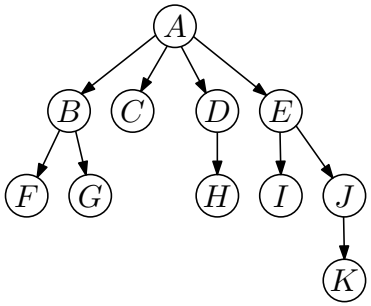
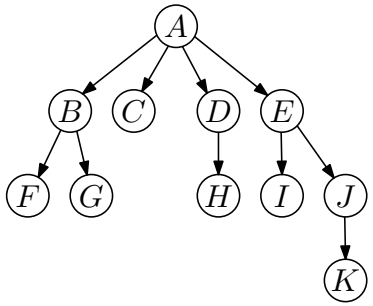
Nonrecursive definition:
A (rooted) tree consists of a set of nodes, and a set of directed edges between nodes.

- One node is the root;
- For every node c that is not the root, there is exactly one edge (p, c) pointing to c;
- For every node c there is a unique path from the root to c.



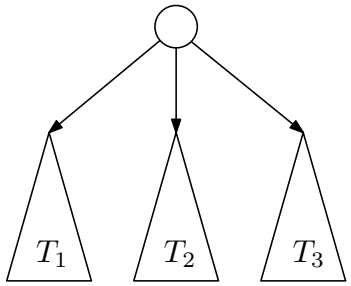
An edge connects parent and child.
A node without children is a leaf.
Nodes with the same parent are siblings.
Depth of v is the length of the path from the root to v.
Height of v is the length of the longest path from v to a leaf.
How many edges does a tree with n nodes have?

A tree with n nodes has $n - 1$ edges.



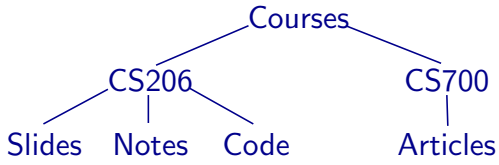
Node	Depth	Height
A	0	3
B	1	1
C	1	0
D	1	1
E	1	2
F	2	0
G	2	0
H	2	0
I	2	0
J	2	1
K	3	0

Recursive definition: A tree consists of a root, and zero or more subtrees T_1, T_2, ..., T_k. There is an edge from the root to the root of each subtree.



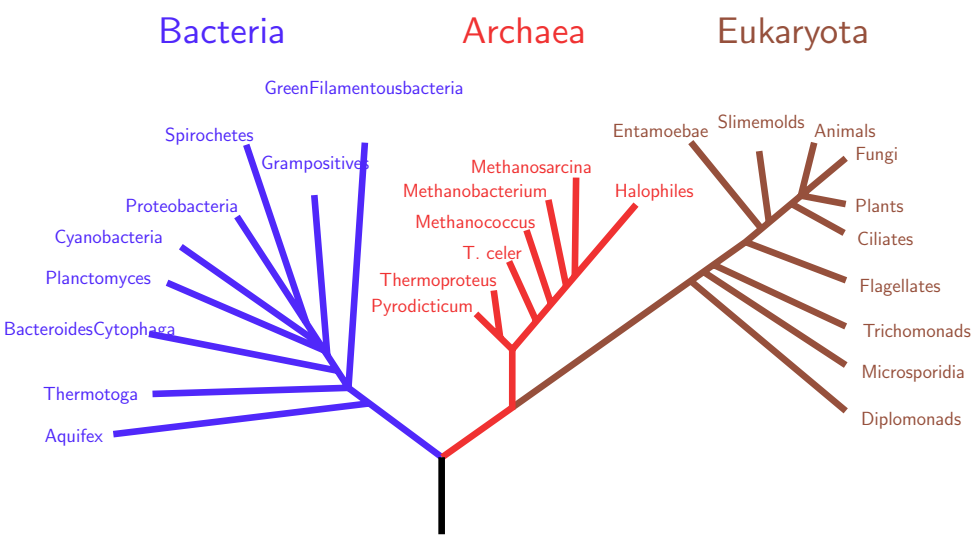
What is the base case of the recursion?

