Data Structures & Algorithms

Q1: What is the difference between a list where memory is contiguously allocated and a list where linked structures are used?

A contiguously allocated list (e.g., an array) stores elements in adjacent memory locations, allowing for constant-time access via indexing but requiring shifting when inserting or deleting elements. A linked list, on the other hand, consists of nodes with pointers to the next element, making insertions and deletions efficient but requiring O(n) access time for indexing.

Q2: When are linked lists faster than contiguously-allocated lists?

Linked lists are faster when frequent insertions and deletions are required, particularly at the beginning or middle of the list. Contiguously allocated lists (arrays) require shifting elements, making them inefficient for such operations.

Q3: Add 23 to the AVL Tree below. What imbalance case is created with inserting 23?

Initial AVL Tree:

markdown

```
       30
      /  \
     25  35
    /
   20
```

Step 1: Insert 23

markdown

```
       30
      /  \
     25  35
    /
   20
     \
     23
```

Step 2: Identify the imbalance

Node 25 now has a balance factor of +2 (Left-heavy), and node 20 has a right child (23), creating an LR (Left-Right) imbalance.

Step 3: Correcting with rotations

Perform Left Rotation on 20, then Right Rotation on 25.

Database Systems

Q4: Why is a B+ Tree better than an AVL Tree when indexing a large dataset?

A B+ Tree is optimized for disk-based storage, reducing the number of disk accesses due to its higher branching factor. Unlike an AVL Tree, which maintains strict balance but has higher depth, a B+ Tree stores all keys in leaf nodes, allowing for efficient range queries and faster retrieval from disk.

Q5: What is disk-based indexing and why is it important for database systems?
Disk-based indexing organizes data in a way that minimizes disk I/O operations, crucial for handling large datasets that exceed RAM capacity. Structures like B+ Trees and Hash Indexes optimize access patterns, reducing the need for frequent disk reads.
Q6: In the context of a relational database system, what is a transaction?
A transaction is a unit of work in a database that consists of multiple operations executed as a single logical unit. It follows ACID properties to ensure consistency and reliability.
Q7: Succinctly describe the four components of ACID-compliant transactions.

- Atomicity: The transaction is all-or-nothing (either fully completes or fully rolls back).
- Consistency: Ensures the database remains in a valid state before and after the transaction.
- Isolation: Concurrent transactions do not interfere with each other.
- Durability: Once committed, a transaction's changes persist, even after a crash.

Q8: Why does the CAP principle not make sense when applied to a single-node MongoDB instance?
The CAP theorem applies to distributed systems, where network partitions can occur. A single-node MongoDB instance does not have partitions, so it always guarantees Consistency and Availability without needing to trade off between them.

Scalability & Performance
Q9: Describe the differences between horizontal and vertical scaling.

- Horizontal scaling (scale-out): Adding more machines to distribute the load (e.g., adding more database servers).
- Vertical scaling (scale-up): Increasing the resources of a single machine (e.g., adding more CPU/RAM).

Q10: Briefly describe how a key/value store can be used as a feature store.
A key/value store (like Redis) can store precomputed features for machine learning models, where the key is a unique identifier (e.g., user ID) and the value contains feature vectors. This enables fast lookups for real-time inference.

NoSQL Databases & Redis
Q11: When was Redis originally released?
Redis was initially released in 2009.
Q12: In Redis, what is the difference between the INC and INCR commands?

- INC does not exist in Redis.
- INCR increases a stored integer value by 1.

Q13: What are the benefits of BSON over JSON in MongoDB?
BSON (Binary JSON) is optimized for storage and retrieval in MongoDB:

- Supports additional data types (e.g., Date, Decimal128).
- Faster parsing due to binary format.
- Supports indexing and query optimization better than plain JSON.

