ADVANCED DATA STRUCTURES & ALGORITHMS

III SEMESTER	L	T	P	C
	3	-	-	3
ADVANCED DATA STRUCTURES & ALGORITHMS				

Course Objectives:

The main objectives of the course is to

- provide knowledge on advance data structures frequently used in Computer Sciencedomain
- Develop skills in algorithm design techniques popularly used
- Understand the use of various data structures in the algorithm design

Course Outcomes: At the end of the course students will be able to

CO1: Discover the performance of an algorithm using asymptotic notation. (K2)

CO2: Use divide and conquer technique to solve problems.(K3)

CO3: Understand greedy and dynamic programming techniques to solve efficient solutions for optimization problem.(K3)

CO4: Recognize problems suitable for back tracking , branch and bound solutions.(K1)

CO5: Understand the complexity classes NP-Hard and NP-Complete and solve related decision problems.(K2)

UNIT–I:

Introduction to Algorithm Analysis, Space and Time Complexity analysis, Asymptotic Notations, Recursive functions,

AVL Trees – Creation, Insertion, Deletion operations and Applications

B-trees – Creation, Insertion, Deletion operations and Applications

UNIT–II:

Heap Trees (Priority Queues)–Min and Max Heaps, Operations and Applications

Graphs–Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications

UNIT–III:

Divide and Conquer: The General Method, Quick Sort, Merge Sort, Heap Sort, Strassen's

matrix multiplication.

Greedy Method: General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths

UNIT–IV:

Dynamic Programming: General Method, All pairs shortest paths, Single Source Shortest Paths– General Weights (Bellman Ford Algorithm), Optimal Binary Search Trees, 0/1 Knapsack, String Editing.

Backtracking: General Method, 8-Queens Problem, Sum of Subsets problem, Graph Coloring, 0/1 Knapsack Problem

UNIT–V:

Branch and Bound: The General Method, 0/1 Knapsack Problem, Travelling Salesperson problem.

NP Hard and NP Complete Problems: Basic Concepts, Cook's theorem NP Hard Graph Problems: Clique Decision Problem (CDP), Chromatic Number Decision Problem (CNDP), Traveling Salesperson

Textbooks:

1. Fundamentals of Data Structures in C++, Horowitz, Ellis; Sahni, Sartaj; Mehta,Dinesh 2nd Edition Universities Press
2. Computer Algorithms/C++ Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran2nd Edition University Press

Reference Books:

1. Data Structures and program design in C, Robert Kruse, Pearson Education Asia
2. An introduction to Data Structures with applications, Trembley & Sorenson, McGrawHill
3. The Art of Computer Programming, Vol.1: Fundamental Algorithms, Donald E Knuth,Addison-Wesley, 1997.
4. Data Structures using C & C++: Langsam, Augenstein & Tanenbaum, Pearson, 1995
5. Algorithms + Data Structures & Programs:, N.Wirth, PHI
6. Fundamentals of Data Structures in C++: Horowitz Sahni & Mehta, Galgottia Pub.
7. Data structures in Java:, Thomas Standish, Pearson Education Asia

Online Learning Resources:

1. https://www.tutorialspoint.com/advanced_data_structures/index.asp
2. http://peterindia.net/Algorithms.html
3. Abdul Bari, 1. Introduction to Algorithms (youtube.com)
4. https://www.swarnandhra.ac.in/dsv

UNIT-1

Introduction to Data Structures and Algorithms

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

For example, we have some data which has, player's **name** "Virat" and **age** 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Dhoni" 30, "Gambhir" 31, "Sehwag"

If you are aware of Object Oriented programming concepts, then a class also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

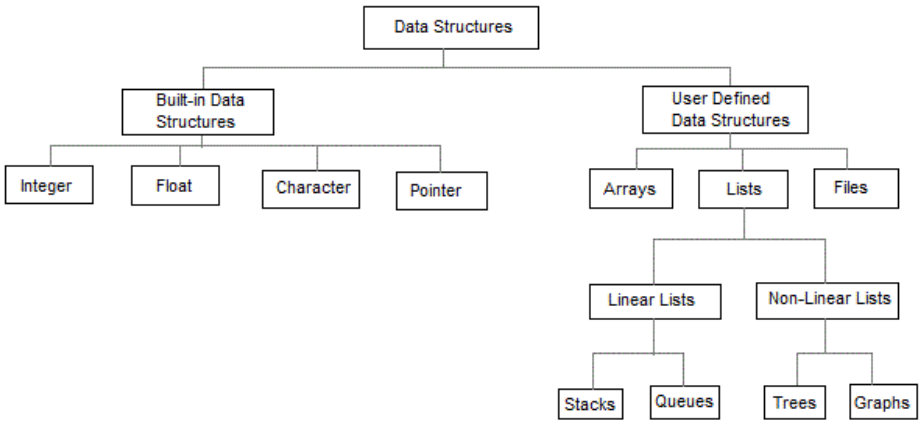
Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.



INTRODUCTION TO DATA STRUCTURES

The data structures can also be classified on the basis of the following characteristics:

Characterstic	**Description**
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**

Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

What is an Algorithm ?

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output.

Or

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just

the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be atleast 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

For example,

An algorithm to add two numbers:

1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

Qualities of a Good Algorithm

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

What is meant by Algorithm Analysis?

Algorithm analysis refers to how to investigate the effectiveness of the algorithm in terms of time and space complexity.

The fundamental purpose of algorithm evaluation is to decide how much time and space an algorithm needs to solve the problem as a feature of the scale of the input.

The time complexity of an algorithm is typically measured in phrases of the wide variety of simple operations (which includes comparisons, assignments, and mathematics operations) that the algorithm plays at the enter records.

The spatial complexity of an algorithm refers to the quantity of reminiscence the algorithm needs to solve the problem as a function of the size of the input.

Algorithm analysis is crucial because it facilitates us to examine different strategies and pick the best one for a given problem.

There are many approaches to research the time and space of algorithms, together with big **O notation, big Omega notation, and big Theta notation**.

Why is Algorithm Analysis important?

1. To forecast the behavior of an algorithm without putting it into action on a specific computer.
2. It is far more convenient to have basic metrics for an algorithm's efficiency than to develop the algorithm and access its efficiency each time a specific parameter in the underlying computer system changes.
3. It is hard to predict an algorithm's exact behavior. There are far too many variables to consider.
4. As a result, the analysis is simply an approximation; it is not perfect.
5. More significantly, by comparing several algorithms, we can identify which one is ideal for our needs.

Types of Algorithm Analysis:

There are numerous types of algorithm analysis, which can be generally used to measure the performance and efficiency of algorithms:

1. **Time complexity evaluation:** This kind of analysis measures the running time of an algorithm as a characteristic of the input length. It typically entails counting the quantity of primary operations completed by way of the algorithm, such as comparisons, mathematics operations, and reminiscence accesses.
2. **Space complexity evaluation:** This form of evaluation measures the amount of memory required via an algorithm as a characteristic of the enter size. It typically includes counting the variety of variables and information systems utilized by the algorithm, as well as the size of each of these records structures.
3. **Worst-case evaluation:** This type of analysis measures the worst-case running time or space utilization of an algorithm, assuming the maximum toughest viable for the algorithm to deal with.
4. **Average-case analysis:** This kind of evaluation measures the predicted walking time or area usage of an algorithm, assuming a probabilistic distribution of inputs.
5. **Best-case evaluation:** This form of analysis measures the nice case running time or area utilization of an algorithm, assuming the input is the easiest possible for the algorithm to address.
6. **Asymptotic analysis:** This sort of analysis measures the overall performance of an algorithm as the enter size methods infinity. It normally includes the usage of mathematical notation to describe the algorithm's time or area usage, including $O(n)$, $\Omega(n)$, or $\Theta(n)$.

These sets of algorithm analysis are all useful for information and evaluating the overall performance of various algorithms, and for predicting how properly an algorithm will scale to large problem sizes.

Applications:

Algorithms are central to computer science and are used in many different fields. Here is an example of how the algorithm is used in various applications.

1. **Search engines:** Google and other search engines use complex algorithms to index and rank websites, ensuring that users get the most relevant search results.
2. **Machine Learning:** Machine-learning algorithms are used to train computer programs to learn from data and make predictions or decisions based on that data. It is used in applications such as image recognition, speech recognition, and natural language processing.
3. **Cryptography:** Cryptographic algorithms are used to secure data transmission and protect sensitive information such as credit card numbers and passwords.
4. **Optimization:** Optimization algorithms are used to find the optimal solution to a problem, such as the shortest path between two points or the most efficient resource allocation path.
5. **Finance:** Algorithms are used in finance for applications such as risk assessment, fraud detection, and frequent trading.
6. **Games:** Game developers use artificial intelligence and algorithms to navigate, allowing game characters to make intelligent decisions and navigate game environments more efficiently
7. **Data Analytics:** Data analytics applications use algorithms to process large amounts of data and extract meaningful insights, such as trends and patterns.
8. **Robotics:** Robotics algorithms are used to control robots and enable them to perform complex tasks such as recognizing and manipulating objects.

Time Complexity & Space Complexity

Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

Space complexity *is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*
Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = Auxiliary Space + Input space

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

   It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

   Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

   For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A().

3. **Data Space**

    Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Now let's learn how to compute space complexity by taking a few examples:

```
{
```

```
    int z = a + b + c;
    return(z);
}
```

In the above expression, variables a, b, c and z are all integer types, hence they will take up 4 bytes each, so total memory requirement will be (4(4) + 4) = 20 bytes, this additional 4 bytes is for **return value**.

And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

Let's have another example, this time a bit complex one,

```
// n is the length of array a[]

int sum(int a[], int n)

{

            int x = 0;                      // 4 bytes for x

            for(int i = 0; i < n; i++)  // 4 bytes for i

            {

               x  = x + a[i];

            }

            return(x);

}
```

- In the above code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.

Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as **Linear Space Complexity.**

Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.

Time Complexity of Algorithms:

For any defined problem, there can be N number of solution. This is true in general.

If I have a problem and I discuss about the problem with all of my friends, they will all suggest me different solutions.

And I am the one who has to decide which solution is the best based on the circumstances.

Similarly for any problem which must be solved using a program, there can be infinite number of solutions.

Let's take a simple example to understand this. Below we have two different algorithms to find square of a number(for some time, forget that square of any number n is n*n):

One solution to this problem can be, running a loop for n times, starting with the number n and adding n to it, every time.

```
/*
   we have to calculate the square of n
*/
for i=1 to n
```

```
    do n = n + n

// when the loop ends n will hold its square

return n
```

Or, we can simply use a mathematical operator * to find the square.

```
/*

    we have to calculate the square of n

*/

return n*n
```

In the above two simple algorithms, you saw how a single problem can have many solutions.

While the first solution required a loop which will execute for n number of times, the second solution used a mathematical operator * to return the result in one line.

So which one is the better approach, of course the second one.

What is Time Complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**.

It's an asymptotic notation to represent the time complexity.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.

Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n atleast and as the value of n will increase the time taken will also increase.

While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation.

This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
   statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
   for(j=0; j < N;j++)
   {
   statement;
   }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N.

When N doubles, the running time increases by N * N.

```
while(low <= high)

{

   mid = (low + high) / 2;

   if (target < list[mid])

      high = mid - 1;

   else if (target > list[mid])

      low = mid + 1;

   else break;

}
```

This is an algorithm to break a set of numbers into halves, to search a particular field. Now, this algorithm will have a **Logarithmic** Time Complexity.

The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here).

This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)

{

   int pivot = partition(list, left, right);

   quicksort(list, left, pivot - 1);

   quicksort(list, pivot + 1, right);

}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort.

Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list).

Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

> **NOTE:** *In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.*

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

Suppose you've calculated that an algorithm takes f(n) operations, where,

```
f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n
```

Since this polynomial grows at the same rate as $\mathbf{n^2}$, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of $\mathbf{n^2}$, the time complexity can be best represented as **Theta(n^2)**.

Time Complexity vs. Space Complexity

Now we know the basics of time and space complexity and how it can be calculated for an algorithm or program.

Time Complexity	Space Complexity

Time Complexity	Space Complexity
Calculates time needed	Calculates memory space needed
Counts time for all statements.	Counts memory space for all variables even inputs and outputs
Depends mostly on input data size	Depends mostly on auxiliary variables size
More important for solution optimization	Less important with modern hardware
Time complexity of merge sort is **O(n log n)**	Space complexity of merge sort is **O(n)**

Asymptotic Analysis: Big-O Notation and More

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

Asymptotic notations are the **mathematical notations** used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Asymptotic Notations and how to calculate them.

In computing, asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance based on the input size.

For example, the running time of one operation is computed as ***f*(n)**, and maybe for another operation, it is computed as ***g*(n2)**.

This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases.

Similarly, the running time of both operations will be nearly the same if **n** is small in value.

