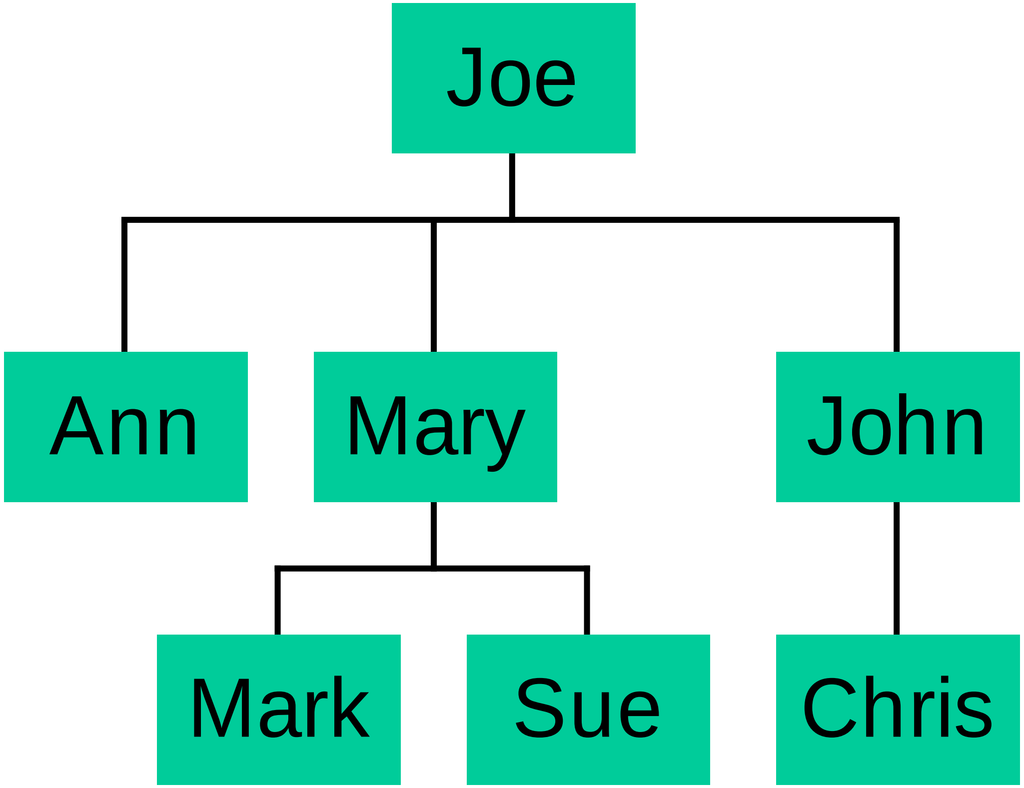
Ch 4: **Trees**

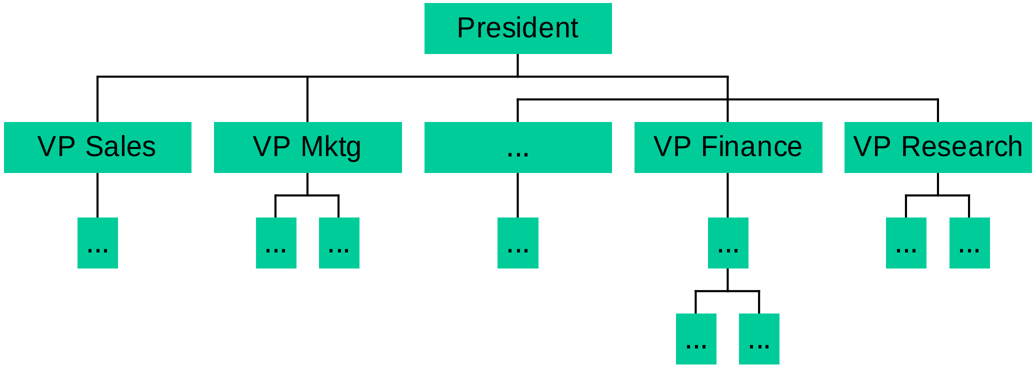
“it’s a jungle out there...”

**“I think that I will never see
a linked list useful as a tree;
Linked lists are used by everybody,
but it takes real smarts to do a tree”**

Trees: examples
(*Family trees*)

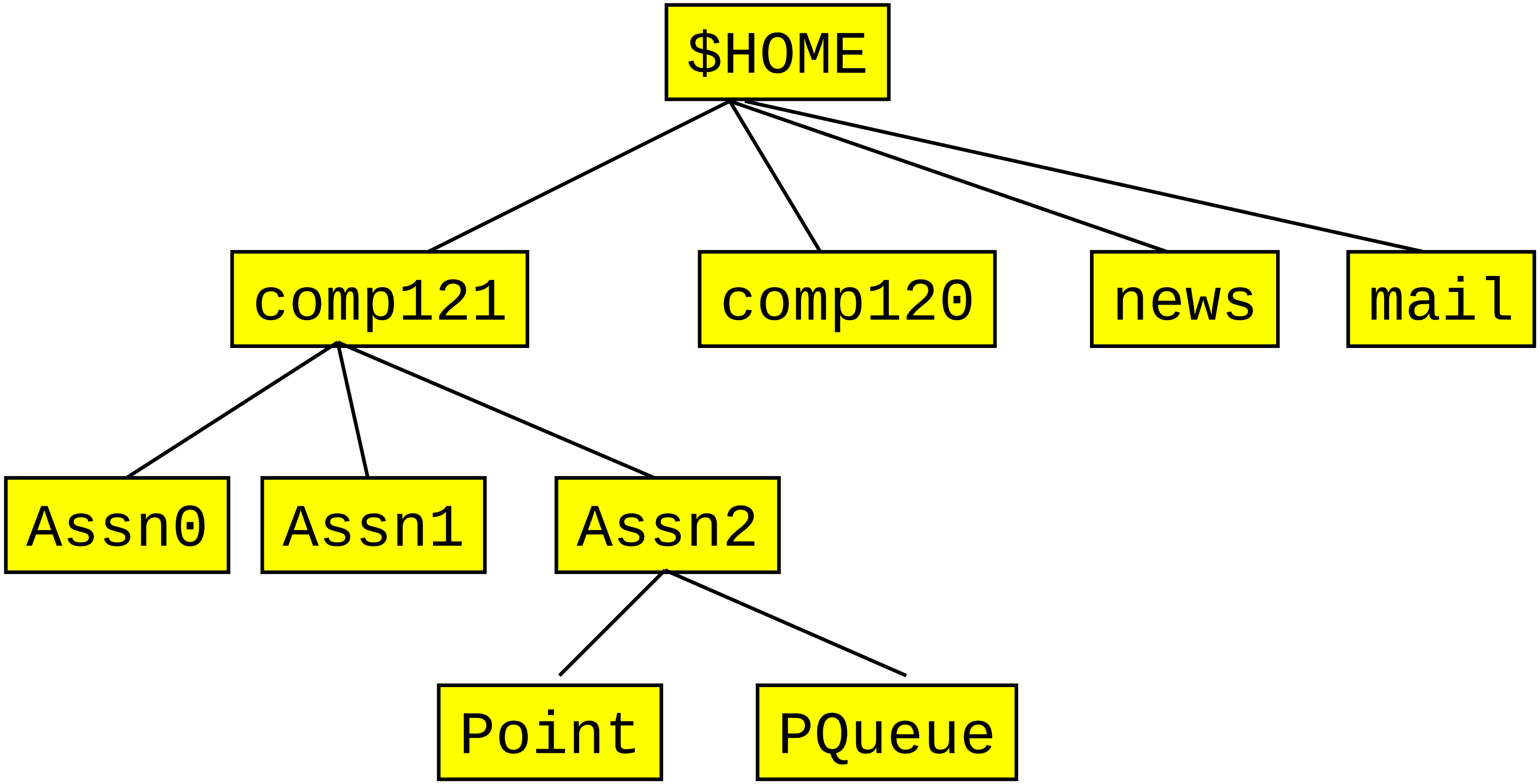


Trees: examples (*corporate structure*)



