

DS 4300

Large Scale Information Storage and Retrieval

Foundations

Mark Fontenot, PhD
Northeastern University

Searching

- Searching is the most common operation performed by a database system
- In SQL, the SELECT statement is arguably the most versatile / complex.
- Baseline for efficiency is **Linear Search**
 - Start at the beginning of a list and proceed element by element until:
 - You find what you're looking for
 - You get to the last element and haven't found it

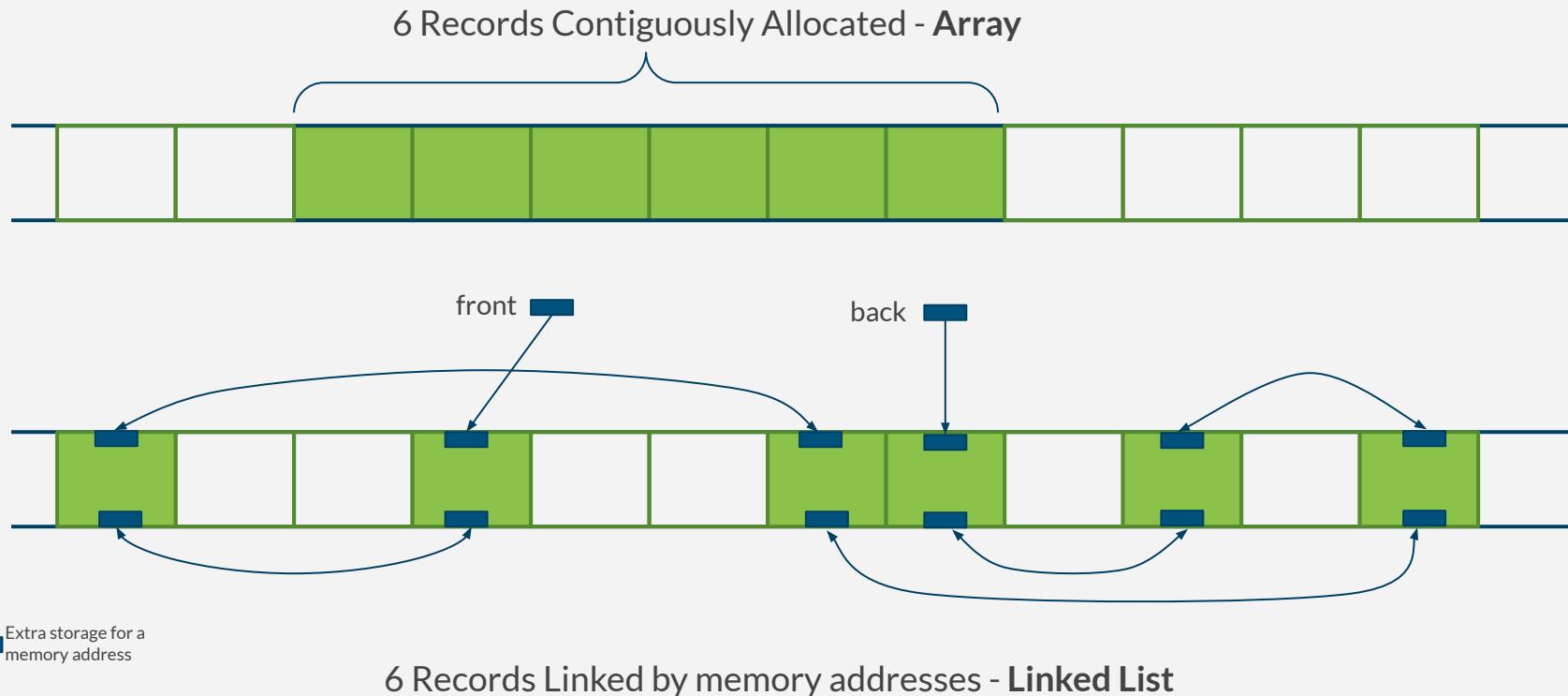
Searching

- **Record** - A collection of values for attributes of a single entity instance; a row of a table
- **Collection** - a set of records of the same entity type; a table
 - Trivially, stored in some sequential order like a list
- **Search Key** - A value for an attribute from the entity type
 - Could be ≥ 1 attribute

Lists of Records

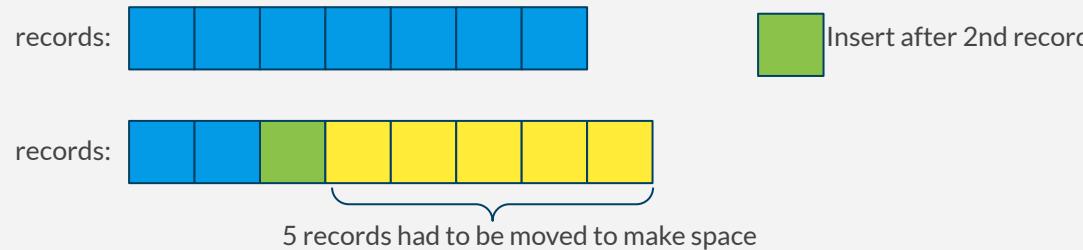
- If each record takes up x bytes of memory, then for n records, we need $n*x$ bytes of memory.
- Contiguously Allocated List
 - All $n*x$ bytes are allocated as a single “chunk” of memory
- Linked List
 - Each record needs x bytes + additional space for 1 or 2 memory addresses
 - Individual records are linked together in a type of chain using memory addresses

Contiguous vs Linked

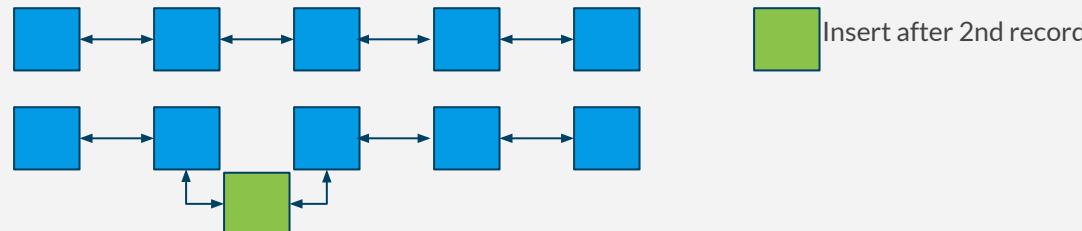


Pros and Cons

- Arrays are faster for random access, but slow for inserting anywhere but the end



- Linked Lists are faster for inserting anywhere in the list, but slower for random access



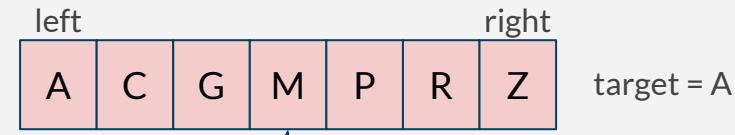
Observations:

- Arrays
 - fast for random access
 - slow for random insertions
- Linked Lists
 - slow for random access
 - fast for random insertions

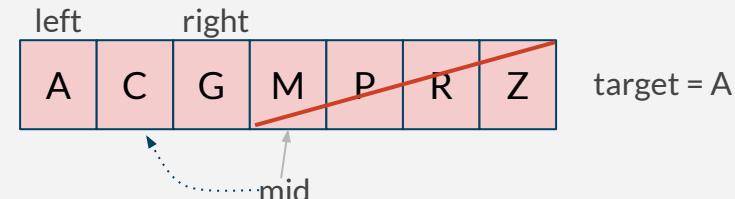
Binary Search

- 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

```
def binary_search(arr, target)
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```



Since $target < arr[mid]$, we reset $right$ to $mid - 1$.



Time Complexity

- Linear Search
 - Best case: target is found at the first element; only 1 comparison
 - Worst case: target is not in the array; n comparisons
 - Therefore, in the worst case, linear search is $O(n)$ time complexity.
- Binary Search
 - Best case: target is found at mid ; 1 comparison (inside the loop)
 - Worst case: target is not in the array; $\log_2 n$ comparisons
 - Therefore, in the worst case, binary search is $O(\log_2 n)$ time complexity.

Back to Database Searching

- Assume data is stored on disk by column id's value
- Searching for a specific id = fast.
- But what if we want to search for a specific *specialVal*?
 - Only option is linear scan of that column
- Can't store data on disk sorted by both id and specialVal (at the same time)
 - data would have to be duplicated → space inefficient

id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

Back to Database Searching

- Assume data is stored on disk by column id's value
- Search time is proportional to number of rows
- Build an external data structure to support faster searching by *specialVal* than a linear scan.
- Can't store all data in memory and search from memory
 - data would have to be duplicated → space inefficient

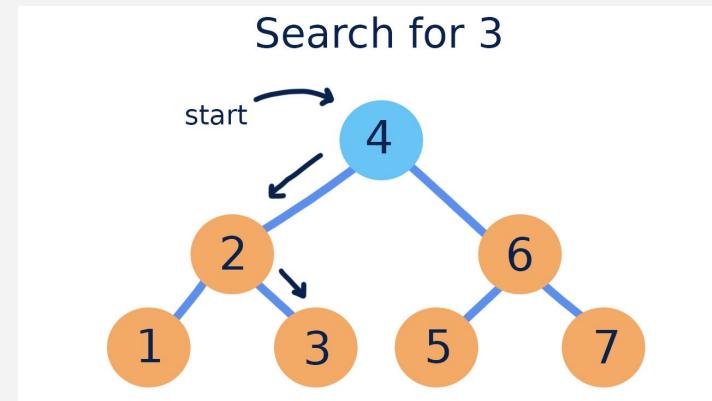
id	specialVal
1	55
2	87
3	50
4	108
5	122
6	149
7	145
8	120
9	50
10	83
11	128
12	117
13	119
14	119
15	51
16	85
17	51
18	145
19	73
20	73

What do we have in our arsenal?

- 1) An array of tuples (specialVal, rowNumber) sorted by specialVal
 - a) We could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table
 - b) But, every insert into the table would be like inserting into a sorted array - slow...
- 2) A linked list of tuples (specialVal, rowNumber) sorted by specialVal
 - a) searching for a specialVal would be slow - linear scan required
 - b) But inserting into the table would theoretically be quick to also add to the list.

Something with Fast Insert and Fast Search?

- Binary Search Tree - a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent.



To the Board!

DS 4300

Moving Beyond the Relational Model

Mark Fontenot, PhD
Northeastern University

Benefits of the Relational Model

- (Mostly) Standard Data Model and Query Language
- ACID Compliance (more on this in a second)
 - Atomicity, Consistency, Isolation, Durability
- Works well with highly structured data
- Can handle large amounts of data
- Well understood, lots of tooling, lots of experience

Relational Database Performance

Many ways that a RDBMS increases efficiency:

- indexing (the topic we focused on)
- directly controlling storage
- column oriented storage vs row oriented storage
- query optimization
- caching/prefetching
- materialized views
- precompiled stored procedures
- data replication and partitioning

Transaction Processing

- **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)
- Help ensure
 - Data Integrity
 - Error Recovery
 - Concurrency Control
 - Reliable Data Storage
 - Simplified Error Handling

ACID Properties

- **Atomicity**
 - transaction is treated as an atomic unit - it is fully executed or no parts of it are executed
- **Consistency**
 - a transaction takes a database from one consistent state to another consistent state
 - consistent state - all data meets integrity constraints

ACID Properties

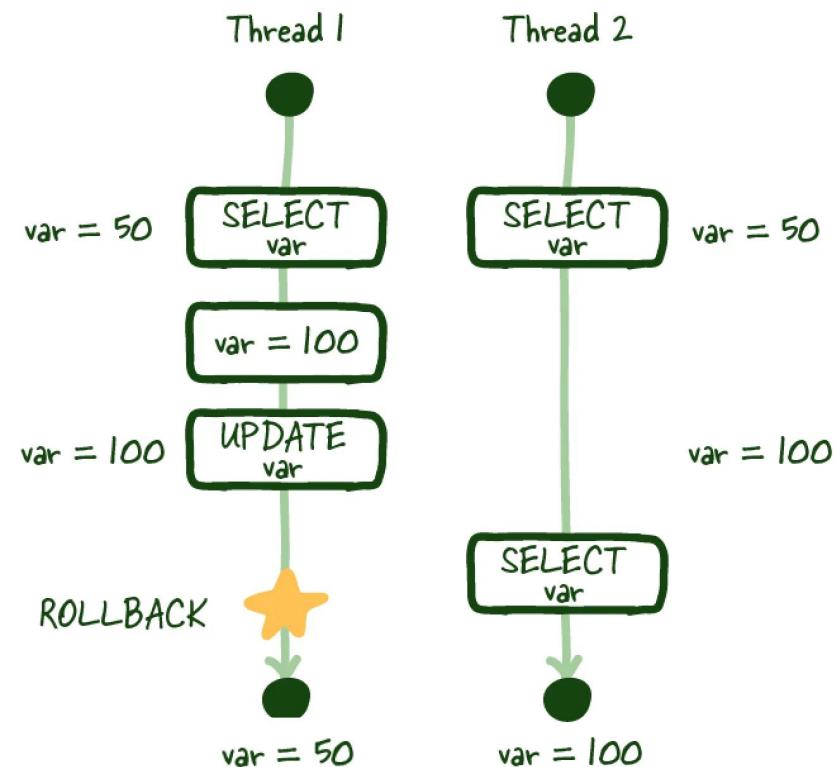
- Isolation

- Two transactions T_1 and T_2 are being executed at the same time but cannot affect each other
- If both T_1 and T_2 are reading the data - no problem
- If T_1 is reading the same data that T_2 may be writing, can result in:
 - Dirty Read
 - Non-repeatable Read
 - Phantom Reads

Isolation: Dirty Read

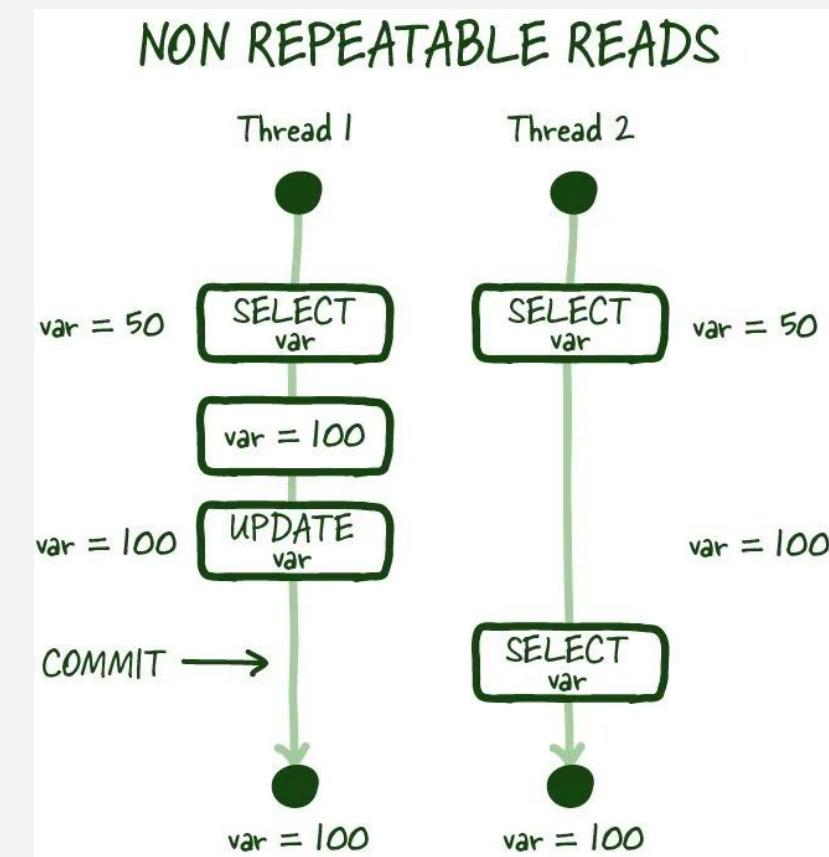
DIRTY READS

Dirty Read - a transaction T_1 is able to read a row that has been modified by another transaction T_2 that hasn't yet executed a COMMIT



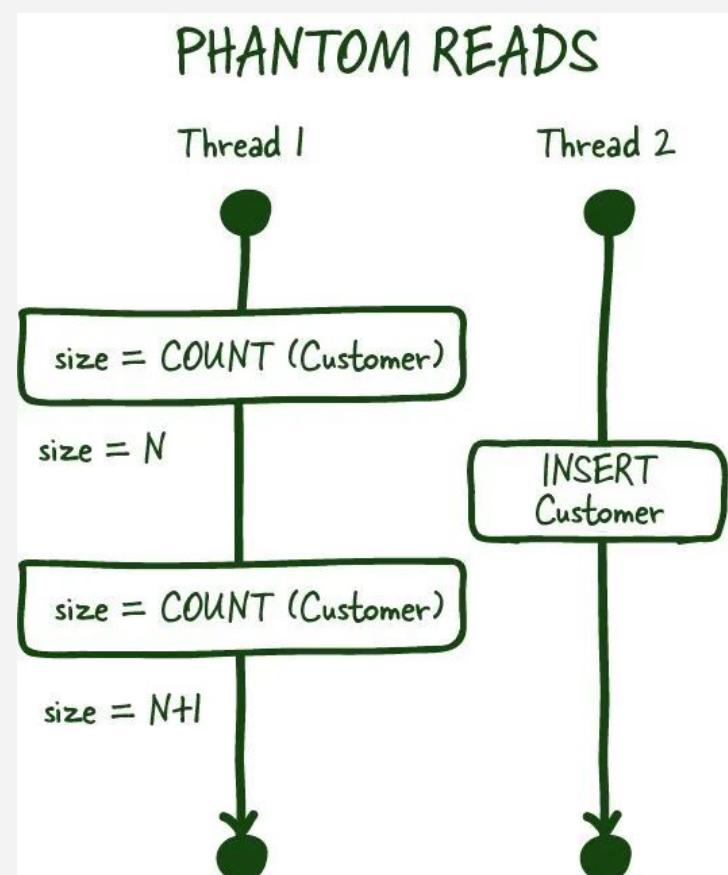
Isolation: Non-Repeatable Read

Non-repeatable Read - two queries in a single transaction T_1 execute a SELECT but get different values because another transaction T_2 has changed data and COMMITTED



Isolation: Phantom Reads

Phantom Reads - when a transaction T_1 is running and another transaction T_2 adds or deletes rows from the set T_1 is using



Example Transaction - Transfer \$\$

```
DELIMITER //

CREATE PROCEDURE transfer(
    IN sender_id INT,
    IN receiver_id INT,
    IN amount DECIMAL(10,2)
)
BEGIN
    DECLARE rollback_message VARCHAR(255)
        DEFAULT 'Transaction rolled back: Insufficient funds';
    DECLARE commit_message VARCHAR(255)
        DEFAULT 'Transaction committed successfully';

    -- Start the transaction
    START TRANSACTION;

    -- Attempt to debit money from account 1
    UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;

    -- Attempt to credit money to account 2
    UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;

    -- Continued Next Slide
```

Example Transaction - Transfer \$\$

```
-- Continued from previous slide

-- Check if there are sufficient funds in account 1
-- Simulate a condition where there are insufficient funds
IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
    -- Roll back the transaction if there are insufficient funds
    ROLLBACK;
    SIGNAL SQLSTATE '45000'      -- 45000 is unhandled, user-defined error
        SET MESSAGE_TEXT = rollback_message;
ELSE
    -- Log the transactions if there are sufficient funds
    INSERT INTO transactions (account_id, amount, transaction_type)
        VALUES (sender_id, -amount, 'WITHDRAWAL');
    INSERT INTO transactions (account_id, amount, transaction_type)
        VALUES (receiver_id, amount, 'DEPOSIT');

    -- Commit the transaction
    COMMIT;
    SELECT commit_message AS 'Result';
END IF;
END //

DELIMITER ;
```

ACID Properties

- **Durability**
 - Once a transaction is completed and committed successfully, its changes are permanent.
 - Even in the event of a system failure, committed transactions are preserved
- For more info on Transactions, see:
 - Kleppmann Book Chapter 7

But ...

Relational Databases may not be the solution to all problems...

- sometimes, schemas evolve over time
- not all apps may need the full strength of ACID compliance
- joins can be expensive
- a lot of data is semi-structured or unstructured (JSON, XML, etc)
- Horizontal scaling presents challenges
- some apps need something more performant (real time, low latency systems)

Scalability - Up or Out?

Conventional Wisdom: Scale vertically (up, with bigger, more powerful systems) until the demands of high-availability make it necessary to scale out with some type of distributed computing model

But why? Scaling up is easier - no need to really modify your architecture. But there are practical and financial limits

However: There are modern systems that make horizontal scaling less problematic.



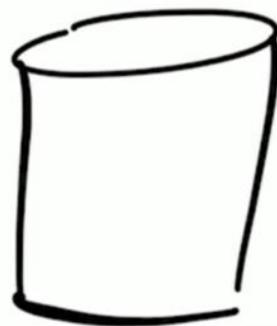
So what? Distributed Data when Scaling Out

A distributed system is “*a collection of independent computers that appear to its users as one computer.*” -Andrew Tennenbaum

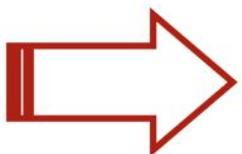
Characteristics of Distributed Systems:

- computers operate concurrently
- computers fail independently
- no shared global clock

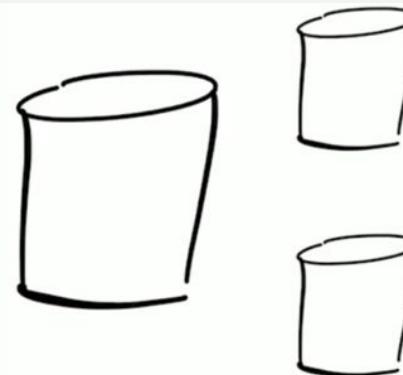
Distributed Storage - 2 Directions



Single
Main
Node

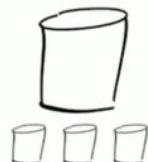


Replication:

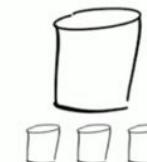


Sharding:

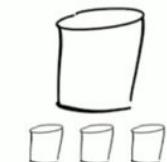
Aaron-
Frances



Frances-
Nancy



Nancy-
Zed

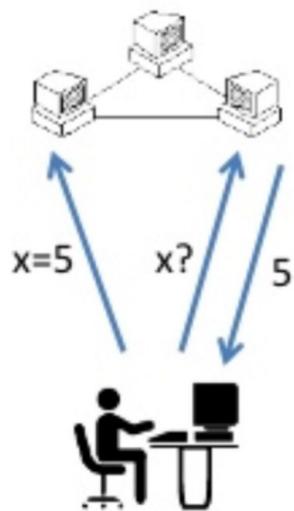


Distributed Data Stores

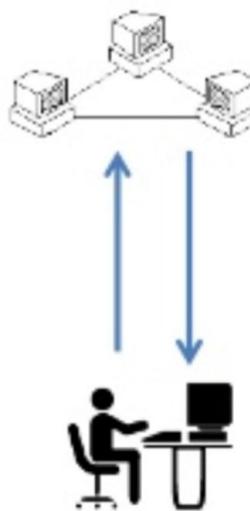
- Data is stored on > 1 node, typically replicated
 - i.e. each block of data is available on N nodes
- Distributed databases can be relational or non-relational
 - MySQL and PostgreSQL support replication and sharding
 - CockroachDB - new player on the scene
 - Many NoSQL systems support one or both models
- But remember: **Network partitioning is inevitable!**
 - network failures, system failures
 - Overall system needs to be **Partition Tolerant**
 - System can keep running even w/ network partition

The CAP Theorem

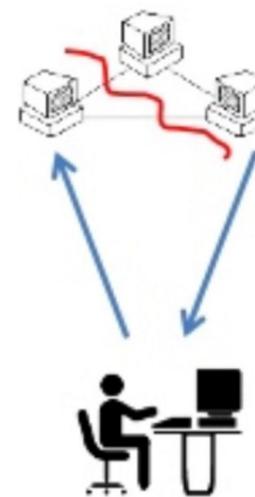
Consistency



Availability



Partition tolerance



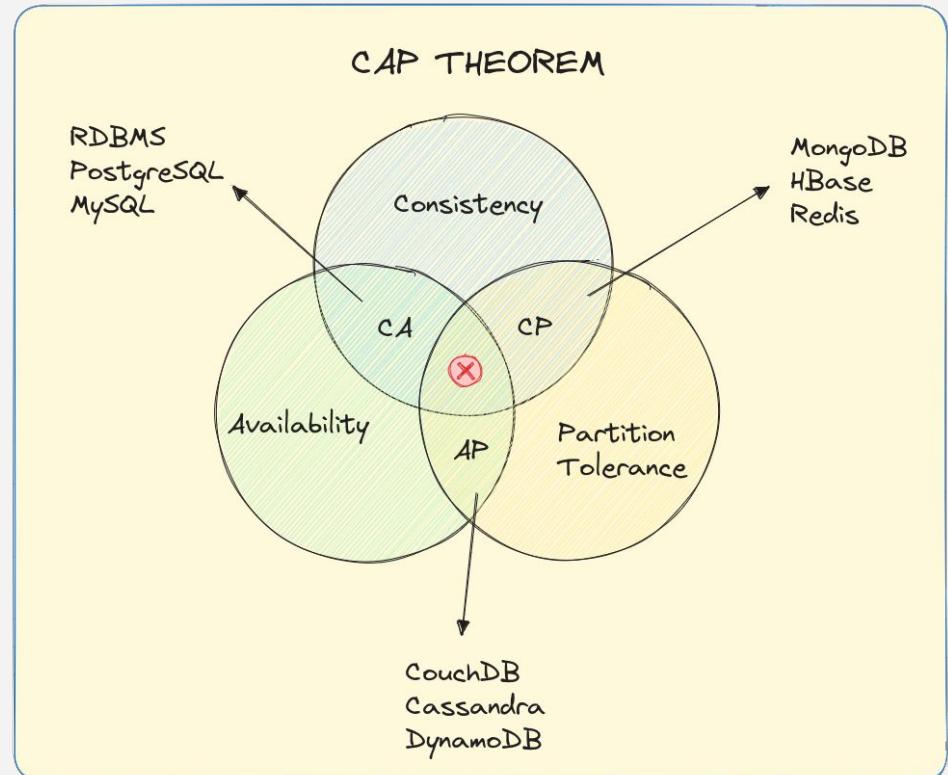
The CAP Theorem

The **CAP Theorem** states that it is impossible for a distributed data store to *simultaneously* provide more than two out of the following three guarantees:

- **Consistency** - Every read receives the most recent write or error thrown
- **Availability** - Every request receives a (non-error) response - but no guarantee that the response contains the most recent write
- **Partition Tolerance** - The system can continue to operate despite arbitrary network issues.

CAP Theorem - Database View

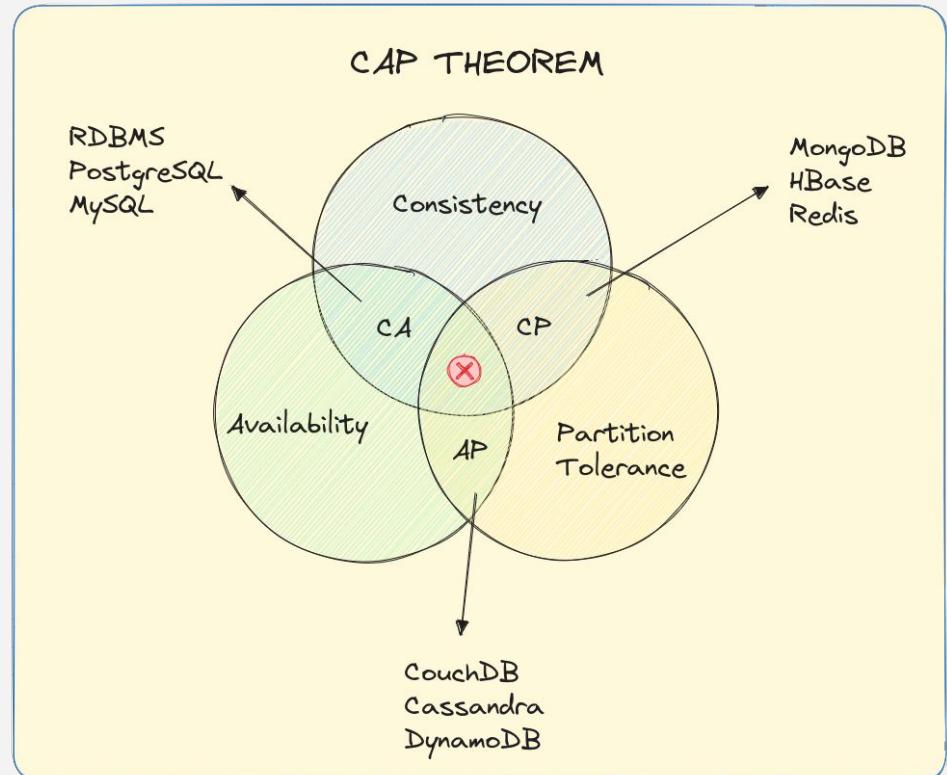
- **Consistency***: Every user of the DB has an identical view of the data at any given instant
- **Availability**: In the event of a failure, the database remains operational
- **Partition Tolerance**: The database can maintain operations in the event of the network's failing between two segments of the distributed system



* Note, the definition of Consistency in CAP is different from that of ACID.

CAP Theorem - Database View

- **Consistency + Availability:** System always responds with the latest data and every request gets a response, but may not be able to deal with network issues
- **Consistency + Partition Tolerance:** If system responds with data from a distributed store, it is always the latest, else data request is dropped.
- **Availability + Partition Tolerance:** System always sends responses based on distributed store, but may not be the absolute latest data.



CAP in Reality

What it is really saying:

- If you cannot limit the number of faults, requests can be directed to any server, and you insist on serving every request, then you cannot possibly be consistent.

But it is interpreted as:

- You must always give up something: consistency, availability, or tolerance to failure.

??
..

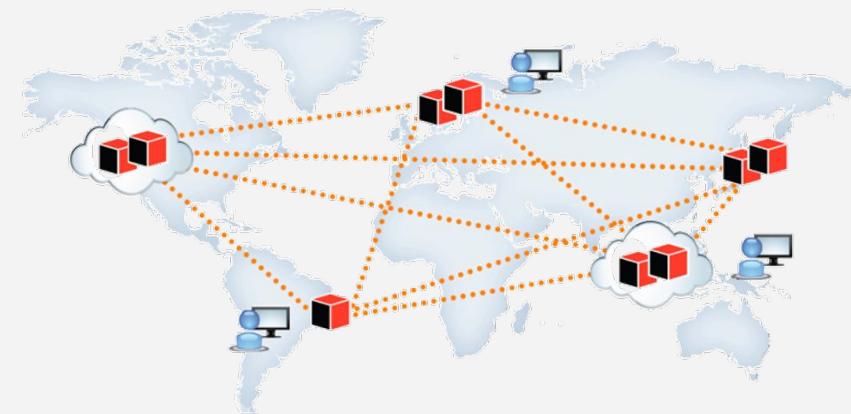
DS 4300

Replicating Data

Mark Fontenot, PhD
Northeastern University

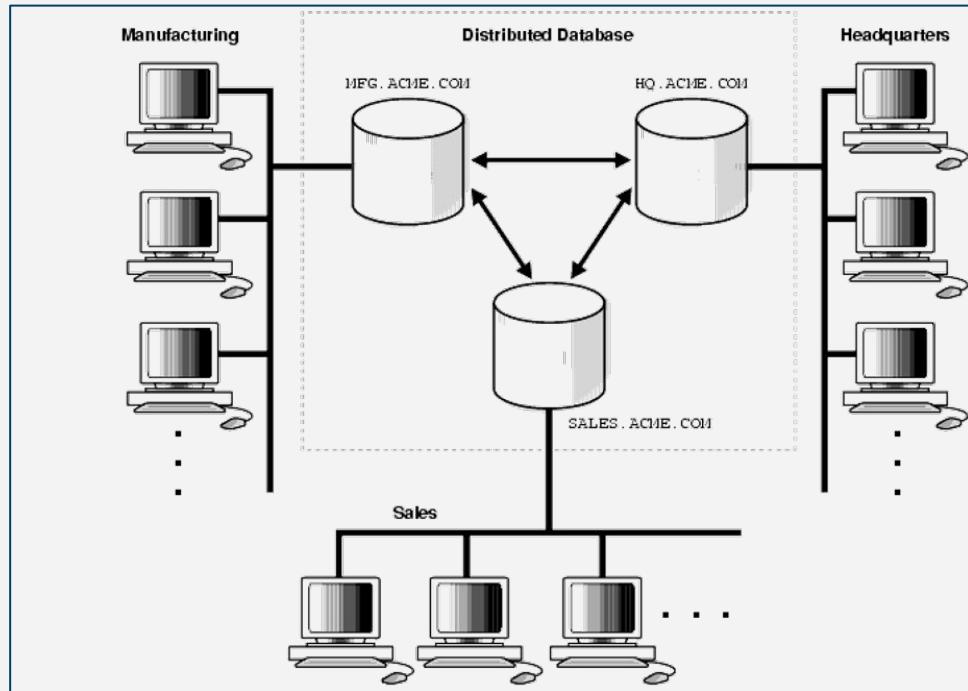
Distributing Data - Benefits

- **Scalability / High throughput:** Data volume or Read/Write load grows beyond the capacity of a single machine
- **Fault Tolerance / High Availability:** Your application needs to continue working even if one or more machines goes down.
- **Latency:** When you have users in different parts of the world you want to give them fast performance too



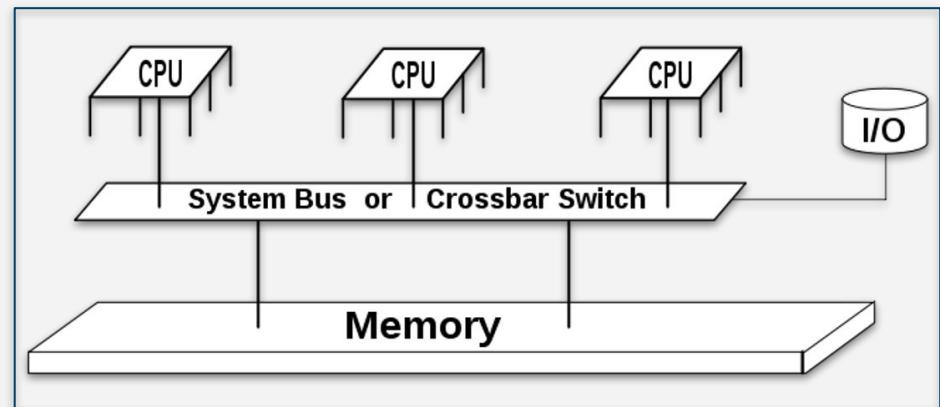
Distributed Data - Challenges

- **Consistency:** Updates must be propagated *across the network*.
- **Application Complexity:** Responsibility for reading and writing data in a distributed environment often falls to the application.



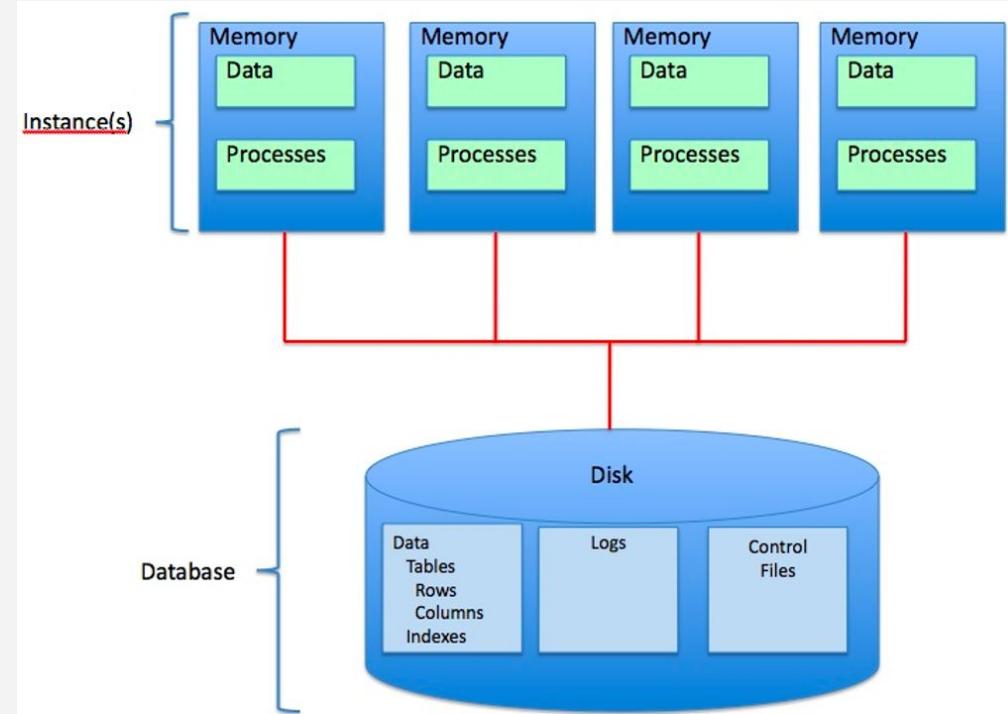
Vertical Scaling - Shared Memory Architectures

- Geographically Centralized server
- Some fault tolerance (via hot-swappable components)



Vertical Scaling - Shared Disk Architectures

- Machines are connected via a fast network
- Contention and the overhead of locking limit scalability (high-write volumes) ... BUT ok for Data Warehouse applications (high read volumes)



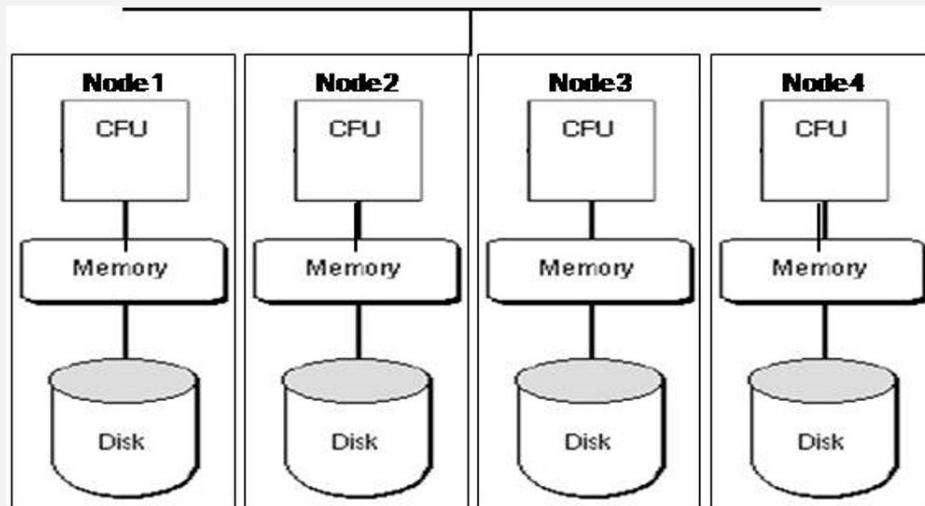
AWS EC2 Pricing - Oct 2024

Instance name	On-Demand hourly rate	vCPU	Memory	Storage	Network performance
t4g.nano	\$0.0042	2	0.5 GiB	EBS Only	Up to 5 Gigabit
t4g.micro	\$0.0084	2	1 GiB	EBS Only	Up to 5 Gigabit
t4g.small	\$0.0168	2	2 GiB	EBS Only	Up to 5 Gigabit
t3.medium	\$0.0416	2	4 GiB	EBS Only	Up to 5 Gigabit
t3.large	\$0.0832	2	8 GiB	EBS Only	Up to 5 Gigabit
t3.xlarge	\$0.1664	4	16 GiB	EBS Only	Up to 5 Gigabit
t3.2xlarge	\$0.3328	8	32 GiB	EBS Only	Up to 5 Gigabit
u-6tb1.112xlarge	\$54.60	448	3216 GiB	EBS Only	100 Gigabit
u-9tb1.112xlarge	\$81.90	448	3216 GiB	EBS Only	100 Gigabit
p5.48xlarge	\$98.32	192	2048 GiB	8 x 3840 GB SSD	3200 Gigabit
u-12tb1.112xlarge	\$109.20	448	12288 GiB	EBS Only	100 Gigabit

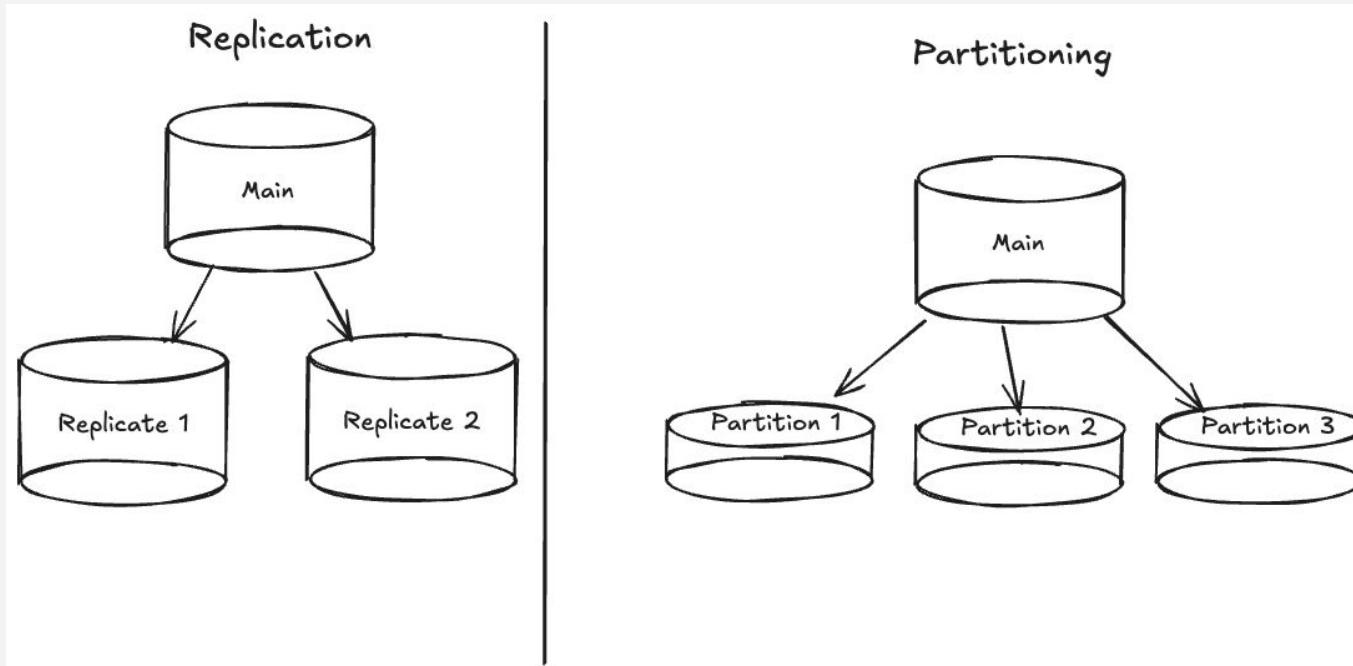
> \$78,000/month

Horizontal Scaling - Shared Nothing Architectures

- Each node has its own CPU, memory, and disk
- Coordination via application layer using conventional network
- Geographically distributed
- Commodity hardware



Data - Replication vs Partitioning



Replicates have
same data as Main

Partitions have a
subset of the data

Replication

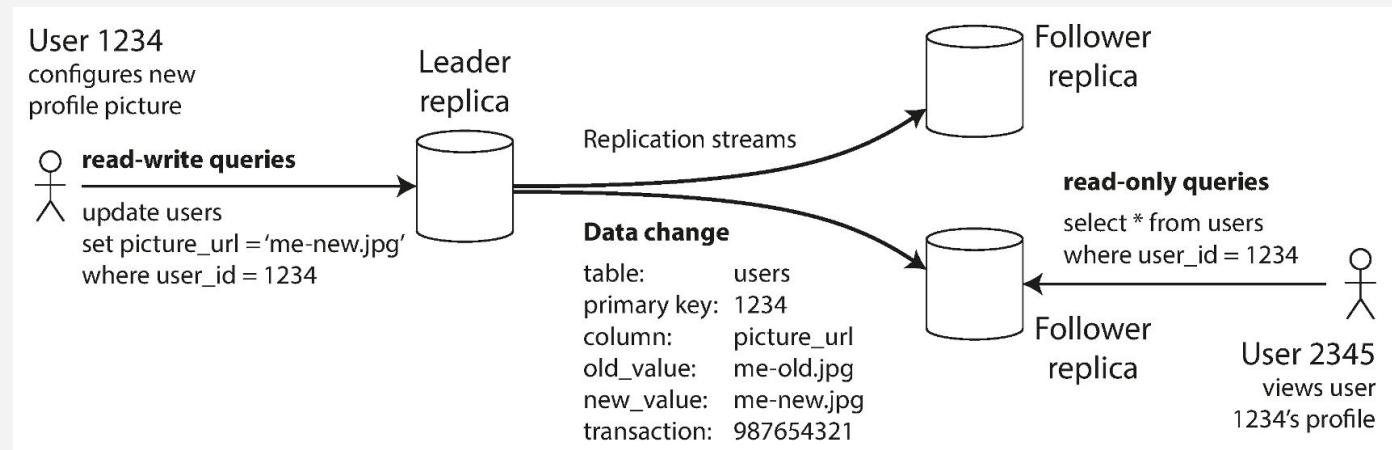
Common Strategies for Replication

- Single leader model
- Multiple leader model
- Leaderless model

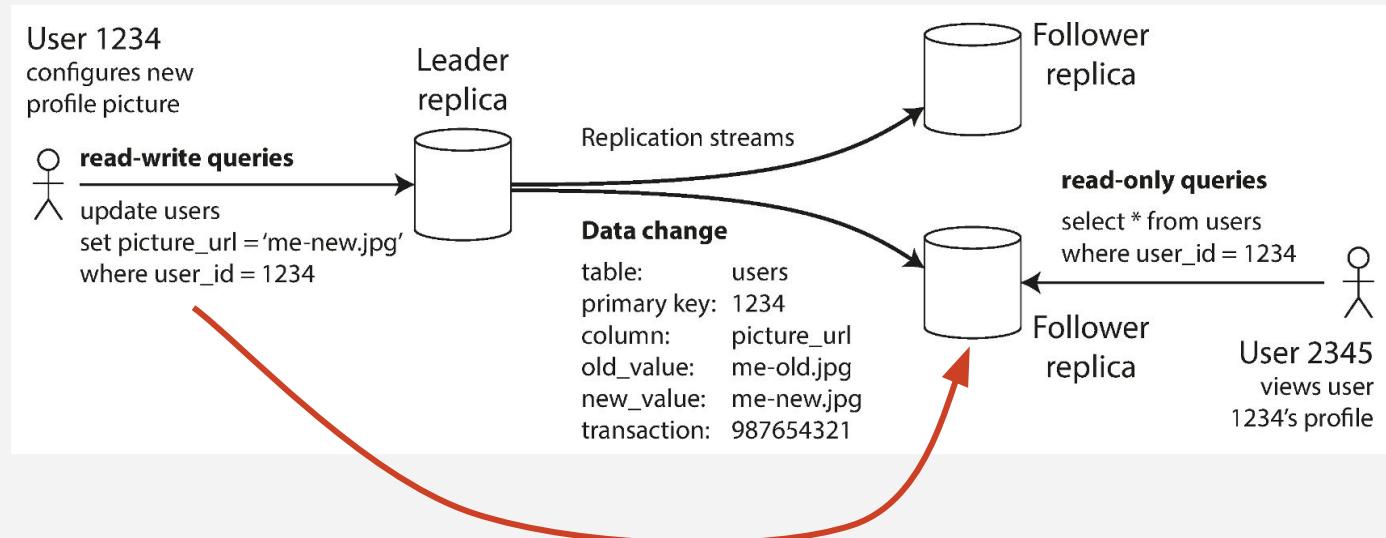
Distributed databases usually adopt one of these strategies.

Leader-Based Replication

- All writes from clients go to the leader
- Leader sends replication info to the followers
- Followers process the instructions from the leader
- Clients can read from either the leader or followers



Leader-Based Replication



This write could NOT be sent to one of the
followers... only the leader.

Leader-Based Replication - Very Common Strategy

Relational:

- MySQL,
- Oracle,
- SQL Server,
- PostgreSQL

NoSQL:

- MongoDB,
- RethinkDB (realtime web apps),
- Espresso (LinkedIn)

Messaging Brokers: Kafka, RabbitMQ

How Is Replication Info Transmitted to Followers?

Replication Method	Description
Statement-based	Send INSERT, UPDATE, DELETEs to replica. Simple but error-prone due to non-deterministic functions like now(), trigger side-effects, and difficulty in handling concurrent transactions.
Write-ahead Log (WAL)	A byte-level specific log of every change to the database. Leader and all followers must implement the same storage engine and makes upgrades difficult.
Logical (row-based) Log	For relational DBs: Inserted rows, modified rows (before and after), deleted rows. A transaction log will identify all the rows that changed in each transaction and how they changed. Logical logs are decoupled from the storage engine and easier to parse.
Trigger-based	Changes are logged to a separate table whenever a trigger fires in response to an insert, update, or delete. Flexible because you can have application specific replication, but also more error prone.

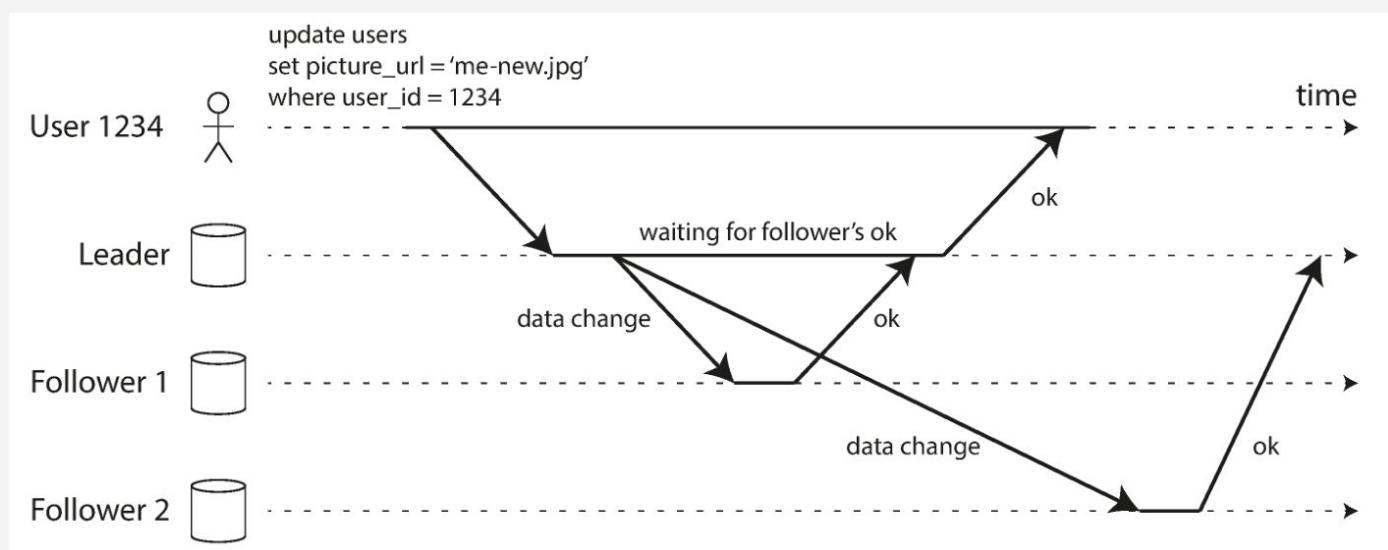
Synchronous vs Asynchronous Replication

Synchronous: Leader waits for a response from the follower

Asynchronous: Leader doesn't wait for confirmation.

Synchronous:

Asynchronous:



What Happens When the Leader Fails?



Challenges: How do we pick a new Leader Node?

- Consensus strategy – perhaps based on who has the most updates?
- Use a controller node to appoint new leader?

AND... *how do we configure clients to start writing to the new leader?*

What Happens When the Leader Fails?



More Challenges:

- If asynchronous replication is used, new leader may not have all the writes
How do we recover the lost writes? Or do we simply discard?
- After (if?) the old leader recovers, how do we avoid having multiple leaders receiving conflicting data? (Split brain: no way to resolve conflicting requests.)
- Leader failure detection. Optimal timeout is tricky.

Replication Lag

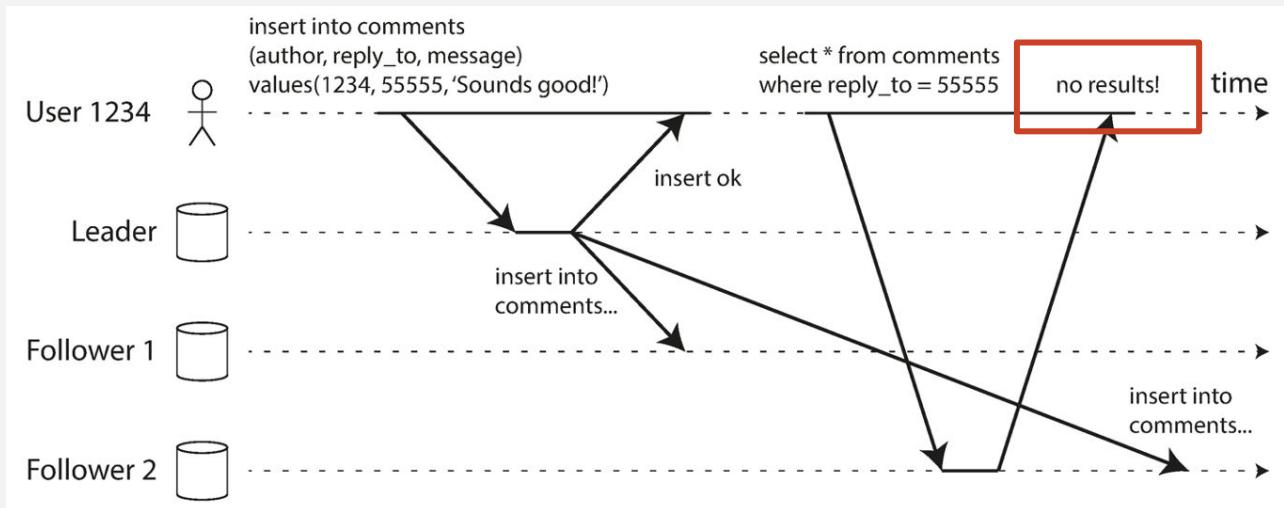
Replication Lag refers to the time it takes for writes on the leader to be reflected on all of the followers.

- **Synchronous replication:** Replication lag causes writes to be slower and the system to be more brittle as num followers increases.
- **Asynchronous replication:** We maintain availability *but at the cost of delayed or eventual consistency*. This delay is called the *inconsistency window*.

Read-after-Write Consistency

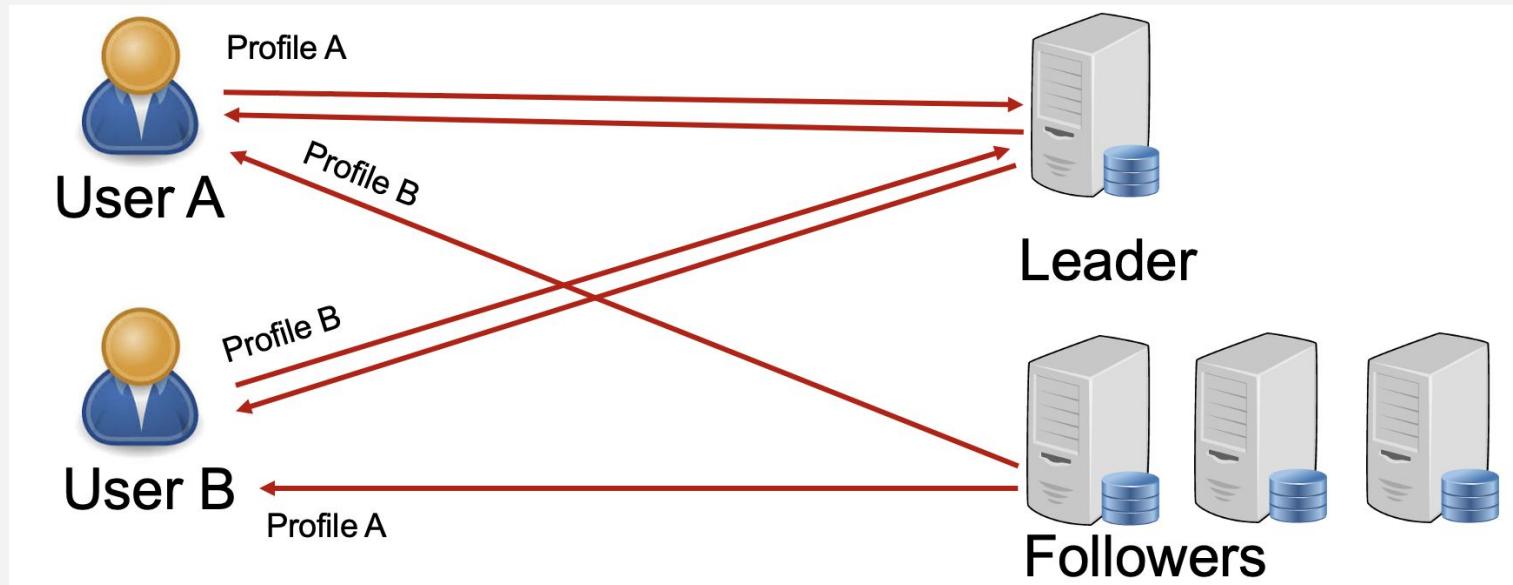
Scenario - you're adding a comment to a Reddit post... after you click Submit and are back at the main post, your comment should show up for you.

- Less important for other users to see your comment as immediately.



Implementing Read-After-Write Consistency

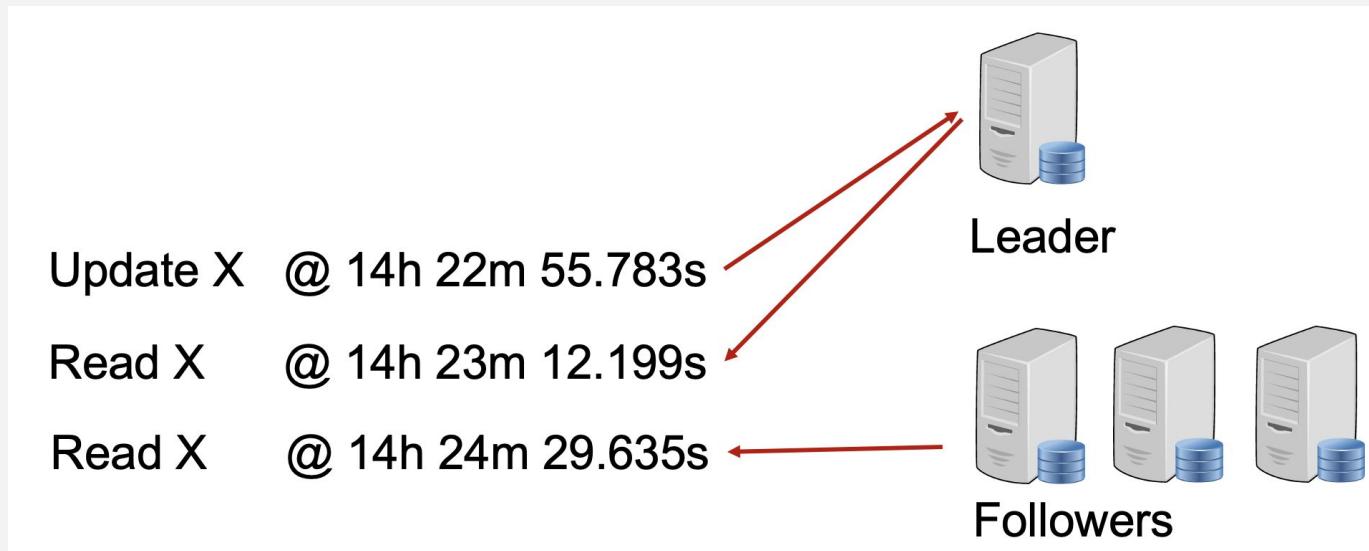
Method 1: Modifiable data (from the client's perspective) is always read from the leader.



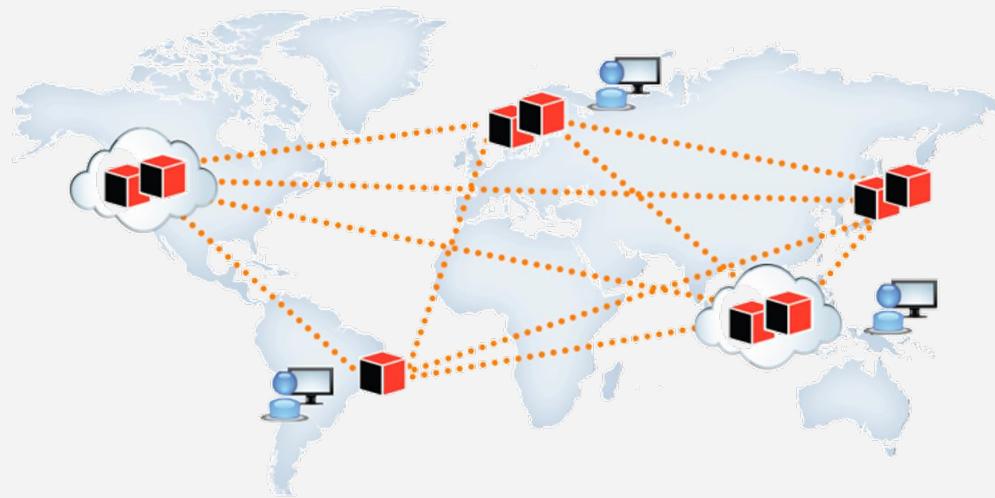
Implementing Read-After-Write Consistency

Method 2: Dynamically switch to reading from leader for “recently updated” data.

- For example, have a policy that all requests within one minute of last update come from leader.



But... This Can Create Its Own Challenges

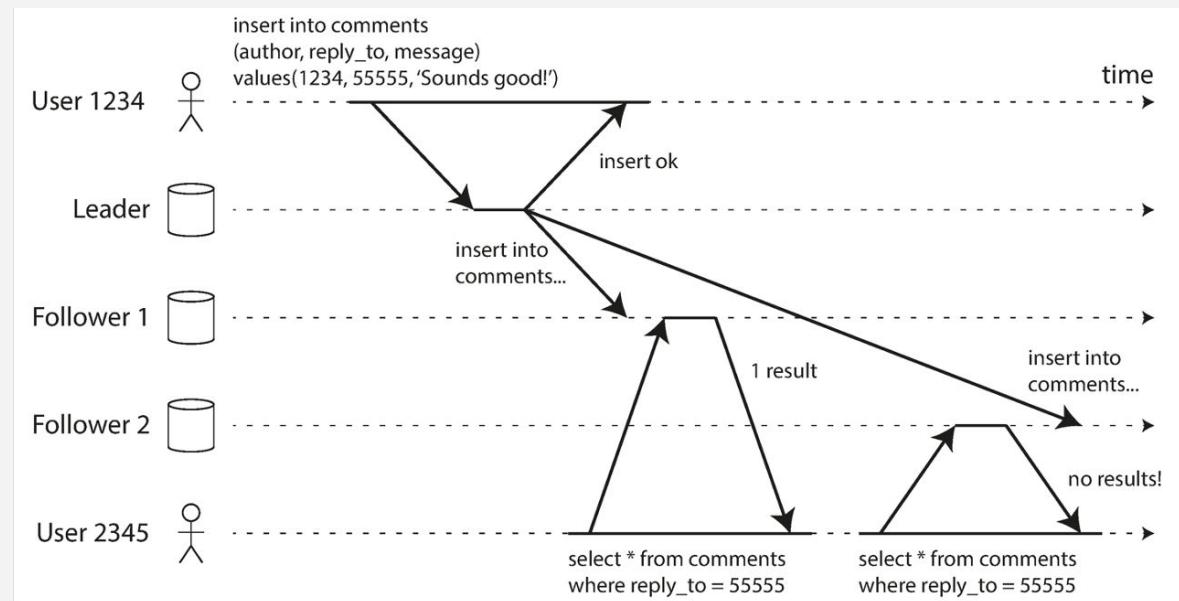


We created followers so they would be proximal to users. BUT... now we have to route requests to distant leaders when reading modifiable data?? :(

Monotonic Read Consistency

Monotonic read anomalies:
occur when a user reads
values out of order from
multiple followers.

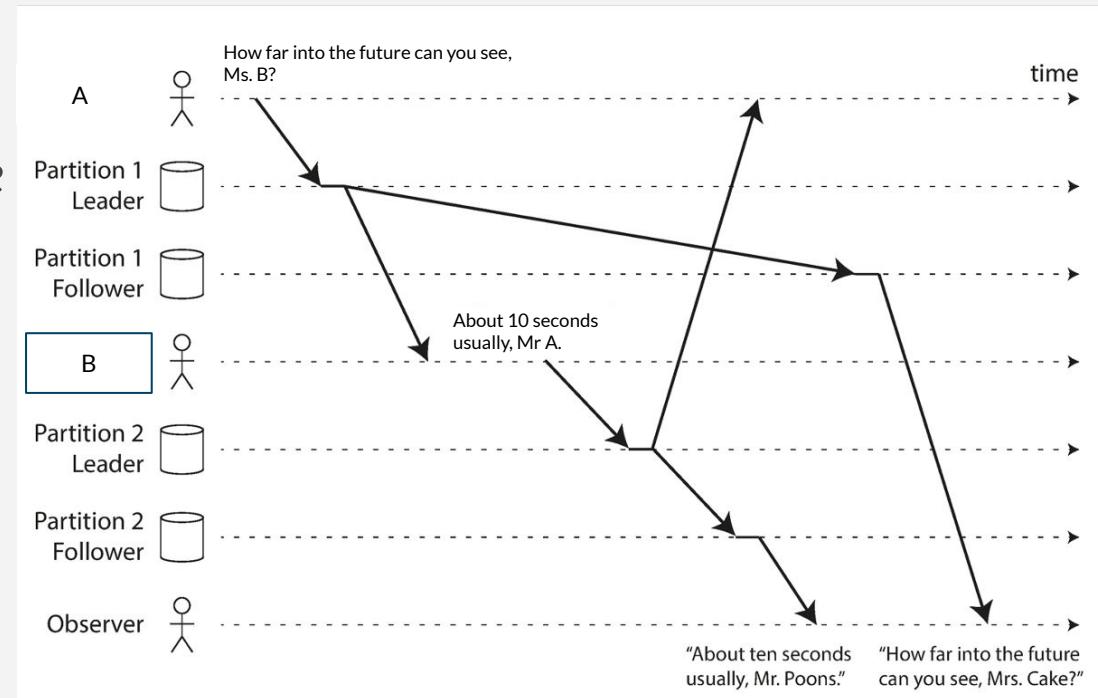
Monotonic read consistency:
ensures that when a user
makes multiple reads, they
will not read older data after
previously reading newer
data.



Consistent Prefix Reads

Reading data out of order can occur if different partitions replicate data at different rates. There is *no global write consistency*.

Consistent Prefix Read Guarantee - ensures that if a sequence of writes happens in a certain order, anyone reading those writes will see them appear in the same order.



??
..

DS 4300

Large Scale Information Storage and Retrieval

B+ Tree Walkthrough

Mark Fontenot, PhD
Northeastern University

Insert: 42, 21, 63, 89

B+ Tree : $m = 4$

21	42	63	89
----	----	----	----

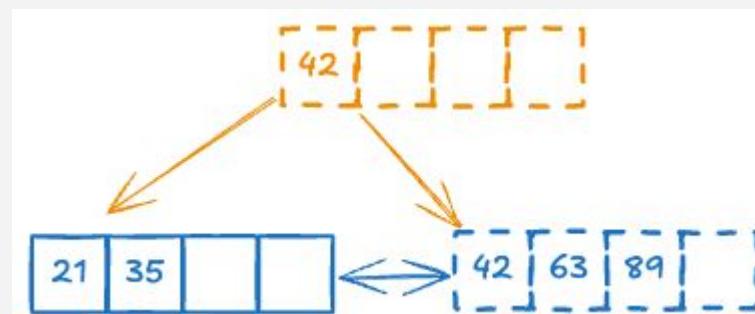
- Initially, the first node is a leaf node AND root node.
- 21, 42, ... represent keys of some set of K:V pairs
- Leaf nodes store keys and data, although data not shown
- Inserting another key will cause the node to split.

Insert: 35

B+ Tree : $m = 4$

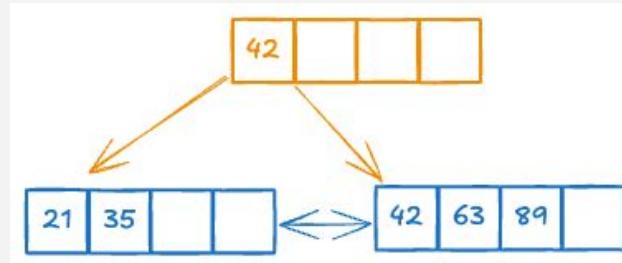


- Leaf node needs to split to accommodate 35. New leaf node allocated to the right of existing node
- 5/2 values stay in original node; remaining values moved to new node
- Smallest value from new leaf node (42) is copied up to the parent, which needs to be created in this case. It will be an *internal* node.

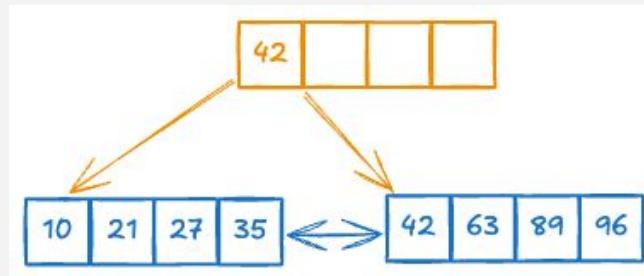


Insert: 10, 27, 96

B+ Tree : $m = 4$

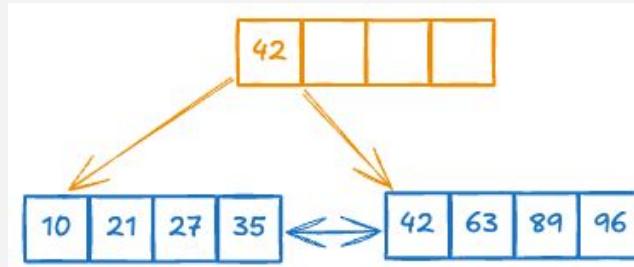


- The insert process starts at the root node. The keys of the root node are searched to find out which child node we need to descend to.
 - EX: 10. Since $10 < 42$, we follow the pointer to the left of 42
- Note - none of these new values cause a node to split

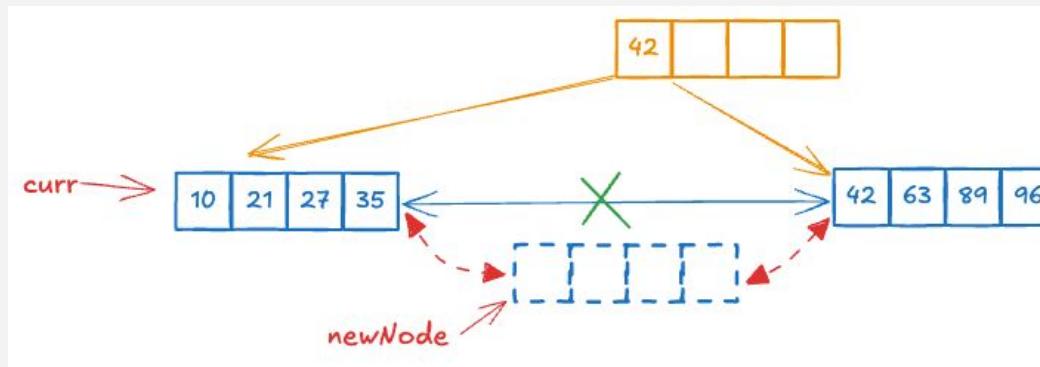


B+ Tree : $m = 4$

Insert: 30

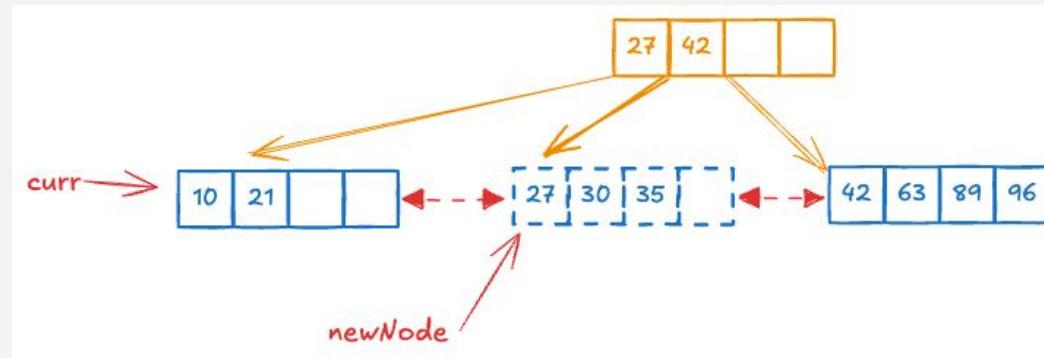


- Starting at root, we descend to the left-most child (we'll call *curr*).
 - *curr* is a leaf node. Thus, we insert 30 into *curr*.
 - BUT *curr* is full. So we have to split.
 - Create a new node to the right of *curr*, temporarily called *newNode*.
 - Insert *newNode* into the doubly linked list of leaf nodes.

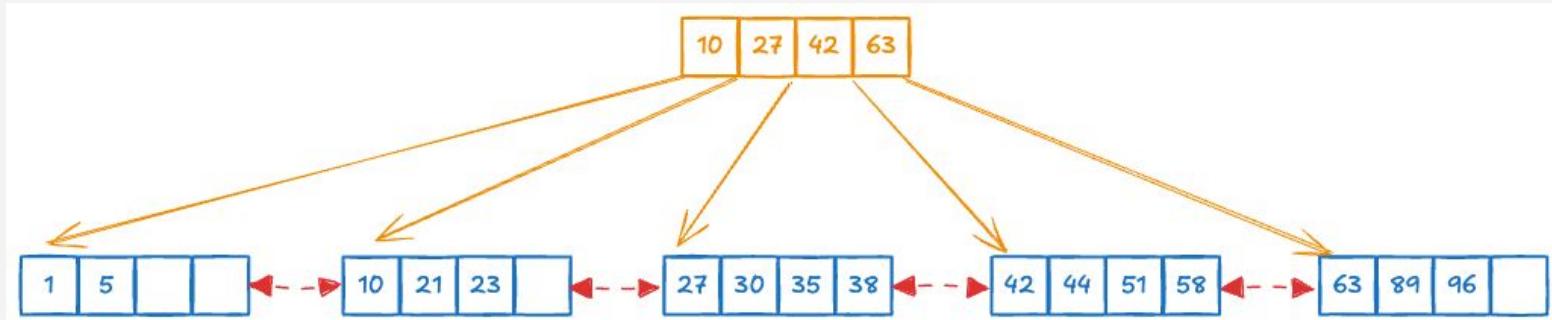


Insert: 30 cont'd.

- re-distribute the keys
- copy the smallest key (27 in this case) from newNode to parent; rearrange keys and pointers in parent node.
- Parent of newNode is also root. So, nothing else to do.



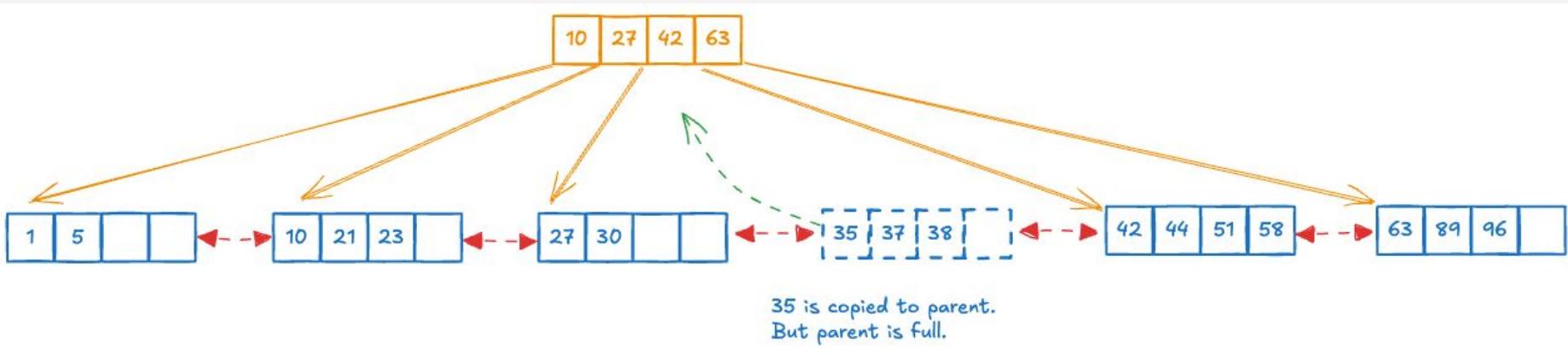
Fast forward to this state of the tree...



- Observation: The root node is full.
 - The next insertion that splits a leaf will cause the root to split, and thus the tree will get 1 level deeper.

Insert 37. Step 1

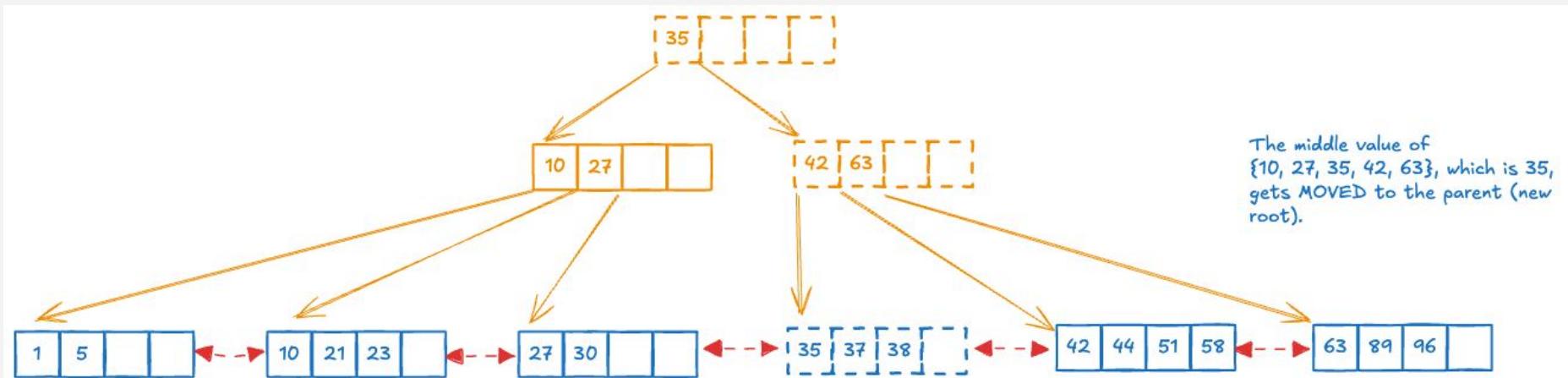
B+ Tree : $m = 4$



Insert 37. Step 2.

B+ Tree : $m = 4$

- When splitting an internal node, we **move** the middle element to the parent (instead of copying it).
- In this particular tree, that means we have to create a new internal node which is also now the root.



DS 4300

NoSQL & KV DBs

Mark Fontenot, PhD
Northeastern University

Distributed DBs and ACID - Pessimistic Concurrency

- ACID transactions
 - Focuses on “data safety”
 - considered a pessimistic concurrency model because it assumes one transaction has to protect itself from other transactions
 - IOW, it assumes that if something can go wrong, it will.
 - Conflicts are prevented by locking resources until a transaction is complete (there are both read and write locks)
 - Write Lock Analogy → borrowing a book from a library... If you have it, no one else can.

See <https://www.freecodecamp.org/news/how-databases-guarantee-isolation> for more for a deeper dive.

Optimistic Concurrency

- Transactions do not obtain locks on data when they read or write
- *Optimistic* because it assumes conflicts are unlikely to occur
 - Even if there is a conflict, everything will still be OK.
- But how?
 - Add last update timestamp and version number columns to every table... read them when changing. THEN, check at the end of transaction to see if any other transaction has caused them to be modified.

Optimistic Concurrency

- Low Conflict Systems (backups, analytical dbs, etc.)
 - Read heavy systems
 - the conflicts that arise can be handled by rolling back and re-running a transaction that notices a conflict.
 - So, optimistic concurrency works well - allows for higher concurrency
- High Conflict Systems
 - rolling back and rerunning transactions that encounter a conflict → less efficient
 - So, a locking scheme (pessimistic model) might be preferable

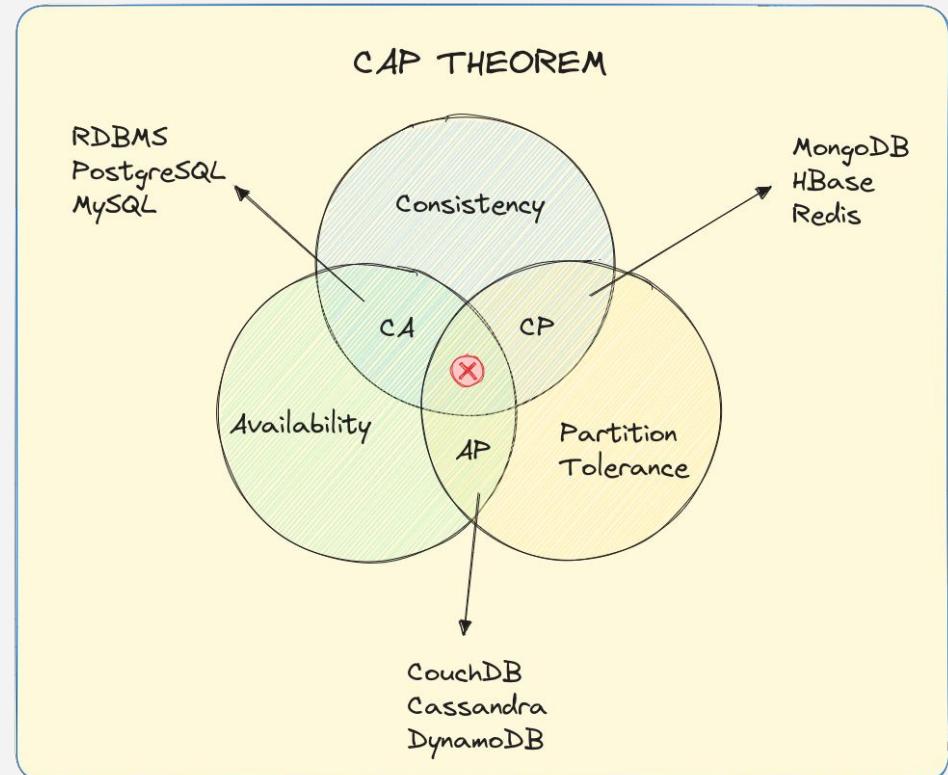
NoSQL

- “NoSQL” first used in 1998 by Carlo Strozzi to describe his relational database system that *did not use SQL*.
- More common, modern meaning is “Not Only SQL”
- *But*, sometimes thought of as non-relational DBs
- Idea originally developed, in part, as a response to processing unstructured web-based data.

CAP Theorem Review

You can have 2, but not 3, of the following:

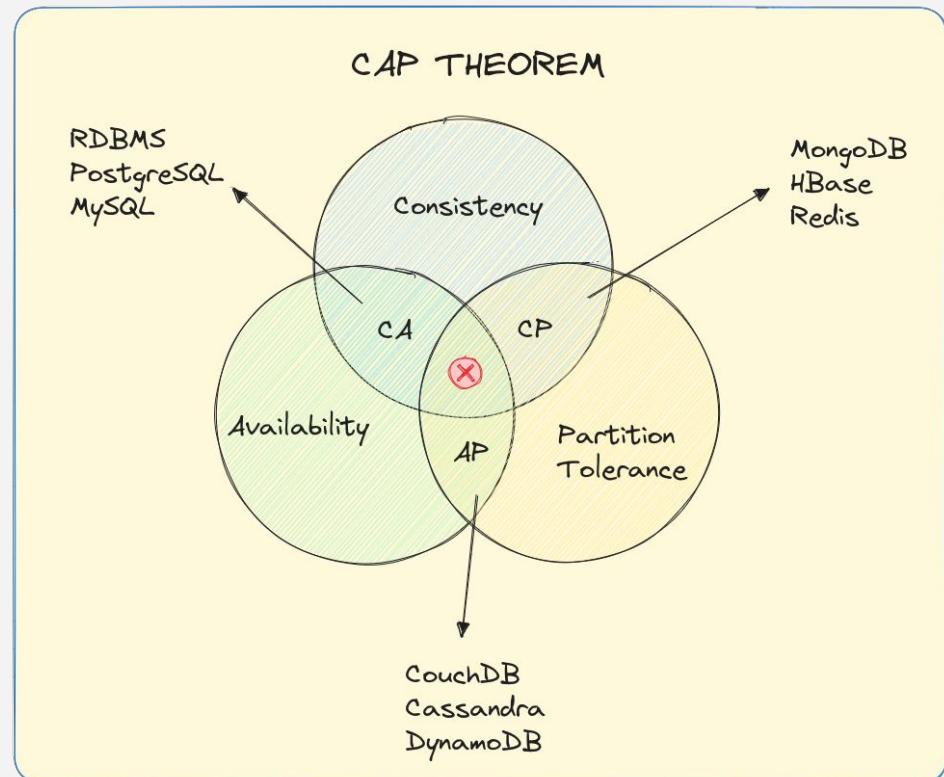
- **Consistency***: Every user of the DB has an identical view of the data at any given instant
- **Availability**: In the event of a failure, the database system remains operational
- **Partition Tolerance**: The database can maintain operations in the event of the network's failing between two segments of the distributed system



* Note, the definition of Consistency in CAP is different from that of ACID.

CAP Theorem Review

- **Consistency + Availability:** System always responds with the latest data and every request gets a response, but may not be able to deal with network partitions
- **Consistency + Partition Tolerance:** If system responds with data from the distrib. system, it is always the latest, else data request is dropped.
- **Availability + Partition Tolerance:** System always sends responses based on distributed store, but may not be the absolute latest data.



ACID Alternative for Distrib Systems - BASE

- **Basically Available**

- Guarantees the availability of the data (per CAP), but response can be “failure”/“unreliable” because the data is in an inconsistent or changing state
- System appears to work most of the time

ACID Alternative for Distrib Systems - BASE

- **Soft State** - The state of the system could change over time, even w/o input. Changes could be result of *eventual consistency*.
 - Data stores don't have to be write-consistent
 - Replicas don't have to be mutually consistent

ACID Alternative for Distrib Systems - BASE

- Eventual Consistency - The system will eventually become consistent
 - All writes will eventually stop so all nodes/replicas can be updated

Categories of NoSQL DBs - Review

First Up → Key-Value Databases

Key Value Stores

key = value

- Key-value stores are designed around:
 - *simplicity*
 - the data model is extremely simple
 - comparatively, tables in a RDBMS are very complex.
 - lends itself to simple CRUD ops and API creation

Key Value Stores

key = value

- Key-value stores are designed around:
 - *speed*
 - usually deployed as in-memory DB
 - retrieving a *value* given its *key* is typically a O(1) op b/c hash tables or similar data structs used under the hood
 - no concept of complex queries or joins... they slow things down

Key Value Stores

key = value

- Key-value stores are designed around:
 - *scalability*
 - Horizontal Scaling is simple - add more nodes
 - Typically concerned with *eventual consistency*, meaning in a distributed environment, the only guarantee is that all nodes will *eventually* converge on the same value.

KV DS Use Cases

- EDA/Experimentation Results Store
 - store intermediate results from data preprocessing and EDA
 - store experiment or testing (A/B) results w/o prod db
- Feature Store
 - store frequently accessed feature → low-latency retrieval for model training and prediction
- Model Monitoring
 - store key metrics about performance of model, for example, in real-time inferencing.

KV SWE Use Cases

- Storing Session Information
 - everything about the current *session* can be stored via a single PUT or POST and retrieved with a single GET VERY Fast
- User Profiles & Preferences
 - User info could be obtained with a single GET operation... language, TZ, product or UI preferences
- Shopping Cart Data
 - Cart data is tied to the user
 - needs to be available across browsers, machines, sessions
- Caching Layer:
 - In front of a disk-based database

Redis DB

- Redis (Remote Directory Server)
 - Open source, in-memory database
 - Sometimes called a data structure store
 - Primarily a KV store, but can be used with other models: Graph, Spatial, Full Text Search, Vector, Time Series
 - From db-engines.com Ranking of KV Stores:

Rank			DBMS	Database Model	Score		
Oct 2024	Sep 2024	Oct 2023			Oct 2024	Sep 2024	Oct 2023
1.	1.	1.	Redis	Key-value, Multi-model	149.63	+0.20	-13.33
2.	2.	2.	Amazon DynamoDB	Multi-model	71.85	+1.78	-9.07
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	24.50	-0.47	-9.80
4.	4.	4.	Memcached	Key-value	17.79	+0.95	-3.05
5.	5.	5.	etcd	Key-value	7.17	+0.12	-1.57
6.	↑7.	↑8.	Aerospike	Multi-model	5.57	+0.41	-0.86
7.	↓6.	↓6.	Hazelcast	Key-value, Multi-model	5.57	-0.16	-2.60
8.	8.	↓7.	Fhcache	Key-value	4.76	-0.03	-1.79

- It is considered an in-memory database system, but...
 - Supports durability of data by: a) essentially saving snapshots to disk at specific intervals or b) append-only file which is a journal of changes that can be used for roll-forward if there is a failure
- Originally developed in 2009 in C++
- Can be very fast ... > 100,000 SET ops / second
- Rich collection of commands
- Does NOT handle complex data. No secondary indexes.
Only supports lookup by Key.

Redis Data Types

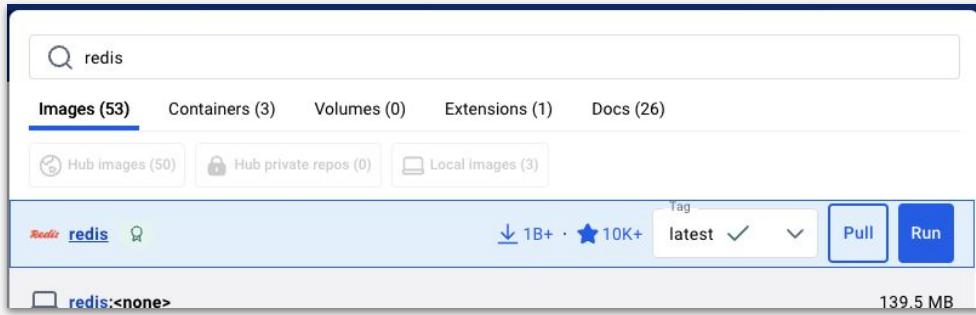
Keys:

- usually strings but can be any binary sequence

Values:

- Strings
- Lists (linked lists)
- Sets (unique unsorted string elements)
- Sorted Sets
- Hashes (string → string)
- Geospatial data

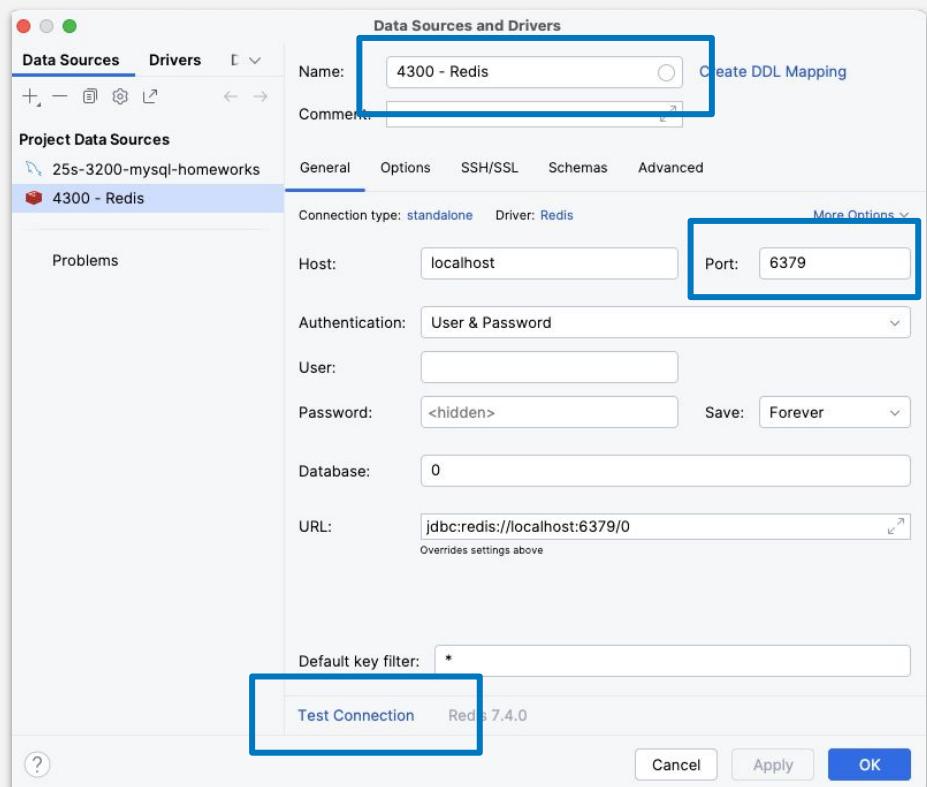
Setting Up Redis in Docker



- In Docker Desktop, search for Redis.
- Pull/Run the latest image (see above)
 - Optional Settings: add **6379** to Ports to expose that port so we can connect to it.
- Normally, you **would not** expose the Redis port for security reasons
 - If you did this in a prod environment, major security hole.
 - Notice, we didn't set a password...

Connecting from DataGrip

- File > New > Data Source > Redis
- Give the Data Source a Name
- Make sure the port is 6379
- Test the connection 



Redis Database and Interaction

- Redis provides 16 databases by default
 - They are numbered 0 to 15
 - There is 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 language libraries available as well.

