Jämförelse mellan algoritmer för prioritetsköer med avseende på effektivitet

Madeleine Berner
<mberne@kth.se>

Martin Pola
<mpola@kth.se>

2 februari 2017

Sammanfattning

Studien jämför två olika algoritmer som kan användas för att implementera prioritetsköer i syfte att finna den mest effektiva. Den ena implementationen bygger på en dubbellänkad lista och den andra bygger på ett spretigt träd. Minnesförbrukningen och exekveringstiden jämförs för respektive algoritms bästa fall, värsta fall och normalfall genom analys och experiment. Målet är att hitta den algoritm som är bäst lämpad för att implementera prioriotetsköer för användning i generella fall. Resultatet från studien visar att spretiga träd är att föredra i de flesta generella fall, men att det i fall med små köstorlekar eller tillämpningar där det är synnerligen viktigt med deterministisk, konstant utplockningstid kan vara fördelaktigast med en dubbellänkad lista.

Abstract

The study compares twp algorithms that could be used to implement priority queues. One implementation is based on a double-linked list while the other implementation is based on a splay tree. The memory usage and execution time of the two algorithms are compared in the best case, worst case and normal case respectively. The goal is to find the algorithm that is best suited for implementing priority queues for general usage. The results show that splay trees are preferable in most cases, but that doubly-linked lists could be better in cases with tiny queue sizes or applications where a deterministic, constant fetch time is of importance.

Innehåll

1 Introduktion

Studien använder experimentella metoder för att utvärdera och jämföra två olika implementationer som båda kan användas för att konstruera en prioritetskö. De två implementationerna som ska behandlas är en dubbellänkad lista och ett spretigt träd (en. *splay tree* [1]). Utvärderingen ska företrädesvis behandla effektiviteten av båda implementationerna. Där effektiviteten inkluderar tids- och minneskomplexiteten hos algoritmerna, vilket med andra ord innebär att den ska se över snabbheten och minneseffektiviteten i de båda implementationerna.

1.1 Bakgrund

Algoritmer för implementationer av prioritetsköer har analyserats och jämförts ett inte oväsentligt antal gånger tidigare. En av studierna [2], utförd av Robert Rönngren et al., jämför dels olika sekventiella och dels olika parallella implementationer. I den här studien jämförs endast sekventiella implementationer, men det gör inte att ovan nämnda studie saknar betydelse.

Särskilt relevant är avsnittet kring prestandamätning, eftersom det i den här studien bland annat finns behov av en tillförlitlig metod för att mäta exekveringstid. För att mäta exekveringstid används slingor av likartade operationer, där ett aritmetiskt medelvärde sedan kan räknas fram genom att både den totala exekveringstiden och det totala antalet operationer är kända [3]. Ett liknande förfarande används i det här experimentet med ett spretigt träd.

1.2 Problem

Vilken av algoritmerna – dubbellänkad lista eller spretigt träd – är mest lämpad för att implementera en prioritetskö som ska vara effektiv för generell användning?

1.3 Syfte

Syftet med studien är att jämföra två olika implementationer av prioritetsköer för att kunna avgöra vilken implementation som är effektivast. Med hjälp av att genomföra ett experiment på implementationernas exekveringstid och minnesförbrukning kan resultatet användas för att dra slutsats om vilken implementation som har lägst exekveringstid i kombination med minst minnesförbrukning.

[1] Daniel Dominic Sleator och Robert Endre Tarjan. "Self-adjusting binary search trees". I: *Journal of the ACM (JACM)* 32.3 (1985), s. 652–686.

[2] Robert Rönngren. "A Comparative Study of Parallel and Sequential Priority Queue Algorithms". I: *ACM Trans. Model. Comput. Simul.* 7.2 (april 1997), s. 157–209. ISSN: 1049-3301. DOI: 10.1145/249204.249205. URL: http://doi.acm.org/10.1145/249204.249205.

[3] Rönngren, "A Comparative Study of Parallel and Sequential Priority Queue Algorithms".

1.4 Mål

Resultatmålet är att studien utifrån det som påverkar effektiviteten i prioritetsköer utvärdera hur två olika algoritmer presterar i de termerna.

Effektmålet är att studien skall kunna användas till att fatta beslut om val av bakomliggande algoritmer och datastrukturer för implementation av prioritetsköer avsedda att användas i generella fall.

1.5 Samhällsnytta, etik och hållbar utveckling

Prioritetsköer som datastruktur har många användningsområden [4,5]. Med en effektiv algoritm för prioritetsköer finns det således potential till att optimera en rad olika typer av mjukvarusystem.

Ett praktiskt exempel på ett område där prioritetsköer används är inom schemaläggningen för moderna operativsystem. Uppgiften hos schemaläggaren är att fördela ut processorkraften mellan de olika processerna och se till att varje process får lagom mycket körtid i lagom stora mängder. Optimalt sker detta när alla processer totalt sett körs på kortast möjliga tid. Eftersom schemaläggning är något som sker i realtid, parallellt med exekveringen, är det viktigt att de prioritetsköer som ligger bakom är effektiva. Om en ineffektiv algoritm för prioritetsköer i en schemaläggare implementeras kan det få drastiska konsekvenser i form av sämre prestanda i operativsystemet. Är operativsystemet vanligt förekommande, något exempelvis Windows 10 och Mac OS X båda är, kan en ineffektivt utformad prioritetskö i schemaläggaren orsaka sämre prestanda hos en hel generations datorer. Det är något som enbart med bättre genomtänkt mjukvara hade kunnat undvikas.

Även från ett etiskt perspektiv är det viktigt att kunna avgöra vilken implementation som är bäst lämpad. I den etiska policy IEEE har tagit fram står det bland annat att "[...] the members of the IEEE [...] agree [...] to improve the understanding of technology; its appropriate application, and potential consequences" [6]. En jämförelse mellan olika algoritmer för implementation av en prioritetskö går hand i hand med den punkten i IEEE:s etiska policy.

2 Algoritmer

I studien används två olika implementationer av prioritetsköer. De två algoritmerna för prioritetsköerna är en dubbellänkad lista och ett spretigt träd [7]. Algoritmerna har olika tidskomplexiteter för värsta fall, bästa fall och normalfall.

[4] Michael Siff. *Notes on Priority Queues*. 1997. URL: http://pages.cs.wisc.edu/~siff/CS367/Notes/pqueues.html (hämtad 2016-11-27).

[5] Robert Sedgewick och Kevin Wayne. *Priority Queue Applications*. 2007. URL: https://www.cs.princeton.edu/~rs/AlgsDS07/06PriorityQueues.pdf (hämtad 2016-11-27).

[6] IEEE. *IEEE Code of Ethics*. URL: http://www.ieee.org/about/corporate/governance/p7-8.html (hämtad 2016-11-18).

[7] Sleator och Tarjan, "Self-adjusting binary search trees".

2.1 Beskrivning av algoritmer för prioritetsköer

Algoritmerna är baserade på två olika datastrukturer och har därmed olika sätt att implementeras.

2.1.1 Dubbellänkad lista

Implementationen av prioritetskön med en dubbellänkad lista bygger på att alla element är sorterade och bär en pekare till nästa och en pekare föregående element. Det finns dessutom två pekare till listans första respektive sista element, som tack vare faktumet att listan är sorterad pekar på elementen med högst respektive lägst prioritet.

Eftersom det är en länkad lista sker insättning genom att implementationen stegar genom listan fram till den plats där det nya elementet ska sättas in. Stegningen kan, tack vare pekarna på de båda ändarna, ske antingen från högsta prioriteten och fallande, eller från lägsta prioriteten och stigande. Det som avgör vilken ingång implementationen tar är det aritmetiska medelvärdet av prioriteterna i listans båda ändar [8]. Om prioriteten på det element som ska sättas in är större än det aritmetiska medelvärdet börjar stegningen från det högst prioriterade elementet och fallande, och analogt från det lägst prioriterade elementet och stigande i det andra fallet.

Vid utplockning returnerar implementationen elementet med högst prioritet. För detta är det lämpligt att använda pekaren till det element i listans ena ände som har högst prioritet.

2.1.2 Spretigt träd

Ett spretigt träd [9] bygger på grunden av ett binärt träd. Skillnaden mellan ett binärt träd och ett spretigt träd är att det spretiga trädet har en uppsättning av tre *spretfunktioner* som ser till att trädet hålls balanserat.

Trädet består av flera noder och börjar med en rot. Roten innehåller antingen den nod som nyligen blev adderad till trädet, eller föräldern till den nod som nyligen togs bort från trädet. Resten av trädet består av noder och längst ut på grenarna finns det noder som kallas för löv. Barnen till en nod är sorterade på prioriteten som är tilldelad till varje nod. Det högra barnet innehåller en nod med högre prioritet och det vänstra barnet innehåller en nod med lägre prioritet.