Trees: examples

(your Unix file system)



$HOME

comp121

comp120

news

mail

Assn0

Assn1

Assn2

Point

PQueue

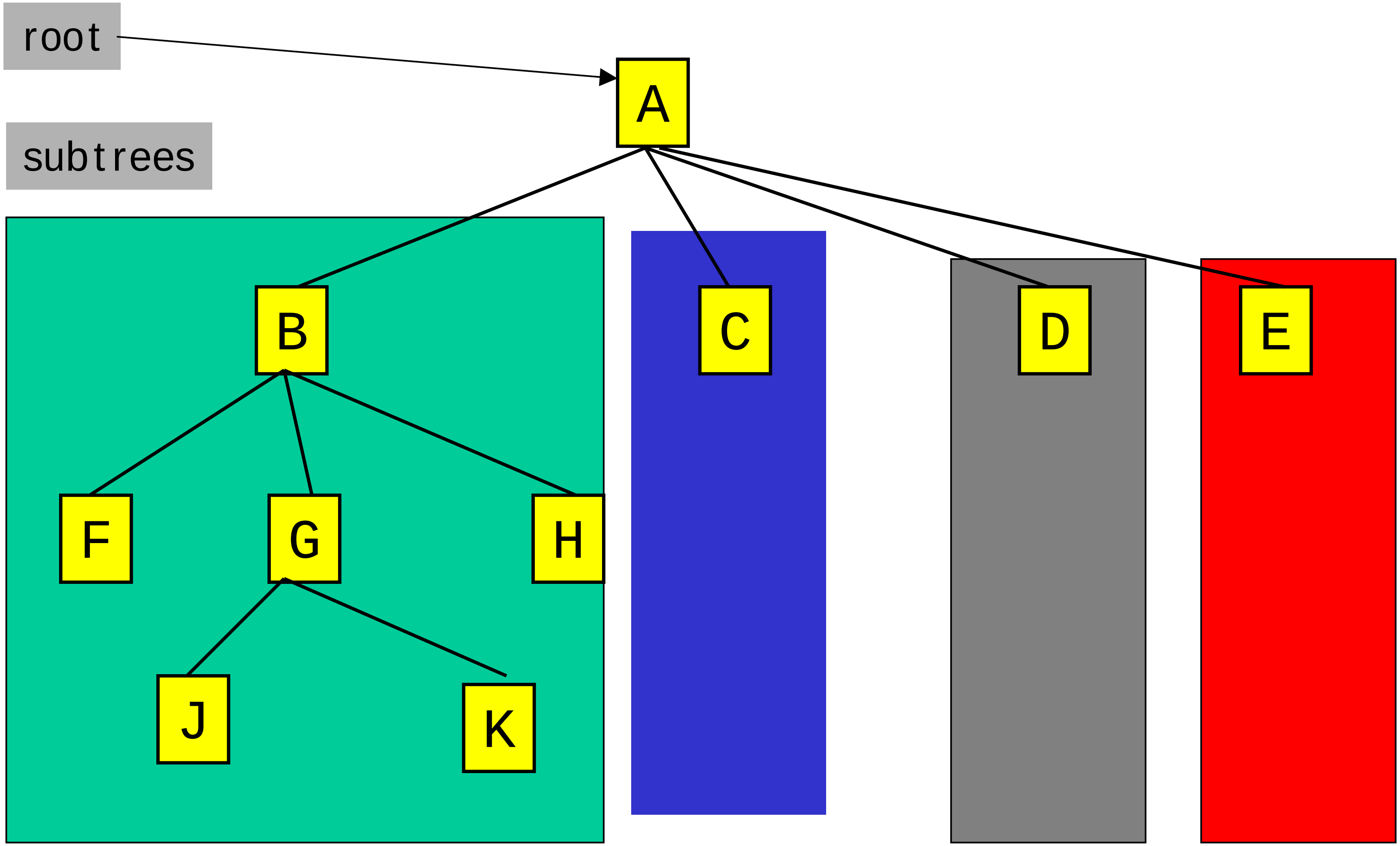
Definitions

- **A tree *t* is a finite nonempty set of elements. One of these elements is called a root, and the remaining elements (if any) are partitioned into trees that are called subtrees of *t*.**

- **children of the root -- root of subtree**
- **parent of an element/ grandparent/ ancestor/ descendent/...**

- **leaves: elements with no children**
- **degree of an element: # children**
- **degree of a tree: max degree of any element**