Usually, the analysis of an algorithm is done based on three cases:

1. **Best Case (Omega Notation (Ω))**
2. **Average Case (Theta Notation (Θ))**
3. **Worst Case (O Notation(O))**

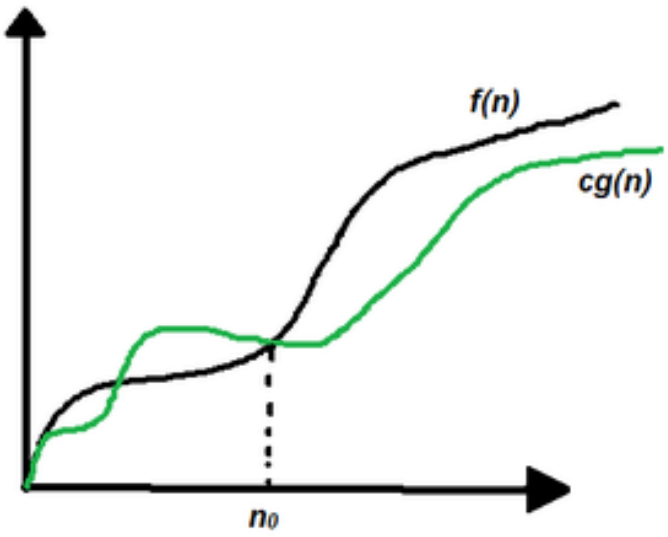
All of these notations are discussed below in detail:

Omega (Ω) Notation:

Omega (Ω) notation specifies the asymptotic lower bound for a function f(n). For a given function g(n), Ω(g(n)) is denoted by:

Ω (g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \leq c*g(n) \leq f(n)$ for all $n \geq n_0$}.

This means that, f(n) = Ω(g(n)), If there are positive constants n_0 and c such that, to the right of n_0 the f(n) always lies on or above c*g(n).



Graphical representation

Follow the steps below to calculate Ω for a program:

1. Break the program into smaller segments.
2. Find the number of operations performed for each segment (in terms of the input size) assuming the given input is such that the program takes the least amount of time.
3. Add up all the operations and simplify it, let's say it is f(n).
4. Remove all the constants and choose the term having the highest order or any other function which is always less than f(n) when n tends to infinity, let say it is g(n) then, Omega (Ω) of f(n) is Ω(g(n)).

For example, consider the below pseudo code.

Sum of large numbers

- Pseudo Code

```
void fun(n) {
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {

      // Do some constant work (no loop or
      // function call)
    }

    for (i = 0; i < n; i++) {
      // Do some constant work (no loop or
      // function call)
    }
}
```

The time taken by the above code can be written as **a*n^2 + b*n + c** where are a, b and c are some machine specific constants.

In this case, the highest growing term is a*n^2.

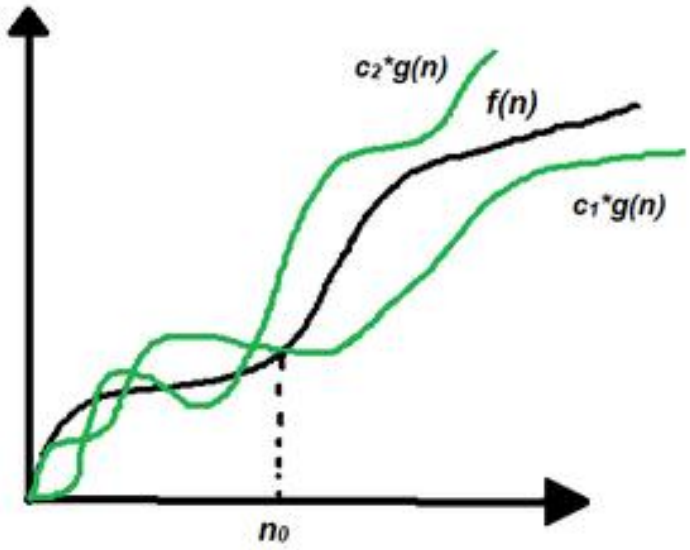
So we can say that the time complexity of the code is either Ω(n^2) or Ω(n) or Ω(Log n) or Ω(1)

Omega notation doesn't really help to analyze an algorithm because it is bogus to evaluate an algorithm for the best cases of inputs.

Theta (Θ) Notation:
Big-Theta(Θ) notation specifies a bound for a function f(n). For a given function g(n), Θ(g(n)) is denoted by:

Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1*g(n) ≤ f(n) ≤ c2*g(n) for all n ≥ n0}.

This means that, f(n) = Θ(g(n)), If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or above c1*g(n) and below c2*g(n).



Graphical representation

Follow the steps below to calculate Θ for a program:

1. Break the program into smaller segments.
2. Find the number of operations performed for each segment(in terms of the input size) assuming the given input is such that the program takes the least amount of time.
3. Add up all the operations and simplify it, let's say it is f(n).
4. Remove all the constants and choose the term having the highest order. Let say it is g(n) then, Omega (Θ) of f(n) is Θ(g(n)).

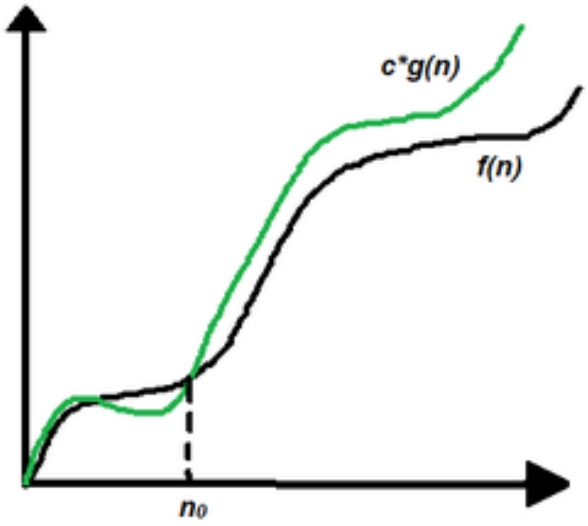
As an **example**. let us consider the above pseudo code only. In this case, the highest growing term is a*n^2. So the time complexity of the code is Θ(n^2)

Big – O Notation:

Big – O (O) notation specifies the asymptotic upper bound for a function f(n). For a given function g(n), O(g(n)) is denoted by:

O (g(n)) = {f(n): there exist positive constants c and n_0 such that f(n) ≤ c*g(n) for all n ≥ n_0}.

This means that, f(n) = O(g(n)), If there are positive constants n_0 and c such that, to the right of n_0 the f(n) always lies on or below c*g(n).



Graphical representation

Follow the steps below to calculate O for a program:

1. Break the program into smaller segments.

2. Find the number of operations performed for each segment (in terms of the input size) assuming the given input is such that the program takes the maximum time i.e the worst-case scenario.
3. Add up all the operations and simplify it, let's say it is **f(n).**
4. Remove all the constants and choose the term having the highest order because for **n** tends to infinity the constants and the lower order terms in **f(n)** will be insignificant, let say the function is **g(n)** then, big-O notation is **O(g(n))** or **O(h(n))** where h(n) has higher order of growth than g(n)

As an **example**.

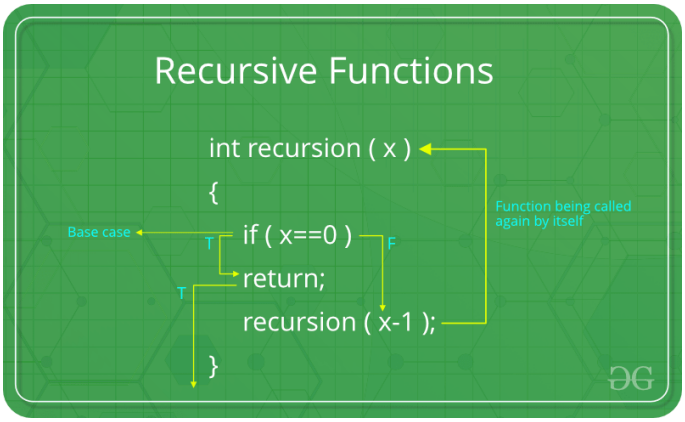
let us consider the above pseudo code only. In this case, the highest growing term is a*n^2.

So the time complexity of the code is O(n^2) or O(n^3) or O(n Log n) or any other term can be put inside O that has higher growth than n^2.

Recursive Functions

A Recursive function can be defined as a routine that calls itself directly or indirectly.

In other words, a recursive function is a function that solves a problem by solving smaller instances of the same problem. This technique is commonly used in programming to solve problems that can be broken down into simpler, similar subproblems.



Need of Recursive Function:

A recursive function is a function that solves a problem by solving smaller instances of the same problem. This technique is often used in programming to solve problems that can be broken down into simpler, similar subproblems.

1. Solving complex tasks:
Recursive functions break complex problems into smaller instances of the same problem, resulting in compact and readable code.

2. Divide and Conquer:
Recursive functions are suitable for divide-and-conquer algorithms such as merge sort and quicksort, breaking problems into smaller subproblems, solving them recursively, and merging the solutions with the original problem.

3. Backtracking:
Recursive backtracking is ideal for exploring and solving problems like N-Queens and Sudoku.

4. Dynamic programming:
Recursive functions efficiently solve dynamic programming problems by solving subproblems and combining their solutions into a complete solution.

5. Tree and graph structures:
Recursive functions are great for working with tree and graph structures, simplifying traversal and pattern recognition tasks**.**

How to write a Recursive Function:

Components of a recursive function:

Base case: Every recursive function must have a base case. The base case is the simplest scenario that does not require further recursion. This termination condition prevents the function from calling itself indefinitely. Without a proper base case, a recursive function can lead to infinite recursion.

Recursive case: In the recursive case, the function calls itself with the modified arguments. This is the essence of recursion – solving a larger

problem by breaking it down into smaller instances of the same problem. The recursive case should move closer to the base case with each iteration.

Let's consider the example of factorial of number:

In this example, the base case is when **n** is **0**, and the function returns **1**. The recursive case multiplies **n** with the result of the function called with parameter **n – 1**. The process continues until the base case is reached.
It's essential to ensure that the recursive function has a correct base case and that the recursive calls lead to the base case, otherwise, the procedure might run indefinitely, leading to a stack overflow (exceeding the available memory allocated for function calls).

Below is the implementation of factorial of a number:

```
#include <stdio.h>
// Function to calculate factorial using recursion
long factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
int main() {
    int num;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    printf("Factorial of %d = %ld\n", num, factorial(num));
    return 0;
}
```

Output

```
Factorial of 4 is:24
```

Time Complexity: O(n)
Auxiliary Space: O(n)

AVL Trees:

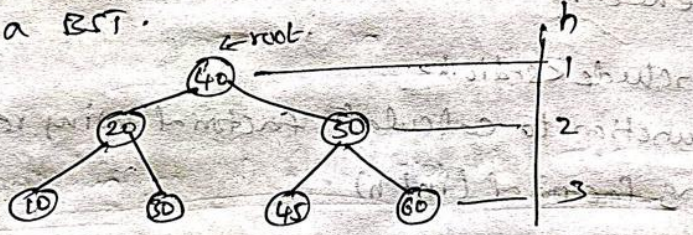
AVL TREES

Abdul Bari

1. What is a Binary Search Tree (BST)
2. Drawbacks of BST
3. How BST can be improved?
4. What is an AVL Tree?
5. Rotations in AVL Tree.
6. How to Create AVL Tree?

Before Learn about AVL Trees, we should know some back ground.

Let us start with Binary Search tree. Below is a BST.



The keys are arranged such that for any node, all the elements in the left hand side is smaller than that node and all the elements in the right hand side are greater than that node.

So for node 20, 30 is greater and 10 is smaller.

→ The Reason for this is it is useful for Searching.

→ Suppose I want to search for a key element key=30;

I start search with the root element, wheather wheather the 30 is less than or greater than root 40.

30 < 40, so go on left side

Now compare 30 with 20

30 > 20, so go on right hand side

Now 30 is found.

∴ The total comparison for the element is 3.

→ Again take another element for searching i.e element/key = 60.

60 is compared with root 40.

60 > 40, so go on right hand side

Now 60 is compare with 50.

60 > 50, so go on right hand side

Now the key 60 is found.

∴ The total comparison for the element is 3

→ one more key = 32.

32 is compared with root 40

32 < 40, go to left hand side.

32 is compared with 20

32 > 20, so go to right hand side

Now 32 is compared with 30

32 > 30, so go to right hand side

But there is no element.

so key 32 is not found., it is unsuccessful search. (search fail)

→ The search for an element x possibility is wheather it is found or not found.

→ For unsuccessful search, the total no of comparisons are 3 only.

So, max no of comparisons requires for searching any elements in the tree depends on the height.

Height of the tree = 3.

∴ The total no of comparisons are 3.

→ The BST is Binary tree only, What is the height of the Binary tree?

The height of the binary tree can be

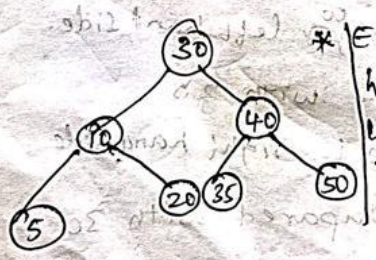
min	log n
max	n.

Problem or draw Cases of BST :

Keys: 30, 40, 10, 50, 20, 5, 35

Key = 50, 40, 35, 30, 20, 10, 5.

