# 1/6

Monday, January 6, 2025 9:13 AM

Given how much data is being processed by minute for a lot o fbig tech companies, there has to be special methods of storing and retrieving that data

Website: Large Scale Information Storage and Retrieval: DS 4300 – Spring 2025
Campus wire: Campuswire

Goal of this class is to learn:
- Understand concepts and limitations of RDBMS
- Understand data replication and distribution for general database usage
- Learn the use cases of models for NoSQL database systems, such as document-based, key-value, and graph based (vector databases !?!?)
- Implement data engineering and big-data AWS systems

How do you set up your data on different databases? E.g. not all data on Instagram is on a single data center
- A data center in Western Europe is not going to be the main host for data in America

Homeworks due Tuesday, but 3% bonus if you submit Sunday night
One free no-questions-asked 48 hour extension, must DM him on Slack
Gradescope (homeworks) and Github (coding) will be the main method of submitting homework
- Only submit PDFs unless otherwise requested
- Regrade requests must be submitted within 48 hours

Get ready to read a lot of system documentation (they all have different query languages……)

Tentative Topic List:
- Data storage and retrieval on data structures level
- How far could you go with a relational model (SQL?), such as consistency, durability, general protection but costly performance. How can you relax these constraints?
- NoSQL Databases
- Data distribution and replication
- Distributed SQL databases and Apache Spark
- AWS and other big data tools

TODO:
- Update docker and operating system
- Get Datagrip or Dbeaver (database access)
- What are the basics of Docker
- AWS account (need to provide credit card info :skull:, privacy.com)
- First homework already up

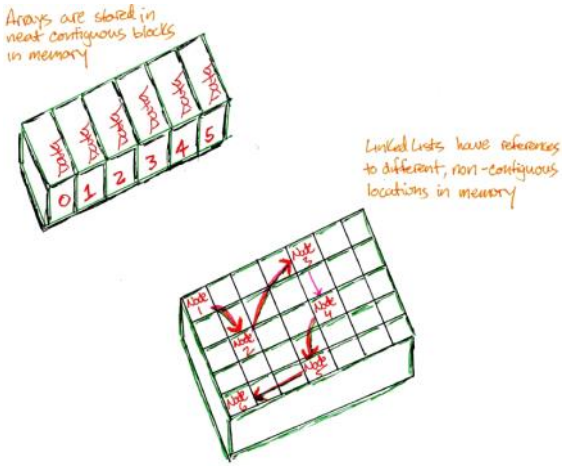
# 1/8

Wednesday, January 8, 2025 9:09 AM

**Foundational Theory For Single-Node Databases**

Searching
- Very common operation needed for a database system
- This is accomplished by SELECT in SQL, very versatile
- Baseline for efficiency is Linear Search (i.e. search each element in a list one by one)
    - Best case is if the element is first in the list, worst case is last

List of Records
- If a record takes up x bytes of memory, then we need nx bytes for n records
- In a **contiguously allocated list**, we can store all nx bytes as a single "chunk" of memory (RAM)
- In a **linked list**, records would need x bytes + space for 1~2 memory addresses which would link records together




Arrays are faster for random access, but slow for inserting anywhere but the end
    - This is because we would have to move all of the subsequent data over if we insert in the middle
    - Arrays don't exist in Python's base data structures (Numpy arrays work however)

Linked Lists are faster for inserting anywhere in the list, but are slower for random access
    - To get element 100, we need to move through the first 99 elements, making searching linear time

Binary Search
Input: Sorted array and a target value
Output: The location of where the target is located, or a flag showing the target was not found

| Binary Search |
| --- |
| def binary_search(arr, target)<br>  left, right = 0, len(arr) - 1<br>  while left <= right:<br>  mid = (left + right) // 2<br>  if arr[mid] == target:<br>  return mid<br>  elif arr[mimd] < target:<br>  left = mid + 1 |

```
  else:
    right = mid
return False
```

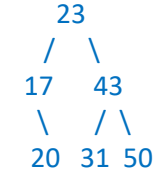
Linear Search:
- Best Case: Target at first element, only 1 comparison
- Worst Case: Target not in the array, n comparisons
    - Would be O(n) time

Binary Search
- Best Case: Target at mid, 1 comparison in loop
- Worst Case: Target not in array, $\log_2 n$ comparisons
    - Would be O(logn) time

How does this apply to Database Searching?
- Assume data is stored on the disk by a column ID
    - This makes searching for a specific ID fast, as it is sorted
- If we want to search for a value associated with that ID, only option would be a linear scan
    - This is because those values would be unsorted
    - We can't have it sorted by both ID and value, as data would have to be duplicated, which is space inefficient
- This brings the need of having an external data structure that can search for values faster than linear scan

Possible Solutions
- An array of tuples (val, Index) sorted by val
    - Binary search can quickly find the value, and by association, its index
    - Insertion would still be slow
- Linked list of tuples sorted by special val
    - Search would still take linear time, even if insertion is quick
- Binary Search would be the best compromise between searching and inserting time
    - Every node in the left subtree is less than its parent, and every node in the right subtree is greater than its parent
    - Each node has a maximum of 2 children
    - Searching is as fast as a sorted array, insertion is as fast as a linked list

Binary Tree Example
Suppose we want to insert these numbers in order: 23, 17, 20, 42, 31, 50
Resulting tree would be:




In a binary search tree, we will always have a way to reference the root (top) of the tree (23)

There are methods of traversing a tree without having to scan each element one by one
- Pre order, post order, in order, **level order**
- Level order will read each "level" one by one
    - Level order output for this example is 23, 17, 43, 20, 31, 50

- Level order can be done via breadth first search
    - First, read in the root and then add the children to an queue (left one first)
    - Then, read in the first value in the queue, remove it, and add its children (if there is any) to the array
    - Repeat until the queue is empty
- Python has a deque (double ended queue, meaning you can insert at either end) that can be used for this

Implementation

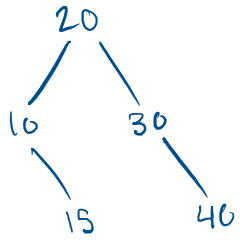
- We would need a BinaryTreeNode class (self, value, left=None, right=None)

```
root = BinaryTreeNode(23)
root.left = BinaryTreeNode(17)
root.right = BinaryTreeNode(43)
root.left.right = BinaryTreeNode(20)
…
```

1/9

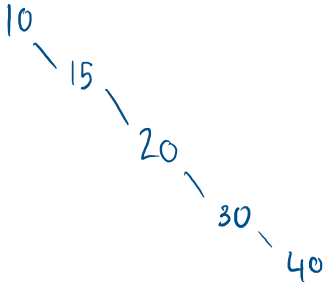
Thursday, January 9, 2025 9:12 AM

Given these values: 20, 30, 10, 15, 40
The resulting binary tree would be:




height = 3

But if the order was 10, 15, 20, 30, 40, then the tree would be

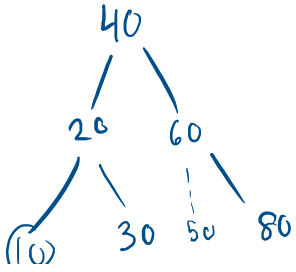



height = 5

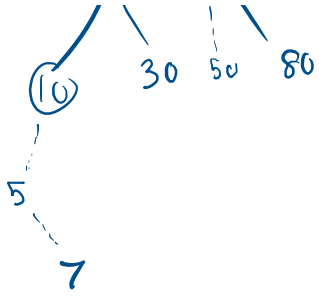
We want to create "balanced" binary trees (rows are filled up as much as possible to minimize height → minimize # of comparisons

AVL Tree: An approximately balanced binary search tree that maintains a balance factor to stay relatively balanced

AVL balance property: $|h(LST) - h(RST)| \leq 1$
i.e heights of left & right tree difference is no more than 1




Inserting 50 will not cause any imbalance
Inserting 5 still keeps balance
Inserting 7 does cause imbalance
↳ 10 is the node of imbalance



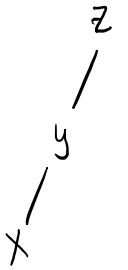

Inserting 7 also caused imbalance
↳ 10 is the node of imbalance

We can use an algorithm to rebalance the subtree starting at the node of imbalance

This won't affect the balance at other subtrees

There are 4 cases of imbalance

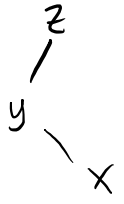
① LL insertion




Inserting into the left subtree of the left child of the node of imbalance (z)

