Søgemaskineprojekt

18. december 2023

af Thomas Halkier Nicolajsen
s146711

Vejledere:
Inge Li Gørtz
Philip Bille

Beskrivelse af
Stephen Alstrup
Theis Rauhe

DTU Compute
Institut for Matematik og Computer Science

DTU Diplom
Center for Diplomingeniøruddannelse

Git Repo: https://github.com/tommydrengen/SearchEngineProject
GUI: https://github.com/tommydrengen/SearchEngineGUI

Indhold

1 Indledning

Søgemaskineprojektets formål er at undersøge hvordan man kan anvende datastrukturer som hægtede lister og hashtabeller, til at udvikle en skalerbar søgemaskine, som understøtter hurtige søgninger. Testdataen er et øjebliksbillede af Wikipedia fra 2010, men søgemskinen kan køres med andre datafiler af simplet tekstformat. Ønsket er at understøtte søgninger af formen: *Hvilke dokumenter har søgeord X*. Datafilerne er præfiks af samme datafil afkortet forskellige steder. De forskellige filstørrelser bruges til at undersøge hvordan søgemaskinernes performance ændres i takt med at filstørrelsen ændres. Performance måles som tidsforbrug for at hhv. søge og bygge, og pladsforbruget for at bygge datastrukturen. Søgetider undersøges også ved (separat) at variere ordlængde og ordplacering for en given filstørrele. Den fulde ukomprimerede fil er på 6.15 GB. Den næststørste er et præfiks på 800 MB, den tredjestørste er halvt så stor på 400 MB, og sådan fortsætter det til 1 MB og 100 KB.

Opgavens basale dels spørgsmål 1-4 er løst i hhv. Index1-4: Index1 er løsning til basal del 1 Index2 til basal del 2 osv.. Index5-8 er avancerede løsninger.

2 Navnekonvention

Navnekonvention for variabler vil være som følger:

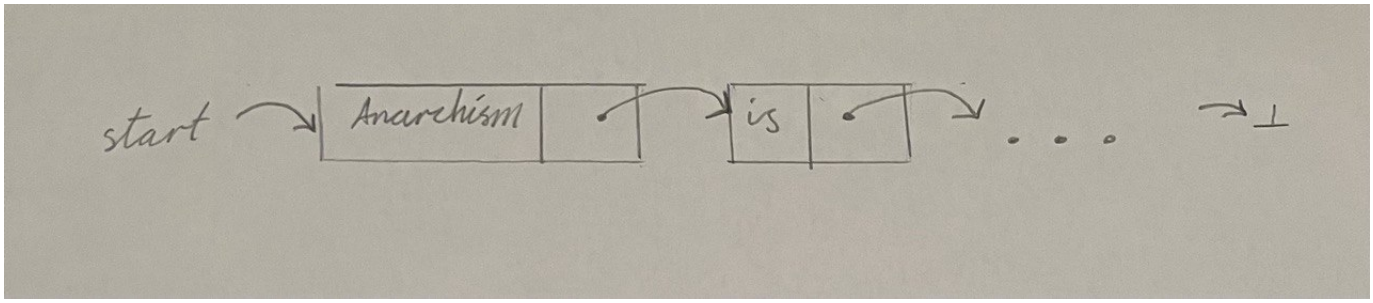
- n: antal ord
- d: antal dokumenter
- l: ordets længde
- x: ordets placering
- u: antal unikke ord
- H: antal indgange /Rows (længden af hashtabellen)
- T: Tid
- P: Plads
- Referencer i koden: start, current, tmp, startDistinct, currentDistinct, tmpDistinct, startReturnItem, currentReturnItem, tmpReturnItem

hvor $u < n$ og $d < n$
Kompleksiteterne vil udtykkes i n og d.

3 Beskrivelse og analyse af de anvendte datastrukturer

3.1 Hægtet liste

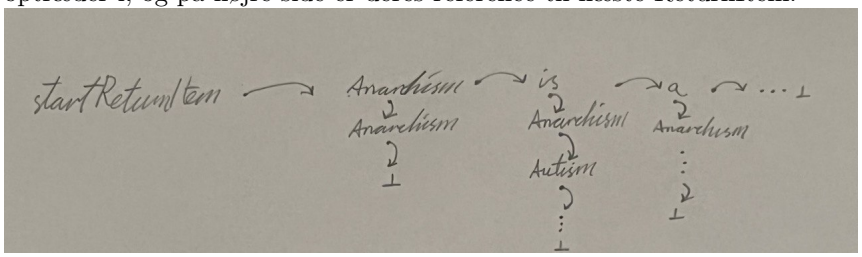
I den basale del bruges en hægtet liste til de første to delopgaver. Til det er defineret et *WikiItem* bestående af en String, og en reference til et næste af samme type.



Figur 1: Hægtet liste

3.2 ReturnItem

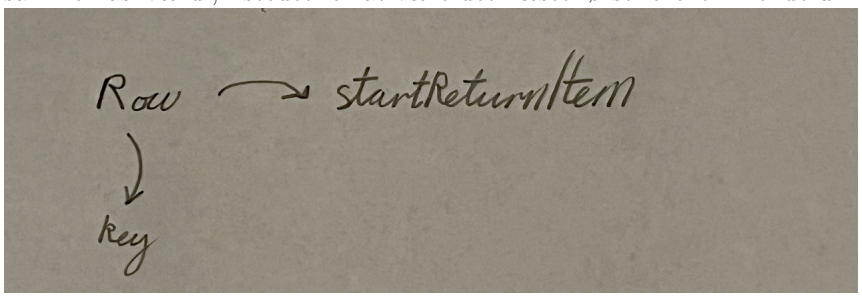
I tredje delopgave er *ReturnItem*-datastrukturen defineret, der er en søgning på et unikt ord, med en *String*, en hægtet liste, og en reference til et objekt af sin egen type.
Her er illustreret ReturnItems: Øverst er deres unikke ord, nedad er en reference til dokumentlisten ordet optræder i, og på højre side er deres reference til næste ReturnItem.



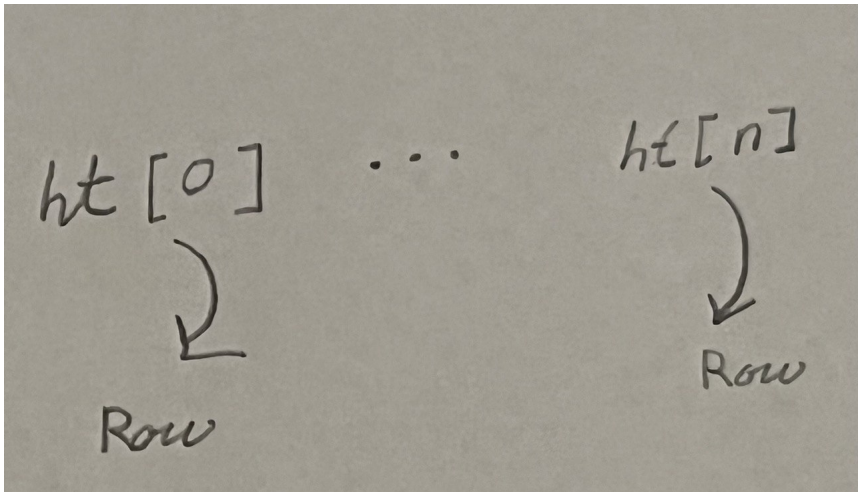
Figur 2: Unikke ord på vandret, og tilhørende dokumentlister på lodret.

3.3 Hashtabel

I fjerde delopgave er *Row* defineret som et *key-value-pair*. Fra den fjerde basale løsning (Index4) og frem holdes unikke ord med deres tilhørende dokumentliste (*ReturnItems*) i et array af *Row*. Hvert startReturnItem er en reference til et objekt med et unikt ord, en hægtet liste af dokumenter med det unike ord, og en reference til næste ReturnItem (med et andet unikt ord). Forskelligt fra forrige, tredje basale løsning (Index3) er, at i hashtabellen har et ReturnItems næste ReturnItems unikke ord *searchstr* samme hashværdi, i stedet for at være det næste først forekommende unikke ord.



Figur 3: Row er et key-value-pair



Figur 4: Hashtabel: Arrayet af *Rows* udgør hashtabellen.