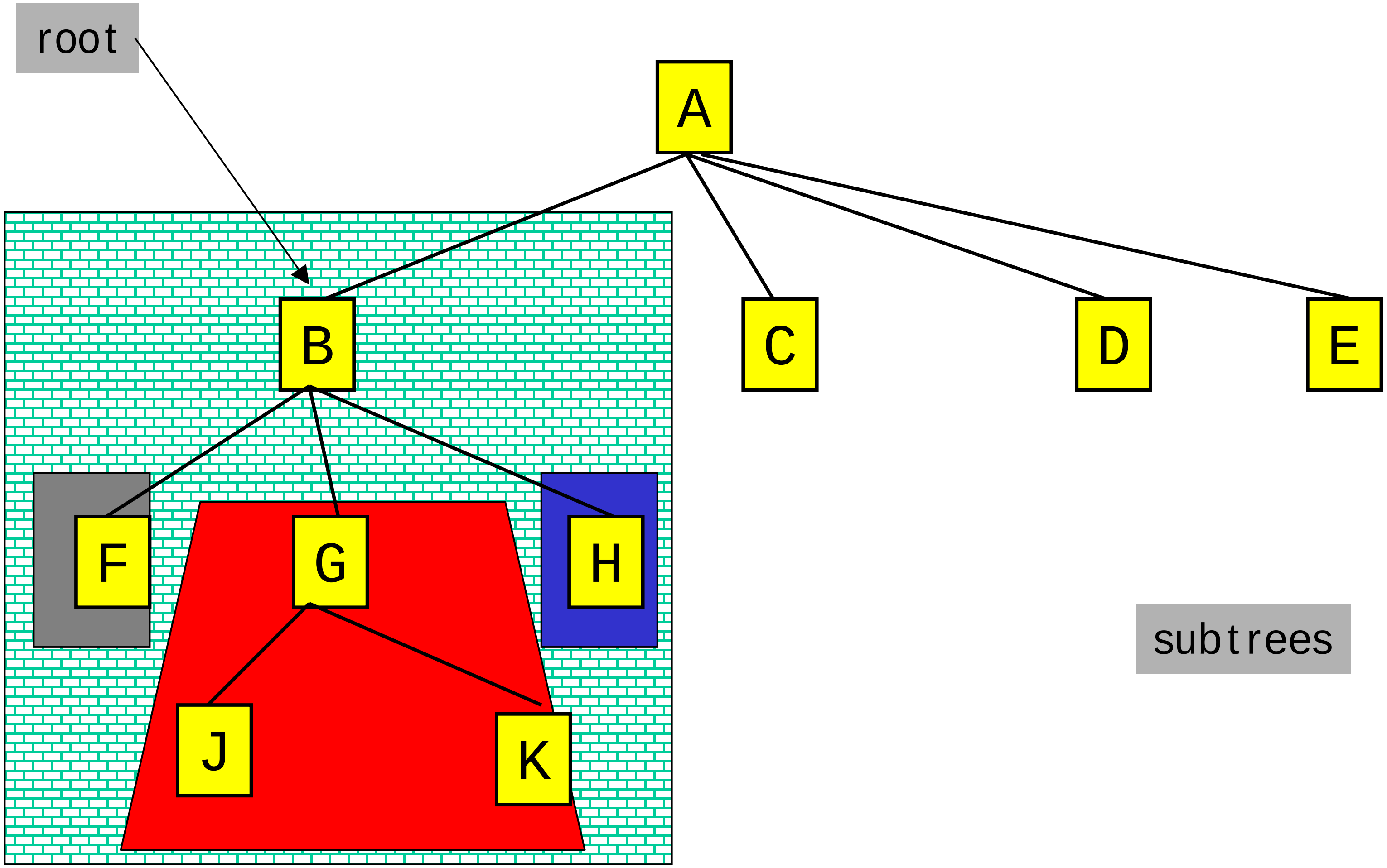
Tree definitions: examples



root

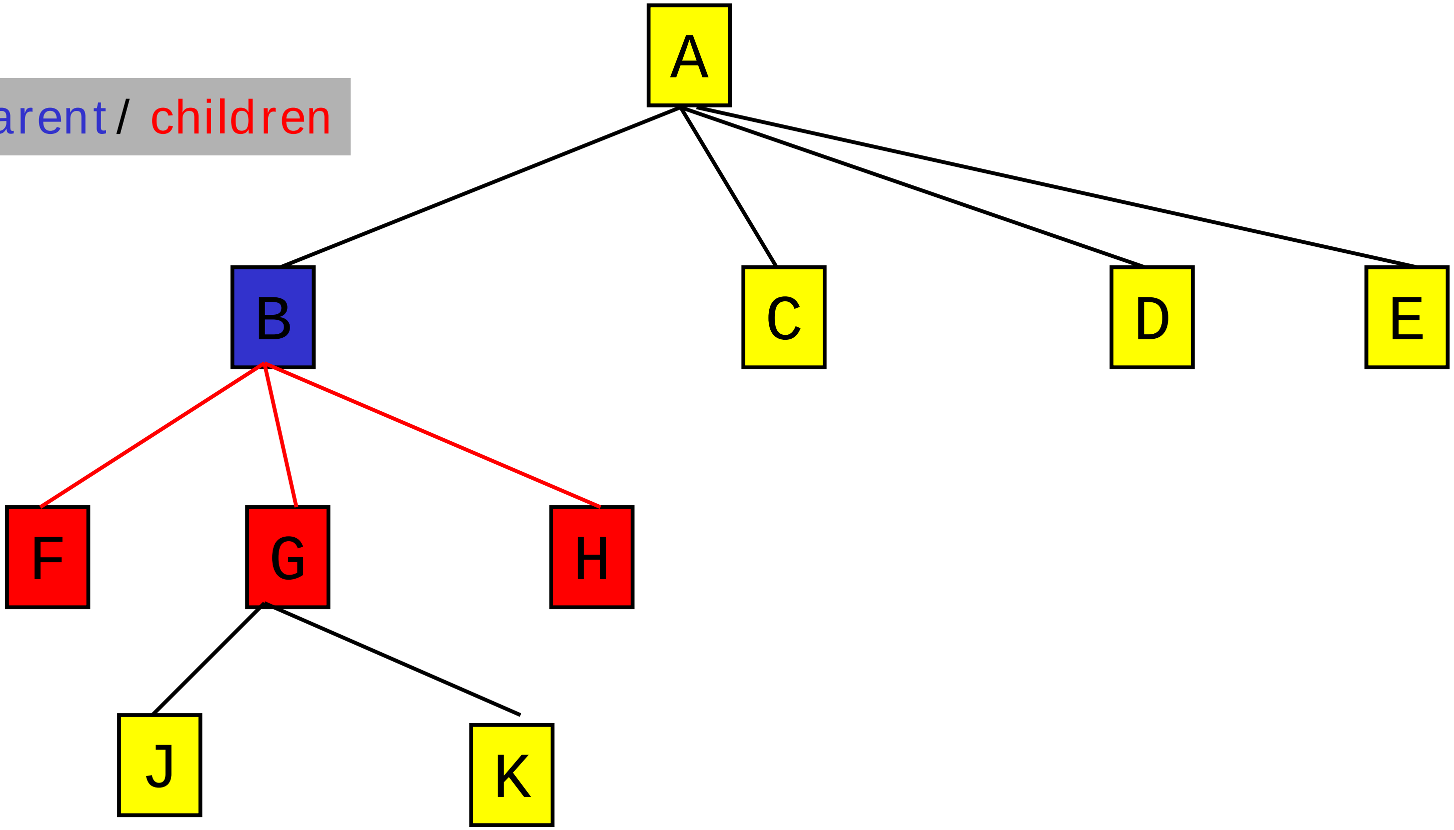
subtrees

Tree definitions: examples

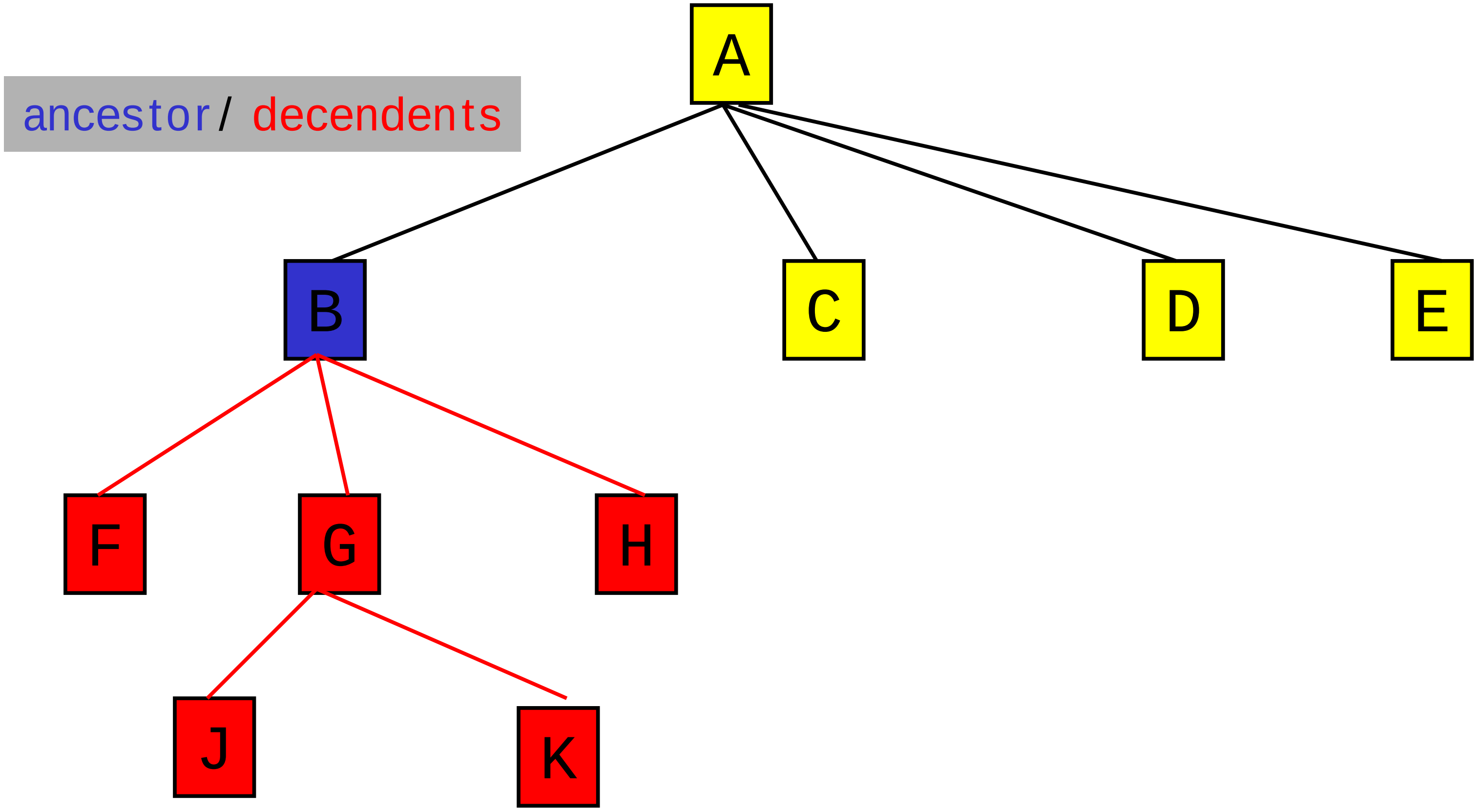


Tree definitions: examples

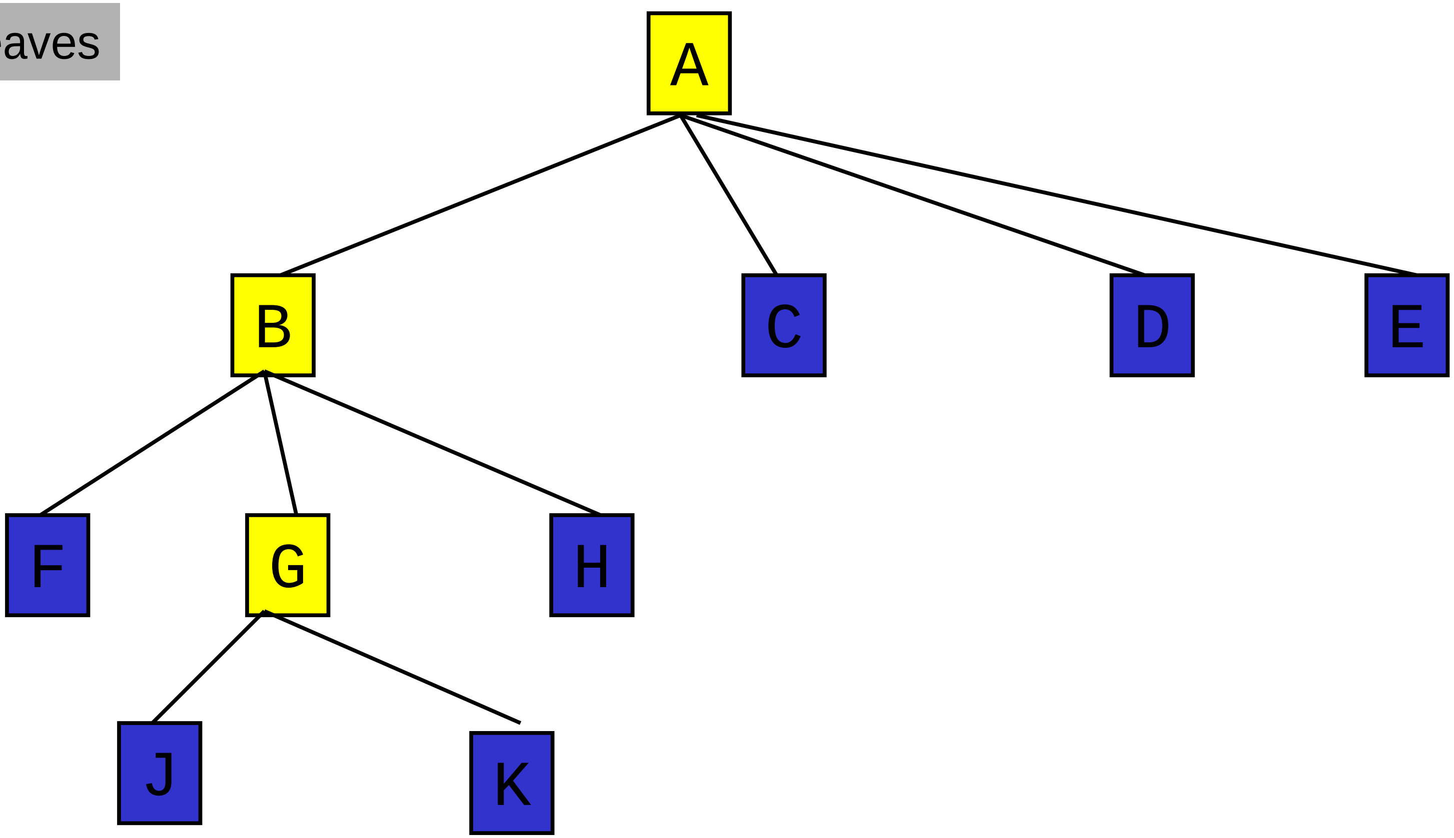
parent / children



Tree definitions: examples

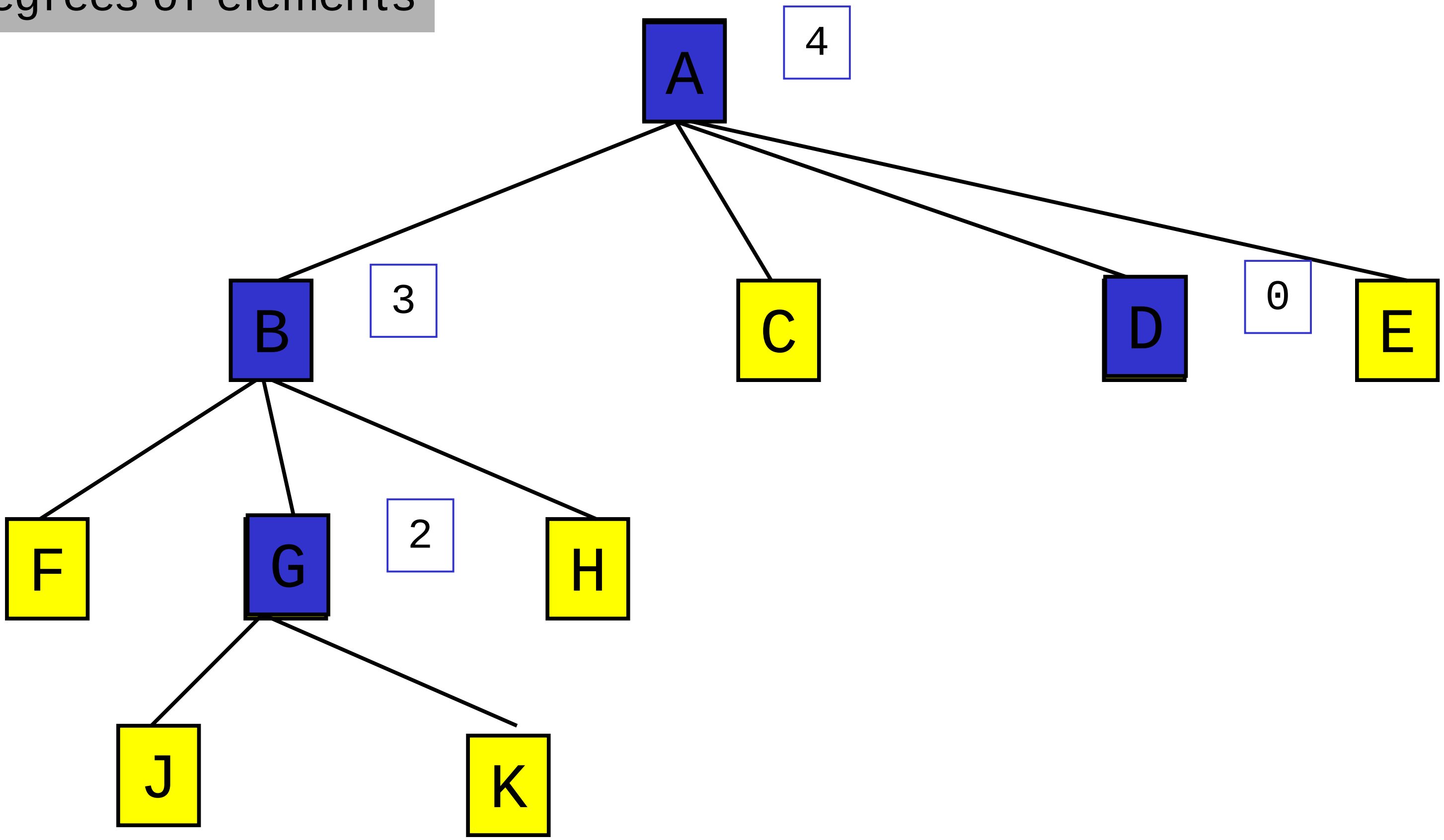


Tree definitions: examples


leaves

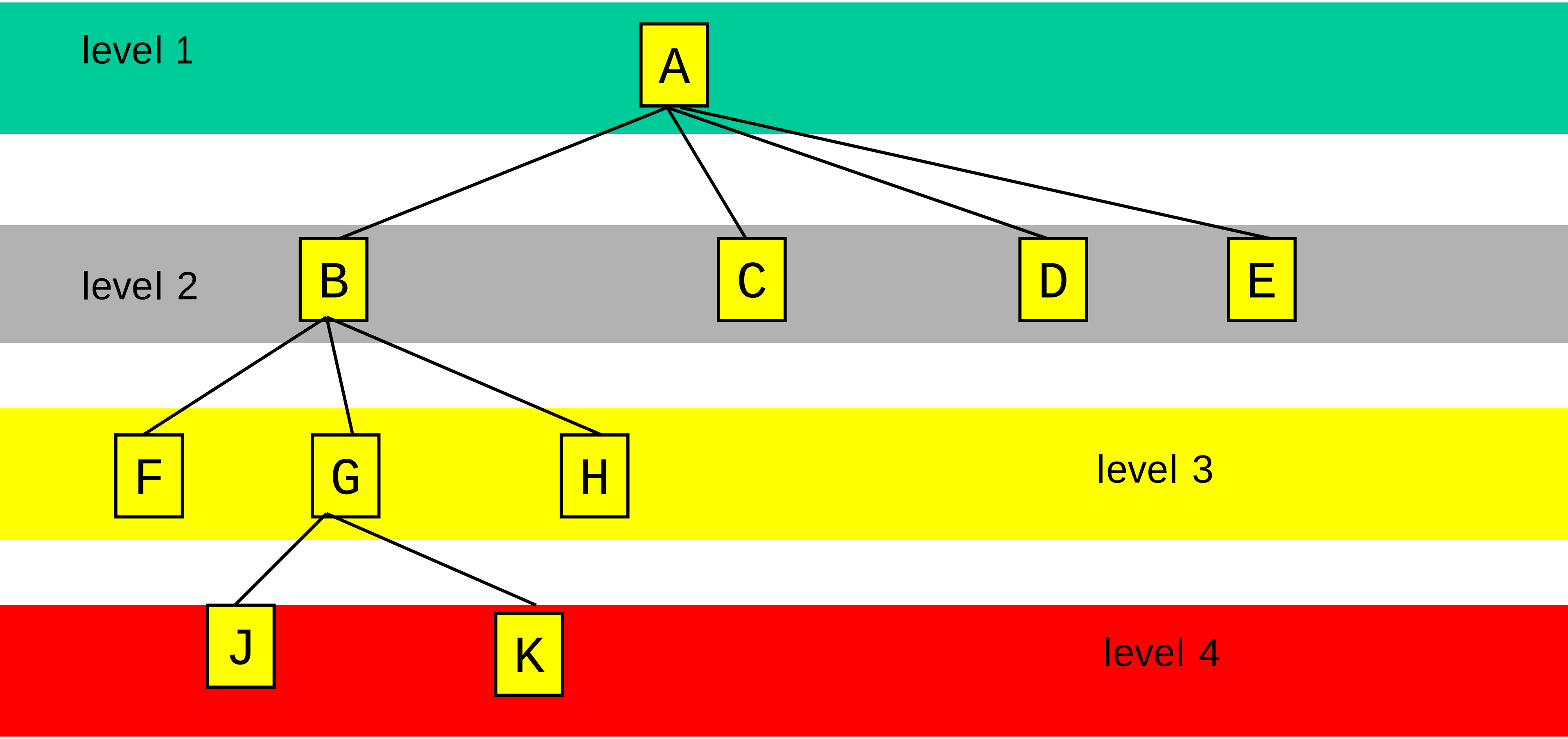
Tree definitions: examples

degrees of elements



Tree definitions: examples

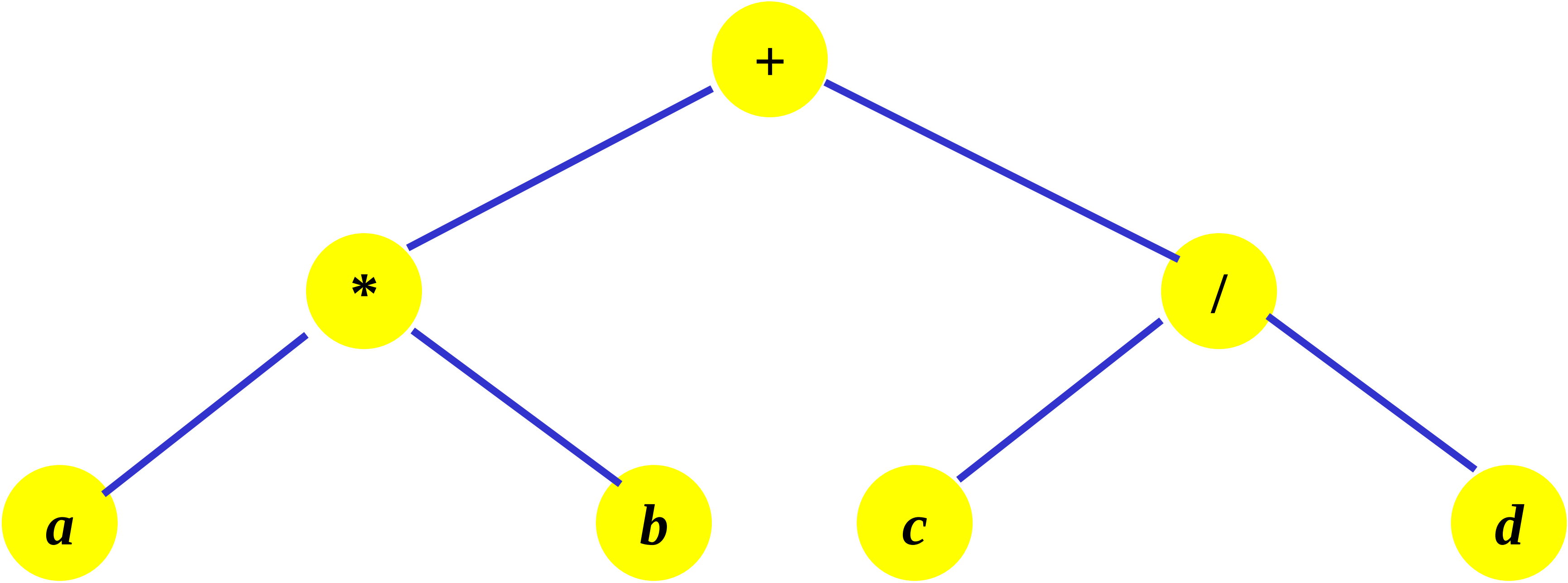
levels



Binary trees

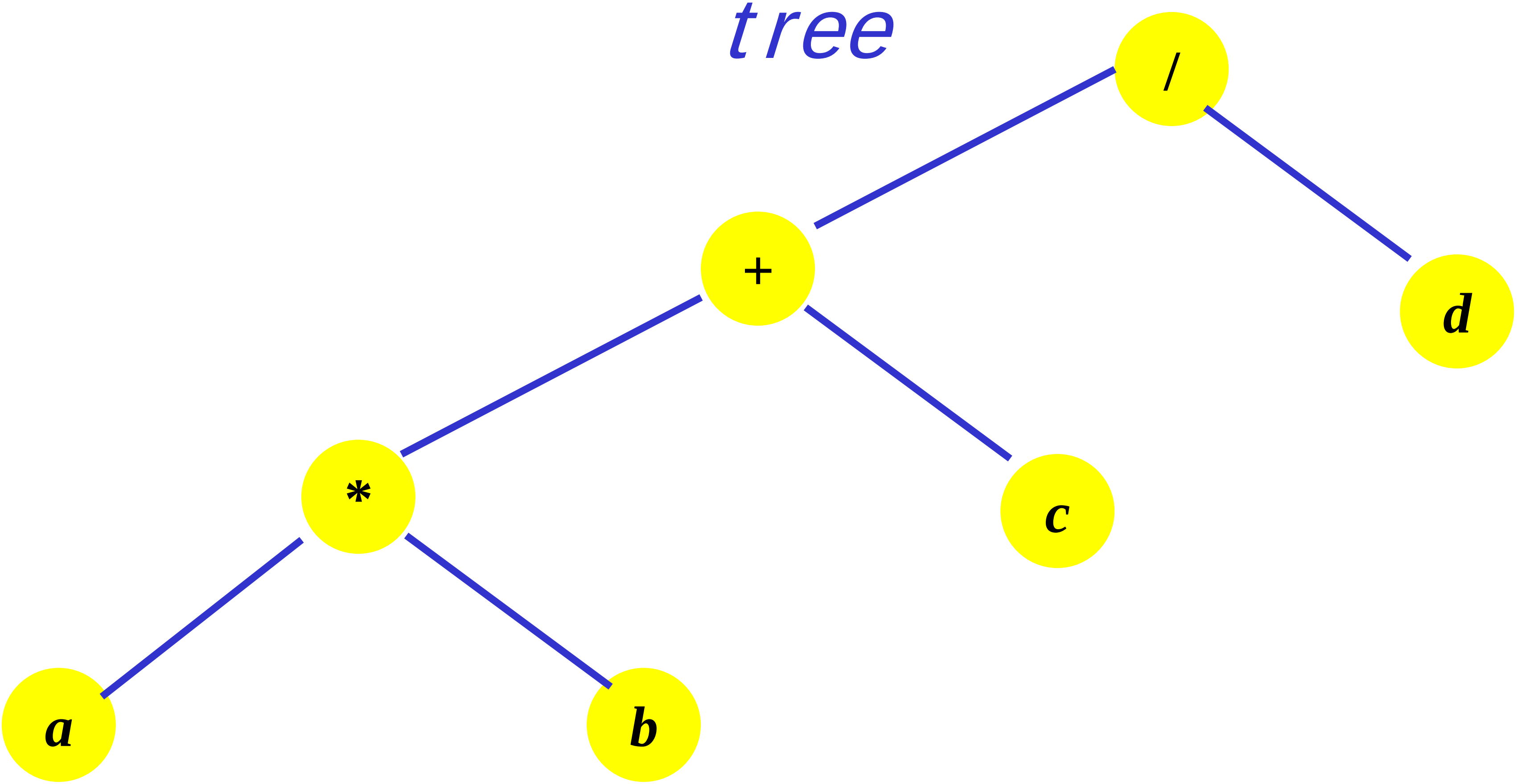
