Huffman Coding

Huffman Coding เป็นชื่ออัลกอริทึมสำหรับการย่อข้อมูล เราเห็นการย่อข้อมูลอยู่สม่ำเสมอเช่นการ Zip ไฟล์ การบันทึกภาพด้วยนามสกุล JPG (lossy) หรือ PNG (lossless) และการบันทึกไฟล์เสียงด้วยนามสกุล mp3 เราจะได้เห็นแนวคิดนี้อีกครั้งในวิชา Networks และ Computer Security

Huffman Coding แปลงข้อความตัวอักขระให้กลายเป็น binary code (String ของ 0 และ 1) เช่น ข้อความ “feed me more food” ถูกแปลงเป็น “10001010011111111010111111110111000001111111100110110001” ในการบ้านนี้เราจะได้ทำความเข้าใจการสร้าง Huffman Tree และการแปลงข้อความจาก binary code ให้เป็นข้อความภาษาอังกฤษด้วย Huffman Tree ที่สร้างขึ้น

ความถี่ของตัวอักษร

นอกจาก Huffman Coding จะย่อข้อมูลแล้ว Huffman Coding ยังย่อข้อมูลอย่างมีประสิทธิภาพด้วย นั่นคือตัวอักษรที่ปรากฏบ่อยในข้อความจะถูกย่อให้สั้นกว่าตัวอักษรที่ปรากฏแค่ไม่กี่ครั้งในข้อความ ดังนั้นเราจำเป็นต้องนับความถี่ของตัวอักษรในข้อความเข้าและเรียงตัวอักษรตามความถี่จากน้อยไปมาก เพื่อความง่ายเราจะกำหนดให้ข้อความเข้าประกอบไปด้วยตัวอักษรภาษาอังกฤษพิมพ์เล็กและช่องว่างเท่านั้น

ในไฟล์ main.cpp มีฟังก์ชัน count_frequency มาให้แล้ว ซึ่งเราสามารถเรียกใช้งานได้เลย (อาจจะศึกษาไว้เผื่อออกสอบ) ค่าที่คืนกลับมาจากฟังก์ชันนี้จะเป็น Pointer ชี้ไปยังโครงสร้าง Frequency ที่เก็บข้อมูลตัวอักษรและความถี่ โดยเรียงจากน้อยไปมาก

โครงสร้างของ Frequency	class Frequency {
	vector<char> c; // ตัวอักษร
	vector<int> f; // ความถี่
	int numChar; // จำนวนตัวอักษรทั้งหมด
	};

Frequency * countFrequency(string s)	คืนค่า Pointer ชี้ไปยังโหนดแรกสุดของ Linked List ที่เก็บข้อมูลตัวอักษรและความถี่ เรียงจากน้อยไปมาก
	ตัวอย่างเช่น countFrequency(“feed me more food”) จะคืนค่า
	Frequency {
	c = {‘r’, ’d’, ’f’, ‘m’, ‘o’, ‘ ‘, ‘e’};
	f = {1, 2, 2, 2, 3, 3, 4};
	numChar = 7;
	}

การสร้าง Huffman Tree

หลังจากที่เราได้ความถี่ของตัวอักษรแต่ละตัวแล้ว เราสามารถใช้ข้อมูลนี้สร้าง Huffman Tree เพื่อใช้สำหรับการย่อข้อมูล โดยโหนดใน Huffman Tree จะเก็บค่าตัวอักษรและความถี่ ในการสร้าง Huffman Tree นั้นเราจะใช้ Queue 2 ตัวเข้ามาช่วย โดยจะกำหนดให้เป็น singleQueue และ mergeQueue อัลกอริทึมสำหรับสร้าง Huffman Tree มีดังนี้

BuildTree(Frequency):

Push โหนดข้อมูล (character:frequency) จาก Frequency ไปยัง singleQueue ตามลำดับจากความถี่น้อยไปมาก
While (จำนวนข้อมูลใน singleQueue และ mergeQueue รวมกันมากกว่า 1):
สร้างโหนด leftChild ด้วยข้อมูลจากโหนดที่มีความถี่น้อยที่สุดระหว่าง singleQueue และ mergeQueue
สร้างโหนด rightChild ด้วยข้อมูลจากโหนดที่มีความถี่น้อยที่สุดระหว่าง singleQueue และ mergeQueue
สร้างโหนด parent และกำหนด frequency เป็นผลรวมของ frequency ใน leftChild และ rightChild
Push parent ไปที่ mergeQueue
Return โหนดที่เหลือ 1 ตัวใน mergeQueue ซึ่งก็คือ root ของ Huffman Tree

