Trees

Until now we have studied linear data structures. But there are many applications where only the linear data structures are not sufficient. For such situations, we use non-linear data structures like trees and graphs.

In computer science, a tree is an abstract model of a hierarchical structure.

- It is used to represent a hierarchical relationship existing among several data items.
- A tree consists of nodes with a parent-child relationship.
- It is a non-linear data structure.
- Searching as fast as in ordered array.
- Insertion and deletion as fast as in linked list

Definition of a Tree

It is a finite set of one or more data items (nodes) such that

- There is a special data item called the root of the tree
- Its remaining data items are partitioned into number of disjoint subsets, each of which is itself a tree again, called subtrees.

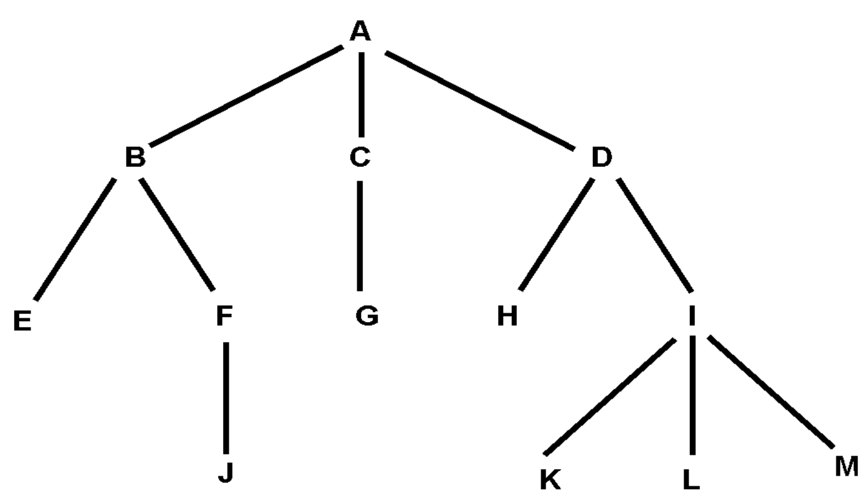


fig: a tree structure

Basic Terminologies in Tree

- **Root: Root** is a special data item, which is the first in the hierarchy. In the above tree,' **A**' is the root item

- **Node:** Each item in a tree is called a **node**. ***A, B, F, G*** etc. are examples of node.

- **Parent:** A node that points to (one or more) other nodes is the **parent** of those nodes while the nodes pointed to are the **children.** B is parent of E, F and E,F are children of B

- **Leaf (External Nodes):** Nodes with no children are **leaf** nodes or **external** nodes or **terminal** nodes. In the above tree E, J, G, H, K, L, and M are terminal nodes

- **Internal Nodes:** Nodes with children are **internal nodes** or **non-terminal nodes**. Note that the root node is also an internal node. Node such as **B, C, D**, are non-terminal nodes

- **Siblings (Brother):** Nodes that have the same parent are **siblings**.

  In the above tree:
  - **E** and **F** are siblings of parent node **B**
  - **K, L**, and **M** are siblings of parent node **I.**

- **Descendents:** The **descendents** of a node consist of its children, and their children, and so on. All nodes in a tree are descendents of the root node (except the root node itself).

- **Ancestors:** The **ancestors** of a node consist of the parent node and the parent of that node also. The root node is the ancestor of all nodes and the root does not have any ancestor.

- **Order (Degree): Order (degree) of a node** is given by the number of its children. If a node has two children, its degree is two.

  In the above tree.
  - degree of node A is 3.
  - degree of node B is 2.
  - degree of node C is 1.

  Similarly, **order (degree) of a tree** is given by the maximum of the degrees of its nodes. In other words, if the degree of a tree is n, its nodes can have, at most, n children. In the above tree the node A has degree 3 which is the maximum degree. **So the degree of the above tree is 3**.

- **Edge:** The link between any two nodes is called an **edge**. It is directed from the parent towards its child.

- **Path:** There is a **single, unique path** from the root to any node. For e.g. the path from node A to J is A, B, F, and J.

- **Height:** A **height of a node** is equal to the maximum path length **from that node to a leaf node**. A leaf node has a height of 0. The **height of a tree** is equal to the height of the **root**. For e.g. the height of node B from leaf node J is 2.

- **Depth:** A **depth of a node** is equal to the maximum path length from root to that node. A root has a depth of 0. **The depth of a tree** is equal to the depth of the deepest node in the tree. It, of course, is equal the tree height. For e.g. the depth of node J is 3.

- **Forest:** it is a set of disjoint tress. In a given tree, if you remove its node then it becomes a forest.

- **Level:** The **level of a node** is 0, if it is root. Otherwise, it is one more than its parent. For e.g.

    - Node **C** is at level 1
    - Node **K** is at level 3

Binary Tree

A binary tree is a finite set of data items which is either empty or consists of a single item called the root and the two disjoint binary trees called the **left subtree** and **right subtree.** In a binary tree the maximum degree of any node is **at most two.** That means, there may be a zero degree node (usually an empty tree) or one degree node or two degree node.

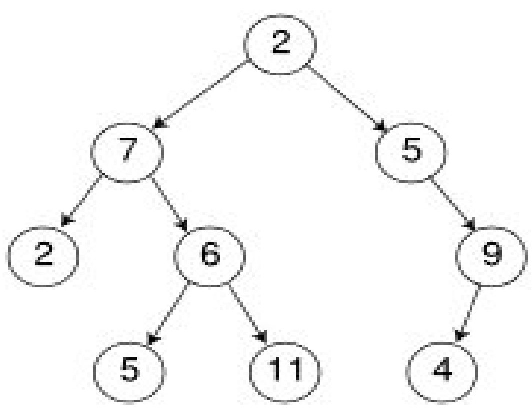


Fig: A simple binary tree, in which every node has at most two children.

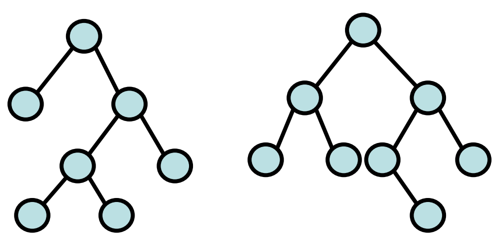
Some properties of a binary tree

- Number of external nodes ⩽ number of internal nodes + 1
- Number of nodes at level i ⩽ 2i
- Number of external nodes ⩽ 2height (Here, height means height of the tree)
- Number of nodes in a tree ⩽ 2height+1-1 (Here, height means height of the tree)
- height ⩾ log2(number of external nodes)
- height ⩾ log2 (number of nodes) — 1

Types of binary trees

There are two types of binary trees.

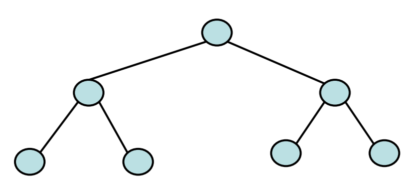
A **Strictly binary tree** or **Full binary tree**, or **Proper binary tree**, is a tree in which every node has zero or two children. i.e. Every internal node has nonempty left and right subtrees. A strictly binary tree with n leaves contains 2n – 1 nodes.



Here the first tree is a Strictly binary tree, while the second one is not.

A **Complete binary tree** or a **Perfect binary tree** is a strictly binary tree in which all leaves are at the same depth. It has following properties.

– Every internal node has 2 children
– Height of n will have 2n+1 – 1 nodes
– Height of n will have 2n leaves
– No of nodes in level l have 2l nodes
– No of external nodes = No of internal nodes + 1



Tree traversals

Visiting the nodes in certain order is called traversing. The reason for visiting the nodes is to process the contents of the node according to our requirement.
In a linear structure like an array or a linked list, there are two ways of traversal
–from front to end and from end to front
These traversals can be achieved, simply, by using loops.

Tree, being a non-linear data structure, there are three different ways for traversal.
Preorder, Inorder and Postorder traversals.

The general rules for these traversals are as follows:

PreOrder traversal (also called depth-first traversal)
In this traversal, the root node is visited prior to the left or right node in a subtree. The rule is

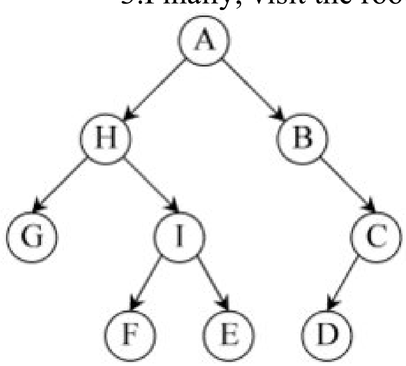
1. Visit the root node (**N**).
2. then, visit the nodes in the left subtree in preorder(**L**).
3. finally, visit the nodes in the right subtree in preorder(**R**).

InOrder traversal
In this traversal, the root node is visited in between the left and the right node in a subtree. The rule is

1. Visit the nodes in the left subtree in inorder (**L**).
2. Visit the root node (**N**).
3. Finally, visit the nodes in the right subtree in inorder(**R**).

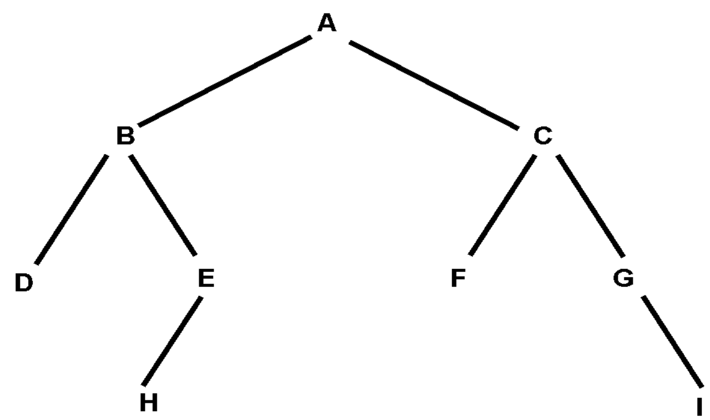
PostOrder traversal (also called breadth first traversal)
In this traversal, the root node is visited after the left and the right node in a subtree. The rule is

1.visit the nodes in the left subtree in postorder(**L**)
2.then, visit the nodes in the right subtree in postorder(**R**).
3.Finally, visit the root node (**N**).



Preorder traversal : A, H, G, I, F, E, B, C, D
Postorder traversal : G, F, E, I, H, D, C,

More examples of tree traversal



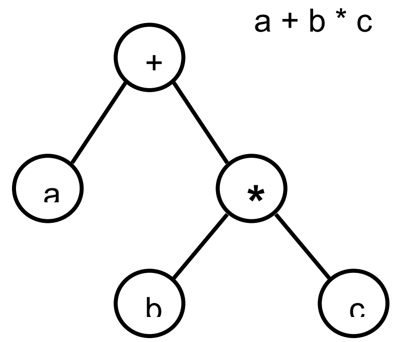
The order of **Preorder traversal** is: *A B D E H C F G I*
The order of **Inorder traversal** is: *D B H E A F C G I*
The order of **Postorder traversal** is *D H E B F I G C A*

Expression Tree

An expression tree is used to represent arithmetic and logical expressions. It is built up from the simple operands and operators of an expression by placing the simple operands as the leaves of a binary tree, and the operators as the interior nodes.
For each binary operator, the left subtree contains all the simple operands and operators in the left operand of the given operator and the right subtree contains everything in the right operand
For a unary operator, one subtree will be empty.

Expression tree traversals

When an expression tree is traversed in inorder, we get an infix expression. When it is traversed in preorder, we get the equivalent prefix expression. Similarly, when it is traversed in postorder, we get the equivalent postfix expression.

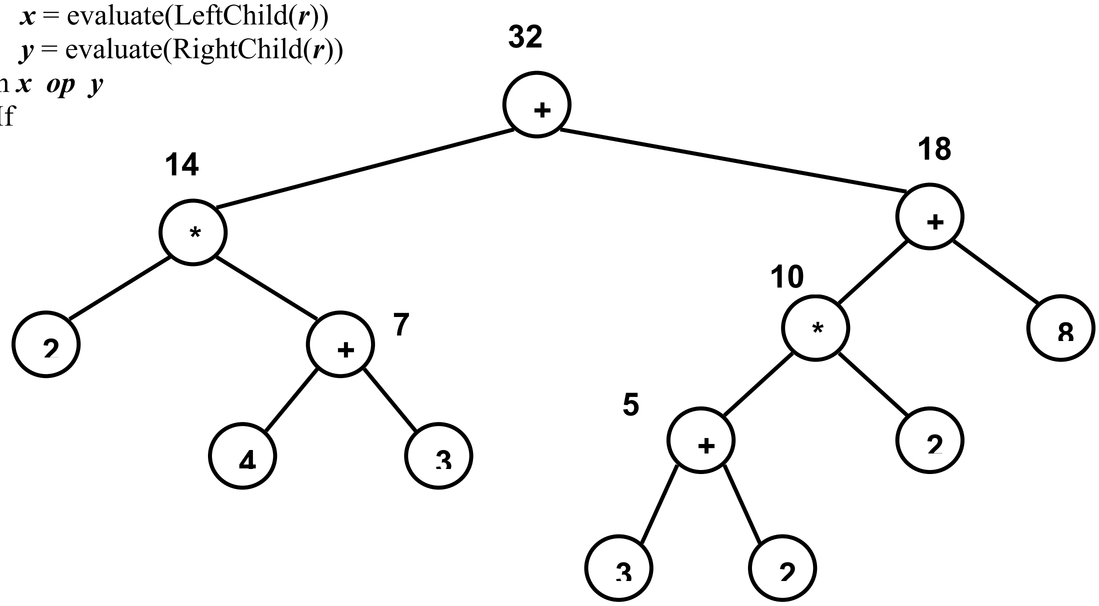


Evaluation of Expression Tree

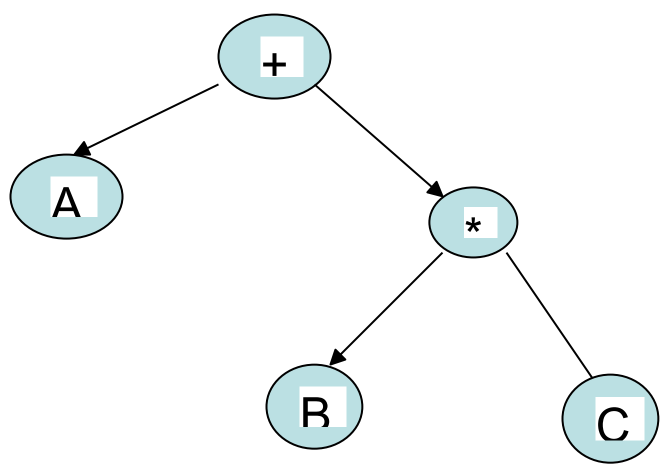
Evaluation of an expression tree is done by applying the operator at the root to the values obtained by recursively evaluating left and right subtrees

An Algorithm

evaluate (***r***)
– If ***r*** is a leaf
    return the value stored in ***r***
– Else
    Let op be the operator stored in ***r***
    r=*leftchild(**r**)*
    x = evaluate(LeftChild(***r***))
    y = evaluate(RightChild(***r***))
  return ***x op y***
– End If



Example of an expression tree from a given expression
1. A + B * C

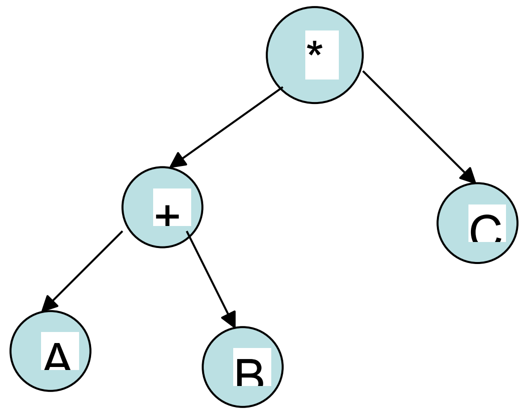


As, already said, When an expression tree is traversed in inorder, we get an infix expression. When it is traversed in preorder, we get the equivalent prefix expression.

