บทที่ 2

แนวความคิดและทฤษฎี

1. การค้นหาข้อมูล

โดยทั่วไปเมื่อมีการกล่าวถึง ฐานข้อมูล (Database) หรือแฟ้มข้อมูล (File) ในคอมพิวเตอร์ มักจะหมายถึง ฐานข้อมูล หรือแฟ้มข้อมูล ที่เกิดจากการรวมกันของ ระเบียน (records) ซึ่งแต่ละระเบียนจะประกอบด้วย เขตข้อมูล (fields) อย่างน้อย 1 เขตข้อมูล การค้นหาข้อมูล จะถูกกำหนดขึ้นจากพื้นฐานของข้อมูลที่เก็บในระเบียน เขตข้อมูลที่ใช้เป็นตัวระบุเกณฑ์ หรือมาตรฐานในการค้นคืน เรียกว่า คีย์ฟิลด์ (key field) (Salton, 1989) ตัวอย่างเช่น การค้นหาข้อมูล ในแฟ้มข้อมูลบุคลากร ซึ่งบรรจุสารสนเทศ เกี่ยวกับพนักงาน สามารถทำได้ โดยการใช้คีย์ที่เป็นชื่อของพนักงาน หรือประเภทของงาน หรือระดับเงินเดือน ในทำนองเดียวกัน แฟ้มข้อมูลเอกสาร (Document file) สามารถค้นหาได้ โดยการใช้คีย์ที่เป็นชื่อของผู้เขียนเอกสาร หรือชื่อเอกสาร หรือคำในเอกสาร เป็นต้น

ในแต่ละกรณี ของการค้นหา แฟ้มข้อมูลจะถูกค้นหา โดยการเปรียบเทียบ คำถามที่ต้องการกับลักษณะเฉพาะของเขตข้อมูลในระเบียนของแฟ้มข้อมูล ระเบียนที่มีค่าคีย์ตรงกับคำถามที่ต้องการ ก็จะถูกค้นคืนออกมา คำถามที่ต้องการอาจเป็นคำถามแบบง่ายๆ (simple query) ที่ประกอบด้วยคีย์เพียง 1 ตัว หรือคำถามที่เป็นช่วง (range query) ที่ประกอบด้วยช่วงของค่าของคีย์ 1 ตัว เช่น ต้องการค้นหาข้อมูล ทุกระเบียนของพนักงาน ที่มีอายุ 22-25 ปี การสอบถามข้อมูลอาจทำการสอบถามโดยใช้คีย์เดียว (single key) หรือสอบถามโดยใช้หลายคีย์ (multiple key) ในกรณีทั่วไป จะใช้วิธีการสอบถามแบบคีย์เดียว เป็นมาตรฐานในการค้นหาข้อมูล ในขณะที่การสอบถามแบบหลายคีย์ อาจทำให้จำนวนของคีย์ที่ต่างกันเกิดความสับสนขึ้นได้

สำหรับการสอบถามแบบคีย์เดียว แฟ้มข้อมูลสามารถจัดข้อมูลให้อยู่ในลำดับที่สอดคล้องกับค่าคีย์ที่ใช้ในการสอบถามได้ ตัวอย่างเช่น แฟ้มข้อมูลของผู้ใช้บริการโทรศัพท์ ปกติจะจัดเรียงตามลำดับตัวอักษรของชื่อผู้ใช้บริการ เนื่องจากการค้นหาข้อมูล ถูกออกแบบให้ค้นหาหมายเลขโทรศัพท์ที่สอดคล้องกับชื่อผู้ใช้บริการ การค้นหาแบบหลายคีย์ จะซับซ้อนขึ้น เนื่องจากมันเป็นไปไม่ได้ที่เราจะเรียงลำดับข้อมูลในแฟ้ม ตามค่าของคีย์ที่แตกต่างกัน ในเวลาเดียวกัน ดังนั้น แฟ้มข้อมูลพนักงาน ที่จัดเก็บแบบเรียงลำดับตามชื่อ จึงไม่สามารถเรียงลำดับตามอายุด้วยได้

วิธีการเข้าถึงแฟ้มข้อมูล ถูกกำหนดขึ้นเพื่อใช้สำหรับการค้นคืนข้อมูล ทั้งแบบการค้นหาแบบคีย์เดียว และการค้นหาแบบหลายคีย์ ซึ่งเราสามารถแบ่งออกได้เป็น 2 กรณี คือ (Kruse, 1987)

1.1 <u>การค้นหาแบบภายใน (Internal Searching)</u>

เป็นการค้นหาข้อมูล ที่มีการเก็บไว้ ในหน่วยความจำของคอมพิวเตอร์

1.2 <u>การค้นหาแบบภายนอก (External Searching)</u>

เป็นการค้นหาข้อมูล ในแฟ้มข้อมูลขนาดใหญ่ ที่มีการเก็บไว้ ในหน่วยความจำภายนอก หรือหน่วยความจำสำรอง (secondary storage) เช่น จานแม่เหล็ก (disk) หรือ เทป (tape) ในการประยุกต์ใช้งานส่วนใหญ่ เราจะเก็บข้อมูล ไว้ในหน่วยความจำภายนอก หรือหน่วยความจำสำรอง ซึ่งมีวิธีการในการค้นหาดังนี้

1.2.1 <u>การค้นหาแบบลำดับ (Sequential Search)</u>

การค้นหาแบบลำดับ จะทำการค้นหาโดย ทำการเปรียบเทียบแต่ละระเบียนที่มีการเก็บไว้ กับข้อมูลที่ต้องการทีละระเบียน และทำการค้นคืนทุกระเบียน ที่มีค่าคีย์ตรงกับค่าคีย์ที่ต้องการ เนื่องจาก ในการค้นหา มีการเปรียบเทียบระเบียนทุกระเบียน ดังนั้นระเบียนในแฟ้มข้อมูลจึงไม่จำเป็นต้องเก็บแบบเรียงลำดับ การบำรุงรักษาแฟ้มข้อมูลจึงทำได้ง่าย สามารถลดค่าใช้จ่าย เกี่ยวกับเนื้อที่ในการเก็บข้อมูลให้น้อยลงได้ เพราะไม่ต้องเก็บสารสนเทศที่เป็นตัวช่วยในการเข้าถึงแฟ้มข้อมูล การค้นหาแบบลำดับสามารถใช้ได้ ทั้งกับการค้นหาแบบคีย์เดียว และการค้นหาแบบหลายคีย์

ตัวอย่าง โปรแกรมการค้นหาข้อมูลแบบลำดับสำหรับคีย์เดียว

```
1. [Initialization]  K_{n+1} <- X
2. [Search file]     do for i = 1,2,...,n+1
                       if K_i = X
                        then if i = n+1
                          then print "unsuccessful"
                        else print R_i
                       end if
                     end do
```

จากโปรแกรม แต่ละระเบียน R_i สมมติว่าถูกระบุโดยคีย์ K_i ค่าคีย์ที่ต้องการค้นหาให้เป็น X ระเบียน R_{n+1} ใดๆ ที่มีค่าคีย์ K_{n+1} จะอยู่ท้ายแฟ้มข้อมูล ในการค้นหา จะต้องทำการเปรียบเทียบทั้งหมด (n+1) ครั้ง ระหว่างระเบียนที่ต้องการ กับระเบียนในแฟ้มข้อมูล ถ้าในการเขียนโปรแกรม ถูกสั่งให้หยุด หลังจากค้นพบระเบียนที่ต้องการ จำนวนครั้งสูงสุดในการค้นหาก็ยังคงต้องเปรียบเทียบ (n+1) ครั้ง สำหรับระเบียบที่ค้นหาไม่พบภายใต้ข้อสมมติฐานเดียวกัน จำนวนครั้งเฉลี่ยของการเปรียบเทียบ สำหรับการค้นหาระเบียนที่มีในแฟ้มข้อมูลจะเท่ากับ

$$(1+2+3+...+n)/n$$

$$1+2+3+...+n = 1/2*n*(n+1)$$

$$n*(n+1)/2*n = 1/2*(n+1)$$

ดังนั้นจำนวนครั้งเฉลี่ยจะเท่ากับ (n+1)/2 ครั้ง นั่นคือ ระเบียนที่ต้องการโดยเฉลี่ย จะค้นพบหลังจากการค้นหาผ่านไปครึ่งหนึ่งของแฟ้มข้อมูล

ถ้าความน่าจะเป็น ของการค้นคืนระเบียน มีการเปลี่ยนแปลงจากระเบียนหนึ่ง ไปยังอีกระเบียนหนึ่ง แฟ้มข้อมูลจะสามารถ เรียงลำดับความน่าจะเป็น ของการค้นคืนระเบียน จากมากไปหาน้อยได้ ซึ่งเป็นการลดจำนวนครั้งเฉลี่ย ในการค้นหาระเบียนที่ต้องการ เมื่อมีการแจกแจงความน่าจะเป็น ในการค้นคืนระเบียนที่ต้องการ ในการค้นหาแบบลำดับสำหรับแฟ้มข้อมูลที่มี n ระเบียน จะได้เป็น

$$\sum_{i=1}^{n} i*P_i$$

เมื่อ i คือจำนวนครั้งของการเปรียบเทียบในการค้นหาระเบียนที่ i และ P_i คือความน่าจะเป็นในการเข้าถึงระเบียนที่ i

พิจารณาตัวอย่าง แฟ้มข้อมูลที่มี 5 ระเบียน (n = 5) เมื่อทุกระเบียนมีความน่าจะเป็น P_i ซึ่ง $P_i = 1/n = 1/5$ การค้นหาเฉลี่ยจะเท่ากับ

$$\sum_{i=1}^{5} i*P_i = 1*1/5 + 2*1/5 + 3*1/5 + 4*1/5 + 5*1/5 = 15/5 = 3$$

หรืออีกนัยหนึ่ง ถ้าระเบียนทั้ง 5 มีความน่าจะเป็นแตกต่างกัน เช่น $P_1 = 0.4$, $P_2 = 0.3$, $P_3 = 0.2$, $P_4 = 0.07$, $P_5 = 0.03$ ค่าเฉลี่ยในการค้นหาจะลดลงประมาณ 1 ใน 3 โดยการจัดระเบียนให้ความน่าจะเป็นในการเข้าถึง อยู่ในลำดับที่เรียงจากมากไปหาน้อย ดังนี้

