Introduction to Algorithm Analysis

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START

Step 2 − declare three integers **a**, **b** & **c**

Step 3 − define values of **a** & **b**

Step 4 − add values of **a** & **b**

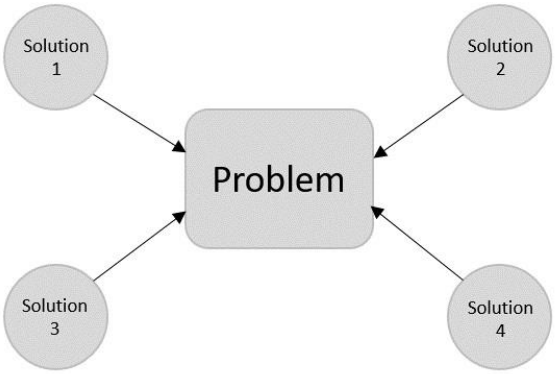
Step 5 − store output of step 4 to **c**

Step 6 − print **c**

Step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD
Step 2 − get values of **a** & **b**
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- ***A Priori* Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior* Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.
- Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −
- Algorithm: SUM(A, B)
- **Step 1** − START
- **Step 2** − C ← A + B + 10
- **Step 3** − Stop
- Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as $g(n^2)$. 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 significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by **T(n)**, where **n** is the input size.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** − Big Oh Notation
- **Ω** − Big omega Notation
- **θ** − Big theta Notation
- **o** − Little Oh Notation
- **ω** − Little omega Notation

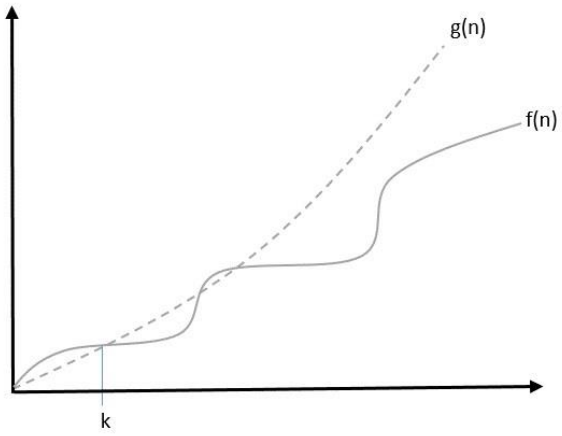
Big Oh, O: Asymptotic Upper Bound

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. is the most commonly used notation. It measures the **worst case time complexity** or the longest amount of time an algorithm can possibly take to complete.

A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that −

f(n)⩽c.g(n)f(n)⩽c.g(n) for n>n0n>n0 in all case

Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.



Example

Let us consider a given function, f(n)=4.n3+10.n2+5.n+1f(n)=4.n3+10.n2+5.n+1

Considering g(n)=n3g(n)=n3,

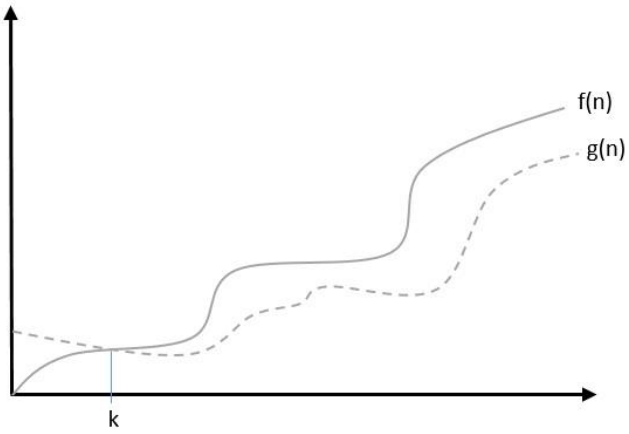
f(n)⩽5.g(n)f(n)⩽5.g(n) for all the values of n>2n>2

Hence, the complexity of ***f(n)*** can be represented as O(g(n))O(g(n)), i.e. O(n3)

Big Omega, Ω: Asymptotic Lower Bound

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the **best case time complexity** or the best amount of time an algorithm can possibly take to complete.

We say that f(n)=Ω(g(n))f(n)=Ω(g(n)) when there exists
constant **c** that f(n)⩾c.g(n)f(n)⩾c.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f*** ; after a certain value of ***n, f*** will never go below ***g***.



Example

Let us consider a given function, f(n)=4.n3+10.n2+5.n+1f(n)=4.n3+10.n2+5.n+1.

Considering g(n)=n3g(n)=n3, f(n)⩾4.g(n)f(n)⩾4.g(n) for all the values of n>0n>0.

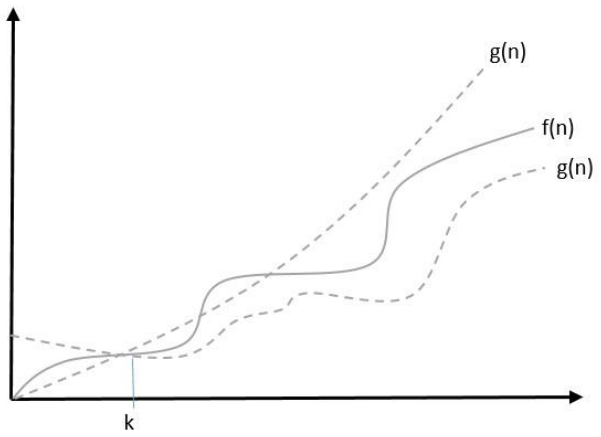
Hence, the complexity of ***f(n)*** can be represented as Ω(g(n))Ω(g(n)), i.e. Ω(n3)

Theta, θ: Asymptotic Tight Bound

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. Some may confuse the theta notation as the average case time complexity; while big theta notation could be *almost* accurately used to describe the average case, other notations could be used as well.

We say that f(n)=θ(g(n))f(n)=θ(g(n)) when there exist
constants $\mathbf{c_1}$ and $\mathbf{c_2}$ that c1.g(n)⩽f(n)⩽c2.g(n)c1.g(n)⩽f(n)⩽c2.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer.

This means function **g** is a tight bound for function **f**.



Example

Let us consider a given function, f(n)=4.n3+10.n2+5.n+1f(n)=4.n3+10.n2+5.n+1

Considering g(n)=n3g(n)=n3, 4.g(n)⩽f(n)⩽5.g(n)4.g(n)⩽f(n)⩽5.g(n) for all the large values of **n**.

Hence, the complexity of ***f(n)*** can be represented as θ(g(n))θ(g(n)), i.e. θ(n3)θ(n3).

Little Oh, o

The asymptotic upper bound provided by **O-notation** may or may not be asymptotically tight. The bound 2.n2=O(n2)2.n2=O(n2) is asymptotically tight, but the bound 2.n=O(n2)2.n=O(n2) is not.

We use **o-notation** to denote an upper bound that is not asymptotically tight.

We formally define ***o(g(n))*** (little-oh of g of n) as the set ***f(n) = o(g(n))*** for any positive constant c>0c>0 and there exists a value n0>0n0>0, such that 0⩽f(n)⩽c.g(n)0⩽f(n)⩽c.g(n).

Intuitively, in the **o-notation**, the function ***f(n)*** becomes insignificant relative to ***g(n)*** as **n** approaches infinity; that is,

$$\lim n \to \infty (f(n)g(n)) = 0 \lim n \to \infty (f(n)g(n)) = 0$$

Example

Let us consider the same function, f(n)=4.n3+10.n2+5.n+1f(n)=4.n3+10.n2+5.n+1

Considering g(n)=n4g(n)=n4,

$$\lim n \to \infty (4.n3+10.n2+5.n+1n4) = 0 \lim n \to \infty (4.n3+10.n2+5.n+1n4) = 0$$

Hence, the complexity of ***f(n)*** can be represented as o(g(n))o(g(n)), i.e. o(n4)o(n4).

Little Omega, ω

We use **ω-notation** to denote a lower bound that is not asymptotically tight. Formally, however, we define ***ω(g(n))*** (little-omega of g of n) as the set ***f(n) = ω(g(n))*** for any positive constant ***C > 0*** and there exists a value n0>0n0>0, such that 0⩽c.g(n)<f(n)0⩽c.g(n)<f(n).

For example, n22=ω(n)n22=ω(n), but n22≠ω(n2)n22≠ω(n2). The relation f(n)=ω(g(n))f(n)=ω(g(n)) implies that the following limit exists

$$\lim n \to \infty (f(n)g(n)) = \infty \lim n \to \infty (f(n)g(n)) = \infty$$

That is, ***f(n)*** becomes arbitrarily large relative to ***g(n)*** as **n** approaches infinity.