Let us create a BST : Insert one element at a time



* Every element should check from root

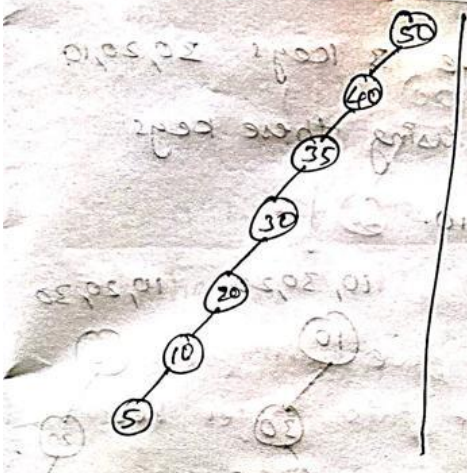
h

log n

For ex 50 is again want to insert, it is not possible, duplication not allowed.

The height of this BST is log n

Keys = 50, 40, 35, 30, 20, 10, 5. [same no of keys but order is different]



{height of this BST = n}

n

So this is the problem with BST, the final height of the BST is not in our control.

→ The height may be logn or the height may be n., it depends how the key elements are inserted.

→ If the height is logn, the time taken to search an element is logn, and if the height is n, the time taken to search an element is n.

It is similar to linear search

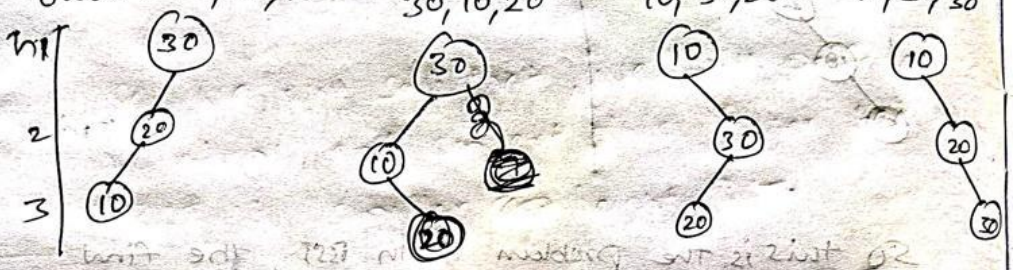
P.T.O

How can we improve BST Trees?

Let us see, I have 3 keys 30, 20, 10 to create a BST by using these keys.

Keys - 30, 20, 10.

order → 30, 20, 10 | 30, 10, 20 | 10, 30, 20 | 10, 20, 30



For different orders, we get different shapes of BST.

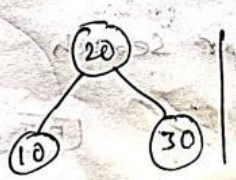
For 3 keys, How many orders are possible?

It means $3! = 6$ orders.

Above, already we have four, so 2 more I will take

20, 10, 30

20, 30, 10



→ If we look at the above trees the height are 3, (Max), and below trees height are 2. (Min height)

→ This means that we have a set of keys we can form a long BST or short BST also.

* What we prefer is a minimum height BST

So that the search time can be reduced.
This is the objective.

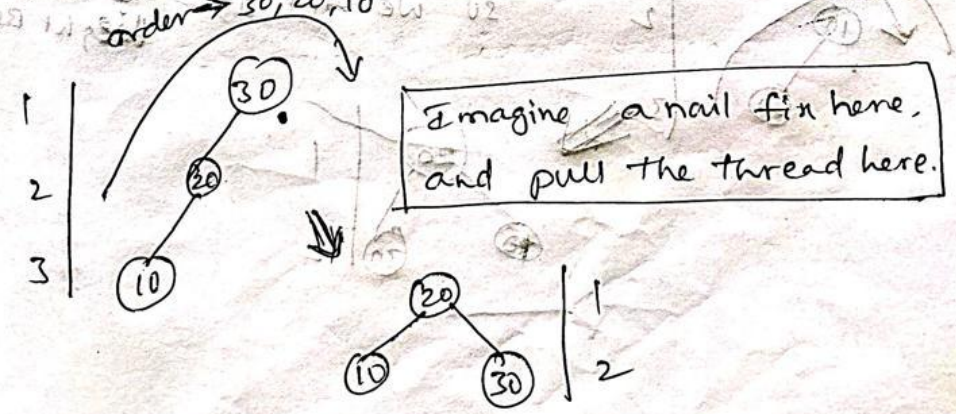
* So, the drawback of the BST is that the same keys can get different shape BST's, can be of max height or of minimum height.

* What we prefer is minimum height BST.

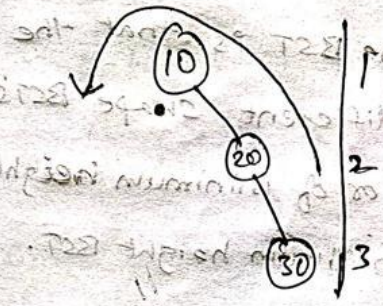
⇒ Now we can see the BST can improved or not.

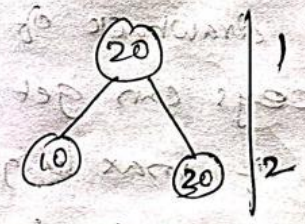
→ By looking into above shapes BST's, we can take minimum instead of max, but we need a method to take the min height BST.

For that we define rotations, we say that we rotate this tree around this node that will pull this 30 this side. 20 will move up and 10 will move at this place, so we say we will perform rotation around this node, and we will change the shape of this node like this.



10, 20, 30





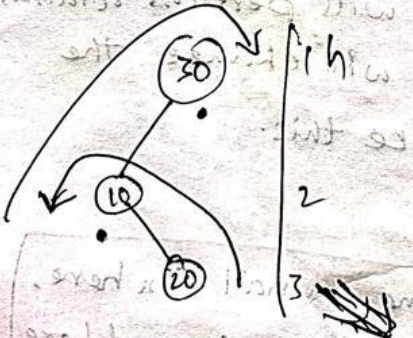
So we will say that we will perform rotations and convert this larger height BST to smaller height BST.

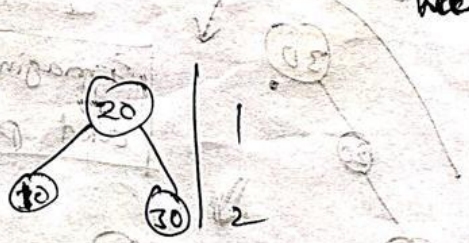
⇒ So, this is way to improve the BST and give idea of AVL Trees.

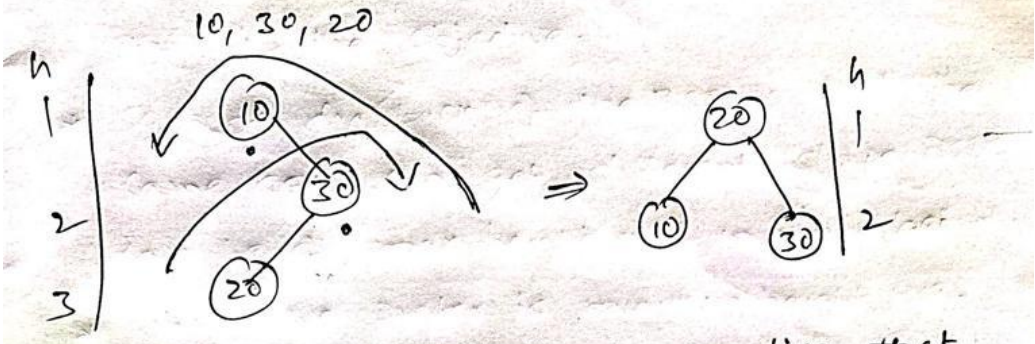
Sometimes one step can't do, First we will rotate around this 10, 10 will come this side and 20 will come up, than we will rotate 30, so 30 this side and 20 will move up and 10 will remain on the Left hand side.

So we will get the min height BST

30, 10, 20







Here we may arise one question that is we have only 3 node, but have many no. of nodes, can rotations done?

→ Rotations are done only for 3 nodes always. What will be size of BT, we will look for only 3 nodes.

AVL Trees: Rotations, Insertion, Deletion with C Example

What are AVL Trees?

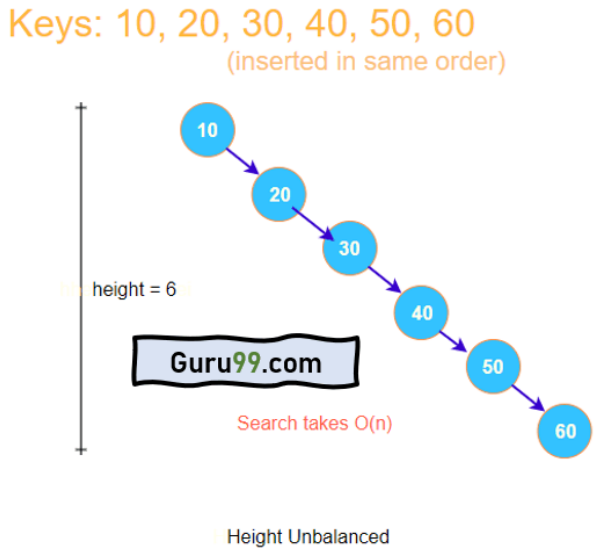
AVL trees are binary search trees in which the difference between the height of the left and right subtree is either -1, 0, or +1.

AVL trees are also called a self-balancing binary search tree. These trees help to maintain the logarithmic search time. It is named after its inventors (AVL) Adelson, Velsky, and Landis.

How does AVL Tree work?

To better understand the need for AVL trees, let us look at some disadvantages of simple binary search trees.

Consider the following keys inserted in the given order in the binary search tree.

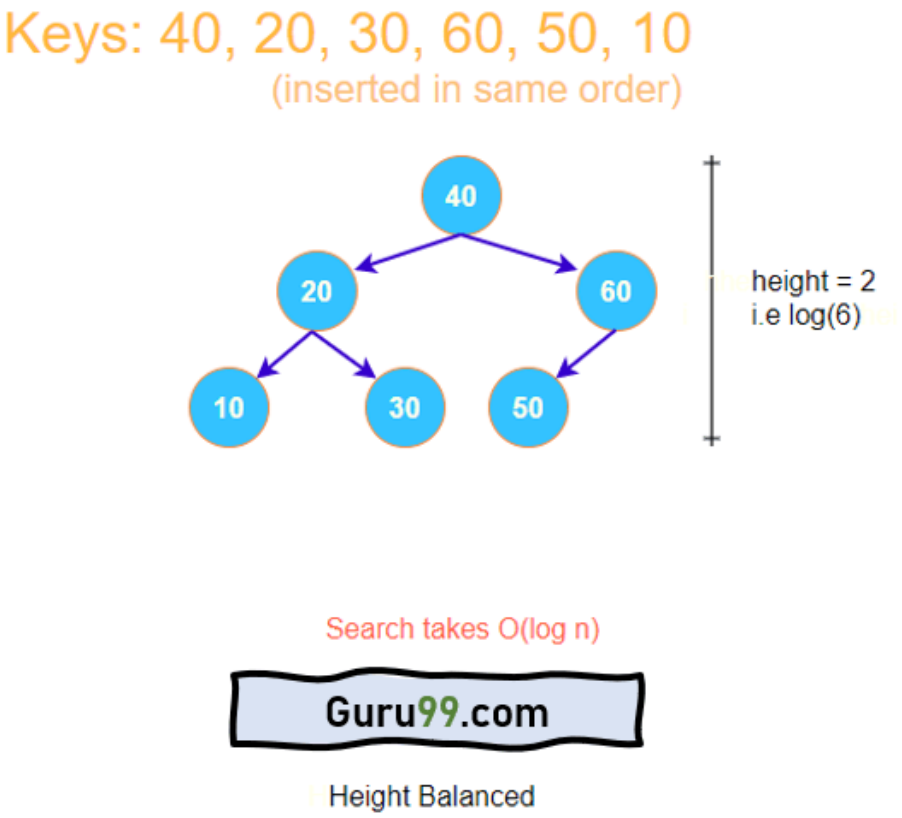


AVL tree visualization

The height of the tree grows linearly in size when we insert the keys in increasing order of their value. Thus, the search operation, at worst, takes O(n).

It takes linear time to search for an element; hence there is no use of using the Binary Search Tree structure. On the other hand, if the height of the tree is balanced, we get better searching time.

Let us now look at the same keys but inserted in a different order.