4 Basal Del

4.1 Index1

I første basale løsning kaldet Index1 er et element af en hægtet liste kaldet WikiItem defineret. Index1 læser inputfilen, og bygger en hægtet liste af ordene i filen. Referencen til første element i den hægtede liste hedder start. Referencerne *current, tmp, current.next, tmp.next* bruges til at bygge og søge. Når filen er læst tager Index1 input fra brugeren som en søgestreng. Denne søgestreng sammenligner Index1 med alle strengværdierne i dens hægtede liste. Det gøres i en uendelig løkke, hvor søgemaskinen skiftevis er i en tilstand af at vente på input fra brugeren og at søge på brugerens ord. Det sker ved at gemme brugerinput i en String *searchstr*, og sætte *current* til at referere til det samme som *start*, og i en løkke sammenligne *searchstr* med *current*, sætte *current* til *current.next*, og printe *searchstr* exists, hvis de to Strings er ens . Løkken stopper når *current.next* er null. Hvis brugeren skriver *exit*, brydes løkken og progammet afsluttes.
Index1 bruger en hægtet liste. Ved opstart læser Index1 inputfilen et ord ad gangen. Så længe der er flere ord i inputfilen, læser den et nyt ord ind i en String og bygger et *WikiItem* af ordet, og sætter det som reference til næste WikiItem for det nuværende WikiItem, indtil der ikke er flere ord i filen. Således er start en reference til første WikiItem, og begyndelsen af den hægtede liste. Sidste element har en null reference som textitnext.

4.1.1 Teoretisk kompleksitetsanalyse: Tid

- Preprocessing

Det nye ord fra stdin lægges i et nyt WikiItem, som det nuværendes næste på konstant tid $O(1)$. Det sker for alle ordene i filen på $O(1 * n) = O(n)$
hvor n er antallet af ord i inputfilen.

- Søgning

Søgeordet sammenlignes med hvert ord i den hægtede liste *start* refererer til. I værste fald sammenlignes søgeordet med alle elementer, som tager lineær tid, og variablen *exists* ændres på konstant tid. Dette resulterer i et lineært tidsforbrug: $\mathrm{O}(n + 1) = \mathrm{O}(n)$.

4.1.2 Teoretisk kompleksitetsanalyse: Plads

Index1 har et WikiItem per ord, og derfor et lineæet pladsforbrug $\mathrm{O}(n)$.

4.2 Index2

Index2 er en udvidelse af Index1, som i søgemetoden læser dokumentnavnene ved at læse første dokumentnavn fra start til første punktum, og resten af dokumentnavnene fra
—END.OF.DOCUMENT— til næste punktum ".", og gemme i en tekststreng *documentName*. Desuden bygger den en String *returnString*, af dokumentnavne med søgeordet, vha. String concatenation.

Søgningen returnerer dokumentnavnene, søgeordet forekommer i. Returstrengen initialiseres som en tom String. Hver gang søgemetoden møder enden af et dokument, skriver den det aktuelle dokumentnavn til en String *documentName*. Samtidig tjekker metoden om *current* (det nuværende element i den hægtede liste) har søgeordet på sin Strengværdi. Hvis den har det, sættes *exists* til true, og antal forekomster *occcurences* tælles én op. Når metoden møder –END.OF.DOCUMENT—, og ordet er observeret i det nuværende dokument, tilføjes dokumentnavnet og antallet af forekomster som en linie til sidst i returstrengen. Inden næste dokumentnavn læses nulstilles antallet af forekomster, og exists sættes til *false*.

Ved søgning bygges dermed en returstreng ud fra brugerens søgeord, med en linie for hvert dokumentnavn ordet opstår i, og antallet af forekomster i dette dokument. Forskellen fra Index1 ligger i søgemetoden.

4.2.1 Teoretisk kompleksitetsanalyse: Tid

- Preprocessing

Ligesom i Index1 bygger konstruktøren i Index2 en hægtet liste af alle ord i inputfilen, som den vha. *current*-referencen lægger bagi WikiItem *start*, som er referencen til første element i den hægtede liste.

- Søgning

Forskellen fra Index1 er at den holder det aktuelle dokumentnavn, som ændres efter hvert —END.OF.DOCUMENT—. Index2 har derfor samme asymptotiske søgetid som Index1 på $O(n)$.

4.2.2 Teoretisk kompleksitetsanalyse: Plads

Plads: Hvert ord lægges i et WikiItem i den hægtede liste med pladsforbrug $O(n)$. Dokumentlisten er en String med maksimalt alle dokumenter $O(d)$. Pladsforbruget er $O(n+d)$=$O(n)$.

Søgning: I søgemetoden Index2 skriver dokumentnavne med søgeordet til en String, på konstant tid per succes. Derfor forbruger den stadig $O(n+d) = O(n)$ tid på at søge da $n < d$.

4.3 Index3

Index3 læser ligesom Index1 og Index2 datafilen ind, og bygger en hægtet liste af alle n ord i filen med en reference ved navn *start*. Desuden bygger den en hægtet liste af unikke ord *startDistinct*: hvert nyt ord tilføjes kun i *startDistinct* i tilfælde af en fejlagtig søgning (*search1()*) på ordet i den hægtede liste af unikke ord *startDistinct*. Det resulterer i en hægtet liste af unikke ord ved navn *startDistinct*.

Der er i Index3 defineret en datastruktur kaldet *ReturnItem*, som er en representation af en søgning på et unikt ord fra datafilen. Den har tre felter:

- En String *searchstr*: Det unikke ord.
- En WikiItem *startDoc*: Referencen til begyndelsen på den hægtede liste af dokumenter ordet forekommer i.
- En reference til næste *ReturnItem*.

4.3.1 Teoretisk kompleksitetsanalyse: Tid

- Preprocessing

Det nye ord fra stdin lægges i et nyt WikiItem*tmp*, som det nuværendes næste på konstant tid $O(1)$ ved hælp af referencen *current* (halen af listen). Det sker for alle n ord i filen på $O(n)$. (Uden referencen *current* ville det tage lineær tid.)

Defefter søges der på alle n ord, for at bygge listen af unikke ord startDistinct. Der er n ord, og en søgning på ét ord tager lineær tid n, så unikordlisten skabes på $O(n^2)$.

Derefter søges der på alle u unikke ord i listen af n ikke-unikke ord, (u søgninger lineære i n, hvor der tilføjes op til d dokumenter, hver på konstant tid): $O(u*1)$, og der tilføjes op til d dokumenter i det unikke ords dokumentliste for at skabe u søgninger på unikke ord (ReturnItems). Hvert nyt ReturnItem kobles på startReturnItem på konstant tid, fordi *currentReturnItem* er referencen til sidste ReturnItem under opbygningen *halen*. Sammenlagt er tidsforbruget for at bygge $O(n+n^2+u*d)$ hvor $u*d < n$

$O(n + n^2 + u * d) \leq O(n^2)$

$O(n^2)$

- Søgning

Søgeordet sammenlignes med hvert ReturnItems unikke ord, på lineær tid i u, og det eventuelle ReturnItem (eller false) returneres på konstant tid, og dokumentlisten printes på d tid. Dette resulterer i et lineært tidsforbrug i antallet af unikke ord og dokumenter: $O(u+d) = O(u+d) = O(n+d)$.
hvor $u \leq n$
$O(n+d)$

4.3.2 Teoretisk kompleksitetsanalyse: Plads

For hvert ord n i inputfilen har Index3 et WikiItem, plus et WikiItem per unikke ord u , og et ReturnItem per unikt ord u , som holder en hægtet liste af dokumentnavne indeholdende ordet. Således er pladsforbruget hhv lineært, lineært, kvadratisk i n. Sammelagt er pladsforbruget kvadratisk: $O(n + u + u * d) \leq O(2 * n + n * d) \leq O(n * d)$
$= O(n * d)$.

4.4 Index4

Ligesom i Index3 skabes en hægtet liste af ord, og en hægtet liste af unikke ord. Der skabes for hvert unikt ord et objekt med en reference til en hægtet liste af dokumentnavne hvor ordet optræder i. Objektet har også en reference til næste objekt af samme type, med et andet unikt ord, en anden reference til en dokumentliste og en næste af samme type.

Index4 er en udvidelse af Index3 hvor der er implementeret *chained hashing*: Index4 har defineret en datastruktur *Row*, som har et tal *key* mellem nul og ni, og et ReturnItem. Index4 har også en hashtabel som har et array af Row H og en hashfunktion, som fordeler ReturnItems (unikke ord med dokumentlister) jævnt i H. Hashfunktionen har værdier fra nul til længden af hashtabellen. *N.B.*: Man kunne med fordel have en hashtabel hvor antallet af indgange var et primtal.

Hver indgang i H er et *key-value-pair*: Da hvert ReturnItem har en unik søgestreng, kan denne bruges som *key*, og hashfunktionen tager derfor den unikke søgestreng som input. Da der er ti *Rows* i *Row*-arrayet, beregner hashfunktionen en hashværdi mellem nul og ni. Hashværdien er ens for for ReturnItems i samme Row, og bruges kun til at vælge Row'et, at indsætte eller søge i.

Hashfunktionen *hash(s)* er implementeret vha. generativ AI: ved at lade hashværdien starte på syv, og for hvert bogstav i ordet, gange med 31 som er et primtal, og lægge bogstavets ASCII-værdi til. Til sidst tages modulus (divisionsresten) af længden af hashtabellen, og får en hashværdi som er entydig for det unikke ord, og med en værdi jævnfordelt på værdierne mellem nul og længden af hashtabellen.[1]

4.4.1 Teoretisk kompleksitetsanalyse: Tid

- Preprocessing

Det nye ord fra stdin lægges i et nyt WikiItem, som det nuværendes næste på konstant tid O(1). Det sker for alle n ord i filen på O(n).

