1 A projekt felépítése és egyéb információ

A main.c forráskód tartalmazza a menü vezérlését. Meghatározza az inputok segítségével, hogy melyik műveleteket szeretne a felhasználó végrehajtani. Az Input.c forráskóddal kezdődnek a dekódolás és enkódolás eljárások. Beolvassa az input fájlból a szöveget alkalmas formátumban—karakterenként vagy stringbe. A beolvasás után, a HuffmanCompression.c és HuffmanDecompression.c forráskódok végzik el a munkát. A HuffmanCompression.c forráskód felhsználja a HashMap.c adatszerkezetet és függvényeit. A HuffmanTree.c forráskódot kódolás és visszaállítás esetén is alkalmas—a HuffmanCompression.c és HuffmanDecompression.c egyaránt felhasználja.

A program működéséhez nem szükséges külső könyvtár szabványos könyvtárokon kívül. A debugmalloc.h forráskódot viszont használja, szóval műszáj hozzáadni a fájlt a projekthez.

2 Adatszerkezetek

A program működéséhez négy alapvető adatszerkezet fontos. Mivel a Huffman kódoló algoritmus egy fa bejárásán alapul, muszáj egy bináris fához hasonló adatszerkezetet felépíteni. Ez a fa szerkezet a HuffmanTree nevű fájlban található. A HuffmanTree struct tartalmaz két egész számot és egy jobb és egy bal HuffmanTree gyereket.

```
typedef struct HuffmanTree {

    int key;
    char symbol;

    int height;

    struct HuffmanTree* leftElementChild;
    struct HuffmanTree* rightElementChild;

    struct HuffmanTree* nextItem;
    struct HuffmanTree* previousItem;

} HuffmanTree;
```

A Huffman kódoló algoritmus egyik fő tulajdonsága a log(n) bejárása. Ha a bejövő bit az input stringben 1, az algoritmus bejárja a jobb gyereket. Egyébként bejárja a bal gyereket. A levél elemek bejárásakor kiíródik a felhasználó által megadott output fájlban a meglévő karakter. A HuffmanTree adatszerkezet duplán láncolt listaként is működik. Ezért van az adataszerkezetnek egy nextItem és previousItem változója. A fa felépítése egyszerűsítéséhez a két adatszerkezet egy struct-ba van téve. Egy adott karakter pozíciója a fában attól függ, hogy milyen gyakran jelenik meg az input szövegben. Kevésbé gyakori betűk mélyebben helyezkednek el a fában. A duplán láncolt lista eleje és vége a Queue nevű struct-ban létezik.

```
typedef struct Queue {

    struct HuffmanTree* lastItem;
    struct HuffmanTree* firstItem;

} Queue;
```

A Queue.c modul a bejövő karaktereket rendezett sorrendbe teszi a frekvenciájuk szerint. A Queue struct-nak két HuffmanTree pointer-e van: az egyik a duplán láncolt lista elejére mutat, a másik pedig a végére. A Queue beszúrása legfeljebb O(n) időben fut, viszont az elvevés a Queue-ból (PopMin) O(1) konstans időben fut. A szöveg feldolgozását egy HashMap adatszerkezet használatával egyszerűsíthető.

```
typedef struct HashMap {

    int* hashMap;

} hMap;
```

Minden ASCII karakter rendelkezik egy decimális számmal 0 és 255 között. A HashMap adatszerkezetnek van egy 255 nagyságú tömbbe. Egy input karakter beolvasásakor O(1) konstans időben megnövelhető a gyakorisága a tömbben. Az utolsó adatszerkezet egybe tárolja egy adott karakternek a bit kódját.

```
typedef struct EncodedElement {

    char* bitsequence;
    char symbol;

} EncodedElement;
```

Ezért van egy char* változó a bit kódnak és egy char változó maga a karakternek. Így könnyen lehet kiolvasni a kódolandó szövegből az output kódolt szöveget.

