TDDD86 – Laboration #6

30 oktober 2019

I den här uppgiften får du fokusera på binära träd och prioritetsköer för att understödja en implementation av Huffmankodning. Filerna du behöver för att komma igång kommer att finnas som `labb6.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 6 redovisning''` och en `git push`. Skicka sedan ett mail till din assistent med ämnet: `[TDDD86] Lab 6 redovisning` . Se till att filen **encoding.cpp** är med.

Huffmankodning

Huffmankodning är en algoritm konstruerad av David Huffman vid MIT år 1952 för att komprimera textuellt data så att en fil använder mindre utrymme. Denna relativt enkla komprimeringsalgoritm är så kraftfull att variationer av den fortfarande används i datornätverk, HD-TV och andra områden.

Vanlig textuell data lagras i ett standardformat med 8 bitar per tecken med en kodning som kallas ASCII som avbildar varje tecken på ett binärt heltalsvärde mellan 0–255. Id+en bakom Huffmankodning är att överge det stela 8-bitar-per-teckenkravet och använda binära koder av olika längd för olika tecken. Fördelen med att göra på det sättet är att om ett tecken förekommer ofta i en fil, som den vanliga bokstaven `'e'`, kan den ges en kortare kod (färre bitar), vilket kan korta ned filen. Avvägningen som måste göras är att vissa tecken kan behöva koder längre än 8 bitar, men detta reserveras för ovanliga tecken, så den extra kostnaden betalar sig.

```
...
1) build character frequency table
2) build encoding tree
3) build encoding map
4) encode data
5) decode data
C) compress file
D) decompress file
F) free tree memory
B) binary file viewer
T) text file viewer
S) side-by-side file comparison
Q) quit

Your choice? c
Input file name: large.txt
Output file name (Enter for large.huf):
Reading 9768 uncompressed bytes.
Compressing ...
Wrote 5921 compressed bytes.
```

Tabellen nedan jämför ASCII-värden för olika tecken med möjliga Huffmankodningar för någon engelsk text. Vanliga tecken, som blanksteg och `'e'` har korta koder, medan ovanligare tecken som `'z'` har längre kodning.

Character	ASCII value	ASCII (binary)	Huffman (binary)
' '	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	0111010
'c'	99	01100011	001100
'e'	101	01100101	1100
'z'	122	01111010	00100011010

Stegen inblandade i Huffmankodning av en given källtextfil till en komprimerad destinationsfil är:

1. Räkna frekvenser: Undersök källfilens innehåll och räkna antalet förekomster av varje tecken.
2. Bygg kodningsträd: Bygg ett binärt träd med en särskild struktur, där varje nod representerar ett tecken och dess antal förekomster i filen. En prioritetskö används för att hjälpa till med att bygga trädet.
3. Bygg kodningstabell: Traversera det binära trädet för att upptäcka den binära kodningen av varje tecken.
4. Koda data: Undersök åter källfilens innehåll och skriv för varje tecken ut den binärkodade versionen av det tecknet i destinationsfilen.

Kodning av fil, steg 1: Räkna frekvenser

Antag, till exempel, att vi har en fil kallad `example.txt` vars innehåll är: `ab ab cab`

I ursprungsfilen använder denna text 10 bytes (80 bitar) data. Den 10:de byten är en särskild "end-of-file"-byte (EOF).

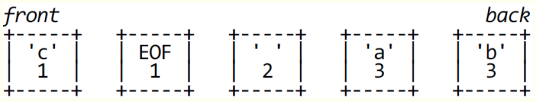
byte	1	2	3	4	5	6	7	8	9	10
char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
ASCII	97	98	32	97	98	32	99	97	98	256
binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010	N/A

I steg 1 av Huffmans algoritm beräknas frekvensen för varje tecken. Antalen representeras som en avbildning:

```
{' ':2, 'a':3, 'b':3, 'c':1, EOF:1}
```