```
SET ds4300 "I Love AVL Trees!"
```

```
SET cs3200 "I Love SQL!"
```

```
KEYS *
```

```
GET cs3200
```

```
DEL ds4300
```

Foundation Data Type - String

- Sequence of bytes - text, serialized objects, bin arrays
- Simplest data type
- Maps a string to another string
- Use Cases:
 - caching frequently accessed HTML/CSS/JS fragments
 - config settings, user settings info, token management
 - counting web page/app screen views OR rate limiting

Some Initial Basic Commands

- **SET** /path/to/resource 0
SET user:1 "John Doe"
GET /path/to/resource
EXISTS user:1
DEL user:1
KEYS user*
- **SELECT** 5
 - select a different database

Some Basic Commands

- **SET** someValue 0
- **INCR** someValue #increment by 1
- **INCRBY** someValue 10 #increment by 10
- **DECR** someValue #decrement by 1
- **DECRBY** someValue 5 #decrement by 5
 - INCR parses the value as int and increments (or adds to value)
- **SETNX** key value
 - only sets value to key if key does not already exist

Hash Type

- Value of KV entry is a collection of *field-value* pairs
- Use Cases:
 - Can be used to represent basic objects/structures
 - number of field/value pairs per hash is $2^{32}-1$
 - practical limit: available system resources (e.g. memory)
 - Session information management
 - User/Event tracking (could include TTL)
 - Active Session Tracking (all sessions under one hash key)

Hash Commands

```
HSET bike:1 model Demios brand Ergonom price 1971
```

```
HGET bike:1 model
```

```
HGET bike:1 price
```

```
HGETALL bike:1
```

```
HMGET bike:1 model price weight
```

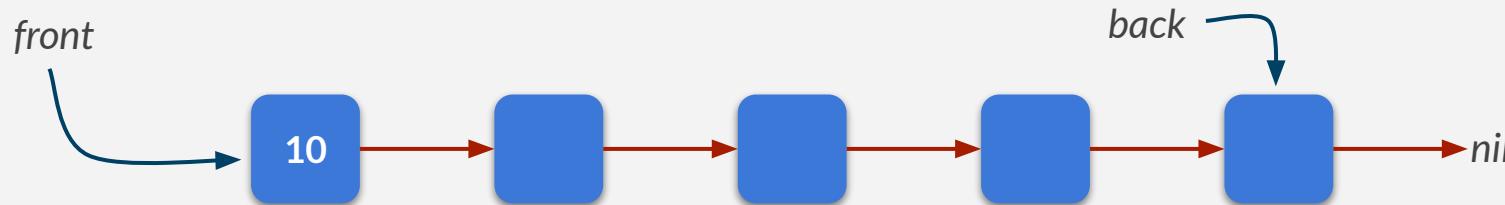
```
HINCRBY bike:1 price 100
```

What is returned?

List Type

- Value of KV Pair is **linked lists** of string values
- Use Cases:
 - implementation of stacks and queues
 - queue management & message passing queues (producer/consumer model)
 - logging systems (easy to keep in chronological order)
 - build social media streams/feeds
 - message history in a chat application
 - batch processing by queueing up a set of tasks to be executed sequentially at a later time

Linked Lists Crash Course



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

List Commands - Queue

Queue-like Ops

LPUSH bikes:repairs bike:1

LPUSH bikes:repairs bike:2

RPOP bikes:repairs

RPOP biles:repairs

List Commands - Stack

Stack-like Ops

```
LPUSH bikes:repairs bike:1
```

```
LPUSH bikes:repairs bike:2
```

```
LPOP bikes:repairs
```

```
LPOP biles:repairs
```

List Commands - Others

Other List Ops

LLEN mylist

LPUSH mylist "one"
LPUSH mylist "two"
LPUSH mylist "three"

LRANGE <key> <start> <stop>

LRANGE mylist 0 3

LRANGE mylist 0 0

LRANGE mylist -2 -1

1) "three"
2) "two"
3) "one"

1) "three"

1) "two"
2) "one"

JSON Type

- Full support of the JSON standard
- Uses JSONPath syntax for parsing/navigating a JSON document
- Internally, stored in binary in a tree-structure → fast access to sub elements

Set Type

- Unordered collection of unique strings (members)
- Use Cases:
 - track unique items (IP addresses visiting a site, page, screen)
 - primitive relation (set of all students in DS4300)
 - access control lists for users and permission structures
 - social network friends lists and/or group membership
- Supports set operations!!

Set Commands

```
SADD ds4300 "Mark"
```

```
SADD ds4300 "Sam"
```

```
SADD cs3200 "Nick"
```

```
SADD cs3200 "Sam"
```

```
SISMEMBER ds4300 "Mark"
```

```
SISMEMBER ds4300 "Nick"
```

```
SCARD ds4300
```

Set Commands

```
SADD ds4300 "Mark"  
SADD ds4300 "Sam"  
SADD cs3200 "Nick"  
SADD cs3200 "Sam"
```

```
SCARD ds4300
```

```
SINTER ds4300 cs3200
```

```
SDIFF ds4300 cs3200
```

```
SREM ds4300 "Mark"
```

```
SRANDMEMBER ds4300
```

??
?

DS 4300

Redis + Python

Mark Fontenot, PhD
Northeastern University

Redis-py

- Redis-py is the standard client for Python.
- Maintained by the Redis Company itself
- GitHub Repo: [redis/redis-py](#)
- In your 4300 Conda Environment:

```
pip install redis
```

Connecting to the Server

```
import redis
redis_client = redis.Redis(host='localhost',
                           port=6379,
                           db=2,
                           decode_responses=True)
```

- For your Docker deployment, host could be *localhost* or *127.0.0.1*
- Port is the port mapping given when you created the container (probably the default 6379)
- db is the database 0-15 you want to connect to
- decode_responses → data comes back from server as bytes. Setting this true converter them (decodes) to strings.

Redis Command List

- Full List > [here](#) <
- Use Filter to get to command for the particular data structure you're targeting (list, hash, set, etc.)
- Redis.py Documentation > [here](#) <
- The next slides are not meant to be an exhaustive list of commands, only some highlights. Check the documentation for a complete list.

String Commands

```
# r represents the Redis client object  
r.set('clickCount:/abc', 0)  
val = r.get('clickCount:/abc')  
r.incr('clickCount:/abc')  
ret_val = r.get('clickCount:/abc')  
print(f'click count = {ret_val}')
```

String Commands - 2

```
# r represents the Redis client object
redis_client.mset({'key1': 'val1',
                    'key2': 'val2',
                    'key3': 'val3'})
print(redis_client.mget('key1',
                        'key2',
                        'key3'))
# returns as list ['val1', 'val2', 'val3']
```

String Commands - 3

- set(), mset(), setex(), msetnx(), setnx()
- get(), mget(), getex(), getdel()
- incr(), decr(), incrby(), decrby()
- strlen(), append()

List Commands - 1

```
# create list: key = 'names'  
# values = ['mark', 'sam', 'nick']  
redis_client.rpush('names',  
                   'mark', 'sam', 'nick')  
  
# prints ['mark', 'sam', 'nick']  
print(redis_client.lrange('names', 0, -1))
```

List Commands - 2

- `lpush()`, `lpop()`, `lset()`, `lrem()`
- `rpush()`, `rpop()`
- `lrange()`, `llen()`, `lpos()`
- Other commands include moving elements between lists, popping from multiple lists at the same time, etc.

Hash Commands - 1

```
redis_client.hset('user-session:123',  
    mapping={'first': 'Sam',  
              'last': 'Uelle',  
              'company': 'Redis',  
              'age': 30  
})
```

```
# prints:  
#{'name': 'Sam', 'surname': 'Uelle', 'company': 'Redis', 'age': '30'}  
print(redis_client.hgetall('user-session:123'))
```

Hash Commands - 2

- hset(), hget(), hgetall()
- hkeys()
- hdel(), hexists(), hlen(), hstrlen()

Redis Pipelines

- Helps avoid multiple related calls to the server → less network overhead

```
r = redis.Redis(decode_responses=True)
pipe = r.pipeline()

for i in range(5):
    pipe.set(f"seat:{i}", f"#{i}")

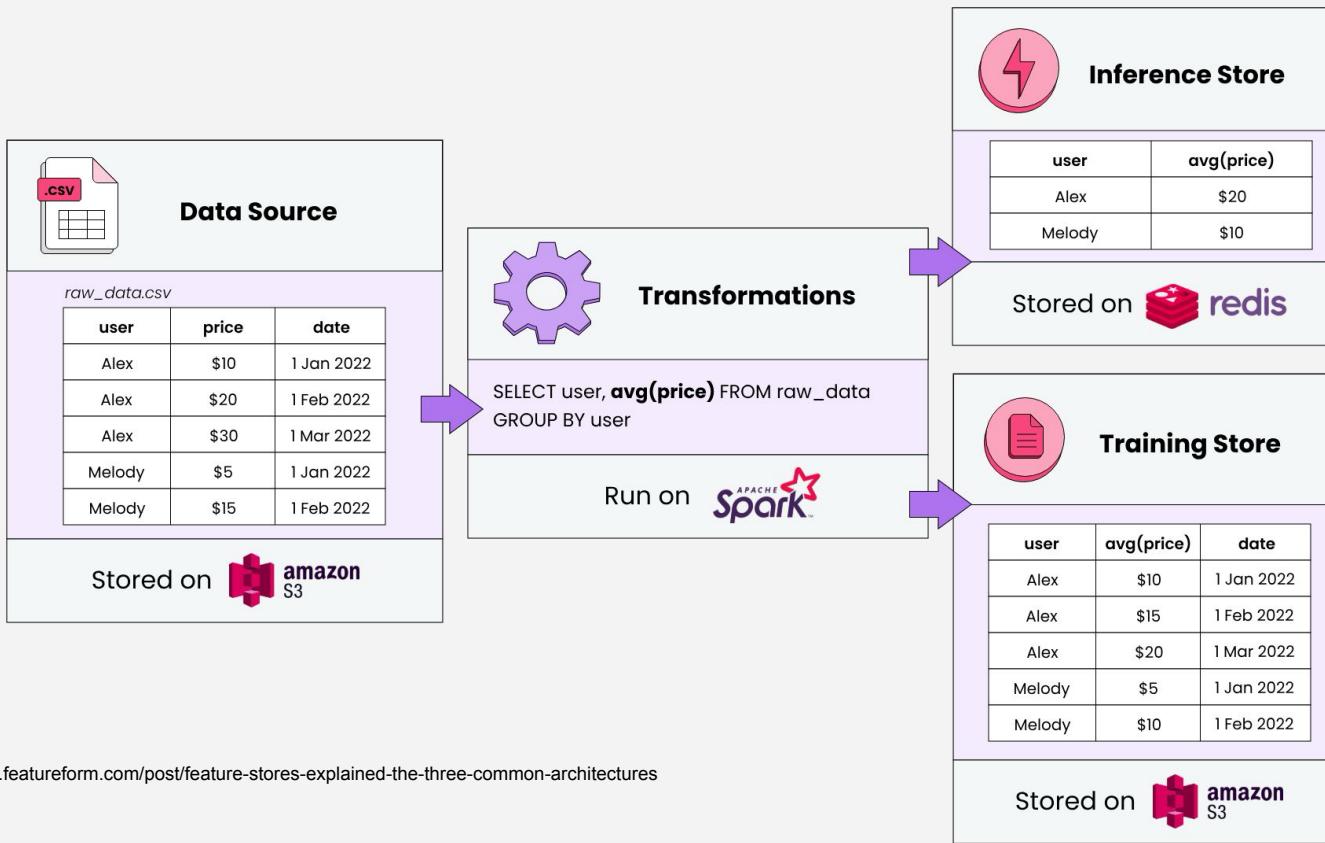
set_5_result = pipe.execute()
print(set_5_result) # >>> [True, True, True, True, True]

pipe = r.pipeline()

# "Chain" pipeline commands together.
get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
print(get_3_result) # >>> ['#0', '#3', '#4']
```

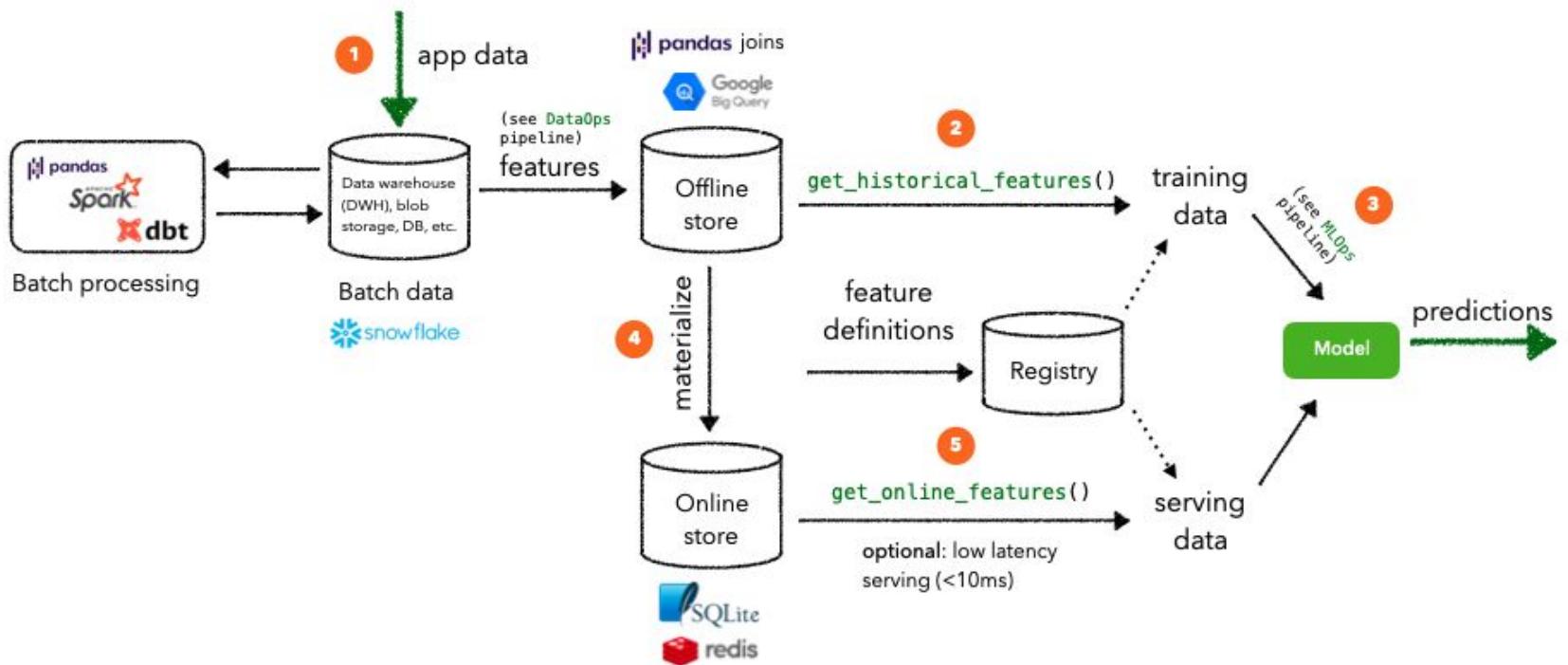
Redis in Context

Redis in ML - Simplified Example



Source: <https://www.featureform.com/post/feature-stores-explained-the-three-common-architectures>

Redis in DS/ML



Source: <https://madewithml.com/courses/mlops/feature-store/>

DS 4300

Document Databases & MongoDB

Mark Fontenot, PhD
Northeastern University

Document Database

A Document Database is a non-relational database that stores data as structured documents, usually in JSON.

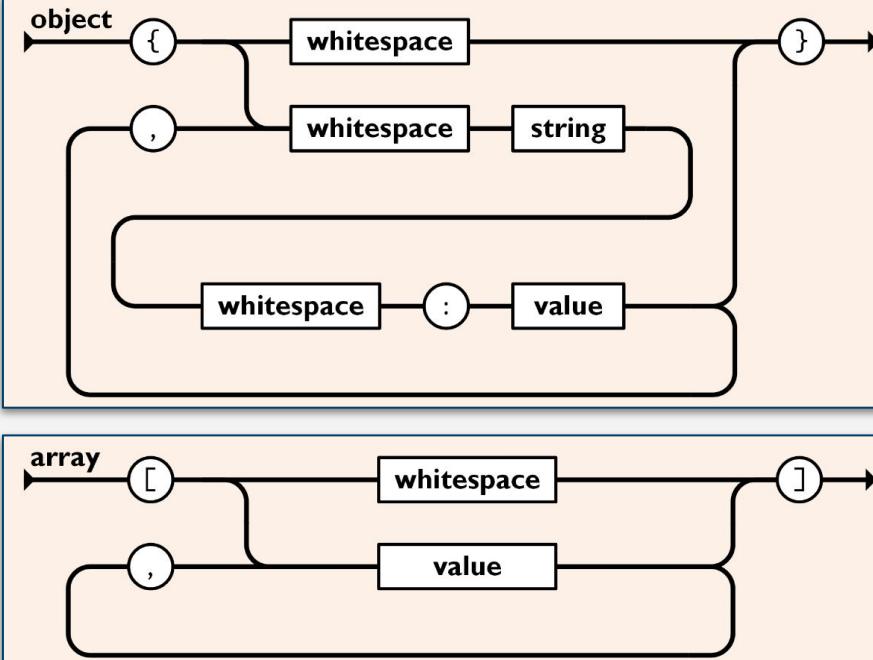
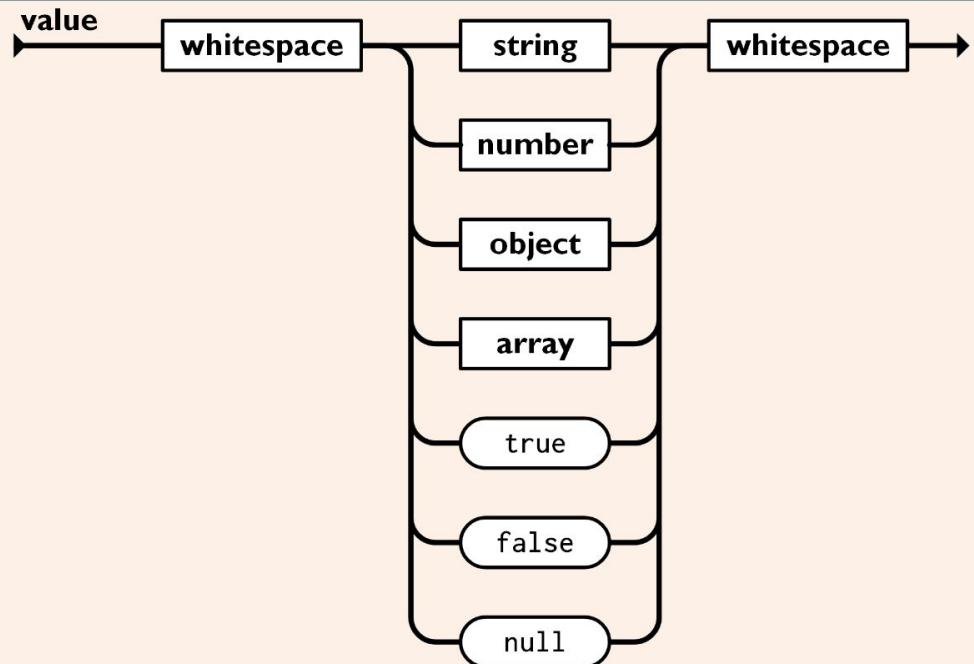
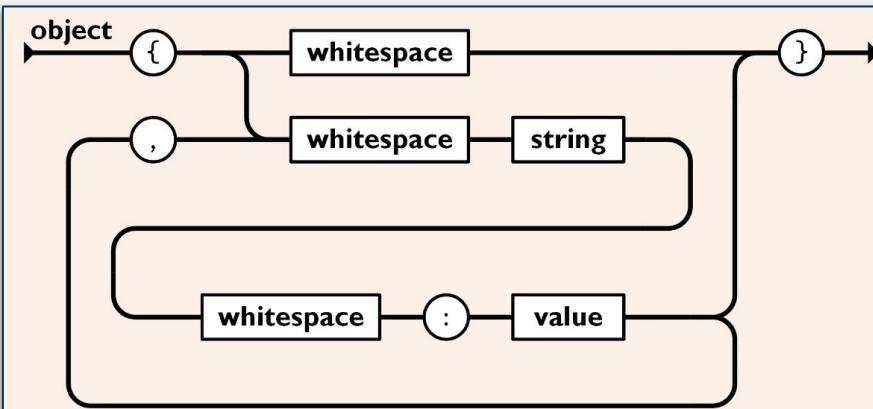
They are designed to be *simple, flexible, and scalable*.

```
{  
  "orders": [  
    {  
      "orderno": "748745375",  
      "date": "June 30, 2088 1:54:23 AM",  
      "trackingno": "TN0039291",  
      "custid": "11045",  
      "customer": {  
        "custid": "11045",  
        "fname": "Sue",  
        "lname": "Hatfield",  
        "address": "1409 Silver Street",  
        "city": "Ashland",  
        "state": "NE",  
        "zip": "68003"  
      }  
    }  
  ]  
}
```

What is JSON?

- **JSON (JavaScript Object Notation)**
 - a lightweight data-interchange format
 - It is easy for humans to read and write.
 - It is easy for machines to parse and generate.
- **JSON is built on two structures:**
 - A **collection of name/value pairs**. In various languages, this is operationalized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
 - An **ordered list of values**. In most languages, this is operationalized as an array, vector, list, or sequence.
- These are two *universal data structures* supported by virtually all modern programming languages
 - Thus, JSON makes a great data interchange format.

JSON Syntax



Binary JSON? BSON

- **BSON → Binary JSON**

- binary-encoded serialization of a JSON-like document structure
- supports extended types not part of basic JSON (e.g. Date, BinaryData, etc)
- **Lightweight** - keep space overhead to a minimum
- **Traversable** - designed to be easily traversed, which is vitally important to a document DB
- **Efficient** - encoding and decoding *must* be efficient
- Supported by many modern programming languages

```
{"hello": "world"} →  
 \x16\x00\x00\x00          // total document size  
 \x02                      // 0x02 = type String  
 hello\x00                  // field name  
 \x06\x00\x00\x00world\x00  // field value  
 \x00                      // 0x00 = type E00 ('end of object')
```

XML (eXtensible Markup Language)

- Precursor to JSON as data exchange format
- XML + CSS → web pages that separated content and formatting
- Structurally similar to HTML, but tag set is extensible

```
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
</CATALOG>
```

XML-Related Tools/Technologies

- **Xpath** - a syntax for retrieving specific elements from an XML doc
- **Xquery** - a query language for *interrogating* XML documents; the *SQL* of XML
- **DTD** - Document Type Definition - a language for describing the allowed structure of an XML document
- **XSLT** - eXtensible Stylesheet Language Transformation - tool to transform XML into other formats, including non-XML formats such as HTML.

Why Document Databases?

- Document databases address the *impedance mismatch* problem between object persistence in OO systems and how relational DBs structure data.
 - OO Programming → Inheritance and Composition of types.
 - How do we save a complex object to a relational database?
We basically have to deconstruct it.
- The structure of a document is *self-describing*.
- They are well-aligned with apps that use JSON/XML as a transport layer

MongoDB

MongoDB

- Started in 2007 after Doubleclick was acquired by Google, and 3 of its veterans realized the limitations of relational databases for serving > 400,000 ads per second
- MongoDB was short for *Humongous Database*
- MongoDB Atlas released in 2016 → documentdb as a service

MongoDB Structure

Database

Collection A

Document 1

Document 2

Document 3

Collection B

Document 1

Document 2

Document 3

Collection C

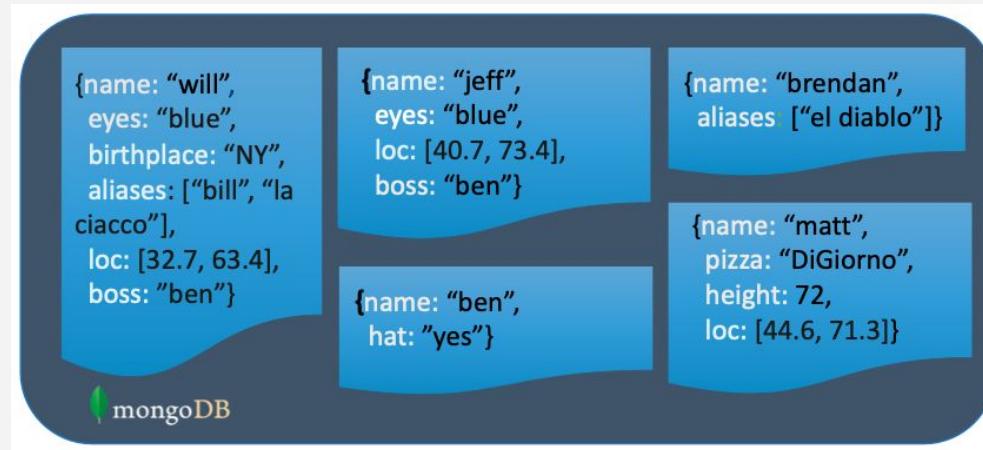
Document 1

Document 2

Document 3

MongoDB Documents

- No predefined schema for documents is needed
- Every document in a collection could have different data/schema



Relational vs Mongo/Document DB

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

MongoDB Features

- Rich Query Support - robust support for all CRUD ops
- Indexing - supports primary and secondary indices on document fields
- Replication - supports replica sets with automatic failover
- Load balancing built in

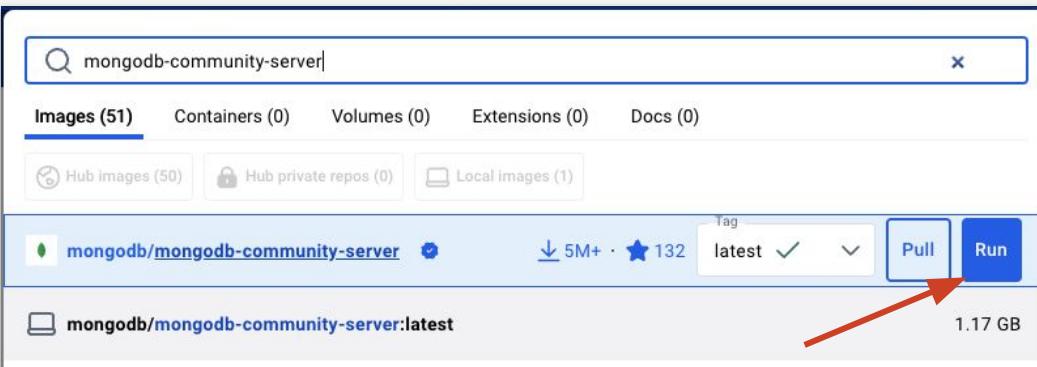
MongoDB Versions

- MongoDB Atlas
 - Fully managed MongoDB service in the cloud (DBaaS)
- MongoDB Enterprise
 - Subscription-based, self-managed version of MongoDB
- MongoDB Community
 - source-available, free-to-use, self-managed

Interacting with MongoDB

- **mongosh** → MongoDB Shell
 - CLI tool for interacting with a MongoDB instance
- MongoDB Compass
 - free, open-source GUI to work with a MongoDB database
- DataGrip and other 3rd Party Tools
- Every major language has a library to interface with MongoDB
 - PyMongo (Python), Mongoose (JavaScript/node), ...

Mongodb Community Edition in Docker



Run a new container
mongodb/mongodb-community-server:latest

Optional settings

Container name: **4300-mongodb**

A random name is generated if you do not provide one.

Ports

Enter "0" to assign randomly generated host ports.

Host port: **27017** Container port: **:27017/tcp**

Volumes

Host path ... Container path +

Environment variables

Variable: MONGO_INITDB_ROOT_USERNAME	Value: mark	-
Variable: MONGO_INITDB_ROOT_PASSWORD	Value: abc123	+

Cancel Run

- Create a container
- Map host:container port 27017
- Give initial username and password for superuser

MongoDB Compass

- GUI Tool for interacting with MongoDB instance
- Download and install from > [here](#) <.

New Connection

Manage your connection settings

URI **Edit Connection String**

Name **Color**

Favorite this connection
Favoriting a connection will pin it to the top of your list of connections

Advanced Connection Options

Advanced Connection Options

General **Authentication** **TLS/SSL** **Proxy/SSH** **In-Use Encryption** **Advanced**

Authentication Method

Username/Password **OIDC** **X.509** **Kerberos** **LDAP** **AWS IAM**

Username Optional

Password Optional

Authentication Database Optional

Authentication Mechanism

Default **SCRAM-SHA-1** **SCRAM-SHA-256**

Save **Connect** **Save & Connect**

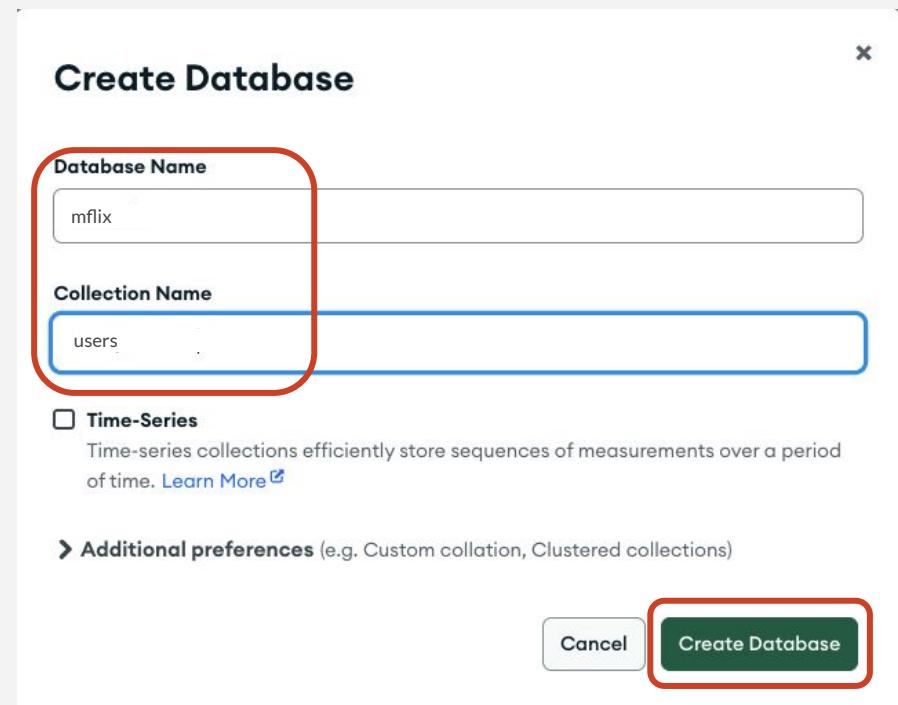
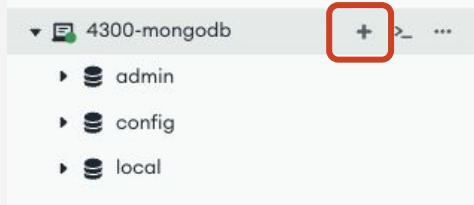


Load MFlix Sample Data Set

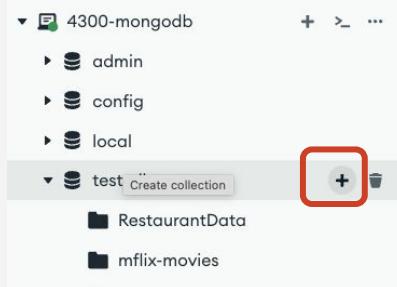
- In Compass, create a new Database named **mflix**
- Download [mflix sample dataset](#) and unzip it
- Import JSON files for users, theaters, movies, and comments into new collections in the mflix database

Creating a Database and Collection

To Create a new DB:



To Create a new Collection:



mongosh - Mongo Shell

- `find(...)` is like `SELECT`

```
collection.find({ ____ }, { ____ })
```

filters

projections

mongosh - find()

- SELECT * FROM users;

```
use mflix  
  
db.users.find()
```

- SELECT *
FROM users
WHERE name = "Davos Seaworth";

mongosh - find()

db.users.find({ "name": "Davos Seaworth" })

filter

```
< {  
    _id: ObjectId('59b99dbecfa9a34dcd7885c9'),  
    name: 'Davos Seaworth',  
    email: 'liam_cunningham@gameofthron.es',  
    password: '$2b$12$jbgNowG97LHNIm4axwXDz.tkFITsmw/aylIY/lZDaJRgnHZjB029e'  
}
```

mongosh - find()

- SELECT *
FROM movies
WHERE rated in ("PG", "PG-13")

```
db.movies.find({rated: {$in: [ "PG", "PG-13" ]}})
```

mongosh - find()

- Return movies which were released in Mexico and have an IMDB rating of at least 7

```
db.movies.find( {  
    "countries": "Mexico",  
    "imdb.rating": { $gte: 7 }  
} )
```

mongosh - find()

- Return movies from the **movies** collection which were released in 2010 and either won at least 5 awards or have a genre of Drama

```
db.movies.find( {  
    "year": 2010,  
    $or: [  
        { "awards.wins": { $gte: 5 } },  
        { "genres": "Drama" }  
    ]  
})
```

Comparison Operators

Name	Description
\$eq	Matches values that are equal to a specified value.
\$gt	Matches values that are greater than a specified value.
\$gte	Matches values that are greater than or equal to a specified value.
\$in	Matches any of the values specified in an array.
\$lt	Matches values that are less than a specified value.
\$lte	Matches values that are less than or equal to a specified value.
\$ne	Matches all values that are not equal to a specified value.
\$nin	Matches none of the values specified in an array.

mongosh - countDocuments()

- How many movies from the **movies** collection were released in 2010 and either won at least 5 awards or have a genre of Drama

```
db.movies.countDocuments( {  
    "year": 2010,  
    $or: [  
        { "awards.wins": { $gte: 5 } },  
        { "genres": "Drama" }  
    ]  
})
```

mongosh - project

- Return the names of all movies from the **movies** collection that were released in 2010 and either won at least 5 awards or have a genre of Drama

```
db.movies.countDocuments( {  
    "year": 2010,  
    $or: [  
        { "awards.wins": { $gte: 5 } },  
        { "genres": "Drama" }  
    ]  
}, {"name": 1, "_id": 0} )
```



1 = return; 0 = don't return

PyMongo

PyMongo

- PyMongo is a Python library for interfacing with MongoDB instances

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

Getting a Database and Collection

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

db = client['ds4300']
collection = db['myCollection']
```

Inserting a Single Document

```
db = client['ds4300']
collection = db['myCollection']

post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}

post_id = collection.insert_one(post).inserted_id
print(post_id)
```

Count Documents in Collection

- SELECT count(*) FROM collection

```
demodb.collection.count_documents({})
```

??
• •

DS 4300

MongoDB + PyMongo

Mark Fontenot, PhD
Northeastern University

PyMongo

- PyMongo is a Python library for interfacing with MongoDB instances

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

Getting a Database and Collection

```
from pymongo import MongoClient

client = MongoClient(
    'mongodb://user_name:pw@localhost:27017'
)

db = client['ds4300'] # or client.ds4300
collection = db['myCollection'] #or db.myCollection
```

Inserting a Single Document

```
db = client['ds4300']
collection = db['myCollection']

post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}

post_id = collection.insert_one(post).inserted_id
print(post_id)
```

Find all Movies from 2000

```
from bson.json_util import dumps  
  
# Find all movies released in 2000  
movies_2000 = db.movies.find({"year": 2000})  
  
# Print results  
print(dumps(movies_2000, indent = 2))
```

Jupyter Time

- Activate your DS4300 conda or venv python environment
- Install pymongo with **pip install pymongo**
- Install **Jupyter Lab** in you python environment
 - **pip install jupyterlab**
- Download and unzip > [this](#) < zip file - contains 2 Jupyter Notebooks
- In terminal, navigate to the folder where you unzipped the files, and run **jupyter lab**

??
• •

8.2. The AVL Tree

The AVL tree (named for its inventors Adelson-Velskii and Landis) should be viewed as a BST with the following additional property: For every node, the heights of its left and right subtrees differ by at most 1. As long as the tree maintains this property, if the tree contains n nodes, then it has a depth of at most $O(\log n)$. As a result, search for any node will cost $O(\log n)$, and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $O(\log n)$, even in the worst case.

The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property. Of course, to be practical, we must be able to implement the revised update routines in $\Theta(\log n)$ time.

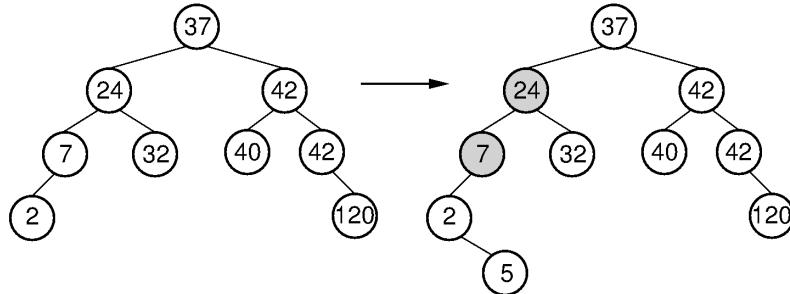


Figure 8.2.1: Example of an insert operation that violates the AVL tree balance property. Prior to the insert operation, all nodes of the tree are balanced (i.e., the depths of the left and right subtrees for every node differ by at most one). After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

Consider what happens when we insert a node with key value 5, as shown in Figure 8.2.1. The tree on the left meets the AVL tree balance requirements. After the insertion, two nodes no longer meet the requirements. Because the original tree met the balance requirement, nodes in the new tree can only be unbalanced by a difference of at most 2 in the subtrees. For the bottommost unbalanced node, call it S, there are 4 cases:

1. The extra node is in the left child of the left child of S.
2. The extra node is in the right child of the left child of S.
3. The extra node is in the left child of the right child of S.
4. The extra node is in the right child of the right child of S.

Cases 1 and 4 are symmetrical, as are cases 2 and 3. Note also that the unbalanced nodes must be on the path from the root to the newly inserted node.

Our problem now is how to balance the tree in $O(\log n)$ time. It turns out that we can do this using a series of local operations known as ***rotations***. Cases 1 and 4 can be fixed using a ***single rotation***, as shown in Figure 8.2.2. Cases 2 and 3 can be fixed using a ***double rotation***, as shown in Figure 8.2.3.

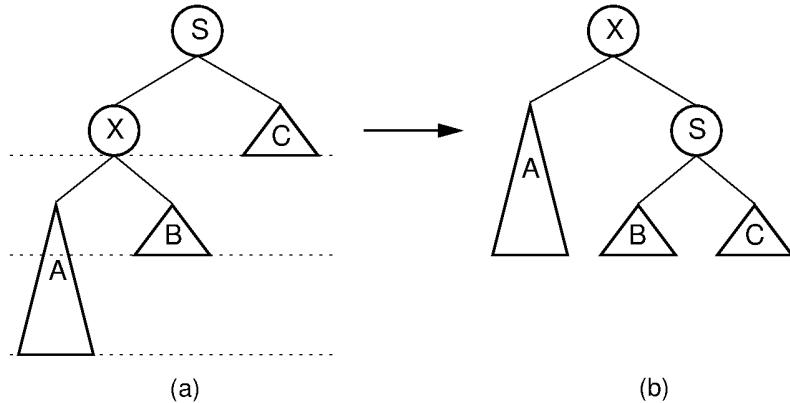


Figure 8.2.2: A single rotation in an AVL tree. This operation occurs when the excess node (in subtree A) is in the left child of the left child of the unbalanced node labeled S. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the right child of the right child of the unbalanced node is handled in the same way.

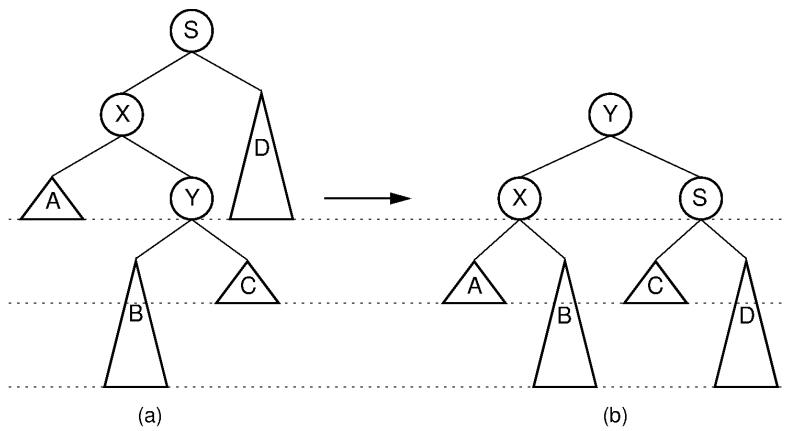


Figure 8.2.3: A double rotation in an AVL tree. This operation occurs when the excess node (in subtree B) is in the right child of the left child of the unbalanced node labeled S. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the left child of the right child of S is handled in the same way.

The AVL tree insert algorithm begins with a normal BST insert. Then as the recursion unwinds up the tree, we perform the appropriate rotation on any node that is found to be unbalanced. Deletion is similar; however, consideration for unbalanced nodes must begin at the level of the *deletemin* operation.

Example 8.2.1

In Figure 8.2.1 (b), the bottom-most unbalanced node has value 7. The excess node (with value 5) is in the right subtree of the left child of 7, so we have an example of Case 2. This requires a double rotation to fix. After the rotation, 5 becomes the left child of 24, 2 becomes the left child of 5, and 7 becomes the right child of 5.



12.6. B-Trees

12.6.1. B-Trees

This module presents the B-tree. B-trees are usually attributed to R. Bayer and E. McCreight who described the B-tree in a 1972 paper. By 1979, B-trees had replaced virtually all large-file access methods other than hashing. B-trees, or some variant of B-trees, are the standard file organization for applications requiring insertion, deletion, and key range searches. They are used to implement most modern file systems. B-trees address effectively all of the major problems encountered when implementing disk-based search trees:

1. The B-tree is shallow, in part because the tree is always height balanced (all leaf nodes are at the same level), and in part because the branching factor is quite high. So only a small number of disk blocks are accessed to reach a given record.
2. Update and search operations affect only those disk blocks on the path from the root to the leaf node containing the query record. The fewer the number of disk blocks affected during an operation, the less disk I/O is required.
3. B-trees keep related records (that is, records with similar key values) on the same disk block, which helps to minimize disk I/O on range searches.
4. B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

A B-tree of order m is defined to have the following shape properties:

The root is either a leaf or has at least two children.

Each internal node, except for the root, has between $\lceil m/2 \rceil$ and m children.

All leaves are at the same level in the tree, so the tree is always height balanced.

The B-tree is a generalization of the 2-3 tree. Put another way, a 2-3 tree is a B-tree of order three. Normally, the size of a node in the B-tree is chosen to fill a disk block. A B-tree node implementation typically allows 100 or more children. Thus, a B-tree node is equivalent to a disk block, and a “pointer” value stored in the tree is actually the number of the block containing the child node (usually interpreted as an offset from the beginning of the corresponding disk file). In a typical application, the B-tree’s access to the disk file will be managed using a **buffer pool** and a block-replacement scheme such as **LRU**.

Figure 12.6.1 shows a B-tree of order four. Each node contains up to three keys, and internal nodes have up to four children.

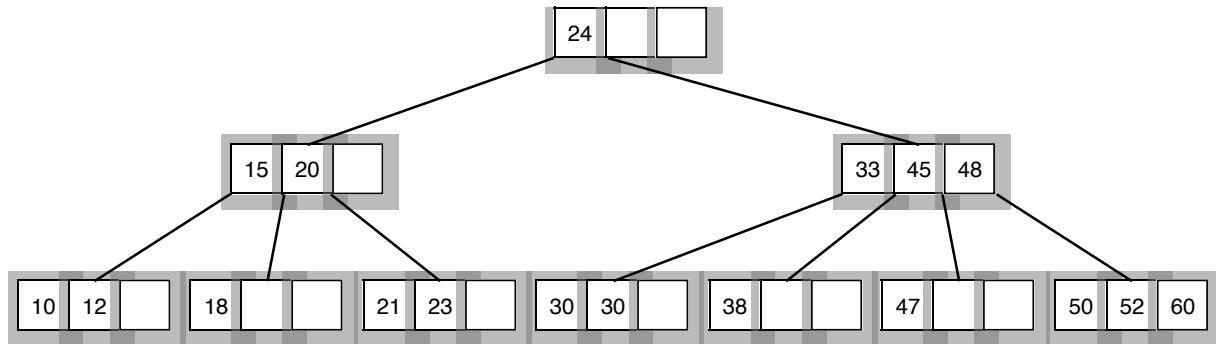


Figure 12.6.1: A B-tree of order four.

Search in a B-tree is a generalization of search in a 2-3 tree. It is an alternating two-step process, beginning with the root node of the B-tree.

1. Perform a binary search on the records in the current node. If a record with the search key is found, then return that record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.

2. Otherwise, follow the proper branch and repeat the process.

For example, consider a search for the record with key value 47 in the tree of Figure 12.6.1. The root node is examined and the second (right) branch taken. After examining the node at level 1, the third branch is taken to the next level to arrive at the leaf node containing a record with key value 47.

B-tree insertion is a generalization of 2-3 tree insertion. The first step is to find the leaf node that should contain the key to be inserted, space permitting. If there is room in this node, then insert the key. If there is not, then split the node into two and promote the middle key to the parent. If the parent becomes full, then it is split in turn, and its middle key promoted.

Note that this insertion process is guaranteed to keep all nodes at least half full. For example, when we attempt to insert into a full internal node of a B-tree of order four, there will now be five children that must be dealt with. The node is split into two nodes containing two keys each, thus retaining the B-tree property. The middle of the five children is promoted to its parent.

12.6.1.1. B+ Trees

The previous section mentioned that B-trees are universally used to implement large-scale disk-based systems. Actually, the B-tree as described in the previous section is almost never implemented. What is most commonly implemented is a variant of the B-tree, called the B⁺ tree. When greater efficiency is required, a more complicated variant known as the B* tree is used.

Consider again the **linear index**. When the collection of records will not change, a linear index provides an extremely efficient way to search. The problem is how to handle those pesky inserts and deletes. We could try to keep the core idea of storing a sorted array-based list, but make it more flexible by breaking the list into manageable chunks that are more easily updated. How might we do that? First, we need to decide how big the chunks should be. Since the data are on disk, it seems reasonable to store a chunk that is the size of a disk block, or a small multiple of the disk block size. If the next record to be inserted belongs to a chunk that hasn't filled its block then we can just insert it there. The fact that this might cause other records in that chunk to move a little bit in the array is not important, since this does not cause any extra disk accesses so long as we move data within that chunk. But what if the chunk fills up the entire block that contains it? We could just split it in half. What if we want to delete a record? We could just take the deleted record out of the chunk, but we might not want a lot of near-empty chunks. So we could put adjacent chunks together if they have only a small amount of data between them. Or we could shuffle data between adjacent chunks that together contain more data. The big problem would be how to find the desired chunk when processing a record with a given key. Perhaps some sort of tree-like structure could be used to locate the appropriate chunk. These ideas are exactly what motivate the B⁺ tree. The B⁺ tree is essentially a mechanism for managing a sorted array-based list, where the list is broken into chunks.

The most significant difference between the B⁺ tree and the BST or the standard B-tree is that the B⁺ tree stores records only at the leaf nodes. Internal nodes store key values, but these are used solely as placeholders to guide the search. This means that internal nodes are significantly different in structure from leaf nodes. Internal nodes store keys to guide the search, associating each key with a pointer to a child B⁺ tree node. Leaf nodes store actual records, or else keys and pointers to actual records in a separate disk file if the B⁺ tree is being used purely as an index. Depending on the size of a record as compared to the size of a key, a leaf node in a B⁺ tree of order m might have enough room to store more or less than m records. The requirement is simply that the leaf nodes store enough records to remain at least half full. The leaf nodes of a B⁺ tree are normally linked together to form a doubly linked list. Thus, the entire collection of records can be traversed in sorted order by visiting all the leaf nodes on the linked list. Here is a Java-like pseudocode representation for the B⁺ tree node interface. Leaf node and internal node subclasses would implement this interface.

```
/** Interface for B+ Tree nodes */
public interface BPNode<Key, E> {
    public boolean isLeaf();
    public int numrecs();
    public Key[] keys();
}
```

An important implementation detail to note is that while Figure 12.6.1 shows internal nodes containing three keys and four pointers, class BPNode is slightly different in that it stores key/pointer pairs. Figure 12.6.1 shows the B⁺ tree as it is traditionally drawn. To simplify implementation in practice, nodes really do associate a key with each pointer. Each internal node should be assumed to hold in the

leftmost position an additional key that is less than or equal to any possible key value in the node's leftmost subtree. B⁺ tree implementations typically store an additional dummy record in the leftmost leaf node whose key value is less than any legal key value.

Let's see in some detail how the simplest B⁺ tree works. This would be the "2 – 3⁺ tree", or a B⁺ tree of order 3.

1 / 28



Example 2-3+ Tree Visualization: Insert

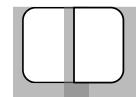


Figure 12.6.2: An example of building a 2 – 3⁺ tree

Next, let's see how to search.

1 / 10



Example 2-3+ Tree Visualization: Search

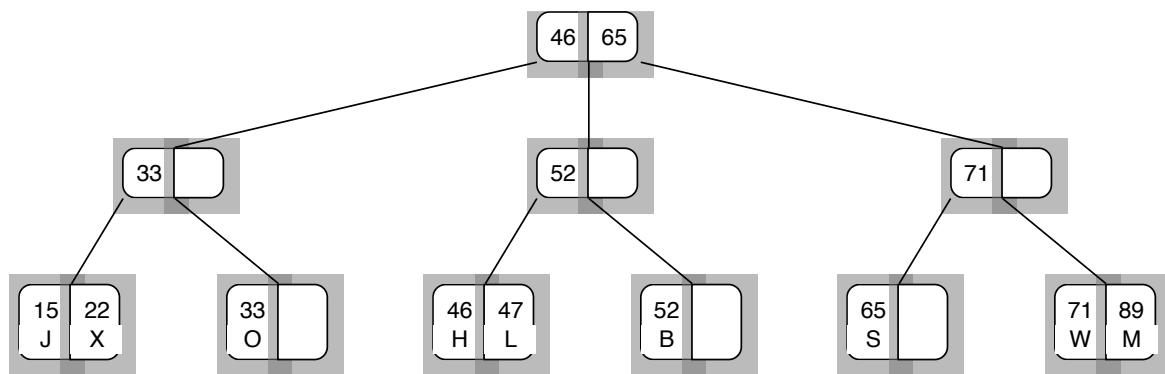


Figure 12.6.3: An example of searching a 2 – 3⁺ tree

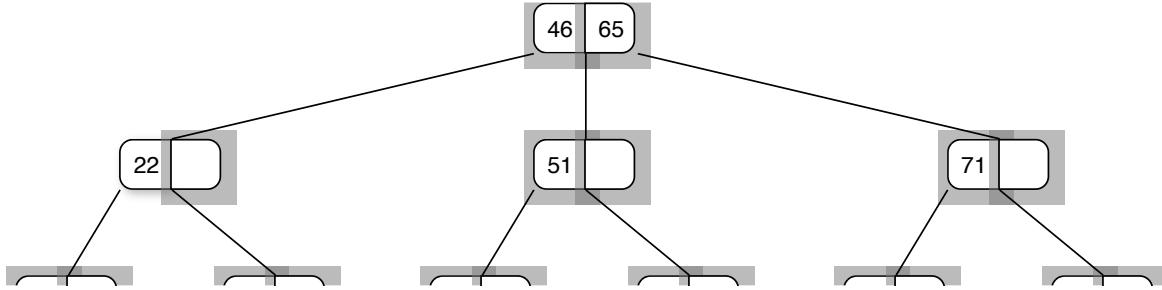
Finally, let's see an example of deleting from the 2 – 3⁺ tree

1 / 33





Example 2-3+ Tree Visualization: Delete

Figure 12.6.4: An example of deleting from a 2 – 3⁺ tree

Now, let's extend these ideas to a B⁺ tree of higher order.

B⁺ trees are exceptionally good for range queries. Once the first record in the range has been found, the rest of the records with keys in the range can be accessed by sequential processing of the remaining records in the first node, and then continuing down the linked list of leaf nodes as far as necessary. Figure illustrates the B⁺ tree.



Example B+ Tree Visualization: Search in a tree of degree 4

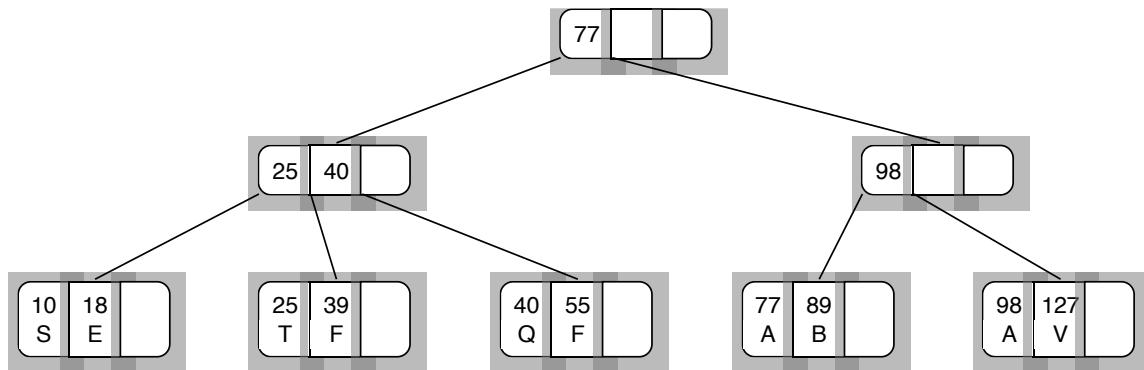


Figure 12.6.5: An example of search in a B+ tree of order four. Internal nodes must store between two and four children.

Search in a B⁺ tree is nearly identical to search in a regular B-tree, except that the search must always continue to the proper leaf node. Even if the search-key value is found in an internal node, this is only a placeholder and does not provide access to the actual record. Here is a pseudocode sketch of the B⁺ tree search algorithm.

```
private E findhelp(BPNode<Key, E> rt, Key k) {
    int currec = binaryle(rt.keys(), rt.numrecs(), k);
    if (rt.isLeaf()) {
        if (((BPLeaf<Key, E>)rt).keys())[currec] == k) {
```

```

        return ((BPLLeaf<Key,E>)rt).recs(currec);
    }
    else { return null; }
}
else{
    return findhelp(((BPInternal<Key,E>)rt).pointers(currec), k);
}
}
}

```

B^+ tree insertion is similar to B-tree insertion. First, the leaf L that should contain the record is found. If L is not full, then the new record is added, and no other B^+ tree nodes are affected. If L is already full, split it in two (dividing the records evenly among the two nodes) and promote a copy of the least-valued key in the newly formed right node. As with the 2-3 tree, promotion might cause the parent to split in turn, perhaps eventually leading to splitting the root and causing the B^+ tree to gain a new level. B^+ tree insertion keeps all leaf nodes at equal depth. Figure illustrates the insertion process through several examples.

1 / 42



Example B+ Tree Visualization: Insert into a tree of degree 4

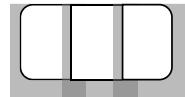


Figure 12.6.6: An example of building a B+ tree of order four.

Here is a Java-like pseudocode sketch of the B^+ tree insert algorithm.

```

private BPNode<Key,E> inserthelp(BPNode<Key,E> rt,
                                    Key k, E e) {
    BPNode<Key,E> retval;
    if (rt.isLeaf()) { // At leaf node: insert here
        return ((BPLLeaf<Key,E>)rt).add(k, e);
    }
    // Add to internal node
    int currec = binaryle(rt.keys(), rt.numrecs(), k);
    BPNode<Key,E> temp = inserthelp(
        ((BPInternal<Key,E>)root).pointers(currec), k, e);
    if (temp != ((BPInternal<Key,E>)rt).pointers(currec)) {
        return ((BPInternal<Key,E>)rt).
            add((BPInternal<Key,E>)temp);
    }
    else{
        return rt;
    }
}

```

Here is an exercise to see if you get the basic idea of B⁺ tree insertion.

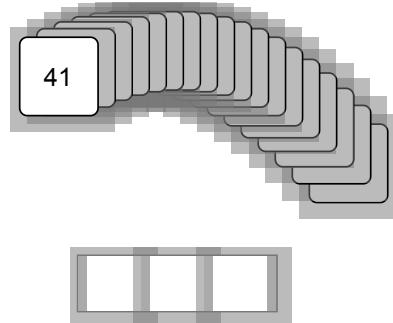
B⁺ Tree Insertion

Instructions:

In this exercise your job is to insert the values from the stack to the B⁺ tree.

Search for the leaf node where the topmost value of the stack should be inserted, and click on that node. The exercise will take care of the rest. Continue this procedure until you have inserted all the values in the stack.

[Undo](#) [Reset](#) [Model Answer](#) [Grade](#)

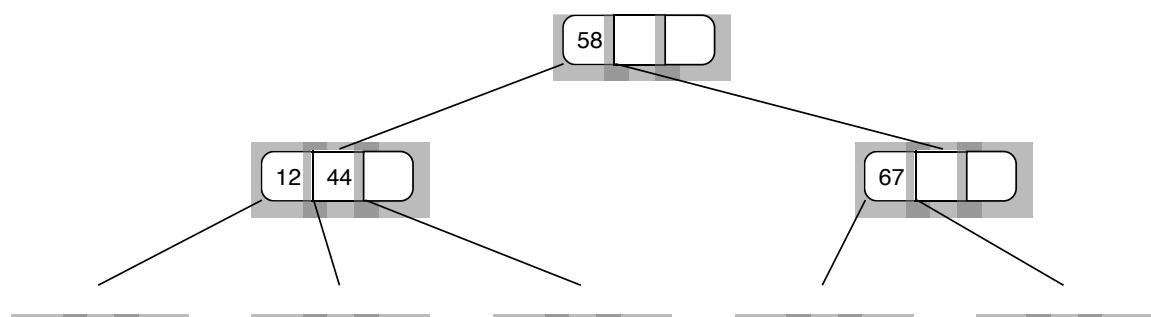


To delete record R from the B⁺ tree, first locate the leaf L that contains R. If L is more than half full, then we need only remove R, leaving L still at least half full. This is demonstrated by Figure .

1 / 23



Example B+ Tree Visualization: Delete from a tree of degree 4



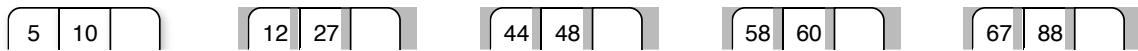


Figure 12.6.7: An example of deletion in a B+ tree of order four.

If deleting a record reduces the number of records in the node below the minimum threshold (called an **underflow**), then we must do something to keep the node sufficiently full. The first choice is to look at the node's adjacent siblings to determine if they have a spare record that can be used to fill the gap. If so, then enough records are transferred from the sibling so that both nodes have about the same number of records. This is done so as to delay as long as possible the next time when a delete causes this node to underflow again. This process might require that the parent node has its placeholder key value revised to reflect the true first key value in each node.

If neither sibling can lend a record to the under-full node (call it N), then N must give its records to a sibling and be removed from the tree. There is certainly room to do this, because the sibling is at most half full (remember that it had no records to contribute to the current node), and N has become less than half full because it is under-flowing. This merge process combines two subtrees of the parent, which might cause it to underflow in turn. If the last two children of the root merge together, then the tree loses a level.

Here is a Java-like pseudocode for the B⁺ tree delete algorithm.

```
/** Delete a record with the given key value, and
   return true if the root underflows */
private boolean removehelp(BPNode<Key, E> rt, Key k) {
    int currec = binaryle(rt.keys(), rt.numrecs(), k);
    if (rt.isLeaf()) {
        if (((BPLear<Key, E>)rt).keys()[currec] == k) {
            return ((BPLear<Key, E>)rt).delete(currec);
        }
        else { return false; }
    }
    else{ // Process internal node
        if (removehelp(((BPInternal<Key, E>)rt).pointers(currec),
                      k)) {
            // Child will merge if necessary
            return ((BPInternal<Key, E>)rt).underflow(currec);
        }
        else { return false; }
    }
}
```

The B⁺ tree requires that all nodes be at least half full (except for the root). Thus, the storage utilization must be at least 50%. This is satisfactory for many implementations, but note that keeping nodes fuller will result both in less space required (because there is less empty space in the disk file) and in more efficient processing (fewer blocks on average will be read into memory because the amount of information in each block is greater). Because B-trees have become so popular, many algorithm designers have tried to improve B-tree performance. One method for doing so is to use the B⁺ tree variant known as the B^{*} tree. The B^{*} tree is identical to the B⁺ tree, except for the rules used to split and merge nodes. Instead of splitting a node in half when it overflows, the B^{*} tree gives some records to its neighboring sibling, if possible. If the sibling is also full, then these two nodes split into three. Similarly, when a node underflows, it is combined with its two siblings, and the total reduced to two nodes. Thus, the nodes are always at least two thirds full. [1]

Finally, here is an example of building a B+ Tree of order five. You can compare this to the example above of building a tree of order four with the same records.



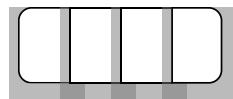


Figure 12.6.8: An example of building a B+ tree of degree 5

[Click here](#) for a visualization that will let you construct and interact with a B⁺ tree. This visualization was written by David Galles of the University of San Francisco as part of his [Data Structure Visualizations](#) package.

[1]

This concept can be extended further if higher space utilization is required. However, the update routines become much more complicated. I once worked on a project where we implemented 3-for-4 node split and merge routines. This gave better performance than the 2-for-3 node split and merge routines of the B^{*} tree. However, the splitting and merging routines were so complicated that even their author could no longer understand them once they were completed!

12.6.1.2. B-Tree Analysis

The asymptotic cost of search, insertion, and deletion of records from B-trees, B⁺ trees, and B^{*} trees is $\Theta(\log n)$ where n is the total number of records in the tree. However, the base of the log is the (average) branching factor of the tree. Typical database applications use extremely high branching factors, perhaps 100 or more. Thus, in practice the B-tree and its variants are extremely shallow.

As an illustration, consider a B⁺ tree of order 100 and leaf nodes that contain up to 100 records. A B-B⁺ tree with height one (that is, just a single leaf node) can have at most 100 records. A B⁺ tree with height two (a root internal node whose children are leaves) must have at least 100 records (2 leaves with 50 records each). It has at most 10,000 records (100 leaves with 100 records each). A B⁺ tree with height three must have at least 5000 records (two second-level nodes with 50 children containing 50 records each) and at most one million records (100 second-level nodes with 100 full children each). A B⁺ tree with height four must have at least 250,000 records and at most 100 million records. Thus, it would require an *extremely* large database to generate a B⁺ tree of more than height four.

The B⁺ tree split and insert rules guarantee that every node (except perhaps the root) is at least half full. So they are on average about 3/4 full. But the internal nodes are purely overhead, since the keys stored there are used only by the tree to direct search, rather than store actual data. Does this overhead amount to a significant use of space? No, because once again the high fan-out rate of the tree structure means that the vast majority of nodes are leaf nodes. A **K-ary tree** has approximately $1/K$ of its nodes as internal nodes. This means that while half of a full binary tree's nodes are internal nodes, in a B⁺ tree of order 100 probably only about 1/75 of its nodes are internal nodes. This means that the overhead associated with internal nodes is very low.

We can reduce the number of disk fetches required for the B-tree even more by using the following methods. First, the upper levels of the tree can be stored in main memory at all times. Because the tree branches so quickly, the top two levels (levels 0 and 1) require relatively little space. If the B-tree is only height four, then at most two disk fetches (internal nodes at level two and leaves at level three) are required to reach the pointer to any given record.

A buffer pool could be used to manage nodes of the B-tree. Several nodes of the tree would typically be in main memory at one time. The most straightforward approach is to use a standard method such as LRU to do node replacement. However, sometimes it might be desirable to “lock” certain nodes such as the root into the buffer pool. In general, if the buffer pool is even of modest size (say at least twice the depth of the tree), no special techniques for node replacement will be required because the upper-level nodes will naturally be accessed frequently.





AVL Tree Data Structure

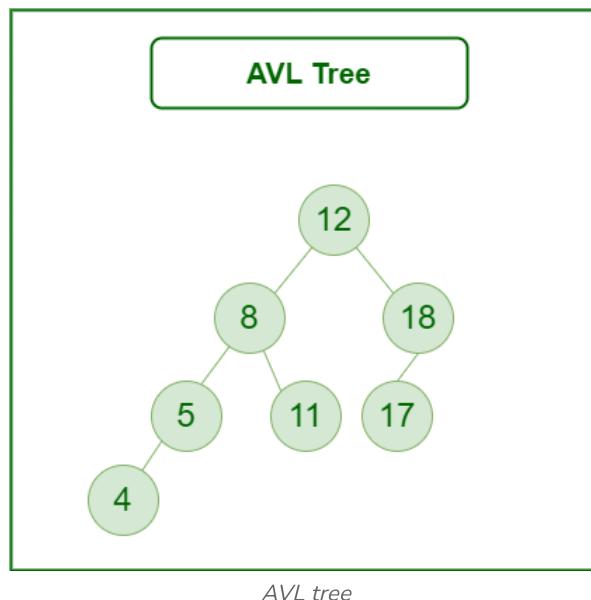
Last Updated : 24 Feb, 2025

An **AVL tree** defined as a self-balancing [Binary Search Tree \(BST\)](#) where the difference between heights of left and right subtrees for any node cannot be more than one.

- The absolute difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node. The balance factor for all nodes must be less than or equal to 1.
- Every AVL tree is also a Binary Search Tree (Left subtree values Smaller and Right subtree values grater for every node), but every BST is not AVL Tree. For example, the second diagram below is not an AVL Tree.
- The main advantage of an AVL Tree is, the time complexities of all operations (search, insert and delete, max, min, floor and ceiling) become $O(\log n)$. This happens because height of an AVL tree is bounded by $O(\log n)$. In case of a normal BST, the height can go up to $O(n)$.
- An AVL tree maintains its height by doing some extra work during insert and delete operations. It mainly uses rotations to maintain both BST properties and height balance.
- There exist other self-balancing BSTs also like [Red Black Tree](#). Red Black tree is more complex, but used more in practice as it is less restrictive in terms of left and right subtree height differences.

Example of an AVL Tree:

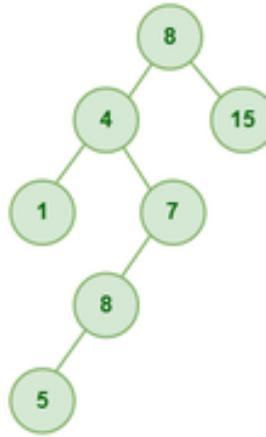
The balance factors for different nodes are : 12 :1, 8:1, 18:1, 5:1, 11:0, 17:0 and 4:0. Since all differences are less than or equal to 1, the tree is an AVL tree.



AVL tree

Example of a BST which is NOT AVL:

The Below Tree is NOT an AVL Tree as the balance factor for nodes 8, 4 and 7 is more than 1.



Not an AVL Tree

Operations on an AVL Tree:

- **Searching :** It is same as normal Binary Search Tree (BST) as an AVL Tree is always a BST. So we can use the same implementation as BST. The

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

- **Insertion** : It does rotations along with normal BST insertion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after insertion
- **Deletion** : It also does rotations along with normal BST deletion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after deletion.

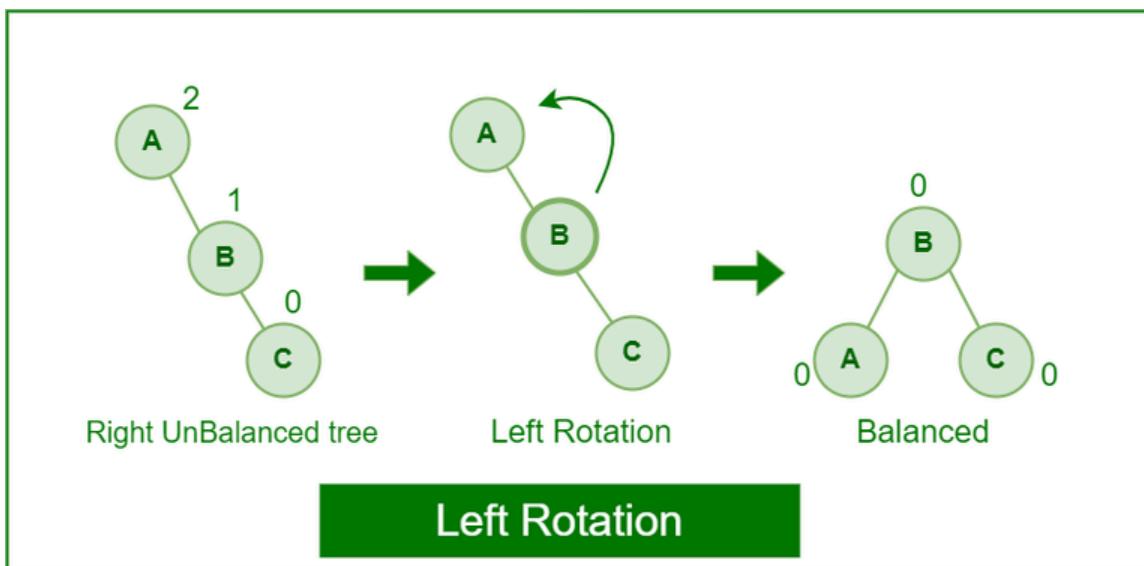
Please refer [Insertion in AVL Tree](#) and [Deletion in AVL Tree](#) for details.

Rotating the subtrees (Used in Insertion and Deletion)

An AVL tree may rotate in one of the following four ways to keep itself balanced while making sure that the BST properties are maintained.

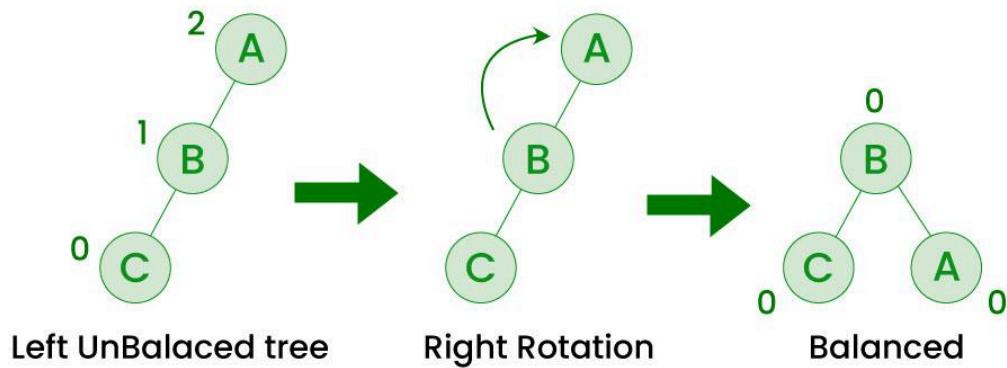
Left Rotation:

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



Right Rotation:

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



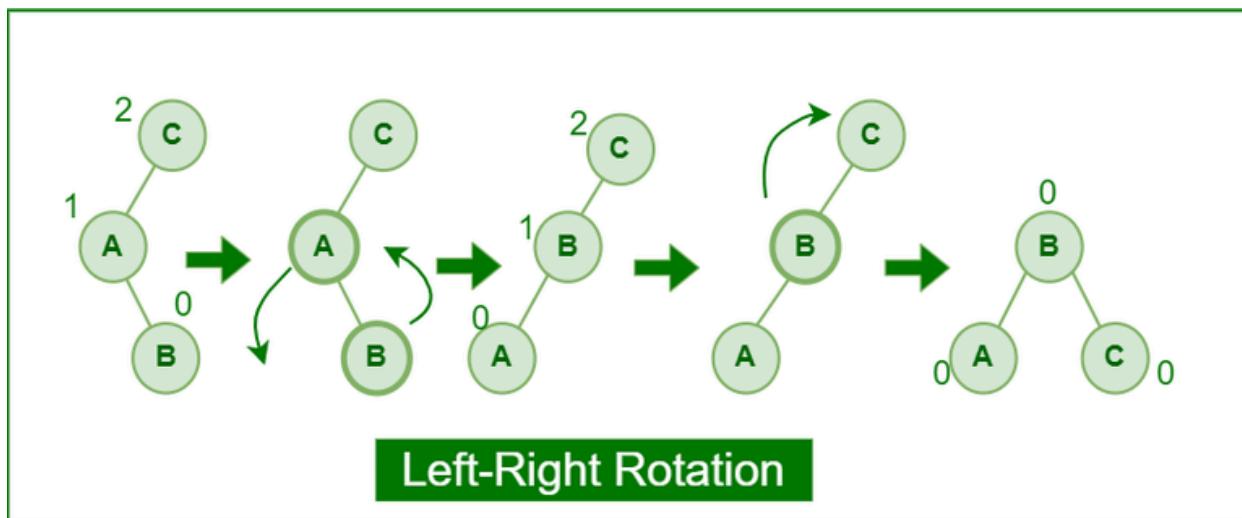
AVL Tree

∞

Right-Rotation in AVL Tree

Left-Right Rotation:

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.

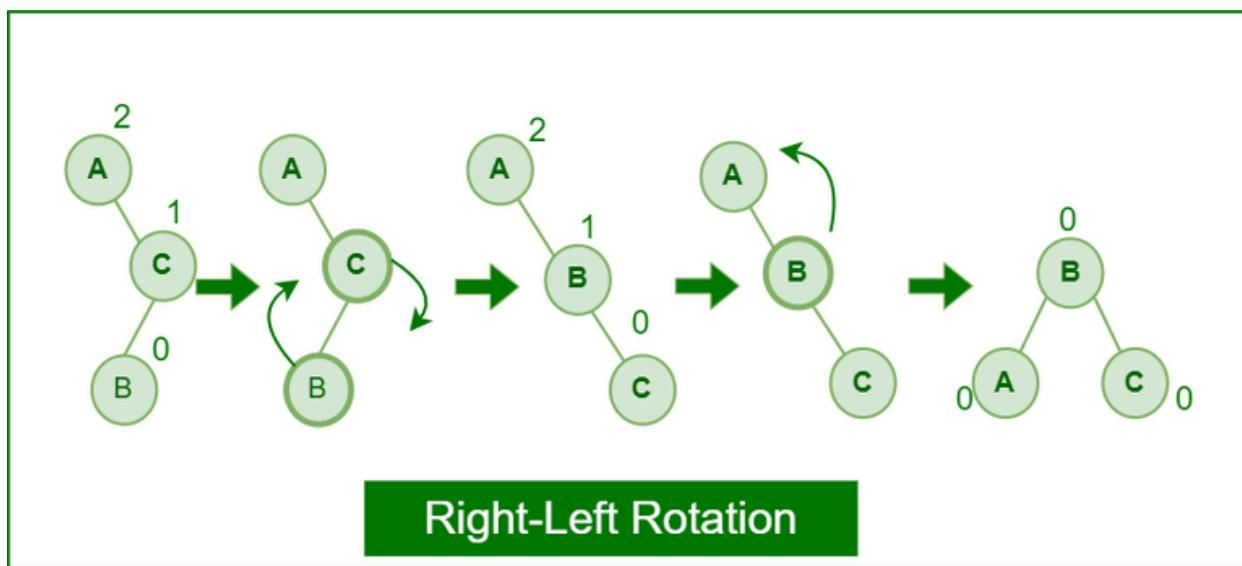


Left-Right Rotation

Left-Right Rotation in AVL tree

Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.

*Right-Left Rotation in AVL tree*

Advantages of AVL Tree:

1. AVL trees can self-balance themselves and therefore provides time complexity as $O(\log n)$ for search, insert and delete.
2. It is a BST only (with balancing), so items can be traversed in sorted order.
3. Since the balancing rules are strict compared to [Red Black Tree](#), AVL trees in general have relatively less height and hence the search is faster.
4. AVL tree is relatively less complex to understand and implement compared to Red Black Trees.

Disadvantages of AVL Tree:

1. It is difficult to implement compared to normal BST and easier compared to Red Black
2. Less used compared to Red-Black trees. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.

Applications of AVL Tree:

1. AVL Tree is used as a first example self balancing BST in teaching DSA as it

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

2. Applications, where insertions and deletions are less common but frequent data lookups along with other operations of BST like sorted traversal, floor, ceil, min and max.
3. Red Black tree is more commonly implemented in language libraries like [map in C++](#), [set in C++](#), [TreeMap in Java](#) and [TreeSet in Java](#).
4. AVL Trees can be used in a real time environment where predictable and consistent performance is required.

Related Articles:

- [Insertion in an AVL Tree](#)
- [Deletion in an AVL Tree](#)
- [Red Black Tree](#)

AVL Tree Data Structure

[Visit Course](#)

[Comment](#)

[More info](#)

[Advertise with us](#)

Next Article

[What is AVL Tree | AVL Tree meaning](#)

Similar Reads

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one. The absolute difference between the heights o...

4 min read

What is AVL Tree | AVL Tree meaning

An AVL is a self-balancing Binary Search Tree (BST) where the difference between the heights of left and right subtrees of any node cannot be more than one. KEY POINTS It is height balanced tree It is a binary search tree...

2 min read

Insertion in an AVL Tree

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. Example of AVL Tree: The above tree is AVL because the...

15+ min read

Insertion, Searching and Deletion in AVL trees containing a parent node pointer

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. The insertion and deletion in AVL trees have been discussed...

15+ min read

Deletion in an AVL Tree

We have discussed AVL insertion in the previous post. In this post, we will follow a similar approach for deletion. Steps to follow for deletion. To make sure that the given tree remains AVL after every deletion, we...

15+ min read

How is an AVL tree different from a B-tree?

AVL Trees: AVL tree is a self-balancing binary search tree in which each node maintains an extra factor which is called balance factor whose value is either -1, 0 or 1. B-Tree: A B-tree is a self-balancing tree data structure...

1 min read

Practice questions on Height balanced/AVL Tree

AVL tree is a binary search tree with an additional property that the difference between height of left sub-tree and right sub-tree of any node cannot be more than 1. Here are some key points about AVL trees: If there are n...

4 min read

AVL with duplicate keys

Please refer to the below post before reading about AVL tree handling of duplicates. How to handle duplicates in Binary Search Tree? This is to augment AVL tree node to store count together with regular fields like key, left...

15+ min read

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

In this article we will see that how to calculate number of elements which are greater than given value in AVL tree. Examples: Input : x = 5 Root of below AVL tree 9 /\ 1 10 / \ 0 5 11 // \ -1 2 6 Output : 4 Explanation:...

[15+ min read](#)

Difference between Binary Search Tree and AVL Tree

Binary Search Tree:A binary Search Tree is a node-based binary tree data structure that has the following properties: The left subtree of a node contains only nodes with keys lesser than the node's key.The right...

[2 min read](#)



Corporate & Communications Address:

A-143, 7th Floor, Sovereign Corporate
Tower, Sector- 136, Noida, Uttar Pradesh
(201305)

Registered Address:

K 061, Tower K, Gulshan Vivante
Apartment, Sector 137, Noida, Gautam
Buddh Nagar, Uttar Pradesh, 201305



[Advertise with us](#)

Company

- [About Us](#)
- [Legal](#)
- [Privacy Policy](#)
- [In Media](#)
- [Contact Us](#)
- [Advertise with us](#)
- [GFG Corporate Solution](#)
- [Placement Training Program](#)
- [GeeksforGeeks Community](#)

Languages

- [Python](#)
- [Java](#)
- [C++](#)
- [PHP](#)
- [GoLang](#)
- [SQL](#)
- [R Language](#)
- [Android Tutorial](#)
- [Tutorials Archive](#)

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

- [DSA for Beginners](#)
- [Basic DSA Problems](#)
- [DSA Roadmap](#)
- [Top 100 DSA Interview Problems](#)
- [DSA Roadmap by Sandeep Jain](#)
- [All Cheat Sheets](#)

- [Machine Learning](#)
- [ML Maths](#)
- [Data Visualisation](#)
- [Pandas](#)
- [NumPy](#)
- [NLP](#)
- [Deep Learning](#)

Web Technologies

- [HTML](#)
- [CSS](#)
- [JavaScript](#)
- [TypeScript](#)
- [ReactJS](#)
- [NextJS](#)
- [Bootstrap](#)
- [Web Design](#)

Python Tutorial

- [Python Programming Examples](#)
- [Python Projects](#)
- [Python Tkinter](#)
- [Web Scraping](#)
- [OpenCV Tutorial](#)
- [Python Interview Question](#)
- [Django](#)

Computer Science

- [Operating Systems](#)
- [Computer Network](#)
- [Database Management System](#)
- [Software Engineering](#)
- [Digital Logic Design](#)
- [Engineering Maths](#)
- [Software Development](#)
- [Software Testing](#)

DevOps

- [Git](#)
- [Linux](#)
- [AWS](#)
- [Docker](#)
- [Kubernetes](#)
- [Azure](#)
- [GCP](#)
- [DevOps Roadmap](#)

System Design

- [High Level Design](#)
- [Low Level Design](#)
- [UML Diagrams](#)
- [Interview Guide](#)
- [Design Patterns](#)
- [OOAD](#)
- [System Design Bootcamp](#)
- [Interview Questions](#)

Interview Preparation

- [Competitive Programming](#)
- [Top DS or Algo for CP](#)
- [Company-Wise Recruitment Process](#)
- [Company-Wise Preparation](#)
- [Aptitude Preparation](#)
- [Puzzles](#)

School Subjects

- [Mathematics](#)
- [Physics](#)
- [Chemistry](#)
- [Biology](#)
- [Social Science](#)
- [English Grammar](#)
- [Commerce](#)

GeeksforGeeks Videos

- [DSA](#)
- [Python](#)
- [Java](#)
- [C++](#)
- [Web Development](#)
- [Data Science](#)
- [CS Subjects](#)

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Home > Tutorials > Data Engineering

AVL Tree: Complete Guide With Python Implementation

An AVL tree is a self-balancing binary search tree where the height difference between the left and right subtrees of any node is at most one, ensuring efficient operations.

