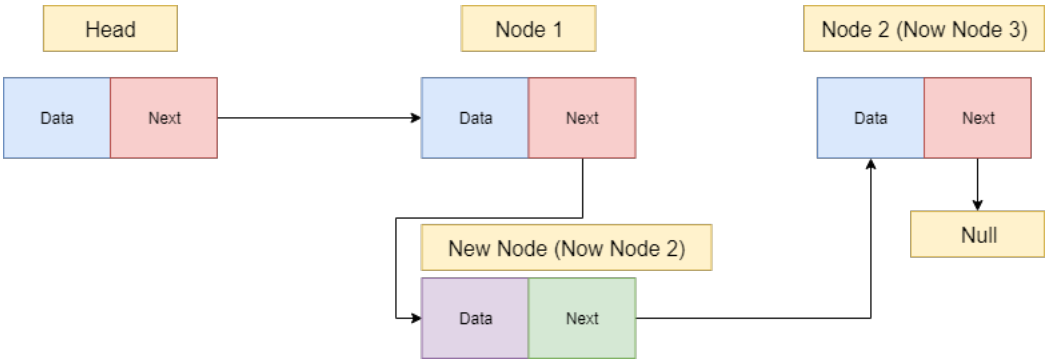
```
Input: Linked List, newElement, tail
output: newElement added at the tail
    newest = newElement
    newest.next points at None
    tailNode.next points at newest
    List.tail points at newest
```



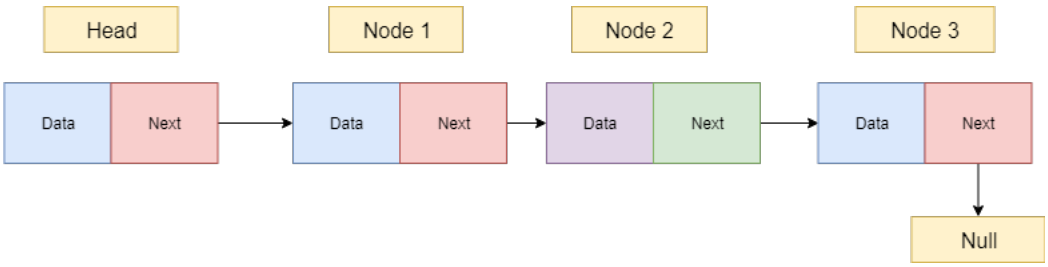
(a) Defining a New Node



(b) Point NewNode.next at Node2 Data



(c) Point Node1.next at New Node Data



(d) New Linked List

Figure 9.3: Inserting Into a Linked List

9.2.4 Inserting a Node at Rank R

In order to insert a node into a linked list at rank r, it is necessary to traverse the linked list until you reach rank r.

Inserting at Rank R Pseudocode

```
Input: LinkedList, newElement, head, desiredRank
Output: newElement is inserted at rank r
if r greater than 0:
    r = desiredRank        newest = newElement
    current points at headNode
    while current.next not equal None and r greater than 1:
        current points at current.next
        r = r - 1
    newest.next points at current.next
    current.next points at newest
else:
    add_head(LinkedList, newElement)
```

9.2.5 Removing from a Linked List

9.2.5.1 Removing at the Head

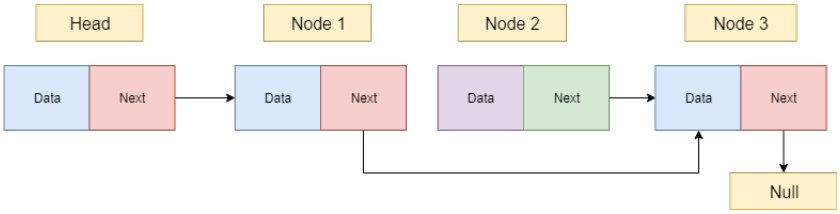
Removing at the head follows the reverse of the procedure for adding at the head. First the list head reference is pointed to the element following the current head, then the current head's pointer is pointed to null.