Derefter søges der på alle n ord, for at bygge listen af unikke ord startDistinct. Der er n ord, og en søgning på ét ord tager lineær tid n, så unikordlisten skabes på O(n^2).

Derefter søges der på alle u unikke ord, i listen af n ikke-unikke ord, (u lineære søgninger, hvor der tilføjes op til d dokumenter, hver på konstant tid), og der tilføjes op til d dokumenter i det unikke ords dokumentliste for at skabe u søgninger på unikke ord (ReturnItems). Hvert nyt ReturnItem hashes på konstant tid og kobles på et Row's startReturnItem på konstant tid, forudsat en reference *tmpReturnItem* har fat i halen af *Row'ets startReturnItem* på plads h[i] i det øjeblik: O(1).

(Det viser sig i midlertid ikke at være tilfældet. Der skulle både have været en *"startReturnItem-* og *"tailReturnItem"*-reference i hver hashtabellens indgange! Og et nyt ReturnItem indsættes på $O(u) = O(n)$ tid.

Sammenlagt skulle tidsforbruget tidsforbruget for at bygge være $O(n + n^2 + u * d + 1)$ hvor $u * d < n$
$O(n + n^2 + u * d + 1) \leq O(n^2)$

men er i stedet
$O(n + n^2 + u * d + d)$ hvor $u * d < n$
$O(n + n^2 + u * d + d) \leq O(n^2)$

- Søgning

Søgeordet hashes, og den beregnede hashværdi bruges som indgang *key* i H, og derefter foregår søgningen som i Index3, men på en brøkdel af tiden.(Idéelt $1/(H)$. Man kunne have afprøvet dette ved at plotte sandsynlighedstætheden for hashværdierne i hashfunktionen).

Søgeordet hashes på konstant tid, sammenlignes derefter med hvert ReturnItems unikke ord på tager lineær tid i u, og et ReturnItem (eller false) returneres på konstant tid, og dokumentlisten udskrives på $O(d)$ tid. Dette resulterer i et lineært tidsforbrug:
$O((n+1)/H + d) = O(n+d) = O(n)$.

4.4.2 Teoretisk kompleksitetsanalyse: Plads

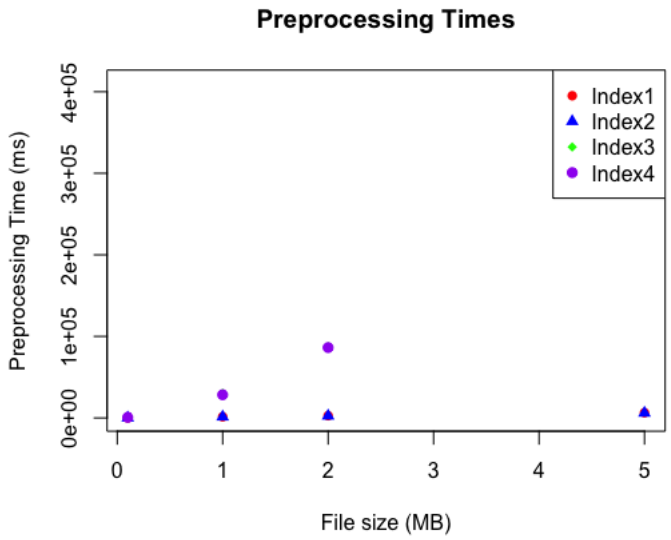
For hvert ord i inputfilen har Index4 et WikiItem, plus et WikiItem per unikke ord, og et ReturnItem per unikt ord, som holder en hægtet liste af dokumentnavne indeholdende ordet. Referencer til ReturnItems fordeles i hashtabellens H indgange, der i gennemsnit får u/H ReturnItems hver. Til sammen er pladsforbruget ligesom i Index3: $O(n + u + u * d) \leq O(2 * n + n * d) \leq O(n * d)$
$= O(n * d)$.

4.5 Eksperimentel evaluering af den basale dels søgemaskiner

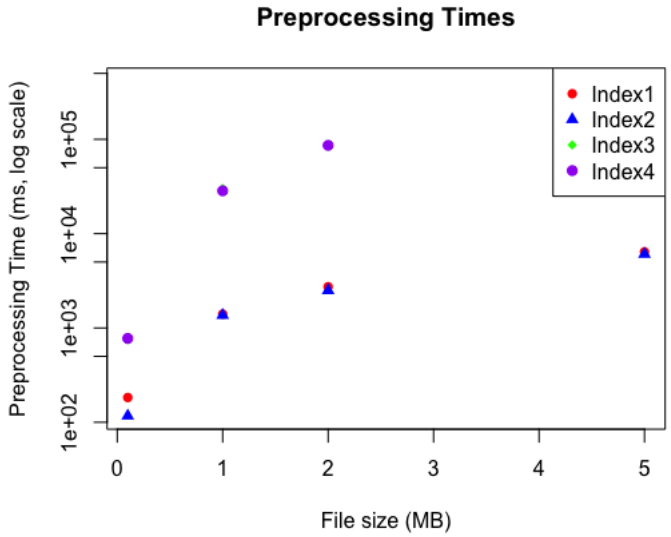
Der er foretaget empiriske undersøgelser af tid- og pladsforbruget af søgemaskinerne i den basale del.
Ved at kalde System.currentTimeMillis() før og efter kaldet til konstruktøren af hver søgemaskine, lægge forskellen i en variabel, fås tidsforbruget for at bygge søgemaskinen. [2]
Det samme gøres før og efter kaldet til search() for at få søgetiden. Og ved at kalde long beforeUsedMem = Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory();
og gøre det samme for long afterUsedMem, og trække den første fra den sidste fås pladsforbruget for datastrukturen, uden overhead fra andet i programmet. [3]

4.5.1 Eksperimentel Kompleksitetsanalyse: Byggetid

Graf over byggetid som funktion af filstørrelse. Voksende filstørrelse på første akse, og tidsforbruget for at bygge på anden akse:



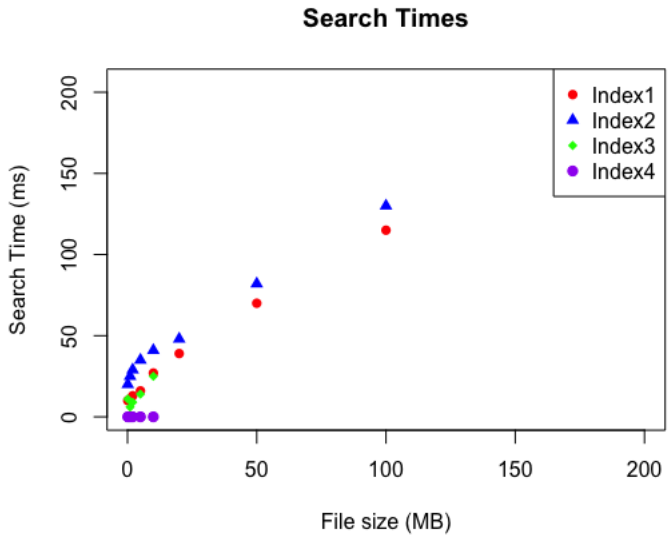
Figur 5: Byggetider: Parvis har (Index1, Index2) og (Index3, Index4) samme byggetider for samme filstørrelser, så punkterne ligger ovenpå hinanden.
Samme graf log-transformeret:



Figur 6: Log Byggetider

4.5.2 Eksperimentel Kompleksitet: Søgetid som funktion af filstørrelse n

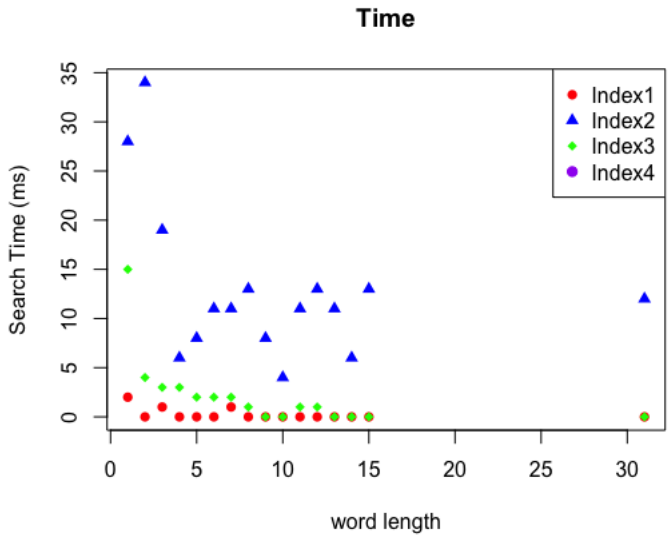
I figuren nedenfor er afbilledet søgetid som funktion af filstørrelse. Søgningerne er foretaget på ikke-eksisterende ord. Dermed er søgetiden uafhængig af placering(erne) af ordet: Index1 og Index2 skal sammenligne søgeordet med alle ord i inputfilen. Index3 og Index4 skal derimod kun sammenligne med alle unikke ord i inputfilen. De har ofret byggetid og plads, for til gengæld at forbedre søgetiderne.
Voksende filstørrelse n på første akse, og tidsforbruget T for at søge på anden akse:



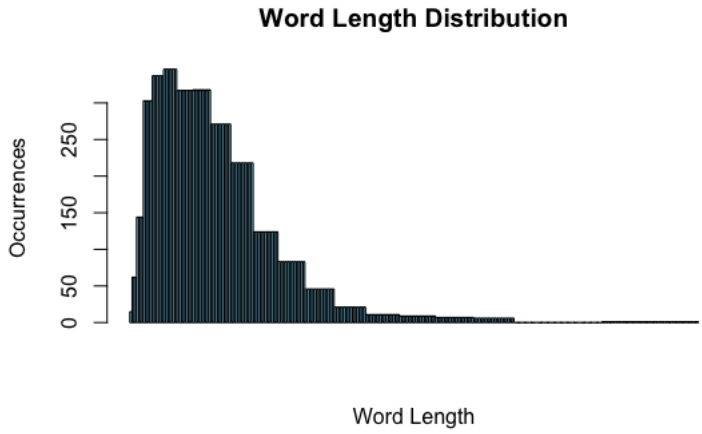
Figur 7: Søgetider som funktion af filstørrelse

4.5.3 Eksperimentel Kompleksitetsanalyse: Søgetid som funktion af ordlængde l

Ved at finde det længste ord i filen, kan man måle tidsforbruget for at søge på det længste ord, det korteste ord, og ord med længde steder imellem. Første fil har ordlængder fra $l \in \{1...31\}$. Man kan finde ord af en ønsket længde i en terminal ved: -woE 'ordlængde' 'filnavn'.
Graf over tidsforbruget som funktion af ordlængde l: voksende ordlængde l på første akse, og tidsforbruget T for at søge på det i datastrukturen (den hægtede liste af længde n i Index1 og Index2) og ReturnItem-listen af længde u i Index3, og i en indgang til en hashtabellen af længde H i Index4):

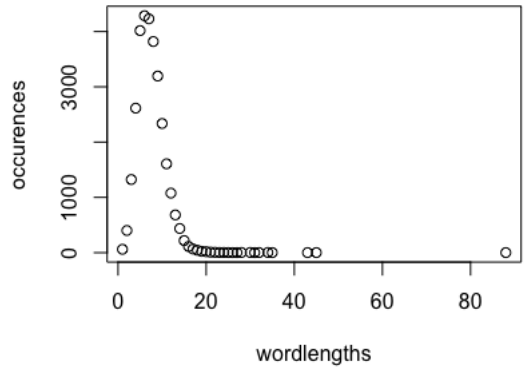


Figur 8: Søgetider som funktion af ordlængde l
Ordlængder: 100Kb-filen har ord af længde l=1-22, og følgende fordeling af ordlængder:



Figur 9: Søgetider som funktion af ordlængde

2MB-filen har følgende fordeling af ordlængder l: Gennemsnittet er 7.5 og standardafvigelsen er 3.0. Der kune tages højde for outliers: Kun tre ord er over 35 karakterer. Længste ord på 88 karakterer er to URL'er.

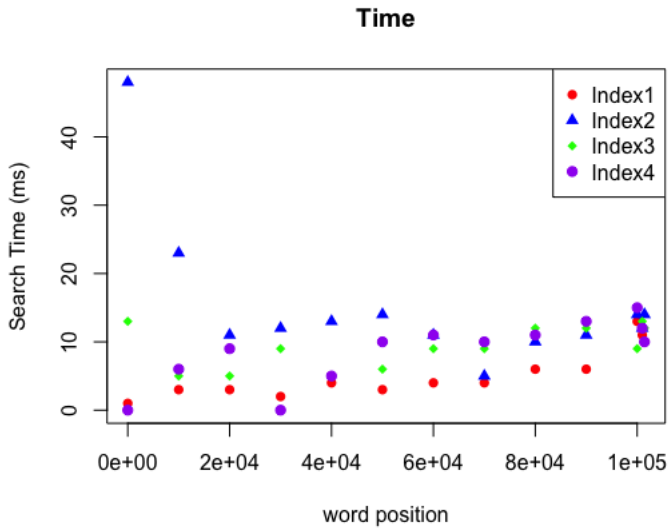


Figur 10: Fordeling af ordlængder l

Med de forskellige ordlængder l kan man bestemme søgetiden som funktion af l. Ord af forskellig længde men med nogenlunde samme placering x, kan benyttes for at opveje for ordets tilfældige placering.

4.5.4 Eksperimentel Kompleksitetsanalyse: Søgetid som funktion af søgeordets placering x

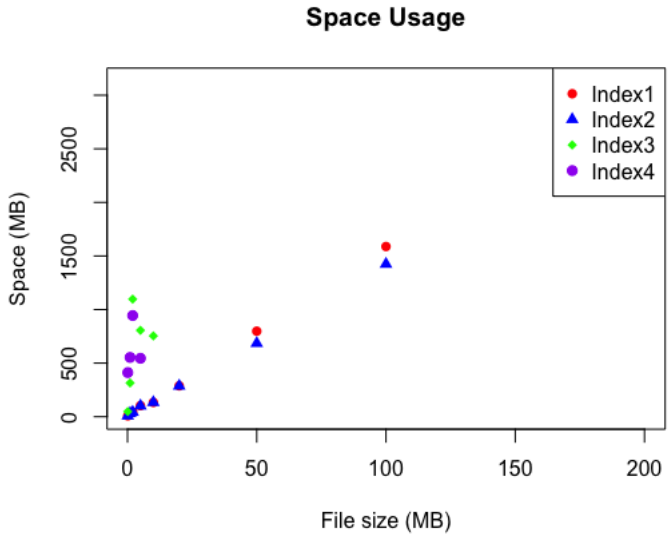
Ved at søge på hvert titusinde ord i 5 MB-filen: ord $x = 1$, 10.000, 20.000,...,101.000,101.436 fås følgende søgetider som funktion af ordplacering.



Figur 11: Sammenligning af forskellige forskellige søgemaskiners søgetid som funktion af søgeordets placering $T(x)$.

4.5.5 Eksperimentel Kompleksitetsanalyse: Pladsforbrug P

Graf over pladsforbrug som funktinon af filstørrelse n:



Figur 12: Sammenligning af forskellige Indexfilers pladsforbrug P som funktion af filstørrelse n. Filstørrelse på den ene akse, og pladsforbruget for at søge i datastrukturen på den anden akse.

4.6 Diskussion af eksperimentel evaluering af den basale dels søgemaskiner

Byggetid: Ved at køre Index1 og Index2 med datafiler af voksende størrelse n observeres en lineært voksende byggetid i n. Punkterne for Index1 og Index2 ligger sammenfaldende. Det samme er tilfældet for Index3 og Index4. Det ses også at Index3 og Index4 har en kvadratisk voksende byggetid i n. Da et hvert sprog har et konstant antal unikke ord u, kan man forvente $u'(n) \to 0$ når $n \to \infty$ at antallet af nye unikke ord u' går mod nul når filstørrelsen vokser, og at sandsynligheden for at møde et nyt ord falder hurtigt (men langsommere og langsommere $u''(n)$), indtil man til sidst når alle ord.

Søgetid: Søgetiden på ikke-eksisterende ord (worst-case søgning) i Index1, Index2 og Index3 vokser lineært med filstørrelsen. Da Index4 benytter en hashtabel, er søgetiden i praksis nær konstant, selv om

den i væste fald er lineær i antallet af unikke ord u (generaliseret til n).

Søgeordets placering x: Hvis man starter med at måle på ordene på de første positioner, er der muligvis en *locality*-fordel i cache, ved at søge på hvert tusindende ord, som træder i kraft efter første søgning. Måske af samme årsag forbedres tiden ofte efter første kørsel.

Plads: Index1 og Index2 har ens, lineært voksende pladsforbrug P som funktion af filstørrelsen n. Index3 og Index4 har kvadratisk pladsforbrug som funktion af n: $O(n^2)$.

For at finde et element i en hægtet liste, starter man med at bruge en midlertidig reference *current*, som sættes til det samme som referencen til starten af den hægtede liste *start*. Hvis man vil fjerne et element, kan man i elementet inden ændre referencen til næste til næstes næste. I så fald er der ikke længere nogen reference til det midterste af de tre elementer, og Javas garbage collector vil frigøre pladsen elementetet optager. Derfor vil en hægtet liste ikke optage mere plads end nødvendigt $O(n)$.