- **A binary tree *t* is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and the remaining elements (if any) are partitioned into two binary trees, called the left subtree and the right subtree of *t***
- **Notes:**
    - **a binary tree may be empty**
    - **each element has exactly two (perhaps empty) subtrees**
    - **the subtrees/ children are ordered**

Example binary tree: an *expression tree*



`(a * b) + (c / d)`

Example binary tree: an *expression tree*



`((a * b) + c) / d)`

Properties of binary trees

- **A binary tree of n elements has ?? edges**
  - **exactly n-1**
- **The height/ depth of a binary tree is the number of levels in it.**
  A binary tree of height h has at least ?? and at most ??? elements
  - **at least h**
  - **at most $2^h - 1$**
- **The height of a binary tree with n elements is at most ?? and at least ???**
  - **at most n**
  - **at least $\lceil \log_2 (n+1) \rceil$**

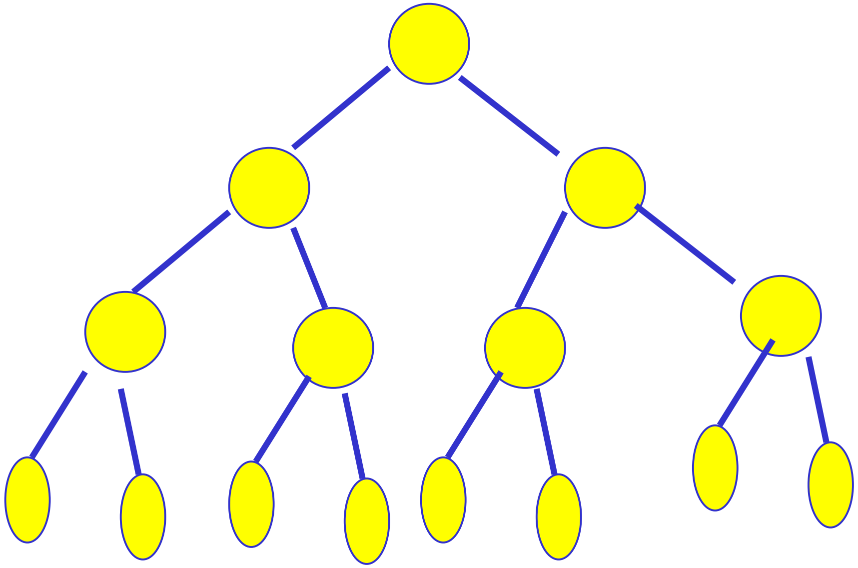
Binary trees (review from yesterday)

- **A binary tree *t* is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and the remaining elements (if any) are partitioned into two binary trees, called the left subtree and the right subtree of *t***
- **Notes:**
  - **a binary tree may be empty**
  - **each element has exactly two (perhaps empty) subtrees**
  - **the subtrees/ children are ordered**

Properties of binary trees (review from yesterday)