Jul 29, 2024 · 18 min read



François Aubry

Passionate teacher crafting engaging online courses for all learners.

TOPICS

Data Engineering

Binary search trees (BSTs) are a powerful [data structure](#) for organizing information, allowing for efficient searching and retrieval of values. However, standard BSTs can become unbalanced, leading to decreased performance in some scenarios.

AVL trees, named after their inventors, Adelson-Velsky and Landis, address this issue by maintaining balance regardless of the order in which data is inserted. This ensures consistently fast search operations, even with large datasets.

By the end of this article, you will understand how to implement an AVL tree in Python and utilize it for highly efficient data lookups.

This article assumes that you have some familiarity with binary search trees (BSTs), as AVL trees are an extension of this concept. If you need a recap, check out this quick introduction to [Binary Search Tree \(BST\)](#).

Before discussing AVL trees in more depth, let's first understand the problem they solve.

Become a Data Engineer

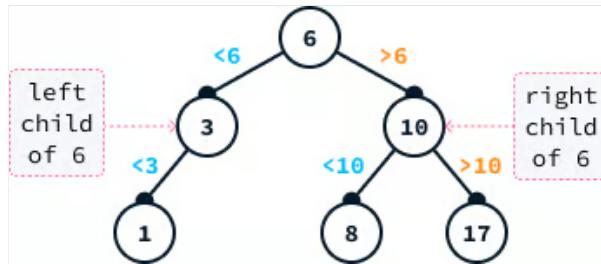
Build Python skills to become a professional data engineer.

[Get Started for Free](#)

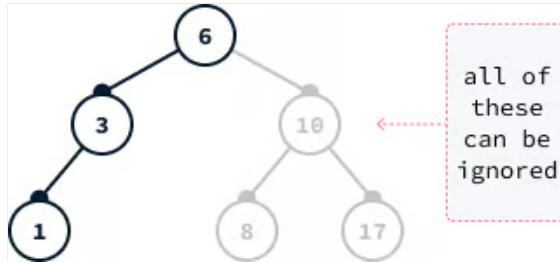
Imbalance in Binary Search Trees

Binary search trees (BSTs) are a type of binary tree [data structure](#) that organizes data using a specific ordering. Each node contains a value and has links to up to two other nodes: the *left* and *right* children. In a BST, the rule is that the value of the left child node must be less than its parent node's value, and the value of the right child node must be greater.

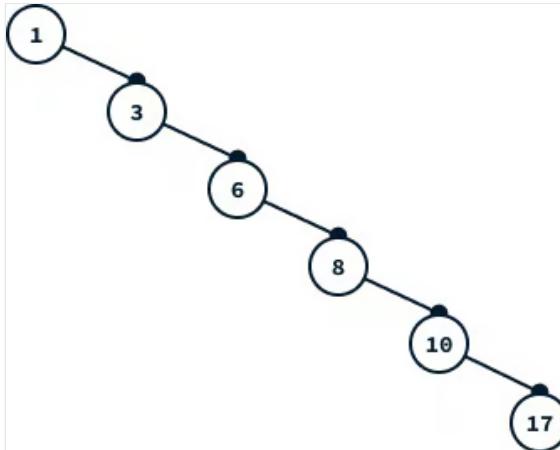




BSTs can be highly efficient for finding specific values because they allow us to eliminate large portions of the tree during the search process. For instance, if we're searching for the value one in the tree above, we can disregard all nodes to the right of six. This is because the order property guarantees that all those values are greater than six.



Ideally, each node should split the data in half, so that half of the values are eliminated at each step down the tree. This leads to extremely fast lookups. However, depending on the order of insertion, it's possible to end up with an unbalanced tree that does not effectively split the data. For instance, inserting values from smallest to largest will result in a linear tree, which performs no better than a list.

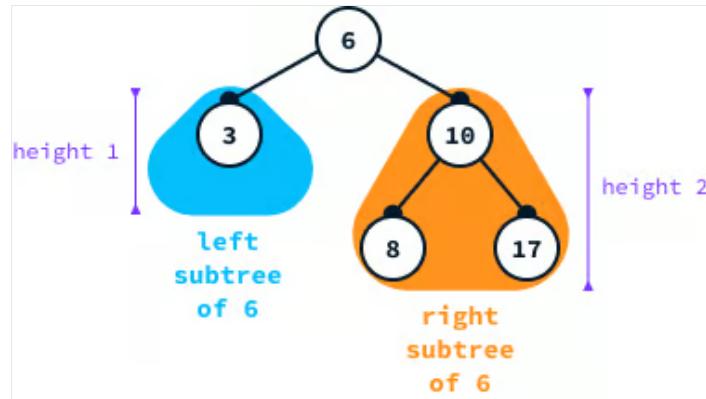


What Is an AVL Tree

An AVL tree is a binary search tree with the following added property:

For each node, the height of the left and right subtrees differ by at most one.

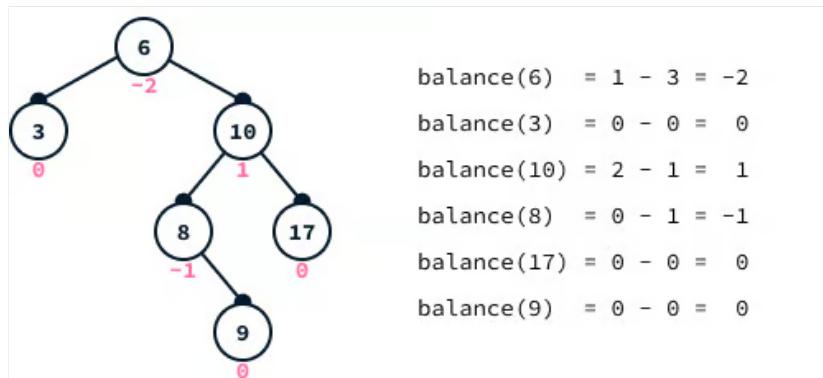
Breaking down this definition, the *left subtree* of a node includes all nodes to its left, while the *right subtree* comprises all nodes to its right. The height of a tree is defined as the length of the longest path from the root node (the topmost node) to any of its descendant leaves (nodes without children).



The balance factor of a node is calculated as the height difference between its left subtree and its right subtree:

$$\text{balance}(N) = \text{height}(\text{left subtree of } N) - \text{height}(\text{right subtree of } N)$$

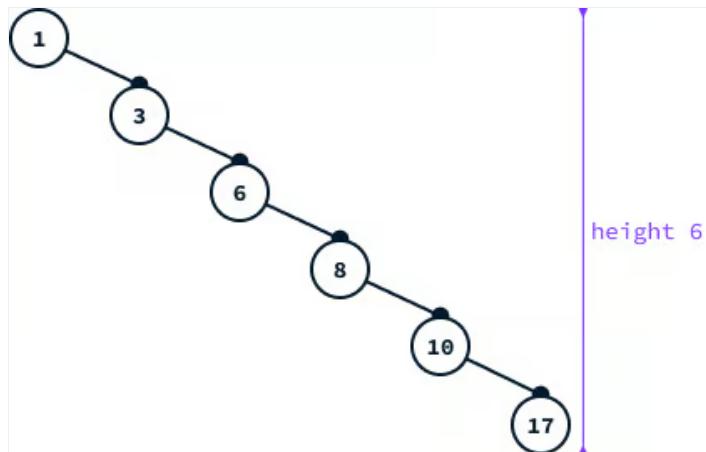
For example, $\text{balance}(6) = 1 - 3 = -2$.



In the diagram, node 6 exhibits a balance factor equal to -2, indicating that the tree does not conform to AVL tree criteria. For a tree to be classified as an AVL tree, the balance factor of each node must be -1, 0, or 1.

Why Use AVL Trees

The efficiency of queries in a binary search tree depends on the tree's height. In the worst-case scenario, the number of nodes that need to be examined is equal to the height of the tree. A key issue with BSTs is that the tree's height can match the number of nodes, meaning a query might necessitate inspecting every single node.



Let $M(h)$ denote the minimum number of nodes required to add to a binary search tree to achieve a height of h . For simple BSTs, we have observed that $M(h) = h$, meaning we can attain a height of h using just h nodes. This implies that the height of a BST can increase

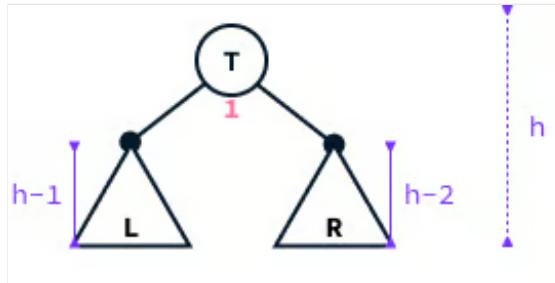
linearly with the number of nodes, leading to a query time that is proportional to the size of the dataset.

Proof that AVL trees have logarithmic height

Let's consider the minimum number of nodes, $M(h)$, required to create an AVL tree with a height of h . Since it is an AVL tree, it is important to note that the balance factor of every node can only be -1, 0, or 1.

However, given our assumption that the tree has the fewest nodes possible to achieve a height of h , the root cannot have a balance of 0. If it did, we could remove a node from the left or the right side to achieve a balance of either -1 or 1, which would still result in a valid AVL tree.

Let's assume the root's balance is 1 (the reasoning would be the same if the balance were -1). This implies that the tree is structured as follows:



On the other hand, both the left and right subtrees must also be AVL trees, each with the minimum number of nodes required for their respective heights (otherwise, it would be possible to remove additional nodes). The total number of nodes in the tree equals 1 (for the root) plus the number of nodes in the left subtree plus the number of nodes in the right subtree.

$$M(h) = 1 + (\text{nodes in } L) + (\text{nodes in } R) = 1 + M(h - 1) + M(h - 2)$$

As the height grows, we need to add more nodes to reach that height. Therefore:

$$M(h - 1) > M(h - 2)$$

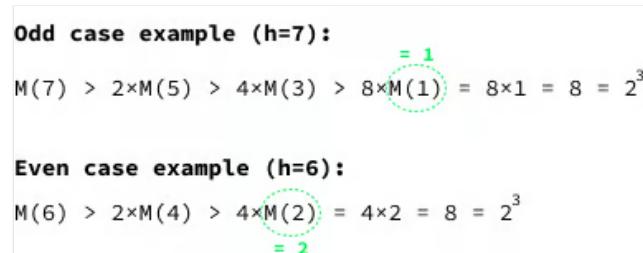
Combining the two, we can say that:

$$M(h) = 1 + M(h - 1) + M(h - 2) > 1 + 2 \times M(h - 2) > 2 \times M(h - 2)$$

We can apply this $h/2$ times until we reach either $M(1) = 1$ or $M(2) = 2$:

$$M(h) > 2 \times M(h - 2) > 2 \times 2 \times M(h - 4) > 2 \times 2 \times 2 \times M(h - 6) > \dots > 2^{(h/2)}$$

For clarity, the following image shows the specific examples when $h = 7$ and $h = 6$:



We conclude that the minimum number of nodes in an AVL tree of height h is at least $2^{(h/2)}$.

$$M(h) > 2^{(h/2)}$$

By applying the logarithm base two to both sides, we obtain:

$$\log_2(M(h)) > \log_2(2^{(h/2)}) = h/2$$

Consequently, by multiplying both sides by two, we deduce that the height is at most twice the base-2 logarithm of the number of nodes:

$$2 \times \log_2(M(h)) > h$$

We have demonstrated that:

The height of an AVL tree containing N nodes is at most $2 \times \log_2(N)$.

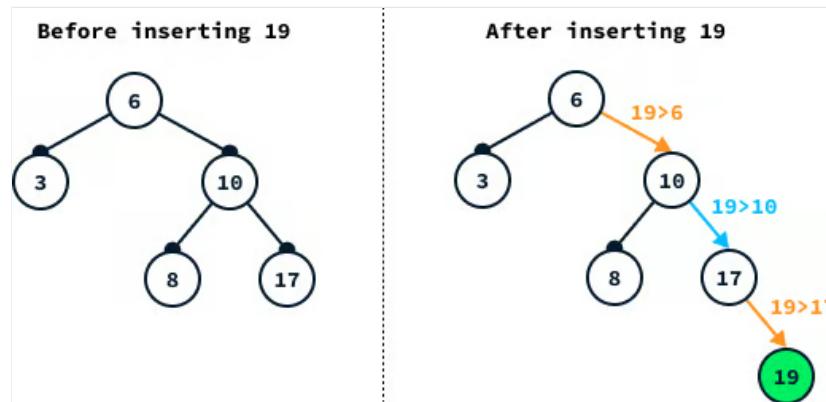
This indicates that queries within an AVL tree only necessitate examining a small segment of the dataset. For instance, given one billion entries, the logarithm equates to approximately 30, meaning that even with one billion data points, it's necessary to inspect only around 60 data points to find a specific entry. This represents a significant enhancement compared to BSTs, which, in the worst-case scenario, would require inspecting the entire one billion data points.

For a more in-depth understanding of algorithmic time complexity and the difference between linear time complexity and logarithmic complexity, check out this blog post on [Big-O Notation and Time Complexity](#).

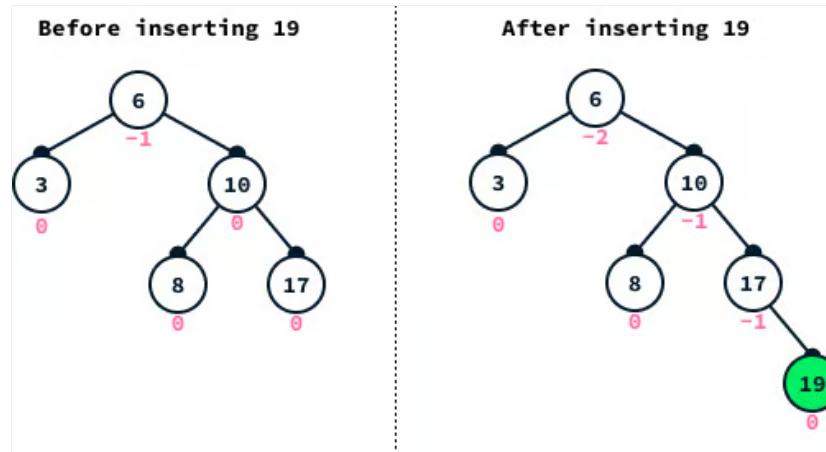
Maintaining the Balance With AVL Trees

AVL trees guarantee fast queries by enforcing that the balance of each node is either -1, 0, or 1. To ensure that the balance of each node is maintained, we need to rebalance the tree after inserting a new value.

Insertion in binary search trees works by following the path from the root down the tree. We go left whenever the value we want to insert is smaller and right otherwise.

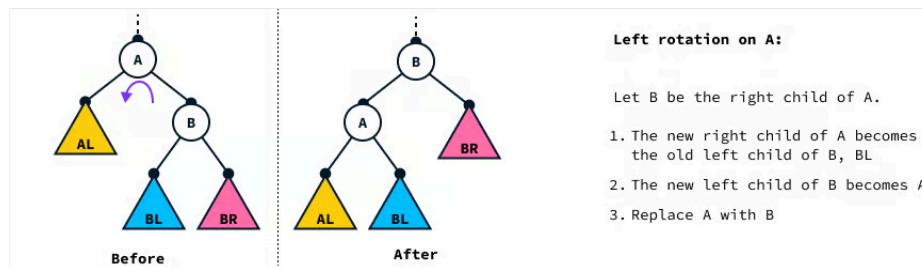


An insertion increases the height by at most one. So, if the balance property isn't respected after insertion, it means that there was either a node with balance -1, which now has balance -2, or a node with balance 1, which now has balance 2. The first case is what happens in the above example.



Single rotations

To restore balance, we rely on tree rotations. A *left rotation* on node A restructures the tree by rotating A to the left, as shown below:



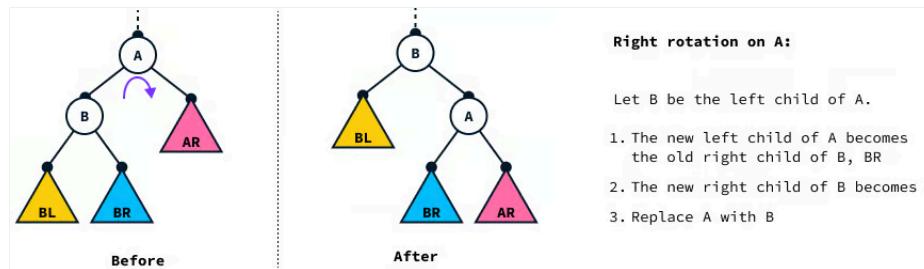
On the diagram:

- BL represents the left subtree of B
- BR is the right subtree
- AL is the left subtree of A

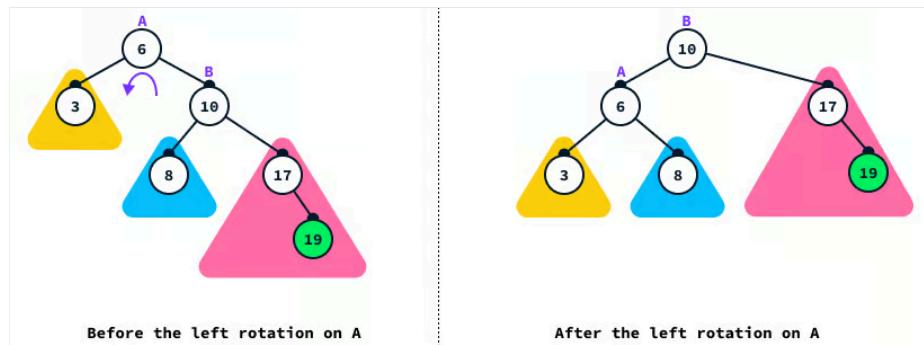
Note that after rotation, the node order is still valid:

1. Node A is smaller than B because B was its right child.
2. Nodes in BL are larger than A because they were on the right of A.
3. Nodes in AL are smaller than B because they are smaller than A.

A right rotation works in the same symmetric way by rotating A to the right.



Let's see a concrete example by fixing the imbalance of the tree after inserting 19 using a left rotation on node 6.

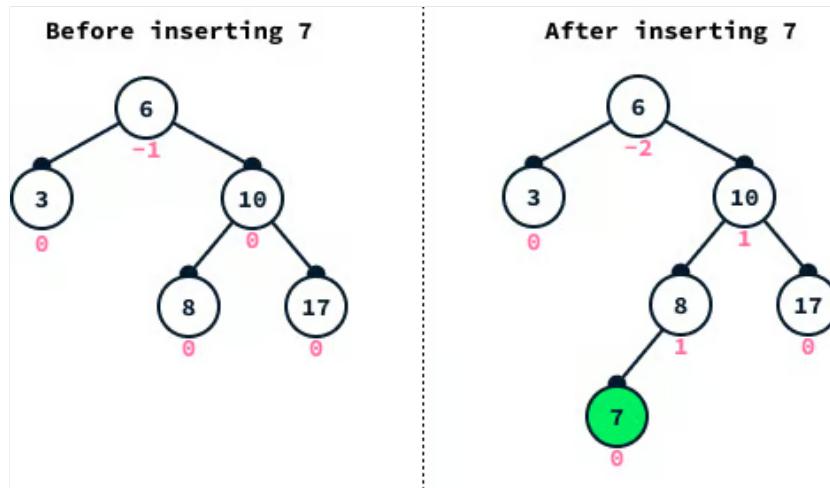


After insertion, we'll fix the imbalance by rotating a node with a balance of -2 or 2. In this case, the balance factor of node 6 was -2, meaning that the tree was leaning to the right, so we applied a left rotation (in the direction opposite of the imbalance). If, instead, the balance is equal to 2, then a right rotation is used instead.

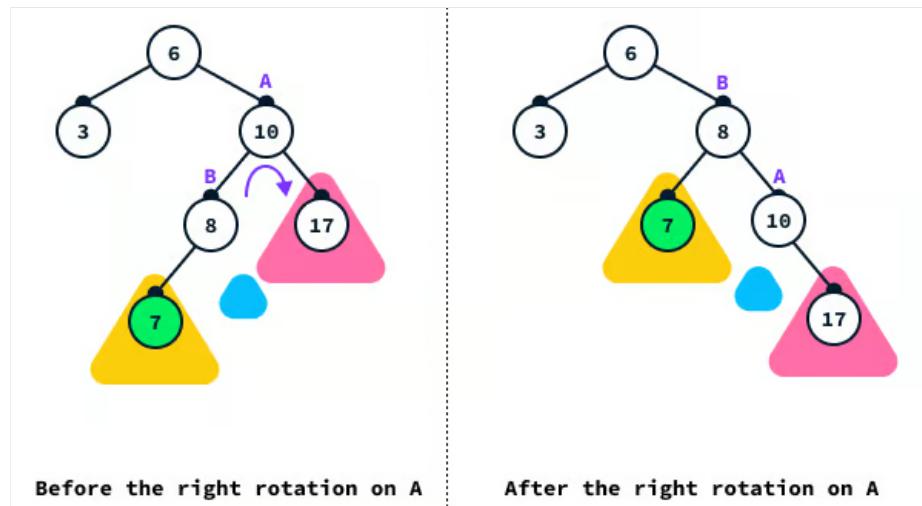
Double rotations

In the previous example, the tree was fully leaning to the right, so a single left rotation was enough to restore balance. However, in some cases, we have a zig-zag type of imbalance

where the tree leans to one side, but the subtree leans to the opposite side. To see this, let's go back to the original tree before we inserted 19 and insert 7 instead:

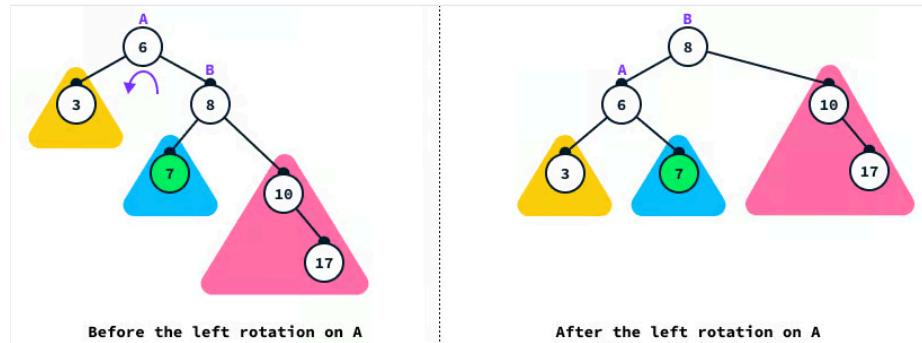


In this case, the tree is still leaning to the right, but the subtree rooted at 10 is leaning to the left. In this case, we first need to rotate node 10 to the right:



Note that node B doesn't have the right child. We still display it in blue in the diagram to make it easier to visualize.

After performing the right rotation, we fall back into the previous case where a left rotation on 6 will restore balance:



How to Implement an AVL Tree in Python

Let's start with the node implementation.

Node implementation

Each node of the tree has five attributes:

- The value it stores (`self.value`)
- The parent node (`self.parent`)
- The left child (`self.left`)
- The right child (`self.right`)
- The height of the subtree rooted at that node (`self.height`)

```
class Node:
    def __init__(self, value, parent = None):
        self.value = value
        self.parent = parent
        self.left = None
        self.right = None
        self.height = 1
```

 Explain code

POWERED BY  databricks

We employ the value `None` to represent missing nodes. The default height is set to 1 because a tree consisting of a single node has a height of 1.

To facilitate the tree implementation, we introduce several methods to the `Node` class.

```
# Inside the Node class
def left_height(self):
    # Get the height of the left subtree
    return 0 if self.left is None else self.left.height

def right_height(self):
    # Get the height of the right subtree
    return 0 if self.right is None else self.right.height
def balance_factor(self):
    # Get the balance factor
    return self.left_height() - self.right_height()
def update_height(self):
    # Update the height of this node
    self.height = 1 + max(self.left_height(), self.right_height())

def set_left(self, node):
    # Set the left child
    self.left = node
    if node is not None:
        node.parent = self
    self.update_height()

def set_right(self, node):
    # Set the right child
    self.right = node
    if node is not None:
        node.parent = self
    self.update_height()
def is_left_child(self):
    # Check whether this node is a left child
    return self.parent is not None and self.parent.left == self

def is_right_child(self):
    # Check whether this node is a right child
    return self.parent is not None and self.parent.right == self
```

 Explain code

POWERED BY  databricks

Note that we use the methods `.set_left()` and `.set_right()` to assign the left and right children, respectively. The rationale behind using these methods, rather than directly

modifying the `self.left` and `self.right` attributes, is that whenever a child is changed, it is necessary to update the parent of the new child as well as the node's height.

AVL tree implementation

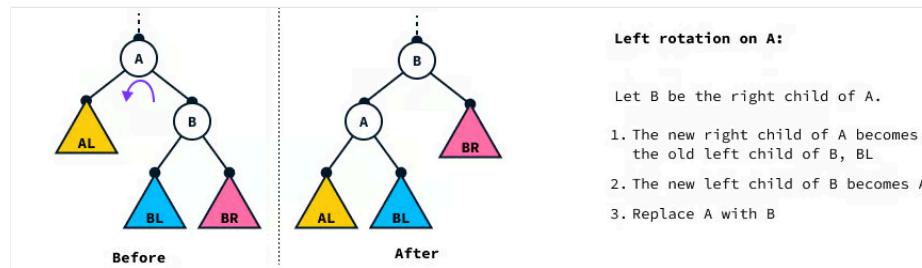
The AVL tree maintains one parameter, the root of the tree, which is the topmost node.

```
class AVLTree:
    def __init__(self):
        self.root = None
```

[Explain code](#)

POWERED BY  databl

To maintain the balance of the tree, we need to implement left and right rotations. Let's review how a left rotation is illustrated in the diagram:



Inside the AVLTree class

```
def rotate_left(self, a):
    b = a.right
    # 1. The new right child of A becomes the left child of B
    a.set_right(b.left)
    # 2. The new left child of B becomes A
    b.set_left(a)
    return b # 3. Return B to replace A with it
```

[Explain code](#)

POWERED BY  databl

Right rotations are implemented symmetrically:

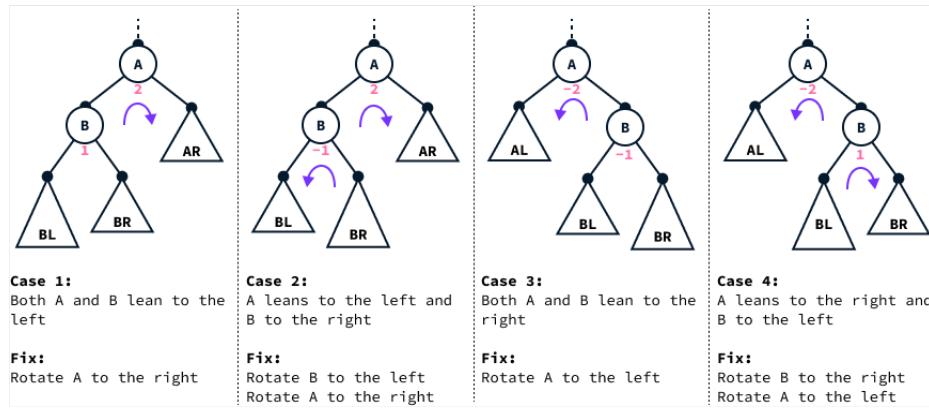
Inside the AVLTree class

```
def rotate_right(self, a):
    b = a.left
    a.set_left(b.right)
    b.set_right(a)
    return b
```

[Explain code](#)

POWERED BY  databl

Using rotations, we can rebalance the tree. A node requires rebalancing when its balance factor reaches either 2 (indicating the tree is leaning to the left) or -2 (indicating the tree is leaning to the right). Overall, there are four scenarios we need to consider:



We implement these four cases in the `.rebalance()` method.

```
# Inside the AVLTree class

def rebalance(self, node):
    if node is None:
        # Empty tree, no rebalancing needed
        return None
    balance = node.balance_factor()
    if abs(balance) <= 1:
        # The node is already balanced, no rebalancing needed
        return node
    if balance == 2:
        # Cases 1 and 2, the tree is leaning to the left
        if node.left.balance_factor() == -1:
            # Case 2, we first do a left rotation
            node.set_left(self.rotate_left(node.left)))
            return self.rotate_right(node))
        # Balance must be -2
        # Cases 3 and 4, the tree is leaning to the left
        if node.right.balance_factor() == 1:
            # Case 4, we first do a right rotation
            node.set_right(self.rotate_right(node.right)))
        return self.rotate_left(node))
```

[Explain code](#)

POWERED BY databricks

Note that in each case, the method returns the root of the subtree that has just been balanced. This new subtree root will be used later to update the children during the rebalancing process.

Adding a node to an AVL tree is similar to the process in a regular BST, with the addition of restoring balance after the node is inserted. To add a node in a BST, we begin at the root and travel down the tree. At each step, we compare the value to be added with the current node's value. If the value to add is smaller, we move left; otherwise, we move right. While descending, we keep track of the parent node so we can insert the new node as its child.

Once we reach an empty node, there are two cases:

1. The parent is `None` in which case the tree is empty so the new node is the new root of the tree.
2. We found the parent so we need to set the new node as either the left or right child, depending on the node's values.

```
# Inside the AVLTree class

def add(self, value):
    self.size += 1
    parent = None
    current = self.root
```

```

while current is not None:
    parent = current
    if value < current.value:
        # Value to insert is smaller than node value, go left
        current = current.left
    else:
        # Value to insert is larger than node value, go right
        current = current.right
    # We found the parent, create the new node
    new_node = Node(value, parent)
    # Case 1: The parent is None so the new node is the root
    if parent is None:
        self.root = new_node
    else:
        # Case 2: Set the new node as a child of the parent
        if value < parent.value:
            parent.left = new_node
        else:
            parent.right = new_node
    # After a new node is added, we need to restore balance
    self.restore_balance(new_node)

```

[Explain code](#)

POWERED BY databricks

The only difference between the `.add()` method of a BST and that of an AVL tree lies in the final step. This involves traversing back up the tree and rebalancing the nodes, starting from the newly added node to the root, and rebalancing each node by employing the `.rebalance()` method.

```

# Inside the AVLTree class
def restore_balance(self, node):
    current = node
    # Go up the tree and rebalance left and right children
    while current is not None:
        current.set_left(self.rebalance(current.left))
        current.set_right(self.rebalance(current.right))
        current.update_height()
        current = current.parent
    self.root = self.rebalance(self.root)
    self.root.parent = None

```

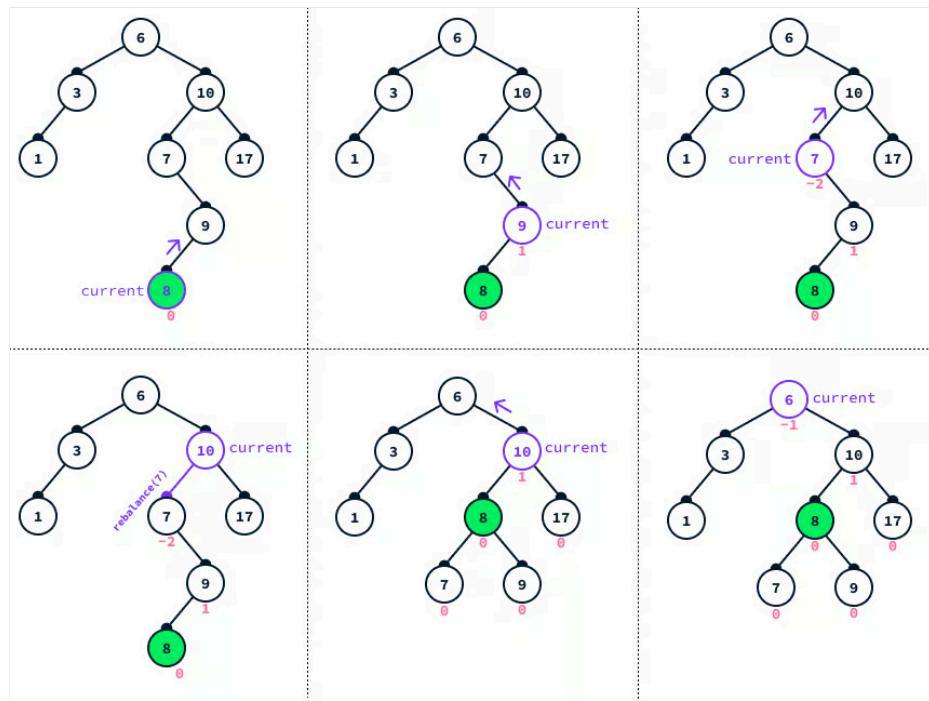
[Explain code](#)

POWERED BY databricks

Note that the `.rebalance()` method does nothing when the node is already balanced—it simply returns that same node. That's why, while doing up the tree, we can call it on both sides, even though only one of them can be unbalanced. The other call simply leaves the tree unchanged.

Recall that we implemented the `.rebalance()` method so that it returns the (potentially) new root of the subtree. The reason for this was so that we could update the left and right children of the current nodes as we go up restoring balance.

The following diagram shows the steps of `.restore_balance()` going up the tree.



In the given example, we first add node 8. Then, the process of restoring balance begins at this node and works its way up the tree, checking and rebalancing the left and right children of each encountered node. This continues until node 10 is reached. Until this point, none of the invocations of the `.rebalance()` method have any effect, as the nodes remain balanced.

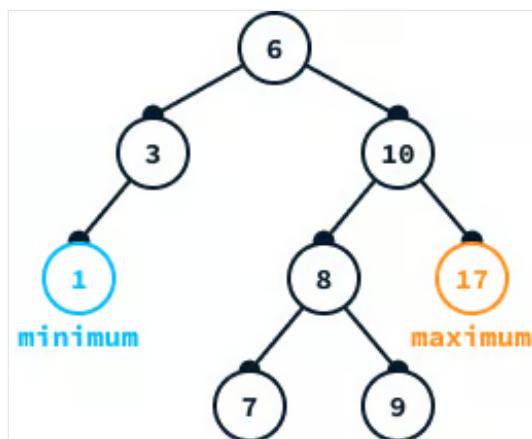
However, upon reaching node 10, it's observed that its left child, node 7, has a balance factor of -2, indicating a need for rebalancing. Consequently, `.rebalance(7)` is called, which results in the substitution of node 10's left child with the new root of the left subtree, node 8, effectively restoring balance to the tree.

Additional AVL Tree Operations

Besides adding and deleting elements while maintaining the tree's balance, AVL trees support several other essential operations.

Minimum and maximum

Due to the ordered nature of BST, the minimum value can be found at the leftmost node of the tree, whereas the maximum value is located at the rightmost node.



We have implemented two helper functions that identify the leftmost and rightmost nodes starting from a given node. These functions are useful for facilitating the implementation of node deletion.

```
# Inside the AVLTree class

def leftmost(self, starting_node):
    # Find the leftmost node from a given starting node
    previous = None
    current = starting_node
    while current is not None:
        previous = current
        current = current.left
    return previous

def minimum(self):
    # Return the minimum value in the tree
    if self.root is None:
        raise Exception("Empty tree")
    return self.leftmost(self.root).value
```

 Explain code

POWERED BY  datacamp

```
# Inside the AVLTree class

def rightmost(self, starting_node):
    # Find the rightmost node from a given starting node
    previous = None
    current = starting_node
    while current is not None:
        previous = current
        current = current.right
    return previous

def maximum(self):
    # Find the maximum value in the tree
    if self.root is None:
        raise Exception("Empty tree")
    return self.rightmost(self.root).value
```

 Explain code

POWERED BY  datacamp

Contains

To determine if a tree contains a specific value, we utilize the tree's order property to guide our search. Starting from the root, we traverse to the left if the value we seek is smaller than the current node and to the right if it's larger. If we reach the end of the tree without finding the desired value, it indicates that the value is not present in the tree.

To facilitate this process, we implement a helper method called `.locate_node()`. This method is particularly useful not only for searching but also for operations such as deleting values from the tree. Additionally, we use the `__contains__()` method, which allows us to utilize the `in` operator to simplify checking for the presence of a value within the tree.

```
# Inside the AVLTree class

def locate_node(self, value):
    # Returns the node containing a given value or None if no
    # such node exists

    current = self.root

    while current is not None:
        if value == current.value:
            return current
```

```

        if value < current.value:

            current = current.left

        else:

            current = current.right

    return None

def __contains__(self, value):

    node = self.locate_node(value)

    return node is not None

```

POWERED BY  databricks

Deletion

Deleting a node in an AVL tree can be particularly challenging when the node is situated in the middle of the tree. If the node is a leaf, meaning it has no children, we can easily delete it by setting its parent's left or right child pointer to `None` based on whether the node is a left or right child.

A special case arises when the node targeted for deletion is the root of the tree. In this scenario, we can remove the node by setting the root to `None`.

```

# Inside the AVLTree class

def delete_leaf(self, node):

    if node.parent is None:

        self.root = None

    elif node.is_left_child():

        node.parent.left = None

        node.parent = None

    else:

        node.parent.right = None

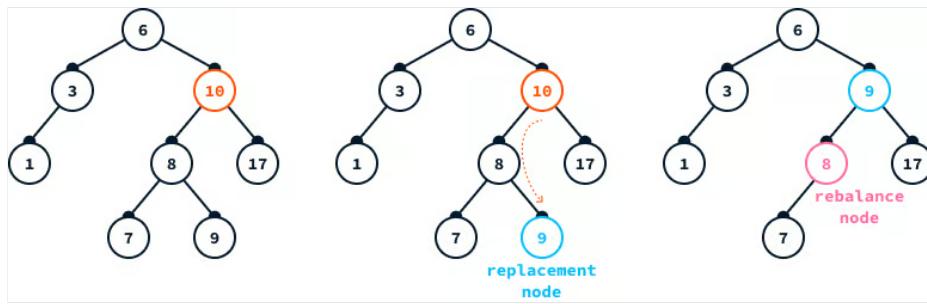
        node.parent = None

```

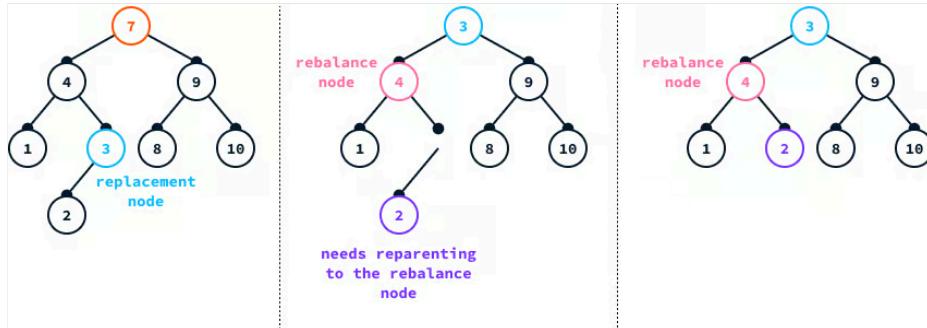
POWERED BY  databricks

To delete a value from an AVL tree, we must first locate the node containing that value using the `.locate_node()` method. Once the node is located, we can remove it with the `.delete_leaf()` method if it is a leaf node. If it's not a leaf node, direct removal would disrupt the tree's structure. To address this, we find a suitable replacement node. If the node to be deleted has a left child, we select the rightmost node in its left subtree. This approach ensures the tree maintains its ordered structure.

The diagram below exemplifies the removal of node 10 from a tree. Given it has a left child, we replace it with the rightmost node in the left subtree. Following the replacement, it's crucial to rebalance the tree, starting from the parent of the newly replaced node.



In this example, the replacement node is a leaf node. However, if the replacement node has children, it's necessary to reassign them to the parent of the replacement node. Since the replacement node is an extreme node (either the leftmost or the rightmost), it can have only one child. Therefore, this reassignment is always feasible.



```
# Inside the AVLTree class

def delete(self, value):

    # Delete a value from the tree

    node = self.locate_node(value)

    if node is None:

        raise Exception("Value not stored in tree")

    replacement = None

    rebalance_node = node.parent

    if node.left is not None:

        # There's a left child so we replace with rightmost node

        replacement = self.rightmost(node.left)

        # Check if reparenting is needed

        if replacement.is_left_child():

            replacement.parent.set_left(replacement.left)

        else:

            replacement.parent.set_right(replacement.left)

    elif node.right is not None:

        # There's a right child so we replace with the leftmost node

        replacement = self.leftmost(node.right)
```

```

# Check if reparenting is needed

if replacement.is_left_child():

    replacement.parent.set_left(replacement.right)

else:

    replacement.parent.set_right(replacement.right)

if replacement:

    # We found a replacement so replace the value

    node.value = replacement.value

    rebalance_node = replacement.parent

else:

    # No replacement so it means the node to delete is a leaf

    self.delete_leaf(node)

if rebalance_node is not None:

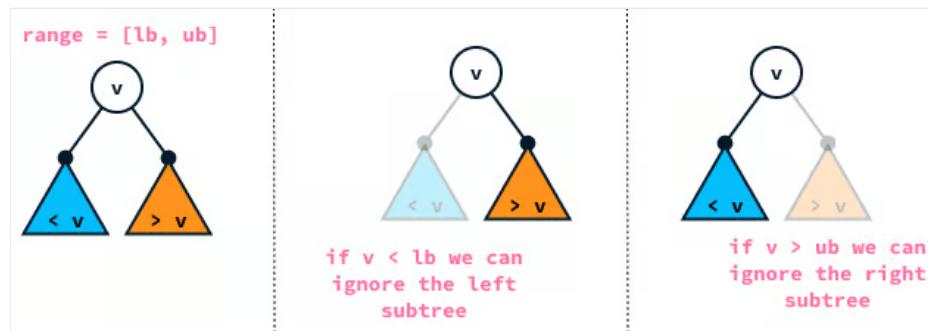
    self.restore_balance(rebalance_node)

```

POWERED BY  databricks

Range query

Range queries involve identifying all values that fall between two specified values. Due to the ordered nature of AVL trees, we can locate these values efficiently.



To locate all values within the range specified by lower bound `lb` and upper bound `ub`, we employ a recursive approach to search the tree. For each node encountered, if its value falls within this range, we include it in our results. Subsequently, we explore both the left and right subtrees to continue our search.

We ignore the left subtree if the node's value is less than the lower bound because all the values in the left subtree are smaller than the node's value. Similarly, we ignore the right subtree if the node's value is greater than the upper bound, as all values in the right subtree will be larger than the node's value.

```

def search(self, node, lb, ub, results):
    # Search for values between lower bound and upper bound

    if node is None:
        return

    if lb <= node.value and node.value <= ub:

```

```

        results.append(node.value)

    if node.value >= lb:
        self.search(node.left, lb, ub, results)

    if node.value <= ub:
        self.search(node.right, lb, ub, results)

def range_query(self, lb, ub):
    # Search for values between lower bound and upper bound
    results = []
    self.search(self.root, lb, ub, results)
    return results

```

POWERED BY  datacamp

Other Self-Balancing Trees

We have implemented an AVL tree capable of performing the following operations:

- Adding a value
- Deleting a value
- Looking up a value
- Querying the minimum and maximum
- Querying all values that fall between two values

[The avltree package](#) package offers a Python implementation reflecting these capabilities.

Other self-balancing binary search trees, such as red-black trees, splay trees, and B-trees, provide similar functionalities. Generally, their performance is comparable across most applications, as they all maintain a guaranteed logarithmic height. However, AVL trees are more finely balanced, optimizing search operations at the cost of potentially slower insertions due to rigorous rebalancing requirements.

Splay trees are particularly effective in scenarios where recently accessed elements are frequently reused, making them an excellent choice for cache implementations.

B-trees are uniquely designed to operate efficiently on disk rather than in memory, making them invaluable for managing large data sets that exceed memory capacities, such as in database index creation.

Further improvements

There are several ways in which our implementation can be improved. Here are a few suggestions for exercises to deepen your understanding of AVL trees:

- In our current implementation, nodes store only a single value. For usage as database indexes, it's necessary to store entire rows since the value will correspond to one of the table's columns. We can enhance our implementation to make the AVL tree function similarly to a dictionary, mapping values to their corresponding rows.
- Our implementation does not support duplicate values. However, it's possible to modify it to allow multiple nodes to share the same value.
- Typically, AVL trees are implemented recursively. We chose to avoid this approach to sidestep the need for a deep understanding of recursion. Although recursive implementations are generally more elegant and concise, they require a strong grasp of the recursion concept.

[DOWNLOAD NOW \(http://bit.ly/3i8lueA\)](http://bit.ly/3i8lueA)

BYJU'S GATE (/gate/) > GATE (/gate/gate-exam/) > GATE Study Material (/gate/study-material/) > GATE Notes For CSE (/gate/gate-notes-for-cse/) > Introduction to Data structures (/gate/introduction-to-data-structure-notes/) > AVL Trees (/gate/avl-trees-notes/)

**1-to-1 online math tutoring
starting at \$25/hr**

[Take a free demo](#)

(<https://ad-tech.byjusweb.com/revive/www/delivery/cl.php?bannerid=32&zzoneid=110&sig=5a9c9eaf02d76506dc9404f420666aca5c687de80af55930f806b04ff9ed4815&dest=https%3A%2F%2Fbyjus.com%2Fus%2Fmat>)

AVL Trees Notes for GATE

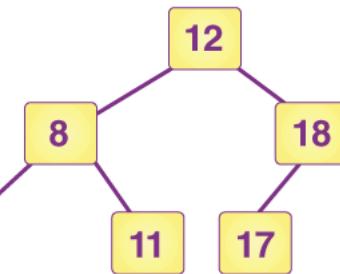
AVL Tree is an essential topic under one of the most important chapters of Computer Science i.e. Data Structure. And, when it comes to a competitive examination like GATE, you have to dive deep into this topic to understand it thoroughly. In this article, we have comprised all the pointers related to the AVL Tree. We hope these notes for CSE topics will help you understand this topic in a better way.

Table of Contents

- What is an AVL Tree?
- Balanced Factor in AVL Tree
- Why Do We Need an AVL Tree?
- Operations on AVL Tree
- AVL Rotations
- Time Complexity in AVL Tree
- Practice Problem – AVL Tree
- FAQs Related to AVL Tree

What is an AVL Tree?

The term AVL tree was introduced by Adelson-Velsky and Landis. It is a balanced binary search tree and is the first data structure like this. In the AVL tree, the heights of the subtree cannot be more than one for all nodes.



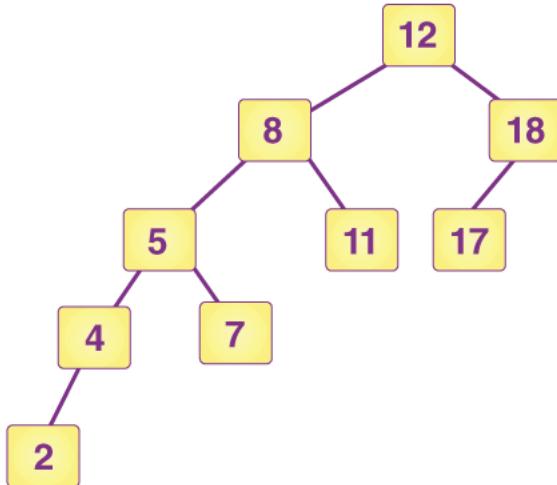
Live 1-to-1 Math classes and more starting @ \$25/hr

[Take a free demo](#)

Hi there! Got any questions?
I can help you...



<https://byjus.com/review/home/delivery.php?bannerid=41&zoneid=201&sig=f381f98bf689c3b7a945cc7ea916c7af78908e8ab3fb2063b0c3a21179e14d5a&dest=https%3A%2F%2Fk>



The above tree is not an AVL tree. And, the reason is simple. Here, the heights of the left and right subtrees are higher than 1.

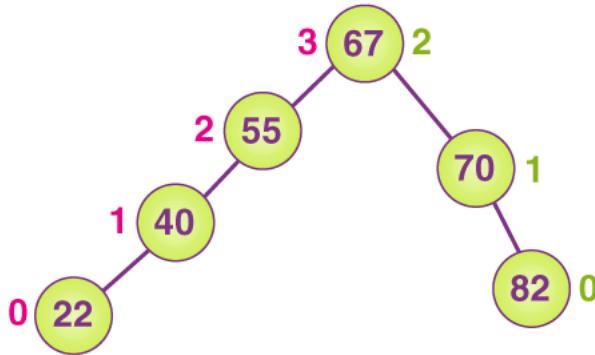
Balance Factor in AVL Tree

In the AVL tree, the term balance factor is very important.

$$\text{Balance Factor} = \text{height(left-subtree)} - \text{height(right - subtree)}$$

The balanced factor should be -1, 0 or +1. Otherwise, the tree will be considered an unbalanced tree.

Example Of AVL Tree & Balance Factor:



In the above example, the height of the left subtree is 3 and the height of the right subtree is 2. That means the balance factor is $<=1$ therefore the tree is supposed to be balanced.

Why Do We Need an AVL Tree?

To reduce the issue of time complexity in a binary search tree, the AVL tree was introduced by Adelson-Velski & Landis. It is a self-balancing tree that helps in reducing the complexity issue.

Operations on AVL Tree

The AVL tree is a balancing binary tree, and therefore it follows the same operations we perform in the binary search tree.

- 1. [10+1 Math classes and more starting @ \\$25/hr](#)
- 2. [Deletion](#)

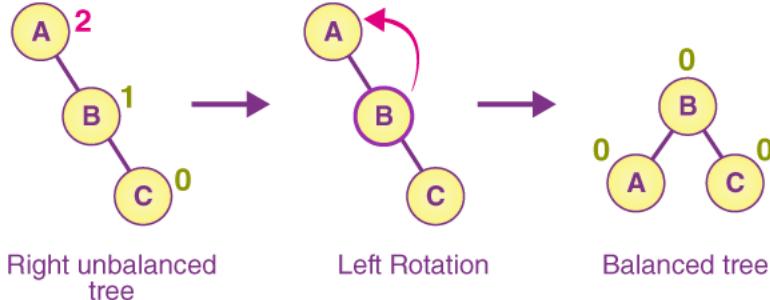
(<https://ad->

<https://byjus.com/review/binary-tree-delivery/> A node can be inserted in the binary search tree. However, there are chances that it may point to a violation in the AVL tree property, and we need to balance the tree.

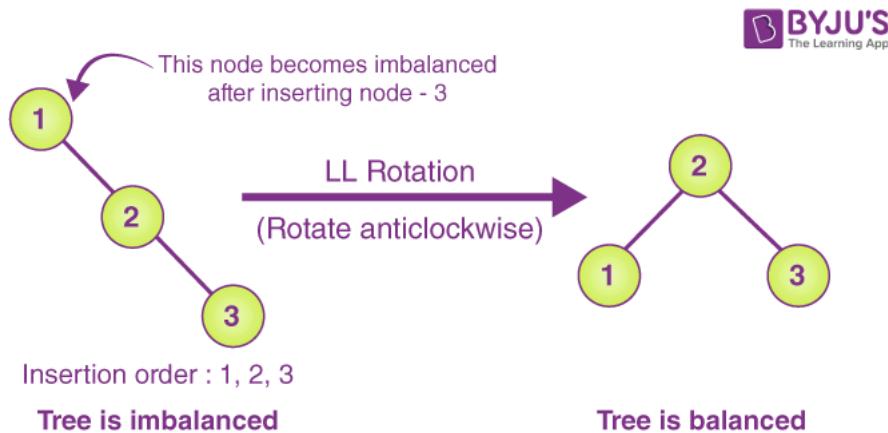
Deletion: The process of deletion is the same as it is executed in a binary search tree. It can affect the balance factor of the tree, therefore, we need to utilize different types of rotations to balance the tree.

AVL Rotation

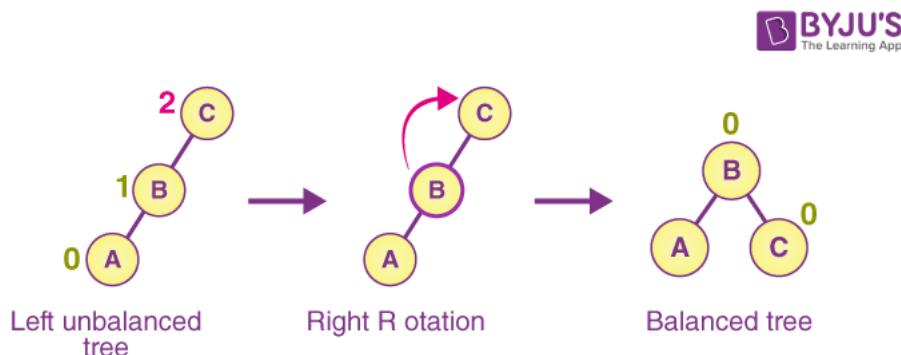
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

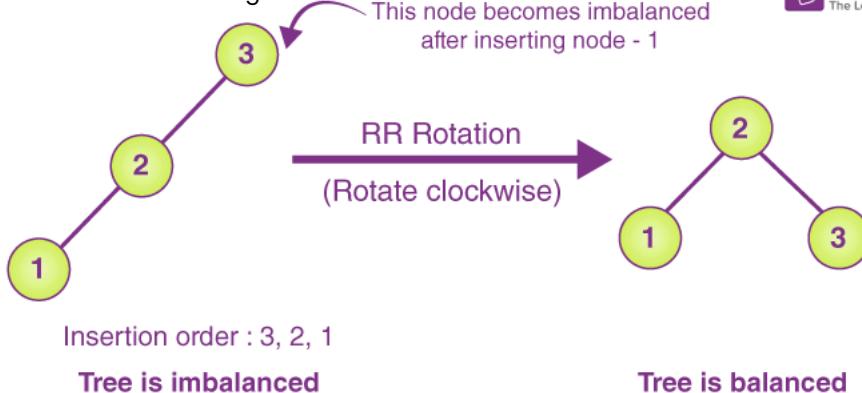


1. Left Rotation: When we perform insertion at the left subtree, then it is a left rotation.

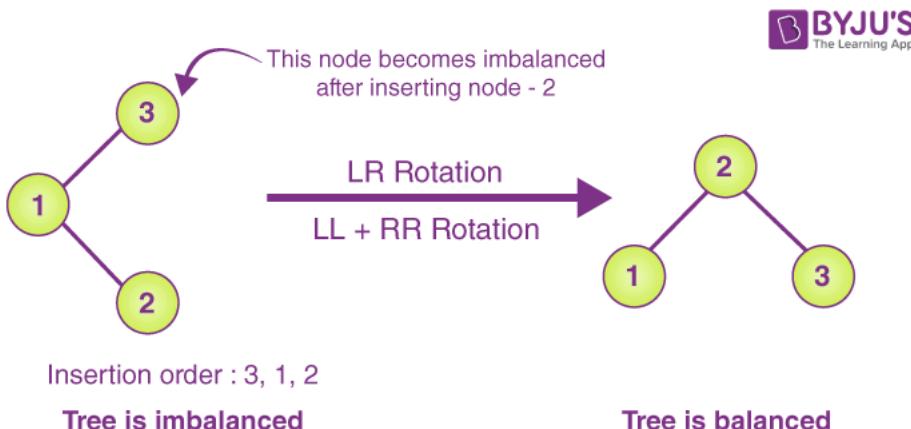


1. Right Rotation: When we perform insertion at the right subtree, then it is a right rotation.





1. Left-Right Rotation



1. Right-Left Rotation

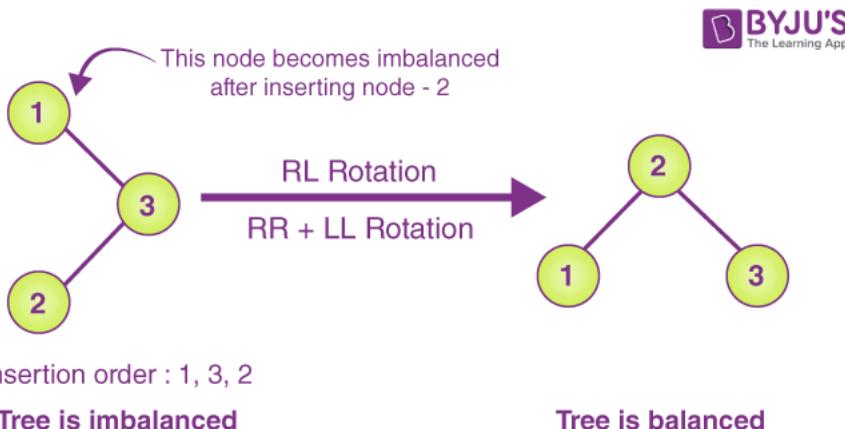


Figure: Right-Left Rotation



Live 1-to-1 Math classes and more
starting @ \$25/hr

(<https://ad->

The Complexity of AVL Tree

<https://byjus.com/review/www/delivery/cl.php?bannerid=41&zoneid=201&sig=f381f98bf689c3b7a945cc7ea916c7af78908e8ab3fb2063b0c3a21179e14d5a&dest=https%3A%2F%2F>

Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Practice Problem – AVL Tree

Q. What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0.

- (A) 2
- (B) 3
- (C) 4
- (D) 5

Frequently Asked Questions Related to AVL Tree

Q1 What is the formula for the balance factor in an AVL tree?

Balance Factor = height(left-subtree) - height(right-subtree)

Q2 What does AVL stand for?

In Computer Science, AVL stands for Adelson-Velski & Landis. This is because the AVL tree was introduced by these two inventors.

Keep learning and stay tuned to get the latest updates on GATE Exam (<https://byjus.com/gate/gate-exam/>) along with GATE Eligibility Criteria (<https://byjus.com/gate/gate-eligibility-criteria/>), GATE 2023 (<https://byjus.com/gate/>), GATE Admit Card (<https://byjus.com/gate/gate-admit-card/>), GATE Syllabus for CSE (Computer Science Engineering) (<https://byjus.com/gate/gate-syllabus-for-computer-science-engineering/>), GATE CSE Notes (<https://byjus.com/gate/gate-notes-for-cse/>), GATE CSE Question Paper (<https://byjus.com/gate/gate-cse-question-paper/>), and more.

Also Explore,

- Introduction to Tree (<https://byjus.com/gate/tree-notes/>)
- AVL Trees (<https://byjus.com/gate/avl-trees-notes/>)
- B Tree (<https://byjus.com/gate/b-tree-notes/>)
- Binary Search Trees (<https://byjus.com/gate/binary-search-trees-notes/>)
- Minimum Spanning Tree (<https://byjus.com/gate/minimum-spanning-tree-notes/>)
- Spanning Tree (<https://byjus.com/gate/spanning-tree-notes/>)
- Tree Topology (<https://byjus.com/gate/tree-topology-notes/>)
- Tree Traversal (<https://byjus.com/gate/tree-traversal-notes/>)
- Graphs and their Applications (<https://byjus.com/gate/graph-and-its-applications/>)
- Stacks and their Applications (<https://byjus.com/gate/stack-and-its-applications/>)
- Queues Notes (<https://byjus.com/gate/queue-notes/>)
- Introduction to Recursion (<https://byjus.com/gate/recursion-notes/>)

GATE Related Links

Unary Operator In C
Live 1-to-1 Math classes and more
starting @ \$25/hr

(<https://byjus.com/gate/unary-operator-in-c/>)

Difference Between Risc And Cisc
(<https://byjus.com/gate/risc-and-cisc/>)

How Is An Encoder Different From A	(https://byjus.com/gate/difference-between-encoder-and-decoder/)	Size Of Data Types In C
Difference Between Unique And Primary Key	(https://byjus.com/gate/difference-between-primary-key-and-unique-key/)	Introduction To C Programming
Java And Javascript Difference	(https://byjus.com/gate/difference-between-java-and-javascript/)	What Is Recursion In C
Html And Html5 Difference	(https://byjus.com/gate/difference-between-html-and-html5/)	Difference Between Function And Procedure

Comments

Leave a Comment

Your Mobile number and Email id will not be published. Required fields are marked *

Mobile Number	Send OTP
---------------	----------

Type your message or doubt here...

*

*

Post My Comment

Join BYJU'S Learning Program

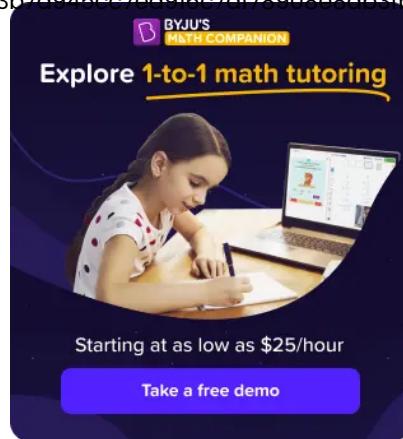
Name
Mobile Number
<input type="button" value="Submit"/>



Live 1-to-1 Math classes and more
starting @ \$25/hr

(<https://ad->

tech.byjusweb.com/revive/www/delivery/cl.php?
bannerid=41&zoneid=201&sig=f381f98bf689c3b7a945cc7ac916c7acf78908e8cb3fb2063b0c3a21179e14d5a&dest=https%3A%2F%2Fk



(<https://ad-tech.byjusweb.com/revive/www/delivery/cl.php?bannerid=39&zoneid=111&sig=62b9c979575895cf02bc7a6b50a9603bc2a83e39bdddc0fce3945253e0e6b4c&dest=https%3A%2F%2Fbyjus.com%2Fus%2Fr>)

 Live 1-to-1 Math classes and more
starting @ \$25/hr

(<https://ad->

tech.byjusweb.com/revive/www/delivery/cl.php?
bannerid=41&zoneid=201&sig=f381f98bf689c3b7a945cc7ac916c7acf78908e8cb3fb2063b0c3a21179e14d5a&dest=https%3A%2F%2Fk



(<https://ad-tech.byjusweb.com/revive/www/delivery/cl.php?bannerid=39&zoneid=333&sig=b23298c62d38f69d01df51a0c12eb285f9e40f17eadd1d09cd1fee428c8c7bb3&dest=https%3A%2F%2Fbyjus.com%2Fus%2Fmat>

 Live 1-to-1 Math classes and more
starting @ \$25/hr

(<https://ad->

tech.byjusweb.com/revive/www/delivery/cl.php?
bannerid=41&zoneid=201&sig=f381f98bf689c3b7a945cc7ac916c7acf78908e8cb3fb2063b0c3a21179e14d5a&dest=https%3A%2F%2Fk



(<https://ad-tech.byjusweb.com/revive/www/delivery/cl.php?bannerid=39&zoneid=334&sig=d11d42d0e240720256ff03884d096b75ba75319d2ce922ec15648de50265f10a&dest=https%3A%2F%2Fbyjus.com%2Fus%2Fmc>)

Explore Other Popular Articles

[Introduction To Array](#)

[Binary Heaps Notes](#)

[Introduction To Data Structures](#)

 Live 1-to-1 Math classes and more
[Linked List Notes](#)
starting @ \$25/hr

(<https://ad-tech.byjusweb.com/revive/www/delivery/cl.php?bannerid=39&zoneid=334&sig=d11d42d0e240720256ff03884d096b75ba75319d2ce922ec15648de50265f10a&dest=https%3A%2F%2Fbyjus.com%2Fus%2Fmc>)

<https://byjusweb.com/revive/www/delivery/cl.php?bannerid=41&zoneid=201&sig=f381f98bf689c3b7a945cc7ea916c7af78908e8ab3fb2063b0c3a21179e14d5a&dest=https%3A%2F%2Fk>

Introduction To Recursion

String In C

Introduction To Tree

Decomposition In DBMS

Derived Data Types In C

COURSES

CBSE (/cbse/)
 ICSE (/icse/)
 CAT (/cat/)
 IAS (/ias/)
 JEE (/jee/)
 NEET (/neet/)
 Commerce (/commerce/)
 JEE Main (/jee/jee-main/)
 NCERT (/ncert/)
 JEE Advanced (/jee-advanced/)
 UPSC Prelims 2022 Question Paper (/free-ias-prep/upsc-prelims-2022-question-papers-pdf/)
 UPSC Prelims 2022 Answer Key (/free-ias-prep/upsc-prelims-answer-key-2022/)
 IAS Coaching (/ias-coaching/)
 CBSE Sample Papers (/cbse/cbse-sample-papers/)
 CBSE Question Papers (/cbse-study-material/cbse-previous-year-question-paper/)

EXAM PREPARATION

Free CAT Prep (/free-cat-prep/)
 Free IAS Prep (/free-ias-prep/)
 Maths (/maths/)
 Physics (/physics/)
 Chemistry (/chemistry/)
 Biology (/biology/)
 JEE 2024 (/jee/jee-2024/)
 JEE Advanced 2023 Question Paper with Answers (/jee/jee-advanced-2023-question-paper/)
 JEE Main Mock Test (/jee/jee-main-mock-test/)
 JEE Main 2024 Question Papers with Answers (/jee/jee-main-2024-question-papers/)
 JEE Main 2023 Question Papers with Answers (/jee/jee-main-2023-question-papers/)
 **JEE Advanced 2022 Question Paper with Answers** (/jee/jee-advanced-2022-question-paper/)
Live 1-to-1 Math classes and more
ET 2023 Question Paper (/neet/neet-2023-question-paper/)
 NEET 2023 Question Paper Analysis (/neet/neet-2023-question-paper-analysis/)

EXAMS

CAT Exam (/cat/exam-info/)
 CAT 2023 (/cat/cat-2023/)
 GATE Exam (/gate/gate-exam/)
 GATE 2024 (/gate/gate-2024/)
 IAS Exam (/ias-exam/)
 UPSC Exam (/free-ias-prep/upsc-exam/)
 UPSC Syllabus (/ias/upsc-syllabus/)
 UPSC 2023 (/free-ias-prep/upsc-2023/)
 Bank Exam (/bank-exam/)
 Government Exams (/govt-exams/)
 Education News (/news/)

CLASSES

Kids Learning (/kids-learning/)
 Class 1st – 3rd (/disney-byjus-early-learn/)
 Class 4th – 5th (/class-4-5/)
 Class 6th – 10th (/class-6-10/)
 Class 11th – 12th (/class-11-12/)
 BYJU'S Tuition Centre (/btc/)

COMPANY

About Us (/about-us/)
 Contact Us (/contact-us/)
 Contact our Financial Partners (<https://byjus.com/our-financial-partners/>)
 Investors (/our-investors/)
 Compliance (/compliance/)
 Careers (/careers-at-byjus/)
 CIRP (/cirp/)
 BYJU'S in Media (/press/)
 Social Initiative – Education for All (/educationforall/)
 BYJU'S APP (/byjus-the-learning-app/)
 FAQ (/faq/)
 Support (/customer-care/)
 Students Stories – The Learning Tree (<https://blog.byjus.com/the-learning-tree/>)
(https://ad-

RESOURCES

- CAT College Predictor (/free-cat-prep/cat-college-predictor/)
- Worksheets (/worksheets/)
- BYJU'S Answer (/question-answer/)
- DSSL (/about-dssl/)
- Home Tuition (/home-tuition/)
- All Products (<https://shop.byjus.com>)
- Calculators (/calculators/)
- Formulas (/formulas/)

FREE TEXTBOOK SOLUTIONS

- (/textbook-solutions/)
- NCERT Solutions (/ncert-solutions/)
- NCERT Exemplar (/ncert-exemplar/)
- NCERT Solutions for Class 6 (/ncert-solutions-class-6/)
- NCERT Solutions for Class 7 (/ncert-solutions-class-7/)
- NCERT Solutions for Class 8 (/ncert-solutions-class-8/)
- NCERT Solutions for Class 9 (/ncert-solutions-class-9/)
- NCERT Solutions for Class 10 (/ncert-solutions-class-10/)
- NCERT Solutions for Class 11 (/ncert-solutions-class-11/)
- NCERT Solutions for Class 11 English (/ncert-solutions-class-11-english/)
- NCERT Solutions for Class 12 English (/ncert-solutions-class-12-english/)
- NCERT Solutions for Class 12 (/ncert-solutions-class-12/)
- RD Sharma Solutions (/rd-sharma-solutions/)
- RD Sharma Class 10 Solutions (/rd-sharma-class-10-solutions/)
- ICSE Selina Solutions (/icse/selina-solutions/)

STATE BOARDS

- Maharashtra (/msbshse/)
- Gujarat (/gseb/)
- Tamil Nadu (/tn-board/)
- Karnataka (/kseeb/)
- Kerala (/kbpe/)
- Andhra Pradesh (/ap-board/)
- Telangana (/telangana-board/)
- Uttar Pradesh (/upmsp/)
- Bihar (/bihar-board/)
- Rajasthan (/rajasthan-board/)
- Madhya Pradesh (/mp-board/)
- West Bengal (/west-bengal-board/)

FOLLOW US

(<https://www.facebook.com/byjuslearningapp/>) (<https://in.linkedin.com/company/byjus>)

[Disclaimer \(/disclaimer/\)](#) [Privacy Policy \(/tnc_app/#privacydesc\)](#) [Terms of Services \(/tnc_app/#tncdesc\)](#) [Sitemap \(/sitemap.xml\)](#)

© 2025, BYJU'S. All rights reserved.



Live 1-to-1 Math classes and more
starting @ \$25/hr

(<https://ad->

AVL trees are a special type of binary search tree that keep themselves balanced. They're named after their inventors Adelson-Velsky and Landis. Here's a simple explanation of AVL trees:

What is an AVL Tree?

An AVL tree is like a regular binary search tree, but with an extra rule: the heights of the left and right subtrees of any node can't differ by more than one¹². This difference is called the balance factor. The balance factor can only be -1, 0, or +1³.

Why Use AVL Trees?

AVL trees help solve the problem of regular binary search trees becoming unbalanced and slow. By keeping themselves balanced, AVL trees ensure that operations like searching, inserting, and deleting always take a reasonable amount of time (specifically, $O(\log n)$ time)².

How AVL Trees Work

1. Balance Factor: For each node, we calculate its balance factor by subtracting the height of its right subtree from the height of its left subtree³.
2. Self-Balancing: When we add or remove nodes, the tree might become unbalanced. If this happens, the tree performs special operations called rotations to fix itself⁴.

Types of Rotations

There are four types of rotations:

1. Left Rotation (LL Rotation): When a node becomes too heavy on the right side⁴.
2. Right Rotation (RR Rotation): When a node becomes too heavy on the left side⁴.
3. Left-Right Rotation (LR Rotation): A combination of a left rotation followed by a right rotation⁴.
4. Right-Left Rotation (RL Rotation): A combination of a right rotation followed by a left rotation⁴.

Examples of Insertions and Rotations

Let's look at some examples of inserting nodes and the rotations that might be needed:

1. Simple Insertion (No Rotation Needed):

Start with: 5

Insert 3:

5

/

3

No rotation needed, tree is balanced.

2. Left Rotation Example:

Start with:

5

7

Insert 8:

5

7

8

This triggers a left rotation:

7

/

5 8

3. Right Rotation Example:

Start with:

7

/

5

Insert 3:

7

/

5

/

3

This triggers a right rotation:

5

/

3 7

4. Left-Right Rotation Example:

Start with:

7
/
3
Insert 5:

7
/
3

5
This triggers a left-right rotation:

5
/
3 7

5. Right-Left Rotation Example:

Start with:
3

7
Insert 5:
3

7
/
5
This triggers a right-left rotation:

5
/
3 7

After each insertion, the tree checks its balance and performs rotations if needed to stay balanced⁷.

Remember, the goal of all these rotations is to keep the tree balanced, which means keeping the height difference between left and right subtrees to no more than one for every node¹².

Example 1: Simple Insertion (No Rotation)

Initial:

5

Action: Insert 3

Result:

5

/

3

Rotation: None

Final:

5

/

3

Example 2: Left Rotation

Initial:

5

7

Action: Insert 8

Result:

5

7

8

Rotation: Left

Final:

7

/

5 8

Example 3: Right Rotation

Initial:

7

/

5

Action: Insert 3

Result:

7

/

5

/

3

Rotation: Right

Final:

5

/

3 7

Example 4: Left-Right Rotation

Initial:

7

/

3

Action: Insert 5

Result:

7

/

3

5

Rotation: Left-Right

Final:

5

/

3 7

Example 5: Right-Left Rotation

Initial:

3

7

Action: Insert 5

Result:

3

7

/

5

Rotation: Right-Left

Final:

5

/

3 7

Example 6: Multiple Insertions (No Rotation)

Initial:

5

Action: Insert 3, then 7

Result:

5

/

3 7

Rotation: None

Final:

5

/

3 7

Example 7: Left Rotation after Multiple Insertions

Initial:

5

/

3 7

Action: Insert 8, then 9

Result:

5

/

3 7

8

9

Rotation: Left

Final:

7

/

5 8

/

3 9

Example 8: Right Rotation after Multiple Insertions

Initial:

5

/

3 7

Action: Insert 2, then 1

Result:

5
/
3 7
/
2
/
1

Rotation: Right

Final:

3
/
2 5
/
1 7

Example 9: Left-Right Rotation after Multiple Insertions

Initial:

7
/
3

Action: Insert 2, then 5

Result:

7
/
3
/
2 5

Rotation: Left-Right

Final:

5
/
3 7
/
2

Example 10: Right-Left Rotation after Multiple Insertions

Initial:

3

7

Action: Insert 8, then 5

Result:

3

7

/

5 8

Rotation: Right-Left

Final:

5

/

3 7

8

Example 11: Double Rotation (Left-Right)

Initial:

8

/

4

Action: Insert 6

Result:

8

/

4

6

Rotation: Left-Right

Final:

6

/

4 8

Example 12: Double Rotation (Right-Left)

Initial:

2

6

Action: Insert 4

Result:

2

6

/

4

Rotation: Right-Left

Final:

4

/

2 6

Example 13: Multiple Rotations

Initial:

5

/

3 7

Action: Insert 1, then 2

Result:

5

/

3 7

/

1

2

Rotation: Left-Right, then Right

Final:

3

/

2 5

/

1 7

Example 14: Insertion Causing Multiple Rotations

Initial:

5

/

3 7

/

1 9

Action: Insert 0

Result:

5
/
3 7
/
1 9
/
0

Rotation: Right, then Right

Final:

3
/
1 5
/\
0 2 7

9

Example 15: Insertion at Root

Initial:

Empty tree

Action: Insert 5

Result:

5

Rotation: None

Final:

5

Example 16: Balancing a Skewed Tree

Initial:

1

2

3

4

Action: Insert 5

Result:

1

2

3

4

5

Rotation: Left (multiple times)

Final:

3

/

2 4

/

1 5

Example 17: Insertion Causing Alternating Rotations

Initial:

5

/

3 7

Action: Insert 4, then 2

Result:

5

/

3 7

/

2 4

Rotation: Left-Right

Final:

4

/

3 5

/

2 7

Example 18: Complex Insertion Scenario

Initial:

8

/

4 12

/

2 6

Action: Insert 1, then 3

Result:

8

/

4 12

/

2 6

/

1 3

Rotation: Right, then Left-Right

Final:

4

/

2 8

/ \

1 3 12

/

6

Example 19: Insertion Causing Multiple Level Rotations

Initial:

10

/

5 15

/ \

3 7 20

Action: Insert 4

Result:

10

/

5 15

/ \

3 7 20

/

4

Rotation: Left-Right, then Right

Final:

10

/
5 15

/ \\\n4 7 20

/\n3

Example 20: Insertion in a Perfectly Balanced Tree

Initial:

4\n/\n2 6\n/ \\\n1 3 5 7

Action: Insert 8

Result:

4\n/\n2 6\n/ \\\n1 3 5 7

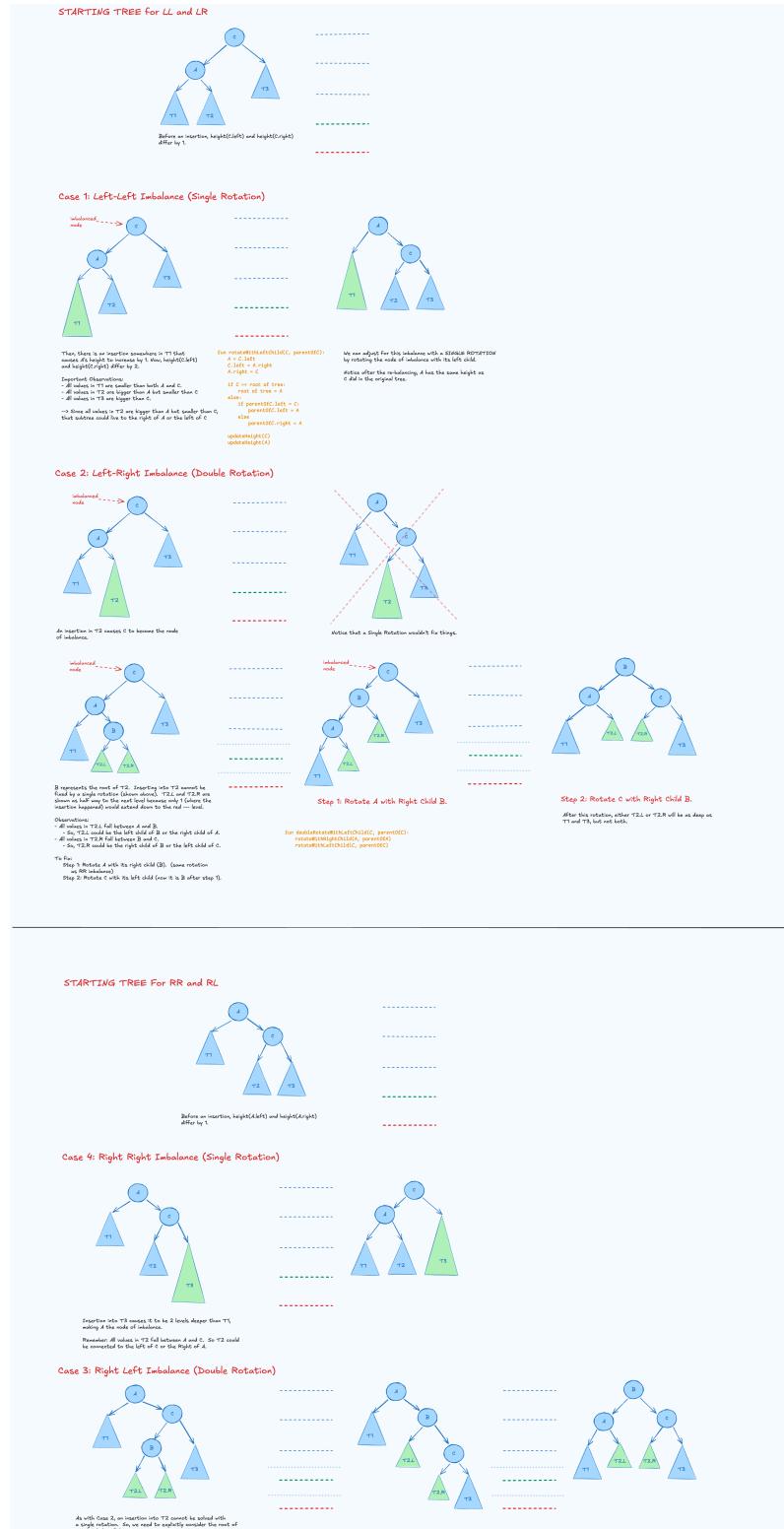
8

Rotation: Left

Final:

4\n/\n2 6\n/ \\\n1 3 5 7

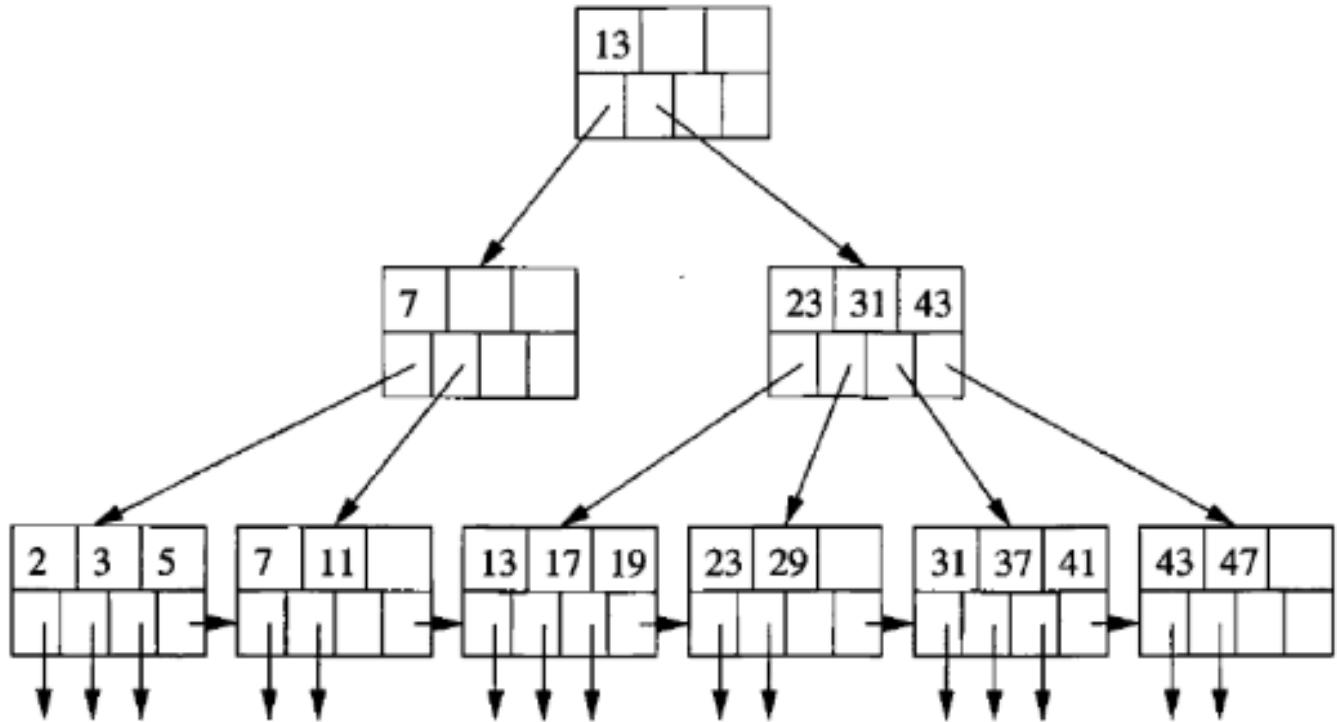
8





B+ Tree: How is Indexing of Databases Implemented?

Posted May 6, 2024 • Updated Oct 6, 2024



In (CS) Learning Note

10 min read

Contents

- [Introduction to B+ Tree](#)
 - [Structure of B+ Trees](#)
 - [Key Features](#)
 - [Advantages of B+ Trees in Database](#)
- [Why Use B+ tree for Database Indexing?](#)
 - [Features of Database Indexing](#)
 - [Can We Use Other Data Structures?](#)

- [Indexing of Databases: From Binary Search Tree to B+ Tree](#)
 - [Connecting Leaf Nodes Using a Linked List for Range Searches](#)
 - [Using Disk for Storing Large Amounts of Indexes](#)
 - [Minimise Tree Height for Reducing Disk IO](#)
 - [Maintain the Size of Each Node](#)
-

1 Introduction to B+ Tree

A B+ Tree is a type of **self-balancing tree** data structure that maintains sorted data and allows **searches, sequential access, insertions, and deletions** in logarithmic time. It is an extension of the B-Tree, enhancing it with optimized leaf nodes for sequential traversal.

In the realm of database management and file systems, the B+ Tree stands out as a highly efficient data structure for storing and managing large blocks of sorted data.

1.1 Structure of B+ Trees

- **Nodes:** B+ Trees consist of internal nodes and leaf nodes. Internal nodes direct the search and do not store actual data records but keys that act as separators directing queries to the correct leaf nodes.
- **Leaf Nodes:** These nodes contain the actual data entries and are linked sequentially, facilitating efficient range queries and sequential access.
- **Root Node:** The top node from which the search begins, potentially spanning multiple levels down to the leaves.

1.2 Key Features

- **High Fan-out:** Each node in a B+ Tree contains a large number of children, which reduces the tree's height and the number of disk I/O operations required.
- **Leaf Node Linkages:** Leaf nodes of B+ Trees are linked, providing an ordered linked list of the entries for quick traversal of all records.

- **Balance:** Every path from the root to a leaf node is of the same length, ensuring that operations remain balanced and efficient.

1.3 Advantages of B+ Trees in Database

- **Efficiency in Range Queries:** The linked leaves allow for fast and efficient range queries, which are common in database operations.
- **Minimized Disk I/O:** High fan-out reduces the depth of the tree, thus reducing the disk reads required during operations.
- **Optimized for Storage Systems:** B+ Trees are designed to match the block size of physical disks, maximizing the use of disk space and reducing overhead.

2 Why Use B+ tree for Database Indexing?

Nowadays, many database engines choose B+ Tree for data indexing. **But why?**

2.1 Features of Database Indexing

The main reason for this is due to the features of database indexing itself.

- **For database searches, we can categorise frequently perform operations into these two categories:**
 1. **Search for data based on a value.** For example, `select name from user where id=1234`
 2. **Search for data based on range values.** For example, `select name from user where id > 1234 and id < 2345`
- **For performance requirements, we mainly consider both time and space**, i.e. execution efficiency and storage space.
 - For execution efficiency, **we want the index to search the data as efficiently as possible;**
 - For storage space, **we want the index not to consume too much memory space.**

2.2 Can We Use Other Data Structures?

For fast search, insertion, deletion, we can also use **hash tables**, **balanced binary search trees**, and **skip lists**.

But for database indexes, they all have their own limitations:

- **Hash tables:**

- **Lack of ordering:** Hash tables do not maintain any ordering among keys, which can be a limitation in certain applications where ordered traversal is required.
- **Not suitable for range searches:** Hash tables are not optimized for range queries, as they rely on exact key matches for retrieval.
- **Hash collisions:** If multiple keys hash to the same index (a collision), additional operations are needed to handle these collisions, which can degrade performance.

- **Balanced binary search trees:**

- **Uncontrollable tree height:** If the amount of data is large or the inserted data has a specific distribution, it will lead to an increase in the height of the tree, which will affect the performance of the search.
- **Not suitable for range searches:** While balanced binary search trees support efficient searching for individual keys, they are not inherently optimized for range queries (e.g., finding all keys within a certain range).

