Avdelning för datavetenskap

Anders Nilsson och Haris Trbakovic

Optimering av listhantering i telekomapplikation

Optimization of list handling in a telecommunication application

Examensarbete (15hp)
Datavetenskap

Datum:	2008-01-17
Handledare:	Kerstin Andersson
Examinator:	Martin Blom
Ev. löpnummer:	C2008:01

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Optimering av listhantering i telekomapplikation

Anders Nilsson och Haris Trbakovic

© 2008 Anders Nilsson, Haris Trbakovic och Karlstads universitet

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Anders Nilsson

Haris Trbakovic

Godkänd, 080117

Handledare: Kerstin Andersson

Examinator: Martin Blom

Sammanfattning

Denna rapport beskriver ett examensarbete som gjordes åt TietoEnator. Målet med examensarbetet var att optimera listhantering i en telekomapplikation. För att utföra detta användes programmeringsspråket C. Uppgiften omfattade fyra delar som bestod av att:

- Undersöka den nuvarande implementationen
- Föreslå förbättring till den nuvarande implementationen
- Implementera den utvalda lösningen
- Utföra mätningar samt dokumentera resultaten

Undersökningen av den nuvarande implementationen gjordes genom flera olika mätningar för att kunna se exekveringstiderna på operationerna. Förslag till förbättringar till den nuvarande implementationen gjordes genom analysering av olika datastrukturer. Implementationen av den utvalda lösningen blev ett AVL-träd och en hjälplista som är en länkad lista. Mätningar och dokumentation gjordes genom att mäta tider på den gamla och nya implementationen där man sedan jämförde exekveringstiderna. Resultatet blev en klar förbättring som t.ex. sökning av ett objekt vilket blev i snitt 3000 ggr snabbare. Det vi har gjort i detta examensarbete kommer TietoEnator att implementera och utveckla vidare.

Optimization of list handling in a telecommunication application

Abstract

This report describes an examination project that was done for TietoEnator. The goal with the project was to optimize list handling in a telecommunication application using the programming language C. The project consists of four parts which are the following:

- To examine the current implementation
- To suggest improvement of the current implementation
- To implement the chosen solution
- To perform measurements and document the results

The examination of the current implementation was done through a several different measurements in order to see the execution times for every operation. Suggestions for improvement of the current implementation were done through analysis of different data structures. The implementation of the chosen solution is an AVL-tree and a help list which is a linked list. Measurements and the documentation were done by measuring the times of the old and the new implementation and thereafter comparing the execution times. In the results we could clearly see an improvement of the execution times for example search for an object was in the average case 3000 times faster. All that was done in this project will be used and developed further by TietoEnator.

Innehållsförteckning

1 Inledning 1

2 Introduktion till telekommunikation 3

2.1 TietoEnator AB 3

2.2 Signaleringsnät 4

2.3 Principer för SS7 Signalering 5

2.3.1 Inledning
2.3.2 Historia
2.3.3 Konstruktion/arkitektur och användningsområde för SS7
2.3.4 SS7 Protokollstack

2.4 Tidskomplexitet 11

3 Undersökning av problemet 13

3.1 Signaling Connection Control 13

3.2 Beskrivning av den nuvarande implementationen 15

3.3 Beskrivning av problemet 21

3.4 Test av nuvarande implementationen 21

3.5 Sammanställning av problemet 24

3.6 Mål 25

4 Förslag till förbättringar 27

4.1 Beskrivning av datastrukturer 27

4.1.1 Hashtabell
4.1.2 Array
4.1.3 Länkad lista
4.1.4 Skiplista
4.1.5 B-träd
4.1.6 BST
4.1.7 Splay-träd
4.1.8 AVL-träd
4.1.9 Röd-Svart-träd

4.2 Sammanställning av de lämpliga lösningarna på problemet 40

5 Implementation 43

5.1 De tre nya strukterna 43

5.2 De nya funktionerna 45

5.2.1 Funktionen för insättning
5.2.2 Funktionen för uppdatering
5.2.3 Funktionerna för sökning och hämtning
5.2.4 Funktionerna för borttagning
5.2.5 Funktion för sökning via roid

6 Resultat 51

6.1 Insättning 51

6.2 Uppdatering 52

6.3 Sökning och hämtning 54

6.4 Borttagning 55

6.5 Sökning via roid 57

7 Slutsats 59

Referenser 61

A Tabeller 62

Figurförteckning

Figur 1: TietoEnator företagsstuktur.[1] 3
Figur 2: Struktur för det vanliga telefonnätet...4
Figur 3: Data- och signaleringstrafik på samma länk (in- band signalering).[5] 6
Figur 4: Data- och signaleringstrafik på olika länkar (out of band signalering).[5] 6
Figur 5: Nätverksstruktur av SS7:s signalering. [4] 7
Figur 6: OSI-modell och SS7-modell. [7]..8
Figur 7: Kopplingen mellan SCC-server och SCC-admin. 14
Figur 8: Exempel på hur objekt kan vara kopplade i MTP3. 15
Figur 9: Exempel på en strukt som används i den nuvarande implementationen. 16
Figur 10: Skapa ett nytt objekt. 16
Figur 11: Insättning av objekt i listan. 17
Figur 12: Insättning av ett objekt i en lista...17
Figur 13: Gå igenom listan...18
Figur 14: Hitta, uppdatera och ta bort-objekt. 18
Figur 15: Hämta ledigt froid. 19
Figur 16: Första objektet i listan...20
Figur 17: Nästa objekt i listan. 20
Figur 18: Ta bort hela listan. 21
Figur 19: List-operationer på länkad lista. 22
Figur 20: Hitta objekt på roid. 24
Figur 21: Hashtabell 28
Figur 22: Array 29
Figur 23: Länkad lista 29
Figur 24: Skiplista 30
Figur 25: B-träd 31
Figur 26: Binärt sökträd 32
Figur 27: Splay-träd 32

Figur 28: Vänsterrotation av AVL-träd. 33
Figur 29: Högerrotation av AVL-träd...34
Figur 30: Första delen av högervänster rotation, av AVL-träd. 35
Figur 31: Resultat efter högervänster rotation, av AVL- träd. 35
Figur 32: Första delen av vänsterhöger rotation av AVL-träd. 36
Figur 33: Resultat efter vänsterhöger rotation..36
Figur 34: Röd-svart-träd som uppfyller röd-svart-villkoret. 37
Figur 35: Ommålning av röd-svarta träd 38
Figur 36: Resultat efter ommålningen av röd-svart träd...38
Figur 37: Rotation av röd-svart träd 39
Figur 38: Resultat efter rotation av röd-svart-träd..39
Figur 39: Strukt för AVL node..44
Figur 40: Strukt för hjälplistans nod. 45
Figur 41: Strukt som innehåller två pekare. 45
Figur 42: Insättning i listan 46
Figur 43: Insättning i listan utan minneskopiering. 47
Figur 44: Uppdatera i listan. Jämförelse mellan implementationena. 48
Figur 45: Funktionerna allokering och frigöring av minnet. 49
Figur 46: Test av insättning i listan. 52
Figur 47: Test av uppdatering av objekt i listan. 53
Figur 48: Sök ett objekt i listan på froid. 54
Figur 49: Test att hämta alla objekt i listan...55
Figur 50: Ta bort ett objekt i listan. 56
Figur 51: Ta bort alla objekt i listan. 56
Figur 52: Hämta ett objekt på roid utan hjälplistan. 57
Figur 53: Hämta ett objekt på roid med hjälplistan. 58

Tabellförteckning

Tabell 1: Betecknar på Tidskomplexitet.. 11
Tabell 2: Tidskomplexitet för de olika datastrukturerna.[9] 40
Tabell 3: Resultat på förbättringar. 59

1 Inledning

Detta examensarbete har utförts av Anders Nilsson och Haris Trbakovic på uppdrag av TietoEnator. Examensarbetet är en 15-högskolepoäng på Datavetenskapsprogrammet på Karlstads Universitet. Arbetet påbörjades 2007-08-29.

Det första mötet kring examensarbetet ägde rum på TietoEnator i augusti 2007. Vi fick en omfattande presentation om TietoEnator och en presentation om uppgiften för examensarbetet.

Det som vi fick i uppgift att göra var att förbättra sökningen på en lista i Signaling Connection Control SCC-admin-servern som används i SS7-protokollet som TietoEnator utvecklar åt Ericsson. Uppgiften omfattade fyra delar som består av att

- Undersöka nuvarande implementationen av listan och förstå hur resten av systemet utnyttjar sig av listan.
- Föreslå förbättringar till den nuvarande implementationen av listan.
- Implementera den utvalda lösningen.
- Utföra mätningar och dokumentera resultaten.

Vi har valt att beskriva de olika delarna i var sitt kapitel.

Inledningsvis i kapitel 2 ger vi en kort introduktion om TietoEnator samt en kort beskrivning av telekommunikation och SS7-protokollet. Syftet med detta kapitel är att läsaren ska kunna få förståelse för telekommunikation och SS7 och just de delar som omfattar projektet.

Vidare i kapitel 3 gör vi en undersökning av problemet där vi beskriver uppgiften samt den nuvarande implementationen och tester som vi har utfört. Vi ger också en sammanställning av problemet.

I kapitel 4 beskriver vi de datastrukturer som vi har undersökt under projektets gång. Vi gör också en sammanställning där vi tar upp vissa för- och nackdelar och tidskomplexitet för de olika datastrukturerna.

I kapitel 5 beskriver vi vår nya implementation och visar med hjälp av koden vilken förbättring vi har gjort. Vi beskriver även skillnaden mellan den nya och gamla implementationen.

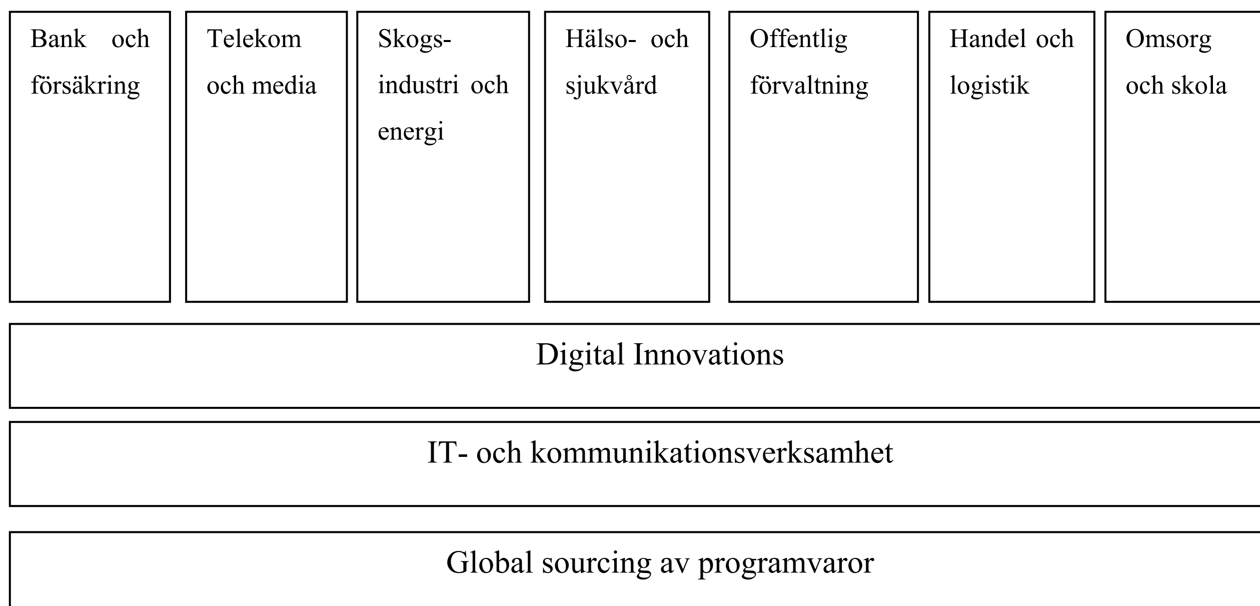
I kapitel 6 beskriver vi hur testerna är utförda. Vi gör också en jämförelse mellan exekveringstiderna för den gamla och nya implementationen. Avslutningsvis, i kapitel 7 gör vi en slutsats av projektet.

2 Introduktion till telekommunikation

I detta kapitel ger vi en kort introduktion av TietoEnator AB, tidskomplexitet, telekommunikation och SS7-protokollet som på uppdrag av Ericsson utvecklas av TietoEnator. Syftet med detta kapitel är att läsaren ska få en kort redogörelse för vad SS7-protokollet är och vad det används till så de enklare kan följa vårt projekt.

2.1 TietoEnator AB

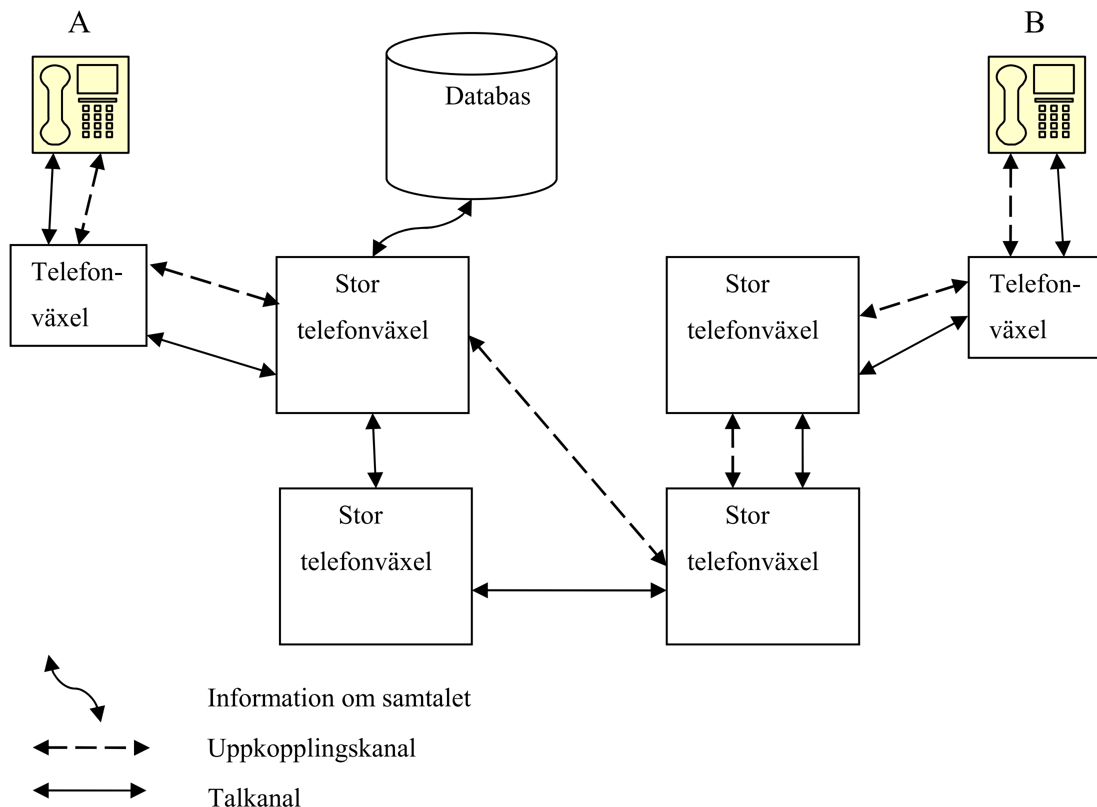
TietoEnator bildades 1999 då det finska företaget Tieto gick samman med det svenska Enator. Då hade Tieto cirka 6500 anställda och företaget Enator cirka 6000 anställda. I dag, 2007, har TietoEnator cirka 16000 anställda och är ett av Europas största företag inom IT. Företaget arbetar med konsultning och utveckling framför allt inom bank, telekom, skogsindustri, hälsa och sjukvård, se Figur 1. Inom telekommunikation samarbetar TietoEnator med Ericsson där de bl.a. har ett uppdrag att utveckla det plattformsoberoende protokollet Signal System Nummer 7 (SS7). [1]



Figur 1: TietoEnator företagsstuktur.[1]

2.2 Signaleringsnät

Telefonsystemet som vi använder idag dvs. vanlig telefoni inklusive mobiltelefoni består av flera telefonväxlar även kallade noder, se Figur 2. Telefonkommunikation går till på så sätt att A sänder en förfrågan till sin lokala telefonväxel om en förbindelse till mottagare B, som i sin tur skickar förfrågan vidare till en större telefonväxel som är kopplad till en databas. Databasen tar reda på var B befinner sig och skickar vidare den informationen tillbaka till den lokala telefonväxeln som sätter upp en uppkopplingskanal mellan A och B. När B svarar så sätts även en talkanal upp mellan A och B. Båda kanalerna behöver inte ta samma väg genom nätet.



Figur 2: Struktur för det vanliga telefonnätet.

De stora telefonväxlarna är direkt kopplade till andra stora telefonväxlar så om någon skulle gå ned så finns det andra som direkt kan ta över dennes uppgifter, se Figur 5. De stora telefonväxlarna använder sig av Connectivity Packet Platform (CPP), en plattform som kör protokollstack SS7. [2]

2.3 Principer för SS7 Signalering

Här tar vi upp något om principerna för SS7 och vi beskriver också hur de uppstod. Dessutom beskriver vi även SS7-protokollet kortfattat eftersom vårt arbetsuppdrag kommer att omfatta delar av SS7-protokollet. Informationen om SS7 är hämtat ur följande källor [3],[4],[5],[6],[7] och [8].

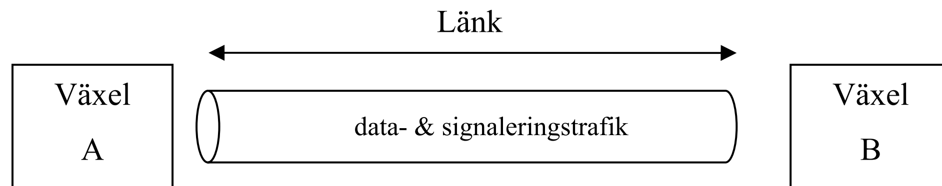
2.3.1 Inledning

SS7 eller det så kallade Signalerings System Nummer 7, även känd som C7 och CCS7, är en mängd protokoll som används inom telekommunikation. SS7:s huvuduppgift är att tillhandahålla kommunikationen i det publika telefonnätet, Public Switched Telephone Network (PSTN). Det tar bl.a. hand om uppkopplingen mellan samtal i hela landet, samt uppkopplingen mellan länder och den har även en central roll inom mobila nätverk.

