INF-1101: DATASTRUKTURER OG ALGORITMER

INDEX ABSTRAKT DATATYPE

1 INTRODUKSJON

Denne rapporten handler om implementering av abstrakt datatype indeks, som brukes til å indeksere tekstdokumenter, og tilbyr autofullføring av søkeord. Dette gir User Interface ( UI ) muligheten til å søke etter et spesifikke ord eller en setning i tekstdokumentene.

2 TEKNIKK & BAKGRUNN

Et bilde som inneholder tekst, skjermbilde, Font

Automatisk generert beskrivelseDette programmet består av to prosesser. Den første prosessen har ansvaret for filhåndtering og separerer hvert enkelte ord fra tekstdokumenter, slik at ordene blir lagret i rekkefølge i en liste. Den andre prosessen brukes til å vises resultatene i UI.

UI kjøres i en løkke, hvor autofullføring skjer før søkeprosessen. Under søkeprosessen så blir informasjon om søketreff hentet via en funksjon som iterere gjennom tekstfilene, slik at UI kan bruke til å markere alle treff som ble funnet i tekstdokumenter.

Hvis det finnes treff i flere dokumenter, så blir resultatene vises et om gangen til det ikke finnes flere, deretter vil UI vise neste treff som finnes i andre fil. Dersom det finnes ingen treff så kan man søke på nytt.

3 DESIGN

3.1 Indeks Struktur

Data strukturen for indeks skal innehold navnet for dokumentet slik at data ikke bli lagret i samme indeks strukturen.

Et bilde som inneholder tekst, skjermbilde, Font, nummer

Automatisk generert beskrivelseLagring av data for tekstdokumenter, skjer via en funksjon som bruke en referanse til et tekstdokument. For å lagre data for flere tekstdokumenter så må indeks strukturen ha et medlem som kan lenke til andre indeks strukturer der tekstdokuments data er lagret.

For å lagre tekstdokument innholdet i indeksen, er det behov for en dynamisk liste i strukturen, hvor det skal brukes både for å sammenligne søkeord under søkeprosessen ,og vise innholdet som er i tekstdokumentet der søkeordet ble funnet.

UI treng også å vite antall ord som er i tekstdokumentet, slik at den kan skrive hvert enkelte ord i skjermen, og markere søketreff som eksisterer i tekstdokument med farge. Dermed er størrelsen også inkludert i strukturen, som oppdateres for hver gang et ord ble lagt til dynamisk liste.

Autofullføringen bruke *Trie Tree* for å finne det korteste ordet med nullterminal som innehold minst tre bokstaver der søkeordet ikke ble ferdig skrevet. Lagring av ord i *Trie Tree* kan håndtering under søkeprosessen, dermed er *Trie Tree* også inkludert i indeks strukturen.

3.2 Search Result Struktur

Et bilde som inneholder tekst, skjermbilde, Font, nummer

Automatisk generert beskrivelseSøke resultatene blir håndtert i en funksjon som har en referanser for indeksstrukturen, og søkeord i sin parameter.

Sammenlign av ord skjer under søkeprosessen, som lagre lokasjon og antall ord som ble funnet i en data struktur kalt *search\_hits\_t*.

Informasjonen som er lagret i *search\_hits\_t* skal senere blir brukt i UI, hvor den bruke lokasjon og antall av treff ordet til å definere hvilke ord som skal markeres i tekstdokumentet.

Når neste ordet skal markeres så hente den neste *search\_hits\_t* , dermed representerer *search\_hits\_t* for hvert enkelte treff, og er lagret i en *Linked-List* slik at programmet kan hente treff resultatene og klargjøre neste søketreff. Dermed er *Linked-List* inkludert i *search\_result* strukturen.

Tekstdokument innholdet er også lagret i indeksstrukturen under søkeprosessen, slik at UI kan bruke innholdet fra tekstdokumentet til å vise tekt på skjermen der søketreff er funnet. Dermed er indeksstrukturen inkludert i *search\_result,* for å ha oversikt på alle søketreff i ulike fil.

4 IMPLEMENTASJON

4.1 Pre-Koden Endring

Et bilde som inneholder skjermbilde, diagram

Automatisk generert beskrivelseEndringen i pre-koden befinnes i *trie\_insert()* funksjonen, slik at lengden av ordet som skal lagres i treet er definert for å ha oversikt på dybden av treet som finnes under hvert eneste bokstavene.