$$\sum_{i=1}^{5} i*P_i = 1*0.4 + 2*0.3 + 3*0.2 + 4*0.07 + 5*0.03 = 2.03$$

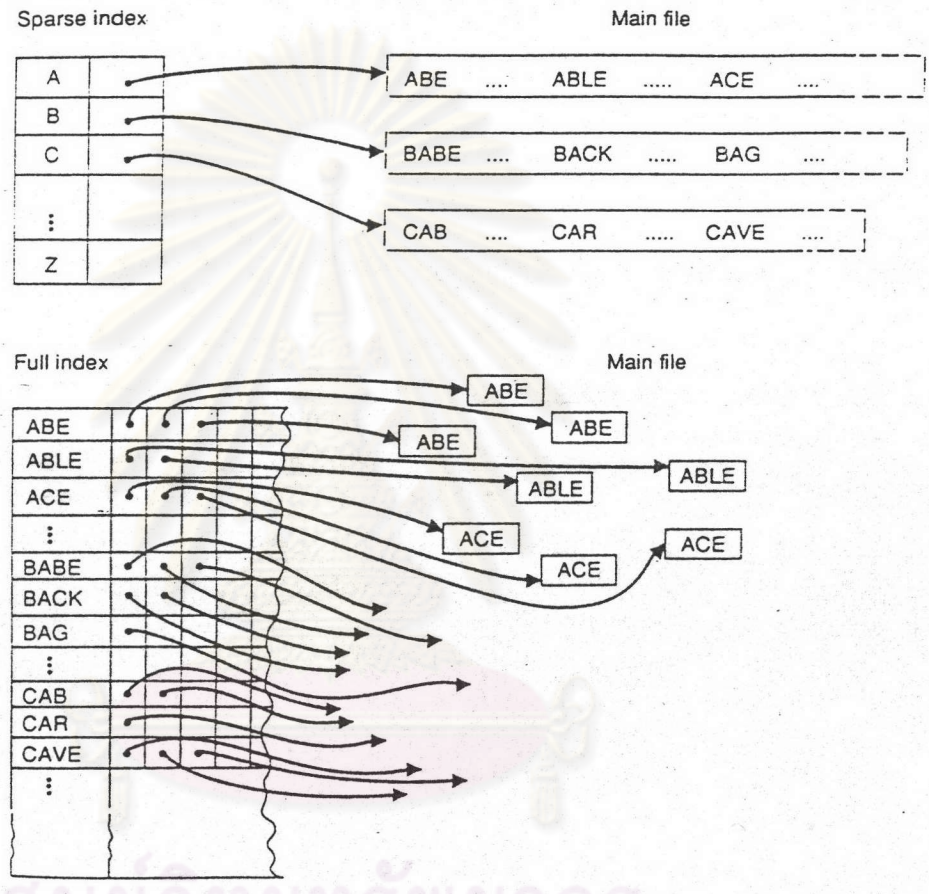
ค่าใช้จ่ายในการค้นหาข้อมูลสามารถลดให้ต่ำลงได้ถ้าความน่าจะเป็นในการเข้าถึงข้อมูลเป็นค่าคงที่ และข้อมูลในแฟ้มถูกจัดให้อยู่ในลำดับที่เหมาะสม สำหรับแฟ้มข้อมูลที่มีขนาดใหญ่ การค้นหาแบบลำดับจะต้องใช้เวลามากขึ้น แม้ว่าการจัดเรียงข้อมูลในแฟ้มจะอยู่ในลำดับที่เหมาะสม ดังนั้นการค้นหาแบบลำดับ จึงเหมาะสำหรับ การค้นหาข้อมูลในแฟ้มข้อมูลที่มีขนาดเล็กเท่านั้น

1.2.2 การค้นหาเชิงดัชนี (Indexed Search)

แฟ้มข้อมูลดัชนี (Index file) เป็นตาราง หรือแฟ้มข้อมูลที่บรรจุค่าคีย์ของระเบียน (record key) พร้อมกับตัวชี้ (pointer) ซึ่งจะแทนที่อยู่ (address) ของระเบียนในแฟ้มข้อมูล ที่สอดคล้องกับค่าคีย์ เมื่อข้อมูลในแฟ้มมีการจัดเรียงลำดับตามค่าคีย์ แฟ้มข้อมูลดัชนี อาจจะบรรจุค่าคีย์เพียงบางคีย์ และตำแหน่งระเบียนของแต่ละคีย์ เรียกว่า สปาร์สอินเด็กซ์ (sparse index) ในการค้นหาระเบียนอื่นๆ ที่มีค่าคีย์สัมพันธ์กันจะค้นหาข้อมูลในแฟ้มที่ชี้ตำแหน่งใกล้เคียงกัน เช่น การค้นหาข้อมูลในแฟ้มข้อมูลโทรศัพท์ ดัชนีอาจจะมีเพียง 26 รายการ แต่ละรายการจะบรรจุ ตัวอักษรเริ่มต้นของชื่อลูกค้า พร้อมกับหมายเลขหน้า ของแฟ้มข้อมูล ของลูกค้าคนแรกที่ชื่อขึ้นต้นด้วยตัวอักษรตัวนั้น ในการค้นหาตำแหน่งของหมายเลขโทรศัพท์ จะทำการค้นหาแบบลำดับ โดยเริ่มต้นจากหน้าที่ระบุในดัชนี

สำหรับแฟ้มข้อมูลที่ไม่ได้จัดเก็บแบบเรียงลำดับตามค่าคีย์ แฟ้มข้อมูลดัชนีจะบรรจุค่าคีย์ พร้อมกับตำแหน่ง ของระเบียนทุกระเบียน ที่สอดคล้องกับค่าคีย์แต่ละตัว เรียกว่า ฟูลอินเด็กซ์ (full index) ดังนั้นแฟ้มดัชนีจะมีขนาดใหญ่ขึ้น แต่การค้นหาข้อมูลในแฟ้มข้อมูลหลัก จะค้นหาได้สมบูรณ์กว่า เพราะตำแหน่งของระเบียน ที่ต้องการทุกระเบียน ถูกระบุอยู่ในดัชนี

แฟ้มข้อมูลดัชนีที่เป็น ฟูลอินเด็กซ์ สามารถใช้ได้ทั้งกับ แฟ้มข้อมูลที่มีการเรียงลำดับ หรือแฟ้มข้อมูลที่ไม่มีการเรียงลำดับก็ได้ เพื่อให้การค้นหาข้อมูลกระทำได้ง่ายขึ้น แฟ้มดัชนีควรมีการเรียงลำดับตามค่าคีย์ ความแตกต่างระหว่าง สปาร์สอินเด็กซ์ และฟูลอินเด็กซ์ แสดงในรูป 2.1 ในสปาร์สอินเด็กซ์ ตัวชี้แต่ละตัวจะชี้ไปยังทุกระเบียน ที่มีค่าคีย์ขึ้นต้นด้วยตัวอักษรตัวนั้น ในฟูลอินเด็กซ์ ตัวชี้หนึ่งตัวจะชี้ไปยังระเบียนในแฟ้มข้อมูลหลักหนึ่งระเบียน



รูปที่ 2.1 แฟ้มข้อมูลดัชนี แบบสปาร์สอินเด็กซ์ และฟูลอินเด็กซ์

1.2.3 การค้นหาแบบทวิภาค (Binary Search)

การค้นหาแบบทวิภาค จะทำการค้นหาโดยการเปรียบเทียบ ข้อมูลที่ต้องการกับข้อมูลที่จุดกึ่งกลาง ของแฟ้มข้อมูล จากนั้นทำการค้นหา เฉพาะในครึ่งแรก หรือครึ่งหลังของแฟ้มข้อมูล โดยขึ้นอยู่กับข้อมูลที่ต้องการว่า มีค่าอยู่ในครึ่งแรกหรือครึ่งหลังของแฟ้มข้อมูล ด้วยวิธีการดังกล่าวนี้ ในแต่ละครั้งของการค้นหา เราสามารถลดขนาดของแฟ้มข้อมูล ที่ต้องการค้นหาลงได้ครึ่งหนึ่ง และด้วยการเปรียบเทียบเพียง 20 ครั้ง วิธีการนี้จะสามารถค้นหาข้อมูลชื่อลูกค้า ในแฟ้มข้อมูลโทรศัพท์ได้ประมาณ 1 ล้านชื่อ

การค้นหาข้อมูลแบบทวิภาค จะต้องมีการเก็บข้อมูลแบบเรียงลำดับ ซึ่งสิ้นเปลืองค่าใช้จ่ายมาก และในการเพิ่มหรือแทรก (insert) คีย์ 1 ตัว จะต้องทำการขยับคีย์อื่นๆ เป็นจำนวนมาก

2. การประมวลผลแฟ้มข้อมูลข้อความ

จากที่กล่าวมาข้างต้น จะเห็นว่าการประมวลผลด้วยคอมพิวเตอร์ จะดำเนินการกับฐานข้อมูล หรือแฟ้มข้อมูลที่มีโครงสร้างแน่นอน แต่ปัจจุบัน ระบบการประมวลผลฐานข้อมูลหรือแฟ้มข้อมูลข้อความเข้ามามีบทบาทมากขึ้น ระบบการประมวลผลข้อความ จะกระทำกับข้อความที่เขียนอยู่ในรูปของภาษาธรรมชาติ

โดยที่โครงสร้างของฐานข้อมูล หรือแฟ้มข้อมูลข้อความ (Text file) จะประกอบด้วยกลุ่มของเอกสาร (Documents) มารวมกัน และเอกสารจะประกอบด้วย กลุ่มของตอน (Paragraphs) มารวมกัน ตอนประกอบด้วยกลุ่มของประโยค (Sentences) มารวมกัน และประโยค ประกอบด้วยกลุ่มของคำ (Words) มารวมกัน (Gethin, 1987)

เนื่องจาก การประมวลผลฐานข้อมูล หรือแฟ้มข้อมูลข้อความ โดยปกติจะกระทำกับข้อความที่เป็นส่วนๆ เช่น คำ ดังนั้น เทคนิคพื้นฐานในการค้นหาข้อมูล ตามวิธีการดังกล่าวข้างต้นจึงไม่เหมาะสมที่จะนำมาใช้ในการค้นหาข้อมูลที่เป็นแฟ้มข้อมูลข้อความโดยตรง

ในระบบงานทั่วๆไปเราจะใช้ระบบแฟ้มข้อมูลผกผัน (Inverted file) ช่วยในการเก็บและการค้นคืนฐานข้อมูลหรือแฟ้มข้อมูลข้อความ เนื่องจากสามารถค้นหาข้อมูลได้รวดเร็ว

ระบบแฟ้มข้อมูลผกผัน จะประกอบด้วย แฟ้มข้อมูลเอกสาร (Document file) และสารบัญหรือที่รู้จักกันในนามว่า "ดัชนีผกผัน" (Inverted index) ดัชนีผกผันจะประกอบด้วยเซตข้อมูลที่เป็นดัชนี และรายการของหมายเลขอ้างอิง ของเอกสารที่สัมพันธ์กับดัชนี หมายเลขอ้างอิงของแต่ละเอกสาร จะระบุเอกสารที่ไม่ซ้ำกัน ดังนั้นในการค้นคืน เอกสารจะถูกระบุโดยดัชนีที่ต้องการค้นหา และเอกสารที่ระบุโดยหมายเลขอ้างอิงที่สัมพันธ์กับดัชนี ก็จะถูกอ่านมาจากแฟ้มข้อมูลเอกสาร (Salton and McGill, 1983)

ตัวอย่าง การเก็บและการค้นคืนสารสนเทศในแฟ้มข้อมูลข้อความ โดยใช้แนวความคิดของระบบแฟ้มข้อมูลผกผัน

