**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:**

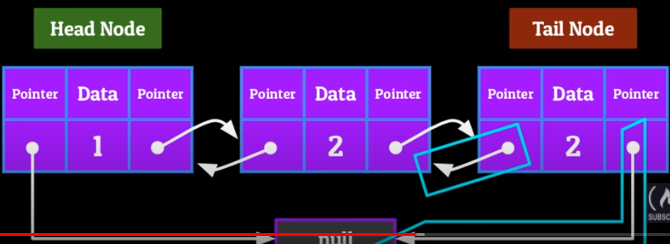
* Starts with a Head Node: **[Data:1 |Reference: null]** when no other Nodes have been created. Add a Node:
* **HN = [1 | ] [2 | ] [3 | null]** …..last Node is Tail Node
* So Tail Node always has a null value; tells computer that we are at the end of the Linked List

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

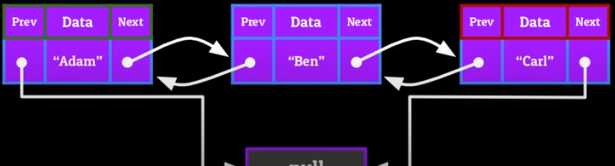
* Data can flow in and out of any point of a Linked List
* **Three Ways to Add/Remove from a Linked List:**
  + Add/Remove from Head
  + Add/Remove from Middle
  + Add/Remove from Tail
* Whenever we change a Node in a Linked List, we also have to change its pointers, which can get complicated
* **Basic 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:**

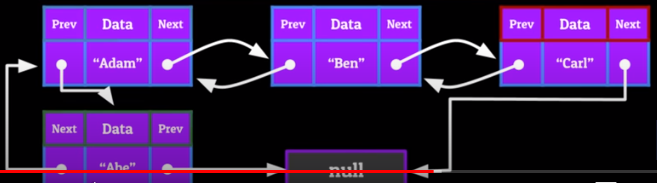
* **Head Node:** will at first point with both previous and next pointers to a **Null** value (no value before it to point to, and nothing after as of yet.)
* **Tail Node:** add one more Node to the head, and the Head’s “NEXT” pointer will point to the Tail Node, while the Tail Node’s “PREVIOUS” pointer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



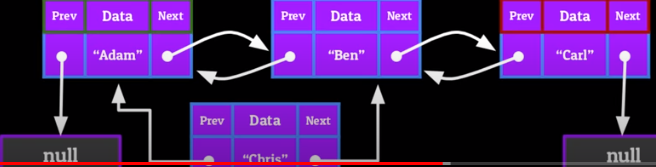
**Remove a Node From the Head of the DLL:**

* 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

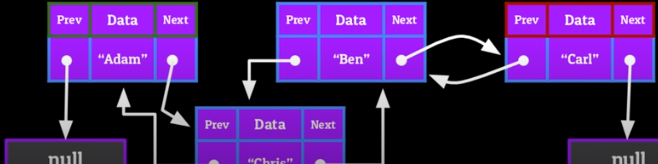
**Inserting into the middle of a Doubly Linked List:**

* **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.

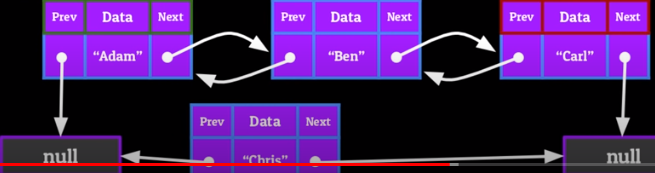
**Set the New Node’s NEXT to point to the Node after it, and it’s PREVIOUS to point to the Node before it.**



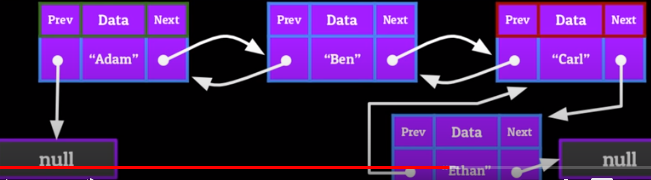
**Set the front node’s NEXT to point to the middle Node (new one) and set the end Node’s PREVIOUS to point to the middle Node (new one)**