ตัวอย่างการทำงานของ BuildTree

กำหนดให้ข้อมูลเข้าเป็น

```
Frequency {
        c = {‘r’, ’d’, ’f’, ‘m’, ‘ ’, ‘o‘, ‘e’};
        f = {1, 2, 2, 2, 3, 3, 4};
        numChar = 7;
}
```

จากนั้นทำการ Push ข้อมูลนี้ไปยัง singleQueue ดังนี้

singleQueue	r:1	d:2	f:2	m:2	‘ ‘:3	o:3	e:4
mergeQueue							

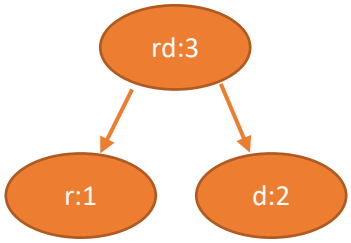
Loop 1: เนื่องจากจำนวนข้อมูลของทั้งสอง Queue มีค่ามากกว่า 1 เราจึงทำการสร้าง leftChild, rightChild, parent

leftChild เก็บข้อมูล r:1 เนื่องจากมีความถี่น้อยที่สุด

rightChild เก็บข้อมูล d:2 เนื่องจากมีความถี่น้อยที่สุดหลัง pop r:1 ออกจาก singleQueue

parent เก็บข้อมูล rd:3 (parent ไม่ได้เก็บข้อมูลตัวอักษร แต่เราจะตั้งชื่อตามตัวอักษรใน Child) โดย 3 มาจากผลรวมของความถี่ของ leftChild และ rightChild

จากนั้นก็ push parent ไปที่ mergeQueue



singleQueue	f:2	m:2	‘ ‘:3	o:3	e:4		
mergeQueue	rd:3						

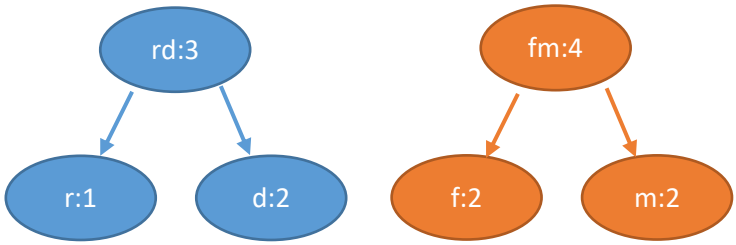
Loop 2: เนื่องจากจำนวนข้อมูลของทั้งสอง Queue มีค่ามากกว่า 1 เราจึงทำการสร้าง leftChild, rightChild, parent

leftChild เก็บข้อมูล f:2 เนื่องจากมีความถี่น้อยที่สุด

rightChild เก็บข้อมูล m:2 เนื่องจากมีความถี่น้อยที่สุดหลัง pop f:2 ออกจาก singleQueue

parent เก็บข้อมูล fm:4 โดย 4 มาจากผลรวมของความถี่ของ leftChild และ rightChild

จากนั้นก็ push parent ไปที่ mergeQueue



singleQueue	‘ ‘:3	o:3	e:4				
mergeQueue	rd:3	fm:4					

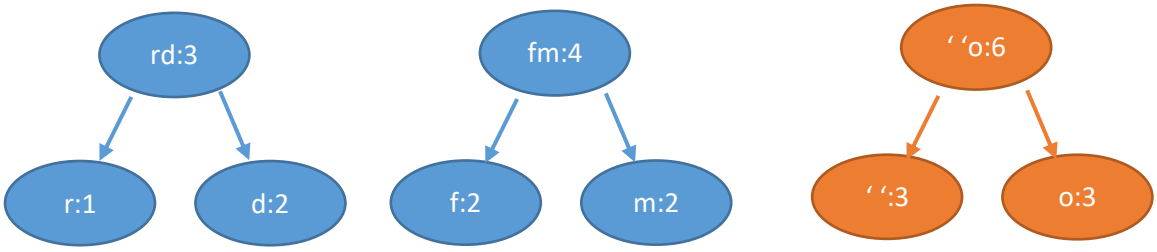
Loop 3: เนื่องจากจำนวนข้อมูลของทั้งสอง Queue มีค่ามากกว่า 1 เราจึงทำการสร้าง leftChild, rightChild, parent

