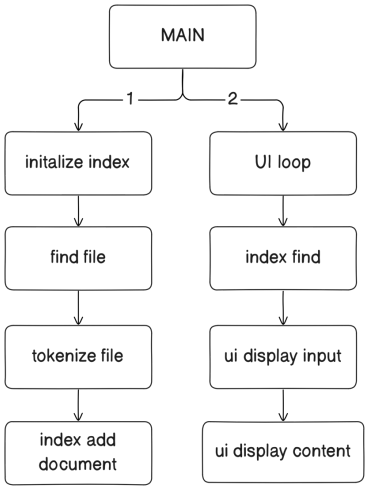
INF-1101: DATASTRUKTURER OG ALGORITMER

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 Unser Interface (UI) muligheten til å søke etter spesifikke ord i tekstdokumentene, hvor indeksen representerer ordets lokasjon i dokumentet.

2 TEKNIKK & BAKGRUNN [1]



Dette 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 søk prosessen så er informasjon hentet via en funksjon som iterere gjennom tekstfilene, der UI kan brukes til å markere alle ordene som ble funnet i tekstdokumenter.

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

3 DESIGN

3.1 Indeks Struktur

Data strukturen for indeks skal innehold navnet på dokumentet for å definere at indeks strukturen er ikke ledig for å lagre nye dokumenter.

```
-----------------------------------
|             index               |
-----------------------------------
|   documentName: char*           |
-----------------------------------
|   stringArray: char**           |
-----------------------------------
|   trieTree: trie_t*             |
-----------------------------------
|   next: index_t*                |
-----------------------------------
|   size: int                     |
-----------------------------------
```

Lagring av data for tekstdokumenter skjer via en funksjon som har en referanser for indeks strukturen i parameteren, den representerer hvert enkelte tekstdokument. Dermed må indeks strukturen ha et medlem som refererer til andre tekstdokumenter også.

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

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

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

3.2 Search Result Struktur

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

```
-----------------------------------
|          search_result          |
-----------------------------------
|   hitsArray: list_t*            |
-----------------------------------
|   index: index_t*               |
-----------------------------------
|   next: search_result_t*        |
-----------------------------------
|   accessedCounter: int          |
-----------------------------------
```

Sammenlign av ord skjer under søkeprosessen, hvor lokasjon og lengde av treff ordene blir lagret i en struktur kalt *search_hits_t* som representere hvert enkelte treff. Informasjonen som er lagret i *search_hits_t* skal senere blir brukt i UI, hvor den bruke lokasjon og lengde 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* or hvert enkelte treff, og lagres i en *Linked-List* slik at programmet kan fjerne treff resultatet etter brukt og ha lokasjon og lengde for neste treff søkeordet klar for bruk. Derfor er *Linked-List* inkludert i *search_result* strukturen.

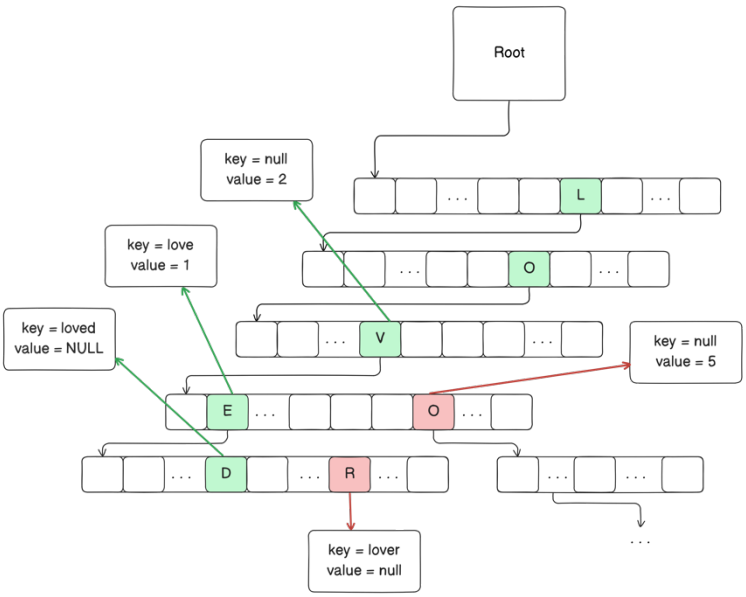
Tekstdokumentet innholdet er også lagret i indeksstrukturen under søkeprosessen, slik at UI-et kan bruke innholdet fra tekstdokumentet hvor treff ordene ble funnet til å vise på skjermen, dermed er indeksstrukturen inkludert i *search_result* .

Når treff ordet eksisterer i flere dokumenter, kan vi skille *search_result* for å representere kun et dokument per struktur, og heller ha en referanser til neste dokument, dette vil holde lagring av søk resultat i orden og lett tilgjengelig.

4 IMPLEMENTASJON

4.1 Pre-Koden Endring

Endringen i pre-koden er i *trie_insert()* funksjonen, der lengde av ordet som skal lagres i treet er definert. Dette gir programmet oversikt på dybden av treet som finnes under hvert eneste bokstavene som ikke er slutten av ordet.

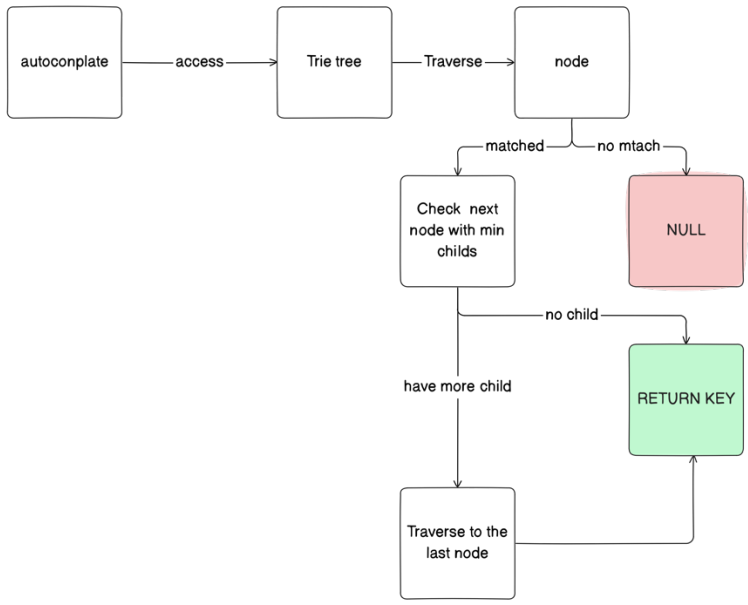


Denne endringen bidrar til effektiviteten ved å finne det ordet som er kortest, hvor verdien for hver bokstav befinnes seg på samme nivået, blir sammenlign. Isteden for iterere igjennom hele treet.

For eksempel i noden V som har en indeks som representere E og en annen indeks for O. Hvor E indeksen har en node som innehold 1 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.

4.2 Autocomplete

Autofullføring 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, og dette brukes til iterasjon.



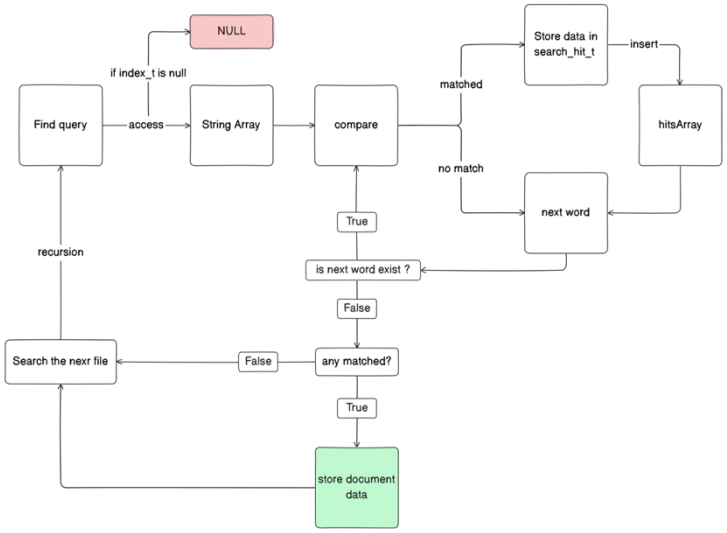
Under finnprosessen i *trie_find*, blir input fra UI-et brukt til å gå gjennom nodene i *Trie Tree*, hvor enhver bokstav fra UI-en input representere et nivå i *Trie Tree*.

Dersom input fra UI-et befinnes i treet i rekkefølge så sjekker programmet om det finnes flere noder som er under den noden som representere den siste bokstaven fra UI-et. Hvis det er flere så skal programmet bruke en metode for å finne en node som har minst barn, slik at den kan bruke denne informasjonen til å velge en node for iterasjon og finne det ordet som er korteste.

Dersom det finnes ikke flere barn og dette noden representere et ord med nullterminal så blir ordet returnert. Hvis endel av input fra UI-et ikke befinnes i treet så returnerer programmet ingenting.

4.3 Søkeresultat

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



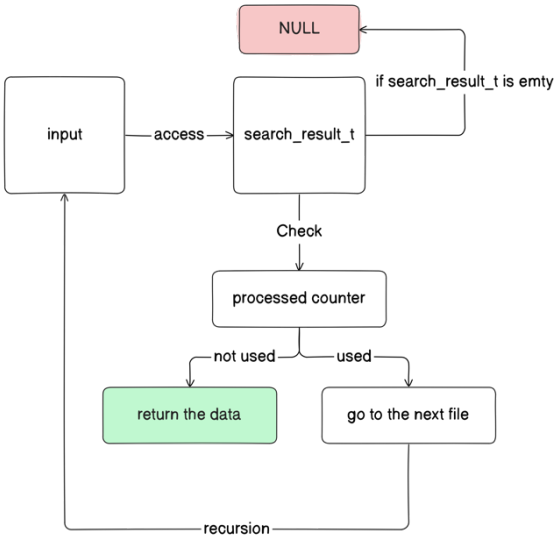
Under søkeprosessen, sjekker programmet og sammenlign et ord om gangen. Resultatene for søkeordet blir lagret i en struktur kalt *search_hits_t* som inneholder informasjon om hvor ordet ble funnet og lengden av ordet.