En nod håller totalt tre pekare och två variabler. En pekare till förälderns nod, en pekare till det vänstra barnet och en pekar till det högra barnet. En variabel som håller i prioriteten samt en variabel som innehåller datan som ska förvaras i prioritetskön.

Det finns tre olika spretfunktioner som även kan genomföras åt det spegelvända hållet. De går under namnen zig-funktion, zig-zack-funktion och zig-zig-funktion. Spretfunktionerna används för att balansera trädet och triggas vid

[8] Robert Rönngren. *Laborationsuppgift*. 2016. URL: https://www.kth.se/social/course/II1304/page/laborationsuppgift/ (hämtad 2016-10-23).

[9] Sleator och Tarjan, "Self-adjusting binary search trees".

insättning och utplockning av element. Valet av spretfunktion vid insättning baseras på att den nyligen insatta noden skall placeras i roten. Målet blir därför att flytta upp den nyligen insatta noden högst upp i trädet och att behålla balansen i resterande delar av trädet. Vid utplockning baseras valet av spretfuktion på att få placera föräldern till den nod som nyligen blev utplockad i roten. Detta genomförs med hjälp av ett eller ett flertal kombinationer av spretfunktioner. För närmare dokumentation av implementationen av ett spretigt träd, se det citerade arbetet [10].

2.2 Analys av tidskomplexitet

En dubbellänkad lista och ett spretigt träd har olika minnes- och tidskomplexitet. Analysen av komplexiteterna ligger till grunden för hypotesen inför studiens experimentella del.

2.2.1 Dubbellänkad lista

Utplockning från prioritetsköer implementerade med dubbellänkade listor sker alltid med konstant tid, $O(1)$, eftersom listan har en pekare till det första elementet, och det är det första elementet som representerar det högst prioriterade elementet. Insättning sker, å andra sidan, inte nödvändigtvis på konstant tid.

I bästa fall sker även insättning av element i prioritetsköer implementerade med dubbellänkade listor på konstant tid $O(1)$, precis som vid utplockning. Detta är dock endast sant i följande tre fall.

1. Listan har 0 element.
2. Prioriteten på det element som sätts in är högre än den högsta prioriteten som redan finns i listan.
3. Prioriteten på det element som sätts in är lägre än den lägsta prioriteten som redan finns i listan.

Fall 2 och 3 implicerar att insättning sker på konstant tid även när listan har 1 element.

I värsta fallet sker insättning på $O(n)$ tid. När ett nytt element ska sättas in i listan väljer algoritmen att gå igenom listan från det element med högst prioritet och nedåt eller från det element med lägst prioritet och uppåt. Avgörande för vilken väg algoritmen väljer är det aritmetiska medelvärdet m mellan listans högsta och lägsta prioriteter. Är prioriteten för det element som ska sättas in större än m kommer algoritmen gå igenom listan fallande från det element med högst prioritet, medan den i andra fallet kommer gå igenom listan stigande från elementet med lägst prioritet. Detta tillvägagångssätt fungerar bäst när elementens prioriteter är jämnt fördelade mellan den högsta och den lägsta prioriteten. Är elementen jämnt fördelade skulle algoritmen behöva gå igenom

[10] Sleator och Tarjan, "Self-adjusting binary search trees".

maximalt $n/2$ element för att sätta in ett nytt element, vilket sker i fallet när det nya elementet ska sättas in i mitten av listan.

Det kan emellertid inte förutsättas att listans element är jämnt fördelade, något som innebär att algoritmen i värsta fallet skulle behöva gå igenom $n-1$ element för att hitta rätt plats för det element som ska sättas in. Ponera en situation där listan består av $n-1$ element med prioritet 1000 och 1 element med prioritet 0. Det aritmetiska medelvärdet av de första och de sista elementen blir $(1000+0)/2 = 500$. Ska ett element med prioritet 750 sättas in kommer algoritmen börja gå igenom elementen i fallande ordning med utgångspunkt från det element med högst prioritet, eftersom $750 > 500$. Det innebär att algoritmen kommer gå igenom de $n-1$ elementen med prioritet 1000 innan den finner att det nya elementet med prioritet 750 ska sättas in, precis före elementet med prioritet 0. Jämförelsevis hade algoritmen kunnat lägga in elementet genom att endast gå igenom ett enda element – det sista – genom att välja att stega elementen i ordning från andra hållet.

Anledningen till att insättning sker näst intill linjär tid, $O(n)$, i värsta fallet beror på att det artmetiska medelvärdet m endast tar hänsyn till de två element i listans ytterkanter. För att prestanda i fall med ojämnt fördelade prioriteter hade det varit lämpligt att räkna ut det aritmetiska medelvärdet av *alla* prioriteter i listan, och basera jämförelsen med prioriteten hos det nya elementet på det. En sådan beräkning kan naturligtvis inte göras vid varje insättning, eftersom att alla insättningar skulle ta minst $O(n)$ tid. Lämpligt är att i stället ha en variabel med summan av alla prioriteter och en variabel med antal element i listan – köstorleken. De två variablerna behöver därmed endast uppdateras vid varje insättning och utplockning, vartefter det går på konstant tid att räkna fram det riktiga aritmetiska medelvärdet av prioriteterna i listan. Det bör dock nämnas att inte ens detta är optimalt, utan endast ett heuristiskt försök att närma sig medianen.

Ett normalfall för en dubbellänkad lista är när elementen är insatta i slumpvis ordning med prioriteter från en likformig fördelning.

2.2.2 Spretigt träd

I ett spretigt träd sker varken insättning eller utplockning med konstant tidskomplexitet [11], eftersom det i båda fallen körs en eller flera slingor med spretfunktioner. För att beräkna tidskomplexiteten används en amorterad funktion och till varje tänkbart tillstånd av trädet tilldelas ett reellt värde, den så kallade *potentialen*. Funktionen ser ut som följer. $amorterad\ kostnad = kostnad + \Delta\Phi$. där $\Delta\Phi$ är skillnaden i potentialen orsakad av en spretfunktion. I en del fall körs en slinga med relativt dyra spretfunktioner, som mycket väl kan närma sig en linjär tidskomplexitet, medan i andra fall körs färre spretfunktioner som resulterar i en mer konstant tidskomplexitet. Det är svårt att exakt säga vilken indata som kommer att generera ett bästa och ett värsta fall på grund av att det är spretfunktionerna som styr tidskomplexiteten. Däremot konvergerar

[11] Sleator och Tarjan, "Self-adjusting binary search trees".

tidskomplexiteten till $O(log(n))$ vid upprepade insättningar eller utplockningar. För det spretiga trädet är den amorterade tidskomplexiteten, som tar hänsyn till en serie av anrop snarare än isolerade anrop, just $O(log(n))$.

Varje nod har även en rank tilldelad. En nod med rank 0 innebär att noden är längst ner i trädet och är alltid ett löv. En nod med rank $log(n)$ är roten av trädet.

Med ovanstående stycke som bakgrund blir tidskomplexiteten för bästa-, värsta- och normalfall *amorterad* $O(log(n))$, där n är antalet noder i trädet och $log(n)$ är maximum av ranken en nod kan ha. Viktigt att notera är att tidskomplexiteten gäller både vid insättning och utplockning, eftersom att båda operationerna ger upphov till att spretfunktionerna aktiveras.

2.3 Analys av minneskomplexitet

Förutom exekveringstid är även minneskomplexitet relevant för effektiviteten hos en prioritetskö.

Givet en modell där alla element har lika stora värden är det endast de interna datastrukturerna för den dubbellänkade listan respektive det spretiga trädet som inverkar på minnesanvändningen. Båda växer givetvis när fler element sätts in i prioritetskön – men växer de växer lika snabbt?

I fallet med den dubbellänkade listan behöver varje element ha en pekare till föregående element och en pekare till nästa element. I fallet med det spretiga trädet behöver varje element ha en pekare till sitt vänstra barn och en pekare till sitt högra barn, eftersom det i grund och botten är ett binärt träd, samt en pekare till dess förälder, för att kunna genomföra spretfunktionerna.

Givet att datastrukturerna för elementen i båda implementationerna innehåller lagringsplats för dels ett prioritetstal och dels ett värde är det endast antalet pekare som skiljer minnesförbrukningen åt. På ett 64-bitarssystem tar varje element i det spretiga trädet upp 64 bitar – åtta byte – mer minne än motsvarande element i den dubbellänkade listan. Det beror på att det finns tre pekare hos det spretiga trädet jämfört med endast två pekare hos den dubbellänkade listan.

Eftersom skillnaden på åtta byte per element är oberoende av hur många element prioritetsköerna har blir minneskomplexiteten i båda fallen linjär, $O(n)$, för en köstorlek på n element.