- A company organigram
- A filesystem

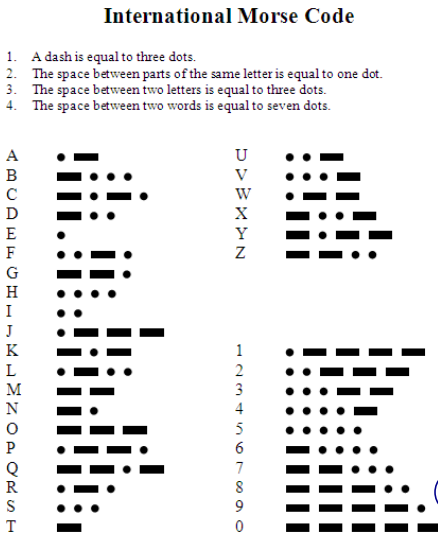


- A structured document (e.g. XML, HTML)
- A recursion tree (function call tree)
- An expression tree
- A decision tree

Phylogenetic tree of life

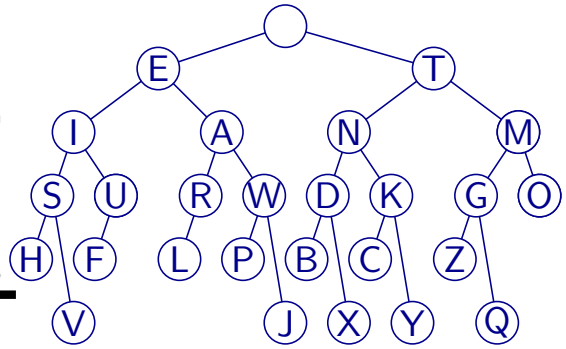


Decision Trees



We want to write a program to decode a transmission in Morse code.

To translate a letter, we make a decision tree.



Expression trees

```
trait Expression

case class Number(val value: Double) extends Expression
case class Variable(val name: String) extends Expression
case class Binary(val op: String,
                  val left: Expression,
                  val right: Expression) extends Expression
case class Unary(val op: String,
                 val child: Expression) extends Expression
case class Function(val fname: String,
                    val arg: Expression) extends Expression
```

Like list nodes, tree nodes are recursively defined types. This tree has two types of leaves (`Number` and `Variable`) and three types of inner nodes (`Binary`, `Unary`, `Function`).

`Binary` nodes have two children, `Unary` and `Function` have only one child.

Case classes in Scala

Case classes add some syntactic features to normal classes:

- Construct objects without `new`:
  ```
  val s = new Number(3)
  val s = Number(3)
  ```
- Arguments are automatically `val` fields:
  ```
  case class Number(val value: Double)
  case class Number(value: Double)
  ```
- A pretty `toString` method.
- `==` and `!=` work by comparing all fields.
- Case classes can be used as patterns:
  ```
  val e = Binary("+", Number(1), Variable("a"))
  val Binary(op, _, _) = e
  println(op)  // prints +
  ```

Evaluating an expression tree:

```
def eval(expr: Expression): Double = {
  expr match {
    case Number(v) => v
    case Variable(name) => variables(name)
    case Binary("+", l, r) => eval(l) + eval(r)
    case Binary("-", l, r) => eval(l) - eval(r)
    case Binary("*", l, r) => eval(l) * eval(r)
    case Binary("/", l, r) => eval(l) / eval(r)
    case Binary("^", l, r) => math.pow(eval(l), eval(r))
    case Unary("-", arg) => -eval(arg)
    case Function("sqrt", arg) => math.sqrt(eval(arg))
    case Function("sin", arg) => math.sin(eval(arg))
    case Function("cos", arg) => math.cos(eval(arg))
    case Function(name, _) =>
      throw new EvaluationError("Unknown function: "+name)
  }
}
```


Can it be evaluated?

An expression can be evaluated if all the variables occurring in it are defined:

```
def canEvaluate(expr: Expression): Boolean = {
  expr match {
    case Number(v) => true
    case Variable(name) => variables contains name
    case Binary(_, left, right) =>
      canEvaluate(left) && canEvaluate(right)
    case Unary(_, arg) => canEvaluate(arg)
    case Function(name, arg) => canEvaluate(arg)
  }
}
```


Displaying expressions

```
def show(expr: Expression): String  = {
  expr match {
    case Number(v) => v.toString
    case Variable(name) =>
        if (variables contains name)
          "(" + show(variables(name)) + ")"
        else
          name
    case Binary(op, left, right) =>
        "(" + show(left) + " " +op+ " " + show(right) + ")"
    case Unary(op, arg) =>
        op + "(" + show(arg) + ")"
    case Function(name, arg) =>
        name + "(" + show(arg) + ")"
  }
}
```


Prefix and postfix notation

The Lisp programming languages (Scheme, Racket) express everything in prefix-notation:

```
(* a (+ 2 (- b 7)))
```

Some programming languages (Forth, Postscript) are based on a stack, and need expressions in postfix notation:

```
a 2 b 7 - + *
```

Compilers can create this code for a stack-based processor.

Tree traversals

A tree traversal is the process of visiting all nodes of a tree, usually in a recursive manner.

All operations on our expression trees (evaluating, testing if it can be evaluated, conversion to string) are actually tree traversals.

We distinguish three main types of tree traversals, depending on when the information in a node is processed:

- Preorder traversal means that a node is processed before its children;
- Postorder traversal means that a node is processed after its children;
- Inorder traversal means that a node is processed between its left child and its right child (and is usually only used for binary trees).