

บทที่ 12

Abstract Data Type



วัตถุประสงค์

หลังจากเรียนจบบทที่ 12 แล้ว นักศึกษาต้องสามารถ:

- เข้าใจหลักการและแนวคิดของ abstract data type (ADT)
- เข้าใจหลักการและแนวคิดของ linear list พร้อมทั้งการดำเนินการและการประยุกต์
- เข้าใจหลักการและแนวคิดของ stack พร้อมทั้งการดำเนินการและการประยุกต์
- เข้าใจหลักการและแนวคิดของ queue พร้อมทั้งการดำเนินการและการประยุกต์
- เข้าใจหลักการและแนวคิดของ tree พร้อมทั้งการดำเนินการและการประยุกต์
- เข้าใจหลักการและแนวคิดของ graph พร้อมทั้งการดำเนินการและการประยุกต์

12.1

เหตุผลเบื้องหลังของ ADT



Abstract Data Type : ADT

- การเขียนโปรแกรมในยุคแรกๆของการใช้คอมพิวเตอร์ยังไม่มี การคิดค้น ADT ถ้าเราต้องการอ่านข้อมูลจากแฟ้มข้อมูลเราจะต้องเขียนคำสั่งด้วย ภาษาโปรแกรมคอมพิวเตอร์เพื่อที่จะอ่านข้อมูลจากสื่อที่เก็บข้อมูลโดยตรงซึ่งยุ่งยาก ต่อมานักวิทยาศาสตร์ทางคอมพิวเตอร์ได้พัฒนา ADT ขึ้น การมี ADT ทำให้เราสามารถเขียนคำสั่งเพื่ออ่านข้อมูลจากแฟ้มข้อมูลได้โดยสะดวก ซึ่งภาษาคอมพิวเตอร์ยุคใหม่ใช้ ADT เป็นหลัก คำสั่งรับข้อมูลเข้าจากคีย์บอร์ดก็ใช้ ADT ซึ่ง ADT มีทั้ง **โครงสร้างข้อมูล** **คุณลักษณะเฉพาะ** และ **กลุ่มของการกระทำ** (operations) ที่ใช้สำหรับอ่านโครงสร้างข้อมูลของ ADT เอง

Abstract Data Type : ADT (ต่อ)

- นอกจากนี้ ADT ยังมีกฎที่ยอมให้เราสามารถแปลงประเภทของโครงสร้างข้อมูลได้ด้วยเช่นจำนวนเต็มและสตริงเป็นต้น ด้วย ADT ผู้ใช้ไม่ต้องไปสนใจในรายละเอียดว่าการกระทำแต่ละอย่างทำอย่างไรโดยรายละเอียดจะถูกซ่อนไว้ ผู้ใช้ไม่จำเป็นต้องรู้ มีหน้าที่ใช้งานเท่านั้น กล่าวอีกอย่างหนึ่งคือ ADT ประกอบด้วยเซตของข้อกำหนดหรือนิยามที่ผู้เขียนโปรแกรมสามารถเรียกใช้ฟังก์ชันได้โดยปิดบังรายละเอียดการทำงานไว้ การทำงานในลักษณะดังกล่าวเป็นที่รู้จักกันในชื่อ “**abstraction**” นั่นคือผู้เขียนโปรแกรมคิดถึงประเด็นสำคัญว่าทำอะไรแต่ไม่ต้องลงรายละเอียดว่าทำอย่างไร



Note:

The concept of abstraction means:

- 1. You know what a data type can do.**
- 2. How it is done is hidden.**



นิยาม “Abstract Data Type”

- **นิยาม:** ADT คือการประกาศหรือการกำหนดชุดข้อมูลและการกระทำที่มีความหมายกับประเภทของข้อมูลนั้น ทั้งข้อมูลและการกระทำจะถูกมัดรวมและปิดบังไม่ให้ผู้ใช้เห็นรายละเอียดภายใน นั่นคือ ADT มีลักษณะสำคัญ 3 ประการคือ
 1. ประกาศประเภทของข้อมูล
 2. ประกาศการกระทำกับข้อมูล
 3. ผูกมัดรวมระหว่างข้อมูลกับการกระทำ



Note:

Abstract Data Type

- 1. Declaration of data**
- 2. Declaration of operations**
- 3. Encapsulation of data and operations**



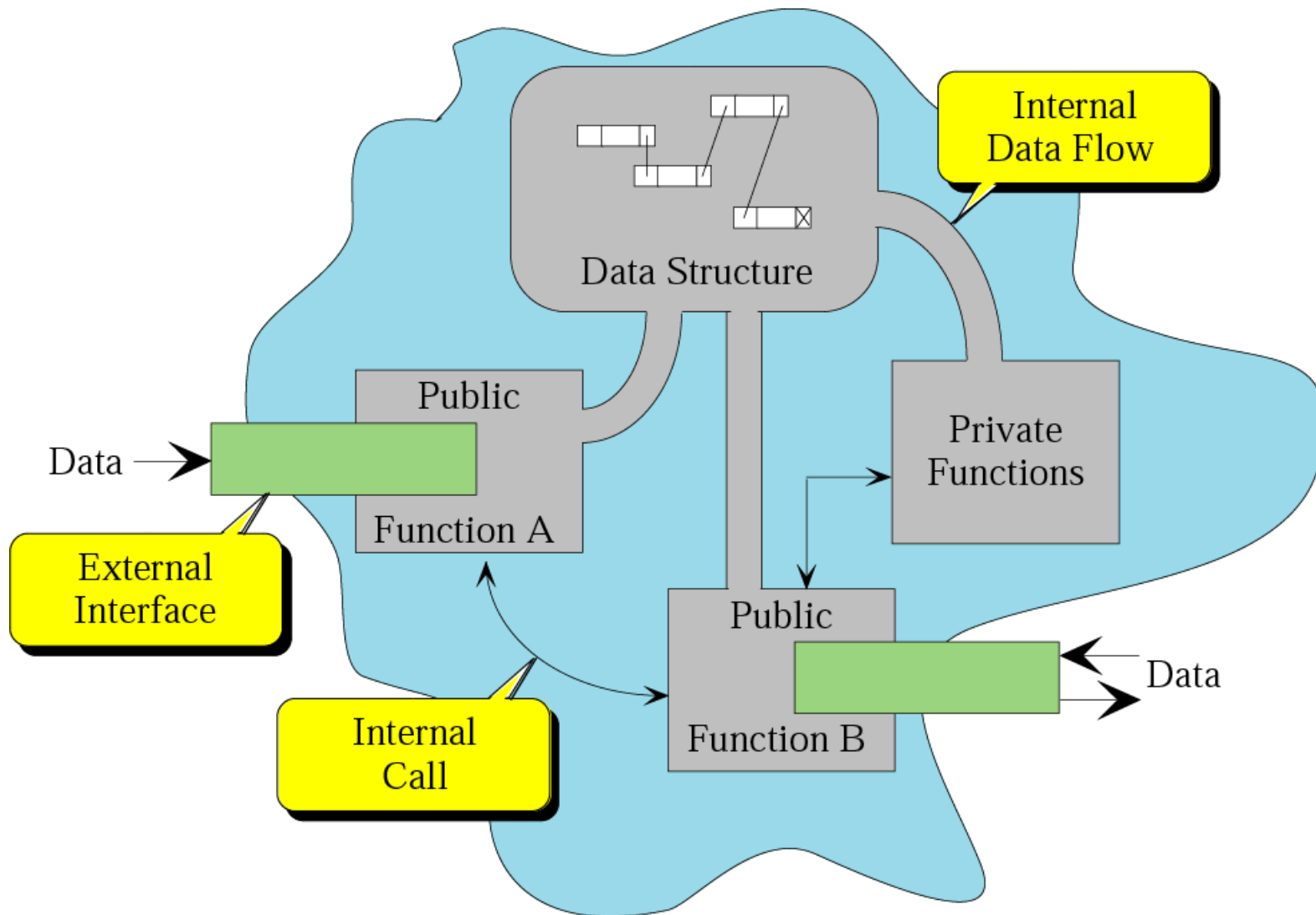
นิยาม “Abstract Data Type” (ต่อ)

- ผู้ใช้ไม่จำเป็นต้องทราบโครงสร้างข้อมูลภายในของ ADT ตัวอย่างเช่น queue เวลาที่ใช้ ตัวโปรแกรมประยุกต์ไม่จำเป็นต้องรู้ว่าโครงสร้างข้อมูลของ queue เป็นอย่างไร การอ้างอิงและการประมวลผลข้อมูลใน queue จะต้องดำเนินการโดยผ่านการกำหนดวิธีการเชื่อมต่อไปยังโครงสร้างของ queue การยอมให้โปรแกรมประยุกต์เข้าถึงโครงสร้างข้อมูลโดยตรงเป็นสิ่งที่ไม่ถูกต้อง เป็นการทำให้ ADT ไม่สามารถนำไปใช้กับการประยุกต์อื่นๆได้

แบบจำลองสำหรับ ADT

- แบบจำลองของ ADT แสดงในรูปที่ 12.1 พื้นที่ส่วนที่เป็นเงาสีฟ้าทั้งหมดแทนแบบจำลอง ภายในพื้นที่นี้จะมีส่วนที่สำคัญสองส่วนคือ **ส่วนที่เป็นโครงสร้างข้อมูล** (data structure) กับ **ส่วนที่เป็นฟังก์ชันการทำงาน** (operational functions) ทั้งสองส่วนถูกบรรจุอยู่ในแบบจำลองและไม่ได้อยู่ในขอบเขตของผู้ใช้ อย่างไรก็ตามโครงสร้างข้อมูลนี้พร้อมที่จะให้การกระทำต่างๆของ ADT ดำเนินการได้ตามที่ต้องการ และการกระทำหนึ่งอาจเรียกให้อีกการกระทำหนึ่งทำงานให้กับตนเองก็ได้ กล่าวอีกอย่างหนึ่งคือทั้งโครงสร้างข้อมูลและฟังก์ชันต่างก็อยู่ในขอบเขตของกันและกันนั่นเอง





การดำเนินการกับ ADT

- ในการทำงานของคอมพิวเตอร์นั้น มีการนำข้อมูลเข้าสู่ระบบ (enter) มีการเข้าถึงข้อมูลที่เก็บอยู่ในระบบ (access) มีการแก้ไขเปลี่ยนแปลงข้อมูลที่เก็บอยู่ในระบบ (modify) และมีการลบทิ้งข้อมูลจากระบบ (delete) โดยผ่านส่วนการกระทำเชื่อมต่อที่เป็นรูปสี่เหลี่ยมผืนผ้า (รูปที่ 12.1) การกระทำแต่ละอย่างจะมีอัลกอริธึมดำเนินการ ผู้ใช้จะเห็นเฉพาะชื่อการกระทำและค่าพารามิเตอร์ของการกระทำเท่านั้น ทั้งชื่อและพารามิเตอร์เท่านั้นที่เชื่อมต่อกับ ADT นอกจากนี้เรายังสามารถสร้างและนิยามการกระทำที่เราต้องการเพิ่มขึ้นอีกก็ได้

12.2

LINEAR LISTS



Linear Lists

เราจะเริ่มด้วยการนิยามและอธิบาย ADT ตัวแรกคือ linear list โดย linear list เป็นรายการเชื่อมโยงที่สมาชิกแต่ละตัวในรายการจะมีตัวตามเพียงตัวเดียวที่ไม่ซ้ำกัน หรือกล่าวอีกอย่างหนึ่งคือ linear list มีโครงสร้างที่เป็นลำดับ (sequential structure) ลักษณะของ linear list แสดงดังรูปที่ 12.2

Linear list แบ่งออกเป็น 2 ประเภทใหญ่ๆคือ **แบบทั่วไป** (general list) กับ **แบบมีข้อจำกัด** (restricted list) มีรายละเอียดดังนี้

1. **General list** เป็นรายการที่ข้อมูลอาจ**แทรกหรือลบทิ้ง** ณ ส่วนใดของลิสต์ก็ได้ และไม่มีข้อจำกัดเกี่ยวกับการกระทำที่จะใช้ทำการประมวลผลรายการในลิสต์ ลิสต์แบบทั่วไปแบ่งได้อีกเป็น 2 ชนิดคือ



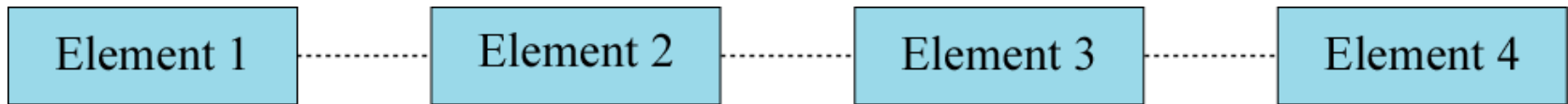
Linear Lists (ต่อ)

random list กับ **ordered list** ใน random list จะไม่มีการจัดลำดับของข้อมูล ส่วนใน ordered list ข้อมูลจะมีการเรียงลำดับตามคีย์ ซึ่งก็อาจประกอบด้วยข้อมูลของฟิลด์ตั้งแต่หนึ่งฟิลด์ขึ้นไปภายในโครงสร้าง ในโครงสร้างอะเรย์หนึ่งมิติที่ได้ศึกษามาแล้ว ข้อมูลในอะเรย์จะทำหน้าที่เป็นคีย์ด้วย

