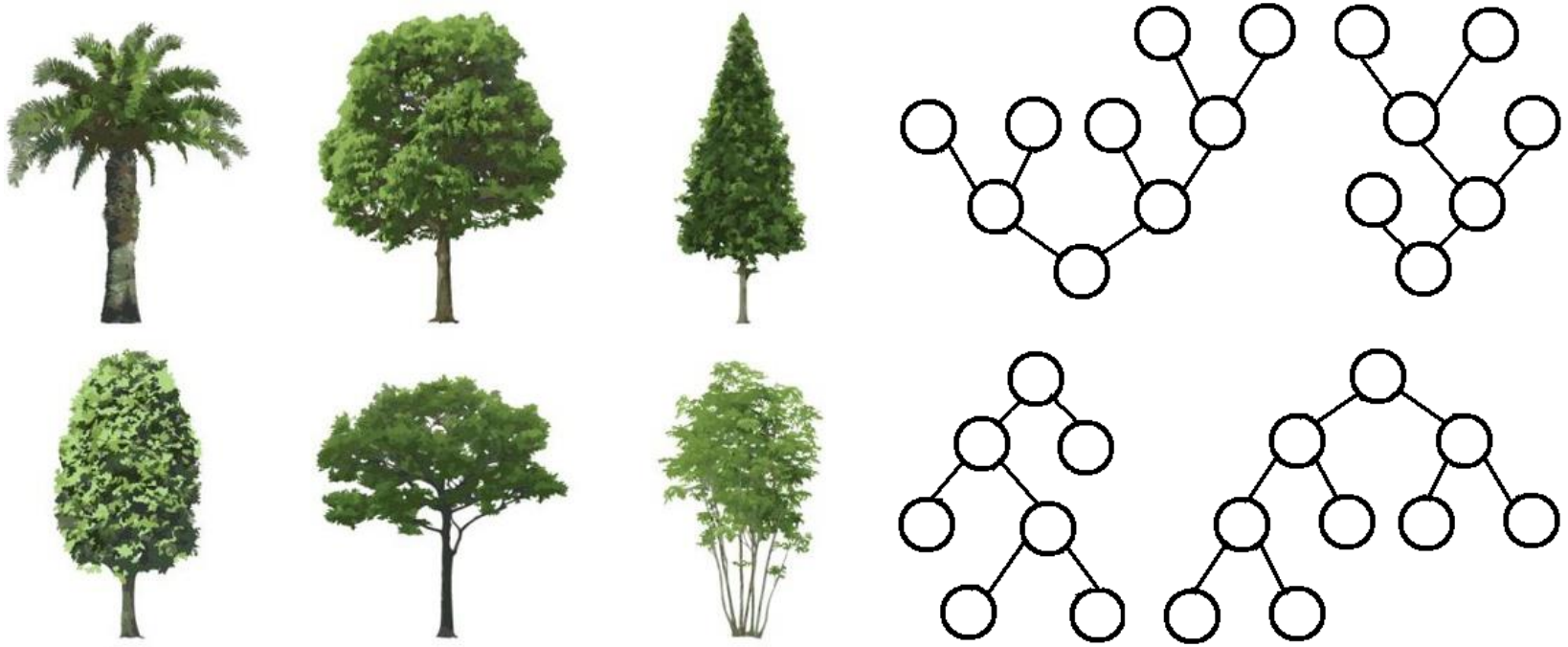
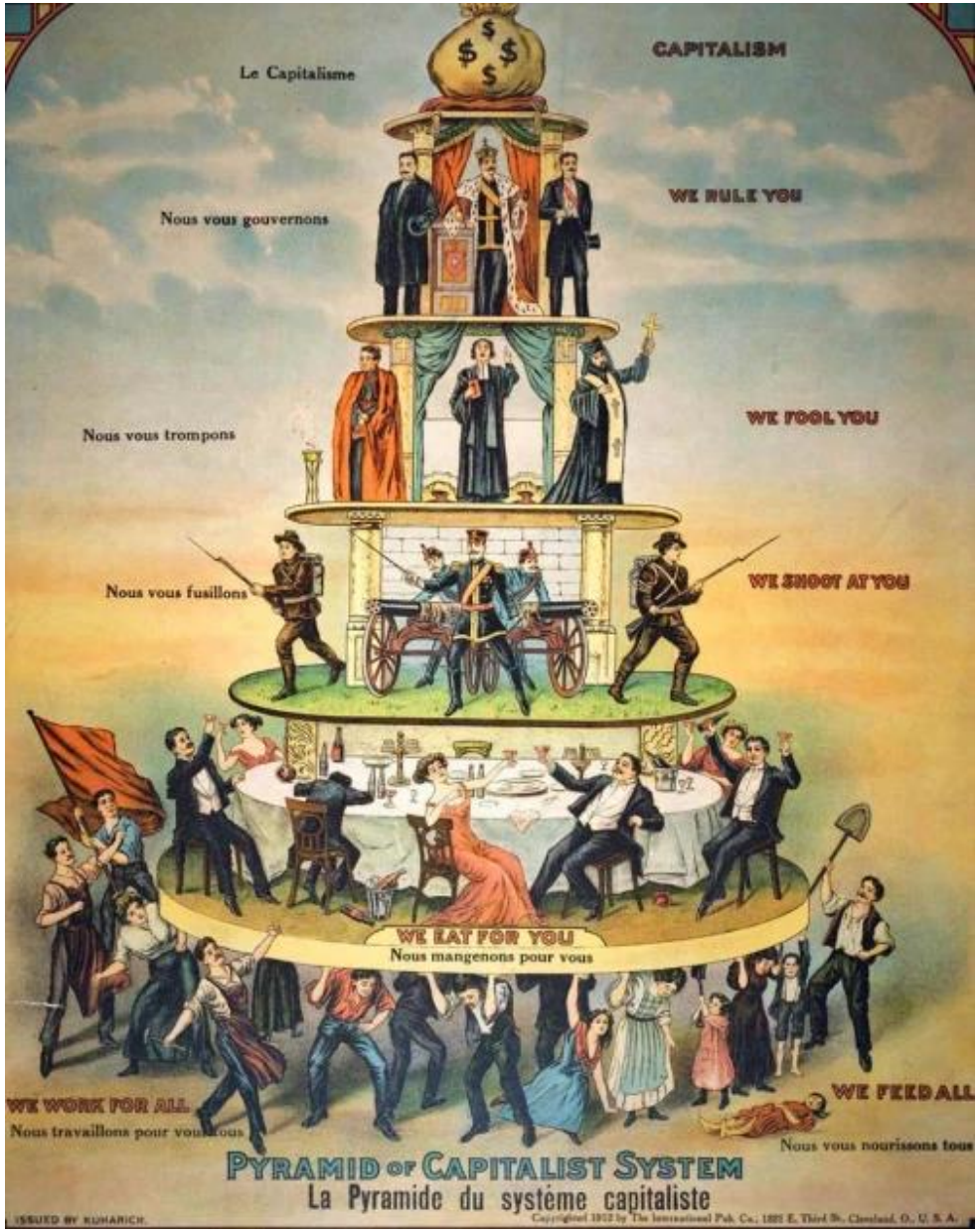
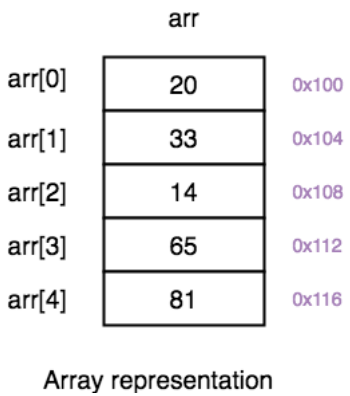
Data Structures (3)
Trees



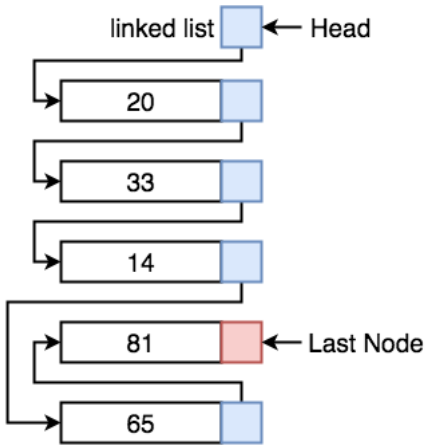
Dr. S.S.Shehaby

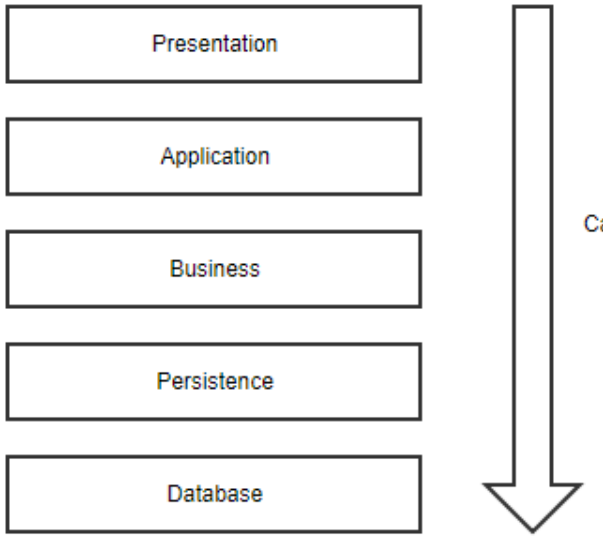
Linear structures, Layers





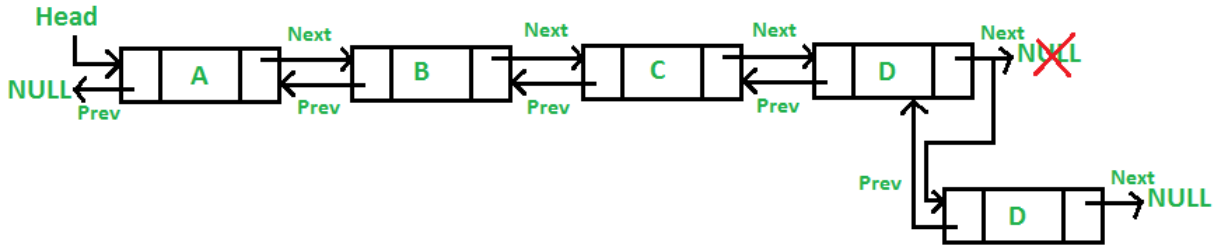
Array representation



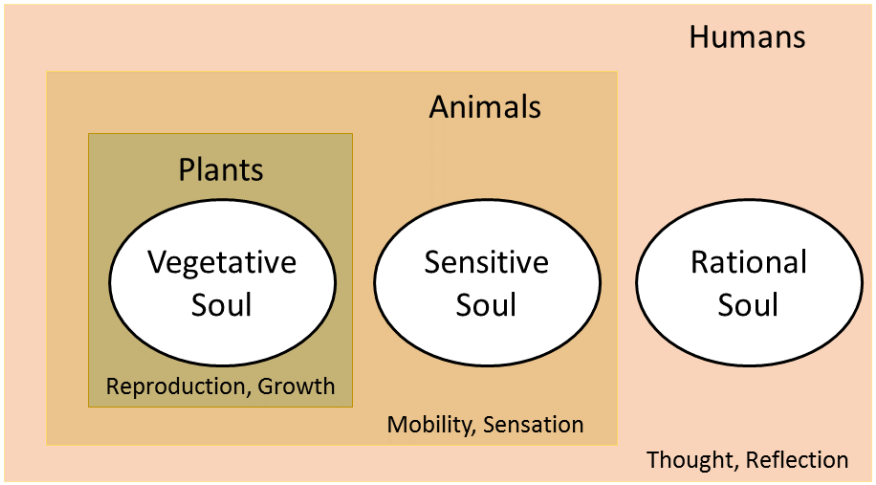


Calls between layers flow downwards

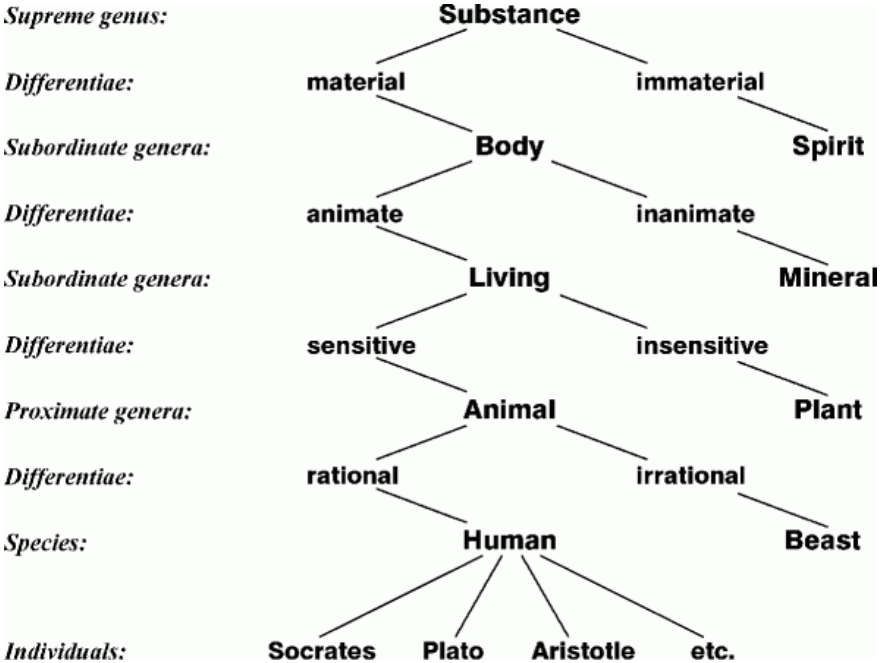
Linear structures, Layers



Trees in Philosophy

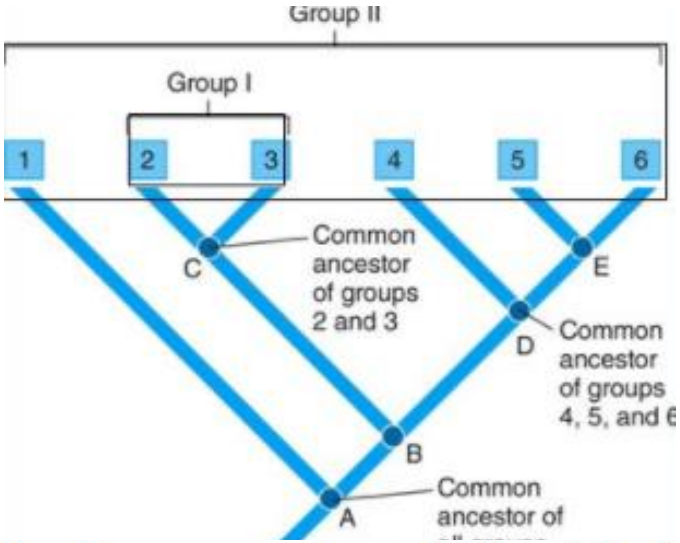


Aristotle: kinds of soul ("psyches"): the vegetative soul, the sensitive soul, and the rational soul.



Porphyry presented Aristotle's classification of categories in a way that was later adopted into tree-like diagrams of dichotomous divisions

Binary Trees/ dichotomies



The Transcendental Signified and Binaries

They are the upper terms in hierarchical binaries: e.g.

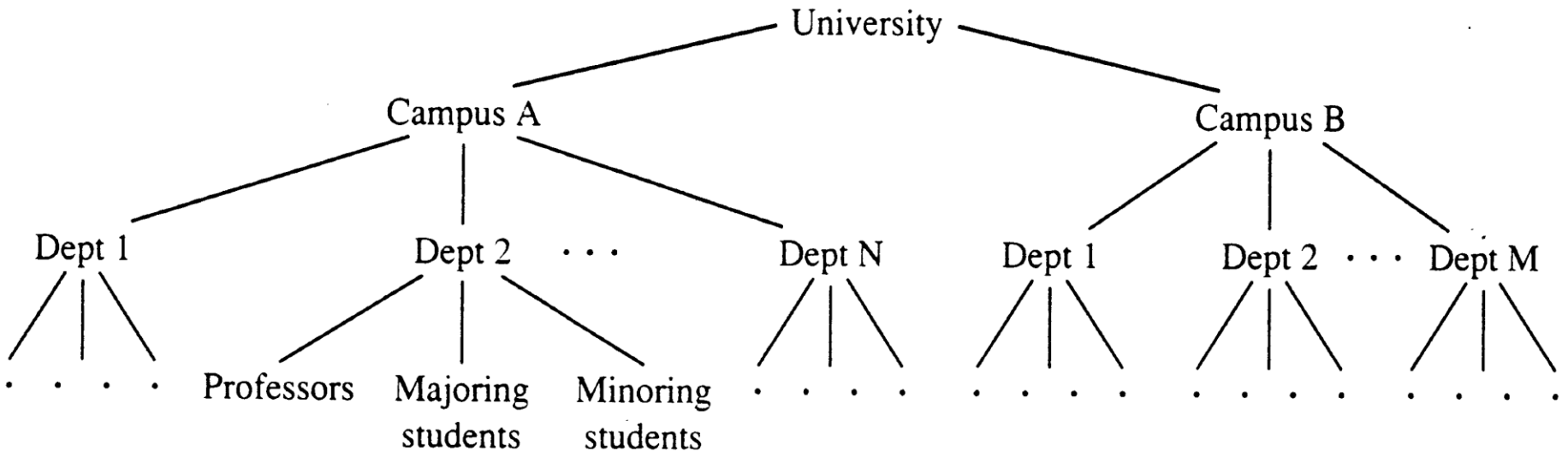
Man	*Light*	*Reason*	*Culture*	*The Public; West, etc.*
Woman	*Dark-ness*	*Emotion*	*Nature*	*The Private; East, etc.*

Aristotle's excluded middle (the principle of excluded middle) states that for any proposition, either that proposition is true or its negation is true. It is the third of the three classic laws of thought

Deconstruction/ Modernism: Jacques Derrida phallogocentrism

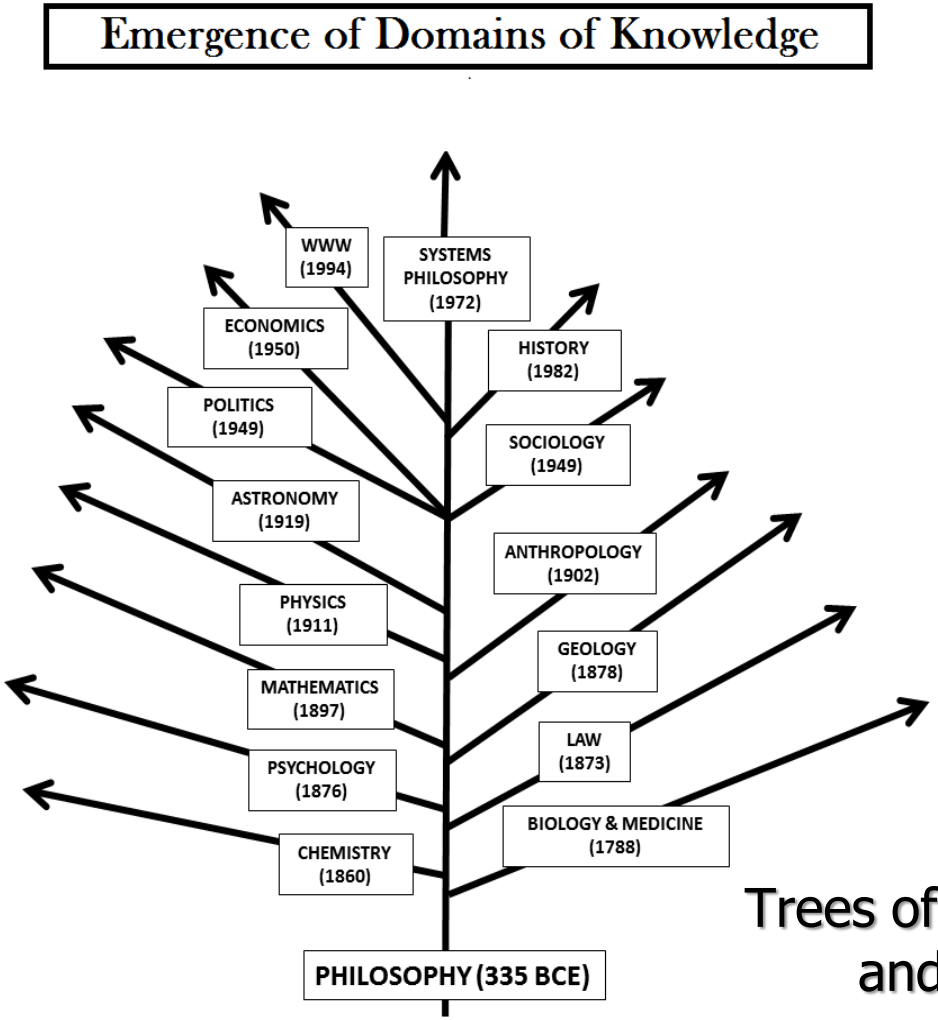
Trees (Organization charts)