- **A binary tree of n elements has $n-1$ edges**
- **The height/ depth of a binary tree is the number of levels in it.**
  A binary tree of height h has at least h and at most $2^h - 1$ elements
- **The height of a binary tree with n elements is at most n and at least $\lceil \log_2 (n+1) \rceil$**

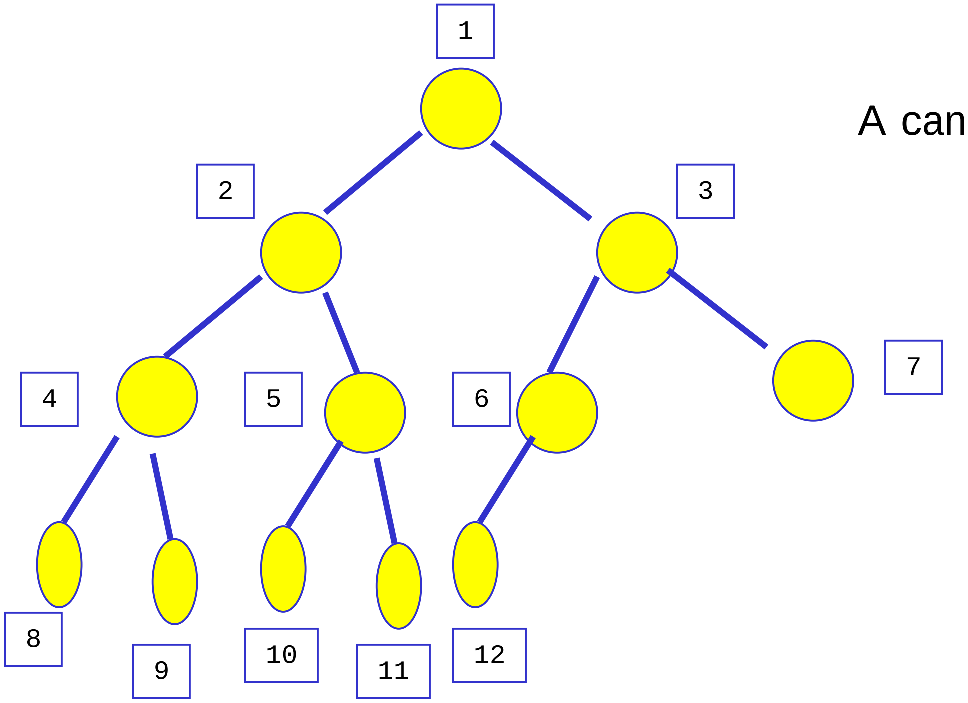
Binary trees: some more definitions



A FULL binary tree

(Ht h ==> #-elements = 2^h -1)

Binary trees: some more definitions



A canonical ordering

A COMPLETE binary tree

(levels fill in from the left)

Representing binary trees

As an array:

Use the canonical ordering -- node with canonical ordering i goes into T[i]

T[i]'s left child at ??

