Mätning av körtid på Javas List interface metoder för sortering, sökning, insättning och borttagning

Johannes Gerding, Dennis Nilsson
och Sofia Tatidis

***Abstract*— This project aims to discover which data structures are most suitable for the following four operations; searching, sorting, inserting and removing.**

`LinkedList` , `ArrayList` , `Stack` of the Java Collection library and the self-implemented singly linked list, `SLL` , were chosen for the tests as they all implement Java's List interface.

As a result of our experiments we found that sorting a list takes approximately the same time for each of the classes with the exception of `SLL` which is three orders slower. Search is quicker for linked lists compared to indexed lists. Linked structures are suitable for insertion and deletion at the start of a list, but slower at removing in the middle of a list. `LinkedList` , a doubly linked list, is time efficient at removing elements at the end of a list while `SLL` , a singly linked list, is slower at removing elements at the end of a list. Indexed structures are quicker at removing at the middle and end of a list, but slower at insertion at the start of a list.

I. INTRODUKTION

Exekveringstiderna av olika operationer på en lista påverkas av vilken datastruktur som används för att implementera listan. I detta projekt undersöks vilka datastrukturer som lämpar sig för sökning, sortering, insättning och borttagning. De listor som testats är Javas `ArrayList` , `LinkedList` och `Stack` som alla implementerar Javas List-interface. Även en egen implementation av en enkellänkad lista, `SLL` , har testats. Även `SLL` implementerar Javas List-interface.

I ett större perspektiv har valet av datastruktur påvekan på exekveringstiden av ett helt system och därmed en hel organisations resursförbrukning.

II. TEORI OCH TIDIGARE FORSKNING

A. Länkade listor

I en länkad datastruktur består listan av noder som har en referens till efterföljande nod, och även föregående nod i fallet av en dubellänkad lista. `SLL` är en enkellänkad lista och har en referens till första noden, och `LinkedList` är en dubbellänkad lista som har referenser till både första och sista noden. Metoder som använder de noder med en referens till sig har konstant tidskomplexitet. Traversering har linjär tidskomplexitet[1].

B. ArrayList

En `ArrayList` består av element där varje element tilldelas ett index beroende på var i listan de befinner sig. Tack vare listans indexerbarhet är det lätt att referera till ett element. Till skillnad från länkade listor krävs därför ingen traversering för att hitta element vid det valda indexet. Detta gör att operationer som sökning av element går snabbt och har konstant tidskomplexitet. Operationer som involverar att lägga till eller ta bort element på något index i listan, förutom sist, har en linjär tidskomplexitet eftersom varje element efter det som påverkas måste flyttas ett steg fram eller bak för att bibehålla rätt indexering[1].

C. Stack

Javas Collections klass `Stack` förlänger klassen `Vector` men med begränsningar på vilka operationer som får utföras. `Stack` en är en representation av en last-in-first-out-`Stack` . Vanligtvis innebär det att endast elementet "högst upp" i högen får plockas bort/läggas till. Men när List-interfacets operationer används kan detta kringgås. Då behandlas `Stack` en som en Vector och får därav samma tidskomplexitet för operationer [3].

Tidskomplexiteten för operationer på en `Vector` är i stort sett samma som för `ArrayList` . Undantaget är att insättning och borttagning går smidigare då `Vector` kan ändra stolek [4].

D. Tidigare forskning

Den första studien vi tog inspiration av är "A HEURISTIC APPROACH OF SORTING USING LINKED LIST" av Shruti Rishabh Pandey. Studien handlar om sortering, dock endast med länkade listor. Här kollar vi främst på stukturen av metoden, samt hur resultaten presenteras. "Related Work" användes även för att få en överblick av sorteringsalgoritmerna och förstå hur mergesort, vilken `javas.Collections.sort()` använder sig av, fungerar [7].

Den andra studien som användes var "AN EMPIRICAL STUDY OF ALGORITHMS PERFORMANCE IN IMPLEMENTATIONS OF SET IN JAVA." av Danjel Kucak, Goran Djambic och Bojan Fulanovic. Denna studie testade olika operationer som `insert()`, `remove()`, `contains()` på olika implementationer av `Set`. Denna studie är relaterad till vårt arbete, då vi i princip gör samma tester fast för olika implemenationer av `List` istället för `Set`. Även här användes studien för att förstå vad vi bör mäta samt hur vi effektivt kan presentera detta[2].

