Greining reiknirita

Verkefni 2

Kvik Huffman kóðun

Hörður Freyr Yngvason

14. ágúst 2011

1 Inngangur

Segjum sem svo, að við viljum senda stafræn gögn yfir tengingu með takmarkaða bandbreidd. Við viljum vitaskuld lágmarka flutningstímann, sem jafngildir því að senda gögnin í eins fáum bitum og unnt er. M.ö.o. viljum við *þjappa* gögnunum á sem hagvkæmastan hátt.

Ein vel þekkt þjöppunaraðferð er (upprunalega) *kyrrlega Huffman-kóðunin* [1]. Þar lítum við á skrá sem streng yfir stafróf ólíku stafanna í skránni, greinum tíðni hvers stafs og þjöppum byggt á þeim upplýsingum. Farnar eru tvær umferðir gegnum skrána. Fyrri umferðin fer í tíðnigreiningu, sem er notuð til að smíða svokallað *Huffman-tré* fyrir gögnin. Huffman-tréð má nota til að búa til *prefix-kóða* stafrófsins og lesa úr þjöppuninni. Seinni umferðin fer í að þjappa gögnunum. Þessi aðferð hentar þó ekki fyrir allar gagnagerðir: Reikniritið krefst þess að sjá öll gögnin áður en nokkru er þjappað, sem gengur ekki fyrir gögn enn í smíði, t.d. beina útsendingu, þar sem næstu stafir eru óþekktir. Í því tilfelli getum við valið um að seinka útsendingunni og þjappa skránni í bútum, eða hverfa frá því að nota kyrrlegu Huffman kóðunina. Einnig þarf að senda Huffman-tréð með boðunum svo hægt sé að lesa þau.

Oft má gera betur; nefnilega, með því að fara aðeins eina umferð í gegnum skrána og þjappa jafnóðum, getum við lágmarkað biðtíma og e.t.v. þjappað gögnunum hraðar. Ein slík aðferð byggir á Huffman-kóðun, köllum hana *kvika Huffman-kóðun* [3]. Hugmyndin er að á meðan þjöppun standi geymi sendandi og viðtakandi báðir eintak af sama Huffman-trénu. Tréð sé Huffman-tré fyrir þann hluta gagnanna, sem búið er að senda. Þá er næsta staf þjappað samkvæmt kóðun þessa trés hjá sendanda og viðtakandi notar sitt tré til að endurheimta stafinn. Síðan uppfæra báðir aðilar trén sín til að endurspegla breytt ástand skilaboðanna, þ.a. báðir hafi enn sama tréð. Með þessu móti er aðeins farin ein ferð um gögnin og þurfum ekki að senda tréð sérstaklega. Meðal útfærsla á þessari hugmynd eru *reiknirit FGK* (Faller, Gallager, Knuth) [2], síðar endurbætt af *reikniriti Vitter* [4].

Í næsta hluta skoðum við þessar hugmyndir formlega, berum þær saman og útskýrum í hvaða skilningi hvor fyrir sig er *ákjósanleg*. Í hluta 3 birtum við niðurstöður samanburðar á upphaflegu kyrrlegu Huffman-kóðuninni og kviku Huffman-kóðun Vitters fyrir ýmis gögn. Öll forrit eru gefin í heild sinni aftast.

2 Kyrrleg og kvik Huffman-kóðun

Byrjum á að samræma rithátt fyrir þennan hluta Látum n vera stærð stafrófsins og a_j vera j-ta staf þess, $j = 1 \ldots, n$. Látum t vera fjölda stafa úr skilaboðunum, sem búið er að vinna með, látum $M_t := (a_{i_j})_{j=1}^{j=t}$ vera runu þeirra og k vera fjölda ólíkra stafa í M_t. Fyrir öll a_j látum við svo w_j vera tíðni (þyngd) a_j í M_t og l_j vera fjarlægð laufs a_j frá rót Huffman-trés M_t. Köllum bitafjölda þjöppunar M_t *kostnað* boðanna.

