Python 자료구조 알고리즘

2019.12.28

강진혁

ROBOT MEDIA LABORATORY

목차

Robot Media Laboratory

ROBOT MEDIA LABORATORY

자료구조
-리스트-

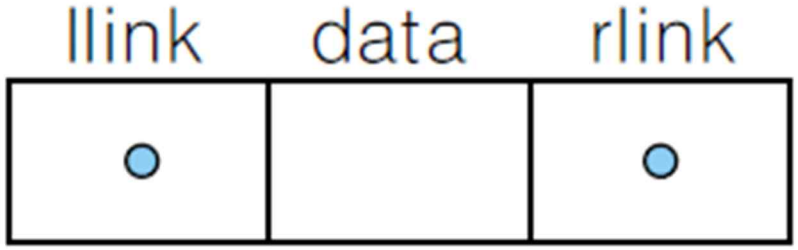
ROBOT MEDIA LABORATORY

선형 리스트의 저장

- 원소들의 논리적 순서와 같은 순서로 메모리에 저장
- 순차 자료구조
  - 원소들의 논리적 순서 = 원소들이 저장된 물리적 순서

이중 연결 리스트(doubly linked list)

- 양쪽 방향으로 순회할 수 있도록 노드를 연결한 리스트
- 이중 연결 리스트의 노드 구조
  - 두 개의 링크 필드와 한 개의 데이터 필드로 구성
  - llink(left link) 필드 : 왼쪽노드와 연결하는 포인터
  - rlink(right link) 필드 : 오른쪽 노드와 연결하는 포인터



순차 자료구조의 문제점

- 삽입연산이나 삭제연산 후에 연속적인 물리 주소를 유지하기 위해서 원소들을 이동시키는 추가적인 작업과 시간 소요
  - 원소들의 이동 작업으로 인한 오버헤드는 원소의 개수가 많고 삽입・삭제 연산이 많이 발생하는 경우에 성능상의 문제 발생
- 순차 자료구조는 배열을 이용하여 구현하기 때문에 배열이 갖고 있는 메모리 사용의 비효율성 문제를 그대로 가짐
- 순차 자료구조에서의 연산 시간에 대한 문제와 저장 공간에 대한 문제를 개선한 자료 표현 방법 필요

연결 자료구조(Linked Data Structure)

- 자료의 논리적인 순서와 물리적인 순서가 일치하지 않는 자료구조
  - 각 원소에 저장되어 있는 다음 원소의 주소에 의해 순서가 연결되는 방식
    - **물리적인 순서를 맞추기 위한 오버헤드가 발생하지 않음**
  - 여러 개의 작은 공간을 연결하여 하나의 전체 자료구조를 표현
    - **크기 변경이 유연하고 더 효율적으로 메모리를 사용**

- 연결 리스트
  - 리스트를 연결 자료구조로 표현한 구조
  - 연결하는 방식에 따라 단순 연결 리스트와 원형 연결 리스트, 이중 연결 리스트, 이중 원형 연결 리스트

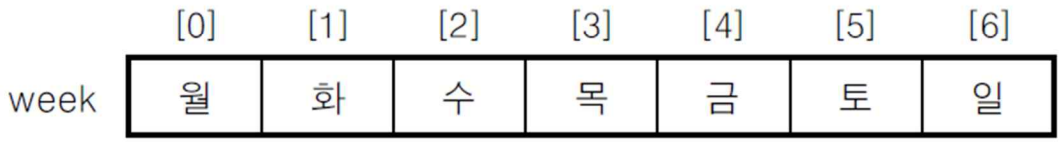
연결 리스트의 노드

- 연결 자료구조에서 하나의 원소를 표현하기 위한 단위 구조
- <원소, 주소>의 구조

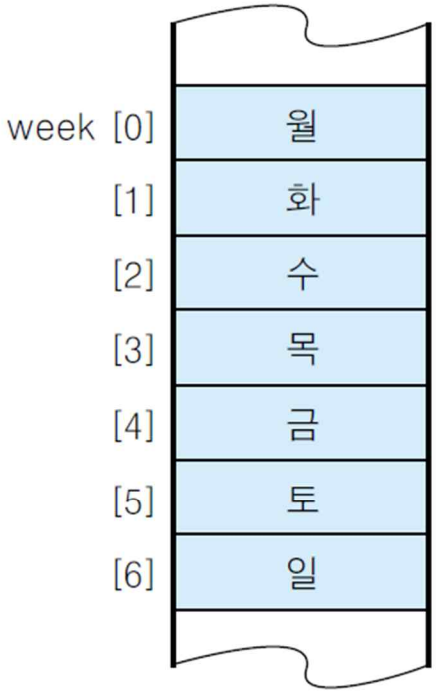
- 데이터 필드(data field)
  - 원소의 값을 저장
  - 저장할 원소의 형태에 따라서 하나 이상의 필드로 구성
- 링크 필드(link field)
  - 다음 노드의 주소를 저장
  - 포인터 변수를 사용하여 주소값을 저장

선형 리스트와 연결 리스트의 비교

- 리스트 week=(월, 화, 수, 목, 금, 토, 일)
- week에 대한 선형 리스트



(a) 논리 구조

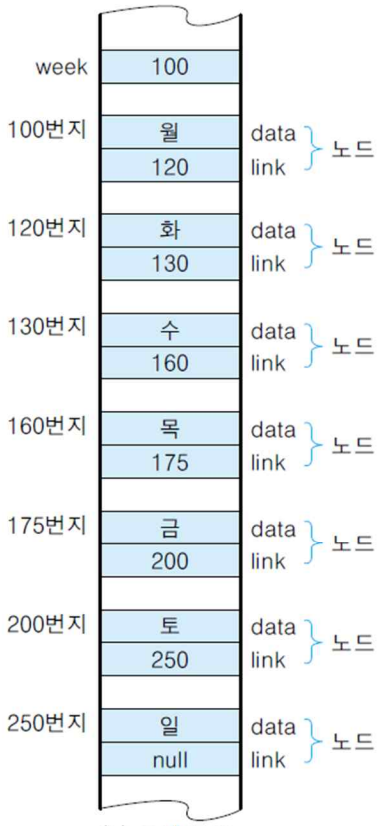


(b) 물리 구조

자료구조

Robot Media Laboratory

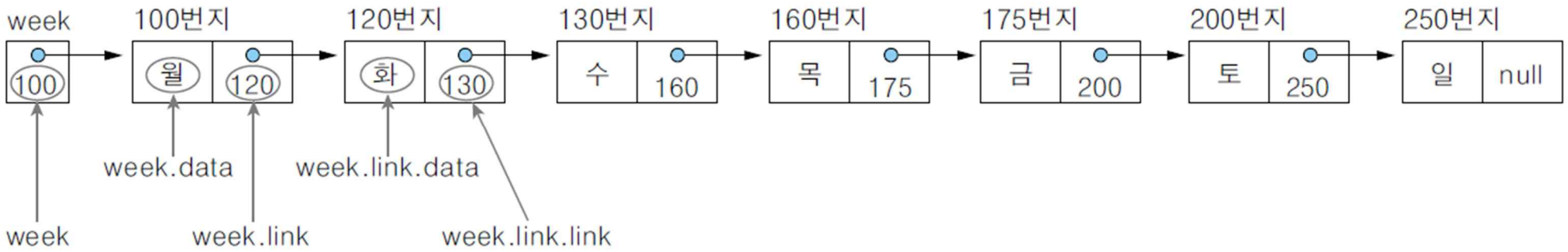
(a) 논리 구조



(b) 물리 구조

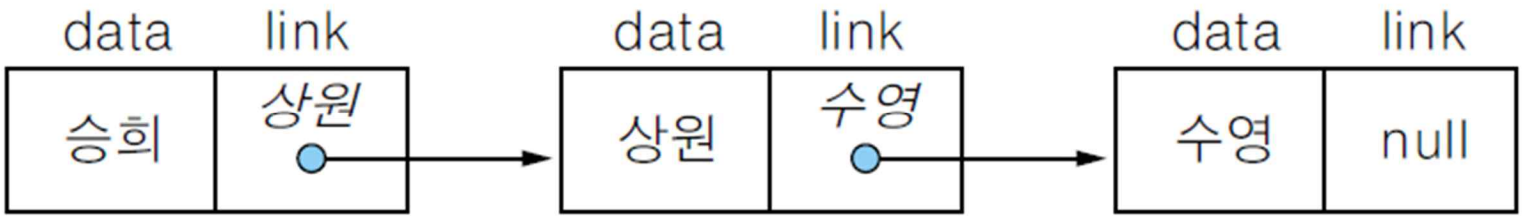
선형 리스트와 연결 리스트의 비교

- **리스트 이름** week - 연결 리스트의 시작을 가리키는 포인터변수
  - 포인터변수 week는 연결 리스트의 첫번째 노드를 가리키는 동시에 연결된 리스트 전체를 의미
- 연결 리스트의 마지막 노드의 링크필드 - 노드의 끝을 표시하기 위해서 **null**(널) 저장
- **공백 연결 리스트** - 포인터변수 week에 null을 저장 (널 포인터)



단순 연결 리스트(singly linked list)

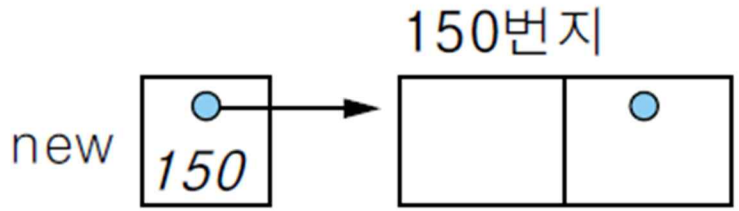
- 노드가 하나의 링크 필드에 의해서 다음 노드와 연결되는 구조를 가진 연결 리스트
- 연결 리스트, 선형 연결 리스트(linear linked list), 단순 연결 선형 리스트(singly linked linear list)
- 예)



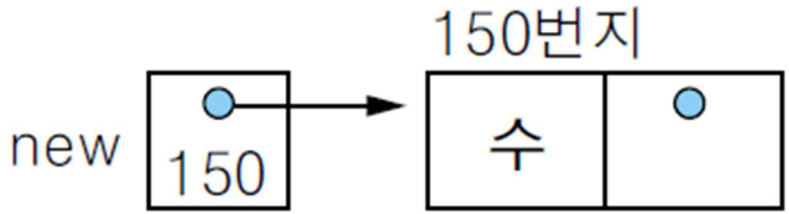
단순 연결 리스트의 삽입

- 리스트 week2=(월, 금, 일)에서 원소 "월"과 "금"사이에 새 원소"수" 삽입하기

① 삽입할 새 노드를 만들 공백노드를 메모리에서 가져와서 포인터변수 new가 가리키게 한다.

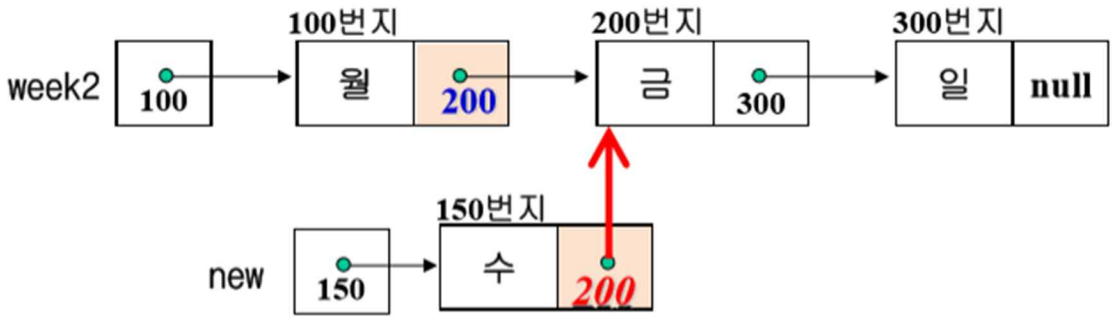


② new의 데이터 필드에 "수"를 저장한다.

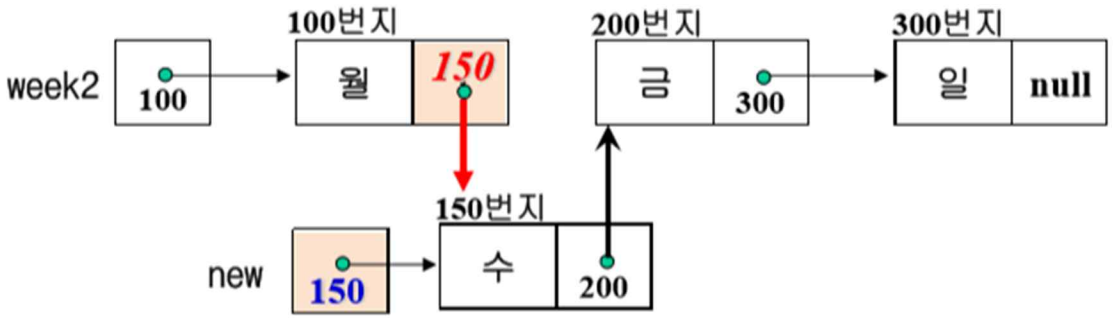


단순 연결 리스트의 삽입

③ new의 앞 노드, 즉 "월"노드의 링크 필드 값을 new의 링크 필드에 저장한다.



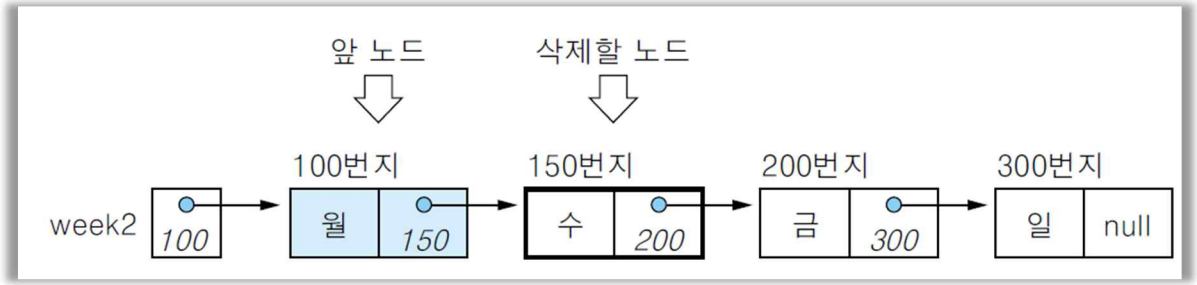
④ new의 값(new가 가리키고 있는 새 노드의 주소)을 "월"노드의 링크 필드에 저장한다



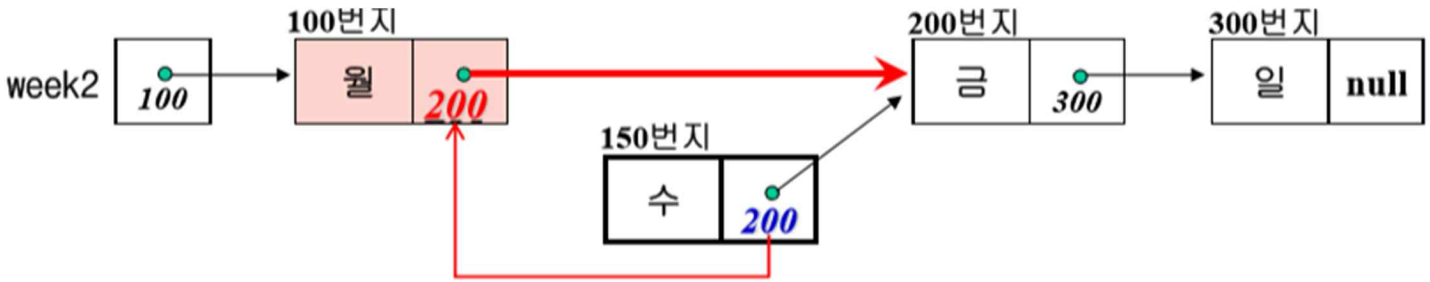
단순 연결 리스트의 삭제

- 리스트 week2=(월, 수, 금, 일)에서 원소 "수" 삭제하기

① 삭제할 원소의 앞 노드(선행자)를 찾는다.

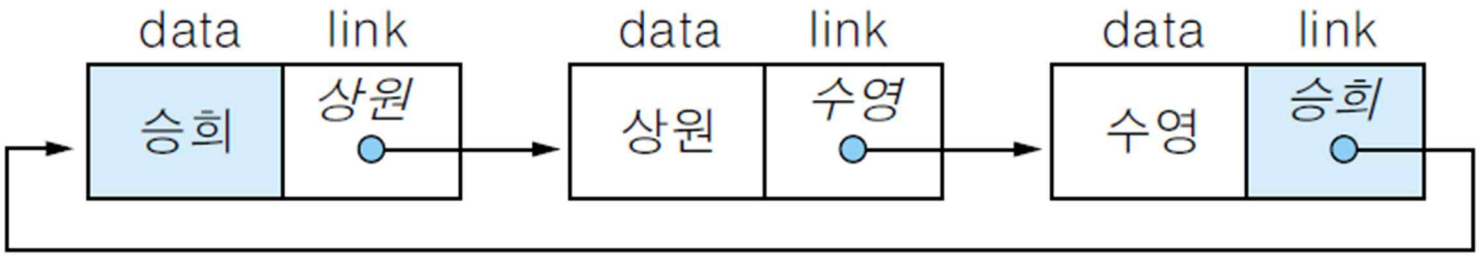


② 삭제할 원소 "수"의 링크 필드 값을 앞 노드의 링크 필드에 저장한다.



