Rokey Study

Rokey 스터디

목 차

Mechanical Engineering

- Stack 구현: 웹 브라우저의 뒤로 가기 기능

```python
# Stack: 웹 브라우저의 뒤로 가기
class Stack:
    def __init__(self):
        self.stack = []  # 내부적으로 리스트를 사용하여 스택을 구현

    def push(self, item):
        self.stack.append(item)  # 스택의 맨 위에 항목 추가

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()  # 스택의 맨 위 항목을 제거하고 반환
        return None  # 스택이 비어 있으면 None 반환

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]  # 스택의 맨 위 항목을 확인(제거하지 않음)
        return None  # 스택이 비어 있으면 None 반환

    def is_empty(self):
        return len(self.stack) == 0  # 스택이 비어 있는지 확인
```

```python
# 웹 브라우저 방문 예시
browser_history = Stack()
browser_history.push("Homepage")
browser_history.push("Page 1")
browser_history.push("Page 2")

print(browser_history.pop())  # 'Page 2'출력
print(browser_history.pop())  # 'Page 1' 출력
print(browser_history.pop())  # 'Homepage' 출력
```

✓ 0.0s

```
Page 2
Page 1
Homepage
```

- 웹 브라우저에서 사용자가 방문한 웹페이지는 스택에 쌓이고 뒤로 가기 버튼을 누르면 스택에서 가장 마지막에 방문한 페이지를 가져온다.

- Queue 구현: 프린트 작업 대기열

```python
# Queue: 프린트 작업 대기열
class Queue:
    def __init__(self):
        self.queue = []  # 내부적으로 리스트를 사용하여 큐를 구현

    def enqueue(self, item):
        self.queue.append(item)  # 큐의 끝에 항목 추가 (삽입)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)  # 큐의 앞쪽 항목을 제거하고 반환 (선입선출)
        return None  # 큐가 비어 있으면 None 반환

    def peek(self):
        if not self.is_empty():
            return self.queue[0]  # 큐의 맨 앞 항목을 확인(제거하지 않음)
        return None  # 큐가 비어 있으면 None 반환

    def is_empty(self):
        return len(self.queue) == 0  # 큐가 비어 있는지 확인
```

```python
printer_queue = Queue()
printer_queue.enqueue("Document 1")
printer_queue.enqueue("Document 2")
printer_queue.enqueue("Document 3")

print(printer_queue.dequeue())  # 'Document 1' 처리
print(printer_queue.dequeue())  # 'Document 2' 처리
print(printer_queue.dequeue())  # 'Document 3' 처리
```

✓ 0.0s

```
Document 1
Document 2
Document 3
```

- 큐를 활용하여 프린터 작업 대기열을 관리하는 프로그램
- 먼저 들어온 문서가 먼저 출력되는 방식

- DFS, BFS 실습

❑ ChatGpt 활용하여 여러 그래프를 받아 DFS, BFS 구현하는 실습 진행
❑ DFS와 BFS 구현하는데 익숙해지려고 노력함

plaintext

```
# GPT_dfs

my_graph = {1:[2,3,4],
            2:[5,6],
            3:[1,7],
            4:[1,8],
            5:[2,9],
            6:[2,9],
            7:[3,10],
            8:[4,10],
            9:[5,6],
            10:[7,8]}
```

Depth First Search

```python
def my_dfs(graph,start_node):
    stack = []                      #1 탐색할 노드를 담을 스택 자료형 생성
    visited = []                    #2 각 노드가 방문한 노드인지를 구분할 수 있는 리스트 생성
    stack.append(start_node)
    while stack:                    #3 탐색 시작 노드를 스택에 삽입
        node = stack.pop()          #4 방문할 노드를 스택에서 하나씩 꺼내기기
        if node not in visited:
            stack.extend(reversed(graph[node])) #5 스택에서 꺼낸 노드가 방문한 노드가 아니면
            print(f'stack: {stack}')            # 인접 노드를 스택에 삽입하고 방문 처리
            visited.append(node)
            print(f'visited: {visited}')
    return visited

print("DFS:",my_dfs(my_graph,1))
```

```
DFS: [1, 2, 5, 9, 6, 3, 7, 10, 8, 4]
```

! 개선할 점

1. `stack.extend(reversed(graph[node]))` **대신** `stack += reversed(graph[node])` **사용 가능**
   - `stack.extend(reversed(graph[node]))` 도 괜찮지만 `stack += reversed(graph[node])` 가 더 직관적이야.

extend를 썼을때와 같은 DFS 결과 도출 !