Removing at the Head Pseudocode

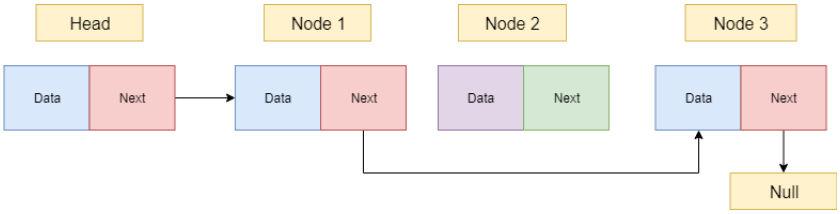
```
Input: LinkedList, head
output: None
    if List.head equals None:
        List is empty
    else:
        List.head points at headNode.next
```

9.2.5.2 Removing at the Tail

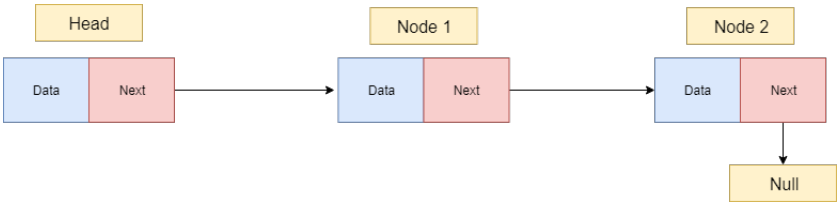
In order to delete at the tail of a linked list, a reference to the preceding node is required. In order to obtain this, the list must be traversed. When located, the node preceding the tail can have its pointer pointed at null, forming the new tail. This is an expensive process and is often not implemented.



(a) Point Node1.next at Node3 data



(b) Point Node2.next at Null



(c) New Linked List

Figure 9.4: Removing From a Linked List

9.3 Linked Lists Applications

Linked lists in the real world tend to use double linked lists, where each node contains references to both the next AND the previous nodes except in the case of the head and tail nodes. Whenever a next and previous button is implemented, there is a high chance a double linked list has been used to create this functionality. The following subsections list places where linked lists are commonly used.

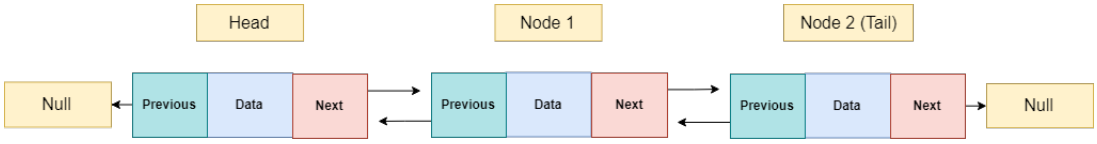


Figure 9.5: Example of a Doubly Linked List

9.3.1 Music Players

Music players - A song can be thought of as a node in a linked list known as a playlist. Because double linked lists point to the next and previous node, skipping back or forward a song is simple, as is the play next functionality. Adding new songs is quick and easy (Big-O of O(1)). Also, going to the start or end of a playlist is possible using the head and tail properties of a linked list.

9.3.2 Image Galleries

When viewing images, a user can cycle through images with the next and previous buttons. This is similar to the behaviour described for the music player.

9.3.3 Web Browsers

The forward and back buttons in a web browser give access to a linked list of websites visited.

9.3.4 Other Use Cases

In the computer science world, linked lists are often used for graphs, stacks, queues and directories.

References

[1] Data Structures in the Real World? - Linked Lists
`https://medium.com/journey-of-one-thousand-apps/data-structures-in-the-real-world-508f5968545a`

[2] Linked Lists vs Arrays
`http://code.cloudkaksha.org/arrays/linkedlist-vs-array`

[3] Data Structures and Algorithms - Linked List
`https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm`

[4] Data Structures and Algorithms in Python
`Goodrich, M., Tamassia, R. and Goldwasser, M. (2013). Data structures and algorithms in Python. Hoboken, N.J.: Wiley.`

[5] The Imposter's Handbook
`Connery, R. (2016). The Imposter's Handbook: A CS Primer For Self-Taught Programmers`