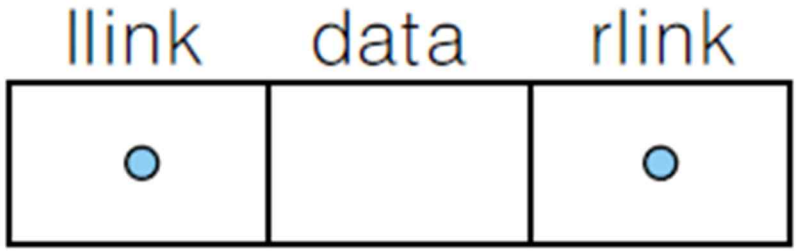
원형 연결 리스트(circular linked list)

- 단순 연결 리스트에서 마지막 노드가 리스트의 첫 번째 노드를 가리키게 하여 리스트의 구조를 원형으로 만든 연결 리스트
  - 단순 연결 리스트의 마지막 노드의 링크 필드에 첫 번째 노드의 주소를 저장하여 구성
  - 링크를 따라 계속 순회하면 이전 노드에 접근 가능

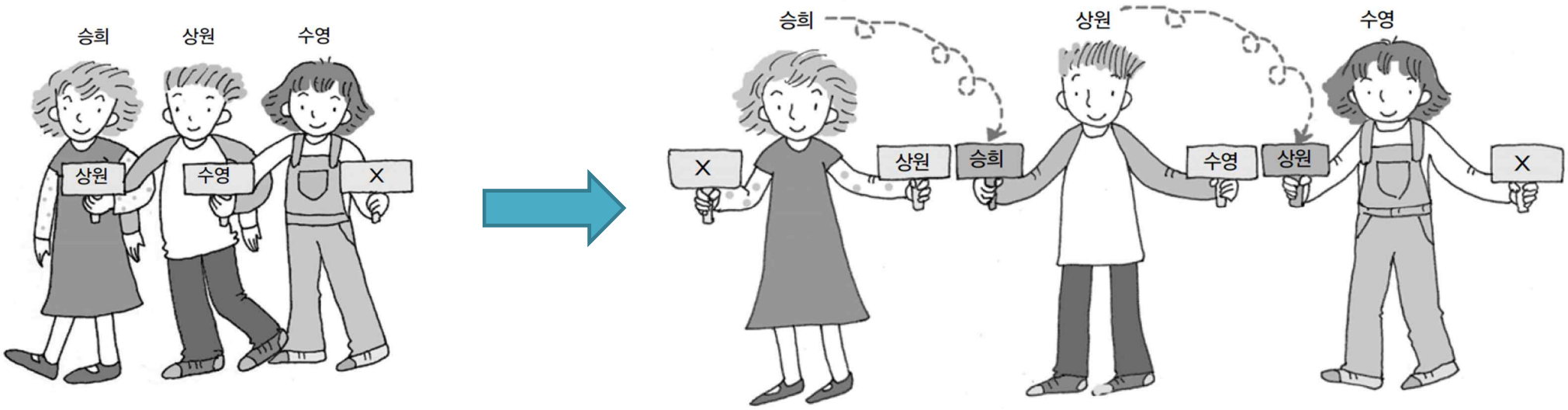


이중 연결 리스트(doubly linked list)

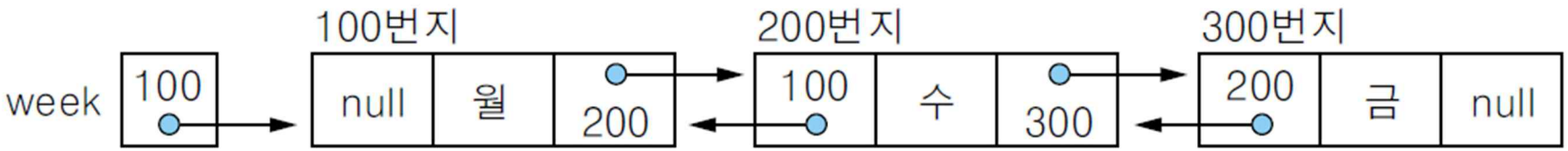
- 양쪽 방향으로 순회할 수 있도록 노드를 연결한 리스트
- 이중 연결 리스트의 노드 구조
  - 두 개의 링크 필드와 한 개의 데이터 필드로 구성
  - llink(left link) 필드 : 왼쪽노드와 연결하는 포인터
  - rlink(right link) 필드 : 오른쪽 노드와 연결하는 포인터



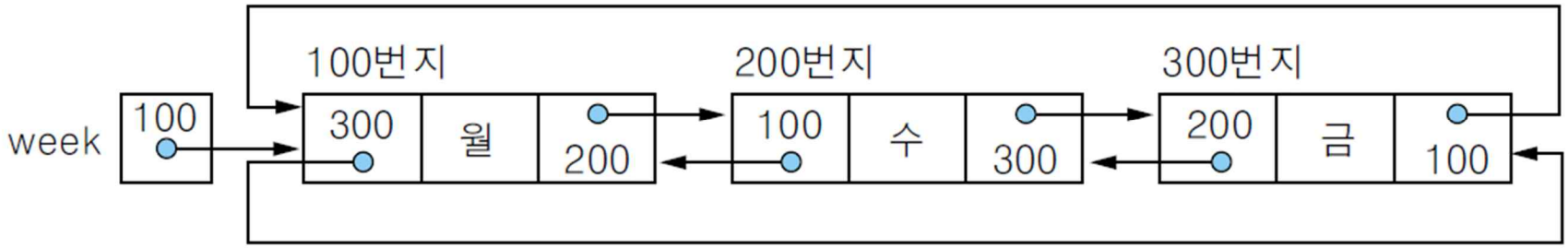
이중 연결 리스트(doubly linked list)



