**Sets & Dictionaries**

#### Q1: Hash Function

The input and output of a hash function are as follows:

Ans: Takes keys as input and gives array indices as output

**✓ Correct**

**Feedback:**

A hash function should be interpreted as a black box that takes keys as input and gives array indices as output.

#### Q2: Drawbacks of Hashing

In a scenario, wherein multiple keys gets hashed to the same index, you have to discard one and keep the other. This behaviour is particularly undesired because you may end up discarding information that you may need to retrieve later. According to you, what could be a possible way to accommodate keys that face this type of situation?

* Ans: Hash is inefficient when there are many collisions.
* Hash collisions are practically not be avoided for large set of possible keys.
* Hash does not allow null values.

#### Q3: Hashtable

Which of the following is the correct definition of HashTables?

Hint: Recall what hash tables stores and how they are related.

Ans: A data structure that maps keys to values

**✓ Correct**

**Feedback:**

A HashTable is a data structure that implements an associative array containing a pair of key and value. Therefore, the hashtable is a data structure that maps keys to values.

#### Q4: Collision

What happens in a ‘collision’ in hashing functions?

Ans: The hash function maps several different elements to the same hash value.

**✓ Correct**

**Feedback:**

If several different elements map to the same hash value using the hash function, then this event is called a ‘collision’.

#### Q5: Hash Functions with Collisions

Consider a hash table of size 10, with a starting index of zero, and a hash function: (5x + 4) mod 10. Assuming that the hash table is initially empty, the sequence 1, 3, 8, 10 is inserted into it. What will be the indices at 4 and 9 of the hash table, respectively, if you replace the already existing values at that hash index?

Ans:   
10, 3

**✓ Correct**

**Feedback:**

1 gets hashed to (5 \* 1 + 4)mod10 = 9mod10 = 9. Then, 3 comes into the sequence and gets hashed to (5 \* 3 + 4)mod10 = 19mod10 = 9. So, 3 replaces 1 in the table and sits at the 9th index. Something similar happens to 8 and 10.

#### Q6: Chaining

A hash function uses NAME as a key for storing contacts in a phone book. First, it converts NAME into an integer by adding up the digits’ values in the alphabetical order (A-1,B-2,....Z-26) and performing a mod 100 operation on the sum.

Also, if there is a collision (different keys hashed to the same index), the function resolves it via chaining, i.e., putting keys that hash to the same index in a list (i.e., a linked list).

What would the indices be of the names ‘ROSS’ (R-18,O-15,S-19,S-19) and ‘SORS’ if they are inserted in the specified order?

Ans: 71 and 71

**✓ Correct**

**Feedback:**

Both ‘ROSS’ and ‘SORS’ have the same characters but are simply ordered differently. Therefore, both names will get hashed to the index 71. Specifically, ROSS will be added as the head element in a list stored at index 71 first. After this, SORS will also be added to the list stored at index 71. However, you will now store SORS as the head of the list and make ROSS the element that comes after it.

#### Q7: Time Complexity With Chaining

The search operation in a hash table that employs chaining is O(n). From your perspective, can you justify why this operation will have a run-time of O(n) when the hash table uses chaining?

#### Ans: At each index, there is a list maintained to store entries with the same hash values. So, you have to traverse a list up to the last element in some cases. This can take an O(n) amount of time in the worst case.

#### Q8: Time Complexity With Chaining

What is the run-time complexity of an add operation in a hash table that uses chaining for collision resolution?

Hint: The add operation in a hash table means adding the new element to the head of the linked list, this way hash table maintains.

Ans: O(1)

**✓ Correct**

**Feedback:**

At each index, there is a list maintained to store entries with the same hash values. However, since you are adding the new entry to the head of the linked list, it will be an O(1) operation.

#### Q9: Chaining

Which data structure is suitable for simple chaining in hash tables?

Hint: Take into consideration the delete operation of each data structure in the options.

Ans: Doubly linked list

**✓ Correct**

**Feedback:**

The delete method of hash with chaining takes the element of a linked list as the input. For example, in a singly linked list (a ←> x <→ k <→ y <→ b), if you are at node ‘k’, to delete node ‘k’, you need to link node ‘x’ to ‘y’. For that, you need to get access to node ‘x’, which is the previous node of node ‘k’. You can get the access to node 'x' from node ‘k’ itself, and then, you can link node ‘x’ to node ‘y’, which takes O(1) time.  
A doubly linked list makes deleting an element easier.

# How a Hash Function Affects Hashing

#### Q10: A Bad Hash Function

Consider the hash function H, which takes ‘name’ as a key as follows:

H(name)=(pos(firstChar(name))%2 (as in the video).

At what index would the name Ross be hashed by this function? (Remember that the index will be 0-based.)

Ans: 1

**✓ Correct**

**Feedback:**

R appears at the 18th position. So, considering 0-based indexing, you get R as 17, and taking mod with 2, you get 1.

#### Q11: A Bad Hash Function

Consider the hash function H, which takes ‘name’ as a key. Suppose the hash is calculated as H(name)=pos(firstChar(name))%2 as in the video. Why is this hash function a bad hash function?

Ans: As you can clearly see, the hash function only fills up two of the slots of the hash table, which are 0 and 1, leaving the rest of it completely empty. Therefore, this function does not utilise the available space well and, hence, clearly is not a good choice.

There is no single hash function that is universally applicable. For a given application, a good hash function should be designed with the following characteristics in mind:

1. The hash function should use all the keys.
2. The hash function should distribute the keys uniformly across the array indices.
3. The hash function should output different hash values for similar yet unequal keys.

#### Q12: Hash Function

Consider a hash function h = [m\*((k \* A) mod 1)], where

1. k is the number to be hashed,
2. m is the number of keys or is the hash table size, and
3. A lies between 0 and 1.

[x] represents the greatest integer not exceeding x, and y mod 1 gives the fractional part of y.

Using the hash function, what value will the key 612 be hashed to if A = 0.5 and the hash table’s size is 500?

Hint: Here, k = 612.

Ans: m = 500//hash table size

k = 612

A = 0.5

h = [500\*((612 \* 0.5) mod 1) = (500\*(306 mod 1)) = 500 \* 0 = 0

#### Q13: Hash Function

Consider a hash function h = [m\*((k \* A) mod 1)], where

1. k is the number to be hashed,
2. m is the number of keys or is the hash table size, and
3. A lies between 0 and 1.

[x] represents the greatest integer not exceeding x, and x mod 1 gives the fractional part of x.

Using the hash function, where will the key 611 be hashed if A = 0.5 and the hash table’s size is 500?

Hint: Here k = 611.

Ans: m = 500//hash table size

k = 611

A = 0.5

h = [500\*((611 \* 0.5) mod 1) = (500\*(305.5 mod 1)) = 500 \* 0.5 = 250

#### Q15: Hash Function

Consider a hash function h = [m\*((k \* A) mod 1)], where

1. k is the number to be hashed,
2. m is the number of keys or is the hash table size, and
3. A lies between 0 and 1.

[x] represents the greatest integer not exceeding x, and y mod 1 gives the fractional part of y.

What is the total number of indices being used by the hash function if A = 0.5 and the hash table’s size is 500?

Hint: Consider the previous two questions and determine the answer in terms of even and odd numbers.

Ans: As you saw in the previous two problems, wherein 612 got hashed to index 0 and 611 got hashed

to 250, you can check other numbers as well, which will show you that even numbers get mapped to 0, and odd numbers get mapped to 250. Doing this will give you ‘two’ indices as the answer.

#### Q16: Hash Function

Consider a hash function h = [m\*((k \* A) mod 1)], where

1. k is the number to be hashed,
2. m is the number of keys or is the hash table size, and
3. A lies between 0 and 1.

[x] represents the greatest integer not exceeding x, and y mod 1 gives the fractional part of y.

Is this hash function good or bad?

Ans: This hash function is bad because only two indices, i.e., 0 and 250, of the hash table are being used to store values, and the remaining 498 are wasted. Many values end up being hashed to the same two indices, and therefore, a lot of the hash table’s memory unused and, hence, also goes to waste.

#### Q17: Hash Function

Suppose you are given a hashtable of size 5, with starting index 0 and a hash function **(2x + 3) mod 5**. How does the hashtable look after inserting 2, 3, 4 and 6 to it using open hashing?

Example:

* Hashtable ‘x, \_, y’ implies:
  + x is at index 0 of hashtable
  + ‘\_’ means index 1 is empty
  + y is at index 2 of hashtable

Note:

* ‘\_’ indicates that the slot is empty in the hash table.

Ans: 6, 4, 2, \_, 3

**✓ Correct**

**Feedback:**

1. For element 2, the value of the hash function (2x + 3) mod 5 is 2. So, insert the value at index 2.
2. For element 3, the value of the hash function (2x + 3) mod 5 is 4. So, insert the value at index 4.
3. For element 4, the value of the hash function (2x + 3) mod 5 is 1. So, insert the value at index 1.
4. For element 6, the value of the hash function (2x + 3) mod 0 is 4. So, insert the value at index 0.

Therefore, after inserting the elements, the hash table looks like **6, 4, 2, \_, 3.**

Note: A hash table is a data structure that is designed to reduce the memory requirement of direct addressing. It is an array coupled with a hash function. For a given application, a hash function is designed to take the keys as input and provide array indices as output. The provider decides the size of the array or hash table according to the number of records to be stored.

The chosen size of a hash table cannot be extremely small because it will result in a collision. A 'collision' is a phenomenon in which two distinct keys get hashed to the same index. Typically, the size of a hash table is taken to be twice the number of records to be stored. The frequency of collision depends on the design of the hash function. A good hash function uniformly distributes all the keys in the hash table and reduces collision.

The problem of collision in hash tables can be solved using several methods such as chaining and open addressing. Chaining refers to replacing array indices with linked lists.

Java provides two APIs for implementing hash tables, which are Hashtable and HashMap. They are extremely similar in terms of functionality, except that Hashtable does not support null values, and HashMap is not synchronised. For a given application, either of these APIs may be chosen. Java further supports generics that allow a developer to implement dictionaries to store and retrieve any type of key-value pair using hash tables.

For example: If we have a multi-threaded application and more than one thread is trying to access a shared variable, then it is our obvious need to not let multiple threads access the same variable at the same instant. So, to avoid this situation of the same variable being accessed by the multiple threads, HashTable() is used. If synchronisation is not needed, HashMap() can be used.

Additional Reading Links for Hashtable and HashMap in Java API are provided below:

* [Hashtable in Java](https://www.geeksforgeeks.org/java-util-hashtable-class-java/)
* [HashMap in Java](https://www.geeksforgeeks.org/java-util-hashmap-in-java/)
* [Open Addressing in Hashing](https://www.geeksforgeeks.org/hashing-set-3-open-addressing/)

#### Q19: The Airline Boarding-Pass Problem

In an airline ticketing system, they issue unique boarding passes to each of the passengers of a particular flight. Each boarding pass is scanned at the gate to verify that it has not already been used. The system looks into the database and checks whether any boarding pass is already in use. It assumes that the person who checked in first has the original ticket. So, any time a duplicate boarding pass is encountered, the system sends the following message: 'This is a duplicate boarding pass.'

Now, the currently running system makes use of the conventional method of searching an array for a boarding pass, which obviously takes time.

According to you, what would be the run-time complexity of this system, assuming that the boarding passes are added to the array in a random order?

Hint: Consider the searching algorithm that would perform well for the unsorted array–the one that was covered in module Searching and Sorting in Session Searching.

Ans: O(n)

**✓ Correct**

**Feedback:**

Since the array is unordered, you can, at best, apply a naive linear search. As it is known that a linear search in an array takes O(n) time to completely traverse the array, in the worst case, the time complexity will turn out to be O(n).

#### Q20: The Airline Boarding-Pass Problem

In an airline ticketing system, they issue unique boarding passes to each of the passengers of a particular flight. Each boarding pass is scanned at the gate to verify that it has not already been used. The system looks into the database and checks whether any boarding pass is already in use. It assumes that the person who checked in first has the original ticket. So, any time a duplicate boarding pass is encountered, the system sends the following message: 'This is a duplicate boarding pass.'

Now, the already running system makes use of the conventional method of searching an array for a boarding pass, which obviously takes O(n) or O(log n) time, depending on whether you can use naive linear search or binary search on the array.

Can you suggest a way in which the system can be designed to improve the overall add and retrieve times, probably in the constant time order?

Ans: The system can be designed with the help of a hash table to which the boarding pass numbers can be added as keys and can then be checked for existence in the table; this would take up a constant amount of time or O(1) for both the operations.

#### Q22: Disadvantages of Chaining

Which of these are possible disadvantages of chaining? (Note: More than one option may be correct.)

Hint: Consider the growing size of the Hash Table.

Ans: Linked lists could grow large in size; this would increase the overall time complexity of the search operation involved.

**✓ Correct**

**Feedback:**

An increasing number of elements being added to the hash table results in more elements getting added to the linked lists at particular indices, which results in them getting large in size.



More memory is required to represent a linked list because longer linked lists require more pointers to represent their underlying structures.

**✓ Correct**

**Feedback:**

Since chaining requires maintaining a linked list at each index, and as a linked list is maintained internally with the use of pointers, which takes up extra memory, chaining leads to the consumption of extra memory.

#### Q23: Hash Function

Which of the following is true about hash functions? (Note: More than one option may be correct.)

Ans: A hash function is any function that can be used to map data of an arbitrary size to data of a fixed size.

**✓ Correct**You missed this!

**Feedback:**

A hash function maps data from a larger domain to a smaller range, thus lowering the requirement of data bytes and leading to a number of keys being possibly hashed to the same index.

The domain of a hash function (the set of possible keys) is larger than its range (the number of different table indices).

**✓ Correct**

**Feedback:**

A hash function maps data from a larger domain to a smaller range, thus lowering the requirement of data bytes and leading to a number of keys being possibly hashed to the same index.



A hash function may give the same hash value for distinct keys.

**✓ Correct**You missed this!

**Feedback:**

A hash function maps data from a larger domain to a smaller range, thus lowering the requirement of data bytes and leading to a number of keys being possibly hashed to the same index.

# Introduction to HashMap

The similarities between Hashtable and HashMap are as follows:

1. Both are the implementations of the Map interface in Java.
2. Both of them perform similar functions.
3. Neither of them maintain any order of elements.

Differences between Hashtable and HashMap are as follows:

|  |  |
| --- | --- |
| **Hashtable** | **HashMap** |
| It is used in the older versions of Java. | It exists only in newer versions of Java, i.e., it is part of Java since version 1.2 |
| It does not allow a key to be null. | It allows at the most one key to be null. |
| It does not allow to store a null value. | It allows storage of any number of null values. |
| It is a bit slower. | It is faster. |

To know more about the differences between Hashtable and HashMap, please refer to [this](https://stackoverflow.com/questions/40471/differences-between-hashmap-and-hashtable) link.

By now, you must have understood some similarities and differences between Hashtable and HashMap in Java.

Now, go through the video given below to learn how to declare a simple HashMap in Java and some important methods of HashMap in Java.

We declare the HashMap in Java using the instruction given below:

HashMap<keyDataType, valueDataType> hashMapName = **new** HashMap<keyDataType, valueDataType>();

Some of the commonly used methods of HashMap are as follows:

|  |  |
| --- | --- |
| **Methods** | **Operations** |
| put(key,value) | This method adds the specified key with the specified value to the HashMap. |
| remove(key) | If the key is present in the HashMap, it removes the key along with the value mapped to it. |
| containsKey(key) | If there is any mapping to the specified key, then it returns true. |
| size() | This returns the number of key-value mappings present in the HashMap. |
| isEmpty() | If there is no key-value mapping present in the HashMap, it returns true. |
| clear() | It removes all mappings present in the HashMap. |
| get(key) | It returns the value mapped to the specified key in the HashMap. |
| keySet() | It returns the set of keys present in the HashMap. |

#### Q24: HashMap

Does simple HashMap store the elements in the insertion order?

Ans: No

**✓ Correct**

**Feedback:**

Simple HashMap does not store the elements in the insertion order.

#### Q25: HashMap

What is the output of the following program?

**import** **java.util.\***;

**public** **class** **Source** {

**public** **static** **void** main(String args[]) {

HashMap<Character, Integer> h = **new** HashMap<Character, Integer>();

h.put(*'a'*, 3);

h.put(*'b'*, 2);

System.out.print(h.containsKey(*'a'*) + *" "*);

System.out.print(h.get(*'a'*) + *" "*);

h.remove(*'a'*);

System.out.print(h.containsKey(*'a'*) + *" "*);

h.clear();

System.out.print(h.isEmpty() + *" "*);

}

}

Ans: true 3 false true

**✓ Correct**

**Feedback:**

The program follows the following steps:

1. Create a HashMap ‘h’.
2. Add a key ‘a’ with the value 3 to ‘h’
3. Add a key ‘b’ with the value 3 to ‘h’
   1. Now, ‘h’ contains ‘a’ and ‘b’ along with their respective values.
4. Print ‘h.containsKey('a')’.
   1. If there is any mapping to the specified key, then ‘containsKey(key)’ returns true. Otherwise, it returns false.
   2. As ‘h’ contains ‘a’, containsKey('a') returns true.
5. Print ‘h.get('a')’
   1. get(key) returns the value mapped to the specified key in the HashMap.
   2. The value of ‘a’ in ‘h’ is 3. Therefore, 3 is printed.
6. Remove the key ‘a’ along with its value mapped to it.
7. Print ‘h.containsKey('a')’.
   1. Since ‘a’ is removed in the above step, ‘h’ does not contain ‘a’. Therefore, ‘h.containsKey('a')’ returns false.
8. Removes all mappings present in the HashMap.
9. Print ‘h.isEmpty()’.
   1. If there is no key-value mapping present in the HashMap, h.isEmpty() returns true. Otherwise, it returns false.
   2. As all mappings are removed from ‘h’, ‘h’ is empty now. Therefore, h.isEmpty() returns true.

# Highest Frequency

uppose for a given array of strings, you need to find the string that occurs most frequently.

For example:

{‘Skinshape’, ‘Mild\_Orange’, ‘Arctic\_Monkeys’, ‘The\_Strokes’, ‘Arctic\_Monkeys’, ‘Summer\_Salt’, ‘Feng\_Suave’, ‘The\_Buttertones’, ‘Mild\_Orange’, ‘Arctic\_Monkeys’}

In this example, the string ‘Arctic\_Monkeys’ repeats 3 times, the string ‘Mild\_Orange’ repeats 2 times, while the rest of the strings repeat 1 time each.

Therefore, your code should return ‘Arctic\_Monkeys’, as it occurs most frequently.

Sounds easy, right? Here’s the catch though, your solution should have time complexity of O(n).

# Highest Frequency

Suppose for a given array of strings, you need to find the string that occurs most frequently.

For example:

{‘Skinshape’, ‘Mild\_Orange’, ‘Arctic\_Monkeys’, ‘The\_Strokes’, ‘Arctic\_Monkeys’, ‘Summer\_Salt’, ‘Feng\_Suave’, ‘The\_Buttertones’, ‘Mild\_Orange’, ‘Arctic\_Monkeys’}

In this example, the string ‘Arctic\_Monkeys’ repeats 3 times, the string ‘Mild\_Orange’ repeats 2 times, while the rest of the strings repeat 1 time each.

Therefore, your code should return ‘Arctic\_Monkeys’, as it occurs most frequently.

Sounds easy, right? Here’s the catch though, your solution should have time complexity of O(n).

# Find Symmetric Pairs

You will be given an array of pairs, and you have to print all the symmetric pairs. Pair (a, b) and a pair (c, d) are called symmetric pairs if a is equal to d and b is equal to c.

Example:

If the given array of pairs is {{1, 2}, {2, 3}, {3, 4}, {4, 3}, {2, 1}}, then the symmetric pairs in the given array of pairs are (1, 2) and (3, 4) because the pair (1, 2) has its symmetric pair (2, 1), and the pair (3, 4) has its symmetric pair (4, 3) in the given array.

The detailed algorithm to print all the symmetric pairs used in the previous video is as follows:

**Algorithm:**

Note: ‘arr’ is the array of pairs

1. for(int i = 0; i < arr.length; i++) // Traverse through the given array
   * int firstC = arr[i][0]; // Get the first and second elements of the current pair
   * int secondC   = arr[i][1];
   * for(int j = i+1; j < arr.length; j++) // Check whether the pairs are symmetric to the current pairs or not
     + int secondO = arr[j][1];
     + int firstO = arr[j][0];
     + if(firstO == sec && secondO == first) //If the current pair is symmetric to the other pair, then print the current pair
       - Print the current pair

**The approach used in the previous video is as follows:**

1. Create a hashmap.
2. Traverse through the array and check for every current pair to identify whether the second element of the current pair is present in the hashmap or not.
   * If it is present, then check whether the first element of the current pair and the value of the key in hashmap are the same or not.
   * If they are the same, then print the key and value in the hashmap as a symmetric pair.
   * Otherwise, add that current pair to the hashmap considering the first element of the pair as the key and the second element of the pair as the value of the key in the hashmap.

**A summary of your learnings from this segment is as follows:**

1. In approach 1, for each pair, you need to traverse through the other pairs of the array to check whether the pairs are symmetric or not.
   1. This approach takes O(n2).
2. In approach 2, you try to minimise the time complexity by storing the pairs to the hashmap and get the symmetric pair from the hashmap.
   1. For each pair, checking whether the symmetric pair is present in the hashmap or not takes O(1) time. Since you are checking for all the pairs, the time complexity of this approach is O(n).
   2. For space complexity of the approach, in the worst case, if there are no symmetric pairs in the array, then the size of the hashmap is O(n). Therefore, the space complexity is O(n).

# First Unique Character

You will be given a string, and you have to find and print the first unique character, i.e., the first non-repeating character of the string.

**Note:** The string may contain duplicate characters.

**Example:**

If the given string is ‘abcdebadf’, then the first unique character is c.

If you observe the given string:

1. The first character ‘a’ is repeated abcdebadf.
2. The second character ‘b’ is repeated abcdebadf.
3. The third character ‘c’ is not repeated abcdebadf.

Therefore, the first unique character, i.e., the first non-repeating character of the string, is ‘c’.

Approach 1

Example:

What is the first unique character of the string ‘adcbdace’?

Now, let’s follow the approach given in the suggested answer to the question given above.

* First, to get the count of each character of the string, create a hashmap of a character and its count where the key of the hashmap represents the character of the string and the value in the hashmap represents the number of times the character is repeated in the string.
* After creating the hashmap, insert all the characters and their respective counts into it by scanning through the string once. The same is done here:
  + Check whether the hashmap contains the character ‘a’ or not. Since there is no character ‘a’ present in the hashmap, insert ‘a’ with count as 1.
  + The next character ‘d’ is not present in the hashmap. So, insert ‘d’ with count as 1 into the hashmap.
  + The next character ‘c’ is not present in the hashmap. So, insert ‘c’ with count as 1 into the hashmap.
  + The next character ‘b’ is not present in the hashmap. So, insert ‘b’ with count as 1 into the hashmap.
  + The next character ‘d’ is already present in the hashmap. So, increment its count by 1, which makes it 2.
  + The next character ‘a’ is already present in the hashmap, so increment its count by 1, which makes it 2.
  + The next character ‘c’ is already present in the hashmap. So, increment its count by 1, which makes it 2.
  + The next character ‘e’ is not present in the hashmap. So, insert ‘b’ with count as 1 into the hashmap.
* After inserting the characters of the string into the hashmap, the hashmap looks like the following:

|  |  |
| --- | --- |
| **Key (character)** | **Value (count)** |
| a | 2 |
| d | 2 |
| c | 2 |
| b | 1 |
| e | 1 |

* Now, scan through the string again, and while scanning through each character, check its count in the hashmap. If the count is 1, then stop scanning further and print that character. Otherwise, scan through the string until you either reach such a character or the end of the string. This same is done here:
  + The count of the first character ‘a’ of the string is 2; so, continue scanning.
  + The count of the next character ‘a’ of the string is 2; so, continue scanning.
  + The count of the next character ‘c’ of the string is 2; so, continue scanning.
  + The count of the next character ‘b’ of the string is 1; so, stop scanning. Since ‘b’ is the first unique character in the given string, print this character.
  + Therefore, the first unique character of the string is ‘b’.

#### Q26: First Unique Character

Suppose you follow the approach given above to find the first unique character of the string. How many times do you need to traverse through the string?

Hint: Consider before hashing the value into the hash table what is done.

Ans: 2

**✓ Correct**

**Feedback:**

You need to scan through the string to store the count of each character in the hashtable, and you need to traverse through the string again to get the first character whose count is 1, i.e., to get the first unique character. Therefore, you need to traverse through the string twice.

#### Q27: First Unique Character

Following the approach you have learnt so far, write the pseudocode to print the first unique character of the string.

Ans: */\*str is the string in which you have to find the first unique character \*/*

HashMap<Character, Integer> h = **new** HashMap<Character, Integer>();

*/\* This adds each character to 'h', while adding it does*

*the following \*/*

**for** (**int** i = 0; i < str.length(); i++) {

**char** ch = str.charAt(i);

*/\* If the current character is not present in 'h', add*

*it to 'h' with the value as 1\*/*

**if** (h.containsKey(ch) == **false**) {

h.put(ch, 1);

}

*/\* Otherwise, increment the value of current character*

*in h\*/*

**else** {

h.put(ch, h.get(ch) + 1);

}

}

*/\* Now, traverse through the string from its first character*

*and check which character has the value as 1 in h \*/*

**for** (**int** i = 0; i < str.length(); i++) {

**char** ch = str.charAt(i);

**if** (h.get(ch) == 1) {

System.out.println(ch);

**break**;

}

}

You need to scan through the string to store the count of each character in the hashtable, which takes O(n) time. Then, you need to traverse through the string again to get the first character whose count is one, i.e., to get the first unique character, which again takes O(n) time. So, the total time taken is O(n)+O(n) time. Therefore, the time complexity of approach 1 to find the first unique character of the string is O(n).

In approach 1, you use a hashmap to store the count of each character of the string. The total number of ASCII characters is 256; in the worst case, you may have 256 distinct characters in the string. To store the count of all those 256 characters, you need a hashmap of size 256, which is of a constant size. Therefore, the space complexity of approach 1 is O(1).

Play Video

3776789

For more details on LinkedHashMap, please refer to [this](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/) link.

Note: ‘n’ is the length of the string.

1. The time complexity of approach 1 is O(2n), which is O(n).
2. The time complexity of approach 2 is (O(n) + O(C)), where C is 256, which is O(n).

But when the value of n is less than 256/2, then the time complexity of approach 1 is less than time complexity of approach 2. Otherwise, the time complexity of approach 1 is greater than the time complexity of approach 2.

Example 1:

n=5

Time complexity of approach 1 is O(2n) = O(10).

Time complexity of approach 2 is (O(n) + O(C)) = O(5) + O(256) = O(256).

For the detailed video solution to the coding console question given above, please refer to the following path:

Resources -> Additional References - Data Structures and Algorithms -> Hashtables Additional Resources -> Video Solutions -> Code Solution

So far in this session, you have learnt about a data structure that is similar to Hashtable in Java, i.e., HashMap, and solved some interesting problems using HashMap. In the upcoming segment, you will be introduced to a new data structure called HashSet, which uses a hash table underneath to store different values.

#### Q28: LinkedHashMap

Which of the following stores the elements in the order of insertion?

Ans: LinkedHashMap

**✓ Correct**

**Feedback:**

Java SDK provides a hashtable-based data structure that tracks the order of element insertion. A LinkedHashMap stores the elements in the order of insertion.

#### Q29: Sets in Java

When to use sets? (Choose all the options that apply.)

Ans: If you want to store distinct elements.

**✓ Correct**

**Feedback:**

Unlike lists, stacks and queues, the elements present in a set do not follow any particular order. They are randomly present in the set.

If you want to store the elements regardless of their order

**✓ Correct**

**Feedback:**

The elements in a set do not follow any particular order.

