BACHELOR'S THESIS 2022

Intervallträd som grund vid tillgänglighetsanalys

Marcus Sjökvist

ISSN 1651-2197
LU-CS/HBG-EX: 2022-01

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

Intervallträd som grund vid tillgänglighetsanalys

LUNDS UNIVERSITET
Campus Helsingborg

LTH Ingenjörshögskolan vid Campus Helsingborg
Institutionen för datavetenskap

Examensarbete:
Marcus Sjökvist

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Lunds universitet
Lund 2022

Sammanfattning

Det svenska transportsystemets infrastruktur kräver många olika tekniska enheter av diverse olika typer för att fungera såsom exempelvis sensorer, kameror, digitala skärmar och routrar. Det är Trafikverket som ansvarar för dessa och för att säkerställa att de når upp till en god standard så finns det ett *Service Level Agreement* (SLA) som ställer krav på driftsäkerheten. Dessa krav är olika beroende på tjänst och utrustning. För att vara mätbara använder dessa krav olika tillgänglighetsmått. För att kunna effektivt följa upp på SLA och upptäcka brister hos tjänster och utrustningar krävs en effektiv algoritm för att söka bland data och beräkna tillgänglighetsmåtten utifrån dessa. Detta examensarbete har undersökt, utifrån tillhandahållen testdata, huruvida detta kan åstadkommas med hjälp av datastrukturen intervallträd. För att undersöka detta byggdes en prototyp i form av en API-applikation som söker och filtrerar bland hundratusentals larmdata och beräknar tillgänglighet utifrån olika mått såsom bland annat procentuell upptid, medeltid till fel och medeltid till återhämtning. En andra enklare prototyp baserat på SQL och databasen MariaDB utvecklades även för att jämföra lösningens prestanda. De tester som utförts visar på att den intervallträdsbaserade prototypen kan användas för problemområdet samt att den även är en effektiv och skalbar lösning. Resultaten av testerna visar att, till priset av en överkomlig uppstartstid, den intervallträdsbaserade prototypen är ungefär två till fyrtio gånger snabbare vid sökning. Två gånger snabbare vid fallet av en singel sökträff och fyrtio gånger snabbare vid tiotusentals sökträffar. Detta examensarbete gjordes i samarbete med Trafikverket.

Nyckelord: Intervallträd, driftsäkerhet, tillgänglighet, självbalanserande binära sökträd, AVL-träd, röd-svart träd

Abstract

The Swedish transportation systems infrastructure requires many different technical devices of various types to work properly, e.g. sensors, cameras, monitors and routers. It is the Swedish Transport Administration who is responsible for these and to ensure that the devices uphold a good standard there is a *Service Level Agreement* (SLA) that defines requirements for operational reliability. The requirements vary depending on the service and equipment and to be measurable these requirements use different accessibility measures. In order to effectively monitor SLAs and detect deficiencies, an efficient algorithm is needed to search the data and calculate the availability metrics. This thesis has investigated, based on provided test data, whether this can be achieved using the interval tree data structure. To investigate this, a prototype was built in the form of an API-application that searches and filters among hundreds of thousands of alarms and calculates availability based on various metrics such as, among others, uptime in percent, mean time to failure and mean time to recovery. A second simpler prototype based on SQL and the MariaDB database was also developed to compare the performance of the solution. The tests performed show that the prototype based on the interval tree can be used as a solution for this problem and that it also is an efficient and scalable solution. The results of the tests show that, at the cost of an affordable start-up time, the interval-tree-based prototype is about two to forty times faster at searching than the SQL-based prototype. Two times faster in the case of a single search hit and forty times faster in the case of tens of thousands of search hits. This thesis was done in collaboration with the Swedish Transport Administration.

Keywords: Interval tree, operational reliability, availability, self-balancing binary search tree, AVL tree, red-black tree

Förord

Detta examensarbete har berört och fördjupar sig inom flera olika kunskapsområden. Detta har skapat utmaningar och därmed även möjligheter för utveckling hos mig. Trots att arbetet utförts under rådande pandemi och därför skett på distans har jag endast möts av entusiasm och stöd av de berörda av examensarbetet inför de utmaningar som uppkommit.

Jag vill ge ett stort tack till Trafikverket och Daniel Isacson för möjligheten att utföra detta examensarbete i samarbete med dem. Jag vill också tacka alla som bidragit och hjälpt till under arbetets gång, särskilt mina handledare Roger Henriksson vid Lunds universitet och Johan Englund på Trafikverket för deras stöd.

Marcus Sjökvist

Innehållsförteckning

Inledning **11**
1.1 Bakgrund 11
1.2 Syfte 12
1.3 Målformulering 13
1.4 Problemformulering 13
1.5 Motivering av examensarbete 13
1.6 Avgränsningar 14

Teknisk bakgrund **15**
2.1 Intervall 15
2.2 Självbalanserande binära sökträd 16
2.2.1 AVL-träd 17
2.2.2 Röd-svart träd 19
2.2.3 Intervallträd 21
2.3 Tillgänglighet 24
2.4 Datarepresentation 25
2.5 Node.js 25
2.4 Express.js 26
2.5 Node-interval-tree 26

Metod **27**
3.1 Planering 27
3.2 Förstudie 28
3.3 Programmering 28
3.4 Källkritik 29

Analys **31**
4.1 Val av datastruktur 31
4.2 Process för trädets uppbyggnad 31
4.3 Datauppdelning 32
4.4 Filtrering 33
4.5 Tillgänglighetsberäkning 33
4.6 Alternativ lösning 35

Resultat **37**
5.1 Slutgiltigt system 37
5.1.1 Intervallträdsprototypen 38
5.1.2 Alternativ prototyp 39
5.1.2 Tillgänglighetsberäkning 39
5.2 Testfall 41
5.2.1 Sökning med olika parametrar 42
5.2.2 Tidskonsumtionens förhållande till antal sökträffar 43
5.2.3 Uppstart 45

Slutsats **47**
6.1 Övergripande slutsats 47
6.2 Svar på problemformuleringar 47
6.3 Reflektion över etiska aspekter 49
6.3 Framtida utvecklingsmöjligheter 49

Terminologi **51**

Källförteckning **53**

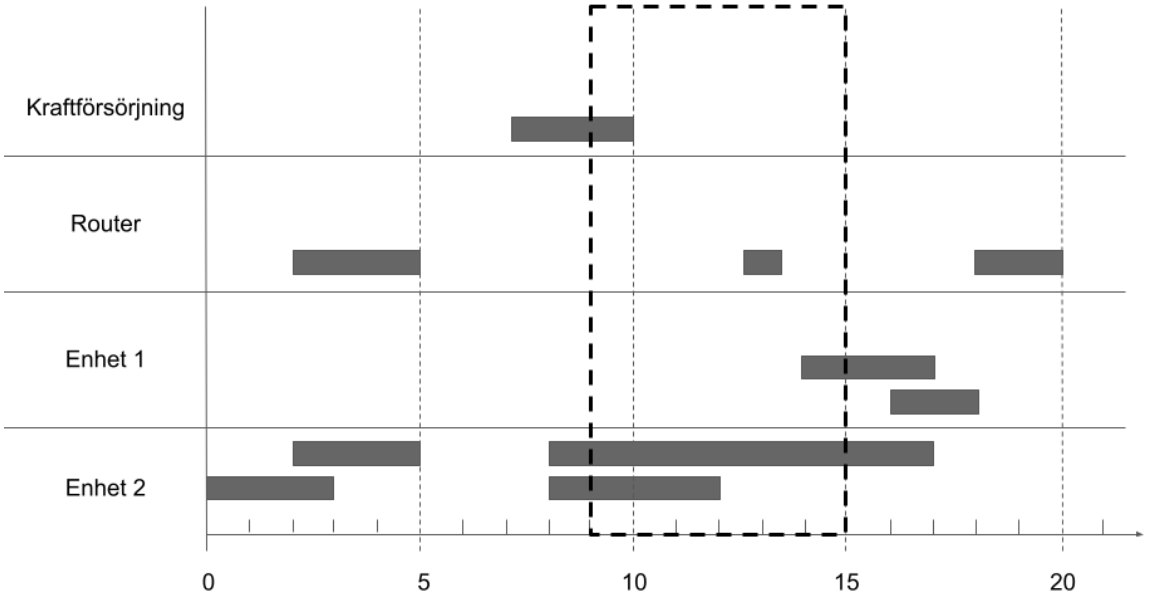
1. Inledning

Här beskrivs beskrivs problemområdet, dess bakgrund och vad examensarbetet ämnar att åstadkomma och varför. Först beskrivs bakgrunden, följt av syfte och målsättning. Därefter motiveras examensarbetet och slutgligen beskrivs de avgränsningar vilka agerat utgångspunkt.

1.1 Bakgrund

Examensarbetet utförs i samarbete med Trafikverket som är en statlig myndighet som ansvarar över långsiktig infrastrukturplanering av det svenska transportsystemet. Detta transportsystem omfattar vägtrafik, järnvägstrafik, sjöfart och luftfart. Trafikverkets fokus ligger på ett smidigare, miljövänligare och säkrare sätt att transportera och resa oavsett var i landet man bor för alla som använder det svenska transportnätet.

Det svenska transportsystemets infrastruktur kräver många olika tekniska enheter av diverse olika typer såsom exempelvis sensorer, kameror, digitala skärmar och routrar för att fungera. Ansvaret för både infrastrukturen och de tekniska enheterna ligger hos Trafikverket. Som en del i att säkerställa att de lösningar som finns implementerade håller en god nivå har Trafikverket olika krav på tillgänglighet. Dessa krav är i form av ett *Service Level Agreement* (SLA) som omfattar olika servicenivåer. Servicenivåerna ska garantera att en viss nivå av tillgänglighet och incidenthanteringsberedskap uppehålls beroende på tjänst. Ett sådant krav kan vara exempelvis 99,99% upptid på ett årsbasis. En tjänst i detta sammanhang består av en eller flera olika ihopkopplade utrustningar som tillsammans skapar en funktionalitet. Ett exempel på en tjänst kan vara en övervakningsfunktion. För att denna tjänst ska vara tillgänglig krävs att flera enheter fungerar som de ska. Till exempel krävs att övervakningskameran, internetuppkopplingen som skickar videon (routern och porten) och att kraftförsörjningen till dessa alla är fungerande för att även tjänsten i sig ska fungera. För att avgöra upptid på tjänsten krävs att alla fyra enheter tas i beaktning. Somliga tjänster innehar redundans vilket innebär en eller flera parallellkopplingar där exempelvis två routrar kan kopplas till en slutenhet. Ett exempel på en generisk tjänst ses i figur 1 där en tjänst beroende av fyra enheter visas tillsammans med eventuella nertider för individuella enheter.



Figur 1. Exempel på tjänst med flera enheter. X-axeln visar tid. Rektangel med streckad linje visar exempel på en tidsram [9, 15] att söka efter genomskärande intervall för att sammanställa upptid. Larm som nertid visas som grå horisontella rektanglar.

Att en tjänst är otillgänglig kan bero på många olika orsaker, planerade eller oplanerade, men varje gång en tjänst blir otillgänglig skapas ett larm av ett övervakningssystem. Larmen lagras som ett intervall över när tjänsten blev otillgänglig till när den åter blev tillgänglig tillsammans med diverse data om enhet, tjänst och kopplingar till andra enheter. Dessa larm är därav oumbärliga i arbetet med att göra uppföljningar på SLA. Över tid inträffar det dock miljontals larm vilket kräver en effektiv algoritm och datastruktur för att kunna analysera dem utifrån en tillgänglighetsaspekt. Övervakningssystemet som levererar den data över larm som används i detta examensarbete har en uppdateringsfrekvens på en minut och detta ska tas i beaktning i examensarbetet i den möjliga mån.

Detta examensarbetet använder sig av intervallträd som utgångspunkt för att presentera en prototyp som söker bland larm och sammanställer tillgänglighetsdata över procentuell upptid, nertid, *MTTR* (medeltid till återhämtning) och *MTBF* (medeltid till fel). Intervallträdet är en algoritm och datastruktur som är till för att effektivt söka och filtrera bland intervall, vilket är en modifiering av ett binärt sökträd.

1.2 Syfte

Syftet med examensarbetet är ta fram en prototyp för back-end-funktionalitet genom användandet av intervallträdet för att kunna söka och filtrera på tjänster och uppkopplade IT-enheters tillgänglighet genom larmdata. Resultatet ska underlätta för uppföljning och övervakning av SLA-servicenivåer för tillgänglighet av IT-lösningar.

1.3 Målformulering

Examensarbetets mål är att ta fram en lämplig datastruktur och algoritm där intervallträdsstrukturen används som utgångspunkt. Denna ska implementeras i form av en fungerande prototyp i lämpligt programmeringsspråk för att stödja funktionaliteterna för insättning och sökning. Sökningen ska kunna filtreras på tidsintervall, enhet och tjänst där hänsyn ska tas till att fler sökfunktionaliteter kan komma att implementeras i framtiden. Prototypen ska kunna erhålla tillgänglighetsmåtten procentuell upptid, *MTTR* och *MTBF*. Lösningen ska vara optimerad för att passa Trafikverkets behov i form av enkelhet av underhåll, vidareutveckling samt effektivitet.

1.4 Problemformulering

Det problem som examensarbetet försöker lösa kan delas upp i följande fyra delproblem:

1. Är intervallträdsstrukturen och algoritmen lämplig för ändamålet?
    a. Är ett intervallträd baserat på ett AVL-träd eller ett röd-svart träd att föredra?
    b. Om inte, finns någon annan datastruktur och algoritm?
2. Hur bör en lösning för att kommunicera med databasen se ut?
3. Hur bör tillgängligheten beräknas?
4. Är slutresultatet en effektiv lösning för datafiltrering av tidsintervall?

1.5 Motivering av examensarbete

En viktig del i Trafikverkets uppdrag är att säkerställa ett tillförlitligt nationellt transportsystem. Därför finns det olika krav på tillgänglighet inom driftsäkerhet. För att kunna uppfylla dessa krav krävs ett effektivt sätt att analysera hur väl tjänsterna håller den tänkta standarden vad gäller upptid och tillgänglighet. Det är därför av stort intresse att undersöka hur de data som finns tillgängligt kan användas för detta syfte. Om examensarbetet utmynnar i en effektiv prototyp kan denna förhoppningsvis vidareutvecklas för att bidra till att uppföljningsarbetet kring driftsäkerheten hos transportsystemens IT-system blir effektivare.

