Lists, stacks, queues

Abstract Data Type (ADT): a set of operations; mathematical abstractions (no implementation details)

working definition: = object + operations

= (lists, stacks, queues) + operations on lists, stacks, queues

Lists

$a_1, a_2, a_3, \ldots, a_n$ is a list of size n, where

empty list is of size = 0

pred(a_1) = undefined

succ(a_n) = undefined

operations: printList, makeEmpty, insert, delete, find, next, previous

implementation: arrays, linked lists

Arrays

คุณสมบัติ เรียงติดต่อกัน

เป็นข้อมูลชนิดเดียวกัน (homogeneous)

$a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$

a_1	a_2	a_3	...	a_i	...	a_{n-1}	a_n

ตำแหน่งของ a_i สามารถคำนวณได้จาก

address(a_i) = address(a_1) + (i-1) * sizeof(a)

	10	14	18	22	26	...	a_{n-1}	a_n
address	82	86	90	94	98		...	...
subscript:	0	1	2	3	4		n-2	n-1

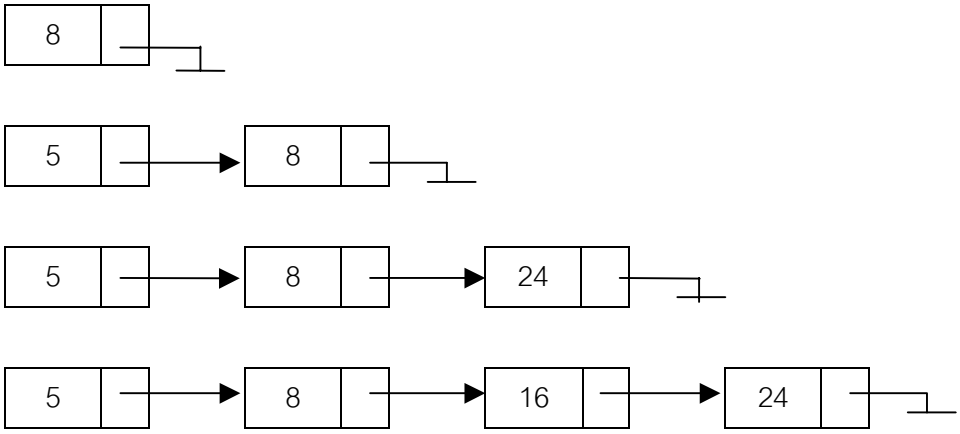
ตำแหน่งที่ 5 = 82 + (5-1) * 4 = 98 (ไม่รวมตัวแรก นับถัดไป 4 ช่อง จึงต้องลบ 1)

หรือใช้ subscript 4 ของ C = 82 + 4 * 4 = 98

Linked lists

มี 2 ชนิด คือ singly linked list และ doubly linked list ในที่นี้จะกล่าวเฉพาะ singly linked list เท่านั้น

Linked list ใช้หลักการใส่ข้อมูลเข้าใน list ตามลำดับที่ควรจะเป็น เช่น สมมุติว่าขณะเริ่มสร้าง linked list ชุดข้อมูลคือ 8, 5, 24, 16 เริ่มอ่านข้อมูลตัวแรก คือ 8 โดยใส่เป็น node แรกของ list ข้อมูลถัดมาเป็น 5 node ใหม่ที่เก็บค่า 5 ก็จะแทรกเข้าด้านหน้าของ node แรกที่เก็บค่า 8 ทำให้กลายเป็น node แรกแทน และเชื่อมต่อกันอย่างอัตโนมัติ ข้อมูลตัวที่สามเป็น 24 ก็จะต่อที่ข้างท้าย ข้อมูลตัวสุดท้าย คือ 16 จะแทรกระหว่าง 8 และ 24 linked list สุดท้ายที่ได้จะเรียงกันเป็นลำดับโดยไม่ต้อง sort แม้แต่น้อย



การ delete ก็ทำในทำนองกลับกัน เช่น ต้องการลบ node ที่มีค่า 16 ก็ traverse list จากต้นไปจนเจอแล้วทำการต่ออ้อมจาก node 8 ข้ามไปยัง node 24 ก่อน แล้วจึง free หน่วยความจำที่ใช้โดย node 16 ทำให้ node 16 ถูกลบทิ้งไปโดยปริยาย ซึ่ง linked list ก็ยังเชื่อมต่อกันในลักษณะเรียงตามลำดับเหมือนเดิม

ตัวอย่าง source code C (ดูใน homepage llist.pdf)

skip: -polynomial เพราะสมมุติกำลังเท่ากันใน element (ช่อง) ที่ตรงกัน

-radix sort ยกไปอธิบายในบทที่ 7

-3.2.8

Stacks

เป็น data structure (ADT) ที่มีลักษณะต่างจาก data structure ชนิดอื่นๆ คือ ทำงานแบบมาทีหลังออกก่อน (Last-in, First-out หรือ LIFO) (เช่นเดียวกับการหยิบถาดอันบนสุดจากกองถาดที่วางซ้อนกัน)

