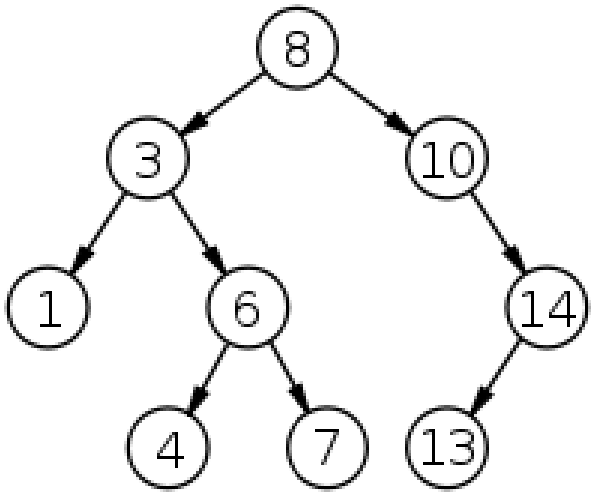
BINARY SEARCH TREE:

THEORY:

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

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

Simple example of binary tree:



Some of the main operation in the Binary Search Trees are:

- Find/ Find Minimum / Find Maximum element in binarysearchtrees
- Inserting an element in binarysearchtrees
- Deleting an element from binarysearchtrees

ALGORITHM:

INSERTION:

Suppose NEWNODE is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed.

1. Input the DATA to be pushed and ROOT node of the tree.
2. NEWNODE = Create a New Node.
3. If (ROOT == NULL)
(a) ROOT=NEW NODE
4. Else If (DATA < ROOT → Info)
(a) ROOT = ROOT → Lchild
(b) GoTo Step 4
5. Else If (DATA > ROOT → Info)
(a) ROOT = ROOT → Rchild
(b) GoTo Step 4

6. If (DATA < ROOT → Info)
(a) ROOT → LChild = NEWNODE
7. Else If (DATA > ROOT → Info)
(a) ROOT → RChild = NEWNODE
8. Else (a) Display ("DUPLICATE NODE")
(b) EXIT
9. NEW NODE → Info = DATA
10. NEW NODE → LChild = NULL
11. NEW NODE → RChild = NULL
12. EXIT

SEARCHING A NODE:

1. Input the DATA to be searched and assign the address of the root node to ROOT.

2. If (DATA == ROOT → Info)

(a) Display “The DATA exist in the tree”

(b) GoTo Step 6

3. If (ROOT == NULL)

(a) Display “The DATA does not exist”

(b) GoTo Step 6

4. If(DATA > ROOT→Info)

(a) ROOT = ROOT→RChild

(b) GoTo Step 2

5. If(DATA < ROOT→Info)

(a) ROOT = ROOT→Lchild

(b) GoTo Step 2

6. Exit

DELETING A NODE:

On deleting a node from the binary search tree on of the following condition may arise:

i. The node to be deleted has no children
ii. The node to be deleted has exactly one children
iii. The node to be deleted has two children

ALGORITHM:

NODE is the current position of the tree, which is in under consideration. LOC is the place where node is to be replaced. DATA is the information of node to be deleted.

1. Find the location NODE of the DATA to be deleted.

2. If (NODE = NULL) (a) Display “DATA is not in tree” (b) Exit

3. If (NODE → LChild = NULL) (a) LOC = NODE (b) NODE = NODE → RChild

4. If (NODE → Rchild= =NULL) (a) LOC = NODE (b) NODE = NODE → LChild

5. If ((NODE → LChild not equal to NULL) && (NODE → Rchild not equal to NULL))

(a) LOC = NODE → RChild

6. While (LOC → LChild not equal to NULL)

(a) LOC = LOC → LChild

7. LOC → LChild = NODE → LChild

8. LOC → RChild= NODE → RChild

9. Exit

Finding a smallest/largest key:

```
Node*findsmall (node *tree)
{
        If tree=null
                Print null tree
        Else if (tree ->left=null)
                Return tree
        Else
                Return findsmall (tree->left)
}
```

Finding maximum value

1. Define a struct pointer as node*n

2. If (n == NULL)

Return 0;

3. Traverse through right link only as

```
While (n->right! = NULL)
        n = n->left;
```

4. Print n->data

Finding the height of the tree:

```
Int height (node*tree)
{
    Int Heightleft, Heightright
    If tree = null
        Return 0
        If (tree->left=null &&tree->right=null)
            Return 1
        Heightleft=height(tree->left)+1
        Heightright=height(tree->right)+1
        If(Heightleft>Heightright)
            Return (heightleft+1)
        Else
            Return (heightright+1)
        }
```