3 Metod

Eftersom studien fokuserar på effektiviteten mellan de två prioritetsköimplementationerna är det centralt att mäta och jämföra *exekveringstid* och *minnesförbrukning*. Mer precist är det mätning och jämförelse av

- den tid det tar för insättning och utplockning i prioritetsköer baserade på dubbellänkade listor respektive spretiga träd, och
- hur mycket minne de använder i förhållande till köstorleken.

3.1 Mätmetoder

För att få en uppfattning av de både implementationernas effektivitet behöver hänsyn tas dels till exekveringstid och dels till minnesförbrukning.

3.1.1 Exekveringstid

För att mäta tiden det tar att exekvera implementationerna är det viktigt att definiera vad som åsyftas med *exekveringstid*. I den mest grundläggande formen är exekveringstid *tiden en process tar från start till slut*. Eftersom implementationerna körs på ett modernt operativsystem är det emellertid inte riktigt så enkelt. I ett modernt operativsystem kommer schemaläggaren sannolikt att påverka exekveringstiden [12], eftersom den emellanåt pausar exekveringen av implementationerna för att exekvera en eller flera andra processer. Även om algoritmerna som används för detta ofta är deterministiska beror de på många parametrar som gör att det blir praktiskt omständligt att analytiskt räkna fram hur lång tid en enskild process konsumerade.

I stället för att låta exekveringstiden vara den faktiska tiden en process tar från start till slut (en. *wall time*) är det med anledning av ovanstående mer ändamålsenligt att endast mäta tiden som den enskilda processen konsumerade – den så kallade *CPU-tiden* (en. *CPU time*). Genom att mäta CPU-tiden kommer eventuella andra processer i operativsystemet inte påverka mätvärdena, trots att de i verkligheten gör att den faktiska tiden för implementationerna ökar.

För den här studien likställs därför *exekveringstid* med *CPU-tid*.

Ett sätt att mäta CPU-tid är att använda den i C tillhandahållna funktionen `clock` [13]. Enligt Linux-manualen returnerar funktionen ett värde som indikerar hur mycket CPU-tid den aktuella processen har konsumerat vid tillfället för funktionsanropet. För att mäta CPU-tiden för en godtycklig funktion går det att omsluta funktionen med två anrop till `clock`, vars värden sedan subtraheras. I avsnitt 3.2.1 valideras `clock` i syfte att troliggöra att dess returnerade värden är korrekta.

Eftersom mätvärdena kommer inkludera den overhead som uppkommer av bland annat funktionsanrop och generering av slumptalsvärden ska det för varje mätning även göras en mätning av denna overhead. För det ändamålet skapas ett tomt skal bestående av insättnings- och utplockningsoperationer som inte gör någonting alls. Genom att mäta exekveringstiden för dels en riktig implementation och dels för det tomma skalet kan den delen av exekveringstiden som uppkom på grund av overheaden subtraheras bort.

3.1.2 Minnesförbrukning

Hur mycket minne implementationerna använder för att lagra sina respektive element beror varken på vilka prioriteter elementen har eller på i vilken ordning

[12] Andrew S. Tanenbaum. *Modern Operating Systems. Third Edition*. Pearson, 2015, s. 145–149. ISBN: 978-93-325-5001-8.

[13] Rik Faith. *clock(3) – Linux manual page*. 2015. URL: http://man7.org/linux/man-pages/man3/clock.3.html (hämtad 2017-01-31).

insättning och utplockning sker, något som tydliggjordes i avsnitt 2.3. För att från ett minnesförbrukningsperspektiv få en uppfattning av effektiviteten hos implementationerna är det fullt tillräckligt att på analytisk väg komma fram till hur mycket minne de använder. Det är således inte nödvändigt att mäta minnesförbrukningen.

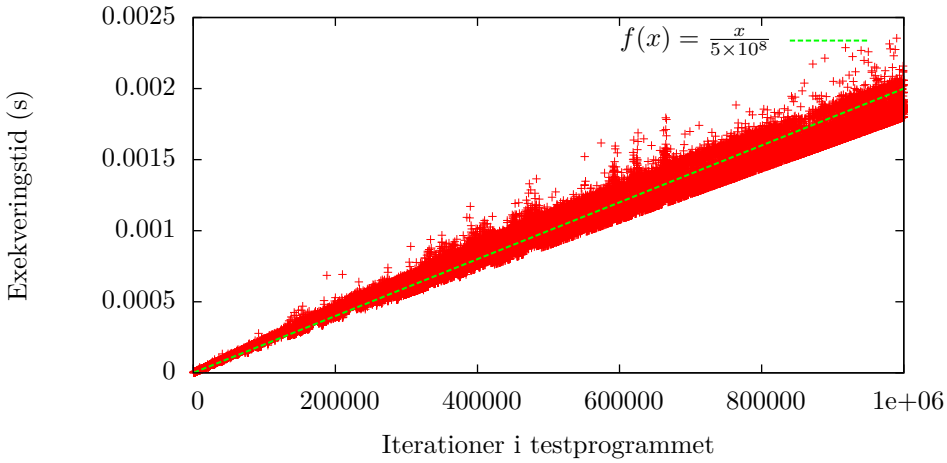
3.2 Validering

I studien används ett verktyg för att mäta exekveringstid och ett verktyg för att generering av slumptal. För att troliggöra att verktygen fungerar som tilltänkt behöver de valideras.

3.2.1 Verktyg för mätning av exekveringstid

Att blint lita på att `clock` inte är behäftat med något fel som gör att den uppmätta tiden avsevärt skiljer sig från den faktiskt tiden är inte aktuellt. Innan förfarandet med jämförelser av två `clock`-anrop kan användas är det därför viktigt att validera dess korrekthet – kan det göras troligt att den uppmätta exekveringstiden inte skiljer sig avsevärt från den faktiskt exekveringstiden? För att göra en sådan validering har ett testprogram konstruerats. Testprogrammet tar ett positivt heltal som parameter och kör en slinga med lika många iterationer som argumentet. Exekveringstiden för testprogrammet, som i således i huvudsak består av nämnda slinga, mäts i enlighet med beskrivningen i avsnitt 3.1.1.

Figur 1 visar exekveringstiden för parametrar från 10^3 upp till 10^6, där testprogrammet har körts 50 gånger för varje enskilt parametervärde. Systemet som valideringen sker på är det som beskrivs i avsnitt 4.3.1 och testprogrammet finns i sin helhet i bilaga A).



Figur 1: Exekveringstid för körningar av testprogrammet med parametrar från 10^3 till 10^6 tillsammans med en linjär funktion $f(x)$

Som figuren visar stiger exekveringstiden proportionellt mot antalet iterationer i testprogrammet, något som är önskvärt eftersom en linjär ökning i antalet instruktioner torde medföra en linjär ökning i tiden det tar att exekvera instruktionerna.

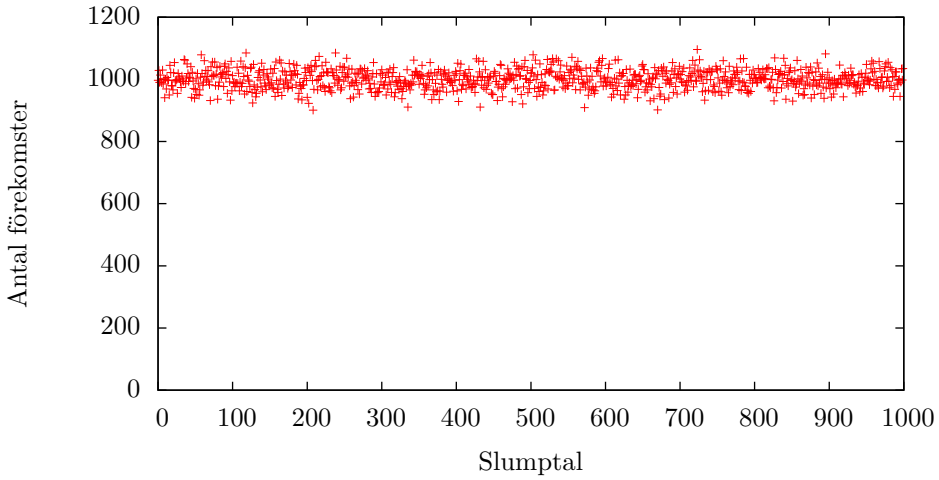
Vidare är exekveringstiderna för de 50 körningarna av respektive parameter någorlunda koncentrerade. Det antyder att förfarandet med jämförelser av två `clock`-anrop ger likartade resultat för upprepade exekveringar av samma uppsättning instruktioner – samma program med samma parameter. Eftersom det ändå finns en del avvikande värden är det viktigt att ta hänsyn till att det sannolikt också kan vara fallet när implementationerna av prioritersköerna exekveras.