- 리스트 week=(월, 수, 금)의 이중 연결 리스트 구성

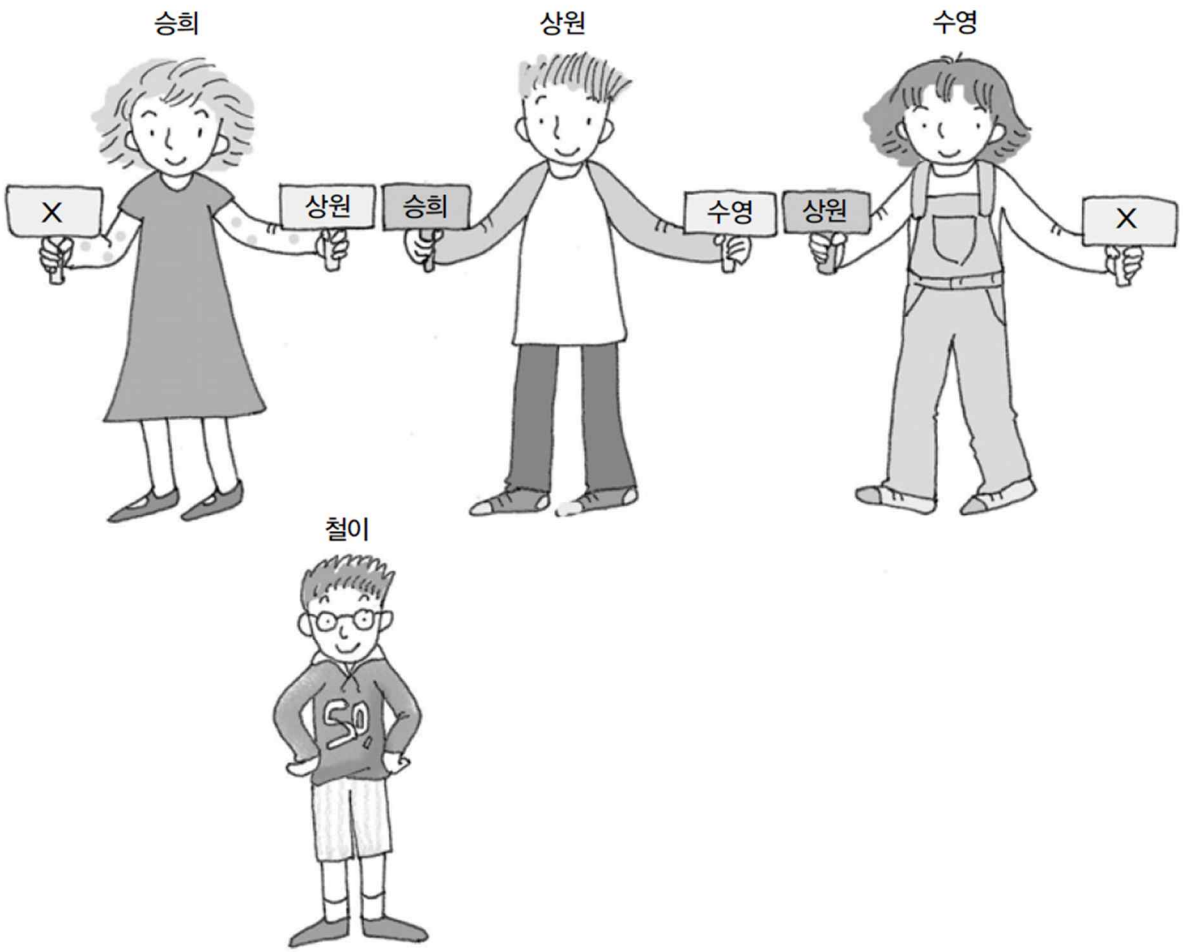


- 원형 이중 연결 리스트
  - 이중 연결 리스트를 원형으로 구성



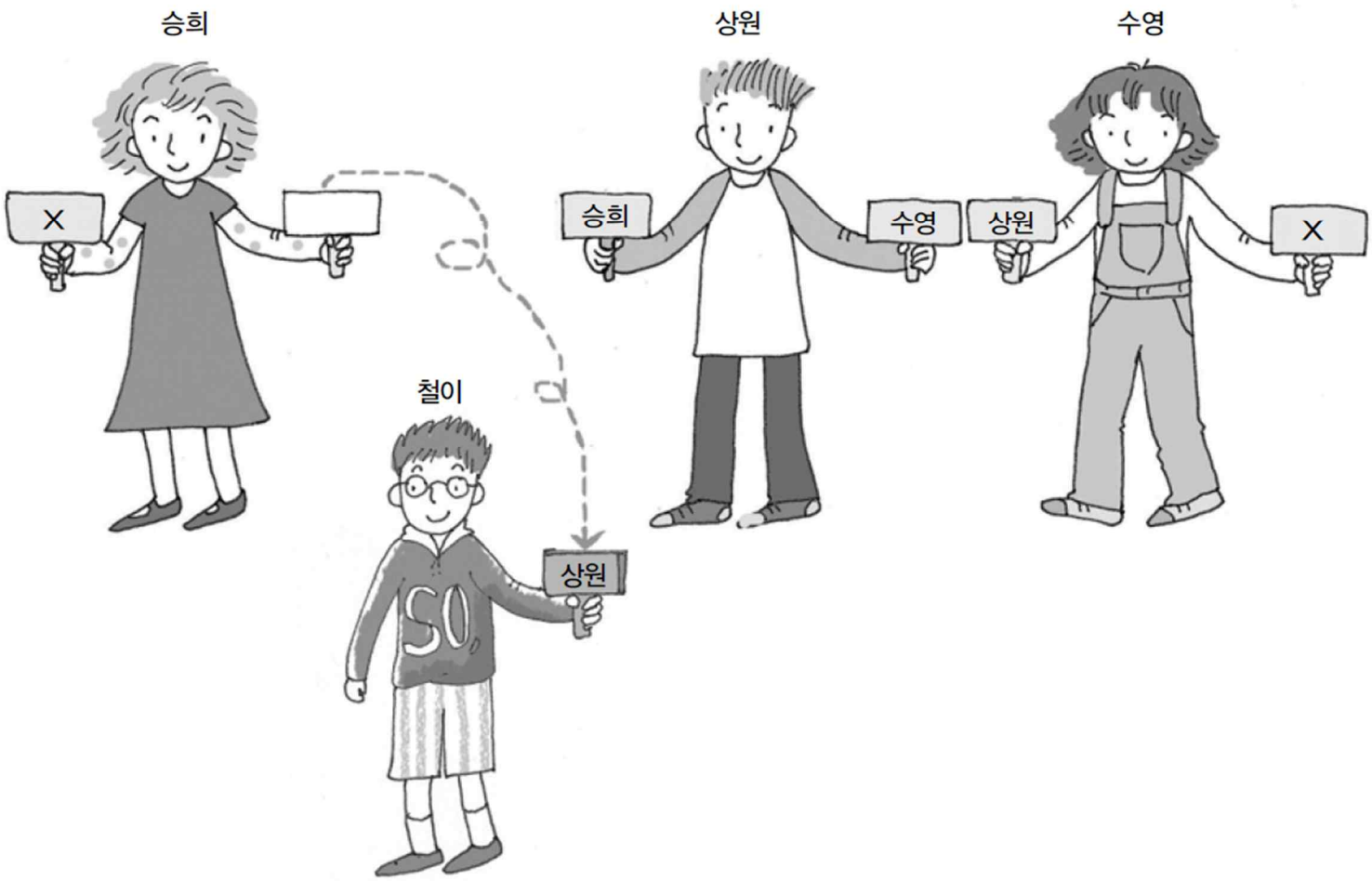
자료구조

Robot Media Laboratory



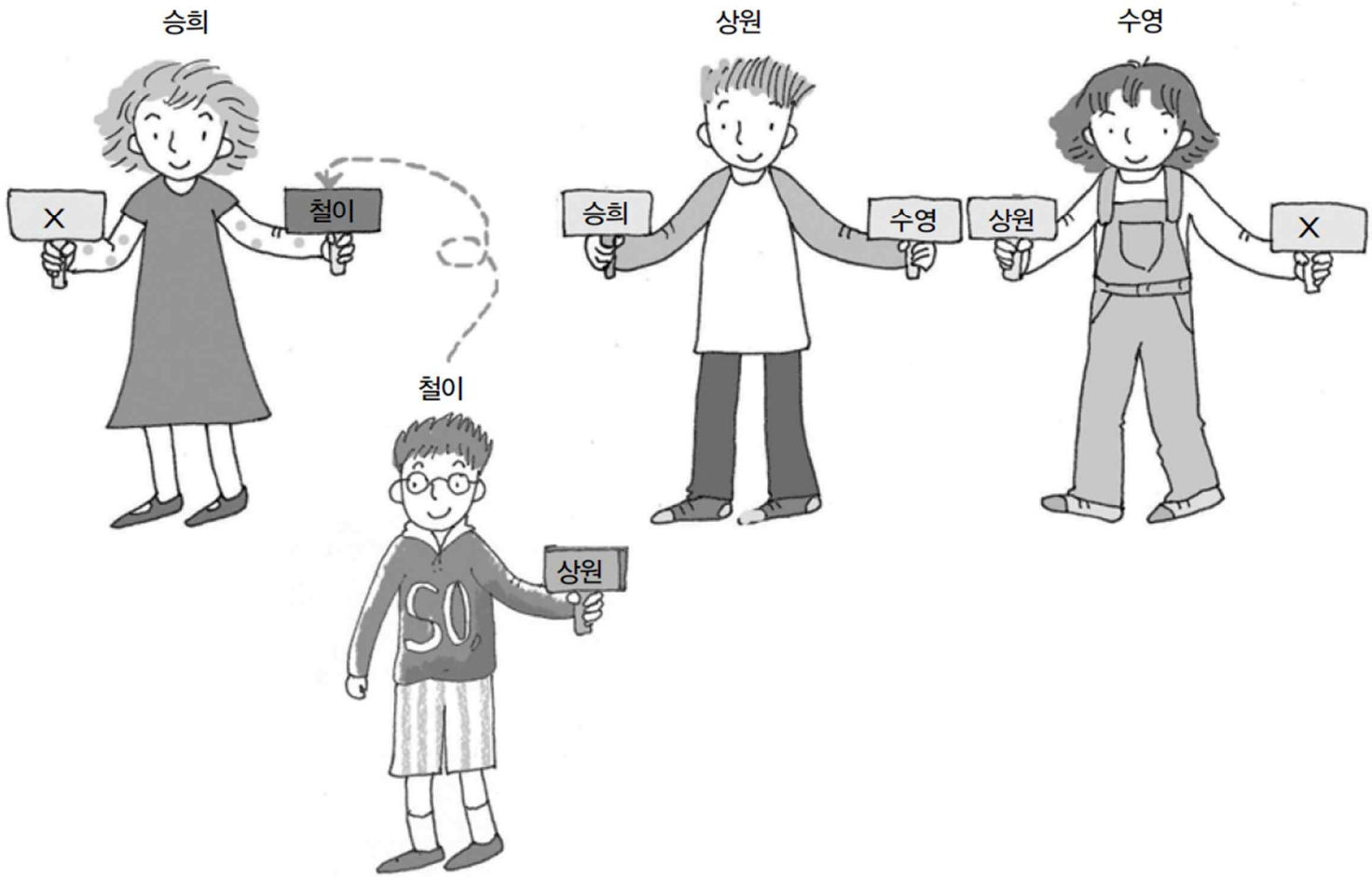
자료구조

Robot Media Laboratory



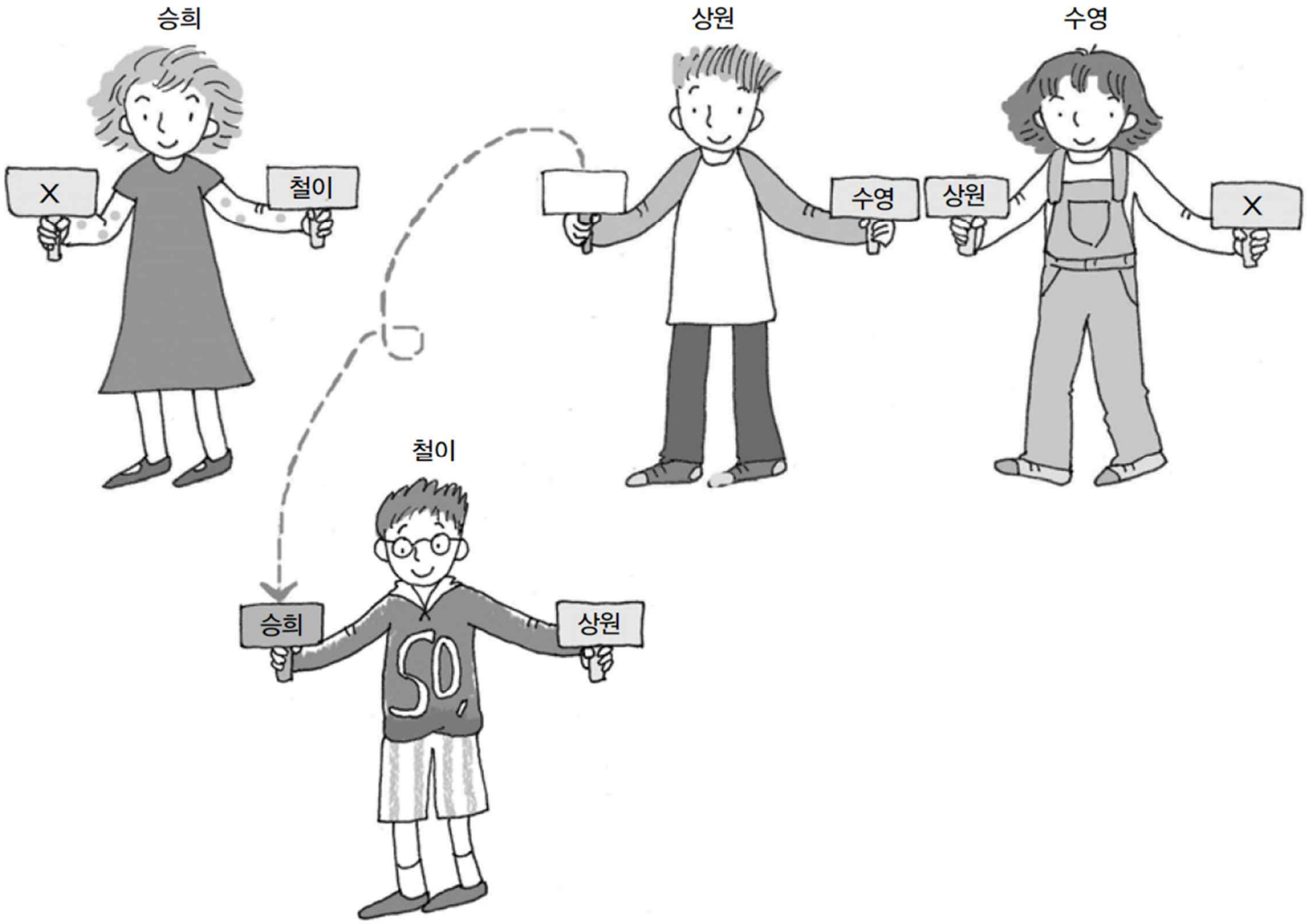
자료구조

Robot Media Laboratory



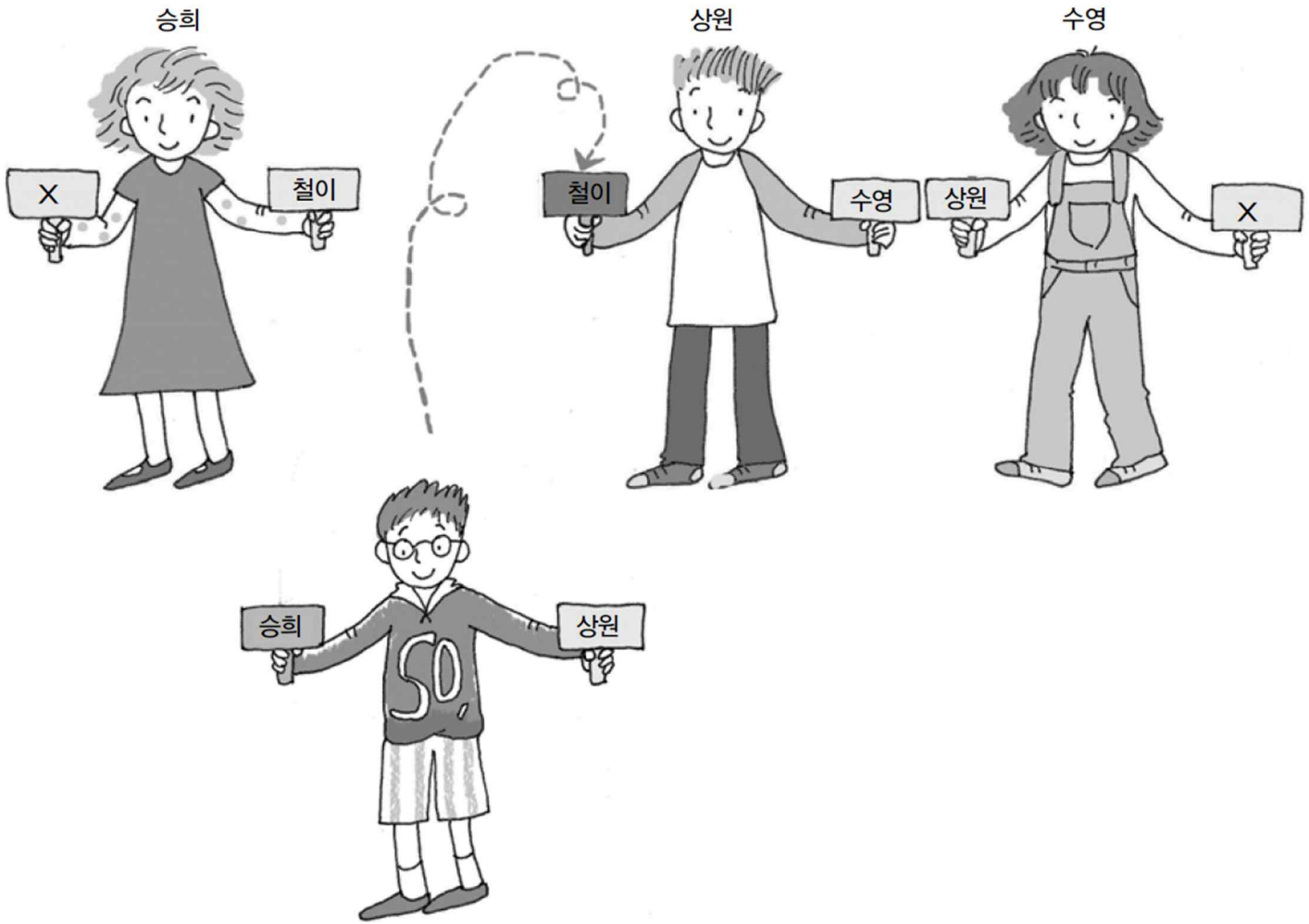
자료구조

Robot Media Laboratory



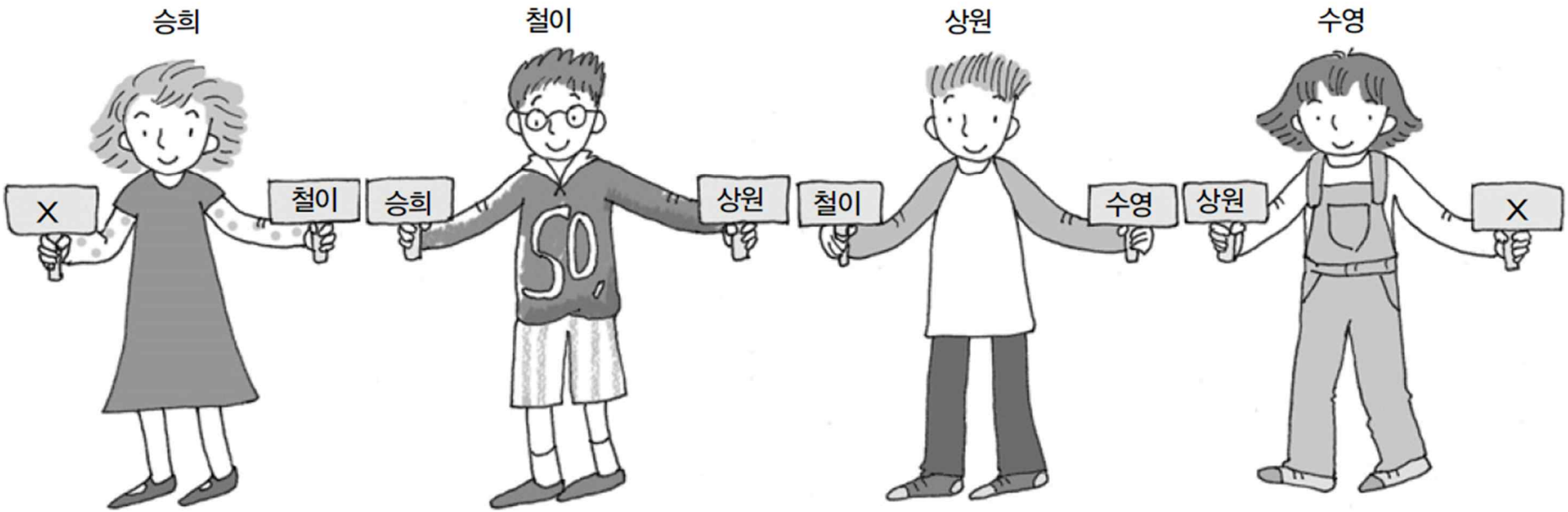
자료구조

Robot Media Laboratory



자료구조

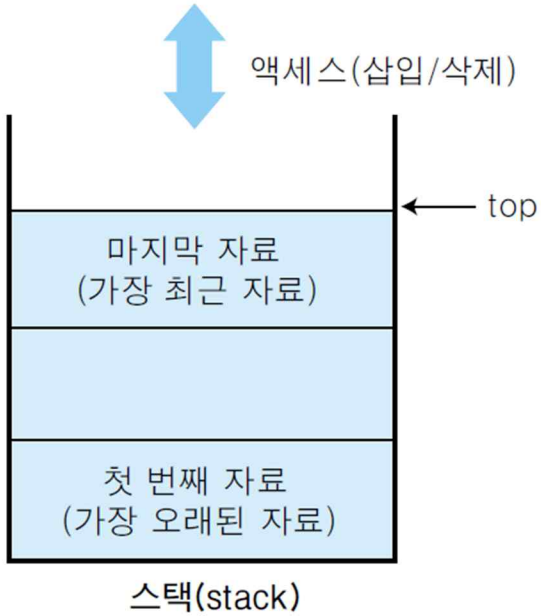
Robot Media Laboratory



자료구조
-스택-

스택(stack)

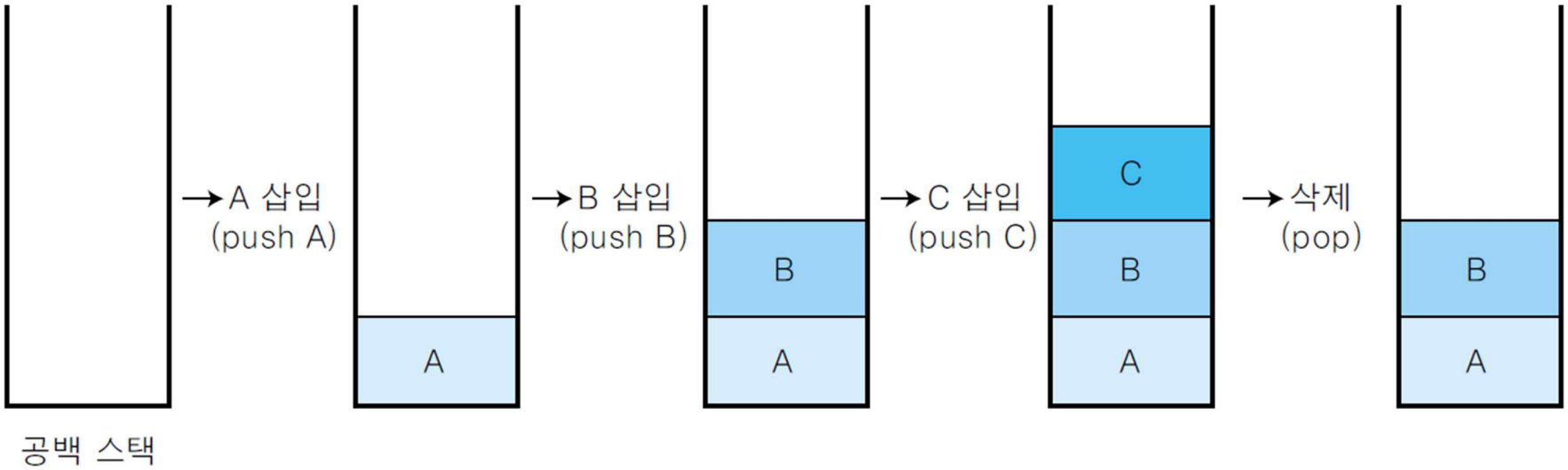
- 접시를 쌓듯이 자료를 차곡차곡 쌓아 올린 형태의 자료구조
- 스택에 저장된 원소는 top으로 정한 곳에서만 접근 가능
  - top의 위치에서만 원소를 삽입하므로, 먼저 삽입한 원소는 밑에 쌓이고, 나중에 삽입한 원소는 위에 쌓이는 구조
  - 마지막에 삽입(**L**ast-**I**n)한 원소는 맨 위에 쌓여 있다가 가장 먼저 삭제(**F**irst-**O**ut)됨 ☞ **후입선출 구조** (**LIFO**, Last-In-First-Out)



스택(stack)

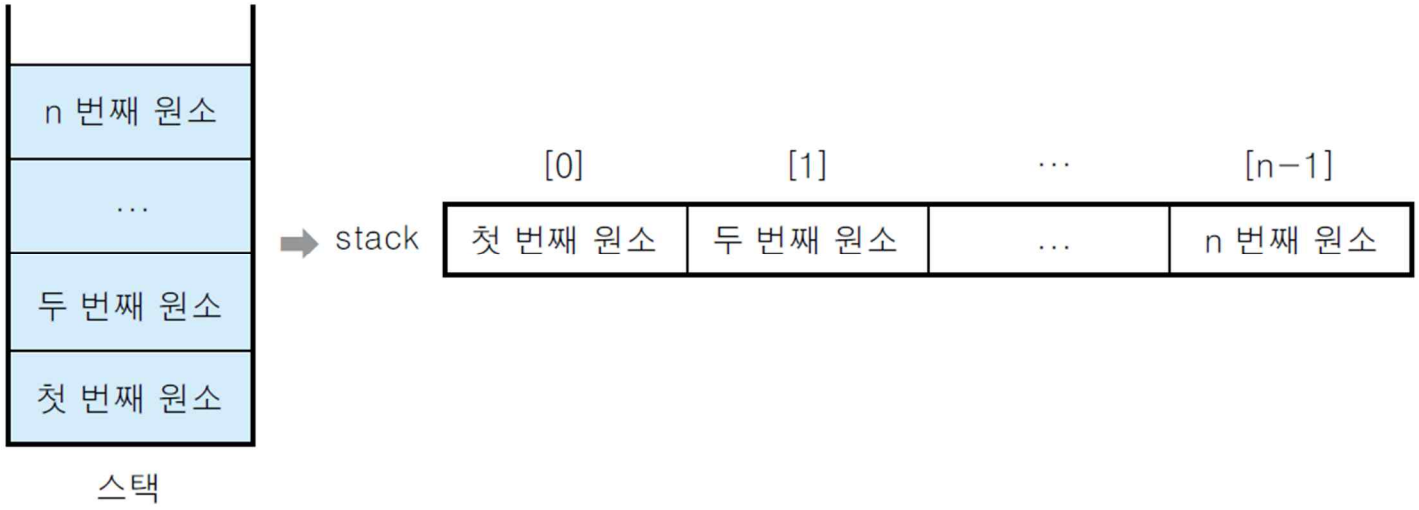
스택의 연산

- 스택에서의 삽입 연산 : push
- 스택에서의 삭제 연산 : pop

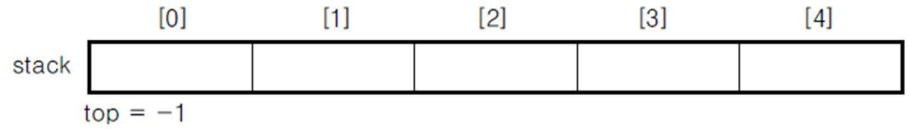


순차 자료구조를 이용한 스택의 구현

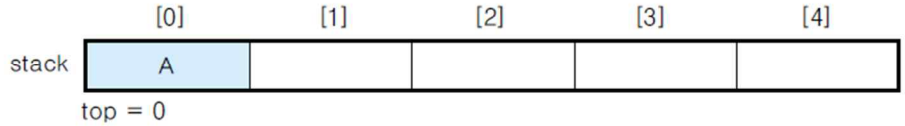
- 순차 자료구조인 1차원 배열을 이용하여 구현
  - 스택의 크기 : 배열의 크기
  - 스택에 저장된 원소의 순서 : 배열 원소의 인덱스
    - 인덱스 0번 : 스택의 첫번째 원소
    - 인덱스 n-1번 : 스택의 n번째 원소
  - 변수 top : 스택에 저장된 마지막 원소에 대한 인덱스 저장
    - 공백 상태 : top = -1 (초기값)
    - 포화 상태 : top = n-1



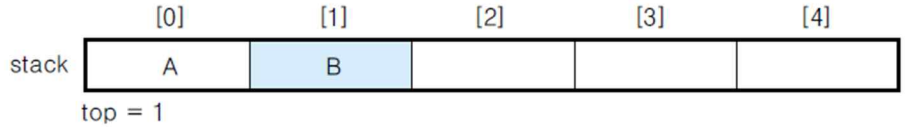
① 공백 스택 생성 : createStack(S, 5) ;



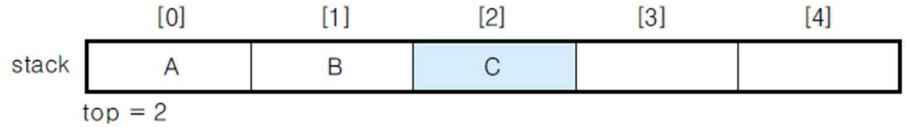
② 원소 A 삽입 : push(S, A);



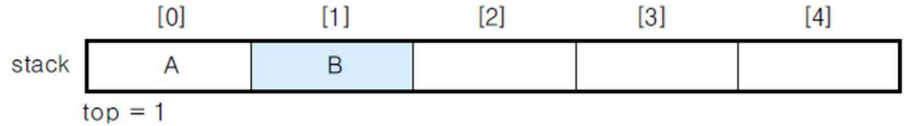
③ 원소 B 삽입 : push(S, B);



④ 원소 C 삽입 : push(S, C);



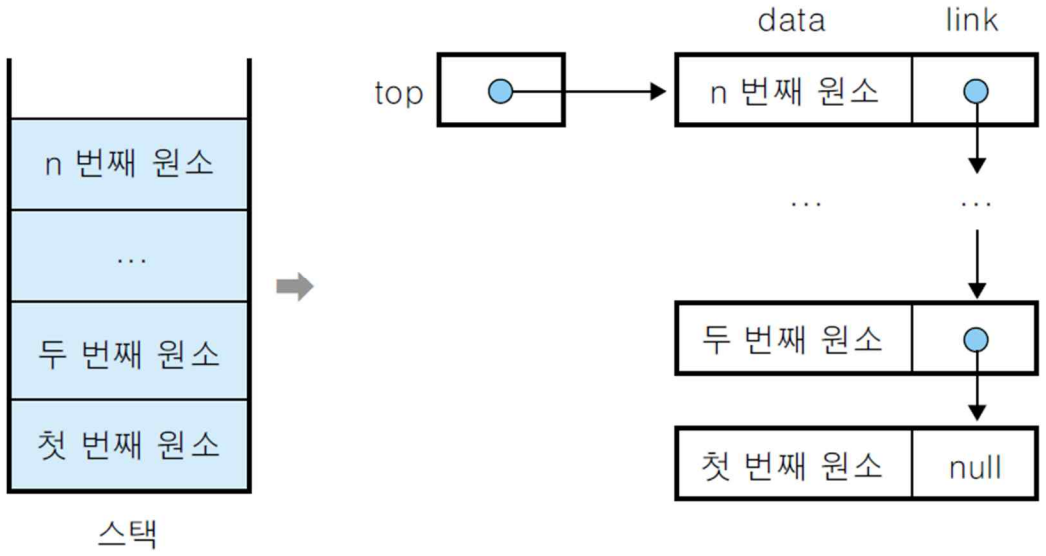
⑤ 원소 삭제 : pop(S);



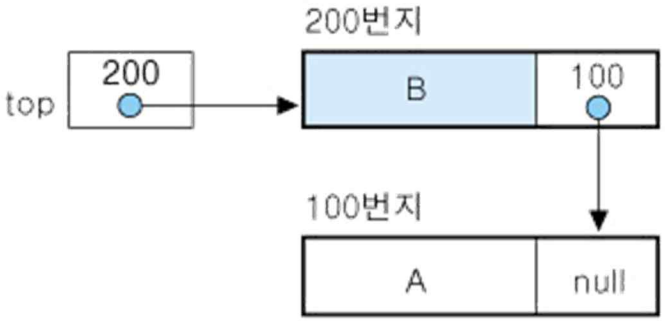
- **순차 자료구조로 구현한 스택의 장점**
  - 순차 자료구조인 1차원 배열을 사용하여 쉽게 구현
- **순차 자료구조로 구현한 스택의 단점**
  - 물리적으로 크기가 고정된 배열을 사용하므로 스택의 크기 변경 어려움
  - 순차 자료구조의 단점을 그대로 가지고 있다.

