5DV204/VT18: OU1 – Huffman

Buster Hultgren Wärn
dv17bhn@cs.umu.se

2018-02-15

Innehåll

1. Inledning . . . . . s. 1
    1.1. Problembeskrivning. . . . . s. 1
    1.2. Kapitel uppdelning. . . . . s. 1
2. Kompilering och körning. . . . . s. 1
3. Givna datatyper. . . . . s. 2
4. Implementation . . . . . s. 2
    4.1. Huffman.c. . . . . s. 2
    4.2. Encode.c. . . . . s. 2
    4.3. Decode.c. . . . . s. 2
    4.4. HuffTree.c. . . . . s. 3
    4.5. BitString.c. . . . . s. 4
5. Gränsyta för implementerade datatyper. . . . . s. 4
    5.1. HuffTree. . . . . s. 4
    5.2. BitString. . . . . s .5
6. Programflöde. . . . . s. 6
7. Resultat. . . . . s. 7
8. Diskussion. . . . . s. 7

1 Inledning

Problembeskrivning

Denna uppgift gick ut på att skriva ett Huffman program som kan komprimera och även läsa komprimerade filer. Detta görs genom en frekvensanalys på hur ofta karaktärer kommer upp. En binär ”trie” byggs upp där karaktärer som dyker upp oftare får lägre djup, och med denna trien kan de karaktärer som används mest komprimeras till allt mindre tecken.

I lösningen ingår det att bygga en frekvensanalys, ett Huffman träd med Huffman tabell, hantera bitar, läsa och skriva till fil.

Kapitel uppdelning

I Kompilering och körning finns instruktioner till hur programmet kompileras och körs.

Givna datatyper berättar om de datatyper som finns för att hjälpa till med implementationen av programmet.

Implementation beskriver kort om vad varje .c fil i programmet har för uppgift, och lite om dess implementation.

Gränsyta för implementerade datatyper ger gränsytan för dem datatyper som har skapats för denna uppgift. Dessa är inte de givna datatyperna.

I Programflöde beskrivs hur programmet exekverar och varför.

Resultat presenterar de mest relevanta resultaten för programkörningen.

I diskussion går jag igenom lite av de svårigheterna som stötts på, och förslag till förändring.

2 Kompilering och körning

Programmet kompileras med en ”make” fil, som ligger med. Make filen innehåller endast ett enkelt kompilerings kommando, som lyder:

makehuffman: huffman.c encode.c decode.c huffTree.c pqueue.c list.c bitString.c

```
gcc -std=c99 -g -Wall -o huffman huffman.c encode.c decode.c huffTree.c pqueue.c
list.c bitString.c
```

För att köra programmet behövs fem parametrar som följande:
./huffman [option] [file0] [file1] [file2]

./huffman – Namnet och kommandot för att exekvera programmet.
[option] – Kommando för att välja ifall en fil ska komprimeras eller en komprimerad fil ska läsas. Giltiga kommandon är –encode och –decode.
file0 – Namn på filen där frekvensanalysen skall göras på.

file1 – Namn på fil som ska läsas. Antingen för att avkomprimera eller komprimera denna fil.
file2 – Detta är filnamnet som programmet skriver till. Antingen den komprimerad, eller den av komprimerade filen.

3 Givna datatyper

Till uppgiften gavs tre stycken givna datatyper. En Dubbelriktad lista, en prioritetskö och en bithanterare – bitset. Listan användes endast i prioritetskön, och prioritetskön användes för att bygga noderna i trädet till.

Bitset är en datatyp för att hantera bitar. Bitar kan läggas in i den, som konverteras till ett endimensionellt char fält där varje char representerar en hel byte. Det är detta fält med chars som sedan skrivs till filen, för att få den komprimerade filen. Däremot fungerade inte bitset som den skulle, vilket tas upp i diskussionen. Ett beslut fattades att inte använda denna, utan skriva en egen istället.

