



부산대학교
PUSAN NATIONAL UNIVERSITY

Stacks

Last In, First Out

Operations, Expressions, and Function Calls

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

The Stack ADT

나중에 넣은 값을 먼저 꺼내는 연산을 정한다

정의

스택(stack)

맨 위(top)라고 부르는 한쪽 끝에서만 값을 넣고 빼는 ADT. last-in first-out(LIFO)는 가장 나중에 넣은 값을 가장 먼저 꺼내는 순서 규칙이다.

쌓은 순서와 꺼내는 순서

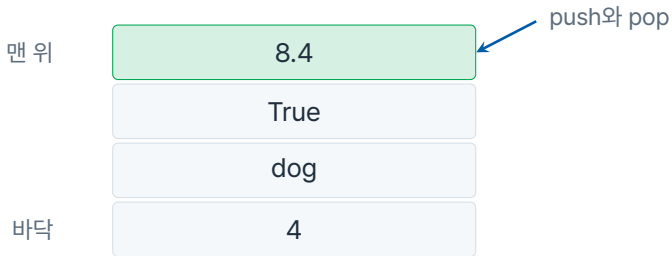
책을 위에 올리고 위에서 꺼내면 마지막에 올린 책이 먼저 나온다



맨 위는 Python 책이 놓인 위쪽 끝이다

Source: Miller and Ranum (2013), Stacks.

맨 위와 바닥



스택의 삽입과 삭제는 항상 같은 끝에서 일어난다

스택 연산

연산	입력과 반환값	효과
<code>push(x)</code>	x , 반환값 없음	x 를 맨 위에 추가
<code>pop()</code>	맨 위 값 반환	맨 위 값을 삭제
<code>peek()</code>	맨 위 값 반환	스택은 바꾸지 않음
<code>is_empty()</code>	True/False 반환	비어 있는지 검사
<code>size()</code>	정수 반환	저장한 값의 수를 반환

Caution 빈 스택에서 `pop()`과 `peek()`은 돌려줄 값이 없으므로 먼저 `is_empty()`로 확인한다

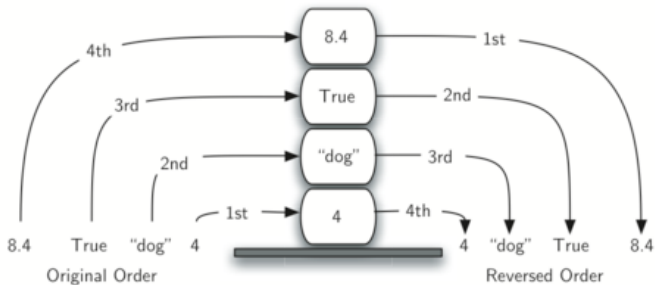
push와 pop이 맨 위를 바꾼다

연산	호출 전	호출 후
push(C)	맨 위는 B	C가 새 맨 위
pop()	맨 위는 C	C를 반환하고 B가 맨 위
peek()	맨 위는 B	B를 반환하고 맨 위는 그대로

LIFO 순서는 값의 종류가 아니라 push와 pop의 호출 순서로 정해진다

LIFO는 순서를 뒤집는다

들어간 순서와 나온 순서가 반대가 된다



4, dog, True, 8.4를 차례로 넣으면 8.4부터 꺼낸다

Source: Miller and Ranum (2013), Stacks.

스택이 필요한 문제의 공통점

문제	나중 항목을 먼저 쓰는 이유
괄호 검사 $((\)(\))$	가장 최근의 여는 괄호부터 짝을 맞춤
후위식 평가 $4\ 5\ 6\ *\ +$	가장 최근 두 피연산자를 먼저 계산
함수 반환	가장 최근에 시작한 호출부터 끝냄

미뤄 둔 항목 가운데 가장 최근 것을 먼저 처리해야 하면 스택을 고려한다

Checkpoint

The Stack ADT

- LIFO에서 다음에 꺼낼 값은 어떻게 정하는가
- `pop()`과 `peek()`은 스택을 어떻게 다르게 바꾸는가
- 빈 스택에서 호출 조건을 따로 정해야 하는 연산은 무엇인가

A Stack in Python

맨 위를 어느 끝에 두는지가 Python 구현 비용을 바꾼다

list의 끝 선택이 연산 비용을 바꾼다

맨 위 위치	push와 비용	pop과 비용
오른쪽 끝	<code>append(x)</code> , 상각 $\Theta(1)$	<code>pop()</code> , 상각 $\Theta(1)$
왼쪽 앞	<code>insert(0,x)</code> , $\Theta(n)$	<code>pop(0)</code> , $\Theta(n)$