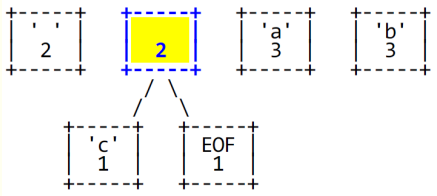
Kodning av fil, steg 2: Bygga kodningsträd

Steg 2 i Huffmans algoritm placeras våra frekvenser i binära trädnoder, där varje nod lagrar ett tecken och antalet gånger tecknet förekommer. Noderna sätts sedan in i en prioritetskö, vilken håller dem prioriterade med lägre antal som högre prioritet, så att ovanliga tecken kommer ut ur kön snabbare. (Prioritetskön är något godtycklig vid lika prioritet, som att `'c'` kommer före EOF och `'a'` före `'b'`).

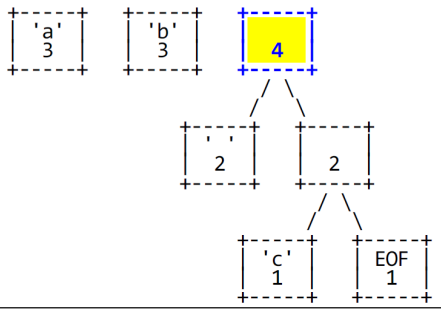


Algoritmen tar nu upprepade gånger bort de två noderna längst fram i kön (de två med lägst frekvenser) och slår samman dem till en ny nod vars frekvens är summan av de två nodernas frekvenser. De två noderna placeras som barn till den nya noden; den först borttagna hamnar som vänster barn och den andra till höger. Den nya noden sätts in i kön i sorterad ordning. Denna process upprepas till kön enbart innehåller en enda trädnod med alla andra noder som dess ättlingar. Den noden blir rot i det färdiga Huffmanträdet. Följande diagram illustrerar processen. LÄgg märke till att noderna med låga frekvenser hamnar långt ner i trädet och att noder med höga frekvenser hamnar nära trädets rot. Denna struktur kan användas för att skapa en effektiv kodning i nästa steg.

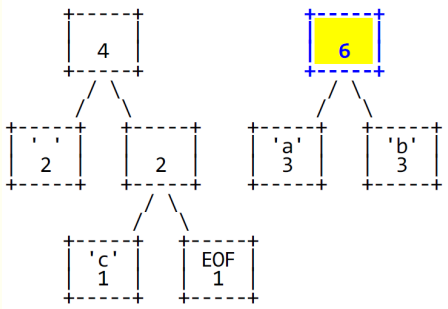
1) 'c'-nod och EOF-nod tas bort och slås samman



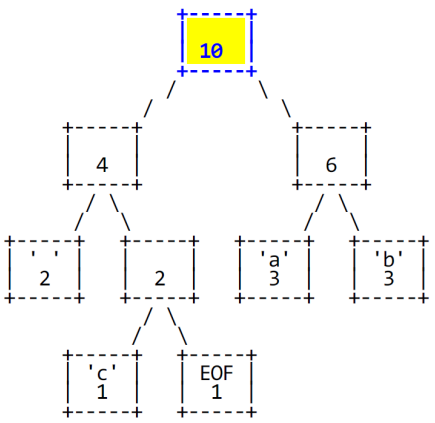
2) ' '-nod och c/EOF-nod tas bort och slås samman



3) 'a'- och 'b'-noderna tas bort och slås samman

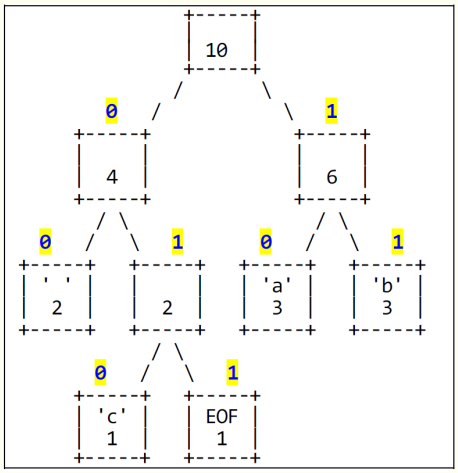


4) ' '/c/EOF-noden och a/b-noden tas bort/sammanslås



Kodning av fil, steg 3: Bygga kodningstabell

Huffmankoden för varje tecken härleds från ditt binära träd genom att tänka på vänster gren som bitvärdet 0 och varje högergren som bitvärdet 1, som i diagrammet till höger.



