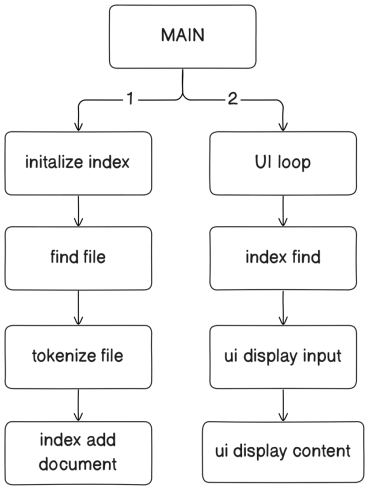
INF-1101: DATASTRUKTURER OG ALGORITMER

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 [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 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.

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

Lagring 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

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

Søke 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*.

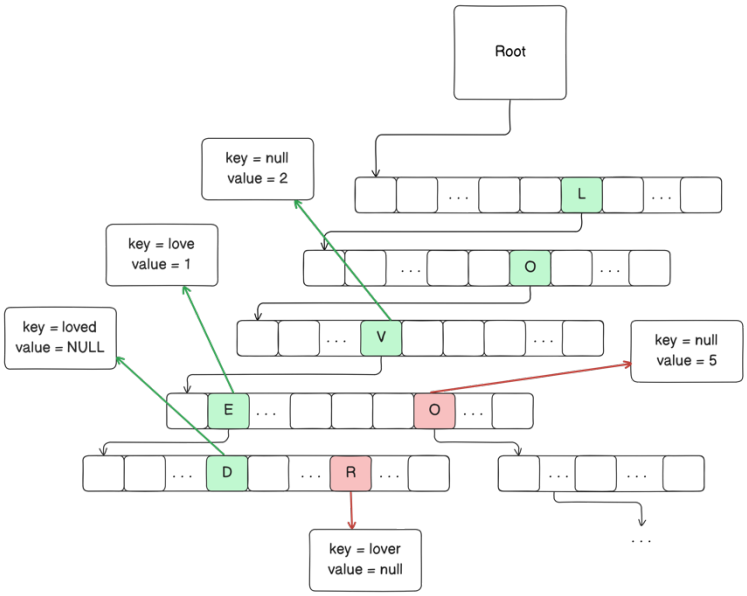
Informasjonen som er lagret i *search_hits_t* skal senere blir brukt i UI, hvor den bruke lokasjon og antall av treff ordet til å definere hvilke ord som skal markeres i tekstdokumentet.

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

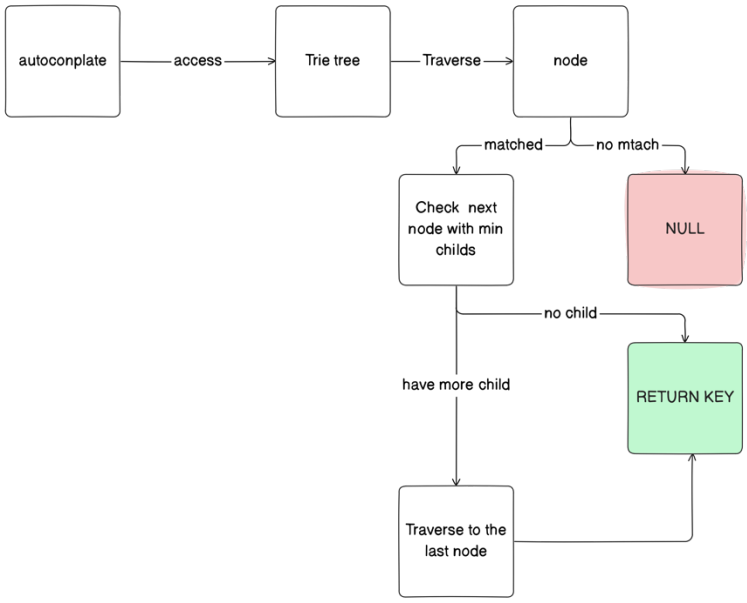


Endringen i pre-koden befinnes i *trie_insert()* funksjonen, slik at lengden av ordet som skal lagres i treet er definert for å ha oversikt på dybden av treet som finnes under hvert eneste bokstavene.

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



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 som skal bruke til iterasjon.

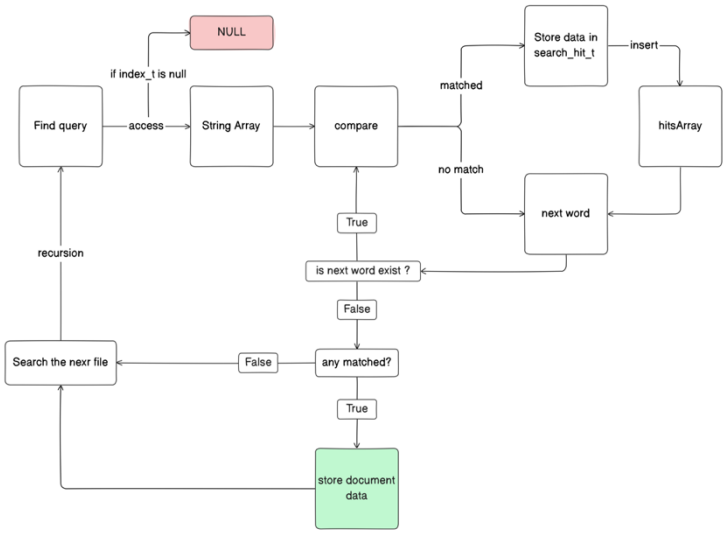
Under finnprosessen i *trie_find*, blir input fra UI-et brukt til å gå gjennom nodene i *Trie Tree*, der enhver bokstavposisjon fra UI-en representere et nivå i *Trie Tree*.

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



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

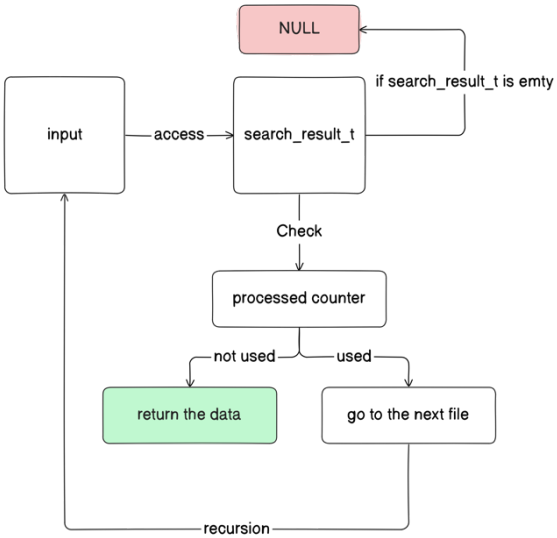
Under søkeprosessen, sjekker programmet og sammenlign et ord om gangen. Resultatene for søkeordet blir lagret i en struktur kalt *search_hits_t* som inneholder informasjon om hvor ordet ble funnet.

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.



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	79317	0.793170	4966	1845
200000	98713	0.493565	4932	2469
400000	195449	0.488623	10743	4498
800000	430172	0.537715	69497	10828
1600000	712957	0.445598	39517	18194
3200000	1429920	0.446850	82712	36880
6400000	3013105	0.470798	158254	73538
12800000	7380012	0.576563	1224985	146755
25600000	15216633	0.594400	2594642	436673
51200000	29289471	0.572060	3740515	670568
SUM	57845749		7930763	1402248

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.

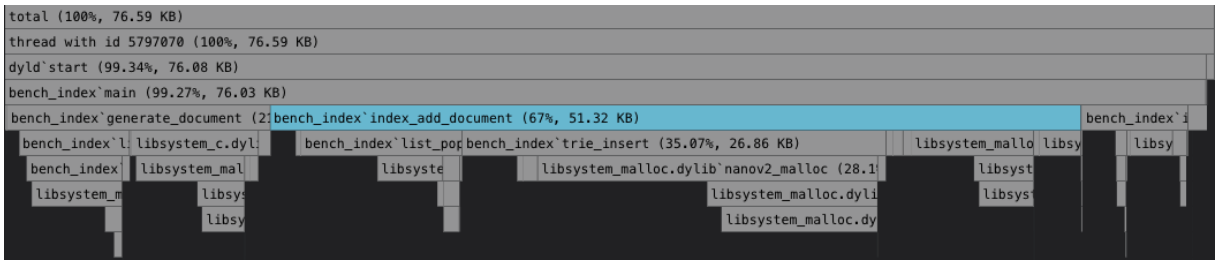
5.1.3 N Words & Search Time

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 : **O(t)**, 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 tref

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]

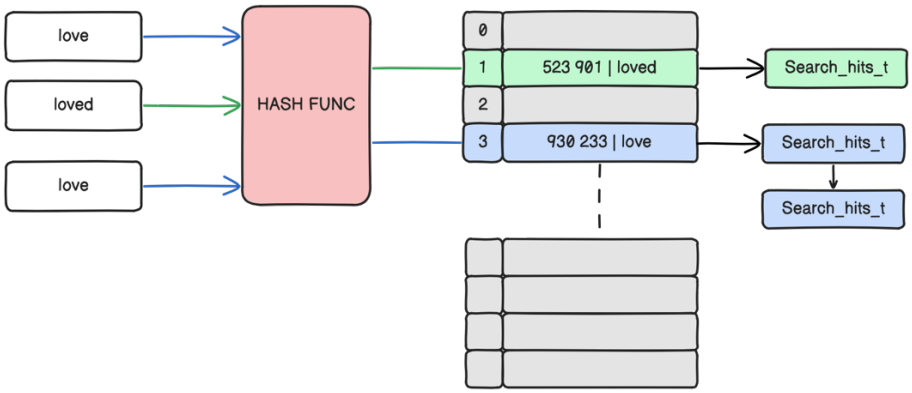
5.3 Optimalt Søketiden

5.3.1 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)**



5.3.2 Implementering Versjon 2

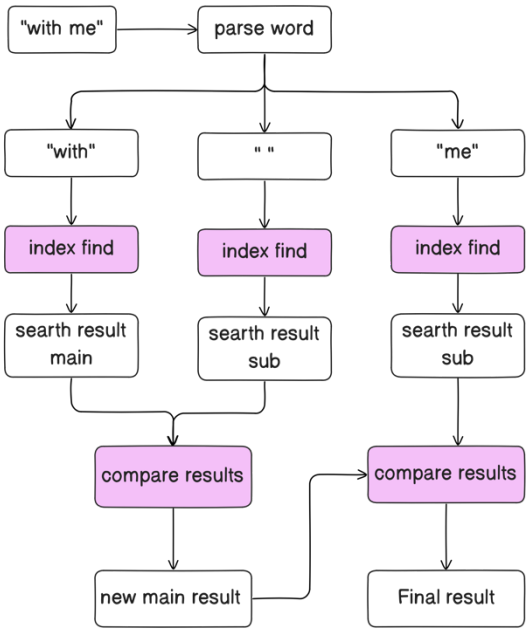
N Words	Time(µs)	Time/Word(µs)	Search Hit Time(µs)	Search Miss Time(µs)
100000	70928	0.709280	8	2
200000	61108	0.305540	2	1
400000	220069	0.550172	3	2
800000	250747	0.313434	3	1
1600000	473891	0.296182	3	1
3200000	949821	0.296819	3	2
6400000	1954358	0.305368	2	2
12800000	3799570	0.296841	3	2
25600000	8498803	0.331984	3	1
51200000	22154903	0.432713	8	1
SUM	38434198	0	38	15

Versjon 2 har redusert betydelig mye 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 forbedret tiden for å lagre av data og er på grunn av redusering av minne duplikasjon som bruker mer tid for allokering og tilgang. Siden Versjon 2 har flere operasjon så vil den bruke mer CPU, men mindre minnet som er en bra trade off.

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)**.



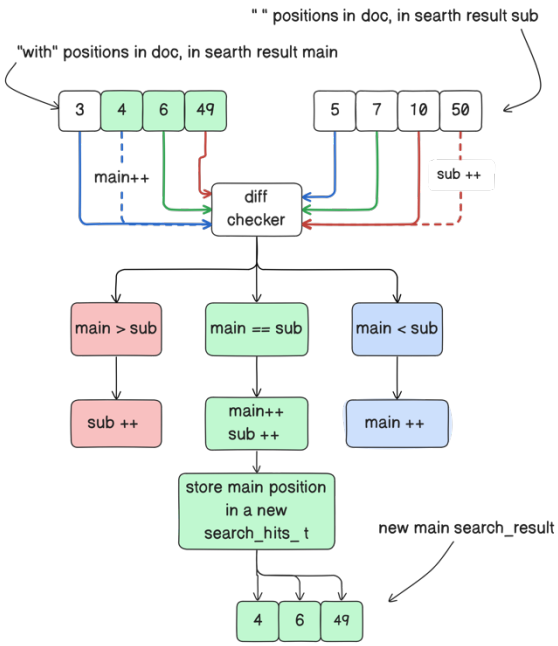
Når en setning skal søkes, blir setningen delt opp i ordviss. Slik at programmet kan hente posisjoner som representere for hvert enkelte ord til å sammenlign .

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



Under 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

Profil type	bench_index (PID 433)	bench_index2 (PID 915)	(PID 433)-(PID 915)
TIME SEC	57,845749	38,434198	-28,744509
#WQ	0	0	0
#POR	12	12	0
MEM	8892M+	7532M	-1360
PURG	0B	0B	
CMPRS	6149M+	5072M+	-1077

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 bruke CPU for iterasjon gjennom treet. Teknikken som ble brukt var utnytting av Hash Map for å unngå lagring av ord som er allerede lagret og gjenbruk Trie Tree som en global datablokk, slik at det ikke blir lagret et nytt treet 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 siden den brukte mer minne for allokering to referanser og en node for hvert element. Det som programmet har behov for, er en liste hvor det ikke trengs en sletting operator, 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”.

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. Ved å begrense forbruk av minnet, kan CPU håndtere og manipulere data mer effektivt, dette vil også forbedre kjøretiden.

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