Datavetenskap

Robert Axelsson & Jimmy Holmén

Generiska datastrukturer för SS7

Examensarbete, C-nivå

2005:4

Generiska datastrukturer för SS7

Robert Axelsson & Jimmy Holmén

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.

Robert Axelsson

Jimmy Holmén

Godkänd, 14 januari 2005

Handledare: Robin Staxhammar

Examinator: Martin Blom

Sammanfattning

Denna rapport beskriver ett examensarbete som gjordes åt TietoEnator i Karlstad hösten 2003. Målet med examensarbetet var att skapa ett bibliotek med generiska datastrukturer i programspråket C. Arbetet bestod av kravinsamling, kravsammanställning, kravanalys, design samt implementation och testning av datastrukturerna i den miljö där biblioteket skall användas. Kravsammanställningen resulterade i sex krav som bibliotekets datastrukturer måste uppfylla. När kravanalysen var avslutad hade tre datastrukturer, som tillsammans uppfyllde de sex ställda kraven, hittats. De tre datastrukturerna var AVL-trädet, den dynamiska arrayen och den länkade listan. Designen samt implementationen och testningen av datastrukturerna genomfördes med framgång.

Generic data structures for SS7

Abstract

This report describes a bachelor's project that was performed at, and on behalf of, TietoEnator in Karlstad, during the autumn 2003. The project goal was to create a library of generic data structures written in the programming language C. The job consisted of requirements collection, compilation of requirements, requirements analysis, and also design, implementation and testing of the data structures in the environment where the library will be used. The requirements compilation resulted in six requirements that had to be met by the data structures of the library. When the requirements analysis was finished, three data structures that together complied with the six requirements had been found. The three data structures were the AVL-tree, the dynamic array and the linked list. The design, implementation and testing of the data structures was successfully pursued.

Innehållsförteckning

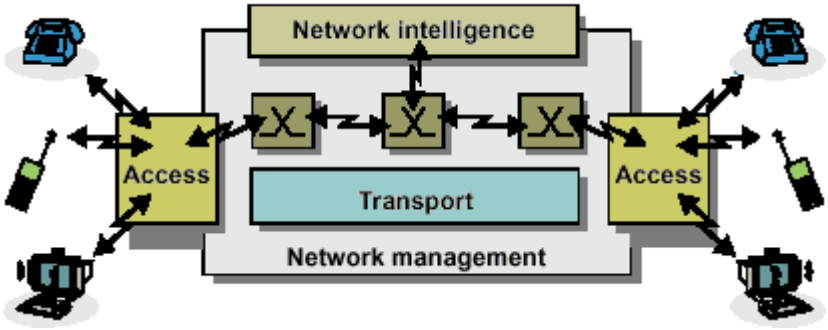
Figurförteckning

Tabellförteckning

1 Inledning

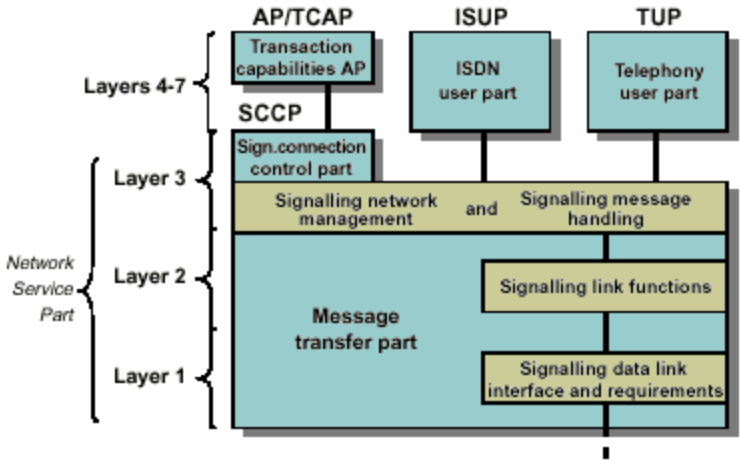
1.1 Bakgrund

I Karlstad har Tieto Enator sedan slutet av 80-talet varit verksamma inom utvecklingen av utrustning för Ericssons signaleringssystem.[3]



Figur 1.1 Signaleringssystemet binder ihop de olika delarna i ett telekommunikationsnätverk

Tieto Enator var först verksamma som konsulter i dåvarande bolaget Ericsson Programatic, sedan som dotterbolag i Ericsson-koncernen, och nu som underleverantörer/samarbetspartners till Ericsson i form av företaget Tieto Enator Telecom Partner AB. Bland annat har man tagit fram en plattformsoberoende stack för Signaleringssystem Nummer 7; Portable SS7. SS7 är en standard för signalering vid telekommunikation.



Figur 1.2 De olika lagren i SS7-stacken.

SS7-stacken [6] är en nätverksstack liknande TCP/IP och den används bl. a. för att sköta kommunikationen i Ericssons telefonväxlar. Tieto Enators implementation av SS7, Portable SS7, är huvudsakligen implementerad i programspråket C, ett programspråk som definitivt inte brukar associeras med begreppet ”plattformsoberoende”. Det som ändå gör Portable SS7 plattformsoberoende är att den är konstruerad med hjälp Common Parts. Common Parts är ett C-bibliotek som är en del av SS7-stacken. SS7-stacken körs på följande plattformar: operativsystemen Solaris 8, AIX 5L, HP Unix (version 11.11), Linux RedHat 9, kernel 2.4.x, TSP 5, CPP4 (OSE 5-x, med heapmanager), GARP (OSE 5-x, utan heapmanager) samt VxWorks (version 5.4).

Kärnan i Common Parts tillhandahåller rutiner för nätverkskommunikation och parallellprogrammering för realtidssystem (t ex timers). Trots att Common Parts under senare tid har växt, ligger fortfarande tyngdpunkten vid rutiner för nätverks- och processprogrammering.

Portable SS7 består av många olika moduler. Varje modul är en implementation av ett lager i stacken. På grund av brister i samordningen mellan de olika delarna i stacken programmeras i dagsläget datastrukturer separat för varje modul. Det blir alltså mycket redundant programmering, dvs. man utvecklar samma sak på flera olika ställen. Utvecklarna av de olika protokollmodulerna har därför ett behov av att ha tillgång till ett gemensamt bibliotek med olika datastrukturer som ska kunna lagra vilken datatyp som helst.

1.2 Mål

Målet är att skapa ett C-bibliotek med generiska datastrukturer. Begreppet generiska datastrukturer används i denna uppsats som benämning på datastrukturer som kan lagra en

godtycklig datatyp. Biblioteket ska vara plattformsoberoende efter kompilering. Följande delmål ingår:

1. Kravinsamling, Kravsammanställning → Kravspecifikation. Vi ska samla in information genom att intervjua de ansvariga för de olika modulerna, utvärdera informationen och skriva en kravspecifikation med verifierbara krav, det vill säga krav formulerade på ett enkelt och tydligt sätt så att det med lätthet kan visas att de går att uppfylla.
2. Design → designspecifikation (implementation proposal). Vi ska definiera hur biblioteket ska vara uppbyggt.
3. Implementation av c-biblioteket ska genomföras med avseende på designspecifikationen.
4. Vi ska dokumentera ett API för klientprogrammeraren där det tydligt framgår hur man använder c-biblioteket.

1.3 Uppsatsens upplägg

Vi börjar med att beskriva sammanställningen av de krav som samlats in av de olika modulansvariga. Efter det kommer kraven att analyseras närmare och förslag på datastrukturer ges, dessa datastrukturer kommer att analyseras för att vi ska kunna välja ut de som passar bäst för biblioteket. Detta följs av att vi beskriver de utvalda datastrukturernas design. Sedan ger vi en mer detaljerad beskrivning av hur datastrukturerna kommer att implementeras för att klara just de specifika kraven som ställs, samt hur dokumentationen av koden kommer att gå till. Vi tar upp de resultat och de rekommendationer som vi efter arbetets slut kommit fram till och avslutningsvis summerar vi exjobbet.

1.4 Definitioner av begrepp

1.4.1 Generisk datastruktur

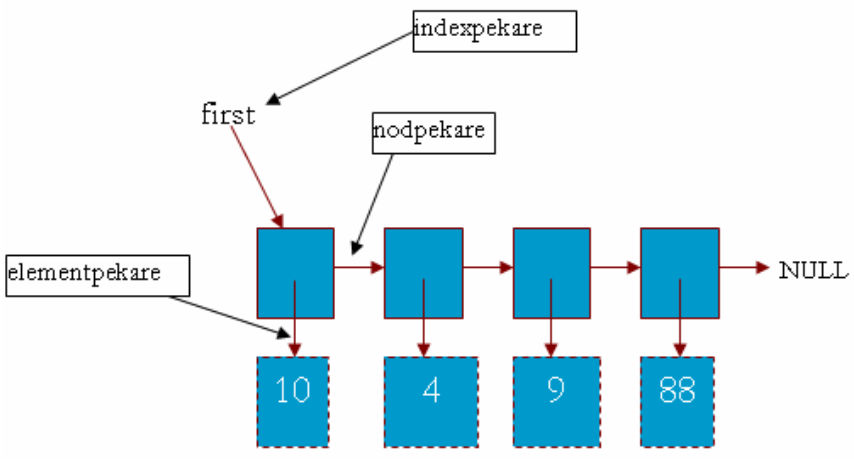
Generiska datastrukturer används i denna uppsats som benämning på datastrukturer som kan lagra en godtycklig datatyp.

1.4.2 Pekare

I projektet används pekare för olika syften och vi har därför valt att i rapporten dela in dem i tre olika kategorier.

- Indexpekare – extern referens till en eller flera noder i datastrukturen.