Q14: Write a MongoDB query based on the movies dataset that returns the titles of all movies released between 2010 and 2015 from the suspense genre.
js

```
db.movies.find(
  {
    releaseYear: { $gte: 2010, $lte: 2015 },
    genre: "Suspense"
  },
  { title: 1, _id: 0 }
)
```

Q15: What does the $nin operator mean in a MongoDB query?
The $nin (Not In) operator filters out documents where a field's value is not in a specified list.
Example:
js

```
db.movies.find({ genre: { $nin: ["Horror", "Thriller"] } })
```

This returns movies not in the Horror or Thriller genres.

Q16: What is the time complexity of searching in an AVL Tree?
O(log n) because AVL trees maintain balance, ensuring a maximum height of O(log n).
Q17: What is the worst-case time complexity of searching in an unbalanced binary search tree?
O(n), as the tree can degenerate into a linked list.
Q20: What is the difference between a Min Heap and a Max Heap?

- Min Heap: The root node contains the smallest value.
- Max Heap: The root node contains the largest value.

Q22: What is the difference between a clustered index and a non-clustered index?

- Clustered Index: Determines the physical order of rows in a table (only one per table).
- Non-Clustered Index: Stores pointers to the actual data (multiple per table).

Q23: What is the difference between a Hash Index and a B-Tree Index?

- Hash Index: Fast lookup for equality checks but poor range queries.
- B-Tree Index: Supports range queries efficiently but has higher overhead.

Q27: How does Redis handle persistence?

- RDB (Redis Database Snapshot): Periodic snapshots of memory.
- AOF (Append-Only File): Logs all write operations for recovery.

AVL Trees

- Self-balancing BST
- O(log n) time complexity for insert, delete, and search
- Used in databases, in-memory caches
- Strict balancing (height difference ≤ 1 at all nodes)

AVL Trees: Balancing and Rotations Explained

An AVL tree is a type of self-balancing binary search tree (BST) where the height of the left and right subtrees of any node differs by at most 1. This ensures that lookup, insertion, and deletion operations always maintain a worst-case time complexity of O(log n).
Each node in an AVL tree maintains a balance factor, which is calculated as:
Balance Factor=Height of Left Subtree−Height of Right Subtree\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}Balance Factor=Height of Left Subtree−Height of Right Subtree
The balance factor must always be -1, 0, or 1 for the tree to be considered balanced. If inserting or deleting a node causes the balance factor to become less than -1 or greater than 1, the tree becomes unbalanced, and rotations are required to restore balance.

Types of AVL Tree Imbalances and How to Fix Them
There are four cases where an AVL tree can become unbalanced after an insertion or deletion. These cases are based on whether the imbalance occurs on the left or right subtree and whether it is caused by inserting a node in a left or right child.

1. Left-Left (LL) Imbalance - Fix with a Right Rotation (Single Rotation)
When does LL Imbalance occur?
- This happens when a new node is inserted into the left subtree of the left child of a node.
- The balance factor becomes greater than 1 at some ancestor node.

Example of LL Imbalance:
- Suppose we insert 10, then 5, then 2.
- The node 10 becomes unbalanced because its left subtree (rooted at 5) is two levels taller than its right subtree.

How to fix LL imbalance?
- Perform a right rotation on the unbalanced node.
- The left child of the unbalanced node becomes the new root.
- The original root moves down and becomes the right child of the new root.

Steps for Right Rotation (fixing LL imbalance at node X):
1. Let Y be the left child of X (the unbalanced node).
2. Move Y up, making it the new root.
3. Make X the right child of Y.
4. If Y had a right child, move it to the left child of X.

Result:
- The tree is now balanced because the left and right subtree heights are restored.

2. Right-Right (RR) Imbalance - Fix with a Left Rotation (Single Rotation)
When does RR Imbalance occur?
- This happens when a new node is inserted into the right subtree of the right child of a node.
- The balance factor becomes less than -1 at some ancestor node.

Example of RR Imbalance:
- Suppose we insert 10, then 15, then 20.

- The node 10 becomes unbalanced because its right subtree (rooted at 15) is two levels taller than its left subtree.