Example

Let us consider same function, f(n)=4.n3+10.n2+5.n+1f(n)=4.n3+10.n2+5.n+1

Considering g(n)=n2g(n)=n2,

$$\lim n \to \infty (4.n3+10.n2+5.n+1n2) = \infty \lim n \to \infty (4.n3+10.n2+5.n+1n2) = \infty$$

Hence, the complexity of ***f(n)*** can be represented as o(g(n))o(g(n)), i.e. ω(n2)ω(n2).

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

Constant	−	O(1)
Logarithmic	−	O(log n)
Linear	−	O(n)
n log n	−	O(n log n)
Quadratic	−	$O(n^2)$
Cubic	−	$O(n^3)$
Polynomial	−	$n^{O(1)}$
Exponential	−	$2^{O(n)}$

AVL TREES

The first type of self-balancing binary search tree to be invented is the AVL tree. The name AVL tree is coined after its inventor's names − Adelson-Velsky and Landis.

In AVL trees, the difference between the heights of left and right subtrees, known as the **Balance Factor**, must be at most one. Once the difference exceeds one, the tree automatically executes the balancing algorithm until the difference becomes one again.

```
BALANCE FACTOR =
HEIGHT(LEFT SUBTREE) − HEIGHT(RIGHT SUBTREE)
```

There are usually four cases of rotation in the balancing algorithm of AVL trees: LL, RR, LR, RL.

LL Rotations

LL rotation is performed when the node is inserted into the right subtree leading to an unbalanced tree. This is a single left rotation to make the tree balanced again −

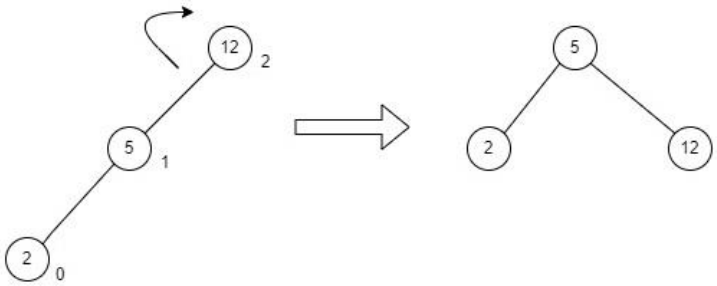


Fig : LL Rotation

The node where the unbalance occurs becomes the left child and the newly added node becomes the right child with the middle node as the parent node.

RR Rotations

RR rotation is performed when the node is inserted into the left subtree leading to an unbalanced tree. This is a single right rotation to make the tree balanced again −

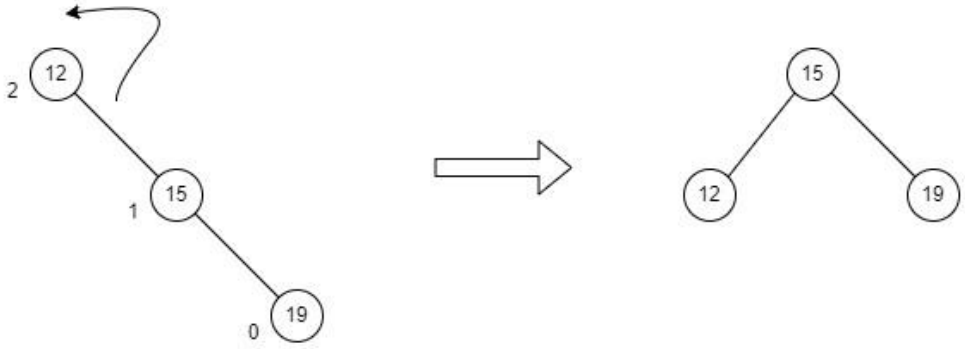


Fig : RR Rotation

The node where the unbalance occurs becomes the right child and the newly added node becomes the left child with the middle node as the parent node.

LR Rotations

LR rotation is the extended version of the previous single rotations, also called a double rotation. It is performed when a node is inserted into the right subtree of the left subtree. The LR rotation is a combination of the left rotation followed by the right rotation. There are multiple steps to be followed to carry this out.

- Consider an example with “A” as the root node, “B” as the left child of “A” and “C” as the right child of “B”.

- Since the unbalance occurs at A, a left rotation is applied on the child nodes of A, i.e. B and C.
- After the rotation, the C node becomes the left child of A and B becomes the left child of C.
- The unbalance still persists, therefore a right rotation is applied at the root node A and the left child C.
- After the final right rotation, C becomes the root node, A becomes the right child and B is the left child.

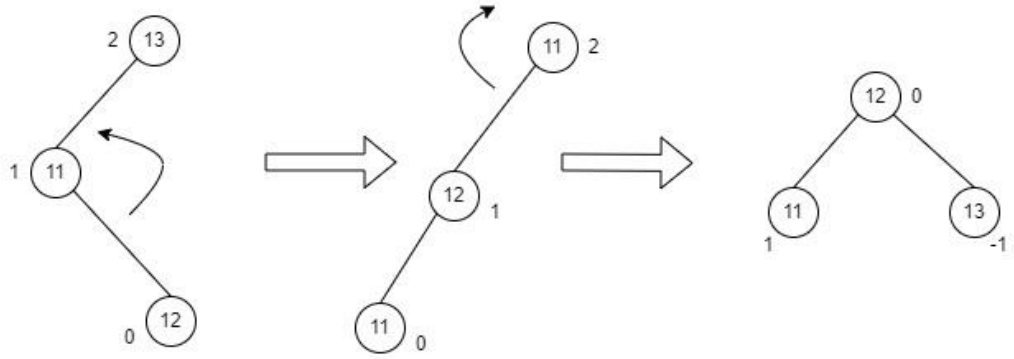


Fig : LR Rotation

RL Rotations

RL rotation is also the extended version of the previous single rotations, hence it is called a double rotation and it is performed if a node is inserted into the left subtree of the right subtree. The RL rotation is a combination of the right rotation followed by the left rotation. There are multiple steps to be followed to carry this out.

- Consider an example with “A” as the root node, “B” as the right child of “A” and “C” as the left child of “B”.
- Since the unbalance occurs at A, a right rotation is applied on the child nodes of A, i.e. B and C.
- After the rotation, the C node becomes the right child of A and B becomes the right child of C.
- The unbalance still persists, therefore a left rotation is applied at the root node A and the right child C.
- After the final left rotation, C becomes the root node, A becomes the left child and B is the right child.

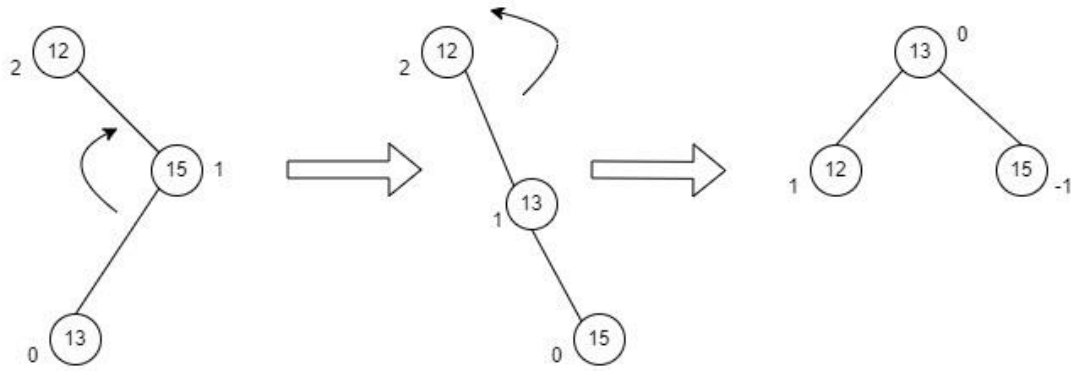


Fig : RL Rotation

Basic Operations of AVL Trees

The basic operations performed on the AVL Tree structures include all the operations performed on a binary search tree, since the AVL Tree at its core is actually just a binary search tree holding all its properties. Therefore, basic operations performed on an AVL Tree are − **Insertion** and **Deletion**.

Insertion operation

The data is inserted into the AVL Tree by following the Binary Search Tree property of insertion, i.e. the left subtree must contain elements less than the root value and right subtree must contain all the greater elements.

However, in AVL Trees, after the insertion of each element, the balance factor of the tree is checked; if it does not exceed 1, the tree is left as it is. But if the balance factor exceeds 1, a balancing algorithm is applied to readjust the tree such that balance factor becomes less than or equal to 1 again.