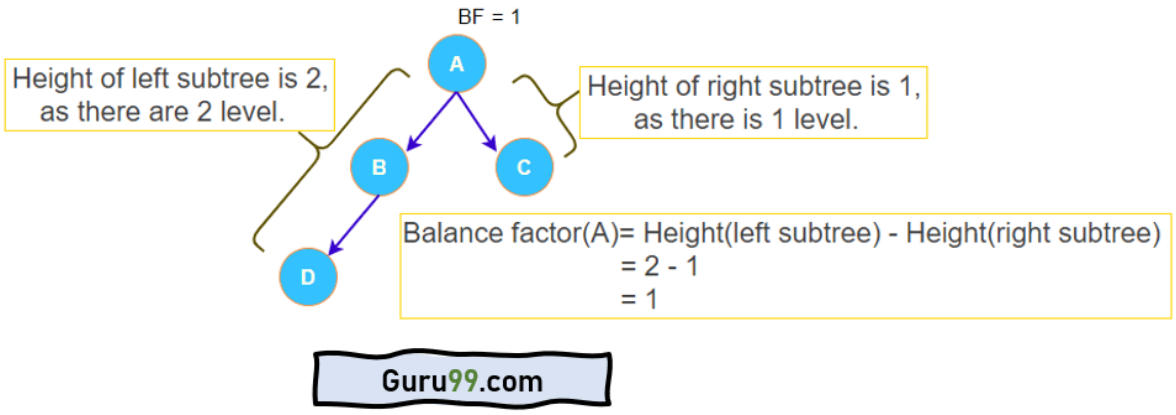
Here, the keys are the same, but since they are inserted in a different order, they take different positions, and the height of the tree remains balanced. Hence search will not take more than O(log n) for any element of the tree. It is now evident that if insertion is done correctly, the tree's height can be kept balanced.

In AVL trees, we keep a check on the height of the tree during insertion operation. Modifications are made to maintain the balanced height without violating the fundamental properties of Binary Search Tree.

Balance Factor in AVL Trees

Balance factor (BF) is a fundamental attribute of every node in AVL trees that helps to monitor the tree's height.

Properties of Balance Factor



Balance factor AVL tree

- The balance factor is known as the difference between the height of the left subtree and the right subtree.
- Balance factor(node) = height(node->left) – height(node->right)
- Allowed values of BF are –1, 0, and +1.
- The value –1 indicates that the right sub-tree contains one extra, i.e., the tree is right heavy.
- The value +1 indicates that the left sub-tree contains one extra, i.e., the tree is left heavy.
- The value 0 shows that the tree includes equal nodes on each side, i.e., the tree is perfectly balanced.

AVL Rotations

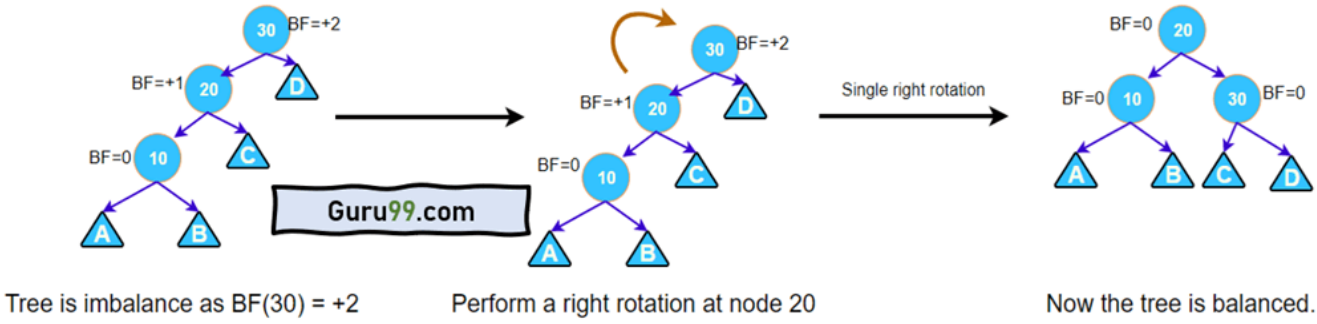
To make the AVL Tree balance itself, when inserting or deleting a node from the tree, rotations are performed.

We perform the following LL rotation, RR rotation, LR rotation, and RL rotation.

- Left – Left Rotation
- Right – Right Rotation
- Right – Left Rotation
- Left – Right Rotation

Left – Left Rotation

This rotation is performed when a new node is inserted at the left child of the left subtree.

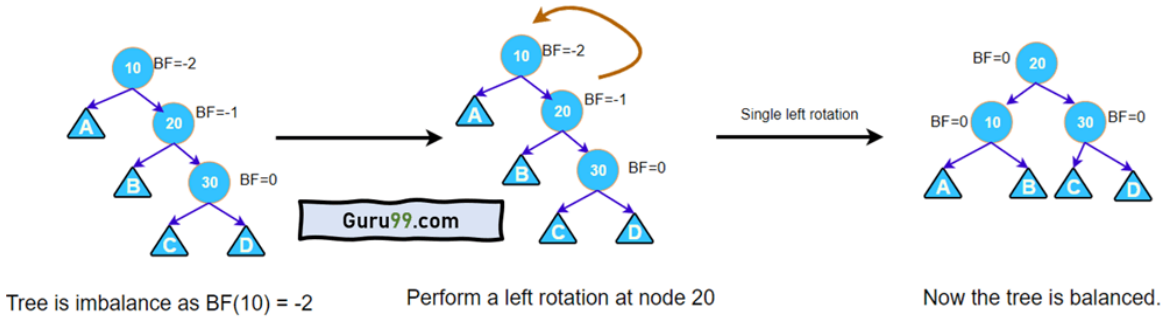


AVL Tree Left – Left Rotation

A single right rotation is performed. This type of rotation is identified when a node has a balanced factor as +2, and its left-child has a balance factor as +1.

Right – Right Rotation

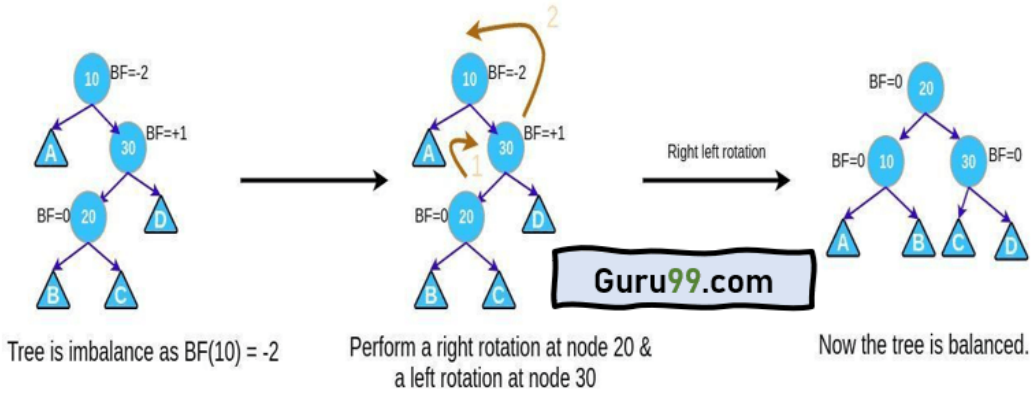
This rotation is performed when a new node is inserted at the right child of the right subtree.



A single left rotation is performed. This type of rotation is identified when a node has a balanced factor as -2, and its right-child has a balance factor as -1.

Right – Left Rotation

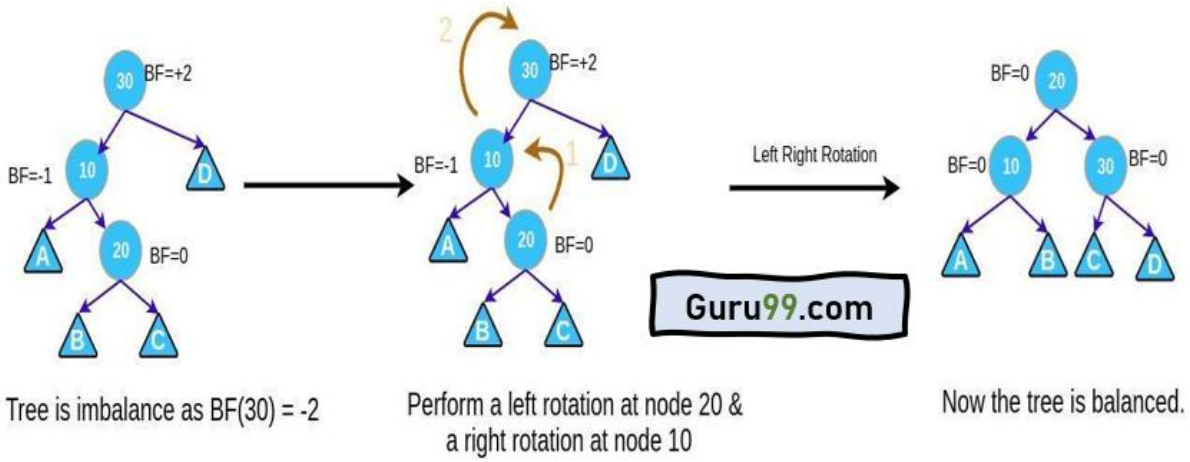
This rotation is performed when a new node is inserted at the right child of the left subtree.



This rotation is performed when a node has a balance factor as –2, and its right-child has a balance factor as +1.

Left – Right Rotation

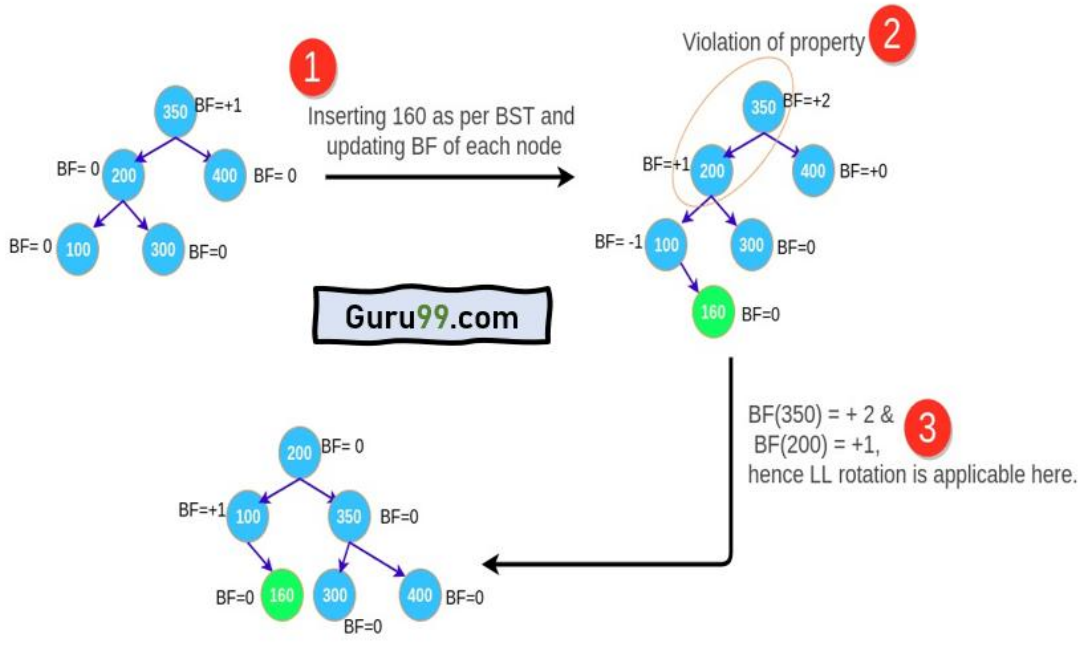
This rotation is performed when a new node is inserted at the left child of the right subtree.



This rotation is performed when a node has a balance factor as +2, and its left-child has a balance factor as -1.

Insertion in AVL Trees

Insert operation is almost the same as in simple binary search trees. After every insertion, we balance the height of the tree. Insert operation takes O(log n) worst time complexity.



AVL tree insertion implementation

Step 1: Insert the node in the AVL tree using the same insertion algorithm of BST. In the above example, insert 160.

Step 2: Once the node is added, the balance factor of each node is updated. After 160 is inserted, the balance factor of every node is updated.

Step 3: Now check if any node violates the range of the balance factor if the balance factor is violated, then perform rotations using the below case. In the above example, the balance factor of 350 is violated and case 1 becomes applicable there, we perform LL rotation and the tree is balanced again.

1. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
2. If BF(node) = -2 and BF(node -> right-child) = 1, perform RR rotation.
3. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation.
4. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.

Deletion in AVL Trees

Deletion is also very straight forward. We delete using the same logic as in simple binary search trees. After deletion, we restructure the tree, if needed, to maintain its balanced height.