III. FRÅGESTÄLLNING

- Vilka List-implementationer lämpar sig bäst för sortering beroende av vilken datastruktur som listan implementerar?
- Vilka List-implementationer lämpar sig bäst för insättning i början av en lista beroende av vilken datastruktur som listan implementerar?
- Vilka List-implementationer lämpar sig bäst för borttagning i början, mitten och slutet av en lista beroende av vilken datastruktur som listan implementerar?
- Vilka List-implementationer lämpar sig bäst för bestämda element som existerar respektive inte existerar, i en sorterad lista för de olika datastrukturerna?

IV. METOD

En javaklass skapades för att mäta exekveringstiderna för alla klasser och respektive operation 600 gånger och skriver varje körnings medelvärde till en fil (efter körning 300 för att ackomodera för insvängningsperioden). En R-funktion körde java-klassen 100 gånger och lade över medelvärden för varje körning i en lista. T-test genomfördes på respektive lista av medelvärden och gav medelvärde av medelvärden, konfidensintervall samt p-värde. Experimentet utfördes på ett Windows 64-system med OpenJDK version 11.0.9.1.

A. Sortering

Sorteringen genomfördes med List-interfacets `sort()`-metod på två olika indatan bestående av 800 slumptal i storleksordning $0-10000$. Ena indatan var delvis sorterad, medan den andra var helt slumpad.

B. Sökning

Sökningen genomfördes med List-interfacests `indexOf()`-metod på en lista av 800 heltal sorterade i fallande ordning. Exekveringstiderna för sökning efter tre element mättes. Element A var det första elementet i listan, Element B ett existerande element i mitten av listan och Element C var ett icke-existerande element.

C. Insättning

Insättning av 800 heltal skedde med List-interfacests `add()`-metod på tom lista. Exekveringstiden för insättning av elementen i listans början mättes tills listan innehöll 800 tal.

D. Borttagning

Borttagning genomfördes med List-interfacests `remove()`-metod på listor med 800 heltal. Exekveringstiden mättes för borttagning av tre element. Ett element på index 0, ett på index 400 och ett på sista index.

V. RESULTAT

Enligt figur 1, som visar resultat för sortering, kan det konstateras att medelvärdet är avvikande högt för `SLL` under delvis sorterad och osorterad data, jämfört med resterande klasser. Dessutom visar figuren det generella mönstret av lägre medelvärde för delvis sorterad data över alla klasser.

För figur 2, som visar resultat för sökning, ses en låg skillnad mellan klasserna för element A och B, till skillnad från icke-existerande element C som skiljer sig mot resterande elementtyper. Den senare kan ses med högre medelvärde för `ArrayList` och `Stack` , och lägre för resterande.

Figur 3, som visar resultat för insättning, visar en tydlig skillnad mellan länkade och icke-länkade datastrukturer, uppskattningsvis en faktor 1000 större. Det kan även noteras att `SLL` erhöll ett lågt medelvärde jämfört med `LinkedList` i samma graf.

I figur 4, som visar resultat för borttagning, ses en trend av nedåtgående medelvärde från första till sista elementplats för de ickelänkade datastrukturerna `ArrayList` och `Stack`. För `LinkedList` kan det konstateras ett lågt medelvärde för första och sista element gentemot mellersta element. Klassen `SLL` har en uppåtgående trend av medelvärde med lägsta resultat för första element. Följaktligen ses det från figur 5 att metoden `pop()`, speciell för `Stack` , erhöll snarlikt medelvärde för det av `ArrayList` på första elementplats.

För alla mätuppställningar, under de olika förhållandena, medförde ett t-test av ett medelvärde ett signifikant resultat (störst funna $p < 1.56 \cdot 10^{-15}$).