Dynamiske arrays er et eksempel på en datastruktur, som ikke nødvendigvis har lige så effektiv udnyttelse af plads: Den fordobles i længde hvis fyldt, og halveres i længde hvis kvart fyldt. Hvis man har et stort array kan det have en negativ påvirkning af tid T og pladsforbrug P, da der er en proportionalitet i tid og pladsforbrug beskrevet af memory hierakiet.

Hvis man derimod havde en række for hvert *ReturnItem*, ville man kunne beregne den nøjagtige plads af objektet, uden at skulle lede efter elementet i en hægtet liste. I så fald ville man forbruge mere plads.

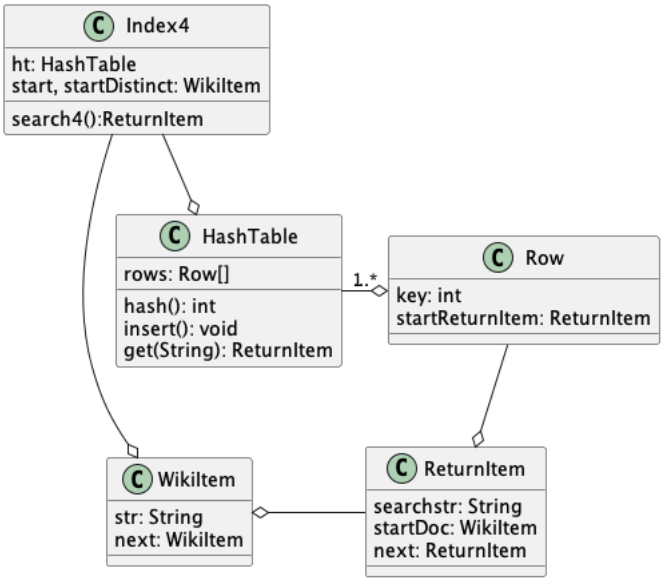
Tid og Plads: Man kan i Index4 overveje, hvor mange rækker/ indgange man ønsker i sin hashtabel. Hvis man har én indgang, altså et array af længde én: Row[1], ville man være i samme situation som i Index3 med hensyn til tid og plads, fordi alle objekterne ville være hægtet på samme række. Man ville godt kunne lave en hashtabel med flere indgange H end der er brugt her, og stadig få en skalérbar søgemaskine. Det ville give færre kollisioner hvor $h(s1) = h(s_2)$ for to forskellige tekststrenge s_1 og s_2, og medføre mindre *chaining*. I bedste fald (mht tid) kunne man nøjes med at beregne et ReturnItems hashværdi, og refereren til det i hashtabellen H. Begge dele på konstant tid.

4.6.1 Hashtabel og Relation til Binære søgetræer

I værste fald kan en hashfunktion placere alle elementer i samme indgang/ spand. I så fald ligger den fulde hægtede liste, man forsøgte at fordele på indgangene, i samme indgang i hashtabellen. Det svarer til at man initialiserer et binært søgetræ ved at indsætte tal i en sorteret rækkefølge, og derfor kun får højre child-nodes, eller kun får venstre child-nodes. I så fald er både hashtabellen og det binære søgetræ reduceret til en linked list. Binære søgetræer kan implementeres som balancerede B-træer f.eks. 2-3-4-træer. Søgetræerne balancerer sig selv ved at foretage rotationer. Dermed kan de garantere logaritmisk søgetid $log(n)$. Med dybde h er alle pladser fra roden optaget, bortset fra i h, hvor pladserne er fyldt fra venstre.

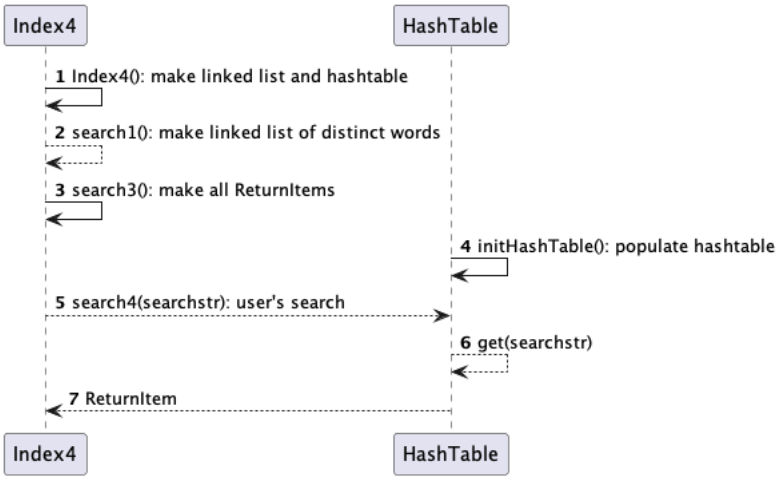
5 Design

Den basale del har følgende klasser:

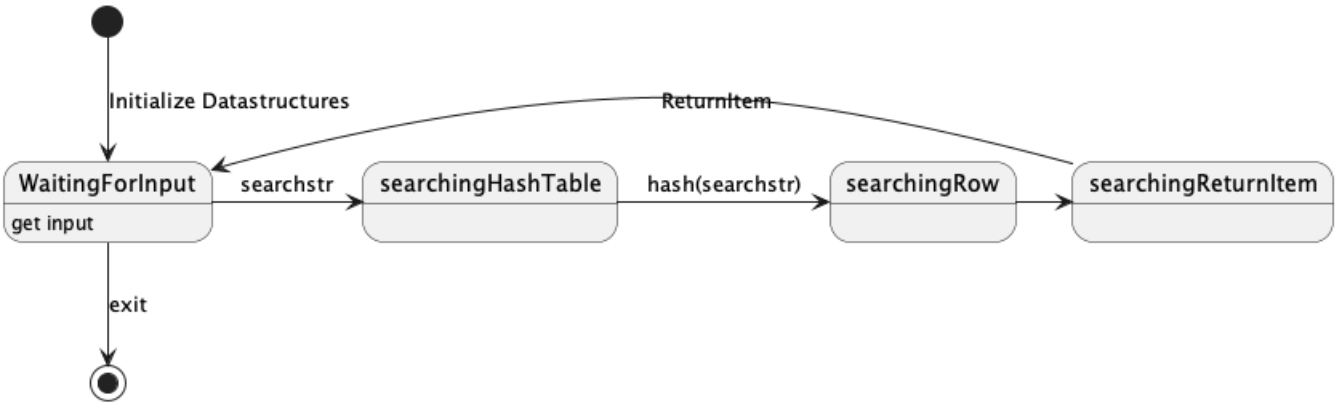


Figur 13: Klassediagram for den basale del

Sekvensdiagram for den basale del illustrerer metodekaldene, deres sammenhæng og deres rækkefølge i programmet.



Figur 14: Sekvensdiagram for den basale del



Figur 15: Tilstandsdiagrammet viser tilstandene programmet gennemgår under kørsel.

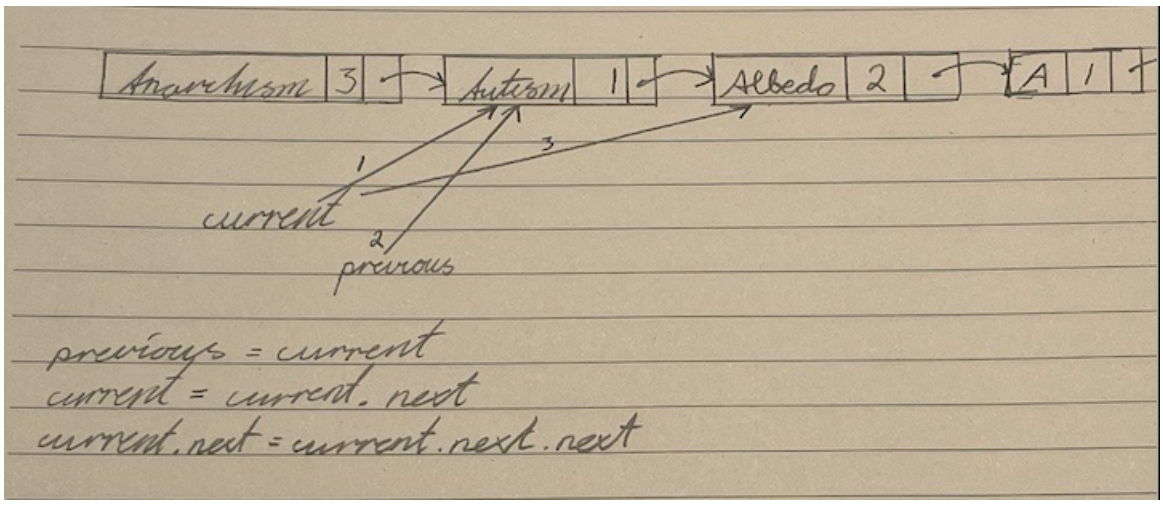
6 Avanceret del

6.1 Index5: Rangering

Index5 er en udvidelse af den basale del, som har rangeret dokumenterne i søgningerne på antallet af forekomster af det unikke ord. Datastrukturen for hver enkelt søgning er ændret fra WikiItem til DocItem, som har en String som dokumentnavn, et heltal til antal forekomster, og en reference til næste. Når hver søgning på et unikt ord *ReturnItem* bygges, tælles antallet af forekomster i det pågældende dokument mellem hver —END.OF.DOCUMENT—. Hvis antallet er forskelligt fra nul tilføjes et DocItem med dokumentnavnet, og antal forekomster til ReturnItem'ets *startDoc*. Brugeren får et ReturnItem, ved søgning, og når det er hentet fra hashtabellens get(searchstr), kalder ReturnItem'et bubble sort på sit startDocItem, som sorterer den hægtede liste på antal forekomster. Når søgningen printes, skrives en linie for hvert dokument med dets rang, dokumentnavn og antal forekomster af det unikke ord.