Utöver nyttan hos Trafikverket så blir hantering och lagring av stora mängder data enbart mer och mer relevant i samhället då alltmer utrustning kopplas upp mot internet. Det är därför av allmänt akademiskt intresse att undersöka olika sätt att strukturera data i ett praktiskt sammanhang.

1.6 Avgränsningar

De avgränsningar som gjorts i arbetet är framtagna genom dialog med Trafikverket. Arbetet kommer inte hantera något grafisk gränssnitt. Ingen live-data kommer användas i arbetet, utan enbart äldre exempeldata, men ska utvecklas med en uppdateringsfrekvens av en minut i åtanke. Sökning efter larm och sammanställning av dem begränsas till att kunna filtreras ut med tjänst, utrustning eller bägge och i åtanke. MariaDB används som DBMS då denna används i störst utsträckning av de utvecklare som arbetar närmast problemområdet hos Trafikverket. Valet av DBMS begränsades även utifrån kriterierna att den skulle vara kostnadsfri och ha en stor spridning och användning bland utvecklare i stort. Examensarbetet ska utvecklas i Javascript med hjälp av ramverket Node.js även detta på grund av att detta används i stor utsträckning bland de mest insatta inom problemområdet hos kunden.

2. Teknisk bakgrund

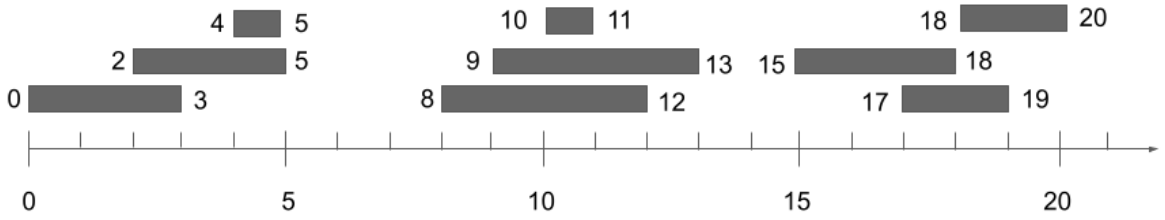
I detta kapitel ges den information om teknik och teori som behövs för att ta del av examensarbetet till fullo. Först introduceras intervall och dess relevanta egenskaper följt av självbalanserande binära sökträd. AVL och det röd-svarta trädet beskrivs på en grundläggande nivå för att därefter gå in mer i detalj på intervallträd. Då borttagning ej är relevant i träden under examensarbetet kommer detta endast nämnas ytligt för AVL-trädet samt det röd-svarta trädet. Därefter beskrivs relevanta driftsäkerhetstermer och definitioner. Sedan beskrivs den testdata som tillhandahållits och slutligen finns en beskrivning av Node.js och Express.js.

2.1 Intervall

Intervall består av ett storleksordnat par av reella värden och kan vara av följande typer:

- Slutna intervall: $[t_1, t_2] = \{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$
- Halvöppna intervall: $[t_1, t_2[= \{t \in \mathbb{R} : t_1 \leq t < t_2\}$ eller $]t_1, t_2] = \{t \in \mathbb{R} : t_1 < t \leq t_2\}$
- Öppna intervall: $]t_1, t_2[= \{t \in \mathbb{R} : t_1 < t < t_2\}$

Den data som används i detta examensarbete är enbart bestående av slutna intervall och därmed kommer dessa att vara i fokus. I figur 2 ses en exempelsamling på intervall där somliga intervall överlappar varandra.

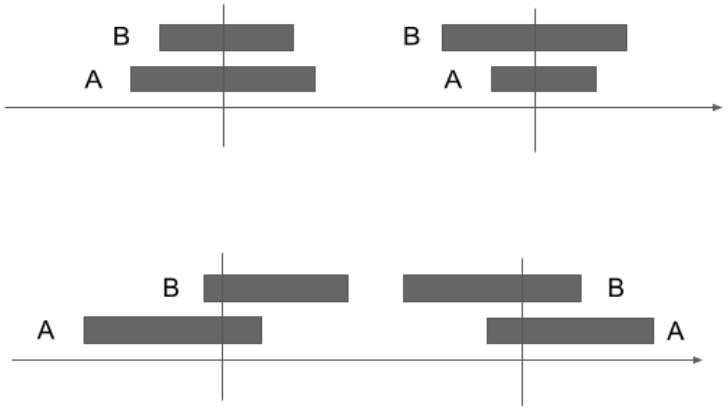


Figur 2. Exempel på en intervallsamling.

Ett intervall har alltid en lägre ändpunkt t_1 samt en övre ändpunkt t_2. I examensarbetet lagras intervallen som en del av dataobjekt. Ett intervall *i* är bestående av två attribut: *i.low* vilket motsvarar $t_1$och *i.high* vilken motsvarar t_2.

Givet att vi har två intervall *A och B* kommer enbart ett av följande fall vara sant enligt intervallets trikotomi [1]:

1. *A* och *B* överlappar där överlappet består av något av fyra fall: A omsluter B, B omsluter A, A överlappar B eller B överlappar A. (*A.low <= B.high & B.low <= A.high*).



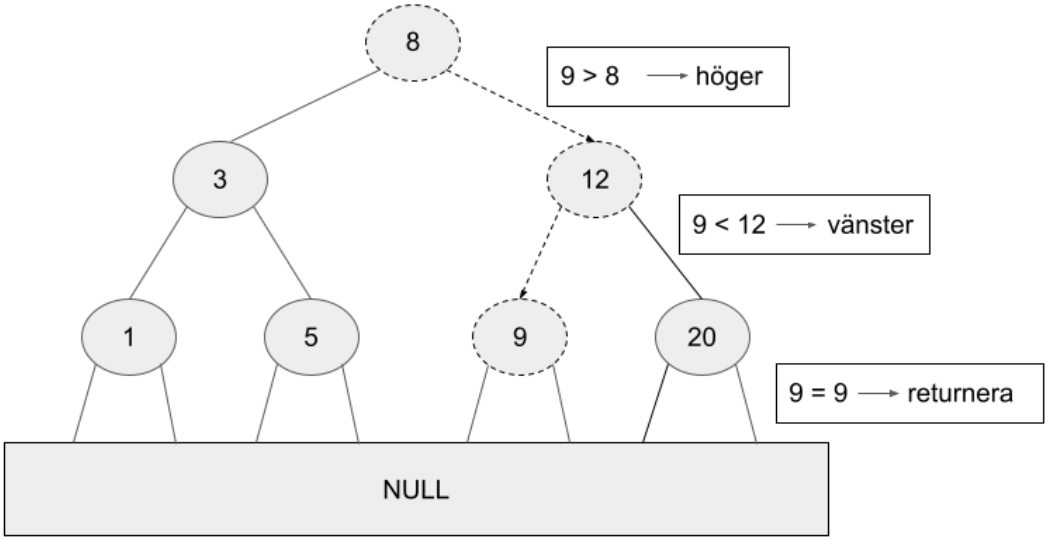
2. *A* är till vänster om *B* (*A.high* < *B.low*).

3. *A* är till höger om *B* (*A.low* > *B.high*).

2.2 Självbalanserande binära sökträd

Ett binärt sökträd är en trädstruktur som stödjer minst tre olika operationer: sökning, insättning och borttagning. Varje nod i trädet får endast ha upp till två barn, ett vänsterbarn och ett högerbarn. Givet en rotnod så måste alla noder i dess vänstra delträd vara mindre och alla noder i höger delträd måste vara större än värdet på rotnoden. Detta ger att en inorder-sökning ger oss alla noder i storleksordning från minst till störst. Ett fullt binärt träd kan som mest innehålla $2^h - 1$ antal noder, där h är trädets höjd [2]. Vid ett fullt och balanserat, det vill säga att antalet noder i höger och vänster delträd är så lika som möjligt, träd medför detta att vid sökning efter en nod kan man efter varje besökt nod halvera det kvarvarande antalet möjliga kandidater. Detta medför också att trädets höjd och tiden för de olika operationerna blir proportionella gentemot varandra genom att de bägge blir O(log n), där n är antalet noder i trädet. Om ett sådant träd inte hålls jämnt försämras dock de egenskaperna. Ett träd där samtliga noder är ett högerbarn till dess förälder resulterar i en tidskonsumtion på O(n) för sökning och insättning vilket är detsamma som en naiv sökning där alla noder jämförs en efter en. I och med att det är så viktigt prestandamässigt att balansera trädet för att minimera dess höjd i trädet har det tagits fram flera olika träd som balanserar sig självt efter insättning och borttagning. Detta åstadkoms med hjälp av specifika regler och trädrotationer. Dessa kallas för självbalanserande binära sökträd.

En algoritm för att söka efter en nod i ett binärt sökträd sker ofta rekursivt och ser ut som följande. Algoritmen börjar med roten och jämför. Om noden som jämförs är null eller stämmer överens med sökvärdet returneras detta och algoritmen är färdig. Om noden är null finns ingen nod i trädet vilken stämmer överens med sökvärdet. Stämmer dessa fall inte in så jämförs noden med det sökta värdet. Är noden mindre kallas sökfunktionen på nytt men med höger barn som nya rotnoden och om noden är större görs samma sak men med vänster barn. Detta repeteras tills null eller det sökta värdet hittats. I figur 3 visualiseras en traversering av ett träd vid sökning efter värdet 9. Sökningen börjar i roten och går sedan ner till höger då 9 är större än nodens värde 8. Därefter jämförs 9 mot 12 och eftersom 9 är lägre går sökningen till vänster. Nästa nod har samma värde som sökvärdet och returneras därför. Om ingen nod hade påträffats vars värde överensstämmer med sökvärdet returneras null.

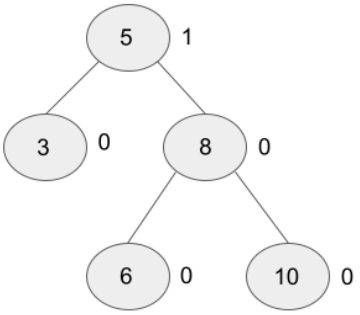


Figur 3. Sökning efter värdet 9 i ett binärt sökträd. Streckad linje visar sökvägen.

2.2.1 AVL-träd

AVL-trädet introducerades år 1962 och är döpt efter dess två skapare Adelson-Velskii och Landis [3]. Det är ett självbalanserande binärt sökträd. Idéen med denna trädstruktur är att hålla trädet balanserat genom att hålla koll på höjden från var nod till löv och automatiskt justera om behov uppstår vid insättning och borttagning. Varje nod har därför extra information i form av en höjdbalans. Denna balans räknas ut som $h_R - h_L$, där h_R och h_L motsvarar höjden i höger respektive vänster delträd. Om nodens delträd har samma höjd är höjdbalansen lika med 0. I figur 4 ses ett träd där rotnodens högra delträd är en nivå högre vilket ger en höjdbalans på +1 i roten. För att trädet ska hållas jämnt tillåts endast balansen vara $-1 \leq (h_R - h_L) \leq 1$. Om höjdbalansen hamnar utanför detta spann vid insättning eller borttagning av en nod måste en ombalansering ske. I och med att höjdbalansen tillåts vara mellan 1 och -1 har studier visat att ett värstafallsscenario ger en 1,44 gånger högre höjd

än ett fullt binärträd [2]. Det är dock mycket sällsynt att detta sker och genomsnittet är mycket nära ett fullt binärträd.

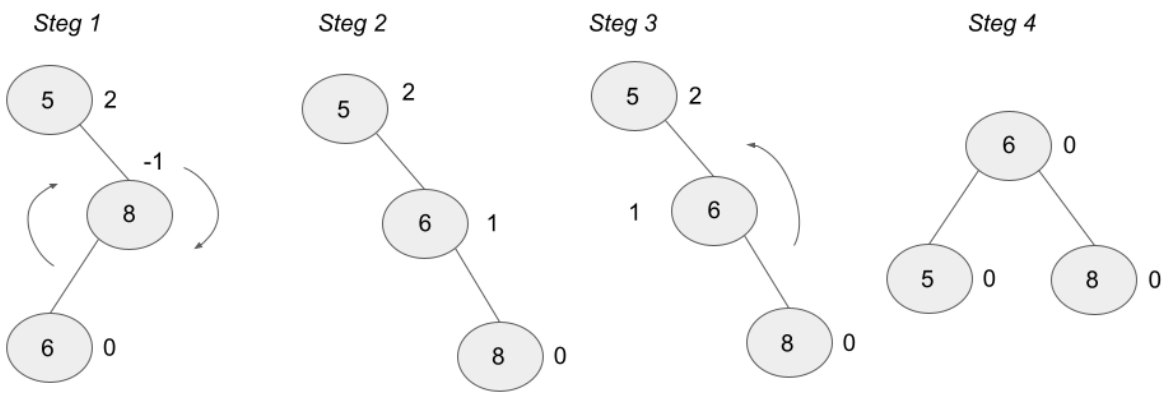


Figur 4. Ett balanserat AVL-träd där höger delträd är högre än vänster.

Om höjdbalansen överstiger 1 på grund av insättning, borttagning eller rotation så är trädet högertungt och behöver roteras till vänster. Om höjdbalansen i stället är mindre än -1 är trädet vänstertungt och behöver roteras till höger. I och med att noderna sätts in en och en kommer vi inte få ett fall där höjdskillnaden i roten är 2 men alla andra noder är 0. En nod kommer enbart ha följande fyra fall:

1. Vänster-vänster (Roten har -2 och vänster barn har -1)
2. Höger-höger (Roten har +2 och höger barn har +1)
3. Vänster-höger (Roten har -2 och höger barn har +1)
4. Höger-vänster (Roten har +2 och vänster barn har -1)

Fall 1 kräver endast en rotation till höger omkring roten och fall 2 åt motsatt riktning. Fall 3 och 4 kräver ytterligare rotation. Där behövs först en rotation runt barnet och sedan en i motsatt riktning runt föräldern. I figur 5 ses ett steg för steg exempel på rotation vid fall 4 (höger-vänster). Först måste trädet roteras till höger runt barnet så att trädet blir ett höger-höger-träd i steg 2. Därefter kan en vänster rotation utföras för att rätta upp trädet.



Figur 5. Balansering av ett höger-vänster AVL-träd.

En sökning för AVL-trädet sker på samma sätt som det generella sättet för binära sökträd. Ett exempel kan ses i figur 3.