- **Skip lists:**

- **Not suitable for persistence:** Skip list implementations are relatively complex, especially when persistent storage is required. Since the structure of a skip list relies on randomness, more work is required to ensure proper persistence on disk.
- **Uncertain Performance:** The performance of skip lists depends heavily on the randomness in their structure. Although skip lists have a good performance in the average case, in the worst case its performance may degrade to $O(n)$, which is unacceptable.

💡 In fact, the B+ tree used for database indexing is very similar to a skip list (non-leaf nodes only store indexes and don't store data). However, **the B+ tree evolved from a binary search tree**, not a skip list.

3 Indexing of Databases: From Binary Search Tree to B+ Tree

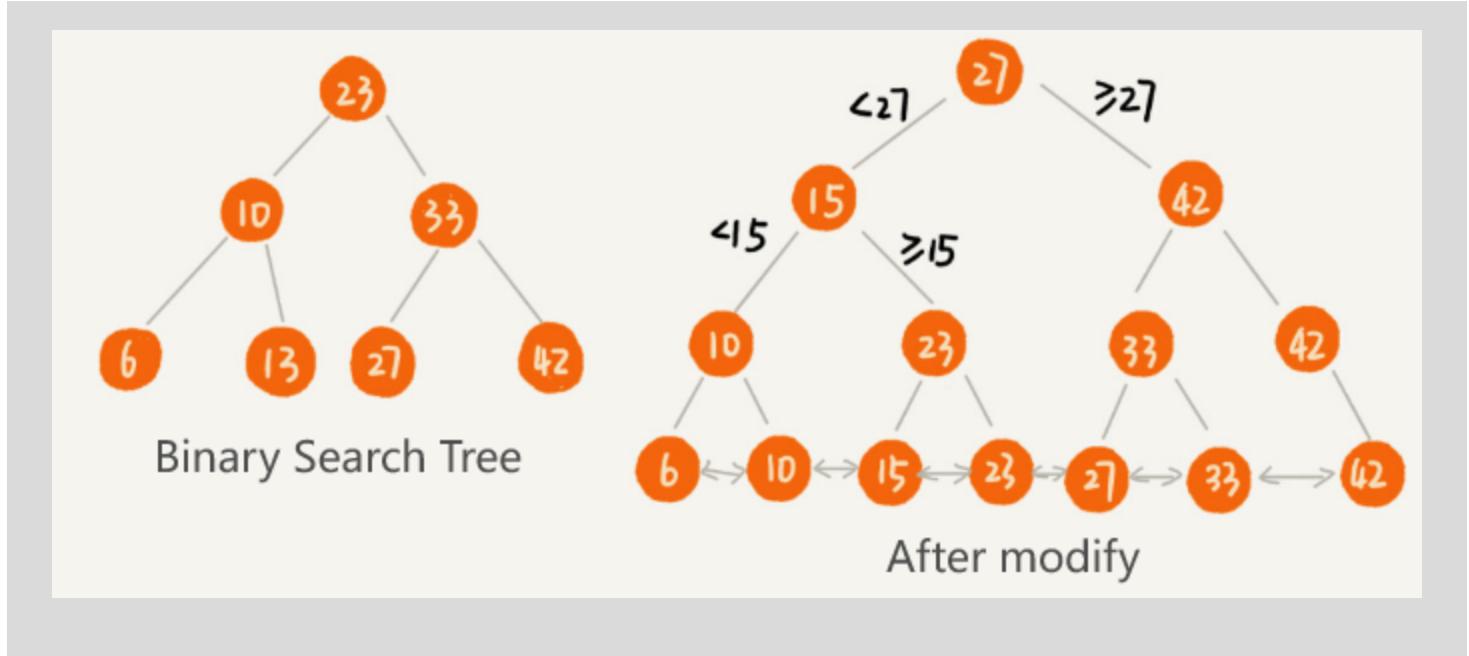
To restore the whole process of inventing the B+ tree, we start with a binary search tree and see how it is transformed into a B+ tree step by step based on requirements.

3.1 Connecting Leaf Nodes Using a Linked List for Range Searches

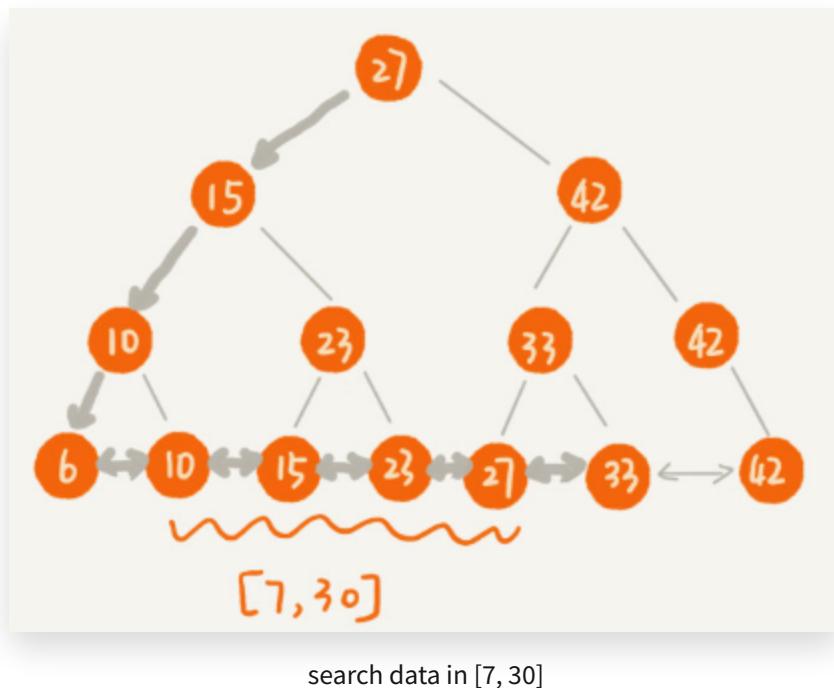
As I said before, Binary Search Tree is not suitable for range search. So that's the first problem we have to solve.

- To make a binary search tree **better support searching data by range**, we can modify it like this:
 - The nodes in the tree do not store the data itself, but rather just serve as indexes.
 - Each leaf node is linked on a linked list, and the data in the linked list is ordered from smallest to largest.

The modified binary tree looks like a skip list:



- After the transformation, if we require data in a certain range:
 - We just need to do a search in the tree to find the first leaf node that satisfies the range.
 - Then traverse back down the link list until the value of the node data in the link list is greater than the termination value of the range.



search data in $[7, 30]$

But this causes another problem:

- **Need extra space to store index nodes.**

3.2 Using Disk for Storing Large Amounts of Indexes

- We also need to consider scenarios where the amount of data is too large.
 - For example, if we build a binary search tree index for a table with 100 million of data, the index will contain about 100 million nodes. Assuming 16 bytes per node, that's about 1GB of memory space.
 - If we were to index 10 tables or even more, the memory requirements would be insatiable.

How can we solve this problem of the index taking up too much memory?

- **We can store the index on the disk.**

But this causes another problem:

- **Disk IO is very time-consuming compared to memory reads and writes.** So the focus of our optimisation is to minimise disk IO operations.

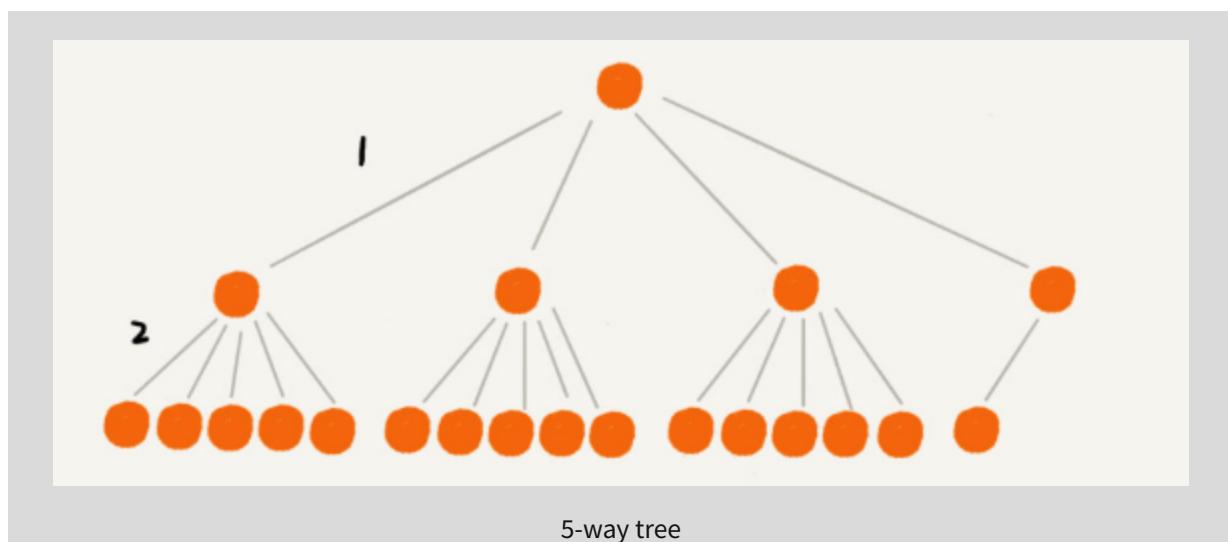
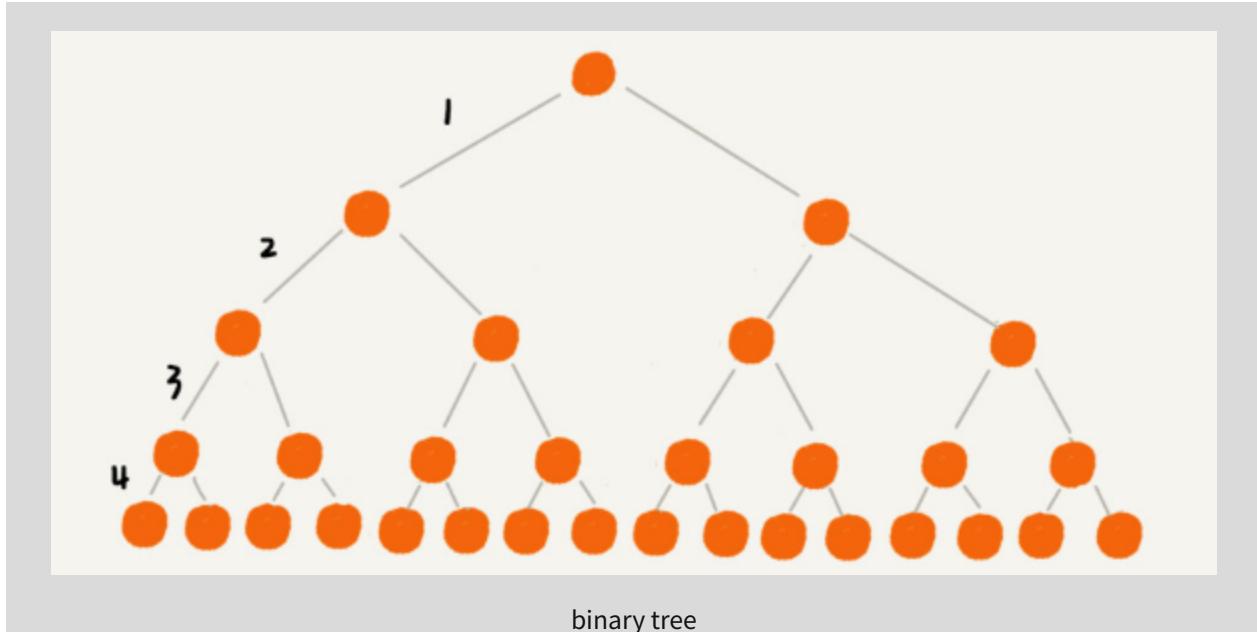
3.3 Minimise Tree Height for Reducing Disk IO

If the tree is stored on the disk, then each read (or access) to a node corresponds to a disk IO operation.

- **The height of the tree is equal to the number of disk IO operations for each data query.**
- So we need to **minimise the height of the tree**

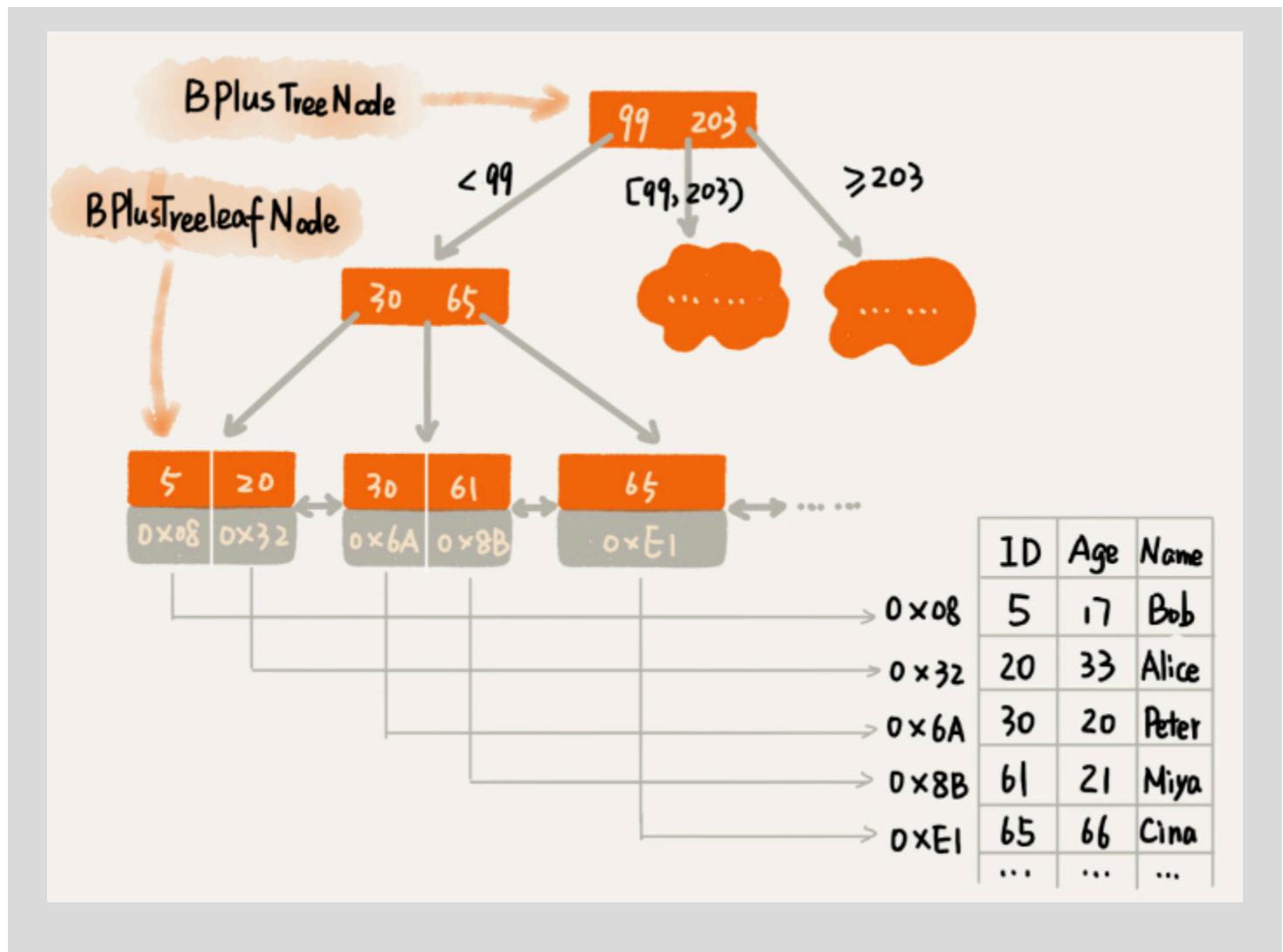
How to reduce the height of the tree?

- We can build the index as an **m-way tree ($m > 2$)**.



- From the above figure, it can be seen that storing the same number of nodes, the m-way tree ($m > 2$) has a lower height than the binary tree.
 - The larger the m in the m-way tree, the smaller the height of the m-way tree.

- So what's the best value for m ?
- No matter the data in memory or on the disk, **the operating system reads the data on a page-by-page basis:**
 - The size of a page is usually 4KB , and this value can be viewed with the `getconfig PAGE_SIZE` command.
 - **The data will be read one page at a time.**
 - If the amount of data to be read **exceeds the size of one page, multiple IO operations will be triggered.**
- Therefore, we should try to **make the size of each node (m size) equal to the size of a page:**
 - Reading a node requires only one disk IO operation.
 - Whilst minimise the height of the tree.



3.4 Maintain the Size of Each Node

In order to minimise IO, the m -value (size of each node) of the B+ tree is pre-calculated based on the size of the page, i.e. **each node can have at most m children**.

With the increase of data in the database, it is possible for some nodes to have more than m children. In this case, the size of the node exceeds the size of a page, and reading such a node results in multiple disk IO operations.

- **How to solve the problem of a B+ tree with the node size greater than m ?**
 - It's very simple. We just need to **split this node into two nodes**.
 - However, after the node is split, it is possible that its parent node has more than m children. But it doesn't matter. We can use the same method to split the parent node into two nodes as well. This cascading reaction works from the bottom up to the root node.
 - **Similarly, when we delete data, we may also need to update the index nodes.** Frequent data deletion results in the number of children of some nodes becoming very small. This can lead to less efficient indexing.
 - For example, we can merge nodes which are too small.

So, **Insertion** in a B+ Tree follows these steps:

1. **Finding the Correct Leaf Node:** Start from the root and traverse down to find the appropriate leaf node where the new key should be inserted.
2. **Inserting in the Leaf Node:** Insert the new key in the sorted order of keys in the leaf node.
3. **Splitting:** If the leaf node overflows (i.e., exceeds the maximum number of entries it can hold), it is split into two, and the median is pushed up to the parent node. This may cause a recursive split up the tree if the parent node also overflows.

Deletion involves removing an entry from the leaf node and then possibly adjusting the tree structure to maintain balance:

1. **Leaf Node Adjustment:** After deletion, if a leaf node underflows (i.e., has fewer entries than the minimum required), it may borrow an entry from a sibling node or merge with a sibling.

2. Propagation Upwards: If a merge occurs, it may reduce the number of entries in the parent node, possibly leading to further merging or adjustments up the tree.

- 💡 • This is why **adding indexes**, while **making searching data more efficient**, also **makes writing data less efficient**:
 - **The process of writing data will involve index updates.** The more indexes there are, the more updates are involved.
 - So in practice, in order to ensure the overall efficiency of database operations, **we only build indexes for the necessary attributes.**

- 💡 In addition to the B+ tree, you may have heard of the B tree (or B-tree). **The B tree is actually a Simplified version of the B+ tree.** Their main differences are:
 - Nodes in the B+ tree store indexes, while nodes in the B tree store data.
 - The leaf nodes in a B tree do not need a linked list to be chained together.
 - This leads to **B tree being unsuitable for** common database operations such as **sorting, range searching, and traversal.**

This means that the B-tree is more like a normal self-balancing m-way tree.

Reference:

- Wang, Zheng (2019) *The Beauty of Data Structures and Algorithms*. Geek Time.
- *Introduction of B-tree* (2023) GeeksforGeeks. Available at:
<https://www.geeksforgeeks.org/introduction-of-b-tree-2/>
- <https://dba.stackexchange.com/questions/155945/b-tree-structure-with-buckets-begginer-question>

A B+ tree is a self-balancing tree data structure that is commonly used in databases and file systems for efficient data storage and retrieval¹⁵. Here are some key characteristics of B+ trees:

1. Structure: B+ trees consist of a root, internal nodes, and leaf nodes¹.
2. Data storage: Unlike B-trees, B+ trees store all data records in the leaf nodes, while internal nodes contain only keys and pointers to child nodes⁵.
3. Leaf node linkage: All leaf nodes in a B+ tree are linked together in a sequential order, forming a linked list⁵.
4. Order: B+ trees have an order 'm', which determines the maximum number of children a node can have (m) and the maximum number of keys it can store (m-1)³.
5. Balance: All leaf nodes are at the same level, ensuring the tree remains balanced⁴.
6. Occupancy: Each node, except the root, must be at least half full, containing between d and 2d entries, where d is the minimum degree of the tree².
7. Efficient operations: B+ trees support efficient insertion, deletion, and search operations, typically with a time complexity of $O(\log n)$ ³.
8. Range queries: The linked leaf nodes enable fast sequential access and efficient range query processing⁵.

These properties make B+ trees particularly well-suited for applications requiring fast data retrieval, especially in systems dealing with large datasets stored on disk¹⁵.

To determine why B+ trees outperform AVL trees for indexing large datasets, you would need to compare their structural properties, performance characteristics, and use-case suitability:

Key Comparison Factors

Aspect	B+ Tree	AVL Tree
Node Structure	Stores data only in leaf nodes; internal nodes hold keys/pointers ¹²	Stores data in all nodes ³

Tree Height	Shorter due to high fanout (100+ children per node) ²	Taller binary structure (max height = $2\log N$) ^{3 4}
Disk I/O Efficiency	Optimized for block storage with fewer disk accesses ^{1 2}	Designed for in-memory operations, less disk-friendly ⁴
Range Queries	Linked leaf nodes enable sequential scans without tree traversal ^{1 2}	Requires full tree traversal for range operations ³
Insertion Cost	Bulk-loading optimizations for large datasets ²	Frequent rotations increase overhead in write-heavy scenarios ³
Memory Usage	Smaller internal nodes (keys only) allow more keys per disk block ^{1 2}	Larger per-node metadata (balance factors) ³

Critical Information from Sources

1. B+ Tree Advantages

- Internal nodes contain only keys/pointers, enabling higher fanout and shorter trees^{1 2}
- Leaf nodes form a linked list for
- O(1)
- O(1) sequential access¹
- Better for systems with block-based storage (e.g., databases) due to reduced disk seeks²

2. AVL Tree Limitations

- Binary structure leads to taller trees (
- $\log N$
- $\log N$ vs B+ tree's
- $\log mN$
- \log
- m
- N , where
- m
- $m \approx 100$)³⁴
- No native support for efficient range scans³
- Rotation overhead becomes significant with large datasets⁴

3. Performance Tradeoffs

- B+ trees require ~60% fewer disk accesses for billion-record datasets¹³
- AVL trees become height-bound (
- $height \propto \log N$
- $height \propto \log N$) vs B+ trees' fanout-bound design²⁴

This comparison shows B+ trees are superior for disk-based large datasets due to their block-storage optimization, while AVL trees remain effective for smaller, in-memory datasets requiring strict balance

B-Trees usually have larger number of keys in single node and hence reducing the depth of the search, in record indexing the link traversal time is longer if the depth is more, hence for cache locality and making the tree wider than deeper, multiple keys are stored in array of a node which improves cache performance and quick lookup comparatively.



English ▾

Get Started

[Home](#) > [Programming and Data Structure](#) > [Binary Heap](#) > [Binary Search Tree](#)

Download Binary Search Tree MCQs...

Binary Search Tree MCQ Quiz - Objective Question with Answer for Binary Search Tree - Download Free PDF

Last updated on Feb 21, 2025

Binary Search Tree MCQs are crucial for assessing one's understanding of this data structure used for efficient searching and sorting operations. Binary search trees organize data in a hierarchical structure, enabling fast retrieval and insertion. Binary Search Tree MCQ evaluate learners' knowledge of tree traversal, node insertion and deletion, tree balancing, and operations on binary search trees. By answering Binary Search Tree MCQs, individuals can enhance their understanding of binary search tree properties, algorithms, and their applications in data processing, databases, and algorithm design.

Latest Binary Search Tree MCQ Objective Questions

Binary Search Tree Question 1:

Consider a completely skewed (left/right) binary search tree with n elements. What is the worst case time complexity of searching an element in this tree?

1. $O(n)$
2. $O(1)$
3. $O(\log n)$
4. $O(n \log n)$

Answer (Detailed Solution Below)

Option 1 : $O(n)$

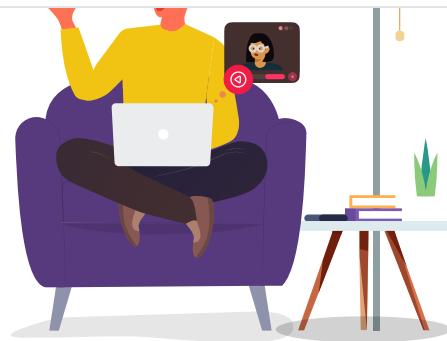


English ▾

[Get Started](#)

India's Super Teachers for all govt. exams Under One Roof

FREE Demo Classes Available*

[Enroll For Free Now](#)

Binary Search Tree Question 1 Detailed Solution

The correct answer is [O\(n\)](#).

Key Points

- **A binary search tree (BST) is a binary tree where each node has a value greater than all the values in its left subtree and less than all the values in its right subtree.**
- In a balanced BST, the time complexity for searching is $O(\log n)$ due to the tree's height being $\log(n)$.
- However, in a completely skewed (left or right) BST, the tree essentially behaves like a linked list.
- This means each node only has one child, and the tree's height becomes n (number of nodes).
- Therefore, the worst-case time complexity of searching an element in a completely skewed BST is $O(n)$ because you may need to traverse all the nodes.

Important Points

- In a balanced BST, operations like insertion, deletion, and search have average time complexities of $O(\log n)$.
- In a completely skewed BST, these operations degrade to $O(n)$ in the worst case.

Additional Information

- **Self-balancing BSTs like AVL trees and Red-Black trees maintain their height close to $\log(n)$, ensuring efficient operations.**
- **Understanding the structure and properties of different types of BSTs is crucial for optimizing search operations in various applications.**

FREE

India's #1 Learning Platform

Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation



English ▾

Get Started

[App Store](#)[Google Play](#)

Binary Search Tree Question 2:

_____ is a Self Balancing binary search tree, where the path from the root to the furthest leaf is no more than twice as long as the path from the root to nearest leaf.

1. Expression tree
2. Game tree
3. Red-Black tree
4. Threaded tree

Answer (Detailed Solution Below)

Option 3 : Red-Black tree

Binary Search Tree Question 2 Detailed Solution

The correct answer is **Red-Black tree**

Key Points

- **Red-Black Tree:** A Red-Black tree is a type of self-balancing binary search tree.
- It maintains balance by ensuring that the longest path from the root to a leaf is no more than twice as long as the shortest path from the root to any leaf.
- This balancing property ensures that the tree remains approximately balanced, allowing operations such as insertion, deletion, and lookup to be performed in **$O(\log n)$** time.

Additional Information

- **Expression Tree:** A binary tree used to represent expressions, not necessarily balanced.
- **Game Tree:** A tree representation of the possible moves in a game, not a self-balancing binary search tree.
- **Threaded Tree:** A binary tree in which null pointers are made to point to the in-order predecessor or successor, not specifically self-balancing.



English ▾

Get Started

India's #1 Learning Platform

Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes

Get Started for Free

Download on the
App StoreGET IT ON
Google Play

Binary Search Tree Question 3:

A binary search tree T contains n distinct elements. What is the time complexity of picking an element in T that is smaller than the maximum element in T?

1. $\Theta(1)$
2. $\Theta(n \log n)$
3. $\Theta(\log n)$
4. More than one of the above
5. None of the above

Answer (Detailed Solution Below)Option 1 : $\Theta(1)$

Binary Search Tree Question 3 Detailed Solution

Explanation:

- If an element in a binary search tree is smaller than any other element in the binary search tree then it is smaller than the maximum element.
- All the elements in the binary search tree is distinct.
- Compare only two elements in the binary search tree to find such elements.

FREE

India's #1 Learning Platform



Trusted by 6.9 Crore+ Students

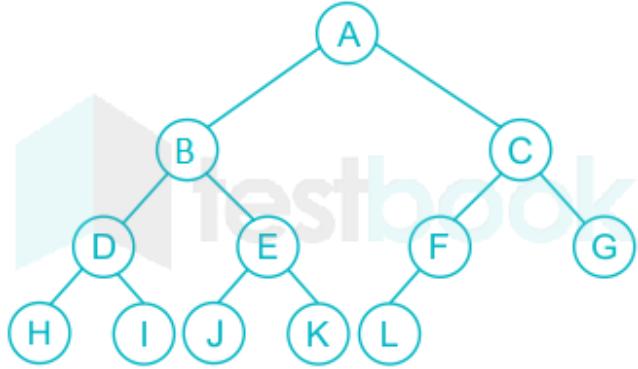
Start Complete Exam Preparation

 Daily Live MasterClasses Practice Question Bank Mock Tests & Quizzes

Get Started for Free

 Download on the App Store GET IT ON Google Play**Binary Search Tree Question 4:**

Which of the following statements about the following binary tree is FALSE?



1. Nodes 'J' and 'K' are siblings.

2. Node 'B' is the ancestor of node 'J'

3. It is a binary search tree.

4. It is a complete binary tree.

5. None of the above

Answer (Detailed Solution Below)



Binary Search Tree Question 4 Detailed Solution

Concept:

Binary search tree: A BST is a tree in which all the nodes follow the two properties.

- 1) The left sub tree of a node has a key less than or equal to its parent node's key.
- 2) The right sub tree of a node has a key greater than its parent's key.

Complete binary tree: A complete binary tree is a binary tree in which every level except possibly the last level is completely filled and all nodes are as left as possible.

Explanation:

In this tree, it is clearly showing that node J and K are siblings.

Also, given tree is satisfying the property of a complete binary tree.

But it is not following the property of binary search tree. So, option 3) it is a binary search tree is incorrect here.

FREE

India's #1 Learning Platform

 Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation



Daily Live
MasterClasses



Practice Question
Bank



Mock Tests &
Quizzes

Get Started for Free



Binary Search Tree Question 5:

Which of the followings are true for a complete binary tree ?

- A. It has always odd number of vertices.
- B. With i internal vertices, it has $i + 1$ leaves.
- C. With ℓ leaves it has $\ell - 1$ vertices.
- D. With $2n - 1$ vertices, it has n leaves.

Choose the correct answer from the options given below :

2. B, C Only

3. A, D Only

4. A, B, C, D

Answer (Detailed Solution Below)

Option 2 : B, C Only

Binary Search Tree Question 5 Detailed SolutionThe correct answer is **2) B, C Only.**
 **Key Points**

- **Statement B:** In a complete binary tree, with i internal vertices, it has $i + 1$ leaves. This is true because in a complete binary tree, every internal node has exactly two children. Therefore, if there are i internal nodes, there will be $i + 1$ leaves.
- **Statement C:** In a complete binary tree, with ℓ leaves it has $\ell - 1$ internal vertices. This is true because in a complete binary tree, the number of internal vertices is always one less than the number of leaves.

 **Additional Information**

- **Statement A:** It has always odd number of vertices. This is incorrect. A complete binary tree can have an odd or even number of vertices.
- **Statement D:** With $2n - 1$ vertices, it has n leaves. This is incorrect. The correct relationship between the number of vertices and leaves in a complete binary tree does not follow this formula.

FREE

India's #1 Learning Platform

Start Complete Exam Preparation

 Trusted by 6.9 Crore+ Students



English ▾

Get Started

Top Binary Search Tree MCQ Objective Questions

Binary Search Tree Question 6

Download Solution PDF

The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the postorder traversal sequence of the same tree?

1. 10, 20, 15, 23, 25, 35, 42, 39, 30
2. 15, 10, 25, 23, 20, 42, 35, 39, 30
3. 15, 20, 10, 23, 25, 42, 35, 39, 30
4. 15, 10, 23, 25, 20, 35, 42, 39, 30

Answer (Detailed Solution Below)

Option 4 : 15, 10, 23, 25, 20, 35, 42, 39, 30

Binary Search Tree Question 6 Detailed Solution

Download Solution PDF

The correct answer is "**option 4**".

CONCEPT:

A Binary Search Tree (BST) is also known as an **ordered tree** or **sorted binary tree**.

It is a binary tree with the following properties:

1. The **left sub-tree** of a node contains only nodes with key-value **lesser** than the node's key value.
2. The **right subtree** of a node contains only nodes with a key-value **greater** than the node's key value.

There are **three types** of traversal:

1. **In-order traversal:** In this traversal, the first left node will traverse, the root node then the right node will get traversed.



English ▾

[Get Started](#)

3. Post-order traversal: In this traversal, the first left node will traverse, the right node then the root node will get traversed.

The in-order traversal of the Binary search tree always returns key values in ascending order.

EXPLANATION:

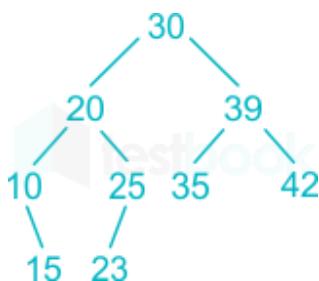
The **pre-order traversal** of given BST is:

30, 20, 10, 15, 25, 23, 39, 35, 42.

So, the **In-order traversal** of the BST is:

10, 15, 20, 23, 25, 30, 35, 39, 42.

The **Binary Search Tree** is:



So the post-order traversal of the tree is:

15, 10, 23, 25, 20, 35, 42, 39, 30

Hence, the correct answer is "option 4".

[Download Solution PDF](#)
[Share on Whatsapp](#)
FREE

India's #1 Learning Platform

[Start Complete Exam Preparation](#)


Trusted by 6.9 Crore+ Students


 Daily Live
MasterClasses

 Practice Question
Bank

 Mock Tests &
Quizzes

[Get Started for Free](#)

[Binary Search Tree Question 7](#)
[Download Solution PDF](#)

II. 5, 8, 9, 12, 10, 15, 25

III. 2, 7, 10, 8, 14, 16, 20

IV. 4, 6, 7, 9 18, 20, 25

1. I and IV only

2. II and III only

3. II and IV only

4. II only

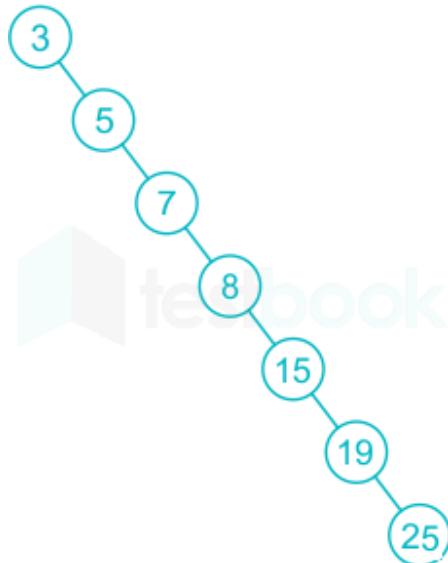
Answer (Detailed Solution Below)

Option 1 : I and IV only

Binary Search Tree Question 7 Detailed Solution

[Download Solution PDF](#)

Statement I: 3, 5, 7, 8, 15, 19, 25



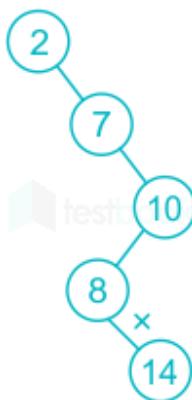
It doesn't violate Binary search tree property and hence it is the correct order of traversal.

Statement II: 5, 8, 9, 12, 10, 15, 25



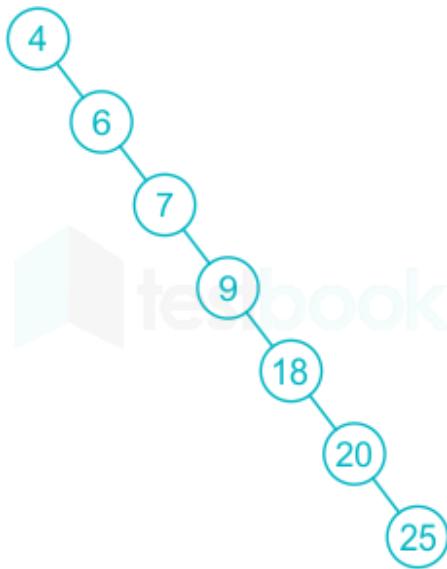
15 is left of 12 which violates binary search tree property.

Statement III: 2, 7, 10, 8, 14, 16, 20



14 is left of 10 which violates binary search tree property.

Statement IV: 4, 6, 7, 9, 18, 20, 25



It doesn't violate Binary search tree property and hence it is the correct order of traversal.



English ▾

Get Started

FREE

India's #1 Learning Platform



Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes

Get Started for Free

Download on the
App StoreGET IT ON
Google Play

Binary Search Tree Question 8

Download Solution PDF

What will be post order traversal of a binary Tree T, if preorder and inorder traversals of T are given by ABCDEF and BADCFE respectively?

1. BEFDCA
2. BFDECA
3. BCFDEA
4. BDFECA

Answer (Detailed Solution Below)

Option 4 : BDFECA

Binary Search Tree Question 8 Detailed Solution

Download Solution PDF

The correct answer is **option 4**.

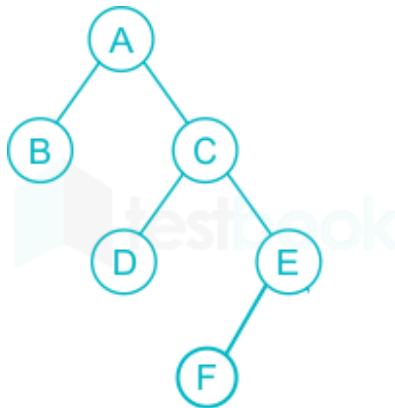
Concept:

The given data,

preorder = ABCDEF

Tree traversal			
Method Sequence	Inorder	Preorder	Postorder
	Left Sub-tree	Root	Left Sub-tree
	Root	Left Sub-tree	Right Sub-tree
	Right Sub-tree	Right Sub-tree	Root

The binary tree for the traversal is,



Post order for the above tree is,

BDFECA

Hence the correct answer is BDFECA.

[Download Solution PDF](#)
[Share on Whatsapp](#)
FREE

India's #1 Learning Platform

Start Complete Exam Preparation


 Daily Live MasterClasses

 Practice Question Bank

 Mock Tests & Quizzes

[Get Started for Free](#)




The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree?

1. 3
2. 4
3. 5
4. 6

Answer (Detailed Solution Below)

Option 1 : 3

Binary Search Tree Question 9 Detailed Solution

[Download Solution PDF](#)

The correct answer is **option 1**

Concept:

A binary search tree (BST) is a node-based binary tree data structure and it follows the following points

1. Left sub-tree nodes key value will exist only if lesser than the parent node key value.
2. Right sub-tree nodes key value will exist only if greater than the parent node key value.
3. Left sub-tree and Right sub-tree must be a Binary search tree.

Explanation:

Step 1: First 10 comes and now that is the **Root** node.

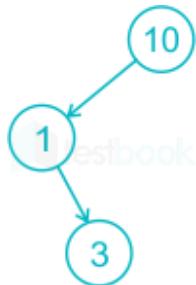
10

Step 2: Now 1 came and $1 < 10$ then insert Node 1 to the **Left** of Node 10.

(1)

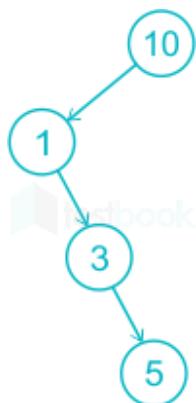
Step 3: Now 3 came and $3 < 10$ go to the **Left** of

Node 10 and check $3 > 1$ then insert Node 3 to the **Right** of Node 1.

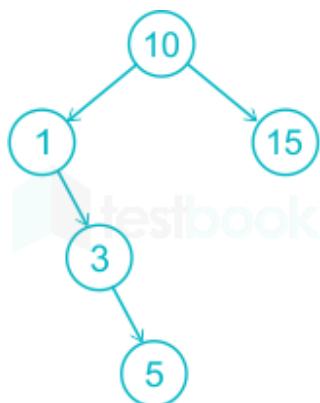


Step 4: Now 5 came and $5 < 10$ go to the **Left** of

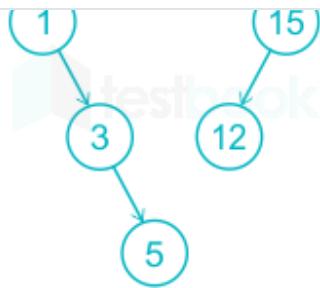
Node 10 and check $5 > 1$ go to the **Right** of Node 1 then check $5 > 3$ then insert Node 5 to the **Right** of Node 3.



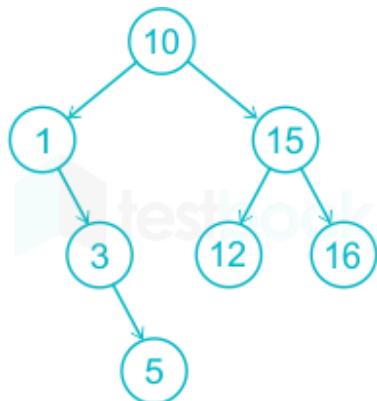
Step 5: Now 15 came and $15 > 10$ then insert Node 15 to the **Right** of Node 10.



Step 6: Now 12 came and $12 > 10$ go to the **Right** of Node 10 and check $15 > 12$ then insert Node 12 to the **Left** of Node 15.



Step 7: Now 16 came and $16 > 10$ go to the **Right** of 10 and check $16 > 15$ then insert 16 to the **Right** of Node 15.



After step 7, we can count the height of the tree as 3.

★ Important Points

Follow the longest path in the tree and count the edges that are height.

Tips To Learn:

Left sub-tree(key) < Node(key) < Right sub-tree(key)

Node(key): Parent node of **Left sub-tree** and **Right sub-tree**

[Download Solution PDF](#)
[Share on Whatsapp](#)
FREE

India's #1 Learning Platform

Start Complete Exam Preparation



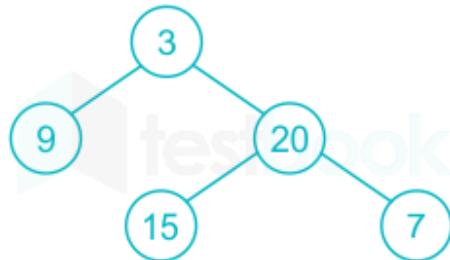
 Daily Live
MasterClasses

 Practice Question
Bank

 Mock Tests &
Quizzes

Binary Search Tree Question 10[Download Solution PDF](#)

Which of the following is a height of a given binary tree?

**Hint:**

The height of a binary tree is equal to the largest number of edges from the root to the most distant leaf node.

1. 1
2. 2
3. 3
4. 4

Answer (Detailed Solution Below)

Option 2 : 2

Binary Search Tree Question 10 Detailed Solution[Download Solution PDF](#)

The correct answer is **option 2**.

Concept:**Binary tree:**

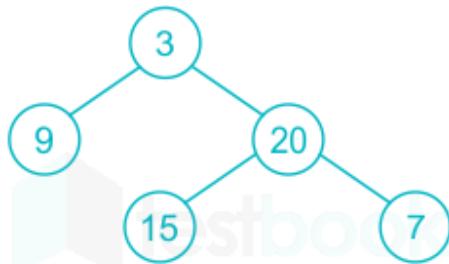
A binary tree is a tree in which no node can have more than two children. Every binary tree has parents, children, siblings, leaves, and internal nodes.



English ▾

[Get Started](#)

Learn mode.

Explanation:**Paths:**

- 3-9 → 1
 3-20-15 → 2
 3-20-7 → 2

Hence the correct answer is 2.[Download Solution PDF](#)[Share on Whatsapp](#)

FREE

India's #1 Learning Platform

Start Complete Exam Preparation

Trusted by 6.9 Crore+ Students



Daily Live MasterClasses



Practice Question Bank



Mock Tests & Quizzes

[Get Started for Free](#)
 [Download on the App Store](#)
 [GET IT ON Google Play](#)
Binary Search Tree Question 11[Download Solution PDF](#)

Suppose a binary search tree with 1000 distinct elements is also a complete binary tree. The tree is stored using the array representation of binary heap trees. Assuming that the array indices start with 0, the 3rd largest element of the tree is stored at index

Answer (Detailed Solution Below) 509

The correct answer is 509.

Concept:

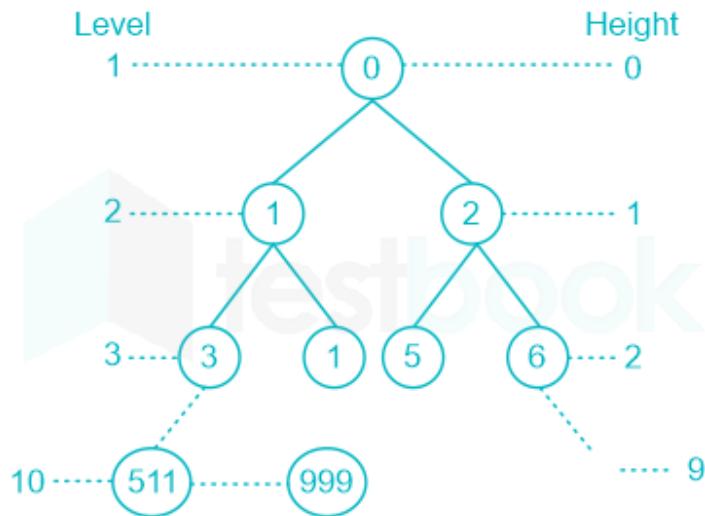
A binary search tree (BST) is a binary tree in which each node has a Comparable key (and an associated value) and the key in any node is greater than the keys in all nodes in that node's left subtree and less than the keys in all nodes in that node's right subtree.

Explanation:

The given data,

A binary search tree with 1000 different elements has been provided. The tree is also stored using the binary heap tree's array representation. The indices of an array begin with 0.

The index number of the first node at each level may be determined using $(2^h - 1)$, where "h" is the tree's height. Also, the number of nodes at each level may be calculated using $(2^n - 1)$, where n is the number of levels.



At 10th level number of nodes = $2^{10-1} = 512$.

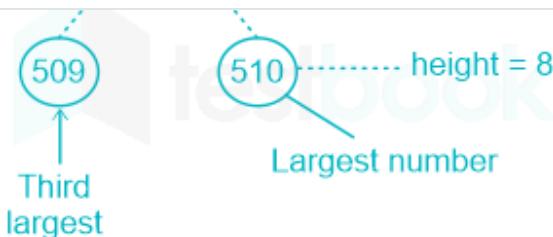
At height 9, the index number of the first node = $2^9 - 1 = 511$. Since the total number of nodes is 1000, we need to check the upper level. Because the rightmost number in the binary search tree is maximum.

At 9th level, no. of nodes = $2^{9-1} = 256$. At height 8, index no. of first node = $2^8 - 1 = 255$.

The index number of the last node in 9 th level, = $255 \times 2 = 510$



English ▾

[Get Started](#)

So the third largest value is stored at = 509

Hence the correct answer is 509.

[Download Solution PDF](#)[Share on Whatsapp](#)

FREE

India's #1 Learning Platform

Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes[Get Started for Free](#)

Download on the
App Store

GET IT ON
Google Play

Binary Search Tree Question 12

[Download Solution PDF](#)

The program written for binary search, calculates the midpoint of the span as $mid := (\text{Low} + \text{High})/2$. The program works well if the number of elements in the list is small (about 32,000) but it behaves abnormally when the number of elements is large. This can be avoided by performing the calculation as:

1. $mid := (\text{High} - \text{Low})/2 + \text{Low}$

2. $mid := (\text{High} - \text{Low} + 1)/2$

3. $mid := (\text{High} - \text{Low})/2$



English ▾

Get Started

Answer (Detailed Solution Below)Option 1 : $\text{mid} := (\text{High} - \text{Low})/2 + \text{Low}$ **Binary Search Tree Question 12 Detailed Solution**

Download Solution PDF

The correct answer is **option 1**. **Key Points**

- In a general scenario, binary search **mid-value** computed with, **mid=(low+high)/2**.
- However, with a **vast list** of elements, "high" would be a very **large value**. As a result, it's possible that it's **beyond the Integer limit**. It's known as an **integer overflow**.
- To stop this Integer overflow, the 'mid' value can also be determined using, **mid = (High - Low)/2 + Low**
- **Integer overflow** is never an issue with this form.

Explanation $\text{mid} := (\text{High} - \text{Low})/2 + \text{Low}$ $\text{mid} := \text{High}/2 - \text{Low}/2 + \text{low}$ $\text{mid} := (\text{High} + \text{Low})/2$ **Alternate Method**

- Option D is removed because it is the same as the incorrect option.
- Taking into account The low index is 10, while the high index is 15.
- Option B returns a mid-index of 3 that is not even in the sub-array index.
- Choice C returns a mid-index of 2 that isn't even in the sub-array index.
- Option A is the best solution.

Hence the correct answer is $\text{mid} := (\text{High} - \text{Low})/2 + \text{Low}$.

Download Solution PDF

Share on Whatsapp

FREE

India's #1 Learning Platform

Start Complete Exam Preparation

Trusted by 6.9 Crore+ Students

Daily Live MasterClasses

Practice Question Bank

Mock Tests & Quizzes

Binary Search Tree Question 13[Download Solution PDF](#)

A binary search tree is constructed by inserting the following numbers in order:

60, 25, 72, 15, 30, 68, 101, 13, 18, 47, 70, 34

The number of nodes in the left subtree is

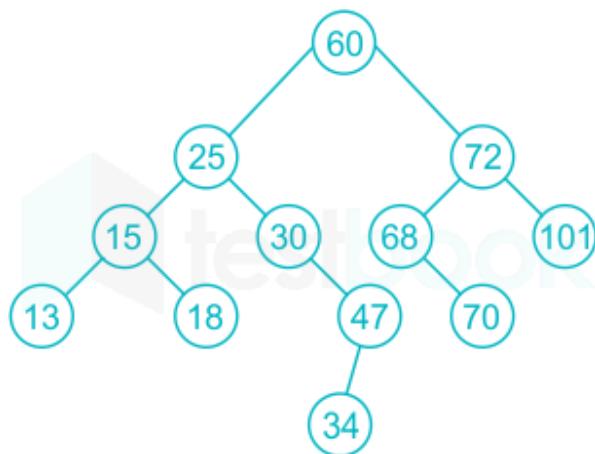
1. 7
2. 6
3. 3
4. 5

Answer (Detailed Solution Below)

Option 1 : 7

Binary Search Tree Question 13 Detailed Solution[Download Solution PDF](#)

Binary Search Tree (BST):





English ▾

[Get Started](#)

Tips and Tricks:

Sorted the nodes in BST: 13 15 18 25 30 34 47 **60** 68 70 72 101

Count the nodes to the left of the root node (60).

[Download Solution PDF](#)
[Share on Whatsapp](#)

FREE

India's #1 Learning Platform

 Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

 Daily Live MasterClasses

 Practice Question Bank

 Mock Tests & Quizzes

[Get Started for Free](#)
 Download on the App Store

 GET IT ON Google Play

Binary Search Tree Question 14

[Download Solution PDF](#)

Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the reversal ordering on natural numbers i.e. 9 is assumed to be smallest and 0 is assumed to be largest. The in-order traversal of the resultant binary search tree is

1. 9, 8, 6, 4, 2, 3, 0, 1, 5, 7
2. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
3. 0, 2, 4, 3, 1, 6, 5, 9, 8, 7
4. 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

Answer (Detailed Solution Below)

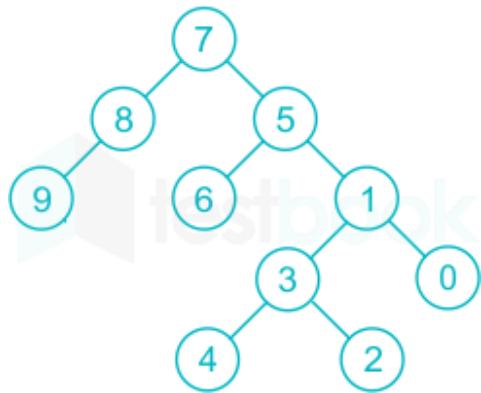


English ▾

[Get Started](#)**Binary Search Tree Question 14 Detailed Solution**[Download Solution PDF](#)

Insert 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 in empty binary search tree by using reversal ordering

Binary search tree:



Inorder traversal = 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

Tips and Trick:

In-order of the binary tree is always sorted in ascending order but binary search tree uses the reversal ordering on natural numbers

Therefore it is sorted in descending order:

[Download Solution PDF](#)[Share on Whatsapp](#)

FREE

India's #1 Learning Platform

Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes[Get Started for Free](#)[Binary Search Tree Question 15](#)[Download Solution PDF](#)

1. $\Theta(\log n)$ for both insertion and deletion
2. $\Theta(n)$ for both insertion and deletion
3. $\Theta(n)$ for insertion and $\Theta(\log n)$ for deletion
4. $\Theta(\log n)$ for insertion and $\Theta(n)$ for deletion

Answer (Detailed Solution Below)

Option 2 : $\Theta(n)$ for both insertion and deletion

Binary Search Tree Question 15 Detailed Solution[Download Solution PDF](#)**Concepts:**

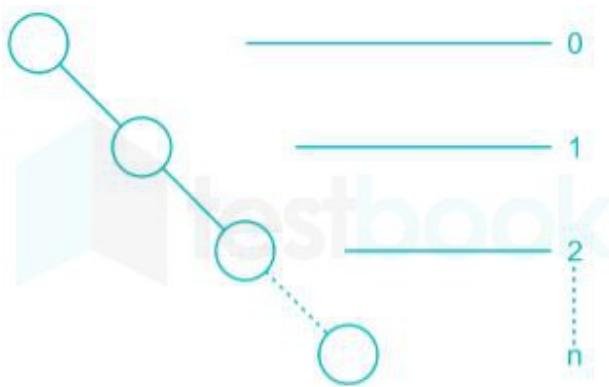
Minimum height of the tree is when all the levels of the binary search tree (BST) are completely filled.

Maximum height of the BST is the worst case when nodes are in skewed manner.

Formula:

Minimum height of the BST with n nodes is $\lceil \log_2(n + 1) \rceil - 1$

The maximum height of the BST with n nodes is $n - 1$.

BST with a maximum height:**Insertion:**



English ▾

Get Started

Deletion:

Traverser the BST to the maximum height

Worst-case time complexity of = $\theta(n - 1) \equiv \theta(n)$ [Download Solution PDF](#)[Share on Whatsapp](#)

FREE

India's #1 Learning Platform

Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes[Get Started for Free](#)

Chapter 12: Binary Search Trees

A **binary search tree** is a binary tree with a special property called the **BST-property**, which is given as follows:

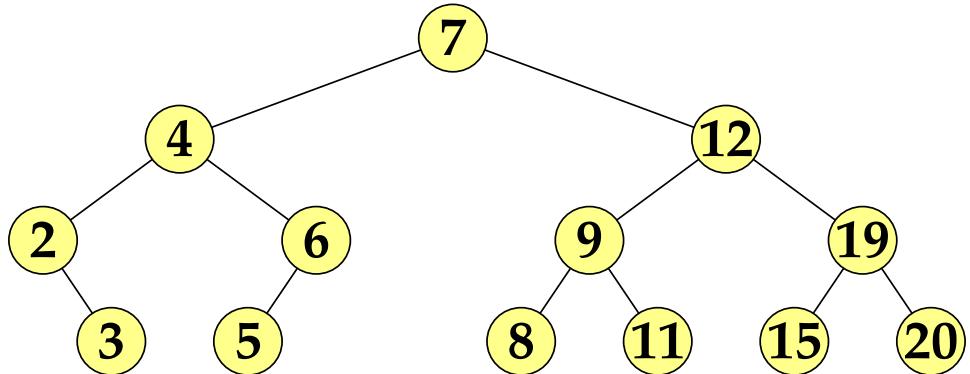
- * *For all nodes x and y , if y belongs to the left subtree of x , then the key at y is less than the key at x , and if y belongs to the right subtree of x , then the key at y is greater than the key at x .*

We will assume that the keys of a BST are pairwise distinct.

Each node has the following attributes:

- p , $left$, and $right$, which are pointers to the parent, the left child, and the right child, respectively, and
- key , which is key stored at the node.

An example



Traversal of the Nodes in a BST

By “traversal” we mean visiting all the nodes in a graph. Traversal strategies can be specified by the ordering of the three objects to visit: the current node, the left subtree, and the right subtree. We assume the the left subtree always comes before the right subtree. Then there are three strategies.

1. **Inorder.** The ordering is: the left subtree, the current node, the right subtree.
2. **Preorder.** The ordering is: the current node, the left subtree, the right subtree.
3. **Postorder.** The ordering is: the left subtree, the right subtree, the current node.

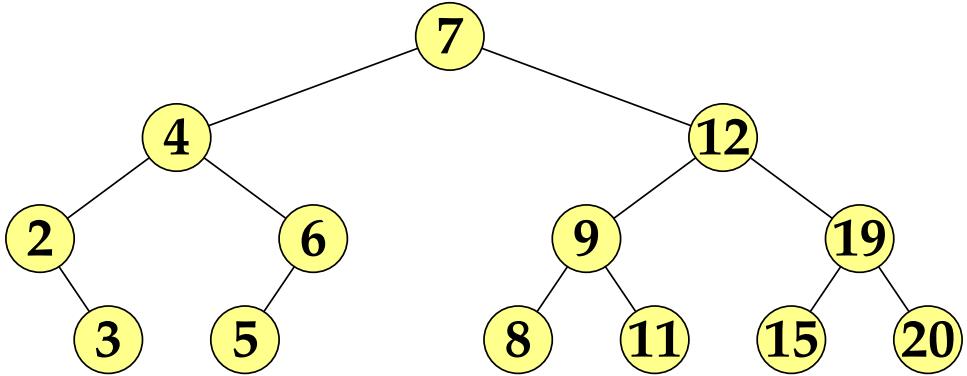
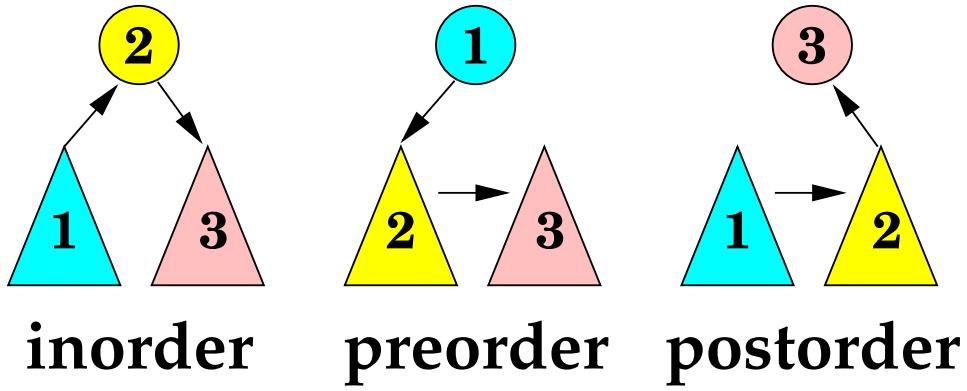
Inorder Traversal Pseudocode

This recursive algorithm takes as the input a pointer to a tree and executed inorder traversal on the tree. While doing traversal it prints out the key of each node that is visited.

Inorder-Walk(x)

- 1: **if** $x = \text{nil}$ **then return**
- 2: **Inorder-Walk($\text{left}[x]$)**
- 3: Print $\text{key}[x]$
- 4: **Inorder-Walk($\text{right}[x]$)**

We can write a similar pseudocode for preorder and postorder.



*What is the outcome of
inorder traversal on this BST?
How about postorder traversal
and preorder traversal?*

Inorder traversal gives: 2, 3,
4, 5, 6, 7, 8 , 9, 11, 12, 15,
19, 20.

Preorder traversal gives: 7, 4,
2, 3, 6, 5, 12, 9, 8, 11, 19,
15, 20.

Postorder traversal gives: 3,
2, 5, 6, 4, 8, 11, 9, 15, 20,
19, 12, 7.

So, inorder travel on a BST
finds the keys in
nondecreasing order!

Operations on BST

1. Searching for a key

We assume that a key and the subtree in which the key is searched for are given as an input. We'll take the full advantage of the BST-property.

Suppose we are at a node. If the node has the key that is being searched for, then the search is over. Otherwise, the key at the current node is either strictly smaller than the key that is searched for or strictly greater than the key that is searched for. If the former is the case, then by the BST property, all the keys in the left subtree are strictly less than the key that is searched for. That means that we do not need to search in the left subtree. Thus, we will examine only the right subtree. If the latter is the case, by symmetry we will examine only the right subtree.

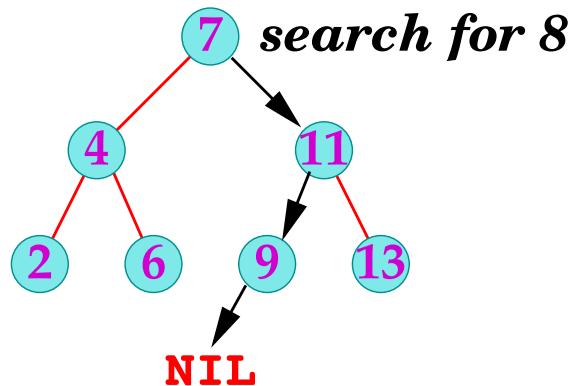
Algorithm

Here k is the key that is searched for and x is the start node.

BST-Search(x, k)

```
1:  $y \leftarrow x$ 
2: while  $y \neq \text{nil}$  do
3:   if  $\text{key}[y] = k$  then return  $y$ 
4:   else if  $\text{key}[y] < k$  then  $y \leftarrow \text{right}[y]$ 
5:   else  $y \leftarrow \text{left}[y]$ 
6: return (“NOT FOUND”)
```

An Example



*What is the running time of
search?*

2. The Maximum and the Minimum

To find the minimum identify the leftmost node, i.e. the farthest node you can reach by following only left branches.

To find the maximum identify the rightmost node, i.e. the farthest node you can reach by following only right branches.

BST-Minimum(x)

- 1: **if** $x = \text{nil}$ **then return** (“Empty Tree”)
- 2: $y \leftarrow x$
- 3: **while** $\text{left}[y] \neq \text{nil}$ **do** $y \leftarrow \text{left}[y]$
- 4: **return** ($\text{key}[y]$)

BST-Maximum(x)

- 1: **if** $x = \text{nil}$ **then return** (“Empty Tree”)
- 2: $y \leftarrow x$
- 3: **while** $\text{right}[y] \neq \text{nil}$ **do** $y \leftarrow \text{right}[y]$
- 4: **return** ($\text{key}[y]$)

3. Insertion

Suppose that we need to insert a node z such that $k = \text{key}[z]$. Using binary search we find a nil such that replacing it by z does not break the BST-property.

BST-Insert(x, z, k)

```
1: if  $x = \text{nil}$  then return “Error”
2:  $y \leftarrow x$ 
3: while true do {
4:   if  $\text{key}[y] < k$ 
5:     then  $z \leftarrow \text{left}[y]$ 
6:     else  $z \leftarrow \text{right}[y]$ 
7:   if  $z = \text{nil}$  break
8: }
9: if  $\text{key}[y] > k$  then  $\text{left}[y] \leftarrow z$ 
10: else  $\text{right}[p[y]] \leftarrow z$ 
```

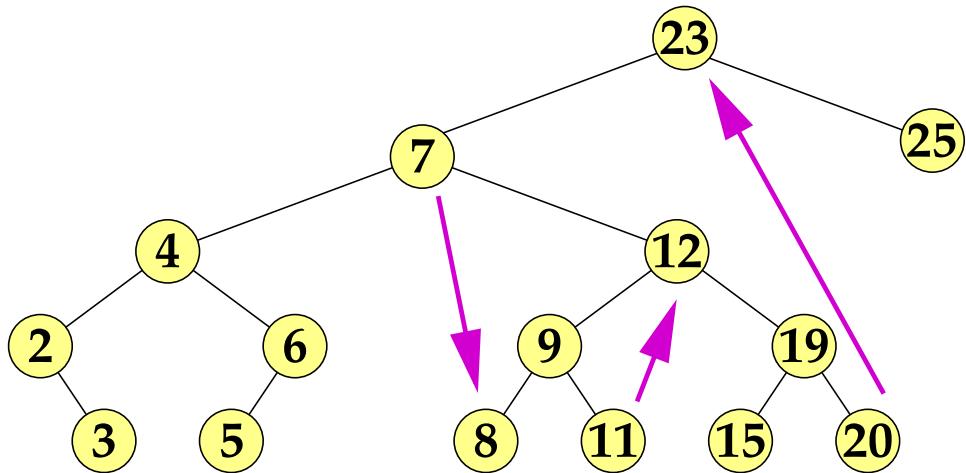
4. The Successor and The Predecessor

The successor (respectively, the predecessor) of a key k in a search tree is the smallest (respectively, the largest) key that belongs to the tree and that is strictly greater than (respectively, less than) k .

The idea for finding the successor of a given node x .

- If x has the right child, then the successor is the minimum in the right subtree of x .
- Otherwise, the successor is the parent of the farthest node that can be reached from x by following only right branches backward.

An Example



Algorithm

BST-Successor(x)

```
1: if  $right[x] \neq \text{nil}$  then
2: {    $y \leftarrow right[x]$ 
3:   while  $left[y] \neq \text{nil}$  do  $y \leftarrow left[y]$ 
4:   return ( $y$ ) }
5: else
6: {    $y \leftarrow x$ 
7:   while  $right[p[x]] = x$  do  $y \leftarrow p[x]$ 
8:   if  $p[x] \neq \text{nil}$  then return ( $p[x]$ )
9:   else return ("NO SUCCESSOR") }
```

The predecessor can be found similarly with the roles of left and right exchanged and with the roles of maximum and minimum exchanged.

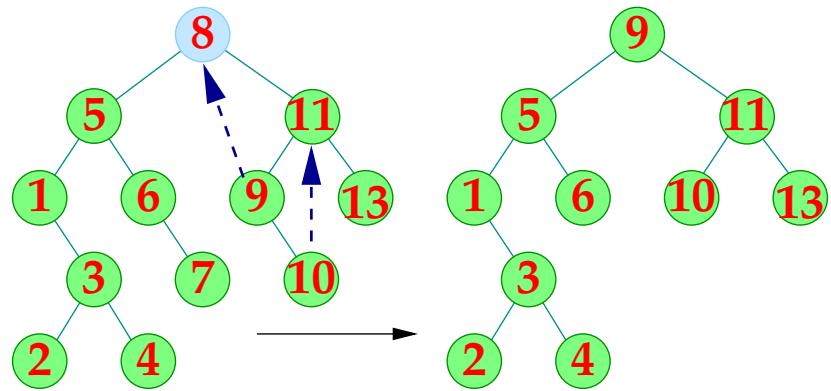
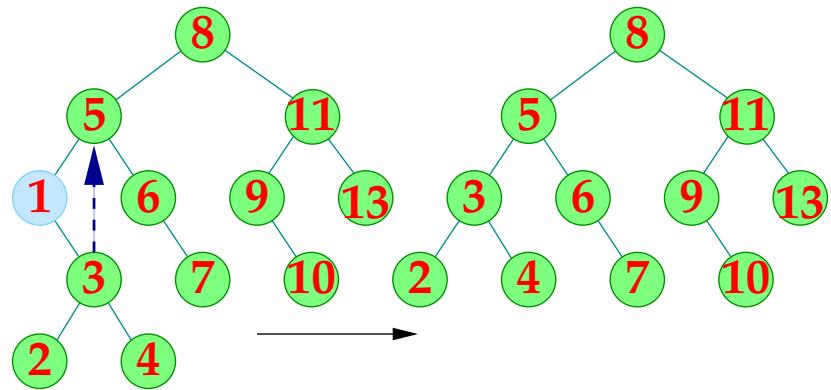
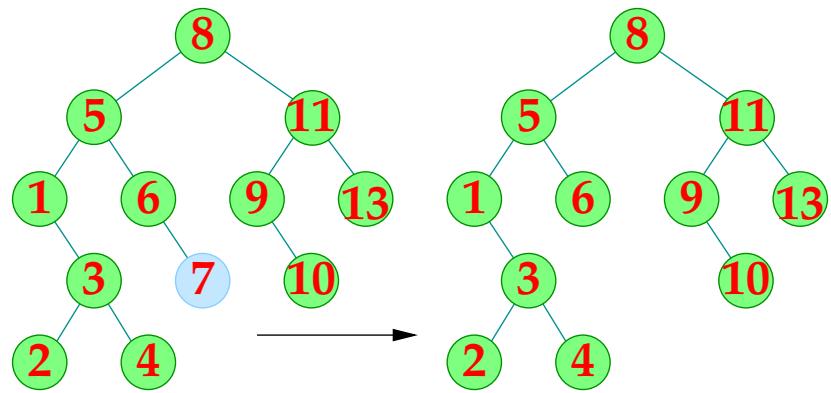
For which node is the successor undefined?

What is the running time of the successor algorithm?

5. Deletion

Suppose we want to delete a node z .

1. If z has no children, then we will just replace z by **nil**.
2. If z has only one child, then we will promote the unique child to z 's place.
3. If z has two children, then we will identify z 's successor. Call it y . The successor y either is a leaf or has only the right child. Promote y to z 's place. Treat the loss of y using one of the above two solutions.



Algorithm

This algorithm deletes z from BST T .

BST-Delete(T, z)

- 1: **if** $left[z] = \text{nil}$ **or** $right[z] = \text{nil}$
- 2: **then** $y \leftarrow z$
- 3: **else** $y \leftarrow \text{BST-Successor}(z)$
- 4: $\triangleright y$ is the node that's actually removed.
- 5: \triangleright Here y does not have two children.
- 6: **if** $left[y] \neq \text{nil}$
- 7: **then** $x \leftarrow left[y]$
- 8: **else** $x \leftarrow right[y]$
- 9: $\triangleright x$ is the node that's moving to y 's position.
- 10: **if** $x \neq \text{nil}$ **then** $p[x] \leftarrow p[y]$
- 11: $\triangleright p[x]$ is reset If x isn't NIL.
- 12: \triangleright Resetting is unnecessary if x is NIL.

Algorithm (cont'd)

```
13: if  $p[y] = \text{nil}$  then  $\text{root}[T] \leftarrow x$ 
14:  $\triangleright$  If  $y$  is the root, then  $x$  becomes the root.
15:  $\triangleright$  Otherwise, do the following.
16: else if  $y = \text{left}[p[y]]$ 
17:   then  $\text{left}[p[y]] \leftarrow x$ 
18:  $\triangleright$  If  $y$  is the left child of its parent, then
19:  $\triangleright$  Set the parent's left child to  $x$ .
20:   else  $\text{right}[p[y]] \leftarrow x$ 
21:  $\triangleright$  If  $y$  is the right child of its parent, then
22:  $\triangleright$  Set the parent's right child to  $x$ .
23: if  $y \neq z$  then
24:   {  $\text{key}[z] \leftarrow \text{key}[y]$ 
25:     Move other data from  $y$  to  $z$  }
27: return ( $y$ )
```

Summary of Efficiency Analysis

Theorem A On a binary search tree of height h , Search, Minimum, Maximum, Successor, Predecessor, Insert, and Delete can be made to run in $O(h)$ time.

Randomly built BST

Suppose that we insert n distinct keys into an initially empty tree. Assuming that the $n!$ permutations are equally likely to occur, what is the average height of the tree?

To study this question we consider the process of constructing a tree T by **inserting in order randomly selected n distinct keys** to an initially empty tree. Here the actually values of the keys do not matter. What matters is the position of the inserted key in the n keys.

The Process of Construction

So, we will view the process as follows:

A key x from the keys is selected uniformly at random and is inserted to the tree. Then all the other keys are inserted. Here all the keys greater than x go into the right subtree of x and all the keys smaller than x go into the left subtree. Thus, the height of the tree thus constructed is one plus the larger of the height of the left subtree and the height of the right subtree.

Random Variables

n = number of keys

X_n = height of the tree of n keys

$$Y_n = 2^{X_n}.$$

We want an upper bound on $E[Y_n]$.

For $n \geq 2$, we have

$$E[Y_n] = \frac{1}{n} \left(\sum_{i=1}^n 2E[\max\{Y_{i-1}, Y_{n-i}\}] \right).$$

$$\begin{aligned} E[\max\{Y_{i-1}, Y_{n-i}\}] &\leq E[Y_{i-1} + Y_{n-i}] \\ &\leq E[Y_{i-1}] + E[Y_{n-i}] \end{aligned}$$

Collecting terms:

$$E[Y_n] \leq \frac{4}{n} \sum_{i=1}^{n-1} E[Y_i].$$

Analysis

We claim that for all $n \geq 1$ $E[Y_n] \leq \frac{1}{4} \binom{n+3}{3}$.
We prove this by induction on n .

Base case: $E[Y_1] = 2^0 = 1$.

Induction step: We have

$$E[Y_n] \leq \frac{4}{n} \sum_{i=1}^{n-1} E[Y_i]$$

Using the fact that

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$$

$$E[Y_n] \leq \frac{4}{n} \cdot \frac{1}{4} \cdot \binom{n+3}{4}$$

$$E[Y_n] \leq \frac{1}{4} \cdot \binom{n+3}{3}$$

Jensen's inequality

A function f is **convex** if for all x and y , $x < y$, and for all λ , $0 \leq \lambda \leq 1$,

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

Jensen's inequality states that for all random variables X and for all convex function f

$$f(E[X]) \leq E[f(X)].$$

Let this X be X_n and $f(x) = 2^x$. Then $E[f(X)] = E[Y_n]$. So, we have

$$2^{E[X_n]} \leq \frac{1}{4} \binom{n+3}{3}.$$

The right-hand side is at most $(n+3)^3$. By taking the log of both sides, we have

$$E[X_n] = O(\log n).$$

Thus the average height of a randomly build BST is $O(\log n)$.

Unlimited PowerPoint Templates

Eye-catching presentations for every need



« DS Menu

Comparison of Trees

Binary Search Trees (BSTs), AVL Trees, Red-Black Trees, B-Trees, and B+ Trees are all types of self-balancing tree data structures that are used to store, retrieve, modify, and delete data in an efficient manner. Each has its unique characteristics, advantages, and disadvantages. Here's a comparison

Feature	BST	AVL Tree	Red-Black Tree	B-Tree	B+ Tree
Structure	Binary tree with ordered keys	Balanced binary tree with rotation operations	Self-balancing binary tree with red and black nodes	Multi-level tree with ordered keys at each level	Multi-level tree with data stored only at leaf nodes
Balancing	No automatic balancing	Guaranteed height balance ($\log n$)	Probabilistically balanced with $O(\log n)$ height	Guaranteed height balance ($\log m N$)	No internal node data, balanced using pointer adjustments
Space complexity	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Good for	Simple searches and insertions, basic data structures	Efficient searches and insertions with guarantees on worst-case performance	Efficient searches and insertions with probabilistically good performance	Large datasets, efficient search and range queries	Efficient storage and search of large datasets with frequent updates
Disadvantages	Unbalanced trees can lead to slow performance	More complex than BST, overhead of balance operations	Slightly more complex than AVL tree, no guarantee of balance	Not suitable for small datasets, higher space complexity	Less flexibility than B-tree, data not stored in internal nodes

Comparison of trees based on the Time complexity

Tree Types	Average Case			Worst Case		
	Insert	Delete	Search	Insert	Delete	Search
Binary Search Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log 2 n)$					
B - Tree	$O(\log n)$					
Red - Black Tree	$O(\log n)$					
Splay Tree	$O(\log 2 n)$					



Jobs.Dell.com

Learn More



- AVL Tree: Choose it if you need guaranteed good performance for searches and insertions even on unbalanced data.
- Red-Black Tree: Opt for it if you want probabilistic good performance and a simpler implementation than AVL trees.
- B-Tree: Select it when dealing with large datasets on disk or other secondary storage due to its efficient search and range queries.
- B+ Tree: Use it for scenarios similar to B-trees, but when data size matters, as it stores data only in leaf nodes, optimizing space usage.

Next Topic : [**Introduction to Graph**](#)

ABOUT

Study Glance provides Tutorials , Power point Presentations(ppts), Lecture Notes, Important & previously asked questions, Objective Type questions, Laboratory programs and we provide Syllabus of various subjects.

CATEGORIES

Tutorials	Questions
PPTs	Lab Programs
Lecture Notes	Syllabus

Copyright © 2020 All Rights Reserved by StudyGlance.

140552



Contiguous and Non-Contiguous memory allocation in Operating System

Memory management

Memory is central to the operation of a computer system. It consists of a large array of words or bytes each with its own address. In uniprogramming system, main memory has two parts one for the operating system and another part is for the program currently being executed. In the multiprogramming system, the memory part of the user is further divided into accommodate processes. The task of the subdivision is carried out by the operating system and is known as memory management.

Memory management techniques

The memory management techniques are divided into two parts...

1. Uniprogramming:

In the uniprogramming technique, the RAM is divided into two parts **one part is for the resigning the operating system and other portion is for the user process**. Here the border register is used which contains the last address of the operating system parts. The operating system will compare the user data addresses with the fence register and if it is different that means the user is not entering in the OS area. Border register is also called boundary register and is used to prevent a user from entering in the operating system area. Here the CPU utilization is very poor and hence multiprogramming is used.

2. Multiprogramming:

In the multiprogramming, the multiple users can share the memory simultaneously. By multiprogramming we mean there will be more than one process in the main memory and if the running process wants to wait for an event like I/O then instead of sitting idle CPU will make a context switch and will pick another process.

- a. **Contiguous memory allocation**
- b. **Non-contiguous memory allocation**

a) Contiguous memory allocation

The Contiguous memory allocation is one of the methods of memory allocation. In contiguous memory allocation, when a process requests for the memory, a **single contiguous section of memory blocks** is assigned to the process according to its requirement.

- All the available memory space remains together in one place. *It means freely available memory partitions are not expanded here and there across the whole memory space.*
- Both the operating system and the user must reside in the main memory. *The main memory is divided into two portions one portion is for the operating and other is for the user program.*
- When any user process request for the memory a single section of the contiguous memory block is given to that process according to its need. *We can achieve contiguous memory allocation by dividing memory into the fixed-sized partition.*

A single process is allocated in that fixed sized single partition. But this will increase the degree of multiprogramming means more than one process in the main memory that bounds the number of fixed

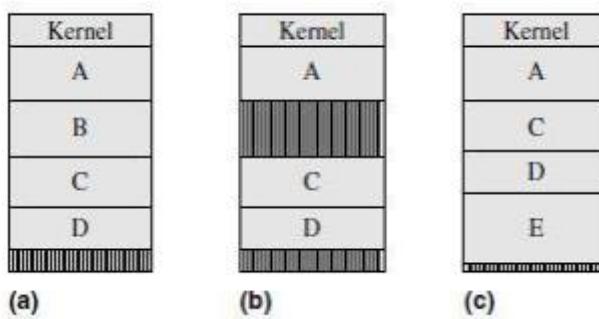
partition done in memory. Internal fragmentation increases because of the contiguous memory allocation.

Contiguous Memory Allocation

Example

Processes A, B, C, and D are in memory in Figure. Two free areas of memory exist after B terminates; however, neither of them is large enough to accommodate another process. The kernel performs compaction to create a single free memory area and initiates process E in this area. It involves moving processes C and D in memory during their execution.

Memory compaction involves *dynamic relocation*, which is not feasible without a relocation register. In computers not having a relocation register, the kernel must resort to reuse of free memory areas. However, this approach incurs delays in initiation of processes when large free memory areas do not exist, e.g., initiation of process E would be delayed in Example 4.8 even though the total free memory in the system exceeds the size of E.



Swapping

The kernel swaps out a process that is not in the *running* state by writing out its code and data space to a *swapping area* on the disk. The swapped out process is brought back into memory before it is due for another burst of CPU time. A basic issue in swapping is whether a swapped in process should be loaded back into the same memory area that it occupied before it was swapped out. If so, it's swapping in depends on swapping out of some other process that may have been allocated that memory area in the meanwhile. It would be useful to be able to place the swapped-in process elsewhere in memory; however, it would amount to dynamic relocation of the process to a new memory area. As mentioned earlier, only computer systems that provide a relocation register can achieve it.

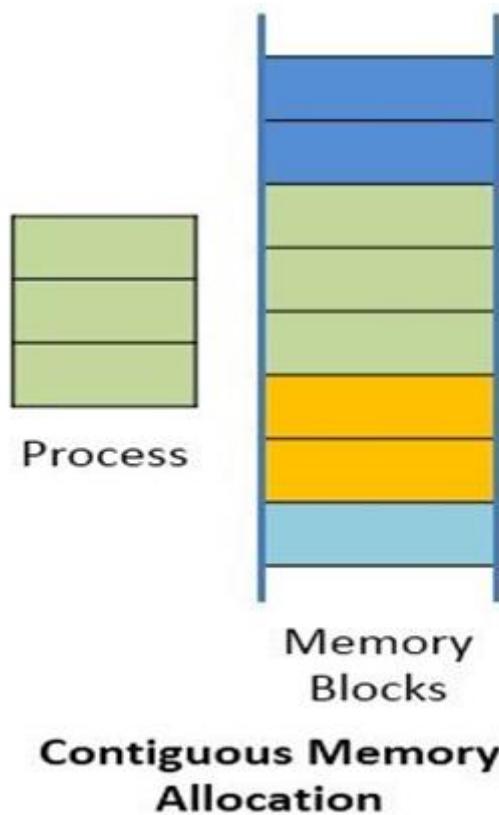
Swapping is a technique of temporarily removing inactive programs from the memory of the system. A process can be swapped temporarily out of the memory to a backing store and then brought back in to the memory for continuing the execution. This process is called swapping. Eg:-In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution.

This process is also called as Roll out and Roll in. physical address is computed during run time. Swapping requires backing store and it should be large enough to accommodate the copies of all memory images. The system maintains a ready queue consisting of all the processes whose memory images are on the backing store or in memory that are ready to run. Swapping is constant by other factors:

To swap a process, it should be completely idle. A process may be waiting for an i/o operation. If the i/o is asynchronously accessing the user memory for i/o buffers, then the process cannot be swapped. Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously. This depends upon address binding.

If the binding is done at load time, then the process is moved to same memory location. If the binding is done at run time, then the process is moved to different memory location. This is because the



→ Fixed sized partition

In the fixed sized partition the system divides memory into fixed size partition (may or may not be of the same size) here entire partition is allowed to a process and if there is some wastage inside the partition is allocated to a process and if there is some wastage inside the partition then it is called internal fragmentation.

Advantage: Management or book keeping is easy.

Disadvantage: Internal fragmentation

→ **Variable size partition**

In the variable size partition, the memory is treated as one unit and space allocated to a process is exactly the same as required and the leftover space can be reused again.

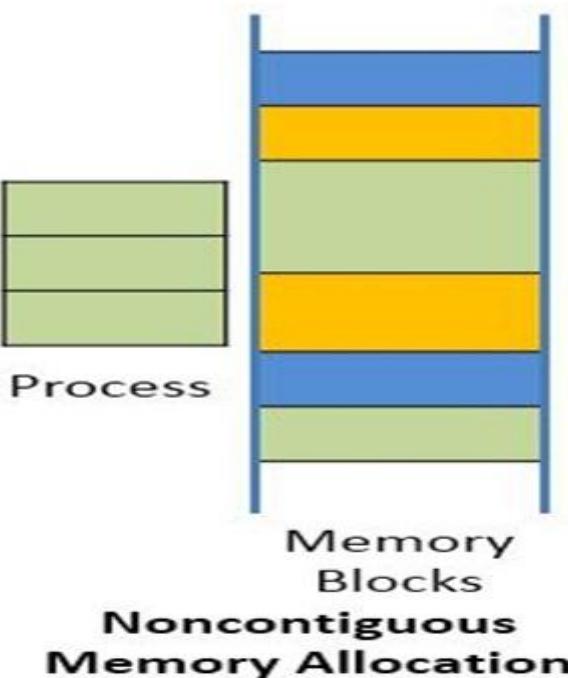
Advantage: There is no internal fragmentation.

Disadvantage: Management is very difficult as memory is becoming purely fragmented after some time.

b) Non-contiguous memory allocation

The Non-contiguous memory allocation allows a process to **acquire the several memory blocks at the different location in the memory** according to its requirement. The non-contiguous memory allocation also **reduces the memory wastage** caused due to internal and external fragmentation. As it utilizes the memory holes, created during internal and external fragmentation.

- The available free memory space are scattered here and there and all the free memory space is not at one place. So this is time-consuming.
- A process will acquire the memory space but it is not at one place it is at the different locations according to the process requirement.
- It reduces the wastage of memory which leads to internal and external fragmentation. This utilizes all the free memory space which is created by a different process.



Non-contiguous memory allocation is of different types,

1. Paging
2. Segmentation
3. Segmentation with paging

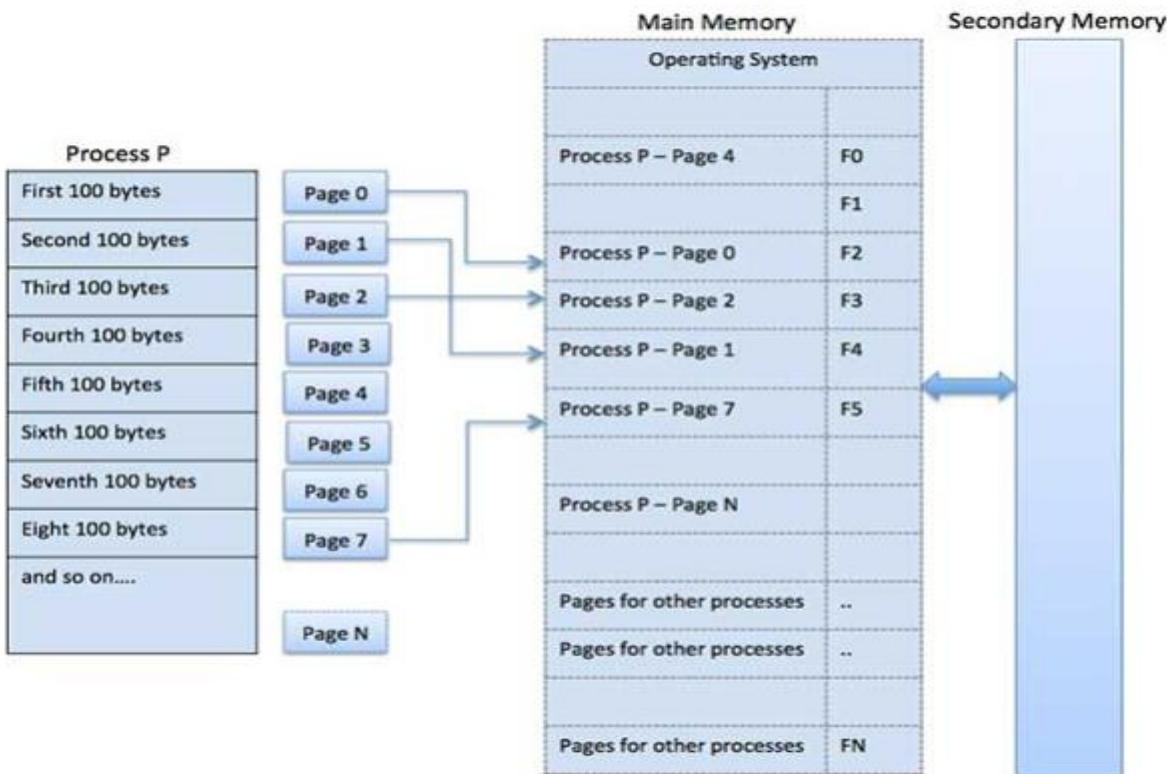
(I) PAGING

A non-contiguous policy with a fixed size partition is called paging. A computer can address more memory than the amount of physically installed on the system. *This extra memory is actually called virtual memory.* Paging technique is very important in implementing virtual memory.