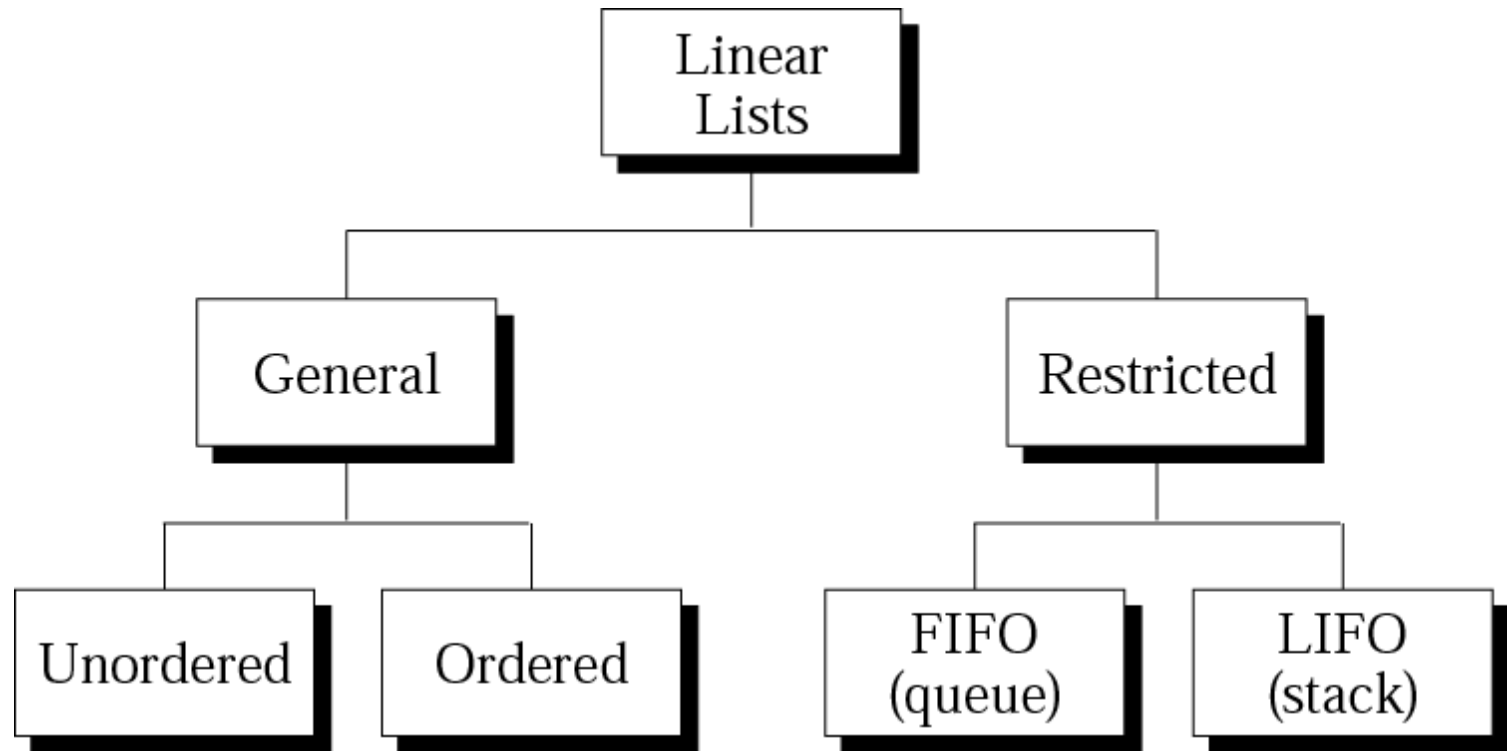
2. **Restricted list** เป็นรายการที่ข้อมูลสามารถแทรกหรือลบทิ้งได้เฉพาะ **ส่วนหัว** (head) หรือ **ส่วนท้าย** (tail) ของลิสต์เท่านั้น ลิสต์ประเภทนี้ 2 ชนิดคือ First In First Out list (**FIFO**) กับ Last In First Out list (**LIFO**) กรณีที่เป็น FIFO เราเรียกว่า **queue** ส่วน LIFO เรียกว่า **stack**



Linear list



ประเภทของ Linear lists

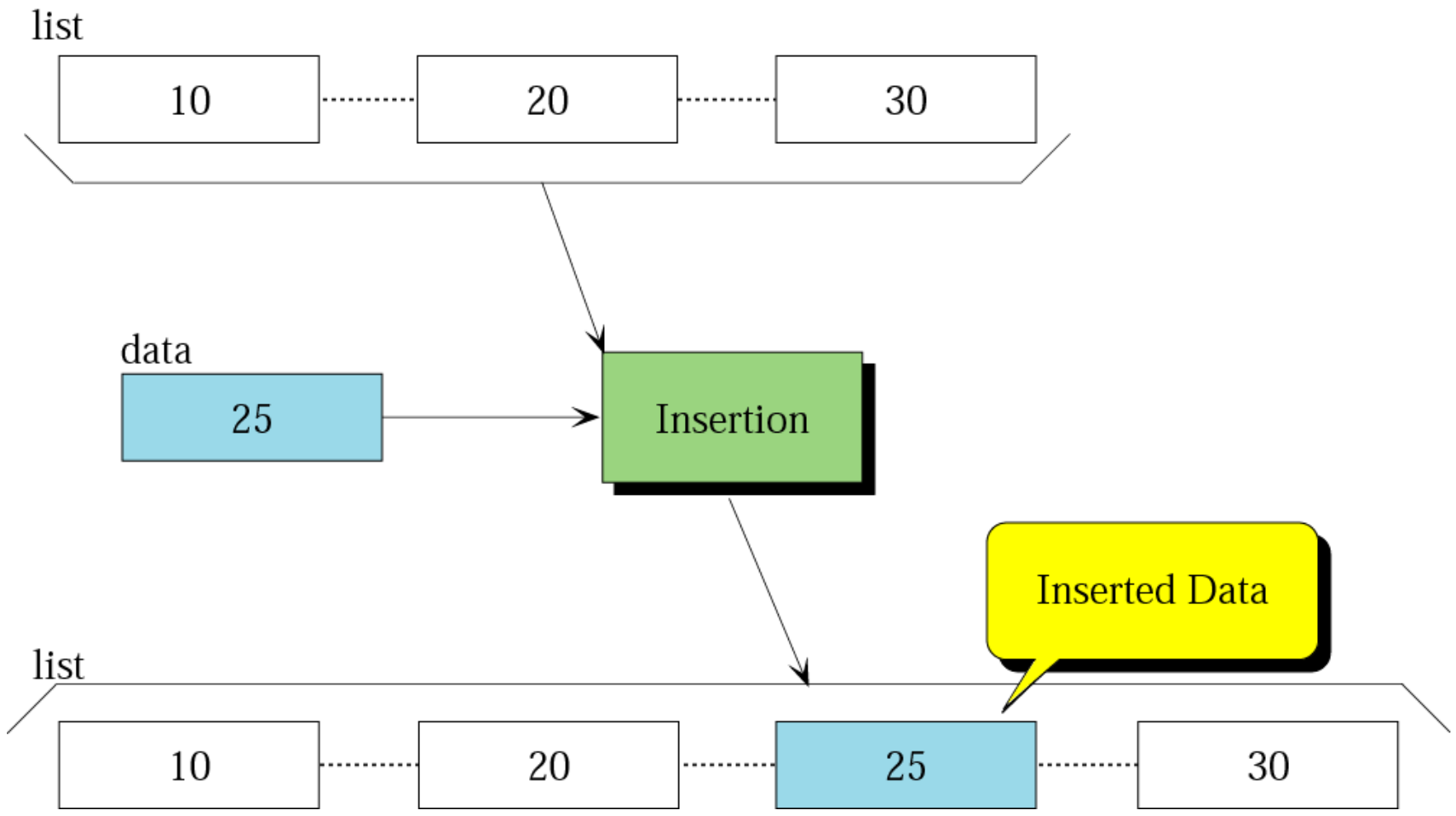


การดำเนินการกับ Linear Lists

- ในหัวข้อนี้จะอธิบายการดำเนินการกับ linear list ซึ่งมี 4 รูปแบบคือ การแทรก (insert) การตัดออก (delete) การค้นคืน (retrieve) และ traverse
 1. **การแทรก** คือการนำข้อมูลแทรกเข้าไปในลิสต์ ในกรณีที่เป็น ordered list การแทรกข้อมูลใหม่เข้าไปจะต้องรักษาลำดับของข้อมูลในลิสต์ด้วยการแทรกเพื่อรักษาลำดับอาจแทรกได้เป็นลำดับแรกหรือลำดับสุดท้ายในลิสต์ก็ได้ แต่ส่วนใหญ่แล้วจะใส่ ณ ตำแหน่งภายในลิสต์ การหาตำแหน่งที่เหมาะสมเราจะต้องใช้อัลกอริธึมค้นหา ในกรณีที่ไม่มีที่ว่างพอที่จะใส่ข้อมูลใหม่เข้าไปในลิสต์ เราเรียนสถานการณ์ดังกล่าวว่า **overflow** รูปที่ 12.4 แสดงการแทรกข้อมูลที่มีค่า 25 เข้าไปในลิสต์



การแทรกโหนดเข้าไปใน Linear list

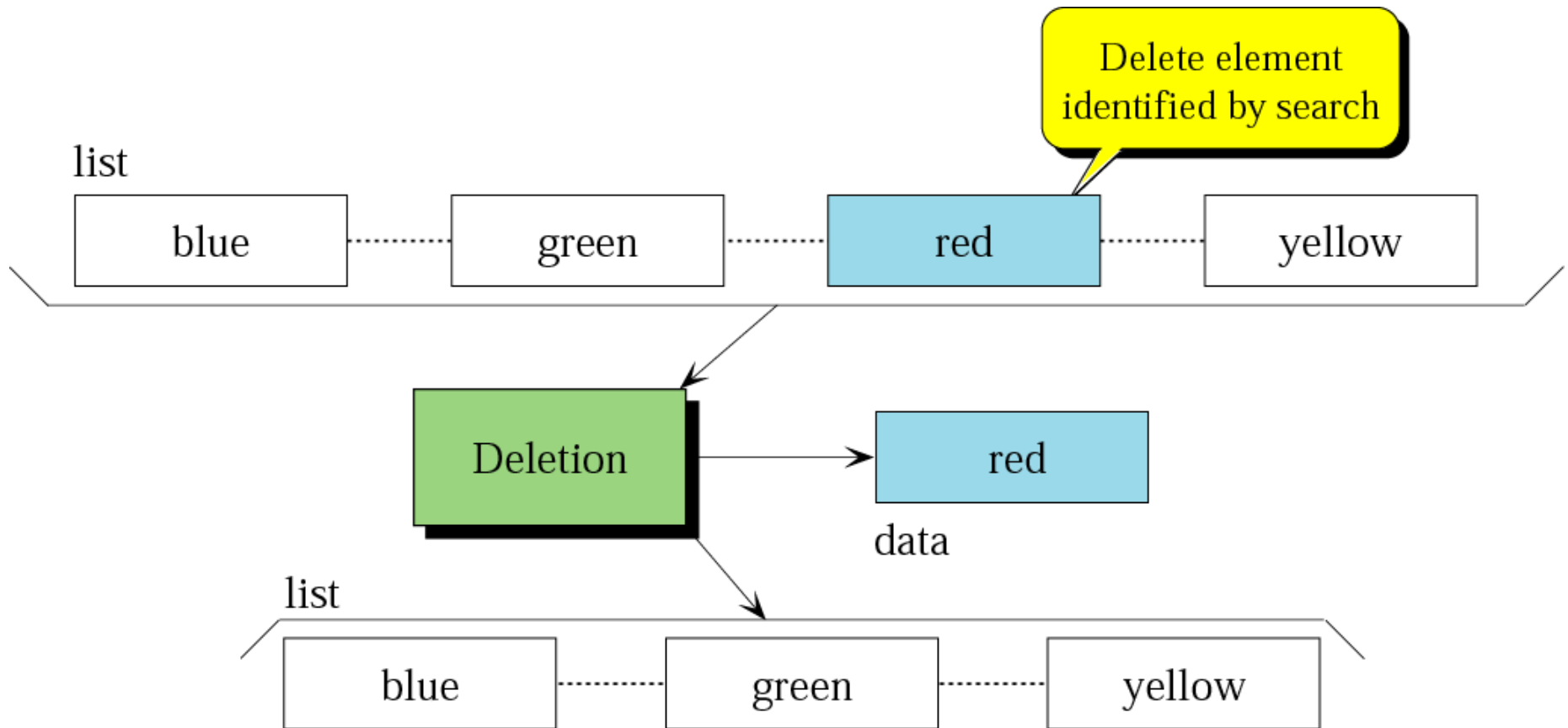


การดำเนินการกับ Linear List (ต่อ)

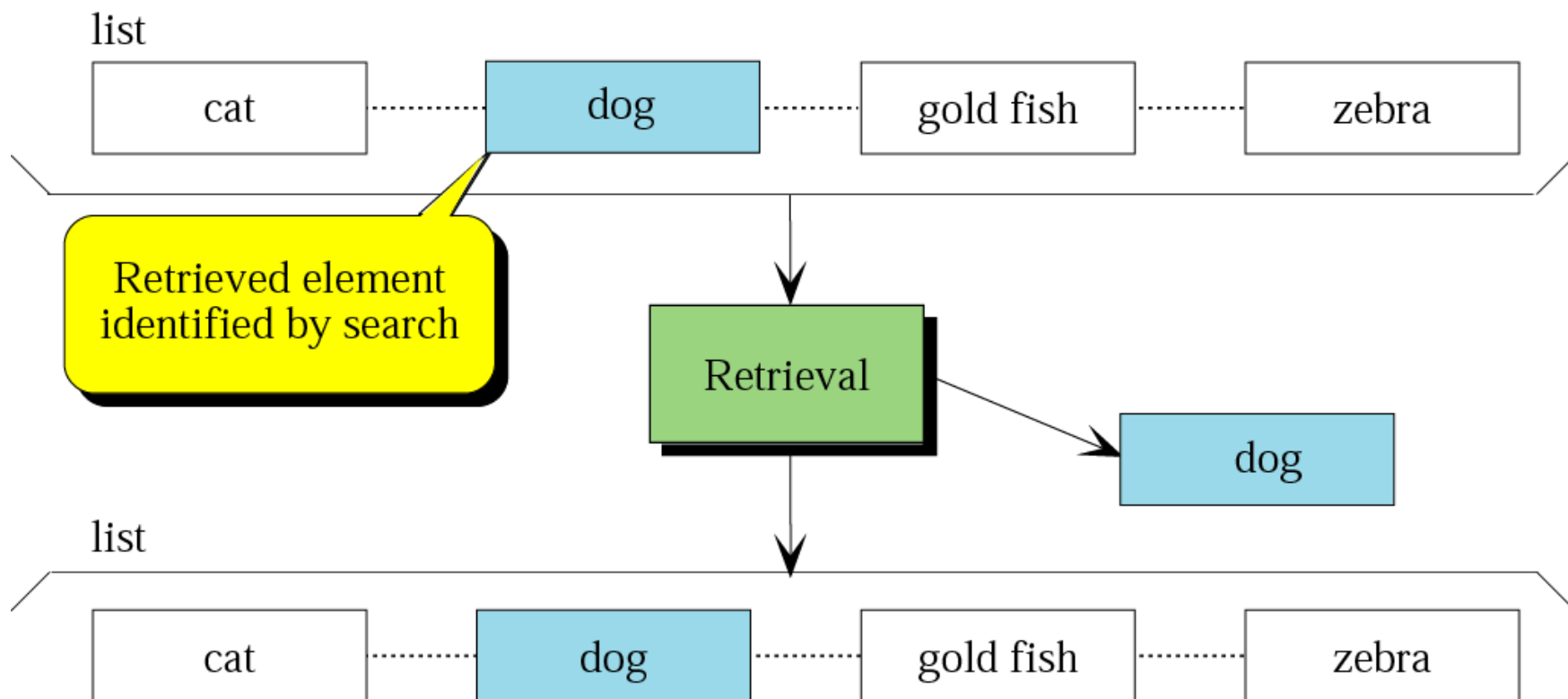
2. **การตัดออก** เป็นการดำเนินการที่ต้องทำการค้นหาตำแหน่งของข้อมูลที่จะตัดออกก่อน อัลกอริธึมสืบค้นตามลำดับ (sequential search algorithm) ใดๆก็สามารถใช้ได้ เมื่อค้นตำแหน่งได้แล้วจึงตัดข้อมูลนั้นออกไปจากลิสต์ ปัญหาอย่างหนึ่งของการตัดออกที่อาจจะเกิดขึ้นคือลิสต์เป็นลิสต์ว่าง (empty list) ในกรณีนี้เราเรียกว่าเกิดสถานะ **underflow** ขึ้นรูปที่ 12.5 แสดงการตัดออกข้อมูล “red” จากลิสต์