**Implementations of the set**

There are three implementations of the **set** interface in Java, which are as follows:

1. HashSet:
   * This is the most commonly used implementation of the set. Here, the elements are stored randomly, and duplicates are not allowed.
2. LinkedHashSet
   * Here, the order of the elements is maintained based on their insertion order, and no duplicates are allowed.
3. TreeSet
   * Here, the order of the elements is maintained by inbuilt ordering or by the explicit comparator (which can arrange it in any sorted order) of TreeSet. Duplicates are not allowed in this either.

You will learn how to declare a new HashSet in Java and gain an understanding of some important internal methods of HashSet in Java.

#### Q30: Sets in Java

Write the output of the following program:

**import** **java.util.\***;

**class** **Source** {

**public** **static** **void** main(String[] args) {

*// 1. Creating a HashSet named "hashSet"*

Set<Integer> hashSet = **new** HashSet<Integer>();

*// 2. Adding the elements to the hashSet*

hashSet.add(1);

hashSet.add(2);

hashSet.add(3);

hashSet.add(4);

*// 3. Adding the the element 3 again to the hashSet*

hashSet.add(3);

*/\* 4. Checking whether the hashSet contains the*

*element 2 or not, and print "2 is contained" \*/*

**if**( hashSet.contains(2)){

System.out.println(*"2 is contained"*);

}

*// else print "2 is not contained"*

**else**{

System.out.println(*"2 is not contained"*);

}

*// 5. Now, remove element 3 from the hashSet*

hashSet.remove(3);

*/\* 6. Check whether the element 3 is present in*

*the hashSet or not, and print the result \*/*

System.out.println(hashSet.contains(3));

}

}

Ans: **Output:**

2 is contained  
false

**Explanation:**

The following points represent those mentioned as comments in the program given above.

1. First, the HashSet named 'hashSet' is created.
2. Then, elements 1, 2, 3 and 4 are added to the hashSet.
3. Again, element 3 is added to the hashSet, but**the HashSet stores only distinct elements. So, 3 is stored as only one element in the hashSet.**
4. Later, we will check whether the hashSet contains element 2 in the hashSet or not.
   * If  it is present, then the program prints '2 is contained'.
   * Otherwise, it prints '2 is not contained'.
5. Next, the element 3 is removed from the hashSet.
6. Since there is no element 3 in the hashSet, the program prints false.

#### Q31: HashSet in Java

What is the output of the following program?

**import** **java.util.\***;

**class** **Source** {

**public** **static** **void** main(String[] args) {

*// creating a HashSet named "hashSet"*

Set<Integer> hashSet = **new** HashSet<Integer>();

*// adding elements to the hashSet*

hashSet.add(6);

hashSet.add(6);

*// removing the element 6 from the hashset*

hashSet.remove(6);

System.out.println(hashSet.contains(6));

}

}

Ans: false

**✓ Correct**

**Feedback:**

The element 6 is added twice to the HashSet, but the HashSet cannot contain duplicate elements. So, it will contain only one single occurrence of element 6. Since the set contains only one instance of element 6, when that element is removed, there will not be any element 6 in the HashSet, and the set becomes empty. Thus, contains(6) returns false.

#### Q32: TreeSet in Java

Which of the following implementations of the Set interface stores the elements in a sorted order (ascending order)?

Ans: TreeSet

**✓ Correct**

**Feedback:**

TreeSet stores the elements in a sorted order.

#### Q33: TreeSet Code in Java

What is the output of the following program?

**import** **java.util.\***;

**class** **Source** {

**public** **static** **void** main(String args[]) {

*// create a TreeSet named "tSet"*

TreeSet tSet = **new** TreeSet();

*// adding elements to it*

tSet.add(*"4"*);

tSet.add(*"2"*);

tSet.add(*"7"*);

tSet.add(*"5"*);

tSet.add(*"9"*);

*// printing the TreeSet*

System.out.println(tSet);

}

}

Ans: [2, 4, 5, 7, 9]

**✓ Correct**

**Feedback:**

The program given above creates a TreeSet, adds elements to it and prints it. A TreeSet stores elements in an ascending order and does not store any duplicate elements.

# Check Array of Contiguous Integers

You will be given the array of ‘n’ integers, and you have to print 'true' if it can form a set of contiguous integers from distinct integers in the given 'n' integers. Otherwise, print 'false'.

Note: The given array of integers may contain duplicates.

Example:

1. If the given array of elements is {4, 5, 8, 9, 7, 6, 7, 3, 3}, then the output should be 'true' because the distinct elements  {4, 5, 8, 9, 7, 6, 3} of the array can form a set of contiguous integers {3, 4, 5, 6, 7, 8, 9}.
2. If the given array of elements is {4, 8, 9, 7, 6, 7, 3, 3} then the output should be 'false' because the distinct elements  {4, 8, 9, 7, 6, 3} of the array cannot form a set of contiguous integers (we are missing the integer ‘5’ in the range 3–9).

#### Q34: Check Array of Contiguous Integers

According to you, what is the simple approach to check whether the given elements are contiguous or not?

Ans: The simple approach is to store all the elements in an array and sort it. Now, traverse through the array and check if the array contains contiguous integers (considering duplicates) by checking whether the difference between every two consecutive elements is 1 or 0.

## ****Approach 1****

Given below is the pseudocode to print true if the distinct integers of the array form contiguous integers; otherwise, print 'false'.

**Pseudocode:**

1. Store all the elements into the array 'elements'.
2. Sort the array in ascending order.
3. For each element of the array, carry out the following:
   1. If the next element exists, then take the difference of the next element and the current element.
   2. If the difference is neither 1 nor 0, then print 'false' and return.
4. Print 'true'.

Can you think of any other approach that does not involde sorting of elements?  
Hint: Use a hash table

## ****Approach 2****

Note: 'temp' is an integer variable.

1. Traverse through the array and store each element to the HashSet. (Note: HashSet contains only distinct elements.)
2. Now, store the first element of the array to temp and get the sum of the following:
   1. The number of contiguous integers starting from temp and smaller than temp, and
   2. The number of contiguous integers starting from temp+1 and greater than temp+1. (Hint: You can get this sum using HashSet).
3. If the sum is equal to the size of HashSet, then return 'true'. Otherwise, it returns 'false'.

#### Q35: Check Array of Contiguous Integers

What is the time complexity of approach 1 given above?

Hint: Revisit the searching and sorting algorithms studied in Searching and Sorting module.

Ans: O(n log n)

**✓ Correct**

**Feedback:**

If you go through the pseudocode, you can observe that you need to sort the array of all the elements (which takes O(n log n) time) and then traverse through the complete array to check whether all the elements are contiguous (include duplicates) or not (which takes O(n) time). Therefore, the time complexity is O(n log n).

#### Q36: Check array of contiguous integers

What will be the output if the array {5, 8, 4, 4, 7, 6, 2, 6, 7, 3} is passed to the below method?

**public** **static** **void** method(**int**[] array ){

**int** count = 0,n = array.length;

HashSet<Integer> hashSet = **new** HashSet<Integer>();

**for** (**int** i = 0; i < n; i++)

hashSet.add(array[i]);

**int** currentElement = array[0];

**while** (hashSet.contains(currentElement) == **true**) {

count++;

currentElement--;

}

System.out.println(count);

}

Ans: 4

**✓ Correct**

**Feedback:**

The method provided above prints the number of distinct consecutive integers present in the ‘hashSet’ that are less than or equal to the first element of the ‘array’, by following the steps given below.

1. Store all the ‘array’ elements to the ‘hashSet’.
2. Get the first element of the ‘array’ and store it to the ‘currentElement’ (here, ‘currentElement’ is an integer variable).
3. Initialise an integer variable ‘count’ as 0 (count is the variable which stores the number of distinct consecutive integers present in the ‘hashSet’ that are less than or equal to the first element of the ‘array’).
4. If the ‘currentElement’ is present in the ‘hashSet’, then increment the count by 1 and decrement the ‘currentElement’” by 1.
5. Repeat the above step until the ‘currentElement’ is present in the ‘hashSet’.

The consecutive integers in the ‘hashSet’ that are less than or equal to the first element ‘5’ of the ‘array’ are {5, 4, 3, 2}. Therefore, the answer is 4.

#### Q36: Check array of contiguous integers

Write the pseudocode for approach 2 given above, to print ‘true’ if the distinct elements of the given array form contiguous integers, otherwise, print ‘false’?

Ans: Note:

* ‘array’ is the given array of integers.
* ‘hashSet’ is a HashSet to store all the distinct elements of the array.
* ‘currentElement’ is an integer variable.

1. Store all the elements of the ‘array’ to the ‘hashSet’.
2. Initialise the integer variable ‘count’ as 0.
3. Initialise the ‘currentElement’ as array[0].
4. While ‘"currentElement’ present in ’hashSet’ do the following:
   1. Iincrement ‘count’ by 1
   2. Decrement the ’currentElement’ by One
5. Now to check for the elements greater than the array[0], modify the ’currentElement’ as array[0] + 1.
6. While ‘currentElement’ present in ‘hashSet’ do the following:
   1. Increment the ’count’ by 1.
   2. Increment the ’currentElement’ by 1.
7. Print ‘true’ 'if the ‘count’ is equal to the size of the ’hashSet’.
   1. Otherwise, print ‘false’.

#### Q37: Check Array of Contiguous Integers

What are the time and space complexities of the pseudocode given in the previous question, respectively?

Note:

* ‘n’ is the number of integers in the array.
* Consider the average case (or the likely case) of the contains() method of HashSet which is O(1).

Hint: Consider the pseudocode in the feedback of the previous question to find out the time and space complexity.

Ans: O(n) and O(n)

**✓ Correct**

**Feedback:**

In the pseudocode of the approach 2, you have to insert all the array integers to the HashSet, which takes O(n) time, and in the worst case, if all the integers of the array are distinct, then you need O(n) space. To check whether the HashSet contains contiguous integers or not, you again need to traverse through the complete HashSet, which again in the worst case it takes O(n) time. Therefore, the time complexity is O(n), and space complexity is O(n).

If we consider the worst-case time complexity of the contains() method of HashSet, which is O(n) in Java 7 and O(log n) in Java 8, then the time complexity of the approach will devolve into O(n2) and O(n log n) time in the Java 7 and Java 8, respectively.

# Pair With a Given Sum

You will be given an array of integers and a target sum. You need to check whether there exist any two integers in the array whose sum is equal to the given target sum. If so, then print ‘true’; otherwise, print ‘false’.

**Example:**

If the given array of integers and the target sum are {1, 7, 40, 8, -7, 3} and 15, respectively, then check whether a pair with the sum equal to the given target sum exists or not.

The output should be ‘true’ because 7 and 8 are the pair of integers from the given array of integers whose sum is equal to the given target sum 15.

#### Q38: Pair With a Given Sum

Write an approach to check for any two integers in the given array whose sum is equal to the given target sum.  
Hint: Sort the array first and then consider applying the two pointers concept.

Ans: **Approach 1:**

1. Sort the array and then initialise the variables ‘first’ and ‘last’ pointing to the first and last indices of the array.
2. Now, calculate the sum of the elements at ‘first’ and ‘last’ indices of the array.
3. If the sum is greater than the target sum, then decrement the variable ‘last’ by 1. However, if the sum is lower than the target sum, then increment the variable ‘first’ by 1.
4. Repeat the two steps given above until the sum matches the target sum or until the variable ‘first’ is smaller than the variable ‘last’.
5. After step 4, if the sum is equal to the target sum, then print ‘true’; otherwise, print ‘false’.

#### Q39: Pair With a Given Sum

What is the time complexity of approach 1 given as the suggested answer to the above question?

Note: Consider the pseudocode suggested in the previous question to calculate the time complexity.

Ans: O(n log n)

**✓ Correct**

**Feedback:**

In approach 1, you need to sort the array, which takes O(n log n) time. Then, you need to take the ‘first’ and ‘last’ pointers pointing to the first and last index of the array, respectively, and traverse through the array until you find the required pair or until the ‘first’ pointer is smaller than the ‘last’ pointer, which takes O(n) time. So, the time complexity is O(n log n).

#### Q40: Pair With a Given Sum

Can you think of a more time-efficient approach than approach 1 given in the earlier question?

Ans: **Approach 2:**

Create a HashSet. For each element of the array, get the difference of target sum and the array element. If the difference is present in the HashSet, it means that there are two integers in the array whose sum is equal to the given target sum. Otherwise, push that element to the HashSet and repeat the same for the remaining array elements.

Go through the following pseudocode to print ‘true’ if there exist two integers in the array whose sum is equal to the given target sum.

**Pseudocode:**

1. Create a HashSet.
2. For each element of the array, carry out the following:
   1. Get the difference of the target sum and the array element.
   2. If the HashSet contains the difference:
      1. Then print ‘true’ and return.
   3. Add the array element to the HashSet.

#### Q41: Pair With a Given Sum

What is the space complexity of approach 2 (using HashSet) whose time complexity is O(n)?  
Note:

* ‘n’ is the number of elements in the given array.
* Consider the average case (or the likely case) of the contains() method of HashSet, which is O(1).

Ans: O(n)

**✓ Correct**

**Feedback:**

You need to traverse through the array, and while traversing, check whether the difference of the target sum and the array element is in the HashSet or not; then, add the array element to the HashSet. In the worst case, if the last two elements of the array are the only two elements whose sum is equal to the target sum, then you have to traverse through the complete array of elements [which takes O(n) time) and push (n-1) elements to the HashSet that requires O(n) extra space to store n-1 elements to the HashSet].

If we consider the worst-case time complexity of the contains() method of HashSet, which is O(n) in Java 7 and O(log n) in Java 8, then the time complexity of the approach will devolve into

O(n2) and O(n log n) time in Java 7 and Java 8, respectively.

1. In approach 1, you need to sort the array and use two pointers to solve the problem.
   1. This approach takes O(n log n) time.
2. In approach 2, you can reduce the time complexity using a HashSet to store the array elements. For each array element, check whether the other element is present in the HashSet or not such that the sum of both the elements is equal to the target sum.
   1. This approach takes O(n) time.

Now, try to solve the coding console question given below via approach 2 because approach 2 is more efficient than approach 1 [approach 1 takes O(n log n) time, whereas approach 2 takes O(n) time].

# Find Itinerary From All Tickets

You are supposed to visit different cities for business and are provided all the travel tickets with the source and destination. Now, you need to figure out the itinerary of your visit to different cities from all those tickets.

Note:

1. The path covered by the tickets is not circular.
2. Other than the final destination, there is exactly one ticket from every city.

Example:

You will be given the source and destination of all the tickets in a map as follows:

**Source  - Destination**

Mumbai  - Indore

Hyderabad  - Warangal

Indore  - Hyderabad

Delhi  - Mumbai

Now, you have to print the itinerary. The itinerary for the given tickets will be as follows:

**Itinerary:**

Delhi  -  Mumbai - Indore  - Hyderabad - Warangal

## ****Approach****

You can print the itinerary from the given source and destination by following the instructions given below.

1. Get the starting point of the itinerary.
2. After that, print the itinerary by traversing through the map.

#### Q42: Find Itinerary From all Tickets

Which of the following statements is true if the source and destination of all the tickets are given in the map data structure as discussed earlier?  
Note: The path covered by the tickets is not circular.

Hint: Recall the hash map structure that was covered in previous modules and decide the source and destination in which that can be stored.

Ans: The starting point of the itinerary is present as a key in the map data structure and not as a value.

**✓ Correct**

**Feedback:**

The map data structure contains the source as key and destination as value. It is already mentioned that the path covered by the tickets is not circular. So, the starting point of the itinerary cannot be reached by any other source of the tickets. Therefore, it cannot be present as a value in the map data structure.

Given below is the pseudocode to print the itinerary from the given map of source and destination of the tickets.

Note:

* ‘tickets’ is the map of source and destination of all the tickets.

**Pseudocode:**

1. Create a map ‘reverseTickets’ representing the reverse mapping of ‘tickets’ by following the steps given below:
   * Iterate through each entry of the ‘tickets’ map data structure.
   * For every entry, store the reverse mapping to the ‘reverseTickets’ map.
2. Now, get the starting point of the itinerary by following the steps given below:
   * For every entry of the ‘tickets’.
   * Check whether the key of the entry is present as the key in the ‘reverseTickets’ or not.
   * If not, then get the key, which is the starting point of the itinerary.
3. After getting the starting point of the itinerary, traverse through the map data structure and print the itinerary.
   * Store the starting point to the ‘temp’ variable.
   * While temp not equal to null:
     + Print temp;
     + Now, get the value of temp from tickets, and store it to temp.

For time complexity of the pseudocode given above, consider traversing through the complete map to create a reverse map of it, which takes  O(n) time. Next, you have to traverse through the map again to get the starting point of the itinerary.  In the worst case, if the starting point is present in the last entry of the map, it again takes O(n) time.  After getting the starting point, iterate through the map and print the itinerary, which also takes O(n) time. Thus, the time complexity of the above-mentioned approach is O(n).

#### Q43: Find Itinerary From all Tickets

What are the space complexities of the pseudocode given above when its time complexity is O(n)?  
Note: ‘n’ is the size of the map (number of all tickets).

Hint: Consider the space required in the pseudocode when the input is first traversed.

Ans:   
O(n)

**✓ Correct**

**Feedback:**

Traverse through the complete map to create a reverse map of it, which takes  O(n) time and O(n) extra space to create a reverse map. Next, you have to traverse through the map again to get the starting point of the itinerary. In the worst case, if the starting point is present in the last entry of the map, it again takes O(n) time. After getting the starting point, iterate through the map and print the itinerary, which also takes O(n) time. Therefore, the time complexity and space complexity is O(n).

# Match Locks and Keys

You will be given an array of locks and an array of keys corresponding to locks. Each lock can be opened by only one key among all the given keys. The locks and keys are present in a random order in their corresponding arrays, and you have to match them.

**Note:**

1. You should not compare any lock with any other lock.
2. Similarly, you should not compare any key with any other key.
3. The size of the array of locks and keys is the same.

**Example:**

Given below is the array of keys and locks. Print the matching lock for each key.

**Input:**

Keys[] = {#, %, !, @, $}

Locks[] = {!, @, #, $, %}

**Output:**

#  #

%  %

!  !

@  @

$  $

## ****Approach 1****

Iterate through the array of keys, and for each iteration, check for the corresponding lock by iterating through the array of locks, and print the right combination.

#### Q44: Match Locks and Keys

Write the pseudocode for approach 1 that is given above?

Ans: **Pseudocode:**

Note:

* ‘locks’ and ‘keys’ are the arrays of locks and keys, respectively.
* The size of the array of locks and keys is ‘n’.

1. for(int i =0; i < n; i++) // iterating through the array of keys
   * for(int j = 0; j < n; j++) // for every keys[i] iterate through the array of locks to get the matching lock for the keys[i]
     + if( keys[i] == locks[j] ) print the keys[i] and locks[i]

Note:

* ‘locks’ and ‘keys’ are the arrays of locks and keys, respectively.
* The size of the array of locks and keys is ‘n’.

**Pseudocode:**

1. for(int i =0; i < n; i++) // iterating through the array of keys
   * for(int j = 0; j < n; j++) // for every keys[i] iterate through the array of locks to get the matching lock for the keys[i]
     + if( keys[i] == locks[j] ) print the keys[i] and locks[i]

#### Q45: Match Locks and Keys

What is the time complexity of approach 1?  
Note: ‘n’ is the size of the array of keys (the number of keys is equal to the number of locks).

Ans:   
O(n2)

**✓ Correct**

**Feedback:**

You need to traverse through the array of keys, and on each iteration, you need to traverse through the array of locks to get the matching lock. Since there are n keys and n locks, for every key of ‘n’ keys, you need to traverse through ‘n’ locks, which takes O(n2) time.

## ****Approach 2****

Traverse the array of locks and create a hashmap of all these locks. Then traverse through the array of keys, and for each key, get the corresponding lock from the hashmap and print them. If there is a lock corresponding to the key in the hashmap, it means that there is a lock for that key; if not, it means that there is no lock corresponding to the key.

#### Q46: Match Locks and Keys

You will be given the array of locks and keys; write the pseudocode to print the key and its corresponding lock in the order of the keys present in the array. If any of the lock corresponding to the key is not present, then print ‘Not possible’ and exit.

Note:

* ‘locks’ and ‘keys’ are the arrays of locks and keys, respectively.
* The size of the array of locks and keys is ‘n’.

Ans: **Pseudocode:**

1. Create a hashmap.
2. Iterate through the array of locks and add locks[i] as the key and the index ‘i’ as value to the hashmap.
3. For every key, check whether the corresponding lock is present or not. If it is present, then store the key[i] to locks[i]. If it is not present, print ‘Not possible’ and return.
   * for (int i = 0; i < n; i++)
     + if (hashMap.get(keys[i]) != null)
       - locks[i] = keys[i];
     + Else
       - Print “Not possible”
       - Return
4. for(int i = 0; i < n; i++)
   * Print keys[i] and locks[i]

**Pseudocode of Approach 2:**

1. Create a hashmap.
2. Iterate through the array of locks and add locks[i] as the key and the index ‘i’ as value to the hashmap.
3. For every key, check whether the corresponding lock is present or not. If present, then store the key[i] to locks[i]. If it is not present, print ‘Not possible’ and return.
   * for (int i = 0; i < n; i++)
     + if (hashMap.get(keys[i]) != null)
       - locks[i] = keys[i];
     + Else
       - Print “Not possible”
       - Return
4. for(int i = 0; i < n; i++)
   * Print keys[i] and locks[i]

For time complexity, you need to create a hashmap of all ‘n’ keys by traversing through the array ‘keys’. This step takes O(n) time for the hashmap.

Next, for every key, you need to check whether the corresponding lock present in the hashmap or not. This step takes O(n) time. Thus time complexity will be O(n).

To conclude:

1. In approach 1, for each key, you need to check the corresponding lock by iterating through the array of locks and print the right combination.
   1. This approach takes O(n2) time.
2. In approach 2, we try to reduce the time complexity by creating a hashmap of all locks, and for each key, we get the corresponding lock from the hashmap and print them.
   1. This approach takes O(n) time.

Now, try to solve the coding console question given below via approach 2 because approach 2 is more time-efficient than approach 1.

#### Q47: Match Locks and Keys

What is the space complexity of the pseudocode of approach 2?  
Note: ‘n’ is the size of the array of keys (number of keys is equal to the number of locks).

Ans: O(n)

**✓ Correct**

**Feedback:**

1. You need to create a hashmap of all ‘n’ keys by traversing through the array ‘keys’. This step takes O(n) time and O(n) extra space for the hashmap.
2. Next, for every key, you need to check whether the corresponding lock present in the hashmap or not. This step takes O(n) time.

Therefore, the time complexity is O(n) and space complexity is O(n).

# Introduction to Trees

The non-linearity of data has led to the evolution of non-linear data structures such as trees. The tree data structure connects every data point to several other data points in such a way that the connection between the different points and the structure of the tree represent a specific relation with the connected points. Also, as you have seen, trees convey the natural flow of information of a particular set of data.

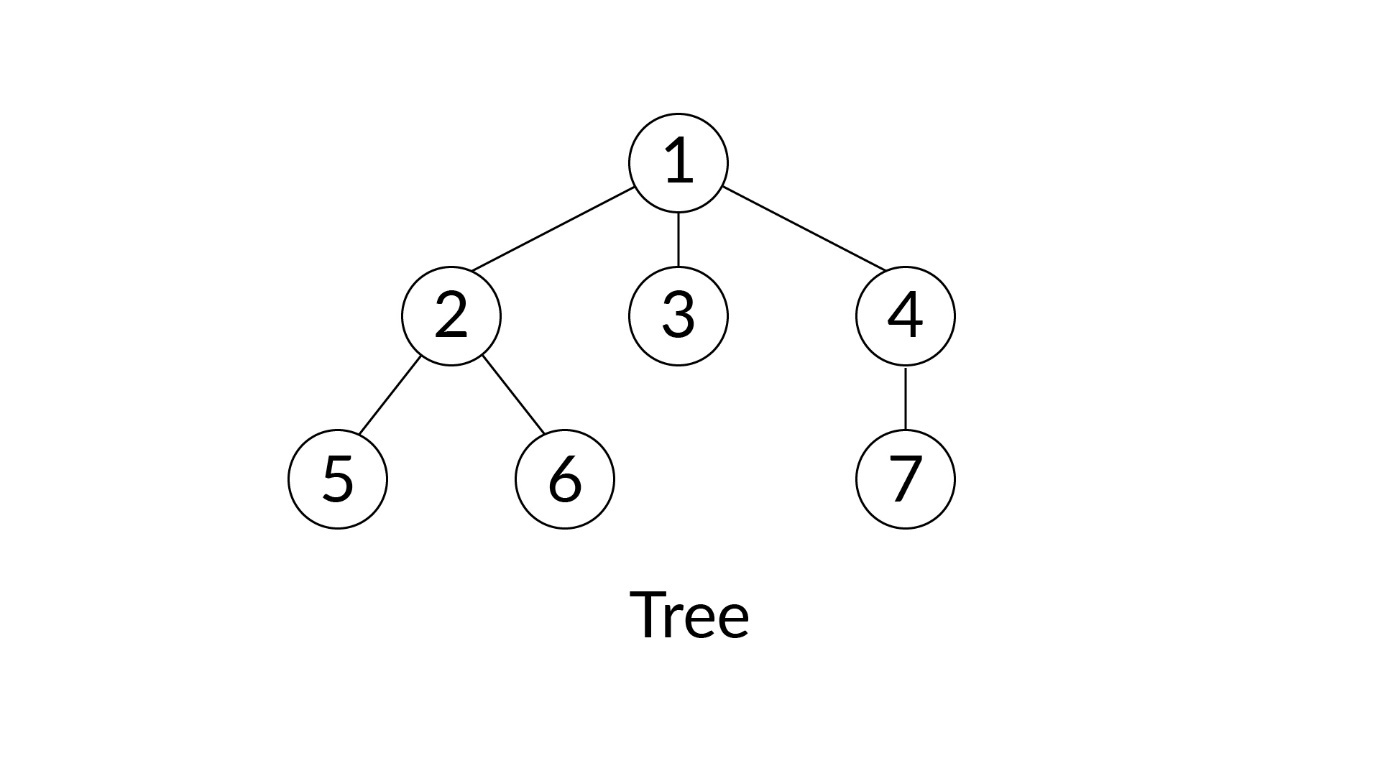
#### Q48: Tree Data Structure

You have watched the video given above. Can you think of one more example of non-linear data that can be represented using a non-linear data structure such as trees? Write your response in the box given below.

Ans: The set of possible moves in a game of chess can again be represented using a tree data structure. This will resemble a decision tree that has already been discussed in the video. This tree will contain possible moves for various situations represented by various nodes of a tree.

## Tree Vocabulary

Now, let's learn about various components of a tree and their names. Refer to the tree presented in the image given below, which follows a hierarchical order:



Tree

All the elements in a tree are called **nodes**. In the tree given above, the nodes are the ones with the values 1, 2, 3, 4, 5, 6 and 7.

Nodes can be of different types. The topmost node in a tree is called the **root node**.  
  
From the root node, descend some other nodes. The root node is called a parent if there is atleast one or more node descending from it directly. In that case, we can call root node 1 as the **parent node**having node 2 as a **child node**. Node 1 is also the parent of nodes 3 and 4. Similarly, node 2 is the parent of nodes 5 and 6, and node 4 is the parent of node 7. 

A node that does not have any child is called a **leaf node**. In the tree given above, nodes 3, 5, 6 and 7 are called **leaf nodes**.

