# 5DV204/VT18: OU1 – Matris Rep.

## Buster Hultgren Wärn

## [dv17bhn@cs.umu.se](mailto:dv17bhn@cs.umu.se)

## 2018-03-08

# Innehåll

Detta inkluderar hela källförteckningen =D.

# 1 Inledning

# 2 Kriterier

Ett antal kriterier har tagits fram som representationen av matrisen behöver följa. Dessa skall kunna mätas objektivt. Kriterierna delas huvudsakligen upp i tre olika delar.

**Hanterbart för användaren**

Detta är förmodligen ett underförstått kriterium, men ändå ett viktigt. Användaren skall kunna sätta in, ta bort och undersöka vissa element i datatypen. Med andra ord, användaren skall ha full kontroll över de element som sätts in i datatypen.

Därmed ges tre essentiella operationer som krävs för datatypen:

1. Lookup – att söka fram en position och dess element – ifall det finns.
2. Insert – Att lägga in ett element i en viss position.
3. Remove – Att ta bort ett element från en viss position.

Viktigt att notera är att dessa operationer utgår ifrån ett elements position och inte dess värde. Positionen agerar på sådant sätt som en nyckel.

**Rumskomplexitet**

Syftet med matrisrepresentationen är att minska på minnet. En matris använder alltid minne för antal rader och antal kolumner. Ifall en matris är gles skall så lite minne används som behövs, därför väljs det andra kriteriet, och det viktigaste, att endast de element som representeras i matrisen skall representeras i datatypen.

Inga gränser ges på hur många rader matrisen kan vara, eller hur många element den kan hålla. Därför sätts det tredje kriteriet till att datatypen skall vara så dynamisk. På detta sätt är det naturligt för datatypen att lägga till fler element utan någon begränsning.

**Tidskomplexitet**

Fördelen med en matris är hastigheten i dess operationer. Då ett speciellt värde skall sättas in eller undersökas i en position krävs ett konstant antal operationer. Detta går till vissa sätt att uppnå dynamiskt via approximationer, men dessa har sina nackdelar i att datatypen inte blir fullt dynamisk. Följande kriterium två och tre behöver skall det endast finnas antal representationer för element. Kriterium fyra menar: de essentiella operationerna specificerade i kriterium ett för datatypen skall ha i tidskomplexitet.

Mer värde kommer läggs på lookup, då denna bör användas mest. För att sätta in eller ta bort ett element ett visst värde måste först positionen tas fram. I vissa fall kommer den vara upptagen och i vissa fall inte. Det är svårt att sätta objektiva gränser på hur mycket fokus bör läggas på detta, därför läggs måste det hållas till de tidigare kriteriernas gränser. Det är även svårt att mäta objektivt hur långt detta kan dras. Därför sätts det endast som – desto mer desto bättre. Kriterium 5 handlar om att datatypen skall kunna optimera lookup så långt det går utan att konfiskera något annat kriterium.

**Sammanfattning av kriterium**

1. Datatypen skall vara hanterbart för användaren och måste infatta operationerna Lookup, Insert och Remove.
2. För antal element i datastrukturen skall minneskomplexiteten inte överskrida .
3. Element skall kunna lägga till och tas bort dynamiskt.
4. Objektiv del: Ingen av operationerna i kriterium 1 skall övergå tidskomplexiteten för antal element. Subjektiv del: desto snabbare operationerna är desto bättre.
5. Objektiv del: Lookup skall optimeras på bekostnad av andra operationer. Detta måste dock ske utan att konfiskera något utav de objektiva delarna från kriterier 1-4. Subjektiv del: Desto snabbare Lookup är desto bättre.

# 3 Representationer

**Dubbellänkad sorterad lista**

Detta är den enklaste representationen av alla fyra som skrivs om i denna rapport – en dubbellänkad lista, som sorteras vid insättning.

