บทที่ 3

โปรแกรมอรรถประโยชน์ด้านการอัดแฟ้มข้อมูล

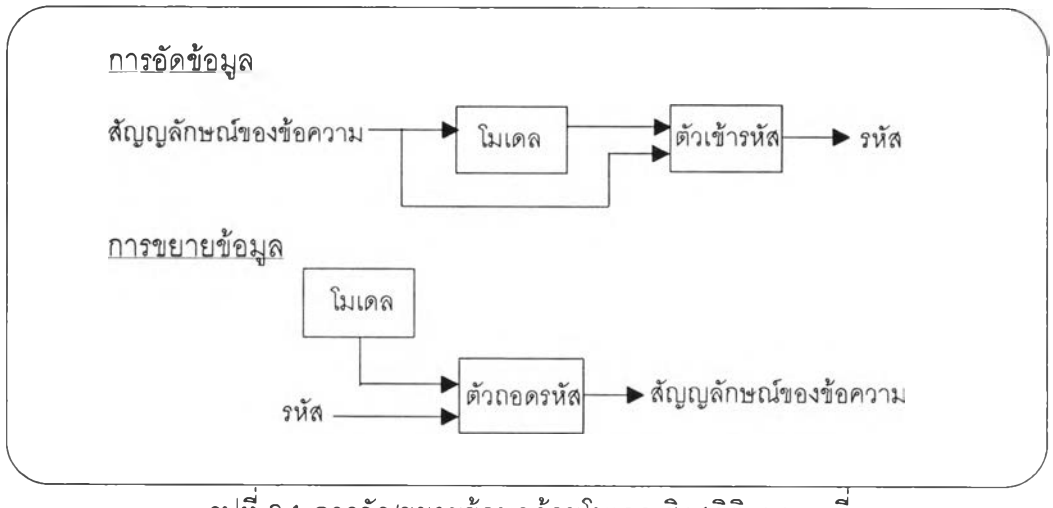
บทนี้จะกล่าวถึงแนวคิดเหตุผลและทฤษฎีที่ใช้ในการพัฒนาโปรแกรมอรรถประโยชน์สำหรับจัดการแฟ้มเอกสารของจุฬาจารึกเพื่อเสริมคุณสมบัติให้จุฬาจารึกทางด้านการอัดแฟ้มข้อมูล นอกจากนี้ยังกล่าวถึงการออกแบบและพัฒนาโปรแกรมอรรถประโยชน์ด้านการอัดแฟ้มข้อมูล บทนี้จึงแบ่งเป็น 4 ส่วน ส่วนแรกกล่าวถึงทฤษฎีพื้นฐานการอัดข้อมูล โมเดลและขั้นตอนวิธีการอัดข้อมูล ส่วนที่สองจะกล่าวถึงการเลือกขั้นตอนวิธีการอัดข้อมูลเพื่อใช้อัดแฟ้มข้อมูลจุฬาจารึกส่วนที่สามจะกล่าวถึงการอัดแฟ้มข้อมูลจุฬาจารึก และส่วนสุดท้ายกล่าวถึงโครงสร้างข้อมูลที่ใช้ในโปรแกรมและการพัฒนาโปรแกรมด้านการอัดแฟ้มข้อมูลจุฬาจารึก

ทฤษฎีพื้นฐานการอัดข้อมูล

การอัดข้อมูล (compression) เป็นการลดพื้นที่จัดเก็บแฟ้มข้อมูลให้เล็กลงเพื่อเพิ่มพื้นที่ใช้ประโยชน์ให้มากขึ้น และช่วยประหยัดเวลาในการส่งผ่านข้อมูลอีกด้วย การอัดข้อมูลจะนำเข้าสัญลักษณ์ของข้อความ (symbol of text) แล้วเปลี่ยนเป็นรหัส (code) ที่สัมพันธ์กัน ถ้าการอัดข้อมูลดังกล่าวมีประสิทธิภาพจะทำให้รหัสที่ได้มีขนาดเล็กกว่าสัญลักษณ์เดิม รหัสดังกล่าวจะไม่สามารถอ่านเข้าใจได้จึงเกิดผลพลอยได้คือการรักษาความปลอดภัยให้ข้อมูล เมื่อต้องการใช้งานแฟ้มข้อมูลเดิมต้องทำในทางตรงข้ามคือการขยายข้อมูล (decompression) จะอ่านรหัสจากการอัดข้อมูลแล้วเปลี่ยนเป็นสัญลักษณ์เดิมของข้อความ การอัดข้อมูลประกอบด้วยโมเดล (model) และวิธีลงรหัส (coding) โมเดลเป็นขั้นตอนที่ศึกษาและรวบรวมลักษณะของสัญลักษณ์ในข้อความเพื่อสร้างรหัสใหม่ที่ใช้แทนสัญลักษณ์เดิม วิธีลงรหัสเป็นขั้นตอนการเข้ารหัส (encode) ที่นำรหัสจากโมเดลมาแทนที่สัญลักษณ์เดิมหรือขั้นตอนการถอดรหัส (decode) ที่นำสัญลักษณ์เดิมของรหัสจากโมเดลกลับคืน ตัวอย่างการอัด/ขยายข้อมูลด้วยโมเดลเชิงสถิติแบบคงที่ (static statistical model) แสดงดังรูปที่ 3.1

การอัดข้อมูลจะอ่านสัญลักษณ์ของข้อความมายังโมเดลเพื่อหาความน่าจะเป็นที่จะพบสัญลักษณ์แต่ละตัวแล้วสร้างรหัสใหม่ให้แต่ละสัญลักษณ์ เช่นวิธี Huffman Coding จะกำหนด

ให้รหัสของสัญลักษณ์ที่มีความน่าจะเป็นสูงมีจำนวนบิตน้อย และรหัสของสัญลักษณ์ที่มีความน่าจะเป็นต่ำมีจำนวนบิตมาก จากนั้นตัวเข้ารหัส (encoder) จะอ่านสัญลักษณ์ของข้อความเข้ามาแล้วนำรหัสจากโมเดลมาแทนที่สัญลักษณ์เดิม ส่วนการขยายข้อมูลจะใช้โมเดลเดิมจากการอัดข้อมูลโดยตัวถอดรหัส (decoder) จะอ่านรหัสเข้ามาแล้วนำสัญลักษณ์เดิมของรหัสจากโมเดลกลับคืนเป็นข้อความตามเดิม ประสิทธิภาพการอัดข้อมูลขึ้นอยู่กับโมเดลและวิธีลงรหัส (coding) กล่าวคือการอัดข้อมูลด้วยวิธีลงรหัสเดียวกันแต่ใช้โมเดลต่างกันจะได้ผลลัพธ์ที่ต่างกันด้วยเช่น การอัดข้อมูลด้วยวิธี Huffman Coding ด้วยโมเดลที่สร้างจากข้อความทั้งหมดในแฟ้มข้อมูลจะมีประสิทธิภาพดีกว่าการอัดข้อมูลด้วยวิธี Huffman Coding ด้วยโมเดลที่สร้างจากข้อความเพียง 10 ตัวเท่านั้น เป็นต้น (Mark Nelson, 1991)



รูปที่ 3.1 การอัด/ขยายข้อมูลด้วยโมเดลเชิงสถิติแบบคงที่

1. โมเดล (model)

โมเดลแบ่งเป็น 2 กลุ่มใหญ่คือ โมเดลเชิงสถิติ (statistical model) จะอ่านเข้ามาทีละสัญลักษณ์แล้วเข้ารหัสตามค่าความน่าจะเป็นของแต่ละสัญลักษณ์นั้น และโมเดลที่ใช้พจนานุกรมอ้างอิง (dictionary-based model) จะอ่านเข้ามาเป็นกลุ่มข้อความแล้วเข้ารหัสทีละกลุ่มข้อความโดยใช้พจนานุกรมอ้างอิง (Mark Nelson, 1991)

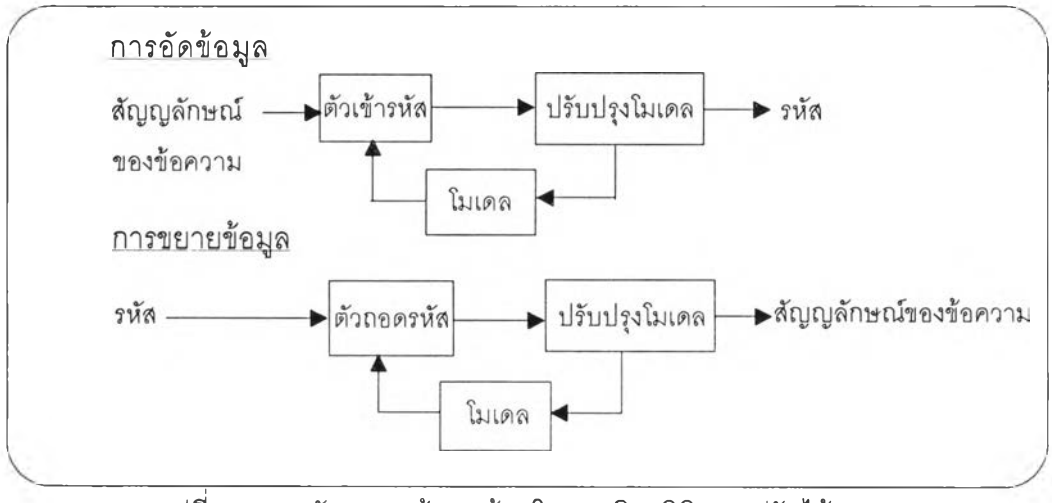
1.1 โมเดลเชิงสถิติ (statistical model) แบ่งเป็น 2 แบบคือ แบบคงที่ (static) และแบบปรับได้ (adaptive) รหัสที่สร้างจากโมเดลแบบคงที่จะไม่เปลี่ยนแปลงตลอดเวลาของการ

อัด/ขยายข้อมูล ในขณะที่รหัสซึ่งสร้างจากโมเดลแบบปรับได้จะเปลี่ยนแปลงตลอดเวลาของการอัด/ขยายข้อมูล โดยเปลี่ยนตามลักษณะของข้อความที่อ่านเข้ามาแต่ละช่วง

1.1.1 โมเดลเชิงสถิติแบบคงที่ (static statistical model) เป็นโมเดลที่ง่ายที่สุดโดยใช้ตารางจัดเก็บรหัสที่สร้างจากความน่าจะเป็นของสัญลักษณ์แต่ละตัว โมเดลแบบนี้จะทำงาน 2 รอบโดยรอบแรกจะอ่านข้อความเข้ามาทั้งหมดเพื่อหาว่ามีสัญลักษณ์ใดบ้าง แต่ละสัญลักษณ์มีความน่าจะเป็นเท่าใด แล้วสร้างรหัสใหม่จากความน่าจะเป็นของสัญลักษณ์นั้น รอบสองจะเข้ารหัสสัญลักษณ์แต่ละตัวด้วยรหัสจากโมเดล จะเห็นว่ารหัสที่ได้จากโมเดลแบบคงที่จะไม่เปลี่ยนแปลงตลอดเวลาของการเข้ารหัส การอัด/ขยายข้อมูลด้วยโมเดลแบบคงที่แสดงดังรูปที่ 3.1 โมเดลเชิงสถิติแบบคงที่ไม่สามารถใช้ตารางสากลที่จัดเก็บรหัสของสัญลักษณ์แต่ละตัวเพื่อใช้กับทุกแฟ้มข้อมูลได้ เนื่องจากข้อความของแฟ้มข้อมูลหนึ่งอาจมีความน่าจะเป็นของสัญลักษณ์แต่ละตัวไม่ตรงกับความน่าจะเป็นในตารางสากล จึงทำให้ข้อความที่อัดข้อมูลแล้วมีขนาดใหญ่กว่าข้อความต้นฉบับได้ ดังนั้นทุกแฟ้มข้อมูลที่ต้องการอัดข้อมูลด้วยโมเดลเชิงสถิติแบบคงที่จะต้องสร้างตารางจัดเก็บรหัสของตนเองทุกครั้ง นอกจากนี้ตารางดังกล่าวต้องจัดเก็บพร้อมกับแฟ้มข้อมูลที่อัดข้อมูลแล้วเพื่อให้ตัวถอดรหัสใช้ในการขยายข้อมูลอีกด้วย

โมเดลที่สร้างรหัสจากความน่าจะเป็นของสัญลักษณ์แต่ละตัวเรียกว่าโมเดลลำดับ 0 (order-0 model) ซึ่งใช้หน่วยความจำขนาดมากสุด 256 ไบต์ โมเดลที่สร้างรหัสจากความน่าจะเป็นที่จะพบคู่ของสัญลักษณ์เรียกว่า โมเดลลำดับ 1 (order-1 model) ซึ่งใช้หน่วยความจำขนาดมากสุด 256x256 = 65536 ไบต์ ดังนั้นโมเดลลำดับ n สร้างรหัสจากความน่าจะเป็นที่จะพบสัญลักษณ์ n ตัวซึ่งใช้หน่วยความจำขนาดมากสุด 256^n ไบต์ พบว่าการอัดข้อมูลที่มีประสิทธิภาพสูงขึ้นควรใช้โมเดลลำดับสูงขึ้น แต่ต้องใช้หน่วยความจำมากขึ้นตามไปด้วย นอกจากนี้รหัสจากโมเดลต้องจัดเก็บพร้อมแฟ้มข้อมูลที่อัดข้อมูลแล้วเพื่อใช้ในการขยายข้อมูล จึงทำให้แฟ้มข้อมูลดังกล่าวมีขนาดใหญ่ไปด้วย ดังนั้นแฟ้มข้อมูลที่อัดข้อมูลด้วยโมเดลลำดับสูงอาจมีขนาดใหญ่กว่าแฟ้มข้อมูลเดิมได้ การอัดข้อมูลด้วยโมเดลแบบคงที่จึงไม่นิยมใช้โมเดลลำดับสูง

1.1.2 โมเดลเชิงสถิติแบบปรับได้ (adaptive statistical model) โมเดลแบบนี้จะทำงานเพียงรอบเดียวโดยอ่านข้อความเข้ามาแล้วเข้ารหัสสัญลักษณ์ด้วยโมเดลที่มีอยู่ พร้อมทั้งปรับปรุงโมเดลพร้อมกันไป การอัด/ขยายข้อมูลด้วยโมเดลแบบปรับได้แสดงดังรูปที่ 3.2



รูปที่ 3.2 การอัด/ขยายข้อมูลด้วยโมเดลเชิงสถิติแบบปรับได้

จะเห็นว่ามีขั้นตอนปรับปรุงโมเดลขึ้นมาหลังการเข้ารหัส/ถอดรหัสเสร็จแล้ว รหัสที่สร้างจากโมเดลแบบปรับได้จะไม่แน่นอนเพราะโมเดลเปลี่ยนแปลงไปตามลักษณะของข้อความที่อ่านเข้ามาแต่ละช่วง จึงไม่ต้องจัดเก็บรหัสจากโมเดลนี้พร้อมแฟ้มข้อมูลที่อัดข้อมูลแล้วเพื่อใช้ในการขยายข้อมูล การอัดข้อมูลด้วยโมเดลแบบปรับได้จึงสามารถใช้โมเดลลำดับสูงได้ ดังนั้นการอัดข้อมูลจึงมีประสิทธิภาพมากขึ้น ข้อเสียของโมเดลแบบปรับได้คือการอัดข้อมูลช่วงแรกไม่มีประสิทธิภาพมากนักเพราะไม่ทราบลักษณะของข้อมูล ถ้าแฟ้มข้อมูลมีขนาดน้อยจะทำให้การอัดข้อมูลไม่มีประสิทธิภาพเท่าที่ควร อย่างไรก็ตามการอัดข้อมูลนิยมใช้โมเดลแบบปรับได้มากกว่า

1.2 โมเดลที่ใช้พจนานุกรมอ้างอิง (dictionary-based model) การอัดข้อมูลด้วยโมเดลที่ใช้พจนานุกรมอ้างอิงจะอ่านข้อความเข้ามาแล้วค้นหาข้อความที่ปรากฏในพจนานุกรม (dictionary) ถ้าพบข้อความที่ซ้ำในพจนานุกรมจะส่งออกโทเค็น (token) ของข้อความนั้นในพจนานุกรมออกมาแทน โมเดลที่ใช้พจนานุกรมอ้างอิงแบ่งเป็น 2 แบบคือแบบคงที่และแบบปรับได้

