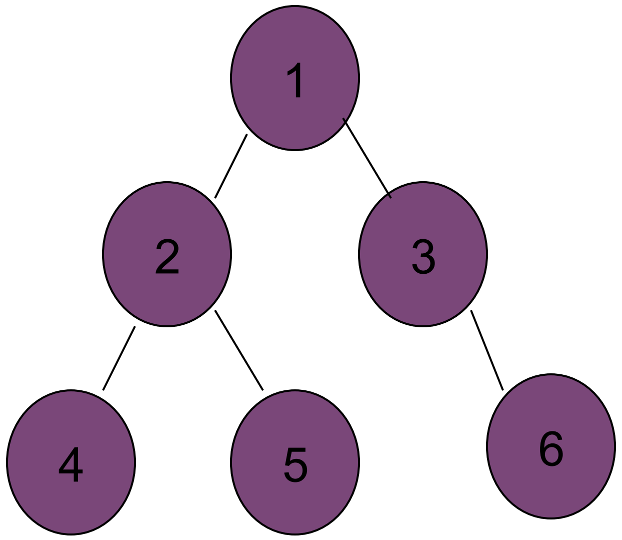
Trees

General Definition of Tree

- A tree consists of nodes connected by edges, which do not form cycle.
- For collection of nodes & edges to define as tree, there must be one & only one path from the root to any other node.
- A tree is a connected graph of N vertices with N-1 Edges.
- Tree is nonlinear data structure.

bitCode technologies



Recursive definition of Tree

A tree is finite set of one or more nodes such that:

a) There is specially designated node called root.

b) The remaining nodes are partitioned into n>=0 disjoint sets T1… Tn where each of these sets is a tree. T1….Tn are called subtree of the root.

+ Tree Terminologies

Node: A node stands for the item of information plus the branches of other items. E.g. 1,2,3,4,5,6

Siblings: Children of the same parent are siblings. E.g. siblings of node 2 are 4 & 5.

Degree: The number of sub trees of a node is called degree. The degree of a tree is the maximum degree of the nodes in the tree. E.g. degree of above tree is 2.

Leaf Nodes: Nodes that have the degree as zero are called leaf nodes or terminal nodes. Other nodes are called non terminal nodes.

bitCode technologies

+ Tree Terminologies

Ancestor: The ancestor of a node are all the nodes along the path from the root to that node.

Level: The level of a node is defined by first letting the root of the tree to be level = 1. If a node is at level x then the children are at level x+1.

Height/Depth: The height or depth of the tree is defined as the maximum level of any node in the tree.

Binary Tree

- If the degree of a tree is 2 then it is called binary tree.
- A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called left sub tree & right sub tree.
- The max number of nodes on level n of binary tree is 2^(n-1), n>= 1.
- The max number of nodes in binary tree of level k is (2^k)-1.
- If every non leaf node in a binary tree has non empty left and right sub trees, the tree is termed as a strictly binary tree.

+ Application

Information retrieval is the most important use of Binary tree.

A binary tree is useful when two way decisions must be made at each point in a process.

e.g. Binary search tree.

In decision trees where each node has a condition and based on the answer to the condition the tree traversal takes place.

Tree traversals

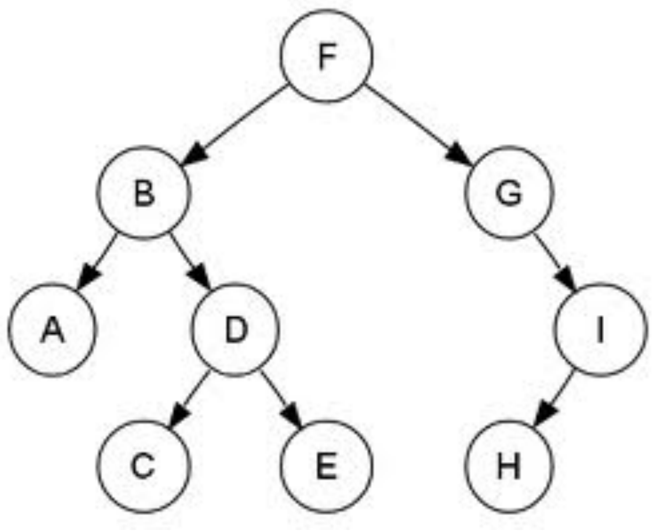
- Preorder
- Inorder
- Postorder

Algorithm for Preorder traversal (DLR)

