แขนหุ่นยนต์ที่ยืดหยุ่นได้
EXTENDIBLE FLEXIBLE ARM

มานิตย์ ขวัญยืน
MANIT KWANYUN

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ. 2555
KMITL-2012-SC-M-002-008

แฮชชิงทรัยแบบขยายได้

EXTENDIBLE TRIE HASHING

มานิตย์ ขวัญยืน

MANIT KWANYUN

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต

สาขาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2555

KMTL-2012-SC-M-002-008

EXTENDIBLE TRIE HASHING

MANIT KWANYUN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2012

KMTL-2012-SC-M-002-008

คณะวิทยาศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ใบรับรองวิทยานิพนธ์

หัวข้อวิทยานิพนธ์	แฮชชิงทรัยแบบขยายได้
	Extendible Trie Hashing
นักศึกษา	นายมานิตย์ ขวัญยืน
รหัสประจำตัว	52650805
ปริญญา	วิทยาศาสตรมหาบัณฑิต
สาขาวิชา	วิทยาการคอมพิวเตอร์
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผศ.ดร.ศรัณย์ อินทโกสุม

คณะกรรมการสอบวิทยานิพนธ์		ลายมือชื่อ
รศ.ดร.จีรพร	วีระพันธุ์	
รศ.ดร.วีระ	บุญจริง	
ดร.เฉลิมศักดิ์	เลิศวงศ์เสถียร	
ผศ.ดร.ศรัณย์	อินทโกสุม	

วัน / เดือน/ ปี ที่สอบ 20 เมษายน พ.ศ. 2555 เวลา 13.00 – 16.00 น.
สถานที่สอบ ณ ห้อง 216 ชั้น 2 อาคารจุฬาภรณวลัยลักษณ์ 1

คณะวิทยาศาสตร์รับรองแล้ว

(รองศาสตราจารย์ ดร.ดุษณี ธนะบริพัฒน์)
คณบดีคณะวิทยาศาสตร์
วันที่ 26 **เดือน** เม.ย. **พ.ศ.** 55

หัวข้อวิทยานิพนธ์	แฮชชิงทรัยแบบขยายได้
นักศึกษา	นายมานิตย์ ขวัญยืน
รหัสประจำตัว	52650805
ปริญญา	วิทยาศาสตรมหาบัณฑิต
สาขาวิชา	วิทยาการคอมพิวเตอร์
พ.ศ.	2555
อาจารย์ที่ปรึกษา	รศ.ดร.วีระ บุญจริง

บทคัดย่อ

งานวิจัยนี้เสนอวิธีการใหม่ในการเข้าถึงระเบียนข้อมูลของแฟ้มยืดหยุ่นที่เรียกว่า แฮชชิงทรัยแบบขยายได้ โดยวิธีการใหม่นี้ได้รวมข้อดีของวิธีแฮชชิงแบบขยายได้เข้ากับวิธีแฮชชิงทรัย เพื่อแก้ปัญหาการไม่สมดุลของทรัยทำให้สามารถเข้าถึงบัคเก็ตได้โดยใช้เวลา $O(m)$ โดยที่ m คือขนาดของคีย์

คำสำคัญ: วิธีเข้าถึง, โครงสร้างข้อมูล, แฮชชิง, แฮชชิงทรัย, แฮชชิงแบบขยายได้

Thesis Title	Extendible Trie Hashing
Student	Manit Kwanyun
Student ID	52650805
Degree	Master of Science
Program	Computer Science
Year	2012
Thesis Advisor	Assoc.Prof.Dr.Veera Boonjing

ABSTRACT

The research proposes a new access method for dynamic files called extendible trie hashing. The method combines an extendible hashing method with a trie hashing as a solution to degeneracy problems. It is able to access a file bucket in $O(m)$ times, where m is a key size.

Keywords: Access method, Data structure, Hashing, Trie hasing, Extendible hashing

กิตติกรรมประกาศ

เริ่มต้นข้าพเจ้าขอขอบคุณรศ.ดร.วีระ บุญจริง อาจารย์ที่ปรึกษาวิทยานิพนธ์เป็นอย่างสูง ที่คอยให้คำปรึกษาและให้คำแนะนำต่างๆ ไม่ว่าจะเป็นการศึกษางานวิจัย การคิดงานวิจัย การดำเนินงานวิจัย การเขียนงานวิทยานิพนธ์ และอื่นๆ จนทำให้วิทยานิพนธ์ของข้าพเจ้าสามารถสำเร็จลุล่วงไปได้ด้วยดีและมีคุณภาพ

ขอขอบคุณเพื่อนๆและพี่ๆทุกคนในสาขาวิทยาการคอมพิวเตอร์สำหรับกำลังใจและความช่วยเหลือต่างๆโดยเฉพาะการดำเนินเรื่องในขั้นตอนต่างๆ ทำให้ข้าพเจ้านั้นสามารถดำเนินเรื่องต่างๆได้จนผ่านครบทุกขั้นตอน

ขอขอบคุณท่านคณาจารย์สาขาวิทยาการคอมพิวเตอร์ทุกท่านสำหรับวิชาความรู้ต่างๆที่มอบให้แก่ข้าพเจ้า ทำให้ข้าพเจ้าได้มีความรู้ความเข้าใจต่างๆในวิชานั้นๆ

ขอขอบคุณคณะกรรมการสอบวิทยานิพนธ์ทุกท่านสำหรับการรับฟังและให้คำเสนอแนะต่างๆเพื่อให้งานวิทยานิพนธ์ของข้าพเจ้ามีคุณภาพมากขึ้น

ขอขอบคุณพี่ฝ่ายบัณฑิตศึกษา คณะวิทยาศาสตร์ ที่ช่วยดำเนินเรื่องต่างๆของข้าพเจ้าจนทำให้วิทยาพนธ์ของข้าพเจ้าสามารถสำเร็จเป็นรูปเล่มโดยสมบูรณ์

และสุดท้ายนี้ ข้าพเจ้าขอขอบคุณบิดา มารดา และครอบครัวอย่างสูง ที่คอยเป็นแรงใจและเป็นแรงสนุบสนับสนุนตลอดมา จนทำให้ข้าพเจ้าสามารถทำวิทยานิพนธ์ได้สำเร็จโดยสมบูรณ์

มานิตย์ ขวัญยืน

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญรูป

สารบัญรูป (ต่อ)

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของปัญหา

แฮชชิงทรัย (Trie hashing (TH)) [3] เป็นวิธีการเข้าถึงวิธีหนึ่งสำหรับการเก็บและเข้าถึงระเบียนของแฟ้มข้อมูลยืดหยุ่น โดยระเบียนถูกเก็บอยู่ในบัคเก็ตซึ่งบรรจุไว้อย่างมาก b ระเบียนบนดิสค์ การค้นคืนระเบียนที่ระบุโดยคีย์ๆหนึ่งนั้นจะประกอบไปด้วย 2 ขั้นตอนได้แก่ 1) คำนวณหาที่อยู่บัคเก็ตโดยใช้ trie 2) เข้าถึงดิสค์ดึงบัคเก็ตที่มีที่อยู่ดังกล่าวลงในหน่วยความจำหลัก จากนั้นจึงทำการสืบค้นหาระเบียนที่ระบุโดยคีย์ดังกล่าว ข้อดีของวิธีนี้คือ ทราบผลของการค้นคืนได้อย่างรวดเร็ว โดยทำการเข้าถึงดิสค์เพียง 1 ครั้ง ถ้า trie นั้นอยู่ในหน่วยความจำหลักได้เพียงพอ แต่ถ้าแฟ้มข้อมูลมีขนาดใหญ่มาก trie จะไม่สามารถบรรจุไหว จึงต้องใช้วิธีแบ่ง trie ออกเป็น subtries และแต่ละ subtrie จะถูกเก็บไว้ในเพจ (pages) บนดิสค์ วิธีนี้เรียกว่า แฮชชิงทรัยหลายระดับ (Multilevel trie hashing (MLTH)) [4] ส่งผลให้ต้องเข้าถึงดิสค์มากขึ้น แต่เพียง 2 ครั้งก็เพียงพอสำหรับ แฟ้มข้อมูลขนาดเป็นจิ๊กกะไบต์แล้ว และนอกจากนี้ TH ยังสามารถทำการประมวลผลแบบ range query ได้ซึ่งได้ทำการอธิบายขั้นตอนไว้ใน [4] เนื่องจากมีคุณสมบัติ weak order preserving กล่าวคือ สำหรับสองบัคเก็ตใดๆ B_i, B_j ซึ่ง $K_i \in B_i$ และ $K_j \in B_j$ จะได้ว่า $K_i < K_j$ สำหรับ $\forall K_i \in B_i$ และ $\forall K_j \in B_j$ สำหรับลำดับคีย์แบบสุ่มนั้นให้ค่าโหลดแฟกเตอร์ 70% และแบบเรียงลำดับนั้นให้ 50%

จากที่กล่าวมาข้างต้นนั้นถือได้ว่าเป็นวิธีเข้าถึงที่มีประสิทธิภาพมากวิธีหนึ่ง แต่ถึงกระนั้นวิธีนี้ก็ยังมีข้อเสียหลักๆอยู่อันได้แก่ 1) เมื่อลำดับคีย์ที่เพิ่มนั้นเรียงลำดับ ทำให้การขยายโหนดนั้นเอนไปทางด้านใดด้านหนึ่งหรือเรียกว่า degeneracy ส่งผลให้ใช้เวลาในการสืบค้นโหนดใบที่อยู่ระดับล่างสุดนาน และเวลาในการสืบค้นขึ้นอยู่กับลำดับคีย์ด้วย กล่าวคือคีย์ที่อยู่ในลำดับต้นๆจะใช้เวลาสืบค้นเร็ว ส่วนคีย์ที่อยู่ในลำดับปลายๆจะใช้เวลาในการสืบค้นนาน 2) หลังจากเกิดระบบขัดข้องจะทำให้ trie ที่ถูกสร้างไว้สูญหาย จึงต้องมีการสร้างใหม่ซึ่งใช้ต้นทุนการสร้างสูง

จากข้อเสียดังกล่าวเหล่านี้ จึงได้มีผู้เสนอวิธีแก้ไขต่างๆไว้มากมาย ไม่ว่าจะเป็นปัญหา degeneracy [6], การปรับปรุงค่าโหลดแฟกเตอร์ [5] หรือการคงอยู่ (persistence) [7] และนอกจากนี้มีผู้พัฒนาดัดแปลงให้เป็นโครงสร้างข้อมูลแบบกระจาย [8] และ [1] อีกด้วย สำหรับในที่นี่ได้ทำการเสนอวิธีใหม่โดยเป็นโครงสร้างข้อมูลแบบปรกติเพื่อแก้ปัญหา degeneracy และเพิ่มความเร็วในการสืบค้นหาที่อยู่บัคเก็ต ซึ่งมีชื่อเรียกว่า แฮชชิงทรัยแบบขยายได้ (Extendible trie hashing (EXTH)) เป็นวิธีที่เกิดจากการรวมกันระหว่างแฮชชิงแบบขยายได้ [2] กับแฮชชิงทรัย โดยนำเอา

โครงสร้างและหลักการเข้าถึงของแฮชชิงแบบขยายได้กับหลักการแบ่งการบัคเก็ตของแฮชชิงทรัย มารวมกัน เพื่อเพิ่มความเร็วในการสืบค้นหาที่อยู่บัคเก็ต ทำให้ทราบผลของการค้นคืนได้รวดเร็วยิ่งขึ้น โดยใช้เวลาในการสืบค้น $O(m)$ โดยที่ m แทน ความยาวของคีย์

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

พัฒนาโครงสร้างข้อมูลสำหรับคำนวณหาที่อยู่บัคเก็ตของวิธีแฮชชิงทรัยเพื่อแก้ปัญหา degeneracy และเพิ่มความเร็วในการสืบค้นด้วย

1.3 สมมติฐานของการศึกษา

การนำโครงสร้างข้อมูลและหลักการเข้าถึงที่อยู่บัคเก็ตของวิธีแฮชชิงแบบขยายได้มารวมกับโครงสร้างข้อมูลของวิธีแฮชชิงทรัย สามารถแก้ปัญหา degeneracy และเพิ่มความเร็วในการสืบค้นหาที่อยู่บัคเก็ตได้

1.4 ทฤษฎีหรือแนวความคิดที่ใช้ในการวิจัย

เนื่องจากการสร้างโครงสร้างข้อมูลสำหรับสืบค้นหาที่อยู่บัคเก็ตของวิธีแฮชชิงทรัยโดยมีลำดับของคีย์เป็นแบบเรียงลำดับนั้น ทำให้เกิดปัญหา degeneracy ซึ่งส่งผลให้ใช้เวลาในการสืบค้นโหนดใบที่อยู่ระดับล่างสุดนาน และเวลาในการสืบค้นขึ้นอยู่กับลำดับคีย์ด้วย ด้วยเหตุนี้จึงมีแนวความคิดที่จะพัฒนาโครงสร้างข้อมูลเพื่อแก้ปัญหาดังกล่าวและเพิ่มความเร็วในการสืบค้นด้วย ดังนั้นจึงได้นำเอาโครงสร้างข้อมูลที่เก็บในลักษณะไดเรกทอรีและหลักการเข้าถึงที่อยู่บัคเก็ตโดยใช้คำขึ้นต้นของคีย์อย่างแฮชชิงแบบขยายได้มารวมกับโครงสร้างข้อมูลของวิธีแฮชชิงทรัย

1.5 ขอบเขตของการวิจัย

งานวิจัยชิ้นนี้เป็นการพัฒนาโครงสร้างข้อมูลสำหรับการสืบค้นหาที่อยู่บัคเก็ตของวิธีแฮชชิงทรัยและโครงสร้างดังกล่าวสามารถอยู่ในหน่วยความจำได้เพียงพอ โดยนำเอาโครงสร้างข้อมูลและหลักการเข้าถึงที่อยู่บัคเก็ตของวิธีแฮชชิงแบบขยายได้มารวมกับโครงสร้างข้อมูลของวิธีแฮชชิงทรัย ทำการเปรียบเทียบความเร็วในการสืบค้นหาที่อยู่บัคเก็ตโดยวัดจากการคำนวณการทำงานจาก pseudo code

1.6 ส่วนประกอบของวิทยานิพนธ์

ส่วนที่เหลือของวิทยานิพนธ์เล่มนี้ได้ถูกแบ่งเนื้อหาออกเป็นบทต่างๆดังนี้

บทที่ 2 แสดงรายละเอียดเกี่ยวกับทฤษฎีพื้นฐานของวิธีแฮชชิงทรัย รวมทั้งงานวิจัยที่เกี่ยวข้องด้วย

บทที่ 3 แสดงรายละเอียดต่างๆเกี่ยวกับวิธีแฮชชิงทรัยแบบขยายได้ ซึ่งประกอบไปด้วยโครงสร้างข้อมูล, การค้นหา, การแบ่งบัคเก็ต, การเพิ่ม, การลบ, การปรับปรุง และการสร้าง

บทที่ 4 แสดงการวิเคราะห์การแก้ปัญหา degeneracy ของวิธีแฮชชิงทรัยแบบขยายได้และขั้นตอนวิธีของการสืบค้นหาที่อยู่บัคเก็ตระหว่างวิธีแฮชชิงทรัยกับวิธีแฮชชิงทรัยแบบขยายได้

บทที่ 5 แสดงการสรุปของเนื้อหาและข้อเสนอแนะ

บทที่ 2
ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

2.1 แฮชชิงทรัย (Trie Hashing)

แฮชชิงทรัย (Trie hashing (TH)) [3] เป็นวิธีการเข้าถึงที่มีประสิทธิภาพวิธีหนึ่งสำหรับการเก็บและเข้าถึงระเบียนของแฟ้มข้อมูลยืดหยุ่น โดยใช้ trie ในการคำนวณหาที่อยู่บัคเก็ต ข้อดีของวิธีนี้คือ ทราบผลของการค้นคืนได้อย่างรวดเร็วเนื่องจากทำการเข้าถึงดิสค์เพียง 1 ครั้งเท่านั้น และสามารถทำการประมวลผลแบบ range query ได้อีกด้วย

2.1.1 โครงสร้าง

อยู่ในรูปของไบนารี่ทรี โดยโหนดภายในเก็บอยู่ในรูปคู่ของค่าดิจิต (digit value) และหมายเลขดิจิต (digit number) ซึ่งแทนด้วย (d, i) ส่วนโหนดภายนอกหรือโหนดใบอาจเก็บเป็นได้ทั้งที่อยู่บัคเก็ตหรือ *nil* ซึ่งสื่อว่าไม่มีที่อยู่บัคเก็ตที่สอดคล้องกับโหนดนี้ ตัวอย่างโครงสร้างนั้นได้แสดงไว้ ณ รูปที่ 2.1(c) ซึ่งรวมทางเดินตรรกะไปยังโหนดต่างๆด้วย (ค่าที่อยู่บนเส้นเชื่อมระหว่างโหนด n ใดๆกับแม่ของ n) โดยแต่ละค่าถูกกำหนดไว้ดังนี้

สายอักขระซึ่งสอดคล้องกับโหนด n ใดๆใน trie นั้นเรียกว่า “ทางเดินตรรกะไปยัง n (logical path to n)” แทนด้วย C_n ถูกนิยามไว้ดังนี้

- ถ้า C_n เป็นรากแล้ว C_n = ':'
- อื่นๆ, ให้ $p = (d, i)$ เป็นแม่ของ n ถ้า n เป็นลูกทางขวาแล้ว $C_n = C_p$ นอกนั้น $C_n = (C_p)_{i-1}d$

จากนิยามของทางเดินตรรกะ จึงได้ว่า $C_{('o', 0)}$ = ':' เนื่องจากเป็นราก, $C_{('i', 0)}$ = 'o' และ $C_{('e', 0)}$ = ':' เนื่องจากเป็นลูกทางซ้ายและลูกทางขวาของ ('o', 0) ตามลำดับ

โหนด n' เป็นลูกตรรกะ (logical child) ของโหนด (d, i) หรือกล่าวได้อีกอย่างหนึ่งว่า (d, i) เป็นแม่ตรรกะ (logical parent) ของ n' ถ้า

1) n' เป็น descendant ของ (d, i) ที่ทำให้ $C_{n'}$ ลงท้ายด้วย d ณ ดิจิตที่ i
2) n' เป็นโหนดใบหรือ $(d', i+1)$

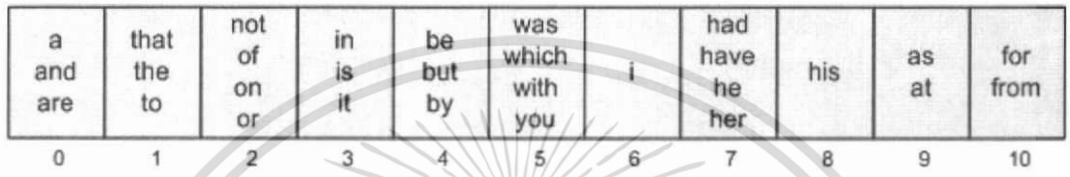
ดังนั้น ('_', 1) จึงเป็นลูกตรรกะของ ('i', 0) และ ('e', 1) เป็นลูกตรรกะของ ('h', 0) ในขณะที่ ('i', 0) ไม่ใช่ลูกตรรกะของ ('o', 0) เนื่องจากไม่ผ่านข้อ 2)

สำหรับ Standard representation นั้นจะอยู่ในรูปของ linked list โดยสมาชิกแต่ละตัวเรียกว่า "เซลล์ (cell)" ดังรูปที่ 2.1(d) ซึ่งประกอบไปด้วย 4 เขตข้อมูลด้วยกันคือ DV (digit value),