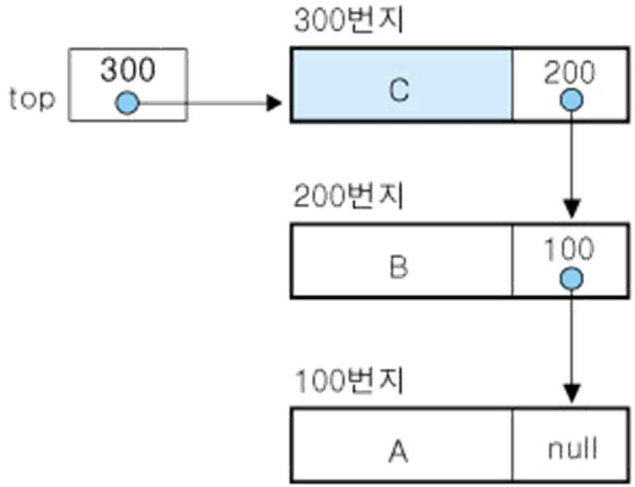
연결 자료구조를 이용한 스택의 구현

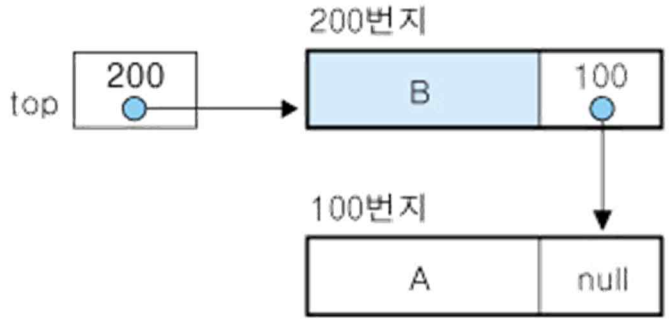
- 단순 연결 리스트를 이용하여 구현
  - 스택의 원소 : 단순 연결 리스트의 노드
    - 스택 원소의 순서 : 노드의 링크 포인터로 연결
    - push : 리스트의 마지막에 노드 삽입
    - pop : 리스트의 마지막 노드 삭제
  - 변수 top : 단순 연결 리스트의 마지막 노드를 가리키는 포인터 변수
    - 초기 상태 : top = null



단순 연결 리스트의 스택의 연산 수행과정

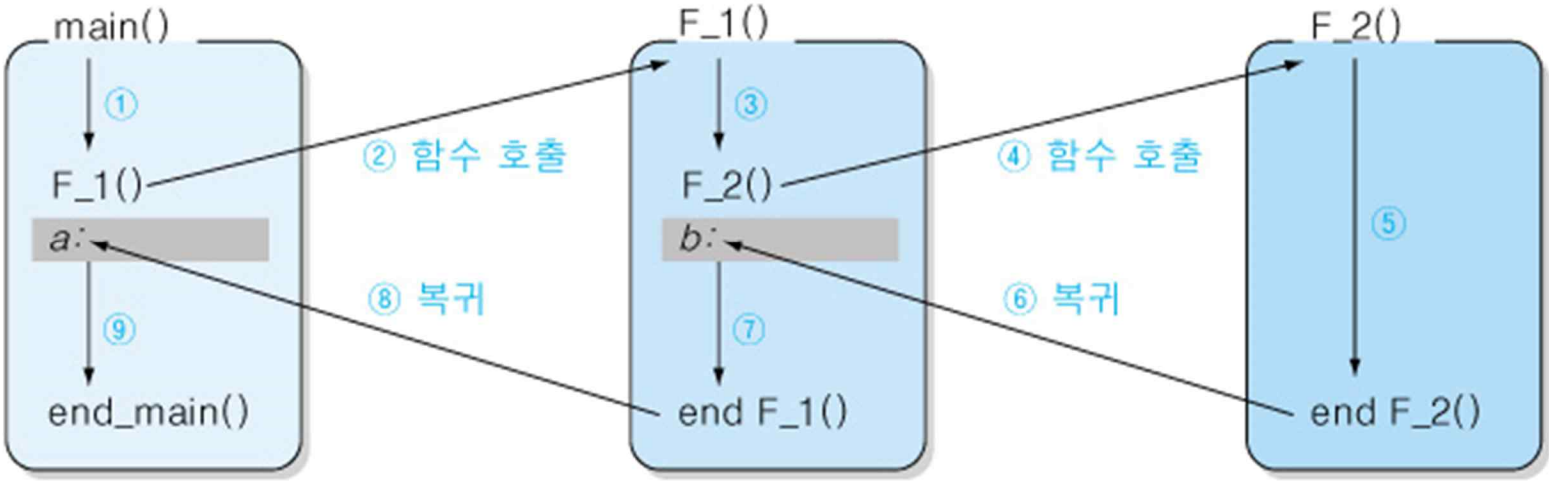






시스템 스택

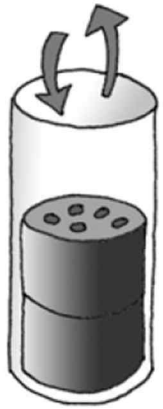
- 프로그램에서의 호출과 복귀에 따른 수행 순서를 관리
  - 가장 마지막에 호출된 함수가 가장 먼저 실행을 완료하고 복귀하는 후입선출 구조이므로, 후입선출 구조의 스택을 이용하여 수행순서 관리
  - 함수 호출이 발생하면 호출한 함수 수행에 필요한 지역변수, 매개변수 및 수행 후 복귀할 주소 등의 정보를 스택 프레임(stack frame)에 저장하여 시스템 스택에 삽입
  - 함수의 실행이 끝나면 시스템 스택의 top 원소(스택 프레임)를 삭제(pop)하면서 프레임에 저장되어있던 복귀주소를 확인하고 복귀
  - 함수 호출과 복귀에 따라 이 과정을 반복하여 전체 프로그램 수행이 종료되면 시스템 스택은 공백스택이 된다.



자료구조
-큐-

큐(Queue)

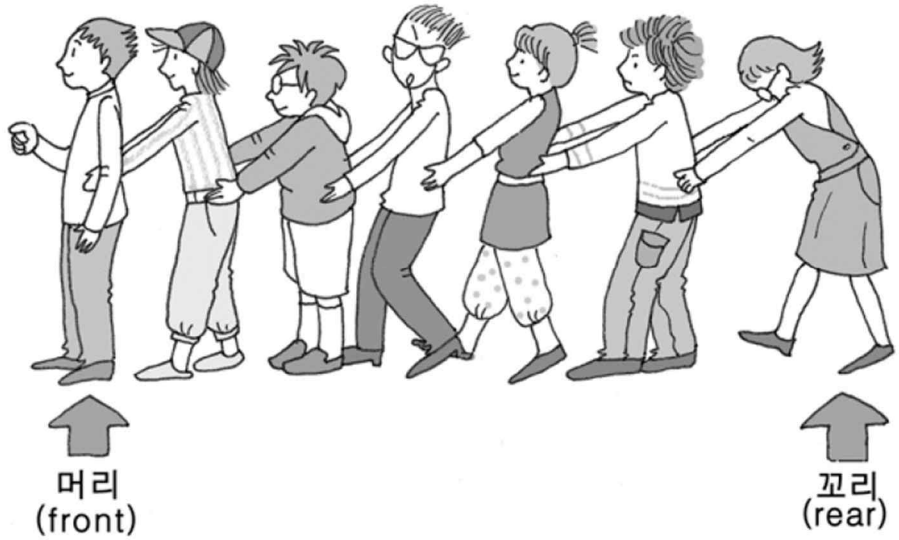
- 스택과 마찬가지로 삽입과 삭제의 위치가 제한되어있는 유한 순서 리스트
- 큐의 뒤에서는 삽입만 하고, 앞에서는 삭제만 할 수 있는 구조
  - 삽입한 순서대로 원소가 나열되어 가장 먼저 삽입(**F**irst-**I**n)한 원소는 맨 앞에 있다가 가장 먼저 삭제(**F**irst-**O**ut)된다.
    ☞ **선입선출 구조** (**FIFO**, First-In-First-Out)
- 스택과 큐의 구조 비교

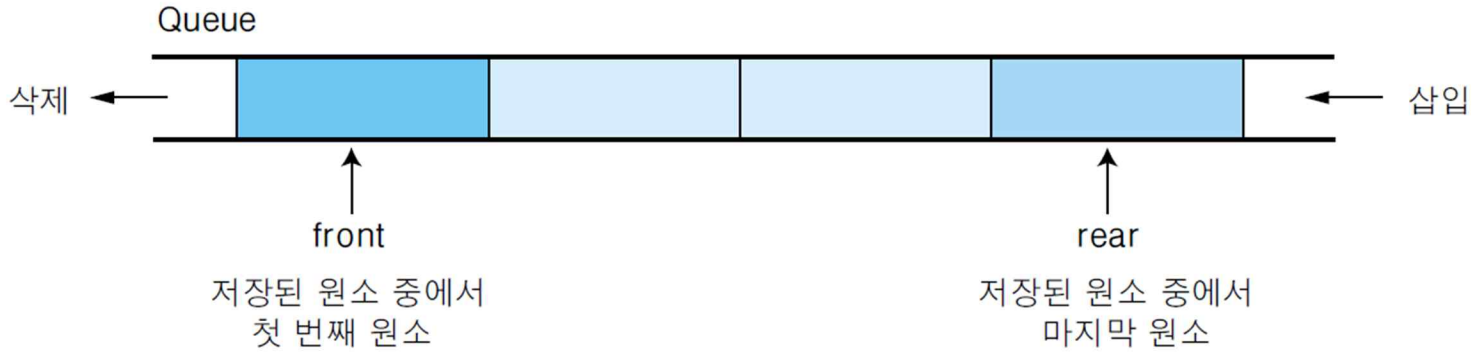


스택

큐

큐의 구조


머리
(front)
꼬리
(rear)


Queue
삭제
삽입
front
저장된 원소 중에서
첫 번째 원소
rear
저장된 원소 중에서
마지막 원소

큐의 연산

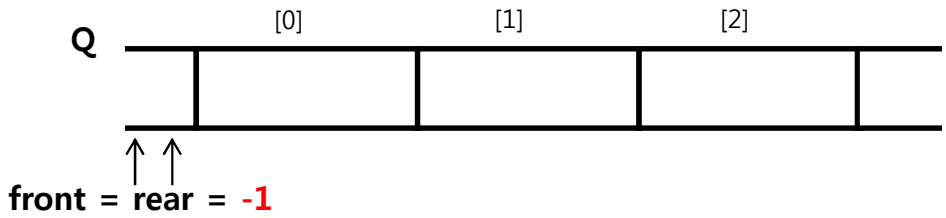
- 삽입 : **enQueue**
- 삭제 : **deQueue**

스택과 큐의 연산 비교

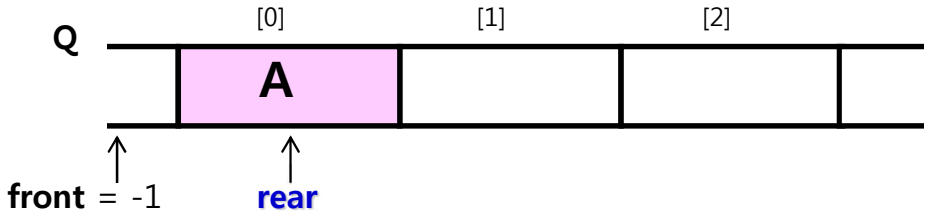
항목 / 자료구조	삽입 연산		삭제 연산	
	연산자	삽입 위치	연산자	삭제 위치
스택	push	top	pop	top
큐	enQueue	rear	deQueue	front

큐의 연산 과정

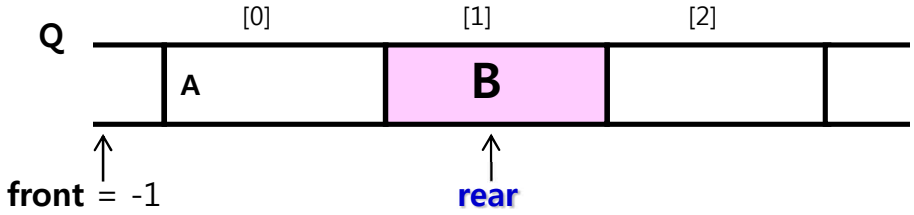
- ① 공백 큐 생성 : createQueue();



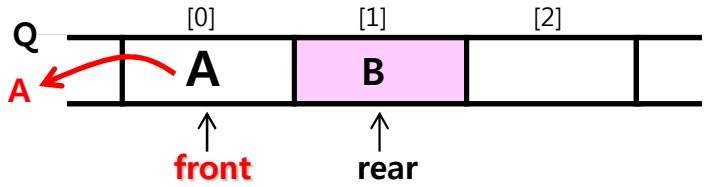
- ② 원소 A 삽입 : enQueue(Q, A);



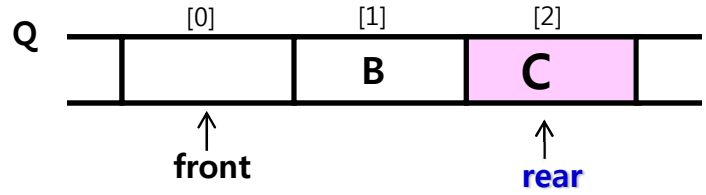
- ③ 원소 B 삽입 : enQueue(Q, B);



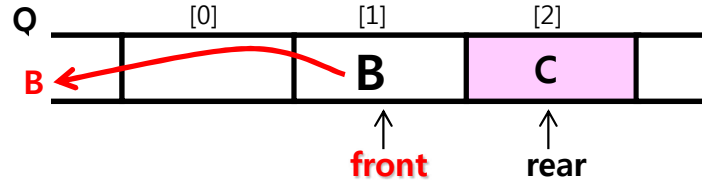
④ 원소 삭제 : deQueue(Q);



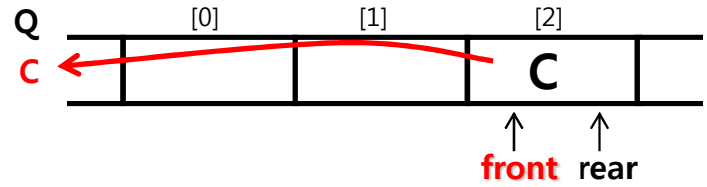
⑤ 원소 C 삽입 : enQueue(Q, C);



⑥ 원소 삭제 : deQueue(Q);



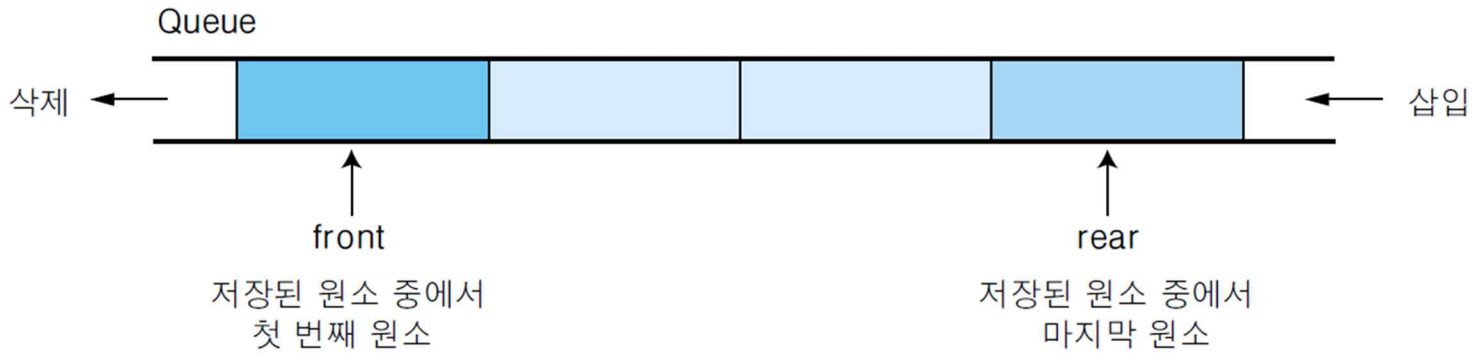
⑦ 원소 삭제 : deQueue(Q);



선형 큐

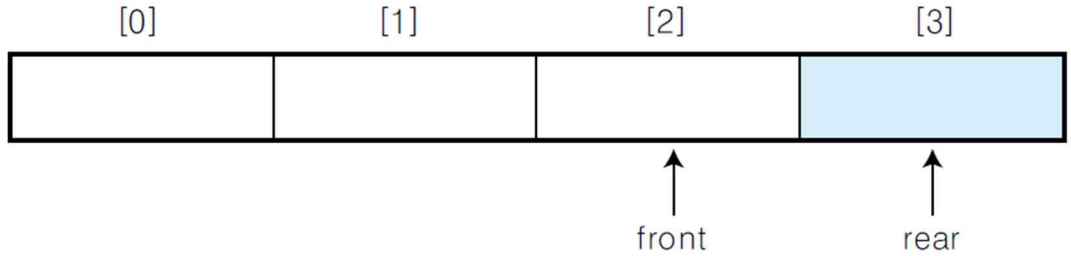
- 1차원 배열을 이용한 큐
  - 큐의 크기 = 배열의 크기
  - 변수 front : 저장된 첫 번째 원소의 인덱스 저장
  - 변수 rear : 저장된 마지막 원소의 인덱스 저장

- 상태 표현
  - 초기 상태 : front = rear = -1
  - 공백 상태 : front = rear
  - 포화 상태 : rear = n-1 (n : 배열의 크기, n-1 : 배열의 마지막 인덱스)



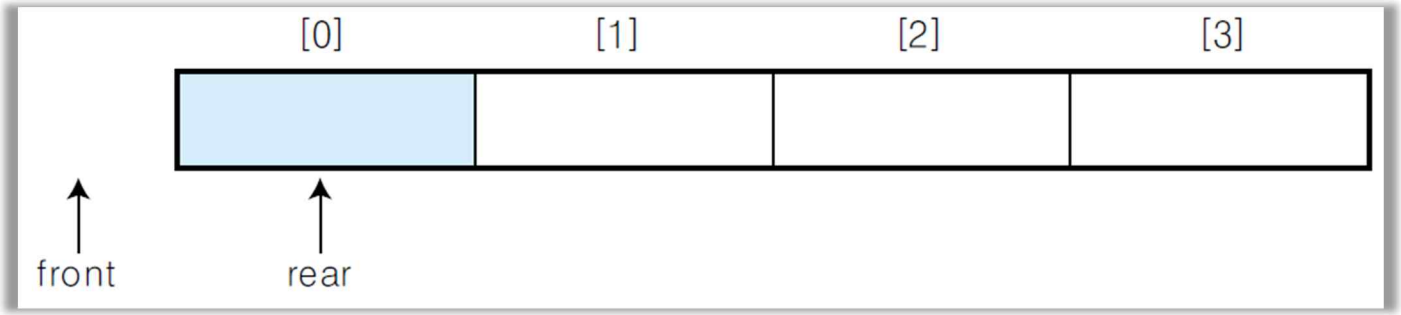
선형 큐의 잘못된 포화상태 인식

- 큐에서 삽입과 삭제를 반복하면서 아래와 같은 상태일 경우, 앞부분에 빈자리가 있지만 rear=n-1 상태이므로 포화상태로 인식하고 더 이상의 삽입을 수행하지 않는다.