- Visit the node first
- Traverse left subtree in preorder
- Traverse right subtree in preorder
- Continue this process till all nodes have been visited

+ Preorder traversal

- Find out Preorder traversal for following tree.



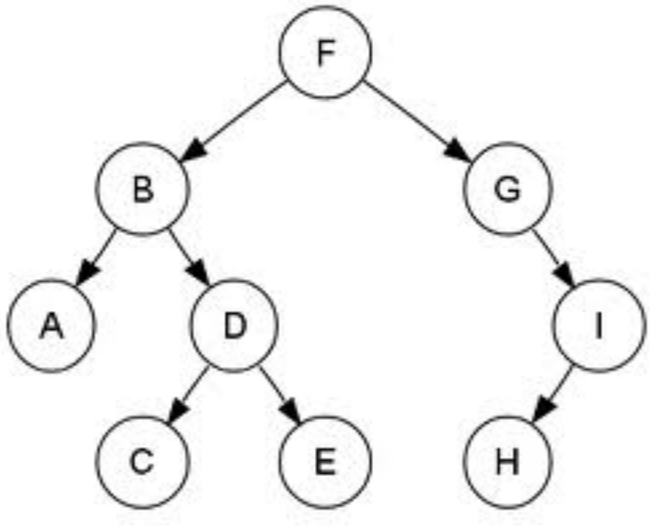
- F B A D C E G I H

+ Algorithm for Inorder traversal (LDR)

- Traverse left subtree in inorder
- Visit the node
- Traverse right subtree in inorder
- Continue this process till all nodes have been visited

Inorder traversal

- Find out Inorder traversal for following tree.



- A B C D E F G H I

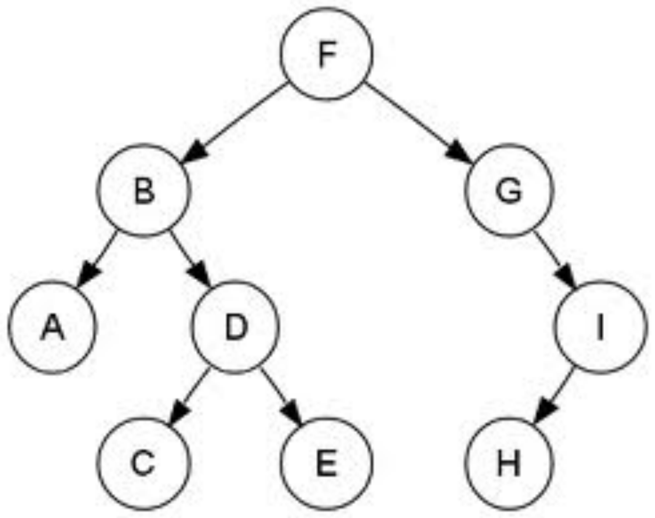
bitCode technologies

+ Algorithm for Postorder traversal (LRD)

- Traverse left subtree in postorder
- Traverse right subtree in postorder
- Visit the node
- Continue this process till all nodes have been visited

+ Postorder traversal

- Find out Postorder traversal for following tree.



