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. Dere skal lagre navnene på alle som tar kurset, og slå opp eget navn i tabellen.

Om dere bruker java/c++, er det naturlig å la *hashtabell* være en klasse. Ikke bruk java sin ferdiglagde hashtabell. Håndter kollisjoner med *lenka lister*. Disse listene implementerer dere selv, som trening i å lage lenka lister. Ikke bruk LinkedList her.

Lag en passende hashfunksjon for navnene.

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. Ikke vekt noen tegn med 0, det er for dumt.

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. Det er ikke nødvendig å skrive ut alle navnene i den lenka listen, det holder med det første.) 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).

`https://www.idi.ntnu.no/emner/idatt2101/hash/navn.txt` (Filen er i unicode) Dere trenger ikke levere fila sammen med øvingen, jeg har min egen kopi.

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.)
- Programmet bruker «navn.txt», *ikke* «mappe/navn.txt». Retting tar for lang tid, om jeg må håndtere hundrevis av mapper. Så det skjer ikke.
- Legg ved utskrift fra kjøringen.
- Antall kollisjoner pr. person er under 0,4 (men ikke eksakt 0, regn med desimaltall…)

Vanlige feil

- Program som ikke håndterer kollisjoner, så alle blir ikke med.
- Hashfunksjonen sprer ikke folk utover, for mange kollisjoner
- Kollisjoner pr. person blir 0, fordi heltallsdivisjon runder nedover.
- Noen leser ikke oppgaven, og bruker ikke lenka lister :-(

Deloppgave 2: Hashtabeller og ytelse

Hashtabeller yter best med få kollisjoner. Vi skal sammenligne forskjellige former for åpen adressering.

1. Finn ut hvor stor hashtabell (m) du vil ha. Den skal ha plass til minst ti millioner tall, men hashfunksjonene du velger kan jo legge egne føringer for størrelsen. (Primtall, toerpotens.)

   Ti millioner tall er for å få nok arbeid så tidtaking fungerer. De som bruker python, kan bruke en million eller hundre tusen, hvis større tabeller går for tregt.

2. Gjør klar en tabell med m unike tilfeldige tall. Poenget er å kunne teste *de samme* tallene i ulike typer hashtabeller. (Unike tilfeldige tall kan f.eks. lages ved å la hvert tall være lik det forrige, pluss et tilfeldig tall mellom 1 og 1000.) Tabellen må stokkes, så tallene ikke ligger i stigende rekkefølge. Noen språk har shuffle-metoder. En annen grei måte, er å la hvert element fra 0 til $m - 1$ bytte plass med et tilfeldig valgt element.

3. Implementer hashtabeller med åpen adressering. En med lineær probing, og, en med dobbel hashing. Alle med eksakt samme størrelse. (Et godt objektorientert design kan ha en baseklasse, og to subklasser.)

   Hashtabellene skal også telle opp alle kollisjoner ved innsetting, så vi kan se hva som fungerer best. Et tall som settes rett inn på den plassen h_1 finner, kolliderer ikke. Om det trengs flere forsøk, teller disse som kollisjoner.

4. Gjennomfør forsøk for å se hvor mange kollisjoner det blir med ulike fyllingsgrader og ulike typer hashing. Prøv å sette inn 50% av tallene, 80%, 90%, 99% og 100%, i hver av de to typene hashtabell.

   La programmet skrive ut antall kollisjoner for hvert eksperiment. (Ikke hvert enkelt tall!) Ett eksperiment er f.eks. «sette inn 90% av tallene, i en hashtabell med lineær probing.» Ta også tiden på hvert eksperiment.

5. Lag en kort rapport, med tabeller (og evt. kurver/illustrasjoner) som viser tidsbruk og hvor mange kollisjoner det blir for ulike fyllingsgrader og type hashtabell.

6. Se om det går an å trekke noen konklusjoner ang. antall kollisjoner og tidsbruk:

   a) Tar fler kollisjoner mer tid?

   b) Er det grenser for hvor full en hashtabell bør være?

   c) Er ytelsen (i tid eller kollisjoner) ulik for ulike typer hashing? Er noen klart best, eller best ved visse fyllingsgrader?

   d) Andre interessante observasjoner?

Krav til godkjenning av del 2:

- Oppgaven er gjort, med to typer åpen adressering og mange fyllingsgrader.
- Hashtabellene klarer å ta imot alle tallene. Å gi opp å plassere et tall, er *feil.*

Tips

- 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. Duplikater kolliderer alltid, derfor sier oppgaven at dere skal ha *unike* tall.
- 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 veldig 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`». (dobbel hashing) Her kan `i` blir svært stor, når tabellen nærmer seg full. Da blir `i*h2` større enn området for `int`, og aritmetisk 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 av flere grunner:

  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. Å unngå multiplikasjon hjelper. Om man kan erstatte mod-operasjonen med en maskering, blir det enda bedre :-)

- Poenget med hashtabeller er hastighet! Prøv å 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.