두 구현은 같은 LIFO 결과를 내지만 연산 비용은 다르다

오른쪽 끝을 맨 위로 둔다

Python list의 끝을 쓰는 ArrayStack과 같은 구조다

```
1 class Stack:
2     def __init__(self):
3         self._items = []
4     def push(self, item):
5         self._items.append(item)
6     def pop(self):
7         return self._items.pop()
8     def peek(self):
9         return self._items[-1]
10    def is_empty(self):
11        return not self._items
12    def size(self):
13        return len(self._items)
14
```

push와 pop이 같은 오른쪽 끝을 사용하므로 LIFO가 된다

Source: Miller and Ranum (2013), Listing stackcode1.

오른쪽 끝 구현의 비용

Python list는 동적 배열이므로 재할당 비용이 그대로 적용된다

연산	보통 하는 일	비용
push	끝에 저장	상각 $\Theta(1)$
pop	끝 값 삭제	상각 $\Theta(1)$
peek	마지막 인덱스 접근	$\Theta(1)$

Caution 재할당이 일어나는 한 번의 push는 $\Theta(n)$ 일 수 있다

왼쪽 앞을 맨 위로 둔다

스택 연산	Python 구현
<code>push(self, item)</code>	<code>self._items.insert(0, item)</code>
<code>pop(self)</code>	<code>return self._items.pop(0)</code>
<code>peek(self)</code>	<code>return self._items[0]</code>
<code>is_empty(self)</code>	<code>return not self._items</code>
<code>size(self)</code>	<code>return len(self._items)</code>

Source: Miller and Ranum (2013), Listing stackcode2.

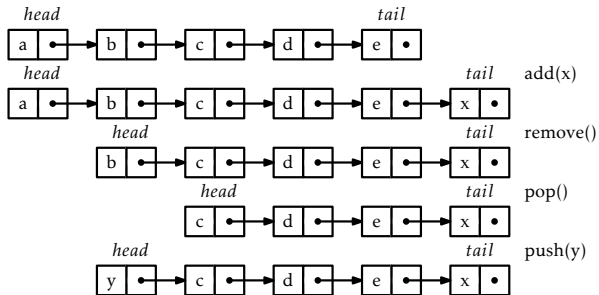
SLList로도 스택을 만들 수 있다

머리(head)를 맨 위로 두면 앞 삽입과 앞 삭제만 사용한다

스택 연산	SLList에서 하는 일	비용
push(x)	새 노드를 머리 앞에 연결	$\Theta(1)$
pop()	머리를 다음 노드로 옮김	$\Theta(1)$
peek()	머리의 값 읽기	$\Theta(1)$

SLList 스택에서 바뀌는 참조

원소가 둘 이상이면 head만, 빈 목록에 넣거나 마지막 노드를 꺼내면 tail도 바뀐다



아래 두 줄이 스택 연산이다. 위의 add(x)와 remove()는 뒤에 넣고 앞에서 꺼내는 큐 연산이다.

Checkpoint

A Stack in Python

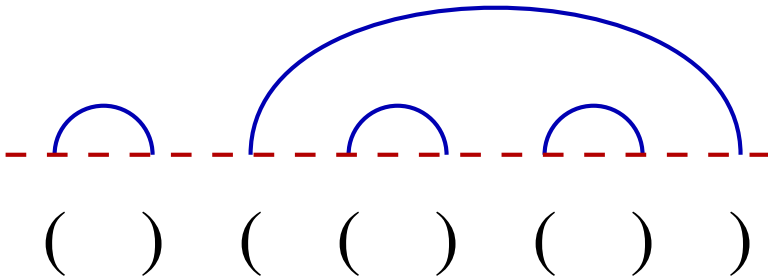
- Python `list`의 오른쪽 끝을 맨 위로 두는 까닭은 무엇인가
- 왼쪽 앞 구현도 LIFO 동작 자체는 맞는 까닭은 무엇인가
- `SLList`로 만든 스택에서 순회가 필요하지 않은 까닭은 무엇인가