1.2.1 โมเดลที่ใช้พจนานุกรมอ้างอิงแบบคงที่ (static dictionary-based model)เริ่มจากรอบแรกอ่านข้อความเข้ามาทั้งหมดแล้วสร้างพจนานุกรมแบบคงที่ ซึ่งเป็นรายการของข้อความที่ใช้อ้างอิง รอบสองจะเข้ารหัสข้อความด้วยโทเค็นของข้อความที่ซ้ำในพจนานุกรม จะเห็นว่าพจนานุกรมจะไม่เปลี่ยนแปลงตลอดตลอดเวลาของการเข้ารหัส ข้อเสียของโมเดลที่ใช้พจนานุกรมอ้างอิงแบบคงที่เช่นเดียวกับโมเดลเชิงสถิติแบบคงที่ กล่าวคือต้องจัดเก็บพจนานุกรม

พร้อมกับแฟ้มข้อมูลที่อัดข้อมูลแล้วเพื่อให้ตัวถอดรหัสใช้ในการขยายข้อมูลอีกด้วย การอัดข้อมูลจะไม่ใช้โมเดลนี้

1.2.2 โมเดลที่ใช้พจนานุกรมอ้างอิงแบบปรับได้ (adaptive dictionary-based model) โมเดลนี้จะเริ่มทำงานโดยใช้พจนานุกรมว่างๆหรือพจนานุกรมเริ่มต้น (default dictionary) จากนั้นอ่านข้อความเข้ามา ถ้าข้อความดังกล่าวซ้ำในพจนานุกรมจะเข้ารหัสด้วยโทเค็นของข้อความนั้นในพจนานุกรม พร้อมทั้งเพิ่มข้อความเหล่านั้นในพจนานุกรมให้เพิ่มขึ้นเรื่อยๆ โมเดลจึงเปลี่ยนแปลงตลอดเวลา โมเดลแบบนี้จึงไม่ต้องจัดเก็บพจนานุกรมพร้อมกับแฟ้มข้อมูลที่อัดข้อมูลแล้วเพื่อใช้ในการขยายข้อมูล การอัดข้อมูลช่วงแรกไม่มีประสิทธิภาพมากนักเพราะไม่ทราบลักษณะของข้อมูล ถ้าแฟ้มข้อมูลมีขนาดน้อยจะทำให้การอัดข้อมูลไม่มีประสิทธิภาพเท่าที่ควร การอัดข้อมูลในปัจจุบันจะใช้โมเดลนี้เพราะเข้าใจง่าย ทำงานเร็ว และสามารถพัฒนาให้อัดข้อมูลอย่างมีประสิทธิภาพมากขึ้นเรื่อยๆ

2. ขั้นตอนวิธีการอัดข้อมูล (data-compression algorithm)

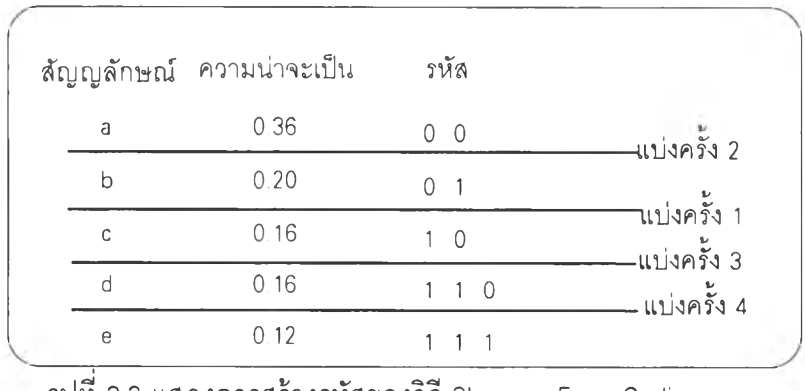
ขั้นตอนวิธีการอัดข้อมูลประกอบด้วยขั้นตอนการเข้ารหัส (encode) และขั้นตอนการถอดรหัส (decode) ขั้นตอนการเข้ารหัสเป็นการนำรหัสจากโมเดลมาแทนที่สัญลักษณ์เดิม และขั้นตอนการถอดรหัสเป็นการนำสัญลักษณ์เดิมของรหัสจากโมเดลกลับคืน ระยะแรกขั้นตอนวิธีการอัดข้อมูลจะใช้โมเดลเชิงสถิติ เช่นวิธี Huffman Coding, Arithmetic Coding เป็นต้น ต่อมาในปีค.ศ. 1977 และ 1978 Jacob Ziv และ Abraham Limpel ได้ปฏิวัติขั้นตอนวิธีการอัดข้อมูลโดยเสนอการอัดข้อมูลด้วยโมเดลที่ใช้พจนานุกรมอ้างอิงแบบปรับได้ 2 วิธีคือวิธี LZ77 และวิธี LZ78 วิธีดังกล่าวเข้าใจง่าย ทำงานเร็ว ใช้หน่วยความจำน้อย สามารถพัฒนาเทคนิคต่างๆเพื่อเพิ่มประสิทธิภาพการอัดข้อมูล วิธี LZ77 ได้รับการพัฒนาให้ใช้ในโปรแกรมอัดข้อมูล PKZIP และ LHarc ต่อมา Terry Welch ได้พัฒนาต่อจากวิธี LZ78 ได้เป็นวิธี LZW แล้วนำไปใช้ในโปรแกรมอัดข้อมูล COMPRESS ของยูนิกซ์จนวิธีนี้ได้รับความนิยมมาก วิธีลงรหัสมีหลายวิธีดังนี้ (Mark Nelson, 1991)

2.1 Shannon-Fano Coding เป็นแนวความคิดของ Claude Shannon และ R.M. Fano การอัดข้อมูลวิธีนี้จะทำงาน 2 รอบ รอบแรกอ่านข้อความเข้ามาทั้งหมดเพื่อหาความน่าจะ

เป็นของสัญลักษณ์แต่ละตัวแล้วสร้างรหัส รอบสองจะอ่านข้อความเข้ามาทีละสัญลักษณ์แล้วเข้ารหัสด้วยรหัสที่สร้างจากรอบแรก การสร้างรหัสในรอบแรกทำดังนี้

1. อ่านข้อความทั้งหมดแล้วหาความน่าจะเป็นของสัญลักษณ์แต่ละตัว
2. เรียงสัญลักษณ์ตามความน่าจะเป็นจากมากไปน้อย
3. แบ่งสัญลักษณ์เป็น 2 กลุ่ม โดยให้ผลรวมความน่าจะเป็นของแต่ละกลุ่มมีค่าเท่ากันหรือใกล้เคียงกัน
4. ให้กลุ่มแรกมีค่าบิตเป็น 0 และกลุ่มหลังมีค่าบิตเป็น 1 ดังนั้นกลุ่มแรกจะมีรหัสขึ้นต้นด้วย 0 และกลุ่มหลังจะมีรหัสขึ้นต้นด้วย 1
5. แต่ละกลุ่มทำซ้ำข้อ 3 และข้อ 4 แล้วนำค่าบิตไปต่อท้ายรหัส ทำเช่นนี้จนแต่ละกลุ่มมีสมาชิก 1 ตัว จะได้รหัสของสัญลักษณ์แต่ละตัว

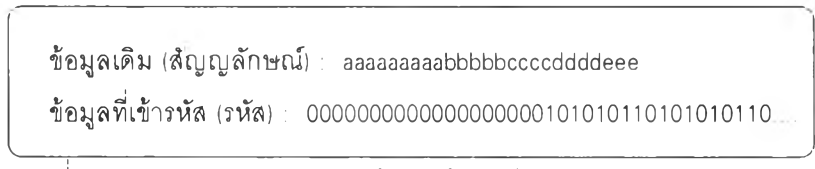
ตัวอย่างข้อความเดิมคือ aaaaaaaaabbbbbccccddddeee รอบแรกอ่านข้อความทั้งหมดแล้วหาความน่าจะเป็นของสัญลักษณ์แต่ละตัว จากนั้นเรียงสัญลักษณ์ตามความน่าจะเป็นจากมากไปน้อยแล้วแบ่งเป็น 2 กลุ่มโดยให้ผลรวมความน่าจะเป็นของแต่ละกลุ่มมีค่าเท่ากันหรือใกล้เคียงกัน จากนั้นให้กลุ่มแรกมีค่าบิตเป็น 0 และกลุ่มหลังมีค่าบิตเป็น 1 จากนั้นทำเช่นเดิมจนแต่ละกลุ่มมีสมาชิกเป็น 1 จะได้รหัสสำหรับใช้แทนสัญลักษณ์แต่ละตัว แสดงดังรูปที่ 3.3



รูปที่ 3.3 แสดงการสร้างรหัสของวิธี Shannon-Fano Coding

รอบสองอ่านข้อความเข้ามาทีละสัญลักษณ์แล้วเข้ารหัสด้วยรหัสที่สร้างจากรอบแรกดังรูปที่ 3.4 จะเห็นว่าข้อความเดิมใช้รหัสแอสกีขนาด 8 บิตแทนสัญลักษณ์แต่ละตัว ข้อความเดิมจึงยาว 200 บิต เมื่ออัดข้อมูลด้วยรหัสใหม่จะใช้เพียง 57 บิตเท่านั้น ต่อมามีวิธี

Huffman Coding ที่อัดข้อมูลได้อย่างมีประสิทธิภาพมากกว่า วิธี Shannon-Fano Coding จึงไม่ได้รับความนิยม



รูปที่ 3.4 แสดงตัวอย่างการอัดข้อมูลด้วยวิธี Shannon-Fano Coding

2.2 Huffman Coding D.A. Huffman ตีพิมพ์ "A Method for the Construction of Minimun Redundancy Codes" ในปีค.ศ. 1952 นิยมเรียกว่าวิธี Huffman Coding เป็นวิธีอัดข้อมูลด้วยโมเดลเชิงสถิติ โดยเข้ารหัสสัญลักษณ์ที่มีความน่าจะเป็นสูงด้วยรหัสที่มีจำนวนบิตน้อย และเข้ารหัสสัญลักษณ์ที่มีความน่าจะเป็นต่ำด้วยรหัสที่มีจำนวนบิตมาก รหัสที่ได้จึงมีความยาวไม่คงที่ วิธี Huffman Coding จะใช้โมเดลที่เป็นตารางระหว่างสัญลักษณ์และรหัส ทุกแฟ้มข้อมูลที่ต้องการอัด/ขยายข้อมูลโดยใช้วิธี Huffman Coding จะต้องสร้างรหัสของตนเองทุกครั้ง ดังนั้นรหัสดังกล่าวต้องจัดเก็บพร้อมแฟ้มข้อมูลที่อัดข้อมูลแล้วเพื่อให้ตัวถอดรหัสใช้ขยายข้อมูลต่อไป

การอัดข้อมูลวิธีนี้จะทำงาน 2 รอบ รอบแรกอ่านข้อความเข้ามาทั้งหมดเพื่อหาความน่าจะเป็นของสัญลักษณ์แต่ละตัวแล้วสร้างรหัส รอบสองจะอ่านข้อความเข้ามาทีละสัญลักษณ์แล้วเข้ารหัสด้วยรหัสที่สร้างจากรอบแรก การสร้างรหัสในรอบแรกทำดังนี้

1. อ่านข้อความทั้งหมดแล้วหาความน่าจะเป็นของสัญลักษณ์แต่ละตัว

2 สร้างต้นไม้แบบทวิภาคโดยใช้ความน่าจะเป็นข้างต้นเพื่อให้ได้รหัสที่สั้นที่สุด ขั้นตอนการสร้างต้นไม้แบบทวิภาคดังนี้

2.1 สร้างบัพใบที่บรรจุสัญลักษณ์แต่ละตัวพร้อมทั้งความน่าจะเป็น ดังรูปที่ 3.6 ก

2.2 นำ 2 บัพใบที่มีความน่าจะเป็นน้อยที่สุดมาสร้างบัพใบแม่ที่มีความน่าจะเป็นเท่ากับผลรวมของความน่าจะเป็นทั้ง 2 บัพใบ ดังรูปที่ 3.6 ข

2.3 ทำซ้ำข้อ 2.1 และ 2.2 จนเหลือเพียงบัพใบเดียว แล้วกำหนดให้กิ่งซ้ายของแต่ละบัพมีค่า 0 และกิ่งขวาของแต่ละบัพมีค่า 1 ดังรูปที่ 3.6 ค

3 จากต้นไม้แบบทวิภาคจะสร้างรหัสของสัญลักษณ์แต่ละตัว

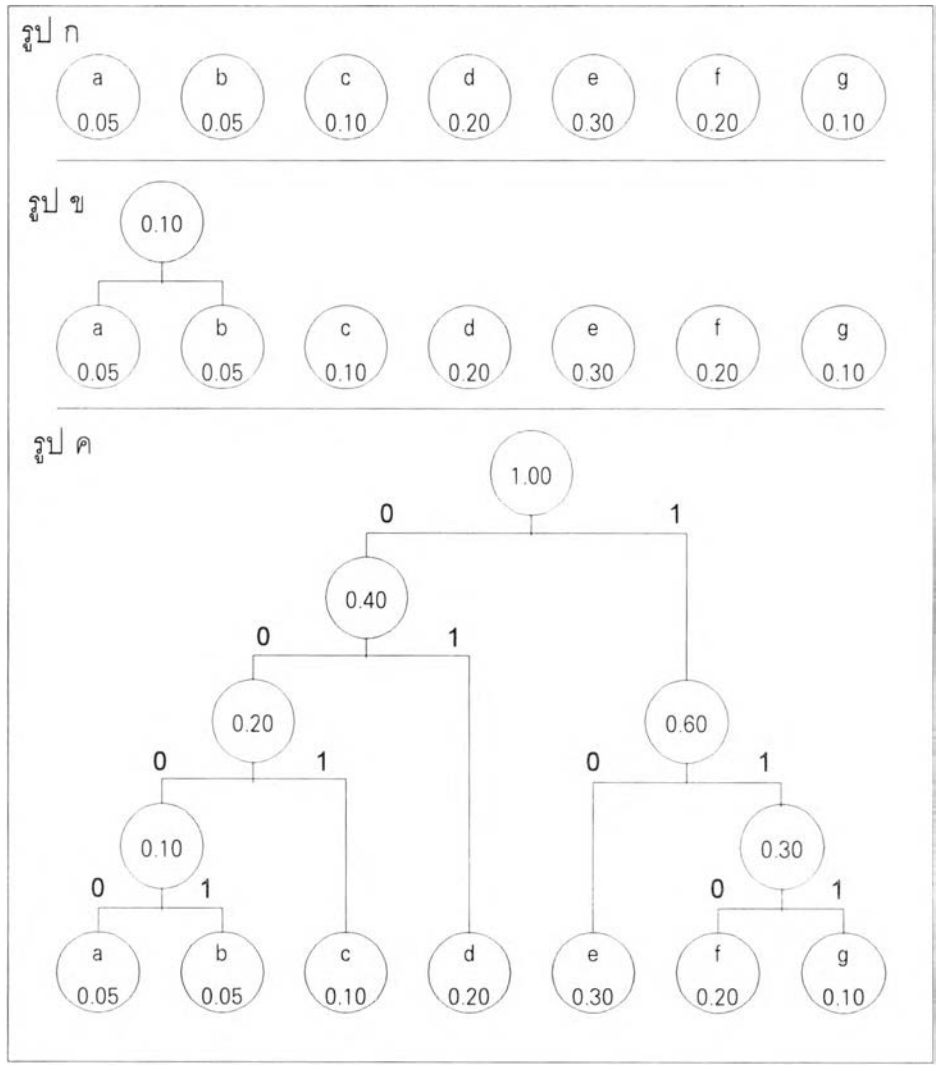
ตัวอย่าง ข้อมูลเดิมคือ abccddddeeeeeeffffgg รอบแรกหาความน่าจะเป็นของสัญลักษณ์แต่ละตัวดังรูปที่ 3.5 จากนั้นสร้างต้นไม้แบบทวิภาคโดยใช้ความน่าจะเป็นดังรูปที่ 3.6 แล้วสร้างรหัสของสัญลักษณ์แต่ละตัวดังรูปที่ 3.7 รอบสองอ่านข้อความเข้ามาทีละสัญลักษณ์แล้วเข้ารหัสด้วยรหัสที่สร้างจากรอบแรกดังรูปที่ 3.8

สัญญลักษณ์	ความน่าจะเป็น
a	0.05
b	0.05
c	0.10
d	0.20
e	0.30
f	0.20
g	0.10

รูปที่ 3.5 แสดงความน่าจะเป็นของสัญลักษณ์แต่ละตัว

วิธี Huffman Coding ข้างต้นใช้โมเดลเชิงสถิติแบบคงที่ลำดับ 0 จึงต้องจัดเก็บรหัสของสัญลักษณ์แต่ละตัวที่มีขนาดสูงสุด 256 ไบต์ลงในแฟ้มข้อมูลที่อัดข้อมูลด้วย ส่งผลให้แฟ้มข้อมูลมีขนาดใหญ่ขึ้น นอกจากนี้การอัดข้อมูลที่มีประสิทธิภาพสูงขึ้นจำเป็นต้องใช้โมเดลลำดับสูงขึ้นด้วยเช่นกัน จึงทำให้แฟ้มข้อมูลมีขนาดใหญ่มากเกินไปเช่น โมเดลลำดับ 1 ต้องเก็บรหัสของสัญลักษณ์แต่ละคู่ที่มีขนาดสูงสุด 65536 ไบต์ เป็นต้น ต่อมาได้พัฒนาเทคนิคให้ Huffman Coding ใช้โมเดลเชิงสถิติแบบปรับได้ซึ่งไม่ต้องจัดเก็บรหัสของสัญลักษณ์ลงในแฟ้มข้อมูล จึงสามารถใช้โมเดลลำดับสูงได้ โมเดลเชิงสถิติแบบปรับได้จะทำงานเพียงรอบเดียว โดยใช้ความน่าจะเป็นของสัญลักษณ์ขณะนั้นสร้างต้นไม้ของรหัส ทำให้ต้นไม่เปลี่ยนแปลงตลอดเวลา โมเดลนี้ทำงานซับซ้อนกว่ามากจึงทำให้ทำงานช้ามากเช่นกัน

ถ้าความน่าจะเป็นของสัญลักษณ์แต่ละตัวไม่ถูกต้องอาจทำให้แฟ้มข้อมูลที่อัดแล้วมีขนาดใหญ่กว่าแฟ้มข้อมูลเดิมได้ รหัสมีความยาวไม่คงที่จึงทำให้การเข้ารหัส/ถอดรหัสทำงานช้า ถ้าแฟ้มข้อมูลที่อัดข้อมูลไว้เกิดผิดพลาดที่บิตใดบิตหนึ่งจะทำให้ไม่สามารถถอดรหัสส่วนที่เหลือได้ วิธี Huffman Coding ได้พัฒนาไปใช้อย่างกว้างขวางทั้งในโปรแกรมอัดข้อมูล เครื่อง FAX และขั้นตอนการอัดข้อมูล JPEG



รูปที่ 3.6 แสดงต้นไม้แบบทวิภาคที่ใช้สร้างรหัส

สัญญลักษณ์	รหัส	สัญญลักษณ์	รหัส
a	0000	e	10
b	0001	f	110
c	001	g	111
d	01		

รูปที่ 3.7 แสดงรหัสของสัญลักษณ์แต่ละตัว

ข้อมูลเดิม (สัญญลักษณ์) : abccddddeeeeeeffffgg
ข้อมูลที่เข้ารหัส (รหัส) : 00000001001001010101011010101010110....