Secondary memory is divided into equal size partition (fixed) called pages. Every process will have a separate page table. The entries in the page table are the number of pages a process. At each entry either we have an invalid pointer which means the page is not in main memory or we will get the corresponding frame number.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

When the frame number is combined with instruction of set D than we will get the corresponding physical address. Size of a page table is generally very large so cannot be accommodated inside the PCB, therefore, PCB contains a register value PTBR(page table base register) which leads to the page table.



Address Translation

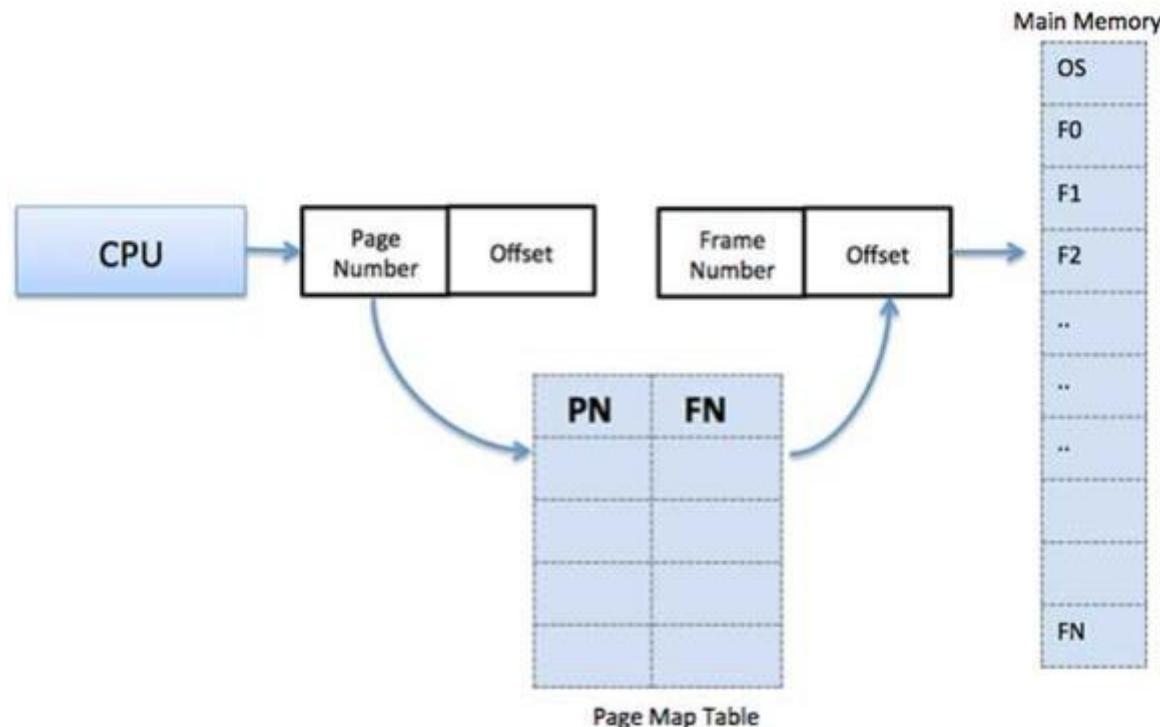
Page address is called **logical address** and represented by **page number** and the **offset**.

Logical address=Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a pages of a process to a frame in physical memory



When the system allocates a frame to any page, it translates this logical address into a physical address and creates entry into the page table to be used throughout execution of the program. When a process is to be executed, its corresponding pages are loaded into any available memory frames.

Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program. This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages:

- It is independent of external fragmentation.
- Paging reduces external fragmentation, but still suffers from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.

Disadvantages:

- It makes the translation very slow as main memory access two times.
- A page table is a burden over the system which occupies considerable space.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

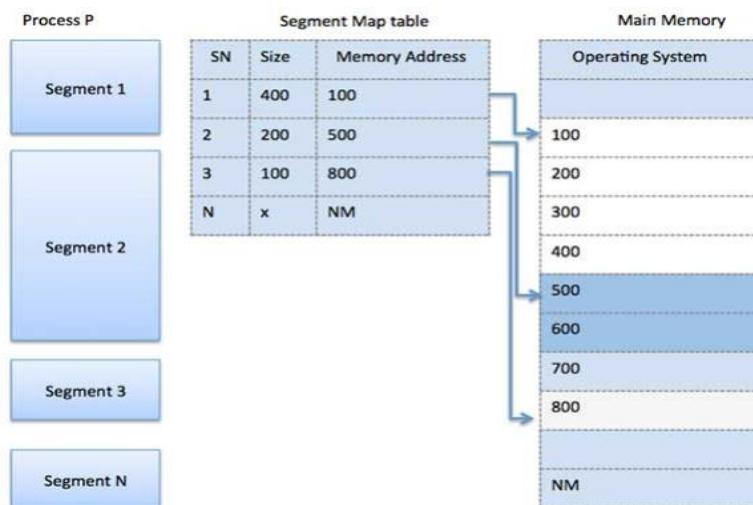
(II) SEGMENTATION

Segmentation is a programmer view of the memory where instead of dividing a process into equal size partition we divided according to program into partition called segments. The translation is the same as paging but paging segmentation is independent of internal fragmentation but suffers from external fragmentation. Reason of external fragmentation is program can be divided into segments but segment must be contiguous in nature.

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program. When a process is to be executed, its corresponding segmentation is loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

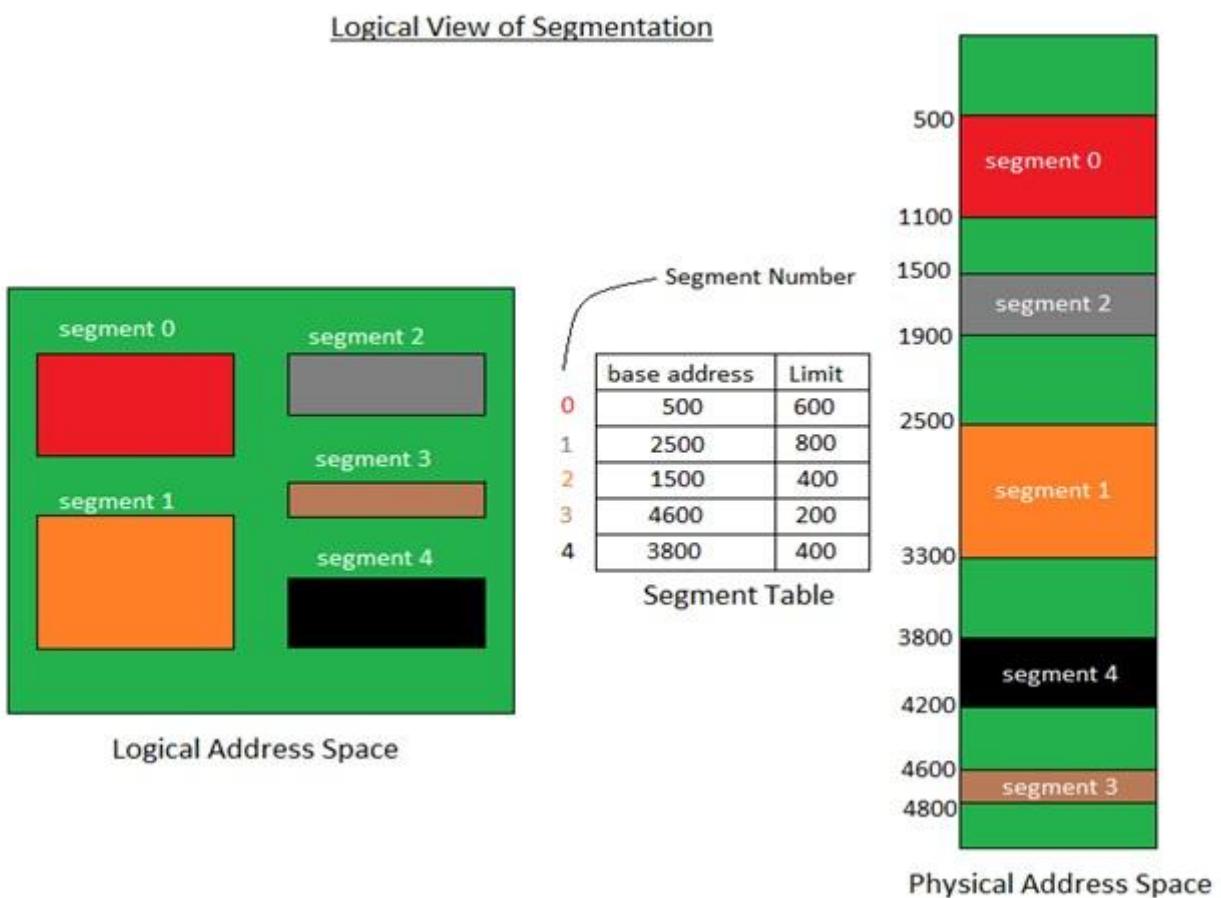
Segmentation memory management works very similar to paging but here segments are of variable length where as in paging pages are of fixed size. A program segment contains the program's main function, utility functions, data structures, and so on.

The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



(III) SEGMENTATION WITH PAGING

In segmentation with paging, we take advantages of both segmentation as well as paging. It is a kind of multilevel paging but in multilevel paging, we divide a page table into equal size partition but here in segmentation with paging, we divide it according to segments. All the properties are the same as that of paging because segments are divided into pages.





Count Balanced Binary Trees of Height h

Last Updated : 10 Dec, 2022

Given a height h , count and return the maximum number of balanced binary trees possible with height h . A balanced binary tree is one in which for every node, the difference between heights of left and right subtree is not more than 1.

Examples :

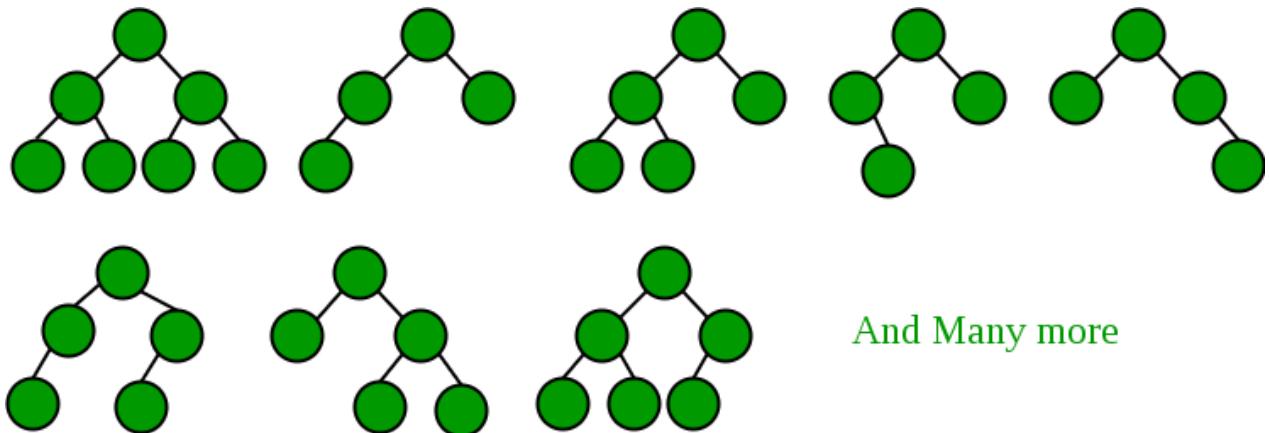
Input : $h = 3$

Output : 15

Input : $h = 4$

Output : 315

Following are the balanced binary trees of height 3.



Height of tree, $h = 1 + \max(\text{left height}, \text{right height})$

Since the difference between the heights of left and right subtree is not more than one, possible heights of left and right part can be one of the following:

3. (h-1), (h-1)

```

count(h) = count(h-1) * count(h-2) +
          count(h-2) * count(h-1) +
          count(h-1) * count(h-1)
= 2 * count(h-1) * count(h-2) +
  count(h-1) * count(h-1)
= count(h-1) * (2*count(h - 2) +
                  count(h - 1))

```

Hence we can see that the problem has optimal substructure property.

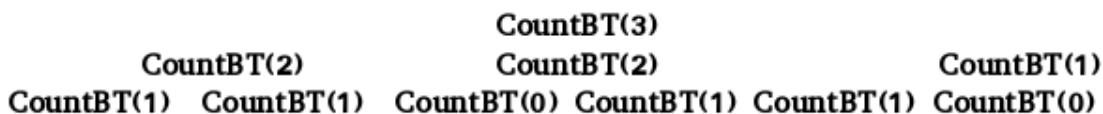
A recursive function to count no of balanced binary trees of height h is:

```

int countBT(int h)
{
    // One tree is possible with height 0 or 1
    if (h == 0 || h == 1)
        return 1;
    return countBT(h-1) * (2 *countBT(h-2) +
                           countBT(h-1));
}

```

The time complexity of this recursive approach will be exponential. The recursion tree for the problem with $h = 3$ looks like :



As we can see, sub-problems are solved repeatedly. Therefore we store the results as we compute them.

An efficient dynamic programming approach will be as follows :

Below is the implementation of above approach:

```
// binary trees of height h.
#include <bits/stdc++.h>
#define mod 1000000007
using namespace std;

long long int countBT(int h) {

    long long int dp[h + 1];
    //base cases
    dp[0] = dp[1] = 1;
    for(int i = 2; i <= h; i++) {
        dp[i] = (dp[i - 1] * ((2 * dp[i - 2])%mod + dp[i - 1])%mod) % mod;
    }
    return dp[h];
}

// Driver program
int main()
{
    int h = 3;
    cout << "No. of balanced binary trees"
         " of height h is: "
         << countBT(h) << endl;
}
```

Java

```
// Java program to count number of balanced
// binary trees of height h.
import java.io.*;
class GFG {

    static final int MOD = 1000000007;

    public static long countBT(int h) {
        long[] dp = new long[h + 1];

        // base cases
        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2; i <= h; ++i)
            dp[i] = (dp[i - 1] * ((2 * dp[i - 2])% MOD + dp[i - 1]) % MOD)
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```
// Driver program
public static void main (String[] args) {
    int h = 3;
    System.out.println("No. of balanced binary trees of height "+h+" is
}
/*
This code is contributed by
Brij Raj Kishore
*/
```

Python3

```
# Python3 program to count number of balanced
# binary trees of height h.

def countBT(h) :
    MOD = 1000000007
    #initialize list
    dp = [0 for i in range(h + 1)]

    #base cases
    dp[0] = 1
    dp[1] = 1

    for i in range(2, h + 1) :
        dp[i] = (dp[i - 1] * ((2 * dp[i - 2])%MOD + dp[i - 1])%MOD) % MO

    return dp[h]

#Driver program
h = 3
print("No. of balanced binary trees of height "+str(h)+" is: "+str(countBT(
# This code is contributed by
# Brij Raj Kishore
```

C#

```
// C# program to count number of balanced
// binary trees of height h.
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

public static long countBT(int h) {
    long[] dp = new long[h + 1];

    // base cases
    dp[0] = 1;
    dp[1] = 1;

    for(int i = 2; i <= h; ++i)
        dp[i] = (dp[i - 1] * ((2 * dp[i - 2]) % MOD + dp[i - 1])) % MOD

    return dp[h];
}

// Driver program
static void Main () {
    int h = 3;
    Console.WriteLine("No. of balanced binary trees of height "+h+" is:");
}
// This code is contributed by Ryuga
}

```

PHP

```

<?php
// PHP program to count
// number of balanced

$mod =1000000007;

function countBT($h)
{
    global $mod;

    // base cases
    $dp[0] = $dp[1] = 1;
    for($i = 2; $i <= $h; $i++)
    {
        $dp[$i] = ($dp[$i - 1] *
                    ((2 * $dp[$i - 2]) %
                     $mod + $dp[$i - 1]) %
                     $mod) % $mod;
    }
    return $dp[$h];
}

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```
$h = 3;
echo "No. of balanced binary trees",
      " of height h is: ",
      countBT($h), "\n";

// This code is contributed by aj_36
?>
```

Javascript

```
// Javascript program to count number of balanced binary trees of height h

let MOD = 1000000007;

function countBT(h) {
    let dp = new Array(h + 1);
    dp.fill(0);

    // base cases
    dp[0] = 1;
    dp[1] = 1;

    for(let i = 2; i <= h; ++i)
        dp[i] = (dp[i - 1] * ((2 * dp[i - 2]) % MOD + dp[i - 1])) % MOD

    return dp[h];
}

let h = 3;
document.write("No. of balanced binary trees of height h is: "+countB
```

Output

No. of balanced binary trees of height h is: 15

Time Complexity: O(n)

Auxiliary Space: O(n), since n extra space has been taken.

Memory efficient Dynamic Programming approach :

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

can replace $dp[i]$, $dp[i-1]$ and $dp[i-2]$ with $dp2$, $dp1$ and $dp0$ respectively. (contributed by **Kadapalla Nithin Kumar**)

Implementation:

C++ ▾

```
// C++ program to count number of balanced
// binary trees of height h.
#include <bits/stdc++.h>
using namespace std;

long long int countBT(int h) {
    if(h<2) {
        return 1;
    }
    const int BIG_PRIME = 1000000007;
    long long int dp0 = 1, dp1 = 1, dp2;

    for(int i = 2; i <= h; i++) {

        dp2 = (dp1 * ((2 * dp0)%BIG_PRIME + dp1)%BIG_PRIME) % BIG_PRIME;

        // update dp1 and dp0
        dp0 = dp1;
        dp1 = dp2;

        // Don't commit following simple mistake
        // dp1 = dp0;
        // dp0 = dp1;
    }
    return dp2;
}

// Driver program
int main()
{
    int h = 3;
    cout << "No. of balanced binary trees"
         " of height h is: "
         << countBT(h) << endl;
}
// This code is contributed by Kadapalla Nithin Kumar
```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

// Java program to count number of balanced
// binary trees of height h.

import java.io.*;
class GFG {

    static final int BIG_PRIME = 1000000007;
    static long countBT(int h) {
        if(h<2) {
            return 1;
        }
        long dp0 = 1, dp1 = 1, dp2 = 3;

        for(int i = 2; i <= h; i++) {

            dp2 = (dp1 * ((2 * dp0)%BIG_PRIME + dp1)%BIG_PRIME) % BIG_PRIME;

            // update dp1 and dp0
            dp0 = dp1;
            dp1 = dp2;

            // Don't commit following simple mistake
            // dp1 = dp0;
            // dp0 = dp1;
        }
        return dp2;
    }
}

// Driver program
public static void main (String[] args) {
    int h = 3;
    System.out.println("No. of balanced binary trees of height "+h+" is "

```

DSA Interview Problems on DP Practice DP MCQs on DP Tutorial on Dynamic Programming Optimal Substru

```

}
/*

```

This code is contributed by
Brij Raj Kishore and modified by Kadapalla Nithin Kumar
*/

Python3

```

# Python3 program to count number of balanced
# binary trees of height h.

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

dp0 = dp1 = 1
dp2 = 3
for _ in range(2, h+1):
    dp2 = (dp1*dp1 + 2*dp1*dp0)%BIG_PRIME
    dp0 = dp1
    dp1 = dp2
return dp2
#Driver program
h = 3
print("No. of balanced binary trees of height "+str(h)+" is: "+str(countBT(
#This code is contributed by Kadapalla Nithin Kumar

```

C#

```

// C# program to count number of balanced
// binary trees of height h.

using System;
class GFG {

    static int BIG_PRIME = 1000000007;

    public static long countBT(int h) {
        // base case
        if(h<2) {
            return 1;
        }

        long dp0 = 1;
        long dp1 = 1;
        long dp2 = 3;
        for(int i = 2; i <= h; ++i) {
            dp2 = (dp1 * ((2 * dp0)% BIG_PRIME + dp1) % BIG_PRIME) % BIG_PRIME;
            dp0 = dp1;
            dp1 = dp2;
        }
        return dp2;
    }

    // Driver program
    static void Main () {
        int h = 3;
        Console.WriteLine("No. of balanced binary trees of height "+h+" is:

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

PHP

```

<?php
// PHP program to count
// number of balanced

$BIG_PRIME =1000000007;

function countBT($h)
{
    global $BIG_PRIME;

    // base cases
    if($h < 2) {
        return 1;
    }

    $dp0 = $dp1 = 1;
    $dp2 = 3;
    for($i = 2; $i <= $h; $i++)
    {
        $dp2 = ($dp1 *
            ((2 * $dp0) % $BIG_PRIME
            + $dp1) %
            $BIG_PRIME) % $BIG_PRIME;
        $dp0 = $dp1;
        $dp1 = $dp2;
    }
    return $dp2;
}

// Driver Code
$h = 3;
echo "No. of balanced binary trees",
      " of height h is: ",
      countBT($h), "\n";

// This code is contributed by aj_36 and modified by Kadapalla Nithin Kumar
?>

```

Javascript

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

function countBT(h) {
    if(h<2) {
        return 1;
    }
    // base cases
    let dp0 = 1;
    let dp1 = 1;
    let dp2 = 3;
    for(let i = 2; i <= h; ++i){
        dp2 = (dp1 * ((2 * dp0)% MOD + dp1) % MOD) % MOD;
        dp0 = dp1;
        dp1 = dp2;
    }
    return dp2;
}

let h = 3;
document.write("No. of balanced binary trees of height h is: "+countB
// This code is contributed by Kadapalla Nithin Kumar

```

Output

No. of balanced binary trees of height h is: 15

Time Complexity: O(n)

Auxiliary Space: O(1)

other Geek.

[Comment](#)
[More info](#)
[Advertise with us](#)

Next Article

Comparison between Height
Balanced Tree and Weight Balanced
Tree

Similar Reads

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Midterm 1 Solutions

1. (10 Points)

Show the AVL tree that results after **each** of the integer keys 9, 27, 50, 15, 2, 21, and 36 are inserted, in that order, into an initially empty AVL tree. Clearly show the tree that results after each insertion, and make clear any rotations that must be performed.

Solution:

See figure 1.

2. (10 Points)

Show the red-black tree that results after each of the integer keys 21, 32, 64, 75, and 15 are inserted, in that order, into an initially empty red-black tree. Clearly show the tree that results after **each** insertion (indicating the color of each node), and make clear any rotations that must be performed.

Solution:

See Figure 2 for bottom-up approach, and Figure 3 for top-down approach.

3. (12 Points)

Indicate for each of the following statements if it is true or false. You must justify your answers to get credit.

- (a) The subtree of the root of a red-black tree is always itself a red-black tree. (Here, the definition of red-black tree is as I have given in class and as described in the textbook.)
- (b) The sibling of a null child reference in a red-black tree is either another null child reference or a red node.
- (c) The worst case time complexity of the insert operation into an AVL tree is $O(\log n)$, where n is the number of nodes in the tree.

Solution:

- (a) FALSE. The root of a red-black must be black, by definition. It is possible for the child of the root of a red-black tree to be red. Therefore, it is possible for the subtree of the root of a red-black tree to have a red root, meaning that it can not be a red-black tree. So, the statement is false.
- (b) TRUE. Let x represent the parent of the null reference, and without loss of generality, suppose $x.\text{right}$ is the null reference. Suppose $x.\text{left}$ refers to a black node. But then, the number of black nodes on the path from the root to the $x.\text{right}$ null reference must be less than the number of black nodes from the root to all nodes in the subtree rooted at $x.\text{left}$. This violates the definition of red-black tree. So, $x.\text{left}$ must be a null reference or a red node.
- (c) TRUE. The work of all AVL tree operations is $O(h)$ where h is the height of the tree. AVL rotations ensure that h is $O(\log n)$. Therefore, insertion must be $O(\log n)$.

4. (10 Points)

Suppose you have an `AVLNode` class that stores integers:

```

public class AVLNode {
    public int item;
    public AVLNode left;
    public AVLNode right;
    public AVLNode (int i, AVLNode l, AVLNode r)
        item = i; left = l; right = r; ;
}

```

Write a complete method that takes a height h , and returns a reference to the root of an AVL tree of height h that contains the minimum number of nodes. The integers stored in the nodes (the item instance variables) should satisfy the binary search tree property, and they should all be distinct (but they do not have to be consecutive). You can define helper methods and/or classes if you wish.

Solution:

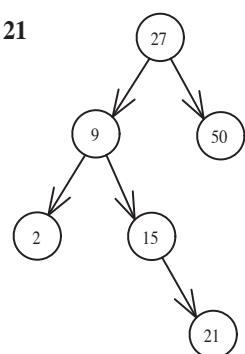
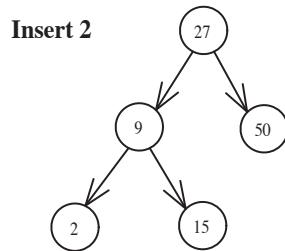
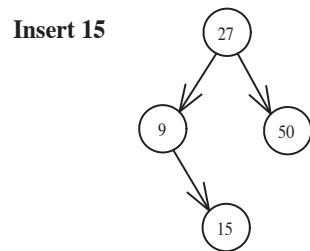
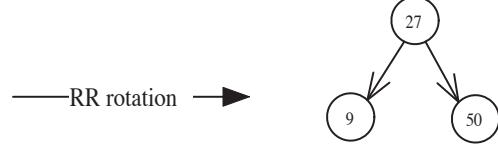
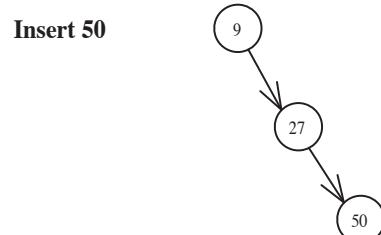
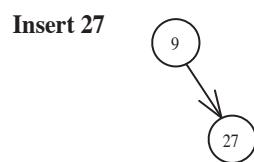
An example solution is as follows.

```

public AVLNode minNodeAVLTree (int h) {
    return minNodeAVLTree(h, 1);
}

private AVLNode minNodeAVLTree (int h, int minValue)
{
    if (h == -1) {
        return null;
    } else {
        /* Numbers do not have to be consecutive. At most  $2^h - 1$  nodes on the left subtree. */
        int rootValue = minValue + (Math.pow(2, h) - 1);
        AVLNode left = minNodeAVLTree (h - 1, minValue);
        AVLNode right = minNodeAVLTree (h - 2, rootValue+1);
        return new AVLNode (rootValue, left, right);
    }
}

```



→ LR rotation →

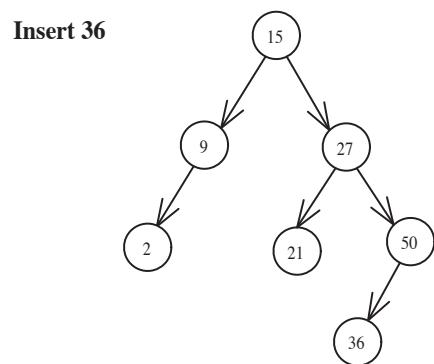
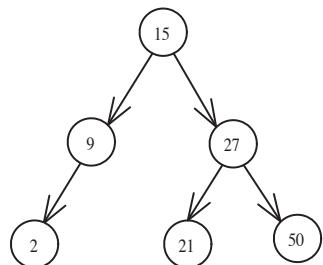
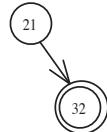


Figure 1: Problem 1. Insertion in an AVL tree

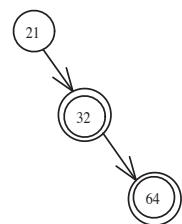
Insert 21



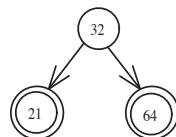
Insert 32



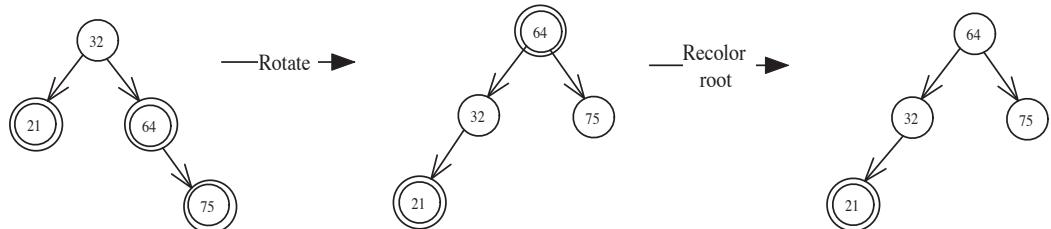
Insert 64



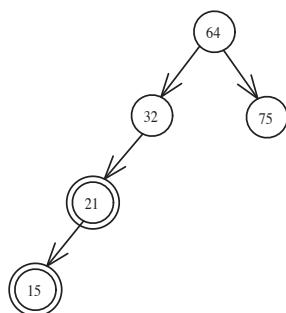
—Rotate →



Insert 75



Insert 15



→

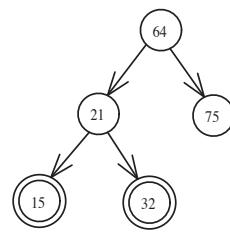


Figure 2: Problem 2. Insertion in a Red-black tree. Bottom-up approach

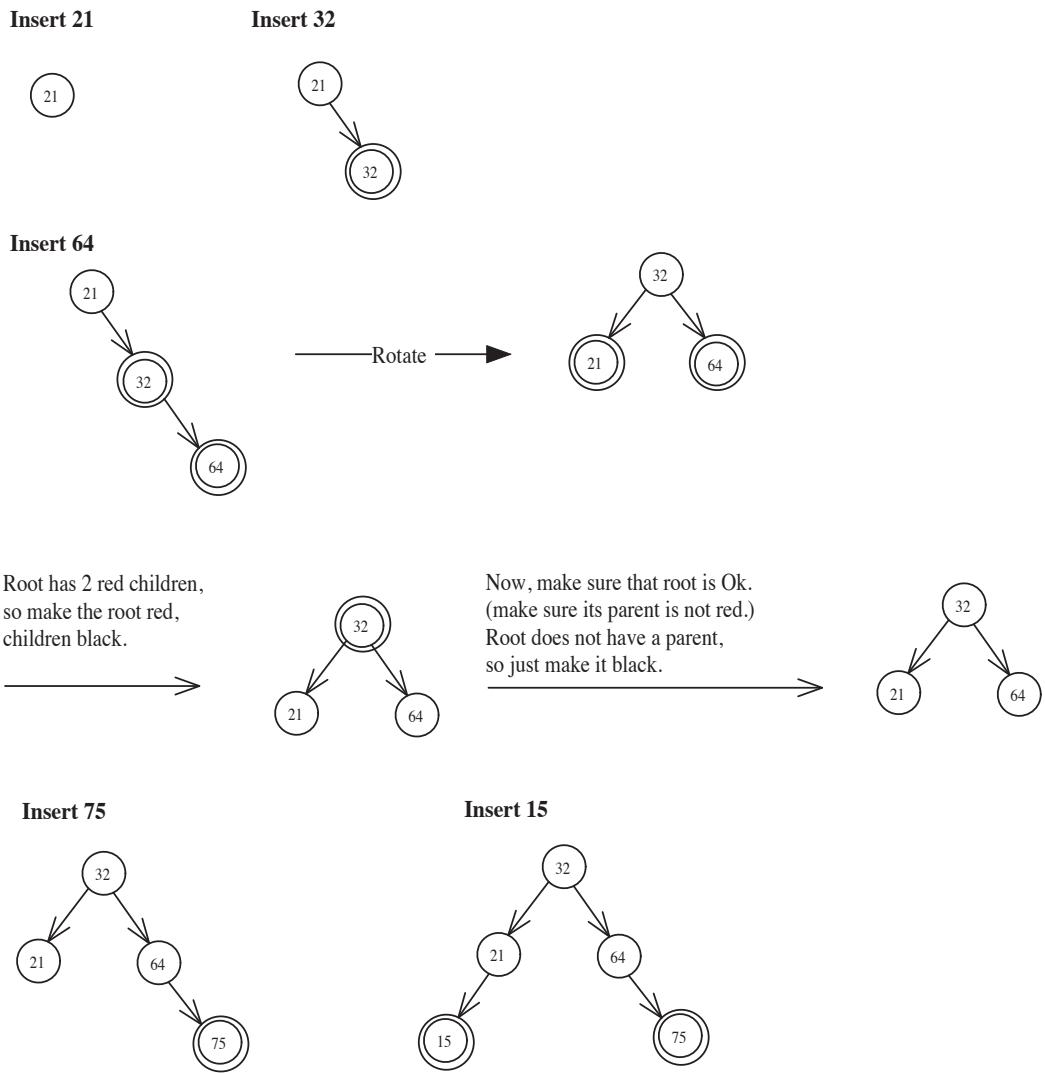


Figure 3: Problem 2. Insertion in a Red-black tree. Top-down approach.

Data Structures in Disk-Based Databases

LSM, BTrees



ANURAG DWIVEDI

OCT 28, 2024



Sha

Previous Series on Evolution of Authentication on the Internet:



Evolution of Authentication on the Internet - 1 #Delegated Authorization

ANURAG DWIVEDI • JANUARY 15, 2024

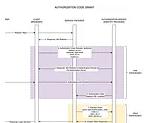
[Read full story →](#)



Evolution of Authentication on the Internet - 2 #SAML

ANURAG DWIVEDI • JANUARY 17, 2024

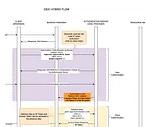
[Read full story →](#)



Evolution of Authentication on the Internet - 3 #OAuth

ANURAG DWIVEDI • JANUARY 27, 2024

[Read full story →](#)



Evolution of Authentication on the Internet - 4 #OIDC

ANURAG DWIVEDI • FEBRUARY 10, 2024

[Read full story →](#)

Evolution of Authentication on the Internet - 5

ANURAG DWIVEDI • FEBRUARY 20, 2024

[Read full story →](#)

Memory vs Disk-Based Databases

Database Management Systems (DBMS) can be traditionally categorized into Memory and Disk-based DBMS. There are obviously other popular categorizations, such as:

1. Relational vs Non-Relational
2. Online Transaction Processing (OLTP) vs Online Analytical Processing (OLAP)
3. Columns vs Row Oriented
4. Time Series / Graph, etc.

However, the distinction of databases according to storage medium (In-Memory vs Disk) remains the focal criteria for evaluating DBMS solutions. The following chart depicts the primary differences between In-Memory and Disk-Based DBMS.

	Memory	Disk
Cost	Expensive	Cheaper
Storage	RAM	HDD / SSD
Durability	Volatile	Durable
Latency	Lower	Higher
Capacity	Smaller	Larger
Use Cases	Real Time Analytics, Caching, Session State Management, etc.	Transactional Systems, Data Warehousing, CMS, etc.
Data Structures	Structures that take advantage of fast RAM access: Lists, Arrays, Hash Tables, etc.	Structures designed to minimize disk seeks and optimize sequential access.
Examples	Redis, Memcached, VoltDB, Apache Ignite, etc.	MySQL, PostgreSQL, MongoDB Cassandra, etc.

* Note that most in-memory databases do offer some durability by using the disk to store logs to recover and prevent permanent loss of volatile data.

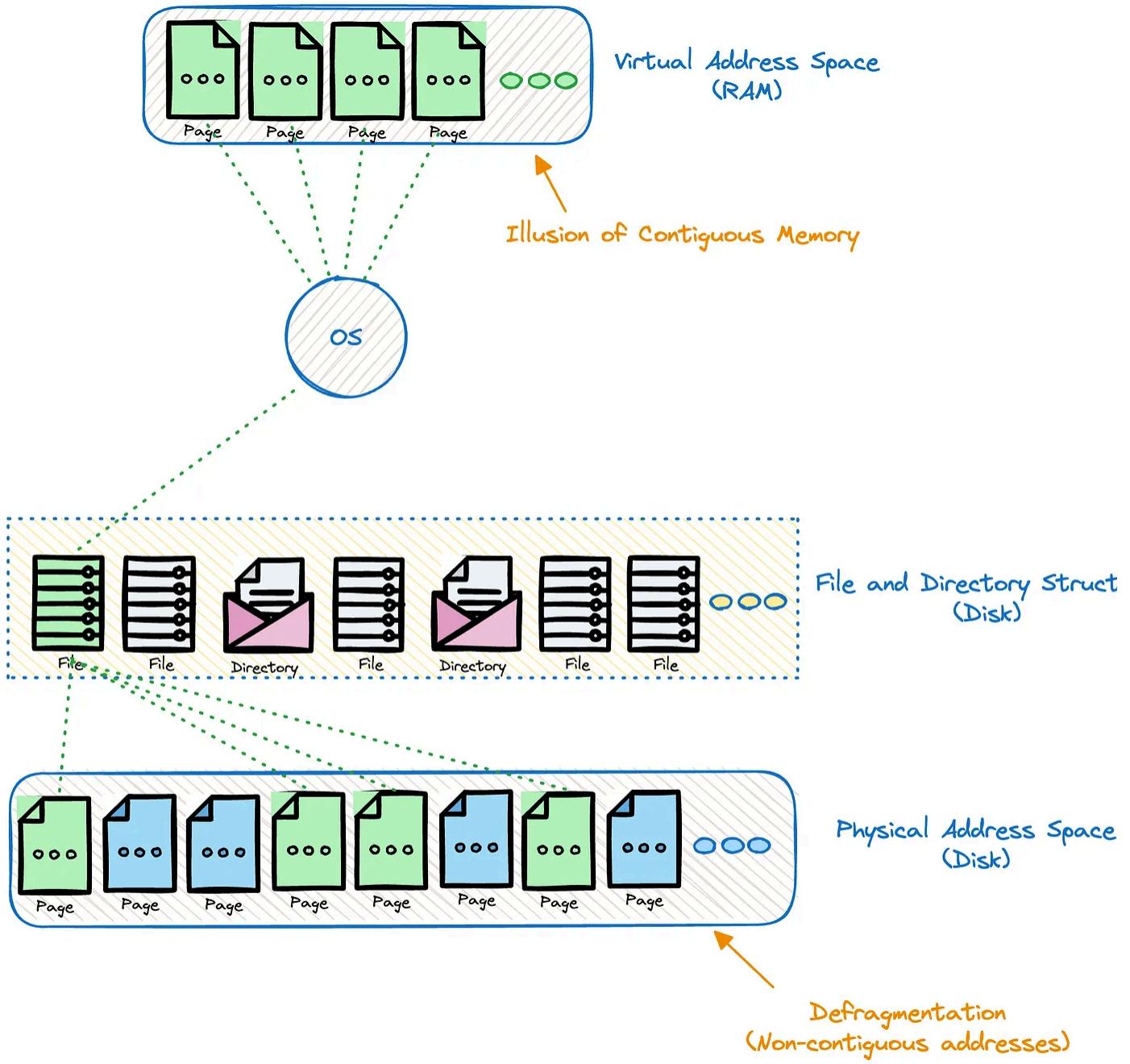
The focus of this post will be on the Data Structures used to design such databases. While the In-Memory data structures are familiar and easier to implement, designi

the data structures for Disk-Based DBMS is more involved with the goal of providing efficient access speeds along with the persistent durability.

How Disk is Organized Overview

The following is a high level overview of how data is read from disk:

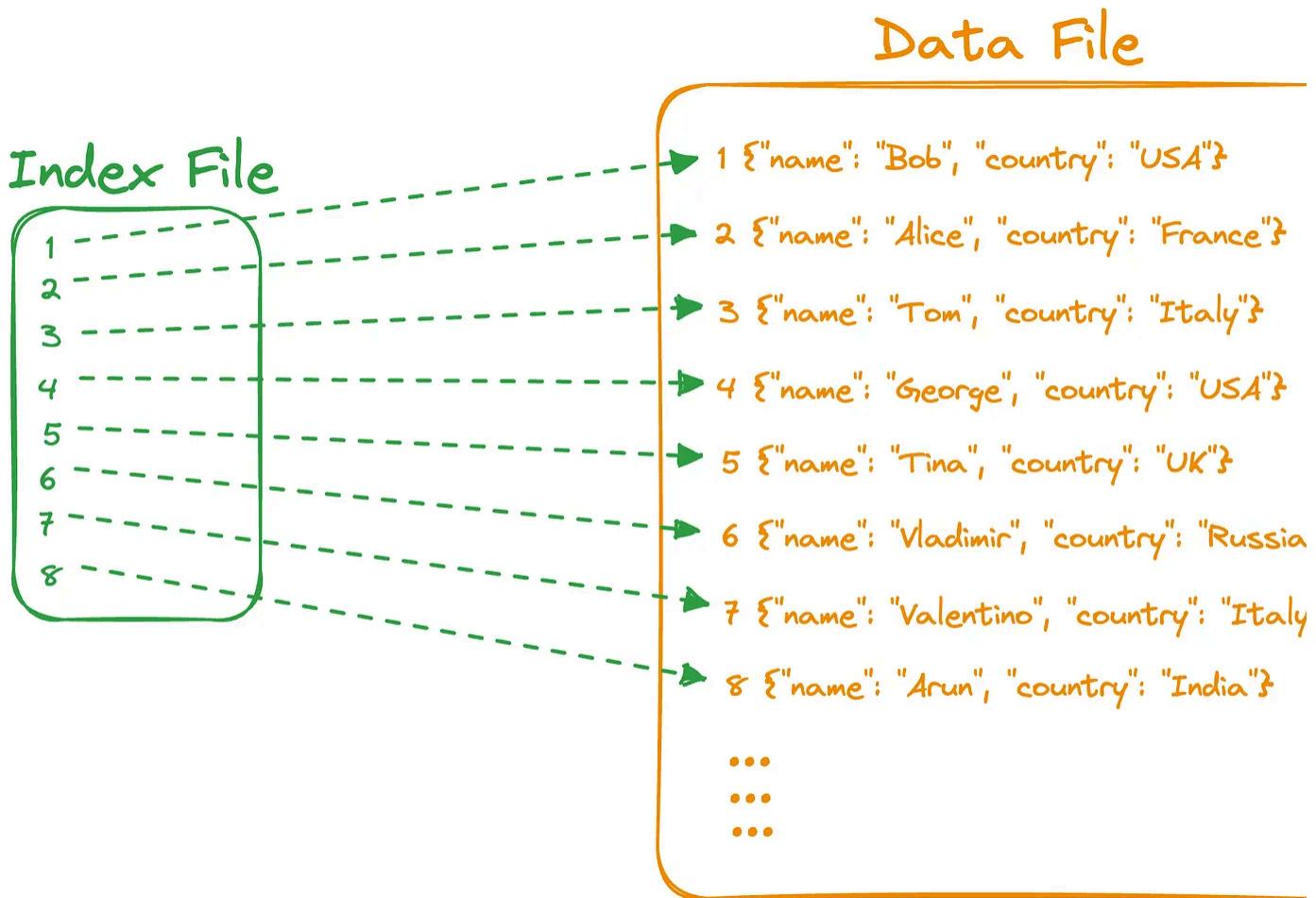
1. Data stored as files.
2. Files are read to memory and written to disk in terms of Pages/Blocks.
3. **The page size is a fundamental unit of data transfer between disk and memory.**
 - a. The operating system defines a fixed-size unit called a "page." Common page sizes range from 4 KB to 16 KB, with 4 KB being a typical choice
4. **Memory Buffering:** The **read data** is often buffered in memory. This means that the operating system loads entire pages into memory, even if the application only requested a smaller portion. Subsequent reads or writes to the same page can then be satisfied from the buffered data in memory, improving performance. Similarly, the **write data** may be buffered to aggregate towards the page/block size to reduce the number of Disk write operations.
5. **Page Alignment:** File data on disk is often organized in page-sized blocks. When a file is read or written, it aligns with these page-sized blocks for efficiency.
6. Data Fragmentation (Refer below)



Data and Index Files

Database systems use files for storing the data using implementation specific formats. Data Files store data records, while index files store record metadata and use it to locate records in data files. There are different ways in which the database may organize the data:

Index Organized (Most Popular)



Above is a simple representation of how the data is logically stored under the Index Organized scheme. The data is stored in Index order. This index can be an auto-generated Primary index or configured via the application developer.

1. Insert Query: The new record is inserted. The Index is updated to include the reference to the record location by the indexing key.
2. Read Query: Index is used to identify the record location.
3. Range Query: Since the data is stored in index order, range traversal is feasible.
4. Delete Query: The record is located using the index. The index is updated to remove the reference to the deleted record. DB systems may implement a further reorganization of the data files to clean up unreferenced records.

Primary vs Secondary Indexes

1. Primary Indexes ensure the uniqueness of each record in the table while Secondary Indexes may hold several entries per search key.
2. The data file is usually ordered based on the primary key (clustered index). Refe *Clustered vs Non-Clustered Indexes*.
3. There can only be one primary index and multiple secondary indexes.
4. Secondary Indexes can use the *Primary Index as an Indirection*. 2 ways in which secondary indexes reference data:
 - a. Direct data reference: Reduce the no of disk seeks, but cost of updating pointers whenever the record is updated or relocated.
 - b. *Primary Index as an Indirection*: reduce cost of pointer updates, higher cost o the read path.

Clustered vs Non-Clustered Indexes

1. A clustered index determines the physical order of data.
2. A non-clustered index is a separate structure from the data rows. It contains a sorted list of key values and pointers to the corresponding data.
3. Only one clustered index, multiple non-clustered indexes.
4. Clustered indexes faster for range queries, while non-clustered indexes offer fa inserts/updates/deletes because the data itself does not need to be reorganized.

Heap Organized

Data File

```
{"name": "Bob", "country": "USA"}  
{"name": "Alice", "country": "France"}  
{"name": "Tom", "country": "Italy"}  
{"name": "George", "country": "USA"}  
{"name": "Tina", "country": "UK"}  
{"name": "Vladimir", "country": "Russia"}  
{"name": "Valentino", "country": "Italy"}  
{"name": "Arun", "country": "India"}  
...  
...
```

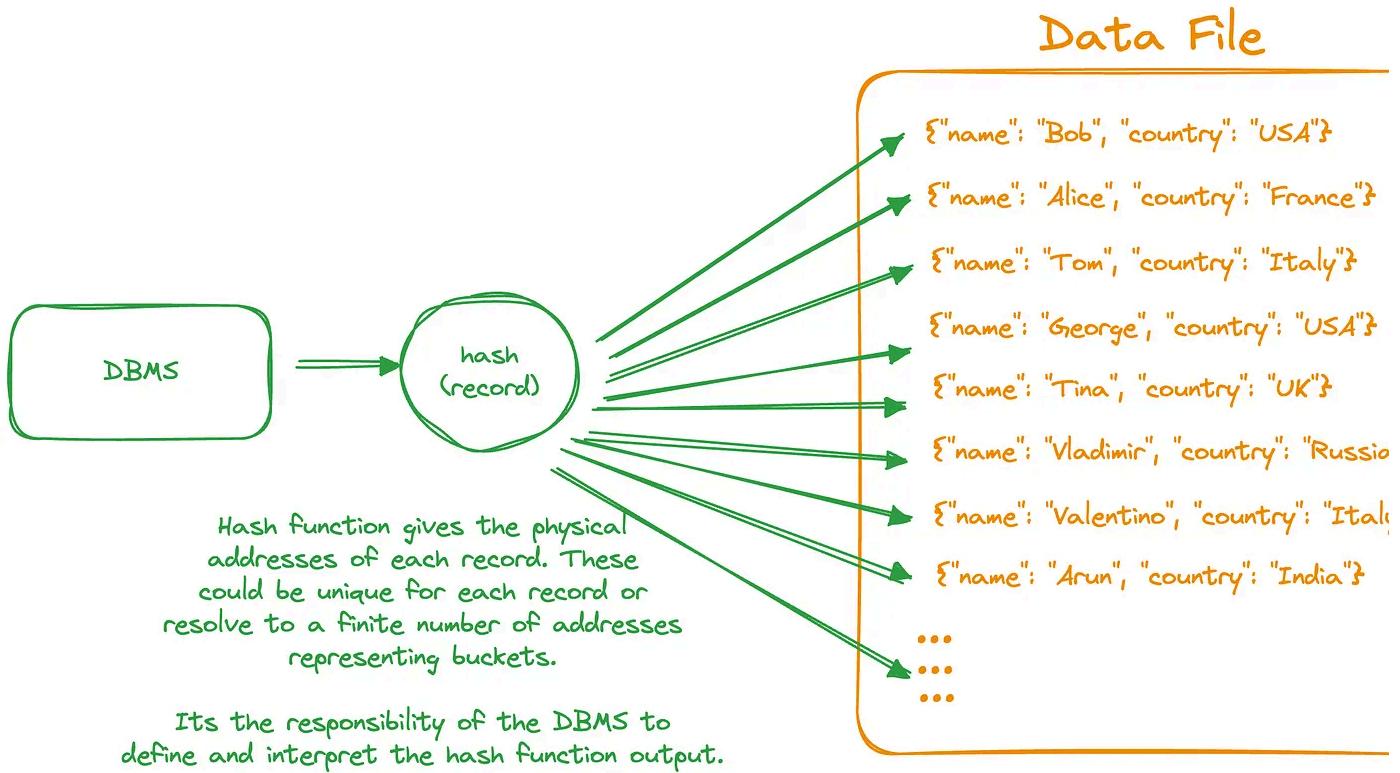
= New Writes appended
to end of File

Above is a simple representation of how the data is logically stored under the Heap Organized scheme. The data is not stored in any particular order.

1. Insert Query: Inserts are efficient with new records appended to the end of the data file.
2. Read Query: Since no order is maintained, Reads need to scan through all records to identify the requested record.
3. Range Query: Since no order is maintained, requires scanning all the records in the data file.

4. Delete Query: The record is located using a full scan. The record data is deleted. DB systems may further implement compaction to remove fragmentation of the data files due to deleted records.

Hash Organized



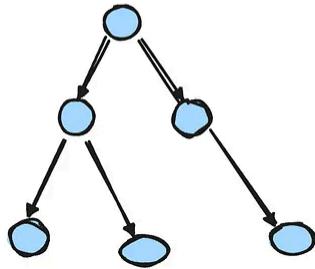
Above is a simple representation of how the data is logically stored under the Hash Organized scheme. This scheme is efficient for exact-match search type queries.

1. Insert Query: Inserts are efficient with the new records written to the location provided by the hash function.
2. Read Query: Efficient Reads as the hash function points to the record location.
3. Range Query: Not suited for range queries, since hash function doesn't preserve order, and neighbouring records may not be stored close to each other.
4. Delete Query: Efficient deletes as the hash function points to the record location.

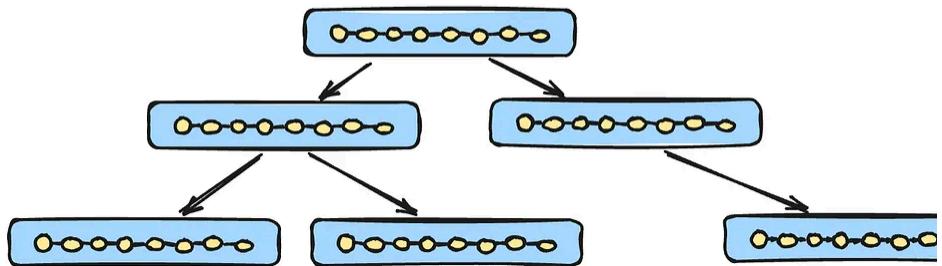
Having glimpsed how the data is laid out on the disk, followed by different indexing strategies, let us review the two most widely used data structures for Disk-Based Databases.

B-Trees

Binary Search Tree (BST)



B-Tree



"B-Trees build upon the foundation of Balanced BSTs and are different in that they have higher fanout (have more child nodes) and smaller height." - Database Internals by Alex Petrov

1. A BST node generally contains a single key, while a B-Tree contains a large number of keys (high fanout)
2. A BST node has two child pointers (left, right). A B-Tree node with ' n ' keys will have ' $n+1$ ' child pointers.
3. Keys inside the B-Tree node are sorted to allow the application of Binary Search when looking for a key inside a B-Tree node.
4. B-Trees maintain balance by ensuring that all leaf nodes are at the same level.

Why B-Trees for Disk Based Storage Systems?

As we outlined under *#How Disk is Organized Overview*, Disk I/O operations typically read and write data in fixed-size blocks or pages.

1. A B-Tree is designed to store as much data as possible in a single node so that each node fits within a disk page.
2. When a node is accessed, the entire node (and all its keys and pointers) is loaded into memory. This reduces the number of disk reads needed since multiple key and pointers are retrieved in a single I/O operation.
3. B-Trees allow efficient caching, taking advantage of locality of reference. Once a node is read into memory, it can be cached, allowing future operations on nearby keys to be faster because they might not require additional disk I/O.

B-Tree Operations

A B-Tree Node with ' k ' keys has a branching factor of ' $k+1$ ' ($k+1$ child nodes).

Therefore the maximum no of B-Tree nodes at height ' h ' is k^h .

Therefore the maximum no. of keys that can be stored in a B-Tree of height ' h ' is $(k^h - 1) * k$.

Most databases can fit into a B-Tree that is 3-4 levels deep, so you don't need to follow many page references to find the page you are looking for. (A 4-level tree of 4KB pages with a branching factor of 500 can store up to 256 TB) - Designing Data-Intensive Applications by Martin Kleppmann.

Get Key

The B-Tree lookup complexity is generally denoted as $\log N$. $\log_k N$ to chase the page reference pointers and $\log_2 k$ for performing binary search within a page(node).

Range Scan

On each level of the B-Tree, we get a more detailed view of the tree.

- The range scan begins by locating the starting key of the range, which is essentially a lookup operation (Get) in the B-Tree, which has a time complexity $\log N$.

- Once the starting key is found, the B-Tree is traversed to collect all the keys within the specified range.
- B-Trees (Read B+ Trees below) are designed for efficient sequential access, so moving from one key to the next within the same node or to an adjacent node is relatively low-cost operation.

Updates Deletes

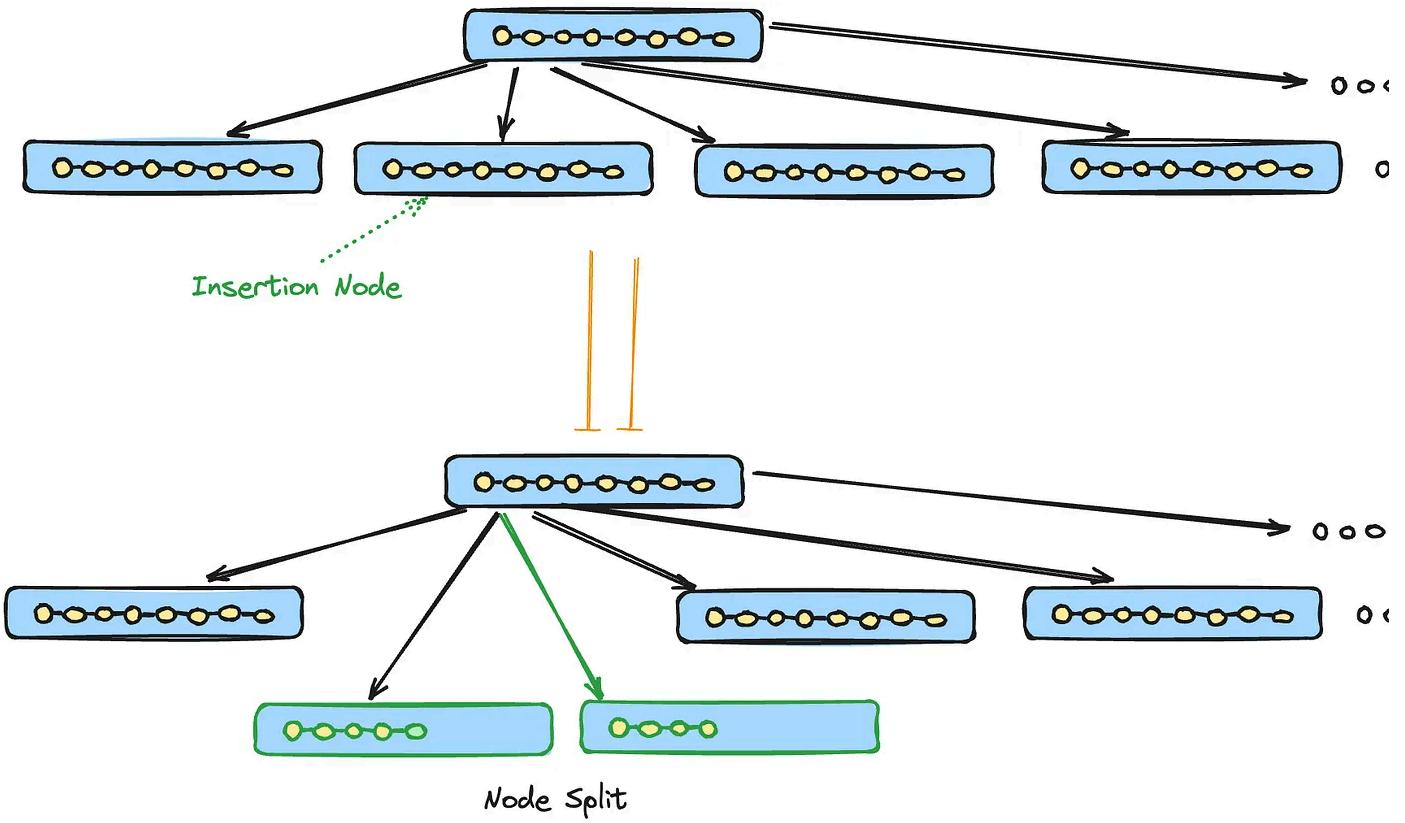
- The lookup algorithm defined above is used to locate the target leaf (node) and find the insertion point.
- Once the target node is located, the key and value are appended to it.
- Similar approach for updates/deletes.

Since B Tree nodes have a defined capacity, insertions and deletions may cause two scenarios:

Node Splits:

If inserting a new key value pair or a new page (node) pointer (Refer B⁺ Trees) cause an overflow, a node is split into two nodes by:

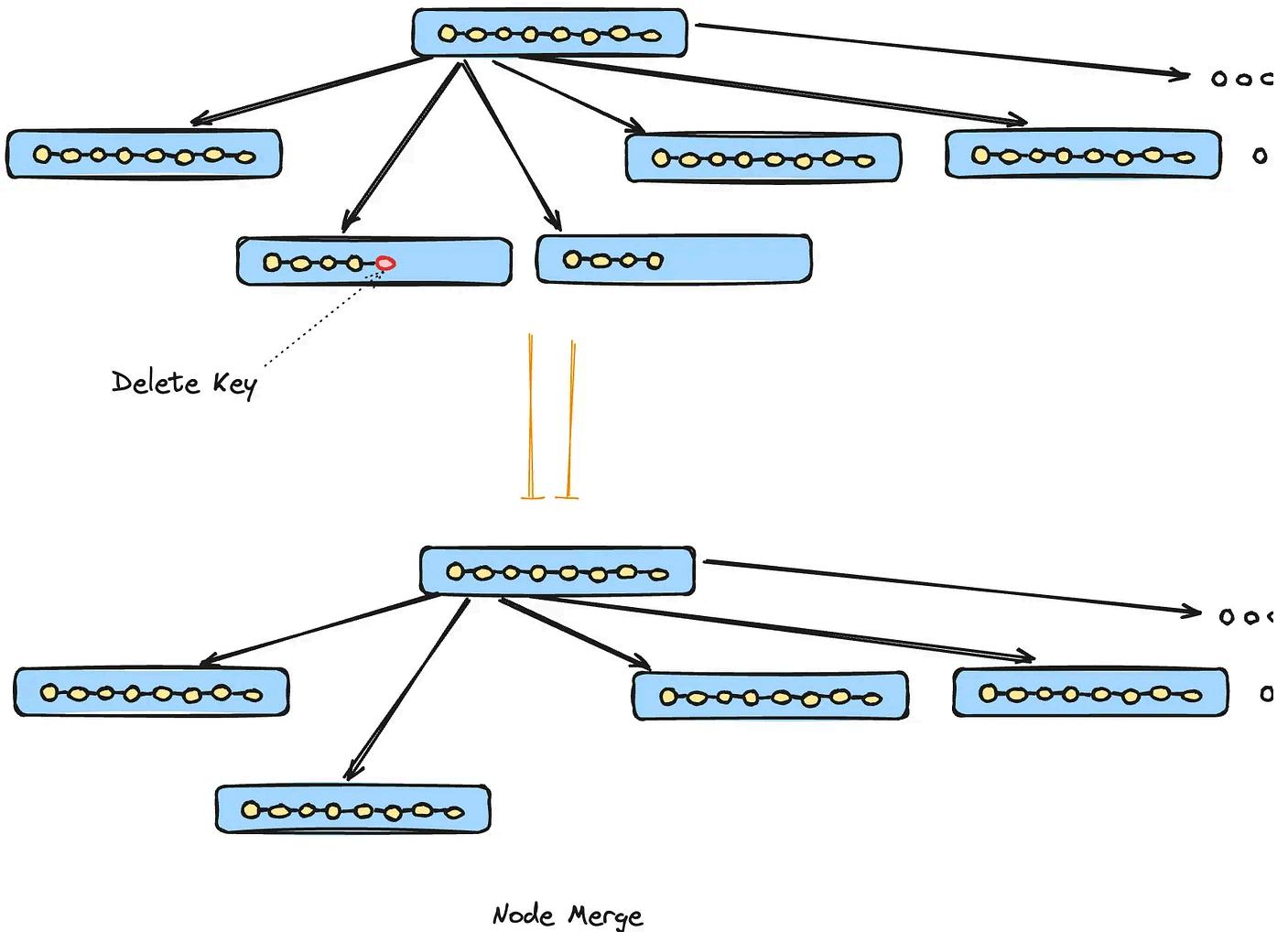
- Allocating a new node
- Copying half the elements from the splitting node to the new one.
- Placing the new element into the corresponding node.
- At the parent of the split node, add a separator key and a pointer to the new no



Node Merges

Deleting a key/value pair may result in two underflow scenarios:

- The contents of adjacent nodes having a common parent, can fit into a single node; hence their contents should be merged.
- Rebalancing: Combined contents don't fit into a single node, and keys are redistributed between them to restore balance.



B⁺ Trees

B Trees allow storing values on any level: root, internal and leaf nodes.

An efficient implementation of B-Trees where values are stored only in leaf nodes is known as B⁺ Trees.

All operations including the retrieval of data records affect only leaf nodes and propagate to higher levels only during Node Splits and Node Merges.

Log Structured Merge Tree (LSM Tree)

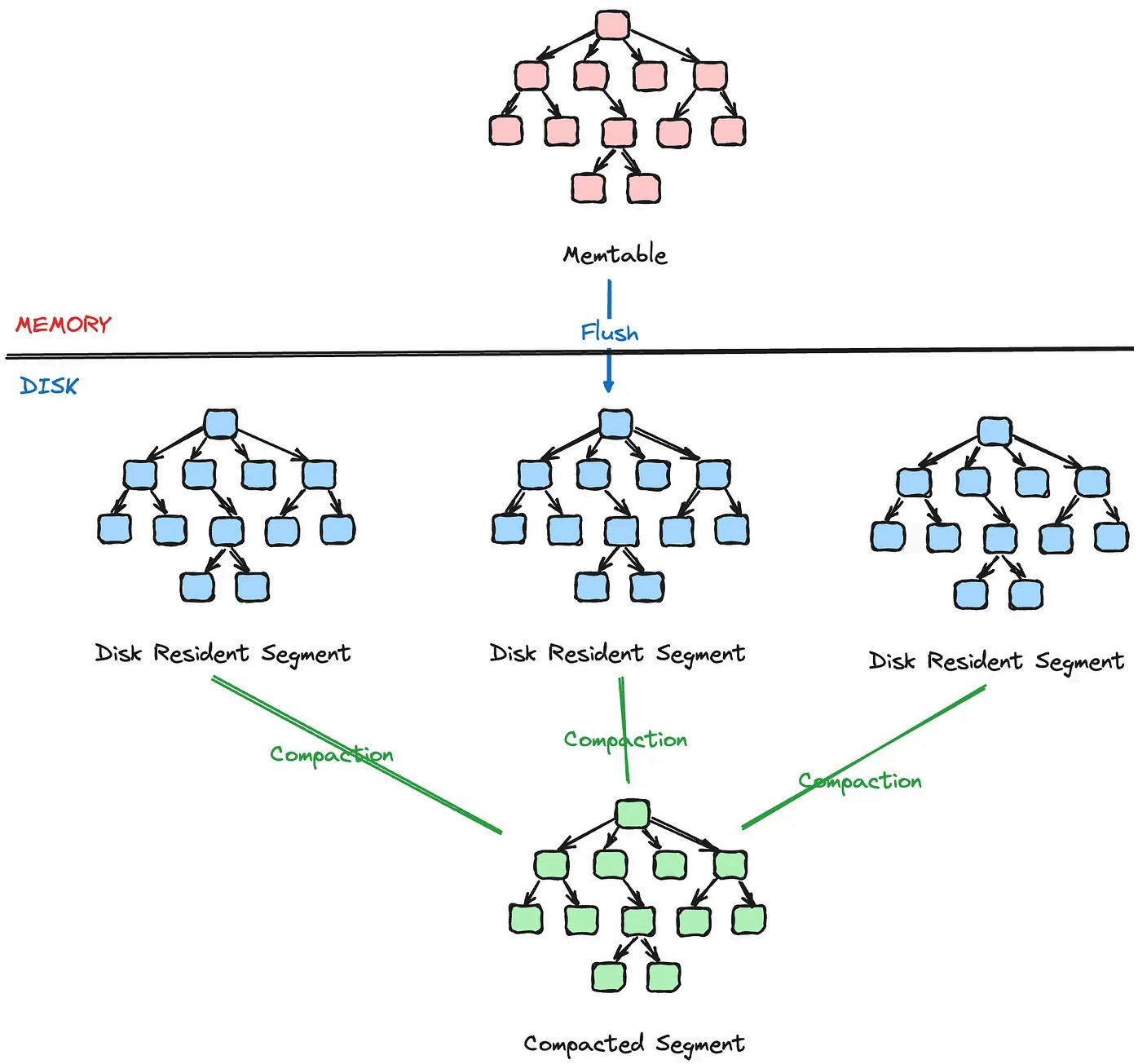
The core concept of an LSM Tree mirrors that of a log file, as both depend on 'append only' and 'immutable' properties.

2024-10-15 22:15:55	"user:1001": {"name": "Alice", "email": "alice@example.com", "age": 28}
2024-10-15 22:17:43	"user:1002": {"name": "Bob", "email": "bob@example.com", "age": 32}
2024-10-15 22:18:32	"user:1003": {"name": "Charlie", "email": "charlie@example.com", "age": 29}
2024-10-15 22:21:28	"user:1001": {"name": "Alice", "email": "alice@example.com", "age": 29}

—Append-Only→

LSM Trees write immutable files and merge them together over time. These files usually contain an index of their own to help readers efficiently locate data (Refer Hash Organized Indexes).

Even though LSM Trees are often presented as an alternative to B-Trees, it is common for B-Trees to be used as the internal indexing structure for an LSM Tree's immutable files.



Memtable is a memory-resident, mutable component. It buffers data records and serves as a target for read and write operations. Memtable contents are persisted on disk when its size grows up to a configurable threshold. Memtable updates incur no disk access and have no associated I/O costs.

A *write-ahead-log (WAL)* file is used to guarantee durability of data records.

Disk-resident components are built by *flushing* contents buffered in memory to disk. These components are used only for reads: buffered contents are persisted, and fi

are never modified.

Writes against an in-memory table, Reads against disk and memory-based tables

2-component LSM Tree: Only 1 disk component. Periodic flushes merged contents a memory-resident segment and the disk-resident segment.

Multi-component LSM Tree: Multiple Disk components, *Memtables* flushed on exceeding threshold. Requires periodic merge process called *Compaction*.

LSM-Tree Operations

Updates/Deletes

Insert, Update and Delete operations do not require locating data records on disk. Instead, redundant records are reconciled during the read.

However, deletes need to be recorded explicitly. This is done by inserting a special delete entry sometimes called a *tombstone*. The reconciliation process picks up tombstones, and filters out the records. The *compaction* process also ignores these records when merging multiple segments.

Lookups

Generally the contents of the disk-resident segments are stored in a sorted fashion. **multiway-merge sort algorithm is used.** The algorithm may use a priority queue type data structure such as a min-heap:

1. Fill the PQ with the first items from each iterator (per segment).
2. Take the smallest element (head) from the queue.
3. Refill the queue from the corresponding iterator (segment), unless this iterator exhausted.

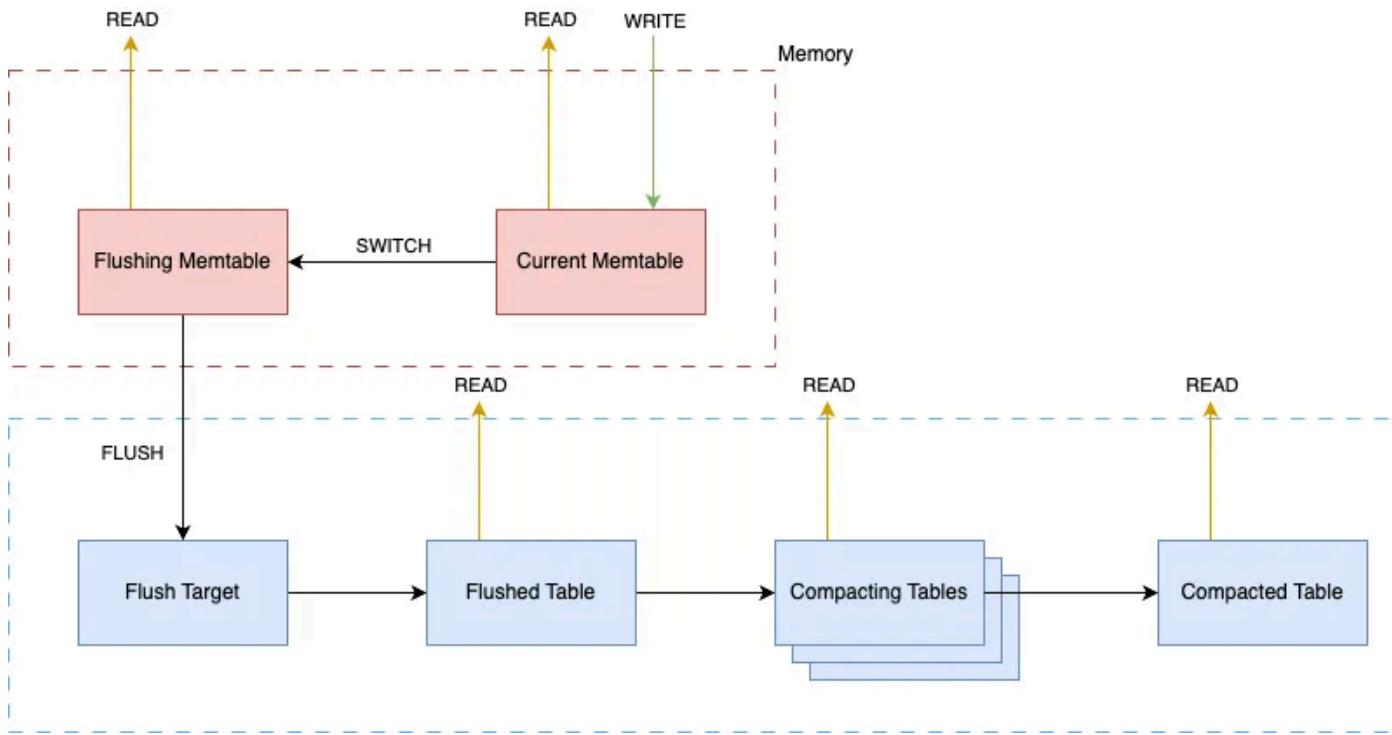
Overall lookup complexity is $N \log N$.

What happens during Merge conflicts?: Reconciliation

A scenario where different segments may hold data records for the same key, such as updates and deletes. The PQ implementation should be able to allow multiple values associated with the same key.

We need to understand which data value takes precedence. Data records hold metadata necessary for this, such as timestamps.

The following figure reproduced from the one in “*Database Internal* by Alex Petrov” (Chapter 7) denotes the Read and Write Paths against the components of an LSM storage engine:



When storing data on disk in an immutable fashion (LSM), we face three problems:

Read Amplification: Resulting from a need to address multiple tables to retrieve data

Write Amplification: Caused by continuous rewrites by the compaction process.

Space Amplification: Arising from storing multiple records associated with the same key.

One of the popular cost models for storage structures: **RUM Conjecture** takes the factors into consideration. RUM Conjecture states that reducing two of these overheads inevitably leads to change for the worse in the third one, and that optimization can be done only at the expense of one of the three parameters.

Ordered vs Unordered LSM Storage

Sorted String Tables (SSTables)

Data records in SSTables are sorted and laid out in key order. SSTables usually consist of two components: index files and data files (Refer Data and Index Files section above).

Index files are implemented using some structure allowing logarithmic lookups, such as B-Trees, or constant-time lookups, such as hash-tables.

SSTables offer support for Range Scans.

Unordered LSM Storage

Unordered stores generally do not require a separate log and allow us to reduce the cost of writes by storing data records in insertion order.

Such stores use a hash-table like data structure (*keydir*), which holds references to the latest data records for the corresponding keys. Old data records may still be present on disk, but are not referenced from *keydir*, and are garbage collected during compaction. This approach is used in one of the storage engines (Bitcask) used in *Riak*. **This approach does not offer support for Range Scans.**

WiscKey is an interesting storage engine that attempts to preserve the write and space advantages of unordered storage, while still allowing us to perform range scans.

Conclusion: Comparing B-Trees and LSM-Trees

1. LSM Trees are immutable and append-only, meaning data is never modified in place; instead, new records are appended. In contrast, B-Trees perform in-place updates.

updates, modifying records directly on disk.

2. **Storage Structure and Data Update:** LSM Trees write data sequentially and use periodic merging for reconciliation. B-Trees locate and update data records at their original disk locations, resulting in more random I/O operations.
3. **Read vs. Write Optimization:** B-Trees are optimized for faster reads by locating data in place, but write performance suffers as updates require locating the record first. LSM Trees, optimized for write performance, avoid locating records by appending new data but require reconciliation across multiple versions during reads.
4. **I/O Pattern:** B-Trees primarily involve random I/O for reads and updates, rewriting entire pages even for minor changes. LSM Trees, by contrast, perform more sequential I/O and avoid page rewrites, enhancing their write performance.
5. **Performance Comparison:** Generally, LSM Trees perform better for writes and B-Trees for reads due to their design trade-offs.
6. **Storage Efficiency:** LSM Trees can achieve better compression and smaller file sizes by minimizing fragmentation with [leveled compaction](#). B-Trees often waste space due to page splits and unutilized space, which accumulates as fragmentation over time.
7. **Compaction Overhead:** A downside to LSM Trees is the potential for compaction processes to interfere with read/write performance, especially under heavy load. B-Trees, by comparison, offer more predictable response times, though at the cost of reduced write throughput.
8. **Disk Bandwidth Utilization:** LSM Trees require the disk's bandwidth to be split between initial writes (e.g., logging and flushing memtables) and compaction, which intensifies with data growth. This can constrain write performance in large-scale applications.
9. **Compaction Tuning Challenges:** If compaction doesn't keep pace with incoming writes, the number of unmerged segments on disk can grow, increasing disk space usage.

usage and slowing reads. Effective compaction tuning is critical for sustaining LSM Tree performance.

10. **Typical Use Cases:** B-Trees are commonly found in relational databases and use cases with balanced, read-optimized workloads. LSM Trees are suited to high-write, large-scale sequential workloads, such as logging and NoSQL databases.
11. **Examples of Each:** Databases with B-Trees include MySQL, PostgreSQL, SQLite and Oracle. Those using LSM Trees include Cassandra, HBase, LevelDB, and RocksDB.

Bibliography

1. [Designing Data-Intensive Applications by Martin Kleppmann](#)
2. [Database Internals by Alex Petrov.](#)

Thanks for reading Ensemble Engineering!
Subscribe for free to receive new posts and support my work.

Type your email... Subscribe

 Message ANURAG DWIVEDI



3 Likes • 1 Restack

Discussion about this post

Comments Restacks



Write a comment...

© 2025 ANURAG DWIVEDI • [Privacy](#) • [Terms](#) • [Collection notice](#)
[Substack](#) is the home for great culture

1. Basic Definition

- Disk-based indexing stores index structures on persistent storage (hard drives/SSDs) rather than in volatile RAM.
- It uses data structures optimized for block storage access patterns (e.g., B+ trees) to minimize disk I/O operations¹⁶.

2. Key Components

Aspect	Description
Data Locality	Stores related data in contiguous disk blocks to reduce seek times ⁵ .
Block-Oriented Design	Structures like B+ trees match disk block sizes for efficient read/write ⁶ .
Persistence	Survives system crashes/reboots, unlike in-memory indexes ⁴ .

3. Why It Matters

1. Handles Large Datasets
 - Supports tables larger than available RAM by storing indexes on disk⁴.
 - Example: A 100 GB database can use disk-based indexes without requiring 100 GB of RAM.
2. Reduces Disk I/O
 - Minimizes full-table scans by allowing direct access to data locations.
 - Source² reports ~30% reduction in disk I/O with proper indexing.
3. Optimizes Query Types
 - Range queries: Linked leaf nodes in B+ trees enable sequential access¹.
 - Exact-match queries: Hash-based indexes provide O(1) lookups¹.
4. Balances Tradeoffs

- Read vs. Write Costs: Indexes speed up queries but slow down inserts/updates⁵.
- Space Overhead: Indexes consume disk space (e.g., 20% extra storage for common workloads)⁶.

4. Critical Disk vs. Memory Tradeoffs

Factor	In-Memory Index	Disk-Based Index
Speed	Nanosecond access (RAM)	Millisecond access (disk)
Capacity	Limited by RAM size	Scales to petabytes
Use Case	Real-time analytics	OLTP databases, large datasets ⁴
Cost	Expensive (RAM costs)	Cost-effective (disk storage)

5. Implementation Essentials

1. Data Structures
 - B+ Trees: Default for most databases (balanced height, sequential leaf nodes)⁶.
 - Hash Indexes: For fast equality checks (e.g., `WHERE id = 123`)¹.
2. Index Types
 - Clustered: Physically orders data rows (e.g., primary key in SQL Server)³.
 - Non-Clustered: Separate structure with pointers to data¹.
3. Storage Models
 - Row-Oriented: Optimized for transactional workloads⁵.
 - Column-Oriented: Better for analytical queries (e.g., aggregations)⁵.

6. Real-World Impact

- Case Study: A database without indexes might require scanning 1M rows for a query, while a B+ tree index reduces this to ~20 disk accesses (
- $O(\log mN)$
- $O(\log mN)$
- $O(\log N)$
- Tradeoff Example: Adding an index improves read performance by 10x but increases write latency by 15%⁵.

By understanding these principles, you can explain how disk-based indexing enables databases to manage massive datasets efficiently while balancing speed, cost, and storage constraints.

To understand why indexing is so efficient, you need to understand how computers connect with the datasource - simplistically in the case of a disk it will 'read' a block of data from the disk (i.e. a page of the librarian index), not sure what it is these days but probably something like 4kb look for what is required and if it doesn't find it, read the next block. If it is reading blocks with whole records (for a sequential search) it might pick up say 100 records, but if indexed it might pick up 1000 (because indexes are smaller) so will find your record 10 times quicker - and with indexing algorithms it will have a better idea of what block to read next - in the analogy above, you look at the first page - books starting with A, but since you are looking for a book starting with Z, you know to go to the last page and work backwards.

Indexing does have a time overhead when a record is inserted, deleted or the indexed field changed because it needs to be updated to maintain the index order. But this more than pays dividends when you want to find that record again.

However there is no point in indexing for the sake of it, just those fields you are going to join on or regularly sort and/or filter on. There is also little point in indexing fields which have few distinct values (like Booleans) or contain a lot of nulls because the index itself will not be that efficient (although with Access you can set the index to ignore nulls). Go back to library analogy - how useful is it to have pages of 'Yes' followed by pages of 'No'?

Clearly the size of the index field will also have an impact - the larger the datatype, the fewer elements can be read in a single block. Longs (typically used for autonumber primary keys) are 4 bytes in length, dates (used for timestamps for example) are 8 bytes in length whilst text is 10 bytes plus the number of characters.

Users will still need to search on text (for a name for example), but performance can be improved by using a numeric link between a relatively short list of author names and the long list of books they have written. Primary key is in the author name table and foreign/family key is in the book table.

Using an initial * in a like comparison, prevents the use of indexing because indexing starts from the first character (think of the librarian index above) so should be avoided as much as possible - how many users would look for 'Smith' by entering 'ith'? This is also a good reason for storing names in two fields (firstname/lastname) rather than a single field so each can be indexed separately.

Sometimes it can't be avoided - but better to either train users to enter a * when required, or provide a button/option for begins with/contains to implement the initial * when required.

DS4300 HW 1

January 12, 2025

1 DS4300 HW 1

Lesrene Browne, Jan 14 2025

1.0.1 Problem 1

Linear search and Binary search aim to find the location of a specific value within a list of values (sorted list for binary search). The next level of complexity is to find two values from a list that together satisfy some requirement.

Propose an algorithm (in Python) to search for a pair of values in an unsorted array of n integers that are closest to one another. Closeness is defined as the absolute value of the difference between the two integers. Your algorithm should not first sort the list. [10 points]

```
[7]: # Problem 1, Part 1 Algorithm

def closest_pair_v1(lst):

    if len(lst) < 2:
        return None # this lst doesn't have enough elements to have pairs

    smallest_closeness = float('inf')
    pair = None

    for curr_pos in range(len(lst)):
        for pos in range(curr_pos+1, len(lst)):
            closeness = abs(lst[curr_pos] - lst[pos])
            if closeness < smallest_closeness:
                smallest_closeness = closeness
                pair = (lst[curr_pos], lst[pos])
    return pair
```

```
[9]: # Problem 1, Part 1 Test

practice_array = [4, 1, 20, 8, -2, 7, 3]

# should return (4,3)
practice_result1 = closest_pair_v1(practice_array)
practice_result1
```

[9]: (4, 3)

Next, propose a separate algorithm (in Python) for a *sorted* list of integers to achieve the same goal. [10 points]

[10]: # Problem 1, Part 2 Algorithm

```
# lst will be sorted before being plugged in
def closest_pair_v2(lst):

    if len(lst) < 2:
        return None # this lst doesn't have enough elements to have pairs

    smallest_closeness = float('inf')
    pair = None

    for pos in range(len(lst)-1): # doesn't need to make it to the last element
        ↪because the pairs will be adjacent since the elements are sorted so the last
        ↪element will already have been considered when the 2nd-to-last element pair
        ↪closeness is evaluated
        closeness = abs(lst[pos] - lst[pos+1])
        if closeness < smallest_closeness:
            smallest_closeness = closeness
            pair = (lst[pos], lst[pos+1])
    return pair
```

[11]: # Problem 1, Part 2 Test

```
sorted_practice_array = sorted(practice_array)

# should return (3,4)
practice_result2 = closest_pair_v2(sorted_practice_array)
practice_result2
```

[11]: (3, 4)

Briefly discuss which algorithm is more efficient in terms of the number of comparisons performed. A formal analysis is not necessary. You can simply state your choice and then justify it. [5 points]

Problem 1, Part 3

The second algorithm is more efficient because it only has to traverse the list once (since the list is already sorted). Since it only traverses the list once it makes less comparisons. In the worst case the time complexity for the first algorithm would be $O(n^2)$, because we have to visit every element of the list at least twice due to the two loops and the time complexity for the second algorithm is $O(n)$, as we scan through the list once which is clearly better.

1.0.2 Problem 2

Implement in Python an algorithm for a level order traversal of a binary tree. The algorithm should print each level of the binary tree to the screen starting with the lowest/deepest level on the first line. The last line of output should be the root of the tree. Assume your algorithm is passed the root node of an existing binary tree whose structure is based on the following BinTreeNode class. You may use other data structures in the Python foundation library in your implementation, but you may not use an existing implementation of a Binary Tree or an existing level order traversal algorithm from any source.

Construct a non-complete binary tree of at least 5 levels. Call your level order traversal algorithm and show that the output is correct. [20 points]

```
[14]: class BinTreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

[22]: # Problem 2 Algorithm

from collections import deque

def reverse_level_order_traversal(root):

    result = []
    de = deque([root])

    while de:
        level_size = len(de)  # number of nodes that are supposed to be in the
        ↪current level
        current_level = []

        for unprocessed_node in range(level_size):
            node = de.popleft()  # taking nodes from the right would show the
            ↪output from right to left or include child nodes which isn't correct
            current_level.append(node.value)

            # adds the children of the current node to the end of the queue
            # so that every node in the current level's children will be in the
            ↪queue to be processed from left to right
            if node.left:
                de.append(node.left)
            if node.right:
                de.append(node.right)

        # once the amount of unprocessed nodes from the current level is
        ↪complete, the current level's list of vals is added as a list to the result
```

```
    result.append(current_level)

    for lst in reversed(result):
        print(lst)
```

[23]: # Problem 2 Test

```
# generating a non-complete BST that has 5 levels
root = BinTreeNode(10)
root.left = BinTreeNode(5)
root.right = BinTreeNode(15)

root.left.left = BinTreeNode(3)
root.left.right = BinTreeNode(7)

root.left.left.left = BinTreeNode(2)

root.right.right = BinTreeNode(20)
root.right.right.left = BinTreeNode(17)
root.right.right.right = BinTreeNode(25)

root.right.right.left.right = BinTreeNode(18)

practice_result3 = reverse_level_order_traversal(root)
practice_result3
```

[18]
[2, 17, 25]
[3, 7, 20]
[5, 15]
[10]

DS4300 HW 3

February 18, 2025

1 Homework Assignment 03

1.1 DS 4300 - Spring 2025

- **EC Due Date:** Feb 16, 2025 @ 11:59pm
- **Regular Due Date:** Feb 18, 2025 @ 11:59pm
- Upload to GradeScope (no question/solutions to Match)

```
[1]: # Set up your connection to Mongo DB here.  
!pip install pymongo  
import pymongo  
from bson.json_util import dumps  
  
# I'm not sure why I can't use a username & password but something goes wrong  
# with the connection everytime I try to incorporate them  
uri = "mongodb://localhost:27017/"  
client = pymongo.MongoClient(uri)
```

Requirement already satisfied: pymongo in ./opt/anaconda3/lib/python3.8/site-packages (4.10.1)

Requirement already satisfied: dnspython<3.0.0,>=1.16.0 in
./opt/anaconda3/lib/python3.8/site-packages (from pymongo) (2.6.1)

1.2 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.

