Linked List och TreeSet; Prestanda på datastrukturer

André Frisk, Gustav Lindqvist, Erik Lundberg, Axel Tobieson Rova

***Abstract*— Med hjälp av tidigare erfarenheter i EDAA35 kursens laboration 5 har vi gjort en undersökning kring hur tiderna och prestandan kring datastrukturerna Linked List och TreeSet hur deras operationer kring insättning och sökning stämmer överens med den teoretiska faktan samt vilken datastruktur som presterar bäst. Det visade sig vara svårt att ta fram ett resultat kring om den specifika tidskomplexiteten stämde överens eller inte. Linked List presterade bäst vid insättning och TreeSet presterade bäst vid sökning.**

I. Introduktion

Under kursen gång av *EDAA35, Utvärdering av programvarusytem*, fick vi lära oss kring hur man analyserar ett program eller system statistiskt med programspråket R och metoder från kompendiet. I detta fall ska vi utföra ett eget experiment på utvalt område. Vår grupp har valt att undersöka datastrukturerna Linked List och TreeSet med inputen i form av Integer. Under detta experiment ska en undersökning utföras kring vilken av datastrukturerna som ger bäst prestanda, både i form av tid/throughput och hur konsekventa de är. Färdiga datastrukturer från Javas bibliotek kommer användas till experimentet.

II. Bakgrund

Detta experiment bygger väldigt mycket på laboration 5 i EDAA35 kursen vars syfte är att testa ett mindre system kring om en egenskriven sorteringsfunktion är bättre än den som finns i Collection.sort() i Java [1]. Linked List och TreeSet som nämnts ovan ska undersökas, där operationerna *add(E e)* och *contains(Object o)* är i fokus (se Javas API dokumentation för mer information). Linked List och TreeSet skiljer sig från varandra då TreeSet egentligen är ett Set (mängd) och inte tillåter dubletter som indata och inte bevarar insättningsordningen för element men är sorterade, medan Linked List är helt tvärtemot som behåller dubletter och insättningsordningen och inte sorterar vid insättning [3]. Operationerna add och contains som ska undersökas är som namnet säger en operation för att lägga till ett objekt i listan och för att se om objektet finns i listan eller inte.

Resultatet bygger mycket på tidskomplexiteten, som är en enhet som kan ge en uppskattning på hur lång tid operationen ska ta, vilket skiljer sig åt beroende på mängden indata [2]. Då TreeSet är av typen Tree datastrukturen behöver trädet iterera igenom sitt träd för att lägga till ett objekt, därför har operationerna add och contains i TreeSet tidskomplexiteten O(log(n)). Linked List operationen add har tidskomplexitet O(1) då den sätter in elementet på en position direkt medan operationen contains har tidskomplexiteten O(n) då den måste iterera igenom hela listan tills den hittar det givna elementet/indexet.

Under laborationen ska JIT undersökas. JIT står för "Just-In-Time"-kompilering vilket betyder att JIT kompilatorn körs efter att programmet har startats och kompilerar koden (bytekod) på vägen vilket oftast gör att programmet körs snabbare [1]. En traditionell kompilator kompilerar all kod till maskinkod innan programmet körs.

I den statistiska delen av experimentet används medelvärde och T-test. T-test är ett test där man testar om det finns en signifikant skillnad mellan de underliggande medelvärdena mellan två talserier [1].

III. Frågeställning

- Hur väl överstämmer mätningarna med det teoretiska resultatet vid tilläggning?
- Hur väl överstämmer mätningarna med det teoretiska resultatet vid sökning?
- Blir resultatet utan JIT-kompilator likt det teoretiska i jämförelse med utan JIT?
- Vilken datastruktur presterar bäst vid given indata?