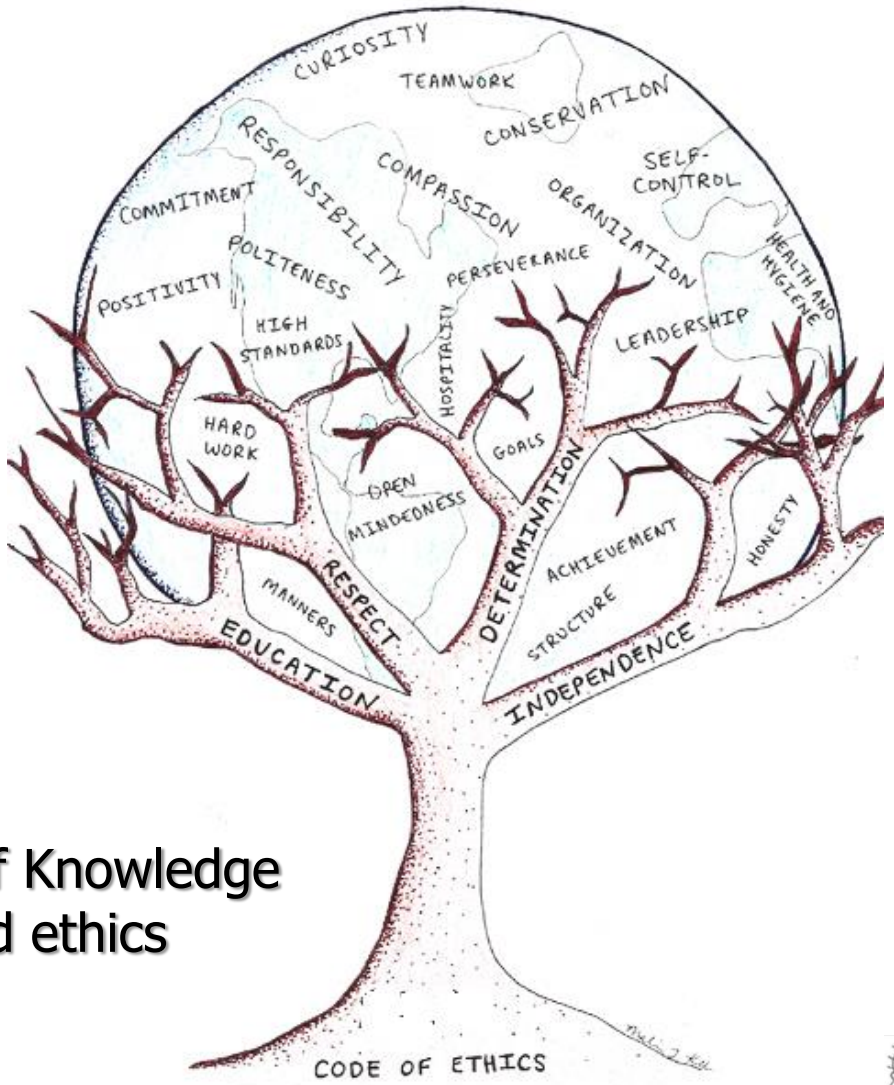
Examples: hierarchy of a university, genealogical tee, grammar tree, organic tree, … (Virtually all areas of science make use of trees to represent hierarchical structures.)



Hierarchical structure of a university shown as a tree.

Tree of Everything (Apropos of Nothing)

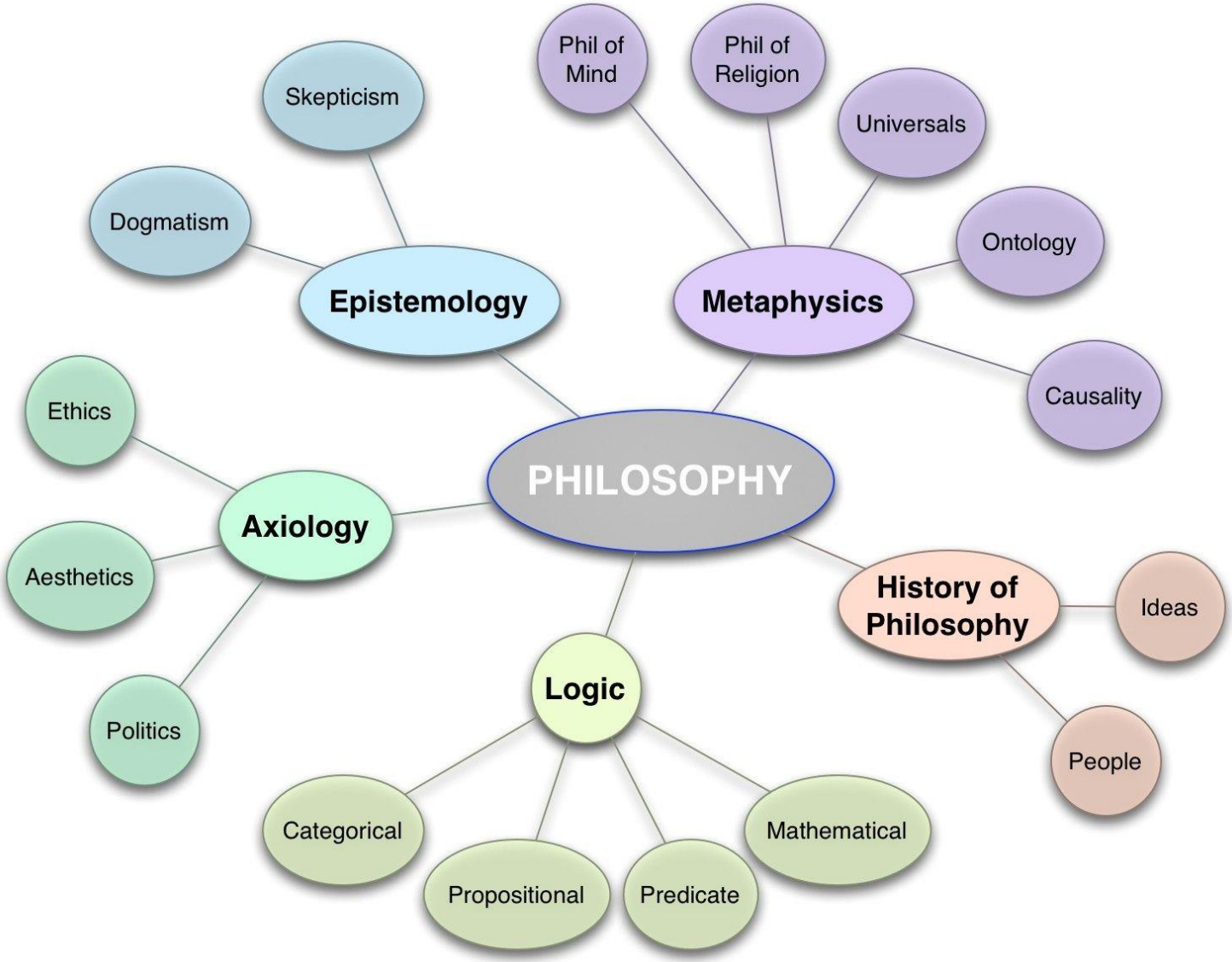




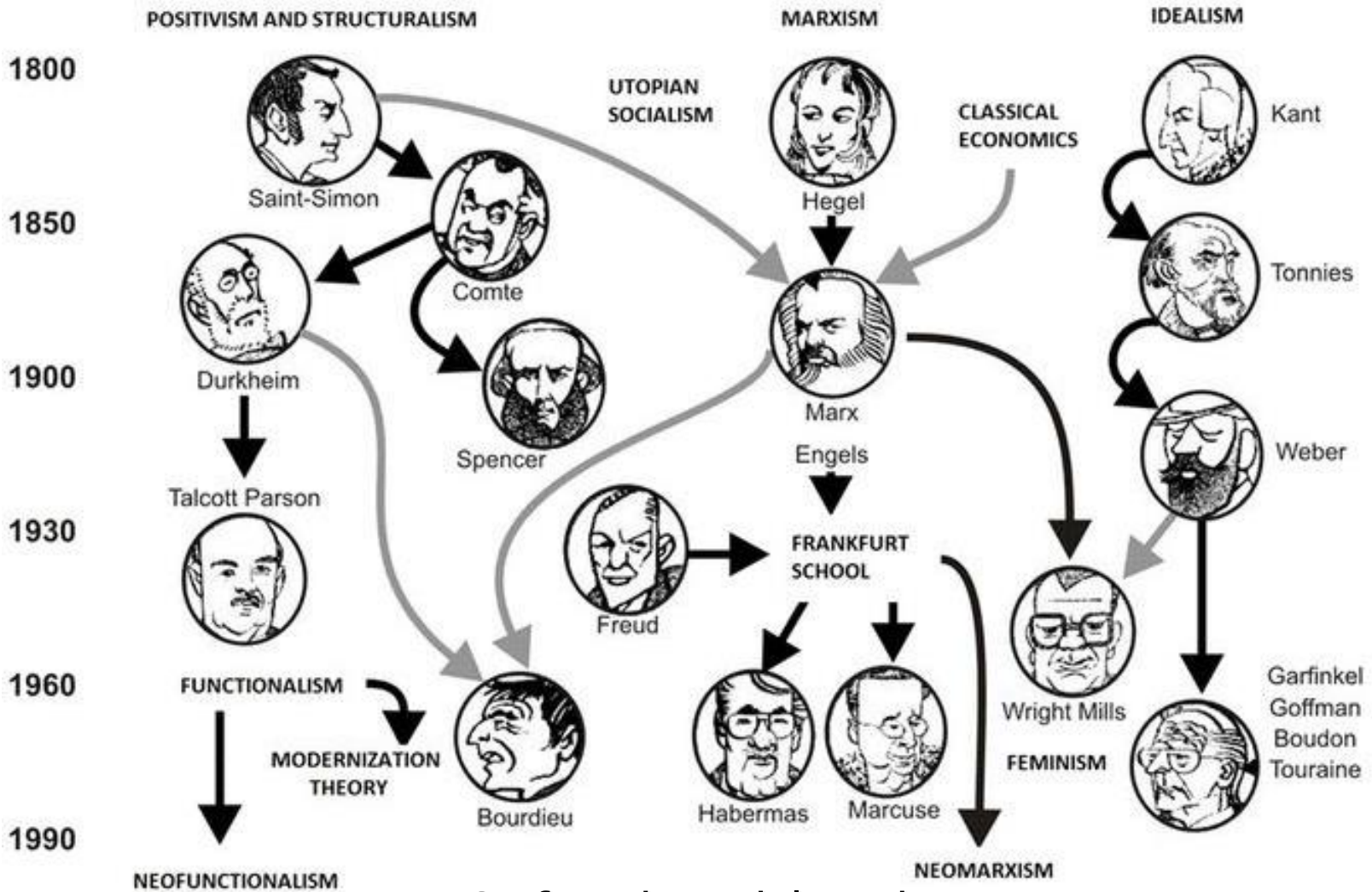
Trees of Knowledge and ethics

`PHILOSOPHY' (e.g. Aristotle, 335 BCE) at first, includes *all* knowledge domains (e.g., MATHEMATICS, GEOMETRY, PHYSICS, CHEMISTRY, BIOLOGY. ART, etc)

Tree : Philosophy



Tree of Everything (Not a Really Tree)

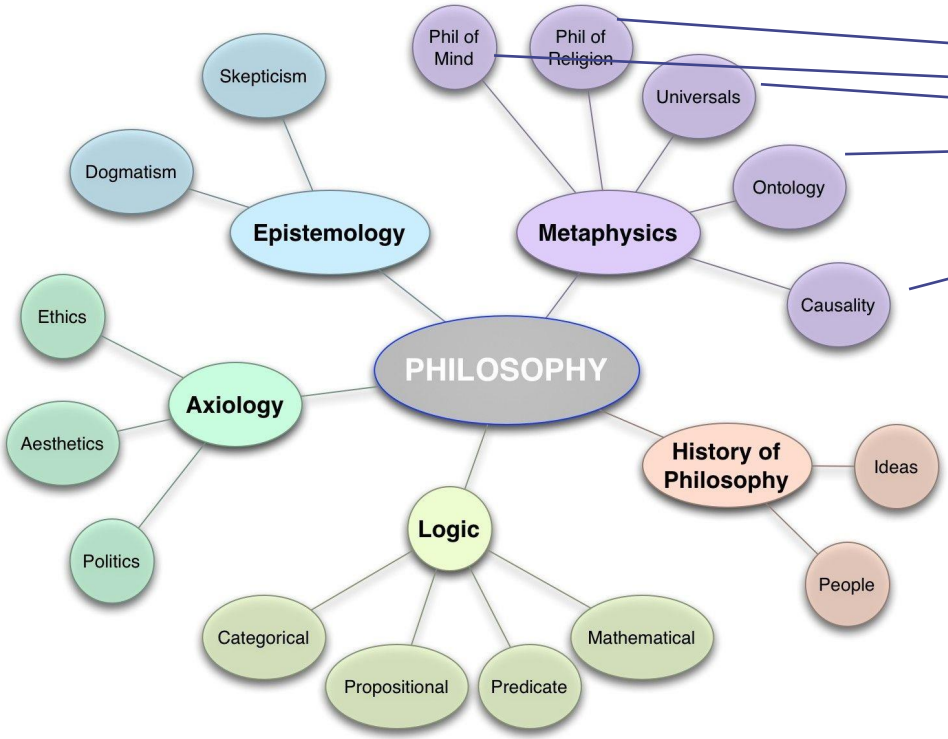


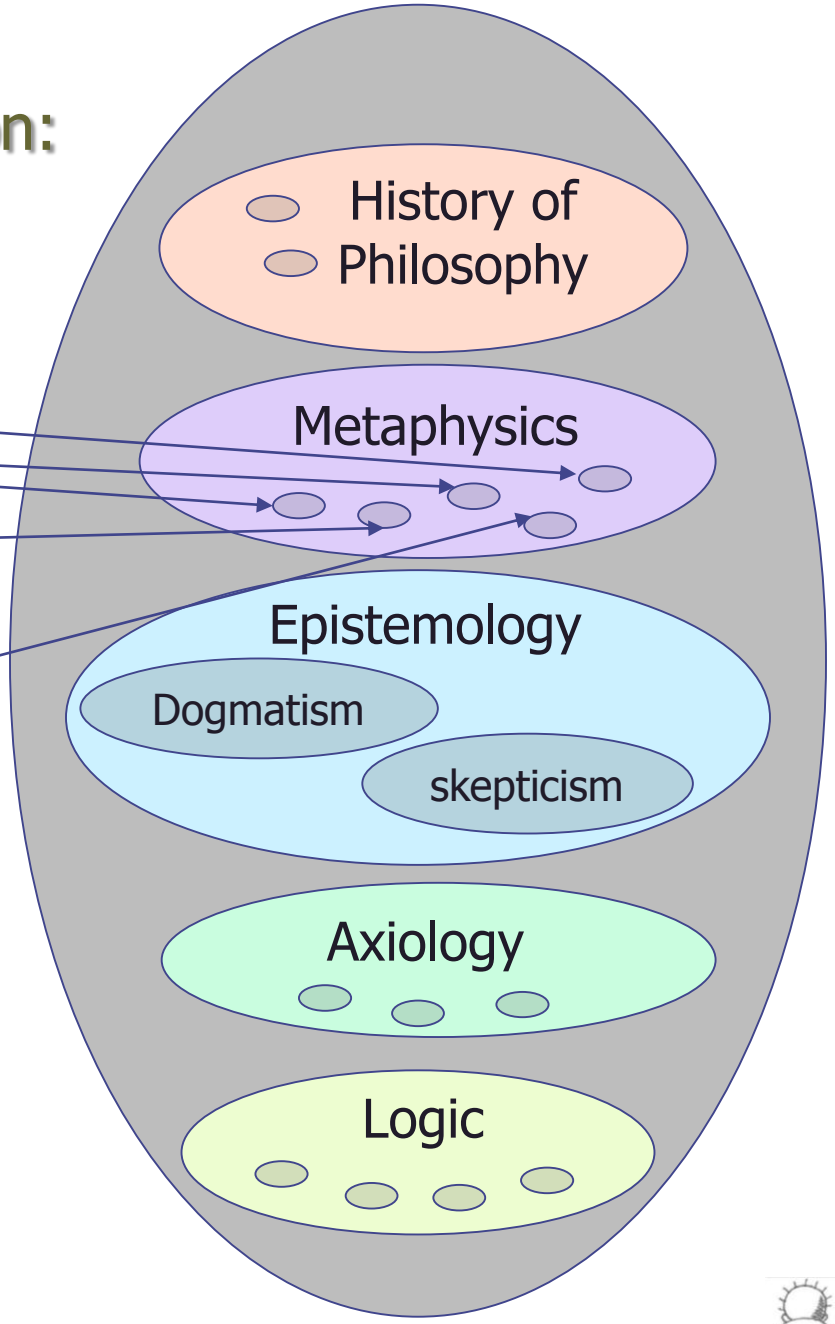
DAG of modern philosophers

Tree as a special type of relation:

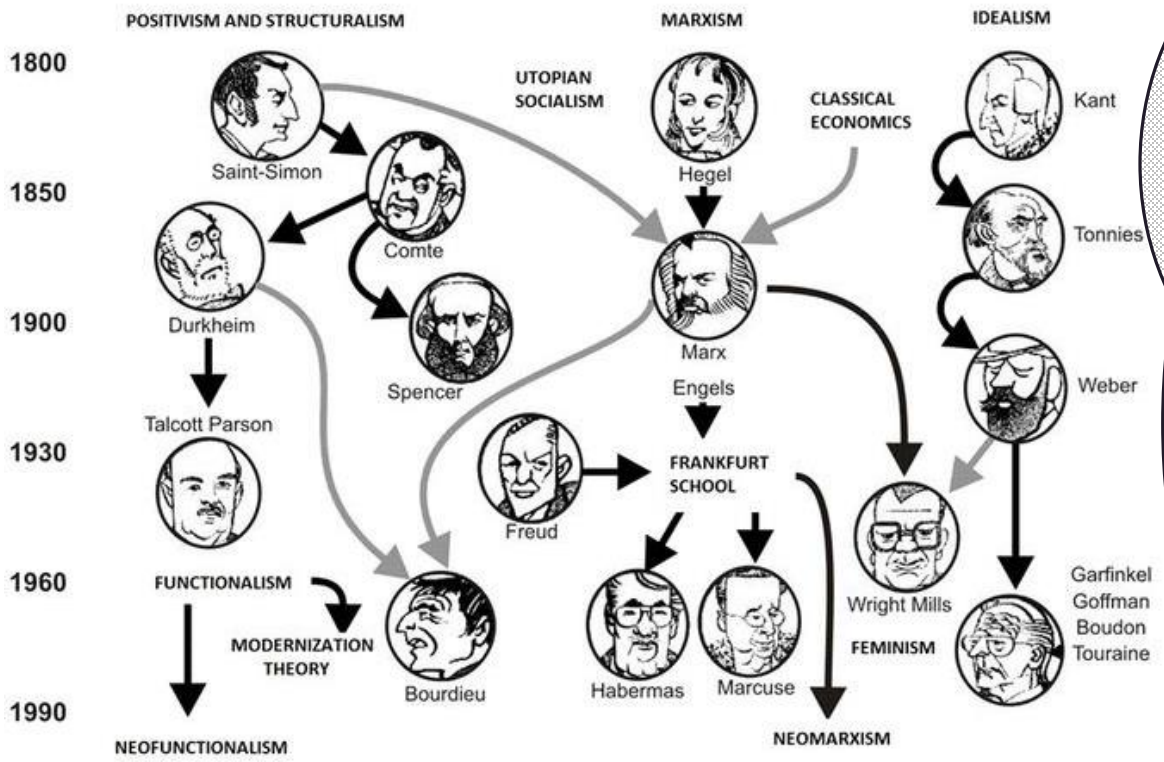
Venn Diagram

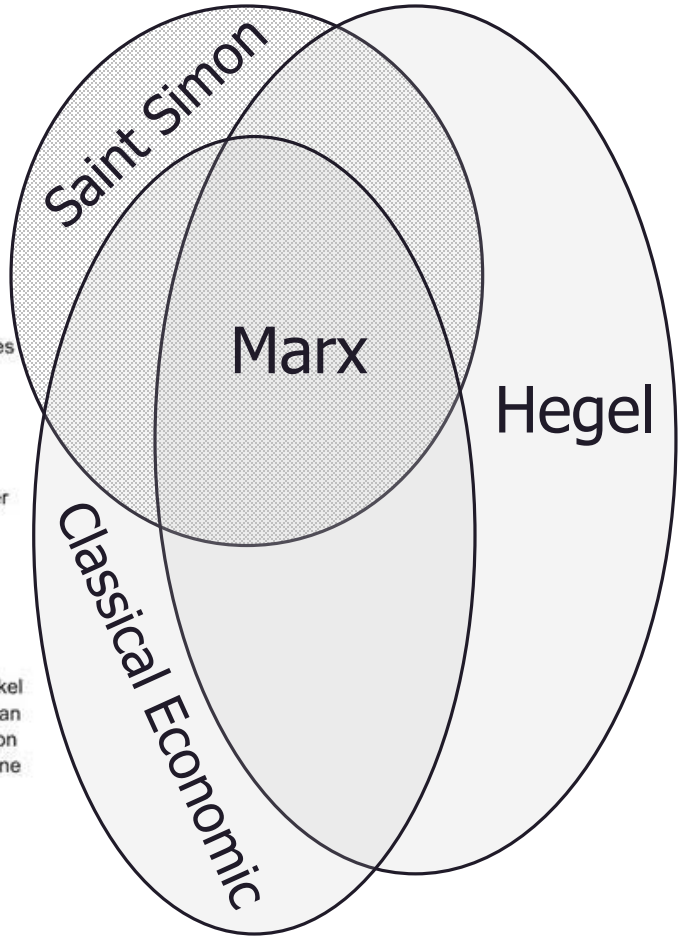
No overlapping





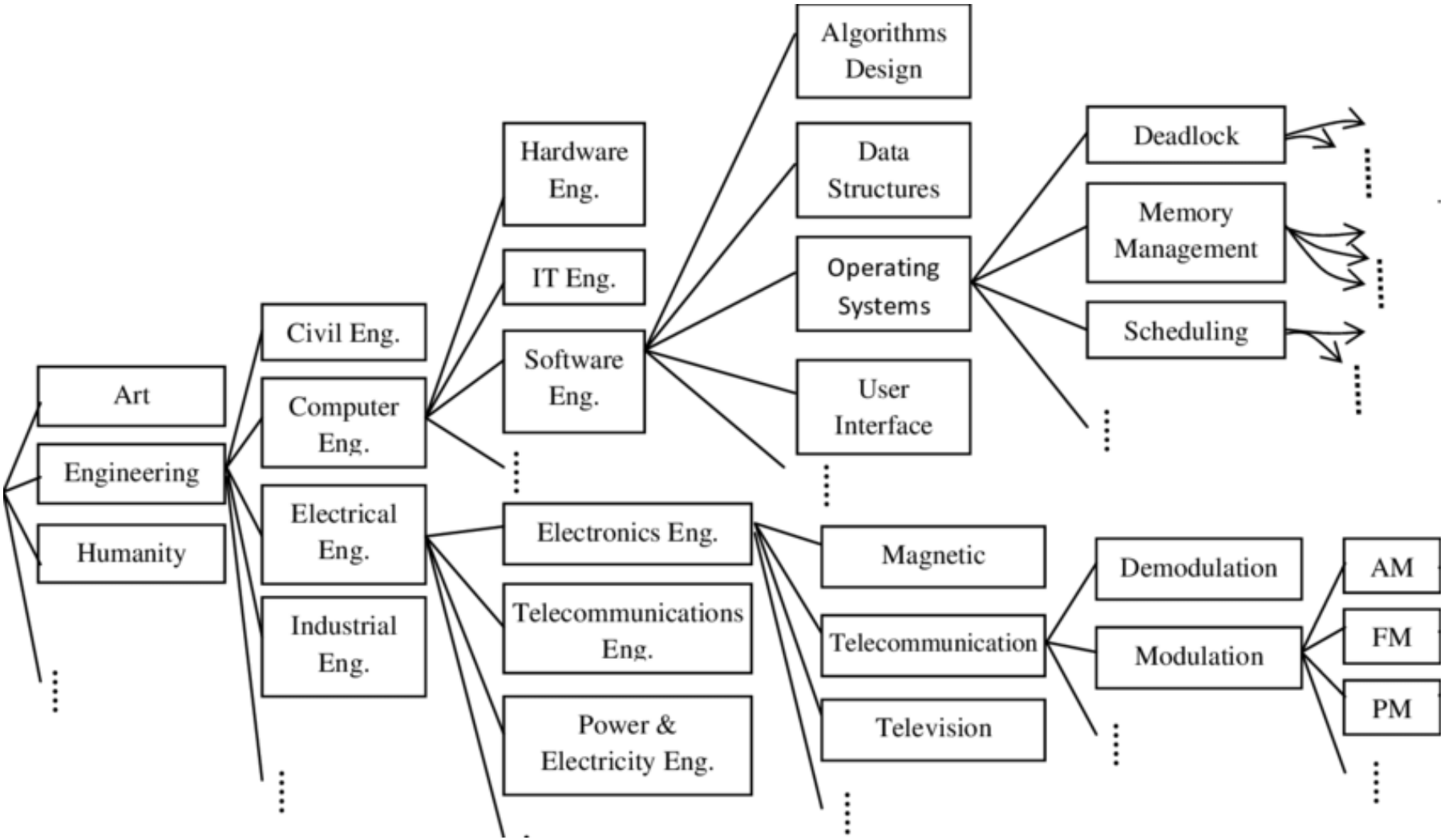
Tree vs DAG as a special type of relation:
Venn Diagram



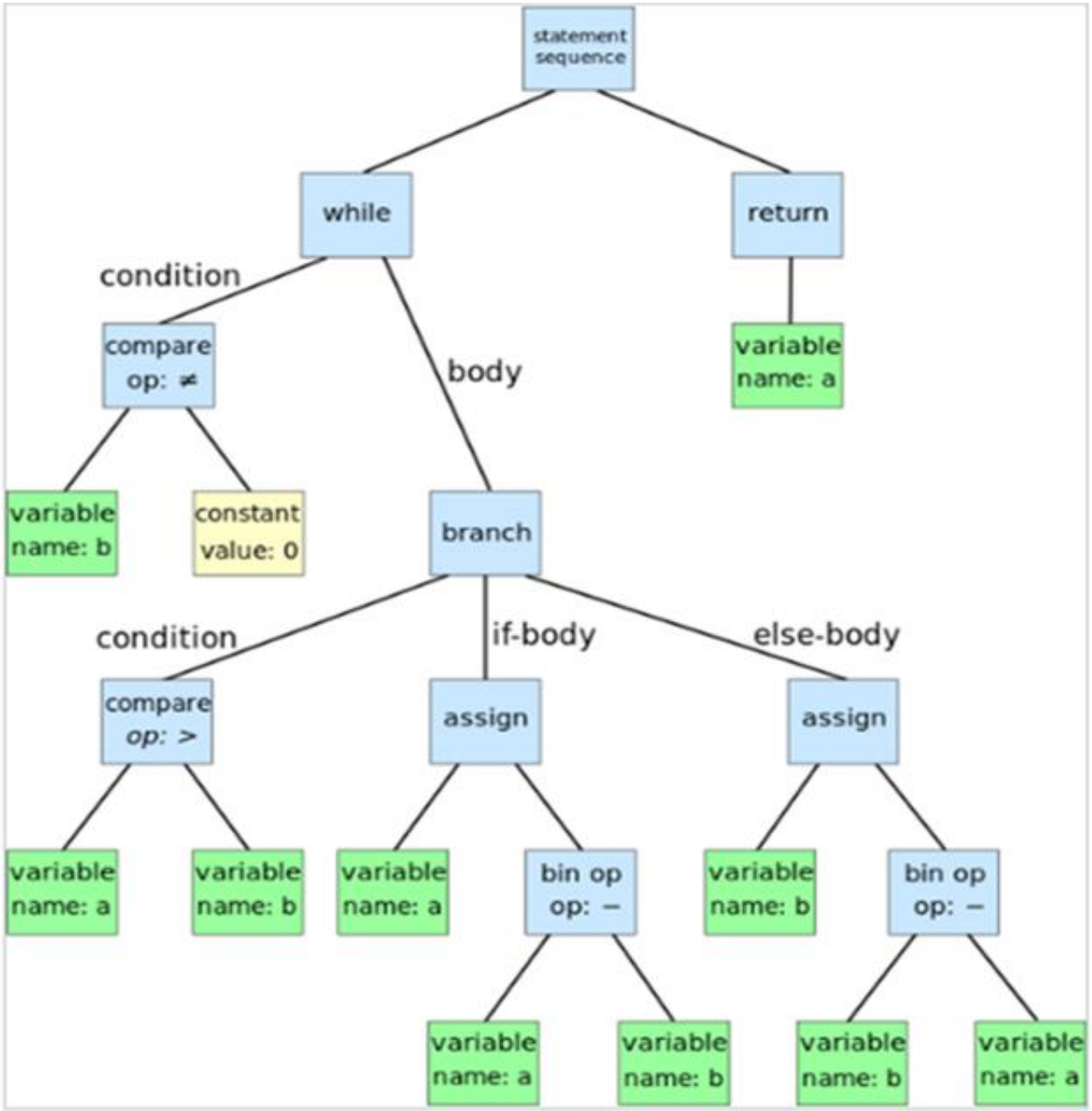


DAG of modern philosophers

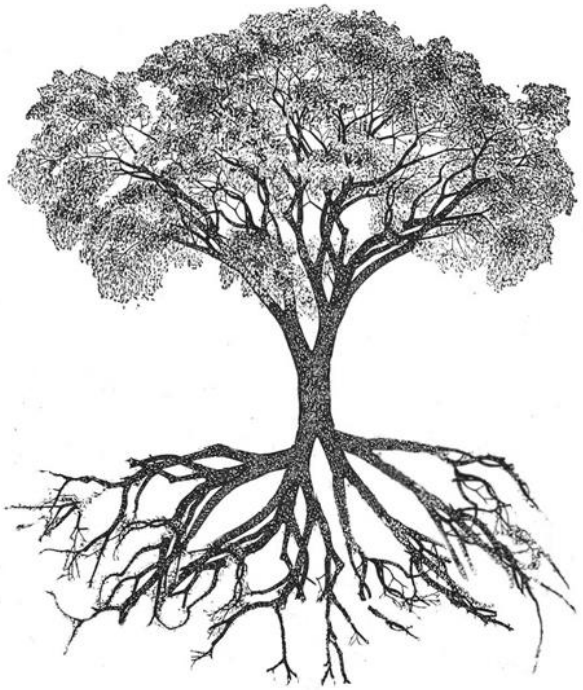
Trees in Science



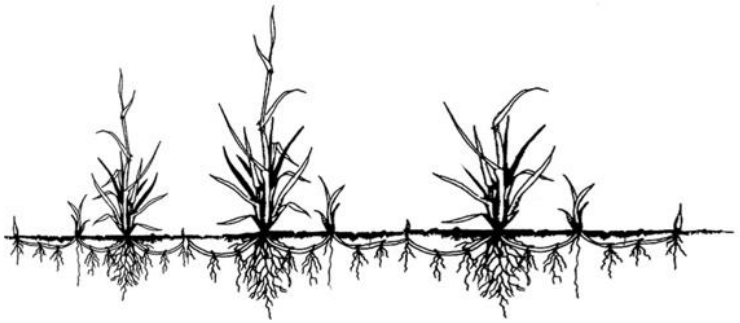
Trees of syntactical language



Trees vs Rhizomes



Tree



Rhizome

It is odd how the tree has dominated Western reality and all of Western thought, from botany to biology and anatomy … the root foundation … The West has a special relation to the forest, and deforestation …

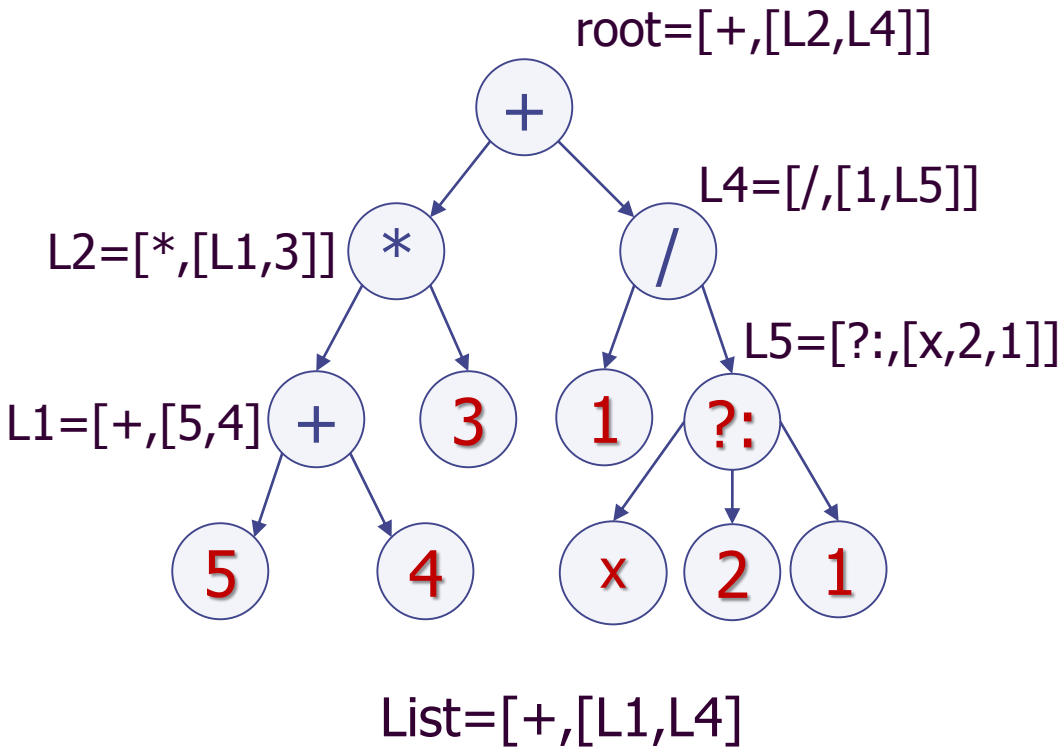
Gilles Deleuze and Félix Guattari: A Thousand Plateaus

Representing Trees (Arithmetic Expressions)

```
INDENT
+
  *
    +
      5
      4
    3
  /
    1
    ?:
      x
      2
      1
```

Levels:

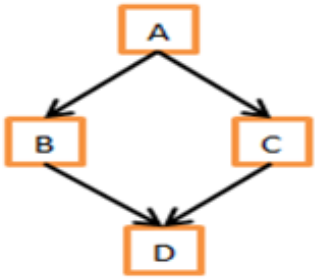
1: +
10: *
20:+
30:5
30:4
20: 3
10: /
20:1
20:?:
30:x
30:2
30:1



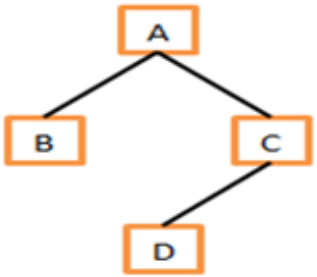
List=[+,[L1,L4]

Trees

- Linked lists are linear structures and it is difficult to use them to organize a hierarchical representation of objects.
- Although stacks and queues reflect some hierarchy, they are limited to only one dimension.
- The Tree data type consists of nodes and arcs. Unlike natural trees, these trees are depicted upside down with the root at the top and the leaves at the bottom
- A Direct (connected) acyclic Graph is a special case of graphs, with no cycles (loops)
- Trees are special case of DAG with the notion parent and children



DAG

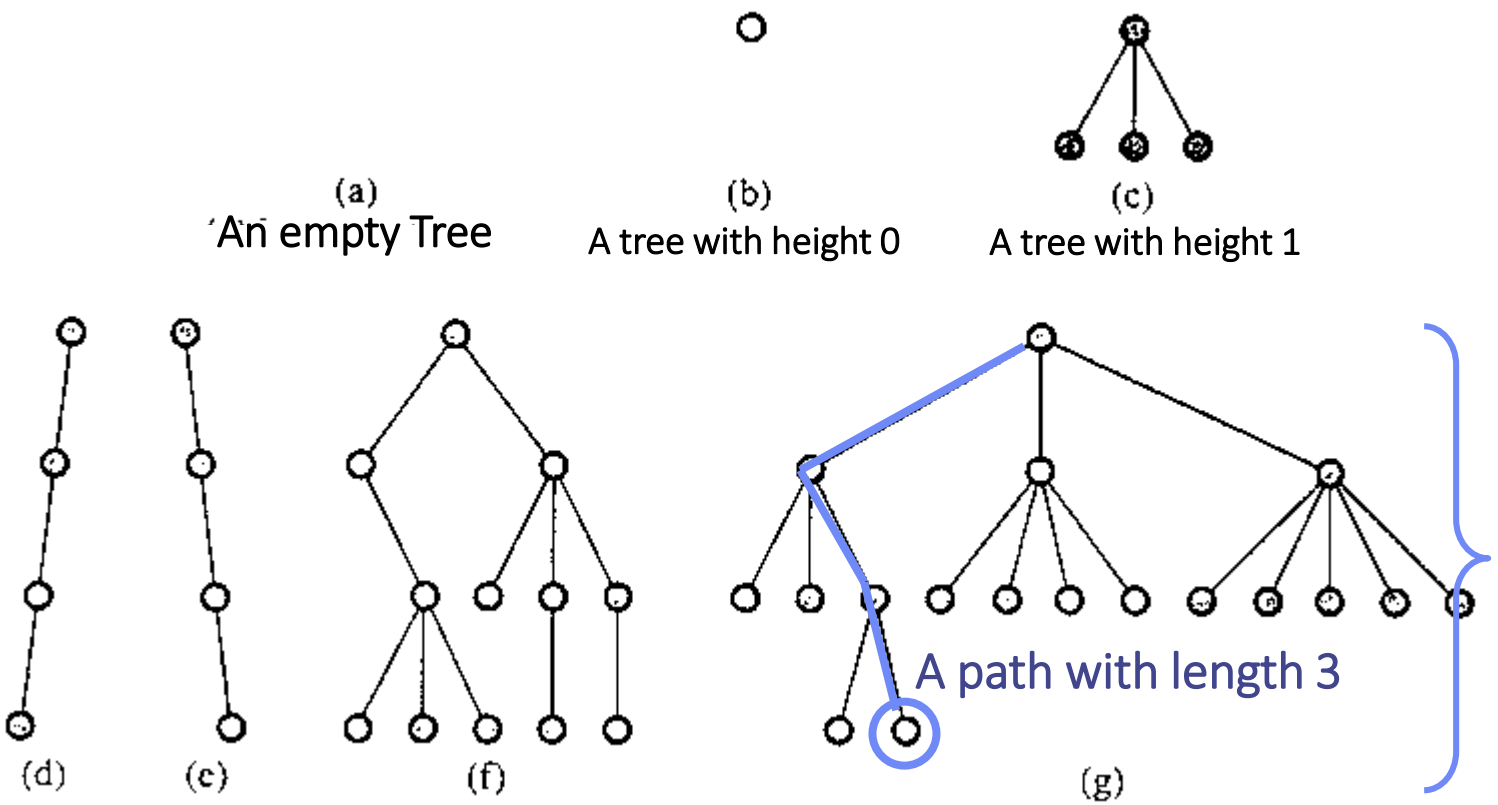


Tree

Trees

- The **root** is a node that has **no parent**; it can have only child nodes. **Leaves**, on the other hand, have **no children**.
- A tree can be defined **recursively** as the following:
- 1. An empty structure is an empty tree
- 2. If t_1, t_2, …, t_k are disjoint trees, then the structure whose root has its children the roots of t_1, t_2, …, t_k is also a tree
- 3. Only structures generated by rules 1 and 2 are trees

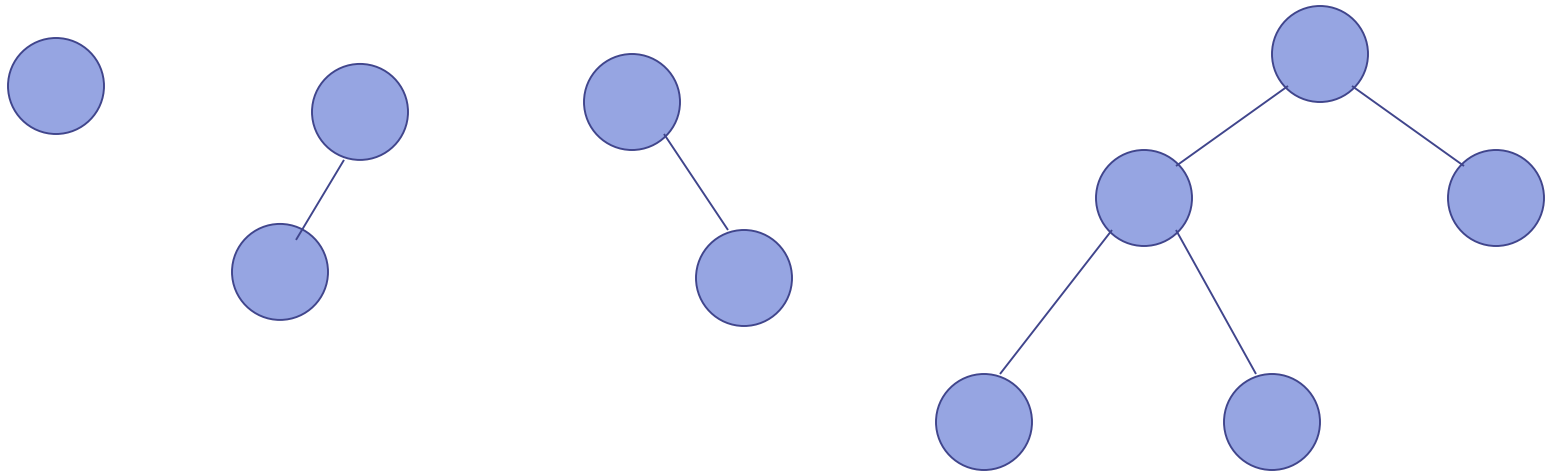
Trees



There is a unique path between any two nodes in a tree. (Hence a tree has no cycles.)

