








자료구조 실습

10/01

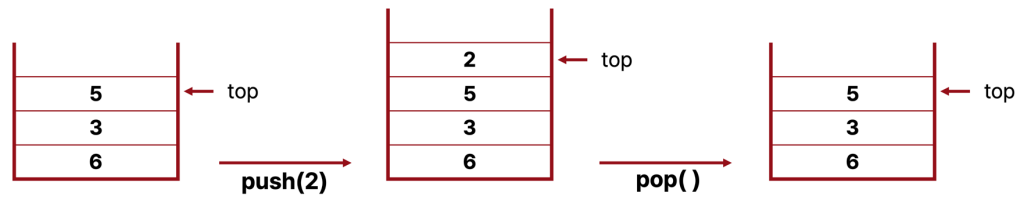




목차

-  • Stack
-  • is_matched
-  • Stack
-  • is_matched
-  • challenge

Stack



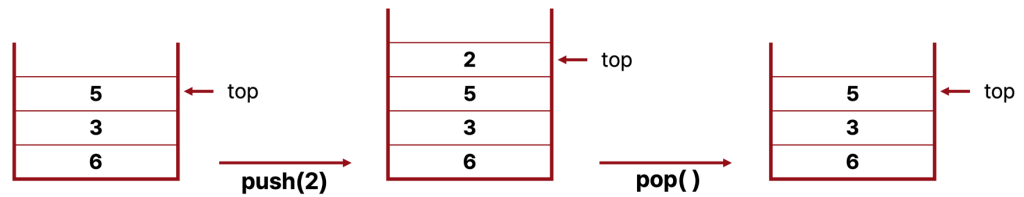
```
class Stack:
    def __init__(self):
        self._data = [] # list로 top은 맨 끝

    def print(self):
        print(self._data)

    def push(self, x):
        self._data.NEED_TO_SOLVE(NEED_TO_SOLVE)

    def pop(self):
        if not self._data:
            raise IndexError("pop from empty stack")
        return self._data.NEED_TO_SOLVE()
```

Stack



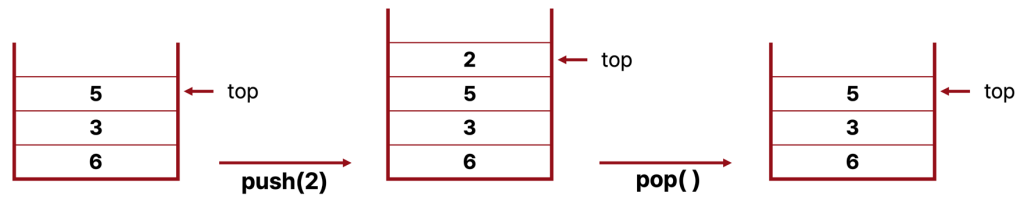
```
class Stack:
    def __init__(self):
        self._data = [] # list로 top은 맨 끝

    def print(self):
        print(self._data)

    def push(self, x):
        self._data.append(x)

    def pop(self):
        if not self._data:
            raise IndexError("pop from empty stack")
        return self._data.NEED_TO_SOLVE()
```

Stack



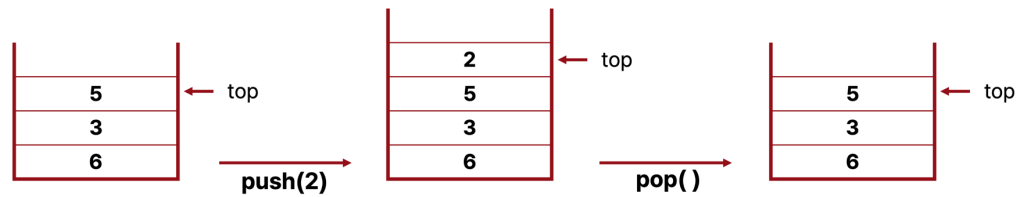
```
class Stack:
    def __init__(self):
        self._data = [] # list로 top은 맨 끝

    def print(self):
        print(self._data)

    def push(self, x):
        self._data.append(x)

    def pop(self):
        if not self._data:
            raise IndexError("pop from empty stack")
        return self._data.NEED_TO_SOLVE()
```

Stack



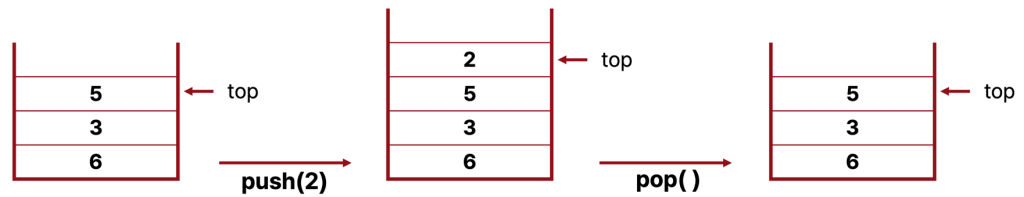
```
class Stack:
    def __init__(self):
        self._data = [] # list로 top은 맨 끝

    def print(self):
        print(self._data)

    def push(self, x):
        self._data.append(x)

    def pop(self):
        if not self._data:
            raise IndexError("pop from empty stack")
        return self._data.pop()
```

Stack



```
class Stack:
    def __init__(self):
        self._data = [] # list로 top은 맨 끝

    def print(self):
        print(self._data)

    def push(self, x):
        self._data.append(x)

    def pop(self):
        if not self._data:
            raise IndexError("pop from empty stack")
        return self._data.pop()

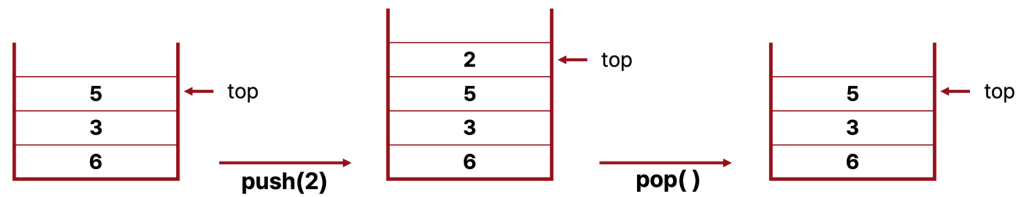
    def top(self):
        if not self._data:
            raise IndexError("top from empty stack")
        return self._data[-1]

    def is_empty(self):
        return len(self._data) == 0

    def size(self):
        return len(self._data)
```

