Per Sedholm
DD1320 (tilda12)
2012-09-05

Övning 2

Listor, pekare, binära träd, rekursion, komplexitet

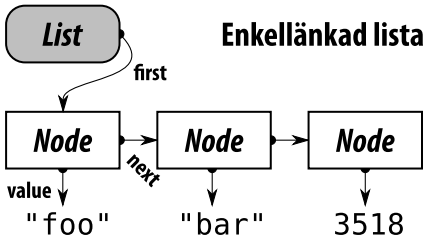
(Länkade) Listor, noder

1. Ta bort andra noden

(a) Skriv en sats som tar bort andra noden ur en länkad lista.

OBS Detta måste fungera:

- om listan är tom
- om listan har ett element
- om listan har två element
- om listan har fler element



Ett exempel på en länkad lista:

- Listan (`List`) har en `first`, som pekar på första noden
- Varje nod (`Node`) har en `next`, och en `value`

```
from MyUtils import LinkedList

def remove_second(lst):
    n = lst.first
    if n != None and n.next != None:
        sec = n.next
        n.next = n.next.next
        return sec
    else:
        raise IndexError('List does not have two elements')

ll = LinkedList()
ll.insert('foo')
ll.insert('bar')
ll.insert('baz')
print(ll)
remove_second(ll)
print(ll)

# <'baz'; 'bar'; 'foo'>
# <'baz'; 'foo'>
```

Vi kan även testa vad som händer när det blir fel:

```
try:
    ll = LinkedList()
    ll.insert('Spanish inquisition')
    print(ll)
    remove_second(ll)
    print(ll)
except IndexError:
    print("<Caught IndexError!>")
```

(b) Nu ska du ta bort andra noden ur en stack. Lös uppgiften abstrakt, alltså med anrop till push och pop!

```
from MyUtils import Stack

s = Stack()

s.push('snap')
s.push('crackle')
s.push('pop')
print(s)
e = s.pop()
s.pop()
s.push(e)
print(s)

# [Stack]{pop; crackle; snap}
# [Stack]{pop; snap}
```

2. Ta bort varannan nod

OBS Tänk igenom och rita bilder för att visa att alla dessa fungerar för *alla* listor:

- tom lista
- lista med ett element
- lista med ojämnt antal element
- lista med jämnt antal element

(a) genom att meka med pekare

- Vi ska ha kvar det aktuella elementet n
- Ifall den har en efterföljare, så ska n_{next} sättas till dess *next*, $n_{\text{next}} \rightarrow_{\text{next}}$, för att hoppa över efterföljaren.

```
def remove_every_second(lst):
    n = lst.first

    while n != None and n.next != None:
        n.next = n.next.next
        n = n.next
```

(b) abstrakt, med hjälp av en extra stack

- Ifall vi har två element, lägg undan ett och släng det andra.
- Varför fungerar det när vi har ett element?
- Hamnar allt i rätt ordning?

```
def remove_every_second(lst):
    stack = Stack()
    while len(lst) >= 2:
        a = lst.getFirst()  # lst.pop(0) för <list>
        #lst.removeFirst()  # ifall inte getFirst gör det
        stack.push(a)       # motsv lst.insert(0, a) för <list>
        b = lst.getFirst()
        #lst.removeFirst()
    while not stack.isEmpty():
        el = stack.pop()
        lst.insertFirst(el) # lst.insert(0, el) för <list>

    return lst
```

(c) abstrakt, med hjälp av rekursion

- Varje anrop tar in listan, *och* returnerar listan till ”ovanstående” rekursionsnivå
- En stack med mindre än två element är klar.
- Annars, ta bort två element, gör rekursivt anrop, stoppa tillbaka.

```
def remove_every_second(lst):
    if len(lst) < 2:
        return lst
    else:
        a = lst.getFirst()
        b = lst.getFirst()
        lst = remove_every_second(lst)
        lst.insertFirst(a)
        return lst
```