3.2.2 Verktyg för generering av slumptal

Några av studiens testfall bygger på att det finns en källa som ger slumptal från en likformig fördelning. Den slumptalsgenerator som används är den i C tillhandahållna funktionen `rand` – men ger den verkligen en likformig fördelning? För att troliggöra detta valideras det genom att titta på fördelningen av 10^6 anrop till `rand`. Resultatet visas i figur 2, där slumptalen har justerats till

att ligga på intervallet $[0 - 1000]$. Utifrån figuren är det rimligt att anta att slumptalsfördelningen i `rand` verkar vara likformigt fördelad.

Bilaga B) innehåller den programkod som användes för att generera slumptalen i den här valideringen. Det slumptalsfrö som användes var 42.



Figur 2: Antal förekomster av 10^6 slumptalsgenererade tal på intervallet $[0 - 1000]$

3.3 Felkällor

Det finns flera felkällor som kan påverka utfallet av mätvärdena, något som i värsta fall kan leda till felaktiga slutsatser kring vilken algoritm som är mest effektiv. Eftersom exekveringstid är det enda som faktiskt *mäts* – minnesförbrukning räknas fram analytiskt – ligger fokus för felkällor på sådant som kan påverka exekveringstiden.

Verktyget som används för att mäta exekveringstiden, `clock`, validerades i avsnitt 3.2.1. I tillhörande figur 1 är det tydligt att den metoden inte är helt fri från fel. Om det beror på själva mätmetoden eller om det kanske snarare beror på det underliggande operativsystemet är inte särskilt relevant – det viktiga är vetskapen om att ett program som exekveras upprepade gånger med samma indata varierar i exekveringstid. För att motverka att sådana fel påverkar utfallet av studien körs varje försök flera gånger, i syfte att med hjälp av ett

konfidensintervall kunna fastställa ett troligt väntevärde för exekveringstiden i respektive testfall.

En annan potentiell källa till fel är den overhead som läggs till exekveringstiden vid en cachemiss. Att få cachemissar är praktiskt taget oundvikligt, men antalet cachemissar ska hållas på en representativ nivå. Eftersom andelen cachemissar är större i början av en exekvering påbörjas mätningen av exekveringstid först en bit in i varje testfall, när prioritetskön i den aktuella implementationen har fyllts upp med data. Genom att *värma upp* cacharna torde mätvärdena inte fläckas av orimligt stora andelar cachemissar.

Utöver det kan konfigurationen av operativsystem och typen av processor påverka utfallet av exekveringstiderna. Eftersom det är komplicerat att ta reda på hur ändringar av dessa parametrar påverkar exekveringstiderna läggs fokus på att hålla dem konstanta under exekveringen av samtliga testfall. Resonemanget bakom det är att risken minskar för att nämnda parametrar har en betydlig inverkan på mätvärdena.

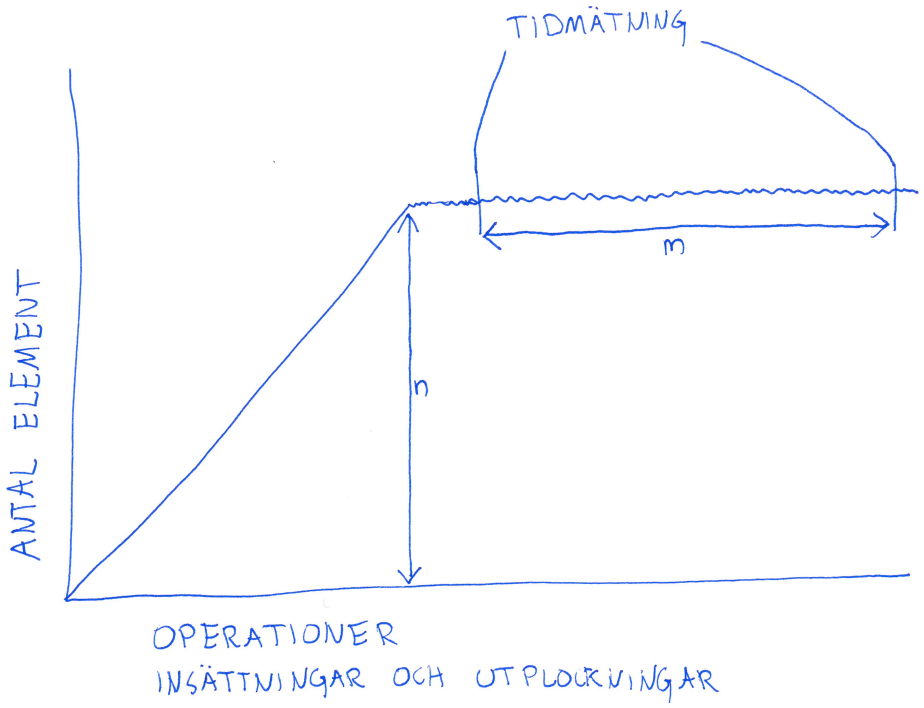
4 Experimentuppställning

Det här avsnittet behandlar hur experimentet implementeras, vilka resurser som används och hur experimentuppställningen valideras.

4.1 Tillvägagångssätt

Experimentet består av en uppsättning testfall, där varje testfall utgörs av ett antal insättnings- och utplockningsoperationer i de båda prioritetsköerna. För att andelen cachemissar inte ska bli orimligt många påbörjas inte mätningen av exekveringstiden förrän prioritetskön har fyllts upp med ett bestämt n element. När prioritetskön har fyllts upp populeras cacharna genom ett fåtal insättningar och utplockningar, vartefter mätningen av exekveringstiden påbörjas. Sedan utförs ett bestämt m antal insättnings- och utplockningsoperationer växelvis, och det är exekveringstiden för de operationerna som mäts.

För att med större säkerhet kunna urskilja skillnader körs varje testfall flera gånger med n bestämt till $[5, 10, .., 1000]$, vilket medför att eventuella trender tydliggörs. Värdet på m – antalet insättnings- och utplockningsoperationer som utförs när cachen är uppvärmd – är emellertid konstant bestämt till 10000. Figur 3 illustrerar förfarandet för ett godtyckligt testfall.



Figur 3: Illustration av när exekveringstiden mäts i ett testfall.

4.2 Testfall

Vilka prioriteter som används vid de insättningsoperationer som beskrevs i föregående avsnitt, 4.1, bestäms av själva testfallen. De fem testfall som används är

- ett som prövar värsta fallet för den dubbellänkade listan,
- ett som prövar bästa fallet för den dubbellänkade listan, samt
- tre som prövar både normalfallet för den dubbellänkade listan och alla fall för det spretiga trädet.

Det som skiljer de tre sista testfallen åt är att olika slumptalsfrön används för slumptalsgeneratorn. De slumptalsfrön som används är 9, 17 och 4711. Resonemanget bakom varför de tre sista testfallen inte bara prövar normalfallet för den dubbellänkade listan, utan även alla fall för det spretiga trädet, återfinns i avsnitt 2.2.2. Samtliga testfall körs 25 gånger per värde på n för att reducera en felkälla – se avsnitt 4.1 och 3.3 för vidare resonemang. Med fem testfall som körs 25 gånger för varje värde på n i $[5, 10, .., 1000]$ för två algoritmer och ett

tomt skal (se avsnitt 3.1.1] blir det totalt

$$5 \times 25 \times 100 \times (2 + 1) = 37500$$

körningar tillika uppmätta exekveringstider.

4.3 Testbädd

Experimentuppställningen inkluderar de resurser som krävs för att genomföra experimentet samt valideringen av uppställningen.

4.3.1 Resurser

Experimentet genomförs på en dator med x86_64-arkitektur och Linux 4.9.6-1-ARCH som operativsystem. Kompilatorn som används för att kunna exekvera C-koden är GCC 6.3.1 20170109 utan särskilda flaggor för optimering. Varje körning kombinerar grunden `queue.c` med precis ett testfall och precis en algoritm på följande vis.

```
gcc -Wall -Wextra -o queue queue.c testfall algoritm
```

4.3.2 Validering av experimentuppställning

Innan körningarna påbörjas ska implementationerna valideras. Syftet med valideringen är att troliggöra att algoritmerna bakom prioritetsköerna är korrekt implementerade. Trots att implementationerna har olika bakomliggande algoritmer ska de utåt sett bete sig likvärdigt, eftersom båda ska kunna användas som prioritetsköer. Det medför att samma indata för båda implementationerna ska resultera i samma utdata. Eftersom prioritetsköer är deterministiska är det möjligt att analytiskt komma fram till vilken utdata en viss indata borde ge. Serier av väl valda indata används därför för att troliggöra att implementationerna fungerade korrekt.

Testfallet för validering av implementationerna finns i bilaga C). Det ska köras för båda algoritmerna och resultatet ska sedan jämföras med det analytiskt framräknade resultatet, som också finns i samma bilaga.