รูปที่ 3.8 แสดงตัวอย่างการอัดข้อมูลด้วยวิธี Huffman Coding

2.3 Arithmetic Coding เป็นวิธีอัดข้อมูลด้วยใช้โมเดลเชิงสถิติ วิธีนี้จะไม่แทนสัญลักษณ์แต่ละตัวด้วยรหัสเช่นเดียวกับวิธี Shannon-Fano Coding, Huffman Coding แต่วิธีนี้จะแทนข้อความทั้งหมดด้วยรหัสซึ่งเป็นเลขทศนิยมที่มีค่าน้อยกว่า 1 และมากกว่าหรือเท่ากับ 0 รหัสซึ่งเป็นตัวเลขนี้จะมีจำนวนบิตของทศนิยมมากขึ้นตามความยาวของข้อความ

วิธีนี้ทำงานสองรอบโดยรอบแรกจะอ่านข้อความทั้งหมดเพื่อหาความน่าจะเป็นของสัญลักษณ์แต่ละตัว แล้วคำนวณเป็นช่วงความน่าจะเป็นสะสมจาก 0 ถึง 1 ดังนั้นสัญลักษณ์ที่มีความน่าจะเป็นสูงจะมีช่วงความน่าจะเป็นสะสมกว้างกว่าสัญลักษณ์ที่มีความน่าจะเป็นต่ำ รอบสองจะอ่านข้อความเข้ามาทีละสัญลักษณ์แล้วเข้ารหัสโดยใช้ช่วงความน่าจะเป็นสะสมมาเพิ่มจำนวนบิตของทศนิยม ดังนั้นข้อความที่ยาวมากจะเข้ารหัสเป็นตัวเลขที่มีจำนวนบิตของทศนิยมมากขึ้นตามไปด้วย วิธีการเข้ารหัสโดยใช้ช่วงความน่าจะเป็นสะสมมาเพิ่มจำนวนบิตของทศนิยมดังนี้

1. ให้ n เป็นจำนวนสัญลักษณ์ทั้งหมดของข้อความ
2. ให้ high = 1 และ low = 0
3. จาก 1 ถึง n ทำดังนี้
   3.1 range = high - low
   3.2 high = low + (range*ค่าสูงสุดของช่วงความน่าจะเป็นสะสม)
   3.3 low = low + (range*ค่าต่ำสุดของช่วงความน่าจะเป็นสะสม)

ตัวอย่าง ข้อมูลเดิมคือ 'BILL GATES' รอบแรกหาความน่าจะเป็นที่จะพบสัญลักษณ์แต่ละตัว แล้วเปลี่ยนเป็นความน่าจะเป็นสะสมของแต่ละสัญลักษณ์ที่มีค่าตั้งแต่ 0 ถึง 1 ดังรูปที่ 3.9 จากนั้นรอบสองจะอ่านข้อความเข้ามาทีละสัญลักษณ์แล้วเข้ารหัสโดยใช้ช่วงความน่าจะเป็นสะสมมาเพิ่มจำนวนบิตของทศนิยมตามวิธีการข้างต้น ดังรูปที่ 3.10 ข้อความ 'BILL GATES' เข้ารหัสเป็นตัวเลขทศนิยม 0.2572167752

สัญญลักษณ์	ความน่าจะเป็น	ช่วงความน่าจะเป็นสะสม
space	0.1	0.0 <= r < 0.1
A	0.1	0.1 <= r < 0.2
B	0.1	0.2 <= r < 0.3
E	0.1	0.3 <= r < 0.4
G	0.1	0.4 <= r < 0.5
I	0.1	0.5 <= r < 0.6
L	0.2	0.6 <= r < 0.8
S	0.1	0.8 <= r < 0.9
T	0.1	0.9 <= r < 1.0

รูปที่ 3.9 แสดงความน่าจะเป็นและช่วงความน่าจะเป็นสะสม

สัญญลักษณ์	low	high
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
space	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

รูปที่ 3.10 แสดงตัวอย่างการเข้ารหัสด้วยวิธี Arithmetic Coding

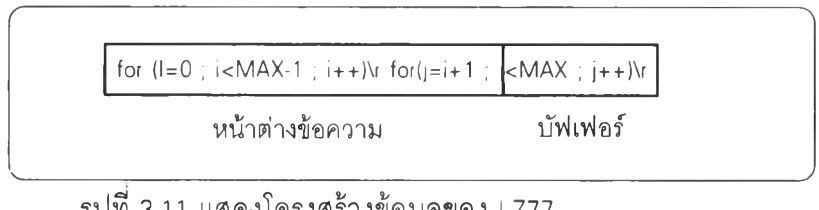
ตัวถอดรหัสจะใช้ช่วงความน่าจะเป็นสะสมนี้ขยายข้อมูลเช่นกัน ดังนั้นช่วงความน่าจะเป็นสะสมต้องจัดเก็บพร้อมแฟ้มข้อมูลด้วย ขั้นตอนการถอดรหัสจะทำงานย้อนกลับโดยถอดรหัสจากตัวเลขทศนิยมเป็นสัญลักษณ์แต่ละตัวตามเดิม วิธี Arithmetic Coding ทำงานซับซ้อนกว่า Huffman Coding จึงทำงานช้ากว่ามาก และวิธีนี้เข้ารหัสเป็นตัวเลขทศนิยมจึงมี

ปัญหาเรื่องความเที่ยงตรง (precision) ด้วย ระยะแรกการอัดข้อมูลจะใช้โมเดลเชิงสถิติเรื่อยมา จนปีค.ศ. 1977 และ 1978 Jacob Ziv และ Abraham Lempel ได้เสนอการอัดข้อมูลด้วยโมเดลที่ใช้พจนานุกรมอ้างอิงแบบปรับได้ซึ่งได้รับความนิยมมากในเวลาต่อมา

2.4 LZ77 Jacob Ziv และ Abraham Lempel ตีพิมพ์ "A Universal Algorithm for Sequential Data Compression" ในปีค.ศ 1977 นิยมเรียกว่า LZ77 วิธีนี้ได้รับความนิยมเพราะเป็นวิธีที่ง่าย ทำงานเร็วและใช้หน่วยความจำน้อย โปรแกรมอัดข้อมูลที่นิยมกันเช่น PKZIP, LHarc พัฒนามาจากวิธี LZ77 วิธีนี้ใช้โมเดลที่ใช้พจนานุกรมอ้างอิงแบบปรับได้ โดยใช้ข้อความที่อ่านเข้ามาก่อนเป็นพจนานุกรมเพื่อเข้ารหัสข้อความที่ตามมาทีหลัง LZ77 มีโครงข้อมูลใช้งานที่แบ่งเป็น 2 ส่วนคือ หน้าต่างข้อความ (text window) และบัฟเฟอร์ (buffer) หน้าต่างข้อความมีขนาดใหญ่ 2 ถึง 16 กิโลไบต์สำหรับข้อความที่อ่านเข้ามาก่อนเพื่อใช้เป็นพจนานุกรม และบัฟเฟอร์มีขนาดเล็ก 16 ถึง 64 ไบต์สำหรับข้อความตามมาทีหลังเพื่อใช้เข้ารหัส ประสิทธิภาพการอัดข้อมูลด้วย LZ77 ขึ้นอยู่กับขนาดของหน้าต่างข้อความที่ใช้เป็นพจนานุกรม

วิธีนี้จะทำงานเพียงรอบเดียวโดย LZ77 จะอ่านข้อมูลเข้ามาในหน้าต่างข้อความและบัฟเฟอร์ จากนั้นค้นหาข้อความในบัฟเฟอร์ที่ซ้ำในหน้าต่างข้อความ เมื่อพบข้อความที่ซ้ำกันจะเข้ารหัสโดยแทนด้วยโทเค็น (token) ซึ่งประกอบด้วย 1 ตำแหน่งของข้อความซ้ำในพจนานุกรม 2. ความยาวข้อความซ้ำ 3 สัญลักษณ์ตัวแรกในบัฟเฟอร์ที่ถัดจากข้อความซ้ำ หลังจากแทนด้วยโทเค็นแล้วจะเลื่อนข้อความในหน้าต่างข้อความออกไป แล้วอ่านข้อความใหม่เข้ามาในบัฟเฟอร์ จากนั้นค้นหาข้อความซ้ำแล้วเข้ารหัสต่อไป

ตัวอย่างโครงสร้างข้อมูลหน้าต่างข้อความและบัฟเฟอร์ ดังรูปที่ 3.11 จะเห็นว่าข้อความในบัฟเฟอร์ "<MAX" ซ้ำในหน้าต่างข้อความที่ตำแหน่ง 14 จำนวน 4 ตัวและสัญลักษณ์ถัดไปในบัฟเฟอร์คือ space จึงเข้ารหัสเป็นโทเค็น (14, 4,) จากนั้นเลื่อนข้อความออกไป 5 ตัว แล้วอ่านข้อความใหม่เข้ามา จากนั้นหาข้อความซ้ำต่อไป



รูปที่ 3.11 แสดงโครงสร้างข้อมูลของ LZ77

การถอดรหัสง่ายมากเนื่องจากไม่ต้องหาข้อความซ้ำ เพียงแต่อ่านข้อมูลเข้ามาในหน้าต่างข้อความและบัฟเฟอร์ จากนั้นแทนที่โทเค็นด้วยข้อความในหน้าต่างข้อความ แล้วเลื่อนข้อความออกไป อ่านข้อมูลใหม่เข้ามาแล้วถอดรหัสเรื่อยไปจนจบ จะเห็นว่าพจนานุกรมเป็นข้อความที่อ่านเข้ามาก่อนจึงไม่ต้องจัดเก็บพจนานุกรมในแฟ้มข้อมูลออก

การอัดข้อมูลใช้เพียงข้อความในหน้าต่างข้อความเท่านั้นเป็นพจนานุกรม การอัดข้อมูลให้มีประสิทธิภาพสูงจึงต้องใช้หน้าต่างข้อความขนาดใหญ่ อย่างไรก็ตามหน้าต่างข้อความไม่สามารถเพิ่มขนาดได้มากนักเนื่องจากมีผลต่อขนาดแฟ้มข้อมูลและเวลาทำงาน เช่น หน้าต่างข้อความขนาด 4 กิโลไบต์จะใช้ 12 บิตแทนตำแหน่งข้อความซ้ำในพจนานุกรม เมื่อหน้าต่างข้อความเพิ่มขนาดเป็น 64 กิโลไบต์จะใช้ 16 บิตแทนตำแหน่งข้อความซ้ำในพจนานุกรม ดังนั้นโทเค็นจึงมีขนาดใหญ่ตามไปด้วย และเวลาที่ใช้หาข้อความซ้ำในพจนานุกรมจะเป็น 16 เท่า จึงส่งผลให้ขนาดแฟ้มข้อมูลใหญ่และใช้เวลาทำงานมาก วิธีนี้เข้ารหัสข้อความไม่มีประสิทธิภาพมากนักเนื่องจากบัฟเฟอร์มีขนาดจำกัด เช่นบัฟเฟอร์ขนาด 16 ไบต์จะสามารถเข้ารหัสข้อความได้ยาวไม่เกิน 16 ตัวอักษร แต่ข้อความที่ซ้ำกันอาจยาวเกิน 16 ไบต์ได้ ข้อความที่เกินนั้นจะไม่ได้เข้ารหัสด้วย นอกจากนี้ LZ77 ยังมีปัญหาในการเข้ารหัสข้อความที่ไม่ซ้ำในพจนานุกรม จะต้องเข้ารหัสข้อความนั้นทีละสัญลักษณ์เป็นโทเค็นเช่นเดิม เช่นหน้าต่างข้อความขนาด 4 กิโลไบต์และบัฟเฟอร์ขนาด 16 ไบต์จะใช้โทเค็น 24 บิต (ตำแหน่งข้อความในพจนานุกรม 12 บิต, ความยาวข้อความ 4 บิต, สัญลักษณ์ถัดไป 8 บิต) จึงแทนที่สัญลักษณ์แต่ละตัวขนาด 8 บิตด้วยโทเค็นขนาด 24 บิต ดังนั้นการอัดข้อมูลที่มีข้อความไม่ค่อยซ้ำกันจะทำให้แฟ้มข้อมูลออกมีขนาดใหญ่กว่าแฟ้มข้อมูลเดิมได้

2.5 LZ78 Jacob Ziv และ Abraham Lempel ได้ตีพิมพ์ "Compression of Individual Sequence via Variable-Rate Coding" ในปีค.ศ 1978 นิยมเรียกว่า LZ78 เป็นวิธีที่ง่าย ทำงานเร็วและใช้หน่วยความจำน้อย วิธีอัดข้อมูลที่ใช้ในโปรแกรมอัดข้อมูล COMPRESS ของยูนิกซ์คือ LZW ซึ่งพัฒนาต่อมาจากวิธี LZ78 นี้ LZ78 ใช้โมเดลที่ใช้พจนานุกรมอ้างอิงแบบปรับได้ที่ต่างไปจาก LZ77 กล่าวคือ LZ77 ใช้ข้อความในหน้าต่างข้อความเป็นพจนานุกรม แต่ LZ78 จะใช้ข้อความทั้งหมดที่อ่านเข้ามาก่อนเป็นพจนานุกรม ประสิทธิภาพการอัดข้อมูลด้วย LZ78 ขึ้นอยู่กับขนาดของพจนานุกรม

วิธีนี้จะทำงานเพียงรอบเดียว การเข้ารหัสเริ่มจากพจนานุกรมว่าง จากนั้นอ่านข้อความเข้ามาทีละสัญลักษณ์แล้วหาในพจนานุกรม กรณีที่พบสัญลักษณ์นั้นใน

พจนานุกรมจะอ่านเพิ่มทีละสัญลักษณ์จนกว่าจะได้ข้อความซ้ำในพจนานุกรม จากนั้นจะเข้ารหัสโดยแทนข้อความซ้ำนั้นด้วยโทเค็น ซึ่งประกอบด้วย 1. รหัสของข้อความในพจนานุกรม 2. สัญลักษณ์ตัวถัดจากข้อความซ้ำ กรณีไม่พบสัญลักษณ์นั้นในพจนานุกรมแสดงว่าไม่มีข้อความซ้ำจะเข้ารหัสด้วยโทเค็นที่มีรหัส 0 และสัญลักษณ์นั้น หลังจากเข้ารหัสด้วยโทเค็นแล้วจะนำสัญลักษณ์นั้นเพิ่มเข้าไปในพจนานุกรมเป็นข้อความใหม่พร้อมรหัสใหม่ไว้ใช้ต่อไป ทำเช่นนี้เรื่อยไปจนจบ พจนานุกรมที่ใช้เป็นต้นไม้แบบหลายทาง (multiway tree) ที่มีกิ่งที่เป็นไปได้ตามจำนวนตัวอักษร 256 กิ่ง และแต่ละกิ่งสามารถแตกย่อยไปได้อีกแล้วแต่ข้อความที่สร้างเป็นพจนานุกรม ขนาดของพจนานุกรมขึ้นอยู่กับขนาดรหัสที่ใช้ในโทเค็นเช่นรหัสขนาด 16 บิตสามารถสร้างพจนานุกรมที่มี 65536 บัพใบ โดยแต่ละบัพใบแทนสัญลักษณ์ 1 ตัว

ข้อมูลเดิม (สัญญลักษณ์) : a b aa ba c cb baa baab bb bbb

ข้อมูลที่เข้ารหัส (โทเค็น) : (0,a) (0,b) (1,a) (2,a) (0,c) (5,b) (4,a) (7,b) (2,b) (9,b)

พจนานุกรม :

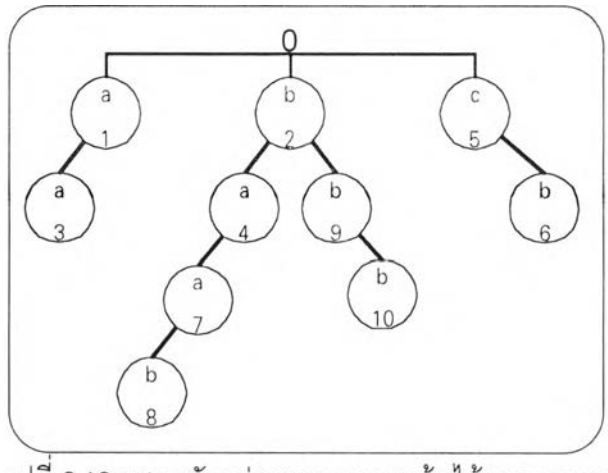
รหัส	กลุ่มข้อความ
0	null
1	a
2	b
3	aa
4	ba
5	c
.....	.....

