Tunbutr, Bryant

CSCI 230

Data Structures II

Lab Project #3

TEXT PROCESSING

Due Date

5/12/2014

Date Turned In

5/14/2014

Project specification

Project is completed, though the display of frequency table is incorrect.

Lessons learned were numerous, the biggest lesson was/is that falling behind in data structures and assembly and playing catch up is very difficult. It is better to start early and keep current, or at the very least not fall too far behind.

In terms of coding, I learned to print to the console at every step possible to check for correct input and output. I also learned to take small steps like first get the file to be stored as a string, then run the algorithms.

Furthermore to ensure the number of comparisons were accurate, I first used the string from the textbook and made sure my results were valid. Basically I learned to test the algorithms and results with the smaller text first.

Summary

KMP O(n+m) is far more efficient than BM O(nm)

Strings can be compressed in far fewer bits with the Huffman Algorithm

Output

Searches in the Declaration of Independence

BM search for pattern Providence

The pattern was found at index 8594

The number of comparisons was 1164

KMP search for pattern Providence

The pattern was found at index 8594

The number of comparisons was 12

BM search for pattern Unanimous

The pattern was found at index 115

The number of comparisons was 23

KMP search for pattern Unanimous

The pattern was found at index 115

The number of comparisons was 9

BM search for pattern zzzz

The pattern was not found

The number of comparisons was 2173

KMP search for pattern zzzz

The pattern was not found

The number of comparisons was 7

BM search for pattern natural

The pattern was found at index 3675

The number of comparisons was 639

KMP search for pattern natural

The pattern was found at index 3675

The number of comparisons was 213

The scanned text

more money needed

The frequency table

m 000

y 0010

0011

o 010

011

e 10

r 1100

1101

d 1110

n 1111

The compressed string

110100110000101100100110000101111100010011111110101110101110

The decompressed string

more money needed

moneyOutput.txt

1110 d

10 e

010 o

0011

1111 n

011

0010 y

000 m

1100 r

1101

\*\*

110100110000101100100110000101111100010011111110101110101110

Source Code

TextProcessing.java

/\* Java Class: TextProcessing

Author: Bryant Tunbutr

Class: Data Structures II

Date: 5/14/14

Description: This sorts text

I certify that the code below is my own work.

Exception(s): N/A

\*/

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.HashMap;

**import** java.util.PriorityQueue;

**import** java.util.Scanner;

**public** **class** TextProcessing {

**static** String *declarationIndepString*;

**static** **int** *comparisonsInt*;

@SuppressWarnings("static-access")

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

TextProcessing textProc4 = **new** TextProcessing();

Scanner declar4 = **new** Scanner(**new** File("usdeclar.txt"));

String declar4String = "";

**while** (declar4.hasNext()) {

declar4String += '\n' + (declar4.nextLine());

}

System.*out*.println("Searches in the Declaration of Independence");

System.*out*.println();

String pattern4 = "Providence";

System.*out*.println("BM search for pattern " + pattern4);

textProc4.*BMmatch*(declar4String, pattern4);

System.*out*.println(*displayIndex*(textProc4.*BMmatch*(declar4String,

pattern4)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

System.*out*.println("KMP search for pattern " + pattern4);

textProc4.*KMPmatch*(declar4String, pattern4);

System.*out*.println(*displayIndex*(textProc4.*KMPmatch*(declar4String,

pattern4)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

System.*out*.println();

String pattern5 = "Unanimous";

System.*out*.println("BM search for pattern " + pattern5);

textProc4.*BMmatch*(declar4String, pattern5);

System.*out*.println(*displayIndex*(textProc4.*BMmatch*(declar4String,

pattern5)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

System.*out*.println("KMP search for pattern " + pattern5);

textProc4.*KMPmatch*(declar4String, pattern5);

System.*out*.println(*displayIndex*(textProc4.*KMPmatch*(declar4String,

pattern5)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

System.*out*.println();

String pattern6 = "zzzz";

System.*out*.println("BM search for pattern " + pattern6);

System.*out*.println(*displayIndex*(textProc4.*BMmatch*(declar4String,

pattern6)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

System.*out*.println("KMP search for pattern " + pattern6);

textProc4.*KMPmatch*(declar4String, pattern6);

System.*out*.println(*displayIndex*(textProc4.*KMPmatch*(declar4String,

pattern6)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

System.*out*.println();

String pattern7 = "natural";

System.*out*.println("BM search for pattern " + pattern7);

