CSCI 255: Lab #11 Huffman Trees & B-Trees

Darwin Jacob Groskleg

Printed: Tuesday, February 18th, 2020

Contents

Questions

Q1. Huffman Trees

Given the following symbols and its frequency in an organization, construct the Huffman tree and generate the Huffman code for following the symbols:

Symbols	A	B	C	D	E
Frequency	0.39	0.09	0.12	0.18	0.22

Submit the Huffman tree and the Huffman code for the symbols in question 1.

Huffman Code

Result : `0.10.110.1110.1111`

See console sample for computed result.

Console Sample

```
~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab11-Huffman Trees and BTrees
make -B main.out
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o main
.o -c main.cpp
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o huff
man_tree.o -c huffman_tree.cpp
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o main
.out main.o huffman_tree.o
~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab11-Huffman Trees and BTrees
./main.out
Sorted char freqs: 0.09(B), 0.12(C), 0.18(D), 0.22(E), 0.39(A),
# Begin Huffman Coded Message ##########
0.10.110.1110.1111
# End of Message ######################
Expected:
0.10.110.1110.1111
~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab11-Huffman Trees and BTrees
```

Figure 1: Code is attached.

Huffman Tree Diagram

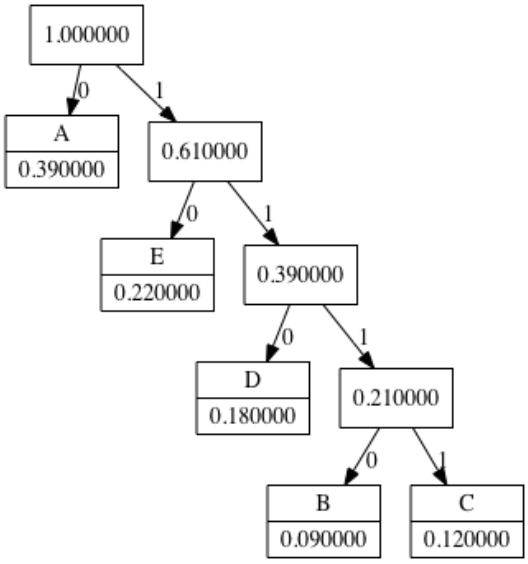

Figure 2: Resulting Tree from program, see attached code.

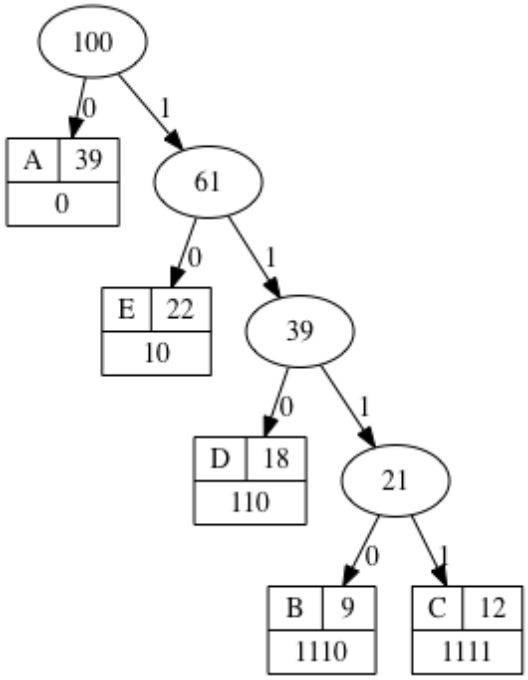

Figure 3: Expected Tree. Made on the Huffman Tree Generator website.

Q2. B-Trees

B-Tree Visualization: https://www.cs.usfca.edu/~galles/visualization/BTree.html

Insertions

Observe the process of B-tree insertion, inserting the following keys to a 5-way B-tree: 3, 7, 9, 23, 46, 1, 5, 15, 30, 24, 13, 11, 8, 19, 4, 31, 35, 60, 2, 6, 12

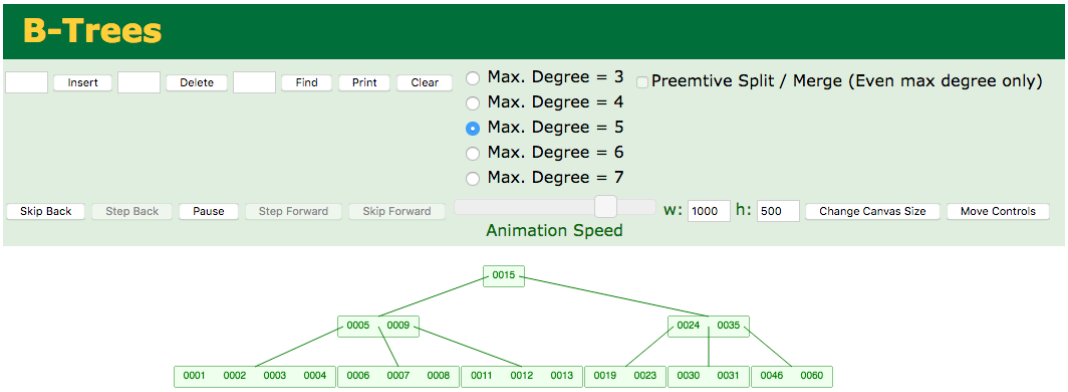

Figure 4: B-Tree Insertion Result

Deletions

Observe the process of B-tree deletion: deleting the following keys from the above tree: 4, 5, 7, 3, 15

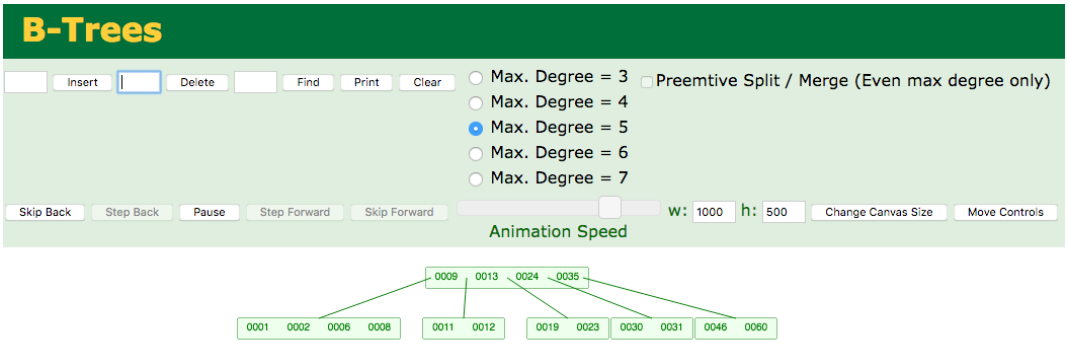

Figure 5: B-Tree Deletion Result

main.cpp

```cpp
/* main.cpp
 * --------
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, November 26, 2019
 *
 * Purpose: Submit the Huffman tree and the Huffman code for the symbols in
 *          question 1.
 */
#include <iostream>
#include <cstdio>
#include "huffman_tree.hpp"

using namespace std;

int main(int argc, char *argv[]) {
    // Construct a huffman tree
    HuffmanTree htree;
    htree.add('A', 0.39);
    htree.add('B', 0.09);
    htree.add('C', 0.12);
    htree.add('D', 0.18);
    htree.add('E', 0.22);

    // A string with the same letter ratios as above              Ratios
    string sample = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"   // 0.39
                    "BBBBBBBBB"                                // 0.09
                    "CCCCCCCCCCC"                              // 0.12
                    "DDDDDDDDDDDDDDDDD"                        // 0.18
                    "EEEEEEEEEEEEEEEEEEEE";                    // 0.22

    // generate the Huffman code for the symbols
    htree.make_decode_tree();

    clog << "# Begin Huffman Coded Message " << string(10, '#') << '\n'
         << htree.huffman_code()
         << "\n# End of Message " << string(23, '#')
         // Source:
         // http://huffman.ooz.ie/tree.dot?text=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBC
         << "\nExpected:\n0.10.110.1110.1111"
         << '\n';

    if (argc > 1 && string(argv[1]) == "--to-dot")
        cout << htree.to_dot();

    return 0;
}
```

huffman_tree.hpp

```
/* huffman_tree.hpp
 * ----------------
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, November 26, 2019
 */
#ifndef HUFFMAN_TREE_HPP_INCLUDED
#define HUFFMAN_TREE_HPP_INCLUDED

#include <list>
#include <unordered_map>
#include <string>     // default args
#include <memory>     // ptr on HFNode

using SymbolToFrequencyMap = std::unordered_map<char, double>;

// Purpose: generating variable-length binary character codes.
//
// Prefix Codes: desirable because they simplify decoding. No codeword is a
// prefix of any other, the codeword that begins an encoded file is unambiguous.
//
// So the Huffman Tree constructs an optimal prefix code called Huffman code.
// In this case it is implemented using a greedy algorithm.
//
// Usage:
//  1. Construct a instance, either empty or by passing a map of chars to their
//     respective occurence ratios.
//  2. Add new char => frequency pairs as necessary, even overwritting existing
//     values.
//  3. Generate decode tree via the state change operaton `make_decode_tree`.
//  4. Use the decode tree to get the huffman code.
class HuffmanTree {
  public:
    HuffmanTree(void) {}

    // Construct from a set of n characters.
    //  CLRS page 431
    //HuffmanTree(std::string text) {
    //    text.size();
    //}

    // Construct from a map of chars & freqs, that is a probability value [0,1]
    // of the occurence of each symbol.
    //HuffmanTree(SymbolToFrequencyMap freq_map)
    //{
    //    freq_map.size();
    //}

    ~HuffmanTree(void) {};

    // Add a single node tree to the list.
    void add(char c, double frequency);

    void make_decode_tree(void);

```

```
55      std::string decode_message(std::string code) const;
56
57      // Usage:
58      //      ht.to_dot()
59      //      ...compile then run:
60      //      ./a.out | dot -Tgif > huff.gif
61      //      open huff.gif
62      std::string to_dot() const;
63
64      // Returns a huffman coded message,
65      // a string of 1's and 0's.
66      // ...use strong types for binary string: convertible to and from string.
67      //std::string encode_message(std::string message) {
68      //}
69
70      // Return the concatenated Huffman code in use.
71      std::string huffman_code() const;
72
73    private:
74      class HFNode
75      {
76        public:
77          HFNode();
78          HFNode(char c, double fr);
79          HFNode(char c, double fr, HFNode* l, HFNode* r);
80          HFNode(HFNode&&) = default;
81         ~HFNode()         = default;
82
83          std::string huffman_code(std::string prefix="") const;
84          bool operator<(const HFNode& rhs) const;
85
86          // Readers
87          inline char   symbol()    const { return   ch; }
88          inline double frequency() const { return freq; }
89          inline auto   leftp ()    const { return  left.get(); }
90          inline auto   rightp()    const { return right.get(); }
91        private:
92          const char ch;
93          const double freq;
94          std::unique_ptr<HFNode> left;
95          std::unique_ptr<HFNode> right;
96      };
97
98      std::list<HFNode> node_list; // will destruct all the nodes
99  };
100
101
102 #endif // HUFFMAN_TREE_HPP_INCLUDED
```

huffman_tree.cpp

```cpp
/* huffman_tree.cpp
 * ----------------
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, November 26, 2019
 */
#include "huffman_tree.hpp"

#include <iostream>   // clog
#include <string>     // to_string
#include <utility>    // std::move
#include <algorithm>  // for move a range?

#include <cassert>

////////////////////////////////////////////////////////////////////////////////
// Tree Node Implementation
////////////////////////////////////////////////////////////////////////////////
HuffmanTree::HFNode::HFNode() :
    ch(0),
    freq(0),
    left(nullptr),
    right(nullptr)
{}

HuffmanTree::HFNode::HFNode(char c, double fr) :
    ch(c),
    freq(fr),
    left(nullptr),
    right(nullptr)
{}

HuffmanTree::HFNode::HFNode(char c, double fr, HFNode* l, HFNode* r):
    ch(c),
    freq(fr),
    left(l),
    right(r)
{}

// copy constructor
//HuffmanTree::HFNode::HFNode(const HFNode &obj) {
//}

std::string HuffmanTree::HFNode::huffman_code(std::string prefix) const {
    std::string hcode = "";
    // build the prefix code in the leaf
    if (ch != 0) // is leaf! Prefix is complete!
        return prefix+'.';
    if (left)
        hcode.append(left->huffman_code(prefix + '0'));
    if (right)
        hcode.append(right->huffman_code(prefix + '1'));

    return hcode;
}
```

```
55
56  bool HuffmanTree::HFNode::operator < (const HFNode& rhs) const {
57      return this->frequency() < rhs.frequency();
58  }
59
60
61  ////////////////////////////////////////////////////////////////////////////
62  // HuffmanTree Implementation
63  ////////////////////////////////////////////////////////////////////////////
64  void HuffmanTree::add(char c, double frequency) {
65      node_list.push_back({c, frequency});
66  }
67
68  // Uses a depth-first traversal?
69  std::string HuffmanTree::huffman_code() const {
70      const HFNode *np = &node_list.front();
71      std::string code = np->huffman_code("");
72      code.erase(code.end()-1);   // remove last .
73      return code;
74  }
75
76  // Repeat until only 1 node on list:
77  // - sort list of tree nodes by freqency
78  // - remove the first two tree nodes
79  // - create a new node with these trees as subtrees.
80  //      - frequency is sum of their frequencies.
81  //      - first extracted is on the left
82  // - Add the new node to the list.
83  //
84  // Prints debug data to stderr.
85  //
86  void HuffmanTree::make_decode_tree(void) {
87      node_list.sort();
88      assert(std::is_sorted(node_list.begin(), node_list.end()));
89
90      std::clog << "Sorted char freqs: ";
91      for (auto& n : node_list)
92          std::clog << n.frequency() << '(' << n.symbol() << "), ";
93      std::clog << '\n';
94
95      while (node_list.size() > 1) {
96          // Use a move constructor to avoid having multiple smart pointers to
97          // the same child node
98          HFNode *cf1 = new HFNode( std::move(node_list.front()) );
99          // popped item will have destructor called only once it goes out of
100         // scope, thus can be pointed to.
101         node_list.pop_front();
102
103         HFNode *cf2 = new HFNode( std::move(node_list.front()) );
104         node_list.pop_front();
105
106         auto freq_sum = cf1->frequency() + cf2->frequency();
107         HFNode cf3{0, freq_sum, cf1, cf2};
108         // push to front, gives precedence in sorting over others of same
109         // frequency...assuming you have a stable-sorting algorithm.
110         node_list.push_front(std::move(cf3)); //move!
```

```
111         node_list.sort();
112     }
113 }
114 
115 // assumes `make_decode_tree` was run first
116 //
117 // given a binary string: "101...",
118 // assumes code is valid message for tree
119 //  - is only 1 or 0
120 //  - is only valid prefix codes
121 //
122 std::string HuffmanTree::decode_message(std::string code) const {
123     std::string message = "";
124     const HFNode *np = &node_list.front();
125     while(!code.empty()) {
126         if (code.front() == '0')
127             np = np->leftp();
128         else
129             np = np->rightp();
130 
131         if (np->symbol() != 0) {
132             message.push_back(np->symbol());
133             np = &node_list.front();
134         }
135         code.erase(0); // pop_front
136     }
137     return message;
138 }
139 
140 std::string HuffmanTree::to_dot() const {
141     std::string dot = "digraph G {\n"
142                       "    edge [label=0]\n"
143                       "    graph [ranksep=0];\n"
144                       "    node [shape=record];\n";
145     const HFNode *np = &node_list.front();
146     std::list<const HFNode *> q; // queue
147     q.push_back(np);
148     auto to_qstring = [] (double r) {
149         return "" + std::to_string(r) + "";
150     };
151     while (q.size() > 0) {
152         np = q.front();
153         if (np->symbol() != 0) {// leaf
154             dot.append("    ");
155             dot.push_back(np->symbol());
156             dot.append(" [label=\"{");
157             dot.push_back(np->symbol());
158             dot.append("|"+to_qstring(np->frequency())+"}\"];\n");
159         }
160 
161         if (np->leftp()) {
162             q.push_back(np->leftp());
163 
164             dot.append("    ");
165             dot.append(to_qstring(np->frequency())+" -> ");
166             if (np->leftp()->symbol() != 0) // child is a leaf
```

```
167                 dot.push_back(np->leftp()->symbol());
168             else
169                 dot.append( to_qstring(np->leftp()->frequency()) );
170             dot.append(";\n");
171         }
172 
173         if (np->rightp()) {
174             q.push_back(np->rightp());
175 
176             dot.append("    ");
177             dot.append(to_qstring(np->frequency())+" -> ");
178             if (np->rightp()->symbol() != 0) // child is a leaf
179                 dot.push_back(np->rightp()->symbol());
180             else
181                 dot.append( to_qstring(np->rightp()->frequency()) );
182             dot.append(" [label=1];\n");
183         }
184 
185         q.pop_front();
186     }
187     dot.append("}");
188     return dot;
189 }
```