3. **การค้นคืน** เป็นการดำเนินการที่ต้องทำการค้นหาตำแหน่งของข้อมูลเหมือนกับการดำเนินการตัดออก เมื่อพบแล้วจึงทำการคัดลอกข้อมูลออกมาใช้งานโดยไม่เปลี่ยนข้อมูลเดิม ถ้าลิสต์ว่างจะไม่สามารถทำได้

การตัดโหนดออกจาก linear list



การค้นคืนข้อมูลจาก Linear list



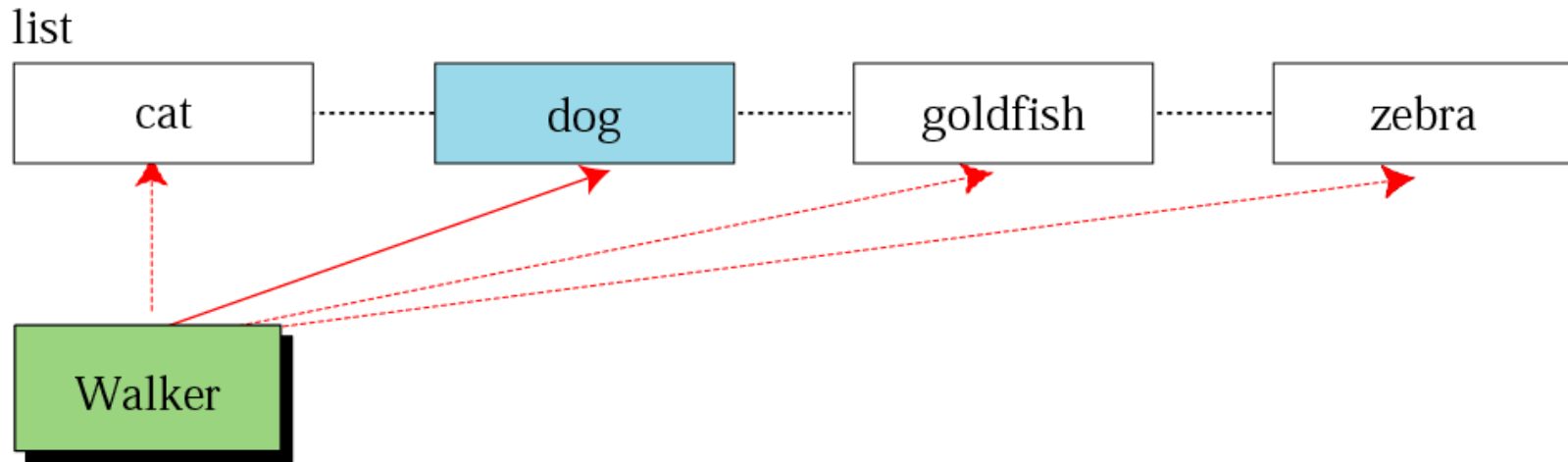
การดำเนินการกับ Linear List (ต่อ)

4. **Traverse** เป็นการกระทำที่สมาชิกแต่ละตัวในลิสต์ถูกประมวลผลตามลำดับที่ละตัว (ดังรูปที่ 12.7) จากรูปจะมีตัวแปรประเภท pointer ชื่อ “**Walker**” ซึ่งไปยังข้อมูลที่กำลังทำการประมวลผลอยู่ การประมวลผลอาจเป็นการค้นคืน การจัดเก็บ การเปลี่ยนแปลงค่า หรือการกระทำอื่นๆก็ได้ การ traverse ตามปกติจะต้องทำซ้ำๆ (loop) โดยไม่ต้องใช้การค้นหา แต่ละรอบของการทำซ้ำ สมาชิกหนึ่งตัวของลิสต์จะถูกประมวลผล การทำซ้ำๆจะหยุดเมื่อสมาชิกทุกตัวได้รับการประมวลผลเรียบร้อยแล้ว



รูปที่ 12-7

การ Traverse linear list



การ Implement Linear List

- การ **implement linear list** ที่นิยมใช้กันมากมี 2 วิธีคือใช้ **Array** กับใช้ **Linked list** รายละเอียดจะอธิบายในหัวข้อต่อไป ส่วนการประยุกต์ linear list นั้นมีมากมายหลายสถานการณ์เช่นในมหาวิทยาลัย linear list สามารถใช้เก็บข้อมูลนักศึกษาที่ลงทะเบียนในแต่ละภาคการศึกษา ในระบบสารสนเทศโรงพยาบาล linear list อาจใช้เก็บข้อมูลยาในคลังยา เป็นต้น ในหัวข้อต่อไป จะอธิบายการประยุกต์ใช้ linear list แทน stack และ queue นักศึกษาจะได้เห็นประโยชน์ของ linear list มากขึ้น

12.3

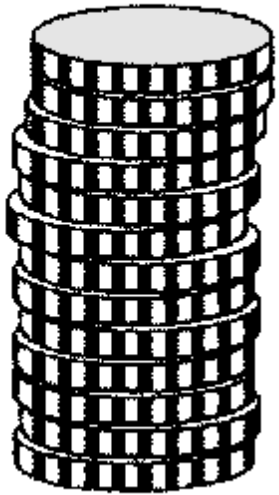
STACKS

Stack

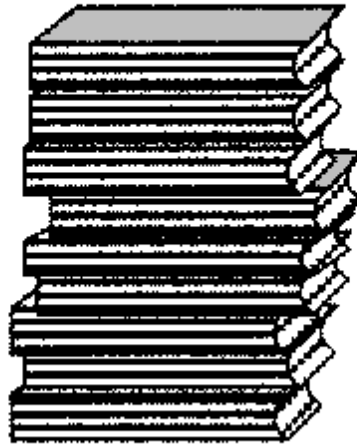
- **Stack** เป็น linear list ที่จำกัดให้การเพิ่มและการลบสมาชิกต้องทำที่ส่วน **หัวของลิสต์** (ภาษาอังกฤษเรียกว่า “**top**”) ถ้าเรา**ใส่** (push) ข้อมูลหลายๆ ตัวเข้าไปใน stack แล้วทำการ**ดึงออก** (pop) ลำดับของข้อมูลที่จะดึงออกจะสลับกับข้อมูลที่เข้า หมายความว่าข้อมูลที่เข้าทีหลังจะถูกดึงออกมาก่อน เช่นถ้าข้อมูลนำเข้าเป็น 5, 10, 15, 20 เมื่อถูกดึงออกจะเป็น 20, 15, 10, 5 ด้วยเหตุนี้ทำให้ stack รู้จักกันในนามโครงสร้างข้อมูลประเภท Last In, First Out (LIFO) รูปที่ 12.8 แสดงให้เห็นตัวอย่างของ stack 3 ประเภท คือ stack กองเหรียญ stack กองหนังสือ และ stack ที่ใช้ในคอมพิวเตอร์



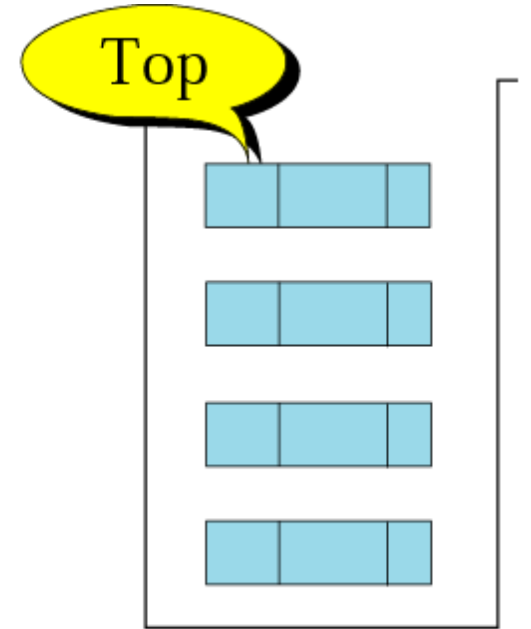
ตัวอย่างของ stack



Stack of Coins



Stack of Books



Computer Stack

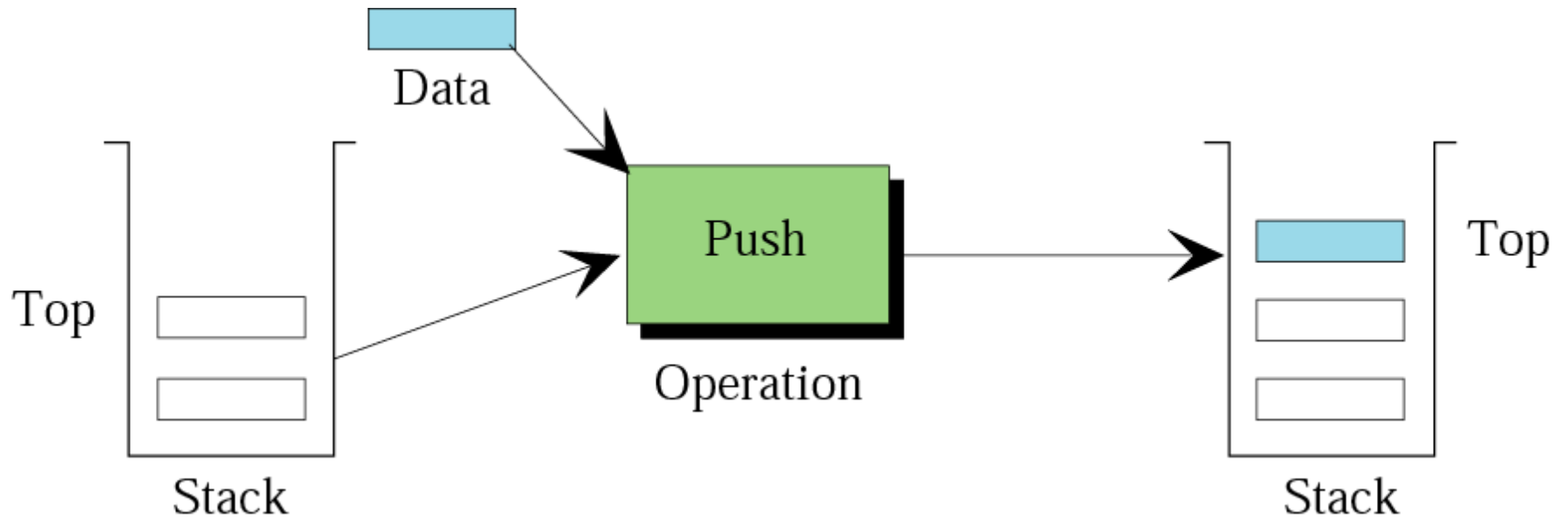
การดำเนินการกับ Stack

- แม้ว่าเราสามารถนิยามการดำเนินการกับ stack ได้หลายรูปแบบ แต่การดำเนินการที่เป็นพื้นฐานของ stack มี 3 แบบคือ push, pop, และ empty รายละเอียดของการดำเนินการแต่ละแบบมีดังนี้

1. **Push** เป็นการเพิ่มสมาชิกหรือข้อมูลเข้าไปใน stack (รูปที่ 12.9) หลังจากทำการ push แล้วสมาชิกหรือข้อมูลใหม่จะอยู่ที่ top ของ stack ปัญหาหนึ่งที่เกิดขึ้นคือ **stack ไม่มีที่ว่างสำหรับข้อมูลใหม่** ในกรณีนี้ เราเรียกว่า stack เกิด **overflow** และข้อมูลนั้นจะไม่สามารถใส่เข้าไปใน stack ได้

รูปที่ 12-9

การดำเนินการ Push กับ stack



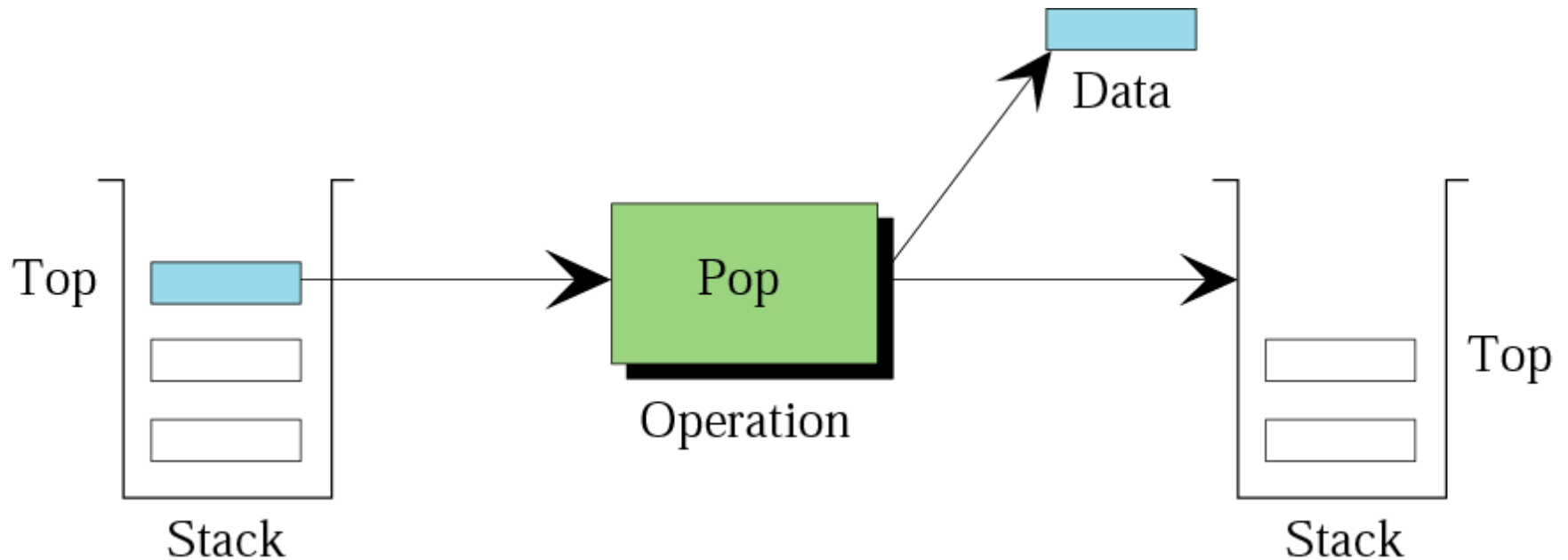
การดำเนินการกับ Stack (ต่อ)

2. **Pop** เป็นการดำเนินการที่ตัด (remove) สมาชิกหรือข้อมูลที่อยู่ **top** ของ stack ออก (รูปที่ 12.10) เมื่อข้อมูลที่ top ของ stack ถูก pop ออก ค่าของ top จะชี้ไปยังสมาชิกหรือข้อมูลตัวถัดไป (กรณีที่เป็น Array ค่าของ top จะลดลง 1 คือ $top = top - 1$) ในกรณีที่ข้อมูลตัวสุดท้ายใน stack ถูก pop ออกไปจะทำให้ stack ว่าง ถ้าการกระทำ pop เรียกใช้กับ stack ที่ว่าง จะทำให้เกิดสถานะ **underflow**

