* How single node relational databases operate
* Searching is the most common operation performed
* In SQL the select statement is arguably the most complex statement
* Baseline for efficiency is linear search - start at beginning of a list and proceed element by element until you find what you’re looking for or you reach the end of the list - should take O(n) time worst case
* Record - a collection of values for attributes of a single entity instance - row of a table
* Collections - a set of records of the same entity type - a table
* Search key - a value for a n attribute from the entity type
* If each record takes of x bytes of memory then for n records we need n\*x bytes of memory
* So then we have 2 different ways to store in memory (ram)
  + Contiguously allocated list
    - All n\*x bytes are allocated as a single chunk of memory
    - Array
    - Pros and cons
      * Arrays faster for random access but slow for inserting anywhere ut the end
      * To find the location of a specific element would be a single mathematical operation – easy
      * But if you wanted to add something you would have to change location of all items in the array
  + Linked list
    - Linked list
    - Each record needs x bytes + additional space for 1 or 2 memory addresses—for the linking
      * X + the 2 memory addresses is the space it takes up
    - Individual records are linked together in a type of chain using memory addresses
    - Pros and cons
      * Linked lists are faster for inserting anywhere in the list but slower for random access
      * If i know i want to insert something I don’t have to move anything down i just have to add it and adjust some memory address pointers
* Binary search
  + Input: array of values in sorted order - target value
  + Output - the location index of where target is located or some value indicating target was not found
  + Is recursive in nature but this is gonna bow shit up if its a large dataset —- has to be REALLY large dataset – if it’s around 30000 recursive calls
  + Can also have an iterative binary search
  + Runtime O(logn)
* Linear search - best cast is 1, worst case O(n)
* Binary search - best case 1, worst case O(logn)
* ^^^^^ only true for applying on array
* Can you apply binary search on linked list? - yes but is VERY inefficient - ends up being less efficient than linear search
* Database searching
  + Assume data is stored on disk by column id’s value
  + Most relational databases store in order of primary key
  + Searching for a specific id is very fast
  + But what if we need to search for a specific specialVal value? – so then we can’t perform by binary search we need to perform by linear search - slow af
  + Can’t store data on disk by both id and specialVal at the same time — unless you want to duplicate the date which is rlly space inefficient
  + So we need an external data structure to support faster searching specialVal than a linear search
  + What r the options:
    - An array of types - sorted by specialVal
      * But every insert into the table would be like inserting into a sorted arry – slow
    - A linked list of tuples sorted by special val
      * But inserting into the table would theoretically be quick to also add to the list
    - Is there a different data structure besides an array or linked list that has quick speed AND quick insert time
      * YES
      * A binary search tree
      * Every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent
      * Binary search tree are the basis of the primary data structure that is used for indexing in your typical database system
  + Creating /inserting into a binary search
    - Given 23 17 20 42 31 50
    - There will always be a reference that will allow you to reference the root of a tree
    - Tree traversals
      * PreOrder
      * PostOrder
      * InOrder
      * LevelOrder\* - need for hw
        + Level order traversal - final order would be 23 17 43 20 31 50
        + When processing 23 also store what you want to store as the children of 23
        + Have output and additional temporary datastructure
        + Start at 23 - add to output, then add 17 and 43 to additional structure
        + Then process 17 the same way then add its left child and right child to data structure
        + Then rpovess 43 then add its left child and right child to data structure
        + What is this data structure called - a queue
        + Python ddoesn’t have in its collections library a queue it instead has a special version called a deque - a double ended queue - can insert and remove from the front and the back
  + BinaryTreeNode
    - Value
    - Left: bintreenode
    - Right: bintreeNode

Class BinaryTreeNode(self, value, left = None, right = None)

Value: int

Left: BinTreeNode

Right: BinTreeNode

So if i wanted to create 23

Root - BinaryTreeNode(23) – will create BinTreeNode with 23 in value and none and none in left and right

# then to assign left node of 23 — this is how you make the link

Root.left = BinaryTreeNode(17)

Root.right = BinaryTreeNode(43)

# then to add 20:

Root.left.right = BinaryTreeNode(20)

* Goal is to minimize height of the tree – have it balanced
* AVL Tree
  + Approximately balanced binary search tree
  + Maintains a balance factor in each node - value that represents how balanced is the tree if that is the root of the tree
  + AVL balanced property:
    - |height(LeftSubTree) -height(RightSubTree)| <= 1
  + Is a self balancing binary search tree
  + Way it’s constructed
    - Similar to bst but with logic added in so that values are inserted in a way that prevents a tree from becoming imbalance
    - Tree can only become imbalance on the path of the most recently inserted node to the root
      * If thinking about in class example where you’re inserting 50 you go up to 60 and check if there’s imbalance and then go up to 40 and check for imbalance
      * One insertion if it causes an imbalance will always be fixed by one node’s rebalancing — if 10–5 and inserting 7 then put 7 in the middle and make 5 and 10 the children
      * Height is O(logn)
    - 4 cases of Imbalance and how to imbalance
      * LL — inserting into the left subtree of the left child of the node of imbalance
        + Can be done in constant time
        + Single rotation
      * LR —- inserting into the right subtree of the left child of the node of imbalance
      * RL —- inserting into the right subtree of the left child of the node of imbalance
      * RR —-- inserting into the right subtree of the right child of the node of imbalance