FIFO는 Q큐 | 4. 스택 (LIFO)

스택이랑 다른거

◦ 데이터를 제한적으로 접근할수 있는 구조

→ 한쪽끝에서만 자료를 넣거나 뺄수 있는 구조

◦ 가장 나중에 쌓은 데이터를 가장 먼저 빼낼수 있는 데이터구조

→ 후입선출 (LIFO)

→ 가장 최근에 들어온 데이터가 가장 먼저 나감

스택 오류

◦ 두가지상황 (대표적)

is full() 꽉 차있는상태거나 (꽉 차있는상태에서 삽입)

비어있는상태에서 참조할때 ~~Pick~~ Peek is Empty() (없는상태에서 삭제,

스택 ADT

데이터 → 후입선출(LIFO)의 접근 방법을 유지하는 항목들의 모음

top이 중요 | 연산

Stack() → 비어있는 새로운 스택을 만든다

연결리스트

isEmpty() 비어있다 → True 아니다 → False

Pop ~~push~~() → 스택맨 위에있는 항목을 꺼내서 반환

Peek ~~pop~~() → 항목을 삭제하지않고 반환

push ~~peek~~() → 항목e를 스택의 맨위에 추가

Size() → 스택내의 모든항목들의 개수를 반환

Clear() → 스택을 공백상태로 만든다

<활용>

◦ 웹페이지 뒤로가기 ◦ 함수의호출

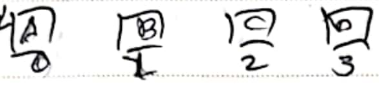
◦ 컴퓨터 내부의 프로세스 구조의 함수호출방식 ◦ 미로탐색

◦ 계산기 ◦ 괄호검사

구현 [배열구조
 └ 연결리스트 → 오제공음

배열구조 array(capacity)

top이 어디있는지 중요



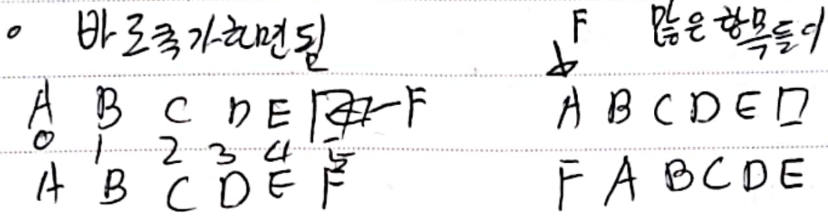
∘ top : 스택 항목을 저장할 파이썬리스트

∘ 항목의 개수는 len()으로 구할 수 있음

삽입/삭제

└> 리스트의 맨 뒤로

∘ 바로 추가하면 됨 — F 많은 항목들이 필드함 비트율적



괄호검사

∘ 괄호들이 같은 유형끼리 쌍을 이루어 잘 사용되었는지 검사

조건

∘ 왼쪽 괄호의 개수와 오른쪽 괄호의 개수가 같아야한다.

∘ 같은 타입의 괄호에서 왼쪽괄호가 오른쪽 괄호보다 먼저나와야한다.

∘ 서로 다른 타입의 괄호를 (쌍)서로 교차하면 안된다.

방법 (스택활용)

∘ 문자를 저장하는 스택을 준비한다 (처음에는 공백상태가 되어야 한다)

∘ 입력 문자열의 문자를 하나씩 읽어 왼쪽 괄호를 만나면 스택에 삽입한다.

∘ 오른쪽 괄호를 만나면 pop() 연산으로 가장 최근에 삽입된 괄호를 꺼낸다.

이때 스택이 비었으면 조건 2에 위배된다

∘ 꺼낸 괄호가 오른쪽 괄호와 짝이 맞지 않으면 조건3에 위배된다.

∘ 끝까지 처리했는데 스택에 괄호가 남아 있으면 조건 1에 위배된다.

중위 → 후위

[[((A * B) + (C / D)) - E]

((A * (B + C)) / D) - E

ABC+*D/E -

A * (B+C) / D - E

(((A * (B+C)) / D) - E)

ABC+*D/E - 요런식?

(x+y) - (W*Z) / V

(((x+y)) - ((W*Z)) / V))

xy+ - WZ*V / -

xy+WZ*V/-

중위 → 후위 → 계산 (결과)

스택 / 복잡함

○ 스택
피연산자면 집어넣고
연산자 만나면 꺼내
그걸 스택을 빈 곳 결과값을 다시
집어넣는다. 이걸 반복
스택 2번씀

후위표기법 장점

○ 괄호를 사용하지도 않아도 계산 순서를 알 수 있음

○ 연산자의 우선순위를 생각할 필요가 없음 (식 자체에 우선순위가 이미 포함되어있음)

요게 젤 중요

○ 수식을 읽으면서 바로 계산할 수 있음 (중위표기법의 경우 수식을 끝까지 읽은 다음에야 계산이 가능함)

8/2 - 3 + (3*4)