- Nodpekare – pekare som kopplar samman noderna i en datastruktur.
- Elementpekare – pekare som används för att koppla ett element till en nod. Eftersom datastrukturerna ska kunna lagra en godtycklig datatyp är elementpekaren definierad som en voidpekare.



Figur 1.3 Olika pekare

1.4.3 Tidskomplexitet

När vi anger tidskomplexiteten för operationer är det, om inget annat anges, alltid värstafallet som avses

1.4.4 Parameter

Med begreppet parameter menar vi de fält som klientprogrammeraren själv kan sätta för att beskriva datastrukturernas egenskaper. Ett exempel på detta är vid skapandet av en array, då man skickar med en parameter som beskriver hur stor man vill att arrayen ska vara.

2 Sammanställning av kravspecifikationen

Detta kapitel ger en beskrivning över hur vi samlade in information till kravspecifikationen. Vidare ges en sammanfattning av den kravspecifikation vi utgick ifrån när vi designade produkten.

2.1 Kravinsamling

Det första steget i kravinsamlingen var att ta reda på vilka moduler som skulle kunna ha nytta av ett c-bibliotek med generiska datastrukturer. Vår handledare på företaget gav oss en lista över de viktigaste modulerna. Vi kontaktade personerna som var ansvariga för de olika modulerna och bad dem ta reda på vilka operationer de utförde på sina datastrukturer. Sedan tog vi reda på vilka tidskrav som fanns på de olika operationerna. Informationen vi fått fram sammanställdes och analyserades. Några fler möten följde där vi tog fram mer detaljerad information, bland annat krav på lagringen av element i datastrukturen. I de fall vi inte fick något tidskrav på en operation valde vi att sätta tidskomplexitetskravet O(n), detta för att ha ett rimligt krav att jobba emot. O(n) är ett rimligt tidskrav som är satt så att det inte stör ut kraven satta av de modulansvariga, detta för att undvika bortfall av annars potentiella datastrukturer.

Modul 1 (OAM [5]**)**

a)

Det behövs en sekvens där upp till 256 element ska kunna lagras. Operationer som ska kunna utföras på sekvensen är sökning, gå till nästa element, insättning samt borttagning. Viktigt är att sökningen ska gå snabbt. Tidskomplexitetskravet för sökning är O(log(n)). Man ska kunna nå nästföljande element i sekvensen på konstant tid, dvs. O(1). Vi fick inget speciellt tidskrav vad det gällde insättningen och borttagningen, vi satte därför ett tidskomplexitetskrav för dessa operationer till O(n).

Antalet element som lagras varierar med hög frekvens.

b)

Det behövs en sekvens där man ska kunna lagra ett godtyckligt antal element. Operationer som ska kunna utföras på sekvensen är sökning, insättning samt borttagning.

Tidskomplexitetskravet är O(log(n)) på samtliga operationer. Antalet element som lagras kommer att variera ofta och variationerna mellan antalet lagrade element är stora.

Antalet element som lagras varierar med hög frekvens.

Modul 2 (M3 [6]**)**

Det behövs en sekvens där upp till 16384 element ska kunna lagras. Operationer som ska kunna utföras på sekvensen är sökning, insättning samt borttagning. Viktigt är att sökningen ska gå snabbt. Tidskomplexitetskravet för sökning är O(log(n)). Insättning och borttagning sker väldigt sällan och behöver inte gå särskilt snabbt. Vi fick inget speciellt tidskrav vad det gällde insättningen och borttagningen, vi satte därför ett tidskomplexitetskrav för dessa operationer till O(n).

Oftast är antalet element som lagras litet, relativt den allokerade storleken. Det finns dock fall när antalet element som behöver kunna lagras är mycket stort.

Modul 3 (MTPL3 [6]**)**

Det behövs en sekvens där upp till 16384 element ska kunna lagras. Operationer som ska kunna utföras på sekvensen är sökning, insättning samt borttagning. Viktigt är att sökningen ska gå snabbt. Tidskomplexitetskravet för sökning är O(log(n)). Insättning och borttagning sker väldigt sällan och behöver därför inte gå särskilt snabbt. Vi fick inget speciellt tidskrav vad det gällde insättningen och borttagningen, vi satte därför ett tidskomplexitetskrav för dessa operationer till O(n).

Oftast är antalet element som lagras litet, relativt den allokerade storleken. Det finns dock fall när antalet element som behöver kunna lagras är mycket stort.

Modul 4 (FEBF/BEIF [6]**)**

a)

Det behövs en sekvens där upp till 512 element ska kunna lagras. Operationer som ska kunna utföras på sekvensen är sökning, insättning samt borttagning. Viktigt är att sökningen ska gå snabbt. Tidskomplexitetskravet för sökning är O(log(n)). Insättning och borttagning sker väldigt sällan och behöver inte gå särskilt snabbt. Vi fick inget speciellt tidskrav gällande dessa operationer, vi satte därför ett tidskomplexitetskrav för dessa till O(n).

Antalet element som lagras varierar med hög frekvens.

b)

Det behövs en sekvens där man ska kunna lagra ett bestämt antal element. Sekvensen ska användas som en buffert för meddelanden. När bufferten är full och ett nytt meddelande ankommer tas det äldsta meddelandet bort. Operationer som ska kunna utföras på sekvensen är i första hand insättning och borttagning. Vi fick vid intervjutillfället inget specifikt tidskrav angivet för operationerna. Eftersom vi vet att insättning och borttagning bara kommer att ske i början och slutet på sekvensen valde vi att sätta tidskomplexitetskravet O(1) för dessa operationer. Det ska även vara möjligt att söka i sekvensen. Vi fick inget speciellt tidskrav på denna operation och valde därför att sätta ett tidskomplexitetskrav på O(n).

Antalet element som lagras varierar med hög frekvens.

Modul 5 (SCTP [6]**)**

Det behövs en sekvens där upp till 512 element ska kunna lagras. Operationer som ska kunna utföras på sekvensen är sökning, insättning samt borttagning. Viktigt är att sökningen ska gå snabbt. Tidskomplexitetskravet för sökning är O(log(n)). Insättning och borttagning sker väldigt sällan och behöver inte gå särskilt snabbt. Vi fick inget speciellt tidskrav gällande dessa operationer, vi satte därför ett tidskomplexitetskrav för dessa till O(n).

Oftast är antalet element som lagras litet, relativt den allokerade storleken. Det finns dock fall när antalet element som behöver kunna lagras är mycket stort.

Modul 6 (ISUP [6]**)**

Det behövs en sekvens där upp till 32640 element ska kunna lagras. Sekvensen ska användas som en buffert för meddelanden. Operationer som ska kunna utföras på sekvensen är insättning samt borttagning. Vi fick vid intervjutillfället inget specifikt tidskrav angivet för operationerna. Eftersom vi vet att insättning och borttagning bara kommer att ske i början och slutet på sekvensen valde vi att sätta tidskomplexitetskravet O(1) för dessa operationer.

Antalet element som lagras varierar med hög frekvens.

Modul 7 (TCAP [6]**)**

a)

Det behövs en sekvens där upp till 65535 element ska kunna lagras. Operationer som ska kunna utföras på sekvensen är sökning, insättning samt borttagning. Viktigt är att sökningen ska gå snabbt. Tidskomplexitetskravet för sökning är O(log(n)). Insättning och borttagning

sker väldigt sällan och behöver inte gå särskilt snabbt Vi fick inget speciellt tidskrav gällande dessa operationer, vi satte därför ett tidskomplexitetskrav för dessa till O(n).

Antalet element som lagras varierar med hög frekvens.

b)

Det behövs en sekvens där man ska kunna lagra ett bestämt antal element. Sekvensen ska användas som en buffert för meddelanden. När bufferten är full och ett nytt meddelande ankommer tas det äldsta meddelandet bort. Operationer som ska kunna utföras på sekvensen är i första hand insättning och borttagning. Vi fick vid intervjutillfället inget specifikt tidskrav angivet för operationerna. Eftersom vi vet att insättning och borttagning bara kommer att ske i början och slutet på sekvensen valde vi att sätta tidskomplexitetskravet O(1) för dessa operationer. Det ska även vara möjligt att söka i sekvensen. Vi fick inget speciellt tidskrav på denna operation och valde därför att sätta ett tidskomplexitetskrav på O(n).

Antalet element som lagras varierar med hög frekvens.

2.2 Kravsammanfattning

Här ges en ingående beskrivning av den kravspecifikation vi konstruerade för projektet.

2.2.1 Krav på minnet

På grund av att SS7-stacken ska kunna köras på många olika plattformar (se avsnitt 1.1) måste minneskraven sättas utifrån de plattformar som har de hårdaste kraven.

Allokeringen av det statiska minnet, d.v.s. det minne som allokeras innan programexekvering, ska hållas på så låg nivå som möjligt. På plattformen TSP (Ericsson Telecom Server Platform) finns bara en begränsad mängd statiskt minne som delas av alla processer och alltså måste utnyttjandet av denna minnesarea hållas nere.

Även den dynamiska minnesallokeringen måste hållas nere, det vill säga heapen. Maximala blockstorleken, alltså det största sammanhängande minnesområdet som får allokeras på heapen, är 64 KB. Begränsningen för heapminnesallokeringen beror på plattformen GARP (Generic Attribute Registration Protocol). Där finns egentligen ingen heap, utan istället finns det något som kallas för OSE-signaler [4]. OSE-signaler är block av förutbestämd storlek som delar minnesutrymme med stacken. Största storlek på en OSE-signal är 64KB.