is_matched

- {{()}} -> match
- {}()[] -> match
- [{()}] -> unmatched
- ([{}]() -> unmatched



```
def is_matched(expression: str) -> bool:
    pairs = {'(': ')', '[': ']', '{': '}'
    st = Stack() # 리스트 대신 우리가 만든 스택 사용

    for ch in expression:
        if ch in '([{':
            st.NEED_TO_SOLVE(ch)

        elif ch in ')]}':
            # 비었거나 top이 짝이 아니면 실패
            if st.NEED_TO_SOLVE() or st.NEED_TO_SOLVE() != pairs[ch]:
                return NEED_TO_SOLVE

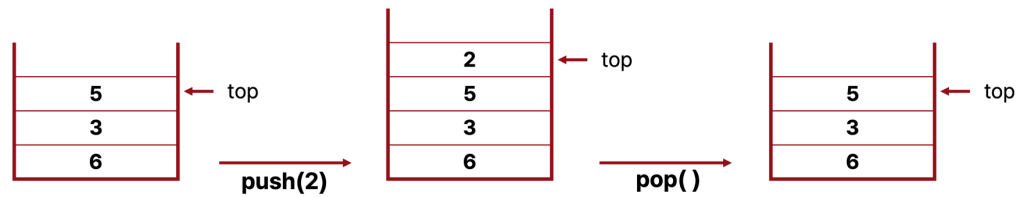
            st.NEED_TO_SOLVE()

        else:
            # 기타 문자는 무시
            pass

    # 모두 처리 후 비어 있어야 완전 매칭
    return st.NEED_TO_SOLVE()
```


is_matched

- {{()}} -> match
- {}()[] -> match
- [{()}] -> unmatched
- ([{}]) -> unmatched



```
def is_matched(expression: str) -> bool:
    pairs = {'(': ')', '[': ']', '{': '}'
    st = Stack() # 리스트 대신 우리가 만든 스택 사용

    for ch in expression:
        if ch in '([{':
            st.push(ch)

        elif ch in ')]}':
            # 비었거나 top이 짝이 아니면 실패
            if st.NEED_TO_SOLVE() or st.NEED_TO_SOLVE() != pairs[ch]:
                return NEED_TO_SOLVE

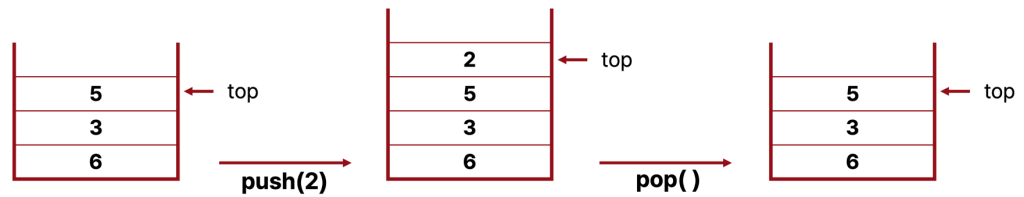
            st.NEED_TO_SOLVE()

        else:
            # 기타 문자는 무시
            pass

    # 모두 처리 후 비어 있어야 완전 매칭
    return st.NEED_TO_SOLVE()
```

is_matched

- {{()}} -> match
- {}()[] -> match
- [{()}] -> unmatched
- ([{}]() -> unmatched



```
def is_matched(expression: str) -> bool:
    pairs = {'(': ')', '[': ']', '{': '}'
    st = Stack() # 리스트 대신 우리가 만든 스택 사용

    for ch in expression:
        if ch in '([{':
            st.push(ch)

        elif ch in ')]}':
            # 비었거나 top이 짝이 아니면 실패
            if st.NEED_TO_SOLVE() or st.NEED_TO_SOLVE() != pairs[ch]:
                return NEED_TO_SOLVE

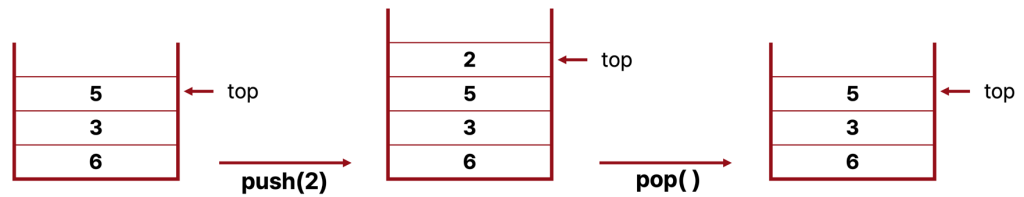
            st.NEED_TO_SOLVE()

        else:
            # 기타 문자는 무시
            pass

    # 모두 처리 후 비어 있어야 완전 매칭
    return st.NEED_TO_SOLVE()
```

is_matched

- {{()}} -> match
- {}()[] -> match
- [{()}] -> unmatched
- ([{}]() -> unmatched



```
def is_matched(expression: str) -> bool:
    pairs = {'(': ')', '[': ']', '{': '}'
    st = Stack() # 리스트 대신 우리가 만든 스택 사용

    for ch in expression:
        if ch in '([{':
            st.push(ch)

        elif ch in ')]}':
            # 비었거나 top이 짝이 아니면 실패
            if st.is_empty() or st.top() != pairs[ch]:
                return False

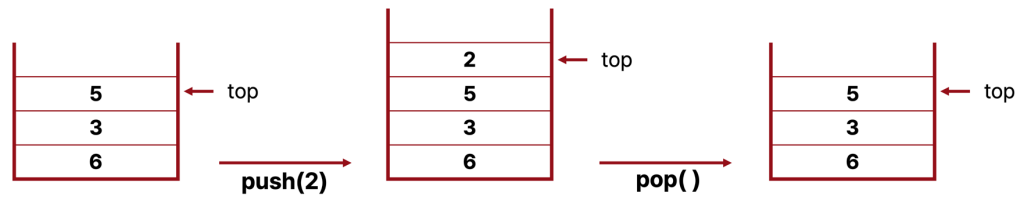
            st.pop()

        else:
            # 기타 문자는 무시
            pass

    # 모두 처리 후 비어 있어야 완전 매칭
    return st.NEED_TO_SOLVE()
```