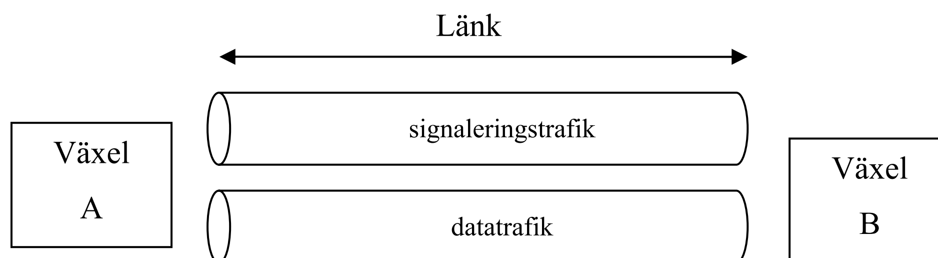
2.3.2 Historia

SS7–protokollet utvecklades av USA:s telekommunikationsbolag American Telephone & Telegraph AT&T [13]. SS7 utformades som standardprotokoll av Internationella teleunionen ITU[14]. Det finns även andra varianter på standarden. Tidigare protokoll som Signaling System Nummer 5 SS5[12] använde sig av var en gammal signaleringsform genom att den sände signaleringsinformation i samma kanal som data. Denna typ av signalering kallas in-band signalering, se Figur 3. Eftersom man hade en och samma kanal för signaleringsinformation och data fick man olika säkerhetsproblem, bl.a. att man lätt kunde ta sig in i systemet och påverka det. Ett annat problem var att in-band signalering var väldigt ineffektivt eftersom den sände signaleringsinformation och data i samma kanal. Under utvecklingen av SS7 löste man säkerhetsproblemet och förbättrade prestanda genom att man

införde olika kanaler, en för data och en för signaleringsinformation. Denna typ av signalering kallas för out of band signalering, *se Figur 4*.



Figur 3: Data- och signaleringstrafik på samma länk (in- band signalering).[5]

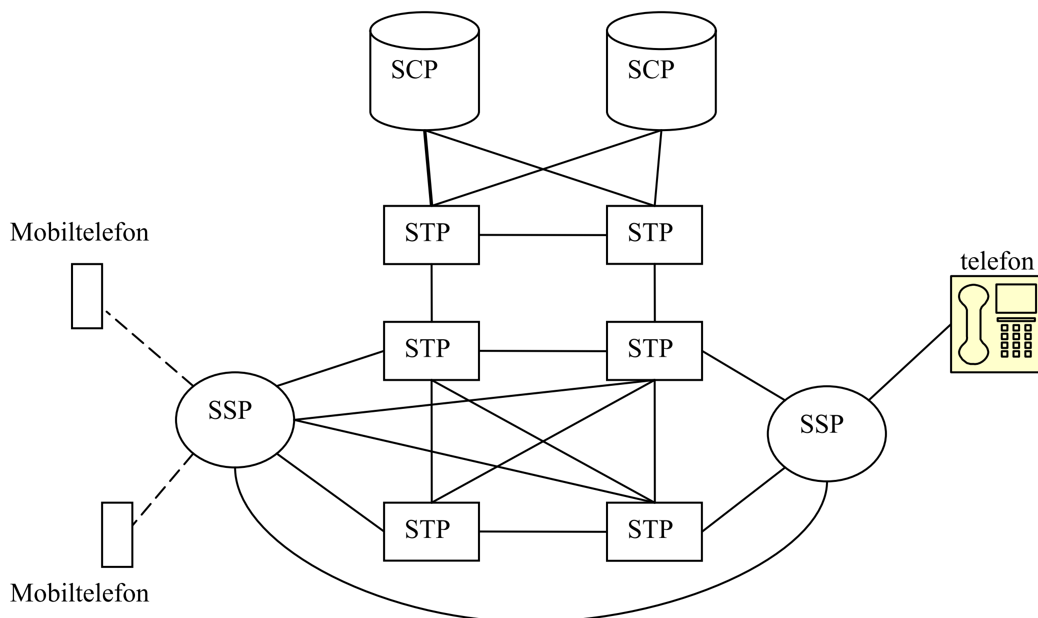


Figur 4: Data- och signaleringstrafik på olika länkar (out of band signalering).[5]

2.3.3 Konstruktion/arkitektur och användningsområde för SS7

SS7:s huvudsakliga uppgift är att ta hand om telefonsamtal över det publika telefonnätet. Eftersom detta kräver mycket av samkörning utförs allt detta av Integrated Services Digital Network (ISDN) User Part (ISUP)[7]. ISUP-meddelanden skickas längs hela uppkopplingsvägen mellan telefonväxlar och på så sätt håller telefonväxlarna reda på vart ett samtal kommer ifrån och vart man ska vidarebefordra det. En annan viktig roll som SS7 har är inom mobila nätverk samt i kopplingen mellan det publika telefonnätet och Internet. SS7:s nätverkskonstruktion består av flera länktyper och tre typer av signaleringsnoder, *se Figur 5*. De tre typerna av signaleringsnoder kallas för Service Switching Point (SSP), Signal Transfer Point (STP) och Service Control Point (SCP) och länktyperna kallas för (A, B, C, D, E och F)[4]. Varje signaleringsnod är sammankopplad med andra signaleringsnoder genom flera olika länkar för att öka stabiliteten i nätet. Beroende på vilken typ av signal som kommer att skickas på länken så används den typen av länk som passar bäst. Som de flesta signaleringsprotokoll är SS7 också uppbyggt av en lagerarkitektur. Varje lager har en speciell

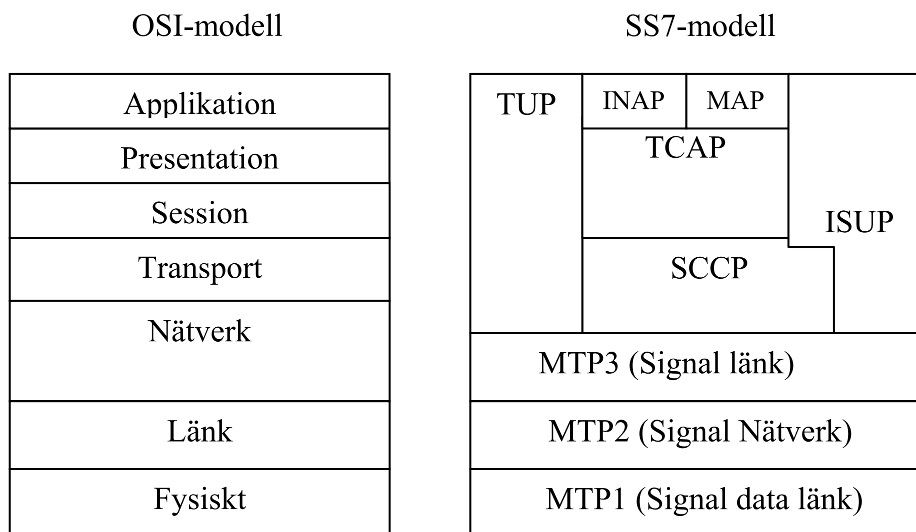
roll och ett ansvar. De 3 lägsta lagren bildar en så kallad Message Transfer Part (MTP). MTP använder sig av signaleringslänkar för att skicka meddelanden samt att leda dem till den förväntade destinationen. Alla lager som befinner sig på högre nivå än 3 har olika funktionalitet och implementeras efter behov av nätverket.



Figur 5: Nätverksstruktur av SS7:s signalering. [4]

2.3.4 SS7 Protokollstack

Funktioner för hårdvara och mjukvara för SS7:s protokollstack är uppbyggda med lagerarkitektur som motsvarar Open Systems Interconnection (OSI-modellen), *se Figur 6.* SS7:s protokollstack består av fyra lager. De första tre lagren i OSI- modellen tillhandahålls av Messege Transfer Part (MTP) i SS7- protokollet. MTP har hand om meddelanderouting. Lager fyra i SS7:s protokoll, innehåller många funktionsblock såsom Signaling Connection och Control Part (SCCP), Telephone User Part (TUP), Integrated Services Digital Network User Part (ISUP), Transaction Capabilities Application Part (TCAP), Intelligent Network Application Part (INAP) och Mobile Application Part (MAP). Detta lager motsvarar OSI-modellens lager 4 till 7.



Figur 6: OSI-modell och SS7-modell. [7]

Vi ska nu kortfattat beskriva de olika lagren i SS7-modellen som Figur 6 visar. Informationen om SS7 är hämtat ur följande källor [3],[4],[5],[6],[7] och [8].

Message Transfer Part

Lager 1, 2 och 3 kallas tillsammans för Message Transfer Part (MTP). Det lägsta lagret, MTP1 motsvarar det fysiska lagret i OSI-modellen. MTP2 motsvarar datalänklagret i OSI-modellen och MTP3 motsvarar nätverkslagret.

Message Transfer Part Nivå 1

MTP1-nivån som representerar det fysiska lagret har ansvar för uppkopplingen mellan olika SS7-signaleringspunkter och konverteringen av meddelanden till en elektronisk signal som sedan skickas över den fysiska länken.

Message Transfer Part Nivå 2

MTP2-nivån representerar datalänklagret. Den har till uppgift att tillhandahålla en pålitlig överföring av signalinformation mellan SS7:s signaleringspunkter. Detta utförs genom att all data som överförs exekveras för att se om det finns några fel och om det finns fel så rättas de om det är möjligt. MTP2 är också ansvarig för sammansättning av utgående meddelanden till paket som kallas för signaleringsenheter. Det finns tre typer av signaleringsenheter: Fill-in Signal Unit (FISU), Link Status Signal Unit (LSSU) och Message Signal Unit (MSU).

Message Transfer Part Nivå 3

MTP3-nivån representerar nätverkslagret. Den består av en funktion som tar hand om mottagna meddelanden och sänder dem vidare till de förväntande destinationerna. En annan funktion som MTP3 har är Network Managment. Den upprätthåller kontrollen av trafik-routing, länkar som har hand om trafiken och rättning av de eventuella fel som uppstår.

ISDN User Part (ISUP)

ISUP är protokollet som tar hand om uppkoppling och nedkoppling av samtal i det publika telefonnätet. ISUP används både till så kallade ISDN och icke-ISDN-samtal. En annan uppgift som ISUP tar hand om är att lägga ihop olika nätverk som t.ex. mobila nätverk och persondators nätverk till det publika telefonnätverket.

Telephone User Part (TUP)

Eftersom man har bytt ut TUP mot ISUP är det endast i några få delar av världen som t.ex. Kina man använder TUP. TUP är ett analogt protokoll som tar hand om uppkoppling och nedkoppling av telefonsamtal i det publika telefonnätet. Eftersom TUP är ett analogt protokoll så kan den endast sända informationen analogt, den digitala sändningen och dataöverföringen utförs av ett annat lager.

Signaling Connection and Control Part (SCCP)

SCCP befinner sig ett lager ovanför MTP-nivå 3 och den används för överföring av signaleringsinformation i kretskopplade och icke-kretskopplade nät. SCCP tar hand om sändning av Transaction Capabilities Application Part (TCAP) meddelanden och ser till att meddelandena kommer till den rätta databasen. De två principerna för dataöverföring som SCCP använder kallas för förbindelselös och förbindelseorienterad överföring. Förbindelseorienterad överföring innebär att man överför data först när man har upprättat en förbindelse mellan två noder. Förbindelselös överföring innebär att man överför data utan att någon förbindelse behöver upprättas. En annan viktig uppgift som SCCP har är routing av data genom att använda Global Title Translation (GTT).

Transaction Capabilities Application Part (TCAP)

TCAP används av applikationer som tar hand om informationen från databaser. Informationen som sänds kallas för TCAP-meddelanden. Dessa meddelanden utväxlar informationen mellan olika databaser som t.ex. databasförfrågningar och Short Message Service (SMS). Eftersom TCAP-meddelanden måste hitta rätt adress till databasen använder den SCCP för transport. Operations, Maintenance Administrative Part (OMAP) och Mobile Application Part (MAP) är exempel på applikationer som använder TCAP.

Mobile Application Part (MAP)

MAP:s huvuduppgift är att tillhandahålla service för användare av mobiltelefoni. MAP använder TCAP till att utföra olika operationer och SCCP för transport av data.

Intelligent Network Application Part (INAP)

INAP är ett signaleringsprotokoll som används i intelligent network (IN). IN är en serie av standarder som tillåter att man kan lägga till nya service i det befintliga nätverket med minsta möjliga uppgraderingskostnader. INAP använder TCAP till att utföra olika operationer och SCCP för transport av data till de förväntade destinationerna.

2.4 Tidskomplexitet

Tidskomplexitet beskriver hur en algoritm beter sig i teorin oberoende av hårdvaran och programspråket. Man betecknar tidskomplexiteten med big-O. Big-O används för att begränsa storleken på funktionen uppåt. Big-O betyder ordo och används för att beskriva algoritmens effektivitet. För att se exempel på vad som gäller för big-O se Tabell 1 .

0(1)	Konstant
0(log n)	Logaritmisk
0(n)	Linjär
$0(n^2)$	Kvadratisk
$0(n^2)$	Kubisk
$0(a^n)$	Exponentiell

Tabell 1: Betecknar på Tidskomplexitet..

I detta kapitel har vi gett en beskrivning av SS7-protokollet för att ge en bild av vad som finns omkring SCC. SCC-admin-server som vi kommer att förbättra list-implementationen på befinner sig i MTP3 och SCCP.

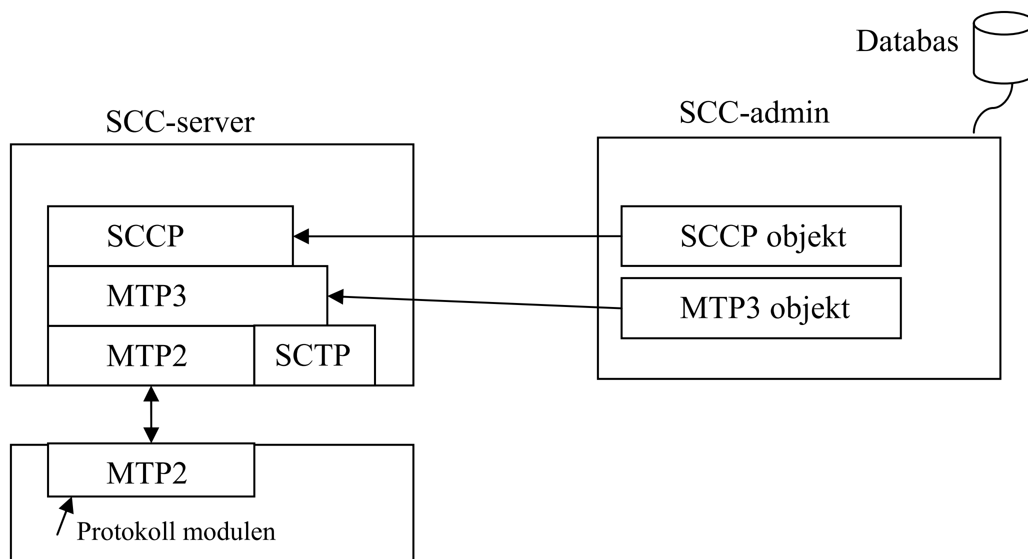
3 Undersökning av problemet

Nästa fas av projektet var att förstå och sätta sig in i problemet och få en bild av det. I detta kapitel beskriver vi uppgiften samt den nuvarande implementaionen.

Vi började med att analysera informationen som vi fick från TietoEnato för att se hur listan fungerar idag och vilka eventuella förbättringar man kan göra. För att undersöka hur den nuvarande listan fungerar skapade vi ett testprogram för att man ska kunna mäta tider på list-operationer. Tiderna vi fick gav oss en bild av hur listan beter sig idag och vilka eventuella förbättringar man skulle kunna göra på den.

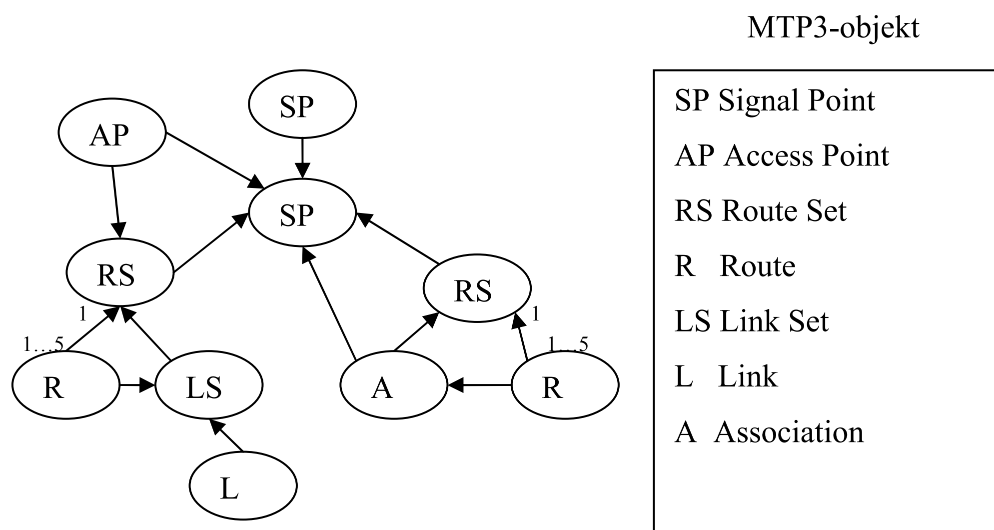
3.1 Signaling Connection Control

SCC befinner sig i de stora telefonväxlarna i CPP-plattformen[15]. De stora telefonväxlarna använder sig av CPP-plattformen för att köra SS7. På lager MPT3 och SCCP finns en SCC-server som styrs av SCC-admin[16]. SCC-servern är kopplad till SCC-admin. SCC-admin innehåller SCCP-objekt som i sin tur innehåller 9 olika objekt-typer, sen finns det ett MTP3-objekt som i sin tur innehåller 7 olika objekt-typer. Tillsammans har SCCP-objekten och MTP3-objekten 16 olika objekt-typer. De olika objekt-typerna ligger i listor, som kan innehålla allt från 1 till flera tusen objekt. Den största listan som används idag innehåller 2048 objekt. Listorna finns i SCC-admin. Dessa listor kommer vi att jobba med i vårt projekt.
Objekten som finns i listorna sparas i en databas vilken är kopplad till SCC-admin. Om SCC-admin skulle gå ner så kan den hämta objekten på nytt från databasen, se Figur 7.