- A C E D B H I G F

bitCode technologies

Level wise printing data in a tree (BFS)

- Insert the root node into queue
- While the queue is not empty do the following
- Delete a node from the queue and print it
- Insert a node corresponding to it's left subtree into the queue if it is not NULL
- Insert a node corresponding to it's right subtree into the queue if it is not NULL
- Go to the step 2
- Stop

+ Classes for Binary Tree

- Class Node will have the following structure

```
class Node {
        int data;
        Node * right;
        Node * left;
  public:
  Node( int data );
  int  getData();
  void setData( int );
  void setRight( Node * );
  Node * getRight();
  void setLeft( Node * );
  Node * getLeft();

};
```

Class BinaryTree will have the following structure

```
class BinaryTree {
        Node * root;
public:
        TBinaryTree();
        void insert( int );
        void inOrder();
        void preOrder();
        void postOrder();
        void deleteData( int );
        ~BinaryTree();
};
```

bitCode technologies

+ Binary search tree

It is a binary tree that is either empty or in which each node contains a key that satisfies the conditions:

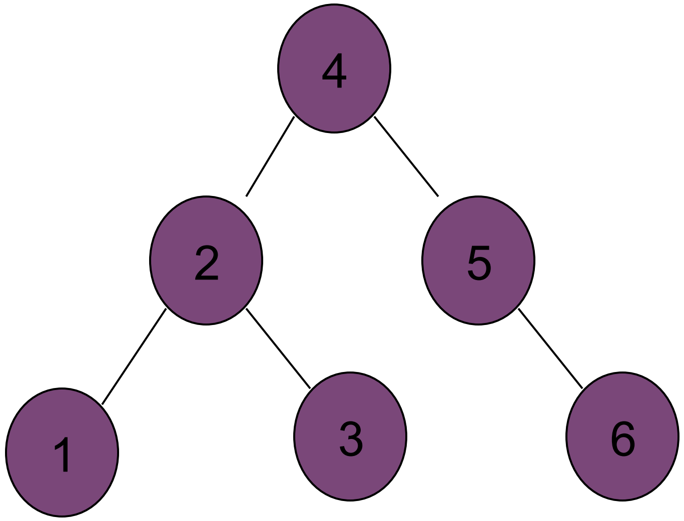
1. All keys (if any) in the left sub tree of the root precede the key in the root.
2. The key in the root precedes all keys (if any) in the right sub tree.
3. The left and right sub trees of the root are again search trees.

