Dsa Suggestion

1. (i) What is Dictionary?
(ii) How to implement dictionaries?
2. (i) Define Hashing?
(ii) Explain Review of Hashing and Hash Function?
3. (i) What is chaining?
(ii) Write about separate chaining and open addressing?
4. (i) Explain Collision Resolution Techniques in Hashing?
(ii) What ADT?
5. (i) Explain Dictionary Abstract Data Type?
(ii) Mention some features of ADT.
6. (i) Write about linear probing and quadratic probing?
(ii) what is chaining?
(iii) Discuss the concept of Rehashing?
7. (i) What is Double Hashing technique?
(ii) What is Data structure? What are the applications of data structure?
(iii) Explain where hashing is used in real time with an example?
8. (i) Explain Recent Trends in Hashing?
(ii) Write about separate chaining and open addressing?
(iii) Convert the infix (a+b)*(c+d)/f into postfix & prefix expression.
9. Differentiate between hashing and extendible hashing?
10. (i) What do you mean by linear open addressing?
(ii) What are the differences between static hashing and dynamic hashing?
11. (i) What is skip list?
(ii) Write about open addressing technique?
(iii) Explain search and update operations on skip lists?
12. (i) Explain Skip List. Why it is called as a Randomized Data structure.
(ii) What are the advantages of Skip List?
13. (i) Explain the Operations Insertion and Searching with a Skip List.
(ii) What are the disadvantages of Skip List?
14. (i) Explain Skip List. Why it is called as a Randomized Data structure.
(ii) Explain the Operations Insertion, Deletion and Searching with a Skip List.
15. (i) What are 2-3 trees how it works with data structures discuss with an example?
(ii) what is AVL tree?
(iii) Explain the properties of Red Black Trees with an example.
16. (i) Explain Splay- trees with neat diagram?
17. (i) Explain Red black trees with an example?
(ii) construct the Red Black tree with following elements:
Insert 2, 1, 4, 5, 9, 3, 6, 7
18. (i) Explain the issues with AVL Tree and recommend how Red Black trees can be a solution for it.
(ii) What is a Binary Tree? Explain it. the preorder, in order and post order Traversals?
19. (i) Explain about the Binary Search Tree? What are the rules to create a BST? Give an

example
20. (i) Write the code for Binary Search Tree Insertion.
(ii) Explain B- trees and its operations?
21. (i) start with an empty height balanced binary search tree (AVL) and insert the elements with following keys in the given order: 3,2,1,4, 5,6,7,15,16. At each step label all nodes with their balance factors and identify the rotation types.
(ii) What is the difference between AVL tree and Splay tree?
(iii) What is computational geometry?
(iv) Explain One Dimensional Range Searching with an example?
22. (i) Explain various computational geometry methods for efficiently solving the new evolving problem?
(ii) Differentiate between Compressed Tries, Suffix Tries.
23. (i) How Two Dimensional Range Searching done in computational geometry explain with an example
24. (i) Write a detailed note on the Huffman Coding Algorithm?
(ii) How to Apply Dynamic Programming to the LCS? Justify it with example.
25. (i) Discuss the working of Brute force pattern matching?
(ii) How Compressed Tries work? Explain its operations?
(iii) Explain Standard Tries with an example?
26. (i) Describe The Knuth-Morris- Pattern Algorithm?
(ii) Discuss the function Suffix Tries with an example?
27. (i) Write about The Boyer- Moore Algorithm?
(ii) Define Tires with example . Explain k-D Trees with an example?
28. (i) Describe Quad trees and its functions?
(ii) How to construct a Priority Search tree? Explain with neat diagram
(iii) What is Priority Range Trees discuss with an example?
29. (i) What is computational geometry?
(ii) What is Priority Range Trees discuss with an example?
30. (i) Explain how to Search a Priority Search Tree works and its operations?

1.

(i) What is a Dictionary in c?

In C programming, a dictionary typically refers to a data structure known as an associative array or map. However, C itself does not have a built-in dictionary data type like some higher-level programming languages such as Python or Java. In C, you can implement a dictionary-like structure using arrays or structures.
One common way to implement a dictionary in C is by using arrays of structures. Each structure in the array represents a key-value pair.

(ii) How to implement dictionaries?

```c
#include <stdio.h>
#include <string.h>

// Define a structure for key-value pairs
struct KeyValuePair {
    char key[50];
    int value;
};

int main() {
    // Create an array of key-value pairs
    struct KeyValuePair dictionary[100];

    // Add some entries to the dictionary
    strcpy(dictionary[0].key, "apple");
    dictionary[0].value = 5;

    strcpy(dictionary[1].key, "banana");
    dictionary[1].value = 8;

    // Access values using keys
    printf("Value for key 'apple': %d\n", dictionary[0].value);
    printf("Value for key 'banana': %d\n", dictionary[1].value);

    return 0;
}
```

Output:

```
Value for key 'apple': 5
Value for key 'banana': 8
```

2.

(i) Define Hashing?

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

In the context of Data Structures and Algorithms (DSA), hashing refers to the technique of using a hash function to map data (keys) to fixed-size hash values. This process allows for efficient data retrieval, storage, and manipulation by associating each key with a unique or nearly unique hash code. Hashing is commonly used in data structures like hash tables, where keys are hashed to determine their storage locations, facilitating fast lookup operations. It is a fundamental concept in DSA for optimizing search and retrieval algorithms.

Hashing in data structures is primarily used for:

1. **Hash Tables:** Efficient data retrieval based on keys.
2. **Collision Resolution:** Handling conflicts when two keys hash to the same index.
3. **Distributed Databases:** Evenly distributing data across nodes.
4. **Cuckoo Hashing:** Resolving collisions using multiple hash functions.
5. **Bloom Filters:** Space-efficient set membership tests.
6. **String Matching:** Efficient algorithms like Rabin-Karp for substring search.

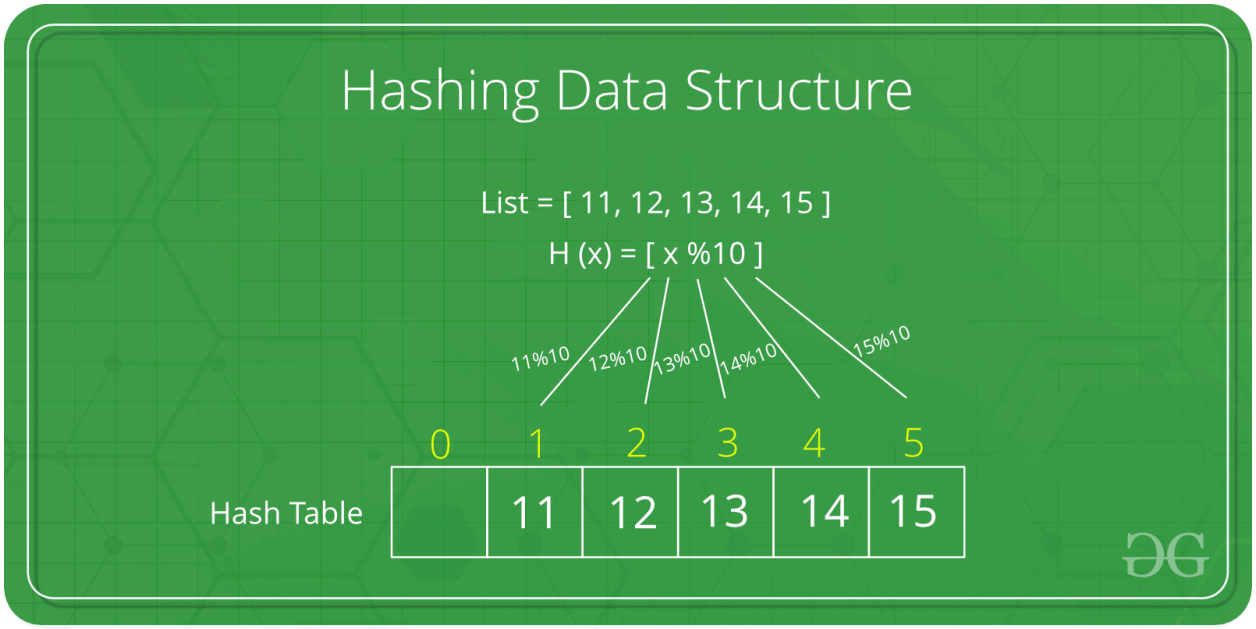
(ii) Explain Review of Hashing and Hash Function?

A review of hashing and hash functions involves understanding their key concepts and applications:

Hashing:
- **Definition**: Hashing is a process of converting input data (or keys) into a fixed-size string of characters using a hash function.
- **Purpose**: It provides a quick and efficient method for data retrieval, integrity verification, and various algorithmic optimizations.
- **Applications**: Hashing is extensively used in data structures like hash tables, distributed databases, and cryptographic protocols.

Hash Function:
- **Definition**: A hash function takes an input (or key) and produces a fixed-size hash value, typically a string of characters.
- **Properties:** Ideally, a good hash function should be deterministic, efficient to compute, and produce a unique hash for different inputs. It should also exhibit the avalanche effect, where a small change in the input results in a significantly different hash value.
- **Use Cases:** Hash functions are employed in password storage, digital signatures, data integrity checks, and various algorithmic applications.

Let a hash function H(x) maps the value x at the index x%10 in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



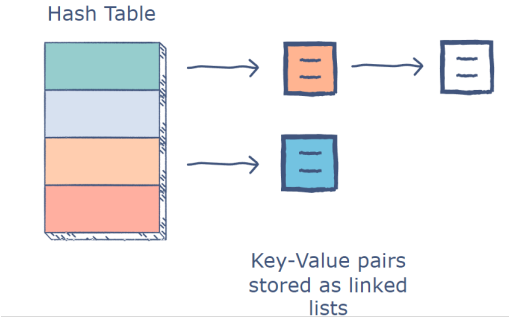
3.

(i) What is chaining in dsa?

Chaining is a collision resolution strategy in hash tables where, upon a collision (two keys hash to the same index), instead of overwriting existing data, multiple key-value pairs are stored at the same index using a linked list. Each index in the hash table contains a linked list of key-value pairs, enabling efficient storage and retrieval, especially when collisions are infrequent.

The chaining technique

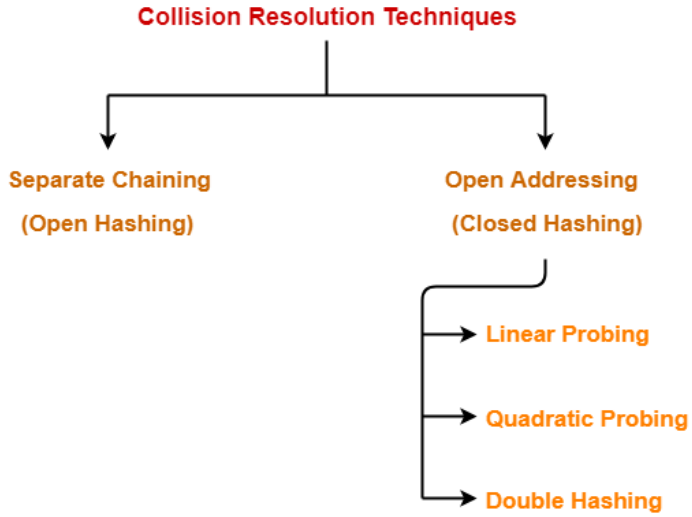
In the chaining approach, the hash table is an array of linked lists i.e., each index has its own linked list.
All key-value pairs mapping to the same index will be stored in the linked list of that index.



The benefits of chaining

- Through chaining, insertion in a hash table always occurs in *O(1)* since linked lists allow insertion in constant time.
- Theoretically, a chained hash table can grow indefinitely as long as there is enough space.
- A hash table which uses chaining will never need to be resized.

(ii) Write about separate chaining and open addressing?



Separate Chaining:

Definition:
Separate Chaining is a collision resolution technique in hash tables where each hash table bucket maintains a linked list of elements that hash to the same index.

Advantages:

1. Ease of Implementation: Separate chaining is relatively easy to implement compared to some open addressing techniques.
2. Dynamic Size: The size of the hash table can be dynamic, and it can handle a large number of elements efficiently.
3. No Limit on Load Factor: It handles a higher load factor well without significantly affecting performance.
4. No Probing Overhead: Since elements with the same hash value are stored in a linked list, there is no need for probing or searching within the table.

Disadvantages:
1. Worst-Case Performance: In the worst case, when all keys hash to the same index, the performance degrades to O(n), where n is the number of elements.
2. Memory Overhead: Additional memory is required for maintaining pointers in linked lists, which can increase memory consumption.

Open Addressing:

Definition:
Open Addressing is another collision resolution technique in hash tables where, if a collision occurs, the system looks for the next available slot within the hash table to place the collided element.

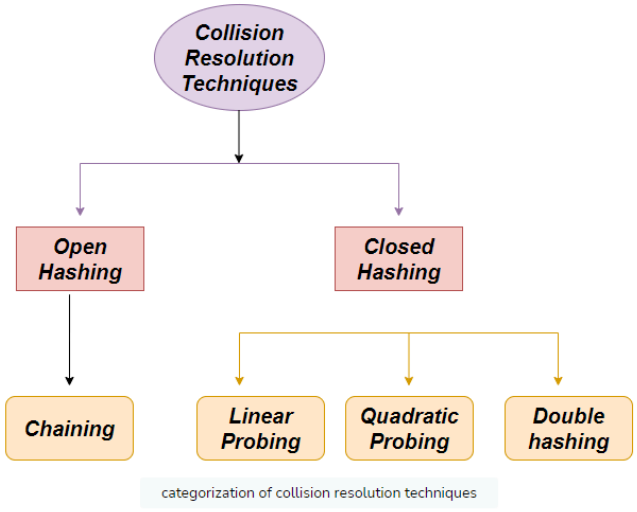
Advantages:
1. Memory Efficiency: It often requires less memory than separate chaining since no additional memory is needed for pointers.
2. Cache Performance: Better cache performance compared to separate chaining, as elements are stored contiguously in memory.
3. Space Efficiency: In situations where the load factor is low, open addressing can be more space-efficient.
4. Predictable Memory Access: Memory access is more predictable due to contiguous storage of elements.

Disadvantages:
1. Complex Deletion: Deletion of elements can be more complex compared to separate chaining, especially in certain open addressing variants.
2. Cluster Formation: Clustering can occur, leading to more collisions, which may affect performance.
3. Limited Load Factor: Performance can degrade if the load factor is high, leading to more collisions and probing.

4.

(i) Explain Collision Resolution Techniques in Hashing?

Collision resolution techniques are methods used to handle situations where two different keys hash to the same index in a hash table. Collisions can occur due to the finite size of the hash table compared to the potentially infinite set of keys. Several techniques are employed to resolve collisions and maintain the integrity of the hash table. Here are some common collision resolution techniques:



categorization of collision resolution techniques

1. Open Hashing:
- In open hashing, also known as open addressing or closed hashing, collisions are handled by storing all colliding elements directly within the hash table.

1.1 Chaining:
- Collisions are resolved by using linked lists (or other data structures) at each hash table slot.
- If two keys hash to the same index, they are simply added to the linked list at that index.

2. Closed Hashing:
- In closed hashing, elements are stored directly in the hash table, and collisions are resolved by finding an alternative location within the same table.

2.1 Linear Probing:
- When a collision occurs, the algorithm searches for the next available slot linearly until an empty slot is found.
- Probing sequence: `hash(key) + i  c` (where `c` is a constant, and `i` is the attempt number).