Figur 7: Kopplingen mellan SCC-server och SCC-admin.

Objekten som finns i de olika listorna innehåller information för att styra SCC-servern. I objekten lagras ett unikt id som är Facode Resource Objekt Id (froid). Froid tilldelas till objektet av SCC-admin när objektet skapas. SCC-admin använder froid så att den kan identifiera objektet och utföra operationer på objektet. Objekten är sorterade efter froid i listan. Det finns också ett alternativt id som heter Resource Objekt Id (roid). Roid tilldelas av SCC-server i efterhand. När objektet skapas i SCC-admin skapas det med roid 0 för att sedan få ett nytt roid av SCC-servern. SCC-server använder roid till att identifiera och utföra operationer på objektet. De övriga delarna i objektet kan variera beroende på vilken typ av objekt man har. De kan t.ex. vara kopplingar till andra typer av objekt. Kopplingarna mellan objekten är inte direkta utan objekten innehåller ett id som refererar till ett annat objekt. Objekt som är sammankopplade behöver inte ligga i samma lista i SCC–admin. Figur 8 visar hur olika MTP3-objekt är kopplade till varandra. SCCP-objekt är kopplade på liknade sätt.



Figur 8: Exempel på hur objekt kan vara kopplade i MTP3.

3.2 Beskrivning av den nuvarande implementationen

Den nuvarande implementationen består av en strukt och ett antal funktioner. Vi kommer att förklara med hjälp av delar av koden hur de fungerar och vilka nackdelar som finns med den nuvarande implementationen. Koden i detta avsnitt är inte komplett utan vi tar med endast delarna som utför operationer på listan.

Varje objekt som lagras i en lista innehåller ett unikt id samt en datapekare. Exempel på hur en strukt som används för att skapa ett objekt i nuvarande listan ser ut finns i Figur 9. Den unika strukt-medlemmens id som kallas för Facode Resource Objekt Id (froid) har samma värde under objektets levnadstid och används för att kunna sortera objekten. Objekten är sorterade i stigande ordning i listan för att underlätta sökningen. På så sätt kan man undvika att hämta strukt-medlemmen data för att avgöra vilket objekt man har. I strukt-medlemmen data lagras informationen om objektet. Informationen som lagras i strukt-medlemmen data finns beskriven i kapitel 3.1, men för att utföra projektet behöver man inte fördjupa sig i den. Det som är intressant med strukt-medlemmens data är att den innehåller ett annat id som kallas för roid. Roid skiljer sig lite från det vanliga id dvs. det så kallade froid. Roid tilldelas

av SCC-server i efterhand. När objektet skapas i SCC-admin skapas det med roid 0 för att sedan få ett nytt roid av SCC-servern. SCC-server använder roid så att den kan identifiera och utföra operationer på objektet. Roid behöver inte alltid vara unikt och dessutom kan det förändras under körning. Till sist består strukten av en nextpekare som pekar på nästa objekt i listan.

```
typedef struct linkList
{
  int id;                /* froid på objektet */
  char* data_p;          /* Data till froid */
  struct linkList *next; /* Pekare till nästa element i listan*/
} SccFroListElement_r;
```

Figur 9: Exempel på en strukt som används i den nuvarande implementationen.

Nu kommer vi att beskriva funktionerna som opererar på listan. Först beskrivs insättning, sedan ta bort, uppdatera, söka, hämta ledigt froid, hämtning av alla objekt och sist ta bort alla objekt i listan. Eftersom funktionerna hämta-första-objekt och hämta-nästa-objekt används alltid tillsammans för att hämta alla objekt, har vi valt att kalla dessa för hämtning av alla objekt.

```
Skapar ett nytt objekt:
nyttObjekt = (SccFroListElement_r *)
malloc(sizeof(SccFroListElement_r));

nyttObjekt->dataobjekt = (char *) malloc(froSize);
if (nyttObjekt->dataobjekt == NULL)
{
     free(nyttObjekt);
}
nyttObjekt->froid = set_froid;
memcpy(nyttObjekt->dataobjekt, nydata, froSize);
```

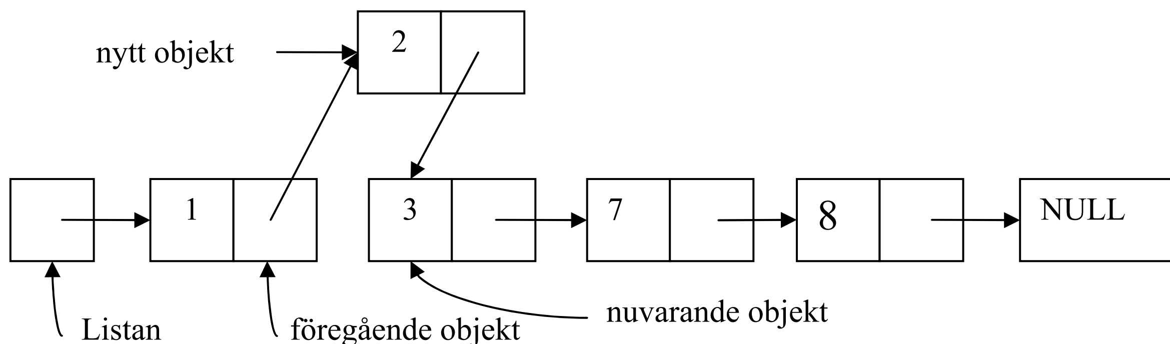
Figur 10: Skapa ett nytt objekt.

Insättning av ett objekt sker på så sätt att man först skapar ett objekt, se Figur 10. Sedan stegar man igenom listan tills man hittar den korrekta platsen med hjälp av froid, se Figur 11. Jämförelse mellan nuvarande objektets froid och froid som man vill sätta in görs och när man upptäcker att nästa froid är större än det man vill sätta in så sätter man in det nya objektet före det elementet som var större.

```
Insättning av objekt i listan
if((nuvarandeObjekt!= NULL) && (nyttObjekt->froid > nuvarandeObjekt
->froid))
{
   while((nuvarandeObjekt!= NULL) && (nyttObjekt->froid >
nuvarandeObjekt->froid))
   {
      föregåndeObjekt = nuvarandeObjekt;
      nuvarandeObjekt = nuvarandeObjekt->nästa;
   }
   föregåndeObjekt->nästa = nyttObjekt;
   nyttObjekt->nästa = nuvarandeObjekt;
}
else
{
    nyttObjekt->nästa = *listan ;
    *listan = nyttObjekt;
}
```

Figur 11: Insättning av objekt i listan.

Figur 12 visar hur insättningen ser ut med hjälp av pekare på föregående objekt och nuvarande objekt.



Figur 12: Insättning av ett objekt i en lista.

I listan sker insättningen av ett objekt oftast i slutet av listan och för att undvika att man måste gå genom hela listan så har man skapat en funktion append som en provisorisk lösning. Append gör en insättning av objekt i slutet av listan, men för att den ska fungera så behöver man ha en pekare till sista objektet i listan. Vid insättning och append görs även minnes kopiering som kostar många CPU cykler.

```
GåIgenomListan(listan, froid)
{
    SccFroListElement_r * nuvarandeObjekt = listan;

    while((nuvarandeObjekt!=NULL)&&(froid!=nuvarandeObjekt->froid))
    {
        föregåndeObjekt = nuvarandeObjekt; /*Används endast av ta bort
funktionen*/
        nuvarandeObjekt = nuvarandeObjekt->nästa;
    }
    if (nuvarandeObjekt!= NULL)
    {
           Här skiljer sig funktionerna ta bort, uppdatera och söka
           från varandra även funktions anropen ser olika ut.
           Skillnaden mellan de kan ses i koden nedanför.
    }
}
```

Figur 13: Gå igenom listan.

Ta bort, uppdatera och sök efter ett objekt fungerar på ungefär samma sätt dvs. man stegar igenom listan tills man hittar objektet med det sökta froid, se Figur 13 och Figur 14. Ett problem med dessa tre funktioner är att de måste gå igenom listan i snitt (listans längd)/2 för varje objekt de ska ta bort, söka eller uppdatera. Minneskopiering används i både uppdateringar och i söknings-funktionerna.

```
Sök efter ett objekt funktionen:

  memcpy(dataobjekt, nuvarandeObjekt->dataobjekt, froSize);

Uppdatera efter ett objekt funktionen:
   memcpy(nuvarandeObjekt->dataobjekt,dataobjekt, froSize);


Ta bort efter ett objekt funktionen:
   if (nuvarandeObjekt == listan)
   {
      listan = nuvarandeObjekt->nästa;
   }
   else
   {
       föregåndeObjekt->nästa = nuvarandeObjekt->nästa;
   }
   free(nuvarandeObjekt->dataobjektet);
   free(nuvarandeObjekt);
```

Figur 14: Hitta, uppdatera och ta bort-objekt.

Hämta ledigt froid funktionen används för att få ett unikt froid till varje objekt i listan.
Det gör den genom att funktionen går igenom listan tills den hittar ett froid som inte är upptagen. Dvs. att funktionen letar efter det först låga numret som inte används som froid av något objekt, se Figur 15. Tidskomplexitet för denna funktion är O(n).

```
int
hämtaLedigtFroid(listan)
{
    SccFroListElement_r * nuvarandeObjekt = listan;
    int ledigtfroid = 1;

    while((nuvarandeObjekt!=NULL)&&(temporär==nuvarandeObjekt->froid))
    {
        nuvarandeObjekt = nuvarandeObjekt->nästa;
        ledigtfroid = ledigtfroid + 1;
    }
    return ledigtfroid;
}
```

Figur 15: Hämta ledigt froid.

Först sätts en ledigtfroid variabel till 1 och jämförs med det första objektet som bör ha ett froid som är 1. Om det första objektets froid är 1 så ökas den ledigtfroid variabeln med 1 och jämförs med nästa objekt i listan som bör ha ett froid som är 2. Detta körs så länge nästa objekts froid är samma som den ledigtfroid variabeln och det nuvarande objektet är skilt från NULL. När den ledigtfroid variabeln och objektets froid skiljer sig åt returneras den ledigtfroid variabeln tillbaka. Här skiljer sig funktionen som hämtar ledigt froid åt från de övriga funktionerna genom att den returnerar resultatet dvs. det först lediga froid som den har hittat medan de andra funktionerna returnerar status på om funktionen har lyckats eller inte.

Hämtning av alla objekt i listan görs med två funktioner hämta-första- och hämta-nästa-objekt. Funktionen hämta-första-objekt hämtar första objektets froid och data genom att ta en kopia på dataobjektet som sedan skickas tillbaka till den funktionen som gjorde anrop, se Figur 16.

```
förstaObjektIlistan(listan, froid, dataobjekt, froSize)
{
    memcpy(dataobjekt, listan-> dataobjekt, froSize);
    froid = listan->froid;
}
```

Figur 16: Första objektet i listan.

Funktionen hämta-nästa-objekt används för att hämta objekt i resten av listan. Den behöver froid från funktionen hämtar-första-objekt för att på så sätt få veta vilket objekt som står på tur att bli hämtat. Funktionen hämta-nästa-objekt letar upp föregående froid i listan och ställer sig där och kopierar nästa objekts data samt froid som den kommer att använda vid nästa körning av funktionen, se Figur 17.

```
nästaObjektIListan(listan, froid, dataobjekt, froSize)
{
    /*Skapare en pekar nuvarandeObjekt som pekar på första elementet
i listan*/
    nuvarandeObjekt = listan;
    while ((nuvarandeObjekt!= NULL) && (froid != nuvarandeObjekt->
froid))
    {
        nuvarandeObjekt = nuvarandeObjekt->nästa;
    }

    if ((nuvarandeObjekt!= NULL) && (nuvarandeObjekt->nästa != NULL))
    {
        memcpy(dataobjekt, nuvarandeObjekt->nästa-> dataobjekt,
froSize);
        id = nuvarandeObjekt->nästa ->id;
}
```

Figur 17: Nästa objekt i listan.

Ett problem med hämta-nästa-funktion en är att den alltid måste börja från början av listan och stega sig igenom listan tills den kommer till det rätta objektet. Operationen som vi kallar hämta alla objekt använder minneskopiering. Operationen används för att söka efter ett objekt på roid.

Funktionen som tar bort hela listan börjar med att ta bort första objekt och forsätter på samma sätt tills hela listan är tom och returnerar status på om den har lyckas eller inte, se Figur 18.

```
while (nuvarandeObjekt != NULL)
{
        ta_bort = nuvarandeObjekt;
        nuvarandeObjekt = nuvarandeObjekt->nästa;

        free(ta_bort->dataobjekt);
        free(ta_bort);
}
```

Figur 18: Ta bort hela listan.

3.3 Beskrivning av problemet

Det stora problemet med projektet är att sökning efter ett specifikt listobjekt i SCC-admin sker i listan med linjär sökning dvs. med en tidkomplexitet som är O(n). I listorna finns det flera olika typer av listobjekt. Listorna kan bestå av allt från 1 till flera tusen objekt. För att hitta ett objekt så använder man sig av ett froid som identifierare när man söker igenom listan. Målen med att förbättra listan är att insättningen, sökningen via froid och sökningen via roid ska gå snabbare eftersom dessa funktioner utförs när systemet är belastat. En förbättring av insättningen kommer även att bidra till att omstartstid på systemet minskar. En annan fördel med förbättringen av listan är att vi skulle kunna få bort de provisoriska lösningar som finns i den nuvarande implementationen. Ett exempel på en provisorisk lösning är att ändra prioriteter på processer vilket gör att man kan vara inne i systemen längre tid och undvika att bli utkastad innan man är klar med objektet.

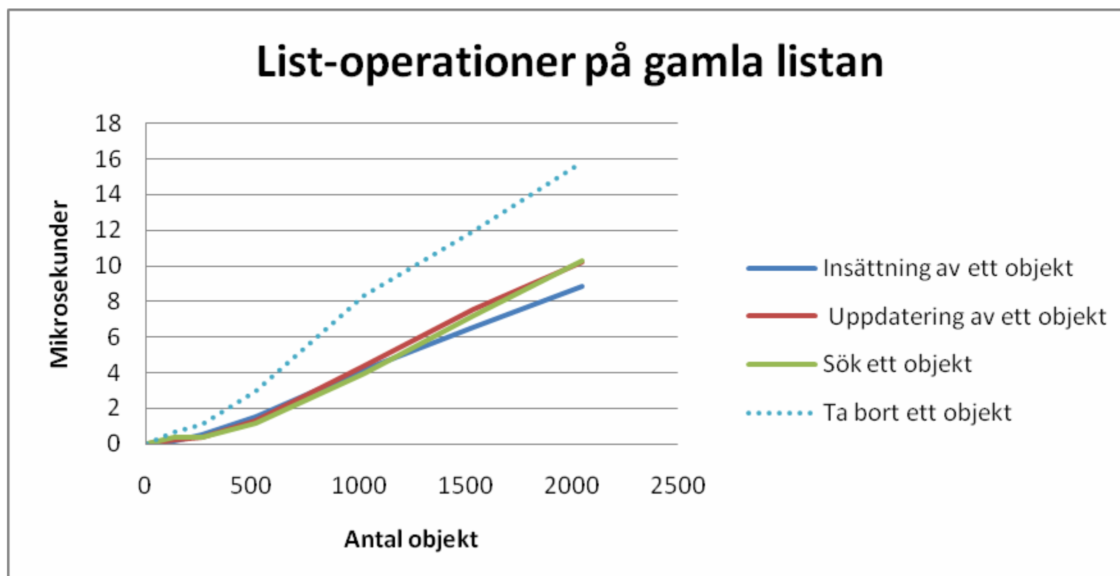
3.4 Test av nuvarande implementationen

Vi skapade ett testprogram för att kunna mäta tiden på de olika list-operationerna. Testprogrammet fungerar så att vi skickar in fyra listor som vi fyller med följande storlekar 512, 1024, 1536 och 2048 objekt. Sedan mäter vi tiden det tar att utföra operationerna på alla objekt i listan. Testprogrammet beskrivs utförligt i kapitel 6. Tider som vi fick på de olika list-operationerna kan ses i diagrammet, se Figur 19. För att se mätvärdena som används för diagrammet se bilaga A. I diagrammen som vi visar i detta kapitel har vi flera mätpunkter

mellan 0 och 512 för att få en fullständig kurva från punkt 0 till 2048. Mätpunkterna mellan 0 och 512 finns inte med i tabellerna i bilagan A.

Diagrammet visar snittiden för ett objekt i mikrosekunder för funktionen insättning, uppdatering, sökning och bortagning av ett objekt. Vi valde att inte ta med i diagrammet funktioner ta bort- och hämta alla objekt i listan eftersom de utför en operation på hela listan till skillnad från de andra funktioner som gör det på ett objekt.
Tidskomplexitet för insättning, uppdatering, sökning och bortagning av ett objekt blir O(n). Anledningen att kurvorna i diagrammet inte är helt linjära beror på att testerna är utförda på en PC med operativtsystem Linux. Eftersom vi mätte tider i Linux-miljö varierar tiderna lite p.g.a. att Linux kör flera processer samtidigt. Testerna skulle från första början göras i CPP-miljö där endast en process körs i taget vilket skulle ge oss betydligt bättre kurvor. På grund av tidsbrist fick vi nöja oss med testerna från Linux-miljö.

För insättningsfunktionen har vi valt att testa sämsta fallet som är insättning i slutet av listan. Anledningen till att vi testade sämsta fallet är att insättningen i den nuvarande listan ofta sker i slutet av listan. Tiden som vi får under mätningen delar vi med antalet objekt för att få snittiden för insättning av ett objekt.



