PG4200 - Innlevering 1

av Lasse André Arnesen og Eivind Vegsundvåg

Øvrige forutsetninger

Kilder

Vi har brukt følgende JavaDokumentasjon og kildekode som begrunnelse for våre påstander:
http://docs.oracle.com/javase/8/docs/api/java/util/List.html
http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html samt kildekode
http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html samt kildekode
Kildekode for edu.princeton.cs.algorithms.Queue og edu.princeton.cs.algorithms.Stack

Målinger utført i nanosekunder

Ettersom enkelte søkemetoder alltid ga resulter i mindre enn ett millisekund, har vi skrevet vår egen stoppeklokke som måler tid angitt i nanosekunder. Ettersom metoden System.nanoTime() ikke kan love nøyaktighet i nanosekunder, har vi valgt å forkorte antall desimaltegn ned til mikrosekunders presisjon. Dette kan dog fortsatt ikke være 100% nøyaktig, og vi har derfor valgt å ikke avrunde disse tallene. Disse burde derfor tolkes med en feilmargin på opp til ett helt mikrosekund i tillegg til enkelte avvik i den faktiske nøyaktigeten da nanoTime() bare lover nøyaktighet i millisekunder.

Versjon av edu.princeton.cs.algorithms og princeton.cs.introcs

Vi har brukt versjon 4.0.1 av dette biblioteket, og den versjonen av edu.princeton.cs.introcs som kreves for å bruke denne versjonen. Det forutsettes derfor at man bruker den nyeste publiserte versjonen (20. september 2014) av dette biblioteket for å kjøre koden.

Oppgave 1

Forutsetninger

Vi har endret testforbedelsene til at de returnerer et tall vi vet befinner seg i listen, og tar så gjennomsnittet av tiden testen bruker på 100 gjennomføringer(Kjøres 100 ganger for å sikre at testene er nøyaktige nok til vårt formål. Burde kjørt de enda fler ganger, men på grunn av tiden det vil ta å kjøre testene har vi satt 100.). Dette gjør vi fordi vi så at med vår størrelse på listene var sannsynligheten for at tallet faktisk eksisterte ekstremt lav. Hvor i listen objektet ligger er helt tilfeldig. Noen tester kan feile søkingen, fordi vi har satt en øvre grense per søk til 10 sekunder. I disse tilfellene vil grafen ikke ha et punkt for verdien.

Vi tester også hvor lang tid det tar å sette opp listen før gjennomsøking, men i disse tilfellene sorterer vi ikke tallene på forhånd, slik at vi får målt tiden det tar å legge til gjenstander i listen

uten at sorteringsfunksjonen til listen står for mesteparten av tidsbruket. Slike tilfeller medfører dog at søkealgoritmen BinarySearch ikke kan brukes. Dette påvirker ikke gjennomføringen av testene.

Deloppgave 1

Se kode for løsning.

Deloppgave 2

Den desidert raskeste kombinasjonen er ArrayList med binær søkefunksjon.