Koden för varje tecken kan bestämmas genom att traversera trädet. För att nå `' '` går vi vänster två gånger från rotnoden, så koden för `' '` är `00`. Koden för `'c'` är `010`, koden för EOF är `011`, koden för `'a'` är `10` och koden för `'b'` är `11`. Genom att traversera trädet kan vi producera en avbildning från tecken till deras binära representationer. Trots att de binära representationerna är heltal, eftersom de består av binära siffror och kan ha godtycklig längd, kommer vi att lagra dem som strängar. För det här trädet blir det:

```
{' ':"00",'a':"10", 'b':"11", 'c':"010", EOF:"011"}
```

Kodning av fil, steg 4: Koda data

Genom att använda kodningstabellen kan vi koda filens text till en kortare binär representation. Med föregående tabell skulle texten `"ab ab cab"` kodas som:

```
10110010110001010011011
```

Följande tabell visar `char`-till-binärkodningen i mer detalj. Totalt behöver det kodade innehållet 22 bitar, eller nästa 3 bytes, jämfört med originalfilens 10 bytes.

char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
binary	10	11	00	10	11	00	010	10	11	011

Eftersom teckenkoderna har olika längd händer det ofta att längden av en Huffmankodad fil inte är en exakt multipel av 8 bitar. Filer lagras som en sekvens av hela bytes, så i sådana här fall fylls de kvarvarande siffrorna i den sista byten med nollor. Du behöver inte bekymra dig om detta, det är en del av det underliggande systemet.

byte	1	2	3
char	a b a	b c a	b EOF
binary	10 11 00 10	11 00 010 1	0 11 011 00

Du kanske oroas av faktumet att tecknen lagras utan något som skiljer dem åt, eftersom koderna kan ha olika längd och tecken kan korsa gränser mellan bytes, som med `'a'` i slutet av den andra byten. Detta kommer dock inte att orsaka problem med att avkoda filen då Huffmankodning per definition har en användbar *prefix-egenskap* där inget teckens kodning någonsin kan förekomma som inledningen av ett annat teckens kodning.

Avkoda en fil

Du kan använda Huffmanträdet för att avkoda text som förut kodats med dess binära mönster. Avkodningsalgoritmen är att läsa varje bit från filen, en i taget, och använda denna bit för att traversera Huffmanträdet. Om biten är 0 går du till vänster i trädet. Om biten är 1 går du till höger. Du gör så till du stöter på en lövnod. Lövnoder representerar tecken, så när du väl når en lövnod matar du ut det tecknet. Antag, till exempel, att vi får samma kodningsträd som ovan och ombeds avkoda filen innehållande följande bitar:

```
1110010001001010011
```

Genom att använda Huffmanträdet går vi från roten till vi hittar tecknen, matar ut dem och återvänder till roten.

1. Vi läser `1` (höger), sedan `1` (höger). Vi når `'b'` och matar ut b. Åter till roten.
2. Vi läser `1` (höger), sedan `0` (vänster). Vi når `'a'` och matar ut a. Åter till roten.
3. Vi läser `0` (vänster), sedan `1` (höger), sedan `0` (vänster). Vi når `'c'` och matar ut c. Åter till roten.
4. Vi läser `0` (vänster), sedan `0` (vänster). Vi når `' '` och matar ut ett blanksteg. Åter till roten.
5. Vi läser `1` (höger), sedan `0` (vänster). Vi når `'a'` och matar ut a. Åter till roten.