3 A program működését vezérlő függvények

3.1 main.c

```
void Encoding();
```

Beolvassa a kódolandó fájlt, az output fájlt és az üres dekódoló tábla fájlt, és meghívja a kódoláshoz megfelelő függvényeket. A kódolás vezérlésére alkalmas.

```
void Decoding();
```

Beolvassa a kódolt fájlt az output és dekódoló tábla fájllal együtt, és meghívja a dekódoló tábla feldolgozó függvényt. A dekódoló tábla feldolgozása eredményével dekódolja a kódolt input szöveget a megfelelő függvények meghívásával. A dekódolás vezérlésére alkalmas.

```
void FreeTreeMemory(HuffmanTree* currentElement);
```

Felszabadítja a HuffmanTree adatszerkezetben lévő memóriát. Paraméterként kapja a fa gyökeret és postorder bejárással felszabadítja a memóriát.

```
void FreeFileNames(char* inputFile, char* outputFile, char* decodeTableFile);
```

Felszabadítja a paraméterként kapott string-eket.

```
void CloseFiles(FILE* inputFile, FILE* outputFile, FILE* decodeFile);
```

Bezárja a paraméterékent kapott fájlokat.

3.2 Input.c

```
char* GetStringFromInput(char* inputString);
```

Beolvassa a standard inputról a felhasználó által begépelt fájl nevét. Referenciaként kapja meg az üres input string-et, és visszatér a beolvasott string-gel. Ha a dinamikus memória lefoglalása sikertelen, visszatér NULL pointer-rel.

```
hMap* ReadFromFile_encode(FILE* inputFile, hMap* hashMap, int* hashMapSize,
        char** inputString);
```

Beolvassa a felhasználó által megadott input fájlban tárolt szöveget, és a karakterek gyakoriságát eltárolja egy dinamikusan lefoglalt HashMap-ben. Paraméterként kapja az input fájlt amiből kiolvassa a szöveget, egy HashMap pointer-t amit majd módosít, egy egész számot referenciaként és egy string-et szintén referenciaként.

```
unsigned char* ReadFromFile_decode(FILE* inputFile, unsigned char* charArray,
        int* length);
```

Beolvassa a felhasználó által megadott dekódolt szöveget tároló fájlt. Lefoglal memóriát a paraméterként kapott char tömbnek, és karakterenként megnyújtja a tömböt és hozzáadja a karaktert. A visszatérési értéke a módosított tömb.

3.3 HashMap.c

```
hMap* InitializeHashMap(hMap* hashMap);
```

Lefoglalja a hashMap változónak és a hashMap adatszerkezetben lévő tömbnek a megfelelő memoriát, és inicializálja a tömbnek a celláit nullára. A visszatérési értéke a módosított hMap.

```
bool IsInMap(hMap* hashMap, char key);
```

A paraméterként adott hMap-ban megnézi, hogy benne van-e a szintén paraméterként kapott karakter. Visszatérési érték true, ha benne van, egyébként false.

```
hMap* AddToElement(hMap* hashMap, char key);
```

Hozzáad a paraméterként kapott HashMap megfelelő karakter cellához 1-et és visszatér a módosított HashMap-pel.

```
void FreeHashMapMemory(hMap* hashMap);
```

Felszabadítja a paraméterként kapott HashMap memóriát.

```
void PrintHashMapItems(hMap* hashMap);
```

A standard output-ra kinyomtatja a paraméterként kapott HashMap karaktereinek gyakoriságát a karakterrel együtt.

3.4 Queue.c

```
void PrintQueueElements(Queue* queue);
```

A standard output-ra kinyomtatja a paraméterként kapott Queue elemeit növekvő sorrendbe.

```
bool InitializeQueue(Queue** queue);
```

Inicializál egy új Queue változót. A paraméterként kapott Queue double pointer, hogy a Queue pointer mutasson a korábban lefoglalt memóriaterületre. A visszatérési értéke true, ha a memória lefoglalása sikeres, egyébként false.

```
bool AddNewElement(Queue** queue, int key, char symbol);
```

A paraméterként kapott Queue-hoz hozzáad egy új dinamikusan foglalt HuffmanTree-t, melynek változói szintén paraméterként kapott. A visszatérési érték true, ha a HuffmanTree lefoglalás sikeres, egyébként false. A Queue módosítása megmarad a függvényen kívül, hiszen a Queue pointer referenciaként (double pointer) adott.

```
void RestoreMin(Queue** queue);
```