Parentheses and Postfix

괄호 검사의 스택 상태를 끝까지 추적하고 후위식에서 같은 LIFO 패턴을 확인한다

괄호 검사

괄호 검사(balanced parentheses)는 여는 괄호와 닫는 괄호가 순서에 맞게 짝지어졌는지 확인한다



가장 최근에 나온 여는 괄호가 다음 닫는 괄호와 먼저 짝을 이룬다

괄호 검사에서 스택이 기억하는 것

스택에는 지금까지 읽은 괄호 중 아직 짝을 찾지 못한 여는 괄호가 남는다

읽은 기호	확인할 조건	스택에 하는 일
여는 괄호	없음	push
닫는 괄호	스택이 비어 있지 않고 맨 위와 짝이 맞음	pop
입력 끝	스택이 비어 있음	모두 짝이 맞음

Caution 닫는 괄호를 만났는데 스택이 비어 있거나 짝이 다르면 즉시 실패한다

((()())를 따라가 본다

기호	동작	스택
(	push	[(]
(	push	[(, (]
)	pop	[(]
(	push	[(, (]
)	pop	[(]
)	pop	[]

끝까지 읽었을 때 스택이 비어 있으므로 괄호가 맞는다

괄호가 맞지 않는 세 경우

입력	문제	검출 시점
) (	닫을 여는 괄호가 없음	빈 스택에서 닫는 괄호를 만남
(((	여는 괄호가 남음	입력을 모두 읽은 뒤
([]	괄호 종류가 어긋남	맨 위와 닫는 괄호 비교

Source: Miller and Ranum (2013), Balanced Symbols.

괄호 검사를 코드로 옮기면

소괄호만 검사하는 코드다. 괄호 종류까지 보려면 꺼낸 여는 괄호와 닫는 괄호의 짝을 비교해야 한다

```
1 def par_checker(text):
2     stack = []
3     for symbol in text:
4         if symbol == "(":
5             stack.append(symbol)
6         elif symbol == ")":
7             if not stack:
8                 return False
9             stack.pop()
10    return not stack
11
```

빈 스택에서 닫는 괄호를 만나거나 입력을 다 읽은 뒤 여는 괄호가 남으면 실패한다

정의

후위식(postfix expression)

연산자를 두 피연산자 사이에 쓰는 보통의 식은 중위 표기식(infix expression)이다.

후위식은 연산자를 두 피연산자 뒤에 쓰는 식이다. $A + B \times C$ 는 $A B C \times +$ 로 나타내며 읽는 순서만으로 계산할 수 있다.

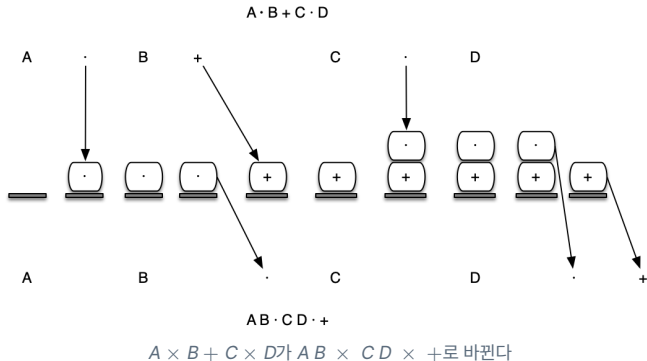
중위 표기식을 후위식으로 바꾼다

읽은 항목	동작
피연산자	출력에 바로 추가
여는 괄호	연산자 스택에 push
닫는 괄호	여는 괄호가 나올 때까지 pop하여 출력하고, 그 여는 괄호는 꺼내 버림
연산자	맨 위가 여는 괄호가 아니고 우선순위가 높거나 같은 동안 pop하여 출력한 뒤 push
입력 끝	남은 연산자를 모두 pop하여 출력

Source: Miller and Ranum (2013), Infix-to-Postfix Conversion.

$A \times B + C \times D$ 의 변환

연산자는 출력할 차례가 올 때까지 스택에서 기다린다



Source: Miller and Ranum (2013), Figure 9.

