Øving 5 Algoritme og datastrukturer IDATT2101

# *Deloppgave 1: Hashtabell med tekstnøkler*

# Implementasjon

For å løse deloppgave 1 ble klassen HashArray implementert. Denne klassen bruker tekststrenger for å lage en hash til tabellen. For at kollisjoner skal bli håndtert av lenka lister, har jeg valgt å implementere klassene Node og LinkedList.

I implementasjonen av hashfunksjonen har jeg lent meg på eksempelet fra forelesningen 26.09.2023. I denne funksjonen konverteres hvert tegn i en streng til tall (gjennom ASCII tabellen), som vi kan bruke for å danne en hash verdi. Videre vektes hvert tegn ved å gange med primtallet 17 slik at forskjellige ord med like mange korresponderende tegn ikke gir samme hash verdi. Figuren under viser hvordan dette er implementert.

Et bilde som inneholder tekst, skjermbilde, Font, line

Automatisk generert beskrivelse

Figur : Utsnitt av hashfunksjonen i deloppgave 1.

For å legge inn navn fra fila *navn.txt* er en egen metode implementert for å lese fra nettadressen: <https://www.idi.ntnu.no/emner/idatt2101/hash/navn.txt>. Denne metoden heter readFile som figuren under viser:

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : utsnitt av metoden readFile som returnerer en liste over alle navnene i fila navn.txt.

Deretter kan vi legge inn disse navnene inn i tabellen ved bruk av metoden putIn som hasher alle navnene ved bruk av hashfunksjonen vist på forrige side. Denne metoden håndterer også kollisjoner ved å legge til en ny node bakerst i den lenka listen. For hver kollisjon som oppstår, blir navnene skrivet ut til terminalen sammen med indeks.

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : Utsnitt av metoden putIn som putter in et navn i tabellen.

# Resultat

Etter å ha lest navn.txt og puttet alle navnene inn i tabellen, blir alle kollisjonene som har inntruffet vist på terminalen. Deretter vises det hvor mange kollisjoner det ble totalt, gjennomsnittlig antall kollisjoner per person og lastfaktoren (). Til slutt for å bevise at jeg er involvert i dette emnet slår jeg opp mitt eget navn i tabellen. Der finner jeg ut at jeg er plassert på indeks 54. Figuren under viser en liten utskrift fra kjøringen.

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : Utsnitt av terminalen fra kjøringen. NB: her har jeg ikke tatt med utskriften av de 44 kollisjonene pga. plass.

# *Deloppgave 2: Hashtabeller og ytelse*

# Implementasjon

For å løse deloppgave 2 har jeg implementert den abstrakte klassen OpenAddressering. Denne klassen holder på en tabell som skal holde på unike og tilfeldige verdier. For denne klassen har jeg deretter implementert to subklasser: LinearProbe og DoubleHash. LinearProbe, som representer den enkleste formen for åpen adressering, prøver hvis det oppstår en kollisjon ved å sette inn et tall til neste posisjon. Hvis det oppstår en kollisjon på denne posisjonen, prøver den neste posisjon osv. Figuren under illustrerer hvordan dette skjer.

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : Utsnitt av metoden putIn som er overskrevet av LinearProbe.

Hashfunksjonen til LinearProbe er ganske enkel i form av at simpelthen ganger med en konstant 2003 som er et primtall (også det året jeg ble født). Til slutt tas modulo av størrelsen til denne tabellen slik at det ikke oppstår IndexOutOfBoundsException. Grunnen til at primtallet 2003 blir ganget med er for å oppnå spredning og redusere antall kollisjoner.

Et bilde som inneholder tekst, Font, skjermbilde, line

Automatisk generert beskrivelse

Figur : Utsnitt av hashfunksjonen til klassen LinearProbe.

For DoubleHash, som representerer den beste formen for åpen adressering, brukes det en annen tilnærming. Her brukes to hashfunksjoner hvor den første angir den første posisjonen vi skal prøve, mens den andre angir hvor langt vi skal hoppe hvis det blir en kollisjon. Figuren under illustrerer dette poenget.

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : Utsnitt av metoden probe i klassen DoubleHash.

Hashfunksjonene til DoubleHash klassen er vist nedenfor. Den første hashfunksjonen er identisk til hashfunksjonen i LinearProbe klassen, men det interessante er det som skjer i den andre hashfunksjonen. Her deklarerer jeg først en variabel *hash* som skal returneres. Men for at probesekvensen skal gå innom alle posisjonene er det nødvendig at variabelen *hash* og størrelsen til tabellen er relativt primiske. Dette kan sjekkes med at den største fellesdivisoren er 1. En enklere tilnærming ville bare ha implementert at størrelsen til tabellen er et primtall, og kan dermed slippe å sjekke for største felles divisor. Jeg synes derimot at størrelsen burde være valgfritt (minst 10 000 000 i størrelse), og derfor burde koden være mer fleksibel. En trade-off med dette er dessverre økt kompleksitet.

Et bilde som inneholder tekst, Font, skjermbilde, line

Automatisk generert beskrivelse

Figur : Utsnitt av den første hashfunksjonen til klassen DoubleHash.

Et bilde som inneholder tekst, skjermbilde, Font, line

Automatisk generert beskrivelse

Figur : Utsnitt av den andre hashfunksjonen til klassen DoubleHash som returnerer en hash som er relativt primisk med tabell størrelsen.

# Tidsmålinger

For å gjøre tidsmålinger med ulike typer hashing på de ulike fyllingsgradene med 50, 80, 90, 99 og 100%, har jeg implementert metoden timer() slik figuren nedenfor illustrerer.

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : Utsnitt av metoden timer som tar tiden for de ulike fyllingsgradene med både dobbel hashing og linear probing.

Metoden deklarer og initialiserer de to type hash-klassene LinearProbe og DoubleHash, og en tabell med unike og tilfeldige long verdier. Metoden generateRandomArray shuffler også tabellen slik at tallene ikke kommer i stigende rekkefølge. For de forskjellige double verdiene som representerer fyllingsgradene har jeg også implementert en metode divideArray som deler opp en tabell i en valgt porsjon. Med for-løkkene itereres hvert eksperiment med forskjellig fyllingsgrad og hash type.  
  
Et resultat med tabell størrelse på akkurat 10 000 000 vil produsere utskriften som figur 11 viser. Grunnen til at jeg har peilet meg på minsteverdien i denne oppgaven, er for å slippe å vente så iherdig lenge på tidsmålingen fra lineær probing med fyllingsgrad på 100%.

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelse

Figur : Utskrift fra tidsmåling med tabellstørrelse på 10 000 000.

# Oppsummert data

Fra tidsmålingen i figur 11 fikk vi disse dataene:



Det er dermed mulig å forme flere diagrammer for å se om det går an å trekke noen slutninger.

Figur : Linjediagram over tid, fyllingsgrad og kollisjoner for dobbel hashing.

Figur : Linjediagram over tid, fyllingsgrad og kollisjoner for lineær probing.

1. Ut ifra disse diagrammene ser kan vi se at det er en direkte korrelasjon mellom kollisjoner og tid. Det beste eksempelet er kanskje figur 13 med lineær probing.
2. For lineær probing ser det ut til at grensen for full en hashtabell burde være tydelig er 99%. På 100% hopper tidsforbruket og antall kollisjoner ekstremt. For dobbel hashing ser det ut til at det går greit med 100% kapasitet ettersom tidsforbruket ikke øker så voldsomt, selv om antall kollisjoner noe.
3. Ut ifra dataene jeg har samlet inn fra kjøringen vist på figur 13, ser vi at kollisjonsytelsen er klart best for dobbel hashing. Men det betyr ikke at lineær probing bruker lengre tid. Tvert imot bruker lineær probing kortere tid på fyllingsgradene 50, 80 og 90% i forhold til dobbel hashing. Dette kan forklares ved at hashfunksjon nr. 2 til DoubleHash klassen har stor tidskompleksitet fordi den prøver å returnere et tall som er relativt primisk med størrelsen til tabellen. Men hvis tabellens størrelse er et primtall, så kommer ikke hashfunksjonen til å gå gjennom denne løkka, og dermed forsvinner en del av kompleksiteten (en del fordi GCD-metoden har fortsatt har kompleksitet). Dette kan vi faktisk se hvis vi velger at tabellstørrelsen er primtallet 10 000 019. Figuren på neste side viser dette.

Et bilde som inneholder tekst, skjermbilde, Font, nummer

Automatisk generert beskrivelse

Figur : Utskrift av tidsmåling hvor størrelsen på tabellene er primtallet 10 000 019

Hvis vi sammenlikner denne utskriften med forrige utskrift som hadde en tabellstørrelse på minsteverdien 10 000 000, ser vi at for alle tidsmålingene av dobbel hashing er kjappere, mens enkelte målinger fra lineær probing er blitt tregere.