Øving 4, algoritmer og datastrukturer

Det er to deloppgaver, gjør begge.

Problemer? Sjekk tipsene på slutten, de fleste nybegynnerfeilene står der.

Deloppgave 1: Hashtabell med tekstnøkler

Implementer en hashtabell som kan lagre tekststrenger. Om dere bruker java, er det naturlig å la *hashtabell* være en klasse. Ikke bruk java sin ferdiglagde hashtabell. Håndter kollisjoner med lenka lister. Lag en passende hashfunksjon.

Strenger kan konverteres til heltall ved hjelp av ascii/unicodeverdier. Bruk en løkke som plukker ut tegnene i en streng og beregner et heltall. Dette tallet kan deretter brukes på vanlig måte med hashfunksjonene. Tegnene i strengen bør vektes ulikt, slik at f.eks. «Caro» og «Cora» får ulik nøkkel, selv om navnene har de samme bokstavene.

Vedlagt er en fil som heter *navn,* som inneholder navn på de som er involvert i dette faget. La programmet legge disse inn i hashtabellen, og implementer oppslag så vi kan spørre hvem som er med i faget. Ikke lag hashtabellen mye større enn nødvendig. Det selvfølgelig lov å ha den litt større for å komme opp til nærmeste toerpotens eller primtall. Men unngå halvfull tabell.

La testprogrammet skrive ut lastfaktoren til slutt, og la det skrive ut hver eneste kollisjon som oppstår under innsetting og søk. (Skriv ut de to navnene som kolliderer.) Tell opp totalt antall kollisjoner under innsetting, og skriv det ut sammen med lastfaktoren. Beregn og skriv ut gjennomsnittlig antall kollisjoner per person. (kollisjoner/personer). Antall kollisjoner per person bør under 3.

`http://www.iie.ntnu.no/fag/_alg/hash/navn` (Filen er i unicode)

Krav for godkjenning del 1:

- Programmet leser fila med navn, og klarer å legge alle i hashtabellen.
- Kollisjoner håndteres med lenka lister
- Programmet skriver ut alle kollisjoner, og lastfaktoren til slutt

- Programmet klarer å slå opp personer i faget ved hjelp av hashtabellen. (Bruk gjerne oppslag på eget navn.)
- Legg ved utskrift fra kjøringen.

Deloppgave 2: Hashtabeller med heltallsnøkler – og ytelse

1. Implementer en hashtabell med plass for ti millioner tall. Hashtabellen kan selvfølgelig ha noe mer enn ti millioner plasser, både for ytelsens skyld, og med tanke på nærmeste primtall eller toerpotens. Men ikke lag hashtabellen mer enn 35% større enn nødvendig.

   Håndter kollisjoner med dobbel hashing, og bruk hashfunksjoner som egner seg for denne anvendelsen.

2. Fyll en annen tabell (ikke hashtabell) med ti millioner tilfeldige tall. Tallene må være spredt over et område som er mye større enn tabellstørrelsen. Vi gjør klar de tilfeldige tallene på forhånd, da randomfunksjonen er mye tregere enn en god hashfunksjon. Vi vil måle tiden på hashoperasjonene, ikke Math.Random.
3. Tell opp kollisjoner. La programmet skrive ut antall kollisjoner og lastfaktor etter tidsmålingen.
4. Ta tiden på å sette inn de tilfeldige tallene i hashtabellen deres. Sammenlign med tidsforbruket for å sette de samme tallene inn i java.util sin HashMap.[1]

Krav til godkjenning av del 2:

- Et program som legger 10 millioner tilfeldige tall inn i en hashtabell, og måler tiden
- Kollisjoner håndteres med dobbel hashing
- Sammenligning med tid for å sette tallene inn i Hashmap. (Det er ikke nødvendig å «slå java», men det går an.)
- Utskrift som viser tidsmålingene, lastfaktor og antall kollisjoner.

[1] De som evt. gjør oppgaven i et annet språk enn java, sammenligner med en ferdiglaget hashtabell. De fleste programmeringsspråk har hashtabeller i sine biblioteker.

Tips

- Hashtabeller er normalt raske. Det kan bli problemer med tidtaking, bruk i så fall trikset fra første øving, hvor operasjonen kjøres flere ganger på rad.
- Hvis det tar flere *minutter* å sette tallene inn i hashtabellen, har dere problemer med hashfunksjonene deres. Vanlige feil her:
  - h_2 blir 0 for noen nøkler. Det fører i så fall til en uendelig løkke så programmet aldri blir ferdig. Pass på at h_2 aldri kan bli 0, og heller aldri lik tabellstørrelsen.
  - h_1 klarer ikke å spre nøklene utover *hele* tabellen. I så fall får dere altfor mange kollisjoner, og tidsforbruket blir kvadratisk. (n nøkler kolliderer, og de $n/2$ siste kolliderer med de $n/2$ første, ca $n^2/4$ kollisjoner.)
  - De tilfeldige tallene ligger i et smalere intervall enn tabellstørrelsen, så det blir veldig mange duplikater blant ti millioner tall. Tall med *samme verdi* kolliderer alltid – og dobbel hashing hjelper heller ikke i slike tilfeller! Bruk et større intervall enn tabellstørrelsen.
  - Dere har brukt et interpretert språk, (f.eks Python) i stedet for et kompilert programmeringsspråk. Bruk i så fall en mindre tabell, kanskje hundre tusen. Python er ikke ment for så store datasett.
- Se opp for «arithmetic overflow», altså operasjoner som lager altfor store heltall. Ta f.eks. en uheldig beregning som «`pos=(h1+i*h2)%m`». Her kan `i` blir svært stor, når tabellen nærmer seg full. Da blir `i*h2` større enn området for `int`, og slik overflow kan føre til negative tall. Deretter følger en «index out of bounds exception» når programmet slår opp en negativ tabellindex.

  Løsningen er enkel. Bruk heller en løkke med «`pos=(pos+h2)%m`» Det er bra fordi:

  1. Negative tall oppstår ikke. Vi har ikke multiplikasjon, bare addisjon som er mindre voldsomt. Mod-operasjonen runder ned så snart `pos` blir litt større enn `m`.
  2. Programmet blir litt raskere fordi vi har en multiplikasjon mindre i løkka. Hashtabeller handler om raskest mulig kode.
- Poenget med hashtabeller er hastighet! Unngå å gjøre unødig arbeide:
  - For å sette inn et tall, må vi beregne h_1. Hvis det ikke kolliderer, er det ikke nødvendig å beregne h_2 for det tallet.
  - Ved kollisjon trengs h_2. Men h_2 trenger bare beregnes én gang, ikke for hvert eneste forsøk på å plassere tallet.