2.2 Quadratic Probing:

- Similar to linear probing, but the probing sequence is based on a quadratic function, such as `hash(key) + i^2  c`.
- Helps reduce clustering compared to linear probing.

2.3 Double Hashing:
- Uses a secondary hash function to determine the step size between probes.
- Probing sequence: `hash(key) + i  hash2(key)` (where `hash2(key)` is the result of a secondary hash function).
- Aims to minimize clustering and provide a more uniform distribution.

(ii) What ADT?

https://www.prepbytes.com/blog/data-structure/abstract-data-type-adt-in-data-structure/
ADT stands for Abstract Data Type. It refers to a high-level description of a set of operations that can be performed on a data structure, independent of the specific implementation. The term "abstract" in ADT implies that the details of how the operations are implemented are abstracted or hidden from the user.

An ADT defines a set of operations that can be performed on the data, as well as the logical relationships between the operations. It serves as a blueprint for designing data structures by specifying the behavior without detailing the internal workings.

Common examples of abstract data types include:

1. Stack:
- Operations: Push, Pop, Peek.
- Behavior: Follows the Last In, First Out (LIFO) principle.

2. Queue:
- Operations: Enqueue, Dequeue, Front.
- Behavior: Follows the First In, First Out (FIFO) principle.

3. List:
- Operations: Insert, Delete, Find.
- Behavior: An ordered collection of elements.

4. Set:
- Operations: Insert, Delete, Contains.
- Behavior: An unordered collection of unique elements.

5. Map (Dictionary):
- Operations: Insert, Delete, Search.

- Behavior: Associative array mapping keys to values.

6. Graph:
- Operations: Add Vertex, Add Edge, Traverse.
- Behavior: Represents a collection of vertices and edges.

The abstraction provided by ADTs is essential in software design and programming because it allows developers to focus on the functionality of data structures without being concerned with the implementation details. Different implementations of the same ADT may exist, providing flexibility and modularity in software design. The choice of a specific data structure implementation depends on factors such as performance requirements, memory constraints, and the nature of the data being manipulated.

5.

(i) Explain Dictionary Abstract Data Type?

The Dictionary Abstract Data Type (ADT) is a data structure that represents a collection of key-value pairs, where each key is unique within the collection. The Dictionary ADT provides operations to insert, delete, and search for a value associated with a specific key. It is also known by other names, such as Map or Associative Array.

Dictionary ADT Operations:

1. Insert(Key, Value):
- Inserts a new key-value pair into the dictionary. If the key already exists, the associated value is updated.

2. Delete(Key):
- Removes the key-value pair associated with the given key from the dictionary.

3. Search(Key):
- Retrieves the value associated with a specific key. If the key is not present, an indication of absence is returned.

4. Size:
- Returns the number of key-value pairs currently in the dictionary.

5. IsEmpty:
- Checks if the dictionary is empty (contains no key-value pairs).

Dictionary ADT Characteristics:

1. Uniqueness of Keys:
- Each key in the dictionary must be unique. If an attempt is made to insert a key that already exists, the existing value is typically updated.

2. No Specific Order:
- The order of key-value pairs in a dictionary is not guaranteed. It doesn't follow any specific order or sequence.

3. Dynamic Size:
- The size of the dictionary can dynamically change as key-value pairs are inserted or deleted.

(ii) Mention some features of ADT.

Abstract Data Types (ADTs) exhibit several key features that define their characteristics and behavior. Here are some fundamental features of ADTs:

1. Encapsulation:
- Definition: ADTs encapsulate data and operations into a single unit.
- Importance: Internal details are hidden, promoting modularity and information hiding.

2. Abstraction:
- Definition: ADTs provide a high-level view of data and operations, abstracting away implementation details.
- Importance: Allows users to focus on functionality without needing to understand internal workings.

3. Data Organization:
- Definition: ADTs define the organization and relationships of data within the structure.
- Importance: Specifies how data is stored, accessed, and manipulated.

4. Operations:
- Definition: ADTs specify a set of operations that can be performed on the data.
- Importance: Outlines the allowed manipulations and interactions with the data.

5. Behavior Specification:
- Definition: ADTs define the expected behavior of operations, including preconditions and postconditions.
- Importance: Provides a clear understanding of how operations affect the ADT's state.

6. Independence of Implementation:
- Definition: ADTs separate the logical view from the physical implementation.
- Importance: Allows for different implementations of the same ADT based on efficiency, memory considerations, or other requirements.

7. Genericity (Generics):
- Definition: ADTs can be designed to work with different data types.
- Importance: Enhances flexibility and reusability by supporting various data types.

8. Dynamic Memory Management:
- Definition: ADTs often handle memory allocation and deallocation dynamically.
- Importance: Enables the ADT to adapt to varying amounts of data during program execution.

9. Error Handling:
- Definition: ADTs may specify error-handling mechanisms for exceptional situations.
- Importance: Ensures robustness by defining how the ADT behaves in unexpected scenarios.

10. State Mutability:
- Definition: Some ADTs support mutable states (can be modified), while others are immutable (cannot be modified).
- Importance: Determines whether the ADT's state can be changed after creation.

11. Concurrency Support:
- Definition: Some ADTs are designed to be thread-safe or support concurrent access.
- Importance: Ensures proper functioning in multithreaded or concurrent environments.

12. Persistence:
- Definition: Some ADTs support persistent storage, allowing data to be saved and retrieved across program executions.
- Importance: Enables data persistence and recovery.

Understanding these features helps developers choose the appropriate ADT for a given problem and implement it effectively based on the specific requirements and constraints of the application.

6.

(i) Write about linear probing and quadratic probing?

Linear Probing:

Definition:

Linear probing is a collision resolution technique used in hash tables. In the event of a collision (when two keys hash to the same index), linear probing searches for the next available slot sequentially, incrementing the index by a fixed constant until an empty slot is found.

Formula:
The linear probing sequence is typically expressed as **hash(key) + i c**, where **hash(key)** is the hash function, **i** is the attempt number, and **c** is a constant.

Advantages:
1. Simplicity: Linear probing is easy to implement and understand.
2. Cache Efficiency: Linear probing tends to have better cache performance due to its sequential memory access pattern.

Disadvantage:
1. Primary Clustering: Linear probing is prone to primary clustering, where consecutive occupied slots form, potentially leading to more collisions in the same region.

Quadratic Probing:

Definition:
Quadratic probing is another collision resolution technique used in hash tables. Instead of searching for the next slot linearly, quadratic probing uses a quadratic function to determine the step size between probes.

Formula:
The quadratic probing sequence is expressed as **hash(key) + i^2 c**, where `hash(key)` is the hash function, **i** is the attempt number, and **c** is a constant.

Advantages:
1. Reduced Clustering: Quadratic probing aims to reduce clustering by dispersing the search pattern in a non-linear fashion.
2. Even Distribution: The non-linear search pattern tends to result in a more even distribution of elements across the hash table.

Disadvantage:
1. Secondary Clustering: Quadratic probing may still suffer from secondary clustering, where keys that initially collided may collide again due to the quadratic function's nature.

(ii) what is chaining?

(iii) Discuss the concept of Rehashing?

Rehashing is the process of recalculating the hashcode of previously-stored entries (Key-Value pairs) in order to shift them to a larger size hashmap when the threshold is reached/crossed,
When the number of elements in a hash map reaches the maximum threshold value, it is rehashed.

The reason for rehashing is that anytime a new key-value pair is placed into the map, the load factor rises, and as a result, the complexity rises as well. And as the complexity of our HashMap grows, it will no longer have a constant O(1) time complexity. As a result, rehashing is used to disperse the items across the hashmap, lowering both the load factor and the complexity, resulting in **search()** and **insert()** having a constant time complexity of O(1). One should note that the existing objects may fall into the same or a different category after rehashing.

Rehashing can be done in the following way:

- Check the load factor after adding a new entry to the map.
- Rehash if it is more than the predefined value (or the default value of 0.75 if none is specified).
- Make a new bucket array that is double the size of the previous one for Rehash.
- Then go through each element in the old bucket array, calling insert() on each to add it to the new larger bucket array.

Some term used in rehashing:

1. Load Factor:
- The load factor is a key metric that indicates how full the hash table is. It is calculated as the number of elements in the hash table divided by the size of the hash table.

Load Factor = Number of Elements/Size of Hash Table

2. Threshold:

- A threshold load factor is defined to determine when rehashing should be triggered. When the actual load factor exceeds this threshold, it indicates that the hash table is becoming too crowded, leading to potential increases in collisions and reduced efficiency.

Example:

Suppose you have a hash table with a load factor threshold of 0.7, and the current load factor exceeds this threshold. The rehashing process might involve the following steps:

1. Double the size of the hash table.
2. Recalculate the hash values for all existing key-value pairs using the new size.
3. Redistribute the elements to their new positions in the resized hash table.

Rehashing is a crucial mechanism in hash table implementations to maintain efficiency, especially in scenarios where the size and characteristics of the data being stored can vary dynamically.

7.

(i) What is Double Hashing technique?

Double Hashing is a collision resolution technique used in hash tables. When a collision occurs (i.e., two keys hash to the same location), double hashing involves applying a second hash function to determine the offset or step size for probing the next available slot in the hash table.

Formula:

The formula for double hashing involves two hash functions: h1(key) and h2(key). When a collision occurs at the hash position h1(key), the double hashing formula is:

$$\text{hash\_position} = (h1(key) + i \cdot h2(key)) \mod \text{table\_size}$$

Here, **i** is the probing attempt, and **table_size** is the size of the hash table.

Advantages:

1. Reduced Clustering: Double hashing tends to reduce clustering compared to simpler collision resolution techniques like linear probing or quadratic probing. This is because the second hash function introduces additional independence in determining the probe sequence.

2. Better Distribution: With a well-designed pair of hash functions, double hashing can result in a more evenly distributed set of keys across the hash table, reducing the likelihood of collisions.

Disadvantages:

1. Complexity: Double hashing requires the definition and management of two hash functions, which can add complexity to the implementation.

2. Difficulty in Designing Hash Functions: Designing two hash functions that avoid clustering and distribute keys evenly can be challenging. A poorly chosen pair of hash functions may lead to inefficient performance.

Example:

Suppose we have a hash table with a size of 10 slots, and we have two hash functions defined as follows:

$$h1(key) = key \mod 10$$

$$h2(key) = 7 - (key \mod 7)$$

Now, let's say we want to insert a key, and the first hash function results in a collision at position 3. We then use the second hash function to determine the step size:

$$\text{hash\_position} = (3 + i \cdot (7 - (key \mod 7))) \mod 10$$

We keep probing with increasing values of \(i \) until we find an empty slot. This process helps in resolving collisions by probing different positions based on both hash functions.

(ii) What is Data structure? What are the applications of data structure?

A data structure is a way of organizing and storing data to perform operations efficiently. It defines the organization and relationships among data elements to facilitate various operations such as insertion, deletion, retrieval, and traversal. Data structures provide a means to manage and manipulate data effectively in computer programs.

Need of data structure:

1. Data structure modification is easy.
2. It requires less time.
3. Save storage memory space.
4. Data representation is easy.
5. Easy access to the large database.

Applications of Data Structures:

1. Arrays:

- Applications:

- Used for storing and accessing elements with constant-time indexing.
- Implementation of matrices, vectors, and dynamic arrays.

- Example:

- Storing a list of temperatures over time.

2. Linked Lists:

- Applications:

- Efficient insertion and deletion operations.
- Implementation of dynamic data structures like stacks and queues.

- Example:

- Managing a playlist in a music player.

3. Stacks:

- Applications:

- Function call management in recursion.
- Undo mechanisms in applications.

- Example:

- Tracking the execution of nested functions.

4. Queues:

- Applications:

- Task scheduling in operating systems.
- Print job management in printers.

- Example:

- Processing tasks in a shared computing resource.

5. Trees:

- Applications:

- Hierarchical structures like file systems.
- Representing hierarchical relationships in databases.

- Example:

- Organizing information in an organizational chart.

6. Graphs:
- **Applications:**
  - Representing relationships between entities in social networks.
  - Pathfinding algorithms in maps and networks.
- **Example:**
  - Modeling connections between users in a social media platform.

7. Hash Tables:
- **Applications:**
  - Efficient lookup and retrieval of key-value pairs.
  - Implementation of symbol tables and dictionaries.
- **Example:**
  - Storing and retrieving data in a phonebook.

8. Heaps:
- **Applications:**
  - Priority queue implementation.
  - Heap sort algorithm.
- **Example:**
  - Managing tasks with different priorities.

9. Graphical Data Structures:
- **Applications:**
  - Spatial data structures for efficient search in geographic information systems.
  - Quadtree and octree structures for image compression.
- **Example:**
  - Storing and querying spatial data in a geographical map.

10. Advanced Data Structures:
- **Applications:**
  - Self-balancing binary search trees for efficient search operations.
  - B-trees and AVL trees for database indexing.
- **Example:**
  - Maintaining a sorted database for quick search and retrieval.

Data structures are fundamental components of algorithm design and play a crucial role in optimizing the efficiency of software applications. They provide a way to organize and manage data in a manner that suits the specific needs and requirements of different algorithms and applications.

(iii) Explain where hashing is used in real time with an example?

Hashing is widely used in real-time applications across various domains due to its efficiency in data retrieval and storage. Here are some examples of real-time applications where hashing is commonly employed:

1. Databases:
- Example: In database management systems, hashing is used for indexing and retrieval of records. Each record is assigned a unique hash key based on its content, allowing for fast access and search operations.

2. Caching Systems:
- Example: Web browsers and content delivery networks (CDNs) use hashing to implement caching systems. Cached content is indexed and retrieved using hash keys, reducing the latency in serving frequently accessed resources.

3. Password Storage:
- Example: Hash functions are employed to securely store passwords. Instead of storing actual passwords, systems store their hash values. During authentication, the hash of the entered password is compared with the stored hash, enhancing security.

4. Distributed Systems:
- Example: Hashing is used in distributed systems for load balancing and data partitioning. Consistent hashing, for instance, ensures that data is evenly distributed across nodes, minimizing the impact of node additions or removals.

5. File Integrity Checking:
- Example: Hash functions are used to generate checksums or hash values for files. These checksums help verify the integrity of files during transmission or storage by comparing the computed hash with the original hash.

6. Digital Signatures:
- Example: Hash functions are a critical component in digital signatures. A hash of the data is signed to ensure data integrity. Any alteration in the data would result in a different hash value, making the signature invalid.

7. Blockchain Technology:
- Example: Hashing is a fundamental part of blockchain technology. Each block in a blockchain contains a hash of the previous block, creating a chain of blocks. This ensures data integrity and immutability of the entire blockchain.

8. Network Routing:
- Example: Hashing is used in routing algorithms to distribute network traffic across multiple paths. Hashing helps ensure a balanced distribution of traffic, optimizing network performance.

9. Distributed Hash Tables (DHTs):
- Example: DHTs, used in peer-to-peer networks, employ hashing to distribute and locate data across nodes efficiently. Each node is responsible for a specific range of hash values.

10. Cryptography:
- Example: Hash functions are used in cryptographic applications, such as creating digital signatures, message authentication codes (MACs), and hash-based message authentication codes (HMACs).

8.

(i) Explain Recent Trends in Hashing?

1. Blockchain and Cryptocurrencies:
- Hash functions are a fundamental component of blockchain technology. The rise of cryptocurrencies and decentralized applications has fueled interest in secure hash functions. Researchers and developers continue to explore cryptographic hash functions suitable for blockchain applications.

