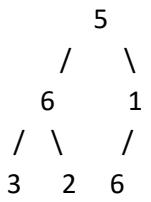
Árboles binarios

Juan Álvarez Rubio
jalvarez@dcc.uchile.cl

Las listas presentadas en la sección anterior representan secuencias de valores que se procesan secuencialmente elemento a elemento, accediendo recursivamente al primer valor de la lista y al resto de la lista, a través de las funciones cabeza y cola. Sin embargo, existen otras formas de representar conjuntos de valores que se organizan jerárquicamente en distintos niveles, como las ramas de un árbol, y que admiten algoritmos más eficientes.

Árboles binarios

Un árbol binario es una estructura u organización de datos que consiste de un valor con un árbol binario a la izquierda y otro árbol binario a la derecha. Por ejemplo, los datos 5, 6, 1, 3, 2 y 6 pueden organizarse en un árbol binario que se puede visualizar de la siguiente manera.



Un árbol binario puede estar vacío, es decir sin ningún valor, como ocurre con el árbol a la derecha del valor 1 y con los árboles binarios a la izquierda y derecha de los valores 3, 2 y 6. Para construir el árbol binario anterior se necesita definir un tipo de datos en un módulo que contenga también funciones para operar con árboles binarios.

Módulo para árboles binarios

El tipo de datos para representar un árbol binario necesita definir tres componentes: una para el valor, otra para el árbol binario a la izquierda, y otra para el árbol binario a la derecha:

```
#AB: valor(any), izq(AB), der(AB)
from estructura import crear
crear("AB","valor izq der")
```

El nombre del tipo es AB, como una abreviatura de Árbol Binario, y las componentes se denominan valor, izq (por árbol binario a la izquierda) y der (por árbol binario a la derecha). Con el tipo AB se puede definir el árbol binario con los valores 5, 6, 1, 3, 2 y 6 indicado anteriormente.

```
A=AB(5, \
     AB(6,AB(3,None,None),AB(2,None,None)),\
     AB(1,AB(6,None,None),None))
```

Considerando que un árbol binario puede estar vacío, se define la variable:

```
arbolVacio=None
```

El módulo incluye una función para verificar si un argumento es un árbol binario y otra función para comprobar si un árbol binario está vacío:

```
#esAB: any -> bool
#True si x es un AB
#ej: esAB(A)->True, esAB(arbolVacio)->True
def esAB(x):
  return x==arbolVacio or type(x)==AB
assert esAB(A)
assert esAB(arbolVacio)
```

```
#vacio: AB -> bool
#True is A es un arbol vacio
#ej: vacio(A)->False, vacio(arbolVacio)->True
def vacio(A):
  assert esAB(A)
  return A==arbolVacio
assert vacio(arbolVacio)
assert not vacio(A)
```

Para determinar la cantidad de valores de un árbol binario se requiere una función recursiva que sume uno a la cantidad de valores que existen en los árboles binarios de la izquierda y de la derecha:

```
#valores: AB -> int
#cantidad de valores de A
#ej: valores(A)->6, valores(arbolVacio)->0
def valores(A):
  assert esAB(A)
  if vacio(A):
    return 0
  else:
    return 1 + valores(A.izq) + valores(A.der)
assert valores(A)==6
assert valores(arbolVacio)==0
```

Como las componentes A.izq y A.der son a su vez árboles binarios del tipo AB, entonces se pueden utilizar como argumentos para invocar recursivamente a la función. Por otra parte, la altura de un árbol binario se define como la cantidad de niveles que tiene. Como los árboles a la izquierda y a la

derecha de un valor pueden tener distintas alturas, entonces la altura del árbol completo será uno más que la mayor altura de los dos árboles:

```
#altura: AB -> int
#numero de niveles de A
#ej: altura(A)->3, altura(arbolVacio)->0
def altura(A):
  assert esAB(A)
  if vacio(A):
      return 0
  else:
      return 1+max(altura(A.izq),altura(A.der))
assert altura(A)==3
assert altura(arbolVacio)==0
```

El módulo AB con todas las funciones anteriores puede ser utilizado para resolver otros problemas. Por ejemplo, para determinar si un valor está contenido en un árbol binario se puede escribir la siguiente función:

```
from AB import *
#enAB: any AB -> bool
#True si x esta en A
#ej: enAB(6,A)->True, enAB(4,A)->False
def enAB(x,A):
  assert esAB(A)
  if vacio(A):
      return False
  elif A.valor==x:
      return True
  else:
      return enAB(x,A.izq) or enAB(x,A.der)
assert enAB(6,A)
assert not enAB(4,A)
```