รูปที่ 3.12 แสดงตัวอย่างการเข้ารหัสด้วยวิธี LZ78

ตัวอย่างข้อมูลเดิมคือ abaabaccbbaabaabbbbbb การเข้ารหัสด้วยวิธี LZ78 แสดงดังรูปที่ 3.12 พจนานุกรมต้นไม้แบบหลายทางของตัวอย่างนี้แสดงดังรูปที่ 3.13 เริ่มจากอ่าน a ซึ่งไม่พบในพจนานุกรมจะเข้ารหัสด้วยโทเค็น (0,a) พร้อมทั้งเพิ่ม a ในพจนานุกรมให้มีรหัส 1, ต่อมาอ่าน b ซึ่งไม่พบในพจนานุกรมจะเข้ารหัสด้วยโทเค็น (0,b) พร้อมทั้งเพิ่ม b ในพจนานุกรมให้มีรหัส 2, ต่อมาอ่าน a ซึ่งพบในพจนานุกรมจึงอ่านเพิ่มอีกตัวเป็น a จะเข้ารหัสด้วยโทเค็น (1,a) พร้อมทั้งเพิ่ม aa ในพจนานุกรมให้มีรหัส 3, ต่อมาอ่าน b ซึ่งพบในพจนานุกรมจึงอ่านเพิ่มอีกตัวเป็น a จะเข้ารหัสด้วยโทเค็น (2,a) พร้อมทั้งเพิ่ม ba ในพจนานุกรมให้มีรหัส 4 เรื่อยไปจนจบ

พจนานุกรมจะไม่จัดเก็บรวมกับแฟ้มข้อมูลที่อัดข้อมูลแล้ว การถอดรหัสจะเริ่มจากพจนานุกรมว่าง จากนั้นอ่านข้อมูลโทเค็นซึ่งประกอบด้วยรหัสและสัญลักษณ์เข้ามาแล้ว

ถอดรหัส ถ้ารหัสเป็น 0 จะถอดรหัสเป็นสัญลักษณ์ในโทเค็น แล้วเพิ่มสัญลักษณ์นั้นและรหัสถัดไปในพจนานุกรม ถ้ารหัสไม่เป็น 0 จะหารหัสนั้นในพจนานุกรมแล้วถอดรหัสเป็นข้อความออกมา พร้อมทั้งเพิ่มสัญลักษณ์ในโทเค็นและรหัสถัดไปในพจนานุกรมด้วย จากนั้นกลับไปอ่านข้อมูลเข้ามาใหม่แล้วทำซ้ำเดิมเรื่อยไปจนจบ



รูปที่ 3.13 แสดงตัวอย่างพจนานุกรมต้นไม้แบบหลายทาง

2.6 LZW การอัดข้อมูลด้วยวิธี LZ77 และ LZ78 ได้รับการพัฒนาปรับปรุงเทคนิคต่างๆเรื่อยมา จนกระทั่งปีค.ศ. 1984 Terry Welch ได้ตีพิมพ์ "A Technique for High-Performance Data Compression" ซึ่งพัฒนาต่อจาก LZ78 ต่อมานิยมเรียกวิธีนี้ว่า LZW เป็นการอัดข้อมูลด้วยโมเดลที่ใช้พจนานุกรมอ้างอิงเช่นกันซึ่งเป็นวิธีที่ง่าย ทำงานเร็วและใช้หน่วยความจำน้อย โปรแกรม COMPRESS ของยูนิกซ์ได้พัฒนามาจากวิธีนี้ ข้อแตกต่างของ LZW จาก LZ78 คือ LZ78 เริ่มทำงานด้วยพจนานุกรมว่าง แต่ LZW เริ่มทำงานด้วยพจนานุกรมเริ่มต้น (default dictionary) ซึ่งเป็นตัวอักษรจำนวน 256 ตัวที่มีรหัสเป็นตัวเลข 0 ถึง 255 นอกจากนี้การเข้ารหัสของ LZ78 เป็นโทเค็นซึ่งประกอบด้วยรหัสและสัญลักษณ์ แต่การเข้ารหัสของ LZW มีเพียงรหัสซึ่งเป็นตัวเลขเท่านั้น เพราะ LZW มีพจนานุกรมเริ่มต้นที่เป็นตัวอักษรทั้งหมดจึงสามารถเข้ารหัสเป็นรหัสตัวเลขเสมอ ประสิทธิภาพการอัดข้อมูลด้วย LZW ขึ้นอยู่กับขนาดของพจนานุกรม

วิธีนี้จะทำงานเพียงรอบเดียวเช่นกัน เริ่มทำงานโดยกำหนดพจนานุกรมเริ่มต้นเป็นตัวอักษร 256 ตัวให้มีค่ารหัส 0 ถึง 255 ดังนั้นรหัสถัดไปที่จะเพิ่มในพจนานุกรมคือ 256 พจนานุกรมที่ใช้เป็นต้นไม้แบบหลายทางเช่นเดียวกับ LZ78 ดังนั้นขนาดของพจนานุกรมขึ้นอยู่กับขนาดรหัสที่ใช้เช่น รหัสขนาด 16 บิตสามารถสร้างพจนานุกรมที่มี 65536 บัพใบ ซึ่งแต่ละบัพใบแทนสัญลักษณ์ 1 ตัว การเข้ารหัสทำดังนี้

1. อ่านสัญลักษณ์ตัวแรกให้ old

2. อ่านสัญลักษณ์ตัวถัดไปให้ char

3. นำข้อความ (old และ char) ไปหาในพจนานุกรม ถ้าพบข้อความนั้นในพจนานุกรมจะนำ char มาต่อท้าย old แล้วอ่านสัญลักษณ์ถัดไปให้ char จากนั้นนำข้อความ (old และ char) ไปหาในพจนานุกรมอีก ทำเช่นนี้เรื่อยไปจนกว่าจะไม่พบข้อความ (old และ char) ในพจนานุกรม

4. เมื่อไม่พบข้อความ (old และ char) ในพจนานุกรมแสดงว่าข้อความ old เท่านั้นที่ซ้ำในพจนานุกรม จะเข้ารหัสข้อความ old จากพจนานุกรม จากนั้นเพิ่มข้อความใหม่ (old และ char) และรหัสถัดไปในพจนานุกรม

5. ให้ old = char แล้ววนกลับไปทำงานข้อ 2 ทำเช่นนี้เรื่อยไปจนจบ

ข้อมูลเดิม (สัญญลักษณ์) a b c ab ca bc cab c e abc f

ข้อมูลที่เข้ารหัส (รหัส) : 97 98 99 256 258 257 260 99 101 25 102

old	char	พจนานุกรม	
a	b	ab	256
b	c	bc	257
c	a	ca	258
ab	c	abc	259
ca	b	cab	260
bc	c	bcc	261
...	...	...	

รูปที่ 3.14 แสดงตัวอย่างการเข้ารหัสของ LZW

ตัวอย่างข้อมูลเดิมคือ abcabcabccabceabcf การเข้ารหัสด้วยวิธี LZW แสดงดังรูปที่ 3.14 เริ่มจากอ่าน a อ่าน b เมื่อหา ab ในพจนานุกรมไม่พบจึงเข้ารหัส a แล้วเพิ่ม ab และรหัสถัดไปในพจนานุกรม, อ่าน c เมื่อหา bc ในพจนานุกรมซึ่งไม่พบจึงเข้ารหัส b แล้วเพิ่ม bc และรหัสถัดไปในพจนานุกรม, อ่าน a แล้วหา ca ในพจนานุกรมซึ่งไม่พบจึงเข้ารหัส c แล้วเพิ่ม ca และรหัสถัดไปในพจนานุกรม, อ่าน b แล้วหา ab ในพจนานุกรมซึ่งพบจึงอ่านเพิ่ม c แล้วหา abc ในพจนานุกรมซึ่งไม่พบจึงเข้ารหัส ab แล้วเพิ่ม abc และรหัสถัดไปในพจนานุกรมทำเรื่อยไปจนจบ

พจนานุกรมจะไม่จัดเก็บรวมกับแฟ้มข้อมูลที่อัดข้อมูลแล้ว การถอดรหัสเริ่มทำงานโดยกำหนดพจนานุกรมเริ่มต้นเป็นตัวอักษร 256 ตัวให้มีค่ารหัส 0 ถึง 255 ดังนั้นรหัสถัดไป

ที่จะเพิ่มในพจนานุกรมคือ 256 การออกรหัสทำดังนี้

1. อ่านรหัสให้ old แล้วถอดรหัสจากพจนานุกรมได้ข้อความออกมา
2. อ่านรหัสต่อไปให้ new แล้วถอดรหัสจากพจนานุกรมได้ข้อความออกมา
3. เพิ่มข้อความใหม่ (ข้อความ old และสัญลักษณ์แรกของ new) และรหัสถัดไปในพจนานุกรม
4. ให้ old = new แล้ววนกลับไปทำงานข้อ 2 ทำเช่นนี้เรื่อยไปจนจบ

รูปที่ 3.15 แสดงตัวอย่างการถอดรหัสของ LZW

ข้อมูลที่เข้ารหัส (รหัส) : 97 98 99 256 258 257 260 99 101 25 102

ข้อมูลที่ถอดรหัส (สัญญลักษณ์) a b c ab ca bc cab c e abc f

old	new	พจนานุกรม	
a	b	ab	256
b	c	bc	257
c	256	ca	258
ab	258	abc	259
ca	257	cab	260
bc	260	bcc	261

ตัวอย่างข้อมูลที่เข้ารหัสคือ 97 98 99 256 258 257 260 99 101 259 102 การถอดรหัสด้วยวิธี LZW แสดงดังรูปที่ 3.15 เริ่มจากอ่าน 97 และ 98 แล้วถอดรหัสได้ a และ b แล้วเพิ่ม ab และรหัสถัดไปในพจนานุกรม, อ่าน 99 แล้วถอดรหัสได้ c แล้วเพิ่ม bc และรหัสถัดไปในพจนานุกรม, อ่าน 256 แล้วถอดรหัสได้ ab แล้วเพิ่ม ca และรหัสถัดไปในพจนานุกรม ทำเรื่อยไปจนจบ

<u>การเลือกขั้นตอนวิธีการอัดข้อมูล</u>

ขั้นตอนวิธีการอัดข้อมูลที่ดีควรทำงานได้เร็ว, อัดข้อมูลได้มาก และใช้หน่วยความจำน้อย Ian H. Witten, Alistair Moffat และ Timothy C. Bell ได้ทดลองเปรียบเทียบการอัดข้อมูลวิธีต่างๆ โดยเขียนด้วยภาษาซีและใช้คอมไพเลอร์เดียวกัน แล้วทดลองบนเครื่อง Sun SPARC 10 Model 512 วิธีการอัดข้อมูลที่ทำการทดลองแสดงดังตารางที่ 3.1 ผลการทดลองเปรียบเทียบความ

เร็วในการอัด/ขยายข้อมูล และอัตราการอัดข้อมูลโดยเฉลี่ยแสดงดังตารางที่ 3.2 (Ian H. Witten, Alistair Moffat and Timothy C. Bell, 1994)

วิธีอัดข้อมูล	คำอธิบาย
pack	Zero-order character-based, semi-static Huffman coder
cacm	Zero-order character-based, adaptive arithmetic coder
ppm	Variable-order character-based, adaptive arithmetic coder
dmc	Variable-order bit-based, adaptive arithmetic coder
huffword	Zero-order word-based, semi-static Huffman coder
grip-f	LZ77, semi-static Huffman coder, fast option
grip-b	LZ77, semi-static Huffman coder, best compression option
lzrwl	LZ77, binary coder, fast option
compress	LZ78/LZW, the Unix compress utility

ตารางที่ 3.1 แสดงวิธีอัดข้อมูลที่ทำการทดลอง

วิธีอัดข้อมูล	ความเร็วการอัดข้อมูล (เมกะไบต์/วินาที)	ความเร็วการขยายข้อมูล (เมกะไบต์/วินาที)	อัตราการอัดข้อมูลเฉลี่ย (%)
ไม่มีการอัดข้อมูล	500	500	100.0
dmc	2	2	32.2
ppm	6	5	29.3
cacm	6	5	61.2
pack	55	30	62.3
huffword	10	55	44.5
compress	30	65	45.5
gzip-b	9	110	33.7
gzip-f	25	120	35.1
lzrwl	90	200	55.7

ตารางที่ 3.2 ความเร็วในการอัด/ขยายข้อมูลและอัตราการอัดข้อมูลโดยเฉลี่ย

จากผลการทดลองจะเห็นว่า compress ซึ่งใช้วิธีอัดข้อมูล LZ78/LZW ไม่เป็นวิธีที่มีความเร็วในการอัด/ขยายข้อมูลสูงสุดหรือเป็นวิธีที่มีอัตราการอัดข้อมูลสูงสุด แต่เป็นวิธีที่มีความเร็วในการอัด/ขยายข้อมูลและอัตราการอัดข้อมูลที่เหมาะสมที่สุด นอกจากนี้วิธี LZ78/LZW อัดข้อมูลด้วยโมเดลที่ใช้พจนานุกรมอ้างอิงแบบปรับได้จึงใช้พื้นที่หน่วยความจำน้อยกว่าวิธีอื่น ดังนั้น LZW จึงเป็นวิธีอัดข้อมูลที่เหมาะสมสำหรับพัฒนาโปรแกรมอรรถประโยชน์สำหรับจัดการแฟ้มเอกสารของจุฬาจารึกทางด้านการอัดแฟ้มข้อมูล

การอัดข้อมูลแฟ้มข้อมูลจุฬาจารึก

แฟ้มข้อมูลจุฬาจารึกได้ออกแบบโครงสร้างแฟ้มข้อมูลให้จัดเก็บข้อมูลอย่างมีระเบียบ โดยจัดเก็บข้อมูลเป็นส่วนต่างๆ 10 ส่วนคือ ส่วน Header, ส่วน Style, ส่วน Section, ส่วน Font, ส่วน Picture, ส่วน Text, ส่วน Index, ส่วน Attribute, ส่วน Overlay Drawing และส่วน TTF Embed เมื่อจุฬาจารึกเปิดแฟ้มข้อมูลจะตรวจสอบว่าเป็นแฟ้มข้อมูลจุฬาจารึกหรือไม่ โดยตรวจสอบตัวระบุแฟ้มข้อมูล (0xA8CCE00) ที่ส่วน Header และตรวจสอบว่าโครงสร้างแฟ้มข้อมูลถูกต้องหรือไม่ โดยตรวจสอบจุดเริ่มต้นของแต่ละส่วน การอัดแฟ้มข้อมูลจุฬาจารึกจะไม่อัดข้อมูลส่วน Header และตัวระบุส่วนของแต่ละส่วน