is_matched

- {{()}} -> match
- {}()[] -> match
- [{()}] -> unmatched
- ([{}]) -> unmatched



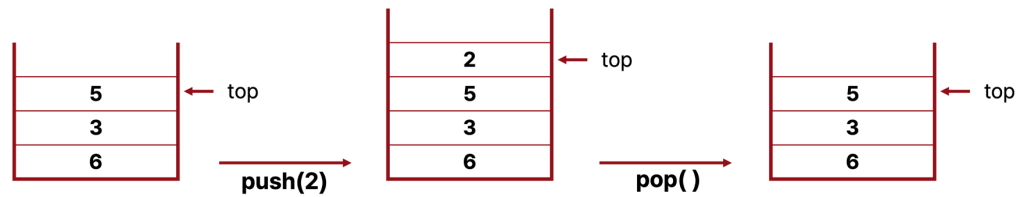
```
def is_matched(expression: str) -> bool:
    pairs = {'(': ')', '[': ']', '{': '}'
    st = Stack() # 리스트 대신 우리가 만든 스택 사용

    for ch in expression:
        if ch in '([{':
            st.push(ch)
        elif ch in ')]}':
            # 비었거나 top이 짝이 아니면 실패
            if st.is_empty() or st.top() != pairs[ch]:
                return False
            st.pop()
        else:
            # 기타 문자는 무시
            pass

    # 모두 처리 후 비어 있어야 완전 매칭
    return st.NEED_TO_SOLVE()
```

is_matched

- {{()}} -> match
- {}()[] -> match
- [{()}] -> unmatched
- ([{}]) -> unmatched

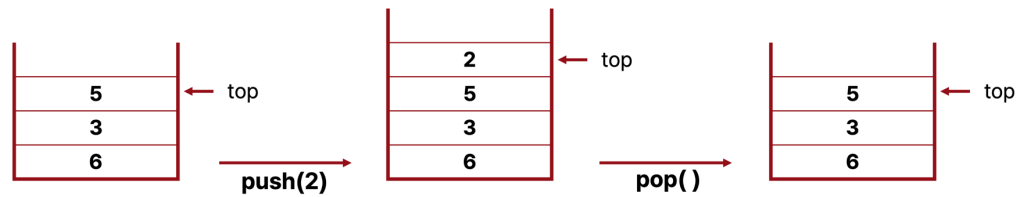


```
def is_matched(expression: str) -> bool:
    pairs = {'(': ')', '[': ']', '{': '}'
    st = Stack() # 리스트 대신 우리가 만든 스택 사용

    for ch in expression:
        if ch in '([{':
            st.push(ch)
        elif ch in ')]}':
            # 비었거나 top이 짝이 아니면 실패
            if st.is_empty() or st.top() != pairs[ch]:
                return False
            st.pop()
        else:
            # 기타 문자는 무시
            pass

    # 모두 처리 후 비어 있어야 완전 매칭
    return st.is_empty()
```

실행



```
if __name__ == "__main__":
    stack = Stack()
    for v in [7, 3, 6]:
        stack.push(v)
        stack.print()
    print('='*20)

    print(stack.pop()) # 6
    stack.print()
    print('='*20)

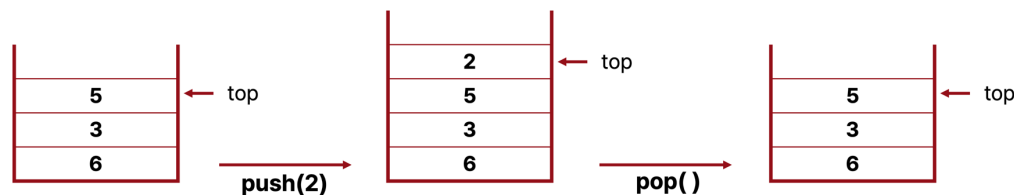
    stack.push(4)
    stack.print()
    print('='*20)

    while not stack.is_empty():
        print(stack.pop())
        stack.print()
    print('='*20)

    tests = [
        "()[]{}", "([{}])", "([])", "((((()))))", "([]",
        "{[()] }a+b*(c-d)",
    ]
    for t in tests:
        print(t, "=>", "MATCH" if is_matched(t) else "UNMATCH")
        print()
```

```
(base) minseo@minseo-MacBookAir-2 1001-stack % python answer_official.py
[7]
[7, 3]
[7, 3, 6]
=====
6
[7, 3]
=====
[7, 3, 4]
=====
4
[7, 3]
3
[7]
7
[]
=====
()[]{} => MATCH
([{}]) => MATCH
([]) => UNMATCH
((((())))) => MATCH
([] => UNMATCH
{[()] }a+b*(c-d) => MATCH
```

실행



```
if __name__ == "__main__":
    stack = Stack()
    for v in [7, 3, 6]:
        stack.push(v)
        stack.print()
    print('='*20)

    print(stack.pop()) # 6
    stack.print()
    print('='*20)

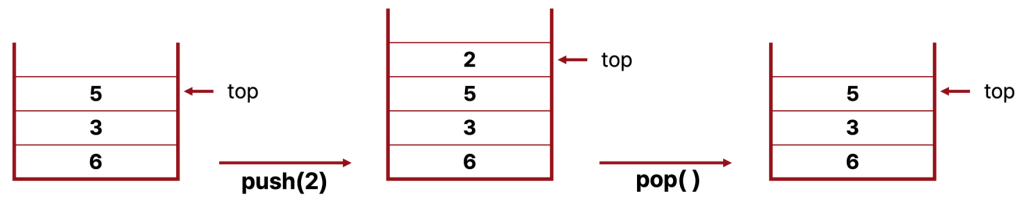
    stack.push(4)
    stack.print()
    print('='*20)

    while not stack.is_empty():
        print(stack.pop())
        stack.print()
    print('='*20)

    tests = [
        "()[]{}", "([{}])", "([])", "((((()))))", "([]",
        "{[()] }a+b*(c-d)",
    ]
    for t in tests:
        print(t, "=>", "MATCH" if is_matched(t) else "UNMATCH")
        print()
```

```
(base) minseo@minseo-MacBookAir-2 1001-stack % python answer_official.py
[7]
[7, 3]
[7, 3, 6]
=====
6
[7, 3]
=====
[7, 3, 4]
=====
4
[7, 3]
3
[7]
7
[]
=====
()[]{} => MATCH
([{}]) => MATCH
([]) => UNMATCH
((((())))) => MATCH
([] => UNMATCH
{[()] }a+b*(c-d) => MATCH
```

