**FCC Intro to Data Structures and Algorithms**

**The Linked List:**

* Sequential access linear data structure in which every element is a separate object called a “Node”, which has 2 parts:
  + The data
  + The reference (or pointer) which points to the next Node in the List
* Sequential access is similar to the Stack or Queue…can only access them in a particular way…not random access like a list
* Linear means they are linked, one right after the next
* Node holds two pieces of information:

|  |  |
| --- | --- |
| * Data | * Reference/pointer to next * Node |

* Data is where the strings, Booleans, integers, etc. are stored
* Reference/pointer is a reference to the next Node in the Linked List

**Setting Up A Linked List:**

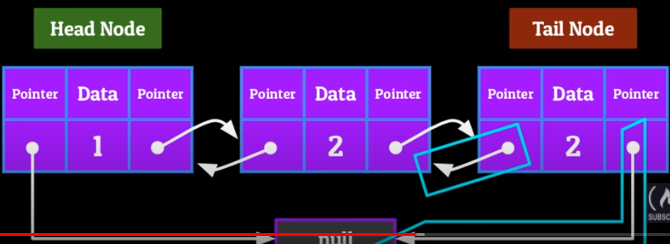
* Starts with a Head Node: **[Data:1 |Reference: null]** when no other Nodes have been created. Add a Node:
* **HN = [1 | ] [2 | ] [3 | null]** …..last Node is Tail Node
* So Tail Node always has a null value; tells computer that we are at the end of the Linked List

**Adding and Removing Elements From a Linked List:**

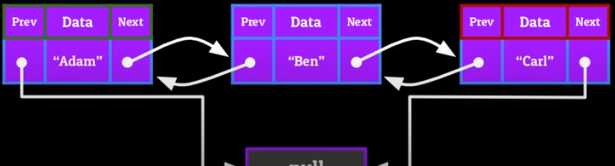
* Data can flow in and out of any point of a Linked List
* **Three Ways to Add/Remove from a Linked List:**
  + Add/Remove from Head
  + Add/Remove from Middle
  + Add/Remove from Tail
* Whenever we change a Node in a Linked List, we also have to change its pointers, which can get complicated
* **Basic Linked List:**
  + **HN = [1 | ] [2 | ] [3 | null]**
* **Adding to the Head of a Linked List:**
  + All that needs to be done is to set the new Node’s pointer to point to the former Head of the Linked List
* **Removing from the Head of a Linked List:**
  + Set the Head Node’s pointer to a null value, and it gets cut off
* **Adding a Node to the Middle of a Linked List:**
  + Make the pointer of the new Node point to the Node after the location we want to insert at
  + Set the Node before the location we want to insert at, to point to the new Node
* **Removing a Node from the Middle of a Linked List:**
  + Make the pointer that points to the soon to be removed Node, point to the Node after the one that is being removed
* **Adding to the Tail of a Linked List:**
  + Make the current tail point to the new Node you are adding
* **Removing from the Tail of a Linked List:**
  + Make the object before the tail point to a null value, signifying that it is the end

**Doubly Linked List – Continued**

**Visualization:**

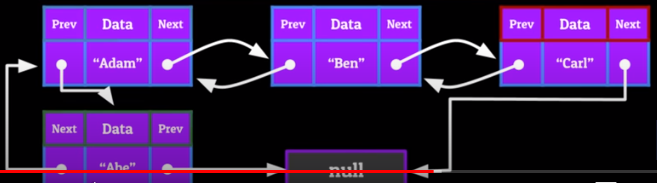
* **Head Node:** will at first point with both previous and next pointers to a **Null** value (no value before it to point to, and nothing after as of yet.)
* **Tail Node:** add one more Node to the head, and the Head’s “NEXT” pointer will point to the Tail Node, while the Tail Node’s “PREVIOUS” pointer will point to the Head Node. The Tail Node’s NEXT pointer will point to Null, as there is nothing after the Tail Node to point to.
* Add another Node, and it’s previous pointer will point to the old tail, while the old Tail’s NEXT pointer will point to the New Tail Node, etc.
* 

**Adding and Removing From a Doubly Linked List**



**Add to the Head of a Doubly Linked List:**

* if New Node == New Head:
  + set the New Node’s “NEXT” to point to the current head, and set the New Node’s “PREVIOUS” to point to Null Value.
  + Take the **CURRENT** Head’s “PREVIOUS” and point it back towards the New Node



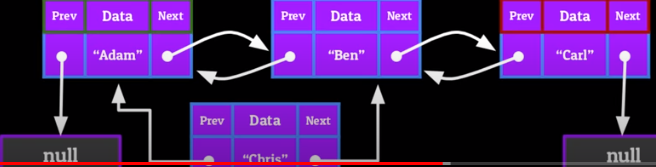
**Remove a Node From the Head of the DLL:**

* To remove the **CURRENT** head, point it’s “NEXT” pointer to a Null value instead of the second Node
* Set the second Node’s “PREVIOUS” pointer to Null as well