ถ้าเอกสารชุดที่ 1 ถึงชุดที่ 6 ประกอบด้วยข้อความต่อไปนี้คือ

Document1 : control of pollution in rivers
Document2 : reduction of atmospheric pollution
Document3 : pollution and its effect on rivers
Document4 : effect of smoke pollution
Document5 : atmospheric smoke content
Document6 : pollution control of rivers

ในการจัดเก็บเอกสารทั้ง 6 ชุด อาจจะจัดเก็บแบบเรียงลำดับในแฟ้มข้อมูลข้อความ โดยใช้เครื่องหมายอัฒภาค (;) เป็นตัวแบ่งข้อมูล ได้ดังนี้คือ

control of pollution in rivers; reduction of atmospheric pollution; pollution and its effect on rivers; effect of smoke pollution; atmospheric smoke content; pollution control of rivers

การเก็บและการค้นคืนสารสนเทศ โดยใช้แนวความคิดของแฟ้มข้อมูลผกผัน จะแบ่งการทำงานออกเป็น 3 ขั้นตอนดังนี้

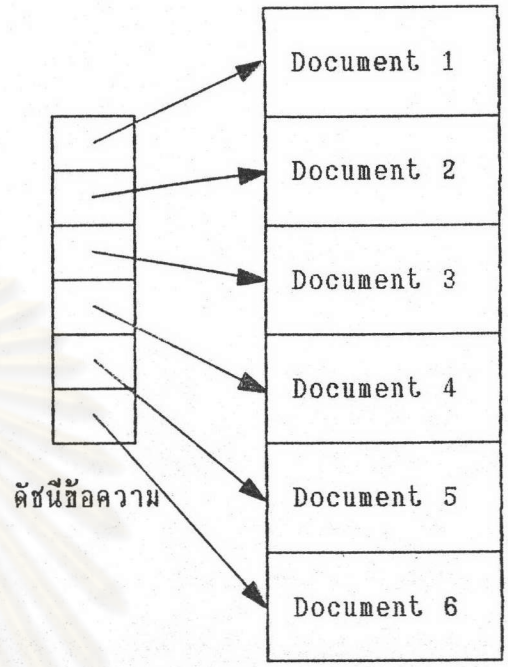
1) โปรแกรมจะทำการอ่านเอกสาร ในแฟ้มข้อมูลข้อความทีละคำ และทำการเก็บคำพร้อมตำแหน่งของคำ ที่ต้องการให้เป็นดัชนีไว้ในรายการ เพื่อสร้างเป็นรายการของดัชนีออกมา คำที่อยู่ในประเภท and, in, its, of, และ on จะไม่มีการเก็บไว้ในรายการ เนื่องจากเป็นคำที่ไม่มีนัยสำคัญ เราจะเรียกคำประเภทนี้ว่า คำหยุด (stop words)

เมื่อโปรแกรมทำการรู้จำเอกสารชุดใหม่ ก็จะเก็บตำแหน่งเริ่มต้นของเอกสารชุดใหม่ไว้ในดัชนีข้อความ (Text index) และด้วยวิธีการเช่นนี้ เมื่อโปรแกรมอ่านมาถึงคำสุดท้ายในเอกสารชุดสุดท้าย ก็จะได้เป็นรายการของคำทั้งหมด ที่ต้องการให้เป็นดัชนีพร้อมทั้งตำแหน่งของคำแต่ละคำ ที่ต้องการค้นหา

จากตัวอย่างข้างต้น จะได้รายการของคำ ดัชนีข้อความ และแฟ้มข้อมูลข้อความดังรูปที่ 2.2

คำ	หมายเลขเอกสาร
control	1
pollution	1
rivers	1
reduction	2
atmospheric	2
pollution	2
pollution	3
effect	3
rivers	3
effect	4
smoke	4
pollution	4
atmospheric	5
smoke	5
content	5
pollution	6
control	6
rivers	6

รายการของคำ



รูปที่ 2.2 รายการของคำ ดัชนีข้อความ และแฟ้มข้อมูลข้อความ

2) รายการของคำที่ได้ จากขั้นตอนที่ 1 ยังคงอยู่ในลำดับที่ค้นพบในเอกสารต้นฉบับ ในขั้นตอนที่ 2 โปรแกรม จะทำการเรียงลำดับคำตามตัวอักษร ได้เป็นรายการของคำที่เรียงลำดับเรียบร้อยแล้ว ดังรูปที่ 2.3

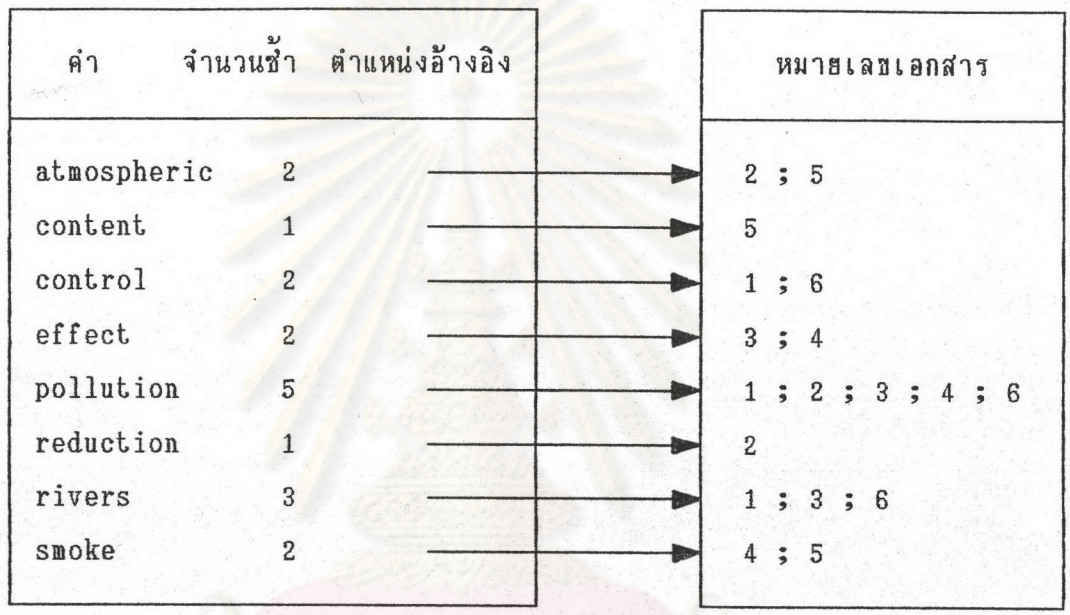
คำ	หมายเลขเอกสาร
atmospheric	2
atmospheric	5
content	5
control	1
control	6
effect	3
effect	4
pollution	1
pollution	2
pollution	3
pollution	4
pollution	6
reduction	2
rivers	1
rivers	3
rivers	6
smoke	4
smoke	5

รูปที่ 2.3 รายการของคำที่เรียงลำดับแล้ว

3) รายการของคำที่เรียงลำดับแล้ว จะบรรจุชุดของข้อมูล 2 ชุด คือชุดของข้อมูลที่เป็นตัวอักษร หรือคำที่เป็นดัชนี และชุดของข้อมูลที่เป็นตัวเลข หรือตำแหน่งของดัชนี การจัดเก็บข้อมูลในคอมพิวเตอร์ ที่มีประสิทธิภาพ คือการจัดเก็บโดยแยกเป็น 2 แฟ้มข้อมูล แฟ้มข้อมูลหนึ่ง สำหรับเก็บข้อมูลที่เป็นตัวอักษร โดยทำการตัดทอนคำที่ซ้ำกัน ให้เหลือเพียงคำเดียว เรียกว่า แฟ้มข้อมูลดิคชันนารี (Dictionary file) และอีกแฟ้มข้อมูลหนึ่ง สำหรับเก็บข้อมูลที่เป็นตัวเลข เรียกว่า แฟ้มข้อมูลผกผัน (Inverted file)

โดยโปรแกรม จะอ่านรายการของคำ ที่เรียงลำดับแล้ว และบันทึกลงในแฟ้มข้อมูลดิคชันนารี พร้อมกับบันทึกจำนวนครั้งของคำที่เกิดขึ้น ลงในตำแหน่งที่ติดกับคำ และบันทึกรายการ ของตำแหน่งของแต่ละคำ ลงในแฟ้มข้อมูลผกผัน ดังรูปที่ 2.4

ถ้าในฐานข้อมูลมีคำเกิดขึ้น n ครั้ง รายการตำแหน่งของคำในแฟ้มข้อมูลผกผันจะยาว n รายการด้วย



รูปที่ 2.4 แฟ้มข้อมูลดิคชันนารี และ แฟ้มข้อมูลผกผัน

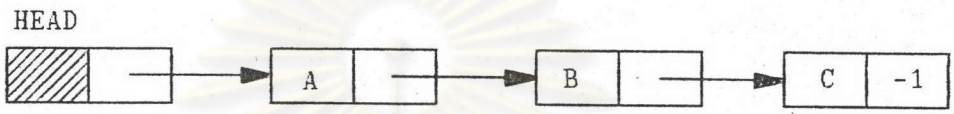
ในตอนสุดท้าย ของขั้นตอนที่ 3 จะได้ระบบแฟ้มข้อมูลผกผันที่สมบูรณ์แบบ โดยคำจะถูกผกผัน จากลักษณะการทำงานแบบธรรมชาติของมนุษย์ ที่อยู่ในแฟ้มข้อมูลข้อความ ไปอยู่ในรูปของคำที่เรียงลำดับแล้ว ในแฟ้มข้อมูลดิคชันนารี โดยตำแหน่งของแต่ละคำ จะถูกบันทึกไว้ในแฟ้มข้อมูลผกผัน และตำแหน่งเริ่มต้นของแต่ละเอกสาร จะถูกเก็บไว้ในดัชนีข้อความ

ในการค้นคืนสารสนเทศ ข้อมูลที่ต้องการจะถูกค้นหา ในแฟ้มข้อมูลดิคชันนารี และตัวชี้ของข้อมูลแต่ละตัว ในแฟ้มข้อมูลดิคชันนารี จะบอกตำแหน่งของรายการ ของหมายเลขเอกสารในแฟ้มข้อมูลผกผัน จากนั้นเอกสารที่มีหมายเลข สอดคล้องกับรายการในแฟ้มข้อมูลผกผันก็จะถูกอ่านออกมา

3. โครงสร้างข้อมูล

3.1 โครงสร้างข้อมูลแบบลิงค์ลิส (Link List)

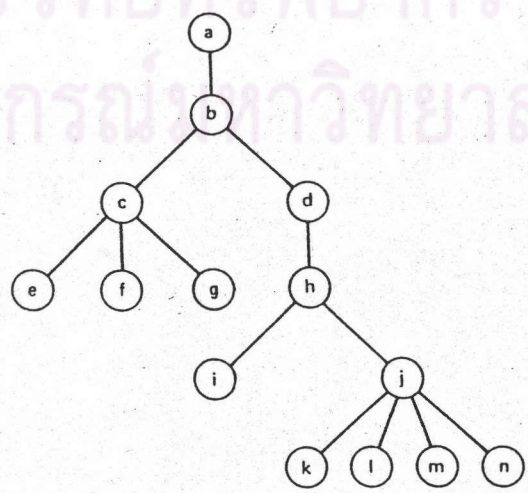
ลิงค์ลิสเป็นโครงสร้างข้อมูลแบบเชิงเส้น ประกอบด้วยโหนด (node) โหนดแรกของลิงค์ลิสจะเรียกว่า โหนดหัว (head node) ภายในโหนดประกอบด้วย เขตข้อมูลที่เก็บสารสนเทศ และเขตข้อมูลที่เก็บตัวชี้ของโหนดถัดไปที่สัมพันธ์กัน เรียกว่า ลิงค์ (link) ตัวชี้ของโหนดสุดท้ายจะว่างเปล่า (null) ซึ่งแทนด้วย -1 (Baron and Shapiro, 1980)