실행



```
if __name__ == "__main__":
    stack = Stack()
    for v in [7, 3, 6]:
        stack.push(v)
        stack.print()
    print('='*20)

    print(stack.pop()) # 6
    stack.print()
    print('='*20)

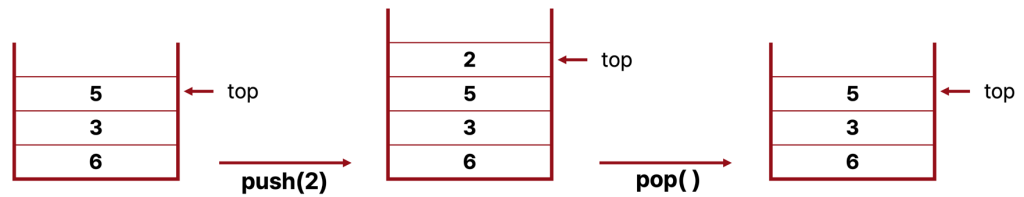
    stack.push(4)
    stack.print()
    print('='*20)

    while not stack.is_empty():
        print(stack.pop())
        stack.print()
    print('='*20)

    tests = [
        "()[]{}", "([{}])", "([])", "((((()))))", "([)",
        "{[()] }a+b*(c-d)",
    ]
    for t in tests:
        print(t, "=>", "MATCH" if is_matched(t) else "UNMATCH")
        print()
```

```
(base) minseo@minseo-MacBookAir-2 1001-stack % python answer_official.py
[7]
[7, 3]
[7, 3, 6]
=====
6
[7, 3]
=====
[7, 3, 4]
=====
4
[7, 3]
3
[7]
7
[]
=====
()[]{} => MATCH
([{}]) => MATCH
([]) => UNMATCH
((((())))) => MATCH
([) => UNMATCH
{[()] }a+b*(c-d) => MATCH
```


실행



```
if __name__ == "__main__":
    stack = Stack()
    for v in [7, 3, 6]:
        stack.push(v)
        stack.print()
    print('='*20)

    print(stack.pop()) # 6
    stack.print()
    print('='*20)

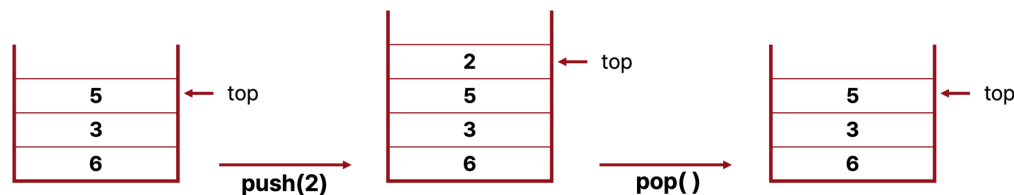
    stack.push(4)
    stack.print()
    print('='*20)

    while not stack.is_empty():
        print(stack.pop())
        stack.print()
    print('='*20)

    tests = [
        "()[]{}", "([{}])", "([])", "((((()))))", "([]",
        "{[()] }a+b*(c-d)",
    ]
    for t in tests:
        print(t, "=>", "MATCH" if is_matched(t) else "UNMATCH")
        print()
```

```
(base) minseo@minseo-MacBookAir-2 1001-stack % python answer_official.py
[7]
[7, 3]
[7, 3, 6]
=====
6
[7, 3]
=====
[7, 3, 4]
=====
4
[7, 3]
3
[7]
7
[]
=====
()[]{} => MATCH
([{}]) => MATCH
([]) => UNMATCH
((((())))) => MATCH
([] => UNMATCH
{[()] }a+b*(c-d) => MATCH
```

실행



```
if __name__ == "__main__":
    stack = Stack()
    for v in [7, 3, 6]:
        stack.push(v)
        stack.print()
    print('='*20)

    print(stack.pop()) # 6
    stack.print()
    print('='*20)

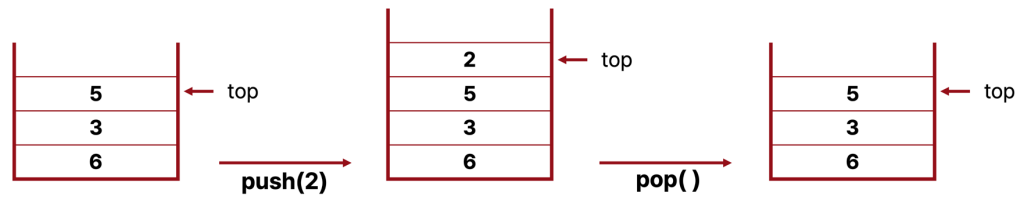
    stack.push(4)
    stack.print()
    print('='*20)

    while not stack.is_empty():
        print(stack.pop())
        stack.print()
    print('='*20)

    tests = [
        "()[]{}", "([{}])", "([])", "((((()))))", "([]",
        "{[()] }a+b*(c-d)",
    ]
    for t in tests:
        print(t, "=>", "MATCH" if is_matched(t) else "UNMATCH")
        print()
```

```
(base) minseo@minseo-MacBookAir-2 1001-stack % python answer_official.py
[7]
[7, 3]
[7, 3, 6]
=====
6
[7, 3]
=====
[7, 3, 4]
=====
4
[7, 3]
3
[7]
7
[]
=====
()[]{} => MATCH
([{}]) => MATCH
([]) => UNMATCH
((((())))) => MATCH
([] => UNMATCH
{[()] }a+b*(c-d) => MATCH
```

Stack



```
// 생성자: top 초기화
StackType::StackType() {
    top = -1;
}
```

```
// push
void StackType::push(ItemType value) {
    if (NEED_TO_SOLVE()) {
        throw std::overflow_error("Stack overflow");
    }
    data[NEED_TO_SOLVE] = value;
}
```

```
// pop
ItemType StackType::pop() {
    if (NEED_TO_SOLVE()) {
        throw std::underflow_error("Stack underflow");
    }
    return data[NEED_TO_SOLVE];
}
```

```
// 객체 생성 시 호출. data 배열은 이미 객체 내부에 자리 잡음(추가 할당 없음).
// 논리적 크기를 -1로 설정 (비어있음을 의미). 데이터 영역은 초기화하지 않아도 됨.
```

