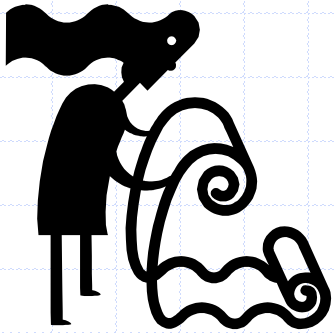
자료구조 복습

의사코드



- **의사코드**(pseudo-code): 알고리즘을 설명하기 위한 고급언어
  - 컴퓨터가 아닌, 인간에게 읽히기 위해 작성됨
  - 저급의 상세 구현내용이 아닌, 고급 개념을 소통하기 위해 작성됨
- 자연어 문장보다 더 구조적이지만, 프로그래밍 언어보다 덜 상세함
- 알고리즘을 설명하는데 선호되는 표기법
- **예:** 배열의 최대값 원소 찾기

Alg ***arrayMax***(*A*, *n*)
  input array *A* of *n* integers
  output maximum element of *A*

1. *currentMax* ← *A*[0]
2. **for** *i* ← 1 **to** *n* – 1
   if (*A*[*i*] > *currentMax*)
   currentMax ← *A*[*i*]
3. **return** *currentMax*

의사코드 문법

- 제어(control flow)
  - **if** (*exp*) **…**
    [elseif (*exp*) **…]***
    [else …]
  - **for** *var* ← exp_1 **to|downto** exp_2
    …
  - **for each** *var* ∈ *exp*
    …
  - **while** (*exp*)
    …
  - **do**
    …
    while (*exp*)
- **주의:**
  들여쓰기(indentation)로
  범위(scope)를 정의
- 연산(arithmetic)

←	치환(assignment)
=, <, ≤, >, ≥	관계 연산자
&, ||, !	논리 연산자
$s_1 \le n^2$	첨자 등 수학적 표현 허용

- 메쏘드(method) 정의, 반환, 호출
  - **Alg** *method*([*arg* [, *arg*]*])
    …
  - **return** [*exp* [, *exp*]*]
  - *method*([*arg* [, *arg*]*])
- 주석(comments)

  input …
  output …
  {This is a comment}

실행시간의 증가율

- 하드웨어나 소프트웨어 환경을 변경하면:
  - $T(n)$에 상수 배수 만큼의 영향을 주지만,
  - $T(n)$의 증가율을 변경하지는 않는다
- 따라서 선형의 **증가율**(growth rate)을 나타내는 실행시간 $T(n)$은 arrayMax의 고유한 속성이다

Big-Oh와 증가율

- Big-Oh 표기법은 함수의 증가율의 **상한**(upper bound)을 나타낸다
- "$f(n) = O(g(n))$"이라 함은 "$f(n)$의 증가율은 $g(n)$의 증가율을 넘지 않음"을 말한다
- Big-Oh 표기법을 사용함으로써, 증가율에 따라 함수들을 서열화할 수 있다

	$f(n) = O(g(n))$	$g(n) = O(f(n))$
$g(n)$의 증가율이 더 빠르면	yes	no
$f(n)$ 의 증가율이 더 빠르면	no	yes
둘이 같으면	yes	yes

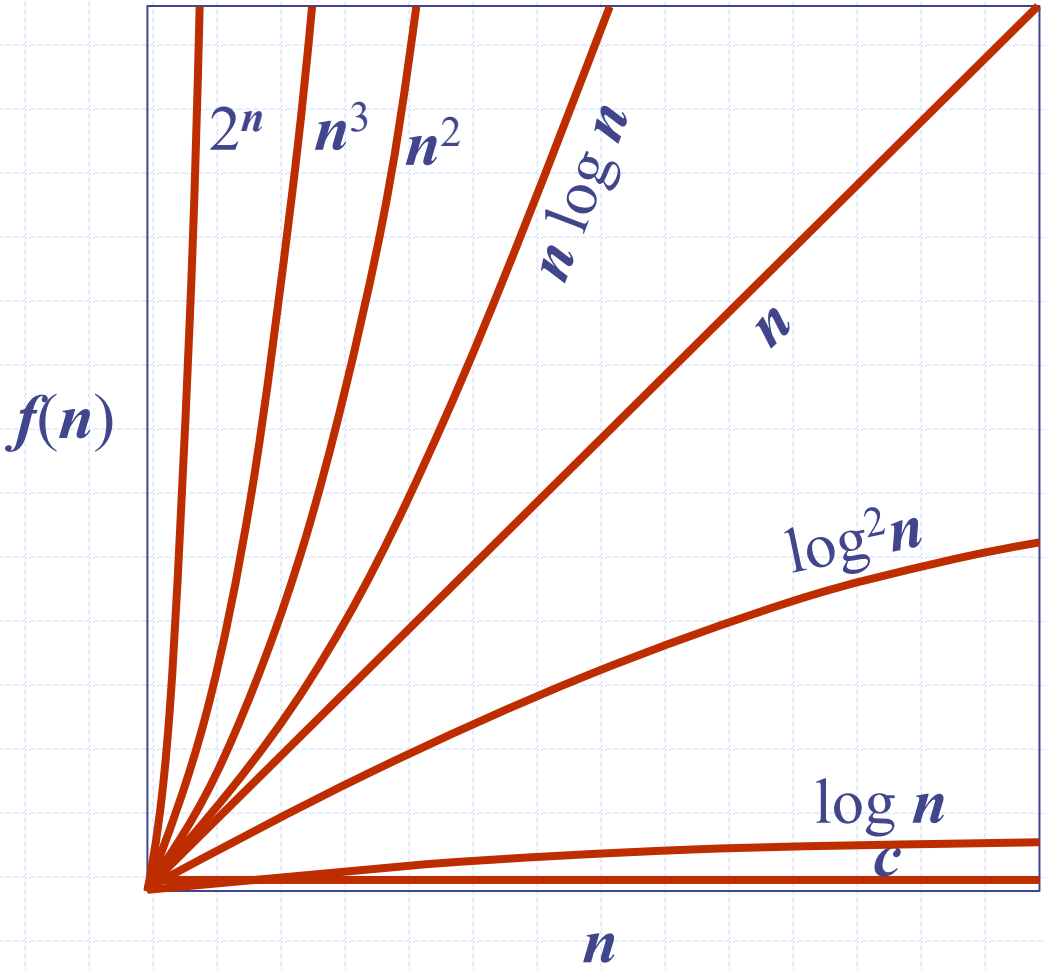
점근분석

- 알고리즘을 **점근분석**(asymptotic analysis) 함으로써 big-Oh 표기법에 의한 실행시간을 구할 수 있다
- 점근분석을 수행하기 위해서는,
  1. 최악의 원시작업 실행회수를 입력크기의 함수로서 구한다
  2. 이 함수를 big-Oh 표기법으로 나타낸다
- **예**
  1. 알고리즘 arrayMax가 최대 $7n-2$개의 원시작업을 실행한다는 것을 구한다
  2. "알고리즘 arrayMax는 $O(n)$ 시간에 수행된다"고 말한다
- 상수계수와 낮은 차수의 항들은 결국 탈락되므로, 원시작업 수를 계산할 때부터 이들을 무시할 수 있다

전형적인 증가율

함수	이름	$f(10^2)$	$f(10^3)$	$f(10^4)$	$f(10^5)$
c	상수(constant)	1	1	1	1
$\log n$	로그(logarithmic)	7	10	14	18
$\log^2 n$	로그제곱(log-squared)	49	100	200	330
n	선형(linear)	100	1,000	10,000	100,000
$n \log n$	로그선형(log-linear)	700	10,000	140,000	1.8×10^6
n^2	2차(quadratic)	10,000	10^6	10^8	10^{10}
n^3	3차(cubic)	10^6	10^9	10^{12}	10^{15}
2^n	지수(exponential)	10^{30}	10^{300}	10^{3000}	10^{30000}

전형적인 증가 함수들의 플롯

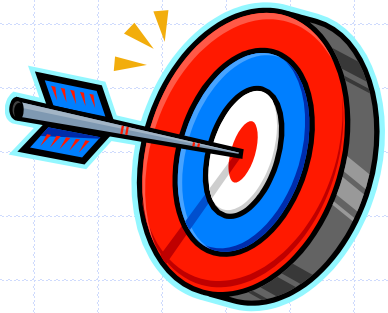


재귀 알고리즘

- 알고리즘 자신을 사용하여 정의된 알고리즘을 **재귀적**(recursive)이라고 말한다
  - **비재귀적**(nonrecursive) 또는 **반복적**(iterative) 알고리즘과 대조
- 재귀의 요소
  - **재귀 케이스**(recursion): 차후의 재귀호출은 **작아진** 부문제들(subproblems)을 대상으로 이루어진다
  - **베이스 케이스**(base case): 부문제들이 충분히 작아지면, 알고리즘은 재귀를 사용하지 않고 이들을 직접 해결한다

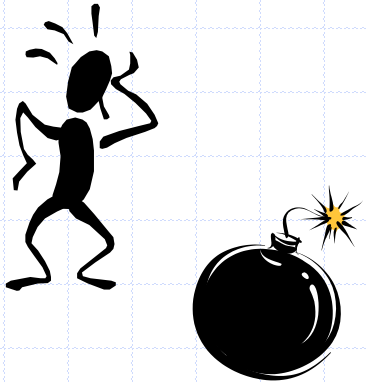
```
Alg sum(n)
1. if (n = 1)                    {base case}
       return 1
   else                          {recursion}
       return n + sum(n − 1)
```

기본 규칙



- **베이스 케이스**
  - 베이스 케이스를 항상 가져야 하며, 이 부분은 재귀 없이 해결 가능
- **진행 방향**
  - 재귀적으로 해결되어야 할 경우, 재귀호출은 항상 베이스 케이스를 향하는 방향으로 진행
- **정상작동 가정**
  - 모든 재귀호출이 제대로 작동한다고 가정!
- **적절한 사용**
  - 꼭 필요할 때만 사용 – 저장/복구 때문에 성능 저하

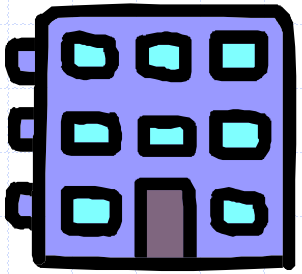
나쁜 재귀



- 잘못 설계된 재귀
  - **베이스 케이스:** 없음
    - **예:** sum1
  - **재귀 케이스:** 도달 불능 – 즉, 베이스 케이스를 향해 재귀하지 않음
    - **예:** sum2
- 나쁜 재귀 사용의 영향
  - 부정확한 결과
  - 미정지(nontermination)
  - 저장을 위한 기억장소 고갈

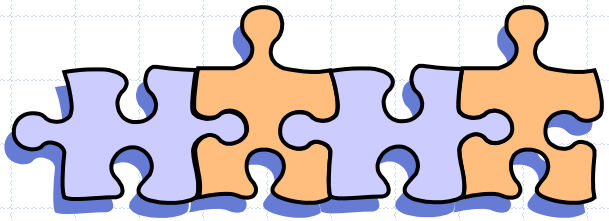
```
Alg sum1(n)
1. return n + sum1(n − 1)

Alg sum2(n)
1. if (n = 1)                    {base case}
       return 1
   else                          {recursion}
       return n + sum2(n + 1)
```

데이터구조의 기본 재료



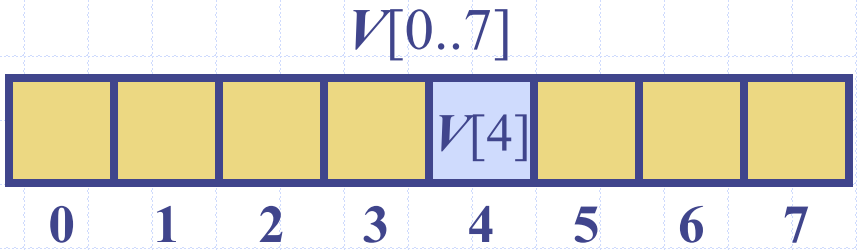
- 건물과 비유
  - **데이터구조:** 재료, 자재, 구조
  - **알고리즘:** 조명, 냉난방, 환기, 개폐 시스템
- 기본 재료
  - **배열**
  - **연결리스트**
- 차후의 고급 데이터구조를 구축하는 기본 재료
- 추상적이기 보다는 구체적(컴퓨터 구조에 가까움)

배열



- **배열**(array): 순차 기억장소에 할당된 유한 개수의 동일 자료형 데이터원소들
  - **배열명**(array name), V: 배열 전체를 일컫는 기호
  - **배열크기**(array size), N: 원소를 저장하는 셀들의 개수
  - **배열첨자**(array index), i: 셀의 순위 (즉, 상대적 위치)
    - 시작: 0 또는, 일반적으로, LB(lower bound)
    - 끝: $N-1$ 또는, 일반적으로, UB(upper bound)
  - **배열원소**(array element), $V[i]$: 배열 V의 첨자 i에 저장된 원소
  - **배열표시**(array denotation): $V[LB..UB]$