$$\begin{cases} x < y \\ x < z \end{cases}$$

② LR case

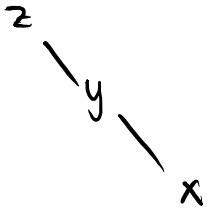



$$\begin{cases} x < z \\ x > y \end{cases}$$

③ RL case

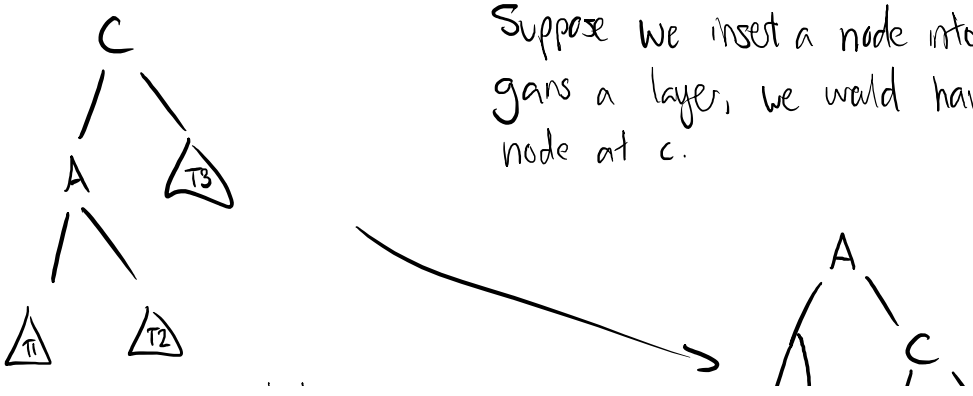

$$\begin{cases} x > z \\ x < y \end{cases}$$

④ RR case

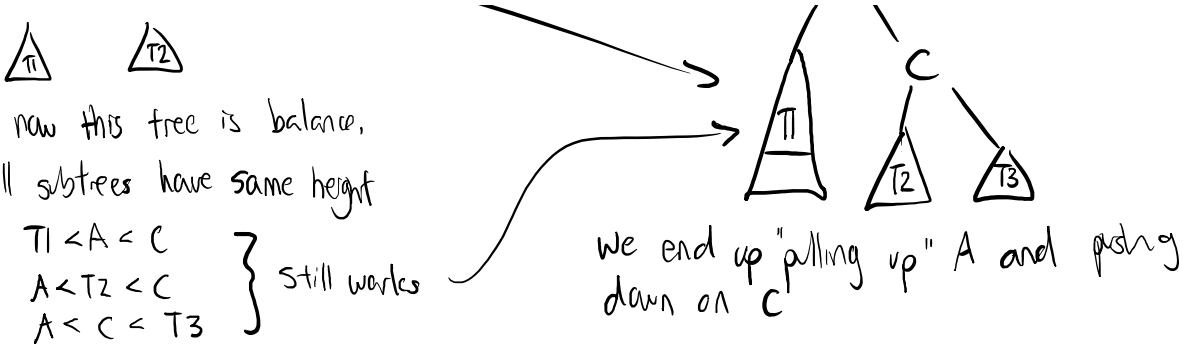



$$\begin{cases} x > z \\ x > y \end{cases}$$

How can we rebalance case 1 (LL)?




Suppose we insert a node into T1 such that it gains a layer, we would have an imbalanced node at c.




Right now this tree is balance, and all subtrees have same height

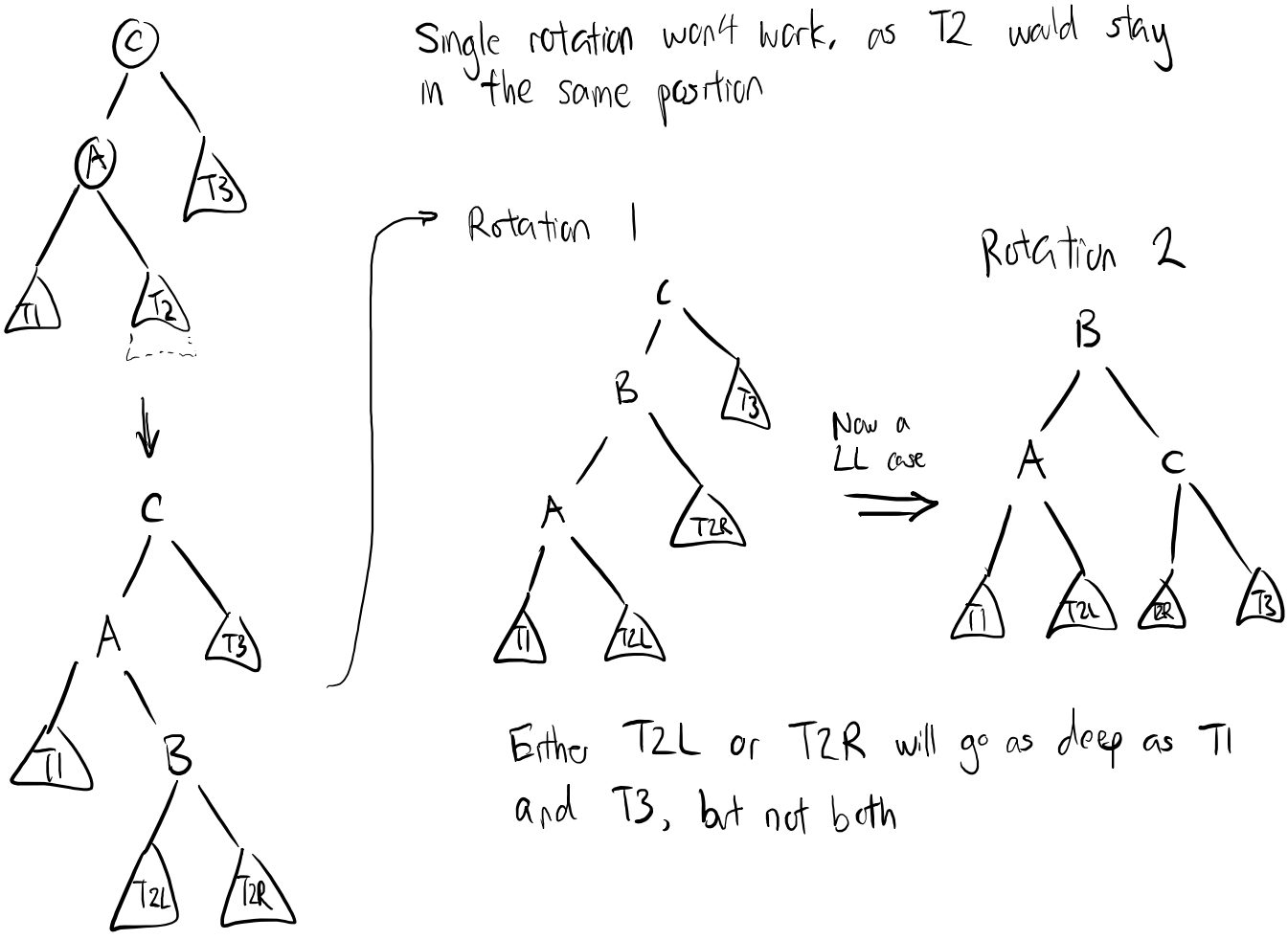
$T1 < A < C$
$A < T2 < C$
$A < C < T3$

} still works

we end up "pulling up" A and pushing down on C

This process is called a single rotation. The 4th case (RR) can be solved in a similar way by mirroring the process

How can we rebalance case 2 (LR)?



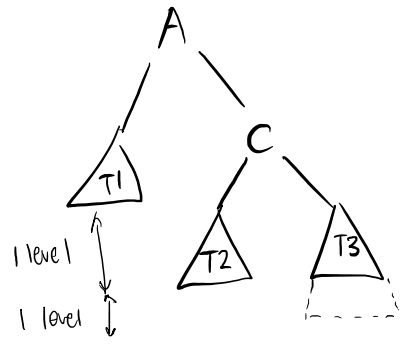

Single rotation won't work, as T2 would stay in the same position

Either T2L or T2R will go as deep as T1 and T3, but not both

1/13

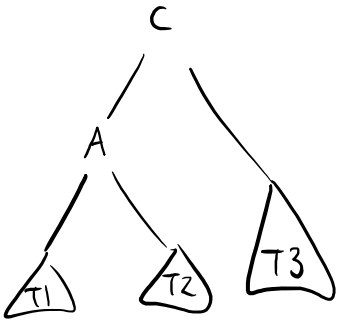
Monday, January 13, 2025 9:19 AM

Recall: Case 4 scenario (RR)




A is node of inbalance

This is just the mirror of case 1




Pseudocode:

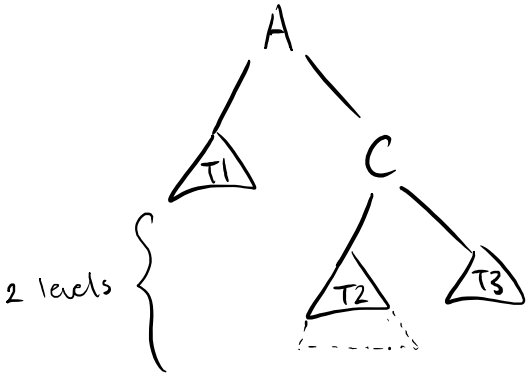
Let x = A

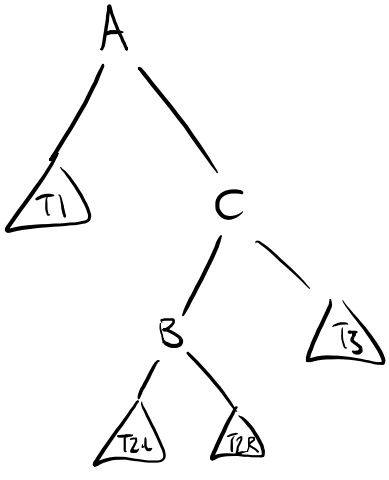
Let y = C

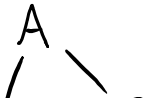
Let x.right = y.left

Let y.left = x

Case 3 scenario: RL




Pseudocode:

Let x = A

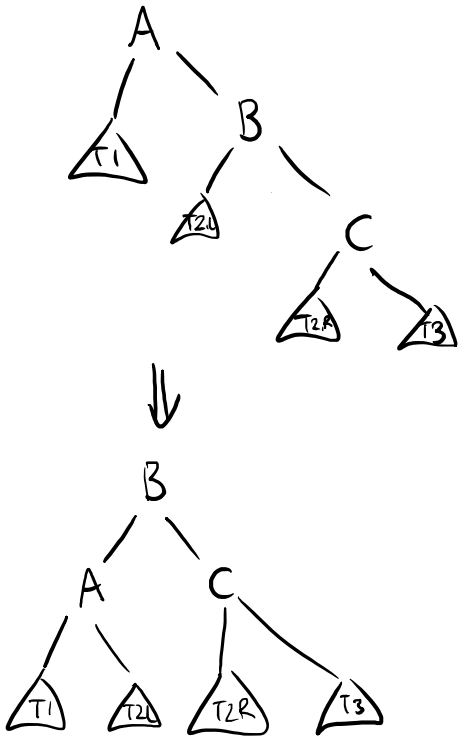
Let y = C

Let x = A

Let y = C

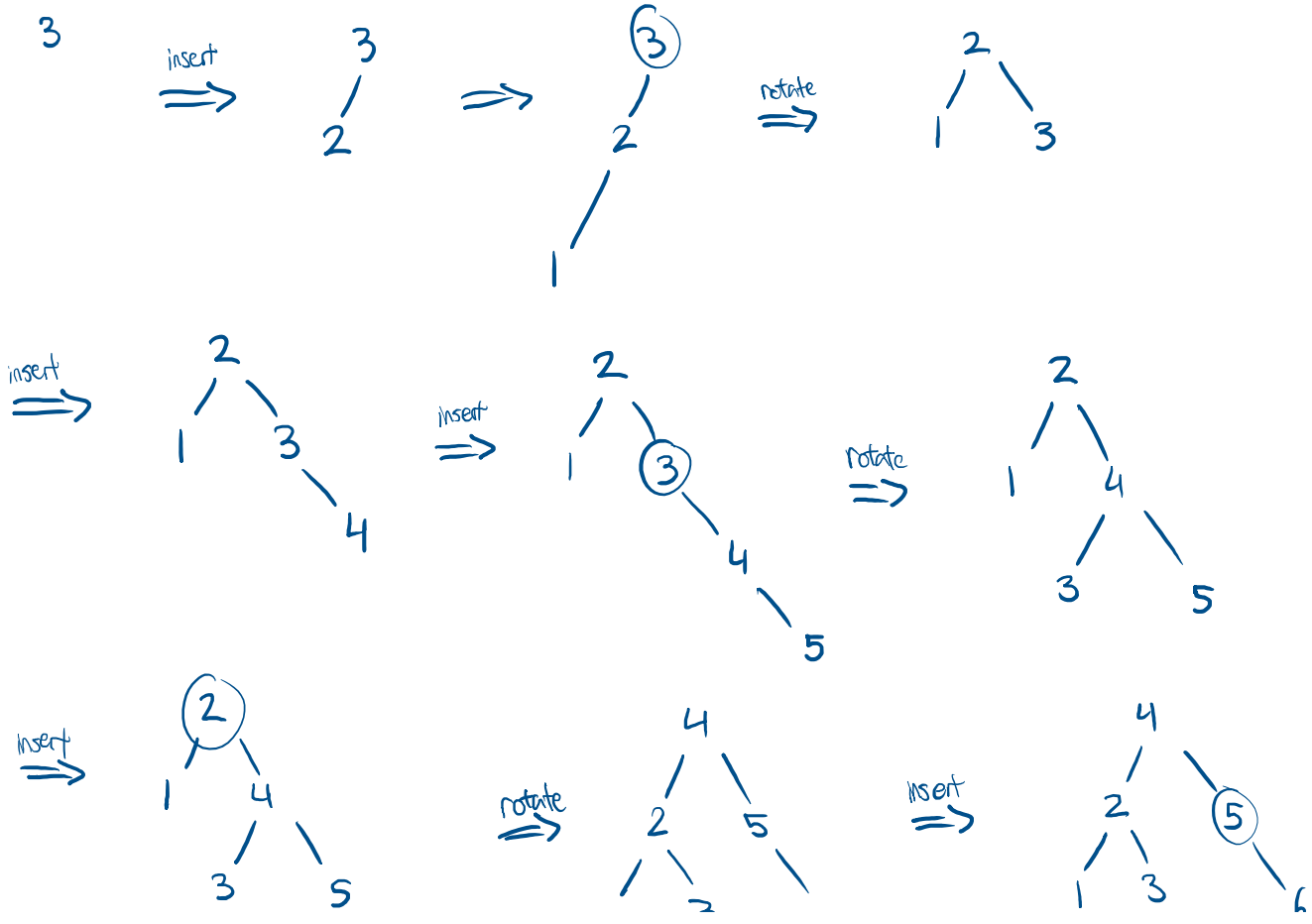
Perform single rotation on B-C

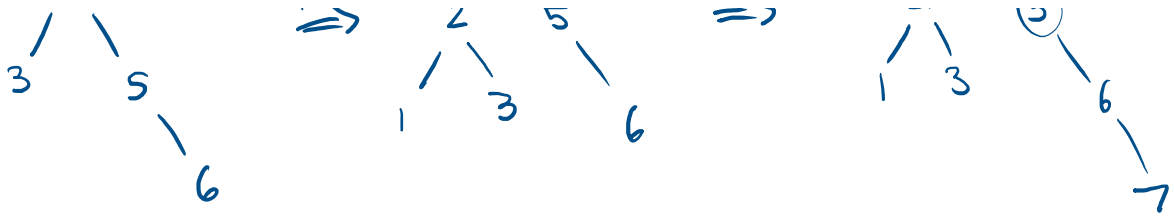
Perform single rotation on B-A

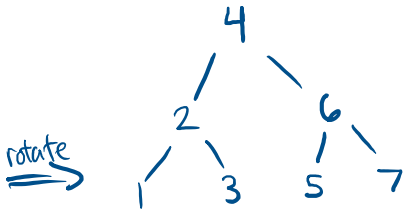



Create a Binary tree given these numbers, in order:

3, 2, 1, 4, 5, 6, 7







Final AVL Tree

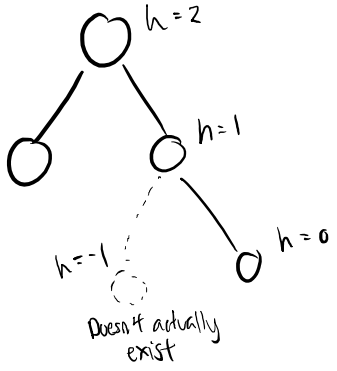
When balancing, check recursively
i.e., check for inbalances layer up

For efficiency, we can store L/R height or balance as a variable for each node

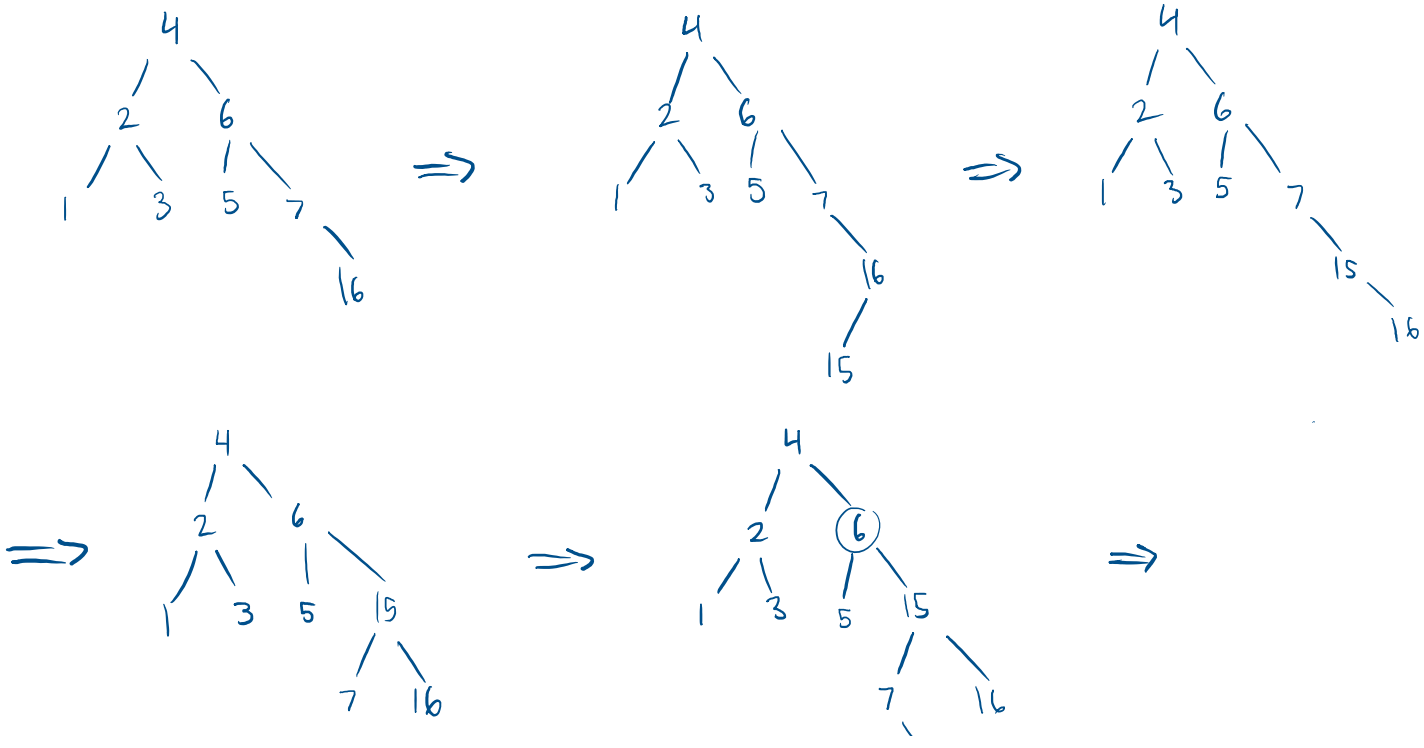
New node initialize as height = 0
Then when going back up, find maximum height of either left & right and add 1

Make sure to update heights after rotation and to check/correct imbalance before updating heights

Height of an empty subtree is -1




What if we insert 16, 15, 14 in that order?