bitCode technologies

+ Example Binary search Tree

Put following numbers in a linked list into a binary search tree

4,2,5,1,3,6



bitCode technologies

+ Exercise

Create a binary search tree for the following sequence of numbers

10, 30, 50, 5, 40, 7, 3, 45, 20

Traverse the created in

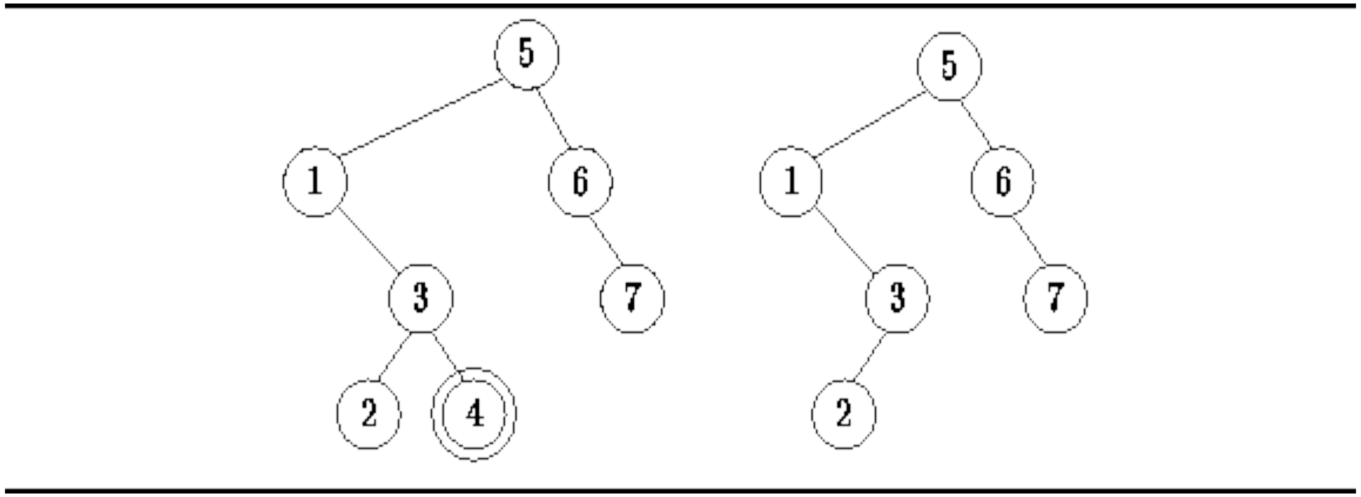
1. Preorder
2. Inorder
3. Postorder

Removing Items from Binary Search Tree

- When removing an item from a search tree, it is imperative that the tree satisfies the data ordering criterion.
- If the item to be removed is in a leaf node, then it is fairly easy to remove that item from the tree since doing so does not disturb the relative order of any of the other items in the tree.

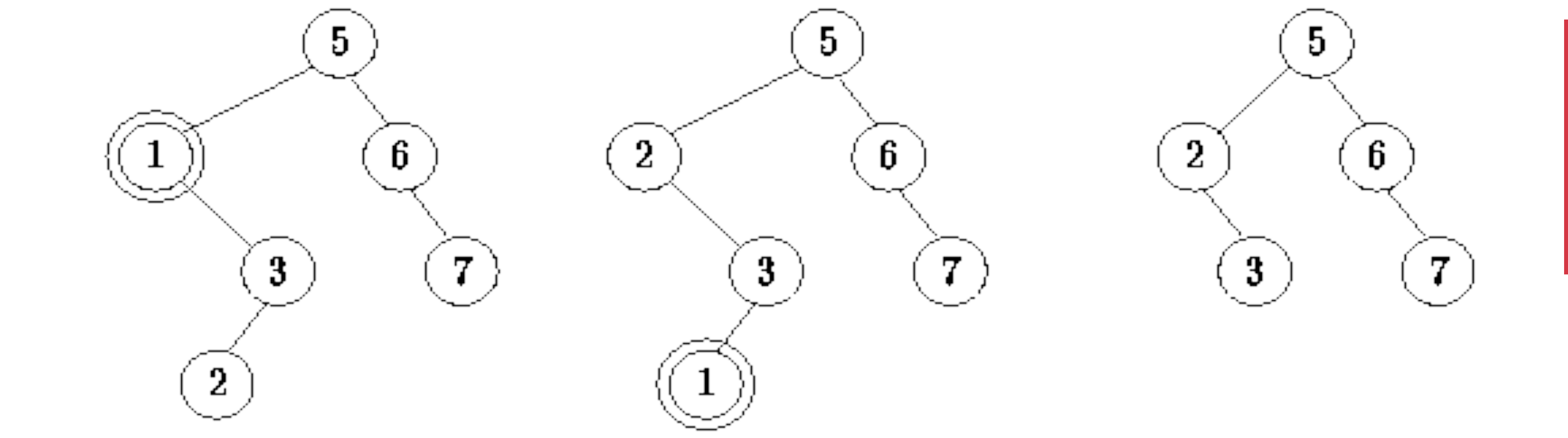
Example

- For example, consider the binary search tree shown in Figure (a). Suppose we wish to remove the node labeled 4. Since node 4 is a leaf, its subtrees are empty. When we remove it from the tree, the tree remains a valid search tree as shown in Figure (b).



Removing Non Leaf Node from a Binary Search Tree

- To remove a non-leaf node, we move it down in the tree until it becomes a leaf node since a leaf node is easily deleted.
- To move a node down we swap it with another node which is further down in the tree.
- For example, consider the search tree shown in Figure (a). Node 1 is not a leaf since it has an empty left sub tree but a non-empty right sub tree.
- To remove node 1, we swap it with the smallest key in its right sub tree, which in this case is node 2, Figure (b).
- Since node 1 is now a leaf, it is easily deleted. Notice that the resulting tree remains a valid search tree, as shown in Figure (c).