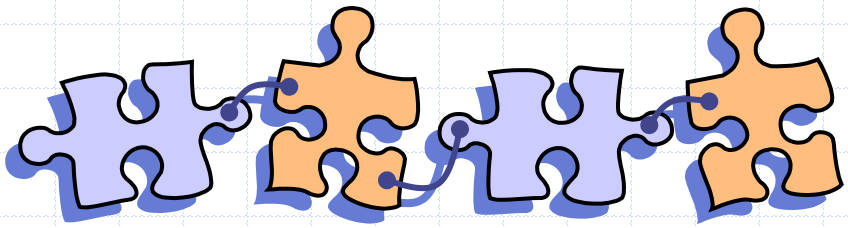
예:
한 학생의 8과목에서의 점수



다차원 배열

- 1차원 배열과 다른 점
  - 배열의 방들은 $n > 1$차원에 할당
  - **배열크기:** 각 차원 크기의 곱
  - **배열첨자**, $i_1, i_2, \ldots, i_n$: 각 차원에서의 방의 순위(즉, 상대적 위치)
    - 시작: $0, 0, \ldots, 0$, 또는 $LB_1, LB_2, \ldots, LB_n$
    - 끝: $N_1 - 1, N_2 - 1, \ldots, N_n - 1$, 또는 $UB_1, UB_2, \ldots, UB_n$
  - **배열원소**, $V[i_1, i_2, \ldots, i_n]$: 배열첨자들 $i_1, i_2, \ldots, i_n$에 저장된 원소
  - **배열표시**
    - $LB = 0$이면, $V[N_1 \times N_2 \ \ldots \times N_n]$
    - 일반적으로, $V[LB_1..UB_1, LB_2..UB_2, \ldots, LB_n..UB_n]$

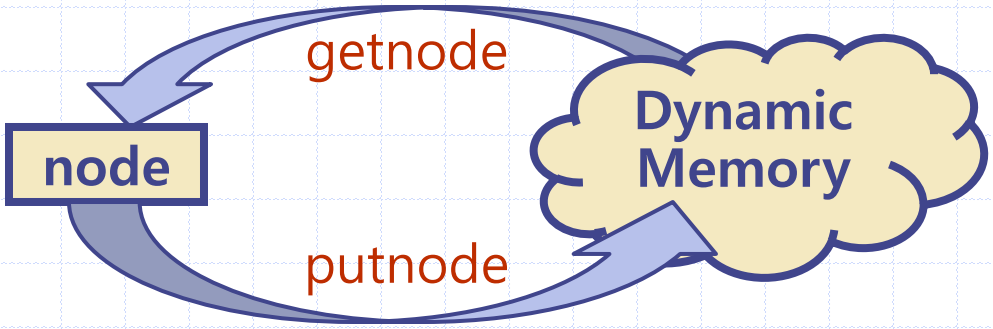
연결리스트



- **연결리스트**(linked list): 동적메모리에 할당된, 링크에 의해 연결된 유한 개수의 데이터원소 노드들
  - **연결리스트 명**(linked list name), L : 연결리스트의 시작 위치, 즉, 첫 노드의 주소
  - **연결리스트 크기**(linked list size), n : 연결리스트내 노드 수
- 연결리스트의 종류
  - 단일연결리스트(singly linked lists)
  - 이중연결리스트(doubly linked lists)
  - 원형연결리스트(circularly linked lists)
  - 헤더 및 트레일러 연결리스트(linked lists with header and trailer)
  - 이들의 복합

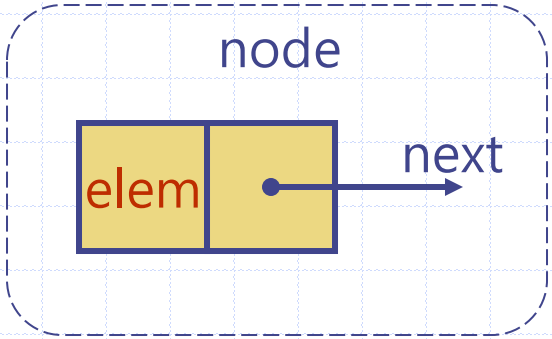
노드에 대한 메모리 할당

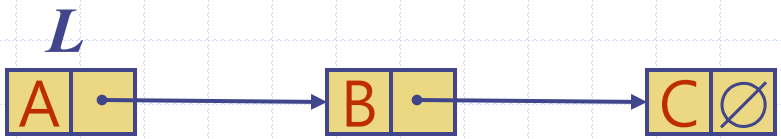
- **노드**(node): 한 개의 데이터원소를 저장하기 위해 동적 메모리(dynamic memory)에 할당된 메모리
- 노드를 위한 메모리의 동적 할당(allocation)과 해제(deallocation)는 실행시간에 system call에 의해 처리
  - getnode(): 노드를 할당하고 그 노드의 주소를 반환 (동적 메모리가 고갈된 시점이면 널포인터를 반환)
  - putnode(i): 주소 ***i***의 노드에 할당되었던 메모리의 사용을 해제하고 이를 동적 메모리에 반환 (메모리 재활용을 위함)



단일연결리스트

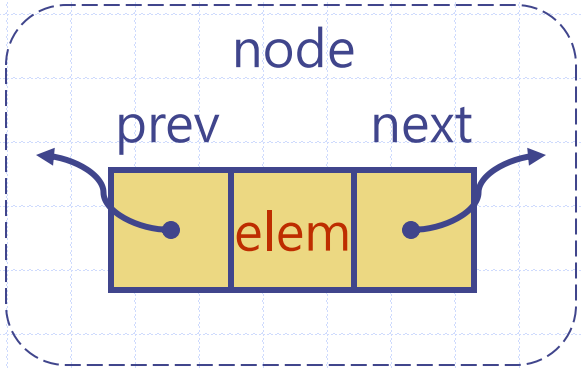
- 연속 노드로 구성된, 가장 단순한 연결 데이터구조 → 구조체 활용 (C언어)
- 노드 저장내용
  - **원소**(element): 데이터원소
  - **링크**(link): 다음 노드의 주소 – 다음 노드가 없는 경우 널링크(∅)를 저장
- 접근점
  - 첫 노드, 즉, **헤드노드**(head node)의 주소



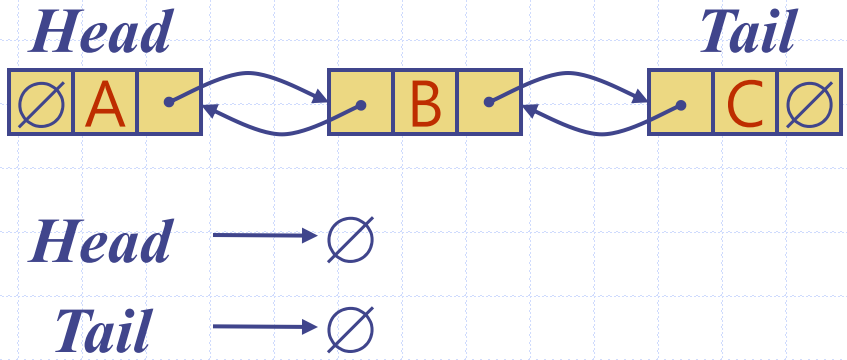


이중연결리스트

- 추가 링크를 사용하여 역방향 순회도 가능
- 노드 저장내용
  - **원소**(element)
  - **링크**(link): 다음 노드의 주소
  - **링크**(link): 이전 노드의 주소

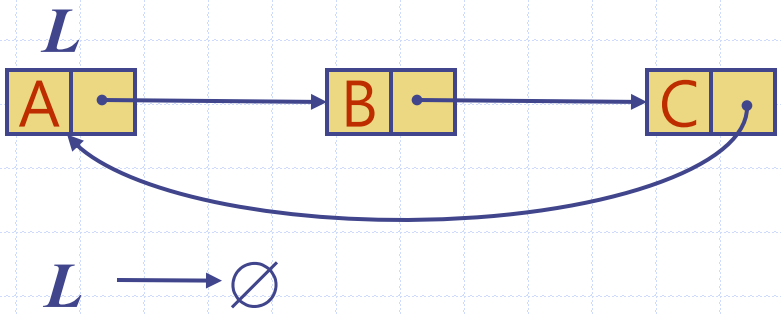


- 접근점
  - 헤드노드(head node)의 주소
  - 테일노드(tail node)의 주소
  - 주의 : 멤버 변수를 가리키는 것이 아님



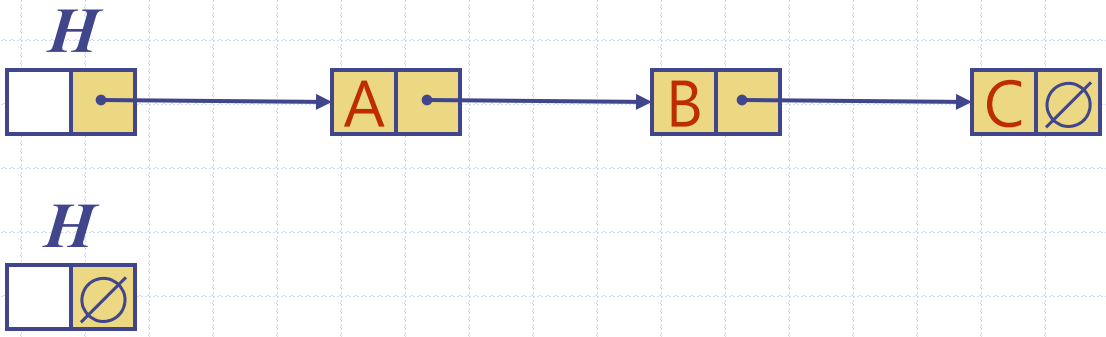
원형연결리스트

- 마지막 노드의 링크가 헤드노드의 주소
- 접근점
  - 헤드노드의 주소



헤더와 트레일러

- 헤드노드 바로 앞에 특별한 **헤더**(header) 노드를 추가하여 작업 편의성을 증진
- 같은 목적으로 테일노드 바로 뒤에 **트레일러**(trailer) 노드 추가 가능
- 특별노드 저장내용
  - 모조 원소(dummy element)
- 접근점
  - 헤더노드(또는 트레일러노드)의 주소

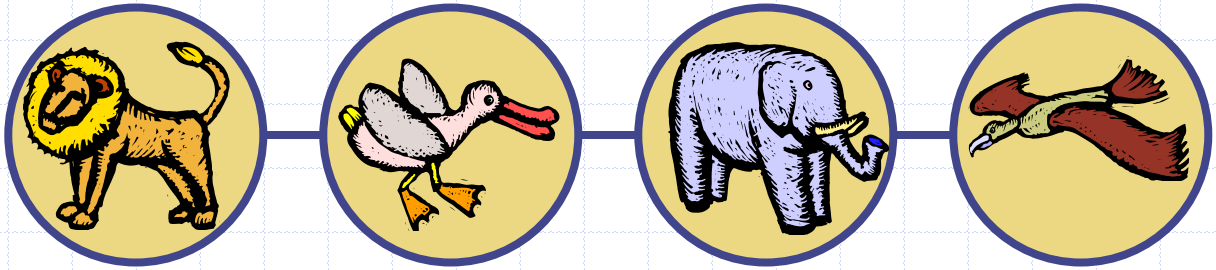


주요 자료구조

- 리스트
- 집합
- 스택
- 큐
- 트리

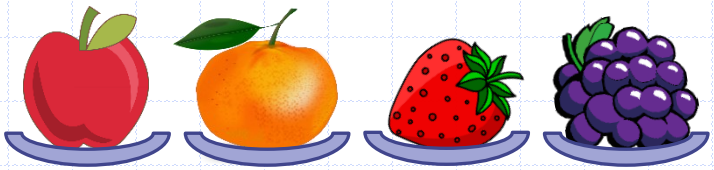
리스트 ADT

- 리스트 ADT는 연속적인 임의 개체들을 모델링
- 원소(element)에 대한 접근 도구
  - **순위**(rank)



리스트 ADT 메쏘드

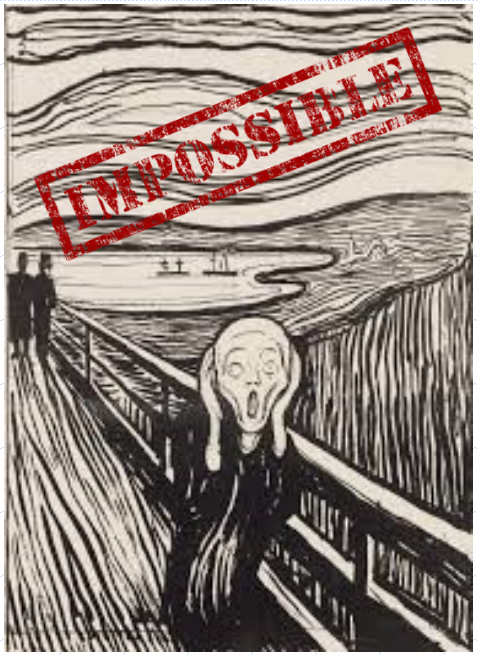
- 원소는 그 **순위**(rank), 즉, 그 원소 앞의 원소 개수를 특정함으로써 접근, 삽입, 또는 삭제