Algorithm

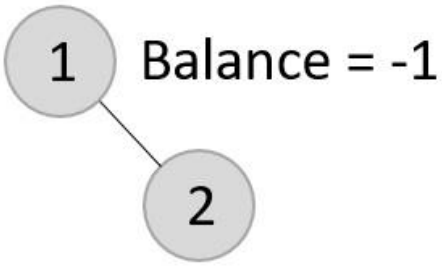
The following steps are involved in performing the insertion operation of an AVL Tree −

Step 1 − Create a node
Step 2 − Check if the tree is empty
Step 3 − If the tree is empty, the new node created will become the root node of the AVL Tree.
Step 4 − If the tree is not empty, we perform the Binary Search Tree insertion operation and check the balancing factor of the node in the tree.
Step 5 − Suppose the balancing factor exceeds ±1, we apply suitable rotations on the said node and resume the insertion from Step 4.

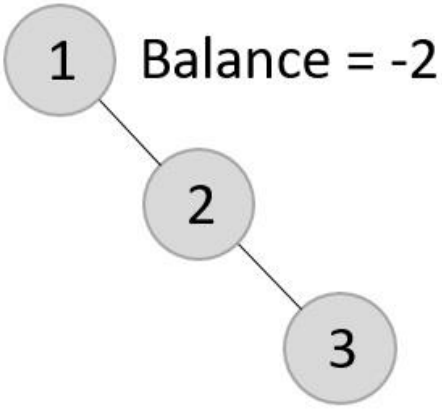
Let us understand the insertion operation by constructing an example AVL tree with 1 to 7 integers.

Starting with the first element 1, we create a node and measure the balance, i.e., 0.

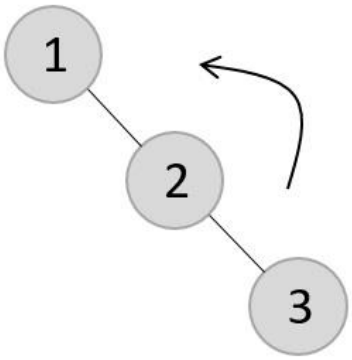
Since both the binary search property and the balance factor are satisfied, we insert another element into the tree.



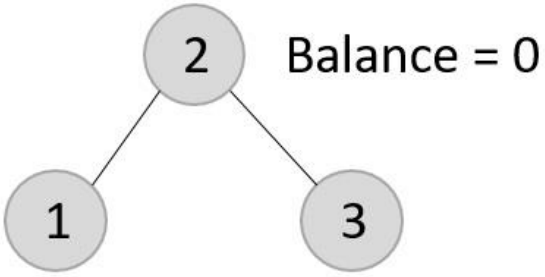
The balance factor for the two nodes are calculated and is found to be -1 (Height of left subtree is 0 and height of the right subtree is 1). Since it does not exceed 1, we add another element to the tree.



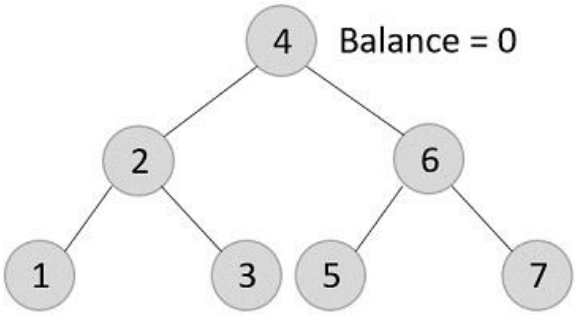
Now, after adding the third element, the balance factor exceeds 1 and becomes 2. Therefore, rotations are applied. In this case, the RR rotation is applied since the imbalance occurs at two right nodes.



The tree is rearranged as −



Similarly, the next elements are inserted and rearranged using these rotations. After rearrangement, we achieve the tree as −




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

```
#include <stdlib.h>
struct Node {
  int data;
  struct Node *leftChild;
  struct Node *rightChild;
  int height;
};
int max(int a, int b);
int height(struct Node *N){
  if (N == NULL)
    return 0;
  return N->height;
}
int max(int a, int b){
  return (a > b) ? a : b;
}
struct Node *newNode(int data){
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = data;
  node->leftChild = NULL;
  node->rightChild = NULL;
  node->height = 1;
  return (node);
}
struct Node *rightRotate(struct Node *y){
  struct Node *x = y->leftChild;
  struct Node *T2 = x->rightChild;
  x->rightChild = y;
  y->leftChild = T2;
```

```
  y->height = max(height(y->leftChild), height(y->rightChild)) + 1;
  x->height = max(height(x->leftChild), height(x->rightChild)) + 1;
  return x;
}
struct Node *leftRotate(struct Node *x){
  struct Node *y = x->rightChild;
  struct Node *T2 = y->leftChild;
  y->leftChild = x;
  x->rightChild = T2;
  x->height = max(height(x->leftChild), height(x->rightChild)) + 1;
  y->height = max(height(y->leftChild), height(y->rightChild)) + 1;
  return y;
}
int getBalance(struct Node *N){
  if (N == NULL)
    return 0;
  return height(N->leftChild) - height(N->rightChild);
}
struct Node *insertNode(struct Node *node, int data){
  if (node == NULL)
    return (newNode(data));
  if (data < node->data)
    node->leftChild = insertNode(node->leftChild, data);
  else if (data > node->data)
    node->rightChild = insertNode(node->rightChild, data);
  else
    return node;
  node->height = 1 + max(height(node->leftChild),
                         height(node->rightChild));
```

```
  int balance = getBalance(node);
  if (balance > 1 && data < node->leftChild->data)
    return rightRotate(node);
  if (balance < -1 && data > node->rightChild->data)
    return leftRotate(node);
  if (balance > 1 && data > node->leftChild->data) {
    node->leftChild = leftRotate(node->leftChild);
    return rightRotate(node);
  }
  if (balance < -1 && data < node->rightChild->data) {
    node->rightChild = rightRotate(node->rightChild);
    return leftRotate(node);
  }
  return node;
}
struct Node *minValueNode(struct Node *node){
  struct Node *current = node;
  while (current->leftChild != NULL)
    current = current->leftChild;
  return current;
}
void printTree(struct Node *root){
  if (root == NULL)
    return;
  if (root != NULL) {
    printTree(root->leftChild);
    printf("%d ", root->data);
    printTree(root->rightChild);
  }
```

```
}
int main(){
  struct Node *root = NULL;
  root = insertNode(root, 22);
  root = insertNode(root, 14);
  root = insertNode(root, 72);
  root = insertNode(root, 44);
  root = insertNode(root, 25);
  root = insertNode(root, 63);
  root = insertNode(root, 98);
  printf("AVL Tree: ");
  printTree(root);
  return 0;
}
```

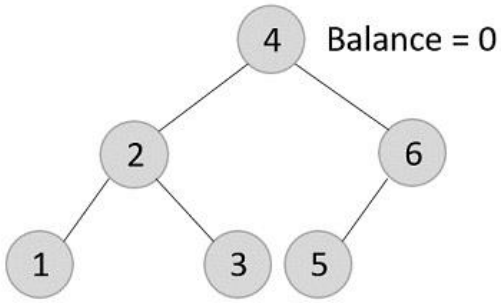
Output

AVL Tree: 14 22 25 44 63 72 98

Deletion operation

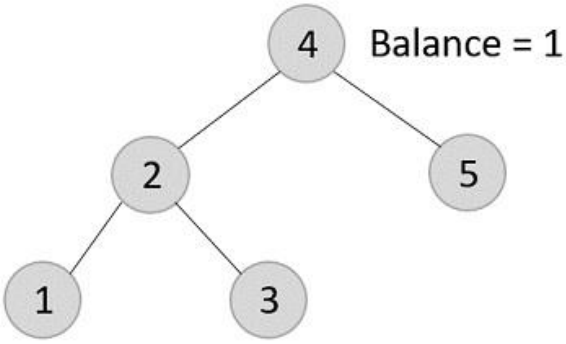
Deletion in the AVL Trees take place in three different scenarios −

- **Scenario 1 (Deletion of a leaf node)** − If the node to be deleted is a leaf node, then it is deleted without any replacement as it does not disturb the binary search tree property. However, the balance factor may get disturbed, so rotations are applied to restore it.
- **Scenario 2 (Deletion of a node with one child)** − If the node to be deleted has one child, replace the value in that node with the value in its child node. Then delete the child node. If the balance factor is disturbed, rotations are applied.
- **Scenario 3 (Deletion of a node with two child nodes)** − If the node to be deleted has two child nodes, find the inorder successor of that node and replace its value with the inorder successor value. Then try to delete the inorder successor node. If the balance factor exceeds 1 after deletion, apply balance algorithms.