Denne endringen bidrar til effektiviteten for å finne det kortest ordet, der verdiene for hver bokstav befinnes seg i det samme nivået, blir sammenlign. Isteden for iterere igjennom hele treet for å finne det korteste ordet,

For eksempel i noden V som har en indeks og representere E og en annen indeks for O. E indeksen har en node som innehold en i sin verdi, dette betyr at det er en node under E, dermed er det kortere vei til slutten av ordet med nullterminal enn O indeksen som har fem noder under seg.

4.2 Autocomplete

Et bilde som inneholder skjermbilde, tekst, kvadrat

Automatisk generert beskrivelseAutofullføring funksjonen får tilgang på *Trie Tree* gjennom en referanser for indeksstrukturen i sin parameter, samt en paramenter for input fra UI-et hvor søkeordet er ikke ferdig skrevet som skal bruke til iterasjon.

Under finnprosessen i *trie\_find,* blir input fra UI-et brukt til å gå gjennom nodene i *Trie Tree*, der enhver bokstavposisjon fra UI-en representere et nivå i *Trie Tree.*

Dersom input fra UI befinnes i treet, så sjekker programmet om det finnes flere noder som er under den siste bokstaven fra UI.

Hvis det er flere så skal programmet bruke en metode for å finne en node som har minst barn, til å velge en indeks i den værende nivå for iterasjon og finne det ordet som er korteste. Dersom det finnes ikke flere barn og den værende posisjon har en node som representere et ord med nullterminal så blir ordet returnert. Hvis endel av input fra UI ikke befinnes i treet så returnerer programmet ingenting.

Etter funksjonen har funnet en node med minst barn, så vil den returnere det første ordet den finn under den noden. Som er alfabetisk rekkefølge.

4.3 Søkeresultat

Et bilde som inneholder tekst, diagram, plan, skjermbilde

Automatisk generert beskrivelse

Søkeprosessen bruke enten resultatet fra Autocomplete eller søkeordet fra UI-et for iterasjon gjennom tekstdokumenteter som er lagret i indeksstrukturen.

Under søkeprosessen, sjekker programmet og sammenlign et ord om gangen. Resultatene for søkeordet blir lagret i en struktur kalt *search\_hits\_t* som inneholder informasjon om hvor ordet ble funnet.

Disse *search\_hits\_t* strukturene blir lagret i en *Linked-List*, slik at UI-et kan bruke søkeresultatene til å markere søketreff på skjerm i sortert rekkefølge den befinnes i tekstdokumenter. Innholdet for tekstdokumentet der søkeordet ble funnet, blir også lagret i *search\_result*.

Dersom indeks strukturen har flere tekstdokumenter som ikke har blitt iterert gjennom, så skal søke resultatene (hvis det finnes) blir lagret i en ny *search\_result* strukturen, Denne nye strukturen blir deretter lenket til den værende *search\_result* som inneholder søkeresultatene som ble funnet i den første tekstdokumenteten. Dette vil fungere som en *Single Linked-List*.

4.4 Søkeresultat iterasjon

Søkeresultatene kan inneholde data som representerer flere tekstdokumenter, for å unngå gjenbruk resultat på det samme dokument, så kan *search\_result* ha en variabel som holder kontroll på hvilket data er det som har blitt hentet av UI-et. Dette gjør det mulig for å hente søkeresultat for ethvert dokument.

Et bilde som inneholder skjermbilde, tekst, design

Automatisk generert beskrivelse

Når dokumentet innholdet skal hentes, så er inputen en referanser for *search\_result* hvor søkeresultatene er lagret. Under lesing av data strukturen så sjekker programmet om nåværende dokumentet har blitt lest.

Hvis dokumentet har ikke blitt lest for henting av dokumentet innholdet, så blir *processed counter* oppdatert til verdi 1 i *search\_result* før programmet returneres *String Array* som bære dokumentets innhold. Dersom dokumentet har blitt lest så gå til neste search*\_result* og gjenta prosessen.

Iterasjon for henting av antall ord som er i tekstdokumentet og søkeresultatene fungerer det samme måten. Bare at henting av søkeresultatene markeres som lest med verdi 3 etter at det siste resultatet er hentet.