Binary Trees

A binary tree is a tree whose nodes have **at most two children** (possibly empty) and each child is designated as either **a left child** or **a right child**.



Examples:

Binary Trees

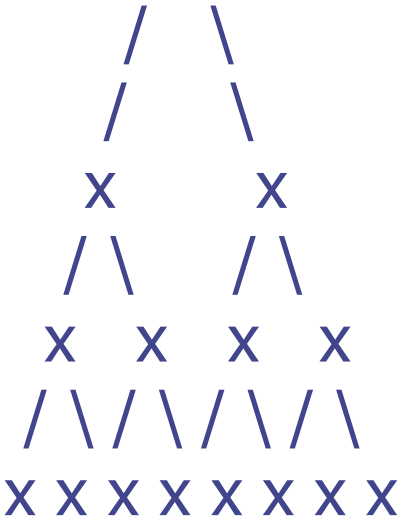
- A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.
- Every node (*excluding a root*) in a tree is connected by a directed edge from exactly one other node. This node is called a parent.
- Each node can be connected to at most two nodes, called children.
- Nodes with no children are called leaves, external nodes.
- Nodes which are not leaves nor the root are called internal nodes. Nodes with the same parent are called siblings.

Binary Trees

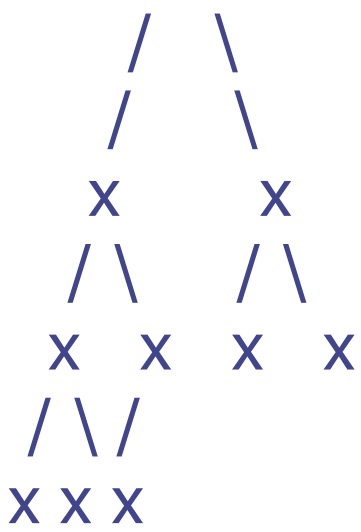
- The depth of a node is the number of edges from the root to the node.
The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- A full binary tree is a binary tree in which each node has exactly zero or two children.
- A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Binary Trees

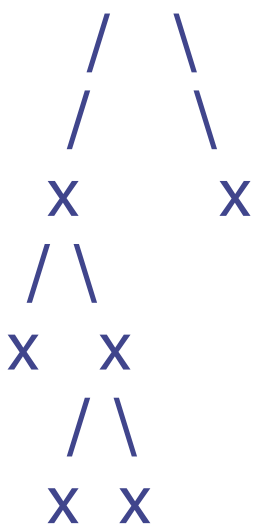
Perfect



complete



strict/full



Binary Trees

In a binary tree, if all nodes at all levels except the last had two children, then there would be 2^0 node at level 1 (root), 2^1 nodes at level 2, and generally, 2^i nodes at level i+1. A tree satisfying this condition is referred to as a **complete binary tree**.

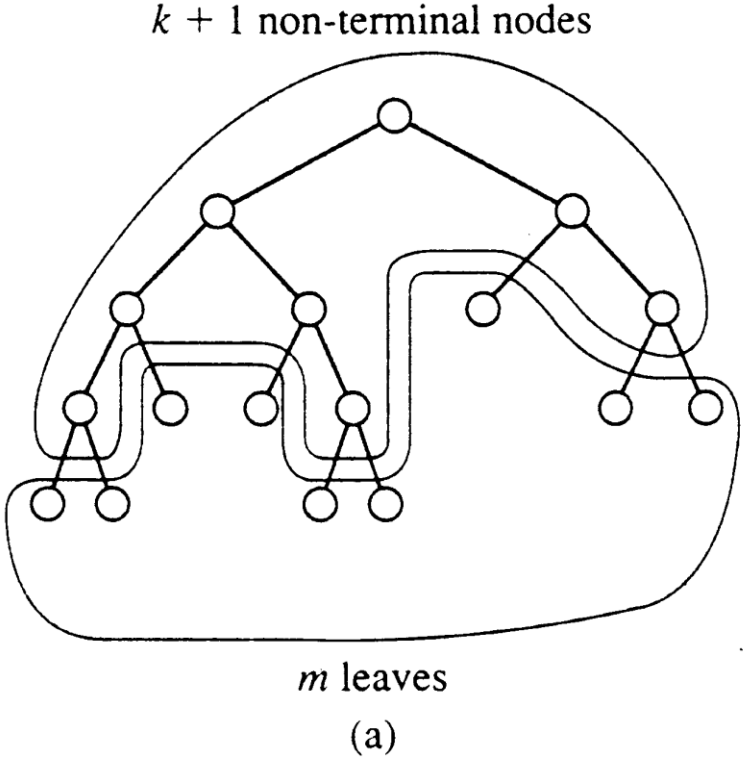
In a complete binary tree, all non-terminal nodes have both their children, and **all leaves are at the same level**.

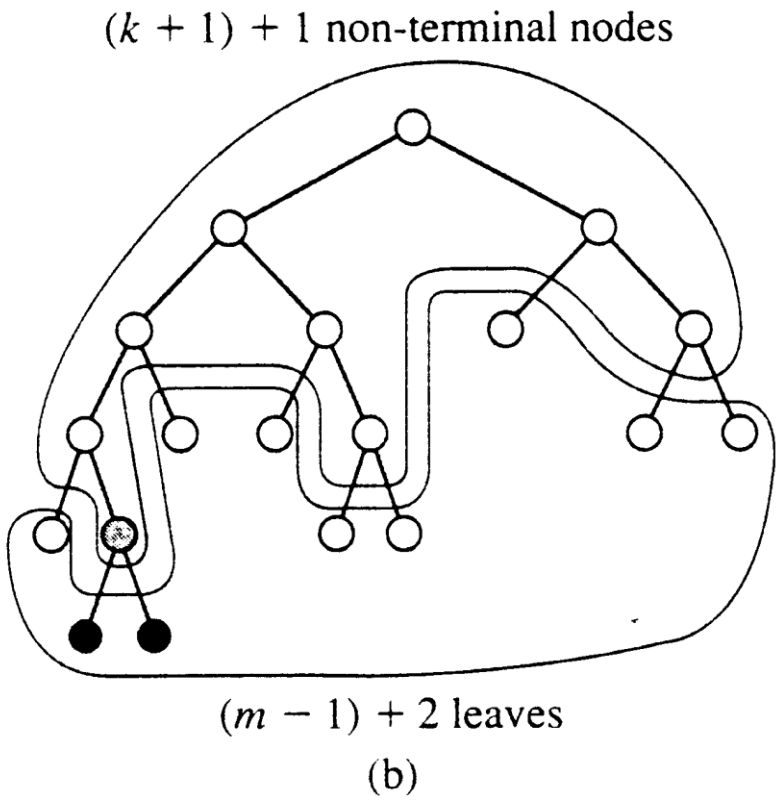
Total number of nodes in a complete binary tree with level k is

$$\underbrace{1+2+\cdots+2^{k-2}}_{\text{nonterminal nodes}}+\underbrace{2^{k-1}}_{\text{terminal nodes(leaves)}}=2^k-1$$

Binary Trees

Lemma: For all non-empty binary trees whose non-terminal nodes have exactly two non-empty children, the number of leaves m is greater than the number of non-terminal nodes k and $m=k+1$.





Number of binary trees (structurally different) with n Nodes

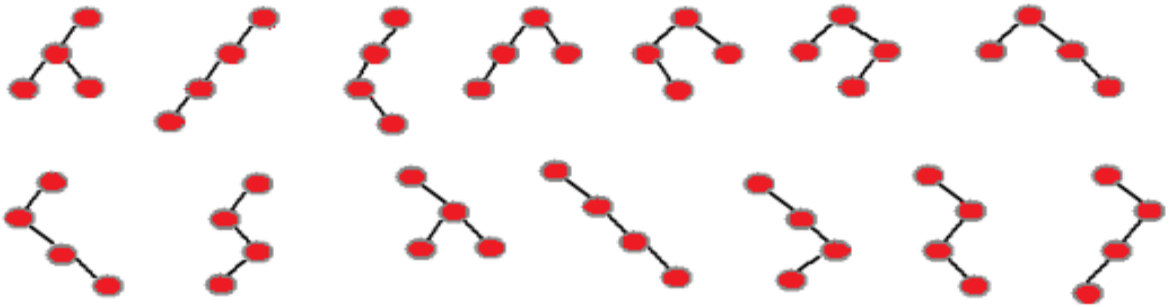
$C_0=C_1=1$

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i \, C_{n-i} \quad \text{for } n \geq 0$$

$C_2=2$

$C_3=5$

$C_4=14$

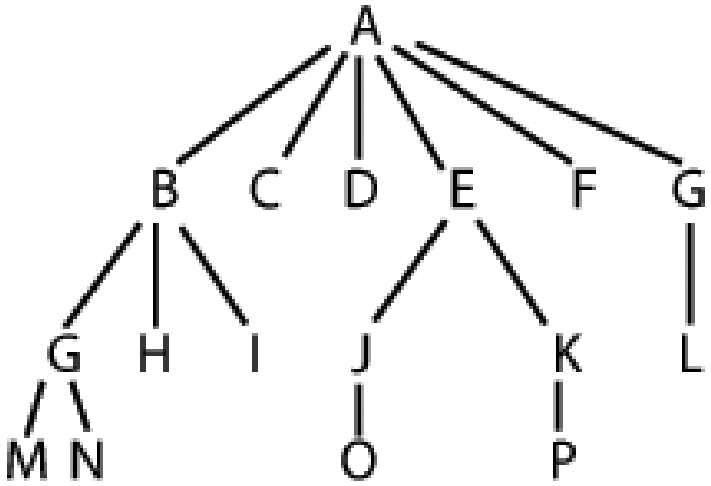


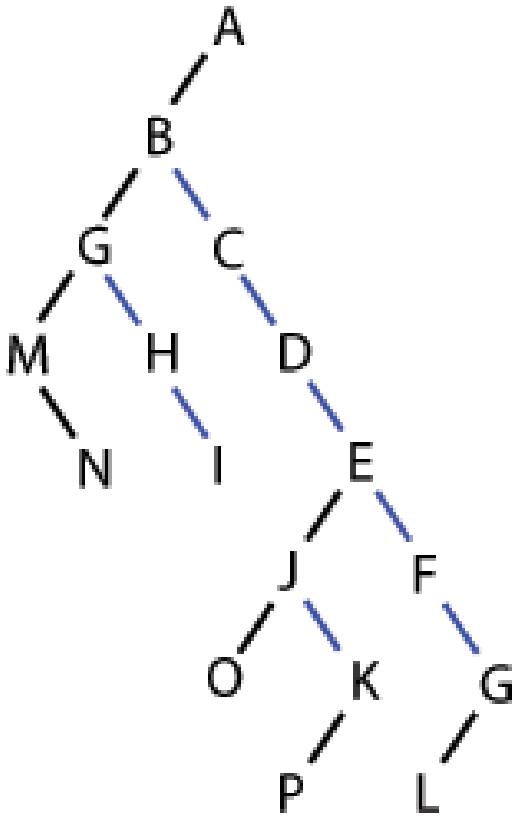
Catalan Number:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \qquad \text{for } n \geq 0.$$

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012....

General To Binary Tree

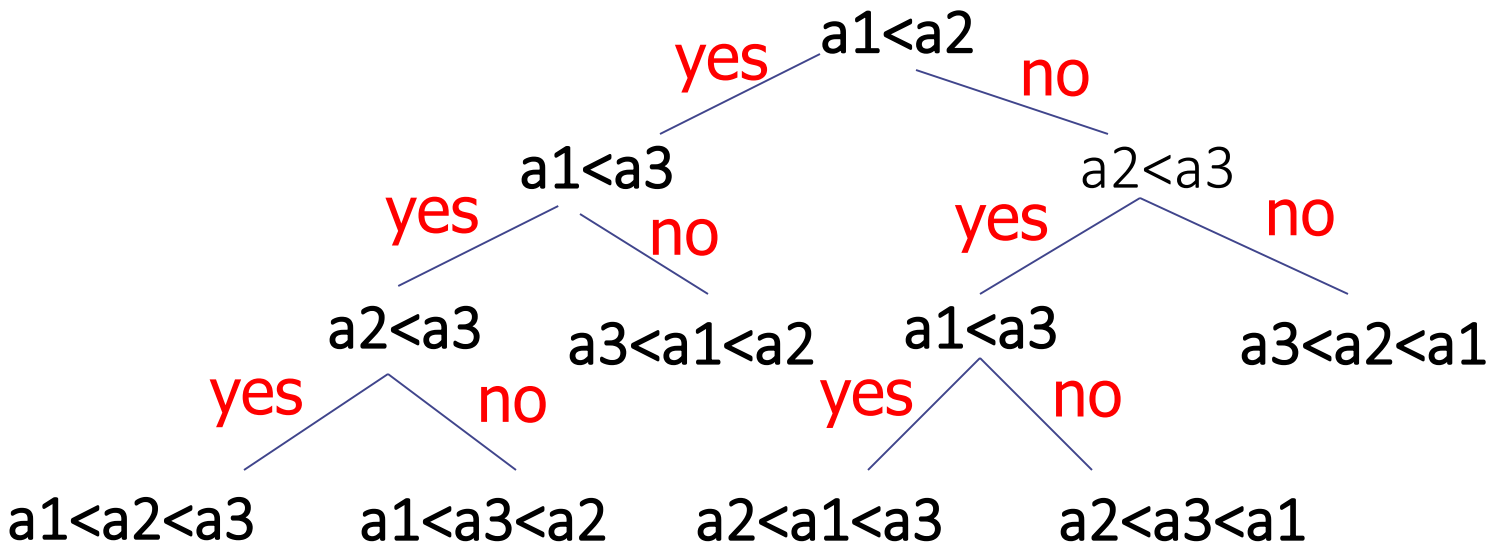




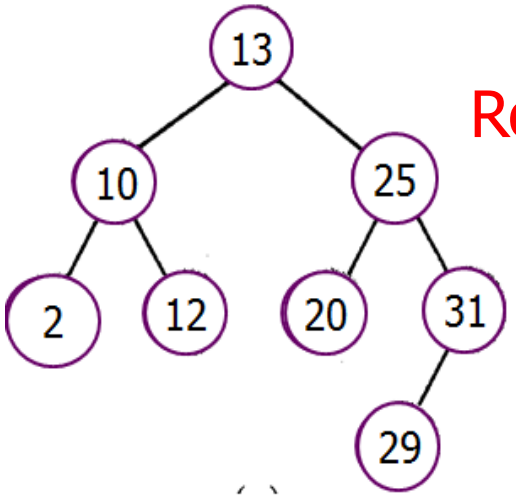
Binary Trees (Example)

A **decision tree** is a binary tree in which all nodes have either zero or two non-empty children.

Example: sort a1, a2, a3



Array Representation

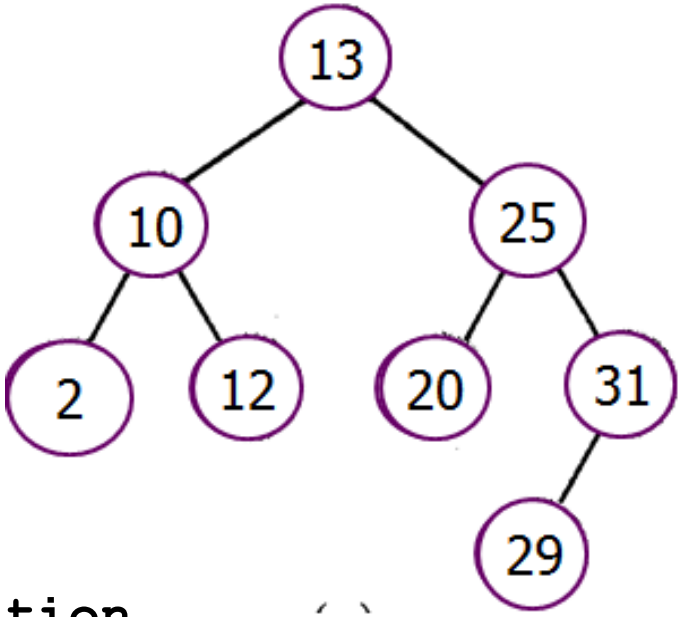


Root

Index	Data	Left	Right
0	13	1	2
1	10	3	4
2	25	5	6
3	2	-1	-1
4	12	-1	-1
5	20	-1	-1
6	31	7	-1
7	28	-1	-1

However, it is hard to predict **how many nodes will be created** during a program execution. (Therefore, how many spaces should be reserved for the array?)

Other Tree Representations (List/Array)



List Representation

```
t=[13 [10 [2 12]] [25 [10 31 [29]]]]
```

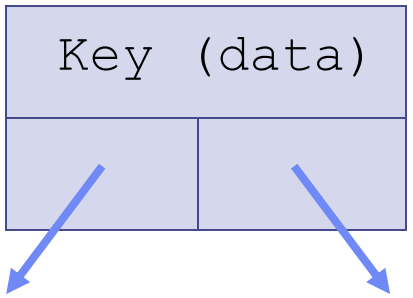
Array Representation

```
int t[]={13,10,25,2,12,20,31,X,X,X,X,X,X,29,X}
```

However ?? **Only Good FOR PERFECT/COMPLETE trees**

Dynamic Linked Representation

```
struct Node {
    TYPE key;
    struct Node *left, *right;
};
typedef struct *Node Tree;
```



Left_child right_child

Dynamic Linked Representation

```
struct Node
{
      int key;
      struct Node *left,*right; // in c++ you don't need the 'struct'
};
typedef struct Node Node;
Node * newNode(int val){
   Node* n=malloc(sizeof(Node));
   n->key=val;n->left=NULL;n->right=NULL;
   return n;
}
void kill(Node* n){free(n);}
typedef  Node * Tree;
```

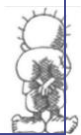

Dynamic List Representation-C++ Styles

```
struct Node { //default public
      int key;
      Node *left,*right;
      Node (int value)   // Constructor
            {key=value;left=right=NULL;} };

class Tree {
   Node * root;
   Tree () {root=NULL;}
}
```

```
class Node { // public
 public:
      int key;
      Node *left,*right;
      Node (int value)
            {key=value;left=right=NULL;} };
class Tree {
   Node * root;
   Tree () {root=NULL;}
}
```

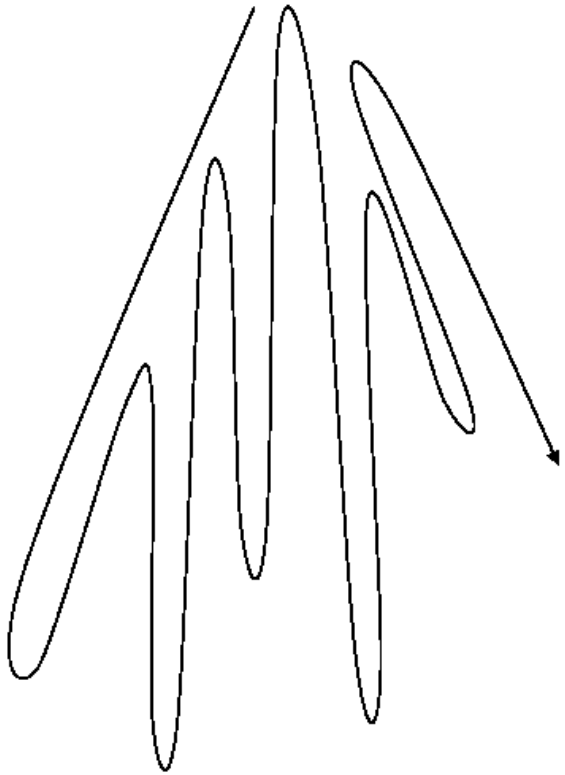
```
class Tree {  // Nested
class Node {
      int key;
      … }

   Node * root;
   Tree () {root=NULL;}
}
```

```
class Node { // Friends
   friend class Tree;
      int key;
      … }
class Tree {
   Node * root;
   Tree () {root=NULL;}
}
```



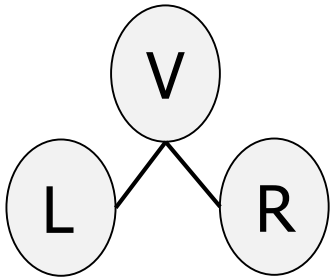
Depth-First Traversal

Depth-First Traversal

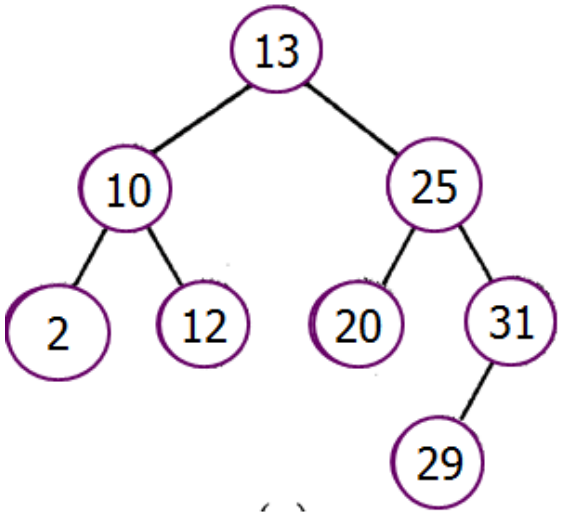


Depth-first traversal proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right). Repeat this process until all nodes are visited.