T[i]'s right child at ??

T[i]'s parent is at ??

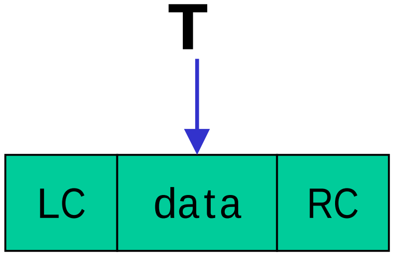
Problems with this approach:

memory inefficient

(works fine for full or complete binary trees)

Representing binary trees

As a pointer to a treeNode:



Representing binary trees

As a pointer to a treeNode:

```
template <class T>
class node{
public:

      T data;
      node <T> * LC;
      node <T> * RC;
};
```

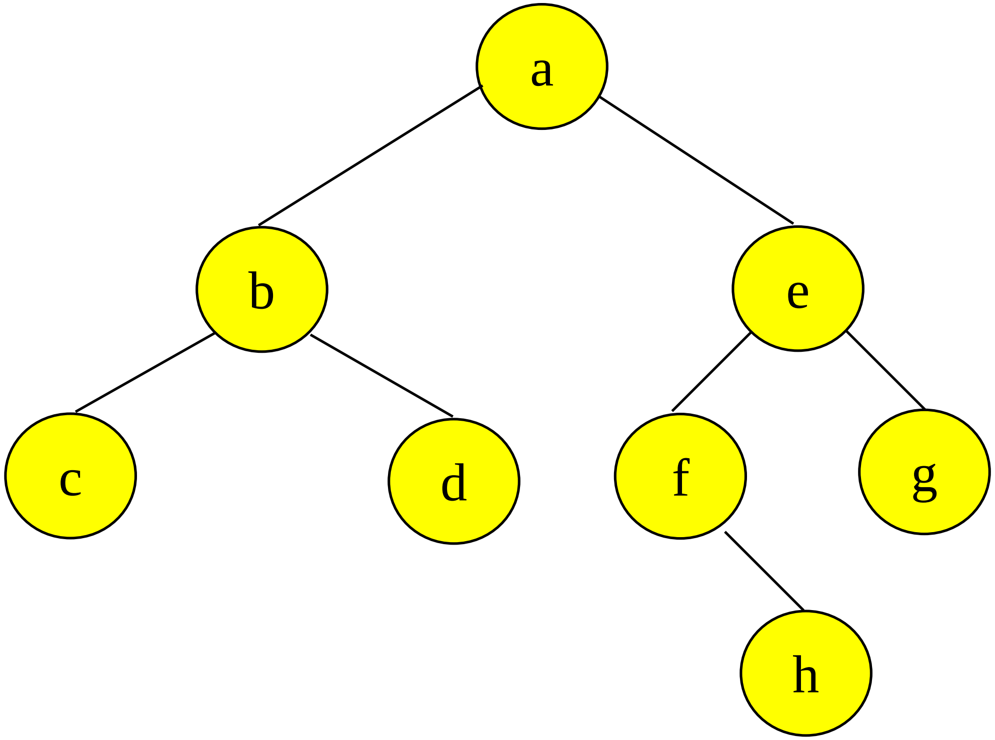
Representing binary trees

As a pointer to a treeNode:

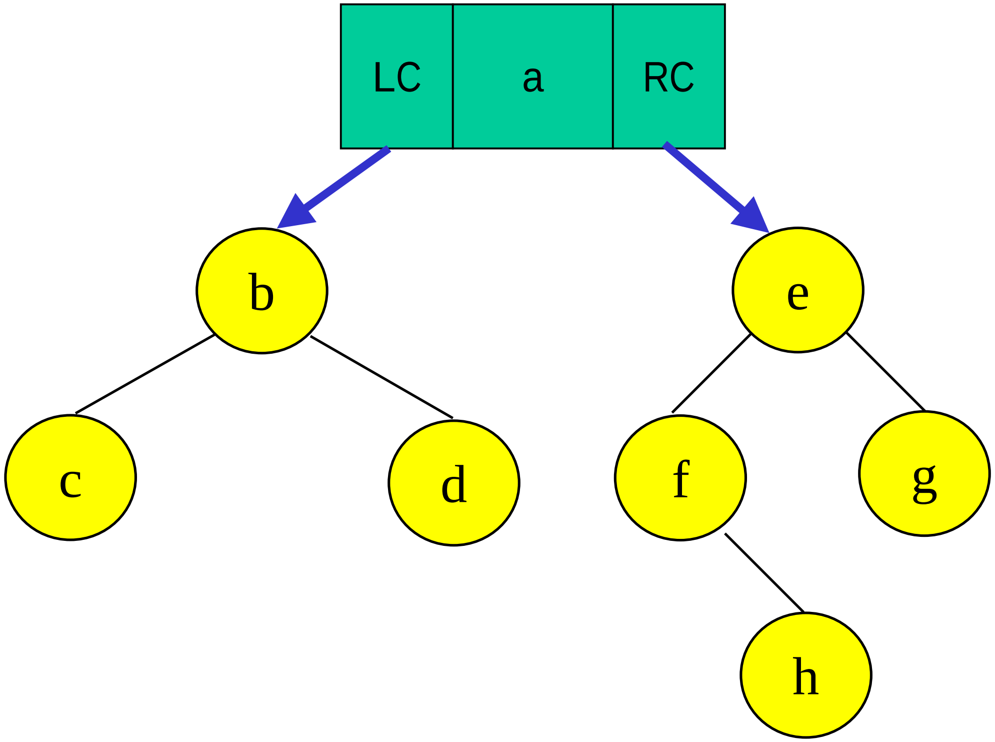
```
template <class T>
class node{
public:
     node(T x, node<T>* t1=NULL, node<T>* t2=NULL);
     T data;
     node <T> * LC;
     node <T> * RC;
};
```

