Instance simplification: presorting, Gaussian elimination, balanced search trees

6.1 - 6.3

Transform and Conquer

- Algorithms based on the idea of transformation
  - Transformation stage: Problem instance is modified to be more amenable to solution
  - Conquering stage: Transformed problem is solved

Transform and Conquer

- Algorithms based on the idea of transformation
  - Transformation stage: Problem instance is modified to be more amenable to solution
  - Conquering stage: Transformed problem is solved
- Major variations are for the transform to perform:
  - Instance simplification
  - Different representation
  - Problem reduction

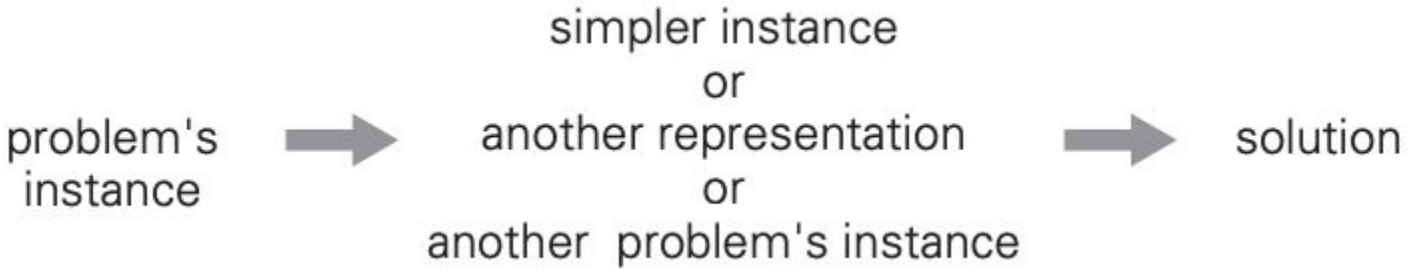
Variations of the transformation

Transformation to a simpler or more convenient instance of the same problem—we call it **instance simplification**.

Transformation to a different representation of the same instance—we call it **representation change**.

Transformation to an instance of a different problem for which an algorithm is already available—we call it **problem reduction**.

Transform and conquer strategy



Presorting

Presorting is an old idea, you sort the data and that allows you to more easily compute some answer

Some simple presorting examples:

1. Element Uniqueness
2. Computing the mode of n numbers

Presorting- Element Uniqueness

Given a list A of n orderable elements, determine if there are any duplicates of any element

Brute Force:	Presorting:
for each x $\in$ A	Sort A
for each y $\in$ {A – x}	for i $\leftarrow$ 1 to n-1
if x = y return not unique	if A[i] = A[i+1] return not unique
return unique	return unique

Presorting- Element Uniqueness

Given a list A of n orderable elements, determine if there are any duplicates of any element

Brute Force:	Presorting:
for each x $\in$ A	Sort A
for each y $\in$ {A – x}	for i $\leftarrow$ 1 to n-1
if x = y return not unique	if A[i] = A[i+1] return not unique
return unique	return unique

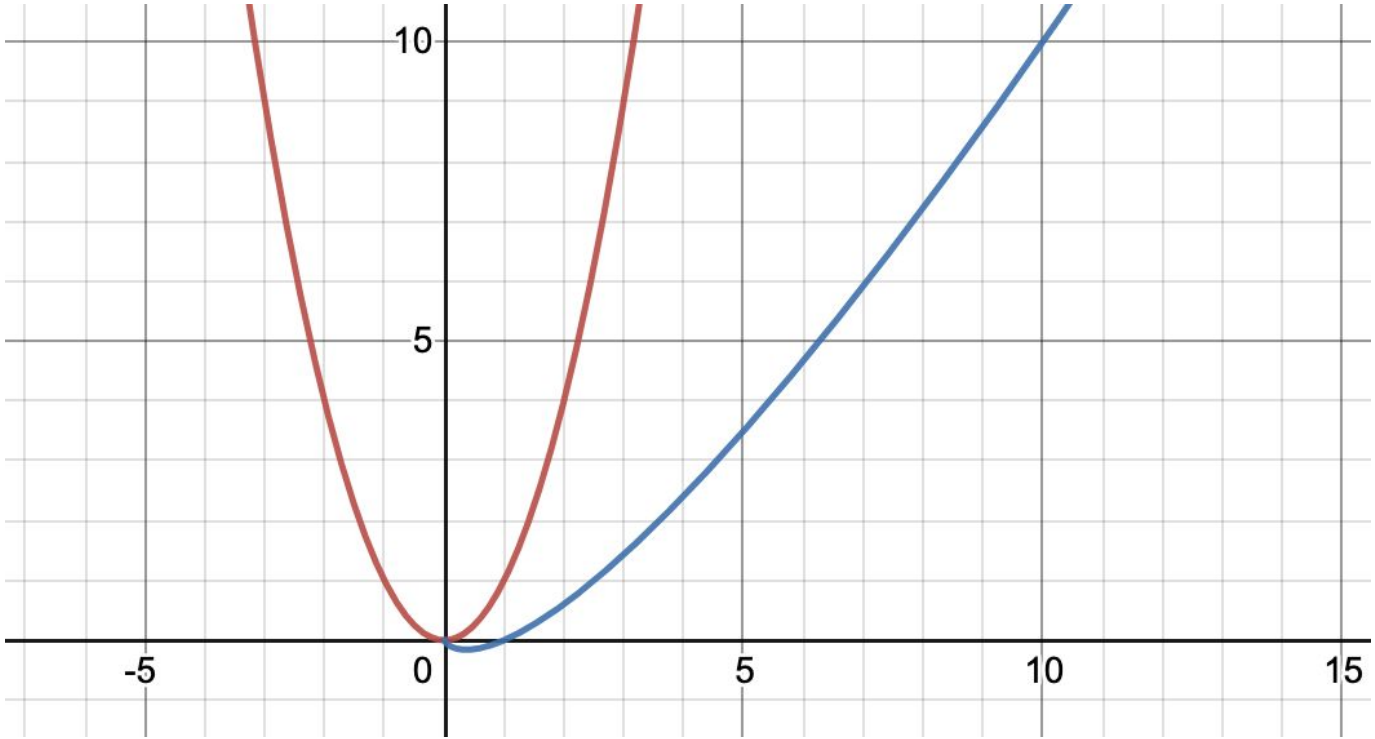
Runtime??

Runtime for presorting- element uniqueness

Brute force worst-case efficiency was in $\Theta(n^2)$

If we use a good sorting algorithm for a presort approach, such as mergesort, with worst-case efficiency in $\Theta(n \log n)$, the worst-case efficiency of the entire presorting-based algorithm will be also in $\Theta(n \log n)$

Runtime for presorting- element uniqueness



Presorting- Computing a mode

A mode is a value that occurs most often in a list of numbers

- e.g. the mode of [5, 1, 5, 7, 6, 5, 7] is 5
- If several different values occur most often any of them can be considered the mode

Presorting- Computing a mode (brute force)

What would the brute force method for finding a mode be?

Presorting- Computing a mode (brute force)

The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency. On each iteration, the ith element of the original list is compared with the values already encountered by traversing this auxiliary list. If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with a frequency of 1.

Presorting- Computing a mode (brute force)

What is the run time for brute force method here?

Presorting- Computing a mode (brute force)

What is the run time for brute force method here?

Brute force worst-case efficiency was in $\Theta(n^2)$

Presorting- Computing a mode (presorting)

What would the presort method for finding a mode be?

Presorting- Computing a mode (presorting)

Let us first sort the input. Then all equal values will be adjacent to each other. To compute the mode, all we need to do is to find the longest run of adjacent equal values in the sorted array.

Presorting- Computing a mode (presorting)

What is the run time for presort method here?

Presorting- Computing a mode (presorting)

What is the run time for presort method here?

The analysis here is similar to the analysis of Element Uniqueness example: the running time of the algorithm will be dominated by the time spent on sorting since the remainder of the algorithm takes linear time.

Consequently, with an n log n sort, this method's worst-case efficiency will be in a better asymptotic class than the worstcase efficiency of the brute-force algorithm.

Homework problems!

Please complete exercises 6.1 #1, 4

Gaussian Elimination

You are certainly familiar with systems of two linear equations in two unknowns:

$$a_{11}x + a_{12}y = b_1$$

$$a_{21}x + a_{22}y = b_2$$

How would we solve this?