2.1 Kyrrleg Huffman-kóðun

Í fyrri umferð reikniritsins úthlutum við k ólíku stökunum $a_j \in M_t$ hnút með gildið a_j og þyngd w_j; setjum hnútana í forgangsbiðröð Q þannig að léttustu stökin fái hæstan forgang. Þar til Q inniheldur aðeins eitt stak, fjarlægjum við ítrekað tvo léttustu hnútana x og y úr Q og setjum í staðinn hnút z, sem hefur x og y sem börn og samanlagaða þyngd þeirra. Þá verður eina stakið eftir í Q rót Huffman-trés fyrir M_t. Kóðunin fyrir staf a_j fæst þá með því að rekja sig frá rót niður í a_j þannig að fyrir hverja stiku bætum við 0 aftan á kóðann ef stikan er til vinstri, en 1 annars.

Þau tré, sem fengist geta á þennan hátt (fyrir eitthvert mál) kallast *Huffman-tré*. Almennt, ef við smíðum einhvern prefix-kóða fyrir stafina a_j í M_t og lítum á tilsvarandi tré, þá er stærðin $\sum_j w_j l_j$ kostnaður skilaboðanna M_t. Huffman-kóðun skilaboðanna er *ákjósanleg*, í þeim skilningi að kostnaður kóðunarinnar er lágmarkaður yfir öll prefix-kóðunartré fyrir þetta sama stafróf.

Keyrslutími útfærslunnar hér að aftan er $O(n \log(n))$, en með því að nota van Emde Boas tré á að vera hægt að ná honum niður í $O(n \log(\log(n)))$ [1].

2.2 Kvik Huffman-kóðun

Í kvikri Huffman-kóðun notum við kvik Huffman-tré, þ.e. Huffman-tréð okkar breytist og lagar sig að skilaboðunum gegnum kóðunina. $(t+1)$-ti stafurinn, $a_{i_{t+1}}$, er kóðaður út frá Huffman-trénu fyrir M_t; sendandi og viðtakandi breyta síðan báðir Huffman-trénu sínu í samræmi við viðbótina þannig að báðir enda með sama Huffman-tréð fyrir M_{t+1}. Vandinn er þá fyrst og fremst að framkvæma þetta skref skilvirkt; sér í lagi þarf að tryggja að tréð sé enn Huffman-tré. Til þess notfærum við okkur *systkinaeiginleika* Huffman-trjáa: Tvíundatré með þyngd ≥ 0 og p lauf er Huffman-tré *þ.þ.a.a.* það uppfylli eftirfarandi tvö skilyrði [3]

1. laufin p hafa þyngdir $w_1, \ldots, w_p \geq 0$ og þyngd innri hnúts er summa barna hans;

2. unnt er að númera hnútana í vaxandi röð eftir þyngd, þannig að hnútar $2j-1$ og $2j$ eru systkini, fyrir $j = 1, \ldots, p-1$, og sameiginelga foreldri þeirra er af hærra númeri.

Í þessu liggur kjarni reiknirits FGK, sem er síðan endurbættur í reikniriti Vitters: Við viljum framkvæma skrefið $H_t \mapsto H_{t+1}$, þar sem H_i er Huffman-tré M_i. Framkvæmum eitt milliskref á H_t: Byrjum með hnút, sem er laufið fyrir $a_{i_{t+1}}$. Skiptum ítrekað á innihaldi hnútsins, þar með talið er undirtré hans, og innihaldi hæst númeraða hnútnum með sömu þyngd, skoðum svo í staðinn

foreldri síðarnefnda hnútsins. Að þessu loknu verður H_t enn Huffman-tré fyrir M_t og við fáum H_{t+1} með því að hækka þyngd laufs $a_{i_{t+1}}$ og forfeðra þess um 1 (því við höfum varðveitt systkinaeiginleikann).