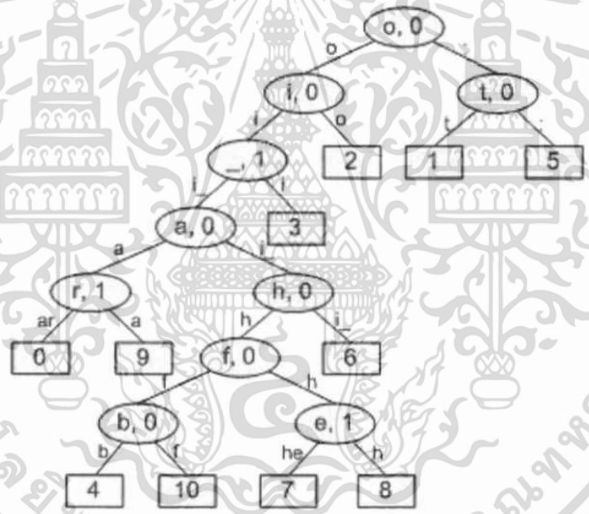
DN (digit number), LP (left pointer) และ RP (right pointer) DV และ DN เก็บค่าดิจิตและตำแหน่งของ DV ตามลำดับ ซึ่งเป็นค่าที่เก็บอยู่ในโหนดภายใน ส่วน LP และ RP อาจเก็บเป็นค่าที่อยู่เซลล์หรือที่อยู่บัคเก็ต ถ้าค่าเป็นลบแสดงว่าเป็นค่าที่อยู่เซลล์ แต่ถ้าค่าเป็นบวกแสดงว่าเป็นค่าที่อยู่บัคเก็ต มีตัวอย่างประกอบอยู่ในรูปที่ 2.1(e)

the, of, and, to, a, in, that, is, i, it, for, as, with, was, his, he, be,
not, by, but, have, you, which, are, on, or, her, had, at, from, this

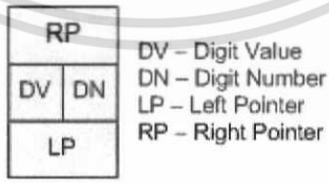
(a) ลำดับการเพิ่ม



(b) บัคเก็ต



(c) Trie และทางเดินตรรกะไปยังโหนดต่างๆ



(d) โครงสร้างของเซลล์

a	is	i	as	was	be	by	had	at	from
-4	2	3	-5	5	6	-7	8	9	10
o, 0	i, 0	_, 1	a, 0	t, 0	h, 0	f, 0	e, 1	r, 1	b, 0
-1	-2	-3	-8	1	-6	-9	7	0	4
0	1	2	3	4	5	6	7	8	9

(e) Standard representation

รูปที่ 2.1 ตัวอย่างแฟ้มข้อมูล

2.1.2 ค้นหาคีย์

ทำการค้นหาโดยการท่องไปใน trie ดังอัลกอรึทึมข้างล่างนี้ ซึ่งให้ผลลัพธ์กลับมาเป็นที่อยู่บัคเก็ตและทางเดินตรรกยะไปยังโหนดที่อยู่บัคเก็ตดังกล่าว

Algorithm 1 TH Key Search: ให้ $c = c_0c_1c_2...$ แทนคีย์ที่ค้นหา, r แทนราก, $n = (d, i)$ แทนโหนดที่เยี่ยมอยู่, $L(n)$ และ $R(n)$ แทนลูกทางซ้ายและลูกทางขวาของ n ตามลำดับ และ C แทนทางเดินตรรกะ

Algorithm 1 TH Key Search

```
1:  n ← r; C = ''; j = 0;
2:  while n is an internal node do
3:    if j = i then
4:      if c_j ≤ d then
5:        n ← L(n); C ← (C)_{i-1}d;
6:        if c_j = d then j ← j + 1;
7:        end if
8:      else n ← R(n);
9:      end if
10:   else if j < i then
11:     n ← L(n); C ← (C)_{i-1}d;
12:   else n ← R(n);
13:   end if
14: end while
15: return n, C
```

จากอัลกอริทึมดังกล่าวจะเห็นได้ว่า ในกรณีที่ $c_j \le d$ เมื่อ $j = i$ หรือกรณีที่ $j < i$ นั้นจะท่องไปทางซ้าย ส่วนกรณีที่เหลือท่องไปทางขวา

ตัวอย่างการค้นหาคีย์ เช่น 'he' เริ่มต้นเปรียบเทียบด้วย c_j = 'h' ทำการเมื่อทำการท่องไป trie จะได้ว่าท่องไปทางซ้ายจนกระทั่งพบโหนด ('a', 0) จึงท่องไปทางขวาและพบกับ ('h', 0) ทำให้

c_j เปลี่ยนแปลงเป็น c_j = 'e' และทำการเปรียบเทียบต่อไปจนสุดท้ายได้คำตอบเป็นที่อยู่บัคเก็ตหมายเลข 6

2.1.3 การแบ่งบัคเก็ต

สำหรับแฮชชิงทรัยแล้วจะใช้คำนำหน้าที่สั้นที่สุดของคีย์กลางเป็นตัวแบ่งคีย์ คีย์ใดในบัคเก็ตที่ล้นซึ่งมีคำนำหน้ามาหลังคำดังกล่าวจะถูกย้ายไปที่บัคเก็ตใหม่ โดยขั้นตอนวิธีนั้นจะเป็นไปตามอัลกอรึทึมข้างล่างนี้

Algorithm 2 TH Bucket Splitting: ให้ n แทนโหนดที่เยี่ยมอยู่, M แทนที่อยู่บัคเก็ตที่ล้น, N แทนที่อยู่บัคเก็ตล่าสุดในแฟ้มข้อมูล, $c' = c'_0c'_1c'_2...$ แทนคีย์กลาง, c แทนคีย์ใดๆในบัคเก็ต M, C แทนทางเดินตรรกะที่ได้จาก Algorithm 1 และ $L(n)$ และ $R(n)$ แทนลูกทางซ้ายและลูกทางขวาของ n ตามลำดับ

Algorithm 2 TH Bucket Splitting

1: $n \leftarrow M$;
2: compute shortest prefix $(c')_i$
3: $N \leftarrow N + 1$;
4: Append bucket N
5: Move all c where $(c)_i > (c')_i$ to bucket N
6: compute the largest $(c')_l$ such that $(c')_l = (C)_l$
7: **for** $j = l + 1$ to $i - 1$ **do**
8: $n \leftarrow (c'_j, j)$; $L(n) \leftarrow M$; $R(n) \leftarrow nil$;
9: $n \leftarrow L(n)$;
10: **end for**
11: $n \leftarrow (c'_i, i)$; $L(n) \leftarrow M$; $R(n) \leftarrow N$;

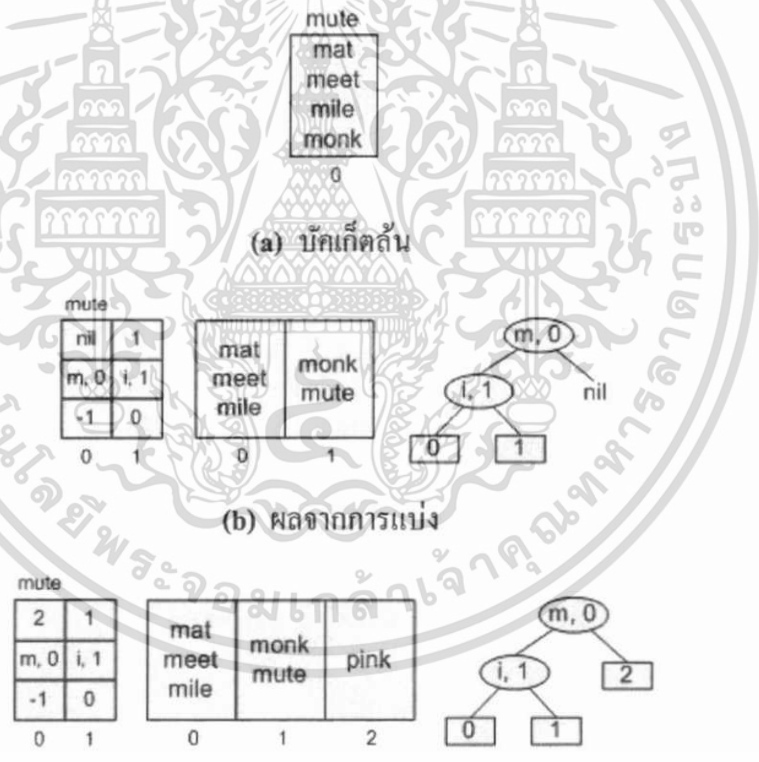
จากอัลกอรึทึมดังกล่าว สามารถสรุปออกมาได้เป็นขั้นตอนดังนี้

1) คำนวณหาคำนำหน้าที่สั้นที่สุดของคีย์กลาง $(c')_i$ ซึ่งสามารถแบ่งคีย์ c โดยที่ $(c)_i > (c')_i$ ไปยังบัคเก็ตใหม่ได้
2) เพิ่มค่า N, ต่อบัคเก็ต N แล้วย้ายคีย์ที่มีสมบัติดังกล่าว ในข้อ 1) ไปยังบัคเก็ต N
3) คำนวณ $(c')_l$ ซึ่งมีค่าตรงกับทางเดินตรรกะ เพื่อใช้หาว่ามีการขยายโหนดเพิ่มขึ้นมากี่โหนด โดยจำนวนโหนดที่ขยายเท่ากับ $i - 1$
   - ถ้าเพียงโหนดเดียว (c'_i, i) ทำการแทนที่ M และ M กลายเป็นลูกทางซ้ายของโหนดนี้ ส่วนลูกทางขวาคือ N

- แต่ถ้ามากกว่า 1 โหนด (c'_{l+1}, l+1) ทำการแทนที่ M และแต่ละโหนด (c'_j, j) โดยที่ $j = l$+1, ..., i-1 จะมีลูกทางซ้ายเป็น (c'_{j+1}, j+1) และลูกทางขวาเป็น *nil* ส่วน (c'_i, i) ยังคงมีลูกทางซ้ายและลูกทางขวาเป็น M และ N ตามลำดับ

2.1.4 การเพิ่ม

เริ่มจากทำการค้นหาบัคเก็ตที่จะทำการเพิ่มคีย์ก่อน โดยการหาที่อยู่บัคเก็ตตามขั้นตอนวิธีตาม Algorithm 1 แล้วทำการดึงบัคเก็ตที่ได้ลงในหน่วยความจำหลัก ถ้าบัคเก็ตดังกล่าวยังไม่เต็มสามารถทำการเพิ่มได้ทันที แต่ถ้าเกิดเต็ม ต้องทำการแบ่งบัคเก็ตออกเป็น 2 บัคเก็ต ตาม Algorithm 2 ซึ่งโดยทั่วๆไปมักจะขยายเพียงโหนดเดียว แต่ถ้าเกิดมีการขยายมากกว่า 1 โหนด จะมีโหนด *nil* เพิ่มขึ้นมาดังตัวอย่างในรูปที่ 2.2(b) และถ้ามีการเพิ่มคีย์ใหม่ โดยพบเจอ *nil* จะมีการเปลี่ยนเป็นบัคเก็ตใหม่ดังรูปที่ 2.2(c)



(a) บัคเก็ตล้น

(b) ผลจากการแบ่ง

(c) เพิ่มคีย์ 'pink'

รูปที่ 2.2 ตัวอย่างการแบ่งในกรณีที่ขยายมากกว่า 1 โหนด

2.1.5 การลบ

เริ่มจากทำการค้นหาบัคเก็ตทำนองเดียวกับการเพิ่ม จากนั้นจึงทำการลบระเบียนที่ระบุโดยคีย์ที่ต้องการลบ ให้โหนดภายนอกที่เก็บค่าที่อยู่บัคเก็ตซึ่งได้จาก Algorithm 1 แทนด้วย P และโหนดพี่น้อง (sibling) ของ P แทนด้วย Q ซึ่งต้องเป็นโหนดภายนอกเช่นกัน หลังจากการลบนั้น ถ้าจำนวนระเบียนในบัคเก็ต P และ Q รวมกันได้ไม่เกิน b ให้ทำการรวมโหนดกัน (merge) เป็นโหนดภายนอกทางซ้าย หด trie ขึ้นไปเรื่อยๆจนกระทั่งพบโหนดพี่น้องไม่ที่เป็น *nil* ยกตัวอย่างเช่น จากรูปที่ 2.1(c) หลังจากลบระเบียนในบัคเก็ต 0 ปรากฏว่าจำนวนรวมของระเบียนในบัคเก็ต 0 และ 9 ไม่เกิน 4 จึงทำการรวมโหนดเหลือโหนด 0 และหด trie ขึ้น เปลี่ยนโหนด 0 เป็นลูกทางซ้ายของ ('a', 0)

2.1.6 การปรับปรุง

สำหรับวิธีการอัพเดทในที่นี่คือการลบและทำการเพิ่มใหม่ เพื่อทำการรักษาคุณสมบัติ weak order preserving เอาไว้

2.2 แฮชชิงแบบขยายได้ (Extendible Hashing)

แฮชชิงแบบขยายได้ (Extendible hashing) เป็นวิธีเข้าถึงวิธีหนึ่งสำหรับการเก็บและการเข้าถึงระเบียนของแฟ้มข้อมูลยืดหยุ่น โดยมีโครงสร้างข้อมูลที่ยืดหยุ่นสำหรับการเก็บข้อมูลและสามารถเข้าถึงระเบียนที่ต้องการได้โดยทำการเข้าถึงดิสก์เพียง 2 ครั้งเท่านั้น ซึ่งเหมาะกับแฟ้มข้อมูลที่มีการเก็บขนาดใหญ่ รายละเอียดต่างๆสำหรับวิธีนี้สามารถแบ่งออกเป็นเนื้อหาย่อยได้เป็นดังนี้

2.2.1 โครงสร้าง

ประกอบไปด้วย 2 ระดับด้วยกันได้แก่ระดับไดเรกทอรี (directoty) และระดับลีฟ (leaves)

2.2.1.1 ระดับไดเรกทอรี

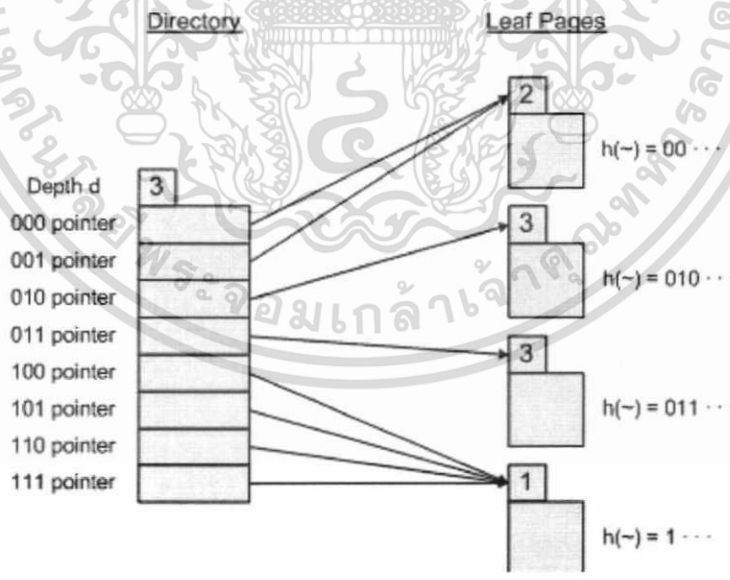
ประกอบด้วยเฮดเดอร์ซึ่งเรียกว่าความลึก (depth) ของไดเรกทอรี แทนค่าด้วย d และไดเรกทอรี ความลึกของไดเรกทอรีนั้นเป็นตัวระบุขนาดของไดเรกทอรีทั้งหมดซึ่งมี 2^d ไดเรกทอรี ส่วนไดเรกทอรีจะมีพอยน์เตอร์ (pointer) ชี้ไปยังลีฟเพจซึ่งเป็นตัวเก็บคีย์และข้อมูลที่ใช้ร่วมกับคีย์ โดยแต่ละพอยน์เตอร์จะถูกกำหนดให้ชี้ไปยังลีฟเพจสำหรับเก็บคีย์ซึ่งได้ค่าแฮชของคีย์ขึ้นต้นด้วย 000...00, 000...01, 000...10, ..., 111...11 ตามลำดับซึ่งค่าขึ้นต้นดังกล่าวมีจำนวน d บิต เรียก

พอยน์เตอร์แต่ละตัวดังกล่าวว่า พอยน์เตอร์ 000...00, พอยน์เตอร์ 000...01, พอยน์เตอร์ 000...01, ..., พอยน์เตอร์ 111..11 ตามลำดับ

2.2.1.2 ระดับลีฟ

ประกอบไปด้วยลีฟเพจซึ่งเป็นตัวเก็บข้อมูลโดยเก็บอยู่ในรูปคู่อันดับ $(K, I(K))$ โดยที่ K แทนคีย์ใดๆ และ $I(K)$ แทนข้อมูลที่ใช้ร่วมกับคีย์ซึ่งเป็นได้ทั้งระเบียนที่มีคีย์ K ระบุอยู่หรือพอยน์เตอร์ไปยังระเบียนที่มีคีย์ K ระบุอยู่ และแต่ละลีฟเพจ (leaf page) จะบรรจุเฮดเดอร์ซึ่งเรียกว่าความลึกเฉพาะ (local depth) ของลีฟเพจ แทนค่าด้วย d' เป็นตัวระบุว่าค่าแฮชของแต่ละคีย์ในลีฟเพจนี้ขึ้นต้นด้วยค่าเดียวกัน d' บิต ($x_{d'}$) และความลึกเฉพาะของแต่ละลีฟเพจจะมีค่าไม่เกินความลึกของไดเรกทอรี พอยน์เตอร์แต่ละตัวนั้นจะชี้ไปยังลีฟเพจที่มีค่าขึ้นต้นด้วย $x_{d'}$ เดียวกัน

ตัวอย่างของโครงสร้างได้แสดงไว้ดังรูปที่ 2.3 โดยกำหนดให้ $d = 3$ จะเห็นได้ว่าในระดับไดเรกทอรีนั้นประกอบด้วยไดเรกทอรีทั้งหมด $2^3 = 8$ ไดเรกทอรี และประกอบด้วยพอยน์เตอร์ 000, พอยน์เตอร์ 001, พอยน์เตอร์ 010, ..., พอยน์เตอร์ 111 ตามลำดับซึ่งชี้ไปยังลีฟเพจต่างๆ ส่วนในระดับลีฟจะเห็นได้ว่า ลีฟเพจที่ 1 มีพอยน์เตอร์ 000 และ 001 ชี้มาซึ่งมีค่าขึ้นต้นด้วย 00, ลีฟเพจที่ 2 มีพอยน์เตอร์ 010 ชี้มา, ลีฟเพจที่ 3 มีพอยน์เตอร์ 011 ชี้มาและลีฟเพจที่ 4 มีพอยน์เตอร์ 100, 101, 110 และ 111 ชี้มา



รูปที่ 2.3 ตัวอย่างโครงสร้างของแฮชชิงแบบขยายได้ เมื่อความลึก $d = 3$

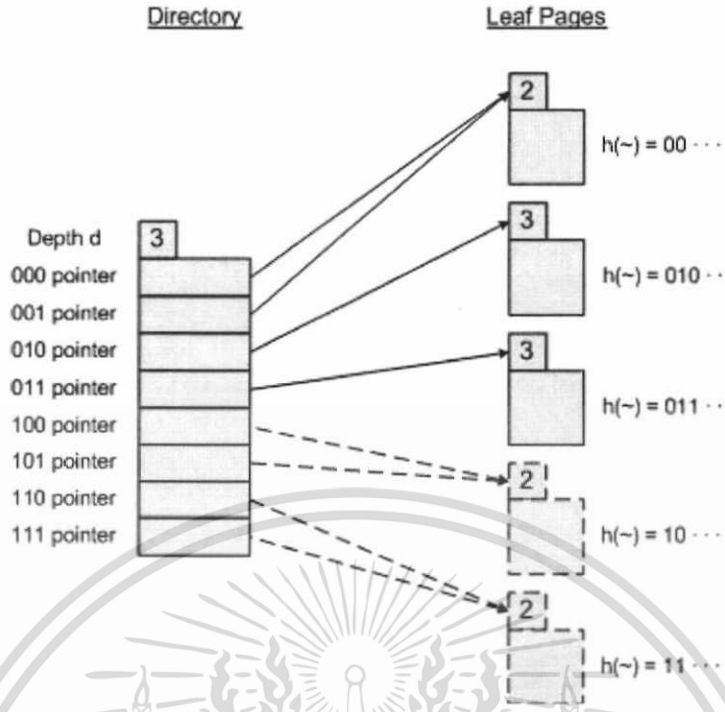
2.2.2 การเข้าถึง

กำหนดให้ K_0 แทนคีย์ที่ต้องการเข้าถึง สามารถทำการเข้าถึงระเบียนโดยมีคีย์ K_0 ระบุอยู่ได้โดยทำการหาค่าแฮช $K_0' = h(K_0)$ ซึ่งอยู่ในรูปเลขฐาน 2 จากนั้นนำ K_0' ที่ได้มาตรวจดูว่าค่าขึ้นต้น d บิตแรกของ K_0' ตรงกับพอยน์เตอร์ใดของไดเรกทอรี เมื่อพบแล้วให้ไปยังลีฟเพจที่มีพอยน์เตอร์ดังกล่าวชี้ ถ้า $I(K_0)$ เก็บเป็นระเบียนก็เป็นอันสิ้นสุดขั้นตอน แต่ถ้า $I(K_0)$ เก็บเป็นพอยน์เตอร์ไปยังระเบียน ต้องทำการชี้ไปยังระเบียนก่อนจึงจะเป็นอันสิ้นสุดขั้นตอน