รูปที่ 2.5 แสดงโครงสร้างข้อมูลแบบลิงค์ลิส

3.2 โครงสร้างข้อมูลแบบทรี (Tree)

ทรีเป็นโครงสร้างข้อมูลแบบลำดับชั้น ประกอบด้วยโหนด โหนดแรกหรือโหนดที่อยู่ระดับบนสุด เรียกว่า โหนดราก (root node) ภายในโหนดประกอบด้วย เขตข้อมูลที่เก็บคีย์ (key) และเขตข้อมูลที่เก็บตัวชี้ (pointer) ที่ชี้ไปยังโหนดในระดับถัดไป ที่สัมพันธ์กัน ทรีย่อยของโหนด ที่มีโครงสร้างเป็นทรี เรียกว่า สับทรี (subtree) ทุกๆ โหนดของสับทรีเป็นโหนดที่อยู่ในลำดับถัดไป จากโหนดรากของสับทรีนั้น และโหนดรากของสับทรี จะเป็นโหนดที่อยู่เหนือโหนดทุกโหนดในสับทรี ทุกๆ โหนดในทรี จะเป็นโหนดแม่ (parent node) ของโหนดลูก (child node) ที่อยู่ในลำดับชั้นถัดไป ที่เชื่อมด้วยตัวชี้จากโหนดแม่ โหนดที่ไม่มีโหนดลูกจะเรียกว่า โหนดใบ (leaf node) (Baron and Shapiro, 1980)

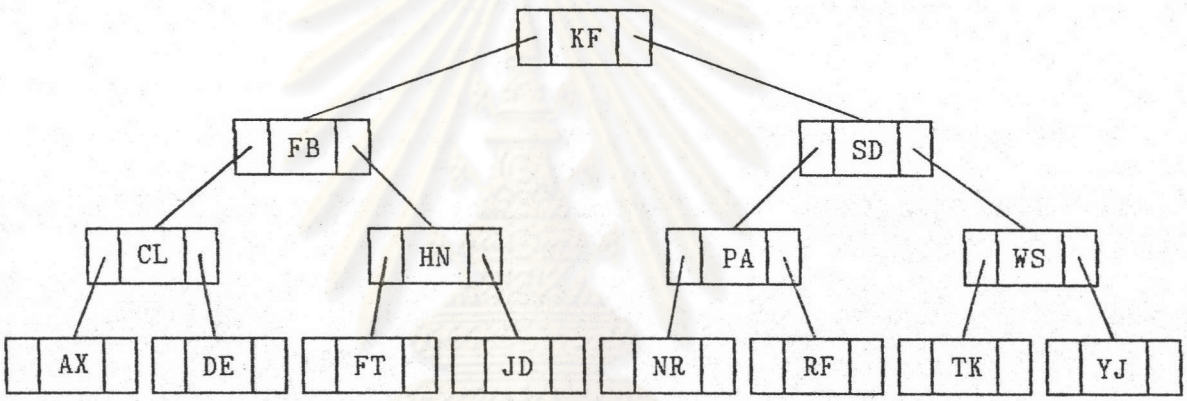


รูปที่ 2.6 แสดงโครงสร้างข้อมูลแบบทรี

จากรูปที่ 2.6 โหนด a เป็นโหนดรากของทรี โหนด b เป็นโหนดลูกของโหนด a และเป็นโหนดแม่ ของโหนด c และ d โหนด c เป็นโหนดรากของสับทรีที่ประกอบด้วยโหนด e, f และ g โหนด e, f, g, i, k, l, m และ n เรียกว่าโหนดใบ

3.2.1 โครงสร้างข้อมูลแบบไบนารีทรี (Binary Tree)

ไบนารีทรีเป็นทรีที่แต่ละโหนดมี 2 สับทรี คือ สับทรีทางซ้าย (left subtree) และสับทรีทางขวา (right subtree) ไบนารีทรีที่ค่าคีย์ในสับทรีทางซ้าย มีค่าน้อยกว่าค่าคีย์ในโหนดราก และค่าคีย์ในสับทรีทางขวามีค่ามากกว่าค่าคีย์ในโหนดราก เรียกว่าไบนารีเสิร์ชทรี (binary search tree) (Baron and Shapiro, 1980)



รูปที่ 2.7 แสดงโครงสร้างข้อมูลแบบไบนารีเสิร์ชทรี

การค้นหาข้อมูลในไบนารีเสิร์ชทรี จะเริ่มต้นจากโหนดราก ทำการเปรียบเทียบค่าคีย์ที่ต้องการค้นหา กับค่าคีย์ในโหนดราก ถ้าค่าคีย์ทั้งสองเป็นค่าเดียวกันก็แสดงว่าค้นพบข้อมูลที่ต้องการ ถ้าค่าคีย์ทั้งสองไม่ตรงกัน จะทำการค้นหาต่อไป โดยขึ้นอยู่กับ ค่าคีย์ที่ต้องการค้นหา ถ้าค่าคีย์ที่ต้องการค้นหามีค่าน้อยกว่าค่าคีย์ในโหนดราก ก็จะค้นหาลงไปในสับทรีทางซ้าย ถ้าค่าคีย์ที่ต้องการค้นหามีค่ามากกว่าค่าคีย์ในโหนดราก ก็จะค้นหาลงไปในสับทรีทางขวา

ถ้าแต่ละโหนด ถูกแทนด้วยระเบียนที่มีความยาวคงที่ เขตข้อมูลที่เป็นตัวชี้ จะเก็บหมายเลขระเบียนของโหนดที่สัมพันธ์กัน ดังนั้น เราสามารถแทนโครงสร้างข้อมูลแบบทรีลงบนหน่วยความจำสำรองได้ รูปที่ 2.8 แสดงข้อมูลของระเบียน 15 ระเบียน ซึ่งแทนไบนารีทรี ในรูป 2.7 จะเห็นว่า เขตข้อมูลตัวชี้จำนวนมากกว่าครึ่งหนึ่ง ในแฟ้มข้อมูลว่างเปล่าเนื่องจากเป็นโหนดใบที่ไม่มีโหนดลูก ในทางปฏิบัติเราจะใช้สัญลักษณ์พิเศษ เช่น -1 เพื่อเป็นตัวชี้ว่า การค้นหาข้อมูลในทรี ไปถึงระดับของโหนดใบ ซึ่งแสดงว่าสิ้นสุดการค้นหาข้อมูล

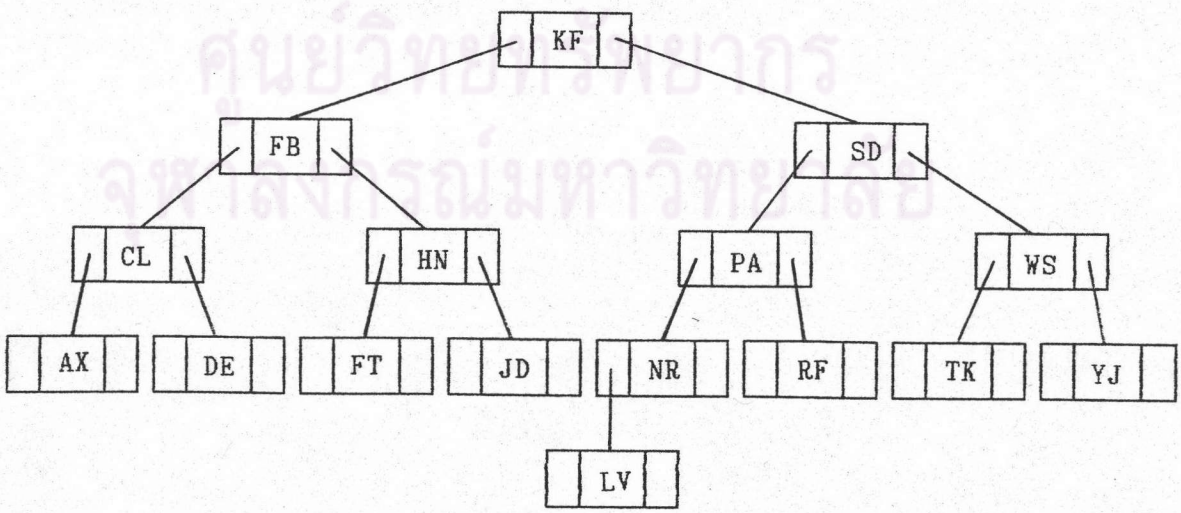
ROOT → 9

	Key	Left child	Right child
0	FB	10	8
1	JD		
2	RF		
3	SD	6	13
4	AX		
5	YJ		
6	PA	11	2
7	FT		

	Key	Left child	Right child
8	HN	7	1
9	KF	0	3
10	CL	4	12
11	NR		
12	DE		
13	WS	14	5
14	TK		