leftChild เก็บข้อมูล ‘ ‘:3 เนื่องจากมีความถี่น้อยที่สุด (Break tie ด้วยการเลือก singleQueue ก่อนเสมอ)

rightChild เก็บข้อมูล o:3 เนื่องจากมีความถี่น้อยที่สุดหลัง pop o:3 ออกจาก singleQueue

parent เก็บข้อมูล ‘ ‘o:6 โดย 6 มาจากผลรวมของความถี่ของ leftChild และ rightChild

จากนั้นก็ push parent ไปที่ mergeQueue



singleQueue	e:4						
mergeQueue	rd:3	fm:4	‘ ‘o:6				

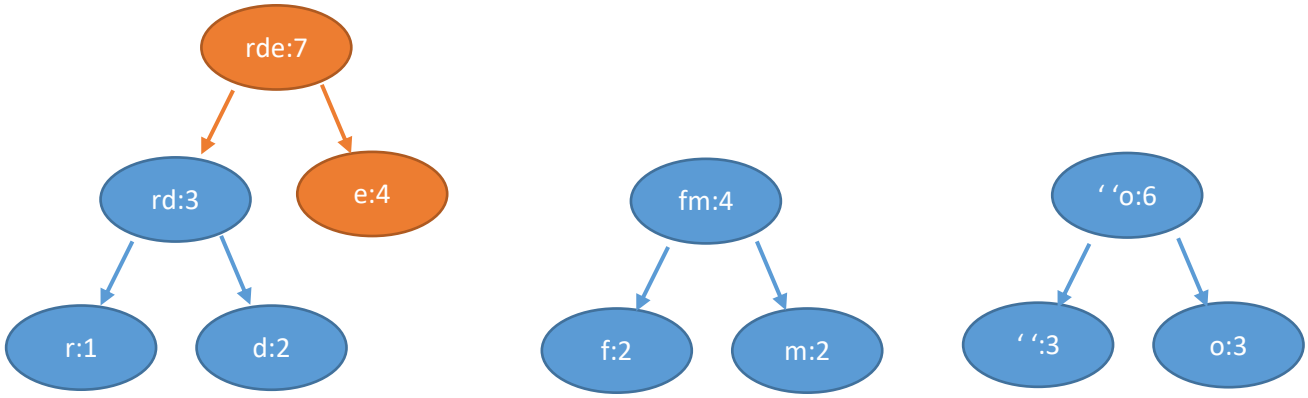
Loop 4: เนื่องจากจำนวนข้อมูลของทั้งสอง Queue มีค่ามากกว่า 1 เราจึงทำการสร้าง leftChild, rightChild, parent

leftChild เก็บข้อมูล rd:3 เนื่องจากมีความถี่น้อยที่สุด

rightChild เก็บข้อมูล e:4 เนื่องจากมีความถี่น้อยที่สุดหลัง pop rd:3 ออกจาก mergeQueue

parent เก็บข้อมูล rde:7 โดย 7 มาจากผลรวมของความถี่ของ leftChild และ rightChild

จากนั้นก็ push parent ไปที่ mergeQueue



singleQueue							
mergeQueue	fm:4	‘ ‘o:6	rde:7				

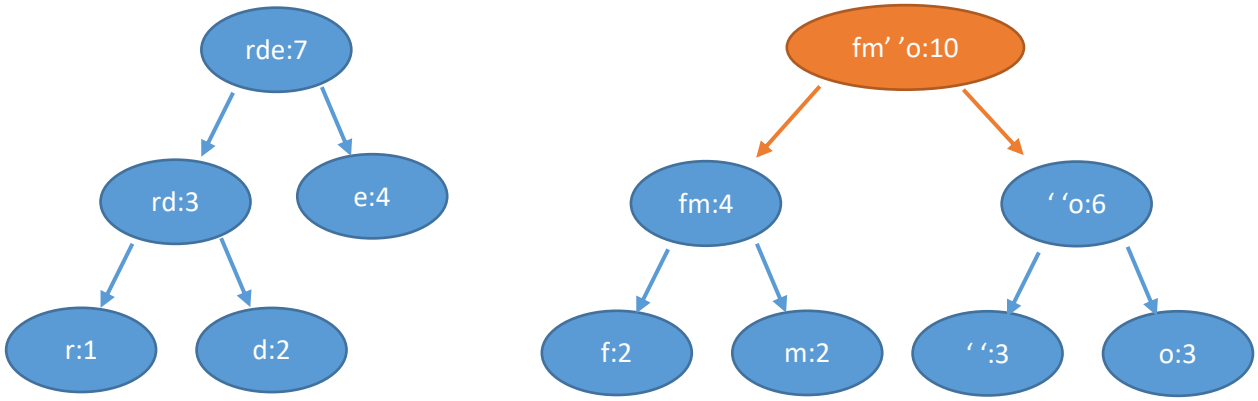
Loop 5: เนื่องจากจำนวนข้อมูลของทั้งสอง Queue มีค่ามากกว่า 1 เราจึงทำการสร้าง leftChild, rightChild, parent