6.2 Bubble Sort

I Index5 sorteres den hægtede liste af dokumenter med bubblesort[1]. Algoritmen tager en hægtet liste af dokumenter af længde d, og opbygger et sorteret hægtet liste bagerst i den hægtede liste. Det gøres ved i hvert gennemløb, at sammenligne nabo-noder fra starten af listen. Hvis $current < next$ bytter de plads, og den største sammenlignes med sin nye næste, indtil den ikke har nogen mindre end sig foran sig. En gradvist voksende sorteret andel opbygges for enden af den hægtede liste. Efter d^2 sammenligninger, og d^2 ombytninger (worst case), er listen sorteret. Bubblesort har køretid på $O(d^2)$. Pladsforbruget er $O(d)$, da den ikke bruger yderligere plads, end det den hægtede liste bruger.[4]



Figur 17: Eks.: Bubble sort swap andet og tredje WikiItem-element. Man kan forestille sig elementerne i bubblesort med forskellige værdier, som bobler med forskellig massefylde: Ligesom det med lavest densitet stiger op, derefter det med næstlavest og tredjelavest densitet, således bobles den højeste værdi på plads i første iteration, den næsthøjeste i anden iteration. På samme vis kunne man forestille sig vand, luft og olie boble sig på plads ift. hinanden fra deres densiteter.

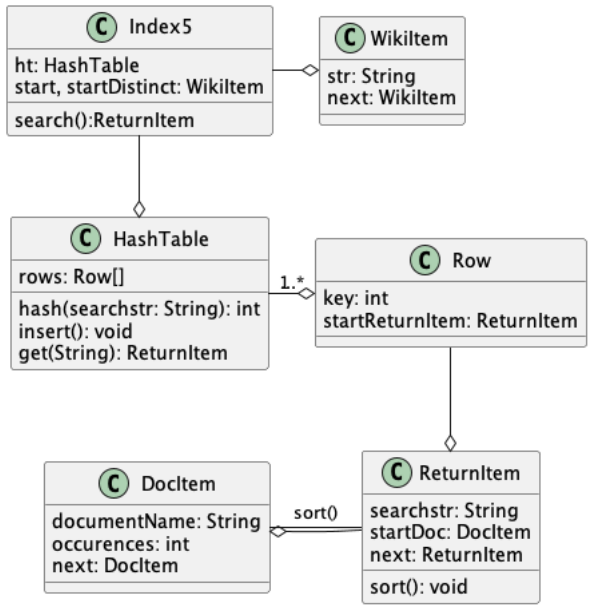
6.3 Teoretisk kompleksitetsanalyse: Tid

- Søgning

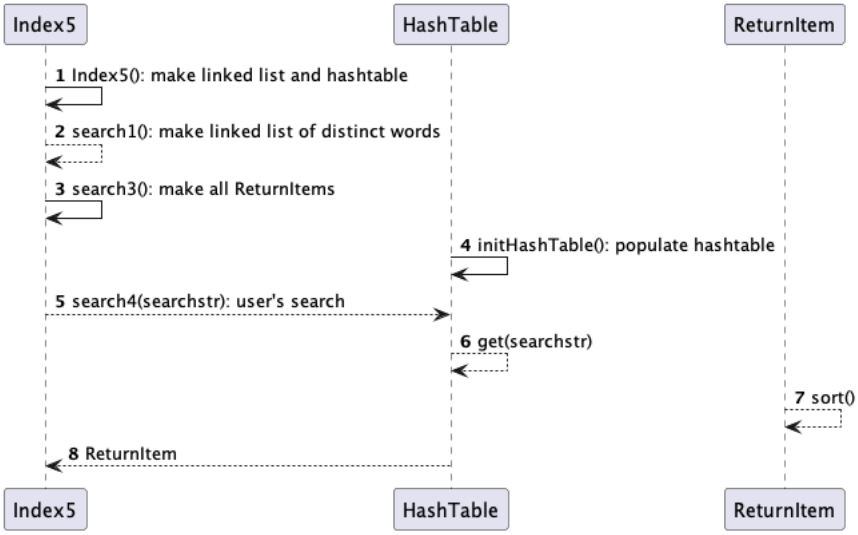
Søgeordet hashes, og den beregnede hashværdi bruges som indgang (key) i Row-arrayet, og derefter foregår søgningen som i Index3, men på en brøkdel af tiden, idéelt 1/(antal indgange). (Man kan afprøve dette ved at plotte sandsynlighedstætheden for hashværdierne). Søgeordet sammenlignes med hvert ReturnItems søgeord u, som tager lineær tid, og et ReturnItem (eller false) findes på konstant tid $O(u+1) = O(n)$. Dokumentlisten sorteres med Bubblesort på $O(d^2)$. Til sammen et tidsforbrug på: $O((n+1)/H + d^2) = O(n + d^2)$

[1] [4]BubbleSort

Klassediagram for Index5 i den avancerede del:



Figur 18: Klasserne, deres attributter og deres relation til hinanden.

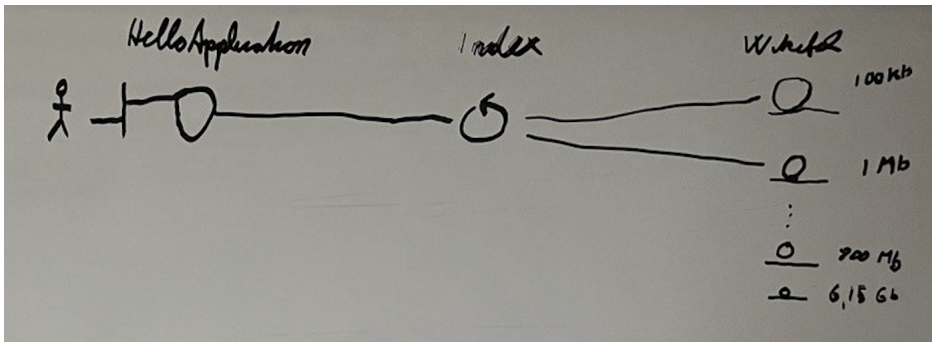


Figur 19: Sekvensdiagrammet illustrerer kommunikationen mellem klasserne.

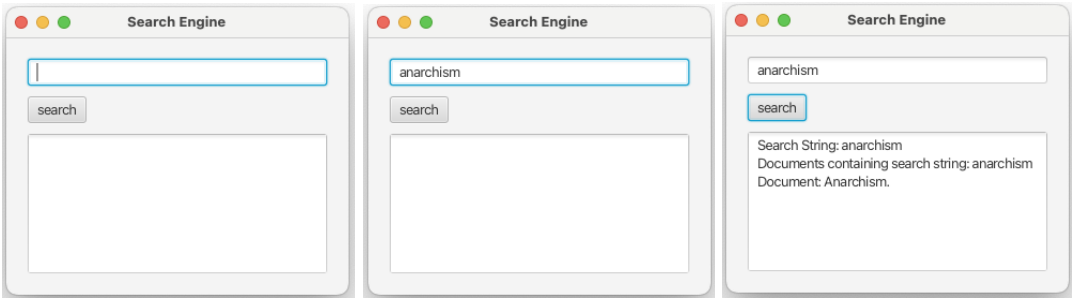
6.4 Index6: Omvendt præfikssøgning

(Præfikssøgning er i Index8). Index6 er en udvidelse af Index4 med omvendt præfikssøgning: Index6 gemmer brugerns søgeord i en midlertidig String *s*. Hvis *s* er i listen af unikke ord, søger Index6 i hashtabellen efter dens ReturnItem og printer den tilhørende dokumentliste. Derefter fjernes et bogstav af søgestrengen, inden der igen søges i hashtabellen. Dette gentages indtil der ikke er flere bogstaver at fjerne fra søgestengen. Hvis man f. eks. søger på *Anarchism*, foretager søgemaskinen en søgning på præfiksene *An* og *A*, fordi de også optræder i listen af unikke ord.

Ved præfikssøgning (Index8) ønsker man at søge på dokumenter, der indeholder unikke ord der har brugerens søgestreng *s* som præfiks: Søger man på "s*", ønskes en mængde af ord (præfiks) der starter med *s*, og på hvert af dem kan man foretage en søgning i hashtabellen, og få en tilhørende dokumentliste.