En insättning i ett AVL träd går till genom att hitta positionen där noden ska insättas genom en vanlig sökning av ett binärt sökträd såsom i figur 3. När insättningsplatsen är funnen återvänder algoritmen rekursivt och för med sig information om att höjden har ökat i detta delträd. Om höjdbalansen i någon nod är utanför giltigt spann på -1 till 1 utförs rotationer enligt tidigare nämnda rotationsfall och höjden uppdateras i noderna till följd. Detta upprepas tills roten är nådd och rekursionen är fullbordad.

En borttagning i ett AVL-träd sker med fördel rekursivt så att information om att delträdet minskat i höjd kan skickas tillbaka rekursivt. Därefter måste trädet roteras om balansen överträder spannet -1 till 1 i varje nod vilken rekursivt traverseras. I och med att en rotering kan minska höjden på delträdet kan därmed antalet roteringar vara lika med antalet besökta noder i värsta fall.

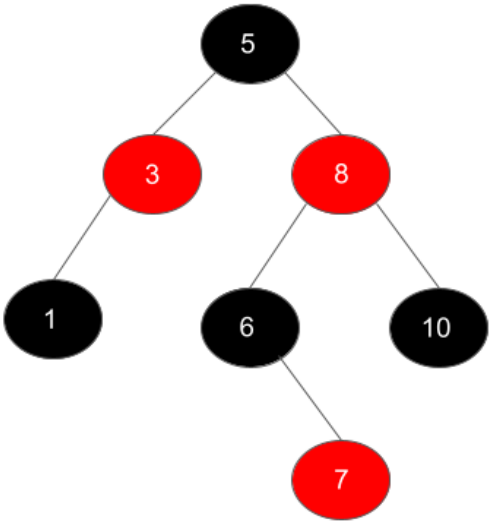
2.2.2 Röd-svart träd

Det röd-svarta trädet har sin grund i det symmetriska binära B-trädet [4] vilket vidareutvecklades till det röd-svarta trädet av L. Guibas och R. Sedgewick [5]. Ett röd-svart träd, se figur 6, är ett balanserat binärt träd som använder sig av en extra bit information i varje nod som indikerar om noden är svart eller röd. Denna färg används i syfte att effektivt kunna balansera trädet. För att balansera sig självt använder det röd-svarta trädet två metoder: färgsättning och rotering. Fördelen med detta träd är att en uppdatering av trädet och följande sortering kan ske under en och samma traversering av trädet. Den högsta höjden ett röd-svart träd kan ha är $2\ log\ n\ +\ 2$ vilket är detsamma som O(log n) [2].

Det röd-svarta trädet har 4 grundregler för att balansera sig självt:

1. En nod är antingen svart eller röd.
2. I samtliga vägar från roten till ett löv finns alltid samma antal svarta noder.
3. En röd nod får aldrig ha ett rött barn.
4. Roten är alltid svart.

Vanligtvis ej betraktat som en av huvudreglerna för det röd-svarta trädet, men värt att poängtera att en null-nod betraktas som svart.



Figur 6. Ett röd-svart träd som följer samtliga grundregler.

En sökning för det röd-svarta trädet sker på samma sätt som det generella sättet för binära sökträd. Ett exempel kan ses i figur 3.

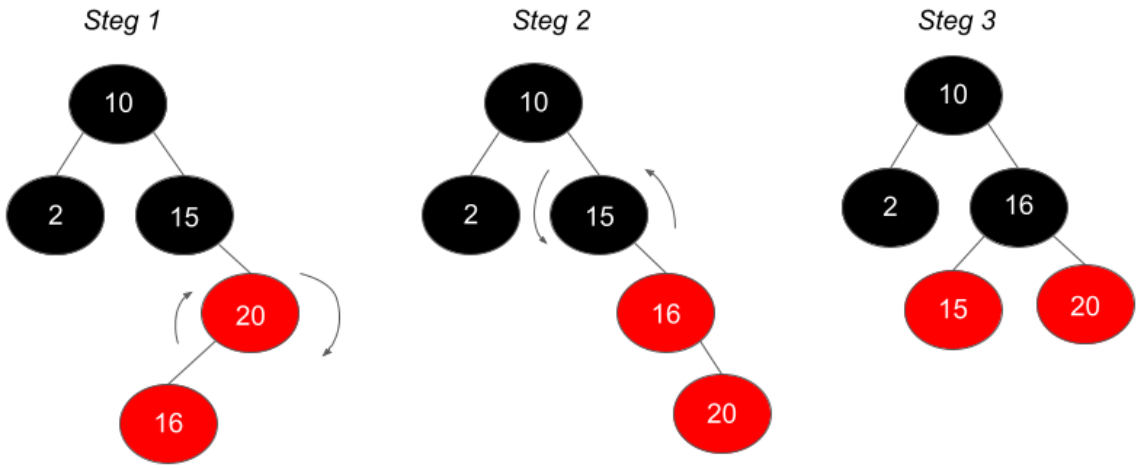
En insättning går till på så vis att vi söker efter korrekt plats med hjälp av en normal sökning för ett binärt sökträd tills en null-nod hittas. Där placeras insättningsnoden in. Om noden sätts in i ett tomt träd blir denna nod roten och färgas därmed svart i enlighet med regel 4. Vid insättning i ett träd som inte är tomt kan en svart nod innebära att huvudregel 2 bryts och justeringar kommer behövas, medan om en röd nod kan sättas in utan att bryta grundregel 3 är insättningen klar. Därför är alltid insättningsnoden röd till en början. Om insättningsnodens förälder är röd bryts regel 3 och hänsyn måste visas till flera olika fall. Om insättningsnodens förälders syskon är röd kan föräldern och dess syskon färgas svart och farföräldrern färgas röd. Det vill säga, det sker en färgförskjutning till dess farförälder. Trädet kontrolleras nu igen utifrån farförälderns perspektiv. Men om insättningsnodens förälders syskon är svart istället för röd krävs trädrotation för att balansera. Här finns 4 olika fall vilka är analoga med de rotationsfall som AVL-trädet innehar.

1. Vänster-vänster (aktuell nod är vänsterbarn liksom dess förälder)
2. Höger-höger (aktuell nod är högerbarn liksom dess förälder)
3. Vänster-höger (aktuell nod är högerbarn men föräldern är vänsterbarn)
4. Höger-vänster (aktuell nod är vänsterbarn men föräldern är högerbarn)

Lösningen för fall 1 och två löses genom en trädrotation runt farföräldern i motsatt riktning. Vid fallet vänster-vänster sker trädrotationen således till höger och ett höger-höger roteras till vänster. Därefter byter för förälder och farföräldern färg för att bibehålla färgbalansen. För fall 3 och 4 krävs ytterligare en rotation runt föräldern och på så sätt omvandla situationen till fall 1 eller 2. Därefter görs samma procedur som för dessa fall. Detta repeteras utifrån farförälderns perspektiv tills roten är nådd.

I figur 7 ses ett exempel på ett höger-vänster träd. Direkt innan steg 1 har en nod med värdet 16 satts in. I och med att detta fall är detsamma som fall 4, höger-vänster, visar första steget

att en höger-rotation runt föräldern krävs. I steg 2 ses en höger-höger situation och därför görs en rotation kring farföräldern. Därefter byter farföräldern och föräldern färg med varandra. Jämför med AVL-trädets rotation i figur 5.



Figur 7. Höger-vänster rotation i ett röd-svart träd.

När det kommer till borttagning sker sökningen efter noden på samma sätt som vanligtvis görs för binära sökträd. Huvudregeln för borttagning i röd-svarta träd är att en nod bara tas bort om det är ett löv eller bara har ett barn. Stämmer inte detta överens med den nod som ska tas bort måste denna bytas ut mot sin "inorder"-föregångare innan borttagning. Röda noder kan tas bort utan bekymmer då detta inte kan göra att trädet bryter mot någon av grundreglerna. Om noden som tas bort är svart men har ett rött barn så tar barnet dess plats och färgas röd. Om ett svart löv tas bort bryts nu grundregel 2 och flera olika fall behövs tas hänsyn till. Då borttagning ej är nödvändigt för en fungerande prototyp tas inte de fallen upp här i examensarbetet.

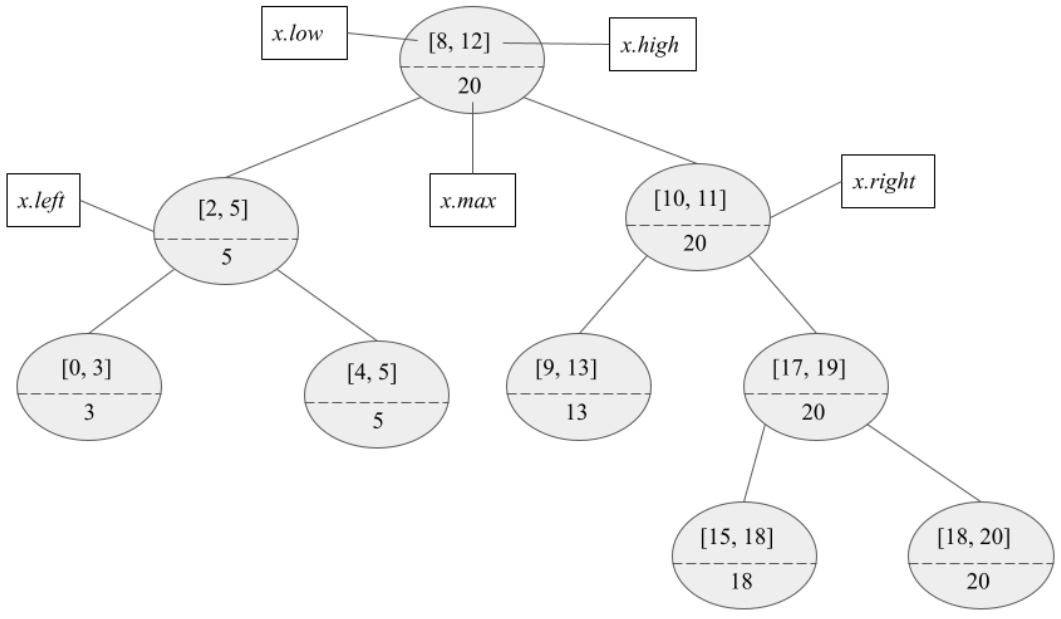
2.2.3 Intervallträd

Intervallträdet är ett binärt sökträd som är till för att lagra och effektivt kunna söka bland intervall. Det beskrevs för första gången av Edelsbrunner år 1980 och McCreight år 1981 enligt [1] där de presenterar intervallträdet som en lösning på problemet med att hitta alla intervall i en datasamling vilka överlappar en viss punkt och samtidigt även kunna hitta alla punkter inom ett intervall. Denna trädstrukturen är inte den enda som specialiserar sig på intervall. Ett exempel på ett sådant är segmentträdet [6], men denna fokuserar mer på att effektivt finna vilka intervall i en mängd som genomskär en viss punkt till skillnad på intervallträdet som fokuserar på vilka intervall i en mängd som överlappar ett visst intervall. Ett intervallträd är konstruerat för att stödja tre grundoperationer: insättning, borttagning och sökning.

Intervallträdet är i grunden en modifiering av binära sökträd. Därför kan trädet se lite olika ut beroende på vilken av alla dessa trädstrukturer som används till grund. I detta arbetet kommer

två av de mest populära självbalanserande binära träden undersökas som potentiell grund: AVL-trädet och det röd-svarta trädet. Ett alternativ till dessa beskrivs i [7] och baseras på användandet av arrayer i varje nod. Ett intervallträd baserat på någon av dessa två träd innehar även samma grundegenskaper som dem. Den maximala höjden för intervallträdet baserat på AVL-trädet kan ej överstiga $1,44 * log(1 + n)$ medan det trädet som baserar sig på det röd-svarta inte balanseras lika strikt och kan som högst ha höjden $2 * log(1 + n)$, där n är antal noder. De grundläggande operationerna sökning, insättning och borttagning sker under $O(log\, n)$ tid. För att skapa intervallträdet krävs $O(n * log\, n)$ tid då insättning sker en gång per nod. En sökning där alla matchande intervall ska returneras kräver $O(log\, n + k)$ där k är antalet returnerade intervall [8].

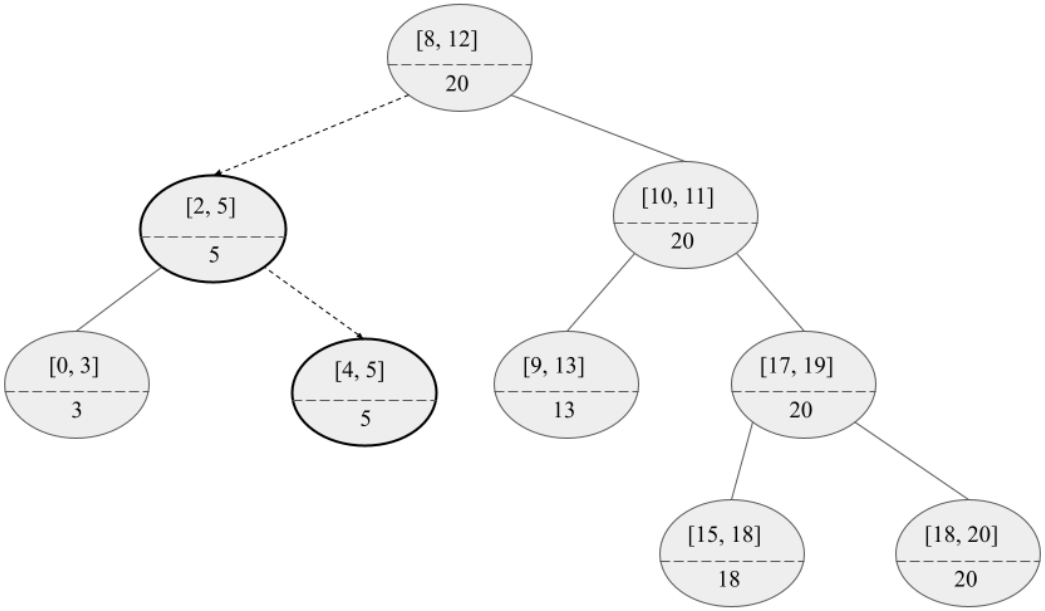
I figur 8 kan ett exempel på en generiskt intervallträd ses. Trädet i figuren saknar den extra information som ett med grunden i AVL-trädet eller det röd-svarta trädet skulle inneha. I figuren kallas rotnoden för *x*. Utöver den grundläggande informationen om vilka noder som är dess vänster och höger barn, *x.left* och *x.right*, så innehåller intervallträdets noder även ytterligare information om dess intervall i form av dess lägre ändpunkt t_1 och dess övre ändpunkt t_2, dessa är detsamma som *x.low* respektive *x.high* i figuren. Nodernas nyckel är dess lägre ändpunkt. Det medför att en inorder-traversering av trädet returnerar noderna i ordning efter när intervallen börjar. Noderna innehåller även *x.max*, vilket är det största värdet något av intervallen i delträdets löv kan inneha.