Figur 19: List-operationer på länkad lista.

På funktionen uppdatera ett objekt testar vi hur lång tid det tar att hitta och uppdatera alla objekt i listan. Den tiden vi får delar vi med antalet objekt i listan för att få en snittid för att uppdatera ett objekt.

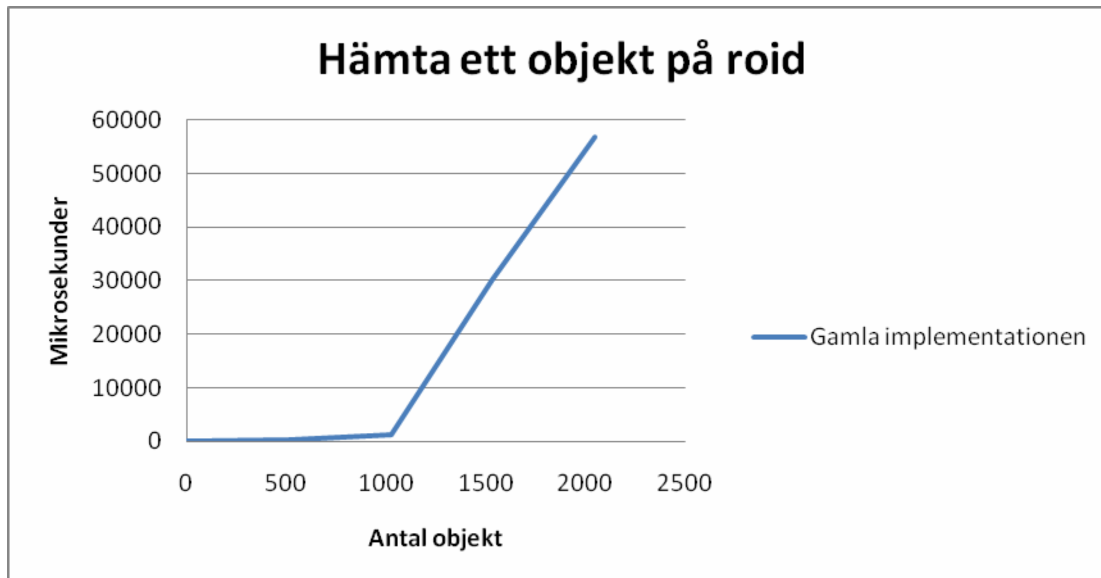
Testet för funktionen att söka ett objekt i listan utförde vi genom att mäta tiden för att söka alla objekt i listan. Tiden vi fick delade vi med antalet objekt i listan för att få snittiden för att söka ett objekt.

Testet för ta bort ett objekt i listan gjorde vi genom att mäta sämsta fallet. Detta innebär att vi tar bort ett objekt som ligger sist i listan. Vi börjar med att ta bort objekt bakifrån från listan dvs. sista sedan näst sista objektet i listan osv. tills listan är tom. Eftersom vi tar bort sista objektet i listan får vi en snittid för att ta bort ett objekt i listan. Tiden vi får delar vi med antalet objekt för att få snittiden för bortagning av ett objekt.

För funktionen som tar bort hela listan mätte vi tiden det tar att ta bort alla objekt i listan. För denna funktion finns det inget bästa eller sämsta fall. Tidkomplexiteten för den funktionen är O(n) eftersom den går genom listan endast en gång för att ta bort alla objekten.

Vi mätte också tiden det tar att hämta alla objekt i listan. Operationen som hämtar alla objekt utförs på så sätt att den hämtar alla objekt i listan utan att användaren behöva ange froid för varje objekt i listan. Eftersom den måste gå igenom listan en gång för varje objekt den ska hämta, har den en tidkomplexitet som är $O(n^2)$.

Ett annat test vi gjorde var att vi sökte efter roid och mätte tiden. Roid kan man inte söka på direkt. Eftersom roid ligger i dataobjektet måste man hämta dataobjektet först för att sedan kunna titta i den för att se vilket roid objektet har. För att söka på roid måste man använda sig av operationen hämta alla objekt i listan.
Vi mätte tiden det tar att söka efter alla roid som finns i lista. Tiden vi fick delade vi med antalet objekt för att få snittiden för att söka ett roid, se diagram i Figur 20. Kurvan i diagrammet stämmer inte helt med teorin eftersom testerna är utförda på en PC med operativtsystem Linux.



Figur 20: Hitta objekt på roid.

I diagrammet kan man se att sökningen tar väldigt långt tid jämfört med de andra funktionerna i diagrammet som finns i Figur 19. Anledningen till det är att hämtning av ett objekt på roid utförs med en tidkomplexitet som är $O(n^2)$. Eftersom vi använder operationen hämta alla objekt som har en tidkomplexitet $O(n^2)$, gör vi detta för varje roid, vilket gör att vi måste gå genom listan ytterligare en gång. Då har vi en tidkomplexitet som är $O(n^3)$ för att hämta alla objekt på roid som vi sen delar med antalet objekt i listan för att få snittiden för hämtning av ett objekt på roid. Detta gör att vi slutligen får en tidkomplexitet som är $O(n^2)$ för hämtning av ett objekt på roid.

3.5 Sammanställning av problemet

Efter att vi hade mätt tiderna på listoperationerna kunde vi konstatera att funktionen insättning-, uppdatering-, sökning- och bortagning-av ett objekt har en tidskomplexitet som är $O(n)$. Bortagning av alla objekt har en tidkomplexitet $O(n)$. Hämtning av alla objekt och hämtning av ett objekt på roid har en tidkomplexitet $O(n^2)$. En annan sak som vi har märkt är att listoperationer sådana som insättning, uppdatering, sökning och hämtning av alla objekt i listan använder minneskopiering som är ineffektivt och kostsamt i CPP miljö enligt TietoEnators tidigare mätningar.

3.6 Mål

Syftet med detta examensarbete är att undersöka olika datastrukturer och hitta en datastruktur som eventuellt kan ge en förbättring på söktiderna för ett specifikt objekt. Exempelvis kan implementationen av en datastruktur som har tidskomplexitet O(log n) eller O(1) vara ett förslag till förbättringar. En annan förbättring vi vill uppnå är att få bort minneskopiering vilket är kostsamt i CPP–miljö enligt TietoEnators tidigare mätningar.

4 Förslag till förbättringar

Nästa del i projektet var att analysera kravspecifikationen. Efter analysen gjorde vi en sammanställning tillsammans med TietoEnator. Sammanställningen gjordes för att lättare kunna förstå vilka förändringar som måste göras med listan för att förbättra den. Eftersom vi vill kunna jämföra den gamla listan och den nya implementationen, måste vi använda samma funktionsanrop. Vi bestämde oss för att titta på olika alternativ på listimplementeringar.

Vi insåg ganska snabbt att kraven som TietoEnator hade på listimplementationen inte kunde lösas med hashtabell eller länkade listor som var våra första alternativ. Vi var tvungna att titta på andra datastrukturer som kan användas för att få ner söktiderna i listan. Binära träd verkade vara det vi behövde för att få ner tiderna i listan. Eftersom de flesta operationer i binära träd görs på O(log n). Efter att ha analyserat hur träddatastrukturer fungerar, märkte vi att de kunde tillfredställa TietoEnators krav. Vi bestämde oss för att implementera ett balanserat binärt träd. Om det inte skulle vara tillräckligt bra, skulle vi även titta på hur vi skulle kunna gå vidare med andra typer av balanserade träd eller om vi var tvungna att byta datastruktur.

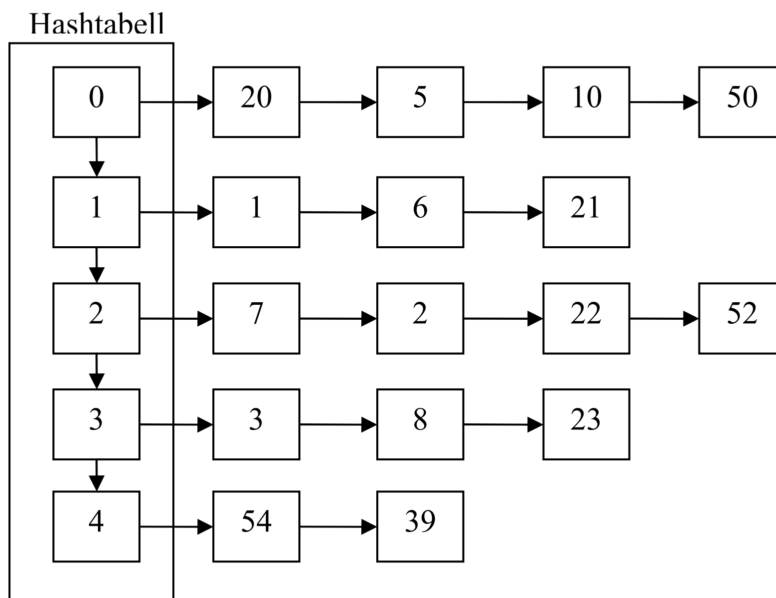
4.1 Beskrivning av datastrukturer

Detta delkapitel tar upp de olika datastukturer som vi har undersökt under projektets gång. Informationen om de olika datastrukturerna är hämtad ur följande källor [9],[10] och [11].

4.1.1 Hashtabell

Hashtabell är en array av en bestämd storlek som innehåller element. En hashtabell fungerar på så sätt att man associerar en så kallad nyckel till ett värde. Detta görs genom att man gör en implementation av en hashtabell som är mellan 0 och (tabellstorlek – 1) stor, därefter mappar man nyckeln till ett nummer mellan 0 och (tabellstorleken – 1) som placeras in i den anvisade platsen. Mappning av nyckel och värdet görs med hjälp av en hashfunktion. Det bästa skulle vara om man kunde ha en hashfunktion som alltid räknar ut och försäkrar sig till att t.ex. två nycklar får olika platser i hashtabellen. Eftersom man har ett bestämt antal platser i en hashtabell, är det omöjligt att få fram en så bra hashfunktion som försäkrar sig om att alla nycklar får olika platser. Det man försöker göra istället är att man försöker hitta en så bra

hashfunktion som möjligt, som fördelar nycklarna någorlunda jämnt mellan platserna i hashtabellen. [9]



Figur 21: Hashtabell

Det som man betraktar som problem vid en implementation av en hashtabell är just när man ska välja en hashfunktion och bestämma vad man ska göra när två nycklar kopplas till samma värde. När en så kallad kollision inträffar dvs. när två nycklar kopplas till samma värde, är ett alternativ att koppla ihop dem i en länkad lista, se Figur 21. Ett annat alternativ är att göra hashtabellen dubbel så stor.

4.1.2 Array

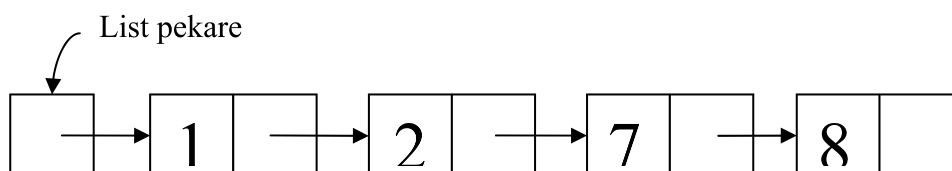
Array är en datastruktur som innehåller en samling av element. Elementen är oftast av samma typ och ligger efter varandra i minnet. Alla element refereras av ett index. För att komma åt de olika elementen använder man sig av index, se Figur 22. [9]

Värde	1	2			5		7	8	9	10
Index Värde	0	1	2	3	4	5	6	7	8	9

Figur 22: Array

4.1.3 Länkad lista

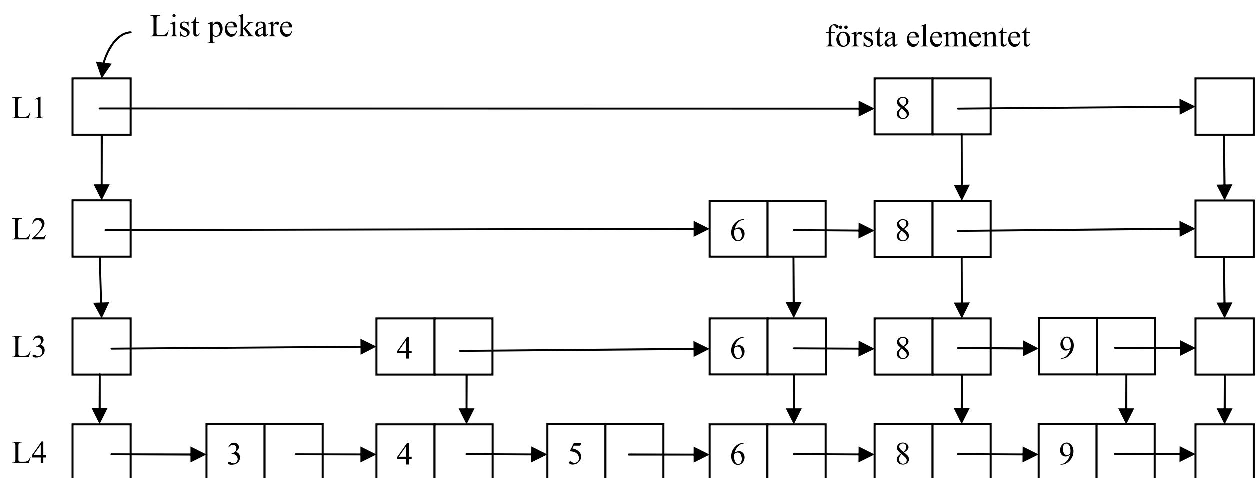
En länkad lista är en datastruktur som består av en sekvens element. Varje element innehåller någon typ av datafält och en pekare dvs. referensen som pekar/refererar till nästa eller föregående element. En länkad lista är dynamisk dvs. den växer i storlek efter behov och den kan lagra vilken typ som helst av data. I början av listan har man en list pekare som pekar på första elementet, se Figur 23. [9]



Figur 23: Länkad lista

4.1.4 Skiplista

Skiplista är en relativt ny form av liststruktur som skapades av William Pugh. En skiplista är uppbyggd av parallella länkade listor, som är sammankopplade för att ge en snabbare sökning jämfört med en vanlig länkad lista.[11] De flesta operationer som t.ex. sökning och insättning utförs på tidskomplexiteten O(log n) med sämsta fall O(n). Med skiplistan får man snabbare åtkomst till element jämfört med länkad lista med kostnaden av flera pekare.

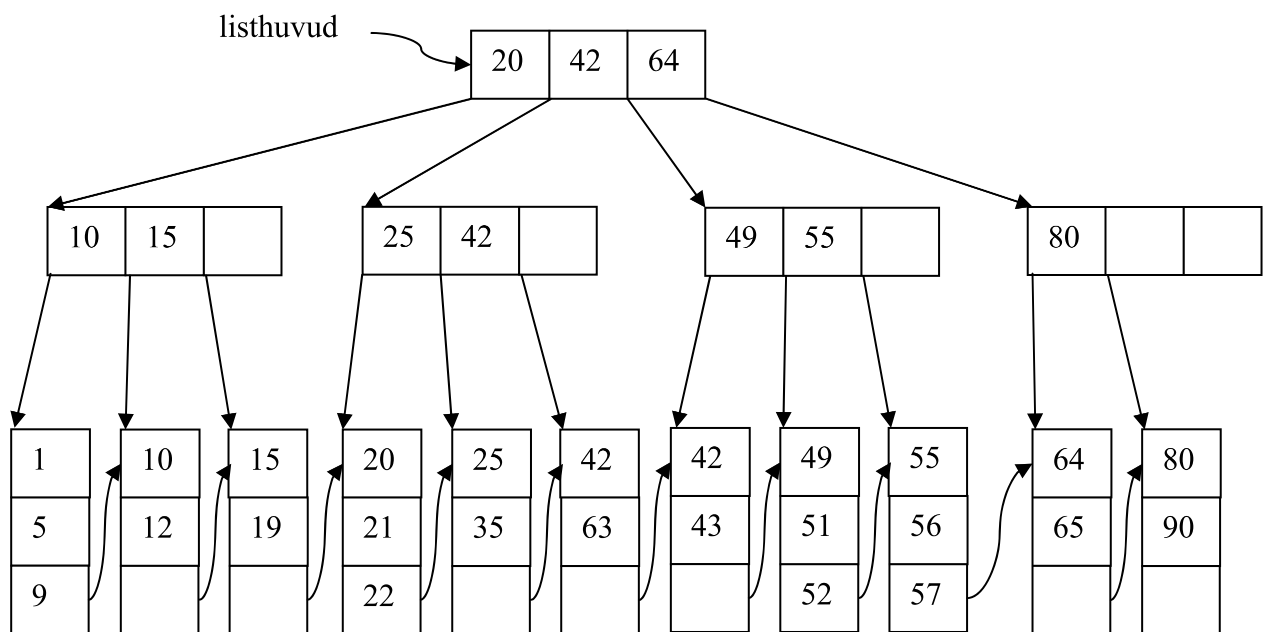


Figur 24: Skiplista

Sökning i algoritmen fungerar så att man alltid börjar i översta listan (L1), se Figur 24. Exempel: sökning med söknyckel 5 går till på så sätt att man jämför första elementets söknyckel 8 med 5. Första elementets söknyckel är större än 5. Då går man tillbaka till listpekaren och går vidare till nästa lista (L2) och jämför 5 och 6. 5 är mindre än 6. Då går man tillbaka till listpekaren (L2) och går ner till nästa lista (L3). I denna lista jämförs 5 och 4. 5 är större än 4. Då går man till nästa element och jämför 6 och 5. 5 är mindre än 6. Då går man tillbaka till element 4 och går ner ett steg till element 4 som ligger i nästa lista (L4). Sen jämförs nästa söknyckel dvs. 5 med 5. Rätta elementet har hittats.

