17차시

Python

파이썬

- 파이썬은 매우 편리한 언어이다

  ex) 리스트 뒤집기 list[::-1]

- 기타 언어들은 어떤 기능을 사용할 때, 구현해야 하는 경우가 많다(대표적으로 C)
- 파이썬은 만들어진 라이브러리가 다른 언어들에 비해 많다
- 학교에서도 가르쳐 주고 딥러닝, AI 등 현재 널리 쓰인다

PS에서 Python

- PS는 Problem Solving의 약자, 대표적으로 백준, codeforce 등 알고리즘 문제를 푸는 것을 PS라고 한다

- 파이썬은 **편리한** 언어이다. 대신 속도가 매우 느리다
- PS에서 파이썬을 사용하기에는 많은 문제점이 있다

PS에서 Python의 문제점

- 외장 라이브러리를 지원하지 않는다
- 느리다. 느려서 맞는 풀이임에도 시간초과가 나는 경우가 있다
- 리스트를 사용하는 경우 굉장히 느리다(느린 파이썬 위에 느린 리스트)

  ex) 리스트의 개수를 셀 때, 파이썬은 하나하나 다 세고 알려준다
- 그래서 파이썬에서는 리스트를 큐나 덱으로 절대절대 사용하면 안된다(진짜 느림)
- 재귀 깊이가 1000으로 제한되어 있다
- ~~그냥 안좋다~~

Python이 왜 느린가

- 리스트는 구현이 느리게 되어 있어서 느리다(파이썬이 느린 것과는 별개)

Python이 왜 느린가

- 우리가 만든 코드를 컴퓨터가 실행할 수 있는 0과 1로 바꿔줘야한다

  ex) c = a + b; 라는 코드는 a를 불러오고, b를 불러오고 더한 후 c에 저장한다는 내용으로 바뀌어야 한다

- C언어, Java와 같은 언어는 컴파일 언어로 프로그램을 실행시키기 전 모든 내용을 변환한다

- Python은 코드를 변환하며 실행하는 인터프리터 언어이다

  변환->실행->변환->실행->... 당연히 프로그램 실행 중에 변환하니까 느리다

그래도 파이썬을 쓰고 싶다

- 재귀 깊이를 신경쓰자
- 재귀 깊이(함수 깊이)는 함수 속에서 얼마나 다시 함수가 호출 되었는지를 의미
- ex) main에서 A함수를 호출하면 main(깊이 0)에서 A함수(깊이 1)로 내려간다
- 깊이가 1인 함수 속에서 함수를 호출하면 깊이가 2가 된다
- 재귀 함수는 본인 함수를 다시 호출하므로 함수 깊이가 꾸준히 증가한다(제한이 1000)
- sys.setrecursionlimit을 이용해 재귀 깊이를 조절하자

그래도 파이썬을 쓰고 싶다

- 입출력 속도를 챙기자
- 모든 프로그래밍 언어에서 입출력은 굉장히 느린 부분 중 하나이다
- PS시에는 일반적인 입출력이 아닌, 더 빠른 입출력을 사용하는 것을 권장한다
- input 대신에 sys.stdin.readline을 사용하자

Python을 빠르게 PyPy

- PyPy는 컴파일러와 인터프리터 방식을 조합한 형태
- 파이썬과 PyPy 모두 모든 코드를 실행할 때 변환한다
- PyPy는 한번 변환한 코드를 기억하고 있는다(Caching)
- 반복문처럼 기존에 변환한 코드를 다시 실행 시 기존에 변환한 코드를 가져다 사용하면서 변환하는 시간을 아낀다

PyPy는 만능인가?

- 메모리가 파이썬에 비해 많이 사용된다(변환한 코드를 기억하기 때문)
- 파이썬은 재귀에 제약이 있지만, PyPy는 재귀 횟수에 제약이 없다.
- 하지만 메모리를 더 사용하기 때문에 동일한 메모리 제한에서 PyPy는 파이썬보다 재귀의 제한이 낮다, 재귀에 굉장히 취약하다
- Immutable(int, float, str, tuple) 연산이 파이썬보다 느리다
- 그래도 파이썬보다는 빠르니 쓰자~~(이왕이면 C++을 하자)~~

Pointer

포인터가 어렵나요?

- 네.

포인터가 정말 어려운 개념인가요?

- 네.

- (그런데 꼭 배워야해요...)

포인터를 이해하려면 자료형을 알아야한다

종류	자료형	크기
정수형 signed/unsigned	char	1 Byte
	bool	1 Byte
	short	2 Byte
	int	4 Byte
	long	4/8 Byte
	long long	8 Byte
실수형 (부동소수점)	float	4 Byte
	double	8 Byte

Long을 이해하려면 포인터를 알아야한다(?)

- 컴퓨터에는 32비트 운영체제와 64비트 운영체제가 있다

  ex) 윈도우 10 32비트, 윈도우 10 64비트

- 뒤에 적힌 비트는 메모리를 나타낼 때 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

32비트 운영체제

- 32비트 운영체제는 메모리의 주소를 나타낼 때 32비트를 사용한다

  32비트: 0 또는 1이 32개

- 제일 작은 숫자: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$
- 제일 큰 숫자: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$

  $1\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$ 에서 1을 뺀 것과 동일

이진수 Binary

- 10진수는 0, 1, 2, …, 점점 숫자가 커지다 각 자리수가 10이 되는 순간 자리수가 넘어 가는 체계
- 2진수는 숫자가 커지다 각 자리수가 2가 되는 순간 자리수가 넘어 가는 체계

  0_2, 1_2, 10_2 , 11_2 , 100_2 , 101_2 , 110_2 , 111_2, 1000_2 , 1001_2 , 1010_2 , 1011_2

이진수 Binary

- 10진수에서 각 자리수의 크기는 10^0, 10^1, 10^2, 10^3, 10^4, ... 10의 거듭제곱으로 커진다
- 2진수에서는 2^0, 2^1, 2^2, 2^3, 2^4, ... 2의 거듭제곱으로 커진다

- 1101_2는$2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1$이므로 10진수로 13이다

32비트 운영체제

- 제일 큰 숫자: 1111 1111 1111 1111 1111 1111 1111 1111_2

  1 0000 0000 0000 0000 0000 0000 0000 0000_2 에서 1을 뺀 것과 동일

- 10진수로 변환하면 $2^{32} - 1$이며 4,294,967,295(42억 9496만 7295)이다

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

32비트 운영체제

- 32비트로 나타낼 수 있는 수는 4,294,967,295(42억 9496만 7295)이므로 약 42억 개의 셀의 주소를 나타낼 수 있다
- 1GB는 10억 Byte이므로 1GB를 사용하기 위해서는 10억 개의 셀이 필요하다
- 따라서 32비트 운영체제는 4GB까지의 메모리까지 사용이 가능하다

32비트 운영체제

- 과거에는 4기가 이상으로 메모리를 사용하지 않았다
- 4기가 이상으로 메모리를 사용하지 않는데 64비트나 사용하는 것은 메모리 낭비
- 컴퓨터가 발전하면서 64비트 운영체제가 나오게 되었다

- 그럼 컴퓨터가 더욱 발전하면 64비트도 부족한가요?
- 네.

Long을 이해하려면 포인터를 알아야한다(?)

- Long은 운영체제의 비트 수를 따라간다

  32비트 운영체제에서는 32비트(4 Byte)

  64비트 운영체제에서는 64비트(8 Byte)

- 따라서 운영체제에 따라 자료형의 크기가 변화하는 것

자료형이라는 것은?

- 어떤 변수를 만들면 해당하는 자료형의 크기만큼 메모리를 사용하고 있다는 것
- 이는 미리 정의된 자료형 뿐만 아니라 우리가 만드는 구조체도 동일하다

- 우측 코드에 있는 구조체의 크기는

  int가 2개, char가 1개이므로

  $4\times2 + 1\times1 = 13$이다

```
struct Size {
    int a;
    int b;
    char c;
};
```

포인터

- 메모리 주소를 저장하고, 특정 자료형으로 저장한 주소에 접근하기 위한 자료형

포인터

- 메모리 주소를 저장하고, 특정 자료형으로 저장한 주소에 접근하기 위한 자료형

포인터

- 포인터 변수에 담기는 것은 메모리 주소이다

- 메모리 주소를 담아야 하므로 운영체제가 메모리 주소를 나타낼 때 32비트를 사용하면 32비트, 64비트를 사용하면 64비트를 담아야 한다
- Long처럼 운영체제에 따라 크기가 다르다

포인터

- 주소를 접근할 때 사용할 자료형이 필요하다

- 메모리에 들어있는 내용은 0과 1뿐이다
- 0100 0001은 무슨 내용인가?를 해석할 자료형이 필요하다

포인터

- 0100 0001_2를 10진수로 변환하면 65이다
- int에 65를 담는 경우, 숫자 그대로 65이다
- char에 65를 담는 경우, 문자 'A를 나타낸다

- 이렇게 해석이 달라지므로, 해당 내용을 어떻게 사용할지 나타내는 자료형이 필요하다

포인터

- 포인터도 자료형이다

- 포인터 변수 또한 메모리에 공간을 차지하며, 앞서 얘기한 것처럼 운영체제에 따라 다른 크기를 갖는다(32비트, 64비트)

포인터 변수는 어떻게 만드나요?

- 기존의 변수는 자료형 변수명;으로 선언
- 포인터 변수는 자료형 *변수명;으로 선언

포인터 변수는 어떻게 만드나요?

- 기존의 변수는 자료형 변수명;으로 선언
- 포인터 변수는 자료형 *변수명;으로 선언

- 자료형 뒤에 *을 하나 붙인다
- 이중, 삼중, … 포인터들도 존재한다. 일단 기본적인 포인터만 생각하자(어려워요)

포인터 변수는 어떻게 만드나요?

- 왜 *을 하나 붙이나요?
- 포인터 변수는 다 똑같이 메모리 주소를 저장하니까 pointer 변수명;으로 사용하지 않나요?

- “주소를 접근할 때 사용할 자료형이 필요하다”
- 해당 주소를 해석할 자료형이 필요해요

포인터 변수를 다시 살펴보면

- 자료형 *변수명;

- 자료형: 포인터 변수에 담겨있는 주소로 가서 해석할 때 사용할 자료형을 의미
- *: 포인터 변수임을 나타냄
- 변수명: 변수명

포인터 변수를 다시 살펴보면

- int *var;

- var은 포인터 변수이며, var에 담겨있는 주소에는 int가 저장되어 있음을 뜻함

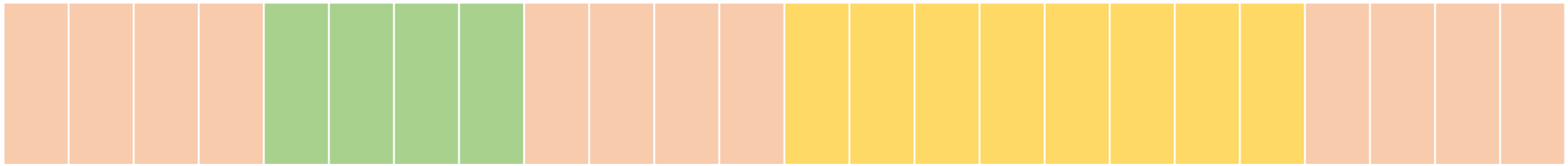
포인터

- 노란색: int *C;

  포인터 변수는 8바이트이므로 8칸 차지

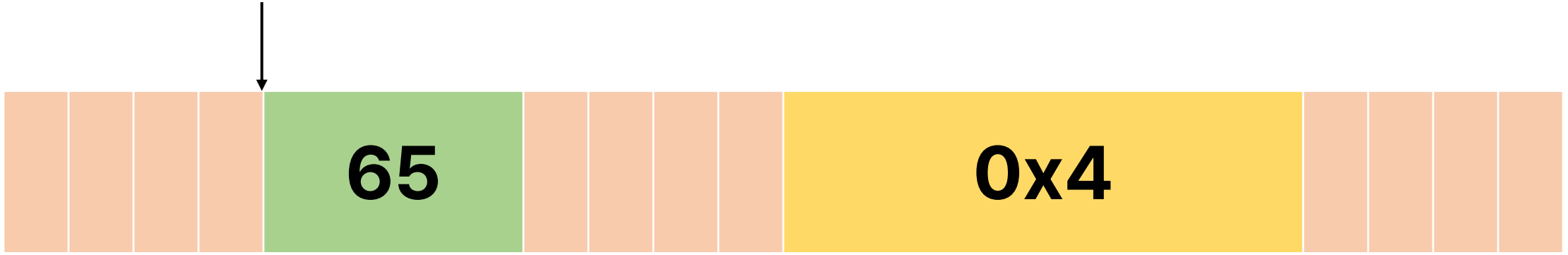
- 초록색: I am int에요

  int는 4바이트 이므로 4칸 차지



포인터

- 포인터는 컴퓨터가 쓰기 편하게 16진수로 되어 있음
- 16진수로 되어 있기 때문에 0x라는 접두사가 붙음
- 포인터가 4를 가르키므로 초록색 int 변수를 가르킴



포인터 사용법

- *과 &연산자를 이용하여 사용
- +, -, * 와 동일하다

- A+B: A와 B를 더한 결과를 알려줌
- A*B: A와 B를 곱한 결과를 알려줌
- -A: A를 음수로 바꾼 결과를 알려줌

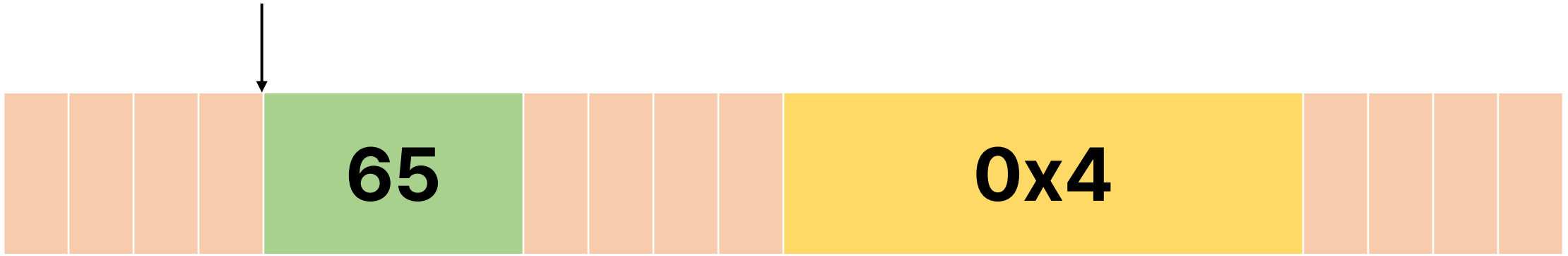
포인터 사용법

- *과 &연산자를 이용하여 사용

- *C: C의 담겨있는 주소로 가서 값을 해석해줌
- &A: A변수의 주소를 알려줌

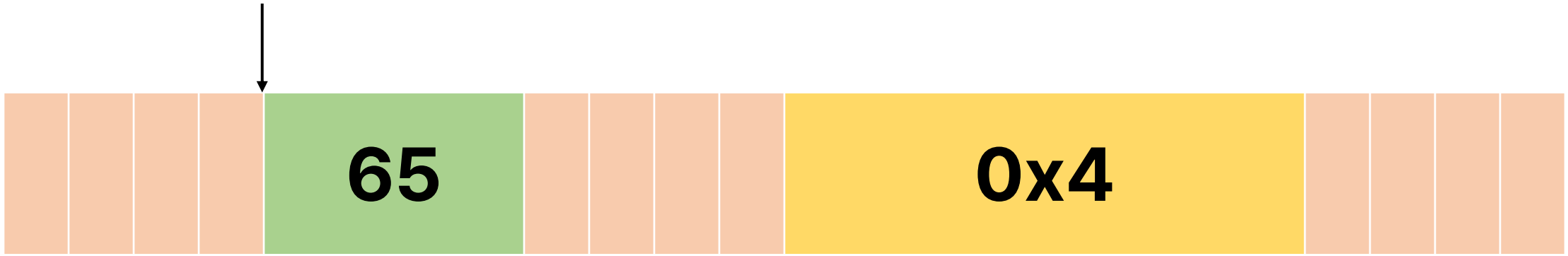
포인터

- 노란색: int *C;
- 초록색: int A;



포인터

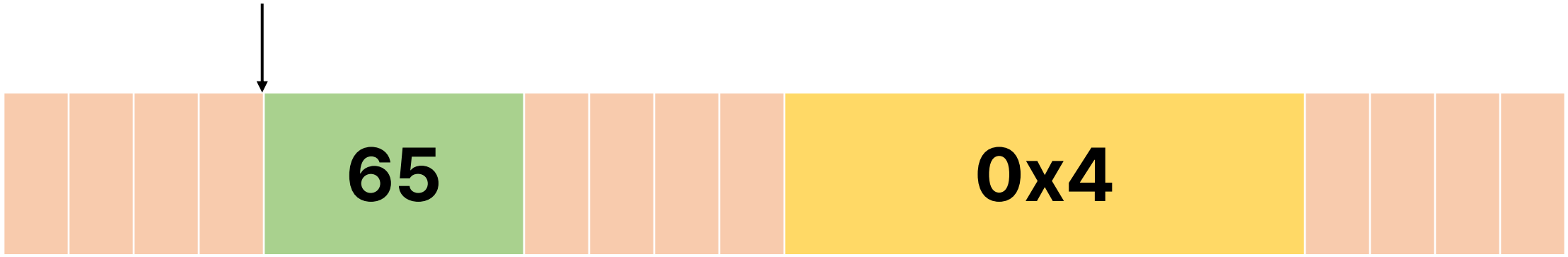
- A: 65
- &A: 0x4
- C: 0x4
- *C: 65



포인터

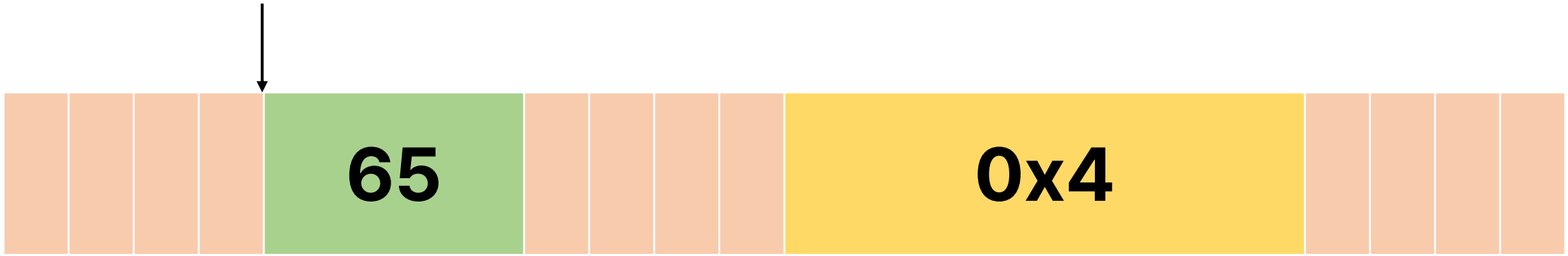
- &C: 0xC
- 포인터 변수도 변수다
- 저장되어 있는 곳을 나타내면 된다. 12번째 칸이므로 16진수로 0xC이다

  0 1 2 3 4 5 6 7 8 9 A B C D E F



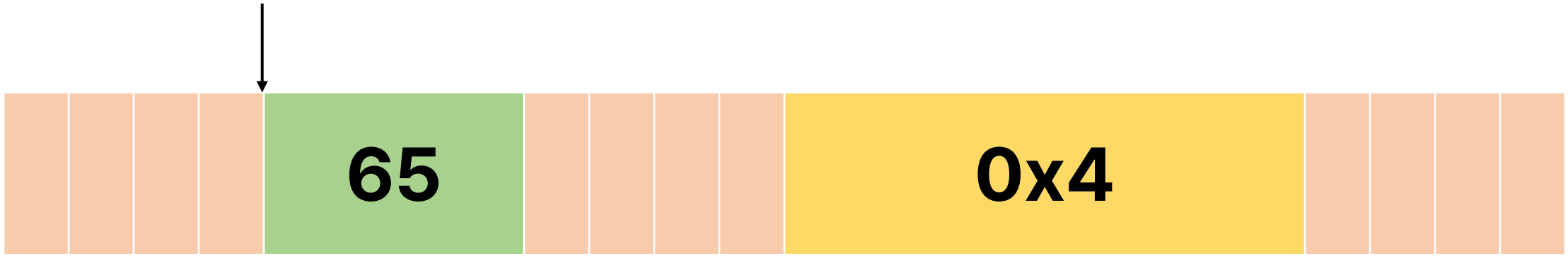
포인터

- *A: ?
- 불가능하다
- int는 포인터 변수가 아니므로 *를 지원하지 않는다



포인터

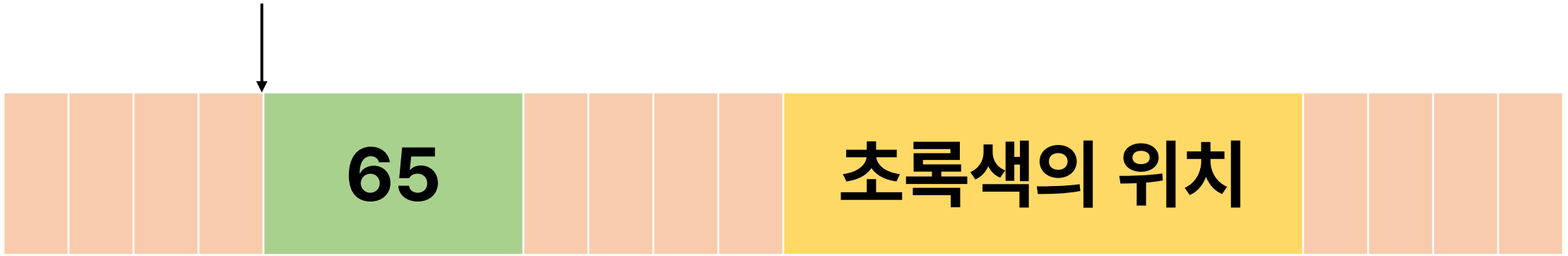
- *A: ?
- 포인터 변수가 아니므로 불가능하다
- int에서 *은 곱셈이다
- 반대로 포인터는 곱셈이 불가능하다. 정확히는 할 필요가 없다



포인터 코드 예시

- 실제 코드로는?
- C라는 int 포인터 변수에 A 변수의 주소를 담겠다

```
int A = 65;
int *C;
C = &A;
```



포인터 코드 예시

- 실제 코드로는?
- C를 해석한 값이, A이다 = A의 위치를 C가 가지고 있다

```
int A = 65;
int *C;
*C = A;
```

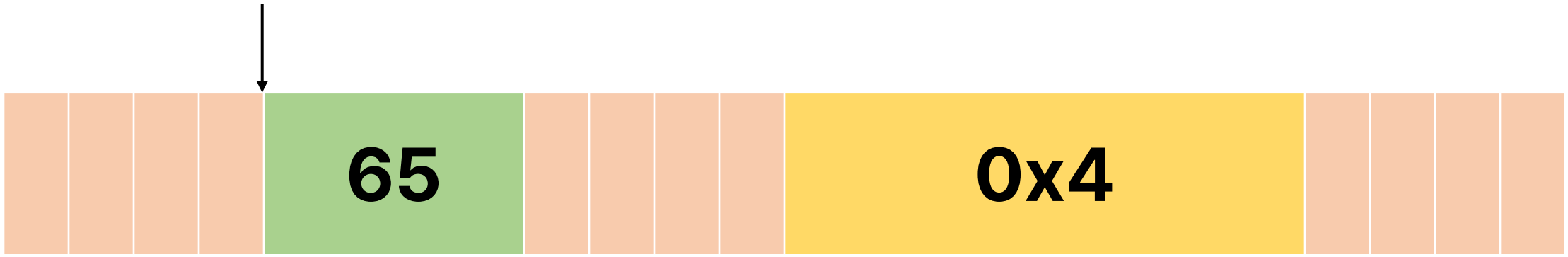
Single Linked List Node

- Node 포인터 변수인 next는 next에 담긴 주소로 가서 Node 구조체로 해석하겠다
- next에 담긴 값은 Node 구조체의 주소이다

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

포인터 심화과정

- &C: 0xC
- 포인터 변수의 주소가 존재한다
- 얘의 주소를 담는 애는?



포인터 심화과정

- 포인터 변수의 주소가 존재한다
- 포인터 변수를 가르키는 포인터 변수를 만들자(이중 포인터)
- 포인터 변수를 가르키는 포인터 변수를 가르키는 포인터 변수를 만들자(삼중 포인터)

…

- 가루 삼겹살을 드셔보시겠습니까

포인터 심화과정

- 포인터도 자료형이다
- 이중 포인터 또한 자료형이다

- 포인터를 만드는 방식은 자료형에 *을 붙인다

이중 포인터를 만들자

- 이중 포인터는 포인터를 가르키는 포인터
- 가르킬 자료형이 포인터
- 포인터 변수이므로 *

- 포인터 * 변수명
- ex) int **변수명

int 이중 포인터

- int **D
- int*, 주소가 나타내는 곳에 int 포인터가 있다
- *: 포인터 변수임을 나타냄
- D: 변수명

- D가 담고 있는 주소에는 int*가 담겨있다

삼중 포인터를 만들자

- 삼중 포인터는 이중 포인터를 가르키는 포인터
- 가르킬 자료형이 이중 포인터
- 포인터 변수이므로 *

- 이중 포인터 * 변수명
- ex) int ***변수명

int 삼중 포인터

- int ***E
- int**: 주소가 나타내는 곳에 int 이중 포인터가 있다
- *: 포인터 변수임을 나타냄
- E: 변수명

- E가 담고 있는 주소에는 int**가 담겨있다

포인터를 왜 쓰나요

- 내가 원하는 변수의 값을 바꾸기 위해서

- 함수를 호출하면 인자로 넣은 값이 복사된다.
- 함수를 사용하는 공간에서 새로운 변수가 만들어지고 같은 값을 갖게 된다

포인터를 왜 쓰나요

```
void change(int a) { a = 5; }

int main() {
    int a = 1;
    change(a);
    cout << a;

return 0;
}
```

포인터를 왜 쓰나요

- a에다 5라는 값을 넣었지만 1이 출력된다
- change라는 함수를 실행하면 함수가 사용할 공간이 만들어진다
- a라는 변수를 위한 공간이 만들어지고 여기에 원래 a의 값(1)이 복사된다
- a에 5를 대입하면 main의 a가 아닌 새로 만들어진 함수 공간 속 a에 5가 들어감
- 함수가 종료되면 main의 a는 변하지 않음

포인터를 왜 쓰나요

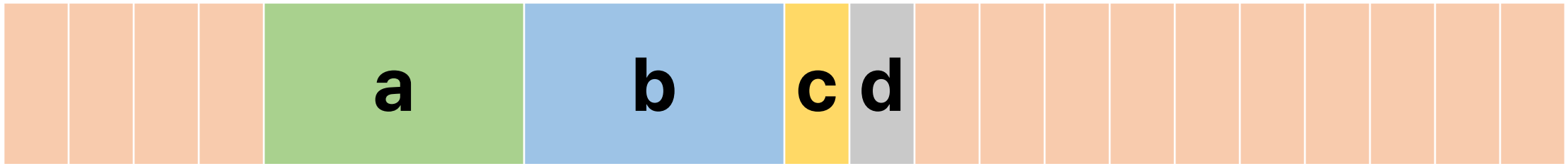
- a가 주소라면?
- 주소는 복사되어도 같은 위치를 가르킨다
- main 속 a의 주소를 복사해 가져가면 다른 공간(함수 만의 공간)에서도 main의 a가 어디 있는 지 알 수 있다
- main의 a의 값을 변경할 수 있다
- **배열처럼 연속적으로 메모리가 할당되어 있다면 원하는 위치의 값을 가져올 수 있다**

포인터를 이렇게 사용할 수 있어요

```
int a[3];
a[0] = 0;
a[1] = 1;
a[2] = 2;
cout << *(int *)((long long)a + sizeof(int)) << endl;     // 1
cout << *(int *)((long long)a + 2 * sizeof(int)) << endl; // 2
```

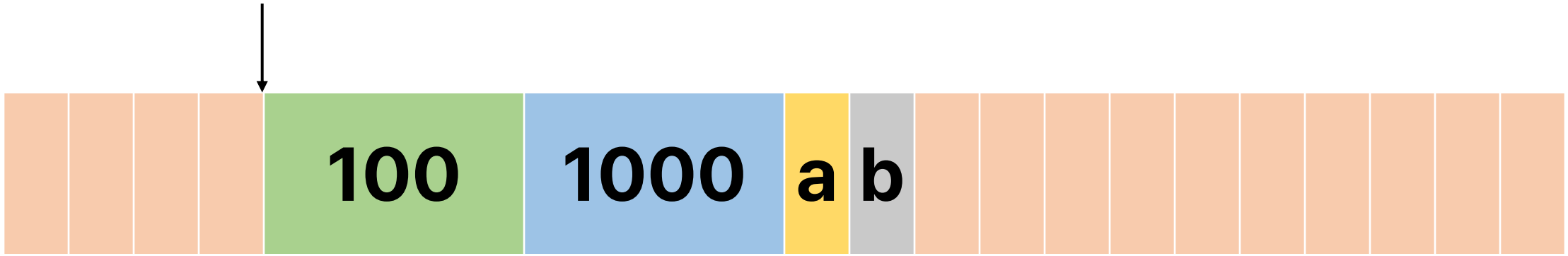
포인터를 이렇게 사용할 수 있어요

```
struct PointerTest {
    int a;
    int b;
    char c;
    char d;
};
```



포인터를 이렇게 사용할 수 있어요

```
PointerTest a;
a.a = 100;
a.b = 1000;
a.c = 'a';
a.d = 'b';
cout << *(int *)((long long)&a) << endl;                                    // 100
cout << *(int *)((long long)&a + sizeof(int)) << endl;                      // 1000
cout << *(char *)((long long)&a + 2 * sizeof(int)) << endl;                 // a
cout << *(char *)((long long)&a + 2 * sizeof(int) + sizeof(char)) << endl; // b
```



Stack

Stack

- Last in First out(LIFO)
- 늦게 들어온 원소가 먼저 나가는 자료구조
- 한 쪽을 통해서만 값이 들어오고 나가는게 가능한 구조

Stack

- 왼쪽 끝이 바닥, 값이 들어오고 나가는 것은 항상 오른쪽에서 진행

Stack Push(Insert)

- 항상 값은 오른쪽에서 들어와서 왼쪽 끝부터 채워진다

Stack Push(Insert)

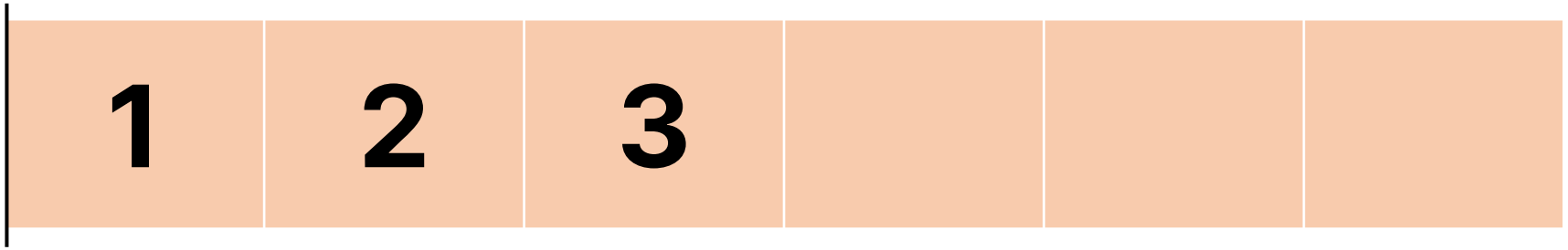
- 1 삽입

Stack Push(Insert)

- 2 삽입

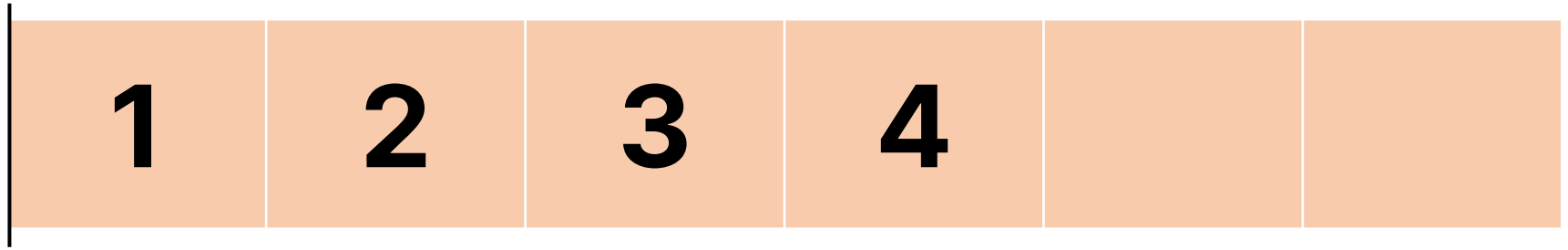
Stack Push(Insert)

- 3 삽입



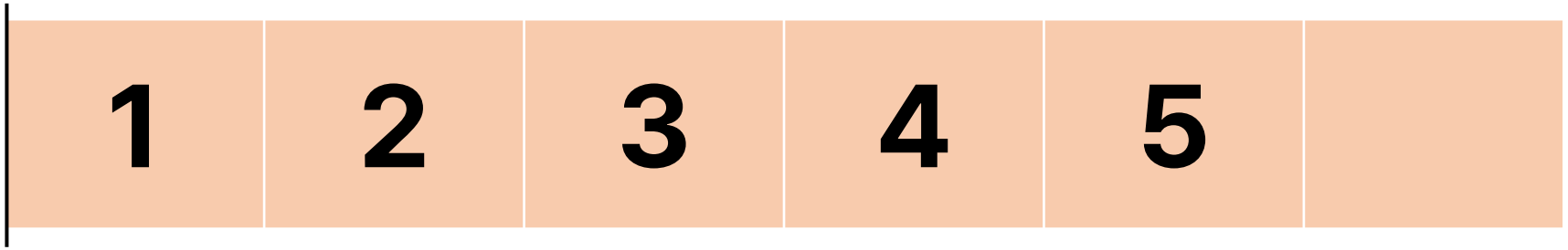
Stack Push(Insert)

- 4 삽입



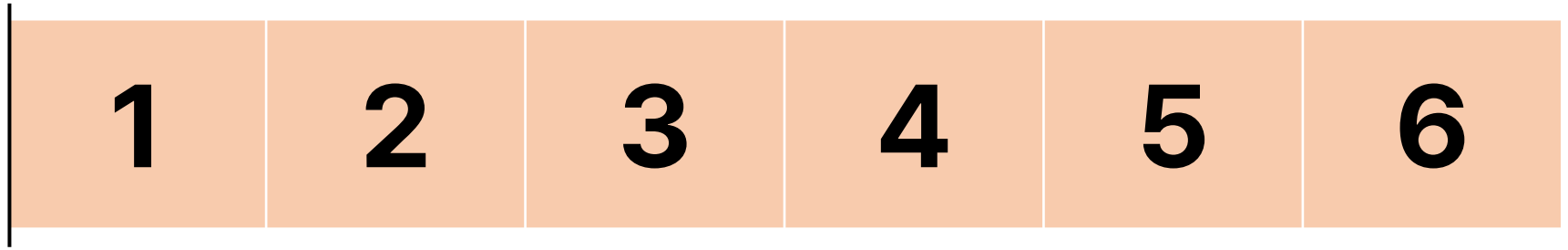
Stack Push(Insert)

- 5 삽입



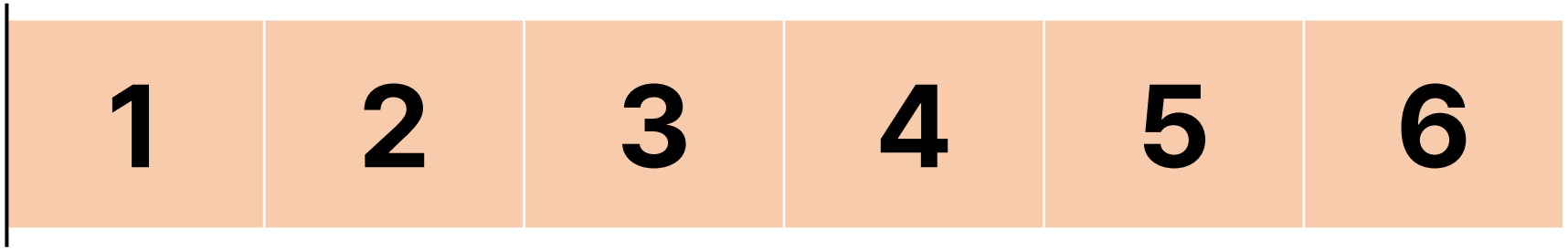
Stack Push(Insert)

- 6 삽입



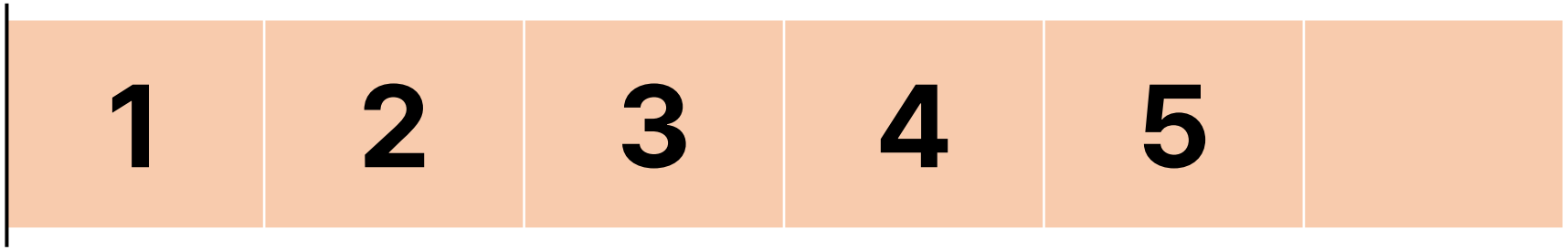
Stack Pop(Delete)

- 값을 뺄 때는 가장 오른쪽에 있는 값부터 뺀다



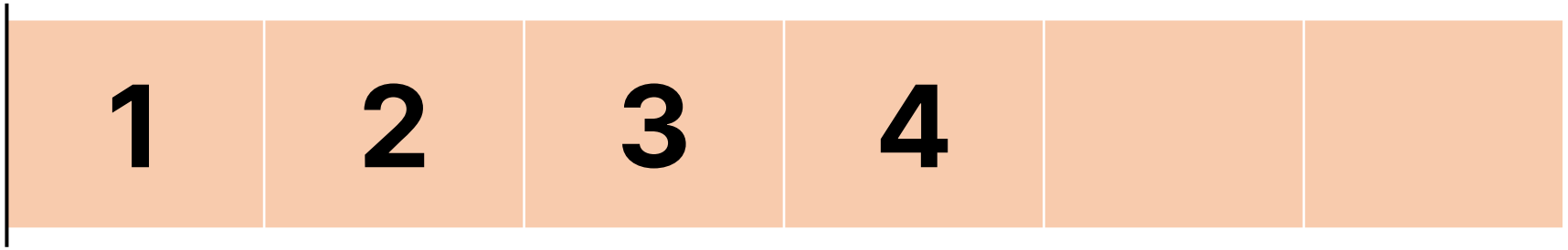
Stack Pop(Delete)

- 삭제(6)



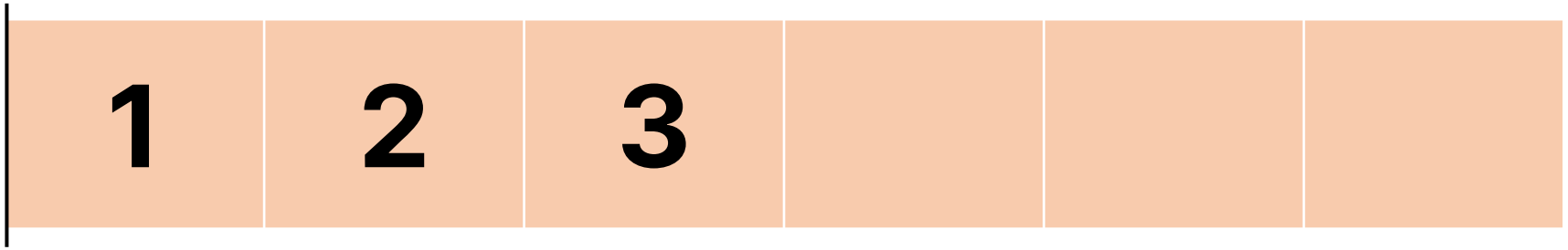
Stack Pop(Delete)

- 삭제(5)



Stack Pop(Delete)

- 삭제(4)



Stack Pop(Delete)

- 삭제(3)

Stack Pop(Delete)

- 삭제(2)

Stack Pop(Delete)

- 삭제(1)

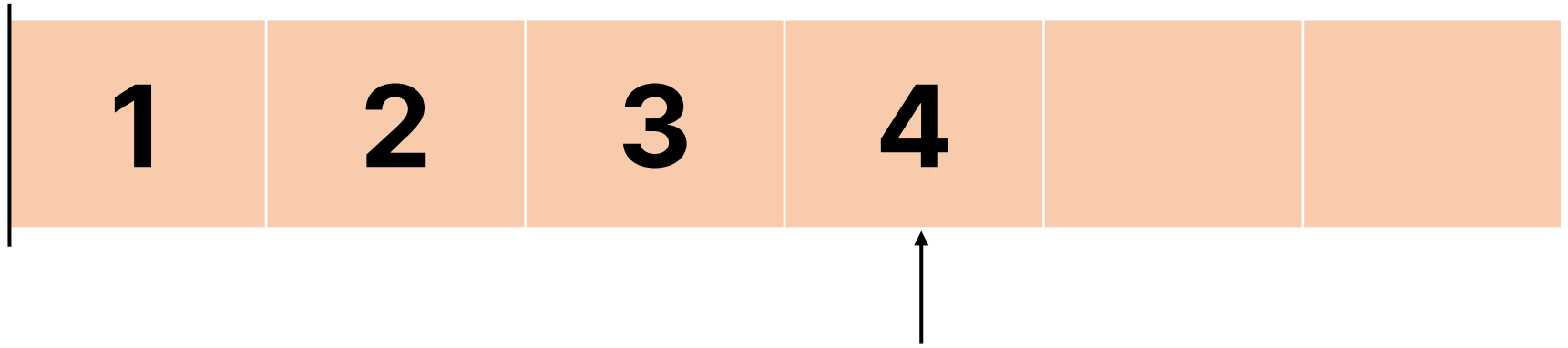
Stack 구현

- 스택을 구현하기 위해서는 2가지가 필요하다

- 데이터를 담을 배열
- top 변수

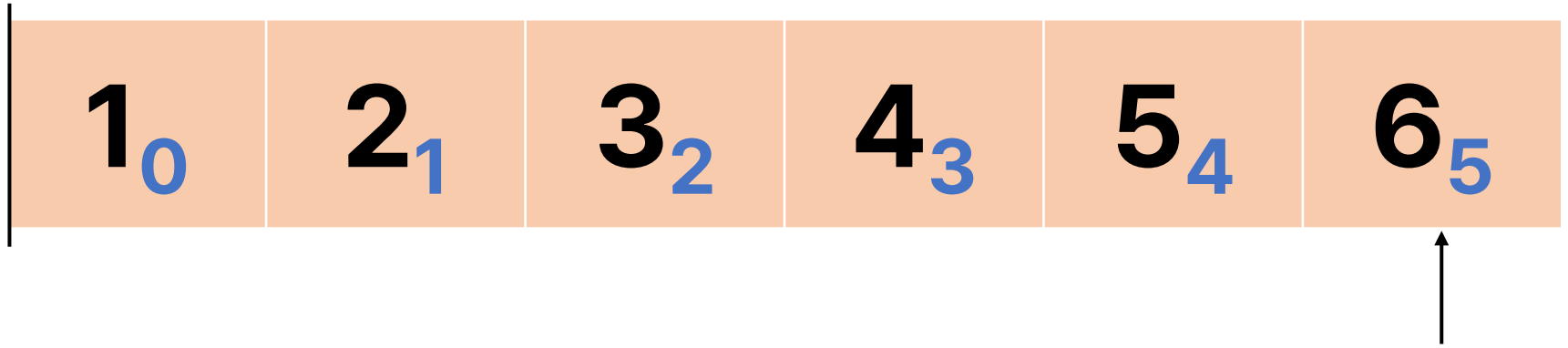
Stack 구현

- top 변수는 마지막 데이터가 존재하는 위치
- 따라서 top 변수의 값은 데이터의 개수 – 1(배열의 인덱스는 0부터 시작하기 때문)



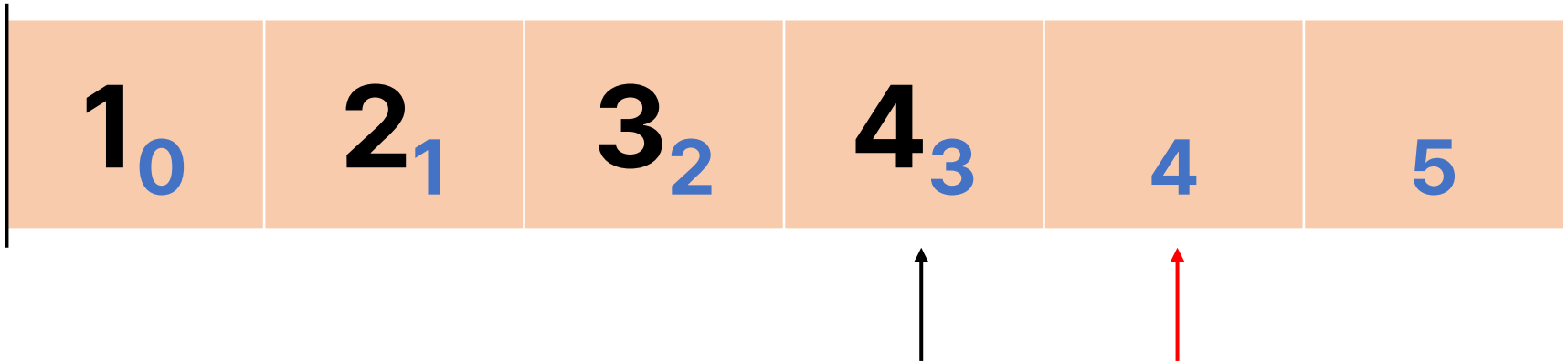
Stack 구현

- 스택에 들어간 개수를 알기 위해서는 top + 1
- top + 1 이 배열의 크기와 동일하다면 스택이 꽉 찬 것



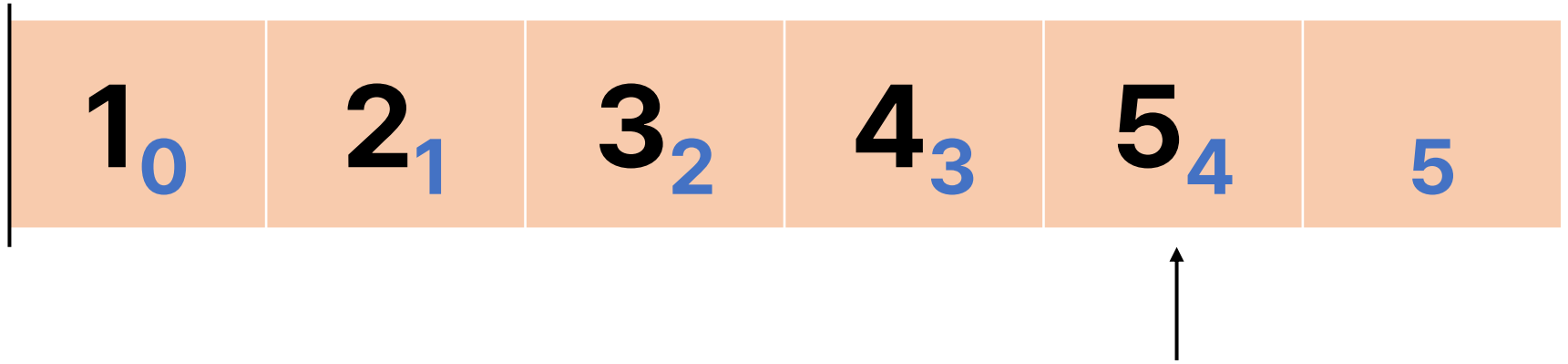
Stack 구현 – Push

- top 변수가 마지막으로 들어간 데이터의 위치를 기억하므로, 새롭게 들어갈 데이터의 위치는 top + 1이다
- 데이터를 넣기 전 배열의 공간이 충분한지 확인해야 한다



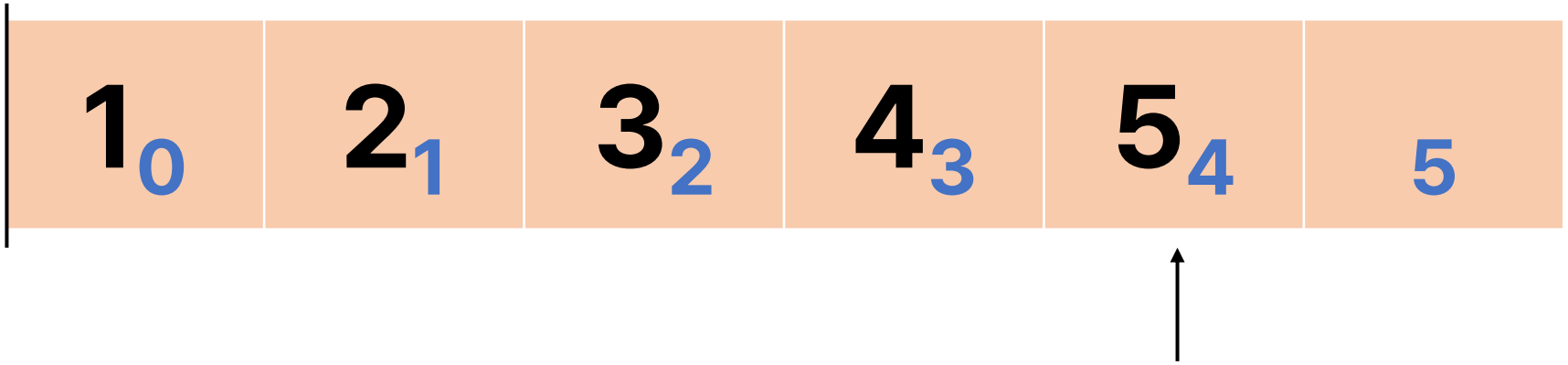
Stack 구현 – Push

- top+1에 데이터를 추가한다(arr[top + 1] = data)
- 데이터를 넣은 이후 top 변수를 갱신한다(top++)



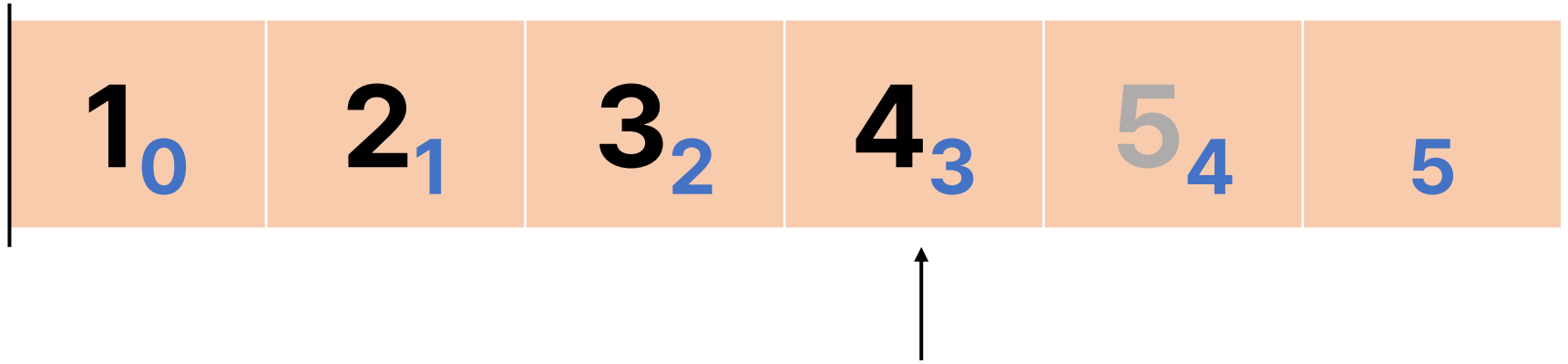
Stack 구현 – Pop

- 데이터가 새롭게 들어갈 때 기존 공간의 데이터를 덮어씌우므로, 지울 공간의 데이터를 특정 값으로 바꿀 필요는 없다
- 데이터를 빼기 전 데이터가 남아있는지 확인을 해야한다



Stack 구현 – Pop

- top 변수를 갱신한다(top--)



Assert

- 프로그램이 정상 작동하는지 알기 위해서는 디버깅이 필요
- 하지만 모든 값을 하나하나 살펴보기에는 너무 시간과 자원이 소모됨
- 잘못된 값이 나온다면 알려줄 경고문이 필요

- assert를 사용해보자

Assert

- 함수를 실행시킨 결과 값이 항상 0 또는 양수가 나와야 한다
- 음수 값이 나온 경우 조건문(if)를 이용해 오류가 나왔음을 확인할 수 있음
- 백준에서는 어떤 값이 어디에서 잘못 나왔는지 알 수 없음

- assert를 이용하면 런타임 에러를 발생시켜 어느 부분에서 문제가 발생했는지 확인

Assert 사용법

- #include <cassert>
- assert(조건문)

- 조건문을 만족하지 않으면 assert에서 런타임 에러를 발생시킴
- 함수를 실행시킨 값이 항상 0 이상이어야 한다면 assert(0 <= func()); 으로 잘못된 값이 발생하는지 확인이 가능하다

Stack 구현

```
#define MAX_SIZE 400000

struct Stack {
    int data[MAX_SIZE];
    int top;
    Stack() : data(), top(0) {}
};
```

Stack 구현

```
struct Stack {
    int top() {
        assert(0 < top and top <= MAX_SIZE);
        return data[top - 1];
    }
};
```

Stack 구현

```
struct Stack {
    void push(int x) {
        assert(0 <= top and top < MAX_SIZE);
        data[top] = x;
        top++;
        return;
    }
};
```

Stack 구현

```
struct Stack {
    void pop() {
        assert(0 < top and top <= MAX_SIZE);
        top--;
    }
};
```

Stack 구현

```
struct Stack {
    int size() { return top; }

    bool empty() { return top == 0; }
};
```

STL Stack

- #include <stack>
- stack<자료형> 변수명;

STL Stack

- void stack.push(x): 스택에 x를 삽입
- void stack.pop(): 스택에서 원소를 하나 제거, 스택이 비어있으면 런타임 에러
- 자료형 stack.top(x): 스택의 제일 위의 원소를 리턴, 스택이 비어있으면 런타임 에러
- bool stack.empty(x): 스택이 비어있으면 true, 아니면 false
- size_type stack.size(x): 스택의 크기를 리턴

Stack은 벡터로 대체할 수 있다

- 벡터에는 스택의 기능을 하는 함수가 모두 존재한다
- vector.push_back(x): 벡터의 제일 뒤에 x를 추가
- vector.pop_back(): 벡터의 제일 뒤의 원소를 제거, 벡터가 비어있으면 런타임 에러
- vector.back(): 벡터의 제일 뒤의 원소를 리턴, 벡터가 비어있으면 런타임 에러
- vector.empty(): 벡터가 비어있으면 true, 아니면 false
- vector.size(): 벡터의 크기를 리턴

Stack은 벡터로 대체할 수 있다

- 굳이 스택과 벡터를 따로 사용하지 않아도 벡터를 스택처럼 사용할 수 있다

Monotone Stack

- 스택에 담겨있는 원소들이 단조성을 띄는 경우, Monotone 스택이라고 한다
- 단조성을 띈다는 것은, 스택의 원소들이 증가하기만 하거나, 감소하기만 하는 경우를 의미
- 스택을 Monotone하게 만들면 가져올 수 있는 이점이 많다(문제가 쉽게 풀린다)
- ex) 5 4 3 2 1 또는 1 2 3 4 5는 Monotone한 경우이다.
- ex) 1 2 4 3 5는 Monotone하지 않다

문제

- 스택 2 BOJ 28278
- 에디터 BOJ 1406
- 단체줄넘기 BOJ 30457
- 오큰수 BOJ 17298

문제 풀이에 앞서

- 알고리즘을 적용시키는 눈을 기르자

- 지금 당장 문제를 풀지 못해도 된다
- 코드를 쓰는 능력은 정말 풀어본 문제의 수, 노력한 양에 비례
- 하지만 문제를 읽고 구상하는 것은 얼마든지 가능
- 문제를 풀지 못해도 어떤 방식으로 이 문제를 코딩할지 생각해보도록 하자

스택 2 BOJ 28278

- 스택 구현, 사용 문제
- std::stack 사용법 또는 스택 구현 및 사용을 연습해보자

에디터 BOJ 1406

- 16차시의 연결 리스트 문제, 어떻게 스택으로 해결이 가능할까?

에디터 BOJ 1406

- 커서의 왼쪽만 생각해보자.
- 일반적으로 글자는 왼쪽에서 오른쪽으로 쓴다
- 왼쪽에 있는 글자(커서에서 멀리 있는 쪽)는 오래된 글자
- 오른쪽에 있는 글자(커서에서 가까이 있는 쪽)는 최신 글자이다

에디터 BOJ 1406

- 커서를 왼쪽으로 옮긴다면, 가장 마지막에 들어간 데이터를 빼내야 하므로, 스택과 동일하다는 것을 알 수 있다
- 이를 이용해 커서를 왼쪽으로 옮기는 것을 커서 왼쪽에 있는 모든 글자를 스택에 넣어두고 스택에서 빼는 것으로 나타낼 수 있다

에디터 BOJ 1406

- 커서를 왼쪽으로 옮기는 것은 글자 입장에서는 커서의 왼쪽에 있던 글자가 오른쪽으로 이동하는 것
- 앞선 설명에서 이 내용을 스택에서 글자가 빠지는 것으로 구현할 수 있었다
- 반대로 커서를 오른쪽으로 이동하는 것은 제일 마지막에 뺐던 글자를 다시 가져오는 것
- 마지막에 다룬 데이터를 가져오므로 이 또한 스택과 일치한다는 것을 알 수 있다

에디터 BOJ 1406

- 이 둘을 조합하면 커서 기준으로 왼쪽과 오른쪽에 존재하는 글자를 스택으로 각각 관리 하면 커서를 왼쪽과 오른쪽으로 이동하는 것을 관리 할 수 있다

에디터 BOJ 1406

- abcd를 처음에 넣었다 생각하자
- 이는 왼쪽 스택에 a, b, c, d가 들어간 것

에디터 BOJ 1406

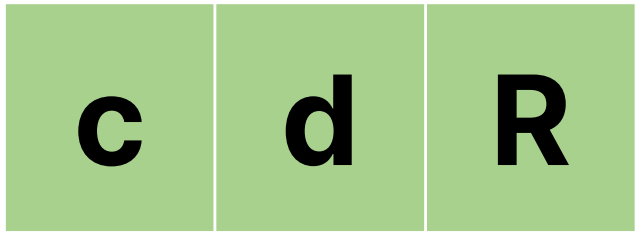
- 커서를 왼쪽으로 옮기는 것은 왼쪽 스택에서 빼서 오른쪽 스택으로 옮기는 것과 동일
- 커서를 오른쪽으로 옮기는 것은 오른쪽 스택에서 빼서 왼쪽 스택으로 옮기는 것과 동일

에디터 BOJ 1406

- 커서 왼쪽 이동

에디터 BOJ 1406

- 커서 왼쪽 이동



에디터 BOJ 1406

- 커서 왼쪽 이동

에디터 BOJ 1406

- 커서 오른쪽 이동

에디터 BOJ 1406

- 커서 오른쪽 이동

에디터 BOJ 1406

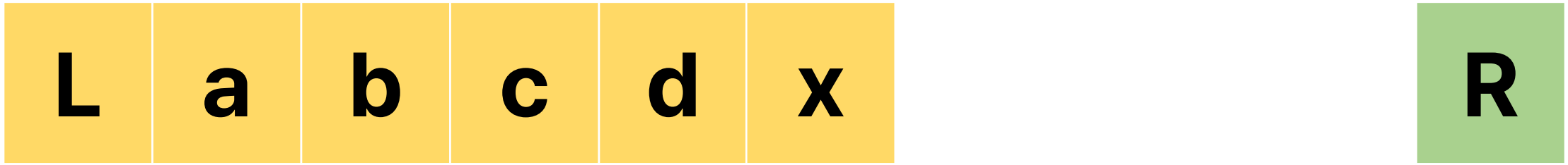
- 커서 오른쪽 이동
- 이동하려 할 때, 빼낼 스택이 비어있다면, 커서가 끝에 도달해 더 이상 움직이지 못하는 것을 의미

에디터 BOJ 1406

- 새로운 글자를 추가하는 것은 왼쪽 스택에 추가하는 것과 동일
- 글자를 제거하는 것은 왼쪽 스택에서 오른쪽 스택으로 이동하지 않고 제거하는 것과 동일

에디터 BOJ 1406

- x 삽입



에디터 BOJ 1406

- L 연산

에디터 BOJ 1406

- y 삽입

에디터 BOJ 1406

- 남은 글자는 왼쪽에서부터 읽으면 된다
- abcdyx

단체줄넘기 BOJ 30457

- N명의 학생들을 줄 세워야 한다
- 학생들은 왼쪽 또는 오른쪽을 보고 있음
- 모든 학생들은 자신이 바라보는 방향에 자기 보다 작은 학생들만 존재해야 한다
- $N \leq 1\,000$

단체줄넘기 BOJ 30457

- 내가 바라보는 방향에 나보다 작은 학생들만 있으면 되므로 키가 제일 큰 학생은 어떠한 제약도 존재하지 않는다
- 제일 큰 학생을 넣은 후, 학생을 한 명씩 넣어보자
- 제일 키가 큰 학생을 넣은 후 다른 학생들은 모두 제일 키가 큰 학생을 등지고 서야 한다