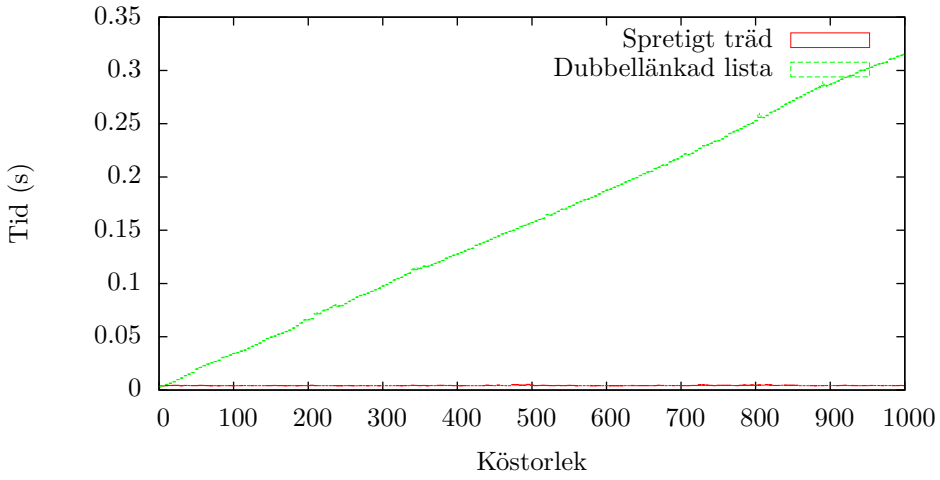
5 Resultat

Mätningar av exekveringstider från körningar av de fem testfallen illustreras i figur 4, figur 6. figur 7, figur 8 och figur 9. För att tydliggöra exekveringstiden för små köstorlekar i det första testfallet är figur 5 en förfinad variant av figur 4. I samtliga figurer används ett 97-procentigt konfidensintervall för att reducera statistisk osäkerhet bland mätvärdena.

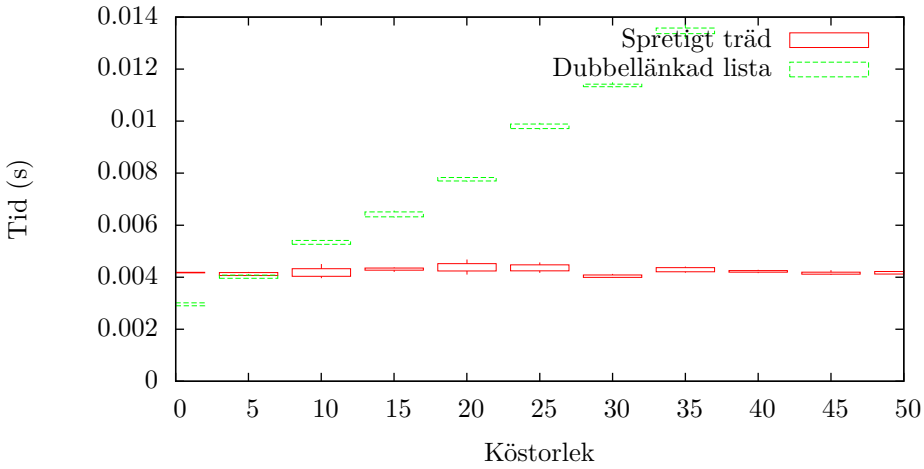
Det spretiga trädet har en märkbart bättre exekveringstid än den dubbellänkade listan för den dubbellänkade listans värsta fall, i synnerhet för större

köstorlekar. Motsatsen råder inte – även om den dubbellänkade listan uppträder bättre än det spretiga trädet i den dubbellänkade listans bästa fall är det spretiga trädet enbart lite sämre.

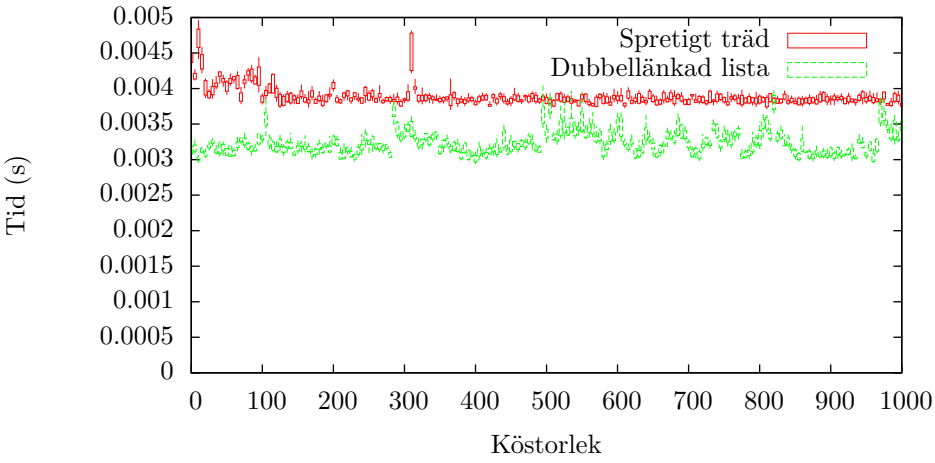
I figur 7, figur 8 och figur 9 härstammar prioriteterna för de insatta elementen från en slumptalsgenerator. För de fallen är den dubbellänkande listan snabbare på att hantera insättningar och utplockningar upp till en köstorlek på ungefär 500 element. När köstorleken överstiger det blir den dubbellänkade listan märkbart långsammare, och det med en tilltagande hastighet.



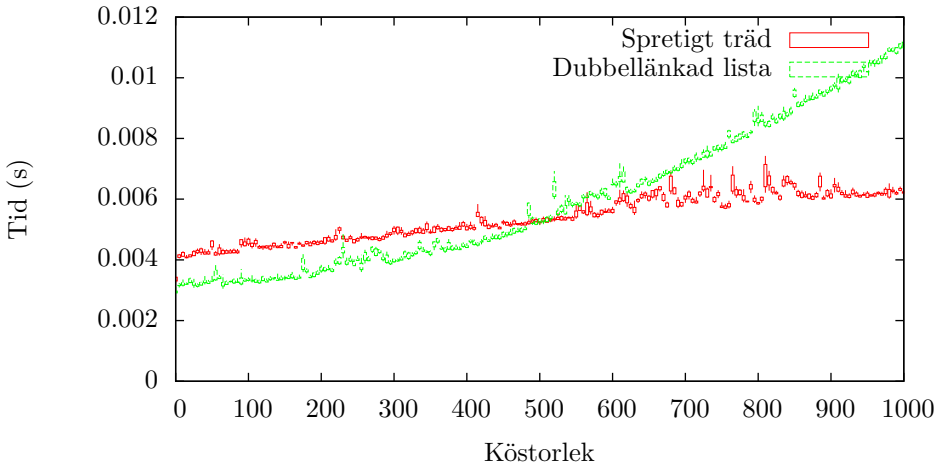
Figur 4: Värsta fallet för dubbellänkade listor



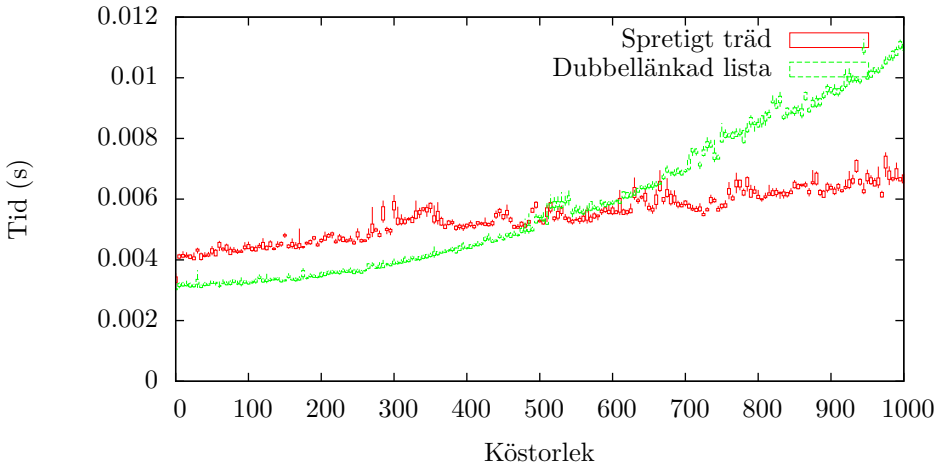
Figur 5: Värsta fallet för dubbellänkade listor med fokus på fallen med relativt få element