4 Implementation

Detta kapitel tar upp de implementerade filerna, och beskriver dess huvudsakliga funktion och mening. Hur filerna samarbetar med varandra kommer i senare kapitel.

Huffman.c

Huffman.c är ”main” filen som kör alla andra olika filer. Dess uppgifter består utav filvalidering, uppbyggnad av Huffmanträd och program flöde och frigörande av en del minne.

De datatyper huffman.c använder sig av är Huffman trädet och prioritetskön.

Encode.c

Encode.c sköter hand om krypteringens flöde. Detta är efter det att ett Huffman träd har skapats, och kommandot ”-encode” har används som parameter för programmet.

Encode.c använder sig av Huffman trädet och bitString.

Decode.c

Decode.c är flödeshanteringen då programmet ska läsa en krypterad fil och ”avkoda” denna fil. Detta sker efter att Huffman trädet har skapats och kommandot ”-decode” har används som parameter för programmet.

Decode.c använder sig av Huffman trädet och bitString.

Hufftree.c

Hufftree.c är en datatyp specifikt gjord för att bygga ett Huffman träd. Dataypen består av en träd del och en nod del.

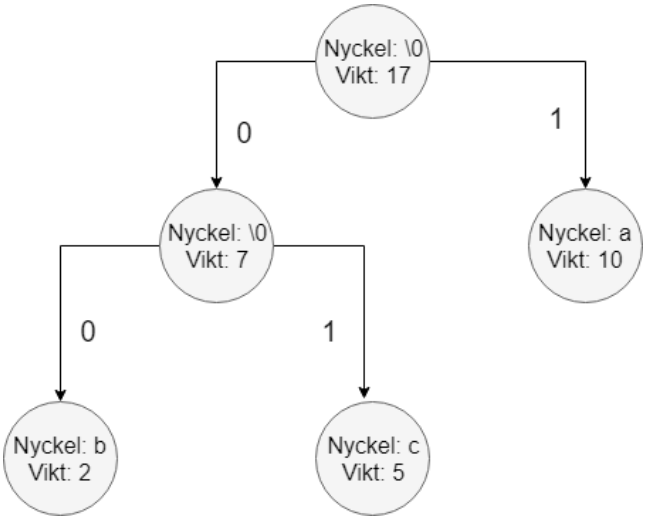
Träddelen innehåller roten till trädet och en Huffmantabell, där roten är en nod kopplat till trädets alla noder. Träddelen har operationer anpassade för att hantera trädet som helhet.

Noddelen är den delen som specificerar en nod, och varje operation en nod har.

Anledningen till uppdelningen är för att enklare kunna skilja på Huffmantabellen, var roten börjar och undersökning av individuella noder. Även fast det finns en inbyggd operation för att traversera trädet och bygga upp en Huffman tabell skulle detta kunna göras med den givna gränsytan för Huffman trädet.

Trädet är ämnat att byggas för att få ett fullt träd, d.v.s. att varje förälder har två barn. Det går dock att undvika detta genom att sätta in NULL som barn, och trädet bör hantera detta utan problem. Trädet kan byggas komplett med dess givna gränsyta.

Etiketten till trädets noder är även nyckel i Huffman tabellen. Ifall det är ett löv, så fylls denna med vilken bokstav som lövet representerar. Ifall det är en förälder så blir etiketten/nyckeln en NULLBYTE för att kunna skiljas ifrån löv. Figur 1 visar ett exempel på ett Huffmanträd.



Figur 1: Huffman träd

Observera att trädet i figur 1 är ett Huffman träd med förälder som har hopslagen vikt av sina barn. Den hopslagna vikten är dock inte nödvändigt för Huffman trädet utan upp till användaren som bygger trädet. Figur 1 innehåller endast tre olika nycklar, och speglar därmed inte hur Huffmanträdet byggs upp med den hela utökade ASCII kodningen.

