NTNU
Kunnskap for en bedre verden

Algoritmer og datastrukturer - IDATT2101

Øving 4 - Hashtabeller

Author:
Henrik Halvorsen Kvamme

Dato: 19th September 2024

Table of Contents

1 Introduksjon **1**

2 Teori **1**

3 Resultater **1**

3.1 Del 1: Hashtabell-implementasjon 1

3.2 Del 2: Ytelsessammenligning 2

4 Konklusjon **2**

1 Introduksjon

Oppgaven er todelt. I del 1 avoppgaven skal man implementere en hashtabell som kan lagre tekststrenger. Dataen vi går ut i fra er navnet på alle som deltar i kurset. Dette er kravene for å få oppgaven godkjent:

- 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.)
- Programmetbruker«navn.txt»,ikke«mappe/navn.txt».Rettingtarforlangtid,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 desimal- tall...)

For del 2 skal man nå håndtere kollisjoner med dobbel hashing. I stedet for streng-nøkler setter vi nå inn veldig mange tall for å måle ytelsen. Dette er kravene for å få den godkjent:

- 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 :-(

2 Teori

For at hashmap implementasjonene skal være effektive bør de ha en god hashfunksjon. En hashfunksjon tar inn en verdi og omformer den til å bli en tallverdi slik at den kan brukes som index til en tabell.

For deloppgave to gjør jeg dette ved å multiplisere verdien med et primtal og tar så modulus av størrelsen til tabellen. Tabellstørrelsen bør helst være et primtal. Jeg satt det til å være det nærmeste primtallet over 10 000 000 for å møte kravet, altså m = 10 000 019. Dobbel hash teknikken bruker to forskellige hash-funksjoner. En for å velge første plassering og en for hvert steg den tar.

3 Resultater

3.1 Del 1: Hashtabell-implementasjon

For deloppgave 1 oppnådde vi følgende resultater:

- Antall kollisjoner: 33
- Kollisjoner per element: 0,266129
- Lastfaktor: 0,716763

Basert på disse resultatene kan vi konkludere at alle krav for å godkjenne oppgaven er oppfylt.

3.2 Del 2: Ytelsessammenligning

For å måle kjøretiden benyttet vi oss av `std::chrono`. Resultatene viser:

Metode	**Tid**	**Relative ytelse**
Vår hashtabell	564 ms	100%
Standard C++ map	11825 ms	4,77%

Table 1: Ytelsessammenligning for innsetting av 75% av m elementer

For vår hashtabell-implementasjon observerte vi:

- Antall kollisjoner: 6 459 557
- Kollisjoner per element: 0,861273
- Lastfaktor: 0,75

Vår implementasjon brukte kun 4,77% av tiden som standard C++ `map` brukte for å sette inn samme antall elementer, noe som demonstrerer en betydelig ytelsesforbedring.

4 Konklusjon

Ut i fra resultatene kan vi konkludere med at min implementasjon er betydelig raskere enn standard c++ biblioteket i dette tilfellet. Det vil ikke si at jeg er noe bedre programmerer enn de som har skrevet map implementasjonen. Grunnen til at min implementasjon var raskere er nok fordi jeg har definert størrelsen fra før av, og at det ikke måtte bestemmes dynamisk.