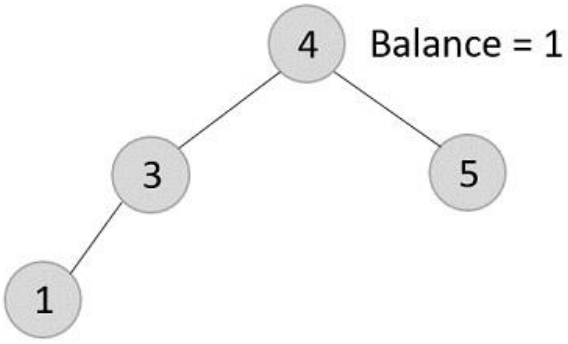
- Deleting element 7 from the tree above −

Since the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree

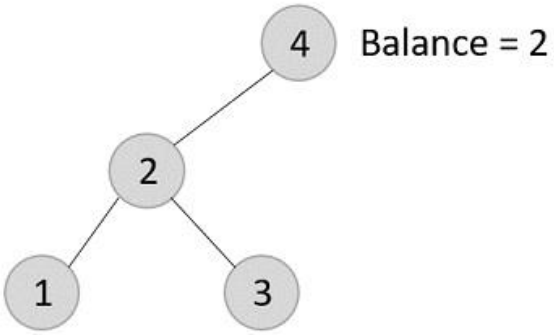


- Deleting element 6 from the output tree achieved −

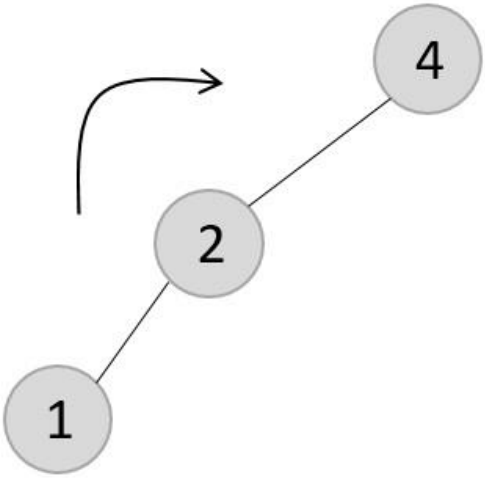
However, element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node: node 5.



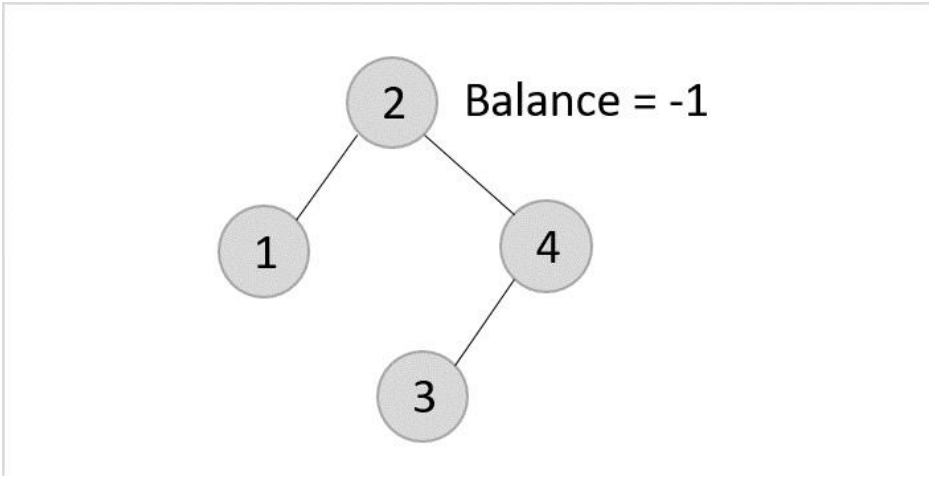
The balance of the tree becomes 1, and since it does not exceed 1 the tree is left as it is. If we delete the element 5 further, we would have to apply the left rotations; either LL or LR since the imbalance occurs at both 1-2-4 and 3-2-4.



The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here).

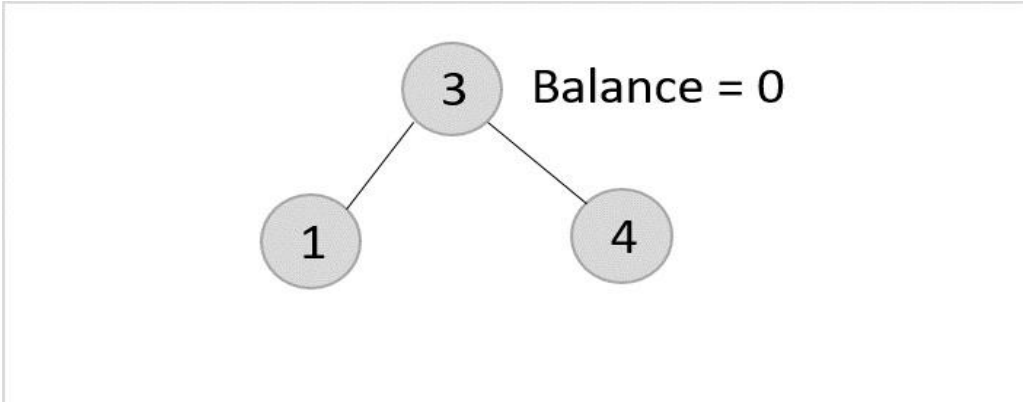


Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4.



- Deleting element 2 from the remaining tree −

As mentioned in scenario 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor.



The balance of the tree still remains 1, therefore we leave the tree as it is without performing any rotations.

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

  int data;

  struct Node *leftChild;

  struct Node *rightChild;

  int height;

};
```

```
int max(int a, int b);

int height(struct Node *N){
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b){
  return (a > b) ? a : b;
}

struct Node *newNode(int data){
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = data;
  node->leftChild = NULL;
  node->rightChild = NULL;
  node->height = 1;
  return (node);
}

struct Node *rightRotate(struct Node *y){
  struct Node *x = y->leftChild;
  struct Node *T2 = x->rightChild;
  x->rightChild = y;
  y->leftChild = T2;
  y->height = max(height(y->leftChild), height(y->rightChild)) + 1;
  x->height = max(height(x->leftChild), height(x->rightChild)) + 1;
```

```
  return x;
}
struct Node *leftRotate(struct Node *x){
  struct Node *y = x->rightChild;
  struct Node *T2 = y->leftChild;
  y->leftChild = x;
  x->rightChild = T2;
  x->height = max(height(x->leftChild), height(x->rightChild)) + 1;
  y->height = max(height(y->leftChild), height(y->rightChild)) + 1;
  return y;
}
int getBalance(struct Node *N){
  if (N == NULL)
    return 0;
  return height(N->leftChild) - height(N->rightChild);
}
struct Node *insertNode(struct Node *node, int data){
  if (node == NULL)
    return (newNode(data));
  if (data < node->data)
    node->leftChild = insertNode(node->leftChild, data);
  else if (data > node->data)
    node->rightChild = insertNode(node->rightChild, data);
  else
```

```
    return node;
  node->height = 1 + max(height(node->leftChild),
                height(node->rightChild));
  int balance = getBalance(node);
  if (balance > 1 && data < node->leftChild->data)
    return rightRotate(node);
  if (balance < -1 && data > node->rightChild->data)
    return leftRotate(node);
  if (balance > 1 && data > node->leftChild->data) {
    node->leftChild = leftRotate(node->leftChild);
    return rightRotate(node);
  }
  if (balance < -1 && data < node->rightChild->data) {
    node->rightChild = rightRotate(node->rightChild);
    return leftRotate(node);
  }
  return node;
}
struct Node *minValueNode(struct Node *node){
  struct Node *current = node;
  while (current->leftChild != NULL)
    current = current->leftChild;
  return current;
}
```

```
struct Node *deleteNode(struct Node *root, int data){
  if (root == NULL)
    return root;
  if (data < root->data)
    root->leftChild = deleteNode(root->leftChild, data);
  else if (data > root->data)
    root->rightChild = deleteNode(root->rightChild, data);
  else {
    if ((root->leftChild == NULL) || (root->rightChild == NULL)) {
      struct Node *temp = root->leftChild ? root->leftChild : root->rightChild;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      struct Node *temp = minValueNode(root->rightChild);
      root->data = temp->data;
      root->rightChild = deleteNode(root->rightChild, temp->data);
    }
  }
  if (root == NULL)
    return root;
```

```
  root->height = 1 + max(height(root->leftChild),
                         height(root->rightChild));
  int balance = getBalance(root);
  if (balance > 1 && getBalance(root->leftChild) >= 0)
    return rightRotate(root);
  if (balance > 1 && getBalance(root->leftChild) < 0) {
    root->leftChild = leftRotate(root->leftChild);
    return rightRotate(root);
  }
  if (balance < -1 && getBalance(root->rightChild) <= 0)
    return leftRotate(root);
  if (balance < -1 && getBalance(root->rightChild) > 0) {
    root->rightChild = rightRotate(root->rightChild);
    return leftRotate(root);
  }
  return root;
}