Figur 8. Intervallträd av intervallsamlingen i figur 2 där NIL-noderna är utelämnade.

En sökning i ett intervallträd baserat på antingen ett AVL-träd eller ett röd-svart träd ter sig likadant då dessas egenskaper främst påverkar balansering. Det som är annorlunda än för ett standard binärt sökträd är att hänsyn ges även till delträdets max-värde i en sökning och

behöver uppdateras för varje ändring i trädet. En sökning i ett intervallträd tar som sagt O(*log n + k*) tid och går till på samma sätt som för vanliga binära sökträd, men i och med att intervallträdets noder innehåller den extra informationen om det största värde som kan hittas i delträdet kan ytterligare delträd avfärdas utan behovet av att traversera dem. Detta sker genom att kolla om lägsta värdet på intervallet vi söker efter är större än det vänstra delträdets maxvärde. Vidare så kan inte det högra delträdet innehålla något överlappande värde om dess lägre ändpunkt är större än sökintervallets övre ändpunkt.



Figur 9. Sökning i intervallträd efter intervallet [4, 7] i intervallträdet i figur 8. Sökväg för att finna korrekt plats är markerad med streckade pilar och sökningen returnerar de fetmarkerade noderna (intervall [2,5] och [4,5]).

I figur 9 ses ett exempel på en sökning efter intervall vilka genomskär sökintervallet [4, 7] som kommer kallas *i*. Sökningen börjar i roten och den genomskär ej *i.* Då den lägre ändpunkten i roten (8) är större än *i.high (7)* finns inget intervall i höger delträd som överlappar. Sökningen fortsätter därav till vänster. Denna nod med intervall [2, 5] överlappar. Dess vänstra barn har ett maxvärde vilket är mindre än *i.low* vilket medför att inga noder kan överlappa. Höger barn däremot har varken en lägre ändpunkt som är större än *i.high* blir detta den nya sökningsutgångspunkten. Även denna nod överlappar. Då noden är ett löv är sökningen klar och de två överlappande intervallen returneras.

För att finna platsen i trädstrukturen där insättningsnoden ska sättas in används samma algoritm för sökningsoperationen. Skillnaden är att korrekt plats hittats och sökningen är klar då en null-nod påträffats. Givet insättningsnoden *i* börjar en insättning med jämförelse med roten, *x*. Om *x.low* är mindre än den i insättningsnoden går vi till dess högra nod. Om *x.low* är större än *i.low* går vi till vänster. Detta repeteras med nästkommande nod tills en null-nod har påträffats. Därefter kontrolleras trädets balans utefter de regler som kommer utav grunden i

detta fallet antingen AVL-trädet eller det röd-svarta trädet. Det som skiljer balansering åt från de andra träden är att även max-värdet i varje nod måste uppdateras rekursivt. Tarjan [9] har bevisat att för ett balanserat binärt sökträd baserat på Bayers träd [4] kan en trädrotation ske under tiden O(1) och det är på detta sätt en insättning kan ske under tiden O(log n), samma som sökning efter en nod.

En borttagning börjar med ett intervall att söka efter, vilken kommer framöver benämnas *i*. Därefter börjar operationen i roten *x* och jämför denna nods intervall med intervallet *i.* Om dessa skär varandra har noden att tas bort hittats. Om dessa inte genomskär varandra kollas nodens barn. Om vänster barn är null (*x.left = null*) görs *x.right* till nya *x* att jämföra med. Om *x.left* inte är null kontrolleras om maxvärdet i slutnoden i vänster delträd är mindre än *i.low* vilket motsvarar den lägre ändpunkten t_1 i det eftersökta intervallet. Är detta maxvärde värde lägre än *i.low* är det heller ingen efterföljande nod som kan genomskära *i* då egenskaperna för överlappande intervall, beskrivet i kapitel 2.1, motbevisar detta . I detta fall blir *x.right* nya *x.* Om inget av tidigare steg stämmer blir x.left nya x. Detta repeteras tills det genomskärande intervall hittas eller en null-nod. En null-nod innebär att det inte finns en nod som genomskär *i.*

För en nod som genomskär intervallet *i* kommer ett av tre fall stämma.

1. Noden har inga barn och kan därmed tas bort utan några speciella åtgärder.
2. Noden har ett barn. Barnet tar då nodens plats i datastrukturen.
3. Noden har två barn. Noden skrivs över av dess nästkommande nod i "inorder"-sökning. Den nästkommande noden tas bort.

I samtliga dessa fall kommer den borttagna noden ur hierarkin kommer alltid vara en nod med inget eller ett barn vilket gör att anpassning endast behövs för dessa två fall. Precis som i fallet med insättning måste trädrotationer utföras om trädet inte förhåller sig till dess balanseringsregler beroende på vilka dessa är, AVL eller röd-svarta trädets regler. Precis som vid insättning måste även nodernas maxvärde uppdateras om noderna roteras.

2.3 Tillgänglighet

Tillgänglighet inom driftsäkerhet kan mätas på många olika sätt. I detta examensarbete används tre av de definitioner och formler som nämns i [10] och som används av avdelningen för kundtillförlitlighetsteknik (Customer Reliability Engineering Team) hos Google. Dessa tre definitioner är upptid, *MTBF* och *MTTR*. Upptiden definieras som $\frac{Upptid}{(Upptid + Nertid)}$ och ger en procentsats. MTBF, som är medeltiden mellan inträffanden av fel, definieras som $\frac{Upptid}{Antal\ fel}$ och MTTR, som är medeltiden det tar att återhämta sig från fel, definieras enligt $\frac{Nertid}{Antal\ fel}$. Utöver de statistiska värdena i sig kan MTBF och MTTR kombineras för att ge en förväntad framtida nertid genom formeln $\frac{Total\ period}{MTBF} * MTTR$.

2.4 Datarepresentation

Tabell 1. Datavyn.

Kolumnnamn	**Datatyp**
förbindelse_id	varchar(10) Primärnyckel
tjänst	varchar(255)
utrustning	varchar(255)
starttid	timestamp
sluttid	timestamp
tillstånd	int(11)
meddelande	text
system_id	varchar(255)

Datat som används i examensarbetet är en statisk mängd exempeldata på 286 026 stycken tupler, men i produktionsmiljö finns flertalet miljoner tupler, där start och sluttid är utspridda över de fyra första månaderna av år 2021. Trots att exempeldatat är statiskt ska dock hänsyn tas till att i produktionsmiljö kommer datat vara dynamiskt och successivt adderas till den totala datamängden. I tabell 1 ges en överblick av det data som används och dess datatyper. Datat i examensarbetet är en samlad data-vy från flera olika databaskällor som till en början lagras i en MariaDB-databas. Kolumnerna i datavyn är *förbindelse_id*, *tjänst*, *utrustning*, *starttid*, *sluttid*, *tillstånd*, *meddelande*, *system_id*. Kolumnen *förbindelse_id* är ett unikt id kopplat till en tjänst. *Tjänst* är en sträng som beskriver vilken sorts tjänst det är som en utrustning tillhör, exempelvis en vägkamera. *Utrustning* beskriver vilken specifik utrustning det är som larmet sker på. *Start- och sluttid* motsvarar den lägre respektive övre ändpunkten i intervallen vilka intervallträdet byggs på. *Tillstånd* är en siffra, 1-5, vilken motsvarar vilken talar om vilken status larmet har. *Meddelande* är strängdata som beskriver vad för sorts larm det är. *System_id* är ett unikt id kopplat till det ordinarie övervakningssystemet för att kunna särskilja vilka larm som är dubbletter då larmen kommer från flera olika system. Fler kolumner kan tillkomma i framtiden om behov av fler filtreringsmöjligheter och om fler attribut hos datat krävs.

2.5 Node.js

Node.js [11] är enligt [12][13] ett open-source asynkront eventbaserat javascript runtime, vilken möjliggör användandet av javascript i backend. Node.js använder sig av samma motor

som webbläsaren Google Chrome, nämligen V8 Javascript-motorn och det är detta innebär att javascript används. Node.js består enbart av en tråd som hanterar alla processer en efter en och använder sig istället av asynkrona IO-operationer vilka pausar och återupptar processer efter prioritet. Detta medför att den overhead som kommer utav att skapa och byta trådar kan undvikas. Node package manager är Node.js egna pakethanterare och i dess register finns över en miljon open source paket.

2.4 Express.js

Express.js [14] är ett minimalistiskt webbapplikationsramverk för Node.js med öppen källkod. Express.js erbjuder grundläggande webbapplikationsfunktioner och en stack av HTTP-verktygsmetoder och mellanprogramvara för att bland annat enkelt skapa webb-API:er.

2.5 Node-interval-tree

Node-interval-tree [15] är ett NPM-paket (Node package manager) för Node.js, innehållandes en implementation av ett intervallträd. Intervallträdet i paketet är specifikt ett intervallträd baserat på AVL-trädet och dess regler för självbalansering. Implementationen kan hantera de tre operationerna: sökning, insättning och borttagning. Node-interval-tree är utvecklat av M. Žarković och P. Krafft.

3. Metod

Detta avsnitt beskriver den metod med vilken arbetet utförts. Först beskrivs den översiktliga planeringen, följt av en fördjupning i förstudierna och programmeringen. Slutligen motiveras de källor som använts till examensarbetets teoretiska grund.

3.1 Planering

Första steget i examensarbetet var att utforska problemet i sig. En dialog fördes med Trafikverket för att kartlägga problemområdet. När problemområdet var kartlagt behövdes avvägningar göras kring vilka problem i problemområdet som är relevanta att lösa i detta examensarbete. I samband med att avgränsningar av problemområdet växer fram via dialog med Trafikverket kunde en bred informationssökning via internet om tillgänglighetsanalyser, algoritmer för intervall och hantering av överlappande intervall göras, där intervallträdet påträffades och sågs som en potentiell lösning. I detta skede fanns tillräckligt med information för att planera och disponera arbetet. Tidsuppskattningar gjorde över de olika delarna och ett Gantt-schema kunde skapas för examensarbetet.

I och med att förhoppningen var att eventuellt kunna använda den slutgiltiga prototypen, eller delar utav den, som examensarbetet resulterar i som utgångspunkt för ett skarpt system fördes också en dialog om vilka eventuella programmeringsspråk och DBMS som främst bör användas. Ur denna dialog växte många av de avgränsningar som examensarbetet har fram. Som programmeringsspråk var Javascript i samband med användandet av ramverket Node.js att föredra då denna kombination används i störst utsträckning av de utvecklarna som är kopplade till problemområdet. Den DBMS, MariaDB, som användes i examensarbetet valdes även den på grund av att den används i större utsträckning av programmerarna i anslutning till problemområdet.

För att kunna avgöra om intervallträd kan vara av nytta inom problemområdet, och i så fall hur, krävs en djup förståelse för dess bakomliggande teori. Detta åstadkoms genom förstudier. Dessa behandlade utöver intervallträdet även tillgänglighetsmått och beräkningar inom driftsäkerhet samt ramverket node.js och hur denna kunde användas. Efter förstudierna var nästa del i examensarbetet att utifrån de teoretiska kunskaperna från förstudierna skapa en skiss på en prototyp och att därefter implementera denna i kod för att ha en fungerande prototyp. Efter att prototypen implementerats planerades även en alternativ enklare prototyp för att jämföra intervallträdsprototypen med denna för att lätt kunna göra jämförelser angående lösningens effektivitet. För att jämföra de två olika lösningarna prestandamässigt utfördes till sist en rad olika tester.

3.2 Förstudie

Efter den ursprungliga dialogen med kunden där problemen som behövde lösas kartlades och analyserades så verkade intervallträdet intuitivt vara en algoritm som passade problemområdet. För att kunna säkerställa att intervallträdet var av nytta och hur det i så fall skulle gagna i just denna kontext behövdes litteraturstudier göras och dess resultat kan ses i den tekniska bakgrunden. För att få en ytlig första uppfattning om intervallträdets grundläggande egenskaper nyttjades internetsökningar där information hämtades från diverse webbsidor och universitetsföreläsningar. Dessa var enbart menade att vara allmänorienterande och inte för att användas till examensarbetets teoretiska grund. Inledningsvis stöttes det på problematik i och med att olika källor inte stämde överens om hur intervallträdets algoritm fungerar och ingen källa påträffades som beskrev olika sorters intervallträd. Detta visade dock att intervallträdet inte var enbart av en strikt sort utan att det kunde byggas på olika sätt med olika grunder. För att kunna jämföra de mest frekvent förekommande varianterna av intervallträd undersöktes därefter olika publicerade litterära verk om datoralgoritmer och strukturer samt vetenskapliga artiklar vilka beskriver intervallträd och de träd som ligger till grund för dem. Dessa verk och artiklar gav en djupare teoretisk förståelse samt bidrog med den teoretiska grunden för examensarbetet. Ett alternativ till de två sorterna av intervallträd vars grund baseras i AVL-träd respektive röd-svarta träd hittades i [7]. Denna avfärdades som potentiell lösning inom detta problemområde då efter en ytlig analys kunde konstateras att denna variation var inriktad på att hitta intervall vilka överlappar en punkt och inte ett intervall. I förstudierna undersöktes också olika sätt att definiera tillgänglighet där främst internetsökning begagnades efter olika institutioners och examensarbetens definitioner, där dock Googles definitioner [10] valdes på grund av hur väl de stämde överens med de mått Trafikverket efterfrågade.

3.3 Programmering

Då Node.js föredrogs av kunden i detta examensarbetet behövdes en programmeringsmiljö där Node.js kunde installeras och användas. I examensarbetets användes Visual Studio Code som integrerad utvecklingsmiljö och där ett Node.js examensarbetet skapades. Allt utvecklande skedde lokalt på en dator. Då implementationer av intervallträdet redan finns under öppen licens valdes en av dessa istället för att skriva om samma kod på nytt. I och med att programmet är tänkt att vara ett API som körs på en server rekommenderades Express.js som möjligt verktyg för detta ändamål och därmed också valdes för implementation. En lokal server skapades på datorn och därefter programmerades avlyssning på en specifik port med hjälp av Express.js. För att kommunicera med API:et programmerades stöd för HTTP-metoden GET. I produktionsmiljö skulle denna omvandlas till att stödja metoden POST för att undvika att data lagras i webbläsarens cacheminne och historik. Detta skulle öka säkerheten hos applikationen. Då den alternativa enklare lösningen behöver tolka GET-metoden annorlunda än intervallträdslösningen skrevs huvudfilen om. Denna lösning behöver omvandla GET-parametrarna till en SQL-sats för att hämta intervallen direkt från databasen när en förfrågan till API:et inkommer. I kapitel 2.1 beskrivs intervallens trikonomi och överlappande intervalls egenskaper vilket användes för denna SQL-sats. Efter att

intervallen hämtats sker sammanfogning av intervall och beräkning av tillgänglighet på samma sätt för bägge prototyper.