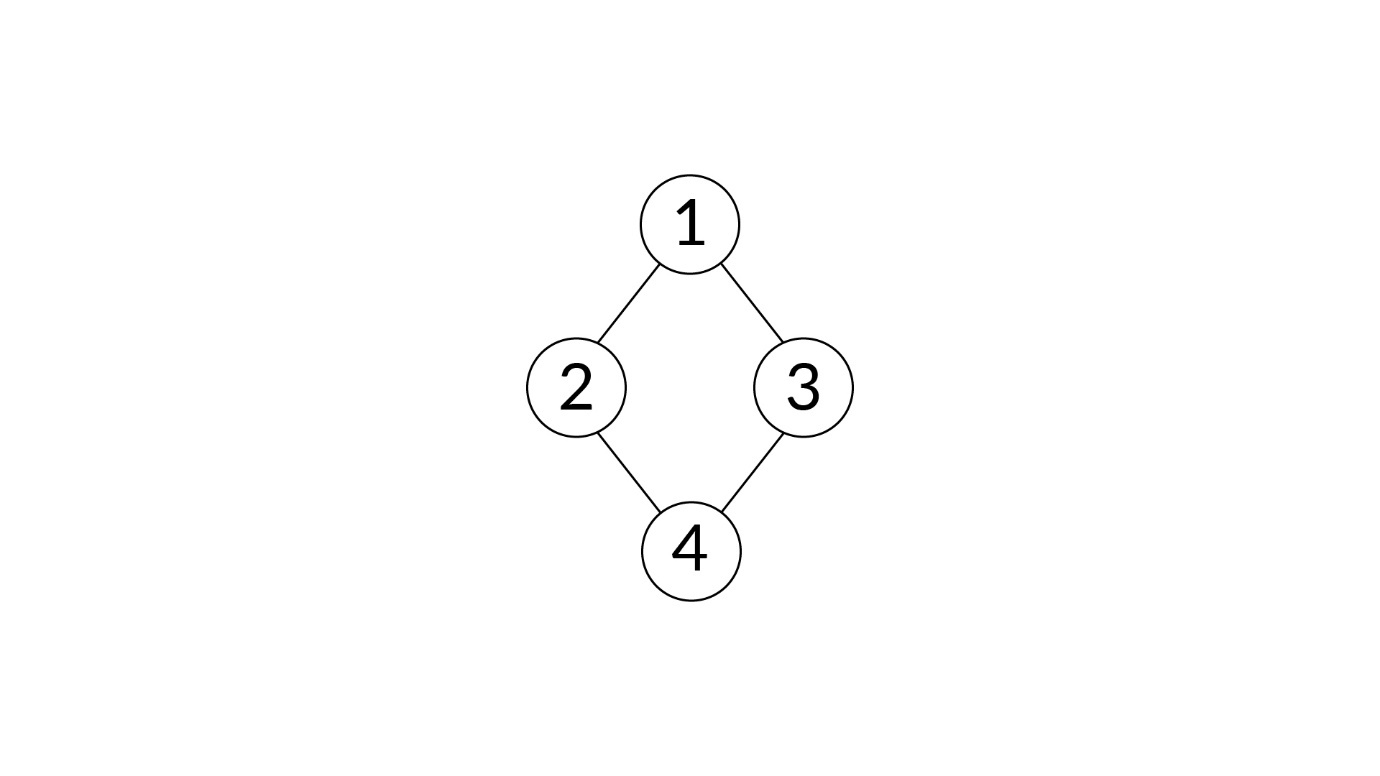
The nodes at the same level descending from the same parent are called **siblings**. In the tree given above, nodes 2, 3 and 4 are siblings.  
  
Following is the vocabulary for the tree given above:  
**Nodes:** 1, 2, 3, 4, 5, 6, 7

**Root Node:**1  
**Parent Nodes & their Children Nodes:**  
Parent - 1, Children - 2, 3, 4  
Parent - 2, Children - 5, 6  
Parent - 4, Child - 7

**Leaf Nodes:** 3, 5, 6, 7  
**Sibling Nodes:** (2, 3, 4), (5, 6)

## Properties of Trees

**1. No Cycle in Tree**  
An important point to remember is that a tree cannot have any cycle. A cycle means that in a tree, a child node cannot have two parent nodes. Take a look at the figure given below.



Not a Tree

You can see that the child node 4 has two parents–node 2 and node 3–which builds a cycle in the figure given above. Thus, the above figure cannot be considered to be a tree.  
  
**2. Number of edges = Number of nodes - 1**  
An edge is a line connecting two nodes in a tree. Every node except the root has a unique edge from its parent to the node. Thus, if there are N nodes in the tree, the number of edges would be N-1, i.e., one per node except for the root. Thus, if there are N nodes in a tree, the number of edges will be equal to N-1.

#### Q49: Find Number of Edges

The link between any two nodes of a tree is known as an edge. In a tree, if there are 31 edges, what will the number of nodes in that tree be?

Ans:   
32

**✓ Correct**

**Feedback:**

Every node, except the root, has a unique edge from its parent to the node. Thus, if there are N nodes in the tree, the number of edges would be N-1, i.e., one per node except for the root. Each edge points to a unique node other than the root. Thus, if there are N-1 edges, there are N nodes.

## Trees vs arrays vs linked lists

Let's discuss the important points of differentiation among arrays, linked lists and trees. These points are enlisted in the table given below.

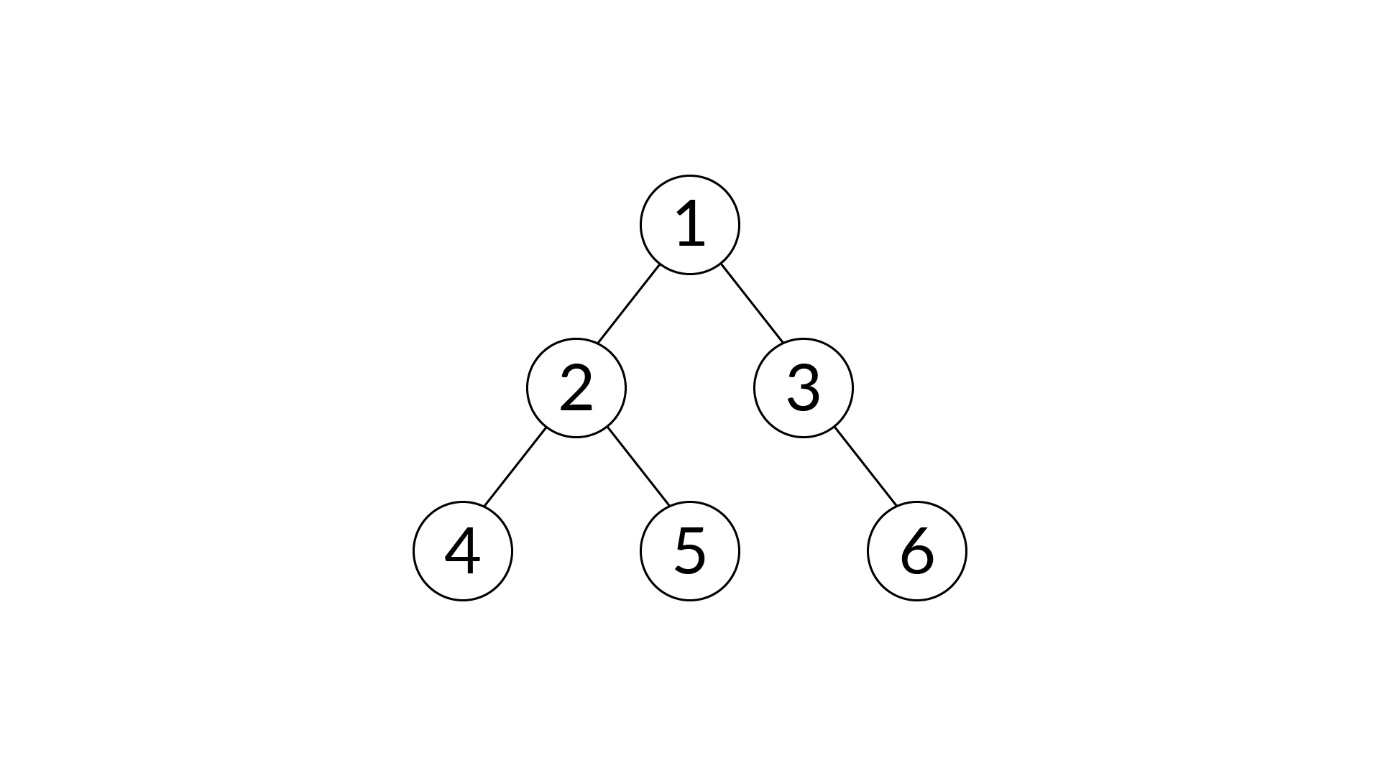
| **Criterion** | **Comparison** | **Justification** |
| --- | --- | --- |
| **Access (or Search)** | Linked lists < trees < arrays | Accessing (or searching) an element in a tree is quicker than accessing linked lists but slower than accessing arrays. |
| **Insertion / Deletion** | Arrays < trees < linked lists (Unordered) | Inserting or deleting an element from a tree is quicker than arrays but slower than unordered linked lists. |
| **Number of Elements** | Linked lists = trees ≠ arrays | Like linked lists and unlike arrays, trees can consist of as many number of elements as needed on the run time. |

Don't worry! You will understand these points of differences when you study trees in detail while following along the module. We have included this table here now so that when we discuss the characteristics of trees, you have a slight idea about the differences among trees, arrays and linked lists

# Introduction to Binary Tree

**A binary tree is a tree that has atmost two children.** In other words, each node can have **either 0 or 1 or 2 children**.

When it comes to the vocabulary of binary tree that can have a maximum of two children, we call the child on the left side–the **left child**–and the child on the right side–the **right child**.   
  
Consider the binary tree given below.

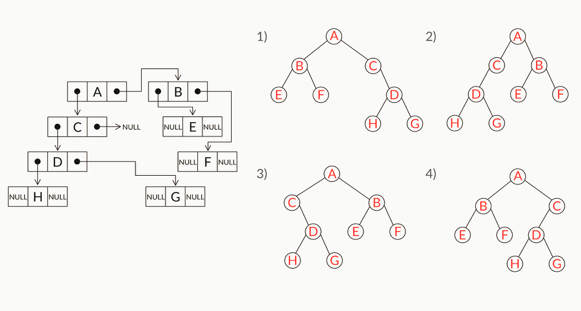


Tree

In the binary tree given above, the root node 1 is a parent that has two child nodes - 2 and 3. 2 is the left child of 1, and 3 is the right child of 1. 4 is the left child of parent node 2 and 5 is the right child of parent node 2. 3 has the right child as 6.

#### Q50: Binary Tree

Assume that a node in a tree structure has data, a pointer to the left child as well as a pointer to the right child as its parts. If the tree’s physical representation with nodes is as shown in the figure, find its logical representation from the given options.



Ans: 2

**✓ Correct**

**Feedback:**

Start with a node that has no parent as the root node and construct the tree based on the right and left pointers. The node 'A' will be the root node because it does not have any parent node. It has the node 'B' to its right and node 'C' to its left. Node 'B' has nodes 'E' and 'F' as its left and right children, respectively. The node 'C' has node 'D' to its left, and node 'D' has nodes 'H' and 'G' as its left and right children, respectively.

# Properties of Binary Tree

**Property 1:**  
**The maximum number of nodes**n**at level**l**of binary tree is**2l−1**.**

**Property 2:**  
**The maximum number of nodes**n**in a binary tree of height**h**is**2h−1**.**

**Property 3:**

**The minimum height**h**of a binary tree with**n**nodes is**log2(n+1)**.**

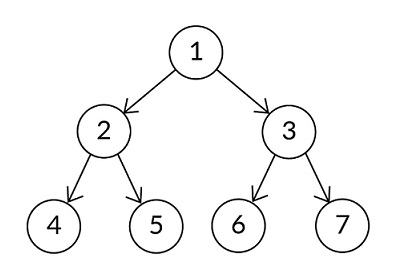
**Property 4:**

**The minimum levels**l**in a binary tree with**L**leaves is**log2L+1**.**

# Types of Binary Trees

**In a full binary tree, every node has two child nodes except the leaf nodes.**

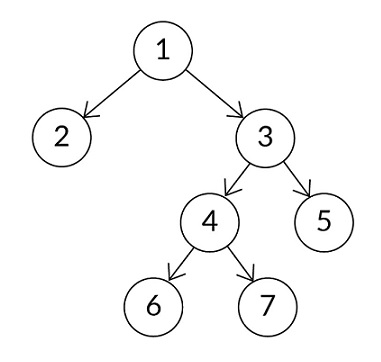
**Example 1:**



Example 1

In the binary tree given above, all the nodes have two children, except the leaf nodes at the last level. Thus, it is a full binary tree.

**Example 2**:



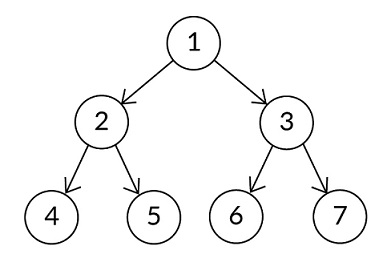
Example 2

The binary tree given above is a full binary tree because all the trees have two children except the leaf nodes–node 2, node 5 and the nodes at the last level. Remember that a node is a leaf node if it does not have any child. A leaf node is not necessarily needed to be present at the last level.

## Complete binary tree

Now here you will understand the concept of a complete binary tree.

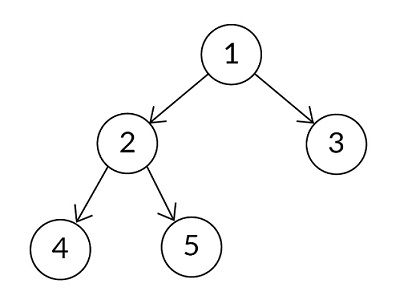
**In a complete binary tree, all the levels are completely filled. The exception to this exists for the last level, which means that the last level may or may not be completely filled.** This, in turn, means that the second-last level may or not have both the children.  
**With this exception, it is also must that the last level must have all the keys to the left side as much as possible.** In other words, a node in the second-last level cannot have a right child without having all the nodes in its left side.   
  
**Example 1:**



Example

In the example given above, all the levels are completely filled. Even the last level is completely filled. Thus, the tree given above is a complete binary tree.

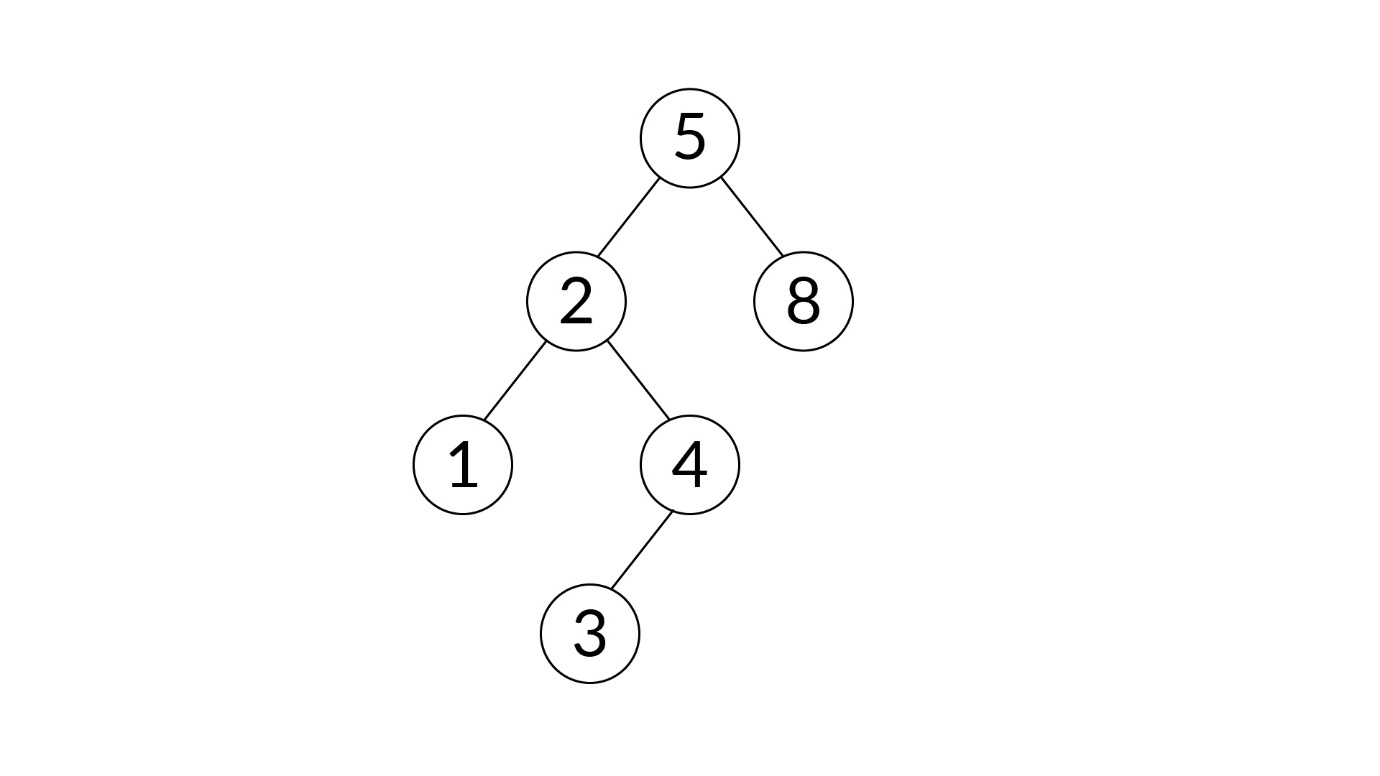
**Example 2:**



Example

In the example given above, all the levels are completely filled except the last level. Node 3 at the second-last level does not have any child. Thus, both the conditions hold true for the binary tree given above to be a complete binary tree.

**Example 3:**



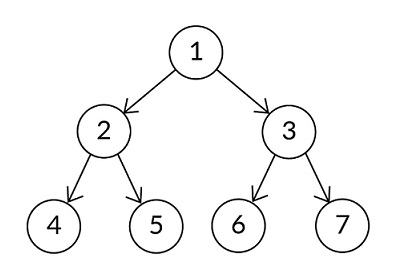
Example

In the example given above, the last level is not completely filled, but that is an exception and is allowed. However, there are two violations of the rule here. The first violation is that the second-last level is not completely filled because node 8 does not have 2 children. The second violation is that the last level does not have the leaf nodes completely on the left side because of node 4 containing a child without node 1 containing 2 children. Owing to these violations, the binary tree given above is not a complete binary tree. Note that even if one of these violations would have occurred, the binary tree given above would not be called a complete binary tree.

## Perfect binary tree

**In a perfect binary tree, all the nodes must have two children nodes except the leaf nodes and all the leaf nodes must be at the same level.**

**Example:**



Example

In the binary tree given above, all the nodes except the leaf nodes have two children. Also, all the leaf nodes are at the same level. Thus, the tree given above is a perfect binary tree.

Note the following:

**The maximum number of nodes *n* in a binary tree of height *h,***

n=2h−1

This is the **second property**that you studied in the segment '*Properties of Binary Trees*' earlier. This property always applies to a perfect binary tree because in a **perfect binary tree** of height *h*, the **number of nodes is always maximum**.

#### Q51: Binary trees

Which of the following statements is correct?

(Note: More than one option may be correct.)

Hint: Revise the definitions of full, complete and perfect binary trees given in this segment to answer the question.

Ans: Any perfect binary tree is also a complete binary tree.

**✓ Correct**

**Feedback:**

In a complete binary tree, all the levels should be filled, except the last level that may or may not be filled. In a perfect binary tree, including the last level, all the levels should be completely filled. Thus, a perfect binary tree is a subset of a complete binary tree. Thus, the statement given in this option holds true.



Any perfect binary tree is also a full binary tree.

**✓ Correct**You missed this!

**Feedback:**

In a full binary tree, all the nodes except the leaf nodes must have two children.  In a perfect binary tree, all the levels are full, which implies that all the nodes except the leaf nodes have two children. Thus, a perfect binary tree is a subset of full binary tree. Thus, the statement given in this option holds true.



Every complete binary tree need not be a perfect binary tree.

**✓ Correct**

**Feedback:**

In a complete binary tree, all the levels should be filled, except the last level that may or may not be filled. In a perfect binary tree, including the last level, all the levels should be completely filled.

Consider example 2 of the complete binary tree given on the platform. It is a complete binary tree but not a perfect binary tree. Thus, the statement given in this option holds true.



Every full binary tree need not be a perfect binary tree.

**✓ Correct**You missed this!

**Feedback:**

In a perfect binary tree, all the nodes, except the leaf nodes, have two children; all the leaf nodes should be at the same level. In a full binary tree, all the nodes, except the leaf nodes, have two children but it is not necessary that every full binary tree will have all the leaf nodes at the same level.

Consider example 2 of the full binary tree given on the platform. It is a full binary tree but not a perfect binary tree. Thus, the statement given in this option holds true.

# Representation of Binary Tree

## General representation of a binary tree

you learnt that **each node** in a binary tree consists of the following: **data**, **left child** and **right child**. The data will consist of the value that a node will contain. The left child will consist of another node. Similarly, the right child also will consist of another node.

A **binary tree**, as a whole, will consist of a **root node** and **zero or more other nodes** that descend from the root node.

## Representation of binary tree in code

After looking at the general representation, you can take a look at the code for representing a binary tree in Java.

you learnt that the code for representing a binary tree in Java is as follows:

**Code representation of a binary tree:**

// class representing a node (element) in a tree

**class** **Node** {

**int** data; // value contained inside a node

Node left, right; // left & right children of a node

// constructor to set the data of a node to the passed value and make it a leaf node

Node(**int** value) {

data = value;

left = right = **null**;

}

}

// class representing a binary tree

**class** **Tree** {

Node root; // root node of the binary tree

// constructor to create an empty tree with no root node

Tree() {

root = **null**;

}

}

// driver class to create tree and test code

**public** **class** **Source** {

**public** **static** **void** **main**(String[] args) {

Tree tree = **new** Tree(); // constructing an empty tree

tree.root = **new** Node(**1**); // adding the root node

tree.root.left = **new** Node(**2**); // adding left child of root node

tree.root.right = **new** Node(**3**); // adding right child of root node

tree.root.left.left = **new** Node(**4**);

tree.root.left.right = **new** Node(**5**);

tree.root.right.left = **new** Node(**6**);

tree.root.right.right = **new** Node(**7**);

}

}

# Tree Traversal-Depth-First Search (DFS)

The two ways of traversing a tree are as follows:

**1. Depth-first search (DFS)**

**2. Breadth-first search (BFS)**

**It is important to note that sometimes, we consider the level of the root node as 0, whereas at other times, we consider the level of the root node to be 1. The level of the root node is usually considered to be 1.**

The following points summarise this idea well:

* **Inorder (stage 2, stage 1, stage 3)**
* **Preorder (stage 1, stage 2, stage 3)**
* **Postorder (stage 2, stage 3, stage 1)**

So, as discussed in the video, the three stages at which a node is visited are as follows:

* **Stage 1:** When only the node is visited and none of its right and left subtrees are visited
* **Stage 2:** When its left subtree is visited
* **Stage 3:**When both its left and right subtrees are visited

Depending on the stage at which the action is performed, the algorithm becomes a particular variant of depth-first traversal. An action can be printing its value, comparing its value with some other nodes, etc.

# DFS-Pseudocode and Code

You explored the variants of DFS. Let’s move on to the next segment, wherein our professor will discuss the codes for these variants. Now, you may be wondering why we are discussing these different traversal techniques extensively. What is the utility of each technique? The answer to this is that given a problem, the answer or solution to it may lie either in a postorder, in-order or pre-order traversal depending on the problem statement. Certain problems have answers hidden in pre-order traversals, a few have answers hidden in postorder traversals, and a few have theirs hidden in in-order traversals.

In the next session on binary search trees, you will observe that if you want to print nodes in the increasing order of their values, an in-order traversal answer. You can refer to the link given below to read more about this.

* [Utilities of pre-order, postorder, and in-order traversals](https://stackoverflow.com/questions/9456937/when-to-use-preorder-postorder-and-inorder-binary-search-tree-traversal-strate)

An alternative and probably simpler way of looking at traversals can be found at the link [here](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/).

The code that was included in the video given above is as follows:

// Method to print the tree in pre-order traversal

**void** **preOrderDFS**(Node node) {

**if** (node == **null**)

**return**;

// visit the parent node (parent of left & right children)

System.out.print(node.data + " ");

// recursively go to left subtree

preOrderDFS(node.left);

// recursively go to right subtree

preOrderDFS(node.right);

}

// Method to print the tree in in-order traversal

**void** **inOrderDFS**(Node node) {

**if** (node == **null**)

**return**;

// recursively go to left subtree

inOrderDFS(node.left);

// visit the parent node (parent of left & right children)

System.out.print(node.data + " ");

// recursively go to right subtree

inOrderDFS(node.right);

}

// Method to print the tree in post-order traversal

**void** **postOrderDFS**(Node node) {

**if** (node == **null**)

**return**;

// recursively go to left subtree

postOrderDFS(node.left);

// recursively go to right subtree

postOrderDFS(node.right);

// visit the parent node (parent of left & right children)

System.out.print(node.data + " ");

}

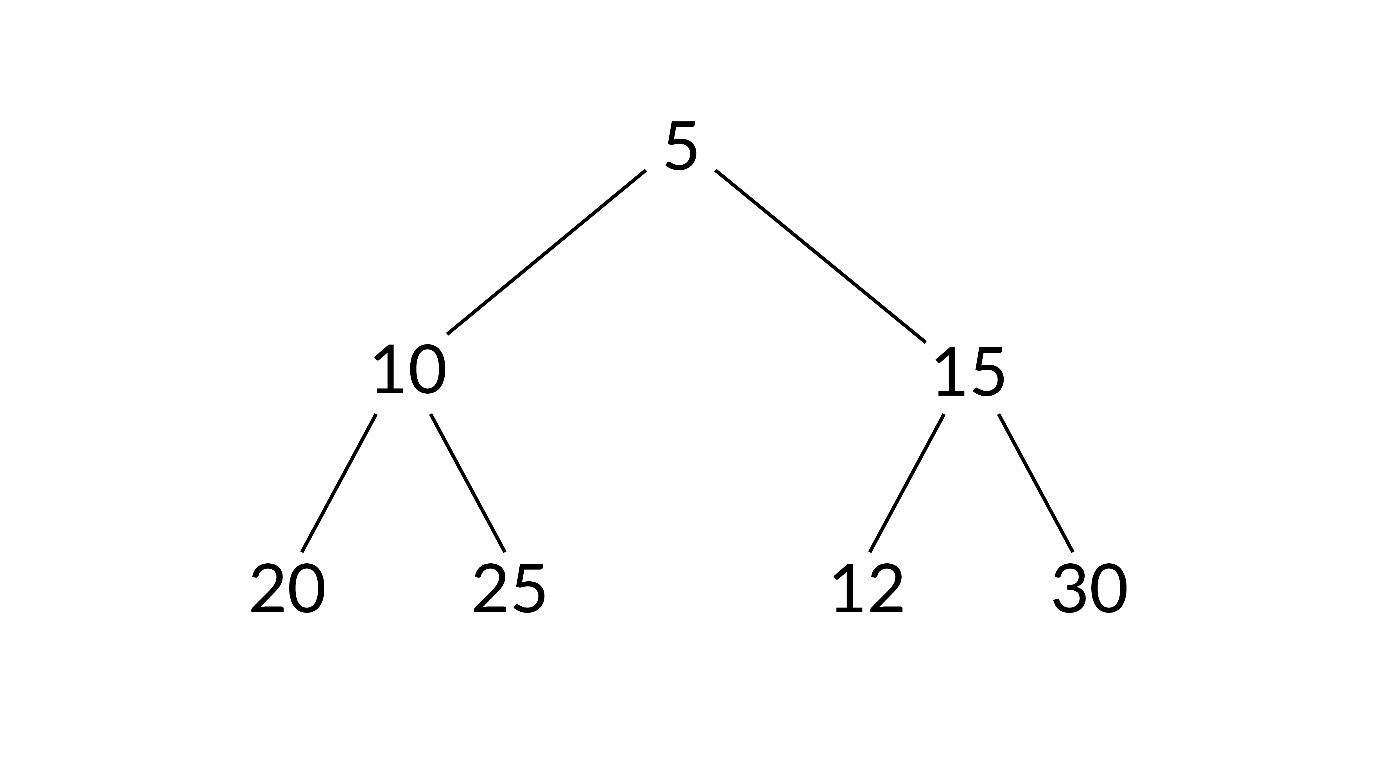
Coding question:

### Print Specific Nodes

**Description**

You are given a binary tree. You need to print all the nodes that are multiples of the root node. The nodes that are multiples of the root node should be printed using in-order traversal.