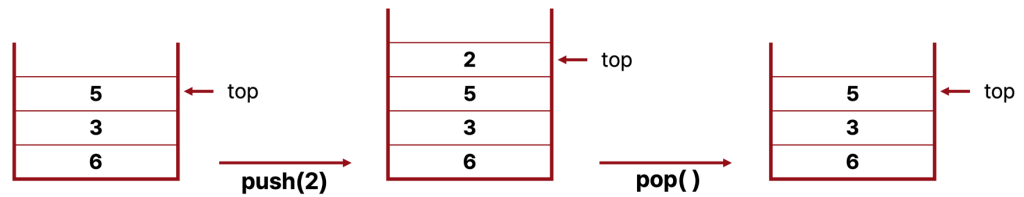
```
// 경계 검사: top가 마지막 인덱스에 도달했다면 더 이상 쓸 공간 없음(오버플로 방지).
// 힙/스택 메모리 직접 사용 없이 예외 객체를 임시 생성해 던짐(런타임 시 스택/힙 내부 처리).
```

```
// top을 1 증가시킨 뒤, 증가된 인덱스에 value를 '직접 대입'.
// 이 순간 data 배열의 해당 슬롯(이미 객체 내부에 존재하는 메모리)에 값이 저장됨.
```

```
// 경계 검사: 비어 있으면 읽을 유효 데이터가 없음(언더플로 방지).
// 추가 메모리 할당 없이 오류 전파(런타임 시스템이 예외 처리).
```

```
// 현재 top 위치의 값을 '복사'해서 반환하고, top을 1 감소.
// 메모리상 원본은 남아있지만(top만 줄임), 논리적으로는 삭제된 것으로 간주.
```

Stack



```
// 생성자: top 초기화
StackType::StackType() {
    top = -1;
}
```

```
// push
void StackType::push(ItemType value) {
    if (isFull()) {
        throw std::overflow_error("Stack overflow");
    }
    data[++top] = value;
}
```

```
// pop
ItemType StackType::pop() {
    if (NEED_TO_SOLVE()) {
        throw std::underflow_error("Stack underflow");
    }
    return data[NEED_TO_SOLVE];
}
```

```
// 객체 생성 시 호출. data 배열은 이미 객체 내부에 자리 잡음(추가 할당 없음).
// 논리적 크기를 -1로 설정 (비어있음을 의미). 데이터 영역은 초기화하지 않아도 됨.
```

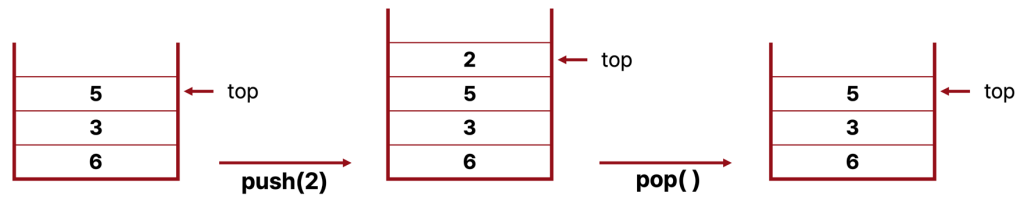
```
// 경계 검사: top가 마지막 인덱스에 도달했다면 더 이상 쓸 공간 없음(오버플로 방지).
// 힙/스택 메모리 직접 사용 없이 예외 객체를 임시 생성해 던짐(런타임 시 스택/힙 내부 처리).
```

```
// top을 1 증가시킨 뒤, 증가된 인덱스에 value를 '직접 대입'.
// 이 순간 data 배열의 해당 슬롯(이미 객체 내부에 존재하는 메모리)에 값이 저장됨.
```

```
// 경계 검사: 비어 있으면 읽을 유효 데이터가 없음(언더플로 방지).
// 추가 메모리 할당 없이 오류 전파(런타임 시스템이 예외 처리).
```

```
// 현재 top 위치의 값을 '복사'해서 반환하고, top을 1 감소.
// 메모리상 원본은 남아있지만(top만 줄임), 논리적으로는 삭제된 것으로 간주.
```

Stack



```
// 생성자: top 초기화
StackType::StackType() {
    top = -1;
}

// push
void StackType::push(ItemType value) {
    if (isFull()) {
        throw std::overflow_error("Stack overflow");
    }
    data[++top] = value;
}

// pop
ItemType StackType::pop() {
    if (NEED_TO_SOLVE()) {
        throw std::underflow_error("Stack underflow");
    }
    return data[NEED_TO_SOLVE];
}
```

// 객체 생성 시 호출. data 배열은 이미 객체 내부에 자리 잡음(추가 할당 없음).
// 논리적 크기를 -1로 설정 (비어있음을 의미). 데이터 영역은 초기화하지 않아도 됨.

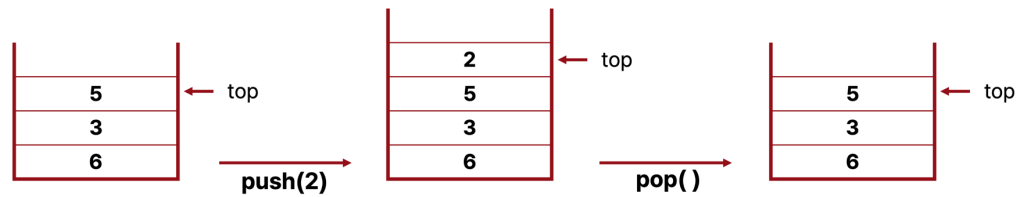
// 경계 검사: top가 마지막 인덱스에 도달했다면 더 이상 쓸 공간 없음(오버플로 방지).
// 힙/스택 메모리 직접 사용 없이 예외 객체를 임시 생성해 던짐(런타임 시 스택/힙 내부 처리).

// top을 1 증가시킨 뒤, 증가된 인덱스에 value를 '직접 대입'.
// 이 순간 data 배열의 해당 슬롯(이미 객체 내부에 존재하는 메모리)에 값이 저장됨.

// 경계 검사: 비어 있으면 읽을 유효 데이터가 없음(언더플로 방지).
// 추가 메모리 할당 없이 오류 전파(런타임 시스템이 예외 처리).

// 현재 top 위치의 값을 '복사'해서 반환하고, top을 1 감소.
// 메모리상 원본은 남아있지만(top만 줄임), 논리적으로는 삭제된 것으로 간주.

Stack



```
// 생성자: top 초기화
StackType::StackType() {
    top = -1;
}

// push
void StackType::push(ItemType value) {
    if (isFull()) {
        throw std::overflow_error("Stack overflow");
    }
    data[++top] = value;
}

// pop
ItemType StackType::pop() {
    if (isEmpty()) {
        throw std::underflow_error("Stack underflow");
    }
    return data[top--];
}
```

// 객체 생성 시 호출. data 배열은 이미 객체 내부에 자리 잡음(추가 할당 없음).
// 논리적 크기를 -1로 설정 (비어있음을 의미). 데이터 영역은 초기화하지 않아도 됨.