4.1.5 B-träd

Ett B-träd är ett träd med noder som innehåller flera element som är av förbestämd storlek se, Figur 25. De inre noderna det vill säga de noder som inte är på understa nivån i trädet används som söknycklar för att hitta element snabbt. All data lagras i noderna på den nedersta nivån i trädet som kallas löv. B-träd är ett balanserat träd som inte behöver ombalanseras så som andra balanserade träd vid insättning och borttagning av element. B-träd används ofta för att hantera stort antal noder som inte ryms i det interna minnet utan behöver lagra noder och data på ett sekundärt minne t.ex. databaser.[9]

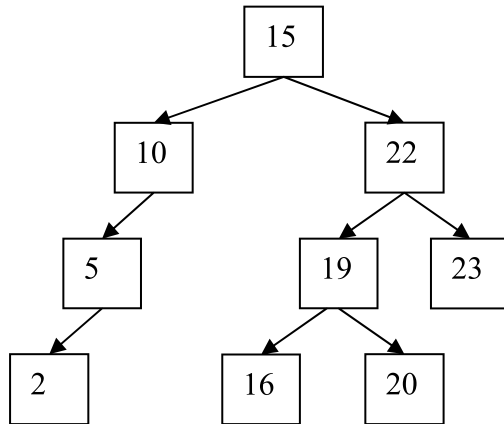


Figur 25: B-träd

Exempel: Om man ska söka efter 21, går sökning till på så sätt att man börjar i den översta noden och jämför 21 med första söknyckeln som är 20. 21 är större än 20. Då går man vidare till nästa söknyckel i noden som är 42 och gör samma jämförelse. Eftersom söknyckeln 21 är mindre än 42 går man ner en nivå i trädet och gör samma sak i den noden och jämför 21 med 25. 21 är mindre än 25. Då går man till nästa nod och jämför med nästa elements söknyckel dvs. 20 med 21. 21 är större än 20. Man går till nästa söknyckel i noden som är 21 dvs. det sökta elementet.

4.1.6 BST

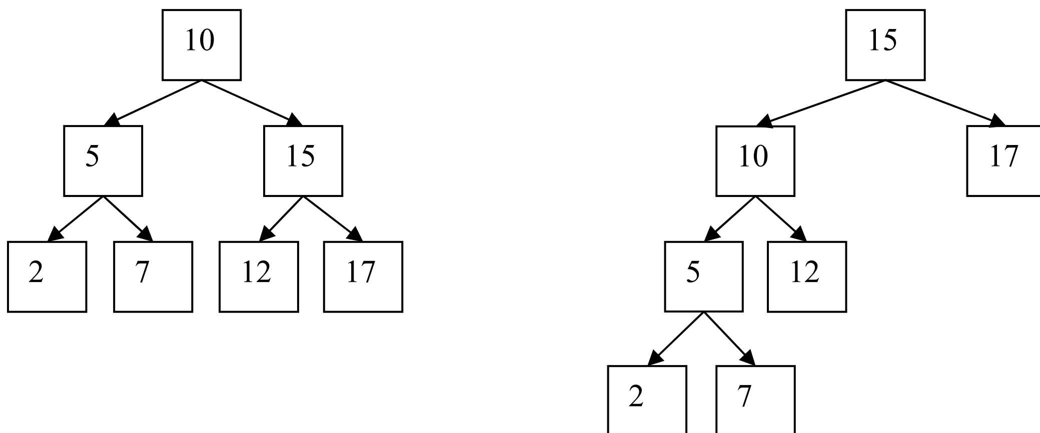
Ett binärt sökträd är en datastruktur där varje nod har högst två barn. Egenskaper som BST har är att varje element innehåller ett värde. Varje värde som är mindre än rotens värde finns i vänstra subträdet och varje värde som är större än rotens värde finns i högra subträdet. Exempel på BST finns i Figur 26.[9]



Figur 26: Binärt sökträd

4.1.7 Splay-träd

Splay-träd är ett binärt träd som har egenskapen att senast hämtade elementet kan hämtas enkelt eftersom det har flyttats upp till roten, se Figur 27. För att flytta upp elementet till roten, använder man sig av Adelson-Velskii and Landis (AVL) rotationer, (se avsnittet om AVL-träd). På så sätt ändras trädstrukturen vid varje sökning. De vanliga operationerna görs med tidskomplexiteten O(log n) med sämsta fall O(n). Implementation av splay-träd är enkel jämfört med AVL och röd-svart-träd som också är ett självbalanserat binärt träd.[9]



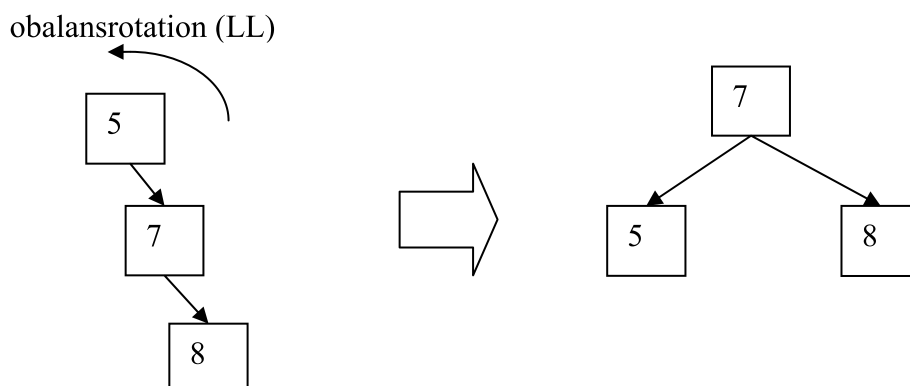
Figur 27: Splay-träd

Sökning i det vänstra trädet i Figur 27 efter element 15 ger trädet till höger. Man börjar med vanlig binär sökning tills man finner elementet man söker efter. Efter att man har hittat det, börjar man rotera trädet med AVL rotationer så att elementet som man sökte når roten.

4.1.8 AVL-träd

Ett AVL-träd är ett binärt sökträd med ett balansvillkor. AVL-träd är identiska med BST förutom att i ett AVL-träd för varje nod i trädet, får höjden av vänster subträd och höger subträd skilja med 1, 0 eller -1. Varje gång man ska kontrollera om trädet är balanserat räknar man ut en så kallad balansfaktor av en nod. Det man räknar på för att få fram balansfaktorn av en nod är höjden av högra subträdet minus höjden av vänstra subträdet. Resultatet man får från den beräkningen kallas för balansfaktor och så länge den är 1, 0 eller -1 så anses trädet balanserat. I alla andra fall så har man obalans, vilket gör att trädet måste balanseras. Balansering av trädet görs med så kallade rotationer. Man har fyra olika fall av rotationer vid obalans i ett AVL-träd. Dessa rotationer kallas för vänster rotation (LL), höger rotation (RR), vänsterhöger rotation (LR) och högervänster rotation (RL). [9]
Nu kommer vi att beskriva de olika rotationerna.

Trädet i Figur 28. har obalans i nod 5. Eftersom nod 5:s högra subträd har höjd 2 och vänstra subträd har höjd 0 ger det en balansfaktor lika med 2 som är större än 1, vilket leder till obalans. För att balansera trädet görs en vänsterrotation (LL).

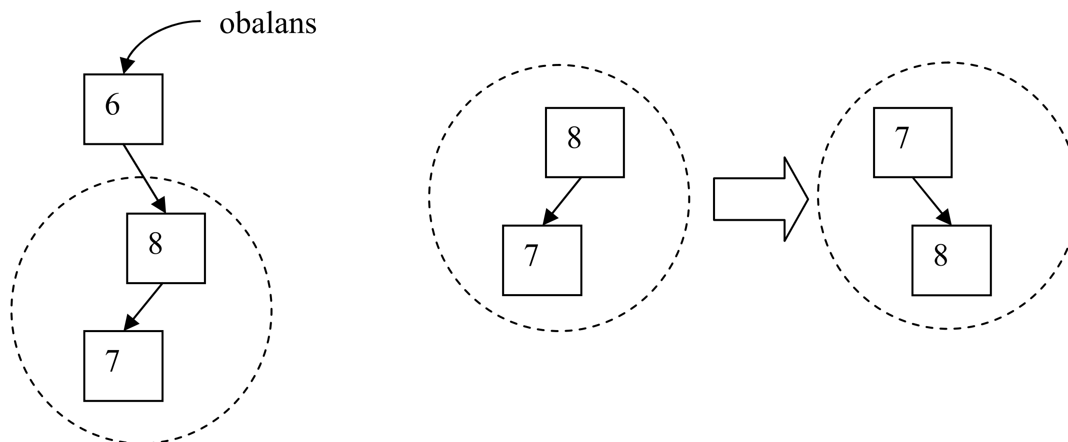


Figur 28: Vänsterrotation av AVL-träd.

Ett träd som ser ut som i Figur 29 har obalans i nod 6. Det blir obalans eftersom höjden av höger subträd och vänster subträd skiljer sig med -2. Detta ger en balansfaktor -2 som leder till obalans. För att balansera trädet görs en högerrotation.

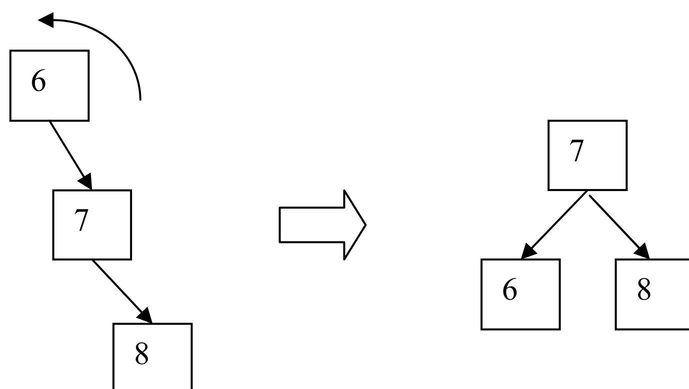
Figur 29: Högerrotation av AVL-träd.

I Figur 30 visas obalans i nod 6. För att balansera trädet i detta fall krävs en dubbel rotation. Först gör vi en rotation på högra subträdet.



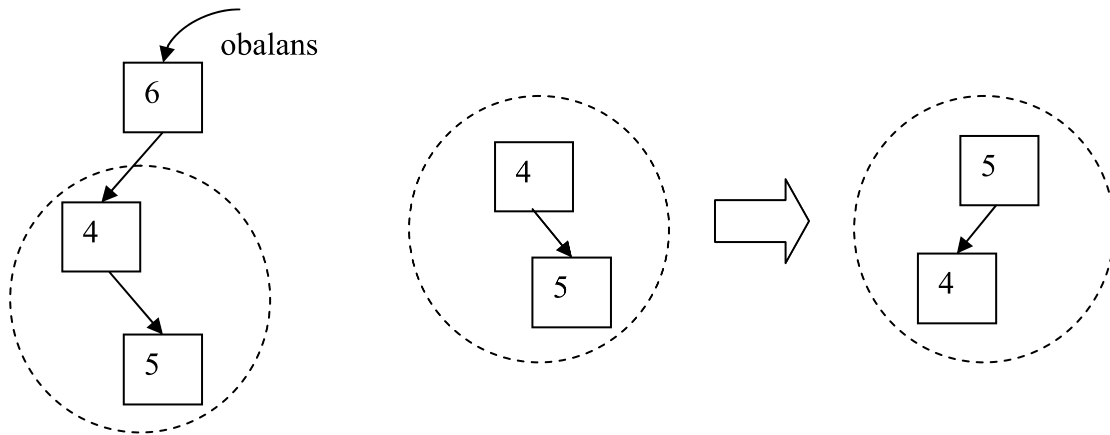
Figur 30: Första delen av högervänster rotation, av AVL-träd.

Efter den utförda rotationen av högra subträdet får vi ett träd som ser ut som i Figur 31.
För att få trädet balanserat utförs ytterligare en vänsterrotation.



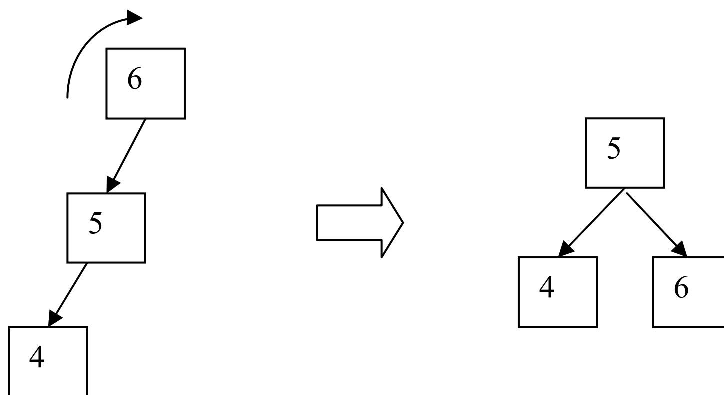
Figur 31: Resultat efter högervänster rotation, av AVL- träd.

Ett träd som ser ut som i Figur 32 har obalans i nod 6. För att balansera trädet i detta fall krävs en dubbel rotation. Först gör vi en rotation på vänstra subträdet.



Figur 32: Första delen av vänsterhöger rotation av AVL-träd.

Efter den utförda rotationen av vänstra subträdet får vi ett träd som ser ut som i Figur 33. Får att få trädet balanserat utförs en högerrotation.

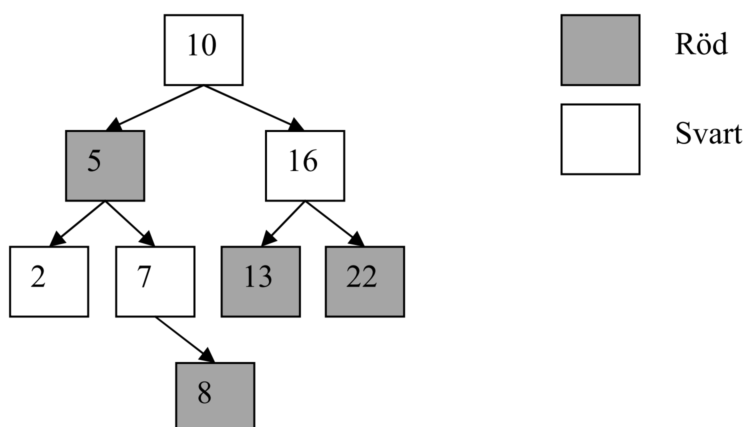


Figur 33: Resultat efter vänsterhöger rotation.

4.1.9 Röd-Svart-träd

Ett röd-svart-träd är ett binärt sökträd som är balanserat. För att ett röd-svart-träd ska hålla sig balanserat måste det uppfylla det så kallade röd-svarta-villkoret, se Figur 34. [10] Detta innebär att:

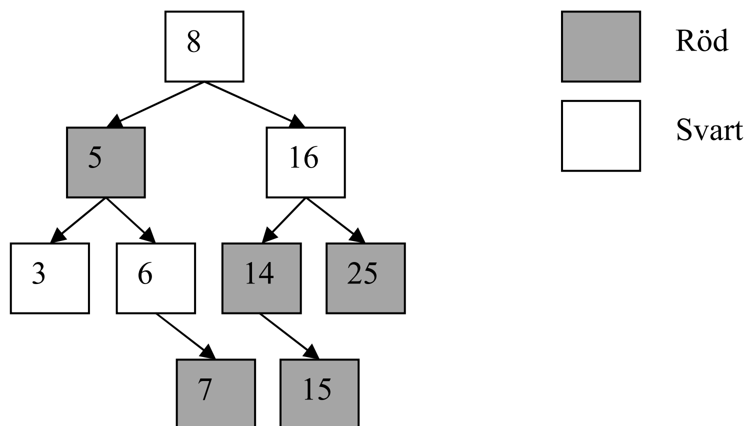
1. Varje nod måste vara röd eller svart.
2. Roten är alltid svart.
3. Om en nod är röd, då måste barnen till den vara svarta.
4. Varje väg från roten ner till ett löv måste innehålla samma antal svarta noder.



Figur 34: Röd-svart-träd som uppfyller röd-svart-villkoret.

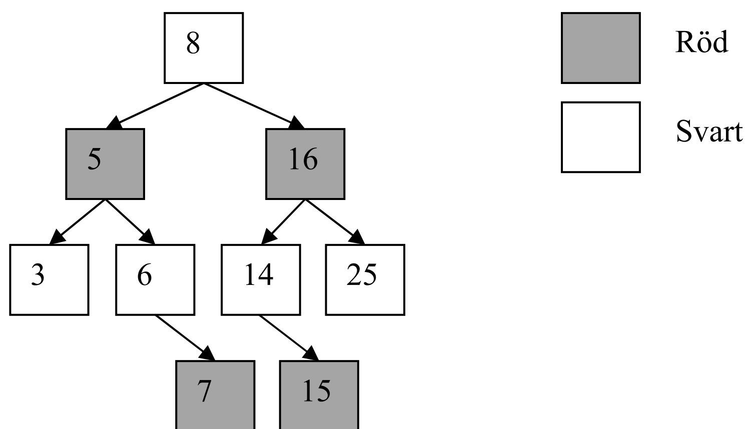
Insättningen i det röd-svarta trädet går till på så sätt att en ny nod som sätts in som ett löv färgas alltid röd. Efter att insättningen har gjorts, är insättningen klar om föräldern till den nya noden är svart. Om föräldern till den nya insatta noden skulle vara röd, har vi då två noder efter varandra som är röda, vilket bryter mot villkor 3. Då ommålas den nya noden eller så görs en rotation för att få bort på varandra följande röda noder.

För ett exempel, på när man har två röda noder efter varandra och hur man löser det med en så kallad ommålning, se Figur 35.



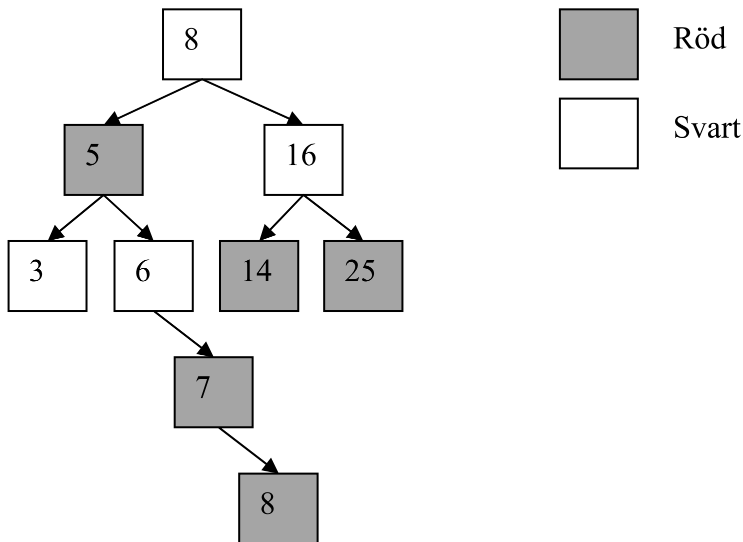
Figur 35: Ommålning av röd-svarta träd

Vid insättning av 15 får vi två röda noder efter varandra. Detta löser man genom ett man målar om 16 till röd och 14 och 25 till svart, se Figur 36.