3. **Empty** เป็นการดำเนินการที่ตรวจสอบว่า stack ว่างหรือไม่? ถ้า stack ว่าง การดำเนินการ empty จะส่งกลับค่า “**true**” แต่ถ้า stack ไม่ว่าง การกระทำ empty จะส่งค่า “**false**” กลับไปยังผู้เรียกใช้

รูปที่ 12-10

การดำเนินการ Pop กับ stack



ตัวอย่างที่ 1 จงแสดงผลลัพธ์จากการดำเนินการกับ stack S:

push (S , 10)

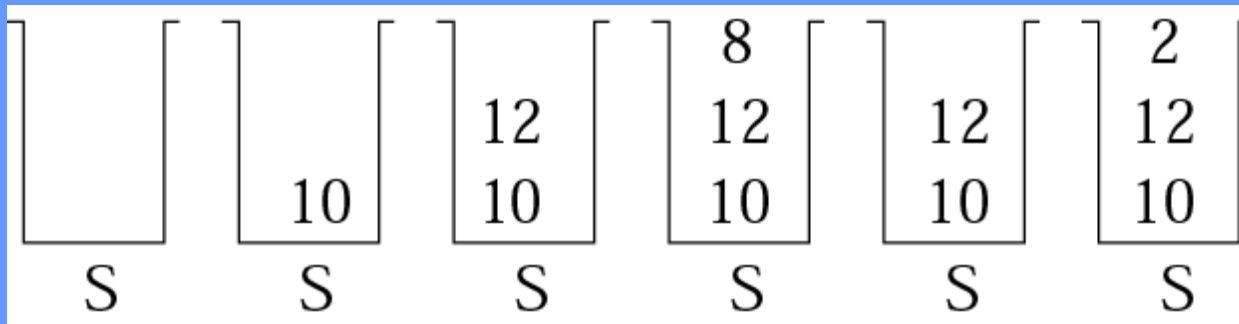
push (S , 12)

push (S , 8)

if not empty (S), then pop (S)

push (S , 2)

วิธีทำ



การ Implement Stack

- ถึงแม้ว่าเราสามารถ implement stack โดยใช้ Array หรือ Linked List ก็ได้ แต่ที่นิยมใช้กันมากคือใช้ linked list เนื่องจากการดำเนินการ pop และ push สามารถทำได้โดยง่าย ส่วนการประยุกต์ใช้ stack มีมากมาย หลายสถานการณ์ แบ่งออกเป็นกลุ่มหลักได้ 4 กลุ่มดังนี้

1. **Reversing data** เป็นการใช้ stack เพื่อทำการสลับที่ของข้อมูล ซึ่งข้อมูลตัวสุดท้ายจะเปลี่ยนตำแหน่งกับข้อมูลตัวแรก ตัวที่สองสลับที่กับตัวรองสุดท้าย ตัวต่อไปก็สลับเช่นเดียวกัน เช่นข้อมูล 1 2 3 4 เปลี่ยน เป็น 4 3 2 1 เป็นต้น

2. **Parsing** เป็นกระบวนการแบ่งแยกข้อมูลออกเป็นส่วนย่อยๆ

การ Implement Stack (ต่อ)

ที่เป็นอิสระต่อกันเพื่อนำไปใช้ประมวลตามวัตถุประสงค์อื่นต่อไป เช่น การแปลโค้ดโปรแกรมเป็นภาษาเครื่อง ตัวแปลภาษาจะต้องแบ่งแยกโปรแกรมออกเป็นส่วนย่อยๆ เช่น คำสำคัญ (keywords) ชื่อ (names) และ เครื่องหมาย (tokens) เป็นต้น

ปัญหาที่พบบ่อยในการเขียนโปรแกรมคือการไม่เท่ากันของวงเล็บเปิดกับวงเล็บปิด ซึ่งเกิดจากจำนวนวงเล็บเปิดไม่เท่ากับวงเล็บปิด การประยุกต์โดยใช้ stack คือเมื่อพบวงเล็บเปิดให้ทำการ push เข้าไปใน stack และเมื่อพบวงเล็บปิดให้ pop วงเล็บเปิดที่อยู่ top ของ stack ทิ้งไป เมื่อประมวลผลเสร็จและ stack ว่าง แสดงว่าจำนวนวงเล็บเท่ากัน

การ Implement Stack (ต่อ)

3. **Postponement** เป็นกระบวนการยืดเวลาการใช้ข้อมูลออกไปจนกว่าจะถึงเหตุการณ์ตามที่กำหนด stack สามารถใช้เป็นเครื่องมือเพื่อบรรลุมัตถุประสงค์นี้ได้ ซึ่งนักศึกษาจะเห็นได้ในหัวข้อต่อไป

4. **Backtracking** เป็นกระบวนการดำเนินการที่ย้อนกลับไปยังข้อมูลก่อนข้อมูลปัจจุบันเช่นการเล่นเกม การวิเคราะห์การตัดสินใจ และวิธีการที่ใช้ในระบบผู้เชี่ยวชาญเป็นต้น

12.4

QUEUES



Queue

- **Queue** เป็น linear list ที่ข้อมูลสามารถ**เพิ่ม** (add) ได้เฉพาะที่**ส่วนท้าย** (rear หรือ tail) ของลิสต์ ส่วนการ**ตัดออก** (delete) กระทำได้เฉพาะที่**ส่วนหน้าหรือหัว** (front หรือ head) ของลิสต์เท่านั้น ข้อกำหนดนี้เป็น การประกันว่าข้อมูลที่ถูกประมวลผลโดยใช้ queue นั้นจะถูกดำเนินการ ตามลำดับของการเข้าสู่ queue กล่าวอีกอย่างหนึ่งคือ queue เป็นกลไกที่มีโครงสร้างเป็นแบบ First In, First Out (FIFO)
- Queue มีความหมายเหมือนกับ**แถวหรือเส้น** (line) เช่นเมื่อท่านต้องการ จ่ายเงินเพื่อซื้อสินค้าตามห้างสรรพสินค้า ถ้ามีคนจำนวนมากท่าน จะต้องเข้าแถวหรือเข้าคิว เครื่องบินที่จะบินขึ้นจากสนามบิน ถ้าเครื่อง

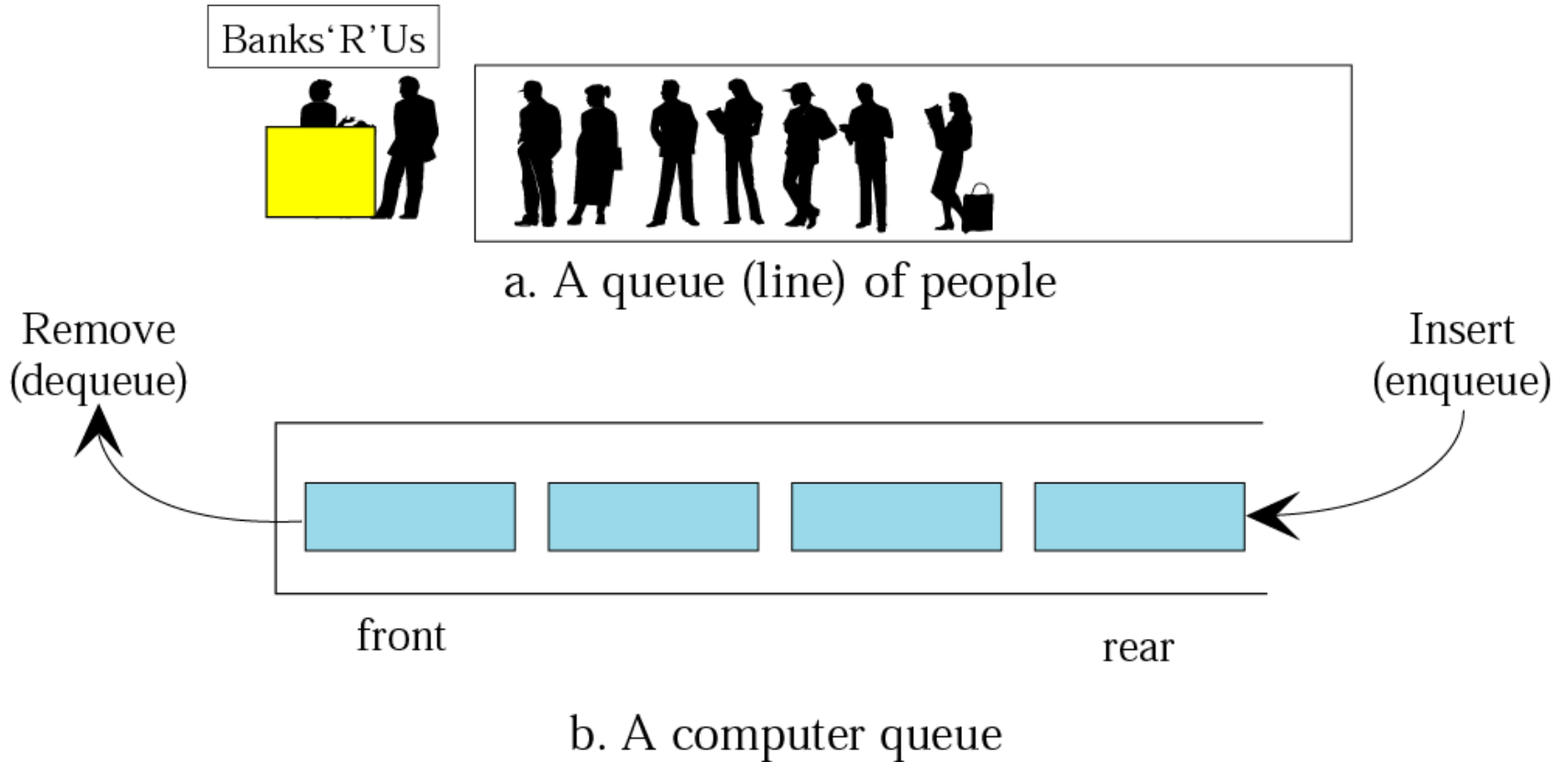


Queue (ต่อ)

- บินมีหลายลำ แต่ละลำก็ต้องเข้าคิวเพื่อรอขึ้น โทรศัพท์ที่เรียกเข้ามายังศูนย์เพื่อรับบริการจะถูกจัดให้อยู่ในคิวตามลำดับเป็นต้น รูปที่ 12.12 แสดงการแทน queue สองชนิดคือคิวของคนกับคิวของข้อมูลในคอมพิวเตอร์ ทั้งคนและข้อมูลเข้าสู่คิวทางด้านท้ายแล้วค่อยๆเคลื่อนไปอยู่ด้านหน้าหรือหัวของคิว จากนั้นจะถูกตัดออกไปใช้งานตามวัตถุประสงค์
- **การดำเนินการกับ queue** เราสามารถกำหนดการดำเนินการกับ queue ได้หลายรูปแบบ แต่การดำเนินการพื้นฐานมี 3 แบบคือ **การใส่ข้อมูลเข้าไปในคิว (insert/enqueue)** **การตัดข้อมูลออกจากคิว (remove/dequeue)** และการทดสอบว่าคิวว่างหรือไม่? (empty) รายละเอียดมีดังนี้



การแทน Queue



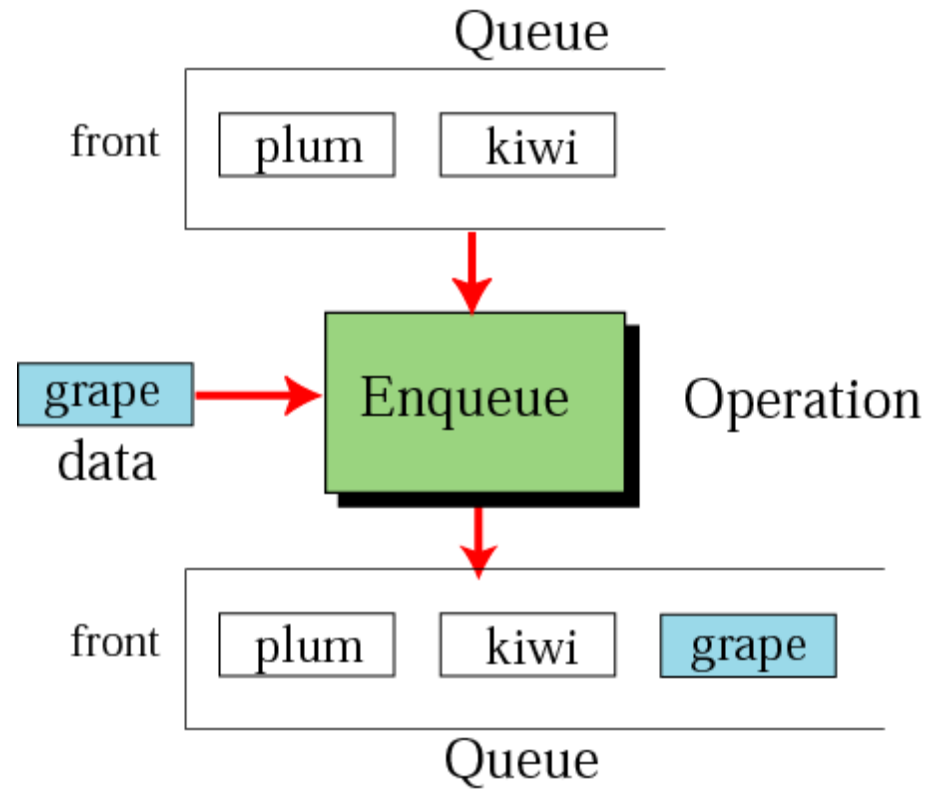
การดำเนินการกับ queue (ต่อ)

1. **Enqueue/Insert** เป็นการดำเนินการที่เพิ่มข้อมูลเข้าไปในคิว ในรูปที่ 12.13 หลังจากข้อมูลถูกใส่เข้าไปในคิวแล้ว ข้อมูลใหม่จะอยู่ที่ตำแหน่งท้ายสุด ปัญหาการเพิ่มข้อมูลเข้าไปในคิวก็เช่นเดียวกับ stack คือถ้าคิวไม่มีที่ว่างหรือคิวเต็มก็จะเกิดสถานะที่เรียกว่า **overflow** ขึ้น

2. **Dequeue/Remove** เป็นการตัดสมาชิกหรือข้อมูลที่อยู่ตรงหัวหรือด้านหน้าออกจากคิว ในรูปที่ 12.14 ข้อมูลที่อยู่ที่หัวคิวถูกตัดออกแล้วส่งค่าไปให้ผู้ใช้อย่างไรก็ตาม ถ้าไม่มีข้อมูลอยู่ในคิวและต้องการตัด ก็จะก่อให้เกิดสถานะที่เรียกว่า **underflow** ขึ้น