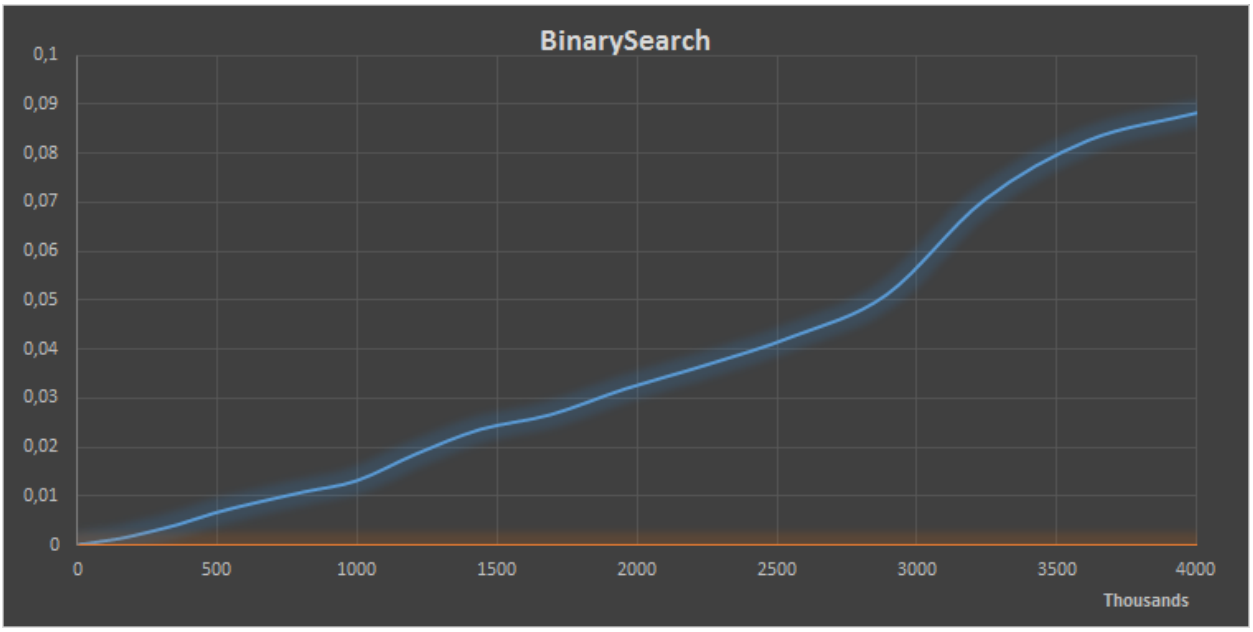
Den desidert tregeste kombinasjonen er LinkedList med søkefunksjonen SequentialIndexSearch.

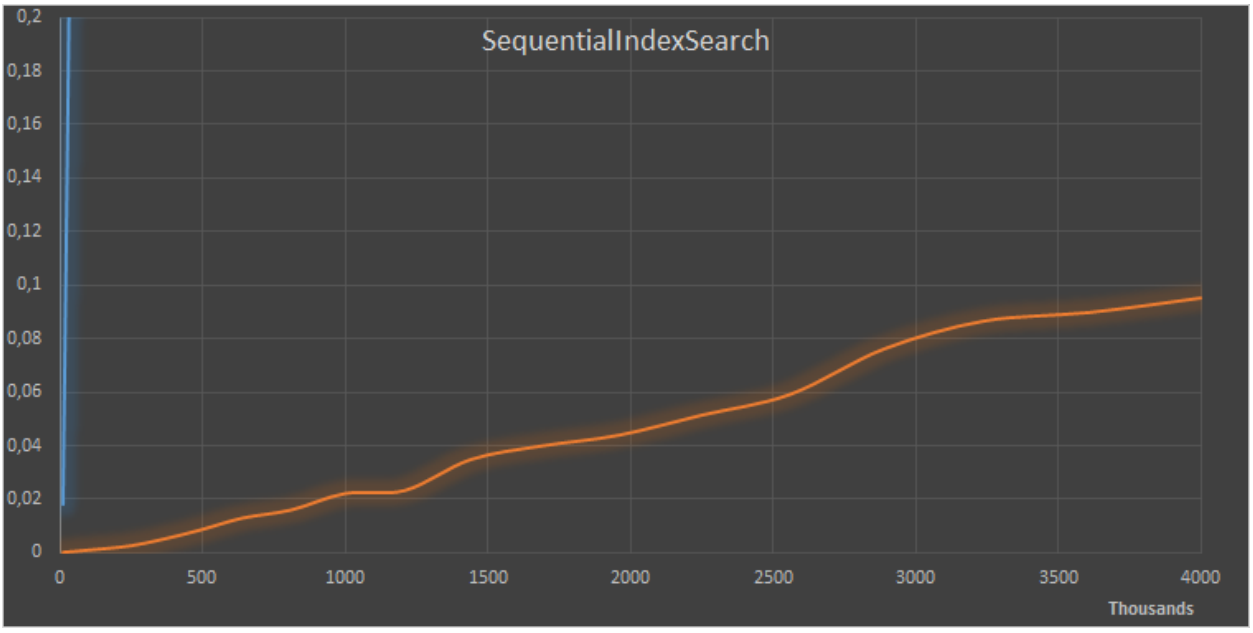
ArrayList går frem som en raskere liste i alle testene som kjøres. Dette ser man spesielt i de to overstående søkemetodene.