Similarly, when it is traversed in postorder, we get the equivalent postfix expression.So, now let us check:

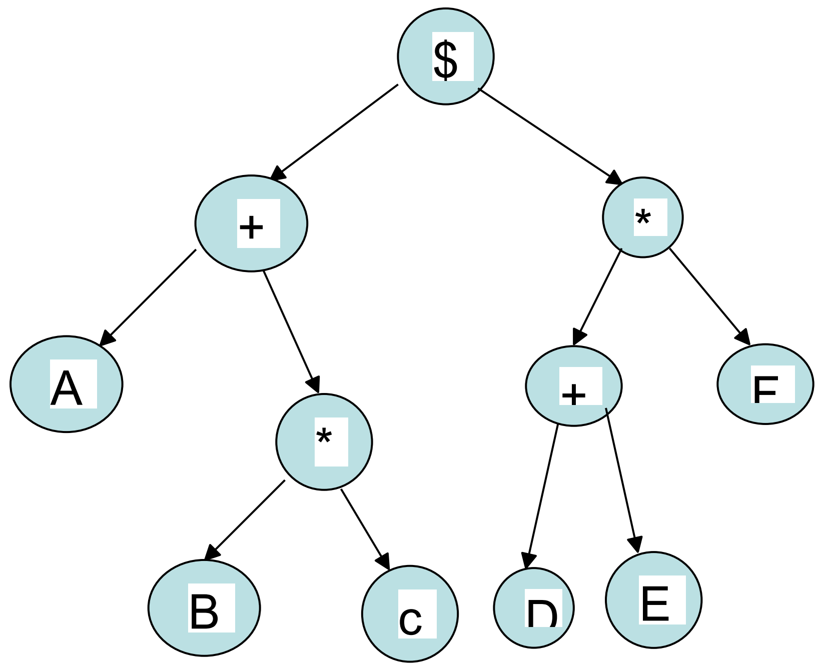
The preorder traversal is: + A * B C (equivalent to prefix expression)
The postorder traversal is: A B C * + (equivalent to postfix expression)

2. **(A + B) * C**



The preorder traversal is: * + A B C (equivalent to prefix expression)
The postorder traversal is: A B + C * (equivalent to postfix expression)

3. (A + B * C) $ ((D + E) * F)



The preorder traversal is: $+A*BC*+DEF (equivalent to prefix expression**)**
The postorder traversal is: ABC*+DE+F*$ (equivalent to postfix expression**)**
Algorithm for construction of an expression tree out of a postfix expression

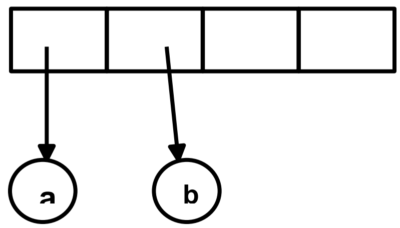
Given a postfix expression, following is the algorithm to build an expression tree. The inorder and preorder traversals of this tree will give infix and prefix expressions respectively.

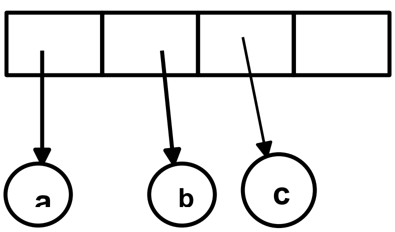
1. Make an empty stack.

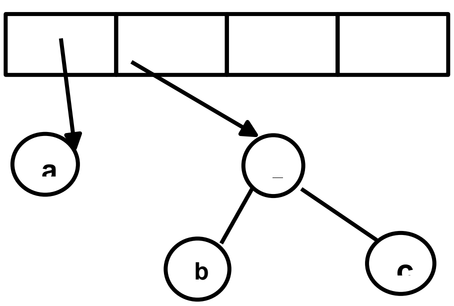
2. For each character from left to right
- if the character is an operand.
  make a single node tree for this operand, push its address into the stack.
  elseif it is an operator, say op
  make a single node tree for this operator
  pop the stack, make it right child of op
  pop the stack, make it left child op
  push the tree (whose root is op) into the stack
- endif

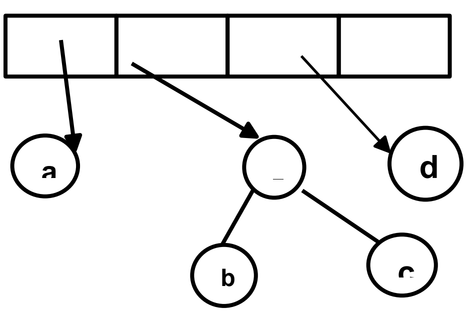
3. Pop the stack to get the required expression tree

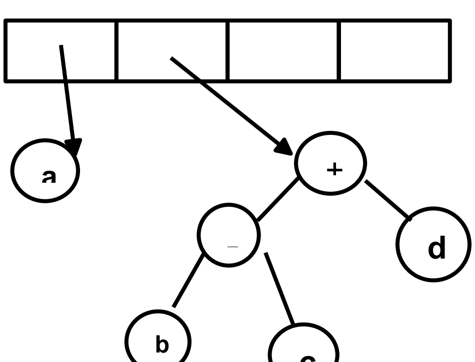
Let the postfix expression be **abc-d+***

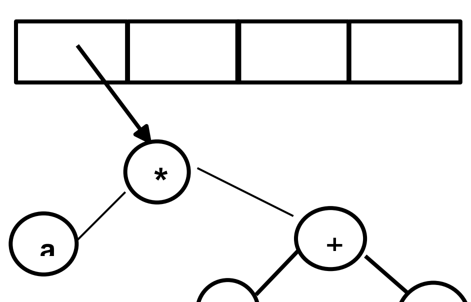












Representation of Binary Tree

A binary tree can be represented in two different ways.
- Using linked list
- Using array

Linked representation and implementation

A binary tree is a collection of nodes. Each node in a binary tree has three fields, the data part, and two pointers that point to the root node of left and right subtree. For a leaf the left and right pointers will be null. Also for an empty subtree, the corresponding pointers will be null. An external pointer is used to point to the root node of the tree.

```
struct bnode
{
        int info;                       //for storing data. Currently it is of type int.
        struct bnode *leftchild;  //pointer to the root of the left subtree
        struct bnode *rightchild; //pointer to the root of the right subtree
};
```

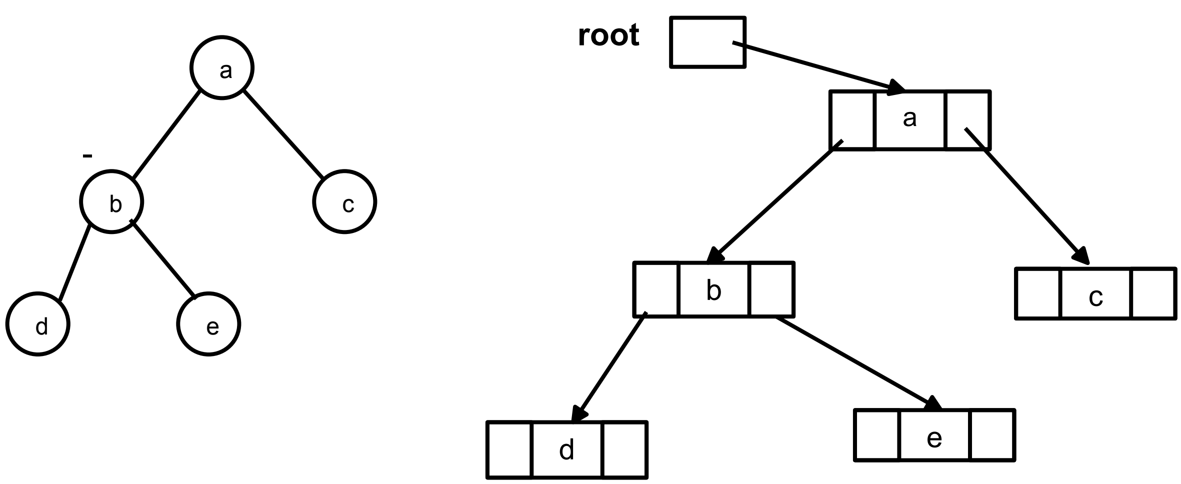


Fig: A binary tree and its linked list

Array Representation of Binary Tree

A Binary Tree can be represented by means of a linear array having index 'k', 0< k<= n, where n is the number of nodes.

The root node is stored at the index 0.
Parent(k) : The Parent of the node at index k is given by floor((k-1)/2) if k is not equal to 0

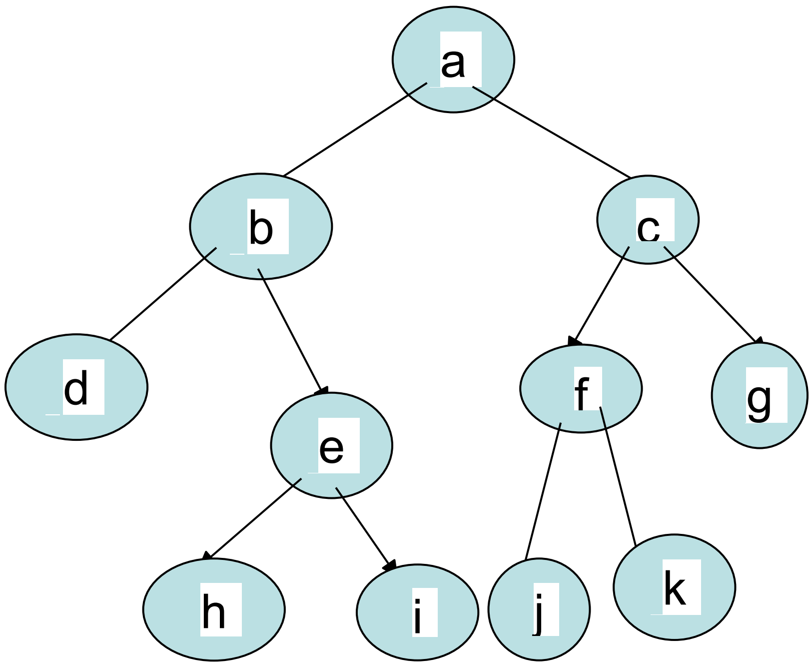
LeftChild(k) : Left child of the node at k is given by 2k+1.

RightChild(k) : Right child of the node at k is given by 2k+2.

Siblings(k) : If the left child at index k is given, then its right sibling is at k+1. Similarly, if right child at index k is given, its left child is at k-1.

Some Binary Trees along with their sequential representation are as follows:

0	**a**
1	**b**
2	**c**
3	**d**
4	**e**
5	**f**
6	**g**
7	
8	
9	**h**
10	**i**
11	**j**
12	**k**
13	
14	



A sequential representation consumes more space for representation binary trees. But for representing complete binary tree, it proves to be efficient as no space is wasted. But, the amount of spaces has to be determined at first.

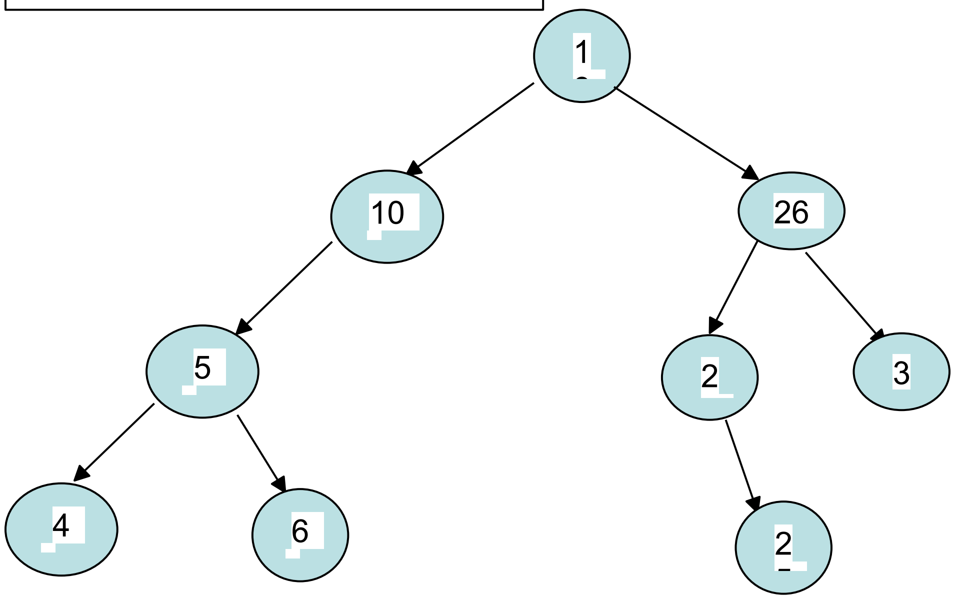
Binary Search Tree (BST)

A Binary Search Tree is a binary tree which is either empty, or in which every node contains a key and satisfies the following criteria:

- All keys (if any) in the left –subtree is smaller than the key in the root.
- All keys (if any) in the right subtree is larger than the key in the root.
- The left and right subtrees again BST.