2.2.3 การแบ่ง

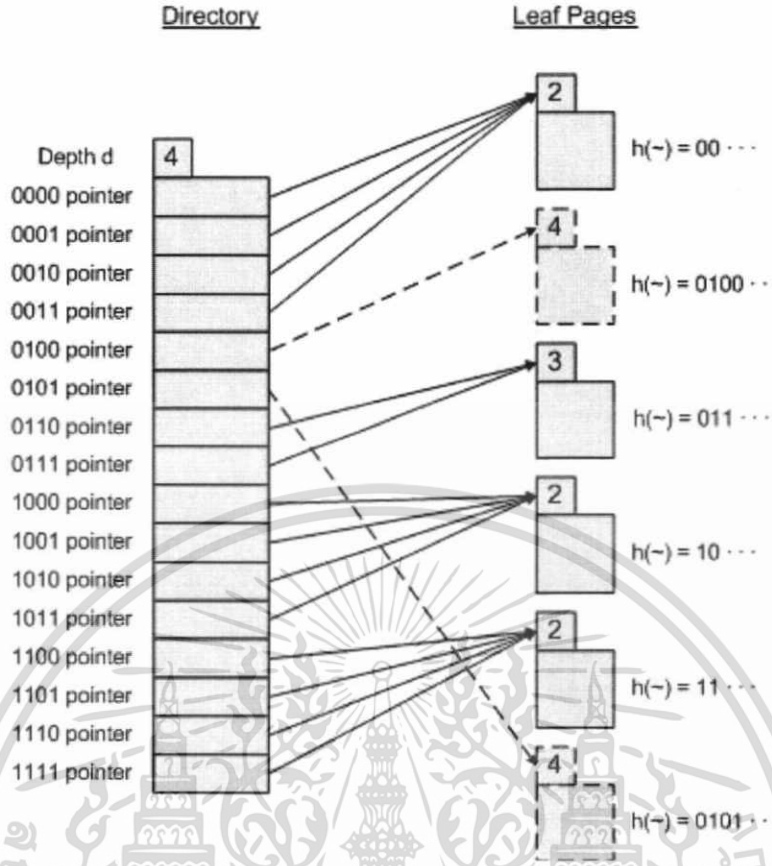
เป็นการแบ่งลีฟเพจออกเป็น 2 เพจเพื่อแก้ปัญหาการล้น (overflow) ของลีฟเพจ ซึ่งวิธีการแบ่งนั้นจะทำโดยสร้างลีฟเพจใหม่ขึ้นมา กำหนดให้ลีฟเพจที่ล้นและลีฟเพจใหม่ดังกล่าวมีความลึกเฉพาะเป็น d'+1 จากนั้นทำการย้ายสมาชิกไปยังลีฟเพจใหม่โดยจะย้ายสมาชิกที่มีคีย์ซึ่งมีค่าแฮชของคีย์ขึ้นต้นด้วย $x_{a'}1$ และสุดท้ายปรับการชี้ของพอยน์เตอร์เพื่อให้สอดคล้องกับรูปแบบของโครงสร้าง หลังจากการแบ่งลีฟเพจถ้าพบว่าความลึกเฉพาะของลีฟเพจที่ทำการแบ่งมีค่ามากกว่าความลึกของไดเรกทอรี จะต้องทำการปรับโครงสร้างข้อมูลในระดับไดเรกทอรีใหม่ โดยการเพิ่มความลึกของไดเรกทอรีไป 1 เพิ่มขนาดของไดเรกทอรีเป็นจำนวน 1 เท่าตัว และปรับการชี้ของแต่ละพอยน์เตอร์เพื่อให้สอดคล้องกับรูปแบบของโครงสร้าง

จากรูปที่ 2.3 เมื่อเกิดการล้นของลีฟเพจที่ 4 ขึ้นจะต้องทำการสร้างลีฟเพจใหม่ขึ้นมา กำหนดให้ลีฟเพจที่ 4 และลีฟเพจใหม่มีความลึกเฉพาะเป็น 1+1 = 2 ทำการย้ายสมาชิกที่มีคีย์ซึ่งมีค่าแฮชของคีย์ขึ้นต้นด้วย 011 ไปยังลีฟเพจใหม่ดังกล่าว และปรับการชี้ของพอยน์เตอร์เพื่อให้สอดคล้องกับรูปแบบของโครงสร้าง ได้ผลลัพธ์ออกมาดังรูปที่ 2.4



รูปที่ 2.4 ตัวอย่างการแบ่งลีฟเพจ

และจากรูปที่ 2.4 ถ้าเกิดการล้นของลีฟเพจที่ 2 ขึ้น หลังจากทำการแบ่งลีฟเพจเป็น 2 เพจ จะเห็นได้ว่าลีฟเพจที่ทำการแบ่งนั้นมีความลึกเฉพาะเป็น 4 ซึ่งมีค่ามากกว่าความลึกของไดเรกทอรี ดังนั้นจึงต้องปรับโครงสร้างข้อมูลระดับไดเรกทอรีใหม่ โดยการเพิ่มความลึกของไดเรกทอรีลงไปอีก 1 เป็น 4 เพิ่มขนาดของไดเรกทอรีอีก 1 เท่าตัวเป็น 8 และทำการปรับการชี้ของแต่ละพอยน์เตอร์ ได้ผลลัพธ์ออกมาดังรูปที่ 2.5



รูปที่ 2.5 ตัวอย่างการปรับโครงสร้างข้อมูลระดับไดเรกทอรีใหม่ในกรณีการแบ่ง

2.2.4 การเพิ่ม

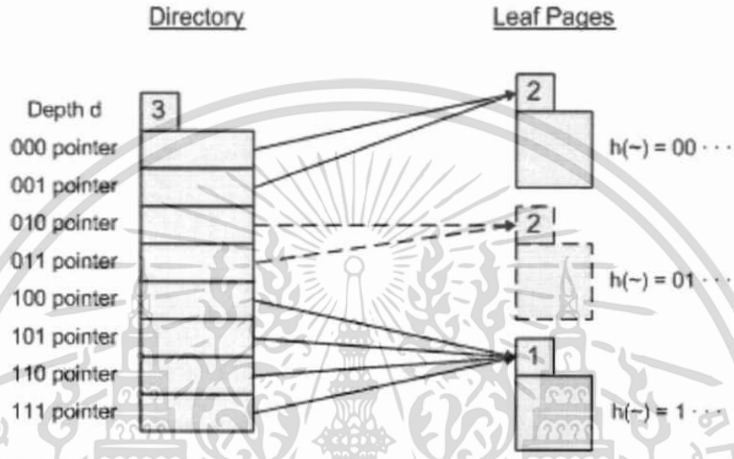
กำหนดให้ $(K_0, I(K_0))$ เป็นข้อมูลที่ต้องการเพิ่ม วิธีการเพิ่มนั้นจะเริ่มจากการทำวิธีการเข้าถึงเพื่อไปยังลีฟเพจที่ต้องการเพิ่ม $(K_0, I(K_0))$ และถัดจากนั้นจึงทำการเพิ่ม $(K_0, I(K_0))$ ลงไป หลังจากการเพิ่มถ้าพบว่าเกิดการล้น (overflow) ของลีฟเพจ ให้ทำวิธีการแบ่งเพื่อแก้ไขปัญหาการล้นดังกล่าว

2.2.5 การลบ

กำหนดให้ K_0 แทนคีย์ที่ต้องการลบ สำหรับวิธีการลบนั้นจะเริ่มจากทำวิธีการเข้าถึงเพื่อไปยังลีฟเพจที่มีการเก็บคีย์ K_0 อยู่และถัดจากนั้นจึงทำการลบ $(K_0, I(K_0))$ หลังจากทำการลบถ้าพบว่าผลรวมของจำนวนสมาชิกในลีฟเพจที่มีการลบ $(K_0, I(K_0))$ ดังกล่าวกับลีฟเพจพี่น้อง (ลีฟเพจที่มีความลึกเฉพาะและค่า $x_{d'-1}$ เท่ากัน) มีค่าไม่เกินความจุของลีฟเพจ ให้ทำการรวมเป็นลีฟเพจเดียวกันและลดความลึกเฉพาะของลีฟเพจลงไป 1 และหลังจากรวมลีฟเพจถ้าพบว่าแต่ละลีฟเพจมีความลึกเฉพาะน้อยกว่าความลึกของไดเรกทอรี ให้ทำการลดความลึกของไดเรกทอรีไป 1 และลดขนาดของ

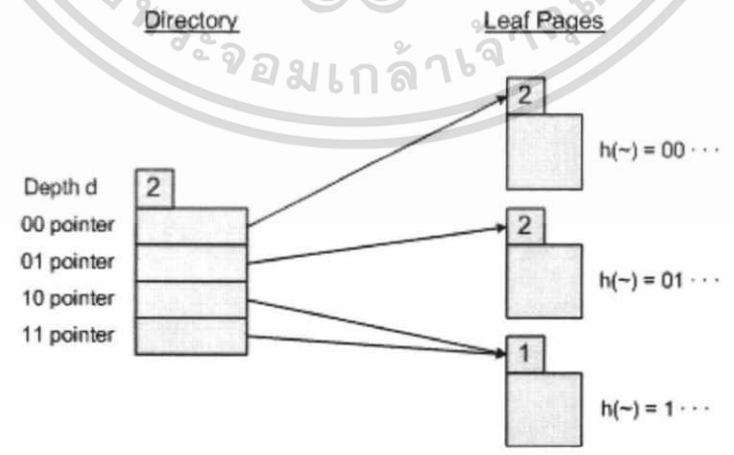
ไดเรกทอรีเป็นจำนวนครึ่งหนึ่งของทั้งหมดพร้อมทั้งปรับการชี้ของแต่ละพอยน์เตอร์ให้สอดคล้องกับรูปแบบของโครงสร้าง

จากรูปที่ 2.3 ถ้าทำการลบคีย์ในลีฟเพจที่ 2 แล้วปรากฏว่าผลรวมของจำนวนสมาชิกในลีฟเพจที่ 2 ($h(\sim) = 010$) กับลีฟเพจพี่น้องซึ่งตรงกับลีฟเพจที่ 3 ($h(\sim) = 011$) มีค่าไม่เกินความจุของลีฟเพจ จะทำการรวมเป็นลีฟเพจเดียวกันและลดความลึกเฉพาะของลีฟเพจจาก 3 เป็น 2 ได้ออกมาเป็นดังรูปที่ 2.6



รูปที่ 2.6 ตัวอย่างการรวมลีฟเพจ

และถัดมาเมื่อพบว่าความลึกเฉพาะของแต่ละลีฟเพจนั้นมีค่าน้อยกว่าความลึกของไดเรกทอรี จะทำการลดความลึกของไดเรกทอรีลงไป 1 เหลือ 2 และทำการลดขนาดของไดเรกทอรีลงไปครึ่งหนึ่งเหลือ 4 พร้อมทั้งปรับการชี้ของแต่ละพอยน์เตอร์ ได้ผลลัพธ์ออกมาดังรูปที่ 2.7



รูปที่ 2.7 ตัวอย่างการปรับโครงสร้างข้อมูลระดับไดเรกทอรีใหม่ในกรณีการลบ

2.2.6 การปรับปรุง

ทำการลบและเพิ่มลงไปเพื่อรักษาความถูกต้องของรูปแบบโครงสร้างไว้

2.3 งานวิจัยที่เกี่ยวข้อง

เนื่องจาก Trie hashing นั้นจัดว่าเป็นวิธีเข้าถึงที่มีประสิทธิภาพมากวิธีหนึ่ง จึงได้มีผู้นำวิธีดังกล่าวมาทำการพัฒนาหรือปรับปรุงเพื่อจุดประสงค์ต่างๆ ดังต่อไปนี้

E. J. Otoo และ S. Effah [6] นำเทคนิคการปรับสมดุลของ Red-Black binary search tree มารวมกับ Trie hashing ทำให้เกิดเป็น Trie hashing ที่มี trie แบบสมดุลเป็นตัวคำนวณ เพื่อแก้ปัญหา degeneracy ซึ่งโครงสร้างดังกล่าวมีความสูง $O(log\ n_p)$ โดยที่ n_p แทนจำนวนบัคเก็ต ให้ค่า storage utilization ของข้อมูลโดยเฉลี่ย 67%

W. A. Litwin และคณะ [5] นำ Trie hashing มาทำการปรับปรุงเพื่อควบคุมให้สามารถเก็บระเบียนลงในบัคเก็ตอย่างคุ้มค่าสูงสุดหรือกล่าวอีกนัยหนึ่งคือมีค่า load factor ที่สูงสุด โดยการกำจัด *nil* node, ควบคุมการแบ่งบัคเก็ต, รวมบัคเก็ต และ redistribution ผลปรากฏว่าให้ค่า load factor สูงสุดถึง 100% สำหรับลำดับการเพิ่มแบบเรียงลำดับ และเปลี่ยนจาก 70% เป็นค่าที่สูงมากกว่า 85% สำหรับลำดับการเพิ่มแบบสุ่ม

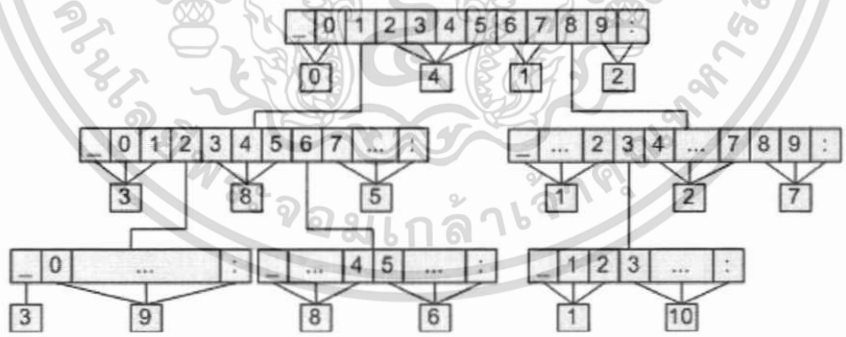
บทที่ 3
แฮชชิงทรัยแบบขยายได้

แฮชชิงทรัยแบบขยายได้ (Extendible trie hashing) เป็นวิธีใหม่ที่พัฒนาขึ้นมาโดยใช้โครงสร้างข้อมูลซึ่งมีชื่อเรียกว่า "extendible trie" ในการคำนวณหาที่อยู่บัคเก็ต โดยโครงสร้างข้อมูลดังกล่าวเกิดจากการรวมกันระหว่างวิธีแฮชชิงแบบขยายได้และแฮชชิงทรัย

3.1 โครงสร้างข้อมูล

โหนดภายในแต่ละโหนดประกอบไปด้วยเขตข้อมูลซึ่งเรียกว่า "ไดเรกทอรี (directory)" โดยจะมีจำนวนเท่ากับ $\sum$ ถ้า $\sum$ แทนเซตของตัวอักษรภาษาอังกฤษ จะมีจำนวนไดเรกทอรีทั้งหมด 28 ไดเรกทอรีเพราะรวม '_' และ ':' ด้วยหรือถ้า $\sum$ แทนเซตของเลขฐาน 10 จะมีไดเรกทอรีทั้งหมด 12 ไดเรกทอรี แต่ละไดเรกทอรีจะมีดิจิตใน $\sum$ กำกับอยู่และเรียงลำดับด้วย ใช้เป็นชื่อไดเรกทอรี ส่วนโหนดภายนอกหรือโหนดใบเป็นที่อยู่บัคเก็ต โครงสร้างนี้ใช้เนื้อที่ในการสร้าง $O(|\sum|^m)$ และมีลักษณะดังต่อไปนี้

1) โหนดภายในแต่ละโหนดมีแม่เป็นไดเรกทอรีเพียงไดเรกทอรีเดียวเท่านั้น
2) โหนดภายนอกแต่ละโหนดอาจมีแม่เป็นไดเรกทอรีเพียงหนึ่งหรือหลายไดเรกทอรีซึ่งอยู่ติดกัน



รูปที่ 3.1 ตัวอย่างโครงสร้างของแฮชชิงทรัยแบบขยายได้เมื่อ $\sum$ แทนเซ็ตของเลขฐานสิบ

3.2 การค้นหาคีย์

สำหรับการค้นหาคีย์นั้น จะใช้ดิจิตของคีย์ ณ ตำแหน่งเดียวกับระดับของโหนดที่เยี่ยมอยู่เป็นเกณฑ์ในการท่องเข้าไป หมายความว่าโหนดที่เยี่ยมอยู่ตรงกับระดับใด ให้เข้าไปที่ไดเรกทอรีที่มีชื่อตรงกับดิจิตของคีย์ ณ ตำแหน่งนั้น ตัวอย่างเช่น ต้องการค้นหาคีย์ 'choose' และโหนดที่เยี่ยมอยู่ที่ระดับ 0 (ราก) ตำแหน่งที่ 0 ของคีย์ตัวนี้คือ 'c' ให้เข้าไปที่ไดเรกทอรี c หรือถ้าอยู่ที่ระดับ 1 ให้เข้า

ไปที่ไดเรกทอรี h เป็นต้น ซึ่งสามารถสรุปเป็นขั้นตอนวิธีได้ดังอัลกอริทึมข้างล่างนี้ โดยให้ผลลัพธ์กลับคืนมาเป็นที่อยู่บัคเก็ตและโหนดๆหนึ่งสำหรับใช้ในขั้นตอนวิธีการแบ่งบัคเก็ต

Algorithm 3 EXTH Key Search: ให้ $k = k_0k_1k_2...$ แทนคีย์ที่ค้นหา, r แทนราก, N แทนโหนดที่เยี่ยมอยู่, $N[A]$ แทนไดเรกทอรี A ใน N และ P แทนโหนดๆหนึ่ง

Algorithm 3 EXTH Key Search

```
1:  N ← r; i = 0;
2:  while N is an internal node do
3:      P ← N; N ← N[k_i]; i ← i + 1;
4:  end while
5:  return N, P
```

ตัวอย่างเช่น จากรูปที่ 3.1 ถ้าคีย์ที่ใช้ค้นหาคือ '14203' ณ ราก (ระดับ 0) เข้าไปที่ไดเรกทอรี 1 ท่องไปยังโหนดถัดไป (ระดับ 1) จากนั้นเข้าไปยังไดเรกทอรี 4 ท่องไปยังที่อยู่บัคเก็ตหมายเลข 8 ฉะนั้นผลลัพธ์ที่คืนกลับมาจึงเป็นที่อยู่หมายเลข 8 และโหนดที่ ได้เข้าเยี่ยมที่ระดับ 1 ดังกล่าว

3.3 การแบ่งบัคเก็ต

การแบ่งบัคเก็ตเมื่อบัคเก็ตเกิดล้นนั้น มีขั้นตอนวิธีเป็นไปตาม Algorithm 4 โดยใช้หลักการเดียวกันกับแฮชชิงทรัย แต่จะมีการกำจัด *nil* ด้วยโดยการแทนเป็นที่อยู่บัคเก็ตที่เพิ่มล่าสุดลงไป เพื่อเป็นการช่วยเพิ่มค่าโหลดแฟกเตอร์

Algorithm 4 EXTH Bucket Splitting: ให้ N แทนโหนดที่เยี่ยมอยู่, $N[A]$ แทนไดเรกทอรี A ใน N, B แทนที่อยู่บัคเก็ตที่ล้น, B' แทนที่อยู่บัคเก็ตล่าสุดในแฟ้มข้อมูล, $c' = c'_0c'_1c'_2...$ แทนคีย์กลาง, c แทนคีย์ใดๆในบัคเก็ต B, P แทนโหนดที่ได้จาก Algorithm 3 และ l แทนระดับ P อยู่

Algorithm 4 EXTH Bucket Splitting

```
1:  N ← P;
2:  compute (c')_l'
3:  B' ← B' + 1;
4:  Append bucket B'
5:  Move all c where (c)_l' > (c')_l' to B'
6:  if l' > l then
7:    for i = l to l' - 1 do
8:      Create new node N'
9:      N[c'_i] ← N'; N ← N[c'_i];
10:     for each alphabet alp in P do
11:       if alp ≤ c'_{i+1} then N[alp] ← B;
```