Athugum sérstaklega tilfellið $k < n$, þ.e. þegar við höfum enn ekki séð allt stafrófið. Þá notum við einn 0-hnút til að tákna þá $n - k$ stafi sem við eigum eftir að sjá. Ef $a_{i_{t+1}}$ er slíkur stafur, þ.e. $a_{i_{t+1}} \notin M_t$, þá kljúfum við 0-hnútinn og smíðum úr öðrum hlutanum laufið fyrir $a_{i_{t+1}}$. Kóðinn sem við sendum fyrir $a_{i_{t+1}}$ er kóðinn frá rót niður í 0-hnútinn, ásamt nokkrum aukabitum sem segja, hver $n - k$ ónotuðu stafanna var sendur.

2.2.1 Reiknirit Vitter

Reiknirit Vitters er svipað, nema hvað við númerum hnútana út frá myndframsetningu trésins, köllum það *óbeina* númeringu: Hnútarnir eru númeraðir í vaxandi röð eftir dýpt, en hnútar af sömu dýpt eru númeraðir í vaxandi röð frá vinstri til hægri. Við viðhöldum þessari númeringu með sérstakri gagnagrind sem við köllum *fleytitré* (e. floating tree). Við höfum jafnframt eftirfarandi fastayrðingu

> Látum w vera þyngd í trénu. Þá eru öll lauf af þyngd w á undan öllum innri hnútum af þyngd w með tilliti til óbeinu númeringarinnar.

Sérhvert Huffman-tré sem uppfyllir þetta skilyrði lágmarkar bæði $\sum_j l_j$ og $\max_j\{l_j\}$. Að lokum, til að einfalda útfærslu, skilgreinum við:

Skilgreining 2.1. *Kubbar* trésins (e. blocks) eru jafngildisflokkar venslanna

> $x \equiv y$ e.f.f. x og y hafa sömu þyngd og eru báðir innri hnútar eða báðir lauf.

Leiðtogi (e. leader) kubbs er sá hnútur kubbsins, sem hefur hæstu óbeinu númeringuna.

2.3 Samanburður á aðferðunum

Erfitt er að fá áþreifanlegt mat á gæðum þjöppunarreiknirits út af fyrir sig, því vel má hugsa sér sérhæfð mynstur sem hægt er að þjappa mun betur en með Huffman-legri kóðun; t.d. er segðin

> fyrstu 10^6 aukastafir π í tugakerfi

greinileg þjöppun á útlistun umræddra aukastafa á pappír. Við einskorðum okkur því hér við að meta kviku kóðunina út frá kyrrlegu kóðuninni. Táknum með S_t kostnað kyrrlegu kóðunarinnar og látum D_t standa fyrir kostnað kviku kóðunar Vitters (fyrir M_t). Nánar tiltekið er kostnaðurinn fjöldi bita, sem sendir eru, nema hvað

- í kyrrlegu kóðuninni teljum við ekki með kostnaðinn við að senda tréð,
- í kviku kóðuninni teljum við ekki aukabitana sem sendir eru, þegar við sjáum staf í fyrsta skipti.

Í grein sinni [3] sýndi Vitter fram á eftirfarandi meginskorður:

Setning 2.1. *1.* $S_t - k \leq D_t \leq S_t + t$

2. Reiknirit Vitter lágmarkar muninn $D_t - S_t$ *yfir öll* M_t *og öll (einnar umferðar) kvik Huffman-reiknirit.*

3 Niðurstöður mælinga

Mælingar aðallega gerðar á stafrófi af stærð 256.

4 Lokaorð

Taka saman lykilatriði í samanburðinum á aðferðunum.

Athuga að kvika kóðunin er mjög veik fyrir gagnatapi. Athuga að fyrir kviku kóðunina þarf stafrófið að vera þekkt fyrir kóðun! Það er frekar stór krafa.