E.g. 18 10 26 5 20 30 4 6 25

Note: All the keys in the left subtree are smaller than root. All the keys in right subtree are greater than key in the root.



- With the above definition, we can also conclude that, no two nodes in a BST have same value.
- The inorder traversal of a BST yields the keys in a sorted order (ascending).
- If the tree is reasonably full, searching for a key takes **$O(\log_2(n))$** time, where n is the number of nodes.

Operations on a Binary Search Tree

Search(x, T): Search for key x in the BST T.if x is found in some node n of T return a pointer to node n or NULL otherwise.

Insert (x,T): Insert a new node with x in the info part in the tree T such that the property of BST is maintained.

Delete(x,T): Delete a node x in the info part from the tree T such that the property of BST is maintained.

FindMin(T): Returns pointer to a node having minimum key in tree T,NULL if T is empty.

FindMax(T): Returns pointer to a node with maximum key in T,NULL if T is empty.

Searching

In BST, searching of the desired data item can be performed by branching into left or right subtree until the desired data item is found. We proceed from the root node.

- If the tree is empty, then we failed to find x, so return NULL.
- Else, compare x with the key stored in the root of T.
    - If x is same as the key in root, return address of the root.
    - If x is smaller than the key in root, search for x in the left subtree of T recursively.
    - If x is greater than the key in root, search for x in right subtree recursively.

Insert

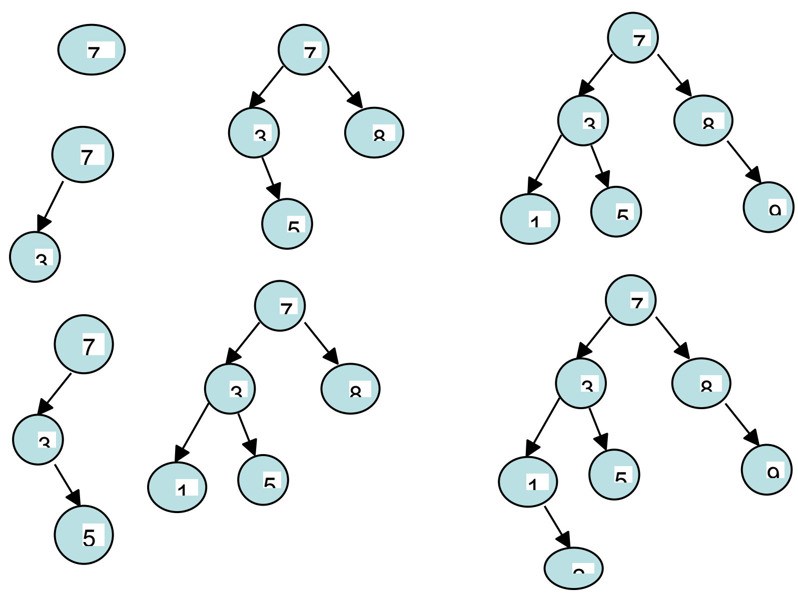
In BST, we don't allow duplicate data items. So, to insert a data item having key 'K' in a binary search tree, we must check that its key is different from those of existing data items by performing a search for data item with the same key K. If the search is unsuccessful, the data item is inserted into BST at the point the search is terminated.

Algorithm

Given a new item with key x, then following algorithm will insert it in BST T.

- If the tree is empty, create a new node n storing the key x and make this new node n the root of the tree. Otherwise, compare x to the key stored at root R.
- If x is identical to key in R, don't change the tree.
- If x is smaller, insert new item into left subtree of the root recursively.
- If x is larger, insert new item into right subtree of the root recursively.

E.g. Trace 7,3,5,8,1,9,2



Now trace 15,10,20,17,25,5,13,33,9,21

Deletion

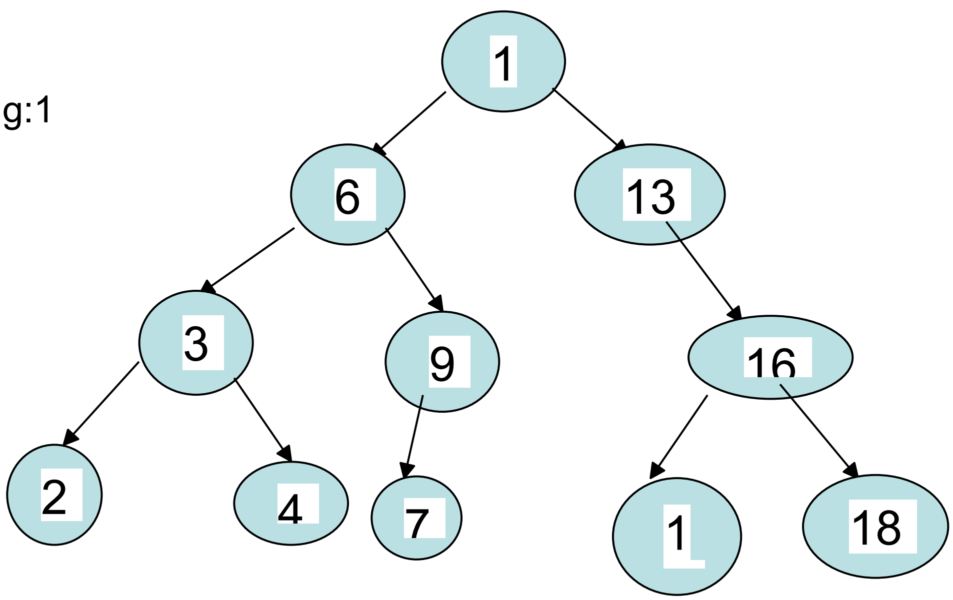
To delete a given node (key), it is searched first. If search is unsuccessful, the algorithm terminates. Otherwise, the following possibilities arise:

- If node is a leaf, it can be deleted immediately.
- If the node has one subtree, the node can be deleted by replacing it with the root of the non-empty subtree.

If the node has two subtrees, we replace the node with the rightmost node of the left subtree T_r or the leftmost node of the right subtree T_r

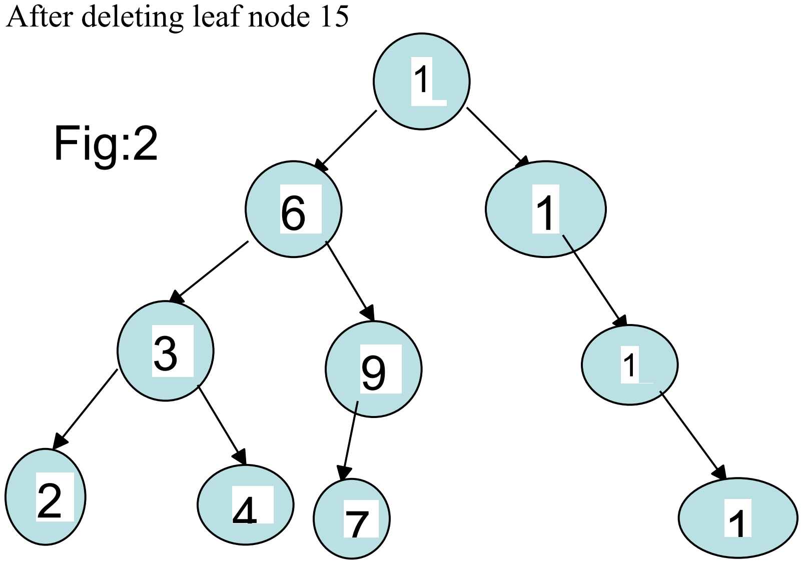
E.g.

Fig:1



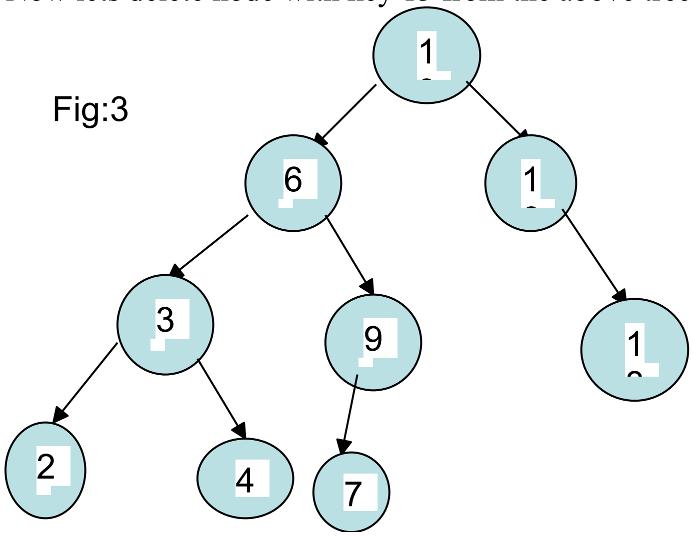
After deleting leaf node 15

Fig:2

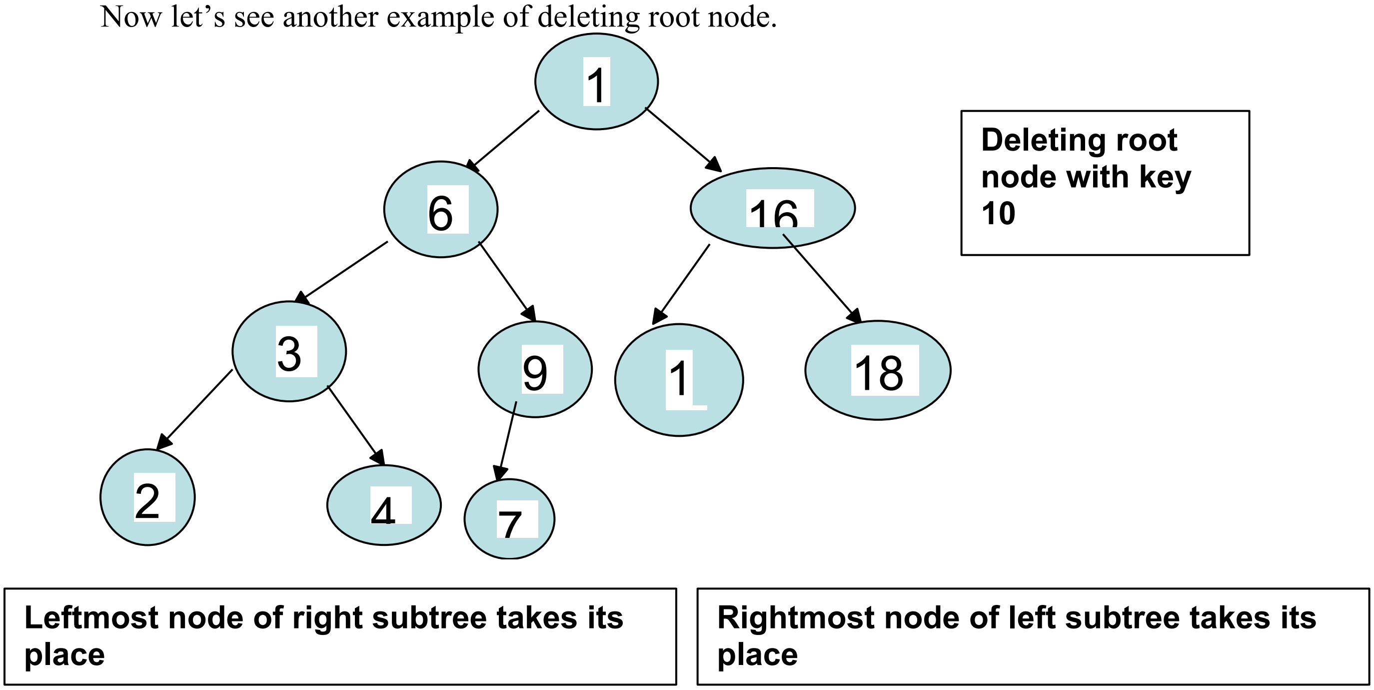


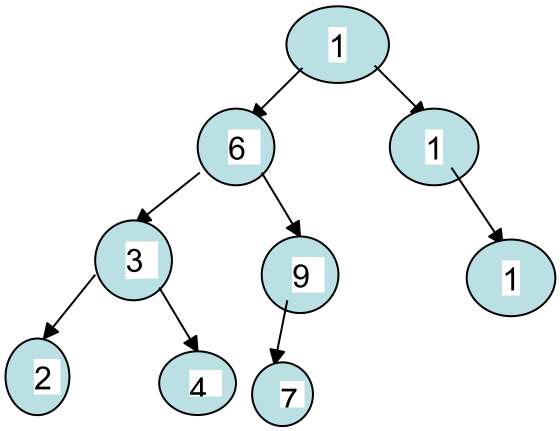
Now lets delete node with key 13 from the above tree.

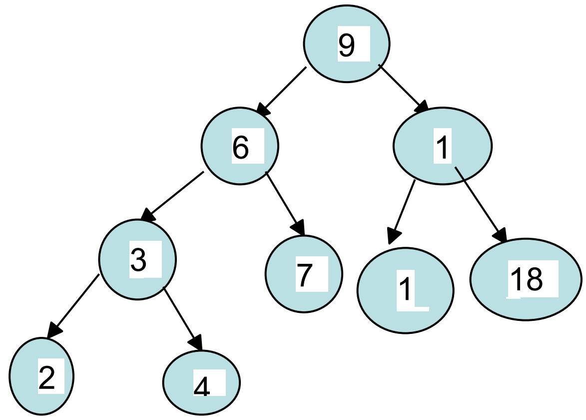
Fig:3



Now let's see another example of deleting root node.



