Arbres

Tecnologia de la Programació

Sebastià Vila-Marta

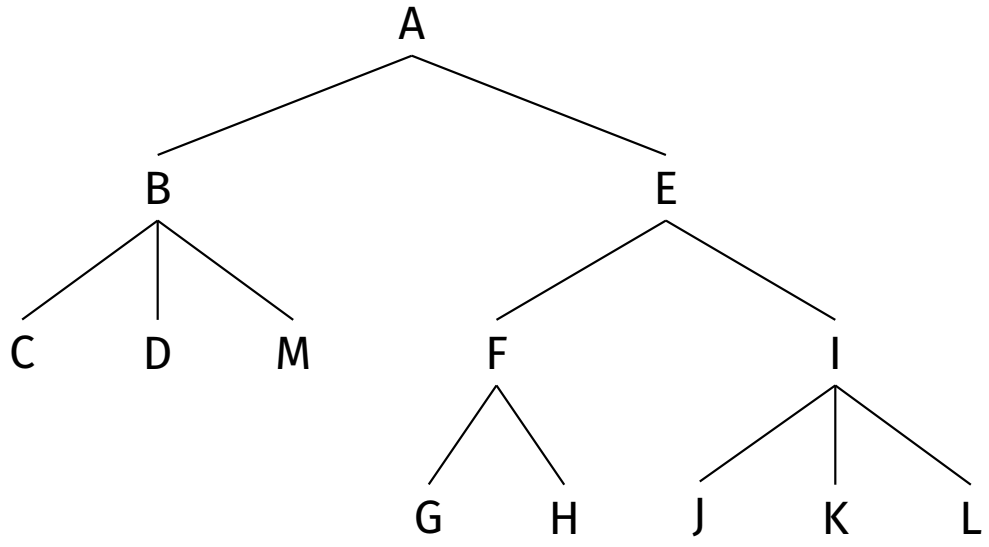
Enginyeria de Sistemes TIC
Universitat Politècnica de Catalunya
http://epsem.upc.edu

19 de maig de 2020

Arbre: Definició i nomenclatura I

- Un arbre és un graf $G = (E, V)$ connex i acíclic.
- Qualsevol node de l'arbre té un node pare excepte el node privilegiat anomenat arrel.
- Qualsevol node té zero o més fills. Un node sense fills l'anomenem fulla.
- Un node que no és fulla és intern.
- Els nodes que tenen el mateix pare s'anomenen germans.
- La longitud del camí més llarg s'anomena alçada de l'arbre.
- La longitud del camí que va de l'arrel al node n és la profunditat d'n.
- Un arbre és n-ari o d'ordre n si qualsevol node té com a molt n fills.

Arbre: Definició i nomenclatura II



Implementació d'arbres I

- Les operacions fonamentals sobre un arbre són les següents:
  - Arbre(n) Crea un arbre format per un únic node n sense fills.
  - arrela(self,l) Essent l una lista d'arbres, arrela els arbres de l com a fills de l'arrel de self i retorna el nou arbre.
  - fills(self) Retorna la llista dels fills de self.
  - es_fulla(self) Retorna **True** ssi self és una fulla.
- Hi ha moltes implementacions possibles per a un arbre.
- Una de les més simples, en el cas de Python, és representar un arbre com una classe amb un atribut, v, que és el valor de l'arrel i una llista f dels fills d'aquest node.

Implementació d'arbres II

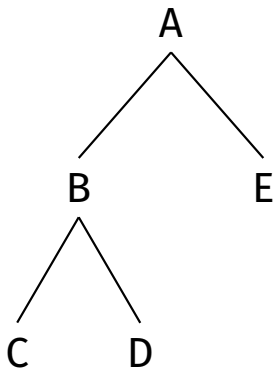
```
class Arbre(object):
    def __init__(self, n):
        self.v = n #public
        self._l = []

    def arrela(self, l):
        self._l = l
        return self

     def fills(self):
        return self._l

    def __len__(self):
        return len(self._l)

    def es_fulla(self):
        return len(self) == 0
```




```
>>> c = Arbre('C')
>>> b = Arbre('B').arrela([c,Arbre('D')])
>>> e = Arbre('E')
>>> a = Arbre('A').arrela([b,e])
>>> len(a)
2
>>> len(e)
0
```

Exemples senzills amb arbres

- Escriu el valor dels néts d'un node n:

```
for f in n.fills():
    for n in f.fills():
        print n.v
```

- Calcula el nombre de fills d'a que són fulles:

```
nfulles = 0
for f in a.fills():
    if es_fulla(f):
        nfulles += 1
```

- Donat un node intern n, calcula el nombre de germans. Compte! aquest càlcul no el podem expressar atès que un node no coneix directament els pare! Ens caldria una definició més rica de node.