PROGRAM CODE:

```
# include <iostream>
# include <cstdlib>
using namespace std;
struct node
{
```

```
    int info;
    struct node *left;
    struct node *right;
}*root;

class BST
{
private:
    int max,min;
    public:
        void find(int, node **, node **);
        void insert(node *, node *);
        void search(int key, node *leaf);
        void del(int);
        void case_a(node *,node *);
        void case_b(node *,node *);
        void case_c(node *,node *);
        void display(node *, int);
        void findmaxmin(node *);
        find_min(node *n);
        find_max(node *n);
        BST()
        {
            root = NULL;
        }
```

```
};
int main()
{
   int choice, num;
   BST bst;
   node *temp;
   while (1)
   {
      cout<<"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"<<endl;
      cout<<"Operations on BST"<<endl;
      cout<<"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"<<endl;
      cout<<"1.Insert Element "<<endl;
      cout<<"2.Delete Element "<<endl;
      cout<<"3.Display"<<endl;
      cout<<"4.find max&min value"<<endl;
      cout<<"5.search"<<endl;
      cout<<"6.Quit"<<endl;
      cout<<"Enter your choice : ";
      cin>>choice;
      switch(choice)
      {
      case 1:{
         temp = new node;
         cout<<"Enter the number to be inserted : ";
           cin>>temp->info;
```

```
    bst.insert(root, temp);
}
case 2:{
    if (root == NULL)
    {
        cout<<"Tree is empty, nothing to delete"<<endl;
        continue;
    }
    cout<<"Enter the number to be deleted : ";
    cin>>num;
    bst.del(num);
    break;
    case 3:
    cout<<"Display BST:"<<endl;
    bst.display(root,1);
    cout<<endl;
}
    break;
case 4:{
    bst.findmaxmin(root);
}
case 5:{
   int key;
    cout<<"enter the data to find:";
    cin>>key;
```

```
            bst.search(key,root);
        }
        case 6:
            exit(1);
        default:
            cout<<"Wrong choice"<<endl;
        }
    }
}

void BST::insert(node *tree, node *newnode)
{
    if (root == NULL)
    {
        root = new node;
        root->info = newnode->info;
        root->left = NULL;
        root->right = NULL;
        cout<<"Root Node is Added"<<endl;

    }
    else if (tree->info == newnode->info)
    {
        cout<<"Element already in the tree"<<endl;
```

```
}
else if (tree->info > newnode->info)
{
   if (tree->left != NULL)
   {
      insert(tree->left, newnode);
        }
        else
        {
      tree->left = newnode;
      (tree->left)->left = NULL;
      (tree->left)->right = NULL;
      cout<<"Node Added To Left"<<endl;

   }
}
else
{
   if (tree->right != NULL)
   {
      insert(tree->right, newnode);
   }
   else
   {
      tree->right = newnode;
```

```
            (tree->right)->left = NULL;
            (tree->right)->right = NULL;
            cout<<"Node Added To Right"<<endl;

        }
    }
}
void BST::del(int item)
{
    node *parent, *location;
    if (root == NULL)
    {
        cout<<"Tree empty"<<endl;
        return;
    }
    find(item, &parent, &location);
    if (location == NULL)
    {
        cout<<"Item not present in tree"<<endl;
        return;
    }
    if (location->left == NULL && location->right == NULL)
        case_a(parent, location);
    if (location->left != NULL && location->right == NULL)
        case_b(parent, location);
```

```
    if (location->left == NULL && location->right != NULL)
        case_b(parent, location);
    if (location->left != NULL && location->right != NULL)
        case_c(parent, location);
    free(location);
}
void BST::case_a(node *par, node *loc )
{
    if (par == NULL)
    {
        root = NULL;
    }
    else
    {
        if (loc == par->left)
            par->left = NULL;
        else
            par->right = NULL;
    }
}
void BST::case_b(node *par, node *loc)
{
    node *child;
    if (loc->left != NULL)
        child = loc->left;
```

```
    else
        child = loc->right;
    if (par == NULL)
    {
        root = child;
    }
    else
    {
        if (loc == par->left)
            par->left = child;
        else
            par->right = child;
    }
}
void BST::case_c(node *par, node *loc)
{
    node *ptr, *ptrsave, *suc, *parsuc;
    ptrsave = loc;
    ptr = loc->right;
    while (ptr->left != NULL)
    {
        ptrsave = ptr;
        ptr = ptr->left;
    }
    suc = ptr;
```

```
    parsuc = ptrsave;
    if (suc->left == NULL && suc->right == NULL)
        case_a(parsuc, suc);
    else
        case_b(parsuc, suc);
    if (par == NULL)
    {
        root = suc;
    }
    else
    {
        if (loc == par->left)
            par->left = suc;
        else
            par->right = suc;
    }
    suc->left = loc->left;
    suc->right = loc->right;
}
void BST::display(node *ptr, int level)
{
    int i;
    if (ptr != NULL)
    {
        display(ptr->right, level+1);
```

```
        cout<<endl;
        if (ptr == root)
            cout<<"Root->:  ";
        else
        {
            for (i = 0;i < level;i++)
                cout<<"       ";
          }
        cout<<ptr->info;
        display(ptr->left, level+1);
    }
}
void BST::findmaxmin(node *bt) {
    min = find_min(bt);
    max = find_max(bt);
    cout<<"maximum value = "<<max<<endl<<"minimum value =
"<<min<<endl;
}
int BST::find_min(node *n) {
    if (n == NULL) {
        return 0;

    }
    while (n->left != NULL) {
        n = n->left;
    }
```

```
    return n->info;
}
int BST::find_max(node *n) {
    if (n == NULL) {
        return 0;
    }
    while (n->right != NULL) {
        n = n->right;
    }
    return n->info;
}
void BST::search(int key, node *leaf)
{
  if(leaf!=NULL)
  {
    if(key==leaf->info)
      cout<<"node found"<<endl;
    else if(key<leaf->info){
        while ((leaf->left)->info!= key)
        leaf = leaf->left;
        cout<<"node found"<<endl;
    }
      else
      {
        while ((leaf->right)->info!= key)
```

```
        leaf = leaf->right;
        cout<<"node found"<<endl;
     }
  }

}
void BST::find(int item, node **par, node **loc)
{

    node *ptr, *ptrsave;
    if (root == NULL)
    {
        *loc = NULL;
        *par = NULL;
        return;
    }
    if (item == root->info)
    {
        *loc = root;
        *par = NULL;
        return;
    }
    if (item < root->info)
        ptr = root->left;
    else
```

```
        ptr = root->right;
    ptrsave = root;
    while (ptr != NULL)
    {
        if (item == ptr->info)
        {
            *loc = ptr;
            *par = ptrsave;
            return;
        }
        ptrsave = ptr;
        if (item < ptr->info)
            ptr = ptr->left;
          else
              ptr = ptr->right;
    }
    *loc = NULL;
    *par = ptrsave;
}
```

OUTPUT:

Enter your choice:1

Enter the number to be inserted:17

Data added.

Enter your choice:1

Enter the number to be inserted:13

Data added to the left.

Enter your choice:1

Enter the number to be inserted:27

Data added to the right.

Enter your choice:4

Maximum value=27

Minimum value=13

Enter your choice:5

Enter the data to search:27

Node found

Enter your choice:3

Enter the data to be deleted:13

Data deleted.