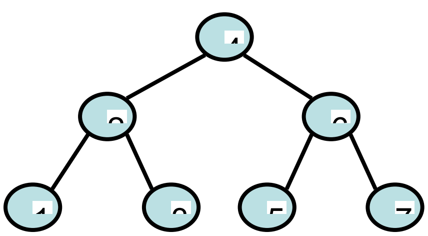


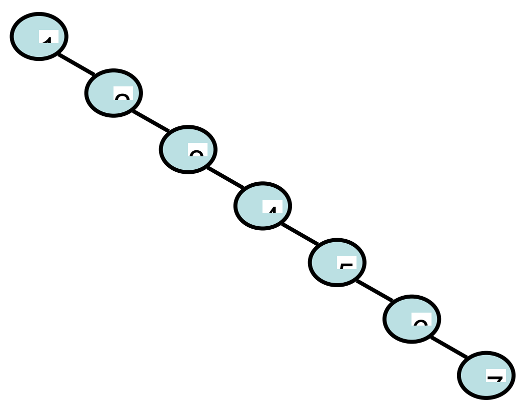


Efficiency of Binary Search Tree

1. Complexity of BST operations is proportional to the length of the path from the root to the node being manipulated, i.e., height of the tree
2. In a well balanced tree, the length of the longest path is roughly log n, where n is the no of nodes in the tree
    a. E.g., 1 million entries → longest path is log21,048,576 = 20
3. For a thin, unbalanced tree, operations become O(n):
    a. E.g., elements are added to tree in sorted order
4. So, Balancing is important to decrease the computational complexity

Example:





In the left tree, the *Search* operation requires 3 comparisons to find 7

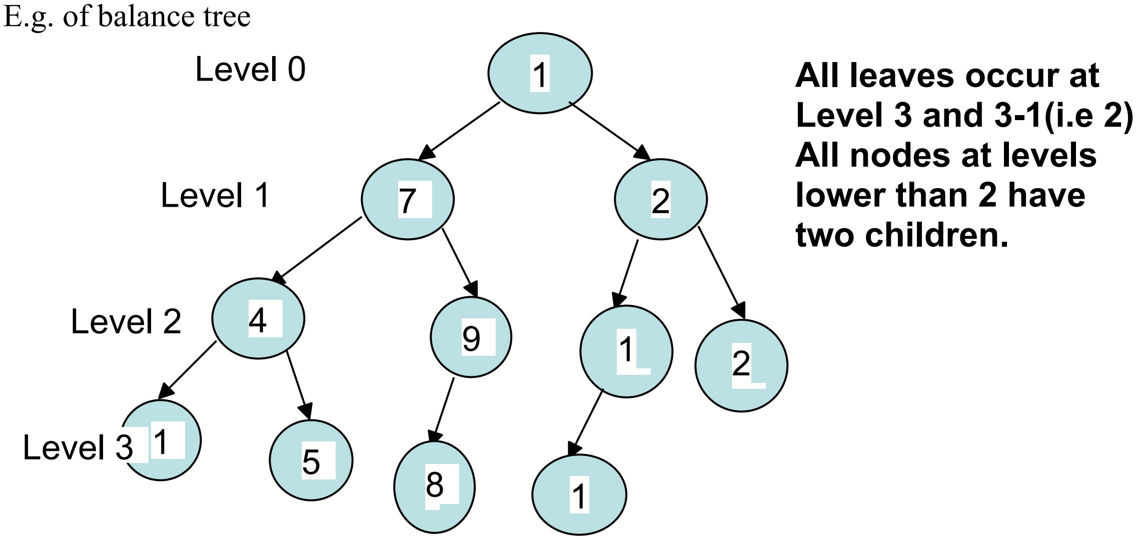
In the right tree, the *Search* operation requires 7 comparisons to find 7

1. We know the **complete binary** tree has the shortest overall path length for any binary tree.

2. The longest path in a complete binary tree with n elements is guaranteed to be no longer than log(n).
3. If we can keep our tree complete, we're set for fast search times but the cost is very high to maintain a complete binary tree.
4. Instead, use **height-balanced binary trees**: for each node, the height difference between the left and right sub trees is at most one.

Height Balanced Trees or AVL Trees

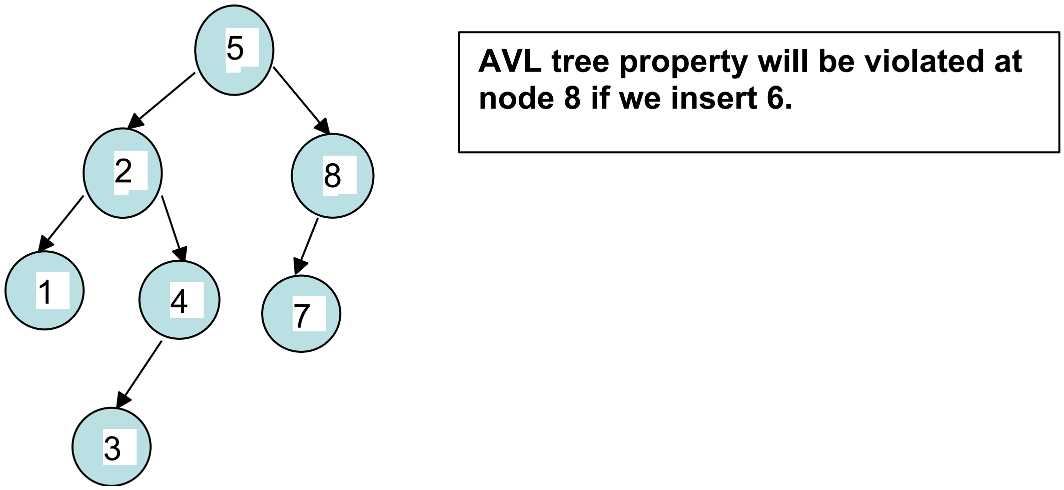
A binary tree of height h is completely balanced if all the leaves occur at level h or h-l, and all nodes at levels lower than h-l have two children.
E.g. of balance tree

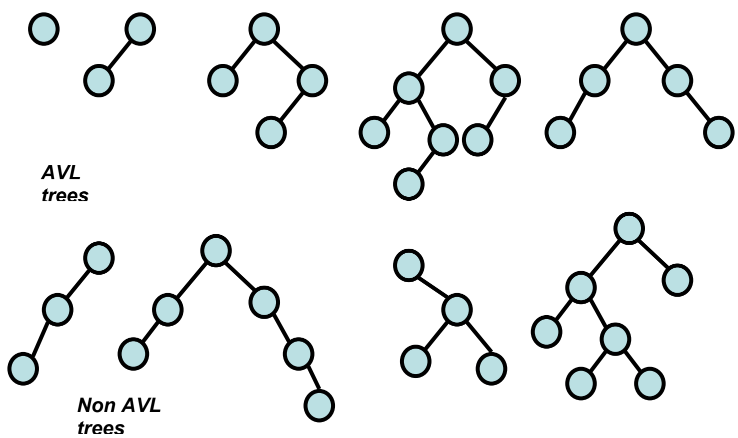


Definition: An AVL tree is a binary search tree in which the heights of the left and right subtree of the root differ by, at most, 1 and in which the left and right subtree are again AVL trees. It is named after their inventors' G.M. Adelson, Velskii and E.M. Landis.

Insertion in AVL Tree

A new node is inserted in an AVL tree by using usual binary search tree algorithm. It, often, is inserted without changing the height of the subtree. Even when the height of the subtree increases, it may be shorter subtree that has grown.
But their may be cases where, a node is added to a subtree that is taller than other subtree. Then, the height difference would be 2.
If this is the case, then the property has to be restored before the insertion step is considered over.
We use 'ROTATIONS' to maintain the AVL Tree properties.
E.g.





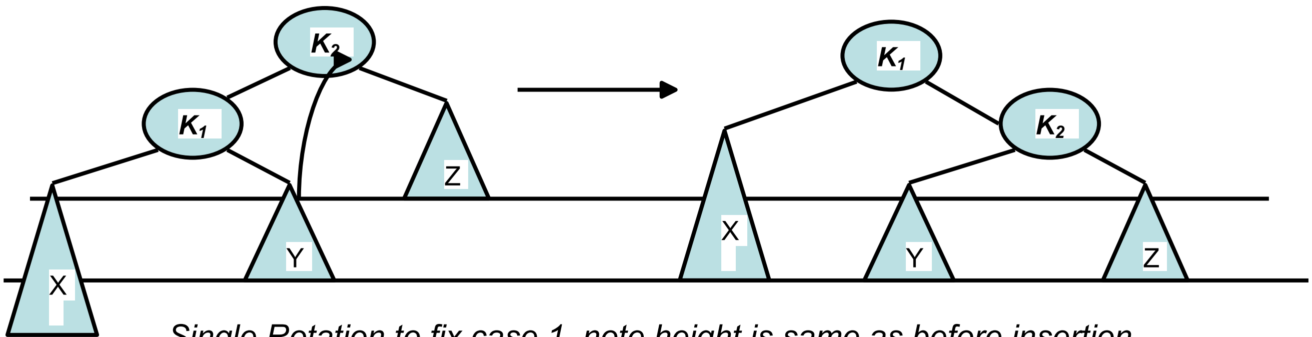
Cases of violation

After insertion, violation might occur in four ways.

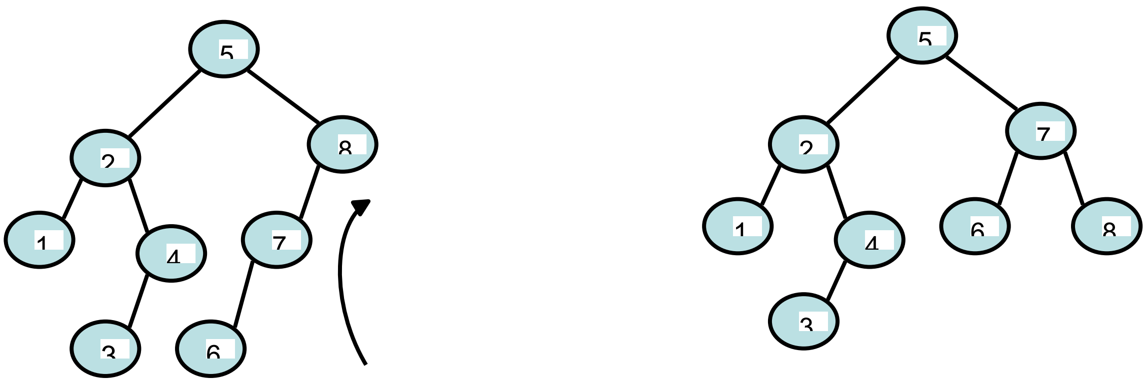
1) Insertion at left subtree of the left child of node n.
2) Insertion at right subtree of the left child of node n.
3) Insertion at left subtree of the right child of node n.
4) Insertion at right subtree of the right child of node n.

The first and forth cases (left-left and right-right) is rebalanced using Single Rotation. The second and third cases (right-left and left-right) are fixed using Double Rotation. This is also known as dogleg pattern.

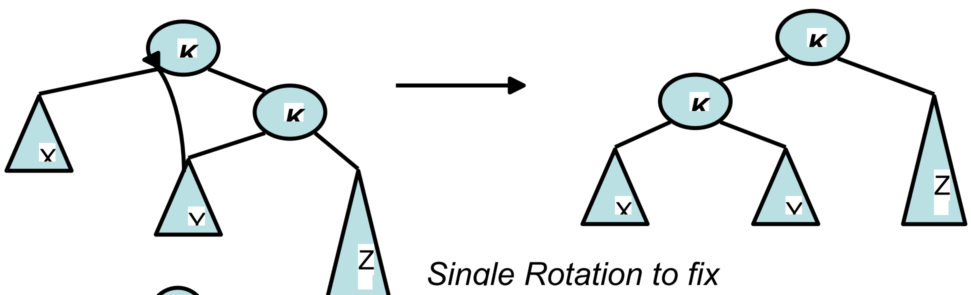
Single Rotation:



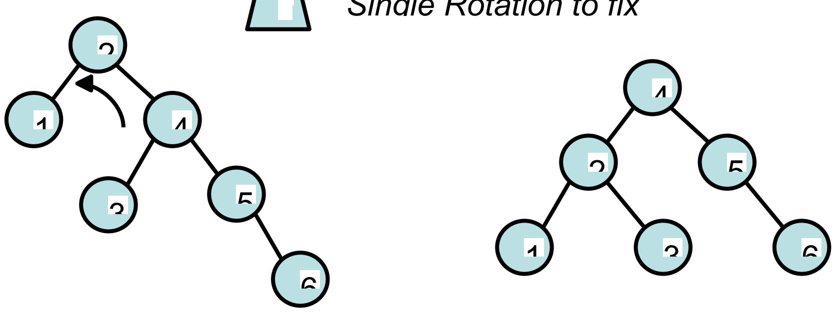
Single Rotation to fix case 1, note height is same as before insertion