How to fix RR imbalance?

- Perform a left rotation on the unbalanced node.
- The right child of the unbalanced node becomes the new root.
- The original root moves down and becomes the left child of the new root.

Steps for Left Rotation (fixing RR imbalance at node X):

1. Let Y be the right child of X (the unbalanced node).
2. Move Y up, making it the new root.
3. Make X the left child of Y.
4. If Y had a left child, move it to the right child of X.

Result:

- The tree is now balanced.

3. Left-Right (LR) Imbalance - Fix with a Left Rotation, then a Right Rotation (Double Rotation)

When does LR Imbalance occur?

- This happens when a new node is inserted into the right subtree of the left child of a node.
- The balance factor at some ancestor node becomes greater than 1, but the imbalance is in the right direction.

Example of LR Imbalance:

- Suppose we insert 10, then 5, then 8.
- The node 10 becomes unbalanced, but the imbalance is not a simple LL case because the new node (8) is inserted into the right subtree of 5.

How to fix LR imbalance?

1. First, perform a left rotation on the left child of the unbalanced node.
2. Then, perform a right rotation on the original unbalanced node.

Steps for Fixing LR imbalance at node X:

1. Let Y be the left child of X.
2. Perform a left rotation on Y, making its right child (Z) the new root of the subtree.
3. Perform a right rotation on X, making Z the new root.

Result:

- The tree is now balanced.

4. Right-Left (RL) Imbalance - Fix with a Right Rotation, then a Left Rotation (Double Rotation)

When does RL Imbalance occur?

- This happens when a new node is inserted into the left subtree of the right child of a node.
- The balance factor at some ancestor node becomes less than -1, but the imbalance is in the left direction.

Example of RL Imbalance:

- Suppose we insert 10, then 15, then 12.
- The node 10 becomes unbalanced, but the imbalance is not a simple RR case because the new node (12) is inserted into the left subtree of 15.

How to fix RL imbalance?

1. First, perform a right rotation on the right child of the unbalanced node.
2. Then, perform a left rotation on the original unbalanced node.

Steps for Fixing RL imbalance at node X:

1. Let Y be the right child of X.
2. Perform a right rotation on Y, making its left child (Z) the new root of the subtree.
3. Perform a left rotation on X, making Z the new root.

Result:

- The tree is now balanced.

Conclusion

AVL trees maintain balance through rotations, ensuring efficient O(log n) operations. Whenever an insertion or deletion causes a node to become unbalanced, we:

1. Check the balance factor to determine the type of imbalance.
2. Apply the correct rotation to restore balance.

B+ Trees

- Balanced tree with all keys in leaf nodes
- Efficient range queries
- Disk-friendly due to high branching factor
- Used in database indexing (MySQL, PostgreSQL, MongoDB, etc.)

Sharding

- Horizontal partitioning of data across multiple servers
- Improves scalability
- Used in distributed databases (e.g., MongoDB, Cassandra)

Redis

- In-memory key-value store
- Supports data structures like lists, sets, and hashes
- Persistence through RDB and AOF
- Used for caching, session storage, real-time analytics

MongoDB

- NoSQL document database
- Uses BSON (Binary JSON) for storage
- Supports flexible schema and sharding
- Used in web applications, big data processing

Basic MongoDB Query Usages

MongoDB is a NoSQL document-based database that uses collections of documents instead of tables with rows (like relational databases). Queries are performed on these documents, which are typically stored in BSON format (Binary JSON). Below are the basic MongoDB queries along with examples and when to use them.

1. Insert Documents

You can insert a single document or multiple documents into a MongoDB collection.