Viðauki: Forritin sem notuð voru í mælingarnar

Huffman-kóðunin er útfærð með [1] til hliðsjónar. Keyrslutíminn ætti að vera $O(n \log n)$, en forritið er takmarkað við að gögnin komist fyrir í minninu. Reiknirit Vitters er útfært í klasanum FloatingTree og er svo gott sem Pascal→C++ þýðing á [4]. Notkunin er

```
./huffman [t]
./vitter [t]
```

þar sem `t` er fjöldi stafa í skránni (-1 og ekkert þýðir þjappa öllu); gögnin eru lesin af aðalinntaki.

src/huffman.cpp

```
#include <iostream>
#include <queue>
#include <map>
#include <vector>
#include <cstdlib>

using namespace std;

class node {
public:
    node *left, *right;
    char data; int weight;

    // build leaf
    node(char d, int f) {
        left = 0; right = 0;
        data = d; weight = f;
    }
    // build internal node
    node(node *lnode, node *rnode) {
        left = lnode; right = rnode;
        weight = lnode->weight + rnode->weight;
    }
    ~node() {
        if(left) {
            delete left;
```

```
            delete right;
        }
    }
};
// for node comparison in the priority queue
struct cmp {
    bool operator() (const node *a, const node *b) const {
        return a->weight > b->weight;
    }
};

typedef map<char,vector<bool> > encoding_table;

// Huffman tré úr tíðnitöflu
node *make_tree(map<char,int> &);
// Kóðunartafla úr Huffman-tré
void make_encoding_table(node *, encoding_table &, vector<bool> &);

int main(int argc, char *argv []) {
    int limit = (argc==1)?(-1):(atoi(argv[1])); // stafir sem lesa skal
    map<char,int> freq;
    char c;
    queue<char> input;
    while((input.size() != limit) && (cin >> noskipws >> c)) {
        input.push(c);
        freq[c]++;
    }
    // kóða strenginn
    node *htree = make_tree(freq); encoding_table enc; vector<bool> prefix;
    make_encoding_table(htree, enc, prefix);
    queue<bool> encoding;

    cout << "Alpha:␣" << freq.size() << endl;
    cout << "Chars:␣" << input.size() << endl;

    while(!input.empty()) {
        vector<bool> b = enc[input.front()]; input.pop();
        for(int i = 0; i != b.size(); i++)
            encoding.push(b[i]);
    }
    cout << "Bits:␣␣" << encoding.size() << endl;
    /*
    // afkóða strenginn
    while(!encoding.empty()) {
        node *path = htree;
        while(path->left) {
            bool b = encoding.front();
            path = (b)?(path->right):(path->left);
            encoding.pop();
        }
        cout << path->data;
    }
    */
    return 0;
}

node *make_tree(map<char,int> &freq) {
    priority_queue<node *, vector<node*>, cmp> q;
    // laufin
    for(map<char,int>::iterator it = freq.begin(); it != freq.end(); it++) {
        node *z = new node(it->first, it->second);
        q.push(z);
```

```
    }
    // innri hnútar
    while(q.size() != 1) {
        node *x = q.top(); q.pop();
        node *y = q.top(); q.pop();
        node *z = new node(x,y);
        q.push(z);
    }
    return q.top(); // rótin
}

void make_encoding_table(node *htree, encoding_table &table, vector<bool> &
     prefix) {
    if(htree->left) {
        prefix.push_back(0);
        make_encoding_table(htree->left, table, prefix);
        prefix.back() = 1;
        make_encoding_table(htree->right, table, prefix);
        prefix.pop_back();
    }
    else {
        table[htree->data] = prefix;
    }
}
```