3.4 Källkritik

De källor som använts i detta arbete, med några få undantag, är alla vetenskapliga artiklar publicerade i vetenskapliga journaler eller böcker. Detta innebär att andra akademiker inom samma vetenskapliga område eller granskare vid det utgivande förlaget granskar texterna innan publicering. Källorna [1][2][6][7] är alla böcker om algoritmer och datastrukturer i allmänhet och utgivna av olika bokförlag. Källorna [3][4][5][8][9] är alla vetenskapliga artiklar publicerade i vetenskapliga journaler. De källor som inte är vetenskapliga böcker eller artiklar är huvudsakligen informationen om ramverk [11][12][13][14] och det npm-paket som använts för intervallträdet [15]. I dessa fall har informationen hämtats direkt från ramverkets eller npm-paketets webbplats. Det sista undantaget är en artikel publicerad på Googles hemsida [10] av deras egna ingenjörer och beskriver deras egna definitioner på tillgänglighetsmått för driftsäkerhet.

4. Analys

I detta kapitel presenteras analysen. Den är uppdelad i sex olika delar och beskriver alla de viktiga valen om hur systemet skulle konstrueras och de problem som uppstått under arbetets gång.

4.1 Val av datastruktur

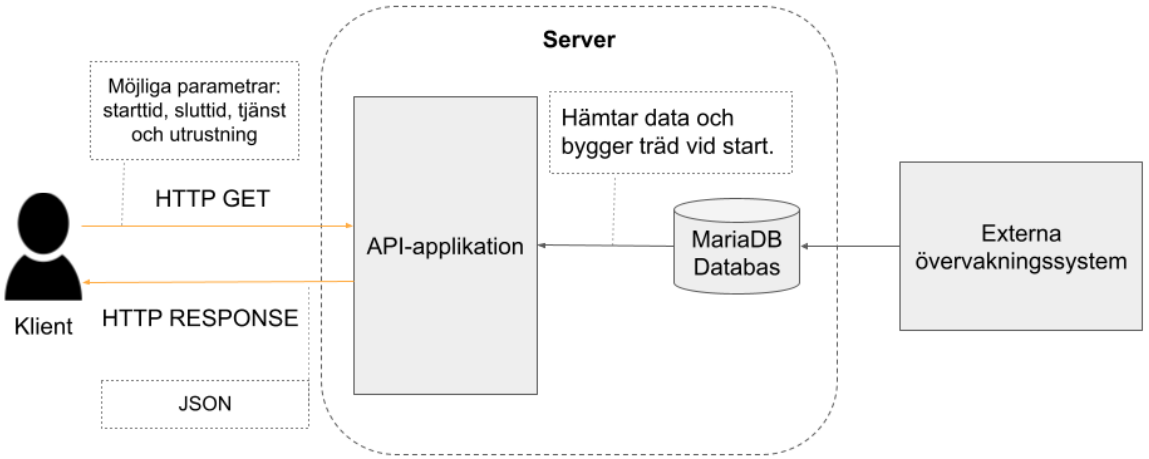
I och med förstudierna kring de bakomliggande teorierna för de bägge varianterna av intervallträdet föll valet på AVL-trädet. Förstudierna visade att då de båda träden har sin grund i det binära sökträdet är de identiska när det kommer till att söka efter intervall samt när man finner den plats i trädet där insättning eller borttagning ska ske vid dessa operationer. Den skillnaden som finns är logiken kring hur dessa träd ska balansera sig självt efter en insättning eller borttagning. Det röd-svarta trädet är något mindre strikt än AVL-trädet vilket leder till att trädets höjd blir i regel ojämnare än hos AVL-trädet. Detta leder till att det röd-svarta trädet kan som mest ha höjden $2 * log(1 + n)$ medan AVL-trädet endast kan ha som mest höjden $1,44 * log(1 + n)$, där n är antalet noder i trädet.

Den princip som gör binära sökträd effektiva är att desto mer balanserad trädet är desto närmre kommer det till att filtrera ut hälften av alla kvarvarande noder vid varje jämförelse. Är trädet obalanserat filtreras färre noder ut och fler jämförelser måste utföras. Detta gör att AVL-trädet bör teoretiskt ha en snabbare sökning medan det röd-svarta bör hantera ändringar i strukturen och eventuella påföljande ombalanseringar något snabbare. Då arbetet eftersträvar att om möjligt kunna hantera sökningar i realtid är sökningen naturligt det viktigaste. Och därmed valdes AVL-trädet som det skäligare alternativet. Då det skulle inneburit ett stort tidssvinn att själv skriva koden för att hantera intervallträdet från grunden användes ett npm-paket [15] för att implementera trädstrukturen. Detta träd ger stöd för att inkludera extra data i varje nod vilket var vad som behövdes i detta examensarbete.

4.2 Process för trädets uppbyggnad

Skillnaden i tidskomplexitet mellan att bygga hela trädet och att söka i intervallträdet är O(n log n) kontra O(log n + k) där n är antalet noder och k är antalet sökträffar. Detta medför att kostnaden i tid är mycket högre vid uppbyggnad kontra vid en sökning och att desto fler noder desto mer uttalad blir skillnad i tidskonsumtion. Detta då varje insättning av en nod kräver en sökning för att hitta rätt plats i strukturen. Med denna egenskap i åtanke bör det undvikas att trädet behöver byggas upp på nytt varje gång för att minska på tidskonsumtionen för algoritmen. För att undvika detta behövs att trädet finns lagrat på något sätt. Exempelvis kan trädstrukturen lagras i en databas som kan effektivt hantera binära trädstrukturer och ombalansering av dessa. I och med examensarbetets avgränsning till att främst bruka MariaDB som DBMS avfärdades denna lösning. MariaDB är en relationsbaserad DBMS och medan det går att använda olika tekniker för att lagra hierarkiska trädstrukturer i sådana

databaser såsom till exempel adjacency lists eller nested sets, vilka beskrivs bland annat av J. Celko [16], är dessa i regel komplexa och ökar i komplexitet när modifikationer så som självbalansering och dess logik ska appliceras. Den enklare lösningen är att ha algoritmen verksam på en server som ett API för att kunna skicka förfrågan till denna och på så sätt hålla trädstrukturerna i applikationens minne. Figur 10 visar den tänkta applikationsarkitekturen, där klienten med hjälp av HTTPS-funktioner skickar förfrågan till API-et. Svaret som kommer från applikationen är i JSON-format. Den nuvarande databasen MariaDB används enbart som backup för att lagra strukturen om servern eller API:et behöver omstartas och då spelar prestandan för att hämta ut datat från databasen mindre roll i och med att servern sällan behöver startas om. Det är främst vid underhåll omstart krävs och detta sker väldigt sällan under tider då en sökning behöver göras.



Figur 10. Planerad applikationsarkitektur.

4.3 Datauppdelning

Då valet om hur trädet lagras gjorts kommer följdfrågan om hur dessa träd ska uppdelas, om alls. Då höjden på trädet är logaritmisk med basen två och sökningens tidskomplexitet är i relation med höjden går dessa hand i hand. Den totala mängden data att hantera uppgår i miljontals, men höjderna mellan ett binärt träd med en miljon noder är, avrundat till en decimal, 19,9 och ett med femtio miljoner noder är 25,6. Då höjden inte blir väsentligt mycket högre när miljontals noder adderas och höjningen av trädhöjden per adderad nod avtar med antalet noder kan det eventuellt fungera med en enda trädstruktur för alla larm. Men noderna ska även innehålla övrig information såsom tjänst och utrustning där även möjligheten att addera fler filtreringsmöjligheter i framtiden så som platsdata och detta behövs tas hänsyn till. I samråd med handledare på Trafikverket insågs det att dela upp larmdata i olika träd beroende på tillhörande tjänst vore mest rimligt då filtreringar baserade på tjänst ansågs vara den som skulle användas mest frekvent. Därav bör denna uppdelning vara av störst nytta. Den stora nackdelen med att dela upp träden baserat på tjänst är att för att söka efter tillgänglighet på en viss sorts utrustning måste samtliga träd genomsökas vilket inte är en effektiv lösning. Detta kommer utav det faktum att tjänst och utrustning har en många-till-många-relation. En

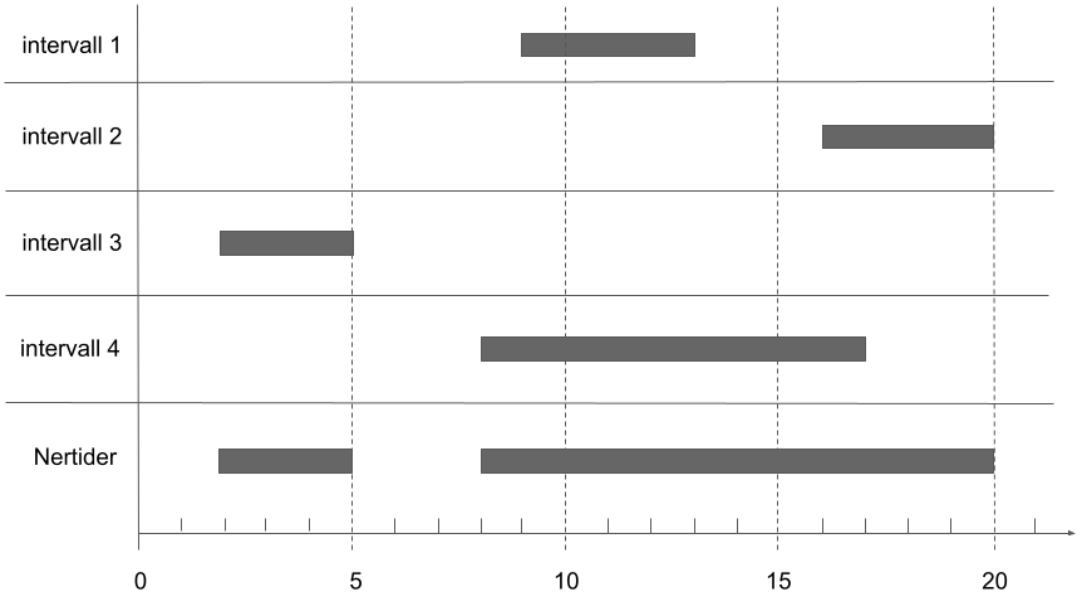
effektivare lösning skulle vara att låta trädet innehålla en lista över vilka utrustningar som tjänsten innehåller. Om inte tjänsten innehåller utrustningen som efterfrågas behöver trädet ej traverseras.

4.4 Filtrering

Intervallträdspaketet som användes gav inte stöd för att kunna filtrera sökningar utifrån andra kriterier än tidsintervall. Det enklaste sättet att lösa detta skulle vara att gå igenom alla resultat efter en sökning och filtrera ut dem baserat på vald filtrering. Detta är dock inte en effektiv lösning. En effektivare lösning skulle vara att göra filtreringen i samband med att överlappande intervall hittas. Därför modifierades sökfunktionaliteten av intervallträdspaketet för att ta emot en extra parameter. Då ett intervall som överlappar sökintervallet påträffas i sökningen jämförs även denna parameter. Om inte parametern också överensstämmer med noden läggs denna inte till i den array som returneras av funktionen. Då träden valts att delas upp utefter tjänst görs denna filtrering per automatik då trädet väljs ut för traversering och den filtreringen som återstår att implementeras är utrustning.

4.5 Tillgänglighetsberäkning

För att kunna räkna ut tjänsters tillgänglighet behövs en algoritm vilken sammanfogar överlappande intervall. Figur 11 visar ett exempel på 4 intervall som ska sammanfogas och det förväntade resultatet visas av raden “Nertider”. Detta åstadkoms genom att använda datastrukturen stack. Stack används här på så sätt att först läggs det första intervallet på stacken. Detta jämförs sedan mot nästkommande intervall enligt egenskaperna hos överlappande intervall i kapitel 2.1. Intervallen A och B överlappar endast om $A.low <= B.high$ och $B.low <= A.high$ är sanna. Om intervallen inte överlappar läggs intervallet på stacken och nästa intervall jämförs nu mot detta intervall istället. Om intervallen istället skulle överlappa tas intervallet av stacken och om det nya intervallets övre ändpunkt är större än hos det nuvarande intervallet uppdateras det nuvarande intervallets övre ändpunkt och placeras tillbaka på stacken. Detta förutsätter att intervallen som itereras för sammanfogning är i kronologisk ordning sorterade efter den lägre ändpunkten. Då intervallträdets sökningsfunktion returnerar noderna och därmed intervallen på detta sättet passar dessa lösningar väl överens. Enda gången hänsyn behöver tas till sorteringen är då flera träd söks igenom och slås ihop.



Figur 11. Fyra olika exempelintervall där överlappande intervall sammansätts. Resultatet av sammansättningen är de två intervallen [2, 5] och [8, 20] i rad "Nertider".

När intervallen är sammanfogade behövs dessa jämföras mot algoritmen för procentuell upptid i kapitel 2.3:

$$\frac{Upptid}{(Upptid + Nertid)}$$

Därför skapades en funktion som tar ett intervall vilket motsvarar total tid och en array av de nertidsintervall där tjänsten varit ur funktion. Nertidsintervallen itereras och adderas för att sedan stoppas in i formeln. Exempelvis skulle exemplet i figur 11 leda till följande procentuell upptid om mätperioden är 0 till 20:

$$Procentuell\ Upptid = \frac{Upptid}{(Upptid + Nertid)} = \frac{5}{(5 + 15)} \approx 25\%$$

Om samma fall används för att även räkna ut MTBF och MTTR enligt definitionerna i kapitel 2.3 blir resultatet:

$$MTBF = \frac{Upptid}{Antal\ fel} = \frac{5}{2} = 2,5\ tidsenheter,$$
$$MTTR = \frac{Nertid}{Antal\ fel} = \frac{15}{2} = 7,5\ tidsenheter.$$

Detta ger att medeltiden mellan inträffande av fel är 2,5 tidsenheter och 7,5 tidsenheter i genomsnitt från att ett fel inträffar till att tjänsten åter är funktionell. Dessa beräkningar görs också för var enskild utrustning för att se om det är en viss utrustning i tjänsten som är felande.