Enter your choice:2

The data are:

17

27

AVL TREE:

THEORY:

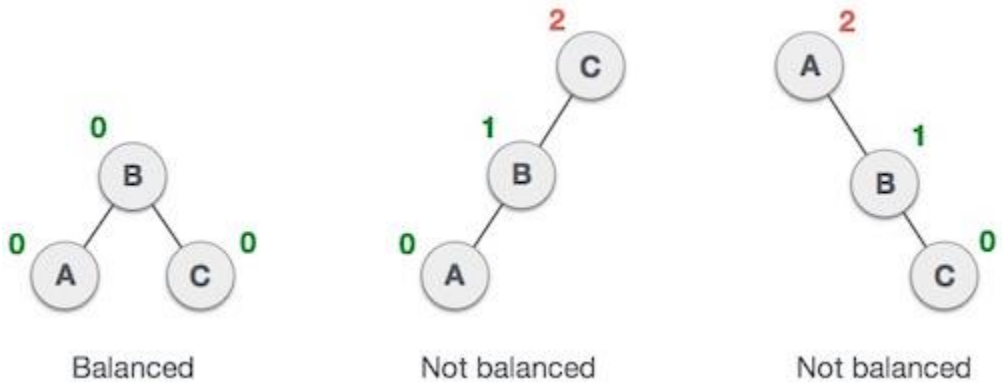
AVL tree is another balanced binary search tree. It is named after their two inventors, Adelson –Velskii and Landis, they were the first person to purpose the dynamically balanced tree. AVL tree checks the height of the left and right sub-trees at each node.

Properties of AVL trees:

A binary tree is said to be an AVL tree, if:

1. It is a binary search tree
2. For any node X, the height of left subtree of X and height of right subtree of X differ at most by 1.

The difference thus calculated is termed as the **Balanced Factor.**



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

Left rotation:

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.

Right rotation:

If a tree becomes unbalanced, when a node is inserted into the left subtree of the left subtree, then we perform a single right rotation.

Left-Right rotation:

If the new node to be inserted is at right subtree of left child of critical node then double rotation RR rotation followed by LL rotation is done which is called left-right rotation.

Right left rotation:

If the new node to be inserted is at left subtree of right child of critical node then double rotation LL rotation followed by RR rotation is done which is called right-left rotation.

ALGORITHM:

INSERTION:

1. Insert the node in the same way as in an ordinary binary tree.

2. Trace a path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.

3. Consider the node with the imbalance and the two nodes on the layers immediately below.

4. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.

5. If these three nodes lie in a dogleg pattern (i.e., there is a bend in the path) apply a double rotation to correct the imbalance.

6. Exit.

SEARCHING OPERATION IN AVL TREE:

1. Read the search the element from the user

2. Compare , the search element with the value of root node in the tree
3. If both are matching, then display “Given node found” and terminate the function
4. If both are not matching, then check whether search element is smaller or larger than that node value
5. If search element is smaller, then continue the search process in left subtree
6. If search element is smaller, then continue the search process in left subtree
7. If search element is larger, then continue the search process in right subtree
8. If we reach to the node with the search value, then display “element is found” and terminate the function
9. If we reach to a leaf node and it is also not matching, then display “element is not found” and terminate the function.

DELETION IN AVL TREE:

Delete (T, k) means delete a node with the key k from the AVL tree T

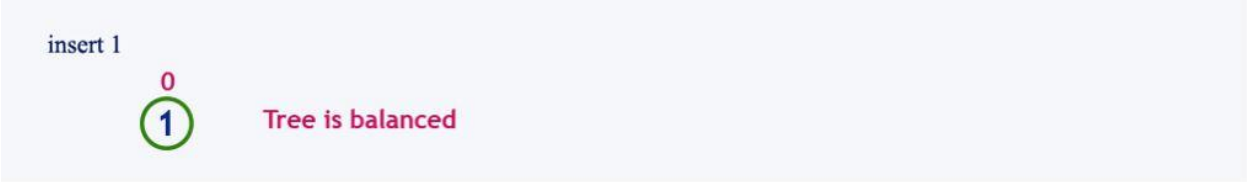
I) First: find the node x where k is stored

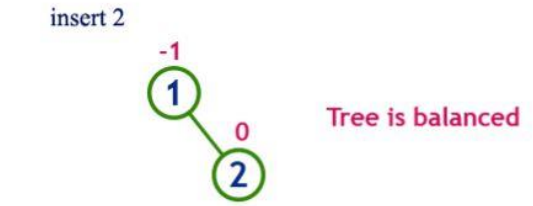
II) Second: delete the contents of node x

There are three possible cases (just like for BSTs):
1) If x has no children (i.e., is a leaf), delete x.
2) If x has one child, let x' be the child of x.
   Notice: x' cannot have a child, since subtrees of T can differ in height by at most one
   -replace the contents of x with the contents of x'
   -delete x' (a leaf)
3) If x has two children,
   -find x's successor z (which has no left child)
   -replace x's contents with z's contents, and
   -delete z.
   [since z has at most one child, so we use case (1) or (2) to delete z]

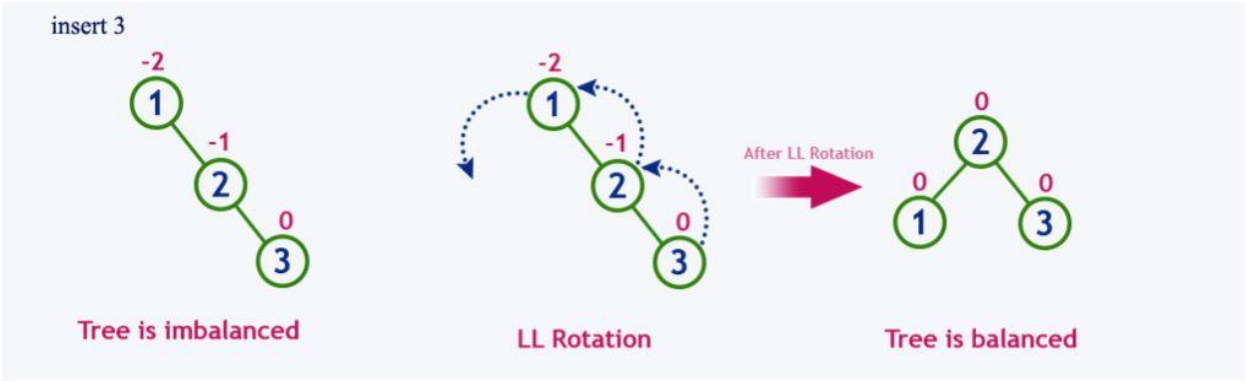
In all 3 cases, we end up removing a leaf.