Gaussian Elimination

You are certainly familiar with systems of two linear equations in two unknowns:

$$a_{11}x + a_{12}y = b_1$$

$$a_{21}x + a_{22}y = b_2$$

Use either equation to express one of the variables as a function of the other and then substitute the result into the other equation, yielding a linear equation whose solution is then used to find the value of the second variable.

Gaussian Elimination

In many applications, we need to solve a system of n equations in n unknowns:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$

$$\ldots$$

$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$$

Gaussian Elimination

In many applications, we need to solve a system of n equations in n unknowns.

The idea of Gaussian elimination is to transform a system of n linear equations in n unknowns to an equivalent system with an upper-triangular coefficient matrix, a matrix with all zeros below its main diagonal

Gaussian Elimination Example

Solve the following system:

$2x_1 - x_2 + x_3 = 1$

$4x_1 + x_2 - x_3 = 5$

$x_1 + x_2 + x_3 = 0$

Gaussian Elimination Example

Add coefficients to matrix

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Gaussian Elimination Example

What operations to get

0s below main diagonal?

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Gaussian Elimination Example

subtract 2*row1

subtract ½*row1

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Gaussian Elimination Example

subtract ½*row2

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

Gaussian Elimination Example

Success

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Gaussian Elimination Example

In our example we replaced an equation with a sum or difference with a multiple of another equation

We might also need to:

- Exchange two equations
- Replace an equation with its nonzero multiple

Note for Gaussian Elimination

Gaussian Elimination does not work on singular matrices (they lead to division by zero)

Homework problems!

Please complete exercises 6.2 #1, 3

Balanced Search Trees

Your book has a lot of info and includes many diagrams- read the text and look over the different diagram examples.

Search trees are a method for storing data in a way that supports fast insert, lookup, and delete operations.

Instead of linear data (like arrays), trees organize data in a hierarchical data structure

Balanced Search Trees

Your book has a lot of info and includes many diagrams- read the text and look over the different diagram examples.

Search trees are a method for storing data in a way that supports fast insert, lookup, and delete operations.

Instead of linear data (like arrays), trees organize data in a hierarchical data structure

Think of a file structure on your computer

Balanced Search Trees

The key issue with search trees is that you want them to be balanced so that lookups can be performed quickly, and yet you don’t want to require them to be perfect because that would be too expensive to maintain when a new element is inserted or deleted.

Balanced Search Trees

In this lecture, we will discuss B-trees, which is a method that handles this tradeoff so that all desired operations can be performed in time

O(log n).

B-Tree in short

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory.

B-Tree in short

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory.

To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory.

B-Tree in short

When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.

B-Tree in short

Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

B-Tree in short

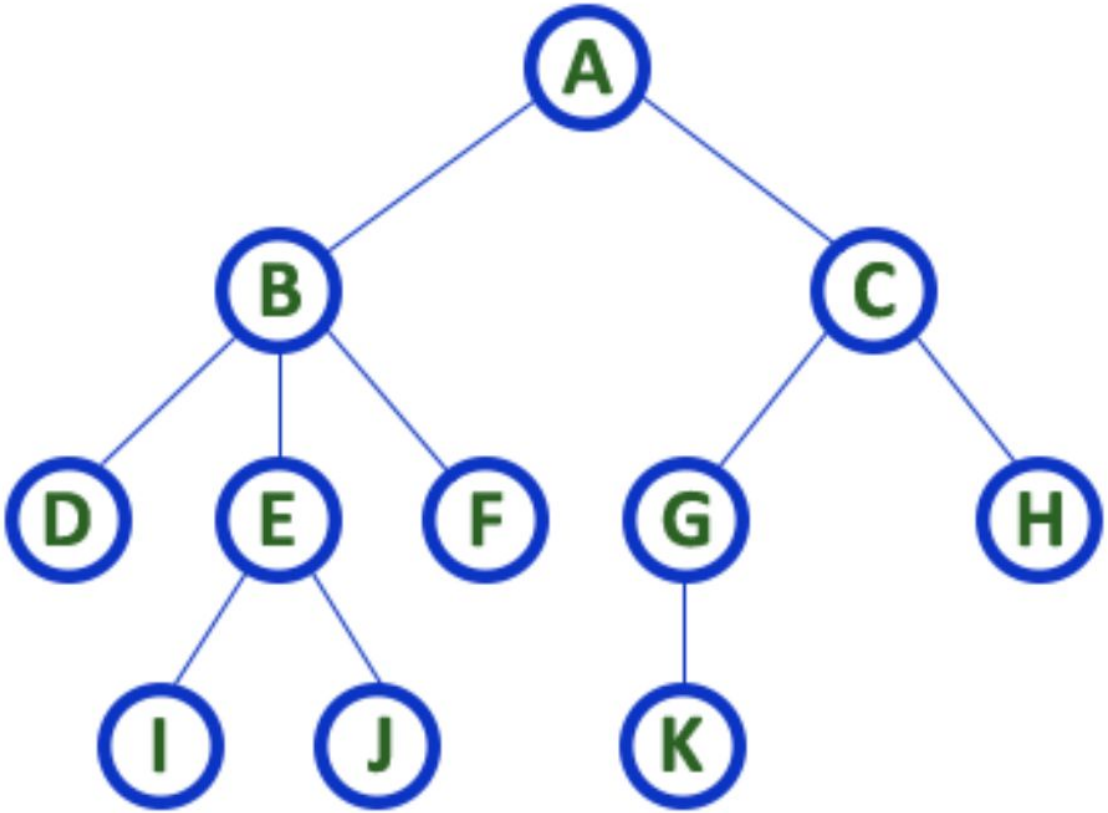
B-trees: a form of balanced search tree that uses flexibility in its node degrees to efficiently keep the tree balanced.

- Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data.
- It is most commonly used in database and file systems.

Tree Terminology

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links/edges.