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

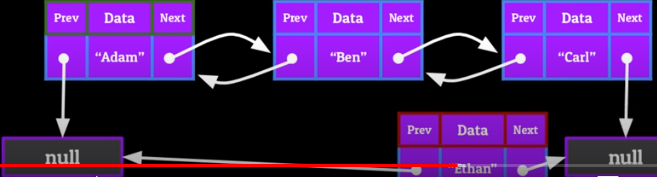
* **Front Node = node on left**
* **Middle Node = new Node we are inserting or deleting**
* **End Node = node on right**
* Set the FRONT NODE’s NEXT to point to the END NODE
* Set the END NODE’s PREVIOUS to point to the FRONT NODE
* Set both pointers of the MIDDLE NODE (one we are removing) to NULL
* 

**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.**

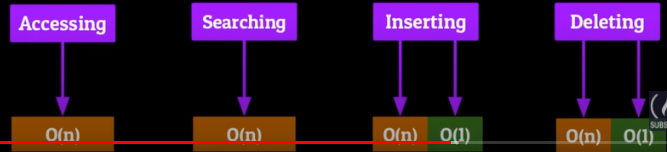
**Doubly Linked Lists – Time Complexity Equations:**

**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)**

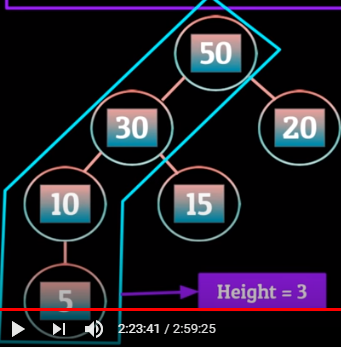
**Dictionary Time Complexity Equations:**

* **Hash Table Mini-Lesson:**

**Trees – Terminology and Visualization**

**Properties of the Tree:**

* **Height-**property of the Tree itself; Number of Edges on the longest possible path down towards a Leaf
* **Depth-** property of each individual node in the Tree; number of Edges required to get from that particular Node, to the root Node. In the picture below, 30 has a depth of 1 (**one edge away from Root Node**), but 15 has a depth of 2 (**two Edges away from Root Node)**

 **🡨-- 3 Edges, so height=3**

**Different Types of Trees:**

Regular Trees are great for storing hierarchical data, but their power can be greatly increased when you start tinkering with how the data is stored in them.

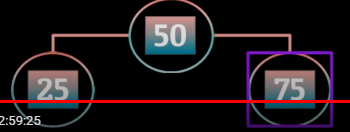
**By creating rules and regulations and what type of data can be stored in the Tree and where it can be stored, you can make the Tree more powerful.**

**Types of Trees:**

* **Binary Search Tree**
* **AVL Tree**
* **Red/Black Tree**
* **N-ary Tree**

**Binary Search Tree:**

* A Binary Search Tree is a variation on the standard Tree that has 3 restrictions on it to help organize data.
  + **Restriction 1:** A Node can have two Children at MOST
  + **Restriction 2:** For any given Parent Node, the Child Node to the left has a value **less than or equal to the parent**, and the Child Node to the right has a value **greater than or equal to the parent**
  + **Restriction 3:** No two Nodes can contain the same value

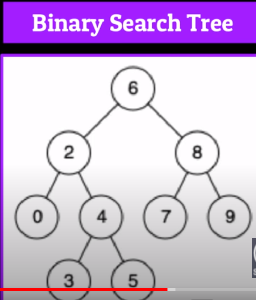
 **🡨 example of Restriction #2**

**The biggest advantage of Binary Search Trees, is that we can search through them in Logarithmic Time. (much faster than O(n))**

**Process:**

* Tell the computer to go left if the value we’re searching for is less than the current Node
* Tell the computer to go right if the value is greater than the current Node
* Binary Search Trees are really popular for storing **large quantities** of data that need to be easily searchable
  + Also translates to accessing, inserting, and deleting Nodes

**Example of Restriction #2:**

 **values >= parent are to the right, <= to the parent are to the left**

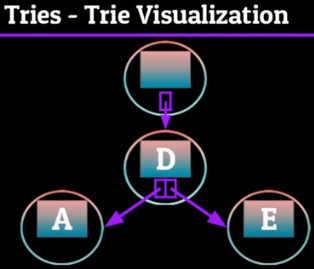
**Common Uses for Trees in Computer Science:**

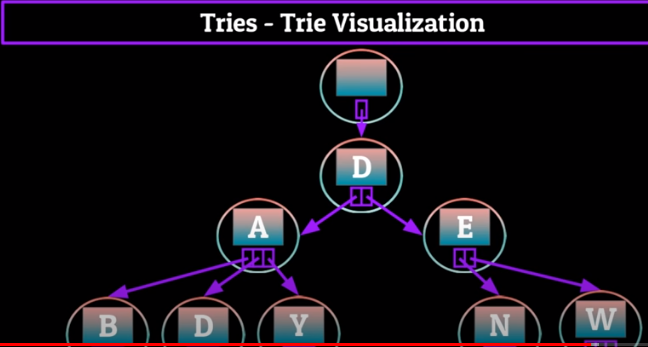
* **Hierarchical data** (file structure systems, family tree, company’s corporate structure)
* Again, when we **modify a Tree** and give it **rules** and **parameters**, it can become much more powerful than a simple Tree.