Càlcul del nombre de fulles

Problema

Dissenyeu una funció que calcula el nombre de fulles d'un arbre a.

Càlcul del nombre de fulles

Problema

Dissenyeu una funció que calcula el nombre de fulles d'un arbre a.

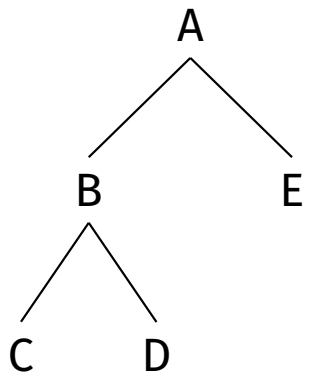
- És un problema recursiu!
- L'arbre és fulla o té fills.
- Cas base: l'arbre és una fulla.
- Cas recursiu: suma de les fulles dels fills.
- Solució:

```
def nfulles(a):
    if es_fulla(a):
        return 1
    else:
        return sum([nfulles(c) for c in a.fills()])
```

Recorregut dels nodes d'un arbre

- Recórrer una estructura de dades implica veure l'estructura com una col·lecció lineal atès que les darreres són les úniques on el concepte de recorregut està ben definit.
- Els recorreguts en arbres tenen natura recursiva i són fonamentals per a resoldre multitud de problemes.
- En el cas dels arbres es distingeixen tres formes de recórrer els nodes:
  - Preordre:
    Primer l'arrel i després els subarbres d'esquerra a dreta.
  - Inordre (només en arbres binaris):
    Primer el subarbre esquerre després l'arrel i finalment el subarbre dret.
  - Postordre:
    Primer els subarbres d'esquerra a dreta i després l'arrel.

Implementació dels recorreguts



- Imprimir en preordre:

```
def preordre(a):
    print a.v,
    for f in a.fills():
        preordre(f)
```

- Per l'arbre anterior, imprimeix:

  A B C D E

- Imprimir en postordre:

```
def postordre(a):
    for f in a.fills():
        postordre(f)
    print a.v,
```

- Per l'arbre anterior imprimeix:

  C D B E A

Exemples basats en recorreguts I

- Calcula l'ordre de l'arbre a:

```
def ordre(a):
    if es_fulla(a):
        return 0
    else:
        return max(len(a),max([ordre(x) for x in a.fills()]))
```

- Calcula l'alçada de l'arbre a:

```
def height(a):
    if es_fulla(a):
        return 0
    else:
        return max([height(x) for x in a.fills()]) + 1
```

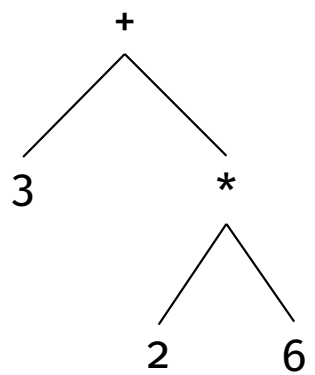
Exemples basats en recorreguts II

- Busca si a conté un node amb valor w:

```
def conte(a, w):
    if a.v == w :
        return True
    else:
        for f in a.fills():
            if conte(f,w):
                return True
        return False
```

Els arbres binaris i les expressions

- Els arbres binaris, d'ordre 2, són freqüents en moltes aplicacions.
- Sovint es defineixen amb operacions especials per accedir al fill esquerre i dret en comptes d'una operació per accedir a la llista de fills.
- Les expressions es poden representar de forma directa mitjançant el seu arbre sintàctic, que és un arbre binari.
- Els nodes interns corresponen a operacions i les fulles a constants. Per exemple, l'expressió 3 + (2 * 6) es pot representar com:



Avaluació d'expressions

- Si l'arbre a descriu una expressió es pot avaluar recursivament assumint que:
  - El valor d'una fulla és el de la constant que representa.
  - El valor d'un node intern és el resultat d'operar els valors dels subarbres amb l'operador corresponent.
- Codificat resulta en:

```
def avalua(a):
    if es_fulla(a):
        return a.v
    elif len(a) == 1:
        # operador unari
        d1 = avalua(a.fill_esq())
        if a.v == '-':
            return -d1
        elif a.v == '+':
            return d1
        else:
            raise Exception()
    elif len(a) == 2:
        # operador binari
        d1 = avalua(a.fill_esq())
        d2 = avalua(a.fill_dre())
        if a.v == '+':
            return d1 + d2
        elif a.v == '-':
            return d1 - d2
        elif a.v == '*':
            return d1 * d2
        elif a.v == '/':
            return d1 / d2
        else:
```