6. Vi läser `0` (vänster), sedan `1` (höger), sedan `0` (vänster). Vi når `'c'` och matar ut c. Åter till roten.
7. Vi läser `1` (höger), sedan `0` (vänster). Vi når `'a'` och matar ut a. Åter till roten.
8. Vi läser `0`, `1`, `1`. Detta är vårt EOF-kodningsmönster, så vi stannar. Den avkodade texten är `"bac aca"`.

Tillhandahållen kod

Vi tillhandahåller en fil `HuffmanNode.h` som deklarerar användbar stödfunktionalitet, inklusive `HuffmanNode`-strukturen som representerar en nod i Huffmankodningsträdet.

```
struct HuffmanNode {
   int character;      // character being represented by this node
   int count;          // number of occurrences of that character
   HuffmanNode* zero;  // 0 (left) subtree (nullptr when empty)
   HuffmanNode* one;   // 1 (right) subtree (nullptr when empty)
   ...
};
```

Datamedlemmen `charcter` har typen `int`, men du kan tänka på den som en `char`. (Typerna `char` och `int` går i stor utsträckning att blanda i C++, men genom att använda `int` här kan vi ibland använda värden på `character` utanför det normala området för `char`, för speciella flaggor.) `character`-medlemmen lagrar tre typer av värden:

1. ett faktiskt `char`-värde;
2. konstanten `PSEUDO_EOF`, vilken representerar pseudo-EOF-värdet du behöver placera i slutet av en kodad bitström; eller
3. konstanten `NOT_A_CHAR`, vilken representerar något som egentligen inte är ett tecken. (Detta kan lagras i förgreningsnoder i Huffmanträdet som har barn, eftersom sådana noder inte representerar några enskilda tecken.)

Bitvis I/O-strömmar

I delar av det här programmet kommer du att behöva läsa och skriva enskilda bitar till filer. I tidigare laborationer har vi läst indata en hel rad eller ett ord i taget, men i det här programmet är det mycket bättre att läsa ett tecken (en byte) i taget. Så du bör använda följande in/ut-strömfunktioner:

`ostream` (output stream) member	Description
`void put(int byte)`	writes a single *byte* (character, 8 bits) to the output stream

`istream` (input stream) member	Description
`int get()`	reads a single *byte* (character, 8 bits) from input; -1 at EOF

Det kan också hända att du vill läsa en indataström och sedan "spola tillbaka" den till början och starta om läsningen igen. För att göra det på en indataströmvariabel med namnet `input` kan du skriva:

```
input.clear();              // removes any current eof/failure flags
input.seekg(0, ios::beg);   // tells the stream to seek back to the beginning
```

Vid läsning eller skrivning av en komprimerad fil är till och med en hel byte för mycket; du vill istället läsa och skriva binärt data en enskild *bit* i taget, vilket inte stöds direkt av de vanliga in/ut-strömmarna. I den här uppgiften tillhandahåller vi därför mer klasserna `obitstream` och `ibitstream` med medlemmar `writeBit` och `readBit` för att göra tillvaron lite enklare.

`obitstream` (bit output stream) member	Description
`void writeBit(int bit)`	writes a single *bit* (0 or 1) to the output stream

`ibitstream` (bit input stream) member	Description
`int readBit()`	reads a single *bit* (0 or 1) from input; -1 at end of file

Vid läsning från en bitindataström (`ibitstream`) kan du detektera filslut genom att antingen titta efter resultatet `-1` från `readBit`, eller genom att anropa `fail()`-funktionen på indataströmmen *efter* att ha försökt läsa

från den, vilket returnerar `true` om det sista `readBit`-anropet misslyckades på grund av att slutet av filen nåtts.

