CS206A: Data Structures

2017-09-15

카이스트 전산학부 정지원(Otfried Cheong) 著
부경대학교 컴퓨터공학과 김태원 譯

2022년 7월 6일

스택

스택(stack, 더미)은 요소 모음을 저장하는 추상 자료형이며, 책의 더미나 식당에 쌓인 식판의 더미처럼 행동한다. 즉 오직 탑(top, 최상위) 요소에만 접근할 수 있다.

형식적으로, 스택은 다음 연산을 지원한다.

- `push(el)`은 스택 위로 요소를 민다(push). 이는 새로운 최상위 요소가 된다.
- `top()`은 스택 최상위에 있는 요소를 반환한다.
- `pop()`은 스택 최상위에 있는 요소를 반환하고 이를 스택에서 제거한다. 그 바로 아래에 있는 요소는 새로운 최상위 요소가 된다.
- `is_empty()`는 스택 위에 요소가 없으면 참을 반환한다.

스택이 비었을 때는 `top()`과 `pop()`이 예외를 일으킬 것이다.

스택은 가끔 LIFO라고 부르는데, 후입선출(後入先出, *Last in First Out*)의 원리에 따라 작동하기 때문이다.

간단한 클라이언트

스택의 LIFO 행태를 이용하는 클라이언트 코드의 일례로, 문자열을 역순으로 출력하는 함수가 아래에 있다.

```
def reverse(s):
    S = Stack()
    for ch in s:
        S.push(ch)
    while not S.is_empty():
        ch = S.pop()
        print(ch, end="")
    print()
```

균형 잡힌 괄호 확인하기

산술 표현식(arithmetic expression)이 제대로 균형 잡혔는지 확인하는 함수를 작성해 보자. 이는 표현식을 전부 파싱하는(parse) 것보다 훨씬 간단하다. 실상, 우리는 표현식 안의 상이한 종류의 괄호만 보면 되고, 나머지 기호는 전부 무시할 수 있다.

가령, `(){[(){}](())}`는 올바르게 균형 잡혔지만, `(){[(){}](())}`는 아니다. (왜냐하면 `)` 괄호가 `{`괄호를 닫기 때문이다.)

우리는 다음 전략을 사용한다.

1. 빈 스택을 만든다.
2. 문자열 안의 각 기호에 대해
   i 문자열이 비었으면, 거짓을 반환한다.
   ii 스택의 탑(top, 최상위)이 닫는 기호와 들어맞지 않는다면, 거짓을 반환한다.
   iii 스택을 팝(pop)한다.
3. 스택이 비었다면 참을 반환하고, 아니라면 거짓을 반환한다.

코드는 `balanced.py`에서 확인할 수 있다.

스택 구현

파이썬 리스트 사용하기.

간단한 스택 구현이 `liststack` 모듈에 주어져 있다. 스택 요소를 저장하고자 파이썬 리스트를 사용한다. `push`는 리스츠 추가를 수행하고, `pop`은 리스트 `pop()`, 리스트의 가장 최근 요소 제거를 수행한다.

모든 연산은 앞서 논한 것처럼 평균적으로 상수 시간(常數, constant time)으로 동작한다. 하지만 어떤 `push()` 연산은 더 오래 걸리는데, 파이썬 리스트에 의해 내재적으로 사용되는 배열이 꽉 차면 새로운 배열이 배정되어야 하기 때문으로, 그러한 `push()` 연산은 스택의 현재 크기에 대해 시간이 선형적으로(linear) 걸린다.

주어진 최대 크기로.

파이썬 프로그램의 경우, 당신에게 더 강력한 보장(guarantee)이 필요할 일은 많지 않다. 하지만 임베디드 회로(embedded circuit)에서 구동하는 `C` 코드를 작성한다면, 함수가 몇 마이크로초(microseconds)안에 반환하기를 보장받아야 할지도 모른다. 그런 응용에서는 각 연산에 대해 상수 시간을 보장하는 스택이 필요하다.