Each individual element is called a node

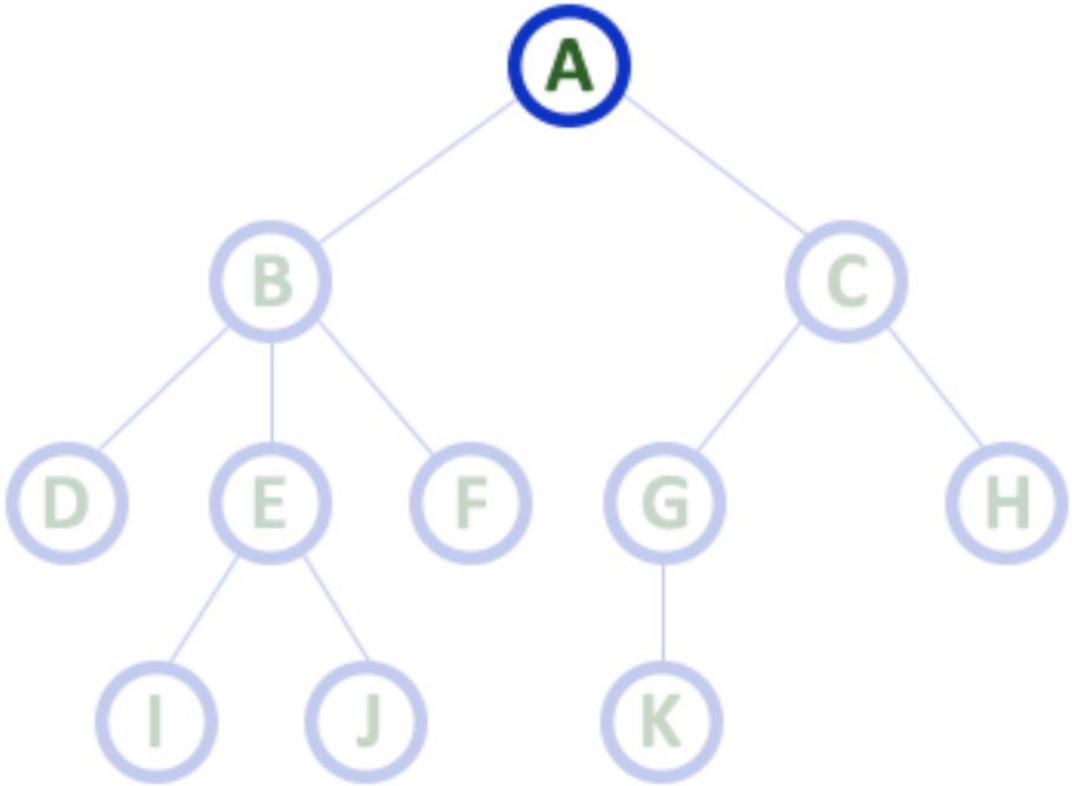


Tree Terminology

Here, A is the root node

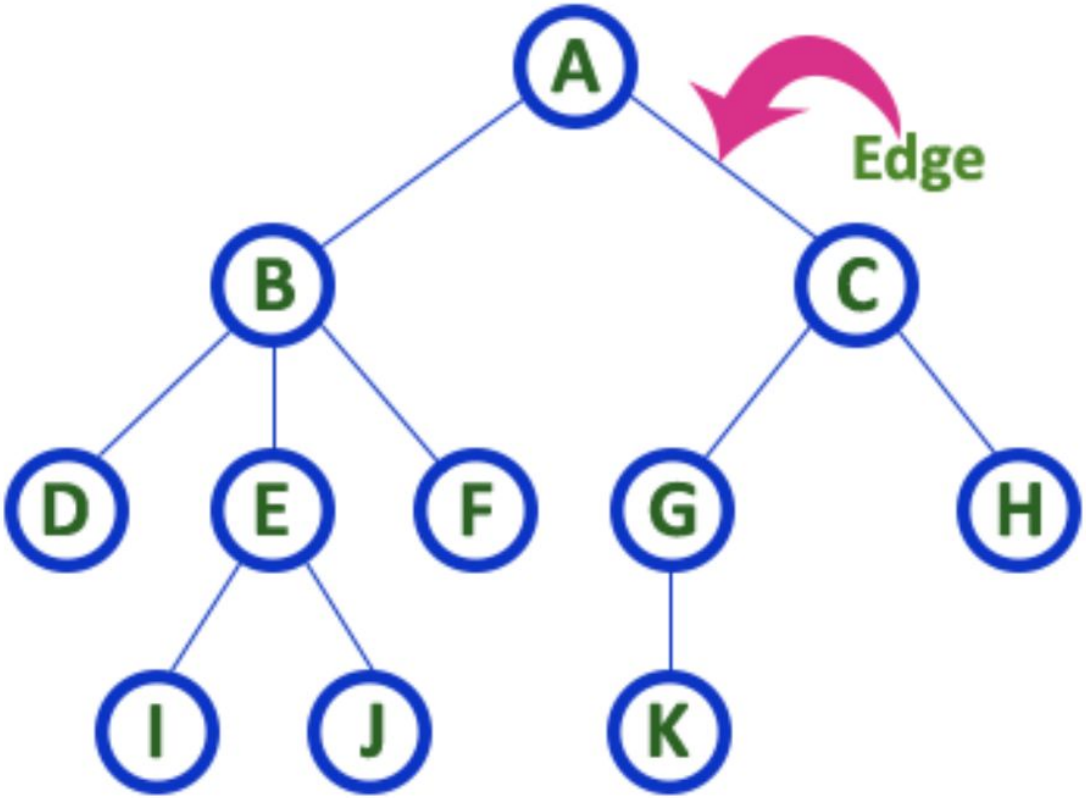
In any tree, the first

node is called the root node



Tree Terminology

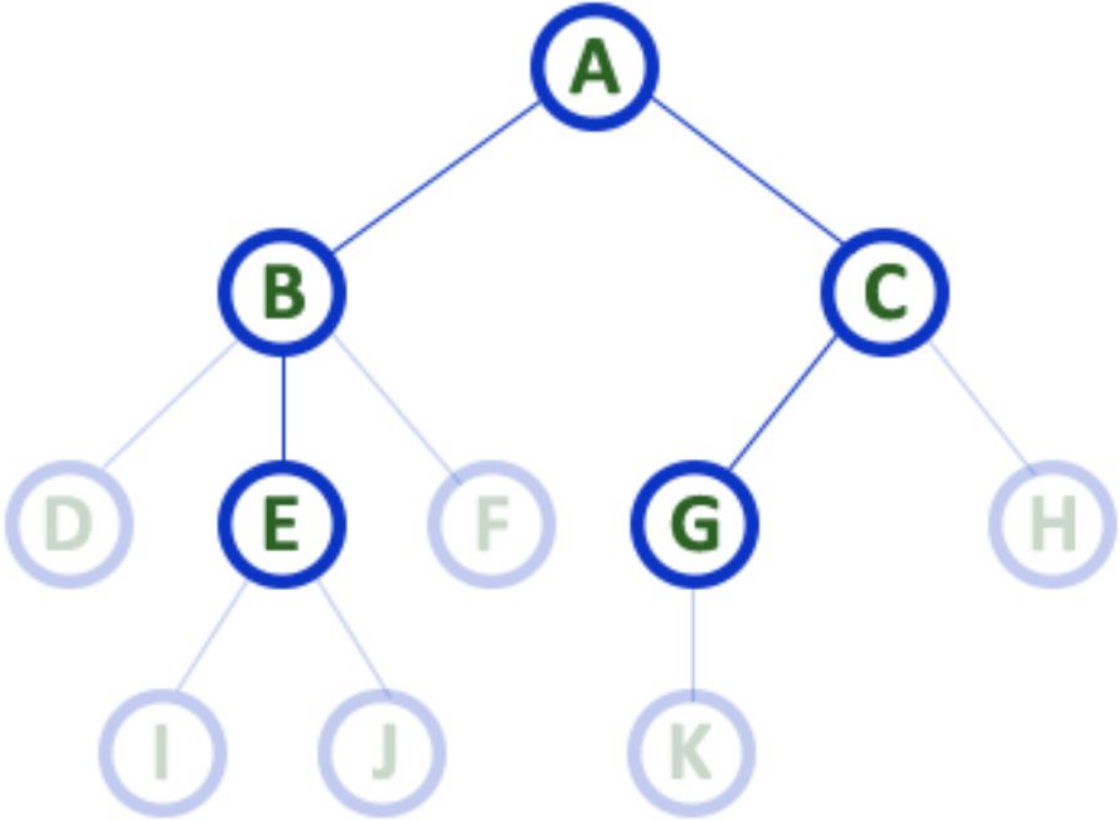
In any tree, an edge is a connecting link between two nodes



Tree Terminology

In any tree, the node which has a child or children is called a parent (predecessor)