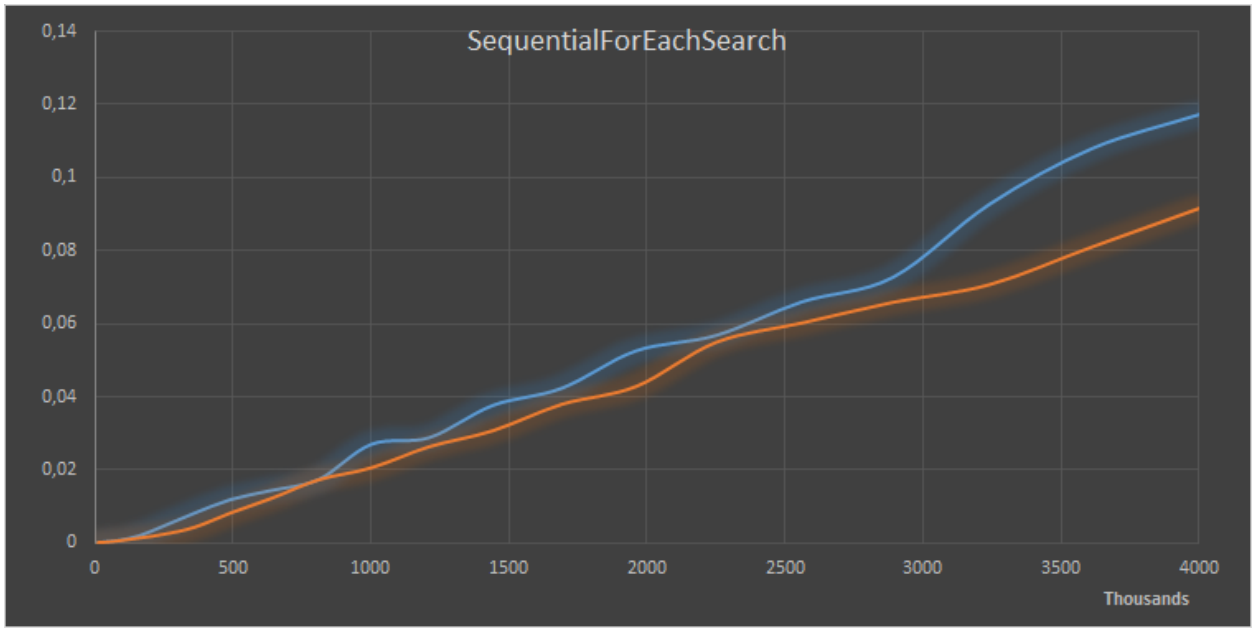
Grafiske fremstillinger

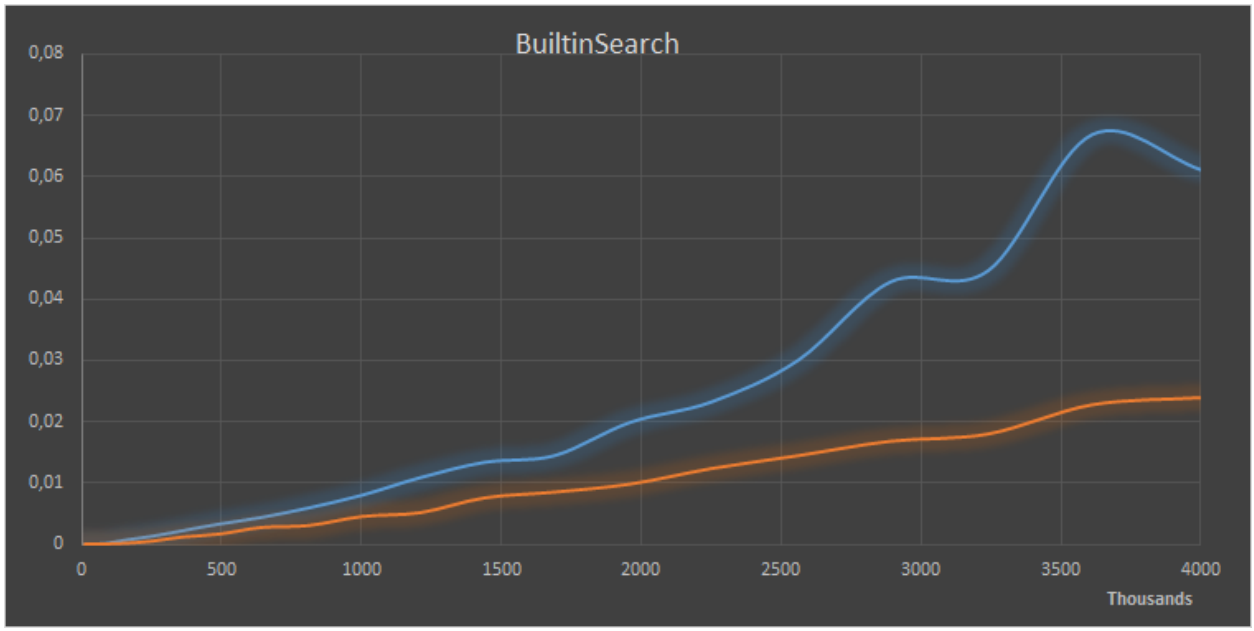
= LinkedList
= ArrayList

X-aksen er størrelsen på listene angitt i tusener, altså maksimum 4 millioner.
Y-aksen er søketid målt i sekunder med nanosekunders presisjon.









Som man ser ut ifra grafen er LinkedList ekstremt dårlig på SequentialForEachSearch, her er det derfor mest interessant å se på ArrayList sine resultater. Grunnen til denne forskjellen er beskrevet i oppgave 1.3

Deloppgave 3

BinarySearch hos ArrayList

Søkefunksjonen forutsetter at objektet implementerer grensesnittet Comparable, som lover at alle objekter av klassen kan sammenlignes som heltall. Det binære søket for ArrayList fungerer ved å direkte hente ut midten av listen, og sammenligne størrelsen av tallet med tallet vi leter etter. Denne sammenligningen bestemmer retningen for det videre søket, som fungerer på samme måte, men med nye avgrensninger. Denne er lynrask for ArrayList da man som regel ikke trenger å sjekke alle verdiene man ønsker å eliminere. Ved å se på dataene vi har samlet inn, kan man avgjøre at denne er logaritmisk.

BinarySearch for LinkedList

Det binære søket for LinkedList har samme forutsetninger som ved ArrayList. Den ytelsesmessige forskjellen oppstår fordi LinkedList må bla gjennom alle verdiene mellom start og den nye verdien den leter etter. Ved å se på dataene vi har samlet inn, kan man avgjøre at denne søketiden har lineær vekst.

SequentialIndexSearch for ArrayList

Det sekvensielle indekssøket for ArrayList henter ut en og en referanse fra listen og sjekker om den er lik objektet man søker etter. Dette er raskere hos ArrayList fordi den søker etter en og en plass i et array, hvor man ikke trenger å vite noe om objektet før eller etter indexen.Ved å se på dataene vi har samlet inn, kan man avgjøre at denne søketiden har lineær vekst.

