Föreläsning 9: Prioritetskö, trappa heapsort

- Prioritetskö, Trappa
- Heapsort
- Bästaförstsökning
- Implementation

Prioritetskö

När man poppar en stack får man ut det senast inpushade. När man tar ut något ur en vanlig kö får man tvärtom ut det som legat längst tid i kön. Man skulle kunna se det som att det som stoppas in tidsstämplas och att det påstämplade talet ger prioriteten för uthämtning.

I en prioritetskö stämplas en prioritet på varje objekt som stoppas in och vid uthämtning får man objektet med högst prioritet.

En abstrakt prioritetskö kan ha följande anrop.

`pri_queue.put``(x)`
: Stoppa in x med påstämplad prioritet p.

`x=` **`pri_queue.get`**`();`
: Hämta det som har högst prioritet.

`pri_queue.isEmpty``();`
: Undersök om prioritetskön är tom.

Om det man vill stoppa in i prioritetskön är ett tal kan man använda talet självt som prioritet och bara skriva `put(x)`. Hur den då skiljer sej från en stack och från en vanlig kö ser man av följande exempel.

```
pq.put(1);
pq.put(3);
pq.put(2);
x = pq.get(); // x blir 3
```

En kö hade skickat tillbaka det först instoppade talet 1; en stack hade skickat tillbaka det senast instoppade talet, 2; prioritetskön skickar tillbaka det *bästa* talet, 3. I denna prioritetskö betraktar vi största talet som bäst - vi har en så kallad maxprioritetskö. Det finns förstås också minprioritetsköer, där det minsta talet betraktas som bäst.

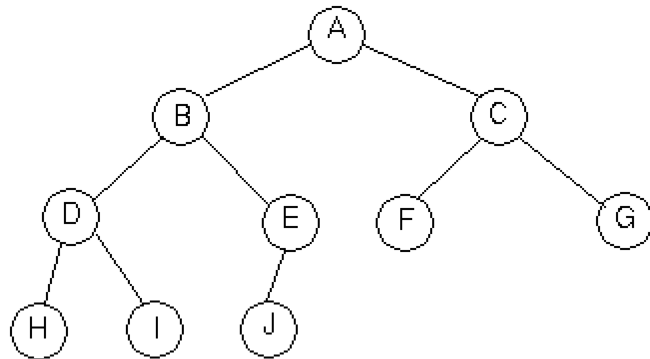
Prioritetsköer har många användningar. Man kan tänka sej en auktion där budgivarna stoppar in sina bud i en maxprioritetskö och auktionsförrättaren efter "första, andra, tredje" gör `pq.get()` för att få reda på det vinnande budet. För att han ska veta vem som lagt detta bud behöver förstås fler uppgifter lagras.

```
pq.put(person)      # person är ett objekt med bud, budgivarens
                    # namn och adress m.m.
winner = pq.get()   # budgivaren  med högst bud
```

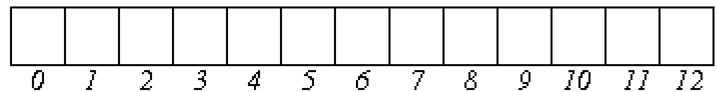
Trappa

Den bästa implementationen av en prioritetskö är en *trappa*, (eng heap), som är en hakvektor `trappsteg` tolkad som binärträd. Roten är `trappsteg[1]` (vi använder inte `trappsteg[0]`), dess båda söner är `trappsteg[2]` och `trappsteg[3]` osv. Allmänt gäller att `trappsteg[i]` har sönerna `trappsteg[2*i]` och `trappsteg[2*i+1]`. Trappvillkoret är att *pappa är bäst*, dvs varje tal ligger på två sämre tal.

Ett nytt tal läggs alltid in sist i trappan. Om trappvillkoret inte blir uppfyllt, dvs om det är större än sin far, byter far och son plats och så fortgår det tills villkoret uppfyllts. Det här kallas *upptrappning* och kan i värsta fall föra det nya talet hela vägen upp till toppen, alltså `trappsteg[1]`.



Man plockar alltid ut det översta talet ur trappan och fyller igen tomrummet med det sista talet i trappan. Då är inte trappvillkoret uppfyllt, så man får byta talet och dess störste son. Denna *nedtrappning* upprepas till villkoret åter gäller.



Både `put` och `get` har komplexitet *log N* om trappan har *N* element. Nackdelen med trappan är man måste bestämma hakvektorns storlek från början.

Sortering med prioritetskö

Om man stoppar in N tal i en trappa och sedan hämtar ut dom ett efter ett får man dom sorterade. Komplexiteten för denna heapsort blir *O(N log N)*, alltså av lika god storleksordning som quicksort. Visserligen är quicksort lite snabbare, men heapsort har inte quicksorts dåliga värstafallsbeteende, och så kan ju en heap användas till andra saker än sortering också.