Depth-First Traversal



There are three tasks of interest in this type of traversal:
V: visiting a node
L: traversal the left subtree
R: traversal the right subtree
An orderly traversal takes place if these tasks are performed in the same order for each node. So there are six possible traversals:



Pre-Order

VLR: 13,10,2,12,25,20,31,29

VRL: 13 25,31,29,20,10,12,2

In-Order

LVR: 2,10,12,13,20,25,29,31

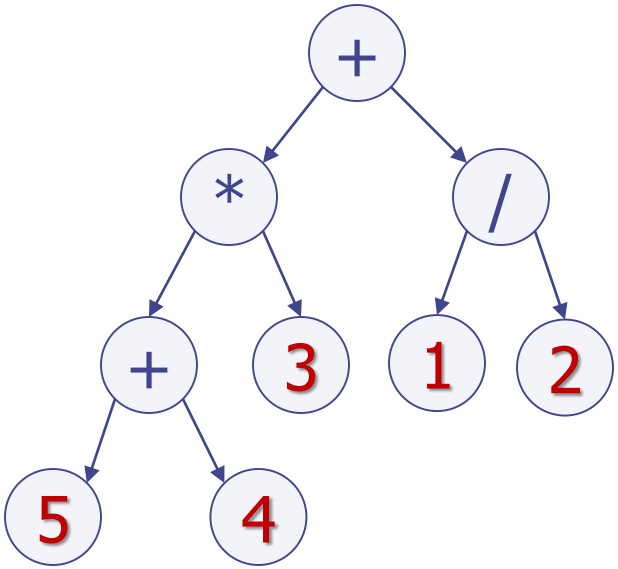
RVL: 31,29,25,20,13,12,10,2

Post-Order

LRV: 2,12,10,20,29,31,25,13

RLV: 29,31,20,25,12,2,10,13

About Traversals & expressions



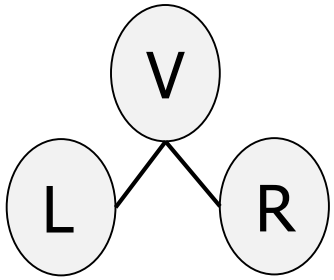
InOrder:5+4*3+1/2

PreOrder:+*+5 4 3 / 1 2

PostOrder:5 4 + 3 * 1 2 / +

Depth-First Traversal Recursive

Assume the move is always from left to right, then there are three standard traversals:
VLR: preorder (binary) tree traversal
LVR: inorder tree traversal
LRV: postorder tree traversal

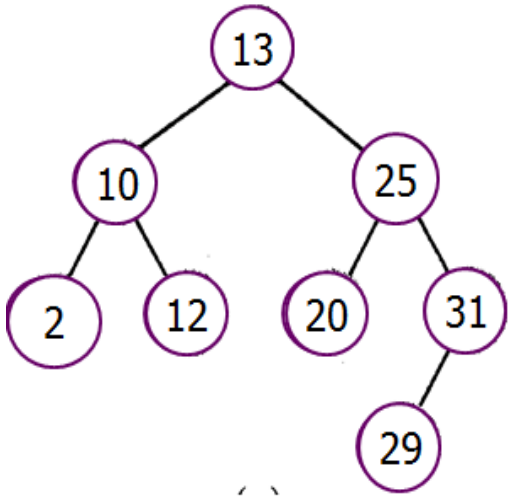



```
Preorder(Node * node)
{
   if (node) {
     visit(node);
     preorder(node->left);
     preorder(node->right);
   }
}
```

```
inorder(Node * node)
{
   if (node) {
     inorder(node->left);
     visit(node);
     inorder(node->right);
   }
}
```

```
Postorder(Node * node)
{
   if (node) {
     postorder(node->left);
     preorder(node->right);
     visit(node);
   }
}
```

In-Order again (c++ Overloading)




```
inorder(Node * node)
{
   if (node) {
     inorder(node->left);
     visit(node);
     inorder(node->right);
   }
}
```

2,10,12,13,20,25,29,31

```
void printTabs(int n) {
   while (n--) printf("\t"); }
void InOrdershow(Node *node
                        ,int depth) {
   if (node)    {
   InOrdershow(node->left,depth+1);
   printTabs(depth);
   printf("%d\n",node->key);
   InOrdershow(node->right,depth+1);
   } else { printTabs(depth);
   printf("-_-\n");};
}
void InOrdershow(Node *node){
   InOrdershow(node,0);
}
```

```
                -_-
        2
                -_-
    10
                -_-
        12
                -_-
13
                -_-
        20
                -_-
    25
                    -_-
                29
                    -_-
        31
                -_-
```

Depth-First Pre-Order Traversal Iterative

```
// Bottom up approach vs Divide/Conquer
void iterative_preorder(Node * p)
{
    Stack stack;// construct
    if (p) push(p,stack);
    while(!empty(stack)) {
          p=pop(stack);
          visit(p);
          if (p->right) push(p->right,stack);
                        /*left child pushed */
          if(p->left) push(p->left,stack);
                        /*after right pushed*/
       }
}
```


C++ Template

- A **template** is a blueprint or formula for creating a generic class or a function for producing *Generic* code:

```
template <class T>
class MyClass
{
private:
   T data;
public:
   MyClass(T value) {data=value;}
   void inc() {data+=1;};
   T get() {return data;}
};
```

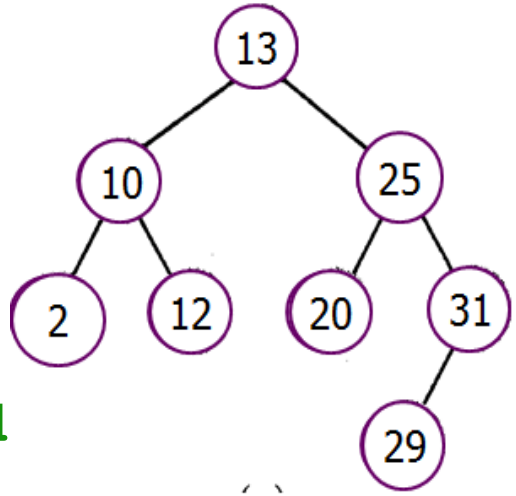
```
int main()
{
   MyClass <int> c1(10);c1.inc();
   MyClass <float> c2(10.1);c2.inc();
   cout << c1.get()<< "\t"<<c2.get();
```

```
11    11.1
```

Depth-First Pre-Order Traversal Iterative (really)

STL Library

Template




```
#include <stack>
void iterative_preorder(Node * p)
{
    stack <Node *> stk;// constructed
    if (p) stk.push(p);
    while(!stk.empty()) {
          p=stk.top();stk.pop(); //POP()->void !
          printf ("%d ",p->key);
          /*right child pushed then left  */
          if (p->right) stk.push(p->right);
         if(p->left) stk.push(p->left);
       }
}
```

13 10 2 12 25 20 31 29

Depth-First Post-Order Traversal Iterative

```
void iterative_postorder(tree_type p)
{
  if (p)  push(p,stack1);
    while(!empty(stack1)) {
      p=pop(stack1);
      push(p,stack2);
      if (p->left) push(p->left,stack1);     /*right child pushed */
      if(p->right) push(p->right,stack1);  /* after left to be  */
    }                                                  /* at the top of the  */
                                                       /* stack  (VRL order) */
   while(!empty(stack2)) {
     p=pop(stack2);  visit(p);
   }
}
```

Reverse the sequence using another stack

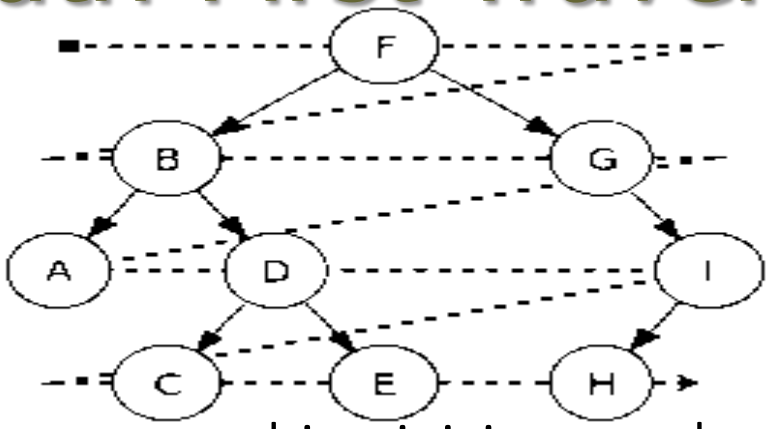
1. Push root to first stack.
2. Loop while first stack is not empty
   2.1 Pop a node from first stack and push it to second stack
   2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack

Depth-First In-Order Traversal Iterative

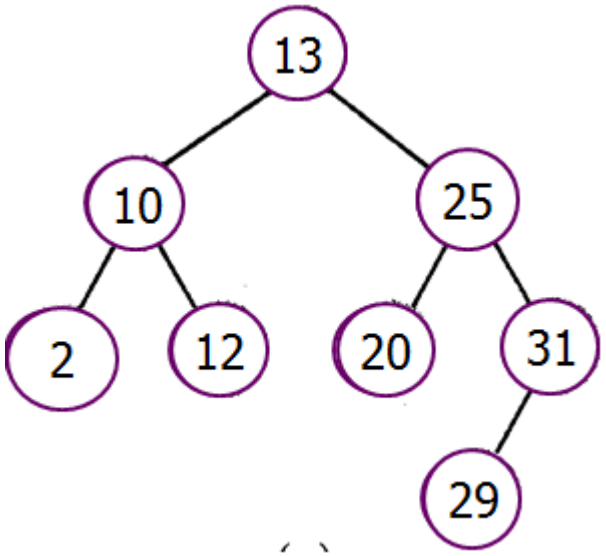
```
void inOrder(struct Node *root)
{
   stack<Node *> s;   Node *curr = root;
   while (curr != NULL || !s.empty())
   {
      while (curr !=  NULL)
      {
         s.push(curr);
         curr = curr->left;
      }
      curr = s.top(); s.pop();
      cout << curr->data << " ";
      curr = curr->right;
   } /* end of while */
}
```

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If stack is not empty then
   a) Pop the top item from stack.
   b) Print the popped item, set current = popped_item->right
   c) Go to step 3.

Breadth-First Traversal



Breadth-first traversal is visiting each node starting from the root and moving down level be level, visiting nodes on each level from left to right.

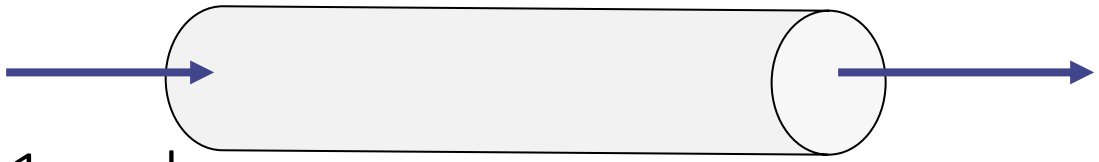


Breadth-first traversal sequence:
13, 10, 25, 2, 12, 20, 31, 29

Breadth-First Traversal

Implementation of breadth-first traversal is quite straightforward when a **queue** is used. After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited.

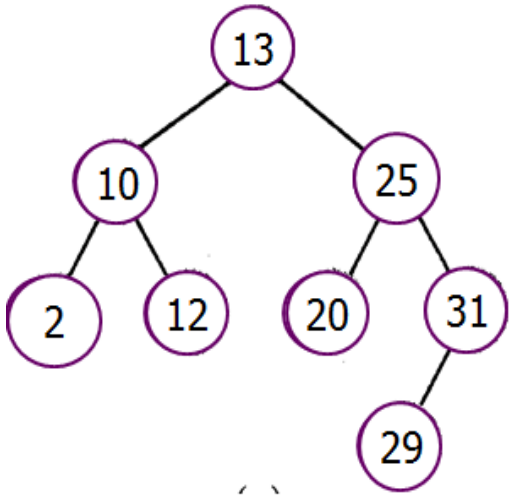
Considering that for a node on level *n*, its children are on level *n*+1, by placing these children at the end of the queue, they are visited after all nodes from level *n* are visited.



Level *n*+1 nodes

Breadth-First Traversal Iterative

```
void breadth_first(tree_type node)
{
   Queue queue;
   if (node) {
      enqueue(node, queue);
      while (!empty(queue)) {
         node=dequeue(queue);
         visit(node);
         if (node->left)
            enqueue(node->left,queue);
         if (node->right)
            enqueue(node->right,queue);
        }
    }
}
```



13 10 25 2 12 20 31 29

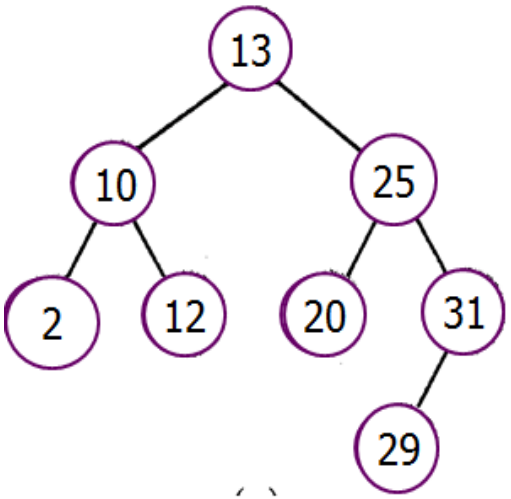
Breadth-First Traversal (recursive)

```
int height(node* node)
{
   if (node == NULL)
      return 0;
   else
   {
     int l = 1+height(node->left);
     int r = 1+height(node->right);
      return l> r ? l :r;
   }
}
void printGivenLevel(node* root, int level)
{
   if (root == NULL) return;
   if (level == 1) cout << root->data << " ";
   else //if (level > 1)
   {
      printGivenLevel(root->left, level-1);
      printGivenLevel(root->right, level-1);
   }
}
```

```
/*Function to print all nodes at a given
    level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
   print(tree->data);
else if level greater than 1, then
   printGivenLevel(tree->left, level-1);
   printGivenLevel(tree->right, level-1);

/*Function to print level order traversal
    of tree*/
printLevelorder(tree)
  for d = 1 to height(tree)
     printGivenLevel(tree, d);
```

Breadth-First Traversal (recursive)

```
void printLevelOrder
                        (node* root)
{
    int h = height(root);
    int i;
    for (i = 1; i <= h; i++)
        printGivenLevel(root, i);
}
```

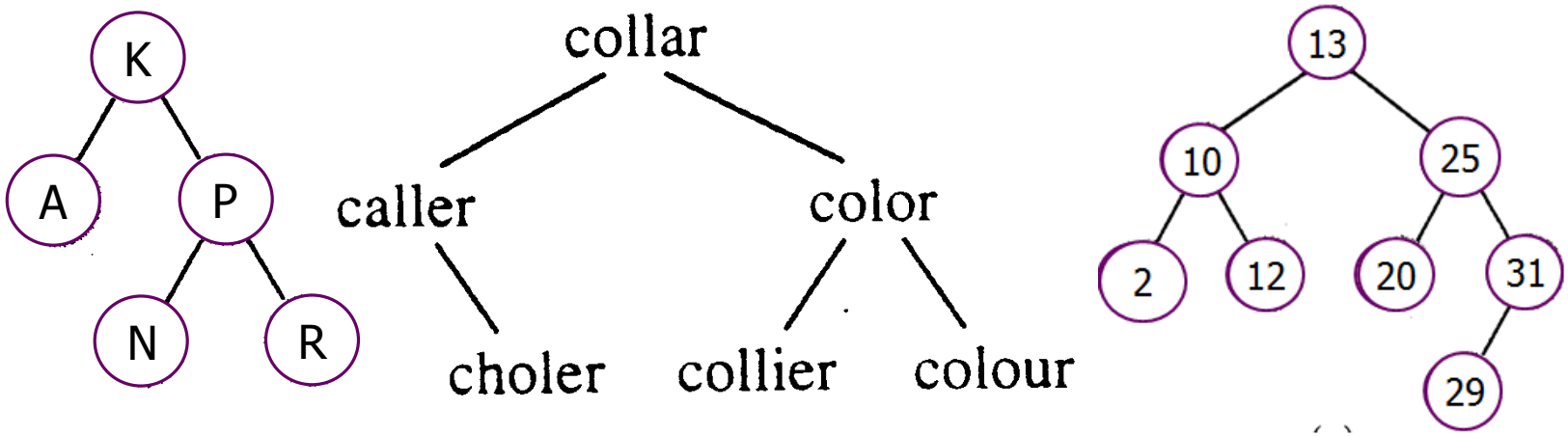



```
/*Function to print all nodes at a given
    level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);

/*Function to print level order traversal
    of tree*/
printLevelorder(tree)
  for d = 1 to height(tree)
     printGivenLevel(tree, d);
```

13 10 25 2 12 20 31 29

Binary Trees (BST)

A **binary search tree** has the following property: if a node *n* of the tree contains the value *v*, then all values stored in its left sub-tree are less than *v*, and all values stored in the right sub-tree are greater than *v*.

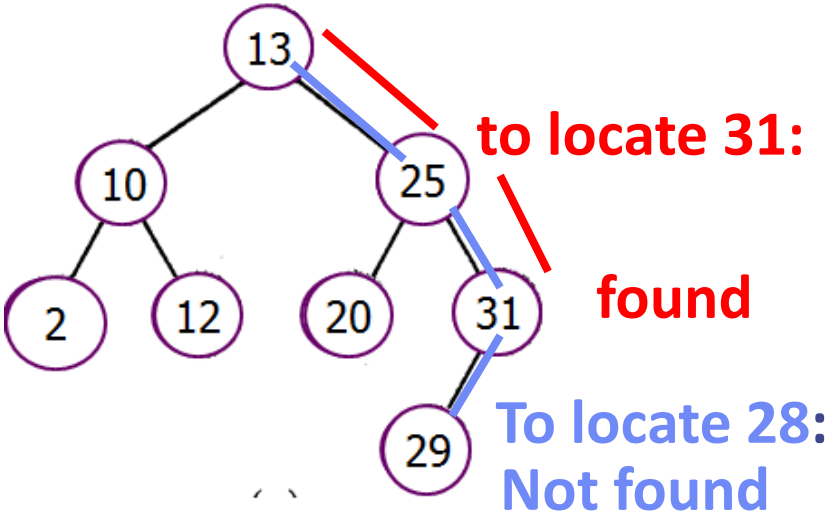


Examples of binary search trees

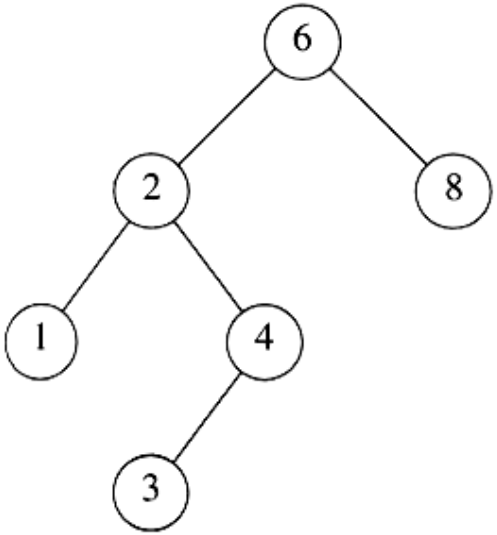
Binary Search Trees

An algorithm for locating an element in the binary search tree is quite straightforward. For every node, compare the key to be located with the value stored in the node (**starting from root**, of course) currently pointed at. If the key is **less than** the value, go to the **left subtree** and try again. If it is **greater than** that value, try the **right subtree**.