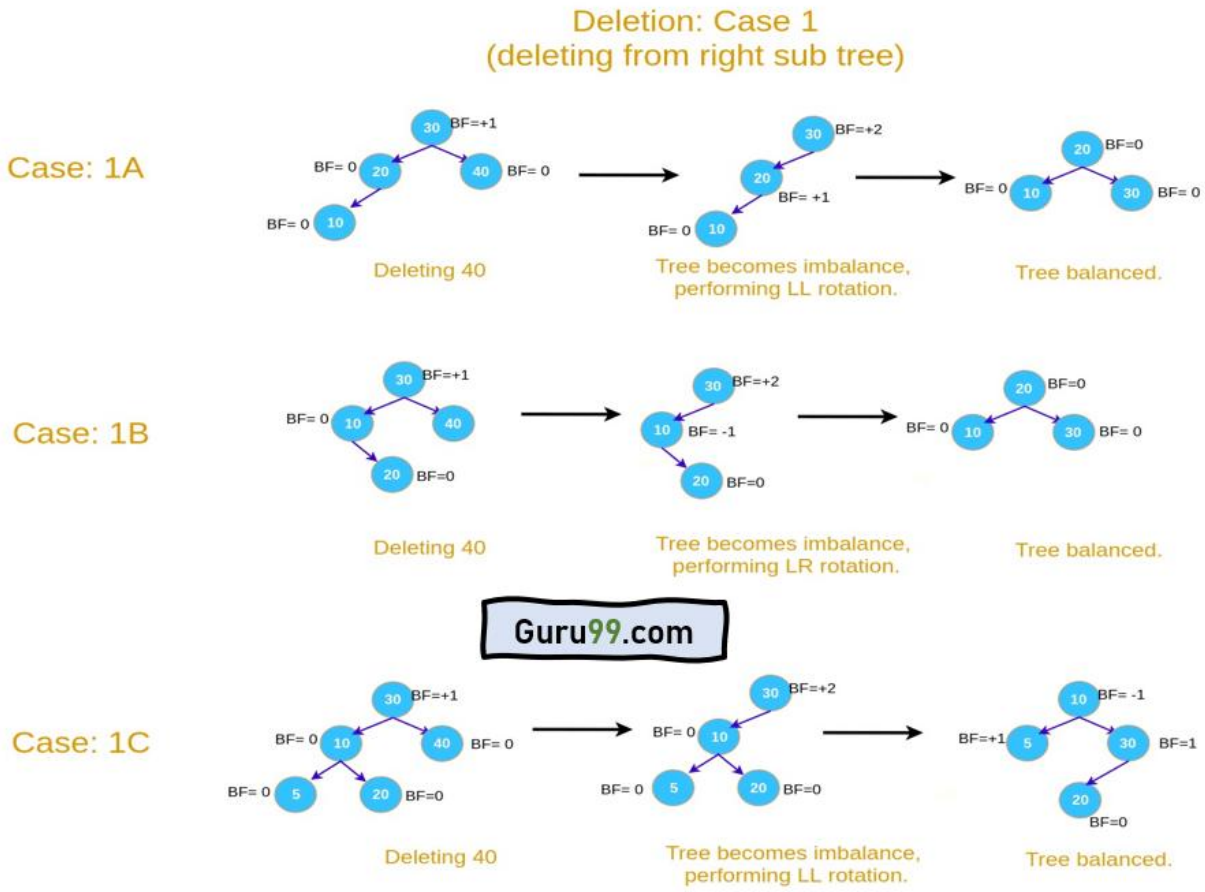
Step 1: Find the element in the tree.

Step 2: Delete the node, as per the BST Deletion.

Step 3: Two cases are possible:-

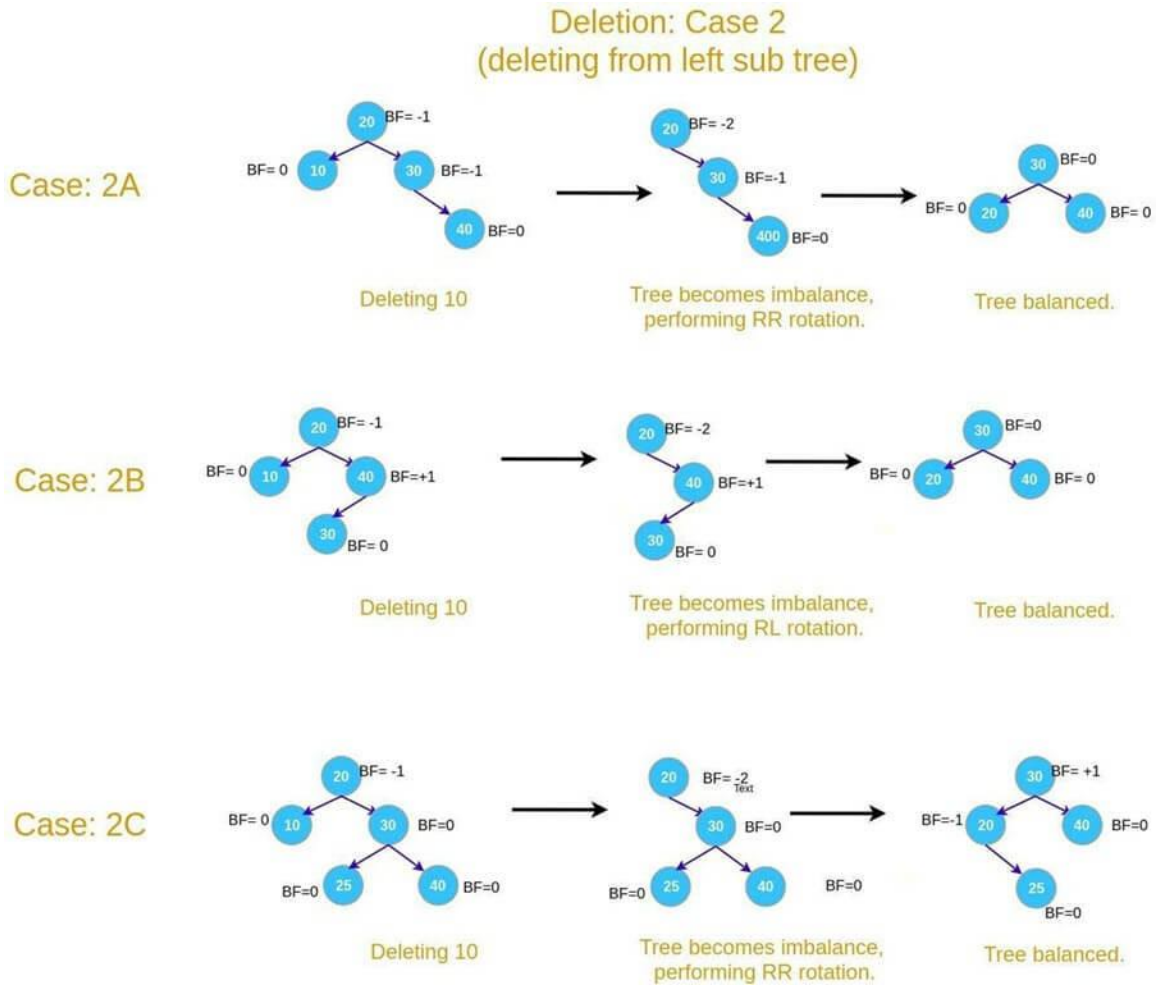
Case 1: Deleting from the right subtree.

- 1A. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
- 1B. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.
- 1C. If BF(node) = +2 and BF(node -> left-child) = 0, perform LL rotation.



Case 2: Deleting from left subtree.

- 2A. If BF(node) = -2 and BF(node -> right-child) = -1, perform RR rotation.
- 2B. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation.
- 2C. If BF(node) = -2 and BF(node -> right-child) = 0, perform RR rotation.



Construct an AVL tree for a given set of elements, which are stored in a file. And implement insert and delete operation on the constructed tree. Write contents of tree into a new file using in-order.

// C program to implement the avl tree

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

// AVL Tree node
struct Node {
```

```
    int key;
    struct Node* left;
    struct Node* right;
    int height;
};

// Function to get height of the node
int getHeight(struct Node* n)
{
    if (n == NULL)
        return 0;
    return n->height;
}

// Function to create a new node
struct Node* createNode(int key)
{
    struct Node* node
        = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // New node is initially added at leaf
    return node;
}

// Utility function to get the maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Function to get balance factor of a node
int getBalanceFactor(struct Node* n)
{
    if (n == NULL)
        return 0;
    return getHeight(n->left) - getHeight(n->right);
```

```
}

// Right rotation function
struct Node* rightRotate(struct Node* y)
{
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height
        = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height
        = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}

// Left rotation function
struct Node* leftRotate(struct Node* x)
{
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height
        = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height
```

```
        = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}

// Function to insert a key into AVL tree
struct Node* insert(struct Node* node, int key)
{
    // 1. Perform standard BST insertion
    if (node == NULL)
        return createNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    // 2. Update height of this ancestor node
    node->height = 1
                + max(getHeight(node->left),
                      getHeight(node->right));

    // 3. Get the balance factor of this ancestor node to
    // check whether this node became unbalanced
    int balance = getBalanceFactor(node);

    // 4. If the node becomes unbalanced, then there are 4
    // cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
```

```
    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

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

// Function to perform preorder traversal of AVL tree
void inOrder(struct Node* root)
{
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->key);
        inOrder(root->right);
    }
}

// Main function
int main()
{
    struct Node* root = NULL;
```

```
    // Inserting nodes
    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 3);

    // Print preorder traversal of the AVL tree
    printf("Inorder traversal of AVL tree: ");
    inOrder(root);