รูปที่ 2.8 โครงสร้างข้อมูลแบบไบนารีทรีที่มีการเก็บในหน่วยความจำสำรอง

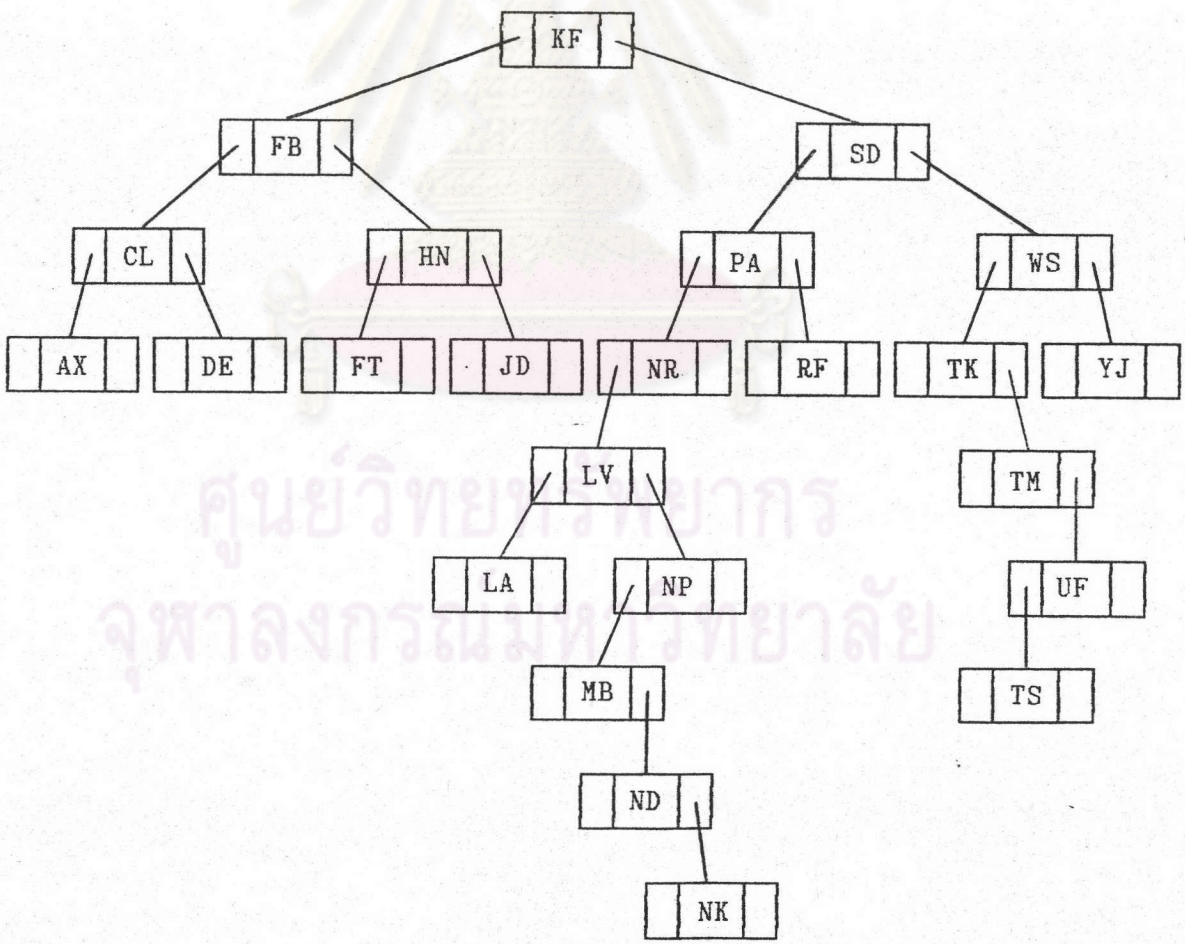
แต่เมื่อพิจารณาถึงค่าใช้จ่ายและประโยชน์ที่ได้จากการใช้โครงสร้างข้อมูลแบบทรี เราไม่ต้องเรียงลำดับข้อมูลในแฟ้ม เหมือนอย่างการค้นหาแบบทวิภาค รูปที่ 2.8 ระเบียนในแฟ้มข้อมูลแบบทรี จะอยู่ในลักษณะสุ่ม ไม่เรียงลำดับ ดังนั้น เมื่อมีการเพิ่มข้อมูลใหม่ เช่น LV ลงไปในแฟ้มข้อมูล เราเพียงแต่เปลี่ยนค่าของตัวชี้ ให้ชี้ไปยังโหนด ที่เหมาะสม ดังรูปที่ 2.9



รูปที่ 2.9 ไบนารีเสิร์ชทรีหลังจากเพิ่มคีย์ LV

การค้นหาข้อมูลในทรี ในรูปที่ 2.9 จะยังคงทำได้ดี เนื่องจากทรีอยู่ในสภาพที่สมดุล ซึ่งสมดุลหมายถึง ความสูงของเส้นทางที่สั้นที่สุด จากโหนดรากไปยังโหนดใบแตกต่างจากเส้นทางที่ยาวที่สุด ไม่เกิน 1 ระดับ พิจารณาต่อไปว่าจะเกิดอะไรขึ้น ถ้าเรายังคงเพิ่มคีย์ 8 คีย์ ต่อไปนี้ คือ NP MB TM LA UP ND TS NK เข้าไปในทรีในตำแหน่งที่ถูกต้อง ตามลำดับที่ปรากฏ ผลลัพธ์ที่ได้แสดงในรูปที่ 2.10

จะเห็นว่าการเพิ่มคีย์ลงไป ทำให้ทรีไม่สมดุล ความยาวของเส้นทางในการค้นหาแต่ละทางไม่เท่ากัน ซึ่งเป็นสิ่งที่ไม่พึงปรารถนาในการค้นหาข้อมูลในทรี และโดยเฉพาะความยุ่งยากที่จะเกิดขึ้น เมื่อมีการเก็บโหนดของทรีในหน่วยความจำสำรอง ซึ่งจะทำให้การค้นหาต้องทำการเข้าถึงแฟ้มข้อมูล ถึง 7, 8, หรือ 9 ครั้ง ในขณะที่ การค้นหาแบบทวิภาคกับข้อมูลที่เรียงลำดับ 24 คีย์ จะใช้เวลาในการเข้าถึงแฟ้มข้อมูลเพียง 5 ครั้ง



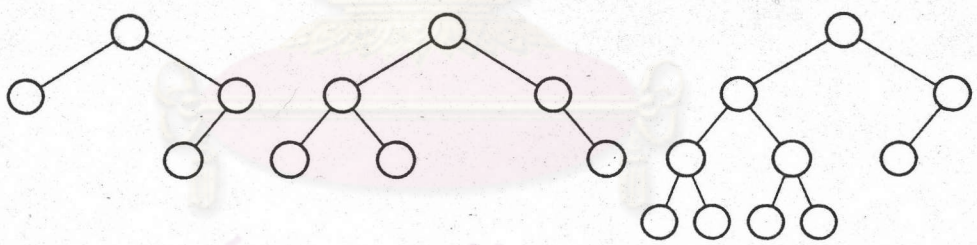
รูปที่ 2.10 ไบนารีเสิร์ชทรีหลังจากเพิ่มคีย์ใหม่ 8 คีย์

ถึงแม้ว่า การใช้โครงสร้างข้อมูลแบบทรี จะทำให้เราสามารถหลีกเลี่ยงการเรียงลำดับข้อมูลได้ แต่เราต้องเสียเวลามากในการค้นหาข้อมูลบางตัว สำหรับทรีที่มีคีย์ 100 คีย์และเป็นทรีที่ไม่สมดุล เราอาจต้องใช้เวลาในการค้นหาถึง 30, 40 ครั้ง หรือมากกว่านี้ ซึ่งเป็นค่าใช้จ่ายที่สูงมาก

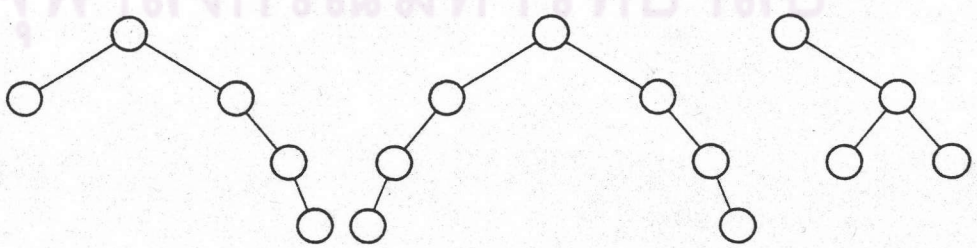
3.2.2 โครงสร้างข้อมูลแบบ AVL ทรี (AVL Tree)

เพื่อแก้ปัญหา เรื่องความไม่สมดุล ของไบนารีทรี ได้มีนักคณิตศาสตร์ชาวรัสเซีย 2 คน คือ G.M Adel'son-Vel'skii และ E.M.Landis ได้คิดค้นทรีแบบใหม่ขึ้นเรียกว่า AVL Tree ซึ่งเป็นทรีที่มีความสูงสมดุล ซึ่งหมายถึง ทรีที่มีข้อจำกัดเกี่ยวกับความแตกต่างระหว่างความสูงของ 2 สับทรีใดๆ ที่มีโหนดรากเดียวกัน ใน AVL Tree อนุญาตให้มีความแตกต่างสูงสุดเพียง 1 ระดับ และ AVL Tree นี้จะเรียกว่า ทรีที่มีความสูงสมดุล 1 (height-balanced 1-tree หรือ HB(1)Tree) AVL Tree ทั่วๆไป คือ HB(k) Trees ซึ่งอนุญาตให้สมดุล k ระดับ (Folk and Zoellick, 1987)

ทรีที่แสดงในรูป 2.11 เป็น AVL หรือ HB(1) เนื่องจากไม่มีสับทรี 2 สับทรี ของโหนดรากใดเลยที่มีความแตกต่างกันเกิน 1 ระดับ ทรีในรูป 2.12 ไม่เป็น AVL



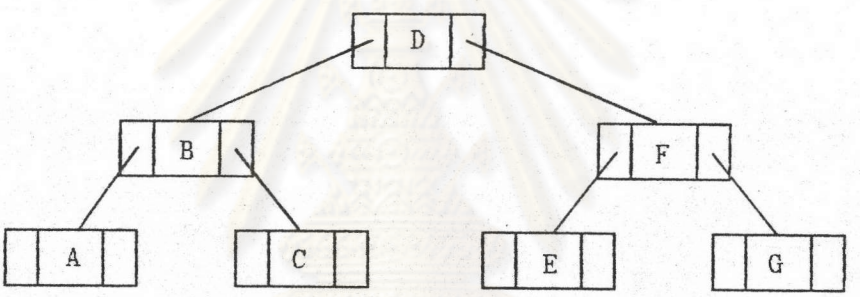
รูปที่ 2.11 แสดง AVL Tree ที่มีความสูงสมดุล 1



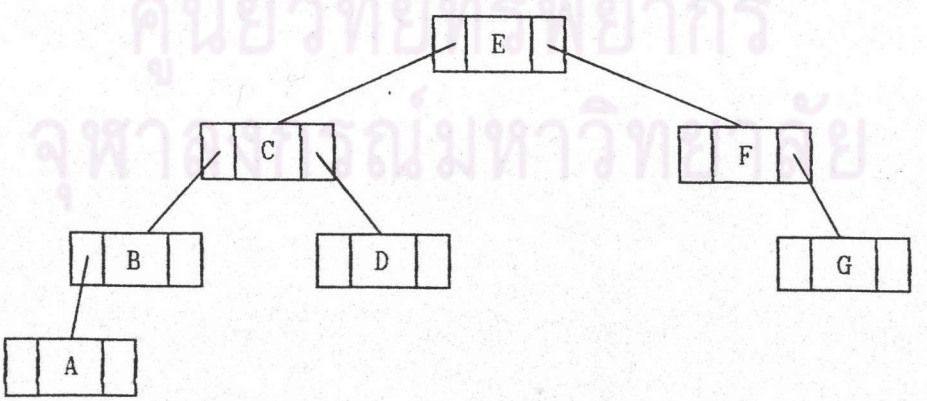
รูปที่ 2.12 แสดงทรีที่ไม่เป็น AVL Tree

ปัญหาทั่วๆ ไปที่เกี่ยวกับการเข้าถึงแฟ้มข้อมูล และการดูแลดัชนี ที่มีขนาดใหญ่เกินกว่าที่จะจัดเก็บ ในหน่วยความจำหลักได้หมด AVL Tree จะเป็นโครงสร้างข้อมูลที่เหมาะสม เนื่องจากเราสามารถกำหนดกระบวนการในการทำงาน เพื่อดูแลเรื่องความสมดุลของความสูงได้ AVL Tree ที่มีความสูงสมดุล สามารถทำการค้นหาข้อมูลได้ดีพอๆ กับทรีที่มีความสูงสมดุลแบบสมบูรณ์ (completely balanced tree) ซึ่งทรีที่มีความสูงสมดุลแบบสมบูรณ์ หมายถึง ทรีที่เส้นทางจากโหนดราก ไปยังโหนดใบทุกโหนดมีความยาวเท่ากัน