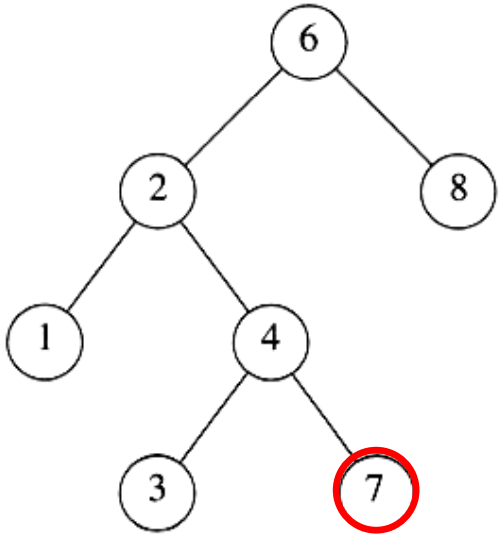
Maximum number of searches
$\propto$ the height of the tree



Binary Search Trees



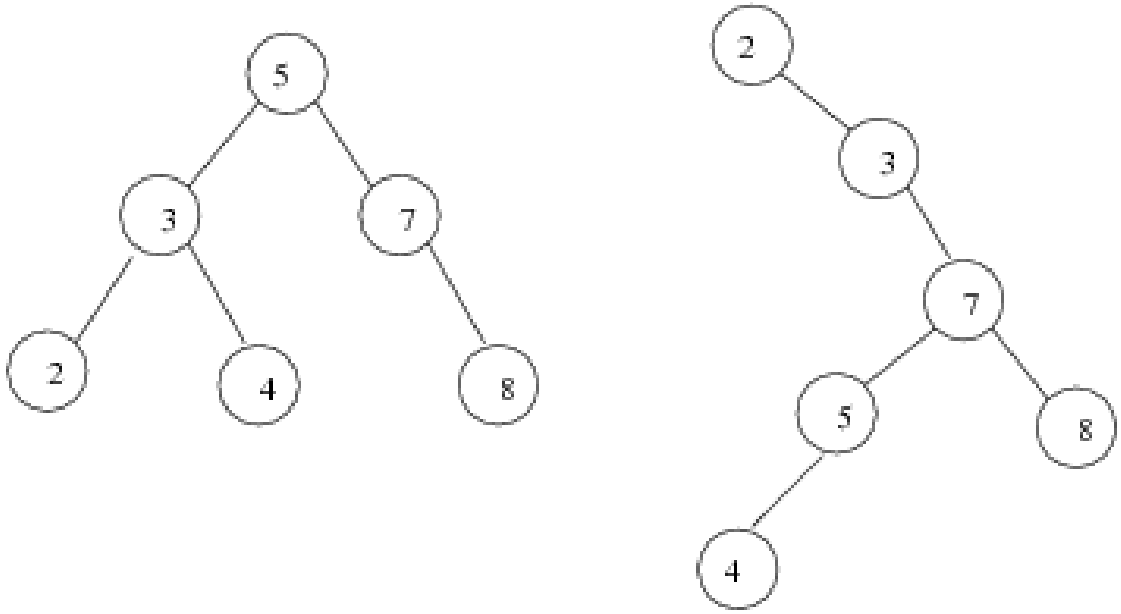
A binary search tree



Not a binary search tree

Binary search trees (Not Unique)

Two binary search trees representing the same set:



- Average depth of a node is O(log N); maximum depth of a node is O(N)

Iterative Searching

```
Node * search(Node *node, TYPE key)
{
   while (node)
     if (key = = node->key)
            return node;
     else if (key < node->key)
            node  =node->left;
            else node = node->right;
    return NULL;
}
```

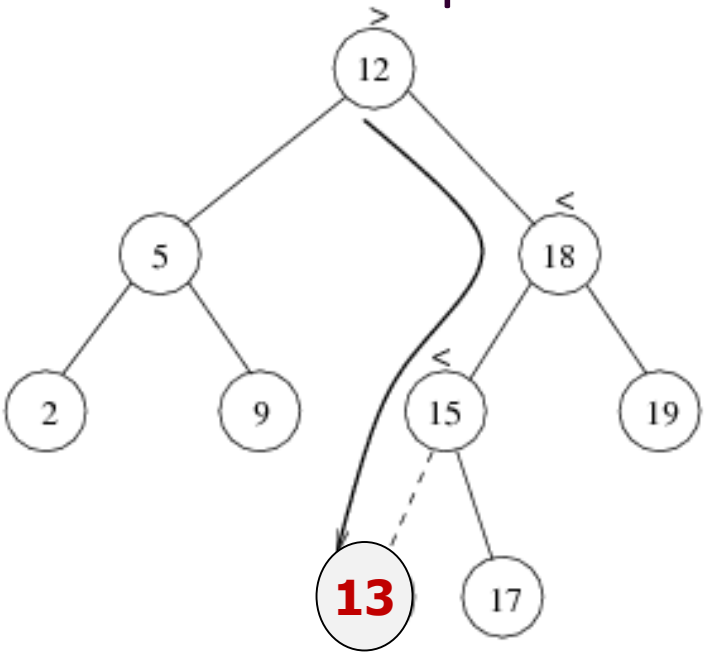

Recursive Searching

```
Node *search(Node * node, TYPE key)
{
   if (node) {
     if (key == node->key) return node;
     if (key < node->key)
            return search(node->left,key);
     /*else*/
     return search(node->right,key);
    }
    else return NULL;
}
```


Insert into BST

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed

Insert 13



- Time complexity = O(height of the tree)

Insertion (recursive)

```
node* insert(node *Node, int key){
    /* Normal BST insertion */
   if (node == NULL)
      return(newNode(key));
   if (key < node->key)
      node->left  = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);
   else // Equal keys are not allowed in BST
      return node;
   return node;
}
```


Insertion (iterative)

```
Node * insertBstIterative(Node * root, int value){
    Node *newNode=new Node(value);
    Node *curr=root,*parent=NULL;
    /*Find a node with a convenient
                                free child!*/
    while(curr){
       parent=curr;
       if (value<curr->key) curr=curr->left;
       else curr=curr->right;
    }
    if (!parent) return root =newNode;
    if (value<parent->key) parent->left=newNode;
    else parent->right=newNode;
    return root;
  }
```

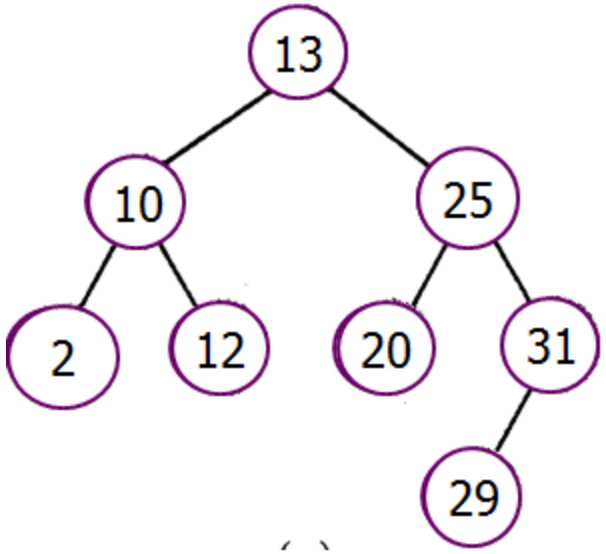
Start with curr=root
downward to find the right place of the new node,curr becomes null keep track of its parent
If parent =NULL the original tree was empty, return a tree with the new node
Else link to the parent

Searching a Binary Trees

The complexity of searching is measured by the **number of comparisons** performed during the searching process. This number depends on the number of nodes encountered on the **unique path leading from the root to the node** being searched for.

Therefore, the complexity is the length of the path leading to this node plus 1. It depends on the **shape of the tree and the position of the node in the tree**.

Searching a Binary Trees



Search number	# of comparisons	internal path length
13	1	0
10	2	1
2	3	2
12	3	2
25	2	1
20	3	2
31	3	2
29	4	3

O(# of comparisons)=O(internal path length)

Searching a Binary Trees

The (total) *internal path length* (IPL) is the sum of all path lengths of all nodes, which is calculated by summing $\sum(i\text{-}1)l_i$ over all levels i, where l_i is the number of nodes on level i.

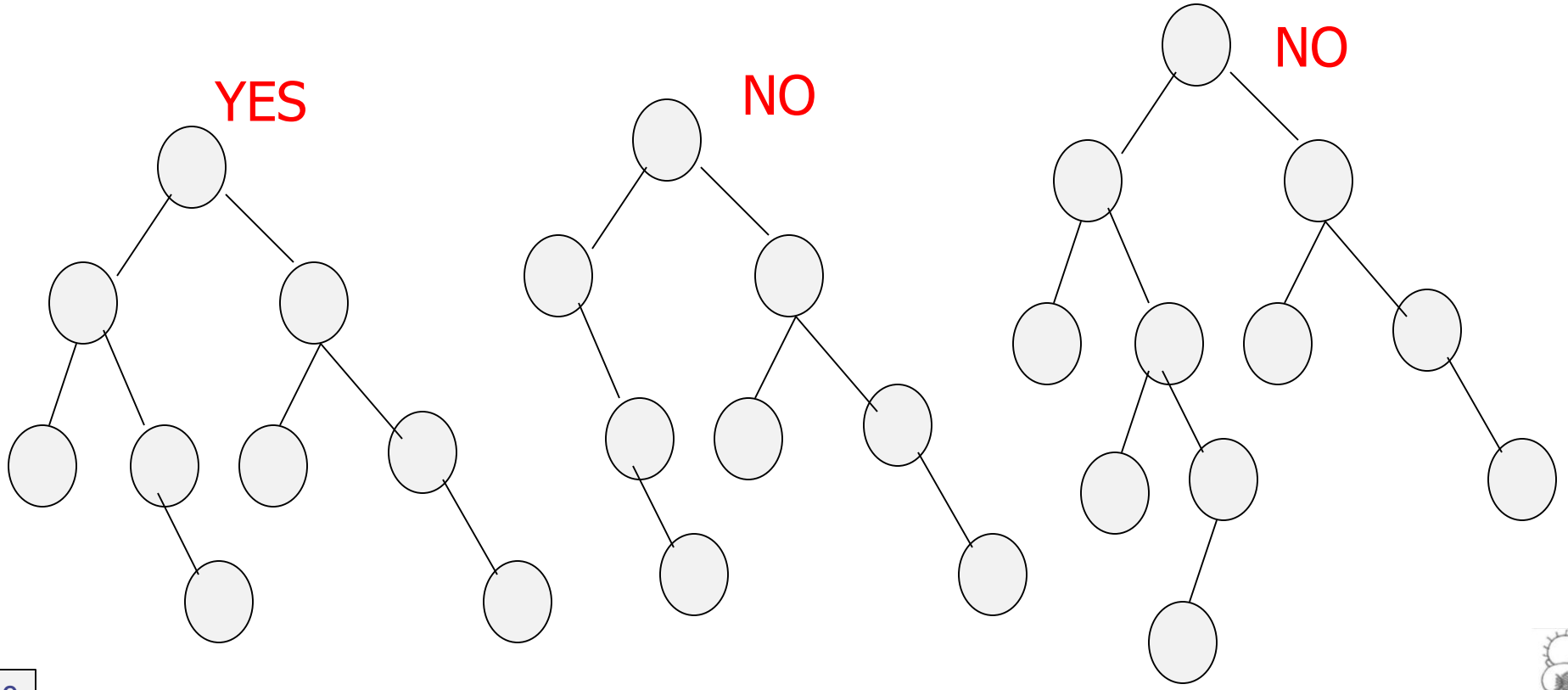
average (internal) path length=IPL/n

In the worst case, when the tree turns into a linked list,

$$path_{worst} = \frac{1}{n}\sum_{i=1}^{n}(i-1) = \frac{n-1}{2} = O(n)$$

Searching a Binary Trees

The best case occurs when all leaves in the tree of height *h* are in at most some (example two) levels and only nodes in the next to the last level can have one child.



Searching a Binary Trees

For a complete binary tree of height h, IPL=

$$\sum_{i=1}^{h-1}(i\times 2^i)=(h-2)\times 2^h+2$$

The total number of nodes in a complete binary tree with height *h* is $n=2^h-1$. Therefore,

$$path_{best}\le\frac{IPL_{complete}}{n}=\frac{(h-2)2^h+2}{2^h-1}\approx h-2=\log_2(n+1)-2$$

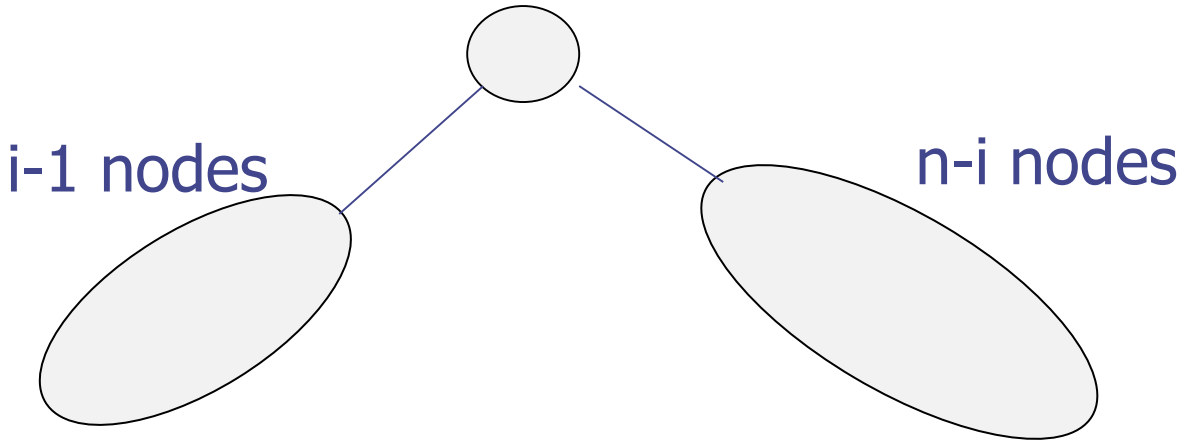
$O(path_{best})=O(\log n)$

The average case is somewhere between $(n-1)/2$ and $\log(n+1)-2$. Is a search for a node in an average position in a tree closer to $O(n)$ or $O(\log n)$? First, the average shape of the tree has to be represented computationally.

Searching a Binary Trees

Assume that the tree contains nodes 1 through *n*. If *i* is the root, then its left subtree has *i*-1 nodes, and its right subtree has *n*-*i* nodes. If $path_{i-1}$ and $path_{n-i}$ are average paths in these subtrees, then the average path of this tree is

$$path_n(i) = \frac{(i-1)(path_{i-1}+1)+(n-i)(path_{n-i}+1)}{n}$$



Searching a Binary Trees

The left subtree can have any number of nodes {from 1(i=1) to n-1 (i=n)}. Therefore, the average path length of an average tree is obtained by averaging all values of pathn(i) over all values of i. This gives this formula

$$path_n = \frac{1}{n}\sum_{i=1}^{n} path_n(i) =$$

$$\frac{1}{n^2}\sum_{i=1}^{n}((i-1)(path_{i-1}+1)+(n-i)(path_{n-i}+1)) =$$

$$\frac{2}{n^2}\sum_{i=1}^{n-1} i(path_i+1) = 2\ln n = 2(\ln 2)(\log n) = 1.386\log n = O(\log n)$$

However inserting the sequences: 1,2,3,4 …, 10,9,8,7…, or 100,1,99,2,98,3,…. Makes a O(n) search !

Iteration versus Recursion

```
Node * find_min(Node *x){
      if (x==NULL) return NULL;
      while (x->left != NULL) x=x-
>left;
      return x;
}

Node * find_min(Node *x){
      if (x==NULL) return NULL;
      if(x->left == NULL) return x;
      return find_min(x->left);
}
```


delete

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

delete

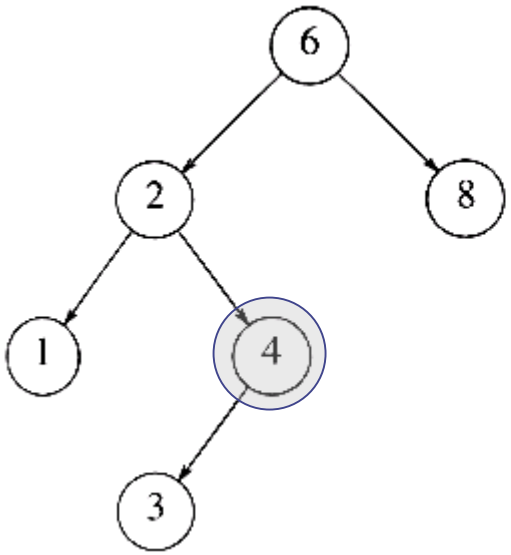
Three cases:

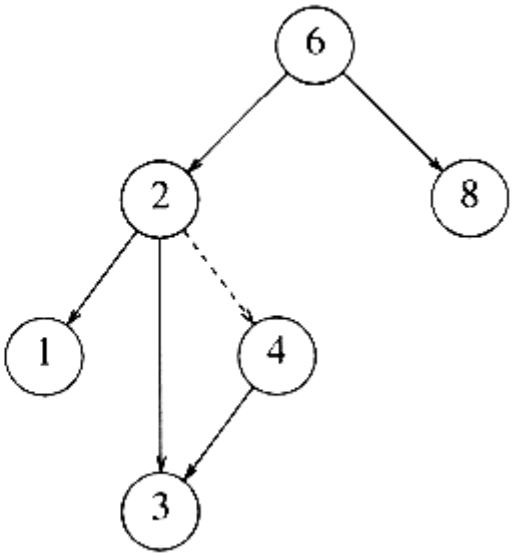
(1) the node is a leaf

- Delete it immediately

(2) the node has one child

- Adjust a pointer from the parent to bypass that node



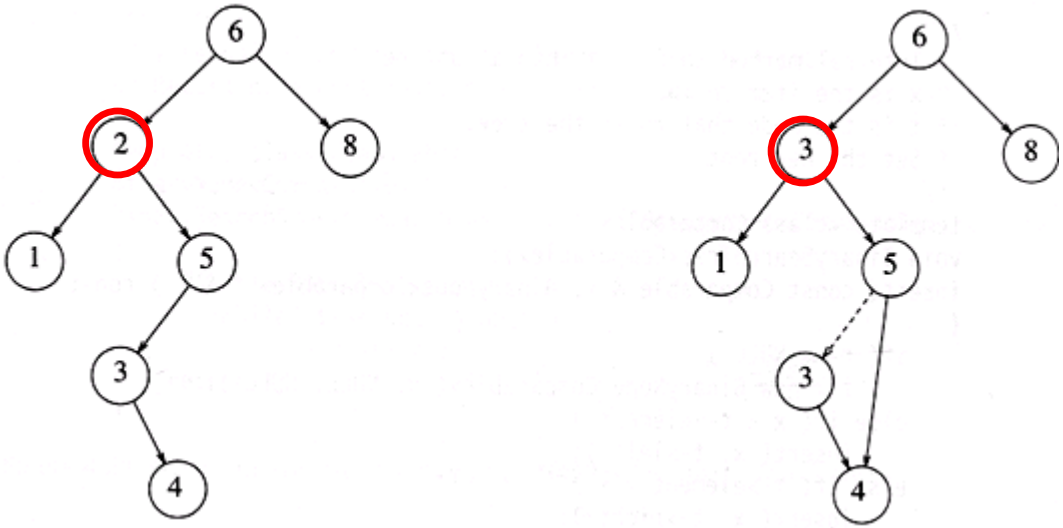


delete

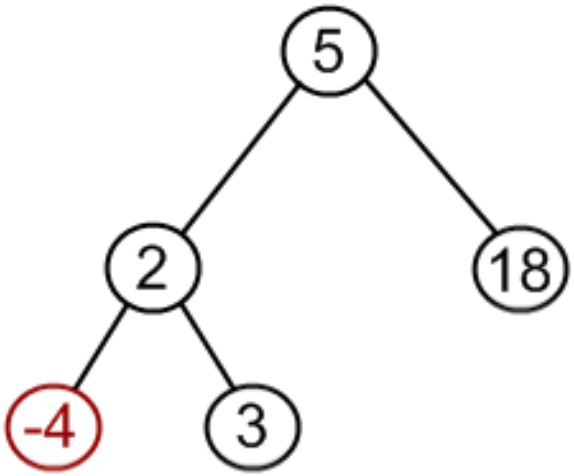
(3) the node has 2 children

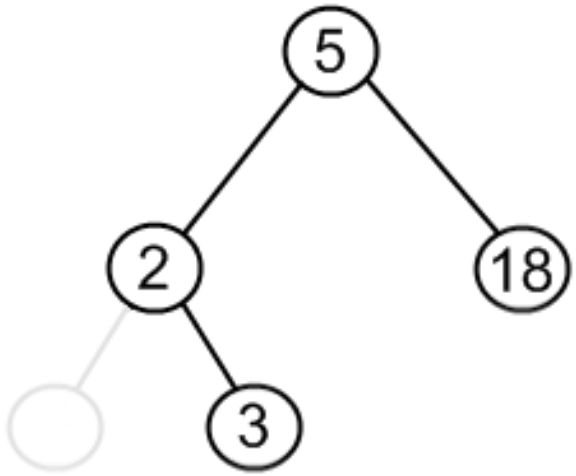
- replace the key of that node with the minimum element at the right subtree
- delete the minimum element
  - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