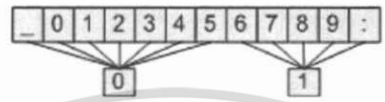
12: **else** $N[alp] \leftarrow B'$;
13: **end if**
14: **end for**
15: **end for**
16: $N \leftarrow P$;
17: **end if**
18: **for** each alphabet *alp* in P where $alp > c'_l$ *and* $N[alp] = B$ **do**
19: $N[alp] \leftarrow B'$;
20: **end for**
21: *nextalp* $\leftarrow$ alphabet where it is next to the last *alp*;
22: **while** $N[nextalp]$ is an internal node **do**
23: **for** each alphabet *alp* in P where $N[alp] = B$ **do**
24: $N[alp] \leftarrow B'$;
25: **end for**
26: *nextalp* $\leftarrow$ alphabet where it is next to the last *alp*;
27: **end while**

จากอัลกอริทึมดังกล่าวสามารถสรุปเป็นขั้นตอนได้ดังนี้

1) คำนวณหาคำนำหน้าที่สั้นที่สุดของคีย์กลาง $(c')_{l'}$ ซึ่งสามารถแบ่งคีย์ c โดยที่ $(c)_{l'} > (c')_{l'}$ ไปยังบัคเก็ตใหม่ได้
2) เพิ่มค่า B', ต่อบัคเก็ต B' แล้วย้ายคีย์ที่มีสมบัติดังกล่าว ในข้อ 1) ไปยังบัคเก็ต B'
3) นำ l และ l' มาทำการเปรียบเทียบกัน
   - ถ้า $l' = l$ จะเป็นการขยายบัคเก็ตเพียงอย่างเดียว โดยไดเรกทอรีที่มีชื่อตั้งแต่หลังดิจิต c'_l เป็นต้นไปที่โยงไปยัง B ในระดับ l และไดเรกทอรีทั้งหมดที่โยงไปยัง B ในระดับหลัง l ลงไปนั้นจะเปลี่ยนเป็นโยงไป ยัง B'
   - ถ้า $l' > l$ จะเป็นทั้งการขยายโหนดและขยายบัตเก็ตด้วย โดยแต่ละระดับ $i = l,...,l'$-1 ไดเรกทอรี c'_i จะโยงไปยังโหนดใหม่ N' และให้ไดเรกทอรีใน N' ที่มีชื่อก่อนหรือตรงกับดิจิต c'_{i+1} โยงไปยัง B ส่วนไดเรกทอรีที่เหลือโยงไปยัง B' นอกจากนี้ ไดเรกทอรีที่มีชื่อหลังดิจิต c'_l เป็นต้นไปที่โยงไป B ในระดับ l และไดเรกทอรีทั้งหมดที่โยงไปยัง B ในระดับหลัง l ลงไปนั้นจะเปลี่ยนเป็นโยงไปยัง B' ด้วย

รูปที่ 3.2 แสดงตัวอย่างผลการแบ่งบัคเก็ตในกรณีต่างๆ โดยทั่วๆไปนั้นมักจะพบในกรณีรูปที่ 3.2(a) โดยในที่นี้ให้ l = 0 และ $(c')_{l'}$ = '5' นั่นคือตั้งแต่หลังไดเรกทอรี 5 เป็นต้นไปในระดับ 0 เปลี่ยนเป็นโยงไปยังที่อยู่หมายเลข 1

และน้อยครั้งที่พบกรณีดังรูปที่ 3.2(b) ซึ่งในที่นี้ให้ $l = 0$ และ $(c')_{l'} =$ '53' ไดเรกทอรี 5 ที่ระดับ 0 ทำการโยงไปยังโหนดใหม่ และให้ไดเรกทอรีที่มีชื่อก่อนหรือตรงกับ 3 โยงไปยังที่อยู่หมายเลข 0 ส่วนที่เหลือโยงไปที่อยู่หมายเลข 1 นอกจากนี้ ไดเรกทอรีที่มีชื่อตั้งแต่หลัง 5 เป็นต้นไปในระดับ 0 เปลี่ยนเป็นโยงไปยังที่อยู่หมายเลข 1 ด้วย และจากรูปที่ 3.2(b) ถ้าบัคเก็ต 0 ล้นอีก และ P ที่ได้จาก Algorithm 3 เป็นโหนดที่ได้เข้าเยี่ยมที่ระดับ 0 นอกจากเปลี่ยนการโยงของไดเรกทอรีที่ระดับ 0 แล้ว ยังต้อง เปลี่ยนการโยงตั้งแต่ระดับ 1 ลงไปด้วยดังรูปที่ 3.2(c)



(a) กรณีที่ $l = l'$

(b) กรณีที่ $l < l'$

(c) กรณีที่มีโหนดมากกว่า 1 ระดับ โยงไปยังบัคเก็ตที่ล้น

รูปที่ 3.2 ตัวอย่างผลการแบ่งบัคเก็ตในกรณีต่างๆ

3.4 การเพิ่ม

ทำโดยการหาที่อยู่บัคเก็ตตามขั้นตอนวิธีตาม Algorithm 3 จากนั้นทำการดึงบัคเก็ตที่มีที่อยู่ดังกล่าวลงในหน่วยความจำหลัก ถ้าบัคเก็ตดังกล่าวยังไม่เต็ม สามารถทำการเพิ่มได้ทันที ในกรณีที่เต็มต้องทำการแบ่งบัคเก็ตตาม Algorithm 4

3.5 การลบ

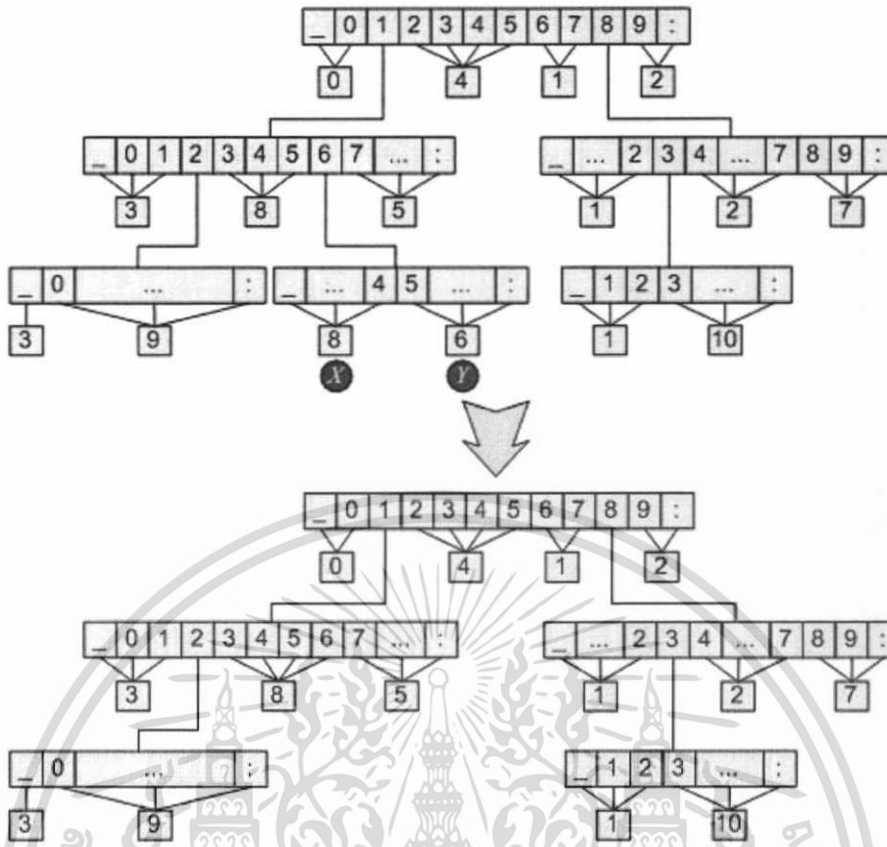
เริ่มจากการค้นหาบัคเก็ตทำนองเดียวกับการเพิ่ม จากนั้นจึงทำการลบระเบียนที่ระบุโดยคีย์ที่ต้องการลบ ให้ Y แทนโหนดภายนอกที่ได้จาก Algorithm 3, X แทนโหนดก่อนหน้า Y ที่ถูกโยงอยู่ติดกัน และ Z แทนโหนดถัดจาก Y ที่ถูกโยงอยู่ติดกัน หลังจากการลบ ถ้ามีคุณสมบัติตรงกับข้อใดข้อหนึ่งดังนี้คือ

1) X เป็นโหนดภายนอกและผลรวมของจำนวนระเบียนในบัคเก็ต X กับ Y มีค่าไม่เกิน b

2) Z เป็นโหนดภายนอกและผลรวมของจำนวนระเบียนในบัคเก็ต Y กับ Z มีค่าไม่เกิน b

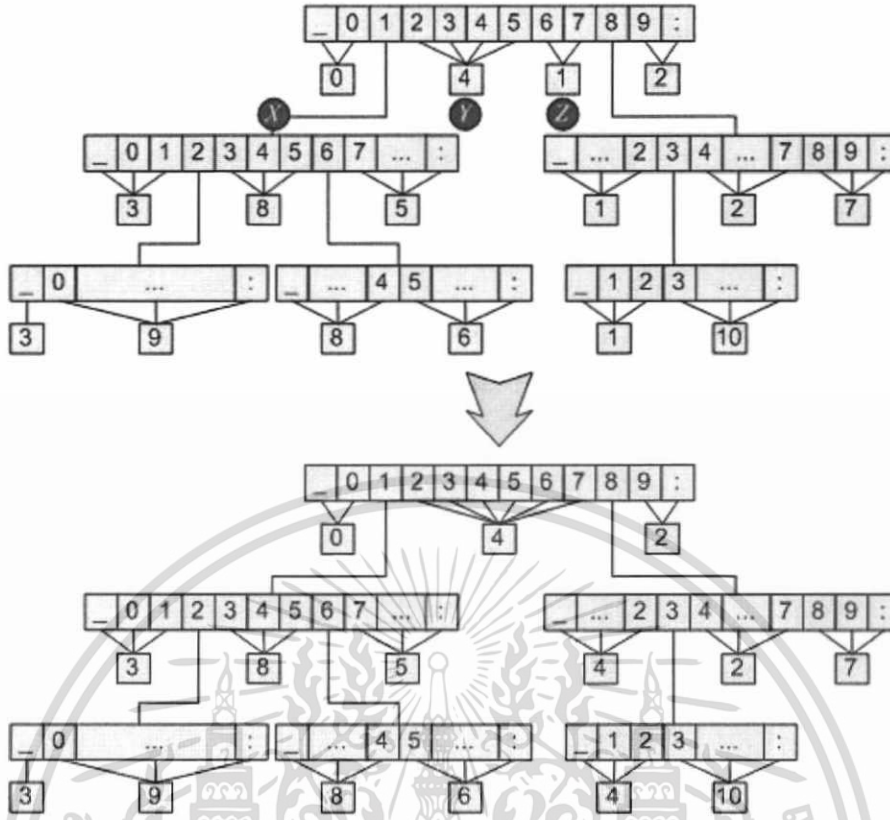
ให้ทำการรวมโหนดซึ่งได้ผลลัพธ์ออกมาเป็นดังนี้คือ ถ้าตรงกับ 1) ให้รวมเป็น X แต่ถ้าตรงกับ 2) ให้รวมเป็น Y ถ้าตรงกันทั้ง 2 ข้อให้เลือกข้อใดข้อหนึ่ง และหลังจากรวมโหนด ถ้าพบว่าเหลือโหนดภายนอกเพียงโหนดเดียวที่ถูกโยงอยู่กับโหนดแม่ ให้ทำการหด extendible trie ขึ้นไป มีตัวอย่างการลบประกอบอยู่ในย่อหน้าข้างล่างนี้ โดยกำหนดให้ในที่นี้ผลรวมของจำนวนระเบียนในสองบัคเก็ตใดๆ มีค่าไม่เกิน b

จากรูปที่ 3.1 ถ้าทำการลบระเบียนในบัคเก็ต 6 จะได้ว่า X คือโหนด 8 ซึ่งเป็นโหนดภายนอก ส่วน Z ไม่มี ดังนั้นจึงตรงคุณสมบัติข้อ 1) จึงรวมโหนดกันเป็นโหนด 8 และกลายเป็นโหนดเดียวที่ถูกโยงกับโหนดแม่อยู่ ณ ขณะนี้ ฉะนั้นจึงทำการหด extendible trie ขึ้นไปด้วย ได้ผลลัพธ์ออกมาดังรูปที่ 3.3



รูปที่ 3.3 ผลลัพธ์จากการลบระเบียนในบัคเก็ต 6

และจากรูปที่ 3.1 ถ้าทำการลบระเบียนในบัคเก็ต 4 จะได้ว่า X คือโหนดภายใน ส่วน Z คือโหนด 1 ดังนั้นจึงตรงคุณสมบัติข้อ 2) จึงรวมโหนดกันเป็นโหนด 4 และนอกจากนั้นไดเรกทอรีในระดับอื่นที่โยงไปยังโหนด 1 ถูกเปลี่ยนเป็นโหนด 4 ดังรูปที่ 3.4



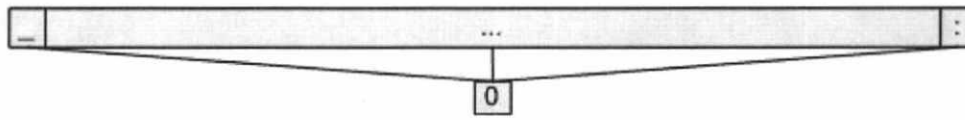
รูปที่ 3.4 ผลลัพธ์จากการลบระเบียนในบัคเก็ต 4

3.6 การปรับปรุง

ทำในลักษณะเดียวกับแฮชชิงทรัยคือ ทำการลบและทำการเพิ่มใหม่เพื่อทำการรักษาคุณสมบัติ weak order preserving เอาไว้

3.7 การสร้าง

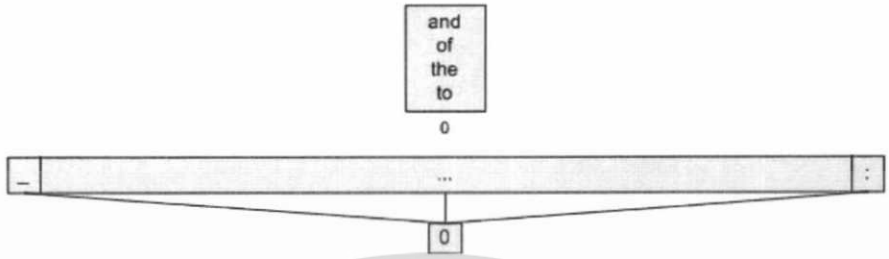
วิธีการสร้าง extendible trie คือการเพิ่มคีย์ลงไปในบัคเก็ตจนเกิดการขยายบัคเก็ตหรือโหนด เมื่อบัคเก็ตล้นในแต่ละครั้ง โดยเริ่มต้นจะมีโหนดภายในอยู่หนึ่งโหนดซึ่งแต่ละไดเรกทอรีนั้นจะโยงไปยังที่อยู่บัคเก็ตหมายเลข 0 ดังรูปที่ 3.5 ผลลัพธ์สุดท้ายของโครงสร้างที่ได้จะขึ้นอยู่กับลำดับของคีย์ที่ใช้ในการเพิ่ม



รูปที่ 3.5 โครงสร้าง ณ สถานะเริ่มต้น

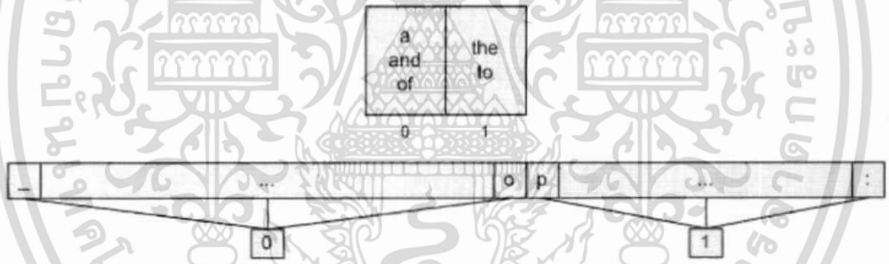
นี่เป็นตัวอย่างแสดงลำดับการสร้าง extendible trie โดยใช้ลำดับคีย์เดียวกับกับรูปที่ 2.1(a) และ bucket capacity = 4

1. เพิ่ม 'the', 'of', 'and', 'to' ลงใน [0]



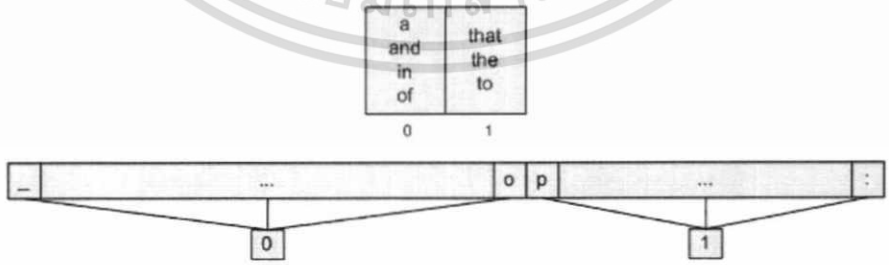
รูปที่ 3.6 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'the', 'of', 'and', 'to' ลงในบัคเก็ต

2. เพิ่ม 'a' ลงใน [0] ทำให้เกิดการล้นบัคเก็ต จึงต้องทำการแบ่งบัคเก็ตตาม Algorithm 4 [c' = 'of', $l = 0$, $l' = 0$] ซึ่งจะทำการสร้างบัคเก็ต 1 และให้ไดเรกทอรีตั้งแต่หลัง 'o' เป็นต้นไปที่โยงไปยังบัคเก็ต 0 โยงไปยังบัคเก็ต 1 แทน



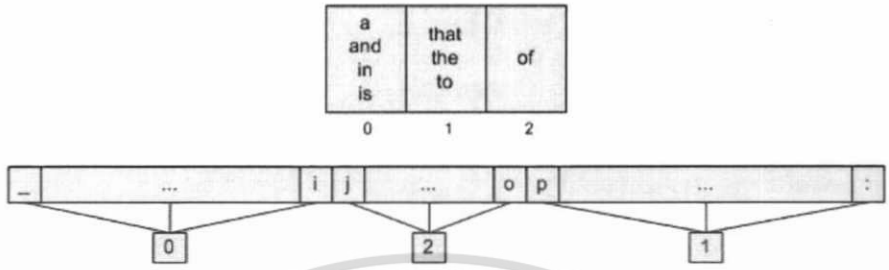
รูปที่ 3.7 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'a' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

3. เพิ่ม 'in' ลงใน [0] และ 'that' ลงใน [1]



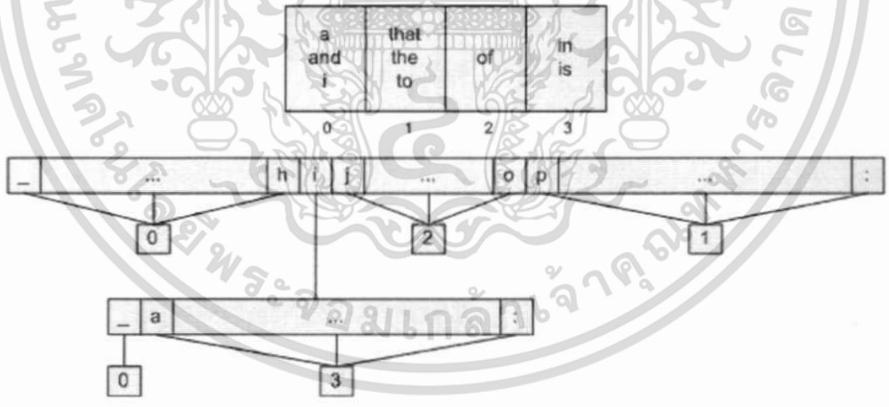
รูปที่ 3.8 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'in', 'that' ลงในบัคเก็ต

4. เพิ่ม 'is' ลงใน [0] เกิดการล้นบัคเก็ต จึงทำการแบ่งบัคเก็ต [c' = 'in', l = 0, l' = 0] ซึ่งจะทำการสร้างบัคเก็ต 2 และให้ไดเรกทอรีตั้งแต่หลัง 'i' เป็นต้นไปที่โยงไปยังบัคเก็ต 0 โยงไปยังบัคเก็ต 2 แทน