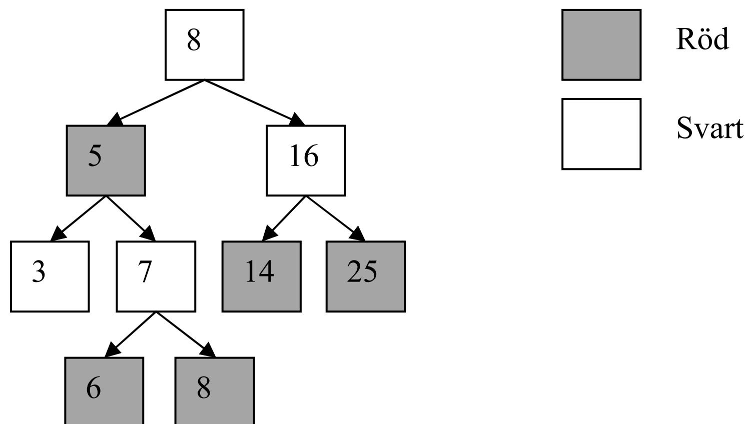
Figur 36: Resultat efter ommålningen av röd-svart träd

För ett exempel, på när man har två röda noder efter varandra och hur man löser det med en rotation, se Figur 37.



Figur 37: Rotation av röd-svart träd

Vid insättning av nod 8 får vi två röda noder efter varandra. Då gör vi en enkelrotation (som för AVL-träd) av noder 6, 7 och 8. Nod 7 målas om till svart och blir förälder till nod 6 och 8 som målas om till rött, se Figur 38.



Figur 38: Resultat efter rotation av röd-svart-träd

4.2 Sammanställning av de lämpliga lösningarna på problemet

I detta delkapitel har vi valt att sammanställa de olika datastrukturer som vi gick igenom under projektets gång och som är beskrivna i kapitel 4.1. Vi tittade också på olika fördelar och nackdelar med de olika datastrukturerna för att enklare kunna välja ut två algoritmer till projektet. Vi har valt att jämföra de olika datastrukturer med tidskomplexitet vilket är oberoende av miljö. Tidskomplexitet ger oss en snabb indikation på hur de olika datastrukturerna beter sig för stora mängder av element.

Datastruktur	**Insättning**		**Borttagning**		**Sökning**	
	Snitt	sämstafallet	snitt	sämstafallet	snitt	sämstafallet
Länkad lista	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
Array	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Hashtabell	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)
Skiplist	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)
B-träd	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
BST	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)
Splay-träd	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)
AVL-träd	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
R&B-träd	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

Tabell 2: Tidskomplexitet för de olika datastrukturerna.[9]

Efter sammanställningen av de olika datastrukturerna i Tabell 2 kunde vi konstatera att i sämsta fallet beter sig länkad lista, skiplista, BST och splayträd med en tidskomplexitet som är O(n). Vi konstaterade ganska tidigt att de har en tidskomplexitet som är densamma som den nuvarande implementationen, dvs. länkad lista, och eftersom förbättringen inte skulle bli så stor valde vi att inte gå vidare med dem.

Array verkar väldigt bra till en början eftersom array allokerar minne i följd och på så sätt kan man få direkt access till objekt via froid. Anledningen att vi inte valde array är just att array kräver att man allocker minnet statiskt. I vårt projekt lagrar vi allt från 1 till 2048 objekt vilket gör att en arrayimplementation inte skulle vara bästa lösningen eftersom vi skulle allockera minnet som inte skulle utnyttjas helt.

Hashtabell har tidskomplexitet som är O(1) för de flesta operationer. Problemet med hashtabell är att vissa operationer i sämsta fall kan bli så illa som O(n). Exempel på en operation som i sämsta fall kan ha tidskomplexitet som är O(n) är sökning. Denna operation har i normalfallet en tidskomplexitet som är O(1). När en så kallad kollision inträffar måste man göra omsökning vilket gör att vi får en söktidskomplexitet som är O(n).[9] En annan nackdel med hashtabell är att man måste beräkna en ny hashnyckel och hasha om alla värden när man vill öka antalet platser i hashtabellen. Detta medför att vi får en dålig tidskomplexitet för insättningen. I vårt projekt vet vi enligt TietoEnator att värdet på roid som man använder under insättningen och sökningen kan variera mycket. Variationen på roid leder till att det kan bli svårt att skapa en optimal hashfunktion som undviker kollisioner.[9] Pga. tidsbrist skapade vi inte några hashfunktioner för att testa det. Eftersom det i vårt projekt är viktigt med en förbättring av sökning och insättning tog vi beslutet att inte implementera en hashtabell.

B-träd och balanserade binära sökträd har i sämsta fall en tidskomplexitet som är betydligt bättre än den nuvarande implementationen, se Tabell 2. Anledningen till att vi tog beslutet att inte implementera B-träd var att vi inte hade tid att utföra tester för att se hur mycket minne B-träd kräver i CPP-miljö. Enligt teorin fick vi veta att B-träd används först och främst för stora mängder data som inte längre får plats på RAM-minnet, t.ex. databaser.[9] En annan anledning var att B-träd har större noder än övriga balanserade sökträd som vi har tittat på. Noden innehåller flera objekt som måste sökas igenom sekventiellt vilket gör att den har tidskomplexitet på $O(\log_m n)$ där m är antal objekt i noden. [9]

AVL- och röd-svart-träd har samma tidskomplexitet nämligen O(log n). Ett AVL-träd ger bättre söktidskomplexitet då det är bättre balanserade. Man kan visa att höjden av ett AVL-träd är ungefär 1.44log(n+2) – 0.328,[9] som i praktiken är lite mer än log(n). Höjden av ett röd-svart-träd är ungefär i snitt 2log(n+1).[9] Ett röd-svart-träd har lite bättre tidskomplexitet vid insättning och borttagning eftersom de inte behöver balanseras lika ofta som ett AVL-träd. Noderna i röd-svart-träd tar mer plats i minnesutrymmet än noderna i ett AVL-träd.

Efter en övervägning bestämde vi oss för att implementera AVL-träd, eftersom teorin visade att en implementation av ett AVL-träd skulle kunna ge oss bättre söktidskomplexitet som är viktigt.[9] Sökningen görs oftare när systemet är hårt belastat medan borttagning sker ofta när systemet inte är så hårt belastat. Insättning är också viktigt att få ned tiderna på eftersom det också sker när systemet är hårt belastat. En implementation av ett AVL-träd kommer enligt teorin eventuellt att ge oss insättningskomplexitet som är lite sämre än röd-svart-träd men den kommer att bli betydligt bättre än den nuvarande implementationens insättning som är O(n).[9] Eftersom AVL-träd har mindre nod storlek än röd-svart-träd samt att implementationen av ett AVL-träd är enklare valde vi att gå vidare med AVL-träd.[9]

5 Implementation

I detta kapitel kommer vi att beskriva skillnaden mellan den nya och den gamla implementationen. Vi kommer även att i mera detalj beskriva delar av de funktioner där man kan se vilken förbättringar vi har gjort.

I den nya implementationen har vi försökt att få användargränssnittet utåt att se ut som den gamla implementationen. Anledningen till att vi har valt att lösa uppgiften på detta sett är att klienten inte ska behöva göra stora ändringar i anropet till den nya implementationen. Det som klienten behöver tänka på i den nya implementationen är att inte längre skicka med storleken på dataobjektet eftersom dataobjektet inte hanteras längre med minneskopiering. Bakom användargränssnittet har vi gjort stora förändringar där vi bl.a. har implementerat ett AVL-träd. AVL-trädet har vi implementerat istället för den gamla implementationen som var en länkad lista. I AVL-trädet är objekten sorterade efter froid och innehåller dataobjektet.

Den nya implementationen innehåller även en hjälplista som är en länkad lista. Hjälplistan används för att hantera roid. Den innehåller froid, roid och är inte sorterad. Hjälplistan har vi valt att lägga till i den nya implementationen eftersom vi ville få ytterligare förbättringar på söktiden för ett roid. Genom att man har froid och roid separat i hjälplistan så går sökningen snabbare eftersom man inte längre behöver hämta dataobjektet för att se vilket värde på roid objektet har.

Den nya implementationen av ett AVL-träd och hjälplistan består av tre strukter och ett antal funktioner. Vi kommer nu med hjälp av delar av koden, att beskriva hur den nya implementationen skiljer sig från den gamla. Koden i detta avsnitt är inte komplett utan vi tar endast med delar som vi anser behövs för att förklara skillnaden.

5.1 De tre nya strukterna

Nu kommer vi att beskriva de tre strukterna som används i den nya implementationen. Den nya implementationen skiljer sig från den gamla genom att vi har ersatt den gamla strukten linkList se Figur 9, med en ny strukt som heter AVL-Node. Strukten AVL-Node används för

att skapa objekten i ett AVL-träd. Den nya strukten finns i Figur 39. AVL-Node strukten innehåller struktmedlemmen id som är froid. I struktmedlemmen data lagras informationen om objektet. Dessa två struktmedlemmar är desamma som de första två struktmedlemmarna i strukten linkList som vi beskrev i kapitel 3.3, se Figur 9. Struktmedlemmen Left innehåller en pekare till vänstra objektet i trädet medan struktmedlemmen Right innehåller en pekare till högra objektet i trädet. Till sist består strukten av struktmedlemmen height som anger höjden på trädet.

```
typedef struct AvlNode
{
    U32             id;        /* id of element  */
    char*           data  ;    /* Data of FRO    */
    AvlTree         Left;      /* strukt AVLNode */
    AvlTree         Right;     /* strukt AVLNode */
    int             Height;
}SccFroListElement_r;
```

Figur 39: Strukt för AVL node

Under vårt projekt behövde vi göra en förbättring på sökningen av ett objekt via roid eftersom man ibland vill kunna söka efter ett objekt via roid istället för froid. Efter att vi hade implementerat ett AVL-träd i den nya implementationen testade vi att göra sökningen av ett objekt via roid. Detta gjordes genom att man hämtade alla dataobjekt i AVL-trädet för att sedan kunna titta vilket roid dataobjektet har. Sökningen i ett AVL-träd dvs. den nya implementationen var betydligt bättre än den gamla, se kapitel 6.5, Figur 52. I den gamla implementationen utförde man sökningen av ett objekt via roid på samma sätt dvs. man hämtade alla objekt för att se vilket värde på roid objektet har. För att hämta alla objekt var man tvungen att gå in i listan och hämta ett objekt och sedan på nytt gå in och hämta nästa objekt, vilket gjorde att sökningen var ineffektiv. Detta ville vi utveckla vidare genom att flytta ut roid från dataobjektet vilket eventuellt skulle leda till en ytterligare förbättring eftersom då skulle man inte behöva gå in i listan ännu gång för att utföra sökning via roid. Lösningen blev en implementation av en hjälplista som är en länkad lista. Varje objekt som lagras i listan innehåller ett unikt froid, roid och en nextpekare som pekar på nästa objekt i listan. Strukten som används för att skapa ett objekt i hjälplistan finns i Figur 40. Vi tittade även på ett alternativ där vi skulle använda ett AVL-träd istället för en hjälplista för sökning via roid. Eftersom vi i vårt projekt fick informationen om roid på ett ganska sent stadium

valde vi att välja ut hjälplistan och gå vidare med den. AVL-träd skulle eventuellt kunna ge ännu bättre sökningstider men det är större risk att det blir fel vid implementationen. Detta skulle leda till mer tester vilket vi pga. tidsbrist inte hade möjlighet att göra.

```
typedef struct linkList
{
   U32 froId;             /* froId of element */
   U32 roId;              /* roId of element */
   struct linkList *next;
} SccRoListElement_r;
```

Figur 40: Strukt för hjälplistans nod.

Vi skapade också en strukt som innehåller två pekare. En pekare håller reda på ett AVL-träd och den andra en hjälplista. Strukten har vi endast för att göra ett bättre användargränssnitt och för att den nya implementationen inte ska skilja sig så mycket från den gamla. I den gamla implementationen skickas ett inargument som är en pekare till en länkad lista. I den nya implementationen måste vi skicka två pekare som inargument dvs. en pekare till AVL-träd och en till hjälplistan. För att skicka två pekare som ett argument har vi skapat en strukt för att lagra AVL-träd och hjälplistan. Dessa pekare skickar vi sedan som ett argument till funktionerna. På detta sätt behöver inte användaren veta om att det finns två pekare. Exempel på en strukt som innehåller två pekare finns i Figur 41.

```
typedef struct
{
    SccFroListElement_r *froList; /* AVL-träd   */
    SccRoListElement_r * roList;  /* hjälplista */
}SccFroList;
```

Figur 41: Strukt som innehåller två pekare.

5.2 De nya funktionerna

Nu kommer vi att beskriva skillnaden mellan funktioner i den gamla och den nya implementationen. Först beskrivs insättning, sedan uppdatering, sökning, hämtning av alla objekt, och hämtning av ledigt froid, bortagning av ett objekt och sist bortagning av alla objekt i listan. Vi beskriver även sökningsfunktionen för ett roid.

5.2.1 Funktionen för insättning

Den nya insättningsfunktionen tar emot ett extra argument som är roid, se Figur 42. Detta kommer inte att medföra några förändringar för användaren eftersom roid alltid finns med. När objektet skapas i SCC-admin skapas det med roid 0 som sedan uppdateras på nytt av SCC-servern. I den gamla implementationen behövde man skicka storleken på data till funktionen för insättning. Detta gjorde man eftersom insättningsfunktionen utförde en minneskopiering, se Figur 10. Eftersom vi tidigare kunde konstatera att minneskopiering kostar många CPU-cykler konstruerade vi den nya insättningsfunktionen utan minneskopiering, se Figur 43. På grund av att vi inte längre behöver utföra minneskopiering i den nya insättningsfunktion skickar vi inte längre storleken på data som argument till funktionen för insättning.

```
Insättning i listan i gamla implementationen:

SccFroList_insertFro( SccFroList** list ,U32 froId, void* fro_p, int
frosize)


Insättning i listan i nya implementationen:

SccFroList_insertFro( SccFroList* list ,U32 froId, U32 roId, void* fro_p)
```

Figur 42: Insättning i listan

Insättningsfunktionen sätter in objekt i ett AVL-träd och en hjälplista. Insättningen i ett AVL-träd fungerar på så sätt att man sätter in dataobjektet via froid och sedan utförs rotationerna, se kapitel 4.1. Insättning i hjälplista via roid sker på två sätt. När värdet på roid är 0, görs ingen insättning och för alla andra värden på roid så sätts det nya objektet först i hjälplistan. Objektet som sätts i hjälplistan innehåller froid och roid. Den största skillnaden är att roid skickas som ett eget argument separat till insättningsfunktionen och inte i dataobjektet som man gjorde tidigare i den gamla implementationen.

Vi valde att sätta in objektet först i hjälplistan eftersom den inte behöver vara sorterad. Om listan skulle vara sorterad, skulle man då behöva flytta om objektet när ett objekt får ett nytt

roid dvs. vid uppdatering av ett objekt. Detta skulle medföra en massa extra arbete eftersom man skulle behöva flytta objekten i listan.

```
Insättning av objekt i listan utan minneskopiering:
newElement_p = ( SccFroListElement_r *) malloc( sizeof(
SccFroListElement_r ) );
if(currentElement_p == NULL )
{
    if( newElement_p == NULL )
    {
        result = SCCFROLIST_ERROR;
    }
    else
    {
        newElement_p->id = froId;
        newElement_p->data_p = fro_p;

        newElement_p->Height = 0;
        newElement_p->Left = newElement_p->Right = NULL;
        currentElement_p = newElement_p;

        result = SCCFROLIST_OK;
    }
}
```

Figur 43: Insättning i listan utan minneskopiering.

5.2.2 Funktionen för uppdatering

Skillnaden mellan den gamla och den nya funktionen för uppdatering vad det gäller hur funktionshuvudet ser ut, är densamma som för insättning. Vi skickar med ett extra argument som är roid. Sedan har vi tagit bort ett inargument som är storleken på dataobjektet då vi inte längre använder minneskopiering i funktionen uppdatera, se Figur 44.

Uppdateringsfunktion uppdaterar ett AVL-träd och en hjälplista. I ett AVL-träd uppdateras dataobjektet via froid. Uppdatering av hjälplistan via roid har flera olika fall.
Eftersom roid kan vara 0 eller byta värden under körning har vi valt att lösa det på följande sätt. Ett objekt till roid 0 som ska uppdateras tas bort från hjälplistan. I det andra fallet när vi ska uppdatera med ett roid skilt från 0 går vi igenom hjälplistan tills vi hittar objektet som ska uppdateras och uppdaterar det. Om objektet som ska uppdateras inte finns i hjälplistan är roid för det 0 för närvarande. Då skapas ett nytt objekt med det nya roid och sätts först i hjälplistan.

```
Uppdatera i listan i gamla implementationen:
SccFroList_updateFro(SccFroListElement_r** theList_p, U32 id, void*
data_p,int froSize)

Uppdatera i listan i nya implementationen:
SccFroList_updateFro( SccFroList* list ,U32 froId, U32 roId, void*
fro_p)

Uppdatera ett dataobjekt med minneskopiering i gamla implementationen:
memcpy(currentElement_p->data_p, data_p, froSize);

Uppdatera ett dataobjekt utan minneskopiering i nya implementationen:
currentElement_p->data_p = data_p;
```

Figur 44: Uppdatera i listan. Jämförelse mellan implementationena.