Mark Nelson ได้ศึกษาและพัฒนาเทคนิคการอัดข้อมูลต่างๆเป็นภาษาซีไว้ในหนังสือ The Data Compression Book ในการวิจัยครั้งนี้นำเทคนิคอัดข้อมูล LZW ซึ่ง Mark Nelson พัฒนาด้วยภาษาซีมาดัดแปลงให้เหมาะสมสำหรับการอัดแฟ้มข้อมูลจุฬาจารึก เทคนิคอัดข้อมูล LZW ซึ่ง Mark Nelson พัฒนาไว้มีข้อเด่นหลายประการ ข้อเด่นเหล่านี้ Mark Nelson ได้นำมาจากเทคนิคอัดข้อมูลของโปรแกรม COMPRESS ของยูนิกซ์ เทคนิคอัดข้อมูล LZW นี้มีขั้นตอนการเข้ารหัส/ถอดรหัสที่กล่าวไว้ในหัวข้อที่ผ่านมา เทคนิคอัดข้อมูลนี้ใช้รหัสขนาด 15 บิตจึงสามารถใช้พจนานุกรมขนาด 32 กิโลไบต์ การเข้ารหัส/ถอดรหัสจะเริ่มต้นด้วยรหัสขนาด 9 บิตก่อน จนกระทั่งพจนานุกรมของรหัสขนาด 9 บิตใช้หมดแล้วจึงเพิ่มขนาดรหัสเป็น 10 บิต ทำเช่นนี้เรื่อยไปจนใช้รหัสขนาด 15 บิต เมื่อเพิ่มขนาดรหัสขึ้นจะต้องใส่รหัส BUMP_CODE ในแฟ้มข้อมูลเพื่อให้ตัวถอดรหัสทราบว่าต้องเพิ่มขนาดรหัสที่ใช้ขยายข้อมูล แฟ้มข้อมูลที่อัดข้อมูลด้วยเทคนิคนี้จะมีขนาดเล็กกว่าแฟ้มข้อมูลที่อัดข้อมูลด้วยรหัสขนาด 15 บิตเท่านั้น เมื่อพจนานุกรมของรหัสขนาด 15 บิตใช้หมดแล้ว เทคนิคอัดข้อมูลนี้จะลบข้อความในพจนานุกรมทั้งหมดกลายเป็นพจนานุกรมว่างเพื่อใช้งานต่ออีกครั้ง ดังนั้นตัวเข้ารหัสต้องใส่รหัส FLUSH_CODE ในแฟ้มข้อมูลเพื่อให้ตัวถอดรหัสลบพจนานุกรมกลายเป็นพจนานุกรมว่างเพื่อใช้งานต่ออีกครั้ง พจนานุกรมที่ใช้เป็นต้นไม้

แบบหลายทาง แต่ละบัพใบเป็นตัวแปรโครงสร้างที่ประกอบด้วย 1. codevalue เป็นรหัสของบัพใบนี้ 2. parentcode เป็นรหัสของบัพใบแม่ 3. character เป็นสัญลักษณ์ของบัพใบนี้ จะเห็นว่าบัพใบไม่มีตัวชี้ไปยังบัพใบลูก เทคนิคนี้ใช้ฟังก์ชันแบบแฮช (hash function) หาตำแหน่งบัพใบลูก ฟังก์ชันแบบแฮชนี้เป็นฟังก์ชันที่ใช้ในโปรแกรม COMPRESS ของยูนิกซ์ ฟังก์ชันแบบแฮชใช้ในการเข้ารหัสเท่านั้นเพราะการเข้ารหัสจะหาข้อความซ้ำในพจนานุกรมซึ่งเป็นการเดินตามกิ่งของต้นไม้ลงไปยังแต่ละบัพใบลูก ส่วนการถอดรหัสไม่ต้องใช้ฟังก์ชันแฮชเพราะแต่ละบัพใบจะมีรหัสของบัพใบแม่และสัญลักษณ์แล้ว จึงสามารถถอดรหัสจากพจนานุกรมได้ทันที เทคนิคอัดข้อมูลนี้จะอ่าน/จัดเก็บรหัสขนาด 9 ถึง 15 บิต จึงไม่สามารถใช้ฟังก์ชันรับเข้า/ส่งออกของภาษาซีได้ เทคนิคอัดข้อมูลนี้จึงใช้ bitfile ควบคุมการอ่าน/จัดเก็บกลุ่มบิตของรหัส bitfile เป็นตัวแปรโครงสร้างที่ประกอบด้วย 1. rack เป็นกลุ่มบิตของรหัสที่อ่านจาก/จัดเก็บในแฟ้มข้อมูล 2. mask เป็นตัวควบคุมการอ่าน/จัดเก็บกลุ่มบิตของรหัสใน rack (Mark Nelson, 1991)

การออกแบบและพัฒนาโปรแกรมอรรถประโยชน์ด้านการอัดแฟ้มข้อมูล

โครงสร้างแฟ้มข้อมูลจุฬาจารึก 78 ได้จัดเก็บข้อมูลแต่ละส่วนแยกเป็นส่วนต่างๆ 10 ส่วนคือ ส่วน Header, ส่วน Style, ส่วน Section, ส่วน Font, ส่วน Picture, ส่วน Text, ส่วน Index, ส่วน Attribute, ส่วน Overlay Drawing และส่วน TTF Embed การอัดแฟ้มข้อมูลจุฬาจารึกจะไม่อัดข้อมูลส่วน Header และตัวระบุส่วนของแต่ละส่วน เพื่อให้จุฬาจารึกตรวจสอบขณะเปิดแฟ้มข้อมูลว่าเป็นแฟ้มข้อมูลจุฬาจารึกหรือไม่ และตรวจสอบว่าโครงสร้างแฟ้มข้อมูลถูกต้องหรือไม่

การวิจัยนี้นำเทคนิคการอัดข้อมูลที่ Mark Nelson พัฒนาเป็นภาษาซีไว้ในหนังสือ The Data Compression Book มาดัดแปลงให้เหมาะสมสำหรับการอัดแฟ้มข้อมูลจุฬาจารึก โปรแกรมอรรถประโยชน์ด้านการอัดแฟ้มข้อมูลจุฬาจารึกจะแบ่งเป็น 2 ส่วนคือส่วนการอัดแฟ้มข้อมูลจุฬาจารึกเป็นขั้นตอนการเข้ารหัสโดยแทนสัญลักษณ์ในแฟ้มข้อมูลเดิมด้วยรหัสที่สัมพันธ์กันจนได้แฟ้มข้อมูลที่อัดข้อมูลไว้ และส่วนการขยายแฟ้มข้อมูลจุฬาจารึกเป็นขั้นตอนการถอดรหัสจากแฟ้มข้อมูลที่อัดข้อมูลไว้ให้กลับคืนเป็นสัญลักษณ์ของแฟ้มข้อมูลเดิม

1. โครงสร้างข้อมูลที่ใช้ในโปรแกรม

โครงสร้างข้อมูลที่ใช้ในโปรแกรมอรรถประโยชน์ด้านการอัดข้อมูลแฟ้มข้อมูลจุฬาจารึกที่สำคัญดังนี้

1.1 ค่าคงที่ เป็นค่าที่กำหนดไว้ตั้งแต่ต้นโปรแกรมโดยไม่มีการเปลี่ยนแปลง ค่าคงที่แสดงดังตารางที่ 3.3

ชื่อค่าคงที่	ค่าที่เก็บ	คำอธิบาย
BITS	15	จำนวนบิตสูงสุดของรหัส
MAX_CODE	32767	ค่าสูงสุดของรหัสขนาด 15 บิต
TABLE_SIZE	35023	จำนวนพจนานุกรมทั้งหมด
TABLE_BANK	137	จำนวนกลุ่มพจนานุกรม ซึ่งพจนานุกรมแต่ละกลุ่มเก็บได้ 256 ข้อความ
END_OF_STREAM	256	รหัสจบอัดข้อมูล
BUMP_CODE	257	รหัสเพิ่มจำนวนบิตของรหัส แสดงว่ารหัสตามจำนวนบิตเดิมใช้หมดแล้วจึงเพิ่มจำนวนบิตอีก 1
FLUSH_CODE	258	รหัสลบพจนานุกรม เมื่อใช้พจนานุกรมเดิมหมดแล้วต้องลบพจนานุกรมให้ว่าง
FIRST_CODE	259	รหัสแรกที่จะเพิ่มในพจนานุกรม
UNUSED	-1	ค่าเริ่มต้นของรหัสที่ยังไม่ใช้ในพจนานุกรม
BUFSIZE	10000	ขนาดหน่วยความจำทั้งหมดของ buffer
HBSIZE	5000	ขนาดหน่วยความจำครึ่งหนึ่งของ buffer
NUMSECTION	9	จำนวนส่วนของแฟ้มของจุฬาจารึกที่อัดข้อมูล
READBUFFER	buffer	หน่วยความจำ buffer ที่ใช้อ่านข้อมูล
WRITEBUFFER	buffer+HBSIZE	หน่วยความจำ buffer ที่ใช้จัดเก็บข้อมูล

ตารางที่ 3.3 แสดงค่าคงที่ในโปรแกรมอรรถประโยชน์ด้านการอัดแฟ้มข้อมูล

1.2 ตัวแปร จะกล่าวเฉพาะตัวแปรที่สำคัญที่ใช้ในโปรแกรมอรรถประโยชน์ด้านอัดแฟ้มข้อมูลจุฬาจารึก ดังนี้

header เป็นตัวแปรโครงสร้างของส่วน Header (โครงสร้างข้อมูลกล่าวไว้ในบทที่ 2)

section เป็นตัวแปรโครงสร้างจัดเก็บตัวระบุส่วนและขนาดส่วนที่อัด/ขยายข้อมูลแล้ว ประกอบด้วยข้อมูลดังนี้

id เป็นตัวแปรตัวเลข ตัวระบุส่วนของแต่ละส่วน

size เป็นตัวแปรตัวเลข ขนาดส่วนของแต่ละส่วนที่อัด/ขยายข้อมูลแล้ว

fsize เป็นตัวแปรชนิดตัวเลข ขนาดแฟ้มข้อมูลที่ต้องการอัด/ขยายข้อมูล

size เป็นตัวแปรชนิดตัวเลข ขนาดข้อมูลที่อ่านเข้ามาใน buffer

secsize เป็นตัวแปรชนิดตัวเลข ขนาดส่วนที่ต้องอัด/ขยายข้อมูลต่อไป

buffer เป็นตัวแปรแถวลำดับชนิดตัวอักษรขนาด 10000 ใช้สำหรับอ่านข้อมูลจากแฟ้มข้อมูลและจัดเก็บข้อมูลที่อัด/ขยายข้อมูลแล้ว

s เป็นตัวแปรชนิดตัวเลข เป็นลำดับที่ของส่วนที่ทำงานอยู่

err เป็นตัวแปรชนิดตัวเลข ใช้ตรวจว่าการอัด/ขยายข้อมูลถูกต้องหรือไม่

ถ้าเป็น 0 แสดงว่า ทำงานถูกต้อง

ถ้าเป็น 1 แสดงว่า ทำงานไม่ถูกต้อง

tmpfile เป็นแฟ้มข้อมูลชั่วคราวจัดเก็บข้อมูลที่อัด/ขยายข้อมูลแล้ว

DICT เป็นตัวแปรโครงสร้าง เป็นพจนานุกรมจัดเก็บข้อความที่ใช้อัด/ขยายข้อมูล ประกอบด้วยข้อมูลดังนี้

codevalue เป็นตัวแปรตัวเลข รหัสของข้อความซ้ำจนถึงตัวอักษรนี้

parentcode เป็นตัวแปรตัวเลข รหัสของข้อความซ้ำก่อนถึงตัวอักษรนี้ 1 ตัว

character เป็นตัวแปรตัวเลข ตัวอักษรสุดท้ายของข้อความนี้

bitfile เป็นตัวแปรโครงสร้าง ใช้ควบคุมการอ่าน/จัดเก็บกลุ่มบิตของรหัส ประกอบด้วยข้อมูลดังนี้

rack เป็นตัวแปรชนิดตัวเลขเก็บบิตของรหัสที่อ่านจาก/จัดเก็บลงแฟ้มข้อมูล

mask เป็นตัวแปรชนิดตัวเลข ใช้กำหนดแต่ละบิต rack ตามแต่ละบิตรหัสและควบคุมการอ่าน/จัดเก็บกลุ่มบิตของรหัสใน rack

nextcode เป็นตัวแปรชนิดตัวเลข รหัสถัดไปที่จะเพิ่มในพจนานุกรมเพื่ออัด/ขยายข้อมูล

current_code_bit เป็นตัวแปรชนิดตัวเลข จำนวนบิตของรหัสที่กำลังใช้งานอยู่มีค่า 9-15 บิต

next_bump_code เป็นตัวแปรชนิดตัวเลข ค่าสูงสุดของรหัสแต่ละกลุ่มบิตของรหัส

character เป็นตัวแปรชนิดตัวเลข ตัวอักษรที่กำลังอัด/ขยายข้อมูล

stringcode เป็นตัวแปรชนิดตัวเลข รหัสของข้อความที่อ่านเข้ามาก่อนหน้านี้ ใช้ในการอัดข้อมูลเท่านั้น

index เป็นตัวแปรชนิดตัวเลข ตำแหน่งของข้อความในพจนานุกรม ใช้ในการ

อัดข้อมูลเท่านั้น

decode_stack เป็นตัวแปรแถวลำดับชนิดตัวอักษรขนาด 35023 เก็บข้อความที่ถอดรหัสทีละตัวอักษรจากพจนานุกรม ใช้ในการขยายข้อมูลเท่านั้น

newcode เป็นตัวแปรชนิดตัวเลข กลุ่มบิตของรหัสที่อ่านเข้ามาใหม่เพื่อถอดรหัส ใช้ในการขยายข้อมูลเท่านั้น

oldcode เป็นตัวแปรชนิดตัวเลข กลุ่มบิตของรหัสเดิมของข้อความที่อ่านเข้ามาก่อนหน้านี้ ใช้ในการขยายข้อมูลเท่านั้น

count เป็นตัวแปรชนิดตัวเลข จำนวนตัวอักษรใน decode_stack ซึ่งเป็นข้อความที่ถอดรหัสทีละตัวอักษรจากพจนานุกรม ใช้ในการขยายข้อมูลเท่านั้น

2. การพัฒนาโปรแกรมอรรถประโยชน์ด้านการอัดข้อมูลแฟ้มข้อมูลจุฬาจารึก

ในที่นี้จะกล่าวถึงขั้นตอนการทำงานทั้งหมดของโปรแกรมอรรถประโยชน์ด้านการอัดแฟ้มข้อมูลจุฬาจารึก ซึ่งแบ่งการทำงานเป็น 2 ส่วนคือส่วนการอัดแฟ้มข้อมูลและส่วนการขยายแฟ้มข้อมูล รายละเอียดของวิธีเข้ารหัสและวิธีถอดรหัสด้วยวิธี LZW ได้กล่าวไว้ในหัวข้อที่ผ่านมา ผังงานอธิบายการอัดแฟ้มข้อมูลแสดงดังรูปที่ 3.16 และผังงานอธิบายการขยายแฟ้มข้อมูลแสดงดังรูปที่ 3.19 แต่ละขั้นตอนมีรายละเอียดมากจำเป็นต้องใช้ผังงานหลักอธิบายการทำงานทั้งหมดก่อน จากนั้นใช้ผังงานย่อยอธิบายรายละเอียดต่อไป