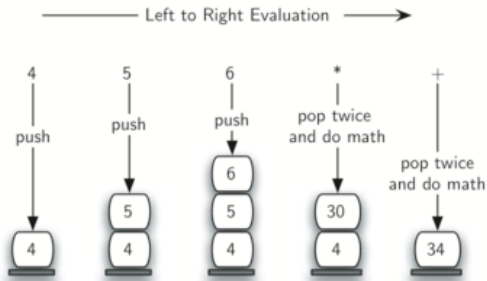
후위식은 왼쪽부터 평가한다

읽은 항목	동작
피연산자	값 스택에 push
연산자	오른쪽 값, 왼쪽 값 순서로 두 번 pop
계산 결과	다시 스택에 push
입력 끝	스택에 남은 하나가 결과

Caution 뿔셈과 나눗셈은 두 값을 pop한 순서를 바꾸면 결과도 바뀐다

4 5 6 × +의 평가

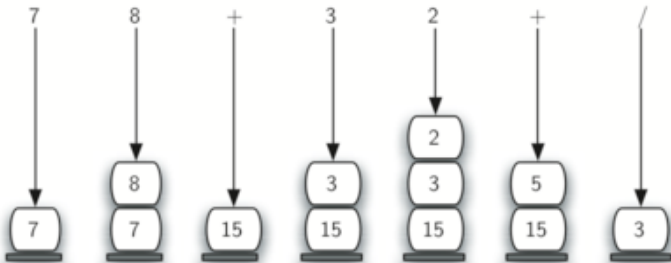
5 × 6을 먼저 계산한 뒤 4를 더한다



마지막 연산 뒤 스택에 34 하나만 남는다

나눗셈에서 순서를 확인한다

$78 + 32 + /$ 는 $(7 + 8)/(3 + 2)$ 다



15와 5를 차례로 만든 뒤 $15/5 = 3$ 을 계산한다

Source: Miller and Ranum (2013), Evaluating Postfix Expressions.

괄호 검사와 후위식의 공통 구조

문제	push하는 것	pop하는 때
괄호 검사	여는 괄호	닫는 괄호를 만날 때
후위식 변환	연산자	출력할 차례가 될 때
후위식 평가	피연산자와 중간 결과	연산자를 만날 때

나중에 필요할 값을 스택에 보관하고 필요한 순간에 맨 위부터 꺼낸다

Checkpoint

Parentheses and Postfix

- 괄호 검사에서 스택에 남아 있는 값은 무엇인가
- 중위 표기식을 후위식으로 바꿀 때 연산자를 스택에 두는 까닭은 무엇인가
- 후위식 평가에서 두 값을 pop하는 순서가 중요한 연산은 무엇인가

The Call Stack

호출 스택의 변화를 추적하고 진법 변환의 LIFO 순서를 확인한다

정의

호출 스택(call stack)

실행 중인 함수 호출을 LIFO 순서로 관리하는 구조. 함수를 호출하면 기록을 push하고, 함수가 반환하면 그 기록을 pop해 호출한 함수로 돌아간다.

정의

호출 프레임(stack frame)

함수 호출 한 번에 필요한 실행 상태. 인자, 지역 변수, 반환 뒤 이어서 실행할 위치를 호출마다 따로 저장하므로 같은 함수의 여러 호출도 서로 섞이지 않는다.

호출과 반환을 따라간다

main이 f를 부르고 f가 g를 부르는 경우

사건	스택 연산	호출 스택, 오른쪽이 맨 위
main 시작	push	[<i>main</i>]
f() 호출	push	[<i>main</i> , <i>f</i>]
g() 호출	push	[<i>main</i> , <i>f</i> , <i>g</i>]
g() 반환	pop	[<i>main</i> , <i>f</i>]
f() 반환	pop	[<i>main</i>]

Adapted from Miller and Ranum (2013), Stack Frames: Implementing Recursion.

정수를 문자열로 바꾸는 재귀 함수

재귀(recursion)는 함수가 자신을 다시 부르는 방식이다. 이 코드는 뒤편으로 인자를 줄인다

```
1 def to_str(n, base):
2     convert_string = "0123456789ABCDEF"
3     if n < base:
4         return convert_string[n]
5     else:
6         return to_str(n // base, base) + convert_string[n % base]
7
8 print(to_str(10, 2))    # 1010
9
```

$n < \text{base}$ 가 기저 사례이고, 재귀 호출은 뒤편 $n // \text{base}$ 로 인자를 줄인다

재귀 호출도 호출 프레임을 하나씩 쌓는다

`to_str(10,2)`가 더 작은 인자로 자신을 다시 부른다

"1"

`to_str(1,2)`의 반환값

```
to_str(2, 2)  n = 2  base = 2  
to_str(2 // 2, 2) + convert_string[2 % 2]
```

```
to_str(5, 2)  n = 5  base = 2  
to_str(5 // 2, 2) + convert_string[5 % 2]
```

```
to_str(10, 2)  n = 10  base = 2  
to_str(10 // 2, 2) + convert_string[10 % 2]
```

`to_str(1,2)`가 "1"을 반환한 뒤에도 n 이 10, 5, 2인 호출 프레임이 남아 있고, 반환값은 맨 위에 놓인다.