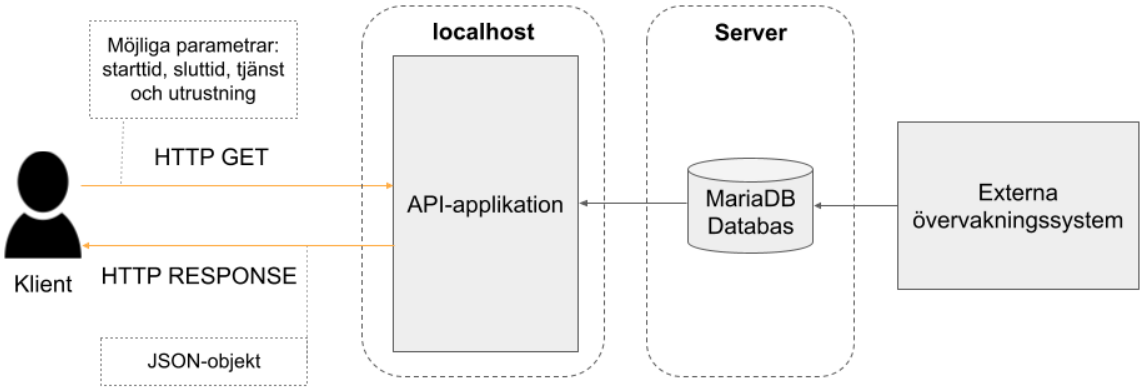
4.6 Alternativ lösning

För att kunna avgöra om lösningen är effektiv krävs en alternativ lösning att jämföra mot. Det enklaste sättet som är snarlikt i applikationsarkitekturen är att på samma sätt som intervallträdslösningen sätta upp ett API på en server som avlyssnar en specifik port efter HTTP-metoden GET beskrivet i figur 10. Skillnaden är att i denna alternativa lösningen hämtas datat direkt från databasen vid förfrågan från klienten med hjälp av en SQL-sats. Autentiseringen och uppkopplingen mot databasen sker i samband med en förfrågan från klienten. SQL-satsen tar hänsyn till samma parametrar som GET-metoden, det vill säga starttid, sluttid, tjänst och utrustning. För att hämta ut alla larm som matchar tidsintervallet begagnas de överlappande intervallens egenskap att intervallen A och B överlappar endast varandra om bägge påståendena $A.low <= B.high$ och $B.low <= A.high$ är korrekta. Data hämtas ut sorterade i kronologisk ordning enligt intervallens lägre ändpunkter. Detta medför att en sortering av intervall aldrig behövs i ett annat skeende som det krävs för intervallträdslösningen om flera träd behöver sökas. Tillgänglighetsuträkningen sker på samma sätt för bägge lösningar och returnera samma resultat. Detta leder till att det till synes inte är någon skillnad alls för klienten mellan de två olika lösningarna förutom eventuella prestandaskillnader.

5. Resultat

I detta kapitel redovisas det slutgiltiga system och de tester som utförts. Först redovisas intervallträdsprototypen, den naiva prototypen och sedan tillgänglighetsberäkningen. Därefter redovisas de olika testen som genomfördes.

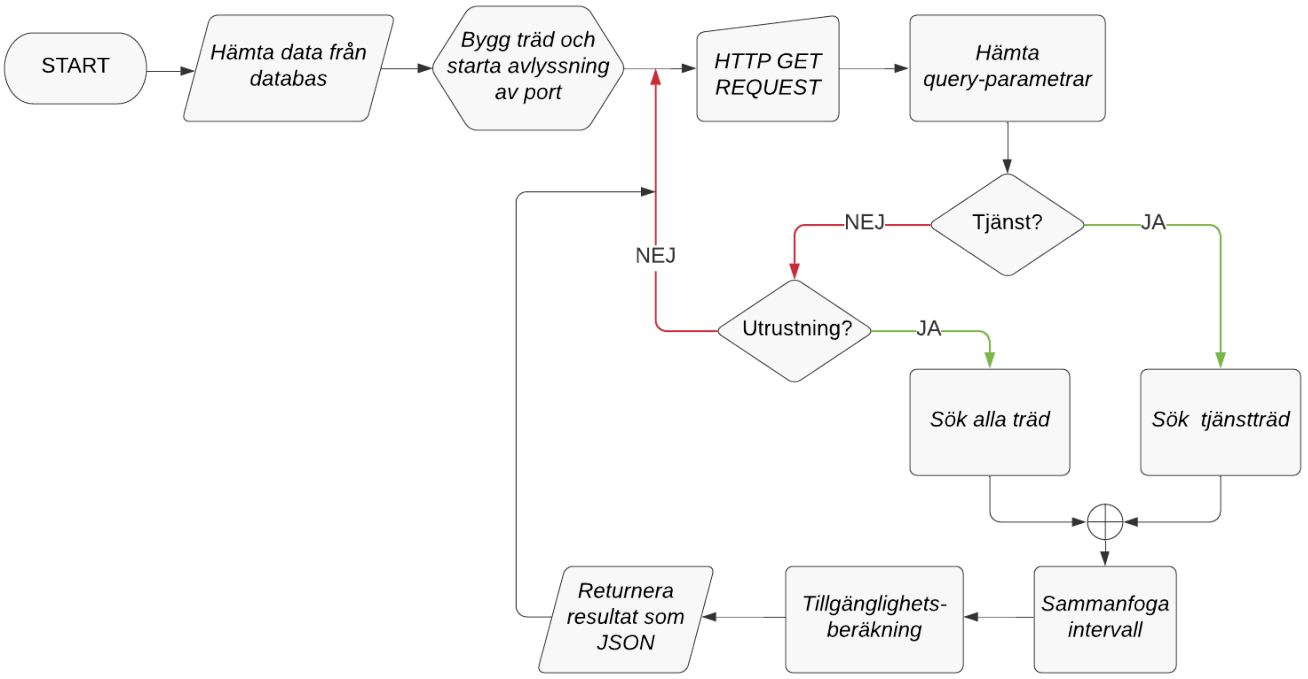
5.1 Slutgiltigt system



Figur 12. Den slutgiltiga applikationsarkitekturen.

Den slutgiltiga prototypen, som kan ses i figur 12, är snarlik den planerade applikationsarkitekturen i figur 10. Den enda skillnaden dem emellan ligger i att applikationen körs på en lokal server. Detta kommer sig av att då inte autentisering och annan säkerhetsfunktionalitet är implementerad i prototypen lämpar det sig inte att ladda upp den på en server där andra klienter kan komma åt den.

5.1.1 Intervallträdsprototypen



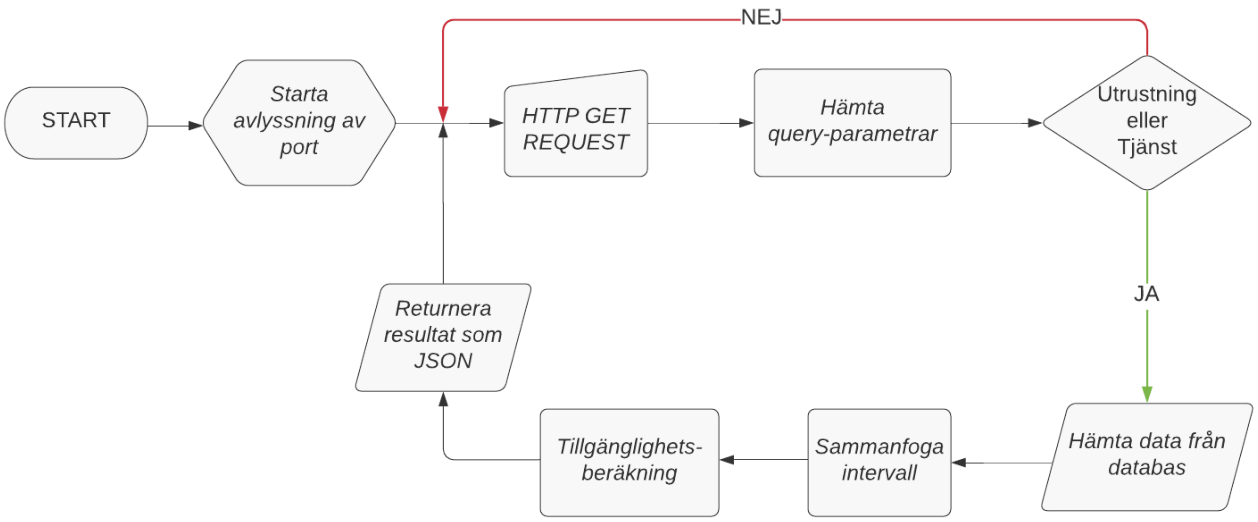
Figur 13. Flowchart för intervallträdsprotypen.

Händelseförloppen och logiken för den intervallträdsbaserade lösningen kan ses i figur 13. För att kunna utföra sökningar behöver applikationen först initieras varvid händelseförloppet börjar. Vid initieringen hämtas data från MariaDB-databasen och använder denna data för att skapa intervallträd. Varje träd består av larm endast från en unik tjänst var och lagras i en array av objekt. Dessa objekt innehåller tjänstens namn och tillhörande träd. Efter att träden byggts upp startas en localhost-server och en avlyssning efter förfrågningar mot en specifik port påbörjas. En sökning går till genom att klienten skickar en HTTP GET-förfrågan till applikationen. Dessa förfrågningar kan skickas på flera olika sätt, men i detta examensarbetes testfall skickades dessa via en webbläsare, se figur 14. När applikationen mottager en förfrågan hämtas parametrarna ut. Först kontrolleras om en sökning efter tjänst angetts. Om ja, söker vi tjänstens träd efter de sökparametrar som hämtats från GET-förfrågan. Om nej, kontrolleras om utrustning angetts och om den inte har angetts returneras tomma resultat och vi återgår till att vänta in en ny förfrågan. Om istället en utrustning angetts kontrolleras en array, kopplat till varje intervallträd, innehållandes information om vilka utrustningar som finns att påträffa i just denna tjänsten. De träd där utrustningen finns sökes därmed efter överlappande larm på denna utrustning. På detta vis behövs inte alla träd genomsökas. När de överlappande intervallen hittats sammanfogas dessa varpå tillgängligheten beräknas. Sedan returneras tillgänglighetsdatan i form av ett JSON-objekt.

```
http://localhost:3000/availability?tjänst=tjänst1&start=1609714800&end=1610233200&utrust
ning =switch1
```

Figur 14. Ett exempel på en URL för att skicka GET-förfrågan.

5.1.2 Alternativ prototyp



Figur 14. Flowchart för den alternativa prototypen.

Den alternativa lösning som framtagits som jämförelseobjekt är en fungerande lösning i sig. Denna prototyp är väldigt snarlik i sitt händelseförlopp, se figur 14, som den huvudsakliga prototypen som beskrivits i 5.1.1. Den enda skillnaden kommer i att vid start behövs ej data hämtas då inga träd behövs byggas på förhand. Istället hämtas data från databasen efter att en förfrågan inkommit till den avlyssnade porten och därefter är händelseförloppet detsamma. Den SQL-query som använts för att hämta ut data från databasen utifrån sökparametrarna kan ses i figur 15. Denna SQL-query utnyttjar de egenskaper för överlappande intervall beskrivet i kapitel 2.1.

```
tjänst = tjänst ? (utrustning ? `förbindelse_id = "${tjänst}" AND` :
        `förbindelse_id = "${tjänst}"`) : ""
utrustning = utrustning ? `utrustning = "${utrustning}"` : ""
let query = `SELECT * FROM \`tabell\` WHERE ${tjänst} ${utrustning}`
  +     ` AND ${starttid} <= UNIX_TIMESTAMP(sluttid) AND`
  +     ` UNIX_TIMESTAMP(starttid) <= ${sluttid}`
  +     ` ORDER BY starttid ASC`
```

Figur 15. Javascript-kod där sökparametrarna infogas i en SQL-query.

5.1.2 Tillgänglighetsberäkning

Då larmen hämtats ut av sökalgoritmerna behöver dessa omvandlas till tillgänglighetsmått. Den lösning som beskrivits i kapitel 4.5 kunde implementeras med javascript-kod, men då datastrukturen *stack* inte finns i varken Javascript eller Node.js behövdes en sådan klass skapas. I den implementerade klassen användes en array för att lagra intervallerna. Alla de tillgänglighetsmått som önskats kunde implementeras så som figur 16 visar.

```
function availability(faultIntervals, period) {
  if (!faultIntervals || faultIntervals.length < 1 || !period) return 100

  let faultTime = 0
  for (let i = 0; i < faultIntervals.length; i++) {
    faultTime += faultIntervals[i].high - faultIntervals[i].low
  }

  let uptimeMs = (period - faultTime)
  let uptime = ((uptimeMs / (uptimeMs + faultTime)) * 100).toFixed(3)
  if (uptime >= 100) uptime = 99.999
  let MTBF = (uptimeMs / faultIntervals.length).toFixed(2)
  let MTTR = (faultTime / faultIntervals.length).toFixed(2)
  return { upptid: uptime, upptidMs: uptimeMs, nertid: faultTime, MTBF:
    MTBF, MTTR: MTTR }
}
```

Figur 16. Funktionen som räknar ut tillgänglighet baserat på nertids-intervallen "faultIntervals".

Ett exempel på den data och de tillgänglighetsmått som returneras i form av ett JSON-objekt kan ses av figur 17. Först i objektet ligger informationen om sökningen och de samtliga larmens gemensamma tillgänglighetsmått. Därefter finns en array som heter "utrustningar". Denna array innehåller samma information som de gemensamma larmen men är uppdelade per utrustning. Varje element i denna array innehåller i sin tur också en array innehållandes de enskilda utrustningarnas nertider nedskalade till endast intervall i unixtid. Samtliga mått på tid är är i millisekunder, men upptid syftar till den procentuella upptiden.

```
{"tjänstid":"Tjänst1",
"utrustning":"",
"sökstart":"2021-01-04 00:00:00",
"sökslut":"2021-02-01 00:00:00",
"antalLarm":41838,
"upptid":99.997,
"upptidMs":2419124523,
"nedtidMs":75477,
"MTBF":"57821.23",
"MTTR":"1.80"
"utrustningar":[
        {"id":"port",
        "intervall":[
                {"low":1610622909,"high":1610622924},
                {"low":1610622943,"high":1610622958},
                ...],
        "availability":{
                "upptid":99.997,
                "upptidMs":2419124666,
                "nertid":75334,
                "MTBF":"57811.56",
                "MTTR":"1.80"}
        },
        {"id":"switch",
        "intervall":[
                {"low":1610673014,"high":1610673044},
                {"low":1611183038,"high":1611183165
        ],
        "availability":{
                "upptid":99.999,
                "upptidMs":2419199843,
                "nertid":157,
                "MTBF":"1209599921.50",
                "MTTR":"78.50"}
        }
]}
```


Figur 17. Exempel på JSON-datat som returneras av API:et.