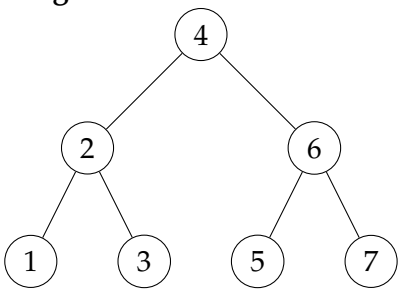
Binära sökträd

1. Trädbygge

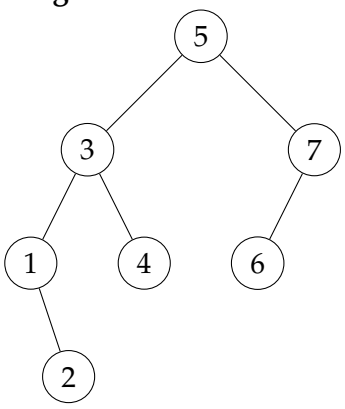
Hur ser det binärträd ut som skapas om man puttar in talen 4 2 1 6 3 7 5 i denna ordning? Och hur ser det ut om man sätter in dom i omvänd ordning, alltså 5 7 3 6 1 4 2? Är båda träden binära sökträd? Är de balanserade?

- I ett *balanserat* träd är höjdskillnaden i alla grenar ≤ 1.
- **Inordning** ⟨ vänster träd ⟩ ⟨ noden ⟩ ⟨ höger träd ⟩
- **Preordning** ⟨ noden ⟩ ⟨ vänster träd ⟩ ⟨ höger träd ⟩
- **Postordning** ⟨ vänster träd ⟩ ⟨ höger träd ⟩ ⟨ noden ⟩

Insättning av 4 2 1 6 3 7 5 $\longrightarrow$ Binärt sökträd: *Ja*. Balanserat: *Ja*.



Insättning av 5 7 3 6 1 4 2 $\longrightarrow$ Binärt sökträd: *Ja*. Balanserat: *Ja*.

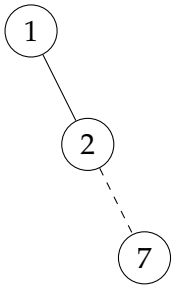


2. Postorderträd

Skriv ut något av träden i inordning och bygg upp ett nytt träd från denna talföljd. Skriv sedan ut dom i preordning och bygg upp nya träd från dessa talföljder. Skriv slutligen ut dom i postorder och bygg upp nya träd från dessa talföljder.

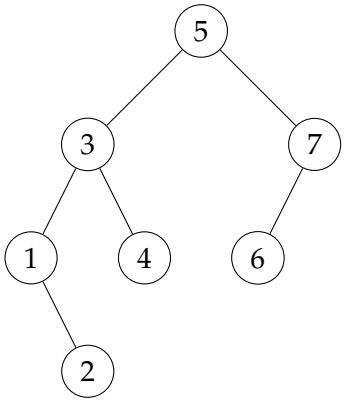
Inordning

Bägge träd blir $1 \ldots 10$ i inordning. Nytt träd blir helt obalanserat (”tarm”)



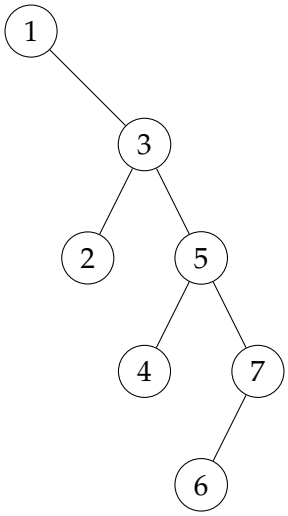
Preordning

Första: $4, 2, 1, 3, 6, 5, 7 \implies$ (ursprungsträdet)
Andra: $5, 3, 1, 2, 4, 7, 6 \implies$