5.2.3 Funktionerna för sökning och hämtning

De nya funktionerna för att söka, hämta-första-objektet och hämta-nästa-objekt, vilka vi kallar för hämta alla objekt, använder inte längre minneskopiering och vi behöver inte längre skicka med storleken på dataobjektet. Dessa funktioner utförs endast på AVL-trädet. Sökningsfunktion söker efter ett objekt via froid och när den hittar det sökta objektet returnerar den en referens till objektet.

Hämta-första-objektet och hämta-nästa-objektet funktionerna fungerar ungefär på samma sätt. Dessa två funktioner kunde vi implementera som en funktion. Anledningen till att vi valde att ha två funktioner är att det inte ska bli för stora ändringar i användargränssnittet.

Den nya funktionen för att hämta ledigt froid utförs endast på AVL-trädet. Den går in i trädet och söker efter ett ledigt froid tills den hittar det. När den har hittat ett ledigt froid returneras det på samma sätt som i den gamla implementationen, se kapitel 3.2.

5.2.4 Funktionerna för borttagning

Funktionerna för att ta bort ett objekt och för att ta bort alla objekt utförs på AVL-trädet och hjälplistan. I borttagning av ett objekt i ett AVL-träd söker man efter ett objekt via froid. När

man har hittat det rätta objektet tar man bort det och utför rotationer på trädet. I borttagning av alla objekt utförs inga rotationer.

```
Allokering av minnet:
void * SccFroList_getItemPtr( int froSize)
{
   return malloc(sizeof(froSize));
}

Frigöring av minnet:
void SccFroList_freeItemPtr( void* ptr)
{
   free(ptr);
}
```

Figur 45: Funktionerna allokering och frigöring av minnet.

Vi har även implementerat två nya funktioner en för allokering av minnet och en för frigöring, se Figur 45. I den gamla implementationen gjorde man en kopia av dataobjektet vilket man jobbade med. Detta medförde att klienten var tvungen att frigöra minnet efter allokeringen. I den nya implementationen har vi en pekare som är kopplad till dataobjektet. Eftersom pekare och dataobjektet är kopplade på detta sätt innebär det att klienten inte kan ta hand om frigöring av minnet själv på samma sätt som i den gamla implementationen. Frigöring av minnet görs numera i borttagningsfunktioner. Om klienten av någon anledning skulle vilja ta bort dataobjektet själv finns det möjlighet till det med frigöringsfunktionen.

5.2.5 Funktion för sökning via roid

Implementationen av funktionen för sökning via ett roid gjorde vi till en början genom att vi använde redan befintliga funktioner i AVL-trädet dvs. hämta-första-objekt och hämta-nästa-objekt. Med hjälp av dessa funktioner kunde vi gå genom AVL-trädet och titta i varje dataobjekt efter det sökta roid. Detta var ineffektivt eftersom man är tvungen att hämta objektet först och sen titta på vilket roid objektet har. Om roid som man söker efter inte finns i den hämtade objektet är man tvungen att gå in i AVL-trädet på nytt och hämta nästa objekt. Eftersom det var ineffektivt bestämde vi oss för att implementera en hjälplista som innehåller roid och froid. På så sätt kan man söka dataobjektet via roid för att få froid till dataobjektet vilket man sedan använder för att söka efter dataobjektet i AVL-trädet. I denna lösning

behöver man inte hämta dataobjektet för att se vilket värde på roid objektet har, utan det räcker med en sökning i listan via roid för att hitta ett dataobjekt.

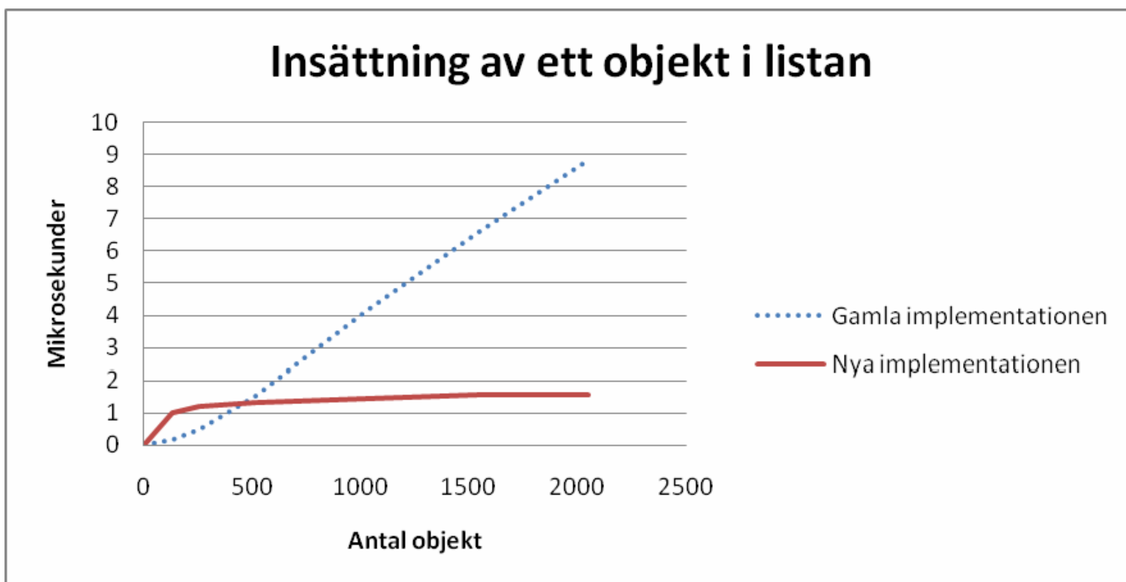
6 Resultat

I detta kapitel kommer vi att beskriva hur testerna är utförda för att jämföra exekveringstiderna för vår implementation jämfört med den gamla. Vi kommer även att ge kommentarer till resultaten samt alternativa lösningar, om sådana finns.

Vi skapade ett testprogram för att testa nya implementationen. I testprogrammet skapas först en array med fyra struktar som innehåller listpekare. Varje listpekare sätts att peka på NULL och storleken på listorna sätts till respektive 512, 1024, 1536 och 2048. Vi använder dessa listor för att se hur listoperationer uppträder tidsmässigt. Vi testade följande operationer på listan: insättning, uppdatering, söka ett objekt, ta bort ett objekt, hämta alla objekt i listan, ta bort alla objekt och hämta objekt på roid. I diagrammen som vi visar i detta kapitel har vi flera mätpunkter mellan 0 och 512 för att få en fullständig kurva från punkt 0 till 2048. Mätpunkterna mellan 0 och 512 finns inte med i tabellerna i bilagan A. Eftersom vi mätte tider i Linux-miljö varierar tiderna lite p.g.a. att Linux kör flera processer samtidigt. Testerna skulle från första början göras i CPP-miljö där endast en process körs i taget vilket skulle ge oss betydligt bättre kurvor. På grund av tidsbrist fick vi nöja oss med testerna från Linux-miljö.

6.1 Insättning

I insättningsfunktionen testar vi att fylla listor med objekt som är sorterade på froid. Normalstorleken enligt TietoEnator är ungefär 1 MB för ett objekt som används i listan. För att se hur funktioner som använder sig av minneskopiering beter sig fyller vi dataobjekt med information på 1 MB. Efter att vi har fyllt dataobjektet med information startar vi tidtagningen och börja sätta in objekt i listorna. Vi börjar med att sätta in från froid 1 till storleken på listan. När vi har fyllt listorna så stoppar vi tidtagningen. Tiden vi får delar vi med antalet objekt i listan för att få fram snittiden det tar att sätta in ett objekt i listan. Hur den gamla insättningen i listan skiljer sig åt tidmässigt från den nya finns i diagrammet, se Figur 46. Man kan se i diagrammet att den nya implementationen är bättre än den gamla när antalet objekt i listan är mer än 500 objekt. Eftersom tiderna skiljer enbart med mindre än en mikrosekund då listan har mindre än 500 objekt så har det ingen större betydelse.

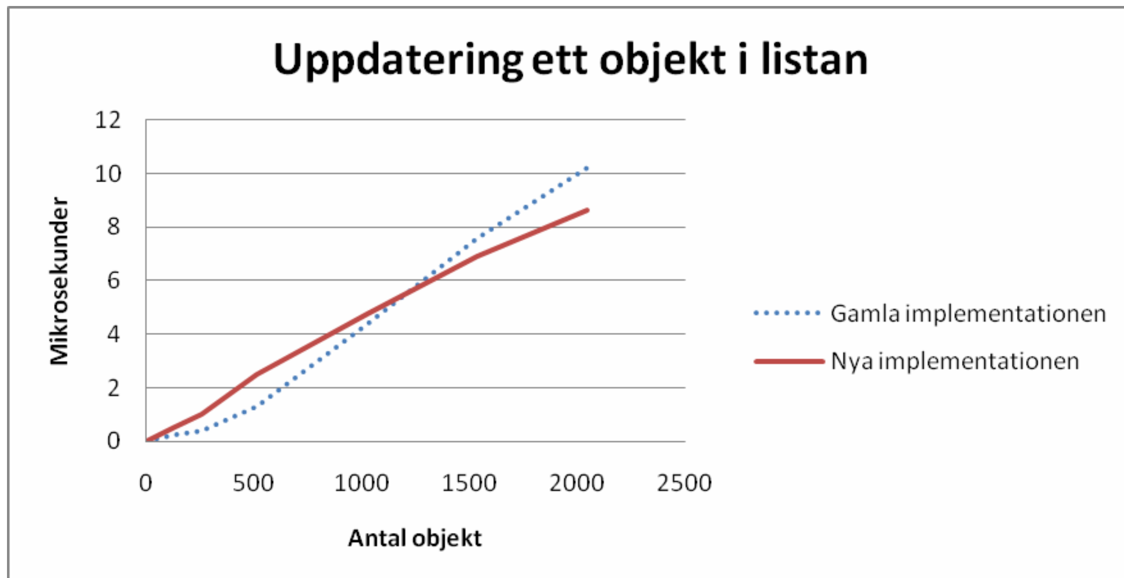


Figur 46: Test av insättning i listan.

Insättning kan variera i tid för den gamla implementationen beroende på vilken ordning objekten kommer i. För den nya implementationen har inte ordningen någon större betydelse eftersom insättningen av objekt kommer oavsett ordning ändå att ha en tidkomplexitet som är ungefär O(log n). Detta kan visas genom att man tar (tiden för insättning) / log(n) ≈ k, där n är antalet objekt i listan och k är en konstant.

6.2 Uppdatering

Testprogrammet för uppdatering av objekt i listan fungerar på ungefär samma sätt som för insättning. Man skapar ett nytt dataobjekt i testprogrammet som man sedan använder för att uppdatera ett redan befintlig dataobjekt i listan dvs. vi ersätter gamla objektet med ett nytt objekt. Tiden vi mätte för den gamla och den nya implementationen kan ses i diagrammet, se Figur 47.



Figur 47: Test av uppdatering av objekt i listan.

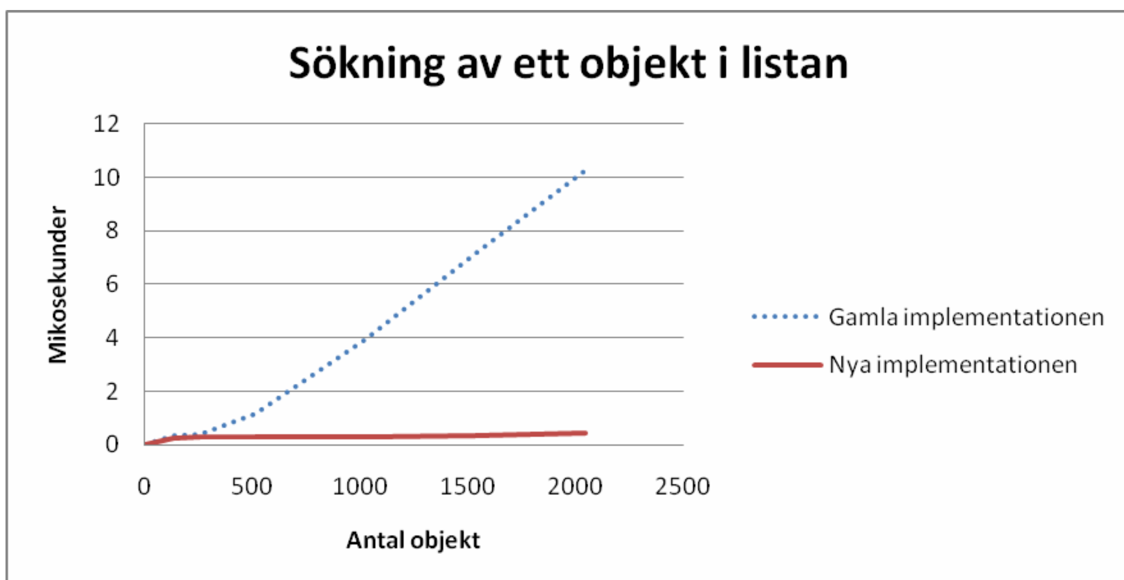
Diagrammet visar tydligt att det inte har blivit någon större förbättring med den nya implementationen. Detta beror på att vi både i den gamla och nya implementationen uppdaterar en länkad lista. I den gamla implementationen uppdaterar vi ett objekt i en länkad lista, medan vi i den nya uppdaterar ett objekt i ett AVL-träd och en hjälplista, som är en länkad lista. Vi testade även att mäta tiden för uppdatering av ett objekt i endast ett AVL-träd för att se hur lång tid det tar. Tiden för funktionen som uppdaterar endast ett AVL-träd bli ungefär samma som tiden för sökning på froid eftersom uppdatera funktionen söker efter ett objekt via froid och flyttar om pekare. Vi har ändå valt att ha kvar implementationen som uppdaterar i ett AVL-träd och en hjälplista, eftersom de tider vi fick var ungefär lika med dem som den gamla implementationen gav och att hjälplistan ger en förbättring vid sökning på roid. Sökning på roid är en viktig funktion att förbättra som vi nämnde tidigare i kapitel 3.

För uppdatering av endast AVL-träd har vi en tidkomplexitet som är O(log n). Eftersom vi har valt att uppdatera både hjälplista och ett AVL-träd i den nya implementationen får vi en tidkomplexitet som är O(n).

Vi kan konstatera att det inte har blivit någon större förbättring på nya funktionen uppdatera ett objekt i listan. Men vi har i alla fall blivit av med minneskopiering som är ineffektivt och kostsamt i CPP miljö som vi nämnde tidigare i kapitel 3.5.

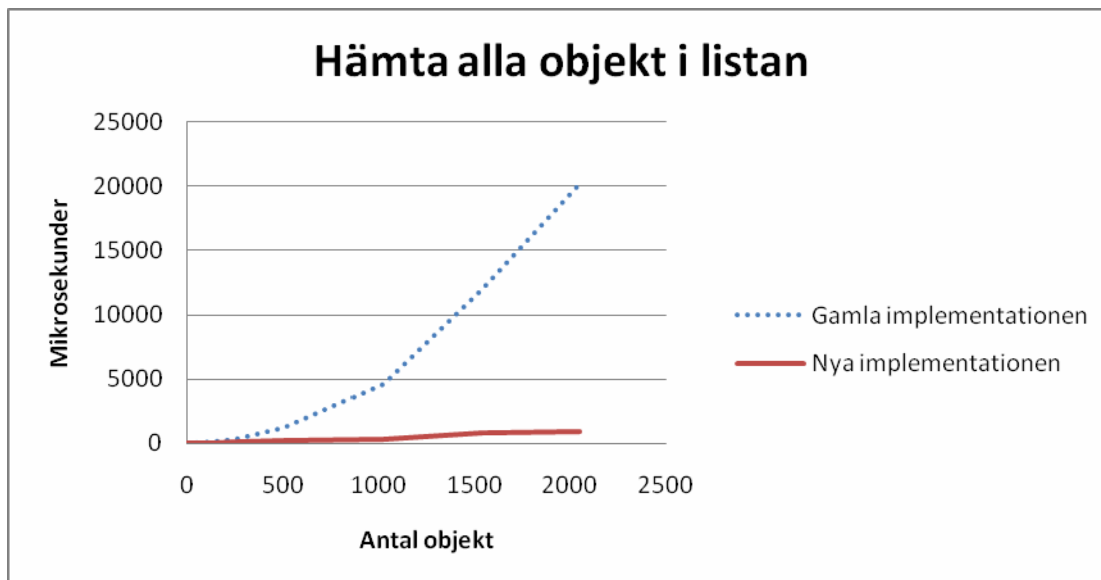
6.3 Sökning och hämtning

Testet för sökning efter ett objekt via froid utförde vi på så sätt att vi sökte efter samtliga objekt i listan och räknade ut ett snittvärde för hur lång tid det tar att hitta ett objekt i listan. Vi fick en tydlig förbättring, som kan ses i diagrammet, Figur 48. Detta var förväntat eftersom vi redan visste att sökning i en länkad lista har en tidkomplexitet O(n) och sökning i ett AVL-träd har tidkomplexitet O(log n), se Tabell 2. För detta test finns det inga vanliga testfall som kan vissa att den nya implementationen är sämre än den gamla av listan.



Figur 48: Sök ett objekt i listan på froid.