Egy új elem beszúrásakor ez a függvény visszaállítja a Queue minimális tulajdonságát, illetve megtalálja az új HuffmanTree helyét nagyságrendileg a Queue-ban. A Queue módosítása megmarad a függvényen kívül, hiszen a Queue pointer referenciaként (double pointer) adott.

```
HuffmanTree* PopMin(Queue** queue, bool* isEmpty);
```

A paraméterként kapott Queue-ból kiveszi a felső elemet O(1) idő alatt. A referenciaként kapott bool változót true-ba állítja, ha a Queue-ban egy HuffmanTree elem maradt, illetve a firstItem és lastItem pointer ugyanarra a memóriacímre mutat. A visszatérési értéke egy pointer a kivett HuffmanTree memória helyére.

```
void AddToLinkedList(Queue** queue, HuffmanTree* newElement);
```

Ez a függvény lényegében abban különbözik a AddNewElement függvénytől, hogy egy már létező HuffmanTree-t beszúr a Queue-ba. A Queue módosítása megmarad a függvényen kívül, hiszen a Queue pointer referenciaként (double pointer) adott. Nincs visszatérési értéke, mert nincs olyan eset, ahol ez a művelet nem végezhető el.

3.5 HuffmanTree.c

```
bool BuildHuffmanTree(Queue** queue, bool* isEmpty);
```

Felépíti a paraméterként kapott Queue-ban lévő elemekből az egyesített Huffman fát amiből ki lehet olvasni az egyes kódokat. Egy szintén paraméterként kapott boolean változót is kap amit true-ba állít ha az összes elemet, kivéve a gyökér elemet, beépítette az egyesített fába. A Queue módosítása megmarad a függvényen kívül, hiszen a Queue pointer referenciaként (double pointer) adott.

```
int ConnectTrees(HuffmanTree* lChild, HuffmanTree* rChild, HuffmanTree** newNode);
```

A korábban említett Huffman fa egyesítéshez szolgáló függvény. Két Queue-beli HuffmanTree elemből egy új HuffmanTree elemet felépít. Az új elem bal gyereke megegyezik a paraméterként kapott lChild HuffmanTree-val, illetve a jobb gyereke a szintén paraméterként kapott rChild HuffmanTree-val. Az új elem referenciaként adott a függvényben való módosítások megmaradására. A visszatérési érték true, ha a memória lefoglalása sikeres, egyébként false.

3.6 HuffmanCompression.c

```
hMap* FillHashMap(hMap* hashMap, unsigned char*input, int* hashMapSize);
```

A paraméterként kapott HashMap-ba letárolja a szintén paraméterként kapott unsigned char* változóban lévő karakterek gyakoriságát. Minden karakter önmagában egy index a HashMap adatszerkezet egész számok tömbjében. A visszatérési érték a módosított HashMap.

```
Queue* HuffScan(unsigned char* input, hMap* hashMap);
```

A paraméterként kapott input szöveg karaktereit a szintén paraméterként kapott HashMap-ban lévő gyakoriságukkal együtt beleszúrja egy újonnan inicializált Queue-ba. A visszatérési értéke az újonnan inicializált Queue vagy NULL, ha a memória lefoglalása sikertelen.

```
void TraverseTree(EncodedElement*** elements, int* index, HuffmanTree* currentElement,
    unsigned char* bitsequence, int height, FILE* decodeTable);
```

Ez egy rekurzív függvény, amely bejárja a paraméterként kapott HuffmanTree-t. Mielőtt bejárja a bal és jobb gyerekeket, megnöveli az eddig felépített bit kódot eggyel vagy nullával. A bal gyerek TraverseTree rekurzív meghívásakor a nullával növelt bit kódot adja tovább, illetve a jobb gyerek meghívásakor az eggyel növelt bit kódot. Ha a currentElement paraméter gyerekei NULL pointer-ek, a referenciaként kapott EncodedElements* tömbbe eltárolja a levél elem karaktert és a végleges bit kódot és beírja a dekódoló tábla fájlba.

```
unsigned char* EncodeInput(EncodedElement** elements, unsigned char* input,
    int queueSize, FILE* decodeTable);
```

