```
#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 */
```