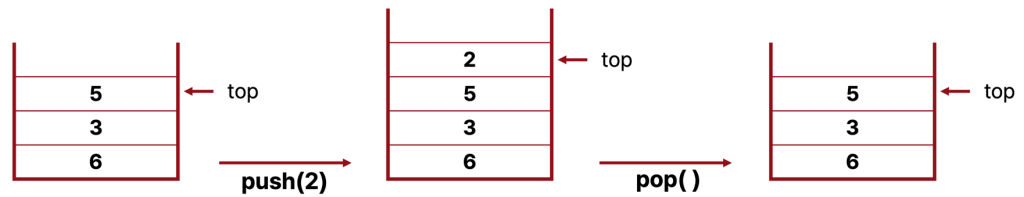
// 경계 검사: top가 마지막 인덱스에 도달했다면 더 이상 쓸 공간 없음(오버플로 방지).
// 힙/스택 메모리 직접 사용 없이 예외 객체를 임시 생성해 던짐(런타임 시 스택/힙 내부 처리).

// top을 1 증가시킨 뒤, 증가된 인덱스에 value를 '직접 대입'.
// 이 순간 data 배열의 해당 슬롯(이미 객체 내부에 존재하는 메모리)에 값이 저장됨.

// 경계 검사: 비어 있으면 읽을 유효 데이터가 없음(언더플로 방지).
// 추가 메모리 할당 없이 오류 전파(런타임 시스템이 예외 처리).

// 현재 top 위치의 값을 '복사'해서 반환하고, top을 1 감소.
// 메모리상 원본은 남아있지만(top만 줄임), 논리적으로는 삭제된 것으로 간주.

Stack



```
// 현재 크기  
int StackType::size() const {  
    return NEED_TO_SOLVE;  
}
```

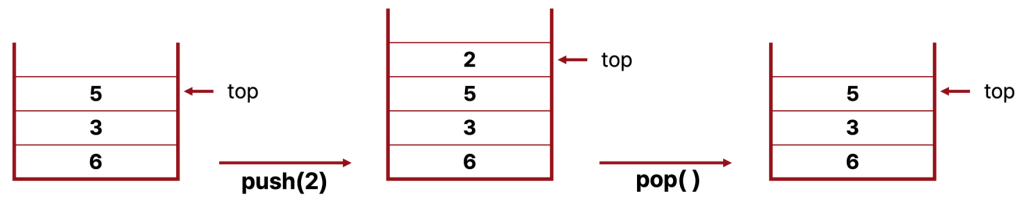
```
// 상태 검사  
bool StackType::isFull() const {  
    return top >= MAX_SIZE - 1;  
}  
  
bool StackType::isEmpty() const {  
    return top < 0;  
}
```

```
// data 배열의 실제 사용량(논리적 크기)을 계산해 돌려줌.  
// top=-1이면 0, top=0이면 1 ... (data의 물리 메모리는 그대로 유지됨).
```

```
// 배열 경계 상한 검사. MAX_SIZE는 컴파일타임 상수.  
// top가 마지막 슬롯에 도달했으면 true. 새로 쓸 수 있는 여유 메모리가 없음.
```

```
// 비어있는지 검사.  
// top==-1이면 비어있음. 메모리를 해제하는 개념이 아니라 '참조하지 않기'로 관리.
```

Stack



```
// 현재 크기  
int StackType::size() const {  
    return top + 1;  
}
```

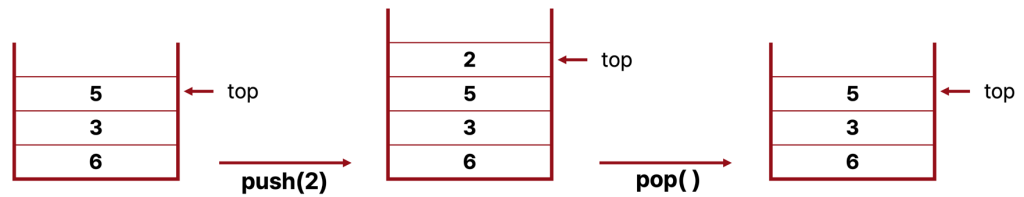
```
// 상태 검사  
bool StackType::isFull() const {  
    return top >= MAX_SIZE - 1;  
}  
  
bool StackType::isEmpty() const {  
    return top < 0;  
}
```

```
// data 배열의 실제 사용량(논리적 크기)을 계산해 돌려줌.  
// top=-1이면 0, top=0이면 1 ... (data의 물리 메모리는 그대로 유지됨).
```

```
// 배열 경계 상한 검사. MAX_SIZE는 컴파일타임 상수.  
// top가 마지막 슬롯에 도달했으면 true. 새로 쓸 수 있는 여유 메모리가 없음.
```

```
// 비어있는지 검사.  
// top==-1이면 비어있음. 메모리를 해제하는 개념이 아니라 '참조하지 않기'로 관리.
```


is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

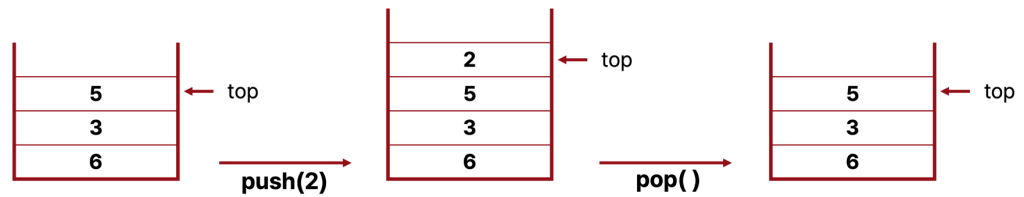
    for (char ch : expression) {                // 열린 괄호라면
        if (ch == '(' || ch == '[' || ch == '{') { // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.
            st.NEED_TO_SOLVE(ch);
        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.NEED_TO_SOLVE()) return false; // 닫을 대상이 없으면 불일치 → 바로 false.

            char open = st.NEED_TO_SOLVE(); // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') || // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return NEED_TO_SOLVE; // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }

    return st.NEED_TO_SOLVE(); // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') { // 열린 괄호라면
            st.push(ch);                          // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.