A, B, C, E, G

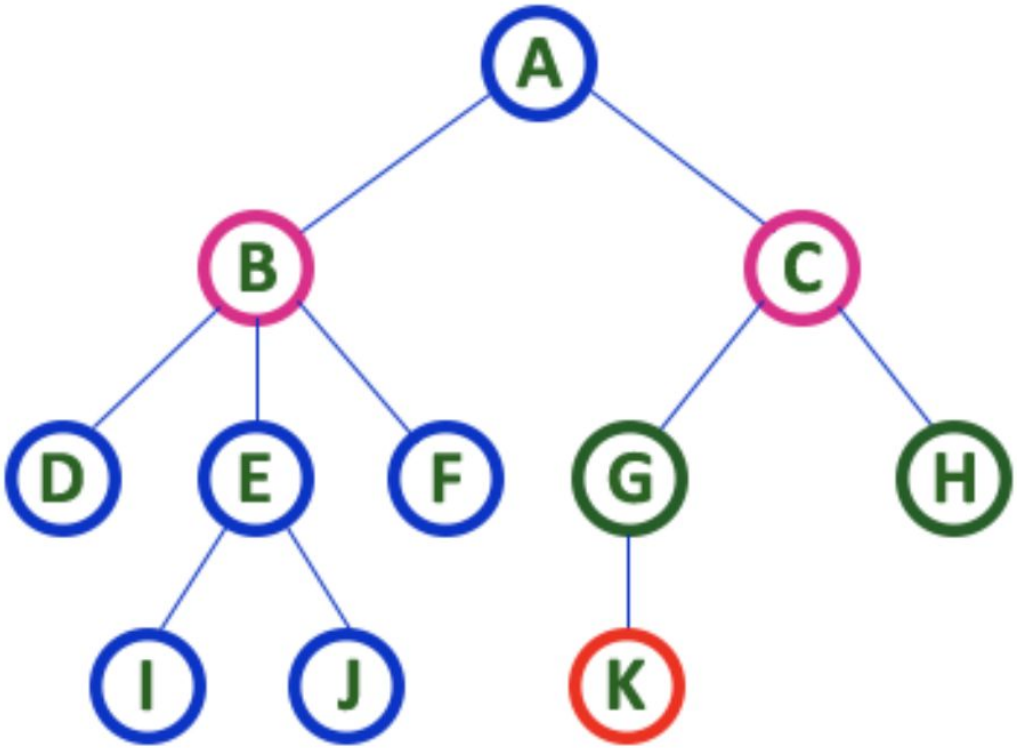


Tree Terminology

In any tree, the node which

is descendant of any node

is called as child node



Tree Terminology

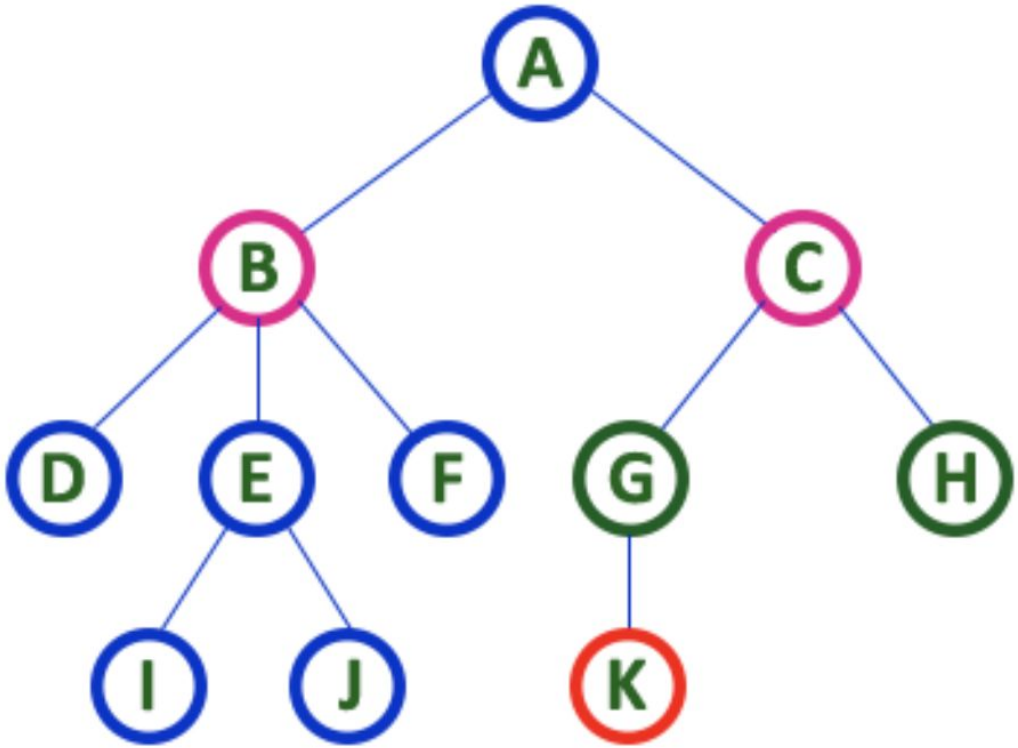
Here:

B + C are children of A

G + H are children of C

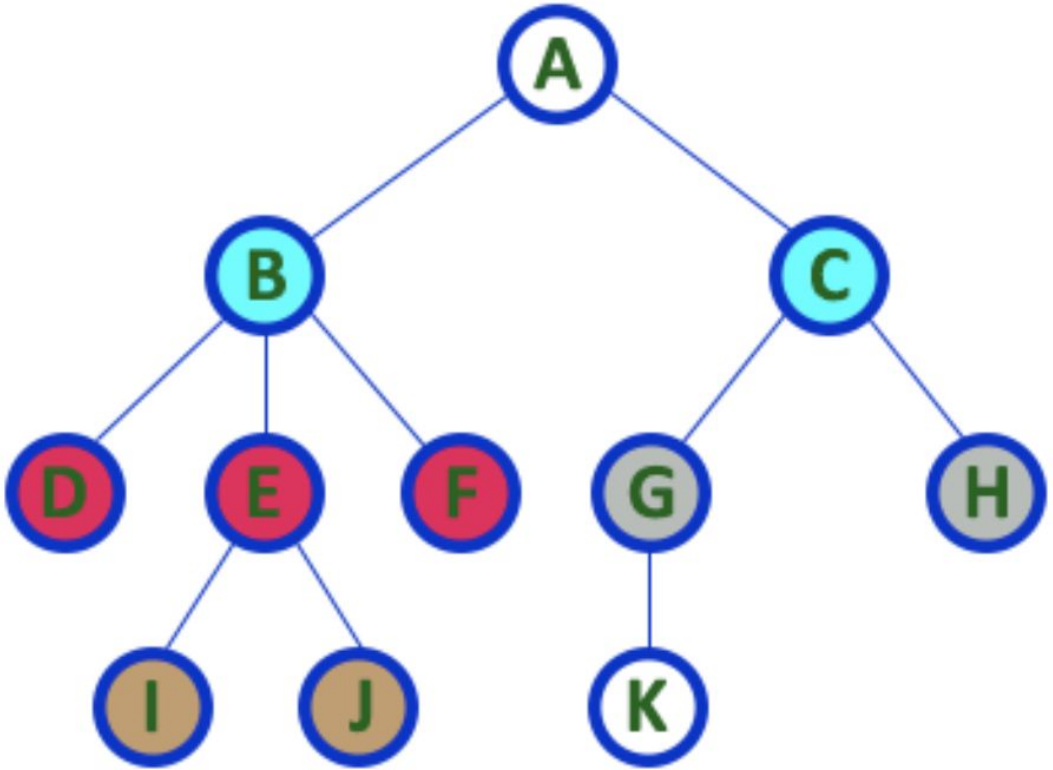
K is a child of G

etc



Tree Terminology

In any tree, nodes which belong to same parent are called as siblings.



Tree Terminology

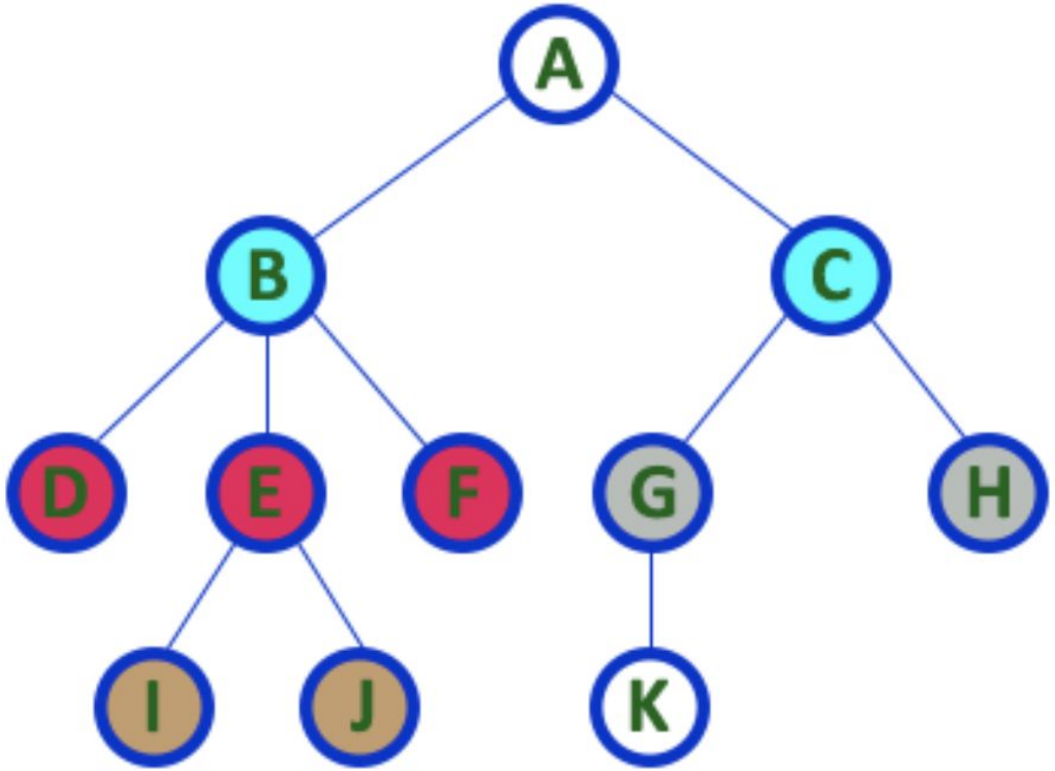
Here:

B + C are siblings

D + E + F are siblings

G + H are siblings

I + J are siblings

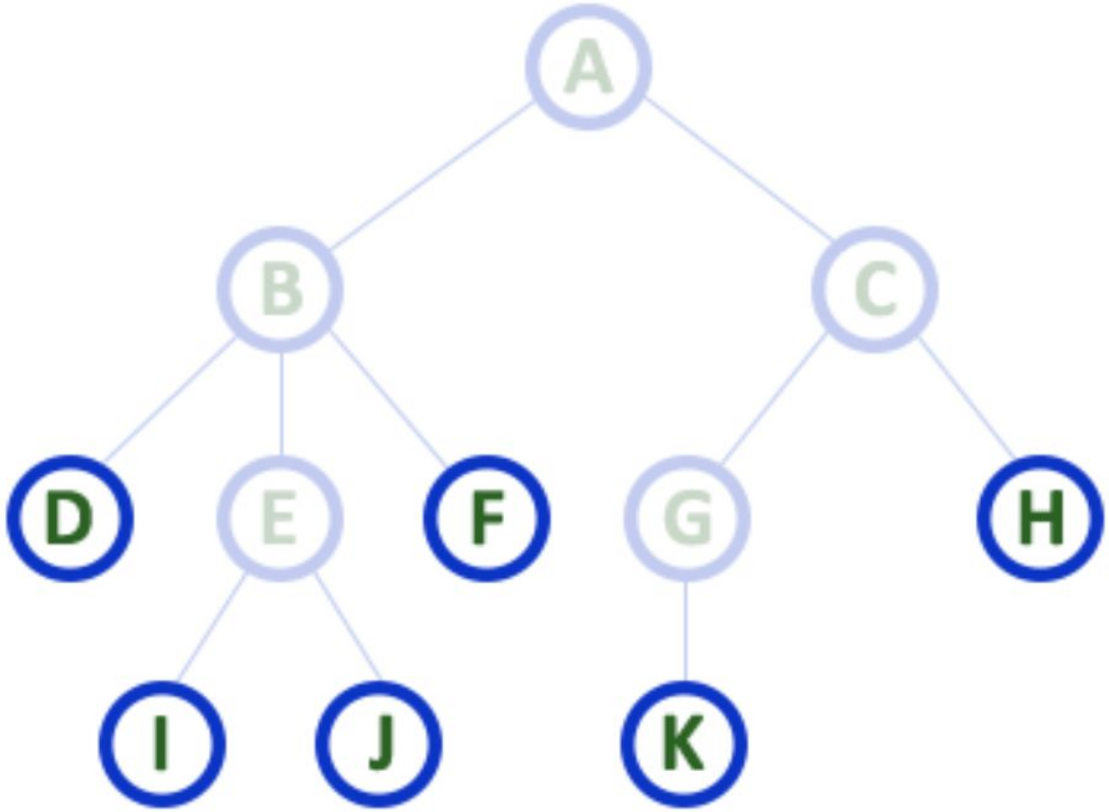


Tree Terminology

In any tree, the node which

does not have a child

is called as leaf node.

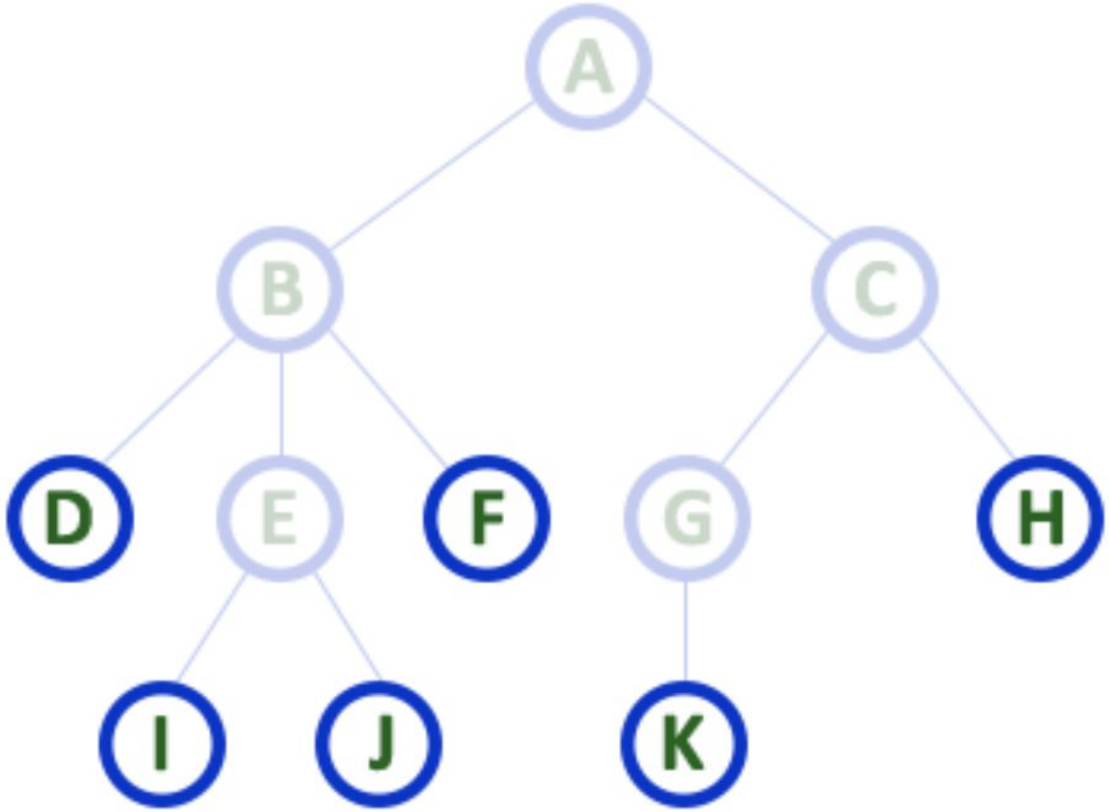


Tree Terminology

Here:

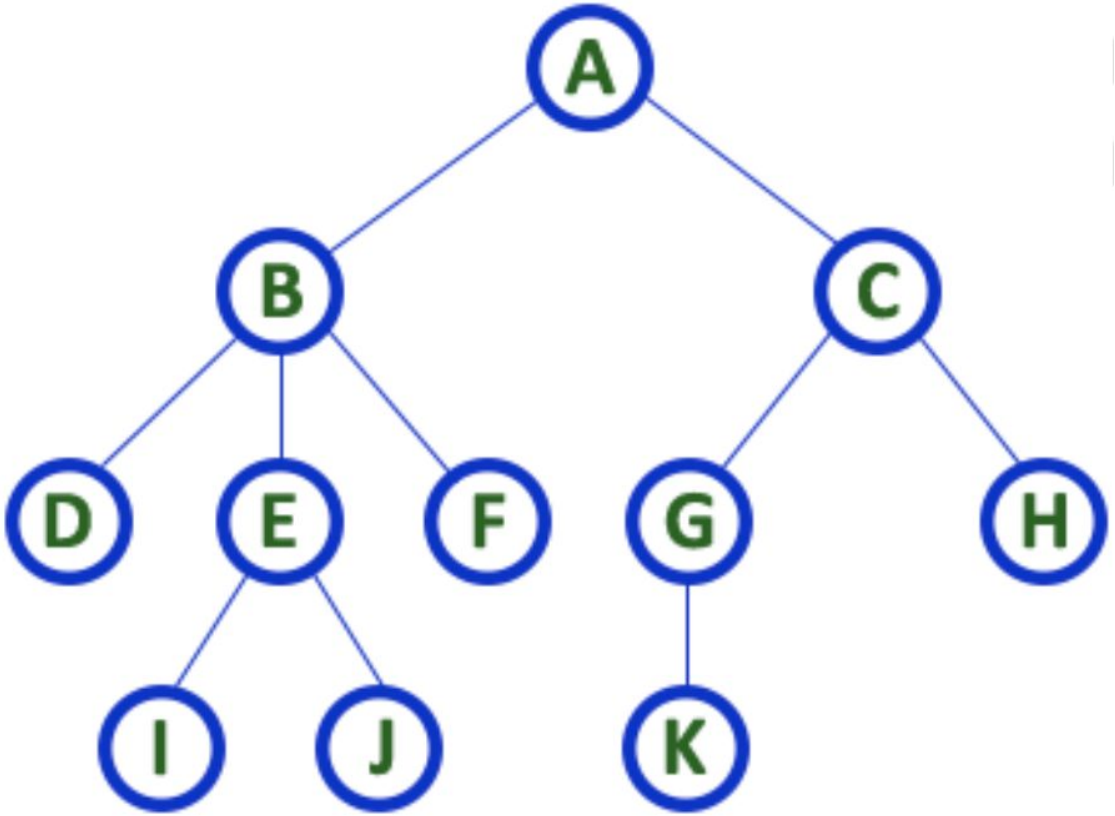
D, I, J, F, K, and H are

leaf nodes



Tree Terminology

In any tree, the total number of children of a node is called as degree of that node.



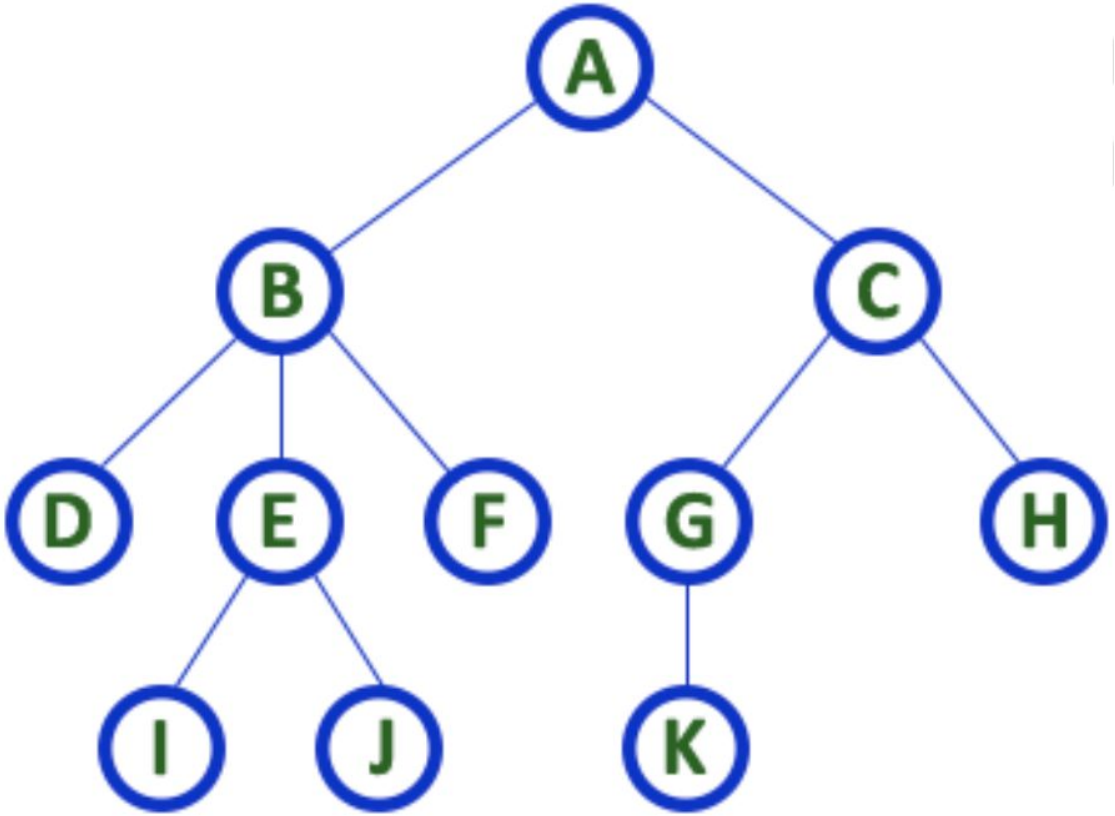
Tree Terminology

Here:

The degree of B = 3

The degree of A = 2

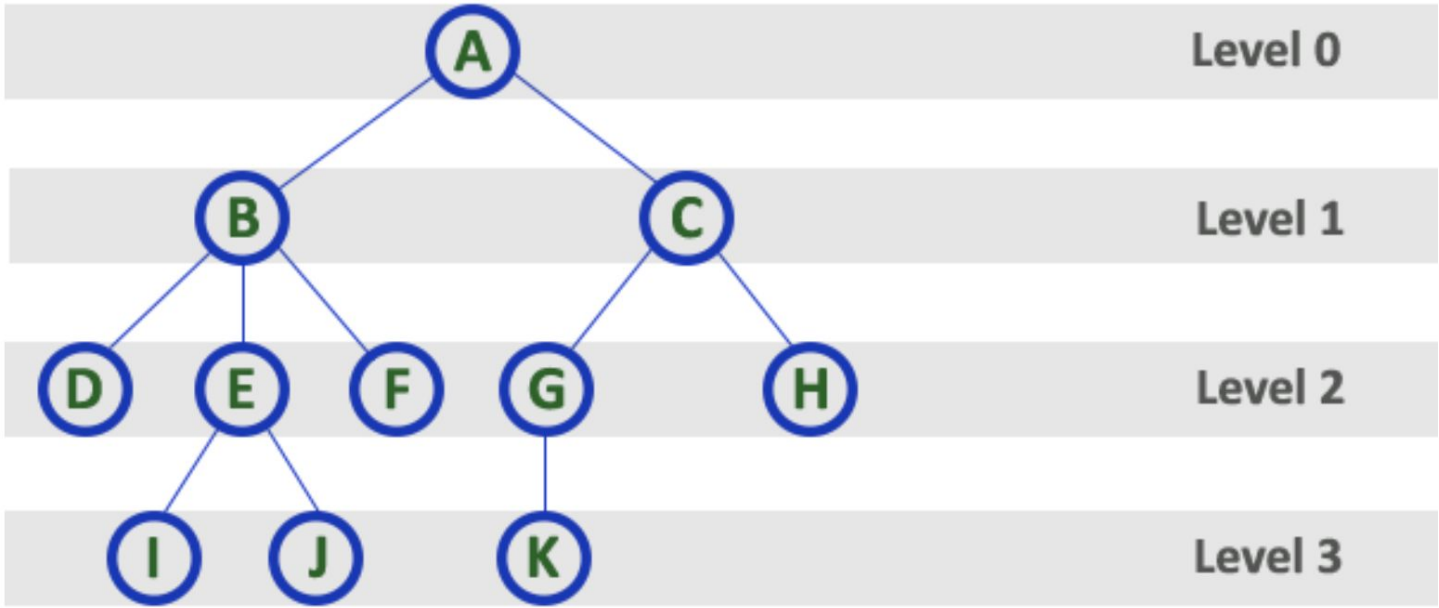
The degree of F = 0



Tree Terminology

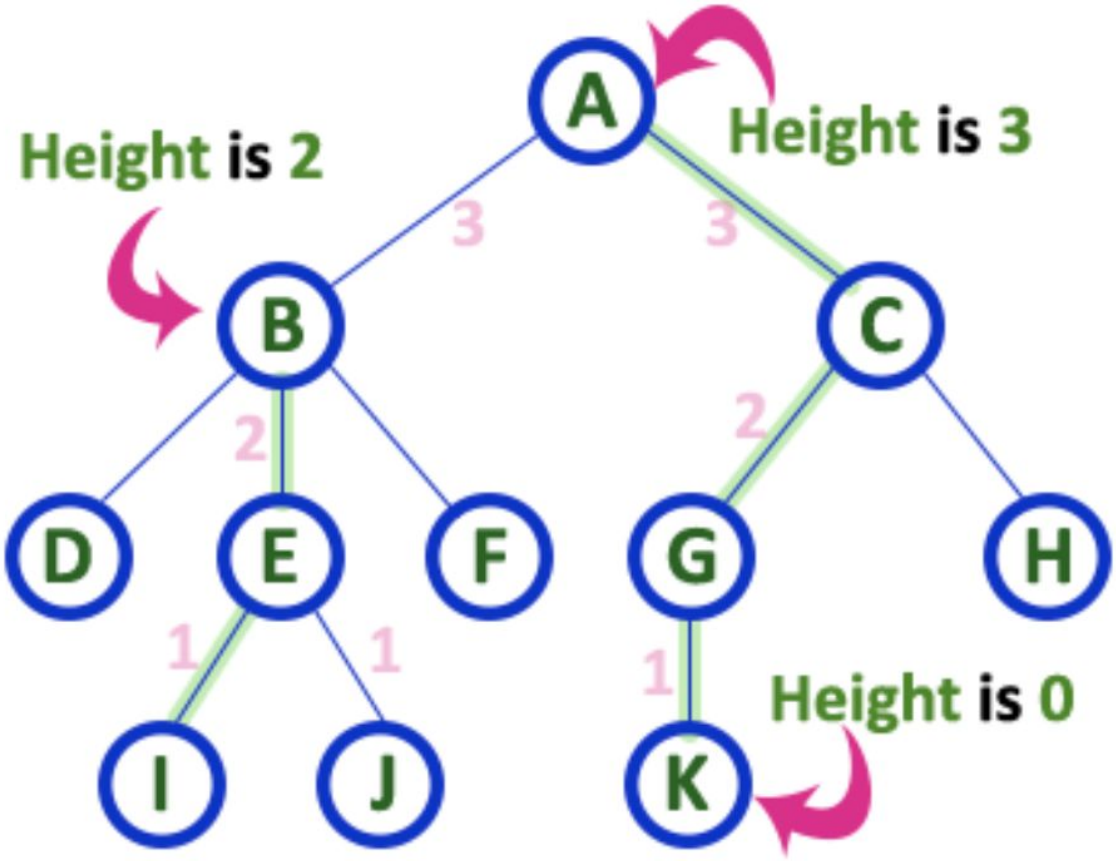
Root is level 0.

Each row of the tree has an incremental level



Tree Terminology

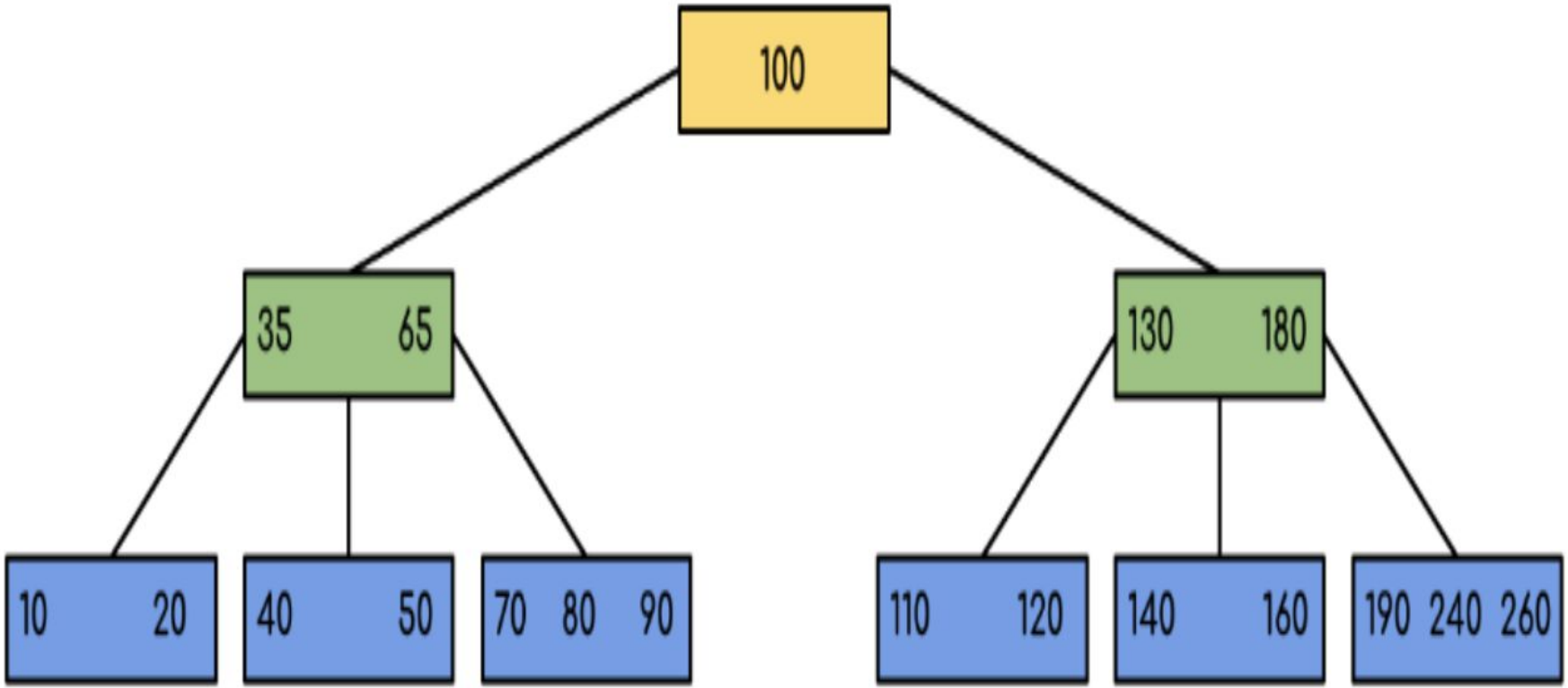
In any tree, the total number of edges from leaf node to a particular node in the longest path is called as height of that node



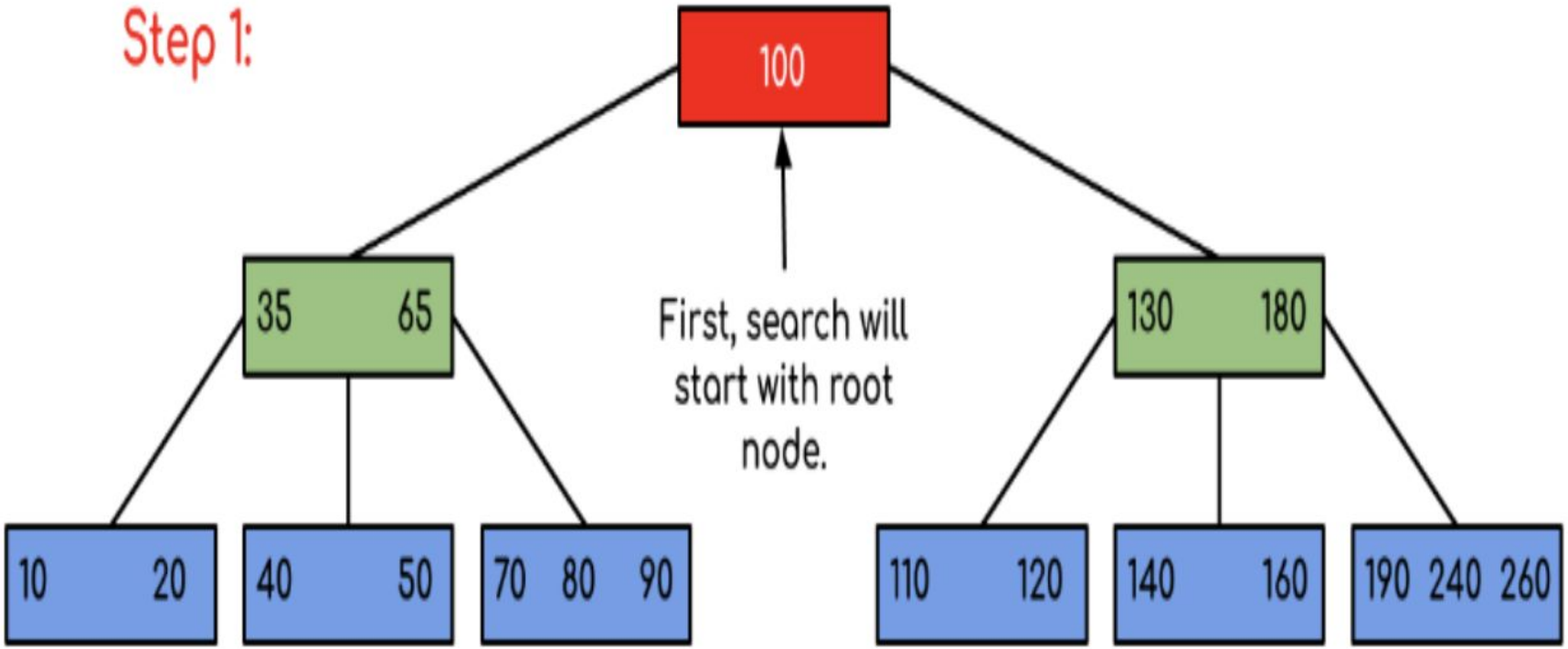
Search operation in B-Tree

Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don’t find k in the leaf node, we return NULL.