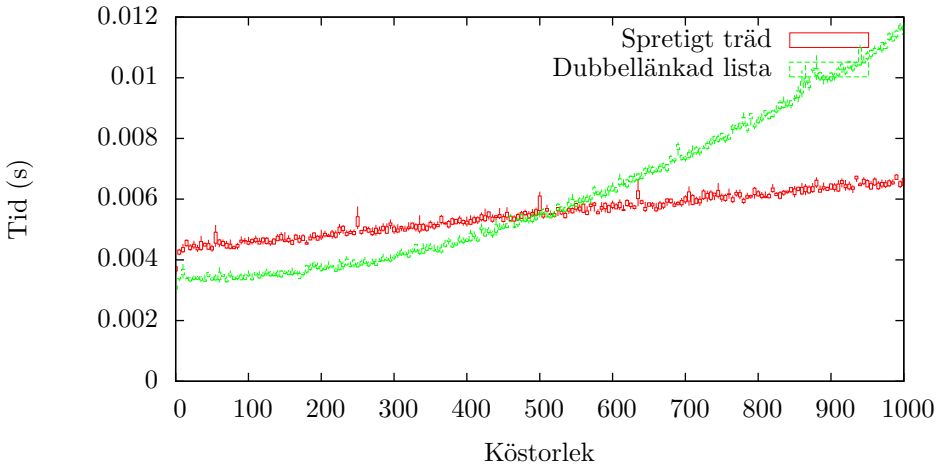
Figur 6: Bästa fallet för dubbellänkade listor



Figur 7: Normalfallet för dubbellänkade listor och samtliga fall för spretiga träd, med slumptalsfrö 9



Figur 8: Normalfallet för dubbellänkade listor och samtliga fall för spretiga träd, med slumptalsfrö 17



Figur 9: Normalfallet för dubbellänkade listor och samtliga fall för spretiga träd, med slumptalsfrö 4711

6 Diskussion

Nästan alla mängder om 25 uppmätta exekveringstider höll sig inom snäva gränser, något som talar för att valet av tidmätningsverktyg var rätt. Det var förvisso redan troliggjort när det validerades i avsnitt 3.2.1, men det är ändå betryggande att mätvärdena från testfallen liknande de från valderingen.

6.1 Exekveringstider

Av figur 4 och framgår det att den dubbellänkade listan är långsammare i jämförelse med det spretiga trädet vid testning av det värsta fallet för den dubbellänkade listan. Det är emellertid helt i sin ordning, eftersom testfallet avsåg just det *värsta* fallet. Figur 5 visar samma testfall med köstorlekar upp till och med 50 element. Den dubbellänkade listan har förvisso ett exekveringstidsmässigt övertag för köstorlekar på upp till 5 element, men för enbart den dubbla köstorleken är den avsevärt långsammare.

Den dubbellänkade listans teoretiska bästa fall, med konstant tidskomplexitet för både insättning och utplockning, återspeglas väl i det andra testfallet,

som visualiserat i figur 6. Förutom att exekveringstiden till synes inte är beroende av köstorleken är den dubbellänkande listan genomgående snabbare än det spretiga trädet. Det spretiga trädet är dock inte långt därifrån; exekveringstiden i det testfallet stiger inte alls med samma hastighet som exekveringstiden för den dubbellänkade listan i föregående testfall (figur 4). Oavsett exekveringstider är detta ett specialfall av en prioritetskö i vilket datastrukturen snarare skulle kunna liknas vid en stack. Eftersom studien syftar till att hitta den mest effektiva algoritmen för *normal användning* är detta specialfall något som inte mer än ringa hänsyn behöver tas till.

I de tre sista testfallen, där prioriteterna vid insättningsoperationer hämtades från en slumptalsgenerator, visar figur 7, figur 8 och figur 9 liknande exekveringstider för alla tre slumptalsfrön. Den dubbellänkade listan var i regel snabbare än det spretiga trädet för köstorlekar upp till 500 element, medan det spretiga trädet sedan tog överhand och exekverade snabbare för större köstorlekar. Skillnaden i exekveringstid för köstorlekar mindre än 500 element är dock inte lika stor som skillnaden i exekveringstid för köstorlekar större än 500 element. Med andra ord innebär det att den dubbellänkade listan jämförelsevis inte är lika snabb för mindre köstorlekar som det spretiga trädet är för större köstorlekar.

6.2 Minnesanvändning

Effektivitet handlar dock inte enbart om exekveringstid, utan i effektivitetsmåttet ska även minnesanvändning vägas in. Genom analys av den bakomliggande C-koden för respektive implementationer är det tydligt att minnesanvändningen inte skiljer mer än med en konstant faktor. Båda implementationerna använder en `struct` för varje element, där det förutom prioritet och värde även sparas en del pekare. För den dubbellänkade listan sparas det två pekare per element; en framåt och en bakåt. Det spretiga trädet sparar tre pekare per element; en till vänsterbarnet, en till högerbarnet och en till dess förälder.

Skillnaden mellan att spara två och tre pekare per element gör att minnesanvändningen endast skiljer sig åt med en konstant faktor, något som gör att den dubbellänkade listan knappast kan ses som en vinnare ur effektivitetssynpunkt endast på grund av det.

7 Avslutning

Målet med studien är att identifiera vilken av två algoritmer som är mest effektiv för att implementera en prioritetskö. Effektiviteten mäts i avseende på exekveringstid och minnesförbrukning.

Den dubbellänkade listan är något bättre när prioritetskön har relativt få element. I takt med att köstorleken stiger blir exekveringstiden snabbare sämre än motsvarande för spretiga träd. Om tillämpningen aldrig, eller åtminstone mycket sällan, kommer behöva många element i sin prioritetskö kan den dubbellänkade listan därför vara ett alternativ. Det gäller i synnerhet om listan

implementeras i en omgivning där primärminnet är jämförelsevis litet – exempelvis i ett inbyggt system.

I de fall där det är mycket viktigare att utplockning sker snabbt än att insättning inte går relativt långsamt är den dubbellänkade listan ett bra alternativ. Där sker utplockning alltid på konstant tid, på bekostnad av ett insättning kan bli väldigt dyrt om det ter sig illa. Ett sådant val bör dock göras med väldig eftertanke, eftersom det krävs en väldigt stor köstorlek för att en logaritmisk operation – utplockning från spretiga träd – ska bli märkbart sämre än en konstant operation. Å andra sidan är utplockning från spretiga träd en *amorterat* logaritmisk operation, vilket innebär att den i enstaka fall kan bli väldigt dyr.

Det spretiga trädet bidrar med en klar fördel när storleken på kön inte är liten eller när storleken på kön är okänd. Även om exekveringstiden är något sämre jämfört med den dubbellänkade listan för små köstorlekar, och minnesanvändningen något större, blir exekveringstiden avsevärt bättre när köstorleken växer.

7.1 Slutsats

Vilken av de två algoritmerna bör användas för implementera effektiva prioritetsköer för användning i generella fall? För prioritetsköer där storleken är relativt liten eller en deterministisk och snabb utplockning är viktigt är dubbellänkade listor att föredra medan spretiga träd är att föredra endera där storleken inte är relativt liten eller där storleken är oförutsägbar.

7.2 Framtida arbete

Jämförelsen i det här studien begränsar definitionen av var som räknas till *generella fall* i flera avseenden. Ett av de tydligaste är att generella fall kan innefatta mer än de växelvisa insättnings- och utplockningsoperationer som beskrevs i avsnitt 4.1. Ett förfarande där en prioritetskö exempelvis först utsätts för fem insättningar och därefter tre utplockningar är inte helt otänkbar. Det leder till att en studie av fall med slumpvalda *operationer*, utöver slumpvalda *prioriteter*, skulle kunna ha bäring på resultatet.

Något som behandlas i både avsnitt 6 och avsnitt 7 är företeelsen att dubbellänkade listor i normalfallet exekverar snabbare än spretiga träd så länge köstorleken är relativt liten. Med det i baktanke hade det varit intressant att se hur en hybridlösning hade presterat – en implementation där den underliggande algoritmen ändras när köstorleken överstiger ett visst tröskelvärde eller understiger ett (annat) tröskelvärde.

Vidare antyder figur 6 att det spretiga trädet, i paritet med den dubbellänkade listan, skulle ha en konstant tidskomplexitet. Av figuren att döma är köstorleken inte en påverkande faktor för någon av implementationerna, trots att den bör vara det åtminstone för det spretiga trädet. Det är sannolikt ett mätfel som tydliggörs om köstorleken stiger till fler än 1000 element, men trots det vore det intressant att studera.

8 Citerade arbeten

[1] Daniel Dominic Sleator och Robert Endre Tarjan. ”Self-adjusting binary search trees”. I: *Journal of the ACM (JACM)* 32.3 (1985), s. 652–686.

[2] Robert Rönngren. ”A Comparative Study of Parallel and Sequential Priority Queue Algorithms”. I: *ACM Trans. Model. Comput. Simul.* 7.2 (april 1997), s. 157–209. ISSN: 1049-3301. DOI: 10.1145/249204.249205. URL: http://doi.acm.org/10.1145/249204.249205.