// Print the tree
void printTree(struct Node *root){
  if (root != NULL) {
    printTree(root->leftChild);
    printf("%d ", root->data);
    printTree(root->rightChild);
```

```
  }
}
int main(){
  struct Node *root = NULL;
  root = insertNode(root, 22);
  root = insertNode(root, 14);
  root = insertNode(root, 72);
  root = insertNode(root, 44);
  root = insertNode(root, 25);
  root = insertNode(root, 63);
  root = insertNode(root, 98);
  printf("AVL Tree: ");
  printTree(root);
  root = deleteNode(root, 25);
  printf("\nAfter deletion: ");
  printTree(root);
  return 0;
}
```

Output

AVL Tree: 14 22 25 44 63 72 98

After deletion: 14 22 44 63 72 98

B Trees

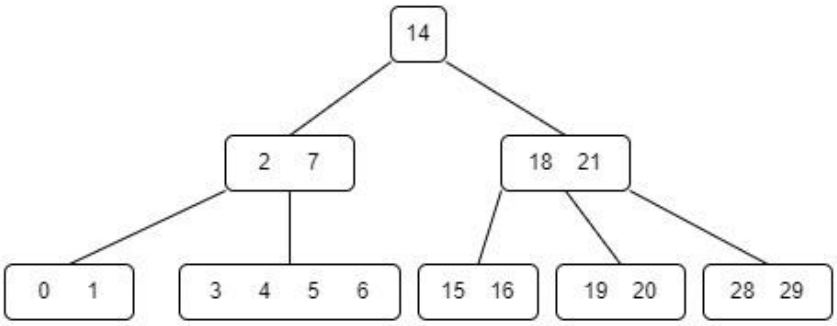
B trees are extended binary search trees that are specialized in m-way searching, since the order of B trees is 'm'. Order of a tree is defined as the maximum number of children a node

can accommodate. Therefore, the height of a b tree is relatively smaller than the height of AVL tree and RB tree.

They are general form of a Binary Search Tree as it holds more than one key and two children.

The various properties of B trees include −

- Every node in a B Tree will hold a maximum of m children and (m-1) keys, since the order of the tree is m.
- Every node in a B tree, except root and leaf, can hold at least m/2 children
- The root node must have no less than two children.
- All the paths in a B tree must end at the same level, i.e. the leaf nodes must be at the same level.
- A B tree always maintains sorted data.



B trees are also widely used in disk access, minimizing the disk access time since the height of a b tree is low.

Note − A disk access is the memory access to the computer disk where the information is stored and disk access time is the time taken by the system to access the disk memory.

Basic Operations of B Trees

The operations supported in B trees are Insertion, deletion and searching with the time complexity of **O(log n)** for every operation.

Insertion operation

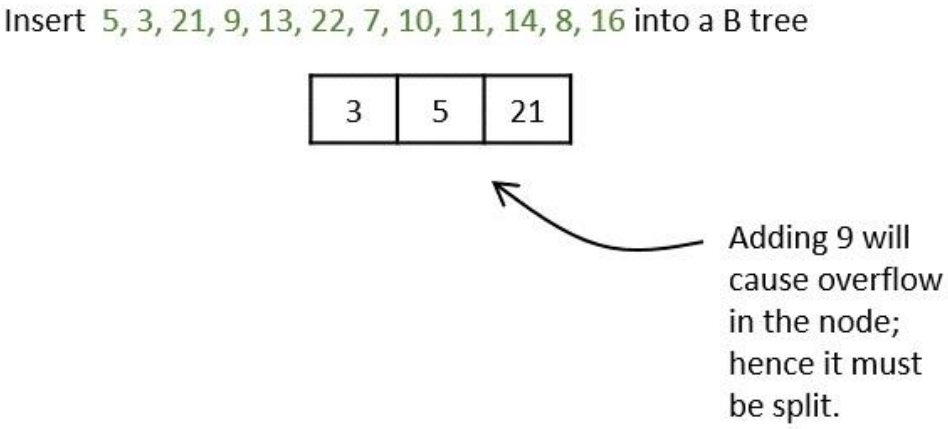
The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached. The insertion is done using the following procedure −

Step 1 − Calculate the maximum (m−1)(m−1) and, minimum (⌈m2⌉−1)(⌈m2⌉−1) number of keys a node can hold, where m is denoted by the order of the B Tree.

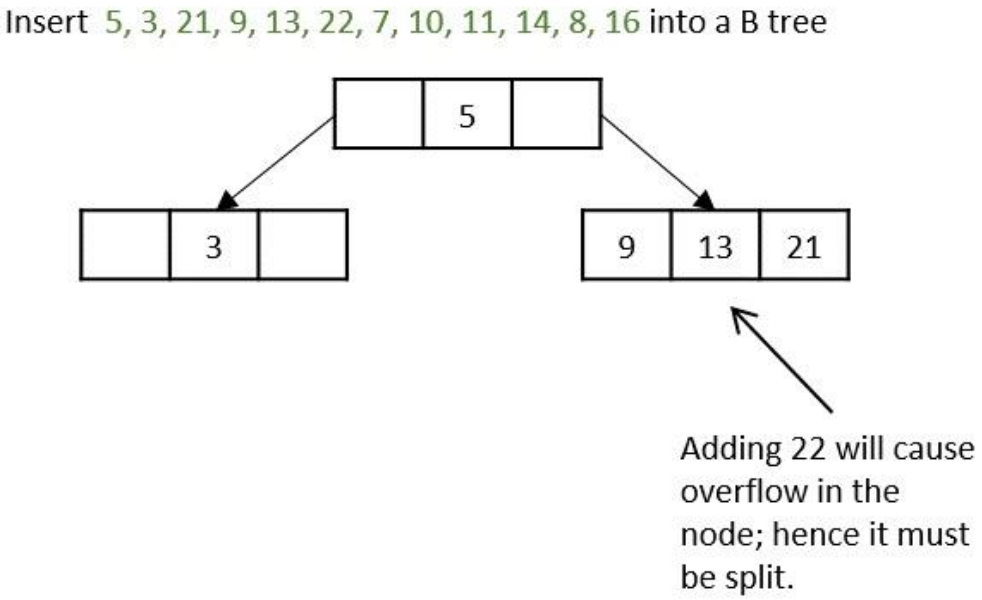
Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order (m) = 4
- Maximum Keys (m – 1) = 3
- Minimum Keys ($\lceil \frac{m}{2} \rceil$) – 1 = 1
- Maximum Children = 4
- Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

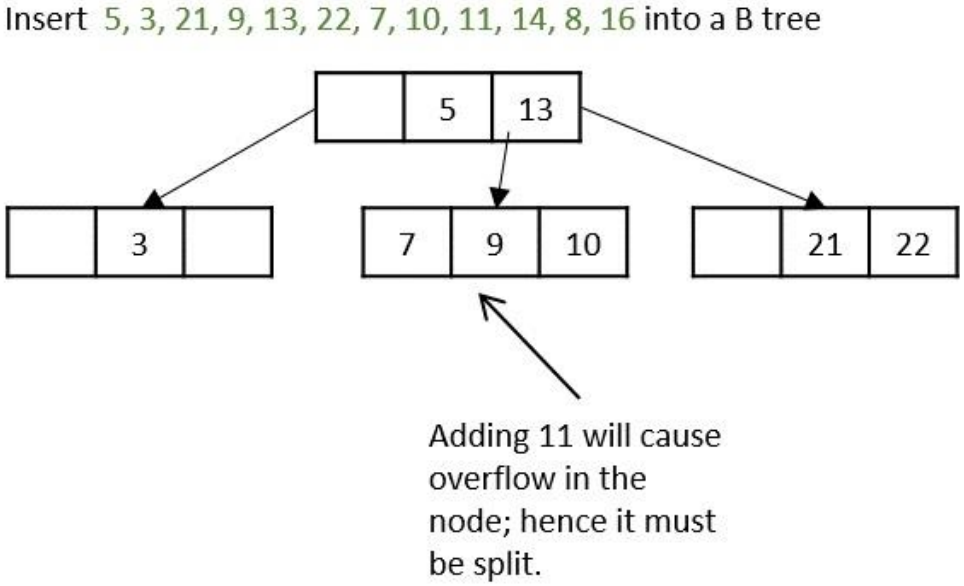
Step 2 − The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children.



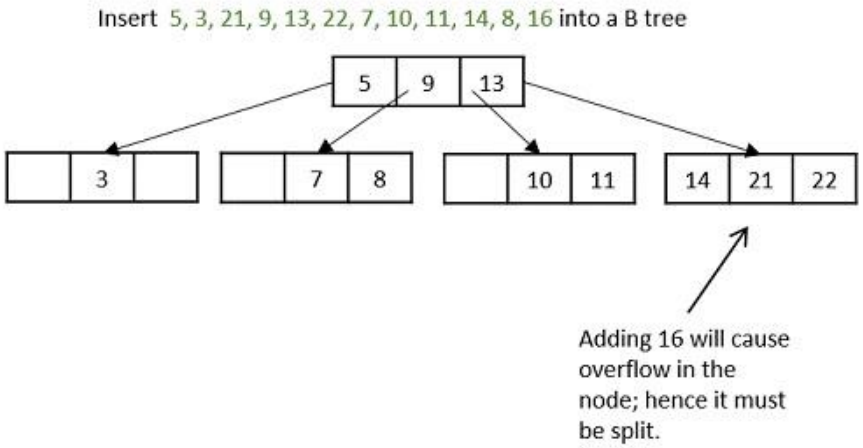
Step 3 − All the leaf nodes must be on the same level.



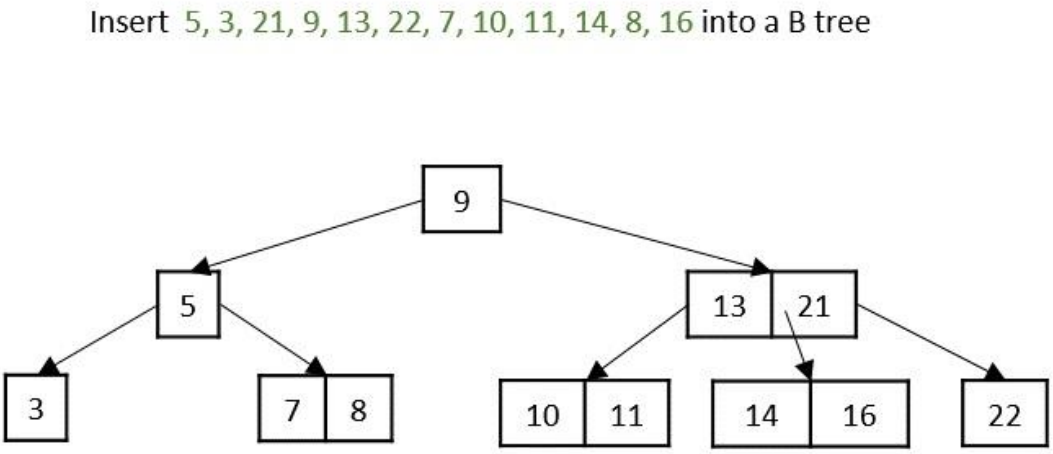
The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.



Another hiccup occurs during the insertion of 11, so the node is split and median is shifted to the parent.



While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.