Huffman tabellen byggs upp enkelt som ett endimensionellt fält med strängar. Varje sträng representerar vägen till varje nod – vänster nod ger en nolla, höger nod en etta – och index i fältet representerar vilken ASCII karaktär som används.

Huffmantabellen för trädet i figur 1 representeras i figur 2. Notera att a, b och c i detta fall representerar dess ASCII tal för läsbarhetens skull:

Nyckel:	0	1	...	...	a	b	c	...	...	255
Värde:					1	00	01			

Figur 2: Huffman tabell

HuffTree använder sig endast av de fysiska datatyper som finns i C.

BitString.c

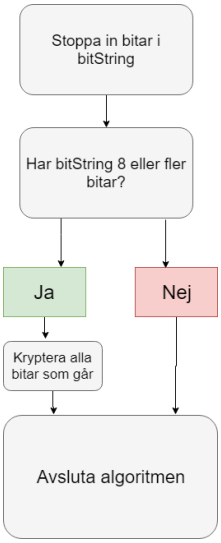
Bitstring är en implementerad datatyp för att hantera bitrepresentation.

BitString hanterar bitar i form utav char strängar. En sträng med bitar stoppas in, och bitString lagrar dessa antingen som bitar eller som en komprimerad byte.

bitString är en dynamisk datatyp, som hela tiden jobbar med att få ner minnet så lågt som möjligt. BitString lagrar endast upp till sju bitar åt gången. Om bitString har åtta bitar eller fler så komprimeras alla bitar som kan till chars. Dock så behövs fortfarande hela den komprimerade texten lagras i bitString.

Figur 3 visar flödet av insättning i bitString.

BitString använder sig endast av de fysiska datatyper som finns i C.



Figur 3: Insättning av bitar i bitString

5 Gränsyta för implementerade datatyper

De datatyper som har implementerats är huffTree och bitString. I detta kapitel beskrivs datatypernas gränsyta. Datatyperna har några fler operationer, men dessa räknas inte med i gränsytan då de endast används som stöd för datatypernas andra operationer.

HuffTree

Då HuffTree består av två olika delar, så finns det operationer för båda dessa.

Trädets operationer följande 6:

Empty (root, size): Skapar ett nytt träd med roten ”root” och storleken ”size”. Returnerar trädet.
IsEmpty (tree): Undersöker ifall ett träd ”tree” är tomt, returnerar i sådana fall 1, annars 0.
Kill (tree): avallokerar allt minne som trädet ”tree” har allokerat. Returnerar ingenting.
GetRoot (tree): Returnerar roten för trädet ”tree”.
Traverse (tree): Traverserar trädet “tree” för att bygga upp Huffman tabellen. Returnerar ingenting.

GetKeyPath (tree, key): Returnerar bitrepresentationen för nyckeln ”key” i trädet ”tree”.

Noderna som bygger upp trädet har sammanlagt åtta operationer som ingår i gränsytan, dessa är:

NewLeaf (weight, key): Skapar ett nytt löv (nod utan barn) med vikten “weight” och nyckeln ”key”. Det nya lövet returneras.
NewNode (node1, node2): Slår ihop två stycken noder – “node1” och “node2” – som blir barn till en ny nod. Den nya noden får summan av barnens vikt. Returnerar den nya noden.
GetKey (node): Returnerar nyckeln från noden ”node”.
IsLeaf (node): Undersöker ifall en nod är ett löv. Returnerar i sådana fall 1, annars 0.
HasLeftChild (node): Undersöker ifall noden ”node” har ett vänstra barn. Returnerar i sådana fall 1, annars 0.
HasRightChild (node): Undersöker ifall noden ”node” har ett högra barn. Returnerar i sådana fall 1, annars 0.
GetLeftChild (node): Returnerar vänstra barnet från noden ”node”.
GetRightChild (node): Returnerar högra barnet från noden ”node”.