VI. DISSKUSION

A. Sortering

Det finns några potentiella anledningar till att den egenimplementerade `SLL` var långsammare än de andra listorna på att sortera 800 element. Vi antar att vår egenimplementerade `toArray()`-metod är ineffektiv med hög tidskomplexitet. Under tiden vi gjorde våra tester ändrade vi `toArray()`-funktionen några gånger, vilket påverkade tiden sorteringen tog. Iteratorfunktionerna implementerade vi delvis själva med några importer från `AbstractList`[5]. Likt `toArray()`-funktionen kanske dessa kan göras mer effektiva med lägre tidskomplexitet. I sin tur skulle det kunna påverka sorteringstiden, om det är så att sorteringsalgoritmen använder sig av någon av dessa iteratorfunktioner.

De andra tre listorna var ungefär lika snabba på att sortera en mängd med tal. P.g.a att `LinkedList` ej är indexerbar var förväntan att sortering i denna lista skulle ta längre tid eftersom många traversingar behöver göras för att hitta rätt element. I vår undersökning visade det sig att `LinkedList` inte var långsammare. Detta kanske inte är fallet om en större datamängd skulle sorteras. Då skulle traverseringarna ta längre tid och `LinkedList` skulle kunna bli långsammare än de icke-länkade strukturerna[6].

B. Sökning

Sökning innebär jämförelse av element och möjlig traversering av listan. Det sker linjärt för alla klasser[1]. Enligt teorin II-A gäller linjär tidkomplexitet för bägge datastrukturer vid traversering. Följaktligen bör medelvärdena teoretiskt sett närma sig varandra för element C mellan klasserna då liststorleken växer godtyckligt[1].

Motsägelsen mellan teori och resultat tyder på att den tidigare konstaterat signifikanta skillnaden i medelvärde som ses för element B och icke-existerande C i figur 2 troligen utgörs av skillnaden mellan hur klasserna går igenom och jämför elementen i sökning. Trenden som ses pekar på att tiden det tar för jämförelse och traversering ökar snabbare gentemot liststorlek för de icke-länkade datastrukturerna än för de länkade under varaiationen av de elementpositioner som presenterats.

Vad som blir den mest prominenta skillnaden är den för datastrukturens sätt att traversera, och därför även vad som i huvudsak kan ha resulterat i utfallet av sökningen. En länkad datastruktur går igenom noderna och hämtar element att jämföra med allt eftersom noderna traverseras, medan en indexerbar datastruktur går igenom de index som begränsas av listans storlek och hämtar element att jämföra genom indexering direkt i listan[1]. Resultaten tyder på att indexerbara klasser tar längre tid att genomsöka helt än länkade klasser, för liststorleken vi testade, trots dess motsägelse av teorin.

Det verkar gynnsamt att bruka en datastruktur skapt för ändamålet; i fallet av sökning i listor med samma storlekordning som vår presterar indexerbara datastrukturer sämre än länkade datastrukturer, för element B och C, och likvärdigt för element A.

C. Insättning

Enligt resultaten givna i figur 3 så är det de länkade strukturerna som genomför insättning i början av listan snabbast.

Detta är i enighet med den teori som presenterades i II-A. De länkade listorna som undersöks, `LinkedList` och `SLL`, har båda referenser till den första noden och på så vis även första elementet. Det går därför snabbt att lägga till noder och element i början av listan.

`ArrayList` och `Stack` implementeras båda av indexerbara datastrukturer. När ett element läggs till i en sådan datastruktur måste resterande element flyttas så att indexeringen blir rätt, som nämns i II-B. Jämfört med de länkade strukturerna som bara genomför en omkoppling av noder så tar det längre tid för indexerbara strukturer att lägga till element.

Våra resultat visar även att den egenimplementerade `SLL` var ungefär en femtedel snabbare än `LinkedList`. Det skulle kunna bero på att `SLL` har ett tydligt specialfall där insättning i början av listan sker direkt.

D. Borttagning

Likt insättning sker borttagning snabbare med hjälp av länkade listor när elementet ligger i början av listan. På samma sätt som insättning är smidig i början av länkade listor är även borttagning det. För indexerbara strukturer gäller att hela listan ändras när borttagning sker i början, vilket tar längre tid.

När elementet som skall tas bort ligger i mitten traverseras länkade listor först innan borttagning kan ske. Traverseringen står för merparten av tidsåtgången. `ArrayList` och `Stack` utnyttjar indexering istället för traversering för att finna det element som skall tas bort. Detta sparar tid jämfört med länkade listor. När element tas bort behöver listans storlek inte ökas och det räcker med att resterande lista flyttas till rätt index, vilket tar kortare tid ju längre bort i listan som elementet ligger.