(a) (b) (c)

- To move a non-leaf node down in the tree, we either swap it with the smallest key in the right subtree or with the largest one in the left subtree.
- At least one such swap is always possible, since the node is a non-leaf and therefore at least one of its subtrees is non-empty.
- If after the swap, the node to be deleted is not a leaf, then we push it further down the tree with yet another swap. Eventually, the node must reach the bottom of the tree where it can be deleted

Creation Binary Tree from Traversal Sequence

- Creation of Binary Tree from Preorder & Inorder Traversals
  - E.g. Inorder E A C K F H D B G
  - Preorder F A E K C D H G B

- Creation of Binary Tree from Postorder & Inorder Traversals.
  - Inorder: B I D A C G E H F
  - Postorder: I D B G C H F E A

bitCode technologies

Expression Tree

- When an expression is represented through a tree, it is known as expression tree.
- The leaves of an expression tree are operands, such as constant variable names and all internal nodes contain operators.
- Preorder traversal of an expression tree gives prefix equivalent of the expression.
- Postorder traversal of an expression gives the postfix equivalent of the expression.

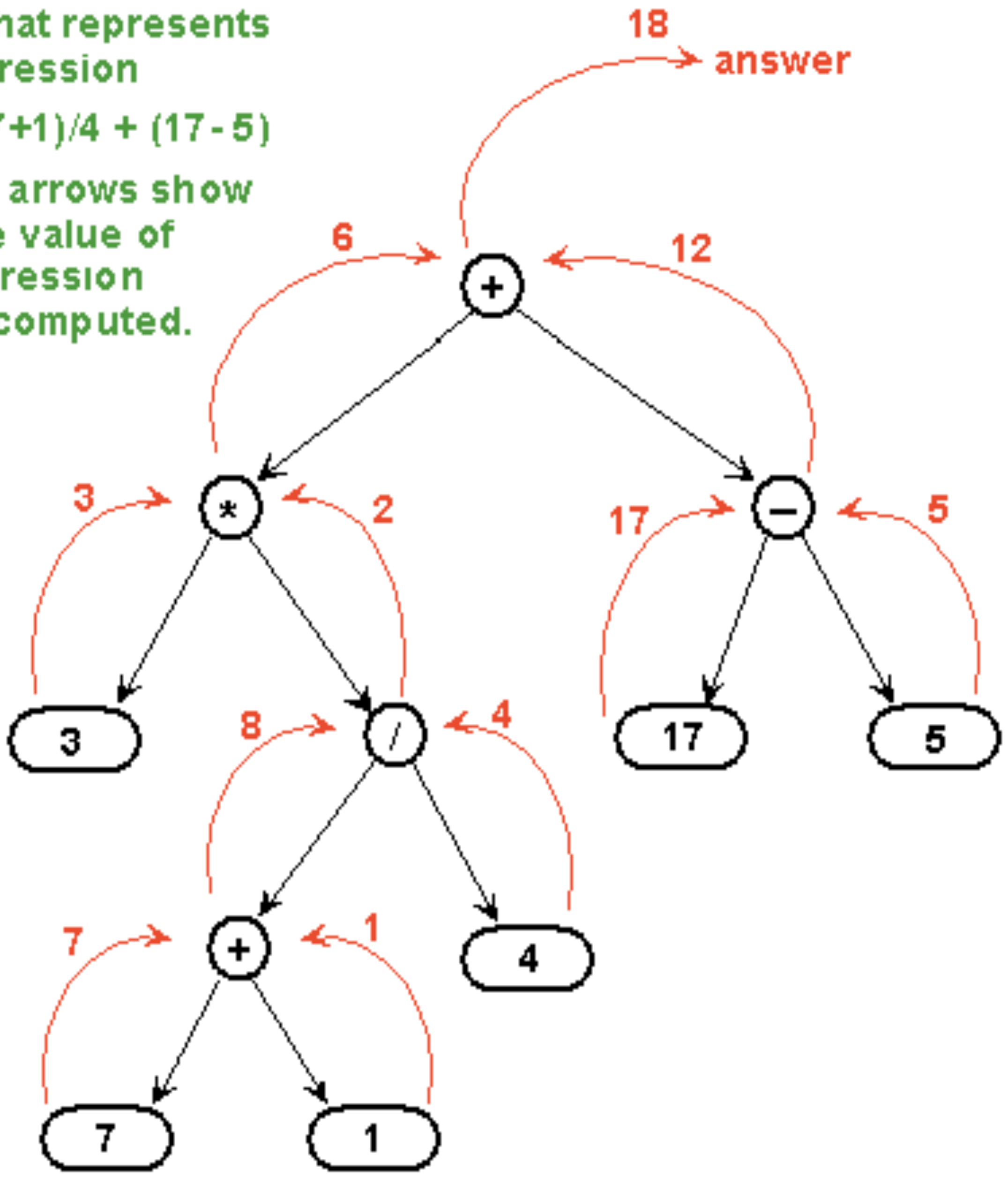
bitCode technologies

Expression Tree

A tree that represents the expression

3 * (7+1)/4 + (17-5)

The red arrows show how the value of the expression can be computed.



bitCode technologies

Threaded Binary Search Tree

- A **threaded binary search tree** may be defined as follows:

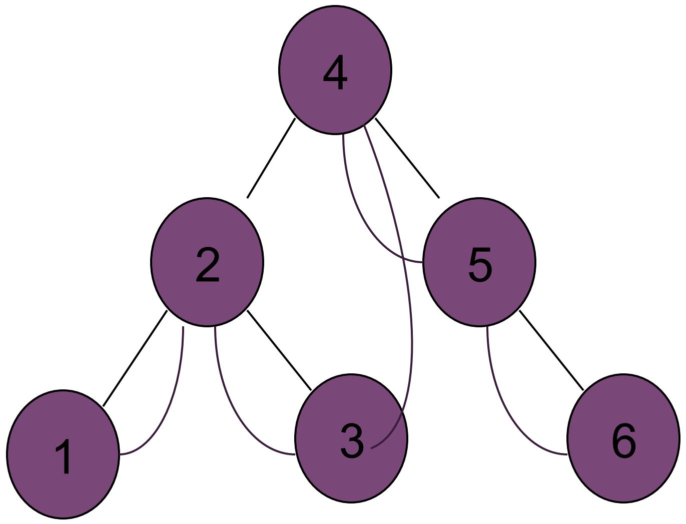
  A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.“

- A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal.

- It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack.

- This can be useful however where stack space is limited.

+ Example



In order Traversal: 1 2 3 4 5 6

+ Classes for Threaded Binary Tree

```
class Node {

  int data;
  Node * left, * right;
  char lflag,rflag;

public:

  Node( int data );
  int getData();
  void setRight( Node * );
  Node * getRight();
  void setLeft( Node * );
  Node * getLeft();
  void setlflag( char );
  char getlflag();
  void setrflag( char );
  char getrflag();

};
```

```
class TBinaryTree {
        Node * root;
public:
        TBinaryTree();
        void insert( int );
        void inOrder();
        void postOrder();
        void preOrder();
        void deleteData( int );
        ~TBinaryTree();
};
```