Problemet med att använda en lista istället för ett fält är att värden är att ifyllda positioners position i listan är okänt. Detta går dock att förbättra genom att sortera listan för att få en approximation om var ett element kan befinna sig. Positioner rankas som höga eller låga beroende på x, y eller en kombination på dem båda. Något sorts huvud sparar position på var första och sista plats i listan finns. Figur 1 visar ett exempel på en sådan lista som sorterar på x position.



Figur 1

För att hitta ett element i denna lista kan sökningen börjas ifrån den ände som har störst chans att vara närmast elementet. T.ex. om listan i figur 1 har 100 element i stigande ordning, där representerar sista elementet i listan och element på position söks kan listan börja sökas från sista element i listan istället för första.

Detta är dock inte optimalt. Listan behöver fortfarande traverseras och Ifall sista elementet i listan skiljer mycket från medelvärdet, så kan listan börja söka ifrån den mindre optimala änden.



Figur 2

Figur 2 visar ett exempel på detta. Ifall element med position skall finnas så kommer listan börja från det lägsta värdet istället för även fast är närmst. Detta är för att 5 som tal är närmare 1 än 100. Vägen kanske är längre men detta vet inte huvudet om.

Det finns dock optimeringar för att göra listan ännu mer effektiv. T.ex. att räkna ut ett bättre medelvärde, eller ha flera stycken positioner i listan. Dock så kan alltid dessa special fall dyka upp vilket gör det svårt att utse ett asymptotiskt medelvärde. Sämsta tiden för att traversera listan är som konstateras via exemplet i figur 2.

Ett annat problem som stöts på är hur sorteringen skall skötas. Det är inte självklart vad som får störst värde eftersom det finns två olika nycklar för varje position – x och y värde. I Figur 1 så sorteras listan endast på x-koordinat, men ett värde på x-axeln kan innehålla flera på y-axeln.

**Träd på bredden**

Denna representation använder sig av den dubbellänkade sorterade listan. Det är ett träd som bygger på bredden med höjd tre. Roten, med djupet 0, agerar som huvud för trädet, där alla barn med djupet 1 är x eller y-koordinaterna för positionerna. Barnen som har djupet 2 har då motsatta koordinat till barnen i djup 1. Det är här elementen läggs in. På detta sätt nås först en koordinat, och sedan dess motsvarande på andra axeln.



Figur 3

Problemet med detta träd är implementation. Roten behöver hålla reda på varenda bar och kunna öka dynamiskt. Detta sker alltså med hjälpen av den dubbellänkade sorterade listan. Varje förälder, inklusive roten, kommer att vara huvudet till en sådan länkad lista. Figur 4 representerar denna implementation.



Figur 4

Detta betyder att alla problem som finns i den dubbellänkade sorterade listan följer med här, men trädet har några bättre lösningar till problemet. Ifall flera element är insatta på samma axel så behöver inte alla dessa element traverseras för att hitta ett element på en annan axel.

T.ex. om skall finnas från figur 1, så behöver inte alla element i axeln – alltså och – traverseras. En hel axel kan på samma gång traverseras igenom om elementet inte finns på den axeln.

Värsta fallet för att hitta ett element är om alla element har lagts in på x eller y-axeln. I sådana fall går det inte att traversera igenom en hel axel åt gången och datatypen kommer ha samma tidskomplexitet som den dubbellänkade sorterade listan – alltså .

**Hash ”matris”**

Denna datatyp är egentligen inte någon matris utav element utan en matris av funktionell matris av hash tabeller. Varje element i matrisen är sin egen hash tabell. Dock byggs inte datatypen som en matris utan får snarare en matris kvadratiska effekt.

Den första tabellen använder x-koordinaten som nyckel för ett element. En hash funktion, med hjälp av x-nyckeln, delar in elementet i en bucket mellan 1 - . Hash tabellen inuti varje hink använder därefter på y-koordinaten som nyckel för varje element. Samma hash funktion, fast nu med y-nyckeln, delar in elementet i en bucket mellan 1 - . Dessa representerar nu en riktad lista. Denna riktade lista kan även vara en dubbellänkad sorterad lista för att optimera operationerna ännu mer, men detta är inte essentiellt. Figur 5 är ett exempel på denna struktur.