    return 0;
}
```

Output

```
Inorder traversal of AVL tree: 1 2 3 4 5 6
```

Advantages of AVL Trees

- The height of the AVL tree is always balanced. The height never grows beyond log N, where N is the total number of nodes in the tree.
- It gives better search time complexity when compared to simple Binary Search trees.
- AVL trees have self-balancing capabilities.

Summary

- AVL trees are self-balancing binary search trees.
- Balance factor is the fundamental attribute of AVL trees
- The balance factor of a node is defined as the difference between the height of the left and right subtree of that node.
- The valid values of the balance factor are -1, 0, and +1.
- The insert and delete operation require rotations to be performed after violating the balance factor.

- The time complexity of insert, delete, and search operation is O(log N).
- AVL trees follow all properties of Binary Search Trees.
- The left subtree has nodes that are lesser than the root node. The right subtree has nodes that are always greater than the root node.
- AVL trees are used where search operation is more frequent compared to insert and delete operations.

B-TREE: Creation, Insertion, Deletion operations and Applications

Introduction of B-Tree:

- The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease.

- When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage.

- B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

- Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees.

- Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height.

- This shallow height leads to less disk I/O, which results in faster search and insertion operations.

- B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

- B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced.

- This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Time Complexity of B-Tree:

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Note: “n” is the total number of elements in the B-tree

Why use B-Tree:

Here, are reasons of using B-Tree

- Reduces the number of reads made on the disk
- B Trees can be easily optimized to adjust its size (that is the number of child nodes) according to the disk size
- It is a specially designed technique for handling a bulky amount of data.
- It is a useful algorithm for databases and file systems.
- A good choice to opt when it comes to reading and writing large blocks of data

History of B Tree:

- Data is stored on the disk in blocks, this data, when brought into main memory (or RAM) is called data structure.
- In-case of huge data, searching one record in the disk requires reading the entire disk; this increases time and main memory consumption due to high disk access frequency and data size.
- To overcome this, index tables are created that saves the record reference of the records based on the blocks they reside in. This drastically reduces the time and memory consumption.
- Since we have huge data, we can create multi-level index tables.
- Multi-level index can be designed by using B Tree for keeping the data sorted in a self-balancing fashion.

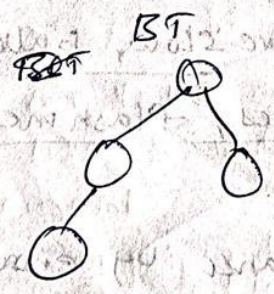
Properties of B-Tree:

* Balanced (m)-way tree → order.
* Generalization of BST in which a node can have more than one key & more than 2 children
* Maintains sorted order (Ascending)
* All leaf nodes must be at same level
* B-Tree of order m has following properties

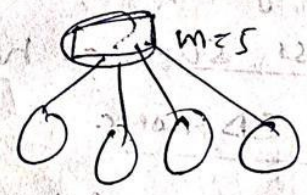
Children:
- → Every node has max m children
- → Min children: leaf → 0, root → 2, internal nodes $\lceil \frac{m}{2} \rceil$ ceil value.

Keys:
- ⇒ Every node has max of (m-1) keys
- Min keys: root node → 1
- All other nodes = $\lceil \frac{m}{2} \rceil - 1$

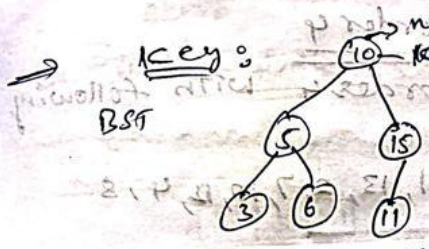
⇒



Atmost 2 children



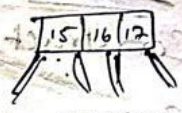
⇒ Key: node, key (value)



BST

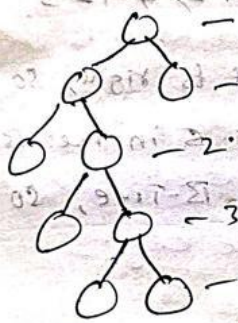
In BST every node have only one key.

B Tree



In B-Trees, node can have more than one key depends on the m value.

⇒ Leafs of same level - BT

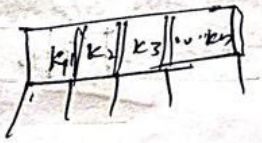


All leaf nodes of are of diff levels (BST)

* In B-tree all leaf nodes must be at same level.

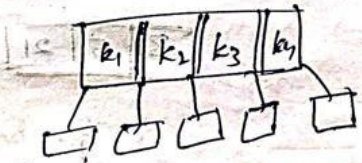
⇒ B-Tree of order m has

→ Every node has max m children



Min children: Leaf → 0
root → 2
Internal $\left(\frac{m}{2}\right)$

Children ⇒



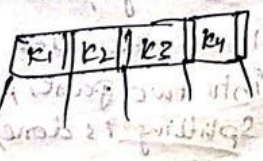
$\frac{5}{2} = 2.5 \Rightarrow 3$ (ceil)

⇒ Every node has max of (m-1) keys

⇒ BT = 5, (5-1) keys

⇒ 4 keys

Key



Min keys: root → 1

All other nodes = $\left\lceil \frac{m}{2} \right\rceil - 1$

Insertion in B-Tree of order 4

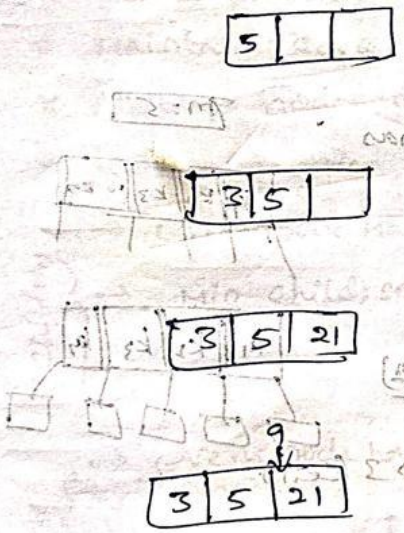
⇒ Construct a B-tree of order 4 with following set of data 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8.

Sol: $m = 4$, $\text{Keys} = m - 1 = 2$

⇒ B-Tree of order 4 ⇒ $m = 4$.

Max no of keys of any node $\Rightarrow m - 1 \Rightarrow 4 - 1 = 3$

⇒ Scan the data from left to right, so first number is 5, insert 5 in the B-Tree, Initially we have empty B-Tree, so one node will be created.



⇒ Next element in the list is 3, where we can't place it in the node, bcz 3 is less than 5. According to B-Tree property all the keys are placed in Asc order.