leftChild เก็บข้อมูล fm:4 เนื่องจากมีความถี่น้อยที่สุด

rightChild เก็บข้อมูล ‘ ‘o:6 เนื่องจากมีความถี่น้อยที่สุดหลัง pop fm:4 ออกจาก mergeQueue

parent เก็บข้อมูล fm’ ‘o:10 โดย 10 มาจากผลรวมของความถี่ของ leftChild และ rightChild

จากนั้นก็ push parent ไปที่ mergeQueue



singleQueue							
mergeQueue	rde:7	fm’ ‘o:10					

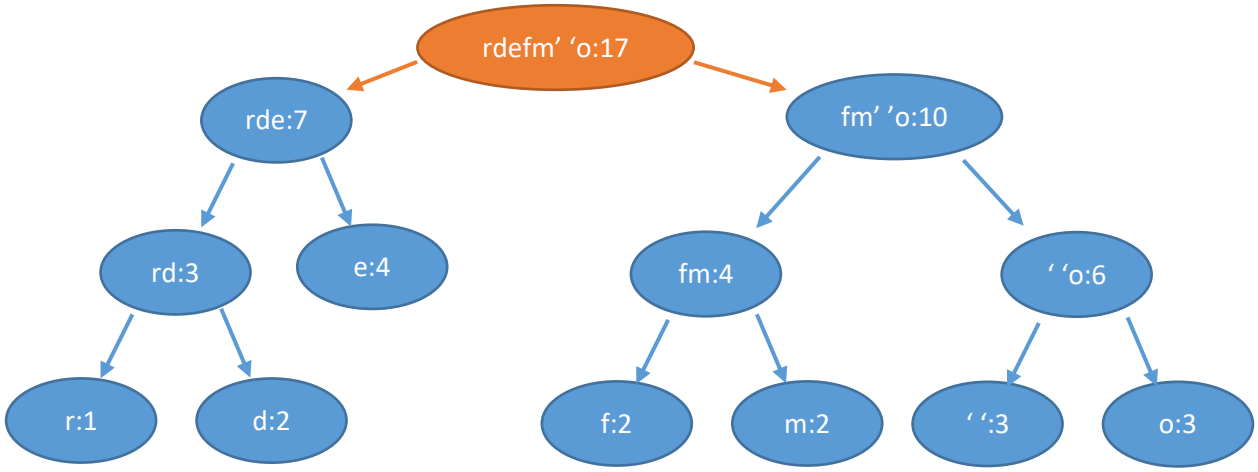
Loop 6: เนื่องจากจำนวนข้อมูลของทั้งสอง Queue มีค่ามากกว่า 1 เราจึงทำการสร้าง leftChild, rightChild, parent

leftChild เก็บข้อมูล rde:7 เนื่องจากมีความถี่น้อยที่สุด

rightChild เก็บข้อมูล fm' 'o:10 เนื่องจากมีความถี่น้อยที่สุดหลัง pop rde:7 ออกจาก mergeQueue

parent เก็บข้อมูล rdefm' 'o:17 โดย 17 มาจากผลรวมของความถี่ของ leftChild และ rightChild

