

5DV169 – OU1: Huffmankodning



March 12, 2018

Jakob Lundin

jalu0018 c14jln

**Innnehållsförteckning**

[Inledning 1](#_Toc508625370)

[1. Frekvensanalys 1](#_Toc508625371)

[2. Byggande av huffman-trädet 2](#_Toc508625372)

[2.1 Datatypen binaryTree 2](#_Toc508625373)

[2.2 Skapande av prioritetskön 2](#_Toc508625374)

[2.3 Byggande av trädet 2](#_Toc508625375)

[3. Kodning 3](#_Toc508625376)

[3.1 Tabell 3](#_Toc508625377)

[3.2  Kodning 3](#_Toc508625378)

[4. Avkodning 3](#_Toc508625379)

[5. Reflektion 3](#_Toc508625380)

# 

# TRÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄD

Inledning

I denna rapport går jag igenom min process I skapandet av mitt huffman-program. Koden som används skrevs under perioden 5/2 till 12/2 2018. Programmet kompileras genom att navigera till programmappen I ett linuxsystem och köra kommandot ”make”, eller kompilera med alla .c-filer I mappen inkuderade. Programmet kan koda och avkoda en fil till en ny fil genom huffman-kodning, genom att köra en frekvensanalys på en tredje fil. Analysen används för att bygga huffman-trädet. Då implementationer kan skilja kan kompabilitet med andra program eller analysfiler ej garanteras. Programmet tar in fyra argument: en flagga för att visa om kodning eller avkodning ska göras på filen: "-encode” för kodning och   
“-decode” för avkodning, sökväg till analysfilen, sökväg till filen som ska kodas/avkodas, sökväg till filen som resultat ska skrivas till. Körning på filen ser därför ut som följande:   
huffman -läge [ANALYSFIL] [INFIL] [UTFIL]

Programmet använder sig av de redan implementerade datatyperna list, bitset och pqueue givna i specifikationen. Det använder också en datatyp binaryTree implementerad för ändamålet.

Programflöde

Programmet kommer i både kodning och avkodning köra en frekvensanalys, och därefter bygga ett huffman-träd baserat på denna analys, efter detta kör den antingen kodning eller avkodning beroende på programläge. Om kodning körs kommer programmet skriva ut originalfilens storlek samt den kodade filens storlek. Om avkodning körs kommer ett meddelande skrivas ut som berättar att avkodning körts korrekt.

För att innan detta validera att alla parametrar är korrekt givna kommer först antalet parametrar inspekteras. Inklusive programnamn ska det finnas 5 argument. Efter detta kollas att lägesparametern är given som en valid sträng, om den ges som något annat än   
”-encode” eller ”-decode” är denna icke giltig. Därefter öppnas alla filer för sig och kontrolleras att de öppnas korrekt. Om någon av dessa valideringar är felaktiga kommer ett felmeddelande ges till terminalen, och programmet terminera.

# Del 1. Frekvensanalys

Ett effektivt huffman-träd byggs upp genom att se till att de tecken som används oftast får så korta bitsträngar som möjligt. Det mest optimala sättet att åstadkomma detta är att köra analysen på filen som ska kodas, men eftersom programmet inte har funktionalitet för att spara gjorda frekvensanalyser blir detta inte möjligt. Programmet är därför gjort för att köra analysen på en annan mindre fil, som är avsett att både kodning och avkodning ska kunna använda sig av. Denna fil är mest effektiv om den är så lik infilen som möjligt i teckenanvändning.

Frekvensanalysen körs genom att skapa ett fält av heltal på storleken av antal existerande tecken, som i de flesta arbetsområden kommer vara 256. Alla värden sätts till 1 för att öka effektivitet senare. Programmet går sedan in i en loop: Ett tecken läses in från analysfilen, och ett värde i fältet där index är det inlästa tecknets ASCII-värde ökas med ett. Loopen terminerar när den nått filens slut. Frekvensanalysen returnerar sedan detta fält för användning i byggande av huffman-trädet.

Del 2. Byggande av huffman-trädet

## 2.1 Datatypen binaryTree

För att kunna bygga huffman-trädet implementeras och används datatypen binaryTree. Datatypen är ett delträdsorienterat träd, som kör alla sina funktioner på de specifika noderna. Alla noderna har en pekare till nodens etikett definierad som datatypen label, som måste definieras i programmet som ska använda sig av typen. Noderna har också en bool-flagga som är sann om noden är ett löv i trädet, och ett fält som innehåller pekare till nodens barnnoder, som kan vara maximalt två stycken.

Datatypen har funktioner för att skapa en nod, och för att sätta eller returnera barnnoder och etiketter. Noden skapas med löv-flaggan satt som sann, och flaggan sätts sedan till falsk när en barnnod läggs till. Datatypen har också en funktion som returnerar om given nod har etikett eller inte, och en funktion för att sätta ihop två delträd. Denna funktion tar in två noder och en etikett, skapar sedan en ny nod med given etikett och de givna noderna som barn. Denna funktion är avsedd att användas på rotnoderna i två träd.

## 2.2 Skapande av prioritetskön

För att bygga upp trädet utifrån frekvensanalysen används en prioritetskö. Datatypen label definieras här som en struktur innehållande ett tecken och ett heltal. Fältet från frekvensanalysen gås igenom, och en ny nod skapas för varje värde. Frekvensvärdet sätts in som heltalet i etiketten, och värdets index sätts in som tecknet i etiketten, eftersom index är ASCII-värdet av karaktären som frekvensanalysen hittat i dokumentet. De skapade noderna läggs in i ett nytt fält. Dessa noder läggs in i en prioritetskö konfigurerad så att noder med lägst frekvensvärde är längst fram i kön.