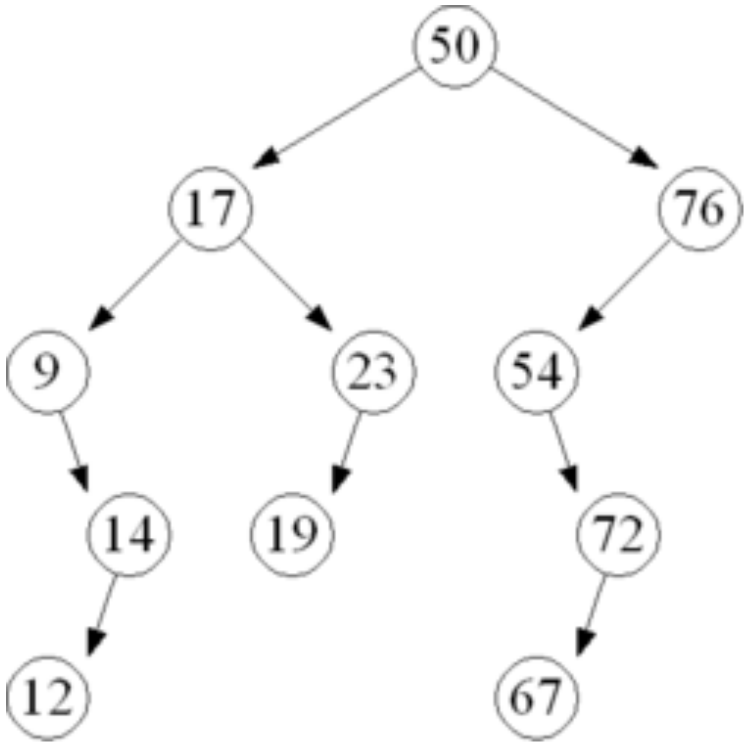
bitCode technologies

+ Introduction to AVL Trees

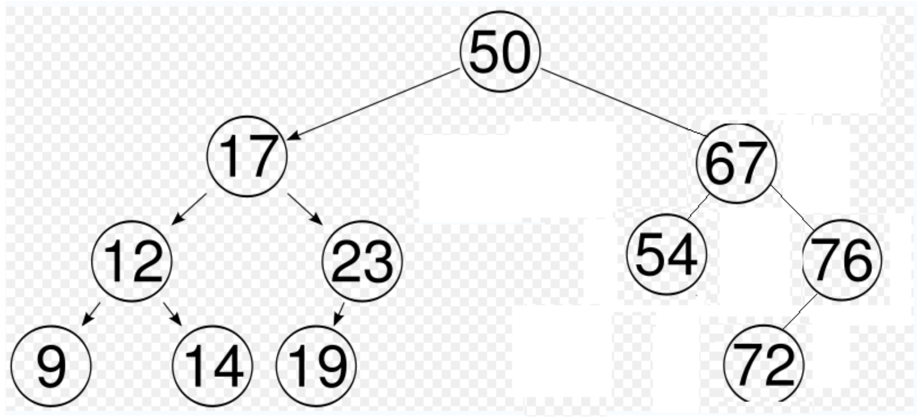
- An **AVL tree** is a self-balancing binary search tree .
- In an AVL tree the heights of the two child sub trees of any node differ by at most one, therefore it is also called height-balanced tree.
- Additions and deletions may require the tree to be rebalanced by one or more tree rotations .
- The **balance factor** of a node is the height of its right sub tree minus the height of its left sub tree.
- A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree.
- The balance factor is either stored directly at each node or computed from the heights of the subtrees.

bitCode technologies

+Example AVL Tree



e.g: Non AVL Tree



e.g. AVL Tree

Insertion into AVL Trees

- Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.
- Retracing is stopped when a node's balance factor becomes 0, 1, or -1.
- If the balance factor becomes 0 then the height of the sub tree hasn't changed because of the insert operation. The insertion is finished.

bitCode technologies

+ Deletion from AVL Tree

- If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node.

- After deletion retrace the path back up the tree to the root, adjusting the balance factors as needed.

bitCode technologies

+ The AVL Tree Rotations **Left Rotation (LL)**

Imagine we have this situation:

```
  a
   \
    b
     \
      c
```

- To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:
  - b becomes the new root.
  - a takes ownership of b's left child as its right child, or in this case, null.
  - b takes ownership of a as its left child.