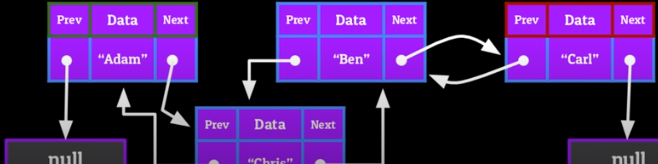
**Inserting into the middle of a Doubly Linked List:**

* **1).** Set the new node’s “PREVIOUS” to point to the Node previous to where you want to insert at.
* **2).** Set the new Node’s “NEXT” to point to the position after the one you are inserting at.
* **3).** Set the NEXT of the Node prior to where you are inserting to point to the New Node, and the PREVIOUS of the Node immediately after where you are inserting to point back to the New Node.

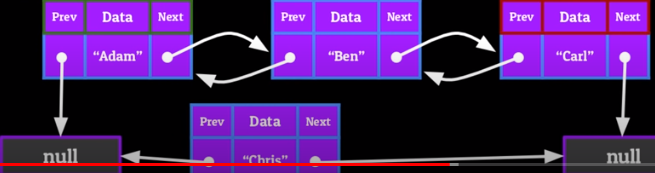
**Set the New Node’s NEXT to point to the Node after it, and it’s PREVIOUS to point to the Node before it.**



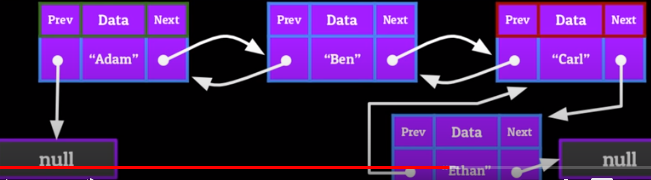
**Set the front node’s NEXT to point to the middle Node (new one) and set the end Node’s PREVIOUS to point to the middle Node (new one)**



**Removing a Node From a Doubly Linked List:**

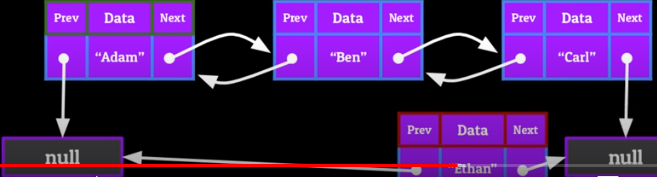
* **Front Node = node on left**
* **Middle Node = new Node we are inserting or deleting**
* **End Node = node on right**
* Set the FRONT NODE’s NEXT to point to the END NODE
* Set the END NODE’s PREVIOUS to point to the FRONT NODE
* Set both pointers of the MIDDLE NODE (one we are removing) to NULL
* 

**Adding to the Tail of a Doubly Linked List:**

* set the NEXT pointer of the current Tail Node to point to the New Tail Node
* set the PREVIOUS pointer of the New Tail Node to point to the current Tail Node
* set the NEXT pointer of the New Tail Node to NULL
* 

**Removing From the Tail of a Doubly Linked List:**

* Point the Current Tail’s PREVIOUS pointer to NULL
* Point the New Tails’s NEXT to NULL



**Key to remember that you only need to program the pseudocode for each Data Structure once, then you can use it over and over forever.**

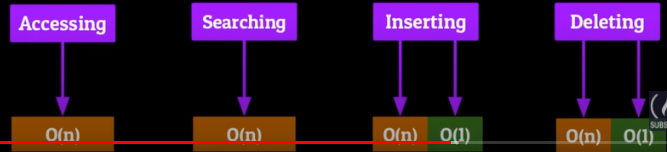
**Doubly Linked Lists – Time Complexity Equations:**

**1). Accessing – O(n)**

**2). Searching – O(n)**

**3). Inserting – O(n) or O(1)**

**4). Deleting – O(n) or O(1)**



**Doubly Linked List – Uses:**

The back and forth functionality of a Doubly Linked List lends itself to be implemented in a lot of Stack-like functionality:

* **A browser cache, which allows you to go back and forth between webpages**
* **“Undo/Redo” functionality in a lot of Word Processors/Excel programs**
* **“Open Recent” functionality in many applications**

**Dictionaries:**

* Dictionaries are one of the most abstract Data Structures
* Dictionaries are also sometimes called **maps** and **associative arrays**
* Dictionary stores information in KEY/VALUE pairs
* Think of a KEY/VALUE pair like a social security number….
* Dictionaries don’t have a numerical index….they use a key as their index.
* Keys can be anything you can think of
* **Two extremely important limitations of dictionaries:**
  + **Every key can only appear once in the dictionary**
  + **Each key can only have one value**
* **There can be duplicate values in a dictionary (two separate, different keys, with the same value)**

**Dictionary Time Complexity Equations:**

* **Hash Table Mini-Lesson:**