**Tries:**

* **A Trie** is a special **Tree** with **restrictions**
  + It is a **Tree-like Data Structure** whose **Nodes** store **letters of an alphabet** in the form of **characters**
  + We can carefully construct this Tree of Characters in a way that allows us to quickly retrieve words in the form of **Strings** by traversing down a Path of the Trie in a certain way

**Trie – Visualization:**

* Starts like a regular Tree, with a Root Node, but in this case the Root Node is empty
* Also stored in the Root Node is an array containing a set of references. At first, these references all point to Null, but can be slowly filled with references to other Nodes.
* First Child Node will have a letter as the Data in the Node, then references pointing to a new letter if “**D + new\_letter” are the beginning letters of any new word.**
* **So D will have a reference to “A”, because “Da” is the start of many English words (Dad, Dark, Date, etc.)**
* **D will not have a reference to “b” because there are no English words where the first two letters are “Db”.**
* The Node will go through the entire alphabet doing the same thing (no reference to C because no “Dc” words, same for “Dd”, “De” will have a reference, because there are words that begin with “De”.
* **The process will repeat for all of the New Child Nodes, but will ONLY have references to letters that combine with the Node and its Parent Node to continue to build words. See below the picture:**
* 
* **From the picture above, A would have references to B, D, Y, etc., because those letters combined with the Parent Node of A (D), continue to make words.**
* **There would be NO reference to A coming FROM A, because “Daa” is not a word/beginning of a word.**
* **See Below:**



**Pros of this Approach:** Depending on the Path you take down the Trie, you can create multiple words from many choices of Strings.

**Cons of this Approach:**  How does the computer know that you’ve reached the end of your desired word, and you don’t want to continue down the path? **Use a flag. Perhaps use a “.” as a flag.**

**Tries – Use Cases in Real Life:**

**1). Spell-check**

**2). Auto-complete**

Big programs like iOS or Google Docs don’t just store Tries with a few words, or even all the words that begin with a certain letter…..they usually store the entire English Dictionary.

Using the Trie in a Spell-Check or Auto-Complete program is a process of elimination. If you type S, it will start down that branch of the Trie, and ignore the other 25 branches.

Then, if you type a “u” to make “Su”, another 95% of possible solutions get deleted.

With each extra letter, the Trie gets narrowed down.

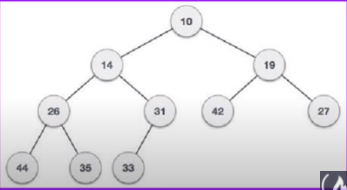
**Heaps – Introduction:**

* Think back to Binary Search Tree:
  + Each Node can have no more than 2 Children Max
  + Child to the left has a value <= the Parent Node
  + Child to the right has a value >= the Parent Node
  + No two Nodes can contain the same value

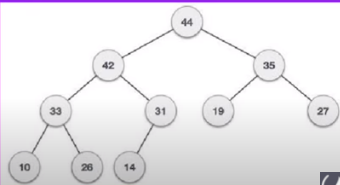
**A Heap:**

* A Heap is a special Tree where all Parent Nodes **compare** to their Child Nodes in some way by being **more** or **less** extreme
  + They will be either **greater than** or **less than** their Child Nodes
  + Whether the Parent is greater or less than its Child Nodes **determines where the data is stored**
  + It is usually dependent on the Parent Node’s value

**Min-Heaps:**

* A **Min-Heap** is a Tree where the value of the **Root Node** is less than every one of its Child Nodes
* This **must be true recursively** for all other Parent Nodes in the Tree (they must all always be less than their children).
* 