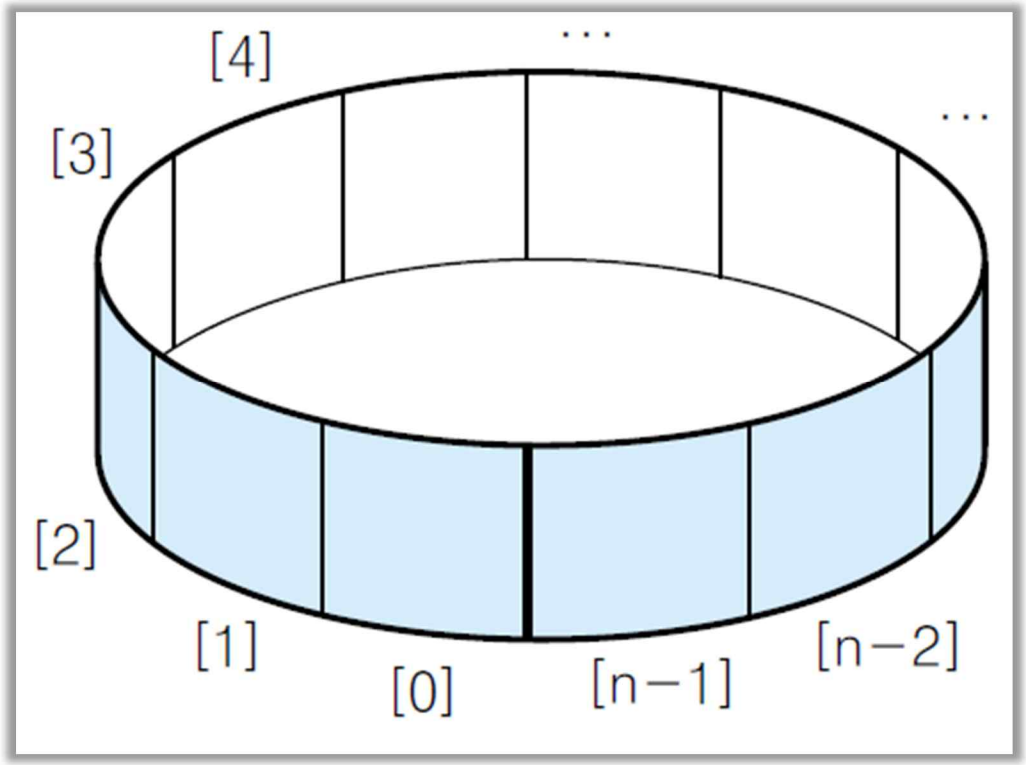
선형 큐의 잘못된 포화상태 인식의 해결 방법-1

- 저장된 원소들을 배열의 앞부분으로 이동시키기
  - 순차자료에서의 이동 작업은 연산이 복잡하여 효율성이 떨어짐



선형 큐의 잘못된 포화상태 인식의 해결 방법-2

- 1차원 배열을 사용하면서 논리적으로 배열의 처음과 끝이 연결되어 있다고 가정하고 사용 ⇒ **원형큐**
- 원형 큐의 논리적 구조



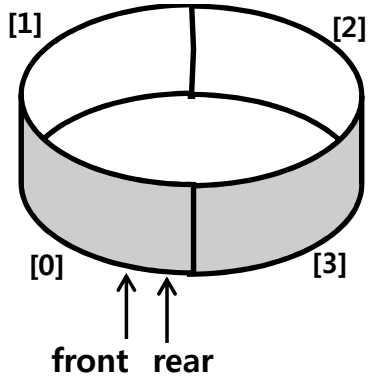
원형 큐의 구조

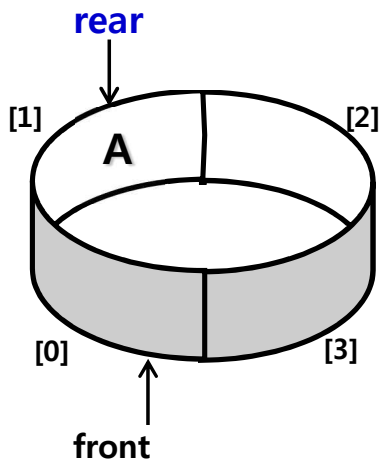
- 초기 공백 상태 : front = rear = 0
- front와 rear의 위치가 배열의 마지막 인덱스 n-1에서 논리적인 다음 자리인 인덱스 0번으로 이동하기 위해서 나머지연산자 mod를 사용
  - 3 ÷ 4 = 0 **...3** (몫=0, 나머지=3)
  - 3 **mod** 4 = **3**

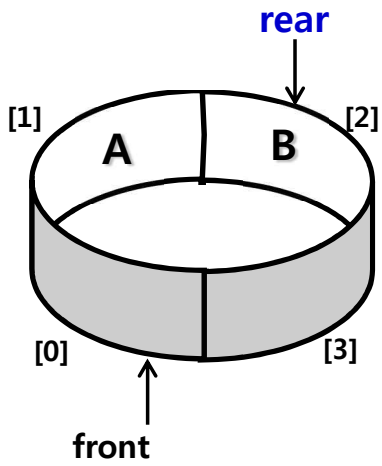
	삽입위치	삭제위치
선형큐	rear = rear + 1	front = front + 1
원형큐	rear = (rear+1) **mod** n	front = (front+1) **mod** n

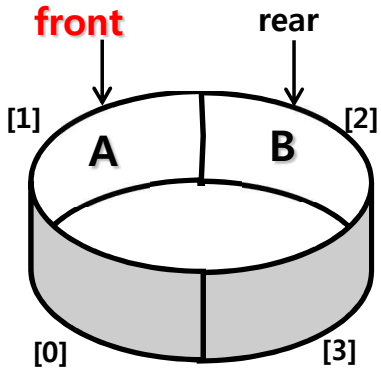
- 사용조건) 공백 상태와 포화 상태 구분을 쉽게 하기 위해서 front가 있는 자리는 사용하지 않고 항상 빈자리로 둔다.

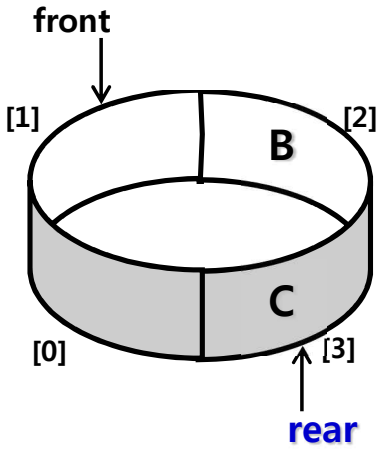
원형 큐에서의 연산 과정

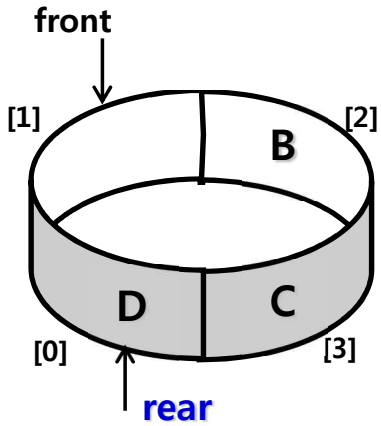

[1]
[2]
[0]
[3]
front
rear


rear
[1]
A
[2]
[0]
[3]
front


rear
[1]
A
B
[2]
[0]
[3]
front

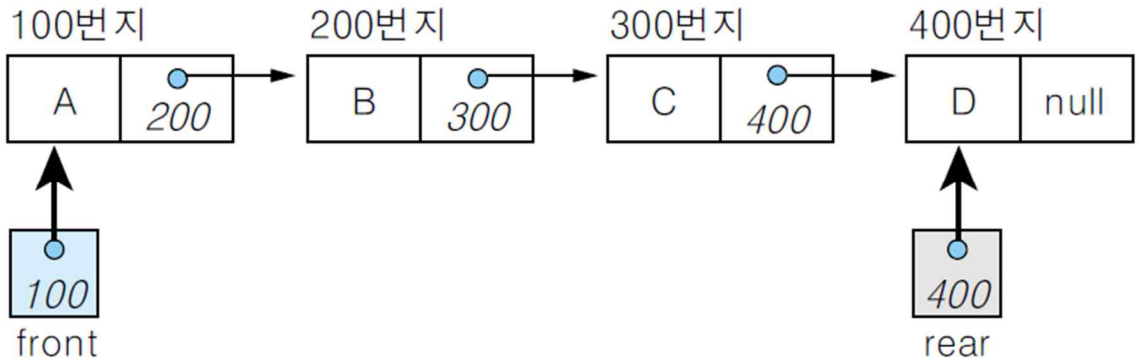

front
rear
[1]
A
B
[2]
[0]
[3]


front
[1]
B
[2]
C
[0]
[3]
rear


front
[1]
B
[2]
D
C
[0]
[3]
rear

연결 큐

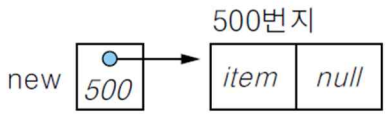
- 단순 연결 리스트를 이용한 큐
  - 큐의 원소 : 단순 연결 리스트의 노드
  - 큐의 원소의 순서 : 노드의 링크 포인터로 연결
  - 변수 front : 첫 번째 노드를 가리키는 포인터 변수
  - 변수 rear : 마지막 노드를 가리키는 포인터 변수
- 상태 표현
  - 초기 상태와 공백 상태 : front = rear = null
- 연결 큐의 구조



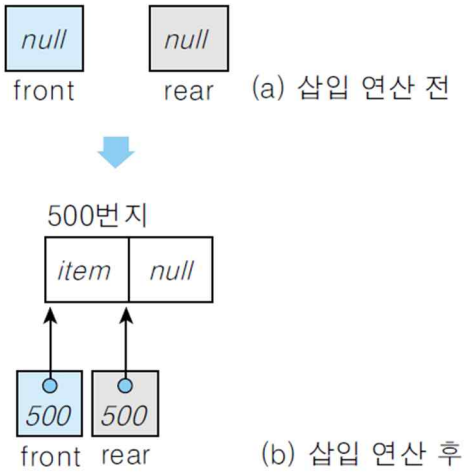
자료구조-삽입연산

Robot Media Laboratory

① 삽입할 새 노드를 생성하여 데이터 필드에 item을 저장한다. 삽입할 새 노드는 연결 큐의 마지막 노드가 되어야 하므로 링크 필드에 null을 저장한다.



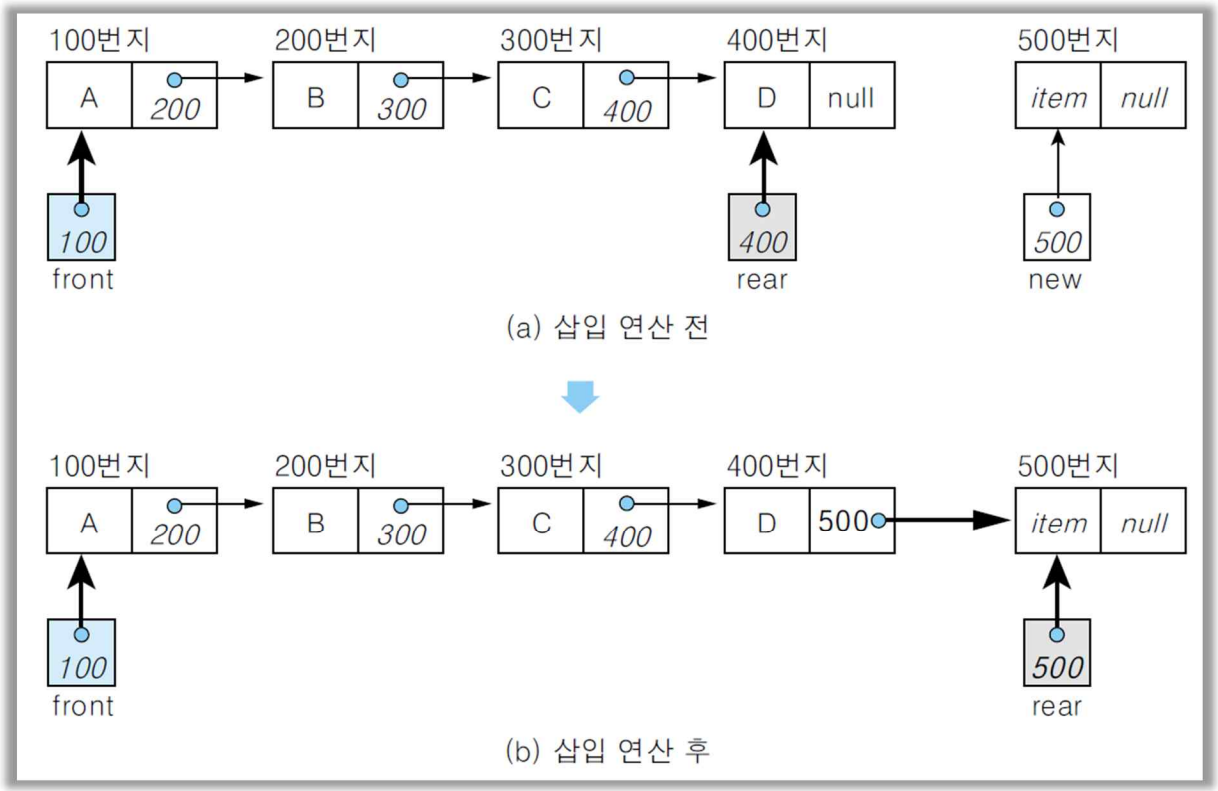
② 새 노드를 삽입하기 전에 연결 큐가 공백인지 아닌지를 검사한다. 연결 큐가 공백인 경우에는 삽입할 새 노드가 큐의 첫 번째 노드이자 마지막 노드이므로 포인터 front와 rear가 모두 새 노드를 가리키도록 설정한다.



자료구조- 삽입연산

Robot Media Laboratory

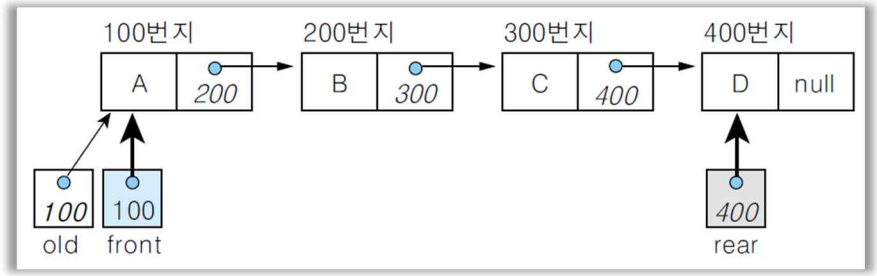
③ 큐가 공백이 아닌 경우, 즉 노드가 있는 경우에는 현재 큐의 마지막 노드의 뒤에 새 노드를 삽입하고 마지막 노드를 가리키는 rear가 삽입한 새 노드를 가리키도록 설정한다.



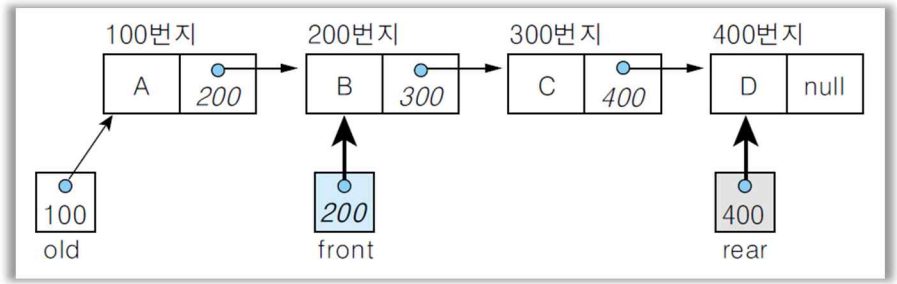
자료구조- 삭제연산

Robot Media Laboratory

① 삭제연산에서 삭제할 노드는 큐의 첫 번째 노드로서 포인터 front가 가리키고 있는 노드이다. front가 가리키는 노드를 포인터 old가 가리키게 하여 삭제할 노드를 지정한다.



② 삭제연산 후에는 현재 front 노드의 다음 노드가 front 노드(첫번째 노드)가 되어야 하므로, 포인터 front를 재설정한다.



