

Algorithms and Data Structures

*Notes on Algorithms, Data
Structures and other concepts*

ERIC LI

February 23, 2019

Contents

1	Disclaimer	2
2	Linked Lists	2
2.1	Structure	2
2.2	Operations	2
2.2.1	Insertion	3
2.2.2	Deletion	3
3	Queues	4
4	Sources	4

1 Disclaimer

The author(s) of this document assume(s) no responsibility or liability for any errors or omissions in the content of this document. The information contained in this site is provided on an “as is” basis with no guarantees of completeness, accuracy, usefulness or timeliness or of the results obtained from the use of this information.

Information provided in this document has been taken from various sources and is by no means a comprehensive record of the source’s views nor of the concepts represented.

Note:

The concepts considered here will be largely captured in the scope of the language C, but can be generally be adapted to other languages, and are conceptually similar.

2 Linked Lists

Linked lists are helpful in addressing some of the limitations that arrays face, such as inherent dynamic memory allocation and ease of modification. Instead of memory being assigned for use (as in the programming language C) linked lists are built on the premise of allocating memory as needed, and inserting or deleting as required by the user or data set.

2.1 Structure

A linked list is a linear data structure constructed from building blocks called nodes. Like other data structures, a node contains two items: data, which can be simple, compound, pointer, etc. and a memory address reference (or link to the next node, called a pointer in Figure 1). The address reference will access the next node’s memory address in the list, and creates a ‘chain-like’ relationship which will naturally produce a linear structure. There are two important nodes that are called the ‘head node’ and the ‘tail node’, which will influence operations and be explained in depth in the following sections.

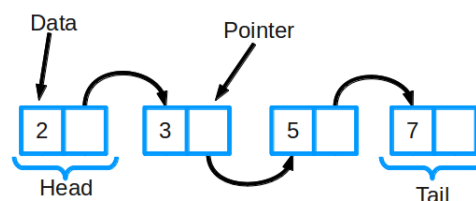


Figure 1: Linked List

2.2 Operations

Linked lists support basic ADT operations of inserting, deleting, and modifying. For operations of inserting and deleting involving the head or tail, implementation requires several changes, but operates in a similar manner nonetheless.

In order to understand operations on linked lists, we must identify what head pointers or head

memory address are. Intuitively, the head address's are used to keep track of the linked list and allow for access on all functions. Without the head pointer, we lose the key to the linked list and are unable to both operate, and modify the list.

Therefore, it **imperative** that we return new head address's when modifications to the linked list occur that will affect it.

2.2.1 Insertion

Conceptually, inserting requires a list traversal to the desired point of insertion and memory address manipulation so that the address reference of the nodes link together appropriately.

Inserting in Linked Lists

Inserting inside the list requires a few steps;

- Traversal of the list to the desired location
- Accessing the previous node's reference to the succeeding node
- Setting the memory address of the succeeding node to the address link of the inserted node (linking inserted node to next node)
- Set the memory address of the inserted node to the address link of the previous node (linking inserted node with previous node)
- Return the new head address

Similarly, this can be adapted for **inserting at tail**.

Insertion at Head

Inserting at the head of the list is by far the quickest insertion due to the lack of list traversal and ease of access to the head. Because of this, inserting at the head requires much fewer steps;

- Access the head memory address
- Set the address to the memory link of the inserted node
- Set the head pointer to the inserted node's memory address (thereby setting the inserted node as the head)
- Return the new head address

As we can see, the steps are much more straight-forward and as such, if the items do not require a methodical order or specific location of insertion, inserting at the head is recommended.

2.2.2 Deletion

Conversely, we have deletion which follows a similar format as insertion of nodes;

Deleting in Linked Lists

Deleting in linked lists requires several steps;

- Traversal of the list to the desired node

- Accessing the previous node's reference to the node to delete
- Setting the previous node's reference to the node to delete's reference
- Free the node to delete from memory
- Return the new head address

While this may be conceptually hard to grasp at first, when the implementation is shown, deleting will be much more clear and evident. Intuitively, we are aiming to delete the node, while allowing the previous node to preserve the connection with the succeeding nodes.

Deletion at Head

With a basic understanding of deleting and inserting in a linked list, deleting at the head, is quite rudimentary;

- Free the head node to delete it from memory
- Return the new head address

3 Queues

Queues are built heavily on concepts from linked lists and naturally, follows the same linear data structure. Queues provide a similar environment for "tail insertion" and "head deletion", which are appropriately named "enqueue" and "dequeue" respectively.

4 Sources

<https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked>
<https://medium.com/@kenny.lin/singly-linked-lists-5cfdec60bea0>