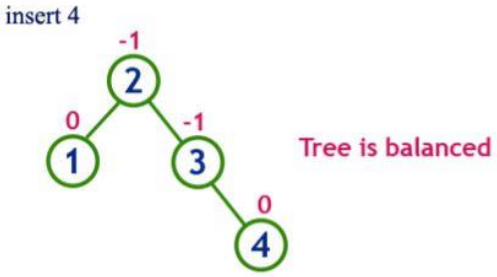
III) Third: Go from the deleted leaf towards the root and at each ancestor of that leaf:



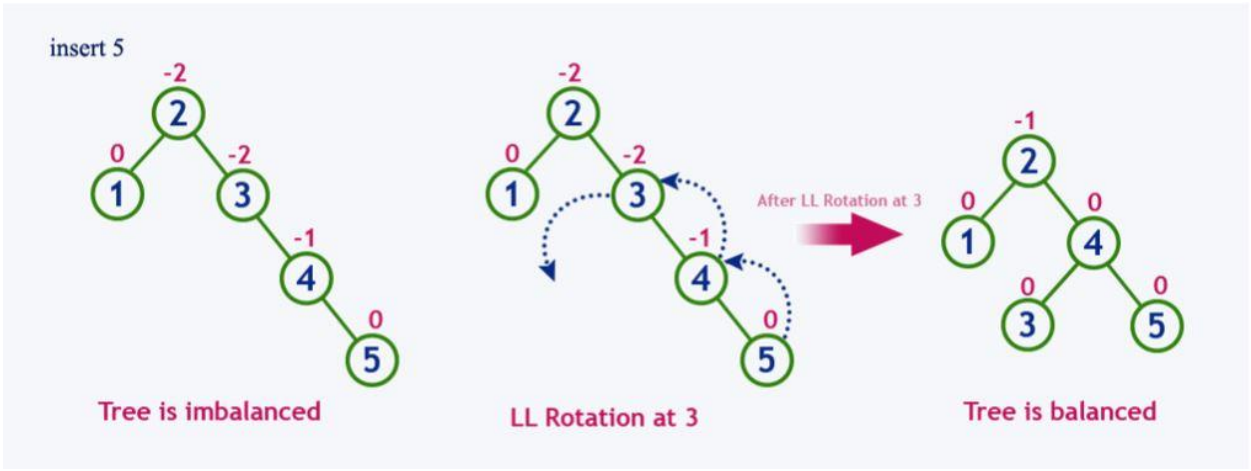


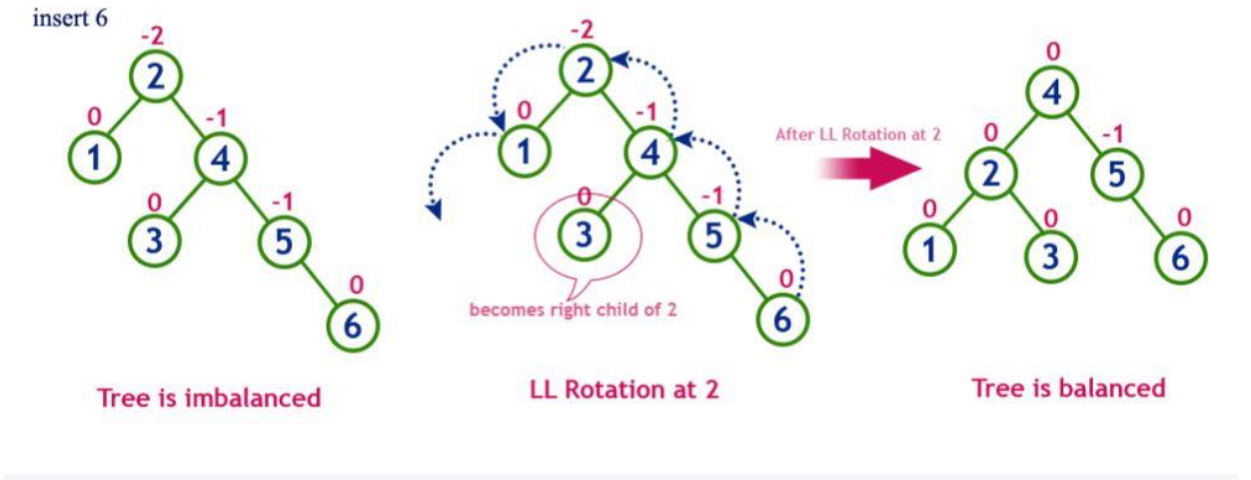
-update the balance factor
-rebalance with rotations if necessary.