Figur 20: *Boundary:* Java FX App, *Control:* Index4.java, *Entity:* wiki-datafil.



Figur 21: Brugergrænseflade

6.5 Index7: Boolsk søgning

Index7 modtager to søgestrenge s_1 og s_2.

AND: En tom streng *returnStringAND* initialiseres. Hvert dokumentnavn i første søgeords dokumentliste sammenlignes med hvert dokumentnavn i den anden søgnings dokumentliste. Hvis de er ens tilføjes de til *returnStringAND*.

OR: En tom streng *returnStringOR* initialiseres. Hvert dokument i første søgnings (ReturnItem's) hægtede liste af dokumenter tiløjes til *returnStringOR*. Derefter tjekkes alle dokumentnavne i det anden ReturnItem, og hvis dokumentnavnet endnu ikke er tilføjet returnStringOR, tilføjes det. Til sidst returneres returnStringOR dokumentlisten med unikke dokumentnavne. Derefter printes de to søgeord, *returnStringAND*, og *returnStringOR*.

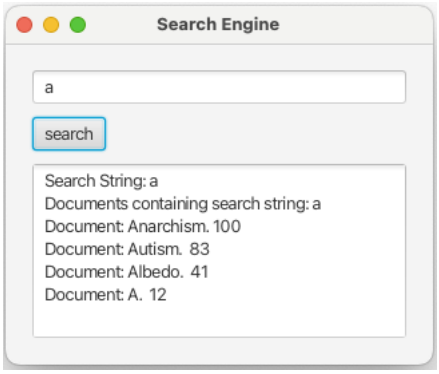
6.6 Index8: Præfikssøgning

Hvis brugeren skriver '*' i slutningen af søgeordet, fjerner Index8 '*', og søger på alle ord der starter med alt før '*'. Det sker ved at kalde en søgning i listen af unikke ord, der kalder String-metoden *startsWith(searchstr)*, i stedet for *equals(searchstr)* som hidtil, og derefter slår det unikke ord op i hashtabellen og returnerer dokumentlisten for hvert af de unikke ord, brugerens søgestreng er præfiks til.

6.7 GUI

Der er lavet en GUI ved hjælp af Java FX. Den bruger den basale løsning Index4 til at søge i. Brugergrænsefladen består af et tekstinputfelt, en knap og et tekstfelt til output. Når den køres starter den Index4, og venter på at brugeren taster ind i tekstboksen og trykker på knappen. Derefter læser den indholdet af tekstboksen ind i en String, søger i hashtabellen efter det ReturnItem der har brugerens ord som sit unikke ord, og fylder nederste *TextArea* med ReturnItemets unikke ord og dens tilhørende dokumentliste.[5]

Index5 er også brugt med GUI, hvor søgeordets dokumentliste er rangeret og sorteret efter antal af forekomster af ordet.



Figur 22: GUI for Index5: Søgemaskine med rangering

6.8 Valg af Brugergrænseflade

Java FX er valgt for at bruge Java Virtual Machine (JVM), da det er mest oplagt at forblive i Java mht. funktionalitet og performance.

6.9 Hashfunktion

Hashfunktionen har til formål dele data op i bidder.
For at skabe en hashfunktion er der afprøvet nogen improviserede forslag, ved simulering. Da hashfunktionen ikke fordelte særlig jævnt, blev der søgt forskellige kilder til at skrive den. Bogstever konverteres til ASCII-værdier for at nå fra bogstaver til tal. Bagefter er det nødvendigt at adderes(+) og multiplicere(∗) med primtal, for at give jævnt fordelte sandsynligheder for modulusfunktionen idét primtal har den egenskab at kun tallet ét går op i tallet. Til sidst tages modulus (%) med antal indgange (dette bør være et primtal). Til sidst er genereret (vha. AI) en hashfunktion der givet en unik String s, beregner en hashværdi h af det unikke ord s $h(s) \in \{1...H\}$, hvor H er længden af hashtabellen. Når hashfunktionen skal bruge en unik attribut fra inputtet s er det heldigt at hvert ReturnItem er kendetegnet ved et unikt ord. Variablen H er sat til 10, men kunne med fordel være et primtal. h(s) skal skal under alle omstændigheder antage værdierne i udfaldsrummet med samme sandsynlighed. $P(h(s) = i) = 1/H, \forall i \in \{1...H\}$

```
1: H := 10                                                    ▷ Tabellængde H
2: procedure HASH(String s, int H)
3:     int hash = 7
4:     for int i in 1 to s.l do
5:         hash = hash * 31 + s.charAt(i)
6:     end for
7:     hash = |hash| mod H
8:     Return hash                                            ▷ hash ∈ {1...H}
   end procedure
```

Figur 23: Hashfunktion: modulus og abs står ikke som matematiske symboler, da symbolerne er ens.

7 Konklusion

7.1 Konklusion på den basale del

Det er lykkedes i den basale del at få søgemaskinen til at modtage et søgeord fra brugeren, og derefter at udskrive alle dokumentnavne der indeholder søgeordet. Det er også lykkedes at skabe en hægtet liste af objekter, som hver repræsenterer en søgning på et unikt ord. Det gøres ved at holde en hægtet liste af dokumentnavne, som ordet optræder i, og ved søgning at gennemløbe objekterne som var det en hægtet liste. Dette gør søgningen hurtigere, fordi søgemaskinen allerede har skabt søgningen på det unikke ord,

og derfor bare skal finde elementet i den hægtede liste af søgninger på unikke ord, ud fra det unikke ord der modtages når brugeren søger.

Som afslutning på den basale del er der i Index4 lavet en hashtabel, som er befolket med søgningerne fra Index3. Efter at Index4 under preprocesseringen har skabt listen af alle unikke ord, og listen af søgningerne på alle unikke ord fra Index3, afsluttes preprocesseringen i Index4 med at alle u søgninger *hashes* på deres unikke ord. (Hvert ReturnItem burde hashes lige når det laves).

Søgefunktionen i Index4 tager et søgeord fra brugeren, beregner ordets hashværdi som indgang i hashtabellen, og slår op i hashtabellen efter dens dokumentliste i dets ReturnItem (hvis det findes). Hashværdien er indekset (key) i arraet af længde H (hashtabellen). Her ligger rækken der har referencen til den hægtede liste af søgninger (ReturnItems) med samme hashværdi. Til sidst sammenlignes søgestrengen med hvert ReturnItems unikke ord, i den valgte indgang. Hvis de er ens svarer søgemetoden med det fundne unikke ord, og dets hægtede liste af dokumentnavne med søgeordet printes ud. Ellers printer metoden en besked om at ordet ikke findes.

For hver søgemaskine er køretid analyseret som funktion af inputstørrelse n for at bygge datastrukturen, køretid som funktion af (inputstørrelse n, ordlængde l og ordplacering x) brugt på at søge i den givne søgemaskinens datastruktur. Afslutningsvis er der analyseret pladsforbruget af datastrukturen (hægtet liste, hægtet liste af hægtede lister(ReturnItems), og hashtabel), som funktion af inputstørrelse n, antal dokumenter d, ordplacering x og ordlængde l.

7.2 Konklusion på den avancerede del

I den avancerede del er der lavet rangering på dokumentlisterne ved at definere DocItem som en WikiItem, der er udbygget til at have et heltal for antal forekomster. Der er implementeret (omvendt) præfikssøgning ved iterativt at fjerne et bogstav ad gangen fra brugerens søgeord, inden der søges på næste præfiks af søgeordet. Der er lavet boolsk søgning ved at bygge en String til henholdsvis *AND* og *OR* for to søgeord s_1 og s_2. Der er lavet præfikssøgning ved at søge på unikke ord, der har brugerens søgeord som præfiks (sand præfiks). Der er lavet en brugergrænseflade som fungerer med den basale løsning Index4, og den første avancerede løsning Index5 (med rangering).

8 Refleksion

8.1 Relation mellem hægtede lister, hashtabeller, binære søgetræer

Hashfunktion kan i værste fald placere alle elementer i samme Row (indgang). I så fald ligger den fulde hægtede liste, man forsøgte at fordele på indgangene i samme indgang i hashtabellen. Det svarer til at man initialiserer et binært søgetræ ved at indsætte tal i en sorteret rækkefølge, og dermed kun får højre child-nodes, eller kun får venstre child-nodes. I så fald er både hashtabellen og det binære søgetræ reduceret til en linked list.

Binære søgetræer kan implementeres som balancerede (B-træer) f.eks. 2-3-4-træer. Søgetræerne balancerer sig selv ved at foretage rotationer. Dermed kan de garantere logaritmisk søgetid $log(n)$. Med dybde h er alle pladser fra roden optaget, bortset fra i h, hvor alle blade sidder ved siden af hinanden i venstre side.

8.2 Søgetider

Umiddelbart gælder det at: Hægtede lister har søgetid på O(n). Hashtabellen har O(n) (worst case, uden antagelser, og med hele den hægtede liste på en enkelt indgang i hashtabellen). Binære søgetræer har ligesom hægtede lister en søgetid på O(n).