BitString

BitString har sammanlagt 7 olika operationer som räknas in i gränsytan:

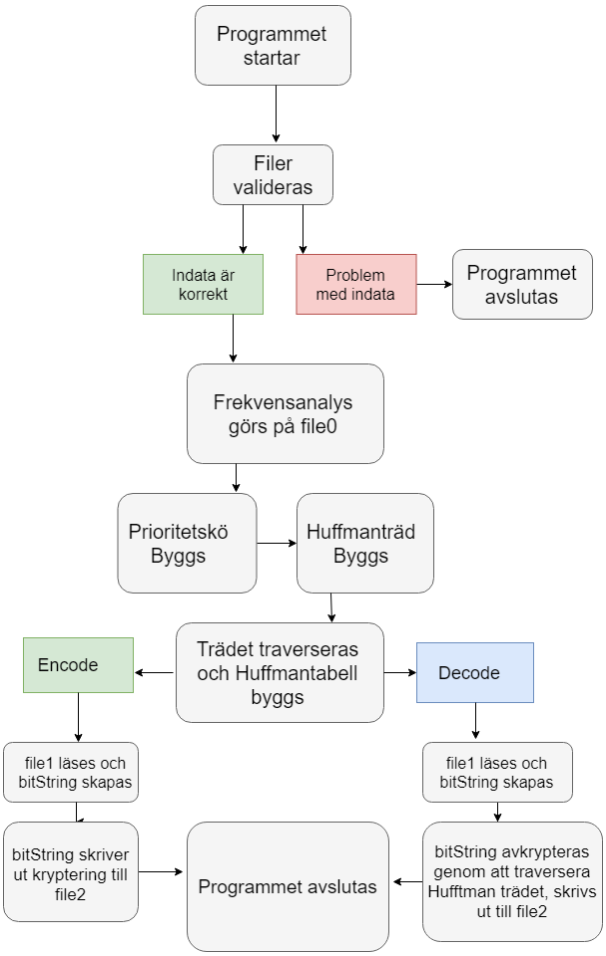
Empty (): Returnerar en tom bitString.
Kill (bs): Avallokerar allt minne som bitString(en) ”bs” har allokerat.
AddBits (bs, bits, nrOfBits): Lägger till bitarna ”bits” i bitString(en) ”bs”. ”nrOfBits” är hur många bitar som läggs in.
AddByte (bs, byte): Lägger till en krypterad ”byte” i bitString(en) ”bs”.
ReadByte (bs, byte, byteNr): Läser en krypterad byte i bitString(en) ”bs”. ”byte” är en representation av 8 bitar. ReadByte pekar tillbaka bitarna till byte.
GetEncode (bs): Returnerar alla krypterade bitar i bitString(en) ”bs”. Lägger även till upp till 7 tomma bitar (0) ifall antalet bitar i ”bs” inte är delbart med 8.
GetSize (bs): Returnerar storleken från bitString(en) ”bs”.

6 Programflöde

Programmet körs från Huffman.c, som styr det huvudsakliga programflödet. Då programmet startar så är dess första uppgift att validera parametrarna. Detta går ut på att kontrollera att det finns exakt 5 stycken, att kommandot är korrekt och att filer kan öppnas för läsning och skrivning.

Efter detta skapas en frekvenstabell upp, som berättar hur ofta varje karaktär i det utökade ASCII kodningen används i file0. Denna tabell används sedan för att skapa upp löv till Huffmanträdet. Löven får vikten av hur ofta dem kommer upp i texten. Efter ett löv så läggs den in i en prioritetskö.

Prioritetskön används för att bygga upp trädet. Löv med minst vikt har högst prioritet, och efter att två löv har slagits ihop till en nod sätts den nya noden in i prioritetskön med hopslagen vikt från dem båda löven. Så om en nod sätts ihop från två löv med vikterna 1 och 4, kommer den nya noden att få vikten 5. Figur 1 i kapitel 4 är ett exempel på hur ett Huffman träd kan byggas. På detta sätt så byggs det hela trädet upp genom att slå ihop olika del träd. Mindre träd byggs upp, sen sammansätts dem till ett större träd. När det bara finns en nod kvar sätts den som rot till trädet.