5.2 Testfall

För att mäta hur prototypen presterar skapades 3 olika tester vardera bestående av olika testfall. Det första testet i 5.2.1 testar olika parametrar för att se hur sökningen presterar i de olika variationerna som är möjliga. Det andra testet i 5.2.2 testar snarlika sökningar men som ger ett spritt antal träffar för att mäta korrelationen mellan antal sökträffar och tidskonsumtion. Det sista testet mäter tiden från att applikationen körs till att den är redo för att ta emot sökningar. I samtliga av dessa tester jämförs intervallträdslösningen mot den naivare SQL-baserade lösningen som hämtar data direkt från databasen. I de två första testerna utförs sökningarna i webbläsaren, i detta fallet Google Chrome, och responsen skrivs ut. Tidsmätningen skedde via Google Chromes debugger där varje mätning görs via hård inläsning och cacheminnet töms innan varje sökning. Dessa mätningar upprepades tre gånger för att ta fram ett genomsnittsresultat. I testet om uppstartstid skedde istället tidsmätningen

med hjälp av Node.js inbyggda *Performance measurement API* och tidskomsumtionen loggades i terminalen.

Om sluttiden ej anges i sökningarna tolkas detta av applikationen som att sluttiden är den sekund då parametrarna inläses då inga larm kan tillkomma efter den tidpunkten och om starttid ej anges tolkas detta som precis ett år innan sluttiden. Detta innebär att en sökning utan tidsparametrar först sätter sluttiden till den tidpunkt som sökningen görs och sedan sätter starttiden till ett år tillbaka från den punkten. Då testdata endast innehåller larm för de fyra första månaderna av år 2021 skulle sökningar utanför dessa månader endast innebär felaktiga tillgänglighetsberäkningar och därmed gjordes inga sökningar utan tidsparametrar i testerna.

5.2.1 Sökning med olika parametrar

Då parametrarna i sökningarna alla är frivilliga, med undantaget att någon av antingen parametern "tjänst" eller "utrustning" måste vara tillhandahållen, är det viktigt att testa hur sökningarna presterar i de olika möjliga variationerna. Dessa testfall ämnar att testa en spridning av olika parameterkombinationer.

Tabell 3. Sökparametrarna för test av sökning med varierande parametrar.

Testfallsparametrar				
Testfall	1	2	3	4
Tjänst	Tjänst 1	Tjänst 1	Tjänst 1	-
Utrustning	-	-	Switch1	Switch2.port
Starttid	2021-01-01 00:00:00	2021-01-01 00:00:00	2021-01-01 00:00:00	2021-01-01 00:00:00
Sluttid	2021-05-01 00:00:00	2021-01-20 00:00:00	2021-01-20 00:00:00	2021-05-01 00:00:00

Testfallsparametrarna som valdes kan ses i tabell 3. Testfall 1 söker efter en specifik tjänst under perioden för testdata som här kallas *Tjänst1* och används då denna hade flest larm av alla tjänster. Detta innebär en sökning genom ett helt intervallträd där alla noder returneras. Testfall 2 är detsamma som testfall 1 men sluttiden är satt till den 20:e januari. Testfall 3 söker på samma tjänst och tidsintervall men fokuserar på en enskild utrustning som här kallas *Switch1*. Fall 4 söker enbart efter en specifik utrustning vilket här är en port tillhörandes en annan switch än i testfall 3. Tabell 4 visar resultatet av dessa testfall.

Tabell 4. Testfallsresultat för sökning med varierande parametrar i millisekunder enligt testfallsparametrarna i tabell 3.

Testfallsresultat				
Testfall	1	2	3	4
Sökträffar	41 842	28 286	1	7
Intervallträd	162	117	107	64
Databas	6 623	4 426	223	210
$\frac{Databas}{Intervallträd}$	40,88	37,83	2,08	3,28

5.2.2 Tidskonsumtionens förhållande till antal sökträffar

Då tidskonsumtionen för sökningar med varierande kombinationer av parametrar kan vara svåra att jämföra gentemot varandra består detta andra test enbart av liknande sökningar men som resulterat i olika mängder träffar. Testet består av fem olika testfall där de bägge prototyperna jämförs.

Tabell 5. Sökparametrarna för test av sökning med varierande antal sökträffar.

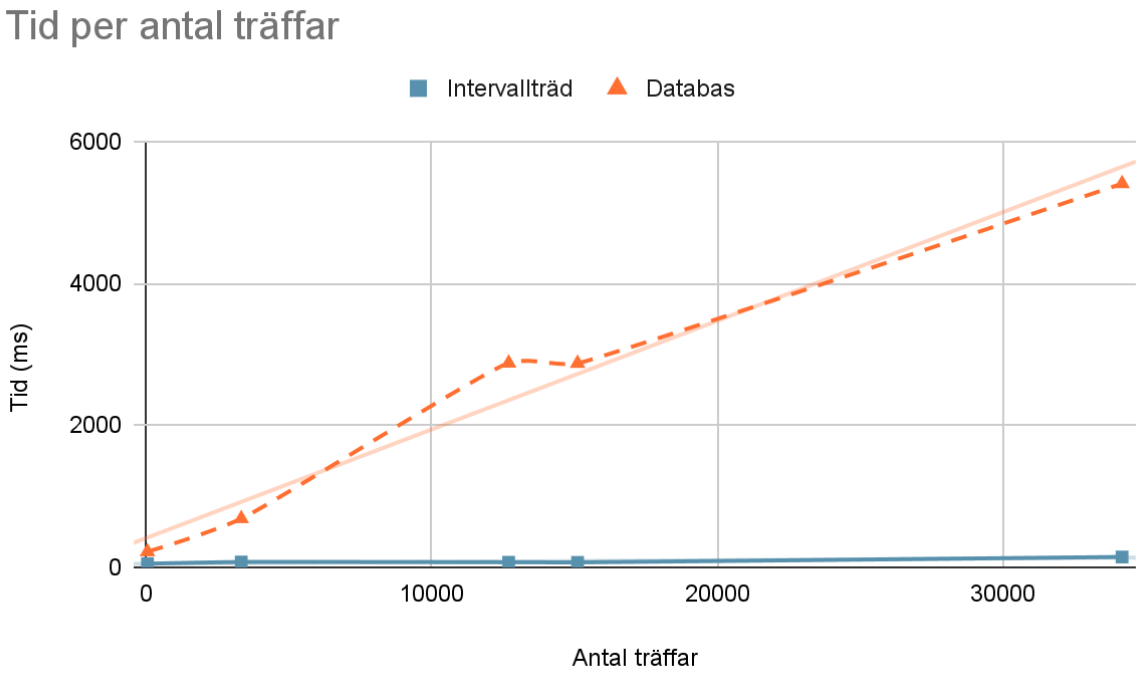
Testfallsparametrar					
Testfall	1	2	3	4	5
Tjänst	Tjänst2	Tjänst3	Tjänst4	Tjänst5	Tjänst1
Utrustning	-	-	-	-	-
Starttid	2021-01-04 00:00:00	2021-01-04 00:00:00	2021-01-04 00:00:00	2021-01-04 00:00:00	2021-01-16 00:00:00
Sluttid	2021-01-10 00:00:00	2021-01-10 00:00:00	2021-01-10 00:00:00	2021-05-01 00:00:00	2021-05-01 00:00:00

Testfallsparametrarna som valdes kan ses i tabell *5*. Testfall 1 till 3 söker efter alla larm under andra veckan år 2021 där vardera sökning söker inom olika tjänster. Testfall 4 söker efter en annan tjänst än de andra testfallen men från fjärde januari till och första maj. Testfall 5 söker efter tjänst 1 (samma tjänst som i testet i 5.2.1) och söker från 16:e januari till och första maj.

Tabell 6. Testfallsresultat för sökning med varierande sökträffar i millisekunder enligt testfallsparametrarna i tabell 5.

Testfallsresultat					
Testfall	1	2	3	4	5
Sökträffar	12 693	15 105	51	3 335	34 161
Intervallträd	72	70	51	75	146
Databas	2 877	2 873	220	686	5 403
$\frac{Databas}{Intervallträd}$	39,96	41,04	4,31	9,15	37,78

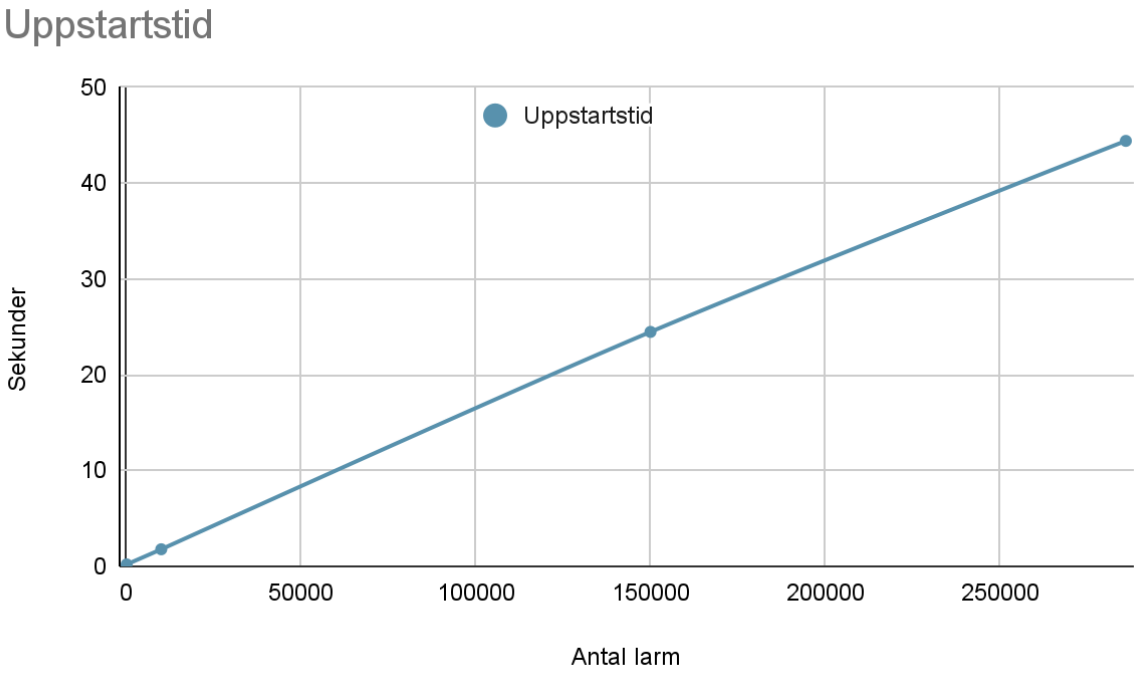
Tabell 6 redovisar resultatet från sökningarna i tabell 5 där tiderna är i millisekunder. Varje mätning upprepades tre gånger och siffrorna i tabellen är genomsnittet av dessa sökningarna. Varje sökning skedde efter hård inläsning och tömning av webbläsarens cacheminne. Graf 1 visar resultaten i tabell 6 grafiskt där även en trend linje visas.



Graf 1. Förhållandet mellan antal sökträffar och tidskonsumtion för de bägge lösningarna. Intervallträdslösningens resultat visualiseras av kontinuerlig linje med kvadratiska datapunkter. Den alternativa lösningens resultat visualiseras av streckad linje med triangulära datapunkter. Bägge linjerna har en trendlinje. Linjen för intervallträdet och dess trendlinje överlappar.

5.2.3 Uppstart

Då intervallträdet behöver hämta data och bygga träden innan sökningar kan ske så tillkommer en uppstartstid. Teoretiskt ska uppbyggnaden av träden ta $O(n * log\, n)$ tid. Fyra mätningar gjordes där samtliga 286 026 testdata inläses. Mätningarna gjordes med hjälp av Node.js inbyggda *Performance measurement API,* istället för att bruka Google Chromes debugger-funktion. För intervallträdsprototypen var den genomsnittliga tidskonsumtionen 44,41 sekunder. För den alternativa lösningen som enbart kompilerar koden och startar avlyssningen på en port var den genomsnittliga tidskonsumtionen 0,0037 sekunder.



Graf 2. Uppstartstiden i sekunder hos intervalltrådslösningen i relation till antal larm som läses in.

Ytterligare mätningar på intervalltrådslösningen gjordes där olika antal testdata inläses och resultat kan ses i graf 2. 4 olika mätningar gjordes där: 286 026, 150 000, 10 000 och 100 larm lästes in där 286 026 var samtlig testdata. Resultat för dessa mätningar var 44,41s; 24,47s; 1,78s och 0,20s respektive.

6. Slutsats

I detta kapitel diskuteras resultatet och problemformuleringarna besvaras utifrån den mån de insikter som framkommit av examensarbetet tillåter. Nästan samtliga frågor har kunnat besvarats och av de efterfrågade funktionaliteterna har i stort sett alla blivit implementerade. Resultatet är en prototyp som uppfyller både syfte och målsättning. Först dras en slutsats om prototypen i sin helhet och sedan diskuteras problemformuleringarna och sist diskuteras framtida utvecklingsmöjligheter.

6.1 Övergripande slutsats

Detta examensarbete har kunnat påvisa att intervallträdet och den framtagna algoritmen för tillgänglighetsberäkning är en möjlig lösning för det problemområde som undersökts. Medan resultaten visar att bägge lösningar fungerar så visar de även att intervallträdssökningen till och med presterar bättre i samtliga tester där den som sämst har presterat ungefär 208% och som bäst ungefär 4 100% av den naiva lösningens effektivitet tidsmässigt. Resultaten visar även på att lösningen är mycket mer skalbar än den alternativa lösningen i och med resultaten som visualiseras i graf 1. Detta resultat visar på att algoritmen även fungerar för större datamängder och är mycket betydelsefullt då i produktionsmiljö ökar datamängden med 50-100 gånger. Nackdelarna med prototypen är uppstartstiden, se graf 2, att lösningen kräver viss vidareutveckling för att användas i produktionsmiljö, se kapitel 6.2, samt att prototypen är mer komplex än den enklare jämförelseprototypen vilket leder till svårare vidareutveckling. Dessa nackdelar kan dock anses vägas upp av det faktum att intervallträdsprototypen presterar så pass mycket bättre i samtliga test av sök och tillgänglighetsfunktionen. Testerna visar att desto fler larm ju effektivare är prototypen jämfört med databaslösningen.