**Example:﻿**

**img\_5-01.jpg 339.83 KB**

﻿Output: 20 10 25 5 15 30

Here, nodes that are multiples of root node 5 are 5, 10, 15, 20, 25 and 30. These nodes should be printed using in-order traversal. The resulting output will be 20 10 25 5 15 30.

**Input:**

The input will be in the following format:

1. The first line should be the number ‘n’. Here, ‘n’ represents the number of nodes present in the binary tree.
2. The second line contains space-separated ‘n’ values. These are the values of respective nodes. The code for inserting the values in the tree has already been addressed in the code.

**Output:**

The output should be all the nodes that are multiples of the root node. Note: Print the nodes using inorder traversal separated by spaces.

**Sample Input:**

5

3 4 5 6 7

**Sample Output:**

6 3

**Sample Input:**

4

0 1 2 3

**Sample Output:**

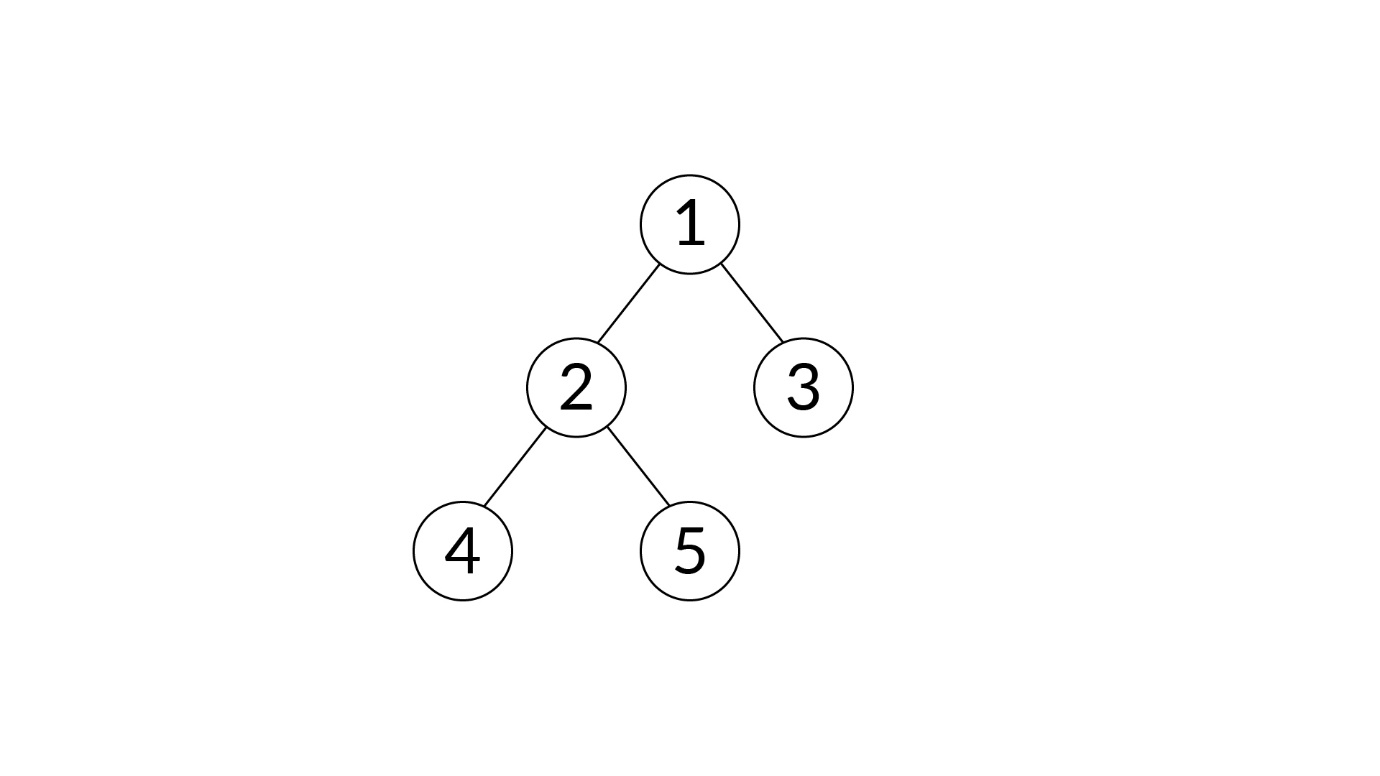
Division by zero is undefined

Ans: in intellij

### 2: Find Height of Binary Tree

**Description**

You are given a binary tree. You need to print the maximum height of the binary tree. If the tree is NULL (empty tree), print the height of the tree as 0.﻿

Image 5-01.jpg 97.24 KB

﻿

**Output:** 3

Here, in the given tree, the maximum height of the binary tree is 3.

**Input:**

The input will be in the following format:

1. The first line should be the number ‘n’. Here, the number ‘n’ is the total number of nodes present in the binary tree.
2. The second line contains space-separated ‘n’ values. The code for inserting the values in the tree has been addressed in the code.

**Output:**

The output should be the height of the tree keeping in mind that the root node is at level 1.

**Sample Input:**

5

1 2 3 4 5

**Sample Output:**

3

**Sample Input:**

0

**Sample Output:**

0

Ans: in Intellij

# Tree Traversal- Breadth-First Search (BFS)

So, in a BFS, nodes at a certain level are visited before proceeding to the next level. So, you first visit the root node, all the nodes at level 1, all the nodes at level 2, etc., in that order. Hence, in this algorithm, you move along the breadth of a tree before hopping on to the next level. This is the reason why **breadth-first search (BFS) traversal** is also called the **level-order traversal**.

#### Q53: Breadth-First Traversal

In your own words, explain the fundamental difference between a BFS and a DFS. Write your response in the box below.

Ans: In a DFS, the nodes along the depth of the tree are traversed first. So, if there are two nodes at the same level of the tree, all the child nodes of one of these nodes are visited before visiting the other node at the same level. On the other hand, in a BFS, all the nodes at the same level are traversed before moving on to the nodes on the next level.

# BFS (Recursive)-Pseudocode and Code

**Code for BFS (Recursive):**

// Method to calculate height of a tree

**int** **height**(Node root) {

**if** (root == **null**)

**return** **0**;

**int** leftHeight = height(root.left);

**int** rightHeight = height(root.right);

**if** (leftHeight > rightHeight)

**return** leftHeight + **1**;

**else**

**return** rightHeight + **1**;

}

// Method to traverse the elements of a tree using BFS (level-order traversal) in recursive way

**void** **levelOrderOrBFS**() {

**int** h = height(root);

**for** (**int** i = **1**; i <= h; i++)

printNodesAtLevel(root, i, **1**);

}

// Method to print nodes at the given level

**void** **printNodesAtLevel** (Node root, **int** level, **int** currentLevel) {

**if** (root == **null**)

**return**;

**if** (level == currentLevel)

System.out.print(root.data + " ");

**else** {

printNodesAtLevel(root.left, level, currentLevel + **1**);

printNodesAtLevel(root.right, level, currentLevel + **1**);

}

}

# BFS (Iterative)-Pseudocode and Code

**Code for BFS (Iterative):**

// Method to traverse the elements of a tree using BFS (level-order traversal) in iterative way using queue

**void** **levelOrderOrBFS**() {

Queue<Node> queue = **new** LinkedList<Node>();

queue.add(root);

**while** (!queue.isEmpty()) {

Node temp = queue.peek();

queue.remove();

System.out.print(temp.data + " ");

// enqueue left child into queue

**if** (temp.left != **null**)

queue.add(temp.left);

// enqueue right child into queue

**if** (temp.right != **null**)

queue.add(temp.right);

}

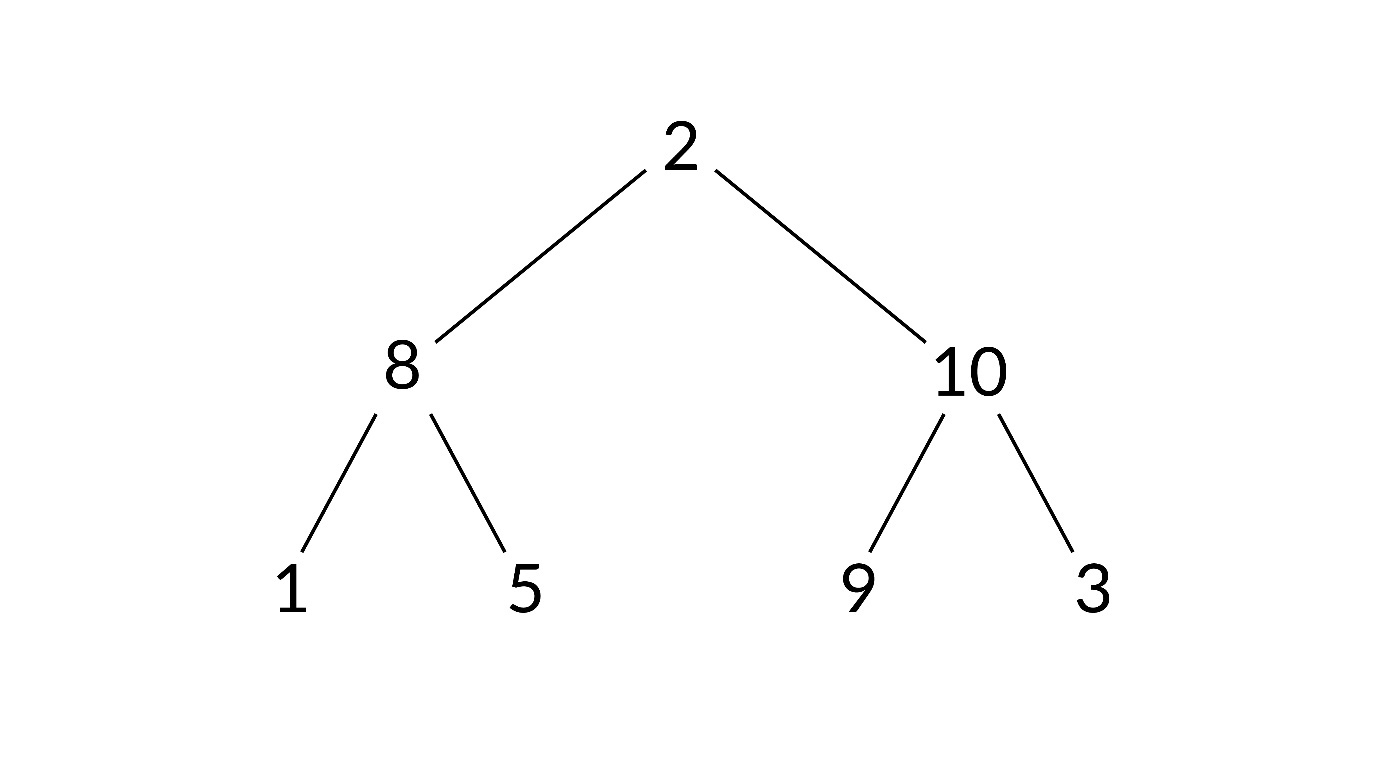
}

### Q1: Print Maximum Value

**Description**

You are given a binary tree. You need to print the node that has the highest value in the binary tree using the BFS (Iterative) approach. If the tree is empty, the output should be -1.

﻿

img\_4-01.jpg 313.96 KB

﻿

**Output:** 10

Here, in the given tree, the node that has the maximum value is 10. Therefore, the output is 10.

**Input:**

The input will be in the following format:

1. The first line should be the number ‘n’. Here, the number ‘n’ is the total number of nodes present in the binary tree.
2. The second line contains space-separated ‘n’ values. No value should be equal to -1. The code for inserting the values in the tree has already been taken care of in the code.

**Output:**

The output should be the maximum value among all nodes in the tree.

**Sample Input:**

5

1 2 3 4 5

**Sample Output:**

5

**Sample Input:**

0

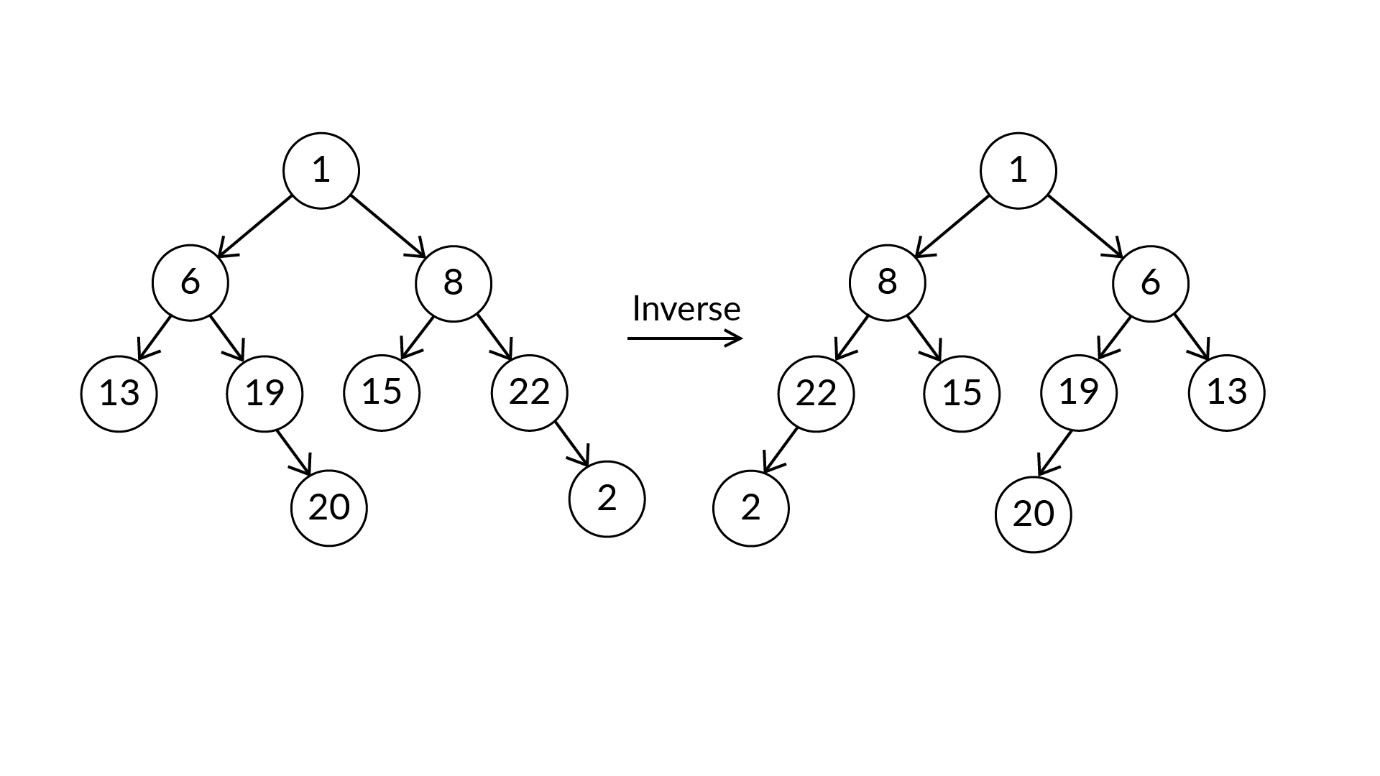
**Sample Output:**

-1

Ans: Intellij

# Mirror a Tree

Suppose you are given a binary tree, and you need to inverse it. The inverse of a tree is a mirror image of the nodes; check the image given below for reference.

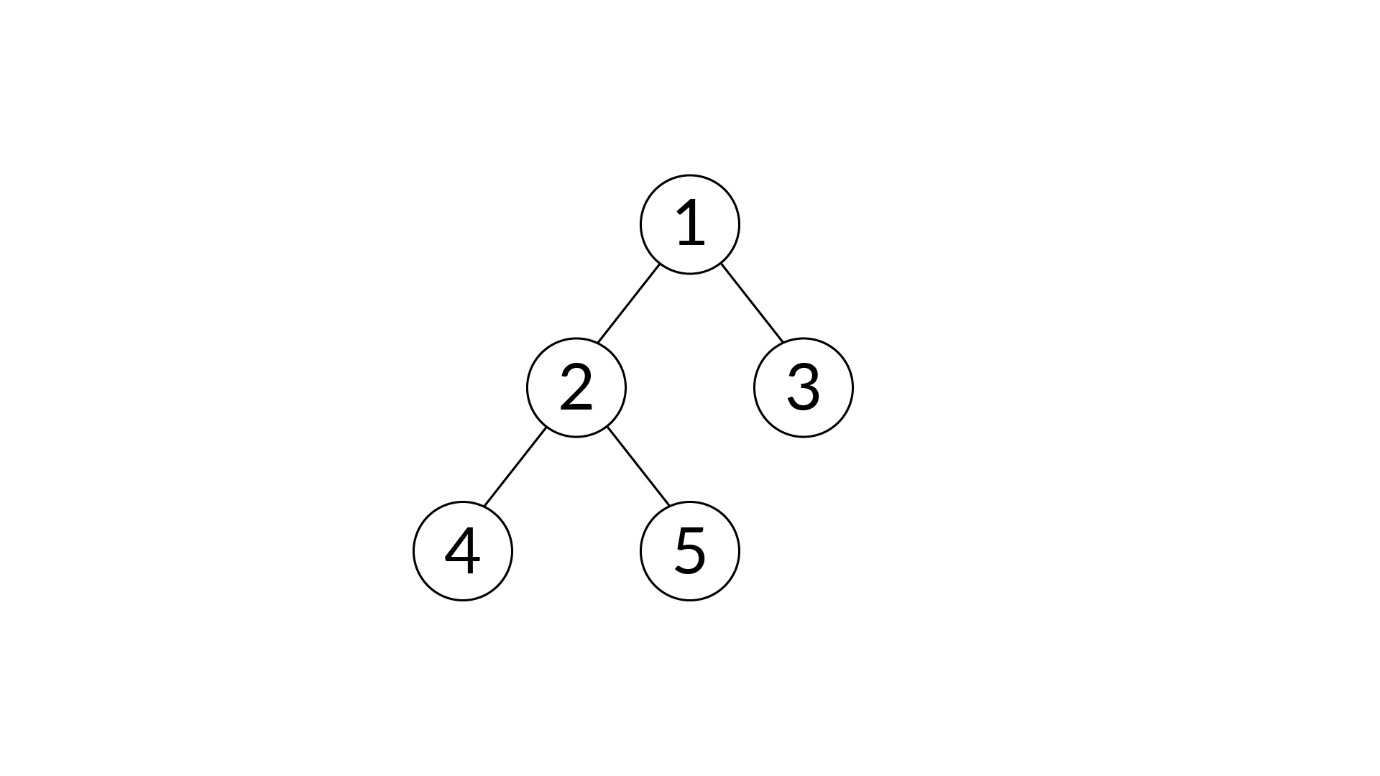


Example

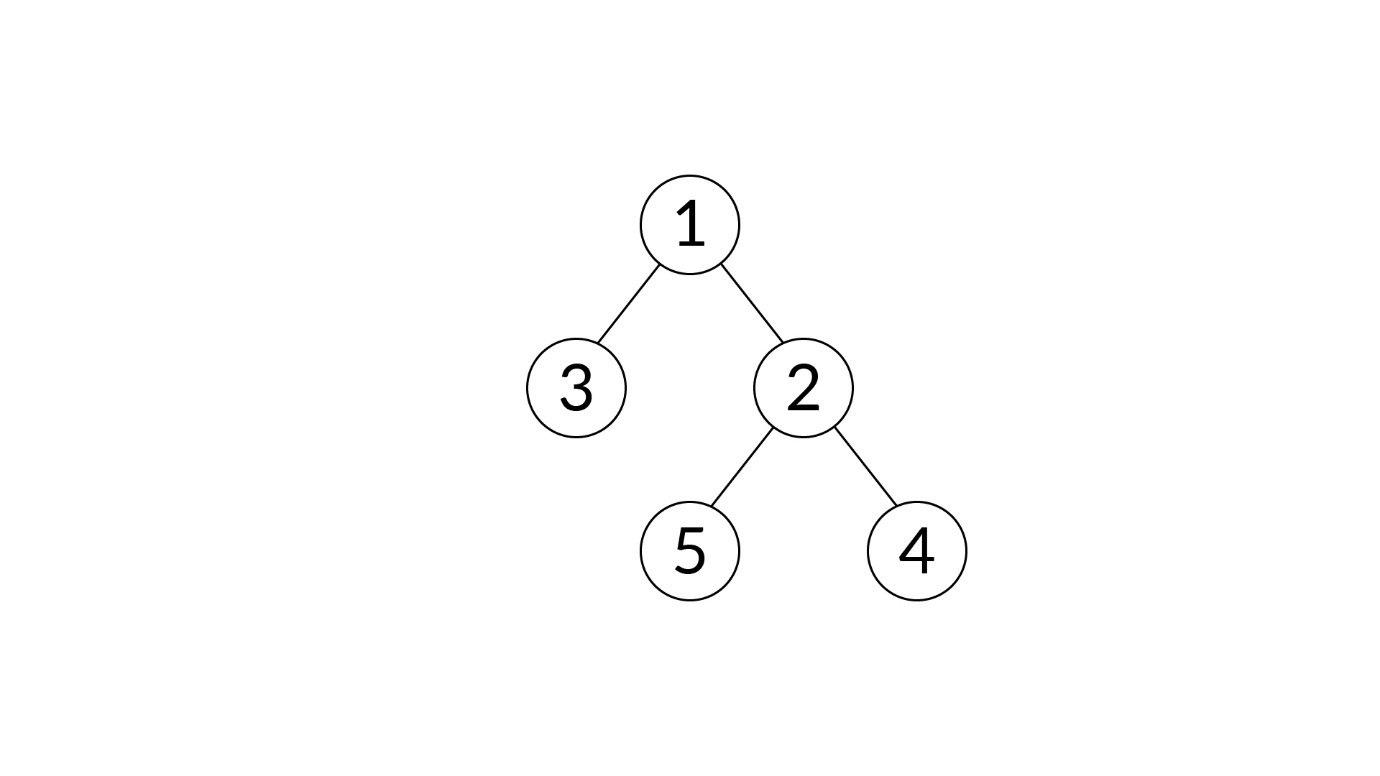
### Code Question: Mirror a Tree

**Description**

Consider the binary tree given below. You are required to convert it into its mirror image.

Image 5-01.jpg 97.24 KB

﻿The mirror image of this binary tree will be:﻿

Image 2-01.jpg 96.56 KB

﻿**Input:**

The input will be in the following format:

1. The first line should be the number ‘n’. Here, the number ‘n’ is the total number of nodes present in the binary tree.
2. The second line contains space-separated ‘n’ values. No value should be equal to -1. This is the level-order traversal of the tree. The code for inserting the values in the tree for the given level order traversal has been addressed in the code.

**Output:**

The output should be the **pre-order traversal**of the **mirror** tree with the nodes **separated by spaces**. The code for pre-order traversal of a tree has been provided to you in the stub code. You only need to complete the code for mirroring a tree.

**Sample Input:**

5

1 2 3 4 5

**Sample Output:**

1 3 2 5 4

**Sample Input:**

6

6 9 8 7 4 5

**Sample Output:**

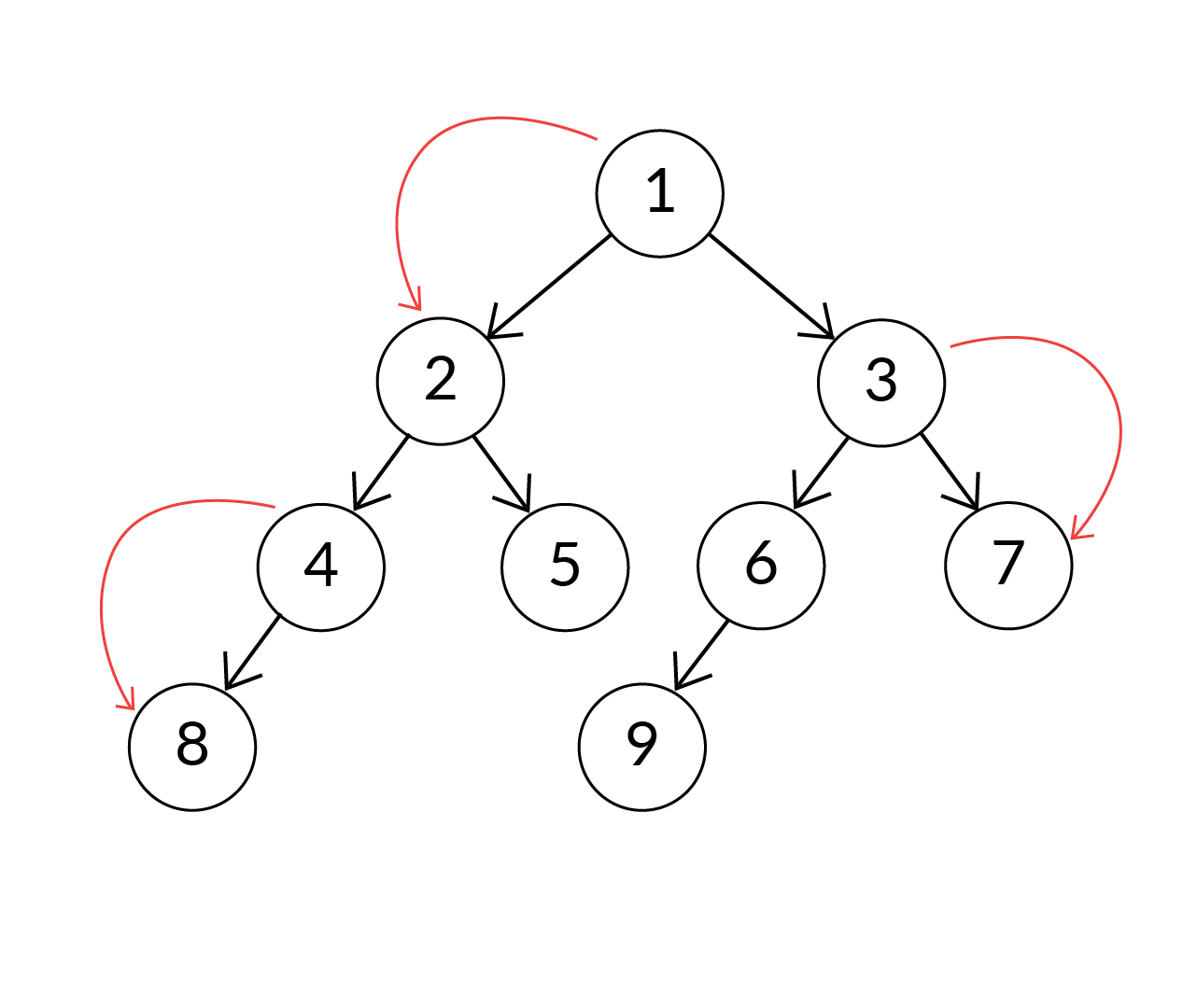
6 8 5 9 4 7

Ans: Intellj

# Spiral Level-Order Traversal

You are given a binary tree and are expected to print the spiral order traversal of the tree, which is equivalent to the level order traversal of the tree in a zig-zag order. For the tree given below, the spiral order traversal is 1, 2, 3, 7, 6, 5, 4, 8, 9.