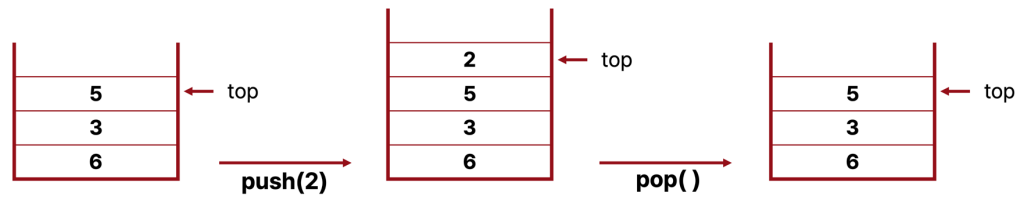
        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.NEED_TO_SOLVE()) return false; // 닫을 대상이 없으면 불일치 → 바로 false.

            char open = st.NEED_TO_SOLVE();        // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') ||      // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return NEED_TO_SOLVE;             // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }

    return st.NEED_TO_SOLVE();                    // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

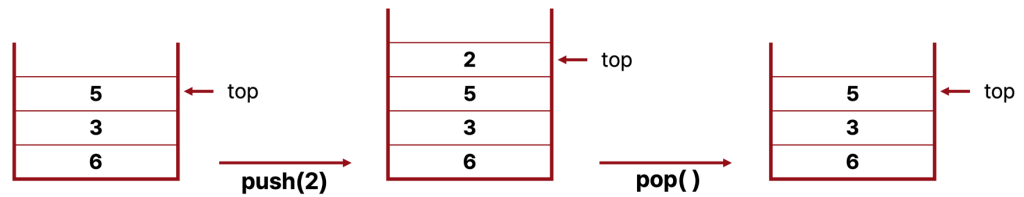
is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') { // 열린 괄호라면
            st.push(ch);                          // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.
        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.isEmpty()) return false;        // 닫을 대상이 없으면 불일치 → 바로 false.
            char open = st.NEED_TO_SOLVE();        // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).
            if ((open == '(' && ch != ')') ||      // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return NEED_TO_SOLVE;             // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }
    return st.NEED_TO_SOLVE();                   // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') { // 열린 괄호라면
            st.push(ch);                          // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.

        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.isEmpty()) return false;        // 닫을 대상이 없으면 불일치 → 바로 false.

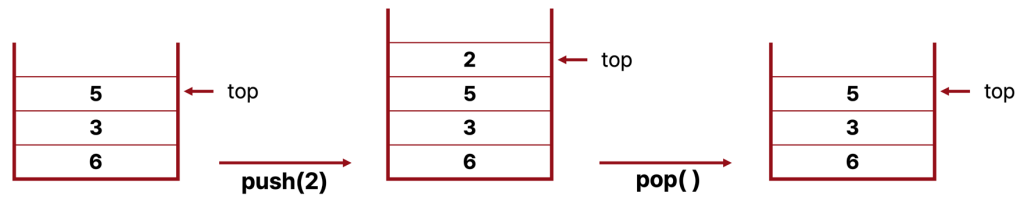
            char open = st.NEED_TO_SOLVE();        // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') ||      // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return NEED_TO_SOLVE;             // 짝이 틀리면 false.
            }

            // 다른 문자는 무시
        }
    }

    return st.NEED_TO_SOLVE();                    // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') {    // 열린 괄호라면
            st.push(ch);                            // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.

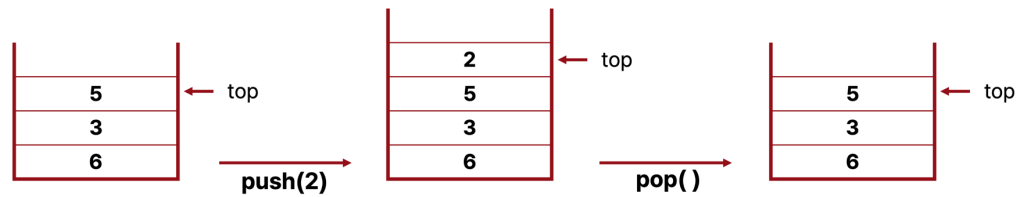
        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.isEmpty()) return false;          // 닫을 대상이 없으면 불일치 → 바로 false.

            char open = st.pop();                    // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') ||          // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return NEED_TO_SOLVE;                // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }

    return st.NEED_TO_SOLVE();                      // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                    // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') {    // 열린 괄호라면
            st.push(ch);                            // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.

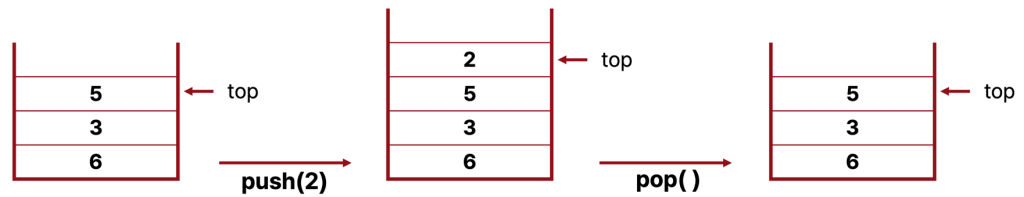
        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.isEmpty()) return false;          // 닫을 대상이 없으면 불일치 → 바로 false.

            char open = st.pop();                    // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') ||        // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return NEED_TO_SOLVE;                // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }

    return st.NEED_TO_SOLVE();                      // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                    // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') {    // 열린 괄호라면
            st.push(ch);                            // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.

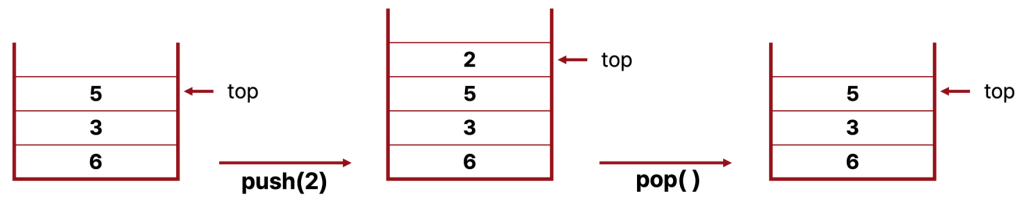
        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.isEmpty()) return false;          // 닫을 대상이 없으면 불일치 → 바로 false.

            char open = st.pop();                    // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') ||          // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return false;                        // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }

    return st.NEED_TO_SOLVE();                      // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                    // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

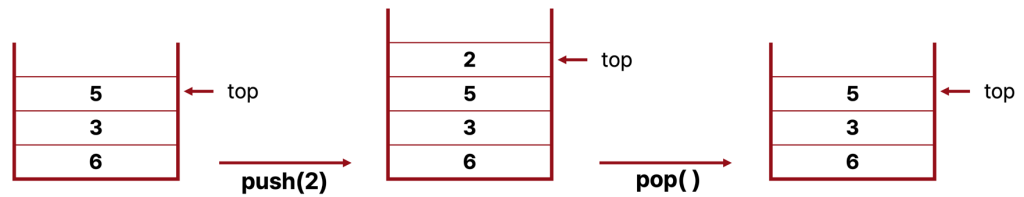
    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') {    // 열린 괄호라면
            st.push(ch);                            // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.