- The tree now looks like this:

```
   b
  / \
 a   c
```

+ The AVL Tree Rotations **Right Rotation (RR)**

- A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:

```
      c
     /
    b
   /
  a
```

- To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:
  - b becomes the new root.
  - c takes ownership of b's right child, as its left child. In this case, that value is null.
  - b takes ownership of c, as it's right child.

- The resulting tree:

```
    b
   / \
  a   c
```

+ The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:

```
a
 \
  c
```

- It's balanced. Let's insert 'b'.

```
a
 \
  c
 /
b
```

- Our initial reaction here is to do a single left rotation. Let's try that.

```
  c
 /
a
 \
  b
```

bitCode
technologies

+ The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

- This is a result of the right subtree having a negative balance.
- Because the right subtree was left heavy, our rotation was not sufficient.
- The answer is to perform a right rotation on the right subtree. We are not rotating on our current root. We are rotating on our right child.
- Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

```
 Before:        c
               /
              b

   After:         b
                   \
                     c
```

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

- After performing a rotation on our right subtree, we have prepared our root to be rotated left. Here is our tree now:

```
  a
   \
    b
     \
      c
```

left rotation.

```
     b
    / \
   a   c
```

Right-Left Rotiation (RL) or "Double right"

- A double right rotation, or right-left rotation, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy.
- This is a mirror operation of what was illustrated in the section on Left-Right Rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.

```
  c
 /
a
 \
  b
```

- The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem.

```
a
 \
  c
 /
b
```

+ Right-Left Rotation (RL) or "Double right"

- The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem .

- The answer is to make our left subtree left-heavy. We do this by performing a left rotation our left subtree. Doing so leaves us with this situation:

```
     c
    /
   b
  /
 a
```

- This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C. The result:

```
   b
  / \
 a   c
```

bitCode technologies

+ Rotations, When to Use

```
IF tree is right heavy   {

   IF tree's right subtree is left heavy
     Perform Double Left rotation
   }
   else   {
     Perform Single Left rotation
   }

}

Else IF tree is left heavy {
     IF tree's left subtree is right heavy   {
        Perform Double Right rotation
     }
   ELSE   {
     Perform Single Right rotation
   }

}
```

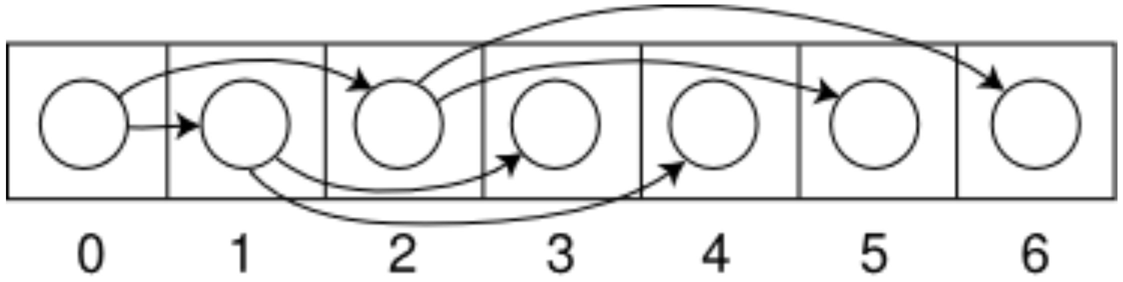
bitCode technologies

+ Binary Tree Representation Using Array

- Binary trees can also be stored as an implicit data structure in arrays, and if the tree is a complete binary tree, this method wastes no space.

- In this compact arrangement, if a node has an index *i*, its children are found at indices 2*i* + 1 and 2*i* + 2, while its parent (if any) is found at index (i-1)/2 (assuming the root has index zero).

- This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal.



- However, it is expensive to grow and wastes space.

bitCode technologies