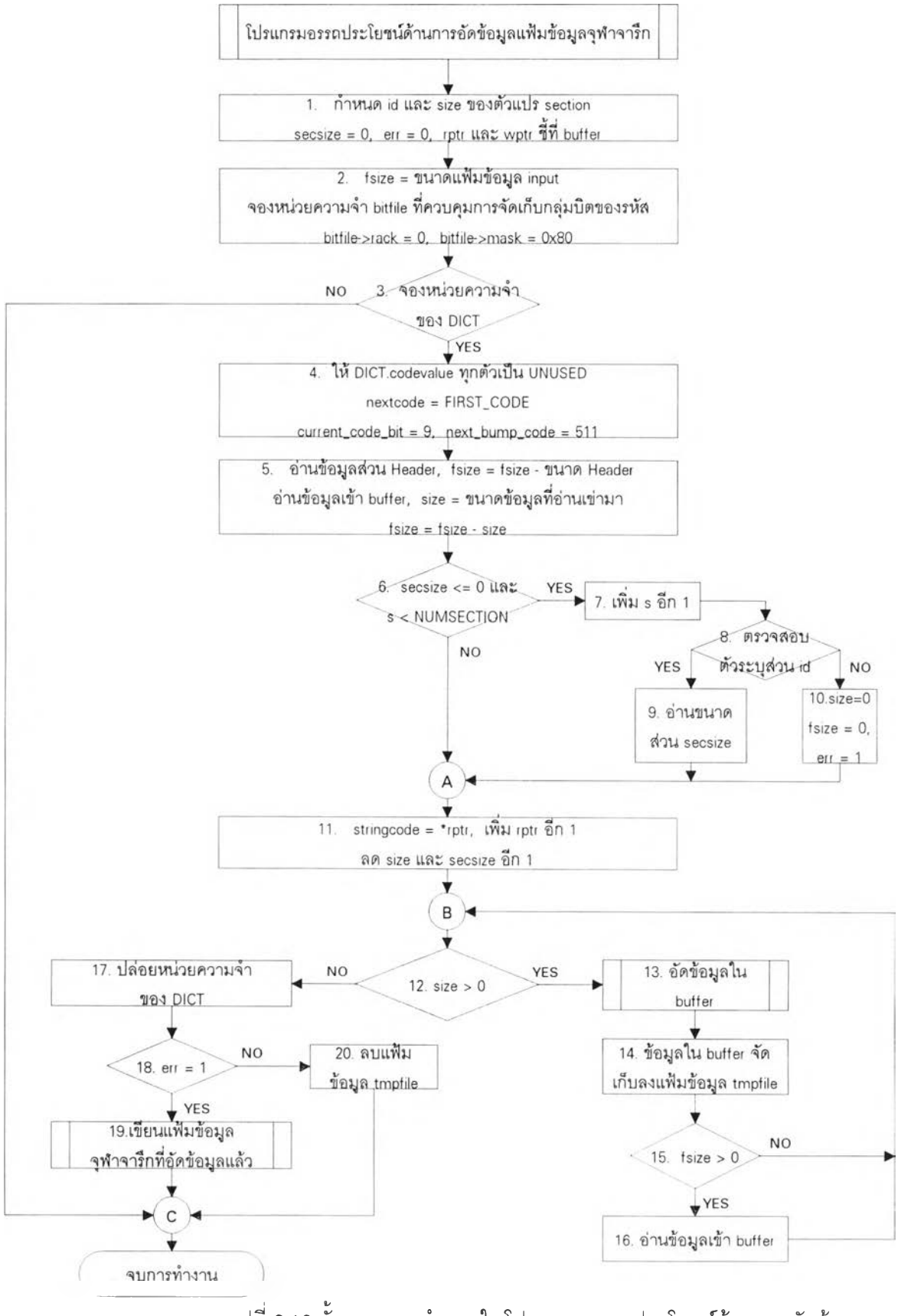
2.1 โปรแกรมอรรถประโยชน์ด้านการอัดแฟ้มข้อมูลจุฬาจารึก ผังงานของขั้นตอนการทำงานทั้งหมดแสดงดังรูปที่ 3.16 มีขั้นตอนการทำงานดังนี้

1. กำหนดค่าเริ่มต้นของตัวแปรที่ใช้ในโปรแกรม โดยกำหนดตัวระบุส่วนใน section.id และให้ขนาดส่วน section.size เป็น 0 ให้ขนาดส่วนที่อ่านเข้ามา secsize เป็น 0 ให้ตัวทดสอบการทำงาน err เป็น 0 และให้อ่าน rptr และเขียน wptr ที่ buffer

2. หาค่า fsize ซึ่งเป็นขนาดแฟ้มข้อมูลเดิมที่ต้องการอัดข้อมูล จากนั้นจองเนื้อที่หน่วยความจำให้ตัวควบคุมการอ่านกลุ่มบิตของรหัส bitfile พร้อมทั้งกำหนดค่าเริ่มต้นใน bitfile โดยให้ rack เป็น 0 และ mask เป็น 0x80

3. จองหน่วยความจำให้พจนานุกรม DICT แล้วตรวจสอบว่าจองหน่วยความจำได้หรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จึงออกจากการทำงาน

4. กำหนดค่าเริ่มต้นให้พจนานุกรมโดยให้ DICT.codevalue ทุกตัวเป็น



รูปที่ 3.16 ขั้นตอนการทำงานในโปรแกรมอรรถประโยชน์ด้านการอัดข้อมูล

UNUSED เพื่อแสดงว่าพจนานุกรมเหล่านั้นยังไม่ได้ใช้งาน จากนั้นกำหนดรหัสถัดไปที่จะเพิ่มในพจนานุกรม nextcode เป็น FIRST_CODE ให้จำนวนบิตของรหัส current_code_bit เป็น 9 และค่าสูงสุดของรหัส next_bump_code ขนาด 9 บิตเป็น 511

5. อ่านข้อมูลเข้าสู่ header พร้อมทั้งลดขนาดแฟ้มข้อมูล fsize จากนั้นอ่านข้อมูลเข้าสู่ buffer พร้อมทั้งลดขนาดแฟ้มข้อมูล fsize โดย size เป็นขนาดข้อมูลที่อ่านเข้ามาได้

6. ตรวจว่าขนาดส่วน secsize เป็น 0 และลำดับส่วน s < NUMSECTION หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าต้องตรวจสอบตัวระบุส่วนและอ่านขนาดส่วน จึงทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 11

7. เพิ่มลำดับที่ส่วน s อีก 1 เพื่อทำงานส่วนถัดไปของแฟ้มข้อมูล

8. ตรวจตัวระบุส่วน id ว่าถูกต้องหรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 10

9. เมื่อตัวระบุส่วน id ถูกต้องจะอ่านขนาดส่วน secsize เข้ามาเพื่ออัดข้อมูลเฉพาะส่วนนี้ จะข้ามไปทำงานข้อ 11

10. เมื่อตัวระบุส่วน id ไม่ถูกต้องจะกำหนดให้ size, fsize เป็น 0 และ err เป็น 1 เมื่อทำงานต่อจะออกจากการทำงาน

11. อ่านตัวอักษรจาก buffer เข้า stringcode พร้อมทั้งลด size,secsize อีก 1

12. ตรวจว่า size > 0 หรือไม่ ถ้าตำตอบคือ ใช่ แสดงว่ายังมีข้อมูลใน buffer เพื่ออัดข้อมูลต่อไป จะทำงานข้อต่อไป ถ้าตำตอบคือ ไม่ใช่ แสดงว่าไม่มีข้อมูลใน buffer จะข้ามไปทำงานข้อ 17

13. อัดข้อมูลใน buffer (รายละเอียดในผังงานย่อยแสดงดังรูปที่ 3.17)

14. ในขั้นตอนการอัดข้อมูลใน buffer ได้จัดเก็บรหัสไว้ใน buffer ด้วยเช่นกัน จึงต้องจัดเก็บรหัสดังกล่าวลงแฟ้มข้อมูลชั่วคราว tmpfile ก่อนที่จะอ่านข้อมูลใหม่เข้ามาใน buffer

15. ตรวจว่า fsize > 0 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่ายังมีข้อมูลในแฟ้มข้อมูล จึงทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่าไม่มีข้อมูลในแฟ้มข้อมูลแล้ว จะย้อนกลับไปทำงานข้อ 12 แล้วออกจากการทำงาน

16. อ่านข้อมูลเข้าสู่ buffer พร้อมทั้งลดขนาดแฟ้มข้อมูล fsize โดย size เป็นขนาดข้อมูลที่อ่านเข้ามาได้ แล้วย้อนกลับไปทำงานข้อ 12

17. ตรวจว่า err เป็น 1 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าอัดข้อมูลไม่ถูก

ต้องจะข้ามไปทำงานข้อ 20 ถ้าคำตอบคือ ไม่ใช่ แสดงว่าอัดข้อมูลถูกต้องจะไปทำงานข้อต่อไป

19. เขียนแฟ้มข้อมูลจุฬาจารึกที่อัดข้อมูลแล้ว โดยคำนวณจุดเริ่มต้นของแต่ละส่วนของแฟ้มข้อมูลจุฬาจารึกใน header กำหนดบิตอัดข้อมูลใน compressencrypt เป็น 1 แล้วเขียน header ลงแฟ้มข้อมูล จากนั้นอ่านรหัสของการอัดข้อมูลจากแฟ้มข้อมูล tmpfile แล้วเขียนลงแฟ้มข้อมูล แล้วลบแฟ้มข้อมูลชั่วคราว tmpfileจากนั้นออกจากการทำงาน

20. เมื่อการอัดข้อมูลไม่ถูกต้องจะลบแฟ้มข้อมูลชั่วคราว tmpfile ทิ้งแล้วออกจากการทำงาน

2.1.1 ส่วนอัดข้อมูลใน buffer การพัฒนาโปรแกรมส่วนนี้นำเทคนิคอัดข้อมูล LZW ซึ่ง Mark Nelson พัฒนาด้วยภาษาซีมาดัดแปลงให้เหมาะสมสำหรับการอัดแฟ้มข้อมูลจุฬาจารึก โดยไม่อัดข้อมูลส่วน Header และตัวระบุส่วนของแต่ละส่วน เพื่อใช้ตรวจสอบว่าเป็นแฟ้มข้อมูลจุฬาจารึกหรือไม่ และตรวจสอบว่าโครงสร้างแฟ้มข้อมูลถูกต้องหรือไม่ ผังงานย่อยแสดงดังรูปที่ 3.17 มีขั้นตอนการทำงานดังนี้

1. ตรวจว่า size > 0 หรือไม่ ถ้าตำตอบคือ ใช่ แสดงว่ายังมีข้อมูลใน buffer เพื่ออัดข้อมูลต่อไป จะทำงานข้อต่อไป ถ้าตำตอบคือ ไม่ใช่ แสดงว่าไม่มีข้อมูลใน buffer จะข้ามไปทำงานข้อ 21

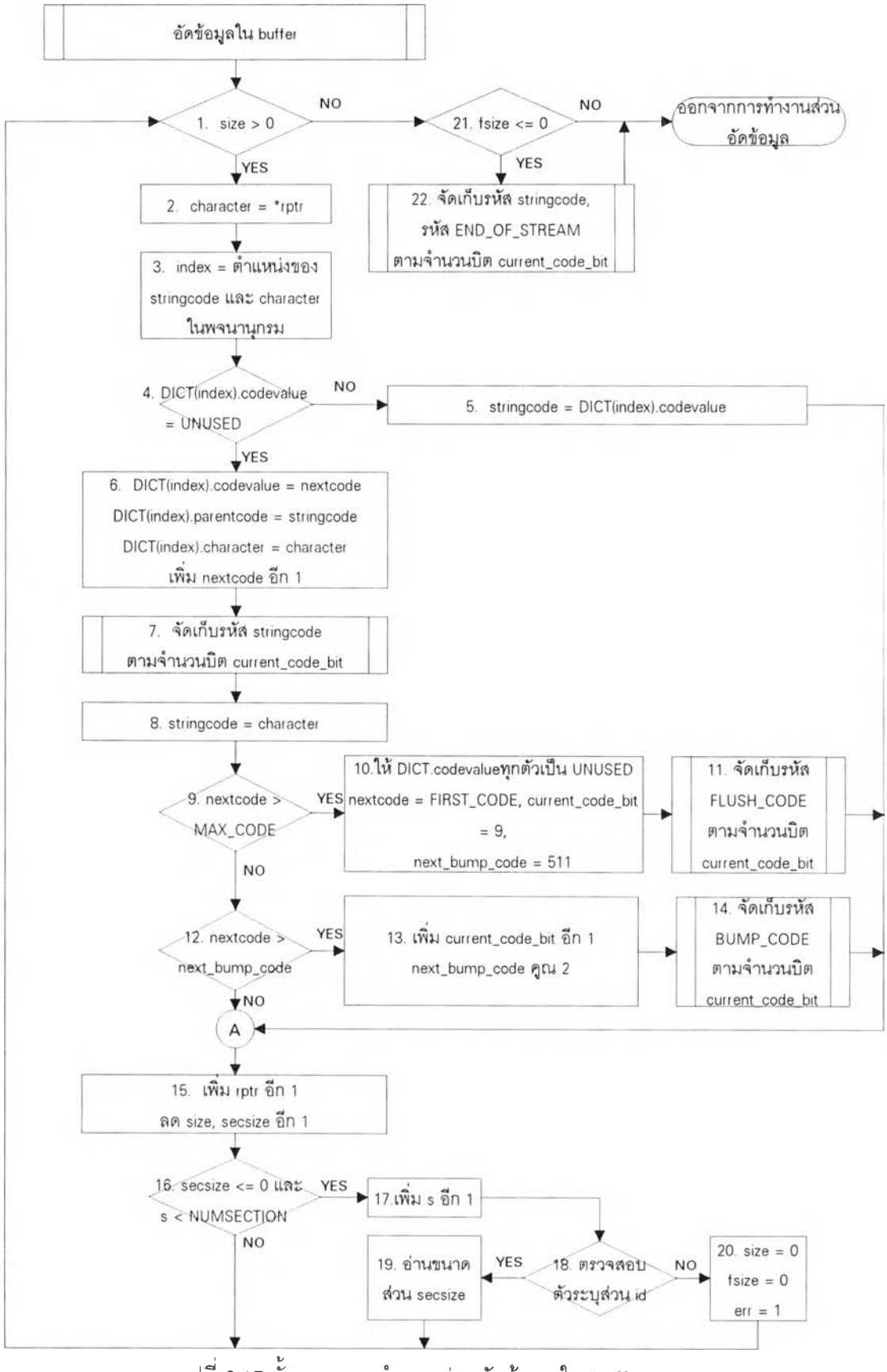
2. อ่านตัวอักษรจาก buffer เข้า character

3. นำข้อความก่อนหน้านี้ stringcode และตัวอักษร character มาค้นหาในพจนานุกรม ได้ตำแหน่งในพจนานุกรม index

4. ตรวจว่า DICT(index).codevalue ว่าเป็น UNUSED หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าข้อความ stringcode และ character ไม่ซ้ำในพจนานุกรม ดังนั้นพจนานุกรมตำแหน่งนี้จึงยังไม่ได้ใช้งาน จะข้ามไปทำงานข้อ 6 ถ้าคำตอบคือ ไม่ใช่ แสดงว่าข้อความดังกล่าวซ้ำในพจนานุกรม จะทำงานข้อต่อไป

5. เมื่อข้อความ stringcode และ character ซ้ำในพจนานุกรมตำแหน่งนี้ จะเก็บ DICT(index).codevalue ในรหัสข้อความที่อ่านเข้ามาก่อนหน้านี้ stringcode จากนั้นข้ามไปทำงานข้อ 15

6. เมื่อข้อความ stringcode และ character ไม่ซ้ำในพจนานุกรม ดังนั้นจะเพิ่มข้อความใหม่ในพจนานุกรมตำแหน่งนี้ โดยเก็บรหัสถัดไป nextcode ใน DICT(index).codevalue เก็บรหัสข้อความก่อนหน้า stringcode ใน DICT(index).parentcode และ



รูปที่ 3.17 ขั้นตอนการทำงานส่วนอัดข้อมูลใน buffer

เก็บตัวอักษร character ใน DICT(index).character พร้อมทั้งเพิ่ม nextcode อีก 1

7. จัดเก็บรหัส stringcode ตามจำนวนบิตของ current_code_bit (รายละเอียดดูเพิ่มเติมในผังงานย่อยแสดงดังรูปที่ 3.18)

8. ให้ตัวอักษร character เป็นรหัสข้อความ stringcode ต่อไป

9. ตรวจว่า nextcode มากกว่า MAX_CODE หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าใช้พจนานุกรมเดิมหมดแล้ว จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่ายังใช้พจนานุกรมเดิมไม่หมด จะข้ามไปทำงานข้อ 12

10. เมื่อใช้พจนานุกรมเดิมหมดแล้วจะลบพจนานุกรมเดิมทิ้ง โดยกำหนดค่าเริ่มต้นให้พจนานุกรม DICT.codevalue ทุกตัวเป็น UNUSED แล้วกำหนดรหัสถัดไปที่จะเพิ่มในพจนานุกรม nextcode เป็น FIRST_CODE ให้จำนวนบิตของรหัส current_code_bit เป็น 9 และค่าสูงสุดของรหัส next_bump_code ขนาด 9 บิตเป็น 511

11. จัดเก็บรหัส FLUSH_CODE ตามจำนวนบิตของ current_code_bit เพื่อให้ตัวถอดรหัสทราบว่าต้องลบพจนานุกรมทิ้ง (รายละเอียดดูเพิ่มเติมในผังงานย่อยแสดงดังรูปที่ 3.18) จากนั้นข้ามไปทำงานข้อ 15

12. ตรวจว่า nextcode มากกว่า next_bump_code หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าใช้รหัสตามจำนวนบิตเดิมหมดแล้ว จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่ายังใช้รหัสตามจำนวนบิตเดิมไม่หมด จะข้ามไปทำงานข้อ 15