IV. Metod

Datastrukturera Linked List och TreeSet importeras från Javas standardbibliotek till det egenskrivan programmet som ska iterera genom olika filer av indata och därefter skriva genomsnittstiden i nanosekunder till ett dokument som används till analysen. Indatan till programmet är tre olika txt-filer med [1..n] element indata där n är mängden objekt i filnamnet. Dessa tre filer innehåller tal med 1000, 10 000 respektive 100 000 där talen är slumpade mellan 1 -> n. Filerna är inte sorterade. Dessa tal genererades av ett mindre egenskrivet program i Java där biliboteket Math och BufferedWriter användes.

Den statistiska delen undersöktes i programspråket R där programmet fick köras 30 gånger var med varje indata fil, där även JIT-kompilatorn stängs av under ett test. Här utförs medelvärde på medelvärdena för att få en bra genomsnittlig tid för operationen. Ett T-test utfördes på varje resultat för att se om det finns någon signifikant skillnad.

V. Resultat

Datan som samlades in från R finns i Appendix B. Graferna i Appendix C är plottade för enbart 10 000 tal som indata för enklare läsbarhet och analys. Datan är uttryckt i nanosekunder och insvängningsförloppet samt jämviksområdet har inte tagits hänsyn till resultatet. T-test testades på medelvärdena och p-värdet från testet gav det minsta möjliga talet som finns i R, vilket tyder på att medelvärdena inte är i närheten nära varandra.

VI. Diskussion

Datan i Appendix B visar väldigt olika resultat kring insättning och sökning i Linked List och TreeSet. Fokus kommer ligga på den nedre delen av tabellen där resultatet är uttryckt i medelvärdet av medelvärdena per operation (Mean tid/n sektionen) där detta medelvärde av medelvärdena delad med mängden indata för att få en ungefärlig summa kring hur långt tid operationen tar för given indata. Notera att för JIT undersöktes enbart 1000 och 10 000 då indata på 100 000 tog alldeles för långt tid för att exekvera.

Vid analys av insättning på Linked List och TreeSet så är Linked List snabbast, vilket stämmer teoretiskt sätt då Linked List operationen add har tidskomplexitet på O(1) vilket i princip ska vara konstant medan TreeSet har O(log(n)) som är snäppet långsammare. Vid högre indata verkar båda datastrukturerna ha mindre genomsnittstid per operation, detta kan bero på att programmet har körts mer då och varje operation tar mindre tid då programmet lär sig. Linked List insättningen går så lågt som 18.40 ns enligt vårt testfall vilket är väldigt snabbt i jämförelse till TreeSet vars snabbaste tid var 218 ns (vid 100 000 tal som indata). Detta är en större skillnad statistiskt och vid större system som har strikta deadlines men i vårt fall märks det inte av. Vid avstängning av JIT-kompilatorn blir givetvis tiden mycket högre och tiden växer och blir längre vid högra indata där Linked List ger ut 983 ns och TreeSet ger 3757,77 ns (vid 10 000 tal som indata). I detta fall visar det sig att Linked List fortfarande visar bättre prestanda i add med eller utan JIT vilket stämmer teoretiskt sätt enligt tidskomplexiteterna. Indata på 1000 och 10 000 tar givetvis längre tid då programmet inte har optimerat sig men även här har Linked List bättre tid än TreeSet. Varför Linked List tar så pass lite tid är att insättning i Linked List alltid sker i slutet av listan vilket är en kostant operation då den inte behöver iterera genom listan. TreeSet är däremot sorterad och måste iterera genom sitt träd vilket gör att trädet måste hitta rätt noder att sätta in elementet i rätt nod vilket resulterar i högre tid då den ständigt måste kolla efter rätt nod/löv.

Vid sökning där Linked List har tidskomplexiteten O(n) och TreeSet har O(log(n)) ska teoretiskt sätt TreeSet ge bäst prestanda med och utan JIT trots indatans storlek. Detta visar sig stämma även här. Med JIT har Linked List sökning med 100 000 tal som indata en tid på 19,7 ms medan TreeSet har en tid på 217,19 ns. Utan JIT ger Linked List med 10 000 tal som indata en tid på 56,4 ms medan TreeSet ger 3758 ns. TreeSet har även bättre tid än Linked List vid lägre storlek på indata. Varför resultatet skiljer sig extremt mycket beror på tidskomplexiteterna. O(log(n)) är mer effektiv än O(n) vilket gör att vid större indata kommer Linked List att köras väldigt länge då den måste gå igenom hela listan och söka medan TreeSet sorterar sin egna lista vilket gör att tiden för sökning blir relativt kort. Hade större N (loopar av programmet när det körs) tagits i detta fall vid sökning och med 100 000 tal som indata utan JIT hade programmet förmodligen tagit timmar.