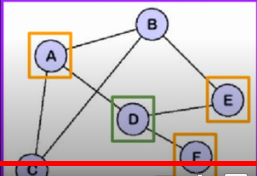
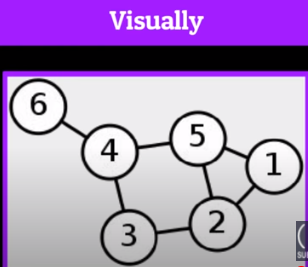
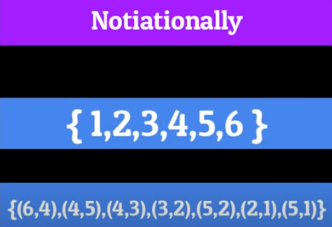
**Max-Heaps:**

* Exact opposite of Min-Heaps. Value at the root must be **greater** than the values of each of its Child Nodes
* Same **must be true recursively throughout the Tree. Example below:**
* 

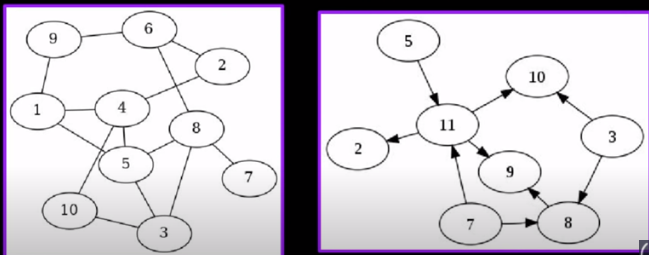
**Heaps – Implementation and Use Case:**

* Heaps are most commonly used in implementing HeapSort
  + HeapSort is a sorting algorithm that takes in a list of elements, builds them into a min or max heap, and then removes the Root Node continuously to make a sorted list.
* **Priority Queues:** 
  + An advanced Data Structure your computer uses to designate tasks and assign computer power based on how urgent the matter is

**The Graph:**

* Graphs are **pieces of information** and the **paths that run between them**
* It is a **nonlinear** Data Structure consisting of **Nodes** and **Edges**
  + There is a **finite** set of **Nodes**
  + **Nodes** are connected by the **Edges**
  + **Every Node** is connected to its surrounding Nodes, except that the final Node doesn’t branch off to any new Nodes, which makes sense…
* Graphs have **multiple starting points** and **multiple branches.**
* 
*  
* **The above two examples shows what a graph looks like when we visualize it (left) and when we write it out in notation (right)**
* In the example above, the first set of curly brackets represents **the number of every Node (or vertice) in the Graph.**
* The bottom set of tuples in curly brackets is called an **Edge Set**. It represents all of the **Node/Edge** relationships in the Graph.
* For example, 6 shares an Edge with 4 so (6, 4). 4 also shares Edges with 5 and 3, so (4, 5) and (4,3), 2 shares Edges with 3, 5, and 1 so (3, 2), (5,2), (2,1)
* If an Edge connects more than one Node/Vertice, it is said to be **adjacent** to the others. So in the example above, 5 is adjacent to 4, 2, and 1 Nodes.

**Directed vs. Undirected Graphs:**

* **Undirected Graph:**
  + A graph in which the direction you traverse the nodes **ISN’T** important
  + Usually indicated by a lack of arrows
  + Any of the above Graphs are Undirected Graphs
* **Directed Graph:**
  + A graph in which the direction you traverse the Nodes **IS** important
  + Usually indicated by arrows that point to which Nodes a certain is allowed to traverse to
  + Edges can point both ways, but don’t have to
* 
* **In the example above, the picture on the left shows an Undirected Graph, where the picture on the right shows a Directed Graph**

**Graphs – Cyclic vs. Acyclic:**

* **Cyclic Graphs:**
  + A cyclic graph is a Node that contains a **path** from **at least** one Node back to itself
  + All **Undirected Graphs** are cyclical
* **Acyclic Graphs:**
  + An acyclic graph is one that contains **no path** from any given Node back to itself
  + These can really only be **Directed Graphs** because they aren’t all connected together to start.

**Weighted Graphs:**

* The process of associating a **numerical value** with each **Edge** (cost)
* Each weight represents some property of the information you’re trying to convey.

**Types of Graphs:**

* **Undirected Cyclical Heaps with Weighted Eges** can be used through **Dijkstra’s shortest path algorithm**
* Compiles a **list** of the shortest possible paths from that source vertex to all other Nodes in the graph
* This graph and algorithm is used in Google Maps, IP Routing, and potentially even telephone services
* **Unweighted Cyclical Graphs (Directed and Undirected)** are used in the “follower” system of a majority of social media networks:
  + **Facebook, Snapchat, Instagram, Twitter**