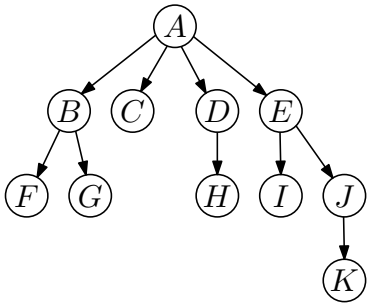
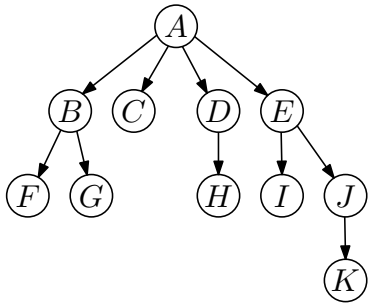
Nonrecursive definition:
A (rooted) tree consists of a set of nodes, and a set of directed edges between nodes.

- 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.



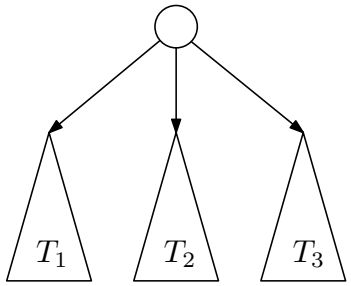
An edge connects parent and child.
A node without children is a leaf.
Nodes with the same parent are siblings.
Depth of v is the length of the path from the root to v.
Height of v is the length of the longest path from v to a leaf.
How many edges does a tree with n nodes have?

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



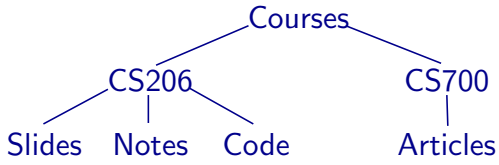
Node	Depth	Height
A	0	3
B	1	1
C	1	0
D	1	1
E	1	2
F	2	0
G	2	0
H	2	0
I	2	0
J	2	1
K	3	0

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.



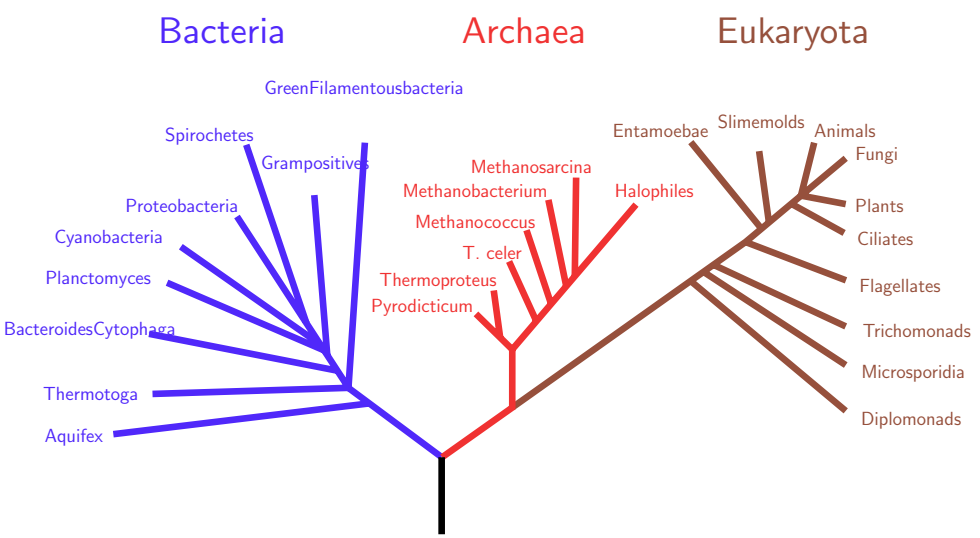
What is the base case of the recursion?