Därför är både `ArrayList` och `Stack` snabbast på att ta bort element i slutet av listan. `LinkedList` har en referens till sista noden eftersom det är en dubbellänkad lista. Genom denna referens till sista noden går det snabbt att ta bort sista elementet. `SLL` saknar referens till sista elementet. Borttagning i slutet kräver därför att hela listan traverseras, vilket förklarar att det tar längst tid för `SLL` att genomföra borttagning av sista elementet.

E. Validitet

I den kontext som innefattar körtidspåverkan på organisationers system kan testsituationen till stor del överföras till verkligheten, men fallerar att innefatta större datamängder, vilket görs tydligt av de till viss del motsägelsefulla resultaten gentemot teorin. En inklusion av stegvis växande datamängder hade gett en tydligare bild av testsituationen och högre extern validitet.

Under de relativt konstanta förhållande som testen utförts kan man till stor grad utesluta alternativa förklaringar, bidragande till god intern validitet. Andra program än de för mätuppställningen har inte körts parallellt, men möjliga bakgrundsprocesser har inte kontrollerats på samma vis. Eftersom mätuppställningen tar hänsyn till JIT kompilering och dess insvängsningsförlopp reduceras här risken för kortvarig intern påverkan av bakgrundsprocesser och inkonsekventa eller varierande resultat.

Överlag är det valda måttet, medelvärde av körtid och dess signifikans, rimligt och lämpligt för att mäta det avsedda, vilket påvisar rapportens begreppsvaliditet.

En möjlig felkälla är den egenimplementerade listklassen `SLL` och de metoder som anropas av `collections.sort()`, förmodligen `toArray()`. Det kan fastställas att egen implementering innebär risk, speciellt för möjliga fel i kod som ofta anropas i den metod som mäts. Denna risk kan t.ex. minimeras genom implementering av interface och mer omfattande felsökning och testning. Det bör nämnas att `JUnit`-tester har skapats för samtliga metoder i listklassen. Svårigheter finns således i att implementera klasser som är tänkta att fungera med metoder tillhörande standardbiblioteket, speciellt om dess komplexitet är hög.

VII. Slutsats och fortsatt arbete

A. Sortering

Slutsatserna vi kan dra från våra resultat angående sortering är att alla listorna, förutom vår egen implementerade `SLL`, var nästintill lika snabba. Generellt sätt var sorteringen av den nästan sorterade listan av tal snabbare i alla listor, men skillnaden var inte lika stor i `SLL` som i de andra listorna.

B. Sökning

Resultaten för sökning implicerar att de länkade listorna är mer effektiva för sökning, gentemot de icke-länkade, av listor för storleken vi testat i denna rapport, och förmodligen inte för listor av större storlek. Söktid för elementen korresponderar med dess plats i listan, eller listans hela traversering, för alla klasser.

C. Insättning

Både `LinkedList` och `SLL` lämpar sig för insättning i början av listan. Detta för att båda är länkade strukturer med referens till första elementet.

D. Borttagning

När borttagning sker i början av listan lämpar sig `LinkedList` och `SLL` bäst eftersom dessa är länkade strukturer med referens till första elementet. Borttagning i mitten av listan går snabbare med `ArrayList` och `Stack` eftersom dessa i grunden är indexerbara strukturer och det ej behövs traverseras genom listan för att hitta det mittersta elementet. Borttagning av sista elementet lämpar sig också bra med `ArrayList` och `Stack` tack vare dess indexerbarhet. Dessutom behöver färre element flyttas till sitt nya index när borttagningen sker längre bak i listan. Även LinkedList är effektiv i borttagning av sista elementet eftersom listan har en referens till det sista elementet och inte behöver traverseras.

E. Fortsatt forskning

Våra resultat visar att sortering av data kan ta lång tid jämfört med de andra operationerna. En intressant fortsatt undersökning är att skifta fokus till sorteringsalgoritmer och utreda vilken sorteringsalgoritm som lämpar sig bättre för vilken typ av data.