Recursion

🏆 **최종 피드백**

✅ **잘한 점**: 기본적인 DFS 구조가 아주 잘 되어 있음!
🚀 **개선하면 좋은 점**: `stack.extend(reversed(...))` 대신 `stack += reversed(...)` 활용, 출력 방식 최적화

지금도 충분히 잘했으니까, 다음엔 **재귀 방식의 DFS**도 한번 구현해 봐! 🔥🔥
궁금한 점 있으면 편하게 물어봐! 😊

재귀 방식의 DFS 구현에 대해
추천 받음

```python
# 1부터 N까지의 합

#1) 반복문으로 구현
def sum_iterative(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total

print(sum_iterative(5))  # 출력: 15
```

```python
#2) 재귀로 구현
def sum_recursive(n):
    if n == 1:  # 기저 조건
        return 1
    return n + sum_recursive(n - 1)  # 자기 자신을 호출

print(sum_recursive(5))  # 출력: 15
```

- 재귀
  - 함수가 자기 자신을 호출하는 기법
  - 재귀 호출을 통해 반복적인 작업 수행 가능
  - 기저 조건이 없으면 무한 루프 발생하여 스택 오버플로우 에러 발생

- 재귀 예제

```python
# 재귀 간단한 예제

def countdown(n):
    if n == 0:
        print('끝!') # 기저 조건 필수!
        return
    print(n)
    countdown(n - 1)
countdown(5)
```

✓ 0.0s

```
5
4
3
2
1
끝!
```

```scss
countdown(5)
 └> countdown(4)
     └> countdown(3)
         └> countdown(2)
             └> countdown(1)
                 └> countdown(0)  # 종료 조건 충족
```

스택의 활용사례인 함수 호출 스택 구조를
재귀에서도 확인할 수 있음

- DFS를 재귀 방식으로도 구현하기 위해 재귀 관련 몇몇 예제들을 코딩하여 재귀에 대해서 이해하려고 노력함

- DFS를 재귀로 구현

```python
def recursive_dfs(graph,node,visited):
    if node not in visited:                 # 현재 노드가 방문 리스트에 없으면 실행
        visited.append(node)                # 현재 노드를 방문 리스트에 추가가
        for neighbor in graph[node]:        # 현재 노드에 연결된 모든 이웃 노드 탐색
            recursive_dfs(graph,neighbor,visited)   # 이웃 노드에 대해 재귀 호출
    return visited                          # 방문 완료된 리스트 반환

print("재귀 DFS:", recursive_dfs(my_graph,1,[]))
```

재귀 호출: 자기 자신으로 호출

```
재귀 DFS: [1, 2, 5, 9, 6, 3, 7, 10, 8, 4]
```

- for 문 안의 recursive_dfs가 재귀 호출
- 재귀 호출을 통해 DFS 방식으로 깊이 우선 탐색 수행
- 방문 리스트를 사용해 중복 방문 방지
- 재귀 호출이 끝나면 함수가 하나씩 스택에서 제거됨

- DFS를 구현할 때의 방식 선택

```python
def recursive_dfs(graph, node, visited):
    if node not in visited:
        visited.append(node)
        for neighbor in graph[node]:
            recursive_dfs(graph, neighbor, visited)
    return visited

print("재귀 DFS:", recursive_dfs(my_graph, 1, []))
```

```python
def stack_dfs(graph, start):
    visited = []
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.append(node)
            stack.extend(graph[node])  # 이웃

    return visited

print("스택 DFS:", stack_dfs(my_graph, 1))
```

❑ 재귀 DFS: 코드가 간결하고 직관적인게 중요할 때 사용
- 장점: 코드가 간결하고 직관적 / 단점: 너무 깊어지면 스택 오버 플로우 위험

❑ 스택 DFS: 깊은 탐색이 필요하거나 성능과 안정성이 중요할 때 사용
- 장점: 스택을 직접 관리하여 스택 오버플로우 방지
- 단점: 코드가 다소 길어지고 복잡해질 수 있음

Breadth-First-Search

- BFS 구현(큐, 재귀)

```python
def my_bfs(start_node,graph):
    queue = list()
    visited = list()
    queue.append(start_node)

    while queue:
        node = queue.pop(0)
        if node not in visited:
            queue+=(graph[node])
            visited.append(node)
            print("queue:",queue)
            print("visited:",visited)
    return visited

print("BFS:",my_bfs(1,my_graph))
```

```
BFS: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
def my_bfs_recursive(queue, visited, graph):
    if not queue:  # 큐가 비면 종료
        return visited

    node = queue.pop(0)  # 큐의 맨 앞 노드 꺼내기
    if node not in visited:
        visited.append(node)  # 방문 처리
        queue.extend(graph[node])  # 인접 노드 추가

    return my_bfs_recursive(queue, visited, graph)  # 재귀 호출

# BFS 실행
start_node = 1
print("BFS (Recursive):", my_bfs_recursive([start_node], [], my_graph))
```

✓ 0.0s

```
BFS (Recursive): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

❑ BFS를 재귀로 구현하는 것 => 비효율적

- BFS는 한 번에 여러 개의 노드를 확장하면서 진행하는데 재귀는 한 개의 노드만 처리 가능해서 BFS의 특성과 맞지 않음
- 따라서 재귀를 사용하면 불필요한 함수 호출이 많아져 속도가 느려지고 성능 저하

Breadth-First-Search

- 문제1: 이 그래프를 보고 DFS와 BFS를 구현하라

plaintext

- 문제2: 위에서 구현한 DFS를 재귀를 통해 구현하라

Thank you for listening!