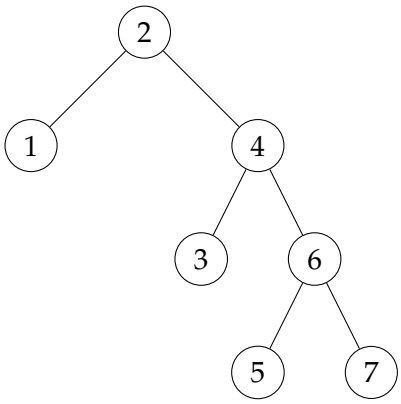


Postordning

Första: $1, 3, 2, 5, 7, 6, 4 \implies$



Andra: $2, 1, 4, 3, 6, 7, 5 \implies$



Rekursion

1. Euklides algoritm för att beräkna GCD

För att beräkna största faktorn som två tal m och n har gemensamt kan vi använda ***Euklides algoritm****:*

> *Om m är jämnt delbar med n så är n den största gemensamma faktorn. Annars är $GCD(m, n) = GCD(n, m \,\%\, n)$.*

Skriv en rekursiv funktion!

```
# -*- coding: utf-8 -*-

def gcd(m, n):
    print("GCD(" + str(m) + ", " + str(n) + ") -> ", end='')
    if m % n == 0:
        print(n)
        return n
    else:
        print("GCD(" + str(n) + ", " + str(m%n) + ")")
        return gcd(n, m % n)

gcd(867, 1989)

# GCD(867, 1989) ->  GCD(1989, 867)    ## 867 < 1989 => 867 % 1989 = 867
# GCD(1989, 867) ->  GCD(867, 255)     ## 1989 = 867*2 + 255
# GCD(867, 255) ->  GCD(255, 102)      ## 867 = 255*3 + 102
# GCD(255, 102) ->  GCD(102, 51)       ## 255 = 102*2 + 51
# GCD(102, 51) ->  51                  ## 102 = 51*2 + 0
```

2. Fibonaccital

Leonardo Fibonacci skrev år 1225 en bok där han beskrev denna intressanta talföljd:

> *Sista december föds en kaninpojke och en kaninflicka. Vid två månaders ålder och varje månad därefter producerar varje kaninpar ett nytt kaninpar. Vi kan skriva en rekursiv formel för antal kaninpar:*

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \end{aligned}$$

Skriv en rekursiv funktion för att beräkna Fibonaccital. Visa vilka rekursiva anrop den ger upphov till vid beräkningen av $f(5)$.

Är det här det effektivaste sättet att beräkna Fibonaccitalen?

```
# -*- coding: utf-8 -*-

def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

print(fib(5))
# fib(5) ->  fib(4) + fib(3)
#   fib(4) ->  fib(3) + fib(2)
#     fib(3) ->  fib(2) + fib(1)
#       fib(2) ->  fib(1) + fib(0)
#         fib(1) ->  1
#         fib(0) ->  0
#       fib(1) ->  1
#     fib(2) ->  fib(1) + fib(0)
#       fib(1) ->  1
#       fib(0) ->  0
#   fib(3) ->  fib(2) + fib(1)
#     fib(2) ->  fib(1) + fib(0)
#       fib(1) ->  1
#       fib(0) ->  0
#     fib(1) ->  1
# 5
```

Man kan visa att antalet anrop faktiskt växer snabbare än själva Fibonaccitalen! (För $n = 40$ är Fibonaccitalet ca hundra miljoner men antalet rekursiva anrop är över 300 miljoner.) Bättre vore här att använda en `for`*-slinga:*

```
def fib(n):
    f0 = 0
    f1 = 1
    for i in range(n-1):
        f2 = f1 + f0
        f0 = f1
        f1 = f2
    return f1
```

Den här implementationen går i linjär tid, $O(n)$.

3. Rekursiv trädsumma

Ge en rekursiv tanke för summan av alla talen i trädet och programmera den så att anropet `tree.sum()` *ger rätt svar.*

```
# Om den står utanför klassen slipper man 'self'
def summa(p):
    if p == None:
        return 0
    else:
        return int(p.value) + summa(p.left) + summa(p.right)

class Bintree(object):
    """_ _ _"""
    def sum(self):
        return summa(self.root)

print("Sum:", b.sum())
```