Plattformarna som c-biblioteket kommer att köras på är 32-bitars-system. Detta ger att storleken på varje nod är minst 4 B eftersom noden innehåller en elementpekare. I en indexerad datastruktur behövs inte några nodpekare men istället måste elementen lagras i en

följd i minnet. Detta innebär att maximalt antal element i en indexerad datastruktur utan nodpekare blir blockstorleken dividerat med elementpekarstorleken (64KB/4B) = 16384.
På stacken, vars minnesallokering begränsas av plattformarna GARP och CPP, får maximalt 16 KB allokeras.

2.2.2 Krav på lagring av element i ADT:erna

När det gäller lagring av element kan man göra en uppdelning i två olika kategorier.

Modul	Krav på lagring
2, 3, 5	En datastruktur som är minneseffektiv vid ett litet antal element men som också ska kunna växa för att ibland kunna lagra ett stort antal element.
1 a,b, 4 a,b, 5, 7a,b	En datastruktur som har som med låg tidskomplexitet kan ändras i storlek då insättning eller borttagning av element sker.

Tabell 2.1 Olika kategorier för lagring

2.2.3 Tidskrav på operationerna som ska kunna utföras på datastrukturerna

Med utgångspunkt i informationen vi fått från intervjuerna konstruerade vi en tabell med modulerna och deras krav på operationer med respektive tidskomplexitetskrav. De krav som ställts på värstafallen får inte under några omständigheter brytas.

Modul	Lägg till	Ta bort	Sök	Gå till nästa	Antal element
1-a	O(n)	O(n)	O(log n)	O(1)	256
1-b	O(log n)	O(log n)	O(log n)	--	obegränsad
2	O(n)	O(n)	O(log n)	--	16 384
3	O(n)	O(n)	O(log n)	--	16 384
4-a	O(n)	O(n)	O(log n)	--	512
4-b	O(1) : sist	O(1) : först	O(n)	--	obegränsad
5	O(n)	O(n)	O(log n)	--	512
6	O(1) : sist	O(1) : först	--	--	32 640
7-a	O(n)	O(n)	O(log n)	--	65 535
7-b	O(1) : sist	O(1) : först	O(n)	--	obegränsad

Tabell 2.2 Modulernas krav.värstafall

För att få en bättre överblick grupperade vi de olika kraven i Tabell 2.3. I kolumnen för information om antal element har vi valt att förenkla indelningen genom att endast ange om kravet för lagring avser ett antal som är större eller mindre än 16384. Det är denna gräns som avgör om man kan lagra elementen i en datastruktur utan nyttjande av nodpekare, se avsnitt 2.2.1. Samtliga modulers krav täcks in av de sex kravkategorier som anges i tabellen nedan.

	Lägg till	Ta bort	Sök	Gå till nästa element	Antal element
1	O(log n)	O(log n)	O(log n)	--	> 16384
2	O(n)	O(n)	O(log n)	--	> 16384
3	O(n)	O(n)	O(log n)	O(1)	<= 16384
4	O(n)	O(n)	O(log n)	--	<= 16384
5	O(1) : sist	O(1) : först	O(n)	--	> 16384
6	O(1) : sist	O(1) : först	--	--	> 16384

Tabell 2.3 Gruppering av kraven

2.2.4 Testkrav

Testning av implementationen skall ske med så kallad ”Basic Test”. Basic Test innebär att man skriver små testfunktioner till varje funktion som testar varje enskild funktion för att kontrollera att funktionerna fungerar och verkligen gör det de är avsedda för. Implementation av testkoden är en del av projektet.

2.2.5 Övriga krav

Implementationen ska följa de programmeringsdirektiv som finns angivna i ”Ericssons programming guide” [3]. ”Ericssons programming guide” är en samling regler och rekommendationer, som är till för att varje ny del som skapas ska bli enhetlig med alla andra delar i systemet. Dokumentationsverktyget Doxygen [2] ska användas för dokumentationen av gränssnittet.

3 Analys av kravspecifikationen

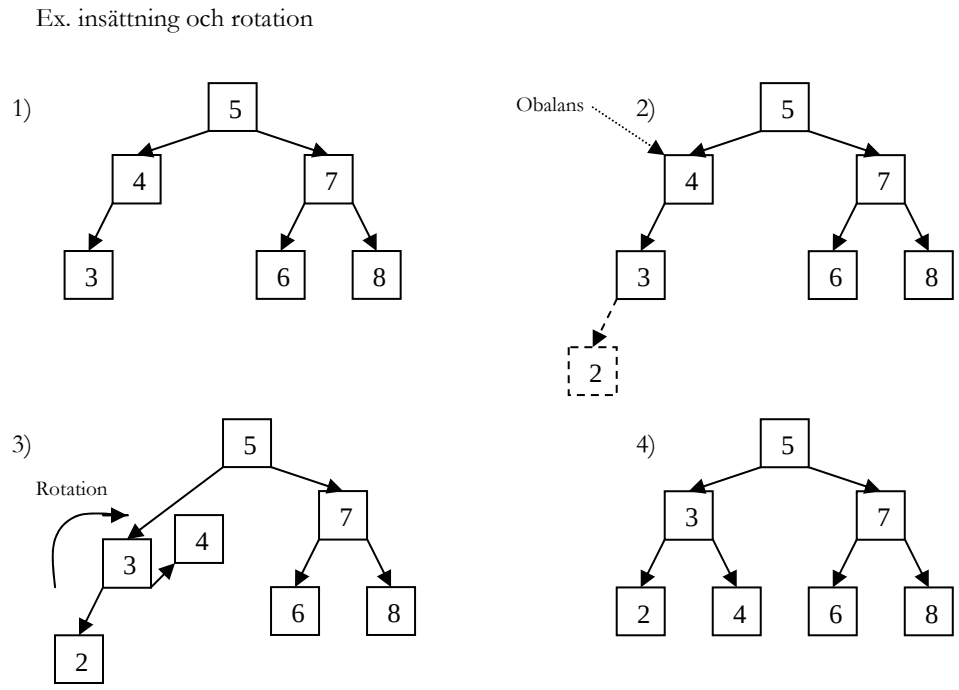
Nästa steg i projektet var att analysera kravspecifikationen, för att få fram ett underlag för designen och implementationen av c-biblioteket. Analysen kom att bestå av två huvuddelar där vi i den första fasen läste in oss på olika datastrukturer och sammanställde tidskomplexiteterna för deras operationer i en tabell. I fas två genomfördes en gruppering av datastrukturerna med avseende på kravspecifikationen. Vidare tog vi beslut om vilka datastrukturer som skulle implementeras.

3.1 Beskrivning av datastrukturer

Med den befintliga lösningen använder sig modulerna endast av statiska datastrukturer. Den teoretiska maxstorleken måste därmed alltid allokeras, se Tabell 2.2. Det är ytterst sällsynt att körningar av systemet kräver lagring av ett stort antal element. De flesta av modulerna använder alltså i normalfallet bara en bråkdel av det allokerade minnet. Med vetskap om de mycket knappa minnesresurserna, se avsnitt 2.2.2, tog vi beslutet att lägga stor vikt vid att undersöka datastrukturer med dynamiska egenskaper. Informationen om de olika datastrukturerna är hämtad ur boken *Data structures and problem solving using C++* [1].

3.1.1 AVL-träd

Ett AVL-träd är ett balanserat binärt sökträd, detta innebär att nivåskillnaden mellan vänster och höger subträd inte får vara större än ett. Om en större nivåskillnad skulle uppstå kommer en rotering att göras för att balansera trädet.

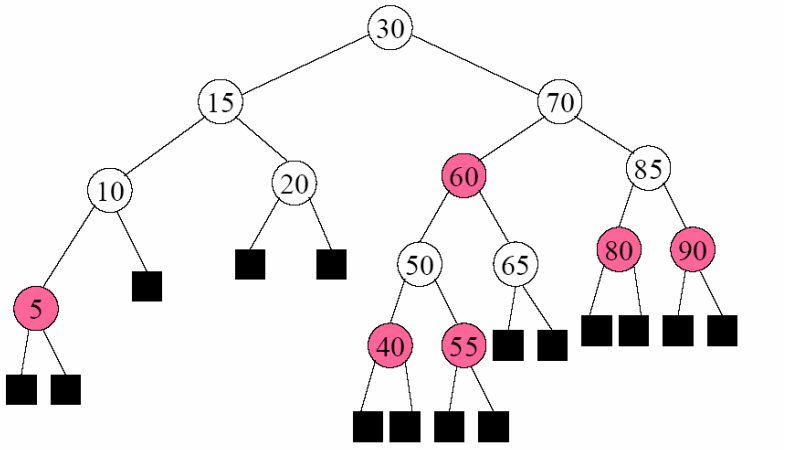


Figur 3.1 AVL-träd, exempel på en rotation

3.1.2 Röd/Svart-träd (R/B-träd)

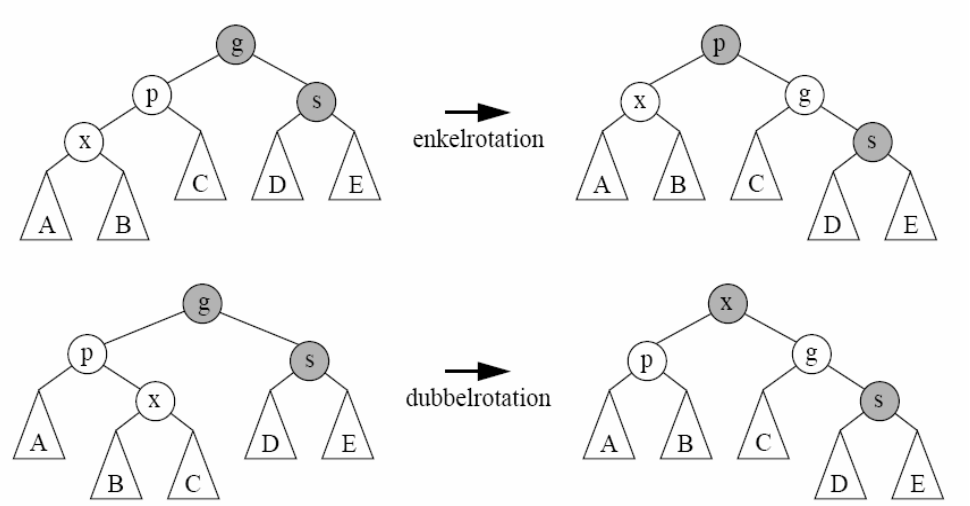
Ett röd/svart-träd är ett balanserat binärt sökträd. För att hålla trädet balanserat sätts vissa regler upp.