System.*out*.println(*displayIndex*(textProc4.*BMmatch*(declar4String,

pattern7)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

System.*out*.println("KMP search for pattern " + pattern7);

textProc4.*KMPmatch*(declar4String, pattern7);

System.*out*.println(*displayIndex*(textProc4.*KMPmatch*(declar4String,

pattern7)));

System.*out*.println("The number of comparisons was " + *comparisonsInt*);

HuffmanCoding huff1 = **new** HuffmanCoding();

// scan file & store as String

String moneyTextFileString = **new** Scanner(**new** File("money.txt"))

.useDelimiter("\\A").next();

System.*out*.println();

System.*out*.println("The scanned text");

System.*out*.println(moneyTextFileString);

System.*out*.println();

// create hashmap<character, frequency> based on input string

HashMap<Character, Integer> huffmanHashMap = **new** HashMap<Character, Integer>();

// count character frequency

**for** (**int** i = 0; i < moneyTextFileString.length(); i++) {

**char** currentChar = moneyTextFileString.charAt(i);

// increment for each instance of the char

**if** (huffmanHashMap.containsKey(currentChar))

huffmanHashMap.put(currentChar,

huffmanHashMap.get(currentChar) + 1);

**else**

huffmanHashMap.put(currentChar, 1);

}

// instantiate priority queue

// use java.util compare to sort in smallest order

huff1.*priorityQueue* = **new** PriorityQueue<Node>(100,

**new** FrequencyComparator());

**int** nodesCountInt = 0;

// iterate through the hashmap and

// add nodes to priority queue

**for** (Character characterChar : huffmanHashMap.keySet()) {

huff1.*priorityQueue*.add(**new** Node(characterChar, huffmanHashMap

.get(characterChar)));

nodesCountInt++;

}

// identify the root of the tree, the largest

Node rootNode = huff1.*huffman*(nodesCountInt);

System.*out*.println("The frequency table");

// build the table for the variable length codes

huff1.*frequencyTable*(rootNode);

System.*out*.println();

String compressed = huff1.*compress*(moneyTextFileString);

System.*out*.println("The compressed string");

System.*out*.println(compressed);

System.*out*.println();

String decompressed = huff1.*decompress*(compressed);

System.*out*.println("The decompressed string");

System.*out*.println(decompressed);

huff1.*saveToFile*(compressed);

}

/\*\*

\* Simplified version of the Boyer-Moore (BM) algorithm, which uses only the

\* looking-glass and character-jump heuristics.

\*

\* **@return** Index of the beginning of the leftmost substring of the text

\* matching the pattern, or -1 if there is no match.

\*/

**public** **static** **int** BMmatch(String text, String pattern) {

**int**[] last = *buildLastFunction*(pattern);

**int** n = text.length();

**int** m = pattern.length();

**int** i = m - 1;

**if** (i > n - 1)

**return** -1; // no match if pattern is longer than text

**int** j = m - 1;

*comparisonsInt* = 0;

**do** {

*comparisonsInt*++;

**if** (pattern.charAt(j) == text.charAt(i))

**if** (j == 0)

**return** i; // match

**else** { // looking-glass heuristic: proceed right-to-left

i--;

j--;

}

**else** { // character jump heuristic

i = i + m - Math.*min*(j, 1 + last[text.charAt(i)]);

j = m - 1;

}

} **while** (i <= n - 1);

**return** -1; // no match

}

**public** **static** **int**[] buildLastFunction(String pattern) {

**int**[] last = **new** **int**[128]; // assume ASCII character set

**for** (**int** i = 0; i < 128; i++) {

last[i] = -1; // initialize array

}

**for** (**int** i = 0; i < pattern.length(); i++) {

last[pattern.charAt(i)] = i; // implicit cast to integer ASCII code

}

**return** last;

}

**public** **static** **int** KMPmatch(String text, String pattern) {

*comparisonsInt* = 0;

**int** n = text.length();

**int** m = pattern.length();

**int**[] fail = *computeFailFunction*(pattern);

**int** i = 0;

**int** j = 0;

**while** (i < n) {

// comparisonsInt++;

**if** (pattern.charAt(j) == text.charAt(i)) {

*comparisonsInt*++;

**if** (j == m - 1)

**return** i - m + 1; // match

// comparisonsInt++;

i++;

j++;

} **else** **if** (j > 0)

j = fail[j - 1];

**else** {

// comparisonsInt++;

i++;

}

}

**return** -1; // no match

}

**public** **static** **int**[] computeFailFunction(String pattern) {

**int**[] fail = **new** **int**[pattern.length()];

fail[0] = 0;