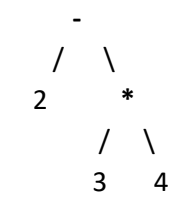
Si el valor buscado esté en el árbol de la izquierda, entonces no es necesario buscarlo en el árbol de la derecha, situación que se resuelve utilizando el operador lógico or. Considerando que un árbol binario puede tener valores que se repiten, entonces es pertinente definir una función que cuenta la cantidad de apariciones o la frecuencia de un valor:

```
from AB import *
#cuenta: any AB -> int
#cantidad de apariciones de x en A
#ej: cuenta(6,A)->2, cuenta(4,A)->0
def cuenta(x,A):
```

```
  assert esAB(A)
  if vacio(A):
      return 0
  elif A.valor==x:
      return 1 + cuenta(x,A.izq) + cuenta(x,A.der)
  else:
      return cuenta(x,A.izq) + cuenta(x,A.der)
assert cuenta(6,A)==2
assert cuenta(4,A)==0
```

Árbol binario para expresiones aritméticas

Los árboles binarios pueden ser utilizados para representar valores que siguen reglas especiales. Por ejemplo, una expresión aritmética puede ser representada en un árbol binario donde los valores intermedios contienen los operadores y los valores en las "hojas" (es decir los valores sin árboles a la izquierda y derecha) contienen los operandores. Por ejemplo, la expresión 2 – 3 * 4 se ´puede representar por el árbol:



La definición del árbol se logra con las instrucciones:

```
from AB import *
A=AB("-", \
     AB(2,None,None),\
     AB("*",AB(3,None,None),AB(4,None,None)))
```

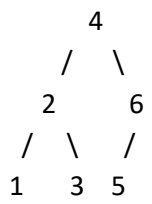
Una vez definido o construido el árbol binario, el valor de la expresión se puede clacular con una función que evalúe recursivamente los árboles a la izquierda (A.izq) y a la derecha (A.der) y calcule el resultado final según el operador guardado en la componente valor:

```
#evaluar: AB -> num
#resultado de la expresión representada en AB A
#ej: evaluar(A) -> -10
def evaluar(A):
  assert type(A)==AB
  if vacio(A.izq) and vacio(A.der): return A.valor
  a=evaluar(A.izq) #primer operando
  b=evaluar(A.der) #segundo operando
  op=A.valor       #operador
```

```
  if op=="+": return a+b
  if op=="-": return a-b
  if op=="*": return a*b
  if op=="/": return a/b
  assert False  #operador incorrecto
assert evaluar(A)==-10
```

Árbol binario de búsqueda

Los árboles binarios más conocidos y utilizados son los árboles binarios de búsqueda, que son árboles binarios que ponen condiciones a los valores de manera de facilitar las búsquedas. La siguiente figura muestra un árbol binario de búsqueda:



Un árbol binario de búsqueda (ABB) es un árbol binario que cumple con las siguientes reglas:

- Los valores del árbol izquierdo son menores que el valor
- Los valores del árbol derecho son mayores que el valor
- El árbol izquierdo es un ABB y el árbol derecho es un ABB

Las siguientes instrucciones definen el árbol anterior:

```
from AB import *
A=AB(4, \
     AB(2,AB(1,None,None),AB(3,None,None)),\
     AB(6,AB(5,None,None),None))
```

La regla de almacenamiento de los valores en el árbol izquierdo o derecho está pensada para buscar eficientemente. La función para comprobar si un valor está en un ABB se incluye en el módulo ABB:

```
#enABB: any AB -> bool
#True si x está en A
#ej: enABB(3,A)->True
#ej: enABB(7,A)->False
def enABB(x,A):
    assert esAB(A)
    if vacio(A):
        return False
    elif x<A.valor:
      return enABB(x,A.izq)
    elif x>A.valor:
      return enABB(x,A.der)
```

```
    else:
      return True  #x==A.valor
assert enABB(3,A)
assert not enABB(7,A)
```

El argumento de búsqueda se compara con el valor para decidir si se encontró o si se debe continuar buscando en el árbol izquierdo o derecho. El proceso de repite hasta encontrarlo o llegar a un árbol vacío. Si el árbol está balanceado, es decir si las alturas de los árboles izquierdo y derecho son iguales (o difieren en uno), entonces se realiza una cantidad máxima de comparaciones similar en cada caso. La función se puede escribir en menos líneas usando instrucciones if sin else:

```
def enABB(x,A):
    assert esAB(A)
    if vacio(A): return False
    if x<A.valor: return enABB(x,A.izq)
    if x>A.valor: return enABB(x,A.der)
    return True  #x==A.valor
```