1. Varje nod är antingen röd eller svart.
2. Roten är alltid svart.
3. En röd nod får inte ha några röda barn.
4. Varje enkel väg från en nod till ett nedstigande löv innehåller samma antal svarta noder.



Figur 3.2 Exempel på ett R/B-träd

När man stegar ner i trädet för att hitta rätt plats att sätta in ett nytt element på kontrollerar man vilka färgförhållanden som råder. Om man ser en nod med två röda barn färgar man noden röd och dess barn svarta. Nodens förälder kan i detta läge vara röd och för att upprätthålla regel 3 utföres en rotation.



Figur 3.3 R/B-träd, enkel- och dubbelrotation

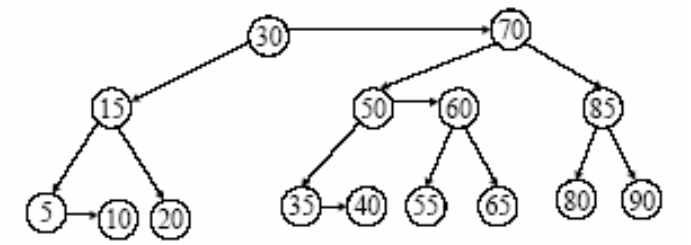
Rotationen skapar inga ytterligare komplikationer eftersom förälderns syskon enligt regel 3 måste vara svarta. Om noden vid insättning hamnar i ett löv färgas den alltid röd.

3.1.3 AA-träd

I ett AA-träd är noderna indelade i nivåer, nivå 1 är längst ner i trädet, där sker alla insättningar.

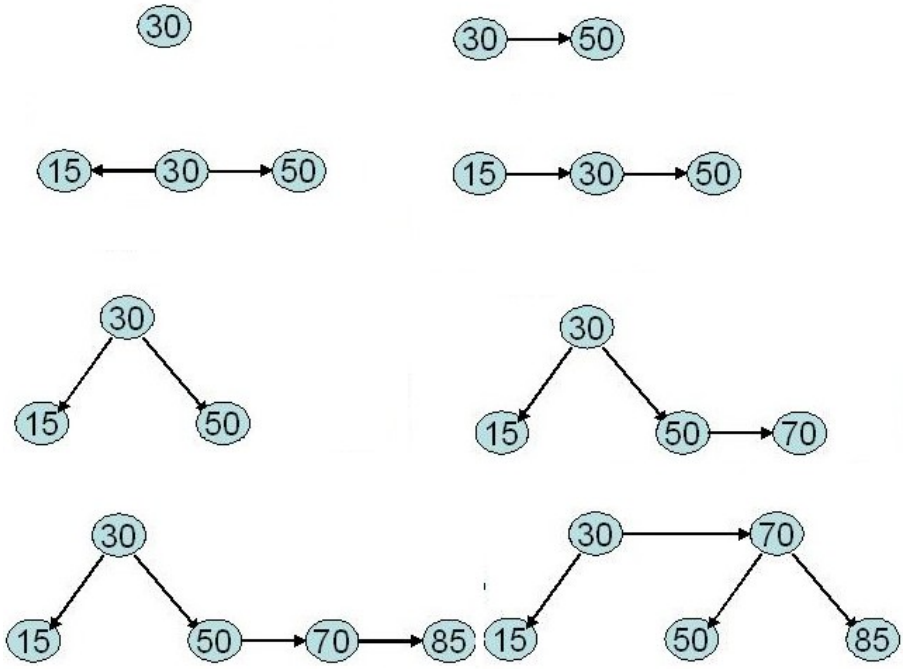
Följande regler gäller:

1. Om noder som tillhör samma väg är på samma nivå är alltid noden längst till vänster förälder och noden längst till höger barn.
2. Längs varje väg ner i trädet finns minst en och som mest två noder per nivå.



Figur 3.4 Exempel på ett AA-träd

För att vid insättning och borttagning hålla trädet balanserat används två operationer, skew och split.



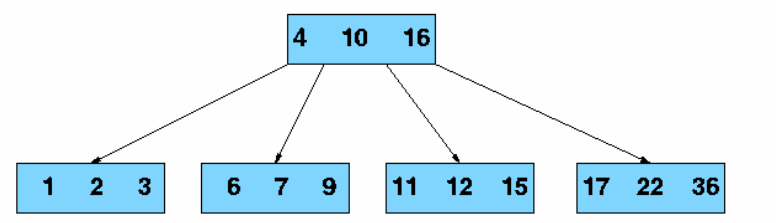
Figur 3.5 Exempel på insättning i ett AA-träd

Vi vill sätta in elementen: 30, 50, 15, 70 och 85 i AA-trädet i nämnd ordning. Vid insättning av de första två elementen (30,15) behöver ingen av balansoperationerna utföras.
För att upprätthålla regel nr 1 måste vi ändra hållet på pilen då insättning av 15 sker. Att byta håll på det viset kallas skew. Vi har nu efter insättningen 3 noder på samma nivå längs samma väg och en split behöver utföras för att upprätthålla regel nr 2. En split innebär att föräldranoden åker upp en nivå relativt sina barn. Insättning av 70 medför inte något brott mot reglerna och därmed behöver ingen av balansoperationerna utföras. Insättningsproceduren för 85 medför endast en splitoperation.

AA-trädet kräver alltså endast två typer av rotationer för att upprätthålla balansen. Detta gör att koden för AA-trädet inte blir lika komplex som för de andra balanserade sökträden. För att göra det möjligt att kontrollera trädets regler måste noderna ha pekare till sina föräldrar vilket ger en större nodstorlek jämfört med AVL-trädet.

3.1.4 B-träd

Ett B-träd är ett balanserat träd där de inre noderna består av söknycklar och alla elementen ligger nere i löven. B-träd används när datadelen i elementet är mycket stor jämfört med söknyckeln eller vid stora datamängder generellt, exempelvis databaser.



Figur 3.6 Exempel på ett B-träd

3.1.5 Array

En array allokeras som en enhet, alltså ligger noderna direkt efter varandra i minnet. Detta gör det möjligt att nå samtliga noder via adresseringen.

0	1	2	3	4	5	6	7	8	9
3	5	6	7	8	9				

Figur 3.7 Array

3.1.6 Hashtabell

Hashtabellen är en variant på arrayen där man stoppar in värdet på en position som räknas fram med hjälp av en hashfunktion. På samma sätt kan man snabbt komma åt just det värde man söker. På detta sätt uppnås direktaccess där man vet vilka nycklar elementen har.

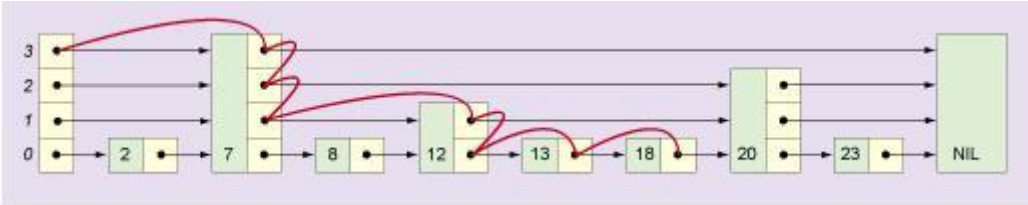
0	1	2	3	4	5	6	7	8	9
10	1		3	14	5			8	

Figur 3.8 Hashtabell

Det är svårt att bestämma bästa och värstafall för hashtabellens operationer. I avsnitt 3.2.2 förs en diskussion kring detta.

3.1.7 Skiplista

En skiplista är en länkad lista där elementen har olika höjd. Höjden på ett element slumpas fram vid insättning. Detta ger en bättre väntad tidskomplexitet för sökning jämfört med en vanlig länkad lista. Sökalgoritmen är en typ av binärsökning där man för värden större än det sökta stegar nedåt i elementet. För värden mindre än det sökta stegar man framåt i listan. På följande sida visas ett exempel på en sökning.

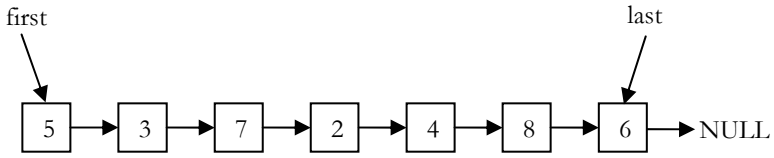


Figur 3.9 Skiplista, sökning efter värdet 18. Sökningen börjar i ”header”-noden på översta nivån (3). Värdet för denna pekare är 7 (<18) och man stegar framåt till 7-noden. Värdet för denna pekare är NIL (> 18). Man stegar då ner en nivå i 7-noden. Värdet för denna pekare är 20 (>18) och man stegar därför ner ytterligare en nivå. Man fortsätter på samma sätt och når tillslut 18-noden

I skiplistan får man direktaccess till fler element än vad man får i en vanlig länkad lista på bekostnad av fler nodpekare. Den väntade tidskomplexiteten för sökning är O(log(n)) men värstafallet har tidskomplexiteten O(n).