2. Post-Quantum Cryptography:
- With the advent of quantum computers, there's a growing interest in post-quantum cryptographic algorithms, including hash functions. Researchers are exploring hash functions that remain secure in a quantum computing environment, considering potential threats to existing cryptographic systems.

3. Homomorphic Hash Functions:
- Homomorphic encryption allows computations on encrypted data without decrypting it. Homomorphic hash functions extend this concept to hashing, enabling secure operations on hashed data. This has potential applications in privacy-preserving computations.

4. Verifiable Delay Functions (VDFs):
- Verifiable Delay Functions, based on the concept of time-lock puzzles, have gained attention. VDFs are used in blockchain consensus mechanisms and other cryptographic protocols to introduce a time delay that can be efficiently verified.

5. Quantum-Secure Hash Functions:
- With the increasing interest in quantum-safe cryptography, researchers are working on developing hash functions and cryptographic algorithms that are resistant to attacks from

quantum computers. Quantum-resistant hash functions aim to ensure data security in a post-quantum era.

6. Blockchain Interoperability:
- As the number of blockchain networks grows, there's a need for interoperability. Hash functions play a role in cross-chain communication and interoperability solutions, allowing different blockchain networks to communicate and share data securely.

7. Hash-Based Data Structures:
- Research continues in the development of hash-based data structures that provide efficient and scalable solutions for various applications. This includes hash tables, hash trees, and other structures used in databases, distributed systems, and networking.

8. Secure Multi-Party Computation (SMPC):
- Hash functions are integral to secure multi-party computation, enabling parties to jointly compute a function over their inputs while keeping those inputs private. This has applications in privacy-preserving collaborative data analysis.

9. Energy-Efficient Hashing:
- With a focus on energy efficiency and sustainability, there is ongoing research into developing energy-efficient hash functions. This is particularly important in applications where resources are constrained, such as in IoT devices and edge computing.

10. Machine Learning and Hashing:
- Hash functions are used in machine learning for tasks such as feature hashing and locality-sensitive hashing. Recent trends involve optimizing and adapting hash functions to improve the efficiency of machine learning algorithms.

(ii) Write about separate chaining and open addressing?

Separate Chaining:

1. Collision Resolution Technique:
- Method: Separate chaining involves maintaining a linked list at each index in the hash table to handle collisions.

2. Data Organization:
- Linked Lists: Each index in the hash table holds a linked list of key-value pairs.

3. Insertion and Retrieval:
- Insertion: Upon collision, new key-value pairs are added to the linked list at the corresponding index.

- Retrieval: During retrieval, the hash table index is first identified, and then a linear search through the linked list is performed to find the desired key.

4. Efficiency:
- Advantages: Suitable when collisions are infrequent, as it provides a flexible way to accommodate multiple values at the same index.
- Drawbacks: Can become inefficient if the linked lists become long, impacting search times.

5. Example:

hashTable[index] -> Linked List of Key-Value Pairs

Open Addressing:

1. Collision Resolution Technique:
- Method: Open addressing is an alternative method for handling collisions in hash tables by placing the colliding key in the next available ("open") slot.

2. Data Organization:
- Array-Based Storage: The hash table is a one-dimensional array, and when a collision occurs, the algorithm looks for the next available slot in the array.

3. Insertion and Retrieval:
- Insertion: Upon collision, the algorithm probes for the next open slot until an empty space is found.
- Retrieval: During retrieval, if the key is not found at the expected index, the algorithm continues probing until the key is located or an empty slot is encountered.

4. Efficiency:
- Advantages: Can be more memory-efficient than separate chaining, especially when the load factor is low.
- Drawbacks: Prone to clustering, and the choice of probing strategy (linear probing, quadratic probing, etc.) impacts performance.

5. Example:
hashTable[index] -> Key-Value Pair or Next Open Slot

(iii) Convert the infix (a+b)(c+d)/f into a postfix & prefix expression.

Infix to Postfix:

Infix to Postfix table

Sr No	Expression	Stack	Postfix
0	(	(	
1	a	(	a
2	+	(+	a
3	b	(+	ab
4	)		ab+
5	*	*	ab+
6	(	*(	ab+
7	c	*(	ab+c
8	+	*(+	ab+c
9	d	*(+	ab+cd
10	)	*	ab+cd+
11	/	/	ab+cd+*
12	f	/	ab+cd+*f
13			ab+cd+*f/

Infix:(a+b)(c+d)/f

1. 1. Reversed string: f/(d+c)*(b+a)
2. 2. Postfix of f/(d+c)*(b+a): fdc+/ba+*

(see table)

Infix to Postfix table

Sr No	Expression	Stack	Postfix
0	f		f
1	/	/	f
2	(	/(	f
3	d	/(	fd
4	+	/(+	fd
5	c	/(+	fdc
6	)	/	fdc+
7	*	*	fdc+/
8	(	*(	fdc+/
9	b	*(	fdc+/b
10	+	*(+	fdc+/b
11	a	*(+	fdc+/ba
12	)	*	fdc+/ba+
13			fdc+/ba+*

3. 3. Reversed string of fdc+/ba+*:

*+ab/+cdf

9.

(i) Differentiate between hashing and extendible hashing.

Feature	Hashing	Extendible Hashing
Collision Resolution	Separate chaining, open addressing (linear, quadratic, double hashing)	Dynamic directory structure for collision resolution
Directory Structure	Fixed-size hash table with a static number of buckets	Dynamic directory that grows or shrinks based on the data
Bucket Size	Fixed	Variable, determined by the directory's depth
Search Time	O(1) on average if few collisions	O(1) on average, efficient due to dynamic directory structure
Space Utilization	May lead to inefficient space utilization due to fixed-size table	Efficient use of space, as the directory dynamically adjusts
Insertion and Deletion	May require rehashing and resizing for dynamic data	Efficient insertion and deletion without the need for global reorganization
Complexity	Simpler to implement, but may require tuning	More complex due to the dynamic directory structure, but offers adaptability
Scalability	May become less efficient as the number of elements increases	More scalable due to the adaptability of the directory
Applications	Common in scenarios with a relatively stable dataset size	Suitable for scenarios with dynamic and changing data sizes

10.

(i) What do you mean by linear open addressing?

Linear Open Addressing:

Linear Probing is a form of open addressing, a technique used to resolve collisions in hash tables. In linear probing, when a collision occurs (i.e., the desired hash position is already occupied), the algorithm searches for the next available slot in a linear manner, checking the next position in sequence.

Formula:

If a collision occurs at position $h(key)$ in the hash table, linear probing attempts to resolve it using the following formula:

$$\text{hash\_position} = (h(key) + i) \mod \text{table\_size}$$

Here, i is the probing attempt, and table_size is the size of the hash table.

Advantages:

1. Simplicity: Linear probing is simple to implement and understand. The logic for finding the next available slot is straightforward, making it a practical choice for basic hash table implementations.

2. Memory Efficiency: Linear probing tends to use memory more efficiently compared to other open addressing techniques, as it can result in a more compact representation of the data in the table.

Disadvantages:

1. Clustering: One of the main drawbacks of linear probing is the clustering phenomenon. Collisions tend to create clusters of occupied slots, which can lead to performance degradation, especially when these clusters become larger.

2. Secondary Clustering: Linear probing is susceptible to secondary clustering, where consecutive collisions tend to create long sequences of filled slots, increasing the likelihood of further collisions.

Example:

Let's consider a hash table with a size of 10 slots and a simple hash function:

$$h(key) = key \mod 10$$

Now, suppose we want to insert a key, and the hash function results in a collision at position 3. With linear probing, we would then look for the next available slot using the formula:

$$\text{hash\_position} = (3 + i) \mod 10$$

If the next position (3 + 1 mod 10) is also occupied, we would check the following position (3 + 2 mod 10), and so on, until an empty slot is found. This linear search continues until an available slot is located, resolving the collision.

(ii) What are the differences between static hashing and dynamic hashing?

Feature	Static Hashing	Dynamic Hashing
Directory Structure	Fixed-size directory	Dynamic directory that grows or shrinks based on data
Bucket Size	Fixed-size buckets	Variable-size buckets, adjusted dynamically
Handling Overflows	May require periodic rehashing and resizing	Handles overflows dynamically without rehashing the entire structure
Memory Usage	May lead to inefficient use of memory due to fixed-size directory	Efficient use of memory, adapts to data distribution
Insertion and Deletion	May require global reorganization for dynamic datasets	Efficient insertion and deletion without global reorganization
Search Time	Stable search times, but may degrade with rehashing	Stable search times due to dynamic directory adjustments
Complexity	Simpler to implement and manage	More complex due to dynamic directory structure
Scalability	Less scalable for dynamic datasets	More scalable, adapts to changing data sizes
Applications	Suitable for scenarios with relatively stable dataset sizes	Well-suited for dynamic and changing data sizes

11.

(i) What is a skip list?

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view

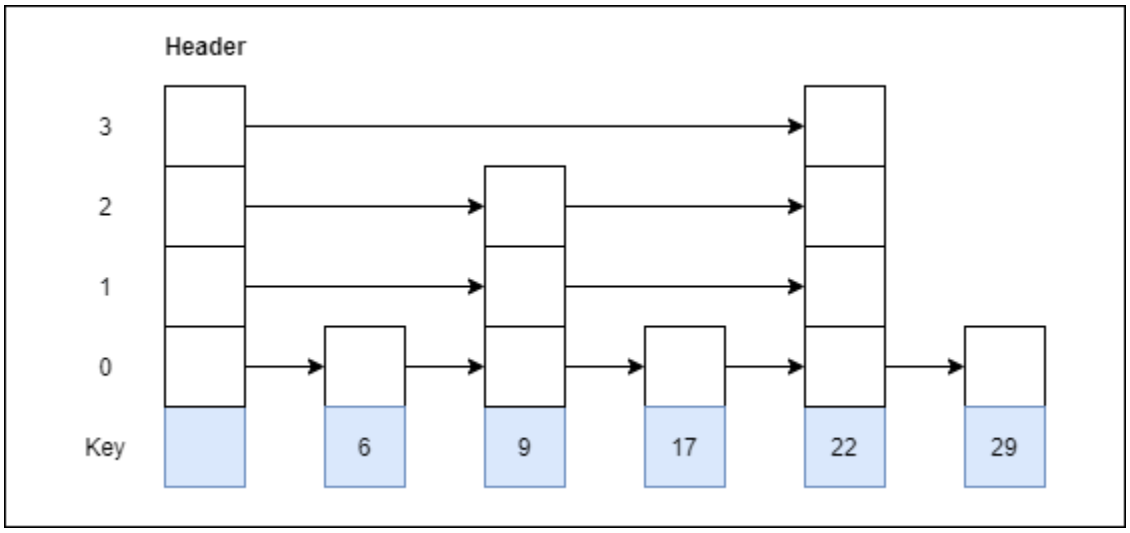
efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped



Complexity table of the Skip list

S. No	Complexity	Average case	Worst case
1.	Access complexity	O(logn)	O(n)
2.	Search complexity	O(logn)	O(n)
3.	Delete complexity	O(logn)	O(n)
4.	Insert complexity	O(logn)	O(n)
5.	Space complexity	-	O(nlogn)

.

Advantages of Skip List:

- The skip list is solid and trustworthy.
- To add a new node to it, it will be inserted extremely quickly.
- Easy to implement compared to the hash table and binary search tree
- The number of nodes in the skip list increases, and the possibility of the worst-case decreases
- Requires only ?(logn) time in the average case for all operations.
- Finding a node in the list is relatively straightforward.

Disadvantages of Skip List:

- It needs a greater amount of memory than the balanced tree.
- Reverse search is not permitted.
- Searching is slower than a linked list
- Skip lists are not cache-friendly because they don't optimize the locality of reference

(ii) Write about open addressing techniques?

Open Addressing, also known as closed hashing, is a simple yet effective way to handle collisions in hash tables. Unlike chaining, it stores all elements directly in the hash table. This method uses probing techniques like Linear, Quadratic, and Double Hashing to find space for each key, ensuring easy data management and retrieval in hash tables.

The main concept of Open Addressing hashing is to keep all the data in the same hash table and hence a bigger Hash Table is needed. When using open addressing, a collision is resolved by probing (searching) alternative cells in the hash table until our target cell (empty cell while insertion, and cell with value x while searching x) is found. It is advisable to keep load factor (α) below 0.5, where α is defined as $\alpha = n/m$ where n is the total number of entries in the hash table and m is the size of the hash table. As explained above, since all the keys are stored in the same hash table so it's obvious that $\alpha \leq 1$ because $n \leq m$ always. If in case a collision happens then, alternative cells of the hash table are checked until the target cell is found. More formally,

- Cells $h_0(x), h_1(x), h_2(x)....h_n(x)$ are tried consecutively until the target cell has been found in the hash table. Where $h_i(x) = (hash(x) + f(i))\%Size$, keeping $f(0) = 0$.
- The collision function f is decided according to method resolution strategy.

There are three main Method Resolution Strategies --

1. Linear Probing
2. Quadratic Probing
3. Double Hashing

2.1 Linear Probing:

- When a collision occurs, the algorithm searches for the next available slot linearly until an empty slot is found.
- Probing sequence: `hash(key) + i  c` (where `c` is a constant, and `i` is the attempt number).

2.2 Quadratic Probing:
- Similar to linear probing, but the probing sequence is based on a quadratic function, such as `hash(key) + i^2  c`.
- Helps reduce clustering compared to linear probing.

2.3 Double Hashing:
- Uses a secondary hash function to determine the step size between probes.
- Probing sequence: `hash(key) + i  hash2(key)` (where `hash2(key)` is the result of a secondary hash function).
- Aims to minimize clustering and provide a more uniform distribution.

(iii) Explain search and update operations on skip lists?

Search Operation on Skip Lists:

1. Starting Point:
- Begin at the top-left corner (head) of the skip list.

2. Level Traversal:
- Move down a level if the key in the forward pointer is less than or equal to the target key.

3. Horizontal Movement:
- Move horizontally until reaching a node with a forward pointer greater than the target key or the end of the level.

4. Reaching Bottom Level:
- Repeat the process, moving down and horizontally, until reaching the bottom level.

5. Search Completion:
- Finish when the target key is found or it's determined that the key is not present.

Update Operation on Skip Lists:

1. Search for Node:
- Conduct a search to find the node with the target key.

2. Update Node's Value:

- Update the found node's value with the new value.

3. Optional: Reorganize Levels:
- Optionally update the same key in higher levels if it exists.

4. Optional: Adjust Levels Dynamically:
- Optionally adjust the height of the node in the skip list.

5. Optional: Rebalance Skip List:
- Optionally perform rebalancing to maintain skip list properties.

Efficiency stems from the hierarchical structure, enabling quick vertical and horizontal movement. Multiple levels reduce average traversal, resulting in logarithmic time complexity. Probabilistic nature and balanced structure contribute to efficient dynamic updates.

12.

(i) Explain Skip List. Why is it called a Randomized Data structure?

Skip List:
Question 11

Why Randomized:
- The term "randomized" comes from the probabilistic nature of promoting elements during insertion.
- Random promotions result in a balanced structure, preventing worst-case scenarios.
- Simplifies insertion, deletion, and adaptability to changing data distributions.

Or,
A Skip List is considered a randomized data structure because its construction involves a randomization process. The randomization in skip lists refers to the probabilistic decisions made during the insertion of elements and the creation of levels in the data structure.

