UMEÅ UNIVERSITET
Institutionen för Datavetenskap
Rapport obligatorisk uppgift

# DV2: Algoritmer och problemlösning 7.5 p, 5DV169

Obligatorisk uppgift nr

4



| | | |
|---|---|---|
| Namn | Axel Sjölinder | |
| E-post | dv14asr | @cs.umu.se |
| Cas-id | axsj003 | @student.umu.se |
| Datum | 31/08 -17 | |

# Innehåll

# 1. Inledning

Ett företag behöver hjälp med att utveckla ett kalkylprogram där bladen består utav en enda stor tabell. Problemet är att vi inte vet hur användaren kommer att representera datan. Då användaren skulle kunna lägga in data med stort mellanrum från varandra, då bladen är oändligt långa. Detta är väldigt resurskrävande, så en mer ekonomisk lösning till problemet måste hittas.  Rapporten tar fram tre representationer som är utvärderade från tre kriterier för att komma fram till vilken av representationerna som är bäst.

Rapporten är uppdelad i fyra större delar.
**Kriterier**, där kriterierna tas upp och beskriver hur de kan mätas på olika sätt.
**Representationer**, här beskrivs tre representationer så att man kan implementera datatypen utifrån beskrivningen.
**Utvärdering**, tar kriterierna och utvärderar de givna representationerna.
**Slutsats**, drar en slutsats kring vilken representation är bäst och varför. [1]

## 2. Kriterier

Dessa kriterier togs fram under ett seminarium med 7 andra studenter från Umeå Universitet datumet 16 februari 2017. [6]

### Tidskomplexitet vid sökning

Tidkomplexitet är något som i slutändan är väldigt viktigt. För att få ett väl fungerande program så måste det kunna utföra en uppgift inom vissa tidsramar.
För att begränsa de olika tidskomplexiteter olika operationer kan ha tittar denna rapport mer på tidskomplexiteten vid operationen sökning. Självklart är en generell tidskomplexitet viktig, men att ha en snabb sökning kan enkelt förbättra det efterfrågade programmet.
Tidskomplexiteten uttrycks med hjälp utav Ordo uttrycket. [2]

### Minneskomplexitet

Från problembeskrivningen kan det finnas stora mellanrum mellan varje insatt värde, därav kan programmet bli väldigt stort. Då kan det vara ett bra val att tänkta på rumskomplexiteten, att jobba med att få programmen så små som möjligt.

### Enkel implementation

Enkel implementation underlättar, förbättrar och förenklar många olika delar. Oftast är de enklaste implementationerna de mest buggfria och de enklaste att underhålla, skulle man vilja utveckla något blir det mycket simplare.

# 3. Representationer

## 3.1 Riktad Lista

En riktad lista kan endast förflytta sig i en riktning, före/efter relationen finns fortfarande kvar men hur den tar sig till elementen kan endast ske i en riktning. Vid insättning av element så sätts elementet in på före den angivna platsen, så skulle elementet sättas in i plats tre så får det nuvarande elementet på plats tre plats fyra. Skulle tre element sättas in först i listan, samtliga sätts in på första plats, då skulle det sista insatta elementet ligga först i listan. [3]

För att representera riktade listan som tabell så skulle en lösning vara att modifiera listan något. Görs listan om så att den även kan utnyttja nycklar kan dessa nycklar representera fältets/tabellens koordinater. Nycklarna leder dess vidare till värdet utav elementet. Insättning skulle kunnas göra efter sorterade nycklar men är inget krav, skulle inte nycklarna sorteras skulle insättningen gå fortare men sökning i riktade listan generellt gå långsammare.
Representationen skulle även kunna ta hänsyn till tomma värden i tabellen och inte ta med dem i listan [4]

## 3.2 Kö

En motsvarar något som kallas first in first out (FIFO). Vilket betyder att det första elementet i kön är det första elementet som visas eller ”dequeuas” vilket betyder borttages ur kön.
Enklaste representationen av en kö är en vanlig kö i en matbutik, kunden längst fram i kön är den kund som tas hand om först.