3.1.8 Länkad lista

En länkad lista är en datamängd där man lagrar värdena i en pekarsekvens. För att nå värden måste man alltså stega igenom datamängden sekventiellt utifrån de indexpekare man definierat upp.



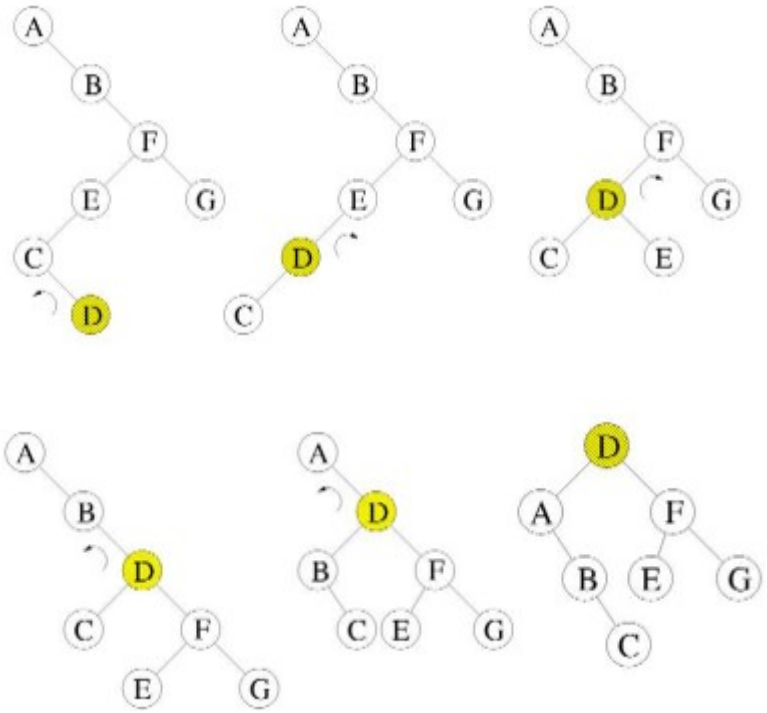
Figur 3.10 Exempel på en länkad lista med två indexpekare.

3.1.9 BST

Ett binärt sökträd består av sammankopplade noder, varje nod innehåller ett värde samt vänster- och högerpekare. Trädet är sorterat så att mindre värden finns i nodens ena subträd och större värden i det andra.

3.1.10 Splay-träd

Ett splay-träd är ett träd utan någon särskild balanseringsregel. Istället utförs en splayfunktion som roterar upp den senast refererade noden så att den hamnar i roten. Tanken med detta är att man vill optimera söktiden för de noder som nyttjas ofta.

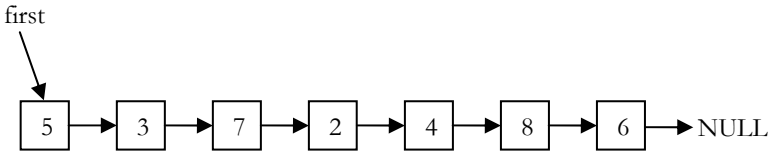


Figur 3.11 Splayträd

3.1.11 Stack

I en stack har man enbart access till det översta elementet i datamängden.

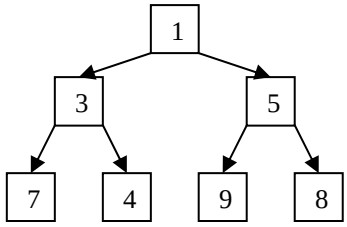
En stack är en datamängd där man bara hanterar värdena som ligger överst, detta innebär då att värdet som sist lades in är det värde som först tas bort – LIFO (Last In First Out).



Figur 3.12 Stack

3.1.12 Prioritetskö

I en prioritetskö sorteras värdena in i en heap, värdet med högst prioritet ligger alltid först men ordningen på resterande värden är inte alltid bestämd. Värdena är alltså inte sorterade.



Figur 3.13 Heap

3.1.13 Sammanställning av informationen om de olika datastrukturerna

I tabellen nedan visas en sammanställning av informationen om de olika datastrukturerna.

	Datastr	Lägg till	Ta bort	Sök	Gå till nästa	Lagring	Nodstlk
1	**AVL- träd**	O(log n)	O(log n)	O(log n)	--	obegr.	12 byte
2	**R/B- träd**	O(log n)	O(log n)	O(log n)	--	obegr.	16 byte
3	**AA- träd**	O(log n)	O(log n)	O(log n)	--	obegr.	16 byte
4	**B-träd**	O(log n)	O(log n)	O(log n)	--	obegr.	??
5	**Array**	O(n)	O(n)	O(log n)	O(1)	<=16384	4 byte
6	**Hashtab**	***	***	***	**	<=16384	4 byte
7	**Skiplist**	O(n)	O(n)	O(n)	O(1)	obegr.	?8 byte
8	**Länkad list**	O(n)	O(n)	O(n)	O(1)	obegr.	8 byte
9	**BST**	O(n)	O(n)	O(n)	--	obegr.	12 byte
10	**Splay-träd**	O(n)	O(n)	O(n)	--	obegr.	?12 byte
11	**Stack**	O(1)*	O(1)*	--	--	obegr.	8 byte
12	**Priokö**	O(nlogn)	O(nlogn)	--	--	obegr.	?12 byte

*Tabell 3.1 Tidskomplexiteter för utvalda operationer för datastrukturerna,(*s = sist insatta element, **ett möjliggörande av operationen kräver en komplex pekarstruktur som måste uppdateras vid varje förändring av datastrukturens datamängd *** härleds i en djupare analys (se avsnitt 3.2.2) Nodstlk = nodstorlek. Nodstorleken visar hur mycket minne som krävs per nod för respektive datastruktur.*

3.2 Gruppering av datastrukturerna

Nästa fas i analysen bestod i att, utgående från informationen vi sammanställt i föregående kapitel 3.1, bestämma vilka datastrukturer som var bäst lämpade för projektet. Vi började med att se över de kombinationer av krav vi kom fram till i avsnitt 2.2.3

Lägg till	Ta bort	Sök	Gå till nästa element	Antal element
O(log n)	O(log n)	O(log n)	--	> 16384
O(n)	O(n)	O(log n)	--	> 16384
O(n)	O(n)	O(log n)	O(1)	<= 16384
O(n)	O(n)	O(log n)	--	<= 16384
O(1) : sist	O(1) : först	O(n)	--	> 16384
O(1) : sist	O(1) : först	--	--	> 16384

Tabell 2.3 Gruppering av kraven (från avsnitt 2.2.3)

Grundat på Tabell 2.3 och Tabell 3.1 kunde vi dela in de olika datastrukturerna i olika kravgrupper.

Det obalanserade binära sökträdet (9), splayträdet (10), stacken (11) och prioritetskön (12) kan vi direkt se att vi inte ska gå vidare med i projektet då dessa inte klarar att uppfylla några av kraven listade i *Tabell 2.3* .

Genom att definiera indexpekare för första och sista elementet får man direktaccess till första respektive sista elementet och det är då möjligt att med en länkad lista (8) tillgodose krav 5 och 6. Eftersom den länkade listan är den enda som tillgodoser krav 5 och 6 kunde vi redan i detta skede bestämma oss för att implementera den.

AVL-trädet (1), R/B-trädet (2), AA-trädet (3) och B-trädet (4) tillgodoser krav 1, 2 och 4. Då de balanserade träden är ganska lika varandra vad gäller tidskomplexitet och minnesutnyttjande krävs en mer noggrann undersökning för att utröna vilken av dessa datastrukturer som skall implementeras.

Det är av intresse att undersöka hashtabellen (6) närmare då det finns möjligheter att även denna kan tillgodose krav 4. Hashtabellen har en väntad tidskomplexitet som är lägre än O(log n) men det krävs en djupare analys för att ta reda på vad som gäller för tidskomplexiteten för värsta fallet.

Arrayen (5) tillgodoser krav 3 och 4. Eftersom arrayen är den enda som tillgodoser krav 3 kunde vi redan i detta skede bestämma oss för att implementera den.

Krav	Datastruktur
1	1, 2, 3, 4
2	1, 2, 3, 4
3	5, 6?
4	1, 2, 3, 4, 5, 6?
5	7
6	7
tillgodoser inget krav	8,9,10,11,12

Tabell 3.2 Gruppering av datastrukturerna

3.2.1 Balanserade sökträd

För att krav 1 och 2 ska kunna tillgodoses måste minst ett av de balanserade träden implementeras. Skillnaderna mellan de olika balanserade sökträden är inte särskilt stora. Det finns emellertid intressanta skillnader i minnesutrymme per nod. Utifrån informationen i avsnitt 3.1 konstruerade vi en tabell med skillnaderna.

Träd	**Nodstorlek**
AVL-träd	Liten
R/B-träd	Stor
AA-träd	Stor
B-träd	Stor

Tabell 3.3 Balanserade sökträd, relativa strukturskillnader

AVL-trädet har vi arbetat med tidigare, detta innebär mindre inläsningstid för att komma igång med att implementera detta träd jämfört med de andra träden. Med vetskap om att minneskraven är hårt ställda samt att tiden för arbetet är knapp kom vi fram till att vi skulle implementera AVL-trädet.

3.2.2 Indexerade datastrukturer

Vi fick under intervjuerna flera gånger önskemålet att vi skulle skapa en hashtabell, och när vi letade efter möjliga ADT:er såg vi att hashtabellen i vissa fall hade bättre tidskomplexitet än arrayen. På grund av detta valde vi att undersöka hashtabellen lite närmare. Söktidskomplexiteten för hashtabellen är i normalfallet O(1), men på grund av kollisionerna som kan uppstå och omsökningarna som detta innebär, så kan värsta fallet för sökning bli så

illa som O(n). För att kunna använda oss av en hashtabell måste vi först hitta en implementation som vi kan bevisa alltid har ett värsta fall för sökningen på max O(log(n)).