Time complexity = O(height of the tree)

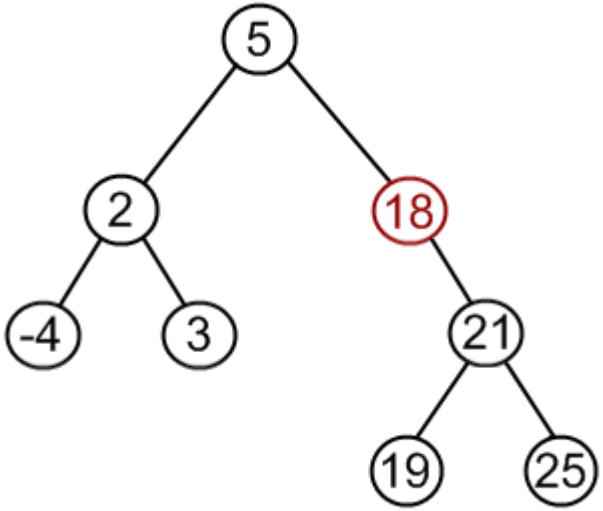


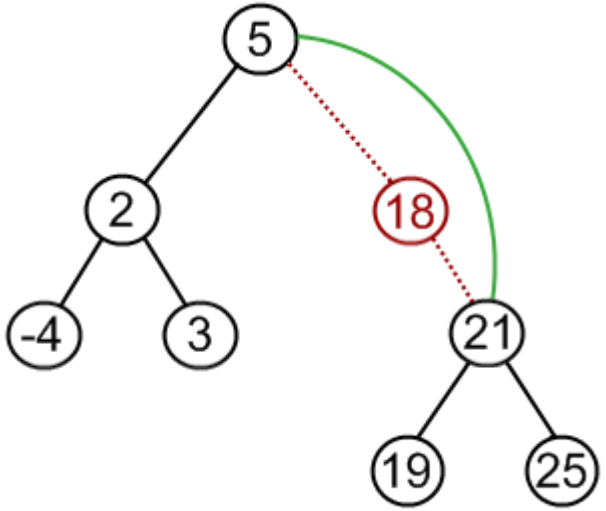
Delete (case 1)

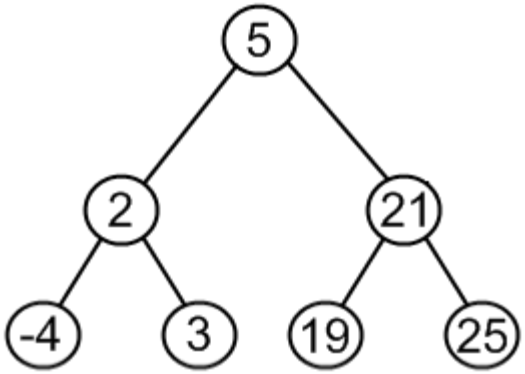




Delete (case 2)

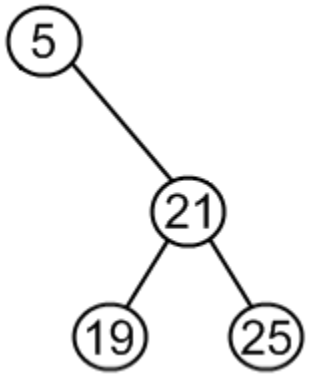


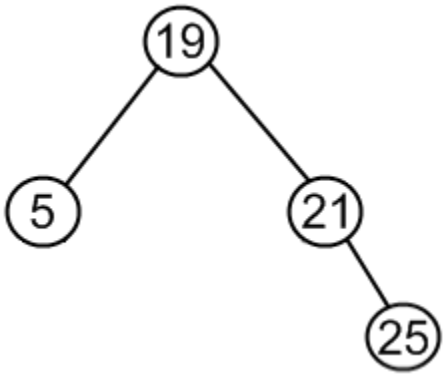




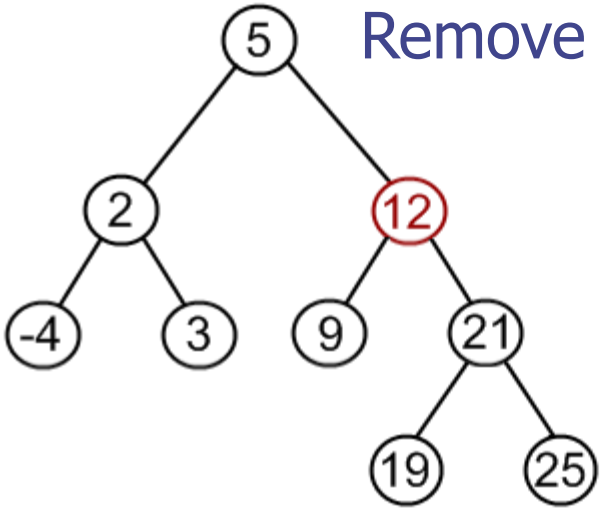
Delete (case 3)

- We can transform a BST to another one with the same data ordering by:
  - choose minimum element from the right subtree (19 in the example);
  - replace 5 by 19;
  - hang 5 as a left child.

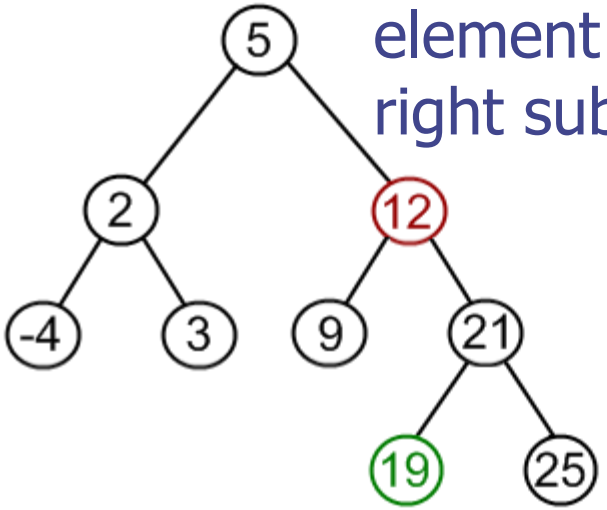




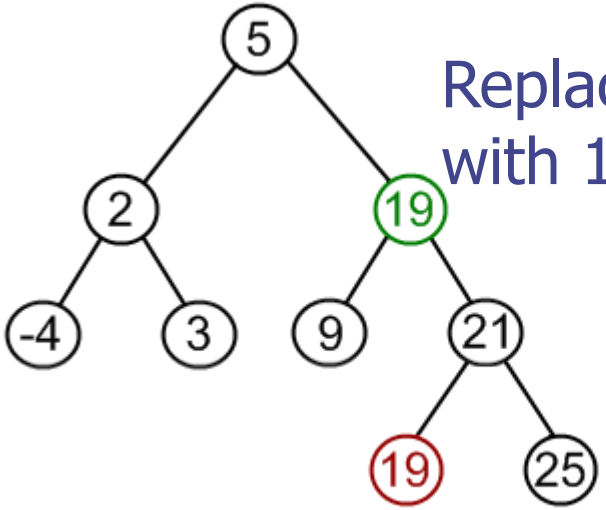
Delete (case 3)



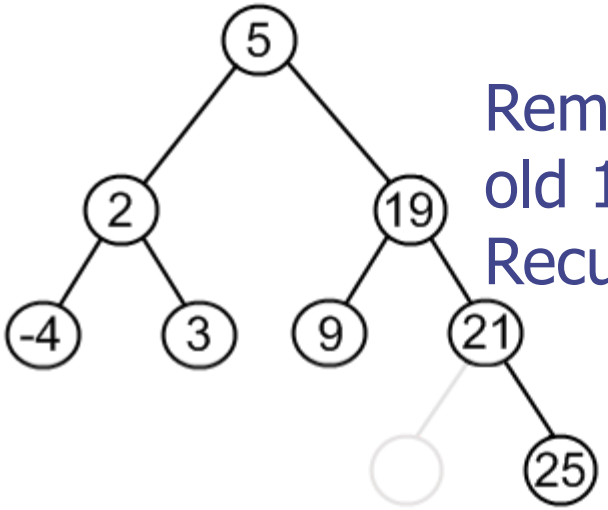
Remove 12



Find minimum element in the right subtree



Replace 12 with 19



Remove the old 19 Recursively

Delete

```
Node* deleteNodeBST(Node* root, int key)
{
    // base case
    if (root == NULL) return root;
    // The key to be deleted is smaller than the root's
    // key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNodeBST(root->left, key);
    // The key to be deleted is greater than the root's
    // key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNodeBST(root->right, key);
    // Key is same as root's key, then This is the node
    // to be deleted
    else
    .....
    {
```

Delete

```
else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            Node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            Node *temp = root->left;
            free(root);
            return temp;
        }
        // node with two children: Get the inorder successor
        // (smallest in the right subtree)
        Node* temp = minValueNode(root->right);
        // Copy the inorder successor's content to this node
        root->key = temp->key;
        // Delete the inorder successor
        root->right = deleteNodeBST(root->right, temp->key);
    }
    return root;
}
```

Some C++ (Optional)

Node, Constructors, Insert Recursive

```
#include <iostream>
#include<stdio.h>
#include<stdlib.h>
using namespace std;
class Node {
  friend class BSTree;
    int key;
    Node *left,*right;
    Node (int value){ key=value;left=right=NULL;} };
class BSTree {
private: // Not needed, Default
    Node * root;
    Node* insertBst(Node* node, int key){
     if (node){
       if (key<node->key)
          node->left=insertBst(node->left,key);
       else node->right=insertBst(node->right,key);
       return node;}
     else return new Node(key);
    }
```

Some C++ (Optional)

Recursive inorder, find min and max

```
void inorder(Node *node){
        if(node) {
            inorder(node->left);
            printf("%d ",node->key);
            inorder(node->right);
        }
    }
    Node* findMax(Node *n)
    {
        if(n == NULL)
            return NULL;
        else if(n->right == NULL) return n;
        else return findMax(n->right);
    }
     Node* findMin(Node *n)
    {
        if(n == NULL) return NULL;
        else if(n->left == NULL) return n;
        else return findMin(n->left);
    }
```

Some C++ (Optional)

Remove Node

```
Node* removeNode(Node* node,int x) {
  Node* temp;
   if(node == NULL) return NULL;
   if(x < node->key) node->left = removeNode(node->left,x);
   else if(x>node->key)
                        node->right=removeNode(node->right,x);
   else if(node->left && node->right) { // full Node
        temp = findMin(node->right);
        node->key = temp->key;
        node->right = removeNode(node->right,node->key); }
        else{
            temp = node;
            if (node->left == NULL)   node = node->right;
            else if(node->right == NULL)  node = node->left;
            delete temp;//free(temp);
       }
       return node;
   }
```

Some C++ (Optional)

Show Tree (In-order recursive) /PUBLIC declaration

```
  void printTabs(int n) {
        while (n--) printf("\t"); }
  void InOrdershow(Node *node,int depth){
        if (node)
        {
        InOrdershow(node->left,depth+1);
        printTabs(depth);printf("%d\n",node->key);
        InOrdershow(node->right,depth+1);
        } else {printTabs(depth);printf("-_-\n");};
    }
public:
    BSTree(Node *node=NULL) {root=node;} // constructor
    Node *getRoot() {return root;}
    Node * insertBst(int key){
        return root=insertBst(root,key);
    }
    void inorder(){inorder(root);printf("\n");    }
    Node*findMax(){return findMax(root);}
    Node*findMin(){return findMin(root);}
```

Some C++ (Optional)

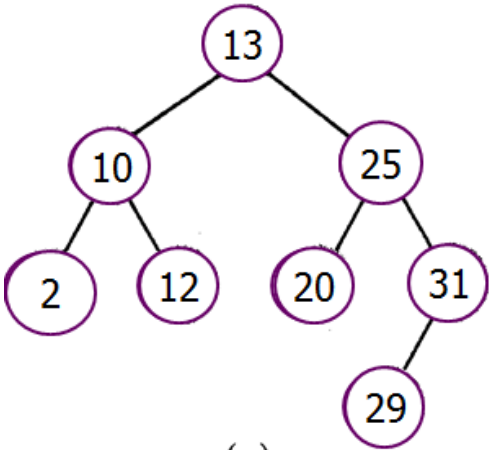
Interactive Insert

```
Node* removeNode(int x){
    return removeNode(root,x);
}
void InOrdershow(){
     InOrdershow(root,0);
}
Node * insertBstIterative(int value){
     Node *newNode=new Node(value);
     Node *curr=root,*parent=NULL;
     //Find a leaf !
     while(curr){
          parent=curr;
          if (value<curr->key) curr=curr->left;
          else curr=curr->right;
     }
     if (!parent) return root =newNode;
     if (value<parent->key) parent->left=newNode;
     else parent->right=newNode;
     return root;
}
```

Some C++ (Optional)

Main()

```
int main()
{
    BSTree *tree=new BSTree();
    int arr[]={13,10,2,12,25,31,29,20};
    int n=sizeof(arr)/sizeof(int);
    for (int i=0;i<n;i++)
        tree->insertBst(arr[i]);
    tree->inorder();tree->InOrdershow();
    tree->removeNode(13);tree-
>inorder();
    return 0;
}
```




```
2 10 12 13 20 25 29 31
              -_-
         2
              -_-
    10
              -_-
         12
              -_-
13
              -_-
         20
              -_-
    25
                   -_-
              29
                   -_-
         31
              -_-
2 10 12 20 25 29 31
```

Enough / Skip

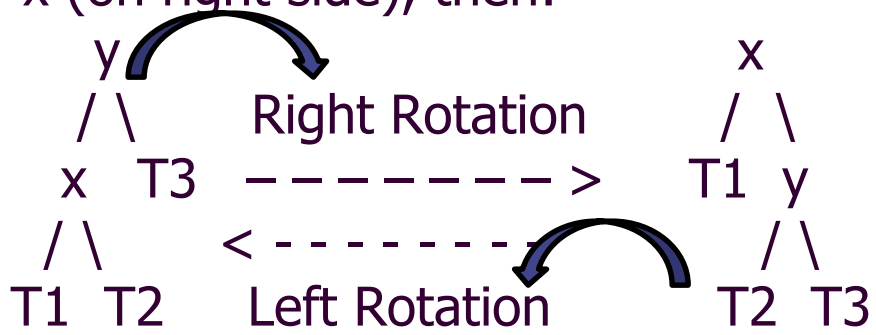
Introducing AVL-tree (The Soviets: Georgy Adelson-Velsky, Evgenii Landis)

- Basic Idea: balance factor= left subtree height – right subtree height:

```
struct Node* newNode(int key)
{
   Node* node = (Node*)
            malloc(sizeof(struct Node));
   node->key   = key;
   node->left   = NULL;
   node->right  = NULL;
   node->height = 1;  // new node is initially added at leaf
                              // Height of NULL is assumed Zero
   return(node);
}

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


Introducing AVL-tree (The Soviets: Georgy Adelson-Velsky, Evgenii Landis)

- Basic Idea: balance factor= left subtree height – right subtree height
- If T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side), then:



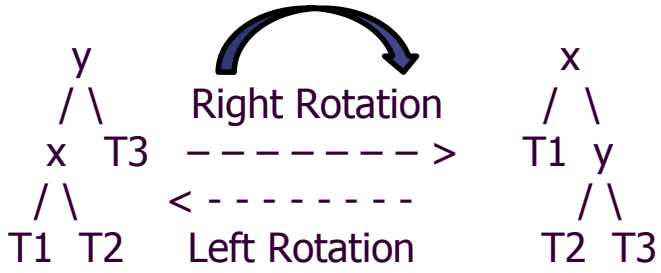
- Keys in both of the above trees follow the following order:
  keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
- So BST property is not violated anywhere.

BST routine

```
struct Node* insert(struct Node* node, int key)
{
if (node == NULL)
        return(newNode(key));
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;
    // Story has a continuation
    node->height = 1 + max(height(node->left),
                           height(node->right));
    int balance = getBalance(node);
...
}
```

Introducing AVL-tree (The Soviets: Georgy Adelson-Velsky, Evgenii Landis)

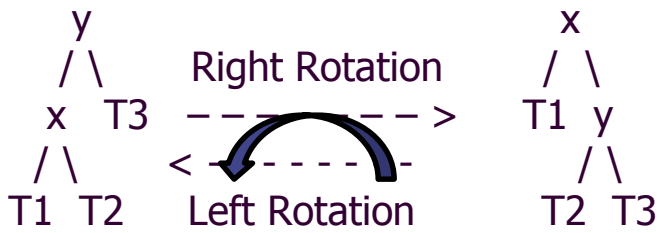
- If T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side), then:




```
Node *rightRotate( Node *y)
{
    Node *x = y->left; Node *T2 = x->right;
    // Perform rotation
    x->right = y; y->left = T2;
    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;
    return x; // Return new root
}
```


Introducing AVL-tree (The Soviets: Georgy Adelson-Velsky, Evgenii Landis)

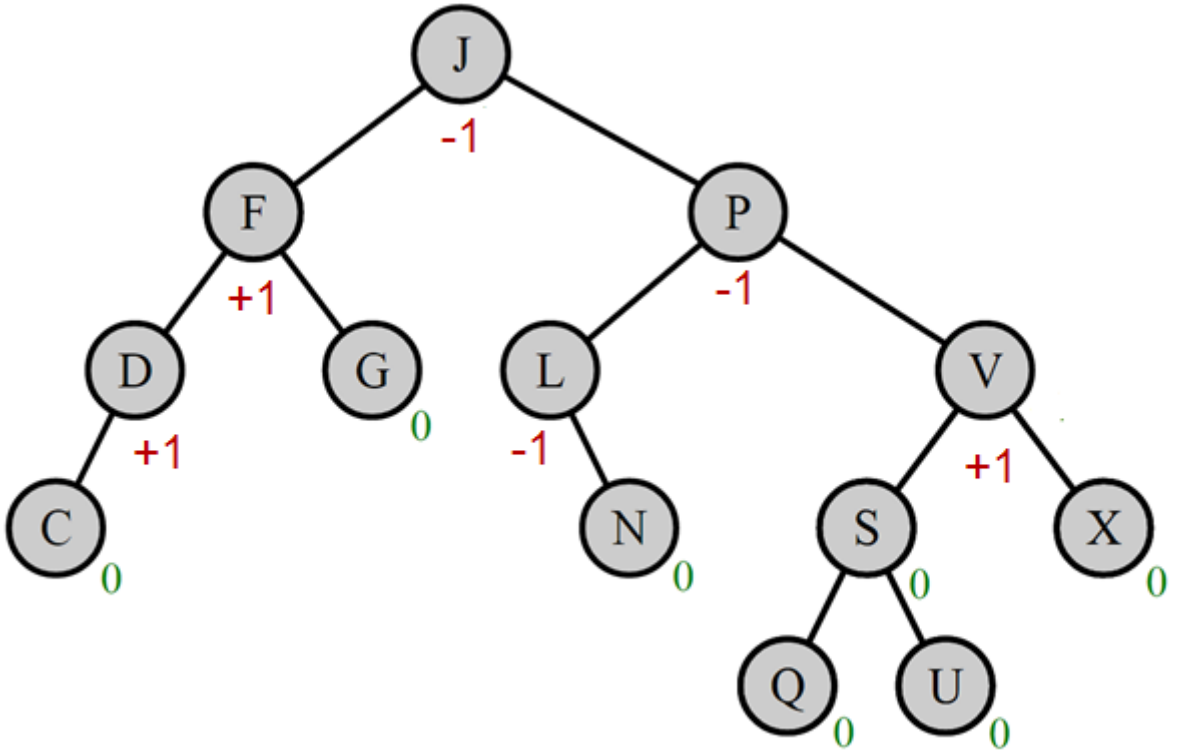
- If T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side), then:




```
Node * leftRotate ( Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;
    // Perform rotation
    y->left = x;  x->right = T2;
    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
    return y;// Return new root
}
```

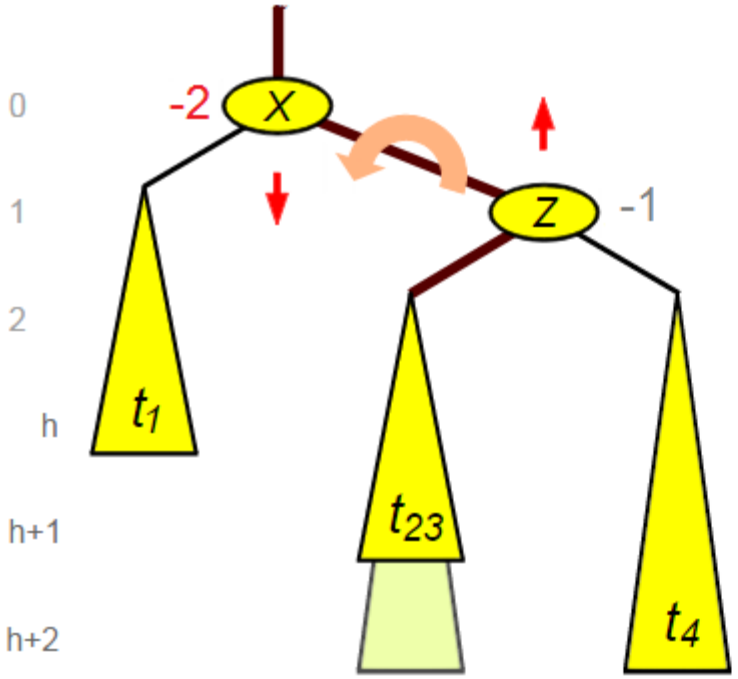
Introducing AVL-tree (The Soviets: Georgy Adelson-Velsky, Evgenii Landis)

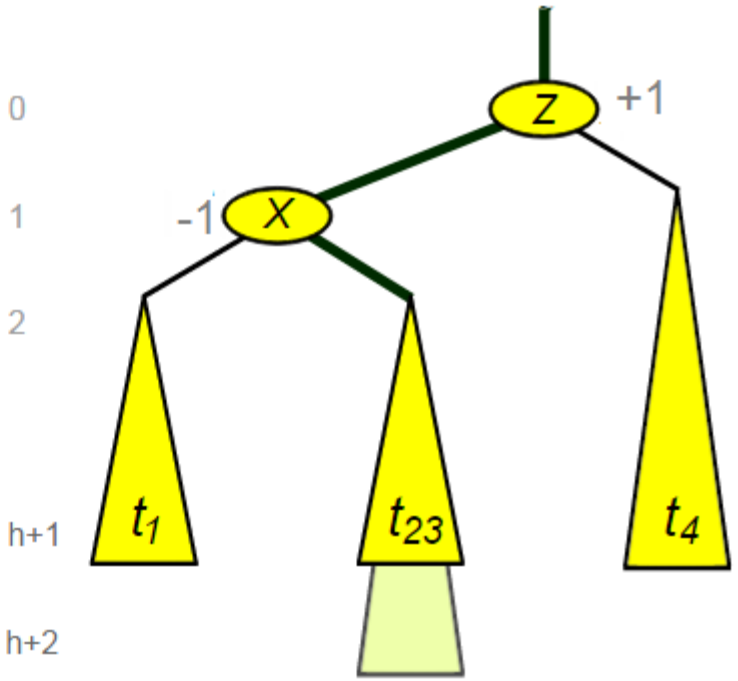
- Basic Idea: balance factor= left subtree height – right subtree height – KEEP it between [-1,1] for all nodes



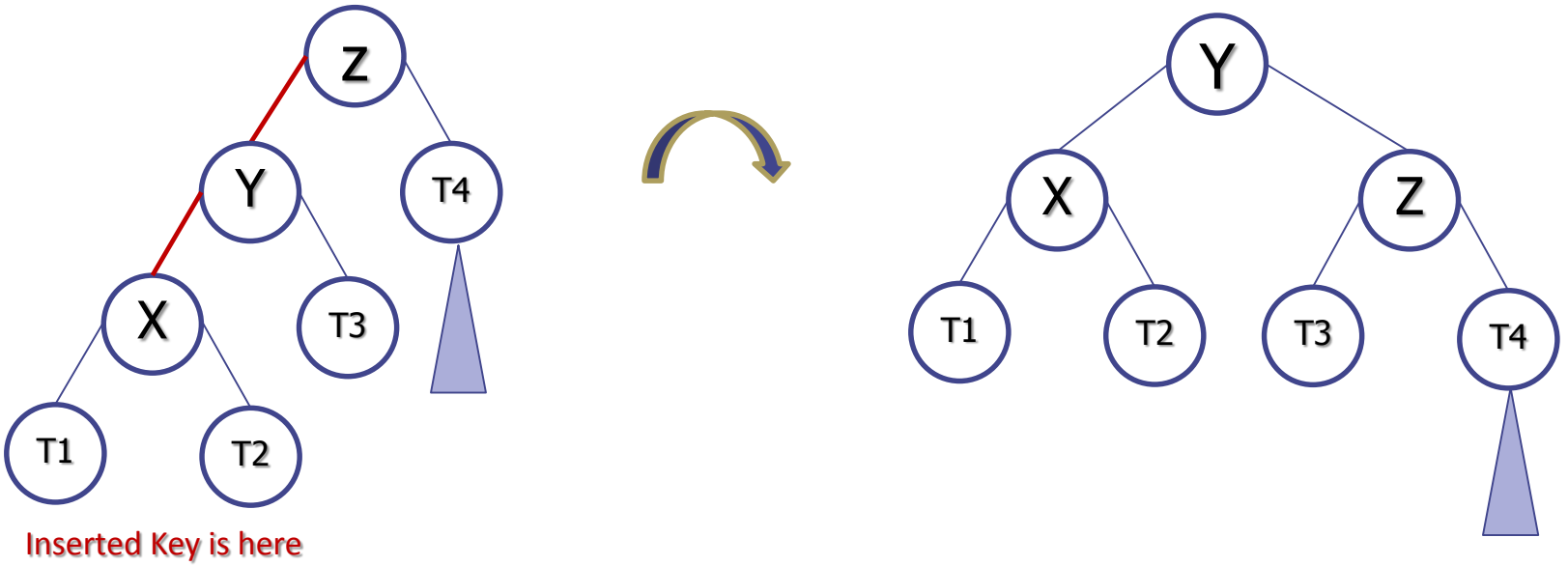
Introducing AVL-tree (The Soviets: Georgy Adelson-Velsky, Evgenii Landis)

- Basic Idea: balance factor= left subtree height – right subtree height – KEEP it between [-1,1] for all nodes



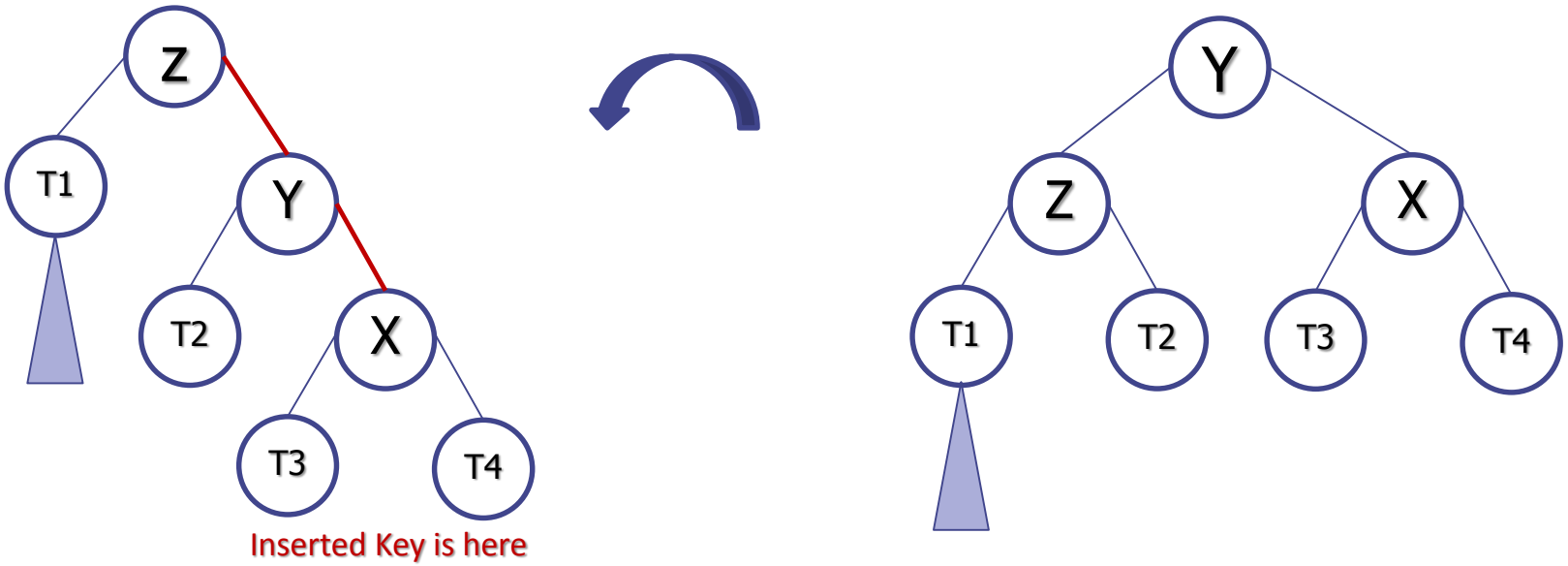


Insertion Violation: Left-Left case



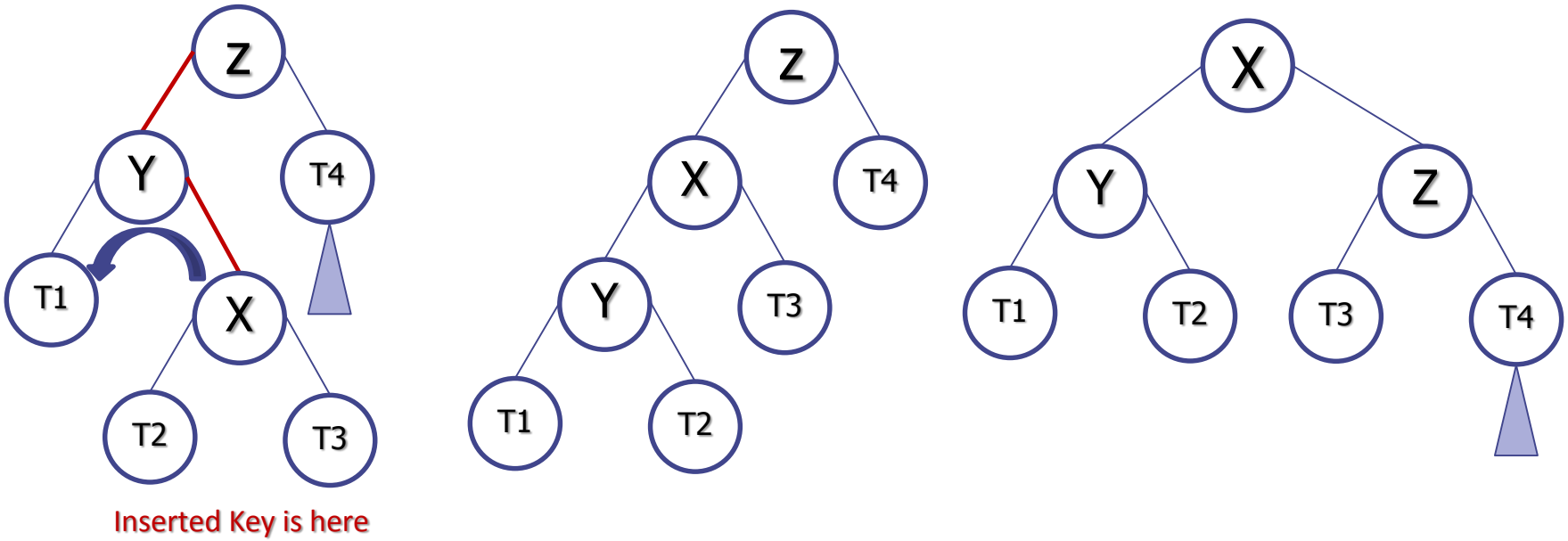

```
if (balance > 1 && key < node->left->key)
{if (VERBOSE) printf("LL:");
        return rightRotate(node); }
```

Insertion Violation: Right-Right case



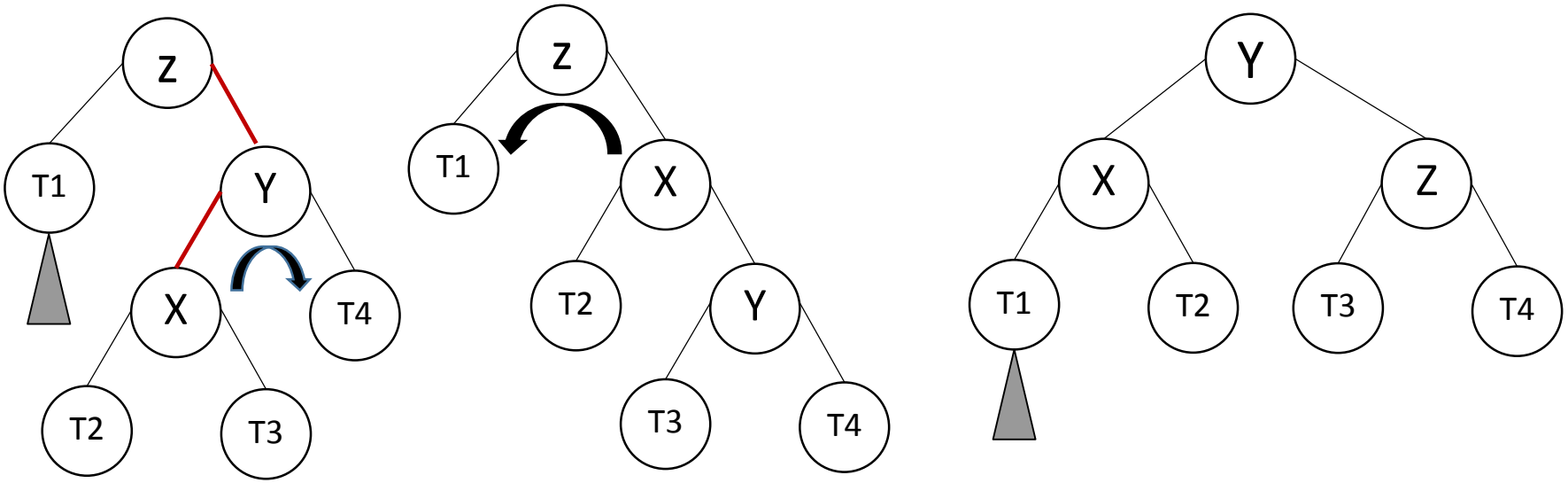

```
if (balance < -1 && key > node->right->key)
        { if (VERBOSE) printf("RR:");
         return leftRotate(node);
        }
```


Insertion Violation: Left-Right case




```
if (balance > 1 && key > node->left->key) {
    if (VERBOSE) printf("RR:");
    node->left =  leftRotate(node->left);
    return rightRotate(node);}
```

Insertion Violation: Right-Left Case



Inserted Key is here

```
if (balance < -1 && key < node->right->key) {
    if (VERBOSE) printf("RL:");
    node->right = rightRotate(node->right);
    return leftRotate(node);  }
```

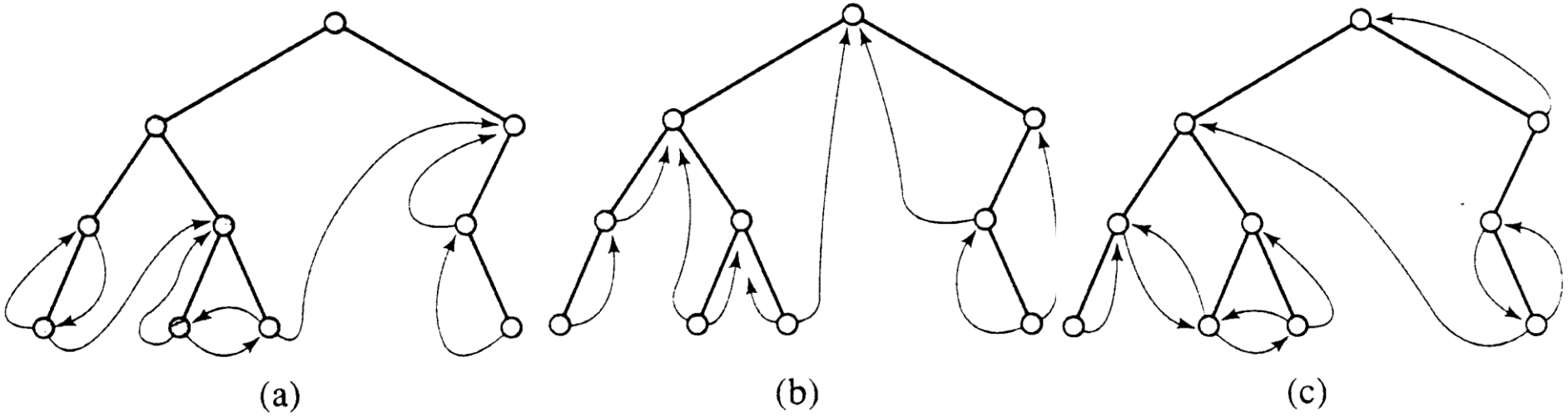

Example

```
root = insert(root, 7);
root = insert(root, 6);
root = insert(root, 5);
root = insert(root, 10);
root = insert(root, 100);
root = insert(root, 20);
root = insert(root, 30);
root = insert(root, 40);
root = insert(root, 50);
root = insert(root, 25);
root = insert(root, 300);
root = insert(root, 400);
root = insert(root, 500);
root = insert(root, 500);
root = insert(root, 720);
root = insert(root, 1);
```

```
LL:     Rotation right around 7
RR:     Rotation left around 7
RR:     Rotation left around 6
LR:     Rotation left around 20
        Rotation right around 100
LR:     Rotation left around 40
        Rotation right around 100
RR:     Rotation left around 10
RR:     Rotation left around 100
RR:     Rotation left around 50
RR:     Rotation left around 400
```


Threaded Binary Tree

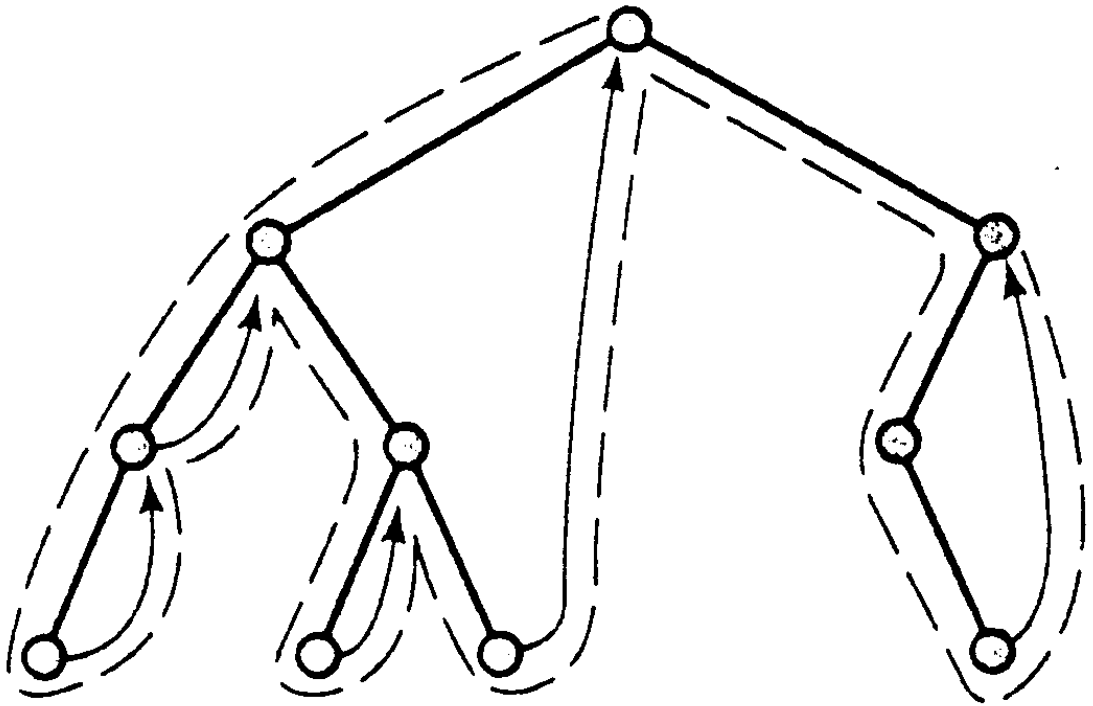
Threaded Trees



Threaded trees to be used for (a) preorder, (b) inorder, and (c) postorder traversals.

Threaded Binary Tree

Only **an extra bit** is needed in tree structure.



```
Struct node_rec {
eltype key;
unsigned successor:1;
struct node_rec *left,
                *right;
}
typedef struct node_rec
      *threaded_tree_type;
```

Inorder traversal's path in threaded tree.

Other types of Binary trees

- Splay Tree
- Tries
- Radix Tree
- **Heap**
- Fibonacci Heap/tree
- Binomial Heap/tree
- Leftist Heap
- Skew Heap

Thanks