이는 만약 스택이 평생 지닐 최대 크기를 미리 안다면 취하기 쉬운 것이다. 그냥 충분한 크기의 리스트를 생성해, 스택의 현재 탑의 인덱스(index)를 우리가 직접 유지한다.

`arraystack` 모듈에서 구현을 확인할 수 있다.

링크드 스택.

최대 스택 크기를 미리 알지 못한다면, 그냥 파이썬 리스트나 배열을 사용할 수 없다. 우리는 더 작은 객체로 작업해야 한다.

해결책은 노드(nodes)라고 부르는 작은 객체를 생성하는 것이다. 각 노드는 스택의 한 요소(에 대한 레퍼런스)와 더불어 다음 노드에 대한 레퍼런스를 저장한다. `Stack` 객체 자체는 스택의 탑에 있는 노드에 관해서만 알 뿐이다. 이 노드는 스택의 탑 바로 아래 노드에 대한 레퍼런스를 지니며, 이런 식으로 계속된다. 마지막 노드는 이어지는 노드에 대한 레퍼런스 대신 `None`을 저장한다.

`linkedstack` 모듈에서 구현을 찾을 수 있다.

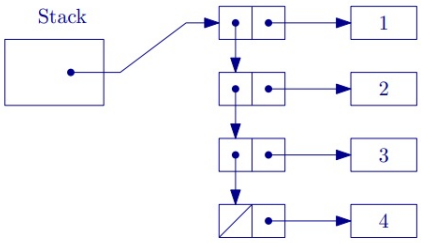

그림 0.1: 작은 노드를 사용한 스택 구현

스택 프레임

앞서 함수의 지역 변수가 자신의 스택 프레임(stack frame)에 저장된다고 배웠다. 이는 함수가 호출될 때마다 생성되는 메모리 블록이며, 함수가 반환할 때마다 파괴되는 것이다.

아마 이미 예측했겠지만, 파이썬 인터프리터는 모든 스택 프레임을 스택에 보관한다. 함수가 호출될 때, 그 스택 프레임은 생성되어 스택에 푸시된다(pushed). 함수가 반환할 때, 그 스택 프레임은 스택에서 팝되어(poped) 파괴된다. 따라서 현재 실행 중인 함수는 항상 스택의 탑에 있다. (`f` 함수가 반환할 때, 스택의 새로운 탑은 `f`를 호출한 함수의 스택 프레임이 되어야 한다.)

스택은 왜 스택 프레임 저장에 대해 작동하는가? 왜냐하면, 함수의 시작 시간과 반환 시간이 마치 균형 잡힌 괄호처럼 근사하게 중첩된 구조(nested structure)를 형성하기 때문이다. 당신은 `f` 함수에 대한 현재 호출 이후 호출된 모든 함수들이 이미 반환되지 않으면 `f`에서 반환할 수 없다.

스택 프레임을 저장하는 실행시간 스택은 파이썬 인터프리터에 내장되어 있다. 이는 파이썬 객체 자체가 아니다.

우리가 재귀를 논했을 적에, 당신은 재귀가 정말 어떻게 작동하는 것인지 의문을 품었을지도 모른다. 파이썬 인터프리터는 어떻게 이들 모든 호출과 재귀 함수 `f`를 구분하는가? 헷갈려 하지는 않는가?

답은 이렇다. 바로 실행시간 스택이 재귀를 가능하게 한다. 재귀 함수 `f`에 대한 각각의 호출은 자신만의 스택 프레임을 자신만의 지역 변수 사본과 함께 지니며, 가장 최근 호출은 항상 스택의 탑에 위치한다. 이는 재귀 깊이(recursion depth)에 한계가 있다는 점에 대한 이유를 설명하기도 하는데, 기본 설정에 따라 파이썬 실행시간 스택은 1000 스택 프레임으로 제한된다.

이 통찰의 귀결 하나는 다음과 같다. 당신은 언제나 스택을 사용해 재귀 함수를 비(非)재귀 함수로 재작성할 수 있다. 본질적으로, 당신은 실행 시간 스택을 “시뮬레이션"하기 위해 스택을 사용한다.