Exemple final: anàlisi d'expressions I

- Per anàlisi (parsing) s'enten el procés informàtic que llegeix una cadena de caracters, la trenca i n'extreu l'estructura sintàctica en forma d'arbre.
- En aquest exemple veurem un petit analitzador que es capaç de crear un arbre sintàctic d'una expressió que pot contenir:
  - Nombres naturals en base 10.
  - Suma i producte binaris.
  - Parèntesis.
- La sintaxi d'aquestes expression es pot explicar usant la notació BNF (Backus-Naur Form) per descriure la gramàtica:

$$
\begin{aligned}
\langle \text{expressió} \rangle &\longrightarrow \langle \text{terme} \rangle + \langle \text{terme} \rangle \\
\langle \text{expressió} \rangle &\longrightarrow \langle \text{terme} \rangle \\
\langle \text{terme} \rangle &\longrightarrow \langle \text{factor} \rangle * \langle \text{factor} \rangle \\
\langle \text{terme} \rangle &\longrightarrow \langle \text{factor} \rangle \\
\langle \text{factor} \rangle &\longrightarrow (\langle \text{expressió} \rangle) \\
\langle \text{factor} \rangle &\longrightarrow \langle \text{natural} \rangle
\end{aligned}
$$

Exemple final: anàlisi d'expressions II

- Començarem per crear una classe anomenada Lexer que ens ajuda a recórrer la cadena que conté l'expressió:

```
class Lexer(object):
    def __init__(self, s):
        self._s = s.replace(' ','')
        self._next = 0
        self._t = self._get_next()

    def accept(self, t):
        if self._t[0] == t:
            ot = self._t
            self._t = self._get_next()
            return ot
        else:
            return None

    def _get_next(self):
        if self._next >= len(self._s):
            return ('e', None)
        c = self._s[self._next]
        if c in ['(', ')', '+', '*']:
            self._next += 1
            return (c, None)
        elif c.isdigit():
            i = self._next + 1
            while (i<len(self._s) and
                   self._s[i].isdigit()):
                i += 1
            v = int(self._s[self._next:i])
            self._next = i
            return ('i', v)
        else:
            raise Exception()
```

Exemple final: anàlisi d'expressions III

- Exemple d'ús:

```
>>> l = Lexer('(231 + 31)')
>>> l.accept('+')
None
>>> l.accept('(')
( '(', None )
>>> l.accept('i')
( 'i', 231 )
>>> l.accept('+')
( '+', None )
```

```
>>> l.accept('i')
( 'i', 31 )
>>> l.accept('i')
None
>>> l.accept(')')
( ')', None )
>>> l.accept('e')
( 'e', None )
```

- Ara, seguint les regles que defineixen la sintaxi, construirem un arbre binari que correspon a l'arbre sintàctic de l'expressió. A tal efecte usarem una tècnica anomenada Recursive Descent Parser.
- Aquesta tècnica només es pot aplicar a certes construccions gramaticals i es basa en «espiar» quin símbol precedeix la part d'expressió que no hem processat i actuar en conseqüència.

Exemple final: anàlisi d'expressions IV

- Un parser d'aquest tipus s'organitza com una col·lecció de funcions recursives:

```
def parse_expressio(l):
    """ l es un Lexer. Retorna un arbre sintactic """
    as_terme1 = parse_terme(l)
    if l.accept('+'):
        as_terme2 = parse_terme(l)
        return Arbre('+').arrela([as_terme1, as_terme2])
    else:
        return as_terme1

def parse_terme(l):
    as_factor1 = parse_factor(l)
    if l.accept('*'):
        as_factor2 = parse_factor(l)
        return Arbre('*').arrela([as_factor1, as_factor2])
    else:
        return as_factor1
```

Exemple final: anàlisi d'expressions V

```
def parse_factor(l):
    if l.accept('('):
        as_expr = parse_expressio(l)
        if not l.accept(')'):
            raise Exception("Manca parentesi")
        return as_expr
    tok = l.accept('i')
    if tok:
        return Arbre(tok[1])
    raise Exception("Manca natural o parentesi")
```

Feina per aquesta setmana

1. Estudi de la teoria. A partir de les transparències i les notes que heu pres.
2. Incrementar el xuletari i el glossari del tema.
3. Solució dels problemes del tema.