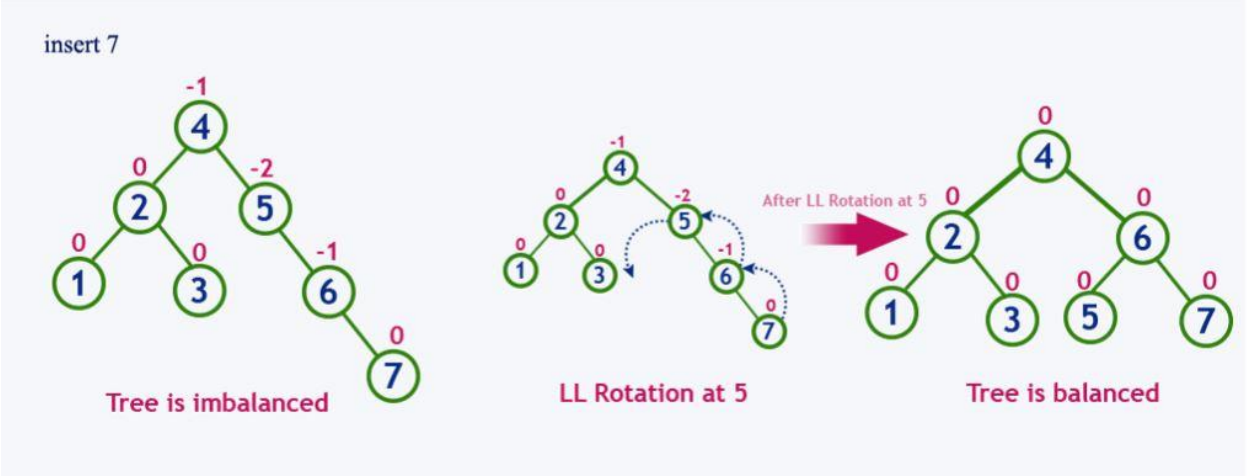


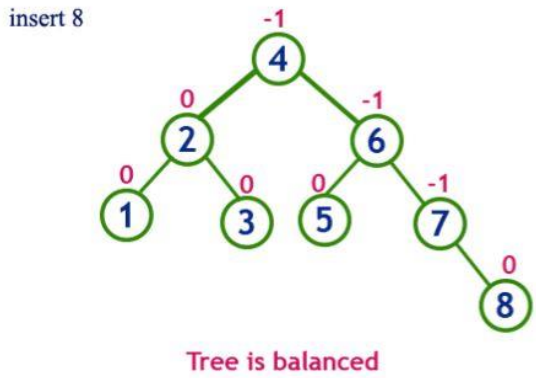


FOR EXAMPLE:


insert 5
Tree is imbalanced
LL Rotation at 3
After LL Rotation at 3
Tree is balanced

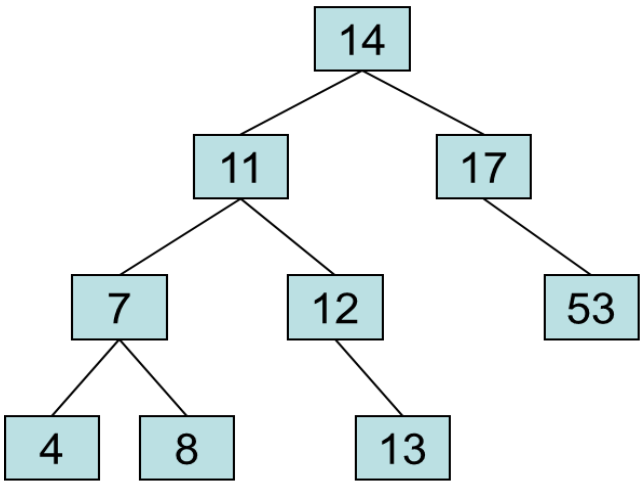

insert 6
Tree is imbalanced
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
Tree is balanced



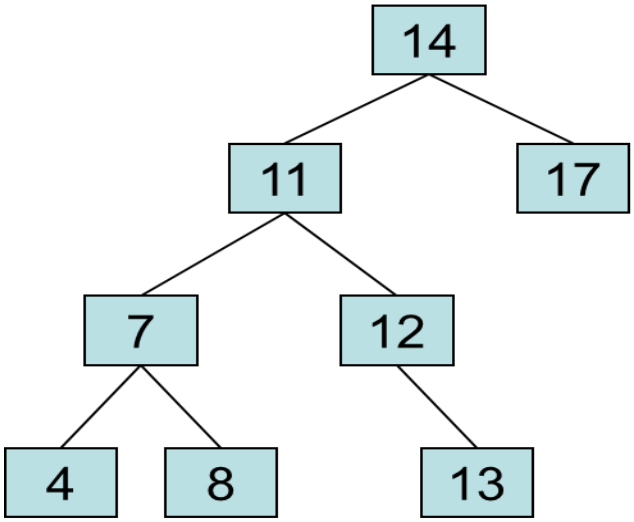


DELETION EXAMPLE:

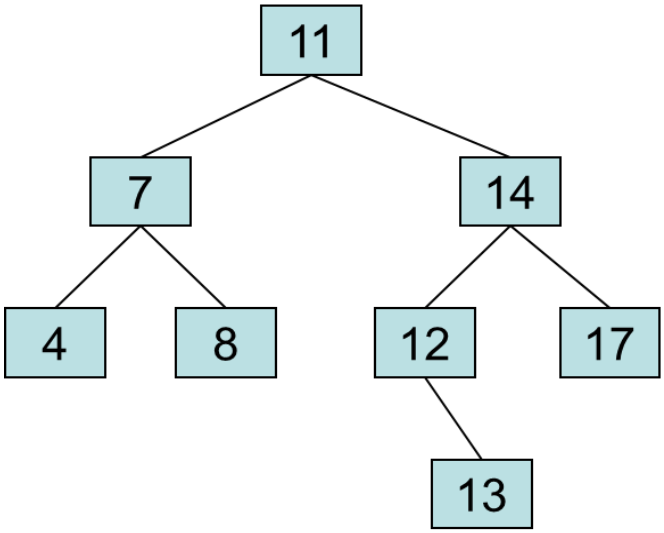
The AVL tree is balanced:



- **Now remove 53, tree will be unbalanced**



- **Now the tree is Balanced after rotation**



APPLICATION OF AVL TREE:

AVL trees are used for frequent insertion. One of the examples I know of is it is used in Memory management subsystem of Linux kernel to search memory regions of processes during preemption.

HUFFMAN TREE:

THEORY:

Huffman tree is a full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet. It is developed by David A. Huffman.