- A company organigram
- A filesystem



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

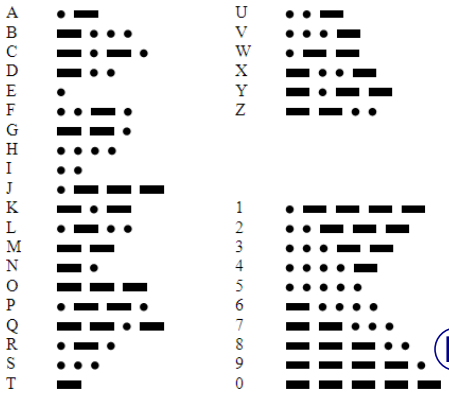
Phylogenetic tree of life



Decision Trees

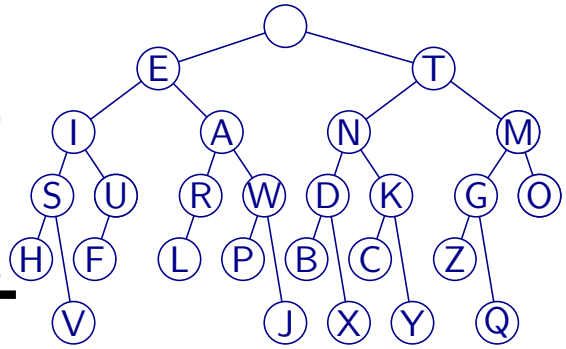
International Morse Code

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to seven dots.



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

To translate a letter, we make a decision tree.



Expression trees

```
class Expression():
  def __init__(self, data, left=None, right=None):
    self.data = data
    self.left = left
    self.right = right

e = Expression("*",
               Expression("a"),
               Expression("+",
                          Expression(2),
                          Expression("-",
                                     Expression("b"),
                                     Expression(7))))
```

expression.py

Like list nodes, tree nodes are recursively defined types. This tree has two types of leaves (for numbers and for variables) and two types of inner nodes (for unary minus and for binary operations).

Displaying expressions

```
def __str__(self):
  t = self.type()
  if t == "number":
    return str(self.data)
  if t == "variable":
    return self.data
  if t == "unary":  # unary minus
    return "-" + str(self.left)
  # it's a binary operation
  return ("(" + str(self.left) + " " + self.data
          + " " + str(self.right) + ")")
```

Building expression trees

We can reuse our expression parser to build an expression tree. Each method `parse_item`, `parse_factor`, `parse_term`, and `parse_expression` now returns an `Expression`.

```
def parse_term(tok):
  expr = parse_factor(tok)
  t = tok[0]
  while t.isSymbol("*") or t.isSymbol("/"):
    tok.pop(0)
    rhs = parse_factor(tok)
    expr = Expression(t.value, expr, rhs)
    t = tok[0]
  return expr
```


Evaluating an expression tree:

```
def evaluate(expr, vars):
  t = expr.type()
  if t == "number":
    return expr.data
  if t == "variable":
    if expr.data in vars:
      return vars[expr.data]
    else:
      raise EvalError("Undefined variable '%s'" % expr.data
  if t == "unary":
    arg = evaluate(expr.left, vars)
    return -arg
  op = expr.data
  lhs = evaluate(expr.left, vars)
  rhs = evaluate(expr.right, vars)
  if op == "+":
    return lhs + rhs
  # and so on...
```

evaluate.py

Prefix notation

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

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

```
def prefix(expr):
  t = expr.type()
  if t == "number":
    return "%g" % expr.data
  if t == "variable":
    return expr.data
  if t == "unary":
    return "(- " + prefix(expr.left) + ")"
  return ("(" + expr.data +
          " " + prefix(expr.left) +
          " " + prefix(expr.right) + ")")
```

prepostfix.py

Postfix notation

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

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

```
def postfix(expr):
  t = expr.type()
  if t == "number":
    return "%g" % expr.data
  if t == "variable":
    return expr.data
  if t == "unary":
    return postfix(expr.left) + " chs"
  return (postfix(expr.left) + " " +
          postfix(expr.right) + " " + expr.data)
```

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, conversion to string, prefix and postfix notation of expressions) 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).