[4] Michael Siff. *Notes on Priority Queues*. 1997. URL: http://pages.cs.wisc.edu/~siff/CS367/Notes/pqueues.html (hämtad 2016-11-27).

[5] Robert Sedgewick och Kevin Wayne. *Priority Queue Applications*. 2007. URL: https://www.cs.princeton.edu/~rs/AlgsDS07/06PriorityQueues.pdf (hämtad 2016-11-27).

[6] IEEE. *IEEE Code of Ethics*. URL: http://www.ieee.org/about/corporate/governance/p7-8.html (hämtad 2016-11-18).

[8] Robert Rönngren. *Laborationsuppgift*. 2016. URL: https://www.kth.se/social/course/II1304/page/laborationsuppgift/ (hämtad 2016-10-23).

[12] Andrew S. Tanenbaum. *Modern Operating Systems. Third Edition*. Pearson, 2015, s. 145–149. ISBN: 978-93-325-5001-8.

[13] Rik Faith. *clock(3) – Linux manual page*. 2015. URL: http://man7.org/linux/man-pages/man3/clock.3.html (hämtad 2017-01-31).

9 Bilagor

Följande sidor innehåller den programkod som användes i studien.

A) Testprogram för validering av verktyg för mätning av exekveringstid

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
   if (argc != 2)
   {
      printf("usage: validate <loop iterations>\n");
      return 1;
   }

   int iterations = atoi(argv[1]);
   volatile int i = 0;

   clock_t time_start = clock();

   while (i < iterations)
   {
      i += 1;
   }

   clock_t time_stop = clock();

   printf("%d %.9f\n", iterations, ((double) (time_stop - time_start)) /
        CLOCKS_PER_SEC);

   return 0;
}
```

B) Testprogram för validering av slumptalsgenerator

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 1000

int main(int argc, char* argv[])
{
   if (argc != 3)
   {
      printf("usage: validate-rand <loop iterations> <seed>\n");
      return 1;
   }

   int iterations = atoi(argv[1]);
   int seed = atoi(argv[2]);

   srand(seed);

   int random_numbers[MAX];

   for (int i = 0; i < MAX; i++)
   {
      random_numbers[i] = 0;
   }

   for (int i = 0; i < iterations; i++)
   {
      random_numbers[rand() % MAX]++;
   }

   for (int i = 0; i < MAX; i++)
   {
      printf("%d %d\n", i, random_numbers[i]);
   }

   return 0;
}
```

C) Testfall för validering av implementationer

```c
#include <stdio.h>
#include <stdlib.h>
#include "../queue.h"

int run_test(int* argc, char* argv[])
{
   insert(1, "string 1");
   insert(2, "string 2");
   insert(3, "string 3");
   insert(4, "string 4");
   insert(5, "string 5");
   insert(6, "string 6");
   printf("fetched element 6: %s\n", fetch());
   insert(7, "string 7");
   insert(8, "string 8");
   insert(9, "string 9");
   printf("fetched element 9: %s\n", fetch());
   insert(10, "string 10");
   printf("fetched element 10: %s\n", fetch());
   printf("fetched element 8: %s\n", fetch());
   printf("fetched element 7: %s\n", fetch());
   printf("fetched element 5: %s\n", fetch());
   printf("fetched element 4: %s\n", fetch());
   printf("fetched element 3: %s\n", fetch());
   printf("fetched element 2: %s\n", fetch());
   printf("fetched element 1: %s\n", fetch());
   printf("fetched no element: %s\n", fetch());
   insert(5, "string 5");
   printf("fetched element 5: %s\n", fetch());
   insert(10, "string 10");
   insert(10, "string 10");
   insert(10, "string 10");
   printf("fetched element 10: %s\n", fetch());
   printf("fetched element 10: %s\n", fetch());
   insert(20, "string 20");
   printf("fetched element 20: %s\n", fetch());
   printf("fetched element 10: %s\n", fetch());
   printf("fetched no element: %s\n", fetch());

   return 0;
}
```

D) Testfall 1

```c
#include <stdio.h>
#include <stdlib.h>
#include "../queue.h"

#define M 100000

int run_test(int* argc, char* argv[])
{
   if (*argc != 2)
   {
      printf("usage: queue <number of elements>\n");
      return 1;
   }

   int elements = atoi(argv[1]);
   char string[] = "value";

   // fyll upp
   insert(0, string);

   for (int i = 1; i < elements; i++)
   {
      insert(rand() % 10000, string);
   }

   stopwatch();

   // vågrörelse
   for (int i = 0; i < M; i++)
   {
      insert(1000, string);
      fetch();
   }

   stopwatch();

   return 0;
}
```

E) Testfall 2

```c
#include <stdio.h>
#include <stdlib.h>
#include "../queue.h"

#define M 100000

int run_test(int* argc, char* argv[])
{
   if (*argc != 2)
   {
      printf("usage: queue <number of elements>\n");
      return 1;
   }

   int elements = atoi(argv[1]);
   char string[] = "value";

   // fyll upp
   for (int i = 0; i < elements; i++)
   {
      insert(0, string);
   }

   stopwatch();

   // vågrörelse
   for (int i = 1; i <= M; i++)
   {
      insert(i, string);
   }

   stopwatch();

   return 0;
}
```

F) Testfall 3, 4 och 5

```c
#include <stdio.h>
#include <stdlib.h>
#include "../queue.h"

#define M 100000

int run_test(int* argc, char* argv[])
{
   if (*argc != 3)
   {
      printf("usage: <number of elements> <seed>\n");
      return 1;
   }

   int elements = atoi(argv[1]);
   int seed = atoi(argv[2]);

   char string[] = "value";

   srand(seed);

   // fyll upp
   for (int i = 0; i < elements; i++)
   {
      insert(rand() % 10000, string);
   }

   stopwatch();

   // vågrörelse
   for (int i = 0; i < M; i++)
   {
      insert(rand() % 10000, string);
      fetch();
   }

   stopwatch();

   return 0;
}
```

G) Implementation med dubbellänkad lista

```c
#include <stdio.h>
#include <stdlib.h>
#include "linkedlist.h"

struct ListElement * first = NULL;
struct ListElement * last = NULL;

// lägg in ett element i prioritetskön
void insert(int priority, char * value)
{
   // skapa det nya elementet
   struct ListElement * element = malloc(sizeof(struct ListElement));

   element->priority = priority;
   element->value = value;
   element->next = NULL;
   element->previous = NULL;

   // prioritetskön är tom
   if (first == NULL)
   {
      first = element;
      last = element;
   }

   // prioritetskön är inte tom -- hitta rätt plats för elementet som ska läggas in
   else
   {
      int average = (first->priority + last->priority) / 2.0d;

      // insättning framifrån
      if (priority > average)
      {
         // specialfall: elementet ska in allra först
         if (element->priority > first->priority)
         {
            struct ListElement * tmp = first;

            first = element;
            element->next = tmp;
            tmp->previous = element;
         }

         else
         {
            struct ListElement * current = first;

            while (current->next != NULL && current->priority >= element->priority)
            {
               current = current->next;
            }

            current = current->previous;
```

```
54
55              struct ListElement * next_old = current->next;
56
57              current->next = element;
58
59              element->next = next_old;
60              element->previous = current;
61
62              if (next_old != NULL) { next_old->previous = element; }
63              else { last = current; }
64           }
65        }
66
67        // insättning bakifrån
68        else
69        {
70           // specialfall: elementet ska in allra sist
71           if (last->priority >= element->priority)
72           {
73              struct ListElement * tmp = last;
74
75              last = element;
76              element->previous = tmp;
77              tmp->next = element;
78           }
79
80           else
81           {
82              struct ListElement * current = last;
83
84              while (current->previous != NULL && element->priority > current
                    ->priority)
85              {
86                 current = current->previous;
87              }
88
89              current = current->next;
90
91              struct ListElement * previous_old = current->previous;
92
93              current->previous = element;
94
95              element->next = current;
96              element->previous = previous_old;
97
98              if (previous_old != NULL) { previous_old->next = element; }
99              else { first = element; }
100          }
101       }
102    }
103 }
104
105 // hämtar det element med högst prioritet
106 char * fetch()
107 {
108    if (first == NULL) { return NULL; }
109
```

```
110     // pekare till början av den sträng vi ska returnera
111     char * value_to_return = first->value;
112
113     // ändra referenserna 'first' och 'last'
114     if (last == first) { last = NULL; }
115     first = first->next;
116
117     return value_to_return;
118 }
119
120 // används för att felsöka prioritetskön
121 void debug_print_queue()
122 {
123     printf("-----------\n");
124
125     struct ListElement * current = first;
126
127     while (current != NULL)
128     {
129         printf("%10d '%s'\n", current->priority, current->value);
130         current = current->next;
131     }
132
133     printf("-----------\n");
134 }
```