Define the weighted path length of a leaf to be its weight times its depth. The Huffman tree is the binary tree with minimum external path weight, i.e., the one with the minimum sum of weighted path lengths for the given set of leaves. So the goal is to build a tree with the minimum external path weight.

Algorithm for Huffman tree:

For the 'n' ext nodes with w1,w2, ……..,wn weights Huffman algorithm is:

1. Suppose there are 'n' weights w1,w2,……..,wn or 'n' external nodes
2. Create an ascending order priority queue for it
3. Take two minimum weighted nodes to form new node (for example if w1 and w2 are minimum weighted nodes then new nodes w1 to w2 is created)
4. Now the remaining nodes will be w1+w2 ,w3,…..wn
5. Create all subtree at last weight

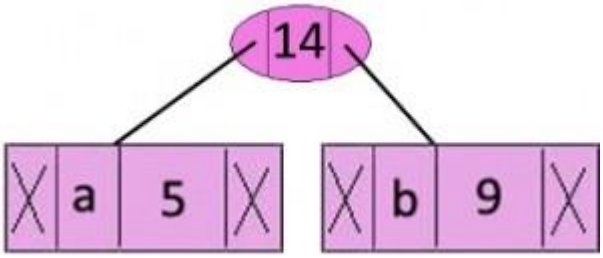
Example of Huffman tree:

Character	Frequency
A	5
B	9
C	12
D	13

E	16
F	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

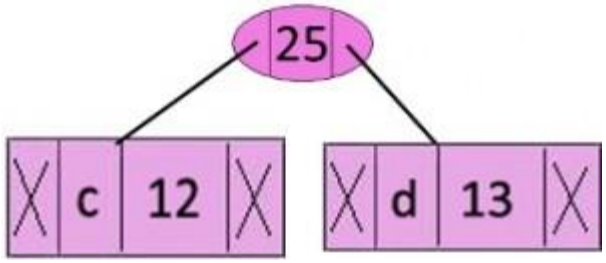
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

Character	Frequency
C	12
D	13
Internal node	14
E	16
F	45

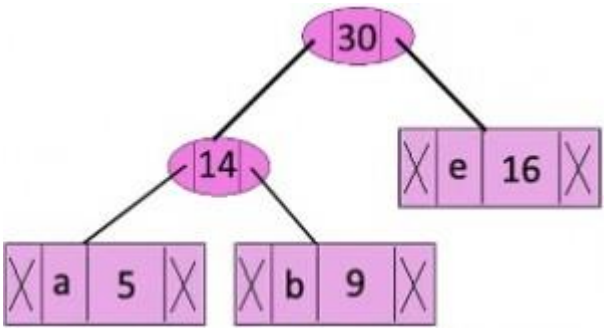
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

Character	Frequency
Internal node	14
E	16
Internal node	25
F	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

Character	Frequency
Internal node	25
Internal node	30

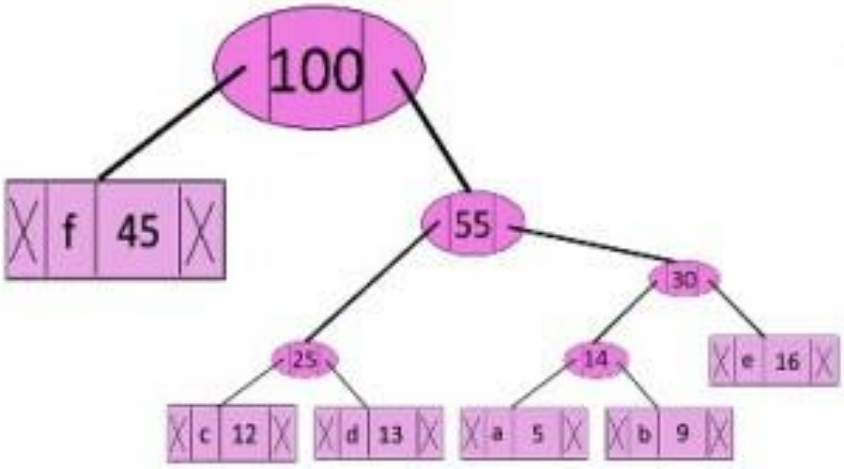
F	45

Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

Now min heap contains 2 nodes.

Character	Frequency
F	45
Internal nodes	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



Now min heap contains only one node.

Character	Frequency
Internal nodes	100

APPLICATION OF HUFFMAN TREE:

a) Variable length coding based on estimated probability of occurrence for each possible value of source character
b) For 26 characters, it must be represented by 5 bits representation (fixed length coding)
c) So, the character will high probability of occurrence is assigned less no. of bits in variable length coding.

B-TREE:

THEORY:

B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node. B-trees are tree data structure that are most commonly found in database and file systems. B-trees keep data sorted and allow amortized logarithmic time insertions and deletions. B-trees generally grow from the bottom up as elements are inserted, whereas most binary tree grow down.

It was developed by Rudolf Bayer but one thing interesting about this 'B' he didn't name it. The most common belief is that B stands for Balanced, as all the leaf nodes are at the same level in the tree.

B-tree of order m has the following properties:

Property #1 - All the leaf nodes must be at same level.

Property #2 - All nodes except root must have at least [m/2] keys and maximum of m-1 keys.

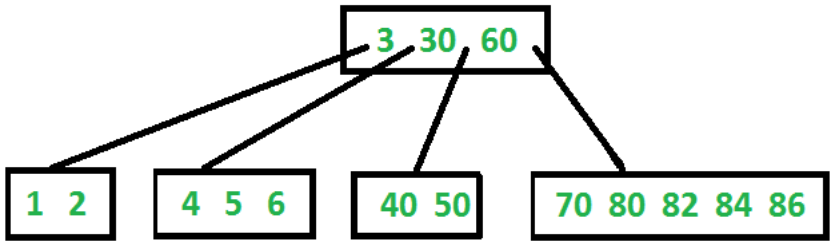
Property #3 - All non-leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

Property #4 - If the root node is a non-leaf node, then it must have at least 2 children.

Property #5 - A non-leaf node with n-1 keys must have n number of children.