- 일반 메쏘드
  - boolean isEmpty()
  - integer size()
  - iterator elements()
- 접근 메쏘드
  - element get(r)
- 갱신 메쏘드
  - element set(r, e)
  - add(r, e),
    addFirst(e),
    addLast(e)
  - element remove(r),
    element removeFirst(),
    element removeLast()

예외

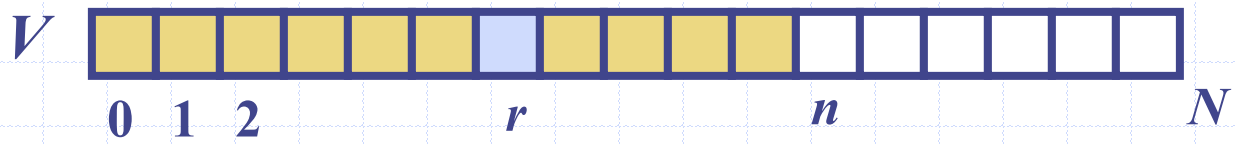
- **예외**(exception): 어떤 ADT 작업을 실행하고자 할 때 발생할 수도 있는 오류 상황



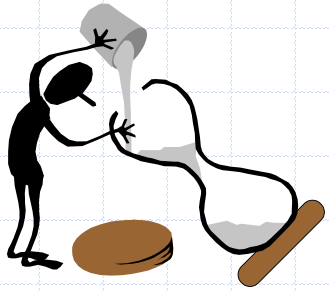
- "실행 불가능한 작업 때문에 예외를 **발령한다**(throw)"고 말한다
- 리스트 ADT에서 발령 가능한 예외들
  - invalidRankException()
  - fullListException()
  - emptyListException()

배열을 이용한 구현

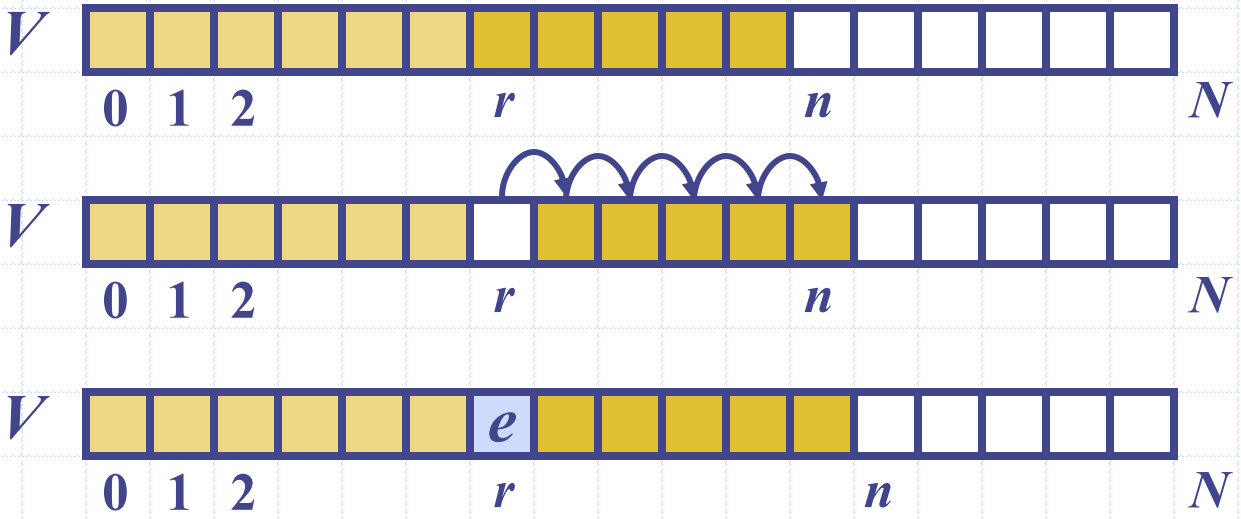
- N개의 단순 또는 복잡한 원소들로 구성된 배열 V
- 변수 n으로 리스트의 크기, 즉 저장된 원소 개수를 관리
- 배열에서 **순위**는 0에서 출발
- 작업 get(r) 또는 set(r, e)는 $\mathbf{O}(1)$ 시간에 $V[r]$을 각각 반환 또는 저장하도록 구현
  - $r < 0$ 또는 $r > n - 1$ 인 경우 예외처리 필요



삽입(insertion)



- 작업 add(r, e)에서는, r 순위로 새 원소 e가 들어갈 빈 자리를 만들기 위해 $V[n-1], \ldots, V[r]$까지의 $n-r$개의 원소들을 **순방향**으로 이동(shift forward)
- 최악의 경우($r=0$), $\mathbf{O}(n)$ 시간 소요



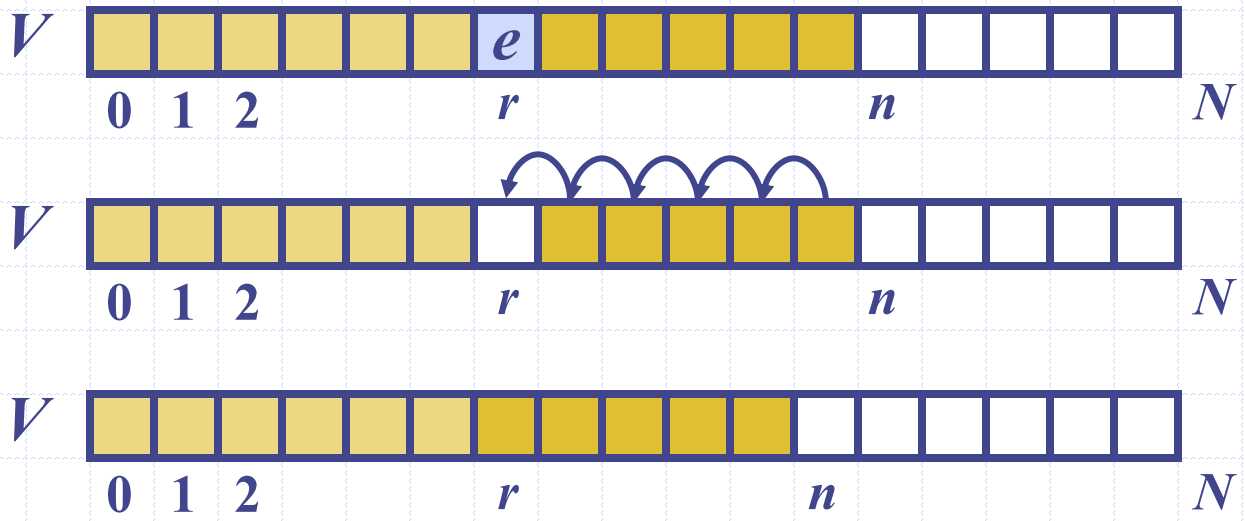
삽입 (conti.)

- 배열의 지정된 순위 *r*에 원소 *e*를 삽입

```
Alg add(r, e)
    input array V, integer N, n, rank
          r, element e
    output none

1. if (n = N)
          fullListException()
2. if ((r < 0) || (r > n))
          invalidRankException()
3. for i ← n – 1 downto r
          V[i + 1] ← V[i]
4. V[r] ← e
5. n ← n + 1
6. return
```

삭제(deletion)

- 작업 remove(r)에서는, 삭제된 원소에 의해 생긴 빈 자리를 채우기 위해 $V[r+1], \ldots, V[n-1]$까지의 $n-r-1$개의 원소들을 **역방향**으로 이동(shift backward)
- 최악의 경우($r=0$), $\mathbf{O}(n)$ 시간 소요



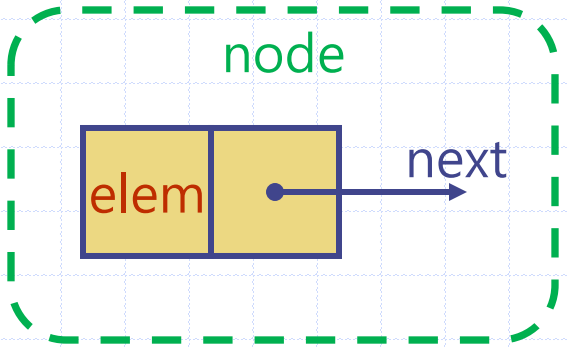
삭제 (conti.)

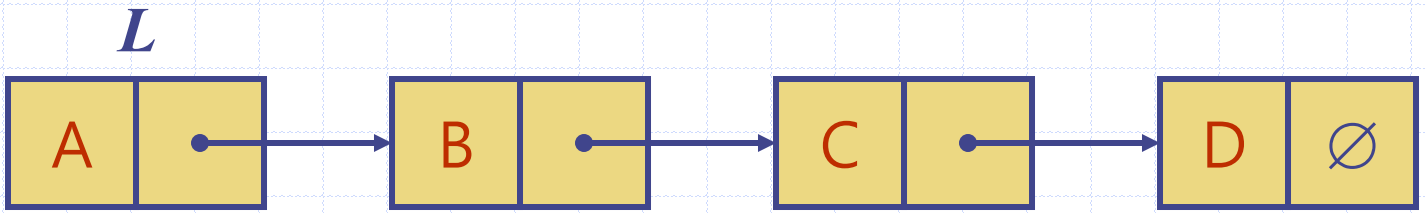
- 배열의 지정된 순위 *r*의 원소를 삭제하여 반환

```
Alg remove(r)
    input array V, integer N, n, rank
          r
    output element e

1. if ((r < 0) || (r > n – 1))
          invalidRankException()
2. e ← V[r]
3. for i ← r + 1 to n – 1
          V[i – 1] ← V[i]
4. n ← n – 1
5. return e
```

단일연결리스트

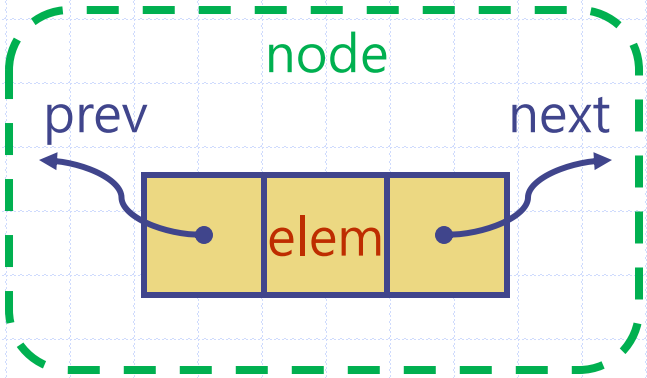
- **단일연결리스트**(singly linked list): 연속 노드로 구성된 구체적인 데이터구조
- 각 노드의 저장 내용
  - **원소**(element) (단순 또는 복잡)
  - 다음 노드를 가리키는 **링크**(link)