Figur 3.14 Procedur för inplacering av ett element i en hashtabell

För att kunna skapa en hashfunktion som är optimal behöver man information om värdena som lagras i tabellen, detta för att kunna hitta en hashnyckel som effektivt undviker möjliga kollisioner. Eftersom den tabell vi kommer att skapa ska kunna lagra vilken datatyp som helst, så kan vi heller inte på förhand veta spridningen på de nyckelvärden som kommer lagras i tabellen. Detta innebär att vi inte kan skapa en optimal hashfunktion [1].

Nästa steg är att ta reda på vilken storlek som ger den bästa prestandan i förhållande till både minneskrav och tidskrav. För att få så lite kollisioner som möjligt bör storleken på hashtabellen vara ett primtal, och hela tiden minst vara dubbelt så stor som antalet värden i tabellen. På grund av minnesbristen så måste vi göra hashtabellen, liksom arrayen, dynamisk. För att lösa detta måste alltså omallokering ske under körning. Första steget för omallokeringen är att skapa en ny hashtabell. Eftersom omallokeringen är ganska kostsam bör det minst allokeras en tabell som är dubbelt så stor som den befintliga, samt att storleken som sagt ska vara ett primtal. Kostnaden för att ta fram ett primtal är $O(\log(n)*n^{½})$[1].

Andra steget är att skapa en ny hashfunktion, och tillsist görs en omhashning dvs. alla värden läggs in i tabellen på nytt med den nya hashfunktionen. Omhashning har tidskomplexiteten O(n), alltså får hela algoritmen för storleksökningen tidskomplexiteten O(n). I vårt fall får hashtabellen liknande tidskomplexitet som den dynamiska arrayen. Skillnaden är att arrayen har ett bättre värsta fall vid sökning. Hashtabellen har inga egentliga fördelar jämfört med arrayen. Den har ungefär samma dynamiska egenskaper som den dynamiska arrayen. Ytterligare en nackdel med hashtabellen är att den, för att vara någorlunda effektiv, kräver mer minne än arrayen. Eftersom hashtabellen ska vara dynamisk blir insättningsprestandan dålig på grund av att man måste beräkna en ny hashnyckel och hasha om alla värden när man vill öka antalet platser i hashtabellen. Vi såg inga avgörande fördelar med hashtabellen eftersom alla krav kan uppfyllas utan den. Dessutom är det mycket tidskrävande att ta fram en

bra hashnyckel och sedan visa att den leder fram till att kravet O(log(n)) uppfylls. Vi tog därför beslutet att inte ta med hashtabellen i designen av c-biblioteket.

3.2.3 Slutsatser

Vi kom fram till att vi behövde designa och implementera minst tre olika datastrukturer för att tillgodose samtliga krav.

Datastruktur	Krav
AVL-träd	`1, 2, 4`
Array	`3, 4`
Länkad lista	`5, 6`

Tabell 3.4 Datastrukturer som skall designas samt vilka krav de tillgodoser

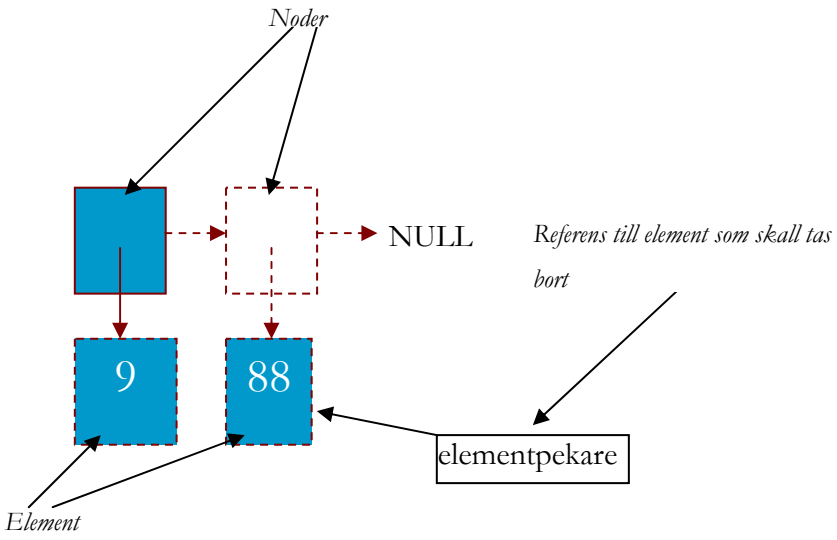
Krav 4 täcks alltså in av både AVL-trädet och arrayen. Eftersom slutprodukten kommer att bli ett bibliotek är det upp till klientprogrammeraren att avgöra vilken(vilka) datastruktur(er) han(hon) tycker lämpar sig för just sin applikation.

4 Design

Vi kommer i detta kapitel att beskriva specifika designdetaljer för våra datastrukturer.

Alla datastrukturer består av noder som innehåller en pekare till elementet. Eftersom biblioteket skrivs i programspråket C, måste noden bestå av en elementpekare av typen void så att lagring av godtycklig datatyp möjliggörs, se avsnitt 1.1.
På grund av att vi inte vet vad som kommer att lagras i datastrukturen måste klientprogrammeraren, för att göra det möjligt för oss att jämföra elementen, skapa en jämförelsefunktion för varje datatyp som ska lagras.

Funktionen tar emot elementen och tar ut det eller de fält som klientprogrammeraren definierat som nyckelvärde. Klientprogrammeraren ansvarar för elementets minnesallokering. Vid borttagning av en nod sparas först en pekare till elementet. Klientprogrammeraren kan sedan anropa en funktion för att nå pekaren och ta bort elementet.



Figur 4.1 Exempel på borttagning av element.

4.1 AVL-träd

De operationer som ska kunna utföras på AVL-trädet är insättning, borttagning av element samt sökning.
AVL-trädets samtliga operationer har tidskomplexiteten O(log(n)).
Varje nod består av två nodpekare, en elementpekare och två fält som håller reda på höjden på vänster och höger subträd. Varje nod allokeras som ett enskilt block, detta innebär att inte finns några begränsningar på antalet noder som kan lagras. Designen kommer för övrigt att vara som ett vanligt AVL-träd [1].

4.2 Dynamisk Array

De operationer som ska kunna utföras på arrayen är insättning, borttagning, sökning samt gå till nästa element.
Arrayen är implementerad för att optimera sökningen. Därför sorteras alltid elementen vid insättning. När arrayen är sorterad kan man tillämpa binärsökning vilken har tidskomplexiteten O(log n). Insättning och borttagning har tidskomplexiteten O(n). Värsta fallet inträffar när man sätter in eller tar bort den första noden i arrayen - då måste alla befintliga noder flyttas ett steg.
När arrayen är full och ett nytt värde sätts in ska antalet platser i arrayen öka med en användardefinierad faktor, grundinställningen är att storleken på arrayen fördubblas, samma faktor används då arrayen ska minskas, och även här är grundinställningen faktor 2.
På en av plattformarna är blockstorleken begränsad till 64k, se avsnitt 2.2.2. För att garantera plattformsoberoende är därför det maximala antalet element som kan lagras i en instans av arrayen 16384. Om man försöker förstora arrayen i ett läge där den nya arrayen skulle bli större än maxstorleken 16384, får man alltid en array med 16384 platser tillbaka. Tröskelvärdet för när arrayen ska halveras bestäms av klientprogrammeraren. Om inget tröskelvärde anges halveras storleken på arrayen då antalet upptagna platser i arrayen minskat till ¼.

Arraystorleken kan bara minska ner till ett av klientprogrammeraren förutbestämt värde, om inget sådant värde anges så kan arrayen bara minskas ner till ursprungsstorleken. Värdena sorteras vid insättning i stigande ordning.

4.3 Länkad lista

Operationer som ska kunna utföras på den länkade listan är insättning, borttagning och sökning. Insättning ska ske i slutet av listan. Borttagningen ska kunna ske på godtycklig position.

Genom att definiera indexpekare för första och sista insatta element blir tidskomplexiteten O(1) vid insättning sist respektive borttagning först i listan. Borttagning av element på godtycklig position i listan föregås av en sekventiell sökning och får därmed tidskomplexiteten O(n).

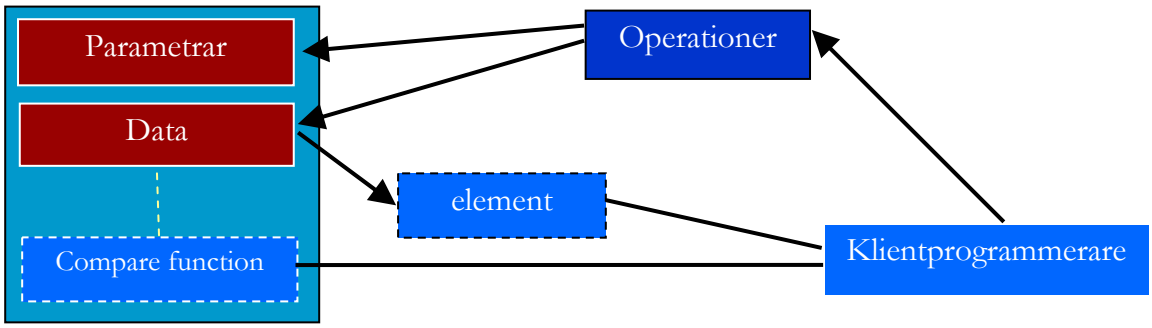
Listan implementeras som en enkellänkad lista med en pekare till första och sista noden. Varje nod allokeras som ett enskilt block, detta innebär att inte finns några begränsningar för hur många noder som kan lagras.