En utveckling av detta skulle vara att kartlägga hur data bäst ska hanteras för att underlätta de olika operationerna. T.ex. fallande/stigande ordning.

I verkligheten förekommer data i olika storleksordning, jämför exempelvis en databas över hela sveriges befolkning med en lista av alla färger en t-tröja kan ha i en webbshop. En fortsatt undersökning är att undersöka hur storleken på datamängden påverkar hur bra operationerna ter sig på de olika datastrukturerna.

APPENDIX

F. Figurer

Se kommande sidor.

REFERENCES

[1] Elliot B. Koffman and Paul A. T. Wolfgang. *Data Structures: Abstraction and Design Using Java*, 2nd Edition. John Wiley Sons, Inc, sid. 65, 75, 88, 90-98, 2010.

[2] KUCAK, D[anijel]; DJAMBIC, G[oran] FULANOVIC, B[ojan], "AN EMPIRICAL STUDY OF ALGORITHMS PERFORMANCE IN IMPLEMENTATIONS OF SET IN JAVA", Annals Proceedings of DAAAM International, 2012.

[3] Oracle. Java Documentation *Class Stack <E>*. https://docs.oracle.com/javase/7/docs/api/java/util/ Stack .html, hämtad 2021-05-17.

[4] Oracle, Java Documentation *Class Vector<E>*, https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html. hämtad 2021-05-17.

[5] Oracle, Java Documentation *Class AbstractList<E>*, https://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html. hämtad 2021-05-17.

[6] Oracle, Java Documentation *Class LinkedList<E>* https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html. hämtad 2021-05-17.

[7] S. R. Pandey, "A HEURISTIC APPROACH OF SORTING USING LINKED LIST," 2018 Second International Conference on Computing Methodologies and Communication (ICCMC), doi: 10.1109/ICCMC.2018.8487780, sid. 446-450, 2018

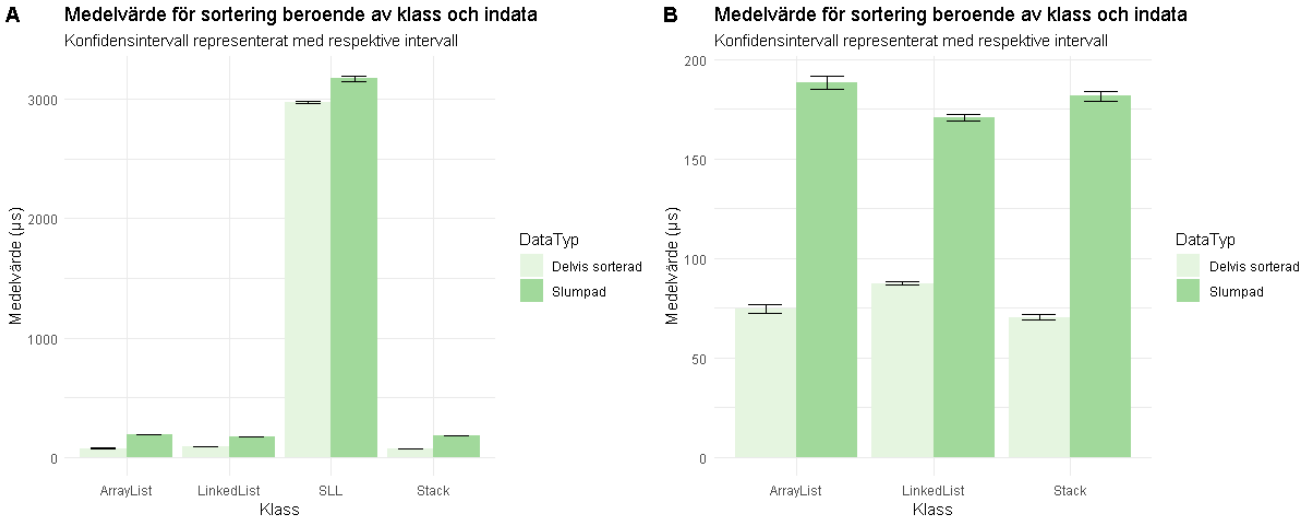

Fig. 1. Medelvärde för sortering med `Collections sort()`, och konfidensintervall av 100 iterationer, beroende av klass och indata av 800 element, som är delvis sorterad eller osorterad