ตัวอย่าง ทรีที่มีความสูงสมดุลแบบสมบูรณ์ ที่สร้างจากคีย์ต่อไปนี้ คือ B C G E F D A แสดงในรูป 2.13 และ AVL Tree ที่สร้างจากคีย์ชุดเดียวกัน และรับเข้ามาในลำดับเดียวกัน แสดงในรูปที่ 2.14



รูปที่ 2.13 แสดงทรีที่มีความสูงสมดุลแบบสมบูรณ์



รูปที่ 2.14 แสดง AVL Tree

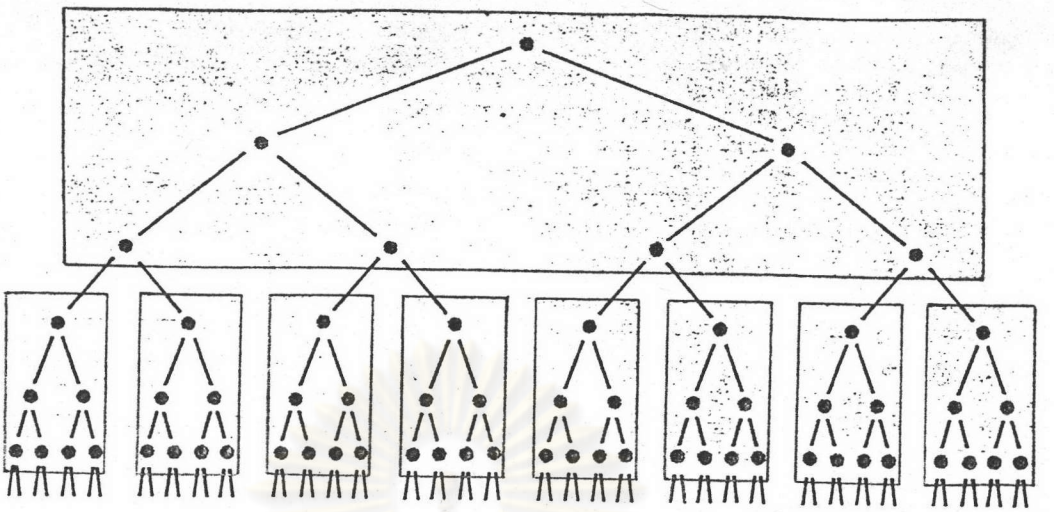
สำหรับทรีที่มีความสูงสมดุลแบบสมบูรณ์ กรณีที่แย่ที่สุดในการค้นหาคีย์ N คีย์ คือต้องค้นหา $\log_2(N+1)$ ระดับ และสำหรับ AVL Tree กรณีที่แย่ที่สุดในการค้นหาคีย์ คือต้องค้นหา $1.44\log_2(N+2)$ ระดับ ดังนั้นถ้ากำหนดให้คีย์ 1,000,000 คีย์ เก็บในทรีที่มีความสูงสมดุลแบบสมบูรณ์ จะต้องใช้เวลาในการค้นหา 20 ระดับ สำหรับบางคีย์ แต่ไม่เกิน 21 ระดับ ถ้าเป็น AVL Tree จำนวนระดับสูงสุดจะเป็น 28 ระดับเท่านั้น

ขั้นตอนในการทำ AVL Tree ซึ่งต้องมีการจัดเรียงลำดับองค์กรของทรีใหม่ ต้องทำการกำหนดตัวชี้ใหม่ ไม่เกิน 5 ตัวชี้ ดังนั้น จะเห็นได้ว่าทรีที่มีความสูงสมดุลสามารถแก้ปัญหา เรื่องการเก็บดัชนีในลักษณะเรียงลำดับได้

3.2.3 โครงสร้างข้อมูลแบบเพจไบนารีทรี (Paged Binary Tree)

เมื่อพูดถึง อุปกรณ์เก็บข้อมูลภายนอก หรือ หน่วยความจำสำรอง (secondary storage) ลักษณะสำคัญประการหนึ่งคือ เวลาในการค้นหาตำแหน่งของข้อมูลจะใช้เวลานาน แต่ถ้าหัวอ่าน (read head) อยู่ในตำแหน่งที่พร้อมอยู่แล้ว การอ่านหรือการเขียนข้อมูลในตำแหน่งที่ติดกัน จะทำได้รวดเร็ว ดังนั้นเมื่อรวมลักษณะการเข้าถึงข้อมูลที่ทำได้ช้าและการถ่ายข้อมูลที่ทำได้เร็วเข้าด้วยกัน ก็จะเป็นลักษณะของการทำเพจกิ้ง (paging) (Folk and Zoellick, 1987)

ในระบบเพจกิ้ง ในการเข้าถึงข้อมูล 2-3 ไบท์ เราไม่ต้องเสียค่าใช้จ่ายสำหรับการเข้าถึงมาก เนื่องจากในการอ่าน เราจะอ่านมาจากแฟ้มข้อมูลทีละเพจ ซึ่งเพจจะประกอบด้วยระเบียนของข้อมูลจำนวนมาก และถ้าบิทต่อไปที่เราต้องการ อยู่ในเพจที่เราเพิ่งจะอ่านเข้ามา เราก็สามารถลดค่าใช้จ่ายในการเข้าถึงแฟ้มข้อมูลลงได้ ดังนั้นเพจกิ้งจึงสามารถแก้ปัญหาเรื่องการค้นหาข้อมูลให้เราได้ โดยการแบ่งไบนารีทรีเป็นเพจ และเก็บแต่ละเพจลงในบล็อค ที่ติดต่อกันบนจานแม่เหล็ก เราจะสามารถลดจำนวนครั้งของการเข้าถึง เพื่อการค้นหาใดๆ ลงได้ รูป 2.15 แสดงโครงสร้างข้อมูลแบบเพจไบนารีทรี ในทรีนี้เราสามารถกำหนดตำแหน่งของโหนด 63 โหนดใดๆ ในทรี โดยทำการเข้าถึงจานแม่เหล็กไม่เกิน 2 ครั้ง จะเห็นว่าทุกเพจเก็บโหนด 7 โหนด และตัวชี้ที่ชี้ไปยังเพจใหม่ 8 เพจ ถ้าเราเพิ่มระดับเพจของทรีขึ้นอีก 1 ระดับ เราจะเพิ่มเพจใหม่ได้ 64 เพจ ดังนั้นเราจะสามารถค้นหาคีย์ใดๆ ของโหนด 511 โหนดได้ โดยการเข้าถึงแฟ้มข้อมูลเพียง 3 ครั้ง ถ้าเพิ่มระดับของเพจขึ้นอีก 1 ระดับเราจะค้นหาคีย์ใดๆ ของโหนด 4095 โหนด โดยการเข้าถึงเพียง 4 ครั้ง ในขณะที่การค้นหาแบบทวิภาค กับข้อมูลที่มี 4095 รายการ จะต้องทำการเข้าถึงแฟ้มข้อมูลมากที่สุด 12 ครั้ง ดังนั้น จึงเป็นที่ชัดเจนแล้วว่าการแบ่งทรีออกเป็นเพจ จะสามารถค้นหาข้อมูลบนหน่วยความจำสำรอง ได้เร็วกว่าวิธีการอื่นๆ ที่เรากล่าวมาแล้ว



รูป 2.15 แสดงโครงสร้างข้อมูลแบบเพจไบนารีทรี

จำนวนครั้งของการเข้าถึงแฟ้มข้อมูลในกรณีที่แย่ที่สุดสำหรับการค้นหาข้อมูลในทรี ที่มีความสมดุลแบบสมบูรณ์ จะเป็น

$\log_2(N+1)$ โดยที่ N เป็นจำนวนคีย์ในทรี

จำนวนครั้งของการเข้าถึงแฟ้มข้อมูลที่ต้องใช้สำหรับ เพจทรีที่มีความสูงสมดุลแบบสมบูรณ์ จะเป็น

$\log_{k+1}(N+1)$ โดยที่ N เป็นจำนวนคีย์ทั้งหมด
และ k คือจำนวนคีย์ใน 1 เพจ

สูตรที่ 2 จะครอบคลุมสูตรที่ 1 ด้วย เพราะจำนวนคีย์ในเพจของไบนารีทรี เป็น 1

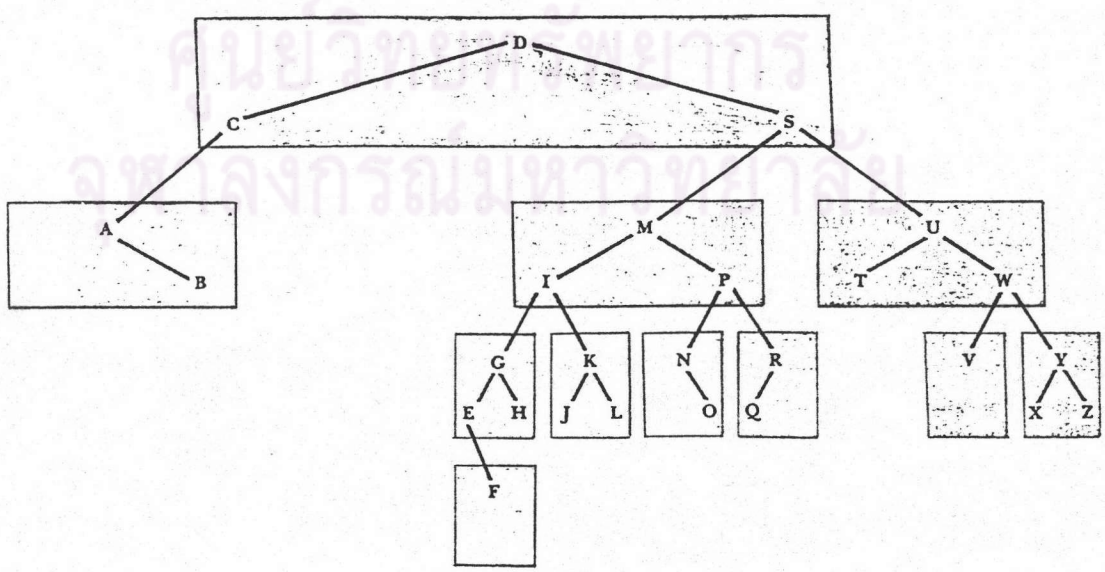
การใช้เพจขนาดใหญ่ไม่ได้ใช้โดยอิสระ เพราะทุกครั้งที่มีการเข้าถึงเพจ ต้องทำการส่งข้อมูลจำนวนมาก ซึ่งข้อมูลส่วนใหญ่ไม่ได้ใช้ เวลาที่ใช้ในการส่งข้อมูลมีค่ามาก แต่อย่างไรก็ตาม ก็สามารถประหยัดเวลา ในการเข้าถึงแฟ้มข้อมูลได้มาก ซึ่งมากกว่าเวลาที่ใช้สำหรับการส่งข้อมูล