- Insert One:

- db.collection.insertOne({
-   name: "Alice",
-   age: 25,
-   city: "New York"
- });

- Insert Many:

- db.collection.insertMany([
-   { name: "Bob", age: 30, city: "Los Angeles" },
-   { name: "Charlie", age: 35, city: "Chicago" }
-]);

2. Find Documents
Use find() to retrieve documents from a collection. You can specify conditions for querying.

- Find All Documents:

- db.collection.find({});

- Find with a Query Condition:

- db.collection.find({ age: { $gt: 30 } });

This finds all documents where the age is greater than 30.

- Find One Document:

- db.collection.findOne({ name: "Alice" });

3. Update Documents
You can update documents in MongoDB using updateOne() or updateMany(). You specify the condition and the update operation.

- Update One Document:

- db.collection.updateOne(
-   { name: "Alice" }, // Condition
-   { $set: { age: 26 } } // Update operation
-);

- Update Many Documents:

- db.collection.updateMany(
-   { city: "Chicago" }, // Condition
-   { $set: { city: "San Francisco" } } // Update operation
-);

4. Delete Documents
You can delete documents using deleteOne() or deleteMany().
- Delete One Document:

- db.collection.deleteOne({ name: "Bob" });

- Delete Many Documents:

- db.collection.deleteMany({ age: { $lt: 30 } });

5. Sorting Documents
You can sort documents based on a field using sort().
- Sort in Ascending Order:

- db.collection.find().sort({ age: 1 });

- Sort in Descending Order:

- db.collection.find().sort({ age: -1 });

6. Projection (Select Specific Fields)
You can specify which fields to include or exclude when querying documents.
- Include Fields:

- db.collection.find({}, { name: 1, age: 1 });

This retrieves only the name and age fields for all documents.
- Exclude Fields:

- db.collection.find({}, { _id: 0, age: 0 });

This retrieves all fields except _id and age.
7. Aggregate Queries
MongoDB supports aggregation for complex queries, such as grouping, filtering, and sorting.

- Example of Aggregation Pipeline:

- db.collection.aggregate([
-   { $match: { city: "New York" } },
-   { $group: { _id: "$age", count: { $sum: 1 } } },
-   { $sort: { _id: 1 } }
-]);

This query:

1. Filters documents where the city is "New York".
2. Groups by age and counts how many people of each age are in the city.
3. Sorts the result by age in ascending order.

8. Using Operators
MongoDB provides various operators for filtering and querying data.

- $gt (Greater Than):

- db.collection.find({ age: { $gt: 25 } });

Finds all documents where the age is greater than 25.

- $lt (Less Than):

- db.collection.find({ age: { $lt: 40 } });

Finds all documents where the age is less than 40.

- $in (In an Array):

- db.collection.find({ city: { $in: ["New York", "Los Angeles"] } });

Finds all documents where the city is either "New York" or "Los Angeles".

- $nin (Not In an Array):

- db.collection.find({ city: { $nin: ["Chicago", "Houston"] } });

Finds all documents where the city is not "Chicago" or "Houston".

```
# Set up your connection to Mongo DB here.
import pymongo
from bson.json_util import dumps

uri = "mongodb://sam:bacchus@localhost:27017"
client = pymongo.MongoClient(uri)
mflixdb = client.mflix
```

Directions:

- Use the mflix sample database to prepare a pymongo query each of the following prompts.
- Be sure to print the results of your query using the dumps function.

Question 1:
Give the street, city, and zipcode of all theaters in Massachusetts.

```
ma_theaters = mflixdb.theaters.find(
   {"location.address.state": "MA"},
   {"_id": 0, "location.address.street1": 1, "location.address.city": 1, "location.address.zipcode":
1}
)

print(dumps(ma_theaters, indent=2))
```

Question 2:
How many theaters are there in each state? Order the output in alphabetical order by 2-character state code.

```
theaters_per_state = mflixdb.theaters.aggregate([
   {"$group": {"_id": "$location.address.state", "Theaters": {"$sum": 1}}},
   {"$project": {"State": "$_id", "Theaters": 1, "_id": 0}},
   {"$sort": {"State": 1}}])

print(dumps(theaters_per_state, indent=2))
```

Question 3
How many movies are in the Comedy genre?
In [ ]:

```
# !!!BEFORE SUBMIT - check if this is correct, might need to do {"$in": ["Comedy"]}!!!
num_comedy_movies = mflixdb.movies.count_documents({"genres": "Comedy"})

print(num_comedy_movies)
```

Question 4:
What movie has the longest run time? Give the movie's title and genre(s).