13. เมื่อใช้รหัสตามจำนวนบิตเดิมหมดแล้วจะต้องเพิ่มจำนวนบิตของรหัส current_code_bit อีก 1 ดังนั้นสามารถใช้รหัสเพิ่มได้อีกจึงต้องคูณ next_bump_code ด้วย 2

14. จัดเก็บรหัส BUMP_CODE ตามจำนวนบิตของ current_code_bit เพื่อให้ตัวถอดรหัสทราบว่าต้องเพิ่มจำนวนบิตของรหัส current_code_bit อีก 1(รายละเอียดดูเพิ่มเติมในผังงานย่อยแสดงดังรูปที่ 3.18)

15. เมื่ออัดข้อมูลตัวอักษรนี้แล้วจะเพิ่ม rptr อีก 1 เพื่อใช้ตัวอักษรถัดไป และลด size, secsize อีก 1

16. ตรวจว่า secsize <= 0 และ s < NUMSECTION หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าข้อมูลส่วน s นี้หมดแล้ว จะทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่าข้อมูลส่วน s นี้ยังไม่หมด จะย้อนกลับไปทำงานข้อ 1

17. เมื่อข้อมูลส่วน s นี้หมดแล้วจะเพิ่ม s อีก 1เพื่อทำงานส่วนถัดไป

18. ตรวจตัวระบุส่วน id ว่าถูกต้องหรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 20

19. เมื่อตัวระบุส่วน id ถูกต้องจะอ่านขนาดส่วน secsize เข้ามาเพื่ออัดข้อมูลเฉพาะส่วนนี้

20. เมื่อตัวระบุส่วน id ไม่ถูกต้องจะกำหนดให้ size, fsize เป็น 0 และ err เป็น 1 แล้วย้อนกลับไปทำงานข้อ 1 เพื่อออกจากการทำงานส่วนนี้

21. เมื่อไม่มีข้อมูลใน buffer แล้วจะตรวจว่า fsize <= 0 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าไม่มีข้อมูลในแฟ้มข้อมูลเช่นกัน จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่ายังมีข้อมูลในแฟ้มข้อมูล จะออกจากการทำงานส่วนนี้เพื่ออ่านข้อมูลเข้า buffer ต่อไป

22. เมื่อไม่มีข้อมูลในแฟ้มข้อมูลแล้ว จะจัดเก็บรหัส stringcode และรหัส END_OF_STREAM ตามจำนวนบิตของ current_code_bit เพื่อให้ตัวถอดรหัสทราบว่ารหัสหมดแล้ว เพื่อเลิกการทำงาน (รายละเอียดดูเพิ่มเติมในผังงานย่อยแสดงดังรูปที่ 3.18) จากนั้นออกจากการทำงานส่วนนี้

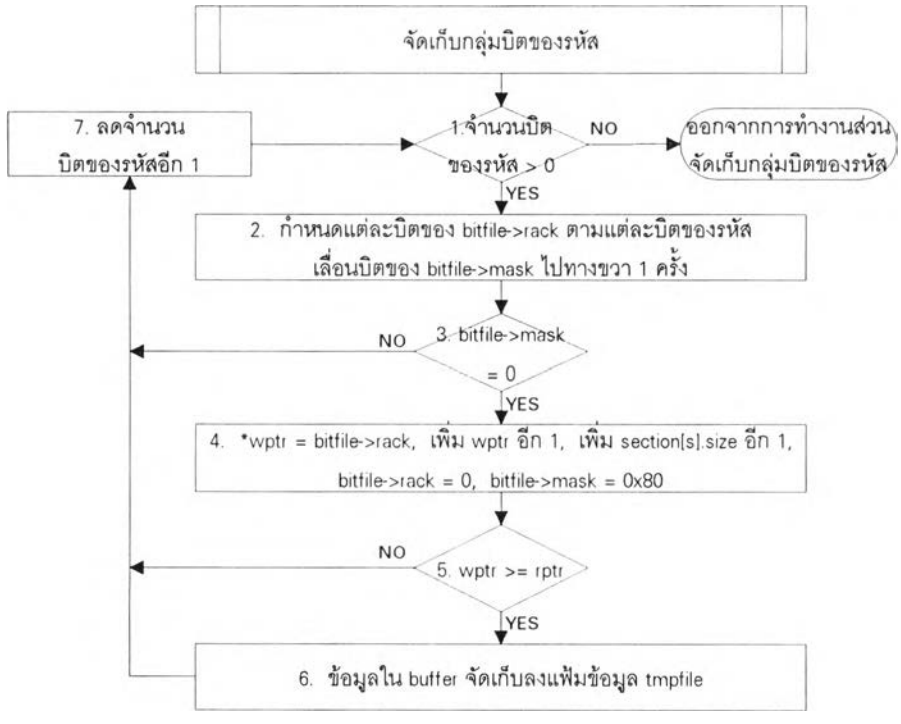
2.1.2 ส่วนจัดเก็บกลุ่มบิตของรหัส เทคนิคอัดข้อมูลนี้จะจัดเก็บรหัสขนาด 9 ถึง 15 บิต จึงไม่สามารถใช้ฟังก์ชันรับเข้า/ส่งออกของภาษาซีได้ เทคนิคอัดข้อมูลนี้จึงใช้ bitfile ควบคุมการจัดเก็บกลุ่มบิตของรหัส ผังงานย่อยแสดงดังรูปที่ 3.18 มีขั้นตอนการทำงานดังนี้

1. ตรวจว่าจำนวนบิตของรหัส > 0 หรือไม่ ถ้าตำตอบคือ ใช่ จะทำงานข้อต่อไป ถ้าตำตอบคือ ไม่ใช่ แสดงว่าจัดเก็บกลุ่มบิตของรหัสเสร็จแล้วจะออกจากการทำงานส่วนนี้

2. นำแต่ละบิตของรหัสมากำหนดแต่ละบิตของ bitfile->rack จากนั้นเลื่อนบิต bitfile->mask ไปทางขวา 1 ครั้ง

3. ตรวจว่า bitfile->mask เป็น 0 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าต้องจัดเก็บกลุ่มบิตของรหัสใน bitfile->rack แล้ว จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 7

4. จัดเก็บกลุ่มบิตของรหัสใน bitfile->rack ใน buffer และเพิ่มขนาดส่วน s อีก 1 จากนั้นกำหนดค่าเริ่มต้นให้ bitfile->rack เป็น 0 และ bitfile->mask เป็น 0x80



รูปที่ 3.18 ขั้นตอนการทำงานส่วนจัดเก็บกลุ่มบิตของรหัส

5. ตรวจว่า wptr >= rptr หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าไม่มีเนื้อที่ใน buffer สำหรับจัดเก็บรหัสแล้ว จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่ายังมีเนื้อที่เหลืออยู่ จะข้ามไปทำงานข้อ 7

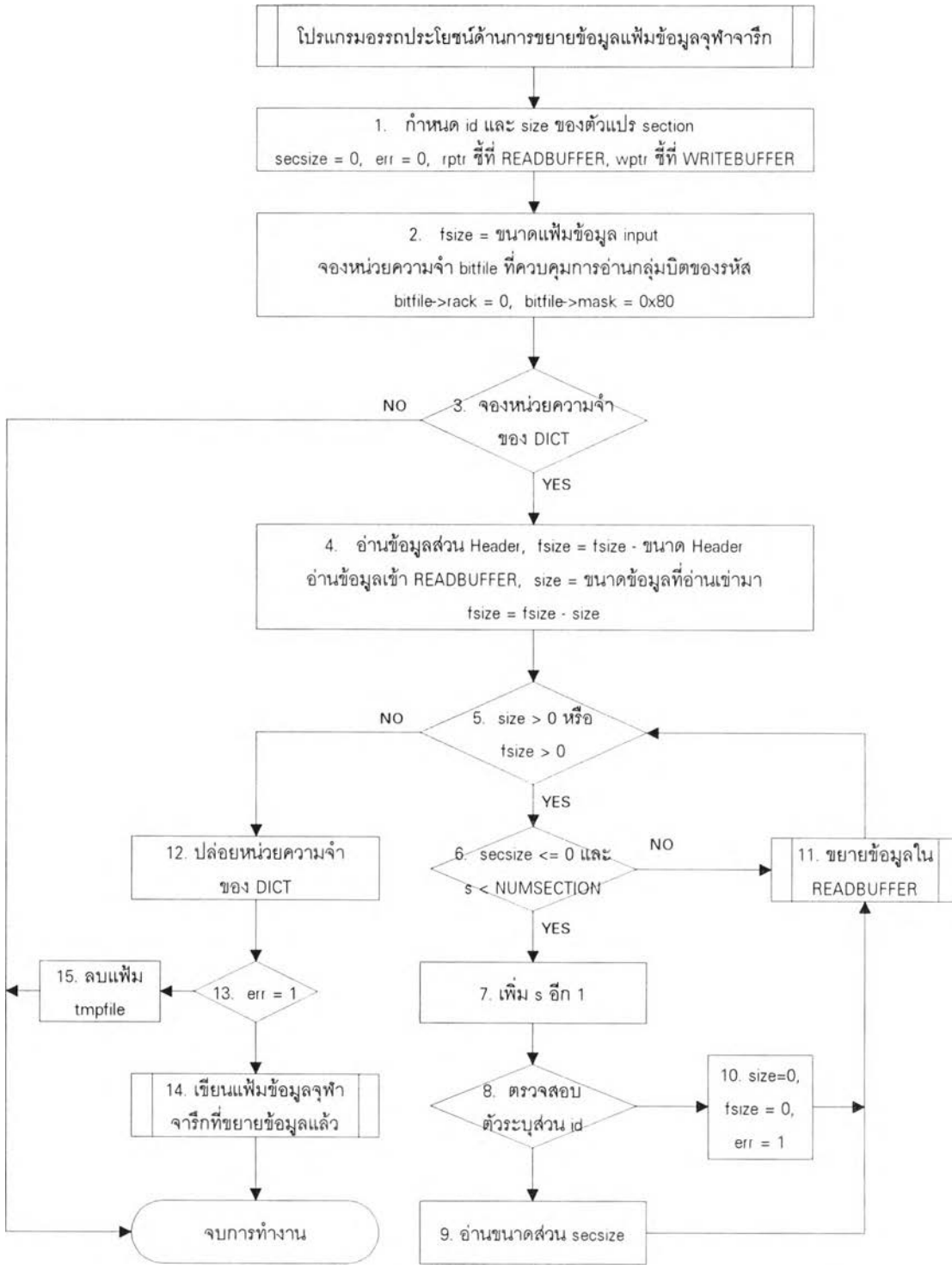
6. จัดเก็บรหัสใน buffer ลงแฟ้มข้อมูลชั่วคราว tmpfile

7. ลดจำนวนบิตของรหัสอีก 1 แล้วย้อนกลับไปทำงานข้อ 1

2.2 โปรแกรมอรรถประโยชน์ด้านการขยายข้อมูลแฟ้มข้อมูลจุฬาจารึก ผังงานของขั้นตอนการทำงานทั้งหมดแสดงดังรูปที่ 3.19 มีขั้นตอนการทำงานดังนี้

1. กำหนดค่าเริ่มต้นของตัวแปรที่ใช้ในโปรแกรม โดยกำหนดตัวระบุส่วนใน section.id และให้ขนาดส่วน section.size เป็น 0 ให้ขนาดส่วนที่อ่านเข้ามา secsize เป็น 0 ให้ตัวทดสอบการทำงาน err เป็น 0 และให้เริ่มอ่านข้อมูล rptr ที่ READBUFFER และเริ่มเขียน ข้อมูล wptr ที่ WRITEBUFFER

2. หาค่า fsize ซึ่งเป็นขนาดแฟ้มข้อมูลเดิมที่ต้องการอัดข้อมูล จากนั้นจองเนื้อที่หน่วยความจำให้ตัวควบคุมการอ่านกลุ่มบิตของรหัส bitfile พร้อมทั้งกำหนดค่าเริ่มต้นใน bitfile โดยให้ rack เป็น 0 และ mask เป็น 0x80



รูปที่ 3.19 ขั้นตอนการทำงานในโปรแกรมอรรถประโยชน์ด้านการขยายข้อมูล

3. จองหน่วยความจำให้พจนานุกรม DICT แล้วตรวจสอบว่าจองหน่วยความจำได้หรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จึงออกจากการทำงาน

4. อ่านข้อมูลเข้าสู่ header พร้อมทั้งลดขนาดแฟ้มข้อมูล fsize จากนั้นอ่านข้อมูลเข้าสู่ READBUFFER พร้อมทั้งลดขนาดแฟ้มข้อมูล fsize โดย size เป็นขนาดข้อมูลที่อ่านได้

5. ตรวจว่า size > 0 หรือ fsize > 0 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่ามีข้อมูลเหลืออยู่จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่าข้อมูลหมดแล้วจะข้ามไปทำงานข้อ12

6. ตรวจว่าขนาดส่วน secsize เป็น 0 และลำดับส่วน s < NUMSECTION หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าต้องตรวจสอบตัวระบุส่วนและอ่านขนาดส่วน จึงทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 11

7. เพิ่มลำดับที่ส่วน s อีก 1 เพื่อทำงานส่วนถัดไปของแฟ้มข้อมูล

8. ตรวจตัวระบุส่วน id ว่าถูกต้องหรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 10

9. เมื่อตัวระบุส่วน id ถูกต้องจะอ่านขนาดส่วน secsize เข้ามาเพื่ออัดข้อมูลเฉพาะส่วนนี้ จะข้ามไปทำงานข้อ 11

10. เมื่อตัวระบุส่วน id ไม่ถูกต้องจะกำหนดให้ size, fsize เป็น 0 และ err เป็น 1 เมื่อทำงานต่อจะออกจากการทำงาน

11. ขยายข้อมูลใน READBUFFER (รายละเอียดดูเพิ่มเติมในผังงานย่อยแสดงดังรูปที่ 3.20) แล้วย้อนกลับไปทำงานข้อ 5

12. เมื่อขยายข้อมูลหมดแล้วจะปล่อยหน่วยความจำของ DICT

13. ตรวจว่า err เป็น 1 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าขยายข้อมูลไม่ถูกต้องจะข้ามไปทำงานข้อ 15 ถ้าคำตอบคือ ไม่ใช่ แสดงว่าขยายข้อมูลถูกต้องจะไปทำงานข้อต่อไป

14. เขียนแฟ้มข้อมูลจุฬาจารึกที่ขยายข้อมูลแล้ว โดยคำนวณจุดเริ่มต้นของแต่ละส่วนของแฟ้มข้อมูลจุฬาจารึกใน header กำหนดบิตอัดข้อมูลใน compressencrypt เป็น 0 แล้วเขียน header ลงแฟ้มข้อมูล จากนั้นอ่านข้อความของการขยายข้อมูลจากแฟ้มข้อมูล tmpfile แล้วเขียนลงแฟ้มข้อมูล จากนั้นออกจากการทำงาน

15. เมื่อการขยายข้อมูลไม่ถูกต้องจะลบแฟ้มข้อมูลชั่วคราว tmpfile ทิ้งแล้วออกจากการทำงาน

2.2.1 ส่วนขยายข้อมูลใน READBUFFER การพัฒนาโปรแกรมส่วนนี้นำเทคนิคขยายข้อมูล LZW ซึ่ง Mark Nelson พัฒนาด้วยภาษาซีมาดัดแปลงให้เหมาะสมสำหรับแฟ้มข้อมูลจุฬาจารึก ผังงานย่อยแสดงดังรูปที่ 3.20 มีขั้นตอนการทำงานดังนี้

1. ตรวจว่า size > 0 หรือ fsize > 0 หรือไม่ ถ้าตำตอบคือ ใช่ แสดงว่ายังมีข้อมูลใน READBUFFER เพื่อขยายข้อมูลต่อไป จะทำงานข้อต่อไป ถ้าตำตอบคือ ไม่ใช่ แสดงว่าไม่มีข้อมูลใน READBUFFER จะออกจากทำงานส่วนนี้

2. กำหนดค่าเริ่มต้นให้พจนานุกรมโดยให้ DICT.codevalue ทุกตัวเป็น UNUSED เพื่อแสดงว่าพจนานุกรมเหล่านั้นยังไม่ได้ใช้งาน จากนั้นกำหนดรหัสถัดไปที่จะเพิ่มในพจนานุกรม nextcode เป็น FIRST_CODE ให้จำนวนบิตของรหัส current_code_bit เป็น 9 และค่าสูงสุดของรหัส next_bump_code ขนาด 9 บิตเป็น 511

3. อ่านรหัสตามจำนวนบิตของ current_code_bit ให้ oldcode (รายละเอียดดูเพิ่มเติมในผังงานย่อยแสดงดังรูปที่ 3.21)

