Datastructures

Mentor

황재상

AIgorithm, System, iOS

Github: jxx-sx

(현) 건국대학교 알고리즘 동아리 AIKon 강의자(그래프, SCC), 멘토

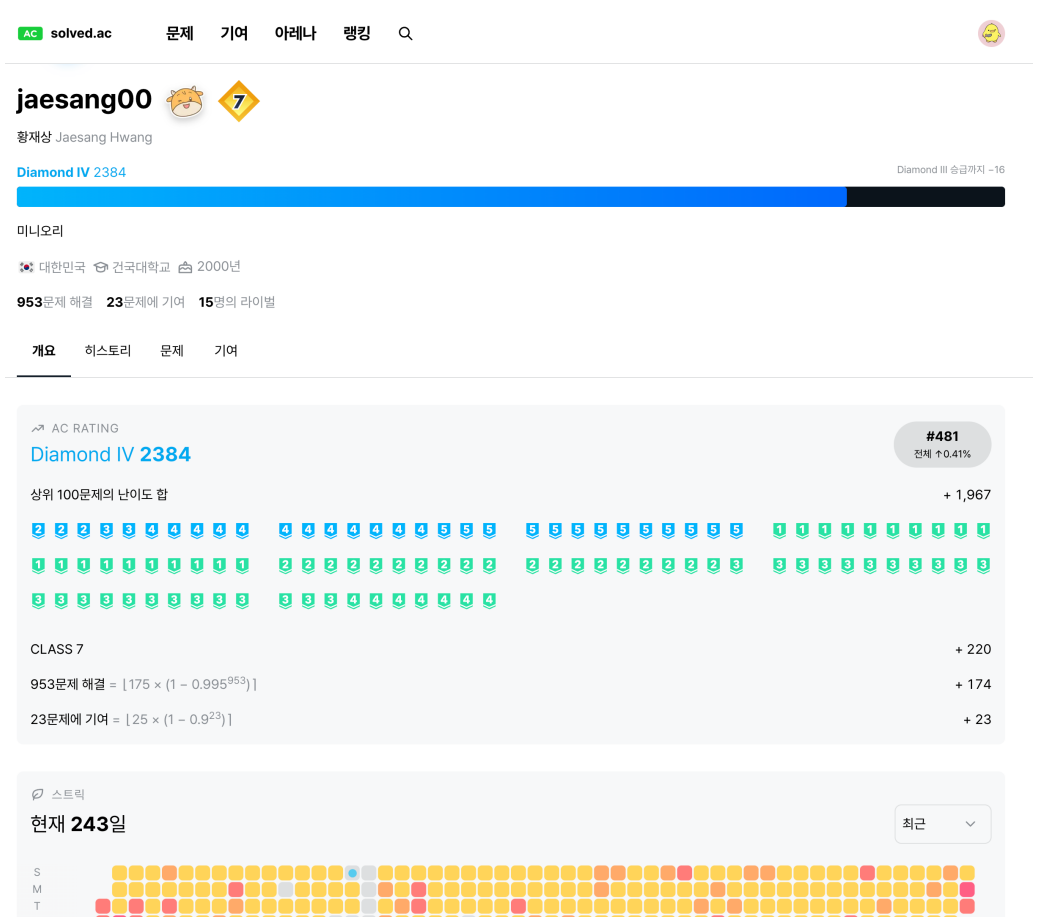
(전) 건국대학교 IT 동아리 KUIT 1st iOS 강의자

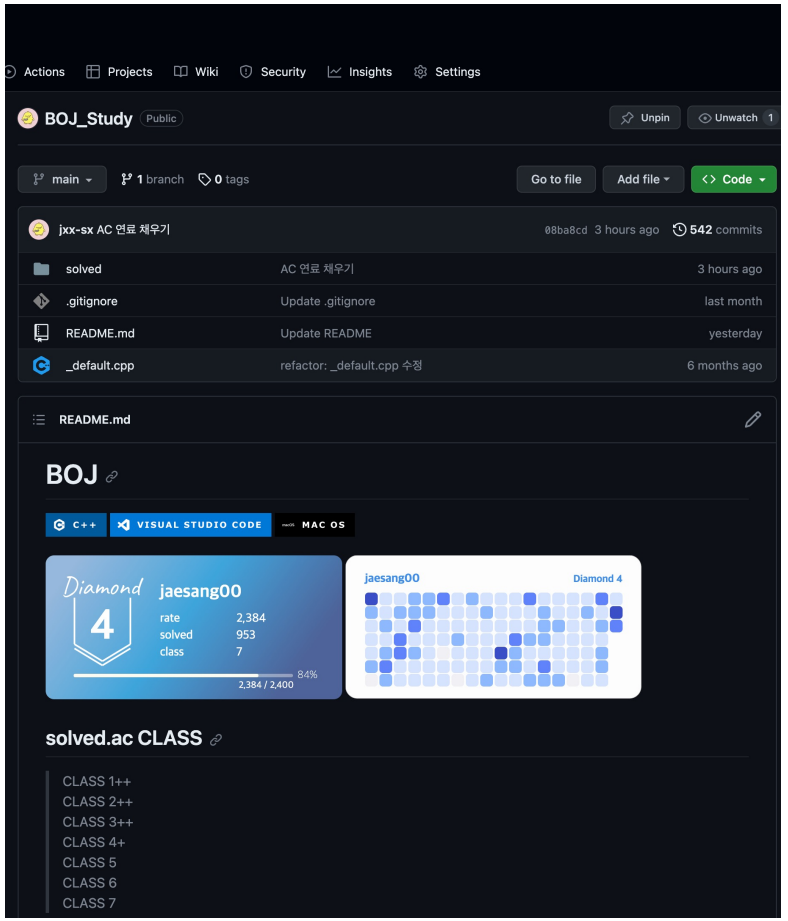
삼성전자 상시 SW 역량테스트 B형

KU 건국대학교 KONKUK UNIV.

Data-**C**entric **C**omputing **L**aboratory

Mentor


solved.ac
문제
기여
아레나
랭킹
jaesang00
7
황재상 Jaesang Hwang
Diamond IV 2384
Diamond III 승급까지 -16
미니오리
대한민국 건국대학교 2000년
953문제 해결 23문제에 기여 15명의 라이벌
개요
히스토리
문제
기여
AC RATING
Diamond IV 2384
#481
전체 ↑0.41%
상위 100문제의 난이도 합
+ 1,967
CLASS 7
+ 220
953문제 해결 = ⌊175 × (1 − 0.995^953)⌋
+ 174
23문제에 기여 = ⌊25 × (1 − 0.9^23)⌋
+ 23
스트릭
현재 243일
최근
S
M
T


Actions
Projects
Wiki
Security
Insights
Settings
BOJ_Study
Public
Unpin
Unwatch 1
main
1 branch
0 tags
Go to file
Add file
Code
jxx-sx AC 연료 채우기
08ba8cd 3 hours ago
542 commits
solved
AC 연료 채우기
3 hours ago
.gitignore
Update .gitignore
last month
README.md
Update README
yesterday
_default.cpp
refactor: _default.cpp 수정
6 months ago
README.md
BOJ
C++
VISUAL STUDIO CODE
MAC OS
Diamond
4
jaesang00
rate 2,384
solved 953
class 7
2,384 / 2,400
8.4%
jaesang00
Diamond 4
solved.ac CLASS
CLASS 1++
CLASS 2++
CLASS 3++
CLASS 4+
CLASS 5
CLASS 6
CLASS 7

자료 구조

- 컴퓨터에서 데이터를 효율적으로 조회/수정할 수 있도록 저장하는 것을 다루는 분야
- 상황에 따라 적절한 자료구조를 채택해 사용
- 크게 선형 자료구조와 비선형 자료구조로 구분

- 선형 자료구조: 스택, 큐, 덱, ...
- 비선형 자료구조: 그래프, 트리, 힙, ...

자료 구조

- 상황에 따라 적절한 자료구조를 채택해 사용
- 조회가 빠른 구조: 배열
- 특정한 목적을 가진 구조: 스택, 큐
- 특정한 형태를 표현한 구조: 그래프, 트리
- 미리 정렬된 구조: B+ Tree, Merge Sort Tree

Array
Vector
Linked List

배열

사람의 입장

- 여러 개의 변수를 한번에 만드는 것

컴퓨터의 입장

- 메모리에서 연속된 공간을 할당하는 것

Memory

- 컴퓨터는 0과 1로 이루어져 있다
- 0과 1을 저장하는데, 0 또는 1 데이터 하나, 또는 저장하는 공간의 단위: Bit
- 1 bit: 0, 1
- 2 bit: 00, 01, 10, 11
- 3 bit: 000, 001, 010, 011, 100, 101, 110, 111
- 4 bit: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, ...

Memory

- 8개의 비트를 묶어서 1 Byte라고 부른다

  1000m가 1Km인 것처럼 하나의 단위

- 컴퓨터는 Byte 단위를 주로 사용한다

Memory

- 컴퓨터의 메모리(RAM)은 1Byte를 저장할 수 있는 셀들로 구성이 되어있다

  8GB RAM: 8GB는 8,000,000,000Byte이므로 8,000,000,000만큼의 셀이 존재

- 각각의 셀들은 주소가 존재한다
- 시작은 0, 내 다음 주소는 +1, 0부터 시작해 메모리 크기만큼 주소가 존재

변수

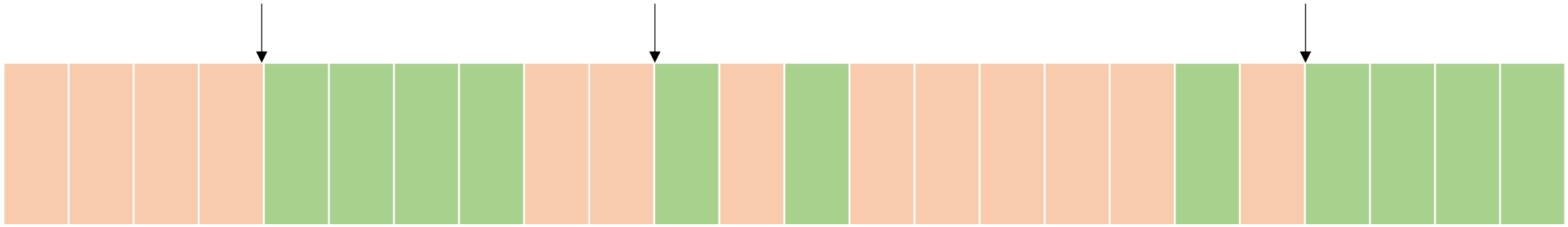
- 변수를 만드는 것은 메모리에 변수의 크기만큼 메모리를 할당하는 것

  그리고 시작 주소를 기억하고 있는다

- 변수 하나의 크기는 자료형에 따라 다르다
- Character(1 바이트), Integer(4 바이트)

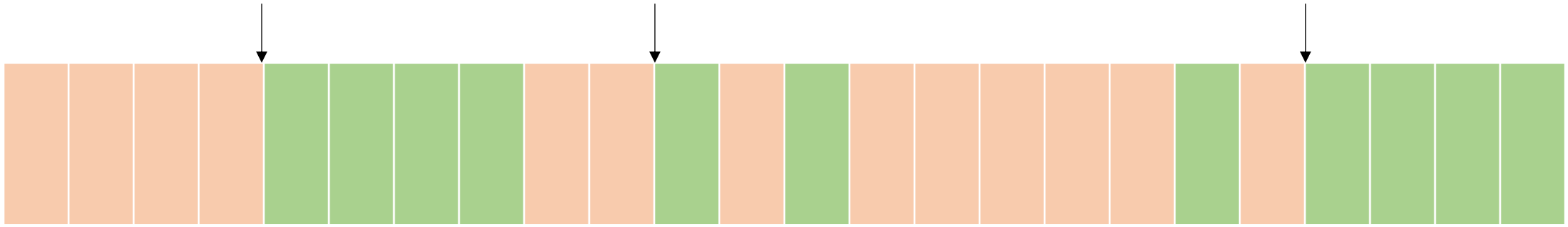
변수

- int a;
- char b;
- int c;



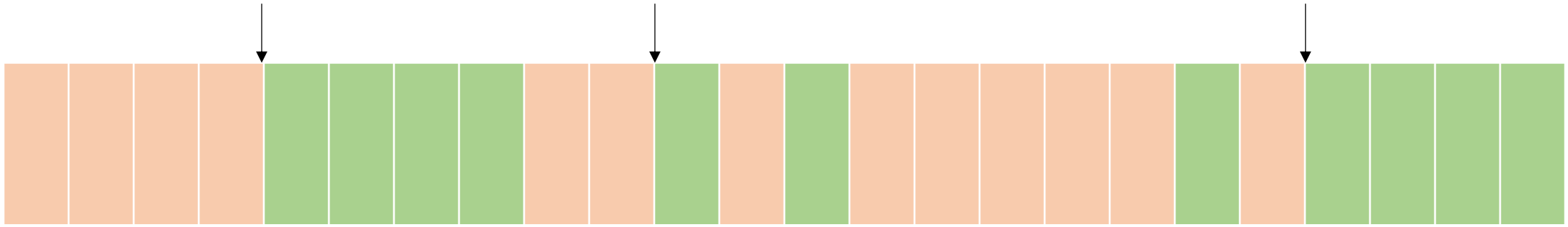
변수

- int a;　　주소: 4
- char b;　　주소: 10
- int c;　　주소: 20



변수

- 자료형을 통해서 공간을 얼마나 차지하고 있을 지, 어떻게 값을 사용할지 알 수 있다
- int a; 공간: [4, 8)
- char b; 공간: [10, 11)
- int c; 공간: [20, 24)