In [84]:
```
longest_movie = mflixdb.movies.find( { }, {"_id": 0, "title": 1, "genres": 1} ).sort("runtime",
-1).limit(1)

print(dumps(longest_movie, indent=2))
```
Question 5:
Which movies released after 2010 have a Rotten Tomatoes viewer rating of 3 or higher? Give the title of the movies along with their Rotten Tomatoes viewer rating score. The viewer rating score should become a top-level attribute of the returned documents. Return the matching movies in descending order by viewer rating.
In [64]:
```
good_recent_movies = mflixdb.movies.aggregate([
    {"$match": {"year": {"$gt": 2010}, "tomatoes.viewer.rating": {"$gte": 3}}},
    {"$project": {"title": 1, "rating": "$tomatoes.viewer.rating", "_id": 0}},
    {"$sort": {  "rating": -1 }}])

print(dumps(good_recent_movies, indent=2))
```
Question 6:
How many movies released each year have a plot that contains some type of police activity (i.e., plot contains the word "police")? The returned data should be in ascending order by year.
In [87]:
```
num_police_movies_yearly = mflixdb.movies.aggregate([
    {"$match": {"plot": {"$regex": "police", "$options": "i"}}},
    {"$group": {"_id": "$year", "movie_count": {"$sum": 1}}},
    {"$project": {"year": "$_id", "movie_count": 1, "_id": 0}},
    {"$sort": {"year": 1 }}])

print(dumps(num_police_movies_yearly, indent=2))
```

Question 7:
What is the average number of imdb votes per year for movies released between 1970 and 2000 (inclusive)? Make sure the results are order by year.
In [ ]:
```
avg_votes = mflixdb.movies.aggregate([
    {"$match": {"year": {"$gte": 1970, "$lte": 2000}}},
    {"$group": {"_id": "$year", "avg_votes": {"$avg": "$imdb.votes"}}},
    {"$project": {"year": "$_id", "avg_votes": 1, "_id": 0}},
    {"$sort": {"year": 1}}])

print(dumps(avg_votes, indent=2))
```

Question 8:
What distinct movie languages are represented in the database? You only need to provide the list of languages.

```
In [ ]:
languages = mflixdb.movies.distinct("languages")

print(languages)
```

-

CAP Theorem

- Consistency (Every read gets the latest write or an error)
- Availability (Every request receives a response)
- Partition Tolerance (System continues working despite network failures)
- No distributed system can achieve all three perfectly

Each of the following key-value pairs in the list below represents a document (the integer) and a word in that document (the letter) from your P01 data set. Insert the key:value pairs sequentially from (20:O) to (32:E) into an initially empty AVL tree. Use the integer as the key to insert. Note: this will not produce an inverted index like you're creating for the practical.
[(20:O), (40:S), (60:T), (80:R), (89:N), (70:E), (30:T), (10:N), (33:A), (31:H), (24:R), (32:E)]

Provide a list of all insertions that caused an imbalance, at what node the imbalance was found, and what imbalance case was found.

When inserting 60:T, the tree became imbalanced at 20:0 and was found to be a case RR.

When inserting 89:N, the tree became imbalanced at 60:T and was found to be a case RR.

When inserting 70:E, the tree became imbalanced at 40:S and was found to be a case RL.

When inserting 30:T, the tree became imbalanced at 40:S and was found to be a case LR.

When inserting 31:H, the tree became imbalanced at 40:S and was found to be a case LL.

When inserting 32:E, the tree became imbalanced at 60:T and was found to be a case LR.