Enhver henting av prosessen har sin egen verdi som beskrive ulike oppgaver :

|  |  |
| --- | --- |
| ID | Fullført Oppgave |
| 1 | Hentet av tekstdokuments innhold |
| 2 | Hentet av tekstdokuments innhold lengde |
| 3 | Hentet av søkeresultatene som er i tekstdokumentet |

5 DISKUSJON & RESULTAT

5.1 Ytelsesanalyse

| N Words | Time(µs) | Time/Word(µs) | Search Hit Time(µs) | Search Miss Time(µs) |
| --- | --- | --- | --- | --- |
| 100000 | 48882 | 0.488820 | 2921 | 1102 |
| 200000 | 95321 | 0.476605 | 5165 | 2330 |
| 400000 | 174706 | 0.436765 | 9785 | 4834 |
| 800000 | 451220 | 0.564025 | 21276 | 9517 |
| 1600000 | 720794 | 0.450496 | 41588 | 18501 |
| 3200000 | 1406772 | 0.439616 | 243239 | 36355 |
| 6400000 | 2879825 | 0.449973 | 172992 | 74355 |
| 12800000 | 6110660 | 0.477395 | 994573 | 146421 |
| 25600000 | 14191109 | 0.554340 | 2162088 | 291924 |
| 51200000 | 26576909 | 0.519080 | 2837075 | 1636506 |
| SUM | 52656198 |  | 6490702 | 2221845 |

Data fra en *benchmark* resultanten viser tiden for programmet brukte til å fullføre ulike arbeid som er også relatert med de valgte data strukturer til oppgavens løsninger.

5.1.1 N Words & Time

Forholde mellom antall ord som blir lagret i indeks og total tid den brukte, har sterkt sammenheng. Hvor tiden dobles for hver gang antall av ord er dobles.

Tidskompleksitet for å bygge en indeks med **n** ord er **O( n )** . Dette skylles på måten data blir håndtert for både lesing og lagring, der antall av ord blir lest og lagret en og en om gangen.

5.1.2 N Words & Time/Word

Tiden for å lagre et ord i forhold til antall ord som skal lagres, økes gradvis. Dette skylles for bruk av *for-løkke* der hvert enkelt ord som skal lagres i indeks får sin egen plass i liste.

5.1.3 N Words & Search Time

Tidskompleksitet for søketiden i den verste tilfelle er **O( n )** , hvor programmet går gjennom liste som inneholder ordene som skal finnes, blir sjekket en og en omgang og dette skylles for bruk av dynamiske liste for lagring.

Tidskompleksitet for flere dokumenter er : , hvor **t** er summen av antall ord i dokumenter.

Grunnen for søketiden for søkeord som ikke eksisterer brukte halvparten mindre i tid er fordi programmet iterere gjennom innholdet uten å lagre søkeresultater. Dermed hoppet programmet over lagring prosessen under søkeprosessen. I tillegg tar det lengere tid for allokering av *search\_hits\_t* , lagring av data og legges til *Linked List* hvor det faktisk finnes treff.

5.2 Forbedringer

Et bilde som inneholder tekst, Font, line, skjermbilde

Automatisk generert beskrivelse

Analyse av *Flame Graph*[ 2 ] viser at lagring prosessen brukte mest av resursen og tid. Det er muligheter for optimalisering av programmet uten å måtte endre på data strukturen som er brukt til lagring, ved gå gjennom profilen kan vi identifisere prosesser som er unødvendig å ha i lagringens prosess.

5.2.1 Optimal lagring prosessen

Gjennomgang av implementering for lagring prosessen avslørte at programmet brukte lengere tid for allokering av minne blokk som skal brukes å produsere nullterminalt ord. Denne prosessen for å legge til nullterminal for hvert ord, involverte tre steg : (1)Allokering, (2)Kopiere og (3)Legg til et nytt element på slutten av ordet, før den ble lagt til treet.

Etter å ha endret denne prosessen til å duplisere og legg til nullterinal på slutten av ordet ved bruk av C-biblioteket funksjonen kalt *strdup* , så ble antall *Samples* redusert med 5 381. Med denne endringen har vi optimalt lagring prosessen med 10% raskere, dermed brukte den også mindre minne.

Et bilde som inneholder tekst, skjermbilde, Font, line

Automatisk generert beskrivelseEt bilde som inneholder tekst, skjermbilde, Font, line

Automatisk generert beskrivelse

Før endring [ 2 ] Etter endring[ 3 ]

5.3 Optimalt Søketiden

5.3.1 Versjon 2 Ideen[ 4 ]

Forbedring av søketiden, kan vi vurdere å erstatte dynamiske list med *Hash Map,* for å unngå iterasjon. Ved å gjøre denne endringen vil vi oppnå søketiden på **O(1)**, dette er optimalt for søkeapplikasjoner hvor lavtid er prioritert. Inkludering av *Hash Map* i indeksstrukturen og implementering under lagringsprosess kan være en god løsning. Dette vil også øker forbruk av ressurs, samt reduseres søketid som er en god trade-off.