배열

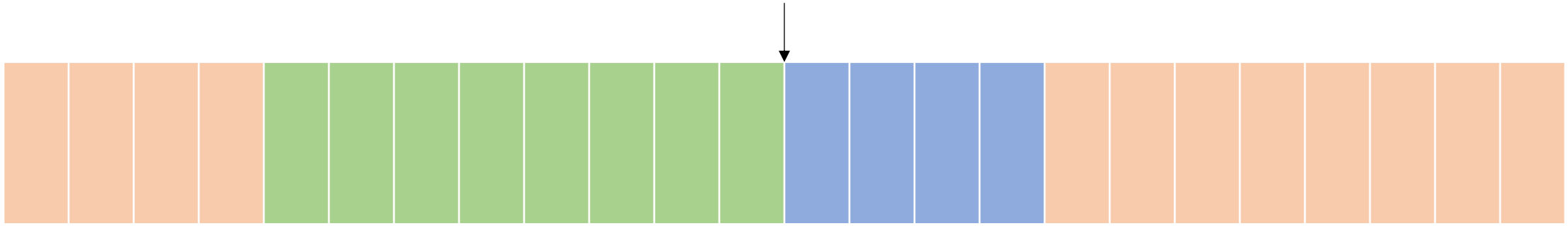
- 배열을 만드는 것은 배열의 크기만큼 메모리에 연속적으로 공간을 할당하는 것
- int 3칸짜리 배열은 int 변수 3개 만큼의 공간을 할당하는 것

- int arr[3]; 주소: 4 공간: [4, 16)

인덱스

- 배열에서 인덱스는 시작점으로부터 일정 공간만큼 뒤에 있는 위치를 의미

- arr[2]: 배열의 시작점으로부터 자료형의 크기로 2칸만큼 뒤에 있는 것을 의미
- 시작점 4, 자료형의 크기 4, 2칸 뒤 이므로 4 + (4 * 2)의 데이터를 의미



배열

- 주로 여러 개의 변수를 한 번에 만들 때 주로 사용
- 인덱스를 활용한 조회가 매우 빠름 -> 데이터의 수정 또한 매우 빠름

- 자료형 변수명[크기];
- int arr[3];

배열의 단점

- 길이가 고정되어 있다
- 길이를 수정하고 싶은 경우, 크기가 다른 배열을 다시 만들어야 한다
- 데이터의 경우 모든 데이터를 새로운 배열에 복사해야 한다

- 중간에 삽입하기가 어렵다
- 중간에 삽입하려는 경우, 데이터들을 옮겨야 한다

Vector

- 길이가 늘어나는 배열
- 배열에 넣어야 할 값의 개수를 알 수 없는 경우 주로 사용
- 배열의 길이를 알지 않고도 사용할 수 있기 때문에 편리성으로 많이 사용

- #include <vector>
- vector<자료형> 변수명;

std::vector

- vector<자료형> v; : 비어있는 벡터를 생성
- vector<자료형> v = vector<자료형>(): 비어있는 벡터를 생성(위와 동일)
- vector<자료형> v = vector<자료형>(size)

  : 크기가 size인 벡터를 생성(초기값)
- vector<자료형> v = vector<자료형>(size, value)

  : 크기가 size이며 값이 value인 벡터를 생성(초기값)

std::vector

- vector<자료형>.push_back(A): 벡터(배열)의 제일 뒤에 원소(A)를 추가
- vector<자료형>.pop_back(): 벡터(배열)의 제일 뒤의 원소를 제거
- vector<자료형>.size(): 벡터(배열)에 삽입되어 있는 원소의 개수를 알려줌
- vector<자료형>.empty(): 벡터(배열)에 비어있는지를 알려줌
- vector<자료형>.front(): 벡터(배열)의 제일 첫번째 원소(vector[0])를 알려줌
- vector<자료형>.back(): 벡터(배열)의 제일 마지막 원소를 알려줌

Vector

- 길이를 늘린다는 것은 새로운 변수나 배열을 더 만드는 것, 메모리를 사용하기 위해 공간을 예약하는 것(메모리 할당)
- 컴퓨터에서 메모리 할당을 하는 것은 매우 비싼 작업이다

- 메모리 할당을 작게 자주, 빈번하게 할당이 일어나는 것은 비효율적
- 반대로 한번에 많이 할당하는 것은 다 사용하지 않는다면 메모리 낭비

Memory Allocation

- 왜 자주 메모리 할당을 하는 것이 비효율적일까?
- 4000B를 할당한다 가정해보자(Int 변수(4 바이트) 1000개)
- 4B씩 1000번 할당하자
- 400B씩 10번 할당하자

- 같은 양인 둘은 동일하지 않을까?

Memory Allocation

- 메모리 할당은 프로그램(C언어)이 하지 않는다

- 프로그램(C언어)에서 운영체제에 얼마만큼의 메모리가 필요하다고 요청
- 운영체제는 요청을 받으면 프로그램을 잠깐 멈추고 운영체제가 일을 하기 시작
- 운영체제는 메모리 할당이 끝나면 프로그램을 다시 실행시킴

Memory Allocation

- 사람이 글을 쓰고 있다고 생각해보자
- 연필을 들고 글을 쓰다가 잠시 멈추고 생각한 이후에 다시 쓰는 것은 생각한 시간만큼 만 멈춰있다
- 지워야 하는 경우, 연필을 내려두고 지우개를 드는 시간이 필요
- 반대로 다시 쓰는 경우, 지우개를 내려두고 연필을 드는 시간이 필요
- 컴퓨터도 마찬가지다

Memory Allocation

- 양손에 지우개랑 연필을 들면 되지 않나요? -> 대학원의 인재

Memory Allocation

- 프로그램 -> 운영체제 -> 프로그램으로 바뀌는데 시간이 소모됨
- 많은 운영체제 호출은 프로그램을 느려지게 만든다
- 한번에 최대한 많이, 그리고 필요한 만큼만 할당하는 것이 중요

- ex) 게임 로딩 동안 미리 게임에서 사용할 메모리들을 할당하고 데이터를 불러오는 것
  게임 중에 많은 운영체제 호출을 피하기 위함

Vector

- 값이 담겨 있는 개수(Size)
- 실제 할당된 배열의 실제 크기(Capacity)

- 그래서 여러 사람들이 탐구해보니 2배씩 늘리는게 효율적이더라
- Capacity보다 많은 값을 담으려는 경우, Capacity가 2배로 증가

Vector

- 처음 벡터를 선언한 경우 Size, Capacity 둘 다 0
- 처음 빈 벡터에서 하나를 담을 때는 Capacity +1
- 그 이후에 Capacity보다 많은 값을 담으려는 경우, Capacity가 2배로 증가

Vector

- vector<int> v;
- v.push_back(1) : 1 삽입
- v.push_back(2) : 2 삽입
- v.push_back(3) : 3 삽입
- v.push_back(4) : 4 삽입
- v.push_back(5) : 5 삽입
- v.push_back(6) : 6 삽입

Vector

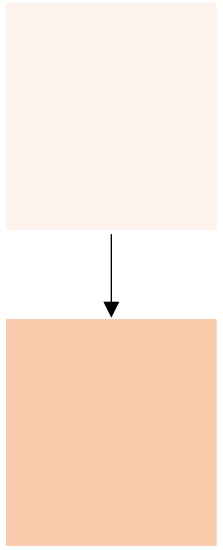
- vector<int> v; (Size: 0, Capacity: 0)
- v.push_back(1) : 1 삽입 (Size: 1, Capacity: 1) <- Capacity $+1$
- v.push_back(2) : 2 삽입 (Size: 2, Capacity: 2) <- Capacity $\times 2$
- v.push_back(3) : 3 삽입 (Size: 3, Capacity: 4) <- Capacity $\times 2$
- v.push_back(4) : 4 삽입 (Size: 4, Capacity: 4)
- v.push_back(5) : 5 삽입 (Size: 5, Capacity: 8) <- Capacity $\times 2$
- v.push_back(6) : 6 삽입 (Size: 6, Capacity: 8)

Vector

- vector<int> v; (Size: 0, Capacity: 0)

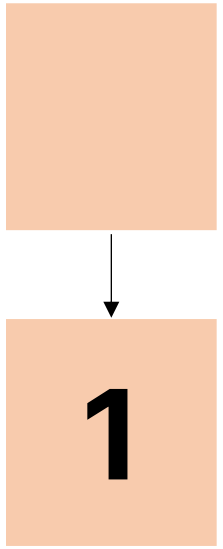
Vector

- v.push_back(1) : 1 삽입 (Size: 1, Capacity: 1) <- Capacity +1



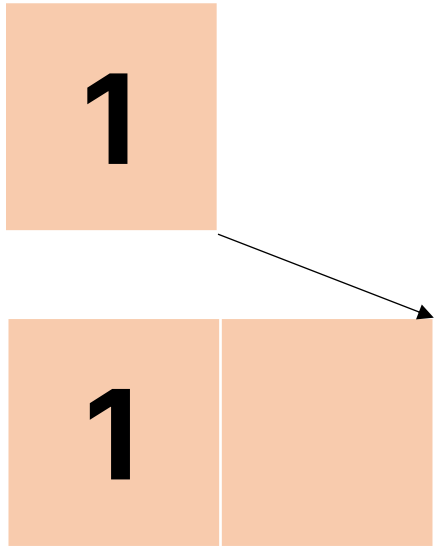
Vector

- v.push_back(1) : 1 삽입 (Size: 1, Capacity: 1) <- Capacity +1



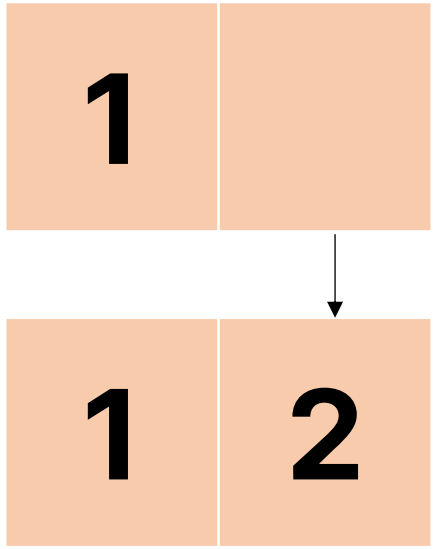
Vector

- v.push_back(2) : 2 삽입 (Size: 2, Capacity: 2) <- Capacity ×2



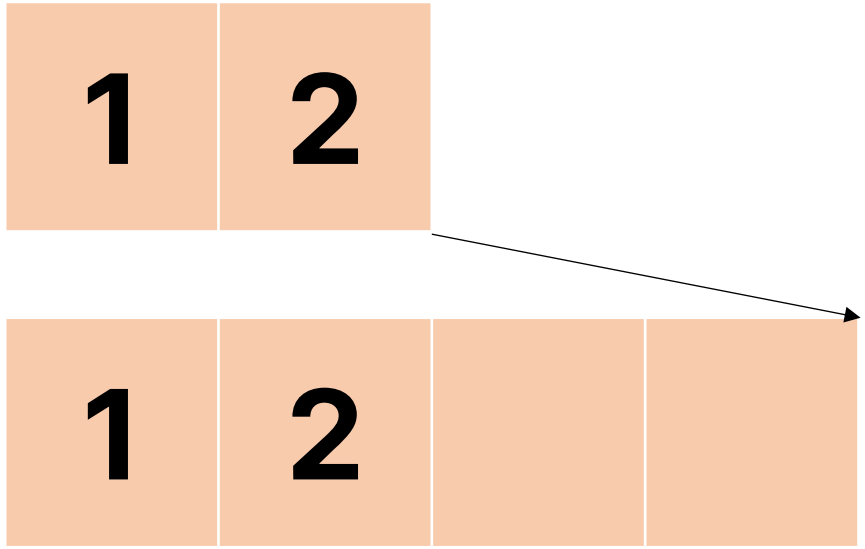
Vector

- v.push_back(2) : 2 삽입 (Size: 2, Capacity: 2) <- Capacity ×2



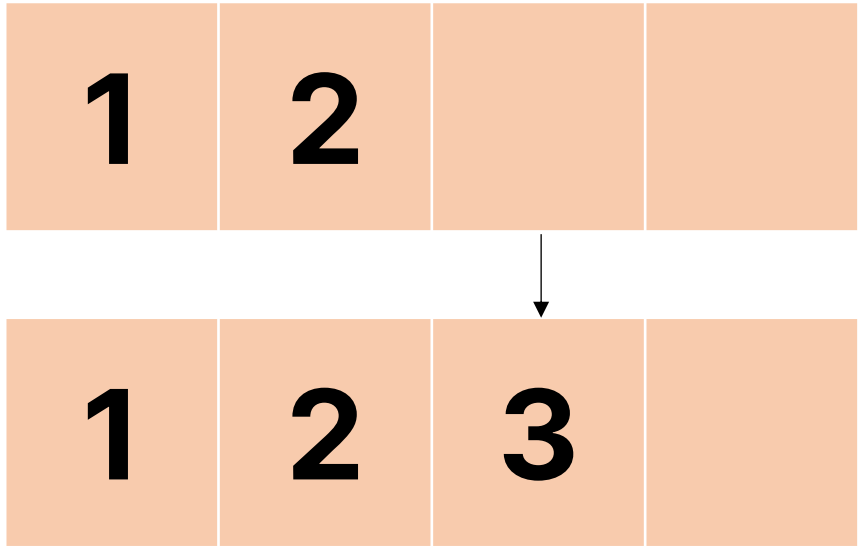
Vector

- v.push_back(3) : 3 삽입 (Size: 3, Capacity: 4) <- Capacity ×2



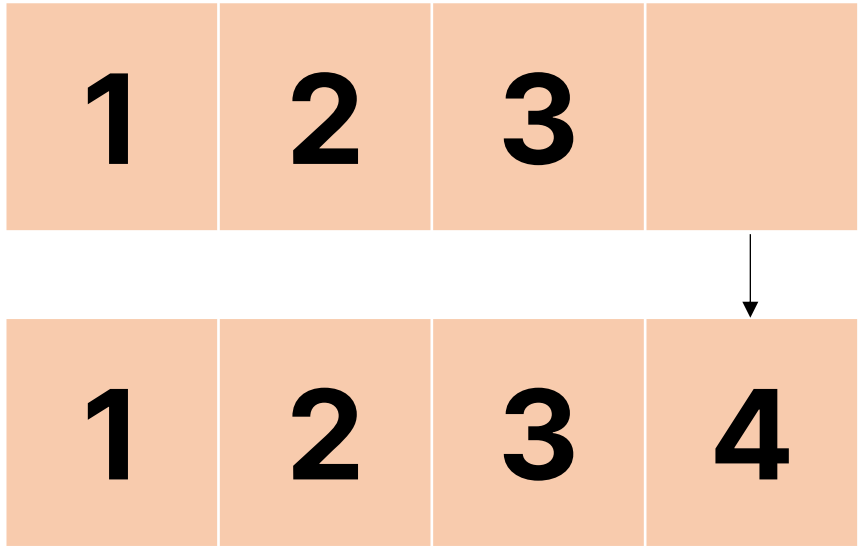
Vector

- v.push_back(3) : 3 삽입 (Size: 3, Capacity: 4) <- Capacity ×2



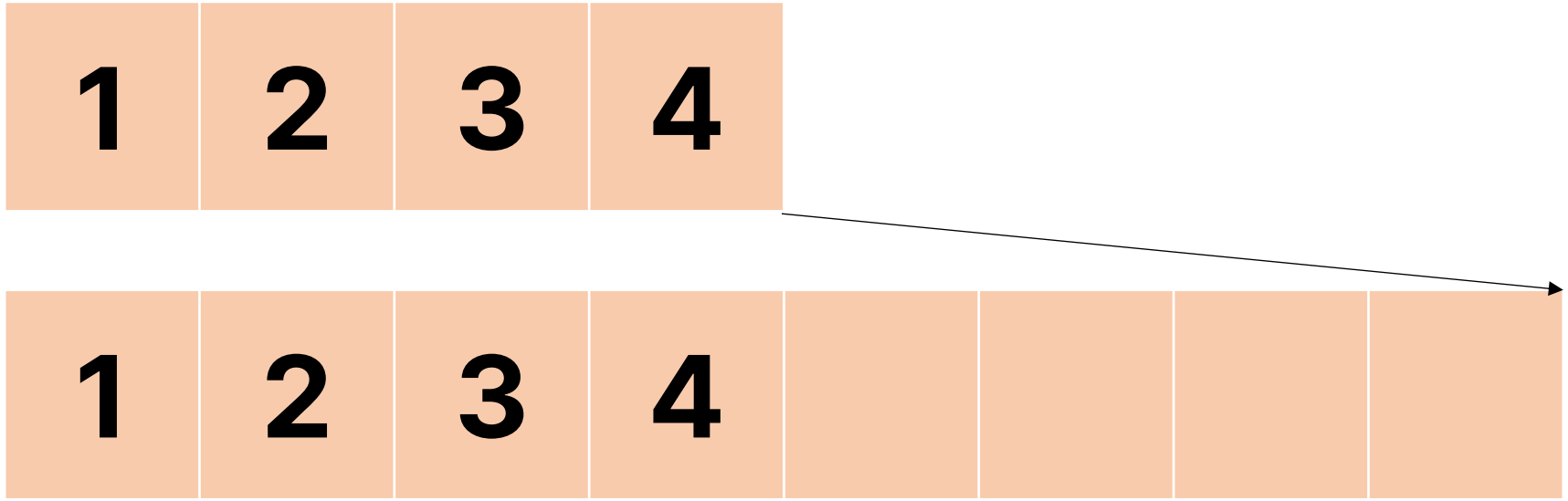
Vector

- v.push_back(4) : 4 삽입 (Size: 4, Capacity: 4)



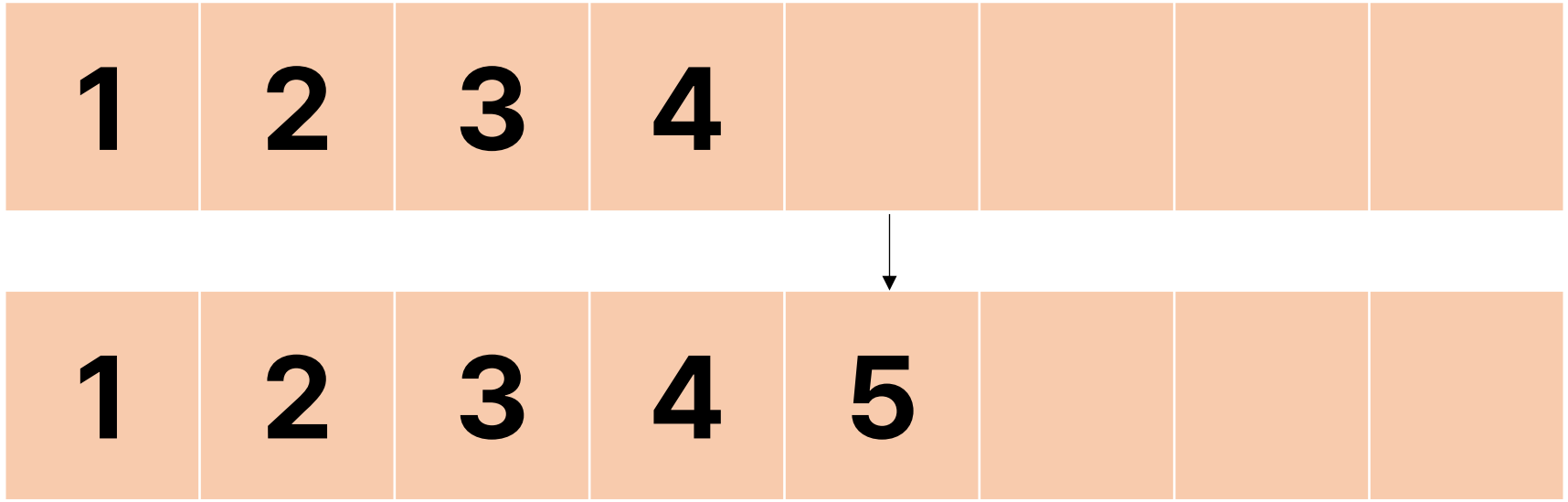
Vector

- v.push_back(5) : 5 삽입 (Size: 5, Capacity: 8) <- Capacity ×2



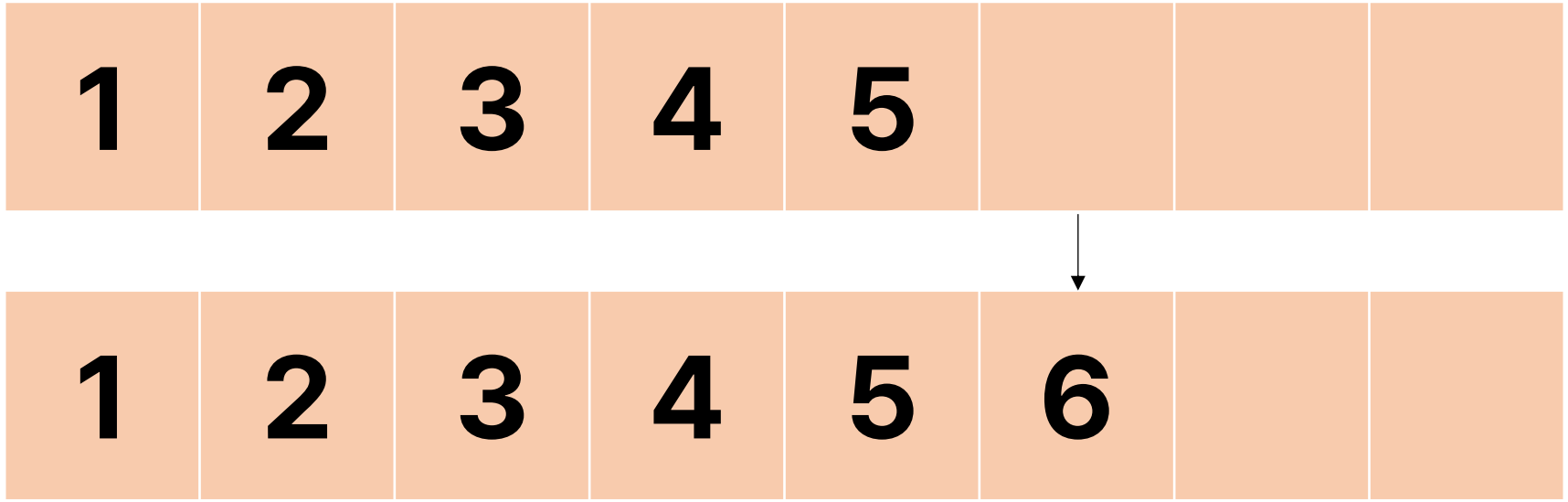
Vector

- v.push_back(5) : 5 삽입 (Size: 5, Capacity: 8) <- Capacity ×2



Vector

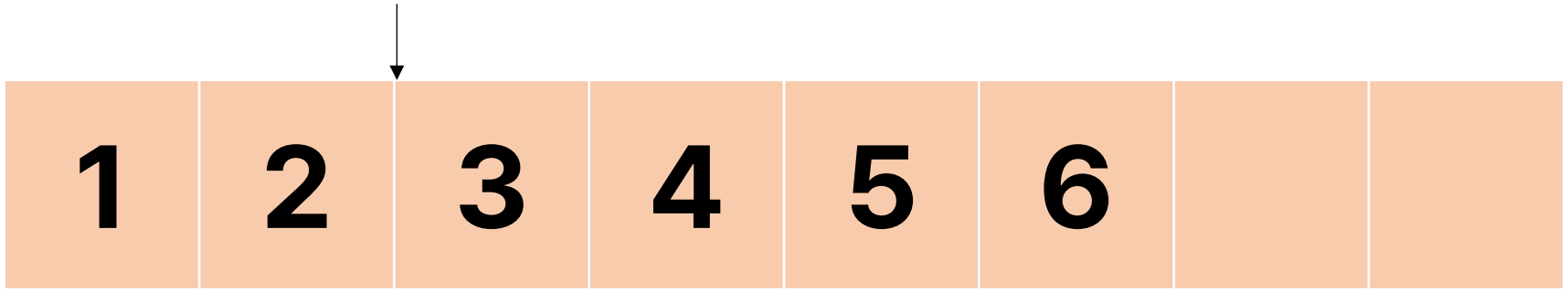
- v.push_back(6) : 6 삽입 (Size: 6, Capacity: 8)



배열과 벡터의 문제점

- 원소를 중간에 삽입하고 싶다
- 2와 3사이에 7을 넣고 싶다

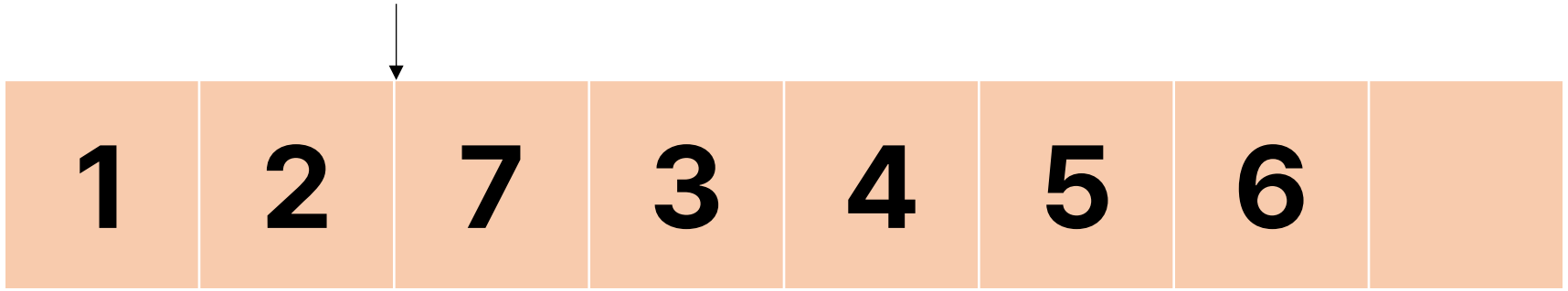
- 어떻게 해야 할까?



배열과 벡터의 문제점

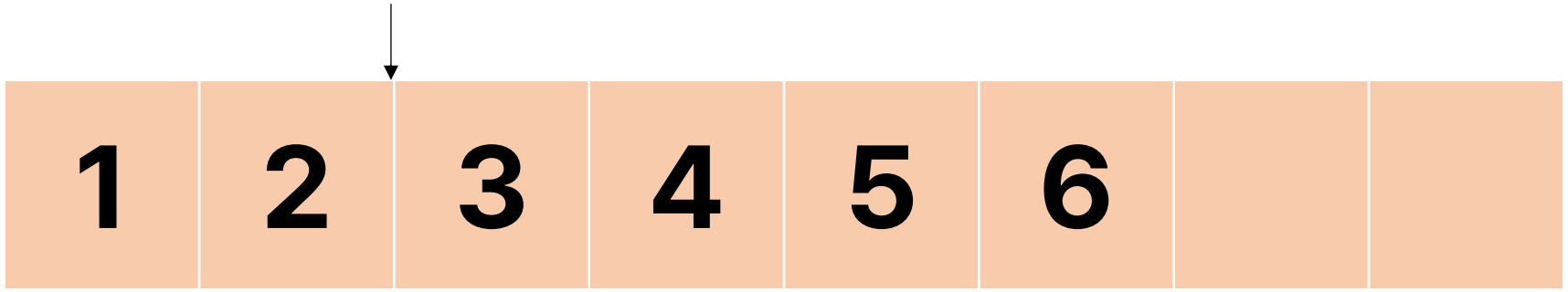
- 3부터 6을 뒤로 한 칸씩 이동시킨다
- 넣으려는 위치에 7을 넣는다

- 실제 코드로는?

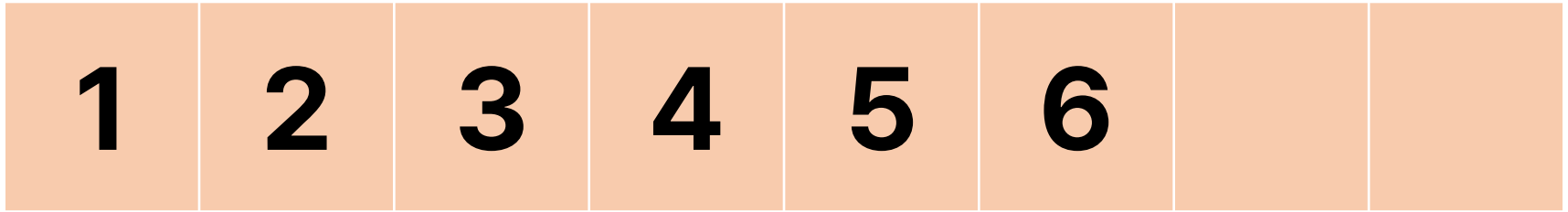


배열과 벡터의 문제점

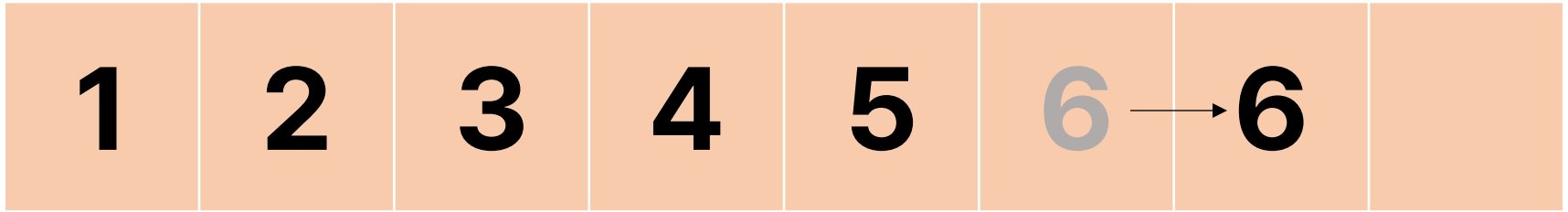
- 우선 배열을 완전히 복사하는 것은 시간과 공간이 많이 소모되므로 뒤에서부터 한 칸씩 옮긴다
- 만일 3부터 옮기는 경우 3이 4를 덮어쓰기하여 4를 찾을 수 없게 된다



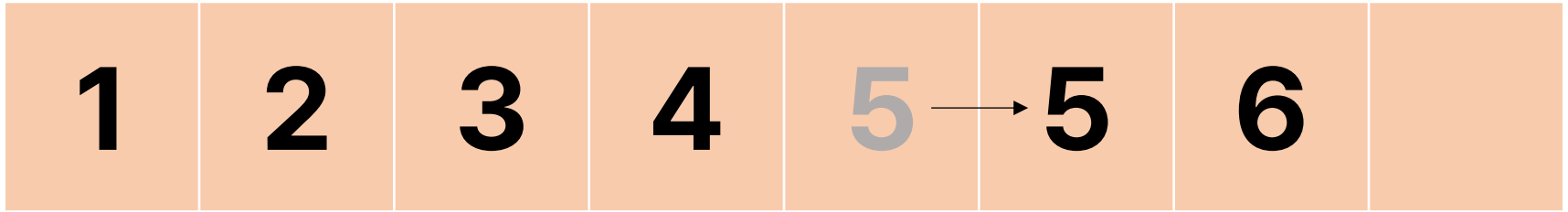
배열과 벡터의 문제점



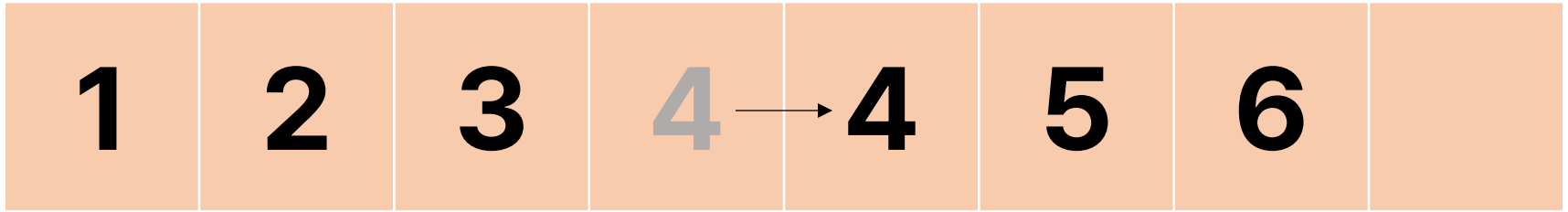
배열과 벡터의 문제점



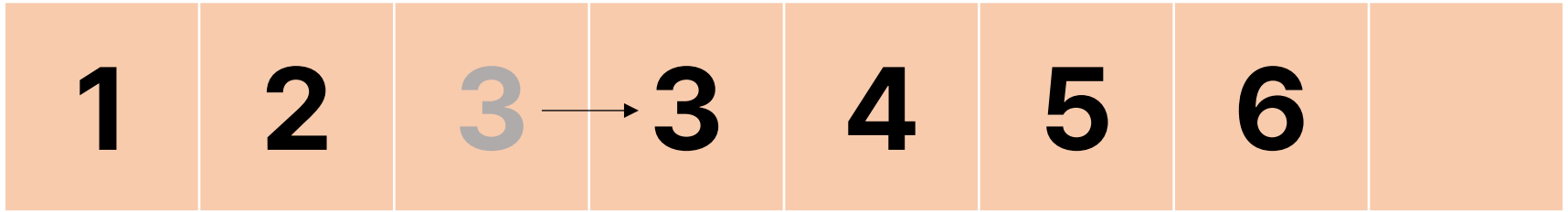
배열과 벡터의 문제점



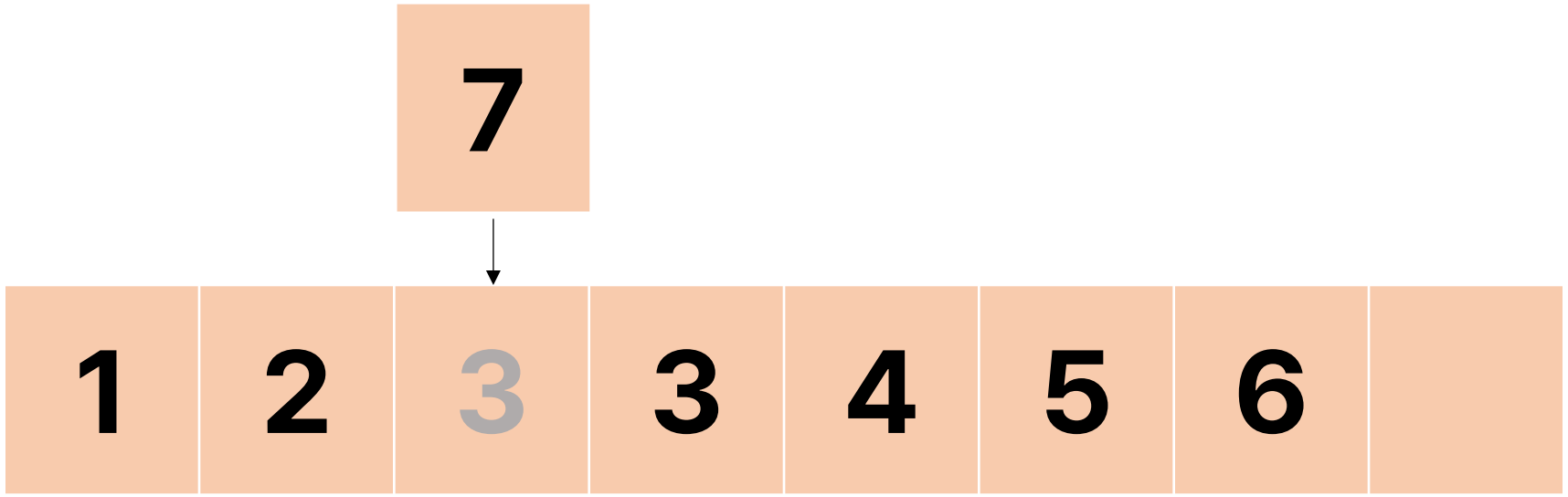
배열과 벡터의 문제점



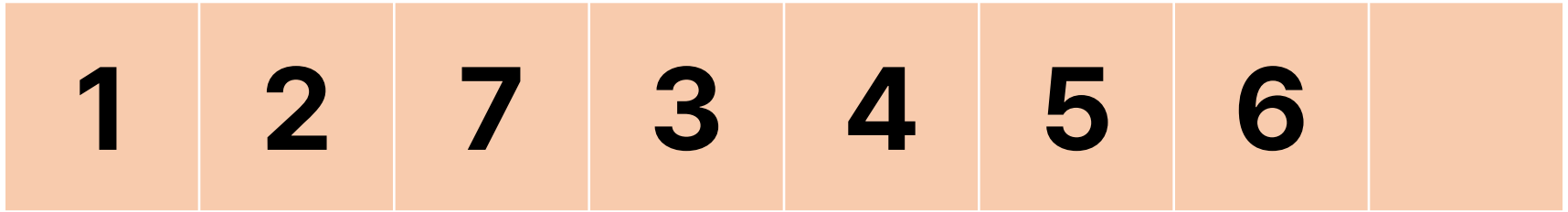
배열과 벡터의 문제점



배열과 벡터의 문제점



배열과 벡터의 문제점



배열과 벡터의 문제점

- 기존의 배열은 중간에 삽입과 삭제가 일어나는 경우, 삽입하려는 위치 이후에 있는 모든 원소를 이동시켜야 한다
  (공간이 여유롭지 않은 경우 삽입이 일어날 수 없으므로 공간이 여유롭다고 가정한다)

- 만일 모든 삽입이 배열의 제일 앞에 삽입되어 한다면, 모든 원소를 한 칸 뒤로 이동시켜야 되므로 매우 비효율적이다

Fast Insert?

- 커서를 생각해보자
- 글 중간에 한 곳을 찍어 글자를 쓰는 경우 모든 글자를 한 칸 씩 이동시키는 것은 매우 오래 걸린다(뒤에 있는 글자가 몇 만 개라면?)
- 커서 왼쪽에 있는 글자(a)와 오른쪽에 있는 글자(c)는 서로 연결되어 있을 것
  “a 다음에 c야”
- b를 그 사이에 넣는다면 “a 다음에 b”, “b 다음에 c”야 라고 추가한다면 매우 빠를 것

Linked List

- 배열은 메모리에서 **연속된 공간**에 이어 붙여 만들지만, 메모리 공간이 부족하다면 할 수 없다
- 작게 분할해서 여러 개를 만들고 앞과 뒤를 연결해주자

- 데이터가 들어가거나 삭제될 때는 앞과 뒤를 연결해둔 **연결**만 수정을 하면 다른 데이터들을 이동을 하지 않고 새로운 데이터를 중간에 넣거나 삭제할 수 있다

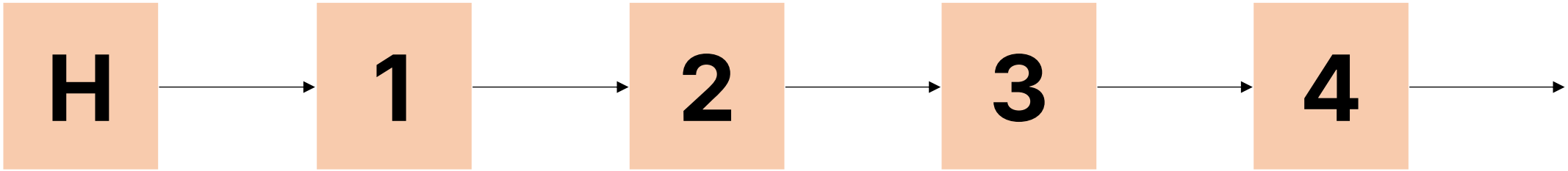
Linked List

- 연결 리스트
- 값을 가지고 있고(변수값), 앞 뒤가 서로 연결된 변수(노드)들로 구성된 자료구조
- 삽입, 삭제와 같은 수정이 빠르다, 몇 번째 변수로 이동해야 하는 경우 느리다

- Single Linked List
- Double Linked List

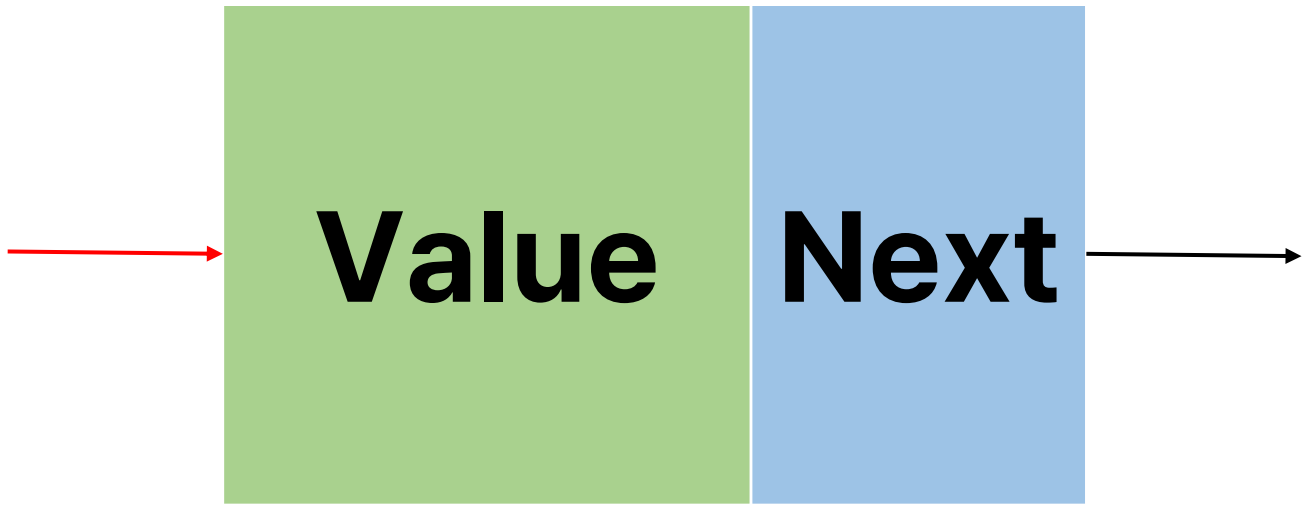
Single Linked List

- 앞에서 뒤로만 연결이 되어있는 형태
- 연결이 한 개(Single)인 형태이다
- 앞에서 뒤로는 탐색이 가능하지만 반대로는 불가능하다
- 뒤로 돌아갈 필요가 없는 경우 사용



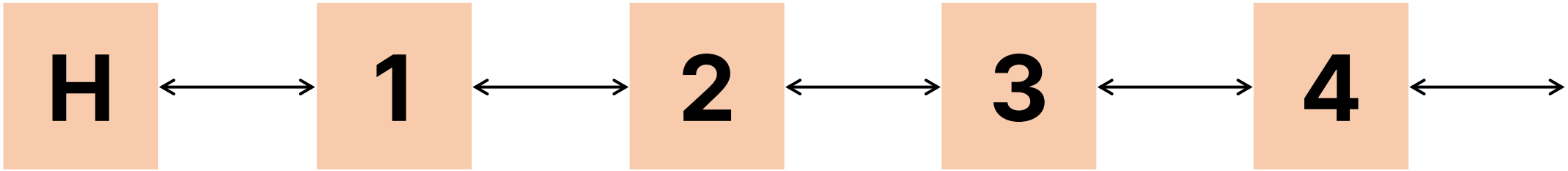
Single Linked List Node

- 변수 값(Value
- 다음 노드의 위치(Next)



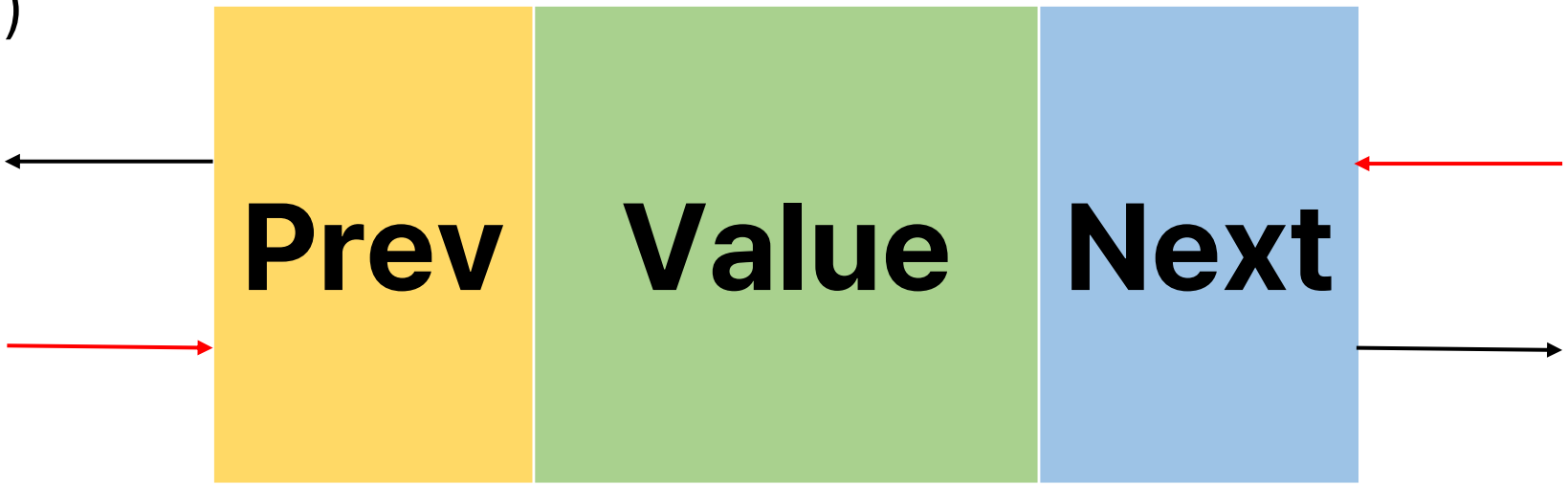
Double Linked List

- 앞에서 뒤로, 뒤에서 앞으로 양방향으로 연결된 형태
- 연결이 두 개(Double)인 형태이다
- 현재 내 앞과 뒤에 어떤 노드들이 있는지 알 수 있어 양 방향으로 이동이 가능



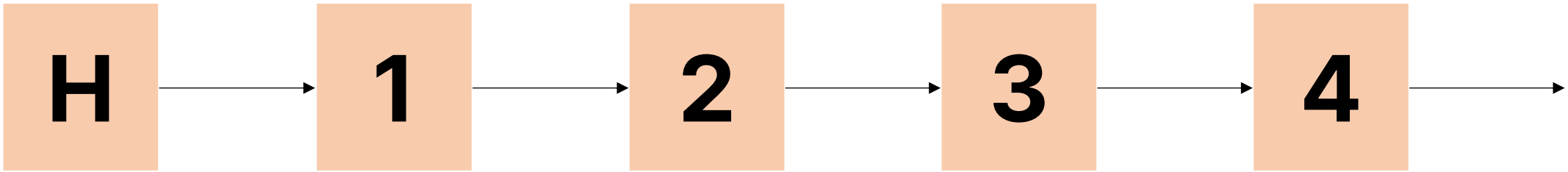
Double Linked List Node

- 변수 값(Value
- 다음 노드의 위치(Next)
- 이전 노드의 위치(Prev)



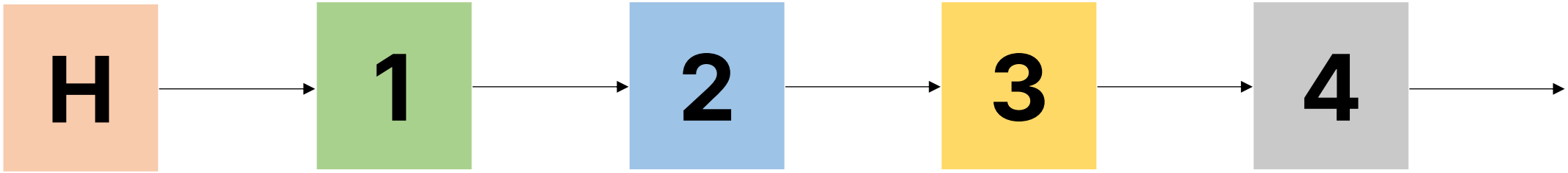
Linked List Find

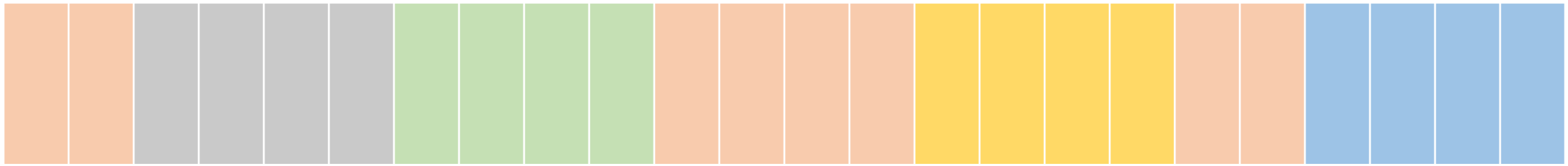
- H: 헤드, 시작점을 의미
- H 뒤에 아무것도 없으면 리스트가 비어 있는 것을 의미



Linked List

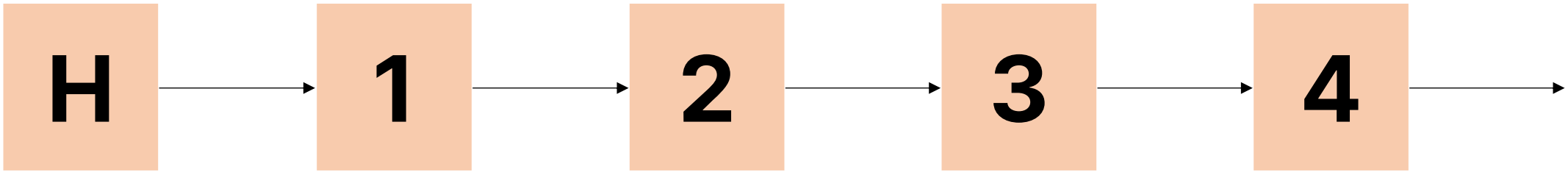
- 연결 리스트는 칸을 배열로 한.번에 만들지 않고 칸 하나하나 만들기 때문에 메모리에서 연속적이지 않다 -> 인덱스를 사용할 수 없다





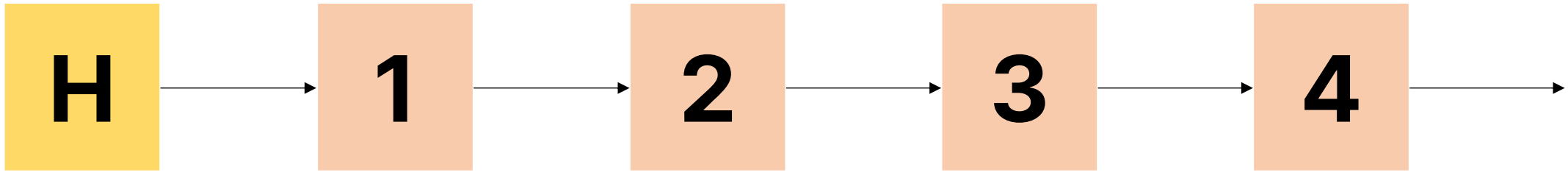
Linked List Find

- 앞에서부터 하나씩 살펴보며 값이 있는지 확인
- 3을 찾아보자



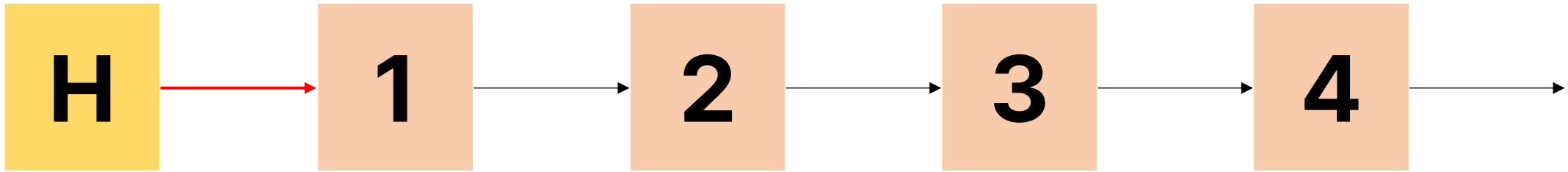
Linked List Find

- 헤드에서 시작



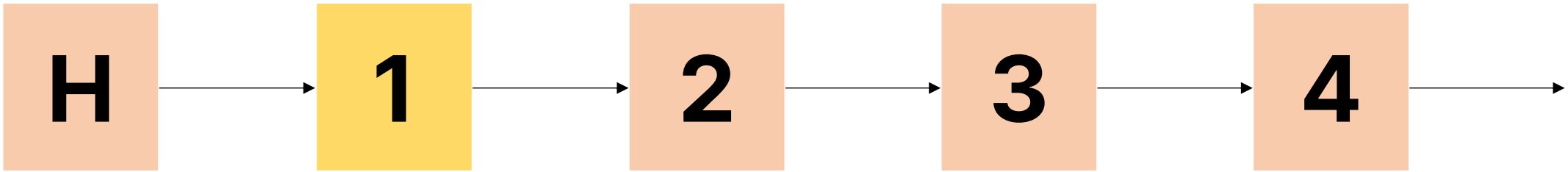
Linked List Find

- 다음 노드가 있는지 확인



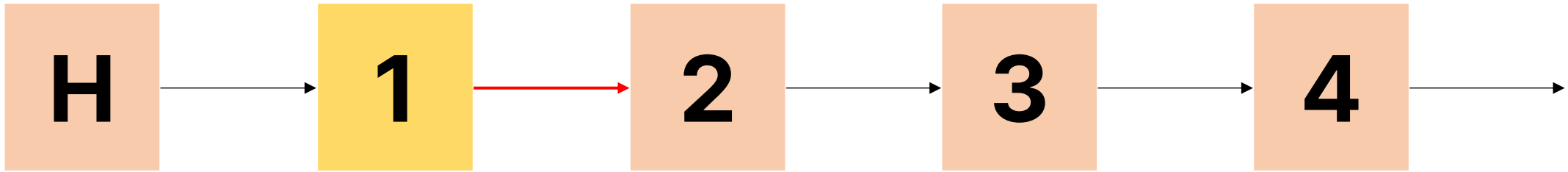
Linked List Find

- 다음 노드가 존재하므로 이동
- 이동한 이후에 찾으려는 값과 동일한지 확인
- 찾으려는 값은 3이지만 1이 담겨 있으므로 계속 진행



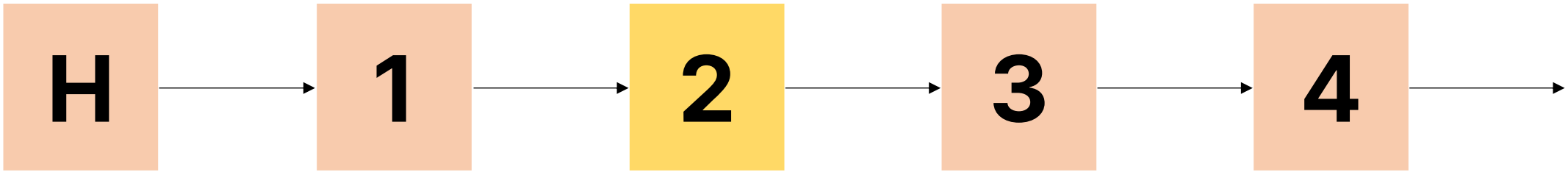
Linked List Find

- 다음 노드가 있는지 확인



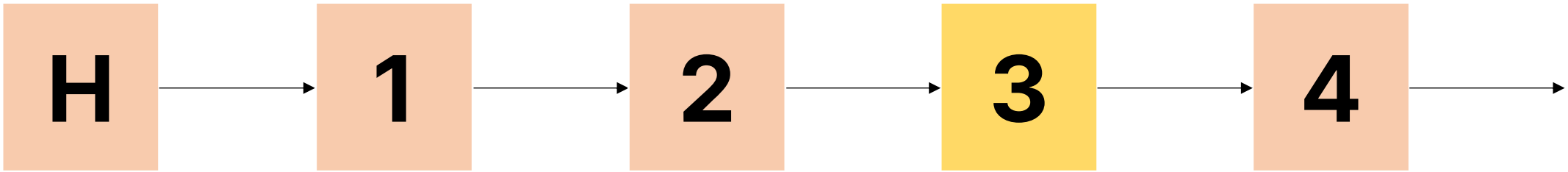
Linked List Find

- 다음 노드가 존재하므로 이동
- 이동한 이후에 찾으려는 값과 동일한지 확인
- 찾으려는 값은 3이지만 2이 담겨 있으므로 계속 진행



Linked List Find

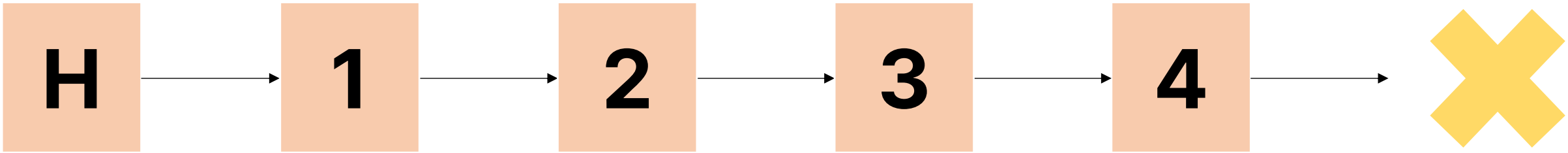
- 찾으려는 값이 있는 경우 탐색에 성공(값을 찾음)(값이 존재)



Linked List Find

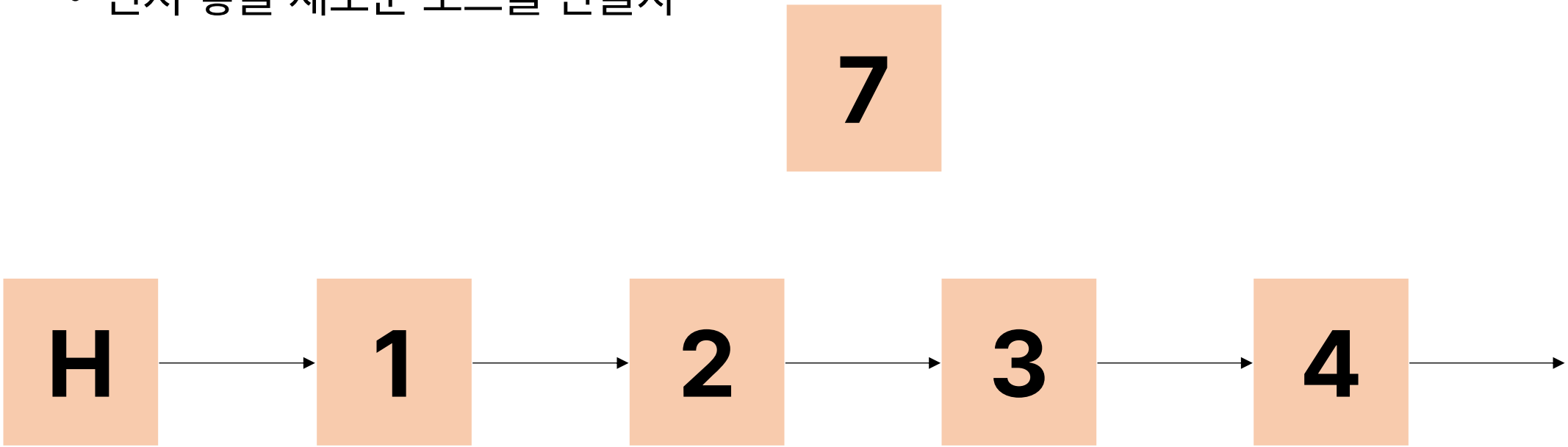
- 만일 찾지 못하고 리스트의 끝까지 간 경우,

  찾으려는 값이 없으므로 탐색에 실패(값을 찾지 못함)(값이 없음)



Linked List Insert

- 2와 3 사이에 7을 넣어보자
- 먼저 넣을 새로운 노드를 만들자



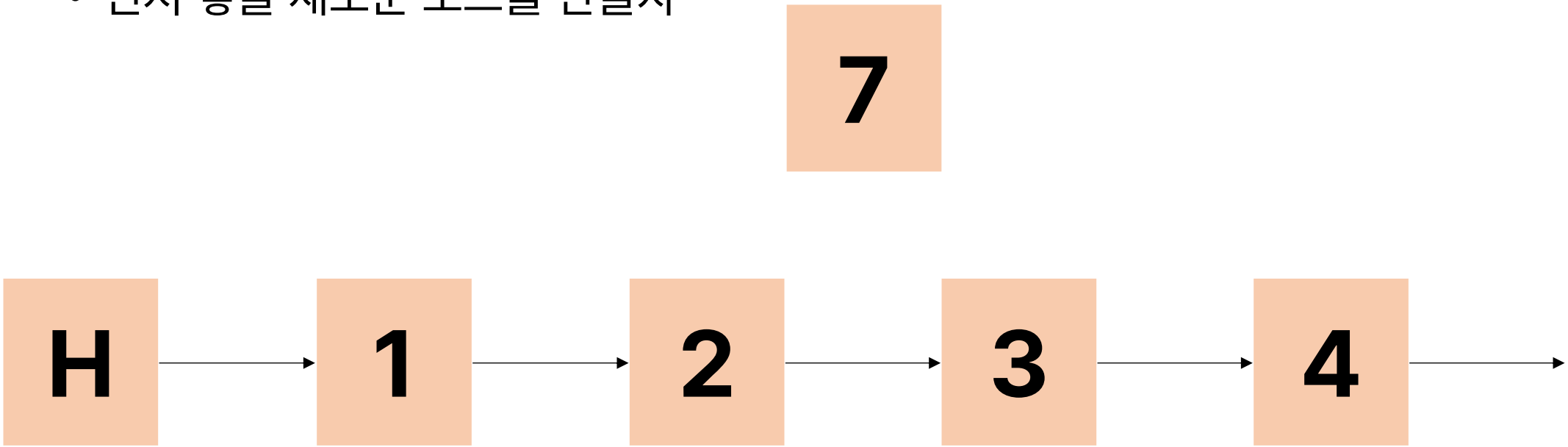
Linked List

- 기차를 생각해보자
- 기존에 있던 기차 중간에 새로운 칸 하나를 넣는다면 그 위치만 분리해서 새로운 기차 칸을 넣고 새로운 칸의 앞, 뒤를 연결해주면 된다

- 하지만 어떤 칸으로 이동하려는 경우, 1번째 칸에서 4번째 칸으로 간다고 한다면 2, 3 번째 칸을 거쳐서 가야한다

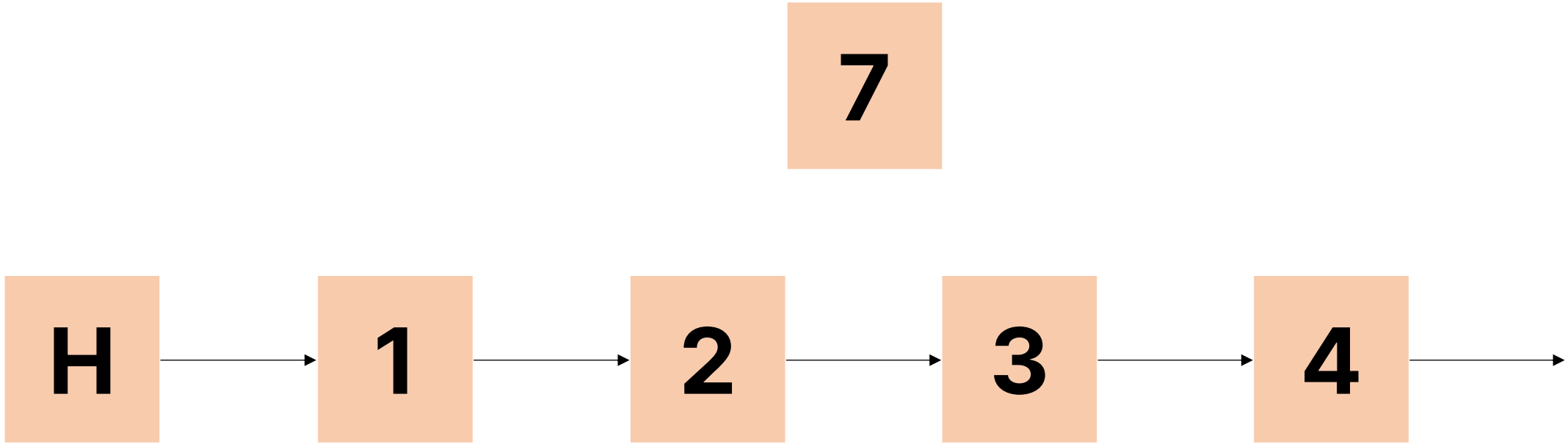
Linked List Insert

- 2와 3 사이에 7을 넣어보자
- 먼저 넣을 새로운 노드를 만들자



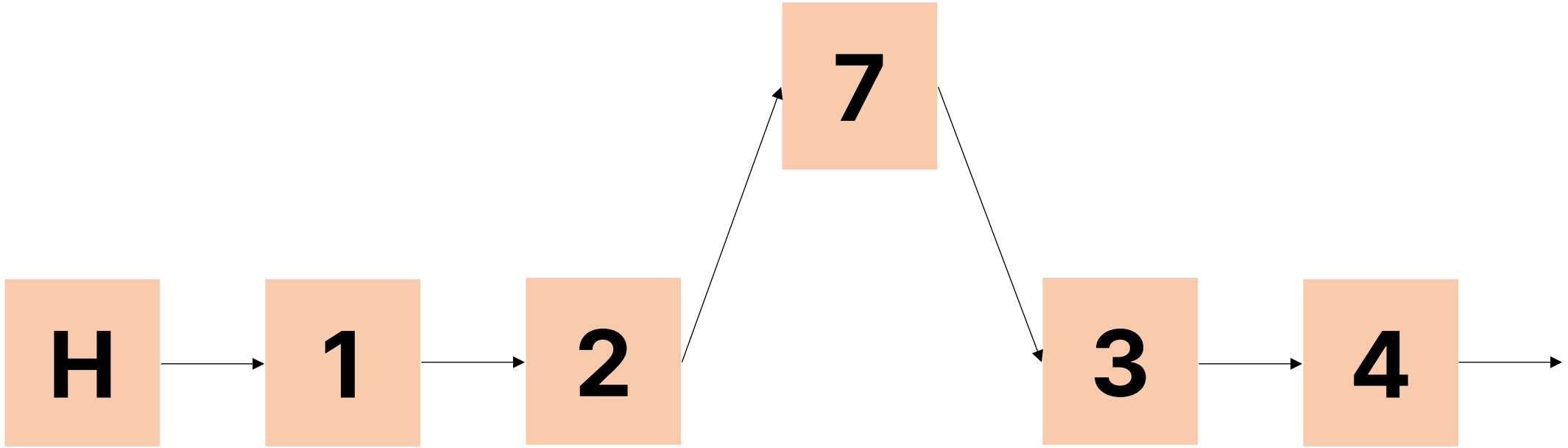
Linked List Insert

- 기존의 2와 3사이에 있던 연결을 바꾸자



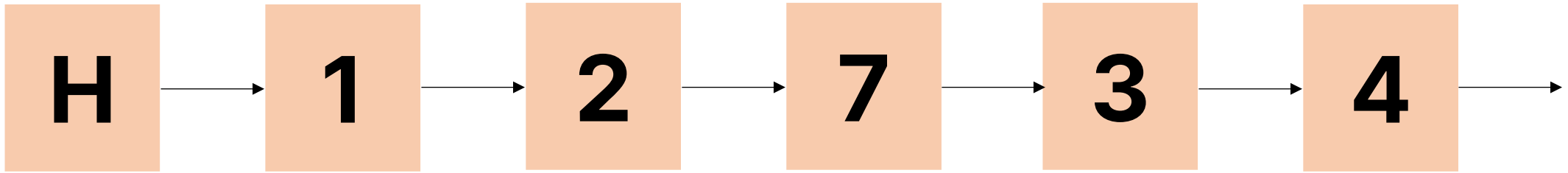
Linked List Insert

- 2 뒤에는 7이 오도록, 7 뒤에는 3이 오도록 연결을 바꾸자



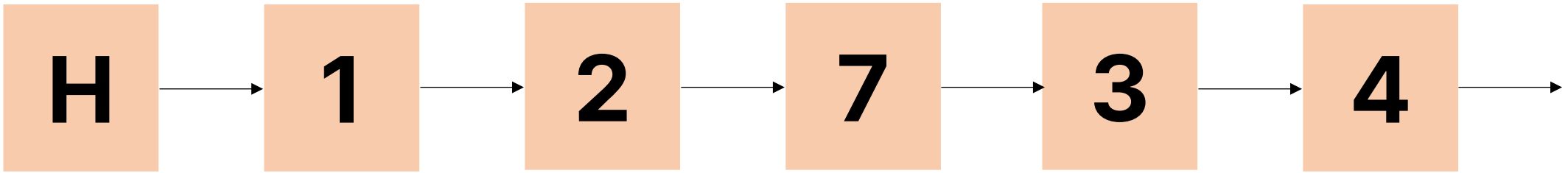
Linked List Insert

- 삽입 완료



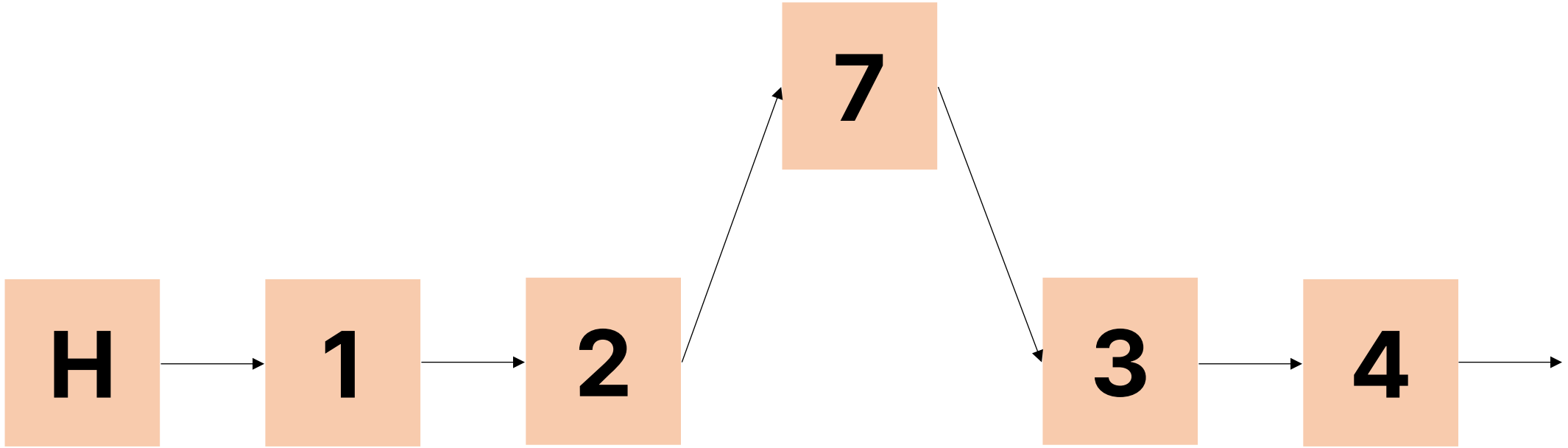
Linked List Delete

- 반대로 7을 삭제해보자



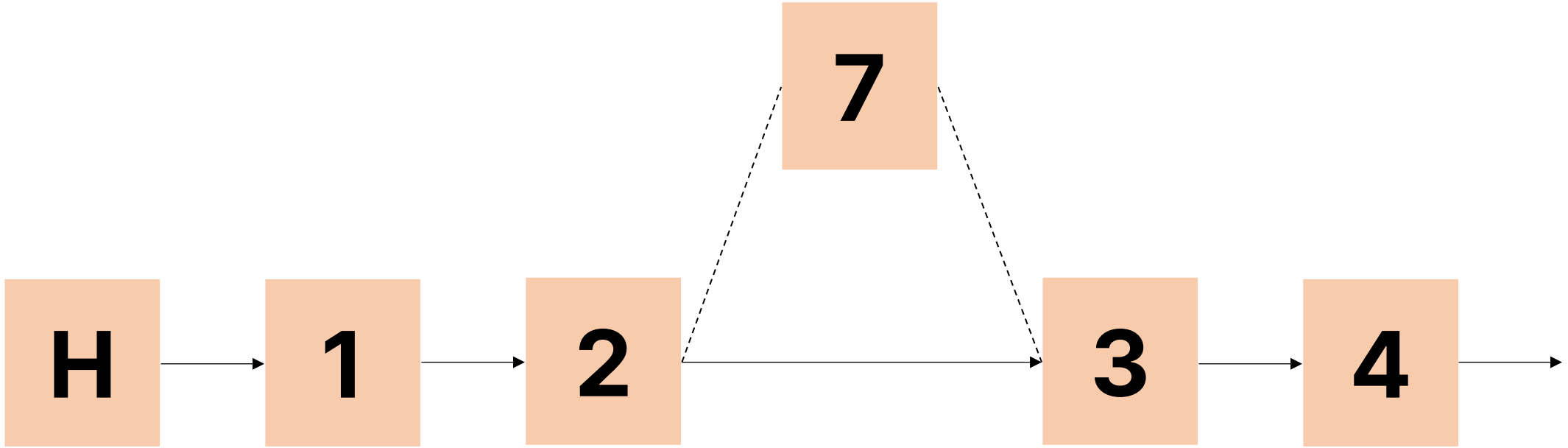
Linked List Delete

- 기존에는 2->7->3이므로, 2->3이 되도록 연결을 수정하면 된다



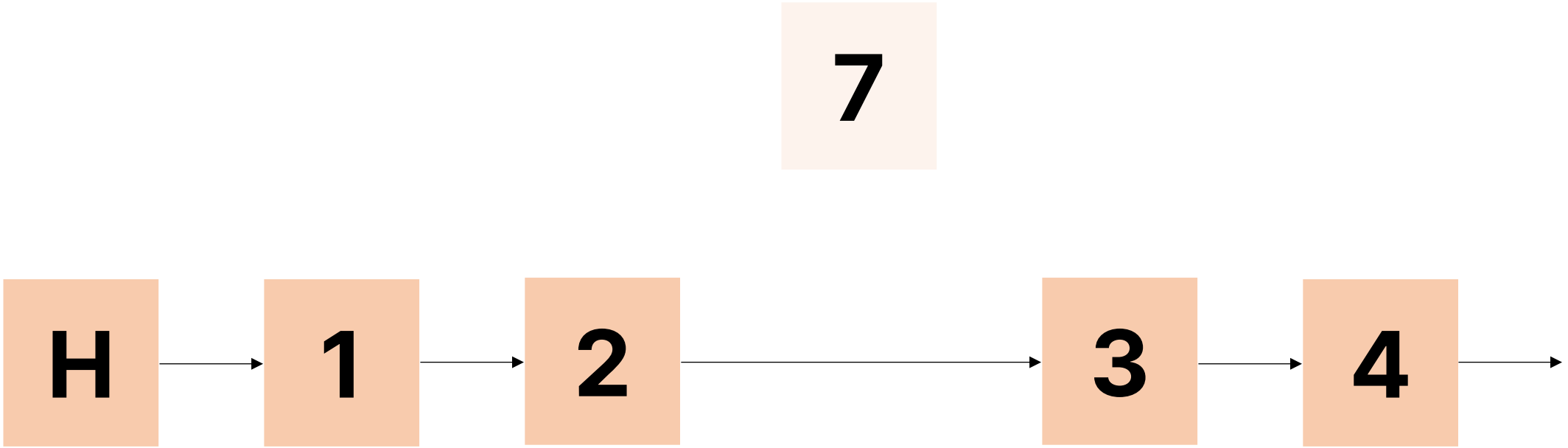
Linked List Delete

- 2->7 연결과 7->3 연결을 없애고 2->3 연결을 새로 만든다



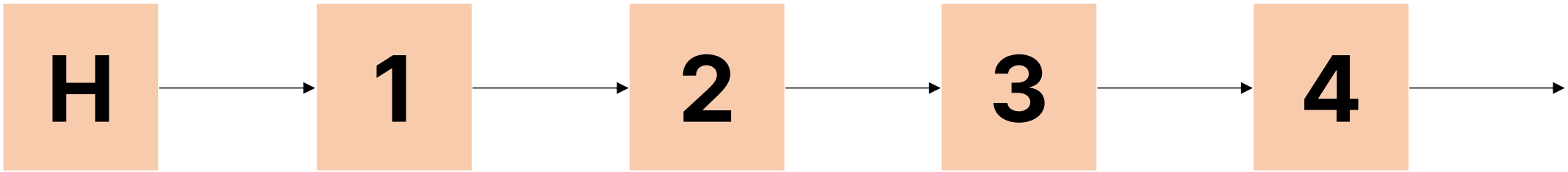
Linked List Delete

- 7 노드를 없앤다



Linked List Delete

- 삭제 완료



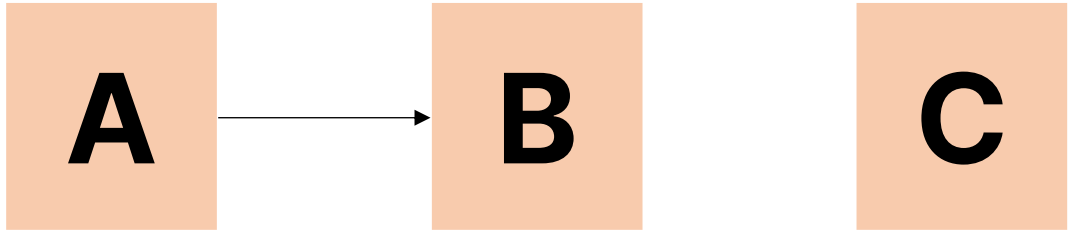
Linked List

- 그렇다면 언제 효율적일까? 일정한 구조를 유지한 채로 많은 양을 옮길 때

- ex) 파일 시스템
- 폴더를 이동한다고 했을 때, 폴더 안에 있는 모든 내용을 새로운 곳에 복사하고 원래 있던 내용을 지우는 것은 매우 오래 걸리고 비효율적임
- 폴더 계층 구조에 따라 상위 폴더->하위 폴더들 형식으로 연결되어 있음

Linked List

- A 폴더 안에 B 폴더가 있고 B 폴더를 C 폴더로 이동해보자

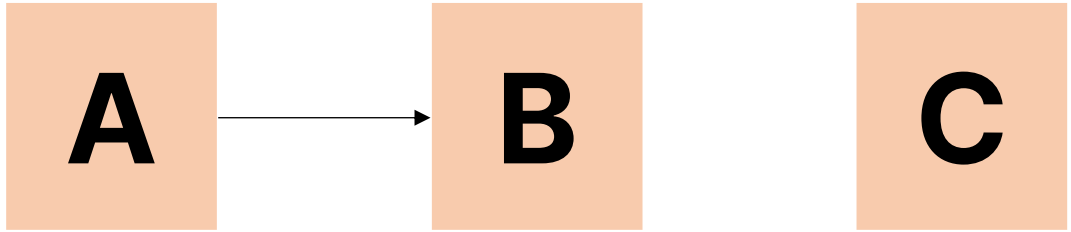


Linked List

- B′ 이라는 복사본을 만들고 모든 내용을 똑같이 이동하는 것은 비효율적이다
- 용량이 작을 때는 괜찮지만 100GB, 1TB 처럼 대용량 폴더인 경우에는?

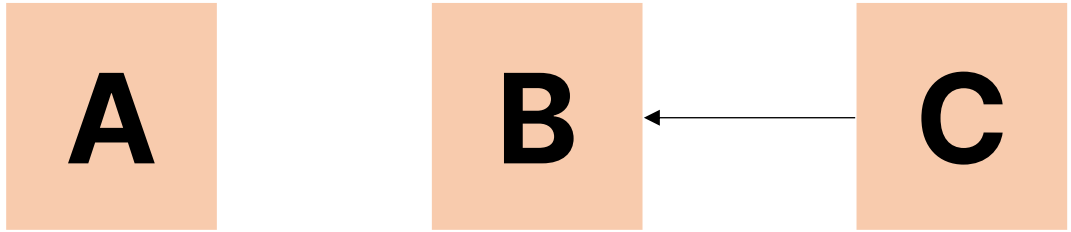
Linked List

- B 폴더는 그대로 놔두고 B폴더가 무슨 폴더 아래 있는지만 바꾸자



Linked List

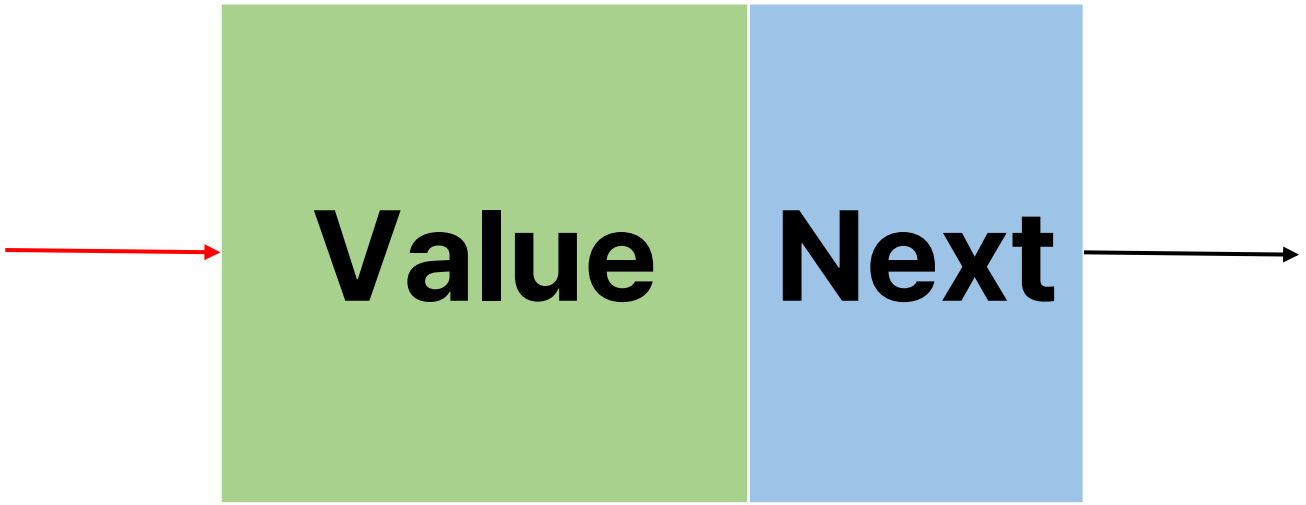
- A 폴더와의 연결을 없애고 C폴더와 새로운 연결을 하면 연결 2개를 수정하면서 이동이 가능하다



구현 Single Linked List Node

```
struct Node {
    int value;
    Node *next;

    Node() : value(0), next(nullptr) {}
    Node(int value) : value(value), next(nullptr) {}
    Node(int value, Node *next) : value(value), next(next) {}
};
```



구현 Single Linked List

```
struct LinkedList {
    Node *head;
    int sz;

    LinkedList() : head(new Node()), sz(0) {}
};
```

구현 Single Linked List Find

```
struct LinkedList {
    Node *find(int x) {
        Node *cur = head;
        while (cur->next != nullptr) {
            cur = cur->next;
            if (cur->value == x)
                return cur;
        }
        return nullptr;
    }
};
```

구현 Single Linked List Insert

```
struct LinkedList {
    bool insert(int data, int index) {
        if (sz < index)
            return false;
        Node *cur = head;
        while (index--)
            cur = cur->next;
        Node *new_node = new Node(data);
        if (new_node == nullptr)
            return false;
        new_node->next = cur->next;
        cur->next = new_node;
        sz++;
        return true;
    }
};
```

구현 Single Linked List Remove

```
struct LinkedList {
    bool remove(int index) {
        if (sz <= index)
            return false;
        Node *cur = head;
        while (index--)
            cur = cur->next;
        Node *remove_node = cur->next;
        cur->next = remove_node->next;
        sz--;
        delete remove_node;
        return true;
    }
};
```

구현 Single Linked List etc

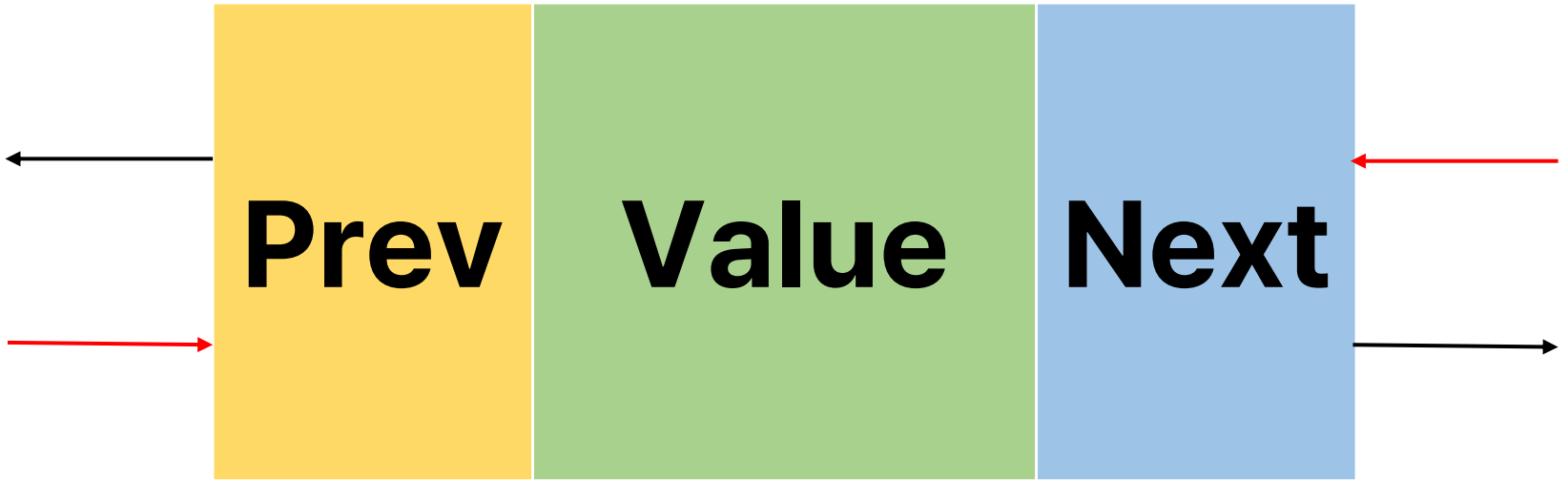
```
struct LinkedList {
    int size() { return sz; }

    bool empty() { return sz == 0; }

    void print() {
        Node *cur = head;
        while (cur->next) {
            cur = cur->next;
            cout << cur->value << ' ‘;
        }
        cout << endl;
    }
};
```

구현 Double Linked List Node

```
struct Node {
    int value;
    Node *next;
    Node *prev;
};
```



구현 Double Linked List Insert

```
bool insert(int data, int index) {
    if (sz < index)
        return false;
    Node *cur = head;
    while (index--)
        cur = cur->next;
    Node *new_node = new Node(data);
    if (new_node == nullptr)
        return false;
    new_node->next = cur->next;
    new_node->prev = cur;
    cur->next = new_node;
    if (new_node->next != nullptr)
        new_node->next->prev = new_node;
    sz++;
    return true;
}
```

구현 Double Linked List Delete

```
bool remove(int index) {
    if (sz <= index)
        return false;
    Node *cur = head;
    while (index--)
        cur = cur->next;
    Node *remove_node = cur->next;
    cur->next = remove_node->next;
    if (remove_node->next != nullptr)
        remove_node->next->prev = remove_node->prev;
    sz--;
    delete remove_node;
    return true;
}
```

문제

- 에디터 BOJ 1406