AVL property violated by insertion of 6, then fixed by single rotation



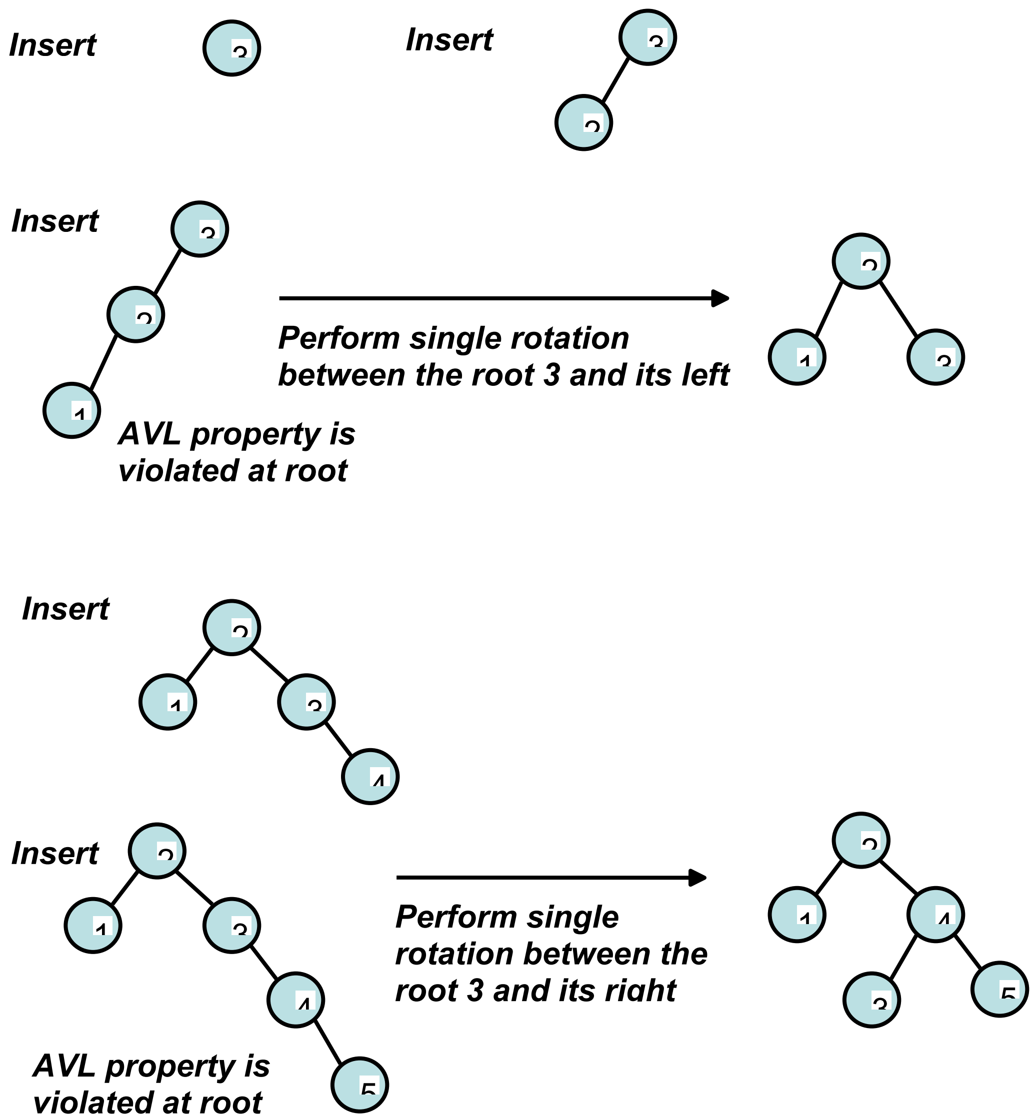
Single Rotation to fix

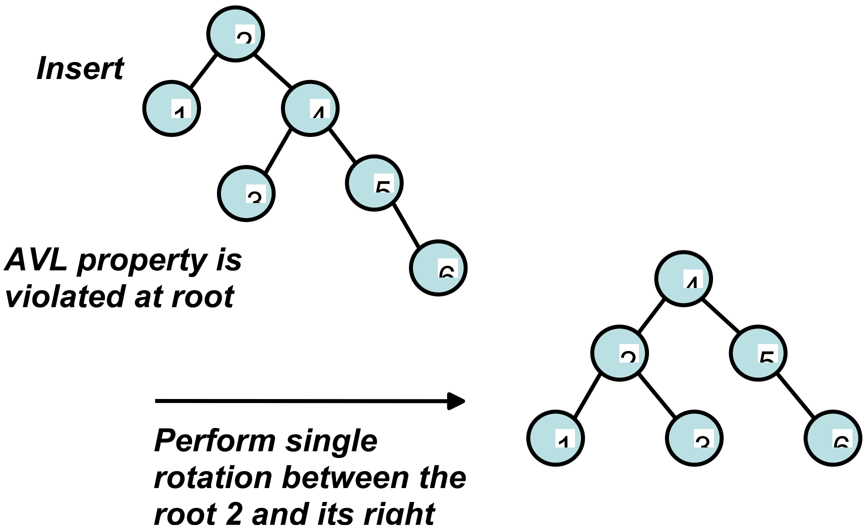


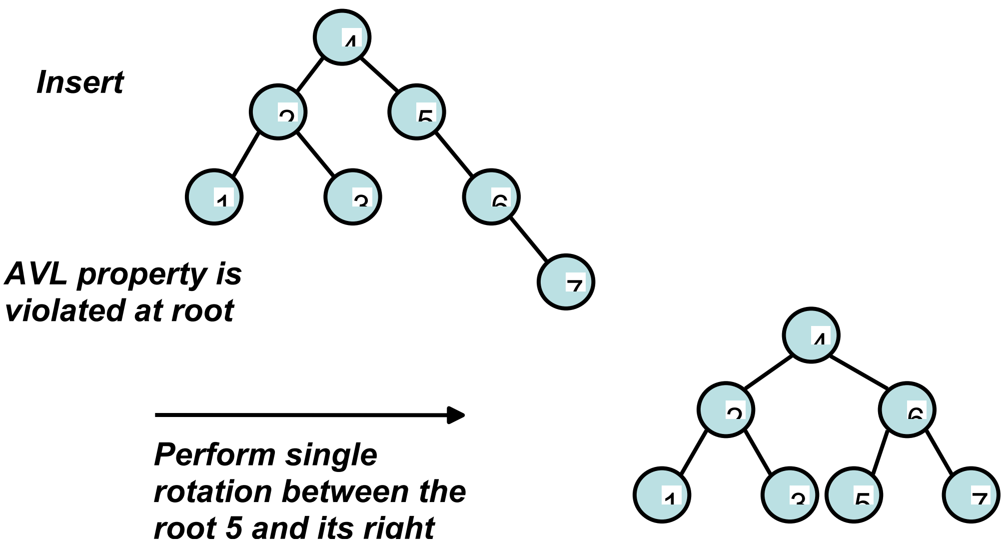
AVL property violated by insertion of 6, then fixed by single

Tracing of Single Rotation:

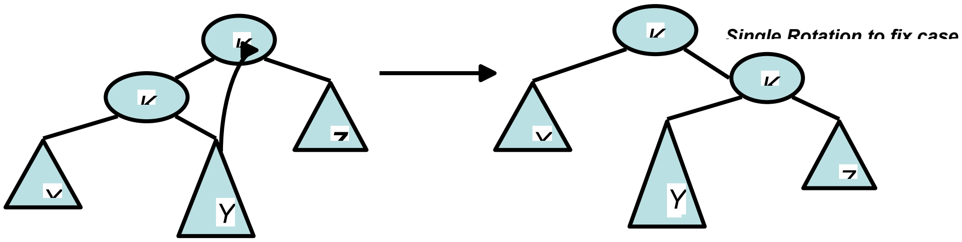
Insert 3, 2, 1, 4, 5, 6, 7 in an empty AVL tree



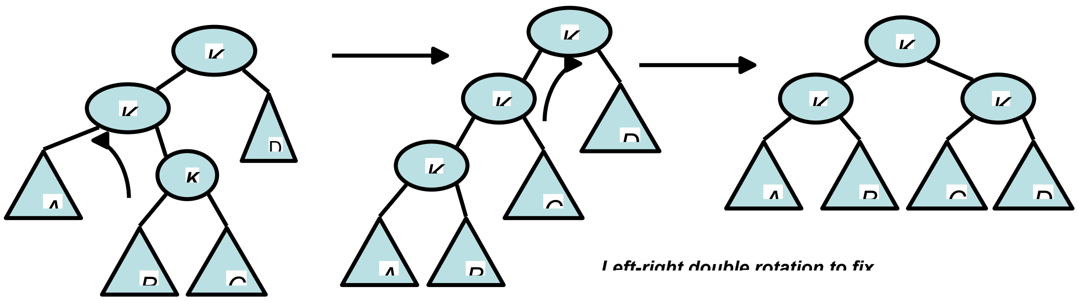




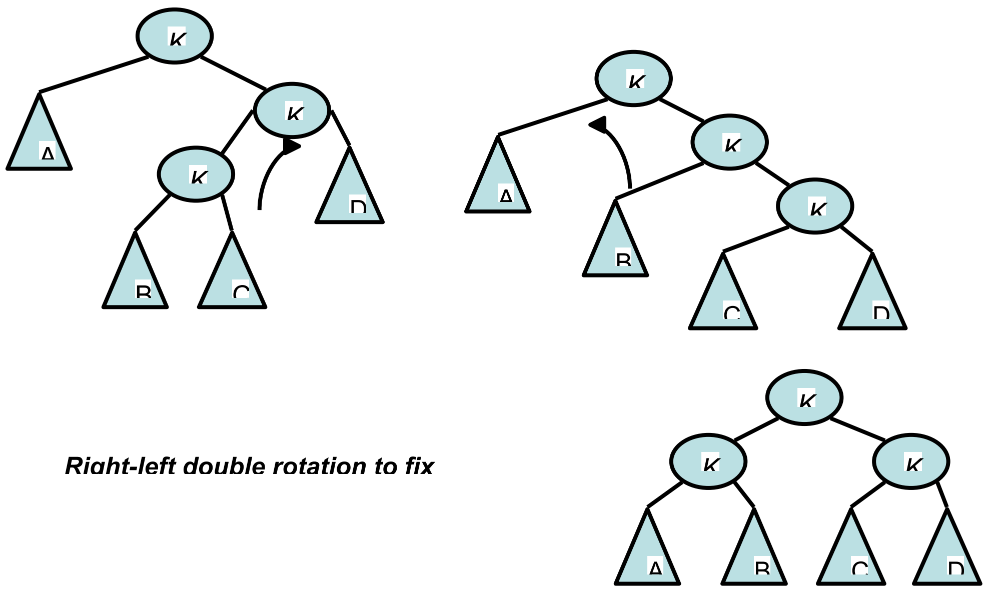
Double Rotation:



Single Rotation to fix case



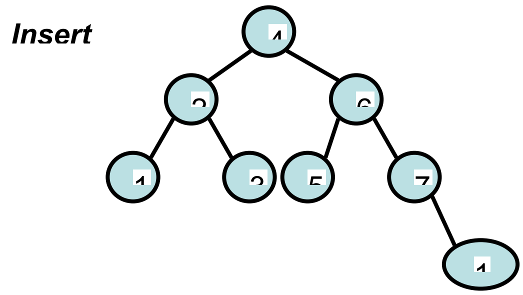
Left-right double rotation to fix

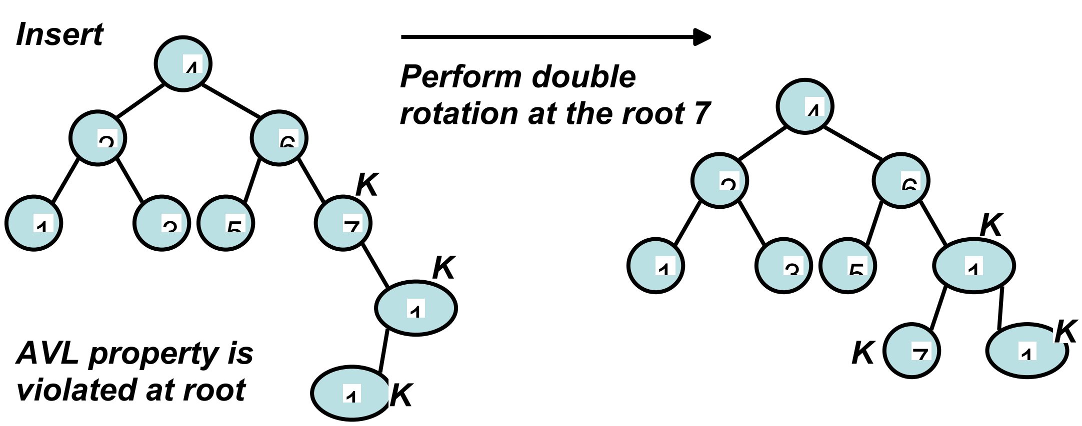


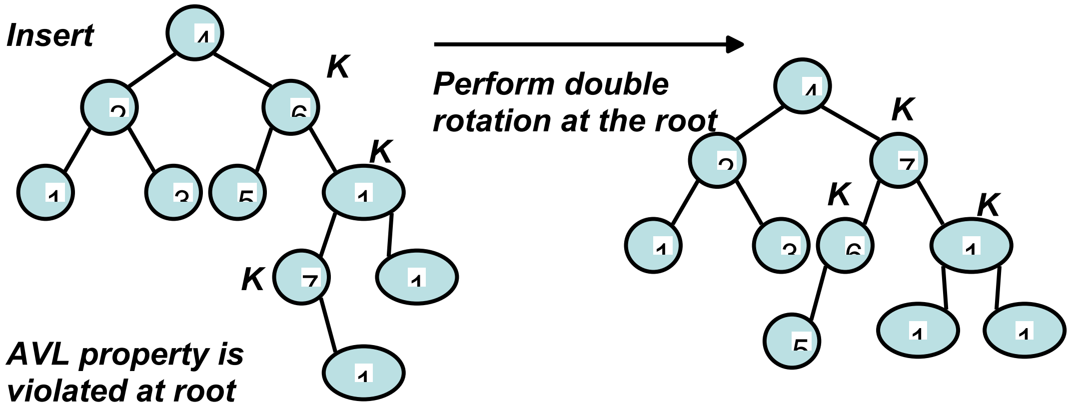
Right-left double rotation to fix

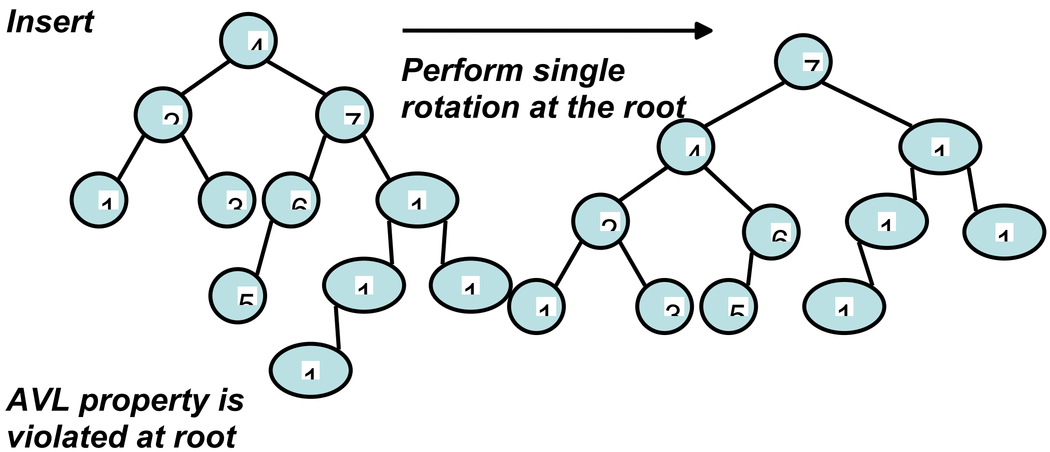
Tracing of Double Rotation:

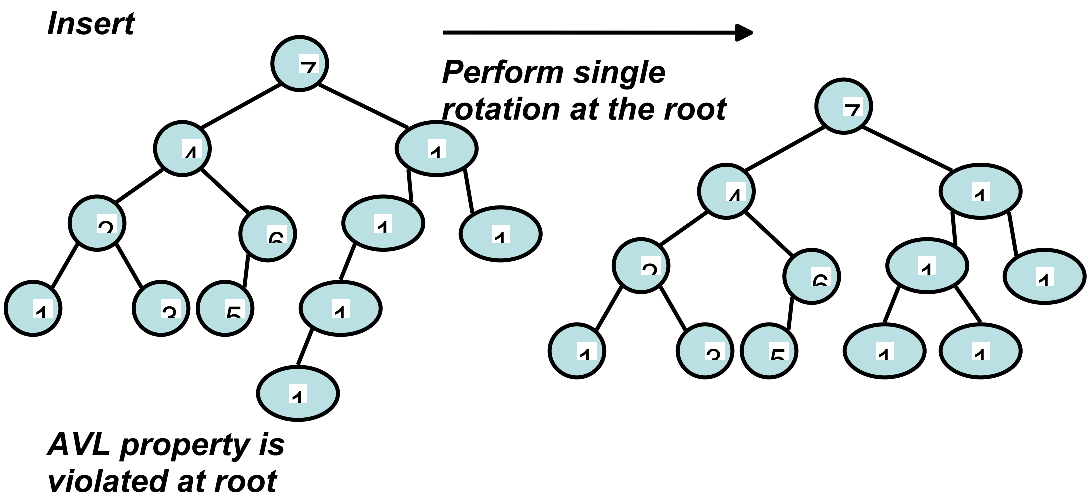
Continue inserting 16,15,14,13,12 in the previous tree






Insert
K
K
K
Perform double
rotation at the root
K
K
K
AVL property is
violated at root


Insert
Perform single
rotation at the root
AVL property is
violated at root



Deletion in AVL Tree:
Like insertions, deletion of nodes in AVL Tree is also similar to deletion in BST. After deletion of nodes, we have to check for imbalance and correct it to rebalance the tree.

Balancing algorithm of Height-balanced (AVL)tree.

1. Insert the new node using the same algorithm as for an ordinary binary search tree.
2. Beginning with new node, calculate the difference in heights of the left or right subtrees of each node on the path leading from the new node back up the tree towards the root(node).
3. If the height balance is within the range -1 to 1 continue their checks until the root node is encountered.
4. If height balance is out of the range then an imbalance is found, perform a rotation of nodes to correct imbalance.
    i. If insertion is at left subtree of the left child of node n or insertion at right subtree of the right child of node n.
        Perform single rotation right or single rotation left respectively.
    ii. Insertion at right subtree of the left child of node n or insertion at left subtree of the right child of node n.
        Perform double rotation left-right or right-left respectively.

Efficiency of AVL Tree:

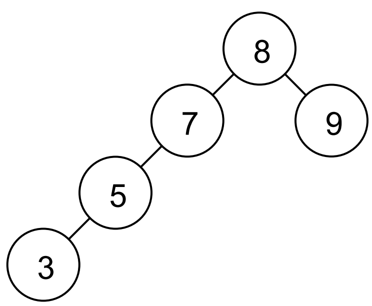
- The maximum height of an AVL tree is 1.44 * log *n*, which is an *O*(log *n*) function
  - This means that in the worst possible case, a lookup in a large AVL tree needs no more than 44% more comparisons than a lookup in a completely balanced tree
- Even in the worst case, then, AVL trees are efficient; they still have *O*(log *n*) lookup times
- On average, for large *n*, AVL trees have lookup times of (log *n*) + 0.25, which is even better than the above worst case figure, though still *O*(log *n*)
- On average, a rotation (single or double) is required in 46.5% of insertions. Only one (single or double) rotation is needed to readjust an AVL tree after an insertion throws it out of balance

Heap

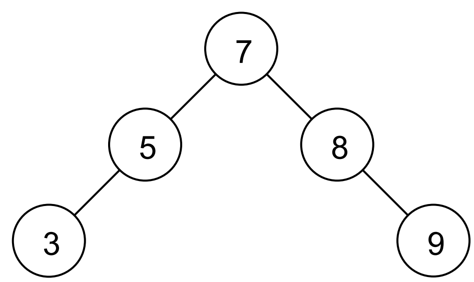
A **binary heap** or simply a heap is an **almost complete binary tree** with the key in each node, such that

- All the leaves of the tree are on two adjacent levels.
- All the leaves on the lowest level occur to the left and all levels except possibly the lowest are filled.
- The key in the root is at least as large as the keys in its children (if any), and the left and rights subtrees are again heaps.

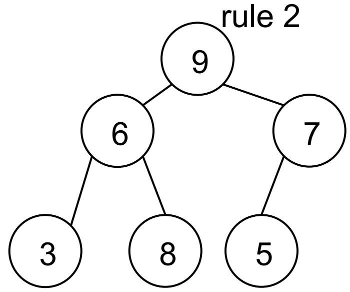
Above definition is of max heap (descending partially ordered tree). For min heap the root has the smallest key.



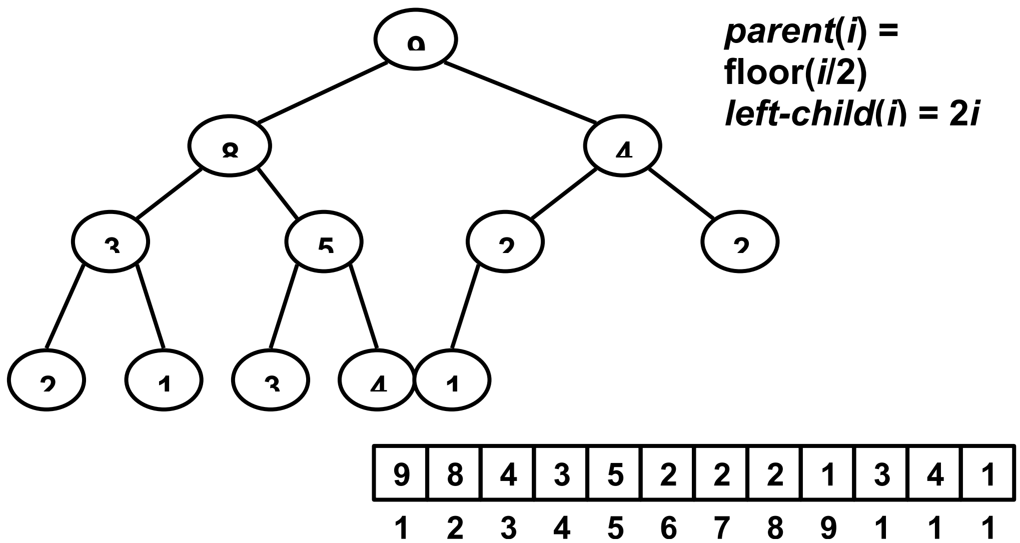
This is not a Heap since it violates rule 1



This is not a Heap since it violates rule 2



This is not a Heap since it violates rule 3



Note that a heap is definitely not a search

Types of Heap:

Max Heap : For max heap, the root has the largest key. The key at the parent node is larger than the keys at their children and descendents.

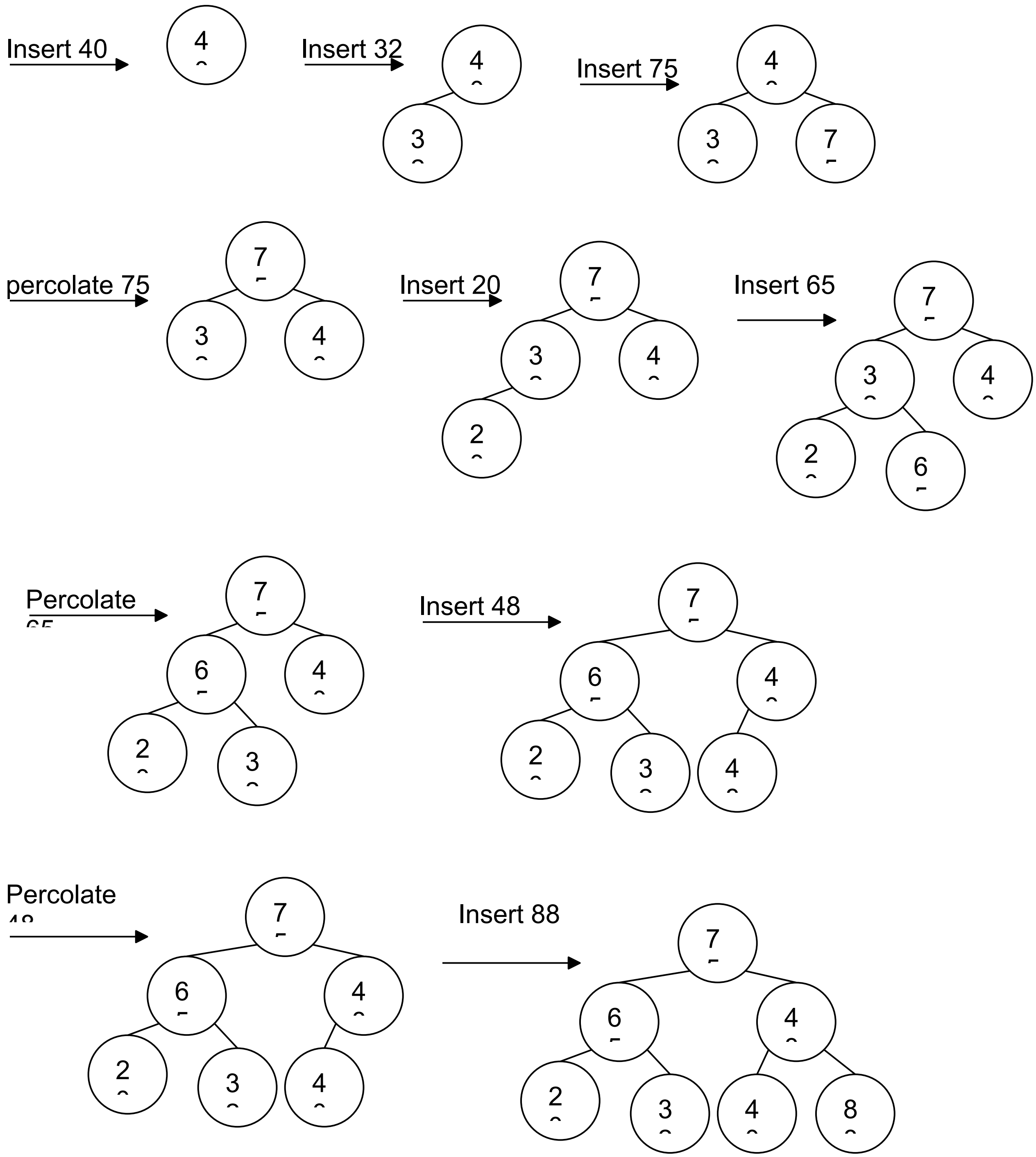
Min Heap : The root has the smallest key. The key at the parent node is smaller than the keys at their children and descendents.

Heaps are represented in array rather than linked structures since it is a special kind of binary tree that have no gaps in an array implementation.

Creating a Heap and Insertion:

A Binary tree with only one node satisfies the properties of a heap.
To insert a new element in to the heap, we create a hole in the next available location, since otherwise the tree will not be complete tree.
If new item can be placed in the hole without violating heap order, then we do insert there. Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up towards the root. The process is continued until the new item can be placed in the hole. This strategy is known as a percolate up. The new element is percolated up the heap until the correct location is formed.

Eg: Inserting following numbers in the max Heap: 40, 32, 75, 20, 65, 48, 88.



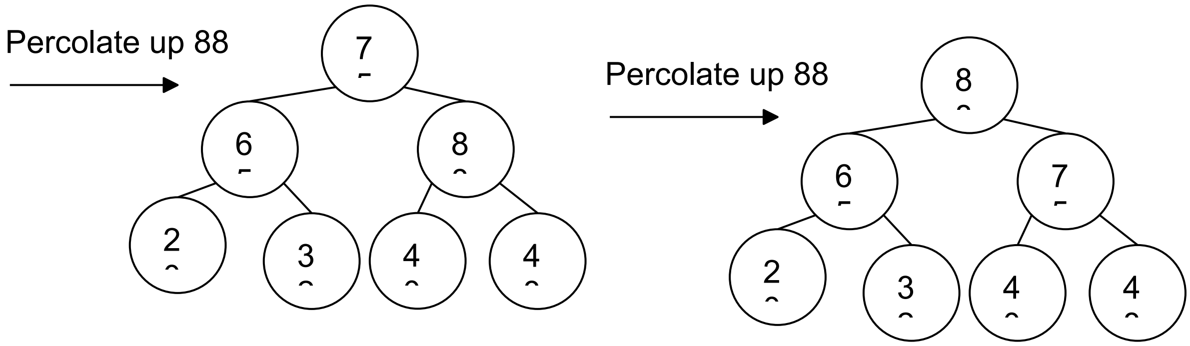


Fig: Creating Max Heap

Deletions in Heap

Deleting in heap is always applied for the item at the root (i.e minimum in min heap and maximum in max heap). When the root item is removed, a hole is created at the root. Since, the heap now becomes one smaller, it follows that the last element ‘x’ in the heap is placed in the hole. If the heap property is violated, we slide the smaller (larger) hole’s children into the hole in the case of min (max) heap. This process of pushing up the smaller item is continued until the item ‘x’ finds its proper place. For deleting in heap lets take above example:

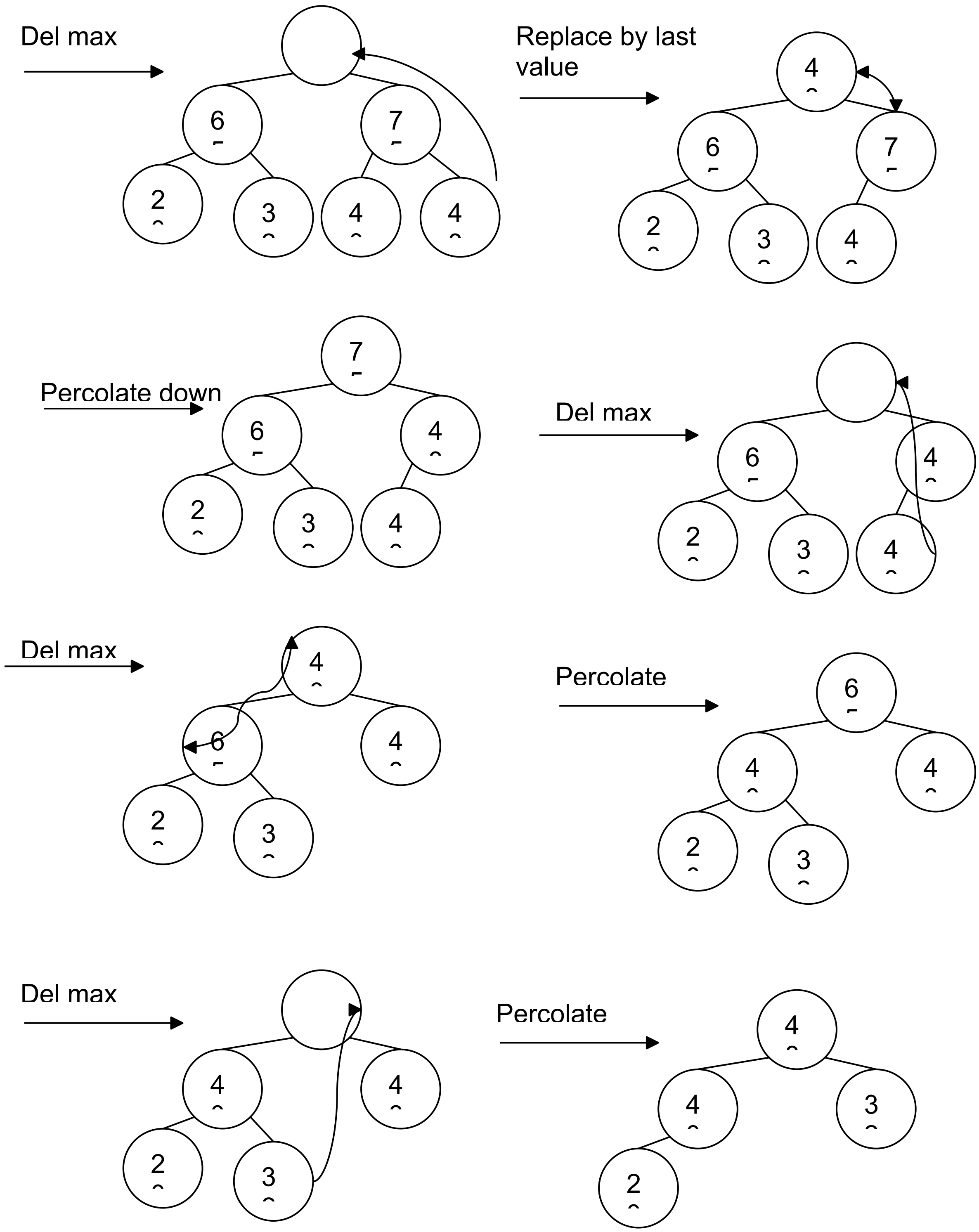


Fig: Deletion in Max Heap

Game Tree

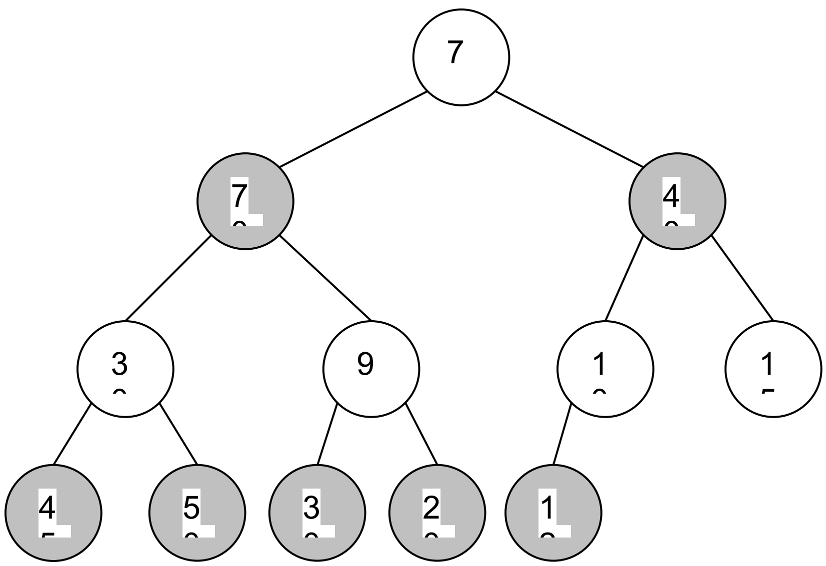
Game tree is most popular algorithm for determining the best move for a player during a game

- Nodes of game tree represents possible game positions
- Children of a node represent positions that can be arrived at via one move from the parent position

For example:

General idea is to increase the game winning possibility for one player

Let's see how the winning strategy is applied.



The tree here is known as min-max tree. Here, we search for best move.

The best move for this time is calculated by analyzing the moves in further levels. For example, in given example, if 7 is our move and our opponent will definitely choose 70, as it can lead up to 50, the maximum value at 4^{th} level. Now our strategy is that, if opponent chooses 70, then do not choose 30, because it will lead to 50. Therefore, choose 9. After level 4, we might get chance of wining the game.

Huffman Algorithm

Lots of compression systems use Huffman Algorithm for the compression of text, images, audio, data etc. The general idea behind the Huffman algorithm is that, use less number of bits to represent mostly used key.
For example:

Let's say we get four kind of signal in a system.
A B C D
For such system, we might use 2-bit representation as;
A 00
B 01
C 10
D 11
And we have data string as,

AAABCAABBCDDAAAABAAABABABADDAAAABC
If the number of A is 10,000, B is 234, C is 45 and D is 134 then the total storage required will be (10,000 + 234 + 45 + 134) * 2 = 20,826 bits to store.

Since, the data 'A' is repeated ten thousand times, if the storage for A can be reduced, then the total space required will automatically be reduced. Therefore, in Huffman algorithm, the data (symbol) that is used most is represented by a single bit.

Now, while assigning the code word, the most frequently used symbol is assigned a code word of length '1', and the final one always gets code word with all ones.

Let's assign code word for the symbols as;
A 0
B 10
C 110
D 111

Now, for the given data string, the binary value will be,

0001011000101011011111110000100001001001001111111000010110

Now, if we calculate the total space required to store the data, it will be:

(10,000 * 1 + 234 * 2 + 45 * 3 + 134 * 3) = 11,005 bits.

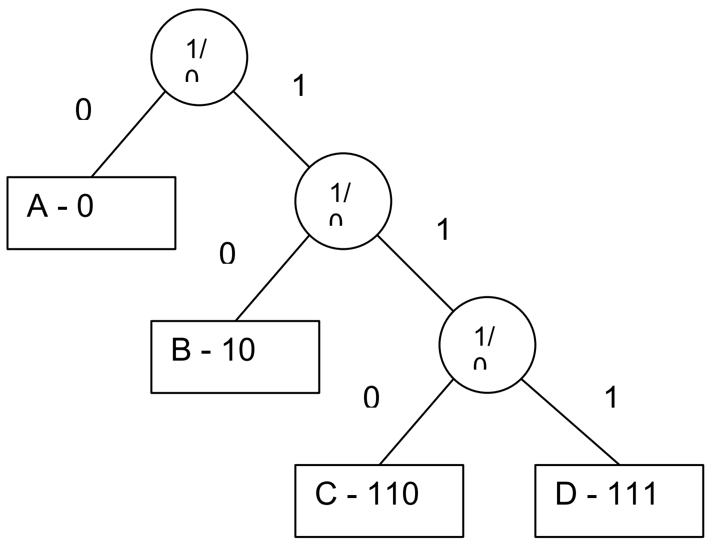
Here, we saved **20,826 – 11,005 = 9,821** bits.
If the data to be stored is very large, then more bits will be saved.

Now the problem is how we recover the original data from compressed string;

0001011000101011011111110000100001001001001111111000010110

Here, a tree is made according to the code word assignment.

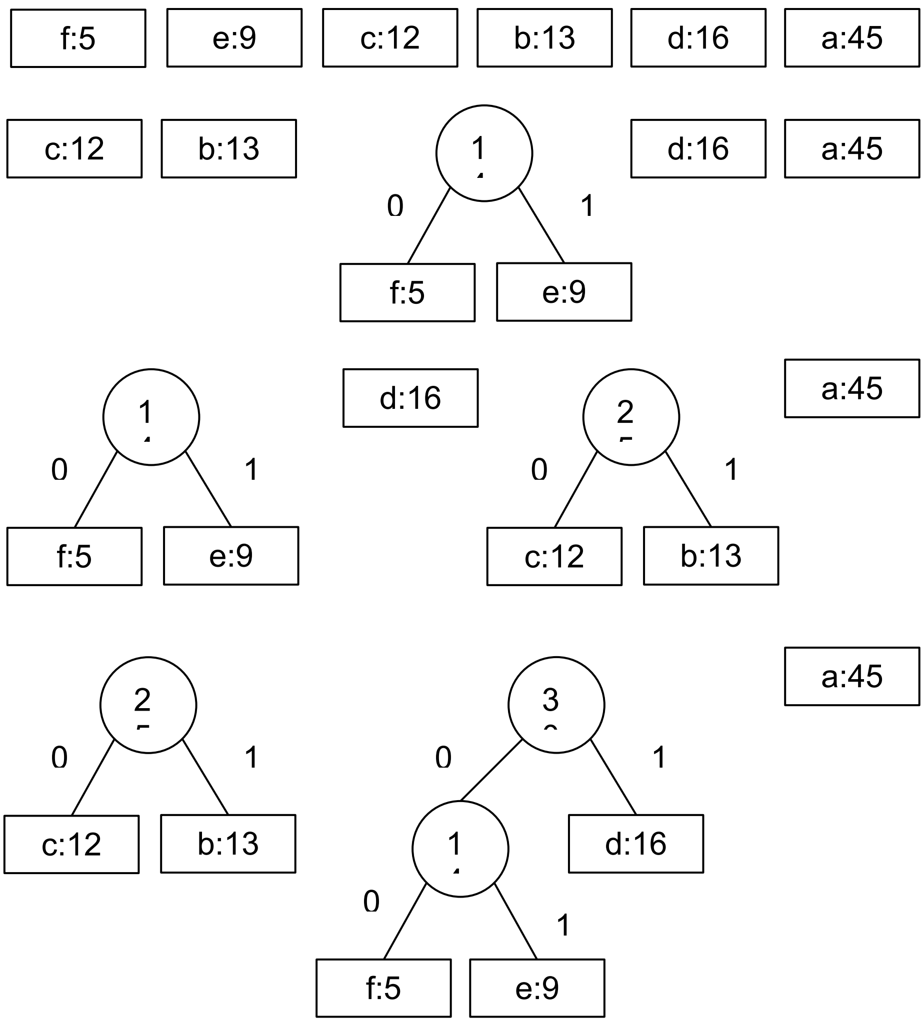
For our example, we have to make a tree like;

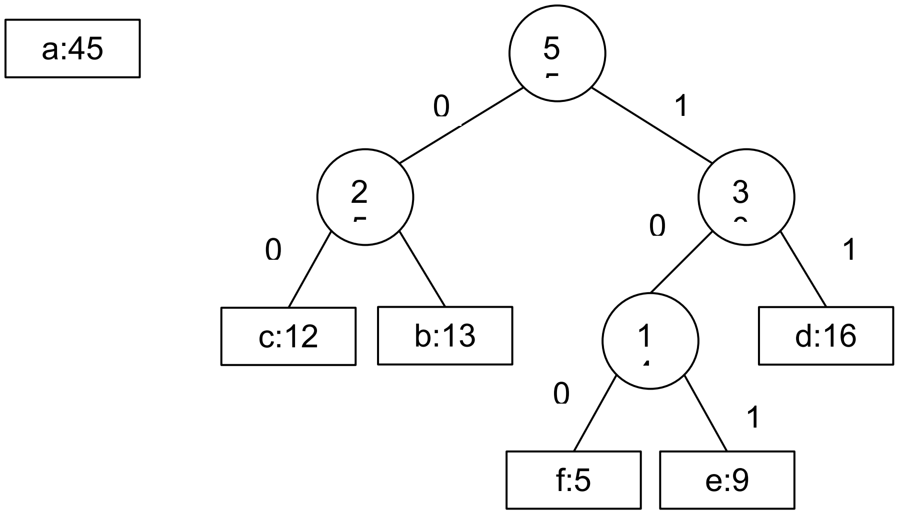


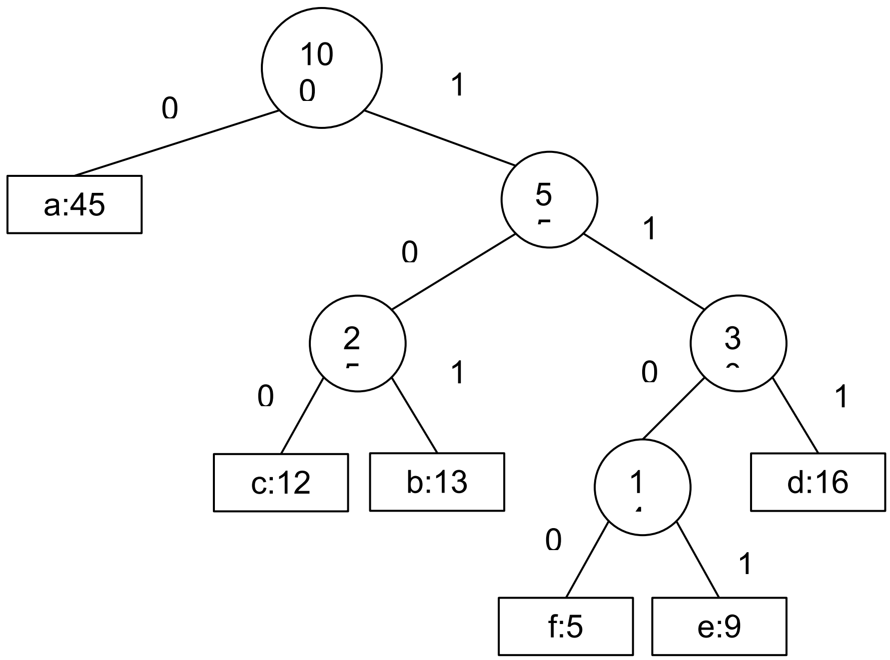
Now, to decode the encoded string, read a bit at a time. Compare with root node. If it is 0, then we are at the end decision point. So, it must be ‘A’. If it was 1, then we have to read the next bit again and check. If the next bit is 0 then it is B. Otherwise, read next bit and check until we do not reach a decision point.