**Example:**

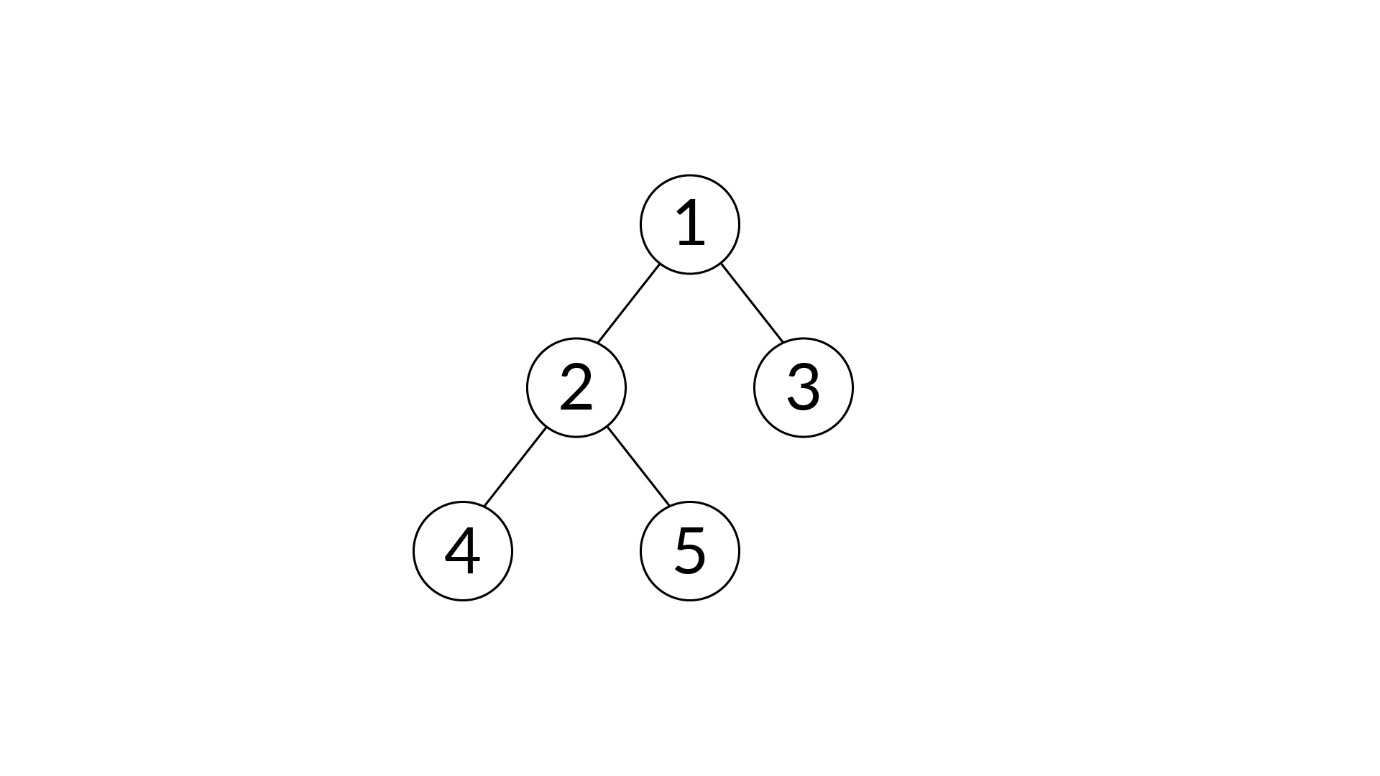


Spiral level-order traversal

### Code ques.: Spiral Level-Order Traversal

**Description**

Consider the binary tree given below. You are required to print the spiral level-order traversal for it.

Image 5-01.jpg 97.24 KB

﻿**Output:**

1 2 3 5 4

**Input:**

The input will be in the following format:

1. The first line should be the number ‘n’. Here, the number ‘n’ is the total number of nodes present in the binary tree.
2. The second line contains space-separated ‘n’ values. This is the level-order traversal of the tree. The code for inserting the values in the tree for the given level-order traversal has been addressed in the code.

**Output:**

The output should be the **spiral level-order traversal**of the tree with the nodes **separated by spaces**.

**Sample Input:**

5

1 2 3 4 5

**Sample Output:**

1 2 3 5 4

**Sample Input:**

**4**

8 2 5 6

**Sample Output:**

8 2 5 6

Ans: Intellij

# Summary

In this segment, you learnt the following:

1. The evolution of tree-based data structures was owing to the non-linearity of data. Every data point in the tree is connected to several other data points such that there is a specific relation between every connection.
2. Tree vocabulary:  
   **Nodes** are all the elements in the tree, with the topmost known as the **root node**. The node from which other nodes descend is the **parent node**, and the descendent is the **child node**. A node with no child is the **leaf node**,and the line that connects two nodes is the**edge.**
3. Then, we discussed binary trees. You learnt that a node in a binary tree can have at the most two children, a left child and a right child.
   1. The **maximum number of nodes** n at level l of a binary tree is2l−1and that in a binary tree of height h is 2h−1.
   2. The **minimum height** hof a binary tree with **n**nodes is log2(n+1).
   3. The **minimum levels** l in a binary tree with L leaves is log2L+1.
4. You then learnt about the following types of binary trees:
   1. **Full binary tree**: Every node has two child nodes, except the leaf nodes.
   2. **Complete binary tree**: All the levels are completely filled (except the last level).
   3. **Perfect binary tree**: All the nodes must have two children nodes, except the leaf nodes, and all the leaf nodes must be at the same level.
5. We then discussed two traversal techniques, which are as follows:  
   1. **Depth-First Search Traversal**: The nodes are visited along the levels.

             a. **Pre-Order Traversal**: First, the node is visited, followed by its left and right subtrees.  
             b. **In-Order Traversal:** First, the node's left subtree is visited, followed by the node and then its right subtree.  
             c. **Post-Order Traversal:** The node's left and right subtrees are visited, followed by the node.  
2. **Breadth-First Search Traversal/Level-Order Traversal** (**Recursive**and **Iterative**approach): Nodes at a certain level are visited before moving on to the next level, i.e., you move along the breadth of the tree.

1. Applications of traversal such as **mirror of a tree** and **spiral level-order traversal** were also discussed in detail.

# Introduction to BSTs

To summarise, the two types of tree data structures that we discussed so far are as follows:

1. **Binary tree:**  
   In this tree, every node can have at the most two children.
2. **Binary search tree:**  
   A binary search tree is also called an ordered binary tree because of the following properties:
   * The values of all the nodes in the left subtree are lower than those of the root node.
   * The values of all the nodes in the right subtree are greater than those of the root node.
   * Each subtree in the binary search tree is itself a binary search tree.

If the concepts are not crystal clear to you, watch the video given below. In the following video, Prof. Sujit will use an example to demonstrate the topics he covered in the previous video. You will also learn how to search in a binary search tree that has a time complexity of O(logn).

#### Q55: Time Complexity for Search

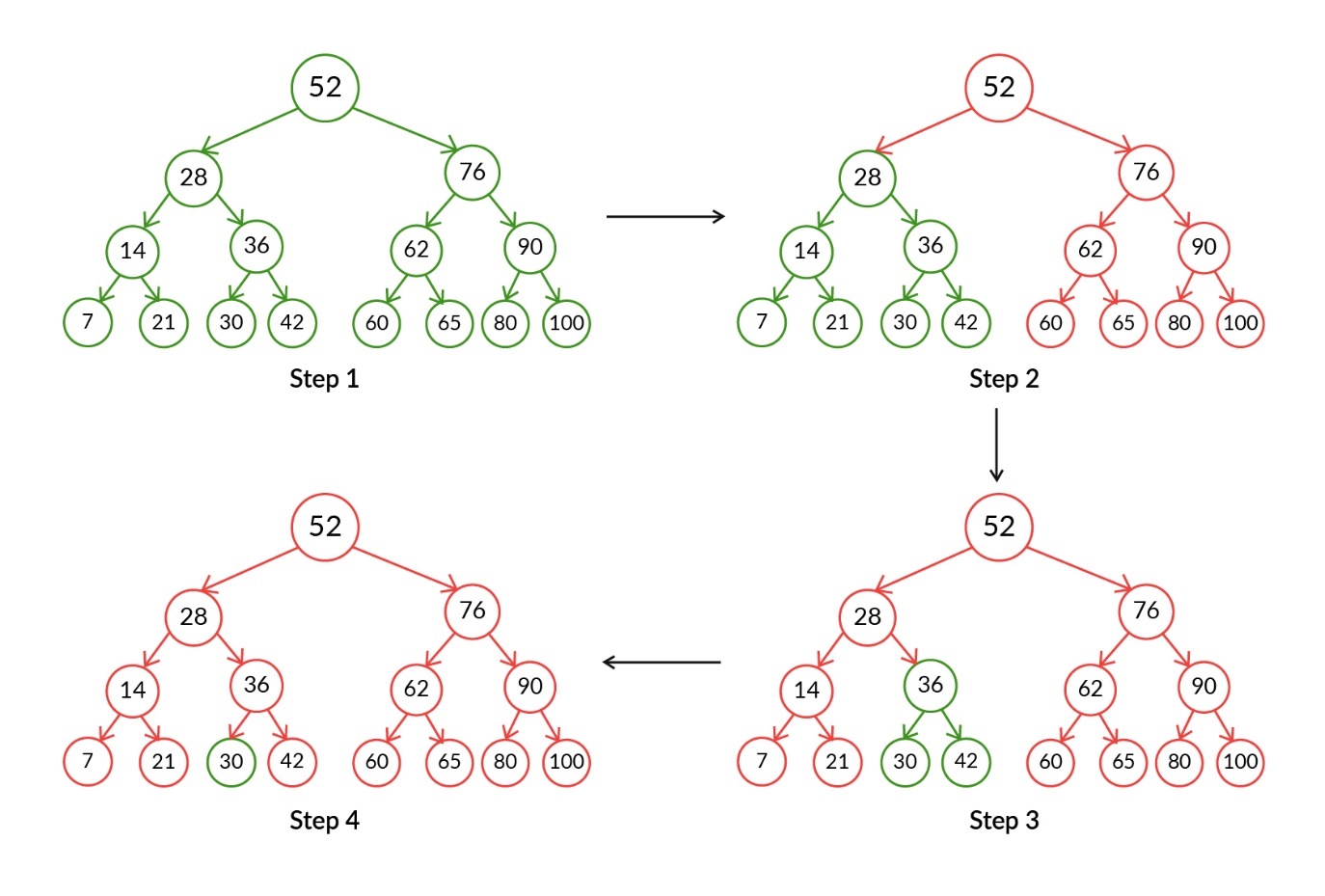
In your own words, explain the reason for the logarithmic or O(log n) efficiency of a search operation in a binary search tree.

Ans: While searching for an element X, you first compare it with the root node. If the element is less than the value of the root node, you move on to the left subtree. In case the element is greater than the value of the root node, you move on to the right subtree. In either of the cases, you are reducing your search space by half. When you move on to either child of the root node, you compare X with it, and again, you move to its left or right depending on whether the value to be searched for is less than or greater than the value of the child node. Once again, the other half is ignored. So, at every step, one half of the sample space is ignored, which makes the operation an O(log n) operation.

# Searching in BST

#### Q56: Binary Search Trees

What are the steps to search for the element 30 in the tree given below (illustrated in step 1)?  
Hint: Go through the images given below for steps 1–4.

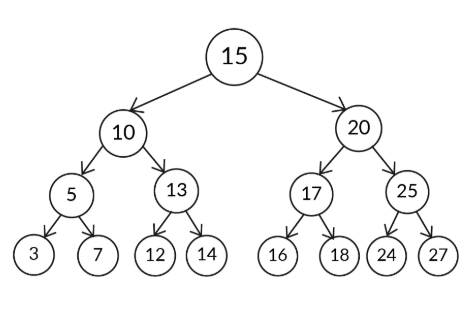


Ans: The following points explain the steps to search the element 30 in the image given above:

1. First, check whether the root node (52) is equal to the element to be searched (30).
   1. Here, it is not equal, and so, you need to search in the left or right subtree of the root node 52.
   2. If the element to be searched is not the root node, then it should be less than or greater than the root node.
   3. In the binary search tree, all the elements to the left of any node will be less than that node. All the elements to the right of any node will be greater than that node.
   4. Here, 30 is less than 52; so, you need to search for the left subtree of 52 because all the elements that are less than 52 will be present in the left subtree of node with value 52.
2. Now, check whether the left child of 52, which is 28, is equal to 30 or not.
   1. 30 is greater than the left child of 52.
   2. According to the property of binary search tree, all the elements greater than 28 will be in the right subtree of 28. 30, which is greater than 28, can be found only in the right subtree of 28.
   3. So, search for 30 in the right subtree of 28.
3. Now, check whether the right child of 28, which is 36, is equal to 30 or not.
   1. 30 is less than the right child of 28, which is 36.
   2. According to the property of the binary search tree, all the elements less than 36 will be in the left subtree of 36, and 30, which is less than 36, can be found only in the left subtree of 36.
   3. So, search for 30 in the left subtree of 36.
4. Now, check whether the left child of 36 is equal to 30 or not.
   1. The left child of 36 is 30, which is equal to the element to be searched for. Therefore, you have found the desired element.
5. In the previous video, you took a look at the code that is used to search for a node with the given value in the given BST.
7. **Code Snippet:**
8. // Method to search for a node with given key in a BST
9. **boolean** **search**(Node node, **int** key) {
10. **if** (node == **null**)
11. **return** **false**;
12. **if** (key == node.data)
13. **return** **true**;
14. **if** (key < node.data)
15. **return** search(node.left, key);
16. **else**
17. **return** search(node.right, key);
18. }

#### Q58: Binary search trees

How many nodes of the binary search tree lie in the range of 10–20 (including both 10 and 20)?



Ans:   
9

**✓ Correct**

**Feedback:**

In a binary search tree, all the elements in the left subtree are less than the root node, and all the elements in the right subtree are greater than the root node. So, to find the nodes in the range 10–20, you start from the root node and then traverse looking for elements more than 10 and less than 20 and maintain the counter accordingly. Therefore, 3 nodes in the right subtree of 10; 3 in the left subtree of 20, nodes 10 and 20; and the root node with value 15 make 9 the correct answer.

#### Q59: Binary search trees

Write the pseudocode of a function to return the number of nodes of the binary search tree that lie in the range a–b (including both a and b)?  
Hint: Use recursion

Ans: /node: passed node, initially it is the root of the binary search tree

//lower: lower limit of a given range

//upper: upper limit of a given range

return\_count(node, lower, upper)

1. If the node is null, then return 0.
2. If the current node is in the range, then increment the count and recur for the left and right children of the current node. This can be done by the following:
   1. Return 1+return\_count(node.left, lower, upper)+return\_count(node.right, lower, upper)
3. If the current node is greater than the upper limit of the given range, then recur for the left child of the current node. This can be done by the following:
   1. return return\_count(node.left, lower, upper)
4. Otherwise, recur for the right child of the current node. This can be done by the following:
   1. return return\_count(node.right, lower, upper)

# Inserting Node in BST

So, whenever a new node is added to a BST, it has to be added as a leaf node. It cannot be an internal node. Also, adding a new node to a BST is similar to searching for a node in the BST. Hence, the ‘insert’ operation follows a run-time efficiency of O(logn).

Now that you have an idea about how to insert a node in a BST, in the next video, Srishti will discuss the pseudocode for the same.

Now that you know the pseudocode for adding a node in BST, we recommend that you try writing its code on your own. If you are stuck or want to match your solution against ours, watch the video given below in which Srishti will explain this.

In the previous video, you took a look at the code in Java to insert a node in a given BST. The following is the insert method that you learnt:

**Code Snippet:**

// Method to insert a node with given key in a BST

Node **insert**(Node node, **int** key) {

**if** (node == **null**) {

node = **new** Node(key);

**return** node;

}

**if** (key < node.data)

node.left = insert(node.left, key);

**else** **if** (key > node.data)

node.right = insert(node.right, key);

**return** node;

}

#### Q59: Adding a New Node

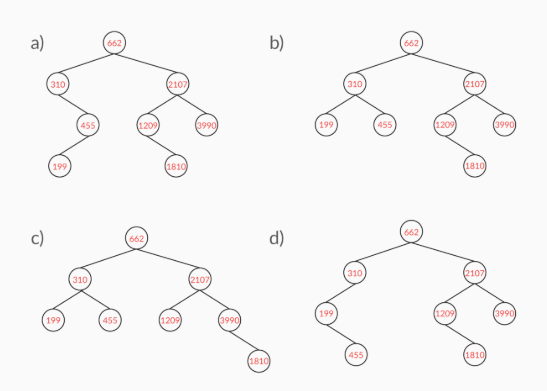
You learnt how to add a new node to a BST. Can you predict the efficiency of this operation in BST? Explain your answer in the box given below.

Ans: While adding a node, you need to find the correct place for it in BST. Doing this is almost similar to the the steps you took when you used the search algorithm. It is similar to searching for an element in the BST that you are supposed to add. Hence, it follows a run-time efficiency of O(logn).

#### Q60: Dynamic Binary Search Results

Suppose a customer wants to buy a pair of shoes within their budget from an e-commerce site that sells footwear. If the site displays its products in the ascending order of their prices, then it will be easy for the customer to select the desired product. Hence, the site owner wants to build a binary search tree of the prices. So, the user can easily get the sorted order of the products by performing an in-order traversal.

Suppose the owner wants to add the following prices to the binary search tree. Choose the tree formed after inserting the prices in the following order: 662, 310, 455, 199, 2107, 3990, 1209, 1810.  
  
(The root node is assumed to be 662.)



Ans: b

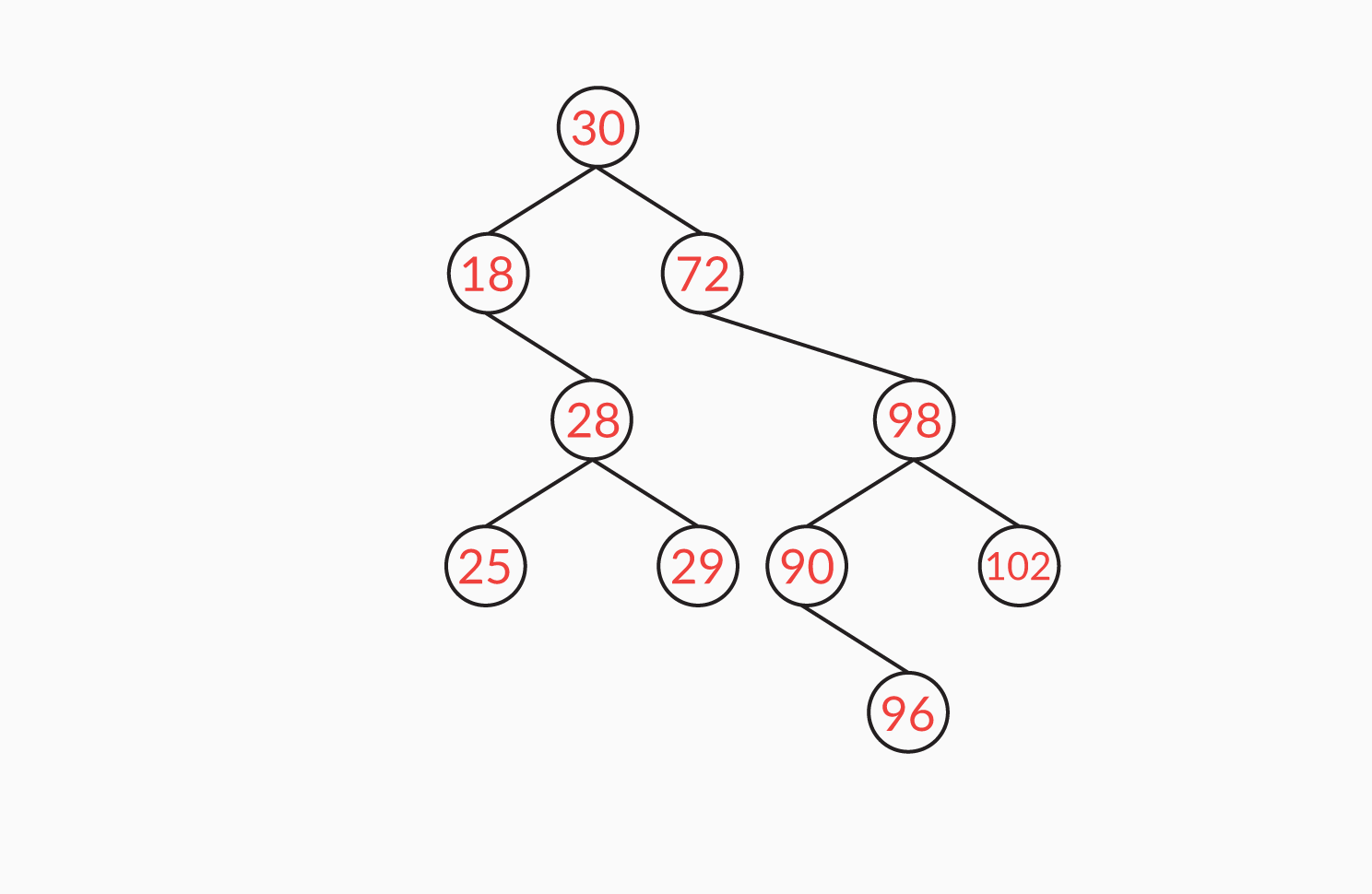
**✓ Correct**

**Feedback:**

You can start by taking the first price in the order as the root. Then, all the prices that are less than the root will be towards the left, and all the prices greater than the root will be towards the right. Once you find a suitable leaf node, you can insert the prices accordingly. In this case, the binary search tree formed satisfies this condition.

#### Q61: Correct Order of BST

A BST is given below. Select the correct order of insertion of elements into the tree from the options.



Hint: Remember that a parent node needs to be inserted before a child node. Hence, a parent node will appear before a child node in the given order.

Ans: 30, 72, 18, 98, 28, 29, 25, 102, 90, 96

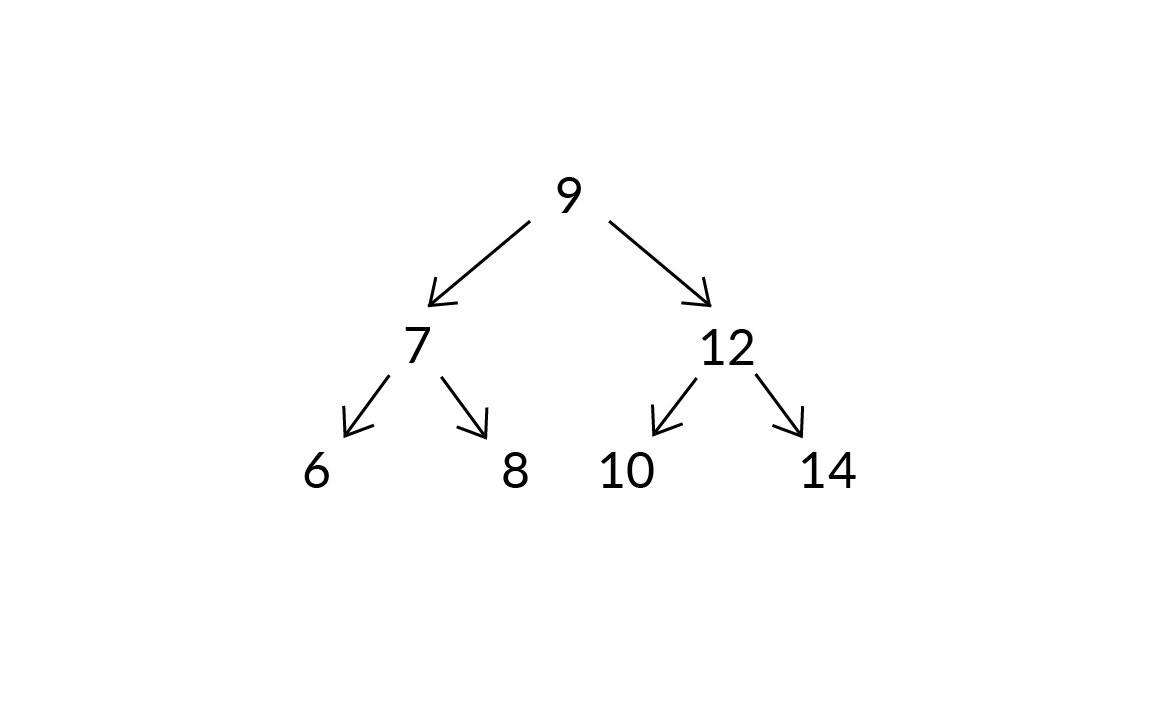
**✓ Correct**

**Feedback:**

Start with the sequence of the elements given and build the tree by inserting each element in the given order. Then, match that tree with the given binary search tree. In this case, the tree that is formed matches the given binary search tree.

#### Q63: Binary search trees

Given below is a binary search tree. How many nodes can you add to the node ‘6’ provided that all the nodes in the binary search tree are positive integers?



Ans:   
5

**✓ Correct**

**Feedback:**

Here, 6 is to the left of 7; so, all the children of 6 should be less than 7. From this, you can observe that there will not be any right child for 6 because the right child of 6 should be greater than 6 and there are no positive integers between 6 and 7. The only possibility is to add the node to the left of 6 so that all the nodes in the left subtree of 6 should be less than 6, and the number of positive integers less than 6 is 5. Therefore, this option (5) is the correct choice.

# Deleting a Node from BST (Do not publish)

**Please visit the link**[**here**](http://btv.melezinek.cz/binary-search-tree.html)**that contains an extremely helpful visual of a binary search tree. Here, you can create your own BST and play around with it with respect to all the operations such as adding a node, deleting a node and searching for a value. The visual will certainly reinforce your understanding.**

To summarise, the deletion of data from binary search trees can be carried out in the following three ways:

1. If the node to be deleted is a leaf node, it is directly removed from the tree.
2. If the node has one child, the node itself is deleted, and its child node is connected to its parent node.
3. If the node has two children, you find its successor to replace it. The successor node is the minimum node in the right subtree or the maximum node in the left subtree.

In the previous video, you took a look at the code in Java to delete a node from BST. The definitions of the methods that you learnt are given below.