Figur 5

Denna representation har olika fördelar och nackdelar.

Hashtabellens hastighet är beroende på fyllnadsgraden, vilket beräknas olika beroende på ifall öppen eller sluten hashning används. Denna implementation kommer endast att fokusera på öppen hashning. Anledningen till detta är att den är generellt mer dynamisk och mindre beroende på tomma buckets vilket leder till en snålare minnesanvändning.

Den genomsnittliga tiden för en operation är, som beskrivs i Datatyper och algoritmer, n/B där n är antalet element och B är antalet hinkar. Alltså, desto fler element som används, desto mindre effektiv blir operationerna. [x]

Detta är både styrkan och nackdelen bland hashtabeller. Används lika många buckets som element, med en bra hashfunktion, så kommer operationerna närma sig konstant tid. Dock så kommer, då antalet element ökar, operationerna att få komplexiteten O(n).

Det finns sätt att öka antalet hinkar dynamiskt, men det finns två problem med denna dynamiska utökning kring denna rapport.

1. Utökningen äter upp mer och mer minne desto effektivare hashtabellen bör vara.

2. Det går alltför långt in på djupet av hashtabeller och det finns ingen konkret information kring implementation.

**AVL träd**

Ett vanligt binärt sökträd som sorteras vid insättning kan öka i djupet upp till O(n) i värsta fall och får därmed O(n) komplexitet. Genom att Balansera dessa träd vid insättning så att de blir kompletta tar operationerna för Lookup O(log(n)) i tidskomplexitet, men det tar tid att fullt balansera.

AVL träd är en variation på binärt sökträd, som hela tiden balanserar trädet till en viss gräns vid insättning och borttagning. Trädet tillåter en höjdskillnad på +-1. På detta sätt Hålls effektiviteten av trädet alltid på O(log(n)) för alla operationer.

För att göra detta komplett med det koordinatsystem som behövs för representation kommer det finnas, precis som i tidigare representationer, olika träd för x och y-axeln. Första trädet represnterar alla x axlar som har fyllda element i matrisen. Alla noder i dessa träd har som värde roten till ett nytt AVL träd. Detta träd innehåller y-värderna för som finns i just denna x-axel.

INSERT SEXY PIC <3

Med denna variant så kommer trädet alltid att behålla sin komplexitet på O(log(n)).

# 4 Utvärdering

För att välja den bästa representationen så bedöms dem efter kriterierna. Alla representationer klarar av de olika kriterierna, och det krävs en mer komplicerad analys, utifrån kriterierna, för att bedöma det bästa. Hashtabellen kommer att undersökas sist, då den är mest beroende på implementation.

Den dubbellänkat sorterade listan är direkt värre än att representera trädet på bredden. De har båda värsta fall komplexitet, O(n), men då det finns flera stycken element samlade på samma x-koordinat så kan trädet hoppa över dessa element ifall det söker en position på en annan x-koordinat.

Den kräver dock mer minne, vilket i värsta fall är den dubbla mängden av antal element. Denna rumskomplexitet får O(n) och klarar fortfarande av kriterium 2.

Bredden trädet står däremot inte upp emot AVL trädet. Båda två kräver samma minneskomplexitet, då dem är byggda lika. Varje x-koordinat får en egen nod, och alla y-koordinater som finns på den x-koordinaten blir sin egen nod. Därmed, har dem samma värsta fall då varje x-koordinat endast har en y-koordinat, det dubbla antalet noder per element.

Så jämförelsen går enbart på tidskomplexiteten

# 5 Slutsats

# 6 Källförteckning

[#] L.E. Janlert, T. Wiberg, ”Datatyper och algoritmer”, Upplaga 2.6, Lund: Studentlitteratur AB, 200, s. 278-279.