Lägg märke till att bitströmmarna också tillhandahåller samma medlemmar som `ostream`- och `istream`-klasserna från STL, som `getlinte`, `<<`, `>>`, etc. Vanligtvis vill du dock inte utnyttja dessa då de opererar på en hel byte (8 bitar) åt gången, eller mer; medan du vill behandla dessa strömmar en bit i taget.

Implementationsdetaljer

I den här uppgiften ska du skriva följande funktioner i filen **encoding.cpp** för att koda och avkoda data med hjälp av Huffmans algoritm som beskriven ovan. Vårt tillhandahållna main-program låter dig testa varje funktion en i taget innan du går vidare till nästa. Du måste skriva följande funktioner; du kan lägga till fler funktioner om du vill, särskilt sådana som hjälper dig att implementera rekursiva algoritmer. Alla medlemmar som traverserar ett binärt träd från rot till löv ska implementera traverseringen rekursivt närhelst det är praktiskt genomförbart.

- `map<int, int> buildFrequencyTable(istream& input)`
  Detta är steg 1 i kodningsprocessen. I den här funktionen ska du läsa indata från en given `istream` (vilken kan var antingen en fil på disk, en strängbuffer, etc.). Di ska räkna och returnera en avbildning från varje tecken (representerat som `int` här) till antalet gånger det tecknet förekommer i filen. Du ska också lägga till en enstaka förekomst av låtsastecknet `PSEUDO_EOF` i din avbildning. Du kan anta att indatafilen existerar och är läsbar, men filen kan vara tom. En tom fil skulle få dig att returnera en avbildning innehållande den enda förekomsten av `PSEUDO_EOF`.
- `HuffmanNode* buildEncodingTree(map<int, int> freqTable)`
  Detta är steg 2 i kodningsprocessen. I den här funktionen tar du in en frekvenstabell (som den du byggt i `buildFrequencyTable`) och använder den för att skapa ett Huffmankodningsträd baserat på dess frekvenser. Returnera en pekare till noden som representerar roten i trädet.
  Du kan anta att frekvenstabellen är giltig: att den inte innehåller andra nycklar än `char`-värden, `PSUDO_EOF` och `NOT_A_CHAR`; alla antal är positiva heltal; den innehåller minst ett nyckel/värde-par; etc.
  När du bygger kodningsträdet behöver du använda en prioritetskö för att hålla reda på vilka noder som står på tur att behandlas. Använd `std::priority_queue` från STL. Klassen `HuffmanNode` har en överlagrad jämförelseoperator som ger rätt beteende i kombination med prioritetskön i STL.
- `map<int, string> buildEncodingMap(HuffmanNode* encodingTree)`
  Detta är steg 3 i kodningsprocessen. I den här funktionen tar du in en pekare till rotnoden i ett Huffmanträd (som det du byggde i `buildEncodingTree` och använder det för att skapa och returnera en kodningstabell baserat på trädets struktur. Varje nyckel i tabellen är ett tecken och varje värde är den binära kodningen för det tecknet representerat som en sträng. Om till exempel tecknet `'a'` har det binära värdet `10` och `'b'` är `11` skulle tabellen lagra nyckel/värde-par `'a':"10"` och `'a':"11"`. Returnera en tom `map` om kodningsträdet är `nullptr`.
- `void encodeData(istream& input, const map<int, string>& encodingMap, obitstream& output)`
  Detta är steg 4 i kodningsprocessen. I den här funktionen läser du in ett tecken i taget från en given indatafil, använder kodningstabellen för att koda varje tecken till binärt format och skriver sedan tecknets kodade binära bitar till utdatabitströmmen. Efter att filens innehåll skrivits ska du skriva en enstaka förekomst av kodningen för `PSUDO_EOF` till utdatat så att du kan identifiera slutet på datat när du packar upp filen senare. Du kan anta att parametrarna är giltiga: att kodningstabellen är giltig och innehåller all information som behövs, att indataströmmen är läsbar och att utdataströmmen är skrivbar. Strömmarna är redan öppna och klara för läsning/skrivning; du behöver inte bekymra dig om att be användaren eller öppna/stänga filerna själv.
- `void decodeData(ibitstream& input, HuffmanNode* encodingTree, ostream& output)`
  Detta är "'avkoda en fil"-steget beskrivet ovan. I den här funktionen ska du göra motsatsen till `encodeFile`; du läser bitar från den givna indatafilen en i taget och går genom kodningsträdet för att skriva det ursprungliga okomprimerade innehållet av filen till den givna utdataströmmen. Strömmarna är redan öppnade och du behöver inte be användaren eller öppna/stänga filerna själv.