```
[2]: mflixdb = client.mflix
```

1.2.1 Question 1:

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

```
[46]: MA_theaters = mflixdb.theaters.aggregate([  
    {"$match": {"location.address.state": "MA"}},  
    {"$project": {"_id": 0, "location.address.street1": 1, "location.address.  
    ↵city": 1, "location.address.zipcode": 1}}
```

```

])
MA_theaters = list(MA_theaters)
print(MA_theaters[:10])
# print(dumps(MA_theaters, indent=4)) # not using dumps because the output is
    ↪super long

```

```

[{'location': {'address': {'street1': '162 Santilli Hwy', 'city': 'Everett',
'zipcode': '02149'}}}, {'location': {'address': {'street1': '14 Allstate Rd',
'city': 'Dorchester', 'zipcode': '02125'}}}, {'location': {'address':
{'street1': '280 School St', 'city': 'Mansfield', 'zipcode': '02048'}}},
{'location': {'address': {'street1': '208 Fortune Blvd', 'city': 'Milford',
'zipcode': '01757'}}}, {'location': {'address': {'street1': '33 Orchard Hill
Park Dr', 'city': 'Leominster', 'zipcode': '01453'}}}, {'location': {'address':
{'street1': '2 Galleria Mall Dr', 'city': 'Taunton', 'zipcode': '02780'}}},
{'location': {'address': {'street1': '84 Middlesex Turnpike', 'city':
'Burlington', 'zipcode': '01803'}}}, {'location': {'address': {'street1': '250
Granite Street', 'city': 'Braintree', 'zipcode': '02184'}}}, {'location':
{'address': {'street1': '999 South Washington Street', 'city': 'North
Attleboro', 'zipcode': '02760'}}}, {'location': {'address': {'street1': '100
CambridgeSide Place', 'city': 'Cambridge', 'zipcode': '02141'}}}]

```

1.2.2 Question 2:

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

```
[45]: count_theaters = mflixdb.theaters.aggregate([
    {"$group": {"_id": "$location.address.state", "theater_count": {"$sum": ↪
        1}}},
    {"$sort": {"_id": 1}}
])

count_theaters = list(count_theaters)
print(count_theaters[:10])
```

```

[{'_id': 'AK', 'theater_count': 4}, {'_id': 'AL', 'theater_count': 19}, {'_id':
'AR', 'theater_count': 16}, {'_id': 'AZ', 'theater_count': 26}, {'_id': 'CA',
'theater_count': 169}, {'_id': 'CO', 'theater_count': 26}, {'_id': 'CT',
'theater_count': 21}, {'_id': 'DC', 'theater_count': 3}, {'_id': 'DE',
'theater_count': 5}, {'_id': 'FL', 'theater_count': 111}]

```

1.2.3 Question 3:

How many movies are in the Comedy genre?

```
[13]: comedy_movies_count = mflixdb.movies.count_documents({"genres": {"$in": ↪
    ["Comedy"]}})
print(dumps(comedy_movies_count, indent=4))
```

6532

1.2.4 Question 4:

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

```
[21]: runtimes = mflixdb.movies.aggregate([
    {"$sort": {"runtime": -1}},
    {"$limit": 1},
    {"$project": {"_id": 0, "title": 1, "genres": 1, "runtime": 1}}
])
print(dumps(longest, indent=4))

{
    "genres": [
        "Action",
        "Adventure",
        "Drama"
    ],
    "runtime": 1256,
    "title": "Centennial"
}
```

1.2.5 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.

```
[44]: by_rating = mflixdb.movies.aggregate([
    {"$match": {"year": {"$gt": 2010}, "tomatoes.viewer.rating": {"$gte": 3.
    ↪0}}},
    {"$sort": {"tomatoes.viewer.rating": -1}},
    {"$project": {"_id": 0, "title": 1, "rating": "$tomatoes.viewer.rating"}},
])
by_rating = list(by_rating)
print(by_rating[:10])
```

```
[{'title': 'Scooby-Doo! Mask of the Blue Falcon', 'rating': 5}, {'title': 'The Dead and the Living', 'rating': 5}, {'title': 'My Last Year with the Nuns', 'rating': 5}, {'title': 'Silenced', 'rating': 5}, {'title': 'Rosemary's Baby', 'rating': 5}, {'title': 'Crocodile Gennadiy', 'rating': 5}, {'title': 'The Dancer', 'rating': 5}, {'title': 'Beethoven's Christmas Adventure', 'rating': 5}, {'title': 'Scattered Cloud', 'rating': 5}, {'title': 'The Color of Rain', 'rating': 5}]
```

1.2.6 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.

```
[47]: count_police = mflixdb.movies.aggregate([
    {"$match": {"plot": {"$regex": "police"}}, },
    {"$group": {"_id": "$year", "movie_count": {"$sum": 1}}}, ,
    {"$sort": {"_id": 1}}
])
count_police = list(count_police)
print(count_police)
```

[{"_id": 1913, "movie_count": 1}, {"_id": 1934, "movie_count": 1}, {"_id": 1935, "movie_count": 1}, {"_id": 1944, "movie_count": 1}, {"_id": 1947, "movie_count": 1}, {"_id": 1948, "movie_count": 1}, {"_id": 1949, "movie_count": 1}, {"_id": 1950, "movie_count": 2}, {"_id": 1951, "movie_count": 2}, {"_id": 1957, "movie_count": 1}, {"_id": 1958, "movie_count": 1}, {"_id": 1959, "movie_count": 2}, {"_id": 1960, "movie_count": 1}, {"_id": 1961, "movie_count": 1}, {"_id": 1963, "movie_count": 2}, {"_id": 1965, "movie_count": 2}, {"_id": 1966, "movie_count": 1}, {"_id": 1967, "movie_count": 2}, {"_id": 1968, "movie_count": 1}, {"_id": 1969, "movie_count": 2}, {"_id": 1970, "movie_count": 2}, {"_id": 1971, "movie_count": 2}, {"_id": 1972, "movie_count": 3}, {"_id": 1973, "movie_count": 6}, {"_id": 1974, "movie_count": 2}, {"_id": 1975, "movie_count": 5}, {"_id": 1976, "movie_count": 1}, {"_id": 1977, "movie_count": 1}, {"_id": 1978, "movie_count": 5}, {"_id": 1979, "movie_count": 3}, {"_id": 1980, "movie_count": 4}, {"_id": 1981, "movie_count": 6}, {"_id": 1982, "movie_count": 3}, {"_id": 1983, "movie_count": 6}, {"_id": 1984, "movie_count": 6}, {"_id": 1985, "movie_count": 7}, {"_id": 1986, "movie_count": 3}, {"_id": 1987, "movie_count": 4}, {"_id": 1988, "movie_count": 3}, {"_id": 1989, "movie_count": 9}, {"_id": 1990, "movie_count": 5}, {"_id": 1991, "movie_count": 6}, {"_id": 1992, "movie_count": 14}, {"_id": 1993, "movie_count": 7}, {"_id": 1994, "movie_count": 6}, {"_id": 1995, "movie_count": 11}, {"_id": 1996, "movie_count": 9}, {"_id": 1997, "movie_count": 10}, {"_id": 1998, "movie_count": 15}, {"_id": 1999, "movie_count": 12}, {"_id": 2000, "movie_count": 14}, {"_id": 2001, "movie_count": 14}, {"_id": 2002, "movie_count": 12}, {"_id": 2003, "movie_count": 13}, {"_id": 2004, "movie_count": 13}, {"_id": 2005, "movie_count": 14}, {"_id": 2006, "movie_count": 24}, {"_id": 2007, "movie_count": 11}, {"_id": 2008, "movie_count": 12}, {"_id": 2009, "movie_count": 11}, {"_id": 2010, "movie_count": 15}, {"_id": 2011, "movie_count": 20}, {"_id": 2012, "movie_count": 17}, {"_id": 2013, "movie_count": 19}, {"_id": 2014, "movie_count": 17}, {"_id": 2015, "movie_count": 2}, {"_id": "1988è", "movie_count": 1}]

1.2.7 Question 7:

What is the average number of imbd votes per year for movies released between 1970 and 2000 (inclusive)? Make sure the results are ordered by year.

```
[48]: avg_votes = mflixdb.movies.aggregate([
    {"$match": {"year": {"$gte": 1970, "$lte": 2000}}},
    {"$group": {"_id": {"release year": "$year"}, "Avg Votes": {"$avg": "$imdb.
    ↴votes"}}},
    {"$sort": {"_id": 1}}
])
avg_votes = list(avg_votes)
print(avg_votes)
```

```
[{'_id': {'release year': 1970}, 'Avg Votes': 4786.925}, {'_id': {'release
year': 1971}, 'Avg Votes': 8528.462264150943}, {'_id': {'release year': 1972},
'Avg Votes': 13582.685950413223}, {'_id': {'release year': 1973}, 'Avg Votes':
14478.785714285714}, {'_id': {'release year': 1974}, 'Avg Votes': 17602.0},
{'_id': {'release year': 1975}, 'Avg Votes': 19615.00934579439}, {'_id':
{'release year': 1976}, 'Avg Votes': 14259.76724137931}, {'_id': {'release
year': 1977}, 'Avg Votes': 14210.393442622952}, {'_id': {'release year': 1978},
'Avg Votes': 9992.5703125}, {'_id': {'release year': 1979}, 'Avg Votes':
15729.419847328245}, {'_id': {'release year': 1980}, 'Avg Votes':
16487.155688622755}, {'_id': {'release year': 1981}, 'Avg Votes':
12193.797619047618}, {'_id': {'release year': 1982}, 'Avg Votes':
15729.898305084746}, {'_id': {'release year': 1983}, 'Avg Votes':
15521.664596273293}, {'_id': {'release year': 1984}, 'Avg Votes':
19255.838383838385}, {'_id': {'release year': 1985}, 'Avg Votes':
15590.121693121693}, {'_id': {'release year': 1986}, 'Avg Votes':
19503.78947368421}, {'_id': {'release year': 1987}, 'Avg Votes':
18074.545045045044}, {'_id': {'release year': 1988}, 'Avg Votes':
16892.521912350596}, {'_id': {'release year': 1989}, 'Avg Votes':
18821.879310344826}, {'_id': {'release year': 1990}, 'Avg Votes':
22580.324444444443}, {'_id': {'release year': 1991}, 'Avg Votes':
19933.7731092437}, {'_id': {'release year': 1992}, 'Avg Votes':
19883.774074074074}, {'_id': {'release year': 1993}, 'Avg Votes':
23896.525547445257}, {'_id': {'release year': 1994}, 'Avg Votes':
38413.69836065574}, {'_id': {'release year': 1995}, 'Avg Votes':
25700.217741935485}, {'_id': {'release year': 1996}, 'Avg Votes':
18431.299754299755}, {'_id': {'release year': 1997}, 'Avg Votes':
26613.216400911162}, {'_id': {'release year': 1998}, 'Avg Votes':
23481.639376218325}, {'_id': {'release year': 1999}, 'Avg Votes':
30182.335922330098}, {'_id': {'release year': 2000}, 'Avg Votes':
25951.459552495697}]
```

1.2.8 Question 8:

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

```
[51]: languages = mflixdb.movies.aggregate([
    {"$unwind": "$languages"},
    {"$group": {"_id": "$languages"}},
```

)

```
languages = [dic["_id"] for dic in languages]
print(languages)
```

```
['Ladakhi', 'Yiddish', 'Klingon', 'Nyanja', 'Oriya', 'Breton', 'Aymara',
'Kannada', 'Cornish', 'Kazakh', 'Manipuri', 'Ladino', 'Hassanya', 'Scottish
Gaelic', 'Nenets', 'Kirghiz', 'Romanian', 'Tulu', 'Luxembourgish', 'Southern
Sotho', 'Gallegan', 'Pawnee', 'Japanese Sign Language', 'Russian Sign Language',
'Nama', 'Nepali', 'Polish', 'Shanxi', 'Ryukyuan', 'Dzongkha', 'Acholi', 'Irish',
'German Sign Language', 'Kurdish', 'Hungarian', 'Faroeese', 'Latvian', 'Italian',
'Maori', 'Ungwatsi', 'Karajè', 'Min Nan', 'Mapudungun', 'Urdu', 'Georgian',
'Slovenian', 'Tswana', 'Norse', 'Hebrew', 'Uzbek', 'Basque', 'Egyptian
(Ancient)', 'Occitan', 'Fulah', 'Tatar', 'Dutch', 'Indian Sign Language',
'Dyula', 'Tonga', 'Kabuverdianu', 'Inupiaq', 'Assamese', 'Swedish', 'Quechua',
'Maltese', 'Shanghainese', 'More', 'Spanish Sign Language', 'Thai', 'Quenya',
'Croatian', 'English', 'Mohawk', 'Scanian', 'Sindarin', 'Wolof', 'Macedonian',
'Cheyenne', 'Portuguese', 'Burmese', 'Japanese', 'Chechen', 'Nyaneka',
'Turkish', 'Maya', 'Visayan', 'Algonquin', 'Romany', 'Samoan', 'Fur', 'Sign
Languages', 'Cree', 'Ancient (to 1453)', 'Jola-Fonyi', 'Peul', 'Kuna', 'Hindi',
'Mandarin', 'Sinhalese', 'Purepecha', 'Serbo-Croatian', 'Albanian', 'Telugu',
'Bambara', 'Apache languages', 'Low German', 'Tok Pisin', 'Shona', 'Turkmen',
'Norwegian', 'Kikuyu', 'Creoles and pidgins', 'Mongolian', 'Gujarati',
'Aidoukrou', 'Uighur', 'Berber languages', 'Bengali', 'Aboriginal',
'Rajasthani', 'Xhosa', 'Spanish', 'Afrikaans', 'Tamil', 'Greenlandic',
'Tarahumara', 'British Sign Language', 'Catalan', 'Mende', 'Assyrian Neo-
Aramaic', 'Old', 'Slovak', 'Old English', 'Ukrainian', 'Tupi', 'Amharic',
'Navajo', 'Latin', 'Finnish', 'Zulu', 'Sardinian', 'Gumatj', 'Shoshoni',
'Persian', 'Dinka', 'Washoe', 'Hmong', 'Creole', 'Somali', 'Swiss German',
'Tuvanian', 'Lingala', 'Filipino', 'Konkani', 'Armenian', 'Athapascan
languages', 'Flemish', 'Azerbaijani', 'Russian', 'Swahili', 'Guarani',
'Sanskrit', 'Scots', 'Vietnamese', 'German', 'Songhay', 'Lao', 'Arapaho',
'Chinese', 'Tzotzil', 'Malayalam', 'Inuktitut', 'Icelandic', 'Esperanto',
'Tajik', 'French Sign Language', 'Hawaiian', 'Kinyarwanda', 'Awadhi', 'Welsh',
'Czech', 'Sioux', 'Hokkien', 'Brazilian Sign Language', 'Balinese', 'Tagalog',
'Frisian', 'Estonian', 'Korean Sign Language', 'Korean', 'Middle English',
'Abkhazian', 'Syriac', 'Dari', 'Haitian', 'Panjabi', 'Tibetan', 'Eastern
Frisian', 'Cantonese', 'Tigrigna', 'Sicilian', 'Aramaic', 'Belarusian', 'Crow',
'Masai', 'Serbian', 'French', 'Marathi', 'Bosnian', 'Corsican', 'American Sign
Language', 'Lithuanian', 'Bhojpuri', 'Bulgarian', 'Greek', 'Malay', 'Ibo',
'Indonesian', 'Saami', 'Pushto', 'Mari', 'Danish', 'Yoruba', 'Arabic', 'Kabyle',
'Ewe', 'Hakka', 'North American Indian', 'Nahuatl', 'Neapolitan', 'Khmer']
```



Tutorials ▾

Exercises ▾

Services ▾



Sign Up

Log in



JAVA

PHP

HOW TO

W3.CSS

C

C++

C#

BOOTSTRAP

REACT

MYSQ

AVL Trees

[Next ➤](#)

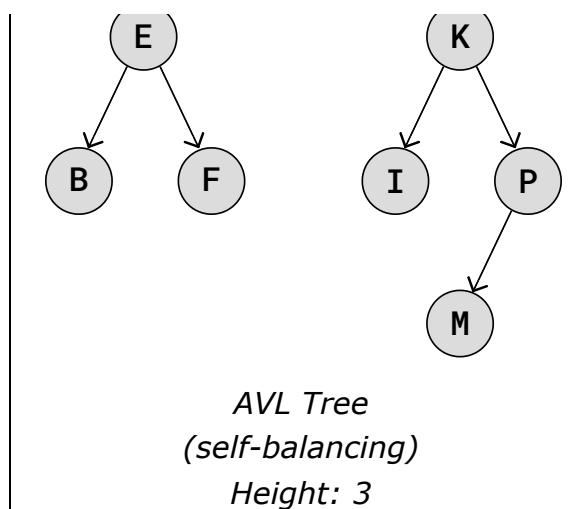
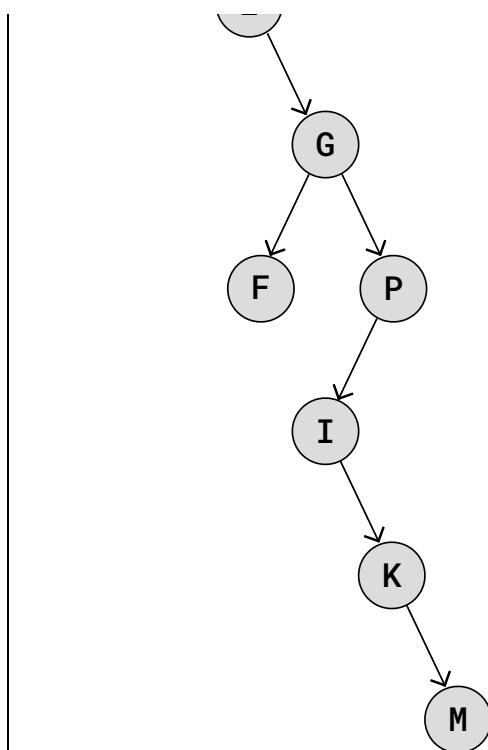
Type of Binary Search Tree named after two Soviet inventors Georgy I Evgenii Landis who invented the AVL Tree in 1962.

Self-balancing, which means that the tree height is kept to a minimum so time is guaranteed for searching, inserting and deleting nodes, with $O(\log n)$.

The main difference between a regular Binary Search Tree and an AVL Tree is that AVL Tree performs rotations in addition, to keep the tree balance.

A Binary Search Tree is in balance when the difference in height between left and right subtrees is less than 2.

By keeping balance, the AVL Tree ensures a minimum tree height, which means that search, insert, and delete operations can be done really fast.



The two trees above are both Binary Search Trees, they have the same nodes, and the same in-order traversal (alphabetical), but the height is very different because the AVL Tree has balanced itself.

Step through the building of an AVL Tree in the animation below to see how the balance factors are updated, and how rotation operations are done when required to restore the balance.



Insert C

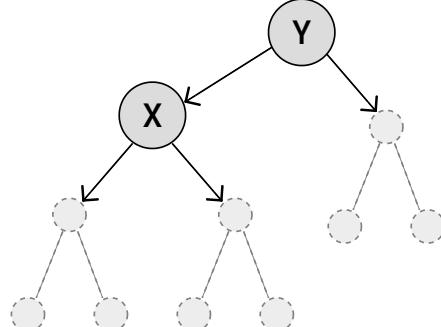
Continue reading to learn more about how the balance factor is calculated, how rotation operations are done, and how AVL Trees can be implemented.

Left and Right Rotations

To restore balance in an AVL Tree, left or right rotations are done, or a combination of left and right rotations.

The previous animation shows one specific left rotation, and one specific right rotation.

But in general, left and right rotations are done like in the animation below.



Notice how the subtree changes its parent. Subtrees change parent in this way during rotation to maintain the correct in-order traversal, and to maintain the BST property that the left child is less than the right child, for all nodes in the tree.



The Balance Factor

A node's balance factor is the difference in subtree heights.

The subtree heights are stored at each node for all nodes in an AVL Tree, and the balance factor is calculated based on its subtree heights to check if the tree has become out of balance.

The height of a subtree is the number of edges between the root node of the subtree and the leaf node farthest down in that subtree.

The **Balance Factor** (BF) for a node (X) is the difference in height between its right and left subtrees.

$$BF(X) = \text{height}(\text{rightSubtree}(X)) - \text{height}(\text{leftSubtree}(X))$$

Balance factor values

- 0: The node is in balance.
- more than 0: The node is "right heavy".
- less than 0: The node is "left heavy".

If the balance factor is less than -1, or more than 1, for one or more nodes in the tree, the tree is considered not in balance, and a rotation operation is needed to restore balance.

Let's take a closer look at the different rotation operations that an AVL Tree can do to regain balance.

The Four "out-of-balance" Cases

When the balance factor of just one node is less than -1, or more than 1, the tree is regarded as out of balance, and a rotation is needed to restore balance.

**Balance**

Left-Left (LL)	The unbalanced node and its left child node are both left-heavy.	A single right rotation.
Right-Right (RR)	The unbalanced node and its right child node are both right-heavy.	A single left rotation.
Left-Right (LR)	The unbalanced node is left heavy, and its left child node is right heavy.	First do a left rotation on the left child node, then do a right rotation on the unbalanced node.
Right-Left (RL)	The unbalanced node is right heavy, and its right child node is left heavy.	First do a right rotation on the right child node, then do a left rotation on the unbalanced node.

See animations and explanations of these cases below.

The Left-Left (LL) Case

The node where the unbalance is discovered is left heavy, and the node's left child node is also left heavy.

When this LL case happens, a single right rotation on the unbalanced node is enough to restore balance.

Step through the animation below to see the LL case, and how the balance is restored by a single right rotation.



Insert D

As you step through the animation above, two LL cases happen:

1. When D is added, the balance factor of Q becomes -2, which means the tree is unbalanced. This is an LL case because both the unbalance node Q and its left child node P are left heavy (negative balance factors). A single right rotation at node Q restores the tree balance.
2. After nodes L, C, and B are added, P's balance factor is -2, which means the tree is out of balance. This is also an LL case because both the unbalanced node P and its left child node D are left heavy. A single right rotation restores the balance.

Note: The second time the LL case happens in the animation above, a right rotation is done, and L goes from being the right child of D to being the left child of P. Rotations are done like that to keep the correct in-order traversal ('B, C, D, L, P, Q' in the animation above). Another reason for changing parent when a rotation is done is to keep the BST property, that the left child is always lower than the node, and that the right child always higher.

The Right-Right (RR) Case

A Right-Right case happens when a node is unbalanced and right heavy, and the right child node is also right heavy.

A single left rotation at the unbalanced node is enough to restore balance in the RR case.



The screenshot shows a section of the W3Schools website dedicated to Data Structures and Algorithms (DSA). The page title is "DSA AVL Trees". The main content is an AVL tree diagram. A node labeled "B" is highlighted with a red circle. At the bottom left of the tree area, there is a button with the text "Insert D". The navigation bar at the top includes links for JAVA, PHP, HOW TO, W3.CSS, C, C++, C#, BOOTSTRAP, REACT, and MYSQL.

The RR case happens two times in the animation above:

1. When node D is inserted, A becomes unbalanced, and both A and B are right heavy. A left rotation at node A restores the tree balance.
2. After nodes E, C and F are inserted, node B becomes unbalanced. This is an RR case because both node B and its right child node D are right heavy. A left rotation restores the tree balance.

The Left-Right (LR) Case

The Left-Right case is when the unbalanced node is left heavy, but its left child node is right heavy.

In this LR case, a left rotation is first done on the left child node, and then a right rotation is done on the original unbalanced node.

Step through the animation below to see how the Left-Right case can happen, and how the rotation operations are done to restore balance.



E

Insert K

As you are building the AVL Tree in the animation above, the Left-Right case happens 2 times, and rotation operations are required and done to restore balance:

1. When K is inserted, node Q gets unbalanced with a balance factor of -2, so it is left heavy, and its left child E is right heavy, so this is a Left-Right case.
2. After nodes C, F, and G are inserted, node K becomes unbalanced and left heavy, with its left child node E right heavy, so it is a Left-Right case.

The Right-Left (RL) Case

The Right-Left case is when the unbalanced node is right heavy, and its right child node is left heavy.

In this case we first do a right rotation on the unbalanced node's right child, and then we do a left rotation on the unbalanced node itself.

Step through the animation below to see how the Right-Left case can occur, and how rotations are done to restore the balance.



F

Insert B

After inserting node B, we get a Right-Left case because node A becomes unbalanced and right heavy, and its right child is left heavy. To restore balance, a right rotation is first done on node F, and then a left rotation is done on node A.

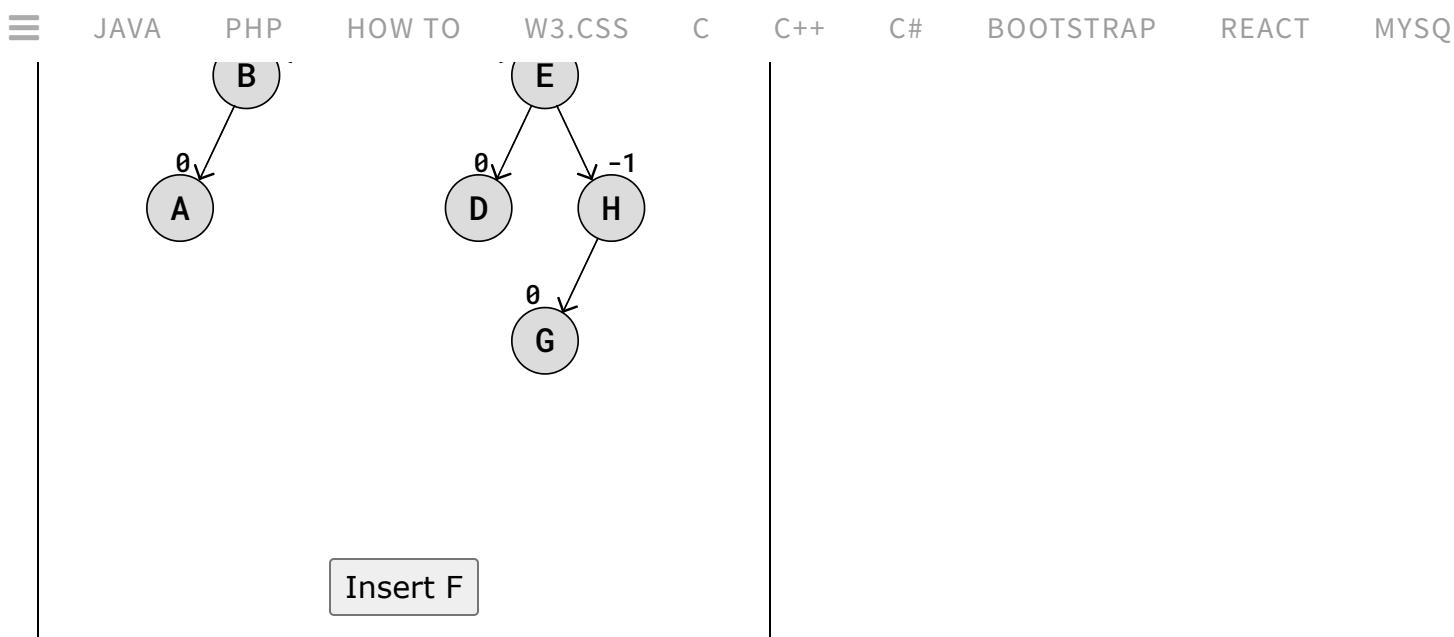
The next Right-Left case occurs after nodes G, E, and D are added. This is a Right-Left case because B is unbalanced and right heavy, and its right child F is left heavy. To restore balance, a right rotation is first done on node F, and then a left rotation is done on node B.

Retracing in AVL Trees

After inserting or deleting a node in an AVL tree, the tree may become unbalanced. To find out if the tree is unbalanced, we need to update the heights and recalculate the balance factors of all ancestor nodes.

This process, known as retracing, is handled through recursion. As the recursive calls propagate back to the root after an insertion or deletion, each ancestor node's height is updated and the balance factor is recalculated. If any ancestor node is found to have a balance factor outside the range of -1 to 1, a rotation is performed at that node to restore the tree's balance.

In the simulation below, after inserting node F, the nodes C, E and H are all unbalanced, but since retracing works through recursion, the unbalance at node H is discovered and fixed first, which in this case also fixes the unbalance in nodes E and C.



After node F is inserted, the code will retrace, calculating balancing factors as it propagates back up towards the root node. When node H is reached and the balancing factor -2 is calculated, a right rotation is done. Only after this rotation is done, the code will continue to retrace, calculating balancing factors further up on ancestor nodes E and C.

Because of the rotation, balancing factors for nodes E and C stay the same as before node F was inserted.

AVL Insert Node Implementation

This code is based on the BST implementation on the previous page, for inserting nodes.

There is only one new attribute for each node in the AVL tree compared to the BST, and that is the height, but there are many new functions and extra code lines needed for the AVL Tree implementation because of how the AVL Tree rebalances itself.

The implementation below builds an AVL tree based on a list of characters, to create the AVL Tree in the simulation above. The last node to be inserted 'F', also triggers a right rotation, just like in the simulation above.

Example



JAVA PHP HOW TO W3.CSS C C++ C# BOOTSTRAP REACT MYSQ

```
self.data = data
self.left = None
self.right = None
self.height = 1

def getHeight(node):
    if not node:
        return 0
    return node.height

def getBalance(node):
    if not node:
        return 0
    return getHeight(node.left) - getHeight(node.right)

def rightRotate(y):
    print('Rotate right on node',y.data)
    x = y.left
    T2 = x.right
    x.right = y
    y.left = T2
    y.height = 1 + max(getHeight(y.left), getHeight(y.right))
    x.height = 1 + max(getHeight(x.left), getHeight(x.right))
    return x

def leftRotate(x):
    print('Rotate left on node',x.data)
    y = x.right
    T2 = y.left
    y.left = x
    x.right = T2
    x.height = 1 + max(getHeight(x.left), getHeight(x.right))
    y.height = 1 + max(getHeight(y.left), getHeight(y.right))
    return y

def insert(node, data):
    if not node:
        return TreeNode(data)
```



```
# Update the balance factor and balance the tree
node.height = 1 + max(getHeight(node.left), getHeight(node.right))
balance = getBalance(node)

# Balancing the tree
# Left Left
if balance > 1 and getBalance(node.left) >= 0:
    return rightRotate(node)

# Left Right
if balance > 1 and getBalance(node.left) < 0:
    node.left = leftRotate(node.left)
    return rightRotate(node)

# Right Right
if balance < -1 and getBalance(node.right) <= 0:
    return leftRotate(node)

# Right Left
if balance < -1 and getBalance(node.right) > 0:
    node.right = rightRotate(node.right)
    return leftRotate(node)

return node

def inOrderTraversal(node):
    if node is None:
        return
    inOrderTraversal(node.left)
    print(node.data, end=", ")
    inOrderTraversal(node.right)

# Inserting nodes
root = None
letters = ['C', 'B', 'E', 'A', 'D', 'H', 'G', 'F']
for letter in letters:
    root = insert(root, letter)
```

[Run Example »](#)

AVL Delete Node Implementation

When deleting a node that is not a leaf node, the AVL Tree requires the `minValueNode()` function to find a node's next node in the in-order traversal. This is the same as when deleting a node in a Binary Search Tree, as explained on the previous page.

To delete a node in an AVL Tree, the same code to restore balance is needed as for the code to insert a node.

Example

Python:

```
def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def delete(node, data):
    if not node:
        return node

    if data < node.data:
        node.left = delete(node.left, data)
    elif data > node.data:
        node.right = delete(node.right, data)
    else:
        if node.left is None:
            temp = node.right
            node = None
            return temp
        else:
            current = minValueNode(node.left)
            node.data = current.data
            node.left = delete(node.left, current.data)
```



```
temp = minValueNode(node.right)
node.data = temp.data
node.right = delete(node.right, temp.data)

if node is None:
    return node

# Update the balance factor and balance the tree
node.height = 1 + max(getHeight(node.left), getHeight(node.right))
balance = getBalance(node)

# Balancing the tree
# Left Left
if balance > 1 and getBalance(node.left) >= 0:
    return rightRotate(node)

# Left Right
if balance > 1 and getBalance(node.left) < 0:
    node.left = leftRotate(node.left)
    return rightRotate(node)

# Right Right
if balance < -1 and getBalance(node.right) <= 0:
    return leftRotate(node)

# Right Left
if balance < -1 and getBalance(node.right) > 0:
    node.right = rightRotate(node.right)
    return leftRotate(node)

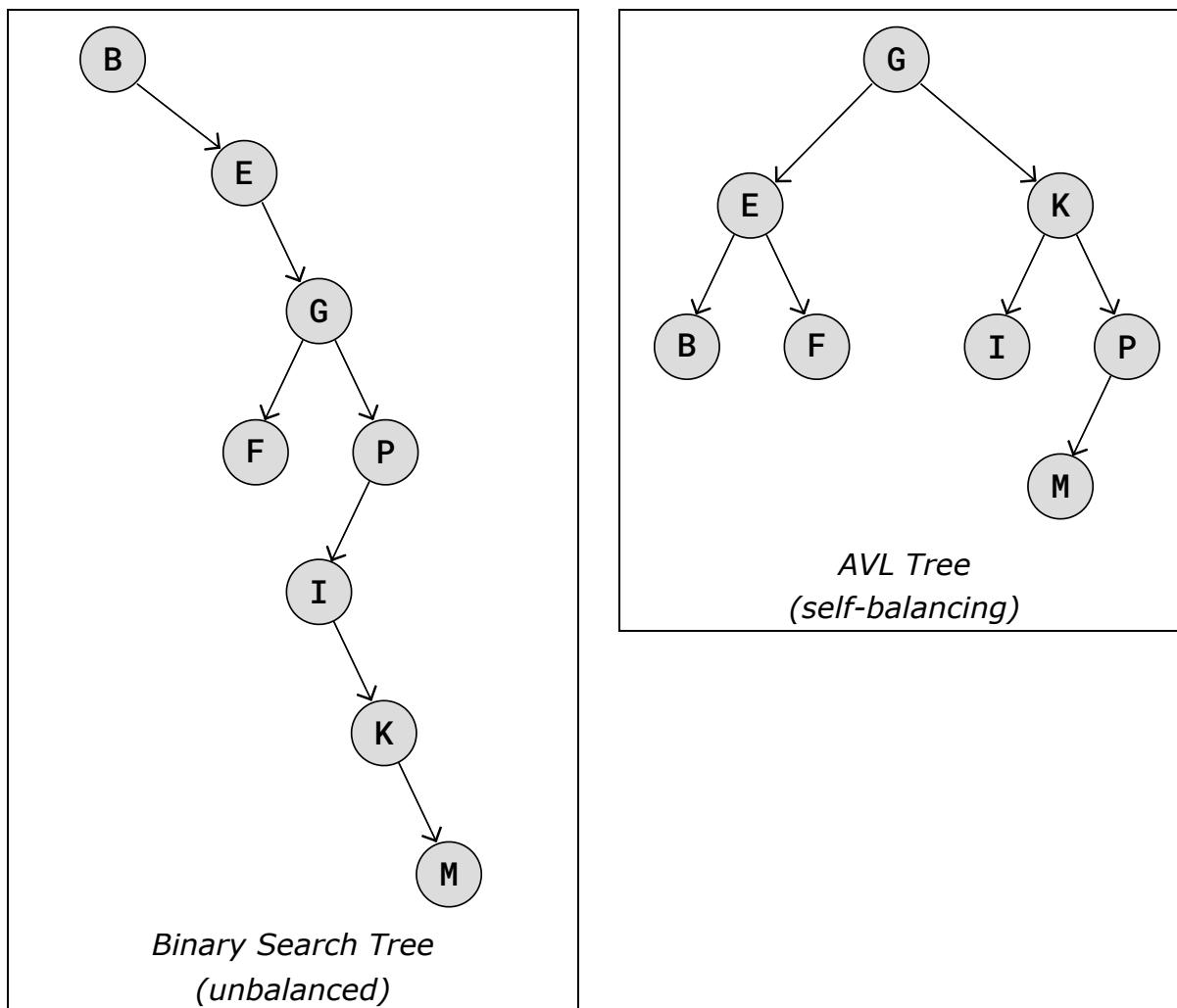
return node
```

[Run Example »](#)



all nodes except 1 must be compared. But searching for "M" in the AVL Tree below only requires us to visit 4 nodes.

So in worst case, algorithms like search, insert, and delete must run through the whole height of the tree. This means that keeping the height (h) of the tree low, like we do using AVL Trees, gives us a lower runtime.



See the comparison of the time complexities between Binary Search Trees and AVL Trees below, and how the time complexities relate to the height (h) of the tree, and the number of nodes (n) in the tree.

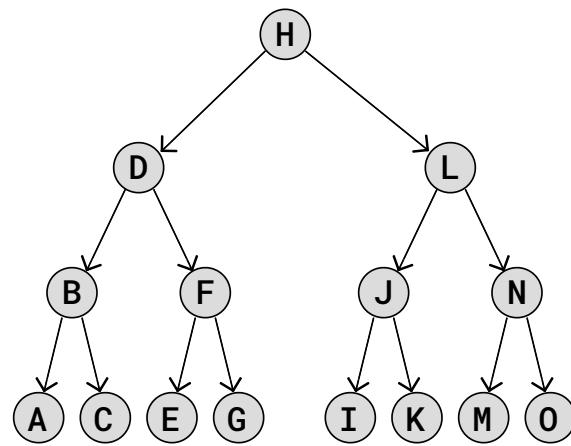
- The **BST** is not self-balancing. This means that a BST can be very unbalanced, almost like a long chain, where the height is nearly the same as the number of nodes. This makes operations like searching, deleting and inserting nodes slow, with time complexity $O(h) = O(n)$.



$O(\log n)$ Explained

The fact that the time complexity is $O(h) = O(\log n)$ for search, insert, and delete on an AVL Tree with height h and nodes n can be explained like this:

Imagine a perfect Binary Tree where all nodes have two child nodes except on the lowest level, like the AVL Tree below.



The number of nodes on each level in such an AVL Tree are:

$$1, 2, 4, 8, 16, 32, \dots$$

Which is the same as:

$$2^0, 2^1, 2^2, 2^3, 2^4, 2^5, \dots$$

To get the number of nodes n in a perfect Binary Tree with height $h = 3$, we can add the number of nodes on each level together:

$$n_3 = 2^0 + 2^1 + 2^2 + 2^3 = 15$$

Which is actually the same as:

$$n_3 = 2^4 - 1 = 15$$

And this is actually the case for larger trees as well! If we want to get the number of nodes n in a tree with height $h = 5$ for example, we find the number of nodes like this:

$$n_5 = 2^6 - 1 = 63$$



$$n_h = 2^{h+1} - 1$$

Note: The formula above can also be found by calculating the sum of the geometric series $2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n$

We know that the time complexity for searching, deleting, or inserting a node in an AVL tree is $O(h)$, but we want to argue that the time complexity is actually $O(\log(n))$, so we need to find the height h described by the number of nodes n :

$$\begin{aligned} n &= 2^{h+1} - 1 \\ n + 1 &= 2^{h+1} \\ \log_2(n + 1) &= \log_2(2^{h+1}) \\ h &= \log_2(n + 1) - 1 \end{aligned}$$

$$O(h) = O(\log n)$$

How the last line above is derived might not be obvious, but for a Binary Tree with a lot of nodes (big n), the "+1" and "-1" terms are not important when we consider time complexity. For more details on how to calculate the time complexity using Big O notation, see [this page](#).

The math above shows that the time complexity for search, delete, and insert operations on an AVL Tree $O(h)$, can actually be expressed as $O(\log n)$, which is fast, a lot faster than the time complexity for BSTs which is $O(n)$.

DSA Exercises

Test Yourself With Exercises

Exercise:



Tutorials ▾

Exercises ▾

Services ▾



Sign Up

Log in



JAVA

PHP

HOW TO

W3.CSS

C

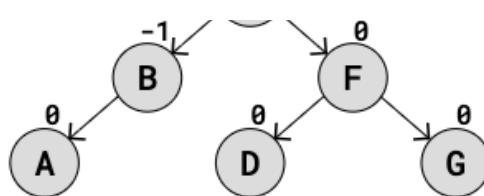
C++

C#

BOOTSTRAP

REACT

MYSQ



What is the balance factor?

The balance factor is the difference between each node's left and right subtree .

[Submit Answer »](#)

[Start the Exercise](#)

[◀ Previous](#)

[Next ▶](#)

Track your progress - it's free!

[Sign Up](#)

[Log in](#)



Tutorials ▾

Exercises ▾

Services ▾



Sign Up

Log in



JAVA

PHP

HOW TO

W3.CSS

C

C++

C#

BOOTSTRAP

REACT

MYSQ





Tutorials ▾

Exercises ▾

Services ▾



Sign Up

Log in



JAVA

PHP

HOW TO

W3.CSS

C

C++

C#

BOOTSTRAP

REACT

MYSQ



PLUS

SPACES

GET CERTIFIED

FOR TEACHERS

FOR BUSINESS

CONTACT US

Top Tutorials

[HTML Tutorial](#)[CSS Tutorial](#)[JavaScript Tutorial](#)



Tutorials ▾

Exercises ▾

Services ▾



Sign Up

Log in



JAVA

PHP

HOW TO

W3.CSS

C

C++

C#

BOOTSTRAP

REACT

MYSQ

Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
Angular Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate


[FORUM](#) [ABOUT](#) [ACADEMY](#)

W3Schools is optimized for learning and training. Examples might be simplified to improve reading and learning.

Tutorials, references, and examples are constantly reviewed to avoid errors, but we cannot warrant full correctness

of all content. While using W3Schools, you agree to have read and accepted our [terms of use](#), [cookie and privacy policy](#).

Copyright 1999-2025 by Refsnes Data. All Rights Reserved. W3Schools is Powered by W3.CSS.

[Join Taro Premium](#)

Taro > Interview Insights >

Google > Explain linked lists, their types, common operations, advantages/disadvantages, and use cases.

Explain linked lists, their types, common operations, advantages/disadvantages, and use cases.

Google Hard

⌚ 2 years ago

Let's explore the concept of linked lists.

1. **Definition:** Explain what a linked list is. What are its key characteristics and how does it differ from an array? Provide a clear and concise explanation, suitable for someone who might be familiar with basic data structures but not necessarily an expert in linked lists. For example, you could start by stating that a linked list is a linear data structure where elements are not stored in contiguous memory locations.
2. **Types:** Describe the different types of linked lists (singly, doubly, circular). Illustrate each type with a simple diagram or example. For instance, for a singly linked list, you might describe how each node contains data and a pointer to the next node, but not to the previous node. For a doubly linked list, explain the presence of pointers to both the next and previous nodes. For a circular linked list, highlight the fact that the last node points back to the first node, forming a cycle.
3. **Operations:** Implement the following operations for a singly linked list:
 - `insertAtHead(data)`: Inserts a new node with the given data at the beginning of the list.
 - `insertAtTail(data)`: Inserts a new node with the given data at the end of the list.
 - `deleteWithValue(data)`: Deletes the first node with the given data from the list.
 - `printList()`: Prints the data of each node in the list in order.

Provide code examples in your preferred programming language. Explain the time complexity of each of these operations. For example, insertion at the head should be O(1), while `deleteWithValue` could be O(n) in the worst case.

4. **Advantages and Disadvantages:**

Discuss the advantages and disadvantages of using linked lists compared to arrays. Consider factors such as memory usage, insertion/deletion speed, and random access. For example, explain how linked lists offer dynamic resizing, unlike arrays, but lack the constant-time random access that arrays provide.

5. Real-world Use Cases: Describe some real-world scenarios where linked lists are commonly used. Consider applications such as implementing stacks and queues, representing graphs, or managing dynamic memory allocation. Provide specific examples, explaining why a linked list might be a suitable choice in each case.

Sample Answer

Linked List

Let's explore the concept of linked lists.

1. Definition

A linked list is a linear data structure in which elements are not stored in contiguous memory locations. Instead, each element, called a node, contains a data field and a reference (or pointer) to the next node in the sequence. This structure allows for dynamic memory allocation and efficient insertion or deletion of elements.

Key Characteristics:

- **Dynamic Size:** Linked lists can grow or shrink dynamically during runtime, as memory is allocated for each new node.
- **Non-Contiguous Memory:** Nodes are not stored in adjacent memory locations, which differs from arrays.
- **Nodes:** Each element is a node containing data and a pointer to the next node.
- **Head:** The first node in the list is called the head. If the list is empty, the head is `null`.
- **Tail:** The last node in the list has its `next` pointer set to `null`.

Difference from Arrays:

Feature	Linked List	Array
Memory	Non-contiguous	Contiguous
Size	Dynamic	Static (or requires resizing)
Insertion/Deletion	Efficient ($O(1)$ if node is known)	Inefficient ($O(n)$ due to shifting)
Random Access	Not supported ($O(n)$ traversal required)	Supported ($O(1)$)

2. Types

Singly Linked List

Each node contains data and a pointer to the next node. Traversal is possible only in one direction.

```
Data: A | Next: --> | Data: B | Next: --> | Data: C | Next: null
```

Doubly Linked List

Each node contains data and two pointers: one to the next node and one to the previous node. This allows traversal in both directions.

```
Prev: null | Data: A | Next: --> | Prev: <-- | Data: B | Next: --> | Prev: <-- | D;
```

Circular Linked List

The last node points back to the first node, forming a cycle. This type of list has no end.

```
Data: A | Next: --> | Data: B | Next: --> | Data: C | Next: --> | (points back to /
```

3. Operations

Here's an implementation of the requested operations for a singly linked list in Python:

```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None  
  
class LinkedList:  
    def __init__(self):  
        self.head = None  
  
    def insertAtHead(self, data):  
        new_node = Node(data)  
        new_node.next = self.head  
        self.head = new_node  
  
    def insertAtTail(self, data):  
        new_node = Node(data)  
        if self.head is None:  
            self.head = new_node  
            return  
        last = self.head  
        while (last.next):  
            last = last.next  
        last.next = new_node  
  
    def deleteWithValue(self, data):  
        current = self.head  
        if current is not None and current.data == data:  
            self.head = current.next  
            return  
        prev = None  
        while (current is not None and current.data != data):  
            prev = current  
            current = current.next  
        if current is None:  
            return  
        prev.next = current.next  
  
    def printList(self):
```

```

temp = self.head
while (temp):
    print(temp.data, end=" ")
    temp = temp.next
print()

# Example Usage
linked_list = LinkedList()
linked_list.insertAtHead(3)
linked_list.insertAtHead(2)
linked_list.insertAtTail(4)
linked_list.printList() # Output: 2 3 4
linked_list.deleteWithValue(3)
linked_list.printList() # Output: 2 4

```

Time Complexity Analysis:

- `insertAtHead(data)` : O(1) - Constant time, as it only involves creating a new node and updating the head.
- `insertAtTail(data)` : O(n) - Linear time, as it may require traversing the list to find the last node.
- `deleteWithValue(data)` : O(n) - Linear time, as it may require traversing the list to find the node to delete.
- `printList()` : O(n) - Linear time, as it requires traversing the entire list to print each node.

4. Advantages and Disadvantages

Advantages of Linked Lists:

- **Dynamic Size:** Can grow or shrink as needed.
- **Efficient Insertion/Deletion:** Insertion and deletion at known positions are O(1).
- **Memory Efficiency:** Uses memory only for the elements it contains.

Disadvantages of Linked Lists:

- **No Random Access:** Accessing an element requires traversing from the head, $O(n)$.
- **Extra Memory Overhead:** Requires extra memory to store pointers.
- **Cache Inefficiency:** Nodes are scattered in memory, reducing cache performance.

Comparison Table:

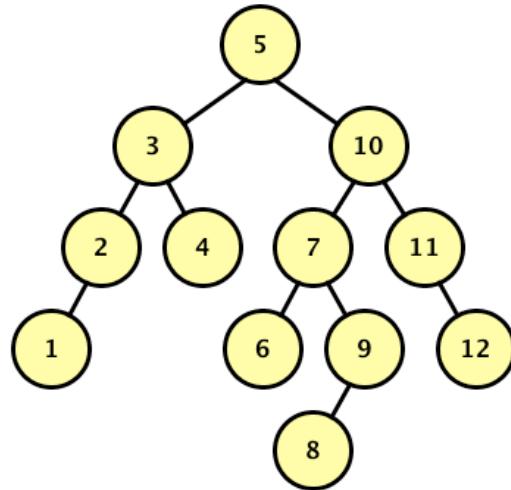
Feature	Linked List	Array
Memory Usage	Dynamic	Static
Insertion/Deletion	Efficient	Costly
Random Access	No	Yes
Memory Allocation	Non-Contiguous	Contiguous

5. Real-world Use Cases

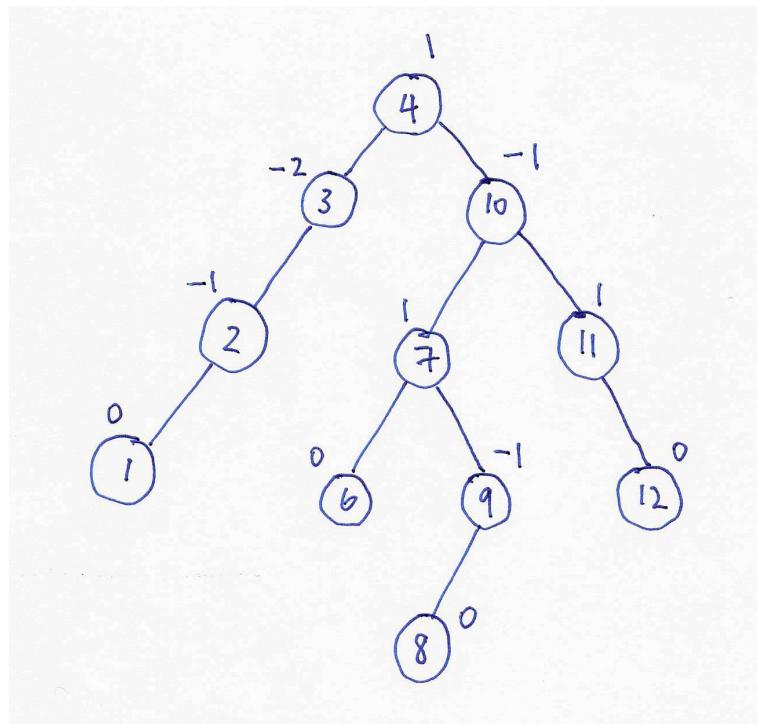
- **Implementing Stacks and Queues:** Linked lists are commonly used to implement stacks and queues due to their efficient insertion and deletion operations.
- **Representing Graphs:** Adjacency lists, which use linked lists, are often used to represent graphs.
- **Dynamic Memory Allocation:** Used in dynamic memory allocation to maintain free blocks of memory.
- **Undo/Redo Functionality:** Text editors and other applications use linked lists to store the history of actions for undo/redo functionality. Each action is a node in the list, making it easy to traverse back and forth.
- **Music Playlists:** A circular linked list is a great way to implement a music playlist so that when the playlist reaches the end, it wraps back to the beginning.

1. AVL Trees (10 Points)

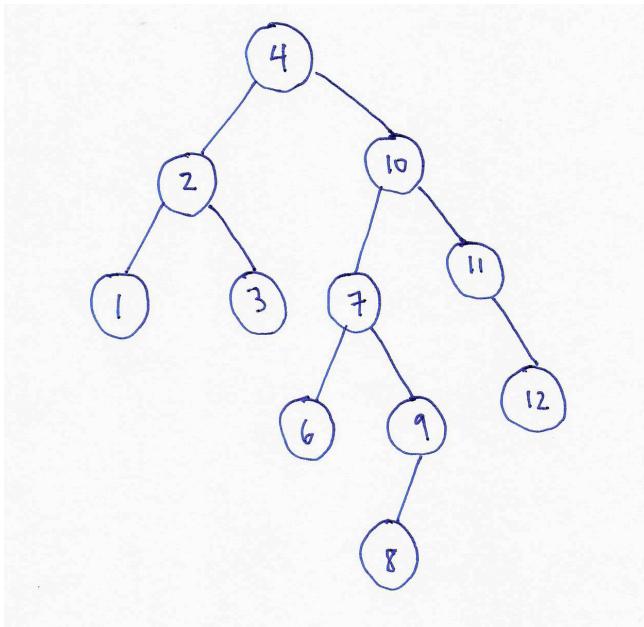
Given the following AVL Tree:



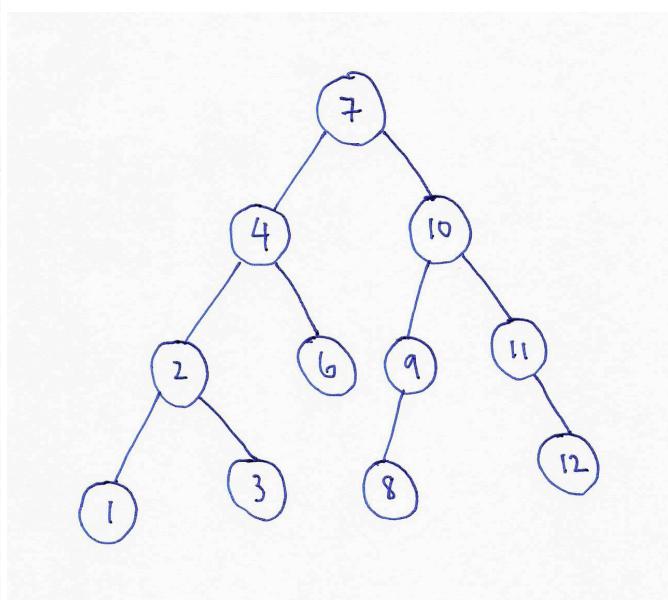
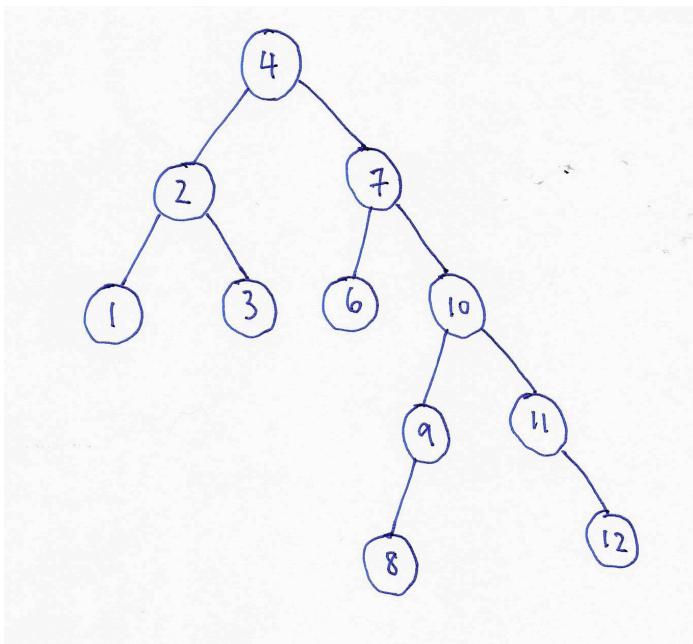
- (a) Draw the resulting BST after 5 is removed, but *before* any rebalancing takes place. Label each node in the resulting tree with its **balance factor**. Replace a node with both children using an appropriate value from the node's left child.



- (b) Now rebalance the tree that results from (a). **Draw a new tree** for each rotation that occurs when rebalancing the AVL Tree (you only need to draw one tree that results from an RL or LR rotation). You do not need to label these trees with balance factors.



We show the intermediate rotation from the RL rotation along with the final answer below:



2. Hashing (9 Points)

Simulate the behavior of a **hash set** storing integers given the following conditions:

- The initial array length is **5**.
- The set uses **quadratic probing** for collision resolution.
- The hash function uses the `int` value (plus any probing needed) mod the size of the table.
- When the load factor reaches or exceeds 0.5, the table enlarges to double capacity and rehashes values stored at smaller indices first.
- An insertion **fails** if more than half of the buckets are tried or if the **properties of a set** are violated.

Given the following code:

```
Set<Integer> set = new HashSet<Integer>(5);  
set.add(86);  
set.add(76);  
set.add(16);  
set.add(66);  
set.add(26);  
set.add(76);
```

(Show your work on the next page to get partial credit in case you make a mistake somewhere.)

- (a) What are the indices of the values in the final hash set?

16: 17 26: 6 66: 7 76: 16 86: 10

- (b) Did any values fail to be inserted? If so, which ones and why did the insertion fail?

The second 76. It is a duplicate value.

- (c) What is the size of the final set? **5**

- (d) What is the capacity of the final set? **20**

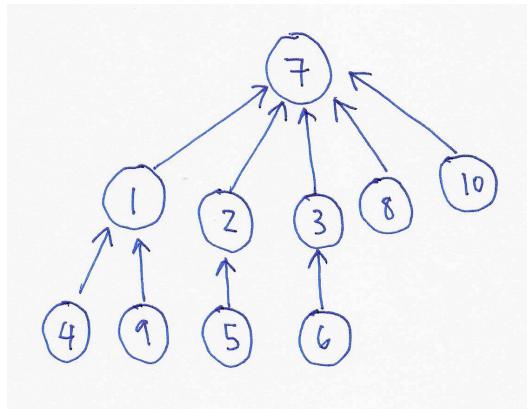
- (e) What is the load factor of the final set? **0.25**

3. Disjoint Sets (9 Points)

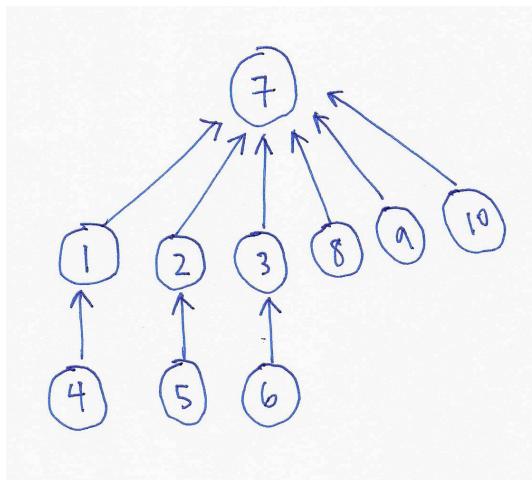
Trace the sequence of operations below, using a disjoint set with union by size (without path compression). If there is a tie when performing a union, the tree with the smaller-valued root should be the root of the union. Assume there are initially 10 sets $\{1\}, \{2\}, \{3\}, \dots, \{10\}$.

```
union(7, 8);  
union(3, 6);  
union(2, 5);  
union(7, 10);  
union(1, 4);  
union(3, 7);  
union(2, 7);  
union(1, 9);  
x = find(7);  
y = find(9);  
union(x, y);
```

- (a) Draw the final forest of up-trees that result from the operations above.



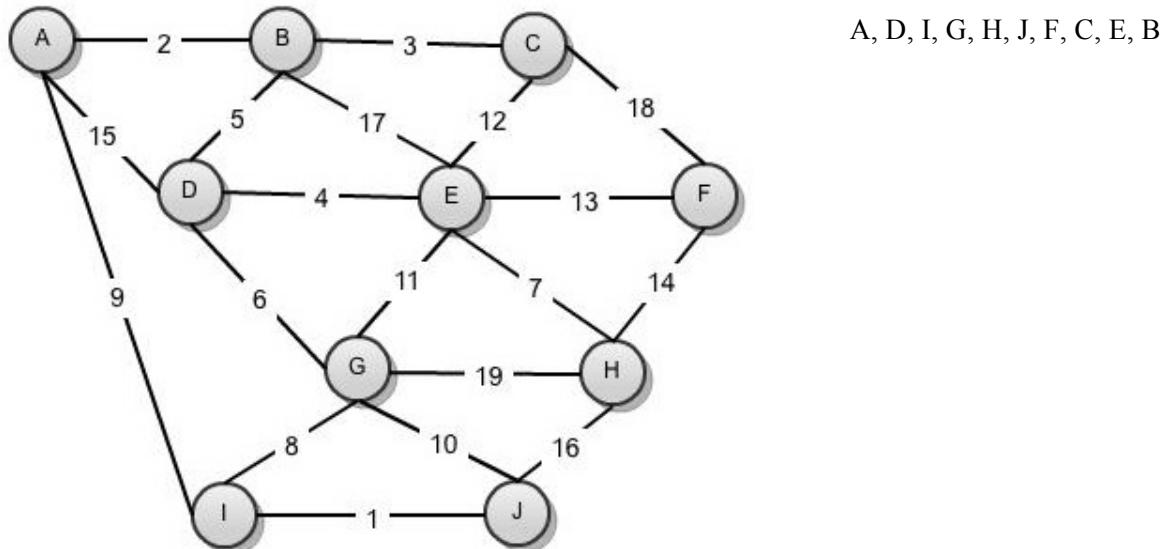
- (b) Draw the new forest of up-trees that results from doing a `find(9)` with path compression on your forest of up-trees from (a).



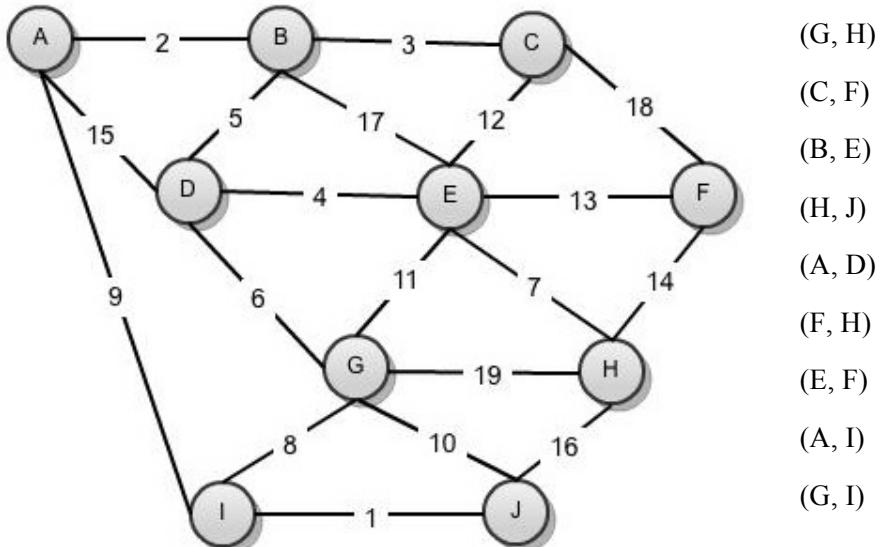
4. Maximum Spanning Trees (10 Points)

For this problem, you will be finding the maximum spanning tree. You can still use Prim's and Kruskal's algorithms if you just switch "smallest" with "biggest" when examining edges.

(a) Using Prim's algorithm starting with vertex "A", list the vertices of the graph below in the order they are added to the maximum spanning tree.

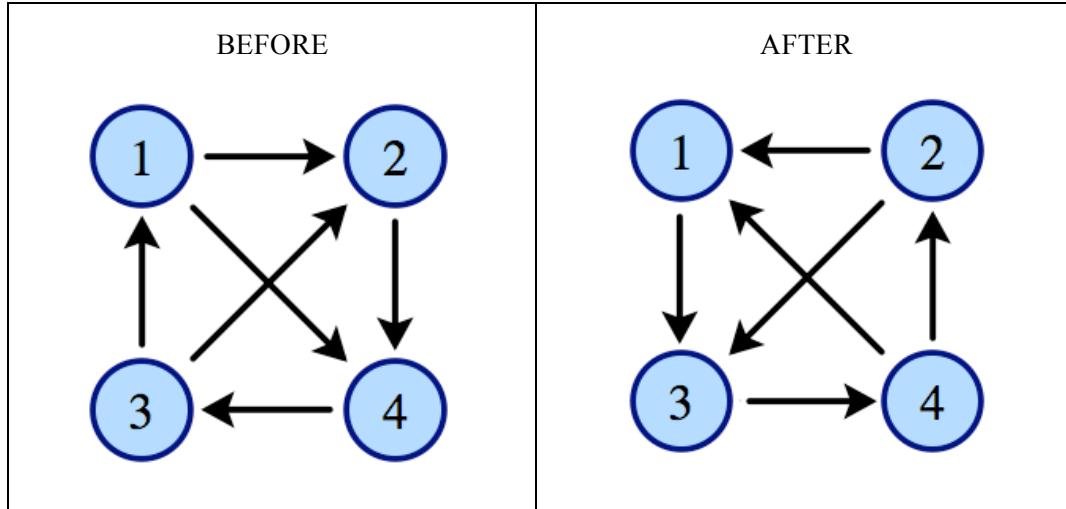


(b) Using Kruskal's algorithm, list the edges of the maximum spanning tree of the graph below in the order that they are added.



5. Graphs (15 Points)

- (a) Suppose that the `Graph` class in Homework #7 (Kevin Bacon) was a **directed** graph. Write a method named `reverse` that would replace all edges (v, w) with (w, v) . The picture below shows a graph before and after a call to `reverse`.



```

public void reverse() {
    Map<V, Map<V, EdgeInfo<E>>> newAdjacencyMap
        = new HashMap<V, Map<V, EdgeInfo<E>>>();

    for (V source : vertices()) {
        for (V target : adjacencyMap.get(source).keySet()) {
            EdgeInfo<E> e = adjacencyMap.get(source).get(target);

            if (!newAdjacencyMap.containsKey(target)) {
                newAdjacencyMap.put(target, new HashMap<V, EdgeInfo<E>>());
            }

            Map<V, EdgeInfo<E>> targetMap = newAdjacencyMap.get(target);
            targetMap.put(source, e);
        }
    }

    adjacencyMap = newAdjacencyMap;
}

```

- (b) What is the running time of your method? Explain your answer.

The method processes each edge exactly once while accessing each vertex on the vertex list: $O(|V| + |E|)$

$O(|E|)$ is also acceptable.

6. What's the Point? (4 Points)

(a) What is the purpose of AVL trees?

AVL trees are binary search trees that are guaranteed to be balanced. They guarantee $O(\log n)$ operations on the tree.

(b) What is the purpose of heaps?

Heaps are how most priority queues are implemented. They allow you to quickly find the minimum value from the values stored in the heap without costly adds.

7. More Graphs (7 Points)

Assume a graph that represents all the web pages (vertices) and the hyperlinks (edges) between them on the Internet.

Is the graph...

(a) **directed**

(b) **cyclic**

(c) **unweighted**

(d) What kinds of web pages have a high in-degree?

Examples:

- Popular web sites like YouTube
- Popular news articles that other pages link to

(e) What kinds of web pages have a high out-degree?

Examples:

- Directory pages like Yahoo!
- Personalized pages like MyUW
- News sites like New York Times

8. Sorting (4 Points)

In the `Arrays` class, there is a static sort method:

```
Arrays.sort(int[] a)
```

The method sorts the specified array into ascending numerical order. The sorting algorithm is a tuned quicksort.

Quicksort has a worst case running time of $O(n^2)$ whereas mergesort has a worst case running time of $O(n \log n)$, but quicksort is generally chosen over mergesort. Why? Explain your answer.

Quicksort generally runs in $O(n \log n)$ time. The constant factor is better than mergesort, because it does the sort with simple swaps, whereas mergesort has to create a new array to store the merged data.

9. Hash Functions (9 Points)

Override the `hashCode` method of the following (poorly-designed) class. Your method should follow proper hashing principles and minimize collisions. You may assume that any non-primitive field in `RentalCar` has an `equals` methods and a good implementation of `hashCode`.

```
public class RentalCar {  
    private int year;  
    private String color;  
    private String model;  
    private Person renter; // null if available  
  
    ...  
  
    public boolean equals(Object other) {  
        if (!(other instanceof RentalCar)) {  
            return false;  
        }  
  
        RentalCar o = (RentalCar)other;  
  
        return  
            (year / 10 == o.year / 10) // car model year in same decade  
            && model.equals(o.model)  
            && ((renter == null && o.renter == null) ||  
                (renter != null && renter.equals(o.renter)));  
    }  
  
    public int hashCode() {  
        int result = 17;  
        result = 37 * result + (year / 10);  
        result = 37 * result + model.hashCode();  
  
        if (renter == null) {  
            result = 37 * result + 1;  
        } else {  
            result = 37 * result + renter.hashCode();  
        }  
  
        return result;  
    }  
}
```

10. B-Trees (9 Points)

(a) What is the purpose of B-Trees?

B-Trees are trees that take into account that most data is not stored in main memory. It's designed to minimize the number of disk accesses that must be performed when operating on the tree.

(b) Describe how to compute M and L in terms of variables and constraints. Define your variables.

For example:

A = number of apples

B = weight of bananas

C = size of carrots

Find smallest integer M where $M \geq 10 \cdot A + \frac{B}{C}$. Equivalently, $M = \left\lceil 10 \cdot A + \frac{B}{C} \right\rceil$

B = size of page block

K = size of key

P = size of branch pointer

R = size of record (including key)

$$M = \left\lceil \frac{B+K}{K+P} \right\rceil$$

$$L = \left\lceil \frac{B}{R} \right\rceil$$

11. Miscellaneous (10 Points)

- (a) Write a method that given an array of integers will print out a list of duplicates (separated by spaces). The contents of the array do not have to be preserved. The duplicates should appear exactly once, though they do not have to be in any particular order. It is OK to have an extra space at the end of the output. The method may use any auxiliary data structures.

Array	Possible Output
{ 1, 2, 3, 4, 1, 2, 3 }	1 2 3
{ 1, 2, 3, 4, 4, 3, 2, 1 }	2 4 3 1
{ 1, 2, 3, 4 }	
{ 1, 1, 1, 2, 2, 1, 1, 1 }	2 1

```
public static void printDuplicates(int[] array) {
    Arrays.sort(array);

    boolean foundDuplicate = false;
    int lastDuplicate = 0;
    for (int i = 1; i < array.length; i++) {
        if (array[i] != array[i-1]) {
            if (foundDuplicate) {
                System.out.print(lastDuplicate + " ");
            }
            foundDuplicate = false;
        } else {
            lastDuplicate = array[i];
            foundDuplicate = true;
        }
    }

    if (foundDuplicate) {
        System.out.print(lastDuplicate);
    }

    System.out.println();
}

public static void printDuplicates(int[] array) {
    Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
    for (int num : array) {
        if (counts.containsKey(num)) {
            counts.put(num, counts.get(num)+1);
        } else {
            counts.put(num, 1);
        }
    }

    for (int num : counts.keySet()) {
        if (counts.get(num) > 1) {
            System.out.print(num + " ");
        }
    }

    System.out.println();
}
```

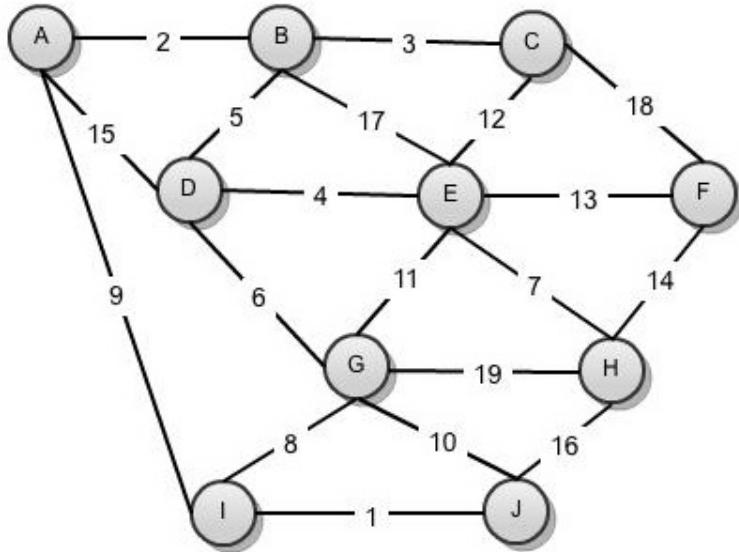
(b) What is the runtime of your implementation? Explain your answer.

The runtime of the first method that uses sorting is dominated by the sort and takes $O(n \log n)$ time. Finding the duplicates from a sorted list takes linear time and is subsumed by the runtime of the sort.

The runtime of the second is linear. It goes over the array and accesses the hash per element. Accessing the hash table takes constant time, so the total running time is linear. Printing the elements requires going over the hash values and only printing values that have a count greater than one. That also takes only linear time. $O(n)$

12. Graph Search (4 Points)

Given the following graph:



For the following problems, assume alphabetical edge order.

- (a) Write (in order) the list of vertices that would be visited by running a breadth-first search (BFS) starting from G
G D E H I J A B C F

- (b) Write (in order) the list of vertices that would be visited by running a depth-first search (DFS) starting from G.
G D A B C E F H J I

Grading criteria:

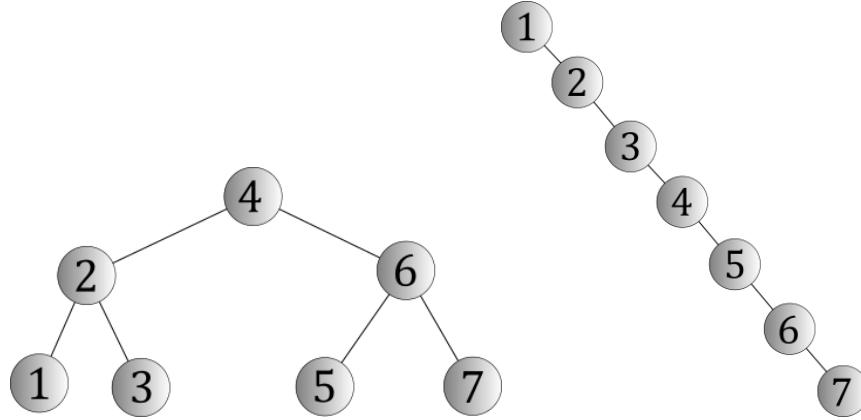
1. AVL Trees (a) 4 points 2 element correctly removed 2 balance factors (b) 6 points 3 attempts rotations to get final AVL tree 3 correct	7. More Graphs 1 point for (a), (b), (c) 2 points each for (d), (e) (answer must be reasonable)
2. Hashing 1 point per blank/answer (e) is ok if it equals (c) / (d) -3 if don't rehash in the right order	8. Sorting 1 point if bad answer 2 points if partial answer
3. Disjoint Sets (a) 6 points -3 per mistake (b) 3 points (all or nothing)	9. Hash Function 1 correctly ignores color 3 correct polynomial accumulation 1 attempt 2 correct 1 handles decade 1 handles model 2 handles null renter 1 handles non-null renter
4. Maximum Spanning Tree 5 points each (-3 points per swap/extra/missing)	10. B-Trees (a) 3 points (1 point if bad answer) (b) 6 points 2 states correct variables (-1 per missing variable) 3 correct equation for M 1 correct equation for L -1 for missing floor
5. Graphs (a) 12 points 1 header 3 loops over all edges 1 attempt 2 correct 2 creates new map for new source (old target) 2 places reversed (w, v) in map 1 sets old adjacency map properly 3 correct If not modifying adjacency map (max 8 points): 1 header 3 loops over all edges 1 attempt 2 correct 2 places reversed edges 2 removed original edge (b) 3 points (1 point if bad answer)	11. Miscellaneous (a) 7 points 1 header 2 finds all duplicates (no false positive) 2 find duplicates exactly once 2 correct (b) 3 points (1 point if bad answer) 12. Graph Search 2 points each (all or nothing)
6. What's the Point? 2 points per section (1 incomplete answer)	

ICS 46 Spring 2022

Notes and Examples: AVL Trees

Why we must care about binary search tree balancing

We've seen previously that the performance characteristics of [binary search trees](#) can vary rather wildly, and that they're mainly dependent on the shape of the tree, with the height of the tree being the key determining factor. By definition, binary search trees restrict what keys are allowed to present in which nodes — smaller keys have to be in left subtrees and larger keys in right subtrees — but they specify no restriction on the tree's shape, meaning that both of these are perfectly legal binary search trees containing the keys 1, 2, 3, 4, 5, 6, and 7.



Yet, while both of these are legal, one is better than the other, because the height of the first tree (called a *perfect binary tree*) is smaller than the height of the second (called a *degenerate tree*). These two shapes represent the two extremes — the best and worst possible shapes for a binary search tree containing seven keys.

Of course, when all you have is a very small number of keys like this, any shape will do. But as the number of keys grows, the distinction between these two tree shapes becomes increasingly vital. What's more, the degenerate shape isn't even necessarily a rare edge case: It's what you get when you start with an empty tree and add keys that are already in order, which is a surprisingly common scenario in real-world programs. For example, one very obvious algorithm for generating unique integer keys — when all you care about is that they're unique — is to generate them sequentially.

What's so bad about a degenerate tree, anyway?

Just looking at a picture of a degenerate tree, your intuition should already be telling you that something is amiss. In particular, if you tilt your head 45 degrees to the right, they look just like linked lists; that perception is no accident, as they behave like them, too (except that they're more complicated, to boot!).

From a more analytical perspective, there are three results that should give us pause:

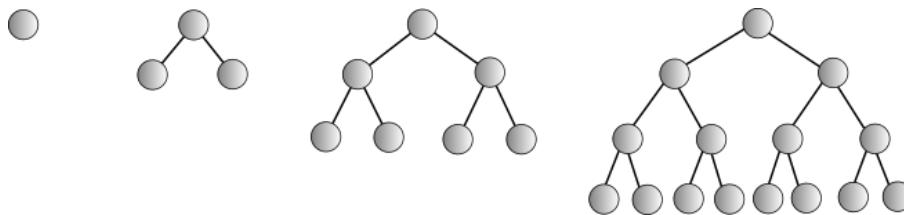
- Every time you perform a lookup in a degenerate binary search tree, it will take $O(n)$ time, because it's possible that you'll have to reach every node in the tree before you're done. As n grows, this is a heavy burden to bear.
- If you implement your lookup recursively, you might also be using $O(n)$ memory, too, as you might end up with as many as n frames on your run-time stack — one for every recursive call. There are ways to mitigate this — for example, some kinds of carefully-written recursion (in some programming languages, including C++) can avoid run-time stack growth as you recurse — but it's still a sign of potential trouble.
- The time it will take you to build the degenerate tree will also be prohibitive. If you start with an empty binary search tree and add keys to it in order, how long does it take to do it?
 - The first key you add will go directly to the root. You could think of this as taking a single step: creating the node.
 - The second key you add will require you to look at the root node, then take one step to the right. You could think of this as taking two steps.
 - Each subsequent key you add will require one more step than the one before it.
 - The total number of steps it would take to add n keys would be determined by the sum $1 + 2 + 3 + \dots + n$. This sum, which we'll see several times throughout this course, is equal to $n(n + 1) / 2$.
 - So, the total number of steps to build the entire tree would be $\Theta(n^2)$.

Overall, when n gets large, the tree would be hideously expensive to build, and then every subsequent search would be painful, as well. So this, in general, is a situation we need to be sure to avoid, or else we should probably consider a data structure other than a binary search tree; the worst case is simply too much of a burden to bear if n might get large. But if we can find a way to control the tree's shape more carefully, to force it to remain more *balanced*, we'll be fine. The question, of course, is how to do it, and, as importantly, whether we can do it while keeping the cost low enough that it doesn't outweigh the benefit.

Aiming for perfection

The best goal for us to shoot for would be to maintain perfection. In other words, every time we insert a key into our binary search tree, it would ideally still be a perfect binary tree, in which case we'd know that the height of the tree would always be $\Theta(\log n)$, with a commensurate effect on performance.

However, when we consider this goal, a problem emerges almost immediately. The following are all perfect binary trees, by definition:



The perfect binary trees pictured above have 1, 3, 7, and 15 nodes respectively, and are the only possible perfect shapes for binary trees with that number of nodes. The problem, though, lies in the fact that there is no valid perfect binary tree with 2 nodes, or with 4, 5, 6, 8, 9, 10, 11, 12, 13, or 14 nodes. So, generally, it's impossible for us to guarantee that a binary search tree will always be "perfect," by our definition, because there's simply no way to represent most numbers of keys.

So, first thing's first: We'll need to relax our definition of "perfection" to accommodate every possible number of keys we might want to store.

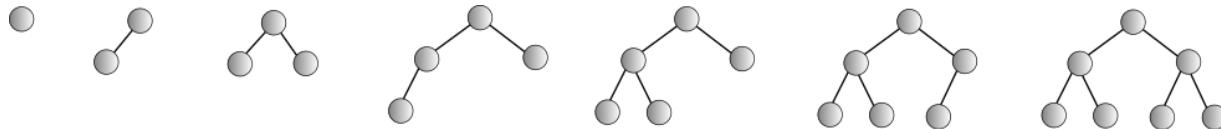
Complete binary trees

A somewhat more relaxed notion of "perfection" is something called a *complete binary tree*, which is defined as follows.

A *complete binary tree* of height h is a binary tree where:

- If $h = 0$, its left and right subtrees are empty.
- If $h > 0$, one of two things is true:
 - The left subtree is a perfect binary tree of height $h - 1$ and the right subtree is a complete binary tree of height $h - 1$
 - The left subtree is a complete binary tree of height $h - 1$ and the right subtree is a perfect binary tree of height $h - 2$

That can be a bit of a mind-bending definition, but it actually leads to a conceptually simple result: On every level of a complete binary tree, every node that could possibly be present will be, *except* the last level might be missing nodes, but if it is missing nodes, the nodes that are there will be as far to the left as possible. The following are all complete binary trees:



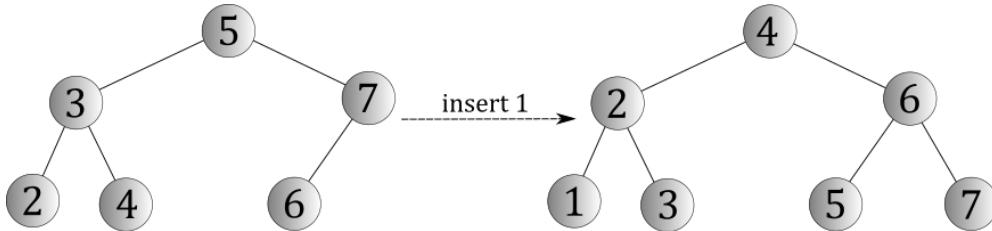
Furthermore, these are the only possible complete binary trees with these numbers of nodes in them; any other arrangement of, say, 6 keys besides the one shown above would violate the definition.

We've seen that the height of a perfect binary tree is $\Theta(\log n)$. It's not a stretch to see that the height of a complete binary tree will be $\Theta(\log n)$, as well, and we'll accept that via our intuition for now and proceed. All in all, a complete binary tree would be a great goal for us to attain: If we could keep the shape of our binary search trees complete, we would always have binary search trees with height $\Theta(\log n)$.

The cost of maintaining completeness

The trouble, of course, is that we need an algorithm for maintaining completeness. And before we go to the trouble of trying to figure one out, we should consider whether it's even worth our time. What can we deduce about the cost of maintaining completeness, even if we haven't figured out an algorithm yet?

One example demonstrates a very big problem. Suppose we had the binary search tree on the left — which is complete, by our definition — and we wanted to insert the key 1 into it. If so, we would need an algorithm that would transform the tree on the left into the tree on the right.



The tree on the right is certainly complete, so this would be the outcome we'd want. But consider what it would take to do it. *Every key in the tree had to move!* So, no matter what algorithm we used, we would still have to move every key. If there are n keys in the tree, that would take $\Omega(n)$ time — moving n keys takes at least linear time, even if you have the best possible algorithm for moving them; the work still has to get done.

So, in the worst case, maintaining completeness after a single insertion requires $\Omega(n)$ time. Unfortunately, this is more time than we ought to be spending on maintaining balance. This means we'll need to come up with a compromise; as is often the case when we learn or design algorithms, our willingness to tolerate an imperfect result that's still "good enough" for our uses will often lead to an algorithm that is much faster than one that achieves a perfect result. So what would a "good enough" result be?

What is a "good" balance condition

Our overall goal is for lookups, insertions, and removals from a binary search tree to require $O(\log n)$ time in every case, rather than letting them degrade to a worst-case behavior of $O(n)$. To do that, we need to decide on a *balance condition*, which is to say that we need to understand what shape is considered well-

enough balanced for our purposes, even if not perfect.

A "good" balance condition has two properties:

- The height of a binary search tree meeting the condition is $\Theta(\log n)$.
- It takes $O(\log n)$ time to re-balance the tree on insertions and removals.

In other words, it guarantees that the height of the tree is still logarithmic, which will give us logarithmic-time lookups, and the time spent re-balancing won't exceed the logarithmic time we would otherwise spend on an insertion or removal when the tree has logarithmic height. The cost won't outweigh the benefit.

Coming up with a balance condition like this on our own is a tall task, but we can stand on the shoulders of the giants who came before us, with the definition above helping to guide us toward an understanding of whether we've found what we're looking for.

A compromise: AVL trees

There are a few well-known approaches for maintaining binary search trees in a state of near-balance that meets our notion of a "good" balance condition. One of them is called an *AVL tree*, which we'll explore here. Others, which are outside the scope of this course, include red-black trees (which meet our definition of "good") and splay trees (which don't always meet our definition of "good", but do meet it on an amortized basis), but we'll stick with the one solution to the problem for now.

AVL trees

AVL trees are what you might call "nearly balanced" binary search trees. While they certainly aren't as perfectly-balanced as possible, they nonetheless achieve the goals we've decided on: maintaining logarithmic height at no more than logarithmic cost.

So, what makes a binary search tree "nearly balanced" enough to be considered an AVL tree? The core concept is embodied by something called the *AVL property*.

We say that a node in a binary search tree has the *AVL property* if the heights of its left and right subtrees differ by no more than 1.

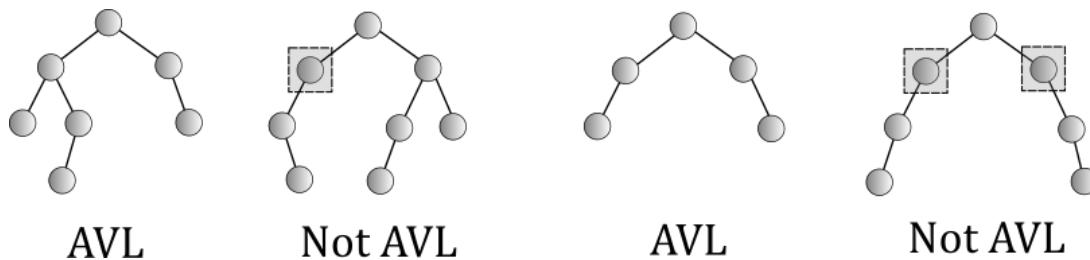
In other words, we tolerate a certain amount of imbalance — heights of subtrees can be slightly different, but no more than that — in hopes that we can more efficiently maintain it.

Since we're going to be comparing heights of subtrees, there's one piece of background we need to consider. Recall that the *height of a tree* is the length of its longest path. By definition, the height of a tree with just a root node (and empty subtrees) would then be zero. But what about a tree that's totally empty? To maintain a clear pattern, relative to other tree heights, we'll say that the *height of an empty tree* is -1. This means that a node with, say, a childless left child and no right child would still be considered balanced.

This leads us, finally, to the definition of an AVL tree:

An AVL tree is a binary search tree in which all nodes have the AVL property.

Below are a few binary trees, two of which are AVL and two of which are not.



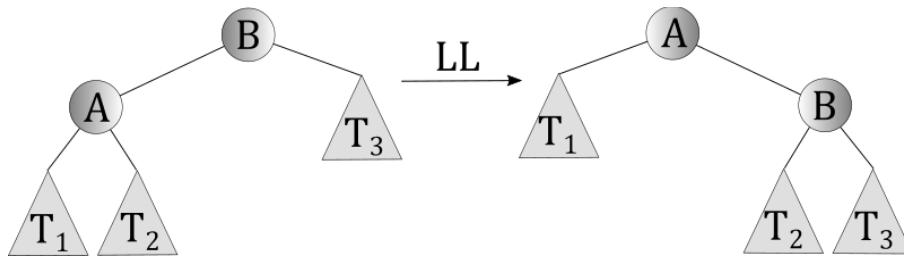
The thing to keep in mind about AVL is that it's not a matter of squinting at a tree and deciding whether it "looks" balanced. There's a precise definition, and the two trees above that don't meet that definition fail to meet it because they each have at least one node (marked in the diagrams by a dashed square) that doesn't have the AVL property.

AVL trees, by definition, are required to meet the balance condition after every operation; every time you insert or remove a key, every node in the tree should have the AVL property. To meet that requirement, we need to restructure the tree periodically, essentially detecting and correcting imbalance whenever and wherever it happens. To do that, we need to rearrange the tree in ways that improve its shape without losing the essential ordering property of a binary search tree: smaller keys toward the left, larger ones toward the right.

Rotations

Re-balancing of AVL trees is achieved using what are called *rotations*, which, when used at the proper times, efficiently improve the shape of the tree by altering a handful of pointers. There are a few kinds of rotations; we should first understand how they work, then focus our attention on when to use them.

The first kind of rotation is called an *LL rotation*, which takes the tree on the left and turns it into the tree on the right. The circle with A and B written in them are each a single node containing a single key; the triangles with T_1 , T_2 , and T_3 written in them are arbitrary subtrees, which may be empty or may contain any number of nodes (but which are, themselves, binary search trees).



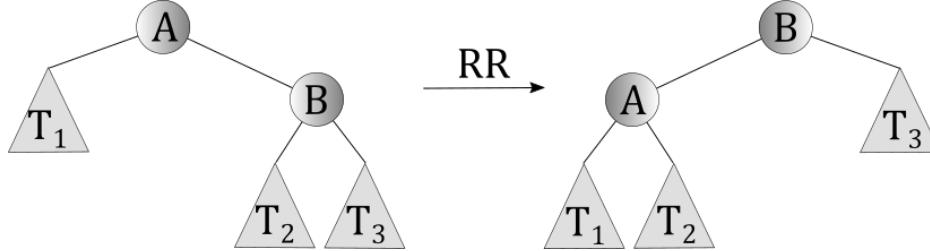
It's important to remember that both of these trees — before and after — are binary search trees; the rotation doesn't harm the ordering of the keys in nodes, because the subtrees T_1 , T_2 , and T_3 maintain the appropriate positions relative to the keys A and B:

- All keys in T_1 are smaller than A.
- All keys in T_2 are larger than A and smaller than B.
- All keys in T_3 are larger than B.

Performing this rotation would be a simple matter of adjusting a few pointers — notably, a constant number of pointers, no matter how many nodes are in the tree, which means that this rotation would run in $\Theta(1)$ time:

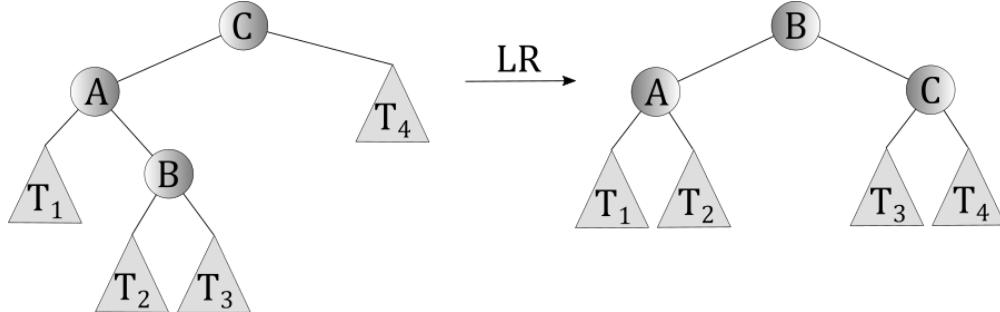
- B's parent would now point to A where it used to point to B
- A's right child would now be B instead of the root of T_2
- B's left child would now be the root of T_2 instead of A

A second kind of rotation is an *RR rotation*, which makes a similar adjustment.



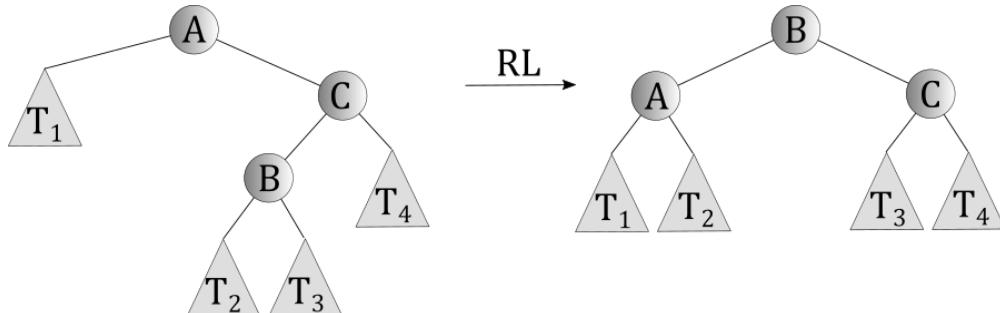
Note that an RR rotation is the mirror image of an LL rotation.

A third kind of rotation is an *LR rotation*, which makes an adjustment that's slightly more complicated.



An LR rotation requires five pointer updates instead of three, but this is still a constant number of changes and runs in $\Theta(1)$ time.

Finally, there is an *RL rotation*, which is the mirror image of an LR rotation.



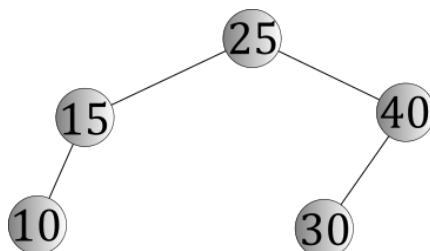
Once we understand the mechanics of how rotations work, we're one step closer to understanding AVL trees. But these rotations aren't arbitrary; they're used specifically to correct imbalances that are detected after insertions or removals.

An insertion algorithm

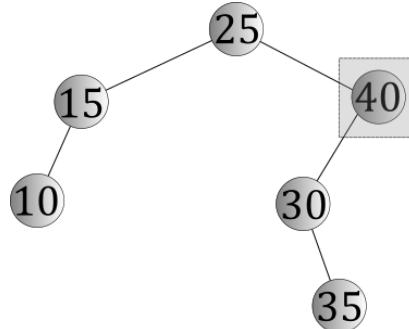
Inserting a key into an AVL tree starts out the same way as insertion into a binary search tree:

- Perform a lookup. If you find the key already in the tree, you're done, because keys in a binary search tree must be unique.
- When the lookup terminates without the key being found, add a new node in the appropriate leaf position where the lookup ended.

The problem is that adding the new node introduced the possibility of an imbalance. For example, suppose we started with this AVL tree:



and then we inserted the key 35 into it. A binary search tree insertion would give us this as a result:



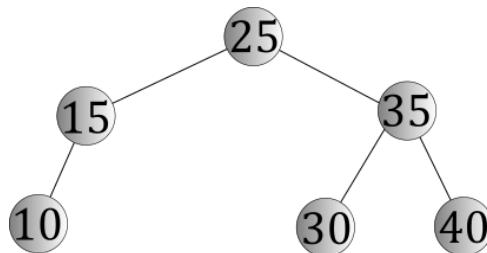
But this resulting tree is not an AVL tree, because the node containing the key 40 does not have the AVL property, because the difference in the heights of its subtrees is 2. (Its left subtree has height 1, its right subtree — which is empty — has height -1.) What can we do about it?

The answer lies in the following algorithm, which we perform after the normal insertion process:

- Work your way back up the tree from the position where you just added a node. (This could be quite simple if the insertion was done recursively.) Compare the heights of the left and right subtrees of each node. When they differ by more than 1, choose a rotation that will fix the imbalance.
 - Note that comparing the heights of the left and right subtrees would be quite expensive if you didn't already know what they were. The solution to this problem is for each node to store its height (i.e., the height of the subtree rooted there). This can be cheaply updated after every insertion or removal as you unwind the recursion.
- The rotation is chosen considering the two links along the path *below* the node where the imbalance is, heading back down toward where you inserted a node. (If you were wondering where the names LL, RR, LR, and RL come from, this is the answer to that mystery.)
 - If the two links are both to the left, perform an LL rotation rooted where the imbalance is.
 - If the two links are both to the right, perform an RR rotation rooted where the imbalance is.
 - If the first link is to the left and the second is to the right, perform an LR rotation rooted where the imbalance is.
 - If the first link is to the right and the second is to the left, perform an RL rotation rooted where the imbalance is.

It can be shown that any one of these rotations — LL, RR, LR, or RL — will correct any imbalance brought on by inserting a key.

In this case, we'd perform an LR rotation — the first two links leading from 40 down toward 35 are a **Left** and a **Right** — rooted at 40, which would correct the imbalance, and the tree would be rearranged to look like this:



Compare this to the diagram describing an LR rotation:

- The node containing 40 is C
- The node containing 30 is A
- The node containing 35 is B
- The (empty) left subtree of the node containing 30 is T_1
- The (empty) left subtree of the node containing 35 is T_2
- The (empty) right subtree of the node containing 35 is T_3
- The (empty) right subtree of the node containing 40 is T_4

After the rotation, we see what we'd expect:

- The node B, which in our example contained 35, is now the root of the newly-rotated subtree
- The node A, which in our example contained 30, is now the left child of the root of the newly-rotated subtree
- The node C, which in our example contained 40, is now the right child of the root of the newly-rotated subtree
- The four subtrees T_1 , T_2 , T_3 , and T_4 were all empty, so they are still empty.

Note, too, that the tree is more balanced after the rotation than it was before. This is no accident; a single rotation (LL, RR, LR, or RL) is all that's necessary to correct an imbalance introduced by the insertion algorithm.

A removal algorithm

Removals are somewhat similar to insertions, in the sense that you would start with the usual binary search tree removal algorithm, then find and correct imbalances while the recursion unwinds. The key difference is that removals can require more than one rotation to correct imbalances, but will still only require rotations on the path back up to the root from where the removal occurred — so, generally, $O(\log n)$ rotations.

Asymptotic analysis

The key question here is *What is the height of an AVL tree with n nodes?* If the answer is $\Theta(\log n)$, then we can be certain that lookups, insertions, and removals will take $O(\log n)$ time. How can we be so sure?

Lookups would be $O(\log n)$ because they're the same as they are in a binary search tree that doesn't have the AVL property. If the height of the tree is $\Theta(\log n)$, lookups will run in $O(\log n)$ time. Insertions and removals, despite being slightly more complicated in an AVL tree, do their work by traversing a single path in the tree — potentially all the way down to a leaf position, then all the way back up. If the length of the longest path — that's what the height of a tree is! — is $\Theta(\log n)$, then we know that none of these paths is longer than that, so insertions and removals will take $O(\log n)$ time.

So we're left with that key question. What is the height of an AVL tree with n nodes? (If you're not curious, you can feel free to just assume this; if you want to know more, keep reading.)

What is the height of an AVL tree with n nodes? (Optional)

The answer revolves around noting how many nodes, at minimum, could be in a binary search tree of height n and still have it be an AVL tree. It turns out AVL trees of height $n \geq 2$ that have the minimum number of nodes in them all share a similar property:

The AVL tree with height $h \geq 2$ with the minimum number of nodes consists of a root node with two subtrees, one of which is an AVL tree with height $h - 1$ with the minimum number of nodes, the other of which is an AVL tree with height $h - 2$ with the minimum number of nodes.

Given that observation, we can write a recurrence that describes the number of nodes, at minimum, in an AVL tree of height h .

$$\begin{aligned} M(0) &= 1 && \text{When height is 0, minimum number of nodes is 1 (a root node with no children)} \\ M(1) &= 2 && \text{When height is 1, minimum number of nodes is 2 (a root node with one child and not the other)} \\ M(h) &= 1 + M(h - 1) + M(h - 2) \end{aligned}$$

While the repeated substitution technique we learned previously isn't a good way to try to solve this particular recurrence, we can prove something interesting quite easily. We know for sure that AVL trees with larger heights have a bigger minimum number of nodes than AVL trees with smaller heights — that's fairly self-explanatory — which means that we can be sure that $1 + M(h - 1) \geq M(h - 2)$. Given that, we can conclude the following:

$$M(h) \geq 2M(h - 2)$$

We can then use the repeated substitution technique to determine a lower bound for this recurrence:

$$\begin{aligned} M(h) &\geq 2M(h - 2) \\ &\geq 2(2M(h - 4)) \\ &\geq 4M(h - 4) \\ &\geq 4(2M(h - 6)) \\ &\geq 8M(h - 6) \\ &\dots \\ &\geq 2^j M(h - 2j) && \text{We could prove this by induction on } j, \text{ but we'll accept it on faith} \\ \text{let } j &= h/2 \\ &\geq 2^{h/2} M(h - h) \\ &\geq 2^{h/2} M(0) \\ M(h) &\geq 2^{h/2} \end{aligned}$$

So, we've shown that the minimum number of nodes that can be present in an AVL tree of height h is at least $2^{h/2}$. In reality, it's actually more than that, but this gives us something useful to work with; we can use this result to figure out what we're really interested in, which is the opposite: what is the height of an AVL tree with n nodes?

$$\begin{aligned} M(h) &\geq 2^{h/2} \\ \log_2 M(h) &\geq h/2 \\ 2 \log_2 M(h) &\geq h \end{aligned}$$

Finally, we see that, for AVL trees of height h with the minimum number of nodes, the height is no more than $2 \log_2 n$, where n is the number of nodes in the tree. For AVL trees with more than the minimum number of nodes, the relationship between the number of nodes and the height is even better, though, for

reasons we've seen previously, we know that the relationship between the number of nodes and the height of a binary tree can never be better than logarithmic. So, ultimately, we see that the height of an AVL tree with n nodes is $\Theta(\log n)$.

(In reality, it turns out that the bound is lower than $2 \log_2 n$; it's something more akin to about $1.44 \log_2 n$, even for AVL trees with the minimum number of nodes, though the proof of that is more involved and doesn't change the asymptotic result.)

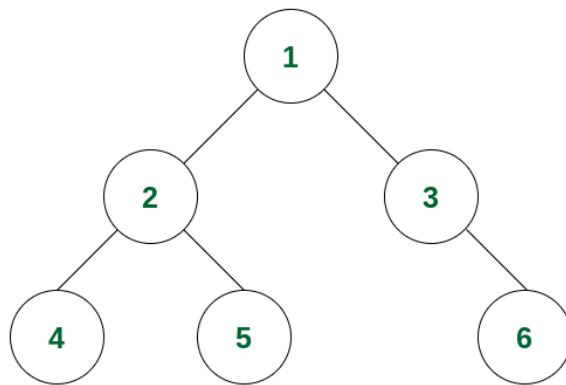


Inorder Traversal of Binary Tree

Last Updated : 21 Oct, 2024

Inorder traversal is defined as a type of [tree traversal technique](#) which follows the Left-Root-Right pattern, such that:

- The **left subtree** is traversed first
- Then the **root node** for that subtree is traversed
- Finally, the **right subtree** is traversed



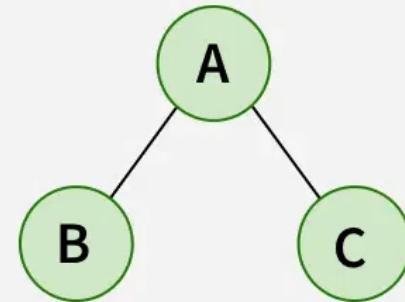
Binary Tree to be traversed

Examples of Inorder Traversal

Input:

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

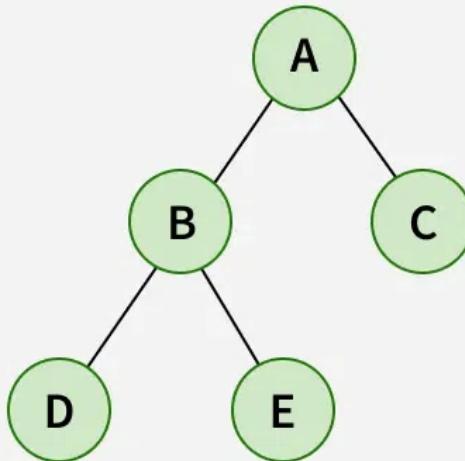
Got It !



Output: BAC

Explanation: The Inorder Traversal visits the nodes in the following order: **Left, Root, Right**. Therefore, we visit the left node B, then the root node A and lastly the right node C.

Input :



Output: DBEAC

Input: NULL

Output:

Output is empty in this case.

Inorder(root):

1. If root is NULL, then return
2. Inorder (root -> left)
3. Process root (For example, print root's data)
4. Inorder (root -> right)

Program to implement Inorder Traversal of Binary Tree:

Below is the code implementation of the inorder traversal:

C++ C Java **Python** C# JavaScript