Here's why Skip Lists are called randomized data structures:

1. Randomized Decision during Insertion:
When a new element is inserted into a skip list, decisions about how many levels the element should have and at which levels it should be placed are made probabilistically. This randomness helps balance the distribution of elements across different levels of the skip list.

2. Probabilistic Structure:
Skip lists maintain a balance between being deterministic and probabilistic. The structure of a skip list is probabilistic in the sense that the decision to include an element at a particular level is based on coin flips or random choices. This probabilistic approach helps in achieving an average-case time complexity that is often more favorable than worst-case scenarios.

3. Efficiency with Randomization:
The use of randomness in skip lists contributes to their efficiency. The probabilistic nature of the structure helps avoid worst-case scenarios that might arise in strictly deterministic structures. On average, the performance of skip lists is good due to the randomization during insertion.

4. Simplicity and Effectiveness:
The randomized nature of skip lists simplifies their design and makes them effective in practice. Randomization can help prevent the creation of imbalanced structures, leading to better average-case performance.

In summary, the term "randomized" in the context of skip lists refers to the probabilistic decisions made during their construction and maintenance. The use of randomness contributes to a more balanced and efficient data structure, with average-case time complexities that are often favorable in practical scenarios.

(ii) What are the advantages of Skip List?

Advantages of Skip Lists:

1. Efficient Search Operations:
- Skip lists provide logarithmic time complexity for search operations, making them efficient for large datasets.

2. Simplicity and Ease of Implementation:
- Skip lists are simpler to implement compared to some other balanced data structures, such as AVL trees or red-black trees.

3. Dynamic Structure:
- Skip lists dynamically adapt to changing data distributions and insertion patterns without requiring complex rebalancing operations.

4. Balanced Distribution:
- The randomized nature of skip lists ensures a balanced distribution of elements across levels, preventing worst-case scenarios.

5. Probabilistic Structure:
- The probabilistic approach to promoting elements during insertion simplifies the balancing process.

6. Ease of Insertion and Deletion:
- Insertion and deletion operations are straightforward and do not require global rebalancing, making them efficient.

7. Skip-Over Mechanism:
- The use of multiple levels enables skipping over elements during searches, reducing the number of comparisons needed.

8. Adaptability:
- Skip lists adapt dynamically to the dataset size and insertion patterns, making them suitable for scenarios with evolving data.

9. Cache-Friendly:
- Skip lists are cache-friendly, as consecutive elements in the same level are stored contiguously in memory.

10. Versatility:
- Skip lists can be used as an alternative to more complex data structures in various applications, striking a balance between simplicity and efficiency.

In summary, the advantages of skip lists lie in their efficient search operations, simplicity, adaptability to dynamic datasets, and ease of insertion and deletion. They offer a practical compromise between complexity and performance in various scenarios.

13.

(i) Explain the Operations Insertion and Searching with a Skip List.

Insertion Operation in Skip List:

1. Search for Insertion Point:
- Traverse levels horizontally and vertically to find the insertion point for the new key.

2. Promotion Decision:
- Decide probabilistically to promote the new key to each level during traversal.

3. Insert and Update Pointers:
- Insert the key at the determined position in the bottom-level list.
- Update pointers horizontally and vertically at each level.

4. Optional Reorganization:
- Optionally reorganize the skip list, adjusting node heights or rebalancing.

Searching Operation in Skip List:

1. Start at the Top:
- Begin at the top-left corner (head) of the skip list.

2. Level Traversal:
- Move down a level if the key in the forward pointer is less than or equal to the target key.

3. Horizontal Movement:
- Move horizontally until reaching a node with a forward pointer greater than the target key or reaching the end.

4. Reaching Bottom Level:
- Repeat the process, moving down and horizontally, until reaching the bottom level.

5. Search Completion:
- Finish when the target key is found or determined to be absent.

(ii) What are the disadvantages of Skip List?

Disadvantages of Skip Lists:

1. Space Overhead:
- Additional pointers for each level can result in increased space consumption.

2. Complex Deletion:
- Deletion operations may involve updating pointers across multiple levels, adding complexity.

3. Not Cache-Friendly:
- Individual levels may not be contiguous in memory, potentially affecting cache efficiency.

4. Limited Use Cases:
- Skip lists may be less suitable for scenarios where memory usage is a critical concern.

5. Implementation Challenges:
- Implementing and maintaining the probabilistic nature of skip lists can be challenging.

6. Alternative Data Structures:
- In certain scenarios, more specialized data structures might offer better performance.

14.

(i) Explain Skip List. Why is it called a Randomized Data structure?

Check out 12 no question

(ii) Explain the Operations Insertion, Deletion and Searching with a Skip List

Operations in Skip List:

1. Insertion:
- Search for the insertion point by traversing levels horizontally and vertically.
- Decide probabilistically to promote the new key to each level during traversal.
- Insert the key at the determined position in the bottom-level list.
- Update pointers horizontally and vertically at each level.
- Optionally reorganize the skip list, adjusting node heights or rebalancing.

2. Searching:
- Start at the top-left corner (head) of the skip list.
- Move down a level if the key in the forward pointer is less than or equal to the target key.
- Move horizontally until reaching a node with a forward pointer greater than the target key or reaching the end.
- Repeat the process, moving down and horizontally, until reaching the bottom level.
- Finish when the target key is found or determined to be absent.

3. Deletion:
- Search for the node with the target key.
- Delete the node from the bottom-level list.
- Update pointers horizontally and vertically to maintain the skip list structure.
- Optionally reorganize the skip list based on certain criteria.

Note: Deletion in skip lists involves more complex pointer updates compared to insertion and searching, and the optional reorganization step is dependent on the specific implementation.

15.

(i) What are 2-3 trees how it works with data structures discuss with an example?

A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.

Here are the properties of a 2-3 tree:

1. each node has either one value or two value
2. a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
3. a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
4. all leaf nodes are at the same level of the tree

Here's a concise explanation of 2-3 trees, their structure, operations, and an example:

2-3 Trees:

- Definition: A self-balancing search tree where each internal node can have either 2 or 3 children (hence the name).
- Key Properties:
    - Maintain sorted order of keys.
    - All leaf nodes are at the same level (height-balanced).
    - Nodes with 2 children are called 2-nodes, those with 3 children are 3-nodes.
-

Structure:

- Root: Can be a 2-node or a 3-node.
- Internal Nodes:

    - 2-nodes store one key and have two children: left child has keys less than the node's key, right child has keys greater than the node's key.
    - 3-nodes store two keys and have three children: left child has keys less than the first key, middle child has keys between the two keys, right child has keys greater than the second key.
-
- Leaf Nodes:
    - Have no children.
    - Can store one or two keys.
-

Operations:

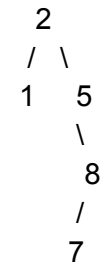
- Insertion: Involves finding the appropriate leaf node and potentially splitting nodes to maintain balance.
- Deletion: Involves finding the node to delete and potentially merging nodes to maintain balance.
- Search: Similar to binary search trees, recursively comparing keys with node values.

Example:

Consider inserting keys 5, 2, 8, 1, 7, and 3 into a 2-3 tree:

1. Insert 5 as the root (2-node).
2. Insert 2 as the left child of 5 (2-node).
3. Insert 8 as the right child of 5 (3-node).
4. Insert 1 as the left child of 2 (3-node).
5. Insert 7 as the right child of 8 (2-node).
6. Insert 3 splits the 3-node (2, 1) into two 2-nodes, promoting 2 as the parent and creating a new root (2) with 5 and 8 as children.

The resulting tree:

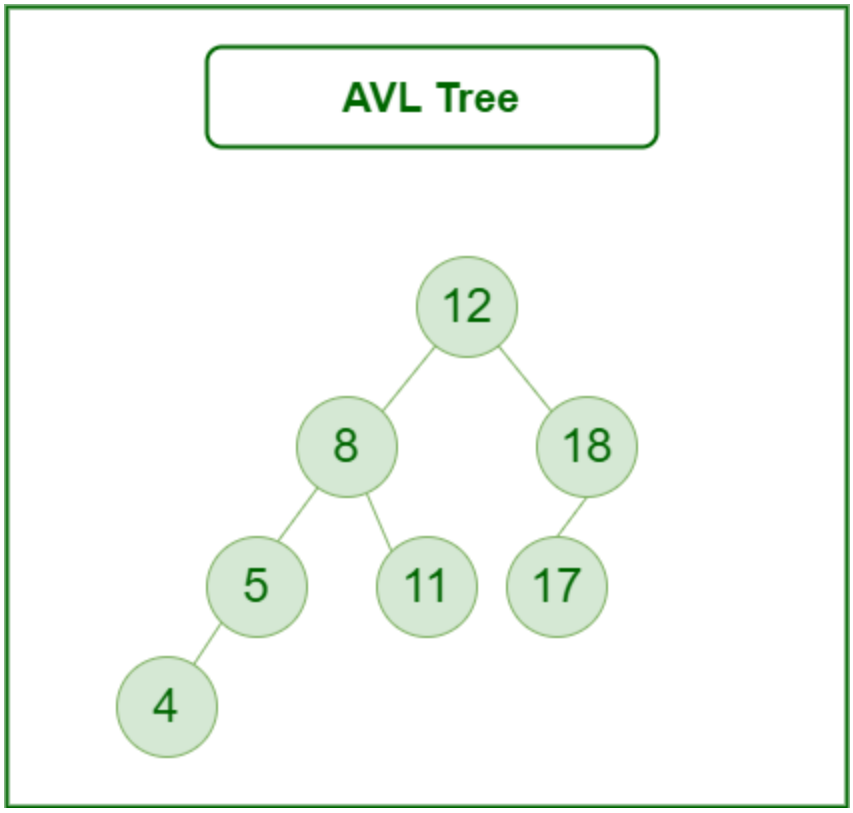


Key Advantages:

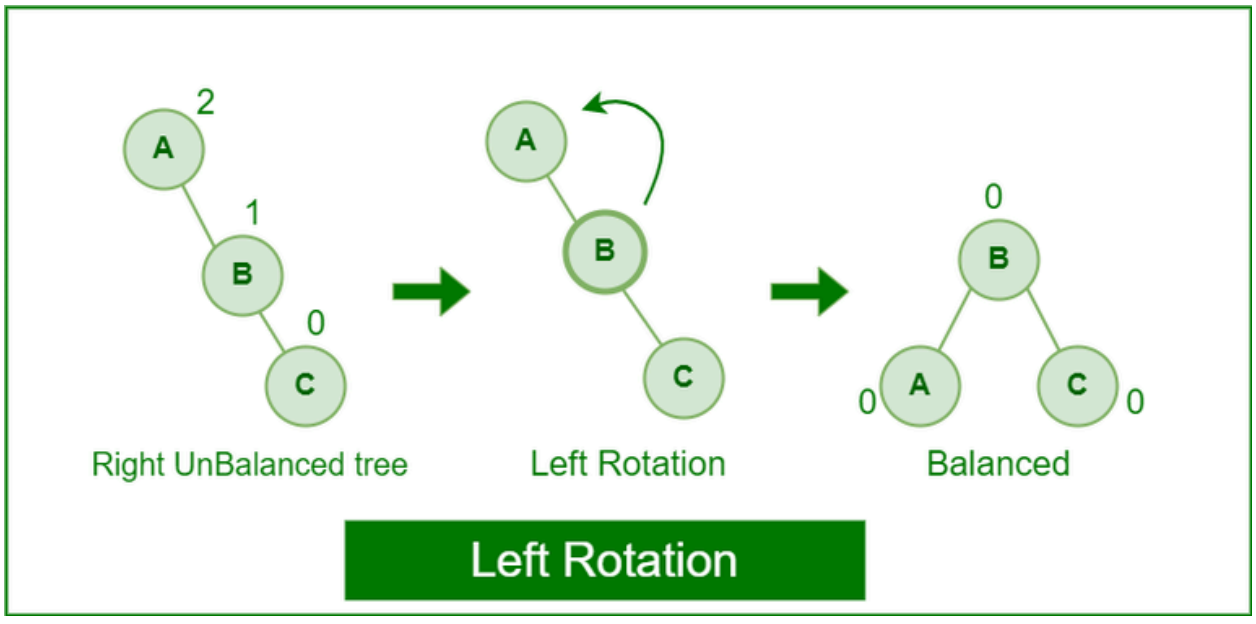
- Efficient search, insertion, and deletion operations (O(log n) time complexity).
- Self-balancing, ensuring good performance even with large datasets.
- Used in database systems, file systems, and other applications requiring efficient data management.

(ii) what is AVL tree?

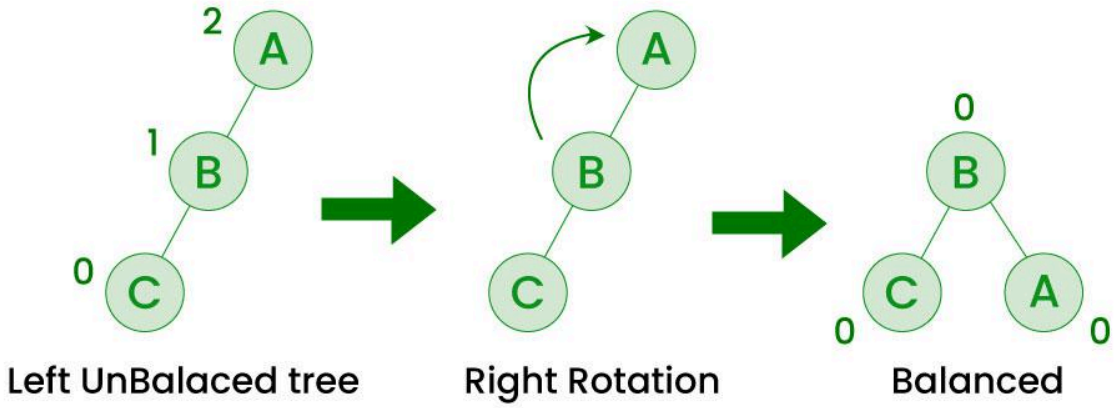
The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.
The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.
Example of AVL Trees:



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.
Rotating the subtrees in an AVL Tree:
An AVL tree may rotate in one of the following four ways to keep itself balanced:
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.

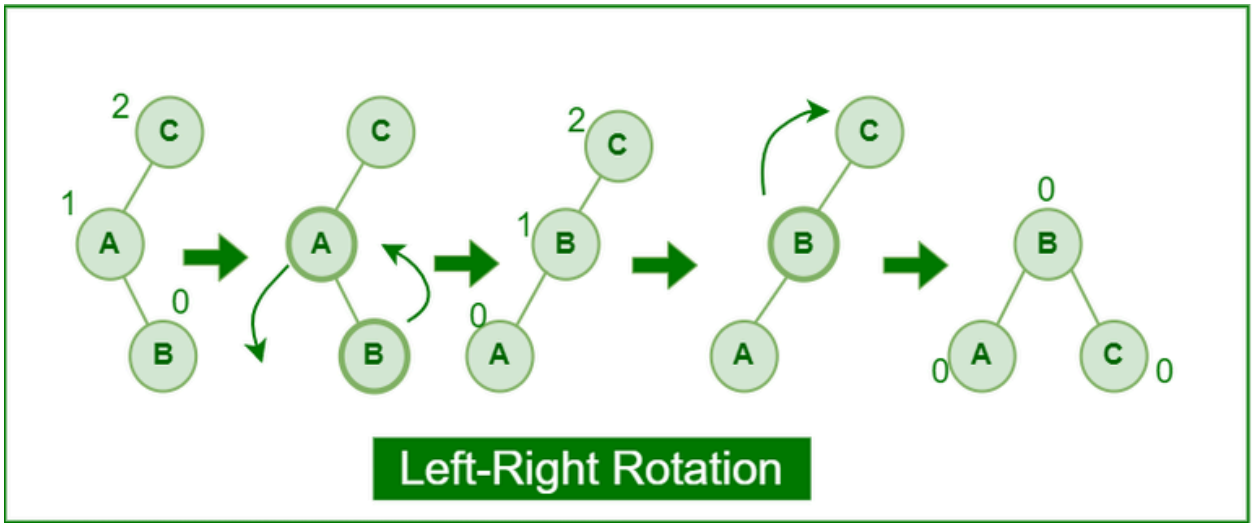


Right Rotation:
If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.