SequentialIndexSearch for LinkedList

Den desidert tregeste søkefunksjonen er det sekvensielle indekssøket for LinkedList. Denne fungerer ved å før hente ut første referanse, og sjekke denne. Den henter så ut og sjekker andre referanse ved å hente første referanse. Videre, henter den og sjekker tredje referanse, ved å hente ut andre referanse, ved å hente ut første referanse. Denne forklaringen blir veldig tungvint for N referanse der N er et stort tall. Dessverre er selve søket like tungvint som forklaringen. For å sjekke frem til den millionte referansen, vil man måtte gjenta denne tungvinte prosessen for hvert enkelt ledd frem til denne referansen. Dette fremgår i grafen vår, hvor SequentialIndexSearch for LinkedList blir kuttet for N = 160.000 etter at tidsbruket ble ekstremt høyt for N = 90.000. Basert på kildene til grafen, kan vi si av firedobling av listestørrelse, blir søketiden 16 ganger lengre. Dette passer med påstanden 4^2=16, og vi kan derfor si at søkefunksjonen er kvadratisk.

SequentialForEachSearch for ArrayList

Det sekvensielle søket for hver referanse i en ArrayList fungerer ved å bruke iteratoren til listen. Iteratoren er et litt raskere sekvenssøk ettersom Iteratoren til en liste har direkte tilgang til

informasjonen listen holder. Dette betyr at, i motsetning til ved det sekvensielle indekssøket, blir uthenting av data gjort direkte mot arrayet. Ved å se på dataene vi har samlet inn, kan man avgjøre at denne har lineær vekst.

SequentialForEachSearch for LinkedList

Det sekvensielle søket for hver referanse i en LinkedList fungerer på så si samme måte hos ArrayList, bortsett fra at den begynner bakfra, og bruker en reversert iterator som spør en annen iterator om det motsattet av hva grensesnittet sier.
Da denne iteratoren holder styr på hvilken referanse peker til, er leteoperasjonen mye mer effektivt enn det sekvensielle indekssøket for denne listen, ettersom den ikke trenger å bla opp alle forgående referanser underveis. Ved å se på dataene vi har samlet inn, kan man avgjøre at denne har lineær vekst.

BuiltInSearch for ArrayList

Den innebyggede søkefunksjonen for ArrayList kjører et søk direkte på en posisjon i sitt array i en løkke. Det kan på mange måter sammenlignes med SequentialIndexSearch. Forskjellen er at SequentialIndexSearch bruker ArrayList sin get-metode for å hente ut verdier, mens den innebyggede søkefunksjonen kjører dette direkte mot arrayet. Dette er tregere enn å hente ut verdien direkte fra en plass i arrayet. Ved å se på dataene vi har samlet inn, kan man avgjøre at denne har lineær vekst.

BuiltInSearch for LinkedList

Den innebyggede søkefunksjonen for LinkedList fungerer ved å bla gjennom de lagrede nodene i listen fra første index(0). Denne fremstår som ytelsesmessig tregere enn den innebyggede søkefunksjonen i ArrayList da LinkedList sin søkefunksjon må først hente noder, før den henter objektet fra noden, og sjekker om dette objektet er det vi leter gjennom. Denne er derfor veldig lik, men mer effektiv enn, det sekvensielle søket gjennom hver referanse. Ved å se på dataene vi har samlet inn, kan man avgjøre at denne har linearitmetisk vekst.

Konklusjon

I de aller fleste tilfeller vil ArrayList være ytelsesmessig det beste valget, rent gjennomstnittsmessig er ArrayList et stykke forran LinkedList. LinkedList (Dette er ikke vist i grafene, men ut ifra hvordan linkedlist og arraylist fungerer) vil være raskere ved sletting og innleggelse av objekter i starten og midten av en liste, da den kun trenger å lagre nye referanser og det nye objektet. I disse tilfellene må en ArrayList flytte alle referanser på indekser bak midten ett steg. Dette kan ta lang tid, om det er snakk om mange objekter. ArrayList kan også kreve lengre tid for de enkelte nye innleggelsene hvor det er snakk om øking av størrelse til array(“inkrementell reallokering”).