Ved å implementere *Hash Map* , kan vi lagre søkeresultater som inkluder lokasjon av ord og antall ord i *search\_hits\_t.* Programmet vil konvertere ord via *Hash-Funksjonen* til en *Map-Key* som representerer et enkelt ord og *Map-Key* vil peker på en *Linked-List* hvor det inneholder søkeresultatene i rekkefølge.

Når dette skal brukes, må programmet konvertere søkeordet via *Hash-Funksjonen* for å sjekke om søkeordet eksisterer i den værende indeks i *Hash Map*. Hvis det finnes så vil programmet returnere *Linked-listen* som inneholder søke resultatene som har blitt lagret under lagring prosess. Dersom det ikke finnes så vil Big-O notasjon være likt om det fantes : **O(1)**

Et bilde som inneholder skjermbilde, tekst

Automatisk generert beskrivelse

5.3.2 Implementering Versjon 2

| N Words | Time(µs) | Time/Word(µs) | Search Hit Time(µs) | Search Miss Time(µs) |
| --- | --- | --- | --- | --- |
| 100000 | 109035 | 1.090350 | 1 | 4 |
| 200000 | 148156 | 0.740780 | 2 | 5 |
| 400000 | 353197 | 0.882992 | 2 | 5 |
| 800000 | 846495 | 1.058119 | 3 | 7 |
| 1600000 | 1532361 | 0.957726 | 2 | 15 |
| 3200000 | 2382315 | 0.744473 | 6 | 8 |
| 6400000 | 5419712 | 0.846830 | 1 | 6 |
| 12800000 | 8781327 | 0.686041 | 2 | 4 |
| 25600000 | 19263469 | 0.752479 | 1 | 5 |
| 51200000 | 45196680 | 0.882748 | 8 | 4 |
| SUM | 84032747 |  | 28 | 63 |

Versjon 2 har redusert betydelig av søketiden, akkurat som forventningen. Det som har blitt gjort var å endre *Hash Map* i pre-koden for å inkludere *Linked-list* akkurat som det ble nevn i seksjonen (5.3.1). Denne endringen har økt tidskompleksitet for lagring av data og er på grunn av hvert eneste ord og *search\_hits\_t*  data strukturen ble lagret i *Hash Map* , der ‘’Versjon 1’’ ikke hadde denne prosessen.

5.3.2.1 Fler-ord Søking

Implementering av Versjon 2 ga programmet en mulighet for å inkludere en ny funksjon for flere ord søking som Versjon 1 ikke hadde.

Under lagring prosessen der lokasjon av hvert eneste ord ble lagret i en *search\_hit\_t* og ble deretter lagt i en *Linked List* som finnes i *Hash Map* . Gir programmet muligheten for å hente ordets posisjon med tidskompleksitet på **O(1)**.

Et bilde som inneholder tekst, skjermbilde, Font, design

Automatisk generert beskrivelseNår en setning skal søkes, blir setningen delt opp i ordviss. Slik at programmet kan hente posisjoner som representere for hvert enkelte ord til å sammenlign .

Når posisjonene skal sammenligne så tar sammenlign funksjonen to paramenter, en som representere *search\_result\_t* for første ordet i setningen, og den andre parameteren er *search\_result\_t* som representere neste ordet.

Etter at sammenlign prosessen er gjort, vil den returnere en ny *search\_result\_t* som kan bli brukt til å sammenlign med neste ord. Med denne logikk, har programmet muligheter for å søke etter en setning med ubegrenset ord.

Lagring av ordets lokasjon i *Hash Map* kreves en del av minne bruk, der data skal bli lagret i minnet. Dermed må programmet utnytte dataene som har blitt lagret i minnet ved å implementere dette.

5.3.2.2 Sammenlign Algoritme

Et bilde som inneholder tekst, skjermbilde, design

Automatisk generert beskrivelseUnder sammenlign prosessen, så blir *Search\_hits\_t* pakket ut fra *Hash Map*, en og en om gangen. Slik at posisjon indeks kan bruke til å sammenligne.

Denne algoritmen involvere tre prosess, utpakket, sammenlign og lage en ny Linked List som bære *Search\_hits\_t.* Slik at det kan brukes i søkeresultat iterasjon ( seksjon 4.4 ).