จากนั้นก็ push parent ไปที่ mergeQueue

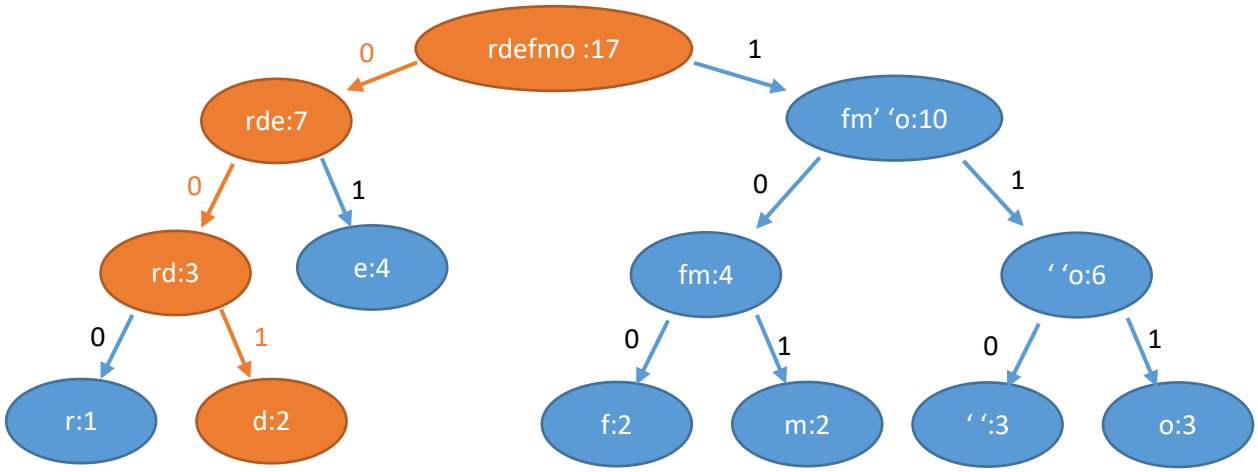


singleQueue							
mergeQueue	rdefm' 'o:17						

เนื่องจากจำนวนข้อมูลของทั้งสอง Queue มีค่าเท่ากับ 1 จึงจบการทำงาน เราจะได้โหนด rdefm' 'o:17 เป็น root ของ Huffman Tree

<u>การ Decode Binary กลับมาเป็นข้อความ</u>

เราสามารถใช้ Huffman Tree มาช่วยในการ Encode และ Decode ข้อความ สำหรับการ encode นั้นเราจะดูที่ path จาก root มายังโหนดที่มีตัวอักษรนั้น ใน path หากเราเลือกเดินทางซ้ายก็จะได้เลข 0 ถ้าเดินทางขวาก็จะได้เลข 1 เอาเลขมาต่อกันตั้งแต่ root ก็จะได้รหัสของตัวอักษรนั้น ๆ เช่น ตัวอักษร d จะถูกแทนด้วยรหัส 001



สำหรับการ Decode นั้นก็ใช้หลักการเดียวกันนั่นคือเริ่มจาก root ถ้าเจอเลข 0 ก็เดินทางต่อไปยัง leftChild ถ้าเจอเลข 1 ก็เดินทางต่อไปยัง rightChild เมื่อเดินทางถึง Leaf Node ก็จะได้ตัวอักษรที่ต้องการ ถ้ารหัสยังไม่จบก็กลับมาเริ่มที่ root ใหม่ ทำแบบนี้ไปเรื่อย ๆ จนหมดรหัสที่จะ Decode ตัวอย่างเช่น ถ้ารหัสเป็น 00111101000 เราจะได้ข้อความเป็น doer

ฟังก์ชันที่ต้องเขียน

ชื่อฟังก์ชัน	**รายละเอียดของฟังก์ชัน**
`void buildTree(Frequency *freq)`	ทำงานตามอัลกอริทึม buildTree(Frequency) ข้างต้น
`void decode(string code)`	พิมพ์ข้อความที่ได้จากการ Decode รหัส code ด้วย Huffman Tree

ในไฟล์ HW09.cpp ได้มีการกำหนด โครงสร้าง treeNode มาให้ แต่นักศึกษาสามารถเพิ่มหรือลดข้อมูลได้เอง รวมทั้งสามารถเขียนโครงสร้างและฟังก์ชันเพิ่มเติมได้

ตัวอย่าง

ข้อมูลเข้า (สำหรับสร้าง Huffman Tree)	ข้อมูลเข้า (สำหรับ Decode)	ข้อมูลออก
`feed me more food`	`00111101000`	`doer`

`aaaaaabbbccccddeeeeeffffffff`
`gggggghhhhhhhhhhiiiiijjjkkkll`
`llllmmmmnnnnnnoooooooppppppp`
`qqqqqqqqqrrrrrrssstttttttuuu`
`vvwwwxxxxxxyyyyyyyyz`
 	
(มีช่องว่างหลัง z ทั้งหมด 8 ตัว)
	`00000010011010111010`
	`10100010110101110100`
	`00001011110010110000`
	`10010110100010001110`
	`110`
		`star wars new hope`