Representera en kö som en tabell kan tänkas mycket likt hur den riktade listan representerar en tabell. Om nycklarna kan användas på samma vis för koordinaterna och varje nyckel representerar ett värde så kan en kö använda samma representation. Det som skiljer kön från den riktade listan är köns funktion att endast lägga värden sist i kön medans den riktade listan kan välja en position, samma med köns funktion att endast ta bort första värdet i kön.
Detta gör att den som vill representera kön som en tabell har valmöjligheten om de vill lägga tid på att sortera kön efter nycklarnas storlek eller inte. Att sortera en kö krävs en del resurser men gör även att kön inte behöver läsa olika delar från tabellen. [5]

### 3.3 Binärt Träd

Ett binärt träd är en datatyp som kan liknas ett träd (därav namnet). Varje nod kan högst ha två barn, ett vänster barn och ett höger barn. Barnen kan därefter ha egna barn på samma sätt som beskrivet innan, på detta sätt växer trädet utifrån behovet.

För att representera ett binärt träd som tabell finns olika lösningar, ett är att ta hjälp utav structs. I structen kan indexen samt värdet sparas, för att hitta elementet sen i trädet letar man det eftersökta värdet med hjälp utav structsen och därefter hittar dess index för tabellen/matrisen. Trädet kan sorteras på ett sådant sätt att vid varje insättning sker en jämförelse av tidigare insatta noder för att besluta sig om det nya värdet är mindre och skall hamna i det vänstra lövet eller högra och då hamna i det högra lövet, startat från toppen (roten).
Enda problemet som kan uppstå med följande implementation är att trädet blir väldigt djupt.
En lösning till det problemet är att ta bort alla tomma värden och inte låta dem tas med i trädet. [6]

# 4. Utvärdering

Här utvärderares representationerna med hjälp utav de valda kriterierna. Rapporten kommer gå genom ett kriterie i taget för varje representation.

## 4.1 Tidskomplexitet vid sökning

### Riktad Lista

Riktad kan endast gå åt ett håll och vilket gör att en sökning måste gå genom hela listan vilket ger sökning en komplexitet på O(n)

### Kö

I en kö kan endast det första elementet inspekteras, vill vi söka nått i mitten av kön måste vi ta ut elementen tills vi kommer till det vi söker vilket gör det lite besvärligt. Men en generell sökning i kö har en komplexitet på O(n).

### Binärt träd

Beroende om det binära trädet har en bra balans eller inte har det en komplexitet på O(n) i sämsta fall och O(log(n)) i bästa fall.

## 4.2 Minneskomplexitet

### Riktad Lista

Riktad lista är en dynamisk datatyp vilket betyder att under listans livstid så kan storleken bli större och mindre. Detta är praktiskt för att hålla nere programmets storlek.

### Kö

En kö är en dynamisk datatyp. Vilket precis som riktad lista gör att det blir praktiskt att anpassa storleken av programmet beroende på hur stort programmet behöver vara.

### Binärt träd

Binära träd är dynamiska så de växer och krymper under användarsessionen vilket gör datatypen väldigt minnes vänlig.

## 4.3 Enkel Implementation

### Riktad Lista

Att bygga en tabell utav en riktad lista fungerar bra, värt att tänka på eftersom det är en riktad lista kommer man behöva traverserna genom hela listan för att hitta det man söker.

### Kö

En implementation utav kö är simpelt. Men vissa problem kan uppstå.. Tillexempel när det kommer till sökningen i kön, ska ett värde i mitten av kön tas bort måste kön ta bort och lägga till värdarna emellan det tänka värdet som skulle tas bort. Med andra ord kön måste flyttas ett snäpp tills vi kommer till vårt värde.

**Binärt Träd**

Att implementera ett binärt träd i C är relativt enkelt med hjälp utav structs och pekare. Dock då ett binärt träd inte liknar ett kalkylblad så blir det inte så självklart att ta den till användning i detta tillfälle.

| | 1.Lista | 2.Kö | 3.Binärt Träd |
|---|---|---|---|
| 1.Tidskomplexitet | O(n)<br>Helt okej | O(n)<br>Helt okej | O(n)/O(log(n))<br>Snabb |
| 2.Minneskomplexitet | Bra | Bra | Bra |
| 3. Implementation | Enkel | Något Komplicerat | Komplicerad |

*Figur 1. De olika komplexiteterna för respektive implementation.*

Förklaring av Figur 1:

1.1 Lista Tidskomplexitet – Helt okej, inte snabbast men finns sämre val.
1.2 Lista Minneskomplexitet – Bra.
1.3 Lista Implementation - Enkel, utav de tre representationerna den enklaste.