```
// C Program for B trees

#include <stdio.h>

#include <stdlib.h>

struct BTree {
```

```
   //node declaration
   int *d;
   struct BTree **child_ptr;
   int l;
   int n;
};
struct BTree *r = NULL;
struct BTree *np = NULL;
struct BTree *x = NULL;
//creation of node
struct BTree* init() {
   int i;
   np = (struct BTree*)malloc(sizeof(struct BTree));
   //order 6
   np->d = (int*)malloc(6 * sizeof(int));
   np->child_ptr = (struct BTree**)malloc(7 * sizeof(struct BTree*));
   np->l = 1;
   np->n = 0;
   for (i = 0; i < 7; i++) {
      np->child_ptr[i] = NULL;
   }
   return np;
}
//traverse the tree
void traverse(struct BTree *p) {
   printf("\n");
   int i;
   for (i = 0; i < p->n; i++) {
      if (p->l == 0) {
```

```
        traverse(p->child_ptr[i]);
      }
      printf(" %d", p->d[i]);
    }
    if (p->l == 0) {
      traverse(p->child_ptr[i]);
    }
    printf("\n");
}
//sort the tree
void sort(int *p, int n) {
    int i, j, t;
    for (i = 0; i < n; i++) {
      for (j = i; j <= n; j++) {
        if (p[i] > p[j]) {
          t = p[i];
          p[i] = p[j];
          p[j] = t;
        }
      }
    }
}
int split_child(struct BTree *x, int i) {
    int j, mid;
    struct BTree *np1, *np3, *y;
    np3 = init();
    //create new node
    np3->l = 1;
    if (i == -1) {
```

```
    mid = x->d[2];
    //find mid
    x->d[2] = 0;
    x->n--;
    np1 = init();
    np1->l = 0;
    x->l = 1;
    for (j = 3; j < 6; j++) {
      np3->d[j - 3] = x->d[j];
      np3->child_ptr[j - 3] = x->child_ptr[j];
      np3->n++;
      x->d[j] = 0;
      x->n--;
    }
    for (j = 0; j < 6; j++) {
      x->child_ptr[j] = NULL;
    }
    np1->d[0] = mid;
    np1->child_ptr[np1->n] = x;
    np1->child_ptr[np1->n + 1] = np3;
    np1->n++;
    r = np1;
  } else {
    y = x->child_ptr[i];
    mid = y->d[2];
    y->d[2] = 0;
    y->n--;
    for (j = 3; j < 6; j++) {
      np3->d[j - 3] = y->d[j];
```

```
      np3->n++;
      y->d[j] = 0;
      y->n--;
    }
    x->child_ptr[i + 1] = y;
    x->child_ptr[i + 1] = np3;
  }
  return mid;
}
void insert(int a) {
  int i, t;
  x = r;
  if (x == NULL) {
    r = init();
    x = r;
  } else {
    if (x->l == 1 && x->n == 6) {
      t = split_child(x, -1);
      x = r;
      for (i = 0; i < x->n; i++) {
        if (a > x->d[i] && a < x->d[i + 1]) {
          i++;
          break;
        } else if (a < x->d[0]) {
          break;
        } else {
          continue;
        }
      }
```

```
      x = x->child_ptr[i];
    } else {
      while (x->l == 0) {
        for (i = 0; i < x->n; i++) {
          if (a > x->d[i] && a < x->d[i + 1]) {
            i++;
            break;
          } else if (a < x->d[0]) {
            break;
          } else {
            continue;
          }
        }
        if (x->child_ptr[i]->n == 6) {
          t = split_child(x, i);
          x->d[x->n] = t;
          x->n++;
          continue;
        } else {
          x = x->child_ptr[i];
        }
      }
    }
  }
  x->d[x->n] = a;
  sort(x->d, x->n);
  x->n++;
}
```