14

# 1/15

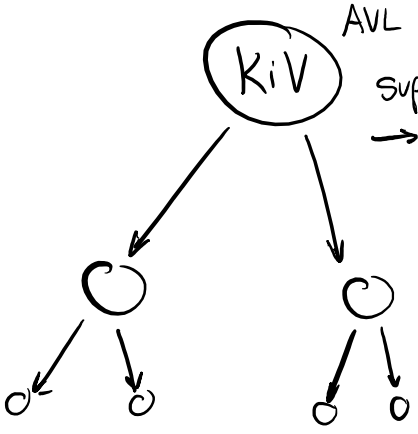
Wednesday, January 15, 2025 9:15 AM

## Memory hierarchy

CPU ← number crunching unit
Registers ← very fast but expensive
L1 Cache ← bigger but slower, cheaper memory
L2 Cache ← follow pattern
RAM (nanoseconds)
SSD / HDD (milliseconds) ← Lots of storage, persistent, but slow

To have a fast database systems, we want to minimize HDD/SSD use

64 bit integer → 8 bytes
2048 byte block size




AVL Tree

Suppose key, value are 64-bit ints
→ 32 bytes

Possible that these nodes are stored on different blocks
→ 7·32 bytes of data, but...
Need to read up to 7·2048 bytes of block

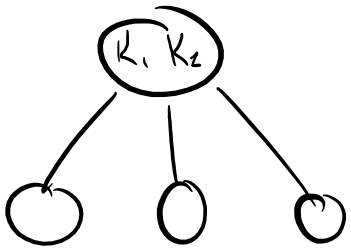
In a perfectly balanced tree, we can read just $\log_2(n)$ nodes to get a desired node if we know location

We want to make better use of the 2048 bytes of the block.

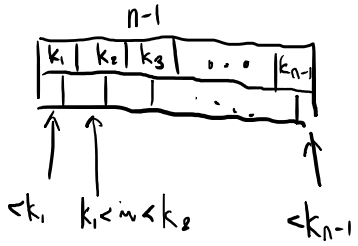
Given a sorted array of 128 integers

Worst case binary search is <u>much</u> faster than a single additional disc access

Worst case binary search is <u>much</u> faster than a single additional disc access (i.e looking through another block)




We can use $B^+$ Tree to maximize # of values in disc blocks




With n-1 keys, we can have n children

With 256 keys, we can store 258 nodes in 2 layers (257 + root)

compared to only 3 nodes in AVL in 2 layers

Much faster because this is binary search in RAM rather than linear search in secondary storage

Nodes will always be at least half full

# 1/16

Thursday, January 16, 2025 9:14 AM

Practical 1 Notes:

```
from typing import List, Optional, Any

Class BSTNode:
    def __init__(self, key: Any):
        self.key: Any = key
        self.values: List[Any] = []
        self.left: Optional['BSTNode'] = None
        self.right: Optional['BISTNode'] = None

    def add_value(self, value: Any) -> None:
        self.values.append(value)

    def get_values_count(self):
        return len(self.values)
```

BST's and AVL Trees should be thought of as inherently recursive structures
- This is because each tree can be thought of as a root node connected to two sub-trees

Functions calls are not free in terms of computation time/effort. If statements are faster

AVLTreeIndex rotation example (don't use this)

```
def _rotate_left(self, x: AVLNode) -> AVLNode:
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2

    ht_x_left = (0 if not x.left else x.left.height)
    ht_x_right = (0 if not x.right else x.right.height)
    x.height = 1 + (ht_x_left if (ht_x_left > ht_x_right) else ht_x_right)

    ht_y_left = (0 if not y.left else y.left.height)
    ht_y_right = (0 if not y.right else y.right.height)
    y.height = 1 + (ht_y_left if (ht_y_left > ht_y_right) else ht_y_right)

    return y
```

1/22

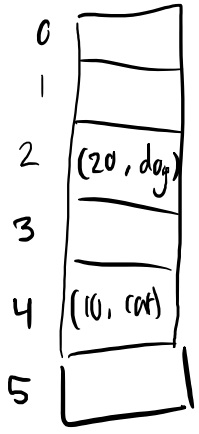
Wednesday, January 22, 2025 9:17 AM

# Hashtables (pretty easy to implement)

↳ This is the dictionary data structure

We are going to create our own hashtable structure

our data: (10, cat), (20, dog), (15, bird)




Apply a hashing function that takes key
→ Will usually use table size m
→ $\lambda$ load factor = $\frac{n}{m}$ ← number of inserts

(mod m always included to ensure value will be inserted)

For example: h(k) = k mod 6

h(10) = 4, therefore cat goes @ 4

h(20) = 2, therefore dog goes @ 2

Note that the hash function is completed in a constant amount of time (i.e. it doesn't depend on n or m)

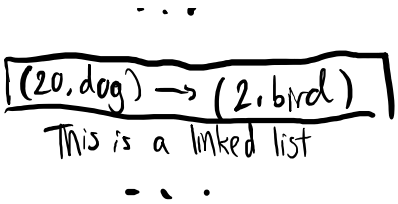
Therefore, insertion is constant time (assuming there are slots)
→ but what happens when we insert into a cell that already has a value associated with it?

In the above example, (20, dog) & (2, bird) have the same destination
→ Solution: Create a list of everything that maps to that location

...

[(20, dog) → (2, bird)]

In project, key = word
value = file location



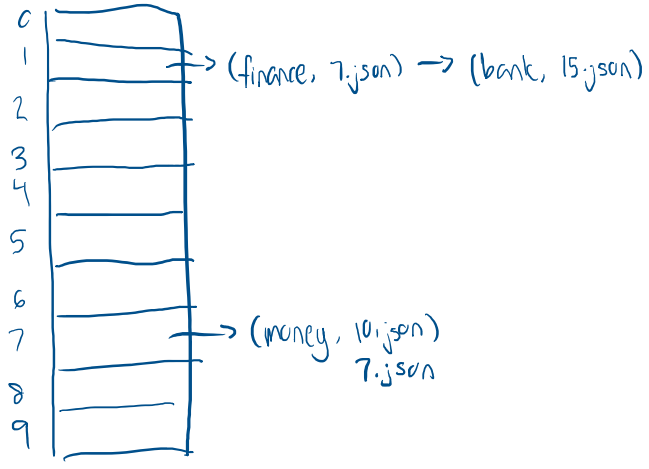

In project, key = word
value = file location

How do we determine the table size?

The bigger the table, the higher the probability that we insert into a location with a low amount of items

We want $\lambda < 0.9$ to avoid having a table that is too small
→ Start with a big table (100k)
Ideally, longest chain would be 5 KV pairs (but you don't have to track this)
Searching for a key-value pair in this scenario is ≈ constant time

A hash table should also have good dispersion
→ i.e. the values are relatively spread out in the hash table




h(x) mod
381 → 1
(finance, 7.json)

767
(money, 10.json)

951
(bank, 15.json)

767
(money, 7.json)

1/23

Thursday, January 23, 2025 9:09 AM

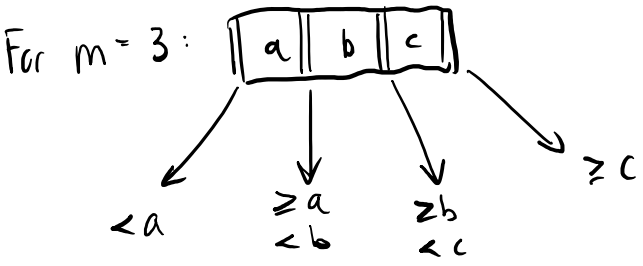
# $B^+$ Trees: Optimized for disc-based indexing

→ Minimizing disc access for indexing ↳ Regular B trees more in-memory

A $B^+$ tree is an m-way tree with order M

M → maximum of keys in each node

M+1 → max children of each node




Properties:

→ All nodes (except the root) must be ½ full at minimum

→ Root node does not have to be half full ↳ based on children

→ Insertions are done at the leaf level

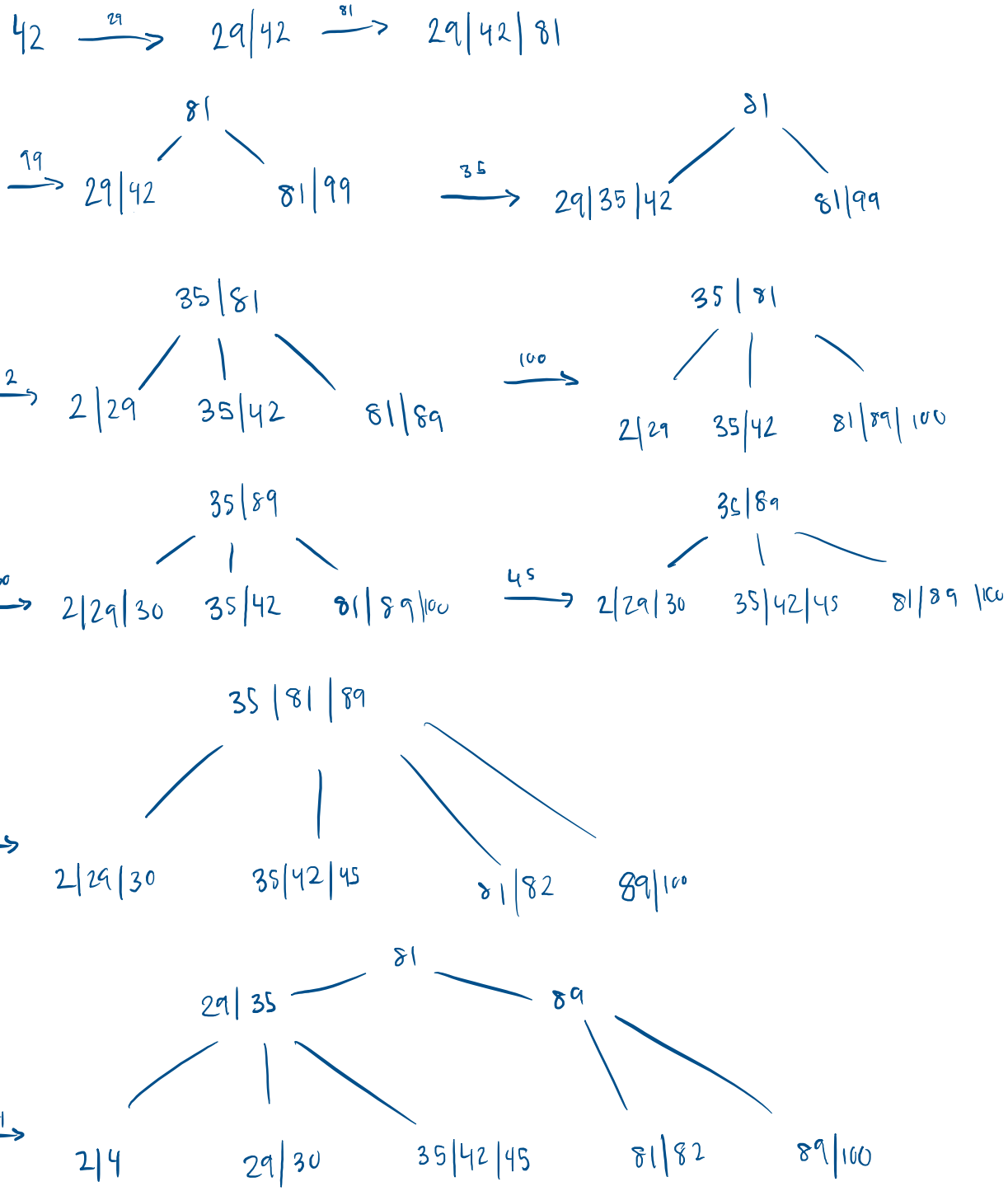
→ Leaves are stored as a DLL

→ Keys in nodes are kept sorted

→ Leaf nodes store actual values, internal nodes just store keys & pointers

Insert 42, 29, 81, 99, 35, 2, 100, 30, 45, 82, 4

Into $B^+$ tree of m = 3


42
29
29|42
81
29|42|81
81
99
29|42
81|99
81
35
29|35|42
81|99
35|81
2
2|29
35|42
81|89
35|81
100
2|29
35|42
81|89|100
35|89
30
2|29|30
35|42
81|89|100
35|89
45
2|29|30
35|42|45
81|89|100
35|81|89
82
2|29|30
35|42|45
81|82
89|100
81
29|35
89
4
2|4
29|30
35|42|45
81|82
89|100

# 1/27

Monday, January 27, 2025 9:13 AM

Benefits of the Relational Model
- Mostly standard model + query language
- Atomicity, Consistency, Isolation, Durability (ACID)
- Works well with highly structured data + can handle large amounts of data
- Well understood + lots of tooling available

Ways that a RDBMS increases efficiency:
- Indexing
- Directly controlling storage
- Column oriented storage vs row oriented storage
- Query optimization
- Caching / prefetching
- Materialized views
- Precompiled stored procedures
- Data replication and partitioning

Transaction - a sequence of one or more of the CRUD operations performed as a single, logical unit of work
- Either the entire sequence succeeds (COMMIT) or the entire sequence fails (ROLLBACK or ABORT)
- Helps ensure data integrity, error recovery, concurrency control storage, and simplified error handling

Atomicity: Transaction is treated as an atomic unit - either all or nothing is done
Consistency: Transaction takes a database from a stable state to another stable state (where all data meets integrity constraints)
Isolation: Two transactions T1 and T2 are executed at the same time but cannot affect each other
    - If T1 is reading the same data that T2 may be writing, could result in dirty read, non-repeatable read, phantom read
        - Dirty read: T1 can read a row that has been modified by T2 that hasn't committed
        - Non-repeatable Read: Two queries in T1 execute a SELECT and get different values because T2 had committed a change
        - Phantom reads: When t1 is running, T2 adds / deletes rows from a set T1 is using
Durability: Once a transaction is completed and committed, its changes are permanent (even in system failure)

# 2/3

Monday, February 3, 2025 9:13 AM

Distributed DBs and ACID - Pessimistic Concurrency
- ACID transactions
    - Data safety
    - Considered a pessimistic concurrency model because it assumes one transaction has to protect itself from othe transactions (i.e. if something can go wrong, it will)
    - Conflicts are prevented by locking read / write until a transaction is complete

Optimistic Concurrency: No locks on data on reading/writing, assumes conflicts are unlikely to occur, and if there is a conflict, everything should be okay
- Add last update timestamp + version number. Read them when changing, then, check at the end of transaction to see if any other transaction has caused them to be modified

Low conflict systems (backups, analytical dbs)
- Read heavy systems
- Conflicets that arise can be handled by rolling back / rerunning a transaction
- Good for optimistic concurrency

High conflict systems
- Rolling back / rerunning transactions that encounter a conflict are less efficient
- Locking scheme (pessimistic) would be preferable

"NoSQL" stands for Not Only SQL (but sometimes thought of as non-relation DBS)
- Designed to relax some of transactional requirements

CAP Theorem:
- In a highly distributed system, you can have 2, but not 3 of the following:
- Consistency: Every user of the DB has an identical view of the data at any given instant
- Availability: In an event of a failure, the database system remains operational
- Partition Tolerance: The database can maintain operations in the event of the network's failing between two segments of the distributed system
- CA: System always responds with latest data and every request gets a response, but may not be able to deal with network partitions
- CP: If system responds with data from the distribution system, it is always the latest, otherwise the request is dropped
- AP: System always responds, but may not be absolute latest data

BASE (alternative to ACID)
- **B**asically **A**vailable (guarantees the availability of the data, but response can be "failure" or "unreliable" if data is in an unstable state. Should work most of the time, however.
- **S**oft state (state of system should change over time to get eventual consistency. Replicas may not be consistent)
- **E**ventual consistency (System should eventually become consistent after read/write ends)

Key Value Stores
- Data model is extremely simple, lends to simple CRUD ops and API creation
- Data model is quick, as retrieving a value based on key is O(1), in-memory
- Data model is very scalable horizontally (add more nodes)

When to use KV DBs?

- EDA/Experimentation Results: Store intermediate results from data preprocessing or testing results without prod db
- Store Features: Used for model prediction if they appear in models frequently
- Model Monitoring: Store key metrics of a model for real-time monitoring

Redis (**Re**mote **Di**rectory **S**erver)
- Open source, in memory database
- Primarily a KV store, but can be used for Graph, Spatial, Full Text Search, Vector, and Time Series models
- Can be very fast in a properly added architecture, but cannot handle complex data
- Supports durability of data by essentially saving snapshots to disk at specific intervals

Redis Data Types:
- Keys are usually strings, but could be any binary sequence
- Values can be strings, linked lists, sets (unique unsorted strings), sorted sets, hashes (string -> string), geospatial data

Redis provides 16 databases by default (0 to 15, no other name associated)
- Direct interaction with Redis is through a set of commands related to setting and getting k/v pairs (and variations)
- Many languages libraries available as well

Commands:
SET ds4300 "I Love AVL Trees!"
KEYS *
GET cs3200
DEL ds4300

Foundation Data Type: String
- Sequence of bytes - text, serialized objects, bin arrays
- Maps a string to another string
- Use caches include frequently accessed HTML/CSS fragments, config settings, token management

# 2/5

Wednesday, February 5, 2025 9:20 AM

SET someValue 0
INCR someValue # Increase by 1
INCRBY someValue X # Increase by X

Value of KV entry is a collection of field-value pairs
- Limits of field / value pairs is 2^32 - 1, practical limit is memory

Hash Commands Syntax:
HSET bike1 model Demios brand Ergonom price 1971
HGET bike1 price
HGETALL bike1
HMGET bike model price weight
MINCRBY bike price 100

Value of KV Pair is linked list of string values

Sequential data structure of linked nodes instead of contiguously allocated memory)
- Each node points to next element of the list, except the last one (points to null)
- O(1) time to insert value at front or end.

List commands:
Queue like:
LPUSH bikeRepairs bike1
LPUSH bikeRepairs bike2
RPOP bikeRepairs
RPOP bikeRepairs

Stack Like:
LPOP bikeRepairs

LLEN mylist
LRANGE key start stop # LRANGE mylist 0 3

Set types are for an unordered list of unique elemnts
SADD ds4300 "Sam"
SISMEMBER ds4300 "Mark"
SCARD ds4300

# 2/6

Thursday, February 6, 2025 9:13 AM

Import redis
Redis_client = redis.Redis(host='localhost', port=6379, db=2, decode_responses=True)
- Port is the port mapping given when you created the container
- db is the database 0-15 you want to connect to

Redis_client.set('clickCount', 0)
Val = r.get('clickCount)
Redis_client.incr('clickCount)

Redis pipelines can avoid multiple related calls to the server -> Less network overhead

# 2/10

Wednesday, February 12, 2025 9:13 AM

A document database is a non-relational database storing data in structured documents such as JSON
- Supposed to be simple, flexible, and scalable

JSON (Javascript Object Notation)
- Lightweight and can work with multipl different database types
- Easy for humans and machines to read and write
- Can have two different structures:
    - Collection of name/value pairs (e.g. record, dictionary, hash table)
    - Ordered List (e.g. array, list, vector)
    - Supported by pretty much all programming languages

BSON (Binary JSON)
- Binary encoded serialization of JSON-like documents
- Allows you to work with types not allowed in regular JSON (e.g. date, binary data)
- Keeps space overhead to minimum
- Easily traversed, which is very important for document db
- Efficient in encoding and decoding

XML (eXtensible Markup Language)
- Predecessor to JSON as data exchange format
- Used with CSS to separate content / formatting for web pages
- Similar to HTML but with much more tags

Why use Document Databases?
- Address impedance mismatch problem between object persistence in object oriented systems and how relational DBs structure data
    - OO Programming -> Inheritance, composition of types
    - How do we save a complex object to a relational database? We basically have to break it down into parts
- Structure of document is self describing
- Work well with apps that use JSON/XML as a transport layer

MongoDB
- Short for humongous database
- Created after Google workers saw the limitations of using a relational databse to serve 400000+ ads per second
- No predefined schema for documents is needed, e.g. every document in a collection could have different schema

| RDBMS | MongoDB |
| --- | --- |
| Database | Database |
| Table/View | Collection |
| Row | Document |
| Column | Field |
| Index | Index |
| Join | Embedded Document |
| Foreign Key | Reference |