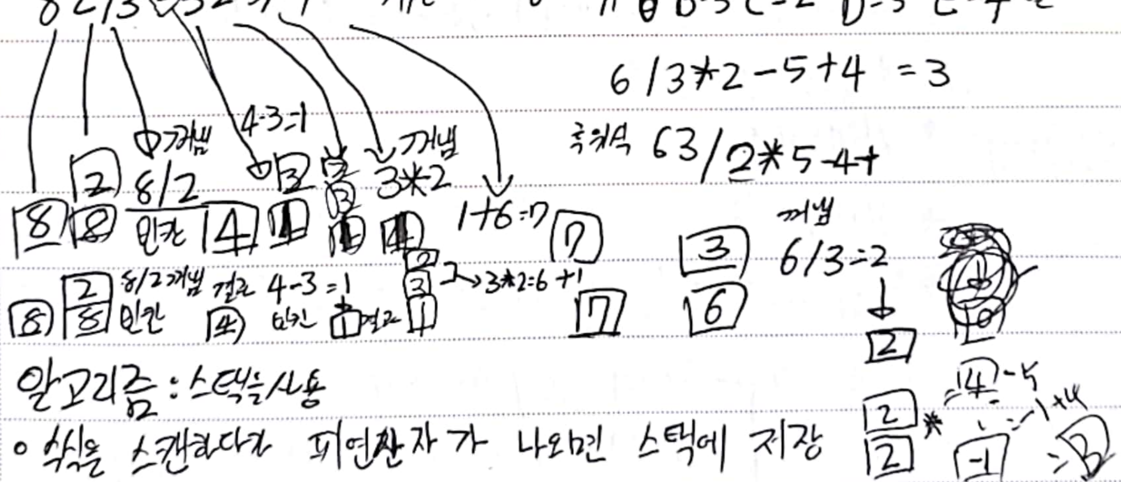
82/3-32*+ 계산하는 방법

A/B*C - D+E = AB/C*D-E+

A=6 B=3 C=2 D=5 E=4 일

6/3*2-5+4 = 3

후위식 63/2*5-4+



알고리즘 : 스택을 사용

○ 수식을 스캔하다가 피연산자가 나오면 스택에 저장

○ 연산자가 나오면 스택에서 피연산자 두개를 꺼내 연산을 실행하고 그 결과를 다시 스택에 저장

• 이 과정을 수식이 모두 처리될때 까지 반복

• 마지막으로 스택에는 최종 계산 결과가 남음

중위 표기와 후위표기

○ 중위와 후위표기법의 공통점: 피연산자의 순서가 동일

○ 연산자들의 순서만 다름 (우선순위 순서)

→ 연산자만 스택에 저장 후 출력, 2+3*4 → 234+*

알고리즘 중위 → 후위

○ 입력된 중위표기 수식을 순서대로 하나씩 스캔한다

(제일중요) ○ 피연산자를 만나면 그대로 출력

○ 연산자를 만나면 스택에 저장했다가 스택보다 우선 순위가 낮은 연산자가 나오면 그때 출력

○ 왼쪽 괄호는 우선순위가 가장 낮은 연산자로 취급

○ 오른쪽 괄호가 나오면 스택에서 왼쪽 괄호 위에 쌓여 있는 모든 연산자를 출력

(핵심: push → / 안 >= 밖 → pop / 안 < 밖 → push)

결론 피연산 그냥출력

연산자 스택 삽입 → 안(op) >= 밖(op) : 안 pop() / 밖 push()

(괄호가없을때) → 안 < 밖 = 밖 push()

(있을때) → 스택에) 이올때 (~ 인던 모든연산자 넣기 (POPC)) 하는거임

└> 스택밖 → 연산자 우선순위가높음 / 안 → 가장낮음

응용 (DFS → 스택, BFS → Q)

미로 탐색 → 미로에 갇힌 생쥐가 출구를 찾는 문제

○ 그래프 탐색 문제 (DFS, BFS)와 유사

○ 가장 간단한 방법

→ 시행 착오 (1은 벽 0은갈수있는데)

→ 하나의 경로를 선택하여 시도해 보고 막히면 다시 다른경로를 시도하는 것

→ 현재 경로가 막혔을때 다시 선택할수 있는 다른 경로들을 어딘가에 저장해야 함

중요한 3가지
- 벽인지 길인지 확인하기
- Visted ( ) 지금 가는길인지 가본길인지
- 상하좌우 범위내에 있어야 됨

탐색알고리즘

- Step1. 시작위치를 스택에 넣는다.
- step2. 스택이 공백이 아니라면 최상의 위치를 꺼낸다. 이것이 현재위치이다. 현위치에 "방문했음" 표시를 한다. 만약 스택이 공백이라면 이 미로에는 출구가 없는 것이므로 (목적주소) 종료한다
- step3 만약 현재위치가 출구라면 탐색은 성공으로 끝난다.
- 그렇지 않다면 이웃 방들을 살펴보고 만약 이웃방들 중 아직방문되지 않았고 갈수 있는 방이라면 그 방의 위치를 모두 스택에 삽입한다

다시 step2로 돌아감

DFS

Depth First Search → 가던길이 막히면 가장최근에 왔던 갈림길로 되돌아가서 다른길을 찾는 방법

=> Stack이용