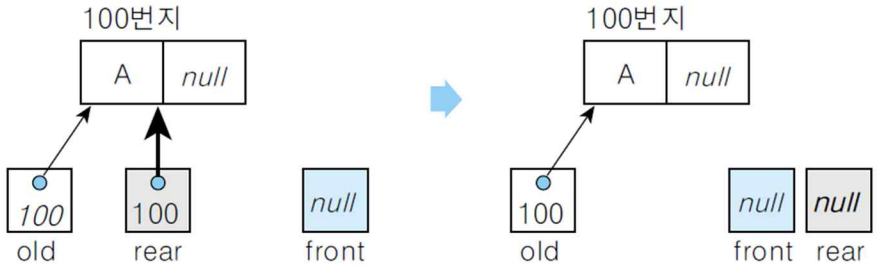
ROBOT MEDIA LABORATORY

자료구조- 삭제연산

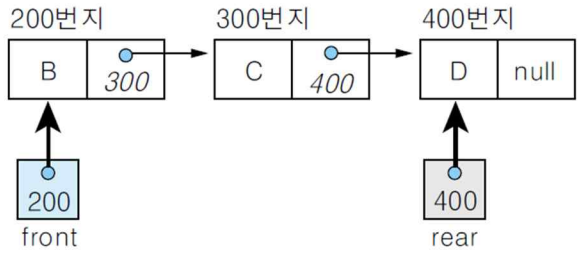
Robot Media Laboratory

③ 현재 큐에 노드가 하나뿐이어서 삭제연산 후에 공백 큐가 되는 경우 :

☞ 큐의 마지막 노드가 없어지므로 포인터 rear를 null로 설정한다.



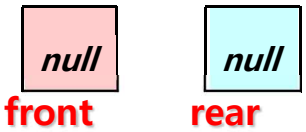
④ 포인터 old가 가리키고 있는 노드를 삭제하고, 메모리 공간을 시스템에 반환 한다



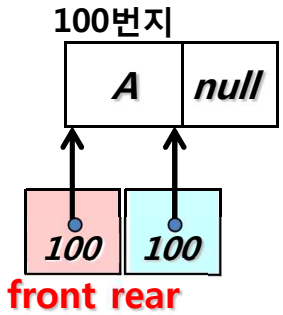
자료구조- 연산 과정

Robot Media Laboratory

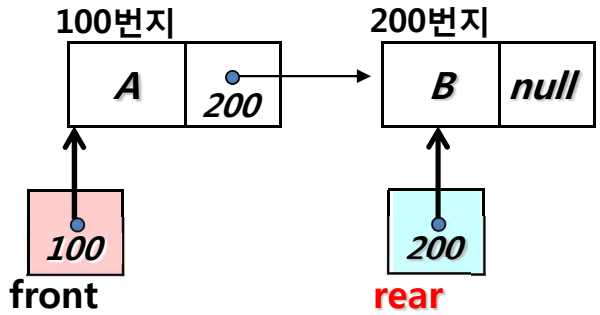
① 공백 큐 생성



② 원소 A 삽입



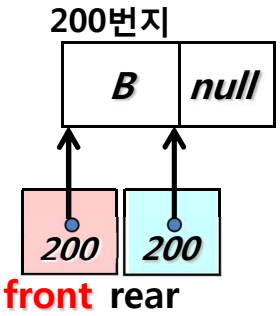
③ 원소 B 삽입



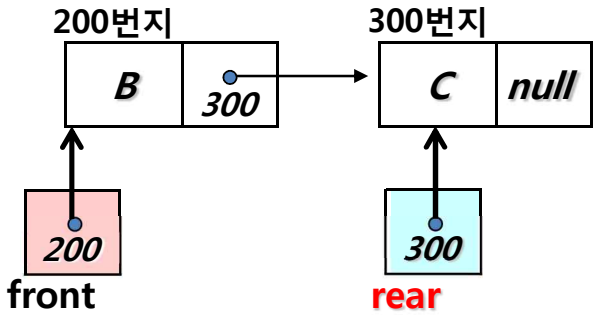
자료구조- 연산 과정

Robot Media Laboratory

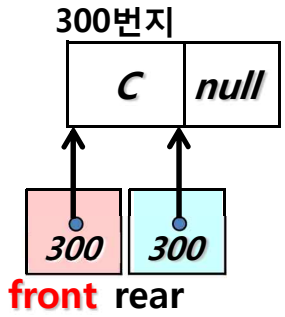
④ 원소 삭제



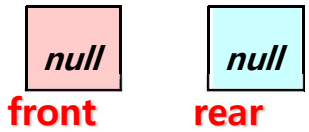
⑤ 원소 C 삽입



⑥ 원소 삭제



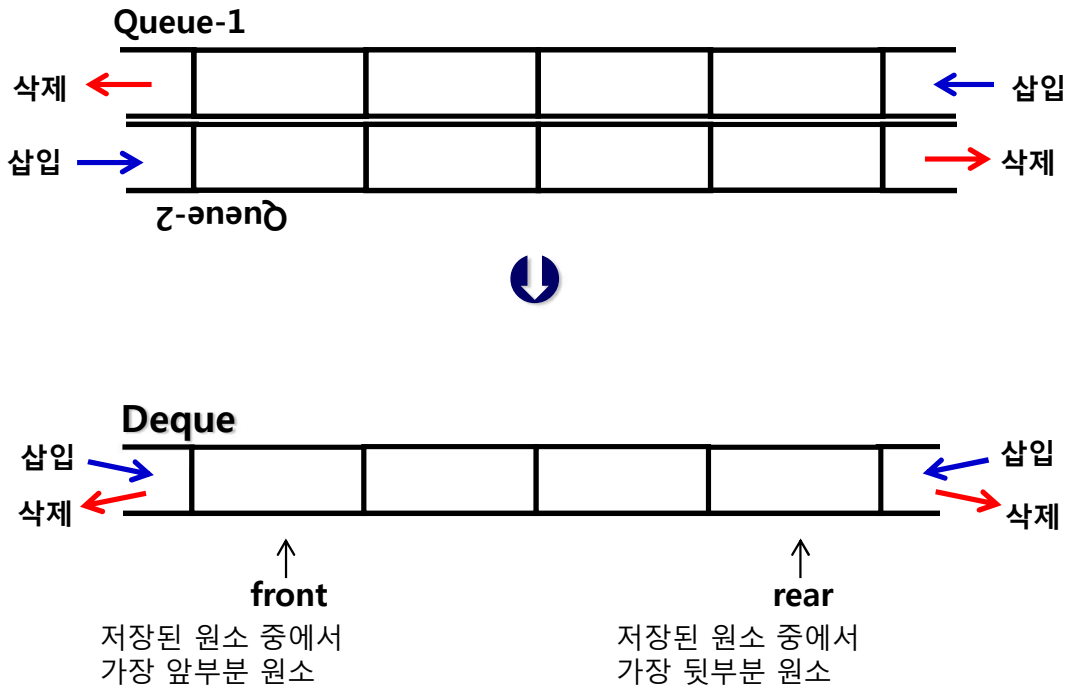
⑦ 원소 삭제



ROBOT MEDIA LABORATORY

덱(Deque, double-ended queue)

- 큐 2개를 반대로 붙여서 만든 자료구조



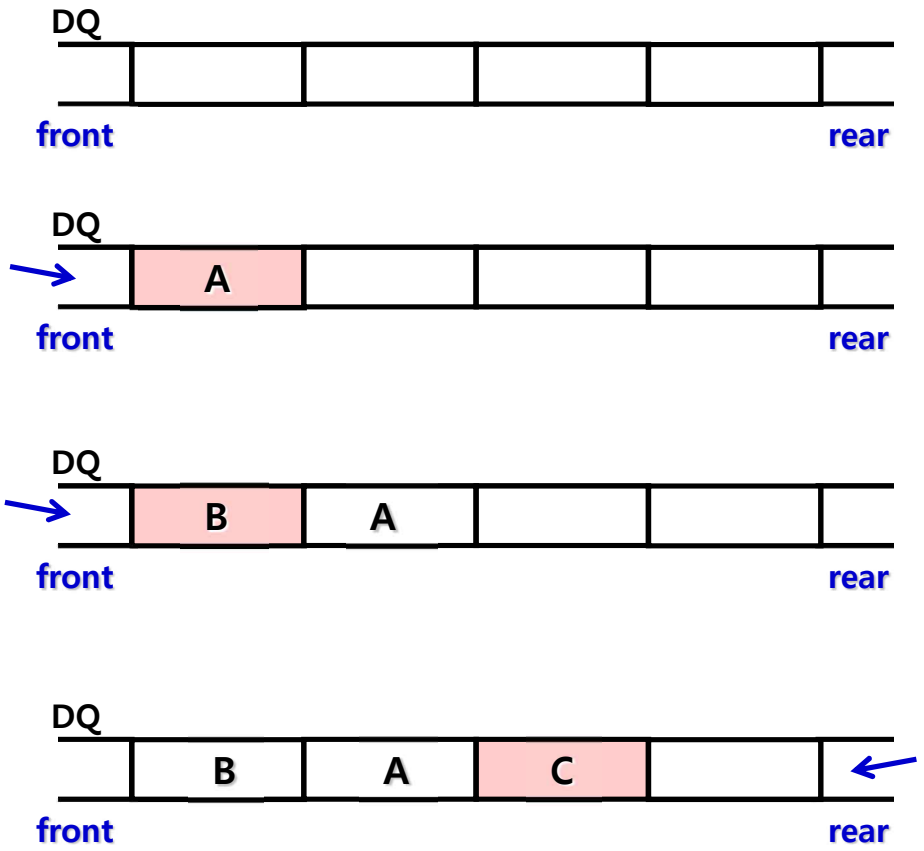
덱에서의 연산 과정

① 공백 덱

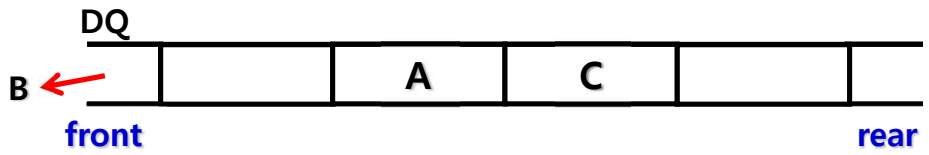
②A 머리삽입

③ B 머리삽입

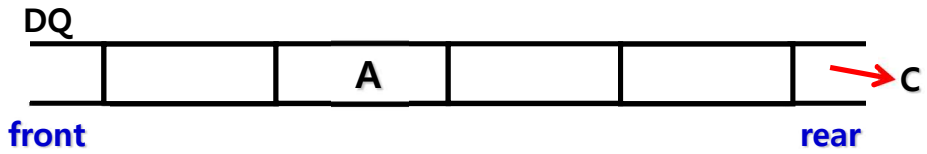
④ C 꼬리삽입



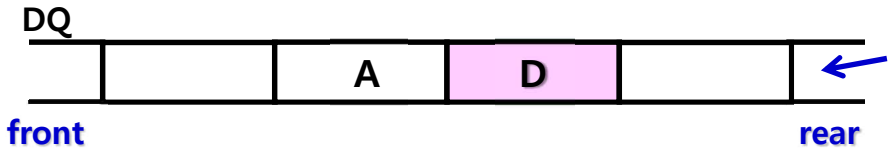
⑤ 머리 삭제



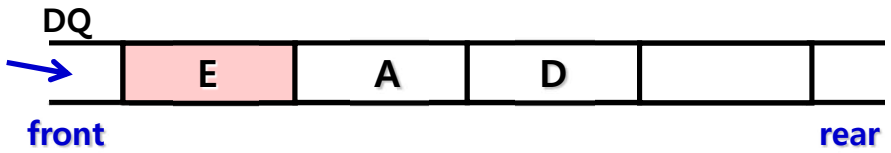
⑥ 꼬리 삭제



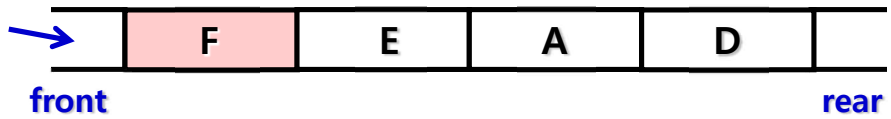
⑦ 꼬리 삽입



⑧ 머리 삽입

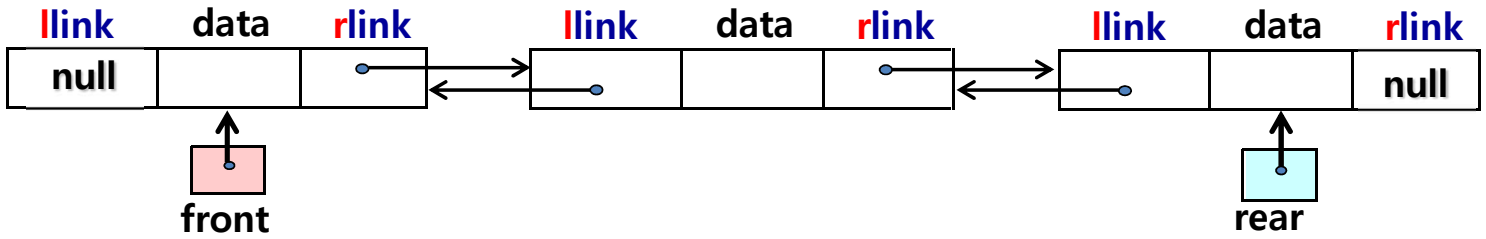


⑨ 머리삽입



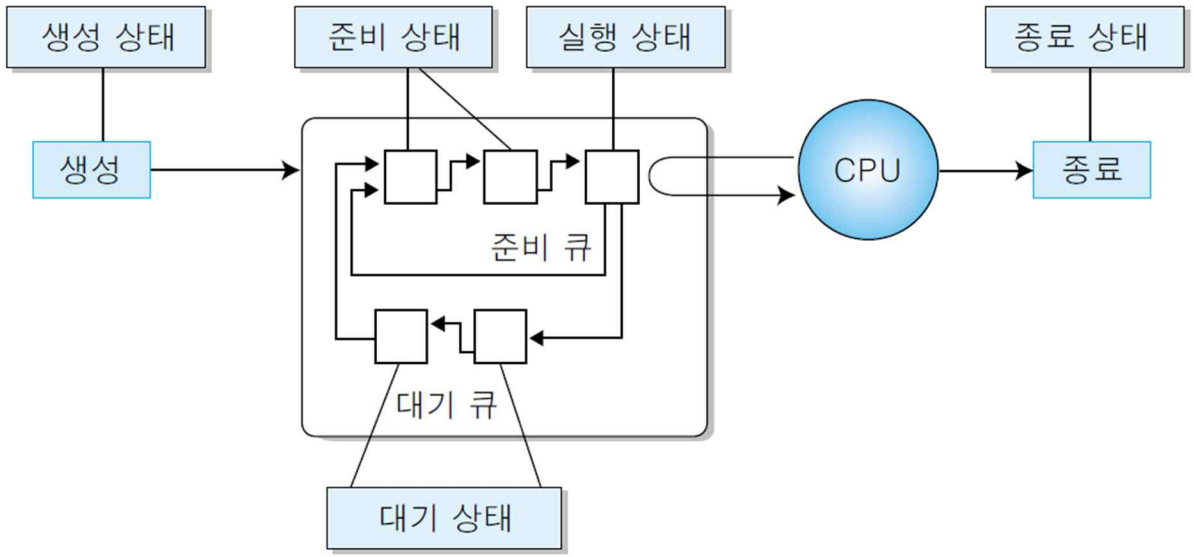
덱의 구현

- 양쪽 끝에서 삽입/삭제 연산을 수행하면서 크기 변화와 저장된 원소의 순서 변화가 많으므로 순차 자료구조는 비효율적
- 양방향으로 연산이 가능한 이중 연결 리스트를 사용한다.



운영체제의 작업 큐

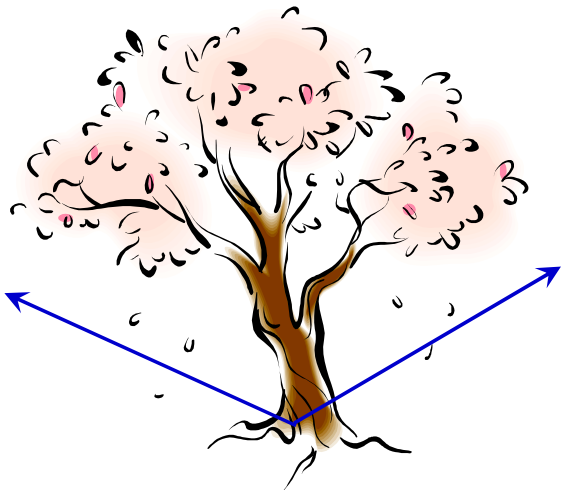
- 프린터 버퍼 큐
  - CPU에서 프린터로 보낸 데이터 순서대로(선입선출) 프린터에서 출력하기 위해서 선입선출 구조의 큐 사용
- 스케줄링 큐
  - CPU 사용을 요청한 프로세서들의 순서를 스케줄링하기 위해서 큐를 사용



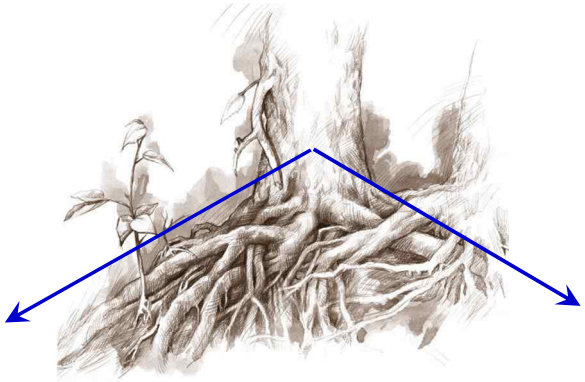
자료구조
-트리-

트리(tree)

- 원소들 간에 1:多 관계를 가지는 비선형 자료구조
- 원소들 간에 계층관계를 가지는 계층형 자료구조
- 상위 원소에서 하위 원소로 내려가면서 확장되는 트리(나무)모양의 구조