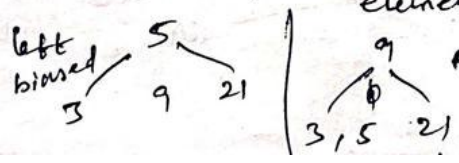
⇒ Next element is 21
21 > 3 & 21 & 5, place it after 5.

⇒ Next element is 9, First we place it in appropriate position.

⇒ Here the node is full because we can insert only 3 keys, where $m = 4$

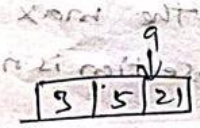
⇒ Here the tree is splitted into two parts; How this Splitting is done, means, the median or middle element is found and that middle element will go up. Now that middle will be parent

* Middle element
⇒ 3/2 ⇒ 1.5
We can take one or two as middle element
5 or 9 as middle element

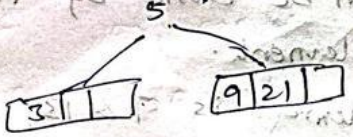


* This will happen when the order is even

above we can see the order is 4.



⇒ We go for (5), bcz it is left biased B-Tree.



⇒ The next no is one, we have find the proper position to insert in the B-tree.

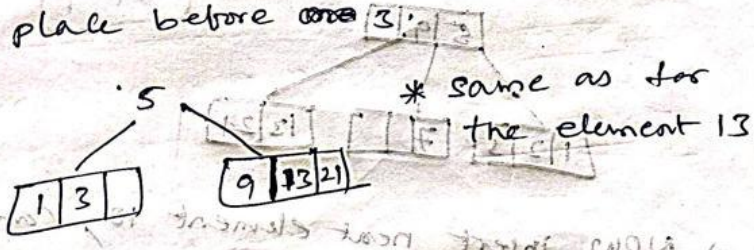
* compare 1 with root node and insert in the leaf node.

⇒ one rule is any new element will be insert in the leaf node.

1 < 5 ⇒ go left

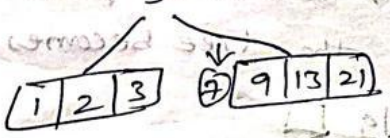
Now insert 1 after 3, but we can't insert after 3, because 1 is less than 3.

So place before 3



* same as for the element 13

⇒ Now (2) is compare with root 5 and move left and compare the keys in the left node (i.e) (1) & (3) and place it middle of keys (1) & (3)



⇒ Now the element (7), compare it with root 5 and move right side and find its appropriate place and then split it.

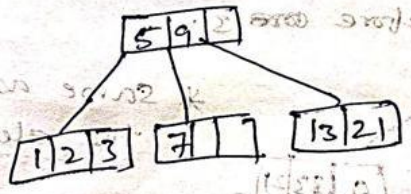
⇒ Now 7 can be place before key (9).
but the node reaches the max elements, so split the node. [Insertion is not possible]

⇒ The splitting can be done by taking the middle element.

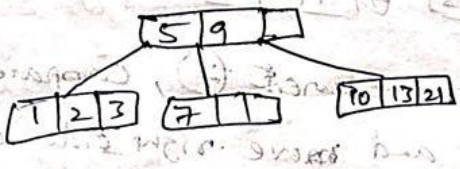
∴ The middle element is 9 & 13, but we go for 9 bcz of left biased B-Tree.

∴ So 9 will go up (parent) one level. Now place it after 5, that node contain only one key.

⇒ Now the tree becomes



⇒ Now insert next element 10, Compare with Root 5 & 9, 10 > 5 and 9, so more right side node. Now the right node already consists of keys 13 and 21. So, compare 10 with key 13 and 21 and it is less than 13 & 21. So place before 13 & 21. Now the tree becomes



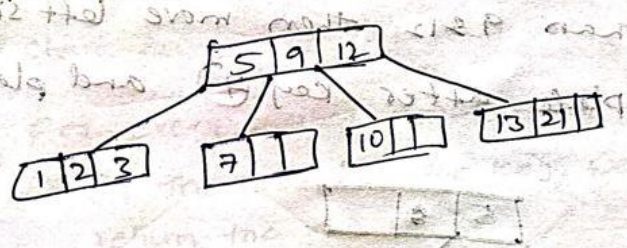
P.T.O

⇒ Now, the next element is 12, compare with root and move right side and compare with the keys 10, 13 and 21 and its appropriate position is between 10 and 13.

⇒ But the node is full (max is 3 keys only), so split the tree, before it find the middle element (i.e)

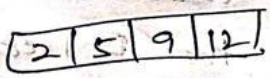
↑ middle element, go one level up.

Now the tree will be.



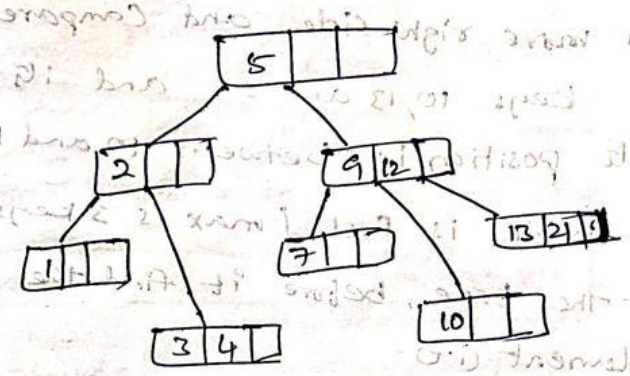
⇒ Now the element is 4, compare with root and move left side and place after (3). But the node is full. So splitting would be done, and the middle element is two, so (2) will go one level up (parent)

Now the tree is

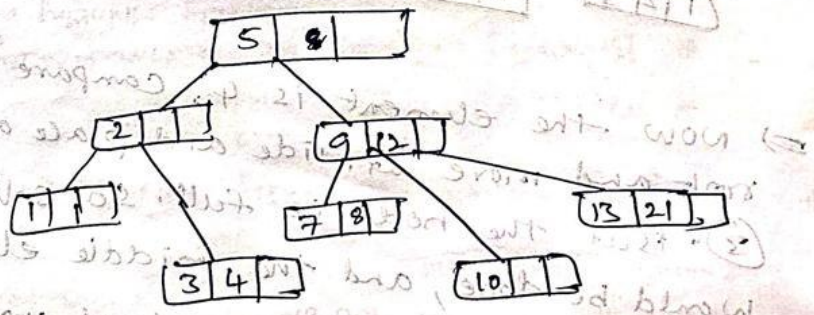


⇒ Here also the node is full, so split the node and the middle element is (5) and it will go one level up.

Now the tree becomes.



Now the last element is (8), Compare it with root, move right and Compare with 9 and 12, so the element (8) is less than 9 & 12, then move left side. and place after key (7) and place it



* Now the Btree is completed.

* We can also take the right biased and construct the B-Tree of order 4.

Traversal in B-Tree :

* Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children and keys. In the end, recursively print the rightmost child.

Search Operation in B-Tree :

* Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

⇒ 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 the leaf node and don't find k in the leaf node, then return NULL.

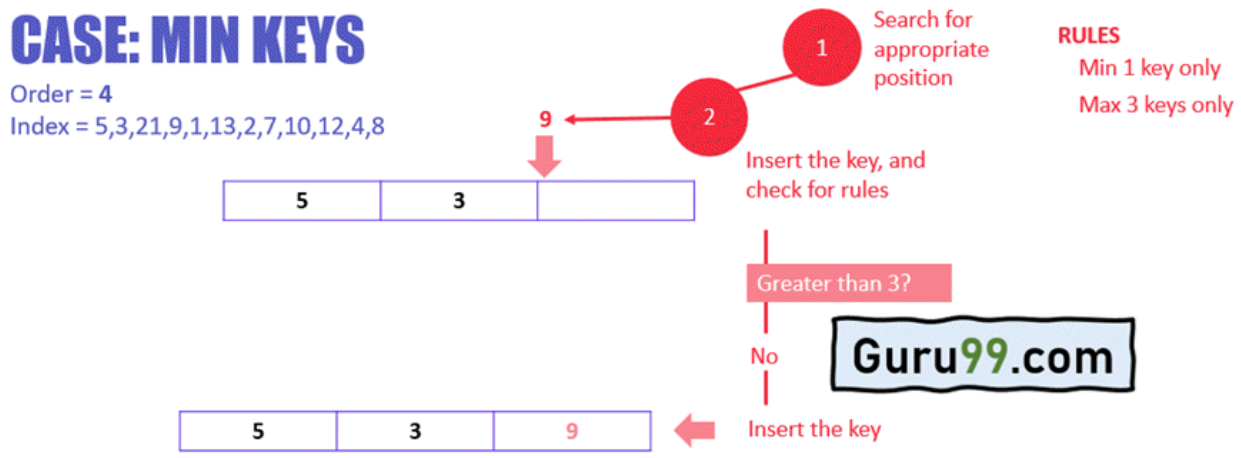
Insert Operation

Since B Tree is a self-balancing tree, you cannot force insert a key into just any node.

The following algorithm applies:

- Run the search operation and find the appropriate place of insertion.
- Insert the new key at the proper location, but if the node has a maximum number of keys already:
- The node, along with a newly inserted key, will split from the middle element.
- The middle element will become the parent for the other two child nodes.
- The nodes must re-arrange keys in ascending order.

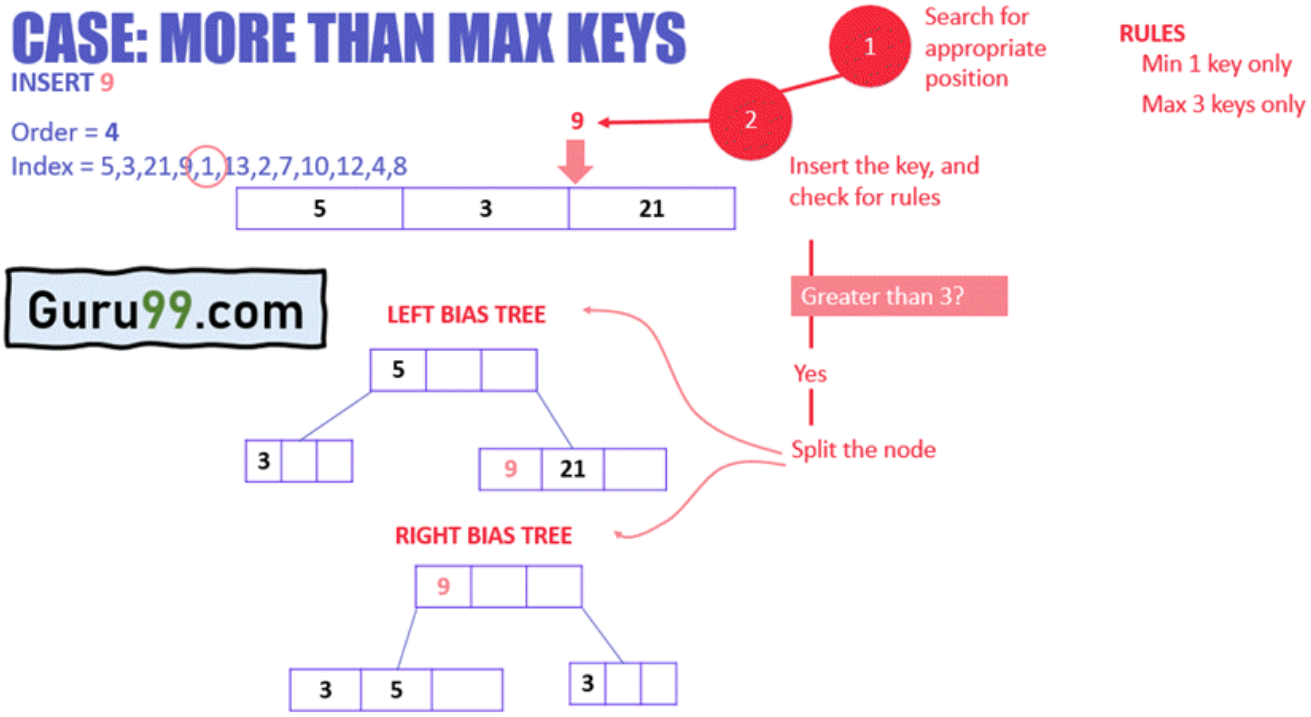
The following is not true about the insertion algorithm:
Since the node is full, therefore it will split, and then a new value will be inserted



In the above example:

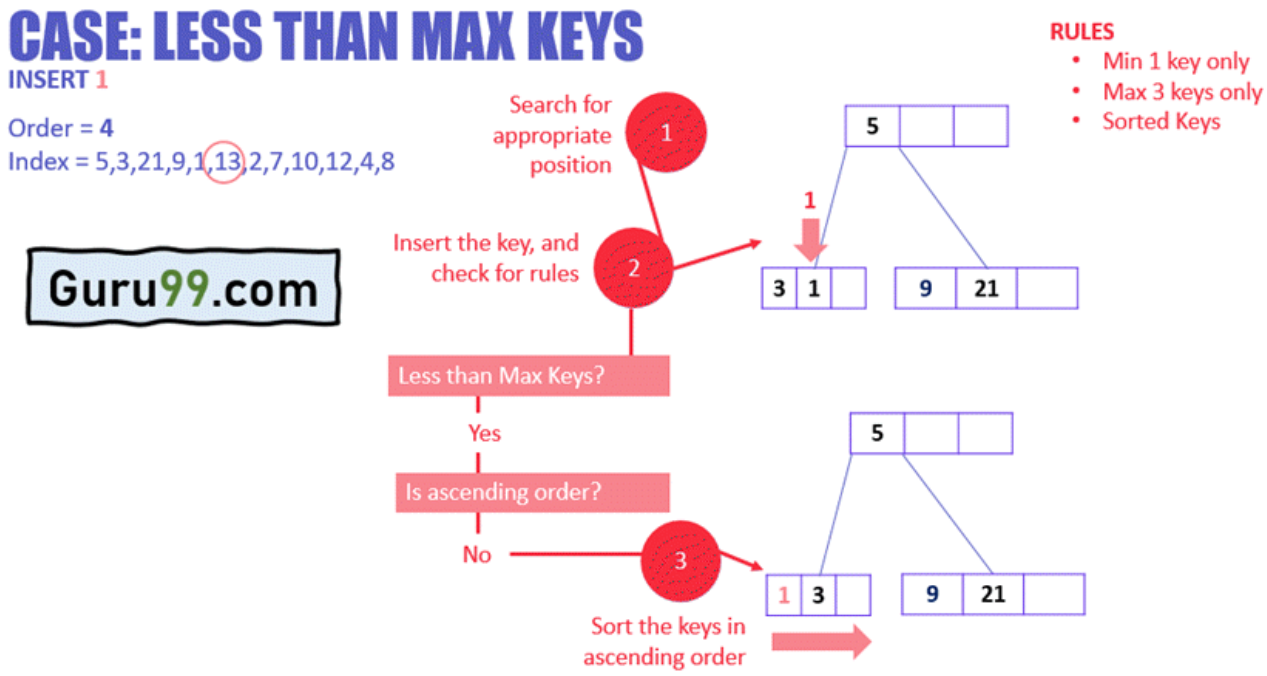
- Search the appropriate position in the node for the key
- Insert the key in the target node, and check for rules

- After insertion, does the node have more than equal to a minimum number of keys, which is 1? In this case, yes, it does. Check the next rule.
- After insertion, does the node have more than a maximum number of keys, which is 3? In this case, no, it does not. This means that the B Tree is not violating any rules, and the insertion is complete.



In the above example:

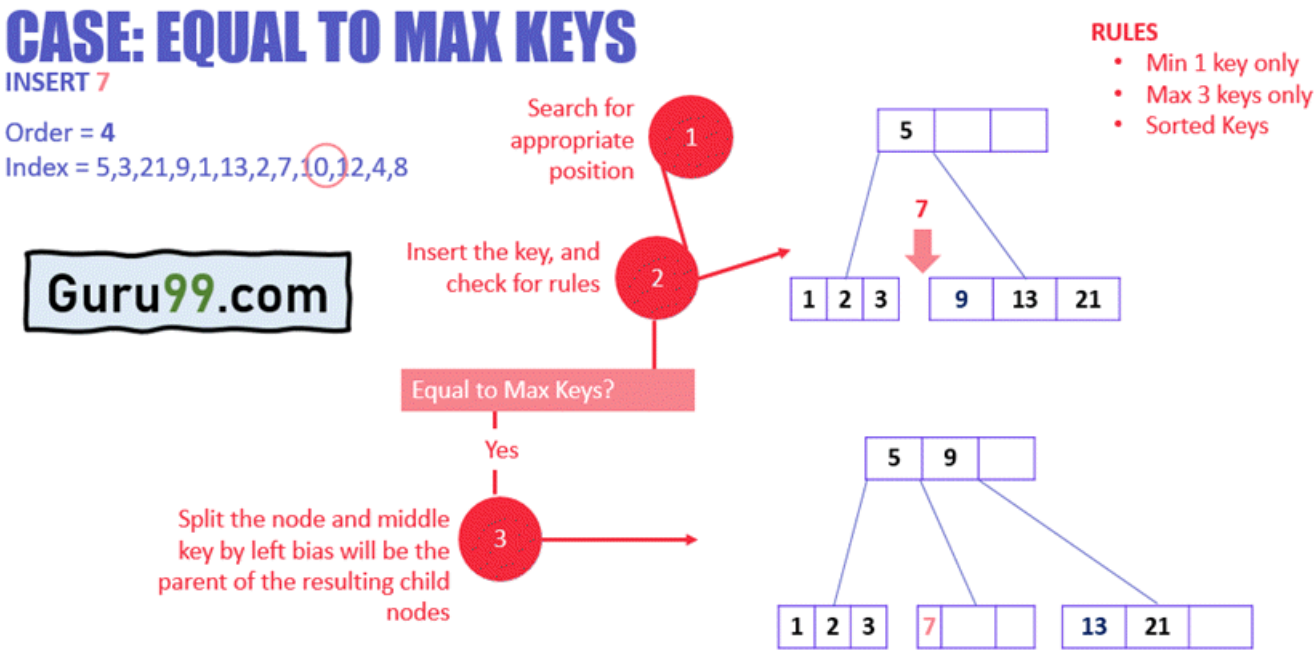
- The node has reached the max number of keys
- The node will split, and the middle key will become the root node of the rest two nodes.
- In case of even number of keys, the middle node will be selected by left bias or right bias.



In the above example:

- The node has less than max keys
- 1 is inserted next to 3, but the ascending order rule is violated
- In order to fix this, the keys are sorted

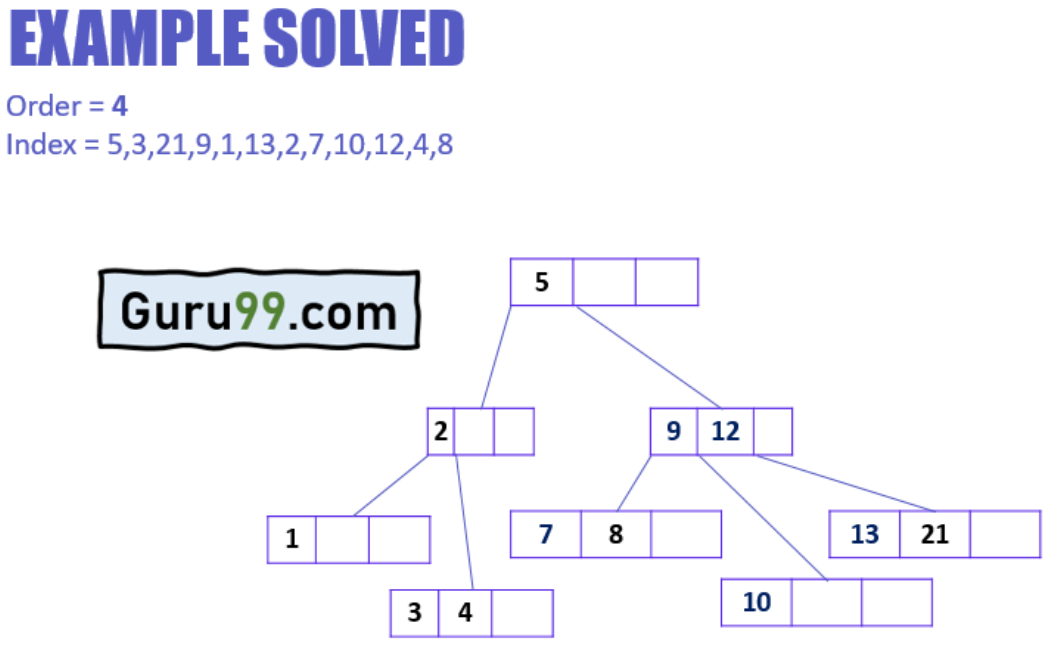
Similarly, 13 and 2 can be inserted easily in the node as they fulfill less than max keys rule for the nodes.



In the above example:

- The node has keys equal to max keys.
- The key is inserted to the target node, but it violates the rule of max keys.
- The target node is split, and the middle key by left bias is now the parent of the new child nodes.
- The new nodes are arranged in ascending order.

Similarly, based on the above rules and cases, the rest of the values can be inserted easily in the B Tree.



Delete Operation

The delete operation has more rules than insert and search operations.

The following algorithm applies:

- Run the search operation and find the target key in the nodes
- Three conditions applied based on the location of the target key, as explained in the following sections

If the target key is in the leaf node

- Target is in the leaf node, more than min keys.
  - Deleting this will not violate the property of B Tree
- Target is in leaf node, it has min key nodes
  - Deleting this will violate the property of B Tree
  - Target node can borrow key from immediate left node, or immediate right node (sibling)
  - The sibling will say **yes** if it has more than minimum number of keys

    - The key will be borrowed from the parent node, the max value will be transferred to a parent, the max value of the parent node will be transferred to the target node, and remove the target value
- Target is in the leaf node, but no siblings have more than min number of keys
    - Search for key
    - Merge with siblings and the minimum of parent nodes
    - Total keys will be now more than min
    - The target key will be replaced with the minimum of a parent node

If the target key is in an internal node

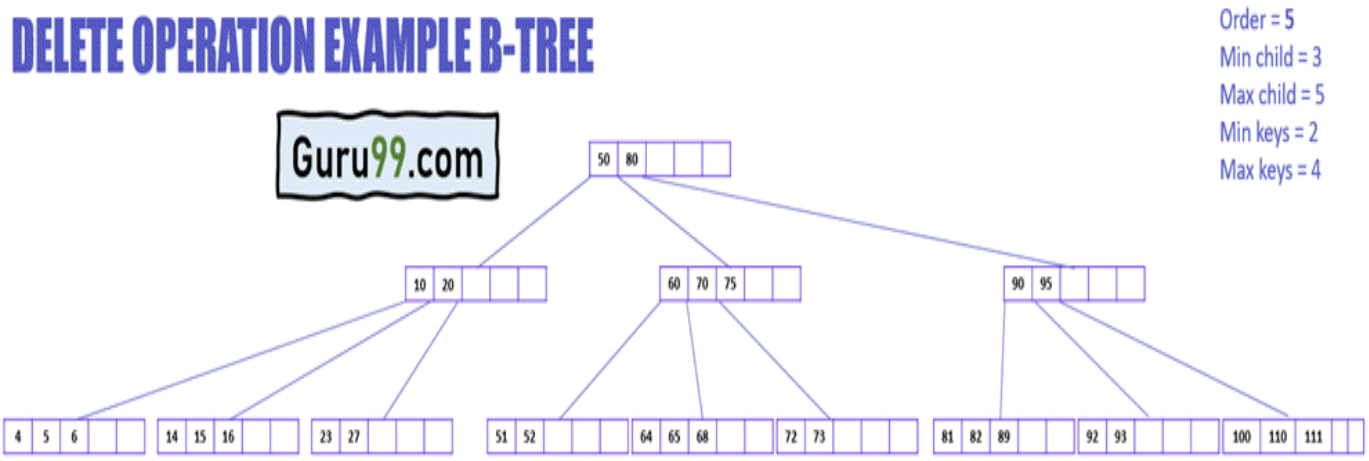
- Either choose, in- order predecessor or in-order successor
- In case the of in-order predecessor, the maximum key from its left subtree will be selected
- In case of in-order successor, the minimum key from its right subtree will be selected
- If the target key's in-order predecessor has more than the min keys, only then it can replace the target key with the max of the in-order predecessor
- If the target key's in-order predecessor does not have more than min keys, look for in-order successor's minimum key.
- If the target key's in-order predecessor and successor both have less than min keys, then merge the predecessor and successor.

If the target key is in a root node

- Replace with the maximum element of the in-order predecessor subtree
- If, after deletion, the target has less than min keys, then the target node will borrow max value from its sibling via sibling's parent.
- The max value of the parent will be taken by a target, but with the nodes of the max value of the sibling.

Now, let's understand the delete operation with an example.

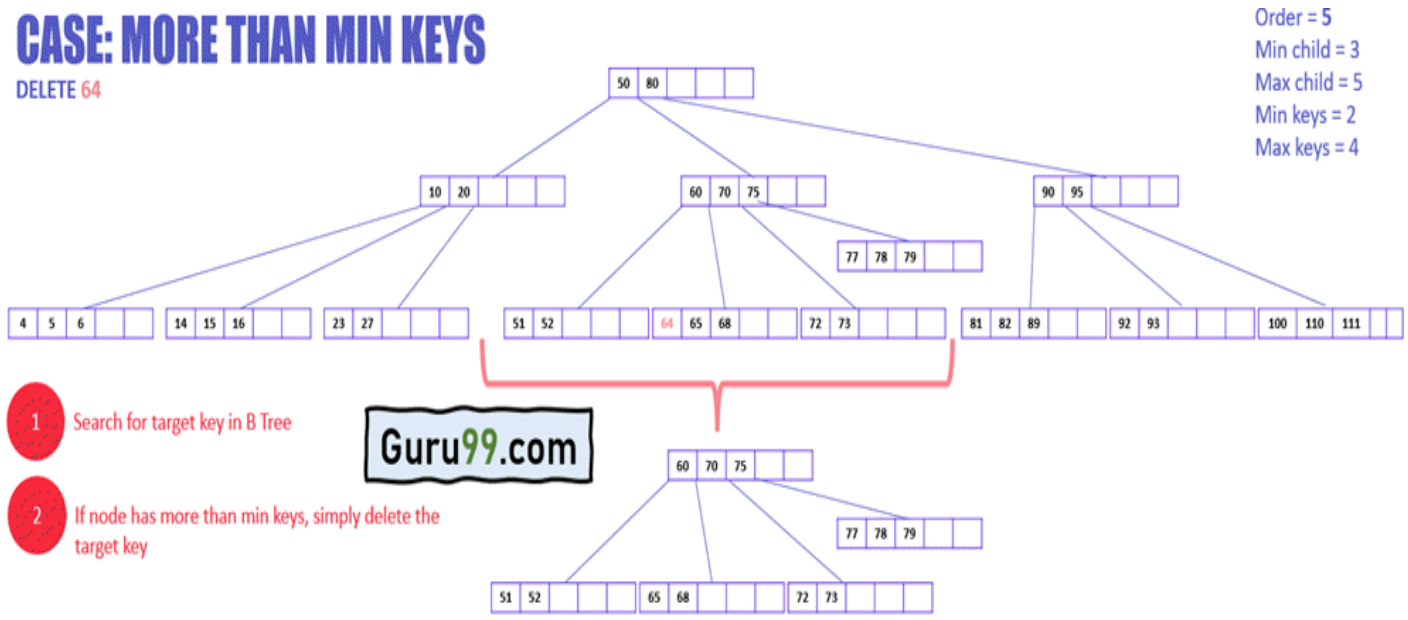
Now, let’s understand the delete operation with an example.



The above diagram displays different cases of delete operation in a B-Tree.

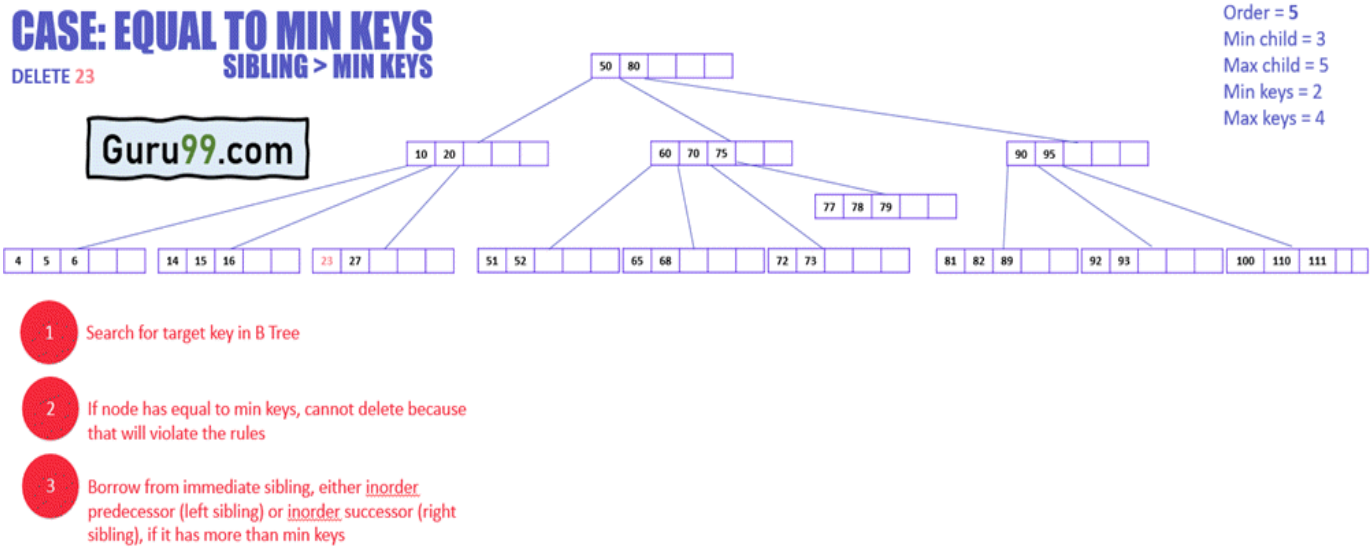
This B-Tree is of order 5, which means that the minimum number of child nodes any node can have is 3, and the maximum number of child nodes any node can have is 5.

Whereas the minimum and a maximum number of keys any node can have are 2 and 4, respectively.



In the above example:

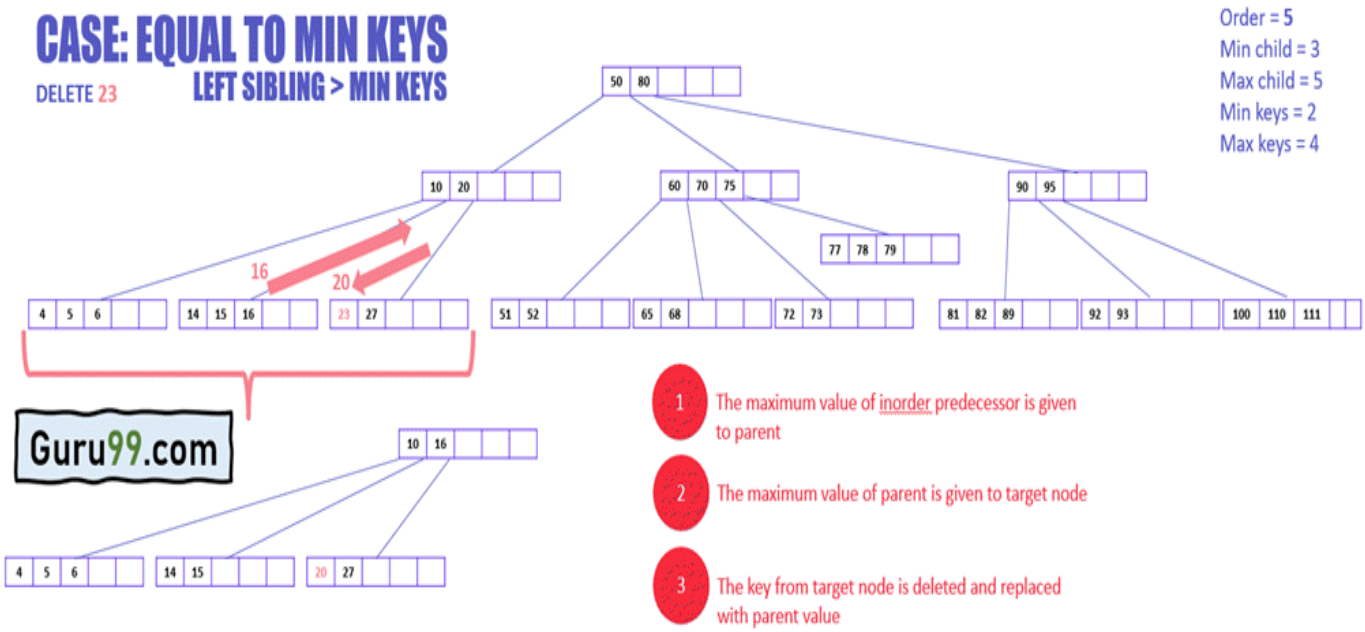
- The target node has the target key to delete
- The target node has keys more than minimum keys
- Simply delete the key



In the above example:

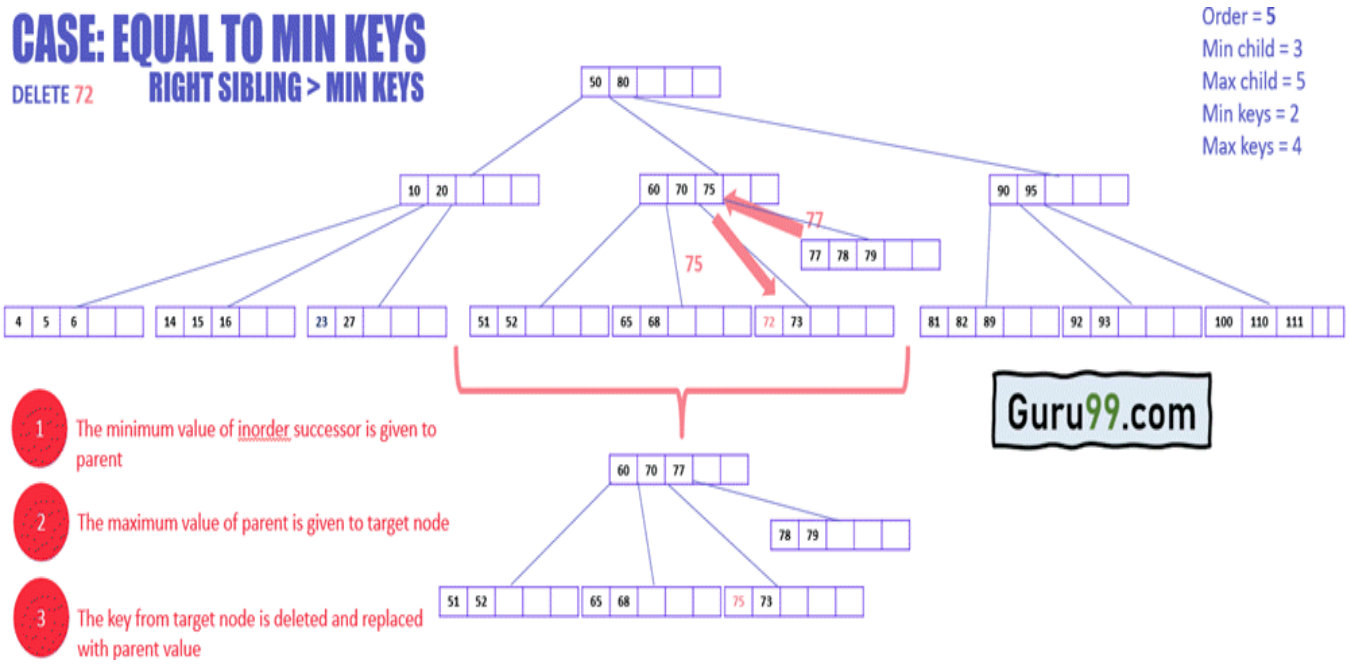
- The target node has keys equal to minimum keys, so cannot delete it directly as it will violate the conditions

Now, the following diagram explains how to delete this key:



- The target node will borrow a key from immediate sibling, in this case, in-order predecessor (left sibling) because it does not have any in-order successor (right sibling)
- The maximum value of the inorder predecessor will be transferred to the parent, and the parent will transfer the maximum value to the target node (see the diagram below)

The following example illustrates how to delete a key that needs a value from its in-order successor.



- The target node will borrow a key from immediate sibling, in this case, in-order successor (right sibling) because it's in-order predecessor (left sibling) has keys equal to minimum keys.
- The minimum value of the in-order successor will be transferred to the parent, and the parent will transfer the maximum value to the target node.

In the example below, the target node does not have any sibling that can give its key to the target node. Therefore, merging is required.

Applications of B-Trees:

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.

- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

Advantages of B-Trees:

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storge utilization.

Disadvantages of B-Trees:

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- Slow in comparison to other data structures.