För att manuellt verifiera att dina implementationer av `encodeFile` och `decodeFile` fungerar korrekt kan du använda vår testkod för att komprimera valfria strängar till sekvenser av nollor och ettor. Nästa sida beskriver en header som du ska lägga till i komprimerade filer, men i `encodeFile` och `decodeFile` ska du inte skirva eller läsa denna header från från filen. Använd istället kodningsträdet du får som parameter. Oroa dig för headrar endast i `compress/decompress`.

Funktionerna på föregående sida implementerar Huffmans algoritm, men de har en stor brist. Avkodningsfunktionen behöver kodningsträdet som parameter. Utan kodningsträdet känner du inte till avbildningen från bitmönster till tecken och kan därför inte avkoda filen.

Vi kommer runt problemet genom att skriva kodningen i den komprimerade filen, som en *header*. Idén är att när vi öppnar vår komprimerade fil vid ett senare tillfälle kommer de filens först bytes lagra vår kodningsinformation, direkt följda av de komprimerade binära bitarna vi genererade tidigare. Det är faktiskt enklare att lagra teckenfrekvenstabellen, avbildningen från steg 1 i processen, och vi kan återskapa kodningsträdet från denna. I vårt `ab ab cab`-exempel lagrar frekvenstabellen följande värden (nycklarna visas som ASCII-värden):

```
{32:2, 97:3, 98:3, 99:1, 256:1}
```

Vi behöver inte skriva kodningsheadern bit-för-bit; skriv helt enkelt vanliga ASCII-tecken för våra koder. Allt du behöver göra för headern är att skriva din `map` till bitutdataströmmen innan du börjar skriva bitarna i den komprimerade filen och läsa in samma `map` först när du packar upp den. Hela filen är nu på 34 bytes; 31 för headern och 3 för det binära komprimerade datat. Här är ett försök till ett diagram:

byte	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	'{'	'3'	'2'	':'	'2'	','	' '	'9'	'7'	':'	'3'	','	' '	'9'	'8'	':'	'3'
byte	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
	','	' '	'9'	'9'	':'	'1'	','	' '	'2'	'5'	'6'	':'	'1'	'}'	*	*	*

```
           _______/  ___/   /
          /         /      /
   10110010  11000101  01101100
```

Nu tittar du kanske på den nya versionen av filen och tänker, "Den där filen är ju inte komprimerad alls; den blev faktiskt *större* än den var förut!. Den gick från 9 bytes (`"ab ab cab"`) till 34!" Detta är sant för det här konstruerade exemplet. Men för en större fil kommer inte kostnaden för headern att vara så dålig relativt hela filstorleken.

Det sista steget är att klistra samman all din kod tillsammans med kod för att läsa och skriva kodningstabellen till filen:

- `void compress(istream& input, obitstream& output)`
  Detta är den övergripande komprimeringsfunktionen. I den här funktionen ska du komprimera den givna indatafilen på den givna utdatafilen. Du får som parametrar in indatafilen som ska kodas och en utbitström till vilken komprimerade bitar av den indatafilen ska skrivas. Du ska läsa indatafilen ett tecken i taget, bygga upp en kodning av dess innehåll och skriva en komprimerad version av den indatafilen, inklusive en header, till den specificerade utdatafilen. Funktionen ska byggas ovanpå övriga kodningsfunktioner och anropa dem vid behov. Du kan anta att strömmarna är både giltiga och läs/skrivbara, men indatafilen kan vara tom. Strömmarna är redan öppna och redo för läsning/skrivning: du behöver inte be användaren eller själv öppna/stänga filerna.
- `void decompress(ibitstrem& input, ostream& output)`
  Den här funktionen ska göra motsatsen till vad `compress` gör; den ska läsa bitar från den givna indatafilen en i taget, inklusive din header som ligger i början av filen, för att kunna skriva ursprungsinnehållet av den filen till filen specificerad av `output`-parametern. Di kan anta att strömmarna är giltiga och läs/skrivbara. Strömmarna är redan öppna och redo för läsning/skrivning: du behöver inte be användaren eller själv öppna/stänga filerna.
- `void freeTree(HuffmanNode* node)`
  Den här funktionen ska frigöra allt minne associerat med trädet vars rotnod representeras av den givna pekaren. Du måste frigöra rotnoden och alla noder i dess delträd. Ingenting ska hända om parametern är `nullptr`. Om din `compress`- eller `decompress`-funktion skapar ett Huffmanträd ska den funktionen också frigöra trädet.

Tips:

- När du skriver bitmönster till den komprimerade filen, skriv inte ASCII-tecknen `'0'` och `'1'` (det skulle inte bli så bra komprimering då!), istället skrivs *bitarna* i komprimerad form en och en med medlemsfunktionerna `readBit` och `writeBit` i `bitstream`-objekten.
- Arbeta steg för steg. Få varje enskild del av kodningsprogrammet att fungera innan du börjar arbeta på nästa. Det går att testa varje enskild funktion separat med vårt klientprogram.
- Börja med små testfiler (två tecken, tio tecken, en mening) för att öva innan du försöker komprimera stora böcker med text. Vilken sorts filer tänker du att Huffman kommer att vara särskilt effektivt på? På

vilken sort är algoritmen mindre effektiv? Finns det filer som växer istället för att krympa med Huffman-kodning?
- Din implementation ska vara robust nog att kunna komprimera filer av godtycklig typ: text, binär, bild eller till och med en som komprimerats tidigare. Ditt program kommer antagligen inte att kunna pressa samman en redan komprimerad fil ytterligare (den kan faktiskt bli större på grund av headern), men det ska vara möjligt att komprimera i flera iterationer och sedan återskapa ursprungsfilen.
- Ditt program behöver endast packa upp giltiga filer komprimerade av ditt eget program. Du behöver inte skydda dig mot användarfel som att försöka packa upp en fil som inte är i korrekt komprimerad format.
- Operationerna för att läsa och skriva bitar är något ineffektiva och att arbeta med stora filer (100KB eller mer) tar en del tid. Bli inte oroad om läs/skriv-fasen är **långsam** för väldigt stora filer.
- Lägg märke till att Qt Creator placerar komprimerade binära filer skapade av din kod i din "build"-katalog. De kommer inte att synas i "res"-katalogen i projektet.

Möjliga utökningar

E7 — bättre kodningstabell (3 poäng):
Vår implementation av kodningstabellen i början av varje komprimerad fil är inte alls effektiv och för små filer kan den ta upp relativt mycket plats. Försök att se om du kan hitta på ett bättre sätt att koda kodningstabellen. Du kan läsa på om *succinct data structures* och se ifall du kan skriva kodningsträdet med en bit per nod och en byte per tecken. Till exempel, du kan observera att dina kodningsträden är fulla (varje nod har null eller två barn) och implementera en sekventiel träd representation[1] för ditt kodningsträd.

Skapa en ny branch som heter E7. Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 E7 redovisning''` och en `git push`. Skicka sedan ett mail till din assistent med ämnet: `[TDDD86] E7 redovisning`.

[1] se `https://www.ida.liu.se/opendsa/Books/TDDD86F19/html/SequentialRep.html`