{data = x; LC = t1; RC = t2;}

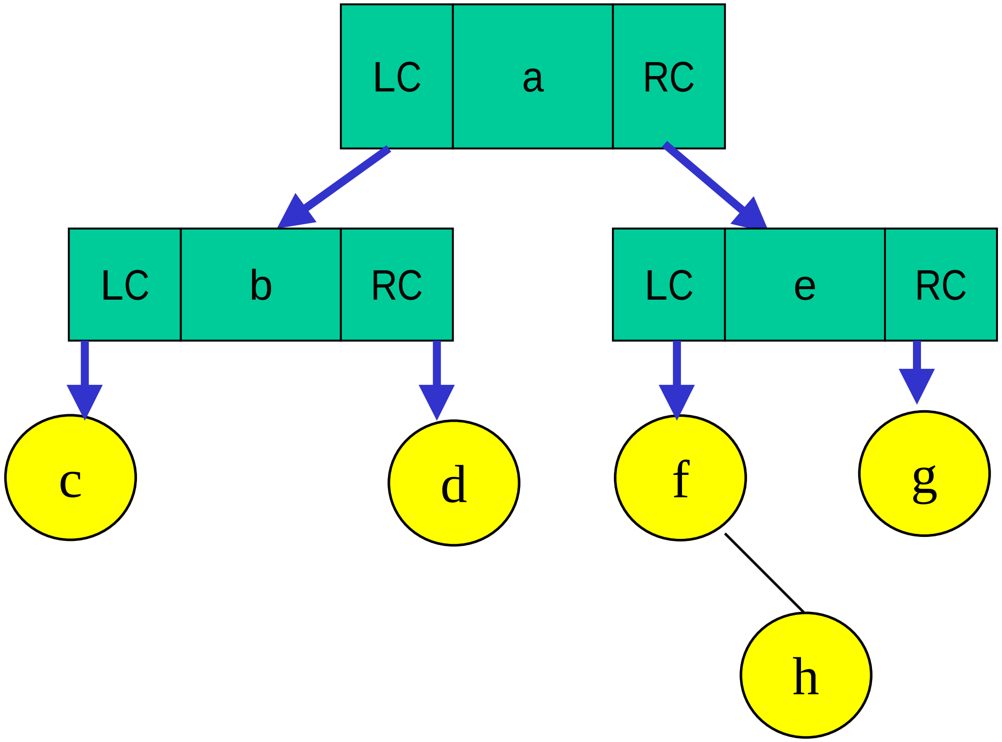
An example tree



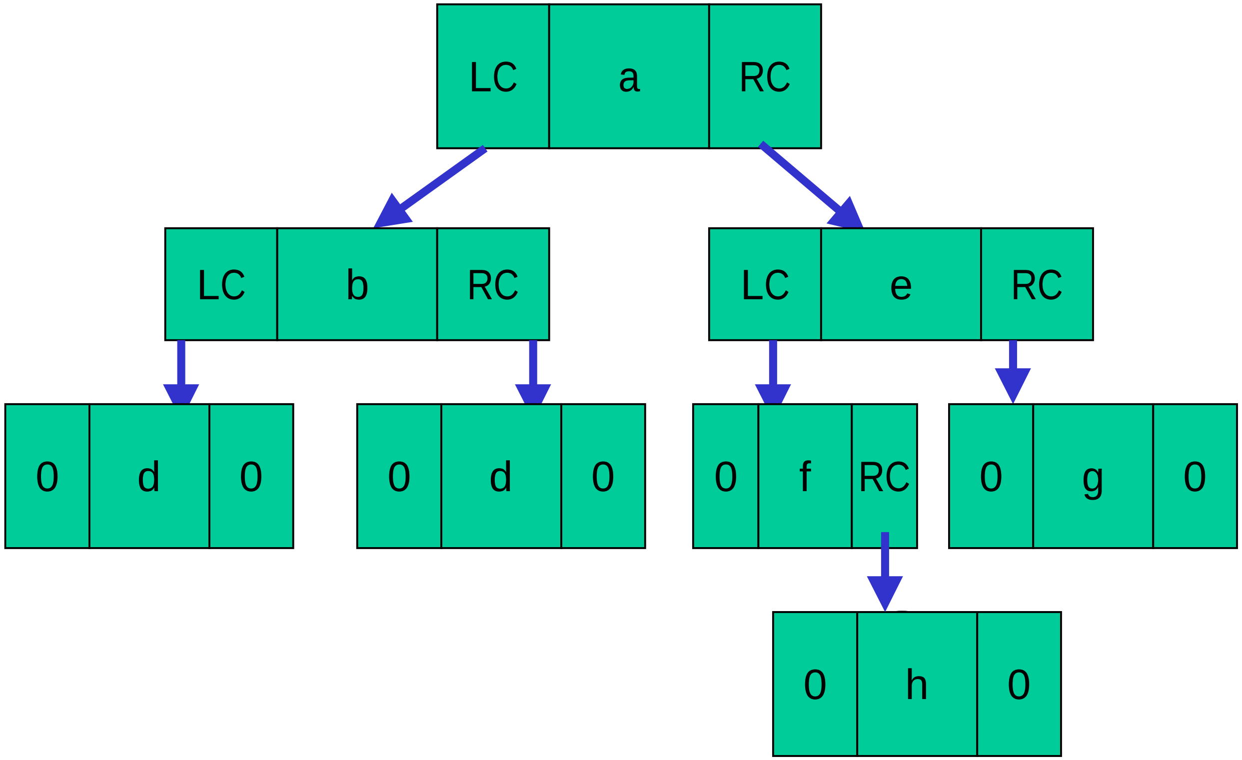
An example tree



An example tree

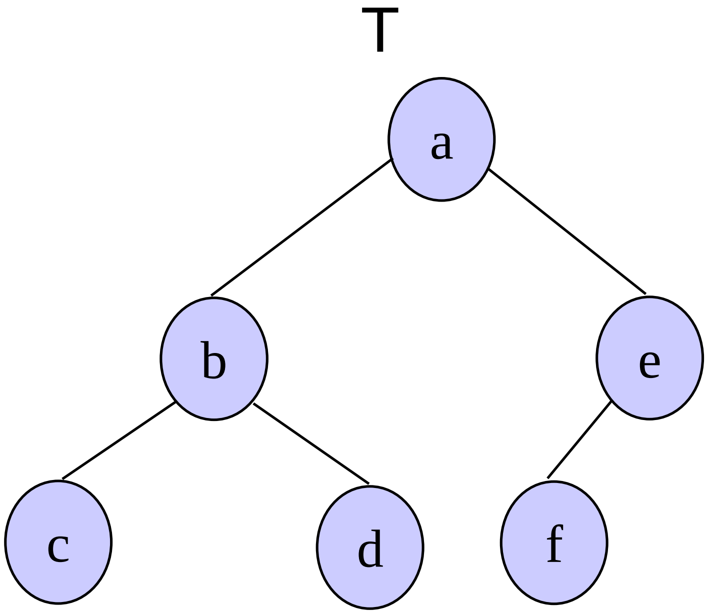


An example tree



Representing binary trees

```
template <class C>
class node{
public:
     node(C x, node<C>* t1, node<C> * t2);
    C data;
    node<C> * LC;
    node<C> * RC;
    };
```



Recursive programming on binary trees: counting elements

```
// in the user (client) program
int countElements(node<char> * T)
{
  if (T == NULL) return 0;
  return (
     1  // itself
    + countElements(T->LC)
    + countElements(T->RC)  );
}
```

Recursive programming on binary trees: computing depth

```
// in the user (client) program
int depth(node<char> * T)
{
  if (T == 0) return 0;
  return (
     1  +
     max (depth(T->LC),
          depth(T->RC)) );
}
```

Recursive programming on binary trees: comparing trees

```
// in the user (client) program
bool identical(node<int> * T1, node<int> * T2)
{
  if (T1 == NULL) return (T2 == NULL);
  if (T2 == NULL) return false;
  if (T1->data != T2->data) return false;
  return ( identical (T1->LC, T2->LC)
               &&
             identical (T1->RC, T2->RC) );
}
```

Recursive programming on binary trees: comparing trees

```
// in the user (client) program
bool what(node<int> * T1, node<int> * T2)
{
  if (T1 == NULL) return (T2 == NULL);
  if (T2 == NULL) return false;

  return ( what(T1->LC, T2->LC)
              &&
             what(T1->RC, T2->RC) );
}
```

Recursive programming on binary trees: preorder traversal

```
// in the user (client) program
void preOrder(node<char> * T)
{
  if (T == 0) return;
    cout << T->data;
    preOrder(T->LC);
    preOrder(T->RC);
}
```

Recursive programming on binary trees: inorder traversal

```
// in the user (client) program
void  inOrder(node<char> * T)
{
  if (T == 0) return;
     inOrder(T->LC);
    cout << T->data;
     inOrder(T->RC);
}
```

Recursive programming on binary trees: postorder traversal

```
// in the user (client) program
void postOrder(node<char> * T)
{
  if (T == 0) return;
    postOrder(T->LC);
    postOrder(T->RC);
    cout << T->data;
}
```

Recursive programming on binary trees: ???

```
// in the user (client) program
treeNode<char * mystery(node<char> * T)
{
  if (T == 0) return NULL;
  return new node<char>(
         T->data,
         mystery(T->LC),
         mystery(T->RC)
         );
}
```

mystery returns a copy

Recursive programming on binary trees: ???

```
// in the user (client) program
treeNode<char * puzzle(node<char> * T)
{
  if (T == 0) return NULL;
  return new node<char>(
       T->data,
       puzzle(T->RC),
       puzzle(T->LC)
       );
}
```

puzzle returns a mirror-image