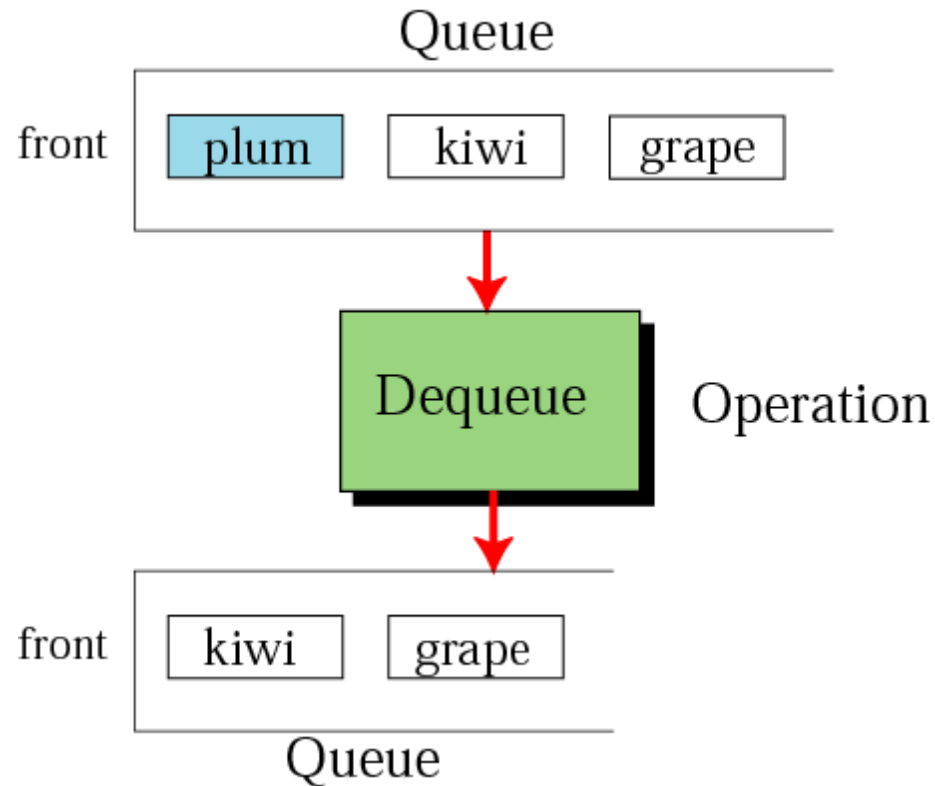
รูปที่ 12-13

การดำเนินการ Enqueue



รูปที่ 12-14

การดำเนินการ Dequeue



การดำเนินการกับ queue (ต่อ)

3. **Empty** เป็นการดำเนินการที่ตรวจสอบว่าคิวว่างหรือไม่? ถ้าว่างจะส่งค่า “true” ไปยังผู้ใช้ ถ้าคิวยังมีข้อมูลอยู่ก็จะส่งค่า “false”

ตัวอย่างที่ 2: จงแสดงผลที่เกิดขึ้นการกระทำกับคิว Q ต่อไปนี้

enqueue (Q , 23)

If not empty (Q) Then dequeue (Q)


enqueue (Q, 20)


enqueue (Q, 19)


If not empty (Q) Then dequeue (Q)


รูปที่ 12-15

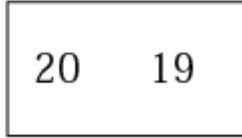
วิธีทำ

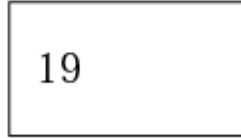
front 
Q

front 
Q

front 
Q

front 
Q

front 
Q

front 
Q

การ Implement Queue

Queue อาจ implement ได้โดยใช้ **Array** หรือ **Linked list** ก็ได้

ส่วนการประยุกต์ของคิวนั้นมีใช้กับทุกภาคส่วนเช่นการประมวลผลของระบบปฏิบัติการคอมพิวเตอร์ การทำงานของระบบเครือข่าย การทำงานในทางธุรกิจเช่นการประมวลคำร้องของลูกค้า และการประมวลใบสั่งซื้อ โดยเฉพาะอย่างยิ่งในระบบคอมพิวเตอร์ คิวมีความจำเป็นสำหรับการประมวลผลงาน และระบบบริการก็จำเป็นต้องใช้คิวเช่น spool ในการพิมพ์ เป็นต้น ในการศึกษาที่สูงขึ้น อาจมีการนิยามคิวเพิ่มเติมที่ซับซ้อนกว่าที่เรียนมาแล้วเช่น **priority queue, circular queue** ที่รายละเอียดมีมากขึ้น



12.5

TREES



TREE

Tree เป็น ADT ที่มีบทบาทการประยุกต์มากมายในการทำงานของระบบคอมพิวเตอร์ เป็นโครงสร้างที่มีประสิทธิภาพมากสำหรับการค้นหาข้อมูลที่มีขนาดใหญ่ ข้อมูลที่มีการเคลื่อนไหวและเปลี่ยนแปลงอยู่เสมอ มีการประยุกต์ใช้ tree ในหลายสาขาเช่น ปัญญาประดิษฐ์ และอัลกอริธึมการเข้ารหัส เป็นต้น ในหัวข้อนี้จะอธิบายแนวคิดพื้นฐานของ tree ส่วนในหัวข้อต่อไปจะอธิบายแนะนำ tree ชนิดพิเศษที่มีชื่อว่า binary tree ซึ่งเป็นโครงสร้างที่นิยมใช้กันมากในศาสตร์คอมพิวเตอร์

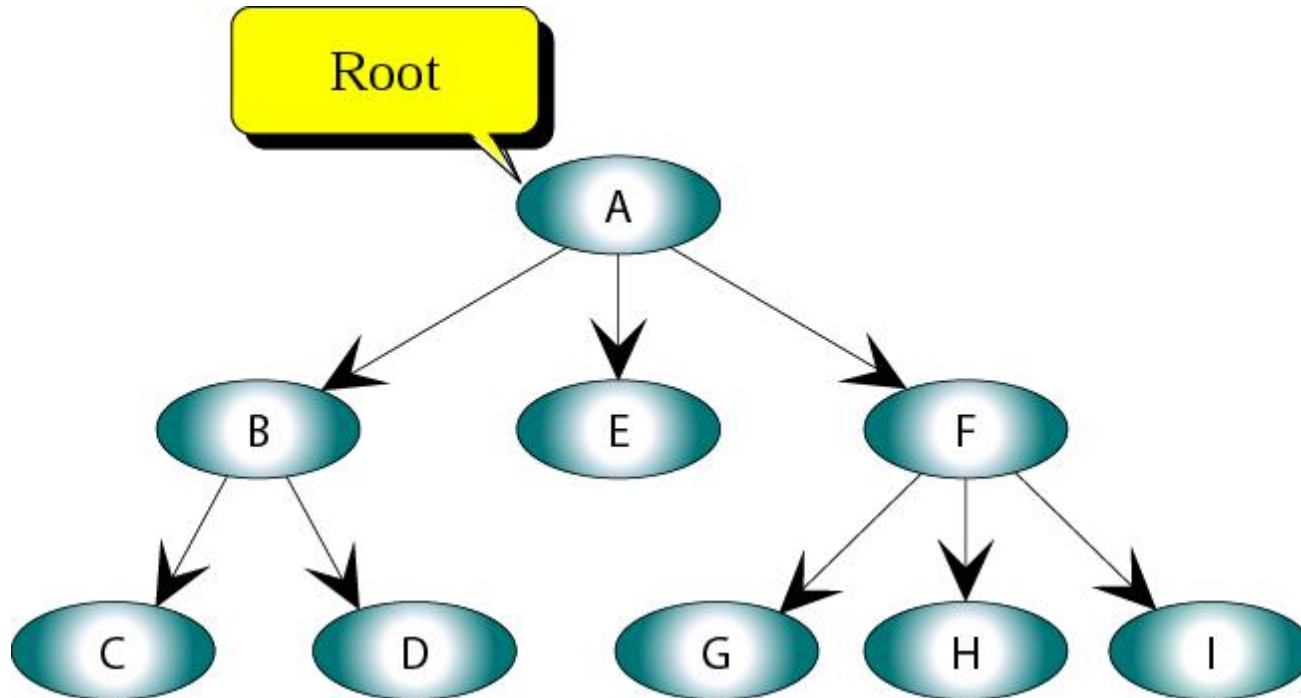
นิยาม: Tree เป็นโครงสร้างที่ประกอบด้วยส่วนสำคัญ 2 ส่วนคือ

1. เซตของ **โหนด** (nodes) ที่มีจำนวนจำกัด และ

TREE (ต่อ)

2. เซตจำกัดของ edges หรือ branches เป็นเส้นที่มีทิศทาง (directed lines) ที่เชื่อมต่อระหว่างโหนด จำนวนของ edge ที่เกี่ยวข้องกับโหนดใด เราจะเรียกว่าเป็น “degree” ของโหนดนั้น เมื่อ edge เชื่อมตรงไปยังโหนดใด จะเรียก edge นั้นว่า “indegree edge” ของโหนดนั้น เมื่อ edge เชื่อมโยงออกจากโหนดใด จะเรียก edge นั้นว่า “outdegree edge” ของโหนดนั้น ผลบวกของ indegree edge กับ outdegree edge ของโหนดใดๆ จะเรียกว่า “degree” ของโหนดนั้น ตัวอย่างในรูปที่ 12.16 degree ของโหนด B เท่ากับ 3

การแทนโครงสร้าง Tree



TREE (ต่อ)

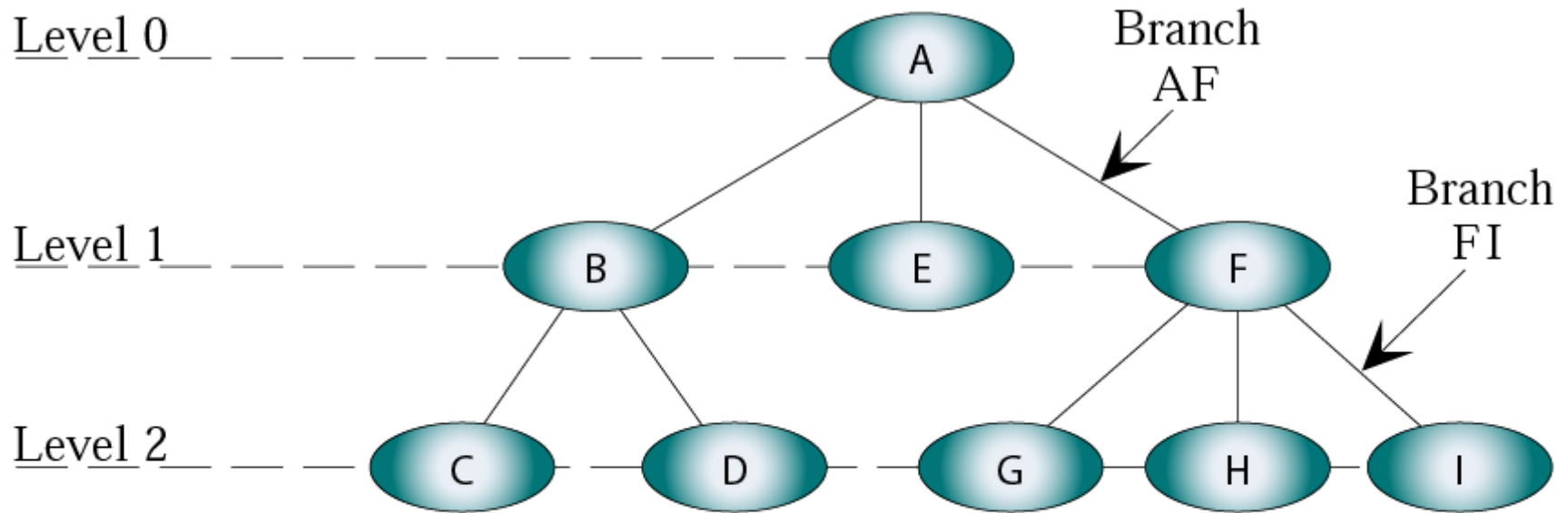
ถ้า tree ไม่เป็น empty tree โหนดแรกใน tree เรียกว่า “root”
ขอให้สังเกตว่าข้อความต่อไปนี้เป็นจริงเสมอ:

1. indegree ของ root มีค่าเท่ากับ 0
2. โหนดทุกโหนดนอกจาก root มี indegree เท่ากับ 1
3. ทุกโหนดใน tree อาจมี outdegree เป็น 0, 1, หรือมากกว่าก็ได้

คำศัพท์เฉพาะ

นิยาม: Leaf คือโหนดที่มี outdegree เท่ากับ 0 รูปที่ 12.16 โหนด C, D, E, G, H, I เป็น leaf ของ tree ส่วนโหนดที่ไม่ได้เป็นทั้ง root และ leaf เรียกว่า “โหนดภายใน” หรือ “internal node” ที่เรียกเช่นนี้อาจเป็น เพราะว่าโหนดเหล่านี้อยู่ภายใน tree ก็ได้ จากรูปที่ 12.16 โหนด B และ โหนด F เป็น internal nodes

นิยาม: โหนดใดๆจะเรียกว่าเป็น “parent node” ถ้าโหนดนั้นมี outdegree มากกว่า 0 (คือโหนดที่มีโหนดตามหรือมี successor)