ArrayList vil også øke sin egen størrelse med 50% når man nærmer seg å ha fylt opp plassene

det allerede har. Dette kan i enkelte sjeldne tilfeller føre til problemer med minne, da inkrementell vekst i ArrayList vil innebære at man på ett tidspunkt holder to arrays. Dette vil først å fremst være et problem på enheter som krever ekstremt lav latency for å gi en god brukeropplevelse(Les touch-enheter). Dette er grunnen til at ArrayList ikke er like forutsigbar som LinkedList i forhold til ytelse. Ved LinkedList vil man kanskje være et stykke bak rent gjennomsnittlig, men en ArrayList kan i teorien gi deg noen ytelses-hopp i en retning man ikke ønsker.

Oppgave 2

Felles for løsningene

Alle løsningene bruker et grensesnitt InputScanner, som krever at en implementerende klasse returnerer sannhetsverdien for om den fant ordet vi søker etter i klassen spesifisert. Vi valgte å gjøre klassen valgfri istedenfor å bruke en instans av In, ettersom dette ville resultert i flere gjennomganger for instansen av In ved gjennomsøking av URL, ettersom denne også må legge til eventuelle ubesøkte lenker.

Oppgave 2.1

Forutsetninger

Vi har brukt metoden readAllString() for en instans av klassen In, som splitter alle linjer på whitespace, slik at strenger som “Petter Northug” ikke blir søkbare, da vi først vil få strengen “Petter” og “Northug”. Dette kan selvfølgelig løses ved å enten implementere hukommelse i søkefunksjonen, eller ved å lese både neste og forrige streng sammen med nåværende streng i en for-each. Disse løsningene krever dessverre mer kompleksitet enn enklest mulig gjennomføring.

Rekursiv løsning

Den rekursive løsningen er tregere enn implementasjonene for kø og stakk ettersom den alltid behandler hver gjenstand som finnes i en mappe, uansett om dette er en mappe eller ikke. Dette antas å være tregere ettersom den må behandle en I/O-sjekk for å identifisere hver fil som en mappe flere ganger.

Løsning med bruk av stakk

Stakkløsningen virker utifra en begrenset test å være marginalt raskere. Denne henter alltid den siste filen som er oppdaget, og derfor vil en fjerning fra stacken bare innebære en koplingsendring, og denne settes til null hvis den allerede er satt til null. Denne løsningen er

derfor den løsningen som bruker mest tid på faktisk I/O-operasjon og begrenses derfor omtrent bare av at stakken må poppes og fillesing.

Løsning med bruk av kø

Køløsningen virker utifra en begrenset test å være relativt rask. Denne vil hente det objektet som først ble lagt til i køen. Fjerning av elementer for køen inkluderer en sjekk om den lenkede listen er tom, i tillegg til faktisk fjerning, og derfor vil denne for flere elementer være noe tregere enn en stakkløsning, ettersom det må avgjøres om neste element skal være null eller holde en referanse til et annet objekt i datastrukturen.

Oppgave 2.2

Forutsetninger

Som i oppgave 2.1, har vi brukt metoden readAllStrings(). Løsningen forutsetter også at eventuelle linker som skal besøkes er direkte lesbare, altså at de ikke åpnes ved hjelp av JavaScript for å servere selve linken eller at et eventuelt klikk åpner linken ved hjelp av JavaScript. Oppgaven søker ikke gjennom sider som har gitt en ikke-200 HTTP-kode.

Implementasjon

Vi har valgt å bruke en kø, da en rekursiv eller stakkløsning ville resultert i for lite bredde i søket. En køløsning gir den beste kombinasjonen av bredde og dybde, da den scanner hele siden for lenker og prioriterer de som blir oppdaget først.

Til motsetning ville en rekursiv løsning først oppdaget første lenken, fulgt denne, oppdaget en lenke, fulgt denne, og så videre. Dette ville resultert i et veldig snevert søk som ikke nødvendigvis ville kunne holdt seg på samme domene som måldomenet. En stakkløsning ville gjort det samme, men for siste lenken som ble oppdaget.

Vi har også valgt å lagre lenkene vi har oppdaget i et HashSet, da dette gir det desidert raskeste søket for contains().