Constructing a Huffman code

	a	b	C	d	e	f
Frequency (in thousands)	**45**	**13**	**12**	**16**	**9**	**5**
Fixed-length codeword	**000**	**001**	**010**	**011**	**100**	**101**
Variable-length codeword	**0**	**101**	**100**	**111**	**1101**	**1100**







Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged.

Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a lift child and 1 if it is an edge to a right child.
The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter.

B-Tree

In binary search tree, each node contains a single key and points to two sub-trees.
We may extend this concept to a general search tree in which each node contains one or more keys.

A multi-way search tree of order n is a general tree in which each node has n or fewer sub-trees and contains one fewer key than it has sub-trees.

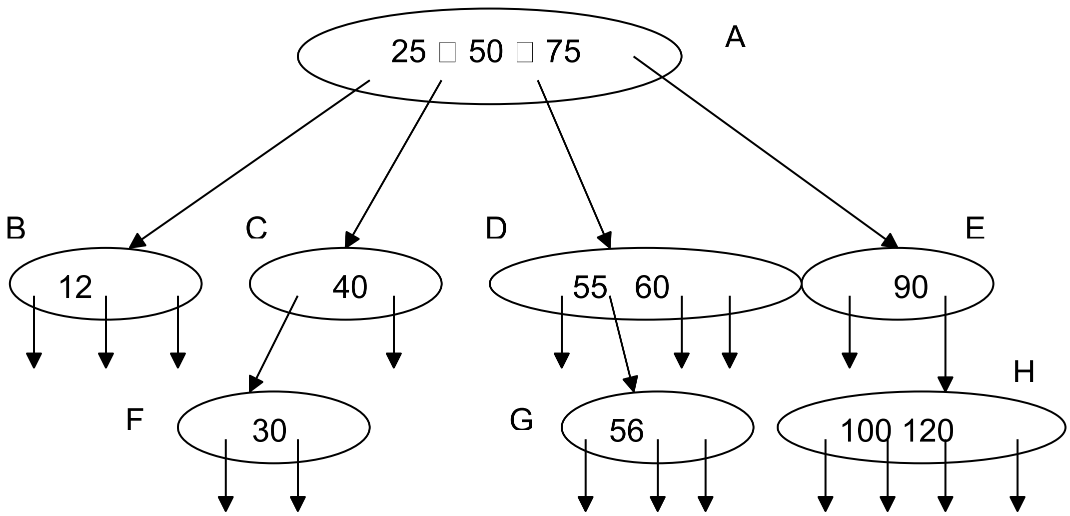


Fig: A multiway search tree of order 4

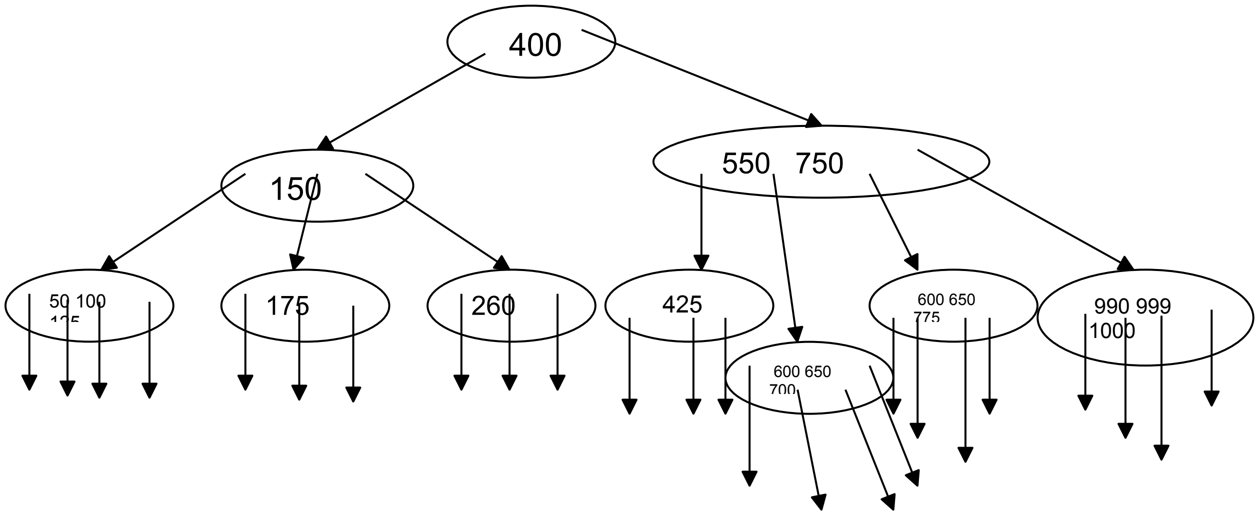
Each node can have at most 3 keys and 4 children, since the order is 4. The nodes A, D and H are called full nodes since it has the maximum no. of keys.

Balanced – multiway Tree
A multiway search tree can be improved to minimize accesses by accomplishing the followings:

1. make the height of the tree as small as possible
2. no empty subtrees appear above the leaves (so that the division of keys into subsets is as efficient as possible).
3. all the leaves be on the same level (so that searches will be guaranteed to terminate with about the same number of accesses).
4. every node (except the leaves) have at least some minimal number of children. These conditions leads us to the definition of B-Tree.

Definition of B-Tree
A B-tree of order m is an m-way search tree in which
1. All leaves are on the same level
2. All internal nodes except the root have at most m nonempty children, and at least (m/2) nonempty children

3. The number of keys in each internal node is one less than the number of its children, and these keys partition the keys in the children in the fashion of a search tree.
4. The root has at most m children, or as few as 2 if it is not a leaf, or none if the tree consists of the root alone



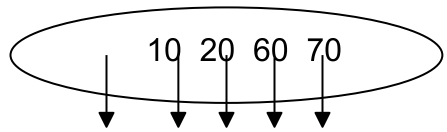
Insertion into a B-Tree
The condition that all the leaves be on the same level forces a characteristic behavior of B-Tree. In contrast to binary search tree, B-Tree are not allowed to grow at their leaves, instead they are forced to grow at the root. The general method of insertion is as follows.

First a search is made to see if the new key is in the tree. This search (if the key is new) will terminate in failure at a leaf. The new key is then added to the leaf node. If the node was not previously full, then the insertion is finished.

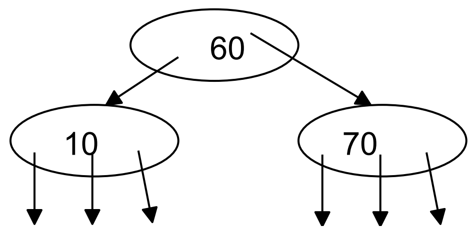
When a key is added to a full node, then the node splits into two nodes on the same level, except that the median key is not put into either of the two new nodes but is instead sent up the tree to b inserted into the parent node.

Illustration: B-Tree of order 5 is as follows:
10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 240

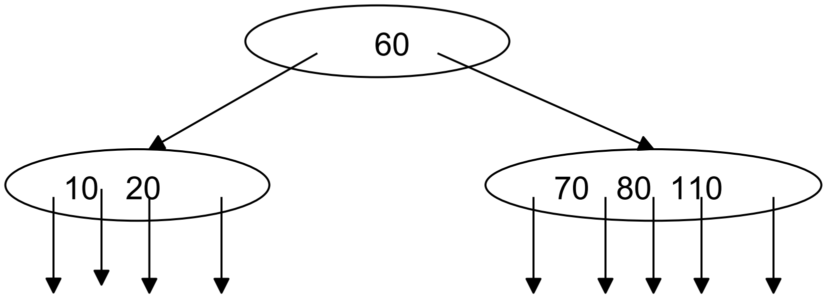
Inserting 10, 70, 60,
20



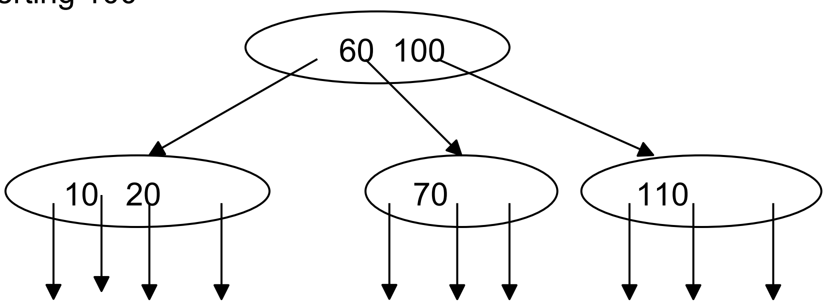
Inserting 110



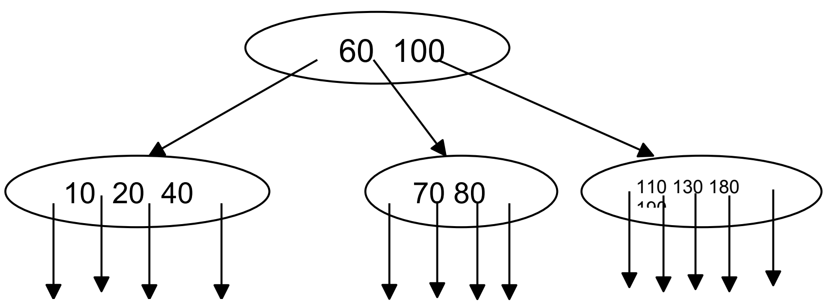
Inserting 40, 80,



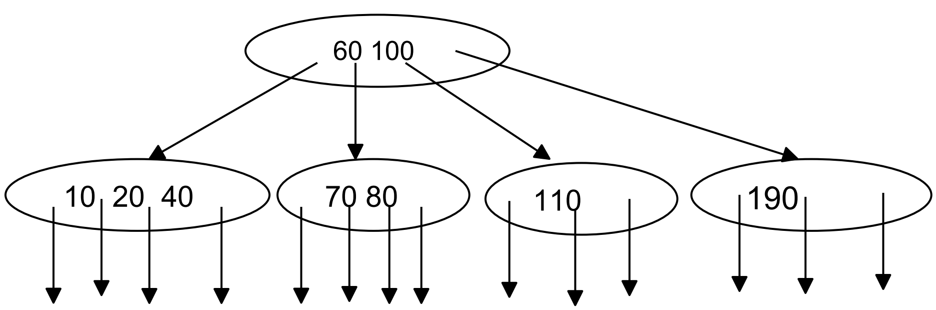
Inserting 100



Inserting 50, 190, 90,



Inserting 240



The first four keys will be inserted into one node as shown in the diagram. They are sorted into proper order as they are inserted. There is no room for fifth key 110, so its insertion causes the node to split into tow, and the medial key, 60 moves up to enter a new node, which is a new root. Since the split nodes are now only half full, the next three keys can be inserted without any difficulty. Note these simple insertions can require rearrangement of the keys within a node. The next insertion 100, again splits a node and this time it is 100 itself that is the median key, so moves up to join 60.

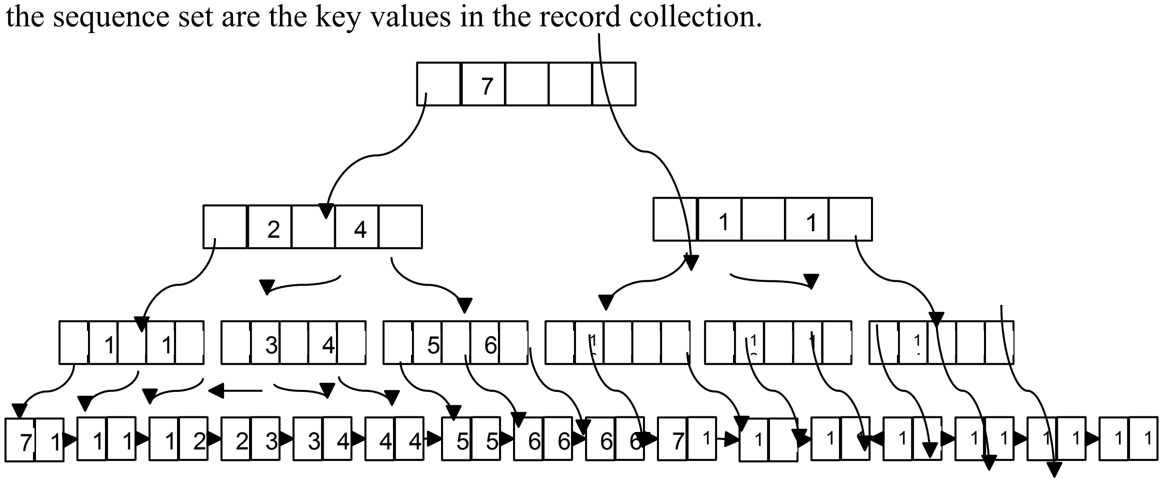
Next several insertions are similar. The final insertion of 240 splits the node containing 110,130, 180 and190 which is again full. Hence, this node in turn splits and 180 is promoted into the root.

Try inserting 30, 120, 140, 200, 210, 160 respectively into the above B-Tree.

B+ Tree

One of the popular techniques for implementing indexed sequential organization is to use a variation on the basic known as B+ tree. In contrast to a B-tree, here all records are stored at the lowest level of the tree; only keys are stored in interior blocks. All the leaves are connected to form a linked list of the keys in sequential order. It has two parts: the index part is the interior nodes; the sequence set is the leaf nodes. The linked leaves are the excellent aspect of B+ tree; the keys can be accessed both directly and sequentially.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context. It is used to provide indexed sequential file organization, the key value in the sequence set are the key values in the record collection.



The ReiserFS filesystem (for Unix and Linux), XFS filesystem (for IRIX and Linux), JFS2 filesystem (for AIX, OS/2 and Linux) and NTFS filesystem (for Microsoft Windows) all use this type of tree for block indexing. Relational databases also often use this type of tree for table indices.