Parents: A, B, F

Children: B, E, F, C, D, G, H, I

Siblings: {B,E,F}, {C,D}, {G,H,I}

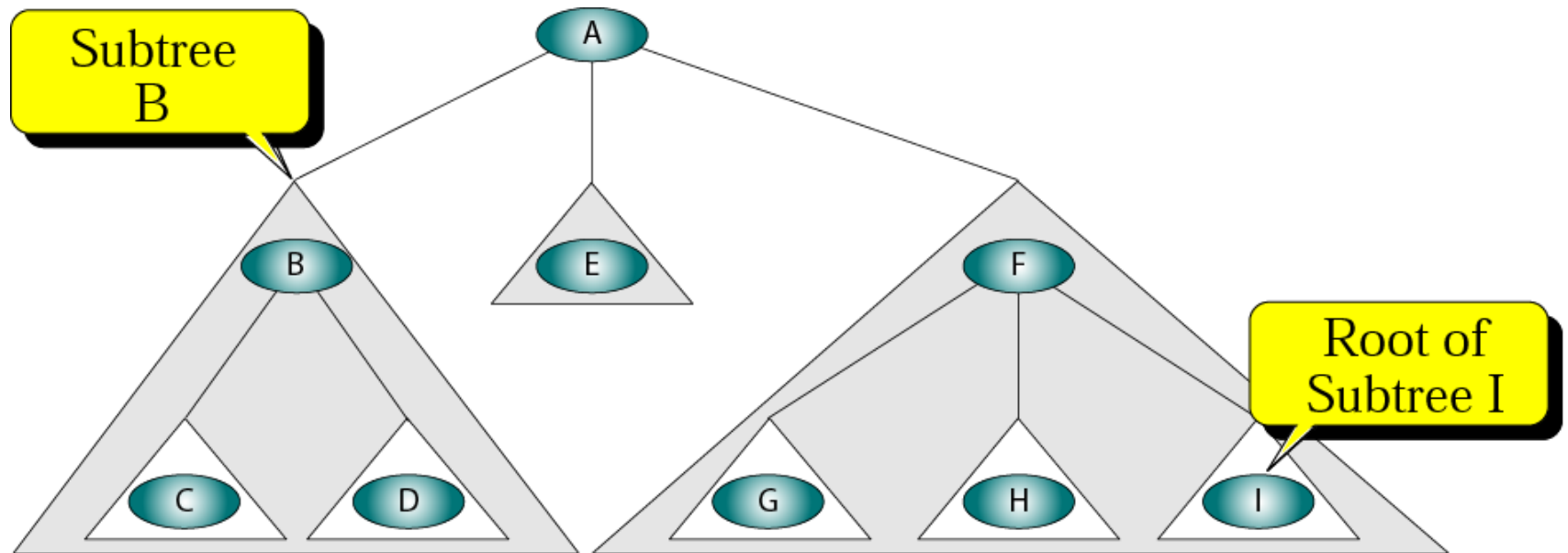
Leaves

Internal nodes

C, D, E, G, H, I

B, F

Subtrees

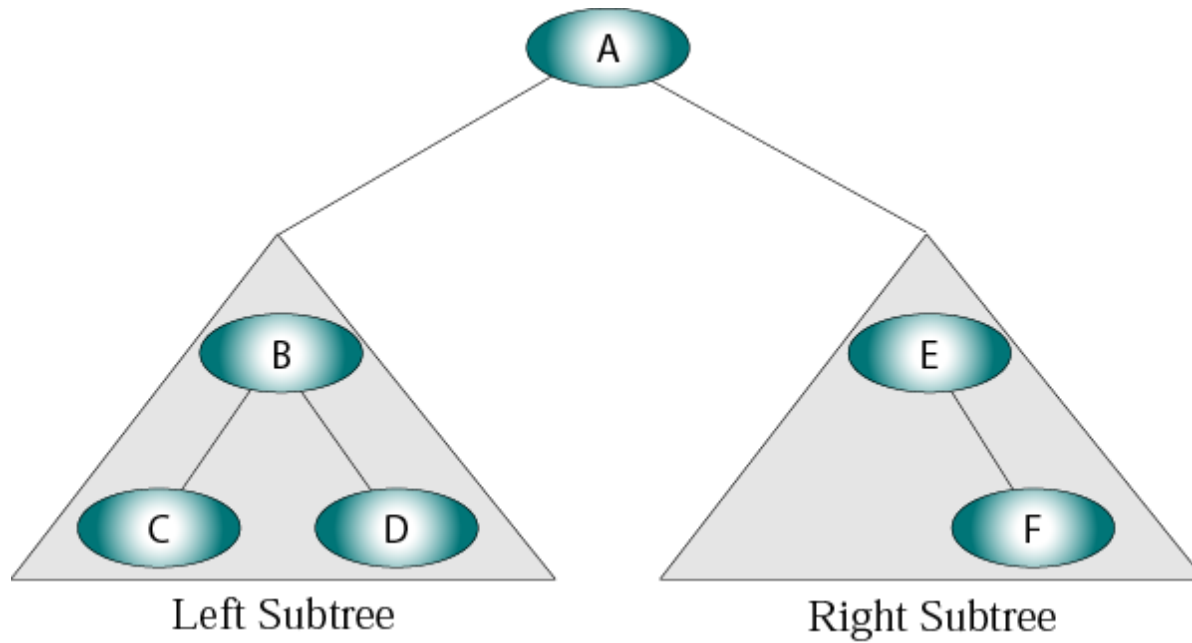


12.6

BINARY TREES



Binary tree



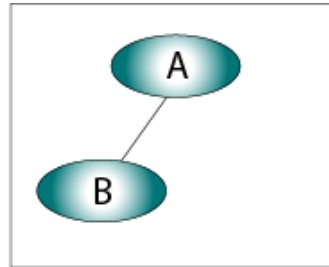
ตัวอย่าง binary trees



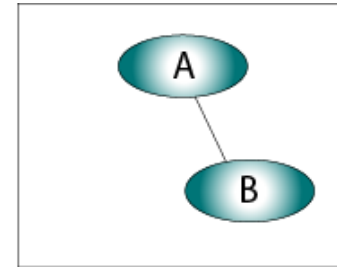
a.



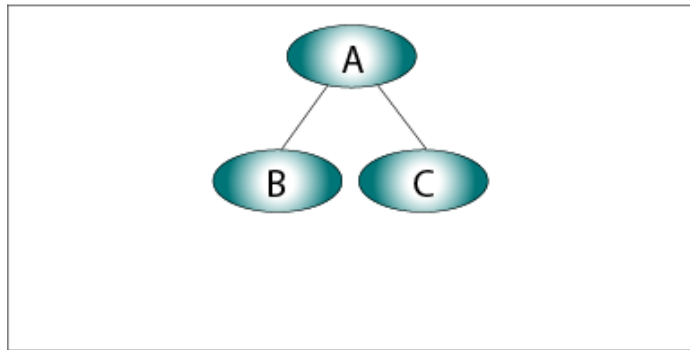
b.



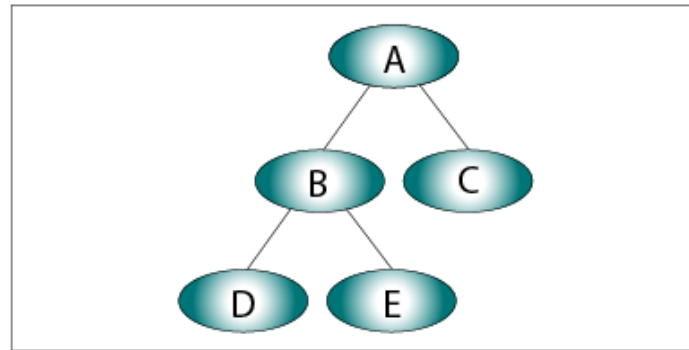
c.



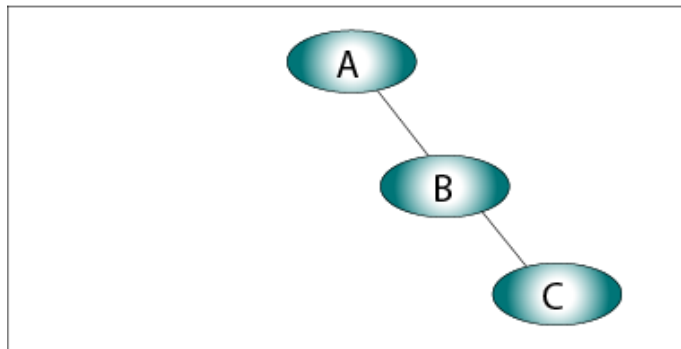
d.



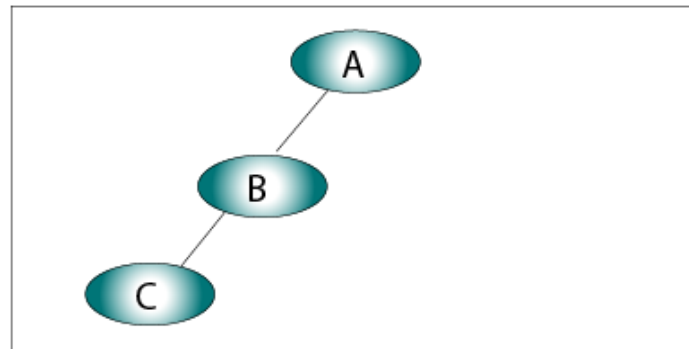
e.



f.

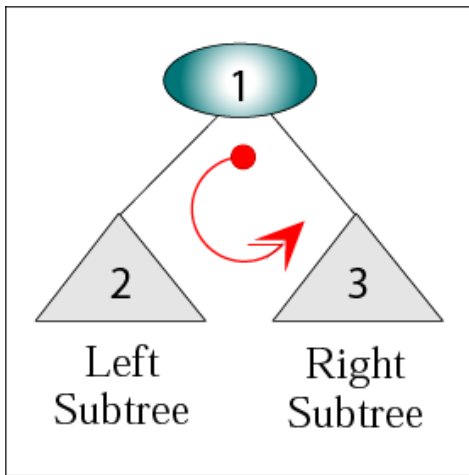


g.

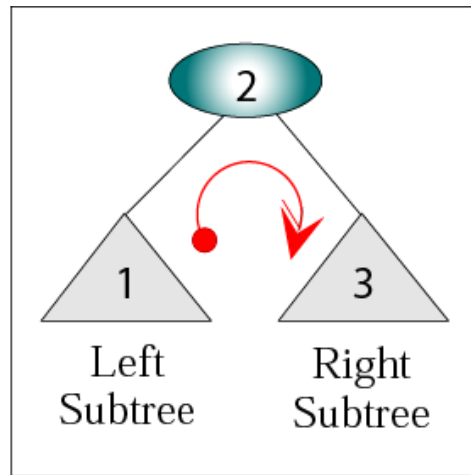


h.

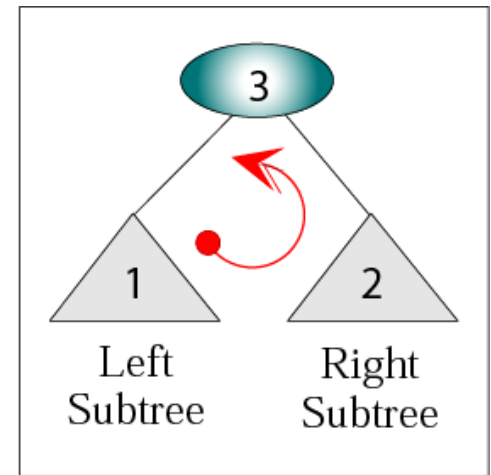
การ Traverse Binary Tree แบบ Depth-first