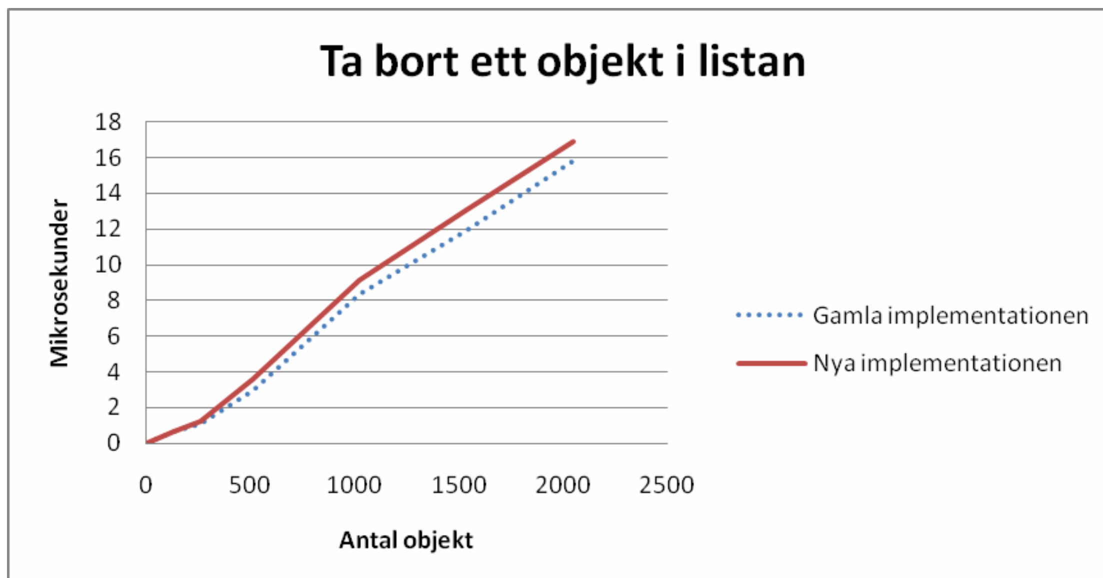
Testet att hämta alla objekt i listan fungerar så att vi mäter tiden för att se hur lång tid det tar att hämta alla objekt i listan. Diagrammet med tiderna för den nya och gamla implementationen kan ses i Figur 49. Man kan se att den nya implementationen är bättre än den gamla. Tidskomplexitet för den gamla implementationen är O(n^2) som vi kom fram till i kapitel 3.4. I den nya implementationen måste vi för varje objekt gå in i ett AVL-träd som har för hämtning av ett objekt en tidkomplexitet O(log n). Detta medför att vi får en slutlig tidkomplexitet O(n log n). Operationen för att hämta alla objektet i listan har blivit bättre i den nya implementationen jämfört mot den gamla.



Figur 49: Test att hämta alla objekt i listan.

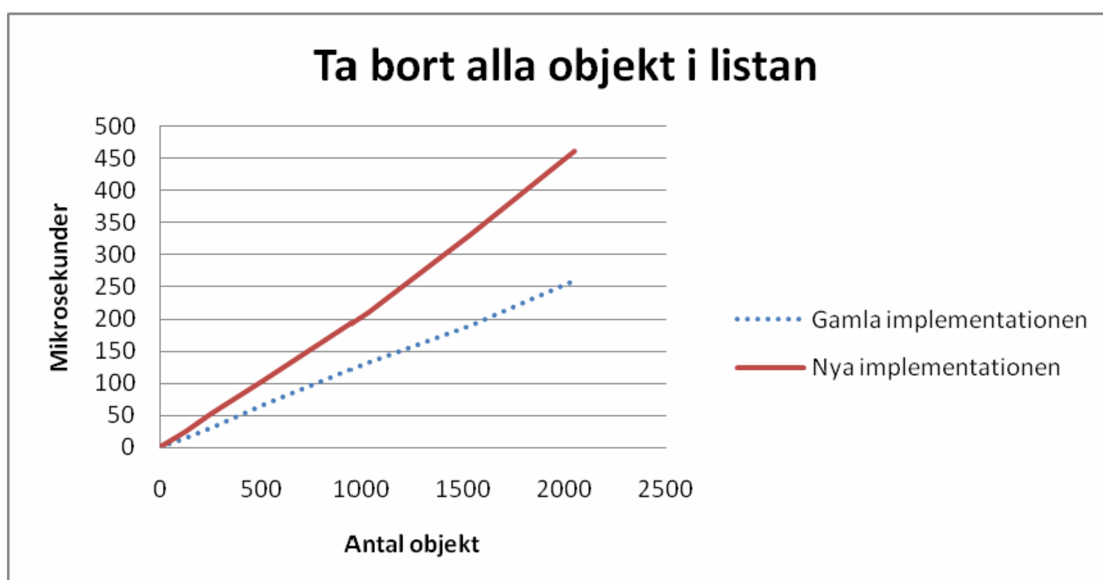
6.4 Borttagning

I funktionen för att ta bort ett objekt i listan valde vi att ta bort i slutet av listan. Vi mätte tiden för att ta bort samtliga objekt i listan. Tiden som vi fick delade vi med antalet objekt för att få snittiden för borttagning av ett objekt. Diagrammet med tiderna för den nya och gamla implementationen finns i Figur 50. I diagrammet kan man se att testet ger oss inga större skillnader tidsmässigt mellan den gamla och nya implementationen. Det beror på att båda implementationerna använder sig av en länkad lista. Tidkomplexitet för den gamla implementationen är O(n) som vi kom fram i kapitel 3.4. För den nya implementationen måste man ta bort ett objekt i hjälplistan och ett AVL-träd vilket ger oss en tidkomplexitet O(n). För funktionen borttagning av ett objekt har vi inte fått någon förbättring men heller ingen försämring. Vi har ändå valt att ha kvar implementationen som tar bort i ett AVL-träd och en hjälplista, eftersom de tider vi fick var ungefär lika med dem som den gamla implementationen gav och att hjälplistan ger en förbättring vid sökning på roid. Sökning på roid är en viktig funktion att förbättra som vi nämnde tidigare i kapitel 3.



Figur 50: Ta bort ett objekt i listan.

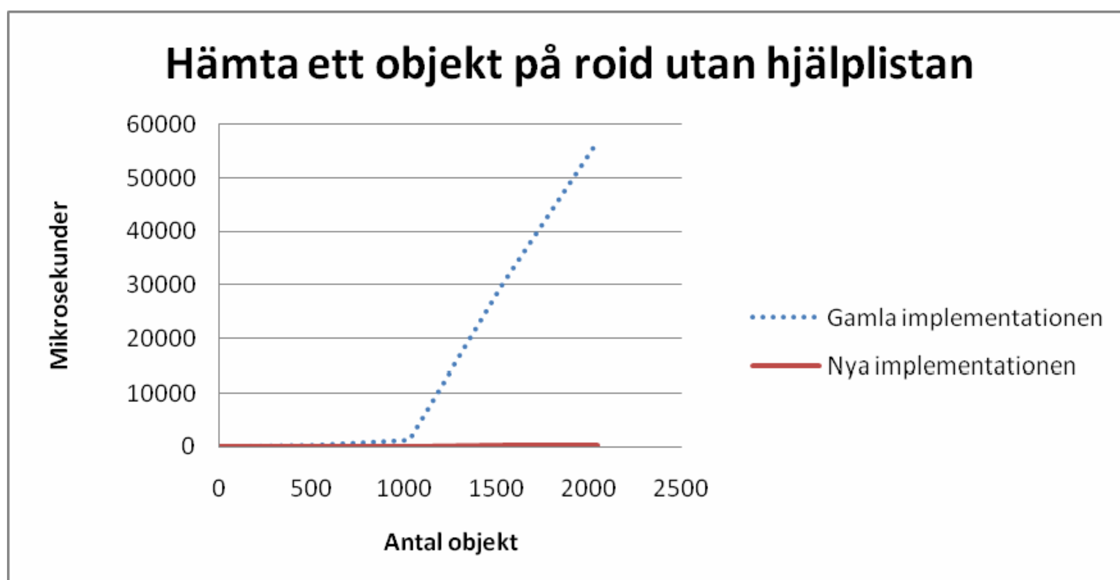
Test av ta bort alla objekt i listan går bara att göra på ett sätt. Detta är att anropa funktionen och den tar bort objekten ett efter ett tills listan är tom. Vi startar tidtagningen när vi anropar funktionen och stoppar tidtagningen när funktionen har tagit bort samtliga objekt. Tiderna som vi fått kan ses i diagrammet, se Figur 51.Tidkomplexitet för den gamla implementationen är O(n) som vi kom fram till i kapitel 3.4. Den nya implementationen har en tidkomplexitet som också är O(n) men tiderna dubbleras eftersom den tar bort alla objekt både i ett AVL-träd och en hjälplista. För funktionen ta bort alla objekt i listan är den gamla implementation en bättre än den nya.



Figur 51: Ta bort alla objekt i listan.

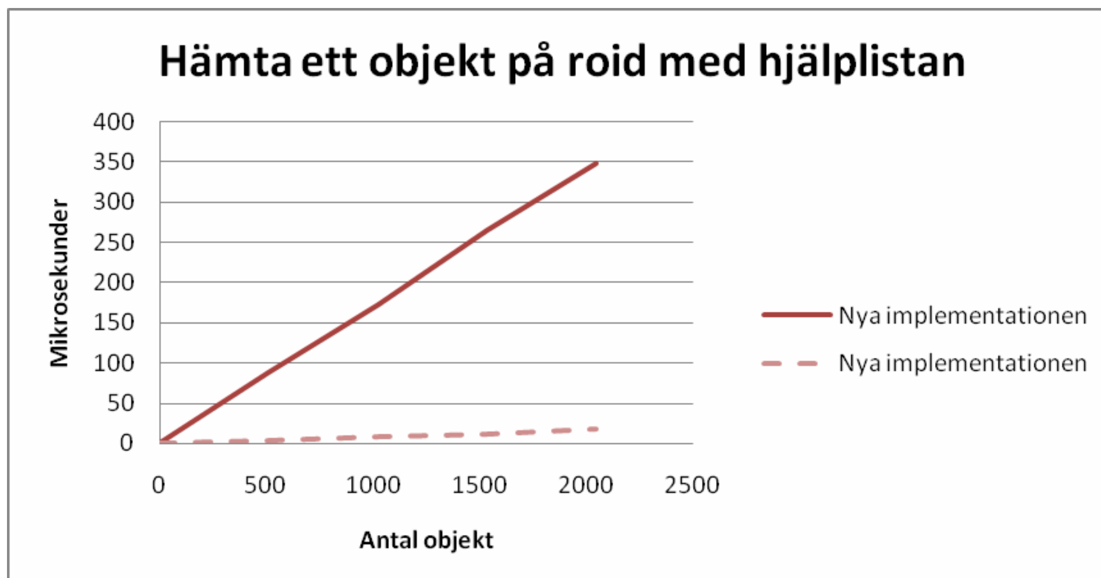
6.5 Sökning via roid

För att utföra testet för funktionen hämta ett objekt på roid mätte vi tiderna för den gamla och den nya implementationen utan hjälplistan. Testet för att hämta ett objekt via roid utfördes på samma sätt som testet för att söka efter ett objekt på froid. Skillnaden är att vi nu söker via roid istället för froid. En annan skillnad är att roid ligger i dataobjektet vilket gör att om vi vill jämföra roid med den sökta roid måste vi hämta dataobjektet först och sedan jämföra. Detta medför att det tar lite längre tid att gå genom listan eftersom man behöver använda sig av operationen som vi kallar hämta alla objekt i listan. För att se tiden se Figur 52. Man kan se att vi har fått en förbättring med den nya implementationen. Hämtning av ett objekt på roid utförs med en tidkomplexitet som är $O(n^2)$ i den gamla implementationen som vi kom fram i kapitel 3.4. För den nya implementationen är tidkomplexitet O(n log n), eftersom man måste gå genom AVL-trädet för varje objekt som ska hämtas.



Figur 52: Hämta ett objekt på roid utan hjälplistan.

Vi mätte också tiden på den nya implementationen där vi använder en hjälplista för att utföra sökningen på roid. Med de tiderna vi har fått kunde vi konstatera att sökning via roid i en länkad lista istället för ett AVL-träd ger oss ytterligare förbättringar, se Figur 53. Tidkomplexitet för den nya implementationen med hjälplista blir O(n), eftersom vi söker i hjälplistan och AVL-träd.



Figur 53: Hämta ett objekt på roid med hjälplistan.

Efter våra tester och resultat kan vi dra en slutsats att vi har uppnått vårt mål dvs. vi har förbättrat insättnings- och sökningsfunktionerna som var viktiga att förbättra enligt TietoEnator. På funktionerna uppdatera, tabort ett objekt och tabort alla objekt har vi inte fått någon förbättring. Dessa funktioner har inte heller blivit sämre än de gamla funktionerna. Uppdatera, tabort ett objekt och tabort alla objekt är funktioner som utförs enligt TietoEnator oftast när systemet inte är belastat vilket inte kräver någon förbättring.

7 Slutsats

Syftet med projektet var att förbättra sökningen som sker linjärt, efter ett specifikt listobjekt i en lista som används i SCC-admin. Vi undersökte olika datastrukturer för att kunna se vilken datastruktur som eventuellt skulle kunna ge oss en förbättring. Efter undersökningen kom vi fram till att balanserat sökträd var det bästa alternativet. De flesta operationerna på balanserade sökträd utförs med en tidkomplexitet som är O(log n). Vi bestämde oss att implementera ett AVL-träd, eftersom det har bättre söktider än de andra balanserade sökträden vi undersökte. Implementationen av ett AVL-träd gav oss resultat som kan ses i Tabell 3. Resultaten i tabellen representerar medelfall. Vi har valt att ha med i tabellen endast de fallen när listan innehåller 2048 objekt. För att se hur listan beter sig i de andra fallen dvs. vid andra storlekar på listan se bilagan A.

Funktioner	Gamla implementationen (mikrosekunder)	Nya implementationen (mikrosekunder)	Förbättringsfaktor
Insättning av ett objekt i lista som består av 2048 objekt	8.84	1.56	6 ggr
Sök ett objekt i lista som består av 2048 objekt	10.26	0.46	20ggr
Hämta ett objekt på roid i lista som består av 2048 objekt	56912.61	18.49	3000ggr
Hämta alla objekt i listan med 2048 objekt	20172.00	892.00	20ggr

Tabell 3: Resultat på förbättringar.

Man räknar ut förbättringsfaktorn genom att dela den gamla tiden med nya tiden. Man kan se i Tabell 3 att vi har fått förbättringar på funktioner insättning, sökning, hämtning av objekt i listan och hämtning av objekt på roid. För att uppnå en förbättring för att hämta objekt på roid har vi implementerat ytterligare en hjälplista som är en länkad lista.
Funktioner som vi inte har fått en förbättring på är ta bort alla objekt, uppdatera och ta bort ett objekt. Eftersom dessa funktioner utförs oftast när systemet inte är belastat t.ex. nattetid gör det inte så mycket att de är lite långsammare.

En annan stor förbättring vi har gjort är att vi inte längre använder minneskopiering vilket kostade många CPU cykler enligt TietoEnators tidigare mätningar.

Referenser

[1] *http://www.tietoenator.se/,* 2007-04-18

[2] Ericsson Telecom AB och Telia AB, Att förstå telekommunikation, Ericsson Telecom AB, Telia AB och Studentlitteratur, 1998

[3] http://www.ss7.net/ss7-tutorial/ss7-overview-by-nettest.pdf

[4] http://www.ss7.net/ss7-tutorial/sunrise-tutorialss7.pdf

[5] http://www.ss7.net/ss7-tutorial/ss7-intro-by-brooktrout.pdf

[6] http://www.ss7.net/ss7-tutorial/iec-ss7.pdf

[7] http://www.ss7.net/ss7-tutorial/ss7-protocol-stack.pdf

[8] http://www.ss7.net/ss7-tutorial/pt-ss7-tutorial.pdf

[9] Mark Allen Weiss. Data structures and problem solving using C++, Addison Wesley 2nd edition, 2000

[10] http://www.cs.chalmers.se/ComputingScience/Education/Courses/d2dat/lecture11-06.pdf

[11] William Pugh, Skip List: A Probabilistic Alternative to Balanced Trees, Communications of the ACM, Juni 1990

[12] http://en.wikipedia.org/wiki/SS5

[13] http://en.wikipedia.org/wiki/AT%26T

[14] http://en.wikipedia.org/wiki/ITU

[15] www.artes.uu.se/industry/031111/5CPP-Artes.ppt

[16] http://www.pt.com/tutorials/ss7/sccp.html

A Tabeller

Test av insättning i listan (microsekunder)		
Antal objekt	Gamla implementationen	Nya implementationen
0	0	0
128	0,1796875	1,015625
256	0,46875	1,1953125
512	1,5078125	1,3359375
1024	4,140625	1,448242188
1536	6,544921875	1,546223958
2048	8,844726563	1,563476563

	Test av uppdatering i listan (mikrosekunder)	
Antal objekt	Gamla implementationen	Nya implementationen
0	0	0
128	0,234375	0,515625
256	0,390625	1
512	1,306640625	2,5
1024	4,368164063	4,752929688
1536	7,570963542	6,885416667
2048	10,22802734	8,616699219

	Test av sökning i listan (mikrosekunder)	
Antal objekt	Gamla implementationen	Nya implementationen
0	0	0
128	0,3515625	0,25
256	0,390625	0,3046875
512	1,15625	0,302734375
1024	3,94921875	0,330078125
1536	7,162109375	0,377604167
2048	10,26611328	0,461425781

	Test av ta bort i listan (mikrosekunder)	
Antal objekt	Gamla implementationen	Nya implementationen
0	0	0
128	0,6171875	0,6484375
256	1,1015625	1,25
512	2,91796875	3,560546875
1024	8,3671875	9,143554688
1536	11,95638021	13,06315104
2048	15,83837891	16,86474609

	Test av ta bort hela listan (mikrosekunder)	
Antal objekt	Gamla implementationen	Nya implementationen
0	0	0
128	0,1171875	0,1796875
256	0,1171875	0,19921875
512	0,126953125	0,203125
1024	0,12890625	0,204101563
1536	0,123697917	0,216145833
2048	0,126953125	0,225097656

	Test av hämta alla objekt i listan (mikrosekunder)	
Antal objekt	Gamla implementationen	Nya implementationen
0	0	0
128	0,6640625	0,265625
256	1,0546875	0,28515625
512	2,248046875	0,298828125
1024	4,522460938	0,315429688
1536	7,806640625	0,491536458
2048	9,849609375	0,435546875

	Test hämta roid (mikrosekunder)		
Antal objekt	Gamla implementationen	Nya implementationen utan hjälplista	Nya implementationen med hjälplista
0	0	0	0
512	203,5820313	88,07421875	2,90625
1024	1140,702148	173,6162109	7,9765625
1536	30357,26888	266,3639323	11,21940104
2048	56912,61816	348,9765625	18,49365234