```
# Structure of a Binary Tree Node
class Node:
    def __init__(self, v):
        self.data = v
        self.left = None
        self.right = None

# Function to print inorder traversal
def printInorder(node):
    if node is None:
        return

    # First recur on left subtree
    printInorder(node.left)

    # Now deal with the node
    print(node.data, end=' ')

    # Then recur on right subtree
    printInorder(node.right)

# Driver code
if __name__ == '__main__':
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    root.right.right = Node(6)

    # Function call
    print("Inorder traversal of binary tree is:")
    printInorder(root)
```



We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Inorder traversal of binary tree is:

4 2 5 1 3 6

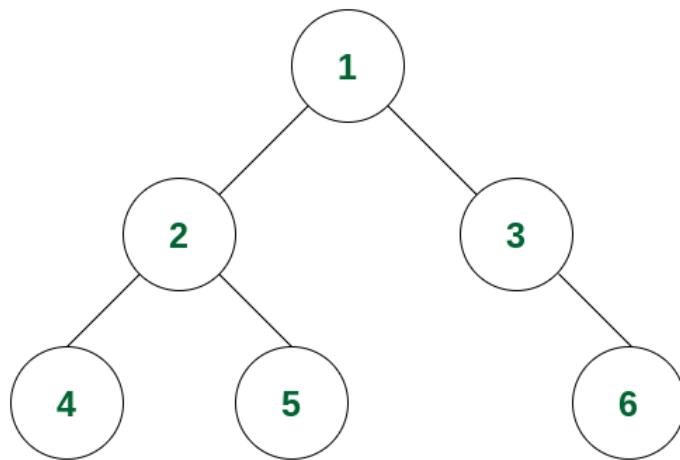
Time Complexity: $O(N)$ where N is the total number of nodes. Because it traverses all the nodes at least once.

Auxiliary Space: $O(h)$ where h is the height of the tree. This space is required for recursion calls.

- In the worst case, h can be the same as N (when the tree is a skewed tree)
- In the best case, h can be the same as $\log N$ (when the tree is a complete tree)

How does Inorder Traversal of Binary Tree work?

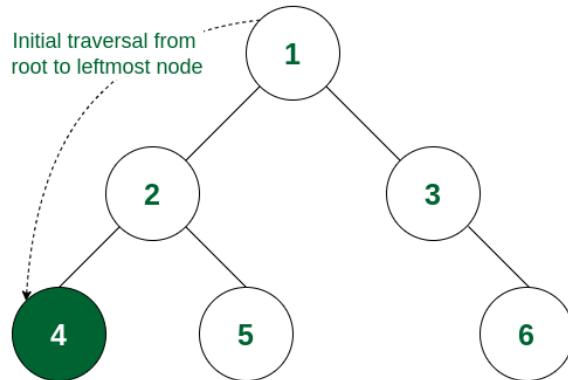
Let us understand the algorithm with the below example tree



Example of Binary Tree

If we perform an inorder traversal in this binary tree, then the traversal will be as follows:

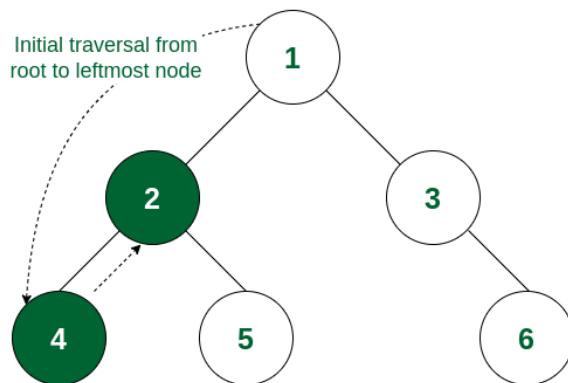
Step 1: The traversal will go from 1 to its left subtree i.e., 2, then from 2 to its left subtree root, i.e., 4. Now 4 has no left subtree, so it will be visited. It also does not have any right subtree. So no more traversal from 4



Leftmost node of the tree is visited

Node 4 is visited

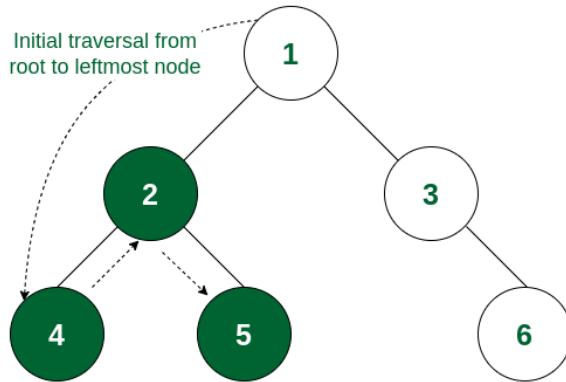
Step 2: As the left subtree of 2 is visited completely, now it reads data of node 2 before moving to its right subtree.



Left subtree of 2 is fully traversed. So 2 is visited next

Node 2 is visited

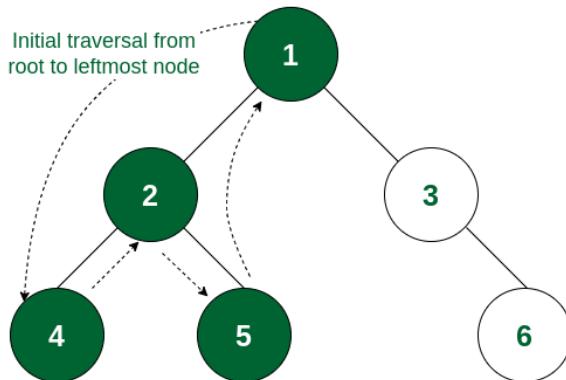
Step 3: Now the right subtree of 2 will be traversed i.e., move to node 5. For node 5 there is no left subtree, so it gets visited and after that, the traversal comes back because there is no right subtree of node 5.



Right subtree of 2 (i.e., 5) is traversed

Node 5 is visited

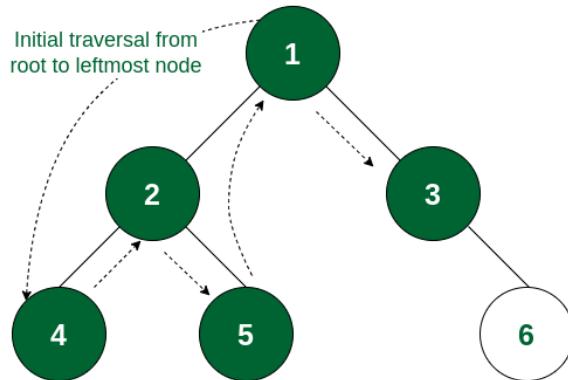
Step 4: As the left subtree of node 1 is, the root itself, i.e., node 1 will be visited.



Left subtree of 1 is fully traversed. So 1 is visited next

Node 1 is visited

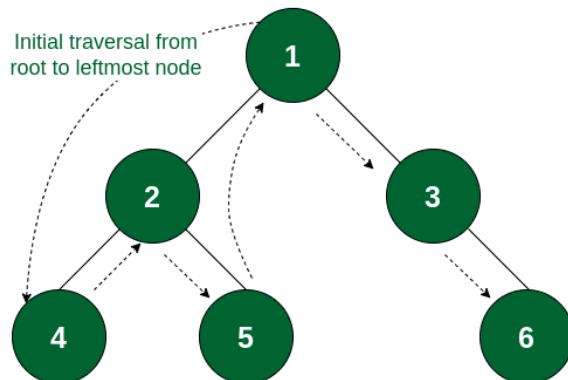
Step 5: Left subtree of node 1 and the node itself is visited. So now the right subtree of 1 will be traversed i.e., move to node 3. As node 3 has no left subtree so it gets visited.



3 has no left subtree, so it is visited

Node 3 is visited

Step 6: The left subtree of node 3 and the node itself is visited. So traverse to the right subtree and visit node 6. Now the traversal ends as all the nodes are traversed.



Right Child of 3 is visited

The complete tree is traversed

So the order of traversal of nodes is 4 -> 2 -> 5 -> 1 -> 3 -> 6.

Use cases of Inorder Traversal:

In the case of BST (Binary Search Tree), if any time there is a need to get the nodes in increasing order

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

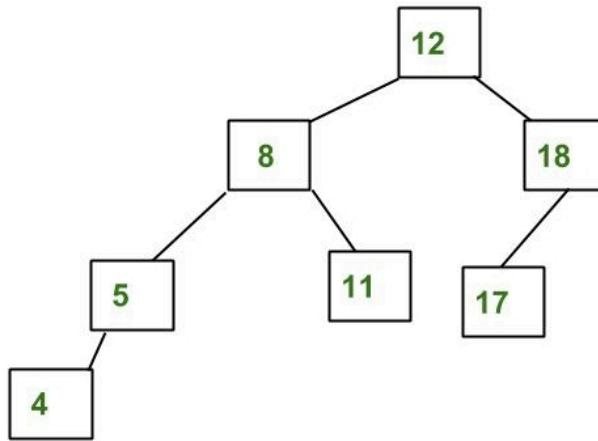


Insertion in an AVL Tree

Last Updated : 22 Feb, 2025

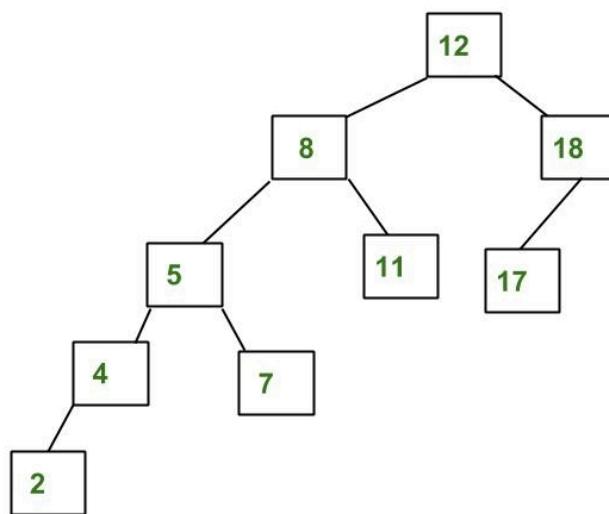
[AVL tree](#) is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Example of AVL Tree:



The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees? Most of the BST operations (e.g., search, max, min, insert, delete, floor and ceiling) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a **skewed Binary tree**. If we make sure that the height of the tree remains $O(\log(n))$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log(n))$ for all these operations. The height of an AVL tree is always $O(\log(n))$ where n is the number of nodes in the tree.

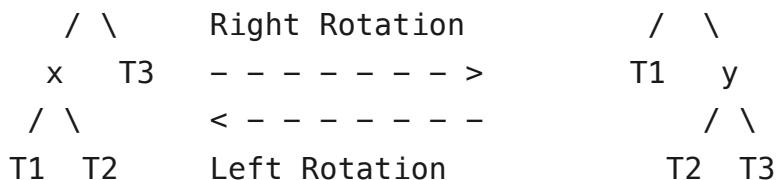
Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property ($\text{keys(left)} < \text{key(root)} < \text{keys(right)}$).

- Left Rotation
- Right Rotation

T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).



Keys in both of the above trees follow the following order

$\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$

So BST property is not violated anywhere.

Steps to follow for insertion:

Let the newly inserted node be w

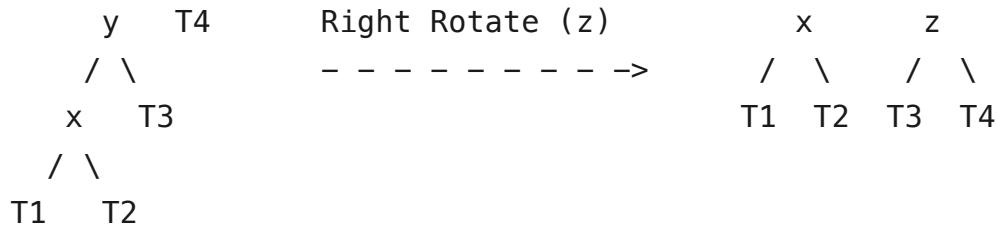
- Perform standard **BST** insert for **w**.
 - Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
 - Re-balance the tree by performing appropriate rotations on the subtree rooted with **z**. There can be 4 possible cases that need to be handled as **x**, **y** and **z** can be arranged in 4 ways.
 - Following are the possible 4 arrangements:
 - **y** is the left child of **z** and **x** is the left child of **y** (Left Left Case)
 - **y** is the left child of **z** and **x** is the right child of **y** (Left Right Case)
 - **y** is the right child of **z** and **x** is the right child of **y** (Right Right Case)
 - **y** is the right child of **z** and **x** is the left child of **y** (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to **re-balance** the subtree rooted with z and the complete tree becomes balanced as the height of the subtree (After appropriate rotations) rooted with z becomes the same as it was before insertion.

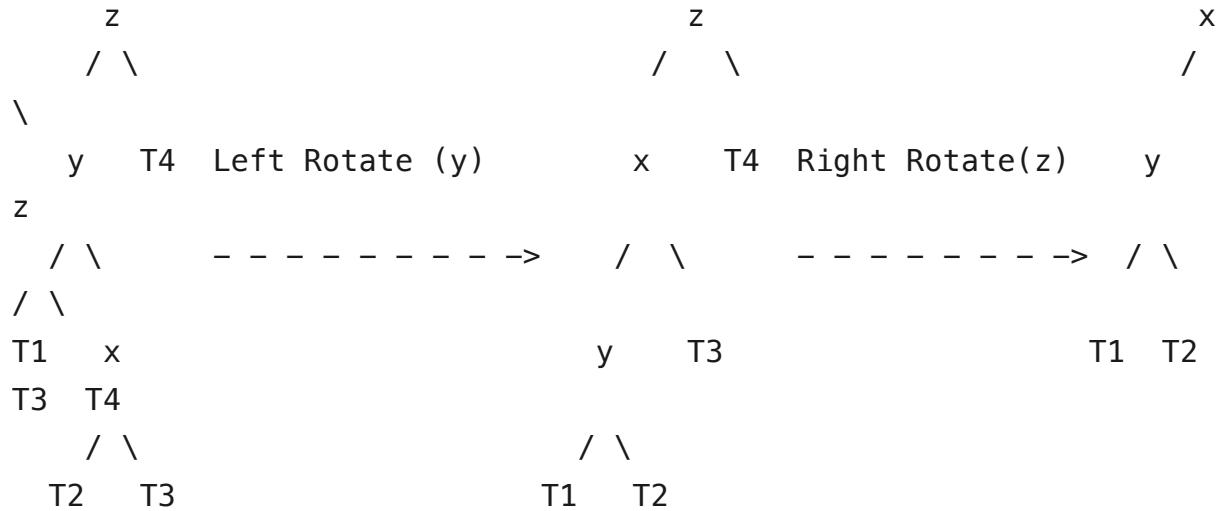
1. Left Left Case

T1, T2, T3 and T4 are subtrees.

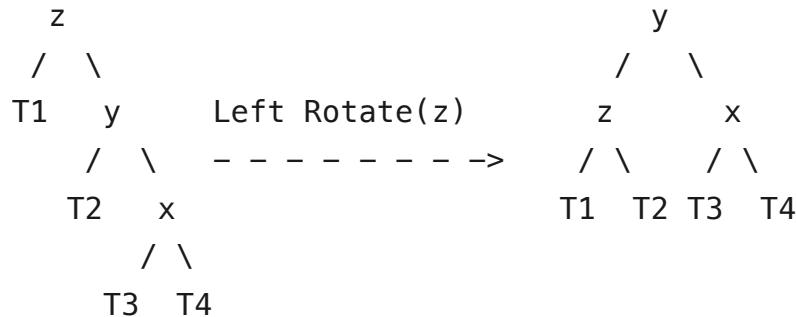
We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy & Privacy Policy](#)



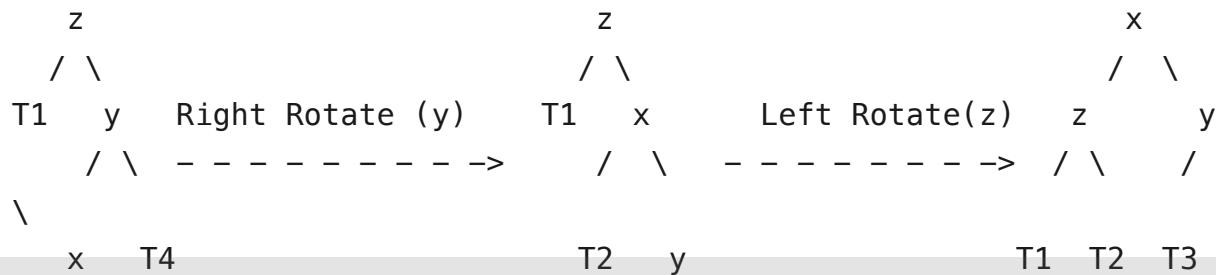
2. Left Right Case



3. Right Right Case



4. Right Left Case



We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

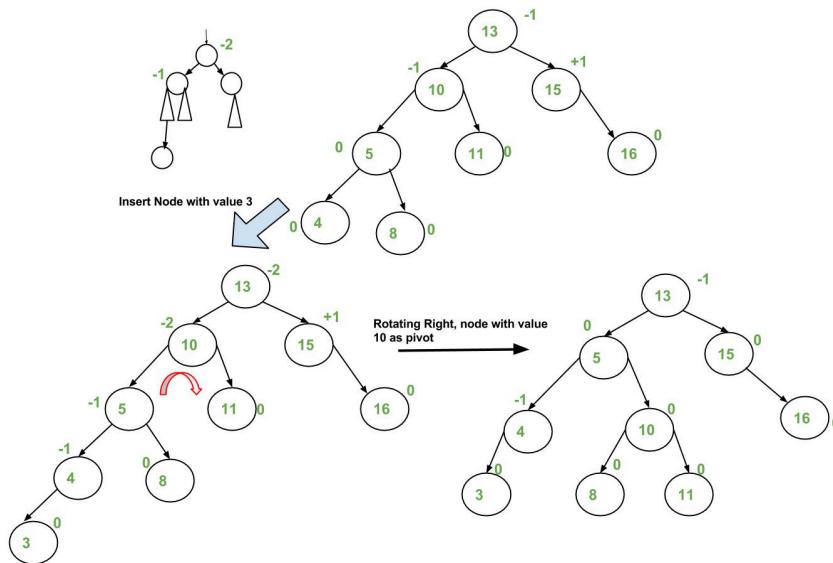
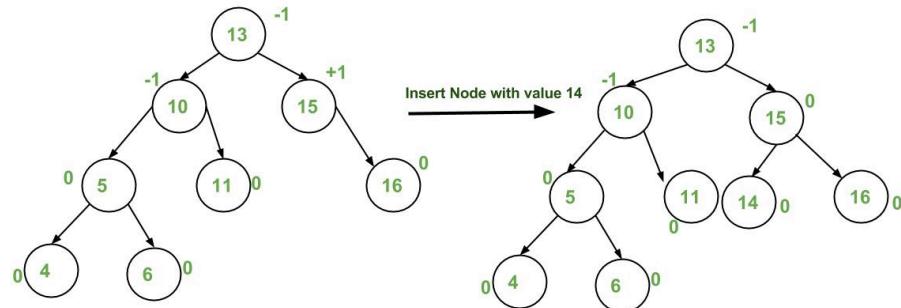
/ \

T2 T3

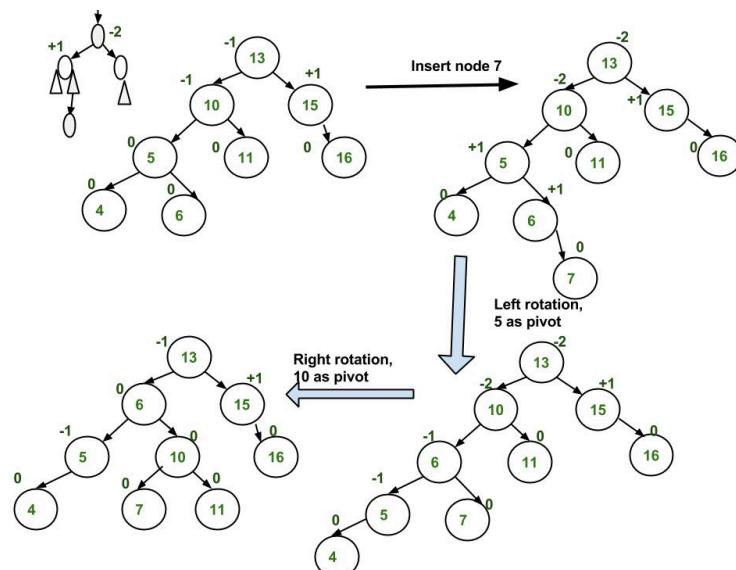
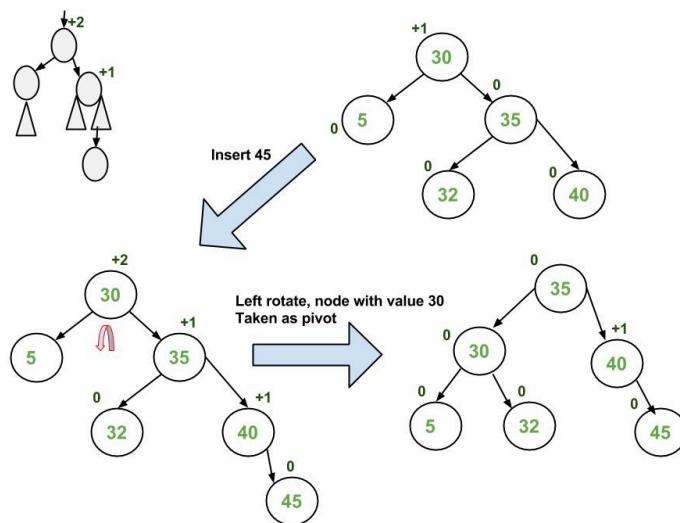
/ \

T3 T4

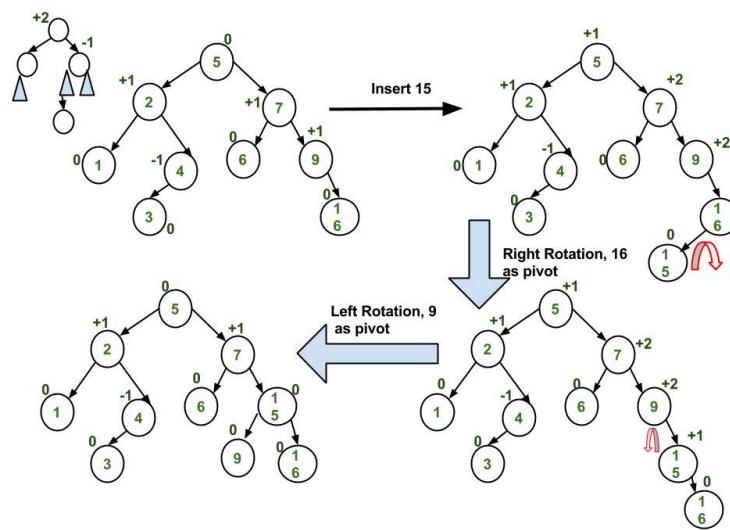
Illustration of Insertion at AVL Tree



We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).



We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).



Approach: The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

Follow the steps mentioned below to implement the idea:

- Perform the normal [BST insertion](#).
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor (**left subtree height – right subtree height**) of the current node.
- If the balance factor is greater than **1**, then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the **Right Right case** or **Right-Left case**. To check whether it is the **Right Right case** or not, compare the newly inserted key with the key in the **right subtree root**.

DSA Interview Problems on Tree Practice Tree MCQs on Tree Tutorial on Tree Types of Trees Basic operations

Below is the implementation of the above approach:

[C++14](#) [C](#) [Java](#) [Python](#) [C#](#) [JavaScript](#)

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

// An AVL tree node
struct Node {
    int key;
    Node *left;
    Node *right;
    int height;

    Node(int k) {
        key = k;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

// A utility function to
// get the height of the tree
int height(Node *N) {
    if (N == nullptr)
        return 0;
    return N->height;
}

// A utility function to right
// rotate subtree rooted with y
Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = 1 + max(height(y->left),
                          height(y->right));
    x->height = 1 + max(height(x->left),
                          height(x->right));

    // Return new root
    return x;
}

// A utility function to left rotate
// subtree rooted with x
Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = 1 + max(height(x->left));
}

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

    // Return new root
    return y;
}

// Get balance factor of node N
int getBalance(Node *N) {
    if (N == nullptr)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in
// the subtree rooted with node
Node* insert(Node* node, int key) {

    // Perform the normal BST insertion
    if (node == nullptr)
        return new Node(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    // Update height of this ancestor node
    node->height = 1 + max(height(node->left),
                           height(node->right));

    // Get the balance factor of this ancestor node
    int balance = getBalance(node);

    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
}

```

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

```

}

// A utility function to print
// preorder traversal of the tree
void preOrder(Node *root) {
    if (root != nullptr) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver Code
int main() {
    Node *root = nullptr;

    // Constructing tree given in the above figure
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
           30
         /   \
       20     40
      / \   /
    10  25  50
    */
    cout << "Preorder traversal : \n";
    preOrder(root);

    return 0;
}

```

Output

Preorder traversal :
 30 20 10 25 40 50

Time Complexity: O(log(n)), For Insertion

Auxiliary Space: O(Log n) for recursion call stack as we have written a recursive method to insert

The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of the

Comparison with Red Black Tree:

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in $O(\log n)$ time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then the AVL tree should be preferred over [Red Black Tree](#).

[AVL Tree | Set 2 \(Deletion\)](#)

[Comment](#)[More info](#)[Advertise with us](#)

Next Article

Insertion, Searching and Deletion in
AVL trees containing a parent node
pointer

Similar Reads

[AVL Tree Data Structure](#)

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one. The absolute difference between the heights o...

4 min read

[What is AVL Tree | AVL Tree meaning](#)

An AVL is a self-balancing Binary Search Tree (BST) where the difference between the heights of left and right subtrees of any node cannot be more than one. KEY POINTSIt is height balanced treeIt is a binary searc...

2 min read

[Insertion in an AVL Tree](#)

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Insertion, Searching and Deletion in AVL trees containing a parent node pointer

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. The insertion and deletion in AVL trees have been discussed...

15+ min read

Deletion in an AVL Tree

We have discussed AVL insertion in the previous post. In this post, we will follow a similar approach for deletion. Steps to follow for deletion. To make sure that the given tree remains AVL after every deletion, we...

15+ min read

How is an AVL tree different from a B-tree?

AVL Trees: AVL tree is a self-balancing binary search tree in which each node maintain an extra factor which is called balance factor whose value is either -1, 0 or 1. **B-Tree:** A B-tree is a self - balancing tree data structure...

1 min read

Practice questions on Height balanced/AVL Tree

AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node can't be more than 1. Here are some key points about AVL trees: If there are n...

4 min read

AVL with duplicate keys

Please refer below post before reading about AVL tree handling of duplicates. How to handle duplicates in Binary Search Tree? This is to augment AVL tree node to store count together with regular fields like key, left...

15+ min read

Count greater nodes in AVL tree

In this article we will see that how to calculate number of elements which are greater than given value in AVL tree. Examples: Input : x = 5 Root of below AVL tree 9 /\ 1 10 / \ 0 5 11 // \ -1 2 6 Output : 4 Explanation:...

15+ min read

Difference between Binary Search Tree and AVL Tree

Binary Search Tree: A binary Search Tree is a node-based binary tree data structure that has the following properties: The left subtree of a node contains only nodes with keys lesser than the node's key. The right...

2 min read



Corporate & Communications Address:

A-143, 7th Floor, Sovereign Corporate
Tower, Sector- 136, Noida, Uttar Pradesh
(201305)

Registered Address:

K 061, Tower K, Gulshan Vivante
Apartment, Sector 137, Noida, Gautam
Buddh Nagar, Uttar Pradesh, 201305



[Advertise with us](#)

Company

- [About Us](#)
- [Legal](#)
- [Privacy Policy](#)
- [In Media](#)
- [Contact Us](#)
- [Advertise with us](#)
- [GFG Corporate Solution](#)
- [Placement Training Program](#)
- [GeeksforGeeks Community](#)

DSA

- [Data Structures](#)
- [Algorithms](#)
- [DSA for Beginners](#)
- [Basic DSA Problems](#)
- [DSA Roadmap](#)
- [Top 100 DSA Interview Problems](#)
- [DSA Roadmap by Sandeep Jain](#)
- [All Cheat Sheets](#)

Web Technologies

- [HTML](#)
- [CSS](#)
- [JavaScript](#)

Languages

- [Python](#)
- [Java](#)
- [C++](#)
- [PHP](#)
- [GoLang](#)
- [SQL](#)
- [R Language](#)
- [Android Tutorial](#)
- [Tutorials Archive](#)

Data Science & ML

- [Data Science With Python](#)
- [Data Science For Beginner](#)
- [Machine Learning](#)
- [ML Maths](#)
- [Data Visualisation](#)
- [Pandas](#)
- [NumPy](#)
- [NLP](#)
- [Deep Learning](#)

Python Tutorial

- [Python Programming Examples](#)
- [Python Projects](#)
- [Python Tkinter](#)

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Bootstrap

Django

Web Design

Computer Science

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

Software Development

Software Testing

DevOps

Git

Linux

AWS

Docker

Kubernetes

Azure

GCP

DevOps Roadmap

System Design

High Level Design

Low Level Design

UML Diagrams

Interview Guide

Design Patterns

OOAD

System Design Bootcamp

Interview Questions

Interview Preparation

Competitive Programming

Top DS or Algo for CP

Company-Wise Recruitment Process

Company-Wise Preparation

Aptitude Preparation

Puzzles

School Subjects

Mathematics

Physics

Chemistry

Biology

Social Science

English Grammar

Commerce

World GK

GeeksforGeeks Videos

DSA

Python

Java

C++

Web Development

Data Science

CS Subjects

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved

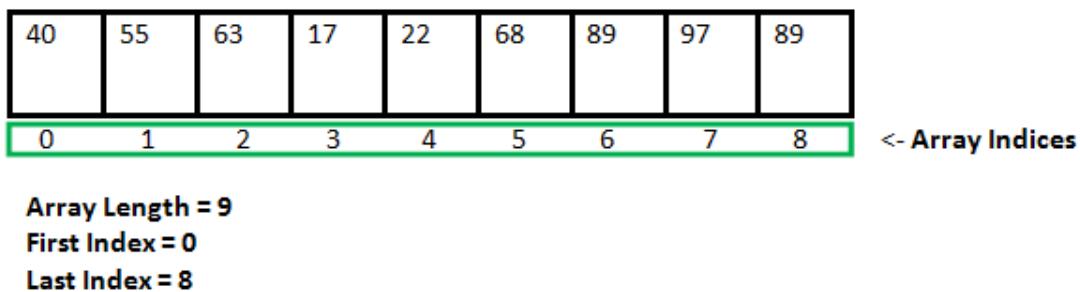
We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).



Linked List vs Array

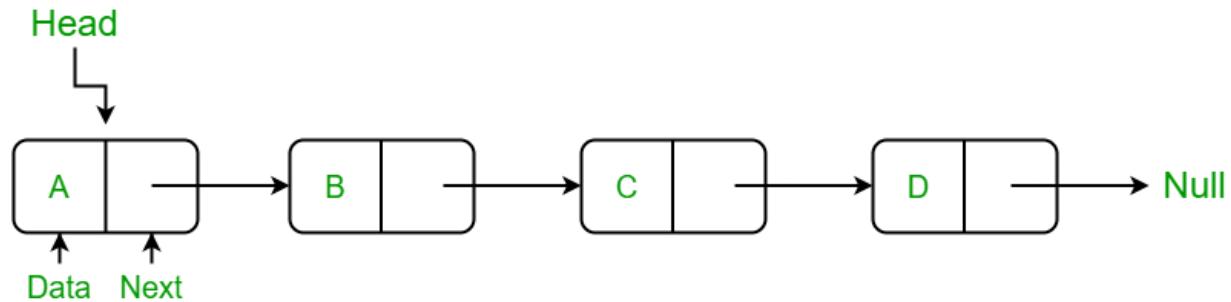
Last Updated : 17 Feb, 2025

Array: [Arrays](#) store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows faster access to an element at a specific index.



Data storage scheme of an array

Linked List: [Linked lists](#) are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element.



Linked-List representation

Advantages of Linked List over arrays :

- **Efficient insertion and deletion:** Linked lists allow insertion and deletion in the middle in $O(1)$ time, if we have a pointer to the target position, as only a few pointer changes are needed. In contrast, arrays require $O(n)$ time for insertion or deletion in the middle due to element shifting.

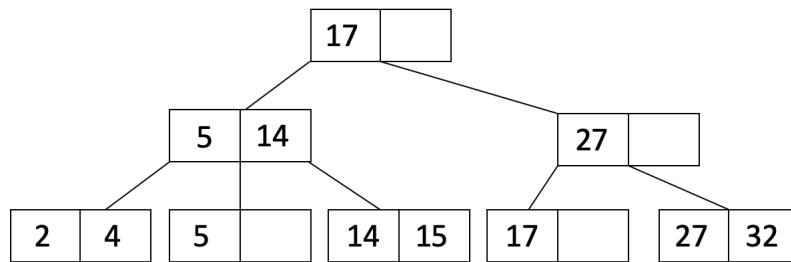
- **Implementation of Queue and Deque :** Simple array implementation is not efficient at all. We must use circular array to efficiently implement which is complex. But with linked list, it is easy and straightforward. That is why most of the language libraries use Linked List internally to implement these data structures.
- **Space Efficient in Some Cases :** Linked List might turn out to be more space efficient compare to arrays in cases where we cannot guess the number of elements in advance. In case of arrays, the whole memory for items is allocated together. Even with dynamic sized arrays like vector in C++ or list in Python or ArrayList in Java. the internal working involves de-allocation of whole memory and allocation of a bigger chunk when insertions happen beyond the current capacity.
- **Circular List with Deletion/Addition :** Circular Linked Lists are useful to implement CPU round robin scheduling or similar requirements in the real world because of the quick deletion/insertion in a circular manner.

Advantages of Arrays over Linked List :

- **Random Access.** : We can access i^{th} item in $O(1)$ time (only some basic arithmetic required using base address). In case of linked lists, it is $O(n)$ operation due to sequential access.
- **Cache Friendliness** : Array items (Or item references) are stored at contiguous locations which makes array cache friendly (Please refer [Spatial locality of reference](#) for more details)
- **Easy to use** : Arrays are relatively very easy to use and are available as core of programming languages
- **Less Overhead** : Unlike linked list, we do not have any extra references / pointers to be stored with every item.

1 Introduction

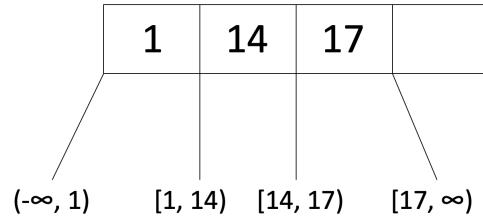
In the previous notes, we went over different file and record representations for data storage. This week, we will introduce index which is a data structure that operates on top of the data files and helps speed up reads on a specific key. You can think of data files as the actual content of a book and an index as the table of contents for fast lookup. We use indexes to make queries run faster, especially those that are run frequently. Consider a web application that looks up the record for a user in a Users table based on the username during the login process. An index on the username column will make login faster by quickly finding the row of the user trying to log in. In this course note, we will learn about B+ trees, which is a specific type of index. Here is an example of what a B+ tree looks like:



2 Properties

- The number d is the order of a B+ tree. Each node (with the exception of the root node) must have $d \leq x \leq 2d$ entries assuming no deletes happen (it's possible for leaf nodes to end up with $< d$ entries if you delete data). The entries within each node must be **sorted**.
- In between each entry of an inner node, there is a pointer to a child node. Since there are at most $2d$ entries in a node, inner nodes may have at most $2d + 1$ child pointers. This is also called the tree's fanout.
- The keys in the children to the left of an entry must be less than the entry while the keys in the children to the right must be greater than or equal to the entry.
- All leaves are at the same depth and have between d and $2d$ entries (i.e., at least half full)

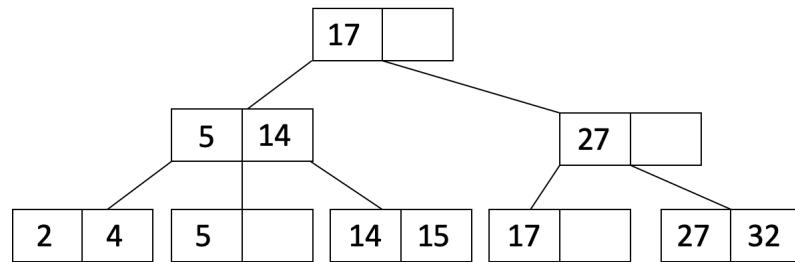
For example, here is a node of an order $d = 2$ tree:



Note that the node satisfies the order requirement, also known as the occupancy invariant ($d \leq x \leq 2d$) because $d = 2$ and this node has 3 entries which satisfies $2 \leq x \leq 4$.

- Because of the sorted and children property, we can traverse the tree down to the leaf to find our desired record. This is similar to BSTs (Binary Search Trees).
- Every root to leaf path has the same number of **edges** - this is the height of the tree. In this sense, B+ trees are **always** balanced. In other words, a B+ tree with just the root node has height 0.
- Only the leaf nodes contain records (or pointers to records - this will be explained later). The inner nodes (which are the non-leaf nodes) do not contain the actual records.

For example, here is an order $d = 1$ tree:



3 Insertion

To insert an entry into the B+ tree, follow this procedure:

- (1) Find the leaf node L in which you will insert your value. You can do this by traversing down the tree. Add the key and the record to the leaf node in order.

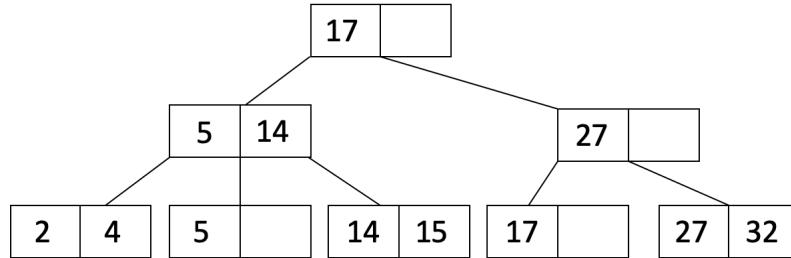
(2) If L overflows (L has more than $2d$ entries)...

- (a) Split into L_1 and L_2 . Keep d entries in L_1 (this means $d + 1$ entries will go in L_2).
- (b) If L was a leaf node, **COPY** L_2 's first entry into the parent. If L was not a leaf node, **MOVE** L_2 's first entry into the parent.
- (c) Adjust pointers.

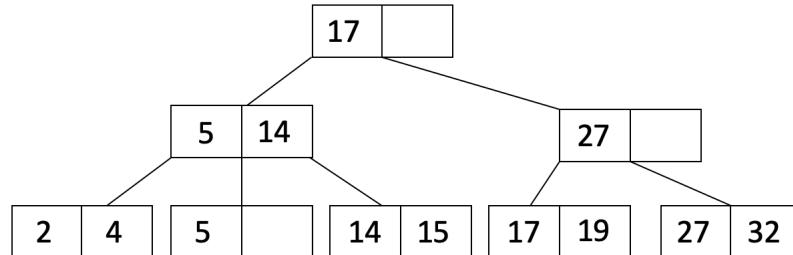
(3) If the parent overflows, then recurse on it by doing step 2 on the parent. (This is the only case that increases the tree height)

Note: we want to **COPY** leaf node data into the parent so that we don't lose the data in the leaf node. Remember that every key that is in the table that the index is built on must be in the leaf nodes! Being in a inner node does not mean that key is actually still in the table. On the other hand, we can **MOVE** inner node data into parent nodes because the inner node does not contain the actual data, they are just a reference of which way to search when traversing the tree.

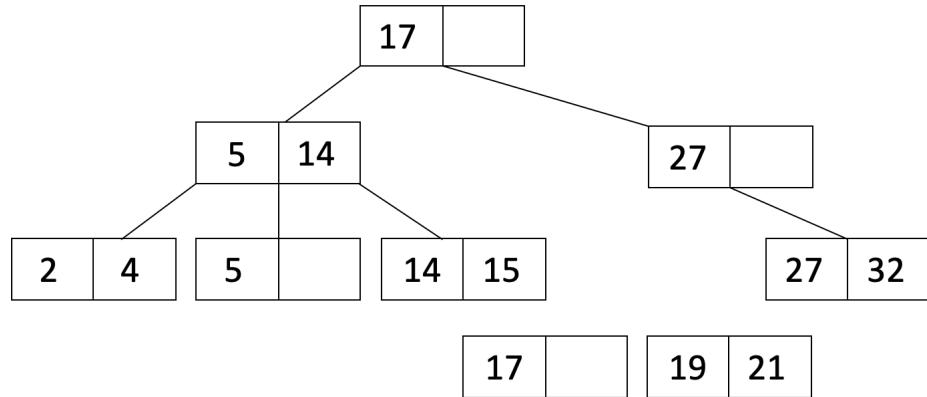
Let's take a look at an example to better understand this procedure! We start with the following order $d = 1$ tree:



Let's insert 19 into our tree. When we insert 19, we see that there is space in the leaf node with 17:

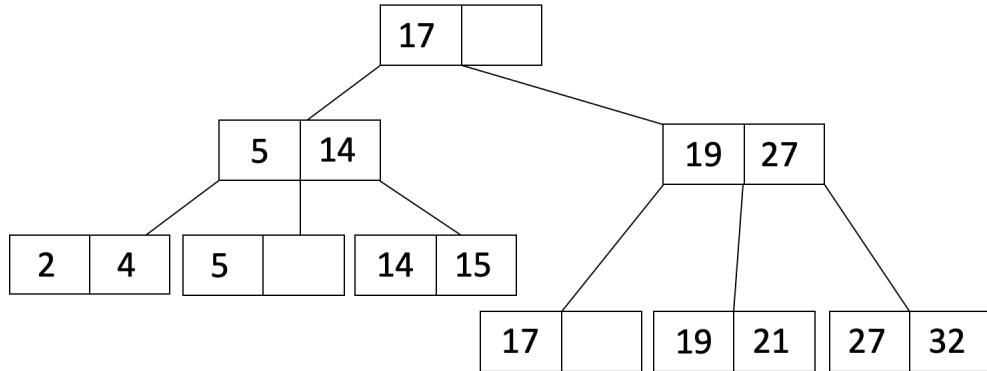


Now let's insert 21 into our tree. When we insert 21, it causes one of the leaf nodes to overflow. Therefore, we split this leaf node into two leaf nodes as shown below:

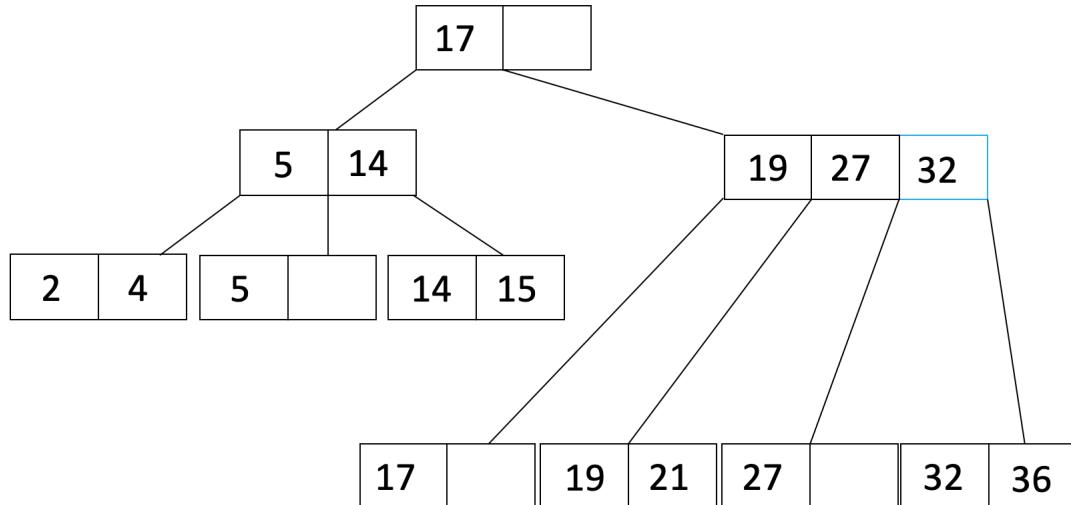


L_1 is the leaf node with 17, and L_2 is the leaf node with 19 and 21.

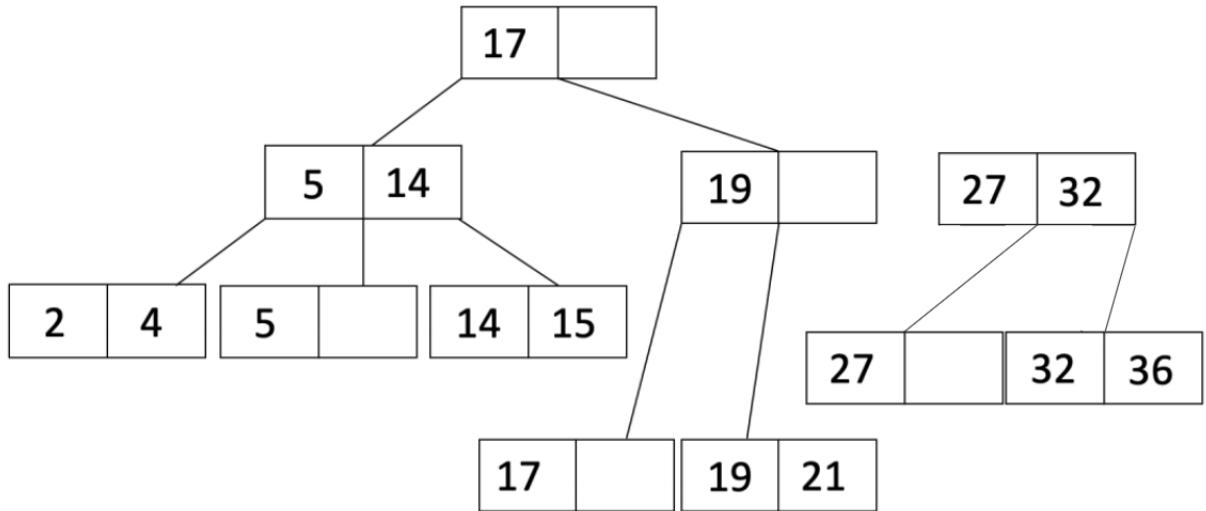
Since we split a leaf node, we will **COPY** L_2 's first entry up to the parent and adjust pointers. We also sort the entries of the parent to get:



Let's do one more insertion. This time we will insert 36. When we insert 36, the leaf overflows so we will do the same procedure as when we inserted 21 to get:

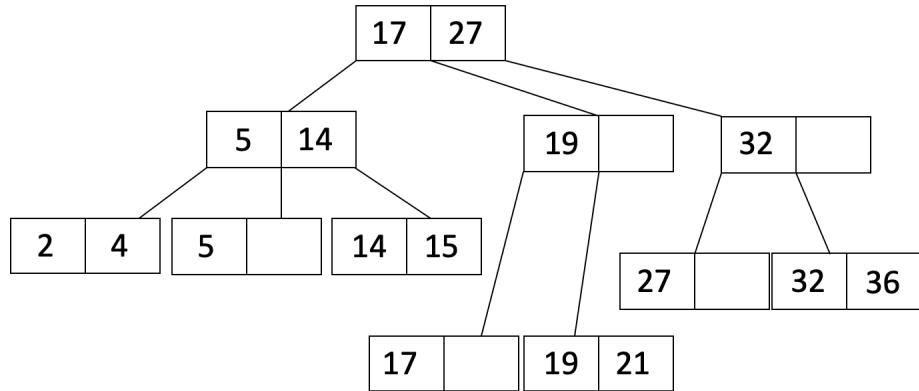


Notice that now the parent node overflowed, so now we must recurse. We will split the parent node to get:



L_1 is the inner node with 19, and L_2 is inner the node with 27 and 32.

Since it was an inner node that overflowed, we will **MOVE** L_2 's first entry up to the parent and adjust pointers to get:



Lastly, here are a couple clarifying notes about insertion into B+ Trees:

- Generally, B+ tree nodes have a minimum of d entries and a maximum of $2d$ entries. In other words, if the nodes in the tree satisfy this invariant before insertion (which they generally will), then after insertion, they will continue to satisfy it.
- Insertion overflow of the node occurs when it contains more than $2d$ entries.

4 Deletion

To delete a value, just find the appropriate leaf and delete the unwanted value from that leaf. That's all there is to it. (Yes, technically we could end up violating some of the invariants of a B+ tree. That's okay because in practice we get *way* more insertions than deletions so something will quickly replace whatever we delete.)

Reminder: We never delete inner node keys because they are only there for search and not to hold data.

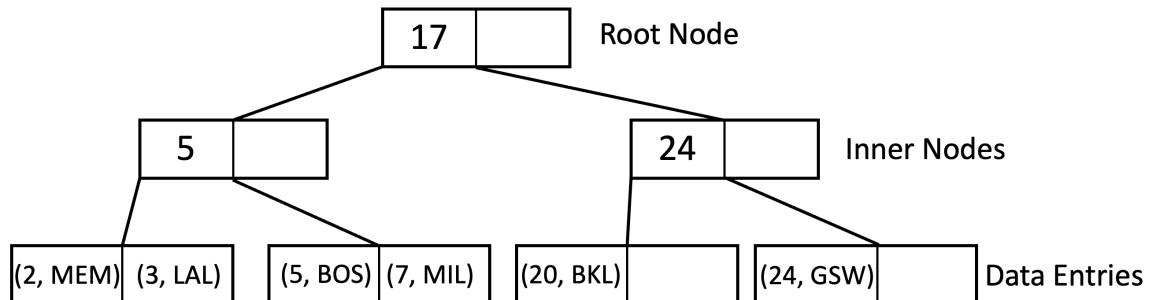
5 Storing Records

Up until now, we have not discussed how the records are actually stored in the leaves. Let's take a look at that now. There are three ways of storing records in leaves:

- **Alternative 1: By Value**

In the Alternative 1 scheme, the leaf pages are the data pages. Rather than containing pointers to records, the leaf pages contain the records themselves.

Points	Name
2	MEM
3	LAK
5	BOS
7	MIL
20	BKL
24	GSW

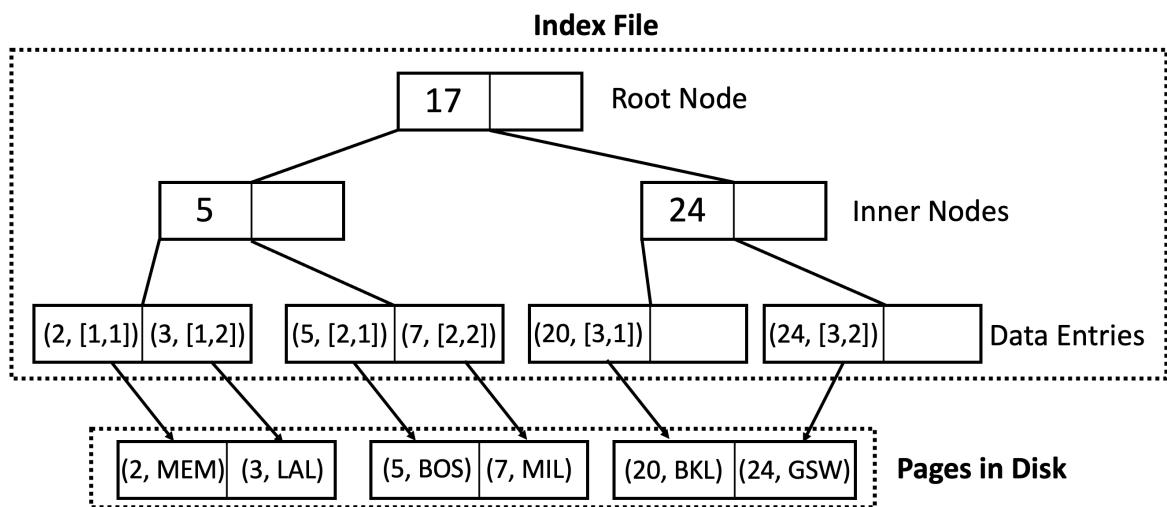


While the Alternative 1 Scheme is perhaps the easiest implementation, it has a significant limitation: if all we have is the Alternative 1 Scheme, we cannot support multiple indexes built on the same file (in the example above we cannot support a secondary index on 'name'). Instead we would have to duplicate the file and build a new Alternative 1 index on top of that file.

- Alternative 2: By Reference

In the Alternative 2 scheme, the leaf pages hold pointers to the corresponding records.

Points	Name
2	MEM
3	LAK
5	BOS
7	MIL
20	BKL
24	GSW



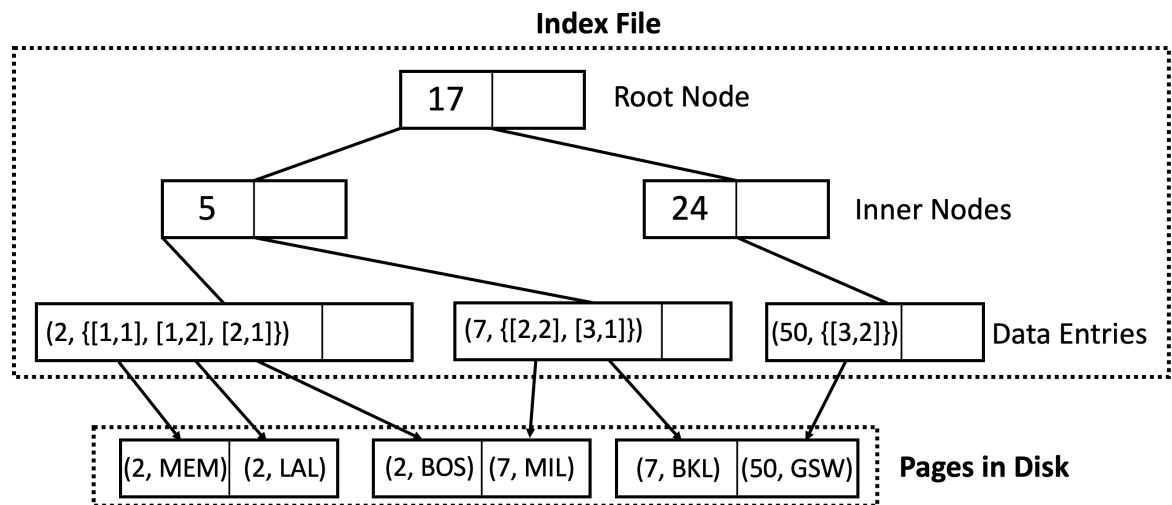
Notation Clarification: In the diagram above, the leafs contain (Key, RecordID) pairs where the RecordID is [PageNum, RecordNum].

Indexing by reference enables us to have multiple indexes on the same file because the actual data can lie in any order on disk.

- Alternative 3: By List of References

In the Alternative 3 scheme, the leaf pages hold lists of pointers to the corresponding records. This is more compact than Alternative 2 when there are multiple records with the same leaf node entry. Each leaf now contains (Key, List of RecordID) pairs.

Points	Name
2	MEM
2	LAK
2	BOS
7	MIL
7	BKL
50	GSW

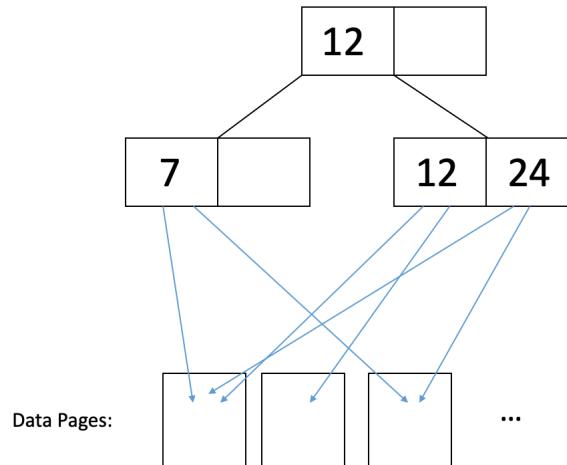


6 Clustering

Now that we've discussed how records are stored in the leaf nodes, we will also discuss how data on the data pages are organized. Clustered/unclustered refers to how the data pages are structured. Because the leaf pages are the actual data pages for Alternative 1 and keys are sorted on the index leaf pages, Alternative 1 indices are clustered by default. Therefore, unclustering only applies to Alternative 2 or 3.

- **Unclustered**

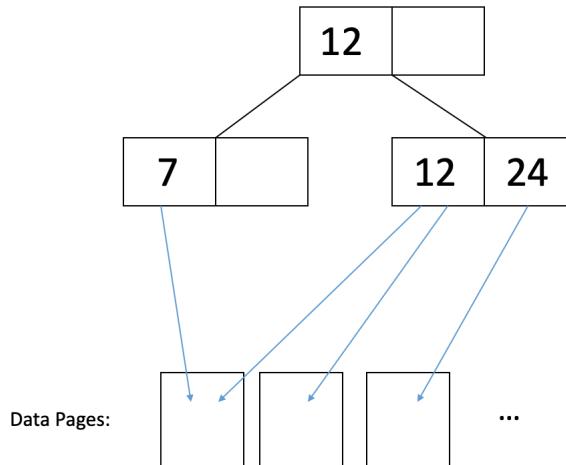
In an unclustered index, the data pages are complete chaos. Thus, odds are that you're going to have to read a separate page for each of the records you want. For instance, consider this illustration:



In the figure above, if we want to read records with 12 and 24, then we would have to read in each of the data pages they point to in order to retrieve all the records associated with these keys.

- **Clustered**

In a clustered index, the data pages are sorted on the same index on which you've built your B+ tree. This does not mean that the data pages are sorted exactly, just that keys are roughly in the same order as data. The difference in I/O cost therefore comes from caching, where two records with close keys will likely be in the same page, so the second one can be read from the cached page. Thus, you typically just need to read one page to get all the records that have a common / similar key. For instance, consider this illustration:



In the figure above, we can read records with 7 and 12 by reading 2 pages. If we do sequential reads of the leaf node values, the data page is largely the same. In conclusion,

- UNCLUSTERED = \sim 1 I/O per record.
- CLUSTERED = \sim 1 I/O per page of records.

- **Clustered vs. Unclustered Indexes**

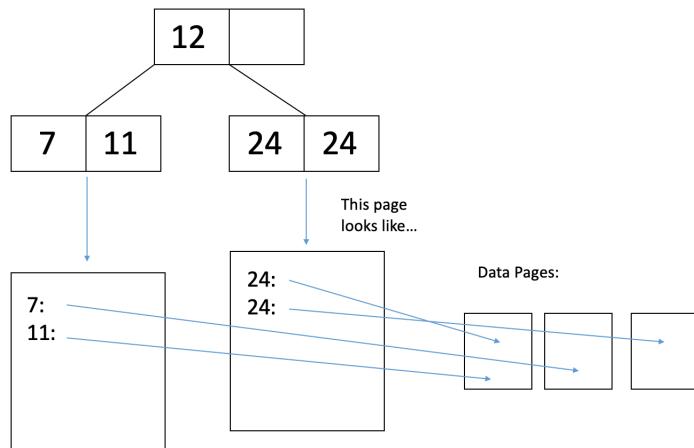
While clustered indexes can be more efficient for range searches and offer potential locality benefits during sequential disk access and prefetching, etc, they are usually more expensive to maintain than unclustered indexes. For example, the data file can become less clustered with more insertions coming in and, therefore, require periodical sorting of the file.

7 Counting IO's

Here's the general procedure. It's a good thing to write on your cheat sheet:

- (1) Read the appropriate root-to-leaf path.
- (2) Read the appropriate data page(s). If we need to read multiple pages, we will allot a read IO for each page. In addition, we account for clustering for Alt. 2 or 3 (see below.)
- (3) Write data page, if you want to modify it. Again, if we want to do a write that spans multiple data pages, we will need to allot a write IO for each page.
- (4) Update index page(s).

Let's look at an example. See the following **Alternative 2 Unclustered B+** tree:



We want to delete the only 11-year-old from our database. How many I/Os will it take?

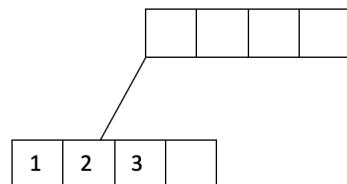
- One I/O for each of the 2 relevant index pages on the roof-to-leaf path (an index page is an inner node or a leaf node).
- One I/O to read the data page where the 11-year-old's record is. Once it's in memory we can delete the record from the page.
- One I/O to write the modified data page back to disk.
- Now that there are no more 11-year-olds in our database we should remove the key "11" from the leaf page of our B+ tree, which we already read in Step 1. We do so, and then it takes one I/O to write the modified leaf page to disk.
- Thus, the total cost to delete the record was 5 I/Os.

8 Bulk Loading

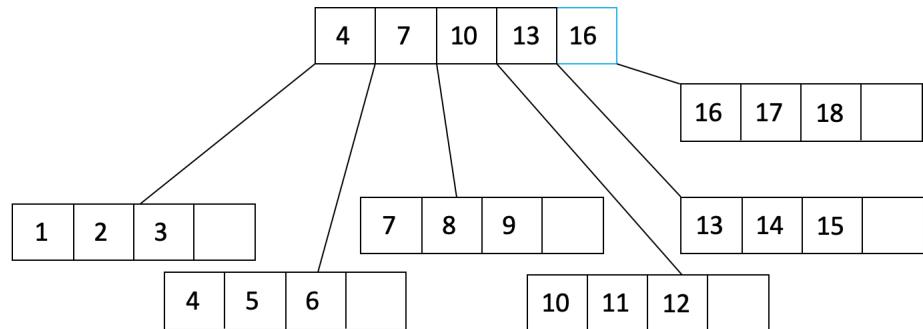
The insertion procedure we discussed before is great for making additions to an existing B+ tree. If we want to construct a B+ tree from scratch, however, we can do better. This is because if we use the insertion procedure we would have to traverse the tree each time we want to insert something new; regular insertions into random data pages also lead to poor cache efficiency and poor utilization of the leaf pages as they are typically half-empty. Instead, we will use **bulkloading**:

- (1) Sort the data on the key the index will be built on.
- (2) Fill leaf pages until some fill factor f . Note that fill factor only applies to leaf nodes. For inner nodes, we still follow the same rule to insert until they are full.
- (3) Add a pointer from parent to leaf page. If the parent overflows, we will follow a procedure similar to insertion. We will split the parent into two nodes:
 - (a) Keep d entries in L_1 (this means $d + 1$ entries will go in L_2).
 - (b) Since a parent node overflowed, we will **MOVE** L_2 's first entry into the parent.
- (4) Adjust pointers.

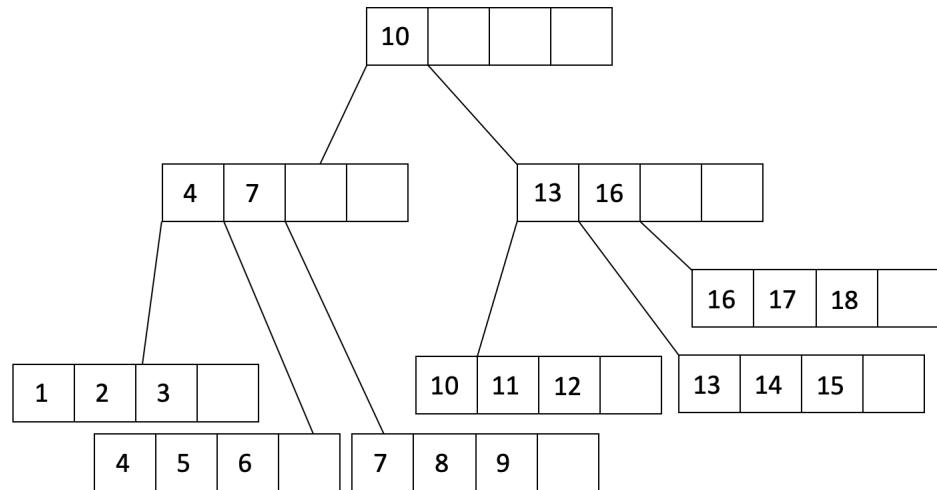
Let's look at an example. Let's say our fill factor is $\frac{3}{4}$ and we want to insert $1, \dots, 20$ into an order $d = 2$ tree. We will start by filling a leaf page until our fill factor:



We have filled a leaf node to the fill factor of $\frac{3}{4}$ and added a pointer from the parent node to the leaf node. Let's continue filling:



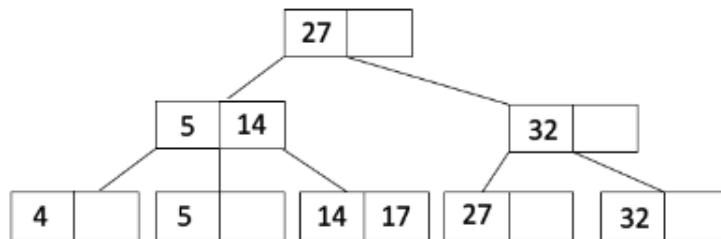
In the figure above, we see that the parent node has overflowed. We will split the parent node into two nodes and create a new parent:



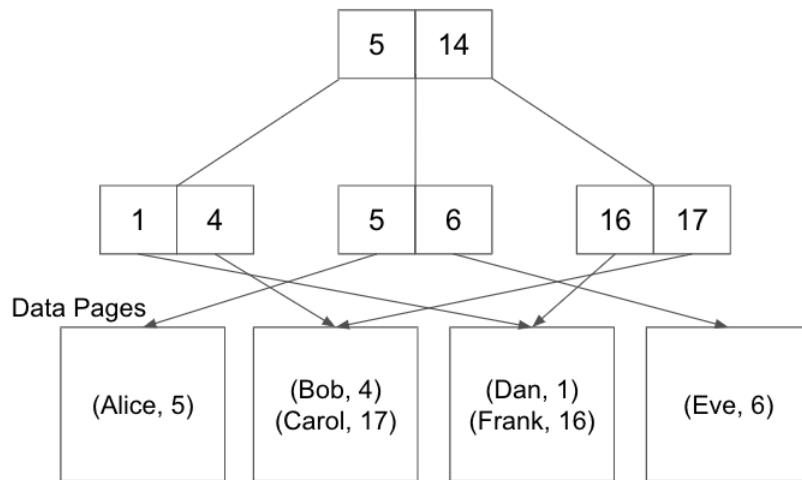
As you can see from the above example, indexes built from bulkloading always start off clustered because the underlying data is sorted on the key. To maintain clustering, we can choose the fill factor depending on future insertion patterns.

9 Practice Questions

- What is the maximum number of entries we can add to the following B+ tree without increasing its height?



- What is the minimum number of entries we can add to the B+ tree above which will cause the height to increase?
- How many I/Os would it cost to insert an entry into our table if we had a height 2, unclustered alternative 3 B+ tree in the worst case? Assume that the cache is empty at the beginning and there are at most 10 entries with the same key. Assume each page is at most $\frac{2}{3}$ full.
- Assume we have the following Alternative 2 unclustered B+ tree built over the age field of a relation. How many I/Os will it take to find all records where $age \geq 6$?



- What is the maximum fan-out of an order d B+ tree? Express your answer as a function of d .

6. What is the maximum number of keys an order 4 B+ tree with height 5 can store? (Remember that a tree of height 0 only contains a root node while a tree of height 1 contains a root node and its leaves.)

10 Solutions

1. The maximum number of entries we can add without increasing the height is the total capacity of a height 2, order $d = 1$ tree minus the current number of entries. The total capacity is $(2d)(2d + 1)^h = (2)(3^2) = 18$. The current number of entries is 6 because the entries are in the leaf nodes. Therefore, we can add a maximum of $18 - 6 = 12$.
2. The minimum number of entries we can add to cause the height to increase will be 3.

We will add 20 to the fullest leaf node which is 14, 17 which will cause it to split into 14 and 17, 20. Then we copy 17 to its parent 5, 14, which causes it to split into 5 and 17 and 14 will be pushed up so the root will have 14, 27.

Then we will insert 21 into the fullest leaf which will now be 17, 20, which will cause it to split into 17 and 20, 21. Then we copy 20 up to its parent which will become 17, 20.

Then we can insert 22 into the fullest leaf 20, 21 which will split into 20 and 21, 22. Then we copy 21 up to the parent 17, 20, which will cause it to split into 17 and 21 because 20 will be pushed up to its parent (the root). This root will also split because it already has 14, 27. When the root splits, we know that the height has increased.

Note that any three numbers between 17 and 27 could be added to cause the height of the tree to increase to 3.

3. First, we need to check if our table already contains this entry. We will search down the B+ tree for the key which will cost us 3 I/Os because the tree is height 2. Then we get to a leaf node and we see that there are at most 10 entries with the same key. We need to check each of these entries to make sure it's not the same as our current entry which is 10 I/Os because our B+ tree is unclustered. In the worst case, none of the 10 entries match the entry we want to insert so we will have to go ahead and add it to the table. We will add the entry to any page that has space (we already have this page in our cache from the previous part so this does not incur an I/O) and then writing this page back to disk which will cost 1 I/O. We also have to update our B+ tree leaf node to include a pointer to this new entry so we will have to write the node back to disk which is 1 I/O. Therefore, we will have a total of $3 + 10 + 1 + 1 = 15$ I/Os.
4. 6 I/Os. The general idea is that we need to look up which leaf node corresponds to age = 6 and scan through all leaf nodes to the right. Since this is an Alternative 2 B+ tree, for each leaf node we will also need to read in the data pages the records lie on (1 I/O per matching record since the index is unclustered).

We will incur:

- 1 I/O to read in the root node: 5, 14
- 1 I/O to read in the 5, 6 leaf node

- 1 I/O to read in the 4th data page, which contains (Eve, 6)
 - 1 I/O to read in the 16, 17 leaf node
 - 1 I/O to read in the 3rd data page, which contains (Frank, 16)
 - 1 I/O to read in the 2nd data page, which contains (Carol, 17)
5. The maximum fan-out is the maximum occupancy plus one, $2d + 1$ by definition.
 6. 472,392. An order of 4 means that the fan-out is 9. At height 0, there is only one node. At height 1, there are at most $9^1 = 9$ leaf nodes. At height 2, there are at most $9^2 = 81$ leaf nodes. At height 5, there are at most $9^5 = 59,049$ leaf nodes. Each node can store 8 keys. So $59,049 * 8 = 472,392$ keys.

11 Past Exam Problems

- Sp22 Quiz Q3
- Sp22 Final Q3
- Fa21 Midterm 1 Q4
- Fa21 Final Q4
- Sp21 Midterm 1 Q3
- Sp21 Final Q1
- Fa20 Midterm 1 Q4



Practice questions on Height balanced/AVL Tree

Last Updated : 28 Dec, 2024

AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node can't be more than 1.

Here are some key points about [AVL trees](#):

- If there are n nodes in AVL tree, minimum height of AVL tree is $\text{floor}(\log_2 n)$.
- If there are n nodes in AVL tree, maximum height can't exceed $1.44 * \log_2 n$.
- If height of AVL tree is h, maximum number of nodes can be $2^{h+1} - 1$.
- Minimum number of nodes in a tree with height h can be represented as:

$$N(h) = N(h-1) + N(h-2) + 1 \text{ for } n > 2 \text{ where } N(0) = 1 \text{ and } N(1) = 2.$$
- The complexity of searching, inserting and deletion in AVL tree is $O(\log n)$.

We have discussed types of questions based on AVL trees.

Type 1: Relationship between number of nodes and height of AVL tree –

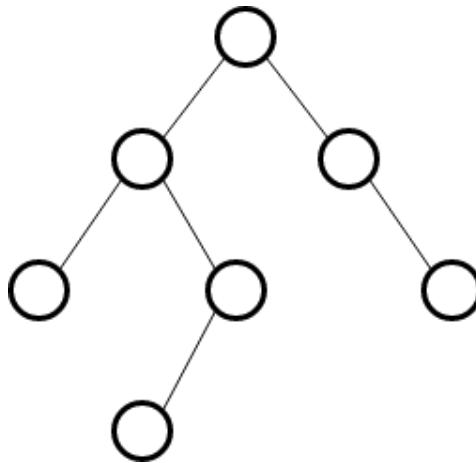
Given number of nodes, the question can be asked to find minimum and maximum height of AVL tree. Also, given the height, maximum or minimum number of nodes can be asked.