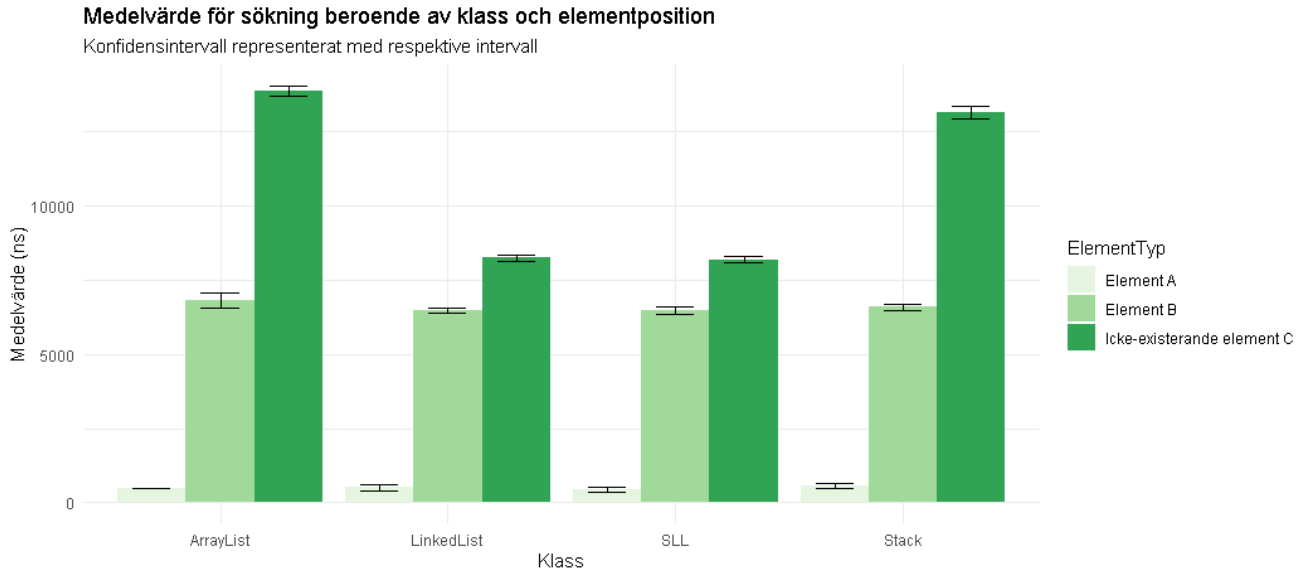

Fig. 2. Medelvärde för sökning av element, och konfidensintervall av 100 iterationer, beroende av klass och elementposition, som är första, mellersta eller icke-existerande element, av totalt 800 element