4. ตรวจว่า oldcode เป็น END_OF_STREAM หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่ารหัสที่ต้องขยายข้อมูลหมดแล้ว จะทำงานข้อต่อไปเพื่อออกจากการทำงานส่วนนี้ ถ้าคำตอบคือ ไม่ใช่ แสดงว่ายังมีรหัสที่ต้องขยายข้อมูลเหลืออยู่ จะข้ามไปทำงานข้อ 6

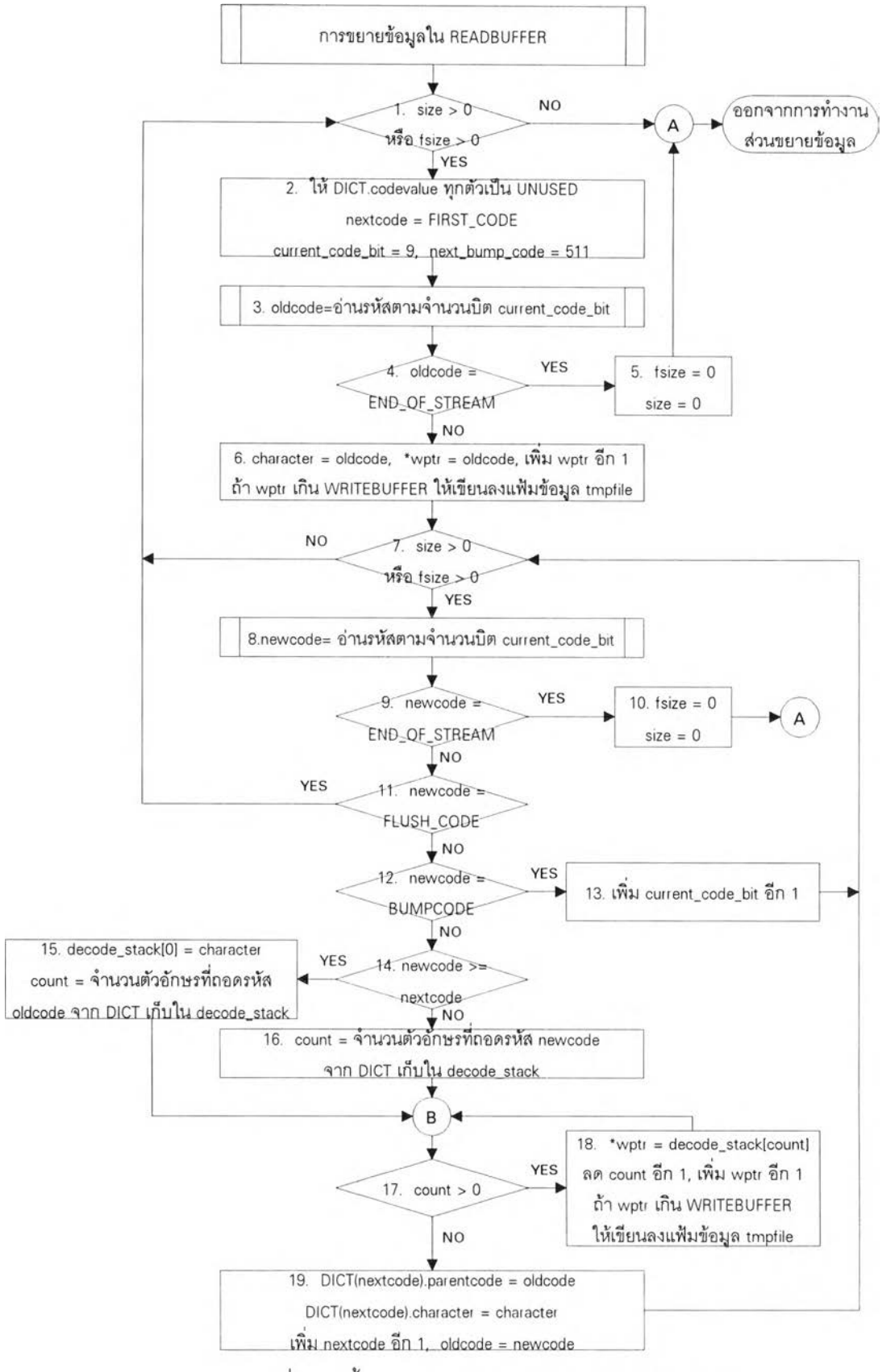
5. เมื่อรหัสที่ต้องขยายข้อมูลหมดแล้ว จะให้ fsize, size เป็น 0 แล้วออกจากการทำงานส่วนนี้

6. ให้ character มีค่าเป็น oldcode แล้วจัดเก็บ oldcode ใน WRITEBUFFER ถ้าพบว่า wptr >= WRITEBUFFER แสดงว่าไม่มีเนื้อที่ใน WRITEBUFFER สำหรับจัดเก็บข้อมูลที่ขยายข้อมูลแล้ว จะจัดเก็บข้อมูลดังกล่าวใน WRITEBUFFER ลงแฟ้มข้อมูลชั่วคราว tmpfile

7. ตรวจว่า size > 0 หรือ fsize > 0 หรือไม่ ถ้าตำตอบคือ ใช่ แสดงว่ายังมีข้อมูลใน READBUFFER เพื่อขยายข้อมูลต่อไป จะทำงานข้อต่อไป ถ้าตำตอบคือ ไม่ใช่ แสดงว่าไม่มีข้อมูลใน READBUFFER จะย้อนกลับไปทำงานข้อ 1

8. อ่านรหัสตามจำนวนบิตของ current_code_bit ให้ newcode (รายละเอียดดูเพิ่มเติมในผังงานย่อยแสดงดังรูปที่ 3.21)

9. ตรวจว่า newcode เป็น END_OF_STREAM หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่ารหัสที่ต้องขยายข้อมูลหมดแล้ว จะทำงานข้อต่อไปเพื่อออกจากการทำงานส่วนนี้ ถ้าคำตอบคือ ไม่ใช่ แสดงว่ายังมีรหัสที่ต้องขยายข้อมูลเหลืออยู่ จะข้ามไปทำงานข้อ 11



รูปที่ 3.20 ขั้นตอนการทำงานส่วนขยายข้อมูลใน buffer

10. เมื่อรหัสที่ต้องขยายข้อมูลหมดแล้ว จะให้ fsize, size เป็น 0 แล้วออกจากการทำงานส่วนนี้

11. ตรวจว่า newcode เป็น FLUSH_CODE หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าต้องลบพจนานุกรมเดิมทิ้ง แล้วกำหนดค่าเริ่มต้นให้พจนานุกรมอีกครั้ง จึงย้อนกลับไปทำงานข้อ 1 ถ้าคำตอบคือ ไม่ใช่ จะทำงานข้อต่อไป

12. ตรวจว่า newcode เป็น BUMP_CODE หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าใช้รหัสตามจำนวนบิตเดิมหมดแล้ว จึงต้องเพิ่มจำนวนบิตของรหัส จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 14

13. เมื่อใช้รหัสตามจำนวนบิตเดิมหมดแล้ว จึงต้องเพิ่มจำนวนบิตของรหัส current_code_bit อีก 1 จากนั้นย้อนกลับไปทำงานข้อ 7

14. ตรวจว่า newcode >= nextcode 0 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่ายังไม่เพิ่มรหัส newcode ในพจนานุกรมจะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่ายังมีรหัสนี้ในพจนานุกรมจะข้ามไปทำงานข้อ 16

15. เนื่องจากยังไม่เพิ่มรหัส newcode ในพจนานุกรมแสดงว่าถอดรหัสเป็นข้อความซ้ำกับข้อความเดิม oldcode จึงใส่ตัวอักษร character ใน decode_stack[0] และถอดรหัส oldcode จากพจนานุกรมเก็บใน decode_stack และ count เป็นจำนวนตัวอักษรที่ถอดรหัสได้ แล้วข้ามไปทำงานข้อ 17

16. เนื่องจากเพิ่มรหัส newcode ในพจนานุกรมแล้ว จึงถอดรหัส newcode จากพจนานุกรมเก็บใน decode_stack และ count เป็นจำนวนตัวอักษรที่ถอดรหัสได้

17. ตรวจว่า count > 0 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่ายังมีตัวอักษรที่ถอดรหัสแล้วใน decode_stack จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่าไม่มีตัวอักษรที่ถอดรหัสแล้วใน decode_stack จะข้ามไปทำงานข้อ 19

18. นำตัวอักษรใน decode_stack[count] จัดเก็บใน WRITEBUFFER ถ้าพบว่า wptr >= WRITEBUFFER แสดงว่าไม่มีเนื้อที่ใน WRITEBUFFER สำหรับจัดเก็บข้อมูลที่ขยายข้อมูลแล้ว จะจัดเก็บข้อมูลดังกล่าวใน WRITEBUFFER ลงแฟ้มข้อมูลชั่วคราว tmpfile นอกจากนี้ ลด count อีก 1 แล้วย้อนกลับไปทำงานข้อ 17

19. เมื่อไม่มีตัวอักษรที่ถอดรหัสแล้วใน decode_stack แล้วจะเพิ่มข้อความในพจนานุกรมที่ตำแหน่ง nextcode โดยเก็บ oldcode ใน DICT(nextcode).parentcode และ

เก็บตัวอักษร character ใน DICT(nextcode).character พร้อมทั้งเพิ่ม nextcode อีก 1 และให้ oldcode มีค่าเป็น newcode จากนั้นย้อนกลับไปทำงานข้อ 7

2.2.2 ส่วนอ่านกลุ่มบิตของรหัส เทคนิคอัดข้อมูลนี้จะอ่านรหัสขนาด 9 ถึง 15 บิต จึงไม่สามารถใช้ฟังก์ชันรับเข้า/ส่งออกของภาษาซีได้ เทคนิคอัดข้อมูลนี้จึงใช้ bitfile ควบคุมการอ่านกลุ่มบิตของรหัส ผังงานย่อยแสดงดังรูปที่ 3.21 มีขั้นตอนการทำงานดังนี้

1. ให้ค่าส่งคืนกลุ่มบิตของรหัส return_value เป็น 0

2. ตรวจว่าจำนวนบิตของรหัส > 0 หรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ แสดงว่าอ่านกลุ่มบิตของรหัสเสร็จแล้ว จะข้ามไปทำงานข้อ 17

3. ตรวจว่า bitfile->mask เป็น 0x80 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าต้องอ่านกลุ่มบิตของรหัสเข้าสู่ bitfile->rack แล้ว จะข้ามไปทำงานข้อ 5 ถ้าคำตอบคือ ไม่ใช่ จะทำงานข้อต่อไป

4. นำแต่ละบิตของ bitfile->rack มากำหนดแต่ละบิตของรหัสใน return_value

5. อ่านกลุ่มบิตของรหัสเข้าสู่ bitfile->rack พร้อมทั้งลด size, secsize อีก 1

6. ตรวจว่า size <= 0 หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าไม่มีข้อมูลใน READBUFFER จะทำงานข้อต่อไป ถ้าคำตอบคือ ใช่ แสดงว่ายังมีข้อมูลใน READBUFFER จะข้ามไปทำงานข้อ 8

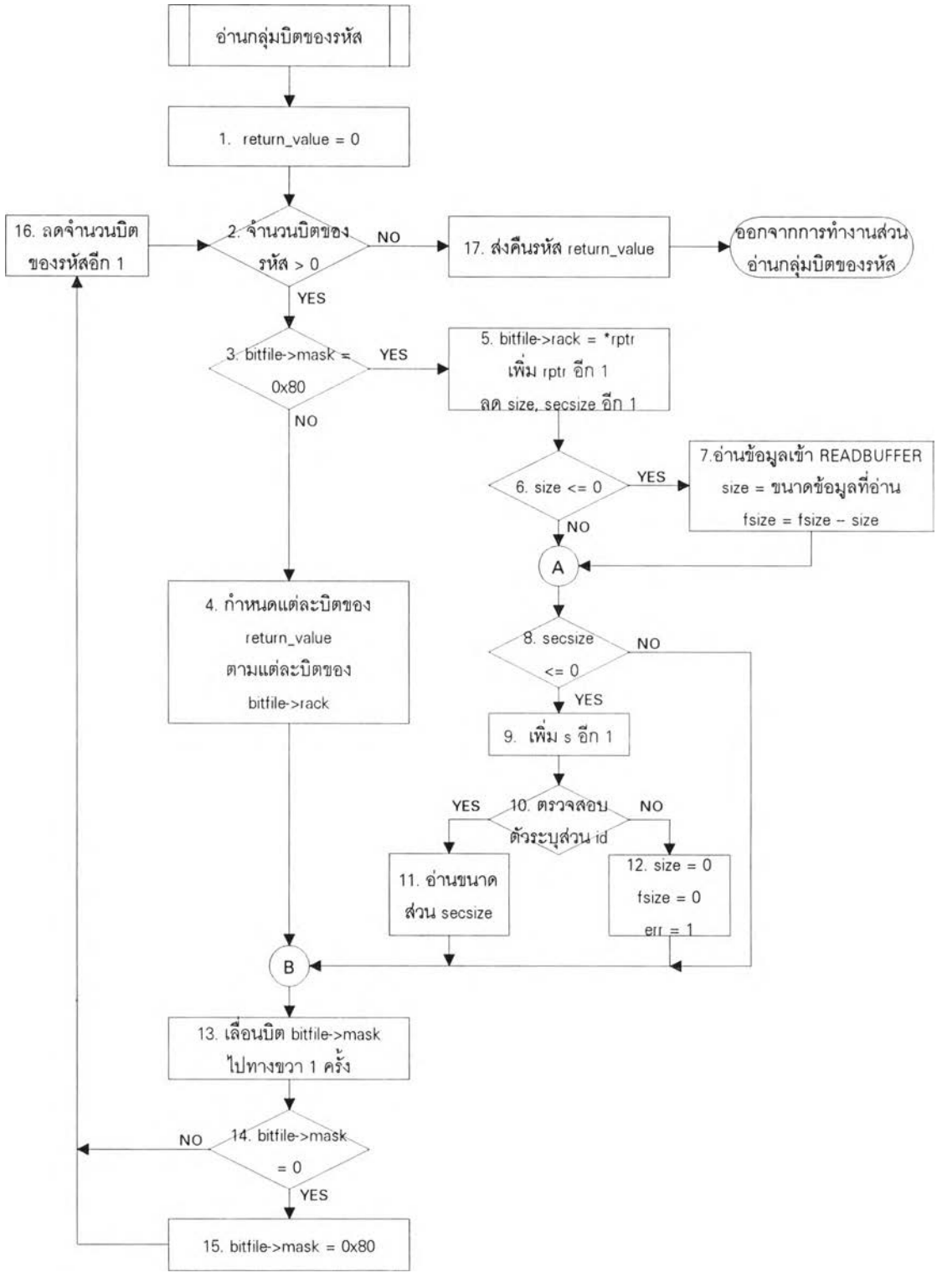
7. อ่านข้อมูลเข้าสู่ READBUFFER พร้อมทั้งลดขนาดแฟ้มข้อมูล fsize โดย size เป็นขนาดข้อมูลที่อ่านเข้ามาได้

8. ตรวจว่าขนาดส่วน secsize <= หรือไม่ ถ้าคำตอบคือ ใช่ แสดงว่าต้องตรวจสอบตัวระบุส่วนและอ่านขนาดส่วน จึงทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 13

9. เพิ่มลำดับที่ส่วน s อีก 1 เพื่อทำงานส่วนถัดไปของแฟ้มข้อมูล

10. ตรวจตัวระบุส่วน id ว่าถูกต้องหรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 12

11. เมื่อตัวระบุส่วน id ถูกต้องจะอ่านขนาดส่วน secsize เข้ามาเพื่ออัดข้อมูลเฉพาะส่วนนี้ จะข้ามไปทำงานข้อ 13



รูปที่ 3.21 ขั้นตอนการทำงานส่วนอ่านกลุ่มบิตของรหัส

12. เมื่อตัวระบุส่วน id ไม่ถูกต้องจะกำหนดให้ size, fsize เป็น 0 และ err เป็น 1 เพื่อบอกว่าการขยายข้อมูลไม่ถูกต้อง

13 เลื่อนบิต bitfile->mask ไปทางขวา 1 ครั้ง

14. ตรวจว่า bitfile->mask เป็น 0 หรือไม่ ถ้าคำตอบคือ ใช่ จะทำงานข้อต่อไป ถ้าคำตอบคือ ไม่ใช่ จะข้ามไปทำงานข้อ 16

15. กำหนดค่าให้ bitfile->mask เป็น 0x80

16. ลดจำนวนบิตของรหัสอีก 1 แล้วย้อนกลับไปทำงานข้อ 2

17. เมื่อจำนวนบิตหมดแล้วจะส่งคืนกลุ่มบิตของรหัสใน return_value แล้วออกจากการทำงานส่วนนี้