Que – 1. What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0.

- (A) 2
- (B) 3
- (C) 4
- (D) 5

Solution: For finding maximum height, the nodes should be minimum at each level. Assuming height as 2, minimum number of nodes required: $N(h) = N(h-1)$

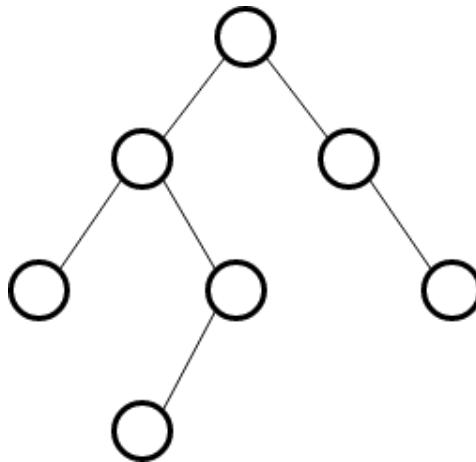
7. It means, height 3 is achieved using minimum 7 nodes. Therefore, using 7 nodes, we can achieve maximum height as 3. Following is the AVL tree with 7 nodes and height 3.



Que – 2. What is the worst case possible height of AVL tree?

- (A) $2 * \log n$
- (B) $1.44 * \log n$
- (C) Depends upon implementation
- (D) $\theta(n)$

Solution: The worst case possible height of AVL tree with n nodes is $1.44 * \log n$. This can be verified using AVL tree having 7 nodes and maximum height.



Checking for option (A), $2 * \log 7 = 5.6$, however height of tree is 3.

Checking for option (B). $1.44 * \log 7 = 4$. which is near to 3.

Out of these, option (B) is the best possible answer.

Type 2: Based on complexity of insertion, deletion and searching in AVL tree

—

Que – 3. Which of the following is TRUE?

- (A) The cost of searching an AVL tree is $\Theta(\log n)$ but that of a binary search tree is $O(n)$
- (B) The cost of searching an AVL tree is $\Theta(\log n)$ but that of a complete binary tree is $\Theta(n \log n)$
- (C) The cost of searching a binary search tree is $O(\log n)$ but that of an AVL tree is $\Theta(n)$
- (D) The cost of searching an AVL tree is $\Theta(n \log n)$ but that of a binary search tree is $O(n)$

Solution: AVL tree's time complexity of searching, insertion and deletion = $O(\log n)$. But a binary search tree, may be skewed tree, so in worst case BST searching, insertion and deletion complexity = $O(n)$.

Que – 4. The worst case running time to search for an element in a balanced in a binary search tree with $n*2^n$ elements is

- (A) $\Theta(n \log n)$
- (B) $\Theta(n 2^n)$
- (C) $\Theta(n)$
- (D) $\Theta(\log n)$

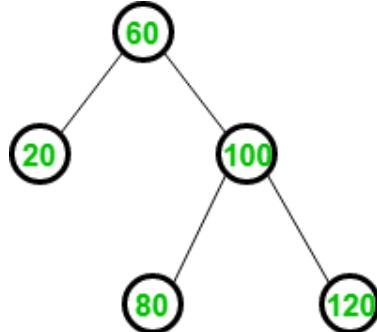
Solution: Time taken to search an element is $\Theta(\log n)$ where n is number of elements in AVL tree. As number of elements given is $n*2^n$, the searching complexity will be $\Theta(\log(n*2^n))$ which can be written as:

$$\begin{aligned}
 &= \Theta(\log(n*2^n)) \\
 &= \Theta(\log(n)) + \Theta(\log(2^n)) \\
 &= \Theta(\log(n)) + \Theta(n \log(2)) \\
 &= \Theta(\log(n)) + \Theta(n)
 \end{aligned}$$

As $\log n$ is asymptotically smaller than n , $\Theta(\log(n)) + \Theta(n)$ can be written as

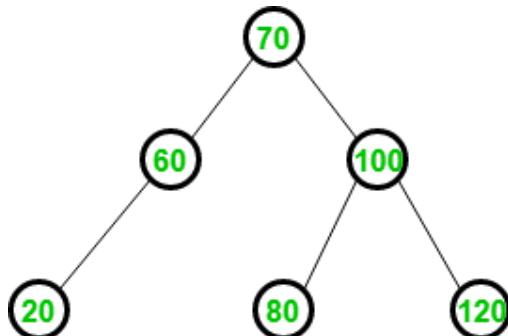
Type 3: Insertion and Deletion in AVL tree – The question can be asked on the resultant tree when keys are inserted or deleted from AVL tree. Appropriate rotations need to be made if balance factor is disturbed.

Que – 5. Consider the following AVL tree.

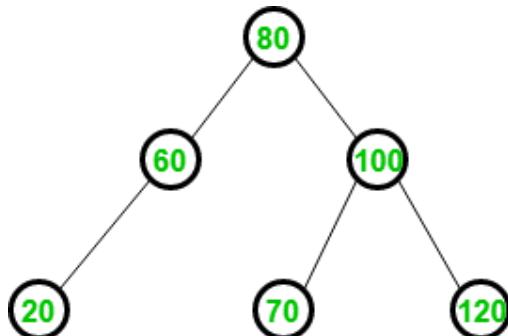


Which of the following is updated AVL tree after insertion of 70?

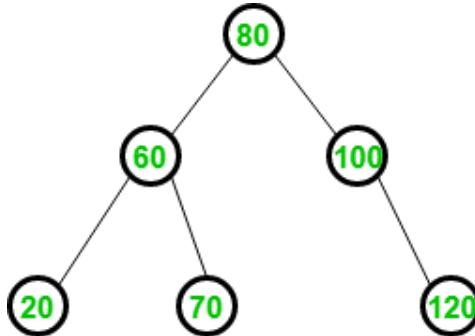
- (A)



- (B)

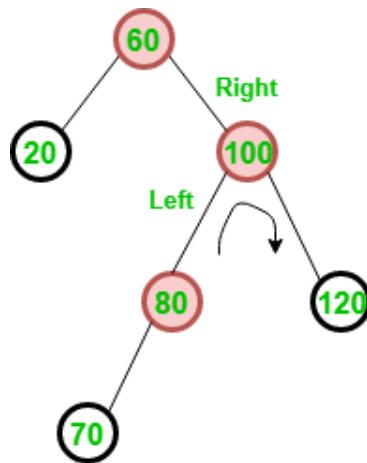


- (C)

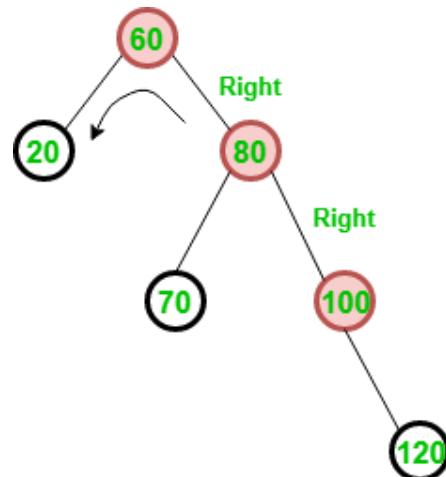


- (D) None

Solution: The element is first inserted in the same way as BST. Therefore after insertion of 70, BST can be shown as:



However, balance factor is disturbed requiring RL rotation. To remove RL rotation, it is first converted into RR rotation as:



After removal of RR rotation, AVL tree generated is same as option (C).



Relationship between number of nodes and height of binary tree

Last Updated : 01 Feb, 2025

Binary Tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used in computer science for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal.

Prerequisite – [Binary Tree Data Structure](#)

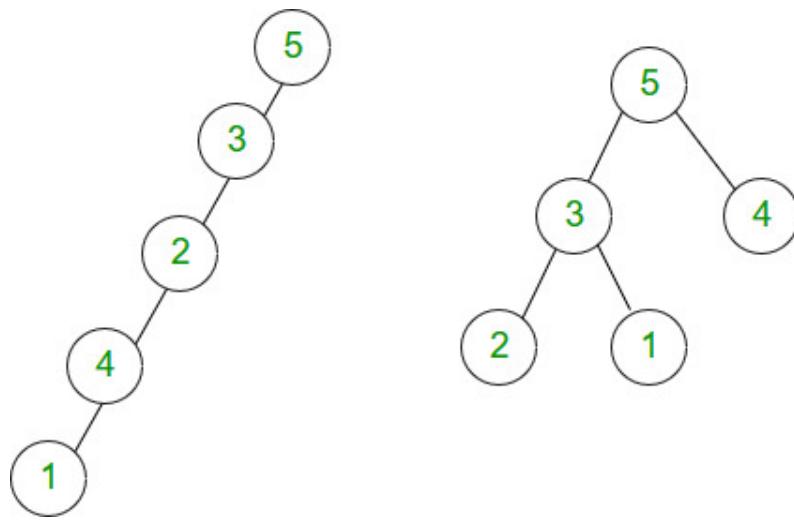
Height of Binary Tree

The **height** of the binary tree is the longest path from root node to any leaf node in the tree. For example, the height of binary tree shown in Figure 1(b) is 2 as longest path from root node to node 2 is 2. Also, the height of binary tree shown in Figure 1(a) is 4.

Calculating minimum and maximum height from the number of nodes –

If there are n nodes in a binary tree, the **maximum height** of the binary tree is $n-1$, and the **minimum height** is $\text{floor}(\log_2(n))$.

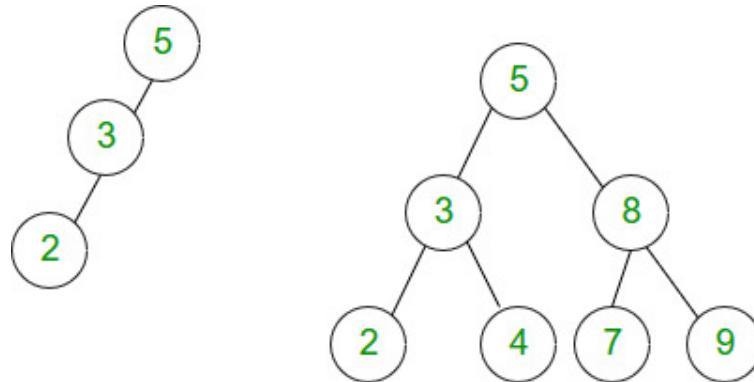
For example, the left skewed binary tree shown in Figure 1(a) with 5 nodes has a height of $5-1 = 4$, and the binary tree shown in Figure 1(b) with 5 nodes has a height $\text{floor}(\log_2 5) = 2$.



Calculating minimum and maximum number of nodes from height:

If binary tree has **height h**, minimum number of **nodes is $h+1$** (in case of left skewed and right skewed binary tree). For example, the binary tree shown in Figure 2(a) with height 2 has 3 nodes.

If binary tree has height h , maximum number of nodes will be when all levels are completely full. Total number of nodes will be $2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$. For example, the binary tree shown in Figure 2(b) with height 2 has $2^{2+1} - 1 = 7$ nodes.



Questions on Finding Height of Binary Tree

Question-1. The height of a tree is the length of the longest root-to-leaf path in it. The maximum and the minimum number of nodes in a binary tree of height 5 are:

- (A) 63 and 6, respectively

- (D) 31 and 5, respectively

Solution:

*According to formula discussed,
max number of nodes = $2^{h+1}-1 = 2^6-1 = 63.$
min number of nodes = $h+1 = 5+1 = 6.$*

Question-2. Which of the following height is not possible for a binary tree with 50 nodes?

- (A) 4
- (B) 5
- (C) 6
- (D) None

Solution:

*According to formula discussed,
Minimum height with 50 nodes = $\text{floor}(\log_2(50)) = 5.$ Therefore, height 4 is not possible.*

[Comment](#)
[More info](#)
[Advertise with us](#)

Next Article

Matrices

Similar Reads

Bayes's Theorem for Conditional Probability

Bayes's Theorem for Conditional Probability: Bayes's Theorem is a fundamental result in probability theory that describes how to update the probabilities of hypotheses when given evidence. Named after the Reveren...

0 min read

We use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our [Cookie Policy](#) & [Privacy Policy](#).

Section 05: Solutions

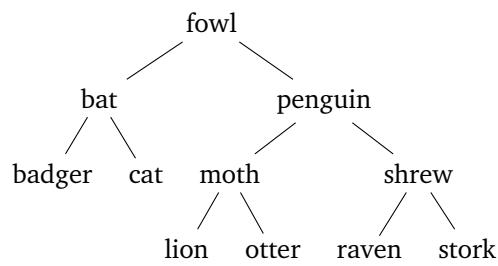
1. Constructing AVL trees

Draw an AVL Tree as each of the following keys are added in the order given. Show intermediate steps.

(a)

{“penguin”, “stork”, “cat”, “fowl”, “moth”, “badger”, “otter”, “shrew”, “lion”, “raven”, “bat”}

Solution:

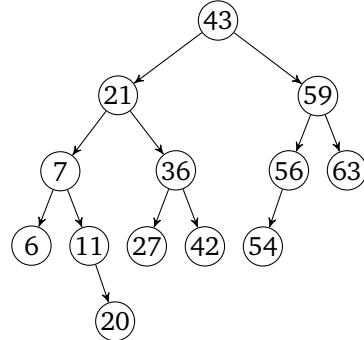


(b)

{6, 43, 7, 42, 59, 63, 11, 21, 56, 54, 27, 20, 36}

Solution:

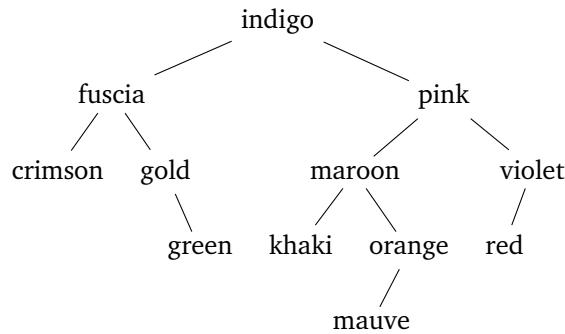
Note: The Section slides have a step-by-step walkthrough of this one!



(c)

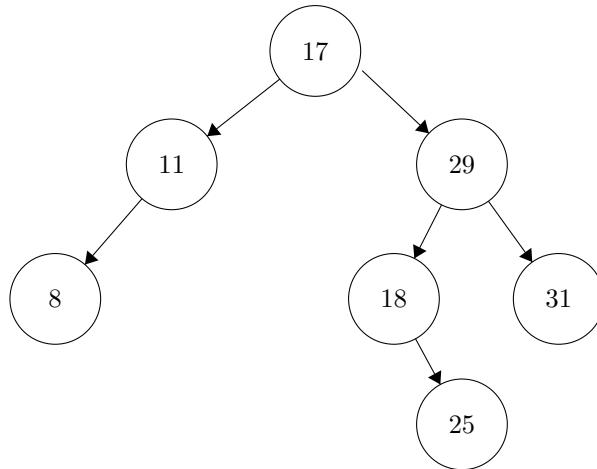
{“indigo”, “fuscia”, “pink”, “goldenrod”, “violet”, “khaki”, “red”, “orange”, “maroon”, “crimson”, “green”, “mauve”}

Solution:



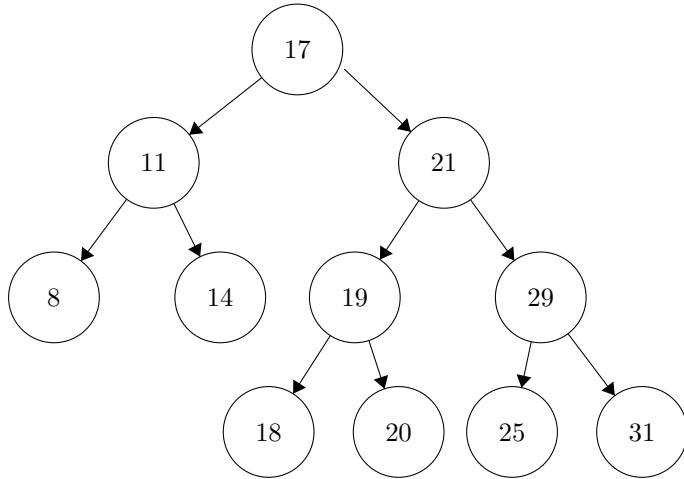
2. Inserting keys

Consider the following AVL tree:



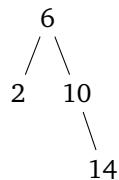
We now add the keys {21, 14, 20, 19} (in that order). Show where these keys are added to the AVL tree. Show your intermediate steps.

Solution:



3. AVL tree rotations

Consider this AVL tree:



Give an example of a value you could insert to cause:

- (a) A single rotation

Solution:

Any value greater than 14 will cause a single rotation around 10 (since 10 will become unbalanced, but we'll be in the line case).

- (b) A double rotation

Solution:

Any value between 10 and 14 will cause a double rotation around 10 (since 10 will be unbalanced, and we'll be in the kink case).

- (c) No rotation

Solution:

Any value less than 10 will cause no rotation (since we can't cause any node to become unbalanced with those values).

4. True or false?

- (a) An insertion in an AVL tree with n nodes requires $\Theta(\log(n))$ rotations.

Solution:

False. Each insertion will require either no rotations, a single rotation, or a double rotation. So, the total number of rotations is in $\Theta(1)$.

- (b) A set of numbers are inserted into an empty BST in sorted order and inserted into an empty AVL tree in random order. Listing all elements in sorted order from the BST is $\mathcal{O}(n)$, while listing them in sorted order from the AVL tree is $\mathcal{O}(\log(n))$.

Solution:

False. Although it is true that listing all elements in sorted order from a BST is $\mathcal{O}(n)$, the statement as a whole is false because an AVL tree traversal is also $\mathcal{O}(n)$, since we still have to look at every node once to traverse the tree.

- (c) If items are inserted into an empty BST in sorted order, then the BST's `get()` is just as asymptotically efficient as an AVL tree whose elements were inserted in unsorted order.

Solution:

False. If items are inserted into a BST in sorted order, it produces a linked list.

In that case, `get()` would take $\mathcal{O}(n)$ time.

- (d) An AVL tree will always do a maximum of two rotations in an insert.

Solution:

True. For an intuition on why this is true, notice that an insertion causes an imbalance because the problem node has one subtree of height k , and the other subtree had height $k + 1$, and the insertion occurred on the subtree with height $k + 1$ to make it height $k + 2$.

Then, our rotation will rebalance the tree rooted from the problem node's position such that each subtree is height $k + 1$. But since the tree prior to insertion was balanced when the problem node had height $k + 2$, and the problem node after rotation still has height $k + 2$, and the problem node is also now balanced, the tree must now be completely balanced.

(We know that rotations do not introduce more imbalances below them, since they are a method of fixing imbalances.)

5. Big- \mathcal{O}

Write down a tight big- \mathcal{O} for each of the following. Unless otherwise noted, give a bound in the worst case.

- (a) Insert and find in a BST.

Solution:

$\mathcal{O}(n)$ and $\mathcal{O}(n)$, respectively. This is unintuitive, since we commonly say that `find()` in a BST is “ $\log(n)$ ”, but we’re asking you to think about *worst-case* situations. The worst-case situation for a BST is that the tree is a linked list, which causes `find()` to reach $\mathcal{O}(n)$.

- (b) Insert and find in an AVL tree.

Solution:

$\mathcal{O}(\log(n))$ and $\mathcal{O}(\log(n))$, respectively. The worst case is we need to insert or find a node at height 0. However, an AVL tree is always a balanced BST tree, which means we can do that in $\mathcal{O}(\log(n))$.

- (c) Finding the minimum value in an AVL tree containing n elements.

Solution:

$\mathcal{O}(\log(n))$. We can find the minimum value by starting from the root and constantly traveling down the left-most branch.

- (d) Finding the k -th largest item in an AVL tree containing n elements.

Solution:

With a standard AVL tree implementation, it would take $\mathcal{O}(n)$ time. If we’re located at a node, we have no idea how many elements are located on the left vs right and need to do a traversal to find out. We end up potentially needing to visit every node.

If we modify the AVL tree implementation so every node stored the number of children it had at all times (and updated that field every time we insert or delete), we could do this in $\mathcal{O}(\log(n))$ time by performing a binary search style algorithm.

- (e) Listing elements of an AVL tree in sorted order

Solution:

$\mathcal{O}(n)$. An AVL tree is always a balanced BST tree, which means we only need to traverse the tree in in-order once.

6. Analyzing dictionaries

- (a) What are the constraints on the data types you can store in an AVL tree?

Solution:

The keys need to be orderable because AVL trees (and BSTs too) need to compare keys with each other to decide whether to go left or right at each node. (In Java, this means they need to implement Comparable). Unlike a hash table, the keys do *not* need to be hashable. (Note that in Java, every object is technically hashable, but it may not hash to something based on the object's value. The default hash function is based on reference equality.)

The values can be any type because AVL trees are only ordered by keys, not values.

- (b) When is using an AVL tree preferred over a hash table?

Solution:

- (i) You can iterate over an AVL tree in sorted order in $\mathcal{O}(n)$ time.
- (ii) AVL trees never need to resize, so you don't have to worry about insertions occasionally being very slow when the hash table needs to resize.
- (iii) In some cases, comparing keys may be faster than hashing them. (But note that AVL trees need to make $\mathcal{O}(\log n)$ comparisons while hash tables only need to hash each key once.)
- (iv) AVL trees *may* be faster than hash tables in the worst case since they guarantee $\mathcal{O}(\log n)$, compared to a hash table's $\mathcal{O}(n)$ if every key is added to the same bucket. But remember that this only applies to pathological hash functions. In most cases, hash tables have better asymptotic runtime ($\mathcal{O}(1)$) than AVL trees, and in practice $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ have roughly the same performance.

- (c) When is using a BST preferred over an AVL tree?

Solution:

One of AVL tree's advantages over BST is that it has an asymptotically efficient find even in the worst case.

However, if you know that `insert` will be called more often than `find`, or if you know the keys will be inserted in a random enough order that the BST will stay balanced, you may prefer a BST since it avoids the small runtime overhead of checking tree balance properties and performing rotations. (Note that this overhead is a constant factor, so it doesn't matter asymptotically, but may still affect performance in practice.)

BSTs are also easier to implement and debug than AVL trees.

- (d) Consider an AVL tree with n nodes and a height of h . Now, consider a single call to `get(...)`. What's the maximum possible number of nodes `get(...)` ends up visiting? The minimum possible?

Solution:

The max number is $h + 1$ (remember that height is the number of edges, so we visit $h + 1$ nodes going from the root to the farthest away leaf); the min number is 1 (when the element we're looking for is just the root).

- (e) **Challenge Problem:** Consider an AVL tree with n nodes and a height of h . Now, consider a single call to `insert(...)`. What's the maximum possible of nodes `insert(...)` ends up visiting? The minimum possible? Don't count the new node you create or the nodes visited during rotation(s).

Solution:

The max number is $h + 1$. Just like a `get`, we may have to traverse to a leaf to do an insertion.

To find the minimum number, we need to understand which elements of AVL trees we can do an insertion at, i.e. which ones have at least one null child.

In a tree of height 0, the root is such a node, so we need only visit the one node.

In an AVL tree of height 1, the root can still have a (single) null child, so again, we may be able to do an insertion visiting only one node.

On taller trees, we always start by visiting the root, then we continue the insertion process in either a tree of height $h - 1$ or a tree of height $h - 2$ (this must be the case since the overall tree is height h and the root is balanced). Let $M(h)$ be the minimum number of nodes we need to visit on an insertion into an AVL tree of height h . The previous sentence lets us write the following recurrence

$$M(h) = 1 + \min\{M(h - 1), M(h - 2)\}$$

The 1 corresponds to the root, and since we want to describe the minimum needed to visit, we should take the minimum of the two subtrees.

We could simplify this recurrence and try to unroll it, but it's easier to see the pattern if we just look at the first few values:

$$M(0) = 1, M(1) = 1, M(2) = 1 + \min\{1, 1\} = 2, M(3) = 1 + \min\{1, 2\} = 2, M(4) = 1 + \min\{2, 2\} = 3$$

In general, $M()$ increases by one every other time h increases, thus we should guess the closed-form has an $h/2$ in it. Checking against small values, we can get an exactly correct closed-form of:

$$M(h) = \lfloor h/2 \rfloor + 1$$

which is our final answer.

Note that we need a very special (as empty as possible) AVL tree to have a possible insertion visiting only $\lfloor h/2 \rfloor + 1$ nodes. In general, an AVL of height h might not have an element we could insert that visits only $\lfloor h/2 \rfloor + 1$. For example, a tree where all the leaves are at depth h is still a valid AVL tree, but any insertion would need to visit $h + 1$ nodes.

7. Ternary Heaps

Consider the following sequence of numbers:

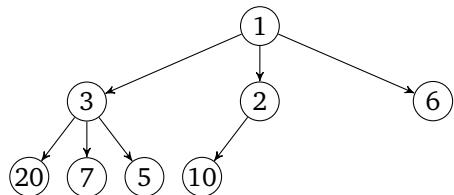
$$5, 20, 10, 6, 7, 3, 1, 2$$

- (a) Insert these numbers into a min-heap where each node has up to *three* children, instead of two.

(So, instead of inserting into a binary heap, we're inserting into a ternary heap.)

Draw out the tree representation of your completed ternary heap.

Solution:



- (b) Draw out the array representation of the above tree. In your array representation, you should start at index 0 (not index 1).

Solution:

1, 3, 2, 6, 20, 7, 5, 10

- (c) Given a node at index i , write a formula to find the index of the parent.

Solution:

$$\text{parent}(i) = \left\lfloor \frac{i-1}{3} \right\rfloor$$

- (d) Given a node at index i , write a formula to find the j -th child. Assume that $0 \leq j < 3$.

Solution:

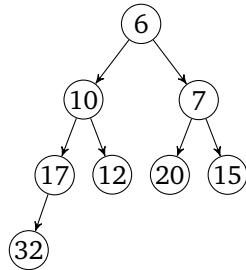
$$\text{child}(i, j) = 3i + j + 1$$

8. Heaps – More Basics

- (a) Insert the following sequence of numbers into a *min heap*:

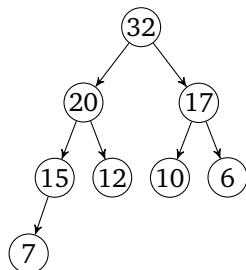
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



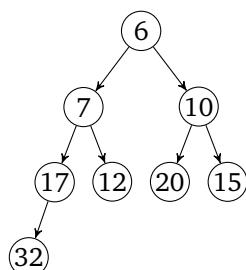
- (b) Now, insert the same values into a *max heap*.

Solution:



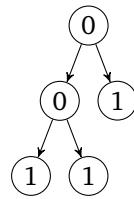
- (c) Now, insert the same values into a *min heap*, but use Floyd's buildHeap algorithm.

Solution:



- (d) Insert 1, 0, 1, 1, 0 into a *min heap*.

Solution:



- (e) Call `removeMin` on the min heap stored as the following array: [2, 5, 7, 8, 10, 9] **Solution:**

[5, 8, 7, 9, 10]

Credit: <https://medium.com/@randerson112358/min-heap-deletion-step-by-step-1e05ff9d3932>

9. Sorting and Reversing (with Heaps)

- (a) Suppose you have an array representation of a heap. Must the array be sorted? **Solution:**

No, [1, 2, 5, 4, 3] is a valid min-heap, but it isn't sorted.

- (b) Suppose you have a sorted array (in increasing order). Must it be the array representation of a valid min-heap?
Solution:

Yes! Every node appears in the array before its children, so the heap property is satisfied.

- (c) You have an array representation of a min-heap. If you reverse the array, does it become an array representation of a max-heap? **Solution:**

No. For example, [1, 2, 4, 3] is a valid min-heap, but if reversed it won't be a valid max-heap, because the maximum element won't appear first.

- (d) Describe the most efficient algorithm you can think of to convert the array representation of a min-heap into a max-heap. What is its running time? **Solution:**

You already know an algorithm – just use `buildHeap` (with `percolate` modified to work for a max-heap instead of a min-heap). The running time is $\mathcal{O}(n)$.

10. Project Prep: Contains

You just finished implementing your heap of ints when your boss tells you to add a new method called `contains`. Your solution should not, in general, examine every element in the heap (do it recursively!).

```

public class DankHeap {
    // NOTE: Data starts at index 0!
    private int[] heapArray;
    private int heapSize;

    // Other heap methods here.....

    /**
     * examine whether element k exists in the heap
     * @param int k, the element to find.
     * @return true if found, false otherwise
     */
    public boolean contains(int k) {
        // TODO!
    }
}

```

- (a) How efficient do you think you can make this method? **Solution:**

The best you can do in the worst case is $\mathcal{O}(n)$ time. If you start at the top (unlike a binary search tree) the node of priority k could be in either subtree, so you might have to check both. Even if in general we might not need to examine every node, in the worst case, this might lead us to check every node.

- (b) Write code for contains. Remember that heapArray starts at index 0! **Solution:**

```

private boolean contains(int k){
    if(k < heapArray[0]){ //if k is less than the minimum value
        return false;
    }else if (heapSize == 0){
        return false;
    }
    return containsHelper(k, 0);
}

private boolean containsHelper(int k, int index){
    if(index >= heapSize){ //if the index is larger than the heap's size
        return false;
    }
    if(heapArray[index] == k){
        return true;
    }else if(heapArray[index] < k){
        return containsHelper(k, index * 2 + 1) || containsHelper(k, index * 2 + 2);
    }else{
        return false;
    }
}

```

11. Running Times

Let's think about the best and worst case for inserting into heaps.

You have elements of priority $1, 2, \dots, n$. You're going to insert the elements into a min heap one at a time (by calling `insert` not `buildHeap`) in an order that you can control.

- (a) Give an insertion order where the total running time of all insertions is $\Theta(n)$. Briefly justify why the total time is $\Theta(n)$. **Solution:**

Insert in increasing order (i.e. $1, 2, 3, \dots, n$). For each insertion, it is the new largest element in the heap, so `percolateUp` only needs to do one comparison and no swaps. Since we only need to do those (constant) operations at each insert, we do $n \cdot \Theta(1) = \Theta(n)$ operations.

- (b) Give an insertion order where the total running time of all insertions is $\Theta(n \log n)$. **Solution:**

Insert in decreasing order. First let's show that this order requires at most $\mathcal{O}(n \log n)$ operations – we have n insertions, each takes at most $\mathcal{O}(\text{height})$ operations. The heap is always height at most $\mathcal{O}(\log n)$, so the total is $\mathcal{O}(n \log n)$.

Now let's show the number of operations is at least $\Omega(n \log n)$. For each insertion, the new element is the new smallest thing in the heap, so `percolateUp` needs to swap it to the top. For the last $n/2$ elements, the heap is height $\Omega(\log n/2) = \Omega(\log n)$, so there are $\Omega(\log n)$ operations for each of the last $n/2$ insertions. That causes $\Omega(n \log n)$ operations.

Since the number of operations is both $\mathcal{O}(n \log n)$ and $\Omega(n \log n)$ is $\Theta(n \log n)$ by definition.

Remark: it's tempting to say something like “there are n inserts and they each have $\Theta(\log n)$ operations, but that's not true. The number of operations for the first few inserts is a constant, since the tree isn't that tall yet.

12. Design Decisions

Finally! You've garnered a coveted interview for an internship at Kelp. Determine which data structure(s) are best suited for the scenarios below in terms of **typical** performance, taking into account the specific types of inputs listed in each problem. Your descriptions of the data structure(s) chosen and your algorithm should be brief but sufficiently detailed so that the runtime is unambiguous. Give a simplified, tight Θ bound for the **worstcase** runtime of your solution.

- (a) Someone has loaded all the reviews stored on Kelp into a text document of n words. Print out the number of occurrences of each unique word. You **do** need to count the time required to build your data structure, but don't worry about resizing for any of the data structures you pick **Solution:**

Data Structure: `HashMap<String, Integer>`

Usage: Iterate through the words, maintaining a mapping from word to number of times encountered.

Worst Case Runtime: $\Theta(n^2)$ (But usually much better!)

- (b) Answer the first question, but assume now that we care more about optimizing for memory usage than runtime.

Solution:

Data Structure: `ArrayMap<String, Integer>`

Usage: Same as previous question.

Worst Case Runtime: $\Theta(n^2)$

- (c) Answer the first question, but assume now that we also want the ability to print the unique words in alphabetical order. **Solution:**

Data Structure: `AVLMap<String, Integer>`

Usage: Same as previous question. Compare keys alphabetically, and do an in-order traversal for the extra feature.

Worst Case Runtime: $\Theta(n \cdot \log(n))$

- (d) Kelp has a collection of n reviews. Each review has a unique date, author name, number of stars and the text contents of the review. We want to query the number of reviews within a certain datetime range. You don't have to count the time required to construct your data structure this time. **Solution:**

Data Structure: `ArrayList<Review>`

Usage: Build an `ArrayList` sorted on the date of the review. At query time, binary search for the indices corresponding to the endpoints: the difference is the number of reviews in the range.

Worst Case Runtime: $\Theta(\log(n))$

- (e) Kelp is now trying to get into the field of visual computing. Your boss gives you n images of size 256x256, each represented as an `int[][]` array, that you'll need to store in a collection. Each image has associated with it a saturation value, which can be calculated from the pixels. Support the following three operations: `add(int[][] img)`, `getAllImgWithSaturation(int saturation)`, and `remove(int[][] img)`. When providing worst-case runtimes for each operation, don't worry about resizing for any of the data structures you pick, or the time required to construct those data structures. **Solution:**

Data Structure: `HashMap< Integer, HashSet< ImageContainer > >`

Usage: Maintain a mapping from saturation to a set of images with that saturation for easy add and get. Construct a helper class, `ImageContainer`, that computes the hashcode of the image and stores it. `remove` computes the saturation and then attempts to remove the image from the `HashSet`.

Worst Case Runtime: $\Theta(n)$

13. Challenge: Recurrences and Heaps

Suppose we have a min heap implemented as a tree, based on the following classes:

```
class HeapNode {
    HeapNode left;
    HeapNode right;
    int priority;

    // constructors and methods omitted.
}

class Heap {
    HeapNode root;
    int size;

    // constructors and methods omitted.
}
```

You just finished implementing your min heap and want to test it, so you write the following code to test whether the heap property is satisfied.

```
boolean verify(Heap h) {
    return verifyHelper(h.root);
}

boolean verifyHelper(HeapNode curr) {
    if (curr == null)
        return true;
    if (curr.left != null && curr.priority > curr.left.priority)
        return false;
    if (curr.right != null && curr.priority > curr.right.priority)
        return false;
    return verifyHelper(curr.left) && verifyHelper(curr.right);
}
```

In this problem, we will use a recurrence to analyze the worst-case running time of `verify`.

- (a) Write a recurrence to describe the worst-case running time of the function above. **Hint:** our recurrences need an input integer, use the height of the subtree rooted at `curr`.

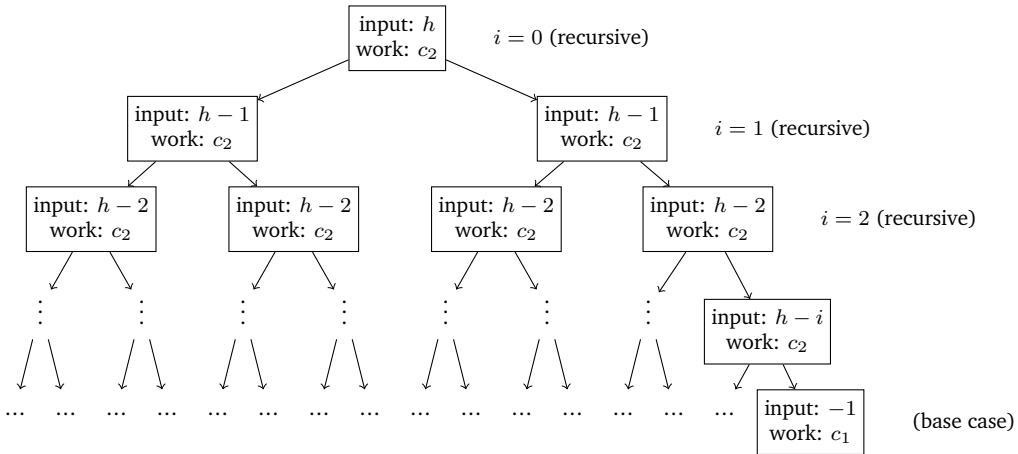
Solution:

$$T(h) = \begin{cases} c_1 & \text{if } h = -1 \\ 2T(h-1) + c_2 & \text{otherwise} \end{cases}$$

Instead of writing explicit numbers, we've written the recurrence with c_1 , c_2 to represent those constants. You will get the same big- \mathcal{O} at the end regardless of what actual numbers you plug in there. Notice that even when a node doesn't exist, we still make a recursive call and do constant work to realize we can stop recursing, so our base case really is when $h = -1$. Since we're doing worst case analysis, both of the subtrees could have $h - 1$ (i.e. the heap has all possible nodes at every level)

- (b) Find an expression (using summations but no recursion) to describe the running time using the tree method. Leave the overall height of the tree h as a variable in your expression.

Solution:



There are 2^i nodes at the i th level. Each recursive node does c_2 work, so the total work on the i th recursive level is $2^i c_2$. The last recursive level is at $i = h$ (where the input is $h - i = h - h = 0$), so the total recursive work is $\sum_{i=0}^h 2^i c_2$. Similarly, the total base case work is $2^{h+1} c_1$. Thus

$$T(n) = 2^{h+1} c_1 + \sum_{i=0}^h 2^i c_2$$

- (c) Simplify to a closed form.

Solution:

$$\begin{aligned} 2^{h+1} c_1 + \sum_{i=0}^h 2^i c_2 &= 2^{h+1} c_1 + c_2 \sum_{i=0}^h 2^i \\ &= 2^{h+1} c_1 + c_2 \frac{2^{h+1} - 1}{2 - 1} \\ &= 2^{h+1} c_1 + c_2 (2^{h+1} - 1) \\ &= (c_1 + c_2) 2^{h+1} - c_2 \end{aligned}$$

- (d) If a complete tree has height h , how many nodes could it have? Use this to determine a formula for the height of a complete tree on n nodes.

Solution:

A complete tree of height h has h completely filled rows, and one partially filled row. The number of nodes in the first h rows is: $\sum_{i=0}^{h-1} 2^i = 2^h - 1$. The final row has between 1 and 2^h nodes, so the total number of nodes is between 2^h and $2^{h+1} - 1$ nodes.

So if we take the \log_2 of the number of nodes, we'll get a number between h and $h + 1$, thus to get the height exactly, we should find:

$$\lfloor \log_2 n \rfloor$$

- (e) Use the formula from the last part to find the big- \mathcal{O} of the verify.

Solution:

Combining the last two parts, we have a formula of:

$$(c_1 + c_2)2^{\lfloor \log_2 n \rfloor + 1} - c_2$$

We'd like to eventually cancel the 2 and the exponent of $\log_2 n$. Let's make that easier by making the +1 in the exponent what it really is – multiplying the expression by 2.

$$(c_1 + c_2)2 \cdot 2^{\lfloor \log_2 n \rfloor} - c_2$$

Can we use that exponents and logs are inverses to cancel? Not if we want an exact formula; but all we care about is the big- \mathcal{O} . Getting rid of the floor will at most increase the exponent by 1, which is just multiplying that expression by 2, so we are just changing a constant factor and can make the substitution:

$$2(c_1 + c_2)2^{\lfloor \log_2 n \rfloor} - c_2 \approx 2(c_1 + c_2)2^{\log_2 n} - c_2 = 2(c_1 + c_2)n - c_2$$

The expression is now clearly $\mathcal{O}(n)$.

14. Challenge: Debugging Heaps of Problems

For this problem, we will consider a hypothetical hash table that uses linear probing and implements the `IDictionary` interface. Specifically, we will focus on analyzing and testing one potential implementation of the `remove` method.

- (a) Come up with at least 4 different test cases to test this `remove(...)` method. For each test case, describe what the expected outcome is (assuming the method is implemented correctly).

Try and construct test cases that check if the `remove(...)` method is correctly using the key's hash code. (You may assume that you can construct custom key objects that let you customize the behavior of the `equals(...)` and `hashCode()` method.)

Solution:

Some examples of test cases:

- If the dictionary contains null keys, or if we pass in a null key, everything should still work correctly.
- If we try removing a key that doesn't exist, the method should throw an exception.
- If we pass in a key with a large hash value, it should mod and stay within the array.
- If we pass in two different keys that happen to have the same hash value, the method should remove the correct key.
- If we pass in a key where we need to probe in the middle of a cluster, removing that item shouldn't disrupt lookup of anything else in that cluster.

For example, suppose the table's capacity is 10 and we pass in the integer keys 5, 15, 6, 25, 36 in that order. These keys all collide with each other, forming a primary cluster. If we delete the key 15, we should still successfully be able to look up the values corresponding to the other keys.

- (b) Now, consider the following (buggy) implementation of the `remove(...)` method. List all the bugs you can find.

```
public class LinearProbingDictionary<K, V> implements IDictionary<K, V> {
    // Field invariants:
    //
    // 1. Empty, unused slots are null
    // 2. Slots that are actually being used contain an instance of a Pair object

    private Pair<K, V>[] array;

    // ...snip...

    public V remove(K key) {
        int index = key.hashCode();

        while ((this.array[index] != null) && !this.array[index].key.equals(key)) {
            index = (index + 1) % this.array.length;
        }

        if (this.array[index] == null) {
            throw new NoSuchKeyException();
        }
        V returnValue = this.array[index].value;
        this.array[index] = null;
        return returnValue;
    }
}
```

Solution:

The bugs:

- We don't mod the key's hash code at the start
- This implementation doesn't correctly handle null keys
- If the hash table is full, the while loop will never end
- This implementation does not correctly handle the “clustering” test case described up above.

If we insert 5, 15, 6, 25, and 36 then try deleting 15, future lookups to 6, 25, and 36 will all fail.

Note: The first two bugs are, relatively speaking, trivial ones with easy fixes. The middle bug is not trivial, but we have seen many examples of how to fix this. The last bug is the most critical one and will require some thought to detect and fix.

(c) Briefly describe how you would fix these bug(s).

Solution:

- Mod the key's hash code with the array length at the start.
- Handle null keys in basically the same way we handled them in `ArrayDictionary`
- There should be a size field, with `ensureCapacity()` functionality.
- Ultimately, the problem with the “clustering” bug stems from the fact that breaking a primary cluster into two in any way will inevitably end up disrupting future lookups.

This means that simply setting the element we want to remove to null is not a viable solution. Here are many different ways we can try and fix this issue, but here are just a few different ideas with some analysis:

- One potential idea is to “shift” over all the elements on the right over one space to close the gap to try and keep the cluster together. However, this solution also fails.

Consider an internal array of capacity 10. Now, suppose we try inserting keys with the hash-codes 5, 15, 7. If we remove 15 and shift the “7” over, any future lookups to 7 will end up landing on a null node and fail.

- Rather than trying to “shift” the entire cluster over, what if we could instead just try and find a single key that could fill the gap. We can search through the cluster and try and find the very last key that, if rehashed, would end up occupying this new slot.

If no key in the cluster would rehash to the now open slot, we can conclude it's ok to leave it null.

This would potentially be expensive if the cluster is large, but avoids the issue with the previous solution.

- Another common solution would be to use lazy deletion. Rather than trying to “fill” the hole, we instead modify each `Pair` object so it contains a third field named `isDeleted`.

Now, rather than nilling that array entry, we just set that field to true and modify all of our other methods to ignore pairs that have this special flag set. When rehashing, we don't copy over these “ghost” pairs.

This helps us keep our delete method relatively efficient, since all we need to do is to toggle a flag.

However, this approach also does complicate the runtime analysis of our other methods (the load factor is no longer as straightforward, for example).

15. Challenge: Design

Imagine a database containing information about all trains leaving the Washington Union station on Monday. Each train is assigned a departure time, a destination, and a unique 8-digit train ID number.

What data structures you would use to solve each of the following scenarios. Depending on scenario, you may need to either (a) use multiple data structures or (b) modify the implementation of some data structure.

Justify your choice.

- (a) Suppose the schedule contains 200 trains with 52 destinations. You want to easily list out the trains by destination.

Solution:

One solution would be to use a dictionary where the keys are the destination, and the value is a list of corresponding chains. Any dictionary implementation would work.

Alternatively, we could modify a separate chaining hash table so it has a capacity of exactly 52 and does not perform resizing. If we then make sure each destination maps to a unique integer from 0 to 51 and hash each train based on this number, we could fill the table and simply print out the contents of each bucket.

A third solution would be to use a BST or AVL tree and have each train be compared first based on destination then based on their other attributes. Once we insert the trains into a tree, we can print out the trains sorted by destination by doing an in-order traversal.

- (b) In the question above, trains were listed by destination. Now, trains with the same destination should further be sorted by departure time.

Solution:

Regardless of which solution we modify, we would first need to ensure that the train objects are compared first by destination, and second by departure time.

We can modify our first solution by having the dictionary use a sorted set for the value, instead of a list. (The sorted set would be implemented using a BST or an AVL tree).

We can modify our second solution in a similar way by using specifically a BST or an AVL tree as the bucket type.

Our third solution actually requires no modification: if the trains are now compared first by destination and second by departure time, the AVL and BST tree's iterator will naturally print out the trains in the desired order.

- (c) A train station wants to create a digital kiosk. The kiosk should be able to efficiently and frequently complete look-ups by train ID number so visitors can purchase tickets or track the location of a train. The kiosk should also be able to list out all the train IDs in ascending order, for visitors who do not know their train ID.

Note that the database of trains is not updated often, so the removal and additions of new trains happen infrequently (aside from when first populating your chosen DS with trains).

Solution:

Here, we would use a dictionary mapping the train ID to the train object.

We would want to use either an AVL tree or a BST, since we can list out the trains in sorted order based on the ID.

Note that while the AVL tree theoretically is the better solution according to asymptotic analysis, since it's guaranteed to have a worst-case runtime of $\mathcal{O}(\log(n))$, a BST would be a reasonable option to investigate as well.

big-O analysis only cares about very large values of n , since we only have 200 trains, big-O analysis might not be the right way to analyze this problem. Even if the binary search tree ends up being degenerate, searching through a linked list of only 200 elements is realistically going to be a fast operation.

What's actually best will depend on the libraries you already have written, what hardware you actually run on, and how you want to balance code that will be sustainable if you get more trains vs. code that will be easy to understand and check for bugs right now.

Types of Rotation in Avl Tree MCQ Quiz - Objective Question with Answer for Types of Rotation in Avl Tree - Download Free PDF

 Last updated on Dec 19, 2024

Latest Types of Rotation in Avl Tree MCQ Objective Questions

Types of Rotation in Avl Tree Question 1:

How many rotations are required during the construction of an AVL tree if the following elements are to be added in the given sequence?

35, 50, 40, 25, 30, 60, 78, 20, 28

1. 2 left rotations, 2 right rotations
2. 2 left rotations, 3 right rotations
3. 3 left rotations, 2 right rotations
4. 3 left rotations, 1 right rotation

Answer (Detailed Solution Below)

Option 3 : 3 left rotations, 2 right rotations



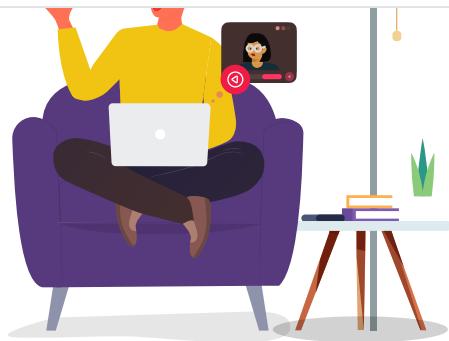
English ▾

Get Started

India's Super Teachers for
all govt. exams Under One Roof

FREE Demo Classes Available*

Enroll For Free Now



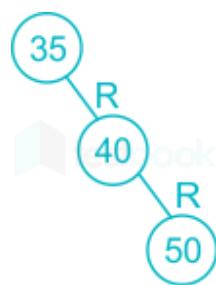
Types of Rotation in Avl Tree Question 1 Detailed Solution

The correct answer is Option 3

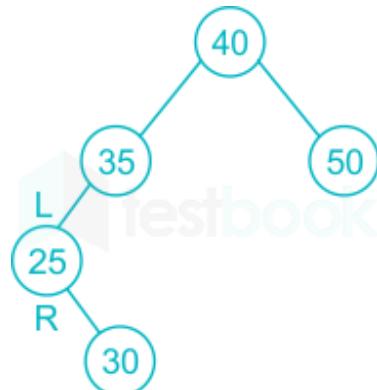
Explanation:



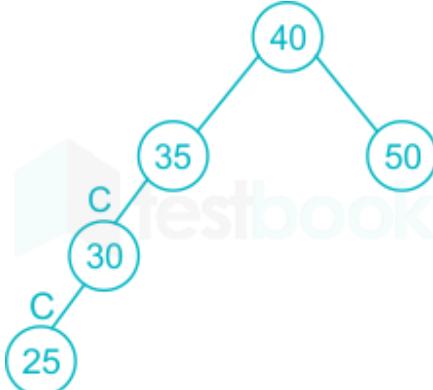
Right Rotate = 1 ⇒



Left Rotate = 1 ⇒

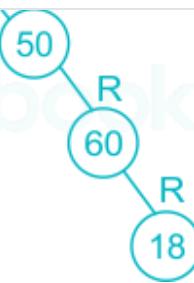
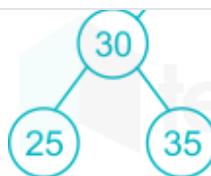
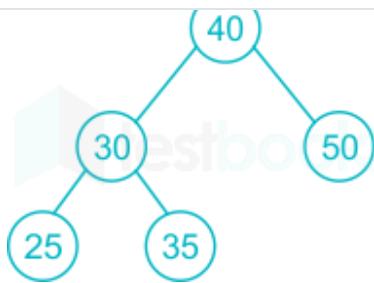


Right Rotate = 1 ⇒



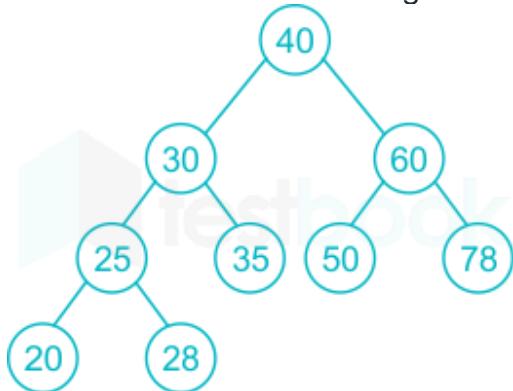


English ▾

[Get Started](#)

Right Rotate = 1 ⇒

Left Rotate = 1 ⇒



FREE

India's #1 Learning Platform

Trusted by 6.9 Crore+ Students

[Start Complete Exam Preparation](#)Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes[Get Started for Free](#)

Types of Rotation in Avl Tree Question 2:

How many rotation are required during the following operation on the AVL tree, which can be constructed from the post order traversal given as

7 28 37 27 52 68 60 56 40

Operation:

- i) add 70, 32, 15
- ii) delete 27, 52

2. 2 left rotation, 2 right rotations

3. 3 left rotation, 2 right rotations

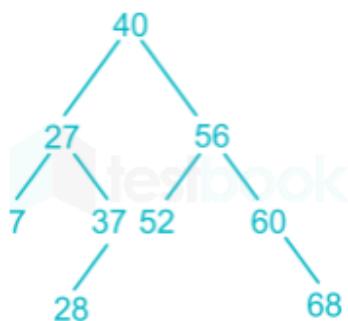
4. 3 left rotation, 1 right rotation

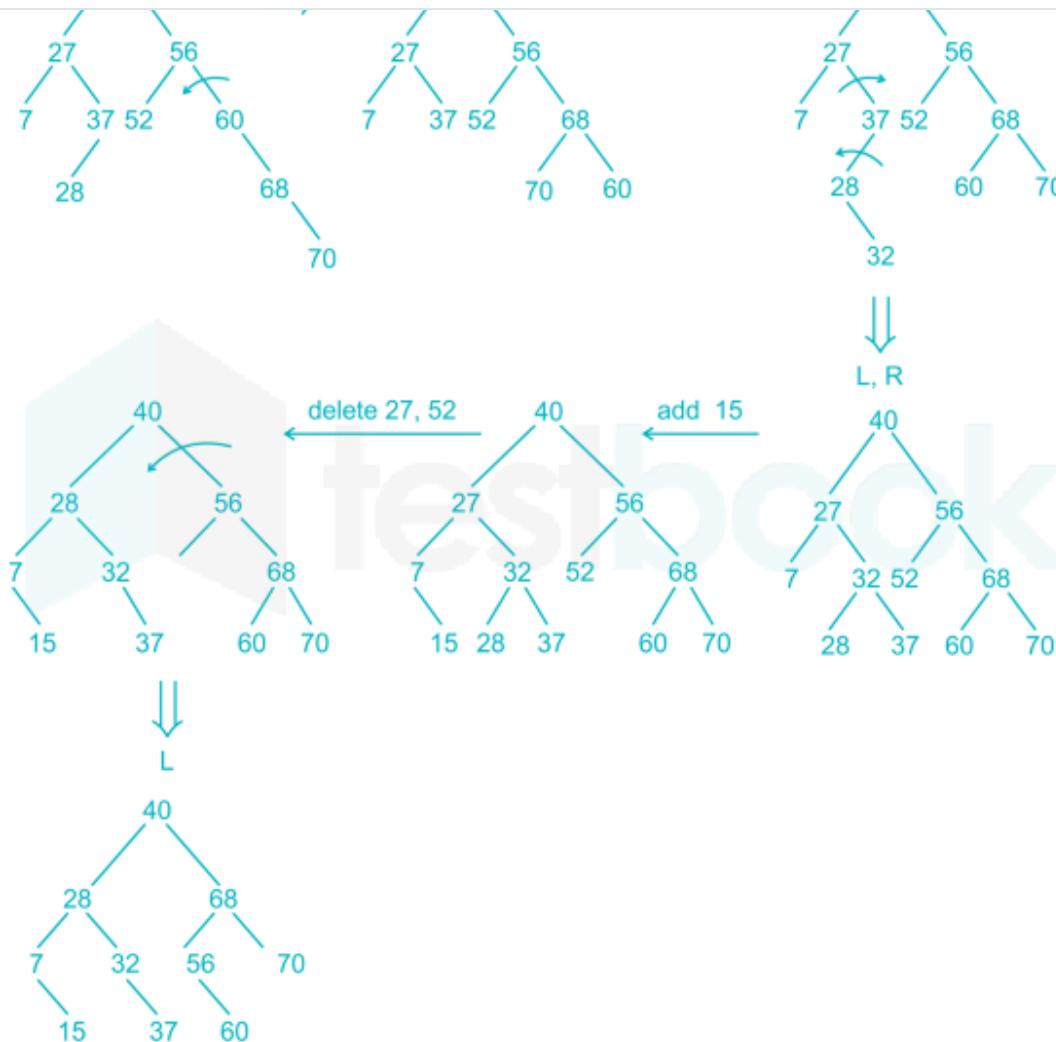
Answer (Detailed Solution Below)

Option 4 : 3 left rotation, 1 right rotation

Types of Rotation in Avl Tree Question 2 Detailed Solution

The original tree is,





FREE

India's #1 Learning Platform

 Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

 Daily Live MasterClasses

 Practice Question Bank

 Mock Tests & Quizzes

Get Started for Free

 Download on the App Store

 GET IT ON Google Play

Types of Rotation in Avl Tree Question 3:

If an in order traversal of a tree is EACKFHDGB, then its preorder traversal can be

2. F A E K C D H G B

3. E A F K H D C B G

4. F E A K D C H B G

Answer (Detailed Solution Below)

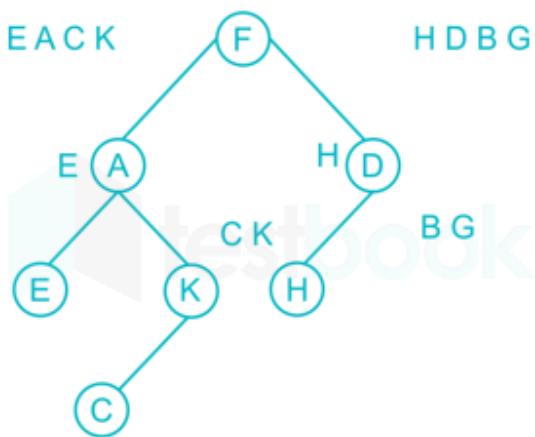
Option 2 : F A E K C D H G B

Types of Rotation in Avl Tree Question 3 Detailed Solution

In order traversal : E A C K F H D B G

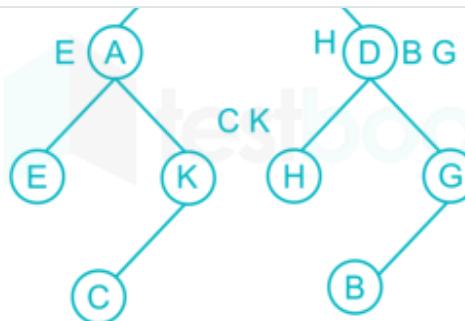
with the given in order traversal, it is not possible to compute a unique preorder traversals. But we can invalidate some of the preorder traversals that is possible from the given in order traversal

a) F A E K C D B H G



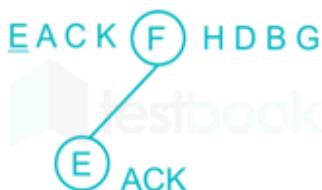
In this case, according to in order, H should come before 'B' and at left of 'D'. But according to preorder 'B' will come Left of 'D'. Hence in order and preorder contradicts for constructing the tree properly so this option is invalid

b) F A E K C D H G B



This preorder and in order traversal construct the tree properly

- c) E A F K H D C B G



Here 'c' comes after 'HD'. So this option is invalid

- d) F E A K D C H B G



Here 'D' (right child) comes before 'c' (left child)

so this is also invalid

FREE

India's #1 Learning Platform

Start Complete Exam Preparation

 Trusted by 6.9 Crore+ Students
Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes

Get Started for Free

 Download on the
App Store

 GET IT ON
Google Play

Top Types of Rotation in Avl Tree MCQ Objective Questions

Types of Rotation in Avl Tree Question 4

Download Solution PDF

55, 50, 40, 25, 30, 00, 70, 20, 20

1. 2 left rotations, 2 right rotations

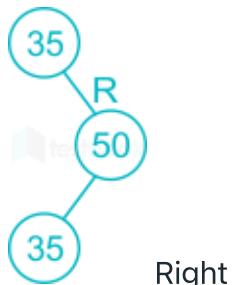
2. 2 left rotations, 3 right rotations

3. 3 left rotations, 2 right rotations

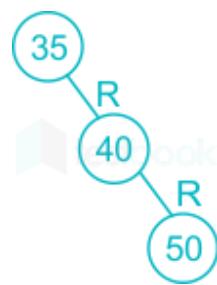
4. 3 left rotations, 1 right rotation

Answer (Detailed Solution Below)

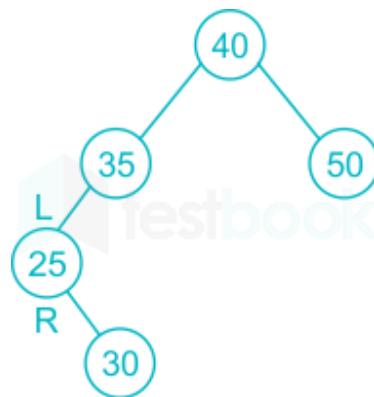
Option 3 : 3 left rotations, 2 right rotations

Types of Rotation in Avl Tree Question 4 Detailed Solution[Download Solution PDF](#)**The correct answer is Option 3****Explanation:**

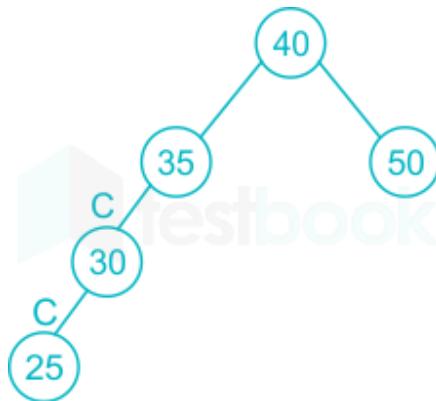
Right Rotate = 1 ⇒



Left Rotate = 1 ⇒

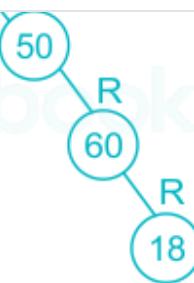
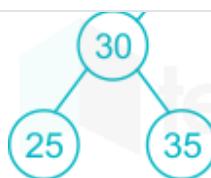
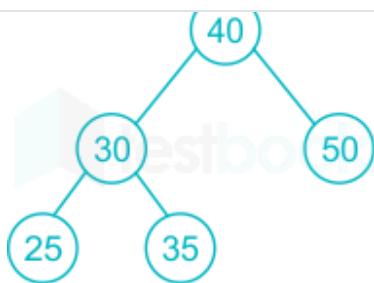


Right Rotate = 1 ⇒

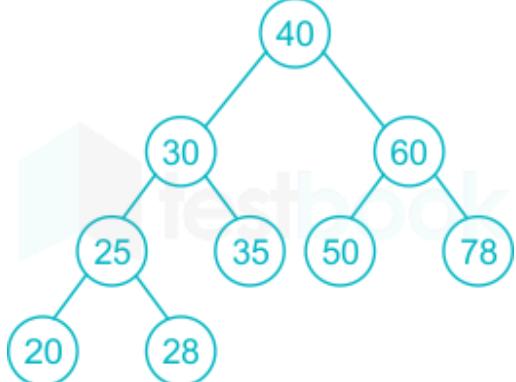




English ▾

[Get Started](#)

Right Rotate = 1 ⇒



Left Rotate = 1 ⇒

[Download Solution PDF](#)[Share on Whatsapp](#)

FREE

India's #1 Learning Platform

Start Complete Exam Preparation

Trusted by 6.9 Crore+ Students



Daily Live MasterClasses



Practice Question Bank



Mock Tests & Quizzes

[Get Started for Free](#)

Types of Rotation in Avl Tree Question 5:

If an in order traversal of a tree is EACKFHDGB, then its preorder traversal can be

1. F A E K C D B H G
2. F A E K C D H G B
3. E A F K H D C B G

Answer (Detailed Solution Below)

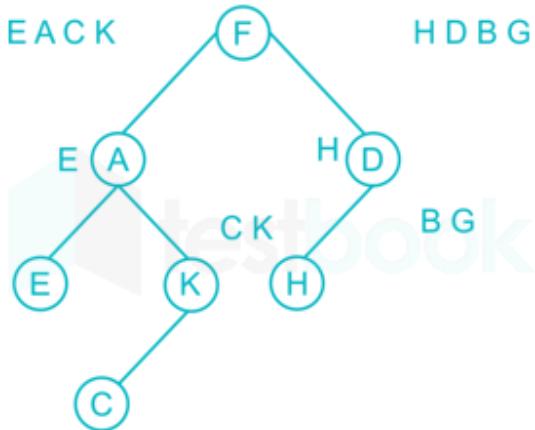
Option 2 : F A E K C D H G B

Types of Rotation in Avl Tree Question 5 Detailed Solution

In order traversal : E A C K F H D B G

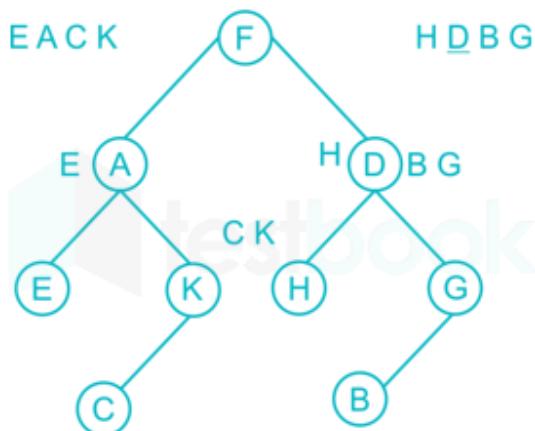
with the given in order traversal, it is not possible to compute a unique preorder traversals. But we can invalidate some of the preorder traversals that is possible from the given in order traversal

a) F A E K C D B H G



In this case, according to in order, H should come before 'B' and at left of 'D'. But according to preorder 'B' will come Left of 'D'. Hence in order and preorder contradicts for constructing the tree properly so this option is invalid

b) F A E K C D H G B



This preorder and in order traversal construct the tree properly



Here 'c' comes after 'HD'. So this option is invalid

d) F E A K D C H B G



Here 'D' (right child) comes before 'c' (left child)

so this is also invalid

FREE

India's #1 Learning Platform

 Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation



Daily Live
MasterClasses



Practice Question
Bank



Mock Tests &
Quizzes

Get Started for Free

 Download on the
App Store

 GET IT ON
Google Play

Types of Rotation in Avl Tree Question 6:

How many rotation are required during the following operation on the AVL tree, which can be constructed from the post order traversal given as

7 28 37 27 52 68 60 56 40

Operation:

- i) add 70, 32, 15
- ii) delete 27, 52

1. 2 left rotation, 1 right rotation

2. 2 left rotation, 2 right rotations

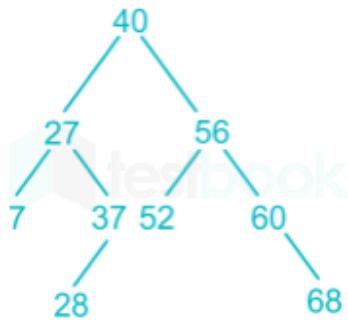
4. 3 left rotation, 1 right rotation

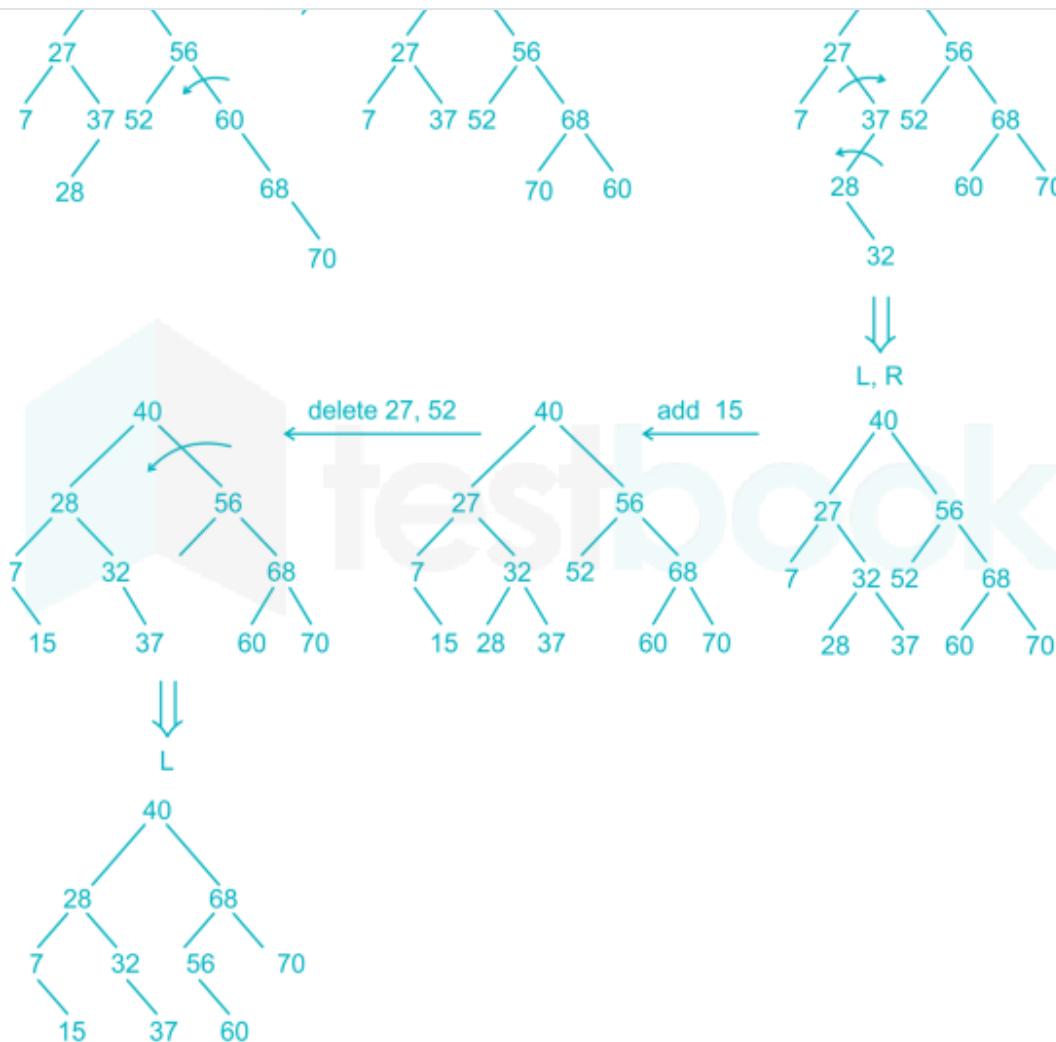
Answer (Detailed Solution Below)

Option 4 : 3 left rotation, 1 right rotation

Types of Rotation in Avl Tree Question 6 Detailed Solution

The original tree is,





FREE

India's #1 Learning Platform

Start Complete Exam Preparation


 Daily Live MasterClasses

 Practice Question Bank

 Mock Tests & Quizzes

Get Started for Free




Types of Rotation in Avl Tree Question 7:

How many rotations are required during the construction of an AVL tree if the following elements are to be added in the given sequence?

 1. 2 left rotations, 2 right rotations

2. 2 left rotations, 3 right rotations

3. 3 left rotations, 2 right rotations

4. 3 left rotations, 1 right rotation

Answer (Detailed Solution Below)

Option 3 : 3 left rotations, 2 right rotations

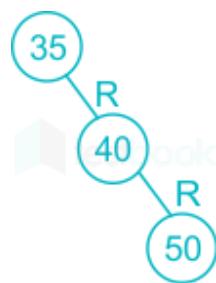
Types of Rotation in Avl Tree Question 7 Detailed Solution

The correct answer is Option 3

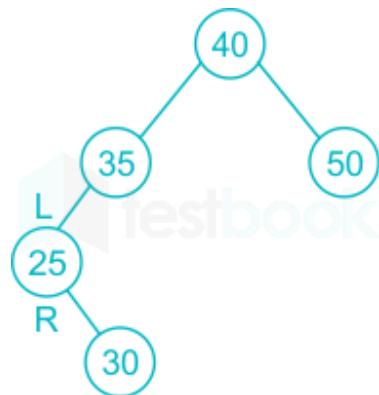
Explanation:



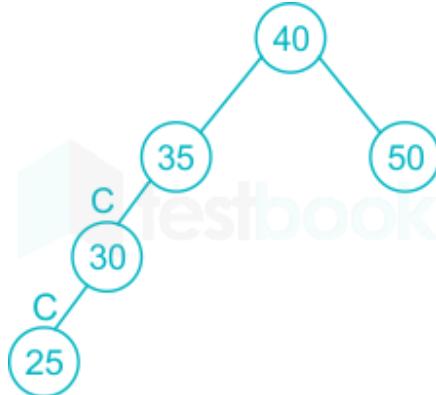
Right Rotate = 1 ⇒

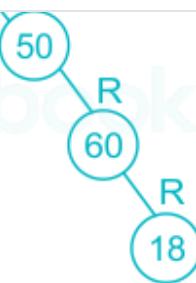
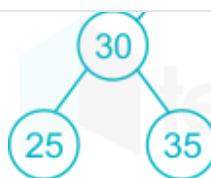
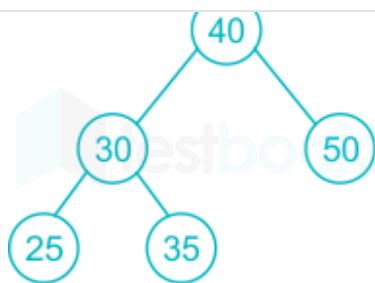


Left Rotate = 1 ⇒

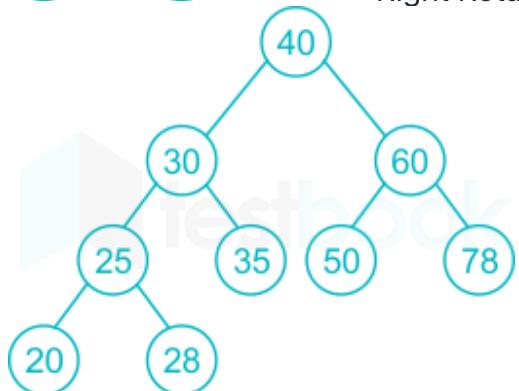


Right Rotate = 1 ⇒





Right Rotate = 1 ⇒



Left Rotate = 1 ⇒

FREE

India's #1 Learning Platform

Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes

Get Started for Free

Download on the
App StoreGET IT ON
Google Play

Types of Rotation in Avl Tree Question 8:

What order should we insert the elements {1,2,...,7} into an empty AVL tree so that we don't have to perform any rotations on it?

1. {4,2,1,6,3,5,7}
2. {4,2,6,1,3,5,7}
3. {4,2,1,6,3,7,5}



English ▾

Get Started

Answer (Detailed Solution Below)

Option 2 : {4,2,6,1,3,5,7}

Types of Rotation in Avl Tree Question 8 Detailed Solution

We should insert in the order {4,2,6,1,3,5,7} to make an AVL tree. The ordering of {2,6} and the ordering of {1,3,5,7} do not matter. We can see the resulting binary search tree is perfectly balanced therefore an AVL tree.

FREE

India's #1 Learning Platform

 Trusted by 6.9 Crore+ Students

Start Complete Exam Preparation

Daily Live
MasterClassesPractice Question
BankMock Tests &
Quizzes[Get Started for Free](#)



Understanding Memory Allocation in Linked Lists: How Nodes and Addresses Connect

Table of Contents

- Basics of Memory Allocation
 - The Heap: A Sequence of Memory Slots
 - Storing and Retrieving Objects
- Memory Allocation in Linked Lists
 - Node Creation
 - Linking Nodes
- Manual vs Automatic Memory management

Linked lists are fundamental data structures in computer science, offering a flexible and dynamic way to organize data. Unlike arrays, which store elements in contiguous memory locations, linked lists consist of nodes scattered throughout memory, connected by pointers/references. This article covers various aspects of memory allocation in linked lists, exploring how nodes are created, stored, and linked together through memory addresses. Visual representation of how linked list nodes are stored in memory is also provided.

Prerequisite: [Basics of linked list data structure](#).

Basics of Memory Allocation

Memory allocation in computer science deals with how an operating system or program manages and organizes its available memory resources. In this section we will cover the basics of memory allocation, focusing especially on the heap memory and how objects are stored and managed.

The Heap: A Sequence of Memory Slots

Think of the heap as a large, contiguous area of computer memory. We can visualize it as a long sequence of slots, each with its own unique address. Using the address of each memory slot, we can access the underlying data stored at the corresponding slot. These addresses are typically represented as numbers, with each subsequent slot having an address one higher than the previous one. For example below is a section of a heap address which slot addresses starting from 100:

M	E	M				S	P	A	C	E
100	101	102	103	104	105	106	107	108	109	110

In the image shown above, the letter M is stored at memory address 100, letter E is stored at memory address 101 and so on. For objects/data which require more than one slot to be stored in memory, the system allocates a series of consecutive slots based on the object's size. The address of the first slot is then returned as a reference to that object.

Storing and Retrieving Objects

Let's say we want to store an object that requires 3 memory slots. The system will scan the memory slots and look for 3 consecutive empty slots and then allocate the slots for the object to be stored. For example, the system might allocate slots 103, 104, and 105 for this object. The program would then receive the address 103 as a reference to the object's location in memory.

M	E	M	O	B	J	S	P	A	C	E
100	101	102	103	104	105	106	107	108	109	110

When the program needs to access or modify the object, it uses this address (103) as the starting point. Since the system knows the object's size, it can read all the necessary slots (103, 104, and 105) to retrieve or update the object's data.

Memory Allocation in Linked Lists

Node Creation

When creating a new node in a linked list, memory is allocated dynamically using heap memory. This allocation process involves reserving a continuous block of memory in heap that is large enough to store two key components:

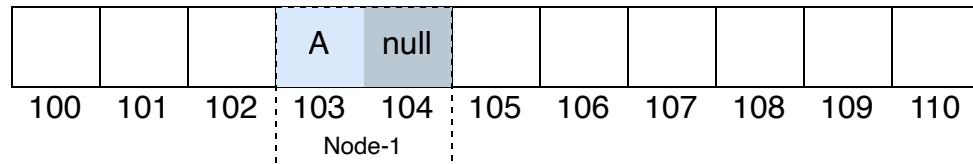
1. The data element of the node

2. A pointer to the next node

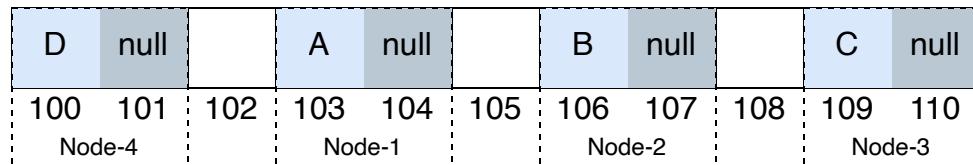
For example, if we're creating a linked list of alphabets/characters, each node would typically require memory for:

- An ASCII character value (usually 4 bytes)
- A pointer to the next node (usually 4 or 8 bytes, depending on the system architecture).
Let's assume that this is a 32 bit system and 4 bits are required for storing pointer/reference address to the next node).

When a new node is created, the system allocates this continuous block of memory (in this case, 8 bytes or two slots of memory) on the heap. Initially, the address of the next node is set to `null`. In memory terms, `null` is represented by an invalid memory address (often `0x0` or `0`), indicating that the current node is not pointing to any other node yet. Below is how the newly created node looks like in memory:



Let's now allocate memory for 3 more nodes for storing characters `B`, `C` and `D`. Below is how all the linked list nodes are stored/represented in memory:



Notice that even though we want the nodes `A`, `B`, `C`, `D` in that sequence in the linked list, the created nodes need not be stored in the same sequence in memory. It can be in any order in memory. The actual order in the linked list depends on how we link these nodes.

Below is how to reserve memory and initialize linked list nodes in various programming languages:

python c cpp javascript java

```
# Node class to store data and link to the next node
class Node:
```

```

def __init__(self, data):
    # Initialize the node with given data
    self.data = data
    # Set the next pointer to None initially
    self.next = None

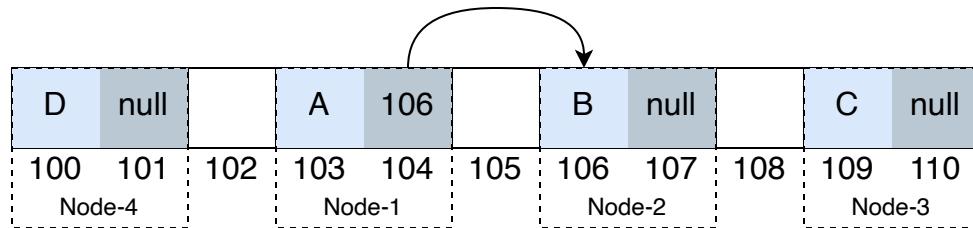
# We need to create the nodes first before linking them
node1 = Node("A")
node2 = Node("B")
node3 = Node("C")
node4 = Node("D")

```

Linking Nodes

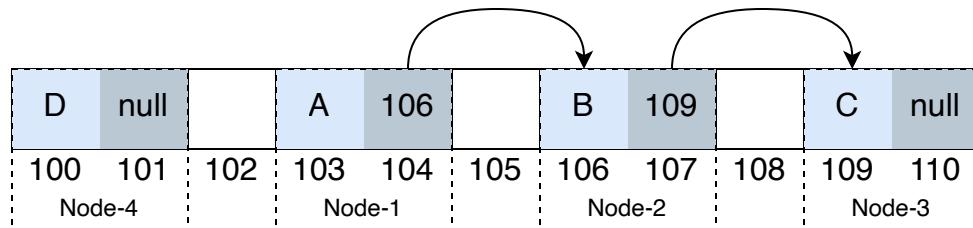
In the previous step, we looked at how we can allocate and fill memory required for each node. Now we can start connecting them to form a linked list. This process involves setting the “next” pointer of each node to the memory address of the subsequent node in the list. Let's first connect node A's next pointer to node B's address by using the expression

`node1.next = address of node2 :`

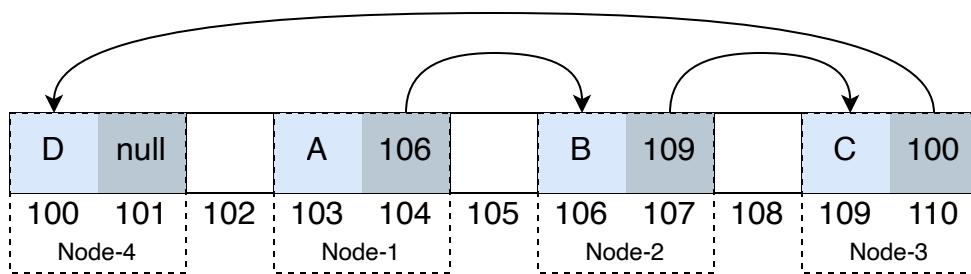


Now let's connect the node B's next pointer to node C's address using the expression

`node2.next = address of node3 :`



Now let's connect the node C's next pointer to node D's address:



Now, address of A (103) is stored as the head node of the linked list. Using this address we can access all the linked list nodes and perform all linked list operations. In the memory layout, although the sequence of linked list nodes is D, A, B, C , since we start from head node address (103), the sequence depends on how each node is linked, irrespective of how those nodes are arranged in the memory slots.

Below is how we programmatically link linked list nodes:

```

node1.next = node2 # A->B
node2.next = node3 # B->C
node3.next = node4 # C->A

# node1 is the head node in the linked list
head = node1 # Node A

```

[Click here for a more detailed article on how to create nodes and connect them together to form a linked list.](#)

Manual vs Automatic Memory management

When we allocate memory for linked list nodes on the heap, we should also make sure that the allocated memory is freed when the nodes are deleted or the linked list is no longer required. In some programming languages, the freeing of allocated memory needs to be manually done. In some languages with garbage collection, the freeing of allocated memory is done automatically.

For example, languages like C and C++ do not have garbage collection and memory needs to be managed manually. For cases like these, programmers are responsible for allocating and deallocating memory. Below is an example on how we do this in C language:

- **Allocation:** `Node* node = malloc(sizeof(Node));` - This allocates memory for storing a linked list node returns the address of allocated memory.

- **Deallocation:** `free(node);` - This releases the previously allocated memory for the node. This needs to be performed when the linked list node is no longer necessary.

Most of the bugs/issues with linked lists occur due to memory which is not deallocated properly for all the nodes which are removed from the linked list. This causes memory leaks and the program would crash over time due to high memory consumption.

Some programming languages, like Java or Python, use automatic memory management systems (often called garbage collectors). These systems track which objects are still in use by the program. So when an linked list node is no longer referenced by any other nodes/variables, the system automatically frees up the memory it occupied. This helps prevent memory leaks and makes programming easier.

◀ How to Create and Connect Nodes in a
Linked List in Various Programming
Languages

Different Types of Linked List Data
Structure ►

Home / Computer science / Data structures /

Arrays ▼

Linked Lists ▲

Overview

Basic Implementation

Memory Management

Different types

Applications

Basic Operations ▼

Basic Problems ▼

Advanced Problems ▼

Advantages and Disadvantages

Linked Lists vs Arrays

Singly Linked List Implementations ▼

Hash Tables ▼

Stacks ▼

Understanding the Difference Between In-Memory and Disk-Based Indexes in SQL

SQL

In the realm of SQL databases, the role of indexes is crucial for optimizing query performance. There are various types of indexing mechanisms, two of the most common being in-memory indexes and disk-based indexes. Understanding the differences between these types of indexes can help database administrators, software engineers, and data scientists make better decisions when it comes to database design and query optimization.



Learn
more

Replay

TOC

1 | The Basics of Indexing in SQL

1-1 | How Indexes Work

1-2 | Types of Indexes

\$990	\$2,750	\$7,040	\$6,040
-------	---------	---------	---------

- | | | | |
|---|--|--|--|
| 3-1 Advantages of Disk-Based Indexes | | | |
| 3-2 Disadvantages of Disk-Based Indexes | | | |
| 4 Comparing In-Memory and Disk-Based Indexes | | | |
| 5 Choosing Between In-Memory and Disk-Based Indexes | | | |
| 5-1 Considerations for Choice | | | |
| 5-2 Best Practices | | | |
| 6 Summary | | | |

The Basics of Indexing in SQL

Indexing is the process that improves the speed of data retrieval operations on a database table at the cost of additional storage space and decreased performance on insert and update operations. When a query requests data, the database engine looks through the index instead of scanning the entire table, thereby improving performance.

How Indexes Work

Indexes work by storing a subset of the table's data in a data structure. This data structure can be quickly traversed to locate the specific rows of the table corresponding to a query. When a table has an index, the SQL database engine will use the index to find the data faster, rather than scanning all rows.

the type of queries they are optimized for. The focus here will be on the in-memory and disk-based indexes.

In-Memory Indexes

In-memory indexes are stored in the computer's main memory (RAM). These types of indexes are extremely fast but are volatile, meaning they are lost if the system shuts down.

Advantages of In-Memory Indexes

- High-speed data retrieval
- Low latency
- Optimized for read-heavy operations

Disadvantages of In-Memory Indexes

- Increased memory usage
- Volatile
- Costly due to high RAM prices

	\$990	\$2,750	\$1,290	\$1,250
=	SHOP NOW	SHOP NOW	SHOP NOW	SHOP NOW

Disk-Based Indexes

Disk-based indexes are stored on a hard drive. These types of indexes are persistent and survive system shutdowns but are slower compared to in-memory indexes.

Advantages of Disk-Based Indexes

- Persistent
- Lower hardware costs
- Suitable for large datasets

Disadvantages of Disk-Based Indexes

- Slower data retrieval
- Higher latency
- May require more complex maintenance

Feature	In-Memory Indexes	Disk-Based Indexes
Speed	Fast	Slower
Persistence	Volatile	Persistent
Cost	High	Low
Data Size	Smaller datasets	Larger datasets

Table1: Comparing Features of In-Memory and Disk-Based Indexes

	\$5,480	\$5,280
=		

Choosing Between In-Memory and Disk-Based Indexes

The choice between in-memory and disk-based indexes depends on various factors such as the size of the dataset, the type of queries, available hardware, and specific requirements of your application.

Considerations for Choice

Best Practices

- Use in-memory indexes for OLAP databases where read performance is critical.
- Use disk-based indexes for OLTP databases with large datasets and more insert/update operations.

Summary

Understanding the characteristics of in-memory and disk-based indexes can have a significant impact on the performance and cost-efficiency of your SQL databases. While in-memory indexes offer speed, they are volatile and may be expensive. Disk-based indexes, on the other hand, provide persistence and are cost-effective but come with their own set of drawbacks like slower speed. By considering the specific requirements and constraints of your system, you can make a more informed choice between the two.