## 2.3 Byggande av trädet

Huffman-trädet byggs ihop genom att lägga ihop de två noder med lägst frekvensvärde, tills det bara finns en nod kvar, som då är trädets rotnod. Detta görs genom att först ta ut ett värde ur kön, tar bort det lästa värdet ur kön, sedan gå in i en loop så länge kön inte är tom: Läser in och tar bort ett andra värde ur kön, och skapar sedan en ny etikett med det summerade frekvensvärdet från de två noderna, och lägger sedan ihop delträden som noden är rot i. Den nya föräldernoden tilldelas då den nyskapade ettiketten. Den nya noden läggs därefter in i prioritetskön. Därefter läses ett nytt värde in från kön som första värde. Genom att göra denna “osymmetriska inläsning” kommer loopen terminera när det senast inlästa elementet var det sista värdet i kön, vilket betyder att denna nod också är rotnod i huffman-trädet. Denna nod returneras.

# 3. Kodning

## 

## 3.1 Tabell

För att kunna koda en sträng med hjälp av det tidigare skapade trädet måste först alla tecken kunna hitta sin associerade bitsekvens. En tabell byggs upp för att para ihop dessa två värden. Datatypen bitset används för att förvara bitsekvenser. Tabellen är uppbyggd som fält av pekare till bitset, där index återigen är karaktärens ASCII-värde. För att hitta rätt bitsekvens för alla tecken används en djupet-först traversering av trädet. Funktionen sparar vilka vägar den gått genom trädet som en bitsekvens, där vänster barn är 0, och höger barn 1. När funktioner hittar en nod med lövflaggan satt som sann, kollar den i nodens etikett för vilken karaktär den innehåller, och kopierar vägen den gått dit in i ett bitset som läggs in i tabellen på index av karaktärens ASCII-värde.

## 3.2  Kodning

När tabellen är skapad läses en karaktär in från infilen. En bitset-pekare skapas som pekar på det bitset i tabellen kopplad till den inlästa karaktären. Programmet går därefter in i en loop: Direkt innanför loopen finns en inre loop där karaktärens bitset itereras genom, där bits från detta bitset kopieras in i ett temporärt bitset. Om karaktärens bitset tar slut läser programmet in en ny karaktär. Därefter hämtas den nya karaktärens bitset från tabellen. Den inre loopen termineras när den körts 8 gånger, då konverteras temporärt bitset till en karaktär, som skrivs in i utfilen. Den yttre loopen termineras när det senast inlästa tecknet är filens slut. Infilen är nu kodad och ska då kunna avkodas med detta program där resultatfilen ska vara identisk med originalet.

# 4. Avkodning

För att avkoda programmet behöver ingen tabell skapas. Funktionen läser in ett tecken, sätter trädets rot till aktivt tecken, och går sedan in i en loop: En inre loop används, som körs 8 gånger: Varje bit från det inlästa tecknet plockas ut och för varje bit traverseras trädet, till vänster barn om bit är 0, och höger om bit är 1. Barnet sätts till aktiv nod. Varje gång funktionen går till en ny nod inspekteras nodens lövflagga. Om sann läses tecknet från nodens etikett och skrivs till utfilen och den aktiva noden pekas tillbaka på trädets rot. Utanför den inre loopen läses ett nytt tecken in. Den yttre loopen körs tills den läst in slutet av filen. Avkodningen är nu färdig.

# 5. Reflektion

Programmet har många svagheter, både i design och implementering. Det är väldigt många gånger i programmets körning där tecken konverteras fram och tillbaka, vilket hade kunnat lösas genom att använda label-datatypen redan i frekvensanalysen. Datatypen binaryTree är inte helt generell, vilket var målet med den. Den saknar en funktion för att ta bort en specifik nod, och varje nod skulle behöva en pekare till sin föräldernod. Koden kan också vara ganska svårläst på vissa ställen. Jag har lärt mig mycket från denna uppgift, dels hur viktigt det är med tydlig och läslig kod, eftersom detta projekt varit större än tidigare programmeringsuppgifter jag gjort. Jag har också lärt mig mycket på den tiden sen detta programmerades, och skulle därför kunna skriva mycket bättre kod om jag gjorde om det nu.

Arbetsprocessen har varit ganska ostrukturerad, då jag har varit ganska dålig på att planera innan jag kodar. Detta har lett till mycket tid spenderat på debugging, och omskrivning av redan skriven kod. Detta tar jag som lärdom till nästa gång. Att rapporten skrivs så lång tid efter koden skrevs är också något av ett problem, då jag inte har koden lika färskt i minnet, men eftersom andra uppgifter har behövt prioriteras var det så det blev.

Denna implementation av huffman-kodning hade kunnat vara mer effektiv om den istället kör frekvensanalysen på en annan fil skulle köra sin analys på den filen som ska kodas och spara detta i början av den kodade filen, och avkodningen läsa av denna frekvensanalys och bygga trädet utifrån den. Då hade frekvensanalysen gjort en så effektiv kodning på filen som möjligt, och inte bara en approximation. Detta skulle förstås ta bort krypteringsaspekten av huffman-kodningen, då analysfilen kan ses som en krypteringsnyckel. Att spara analysen i filen skulle också vara mer effektivt på större filer, eftersom analysdatat också tar en del uttrymme, men eftersom det inte finns stor anledning att komprimera så små filer är det inte särskilt relevant.

Källor:

Allt i denna uppgift kommer från uppgiftspecifikation, föreläsningar och från mig själv.