Facit och kommentarer till tentamen 2019-08-23 i DA3018

1.
   - En hash-funktion bör ha god spridning över sin domän och ha liten sannolikhet för krockar.
   - Man använder ordo-notation eftersom man är intresserad av att uttrycka hur en funktion växer utan att behöva hantera mindre viktiga termer och svårbestämda konstanter.
   - Fördel: Går att testa om en kant finns i konstant tid. Nackdelar: Använder onödigt mycket minne för glesa grafer. Onödigt arbetsamt att iterera igenom kanter, särskilt på glesa grafer.
   - Eftersom det enda man måste göra är att avgöra om alla på varandra följande element uppfyller $<$, så är det linjär tid.
   - Stacken följer principen “First in, last out”: man lägger nya element på “toppen” och man har bara tillgång till det senast inlagda elementet. Standardoperationerna är “push” (lägg ett nytt element på stacken), “pop” (plocka bort och returnera senast tillagda element), och “isEmpty” (predikat som returnerar Sant om stacken är tom och Falskt annars).

2.
   - Antag att `elements` har längd m och `data_list` har längd n. Funktionen itererat igenom `elements` (m iterationer). I varje iteration två tilldelningar och en jämförelse, som innebär $O(1)$ arbete, samt en uppslagning i `t` (en uppslagstabell) och ett anrop till `index_of`. Vi kan anta att uppslagstabellen är bra implementerad och tar tid $O(1)$ i genomsnitt. Anropet till `index_of` måste antas ta $O(n)$ tid eftersom det tillämpas på en enkellänkad lista och man måste då vara beredd på att iterera igenom alla element. Sammantaget ger det tidskomplexiteten $O(mn)$.
   - Funktionen är inte lämplig, i alla fall inte utan ändringar. Om ett ord finns på mer än en position kommer bara den sista lagras i uppslagstabellen, så det är inte tillräckligt med information som sparas.

3. (a) Funktionen identifierar det minsta elementet, men plockar inte bort det från `input_list`. Därför kommer ett och samma element läggas till i `sorted_list` i varje iteration. Om vi ändrar koden till

```
def selection_sort(input_list):
    sorted_list = []
    n = len(input_list)
    for i from 0 to n-1:
        elem = min(input_list)    // Find smallest
            elem
        input_list.remove(elem)   // Remove that
            element
        sorted_list.append(elem) // Put it last in
            the output list
    return sorted_list
```

får vi önskad funktion: minsta kvarvarande element identifieras och läggs sist i en ny lista. Metoden `remove` används i Python och betyder ”ta bort första instansen av detta element”. Man kan tänka sig liknande (och mer effektiva) sätt att göra samma sak.

(b) Funktionen gör n (= längden på indatalistan) iterationer och i varje iteration görs det dels lite arbete i konstant tid (inklusive `append`), dels anrop till **`min`** och `remove` som båda måste räknas ta $O(n)$. Sammantaget ger det tidskomplexiteten $O(n^2)$.

(c) Vi måste se till att det är det första instansen av minsta elementet som plockas bort från `input_list`. Om det finns två element $x_1 = x_2$ i indata är det inte stabilt att flytta x_2 till `sorted_list` innan x_1.

(d) Elementen, i den ordning de är givna i tabellen, uppfyller heap-villkoret. Dvs, om vi svarar 30 26 25 23 22 22 20 17 15 9 4 4 så kommer element i vara större än element $2i$ och $2i+1$ (1-indexerat).

(e) Standardalgoritmen för att ta bort största elementet är att flytta upp sista elementet till första position samtidigt som man förkortar heapen. Sen använder man algoritmen *heapify* på första positionen tills att heap-villkoret är uppfyllt igen.

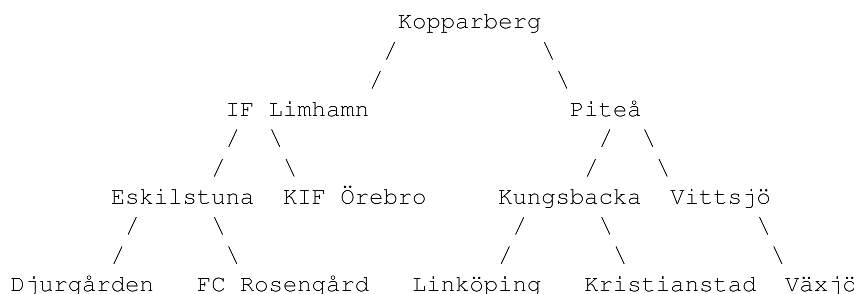
Det gör att heapen utvecklas så här:

Iteration	heap											
Start	30	26	25	23	22	22	20	17	15	9	4	4
1	**4**	26	25	23	22	22	20	17	15	9	4	
2	26	**4**	25	23	22	22	20	17	15	9	4	
3	26	23	25	**4**	22	22	20	17	15	9	4	
4	26	23	25	20	22	22	**4**	17	15	9	4	

Efter fjärde iterationen dominerar är $26 \geq 23 \geq 20 \geq 4$, och därmed är heap-villkoret uppfyllt.

(f) Nej, `find(key)` kan vi inte få i tid $O(\log n)$, eftersom en heap har elementen partiellt ordnade. Vi kan inte binärsöka i den array som ligger till grund för heapen, och vi kan inte söka från första elementet, eftersom vi kan inte veta i vilket delträd under roten som `key` ligger.

4. (a) Här är ett av många möjliga balanserade sökträd:



Det är balanserat eftersom höjden på varje delträd skiljer sig högst 1 från dess syskondelträd. Jag nöjer mig med argumentet att det finns ungefär lika många element i varje par av syskondelträd.

(b) Exempelkod:

```
def print_sorted_keys(bst):
    if not is_leaf(bst):
        print_sorted_keys(bst.left())
        print(bst.get_key())
        print_sorted_keys(bst.right())
```

Denna rekursiva algoritm är korrekt eftersom den tar hand om det triviala basfallet och i annat fall ser till att rekursivt skriva ut det vänstra delträdet (vars element alla ska skrivas ut innan aktuell nyckel), därefter aktuell nyckel, samt avslutningsvis skriver ut det högra delträdet. Rekursionen följer sökträdets ordning.

(c) Möjlig pseudokod:

```
def check_if_present(key, bst):
    if bst:
        if key == bst.get_key():
            return True
        elif key < bst.get_key():
            return check_if_present(key, bst.left()
                )
        else:
            return check_if_present(key, bst.right
                ())
    else:
        return False
```

(d) Antag att det finns n element i sökträdet. I värsta fall måste vi söka ända till lägsta nivån och upptäcka att nyckeln inte finns i trädet. Om elementet finns i trädet kan vi avbryta tidigare. Vi vill visa att höjden på ett balanserat träd är $O(\log n)$.
XXX

5. (a) En möjlig kärnfull beskrivning: ”Sortera hörnen efter gradtal, så att de med få grannar kommer för de med många grannar. Iterera sen över alla hörn v, och lägg v i lösningsmängden om vi inte tidigare har bestämt att v är granne till ett hörn som är i lösningsmängden. Avfärda v:s grannar så att de inte kan väljas.”

Detta är ett exempel på en *snål* heuristik. Notera att oberoende mängd inte har en någon känd effektiv algoritm, utan tillhör en klass av ”svåra problem” som kallas NP-fullständiga. Det kommer ni läsa om i nästa algoritm kurs (DA3004).

(b) Jag antar att vi implementerar mängder med hjälp av hashtabeller. Insättning och test av medlemskap går i förväntad tid $O(1)$. Grafen antas vara lagrad med grannlistor, så att iterera igenom grannar går i linjär tid.

Vi kan vara pessimistiska och anta att ett hörns gradtal inte är givet, utan måste beräknas från dess lista med grannar. Att bestämma alla hörns gradtal tar tid $O(|V| + |E|)$, eftersom vi måste gå igenom alla hörn och deras kantlistor, och summan av alla kantlistors längde är $2|E|$.

Efter att gradtalen är beräknade ska vi sortera hörnen, vilket tar tid $O(|V| \log |V|)$.

Därefter är det dags att iterera igenom alla hörn och bestämma om de tillhör lösningsmängden eller inte. Det beslutet och associerat arbete tar konstant tid. Men sen måste vi för varje utvalt hörn gå igenom dess grannar och se till att de inte får vara med i lösningen. Det arbetet är linjärt i grannlistans längd, och sammantaget tar allt det arbetet tid $O(|E|)$.

Sammantaget är tidskomplexiteten för oberoende mängd $O(|V| + |E| + |V| \log |V| + |V||E|) = O(|V||E|)$.