**int** m = pattern.length();

**int** j = 0;

**int** i = 1;

**while** (i < m) {

// comparisonsInt++;

**if** (pattern.charAt(j) == pattern.charAt(i)) { // j + 1 characters

// match

*comparisonsInt*++;

fail[i] = j + 1;

i++;

j++;

} // j follows a matching prefix

**else** **if** (j > 0) {

*comparisonsInt*++;

j = fail[j - 1];

} **else** { // no match

fail[i] = 0;

i++;

// comparisonsInt++;

}

}

**return** fail;

}

**public** **static** String displayIndex(**int** index) {

String indexDisplay = "";

**if** (index == -1) {

indexDisplay = "The pattern was not found";

} **else** {

indexDisplay = "The pattern was found at index " + index;

}

**return** indexDisplay;

}

}

HuffmanCoding.java

/\* Java Class: HuffmanCoding

Author: Bryant Tunbutr

Class: Data Structures II

Date: 5/14/14

Description: This sorts text

I certify that the code below is my own work.

Exception(s): N/A

\*/

**import** java.util.\*;

**import** java.io.\*;

**public** **class** HuffmanCoding {

**public** **static** PriorityQueue<Node> *priorityQueue*;

**public** **static** HashMap<Character, String> *charToCodeHashMap*;

**public** **static** HashMap<String, Character> *codeToCharHashMap*;

// build the tree based on the frequency of the characters

**public** **static** Node huffman(**int** n) {

Node a, b;

// remove two smallest nodes from PQ, add frequencies & add notes back

// to PQ

// repeat until only 1 remaining node which becomes root

**for** (**int** i = 1; i <= n - 1; i++) {

Node node = **new** Node();

// get the two smallest nodes from priority queue

node.left = a = *priorityQueue*.poll();

node.right = b = *priorityQueue*.poll();

// add frequencies and add that node back to priority queue

node.freq = a.freq + b.freq;

*priorityQueue*.add(node);

}

// last node remaining, the root of Huffman Tree

**return** *priorityQueue*.poll();

}

// frequency table for the compression and decompression

**public** **static** **void** frequencyTable(Node root) {

*charToCodeHashMap* = **new** HashMap<Character, String>();

*codeToCharHashMap* = **new** HashMap<String, Character>();

*postOrderTraversal*(root, **new** String());

}

// recursive method

// adding a zero if going left, one if going right

// post order traversal from root to leaves

**public** **static** **void** postOrderTraversal(Node node, String string) {

**if** (node == **null**)

**return**;

*postOrderTraversal*(node.left, string + "0");

*postOrderTraversal*(node.right, string + "1");

// visit only nodes that have keys

**if** (node.letterChar != '\u0000') {

// put node letters into hashmap

*charToCodeHashMap*.put(node.letterChar, string);

*codeToCharHashMap*.put(string, node.letterChar);

System.*out*.println(node.letterChar + " " + string);

}

}

// needs already defined dictionary and tree

**public** **static** String compress(String inputString) {

String compressedString = **new** String();

**for** (**int** i = 0; i < inputString.length(); i++)

compressedString = compressedString

+ *charToCodeHashMap*.get(inputString.charAt(i));

**return** compressedString;

}

// needs already defined dictionary and tree

**public** **static** String decompress(String inputString) {

String tempString = **new** String();

String decompressedString = **new** String();

**for** (**int** i = 0; i < inputString.length(); i++) {

tempString = tempString + inputString.charAt(i);

**if** (*codeToCharHashMap*.containsKey(tempString)) {

decompressedString = decompressedString

+ *codeToCharHashMap*.get(tempString);

tempString = **new** String();

}

}

**return** decompressedString;

}

**public** **static** **void** saveToFile(String compressedString)

**throws** FileNotFoundException {

PrintWriter oFile = **new** PrintWriter("moneyOutput.txt");

**for** (String s : *codeToCharHashMap*.keySet()) {

oFile.println(s + " " + *codeToCharHashMap*.get(s));

}

oFile.println("\*\*");

oFile.println(compressedString);

oFile.close();

}

}

**class** Node {

**public** **char** letterChar;

**public** **int** freq;

**public** Node left, right;

**public** Node(**char** l, **int** f) {

letterChar = l;

freq = f;

}

**public** Node() {

}

**public** String toString() {

**return** letterChar + " " + freq;

}

}

// use java.util compare to sort in smallest order

**class** FrequencyComparator **implements** Comparator<Node> {

**public** **int** compare(Node a, Node b) {

**int** freqA = a.freq;

**int** freqB = b.freq;

**return** freqA - freqB;

}

}