src/vitter.cpp

```
// Reiknirit Vitters fyrir kvika Huffman-kóðun
#include <cstdlib>
#include <iostream>
#include <stack>
#include <queue>
#include <vector>

class FloatingTree {
    int n,M,R,E,Z, availBlock,
        *block, *weight, *parent, *parity, *rtChild, *first, *last, *prevBlock,
             *nextBlock;
    int *alpha,     // q -> j
        *rep;       // j -> q
public:
    // tekur stærð stafrófsins sem stika
    // svarar til 'Initialize'
    FloatingTree(int);
    void encodeAndTransmit(int, std::queue<int> &);
    int receiveAndDecode(std::queue<int> &);
    int findChild(int, int);
    void update(int);
private:
    void interchangeLeaves(int, int);
    void findNode(int k,int&q,int&lti,int&bq,int&b,int&op1,int&op2,int&nbq,int&
         par,int&bpar);
    void slideAndIncrement(int&q,int&lti,int&bq,int&b,int&op1,int&op2,int&nbq,
         int&par,int&bpar,bool&);
};

int main(int argc, char *argv []) {
    int limit = (argc==1)?(-1):(std::atoi(argv[1])); // stafir sem lesa skal
    int size = 256;
    FloatingTree encoder(size);
    FloatingTree decoder(size);
    unsigned char c;  // mikilvægt!!!
```

```
    int inp_size = 0,
        enc_size = 0;
    while((inp_size != limit) && (std::cin >> std::noskipws >> c)) {
        std::queue<int> transfer;
        encoder.encodeAndTransmit(c+1, transfer);
        encoder.update(c+1);
        inp_size++;
        enc_size += transfer.size();
        /*
        // Skrifa út
        int dec = decoder.receiveAndDecode(transfer);
        decoder.update(dec);
        std::cout << char(dec-1);
        */
    }
    std::cout << "Chars: " << inp_size << std::endl;
    std::cout << "Bits:  " << enc_size << std::endl;
    return 0;
}

FloatingTree::FloatingTree(int size) {
    n = size;
    M = 0; E = 0; R = -1; Z = 2*n - 1;
    // númerum allt frá 1
    alpha = new int[n+1]; rep = new int[n+1];
    for(int i = 1; i <= n; i++) {
        M++; R++;
        if(2*R == M) { E++; R = 0; }
        alpha[i] = i; rep[i] = i;
    }
    block = new int[Z+1]; weight = new int[Z+1]; parent = new int[Z+1];
    parity = new int[Z+1]; rtChild = new int[Z+1]; first = new int[Z+1];
    last = new int[Z+1]; prevBlock = new int[Z+1]; nextBlock = new int[Z+1];
    // initialize node n as the 0-node
    block[n] = 1; prevBlock[1] = 1; nextBlock[1] = 1; weight[1] = 0;
    first[1] = n; last[1] = n; parity[1] = 0; parent[1] = 0;
    // initialize available block list
    availBlock = 2;
    for(int i = availBlock; i <= Z-1; i++)
        nextBlock[i] = i+1;
    nextBlock[Z] = 0;
}
void FloatingTree::encodeAndTransmit(int j, std::queue<int> &out) {
    int q = rep[j];
    std::stack<bool> stack;
    if(q <= M) {
        // Encode letter of zero weight
        q--;
        int t;
        if(q < 2*R)
            t = E+1;
        else {
            q -= R;
            t = E;
        }
        /*
        // Bætir aukabitum á hlaðann til að tákna stafinn.
        // Nauðsynlegt til að geta afkóðað, en þessir bitar
        // eru ekki taldir með í mælingum
        for(int i = 0; i != t; i++) {
            stack.push(q%2);
            q /= 2;
```

```
        }
        */
        q = M;
    }
    int root = (M == n)?n:Z;
    while(q != root) {
        // Traverse up the tree
        stack.push((first[block[q]]- q + parity[block[q]])%2);
        q = parent[block[q]]- (first[block[q]]- q + 1 - parity[block[q]])/2;
    }
    // senda kóðann fyrir stafinn
    while(!stack.empty()) {
        out.push(stack.top());
        stack.pop();