Vid en snabb analys av medianresultatet i tabellen per operation stämmer slutsatserna ovan. Linked List har bättre prestanda och tid vid insättning men TreeSet har bättre prestanda och tid vid sökning.

Jonas Skeppstedt har en tabell i sin bok *Algoritmer: en kortfattad introduktion* på sida 19. Tabellen visar genomsnittlig tid för olika storlekar på indata och tidskomplexiteter [2]. Dessvärre finns inte O(log(n)) med, men den är snabbare än O(n) men långsammare än O(1). Dessa tider beror helt på processorn med men tabellen togs fram med en 4 GHz CPU. Enligt Skeppstedts tabell ska en O(n) med inputdata på 1000, 10 000 och 100 000 ta 2500 ns, 25 000 ns och 250 000 ns. Enligt tabellen i Appendix B är det 3600 ns, 44 040 ns och 197 000 ns. I jämförelse med Skeppstedts tabell stämmer inte dessa värdena överens med varandra. Men det kan bero på processorn och hur många gånger programmet får köras innan så exekveringstiden sjunker. Tabellen hade ingen information om O(1) eller O(log(n)). Utan JIT är värdena inte i närheten av tabellens värden.

Något vi hade kunnat göra annorlunda för att få ett mer pålitligt resultat är att ta hänsyn till insvängningsförloppet samt jämviktsområdet. Om uträkning i jämviktsområdet av medelvärdena görs kommer resultatet vara mer konstant då javas minnesmodell lär sig varje gång programmet exekveras och tiden kortas ner. Nu används data från de första körningarna vilket kommer skapa spikar i medelvärdet som kommer ge ett högre medelvärde än vad som egentligen borde användas. Programmet hade dessutom kunnat köras mer än 30 gånger i R-scriptet då det kommer ge ut mer precision i datan. Vid högre förlopp tar programmet väldigt långt tid att exekvera (uppemot timmar) vilket gör att undersökningen ej är optimal för våra datorer. Fler grafer hade kunnat tas ut vid 1000 och 100 000 indata för bättre grafisk jämförelse av resultatet.

VII. Slutsats

Att mäta tid, prestanda och att ange tidskomplexitet är ett svårt ämne. Med detta experiment har det visat sig vara svårt att få fram teoretisk fakta kring om tidskomplexiteterna stämmer eller inte för våra datastrukturer och deras operationer. Även med hjälp av Skeppstedts tabell och Google blev det ingen rak slutsats på om mätningarna överstämmer med tidskomplexiteten. Dock blev det teoretiska resultatet korrekt gällande att de olika operationernas tidskomplexiteter stämmer överens i form av vilken som ska ta mest tid. Vi hade tidskomplexiteterna O(1), O(log(n)) och O(n) hos operationerna. Denna ordning från vänster till höger ger snabbast tidskomplexitet, vilket överensstämmer med resultatet. Linked Lists instättning med tidskomplexitet O(1) visade sig vara mycket snabbare än TreeSets insättning med tidskomplexitet O(log(n)). Linked Lists sökning med tidskomplexiteten O(n) var betydligt långsammare än TreeSets sökning med tidskomplexiteten O(log(n)). Alltså visade det sig att Linked Lists insättning och TreeSets sökning hade bäst tid och prestanda. Trots olika mängder indata var det en tydlig skillnad kring den totala tiden för varje operation per datastruktur. Testerna som undersökte utan JIT-kompilering går inte riktigt att validera hurvida det är likt det teoretiska

värdet eller inte. Vi kan dock hävda att tiden det tar att köra utan JIT är mycket högre och tiden stämmer högst sannolikt inte in på det teoretiska resultatet.