        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.isEmpty()) return false;          // 닫을 대상이 없으면 불일치 → 바로 false.

            char open = st.pop();                    // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') ||          // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return false;                        // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }
    return st.NEED_TO_SOLVE();                      // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                    // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```


is_matched



```
bool is_matched(const std::string &expression) {
    StackType st;                                // StackType 객체 st를 생성.

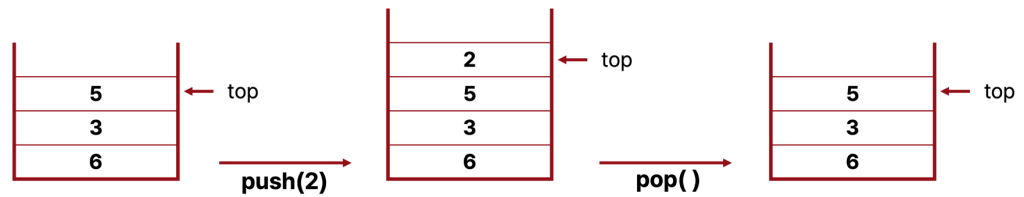
    for (char ch : expression) {
        if (ch == '(' || ch == '[' || ch == '{') {    // 열린 괄호라면
            st.push(ch);                            // 스택에 문자 1개를 '배열 슬롯에 직접 저장'.

        } else if (ch == ')' || ch == ']' || ch == '}') { // 닫힌 괄호라면
            if (st.isEmpty()) return false;          // 닫을 대상이 없으면 불일치 → 바로 false.

            char open = st.pop();                    // 스택 최상단 문자를 복사 반환(top 감소). data 슬롯을 실제로 지우진 않음(논리 삭제).

            if ((open == '(' && ch != ')') ||          // 열린/닫힌 괄호 페어 검증.
                (open == '[' && ch != ']') ||
                (open == '{' && ch != '}')) {
                return false;                        // 짝이 틀리면 false.
            }
        }
        // 다른 문자는 무시
    }
    return st.isEmpty();                            // 모든 문자를 처리한 뒤, 스택이 비어 있어야 완전 매칭.
                                                    // 비어 있지 않다면 data에 남은 값이 있지만 '미달함' 상태 → false.
}
```

실행

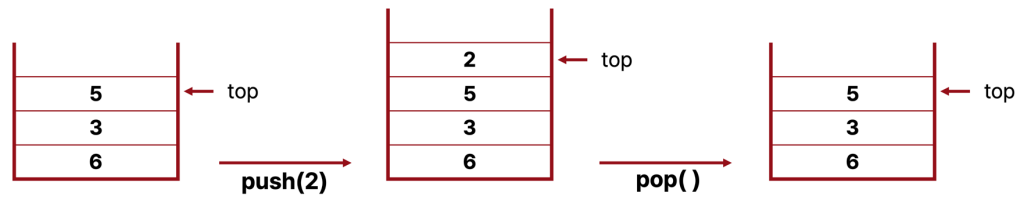


- clang++ -std=c++11 main.cpp answer.cpp -o main
- clang++ -std=c++11 main.cpp answer_official.cpp -o main
- ./main

```
// 데모: LIFO 동작 확인
static void demo_stack_basic() {
    StackType st;
    st.push('A'); st.push('B'); st.push('C');
    std::cout << "[DEMO] push A, B, C -> pop (LIFO)\n";
    while (!st.isEmpty()) {
        std::cout << "pop => " << st.pop()
                  << " (size=" << st.size() << ")\n";
    }
}
```

```
(base) minseo@minseo-MacBookAir-2 1001-stack % ./main
[DEMO] push A, B, C -> pop (LIFO)
pop => C (size=2)
pop => B (size=1)
pop => A (size=0)
```

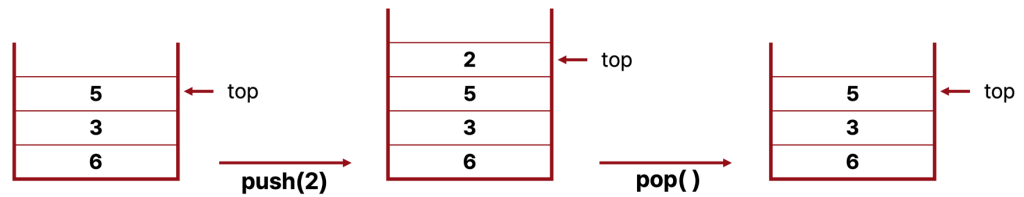
실행



- ./main --judge challenge_inputs.txt

```
(base) minseo@minseo-MacBookAir-2 1001-stack % ./main --judge challenge_inputs.txt
MATCH
MATCH
UNMATCH
MATCH
MATCH
MATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
UNMATCH
```

실행



- ./main --judge “내가 직접 만든 괄호 문제”

```
(base) minseo@minseo-MacBookAir-2 1001-stack % ./main --judge "(){}"  
MATCH  
(base) minseo@minseo-MacBookAir-2 1001-stack % ./main --judge "(){}]"  
UNMATCH  
(base) minseo@minseo-MacBookAir-2 1001-stack % ./main --judge "(){{}}[[[]]]{()}90(){}]"  
UNMATCH  
(base) minseo@minseo-MacBookAir-2 1001-stack % ./main --judge "{[()] }a+b*(c-d)}"  
UNMATCH  
(base) minseo@minseo-MacBookAir-2 1001-stack % ./main --judge "{[()] }a+b*(c-d)"  
MATCH
```



수고하셨습니다

자료구조 실습 10/01

EOF