3.2.3.1 ปัญหาในการสร้างเพจทรีจากบนลงล่าง

การแบ่งทรีออกเป็นเพจเป็นวิธีการที่เหมาะสมเมื่อเทียบกับลักษณะทางกายภาพ ของอุปกรณ์หน่วยความจำสำรอง เช่น ดิสค์ ปัญหาเมื่อเราตัดสินใจใช้เพจทรีก็คือ เราจะสร้างมันอย่างไร สมมติว่าเรามีชุดของคีย์ทั้งหมดอยู่ วิธีการง่ายๆ ก็คือ เราทำการเรียงลำดับชุดของคีย์ และสร้างทรีจากชุดของคีย์ที่เรียงลำดับแล้ว ถ้าเราเริ่มต้นสร้างทรีจาก โหนดราก เราจะรู้ว่าคีย์ที่อยู่ตรงกลาง ในชุดของคีย์ที่เรียงลำดับแล้ว จะต้องเป็นคีย์ในเพจรากของทรี หรือพูดสั้นๆ ว่า เรารู้ว่าเราจะเริ่มต้นตรงไหน และแน่ใจได้ว่า จุดเริ่มต้นจะแบ่งชุดของคีย์ ให้อยู่ในลักษณะที่สมดุล แต่ในทางปฏิบัติจริงๆ ปัญหามีความซับซ้อนมากกว่านี้ นั่นคือถ้าเราได้รับคีย์ในลักษณะสุ่ม และทำการเพิ่มลงไปในทรีทันที ที่ได้รับคีย์มา สมมติว่าเราต้องการสร้างเพจไบนารีทรี ที่บรรจุค่าคีย์สูงสุด เพจละ 3 คีย์ ตามลำดับของคีย์ที่เป็นตัวอักษรต่อไปนี้

C D S T A M P I B W N G U R K E H O L J Y Q Z F X V

เมื่อเราเพิ่ม หรือแทรกคีย์ เราต้องทำการหมุนภายในเพจ เพื่อให้แต่ละเพจอยู่ในลักษณะที่สมดุลที่สุดเท่าที่จะทำได้ ทรีที่ได้แสดงในรูป 2.16 ถึงแม้ทรีที่ได้จะไม่มีอะไรผิดลักษณะ แต่การสร้างเพจไบนารีทรีจากบนลงล่าง จะมีความยุ่งยากมาก เพราะเมื่อเราเริ่มต้นสร้างทรีจากราก คีย์เริ่มต้นจึงจำเป็นต้องเป็นราก จากตัวอย่างข้างต้นมีคีย์อย่างน้อย 2 คีย์ คือ C และ D ซึ่งเป็นคีย์ที่เราไม่ต้องการให้อยู่ตรงตำแหน่งนั้น แต่เนื่องจากมันเป็นคีย์แรก และอยู่ในลำดับที่ติดกัน จึงต้องนำไปเป็นคีย์เริ่มต้นของคีย์ตัวอื่นๆ ผลที่เกิดขึ้น จึงทำให้ทรีไม่สมดุล



รูปที่ 2.16 เพจไบนารีทรีที่สร้างจากคีย์ที่รับเข้ามาในลักษณะสุ่ม

จากการที่คีย์ที่ไม่เหมาะสม ไปอยู่ในตำแหน่งรากของทรี และเราไม่สามารถหมุนเพจทั้งหมดของทรีแบบง่ายๆ เช่นที่เราหมุนคีย์เดี่ยวๆ ในทรีทั่วๆ ไปได้ ถ้าเราทำการหมุนทรี เพจรากเริ่มต้น จะเลื่อนลงไปทางซ้าย เลื่อนคีย์ C และ D ลงไปในตำแหน่งที่ดีกว่า แต่จะทำให้คีย์ออกนอกเพจ ดังนั้น เราจึงต้องทำการกระจายเพจ ซึ่งหมายถึงการจัดการเพื่อสร้างเพจใหม่ ซึ่งต้องทำทั้งการสร้างสมดุลภายในเพจ และการจัดการเพจอื่นๆ ที่เกี่ยวข้องให้อยู่ในลักษณะที่เหมาะสมด้วย ขบวนการในการสร้างเพจใหม่ ที่มีผลเฉพาะภายในเพจ ทำได้ยาก เนื่องจาก ในการจัดการเพจใหม่ และปรับทรีให้เหมาะสม จะกระทบไปยังส่วนอื่นๆ ของทรีเป็นส่วนมาก และการกระทำเช่นนี้จะยิ่งยุ่งยากมากขึ้น เมื่อเพจมีขนาดใหญ่ขึ้น

แม้ว่าเราจะตัดสินใจแล้วว่า แนวความคิด ในการจัดเก็บคีย์เป็นเพจจะเป็นการจัดเก็บที่ดีมาก ในการที่จะลดเวลาในการเข้าถึงแฟ้มข้อมูล ให้น้อยลง เราก็ยังต้องประสบกับปัญหาอีก 3 ประการคือ

1) เราจะแน่ใจได้อย่างไรว่า คีย์ที่อยู่ในโหนดรากจะเป็นตัวแบ่งคีย์ที่ดี และชุดของคีย์ที่ได้จากการแบ่งจะเท่าเทียมกัน

2) เราจะหลีกเลี่ยงกลุ่มของคีย์บางคีย์ เช่น C, D, และ S ในตัวอย่างของเราได้อย่างไร ที่จะไม่ให้ใช้เพจร่วมกัน

3) เราจะแน่ใจได้อย่างไรว่า เพจที่มีขนาดใหญ่ เช่น เพจที่บรรจุข้อมูลเพจละ 8191 คีย์ จะไม่บรรจุชุดของคีย์เพียงจำนวนน้อย เช่น 20-30 คีย์

3.2.4 โครงสร้างข้อมูลแบบบีทรี (B-Tree)

บีทรีเป็นทรีที่มีการสร้างจากล่างขึ้นบน แทนที่จะสร้างจากบนลงล่าง ในบีทรี เพจหรือโหนด จะบรรจุชุดของคีย์ที่เรียงลำดับ และชุดของตัวชี้ จำนวนของตัวชี้ จะมากกว่าจำนวนของคีย์อยู่ 1 เสมอ จำนวนตัวชี้สูงสุดที่สามารถเก็บได้ใน 1 โหนด จะเรียกว่า ลำดับ ของบีทรี ตัวอย่างเช่น บีทรีลำดับ 8 แต่ละเพจหรือโหนด จะสามารถเก็บคีย์ได้มากที่สุด 7 คีย์ และเก็บตัวชี้ได้มากที่สุด 8 ตัวชี้ (Folk and Zoellick,1987)

ในการประยุกต์ใช้งานจริงๆ จะมีสารสนเทศอื่นๆ ที่เก็บพร้อมกับคีย์ด้วยเช่น ค่าอ้างอิงไปยังระเบียนที่เก็บข้อมูล ที่สัมพันธ์กับคีย์นั้น

จุดประสงค์ของการสร้างบีทรี ก็เพื่อค้นหาข้อมูล โดยทำการเข้าถึงแฟ้มข้อมูลให้น้อยครั้งที่สุด ดังนั้น เราจึงพยายามสร้างทรี ที่มีความสูงน้อยที่สุดเท่าที่จะเป็นได้ ซึ่งเราสามารถทำได้ดังนี้

1) จะไม่มีสับทรีที่ว่างเปล่า ปรากฏอยู่เหนือโหนดใบ ดังนั้นการแบ่งคีย์ออกเป็นกลุ่มย่อย จะมีประสิทธิภาพมากที่สุด เท่าที่จะเป็นได้

2) ทุกๆ โหนดใบ จะอยู่ในระดับเดียวกัน ดังนั้นการค้นหาข้อมูลจึงรับประกันได้ว่า จะจบลงด้วยจำนวนครั้ง ของการเข้าถึงแฟ้มข้อมูลที่เท่ากัน

3) ทุกๆ โหนด ยกเว้นโหนดใบ จะมีโหนดลูกอย่างน้อยที่สุดครึ่งหนึ่งของโหนดลูก ที่สามารถมีได้มากที่สุด

ดังนั้น บีทรีที่มีลำดับ m จะมีลักษณะดังนี้

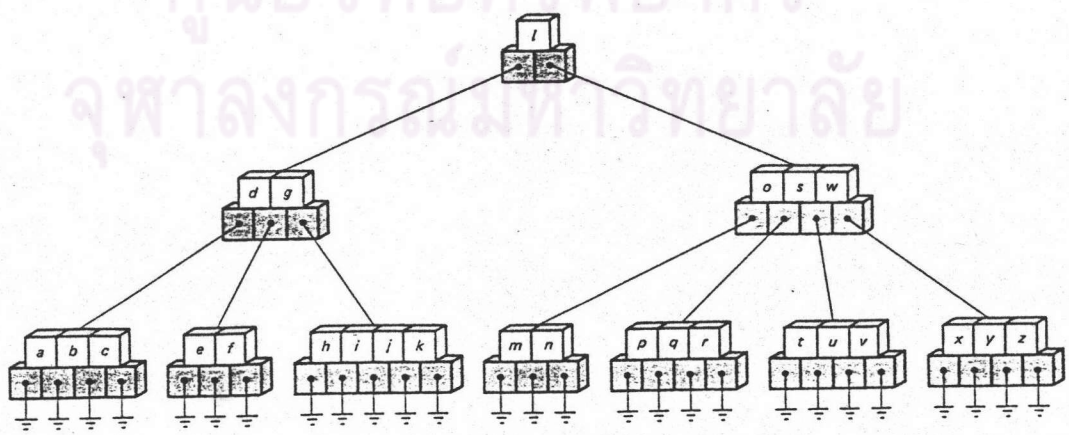
1) ทุกๆ โหนดใบจะอยู่ในระดับเดียวกัน

2) ทุกๆ โหนด ยกเว้นโหนดราก จะมีโหนดลูก ที่ไม่ว่างเปล่ามากที่สุด m โหนด และอย่างน้อยที่สุด $\lceil m/2 \rceil$ โหนด

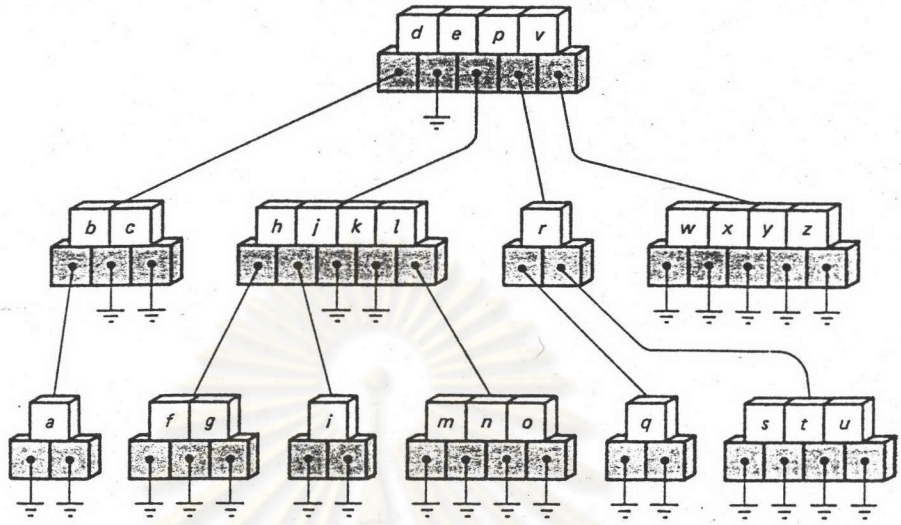
3) จำนวนของคีย์ในแต่ละโหนด จะมีจำนวนน้อยกว่าจำนวนโหนดลูกของมันอยู่ 1