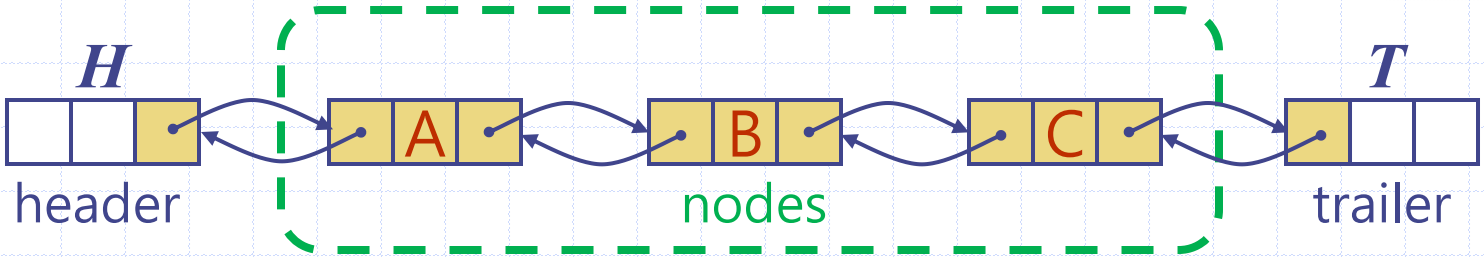




이중연결리스트

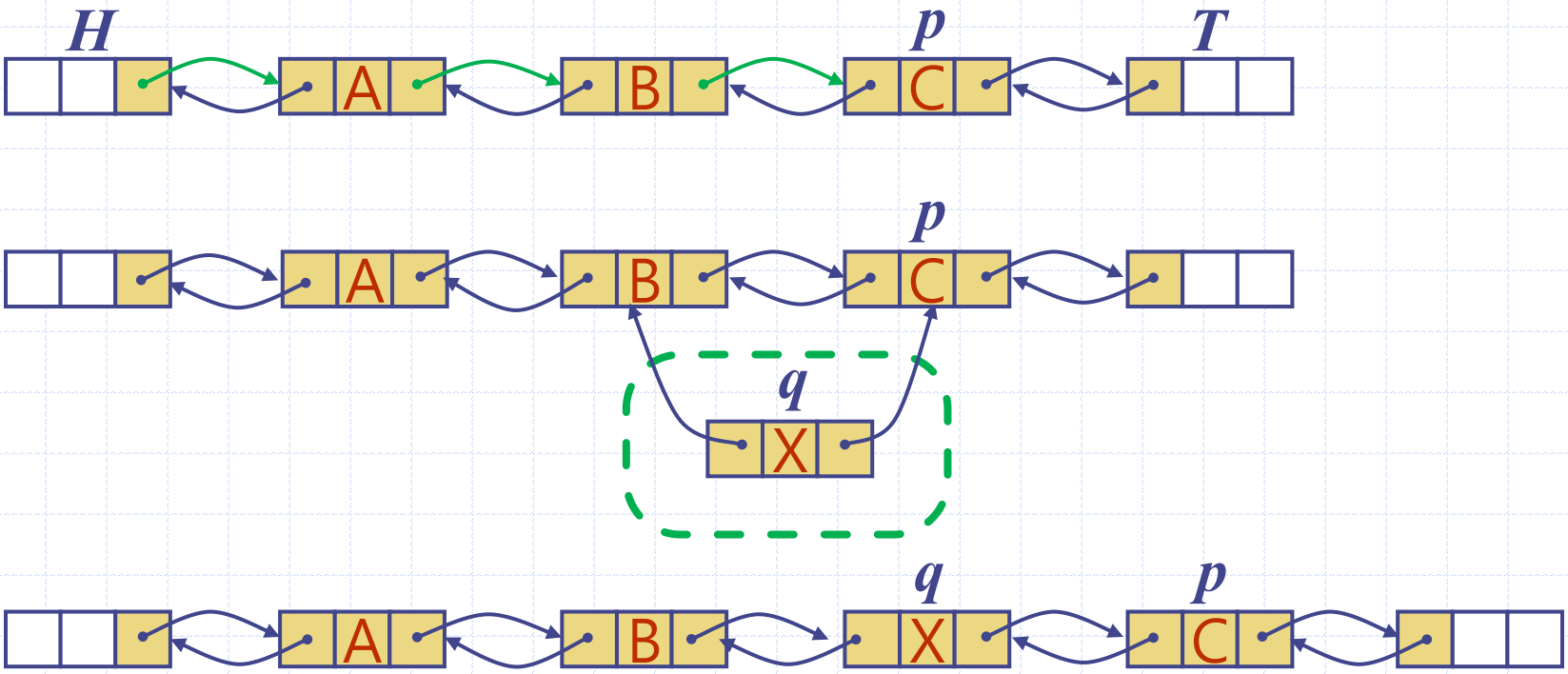
- **이중연결리스트**(doubly linked list)를 이용하면 리스트 ADT를 자연스럽게 구현 가능
- 각 노드의 필드
  - **원소**
  - 이전 노드를 가리키는 **링크**
  - 다음 노드를 가리키는 **링크**
- 특별 헤더 및 트레일러 노드





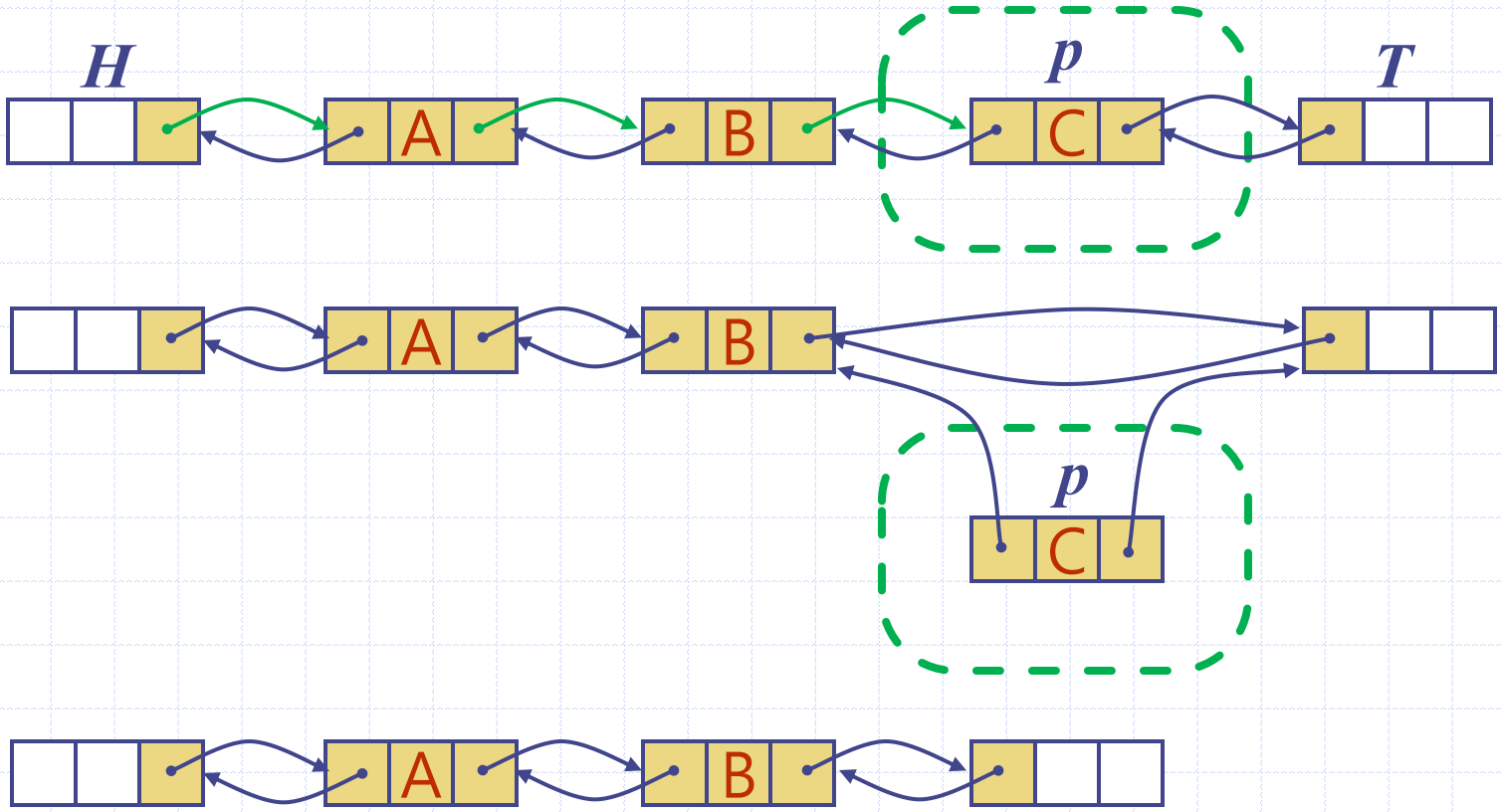
삽입

- 작업 add(r, X)의 시각화 – 여기서 r = 3

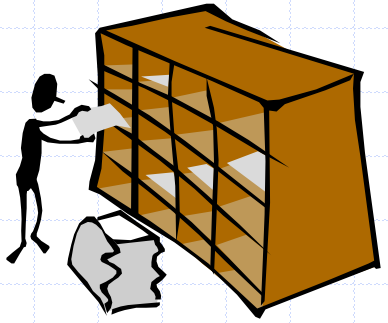


삭제

- remove(r)의 시각화 – 여기서 r = 3



그룹



- **개념:** 데이터원소들이 각각 상이한 **그룹**(즉, 카테고리)에 속한다
- **전제:** 각 그룹의 크기는 다양

- **예**
  - 쇼핑몰의 상품들
    - Maker X: x1
    - Maker Y: none
    - Maker Z: z1, z2
  - 대학의 강좌들
    - Prof. Kook: DS
    - Prof. Park: (no lecture)
    - Prof. Shin: DB, MM
  - 다항식의 항들
    - Exp 1: $3x^4$
    - Exp 3: $5x^3$, -4

설계 방안

A. 레코드의 리스트 사용
   1. 배열을 이용한 구현
   2. 연결리스트를 이용한 구현
B. 부리스트(sublist)들의 리스트 사용
   1. 2D 배열을 이용한 구현
   2. 연결리스트의 배열을 이용한 구현

리스트 확장: 공유

- **문제 상황:** 데이터 원소(element)들이 상이한 그룹(group)에 의해 **공유**(share)됨 (**예:** auction)
- **전제:** 각 관련 그룹에게 공유 데이터원소를 복제하는 것은 시간과 기억장소가 낭비되므로 허용하지 않는다

- **예**
  - 쇼핑몰의 상품들
    - Buyer A: x1, y1
    - Buyer B: z1
    - Buyer C: y1, z1, z2
  - 대학 강좌들
    - Student A: DS, OS
    - Student B: DB
    - Student C: OS, DB, MM
  - 인터넷 블로그들
    - Blogger A: a, b
    - Blogger B: c
    - Blogger C: b, c, d

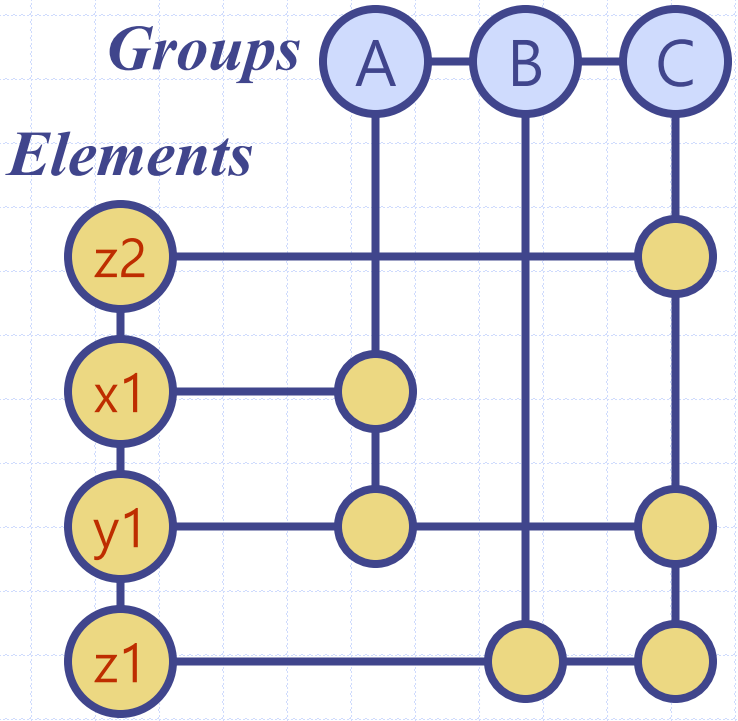
설계 방안: 공유

A. 레코드의 리스트 사용
   1. 배열을 이용한 구현
   2. 연결리스트를 이용한 구현
B. 포인터의 리스트 사용
   1. 배열을 이용한 구현
   2. 연결리스트를 이용한 구현
C. 다중리스트 사용
   1. 2D 배열 사용
   2. 다중 연결리스트를 이용한 구현

설계 방안 C: 다중리스트 사용

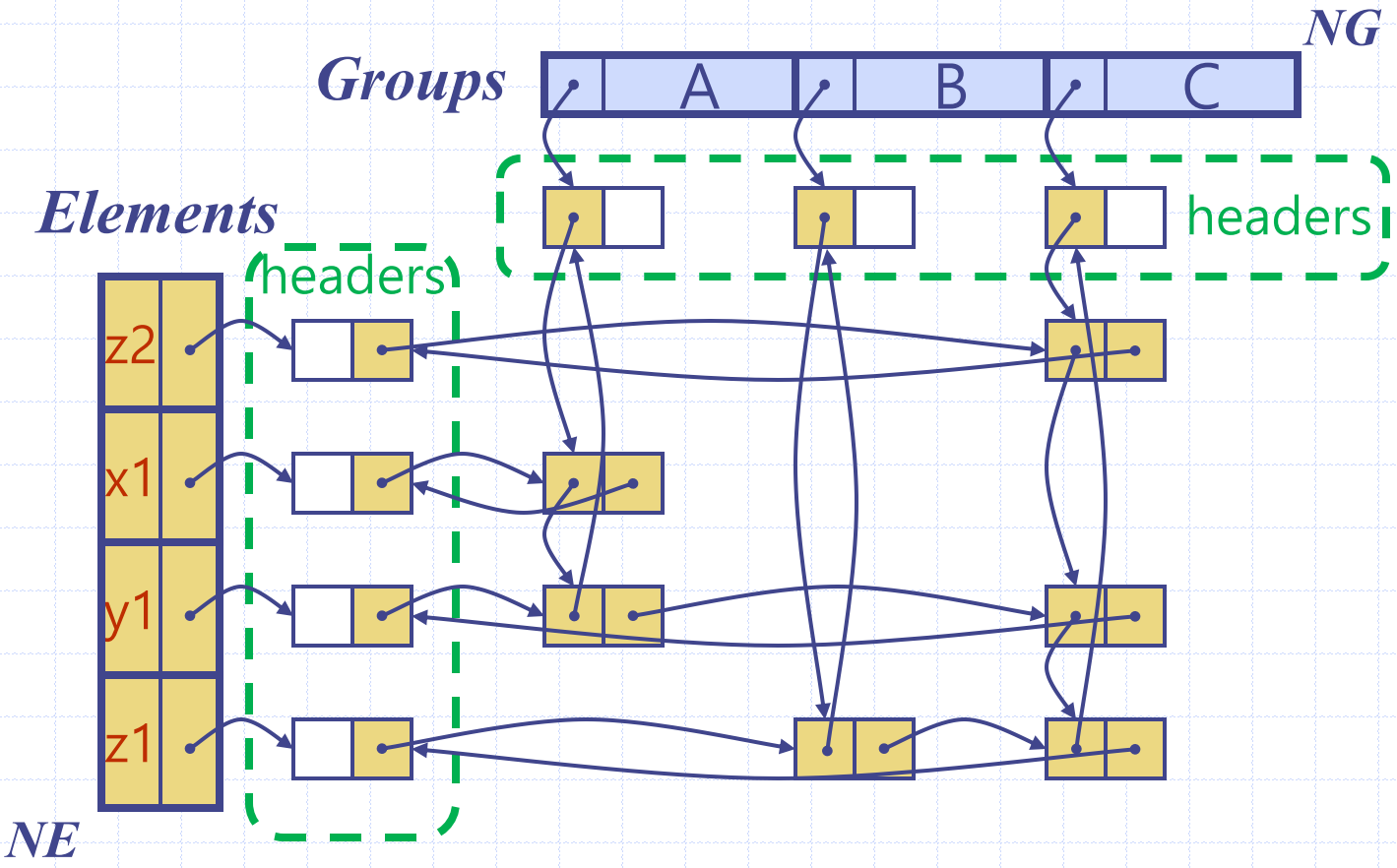
- 공유를 표현하기 위해, **원소**들의 리스트와 **그룹**들의 리스트가 상호 교차하는 형태의 다중리스트(multilist)를 사용
- 교차점 서브리스트는 (원소, 그룹) 관련성 여부를 표현
- **장점:** 특정원소 및 특정그룹 관련 작업 모두 격리 처리 가능