    }
}

int FloatingTree::receiveAndDecode(std::queue<int> &in) {
    int q = (M==n)?n:Z; // set q to the root node
    int receive;
    while(q > n) {
        receive = in.front(); in.pop();
        q = findChild(q, receive);
    }
    if(q == M) { // decode 0-node
        q = 0;
        for(int i = 0; i != E; i++) {
            receive = in.front(); in.pop();
            q = 2*q + receive;
        }
        if(q < R) {
            receive = in.front(); in.pop();
            q = 2*q + receive;
        }
        else
            q += R;
        q++;
    }
    return alpha[q];
}

int FloatingTree::findChild(int j, int _parity) {
    int delta = 2*(first[block[j]] - j) + 1 - _parity,
        right = rtChild[block[j]],
        gap = right - last[block[right]];
    if(delta <= gap)
        return right - delta;
    else {
        delta = delta - gap - 1;
        right = first[prevBlock[block[right]]];
        gap = right - last[block[right]];
        if(delta <= gap)
            return right-delta;
        else
            return first[prevBlock[block[right]]] - delta + gap + 1;
    }
}

void FloatingTree::interchangeLeaves(int x, int y) {
    int tmp;
    rep[alpha[x]] = y; rep[alpha[y]] = x;
    tmp = alpha[x]; alpha[x] = alpha[y]; alpha[y] = tmp;
```

```
}

void FloatingTree::update(int k) {
    int q, leafToIncrement, bq, b, oldParent, oldParity, nbq, par, bpar;
    bool slide;
    findNode(k,q,leafToIncrement,bq,b,oldParent,oldParity,nbq,par,bpar);
    while(q > 0) {
        slideAndIncrement(q,leafToIncrement,bq,b,oldParent,oldParity,nbq,par,
            bpar, slide);
    }
    if(leafToIncrement != 0) {
        q = leafToIncrement;
        slideAndIncrement(q,leafToIncrement,bq,b,oldParent,oldParity,nbq,par,
            bpar, slide);
    }
}

// viðföngin eiga að vera tilvísanir í tilviksbreytur update
void FloatingTree::findNode(
        int k,
        int &q, int &leafToIncrement, int &bq, int &b,
        int &oldParent,int &oldParity, int &nbq, int &par, int&bpar)
{
    q = rep[k]; leafToIncrement = 0;
    if(q <= M) { // a zero weight becomes positive
        interchangeLeaves(q,M);
        if(R == 0) {
            R = M/2;
            if(R > 0)
                E--;
        }
        M--; R--; q = M+1; bq = block[q];
        if(M > 0) {
            // Split the 0-node into an internal node with two children.
            // The new 0-node is node M, the old 0-node is node M+1; the
            // parent of nodes M and M+1 is node M+n.
            block[M] = bq; last[bq] = M; oldParent = parent[bq];
            parent[bq] = M+n; parity[bq] = 1;
            // Create a new internal block of zero weight for node M+n-1
            b = availBlock; availBlock = nextBlock[availBlock];
            prevBlock[b] = bq; nextBlock[b] = nextBlock[bq];
            prevBlock[nextBlock[bq]] = b; nextBlock[bq] = b;
            parent[b]= oldParent; parity[b] = 0; rtChild[b] = q;
            block[M+n] = b; weight[b] = 0;
            first[b] = M+n; last[b] = M+n;
            leafToIncrement = q; q = M+n;
        }
    }
    else { // interchange q with the first node in q's block
        interchangeLeaves(q,first[block[q]]);
        q = first[block[q]];
        if( (q == M+1) && (M > 0) ) {
            leafToIncrement = q;
            q = parent[block[q]];
        }
    }
}
void FloatingTree::slideAndIncrement(
        int &q, int &leafToIncrement, int &bq, int &b, int &oldParent,
        int &oldParity, int &nbq, int &par, int &bpar, bool &slide)
{
    // q is currently the first node in its block
```

```
bq = block[q]; nbq = nextBlock[bq];
par = parent[bq]; oldParent = par; oldParity = parity[bq];
if(((q <= n) && (first[nbq] > n) && (weight[nbq] == weight[bq]))
        || ((q > n) && (first[nbq] <= n) && (weight[nbq] == weight[bq]+1)))
{ // slide q over the next block
    slide = true;
    oldParent = parent[nbq]; oldParity = parity[nbq];
    // adjust child pointers for next higher level in tree
    if(par > 0) {
        bpar = block[par];
        if(rtChild[bpar] == q)
            rtChild[bpar] = last[nbq];
        else if(rtChild[bpar] == first[nbq])
            rtChild[bpar] = q;
        else
            rtChild[bpar]++;
        if(par != Z) {
            if(block[par+1] != bpar) {
                if(rtChild[block[par+1]] == first[nbq])
                    rtChild[block[par+1]] = q;
                else if(block[rtChild[block[par+1]]] == nbq)
                    rtChild[block[par+1]]++;
            }
        }
    }
    // adjust parent pointers for block nbq
    parent[nbq] = parent[nbq]-1+parity[nbq]; parity[nbq] = 1-parity[nbq];
    nbq = nextBlock[nbq];
}
else slide = false;
if( (((q <= n) && (first[nbq] <= n)) || ((q > n) && (first[nbq] > n)))
        && (weight[nbq] == weight[bq]+1))
{ // merge q into the block of weight one higher
    block[q] = nbq; last[nbq] = q;
    if(last[bq] == q) {
        // q's old block disappears
        nextBlock[prevBlock[bq]] = nextBlock[bq];
        prevBlock[nextBlock[bq]] = prevBlock[bq];
        nextBlock[bq] = availBlock; availBlock = bq;
    }
    else {
        if(q > n)
            rtChild[bq] = findChild(q-1, 1);
        if(parity[bq] == 0)
            parent[bq]--;
        parity[bq] = 1-parity[bq];
        first[bq] = q-1;
    }
}
else if(last[bq] == q) {
    if(slide) {
        // q's block is slid forward in the block list
        prevBlock[nextBlock[bq]] = prevBlock[bq];
        nextBlock[prevBlock[bq]] = nextBlock[bq];
        prevBlock[bq] = prevBlock[nbq]; nextBlock[bq] = nbq;
        prevBlock[nbq] = bq; nextBlock[prevBlock[bq]] = bq;
        parent[bq] = oldParent; parity[bq] = oldParity;
    }
    weight[bq]++;
}
else {
    // a new block is created for q
```

```
        b = availBlock; availBlock = nextBlock[availBlock];
        block[q] = b; first[b] = q; last[b] = q;
        if(q > n) {
            rtChild[b] = rtChild[bq];
            rtChild[bq] = findChild(q-1, 1);
            if(rtChild[b] == q-1)
                parent[bq] = q;
            else if(parity[bq] == 0)
                parent[bq]--;
        }
        else if(parity[bq] == 0)
            parent[bq]--;
        first[bq] = q-1; parity[bq] = 1-parity[bq];
        // insert q's block in its proper place in the block list
        prevBlock[b] = prevBlock[nbq]; nextBlock[b] = nbq;
        prevBlock[nbq] = b; nextBlock[prevBlock[b]] = b;
        weight[b] = weight[bq]+1;
        parent[b] = oldParent; parity[b] = oldParity;
    }
    // move q one higher level in the tree
    if(q <= n)
        q = oldParent;
    else
        q = par;
}
```

Heimildir

[1] Cormen, T.H; Leiserson, C.E; Rivest, R.L; Stein, C. 2009. Introduction to Algorithms, 3rd ed. 428-435.

[2] Knuth, D.E. 1985. „Dynamic Huffman coding“. *Journal of Algorithms 6*, 2, 163-180.

[3] Vitter, J.S. 1987. „Design and Analysis of Dynamic Huffman Codes“. *J. ACM 34*, 4, 825-845.

[4] Vitter, J.S. 1989. „Algorithm 673: Dynamic Huffman Coding“. *ACM Trans. Math. Softw. 15*, 2, 158-167.