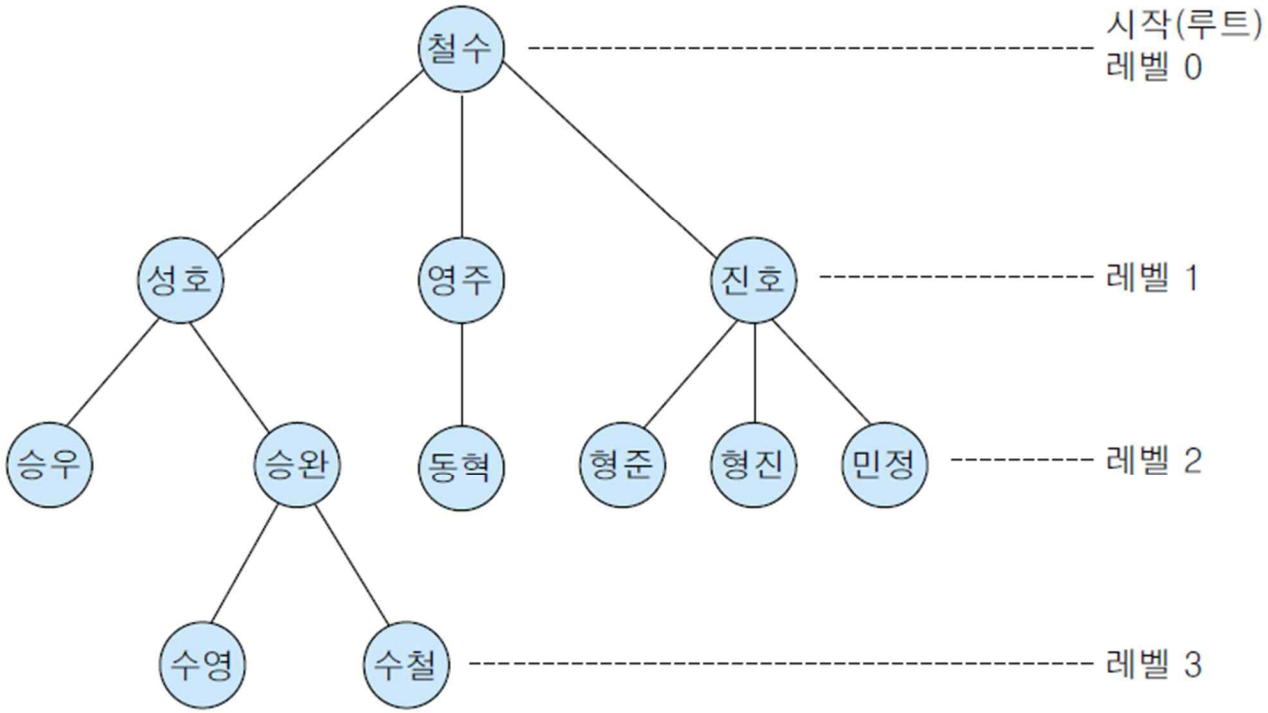
하나의 줄기에서 가지로 뻗어나가면서 확장되는 구조



하나의 그루터기에서 뿌리로 뻗어나가면서 확장되는 구조

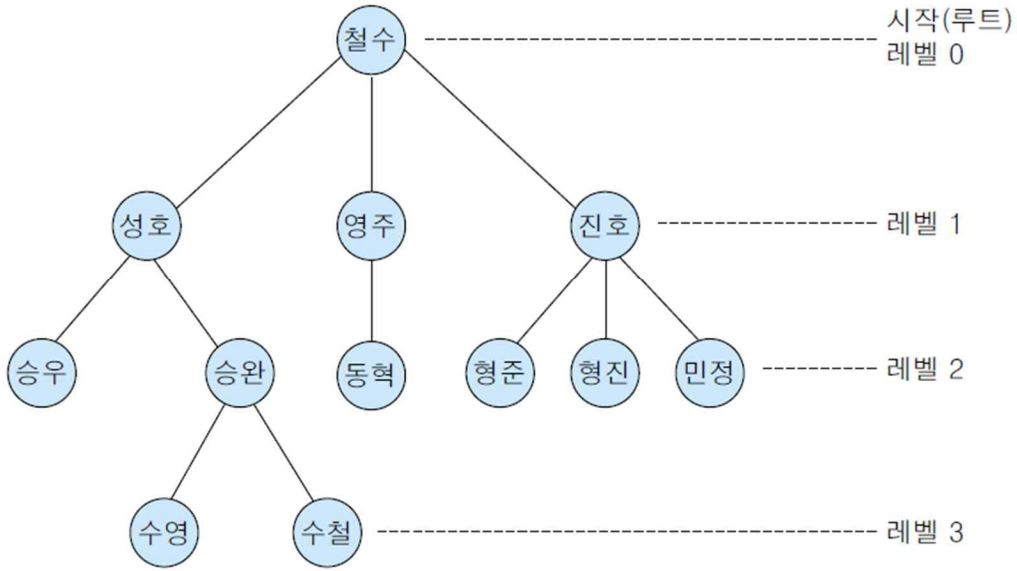
트리 자료구조의 예 – 가계도

- 가계도의 자료 : 가족 구성원
- 자료를 연결하는 선 : 부모-자식 관계 표현



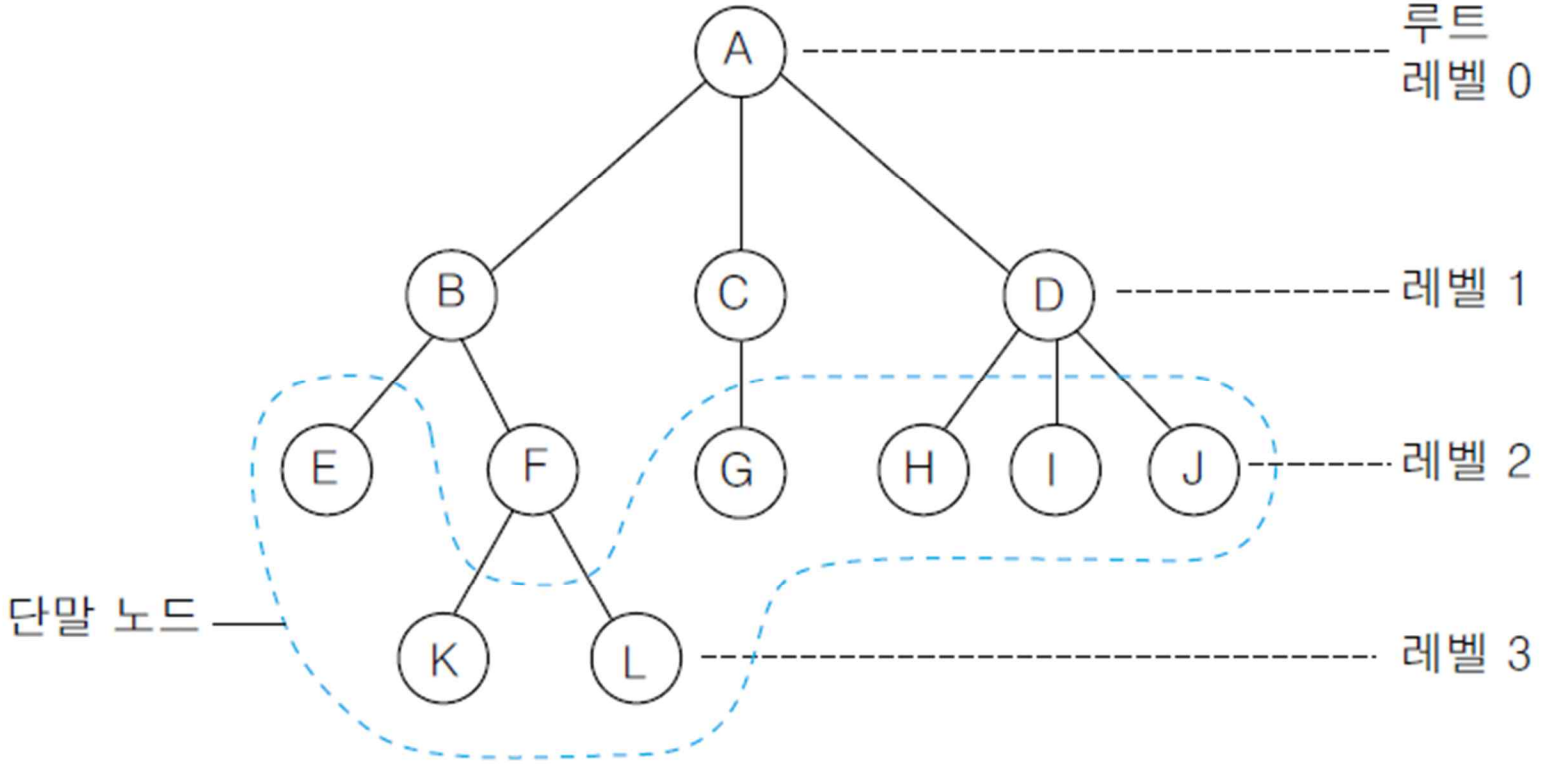
자료구조

Robot Media Laboratory



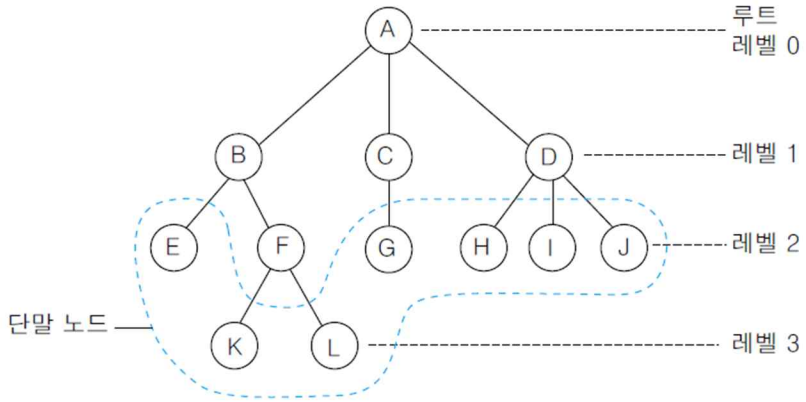
- 철수의 자식 – 성호, 영주, 진호
- 성호, 영주, 진호의 부모 – 철수
- 같은 부모의 자식들끼리는 형제관계
  - 성호, 영주, 진호는 형제관계
- 조상 – 현재 위치에서 연결된 선을 따라 올라가면서 만나는 사람들
  - 수영의 조상 : 승완, 성호, 철수
- 자손 - 현재 위치에서 연결된 선을 따라 내려가면서 만나는 사람들
  - 성호의 자손 : 승우, 승완, 수영, 수철
- 선을 따라 내려가면서 다음 세대로 확장
- 가족 구성원 누구든지 자기의 가족을 데리고 분가하여 독립된 가계를 이룰 수 있다.

- 트리 A



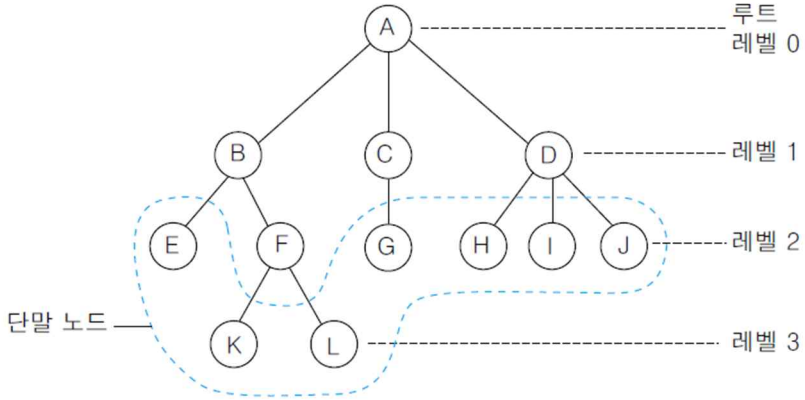
자료구조

Robot Media Laboratory



- **노드(node) – 트리의 원소**
  - 트리 A의 노드 - A,B,C,D,E,F,G,H,I,J,K,L
- **루트 노드(root node) – 트리의 시작 노드**
  - 트리 A의 루트노드 - A
- **간선(edge) – 노드를 연결하는 선. 부모 노드와 자식 노드를 연결**
- **형제 노드(sibling node) – 같은 부모 노드의 자식 노드들**
  - B,C,D는 형제 노드
- **조상 노드 – 간선을 따라 루트 노드까지 이르는 경로에 있는 모든 노드들**
  - K의 조상 노드 : F, B, A
- **서브 트리(subtree) – 부모 노드와 연결된 간선을 끊었을 때 생성되는 트리**
  - 각 노드는 자식 노드의 개수 만큼 서브 트리를 가진다.
- **자손 노드 – 서브 트리에 있는 하위 레벨의 노드들**
  - B의 자손 노드 – E,F,K,L

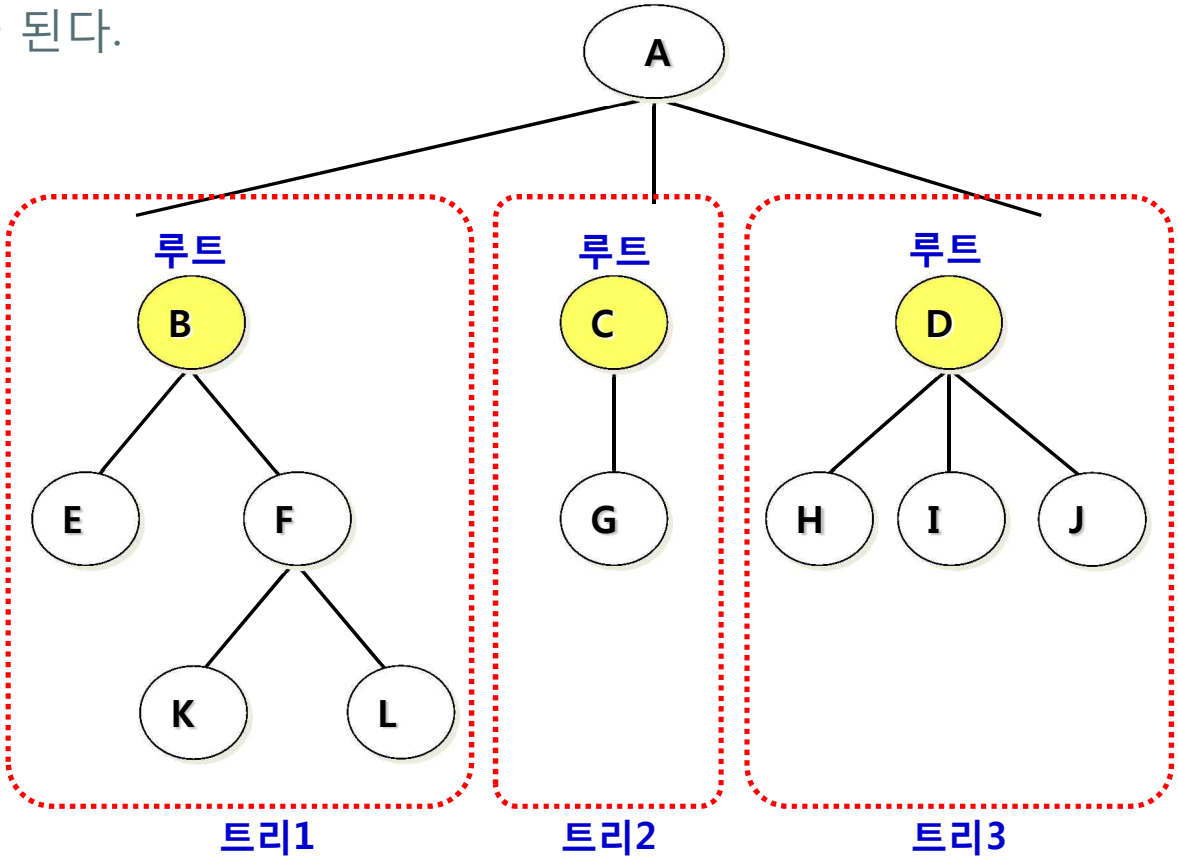
ROBOT MEDIA LABORATORY



- **차수(degree)**
  - 노드의 차수 : 노드에 연결된 자식 노드의 수.
    - A의 차수=3, B의 차수=2, C의 차수=1
  - 트리의 차수 : 트리에 있는 노드의 차수 중에서 가장 큰 값
    - 트리 A의 차수=3
  - **단말 노드**(리프 노드) : 차수가 0인 노드. 자식 노드가 없는 노드
- **높이**
  - 노드의 높이 : 루트에서 노드에 이르는 간선의 수. 노드의 레벨
    - B의 높이=1, F의 높이=2
  - 트리의 높이 : 트리에 있는 노드의 높이 중에서 가장 큰 값. 최대 레벨
    - 트리 A의 높이=3

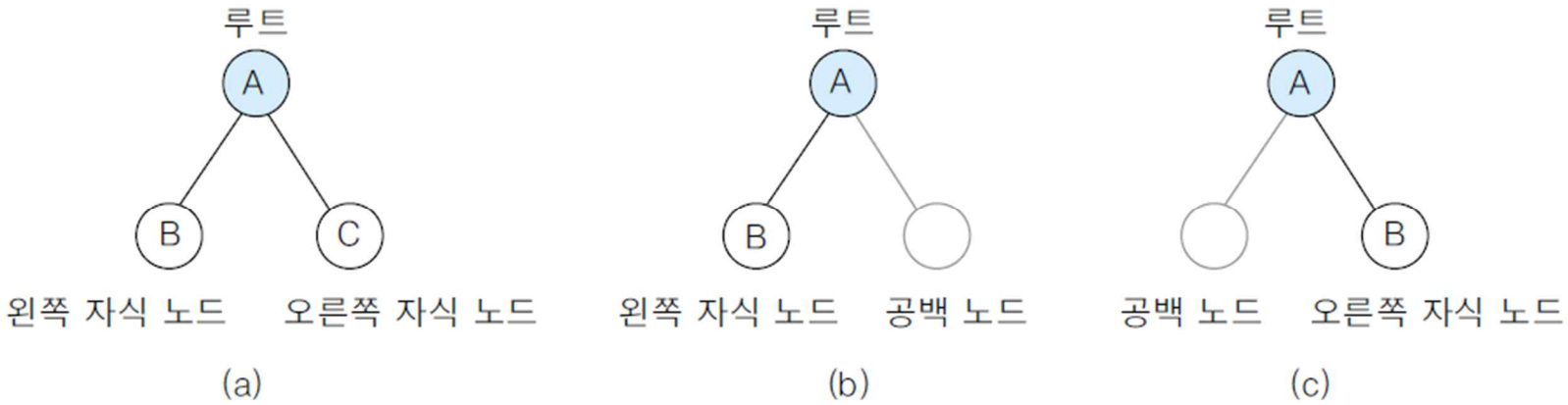
포리스트(forest) : 서브트리의 집합

- 트리A에서 노드 A를 제거하면, A의 자식 노드 B, C, D에 대한 서브 트리가 생기고, 이들의 집합은 포리스트가 된다.