**Code Snippet:**

**package** **bintree**;

**public** **class** **Node**<T> {

**protected** T value;

**protected** Node<T> parent;

**protected** Node<T> left;

**protected** Node<T> right;

**public** **Node**(T value, Node<T> parent, Node<T> left, Node<T> right) {

**this**.value = value;

**this**.parent = parent;

**this**.left = left;

**this**.right = right;

}

**public** T **getValue**() {

**return** **this**.value;

}

**public** Node<T> **getParent**() {

**return** **this**.parent;

}

**public** Node<T> **getLeft**() {

**return** **this**.left;

}

**public** Node<T> **getRight**() {

**return** **this**.right;

}

**public** String **toString**() {

**return** **this**.value.toString();

}

**public** **void** **setValue**(T value) {

**this**.value = value;

}

**public** **void** **setParent**(Node<T> node) {

**this**.parent = node;

}

**public** **void** **setRight**(Node<T> node) {

**this**.right = node;

}

**public** **void** **setLeft**(Node<T> node) {

**this**.left = node;

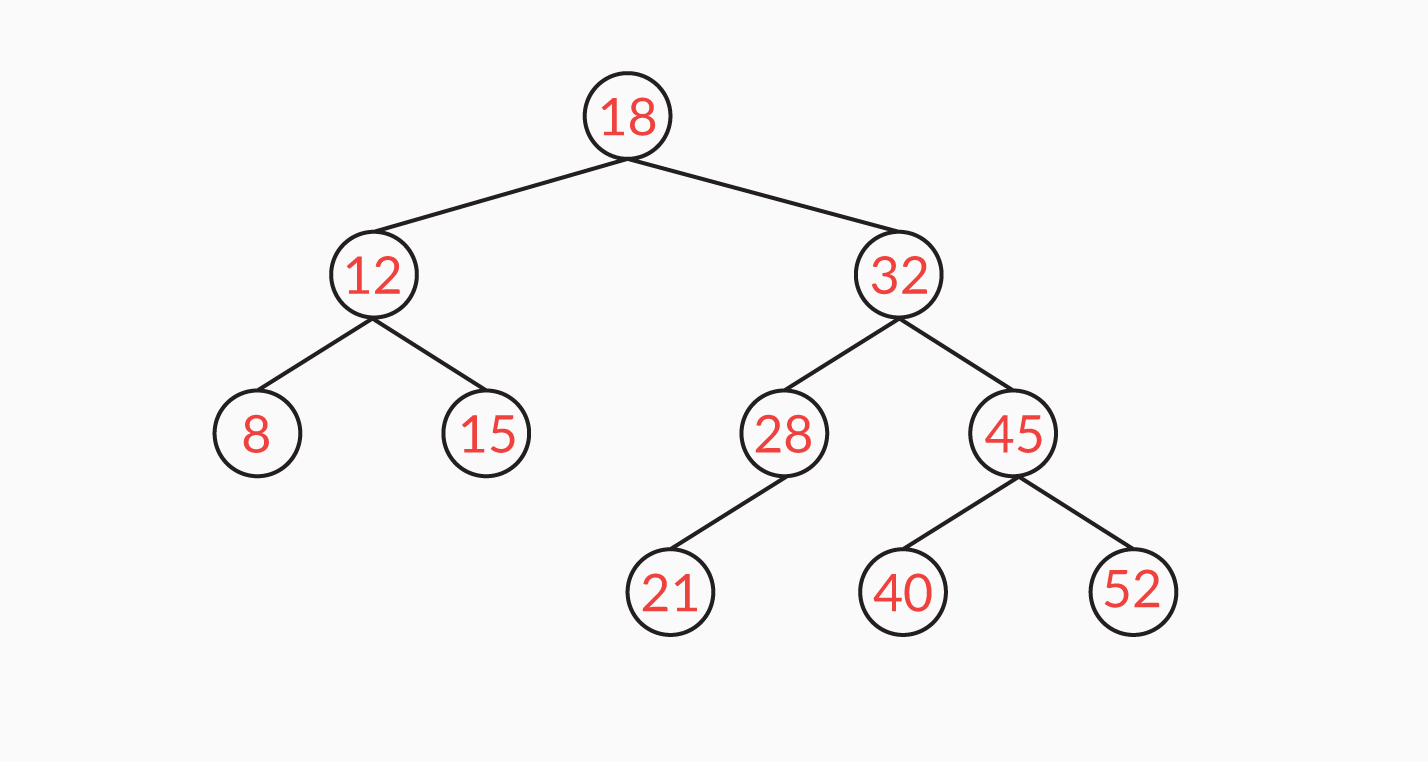
}

}

#### Q64: Dynamic Binary Search

Suppose in the given tree, node 32 is deleted. Which node can replace it? (Note: More than one option may be correct.)

Hint: Recall the rule of removing the node with two children.



Ans: 28

**✓ Correct**

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '28' is the maximum element in the left subtree. So, node '32' can be replaced by node '28'. Thus, this is one of the correct options.

40

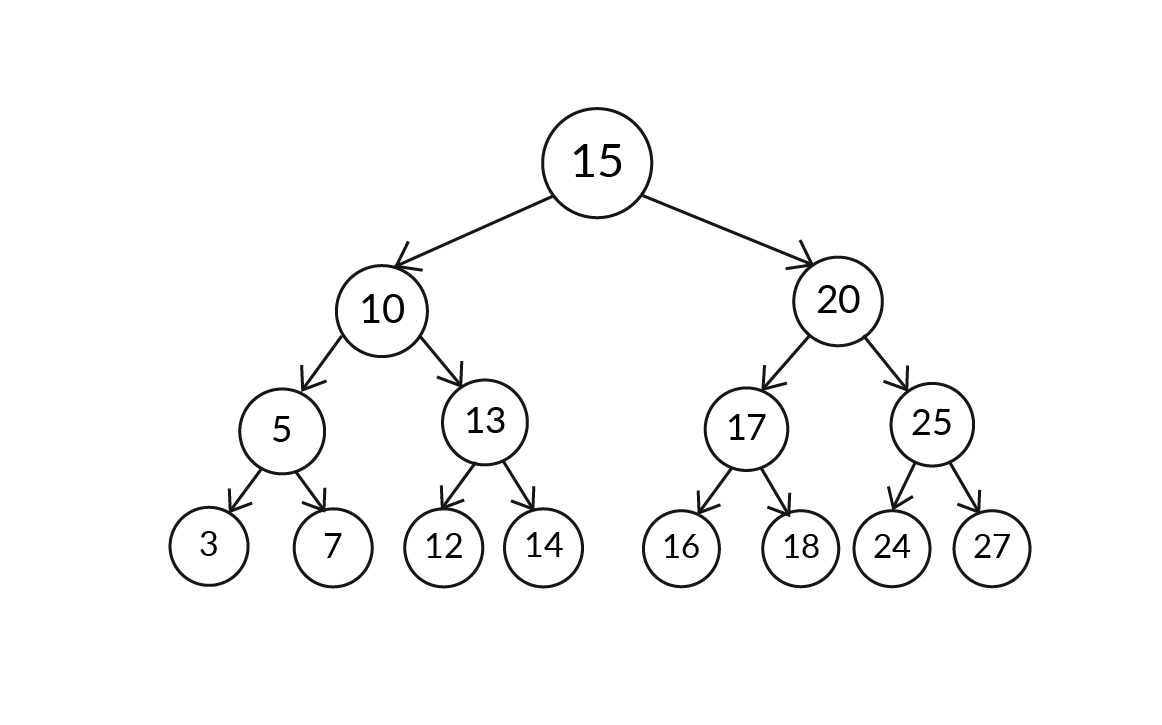
**✓ Correct**You missed this!

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '40' is the minimum element in the right subtree. So, node '32' can be replaced by node '40'. Thus, this is one of the correct options.

#### Q65: Binary search trees

Suppose in the given tree, node 15 is deleted. Which node can replace it? (Note: More than one option may be correct.)



Ans: 14

**✓ Correct**

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '14' is the maximum element in the left subtree. So, node '15' can be replaced by node '14'.



16

**✓ Correct**You missed this!

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '16' is the minimum element in the right subtree. So, node '15' can be replaced by node '16'.

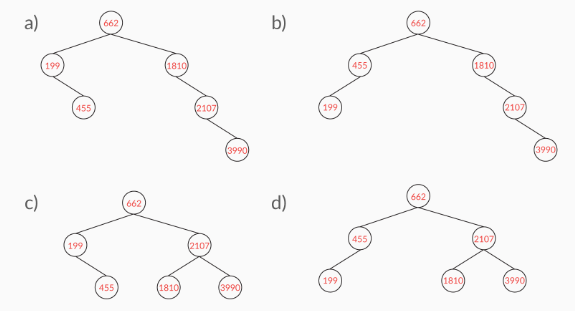
#### Q66: Dynamic Binary Search

Suppose a customer wants to buy a pair of shoes within their budget from an e-commerce site that sells footwear. If the site displays its products in the ascending order of their prices, then it will be easy for the customer to select the product they want. Hence, the site owner wants to build a binary search tree of the prices so that the user can easily get the sorted order of the products by performing an in-order traversal.

The binary search tree is built after inserting the prices in the following order: 662, 310, 455, 199, 2107, 3990, 1209, 1810; take 662 as the root node.

Now, after building the binary search tree, the owner wants to remove the outdated models. Hence, they want to delete the prices in the binary search tree. Choose all the trees that can be formed after removing the prices 1209 and 310 one after another.

Note: The prices are removed one after another. So, solve the problem sequentially to obtain the correct answer, i.e., remove the price 310 from the tree that results after removing the price 1209.



Ans: d

**✓ Correct**You missed this!

**Feedback:**

If the price to be deleted is a leaf node, it can be removed directly from the tree. If it has one child, then the node is deleted, and its child node is connected to its parent price node. If a price node that has two children is to be deleted, it is replaced by the minimum element in its right subtree or the maximum element in its left subtree. Based on this condition, in this case, the price '310' can be replaced by '455' because it is the minimum element of the right subtree of price '310'. Also, the price '1810' replaces '1209' to become the left child of '2107'.

c

**✓ Correct**You missed this!

**Feedback:**

If the price to be deleted is a leaf node, it can be removed directly from the tree. If it has one child, then the node is deleted, and its child node is connected to its parent price node. If a price node that has two children is to be deleted, it is replaced by the minimum element in its right subtree or the maximum element in its left subtree. Based on this condition, in this case, the price '310' can be replaced by '199' because it is the maximum element of the left subtree of price '310'. Also, the price '1810' replaces '1209' to become the left child of '2107'.

# Deleting a Node from BST

Please visit the link [here](http://btv.melezinek.cz/binary-search-tree.html) that contains an extremely helpful visual of a binary search tree. Here, you can create your own BST and play around with it with respect to all the operations such as adding a node, deleting a node and searching for a value. The visual will certainly reinforce your understanding.

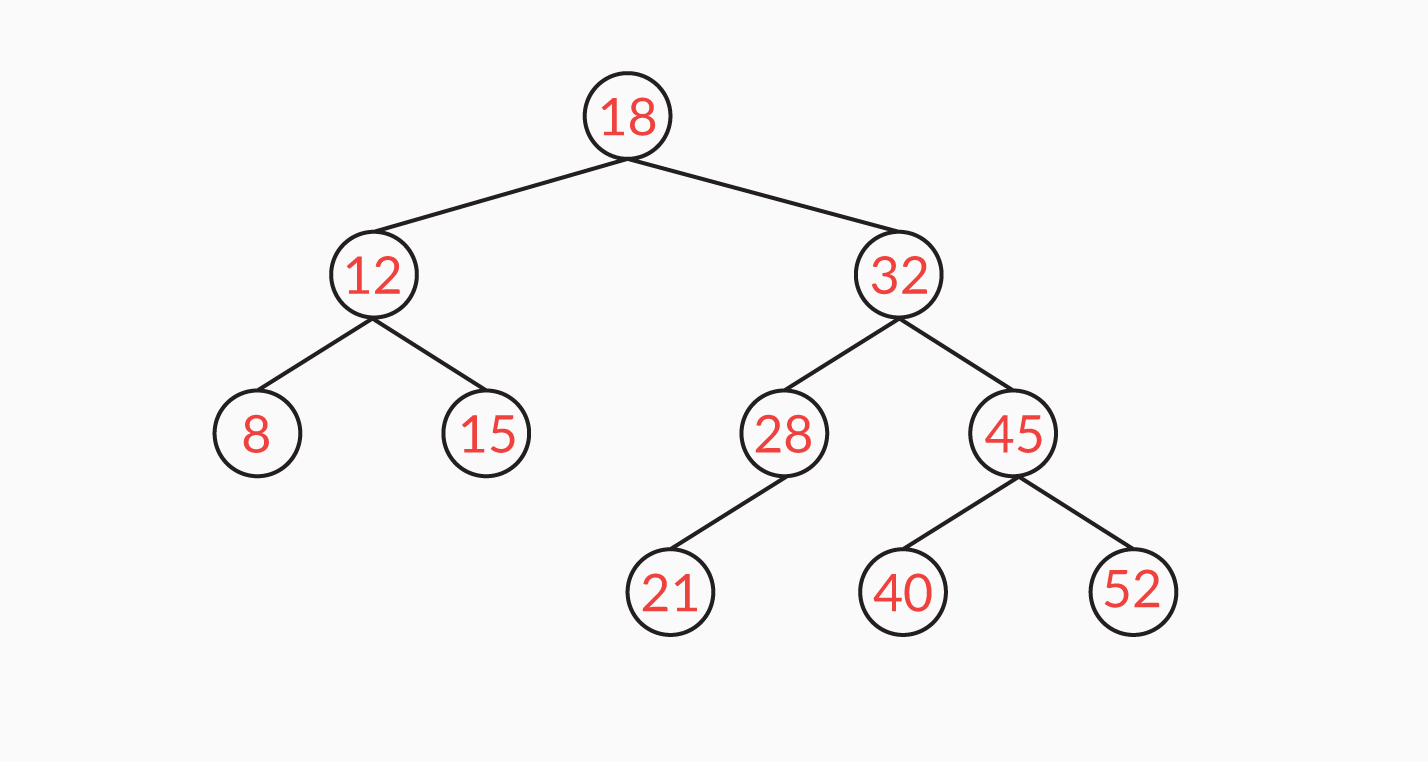
To summarise, the data from BST can be deleted in the following three ways:

1. If the node to be deleted is a leaf node, it is directly removed from the tree.
2. If the node has one child, the node itself is deleted and its child node is connected to its parent node.
3. If the node has two children, you find its successor to replace it. The successor node is the minimum node in the right subtree or the maximum node in the left subtree.

#### Q66: Dynamic Binary Search

Suppose in the given tree, node 32 is deleted. Which node can replace it? (Note: More than one option may be correct.)

Hint: Recall the rule of removing the node with two children.



Ans: 28

**✓ Correct**

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '28' is the maximum element in the left subtree. So, node '32' can be replaced by node '28'. Thus, this is one of the correct options.

40

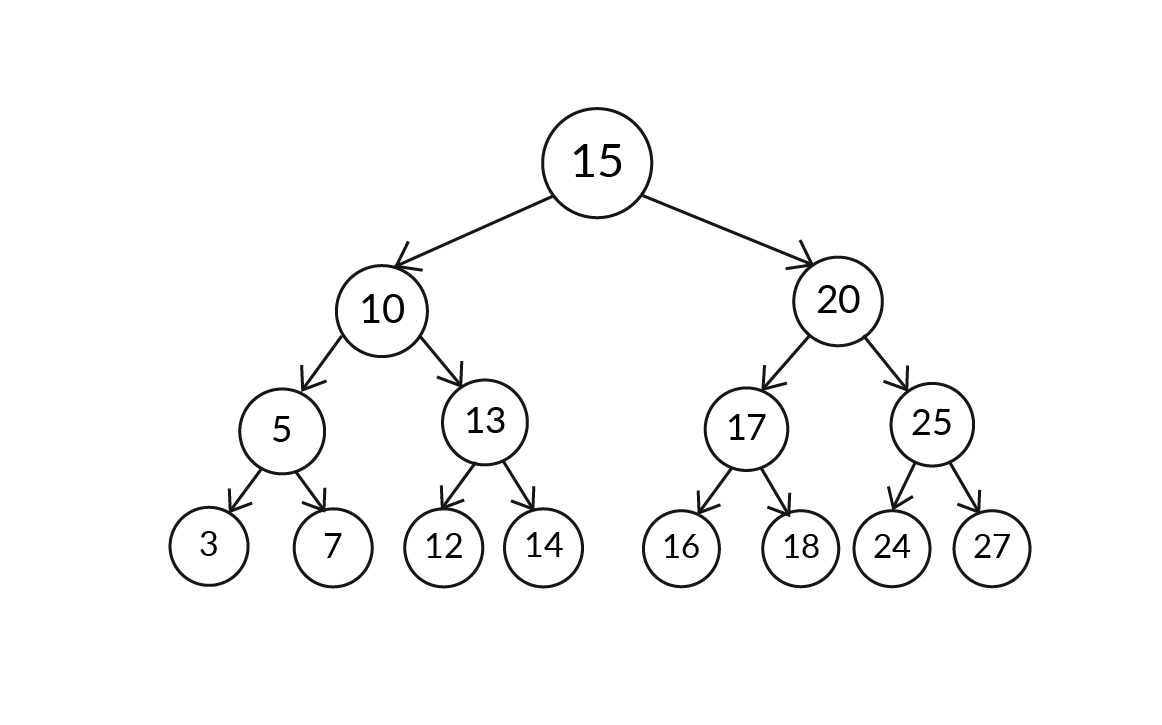
**✓ Correct**

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '40' is the minimum element in the right subtree. So, node '32' can be replaced by node '40'. Thus, this is one of the correct options.

#### Q67: Binary search trees

Suppose in the given tree, node 15 is deleted. Which node can replace it? (Note: More than one option may be correct.)



Ans: 14

**✓ Correct**

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '14' is the maximum element in the left subtree. So, node '15' can be replaced by node '14'.



16

**✓ Correct**

**Feedback:**

If a node with two children is to be deleted from a binary search tree, it can be replaced by the minimum element in its right subtree or the maximum element in its left subtree. In this case, '16' is the minimum element in the right subtree. So, node '15' can be replaced by node '16'.

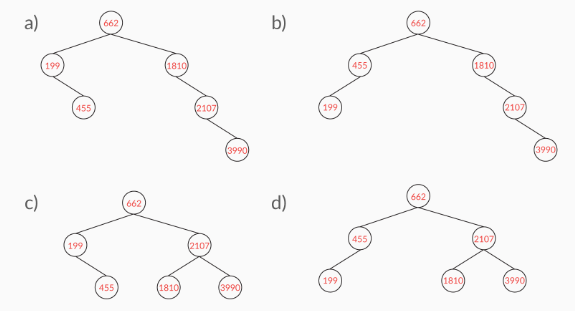
#### Q68: Dynamic Binary Search

Suppose a customer wants to buy a pair of shoes within their budget from an e-commerce site that sells footwear. If the site displays its products in the ascending order of their prices, then it will be easy for the customer to select the product they want. Hence, the site owner wants to build a binary search tree of the prices so that the user can easily get the sorted order of the products by performing an in-order traversal.

The binary search tree is built after inserting the prices in the following order: 662, 310, 455, 199, 2107, 3990, 1209, 1810; take 662 as the root node.

Now, after building the binary search tree, the owner wants to remove the outdated models. Hence, they want to delete the prices in the binary search tree. Choose all the trees that can be formed after removing the prices 1209 and 310 one after another.

Note: The prices are removed one after another. So, solve the problem sequentially to obtain the correct answer, i.e., remove the price 310 from the tree that results after removing the price 1209.



Ans: d

**✓ Correct**You missed this!

**Feedback:**

If the price to be deleted is a leaf node, it can be removed directly from the tree. If it has one child, then the node is deleted, and its child node is connected to its parent price node. If a price node that has two children is to be deleted, it is replaced by the minimum element in its right subtree or the maximum element in its left subtree. Based on this condition, in this case, the price '310' can be replaced by '455' because it is the minimum element of the right subtree of price '310'. Also, the price '1810' replaces '1209' to become the left child of '2107'.

c

**✓ Correct**You missed this!

**Feedback:**

If the price to be deleted is a leaf node, it can be removed directly from the tree. If it has one child, then the node is deleted, and its child node is connected to its parent price node. If a price node that has two children is to be deleted, it is replaced by the minimum element in its right subtree or the maximum element in its left subtree. Based on this condition, in this case, the price '310' can be replaced by '199' because it is the maximum element of the left subtree of price '310'. Also, the price '1810' replaces '1209' to become the left child of '2107'.

# Lowest Common Ancestor in BST

The lowest common ancestor of nodes n1 and n2 is the lowest possible node in the tree whose descendants include nodes n1 and n2.

Note:

1. Descendants of a node are all the nodes that are derived from that node.
2. Consider that the node is a descendant of itself.

Replay

Mute

Current Time 2:54

/

Duration 2:54

Loaded: 100.00%

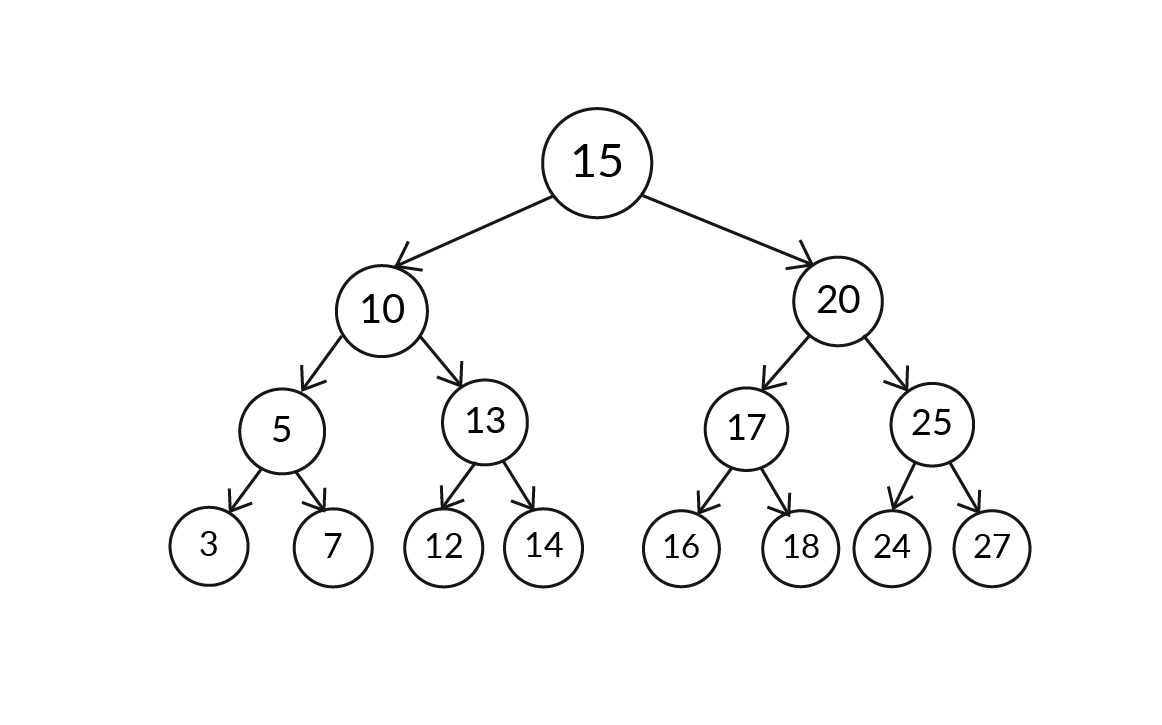
1x

Playback Rate

Quality Levels

Picture-in-PictureFullscreen

3776789



BST

If this BST is given, then the common ancestor of nodes 5 and 13 is 10 because 10 is the lowest possible node in the tree wherein its descendants include node 5 and node 13.

The lowest common ancestor of nodes 20 and 24 is 20 because node 20 is a descendant of itself, 24 is a descendant of 20 and node 20 is the lowest possible node in the tree whose descendants are node 20 and node 24.

#### Q69: Lowest common ancestor in BST

What approach will you follow to find the lowest common ancestor (LCA) of any two nodes in BST?  
Hint: Use BST properties

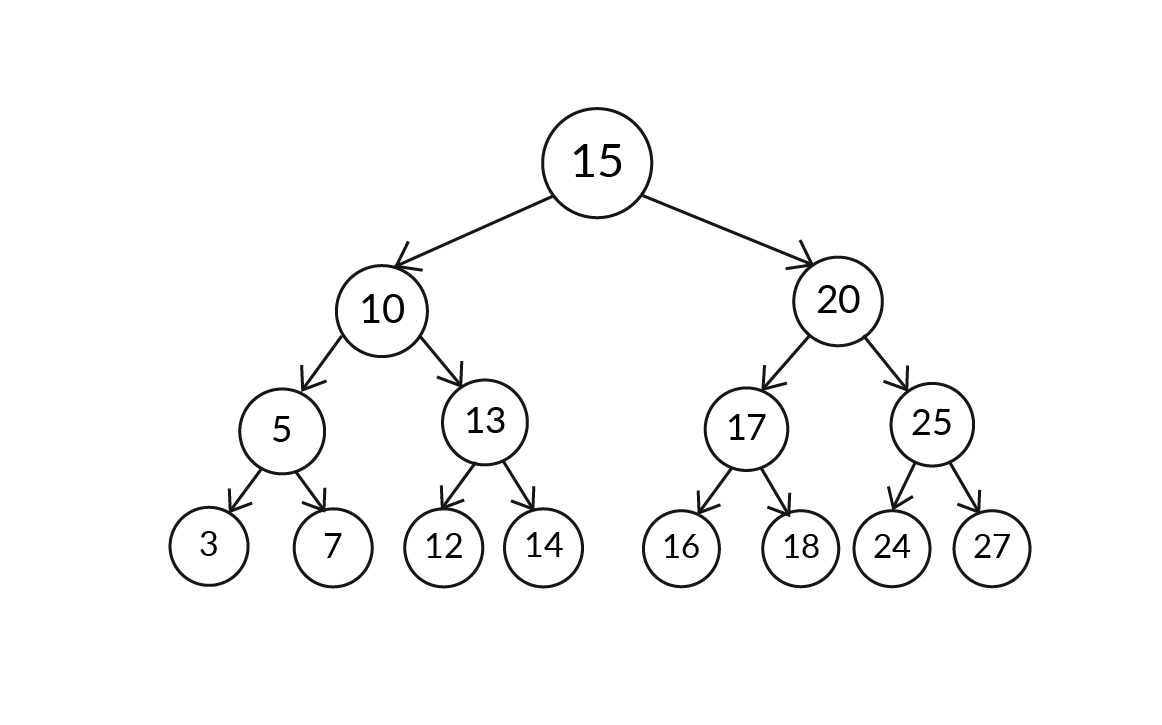
Ans: Suppose you want to find the lowest common ancestor of nodes n1 and n2.

Let's assume the root node as 'root'.

1. Recursively traverse through the BST from the root node.
2. If the root is greater than both n1 and n2, then recursively traverse through the left subtree of the root node to find out the LCA of the two given nodes because the LCA  of the given nodes lies in the left subtree.
3. If the root node is less than both n1 and n2, then recursively traverse through the right subtree of the root node to find out the LCA of the two given nodes because the LCA of the given nodes lies in the right subtree.
4. Otherwise, the root itself is the lowest common ancestor of the binary search tree.

Gain an understanding of the approach given in the suggested answer to the question given above with an example.

Question: What is the LCA of 5 and 14?



Tree

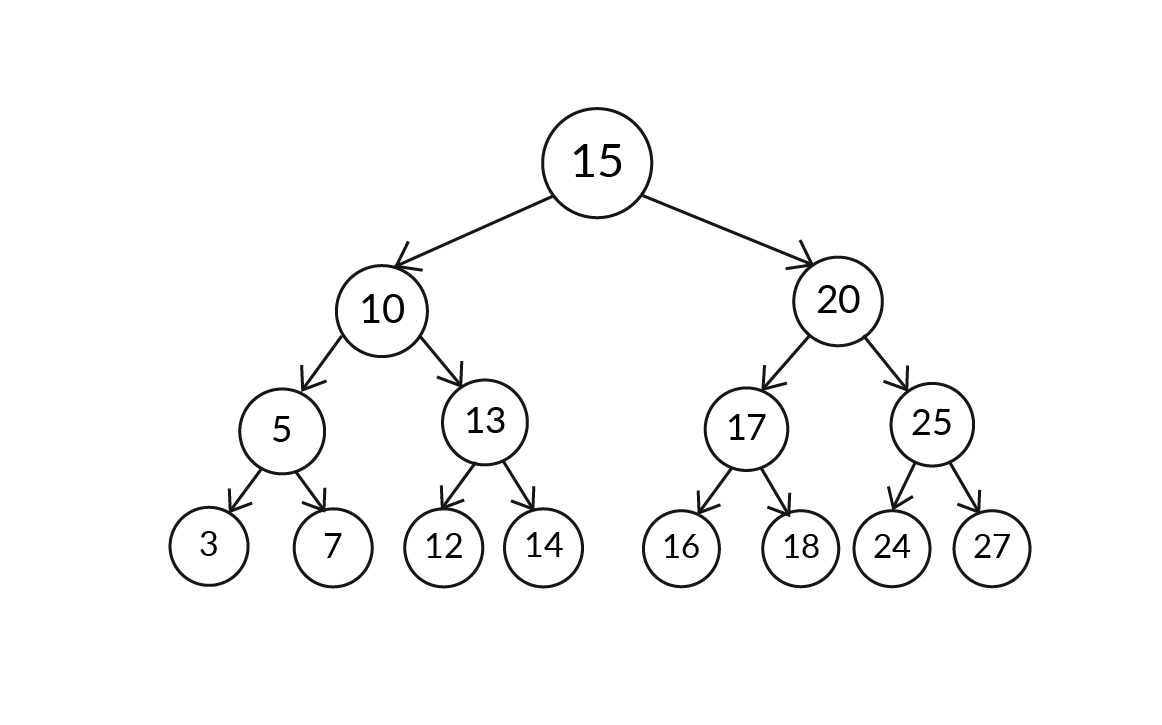
1. Traverse through the root node of the tree.
2. The root node 15 is the common ancestor of 5 and 14 because the descendants of 15 contain nodes 5 and 14, but it is not the lowest common ancestor of node 5 and node 14. (Here, you can observe that 5 and 14 are less than node 15.)
3. Based on the property of BST, you need to traverse through the left subtree of the root node, as both nodes 5 and 14 are less than 15 (root node). The first left descendant of the root node is node 10.
4. Node 10 is the common ancestor of nodes 5 and 14. To get the lowest common ancestor. you can traverse through the tree further, but it does not give you the common ancestor of nodes 5 and 14; hence, the traversal stops here. (Also, you can observe that 10 is not greater than both 5 and 14, and 10 is not less than both 5 and 14.)
   * Since it is not possible to go further below, return 10 as the LCA of nodes 5 and 14.