Figur 4: Programflöde

Trädet är uppbyggt, men det är inte helt klart än, en Huffman tabell måste skapas. Trädet traversernas då rekursivt. Varje nod undersöks, först den vänstra, sedan den högra. Vägen för att nå varje löv sparas i trädets Huffman tabell. Figur 2 i kapitel 4 är ett exempel på hur en Huffman tabell kan se ut.

När Huffman tabellen är färdig så är det dags för att antingen kryptera eller läsa en krypterad fil, beroende på kommando som används vid körning.

Då en fil (file1) skall krypteras körs Encode.c, som börjar med att fylla en bitString. Den gör detta genom att läsa av varje tecken i filen som skall krypteras och lägger in motsvarande ”kryptering” från den uppbyggda Huffmantabellen.

bitStringen skapar sedan upp en krypterad text, och texten skrivs sedan ut till file2.

När allt detta är färdigt så av allokeras minne, och programmet har kört färdigt.

Ifall en krypterad fil (file1) skall läsas, så är både flödet och algoritmerna annorlunda. Den krypterade filen läses av, och sparas som en bitString. Sedan behöver varje krypterat tecken läsas av, och detta

görs genom att traversera Huffman trädet. Varje etta och nolla blir en väg i trädet, och när ett löv har hittats så skrivs dess nyckel ut i file2.

När tecknet EOF har lästs, så är filen av krypterad. Allt minne av allokeras och programmet är avslutat.

7 Resultat

Komprimeringen fungerar så att den kan komprimera en fil, och sedan läsa denna komprimerade fil exakt så som den var skriven.

För att veta ifall komprimeringen var tillräckligt effektiv gavs två stycken textfiler. En textfil att göra frekvensanalys på, och en annan fil att komprimera. Filen att komprimera ligger på ca 33 983 bytes, och vid en lyckad komprimering skall den ligga på ca 23 124 bytes.

Detta program komprimerar filen till ca 23 781 bytes, vilket är ca 2,8 % över den givna gränsen. Ingen specifikation gavs på hur exakt programmet behöver följa denna gräns, så det antas att programmet har klarat måttet.

8 Diskussion

Vi som implementerade denna uppgift fick en uppskattning på att uppgiften skulle ta ca 30 timmar, vilket var långt från verkligheten. Jag påstår – efter anekdotiska upplevelser – att uppgiften snarare kräver 60 timmar eller mer, och det endast för koden. Jag föreslår att i framtiden ta hänsyn till detta.

Det första stora problemet jag själv stötte på var bit hanteringen och bitset. För det första var, i min åsikt, gränsytan otydlig och dåligt kommenterad. T.ex. funktionen bitzet_size hämtar ”storleken” av bitset, men det framgår aldrig vad storleken är. Är detta hur många bitar som ligger i bitset, hur många bytes? Är det hur mycket minne som krävs för att skriva till fil? För att ta reda på detta behöver källkoden granskas.

Men detta var inte det enda problemet med bitset. Bitset hanterar inte den utökade ASCII kodningen, utan detta behöver modifieras själv. Bitset verkar även ”flippa” på bitarna så att de skrivs ut i fel ordning, och en funktion behövs för att läsa dem korrekt. Detta har jag garanterat såväl med ”klasskamrater” såväl med äldre kursare som skrev uppgiften förra året (dock kan jag inte referera till några).

Alla dessa problem ledde mig till att istället för att sitta och modifiera den givna datatypen och skriva hjälpfunktioner till den så skrev jag min egen – bitString.

Resten av problemen – vilket var många – var logik eller minnesfel som var på grund av mitt egna feltänkande.