5 Implementation och test

Detta kapitel ger en detaljerad beskrivning av implementationen av datastrukturerna samt en beskrivning över de testrutiner vi utförde.

5.1 Detaljerad beskrivning implementationen

För varje datastruktur finns ett antal parametrar som måste hållas reda på. Detta åstadkommes genom att parametrarna grupperas ihop med datastrukturen i en struct. Datastrukturen och dess parametrar får inte modifieras direkt utan detta sker genom operationer som tillhör datastrukturen. Detta möjliggör framtida modifieringar av c-biblioteket utan att det påverkar användargränssnittet. Man skulle exempelvis kunna implementera en lista m h a en array istället för en länkad lista.



Figur 5.1 Schema över relationen mellan datastruktur och klientprogrammerare. (med begreppet data avses de lagrade elementens noder)

Vid initieringen anger klientprogrammeraren vilka värden de olika parametrarna ska anta. Från klientprogrammerarens sida ses structen som själva datastrukturen, och när en datastruktur behövs skapar klientprogrammeraren en ny instans av structen. Eftersom vi inte vet vilken typ av element som skall sparas får klientprogrammeraren själv sköta allokering och avallokering av minnesutrymmet för elementet.

Då insättning sker skickas bara pekaren till det skapade elementen med till datastrukturen. Vid borttagning av en nod sparas en pekare till elementet i structen som klientprogrammeraren sedan kan nå genom att anropa en operation.

```
int insert(void* elementPointer, struct DataStructure* list)
{
    nodeType newNode;

    /* creation of the new node, allocation of memory */
    newNode = (nodeType)Malloc(list->userId, sizeof(nodeType));

    if(newNode == NULL)
        return ERRORMSG;

    newNode->elementPointer = elementPointer;
    newNode->next = NULL;

    /* insertion of the new node */
    if(isEmpty(list))
    {
        list->first = list->last = newNode;
        list->length = 1;
        return 0;
    }
    else
    {
        list->last->next = newNode;

        /* the new node becomes the last_sp node*/
        list->last = newNode;
        list->length++;
    }

    return 0;
}
```

Figur 5.2 Pseudokod för insättning av ett element på sista platsen i en lista

```
int remove(struct DataStructure* list)
{
    nodeType tmpNodePointer;

    if(isEmpty(list))
        return ERRORMSG;

    if(list->first == list->last)
    {
        list->elementToBeRemoved = list->last->elementPointer;

        kill(list, list->last);

        list->first = list->last = NULL;
        list->length = 0;
    }
    else
    {
        tmpNodePointer = list->first->next;

        list->elementToBeRemoved = list->first->elementPointer;

        kill(list, list->first);

        list->first = tmpNodePointer;
        list->length--;
    }

    return 0;
}
```

Figur 5.3 Pseudokod för borttagning av ett element på första platsen i en lista

Att vi inte vet vad som sparas leder också till att vi inte vet hur elementen ska jämföras, därför får klientprogrammeraren skriva funktionen som jämför två element. Vid initieringen av en instans sätter han(hon) en referens till den funktionen i structen.

```
h-file:
...
typedef int (*compareFuncPointerType) (const void*, const void*);
...

c-file:
int init(compareFuncPointerType compareFuncPointer, struct DataStructure* ds, ...)
{
    ...
    ...
    ds->compareFuncPointer = compareFuncPointer;
    ...
    return 0;
}
```

Figur 5.4 Pseudokod för delar av initieringen av en datastrukturinstans

```
int compareInt(const void* p1, const void* p2)
{
    int i, j;

    i = *((int*) p1);
    j = *((int*) p2);

    if(i > j)
        return LARGER;
    else if(i < j)
        return SMALLER;

    return 0;
}
...
int f(...)
{
struct DataStructure* ds;
compareFuncPointerType compareFuncPointer = compareInt;
init(compareFuncPointer, &ds, ...);
...
}
```

Figur 5.5 Exempel på hur en klientprogrammerare kan definiera en funktion för att jämföra element

```
int search(void* elementInPointer,
           void** elementOutPointer,
           struct DataStructure* tree)
{
    tree->currentNode = tree->root;

    while(NULL != tree->currentNode)
    {
        if(SMALLER ==
            tree->compareFuncPointer(elementInPointer,
                                     tree->currentNode->elementPointer))
        {
            tree->currentNode_sp = tree->currentNode->left;
        }
        else if(LARGER ==
                tree->compareFuncPointer(elementInPointer,
                                         tree->currentNode->elementPointer))
        {
            tree->currentNode_sp = tree->currentNode->right;
        }
        else
        {
            *elementOutPointer = tree->currentNode->elementPointer;
            return 0;  /* value found */
        }
    }

    *elementOutPointer = NULL;
    return ERRORMSG; /* value not found */
}
```

Figur 5.6 Pseudokod för binärsökning i ett träd.

Varje gång någon modul läser eller skriver till minnet måste denna identifera sig med ett unikt ID. Vid initieringen av en datastruktur kopplas den aktuella modulens ID till den skapade

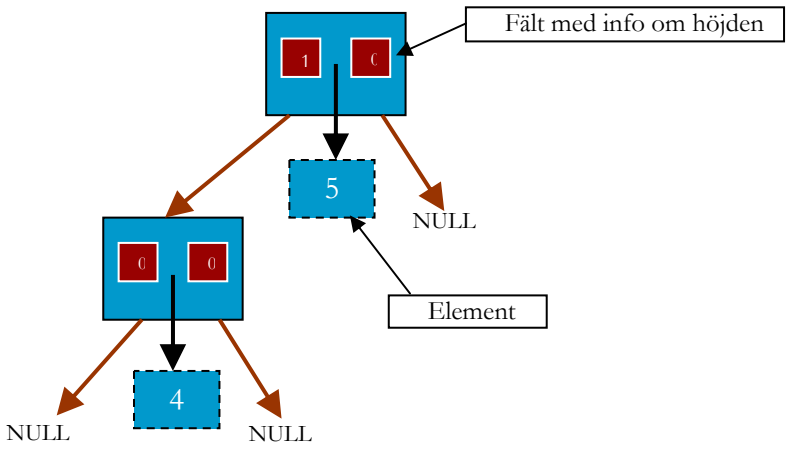
instansen. Detta ID används som identifierare när funktionerna i datastrukturerna anropar delar i Common Parts.

5.1.1 AVL-träd

AVL-structen innehåller:

- en pekare till AVL-trädet
- en funktionspekare till jämförelsefunktionen
- en nodpekare till den nod som innehåller det element man senast nyttjade
- ett fält som anger modulens användar-ID
- en elementpekare till det element som tillhör i den senast borttagna noden.

När klientprogrammeraren skapar ett nytt AVL-träd skapas en ny instans av AVL-strukten.



Figur 5.7 AVL-trädets noduppbyggnad.

Parametrarna sätts och klientprogrammerarens ID associeras med AVL-structen. Eventuell balansering sker vid insättning och borttagning. När man sätter in eller tar bort en nod görs ett rekursivt anrop som stegar igenom trädet och söker upp rätt position. Efter att noden satts in eller tagits bort uppdateras fälten som håller reda på höjdskillnaden i de noder som traverserats. Höjdskillnaden kontrolleras och om den är större än ett utförs en balansering av trädet i form av en enkel- eller dubbelrotation.

5.1.2 Dynamisk array

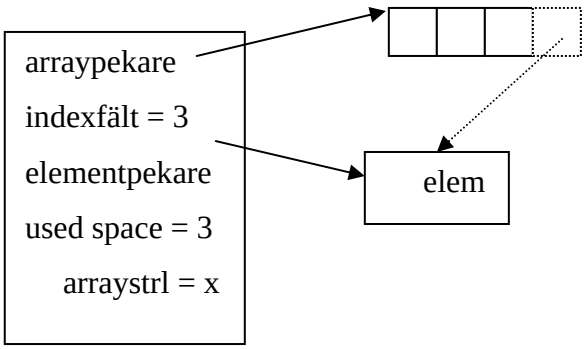
Den dynamiska arrayen abstraheras med en array-struct.

Array-structen innehåller:

- en pekare till arrayen
- en funktionspekare till jämförelsefunktionen
- ett fält som anger index till det element man senast nyttjade

- ett fält som anger modulens användar-ID
- en pekare till det element man senast tog bort
- ett fält som anger antalet platser i arrayen
- ett fält som anger tröskelvärdet för arrayen (vid vilket värde arrayen ska minska i storlek)
- ett fält som anger hur liten arrayen får vara
- ett fält som anger med vilken faktor arrayen ska växa
- ett fält som anger antalet upptagna platser i arrayen
- ett fält som anger hur stor arrayen får vara

När klientprogrammeraren skapar en ny array skapas en ny instans av arraystructen. Parametrarna sätts och klientprogrammerarens ID associeras med arraystructen. Om klientprogrammeraren inte anger tröskelvärde etc., sätts de till ett standardvärde, se kap 4.2



Figur 5.8 Exempel på en arraystruct

Vid insättning sorteras elementet in på rätt plats. Innan själva insättningen sker en kontroll där man jämför arrayens storlek med antalet använda platser. Om arrayen är full anropas en funktion som allokerar fler platser för arrayen. Allokeringen styrs av en parameter och är ”faktorbaserad”, dvs. arrayen växer med en multipel. Element med högre nyckelvärden flyttas en position uppåt i arrayen. Vid borttagning sparas först en pekare till elementet, sedan flyttas alla element med högre nyckelvärden ner ett steg. Efter borttagning av ett element sker en kontroll. Om antalet upptagna platser i arrayen understiger tröskelvärdet anropas en funktion som avallokerar en viss del av arrayens platser. Linjärsökning används vid insättning och borttagning. Detta ger ett bättre värsta fall än binärsökning. Insättning på första positionen i arrayen ger vid; linjärsökning + framflyttning: O(1*n), binärsökning + framflyttning: O(log(n) * n). Man når nästa element i datastrukturen genom att anropa en funktion med aktuellt elements index som inparameter. Funktionen returnerar detta index + 1.