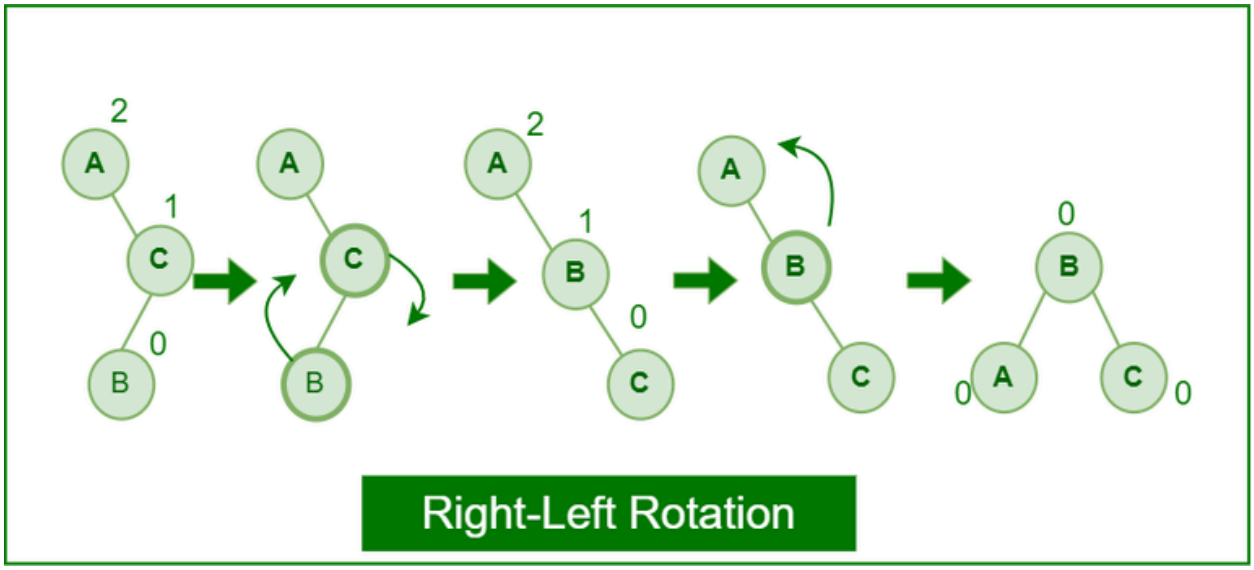


AVL Tree

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



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



Applications of AVL Tree:

1. It is used to index huge records in a database and also to efficiently search in that.
2. For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
3. Database applications, where insertions and deletions are less common but frequent data lookups are necessary
4. Software that needs optimized search.
5. It is applied in corporate areas and storyline games.

Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed log(N), where, N is the total number of nodes in the tree.

Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

(iii) Explain the properties of Red Black Trees with an example.

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

Properties of Red Black Tree:

The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –
1. Root property: The root is black.
2. External property: Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.
3. Internal property: The children of a red node are black. Hence possible parent of red node is a black node.
4. Depth property: All the leaves have the same black depth.
5. Path property: Every simple path from root to descendant leaf node contains same number of black nodes.
The result of all these above-mentioned properties is that the Red-Black tree is roughly balanced.

Rules That Every Red-Black Tree Follows:

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. Every leaf (e.i. NULL node) must be colored BLACK.

Example:

Consider the following red-black tree:

In this example:
- Nodes are labeled with their values.
- Node colors are represented by (R) for red and (B) for black.
- The tree follows the red-black tree properties:
  - Binary search tree property.

- Root (20) and leaves are black.
- No consecutive red nodes along any path.
- Black height is the same for all paths.

This red-black tree maintains balance through rotations and color changes during insertions and deletions.

16.

(i) Explain Splay- trees with neat diagram?

A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. What makes the splay tree unique are two trees. It has one extra property that makes it unique is splaying.

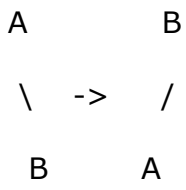
A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

Operations in a splay tree:

- **Insertion:** To insert a new element into the tree, start by performing a regular binary search tree insertion. Then, apply rotations to bring the newly inserted element to the root of the tree.
- **Deletion:** To delete an element from the tree, first locate it using a binary search tree search. Then, if the element has no children, simply remove it. If it has one child, promote that child to its position in the tree. If it has two children, find the successor of the element (the smallest element in its right subtree), swap its key with the element to be deleted, and delete the successor instead.
- **Search:** To search for an element in the tree, start by performing a binary search tree search. If the element is found, apply rotations to bring it to the root of the tree. If it is not found, apply rotations to the last node visited in the search, which becomes the new root.
- **Rotation:** The rotations used in a splay tree are either a Zig or a Zig-Zig rotation. A Zig rotation is used to bring a node to the root, while a Zig-Zig rotation is used to balance the tree after multiple accesses to elements in the same subtree.

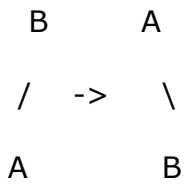
1. Zig Rotation:

- Single rotation to the right.



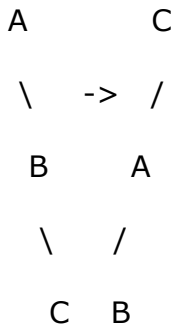
2. Zag Rotation:

- Single rotation to the left.



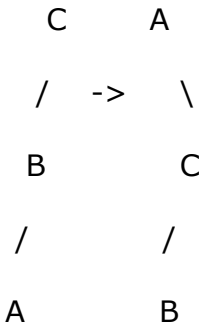
3. Zig-Zig Rotation:

- Double rotation to the right.



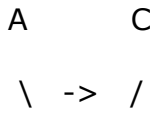
4. Zag-Zag Rotation:

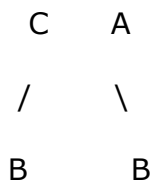
- Double rotation to the left.



5. Zig-Zag Rotation:

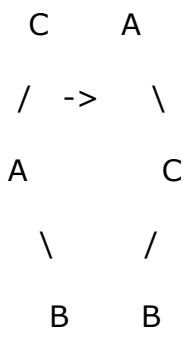
- First, a zag rotation (left), followed by a zig rotation (right).





6. Zag-Zig Rotation:

- First, a zig rotation (right), followed by a zag rotation (left).



These rotations are used in the splaying process to move the most recently accessed node to the root, maintaining the dynamic balance and improving the efficiency of the tree for frequently accessed nodes.

https://www.javatpoint.com/splay-tree
Here's an explanation of Splay trees with diagrams:

Splay Trees:

- Self-adjusting binary search trees with a unique property: they rearrange themselves after each access (search, insertion, deletion), bringing the most recently accessed node to the root.
- This "splaying" process keeps frequently accessed items near the top, improving average access time.

Key Concepts:

- Splaying: A series of tree rotations that move a target node to the root while maintaining the binary search tree property.
- Rotations: Operations that change the structure of the tree while preserving the order of elements.
    - Zig: Right rotation around a node's parent.
    - Zig-zig: Two consecutive right rotations.
    - Zig-zag: Right-left rotation.
-

Diagram (Before Splaying):

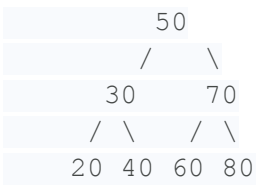
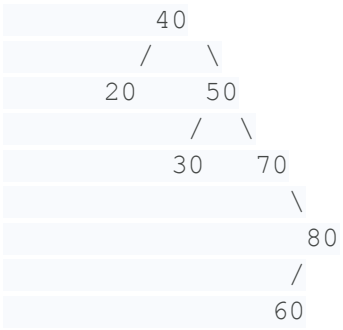


Diagram (After Splaying Node 40):



Operations:

- Search: Locate a node, then splay it to the root.
- Insertion: Insert a node as in a regular BST, then splay it to the root.
- Deletion: Splay the node to the root, then delete it (and adjust its child).

Advantages:

- Good for frequently accessed items and non-uniform access patterns.
- Simpler to implement than other self-balancing trees like AVL or Red-Black trees.
- Adapts well to changing access patterns.

Disadvantages:

- Worst-case time complexity for operations is O(n), although amortized time complexity is O(log n).
- Not as space-efficient as some other self-balancing trees.

Common Use Cases:

- Cache implementations
- File systems

- Web servers
- Network routing tables
- Any application with non-uniform access patterns or where frequent items need to be quickly retrieved.

17.

(i) Explain Red black trees with an example?

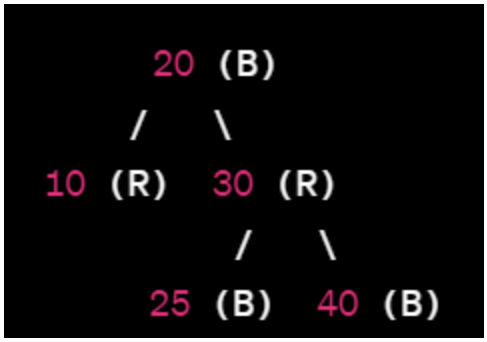
The red-Black tree is a binary search tree. The prerequisite of the red-black tree is that we should know about the binary search tree. In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.

Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc. In a binary search tree, the searching, insertion and deletion take **O(log2n)** time in the average case, **O(1)** in the best case and **O(n)** in the worst case.

Properties of Red Black Tree

- Property 1: Red - Black Tree must be a Binary Search Tree.
- Property 2: The ROOT node must be colored BLACK.
- Property 3: The children of the Red colored nodes must be colored BLACK. (There should not be two consecutive RED nodes).
- Property 4: In all the paths of the tree, there should be the same number of BLACK colored nodes.
- Property 5: Every new node must be inserted with a RED color.
- Property 6: Every leaf (e.i. NULL node) must be colored BLACK.

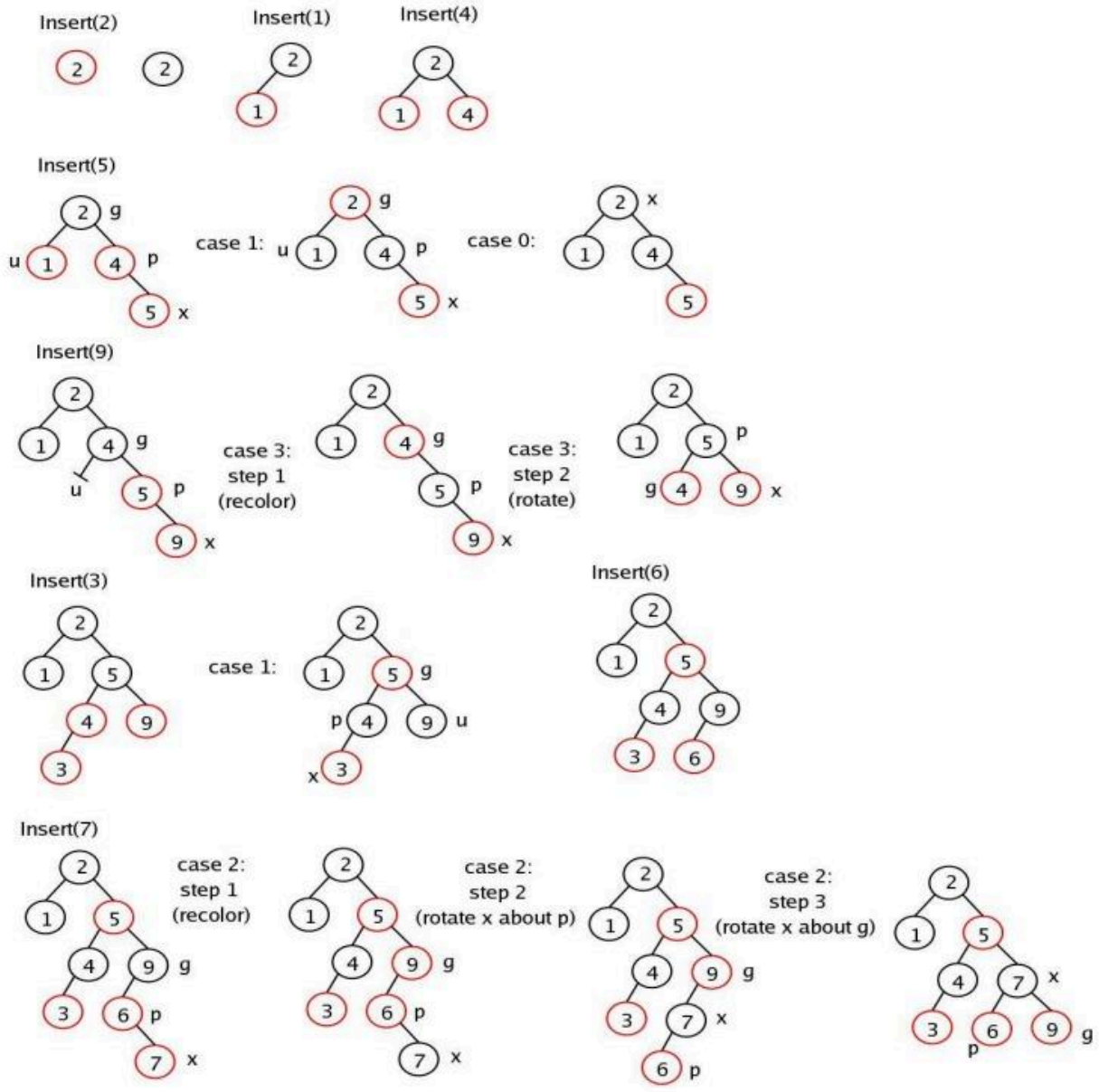
Example:



- Nodes are labeled with their values.
- Node colors are represented by (R) for red and (B) for black.
- Follows the red-black tree properties:
  - Binary search tree property.
  - Root (20) and leaves are black.
  - No consecutive red nodes along any path.
  - Black height is the same for all paths.

This red-black tree maintains balance through rotations and color changes during insertions and deletions.

(ii) construct the Red Black tree with following elements: Insert 2, 1, 4, 5, 9, 3, 6, 7



18.

(i) Explain the issues with AVL Tree and recommend how Red Black trees can be a solution for it.

Issues with AVL Trees:

1. Complexity of Rotations:
- AVL trees require more complex rotations (single and double rotations) during insertion and deletion to maintain balance.

2. Frequent Rotations:
- Frequent rotations can lead to more overhead and decreased performance, especially in scenarios with frequent insertions and deletions.

3. Memory Overhead:
- AVL trees may require an additional balance factor for each node, increasing memory overhead.

4. Strict Balance Requirement:
- Strict balance requirements of AVL trees make them more sensitive to insertions and deletions, potentially leading to frequent rebalancing.

Red-Black Trees as a Solution:

1. Simpler Rotations:
- Red-Black Trees use simpler rotations during insertion and deletion, leading to less overhead in terms of both implementation complexity and performance.

2. Less Strict Balance:
- Red-Black Trees are less strict in terms of balance requirements, making them more adaptable to dynamic datasets with frequent changes.

3. Memory Efficiency:
- Red-Black Trees use one bit per node to store color information, reducing memory overhead compared to AVL trees.

4. Ease of Implementation:
- Red-Black Trees are generally easier to implement and maintain than AVL trees, making them a practical choice for many applications.

5. Balanced Performance:

- While AVL trees guarantee stricter balance, Red-Black Trees provide a good balance between simplicity and performance, making them well-suited for a wide range of scenarios.

In summary, Red-Black Trees offer a more balanced solution, providing efficient search, insertion, and deletion operations with less stringent balance requirements and simpler rotations compared to AVL trees. The trade-off in strict balance guarantees is often justified by the ease of implementation and improved overall performance in practical applications.

(ii) What is a Binary Tree? Explain it. the preorder, in order and post order Traversals?

Binary Tree:

A 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. The topmost node in a binary tree is called the root, and nodes with no children are called leaves. The arrangement of nodes and their connections forms a tree-like structure, with each node having a parent (except the root) and zero, one, or two children.

Binary Tree Traversals:

Traversing a binary tree means visiting all the nodes of the tree and processing each node's data in a specific order. The three main methods for binary tree traversal are Preorder, Inorder, and Postorder.

1. Preorder Traversal:
- In a preorder traversal, the root node is visited first, followed by the left subtree and then the right subtree.
- The order of operations is Root -> Left -> Right.

```
Preorder: A -> B -> D -> E -> C -> F
```

2. Inorder Traversal:
- In an inorder traversal, the left subtree is visited first, followed by the root node, and then the right subtree.
- The order of operations is Left -> Root -> Right.

```
Inorder: D -> B -> E -> A -> F -> C
```

3. Postorder Traversal:
- In a postorder traversal, the left subtree is visited first, followed by the right subtree, and then the root node.
- The order of operations is Left -> Right -> Root.

```
Postorder: D -> E -> B -> F -> C -> A
```

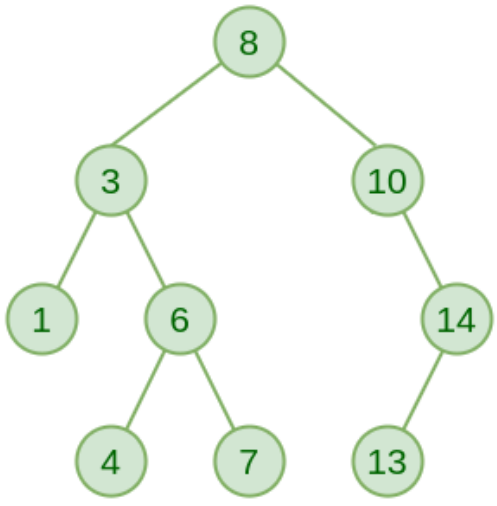
These traversals provide different perspectives on the structure of the binary tree and are often used in different applications based on the specific requirements of the problem at hand.

19.

(i) Explain about the Binary Search Tree? What are the rules to create a BST? Give an example

Binary Search Tree (BST):
Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree

Insertion Rule:

When inserting a new node, it must be placed in a location that preserves the ordering property of the BST. Starting from the root, traverse the tree to find the appropriate position based on the comparison of values.

Deletion Rule:

When deleting a node, there are three cases to consider:

Node with No Children (Leaf Node):

- Simply remove the node.

Node with One Child:

- Replace the node with its child.

Node with Two Children:

- Find the node's in-order successor (or predecessor) in the right (or left) subtree.
- Replace the node's value with the in-order successor's (or predecessor's) value.
- Delete the in-order successor (or predecessor) since it will have at most one child.

A Binary Search Tree (BST) is a binary tree data structure where each node has at most two children, and for every node, all elements in its left subtree are less than the node's value, and all elements in its right subtree are greater than the node's value. This ordering property makes searching, insertion, and deletion operations efficient.

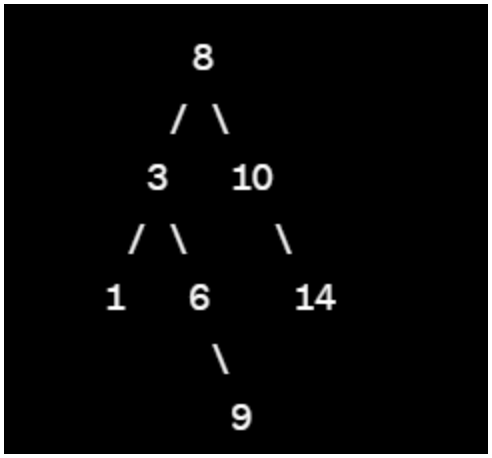
Rules to Create a BST:

1. Binary Tree Structure:
   - Each node has at most two children: left and right.

2. Ordering Property:
   - For each node:
     - All elements in the left subtree are less than the node's value.
     - All elements in the right subtree are greater than the node's value.

Example:

Let's create a simple BST with the following elements: 8, 3, 10, 1, 6, 9, 14.



- The root node is 8.
- For the left subtree of 8 (nodes 3, 1, 6), all elements are less than 8.
- For the right subtree of 8 (nodes 10, 9, 14), all elements are greater than 8.
- The ordering property holds for each node, making it a Binary Search Tree.

20.

(i) Write the code for Binary Search Tree Insertion.

```c
#include <stdio.h>
#include <stdlib.h>

// Node structure
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a node into BST
struct Node* insertNode(struct Node* root, int value) {
    // If the tree is empty, return a new node
```

```c
    if (root == NULL)
        return createNode(value);

    // Otherwise, recur down the tree
    if (value < root->key)
        root->left = insertNode(root->left, value);
    else if (value > root->key)
        root->right = insertNode(root->right, value);

    // Return the (unchanged) node pointer
    return root;
}

// Function to perform in-order traversal (for testing)
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->key);
        inOrderTraversal(root->right);
    }
}

// Driver program
int main() {
    struct Node* root = NULL;
    int keys[] = {8, 3, 10, 1, 6, 9, 14};

    // Insert keys into the BST
    for (int i = 0; i < sizeof(keys) / sizeof(keys[0]); i++) {
        root = insertNode(root, keys[i]);
    }

    // Print in-order traversal to check BST
    printf("In-order traversal: ");
    inOrderTraversal(root);

    return 0;
}
```

(ii) Explain B- trees and its operations?

B-Trees:

1. Definition:
- A self-balancing tree data structure that maintains sorted data and allows searches, insertions, and deletions in logarithmic time.

2. Node Structure:
- Each node can have multiple keys and children.
- Degree (order) of the tree defines the maximum number of children a node can have.

3. Balancing Property:
- All leaf nodes are at the same level.
- Every non-leaf node has at least two children.

4. Search Operation:
- Similar to binary search.
- Traverse nodes and keys based on comparison until the key is found or a leaf node is reached.

5. Insertion Operation:
- Insert the key in the appropriate leaf node.
- If the node overflows, split it, promote the median key, and update the parent.

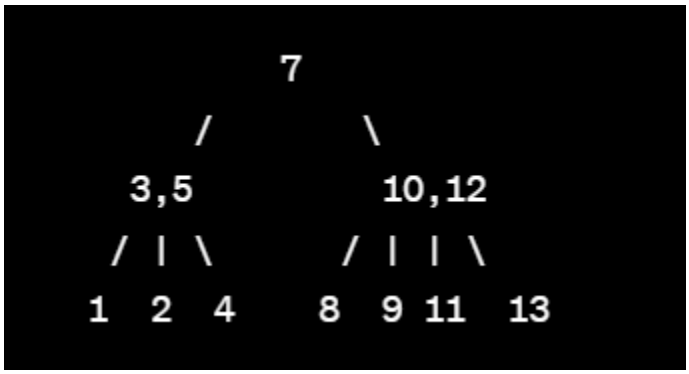
6. Deletion Operation:
- Delete the key from the leaf node.
- If the node underflows, borrow a key from a neighboring node or merge nodes.

7. Advantages:
- Efficient for large datasets and disk-based storage.
- Maintains balance, ensuring consistent performance.

8. Applications:
- Used in databases and file systems for indexing and storage.

Example B-Tree:

Consider a B-Tree of order 3:



In this B-Tree, each node can have up to 2 keys. The search, insertion, and deletion operations ensure balance and efficient access in logarithmic time.

21.

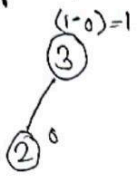
(i) start with an empty height balanced binary search tree (AVL) and insert the elements with following keys in the given order: 3,2,1,4, 5,6,7,15,16. At each step label all nodes with their balance factors and identify the rotation types.

Construct AVL Tree:
3, 2, 1, 4, 5, 6, 7, 15, 16

step 1:

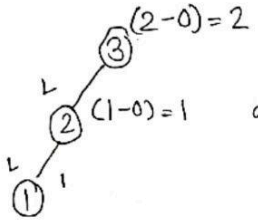
Insert 3

step 2:



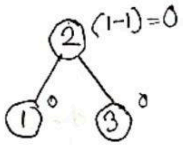
Insert 2

step 3:

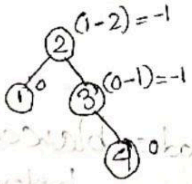


Insert 1

step 4:
LL unblanced Tree
clock wise Rotation

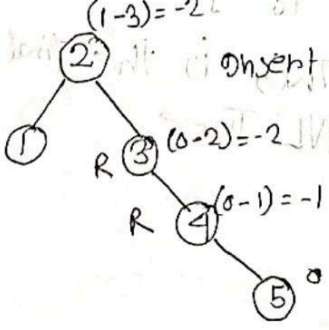


step 5:



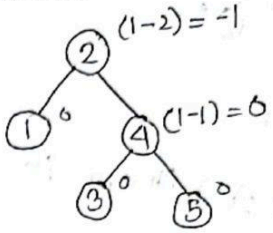
Insert 4

Step 6:

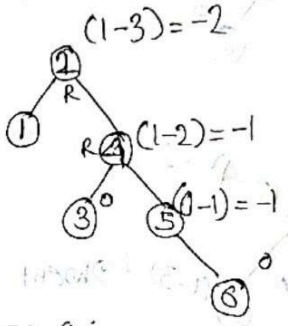


Insert 5

Step 7:
RR unblanced Tree
anticlock wise rotation

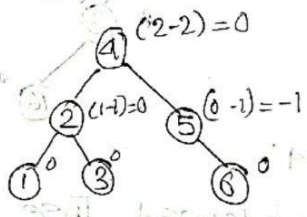


Step 8:

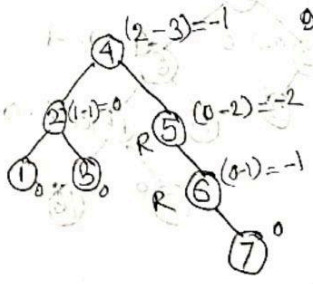


Insert 6

Step 9:
RR unblanced Tree
anticlockwise Rotation



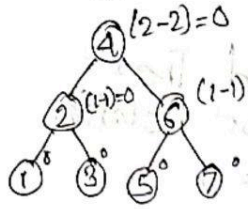
step 10:



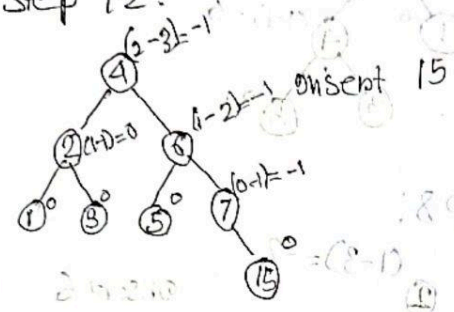
Insert 7

Step 11:

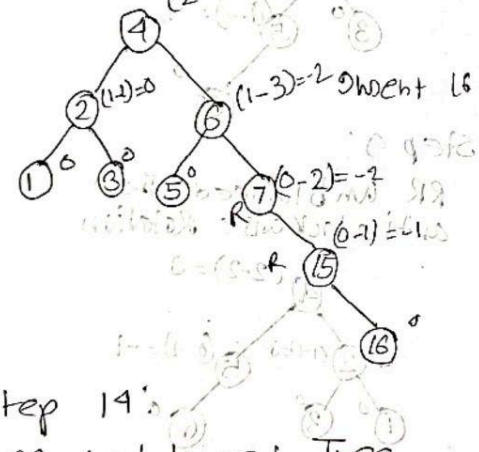
RR unblanced Tree
anti clock aise Rotation



step 12:

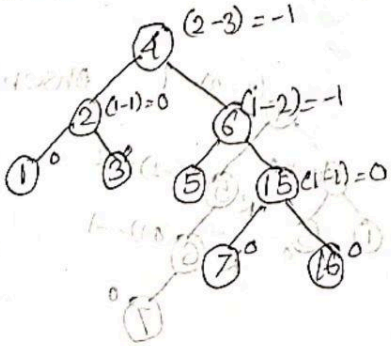


step 13:



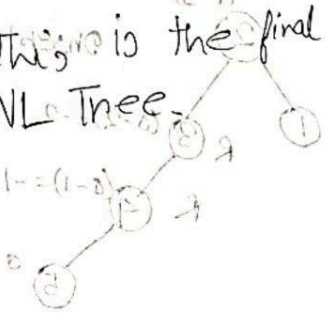
Step 14:

RR unb lanced Tree
anticlock aise Rotation



∴ so all node blanced factor range between -1 to 1

so, This is the final AVL Tree.



(ii) What is the difference between AVL tree and Splay tree?

Feature	AVL Trees	Splay Trees
Balance Maintenance	Strictly maintained through rotations after each insertion and deletion.	Achieves balance by restructuring the tree during access operations (splaying).
Performance	Better for scenarios with a mix of search, insert, and delete operations.	Efficient for repeated access patterns due to splaying. Good for sequences of accesses.
Insertion and Deletion	Potentially slower due to the need for more complex rotations.	Can be faster as splaying brings frequently accessed nodes closer to the root.
Memory Overhead	Typically requires an additional balance factor per node, resulting in higher memory overhead.	Less memory overhead since it doesn't explicitly store balance factors.
Access Patterns	Well-suited for scenarios where the workload involves a mix of operations.	Well-suited for scenarios where certain nodes are accessed more frequently than others.
Complexity	Generally more complex to implement due to strict balance requirements.	Simpler to implement as it relies on local restructuring during access.
Guarantees	Provides strict height balance guarantees, ensuring logarithmic height.	Guarantees amortized logarithmic time for access operations, but doesn't strictly maintain height balance.
Applications	Databases, file systems, scenarios with dynamic datasets.	Caching, scenarios with repeated access patterns, virtual memory systems.

(iii) What is computational geometry?

Computational Geometry:

- Definition: A branch of computer science and mathematics focusing on the study of algorithms and data structures to solve geometric problems.

- Objective: Develop efficient algorithms for geometric objects like points, lines, polygons, and solve problems in areas such as computer graphics, robotics, computer vision, and geographic information systems.

- Common Problems:
  - Convex Hull
  - Point Location
  - Intersection Testing
  - Triangulation
  - Closest Pair of Points
  - Voronoi Diagram
  - Delaunay Triangulation
  - Range Searching
  - Motion Planning

- Applications: Computer graphics, CAD, robotics, computer vision, GIS, and more.

- Key Focus: Designing algorithms to handle geometric data and structures for practical problem-solving.

(iv) Explain One Dimensional Range Searching with an example?

One-Dimensional Range Searching:

- Definition: A computational geometry problem that involves finding all points within a specified one-dimensional range on a line.

- Problem:
  - Given a set of points on a line and a query range [a, b], find all points that lie within the range [a, b].

- Example:
  - Suppose you have the following points on a line: [2, 5, 8, 12, 15, 18].
  - Query Range: [6, 14].
  - Result: Points within the range are [8, 12].

- Algorithm:
  - Use data structures like binary search trees or arrays to efficiently locate and retrieve points within the specified range.

- Applications:
  - Commonly used in databases, information retrieval systems, and spatial databases for efficient range queries in one-dimensional space.

22.

(i) Explain various computational geometry methods for efficiently solving the new evolving problem?

Certainly! Here are various computational geometry methods for efficiently solving geometric problems:

1. Convex Hull:
   - Methods: Gift wrapping, Graham's scan, QuickHull.
   - Applications: Finding the smallest convex polygon enclosing a set of points.

2. Voronoi Diagram:
   - Methods: Sweeping line algorithms, Fortune's algorithm.
   - Applications: Dividing a plane based on proximity to a set of points.

3. Delaunay Triangulation:
   - Methods: Incremental, divide and conquer.
   - Applications: Generating a triangulation with no points inside the circumcircle.

4. Line Segment Intersection:
   - Methods: Sweep line algorithm, Bentley–Ottmann algorithm.
   - Applications: Identifying intersections between line segments.

5. Range Searching:
   - Methods: Range trees, segment trees.
   - Applications: Searching for points within specified geometric ranges.

6. Closest Pair of Points:
   - Methods: Divide and conquer, incremental.
   - Applications: Identifying the pair with the smallest Euclidean distance.

7. Polygon Triangulation:
   - Methods: Seidel's algorithm, ear clipping.
   - Applications: Decomposing a polygon into triangles.

8. Geometric Graphs:
   - Methods: Planar graph algorithms, visibility graphs.
   - Applications: Analyzing relationships and connectivity in geometric spaces.

9. Motion Planning:
   - Methods: Visibility-based, probabilistic roadmaps (PRMs).
   - Applications: Planning the motion of robotic systems in geometric environments.

10. Randomized Algorithms:
    - Methods: Monte Carlo and Las Vegas algorithms.
    - Applications: Efficient handling of large-scale geometric datasets.

These methods cover a broad range of computational geometry problems, each tailored to specific types of geometric challenges.

(ii) Differentiate between Compressed Tries, Suffix Tries.

Feature	Compressed Tries	Suffix Tries
Purpose	Used for efficient storage of dictionaries and word lists.	Specifically designed for storing suffixes of a string.
Compression	Nodes with a single child are compressed into a single node.	No specific compression, each character is a distinct node.
Storage Efficiency	More efficient in terms of storage due to compression.	Less storage-efficient compared to compressed tries.
Construction	May involve additional steps for compression during construction.	Simpler construction as it directly represents suffixes.
Search Complexity	Search operations might require decompression, affecting speed.	Straightforward search operations without decompression.
Memory Overhead	Lower memory overhead due to compression.	Higher memory overhead as each character is a separate node.
Applications	Dictionaries, autocomplete, spelling suggestions.	Pattern matching, substring search, bioinformatics.

23.

(i) How Two Dimensional Range Searching done in computational geometry explain with an example

Here's a concise explanation of 2D range searching with examples:

Problem:

- Efficiently find all points within a rectangular query region in a 2D space.

Methods:

- Range Trees:
    - Balanced binary search tree.
    - Nodes represent vertical line segments.
    - Prune branches not intersecting the query range.
-
- K-D Trees:
    - Binary search tree that recursively partitions space.
    - Nodes represent hyperplanes dividing space.
    - Focus on subtrees intersecting the query range.
-

Example (Range Tree):

- Points: {(2,5), (4,1), (7,8), (1,3), (6,4)}
- Query range: [3,7] x [2,5]
- Tree:

```
      5
     / \
    2   7
   / \   \
```

```
    1   4    8
     \    \
      3    6
```

- Returns: (4,1), (2,5), (6,4)

Example (K-D Tree):

- Same points, vertical split at x=4
- Tree:

```
       4
      / \
     1   7
    / \   \
  (1,3)(2,5)(7,8)
      \
       (6,4)
```

- Returns: (2,5), (6,4)

Time Complexity:

- Range Trees: O(log n + k)
- K-D Trees: O(log^d n + k) (d = dimensionality)

Choosing a Method:

- Consider data distribution and query patterns.
- K-D Trees often better for higher dimensions.
- Range Trees simpler to implement.
- Other options: quadtrees, R-trees.

Two-Dimensional Range Searching:

Definition:

- A computational geometry problem involving the efficient retrieval of points or objects within a specified rectangular range in a two-dimensional space.

Problem:
- Given a set of points in a plane, perform queries to find all points within a specified rectangular range.

Example:
- Suppose we have a set of points in a 2D space: [(2, 4), (5, 7), (9, 2), (3, 8), (6, 5)].
- Query Range: Rectangle with corners (3, 3) and (7, 6).
- Result: Points within the range are (5, 7), (6, 5).

Algorithm:
- Use data structures like range trees, quad-trees, or kd-trees.
- Divide the space recursively into smaller regions.
- Store points efficiently in each region to speed up range queries.

Applications:
- Geographic Information Systems (GIS), databases, image processing.

24.

(i) Write a detailed note on the Huffman Coding Algorithm?

Huffman Coding Algorithm:

- Objective: Lossless data compression by assigning variable-length codes to input characters based on their frequencies.

- Steps:
  1. Frequency Counting: Determine the frequency of each character in the input.

  2. Priority Queue: Create a priority queue (min-heap) based on character frequencies.

  3. Huffman Tree Construction: Build a binary tree by repeatedly combining the two lowest-frequency nodes until a single node remains.

  4. Code Assignment: Assign binary codes (0 and 1) to edges, with shorter codes for higher frequency characters.

  5. Encoding: Replace characters in the input with their Huffman codes.

6. Decoding: Traverse the Huffman tree to reconstruct the original data from the encoded input.

- Properties:
  - Prefix Code: No code is a prefix of another, ensuring unambiguous decoding.

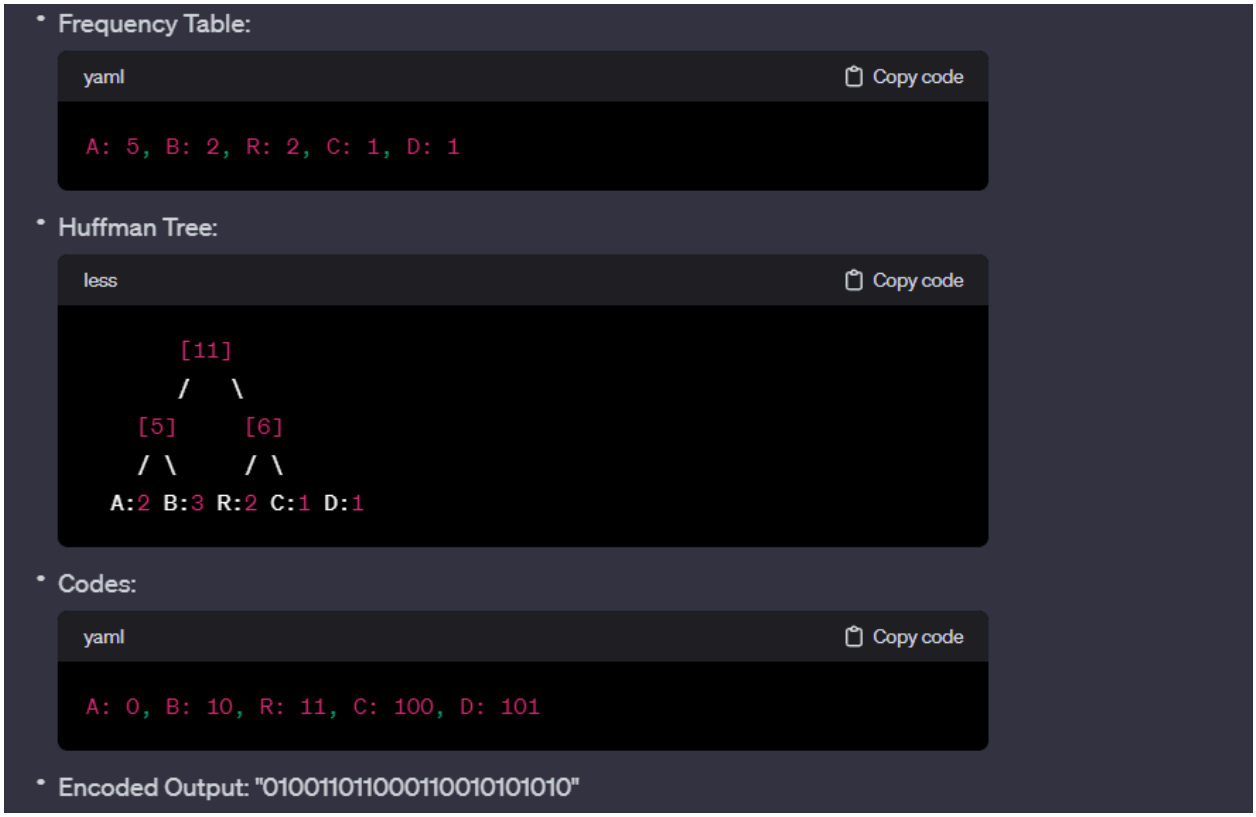
  - Optimal: Guarantees a minimum-length representation for the given character frequencies.

- Applications:
  - Text compression in file formats (e.g., ZIP files).

  - Image compression (e.g., JPEG).

  - Network protocols for data transmission efficiency.

- Example:
  - Input: "ABRACADABRA"



Note: The efficiency of Huffman coding comes from assigning shorter codes to more frequent characters, resulting in overall compression.

(ii) How to Apply Dynamic Programming to the LCS? Justify it with example

Dynamic Programming for Longest Common Subsequence (LCS):

Approach:

Dynamic Programming is commonly used to solve the LCS problem efficiently. The key idea is to build a table where each cell represents the length of the LCS of the corresponding substrings of the input sequences.

Steps:

1. Create a Table: Initialize a table with dimensions (m+1) x (n+1), where m and n are the lengths of the input sequences.

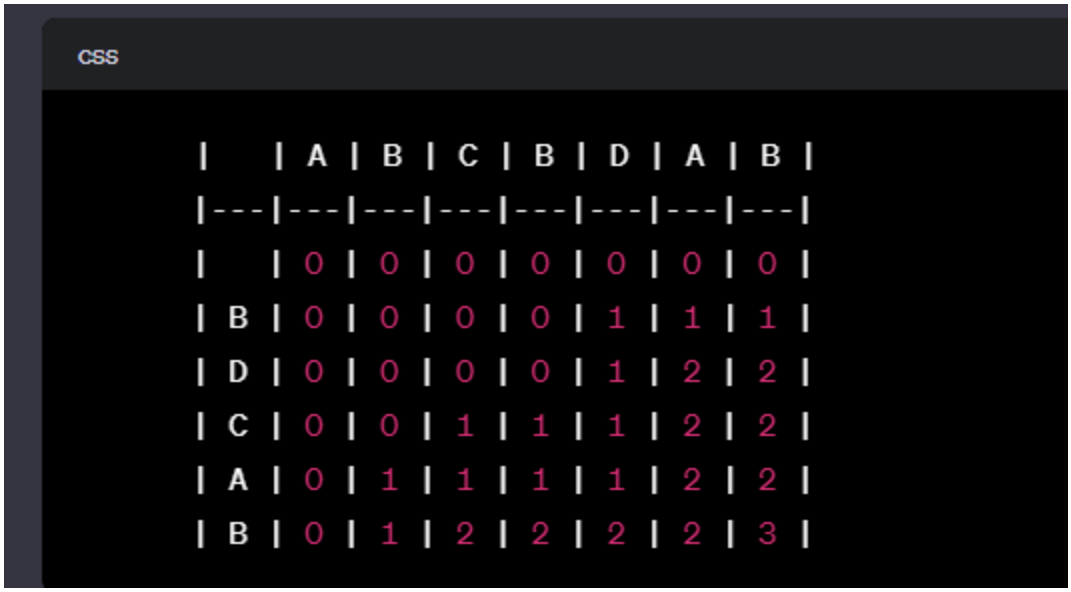
2. Base Cases: Set the values in the first row and first column to 0. These represent the LCS of an empty string with any other string, which is always 0.

3. Fill the Table: Traverse the table row by row, filling in each cell based on the following rules:
   - If `X[i-1] == Y[j-1]`, set `table[i][j] = table[i-1][j-1] + 1`.
   - Otherwise, set `table[i][j] = max(table[i-1][j], table[i][j-1])`.

4. Reconstruct LCS: Once the table is filled, backtrack from the bottom-right cell to reconstruct the actual LCS.

Example:

Consider the sequences `X = "ABCBDAB"` and `Y = "BDCAB"`. Let's construct the LCS table step by step:



- In the table, each cell `table[i][j]` represents the length of the LCS of the substrings `X[0...i-1]` and `Y[0...j-1]`.

- The bottom-right cell contains the length of the overall LCS, which is 3.

- Backtracking from the bottom-right cell (`table[6][6]`), we can reconstruct the LCS: "BCB".

This dynamic programming approach efficiently computes the LCS of two sequences in O(mn) time, where m and n are the lengths of the input sequences.

25.

(i) Discuss the working of Brute force pattern matching?

Brute Force Pattern Matching:

- Approach:
  - The simplest pattern matching algorithm.
  - Compares each character of the pattern with each character of the text.

- Steps:
  1. Start Matching:
     - Begin comparing the pattern with the text from the first character.

  2. Shift and Compare:
     - Move one position at a time in the text.

   - Compare the entire pattern with the corresponding substring in the text.

  3. Matching Condition:
   - If a mismatch is found, shift the pattern by one position and continue.

  4. Repeat:
   - Continue until the entire text is processed.

- Example:
  - Text: "ABABCABAB"
  - Pattern: "ABAB"

  ```
  Step 1: ABABCABAB (Match)
  Step 2: ABABCABAB (Mismatch)
  Step 3: ABABCABAB (Mismatch)
  Step 4: ABABCABAB (Match)
  ```

- Complexity:
  - Time Complexity: O((n-m+1) m), where n is the length of the text and m is the length of the pattern.
  - Inefficient for large texts and patterns.

- Advantages:
  - Simple and easy to implement.
  - Works for any pattern and text.

- Disadvantages:
  - Inefficient for large datasets.
  - Time complexity can be high in worst-case scenarios.

Brute Force pattern matching is straightforward but may not be suitable for large-scale applications due to its inefficiency. It serves as a baseline algorithm for pattern matching.

(ii) How Compressed Tries work? Explain its operations?

Compressed Tries:

- Definition:
  - A data structure that represents a trie (tree-like structure for storing a dynamic set or associative array) with compression to reduce storage space.

- Operations:

  1. Insertion:
     - Traverse the trie, adding nodes as needed.
     - Identify common prefixes and compress them into a single node.

  2. Search:
     - Traverse the trie following the path defined by the input key.
     - Check for compressed nodes and decompress if needed.
     - Verify the presence of the key.

  3. Deletion:
     - Remove the node representing the key.
     - Adjust the trie structure, potentially decompressing nodes.

  4. Compression:
     - Identify nodes with a single child.
     - Merge them into a single node, storing the common prefix.

  5. Decompression:
     - If necessary during search or delete operations, decompress nodes by expanding the stored prefix.

- Example:

  Suppose we have the keys "bat," "batman," "batwoman," and "batter."

  ```
        Root
         |
         b
         |
         a
       / | \
      t  n  w
     / \
    t   e
   / \
  e   r
  ```

  After compression:

  ```
  ```

```
        Root
         |
         b
         |
         a
       / | \
      t  n  w
       \
        e
        |
        r
```

- Advantages:
  - Space-efficient for storing large sets of keys with common prefixes.
  - Reduces memory overhead compared to an uncompressed trie.

- Disadvantages:
  - More complex to implement than a standard trie.
  - May incur overhead during search and delete operations due to decompression.

Compressed tries provide a balance between space efficiency and trie-based data storage, particularly useful when dealing with large datasets with shared prefixes among keys.

(iii) Explain Standard Tries with an example?

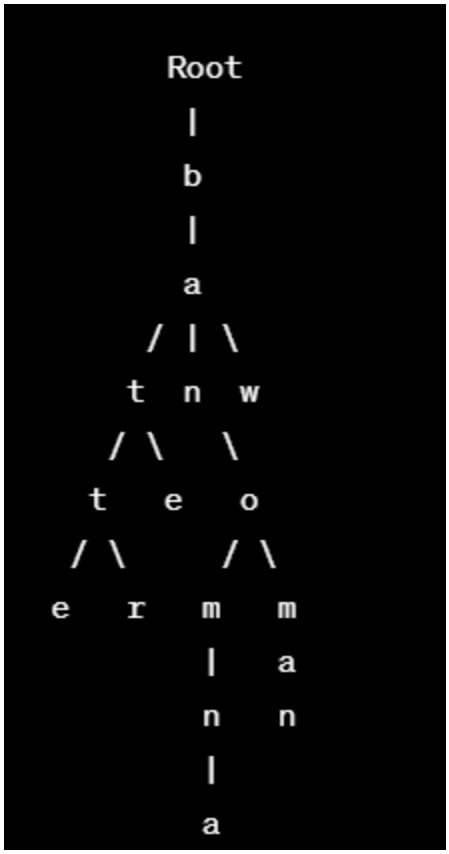
Standard Tries:

- Definition:
  - Tree-like data structure used for storing a dynamic set or associative array, commonly employed for string keys.

- Nodes:
  - Each node corresponds to a single character.
  - Nodes are connected to form paths from the root to specific keys.

- Example:

  Suppose we have keys "bat," "batman," "batwoman," and "batter."



- Operations:

  1. Insertion:
     - Traverse the trie, adding nodes for each character.
     - Create new nodes as necessary.

  2. Search:
     - Traverse the trie following the path defined by the input key.
     - Check for the presence of the key.

  3. Deletion:
     - Remove the node representing the key.
     - Adjust the trie structure accordingly.

- Advantages:
  - Simple to implement.
  - Efficient for search operations.

- Disadvantages:
  - May have high memory overhead, especially when storing keys with similar prefixes.

- Inefficient for certain queries involving keys with common prefixes.

Standard tries are a fundamental data structure for key-based storage, but they can become inefficient when dealing with datasets containing keys with shared prefixes.

26.

(i) Describe The Knuth-Morris- Pattern Algorithm?

https://www.javatpoint.com/daa-knuth-morris-pratt-algorithm
Knuth-Morris-Pratt (KMP) Algorithm:

- Objective:
  - Efficient pattern matching algorithm to find occurrences of a pattern within a text.

- Key Idea:
  - Utilizes information from previous matches to avoid unnecessary character comparisons.

- Steps:

  1. Preprocessing (Compute LPS):
     - Create a Longest Prefix Suffix (LPS) array for the pattern.
     - LPS[i] represents the length of the longest proper prefix which is also a proper suffix of the substring pattern[0...i].

  2. Pattern Matching (Search):
     - Traverse the text and pattern simultaneously.
     - If a mismatch occurs at position j in the pattern, shift the pattern by LPS[j-1] positions, skipping unnecessary comparisons.

- Example:

  Text: "ABABDABACDABABCABAB"
  Pattern: "ABABCABAB"

  ```
  Preprocessing (LPS Array):
  A B A B C A B A B
  0 0 1 2 0 1 2 3 4

  Pattern Matching:
  ```

```
A B A B C A B A B
A B A B C A B A B (Match)
```

- Time Complexity:
  - O(m + n), where m is the length of the pattern and n is the length of the text.

- Advantages:
  - Avoids unnecessary character comparisons, improving efficiency.

- Applications:
  - String matching, text processing, bioinformatics.

The Knuth-Morris-Pratt algorithm is particularly useful when searching for occurrences of a pattern in a large text, providing efficient performance due to its clever preprocessing step.

(ii) Discuss the function Suffix Tries with an example?

Suffix Tries:

- Definition:
  - A tree-like data structure that represents all suffixes of a given string.

- Nodes:
  - Each node corresponds to a character.
  - Edges represent the characters of the string.

- Example:

  Consider the string "BANANA."

- Construction:
  - Insert all suffixes of the string into the trie.

- Operations:

  1. Insertion:
     - Traverse the trie, adding nodes for each character in the suffix.
     - Create new nodes as necessary.

  2. Search:
     - Traverse the trie following the path defined by the input suffix.
     - Check for the presence of the suffix.

  3. Deletion:
     - Remove the nodes representing the suffix.
     - Adjust the trie structure accordingly.

- Advantages:
  - Efficient for pattern matching and substring searches.
  - Allows quick determination of common substrings.

- Applications:
  - String algorithms, bioinformatics, data compression.

Suffix Tries provide a compact representation of all suffixes of a string, facilitating efficient substring searches and pattern matching. They are commonly used in various applications where substring information is crucial.

27.

(i) Write about The Boyer- Moore Algorithm?

Boyer-Moore Algorithm:

- Objective:
  - Efficient string searching algorithm, particularly effective for searching in large texts.

- Key Ideas:
  1. Bad Character Rule:
     - If a mismatch occurs, align the pattern with the rightmost occurrence of the mismatched character in the text.
  2. Good Suffix Rule:
     - Align the pattern based on the longest suffix of the pattern that matches a suffix of the text.

- Preprocessing:
  - Construct two tables for the pattern:
    1. Bad Character Shift Table: Records the rightmost occurrence of each character in the pattern.
    2. Good Suffix Shift Table: Records the shift amount based on matching suffixes.

- Steps:

  1. Start Matching:
     - Align the pattern with the beginning of the text.

  2. Compare Right to Left:
     - Start comparing characters from the rightmost side of the pattern to the leftmost side.

  3. Mismatch Handling:
     - Use the Bad Character and Good Suffix rules to determine the shift amount.

  4. Shift and Continue:
     - Shift the pattern to the right by the calculated amount.
     - Continue until a match is found or the end of the text is reached.

- Example:

Text: "THIS IS A TEST TEXT"
Pattern: "TEST"

```
Bad Character Shift Table:
T: 1, E: 2, S: 3

Good Suffix Shift Table:
EST: 3, ST: 3, T: 4
```

```
Step 1: THIS IS A TEST TEXT
            TEST  (Match)
```

- Advantages:
  - Fastest string searching algorithm in practice.
  - Efficient for large texts.

- Applications:
  - Text editors, search engines, data processing.

The Boyer-Moore algorithm is widely used for efficient substring searching, especially in scenarios where the text is large. Its combination of the Bad Character and Good Suffix rules contributes to its effectiveness.

(ii) Define Tires with example . Explain k-D Trees with an example?

Tries (Prefix Trees):

- Definition:
  - A tree-like data structure used to store a dynamic set or associative array where keys are usually strings.

- Nodes:
  - Each node represents a character.
  - Edges represent the characters of the keys.

- Example:

Suppose we have keys "bat," "batman," "batwoman," and "batter."

```
    Root
    |
    b
    |
    a
  / | \
 t  n  w
/ \
e  r
```

k-D Trees (k-Dimensional Trees):

- Definition:
  - A space-partitioning data structure for organizing points in a k-dimensional space.

- Nodes:
  - Each node represents a point in k-dimensional space.
  - Splitting planes divide the space.

- Example:

  Consider points (2,3), (5,4), (9,6), (4,7), (8,1).

```
      (5,4)
     /     \
(2,3)       (9,6)
   \         /
  (4,7)   (8,1)
```

- Operations:

  1. Insertion:
     - Recursive traversal based on comparisons in each dimension.

  2. Search:
     - Navigate the tree based on comparisons in each dimension.

  3. Deletion:

- Remove the node and adjust the tree structure.

- Advantages:
  - Efficient for multidimensional spatial data.
  - Enables quick spatial searches.

- Applications:
  - Database systems, computational geometry, machine learning.

Tries are used for string key-based storage, while k-D Trees are used for efficient spatial data organization in multidimensional spaces. Both data structures serve distinct purposes in different scenarios.

28.

(i) Describe Quad trees and its functions?

Quad Tree
Quad Trees:

- Definition:
  - A tree data structure in which each internal node has exactly four children, representing four quadrants of a 2D space.

- Functions:

  1. Spatial Partitioning:
     - Divides a 2D space into four quadrants.

  2. Representation:
     - Efficiently represents spatial data and coordinates.

  3. Insertion:
     - Inserts a point into the quad tree based on its coordinates.

  4. Search:
     - Locates points or regions efficiently by traversing the tree.

  5. Nearest Neighbor Search:
     - Facilitates finding the nearest neighbor of a given point.

  6. Range Queries:

- Allows searching for points within a specified rectangular region.

- Example:

  Consider points (2,3), (5,4), (9,6), (4,7), (8,1) in a 2D space.

  ```
        (5,4)
       /     \
  (2,3)       (9,6)
     \         /
    (4,7)   (8,1)
  ```

- Advantages:
  - Efficient for spatial data representation and queries.
  - Useful in computer graphics, geographical information systems (GIS).

- Applications:
  - Image processing, collision detection, geographical mapping.

Quad trees provide an efficient way to organize and search spatial data in a 2D space, particularly in applications where spatial relationships are essential.

(ii) How to construct a Priority Search tree? Explain with neat diagram

Constructing a Priority Search Tree (PST) involves recursively partitioning the input points based on their x-coordinates and y-coordinates. Below are the steps to construct a Priority Search Tree with a neat diagram:

1. Sort Points:
   - Sort the input points based on their x-coordinates. If two points have the same x-coordinate, sort them based on their y-coordinates.

2. Build the PST:
   - Recursively build the PST by following these steps:

     a. Choose a Splitting Line:
        - Select the median x-coordinate from the sorted points.

     b. Split Points:

- Partition the points into two sets based on the chosen splitting line. All points to the left have x-coordinates less than the splitting line, and those to the right have x-coordinates greater than the splitting line.

c. Recursive Construction:
- Recursively construct PSTs for the points to the left and right of the splitting line.

d. Construct Vertical Decomposition:
- For each node in the PST, construct a vertical decomposition of the points based on their y-coordinates. This forms a binary search tree for the y-coordinates.

3. Example Diagram:

Let's consider a set of points: (3, 1), (6, 5), (7, 8), (9, 3), (12, 6), (15, 10), (18, 12).

a. Sort points based on x-coordinates: (3, 1), (6, 5), (7, 8), (9, 3), (12, 6), (15, 10), (18, 12).

b. Choose the median x-coordinate, which is 9.

c. Split the points into two sets based on x-coordinates: Points to the left - (3, 1), (6, 5), (7, 8), (9, 3); Points to the right - (12, 6), (15, 10), (18, 12).

d. Recursively construct PSTs for both sets.

Repeat the process for the left and right nodes until all points are considered.

The resulting Priority Search Tree will have a structure where each internal node represents a splitting line and has a vertical decomposition of points based on their y-coordinates.

```plaintext
                (9, 3)
               /      \
          (6, 5)        (15, 10)
         /     |        /          \
    (3, 1) (7, 8)    (12, 6)    (18, 12)
```

This example demonstrates the basic steps for constructing a Priority Search Tree with a simple set of points. The actual tree structure may vary depending on the specific points and their coordinates.

(iii) What is Priority Range Trees discuss with an example?

Priority Range Trees:

- Definition:
  - Extension of Range Trees that incorporates priority values with 2D points for efficient range queries.

- Key Concepts:
  1. Range Trees for spatial dimensions.
  2. Priority values associated with points.
  3. Priority Range Queries consider both spatial and priority dimensions.

- Example:
  - Points: (2,3)[5], (4,7)[2], (6,5)[8], (9,4)[1], (12,9)[4].

- Construction Steps:
  1. Sort by x-coordinates.
  2. Build 1D Range Tree for priorities.
  3. Build 2D Range Tree for spatial coordinates.
  4. Combine both trees.

- Querying:
  - Specify ranges in spatial dimensions and priority values.

- Use Cases:
  - Efficient querying of points based on both spatial and priority criteria.

29.

(i) What is computational geometry?

Computational Geometry:

- Definition:
  - Branch of computer science and mathematics that focuses on the study of algorithms and data structures for solving geometric problems.

- Key Aspects:
  - Involves the analysis and design of efficient algorithms for geometric objects and spatial data.
  - Applications in areas like computer graphics, computer-aided design, robotics, and geographic information systems (GIS).

- Examples of Problems:
  - Convex hull computation, nearest neighbor search, intersection detection, and geometric optimization.

- Purpose:
  - Addresses problems related to the manipulation, analysis, and representation of geometric objects in a computational setting.

- Significance:
  - Essential in various fields where spatial relationships and geometric structures play a crucial role in problem-solving and decision-making.

(ii) What is Priority Range Trees discuss with an example?

Priority Range Trees:

- Definition:
  - Extension of Range Trees that efficiently handles range queries on 2D points with associated priority values.

- Key Features:
  1. Incorporates priority values with spatial coordinates.
  2. Enables range queries considering both spatial and priority dimensions.

- Construction Steps:
  1. Sort points by x-coordinates.
  2. Build a 1D Range Tree for priority values.
  3. Build a 2D Range Tree for spatial coordinates.
  4. Combine both trees.

- Example:
  - Points: (2,3)[5], (4,7)[2], (6,5)[8], (9,4)[1], (12,9)[4].

- Querying:
  - Perform efficient range queries with conditions on both spatial coordinates and priority values.

- Use Cases:
  - Efficiently handle queries that involve both spatial and priority dimensions for applications like spatial databases or GIS.

30.

(i) Explain how to Search a Priority Search Tree works and its operations?

Priority Search Tree (PST) Search:

- Objective:
  - Efficiently search for points in a Priority Search Tree based on spatial and priority criteria.

- Operations:

  1. Spatial Range Query:
     - Use the 2D Range Tree to find points within a specified spatial rectangle.

  2. Priority Range Query:
     - Utilize the 1D Range Tree for priority values to filter points within a specified priority range.

  3. Combined Result:
     - The final result is the set of points satisfying both spatial and priority conditions.

- Example:
  - Given points (2,3)[5], (4,7)[2], (6,5)[8], (9,4)[1], (12,9)[4].

- Search Process:
  1. Perform a spatial range query to find points within a specified spatial rectangle.
  2. Use the 1D Range Tree to filter points based on priority values.
  3. Obtain the final set of points meeting both spatial and priority criteria.

- Efficiency:
  - Achieves efficient searching by leveraging both spatial and priority dimensions.

- Applications:
  - Suitable for applications requiring queries based on spatial and priority conditions, such as spatial databases or GIS.