예: 쇼핑몰의 상품들

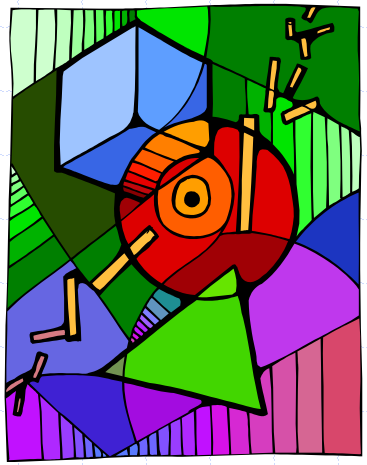


다중연결리스트 이용 (conti.)

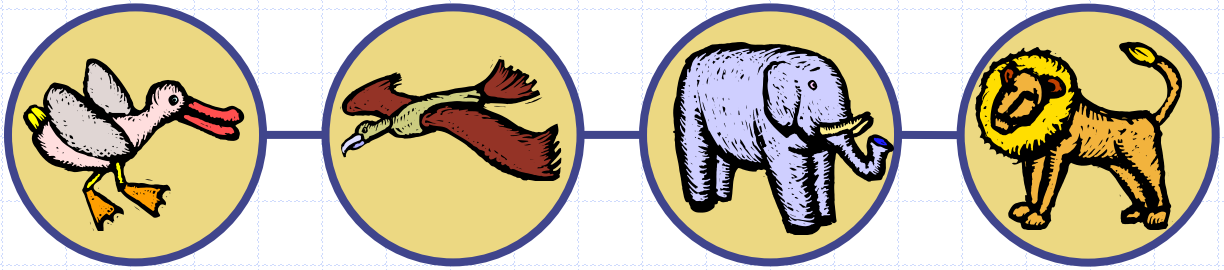
예: 쇼핑몰의 상품들



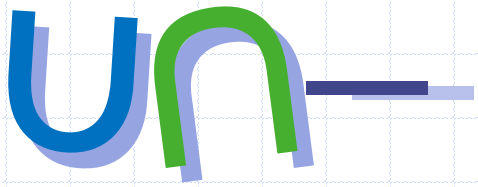
집합 ADT



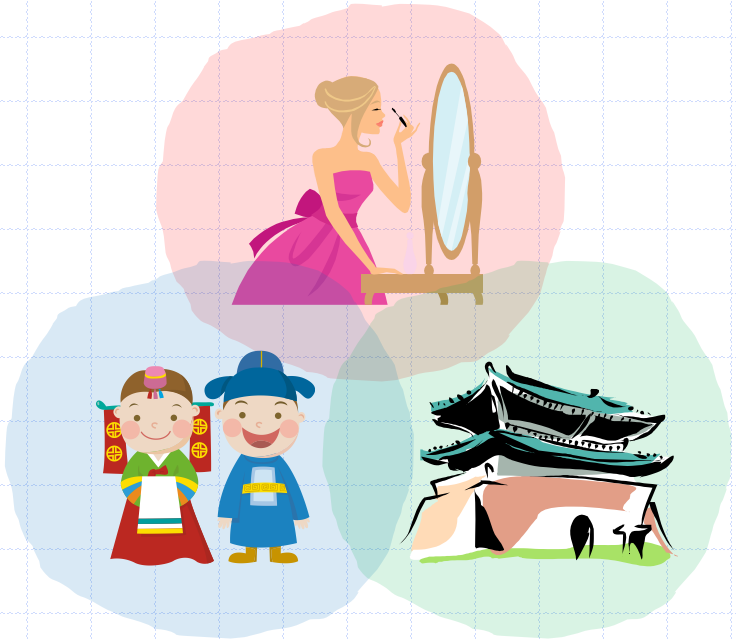
- 집합 ADT는 **유일한** 개체들을 담는 용기를 모델링한다
- 집합 ADT 관련 작업들의 효율적인 구현을 위해, 집합을 집합 원소들의 **정렬된** 리스트로 표현



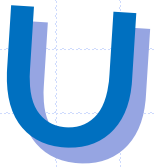
집합 ADT 메쏘드



- 집합 A에 관한 주요 메쏘드
  - set union(B): 집합 B와의 **합집합**을 반환
  - set intersect(B): 집합 B와의 **교집합**을 반환
  - set subtract(B): 집합 B를 차감한 **차집합**을 반환
- 집합 A와 B에 관한 주요 작업의 실행시간은 최대 $O(|A| + |B|)$이 되어야 함



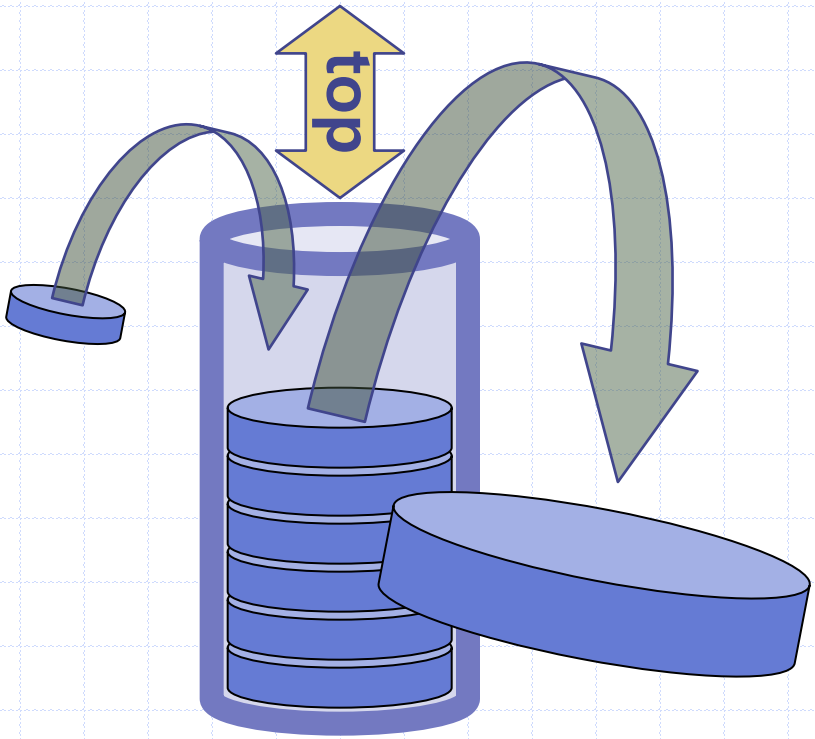
합집합(union)



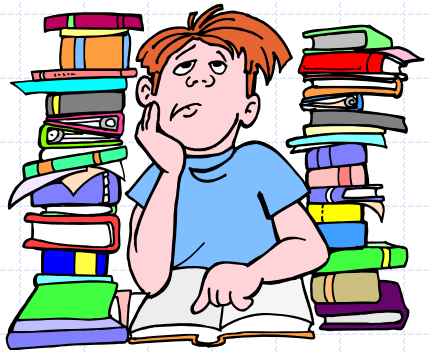
```
Alg union(B)
    input set A, B
    output set A∪B

1. C ← empty list
2. while (!A.isEmpty() & !B.isEmpty())
        a, b ← A.get(1), B.get(1)
        if (a < b)
            C.addLast(a)
            A.removeFirst()
        elseif (a > b)
            C. addLast(b)
            B.removeFirst()
        else {a = b}
            C.addLast(a)
            A.removeFirst()
            B.removeFirst()
3. while (!A.isEmpty())
        a ← A.get(1)
        C.addLast(a)
        A.removeFirst()
4. while (!B.isEmpty())
        b ← B.get(1)
        C.addLast(b)
        B.removeFirst()
5. return C
                                {Total O(|A| + |B|)}
```

스택 ADT

- 스택 ADT는 임의의 개체를 저장
- 삽입과 삭제는 **후입선출**(Last-In First-Out, LIFO) 순서를 따른다
- 삽입과 삭제는 스택의 **top**이라 불리는 위치에서 수행



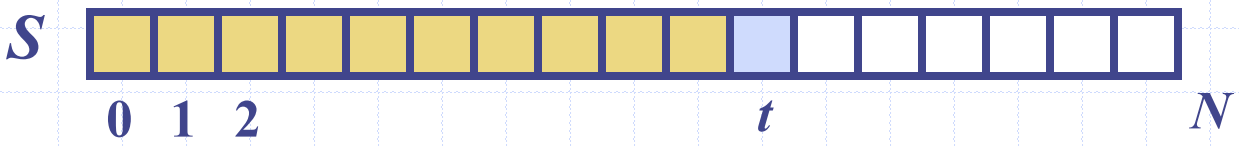
스택 ADT 메쏘드



- 주요 스택 메쏘드
  - push(e): 원소를 삽입
  - element pop(): 가장 최근에 삽입된 원소를 삭제하여 반환
- 보조 스택 메쏘드
  - element top(): 가장 최근에 삽입된 원소를 (삭제하지 않고) 반환
  - integer size(): 저장된 원소의 수를 반환
  - boolean isEmpty(): 아무 원소도 저장되어 있지 않고 비어 있는지 여부를 반환
  - iterator elements(): 스택 원소 전체를 반환
- 예외
  - emptyStackException(): 비어 있는 스택에서 삭제나 top을 시도할 경우 발령
  - fullStackException(): 만원 스택에서 삽입을 시도할 경우 발령

배열에 기초한 스택

- 크기 N의 배열을 사용
- 원소들을 배열의 왼쪽에서 오른쪽으로 추가
- 변수 t를 사용하여 **top** 원소의 첨자를 관리

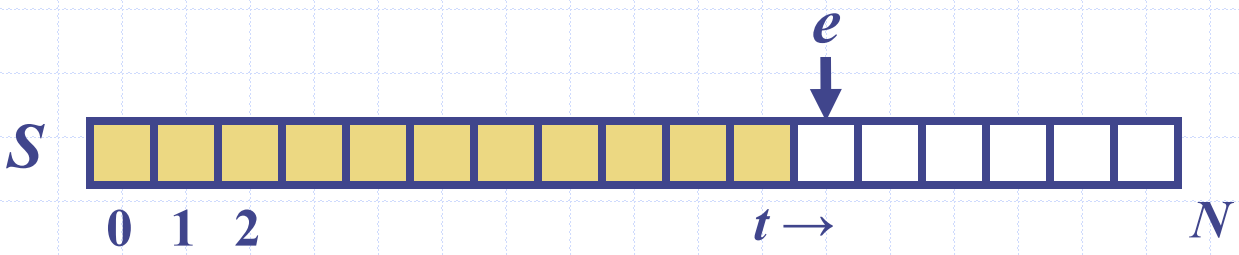


삽입

- 스택이 만원인 경우, push 작업은 fullStackException을 발령
  - 배열에 기초한 구현의 한계
  - **구현상의 오류**일 뿐 스택 ADT 취급 상 **논리적 오류**는 아님

Alg *push*(*e*)
  input stack *S*, size *N*, top *t*,
    element *e*
  output none

1. **if** ($t = N - 1$)
    fullStackException()
2. $t \leftarrow t + 1$
3. $S[t] \leftarrow e$
4. **return**

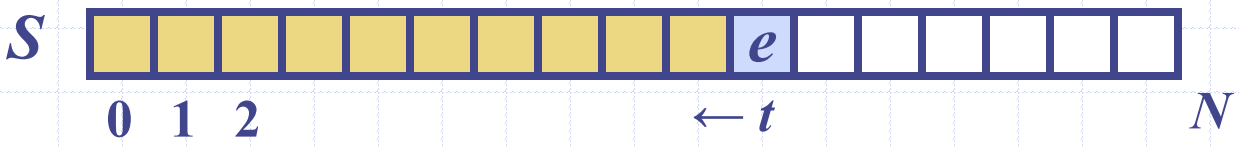


삭제

- 스택이 빈 경우, pop 작업은 emptyStackException을 발령
  - 스택 ADT 취급 상 **논리적 오류**

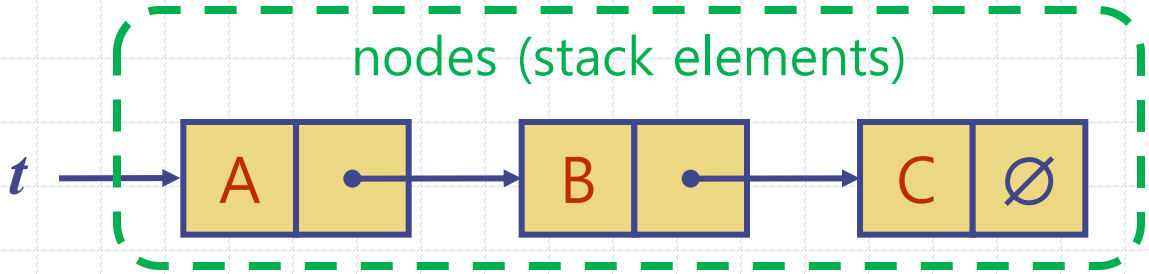
Alg ***pop***()
  input stack ***S***, size ***N***, top ***t***
  output element

1. **if** (***isEmpty***())
   emptyStackException()
2. ***t*** ← ***t*** – 1
3. **return** ***S***[***t*** + 1]

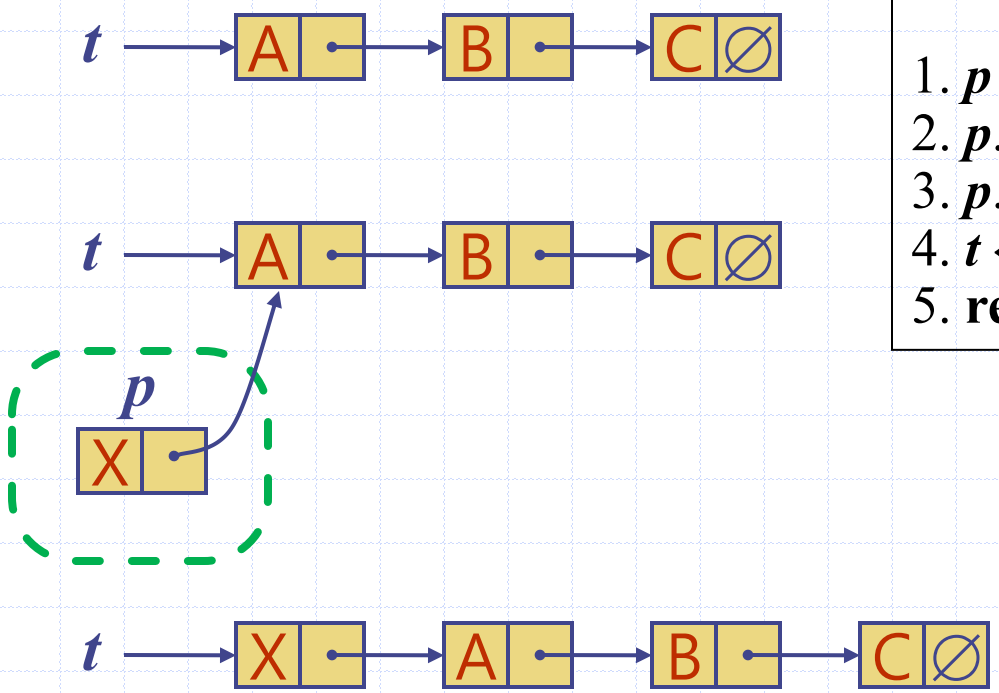


연결리스트에 기초한 스택

- **단일연결리스트**를 사용하여 스택 구현 가능
  - 삽입과 삭제가 특정위치에서만 수행되므로, **헤더노드**는 불필요
- top 원소를 연결리스트의 첫 노드에 저장하고 이곳을 t로 가리키게 한다
- 기억장소 사용: $\mathbf{O}(n)$
- 스택 ADT의 각 작업: $\mathbf{O}(1)$



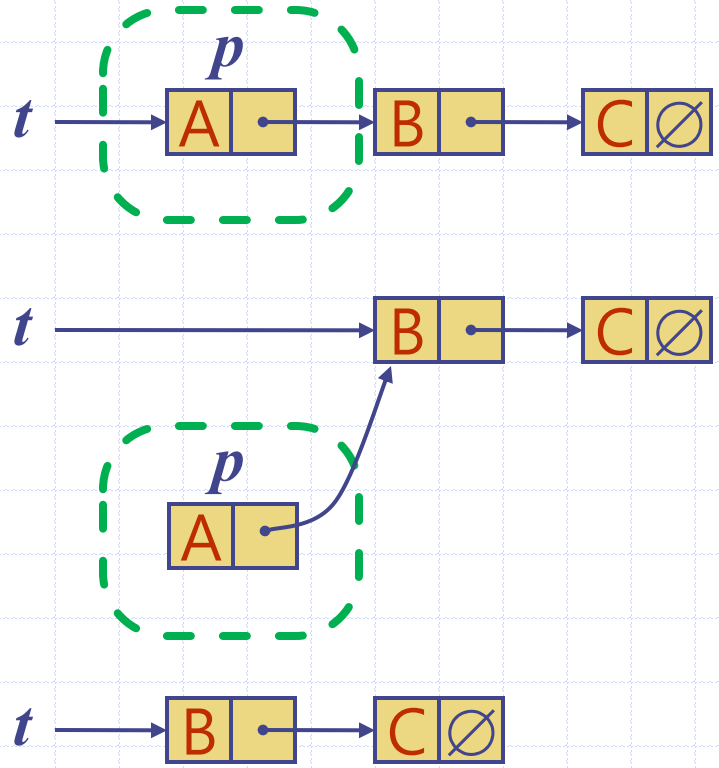
삽입



Alg ***push*(*e*)**
input top ***t***, element ***e***
output none

1. ***p*** ← ***getnode***()
2. ***p*.elem** ← ***e***
3. ***p*.next** ← ***t***
4. ***t*** ← ***p***
5. **return**

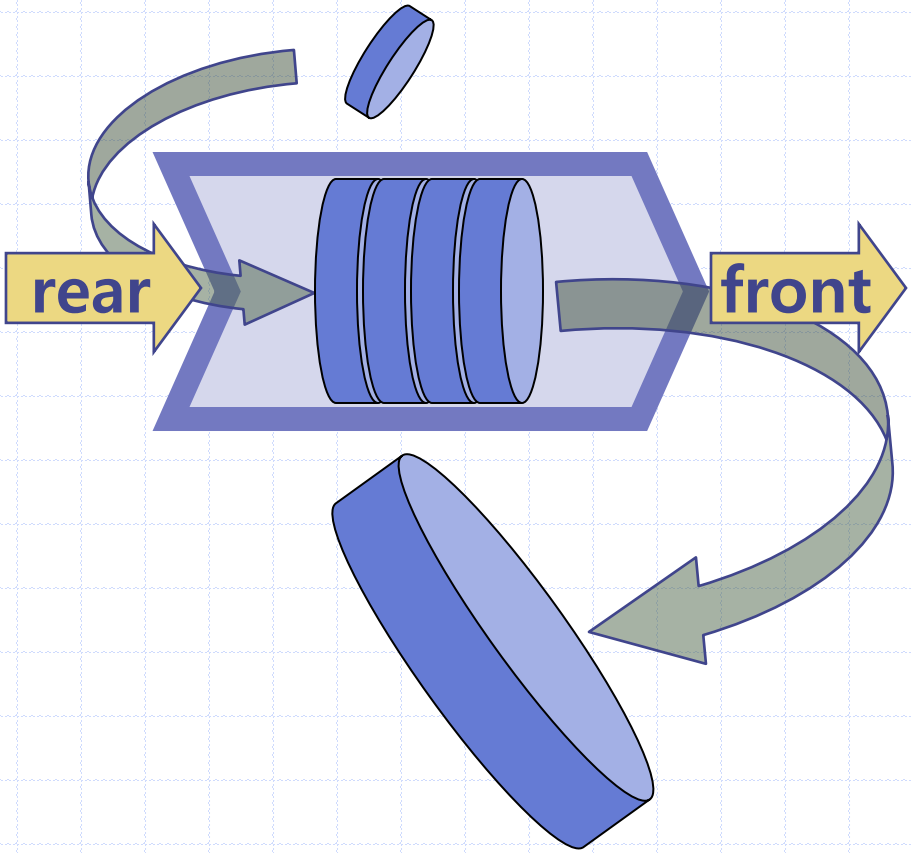
삭제




```
Alg isEmpty()
    input top t
    output boolean

1.  return t = ∅

Alg pop()
    input top t
    output element

1. if (isEmpty())
        emptyStackException()
2. e ← t.elem
3. p ← t
4. t ← t.next
5. putnode(p)
6. return e
```

큐 ADT

- 큐 ADT는 임의의 개체들을 저장
- 삽입과 삭제는 **선입선출**(First-In First-Out, FIFO) 순서를 따른다
- 삽입은 큐의 **뒤**(rear), 삭제는 큐의 **앞**(front)이라 불리는 위치에서 수행

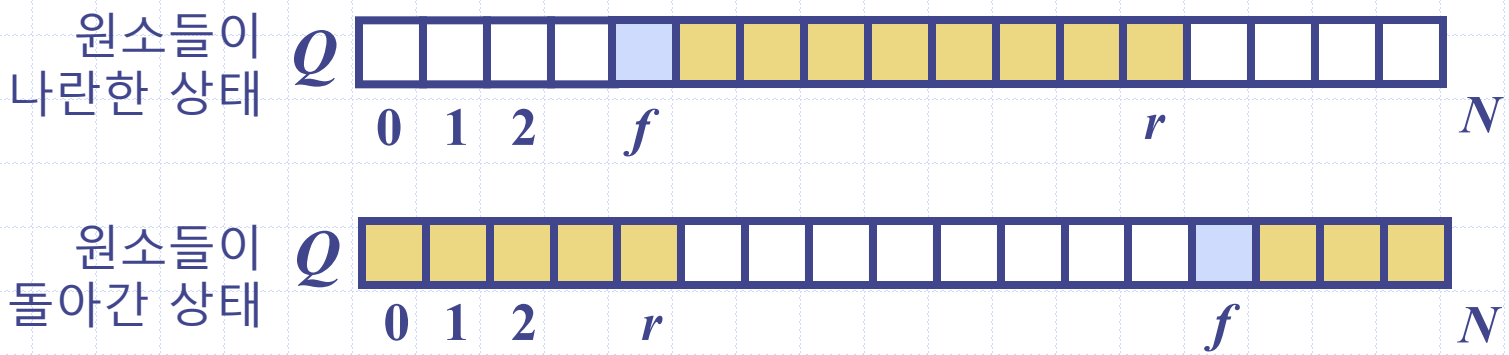


큐 ADT 메쏘드

- 주요 큐 메쏘드
  - enqueue(e): 큐의 뒤에 원소를 삽입
  - element dequeue(): 큐의 앞에서 원소를 삭제하여 반환
- 보조 큐 메쏘드
  - element front(): 큐의 앞에 있는 원소를 (삭제하지 않고) 반환
  - integer size(): 큐에 저장된 원소의 수를 반환
  - boolean isEmpty(): 큐가 비어 있는지 여부를 반환
  - iterator elements(): 큐 원소 전체를 반환
- 예외
  - emptyQueueException(): 비어 있는 큐에 대해 삭제 또는 front를 수행 시도할 경우 발령
  - fullQueueException(): 만원 큐에 대해 삽입을 수행 시도할 경우 발령

배열에 기초한 큐

- 크기 N의 배열을 **원형**으로 사용
  - 선형배열을 사용하면 비효율적임
- 두 개의 변수를 사용하여 front와 rear 위치를 기억
  - f: front 원소의 첨자
  - r: rear 원소의 첨자
- **참고:** f가 front 원소가 저장된 위치의 한 셀 앞을 가리키도록 정의하는 방식도 가능 – 단, 이에 상응하여 큐 관련 메쏘드 수정 필요
- 빈 큐를 만원 큐로부터 차별하기 위해:
  - **한 개의 빈 방**을 예비
  - **대안:** 원소 개수를 유지