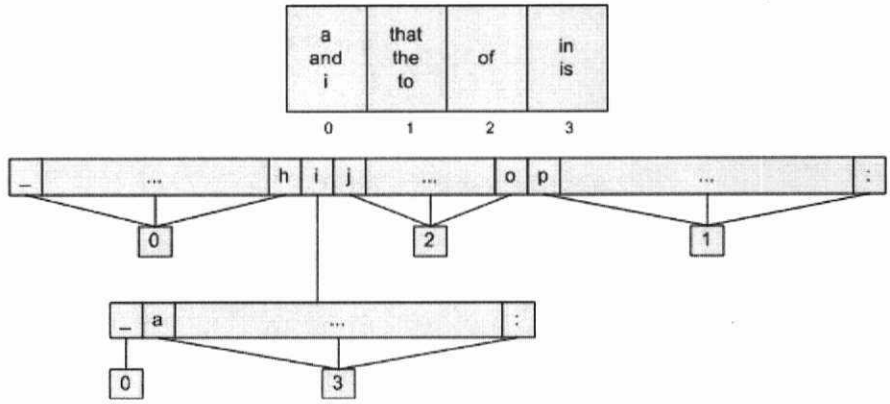
รูปที่ 3.9 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'is' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

5. เพิ่ม 'i' ลงใน [0] และทำการแบ่งบัคเก็ต [c' = 'i', l = 0, l' = 1] ซึ่งจะทำสร้างบัคเก็ต 3 และให้ไดเรกทอรีตั้งแต่หลัง 'i' เป็นต้นไปที่โยงไปยังบัคเก็ต 0 โยงไปยังบัคเก็ต 3 แทนซึ่งในที่นี้ไม่มี ส่วนไดเรกทอรี 'i' ให้ทำการโยงไปยังโหนดที่สร้างขึ้นใหม่ N' และให้ไดเรกทอรีในโหนด N' ที่มาก่อนหรือตรงกับ '_' โยงไปยังบัคเก็ต 0 ส่วนที่เหลือโยงไปยังบัคเก็ต 3 เช่นกัน



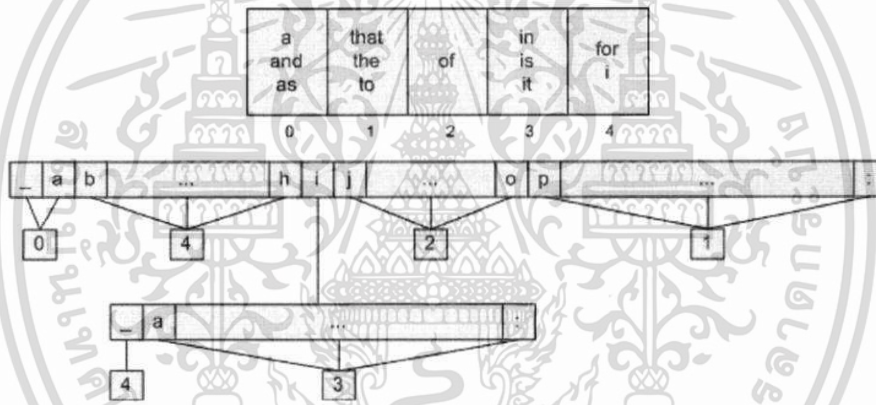
รูปที่ 3.10 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'i' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

6. เพิ่ม 'it' ลงใน [3] และ 'for' ลงใน [0]



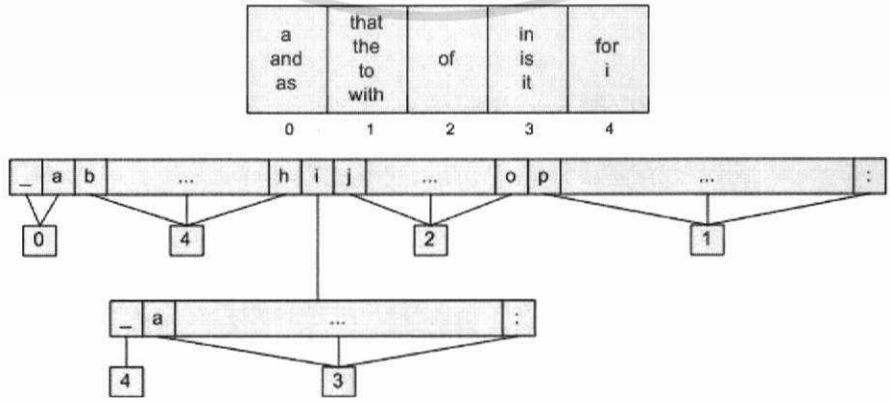
รูปที่ 3.11 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'it', 'for' ลงในบัคเก็ต

7. เพิ่ม 'as' ลงใน 0 และทำการแบ่งบัคเก็ต [c' = 'as', l = 0, l' = 0] ซึ่งจะเห็นได้ว่าหลังระดับ 0 ลงไป ไดเรกทอรีทั้งหมดที่โยงไปยังบัคเก็ต 0 จะถูกเปลี่ยนการโยงเป็นบัคเก็ต 4 ด้วย



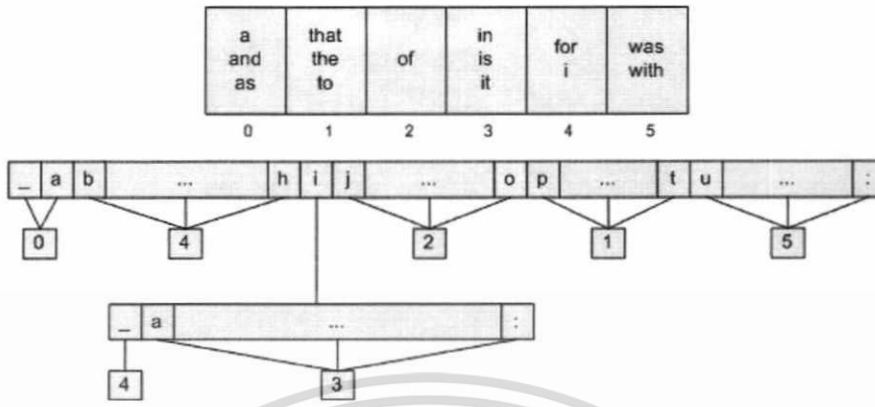
รูปที่ 3.12 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'as' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

8. เพิ่ม 'with' ลงใน 1



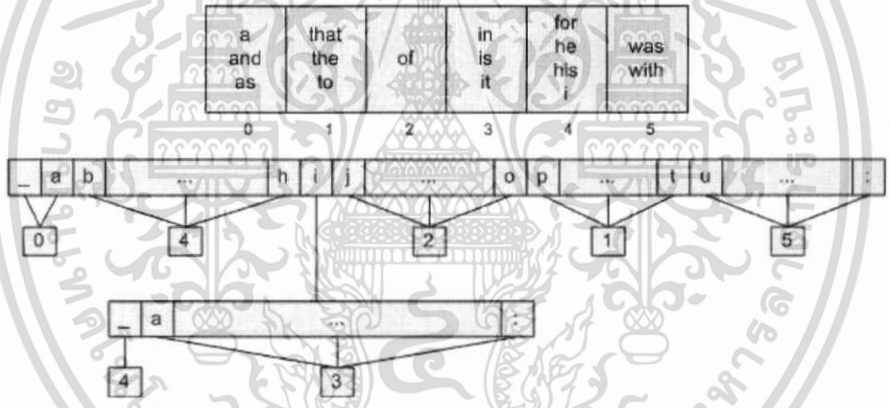
รูปที่ 3.13 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'with' ลงในบัคเก็ต

9. เพิ่ม 'was' ลงใน [1] และทำการแบ่งบัคเก็ต [c' = 'to', $l = 0$, $l' = 0$]



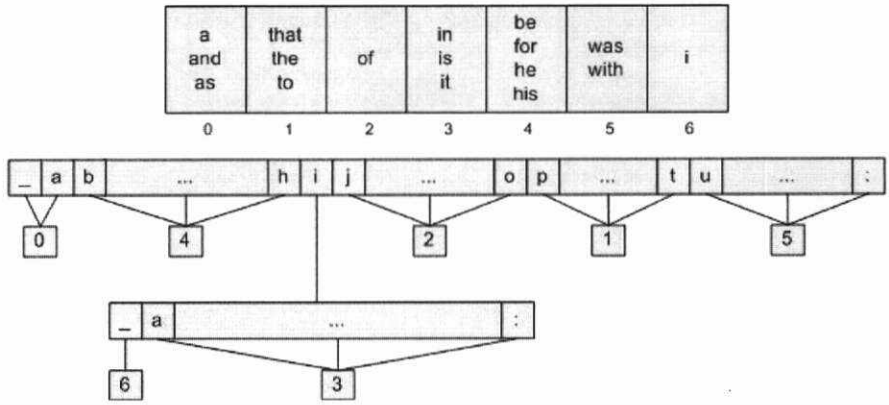
รูปที่ 3.14 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'was' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

10. เพิ่ม 'his', 'he' ลงใน [4]



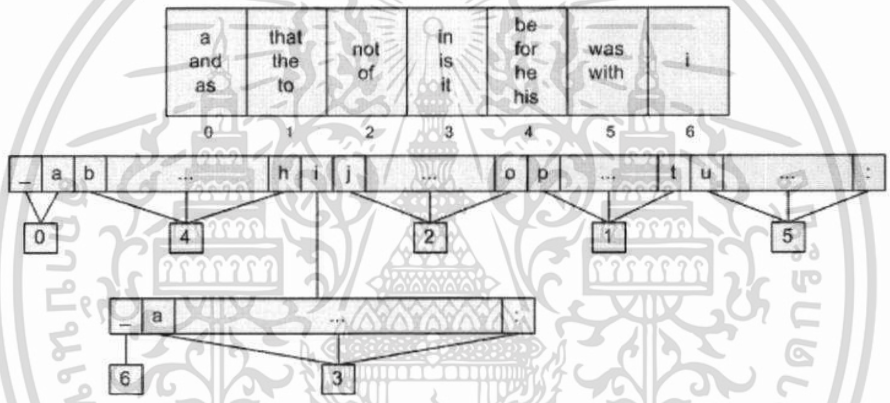
รูปที่ 3.15 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'his', 'he' ลงในบัคเก็ต

11. เพิ่ม 'be' ลงใน [4] และทำการแบ่งบัคเก็ต [c' = 'he', $l = 0$, $l' = 0$]



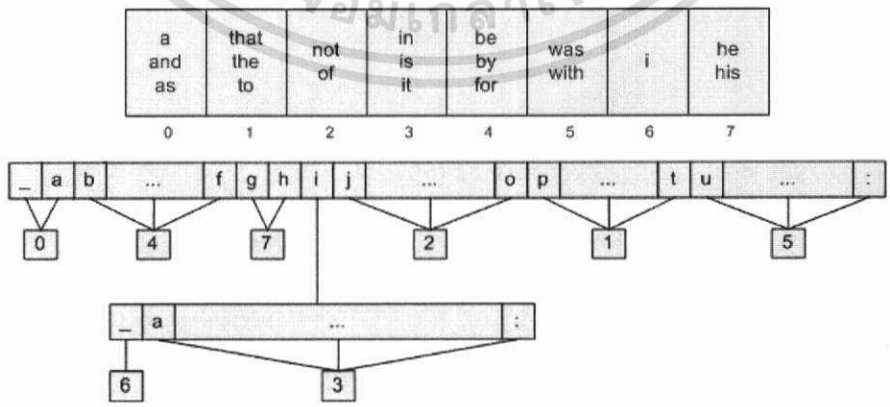
รูปที่ 3.16 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'be' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

12. เพิ่ม 'not' ลงใน [2]



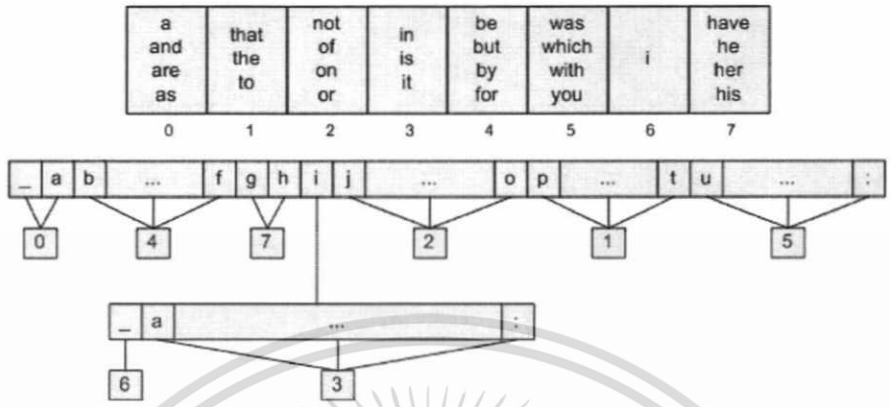
รูปที่ 3.17 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'not' ลงในบัคเก็ต

13. เพิ่ม 'by' ลงใน [4] และทำการแบ่งบัคเก็ต [c' = 'for', $l = 0$, $l' = 0$]



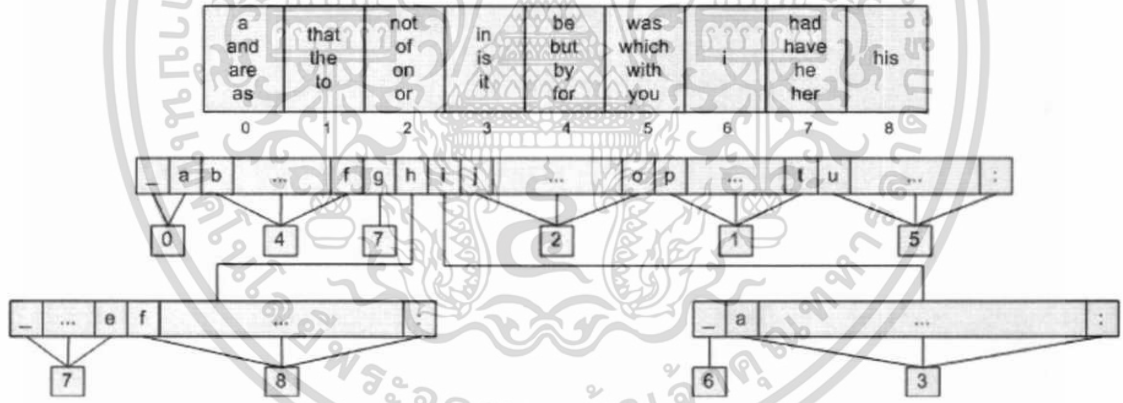
รูปที่ 3.18 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'by' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

14. เพิ่ม 'but' ลงใน [4], 'have', 'her' ลงใน [7], 'you', 'which' ลงใน [5], 'are' ลงใน [0] และ 'on', 'or' ลงใน [2]



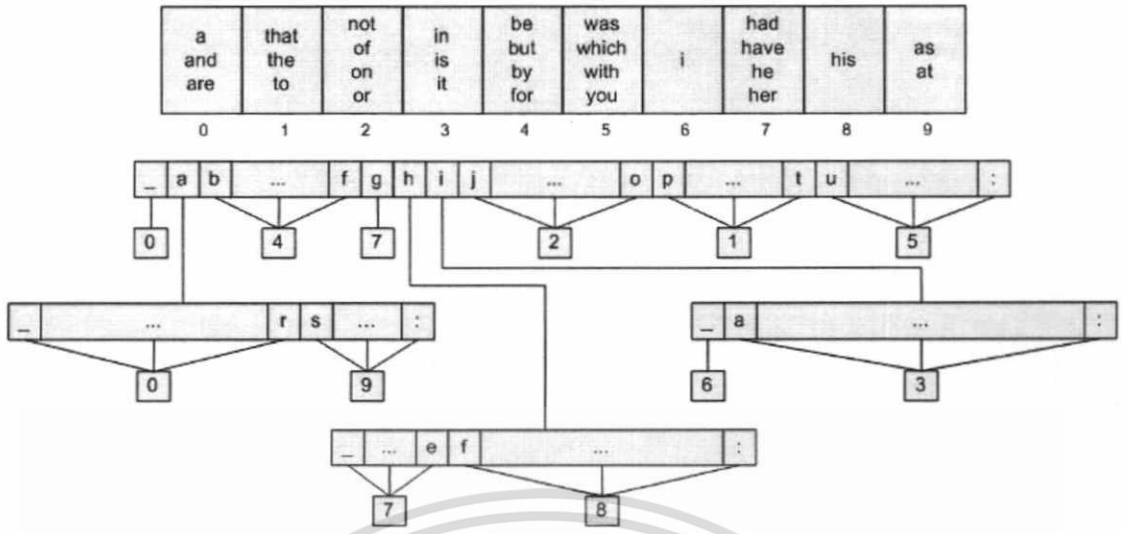
รูปที่ 3.19 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'but', 'have', 'her', 'you', 'which', 'are', 'on', 'or' ลงในบัคเก็ต

15. เพิ่ม 'had' ลงใน [7] และทำการแบ่งบัคเก็ต [c' = 'he', $l = 0$, $l' = 1$]



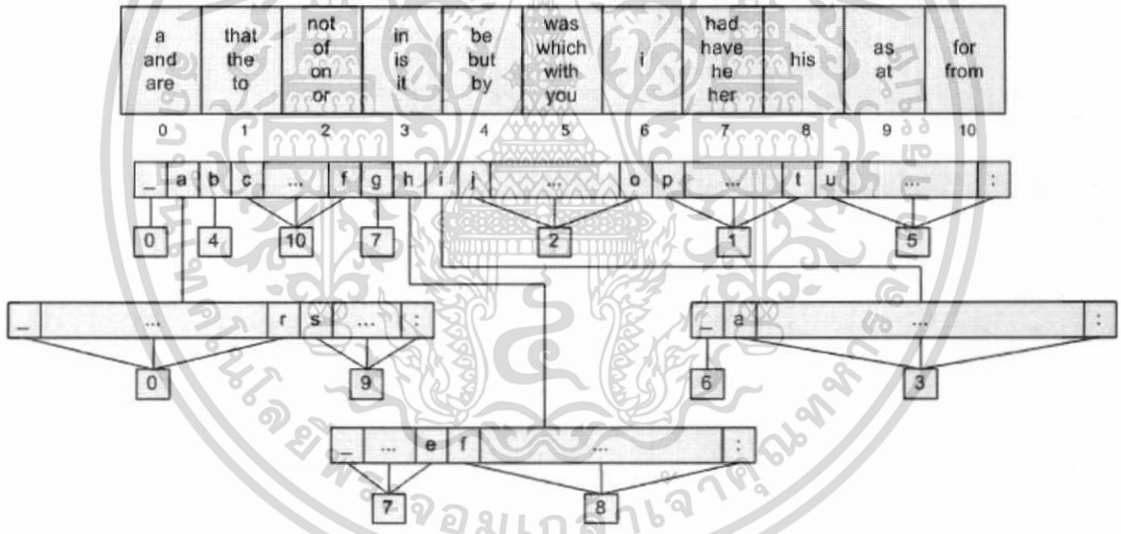
รูปที่ 3.20 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'had' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

16. เพิ่ม 'at' ลงใน [0] และทำการแบ่งบัคเก็ต [c' = 'are', $l = 0$, $l' = 1$]



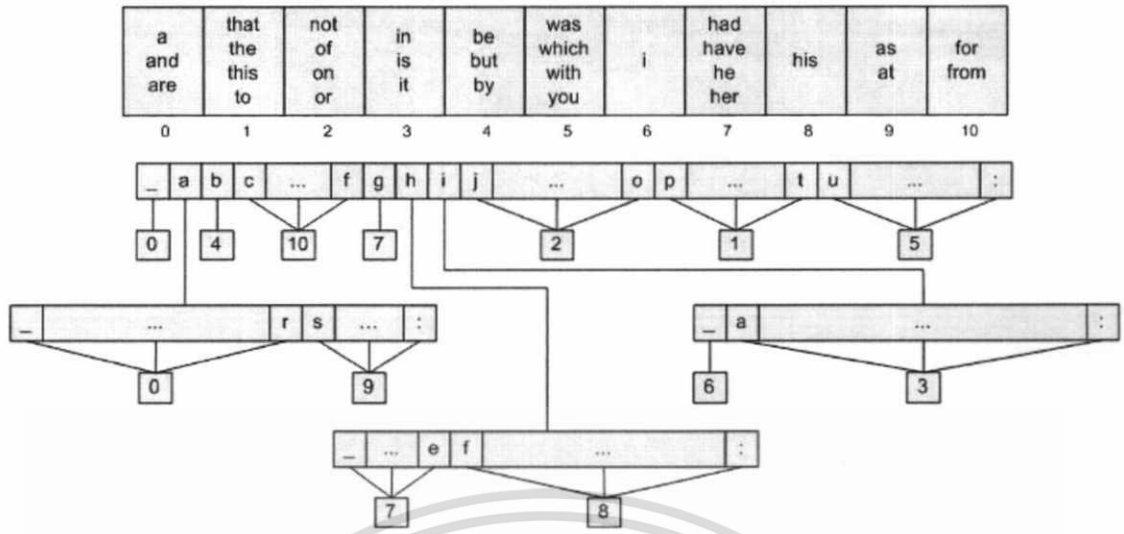
รูปที่ 3.21 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'at' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