Men chained hashing fås under antagelse af simple uniform hashing (SUHA) en søgetid på O($1 + \alpha$), $\alpha = n/m$, hvor *alpha* er *load factor*, og m er længden (her kaldet H). Køretiden afhænger af $\alpha = n/m$. [8]

Og binære søgetræer kan være selvbalancerende ved hjælp af rotationer, og opnå en Worst case søgetid (afstand til hvert blad) på O($log\ n$).

Asymtotisk gælder at:

$1 < log(n) < n, n \to \infty$

Så under antagelsen (der ikke er bevist her) er hashtabeller hurtigst, balancerede binære søgetræer næsthurtigst, og hægtede lister det langsommeste af de tre. Open adressing er et alternativ til chaining, hvor køretiden er givet under andre antagelser end $SUHA$.

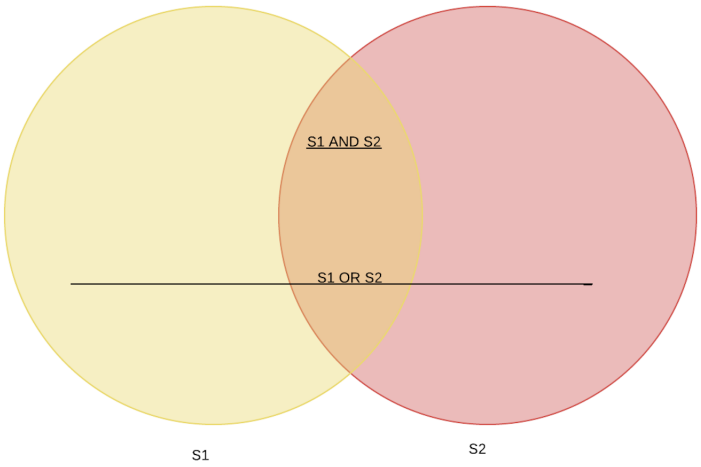
Afhængig af sammenhængen kan man prioritere best-case, worst-case, eller amortiseret køretid. I distribuerede systemer prioriteres amortiseret køretid, da man ikke ved hvor længe brugerdata bliver *cached* på en node (god performance belønnes med point). Point bruges til at vælge noden at starte med at spørge næste gang.

8.3 Hægtede lister, ReturnItems og Hashtabellen

Referencerne *start*, *tmp* og *current* bruges til at referere til en plads den hægtede liste. Det samme gælder for referencerne (*startDistinct*, *tmpDistinct* og *currentDistinct*), (*startReturnItem*, *tmpReturnItem* og *currentReturnItem*. Når man bruger hashtabellen vælger man det startReturnItem der ligger på pladsen i hashtabellen, der er ens med hashværdien af det unikke ord man indsætter eller søger efter.

8.4 JVM garbage collector

Javas virtuelle maskine har en garbage collector som fører log over hvilke variabler der har referencer der peger på dem. Når man i Java skriver $< Type, variabelnavn >$ allokeres hukkommelse (i RAM) til en variabel af den type og størrelse. Mængden af plads i hukommelsen er for simple datatyper mellem én og otte bytes afhængig af typen, men altid mindst én byte, da JVM bruger bytecode, i stedet for bit. En byte er en blok af otte bit af værdien ét eller nul.



Figur 24: Venn diagram: boolsk søgning for to søgestrenge s_1 og s_1's dokumentlister S_1 og S_2.

8.5 Boolsk søgning

8.5.1 Merge vs flere ReturnItems

Man kan forestille sig at man i stedet for at *"merge"* to *ReturnItems'* dokumentlister, lavede et *returnItemAND* og *returnItemOR*, men det ville give $(u(u-1))/2$ *returnItems*, og ikke være skalerbart. *"Merge"*-tilgangen er et mere praktisk valg, da der ikke kommer flere ReturnItems ud af det, og fordi der ønskes en skalerbar søgemaskine.

8.5.2 Boolsk søgning: Antal søgeord

Boolsk søgning kunne udvides til mere end to søgeord. Antallet af mulige kombinationer fås som flg. sum af binomialkoefficienter (vælg z ud af u), hvor z søgeord er til stede: $\sum_{z=1}^{U} \binom{u}{z} = \sum_{z=1}^{U} \frac{u!}{z!(u-z)!}$
Der kan blandes lige så mange søgestrenge, som der er unikke ord u. Med u søgeord vokser mængden af dokumentlister med $u!$ (u fakultet). Man kan forvente at antallet af dokumenter d i AND falder da ingen dokumenter indeholder alle u unikke ord. Man kan desuden forvente at antallet af dokumenter i OR stiger da alle dokumenter indeholder mindst ét af de unikke ord.

9 Perspektivering og idéer til videreudvikling

Index5 og frem kan kombineres, for en mere avanceret søgemaskine. Index5 og Index6 kunne kombiners, så man søgte på alle præfiks, og foretog rangering af dokumentlisten tilhørende hvert præfiks Man kunne desuden rangere præfiks på deres længde l' (korteste først).

Kompression: Data kan komprimeres, ved at kun at gemme hvert unikt ord ved første forekomst, og derefter nøjes med at gemme placeringerne af de næste forekomster som referencer tilhørende det unikke ord. Det kræver at samme ord optræder mange gange, hvilket også er tilfældet for de fleste ord. I så fald ville man kunne genskabe data præcis som inden kompression (*loss-less compressoin*), i modsætning til når man mister data f. eks. når man komprimerer billeders RGB-værdier på et område af pixels til gennemsnittet af de omgivende pixels værdier, og dermed får pixelerede/grynede output. Programmet kunne tage den komprimerede .txt.bz-fil, pakke ud, og repræsentere som unikke ord med et sæt af placeringer.

Finite Automaton kan implementeres som beskrevet i CLRS kapitel 32, hvor man finder et pattern P i en tekst T. Med Knuth-Morris-Pratt algoritmen kan man forbedre matching tiden fra $O(n - m + 1)$ i den naive algoritme til $O(n)$. hvor n er ordlængde n og m er mønsterlængde. [9]
I forbindelse med præfikssøgemaskinen kunne man anvende KMP-algoritmen med alle u ord i unikordlisten startDistinct, hvor alle ord i unikordlisten både tages som tekstinput T, og pattern P.

Dynamic Indexing Man kan fjerne en søgning på et unikt ord ved at løbe hen til et valgt returnItem, som har det ReturnItem man ønsker at fjerne som *next*, og sige currentReturnItem.next = currentReturnItem.next.next. På den måde kan man afkorte den hægtede liste af *ReturnItems*. Det kan være en fordel mht. plads hvis dette ReturnItem er en søgning på et ord der er hyppigt på tværs af dokumenter, og derfor har en lang dokumentliste.

Trie Suffikstræer er en datastruktur hvor ord med fælles præfiks deler undertræ. Efter deres fælles præfiks forgrener de sig.

GUI Man kunne starte med at lade brugeren vælge en søgemaskine, og skifte skærmbillede afhængig af valget. Hvis brugeren vælger Index7, som har boolean search, kunne der komme to felter til søgeord frem i GUI'en. Man kunne i den forbindelse også lade brugeren vælge mellem filstørrelserene. Desuden kunne man have koden inddeling i fire lag: UI, UI-logik, søgemaskinelogik, data. Med Boundary Control Entity-opdelingen opnås indkapsling og separering af logik og brugergrænseflade. Det gør det lettere at skifte mellem løsningerne i de forskellige Indexfiler.

Litteratur

[1] **Hashing MIT OpenCourseware forelæsning af Erik Demaine**
https://www.youtube.com/watch?v=0M_kIqhwbFo&t=650s

[2] **Tidsforbrug**
https://www.geeksforgeeks.org/measure-time-taken-function-java

[3] **Pladsforbrug**
https://stackoverflow.com/questions/37916136/how-to-calculate-memory-usage-of-a-java-program

[4] **Bubble Sort**
https://www.geeksforgeeks.org/bubble-sort-for-linked-list-by-swapping-nodes/

[5] **GUI Java FX**
https://www.youtube.com/watch?v=FLkOX4Eez6o&list=PL6gx4Cwl9DGBzfXLWLSYVy8EbTdpGbUIG&index=1
Git repository (muligvis nede):
https://github.com/thenewboston-developers?q=&type=all&language=&sort=

[6] **GUI Java FX: opdeling**
https://stackoverflow.com/questions/33881046/how-to-connect-fx-controller-with-main-app

[7] **Android**
https://docs.google.com/document/d/1LYW4yekzcLuXNAxRywkHjSDfIXnI6xlENd66Q4lDHz0/edit#heading=h.1kzvzhq3t3u

[8] **suha**
Simple Uniform Hashing
https://en.wikipedia.org/wiki/SUHA_%28computer_science%29/

[9] **CLRS**
Introduction to Algorithms, Fourth Edition By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein