T-201-GSKI, GAGNASKIPAN
VOR 2014

BINARY TREES

SKILAVERKEFNI 2

Assignment grading. A full mark is given for code that is implemented as specified and accepted by Mooshak. Partially completed solutions (that compile on Mooshak), will be graded by hand and get a maximum grade of 7 for each part. *Note*. Points may be deducted for a solution which fails to meet the implementation requirements specified below, regardless of whether it is accepted by Mooshak or not.

Important message from the establishment

While seemingly long, the text below is *actually* important to read. Before you start, go ahead and get comfortable, and spend a little time to really understand what follows. We guarantee that this will reduce the number of issues and bugs you will run into manifold. It pays dividends.

Hand-in

You do **not** have to hand in your code to MySchool. It suffices to submit your solution to Mooshak.

Introduction

Freebase powers the Google knowledge graph with more than 23 million facts about the world. A fact can be on a single person, place, or thing. The great thing about Freebase is that *anyone* on the web can download it as a giant text file. The problem, however, is that Freebase requires 250GB of disk space, which introduces a lot of problems when distributing it through a network. Fortunately, there exist compression algorithms to deal with this issue. This is similar to what you may be used to with ZIP files, RAR files, and other archiving software. Google's choice of software for compressing and decompressing Freebase is *gzip*, a free program that uses a combination of Huffman coding and the Lempel-Ziv algorithm. Compressed with gzip, the Freebase database becomes merely a tenth of its original size, or 22GB.

In this assignment, we are going to implement Huffman coding. By doing so, we will become familiar with the binary tree data structure which is ubiquitous in computing applications. Our focus with the assignment will be on Huffman coding — we will not consider the Lempel-Ziv algorithm here.

Date: February 14, 2014.

Huffman coding

Some letters in written text appear more frequently than others. For instance, the number of 'e' characters in this sentence is significantly greater than the number of 'g' characters. Huffman coding exploits this fact to represent letters as compactly as possible. For instance, while the ASCII encoding scheme uses 7 bits to represent each character, Huffman coding on average requires only about 4. To determine a Huffman coding for a given text string, we must build a binary tree. This tree is then used to both encode and decode the data.

It would be amazing if we could take any file and compress it indefinitely, like putting your ZIP file in another ZIP file. Unfortunately, we eventually hit a wall when the data that is being compressed kind of looks random. In particular, Huffman coding works extremely well when encoding text, such as Freebase, because language text has a lot of inherent redundancy built into it. (You cn raed tihs sntanece fr itsncae). Although we will only consider text in this assignment, keep in mind that it also works for virtually any kind of data, such as photos, videos or music. In fact, most likely all the photos, videos and music you have been enjoying for the past few days are all compressed with some variant of a Huffman tree.

Build Encoding tree (40%)

The algorithm for building a Huffman encoding tree takes as input a text string and returns a binary tree. The first step of this algorithm is to count the number of occurrences of each character in the given text string. For instance, consider the word "mississippi" which has the character frequency distribution:

```
m 1
i 4
s 4
p 2
```

This table will determine the structure of our binary tree. The next step is to place the counts from the frequency table into binary tree nodes, with each node storing a character and a count of its frequency. These nodes are placed in a queue, ordered by frequency, with the least occurring character in the front. In the case of "mississippi", our queue would look like Figure 1.

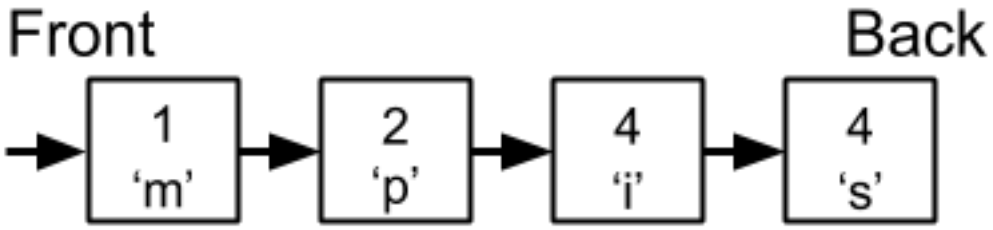

Figure 1. Queue containing the characters in 'mississippi', ordered by frequency

With this queue, we build the encoding tree. The algorithm for doing so is as follows:

1. pop the two nodes from the queue (the two with the smallest frequencies), let `a` denote the first removed node and `b` the second,
2. place `a` and `b` as children of a new node `p` whose frequency is the sum of the frequency of `a` and `b`; `a` becomes the left child, and `b` the right child,
3. push `p` into the queue in sorted order.
4. Repeat steps 1–3 until the queue contains only one binary tree node with all the others as its children. This will be the root of our finished encoding tree.

Figure 2 shows three iterations of this algorithms for the word "mississippi" with a complete encoding tree appearing in round 3.

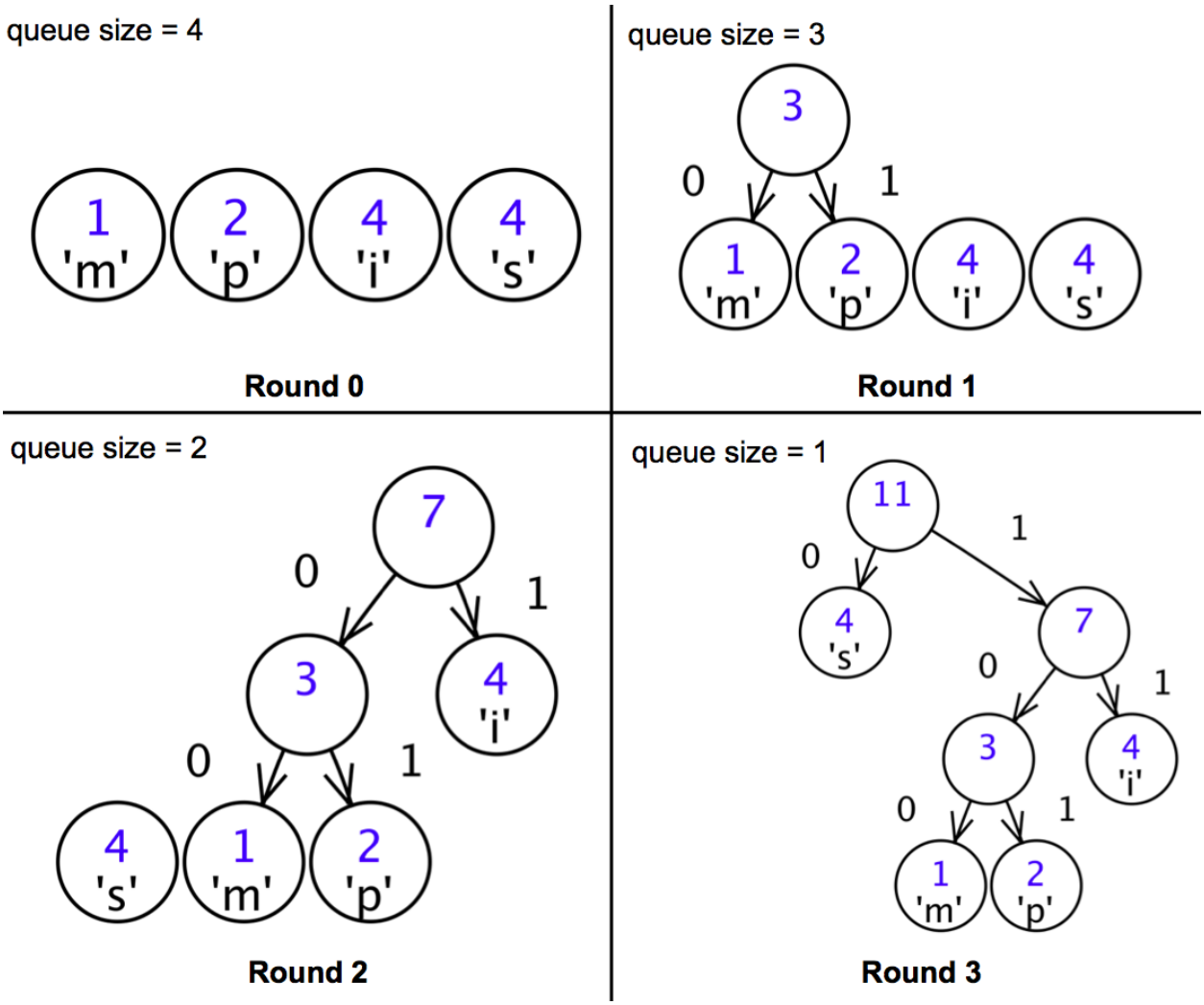

FIGURE 2. Three iterations of running the algorithm to build an encoding tree for 'mississippi'

Notice that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. This structure can be used to create an efficient encoding.

In this first part of the assignment, we will build the encoding tree given a character frequency distribution.

Implementation. A template for a main-program that constructs a Huffman encoding tree is supplied in `main.cpp`. You are provided with two data structures to aid in the implementation, `HuffmanNode` and `HuffmanPriorityQueue`. These should not be modified. `HuffmanNode` is a structure representing a node in a Huffman encoding tree. The interface of `HuffmanNode` is as follows.

```
struct HuffmanNode
{
    HuffmanNode(int frequency, char letter);
    bool is_leaf();

    // The number of occurrences of the characters represented by this node and
    // both of its subtrees (i.e. 'left' and 'right').
    int frequency;
    // If this node is a leaf, 'letter' stores character being represented by
    // this node; otherwise 'letter' is the null character ('\0').
    char letter;
    // The 0-subtree.
    HuffmanNode *left;
    // The 1-subtree.
    HuffmanNode *right;
};

typedef HuffmanNode* NodePtr;
```

The `HuffmanPriorityQueue` is an abstract data type that can be used to keep track of which nodes need to be joined in each round. The `HuffmanPriorityQueue` is a *priority queue* because `HuffmanNodes` are not removed in the order they are placed in the queue, as they would in a regular queue. Instead, the node with the lowest frequency count is returned. Below is an example code for creating and using a `HuffmanPriorityQueue`

```
HuffmanPriorityQueue queue; // Construct a queue
NodePtr h1 = new HuffmanNode(2, 'p'),
        h2 = new HuffmanNode(1, 'm');
queue.push(h1); // Push new node into queue
queue.push(h2); // Push new node into queue

// Access node with the lowest frequency count, in this case 'm'
NodePtr a = queue.top();

// Remove node with the lowest frequency count, in this case 'm'
queue.pop();
```

Write a main-program in `main.cpp` that constructs a Huffman encoding tree from a given frequency distribution. The function `build_tree` should be used to input the frequency distribution table and build the Huffman tree. The function should then return a pointer to the root of the tree.

You must also implement the function `free_memory`, that deallocates the memory allocated for the Huffman tree.

Input and output description. The input is a character frequency distribution table. The first line is a number `n` which then is followed by `n` lines. Each line contains a character and its respective frequency count, separated by a space. You may assume that white-space characters are not included in the frequency distribution table.

The output consists of the textual representation of the Huffman encoding tree. You are provided with an implementation of the function `print_tree` for visualizing the tree.

Input example 1

```
4
m 1
i 4
s 4
p 2
```

Output example 1

```
(  [11]
  0(s [4])
  1(  [7]
    0(  [3]
      0(m [1])
      1(p [2])
    )
    1(i [4])
  )
)
```

Input example 2

```
4
a 3
i 1
k 5
l 1
```

Output example 2

```
(  [10]
  0(  [5]
    0(  [2]
      0(i [1])
      1(l [1])
    )
    1(a [3])
  )
  1(k [5])
)
```

Decoding (30%)

Grab your favorite beverage, get cozy on the couch, and take a moment to think about what you have accomplished so far. By using nodes, binary trees and a queue, you have written a program that has the potential to optimally encode and decode *any* data. That's pretty spectacular. Let's put your program into use and extend it to decode a Huffman-encoded string.

You can use a Huffman tree to decode text that was previously encoded with its binary patterns. The decoding algorithm is to read each bit (i.e., character) from the encoded string, one at a time, and use this bit to traverse the Huffman tree. If the bit is a 0, you move to the left child in the tree. If the bit is 1, you move to the right one. You repeat the travesal until you hit a leaf node. Leaf nodes represent actual characters, so once you reach a leaf, you output that character and return back to the root node. For example, suppose we are given the same encoding tree above for 'mississippi', and we are asked to decode a file containing the following bits:

```
1001100
```

Using the Huffman tree, we walk from the root until we find a leaf, then output letter within the leaf node and go back to the root:

- We read a 1 (right), then a 0 (left), then a 0 (left). We reach 'm' and output m. Back to the root.
- We read a 1 (right), then a 1 (right). We reach 'i' and output i. Back to the root.
- We read a 0 (left). We reach 's' and output s. Back to the root.
- We read a 0 (left). We reach 's' and output s. Back to the root.

Decoding the bits '1001100' outputs the word 'miss'. Notice that if you had written these four characters in ASCII, you would require 28 bits instead of 7.

Implementation. Extend the main program to perform decoding. You should implement the function `string decode(NodePtr root, string encoded_str)` in your main program which implements the algorithm explained above.

Input and output description. The input begins with a character frequency distribution table as described in the previous part. The line following the table contains a single integer `k`, followed by `k` lines, each containing a bit string encoded with the given character distribution table.

The output consists of `k` text strings, each on a separate line, which are the decoded representations of the given `k` encoded bit strings.

Input example 1

```
4
m 1
i 4
s 4
p 2
4
1001100
```

```
100110011001110110111
0000
11111111
```

Output example 1

```
miss
mississippi
ssss
iiii
```

Input example 2

```
4
a 3
i 1
k 5
l 1
4
0010111
10111010010111000
101001001000
01001001000
```

Output example 2

```
lakk
kakkalakki
kalli
alli
```

Encoding (30%)

The Huffman code for each character is derived from your binary tree by thinking of each left branch as a bit value of 0 and each right branch as a bit value of 1, as shown in Figure 3.

The code for each character can be determined by traversing the tree. To reach 'p' we go right once, left once and right once, obtaining the code '101'. The code for 's' is '0', the code for 'm' is '100' and the code for 'i' is '11'.

Recall that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. Frequently used characters therefore have short codes and less frequently used characters have longer codes. This is what makes Huffman coding efficient.

In this last part of the assignment, your task is to implement the encoding process.

One way to do so is to create a helper function `bool contains(NodePtr root, char letter)` that returns `true` if the tree `root` contains the letter `letter`, and `false` otherwise.

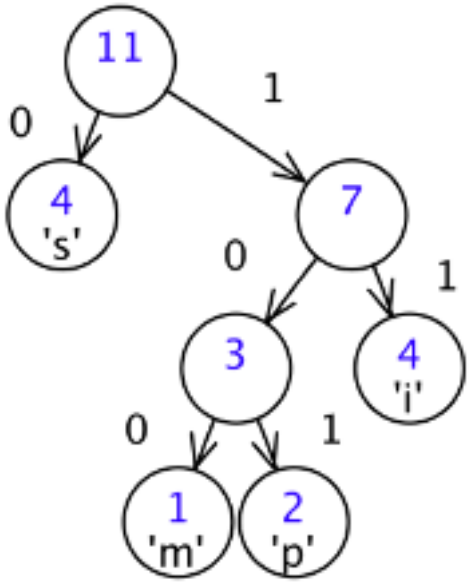

FIGURE 3. Huffman encoding tree for 'mississippi'

Using the `contains` helper function, we can encode a character `c` as follows. We start traversing the encoding tree at the root. For each node we visit, we check if the left subtree contains `c`. If it does, we know that the next bit of the encoding is `0`, and we move to (and visit) the left child. Otherwise, the right subtree contains `c`, in which case we know that the next bit of the encoding is `1` and we move to the right child.

Implementation. Extend the main program to perform encoding. You should implement the function `string encode(NodePtr root, char letter)` in your main program which you can implement using the algorithm explained above.

Input and output description. The input begins with a character frequency distribution table as described in the previous parts. In the line appearing after the table is a number `k`. Then follow `k` lines each containing a single text string. You may assume that all characters in the `k` text strings also appear in the character distribution table. You may also assume that the text strings do not contain white space characters.

The output consists of `k` strings, each on a separate line, which are the encodings of the given `k` text strings.

Input example 1

```
4
m 1
i 4
s 4
p 2
4
miss
mississippi
ssss
iiii
```

Output example 1

```
1001100
100110011001110110111
0000
11111111
```

Input example 2

```
4
a 3
i 1
k 5
l 1
4
lakk
kakkalakki
kalli
alli
```

Output example 2

```
0010111
10111010010111000
101001001000
01001001000
```

Hints.

- Take your time to read this assignment description carefully and understand what you have to do for each part *before* you begin coding.
- No seriously. This problem involves various abstractions and needs to be thought about. If you invest the time to really read every word in this problem description, you will run into far fewer issues. It really is worth it.
- If you are uncertain about something, we encourage you to post a question on Piazza.
- This short video explanation, https://www.youtube.com/watch?v=N-kOx53fU_Q, may help you get started on the assignment.
- A great tool for visualizing Huffman trees is available online, http://www.cs.usfca.edu/~galles/visualization/java/download.html. Download the 'visualization.jar' file and run it locally on your computer. After running application, click on the 'Algorithms' menu in the menubar and select 'Huffman coding'.
- We suggest you begin early working on this project. You may find it a challenging experience to program with binary trees, in particular if you have little experience with recursion.
- Work step-by-step. Get each part of the encoding program working before starting on the next one.
- Start out with small test files to practice on *locally* before you try to submit your solution to Mooshak. This may improve your work-flow and save you time.

- In your implementation, it is normal for the functions `build_tree`, `encode` and `decode` to be iterative (i.e., non-recursive). The functions `contains` and `free_memory` may be easier to implement recursively.
- Don't forget to call `free_memory` at the end of your program to deallocate the allocated memory.

Submitting. To submit this problem to Mooshak you must create a zip file containing *HuffmanNode.h*, *HuffmanNode.cpp*, *HuffmanPriorityQueue.h* and *main.cpp*.

This assignment was developed by Gunnar J. Viggósson, Hjalti Magnússon and Ólafur P. Geirsson, with sections adapted from a project by Marty Stepp at Stanford.

School of Computer Science, Reykjavík University, Menntavegi 1, 101 Reykjavík

E-mail address: `hjaltim@ru.is`