Disse *search_hits_t* strukturene blir lagret i en *Linked-List*, slik at UI-et kan bruke søkeresultatene til å markere søkeordene på skjerm i sortert rekkefølge. 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 nåværende *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.



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	27726	0.277260	2267	1030
200000	61899	0.309495	5856	2006
400000	163580	0.408950	8887	4681
800000	252430	0.315538	16023	8455
1600000	516481	0.322801	74200	17701
3200000	1039747	0.324921	61003	38888
6400000	2296531	0.358833	132487	67963
12800000	4922520	0.384572	298842	137532
25600000	14073442	0.549744	881552	282883
51200000	29919955	0.584374	2728076	549530

Data fra en *benchmark* resultanten viser tidskompleksitet for hvor lang tid programmet bruker for å fullføre ulike arbeid som er også relatert med de valgte data strukturer til oppgavens løsning.

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.

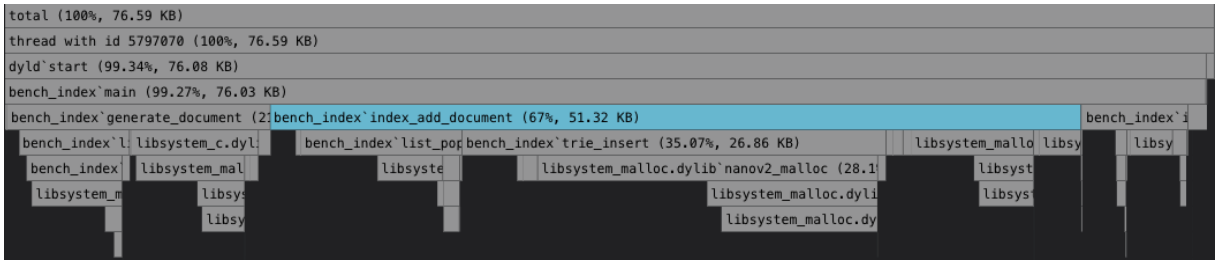
5.1.3 N Words & Search Time

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 : **O(t)**, hvor **t** er summen av antall ord i dokumenter.

Grunnen for at det brukte halvparten mindre i tid for å finne et ord som ikke eksisterer 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



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.

`bench_index`index_add_document`

% of RAM:	67%
RAM:	51.32 KB
Samples:	52,547

Før endring [2]

`bench_index`index_add_document`

% of RAM:	67.78%
RAM:	46.06 KB
Samples:	47,166

Etter endring[3]

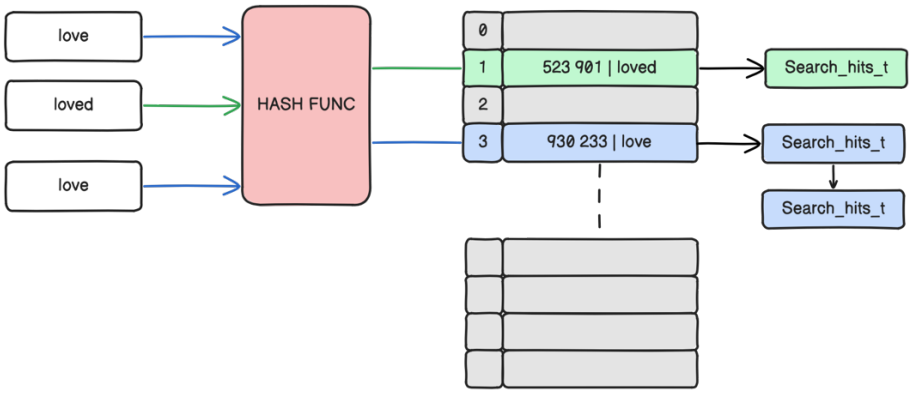
5.3 Optimalt Søketiden

5.3.1 Ideer[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 lengde 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)



5.3.2 Resultat

N Words	Time(µs)	Time/Word(µs)	Search Hit Time(µs)	Search Miss Time(µs)
100000	42203	0.42203	0	4
200000	91223	0.456115	0	4
400000	170045	0.425113	1	4
800000	490639	0.613299	1	4
1600000	803354	0.502096	4	1
3200000	1641985	0.51312	0	5
6400000	3339325	0.52177	1	4
12800000	7230927	0.564916	0	3
25600000	18836175	0.735788	5	4
51200000	53879435	1.052333	6	5

Implementering av ‘’Versjon 2’’ har redusert betydelig av søketiden, akkurat som forventningen. Det som har blitt gjort var å endre *Hash Map* i pre-koden hvor *Linked-list* blir inkludert akkurat som det ble nevn i ideen seksjonen (5.3.1). Denne endringen har økt tidskompleksitet for lagring av data og er på grunn av hvert eneste ord ble lagret i *Hash Map* hvor ‘’Versjon 1’’ ikke hadde.

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 bedre data struktur som kan redusere søketiden, akkurat som det har blitt gjort i ''Versjon 2''.

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