```
int main() {
  int i, n, t;
  insert(10);
  insert(20);
  insert(30);
  insert(40);
  insert(50);
  printf("Insertion Done");
  printf("\nB tree:");
  traverse(r);
  return 0;
}
```

Output

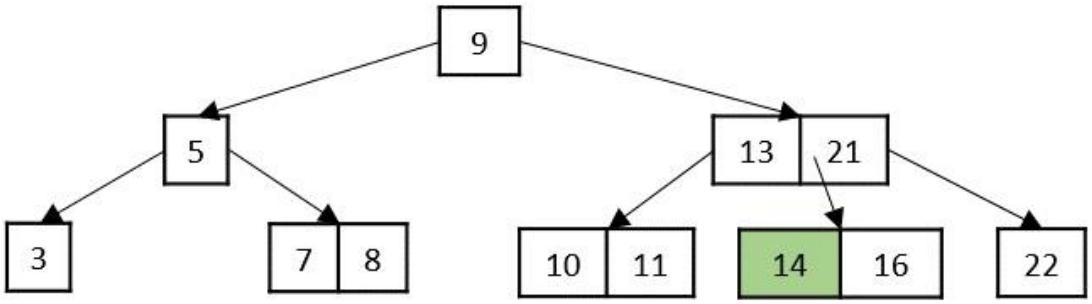
```
Insertion Done
B tree:
 10 20 30 40 50
```

Deletion operation

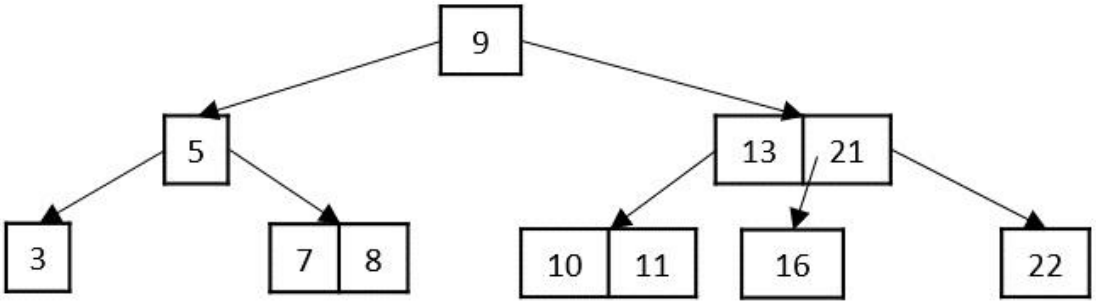
The deletion operation in a B tree is slightly different from the deletion operation of a Binary Search Tree. The procedure to delete a node from a B tree is as follows −

Case 1 − If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node.

Delete key 14

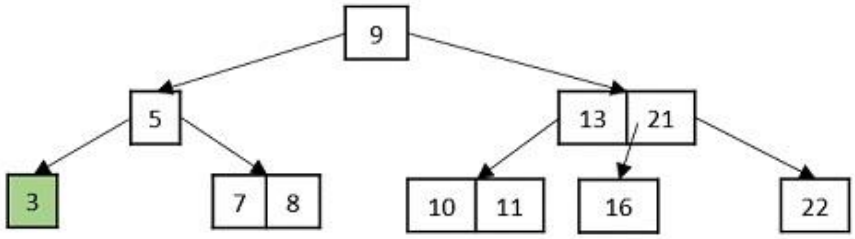


Delete key 14

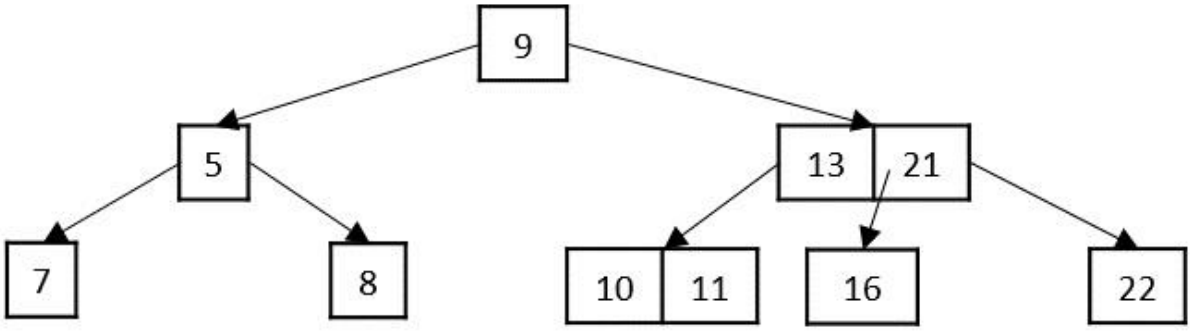


Case 2 − If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling. In case if both siblings have exact minimum number of keys, merge the node in either of them.

Delete key 3

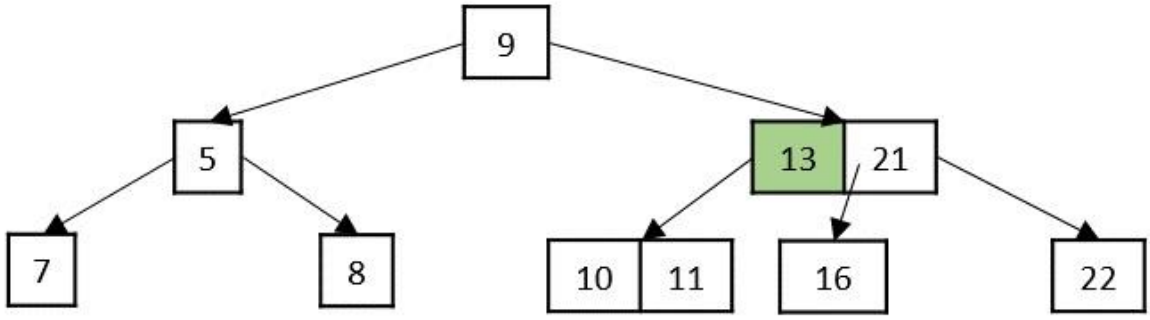


Delete key 3

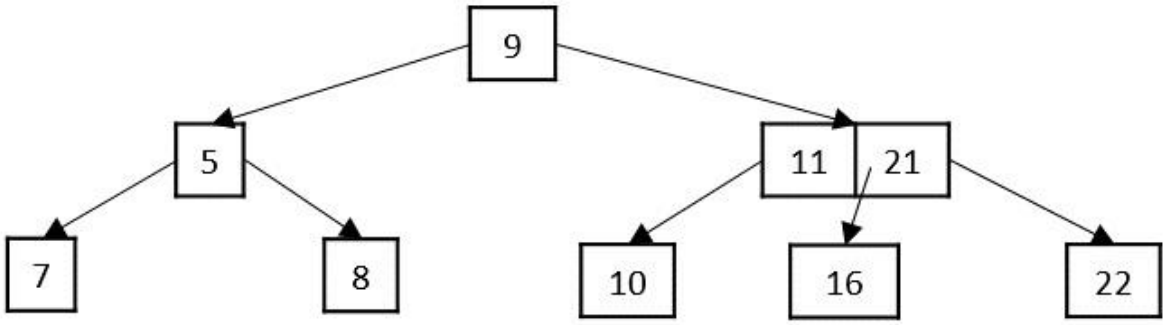


Case 3 − If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have minimum number of keys, they're merged together.

Delete key 13

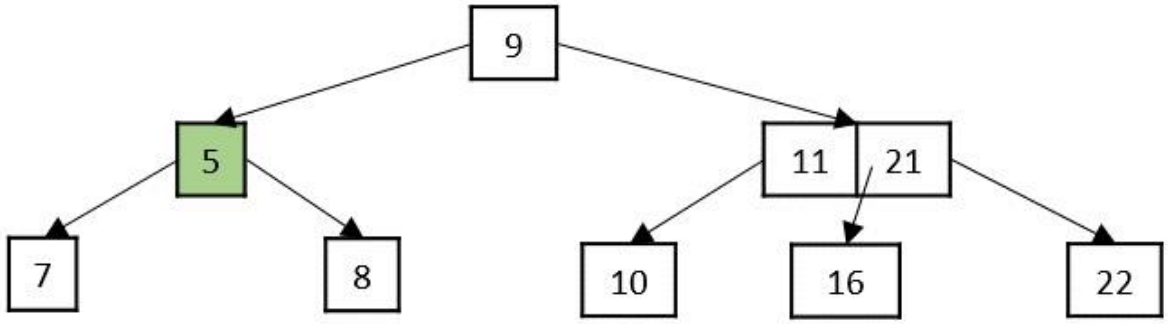


Delete key 13

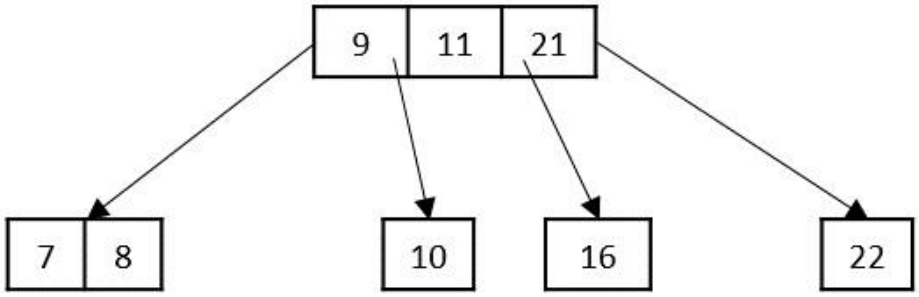


Case 4 − If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its sibling with its parent.

Delete key 5



Delete key 5




```
//deletion operation in BTree
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
#define MIN 2
struct BTreeNode {
  int item[MAX + 1], count;
  struct BTreeNode *linker[MAX + 1];
};
```

```
struct BTreeNode *root;
// creating node
struct BTreeNode *createNode(int item, struct BTreeNode *child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  newNode->item[1] = item;
  newNode->count = 1;
  newNode->linker[0] = root;
  newNode->linker[1] = child;
  return newNode;
}

// adding value to the node
void addValToNode(int item, int pos, struct BTreeNode *node,
        struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
  node->item[j + 1] = node->item[j];
  node->linker[j + 1] = node->linker[j];
  j--;
}
  node->item[j + 1] = item;
  node->linker[j + 1] = child;
  node->count++;
}

// Spliting the node
void splitNode(int item, int *pval, int pos, struct BTreeNode *node,
       struct BTreeNode *child, struct BTreeNode **newNode) {
```