17. เพิ่ม 'from' ลงใน [4] และทำการแบ่งบัคเก็ต [c' = 'by', $l = 0$, $l' = 0$]



รูปที่ 3.22 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'from' ลงในบัคเก็ตและทำการแบ่งบัคเก็ต

18. เพิ่ม 'this' ลงใน [1]



รูปที่ 3.23 ผลลัพธ์ของโครงสร้างหลังจากเพิ่ม 'this' ลงในบัคเก็ตและเป็นอันสิ้นสุดการสร้าง

บทที่ 4
วิเคราะห์ผล

เนื่องจากจุดประสงค์ของการทำงานวิจัยชิ้นนี้คือเพื่อทำการแก้ปัญหา degeneracy และเพิ่มความเร็วในการค้นหาที่อยู่บัคเก็ต ดังนั้นในที่นี้จึงทำการอภิปรายวิเคราะห์การแก้ปัญหา degeneracy ของวิธีแฮชชิงทรัยแบบขยายได้ และทำการเปรียบเทียบวิเคราะห์เวลาที่ใช้ในการทำงานระหว่างวิธีแฮชชิงทรัยกับวิธีแฮชชิงทรัยแบบได้เฉพาะการค้นหาที่อยู่บัคเก็ตเท่านั้น โดยคำนวณจาก pseudo code

4.1 การแก้ปัญหา degeneracy

สำหรับวิธีแฮชชิงทรัยแบบขยายได้นั้น เนื่องจากการหลักที่ใช้ในการท่องเข้าไปใน extendible trie คือการเข้าไปในไดเรกทอรีที่มีชื่อตรงกับดิจิตของคีย์ ณ ตำแหน่งเดียวกับระดับของโหนดที่เยี่ยมอยู่ นั่นหมายความว่าความสูงของโครงสร้างข้อมูลนี้จะไม่เกิน m ระดับ และลำดับคีย์ที่ใช้ในการเพิ่มซึ่งไม่ว่าจะเป็นแบบสุ่มหรือเรียงลำดับนั้นไม่มีผลกับเวลาที่ใช้ในการสืบค้นด้วย ดังนั้นวิธีนี้จึงสามารถแก้ปัญหา degeneracy ได้

4.2 เวลาที่ใช้การค้นหาที่อยู่บัคเก็ต

สำหรับ pseudo code ที่นำมาใช้ในการวิเคราะห์การทำงานของแต่ละวิธีนั้นเป็นดังนี้

แฮชชิงทรัย

ใช้ Algorithm 1 ในการวิเคราะห์ผล โดยแต่ละบรรทัดใช้เวลาในการทำงานดังนี้

บรรทัดที่ 1 ทำการกำหนดค่าเริ่มต้น ใช้เวลาทำงาน $O(1)$

บรรทัดที่ 2 ทำการเช็คเงื่อนไขว่าโหนดที่เยี่ยมอยู่เป็นโหนดภายในหรือไม่ ถ้าเป็นให้ทำงานอยู่ในลูปตั้งแต่บรรทัดที่ 3 – 14 จนกระทั่งโหนดที่เยี่ยมอยู่ไม่ใช่โหนดภายใน ใช้เวลาทำงาน $O(km)$ เนื่องจากโครงสร้างของ trie ที่เกิดขึ้นนั้นจะขึ้นอยู่กับผลจากการแบ่งบัคเก็ต โดยในการแบ่งแต่ละครั้งจะมีการขยายโหนดภายในไม่เกิน m ซึ่งถ้าเกิดการแบ่ง k ครั้ง จะได้ว่ามีจำนวนโหนดภายในทั้งหมดไม่เกิน km

บรรทัดที่ 3 ทำการเช็คเงื่อนไขว่าตำแหน่งที่ j ตรงกับหมายเลขดิจิตหรือไม่ ใช้เวลาทำงาน $O(1)$ ถ้าตรงกันให้ทำงานในบรรทัดที่ 4 - 9

บรรทัดที่ 4 ทำการเช็คเงื่อนไขว่าอักขระของคีย์ ณ ตำแหน่งที่ j มาก่อนค่าดิจิตหรือไม่ ใช้เวลาทำงาน $O(1)$ ถ้ามาก่อนให้ทำงานในบรรทัดที่ 5 – 7

บรรทัดที่ 5 ทำการท่องไปทางซ้าย และกำหนดค่าทางเดินตรรกะใหม่ ใช้เวลาทำงาน $O(1)$

บรรทัดที่ 6 ทำการเช็คเงื่อนไขว่าอักขระของคีย์ ณ ตำแหน่งที่ j ตรงกับค่าดิจิตหรือไม่ ถ้าตรงกัน ให้ทำเพิ่มค่า j ใช้เวลาทำงาน $O(1) + O(1) = O(1)$

บรรทัดที่ 7 เป็นการสิ้นสุด if ของบรรทัดที่ 6

ดังนั้นในส่วนเงื่อนไขตั้งแต่บรรทัดที่ 4 – 7 ใช้เวลาทำงานทั้งหมด $O(1) + O(1) + O(1) = O(1)$

บรรทัดที่ 8 เป็นเงื่อนไขที่เหลือ ทำการท่องไปทางขวา ใช้เวลาทำงาน $O(1)$

บรรทัดที่ 9 เป็นการสิ้นสุด if-else ตั้งแต่บรรทัดที่ 4 – 8

ดังนั้นในส่วนเงื่อนไขตั้งแต่บรรทัดที่ 3 – 9 ใช้เวลาทำงานทั้งหมด $O(1) + \text{MAX}\{O(1), O(1)\} = O(1)$

บรรทัดที่ 10 ทำการเช็คเงื่อนไขว่าตำแหน่งที่ j น้อยกว่าหมายเลขดิจิตหรือไม่ ใช้เวลาทำงาน $O(1)$ ถ้าน้อยกว่าให้ทำงานในบรรทัดที่ 11

บรรทัดที่ 11 ทำการท่องไปทางซ้าย และกำหนดค่าทางเดินตรรกะใหม่ ใช้เวลาทำงาน $O(1)$

ดังนั้นในส่วนเงื่อนไขตั้งแต่บรรทัดที่ 10 – 11 ใช้เวลาทำงานทั้งหมด $O(1) + O(1) = O(1)$

บรรทัดที่ 12 เป็นเงื่อนไขที่เหลือ ทำการท่องไปทางขวา ใช้เวลาทำงาน $O(1)$

บรรทัดที่ 13 เป็นการสิ้นสุด if-else ตั้งแต่บรรทัดที่ 3 – 12

ดังนั้นในส่วนเงื่อนไขตั้งแต่บรรทัดที่ 3 – 13 ใช้เวลาทำงานทั้งหมด $\text{MAX}\{O(1), O(1), O(1)\} = O(1)$

บรรทัดที่ 14 เป็นการสิ้นสุด while ตั้งแต่บรรทัดที่ 2 – 13

ดังนั้นในส่วนลูปตั้งแต่บรรทัดที่ 2 – 14 ใช้เวลาทำงานทั้งหมด $O(km)O(1) = O(km)$

บรรทัดที่ 15 ทำการคืนผลลัพธ์ ใช้เวลาทำงาน $O(1)$

เพราะฉะนั้นจึงสามารถสรุปได้ว่าใช้เวลาในการทำงานทั้งหมดนั้น มีค่าเท่ากับ

$= O(1) + O(km) + O(1)$

$= O(km)$

แฮชชิงทรัยแบบขยายได้

ใช้ Algorithm 3 ในการวิเคราะห์ผล โดยแต่ละบรรทัดใช้เวลาในการทำงานดังนี้

บรรทัดที่ 1 ทำการกำหนดค่าเริ่มต้น ใช้เวลาทำงาน $O(1)$

บรรทัดที่ 2 ทำการเช็คเงื่อนไขว่าโหนดที่เยี่ยมอยู่เป็นโหนดภายในหรือไม่ ถ้าเป็นให้ทำงานอยู่ในลูปตั้งแต่บรรทัดที่ 3 – 4 จนกระทั่งโหนดที่เยี่ยมอยู่ไม่ใช่โหนดภายใน ใช้เวลาทำงาน $O(m)$ เนื่องจากใช้ดิจิตของคีย์ ณ ตำแหน่งเดียวกับระดับของโหนดที่เยี่ยมอยู่ เป็นเกณฑ์ในการท่องเข้าไป และคีย์นั้นมีความยาวไม่เกิน m

บรรทัดที่ 3 ทำการท่องไปใน extendible trie ใช้เวลาทำงาน $O(1)$

บรรทัดที่ 4 เป็นการสิ้นสุด while ตั้งแต่บรรทัดที่ 2 – 3

ดังนั้นในส่วนลูปตั้งแต่บรรทัดที่ 2 – 4 ใช้เวลาทำงานทั้งหมด $O(m)O(1) = O(m)$

บรรทัดที่ 5 ทำการคืนผลลัพธ์ ใช้เวลาทำงาน $O(1)$

เพราะฉะนั้นเวลาที่ใช้ในการทำงานทั้งหมดจึงเท่ากับ

$= O(1) + O(m) + O(1)$

$= O(m)$

บทที่ 5
สรุปและข้อเสนอแนะ

แฮชชิงทรัย (Trie hashing (TH)) เป็นวิธีการเข้าถึงวิธีหนึ่งสำหรับการเก็บและเข้าถึงระเบียนของแฟ้มข้อมูลยืดหยุ่น โดยใช้ trie เป็นตัวคำนวณหาที่อยู่บัคเก็ต วิธีการเข้าถึงวิธีนี้จัดได้ว่าเป็นวิธีที่มีประสิทธิภาพมากวิธีหนึ่ง เนื่องจากสามารถทราบผลค้นคืนได้อย่างรวดเร็ว โดยทำการเข้าถึงดิสค์เพียง 1 ครั้งเท่านั้นถ้าบรรจุ trie อยู่ในหน่วยความจำได้เพียงพอ และยังสามารถทำการประมวลผลแบบ range query ได้อีกด้วย สำหรับลำดับคีย์แบบสุ่มนั้นให้ค่าโหลดแฟกเตอร์ 70% และแบบเรียงลำดับนั้นให้ 50% แต่ทว่าวิธีการเข้าถึงดังกล่าวยังมีข้อเสียอยู่หลักๆ 2 ข้อด้วยกันอันได้แก่ 1) ถ้าลำดับคีย์ที่เพิ่มนั้นเรียงลำดับ จะใช้เวลาในการสืบค้นโหนดใบที่อยู่ระดับล่างสุดนาน และเวลาในการสืบค้นขึ้นอยู่กับลำดับคีย์ด้วย กล่าวคือคีย์ที่อยู่ในลำดับต้นๆจะใช้เวลาสืบค้นเร็ว ส่วนคีย์ที่อยู่ในลำดับปลายๆจะใช้เวลาในการสืบค้นนาน เนื่องจากลักษณะการขยายโหนดนั้นเอนเอียงไปทางด้านใดด้านหนึ่งหรือเรียกว่า degeneracy 2) ต้องใช้ต้นทุนในการสร้าง trie ใหม่สูง หากเกิดการขัดข้องของระบบเกิดขึ้น

ดังนั้นจึงได้คิดค้นวิธีการแก้ปัญหา degeneracy ของวิธีแฮชชิงทรัยขึ้นมา และเพิ่มความเร็วในการสืบค้นหาที่อยู่บัคเก็ตด้วย โดยการปรับโครงสร้างข้อมูลสำหรับการสืบค้นที่อยู่บัคเก็ตใหม่ ซึ่งได้นำเอาโครงสร้างและหลักการเข้าถึงของแฮชชิงแบบขยายได้กับหลักการแบ่งบัคเก็ตของแฮชชิงทรัยมารวมกัน เกิดเป็นโครงสร้างข้อมูลใหม่เรียกว่า extendible trie และเรียกวิธีใหม่นี้ว่า "แฮชชิงทรัยแบบขยายได้ (Extendible trie hashing)" สำหรับวิธีแฮชชิงทรัยแบบขยายได้นั้นจะใช้ดิจิตของคีย์ ณ ตำแหน่งเดียวกับระดับของโหนดที่เยี่ยมอยู่เป็นเกณฑ์ในการท่องเข้าไปเพื่อสืบค้นหาที่อยู่บัคเก็ต โดยใช้เวลาในการทำงาน $O(m)$ ซึ่งเร็วกว่าวิธีแฮชชิงทรัยที่ใช้เวลาในการทำงาน $O(km)$ โดยที่ k แทนจำนวนครั้งของการแบ่งบัคเก็ต และ m แทนความยาวของคีย์ ด้วยหลักการท่องดังกล่าวนี้ส่งผลให้สามารถแก้ปัญหา degeneracy ได้อีกด้วย

ถึงแม้ว่าวิธีแฮชชิงทรัยแบบขยายได้จะช่วยแก้ปัญหา degeneracy และเพิ่มความเร็วในการสืบค้นหาที่อยู่บัคเก็ต แต่ยังมีข้อเสียคือต้องใช้พื้นที่ในหน่วยความจำอย่างมากถึง $O(|\Sigma|^m)$ จึงเหมาะกับการคำนวณกับระเบียนที่มีขนาดของ Σ และความยาวของคีย์น้อยๆ

เอกสารอ้างอิง

[1] M. Aridj and D. E. Zegour. TH*: Scalable distributed trie hashing. IJCSI International Journal of Computer Science Issues, 7(6):109–115, November 2010.

[2] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. ACM Trans. Database Syst., 4:315–344, September 1979.

[3] W. Litwin. Trie hashing. In Proceedings of the 1981 ACM SIGMOD international conference on Management of data, SIGMOD '81, pages 19–29, New York, NY, USA, 1981. ACM.

[4] W. Litwin, D. Zegour, and G. Levy. Multilevel trie hashing. In Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology, EDBT '88, pages 309–335, London, UK, 1988. Springer-Verlag.

[5] W. A. Litwin, N. Roussopoulos, G. Levy, and W. Hong. Trie hashing with controlled load. 17(7):678–691, 1991.

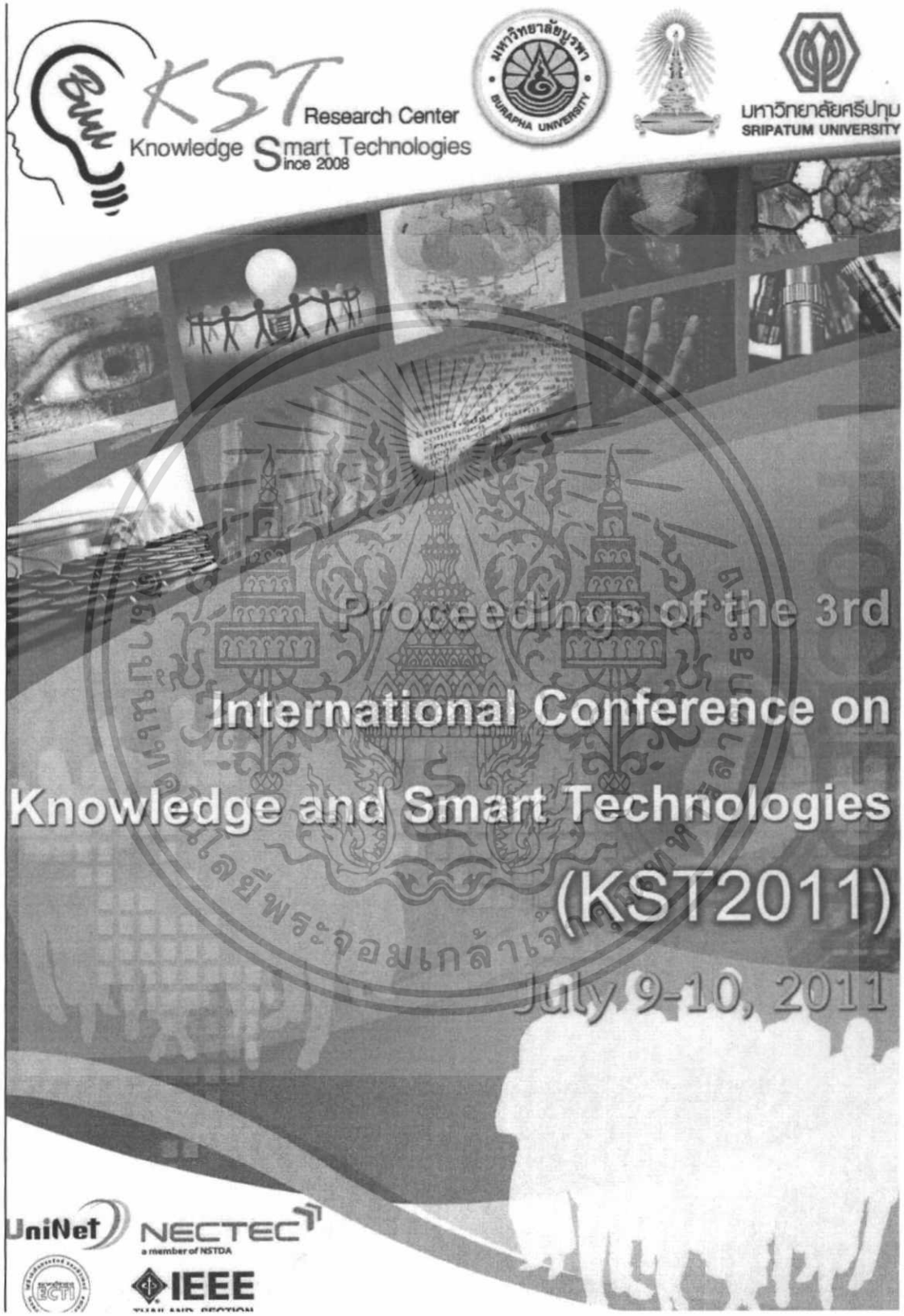
[6] E. J. Otoo and S. Effah. Red-black balanced trie hashing, 1995.

[7] L. Torenvliet and P. v. E. Boas. The reconstruction and optimization of trie hashing functions. In Proceedings of the 9th International Conference on Very Large Data Bases, pages 142–156, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[8] D. E. Zegour. Scalable distributed compact trie hashing (CTH*). Information & Software Technology, 46:923–935, 2004.

ภาคผนวก ก

งานวิจัยที่ตีพิมพ์



แฮชชิงทรัยแบบขยายได้

Extendible Trie Hashing

Manit Kwanyun[1] and Veera Boonjing[1, 2]

[1] Software Systems Engineering Laboratory, Department of Computer Science

Faculty of Science King Mongkut's Institute of Technology Ladkrabang, Bankok, Thailand

[2]National Centre of Excellence in Mathematics, PERDO, Bangkok 10400, Thailand

{amashinji, boonjv}@gmail.com

บทคัดย่อ

งานวิจัยนี้เสนอวิธีการใหม่ในการเข้าถึงระเบียนข้อมูลของแฟ้มยืดหยุ่นที่เรียกว่า แฮชชิงทรัยแบบขยายได้ โดยวิธีการใหม่นี้ได้รวมข้อดีของวิธีแฮชชิงแบบขยายได้เข้ากับวิธีแฮชชิงทรัย เพื่อแก้ปัญหาการไม่สมดุลของทรัยทำให้สามารถเข้าถึงบัคเก็ตได้โดยใช้เวลา O(m) โดยที่ m คือขนาดของคีย์

Abstract

The research proposes a new access method for dynamic files called extendible trie hashing. The method combines an extendible hashing method with a trie hashing as a solution to degeneracy problems. It is able to access a file bucket in O(m) times, where m is a key size.