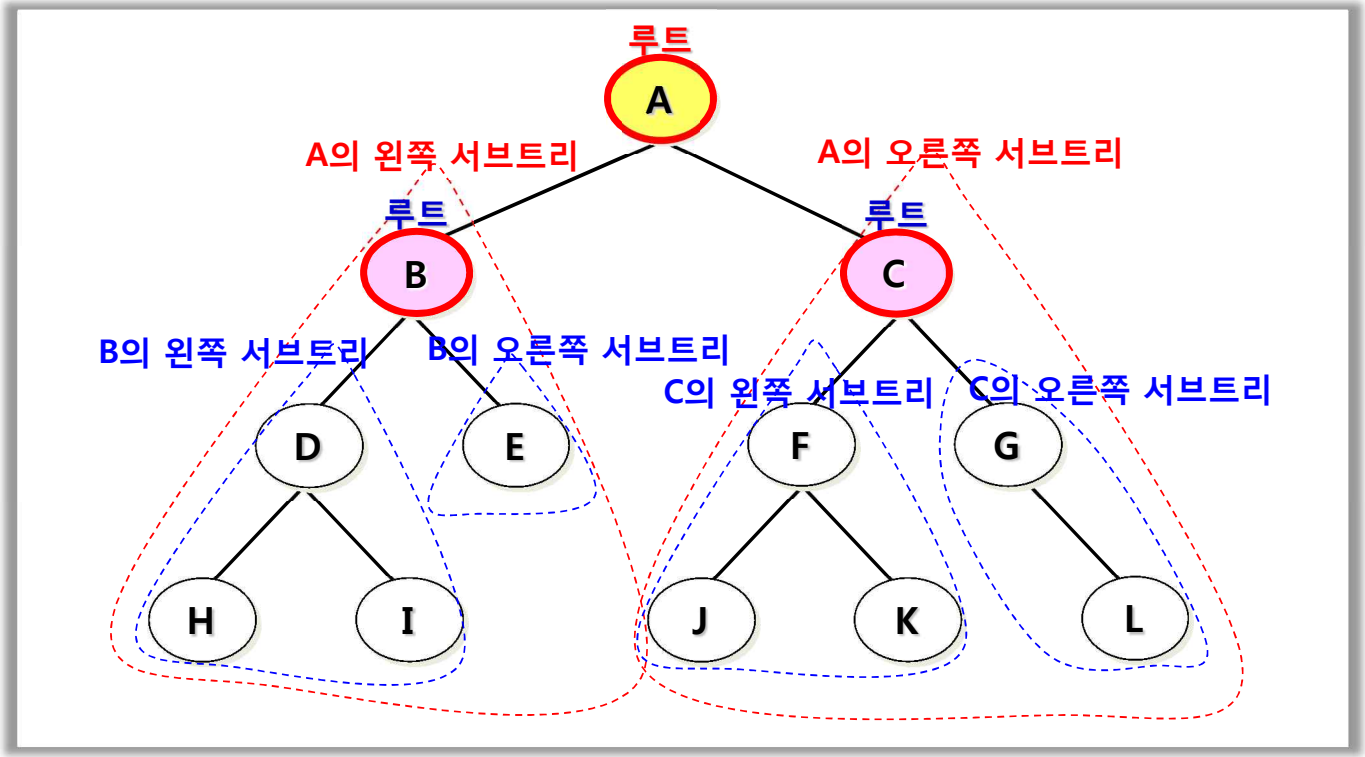
이진트리

- 트리의 노드 구조를 일정하게 정의하여 트리의 구현과 연산이 쉽도록 정의한 트리
- 이진 트리의 모든 노드는 왼쪽 자식 노드와 오른쪽 자식 노드 만을 가진다.
  - 부모 노드와 자식 노드 수와의 관계 ☞ 1:2
  - 공백 노드도 자식 노드로 취급한다.
  - 0 ≤ 노드의 차수 ≤ 2



이진트리는 순환적 구성

- 노드의 왼쪽 자식 노드를 루트로 하는 왼쪽 서브트리도 이진 트리
- 노드의 오른쪽 자식 노드를 루트로 하는 오른쪽 서브 트리도 이진 트리



이진트리의 특성

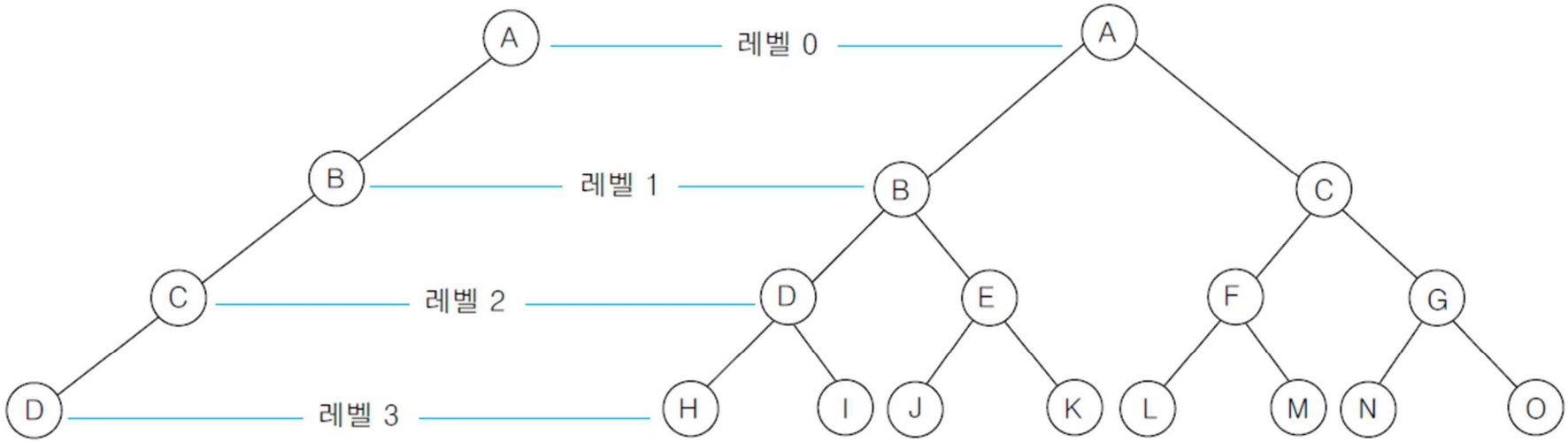
- 정의1) **n개의 노드**를 가진 이진 트리는 항상 **(n-1)개의 간선**을 가진다.
  - 루트를 제외한 (n-1)개의 노드가 부모 노드와 연결되는 한 개의 간선을 가짐

- 정의2) 높이가 h인 이진 트리가 가질 수 있는 **노드의 최소 개수는 (h+1)**개가 되며, 최대 개수는 **$(2^{h+1}-1)$**개가 된다.
  - 이진 트리의 높이가 h가 되려면 한 레벨에 최소한 한 개의 노드가 있어야 하므로 높이가 h인 이진 트리의 최소 노드의 개수는 (h+1)개
  - 하나의 노드는 최대 2개의 자식 노드를 가질 수 있으므로 레벨 i에서의 노드의 최대 개수는 2^i개 이므로 높이가 h인 이진 트리 전체의 노드 개수는

    $\sum 2^i = 2^{h+1}-1$ 개

이진트리의 특성

- 높이가 3이면서 최소의 노드를 갖는 이진트리와 최대의 노드를 갖는 이진트리

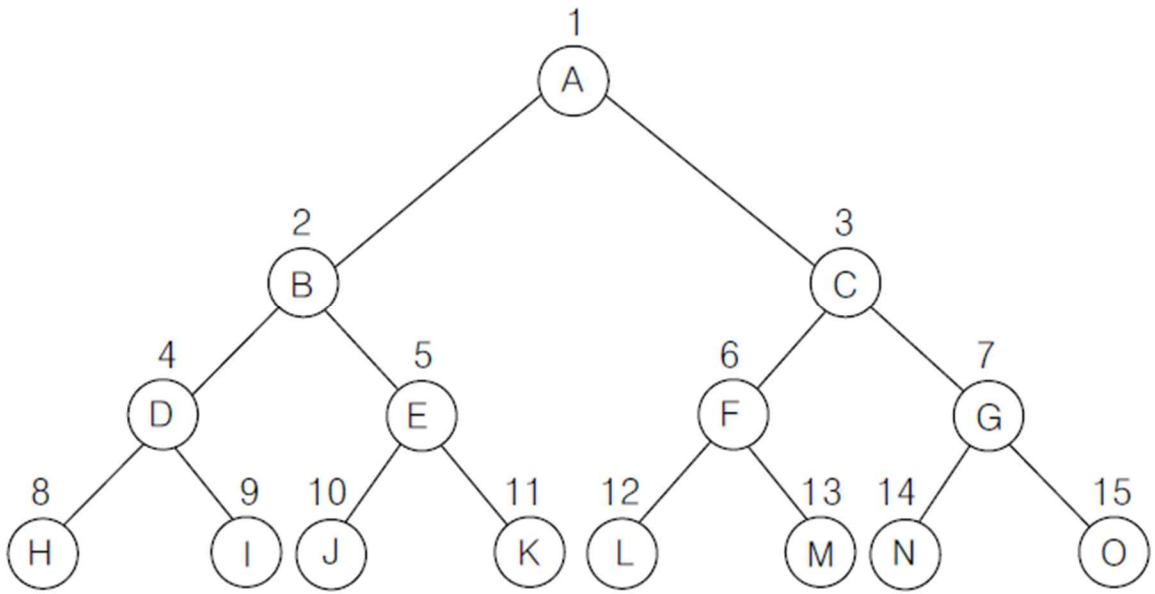


(a) 최소의 노드를 갖는 이진 트리

(b) 최대의 노드를 갖는 이진 트리

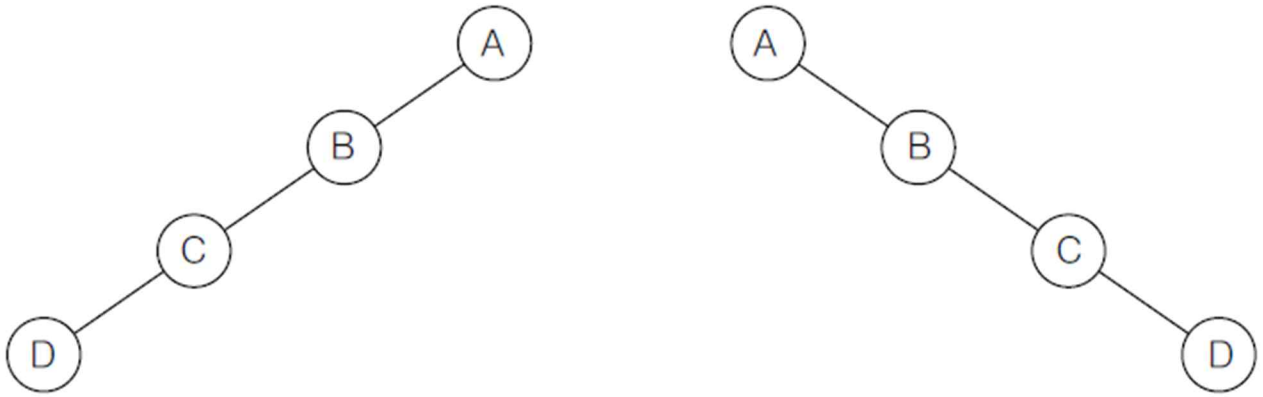
이진 트리의 종류

- 포화 이진 트리(Full Binary Tree)
  - 모든 레벨에 노드가 포화상태로 차 있는 이진 트리
  - 높이가 h일 때, 최대의 노드 개수인 $(2^{h+1}-1)$ 의 노드를 가진 이진 트리
  - 루트를 1번으로 하여 $2^{h+1}-1$까지 정해진 위치에 대한 노드 번호를 가짐



편향 이진 트리(Skewed Binary Tree)

- 높이 h에 대한 최소 개수의 노드를 가지면서 한쪽 방향의 자식 노드만을 가진 이진 트리
- 왼쪽 편향 이진 트리
  - 모든 노드가 왼쪽 자식 노드만을 가진 편향 이진 트리
- 오른쪽 편향 이진 트리
  - 모든 노드가 오른쪽 자식 노드만을 가진 편향 이진 트리

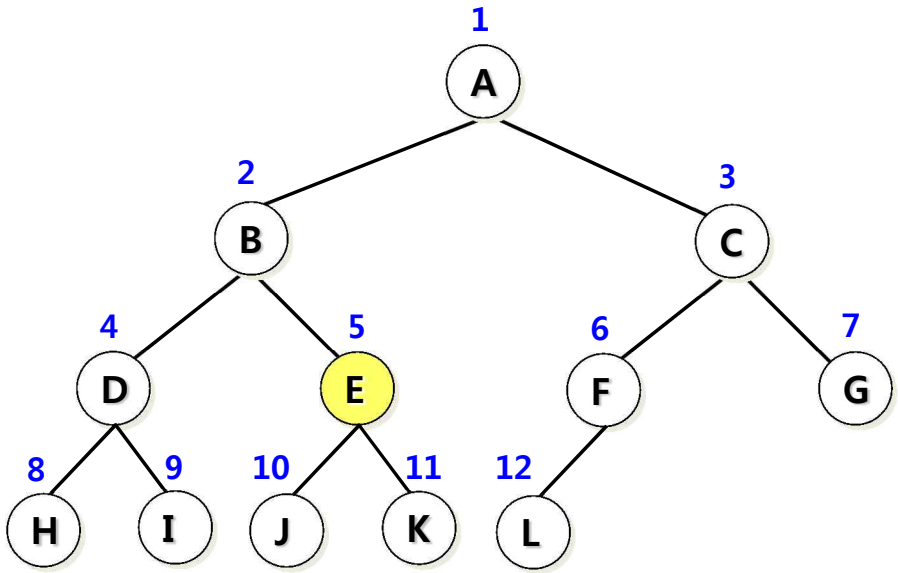


(a) 왼쪽 편향 이진 트리 (b) 오른쪽 편향 이진 트리

순차 자료구조를 이용한 이진트리의 구현

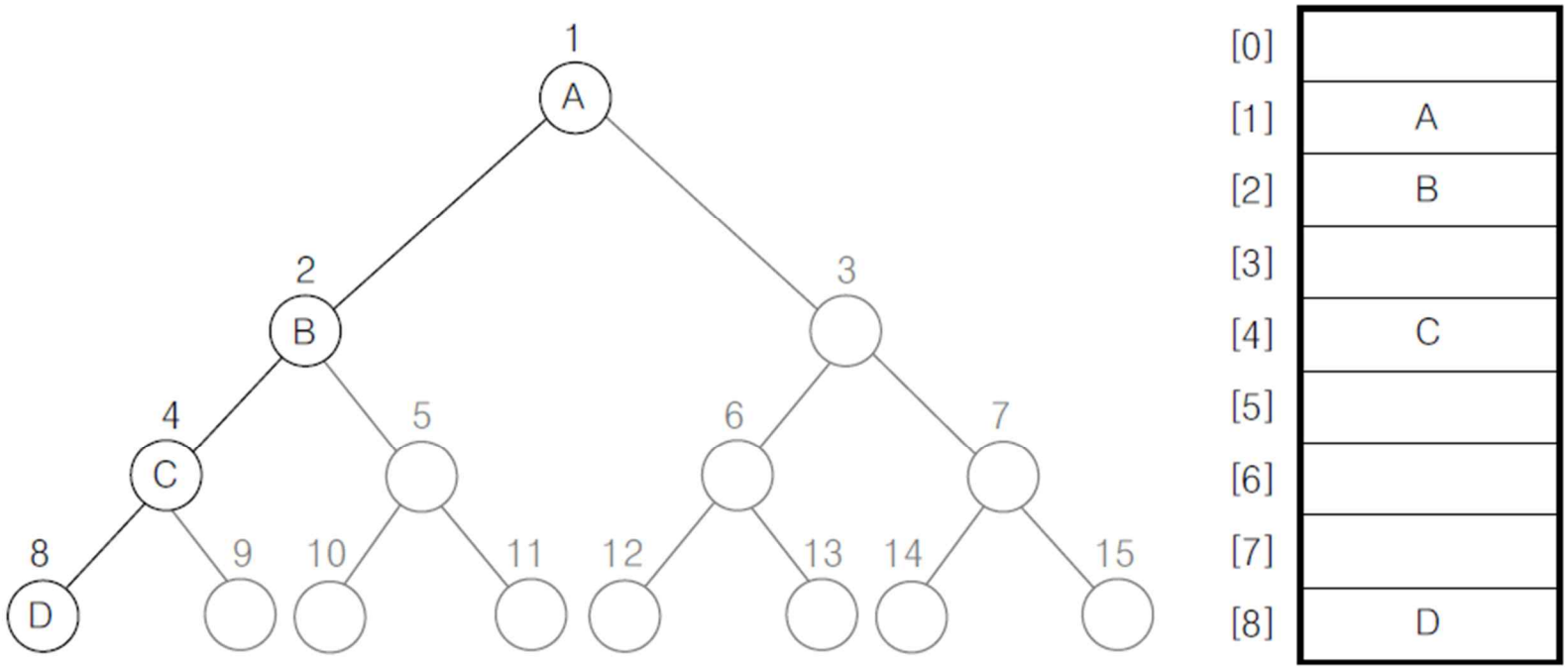
- 1차원 배열의 순차 자료구조 사용
  - 높이가 h인 포화 이진 트리의 노드번호를 배열의 인덱스로 사용
  - 인덱스 0번 : 실제로 사용하지 않고 비워둠.
  - 인덱스 1번 : 루트 저장

완전 이진 트리의 1차원 배열 표현



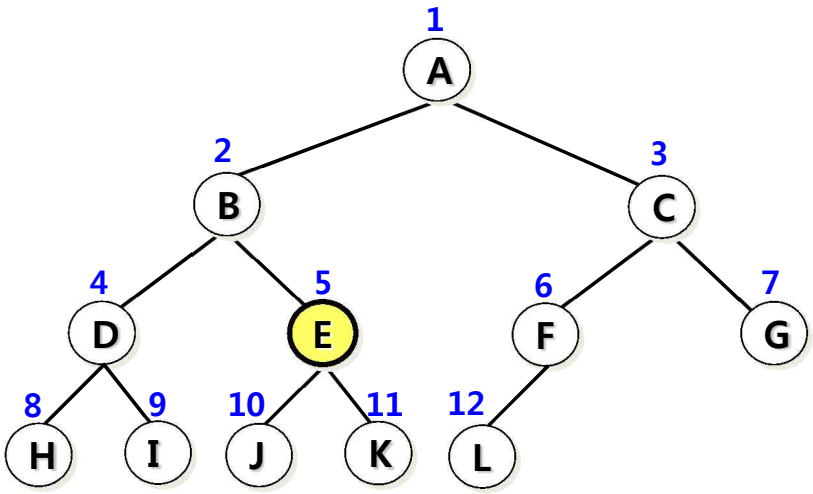
[0]	
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I
[10]	J
[11]	K
[12]	L

왼쪽 편향 이진 트리의 1차원 배열 표현



이진 트리의 1차원 배열에서의 인덱스 관계

노드	인덱스	성립 조건
노드 i의 부모 노드	⌊ i / 2 ⌋	i > 1
노드 i의 왼쪽 자식 노드	2 x i	(2 x i) ≤ n
노드 i의 오른쪽 자식 노드	(2 x i) + 1	(2 x i + 1) ≤ n
루트 노드	1	n > 0



- **이진 트리의 순차 자료구조 표현의 단점**
  - 편향 이진 트리의 경우에 사용하지 않는 배열 원소에 대한 메모리 공간 낭비 발생
  - 트리의 원소 삽입/삭제에 대한 배열의 크기 변경 어려움