# 2/12

Wednesday, February 12, 2025 9:13 AM

MongoDB + PyMongo (different from Mongo CSH)
These are document based databases

```
From pymongo import MongoClient
Client = MongoClient(
        'mongodb://user_name:pw@localhost:27017'
)

Db = client['ds4300']
Collection = db['mycollection']


# Inserting a single document
Post = {
        'author': 'Mark',
        'text': 'I love Mongo',
        'tags': ['Mongo', 'Python']
}
Collection.insert_one(post).inserted_id
```

# 2/13

Thursday, February 13, 2025 9:20 AM

Aggregates in PyMongo uses pipelines
- A pipeline is a sequence of stages through which documents proceed
- E.g. match, project, sort, limit, unwind, group, lookup

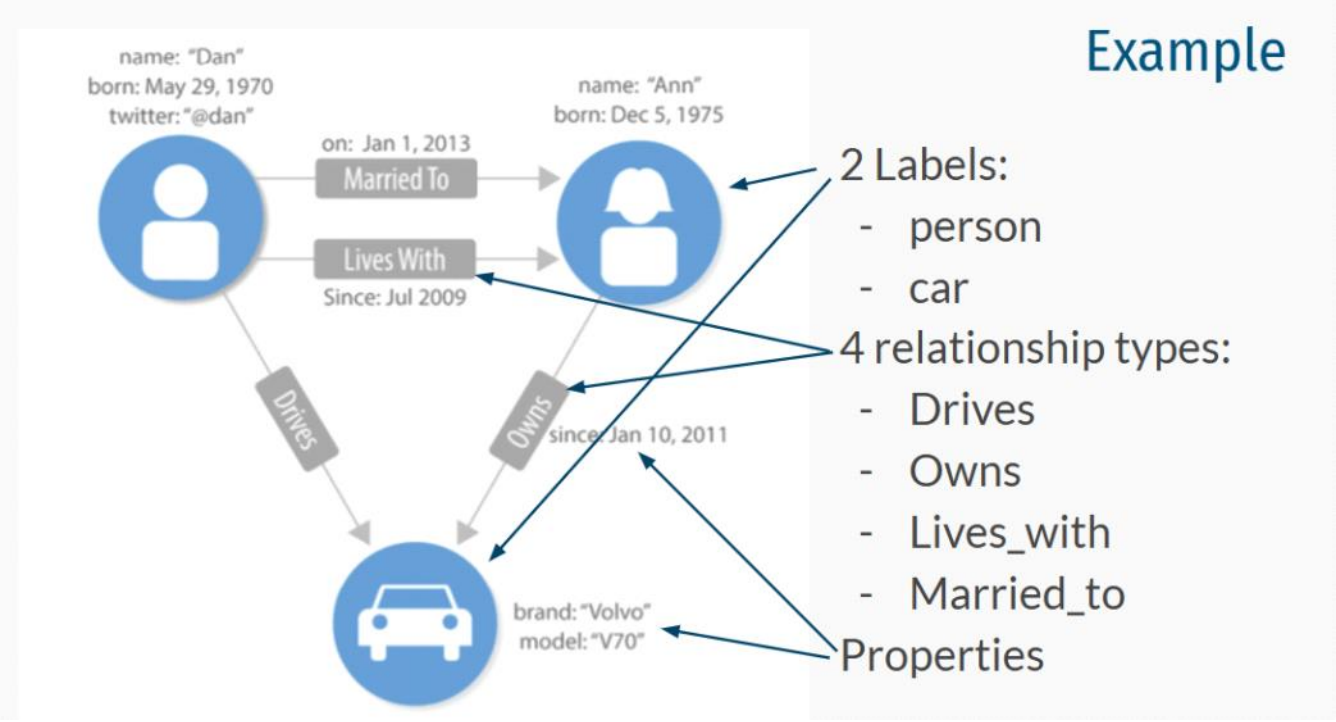
# 2/19

Wednesday, February 19, 2025 9:22 AM

Introduction to the Graph Data Model
- Data model based on the graph data structure
- Composed of nodes and edges (edges connect nodes)
- Each is uniquely identified and can contain properties
- Support queries based on graph-oriented operations, such as traversal, shortest path, etc

Places were Graphs show up:
- Social networks (e.g. Instagram, social interactions for psychology)
- The web (big graph of pages connected by hyperlinks)
- Chemical and biological data (genetics, chemistry)

Labeled Property Graph:
- Composed of a set of nodes (vertexes) and relationships (edges
- Labels are used to mark a node as part of a group
- Properties are attributes (similar to KV) and can exist on nodes and relationships
- Possible to have nodes without any relationships, but not relationships unattached to any node




A path is an ordered sequence of nodes connected by edges in which no nodes or edges are repeated

Connected graph: There is a path between any two nodes in the graph
Weighted graph: Edges have a weight property
Directed graph: Edges have a start and end (i.e. a direction)
Acyclic graph: Graph contains no cycles

Sparse graphs are graphs with very few edges (adjacency list is better)
- Dense graphs are better suited through adjacency matrixes

Clique: Graphs where every node is connected with each other

Rooted Trees: Root node with no cycles
Binary Tree: Up to 2 child nodes and no cycle
Spanning Tree: Subgraph of all nodes but not all relationships and no cycles

Pathfinding:
- Finding the shortest path between two nodes (fewest edges or lowest weight)
- Average shortest path can be used to monitor efficiency and resiliency of networks
- Minimum spanning tree, cycle detection, max/min flow are other types of pathfinding

Types of Graph Algorithms (Centrality + Community Detection)
- Centrality: Determining which node is more improtant in a network compared to other nodes
- Community Detection: Evaluate clustering or partitioning nodes in a graph

Neo4j
- Very popular graph database system for supporting transactional / analytical processing of graph-based data
- Considered schema optional, ACID compliant, supports distributed computing and various types of indexing

# 2/20

Thursday, February 20, 2025 9:19 AM

Neo4j - Query Language and Plugins

Cypher:
- Neo4j's graph query language created in 2011
- Supposed to be SQL-esque (though not completely like SQL)
- Provides a visual way to matching patterns and relations
    (nodes)-[:CONNECT_TO]->(otherNodes)

APOV Plugin:
- Awesome Procedures on Cypher (lmao)
- Add on library that provides hundreds of procedures and functions

Docker Compose:
- Supports multi-container management
- Set-up is declarative using YAML docker-compose.yaml file
- 1 command can be used to start, stop, or scale a number of services at one time
- Provides a consistent method for producing an identical environment (no more "it works on my machine!")

Do not put "secrets" like a password into a docker compose file. Use an .env file instead.
Service is a piece of software that runs on your computer at all times

Important Docker Commands:
Docker compose up
Docker compose up -d
Docker compose down
Docker compose start
Docker compose stop
Docker compose build

# 2/24

Monday, February 24, 2025 9:16 AM

Project overview:

Take class notes -> Generate embeddings in chunks of file name, pdf page, vector
-> Put into redis stack

When asking a question, find the 10 most similar chunks to get context, while also embedding the question as well

This system is called a Retrieval Augmented Generation (RAG)

# 2/27

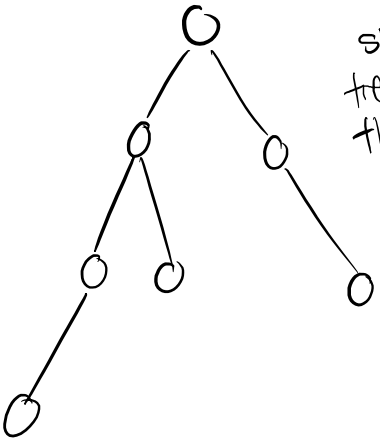
Thursday, February 27, 2025 9:21 AM

# 3/10

Monday, March 10, 2025 9:15 AM

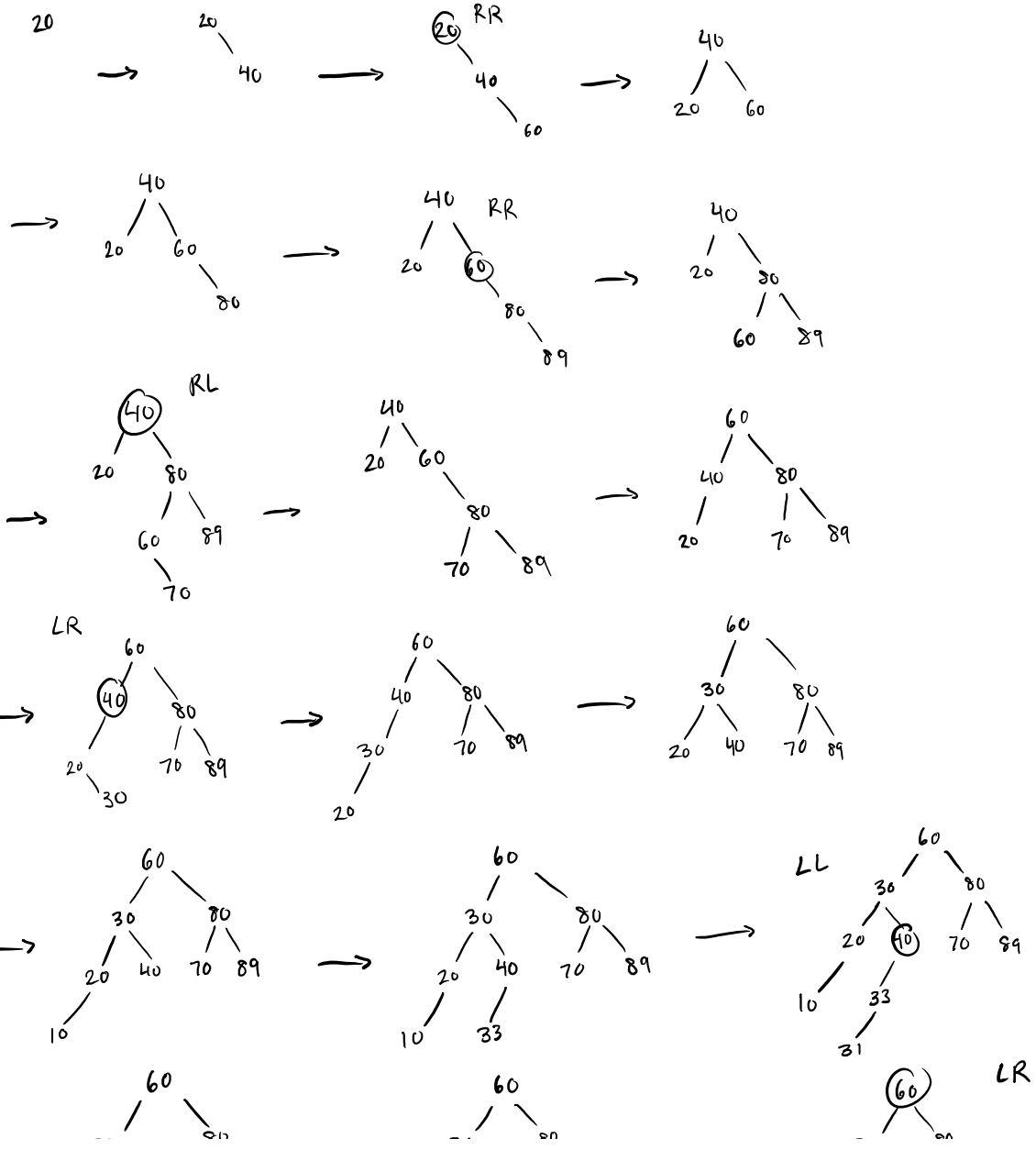
# HW 2

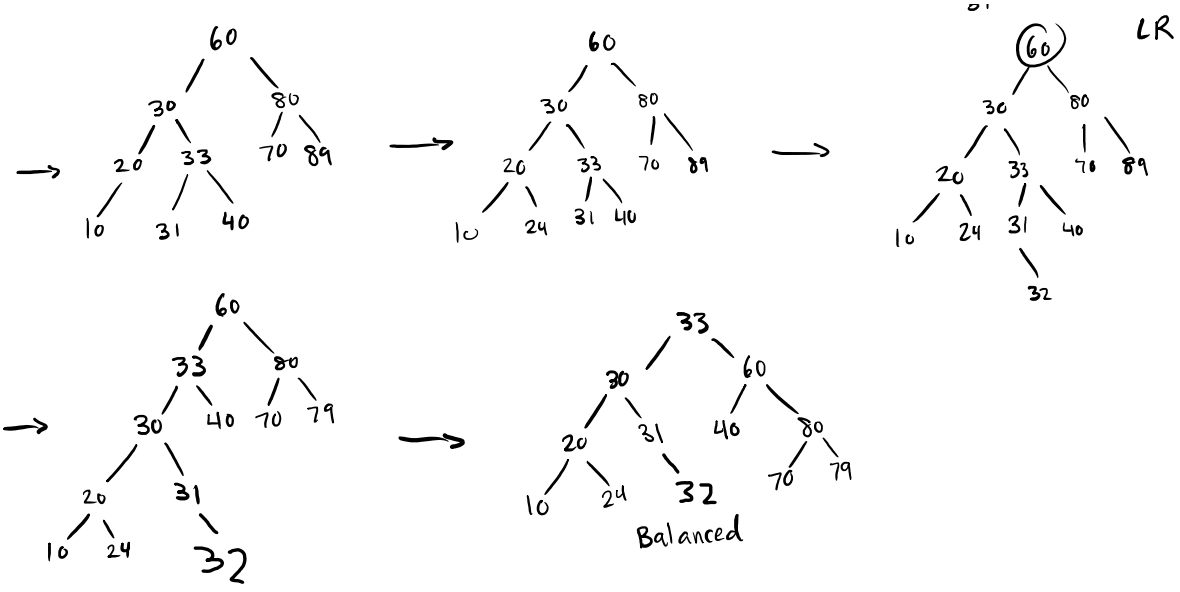
Wednesday, January 22, 2025 9:37 AM



smallest AVL tree with depth 4 that is balanced

Order: 20, 40, 60, 80, 89, 70, 30, 10, 33, 31, 24, 32






Hashmap A:

0: (20, O) → (40, S) → (60, T) → (80, R) → (70, E) → (30, T) → (10, N)
1: (31, H)
2: (32, E)
3: (33, A)
4: (24, R)
5:
6:
7:
8:
9: (89, N)

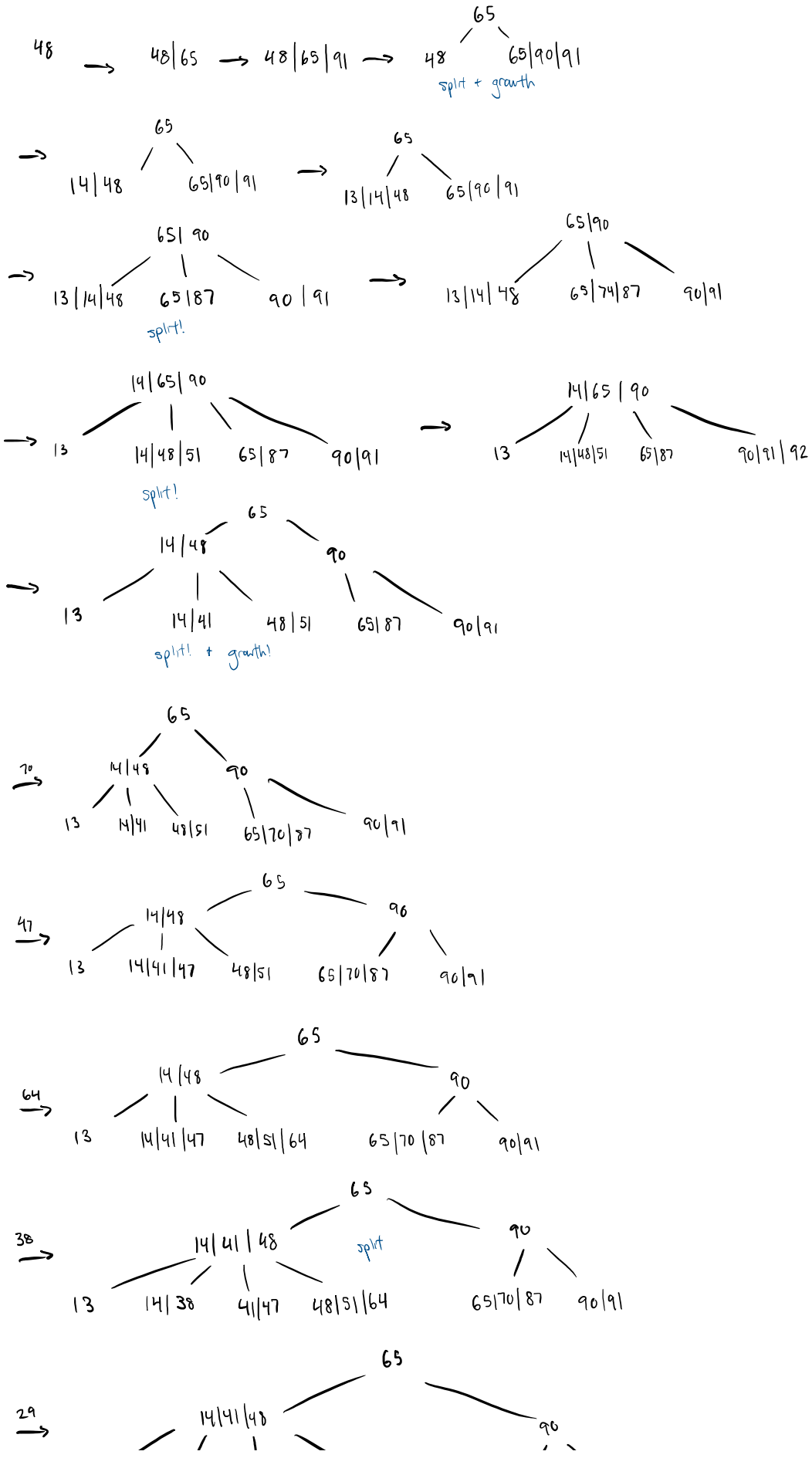
Hashmap B:

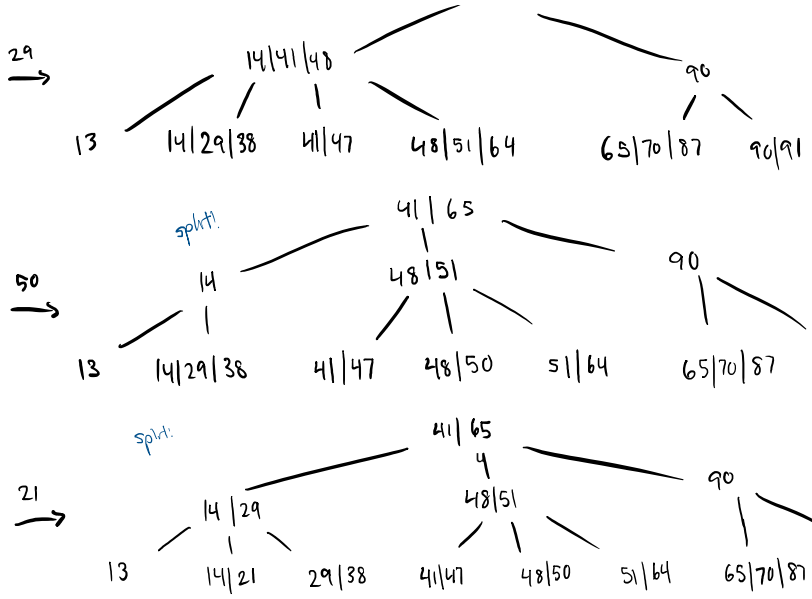
(20, O, 79
(40, S, 83
(60, T, 84
(80, R, 82
(89, N, 78
(70, E, 69
(30, T, 84
(10, N, 78
(33, A, 65
(31, H, 72
(24, R, 82
(32, E, 69