a. Preorder Traversal

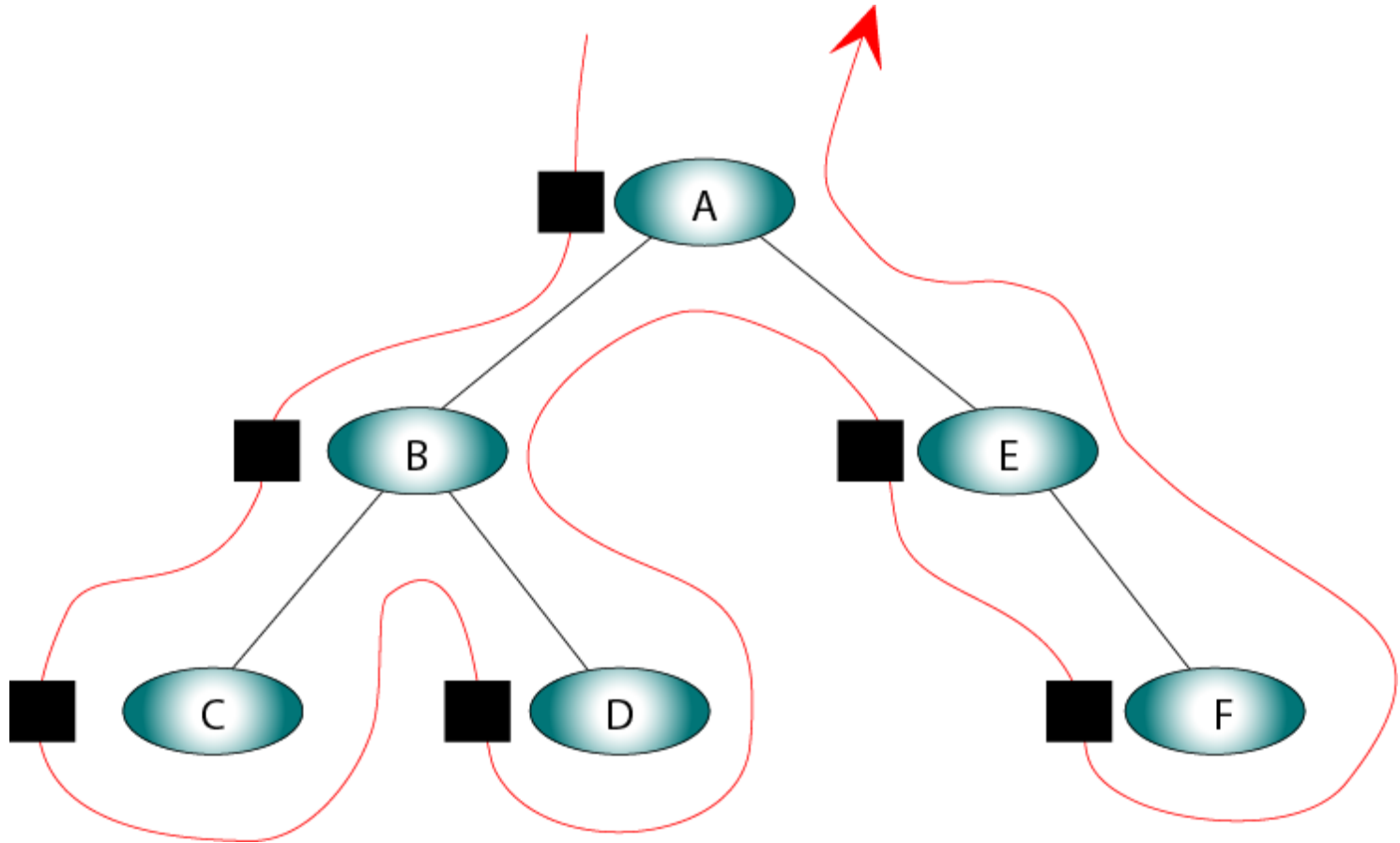


b. Inorder Traversal

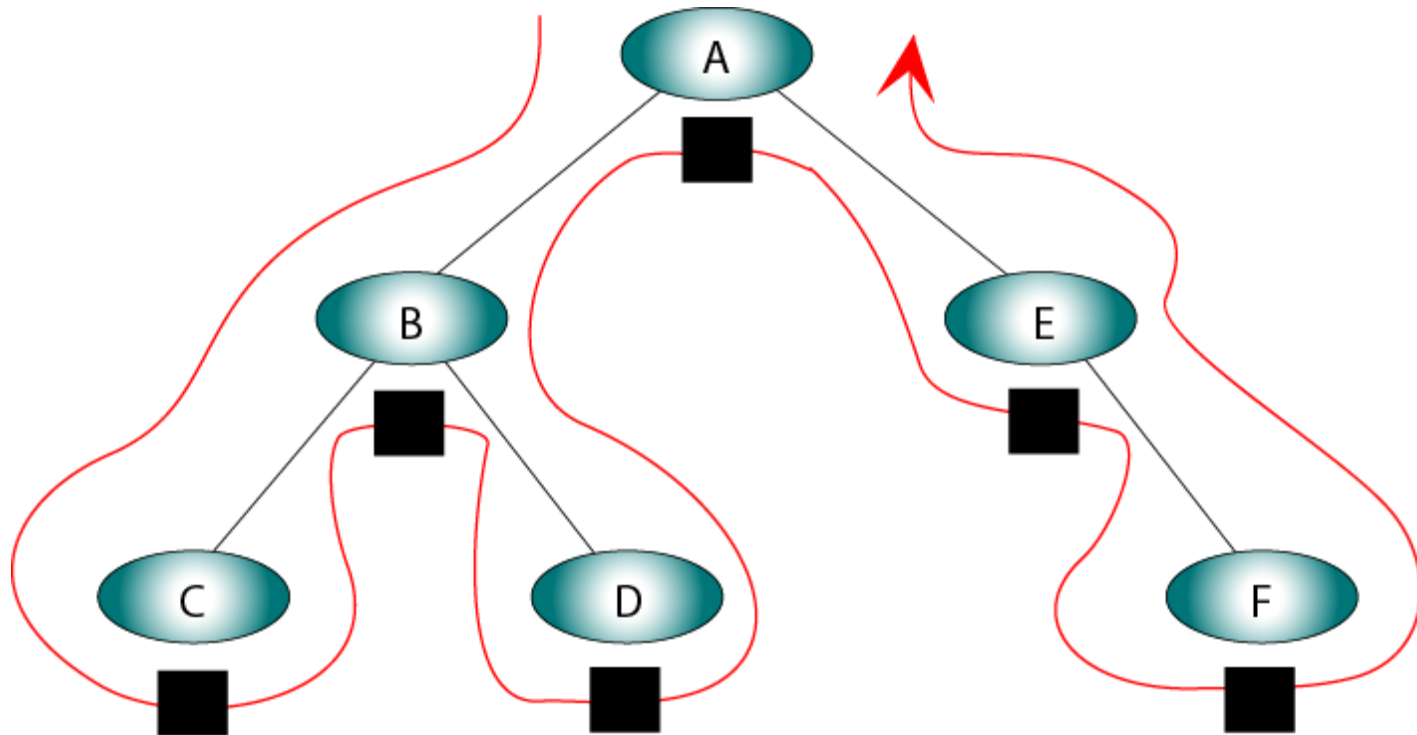


c. Postorder Traversal

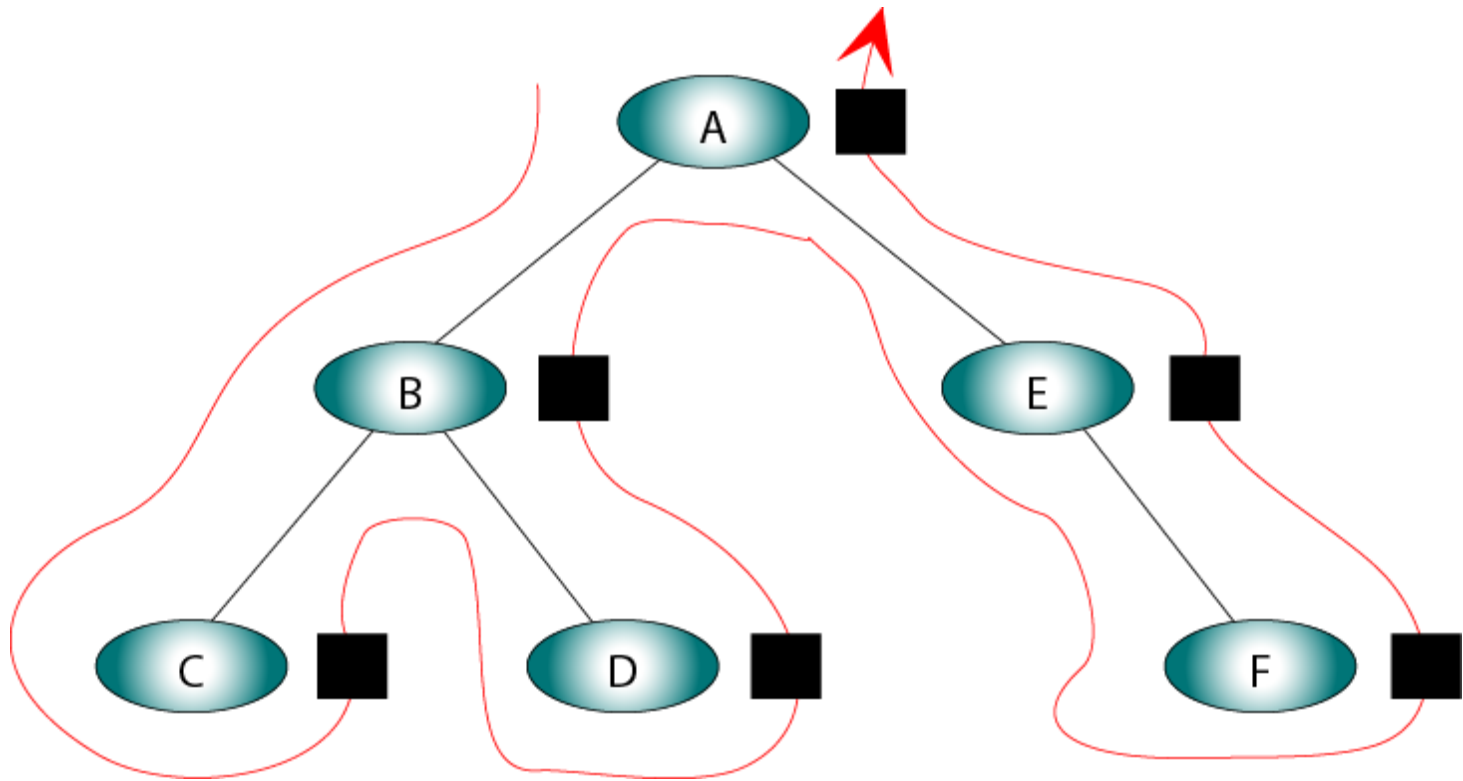
การ Traverse Binary Tree แบบ Preorder



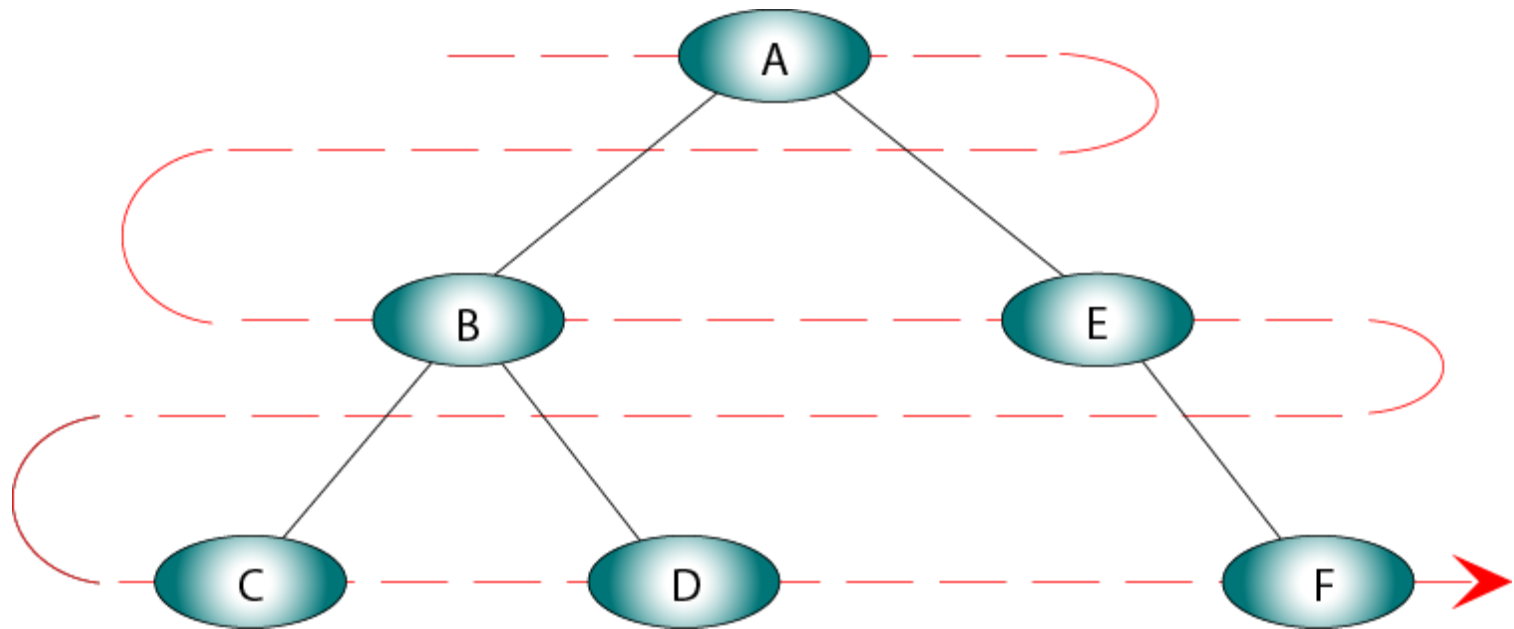
การ Traverse Binary Tree แบบ Inorder



การ Traverse Binary Tree แบบ Postorder

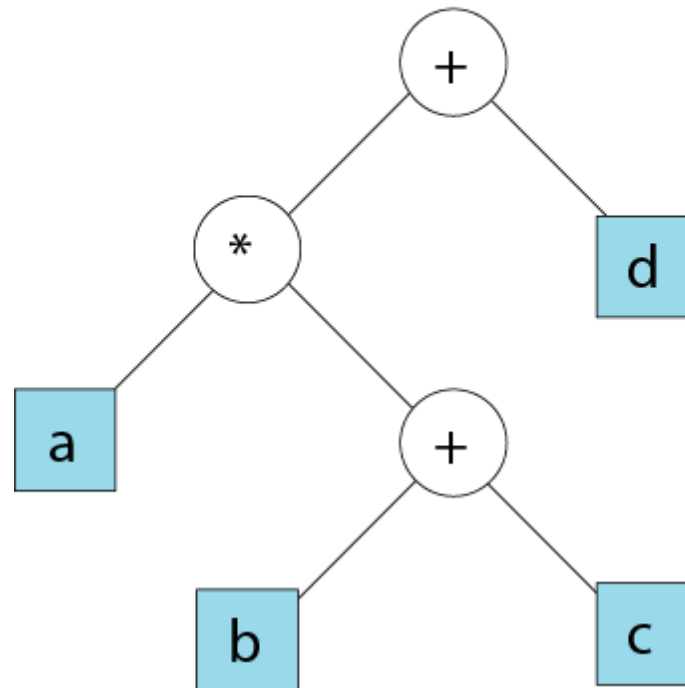


การ Traverse Binary Tree แบบ Breadth-first



Expression tree

$a * (b + c) + d$

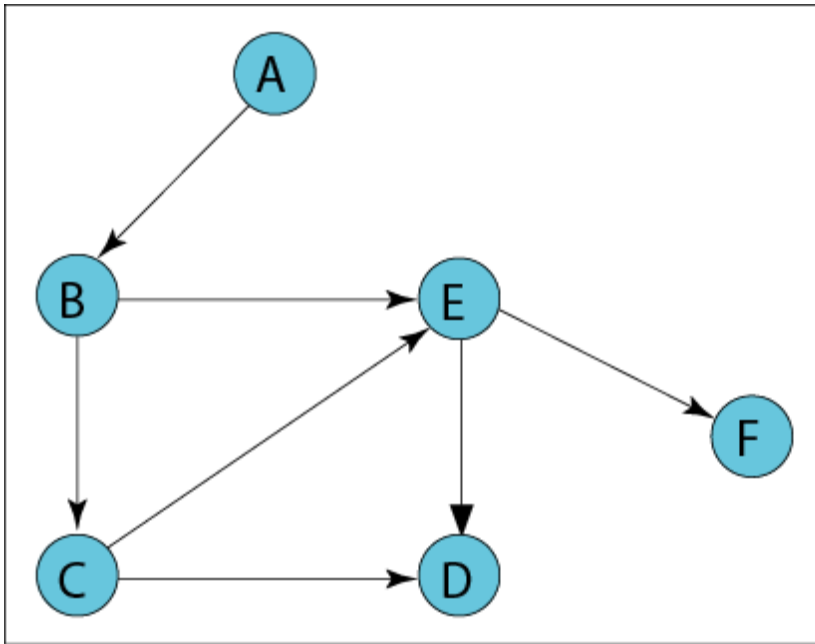


12.7

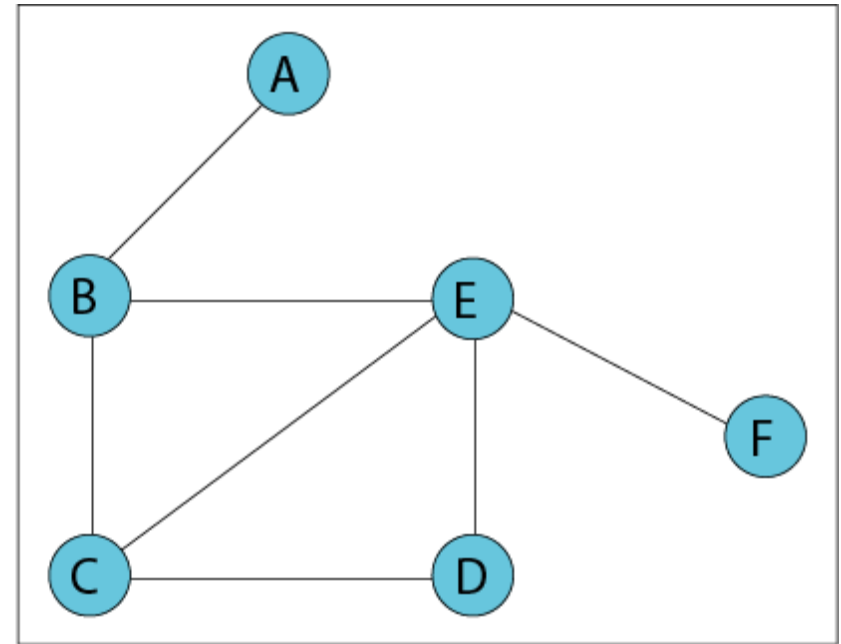
GRAPHS



Directed and undirected graphs

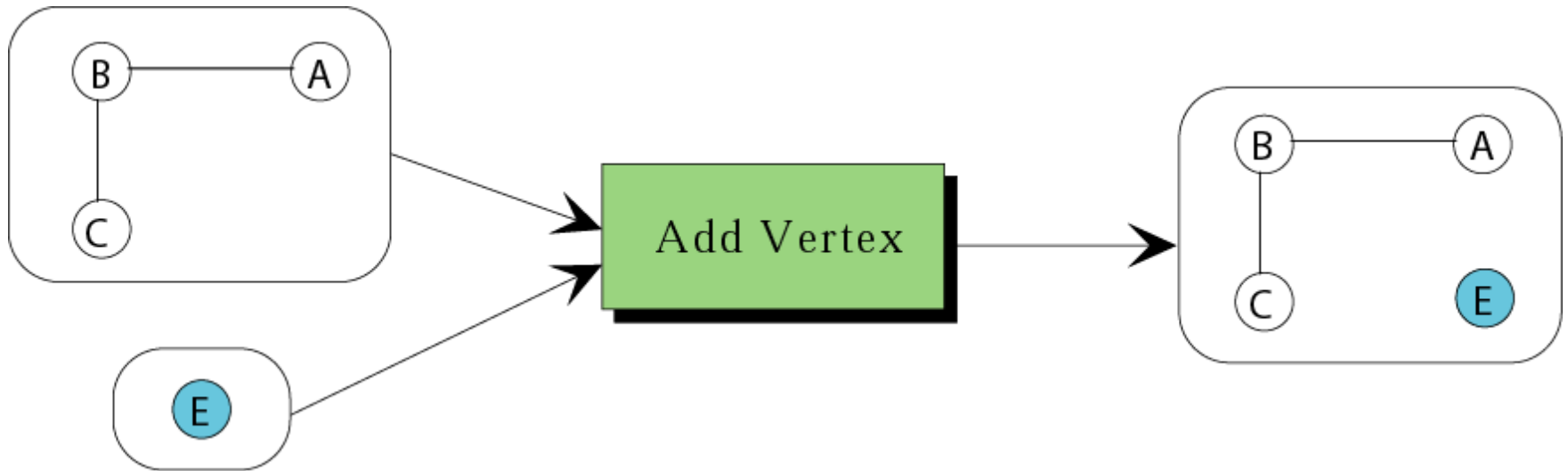