Med första exemplet från trädbygge:

```
# Sum: 28
#
# summa(<4>) = 4 + summa(<2>) + summa(<6>)
#   summa(<2>) = 2 + summa(<1>) + summa(<3>)
#     summa(<1>) = 1 + summa(None) + summa(None)
#       summa(None) = 0
#       summa(None) = 0
#     summa(<3>) = 3 + summa(None) + summa(None)
#       summa(None) = 0
#       summa(None) = 0
#   summa(<6>) = 6 + summa(<5>) + summa(<7>)
#     summa(<5>) = 5 + summa(None) + summa(None)
#       summa(None) = 0
#       summa(None) = 0
#     summa(<7>) = 7 + summa(None) + summa(None)
#       summa(None) = 0
#       summa(None) = 0
```

4. Rekursiv trädhöjd

Ge en rekursiv tanke för höjden av ett träd. Höjden är den maximala nivån som nån av trädets noder befinner sig på. Ett träd med bara en rotnod har höjden 0*, och ett tomt träd har höjden* -1*.*

Rekursiv tanke: Höjden är $1+\max(\text{höjd vänster}, \text{höjd höger})$ . . . men tomt träd har höjden -1.

Krångligare att läsa utskrifterna, notera att en extra nivå lagts till.

```
def height(p):
    if p == None:
        return -1
    else:
        h1 = height(p.left)
        h2 = height(p.right)
        return max(h1, h2) + 1

class Bintree(object):
    """_ _ _"""
    def height(self):
        return heightl(self.root)

print "Height:", b.height()

# Height: 3
#
#       height(None) = -1
#       height(None) = -1
#     height(<1>) = max(-1,  -1) + 1
#       height(None) = -1
#       height(None) = -1
#     height(<3>) = max(-1,  -1) + 1
#   height(<2>) = max(0,  0) + 1
#       height(None) = -1
#       height(None) = -1
#     height(<5>) = max(-1,  -1) + 1
#         height(None) = -1
#         height(None) = -1
#       height(<10>) = max(-1,  -1) + 1      # <7> replaced with   <7>
#         height(None) = -1                  #                    /   \
#         height(None) = -1                  #                 <10> <11>
#       height(<11>) = max(-1,  -1) + 1
#     height(<7>) = max(0,  0) + 1
#   height(<6>) = max(0,  1) + 1
# height(<4>) = max(1,  2) + 1
```

Komplexitet

1. Körtider

En viss algoritm tar 0.5 ms när antal indata är 100.

Hur lång kommer körtiden att bli när antal indata är 500 om man vet att körtiden är:

- *linjär, dvs* $O(n)$
- $O(n \cdot log(n))$
- *kvadratisk, dvs* $O(n^2)$
- *kubisk, dvs* $O(n^3)$

Linjär: $t \approx k \cdot n$

$$\left.\begin{array}{ll} n = 100 & k \cdot 100 = \frac{1}{2} \\ n = 500 & k \cdot 500 = x \end{array}\right\} k = \frac{0.5}{100} \rightarrow x = \frac{500}{200} = 2.5 \qquad \text{(5 ggr längre)}$$

$O(n\,log(n))$: $t \approx k \cdot n\,log(n)$

$$\left.\begin{array}{ll} n = 100 & k \cdot 100 \cdot log(100) = \frac{1}{2} \\ n = 500 & k \cdot 500 \cdot log(500) = x \end{array}\right\} x = \frac{5 \cdot 500}{2 \cdot 200} \approx 3.37 \qquad \text{(6.74 ggr längre)}$$

Kvadratisk: $t \approx k \cdot n^2$

$$\left.\begin{array}{ll} n = 100 & k \cdot 100^2 = \frac{1}{2} \\ n = 500 & k \cdot 500^2 = x \end{array}\right\} x = \frac{5 \cdot 5}{2} = 12.5 \qquad \text{(25 ggr längre)}$$

Kubisk: $t \approx k \cdot n^3$

$$\left.\begin{array}{ll} n = 100 & k \cdot 100^3 = \frac{1}{2} \\ n = 500 & k \cdot 500^3 = x \end{array}\right\} x = \frac{5 \cdot 5 \cdot 5}{2} = 62.5 \qquad \text{(125 ggr längre)}$$