I sammenlign prosessen så blir sub ordets posisjon verdien trekkes ned med dets posisjon den befinnes i søkesettingen. Dette er en effektiv metode for å definere når hoved ordets posisjon skal bli lagret i en ny *Search\_hits\_t.*

I denne prosessen så bestemmer sub-ordets posisjon for hvilke posisjonsverdier i hovedordet som skal bli lagret. Slik at UI kan bruke det til å markere setningen ved å bruke hoved ordet posisjonsverdi som en start punkt.

5.4 Resultat

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| VERSJON | Total tid for lagring prosess | Total tid treff søk | Total tid ingen treff søk | | |
| 1 | 55 sek | 6.5 sek | 2 sek | | |
| 2 | 33 sek | 0.0 sek | 0.0 sek | | |
| VERSJON | COMMAND | %CPU | MEM | FAULTS |
| 1 | bench\_index | 85.3 | 8085M+ | 3831231+ |
| 2 | bench\_index | 96.8 | 12G+ | 4600126+ |

Endringen i Versjon 2 førte til optimalt for lagring prosessen hvor duplikasjon av ord er ikke lengere med, dette er en booster for programmet hvor den brukte kun det samme minne området som er allokert under filbehandling.

CPU er også optimalt for lagring prosessen, der hvert enkelte ord blir ikke lagret i Trie Tree, siden Trie Tree har behov for CPU for iterasjon gjennom treet. Teknikken som ble brukt var å utnytte Hash Map for å definere om det værende ordet er allerede lagret. Oppå dette så har programmet spart mye av RAM ved å gjenbruke Trie Tree som en global datablokk, slik at det ikke blir lagret et nytt tree for hvert file.

Det som kunne ha gjort mer er å oppgradere Hash Map for å bruke en dynamiske list for å lagre søketreff. Dette vil være bedre enn Double Linked List hvor den brukte mer RAM for allokering referanser. Det er behov i dette prosjektet er en list hvor det ikke er behov for sletting av element, dermed vil en dynamiske list være mer effektiv for både minnet og CPU.

6 KONKLUSJON

Under implementering for filhåndteringen hvor iterasjon for søkeresultater skal lese et nytt dokument, har jeg brukt en de-referanser til å endre pekerens adresse for å peke på en ny fil og slette søkeresultatene og dokumentets innhold som er lagret på leste filen.

Dette gjordet at UI-et klarte ikke å hente dokumentets innhold og det som ble lært var dypere forståelsen i programspråket C hvor en minneblokk kan ha flere referanser og de får lov til å endre innholdet som er i minne blokken. Dette funnet førte til inkludering en variabel som holder på kontroll over leste filer slik at innholdet ikke blir slette og UI-et har tilgang på leste filer.

Fordelen med enkelt implementasjon bidrar til forståelse innenfor valg av data strukturer, og teknikker som ble brukt til å løse ulike problemer. Ulempen med dette kan være at programmet er ikke optimalt, men kan lett bli fullført etter at man har analysert profilen og finne en fungerende data struktur for ulike problemer, akkurat som det har blitt gjort i ‘’Versjon 2’’.

Endring i prekoden gir en stor mulighet for å bruke mine ideer til å løse ulike oppgaver, dersom jeg ikke kunne endre Hash Map til å inkludere en Linked List som bære referanser for søkeresultater så hadde jeg implementert en ny Hash Map som er effektiv kun for dette oppgaven, dette gjelder det samme for Trie Tree.

Dette prosjektet har gitt meg dypere forståelsen om hvordan CPU og RAM blir brukt av programmet, og ha kontroll over min implementasjon slik at den ikke bruke mer resurs enn behov. CPU kan bli redusert av å skrive mindre kode, mens RAM kan bli redusert ved å lagre minst mulig data som CPU kan hente og manipulere.

7 REFERANSER

[ 1 ] Tolkingen for tekniske fundament av programmet fra Flame Graph.

<https://flamegraph.com/share/6c124d68-f4a9-11ee-a542-36adf59c1176>

[ 2 ] Før endring tabell

<https://flamegraph.com/share/68108c71-f72e-11ee-a542-36adf59c1176>

[ 3 ] Etter endring tabell

<https://flamegraph.com/share/54d14458-f7c7-11ee-8204-e6b38c1ccd74>

[ 4 ] Hash Map illusjon og ideer

<https://en.wikipedia.org/wiki/Hash_table>