a. Directed Graph

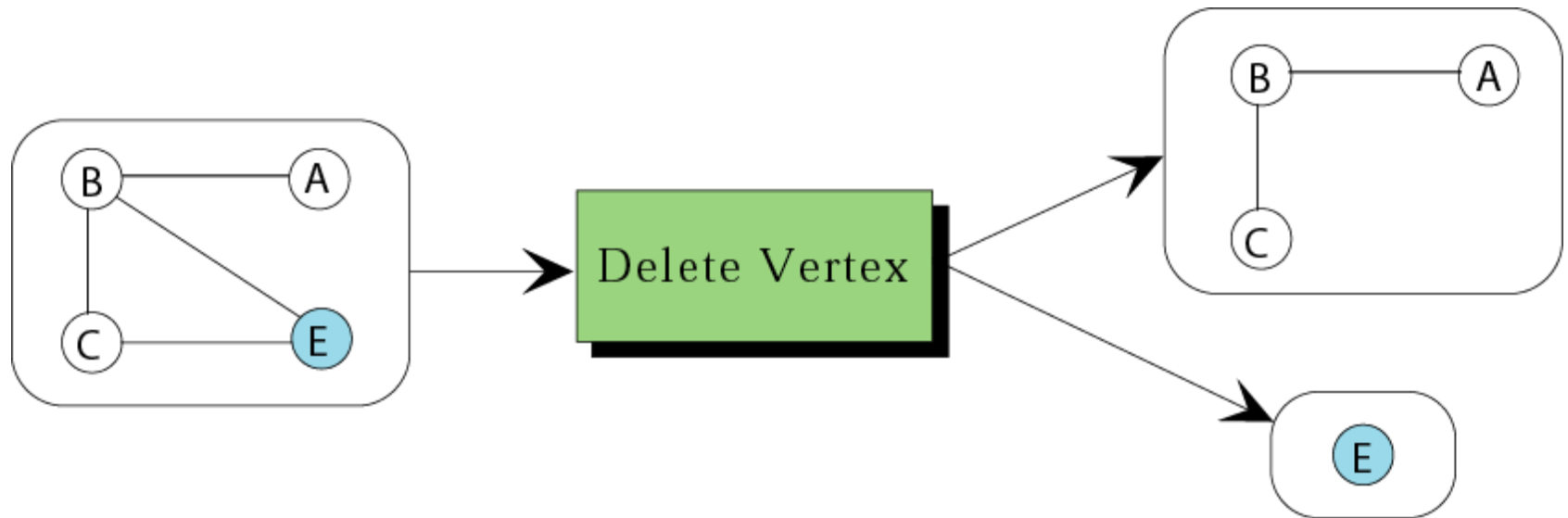


b. Undirected Graph

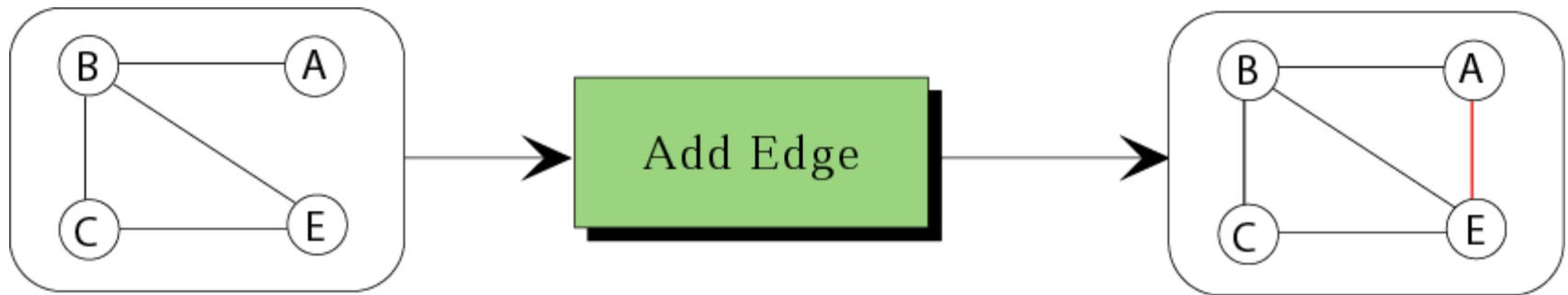
Add vertex



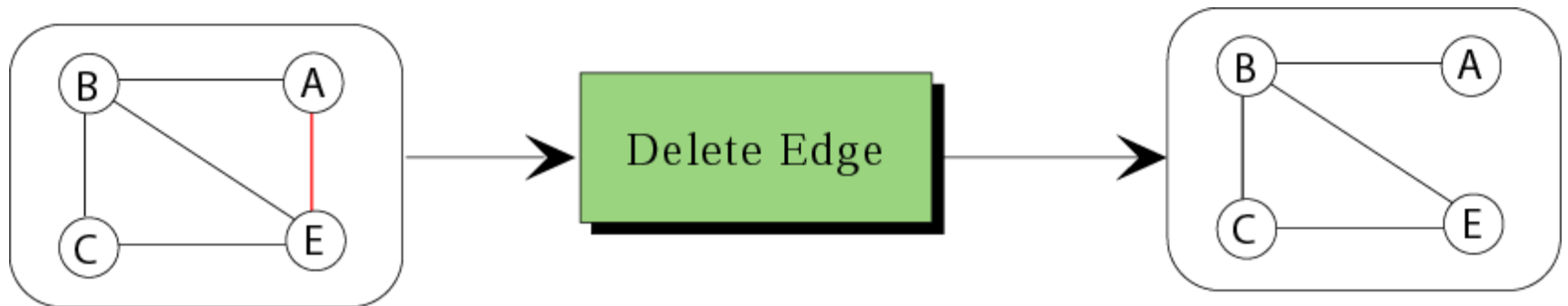
Delete vertex



Add edge



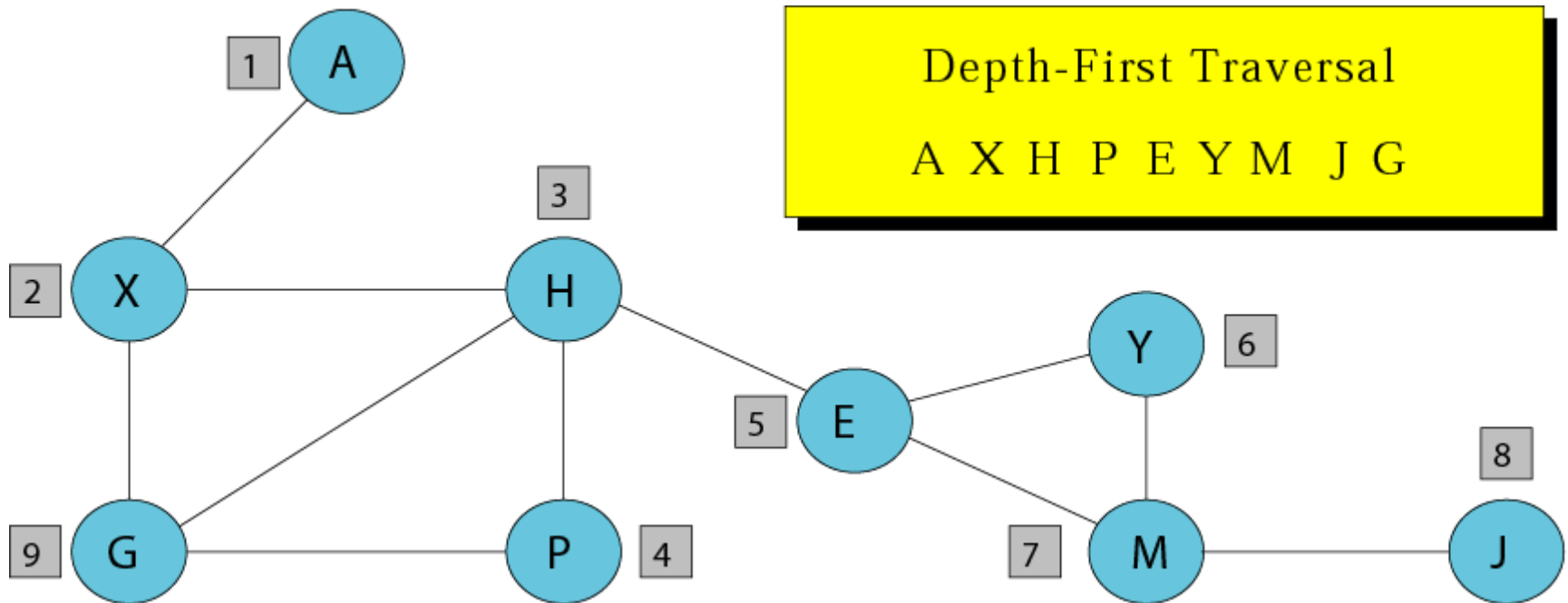
Delete edge



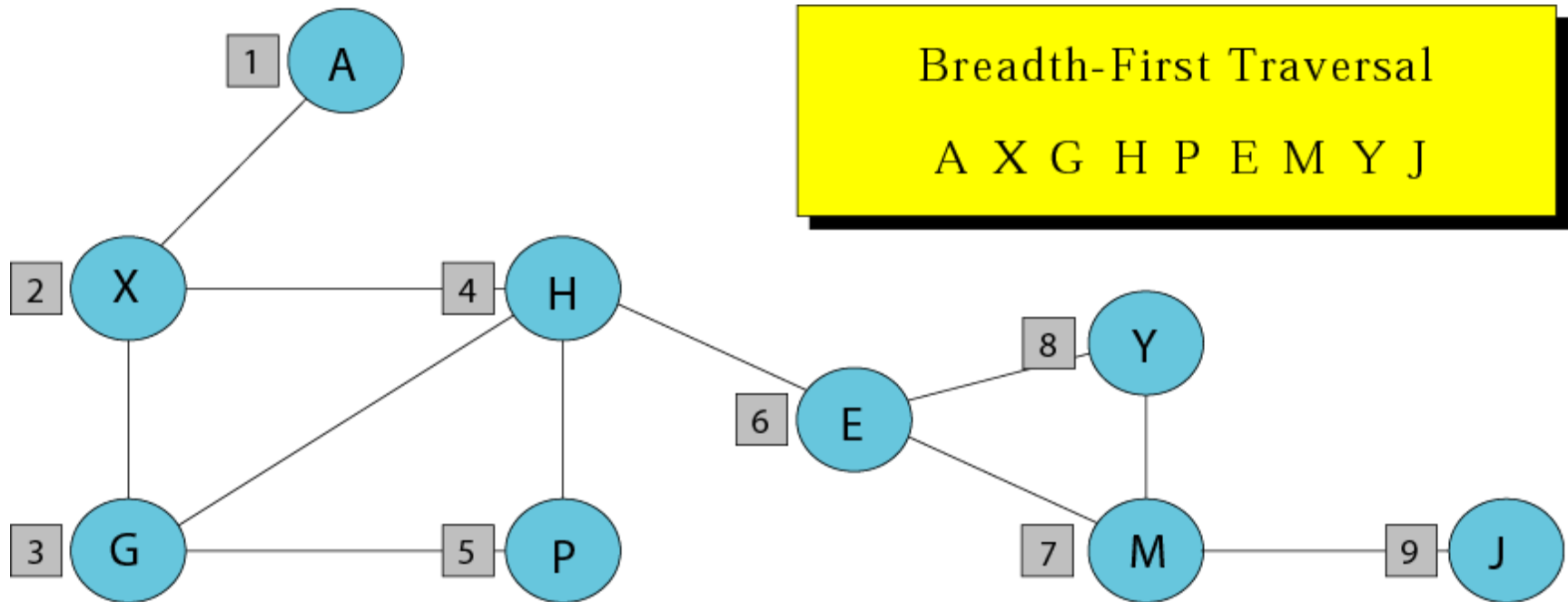
Find vertex



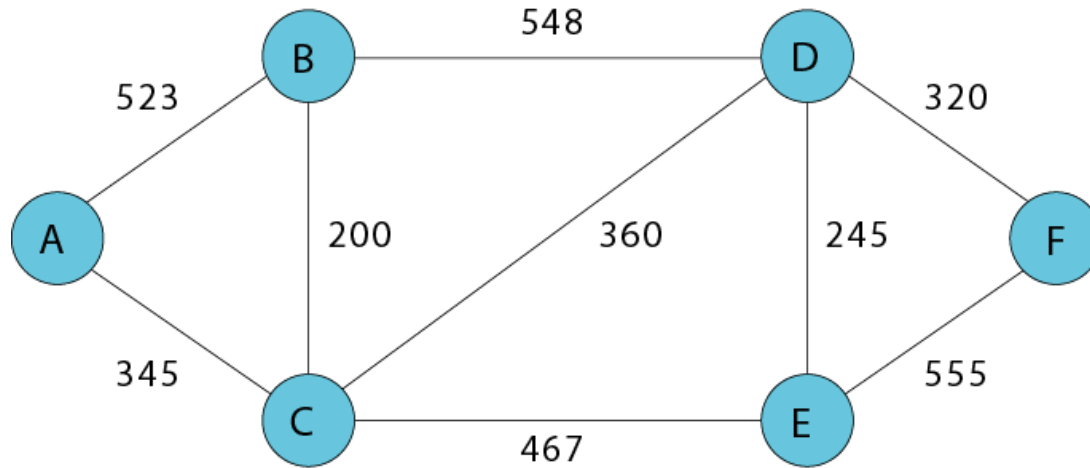
การ Traverse Graph แบบ Depth-First



การ Traverse Graph แบบ Breadth-First



Graph implementations



A
B
C
D
E
F

Vertex Array

	A	B	C	D	E	F
A	0	523	345	0	0	0
B	523	0	200	548	0	0
C	345	200	0	360	467	0
D	0	548	360	0	245	320
E	0	0	467	245	0	555
F	0	0	0	320	555	0

Adjacency Matrix

Graph implementations