Property #6 - All the key values within a node must be in Ascending Order.

Example of B-tree:



OPERATIONS ON B-TREE:

1. Search
2. Insertion
3. Deletion

ALGORITHM:

SEARCH:

1. Read the search element from the user
2. Compare, the search element with first key value of root node in the tree.
3. If both are matching, then display "Given node found!!!" and terminate the function
4. If both are not matching, then check whether search element is smaller or larger than that key value.
5. If search element is smaller, then continue the search process in left subtree.
6. If search element is larger, then compare with next key value in the same node and repeat step 3, 4, 5 and 6 until we found exact match or comparison completed with last key value in a leaf node.

7. If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

INSERTION :

In a B-Tree, the new element must be added only at leaf node. That means, always the new key Value is attached to leaf node only. The insertion operation is performed as follows...

1. Check whether tree is Empty.
2. If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
3. If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
4. If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
5. If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.
6. If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

DELETION

1. First, search for the value which will be deleted. Then, remove the value from the node which contains it.

2. If no node is in an illegal state then the process is finished. 3. If some node is in an illegal state then there are two possible cases:

(a) Its sibling node, a child of the same parent node, can transfer one or more of its child nodes to the current node and return it to a legal state. If so, after updating the separation values of the parent and the two siblings the process is finished.

(b) Its sibling does not have an extra child because it is on the lower bound. In that case both siblings are merged into a single node and we recurse onto the parent node, since it has had a child node removed. This

continues until the current node is in a legal state or the root node is reached, upon which the root's children are merged and the merged node becomes the new root.

For example:

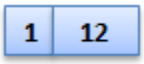
key: - 1,12,8,2,25,6,14,28,17,7,52,16,48,68,3,26,29,53,55,45,67

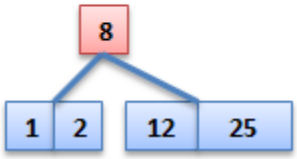
Order = 5

Procedure for adding key in b-tree

Step1. Add first key as root node.

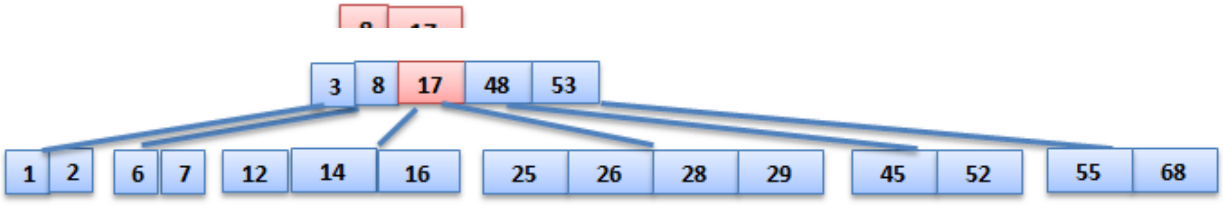
Step2. Add next key at the appropriate place in sorted order.

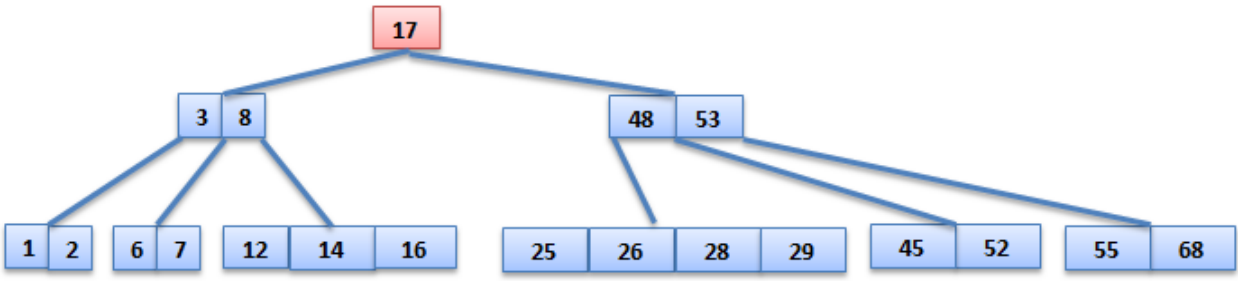




Step3. Same process applied until root node full. if root node full then splitting process applied.

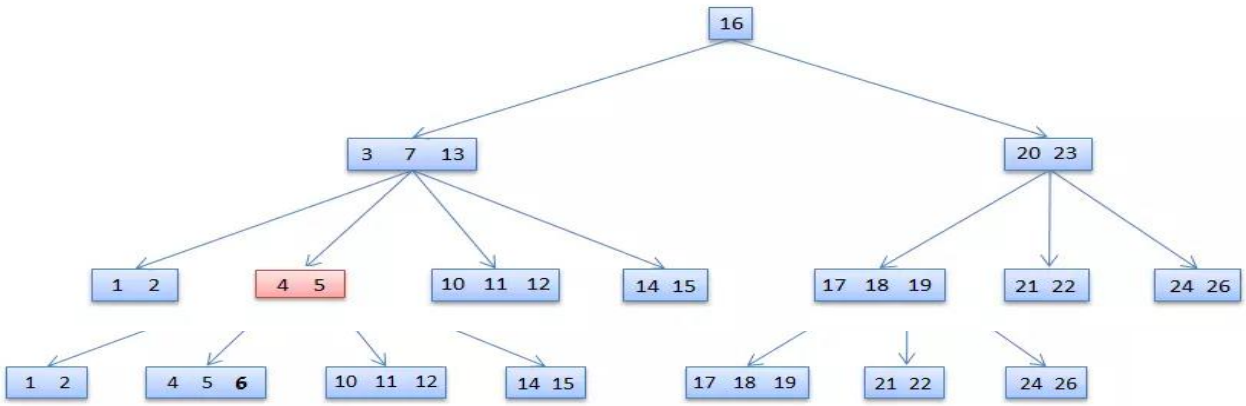
Some important steps:





Deletion operation in B-Tree:

Case-I



If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted. key k is in node x and x is a leaf, simply delete k from x.