6.2 Svar på problemformuleringar

- *Är intervallträdsstrukturen och algoritmen lämplig för ändamålet?*
    - *Är ett intervallträd baserat på ett AVL-träd eller ett röd-svart träd att föredra?*
    - *Om inte, finns någon annan datastruktur och algoritm?*

I detta examensarbete har en fungerande prototyp kunnat skapas. Den har all den funktionalitet som efterfrågats och dessutom rimliga resultat. Det finns klart för- och nackdelar med intervallträdet som algoritm i det problemområde som undersökts. Den största nackdelen är uppstartstiden när träden byggs upp där den alternativa prototypens tidskonsumtion är minimal i jämförelse. Vi kan se i graf 2 att denna uppstartstid ökar relativt linjärt i takt med antalet larm. I testet där ungefär 40 sekunder behövs för att hämta datat och bygga träden är det någorlunda överkomligt, men då en produktionssättning skulle innebära en ökning av antal larm från cirka 300 000 till tiotals miljoner och tiden att bygga skulle snarare handla om minst en halvtimme räknat på att antalet larm är 50 gånger större i produktionsmiljö. Detta är dock inte ett avgörande problem om trädet tillåts byggas upp och ligga sparat i applikationens minne på servern och om man också underhåller träden. För detta behövs det adderas funktionalitet som uppdaterar träden när nya larm tillkommer och om något larm tas bort i efterhand ska dessa också kunna tas bort utan att behöva bygga om alla

träd från grunden. Testfallsmätningarna gjordes med en ganska långsam nätverksförbindelse utanför Trafikverkets nät. När ett test av applikationen gjordes på plats i ett av Trafikverkets kontor gavs ett mycket bättre resultat. Detta pekar på att om den planerade strukturen i figur 10 efterföljs kommer prestandan vara mycket högre för uppbyggnad av intervallträden då applikation och databas kommer finnas på samma nätverk och server med en snabb nätverksuppkoppling. De främsta fördelarna med intervallträdsprototypen är sökning och skalbarhet. Sökningarna har visats vara mycket effektivare i samtliga testfall, se Tabell 6 och Tabell 4. Graf 1 visar också på att lösningen är mer skalbar för större mängder larm och bör prestera mycket bättre i längden då antalet larm endast kommer att öka över tid.

Efter förstudierna kunde konstateras att de bägge varianterna, AVL och röd-svart, bör inte skilja särskilt mycket när det kommer till prestanda. Sökningarna går till identiskt då båda är binära sökträd. Det som skulle skilja dem åt är balanseringen efter insättning eller borttagning då de följer olika regeluppsättningar. Teoretiskt kan dock konstateras att en bas med AVL-trädet bör prestera något bättre i sökningar i och med att reglerna för höjdbalansen i detta träd är striktare. Detta bör leda till ett jämnare träd höjdmässigt och därmed också snabbare sökningar. De striktare reglerna bör också leda till fler trädrotationer vilket skulle kräva mer tid, men då sökningens effektivitet är av större vikt är detta överkomligt.

- *Hur bör en lösning för att kommunicera med databasen se ut?*
  I och med att fördelarna med intervallträdet mångt och mycket går förlorad om trädet ej byggs på förhand kan konstateras att trädet inte ska byggas upp i samma stund som en sökning ska ske. Detta kommer utav den egenskapen att sökningar i intervallträdet kan ske i O(log n + k) tid medan att bygga upp dem tar O(n log n) tid, där n är antalet noder och k är antalet sökträffar. Träden behöver därmed finnas konstruerade på förhand och redo att traverseras vid en sökning. Det finns olika sätt att lagra ett självbalanserat binärt sökträd. I detta examensarbetet bedömdes en godtaglig lösning vara att låta applikationen köras på en server, bygga träden vid uppstart och sedan hålla dem i minnet. En datavy över larmen tillhandahölls av Trafikverket som gjordes om till en permanent tabell i databasen. Kommunikationen med denna tabell sker i prototypen endast när applikationen startas då datat hämtas och träden byggs upp. En alternativ metod hade exempelvis kunnat vara att lagra träden i databasen med hjälp av olika metoder, men dessa avfärdades i detta examensarbete i första skedet på grund av avgränsningarna kring databasen och den ökande komplexiteten som skulle tillkommit.

- *Hur bör tillgängligheten beräknas?*
  Hur tillgängligheten bör beräknas är det svårt att kunna med säkerhet ge ett slutgiltigt svar då detta kan definieras och lösas på många olika sätt. I detta examensarbetet valdes att använda Googles definitioner av tillgänglighetsmått då dessa omfattar just de mått Trafikverket efterfrågade. En fungerande implementation kunde göras i detta program. Intervallen kunde delas upp efter utrustning och tjänst och överlappande intervall kunde sammanfogas med hjälp av en stödklass i form av datastrukturen stack. När intervallen väl var sammanfogade itereras dessa och upptid samt nertid i millisekunder kan summeras. Tillsammans med längden av arrayen av intervall finns nu alla de delmått som behövs för att definiera de tre

driftsäkerhets mått enligt Googles definitioner och dessa ekvationer implementerades. Då denna lösning fungerar, returnerar de mått kunden efterfrågat och ger rimliga resultat tidsmässigt undersöktes ej ytterligare lösningar.

- *Är slutresultatet en effektiv lösning för datafiltrering av tidsintervall?*
  De test som gjorts har kunnat visa på ett slutresultat som är effektivt och ser ut att bli ännu mer effektivt i produktionsmiljö. Då fördelarna med intervallträdet är att sökningar efter överlappande intervall inom ett intervall är effektiva enligt dess egenskap av O(log n + k) per sökning. I testfall 1 i kapitel 5.2.1 traverseras alla noder i trädet och då alla noder måste besökas blir sökningen endast detsamma som en for-loop, O(n). Detta bör vara högst ofördelaktigt för intervallträdets effektivitet. Att denna sökning ändå var ungefär 8 gånger snabbare än den mer naiva jämförelselösningen talar för att prototypen är effektiv. I sökningar med många resultat och där inte hela intervallträdet behöver traverseras visar testfallen som mest en 41 gånger snabbare sökning och beräkning än att hämta ut datat, som alternativt kunnat göras, ur en databas i testfall 2 i kapitel 5.1.2. Graf 1 visar också som tidigare nämnt att lösningen är mycket mer skalbar och effektiv desto fler larm som behöver hanteras. Prototypen lyckas även mitigera effekterna av nackdelen med uppstartstid då prototypen bygger träden vid uppstart och håller dessa i minnet.

6.3 Reflektion över etiska aspekter

Mycket av det data som hanterats i detta examensarbete är av konfidentiell natur och har därför krävt vissa försiktighetsåtgärder. En sådan åtgärd är att all data har anonymiserats. Detta innebär att exempelvis samtliga tabeller, kolumner och namn har bytts ut. All data och information som inte är väsentlig för examensarbetets syfte och fullbordande har även filtrerats bort. På grund av både enkelhet och säkerhet har en statisk datamängd använts som testdata. Datat som använts är hämtat ur ett intervall av de fyra första månaderna av år 2021 och mellanlagras i en databas istället för att koppla upp mot övervakningssystemen direkt.

Då detta examensarbete avgränsat bort autentiseringsfunktionalitet i prototypen har applikation ej laddats upp på en server i syfte att inga obehöriga klienter ska kunna ställa frågeställningar till applikationen. På grund av detta har applikationen endast körts och testats på en lokal server. Då ingen autentisering implementerats har heller inga frågor om personuppgifter och datainsamling uppkommit utan dessa frågor behövs endast besvaras under eventuell vidareutveckling.

6.3 Framtida utvecklingsmöjligheter

Det finns vissa funktionaliteter som saknas i prototypen vilka skulle behövas vid en eventuell produktionssättning. Dessa är främst säkerhetsfrågor vilka skulle enligt dialog med kunden förbises vid implementationen av prototypen, men som skulle behövas i praktiken i ett senare skede. Prototypen saknar autentiseringsfunktionalitet utöver uppkopplingen till databasen. Det bör finnas en autentisering för att kontrollera klientens behörighet att ställa frågor till

applikationen. HTTP-metoden POST bör även användas i produktionsmiljö istället för GET för att säkrare överföra datan och inte riskera att data lagras i webbläsarens cacheminne eller historik. Utöver autentiseringsfrågor bör även kod för att underhålla träden implementeras. Ett skript för att hämta ny data till databasen skulle behövas och även exempelvis ett skript eller en trigger-funktion i databasen som uppdaterar, i form av insättning och borttagning, de träd som applikationen håller i minnet. Slutligen innehåller inte heller prototypen någon algoritm för att hantera de fall där en tjänst innehar redundans, det vill säga att tjänsten inte är otillgänglig om enbart en av de redundanta utrustningarna är nere. Detta skulle kunna lösas hjälp av en extra kolumn i datat i databasen som talar om ifall och vilka utrustningar är redundanta och att vid sammanfogning av intervall kontrolleras detta värde. Om inte båda utrustningarna har larm under samma intervall ignoreras dessa i beräkningen av tjänstens tillgänglighet. I ett senare skede kunde även fler filtreringsparametrar behövas och detta går att tillägga genom modifiering inuti intervallträdspaketet [15]. Detta är bevisat som möjligt av prototypen då filtrering på utrustning sker just på detta sätt.

Medan prototypen fungerar och ger goda resultat finns även rimliga alternativa lösningar eller modifikationer som eventuellt skulle kunna förbättra prototypen och som ej undersökts i detta examensarbete. Ett sådant fall är eventuella alternativa uppdelningar av träden om detta behövs alls, exempelvis att ha ett stort träd eller dela upp efter utrustningar. Eventuellt skulle även andra definitioner av tillgänglighet kunnas användas eller en annan lösning för att sammanfoga intervallen. Prototypen är gjord specifikt för Trafikverkets IT-miljö men skulle även med få anpassningar kunna användas inom alla olika sorters IT-system så länge det finns intervalldata tillgänglig.

7. Terminologi

AVL-träd	Ett självbalanserande binärt sökträd som bibehåller balansen genom att övervaka avståndet från noder till löv.
Express.js	Ett minimalistiskt webbapplikationsramverk för Node.js som tillhandahåller bland annat grundläggande webbapplikationsfunktioner.
Intervallträd	Ett binärt sökträd för att söka bland intervall. Kan implementeras på olika sätt.
MTBF	Förkortningen står för “mean-time-before-failure”. Tillgänglighetsmått på medeltiden till att ett fel inträffar.
MTTR	Förkortningen står för “mean-time-to-repair”. Tillgänglighetsmått på medeltiden för återhämtning efter ett fel.
Node.js	Ett open-source asynkront eventbaserat javascript runtime, vilken möjliggör användandet av javascript i backend.
NPM	Förkortningen står för “Node package manager” och är en pakethanterare för Javascript-ramverket Node.js
Runtime	Refererar i detta examensarbete till den miljö och tid där koden exekveras. Detta sker i ett mer hårdvarunära språk än annars vanligt för exempelvis Javascript.
Röd-svart träd	Ett självbalanserande binärt sökträd som bibehåller balansen genom att färga noder röd eller svart baserat på ett regelverk.
Service level Agreement (SLA)	Ett serviceavtal som sätter krav på leveranser angående vilka kvalitetsnivåer dessa ska ha.

8. Källförteckning

[1] T. H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to algorithms*, 3. uppl., Cambridge, USA, MIT Press, 2009.

[2] Elliot B. Koffman, Paul A. T. Wolfgang. *Data Structures: Abstraction and Design Using Java*, 3rd Edition. Wiley 2016

[3] G. M. Adel'son-Vel'skii, E. M. Landis, “An algorithm for organization of information”, *Dokl. Akad. Nauk SSSR*, 146:2 (1962), 263–266

[4] R. BAYER, “Symmetric binary B-Trees: Data structure and maintenance algorithms”, *Acta Informatica* 1, ss. 290–306, 1972.

[5] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees", *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, ss. 8-21, 1978, doi: 10.1109/SFCS.1978.3.

[6] D. T. Lee, “Interval, Segment, Range, and Priority Search Trees”, *Handbook of Data Structures and Applications*, D.P. Mehta & S.Sahni (redaktörer), kapitel 18, CHAPMAN & HALL/CRC, USA, Okt. 2004.

[7] P. Brass, *Advanced Data Structures*., New York, USA, Cambridge University Press, 2008.

[8] E. Hanson & M. Chaabouni, *The IBS-tree: A Data Structure for Finding All Intervals That Overlap a Point*, USAF Wright R. & D. Center, Dayton, USA, Nov. 1994

[9] R. E. Tarjan, “Updating a balanced search tree in O(1) rotations”, *Information Processing Letters*, vol. 16, ss.253-257, 1983.

[10] A. Ross, A. Hilton, M. Brown och D. Rensin, "Available . . . or not? That is the question—CRE life lessons", Google, [Online]. Jan. 2017. Tillgänglig: https://cloud.google.com/blog/products/gcp/available-or-not-that-is-the-question-cre-life-lessons (hämtad: 2021-11-04).

[11] Node.js, version 4.17.1, [mjukvara], OpenJS Foundation, 2009. Tillgänglig: https://nodejs.org/en/

[12] OpenJS Foundation, "About Node.js®", [Online]. Tillgänglig: https://nodejs.org/en/about/ (hämtad: 2020-10-12).

[13] OpenJS Foundation, "Introduction to Node.js", [Online]. Tillgänglig: https://nodejs.dev/learn/introduction-to-nodejs (hämtad: 2020-10-12).

[14] Express.js, version 4.17.1, [mjukvara], TJ Holowaychuk, StrongLoop et al., 2010. Tillgänglig: https://expressjs.com/

[15] node-interval-tree, version 1.3.3, [mjukvara], M. Žarković och P. Krafft, 2017. Tillgänglig: https://github.com/ShieldBattery/node-interval-tree

[16] J. Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*, 2nd Edition. Morgan Kaufmann 2012