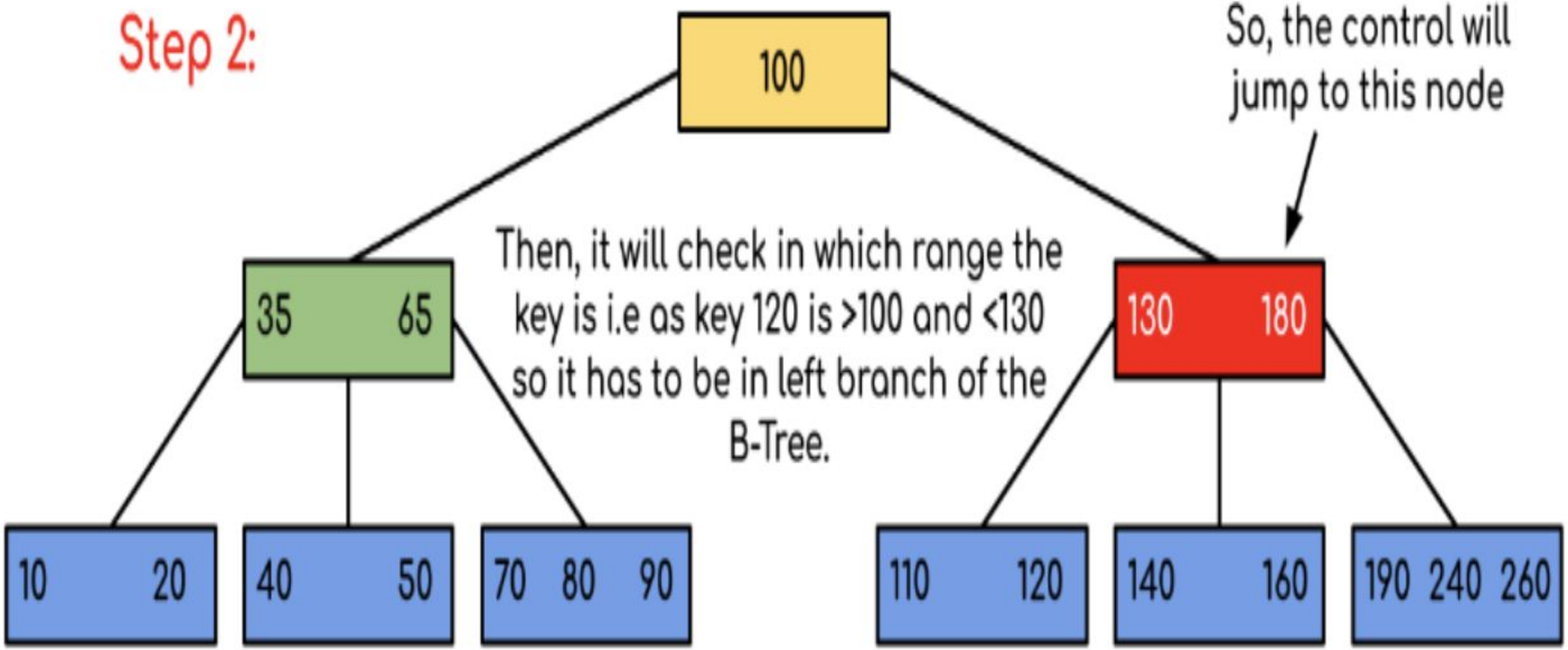
Example: searching 120 in given tree



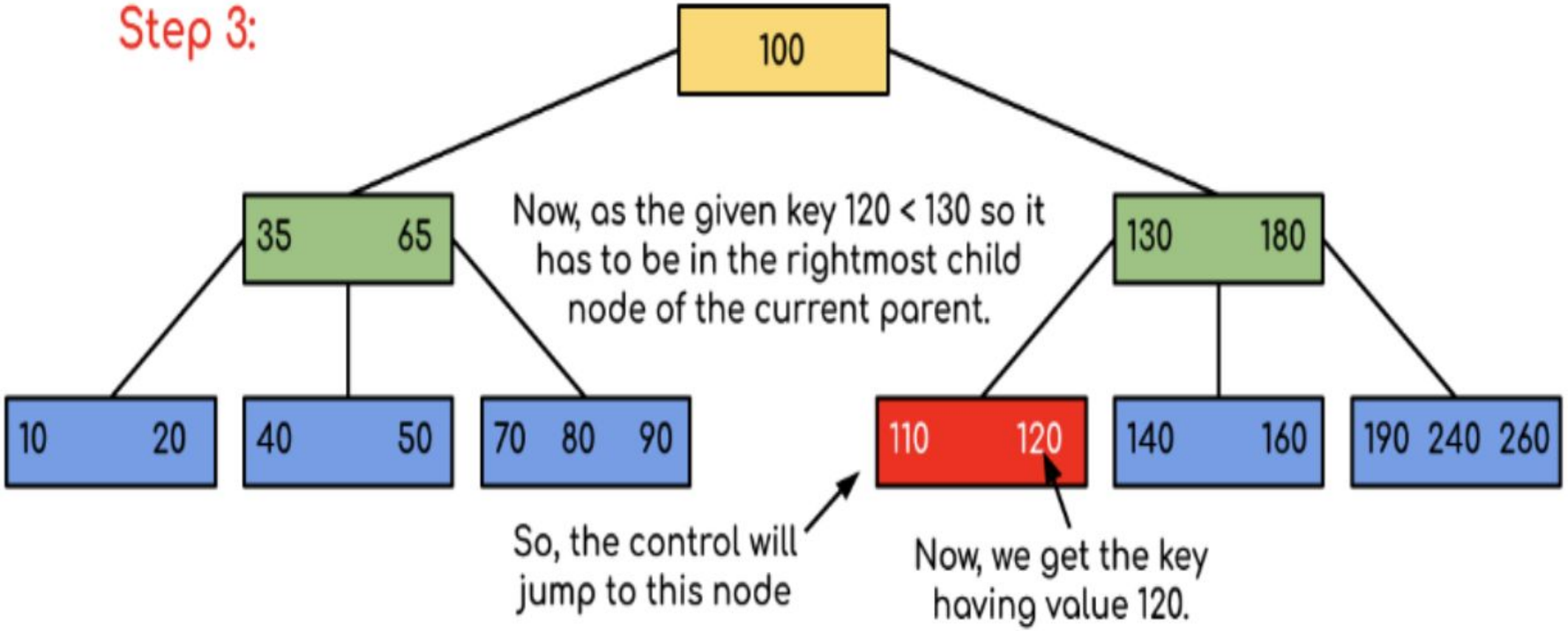
Step 1


Step 1:
100
35
65
First, search will start with root node.
130
180
10
20
40
50
70
80
90
110
120
140
160
190
240
260

Step 2


Step 2:
100
So, the control will jump to this node
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260

Step 3


Step 3:
100
35
65
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
130
180
10
20
40
50
70
80
90
110
120
140
160
190
240
260
So, the control will jump to this node
Now, we get the key having value 120.

Example discussion

In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present.

Example discussion

Similarly if within the above example we are looking for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node.

Example Discussion

And similarly, if we are looking for 90 then as 90 < 100 so it’ll go to the left subtree automatically and therefore the control flow will go similarly as shown within the above example.