6 deleted:

Case-II

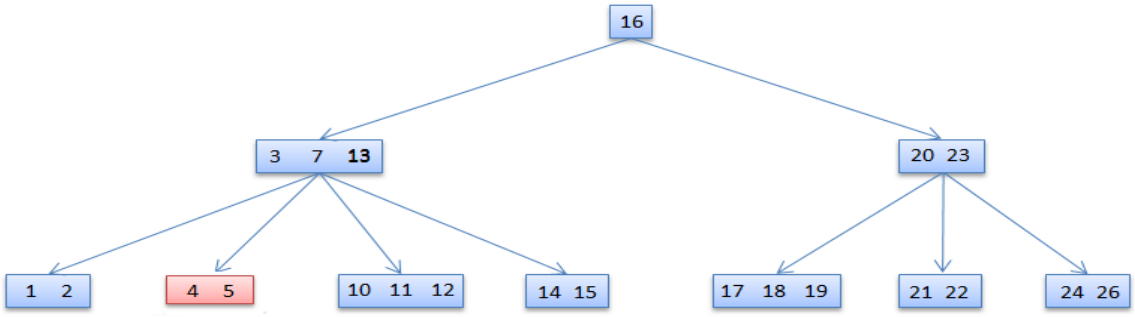
If key k is in node x and x is an internal node, there are three cases to consider

Case-II-a

If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y. Recursively delete k' and replace k with k' in x

Case-II-b

Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before. Note that finding k'



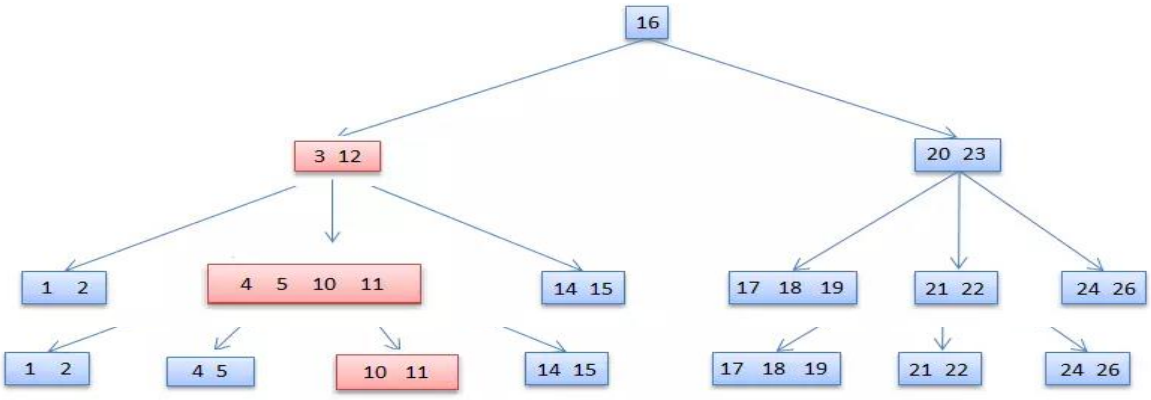
and deleting it can be performed in a single downward pass.

13 deleted:

Case-II-c

Otherwise, if both y and z have only t−1 (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x. y now contains 2t − 1 keys, and subsequently k is deleted.

7 deleted:



Case-III

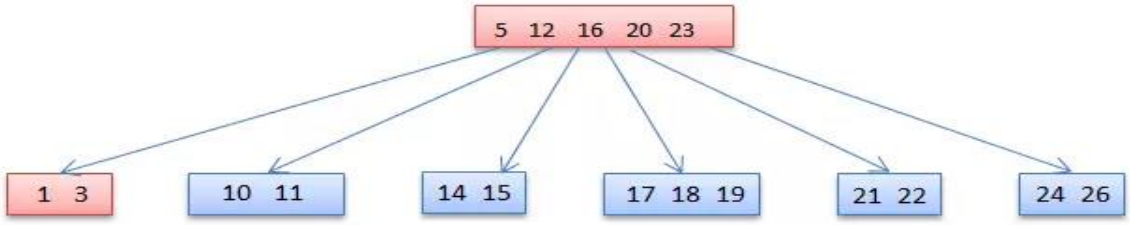
If key k is not present in an internal node x, determine the root of the appropriate subtree that must contain k. If the root has only t − 1 keys, execute either of the following two cases to ensure that we descend to a

node containing at least t keys. Finally, recurse to the appropriate child of x.

Case-III-a

If the root has only t−1 keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

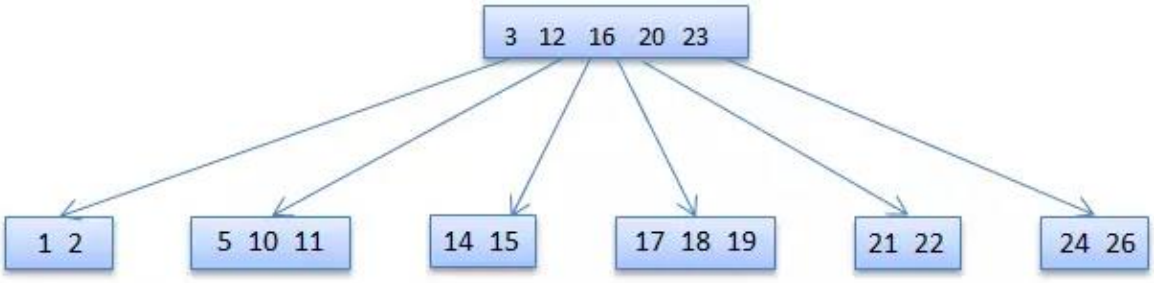
2 deleted:



Case-III-b

If the root and all of its siblings have t−1 keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

4 deleted:



CONCLUSION

So here in this project we studied about the several topics of data structure and algorithm which are the must in computer engineering. Through this we become capable for storing and organizing the data in the particular order. Data structures can also be used to group and organize other data structures.