H) Implementation med spretigt träd

```c
#include <stdio.h>
#include <stdlib.h>
#include "splaytree.h"

struct TreeElement* root = NULL;
int count = 0;

// lägg in ett element i prioritetskön
void insert(int priority, char * value)
{
   // skapa det nya elementet
   struct TreeElement * element = malloc(sizeof(struct TreeElement));

   element->priority = priority;
   element->value = value;
   element->left = NULL;
   element->right = NULL;
   element->parent = NULL;
   element->id = ++count;

   // trädet är tomt
   if (root == NULL)
   {
      root = element;
   }

   // trädet är inte tomt -- hitta rätt plats för elementet som ska läggas
       in
   else
   {
      insert_binary_tree(element, root);

      while (root != element)
      {
         splay(element);
      }
   }
}

void insert_binary_tree(struct TreeElement * element, struct TreeElement *
     focus)
{
   // sätt in till vänster
   if (element->priority < focus->priority)
   {
      if (focus->left == NULL)
      {
         focus->left = element;
         element->parent = focus;
      }

      else
      {
         insert_binary_tree(element, focus->left);
      }
```

```
54      }
55
56      // sätt in till höger
57      else
58      {
59         if (focus->right == NULL)
60         {
61            focus->right = element;
62            element->parent = focus;
63         }
64
65         else
66         {
67            insert_binary_tree(element, focus->right);
68         }
69      }
70   }
71
72   void splay(struct TreeElement * x)
73   {
74      struct TreeElement * p = x->parent;
75
76      // zig
77      if (p == root)
78      {
79         // höger
80         if (x == p->left)
81         {
82            struct TreeElement * b = x->right;
83
84            root = x;
85            x->parent = NULL;
86
87            x->right = p;
88            p->parent = x;
89
90            p->left = b;
91            if (b != NULL) { b->parent = p; }
92
93            return;
94         }
95
96         // vänster
97         if (x == p->right)
98         {
99            struct TreeElement * b = x->left;
100
101           root = x;
102           x->parent = NULL;
103
104           x->left = p;
105           p->parent = x;
106
107           p->right = b;
108           if (b != NULL) { b->parent = p; }
109
110           return;
```

```
111         }
112      }
113
114      if (p != root)
115      {
116         struct TreeElement * g = p->parent;
117
118         // zig-zig höger
119         if (g->left != NULL && g->left->left == x)
120         {
121            struct TreeElement * b = x->right;
122            struct TreeElement * c = p->right;
123
124            if (g == root)
125            {
126               root = x;
127               x->parent = NULL;
128            }
129
130            else
131            {
132               if (g == g->parent->left) { g->parent->left = x; }
133               if (g == g->parent->right) { g->parent->right = x; }
134
135               x->parent = g->parent;
136            }
137
138            x->right = p;
139            p->parent = x;
140
141            p->left = b;
142            if (b != NULL) { b->parent = p; }
143
144            g->left = c;
145            if (c != NULL) { c->parent = g; }
146
147            p->right = g;
148            g->parent = p;
149
150            return;
151         }
152
153         // zig-zig vänster
154         if (g->right != NULL && g->right->right == x)
155         {
156            struct TreeElement * b = x->left;
157            struct TreeElement * c = p->left;
158
159            if (g == root)
160            {
161               root = x;
162               x->parent = NULL;
163            }
164
165            else
166            {
167               if (g == g->parent->left) { g->parent->left = x; }
```

```
168             if (g == g->parent->right) { g->parent->right = x; }
169
170             x->parent = g->parent;
171          }
172
173          x->left = p;
174          p->parent = x;
175
176          p->right = b;
177          if (b != NULL) { b->parent = p; }
178
179          g->right = c;
180          if (c != NULL) { c->parent = g; }
181
182          p->left = g;
183          g->parent = p;
184
185          return;
186       }
187
188       // zig-zag vänster-höger
189       if (p->parent->left != NULL && p->parent->left->right == x)
190       {
191          struct TreeElement * b = x->left;
192          struct TreeElement * c = x->right;
193          struct TreeElement * d = g->right;
194
195          if (g == root)
196          {
197             root = x;
198             x->parent = NULL;
199          }
200
201          else
202          {
203             if (g == g->parent->left) { g->parent->left = x; }
204             if (g == g->parent->right) { g->parent->right = x; }
205
206             x->parent = g->parent;
207          }
208
209          p->right = b;
210          if (b != NULL) { b->parent = p; }
211
212          g->left = c;
213          if (c != NULL) { c->parent = g; }
214
215          g->right = d;
216          if (d != NULL) { d->parent = g; }
217
218          x->left = p;
219          p->parent = x;
220
221          x->right = g;
222          g->parent = x;
223
224          return;
```

```
225         }
226
227         // zig-zag höger-vänster
228         if (p->parent->right != NULL && p->parent->right->left == x)
229         {
230            struct TreeElement * b = x->right;
231            struct TreeElement * c = x->left;
232            struct TreeElement * d = g->left;
233
234            if (g == root)
235            {
236               root = x;
237               x->parent = NULL;
238            }
239
240            else
241            {
242               if (g == g->parent->left) { g->parent->left = x; }
243               if (g == g->parent->right) { g->parent->right = x; }
244
245               x->parent = g->parent;
246            }
247
248            p->left = b;
249            if (b != NULL) { b->parent = p; }
250
251            g->right = c;
252            if (c != NULL) { c->parent = g; }
253
254            g->left = d;
255            if (d != NULL) { d->parent = g; }
256
257            x->right = p;
258            p->parent = x;
259
260            x->left = g;
261            g->parent = x;
262
263            return;
264         }
265      }
266   }
267
268   // hämtar det element med högst prioritet
269   char * fetch()
270   {
271      if (root == NULL) { return NULL; }
272
273      // gå kanterna till höger och hitta hörnet längst till höger
274      struct TreeElement * focus = root;
275
276      while (focus->right != NULL)
277      {
278         focus = focus->right;
279      }
280
281      struct TreeElement * p = focus->parent;
```

```
282 ||
283 ||    // ta bort elementet från trädet
284 ||    if (p == NULL)
285 ||    {
286 ||       root = focus->left;
287 ||       if (focus->left != NULL) { focus->left->parent = NULL; }
288 ||    }
289 ||
290 ||    else
291 ||    {
292 ||       if (focus->right == NULL && focus->left == NULL)
293 ||       {
294 ||          focus->parent->right = NULL;
295 ||       }
296 ||
297 ||       else if (focus->left != NULL && focus->right == NULL)
298 ||       {
299 ||          focus->parent->right = focus->left;
300 ||          focus->left->parent = focus->parent;
301 ||       }
302 ||
303 ||       else
304 ||       {
305 ||          printf("error removing!\n");
306 ||       }
307 ||
308 ||       // spreta ut
309 ||       while (root != p)
310 ||       {
311 ||          splay(p);
312 ||       }
313 ||    }
314 ||
315 ||    return focus->value;
316 || }
317 ||
318 || // används för att felsöka prioritetskön
319 || void debug_print_queue()
320 || {
321 ||    struct TreeElement * focus = root;
322 ||
323 ||    FILE *f = fopen("dot.dot", "w");
324 ||    fprintf(f, "digraph\n{\n");
325 ||    debug_print_queue2(f, focus);
326 ||    fprintf(f, "}");
327 ||    fclose(f);
328 ||
329 ||    system("/usr/bin/dot -Tsvg -o dot.svg dot.dot");
330 || }
331 ||
332 || void debug_print_queue2(FILE * f, struct TreeElement * focus)
333 || {
334 ||    fprintf(f, "\t%d [label = %d]\n", focus->id, focus->priority);
335 ||
336 ||    if (focus->parent != NULL) { fprintf(f, "\t%d -> %d [label = P]\n", focus
      ||         ->id, focus->parent->id); }
337 ||
```

```
338     if (focus->left != NULL)
339     {
340         fprintf(f, "\t%d -> %d [label = L]\n", focus->id, focus->left->id);
341 
342         debug_print_queue2(f, focus->left);
343     }
344 
345     if (focus->right != NULL)
346     {
347         fprintf(f, "\t%d -> %d [label = R]\n", focus->id, focus->right->id);
348 
349         debug_print_queue2(f, focus->right);
350     }
351 }
```

I) Tomt skal

```c
#include <stdlib.h>

void insert(int priority, char * value)
{

}

char * fetch()
{
   return NULL;
}

void debug_print_queue()
{

}
```

J) Rådata från experimentuppställning

Rådata från körningarna av testfallen finns att hämta på följande WWW-adress.
https://people.kth.se/~mpola/grundutbildning/II1304/experiment/data.tar.gz