4) โหนดราก จะมีโหนดลูกอย่างมากที่สุด m โหนด แต่อาจจะมีเพียง 2-3 โหนดก็ได้ ถ้ามันไม่ใช่โหนดใบ หรืออาจจะว่างเปล่าถ้าทรีประกอบด้วยโหนดรากเพียงโหนดเดียว

ทรีในรูป 2.17 แสดง บีทรีลำดับ 5 ซึ่งคีย์เป็นตัวอักษร 26 ตัว ทรีในรูป 2.18 เป็นทรีที่มีการค้นหาข้อมูลได้หลายทาง และมีจำนวนโหนดลูก สูงสุด 5 โหนด เราเรียกว่า ทรีที่มีการค้นหาข้อมูลได้ 5 ทาง (5-way search tree) แต่ไม่ใช่บีทรี เนื่องจาก มีโหนดที่มีโหนดลูกว่างเปล่า และโหนดใบไม่ได้อยู่ในระดับเดียวกัน



รูปที่ 2.17 แสดงโครงสร้างข้อมูลแบบบีทรีลำดับ 5



รูปที่ 2.18 แสดงทรีที่มีการค้นหาข้อมูลได้ 5 ทาง

3.2.4.1 <u>การเพิ่มข้อมูลเข้าไปในบีทรี</u>

การที่โหนดใบทุกโหนดในบีทรี อยู่ในระดับเดียวกัน ทำให้บีทรี มีลักษณะที่ดี การเจริญเติบโตของบีทรี จะตรงกันข้ามกับไบนารีทรี คือจะเจริญเติบโตที่โหนดราก แทนที่จะเจริญเติบโตที่โหนดใบ

การเพิ่มข้อมูลในบีทรี จะทำการค้นหาดูว่าคีย์ใหม่ที่จะเพิ่มเข้าไปอยู่ในบีทรีหรือยัง การค้นหาจะไปหยุดที่โหนดใบ ถ้าโหนดใบยังไม่เต็ม คีย์ใหม่จะถูกเพิ่มเข้าไปในโหนดใบ แต่ถ้าโหนดใบนั้นเต็มแล้ว ก็จะทำการแบ่ง โหนดออกเป็น 2 โหนด ที่อยู่ในระดับเดียวกัน ยกเว้นคีย์ที่อยู่ตรงกลาง จะไม่ใส่ลงไปในโหนดใหม่ทั้ง 2 แต่จะส่งขึ้นไปใส่ในโหนดแม่แทน เมื่อคีย์ถูกเพิ่มเข้าไปในโหนดรากที่เต็มแล้ว โหนดรากก็จะถูกแบ่งออกเป็น 2 โหนด และคีย์ที่อยู่ตรงกลาง ก็จะถูกส่งขึ้นไปเป็นโหนดรากตัวใหม่

ขบวนการดังกล่าว จะเห็นได้ชัดเจน ด้วยการศึกษาจากตัวอย่างการเจริญเติบโตของบีทรี ลำดับ 5 ที่แสดงในรูป 2.19 โดยเราจะเพิ่มคีย์ต่อไปนี้ลงในบีทรีที่ยังว่างเปล่า

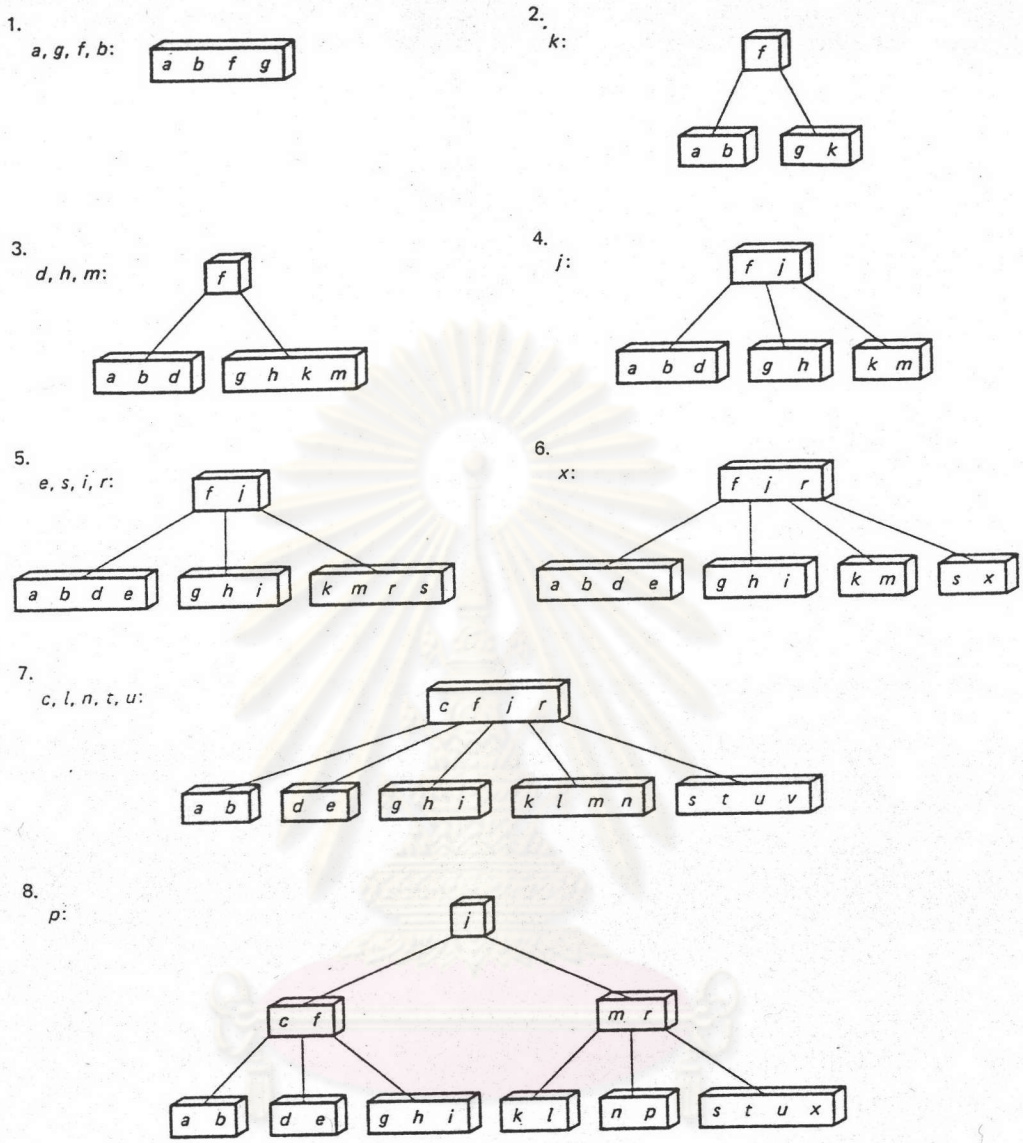
a g f b k d h m j e s i r x c l n t u p

4 คีย์แรกจะถูกใส่ลงในโหนด 1 โหนด ดังภาพแรกของรูปที่ 2.19 คีย์ทั้ง 4 จะถูกจัดให้เรียงอยู่ในลำดับที่ถูกต้อง ตามที่ถูกใส่เข้าไป โหนดแรกจะเต็ม และไม่มีที่ว่างเหลือ สำหรับคีย์ตัวที่ 5 คือ k ดังนั้นเมื่อคีย์ k ถูกเพิ่มเข้าไป จะทำให้โหนดถูกแบ่งออกเป็น 2 โหนด และคีย์ที่อยู่ตรงกลาง คือ f จะถูกย้ายขึ้นไป เป็นโหนดรากตัวใหม่ เนื่องจากในการแบ่งโหนด จะทำให้โหนดมีข้อมูลอยู่เพียงครึ่งโหนดเท่านั้น ดังนั้นคีย์ 3 คีย์ถัดไป สามารถ เพิ่มเข้าไปได้โดยไม่ยุ่งยาก หลังจากเพิ่มคีย์เข้าไปในโหนดแล้ว จะต้องจัดการคีย์ที่อยู่ในโหนดใหม่ ให้อยู่ในลำดับที่ถูกต้องด้วย การเพิ่มคีย์ตัวต่อไป คือ j จะทำให้โหนดถูกแบ่งอีกครั้ง และในครั้งนี้ j เป็นคีย์ที่อยู่ตรงกลาง และถูกย้ายขึ้นไปอยู่กับ คีย์ f ในโหนดราก การเพิ่มคีย์ในครั้งต่อๆ ไป ก็จะกระทำในลักษณะเดียวกัน การเพิ่มคีย์ตัวสุดท้ายคือ p น่าสนใจมาก เนื่องจากการเพิ่มคีย์ตัวนี้ การแบ่งครั้งแรก จะแบ่งโหนดที่บรรจุคีย์ k l m n และส่งคีย์ที่อยู่ตรงกลางคือ m ขึ้นไปใส่ในโหนดที่บรรจุคีย์ c f j r ซึ่งเต็มแล้ว ดังนั้นโหนดนี้จึงต้องแบ่ง และโหนดรากตัวใหม่จะถูกสร้างขึ้น เพื่อบรรจุคีย์ j

ข้อสังเกต 2 ประการเกี่ยวกับการเจริญเติบโตของบีทรี

1) เมื่อโหนดถูกแบ่ง ก็จะเกิดการสร้างโหนดใหม่ 2 โหนด ซึ่งจะมีคีย์อยู่เพียงครึ่งหนึ่งของโหนดเท่านั้น การเพิ่มคีย์ครั้งต่อมา จะทำในลักษณะที่คล้ายกัน โดยไม่ต้องแบ่งโหนดอีก ดังนั้นการแบ่งโหนด 1 ครั้ง จึงเป็นการเตรียมสำหรับการเพิ่มคีย์ใหม่อีก หลายๆ คีย์

2) คีย์ที่อยู่ตรงกลาง จะถูกส่งขึ้นไป ซึ่งไม่จำเป็นต้องเป็นคีย์ที่กำลังถูกเพิ่มเข้าไปใหม่ ดังนั้นการเพิ่มคีย์หลายๆ ครั้งทำให้บีทรีเกิดความสมดุล ไม่ว่าลำดับของคีย์ที่เข้ามาจะอยู่ในลำดับใดก็ตาม



รูปที่ 2.19 แสดงการเจริญเติบโตของบีทรี