**Try to write the pseudocode for printing the LCA of two nodes in a given BST on your own.**  
  
After writing the pseudocode, you can watch the video given below in which Srishti will explain how to print the lowest common ancestor of two nodes in a given BST.

In the next video, you will dry run the pseudocode while looking at an example.

#### Q70: Lowest common ancestor in BST

What is the LCA of nodes 17 and 18 in the BST given below?



Ans: 17

**Suggested Answer**

1. Traverse through the root node.
2. Here, 17 and 18 are greater than the node 15. So, traverse through the right side of the root node, which is node  20 here.
3. Nodes 17 and 18 are less than node 20. So, traverse through the left side of node 20, which is node 17 here.
4. Node 18 is a descendant of node 17 in the given BST, and node 17 is a descendant of itself.
5. Therefore, node 17 is the LCA of node 17 and node 18.

#### Q71: Lowest common ancestor in BST

What is the worst-case time and space complexity of the approach given above, respectively?

Note:

1. ‘n’ is the number of nodes in the tree.
2. ‘h’ is the height of the tree (recall the definition of the height of the binary tree discussed in the previous session).

Ans: O(h) and O(h)

**✓ Correct**

**Feedback:**

At any time throughout the algorithm, you are either traversing through the left child or through the right child of any node. In the worst case, if both nodes n1 and n2 are leaf nodes, then you need to traverse through the complete height of the tree, not the entire tree. Therefore, the time complexity of the approach given above is O(h). The worst case space complexity is the O(h) space because the recursion stack uses it to store the h function calls in the stack during recursion.

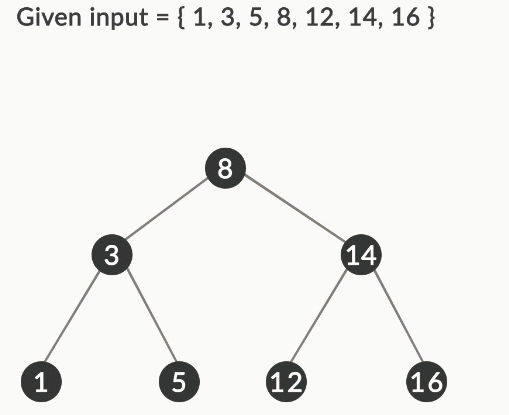
# Balanced BSTs

Apart from handling storage, binary search trees also need to process data efficiently. Sometimes, storing data in a given order can lead to an unbalanced binary search tree or a linked list in the worst case. For example, imagine adding the values 1, 2, 3, 4 and 5 in this order to create a binary search tree. The final tree will look like a linked list.

So, is there a particular method that helps in improving the search operation efficiency of binary search trees? In this lecture, Prof. Sujit will explain how you can optimise search operations in binary search trees.

In the video, you observed that if data used to create a tree that has values in the increasing or decreasing order, the resultant tree will effectively be a linked list, and the efficiency of the search operation will become O(n). So, preprocessing data can improve the search efficiency of binary search trees.

Hence, the sorted data can be stored by choosing the median of all the elements as the root of the tree and iterating the same for all the subtrees, as shown in the figure given below.



BST

Here, 8 is the middle element (median). So, 8 is selected as the root of the BST. The middle element of the left part of the array is 3 (median). So, 3 is the root of the left subtree. The elements 1 and 5 will become its left and right children, respectively. Similarly, 14 will be the middle element (median) of the right part of the array or the root node of the right subtree, and nodes with values 12 and 16 will become its left and right child, respectively.

Storing the data in this way will always produce balanced binary search trees and, in turn, will enhance the efficiency of the search process to O(logn), as it reduces the number of comparisons required in the trees.

The algorithm given below can be followed to create a balanced BST from the elements in a sorted array:

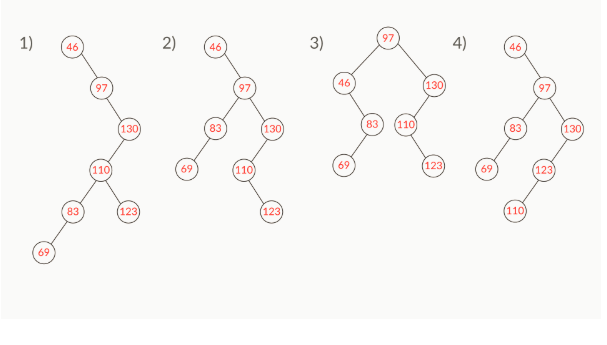
1) Get the middle element of the sorted array and make it the root of the BST.  
2) Recursively repeat the same for the left half and the right half of the array.  
      a) Get the middle of the left half and make it the left child of the root created in Step 1.  
      b) Get the middle of the right half and make it the right child of the root created in Step 1.

Additionally, you can read about self-balancing BSTs that will attempt to balance themselves when new nodes are added to the trees. You can visit the [link](https://www.cpp.edu/~ftang/courses/CS241/notes/self%20balance%20bst.htm) to read more about self-balancing trees.

#### Q71: Inserting Nodes to BST

Identify the binary search tree formed after inserting the numbers 46, 97, 130, 110, 123, 83 and 69 in the given order.

Note that the first element in the given order must be the root node and insert the numbers further based on the rules of BST in the same order.



Ans: 2

**✓ Correct**

**Feedback:**

Starting from the root node, insert all the elements that are less than the root in its left subtree and all the elements that are greater than the root in its right subtree and recursively build the tree. The binary tree given in this option is a BST.

#### Q72: Calculate Height of Balanced BST

Insert the elements 46, 97, 130, 110, 123, 83 and 69 after sorting them and by choosing the median as the root node. What will be the ratio of the height of the tree formed in this case to the height of the tree when the elements are inserted in the given unsorted order?

Hint: Assume the root node to be at height 0.

Ans: 2/4

**✓ Correct**

**Feedback:**

The height of a tree is the number of nodes on the longest path between the root node and a leaf.

This is the sorted order of the given data: 46, 69, 83, 97, 110, 123, 130.  
Take the median of the above list; thus, 97 becomes the root node of the balanced BST. Similarly, take the median of the left subtree and the right subtree and construct the entire balanced BST. This balanced BST has a height of 2 (start counting from 0 and not from 1), and in the previous case, the height of the tree formed was 4. So, the ratio of the height of the tree in this case to the height of the tree in the previous case will be '2/4'. This option is correct.

#### Q73: Optimising Search Efficiency

A balanced binary search tree is constructed with a given data set of 128 elements. What will be the cost of an insertion operation and that of a search operation in the worst case?

Hint: The cost of an operation is essentially its time complexity.

Ans: 7, 7

**✓ Correct**

**Feedback:**

In balanced binary search trees, even in the worst case, the costs of fundamental operations remain logarithmic. ( log(128)(insertion), log(128)(search) = 7, 7).

#### Q74: Comparing Search Efficiency in Balanced BST and BST

Suppose a binary search tree contains ​264 elements. What is the ratio of the worst-case cost of a search operation in a balanced tree to that of an unbalanced tree?

Recall the efficiency of the search operation in balanced and unbalanced trees that is discussed in the segment given above.

Ans:   
2−58

**✓ Correct**

**Feedback:**

In a balanced binary search tree, the time complexity of the search operation is O(logn), whereas in an unbalanced binary search tree, it is O(n).

=(log(264)/(264)

=26/264

=2(−58)

This also means that a balanced binary search tree performs search operations much faster than an unbalanced binary search tree.

#### Q75: Binary search trees

Suppose there are 15 consecutive numbers inserted into a binary search tree in any order. What could be the minimum possible height of the binary search tree (recall the definition discussed in the session on 'Binary Tree')?  
Note: Consider that the height starts from 0 and not from 1.

Ans: 3

**✓ Correct**

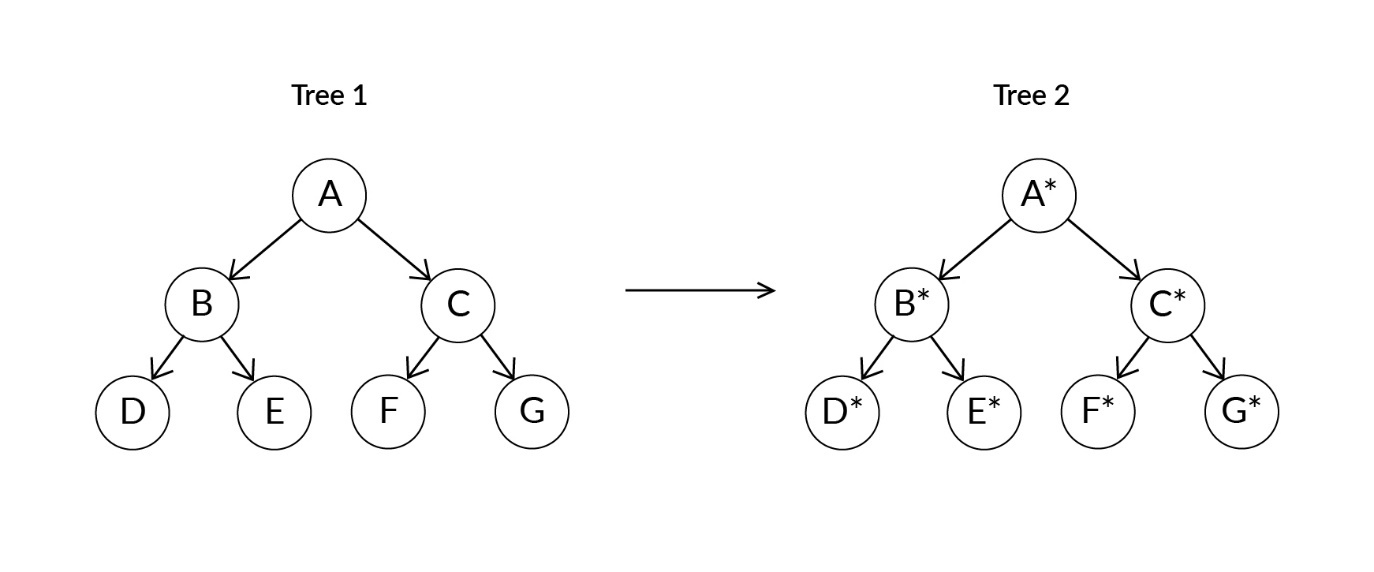
**Feedback:**

The minimum possible height of the binary search tree of n nodes is the floor of log2n (floor is a function that takes a real number as an input and outputs the integer that is less than or equal to the input. Example: floor(3.98) is 3 and floor(3.11) is 3), which is the height of the balanced binary search tree

#### Q76: Binary search trees

A binary tree has been given below. Write a pseudocode to modify the tree consisting of the sum of each node and its children.

(Note: All are its children and not immediate children.)



For example, convert tree 1 to tree 2, such that   
D\* = D  
E\* = E  
F\* = F  
G\* = G  
B\* = D\* + E\* + B  
C\* = F\* + G\* + C  
A\* = B\* + C\* + A

Ans: //node: initially it is the root of the tree

int sumBinaryTree(Node node)

1. If the node is null, then return 0.
2. Recursively call for left and right subtrees and store the sum as the new value of this node
   1. This can be done by the following

node.value += sumBinaryTree(node.leftchild) + sumBinaryTree(node.rightchild);

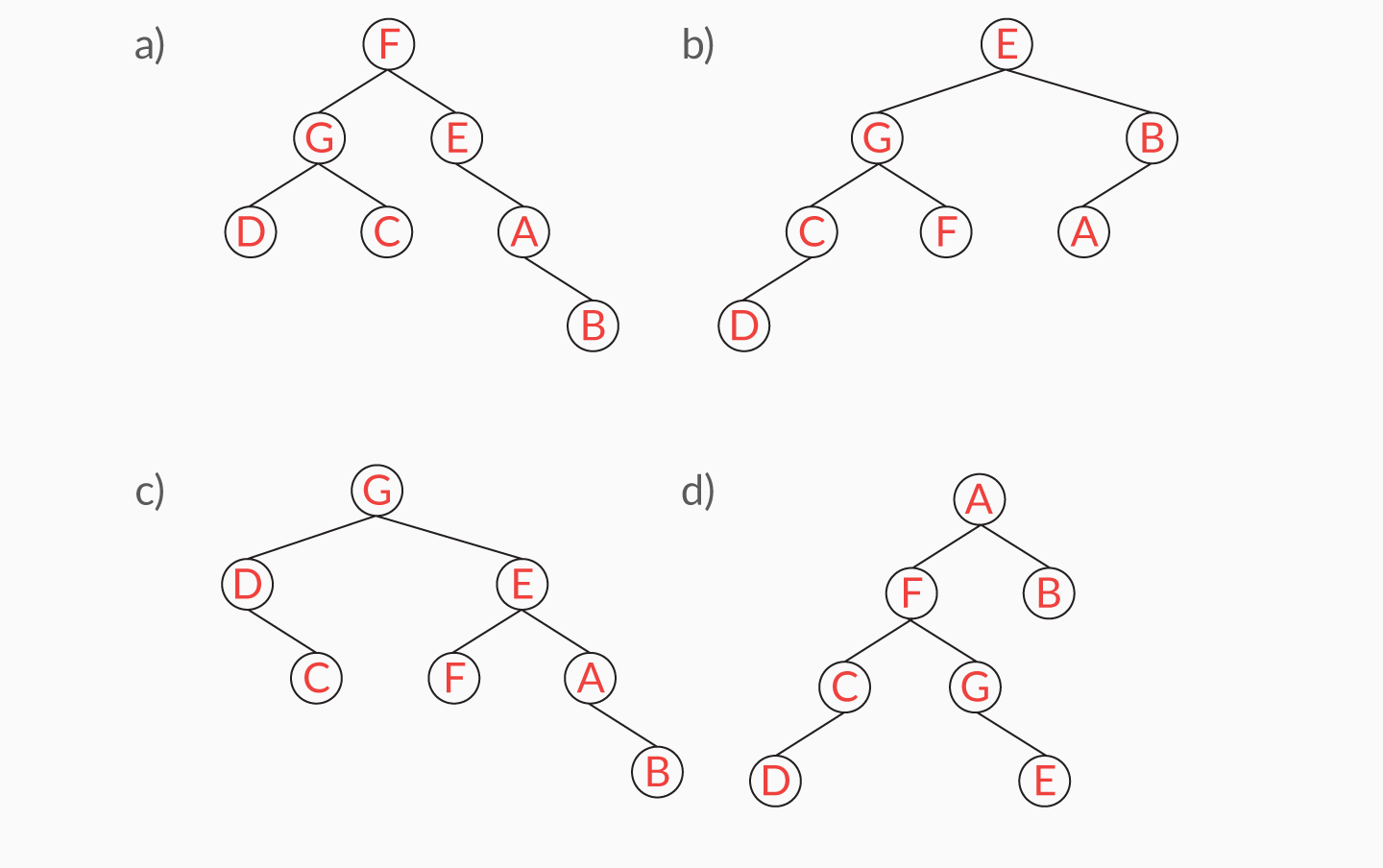
1. Now, the modified node value is the sum of the values of nodes in left and right subtrees and old value of this node. Now, return this modified node value.
   1. This can be done by the following:

return node.value;

#### Q77: Tree Traversals

Suppose the in-order traversal of the elements of a tree is DCGFEAB. Which of the following options corresponds to the binary search tree?

Recall that in-order traversal of a binary search tree gives a sorted array.



Ans: b

**✓ Correct**

**Feedback:**

In-order traversal of the binary search tree produces the elements in the ascending order. In a binary search tree, all the elements in the left subtree are less than the root node, and all the elements in the right subtree are greater than or equal to the root node.



c

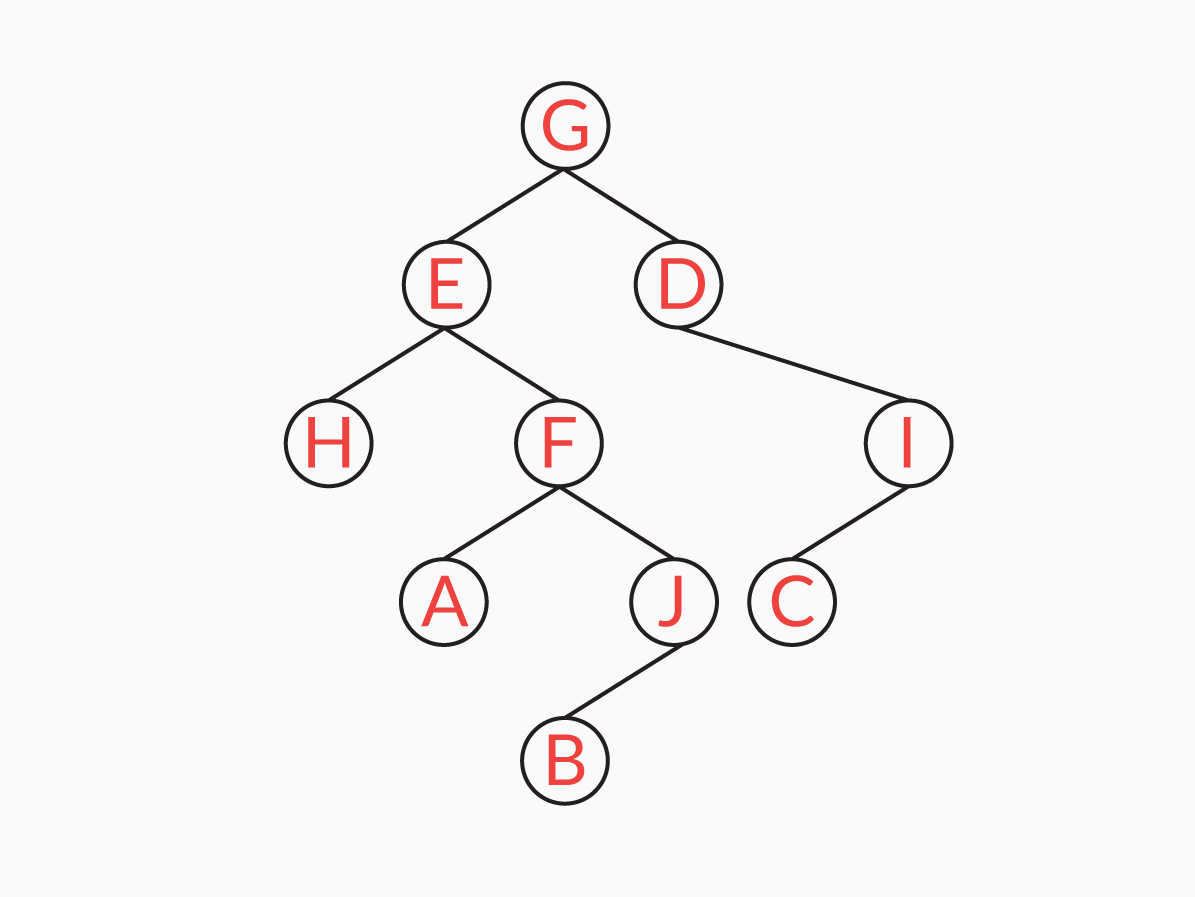
**✓ Correct**

**Feedback:**

In-order traversal of the binary search tree produces the elements in the ascending order. In a binary search tree, all the elements in the left subtree are less than the root node, and all the elements in the right subtree are greater than or equal to the root node.

#### Q78: ree Traversals

Consider the binary search tree given below and answer the following questions:



What is the pre-order traversal of the given binary search tree?

Ans: G, E, H, F, A, J, B, D, I, C

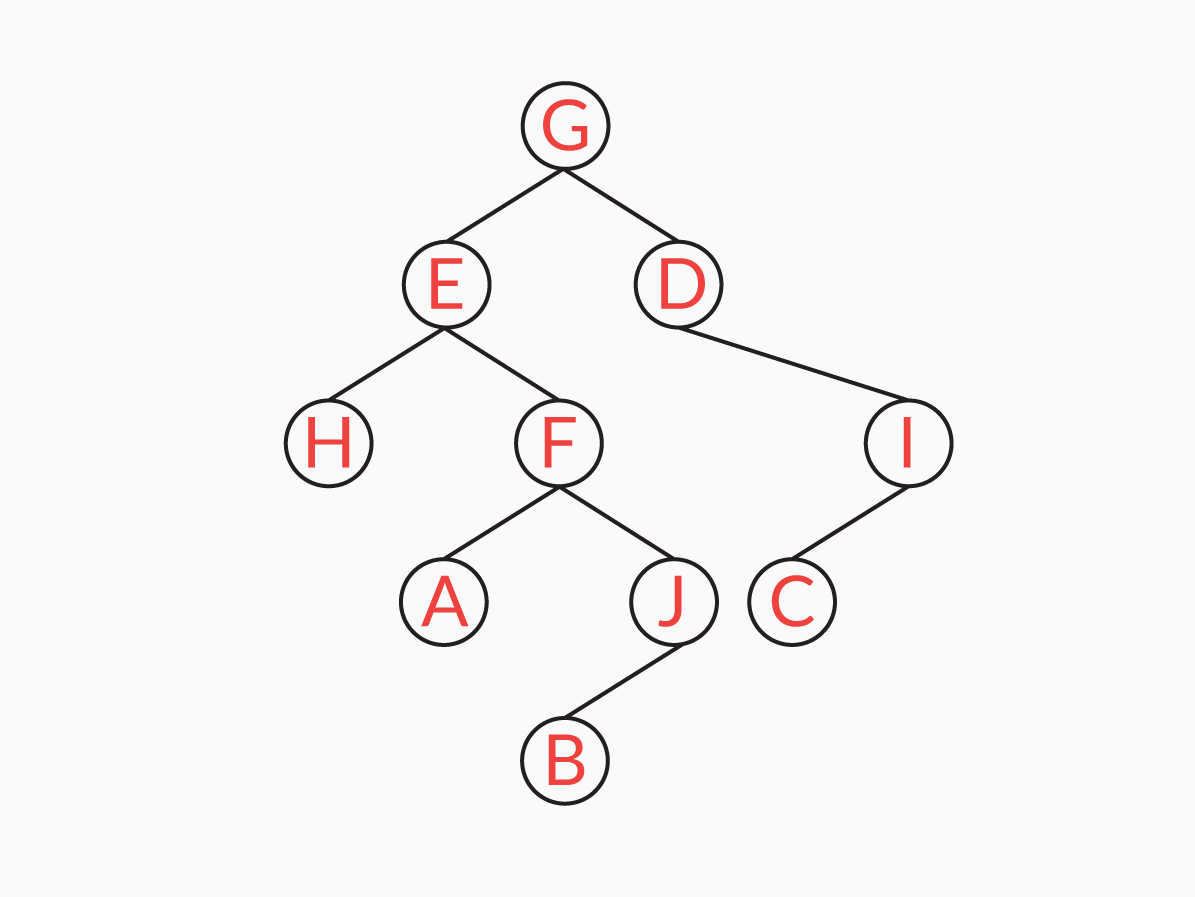
**✓ Correct**

**Feedback:**

The algorithm for a pre-order traversal is as follows: 1) visit the root, 2) traverse the left subtree and 3) traverse the right subtree.

#### Q79: Tree Traversals

Consider the binary search tree given below and answer the following questions.



What will the postorder traversal of the given binary search tree be?

Ans: H, A, B, J, F, E, C, I, D, G

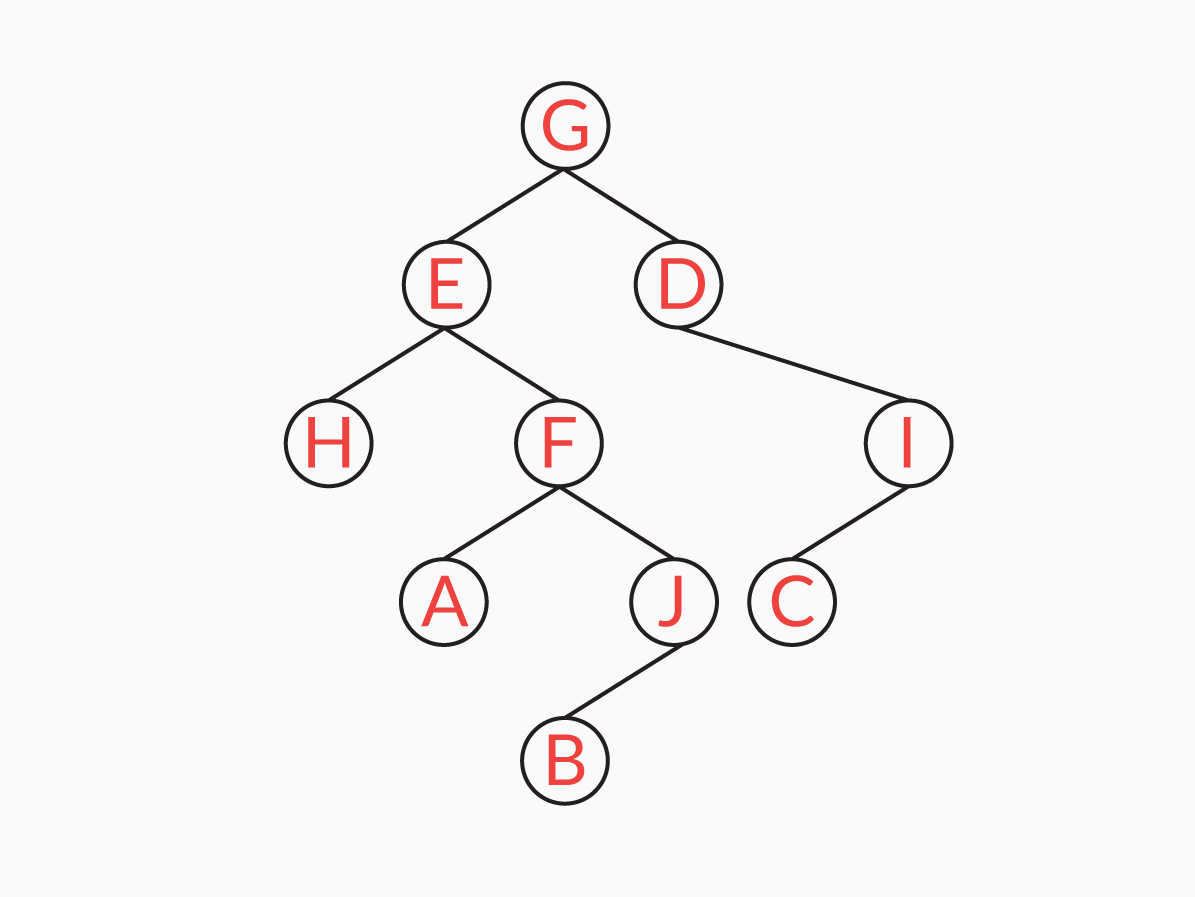
**✓ Correct**

**Feedback:**

The algorithm for a postorder traversal is as follows: 1) traverse the left subtree, 2) traverse the right subtree and 3) visit the root.

#### Q80: Tree Traversals

Consider the binary search tree given below and answer the following questions.



What will the in-order traversal of the given binary search tree be?

Ans: H, E, A, F, B, J, G, D, C, I

**✓ Correct**

**Feedback:**

The algorithm for an in-order traversal is as follows: 1) traverse the left subtree, 2) visit the root and 3) traverse the right subtree.

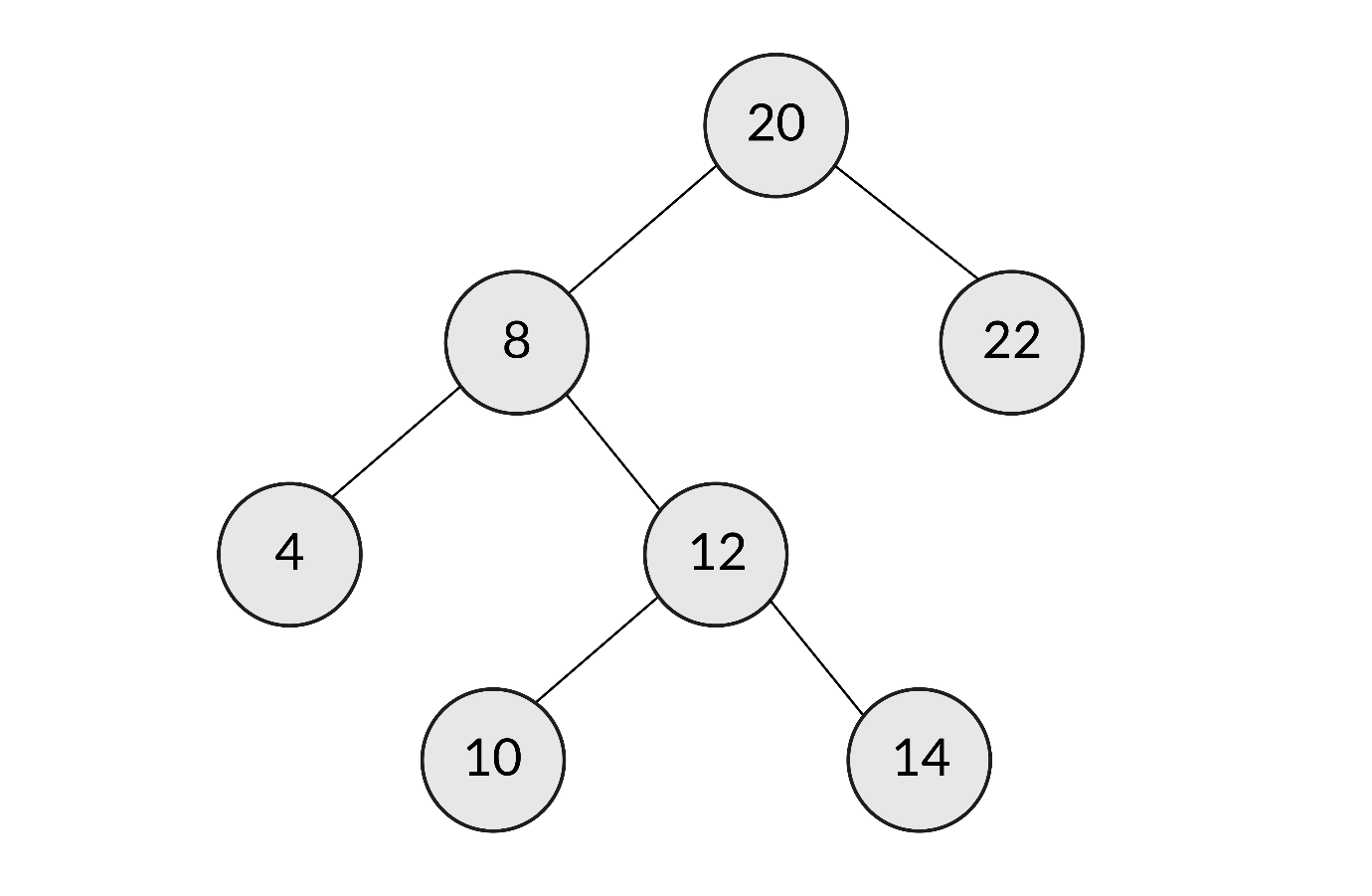
Practice Coding Question

### Q81: Count leaf nodes

**Description**

A node is a leaf node if its left and right child nodes are NULL.

Consider the example of the following binary search tree:

image.png 691.39 KB

Here, the leaf nodes would be 4, 10, 22 and 14 because they are the only ones that do not have any children (neither left child nor right child). Thus, when the input is the tree given above, the output is 4.

**Input:**The input will be in the following format:

1. The first line will be ‘n’, which represents the number of elements to be inserted into the BST.
2. The next line will be n elements separated by spaces, which represents the elements to be inserted into the BST.

**Output:**The output should be in the following format:

       The value of the number of leaf nodes

**Sample Input:**

7

20 8 4 22 12 10 14

**Sample Output:**

4

**Sample Input:**

6

3 4 2 5 7 1

**Sample Output:**

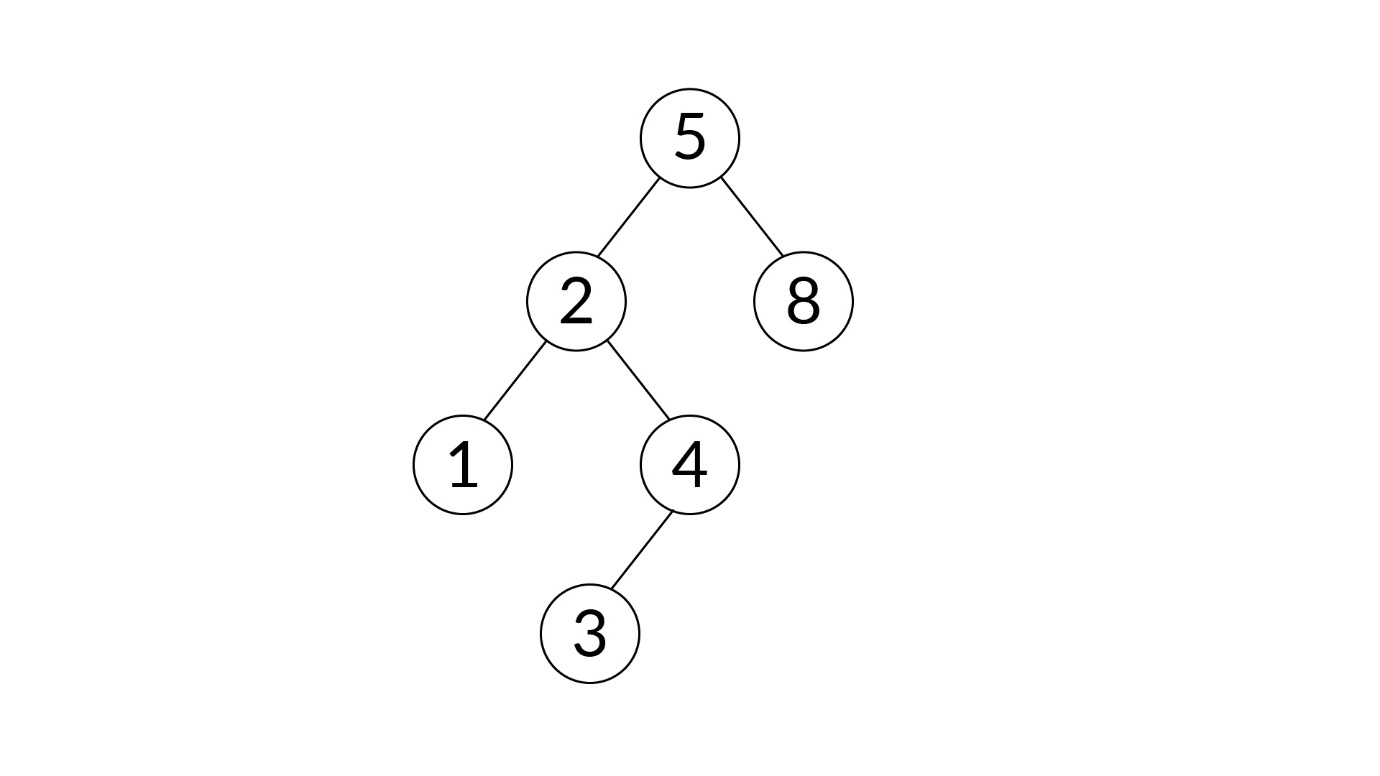
2

Ans: Intellij

### Q82: Nodes that do not have sibling

**Description**

A sibling is a node that has the same parent. In a binary search tree, there can be at the most one sibling. Print all such nodes in a single line that do not have a sibling.﻿﻿﻿

Image 6-01.jpg 104.88 KB

﻿**Input:** The input will be in the following format:

1. The first line will be ‘n’, which represents the number of elements to be inserted into the BST.
2. The next line will be n elements separated by spaces, which represents the elements to be inserted into the BST.

**Output:** The output should be in the following format:

* The values of all the nodes separated by spaces in a single line that do not have a sibling in the BST [first, the nodes of the left subtree (level wise from the top) are printed, followed by the ones in the right subtree in the same format].

**Sample Input:**

6

5 2 8 1 4 3

**Sample Output:**

3

**Sample Input:**

7

22 1 45 27 19 41 3

**Sample Output:**

19 3 27 41

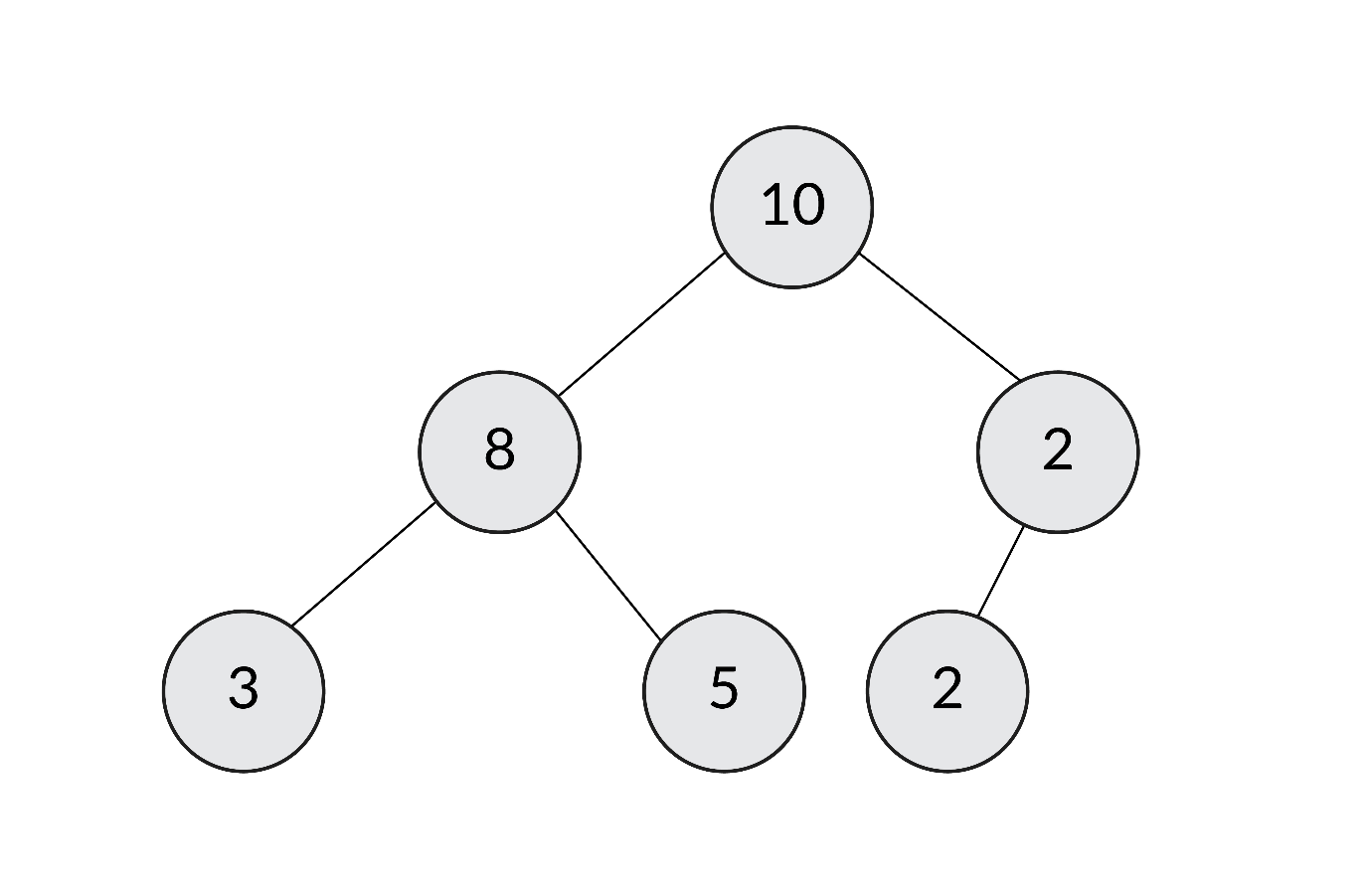
Note that the root node should not be printed because there is no parent of the root node to determine whether it is a sibling or not.

Ans: Intellij

### Q83: Root-to-leaf paths

**Description**

Suppose a binary tree such as this one has been provided.﻿

image.png 651.3 KB

Since there are leaf nodes in this case, all the paths from the root node to the leaf nodes would be as follows:

10 8 3

10 8 5

10 2 2

**Input:** The input will be in the following format:

1. The first line will be ‘n’, which represents the number of elements to be inserted into the binary tree.
2. The next line will be n elements separated by spaces, which represent the elements to be inserted into the binary tree. The code for inserting the values in the tree has been addressed in the code.

**Output:** The output should be in the following format:

You need to print the path from the root node of the tree until the leaf nodes. The path is printed on separate lines with every node separated by spaces. Note that the path is from the left to the right order leaf nodes; so, the path for the leftmost node gets printed first.

**Sample Input:**

6

10 8 2 3 5 2

**Sample Output:**

10 8 3

10 8 5

10 2 2

**Sample Input:**

4

1 7 3 9

**Sample Output:**

1 7 9

1 3

Ans: Intellij

#### Q84: Binary search trees

For a given sorted array, write the pseudocode of a function to create a balanced binary search tree of those array elements and return the root node of the newly created binary search tree.

Ans: //sorted\_arr[]: a sorted array of elements to be inserted into the binary search tree

//start: initially it is starting index of the sorted array

//end: initially it is the last index of the sorted array

Node sortedArrayToBST(sorted\_arr[], int start, int end)

1. If start is greater than end, then return null
2. Get the middle element and make it root
   1. int mid = (start + end) / 2;
   2. Node node = new Node(arr[mid]);
3. Recursively construct the left subtree and make it a left child of the root
   1. node.leftchild = sortedArrayToBST(arr, start, mid - 1);
4. Recursively construct the right subtree and make it a right child of the root
   1. node.rightchild = sortedArrayToBST(arr, mid + 1, end);
5. At the end of recursion return the root of the node:
   1. return node;

#### Q86: Binary search trees

Suppose two arrays named "values" and "levels" are given. The array "values" contains the values of each node, and the array 'levels' contains the level of each node. Write the approach to be followed to construct a binary search tree of the given data.

1. Ans: Sort the array “values”; in the level order
   1. If there are multiple elements in the same level, then sort these in the increasing order of the value of the element
   2. You can do this using comparator in Java.
      1. For more details, visit this link: [How to sort by two fields in Java](https://stackoverflow.com/questions/4805606/how-to-sort-by-two-fields-in-java)
2. Insert each element of the array “values” that is sorted above into the binary search tree in the same order.
   1. For more details, you can refer to the segment 'Addiing a node to a BST' in session 2 of this module.

# Summary

In this session, you learnt about a special variant of binary trees, i.e., binary search trees.

You also learnt that in a binary search tree, all the nodes to the left side of the root node should be less than the root node, and all the nodes to the right side of the root node should be greater than the root node. This condition is mandatory for each node of a binary search tree. Further, you learnt about the algorithms to add, insert and delete a node from a BST. You learnt that all these operations in a BST have an O(log N) time complexity.

We discussed that for a BST to yield O(log N) time complexity, the insertion of elements should not be in the increasing or decreasing order of values. In that case, the BST created is essentially a linked list and yields O(N) time complexity for most of the operations.

#### Q87: Merge K sorted linked list

Can you write a simple approach to merge k sorted linked lists into a single linked list that is also sorted?

Ans: The simple approach would be to take all the elements from the second linked list to kth linked list and continue inserting each element into the first linked list in the sorted order.

1. For i=2 and i <= k
   * Traverse through all the elements in the ith linked list
   * Take each element from the ith linked list
   * Insert the element into the first linked list so that the first linked list maintains the sorted order

This approach involves adding an element to the sorted linked list in the sorted order, which is similar to insertion sort. The worst-case time complexity of the insertion sort of n elements is O(n2). Therefore, this approach needs O((nk)2)  time, where n is the number of elements in the linked list.

#### Q88: Merge K sorted linked list

What is the worst-case time and space complexity of approach 2 that is followed to merge a K sorted linked list?

Note:

1. ‘K’ is the number of the linked list.
2. All the linked lists are of the same size ‘n’.

Ans: O(nk log k) and O(k)

**✓ Correct**

**Feedback:**

Adding and removing an element from the heap takes O(log k) time. So, we will do this for all the ‘nk’ elements so that it will be O(nk log k). Heap takes O(k) space to store all the k elements.

#### Q89: Merge K sorted linked list

What is the worst-case time and space complexity of approach 2, where we used a min-heap to merge K sorted linked list?  
Note:

1. ‘n’ is the number of nodes in the linked list
2. ‘K’ is the number of linked list

Ans: O(nk log k) and O(k)

**✓ Correct**

**Feedback:**

Adding and removing an element from the heap takes O(log k) time. So, we are doing this for all the ‘nk’ elements so that it will be O(nk log k). Heap takes O(k) space to store all the k elements.

Graded Question

#### Q1: Choosing a Hash Function

Suppose you have 500 keys to be stored in a hash table, and the hash table can store up to 1,000 keys. What would a good hash function be in this scenario?

Hint: Consider the Mod X hash function among the options that will yield most of the keys that can be stored and utilised in the hash table.

Ans: Mod 1000

**✓ Correct**

**Feedback:**

Mod 1000 gives indices from 0 to 999, and hence, most of the indices are utilised unless the keys are skewed.

#### Q2: Hash Function

Suppose a hash table of size 13 is given. At which index positions would the keys 28 and 135 be hashed if we use the function H(x) = x mod 13?

Ans: 2, 5

**✓ Correct**

**Feedback:**

Calculating the mod with 13 for both the numbers, 28 and 135, you get 2 and 5, respectively.

#### Q3: HashMap in Java

What is the output of the following program?

**import** java.util.HashMap;

**import** java.util.Map;

**class** Maps {

**public** **static** **void** main(String args[]) {

Map<String, Integer> hashMap = **new** HashMap<>();

hashMap.put("A", **new** Integer(**1**));

hashMap.put("B", **new** Integer(**2**));

hashMap.put("C", **new** Integer(**3**));

hashMap.put("A", **new** Integer(**2**));

hashMap.put("C", **new** Integer(**2**));

System.out.println(hashMap.entrySet());

}

}

Ans: [A=2, B=2, C=2]

**✓ Correct**

**Feedback:**

The entry set displays the key-value pairs of the HashMap, and as 2 is added as the values to the keys A and B, it overwrites the previous value already.

#### Q4: HashMap

Suppose a HashMap is declared as follows:

HashMap<Integer,String> hash= **new** HashMap<Integer,String>();



Which of these statements will not return an error?

Ans:   
hash.get(5);

**✓ Correct**

**Feedback:**

The HashMap is parameterised with 'int' as the key and 'string' as the value. Here, int is the input of the 'get' function, which is expected; so, there will not be an error in this statement.

#### Q6: Hash Function

Suppose you want a hash function that distributes the keys uniformly to the indices 0–9 in a hash table. What would be the suitable hash function in this case that would help to achieve this distribution over the hash table.

Hint: Think of the hash function that gets you all the digits from 0–9 in the unit's place. Also, recall how you can use the modulus operator for the same.

Ans: h(i)=i3mod10

**✓ Correct**

**Feedback:**

When you cube a digit from 0–9, you get all the possible digits from 0–9 in the unit's place, from which, by calculating the mod, you can get the desired range.

#### Q7: Number of Nodes in the Tree

Imagine that in a BST, node B (value = 64) is the immediate child of node A (value = 32). Also, node B does not have any children. The BST is made of distinct positive integers that are less than 100. Which of the following options is surely **false**?

Hint: Eliminate the range of numbers using the properties of BST.

Ans: Number of nodes in the tree = 74

**✓ Correct**

**Feedback:**

The information given conveys that the tree does not have the nodes 33, 34, 35, …., 63. This is because the nodes from 33 to 63 can only be present in the tree as the left subtree of node 64. However, the question specifies that node B does not have any children. Therefore, you can find the total number of nodes by subtracting 31 from 99, 99 being the total number of nodes that can be used from 1 to less than 100. Moreover, 31 is the number of nodes from 33 to 63, which are not present in the tree.

Q8: Maximum depth of a tree

**✓ Correct**

**Feedback:**

To find the maximum depth of a tree, the program has to recursively traverse through all the subtrees while taking the maximum depths of the left and right subtrees and then adding 1 to them. You can visit this [link](https://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/)to get a better understanding.

#### Q9: Traversal in a BST

Suppose you have a binary search tree that contains the numbers between 1 and 157. You are supposed to search for a node with the value 56. Which of the following sequences below cannot be a sequence of nodes that the program has walked through?

Take the first node in the sequences as the root node.

Ans: 9, 75, 46, 67, 40, 58, 56

**✓ Correct**

**Feedback:**

When you start searching for a value in a BST, you always need to start at the root node. Then, you need to keep traversing depending on whether the search key is less than or greater than the node that you are currently at. In this case, when you start with 9 as the root node, you should realise that 56 is greater than 9. So, in conclusion, you should ignore the left subtree and hop on to the right subtree of the root node. In this option, when you reach 46, all the elements after that should belong to the right subtree of 46 (56 being greater than 46). However, as you can see, the list has an element 40 that comes after 46, and this is not possible in a BST. So, this option satisfies the question.

#### Q10: Search Operation

A binary search tree is created using all the integers from 1 to 64. The tree is created in such a way that the height of the tree is 6. What should be the worst-case time complexity for searching a value in this binary search tree?

Ans: O(log N), where N is the number of nodes in the tree

**✓ Correct**

**Feedback:**

The number of nodes in the tree is 64. For this tree to be perfectly balanced, the height of the tree should be log264 or 6. This matches the height given in the question. So, this being a perfectly balanced BST shall exhibit a time complexity of O(log N) for all search activities.

#### Q11: Depth of a Node

The following numbers are inserted in a BST in the following order: 43, 1, 38, 54, 75, 29, 6, 44. What is the depth of node 29 in this BST? Assume the root node to be at level 0.

Ans: 3

### Q12: Balanced BST

**Description**

You have already written the pseudo-code for the similar problem in the ‘Practice Questions’ segment. It is now time to write the code!

Given a binary tree, construct a balanced binary search tree consisting of the sum of each node and its children (Note: All its children and not immediate children). The output should be the postOrder traversal of the new binary search tree.

**Example:**

**Sample Input:**

6

6 10 20 1 51 43

1 10 6 20 43 51

**Sample Output:**

20 6 51 131 94 36

S**ample Input:**

5

40 60 30 20 50

50 20 30 60 40

**Sample Output:**

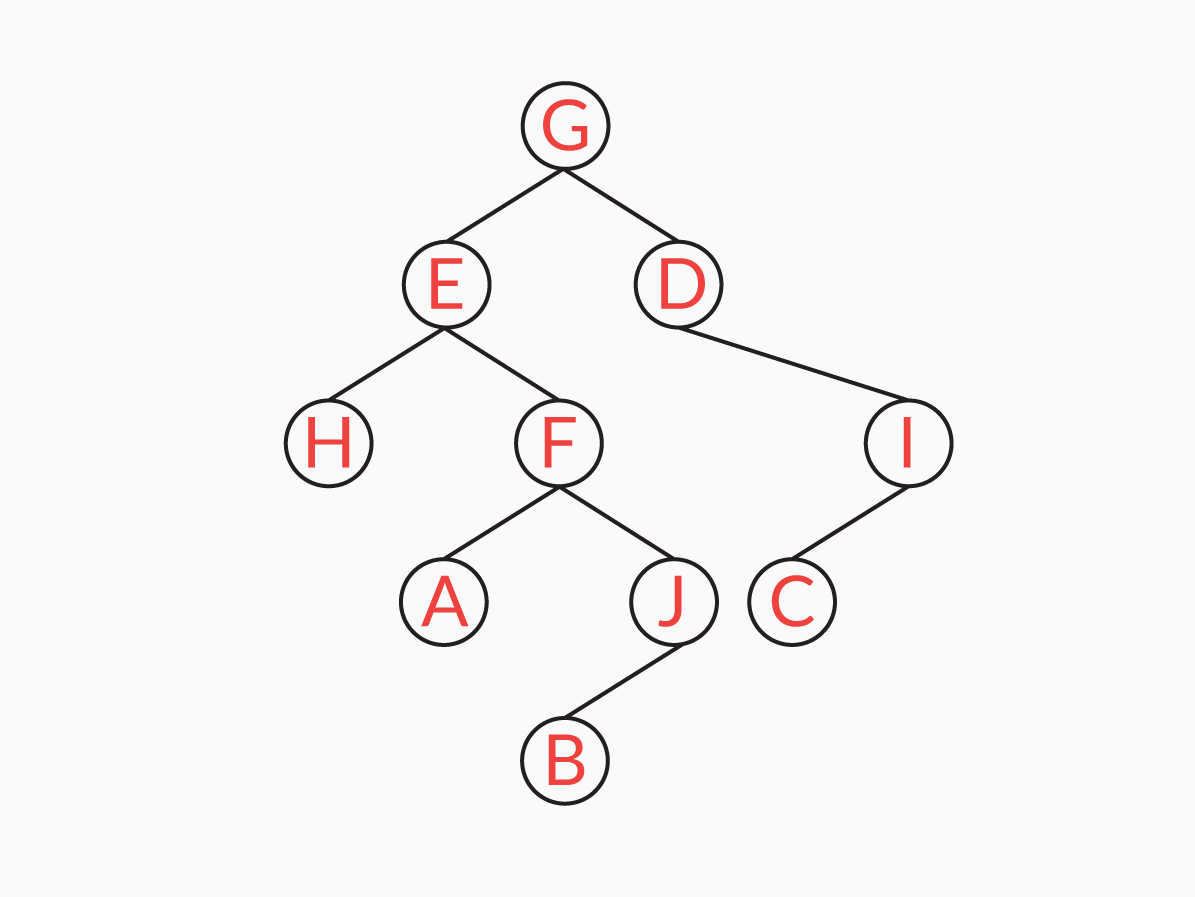
100 40 200 150 130

In the input, the first value 6 is the size of the tree. The values in the second line correspond to the inorder traversal of the given binary tree, and the last line corresponds to the pre-order traversal of the given binary tree.

The output is the **post-order traversal** of the new binary search tree that is supposed to be constructed with node values **separated by spaces**.

Hint: Suppose you have the following binary tree with you.

﻿

7.png 34.9 KB

﻿

You can follow the steps given below.﻿

1. Create a new binary tree with the given condition as explained below.
2. Since B is a leaf node, it will be printed there as it is.
3. Similarly, A, being a leaf node, will be there as it is.
4. J will be replaced by (J+B).
5. F will be replaced by A + NEW VALUE OF J, i.e., J+B. So, F will be replaced by A + J + B.
6. E will be replaced by H + new value of F
7. And so on
8. Once this new binary tree is constructed, you can print any traversal of this new binary tree and sort that in the ascending order.
9. That sorted list can be used to create a balanced BST.

Ans: Intellij