5.1.3 Länkad lista

List-structen innehåller:

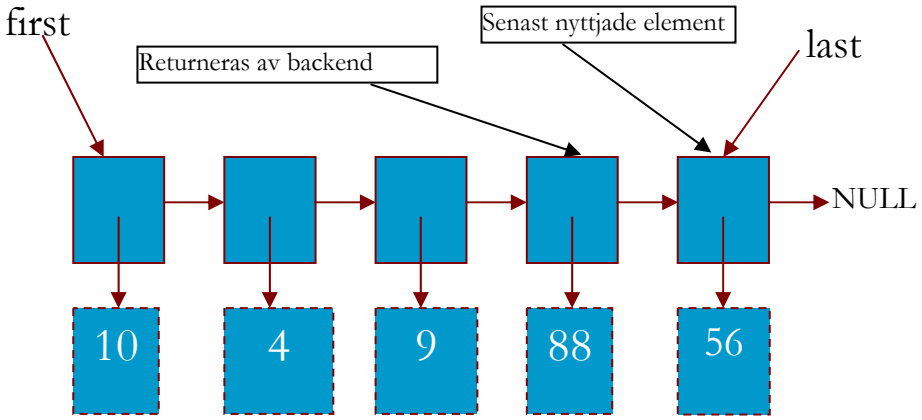
- pekare till första och sista elementet i listan
- en funktionspekare till en jämförelsefunktion
- en pekare till noden som innehåller det senast nyttjade elementet
- en elementpekare till det element man senast tog bort

Eftersom det finns ett krav på att kunna ta bort element även mitt inne i listan, så har vi skapat en linjärsökningsfunktion. Linjärsökningen består av två funktioner – frontend och backend. Frontend utgör gränssnittet mot klientprogrammeraren medan backend används internt av biblioteket. Backendfunktionen returnerar pekare till noden före den sökta noden. Denna lösning gör det möjligt att implementera listan enkellänkad. Om vi bara hade haft en funktion för hantering av linjärsökningen hade vi behövt implementera listan dubbellänkad. (När man gör på detta sätt så returneras ”rätt” information. Man når de två noder man vill komma åt: current och noden innan. För att ta bort current-noden behöver man styra om pekaren från noden innan till current->next.)

```
int linearSearch(struct DataStructure* list,
                 void* elementPointer)
{
    if(isEmpty(list))
        return ERRORMSG; //treated as element not found

    if(EQUAL ==
        list->compareFuncPointer(elementPointer,
                                 list->first->element_p))
    {
        //store the nodePointer to the element that was found:
        list->currentNode = list->first;
        return 0;
    }

    if(EQUAL == linearSearchBackEnd(list, elementPointer))
    {
        list->currentNode = list->currentNode->next;
        return 0;
    }

    list->currentNode = NULL;
    return ERRORMSG; //element not found
}

int linearSearchBackEnd(struct DataStructure* list,
                 void* elementPointer)
{
    list->currentNode = list->first;

    /* list->first is already checked in linearSearch, so skip it */
    while(list->currentNode->next != NULL)
    {
        if(EQUAL ==
            list->compareFuncPointer(elementPointer,
                                     list->currentNode->next->elementPointer))
        {
            return 0; //element found
        }

        list->currentNode = list->currentNode->next;
    }

    return ERRORMSG; //element not found
}
```

Figur 5.9 Pseudokod som beskriver implementationen av linjärsökning för den länkade listan

Frontendfunktionen fungerar som en vanlig linjärsökning, dvs. den får in ett nyckelvärde och returnerar pekaren till det matchande elementet. Frontend använder sökfunktionen som vi implementerar i backend. Frontend returnerar sedan pekaren som pekar på det elementet framför det element pekaren från backend pekar på (se figur).



Figur 5.10 Exempel på en Länkad lista. Senast utförda operation: Sökning efter 56. Frontend tar emot en pekare av backend och returnerar denna→next.

Specialfallet att det sökta värdet ligger först i listan tas om hand innan backend anropas. Frontend returnerar i detta fall pekaren till första noden.

5.2 Testrutiner

Innan implementeringen av de olika funktionerna i C-biblioteket skapas testfunktioner. Dessa testfunktioners enda uppgift är att anropa den riktiga funktionen med den indata som funktionen behöver. Indatan som ska skickas med till funktionen väljs ut till något rimligt och hårdkodas sedan i testfunktionen. Efter det att en testfunktion skapats implementeras den riktiga funktionen. För varje punkt i kravspecifikationen skapas en testfunktion, och på detta sätt testas alla funktionerna var och en för sig. Testfunktionerna utgör en del i biblioteket med datastrukturerna. Dessa är till för att klientprogrammeraren skall kunna testa mjukvara enligt företagets mall.

6 Resultat och rekommendationer

Målet var att skapa ett bibliotek med generiska datastrukturer som tillsammans uppfyller de olika krav som ställdes av de olika modulansvariga. Det färdiga biblioteket med datastrukturerna uppfyller målet men är ändå inte särskilt omfattande då det bara kom att bestå av tre olika datastrukturer. Detta är dock ur företagets synvinkel en fördel då det nuvarande systemet är onödigt komplext.

De fyra delmålen har uppfyllts på följande sätt:

1. Vi har bemött kravspecen i en analys där vi tog fram datastrukturer som täckte in de ställda kraven.
2. Vi har skrivit en implementation proposal (designspecifikation) enligt företagets mall. Designspecifikationen finns även med i denna rapport i en sammanfattad version, se kap 4.
3. Implementation av datastrukturerna har genomförts med avseende på ovan nämnda implementation proposal.
4. C-biblioteket dokumenterades m h a doxygen [2]. Dokumenteringen utfördes under programmeringen genom tilläggande av kommentarer med ett speciellt doxygensyntax.

Eftersom vår lösning är inriktad mot dynamiska datastrukturer ger den stora vinster vad det gäller minnesåtgången. En målsättning vi hade i början av projektet var att vi även skulle förbättra prestandan, dvs. tidskomplexiteten för de olika operationerna som modulerna utför.

Vi fann att AVL-trädet kommer att utgöra en avsevärd förbättring då insättning och borttagning av noder får en bättre tidskomplexitet, från O(n) till O(log(n)). Biblioteket har en hög modularitet, och att lägga till eller byta ut funktioner eller att lägga till ytterligare datastrukturer utgör inte några problem. Varje datastruktur är uppbyggd av många små moduler och varje modul har ett bestämt gränssnitt, och följs det gränssnittet kan i princip allt bytas ut. Sökningsmodulen skulle till exempel kunna bytas ut trots att den används av såväl insättning som borttagning, bara man behåller det förutbestämda gränssnittet.

Utav anledningen att en hashtabell ifrån många håll efterfrågades under den tiden då arbetet pågick ges nu ett förslag på hur man skulle kunna vidareutveckla biblioteket med en just sådan. Problemet med hashtabellen var att något värsta fall under O(n) inte kunde garanteras, detta på grund av att man måste veta vilka värden som kommer att lagras för att kunna skapa en optimal hashfunktion. För att lösa problemet kan man skapa endast själva

skalet för hashtabellen, det vill säga det yttre gränssnittet mot den och de grundläggande delarna för den som till exempel dess dynamiska egenskaper. Tanken är sedan att klientprogrammeraren skriver själva kärnan, det vill säga hashfunktionen. På detta sätt kan en optimerad hashtabell uppnås. Även kollisionshanteringen skulle kunna ställas in eller implementeras av klientprogrammeraren. På detta sätt kan klientprogrammeraren optimera minnesåtgången för att passa just den elementyp han avser lagra.

7 Summering av projektet

Detta projekt har gett oss en god inblick i hur det är att jobba på ett företag. Vi upptäckte att det var mycket viktigt att ta god tid på sig vid de olika intervjuerna och verkligen se till att förstå allt som sades och skriva ner alla krav. Om vi hade gjort detta hade vi sparat mycket tid. I projektet fick vi även göra många egna tolkningar utifrån den information vi erhöll från intervjuerna. Detta var en nödvändighet då de flesta modulansvariga inte visste så mycket om hur de olika modulerna hängde ihop i systemet.

Det dröjde ett tag innan vi fick grepp om projektets utformning. Informationen i intervjuerna användes alltså även till att skapa en tydligare bild av projektet. När vi fått ett bättre grepp om vad som skulle göras insåg vi också att vissa krav som framkommit i intervjuerna låg utanför projektet.

I början av projektet var det ganska svårt att uppskatta hur mycket tid man skulle behöva lägga ner. Svårast var dock att veta hur tiden skulle fördelas mellan projektets olika delmoment.

8 Referenser

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

[2] http://www.doxygen.org, 13 dec 2004

[3] http://www.ericsson.com/support/telecom/part-e/e-2-2.shtml, 4 feb 2002

[4] http://www.enea.com/html/kurser-en/oseintro.jsp, 9 aug 2004

[5] http://www.ericsson.com/support/telecom/part-g/g-8-3.shtml, 4 feb 2002

[6] http://www.ccpu.com/pages/solutions/products/trilliumProtocolTechnology/ss7/page.html, 2004