Operations: ได้แก่ push, pop

การทำงานของ stack อาศัย pointer ที่เรียกว่า top-of-stack (TOS) เป็นตัวบอกตำแหน่งที่จะใส่ (push) ข้อมูลลงไป หรือดึง (pop) ข้อมูลกลับขึ้นมาโดยขยับ TOS ขึ้นลง ซึ่งจะเป็นไปตามขั้นตอนดังนี้

สมมุติให้ stack เริ่มจากล่างขึ้นบน (เช่นเดียวกับการวางของซ้อนๆ กันบนพื้น ซึ่งจะค่อยๆ สูงขึ้นเรื่อยๆ) โดยตำแหน่งปกติ TOS จะชี้ที่ช่องว่างเสมอ ขณะเริ่มทำงาน TOS จะชี้ที่ว่าง (พื้น) เสมอ

push: จะใส่ข้อมูลลงในช่องที่ TOS ชี้อยู่ แล้วขยับ TOS ขึ้นไปชี้ที่ช่องว่างถัดไป

pop: จะขยับ TOS ลง 1 ช่อง (เพราะขณะนั้น TOS ชี้ไปที่ช่องว่างอยู่) เพื่อชี้ที่ช่องข้อมูลบนสุด ดึงข้อมูลออก ช่องนั้นก็จะว่าง ซึ่งก็เท่ากับว่า TOS ชี้ที่ช่องว่างเสมอ

stack มีการใช้งานอย่างกว้างขวาง โดยเฉพาะในการเขียน postfix notation (Reverse Polish Notation) การทำ function call (โดยสร้าง stack frame สำหรับ parameters) หรือการประยุกต์ในตัวแปลภาษา (compiler) เป็นต้น

ตัวอย่าง หน้า 72-74 และ 75-76

implementation: สามารถใช้ array หรือ singly linked list ในการจำลอง stack และการทำงานได้

Queues

เป็น data structure (ADT) ที่มีรูปแบบการทำงานง่ายๆ คือ ทำงานแบบมาก่อนออกก่อน (First-in, First-out หรือ FIFO) (เช่นเดียวกับการเข้าแถวซื้อตั๋ว)

Operations: ได้แก่ enqueue, dequeue

การจำลองสถานการณ์ของ queue นิยมใช้ array (จะใช้ singly linked list ก็ได้) โดยมี pointer 2 อันเป็นตัวบอกตำแหน่งที่จะใส่หรือดึงข้อมูลจาก queue คือ front และ rear (ดูตัวอย่างในหน้า 80-81 และใน PowerPoint) ซึ่งมีหลักการทำงานอย่างง่ายๆ ดังนี้

> การเพิ่มข้อมูลจะเริ่มจากด้าน front (เดิม front ชี้ตรงช่องว่าง ใส่ข้อมูลตรงที่ชี้ แล้วขยับ front ไปชี้ช่องว่างถัดไป) ส่วนการดึงข้อมูลก็ดึงออกทาง rear (เดิม rear ชี้ตรงช่องว่าง ขยับไปชี้ช่องถัดไปซึ่งมีข้อมูล แล้วดึงข้อมูลออกมา ช่องนั้นก็จะว่าง)

มีข้อน่าสังเกตุเกี่ยวกับ implementation ข้างต้น คือ กรณีที่ข้อมูลเต็ม queue และกรณีที่ queue ว่าง สำหรับกรณีแรก ใช้ความสัมพันธ์ที่ว่า full: front = rear ส่วนกรณีหลังจะเป็น empty: front = rear – 1 (หนังสือแต่ละเล่มอาจมีรายละเอียดการ implement ที่ต่างกันในทั้ง 2 กรณี) สาเหตุที่ต้อง

กำหนดเช่นนี้ เพื่อป้องกันไม่ให้ pointer ทั้ง 2 ตัววิ่ง**แซง**กัน ทำให้เกิดปรากฏการณ์ที่เรียกว่า underflow (การพยายามดึงข้อมูลออกจาก queue ทั้งๆ ที่ queue ว่าง) และ overflow (การพยายามบันทึกข้อมูลซ้อนกัน ทั้งๆ ที่ queue เต็ม) อีกทั้งยังเป็นการป้องกันไม่ให้เกิดสภาพกำกวม กล่าวคือ หาก front และ rear มาชี้ที่ช่องเดียวกันแล้ว เราไม่สามารถบอกได้ว่าขณะนั้น queue เต็มหรือว่าง จึงจำเป็นต้องมีการกำหนดนิยามของ full ละ empty ให้ชัดเจน ลักษณะที่กล่าวมานี้เป็น queue พิเศษแบบหนึ่งที่รู้จักกันในนามของ circular queue

implementation: มักนิยมใช้ array ในการจำลอง queue (โดยใช้ pointer 2 ตัวคือ front และ rear ช่วย) เพราะนอกจากจะง่ายต่อการเขียนโปรแกรมแล้ว ยังง่ายต่อการปรับเปลี่ยนรูปแบบของ queue เช่น เปลี่ยนจาก queue ธรรมดาเป็น circular queue ซึ่งทำได้ยากกว่าหาก implement โดย linked list

คำถามชวนคิด

เป็นไปได้หรือไม่ที่จะใช้ stack มาจำลองการทำงานของ queue พูดง่ายๆ คือ เอา LIFO ADT มาทำเป็น FIFO ADT