I java ser programmet ut så här, ett pytonprogram finns på sidan 3.

```
class Heapsort{
   public static void main(String[] args){
      Heap heap = new Heap();
      System.out.println("Skriv några ord (retur avslutar)");
      while (!Mio.eoln())
         heap.put(Mio.getWord());
      while (!heap.isEmpty())
         System.out.println(heap.get());
   }
}
```

Bästaförstsökning

Labb 5 behandlar problemet att finna kortaste vägen från FAN till GUD. Man har då ett problemträd med FAN som stamfar, på nivån därunder sönerna MAN, FIN, FAT osv, på nästa nivå fans sonsöner osv. Om man lägger sönerna i en kö kommer man att gå igenom problemträdet nivå för nivå, alltså breddenförst. Om man byter kön mot en stack blir sökningen djupetförst. Med en prioritetskö får man *bästaförstsökning*, dvs den mest lovande sonen prioriteras och får föda söner.

Exempel 1: Sök billigaste transport från Teknis till Honolulu. All världens resprislistor finns tillgängliga. Problemträdets poster innehåller en plats, ett pris och en faderspekare. Överst i trädet står Teknis med priset noll. Sönerna är alla platser man kan komma till med en transport och priset, till exempel `T-centralen, 9.50`. Man söker en Honolulupost i problemträdet. Med breddenförstsökning får man den resa som har så få transportsteg som möjligt. Med bästaförstsökning får man den billigaste resan.

Exempel 2: Sök effektivaste processen för att framställa en önskad substans från en given substans. All världens kemiska reaktioner finns tillgängliga med uppgift om utbytet i procent. Problemträdets poster innehåller substansnamn och procenttal. Överst i trädet står utgångssubstansen med procenttalet 100. Sönerna är alla substanser man kan framställa med en reaktion och utbytet, till exempel `C2H5OH, 96%`. Med en max-prioritetskö får man fram den effektivaste process som leder till målet.

Körexempel

```
a = PriorityQueue()
infil = file("numbers.txt")
for number in infil:
    print "put %2d into a ->" % int(number),
    a.put(int(number.strip()));
    print a.size, a.elements[1:a.size + 1]
print "=================================================="
while not a.isEmpty() :
    x = a.get();
    print "got %2d a -> %s" % (x , a.elements[1:a.size + 1])
```

utskrift

```
put  3 into a -> [3]
put 43 into a -> [43, 3]
put  6 into a -> [43, 3, 6]
put 12 into a -> [43, 12, 6, 3]
put 52 into a -> [52, 43, 6, 3, 12]
put  7 into a -> [52, 43, 7, 3, 12, 6]
put 75 into a -> [75, 43, 52, 3, 12, 6, 7]
put 65 into a -> [75, 65, 52, 43, 12, 6, 7, 3]
put 29 into a -> [75, 65, 52, 43, 12, 6, 7, 3, 29]
==================================================
got 75 a -> [65, 43, 52, 29, 12, 6, 7, 3]
got 65 a -> [52, 43, 7, 29, 12, 6, 3]
got 52 a -> [43, 29, 7, 3, 12, 6]
got 43 a -> [29, 12, 7, 3, 6]
got 29 a -> [12, 6, 7, 3]
got 12 a -> [7, 6, 3]
got  7 a -> [6, 3]
got  6 a -> [3]
got  3 a -> []
```

Implementation

```
class PriorityQueue:
    size = None
    elements = None
    cmp = None
    def __init__(self) :
        self.size = 0
        self.elements = ["empty"]  # index noll används inte
        self.cmp = cmp

    def isEmpty(self) :
        return self.size < 1

    def put(self, data) :
        self.size += 1
        self.elements.append(data)
        i = self.size
        while (i > 1 and self.cmp(self.elements[i/2], self.elements[i]) < 1) :
            (self.elements[i/2], self.elements[i]) = \
                                 (self.elements[i], self.elements[i/2])
            i = i / 2

    def get(self) :
        if not self.isEmpty() :
            data = self.elements[1]
            self.elements[1] = self.elements[self.size]
            self.size -= 1
            i = 1
            while i <= self.size / 2 :
                j = self.biggestChild(i)
                if self.cmp(self.elements[i], self.elements[j]) < 1 :
                    (self.elements[i], self.elements[j]) = \
                                       (self.elements[j], self.elements[i])
                i = j
            return data
        else: return None

    def biggestChild(self, i) :
        if (2 * i  + 1 > self.size) : return 2 * i
        if self.cmp(self.elements[2 * i], self.elements[2 * i + 1]) > 0 :
            return 2 * i
        else :
            return 2 * i + 1
```

med **`a.cmp = lambda a,b : a < b`** får man en annan utskrift

```
put  3 into a -> [3]
put 43 into a -> [3, 43]
put  6 into a -> [3, 43, 6]
put 12 into a -> [3, 12, 6, 43]
put 52 into a -> [3, 12, 6, 43, 52]
put  7 into a -> [3, 12, 6, 43, 52, 7]
put 75 into a -> [3, 12, 6, 43, 52, 7, 75]
put 65 into a -> [3, 12, 6, 43, 52, 7, 75, 65]
put 29 into a -> [3, 12, 6, 29, 52, 7, 75, 65, 43]
```