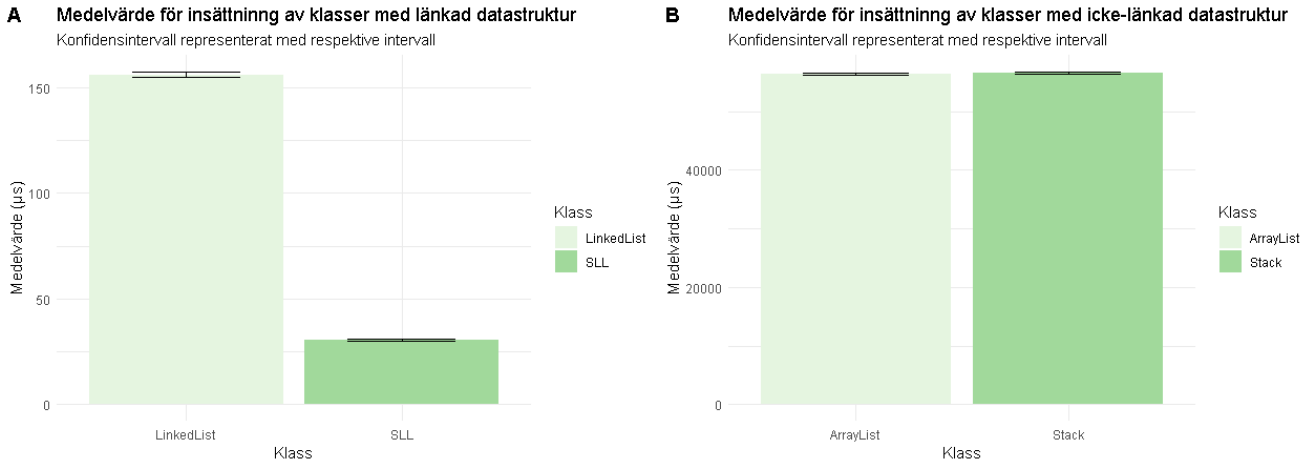

Fig. 3. Medelvärde för insättning på första plats, och konfidensintervall av 100 iterationer, beroende av klass, och uppdelad enligt länkad och icke-länkad datastruktur

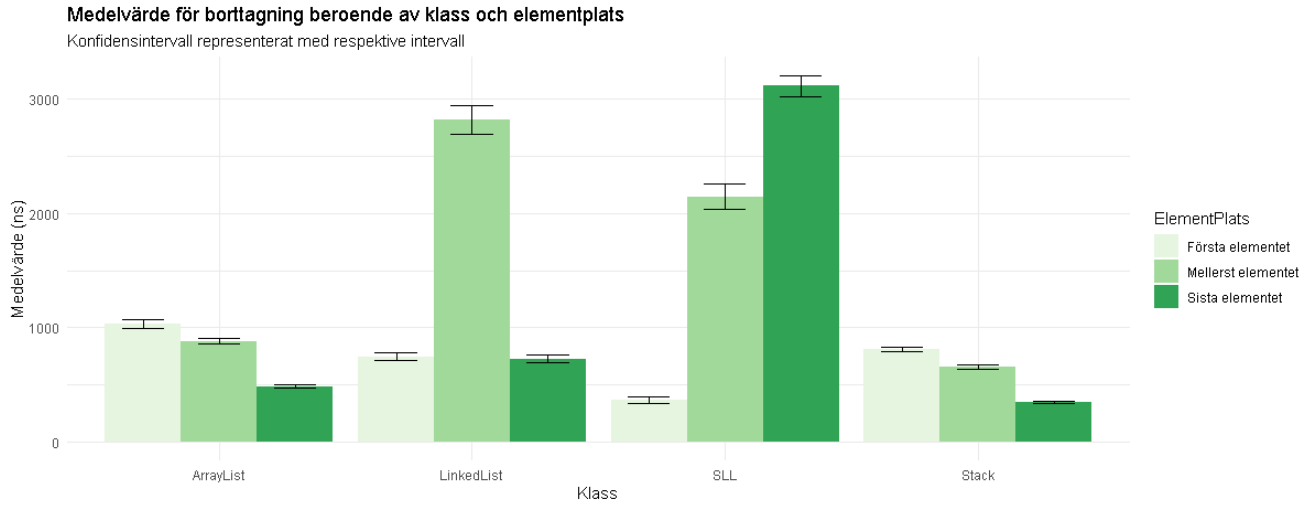

Fig. 4. Medelvärde för borttagning, och konfidensintervall av 100 iterationer, beroende av klass och elementplats, som är första, mellersta eller sista elementplats, av 800 element

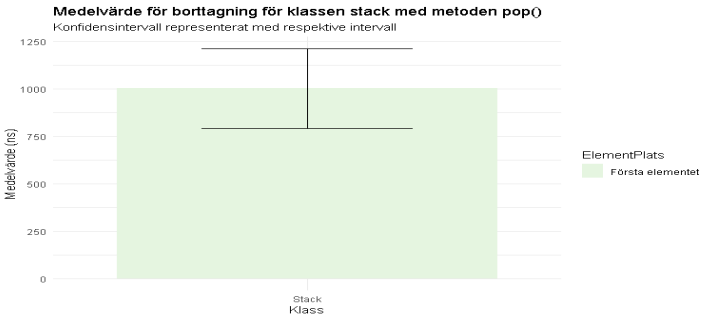

Fig. 5. Medelvärde för borttagning först, och konfidensintervall av 100 iterationer, för `Stack` med metoden `pop()`, av 800 element