REFERENCES

[1] Martin Höst. EDAA35 Utvärdering av programvarusystem. Institutionen för Datorvetenskap, Lunds Tekniska Högskola, 2021.

[2] Jonas Skeppstedt. Algoritmer: en kortfattad introduktion. Skeppberg AB, 2020

[3] Elliot B. Koffman and Paul A. T. Wolfgang. Data Structures: Abstraction and Design Using Java, Third Edition. Wiley, Temple University, 2016.

APPENDIX

A. *Sökträng*

Till projektet skulle vi använda oss av SCOPUS databasen för att eventuelt hitta något relaterbart arbete till projektet. Vi använde oss av sökströngen TITLE-ABS-KEY (time-complexity). Tyvärr hittades inget användbart för oss men det som hittades hade väldigt mycket med algoritmer och inbyggda system att göra då mycket diskuterades krign processorer och plattformar. Sökströngen gav resultat bak till 1990-talet.

B. *Data*

Mean tid/n	1000	10000	100000	1000 utan JIT	10000 utan JIT
List insättning	264790	919507	1840153	1019400	9826217
List sökning	3599930	440402333	1.97E+10	69219540	5.64E+09
Tree Insättning	1242017	3592780	30011170	4385857	47609183
Tree sökning	349160	1724357	21719563	3125547	37577763
Median tid/n	1000	10000	100000	1000 utan JIT	10000 utan JIT
List insättning	125050	642600	1080050	886700	9481100
List sökning	2966100	436532350	1.24E+10	69146500	5.60E+09
Tree Insättning	358850	2727150	26386600	4095150	46684800
Tree sökning	194100	1464050	21265450	2897200	36722650
Delar på N för att få ut tid för varje					
Mean tid/n	1000	10000	100000	1000 utan JIT	10000 utan JIT
List insättning	264.79	91.9507	18.40153	1019.4	982.6217
List sökning	3599.93	44040.2333	1.97E+05	69219.54	5.64E+05
Tree Insättning	1242.017	359.278	300.1117	4385.857	4760.9183
Tree sökning	349.16	172.4357	217.19563	3125.547	3757.7763
Median tid/n	1000	10000	100000	1000 utan JIT	10000 utan JIT
List insättning	125.05	64.26	10.8005	886.7	948.11
List sökning	2966.1	43653.235	1.24E+05	69146.5	5.60E+05
Tree Insättning	358.85	272.715	263.866	4095.15	4668.48
Tree sökning	194.1	146.405	212.6545	2897.2	3672.265