같은 변수 이름도 호출마다 값이 다르다

호출	n	base	계산할 나머지
to_str(10,2)	10	2	0
to_str(5,2)	5	2	1
to_str(2,2)	2	2	0
to_str(1,2)	1	2	없음(기저 사례, "1" 반환)

각 호출 프레임이 지역 변수를 따로 저장하므로 재귀 호출이 서로 섞이지 않는다

Source: Miller and Ranum (2013), Stack Frames: Implementing Recursion.

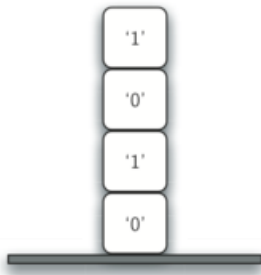
반환은 호출의 역순이다

반환하는 호출	반환 문자열	다음 계산
<code>to_str(1,2)</code>	"1"	호출한 프레임으로 전달
<code>to_str(2,2)</code>	"10"	호출한 프레임으로 전달
<code>to_str(5,2)</code>	"101"	호출한 프레임으로 전달
<code>to_str(10,2)</code>	"1010"	최종 반환

Source: Miller and Ranum (2013), Stack Frames: Implementing Recursion.

명시적 스택으로 같은 순서를 만든다

재귀 호출 대신 나머지 문자를 스택에 직접 쌓을 수 있다



나머지를 0, 1, 0, 1 순서로 쌓은 뒤 맨 위부터 꺼내면 "1010"이 된다

Source: Miller and Ranum (2013), Stack Frames: Implementing Recursion.

명시적 스택을 쓰는 변환 코드

나머지 글자를 push한 뒤 스택이 빌 때까지 pop해 이어 붙인다

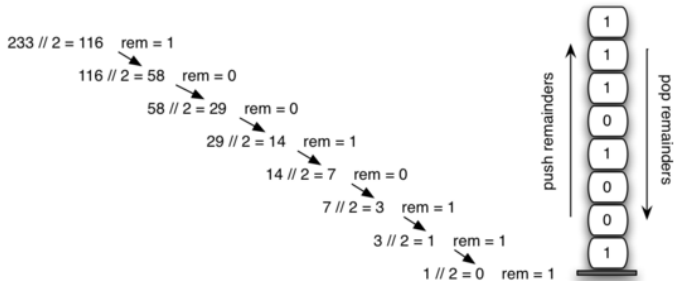
```
1 def to_str(n, base):
2     r_stack = Stack()
3     convert_string = "0123456789ABCDEF"
4     while n > 0 or r_stack.is_empty():
5         if n < base:
6             r_stack.push(convert_string[n])
7         else:
8             r_stack.push(convert_string[n % base])
9         n = n // base
10    res = ""
11    while not r_stack.is_empty():
12        res = res + str(r_stack.pop())
13    return res
14
```

재귀판은 호출 스택이, 이 판은 작성한 코드가 같은 LIFO 순서를 만든다

Adapted from Miller and Ranum (2013), Stack Frames: Implementing Recursion.

10진수 233의 2진수 변환에도 LIFO 순서를 쓴다

2로 나눈 나머지는 계산된 순서와 반대로 읽어야 한다



나머지를 차례로 push하고 맨 위부터 pop하면 11101001₂가 된다

명시적 스택과 호출 스택

구조	쌓는 것	push와 pop을 하는 주체
명시적 스택	괄호, 값, 연산자, 나머지	작성한 코드
호출 스택	함수 호출의 실행 상태	Python 실행 환경

저장하는 대상은 달라도 두 구조 모두 가장 최근에 넣은 것부터 꺼낸다

Checkpoint

The Call Stack

- 함수를 호출하고 반환할 때 호출 스택에는 어떤 연산이 일어나는가
- 같은 함수의 지역 변수가 재귀 호출마다 섞이지 않는 까닭은 무엇인가
- 명시적 스택과 호출 스택은 쌓는 대상이 어떻게 다른가

What Changes When the Oldest Item Leaves First?

Next Lecture: Queues and Deques

- FIFO와 enqueue, dequeue
- 원형 큐와 양끝 연산
- 대기 순서를 사용하는 시뮬레이션