```
  int median, j;
  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;
  *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->item[j - median] = node->item[j];
    (*newNode)->linker[j - median] = node->linker[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    addValToNode(item, pos, node, child);
  } else {
    addValToNode(item, pos - median, *newNode, child);
  }
  *pval = node->item[node->count];
  (*newNode)->linker[0] = node->linker[node->count];
  node->count--;
}

// Set the value in the node
int setValueInNode(int item, int *pval,
        struct BTreeNode *node, struct BTreeNode **child) {
int pos;
```

```
if (!node) {
  *pval = item;
  *child = NULL;
  return 1;
}
if (item < node->item[1]) {
  pos = 0;
} else {
  for (pos = node->count;
    (item < node->item[pos] && pos > 1); pos--);
    if (item == node->item[pos]) {
      printf("Duplicates not allowed\n");
    return 0;
  }
}
if (setValueInNode(item, pval, node->linker[pos], child)) {
  if (node->count < MAX) {
    addValToNode(*pval, pos, node, *child);
  } else {
    splitNode(*pval, pval, pos, node, *child, child);
    return 1;
  }
 }
  return 0;
}

// inserting elements in BTree
void insert(int item) {
  int flag, i;
```

```
  struct BTreeNode *child;
  flag = setValueInNode(item, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}

// Copy the successor
void copySuccessor(struct BTreeNode *myNode, int pos) {
  struct BTreeNode *dummy;
  dummy = myNode->linker[pos];
  for (; dummy->linker[0] != NULL;)
    dummy = dummy->linker[0];
  myNode->item[pos] = dummy->item[1];
}

// Remove the value in BTree
void removeVal(struct BTreeNode *myNode, int pos) {
  int i = pos + 1;
  while (i <= myNode->count) {
    myNode->item[i - 1] = myNode->item[i];
    myNode->linker[i - 1] = myNode->linker[i];
    i++;
  }
  myNode->count--;
}

// right shift
void rightShift(struct BTreeNode *myNode, int pos) {
  struct BTreeNode *x = myNode->linker[pos];
```

```
  int j = x->count;
  while (j > 0) {
    x->item[j + 1] = x->item[j];
    x->linker[j + 1] = x->linker[j];
  }
  x->item[1] = myNode->item[pos];
  x->linker[1] = x->linker[0];
  x->count++;
  x = myNode->linker[pos - 1];
  myNode->item[pos] = x->item[x->count];
  myNode->linker[pos] = x->linker[x->count];
  x->count--;
  return;
}

// left shift
void leftShift(struct BTreeNode *myNode, int pos) {
  int j = 1;
  struct BTreeNode *x = myNode->linker[pos - 1];
  x->count++;
  x->item[x->count] = myNode->item[pos];
  x->linker[x->count] = myNode->linker[pos]->linker[0];
  x = myNode->linker[pos];
  myNode->item[pos] = x->item[1];
  x->linker[0] = x->linker[1];
  x->count--;
  while (j <= x->count) {
    x->item[j] = x->item[j + 1];
    x->linker[j] = x->linker[j + 1];
```

```
    j++;
  }
  return;
}

// Merge the nodes
void mergeNodes(struct BTreeNode *myNode, int pos) {
  int j = 1;
  struct BTreeNode *x1 = myNode->linker[pos], *x2 = myNode->linker[pos   - 1];
  x2->count++;
  x2->item[x2->count] = myNode->item[pos];
  x2->linker[x2->count] = myNode->linker[0];
  while (j <= x1->count) {
    x2->count++;
    x2->item[x2->count] = x1->item[j];
    x2->linker[x2->count] = x1->linker[j];
    j++;
  j = pos;
  while (j < myNode->count) {
     myNode->item[j] = myNode->item[j + 1];
     myNode->linker[j] = myNode->linker[j + 1];
     j++;
  }
  myNode->count--;
  free(x1);
  }
}

// Adjust the node in BTree
```

```
void adjustNode(struct BTreeNode *myNode, int pos) {
  if (!pos) {
    if (myNode->linker[1]->count > MIN) {
      leftShift(myNode, 1);
    } else {
      mergeNodes(myNode, 1);
    }
  } else {
    if (myNode->count != pos) {
      if (myNode->linker[pos - 1]->count > MIN) {
        rightShift(myNode, pos);
      } else {
        if (myNode->linker[pos + 1]->count > MIN) {
          leftShift(myNode, pos + 1);
        } else {
          mergeNodes(myNode, pos);
        }
      }
    } else {
      if (myNode->linker[pos - 1]->count > MIN)
        rightShift(myNode, pos);
      else
        mergeNodes(myNode, pos);
    }
  }
}

// Delete a value from the node
int delValFromNode(int item, struct BTreeNode *myNode) {
```

```
int pos, flag = 0;
if (myNode) {
  if (item < myNode->item[1]) {
  pos = 0;
  flag = 0;
}else {
  for (pos = myNode->count; (item < myNode->item[pos] && pos > 1); pos--);
    if (item == myNode->item[pos]) {
      flag = 1;
    } else {
      flag = 0;
    }
 }
 if (flag) {
   if (myNode->linker[pos - 1]) {
     copySuccessor(myNode, pos);
     flag = delValFromNode(myNode->item[pos], myNode->linker[pos]);
     if (flag == 0) {
       printf("Given data is not present in B-Tree\n");
      }
     } else {
       removeVal(myNode, pos);
    }
 } else {
  flag = delValFromNode(item, myNode->linker[pos]);
 }
 if (myNode->linker[pos]) {
  if (myNode->linker[pos]->count < MIN)
    adjustNode(myNode, pos);
```

```
    }
  }
  return flag;
}

// Delete operaiton in BTree
void delete (int item, struct BTreeNode *myNode) {
  struct BTreeNode *tmp;
    if (!delValFromNode(item, myNode)) {
      printf("Not present\n");
      return;
    } else {
    if (myNode->count == 0) {
      tmp = myNode;
      myNode = myNode->linker[0];
      free(tmp);
    }
  }
  root = myNode;
  return;
}

void display(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      display(myNode->linker[i]);
      printf("%d ", myNode->item[i + 1]);
    }
```

```
    display(myNode->linker[i]);
  }
}
int main() {
  int item, ch;
  insert(8);
  insert(9);
  insert(10);
  insert(11);
  insert(15);
  insert(16);
  insert(17);
  insert(18);
  insert(20);
  insert(23);
  printf("Insertion Done");
  printf("\nBTree elements before deletion: \n");
  display(root);
  int ele = 20;
  printf("\nThe element to be deleted: %d", ele);
  delete (ele, root);
  printf("\nBTree elements before deletion: \n");
  display(root);
}
```

Output

```
Insertion Done
BTree elements before deletion:
8 9 10 11 15 16 17 18 20 23
The element to be deleted: 20
BTree elements before deletion:
8 9 10 11 15 16 17 18 23 8 9 23
```

Applications of AVL Tree:

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

Advantages of AVL Tree:

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

Disadvantages of AVL Tree:

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

Application of B-Tree:

B-trees are commonly used in applications where large amounts of data need to be stored and retrieved efficiently. Some of the specific applications of B-trees include:

- **Databases:** B-trees are widely used in databases to store indexes that allow for efficient searching and retrieval of data.
- **File systems:** B-trees are used in file systems to organize and store files efficiently.
- **Operating systems:** B-trees are used in operating systems to manage memory efficiently.
- **Network routers:** B-trees are used in network routers to efficiently route packets through the network.
- **DNS servers:** B-trees are used in Domain Name System (DNS) servers to store and retrieve information about domain names.
- **Compiler symbol tables:** B-trees are used in compilers to store symbol tables that allow for efficient compilation of code.

Advantages of B-Tree:

B-trees have several advantages over other data structures for storing and retrieving large amounts of data. Some of the key advantages of B-trees include:

- **Sequential Traversing:** As the keys are kept in sorted order, the tree can be traversed sequentially.
- **Minimize disk reads:** It is a hierarchical structure and thus minimizes disk reads.
- **Partially full blocks:** The B-tree has partially full blocks which speed up insertion and deletion.

Disadvantages of B-Tree:

- **Complexity:** B-trees can be complex to implement and can require a significant amount of programming effort to create and maintain.
- **Overhead:** B-trees can have significant overhead, both in terms of memory usage and processing time. This is because B-trees require additional metadata to maintain the tree structure and balance.
- **Not optimal for small data sets:** B-trees are most effective for storing and retrieving large amounts of data. For small data sets, other data structures may be more efficient.
- **Limited branching factor:** The branching factor of a B-tree determines the number of child nodes that each node can have. B-trees typically have a fixed branching factor, which can limit their performance for certain types of data.