Fig. 1. Insamlad data för datastrukturerna uttyckt i nanosekunder

C. *Grafer*

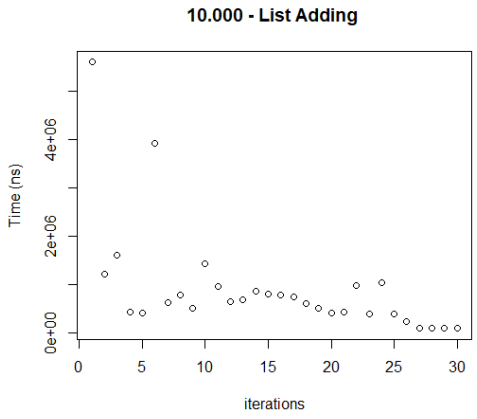

Fig. 2. Insättning i Linked List, 10 000 tal

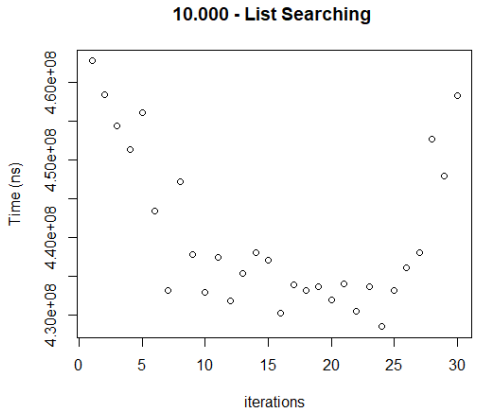

Fig. 3. Sökning i Linked List, 10 000 tal

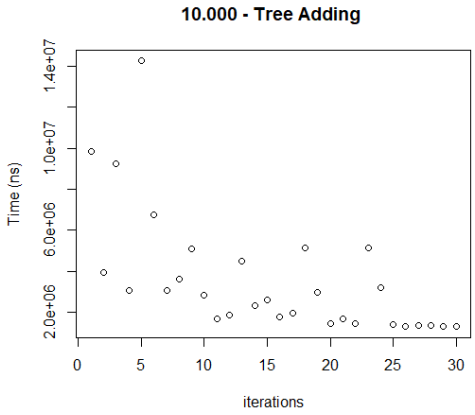

Fig. 4. Insättning i TreeSet, 10 000 tal

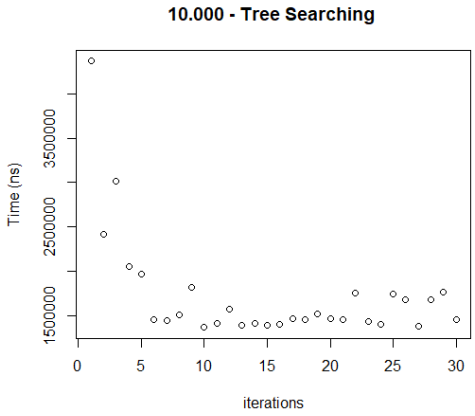

Fig. 5. Sökning i TreeSet, 10 000 tal

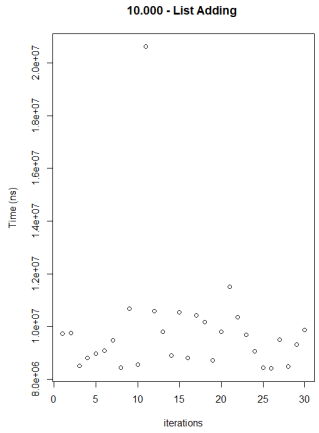

Fig. 6. Insättning i Linked List utan JIT, 10 000 tal

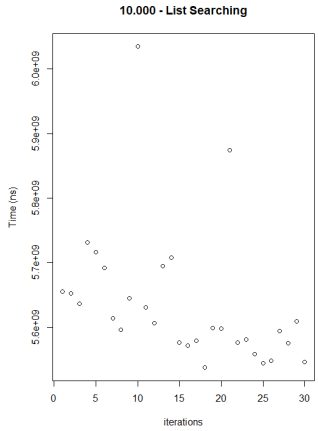

Fig. 7. Sökning i Linked List utan JIT, 10 000 tal

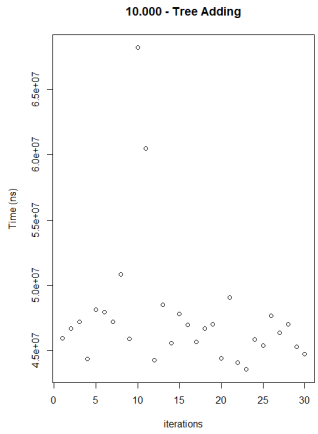

Fig. 8. Insättning i TreeSet utan JIT, 10 000 tal

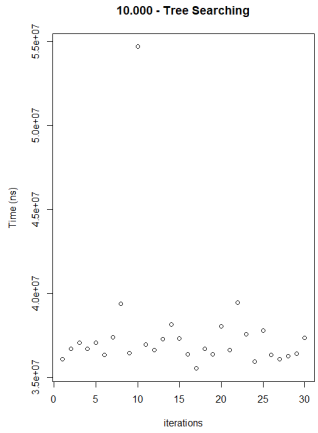

Fig. 9. Sökning i TreeSet utan JIT, 10 000 tal