  제일 키가 큰 학생을 바라보는 경우, 나보다 키가 큰 학생이 존재하게 되므로

단체줄넘기 BOJ 30457

- 제일 키가 큰 학생을 기준으로 양 옆에 서로 양 끝을 바라보는 학생들이 반드시 오게 된다
- 그 이후에는 한 쪽 끝을 바라보고 있는 학생 앞에는 그 학생보다 키가 작은 학생이 와야 하며, 그 학생 또한 키가 큰 학생을 바라보지 못하므로, 같은 방향을 보게된다

- 계속하여 반복적으로 일어난다

단체줄넘기 BOJ 30457

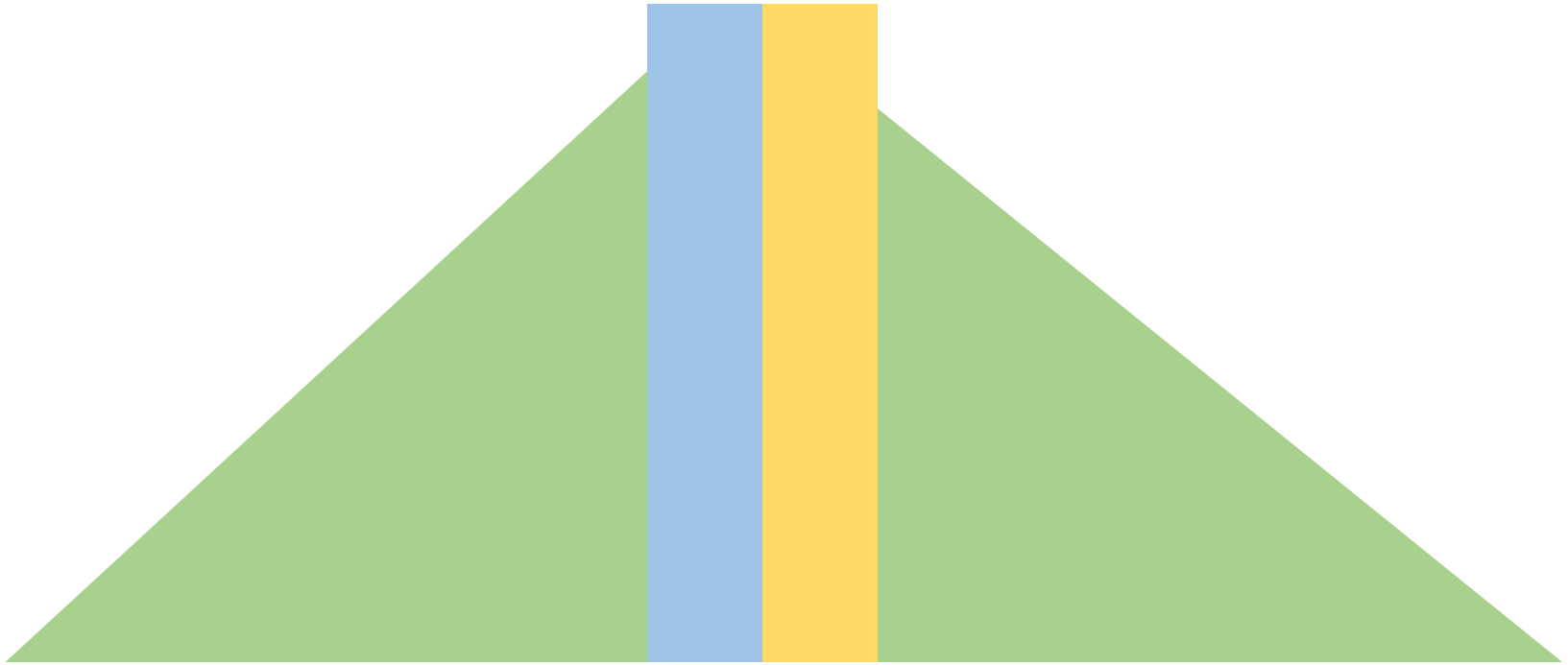
- 이를 이용하여 대략적인 모양이 삼각형으로 나오는 것을 알 수 있다

단체줄넘기 BOJ 30457

- 하지만 키가 동일한 학생도 존재한다
- 점차 키가 작은 학생이 들어가야 하지만 제일 키가 큰 학생은 한 쪽 방향만 바라보고 있다
- 키가 제일 큰 학생과 동일한 키를 가진 학생이 존재한다면, 두 학생이 서로 등지고 서게 할 수 있다

단체줄넘기 BOJ 30457

- 키가 제일 큰 친구를 두 명을 넣고 점차 작아지면 된다



단체줄넘기 BOJ 30457

- 제일 많은 학생을 넣어야 하므로 키가 제일 큰 학생부터 그 다음 큰 학생 순서로 순차적으로 넣어야 제일 많이 넣을 수 있다
- 정렬 후 키가 큰 친구부터 넣는다

단체줄넘기 BOJ 30457

- 큰 친구부터 넣으므로, 현재 넣을 학생의 키는 이전 학생의 키보다 작거나 동일하다
- 넣은 모든 학생과 비교할 필요 없이 마지막에 넣은 학생과 동일한지만 확인하면 된다
- 마지막에 넣은 데이터를 확인 -> 스택

단체줄넘기 BOJ 30457

- 왼쪽을 바라보는 학생들을 저장할 스택
- 오른쪽을 바라보는 학생들을 저장할 스택
- 이 두 개를 준비한다

단체줄넘기 BOJ 30457

- 학생들을 정렬한 후, 제일 키가 큰 학생 두 명을 각각 스택 하나에 한 명씩 넣는다
- 그리고 키가 큰 학생부터 살펴보며 왼쪽과 오른쪽 스택 중 한 곳에 넣는다.
- 두 스택 모두 들어갈 수 없는 경우 학생을 넣을 수 없는 경우이므로, 넣지 않는다

오큰수 BOJ 17298

- 수열이 주어질 때, k번째 오큰수는 수열의 k번째 숫자보다 오른쪽에 있으면서 k번째 숫자보다 큰 수 중에 제일 왼쪽에 있는 수를 찾는 것
- 그러한 수가 없는 경우 오큰수는 -1이다
- $N \leq 1\,000\,000$

오큰수 BOJ 17298

- 예제를 그려보자
- 예제 1

  3 5 2 7

  5 7 7 -1

오큰수 BOJ 17298

- 예제를 그려보자
- 예제 2

  9 5 4 8

  -1 8 8 -1

오큰수 BOJ 17298

- 오른쪽에서 나오는 수 중에서 제일 먼저 나오는 큰 수가 오큰수
- 오른쪽에서 나오므로 왼쪽부터 오른쪽으로 보자
- 내가 지금까지 나온 높이들을 기억하고 있다가 나보다 큰 수가 나오면 오큰수로 정하면 됨

오큰수 BOJ 17298

- 다시 정리하면
- 왼쪽부터 숫자를 보면서 기억함
- 나보다 큰 수가 오는 경우 오큰수가 확정됨
- 그 이후에 더 큰 수가 나오더라도 제일 왼쪽에 있는 수를 고르므로 영향을 주지 않음
- 나보다 작은 수가 나오는 경우 오큰수가 확정되지 않으므로 기다림

오큰수 BOJ 17298

- 3 5 2 7
- 3이 먼저 들어감

오큰수 BOJ 17298

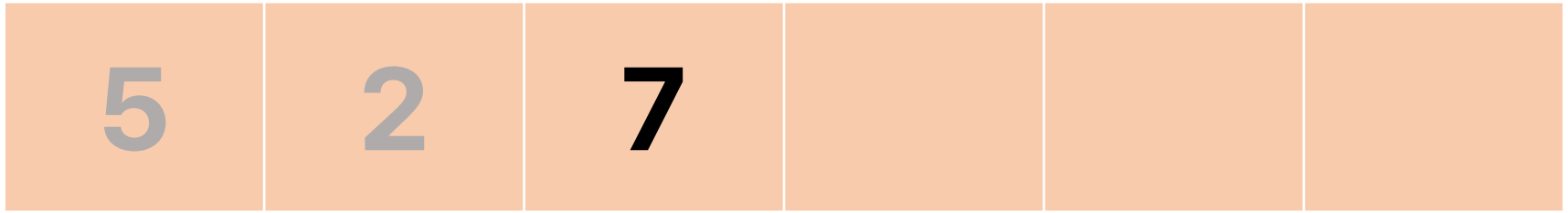
- 3 5 2 7
- 5가 들어가는 경우, 3보다 큰 수가 나오므로 3의 오큰수는 확정이 남

오큰수 BOJ 17298

- 3 5 2 7
- 2는 5보다 크지 않으므로 들어감

오큰수 BOJ 17298

- 3 5 2 7
- 7이 들어가는 경우, 7보다 작은 수들은 전부 오큰수가 정해지게 됨



오큰수 BOJ 17298

- 자료구조에는 작은 수가 나올때만 저장되고, 큰 수가 나오는 경우 다 나오게 된다
- Monotone 하게 관리됨
- 즉, 점차 작아지는 순서대로 자료구조에 저장되게 된다

오큰수 BOJ 17298

- 마지막에 저장된 수가 제일 작고 점차 커지므로, 마지막에 넣은 수부터 현재 수보다 같거나 큰 수가 나올때까지 자료구조에서 빼내가며 오큰수를 지정해주면 된다

- 스택을 사용하면 된다는 것을 의미