2.1 Kö Tidskomplexitet – Helt okej, inte snabbast men finns sämre val.
2.2 Kö Minneskomplexitet - Bra
2.3 Kö Implementation – Något komplicerad, inte den enklaste men inte heller den mest komplicerade av de tre.

3.1 Binärt Träd Tidskomplexitet –Snabbast utav de tre.
3.2 Binärt Träd Minneskomplexitet - Bra
3.3 Binärt Träd Implementation – Den mest komplicerade utav de tre.

# 5. Slutsats

För att dra en slutsats av tabellen.

**Lista**
Lista har en helt okej tidskomplexitet, det finns datatyper med bättre men absolut ingen skam. Minneskomplexiteten är bra då datatypen är dynamisk. Att implementera en lista är också simpelt.

**Kö**
Den generella tidskomplexiteten utav sökning i kön ligger på O(n) vilket är inte någon direkt skam. Men när det kommer till det användningsområde vi behöver den till kommer tidskomplexiteten vid sökning bli sämre än så, utav den simpla anledningen att måste flytta om i kön för att hitta det element vi söker efter. Ett plus är att kön är dynamisk och hjälper till att hålla minneskomplexiteten nere. Att implementera en kö till vårat syfte kan fungera, men är knappast praktiskt.

**Binärt Träd**

Har en väldigt bra tidskomplexitet, den bästa utav de tre valda representationerna. Även minneskomplexiteten är bra. Men att implementera ett binärt träd för att fungera för uppgiften i fråga kan vara lite mer komplicerat.

Men vilka är då bäst?

Denna fråga kan vara svår att svara på då problembeskrivningen är något luddig, företaget specificerar aldrig vad de önskar ska vara bäst. Det gör att vi som leverantör får ta ett beslut. Vill vi ha en snabb-, eller en låg minnes-, eller en datatyp som är enkel att implementera? Personligen skulle jag välja riktade listan på grund utav den enkla implementationen, jag förlorar lite tidskomplexitet men det kan jag köpa för det implementation vänliga jag tjänar.

# 6. Källförteckning

[1] Problembeskrivning utav ou4, OU4- Val av datarepresentation *Umeå Universitet*, VT15 [Online] Tillgänglig: https://www.cambro.umu.se/portal/site/57303VT17-1/page/3fec7ad7-6222-4bbd-b91c-a2f2bbcd5924 [Hämtad 21 februari]

[2] Lars-Erik Janlert och Torbjorn Wiberg, *Datatyper och algoritmer*. Upplaga: 2:6. Lund: Studentlitteratur, 2000, s.36.
[3] Lars-Erik Janlert och Torbjorn Wiberg, *Datatyper och algoritmer*. Upplaga: 2:6. Lund: Studentlitteratur, 2000, s.65-66.
[4] Lars-Erik Janlert och Torbjorn Wiberg, *Datatyper och algoritmer*. Upplaga: 2:6. Lund: Studentlitteratur, 2000, s.50-54.
[5] Lars-Erik Janlert och Torbjorn Wiberg, *Datatyper och algoritmer*. Upplaga: 2:6. Lund: Studentlitteratur, 2000, s.155-156.
[6] Lars-Erik Janlert och Torbjorn Wiberg, *Datatyper och algoritmer*. Upplaga: 2:6. Lund: Studentlitteratur, 2000, s.194-197.
[7] Problembeskrivning utav ou4, OU3- Framtagande av kriterier *Umeå Universitet*, VT15 [Online] Tillgänglig: https://www.cambro.umu.se/portal/site/57303VT17-1/page/3fec7ad7-6222-4bbd-b91c-a2f2bbcd5924