Python 3.6 — 10 poeng

Anta at Python sine ordbøker (`dict`), frem til versjon 3.6, var implementert med linear probing slik det har blitt undervist i IN2010. (Dette er en liten modifikasjon av sannheten).

I desember 2012 kom Raymond Hettinger med et forslag til en endring av den underliggende representasjonen av ordbøker. I forslaget beskriver Hettinger at den gjeldende representasjonen er unødvendig ineffektiv.

Den organiserer dataene i et array `entries`, der hvert element har plass til en nøkkel k og en verdi v. Arrayet har størrelse N, og mye ledig plass. Hettinger foreslår å omorganisere dataene. Nøkkel-verdi-parene bør i stedet lagres i et tettpakket, dynamisk array `entries` med n elementer, der n er antall nøkkel-verdi-par. Elementene i `entries` kan refereres til fra et annet array `indices` med N elementer (som fremdeles har mye ledig plass).

For eksempel, følgende ordbok:

```
d = {}
d["timmy"] = "red"
d["barry"] = "green"
d["guido"] = "blue"
```

var tidligere representert slik:

```
entries = [(_, _),
           ("barry", "green"),
           (_, _),
           (_, _),
           (_, _),
           ("timmy", "red"),
           (_, _),
           ("guido", "blue")]
```

der `(_, _)` indikerer en ledig plass. Forslaget er å representere den slik:

```
indices = [_, 1, _, _, _, 0, _, 2]
entries = [("timmy", "red"), ("barry", "green"), ("guido", "blue")]
```

Forslaget over førte til at implementasjonen av ordbøker i Python ble endret fra og med versjon 3.6.

(a) Forklar kort, med naturlig språk, hvordan algoritmene for oppslag og innsetting med linear probing må endres for å ta høyde for den nye strukturen.

> Maksimalt 4 poeng. Omtrent 2 poeng for oppslag og 2 poeng for innsetting.
>
> I den nye strukturen vil oppslaget fremdeles starte med å beregne hashen $i = h(k, N)$ av nøkkelen k, der N er størrelsen av `indices`. Vi vil så slå opp `indices`$[i]$, og dersom den ikke er tom, så vil vi sammenligne nøkkelen i `entries`, som da kan finnes på `entries`$[$`indices`$[i]]$. Dersom nøkkelen ikke stemmer overens med nøkkelen k, vil vi øke i med 1 (modulo N), og slå opp på nytt. Dersom vi treffer en tom plass avsluttes søket.
>
> Det viktigste poenget for oppslag er at vi slår opp i `entries` via `indices`.
>
> For innsetting finner vi første ledige plass i `indices` med samme teknikk som ved oppslag. Hvis den første ledige plassen er j, så settes `indices`$[j]$ til størrelsen av `entries`. Til slutt legger vi til nøkkel-verdi-paret på slutten av `entries`.

(b) Hva er en rehash? Hva er det som forårsaker en rehash?

> Maksimalt 2 poeng.
>
> En rehash forårsakes av at load faktoren $\frac{n}{N}$ er for høy, altså at det ikke er så mye ledig plass. Da øker vi størrelsen på det underliggende arrayet (som var av størrelse N), og setter inn alle elementene på nytt (med innsettingsalgoritmen til linear probing).

(c) Hvordan vil en rehash foregå i den nye strukturen?

Maksimalt 2 poeng.

I den nye strukturen vil `entries` kunne forbli uendret. Vi vil heller øke størrelsen på `indices`, og gjøre innsetting på alle elementene som beskrevet ovenfor (med unntak av å legge til i `entries`).

(d) Hva er fordelene med den nye strukturen?

Maksimalt 2 poeng.

Her er det flere fordeler:

- Elementene kan hentes ut i en forutsigbar rekkefølge (altså, rekkefølgen de ble satt inn i).
- Iterering krever n steg i stedet for N steg (og N er alltid større enn n).
- Rehashing påvirker ikke `entries`.
- Lavere minnebruk, som følge av at det største arrayet `indices` kun trenger å inneholde indekser. I praksis bruker den 20% til 25% mindre minne. Disse tallene tallene er ikke mulig å forutsi fra beskrivelsen over, men man kan kanskje se at det vil resultere i lavere minnebruk.

Det gis full uttelling hvis to av momentene ovenfor blir nevnt. Listen over er ikke nødvendigvis komplett, så andre gode poenger kan også gi uttelling.