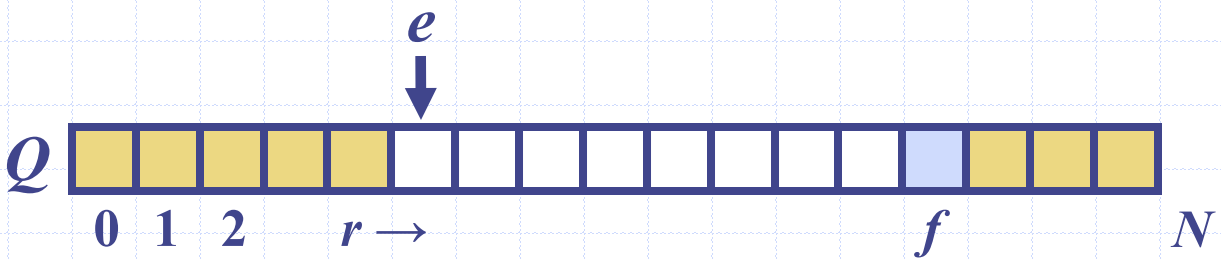


삽입

- 큐가 만원인 경우, enqueue 작업은 fullQueueException을 발령
  - 배열에 기초한 구현의 한계
  - **구현상의 오류**일 뿐, 큐 ADT 취급 상 **논리적 오류**는 아님

Alg ***enqueue***(***e***)
  input queue ***Q***, size ***N***, front ***f***,
    rear ***r***, element ***e***
  output none

1. **if** (***isFull***())
    fullQueueException()
2. ***r*** ← (***r*** + 1) % ***N***
3. ***Q***[***r***] ← ***e***
4. **return**

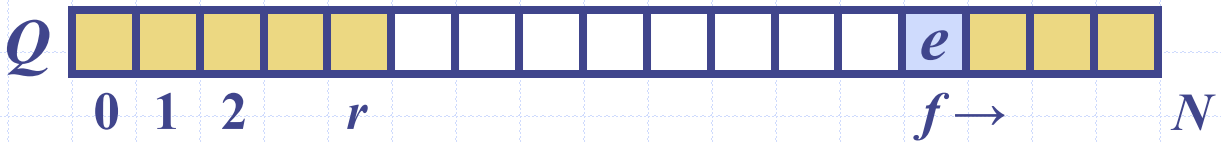


삭제

- 큐가 빈 경우, dequeue 작업은 emptyQueueException을 발령
  - 큐 ADT 취급 상 **논리적 오류**

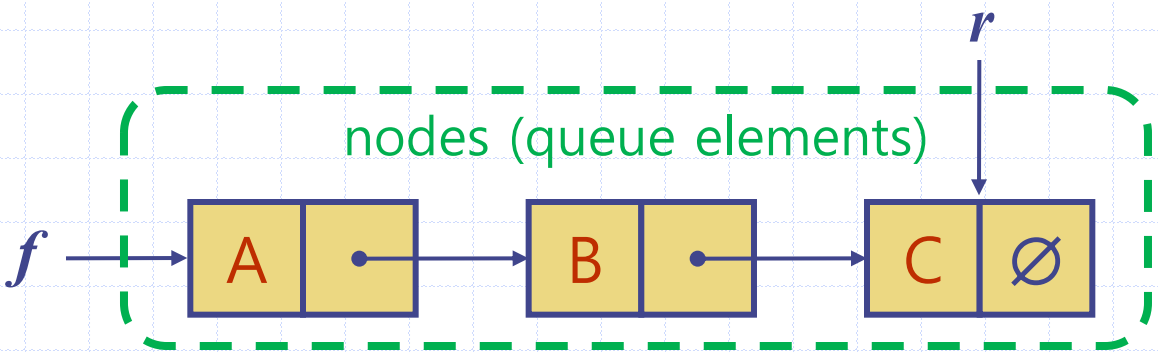
Alg ***dequeue***()
  input queue ***Q***, size ***N***, front ***f***,
    rear ***r***
  output element

1. **if** (***isEmpty***())
    emptyQueueException()
2. ***e*** ← ***Q***[***f***]
3. ***f*** ← (***f*** + 1) % ***N***
4. **return** ***e***



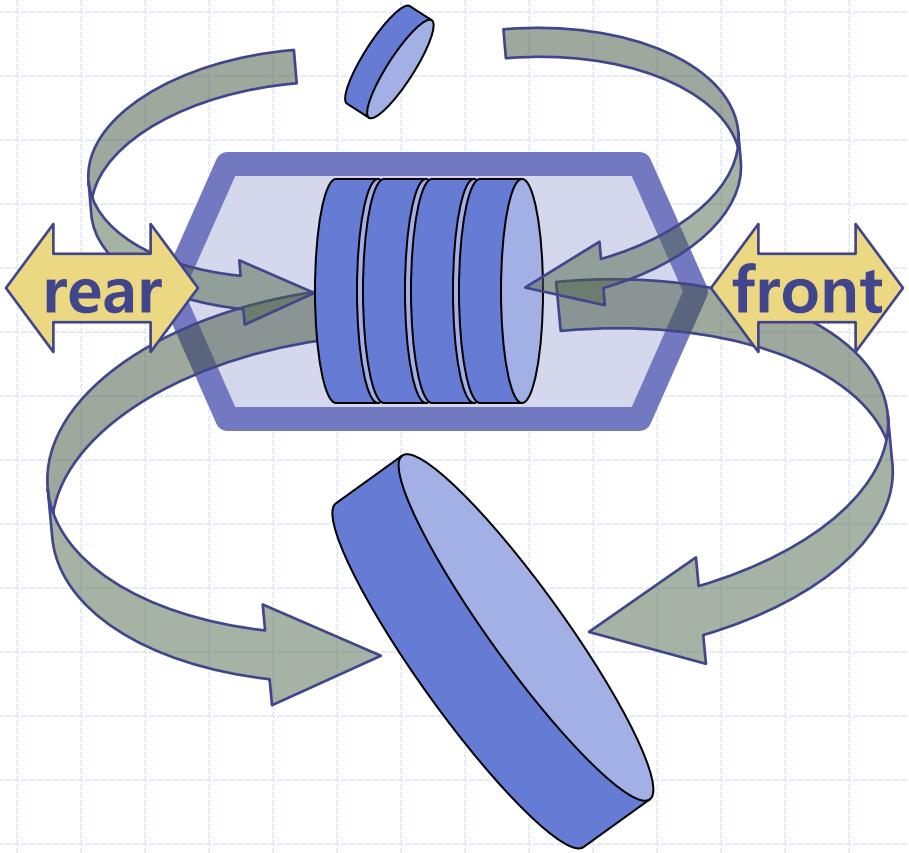
연결리스트에 기초한 큐

- **단일연결리스트**를 사용하여 큐 구현 가능
  - 삽입과 삭제가 특정위치에서만 수행되므로, **역방향링크**는 불필요 (**참고:** 스택의 경우 **헤더노드** 불필요)
- front 원소를 연결리스트의 첫 노드에, rear 원소를 끝 노드에 저장하고 f와 r로 각각의 노드를 가리키게 한다
- 기억장소 사용: $\mathbf{O}(n)$
- 큐 ADT의 각 작업: $\mathbf{O}(1)$



데크 ADT

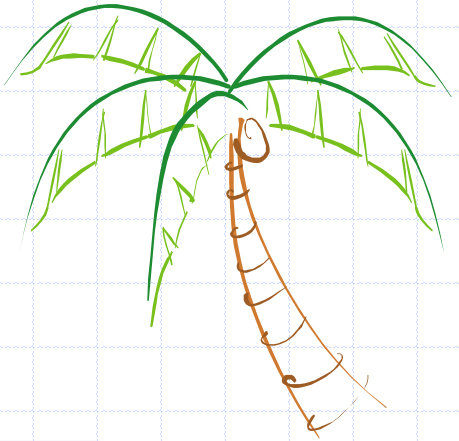
- 데크 ADT는 임의의 개체들을 저장
- **데크**(double-ended queue, **deque**)는 스택과 큐의 합체 방식으로 작동
- 삽입과 삭제는 **앞**(front)과 **뒤**(rear)라 불리는 양쪽 끝 위치에서 이루어진다



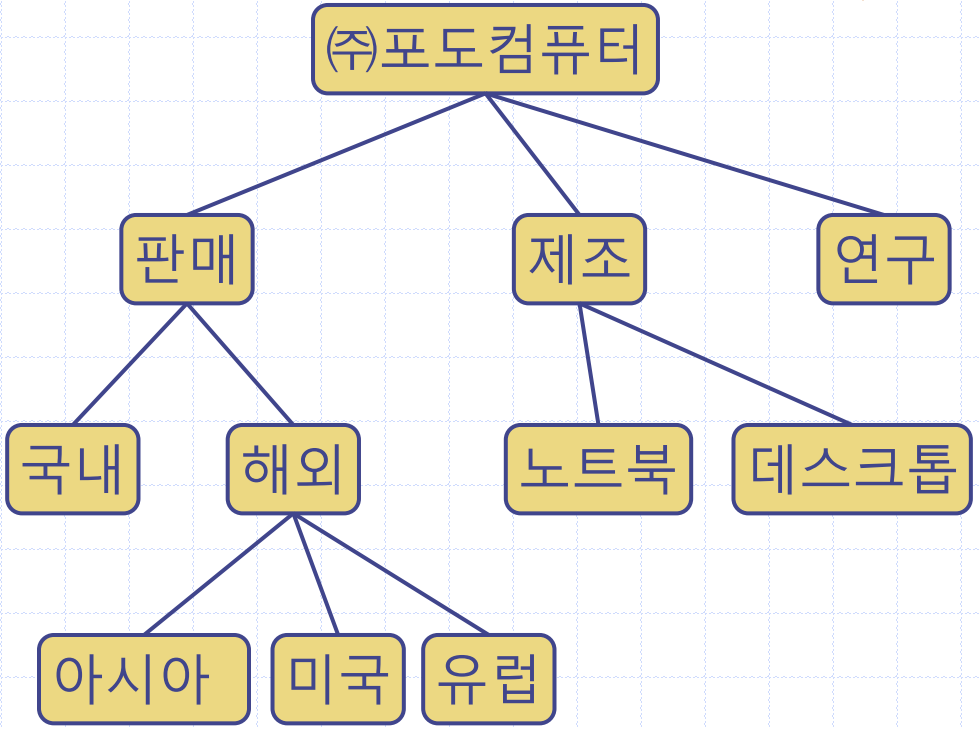
데크 ADT 메쏘드

- 주요 메쏘드
  - push(e): front 위치에 원소를 삽입
  - element pop(): front 위치의 원소를 삭제하여 반환
  - inject(e): rear 위치에 원소를 삽입
  - element eject(): rear 위치의 원소를 삭제하여 반환
- 보조 메쏘드
  - element front(): front 위치의 원소를 반환
  - element rear(): rear 위치의 원소를 반환
  - integer size(): 데크에 저장된 원소의 수를 반환
  - boolean isEmpty(): 데크가 비어 있는지 여부를 반환
- 예외
  - emptyDequeException(): 비어 있는 데크로부터 삭제를 시도할 경우 발령
  - fullDequeException(): 만원인 데크에 대해 삽입을 시도할 경우 발령

트리 ADT

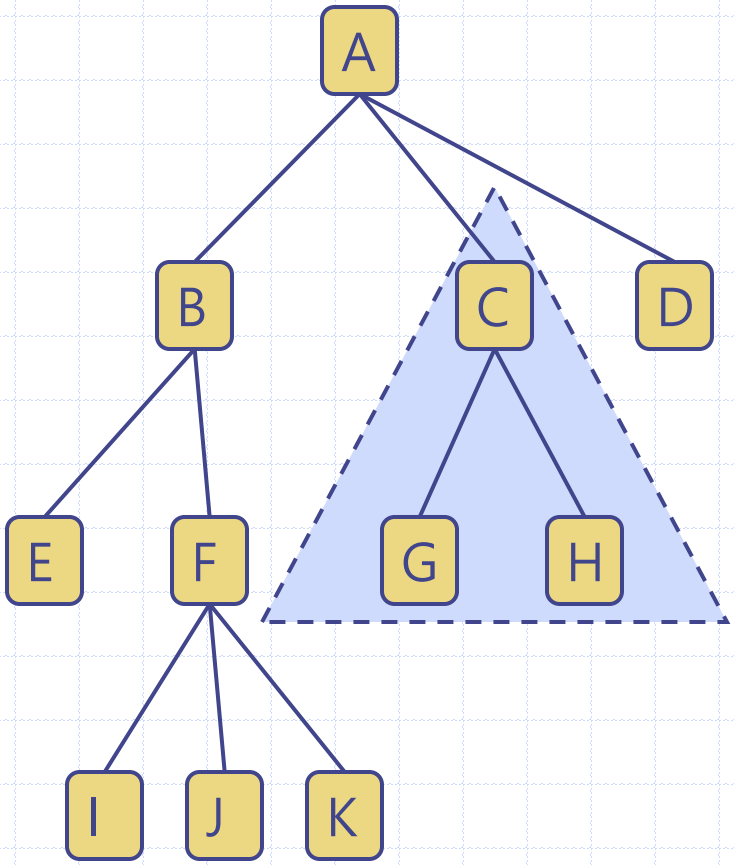


- 트리 ADT는 **계층적**으로 저장된 데이터원소들을 모델링
- 맨위의 원소를 제외하고, 각 트리 원소는 **부모**(parent) 원소와 0개 이상의 **자식**(children) 원소들을 가진다
- **전제:** 트리는 비어 있지 않다 – 알고리즘 단순화