0:
1:
2: (80, R) → (31, H) → (24, R)
3: (40, S)
4: (60, T) → (30, T)
5: (33, A)
6:
7:
8: (89, N) → (10, N)
9: (20, O) → (70, E) → (32, E)

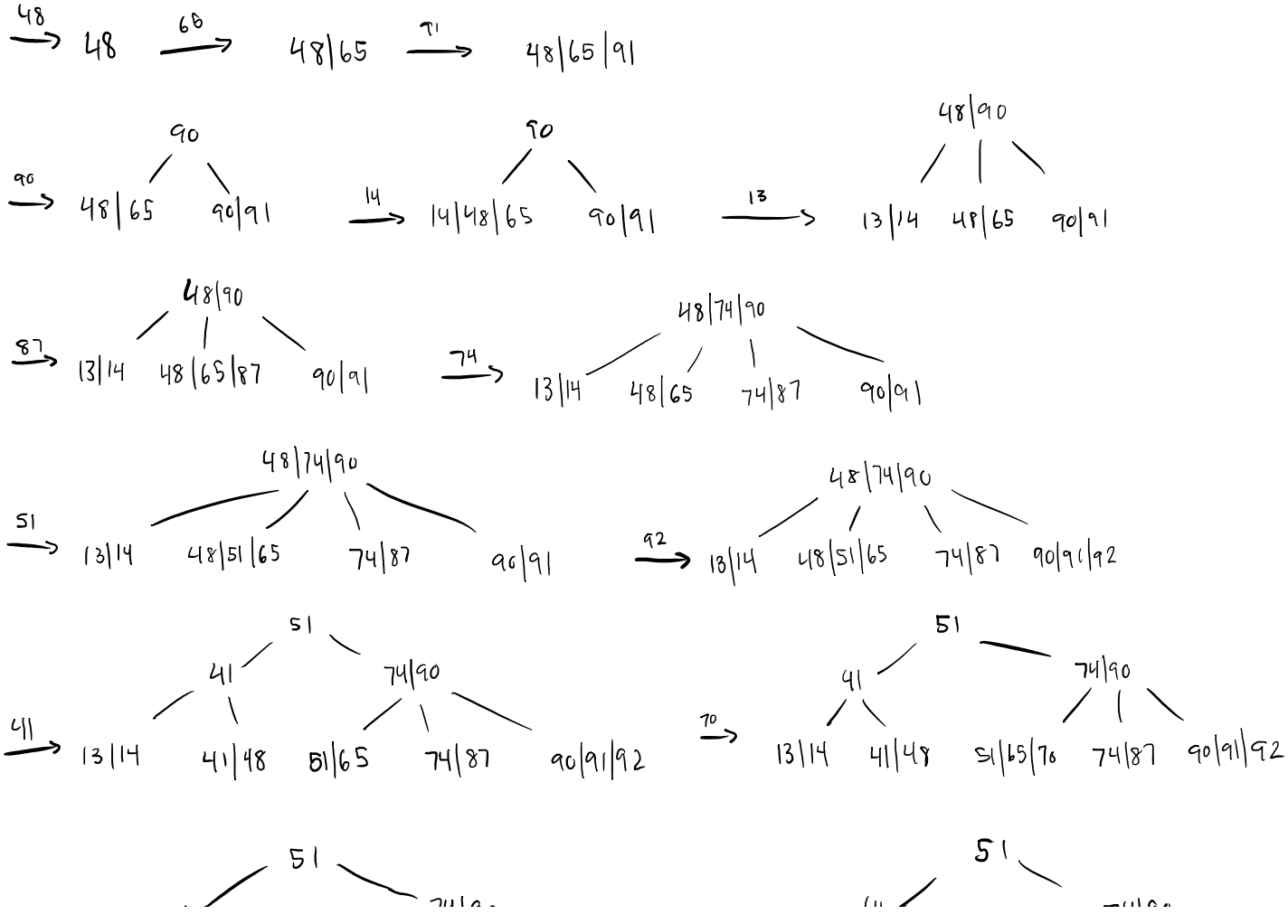
48, 65, 91, 90, 14, 13, 87, 74, 51, 92, 41, 70, 47, 64, 38, 29, 50, 21

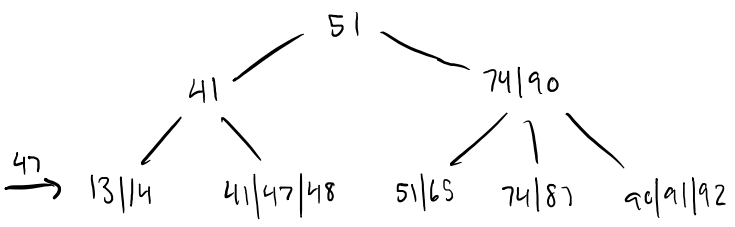
65

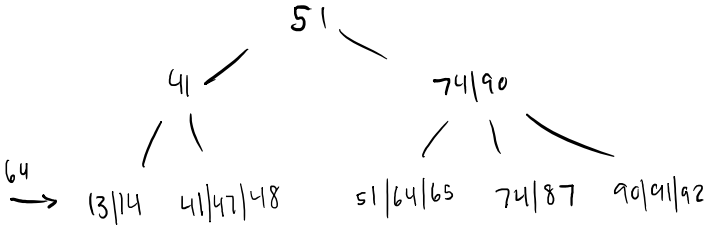

48
48|65
48|65|91
65
48
65|90|91
split + growth
65
14|48
65|90|91
65
13|14|48
65|90|91
65|90
13|14|48
65|87
90|91
split!
65|90
13|14|48
65|74|87
90|91
14|65|90
13
14|48|51
65|87
90|91
split!
14|65|90
13
14|48|51
65|87
90|91|92
65
14|48
90
13
14|41
48|51
65|87
90|91
split! + growth!
65
14|48
90
13
14|41
48|51
65|70|87
90|91
70
65
14|48
90
13
14|41
48|51
65|70|87
90|91
47
65
14|48
90
13
14|41|47
48|51
65|70|87
90|91
65
14|48
90
13
14|41|47
48|51|64
65|70|87
90|91
64
65
14|41|48
90
13
14|41|47
48|51|64
65|70|87
90|91
split
38
65
14|41|48
90
13
14|38
41|47
48|51|64
65|70|87
90|91
29
65
14|41|48
90



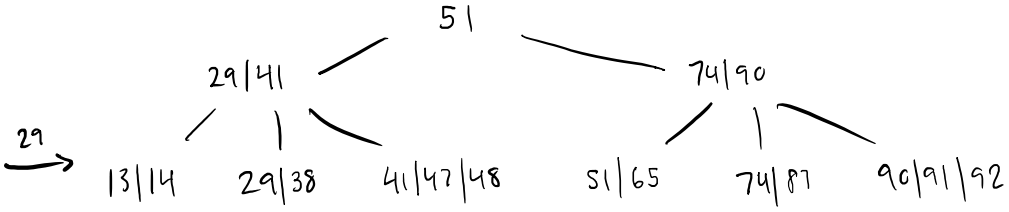

48, 65, 91, 90", 14, 13, 87, 74, 51, 92, 41", 70, 47, 64, 38, 29, 50, 21

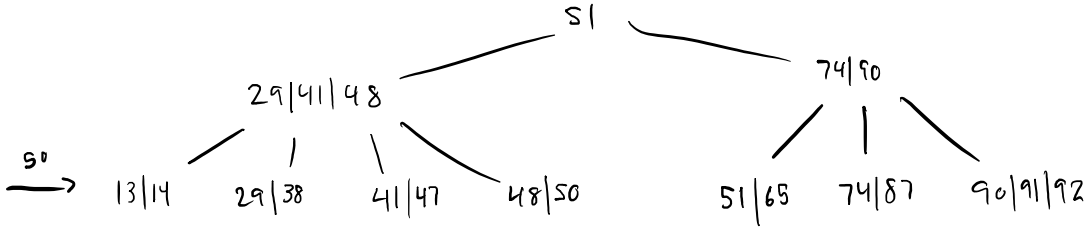


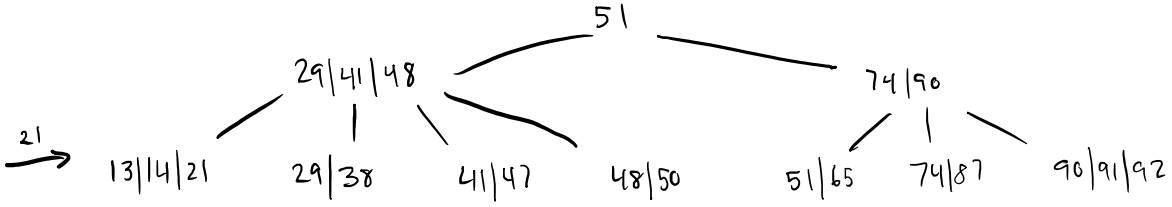

51
41
74|90
47
13|14
41|47|48
51|65
74|87
90|91|92


51
41
74|90
64
13|14
41|47|48
51|64|65
74|87
90|91|92


51
41
74|90
38
13|14|38
41|47|48
51|65
74|87
90|91|92


51
29|41
74|90
29
13|14
29|38
41|47|48
51|65
74|87
90|91|92


51
29|41|48
74|90
50
13|14
29|38
41|47
48|50
51|65
74|87
90|91|92


51
29|41|48
74|90
21
13|14|21
29|38
41|47
48|50
51|65
74|87
90|91|92