คำสำคัญ: Access method, Data structure, Hashing, Trie hasing, Extendible hashing

1. บทนำ

แฮชชิงทรัย (Trie hashing (TH)) [3] เป็นวิธีเข้าถึงวิธีหนึ่งสำหรับการเก็บและการเข้าถึงระเบียนของแฟ้มข้อมูลยืดหยุ่น โดยระเบียนถูกเก็บอยู่ในบัคเก็ตซึ่งบรรจุไว้อย่างมาก b ระเบียนบนดิสค์ การค้นคืนระเบียนที่ระบุโดยคีย์ๆหนึ่งนั้นจะประกอบไปด้วย 2 ขั้นตอนได้แก่ 1) คำนวณหาที่อยู่บัคเก็ตโดยใช้ trie 2) เข้าถึงดิสค์ดึงบัคเก็ตที่มีที่อยู่ดังกล่าวลงในหน่วยความจำหลัก จากนั้นจึงทำการสืบค้นหาระเบียนที่ระบุโดยคีย์ดังกล่าว ข้อดีของวิธีนี้คือ ทราบผลของการค้นคืนได้อย่างรวดเร็ว โดยทำการเข้าถึงดิสค์เพียง 1 ครั้ง ถ้า trie นั้นอยู่ในหน่วยความจำหลักได้เพียงพอ แต่ถ้าแฟ้มข้อมูลมีขนาดใหญ่มาก trie จะไม่สามารถบรรจุไหว จึงต้องใช้วิธีแบ่ง trie ออกเป็น subtries และแต่ละ subtrie จะ ถูกเก็บไว้ในเพจ (pages) บนดิสค์ วิธีนี้เรียกว่า แฮช-ชิงทรัยหลายระดับ (Multilevel trie hashing (MLTH)) [4] ส่งผลให้ต้องเข้าถึงดิสค์มากขึ้น แต่เพียง 2 ครั้งก็เพียงพอสำหรับแฟ้มข้อมูลขนาดเป็นจิ๊กกะไบต์แล้ว และนอกจากนี้ TH ยังสามารถทำการประมวลผลแบบ range query ได้ซึ่งได้ทำการอธิบายขั้นตอนไว้ใน [4] เนื่องจากมีคุณสมบัติ weak order preserving กล่าวคือ สำหรับสองบัคเก็ตใดๆ B_i, B_j ซึ่ง $K_i \in B_i$ และ $K_j \in B_j$ จะได้ว่า $K_i < K_j$ สำหรับ $\forall K_i \in B_i$ และ $\forall K_j \in B_j$ สำหรับลำดับคีย์แบบสุ่มนั้นให้ค่าโหลดแฟกเตอร์ 70% และแบบเรียงลำดับนั้นให้ 50%

จากที่กล่าวมาข้างต้นนั้นถือได้ว่าเป็นวิธีเข้าถึงที่มีประสิทธิ-ภาพมากวิธีหนึ่ง แต่ถึงกระนั้นวิธีนี้ก็ยังมีข้อเสียหลักๆอยู่อันได้แก่ 1) เมื่อลำดับคีย์ที่เพิ่มนั้นเรียงลำดับ ทำให้การขยายโหนดนั้นเอนไปทางด้านใดด้านหนึ่งหรือเรียกว่า degeneracy ส่งผลให้ใช้เวลาในการสืบค้นโหนดใบที่อยู่ระดับล่างสุดนาน 2) หลัง

จากเกิดระบบขัดข้อง จะทำให้ trie ที่ถูกสร้างไว้สูญหาย จึงต้องมีการสร้างใหม่ซึ่งใช้ต้นทุนการสร้างสูง

จากข้อเสียดังกล่าวเหล่านี้ จึงได้มีผู้เสนอวิธีแก้ไขต่างๆไว้มากมาย ไม่ว่าจะเป็นปัญหา degeneracy [6], การปรับปรุงค่าโหลดแฟกเตอร์ [5] หรือการคงอยู่ (persistence) [7] และนอกจากนี้มีผู้พัฒนาดัดแปลงให้เป็นโครงสร้างข้อมูลแบบกระจาย [8] และ [1] อีกด้วย สำหรับในที่นี่ได้ทำการเสนอวิธีใหม่โดยเป็นโครงสร้างข้อมูลแบบปรกติเพื่อแก้ปัญหา degeneracy และเพิ่มความเร็วในการสืบค้นที่อยู่บัคเก็ต ซึ่งนำเอาโครงสร้างและหลักการเข้าถึงของแฮชชิงแบบขยายได้ [2] กับหลักการแบ่งการบัคเก็ตของแฮชชิงทรัยมารวมกัน ใช้เวลาในการสืบค้น $O(m)$ โดยที่ m แทน ความยาวของคีย์

เนื้อหาได้ถูกแบ่งออกเป็นหัวข้อต่างๆดังนี้ หัวข้อที่ 2 เป็นส่วนของรายละเอียดพื้นฐานของแฮชชิงทรัย หัวข้อที่ 3 มาทำความรู้จักกับแฮชชิงทรัยแบบขยายได้ หัวข้อที่ 4 วิเคราะห์ประสิทธิภาพเมื่อเทียบกับแฮชชิงทรัย และหัวข้อที่ 5 เป็นการสรุปเนื้อหา

2. พื้นฐานแฮชชิงทรัย (Trie hashing)

2.1. คำศัพท์และสัญลักษณ์

- Σ - เซตซึ่งประกอบด้วยสมาชิกเป็นตัวอักขระ (alphabet) ที่สามารถเรียงลำดับได้โดยมี '_' (space) เป็นสมาชิกต่ำสุด และ ':' เป็นสมาชิกสูงสุด ขนาดของ Σ แทนด้วย $|\Sigma|$
- ดิจิต (digit) - สมาชิกแต่ละตัวที่อยู่ใน Σ
- สายอักขระ (string) - สมาชิกแต่ละตัวที่อยู่ใน Σ^*
- คีย์สเปซ (key space) - สับเซตใน Σ^* ซึ่งสมาชิกแต่ละตัวนั้นมีความยาว m และสมาชิกตัวนั้นสามารถเป็นสายอักขระที่มีความยาวต่ำกว่า m ได้ ซึ่งจะใช้ '_' ในการต่อท้ายจนกระทั่งมีความยาว m
- คีย์ (key) - สมาชิกแต่ละตัวที่อยู่ใน key space ซึ่งจะถูกนำมาใช้ในการสร้างแฮช
- บัคเก็ต (bucket) - เซลล์ที่ถูกแบ่งจากดิสก์ (disk) ทำหน้าที่เก็บระเบียนซึ่งสามารถเก็บได้สูงสุด b ระเบียน และค่า b ดังกล่าวนั้นเรียกว่า "ความจุบัคเก็ต (bucket capacity)" และแต่ละบัคเก็ตจะมีหมายเลข 0, 1, 2, ... กำกับอยู่ด้วยซึ่ง เรียกว่า "ที่อยู่บัคเก็ต (bucket address)"
- คีย์กลาง (middle key) - คีย์ที่อยู่ในตำแหน่ง $\lceil (b+1)/2 \rceil$ ของบัคเก็ตที่ล้น
- โหลดแฟกเตอร์ (load factor) : ค่าที่คำนวณจาก จำนวนระเบียนที่ใช้เก็บ / ขนาดของบัคเก็ตทั้งหมด ค่ายิ่งมากแสดงว่ายิ่งเป็นแฮชที่ดี
- $(c)_l$ - คำนำหน้า l+1 ตัวของสายอักขระ c (c เป็นค่าว่างถ้า $l < 0$)

2.2. โครงสร้าง

อยู่ในรูปของไบนารี่ทรี โดยโหนดภายในเก็บอยู่ในรูปคู่ของค่าดิจิต (digit value) และหมายเลขดิจิต (digit number) ซึ่งแทนด้วย (d, i) ส่วนโหนดภายนอกหรือโหนดใบอาจเก็บเป็นได้ทั้งที่อยู่บัคเก็ตหรือ *nil* ซึ่งสื่อว่าไม่มีที่อยู่บัคเก็ตที่สอดคล้องกับโหนดนี้ จาก Figure 1 [5] ตัวอย่างโครงสร้างนั้นได้แสดงไว้ ณ Figure 1(c) ซึ่งรวมทางเดินตรรกะไปยังโหนดต่างๆด้วย (ค่าที่อยู่บนเส้นเชื่อมระหว่างโหนด n ใดๆกับแม่ของ n) โดยแต่ละค่าถูกกำหนดไว้ดังนี้

สายอักขระซึ่งสอดคล้องกับโหนด n ใดๆใน trie นั้นเรียกว่า "ทางเดินตรรกะไปยัง n (logical path to n)" แทนด้วย C_n ถูกนิยามไว้ดังนี้

- ถ้า C_n เป็นรากแล้ว C_n = ':'
- อื่นๆ, ให้ $p = (d, i)$ เป็นแม่ของ n ถ้า n เป็นลูกทางขวาแล้ว $C_n = C_p$ นอกนั้น $C_n = (C_p)_{i-1}d$

สำหรับ Standard representation นั้นจะอยู่ในรูปของ linked list โดยสมาชิกแต่ละตัวเรียกว่า "เซลล์ (cell)" ดัง Figure 1(d) ซึ่งประกอบไปด้วย 4 เขตข้อมูลด้วยกันคือ DV (digit value), DN (digit number), LP (left pointer) และ RP (right pointer) DV และ DN เก็บค่าดิจิตและตำแหน่งของ DV ตามลำดับ ซึ่ง

เป็นค่าที่เก็บอยู่ในโหนดภายใน ส่วน LP และ RP อาจเก็บเป็นค่าที่อยู่เซลล์หรือที่อยู่บัคเก็ต ถ้าค่าเป็นลบแสดงว่าเป็นค่าที่อยู่เซลล์ แต่ถ้าค่าเป็นบวกแสดงว่าเป็นค่าที่อยู่บัคเก็ต มีตัวอย่างประกอบอยู่ใน Figure 1(e)

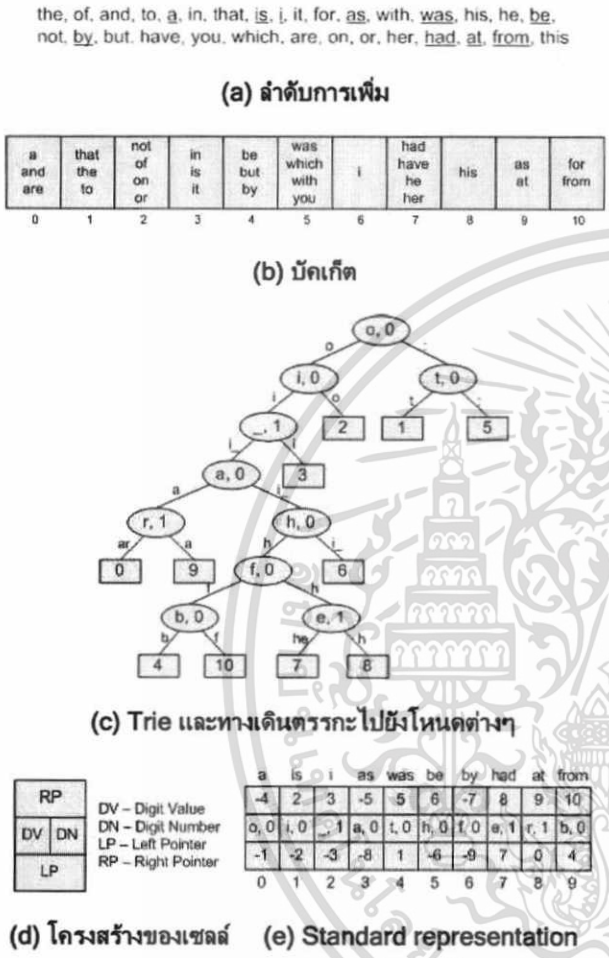


Figure 1. ตัวอย่างแฟ้มข้อมูล

2.3. ค้นหาคีย์

ทำการค้นหาโดยการท่องไปใน trie ดังอัลกอริทึมข้างล่างนี้ ซึ่งให้ผลลัพธ์กลับมาเป็นที่อยู่บัคเก็ตและทางเดินตรรกะไปยังโหนดที่อยู่บัคเก็ตดังกล่าว

Algorithm 1 TH Key Search: ให้ $c = c_0c_1c_2...$ แทนคีย์ที่ค้นหา, r แทนราก, $n = (d, i)$ แทนโหนดที่เยี่ยมอยู่, $L(n)$ และ $R(n)$ แทนลูกทางซ้ายและลูกทางขวาของ n ตามลำดับ และ C แทนทางเดินตรรกะ

Algorithm 1 TH Key Search

```
1:  n ← r; C =':'; j = 0;
2:  while n is an internal node do
3:    if j = i then
4:      if c_j ≤ d then
5:        n ← L(n); C ← (C)_{i-1}d;
6:        if c_j = d then j ← j + 1;
7:        end if
8:      else n ← R(n);
9:      end if
10:   else if j < i then
11:     n ← L(n); C ← (C)_{i-1}d;
12:   else n ← R(n);
13:   end if
14: end while
15: return n, C
```

จากอัลกอริทึมดังกล่าวจะเห็นได้ว่า ในกรณีที่ $c_j \leq d$ เมื่อ $j = i$ หรือกรณีที่ $j < i$ นั้นจะท่องไปทางซ้าย ส่วนกรณีที่เหลือท่องไปทางขวา

2.4. การแบ่งบัคเก็ต

ใช้คำนำหน้าที่สั้นที่สุดของคีย์กลางเป็นตัวแบ่งคีย์ คีย์ที่มีคำนำหน้ามาหลังคำดังกล่าวจะถูกย้ายไปยังบัคเก็ตใหม่ โดยขั้นตอนวิธีนั้นจะเป็นไปตามอัลกอริทึมข้างล่างนี้

Algorithm 2 TH Bucket Splitting: ให้ n แทนโหนดที่เยี่ยมอยู่, M แทนที่อยู่บัคเก็ตที่ล้น, N แทนที่อยู่บัคเก็ตล่าสุดในแฟ้มข้อมูล, $c' = c'_0c'_1c'_2...$ แทนคีย์กลาง, c แทนคีย์ใดๆในบัคเก็ต M, C แทนทางเดินตรรกะที่ได้จาก Algorithm 1 และ $L(n)$ และ $R(n)$ แทนลูกทางซ้ายและลูกทางขวาของ n ตามลำดับ

Algorithm 2 TH Bucket Splitting

```
1:  n ← M;
2:  compute shortest prefix (c')_i
```

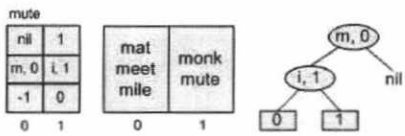
3: $N \leftarrow N + 1$;
4: Append bucket N
5: Move all c where $(c)_i > (c')_i$ to bucket N
6: compute the largest $(c')_i$ such that $(c')_i = (C)_i$
7: **for** $j = l + 1$ to $i - 1$ **do**
8: $n \leftarrow (c'_j, j)$; $L(n) \leftarrow M$; $R(n) \leftarrow nil$;
9: $n \leftarrow L(n)$;
10: **end for**
11: $n \leftarrow (c'_i, i)$; $L(n) \leftarrow M$; $R(n) \leftarrow N$;

จากอัลกอริทึมดังกล่าว สามารถสรุปออกมาเป็นขั้นตอนได้ดังนี้คือ เริ่มจากคำนวณหาคำนำหน้าที่สั้นที่สุดของคีย์กลาง $(c')_i$ เพื่อใช้ในการเลือกคีย์ที่จะย้าย ทำการต่อบัคเก็ตใหม่ ย้ายคีย์ที่ถูกเลือกไปยังบัคเก็ตใหม่ และสุดท้ายนี้คำนวณ $(c')_i$ ซึ่งมีค่าตรงกับทางเดินตรรกะ เพื่อทำการหาจำนวนโหนดที่ขยายและขยายโหนดเป็นลำดับต่อไป

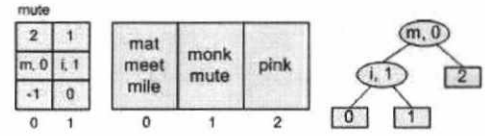
2.5. การเพิ่ม

เริ่มจากทำการหาที่อยู่บัคเก็ตตาม Algorithm 1 จากนั้นทำการดึงบัคเก็ตที่มีที่อยู่ดังกล่าวลงในหน่วยความจำหลัก ถ้าบัคเก็ตยังไม่เต็ม สามารถทำการเพิ่มได้ทันที แต่ถ้าเกิดเต็ม ต้องทำการแบ่งบัคเก็ตตาม Algorithm 2 จาก Figure 2 ซึ่งได้สรุปจาก [3] ถ้าเกิดมีการขยายมากกว่า 1 โหนด จะมีโหนด *nil* เพิ่มขึ้นมาดังตัวอย่างใน Figure 2(b) และถ้ามีการเพิ่มคีย์ใหม่ โดยพบเจอ *nil* จะมีการเปลี่ยนเป็นบัคเก็ตใหม่ดัง Figure 2(c)

(a) บัคเก็ตฉัน



(b) ผลจากการแบ่ง



(c) เพิ่มคีย์ 'pink'

Figure 2. ตัวอย่างการแบ่งในกรณีที่ขยายมากกว่า 1 โหนด

2.6. การลบ

เริ่มจากทำการค้นหาบัคเก็ตทำนองเดียวกับการเพิ่ม จากนั้นจึงทำการลบระเบียนที่ระบุโดยคีย์ที่ต้องการลบ ให้โหนดภายนอกที่เก็บค่าที่อยู่บัคเก็ตซึ่งได้จาก Algorithm 1 แทนด้วย P และโหนดพี่น้อง (sibling) ของ P แทนด้วย Q ซึ่งต้องเป็นโหนดภายนอกเช่นกัน หลังจากการลบนั้น ถ้าจำนวนระเบียนในบัคเก็ต P และ Q รวมกันได้ไม่เกิน b ให้ทำการรวมโหนดกัน (merge) เป็นโหนดภายนอกทางซ้าย หด trie ขึ้นไปเรื่อยๆ จนกระทั่งพบโหนดพี่น้องไม่ที่เป็น *nil* ยกตัวอย่างเช่น จาก Figure 1(c) หลังจากลบระเบียนในบัคเก็ต 0 ทำการรวมโหนด 0กับโหนด 9 เป็นโหนด 0 และหดไปเป็นลูกทางซ้ายของ ('a', 0)

2.7. การปรับปรุง

สำหรับวิธีการปรับปรุงในที่นี้คือการลบและทำการเพิ่มใหม่เพื่อทำการรักษาคุณสมบัติ weak order preserving เอาไว้

3. แฮชชิงทรัยแบบขยายได้ (Extendible trie hashing)

เป็นวิธีใหม่ที่พัฒนาขึ้นมาโดยใช้โครงสร้างข้อมูลซึ่งมีชื่อเรียกว่า "extendible trie" ในการคำนวณหาที่อยู่บัคเก็ต โดยโครงสร้างข้อมูลดังกล่าวเกิดจากการรวมกันระหว่างวิธีแฮชชิงแบบขยายได้และแฮชชิงทรัย

3.1. โครงสร้างข้อมูล

โหนดภายในแต่ละโหนดประกอบไปด้วยเซตข้อมูลซึ่งเรียกว่า "ไดเรกทอรี (directory)" โดยจะมีจำนวนเท่ากับ Σ แต่ละไดเรกทอรีจะมีดิจิตใน Σ กำกับอยู่และเรียงลำดับด้วย ใช้เป็น

ชื่อไดเรกทอรี โหนดภายนอกเป็นที่อยู่บัคเก็ต โครงสร้างนี้ใช้เนื้อที่ในการสร้าง $O(|\Sigma|^m)$ และมีลักษณะดังต่อไปนี้

1) โหนดภายในแต่ละโหนดมีแม่เป็นไดเรกทอรีเพียงไดเรกทอรีเดียวเท่านั้น
2) โหนดภายนอกแต่ละโหนดอาจมีแม่เป็นไดเรกทอ-รีเพียงหนึ่งหรือหลายไดเรกทอรีซึ่งอยู่ติดกัน

