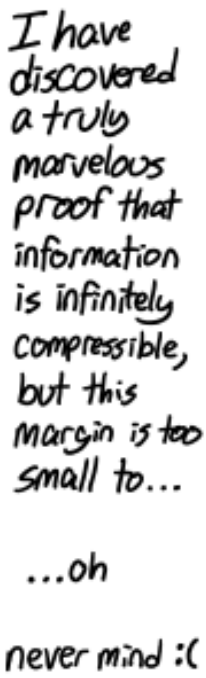
COMP 322 - Assignment 3 - Winter 2016

Due: Apr 08th, 2016 at 11:30PM

Introduction

In the third homework assignment we put heaps to work by building Huffman encoders and decoders, as used in data compression. Huffman coding if one of the great computer science topics. You will surely see it again in other courses, but you should get a taste of its implementation using C++. The two most important lessons to take away from this assignment is *code modularity* (a.k.a. "mind your own business") and *structured programming* (a.k.a. "keep it simple").

The *best* way to achieve these two tasks is to learn how to read the code you work with before you start writing your own. Yes, this last homework is more of a reading assignment. ***Attached with this document is the content of the header file, which you should have in handy when working on your code.***



Before you start working on the assignment

- Feel free to submit any solution for the 0-credit question. Do not worry about style or commenting, unless you want the instructor or the TAs to help you with it.
- Collaboration is strongly encouraged for all questions, but make sure your graded submission reflects individual work. The Discussion Board, me and the TAs are there for you to use as well. I expect many of you for my Office Hours, and, yes, I use the arrival time as the priority.
- It might happen that some topics in this homework won't be covered in class. This is normal as we only have one hour of lecture per week. You should not have trouble finding out answers in C++ online tutorials, and as always, don't forget to e-mail us or discuss with other students.
- We provide a `Makefile` and a header file `huffman.h`. Your submission should compile using the `make` command on Trottier (3rd floor) machines when run on the **original** `Makefile`, **original** `huffman.h` and your submission `huffmanStudent.cpp`. SUBMIT ONLY THE `.cpp` FILE.

Huffman Coding

Huffman coding works on a very clever principle for the problem of encoding multiple items that come from a large data set: *use longer codes for items that are rare and shorter codes for items that occur more often.* Take for example the case of encoding the bands names that come from a playlist of songs that I recently created. The playlist contains 8 songs by *The Beatles*, 4 songs by *Led Zeppelin*, 2 songs by *Queen*, and one song by *Nirvana*. Assuming that I want binary codes that are unique, I could use the following encoding, based on the above principle:

- The Beatles: 0
- Led Zeppelin: 10
- Queen: 110
- Nirvana: 111

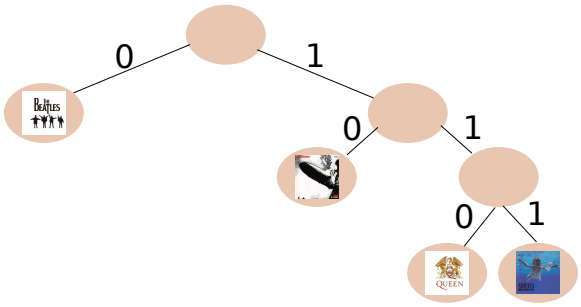


Figure 1: The Huffman tree corresponding to the example above.

One might ask the following: "Why don't we use the code 1 for Led Zeppelin, and consequently 11 for Queen and 10 for Nirvana?". We consider here the case of prefix encoding, where no code is the prefix of another. As such, I could read the following code

```
110101110010
```

and translate it to: "Queen, Led Zeppelin, Led Zeppelin, Nirvana, The Beatles, The Beatles, Led Zeppelin". No waste in delimiting codes! Note that each prefix coding scheme can be represented as a binary tree as illustrated in Figure 1. The code of each leaf if the labels on the edges corresponding to the path from the root to each particular leaf. Since only leafs represent actual encoded items, one is guaranteed that no code is the prefix of another code.

Assignment requirements

Question 0 (0 credits): Read the header file `huffman.h` to familiarize yourself with all the classes involved in this assignment. One important task you will have to do when working as programmers is to build on others' work. You might not have the luxury of building all your desired data structures and classes as you wish. Understanding what you are working with is the fist and most important step in buiding simple and reliable code.

NOTE 1: DO NOT waste your time to understand every line of code! Simply be aware of what are the attributes at your service, their accessibility (i.e. `private`, `protected`, or `public`), and only read the content of a method if you wish to clarify the way it uses your own code.

NOTE 2: Most of the methods that you are asked to implement is required to throw ***exeptions***. We will test to make sure your code throws the proper exception in each case.

Building a Huffman Tree based on frequency of items

Assume that one has a count of all items that one would like to encounter. For example, if we encode words in a dictionary, we would take the word count in a huge database of texts on a particular topic. If we encode my favorite bands, then one has access to the number of times I listened to each artist. How can we build a tree such that the rare items are at the bottom and the frequent items are at the top?!? We build it bottom to top!

- We maintain a heap where the priority of an item is its count.
- We extract the two highest priority items in the heap.
- We create a new node that has the above two items as children.
- We add back to the heap the parent of those two items with a count that is the sum of the counts of those two items.
- We continue until there is only one item in the heap. That item is our Huffman Tree!

Question 1 (0 credits): You are given a template `Heap<T>`, which is based on Assignment 2. Complete the implementation of a special Huffman Heap that works as described above. A `HuffmanHeap` extends `Heap<TreeNode *>` privately. Complete this class by implementing

```
HuffmanTree::HuffmanHeap::HuffmanHeap(istream &instr);
void HuffmanTree::HuffmanHeap::pop();
```

The constructor should initialize the Heap based on word counts. Each item in the heap should point to a `new TreeNode` which has no children. The `pop` method should remove two items from the heap and perform the actions described above.

Question 2 (30 credits): Retrieving Huffman Codes from Huffman Trees

After we have built a tree, we would like to use that tree to encode files. What are the codes corresponding to every tree? To maintain this, we implement a class `HuffmanCode` that extends `map<string,string>`, and we build such a map by performing Depth First Search(DFS) on the Huffman Tree: go down the branches of the tree until a leaf is encountered, while keeping track of the path that has been taken. When such a node is reached save the association in the `HuffmanCode`, which happens to be a `map`.

```
void HuffmanTree::dfs(HuffmanCode& hc, TreeNode *tn, string &s)
```

In the above, `hc` is the map that stores all code associations, `tn` is a pointer to the `TreeNode` that is investigated, and `s` is a string of 0s and 1s that keeps track of the path from the root to the node `tn`.

- If `tn` has a word associated with it, then this node is a leaf. Simple add the association to `hc`.
- If `tn` is not a leaf, then investigate the child associated to 0. If it is not `nullptr`, then attach a 0 to s, and recursively call the function `dfs` on the child. Do the same for the child associated to 1.
- Remember to remove the last character from `s` AFTER a recursive call on `dfs`.

Question 3 (10 credits): Retrieving Huffman Codes from a source input stream

After encoding a file, one might want to share the encoding scheme, otherwise it would be impossible to decode any encrypted file. For this purpose, assume that the input stream (whether a file or not) stores on each line two words: first a vocabulary word and second the sequence of 0s and 1s associated with this word. Implement the following constructor that builds a `HuffmanCode` from an input sequence.

```
HuffmanCode::HuffmanCode(istream &input);
```

EXCEPTION: The method should throw `0` if the stream is not in the proper format, for example if a line has more than two words, or the second word is not a sequence of 0s and 1s.

Question 4 (15 credits): Encoding an input stream to an output stream

Given a `HuffmanCode`, implement the encoding method, which takes as parameters two streams: an input stream that contains words in a dictionary, and an output stream which will contain only 0s and 1s. The method will read each word one by one and push to the output stream all corresponding codes.

```
void HuffmanEncoder::encode(istream &fin, ostream &fout) const;
```

EXCEPTION: The method should throw `1` if `fin` contains words that are not in the dictionary of code.

Question 5 (30 credits): Retrieving Huffman Trees from Huffman Codes

Given a `HuffmanCode`, implement a methods that builds a Huffman Tree. Based on all the words in the `HuffmanCode` object, and their corresponding codes, build a tree that has all the paths of 0s and 1s, and all the words at the leafs of this tree.

```
HuffmanTree::HuffmanTree(const HuffmanCode &hc);
```

EXCEPTION: The method should throw `1` if the code is not a prefix code.
EXCEPTION: The method should throw `2` if the codes are not all sequences of 0s and 1s.

Question 6 (15 credits): Decoding an encoded message

Last but not least, implement the methods that takes an input stream and decodes all the words corresponding to the sequence of 0s and 1s that it reads from the stream. The method should add all these items to the `queue` of `savedWords`, which is a private member of the class `HuffmanDecoder`.

```
void HuffmanDecoder::push(istream &f);
```

EXCEPTION: The method should throw `0` if the sequence has characters other than 0 and 1.
EXCEPTION: The method should throw `1` if the last word was not fully completed.

```
#ifndef huffman_h
#define huffman_h

#include "heap.h"
#include <iostream>
#include <vector>
#include <unordered_map>
#include <map>
#include <queue>

using namespace std;


class HuffmanCode : public map<string,string> {
public:
    // --------- HuffmanCode() --------
    // default constructor
    HuffmanCode() {}
    // ---------
    // --------- HuffmanCode(istream &) --------
    // creates a code from an input stream. The input stream has a line for every
    // code association. A line is in the form '<word> <code>'. The code should
          contain only
    // 0s and 1s
    // EXCEPT: throws 0 if the stream is not in the proper format (i.e. a line
          has more than two words, or the second word is not a sequence of 0s and
          1s
    HuffmanCode(istream &input); //TODO
};

class HuffmanTree {
private:
    // ----- TreeNode ------
    // A node in the Huffman Tree
    struct TreeNode {
        // ---------
        // links to children. these are nullptr is any of the children is non-
               existent
        TreeNode* children[2];
        // ---------
        // link to the string representation of the word
        string *word;
        // ---------
        // --------- TreeNode() --------
        // default constructor : all links are nullptr
        TreeNode() {
            children[0] = children[1] = nullptr;
            word = nullptr;
        }
        // ---------
        // --------- TreeNode(string) --------
        // children are nullptr both, and word is a link to a new string
        TreeNode(string s) {
            word = new string(s);
            children[0] = children[1] = nullptr;
        }
```

```
    // ---------
    // --------- TreeNode(string) --------
    // children are nullptr taken from the params(could be nullptr) and
    // the word is a nullptr
    TreeNode(TreeNode *t1, TreeNode *t2) {
        word = nullptr;
        children[0] = t1;
        children[1] = t2;
    }
    // ---------
    // --------- TreeNode(string) --------
    // copy constructor
    TreeNode(const TreeNode &t) {
        if( t.children[1] != nullptr) children[0] = new TreeNode(*t.children
                [0]);
        if( t.children[1] != nullptr) children[1] = new TreeNode(*t.children
                [1]);
        if( t.word != nullptr) word = new string(*t.word);
    }

    // --------- ~TreeNode() --------
    // destructor - deallocates the memory at the node and its descendants
    ~TreeNode() {
        if (word != nullptr) delete word;
        if (children[0] != nullptr) delete children[0];
        if (children[1] != nullptr) delete children[1];
    }
};
// -----
// ----- HuffmanHeap --------
class HuffmanHeap : Heap<TreeNode *> {
public:
    // -----
    // ----- HuffmanHeap(istream &) --------
    // constructor from an input file: a TreeNode is constructed for every
            node in the input file
    // with no children and with priority equal to the frequency of the word
            in the file
    HuffmanHeap(istream &); //TODO
    // -----
    // ----- pop() --------
    // removes the items with the two highest priorities, creates a new
            TreeNode with these two items as children and no string content,
            and adds this new Node in the heap with priority equal to the sum
            of the priorities of the two removed nodes.
    // does not do anything if the number of elements is less than 2
    void pop(); //TODO
    // -----
    // ----- lastElement() -----
    // returns the top element when the heap has only one element
    // EXEPT: throws 1 if the number of elements is not 1
    TreeNode* lastElement() {
        if (content.size() != 1) throw 1;
        return *(content[0]->data);
    }
    // -----
```

```
        // ----- hasOneElementLeft() -----
        // returns true only if the heap has one element
        bool hasOneElementLeft() const {
            return (content.size() == 1);
        }

    };
    // -----
    TreeNode *root; //the root of the tree
    TreeNode *iter; //an iterator that keeps track of a moving position in the
          tree
    // ----
public:
    // ----
    // ---- HuffmanTree(const HuffmanCode &) -----
    // constructor builds a tree that corresponds to the codes given as parameter
    // EXCEPT: throws 1 if the code is not a prefix code
    // EXCEPT: throws 2 if the codes are not all sequences of 0s and 1s
    HuffmanTree(const HuffmanCode &); //TODO
    // ----
    // ---- HuffmanTree(istream &) -----
    // constructor builds a tree from a file containing a block of text. It
          builds a Huffman Heap
    // and pops from it until only one element is left
    HuffmanTree(istream &input) {
        HuffmanHeap h(input);
        while(!h.hasOneElementLeft()) {
            h.pop();
        }
        root = h.lastElement();
        iter = root;
    }
    // ----
    // ---- HuffmanTree(const HuffmanTree &) ----
    // copy constructor
    HuffmanTree(const HuffmanTree &h) {
        root = new TreeNode(*h.root);
        iter = root;
    }
    // ----
    // ---- resetIterator() ----
    // moves iterator back to the root
    void resetIterator() {
        iter = root;
    }
    // ----
    // ---- moveDownOnZero() ----
    // moves iterator down a 0 branch
    // EXCEPT: throws 0 if no such branch exists
    void moveDownOnZero() {
        if (iter->children[0] == nullptr) throw 0;
            iter = iter->children[0];
    }
    // ----
    // ---- moveDownOnOne() ----
    // moves iterator down a 1 branch
```

```
    // EXCEPT: throws 1 if no such branch exists
    void moveDownOnOne() {
        if (iter->children[1] == nullptr) throw 1;
        iter = iter->children[1];
    }
    // ----
    // ---- getWordFromIter() ----
    // returns a pointer to the string corresponding to
    // the iterator
    // EXCEPT: throws 2.0 if no such string exists
    const string *getWordFromIter() const{
        if ( iter->word == nullptr) throw 2.0;
        return iter->word;
    }
    // ----
    // ---- dfs(HuffmanCode& hc, TreeNode *tn, string &s) ----
    // performs dfs starting at node tn. The string keeps track of the branches
       one has
    // to follow from the root to this node. It adds to hc all leaf nodes
    //
    void dfs(HuffmanCode& hc, TreeNode *tn, string &s);
    // ----
    // ---- getCode() ----
    // returns a HuffmanCode corersponding to the current HuffmanTree
    HuffmanCode getCode() {
        HuffmanCode toRet;
        string ss("");
        dfs(toRet, root, ss);
        return toRet;
    }
};

class HuffmanDecoder : public HuffmanTree {
private:
    // ----
    // savedWords keeps track of words that have been decoded every time
    // the method push() is called
    queue<const string*> savedWords;
    // ----
public:
    // -------
    // ---- HuffmanDecoder(istream &) -----
    // overrides constructor of HuffmanTree
    HuffmanDecoder(istream &input) : HuffmanTree(input) {}
    // -------
    // ---- HuffmanDecoder(const HuffmanCode &) -----
    // overrides constructor of HuffmanTree
    HuffmanDecoder(const HuffmanCode &hc): HuffmanTree(hc) {}
    // -------
    // ---- push(istream &) ----
    //It decodes a sequence of 0s and 1s that is stored in the stream f. All
       decoded words are
    // added to a queue of decoded words, which can be extracted using the method
       next()
    // EXEPT: throws 0 if the sequence has characters other than 0 and 1
    // EXEPT: throws 1 if the last word was not fully completed
```

```
    void push(istream &f); //TODO
    // -------
    // ---- next() ------
    // extracts a single word that were decoded in the method push
    // EXEPT: throws 1 if the queue of words is empty
    string next() {
        if(savedWords.empty()) throw 1;
        string toRet = *savedWords.front();
        savedWords.pop();
        return toRet;
    }
};

class HuffmanEncoder {
private:
    // ----
    // the HuffmanCode used to decode the word
    const HuffmanCode &code;
public:
    // ----
    // ---- HuffmanEncoder(const HuffmanCode &) ---
    // constructor that initialized the code used for decoding
    HuffmanEncoder(const HuffmanCode &t) : code(t) {}

    // ----
    // ---- encode(istream &fin, ostream &fout)
    // reads content from fin and pushes the corresponding encoding to fout
    // EXCEPT: throws 1 if fin contains words that are not in the dictionary of
          code
    void encode(istream &fin, ostream &fout) const; //TODO
};

#endif /* huffman_h */
```