Para agregar un valor a un ABB se debe crear un nuevo ABB (puesto que las estructuras no pueden modificarse) con los mismos valores que el árbol original, pero agregando el nuevo valor donde corresponda según las comparaciones y en forma similar al proceso de búsqueda.

```
#insertar: any AB -> AB
#nuevo ABB igual a A pero con x
#ej: insertar(2,arbolVacio) -> AB(2,None,None)
#ej: insertar(1,AB(2,None,None)) ->
#    AB(2,AB(1,None,None),None) insertar a la izq
#ej: insertar(3,AB(2,None,None)) ->
#    AB(2,None,AB(3,None,None)) insertar a la der
#ej: insertar(2,AB(2,None,None))->AB(2,None,None) ya existía
def insertar(x,A):
  assert esAB(A)
  #si A esta vacío, crear nuevo árbol con x
  if vacio(A): return AB(x,None,None)
  #insertar x a la izquierda o derecha
  v=A.valor
  if x<v: return AB(v, insertar(x,A.izq), A.der)
  if x>v: return AB(v, A.izq, insertar(x,A.der))
  #si x ya existe, devolver el mismo arbol
  return A
assert insertar(2,arbolVacio)==AB(2,None,None)
assert insertar(1,AB(2,None,None))==AB(2,AB(1,None,None),None)
assert insertar(3,AB(2,None,None))==AB(2,None,AB(3,None,None))
assert insertar(2,AB(2,None,None))==AB(2,None,None)
```

Uso de un Árbol Binario de Búsqueda

Las funciones del módulo con las funciones para operar un árbol binario de búsqueda se pueden utilizar para leer valores y escribirlos ordenados. El programa lee valores que terminan con un punto y los escribe ordenados en la forma indicada en el siguiente diálogo:

```
valor?Juan
valor?Rosa
valor?Ana
valor?.
Ana
Juan
Rosa
```

El programa puede implementarse simplemente con las instrucciones:

```
from ABB import *
escribir(leer())
```

O, en forma equivalente, el programa puede escribirse con las instrucciones:

```
from ABB import *
A=leer()
escribir(A)
```

La función escribir muestra los valores del árbol en orden ascendente escribiendo primero los valores del árbol izquierdo (que son todos menores que el valor), en seguida escribe el valor, y finalmente escribe los valores del árbol derecho (que son todos mayores que el valor).

```
#escribir: AB ->
#escribe valores de A en orden ascendente
#ej: escribir(AB(2,AB(1,None,None),AB(3,None,None)) escribe 1 2 3
def escribir(A):
    assert esAB(A)
    if vacio(A): return
    escribir(A.izq)
    print(A.valor)
    escribir(A.der)
```

La función leer construye y entrega un árbol binario de búsqueda con los valores que se leen usando la función insertar del módulo ABB:

```
#leer: str str -> AB
```

```
#ABB con valores que se leen y terminan con fin
#ej: leer()->AB('a',None,AB('c',AB('b',None,None),None))
#            si lee b c a .
def leer(pregunta='valor?',fin='.'):
    assert type(pregunta)==str
    assert type(fin)==str
    valor=input(pregunta)
    if valor==fin:
        return arbolVacio
    else:
        return insertar(valor,leer(pregunta,fin))
```

Por la manera en que está escrita la función, al leer los datos b, c y a, la función construye el siguiente árbol binario de búsqueda:

```
a
 \
  c
 /
b
```

Alternativamente, la función leer se puede reescribir de manera que el árbol se construya respetando el orden de lectura de los valores. Por ejemplo, si se leen los datos b, c y a, se construirá y entregará el siguiente árbol:

```
 b
/ \
a  c
```

#leer: str str -> AB

```
#ABB con valores que se leen y terminan con fin
#ej: leer()->AB('b',AB('a',None,None),AB('c',None,None))
#            si lee b c a .
def leer(pregunta='valor?',fin='.',A=arbolVacio):
    assert type(pregunta)==str
    assert type(fin)==str
    valor=input(pregunta)
    if valor==fin:
        return A
    else:
        return leer( pregunta, fin, insertar(valor,A) )
```