A felhasználó által begépelt input szöveghez hozzárendeli a végleges kódot. Az egyes karaktereket megtalálja a paraméterként kapott EncodedElement tömbben, és a karakter bit kódot egyesével átalakítja bitekre, amiket aztán össze tud vagy-olni egy dummy karakterrel. Miután a dummy karakter 8 bitjét betöltötte, beírja az output fájlba, és újra kezdi a dummy karakter feltöltését bitekkel. A visszatérési érték a kódolt output.

```
FILE* HuffEncode(Queue* queue, int queueSize, unsigned char* input,
    FILE* outputFile, FILE* decodeTable);
```

Ez a függvény a korábban említett összes HuffmanCompression függvényeket összeköti és vezérli, hogy a végleges kódot elő tudja állítani a program. A visszatérési értéke NULL, ha bárhol a függvényben és a függvény hívásokban volt sikertelen memória lefoglalás. Egyébként, visszatér az output fájllal.

3.7 HuffmanDecompression.c

```
HuffmanTree* RecreateHuffmanTree(HuffmanTree* huffmanTree);
```

Inicializál egy új HuffmanTree-t, amely azonos lesz a kódolás eljárás által felépített HuffmanTree-val. A visszatérési értéke NULL pointer, ha a memória lefoglalása sikertelen. Egyébként visszatér az új HuffmanTree-val.

```
HuffmanTree* HuffmanDecode(FILE* decodeFile, int* padding, HuffmanTree* huffmanTree);
```

Ez a függvény soronként beolvassa a paraméterként kapott dekódoló fájlban lévő karakter-bitkód adatpárokat. Egyéb függvények segítségével a karakter adatot elhelyezi a korábban inicializált paraméterként kapott HuffmanTree-ba a bitkód adat szerint. A visszatérési értéke NULL pointer, ha bárhol függvényben volt sikertelen memóriafoglalás vagy megnyújtás. Egyébként visszatér a módosított HuffmanTree-val.

```
HuffmanTree* ProcessCode(unsigned char* inputLine, HuffmanTree* huffmanTree);
```

A paraméterként kapott karakter-bitkód adatpárt tartalmazó input string-et lebontja a karakterre és a bitkódra. Ezt a két adatot, a szintén paraméterként kapott HuffmanTree-val együtt, továbbítja a fa beszúró függvényhez. A visszatérési értéke a módosított HuffmanTree, ha nincs sikertelen memóriafoglalás.

```
HuffmanTree* CreateNewTreeElement(HuffmanTree* newElement, unsigned char symbol);
```

Lefoglal memóriát egy új HuffmanTree elemnek. A visszatérési érték NULL pointer, ha a memória lefoglalása sikertelen. Egyébként visszatér az új elemmel.

```
HuffmanTree* AddToHuffmanTree(unsigned char* input, unsigned char inputSymbol,
        HuffmanTree* huffmanTree);
```

Lefoglal memóriaterületet egy új HuffmanTree elemnek a CreateNewTreeElement függvény segítségével. A paraméterként kapott input bitkód szerint megtalálja az új elem helyét a fában. Ha a bejövő karakter az inputban egy, a HuffmanTree pointer jobbra lép a fában, illetve a bal gyereket járja be, ha nulla. A visszatérési érték a módosított HuffmanTree, vagy ha bárhol a függvényben vagy a CreateNewTreeElement függvényben történt sikertelen memóriafoglalás, akkor NULL pointer. A CreateNewTreeElement függvényt akkor is hívja, ha egy átmeneti—nem végleges levél—elemet kell előállítani a fa bejárása közben.

```
void PrintToFile(FILE* outputFile, HuffmanTree* huffmanTree, unsigned char* codedText,
        int padding, int length);
```

Lényegében ugyanúgy működik mint a HuffmanCompression EncodeInput függvény. A kódolt fájlban lévő szöveg bitjeit beolvassa és visszaállítja az eredeti szöveget. Egy eredeti karakter előállításakor beírja az eredményt a paraméterként kapott output fájlba. Nagy esélye van annak, hogy a kódolt szöveg utolsó karakterét ki kellett egészíteni valamennyi nulla bittel. Ez a tömítés kiíródott a dekódoló tabla fájlba, és vissza van olvasva dekódoláskor. Ez a függvény felhasználja a tömítési érteket, hogy pontosan vissza tudja állítani a kódolt szöveget.