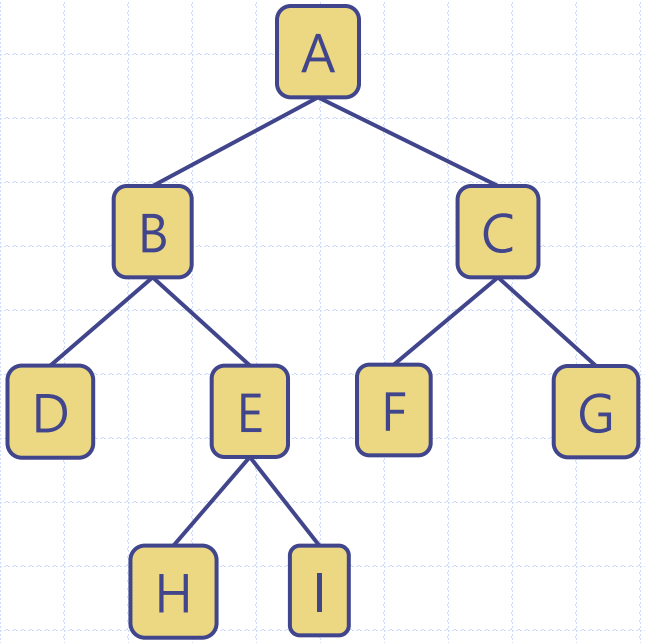
트리 용어

- **루트**(root): 부모가 없는 노드(A)
- **내부노드**(internal node): 적어도 한 개의 자식을 가진 노드(A, B, C, F)
- **외부노드**(external node), 또는 **리프**(leaf): 자식이 없는 노드(E, I, J, K, G, H, D)
- **형제**(siblings): 같은 부모를 가진 노드들(G, H)
- 노드의 **조상**(ancestor): 부모(parent), 조부모(grandparent), 증조부모(grand-grandparent), 등
- 노드의 **자손**(descendant): 자식(child), 손주(grandchild), 증손주(grand-grandchild), 등
- **부트리**(subtree): 노드와 그 노드의 자손들로 구성된 트리

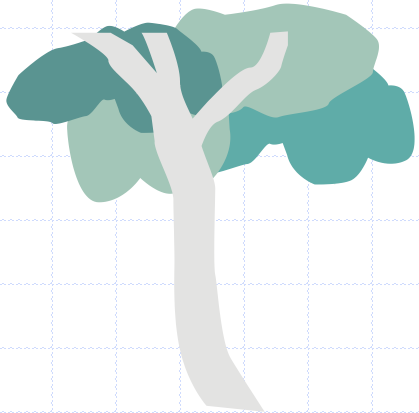


이진트리 ADT

- 이진트리 ADT는 순서트리를 모델링
- **전제: 적정이진트리**로 구현
  - 트리의 각 내부노드가 두 개의 자식을 가짐 – **왼쪽**(left) 및 **오른쪽**(right) 자식
  - 좌우 자식노드 가운데 하나가 비어 있는 경우라도 적정이진트리로 구현 가능
- **이진트리**의 재귀적 정의
  - 루트가 자식의 순서쌍을 가지며, 자식이 내부노드인 경우 이진트리다
- **전제:** 이진트리는 비어있지 않다
- **응용**
  - 수식 표현
  - 의사결정 과정
  - 검색

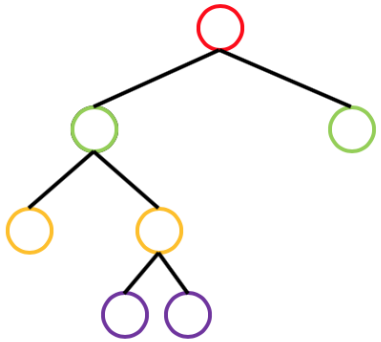


이진트리 ADT

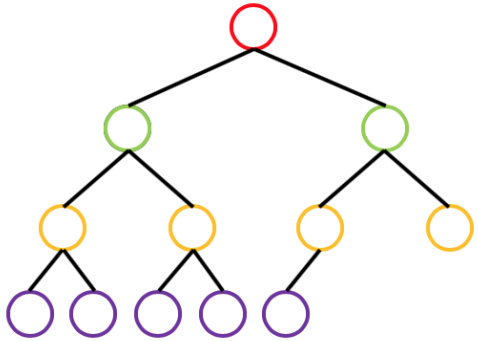


- 적정이진트리
- 완전이진트리
- 포화이진트리

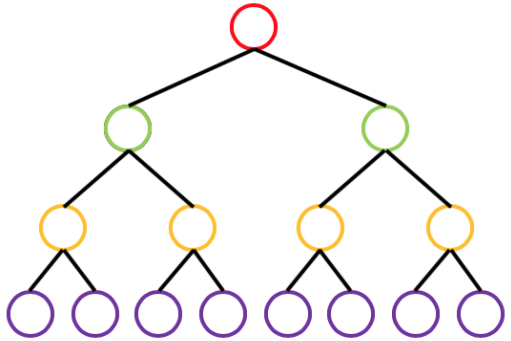
정 이진 트리 Full binary tree
적정 이진 트리 Proper binary tree



완전 이진 트리 Complete binary tree



포화 이진 트리 Perfect binary tree



https://sean-ma.tistory.com

이진트리 순회

- 이진트리의 순회는 트리 순회의 **특화**(specialization)
- **선위순회**(preorder traversal): 노드를 그의 왼쪽 및 오른쪽 부트리보다 **앞서** 방문
- **후위순회**(postorder traversal): 노드를 그의 왼쪽 및 오른쪽 부트리보다 **나중에** 방문

```
Alg binaryPreOrder(v)
1. visit(v)
2. if (isInternal(v))
        binaryPreOrder(leftChild(v))
        binaryPreOrder(rightChild(v))

Alg binaryPostOrder(v)
1. if (isInternal(v))
        binaryPostOrder(leftChild(v))
        binaryPostOrder(rightChild(v))
2. visit(v)
```

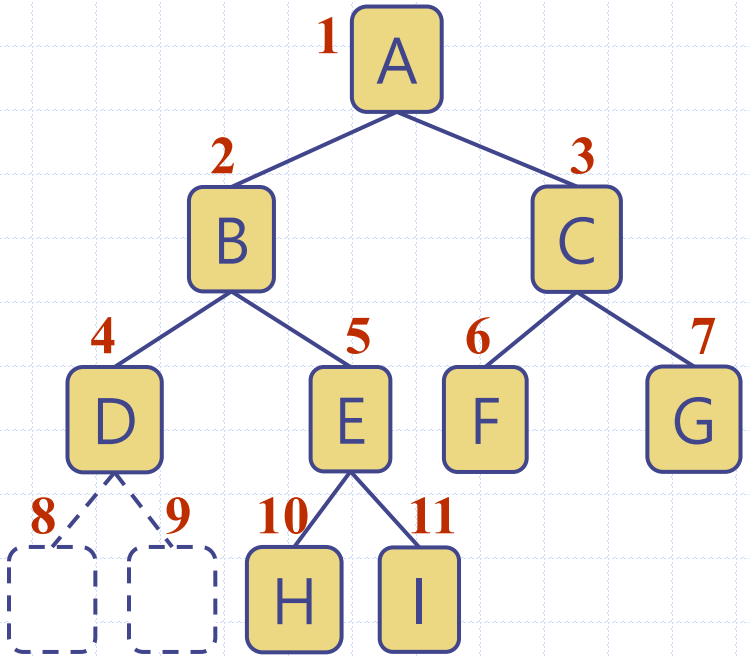
중위순회

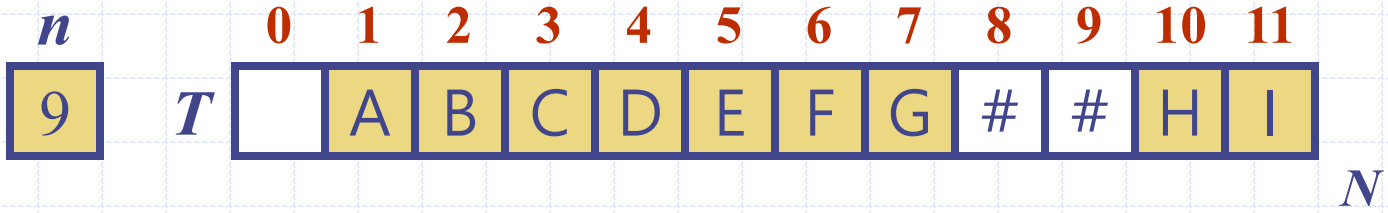
- **중위순회**(inorder traversal): 노드를 그의 왼쪽 부트리보다는 **나중에**, 오른쪽 부트리보다는 **앞서** 방문
- **실행시간:** $O(n)$ – 단, n은 이진트리 내 총 노드 수
- **응용**
  - 이진트리 그리기
  - 수식 인쇄

```
Alg inOrder(v)
1. if (isInternal(v))
       inOrder(leftChild(v))
2. visit(v)
3. if (isInternal(v))
       inOrder(rightChild(v))
```

배열에 기초한 이진트리

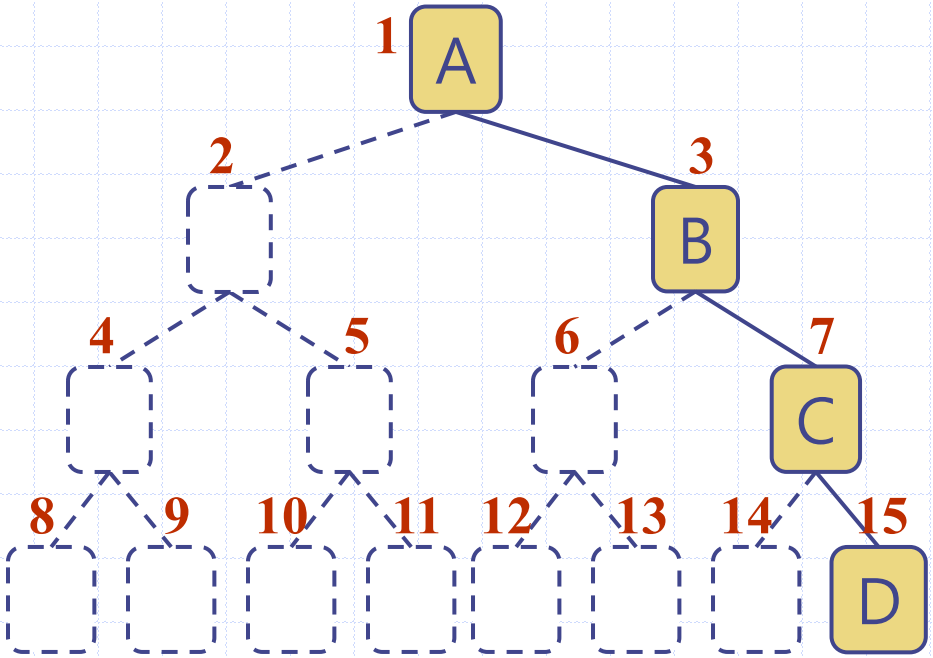
- **1D 배열**을 이용하여 이진트리 표현 가능
- 랭크 i의 노드에 대해:
  - **왼쪽 자식**의 위치는 순위 $2i$
  - **오른쪽 자식**의 위치는 순위 $2i + 1$
  - **부모**의 위치는 순위 $\lfloor i/2 \rfloor$
- 노드 간의 링크 저장 불필요
- 순위 0 셀은 미사용
- 비어 있는 셀은 특별값을 저장
  - 널마커(**예:** '#'), 또는
  - 널포인터(포인터배열인 경우)

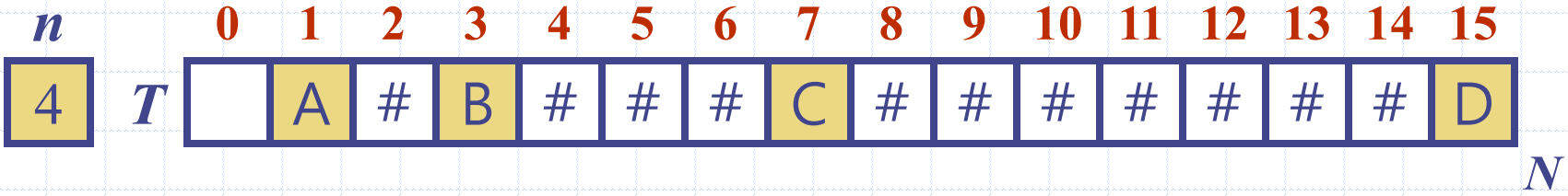




최선과 최악의 경우

- ***MAX***를 노드 순위 중 최대값이라 하면, 배열크기 $N = MAX$
- 최선의 경우, $N = n$ (**완전이진트리**, complete binary tree)
- **최악**의 경우, $N = 2^n - 1$ (단, 부적정이진트리 경우임)





연결이진트리

- 노드 저장내용
  - 원소
  - 부모노드 (필요 시 사용)
  - 왼쪽 자식노드
  - 오른쪽 자식노드