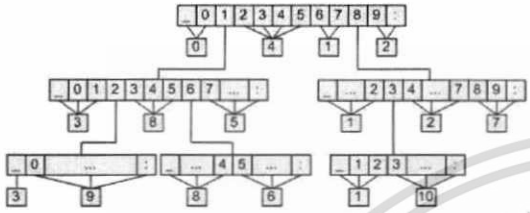


Figure 3. ตัวอย่างโครงสร้างของแฮชชิงทรัยแบบขยายได้เมื่อ Σ แทนเซ็ตของเลขฐานสิบ

3.2. การค้นหาคีย์

สำหรับการค้นหาคีย์นั้น จะใช้ดิจิตของคีย์ ณ ตำแหน่งเดียวกับระดับของโหนดที่เยี่ยมอยู่เป็นเกณฑ์ในการท่องเข้าไป หมายความว่าโหนดที่เยี่ยมอยู่ตรงกับระดับใด ให้เข้าไปที่ไดเรกทอรีที่มีชื่อตรงกับดิจิตของคีย์ ณ ตำแหน่งนั้น ตัวอย่างเช่น ต้องการค้นหาคีย์ 'choose' ถ้าเยี่ยมอยู่ที่ระดับ 0 (ราก) ให้เข้าไปที่ไดเรกทอรี c หรือถ้าอยู่ที่ระดับ 1 ให้เข้าไปที่ไดเรกทอรี h เป็นต้น ซึ่งสามารถสรุปเป็นขั้นตอนวิธีได้ดังอัลกอริทึมข้างล่างนี้ โดยให้ผลลัพธ์กลับคืนมาเป็นที่อยู่บัคเก็ตและโหนดๆหนึ่งสำหรับใช้ในขั้นตอนวิธีการแบ่งบัคเก็ต

Algorithm 3 EXTH Key Search: ให้ $k = k_0k_1k_2...$ แทนคีย์ที่ค้นหา, r แทนราก, N แทนโหนดที่เยี่ยมอยู่, $N[A]$ แทนไดเรกทอรี A ใน N และ P แทนโหนดๆหนึ่ง

Algorithm 3 EXTH Key Search

```
1: N ← r; i = 0;
2: while N is an internal node do
3:    P ← N; N ← N[k_i]; i ← i + 1;
4: end while
5: return N, P
```

ตัวอย่างเช่น จาก Figure 3 ถ้าคีย์ที่ใช้ค้นหาคือ '14203' จะทำการท่องไปที่ไดเรกทอรี 1 ณ ระดับ 0 และท่องไปยังไดเรกทอรี 4 ณ ระดับ 1) พบที่อยู่บัคเก็ตหมายเลข 8 ฉะนั้นจึงคืนค่ากลับมาเป็นที่อยู่หมายเลข 8 และโหนดที่ได้เข้าเยี่ยมที่ระดับ 1 ดังกล่าว

3.3. การแบ่งบัคเก็ต

การแบ่งบัคเก็ตเมื่อบัคเก็ตเกิดล้นนั้น มีขั้นตอนวิธีเป็นไปตาม Algorithm 4 โดยใช้หลักการเดียวกันกับแฮชชิงทรัย แต่จะมีการกำจัด *nil* ด้วยโดยการแทนเป็นที่อยู่บัคเก็ตที่เพิ่มล่าสุดลงไป เพื่อเป็นการช่วยเพิ่มค่าโหลดแฟกเตอร์

Algorithm 4 EXTH Bucket Splitting: ให้ N แทนโหนดที่เยี่ยมอยู่, $N[A]$ แทนไดเรกทอรี A ใน N, B แทนที่อยู่บัคเก็ตที่ล้น, B' แทนที่อยู่บัคเก็ตล่าสุดในแฟ้มข้อมูล, $c' = c'_0c'_1c'_2...$ แทนคีย์กลาง, c แทนคีย์ใดๆในบัคเก็ต B, P แทนโหนดที่ได้จาก Algorithm 3 และ l แทนระดับ P อยู่

Algorithm 4 EXTH Bucket Splitting

```
1:  N ← P;
2:  compute (c')_r
3:  B' ← B' + 1;
4:  Append bucket B'
5:  Move all c where (c)_r > (c')_r to B'
6:  if l' > l then
7:    for i = l to l' - 1 do
8:      Create new node N'
9:      N[c'_i] ← N'; N ← N[c'_i];
10:     for each alphabet alp in P do
11:       if alp ≤ c'_{i+1} then N[alp] ← B;
12:       else N[alp] ← B';
13:       end if
14:     end for
15:   end for
16:   N ← P;
17: end if
18: for each alphabet alp in P where alp > c'_i and N[alp] =
    B do
```

19: $N[alp] \leftarrow B'$;
20: **end for**
21: *nextalp* ← alphabet where it is next to the last *alp*;
22: **while** $N[nextalp]$ is an internal node **do**
23: **for** each alphabet *alp* in P where $N[alp] = B$ **do**
24: $N[alp] \leftarrow B'$;
25: **end for**
26: *nextalp* ← alphabet where it is next to the last *alp*;
27: **end while**

จากอัลกอริทึมดังกล่าวสามารถสรุปเป็นขั้นตอนได้ดังนี้

1. คำนวณหาคำนำหน้าที่สั้นที่สุดของคีย์กลาง $(c')_r$ ซึ่งแบ่งคีย์ c โดยที่ $(c)_r > (c')_r$ ไปยังบัคเก็ตใหม่
2. เพิ่มค่า B', ต่อบัคเก็ต B' แล้วย้ายคีย์ที่มีสมบัติดังกล่าวในข้อ 1) ไปยังบัคเก็ต B'
3. นำ l และ l' มาทำการเปรียบเทียบกัน
   1) ถ้า $l' = l$ ไดเรกทอรีที่มีชื่อตั้งแต่หลังดิจิต c'_l เป็นต้นไปที่โยงไปยัง B ในระดับ l และไดเรกทอรีทั้งหมดที่โยงไปยัง B ในระดับหลัง l ลงไปนั้นโยงไป ยัง B'
   2) ถ้า $l' > l$ แต่ละระดับ $i = l,...,l'$-1 ไดเรกทอรี c'_i จะโยงไปยังโหนดใหม่ N' และให้ไดเรกทอรีใน N' ที่มีชื่อก่อนหรือตรงกับดิจิต c'_{i+1} โยงไปยัง B ส่วนไดเรกทอรีที่เหลือโยงไปยัง B' นอกจากนี้ไดเรกทอรีที่มีชื่อหลังดิจิต c'_i เป็นต้นไปที่โยงไป B ในระดับ l และไดเรกทอรีทั้งหมดที่โยงไปยัง B ในระดับหลัง l ลงไปนั้นโยงไปยัง B' ด้วย

Figure 4 แสดงตัวอย่างผลการแบ่งบัคเก็ตในกรณีต่างๆ โดยทั่วๆไปนั้นมักจะพบในกรณี Figure 4(a) โดยในที่นี้ $l = 0$ และ $(c')_r$ = '5' นั่นคือตั้งแต่หลังไดเรกทอรี 5 เป็นต้นไปในระดับ 0 โยงไปยังที่อยู่หมายเลข 1

และน้อยครั้งที่พบกรณีดัง Figure 4(b) ซึ่งในที่นี้ $l = 0$ และ $(c')_r$ = '53' จะได้ว่าไดเรกทอรี 5 ทำการโยงไปยังโหนดใหม่และไดเรกทอรีที่มีชื่อตั้งแต่หลัง 5 เป็นต้นไปโยงไปยังที่อยู่หมายเลข 1 ส่วนโหนดใหม่ ไดเรกทอรีที่มีชื่อก่อนหรือตรงกับ 3 โยงไปยังที่อยู่หมายเลข 0 ส่วนที่เหลือโยงไปที่อยู่หมายเลข 1 และจาก Figure 4(b) ถ้าบัคเก็ต 0 ล้นอีก และ P ที่ได้จาก Algorithm 3 เป็นโหนดที่ได้เข้าเยี่ยมที่ระดับ 0 นอกจากเปลี่ยนการโยงของไดเรกทอรีที่ระดับ 0 แล้ว ยังต้องเปลี่ยนการโยงตั้งแต่ระดับ 1 ลงไปด้วยดัง Figure 4(c)

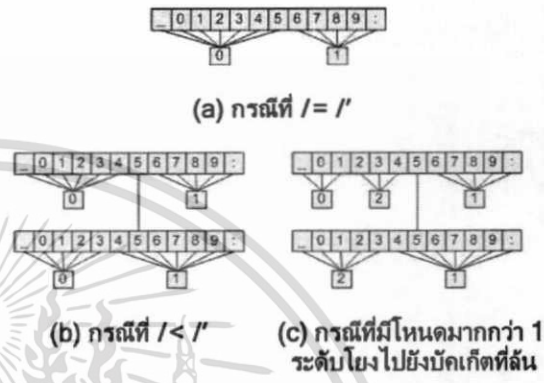


Figure 4. ตัวอย่างผลการแบ่งบัคเก็ตในกรณีต่างๆ

3.4. การเพิ่ม

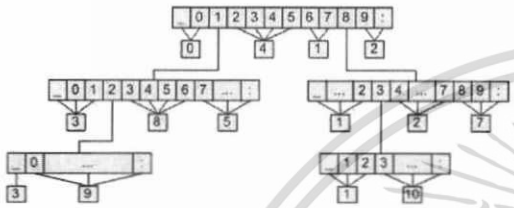
ทำโดยการหาที่อยู่บัคเก็ตตามขั้นตอนวิธีตาม Algorithm 3 จากนั้นทำการดึงบัคเก็ตที่มีที่อยู่ดังกล่าวลงในหน่วยความจำหลัก ถ้าบัคเก็ตดังกล่าวยังไม่เต็ม สามารถทำการเพิ่มได้ทันที ในกรณีที่เต็มต้องทำการแบ่งบัคเก็ตตาม Algorithm 4

3.5. การลบ

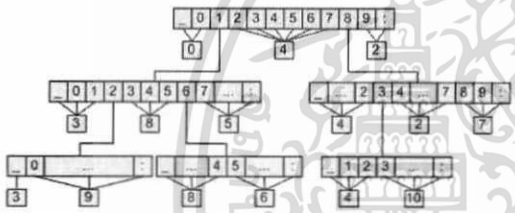
เริ่มจากทำการค้นหาบัคเก็ตทำนองเดียวกับการเพิ่ม จากนั้นจึงทำการลบระเบียนที่ระบุโดยคีย์ที่ต้องการลบ ให้ Y แทนโหนดภายนอกที่ได้จาก Algorithm 3, X แทนโหนดก่อนหน้า Y ที่ถูกโยงอยู่ติดกัน และ Z แทนโหนดถัดจาก Y ที่ถูกโยงอยู่ติดกัน หลังจากการลบ ถ้ามีคุณสมบัติตรงกับข้อใดข้อหนึ่งดังนี้คือ

1) X เป็นโหนดภายนอกและผลรวมของจำนวนระเบียนในบัคเก็ต X กับ Y ไม่เกิน b
2) Z เป็นโหนดภายนอกและผลรวมของจำนวนระเบียนในบัคเก็ต Y กับ Z ไม่เกิน b

ให้ทำการรวมโหนดซึ่งได้ผลลัพธ์ออกมาเป็นดังนี้คือ ถ้าตรงกับ 1) ให้รวมเป็น X แต่ถ้าตรงกับ 2) ให้รวมเป็น Y ถ้าตรงกันทั้ง 2 ข้อให้เลือกข้อใดข้อหนึ่ง และหลังจากรวมโหนด ถ้าพบว่าเหลือโหนดภายนอกเพียงโหนดเดียวที่ถูกโยงอยู่ ให้ทำการหด extendible trie ขึ้นไป มีตัวอย่างประกอบใน Figure 5 โดย Figure 5(a) เป็นผลลัพธ์จากการลบระเบียนในบัคเก็ต 6 และตรงกับข้อ 1) ส่วน Figure 5(b) เป็นผลลัพธ์จากการลบระเบียนในบัคเก็ต 4 และตรงกับข้อ 2)



(a) กรณีลบระเบียนในบัคเก็ต 6



(b) กรณีลบระเบียนในบัคเก็ต 4

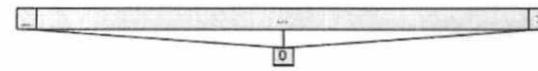
Figure 5. ตัวอย่างการลบของวิธี Extendible trie hashing

3.6. การปรับปรุง

ทำในลักษณะเดียวกับแฮชชิงทรัยคือ ทำการลบและทำเพิ่มใหม่ เพื่อทำการรักษาคุณสมบัติ weak order preserving เอาไว้

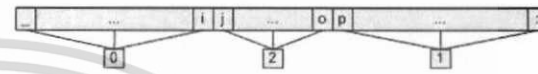
3.7. การสร้าง

โครงสร้างที่ได้มานั้น มาจากการเพิ่มคีย์ โดยเริ่มต้นจะมีโหนดภายในอยู่หนึ่งโหนดซึ่งแต่ละไดเรกทอรีนั้นจะโยงไปยังที่อยู่บัคเก็ตหมายเลข 0 ดัง Figure 6(a) ทำการเพิ่มคีย์ลงไปเพื่อทำการขยายโหนด ผลลัพธ์สุดท้ายของโครงสร้างที่ได้จะขึ้นอยู่กับการแบ่งบัคเก็ตเมื่อบัคเก็ตล้นในแต่ละครั้ง ตัวอย่างการสร้างในที่นี้ใช้ลำดับคีย์เดียวกันกับ Figure 1(a) ผลของการแบ่งนั้นในที่นี้ได้แสดงผลไว้ดัง Figure 6(b) – 6(d) เมื่อทำการเพิ่มคีย์ครบถ้วน จะได้ผลลัพธ์ออกมาดัง Figure 6(e)

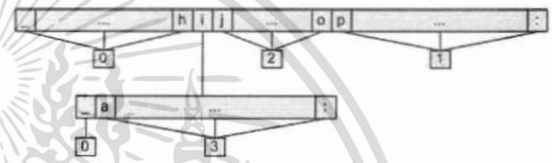


(a) สถานะเริ่มต้น

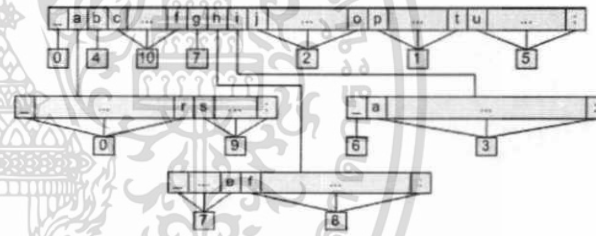
(b) ผลลัพธ์หลังจากทำการแบ่งครั้งที่ 1



(c) ผลลัพธ์หลังจากทำการแบ่งครั้งที่ 2



(d) ผลลัพธ์หลังจากทำการแบ่งครั้งที่ 3



(e)โครงสร้างหลังจากเพิ่มคีย์ครบ

Figure 6. สถานะเริ่มต้นและตัวอย่างโครงสร้างหลังจากการแบ่งในแต่ละครั้ง

4. วิเคราะห์

เนื่องจากจุดประสงค์ของการทำงานวิจัยชิ้นนี้คือเพื่อเพิ่มความเร็วในการค้นหาที่อยู่บัคเก็ต ดังนั้นจึงเปรียบเทียบเวลาที่ใช้ในการทำงานเฉพาะการค้นหาที่อยู่บัคเก็ตเท่านั้น โดยคำนวณจาก pseudo code

แฮชชิงทรัย - เริ่มจากบรรทัดที่ 1 นั้นเป็นส่วนของการกำหนดค่าเริ่มต้น ใช้เวลาทำงาน $O(1)$ ถัดมาตั้งแต่บรรทัดที่ 2 - 14 เป็นส่วนของลูป while ซึ่งทำหน้าที่ท่องไปใน trie โดย

โครงสร้างนั้นจะขึ้นอยู่กับผลจากการแบ่งบัคเก็ต ในการแบ่งแต่ละครั้งจะมีการขยายโหนดภายในไม่เกิน m ซึ่งถ้าเกิดการแบ่ง k ครั้ง จะได้ว่ามีจำนวนโหนดภายในทั้งหมดไม่เกิน km ดังนั้นในบรรทัดที่ 2 จึงใช้เวลาทำงาน $O(km)$ และแต่ละรอบในลูป while นั้นเป็นการทำงานในส่วนของ if-else ซึ่งจะเห็นได้ว่าไม่ว่าจะเป็นส่วนของเงื่อนไขใดก็ตาม เวลาที่ใช้ทำงานคือ $O(1)$ ดังนั้นเวลาที่ใช้ทำงานคือ $O(1)$ นั่นคือตั้งแต่บรรทัดที่ 2 - 14 นั้น ใช้เวลาทำงาน $O(km)O(1) = O(km)$ และสุดท้ายที่บรรทัดที่ 15 เป็นส่วนของการคืนผลลัพธ์ ใช้เวลาทำงาน $O(1)$ เพราะฉะนั้นจึงสามารถสรุปได้ว่าใช้เวลาในการทำงานทั้งหมดนั้น มีค่าเท่ากับ

$= O(1) + O(km) + O(1)$

$= O(km)$

แฮชชิงทรัยแบบขยายได้ - บรรทัดที่ 1 เป็นส่วนของการกำหนดค่าเริ่มต้นซึ่งใช้เวลาทำงาน $O(1)$ เช่นกัน ถัดมาตั้งแต่บรรทัดที่ 2 - 4 เป็นส่วนของ while ซึ่งทำหน้าที่ท่องไปใน extendible trie โดยทำการท่องไม่เกิน m ระดับ เนื่องจากใช้ดิจิตของคีย์ ณ ตำแหน่งเดียวกับระดับของโหนดที่เยี่ยมอยู่ เป็นเกณฑ์ในการท่องเข้าไป และคีย์นั้นมีความยาวไม่เกิน m ดังนั้นที่บรรทัดที่ 2 จึงใช้เวลา $O(m)$ และแต่ละรอบในลูป while นั้นใช้เวลาทำงาน $O(1)$ นั่นคือใช้เวลาทำงานในส่วนของ while $O(m)O(1) = O(m)$ และสุดท้ายในบรรทัดที่ 5 เป็นส่วนของการคืนผลลัพธ์ ใช้เวลาทำงาน $O(1)$ เพราะฉะนั้นเวลาที่ใช้ในการทำงานทั้งหมดจึงเท่ากับ

$= O(1) + O(m) + O(1)$

$= O(m)$

5. สรุป

งานวิจัยชิ้นนี้ได้จัดทำขึ้นมาเพื่อทำการแก้ปัญหา degeneracy ของแฮชชิงทรัย และเพิ่มความเร็วในการสืบค้นหาที่อยู่บัคเก็ตด้วย โดยใช้วิธีแฮชชิงทรัยแบบขยายได้ซึ่งใช้ extendible trie เป็นตัวคำนวณหาที่อยู่บัคเก็ต สำหรับวิธีนี้จะใช้เวลาในการสืบค้นหาที่อยู่บัคเก็ต $O(m)$ แต่ทว่าต้องใช้พื้นที่ในหน่วยความจำอย่างมากถึง $O(|\sum|^m)$ ซึ่งถ้าสามารถนำมาปรับปรุงให้ใช้พื้นที่น้อยกว่านี้ได้ จะช่วยให้สามารถคำนวณในปริมาณระเบียนที่มากขึ้นกว่านี้ด้วย

เอกสารอ้างอิง

[1] M. Aridj and D. E. Zegour. TH*: Scalable distributed trie hashing. IJCSI International Journal of Computer Science Issues, 7(6):109–115, November 2010.

[2] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. ACM Trans. Database Syst., 4:315–344, September 1979.

[3] W. Litwin. Trie hashing. In Proceedings of the 1981 ACM SIGMOD international conference on Management of data, SIGMOD '81, pages 19–29, New York, NY, USA, 1981. ACM.

[4] W. Litwin, D. Zegour, and G. Levy. Multilevel trie hashing. In Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology, EDBT '88, pages 309–335, London, UK, 1988. Springer-Verlag.

[5] W. A. Litwin, N. Roussopoulos, G. Levy, and W. Hong. Trie hashing with controlled load. 17(7):678–691, 1991.

[6] E. J. Otoo and S. Effah. Red-black balanced trie hashing, 1995.

[7] L. Torenvliet and P. v. E. Boas. The reconstruction and optimization of trie hashing functions. In Proceedings of the 9th International Conference on Very Large Data Bases, pages 142–156, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[8] D. E. Zegour. Scalable distributed compact trie hashing (CTH*). Information & Software Technology, 46:923–935, 2004.

ประวัติผู้เขียน

ชื่อ – นามสกุล นายมานิตย์ ขวัญยืน

วัน เดือน ปี เกิด 1 เมษายน พ.ศ. 2530

ที่อยู่ 122/1 ถ.ราษฎร์บำรุง ซ.ปรางศรี 2 ต.เนินพระ อ.เมือง จ.ระยอง 21000

ประวัติการศึกษา

2551 เกียรตินิยมอันดับ 2 วิทยาศาสตรบัณฑิต สาขาคณิตศาสตร์ประยุกต์
คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง