

PART 1 알고리즘 기초

1-1 알고리즘 이란?

- 어떠한 문제를 해결하기 위해 정해놓은 일련의 절차
- 올바른 알고리즘
 - 어떠한 경우에도 실행결과가 똑같이 나오는 것

1-1-1. 조건문과 분기

· 분기

— 프로그램의 실행흐름을 다른 곳으로 변경하는 명령

1-1-2. 순서도 기호

· 데이터



· 선



· 처리



· 단말



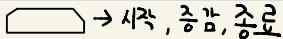
· 미리정의한 처리



· 판단



· 루프 범위



1-2 반복하는 알고리즘

· 반복구조 - 루프 (loop)

— 어떤 조건이 성립하는 동안 반복해서 처리

1-2-1 while 문

· 사전 판단 반복구조

1-2-2 for 문

· 변수가 하나만 있을 때

— 카운터용 변수

· 반복을 제어할 때 사용

/ - 2 - 3. 다중루프

- 반복문 중첩
- 변수
 - 파이썬: 객체에 연결된 이름
 - 전역 변수
 - 함수 외부에서 정의
 - 지역 변수
 - 함수 내부에서 정의

PART 2 기본 자료구조와 배열

2-1. 자료구조와 배열

- 배열
 - 물음 단위로 값을 저장
 - 원소
 - 저장된 객체
- 인덱스
 - 원소들의 순서
- 언팩
 - 배열의 원솟값들을 풀어 여러 변수에 대입

2-1-1. 인덱스

- 배열 객체 [i]
- 새로운 값 정의 불가능

2-1-2. 슬라이스

- 배열 객체 [시작:종료:증감]
- 값의 변경 유/우
 - 이터터 불 → 불가능
 - 뮤타불 → 가능
- 자료구조
 - 논리적인 관계로 이루어진 데이터 구성

2-2 배열이란?

· 스캔

- 배열 원소를 차례대로 살펴보는 방식

· 모듈 - py

- 하나의 스크립트 프로그램

- if __name__ == '__main__':

· 모듈이 직접 실행될 때 → TRUE

· 모듈이 임포트 될 때 → FALSE

· enumerate (이터레이션)

→ (인덱스, 값) 반복 → 투플

→ enumerate (이터레이션, i)

· (인덱스, 값) ⇒ 인덱스: i 부터 시작

· 이터레이션

- 반복 가능한 자료형

· 문자열, 리스트, 투플, 집합, 딕셔너리

2-2-1. 리스트 역순으로 정렬

· 정렬 값 반환

- 리스트 객체. reverse()

· 역순 정렬 리스트 생성

- = list(reversed(리스트 객체))

· 함수 사이에 인수 주고 받기

- 인수 - 유터보

· 함수 안에서 매개변수값 변경 → 객체 자체를 업데이트

- 인수 - 이유터보

· 함수 안에서 매개변수 값 변경 → 실제 인수에 영향 X

· 리스트의 원소와 복사

- 파일 → 리스트 내 각 원소들의 자료형을 다르게 할 수 있음

- 복사

· 얕은 복사 : 객체가 참조 자료형의 멤버를 포함 → 참조 값만 복사

· 깊은 복사 (=전체복사) : 참조 값과 참조하는 객체 자체를 복사 → 객체가 갖고 있는 모든 것 복사

PART 3 검색 알고리즘

3-1 검색 알고리즘 이란?

- 데이터 집합에서 원하는 값을 가진 원소를 찾아 내는 방식

3-1-1 검색과 키

- 키

- 검색조건이 주목한 어떤 항목

- 주로 데이터의 일부

3-1-2 검색의 종류

- 1) 배열 검색

- 선형검색 : 우측위로 늘어놓은 데이터 집합

- 이진검색 : 일정한 규칙으로 늘어놓은 데이터 집합

- 해시법 : 추가·삭제가 자주 발생하는 데이터집합

- 체인법, 오픈 주소법

- 2) 연결 리스트 검색

- 3) 이진 검색트리 검색

3-2 선형검색

- 순차검색

- 직선(선형) 모양으로 늘어선 배열에서 검색

- → 원하는 키값을 가진 원소를 찾을 때 까지 맨 앞부터 순서대로 검색

- 원소의 값이 정렬되지 않은 배열에서 검색에 사용되는 유일한 방법

3-2-1 종료조건

- . 검색 실패

- 검색할 값을 찾지 못하고 맨 끝까지 간 경우

- . 검색 성공

- 검색할 값과 같은 원소를 찾는 경우

3-2-2 보초법

- 종료조건의 검사 이용(시간)을 뺀으로 줄임

- (원래 데이터) + 보초 (= 키값)

- > 인덱스 == len(원래데이터+보초) ⇒ 검색 실패

3-3 이진검색 - binary search

- 정렬된 배열 데이터 사용 - 오름차순
- 선행검색 대비 빠른 속도

3-3-1. 이진검색

· 반복

- 1) 배열의 중간값과 key 값의 크기 비교

· 중간값 : 2진수 \Rightarrow 10진수

· 중간값 < key

- 배열 \Rightarrow [중간값+1 ~ 끝값]

· 중간값 > key

- 배열 \Rightarrow [시작값 ~ 중간값-1]

- 종료조건

· 중간값과 key 값이 일치할 때

· 검색범위가 더 이상 없을 경우

· 비교 횟수

- 평균 : $\log_2 n$

3-3-2. 복잡도

- 알고리즘의 성능 평가 기준

시간 복잡도

- 실행하는데 걸리는 시간

공간 복잡도

- 메모리와 파일 공간의 필요한 정도

- 복잡도 표기 - Big O

- $O()$

• $| \log n \ n \ n \log n \ n^t \ n^3 \ n^k \ 2^n$
작다 \longleftrightarrow 크다

$$\cdot O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

3-4 해시법

- 정색

- 데이터의 추가·삭제에 효율적

3-4-1. 정렬된 배열에서 원소 추가하기

- 1) 추가할 자리를 찾기 위해 이진 정색법

- 2) 추가할 자리 이후의 모든 원소를 한 칸씩 뒤로 이동

⇒ O(n) 의 복잡도 소요

- 3) 추가할 자리에 새로운 원소 추가

3-4-2. 해시법 - hashing

- 데이터를 저장할 위치 = 인덱스를 간단한 연산으로 구함

- 해시값

- 데이터에 접근할 때 기준이 됨

- 해시 함수

- 키를 해시값으로 변환하는 과정

3-4-3. 해시충돌

- 저장 할 버킷(해시값)이 충복되는 현상

- 키 : 해시값 \Leftrightarrow n:1 관계

- 대처법

- 체인 법

- 해시값이 같은 원소를 연결 리스트로 관리

- 오픈 주소법

- 빈 버킷을 찾을 때 까지 해시를 반복

3-4-4. 체인법 (=오픈 해시법)

- 해시값이 같은 데이터 저장하기

- 연결 리스트에 의해 체인 모양으로 연결

- 1) Node 클래스 만들기

- 개별 버킷을 나타냄

- Key : 키(임의의 자료형)

- Value : 값(임의의 자료형)

- Next : 뒤쪽 노드를 참조(Node형)

· 2) ChainedHash 클래스 만들기

- Capacity : 해시테이블의 크기 (배열 table 의 원소수)
 - 크기는 소수를 선호한다.
- table : 해시테이블을 저장하는 list 형 배열
- hash_value() 함수
 - 인수 key에 대응하는 해시값 반환
 - 해시값 : $key \% capacity$
 - key가 int 형인 경우와 아닌 경우를 구분

· 3) search() 함수

- 1. 해시함수를 사용하여 key \rightarrow 해시값 번환
- 2. 해시값을 인덱스로 하는 버킷에 주목
- 3. 버킷이 참조하는 연결리스트를 스캔
 - key와 같은 값 발견 \rightarrow 검색성공
 - 맨 끝까지 발견 X \rightarrow 검색실패

· 4) 원소를 추가하는 add() 함수

- 1. 해시함수를 사용하여 key \rightarrow 해시값 번환
- 2. 해시값을 인덱스로 하는 버킷에 주목
- 3. 버킷이 참조하는 연결리스트 선형검색
 - key와 같은 값 발견 \rightarrow 추가실패
 - 맨 끝까지 발견 X \rightarrow 추가 성공

· 5) 원소를 삭제하는 remove() 함수

- 1. 해시함수를 사용하여 key \rightarrow 해시값 번환
- 2. 해시값을 인덱스로 하는 버킷에 주목
- 3. 버킷이 참조하는 연결리스트 선형검색
 - key와 같은 값 발견 \rightarrow 값 삭제
 - 맨 끝까지 발견 X \rightarrow 삭제 실패

· 6) 원소를 출력하는 dump() 함수

- 해시 table의 내용을 한꺼번에 출력

3-4-5. 오픈 주소법 (= 단한 해시법)

- 충돌 발생 → 재해시를 수행하여 빈 버킷을 찾는 방법

- 1) 원소 추가하기

- 빈 버킷을 찾을 때까지 해시함수 재정의 후 재해시

- 2) 원소 삭제하기

- 각 버킷에 속성 부여

- 문자 → 데이터가 저장되어 있음

- - → 비어 있음

- * → 삭제 완료

- 3) 원소 검색하기

- 부여된 속성에 의해

- 4) 원소 출력하기 - dump()

PART4 스택과 큐

- 데이터를 임시 저장하는 기본 자료구조

4-1 스택이란?

4-1-1. 스택 알아보기

- LIFO 방식

- 데이터의 입력과 출력 순서

- 후입 선출

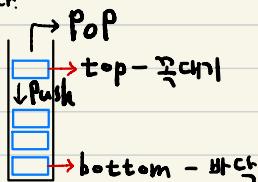
- 가장 나중에 넣은 것을 먼저 꺼낸다.

- Pop

- 스택에서 데이터를 꺼내는 작업

- Push

- 스택에 데이터를 넣는 작업



4-1-2. 스택 구현하기

- 크기가 결정된 고정길이 스택 만들기

- 스택 배열 - `stk`

- 데이터를 저장하는 스택 본체인 `list`형 배열

• 스택 크기 - Capacity

- 스택의 최대 크기를 나타내는 int형 정수
- $= \text{len}(\text{stk})$

• 스택 포인터 - ptr

- 스택에 쌓여 있는 데이터의 개수를 나타내는 정수

• 예외 처리 클래스

- Empty

- 비어 있는 스택에 pop 또는 peek(들여다보기) 할 때 예외 처리

- Full

- 가득 차 있는 스택에 push 할 때 예외 처리

• 초기화 함수 - `__init__()`

- 스택 본체 : `stk = [None] * Capacity`
- 스택의 크기 : `Capacity`
- 스택 포인터 : `ptr = 0`

• 스택 내의 데이터 개수 함수 - `__len__()`

- `return self.ptr`

• 스택이 비어 있는지 확인하는 함수 - `is_empty()`

- `return self.ptr == 0`

• 스택이 가득 차 있는지 확인하는 함수 - `is_full()`

- `return self.ptr >= self.capacity`

• 스택에 데이터를 push 하는 함수 - `push(self, value)`

- If 스택이 가득 차 있는 경우

- 예외 처리

- else

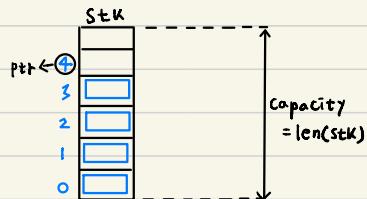
- 전달 받은 value \rightarrow 스택에 저장

- 포인터 +1

• 스택에 있는 데이터를 pop 하는 함수 - `pop(self)`

- If 스택이 비어 있는 경우

- 예외 처리



- else

- 포인터 - i
- return self.stk[self.ptr]
- 데이터를 들여다 보는 함수 - peek(self)
 - If 스택이 비어 있는 경우
 - 예외처리

- else

- return self.stk[self.ptr-1]
- 스택의 모든 데이터 삭제 함수 - clear(self)
 - self.ptr = 0
- 데이터를 검색하는 함수 - find(self, value)
 - 꼭대기 쪽부터 선형검색
 - 검색실패시 → return -1
- 데이터 개수를 세는 함수 - count(self, value)
 - 바닥쪽 부터 선형검색
 - 정색성공시 결과값 +1
- 데이터가 포함되어 있는지 확인하는 함수 - __contains__(self, value)
 - return self.count(value)
 - 있으면 → True 반환
 - 없으면 → False 반환
- 스택의 모든 데이터를 출력하는 함수 - dump(self)
 - If self.is_empty()
 - print('스택이 비어 있습니다.')
 - else
 - print(self.stk[:self.ptr])

- 딘더(dunder) 함수

- __ 항수 이름 __ → double underscore
- __ 항수 이름 __ () → 항수 이름 (액체)
로 간단히 표현

— 데크 - deque

· 맨 앞과 맨 끝 양쪽에서 원소를 추가·삭제하는 구조

4-2 큐란?

4-2-1. 큐 알아보기

· FIFO 방식

— 데이터의 입력과 출력순서

— 선입선출

• 가장 먼저 넣은 데이터를 가장 먼저 꺼내는 방식

· enqueue - 인큐

— 큐에 데이터를 추가하는 작업

· dequeue - 디큐

— 큐에 있는 데이터를 꺼내는 작업

• front - 프런트

— 데이터를 꺼내는 쪽

• rear - 리어

— 데이터를 넣는 쪽

4-2-2. 배열로 큐 구현하기

• enqueue

— 처리의 복잡도 - $\bigcirc(1)$

• 비고적 적은 비용

• dequeue

— 처리의 복잡도 - $\bigcirc(n)$

• 맨 뒤 남아있는 원소들을 한 칸씩 앞으로 옮기는 과정

• \rightarrow 프로그램의 효율성 기대 X

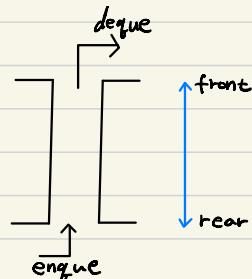
4-2-3. 링 버퍼 와 큐

• dequeue 할 때 배열안의 원소를 옮기지 않음

• front, rear \rightarrow 논리적인 순서

\rightarrow 물리적인 순서 X

• 처리의 복잡도 - $\bigcirc(1)$



4-2-4. 링 버퍼로 큐 구현하기

- 예외 처리 클래스

- Empty (Exception)

- 비어 있는 큐에 dequeue(), peek() 함수 호출 시

- Full (Exception)

- 가득 차 있는 큐에 enqueue() 함수 호출 시

- 초기화 함수 - `__init__(self, capacity)`

- que : 큐의 list 형 배열

- capacity : 큐의 최대 크기, int 형

- front, rear : 맨 앞, 뒤 원소를 나타내는 인덱스

- no : 현재 큐에 쌓여 있는 데이터 개수

- 큐에 있는 데이터 개수 반환 - `__len__(self)`

- return self.no

- 큐가 비어 있는지 확인 - `is_empty(self)`

- return self.no <= 0

- 큐가 가득 차 있는지 확인 - `is_full(self)`

- return self.no >= self.capacity

- 데이터를 큐에 넣는 함수 - `enqueue(self, x)`

- if 큐가 가득 차 있는 경우

- 예외 처리

- else

- que[rear]에 데이터 넣음

- rear, no +

- if rear == capacity

- rear = 0

- 데이터를 큐에서 꺼내는 함수 - `dequeue(self)`

- if 큐가 비어 있을 경우

- 예외 처리

- else

- que[front]에 저장된 값을 꺼내어 x에 저장

- front + 1, no - 1
- if front == capacity
 - front = 0
- return X
 - X 값 반환
- 데이터를 들여다 보는 함수 - peek(self)
 - if 큐가 비어 있는 경우
 - 예외처리
 - else
 - return self.que[self.front]
 - 맨 앞의 데이터 반환
- 검색하는 함수 - find (self, value)
 - 맨 앞부터 선형검색
 - 맨 앞 = front
 - 인덱스 i → (i + front) % Capacity
 - 검색 실패시 → return -1
 - 데이터 개수를 세는 함수 - count (self, value)
 - 맨 앞부터 선형검색
 - 맨 앞 = front
 - 인덱스 i → (i + front) % Capacity
 - 정렬성증시 결과값 + 1
 - 데이터가 큐에 있는지 확인하는 함수 - __contains (self, value)
 - return self.count (value)
 - 큐의 전체 원소를 삭제하는 함수 - clear (self)
 - self.no = self.front = self.rear = 0
 - enqueue, dequeue는 no, front, rear의 값을 바탕으로 수행
 - ⇒ 실제 que의 원소값 변경 X
 - 큐의 전체 데이터를 출력하는 함수 - dump (self)
 - if self.is_empty()
 - print('큐가 비어있습니다.')
 - else
 - print (self.que[:self.ptr])

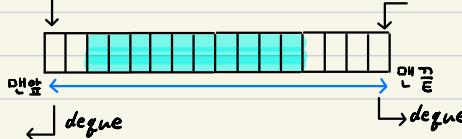
- else

```
. print(self.que[(i+self.front)%self.capacity], end=' ')
```

- deque - 양방향 대기열 덱

- 맨 앞과 맨 끝 양쪽에서 원소를 추가·삭제 하는 구조

. enqueue



- 링 버퍼의 활용

- 가장 최근에 들어온 n 개 만 저장하고 나머지는 버림.

. 봉법

- 크기가 n 인 링 버퍼 생성

- $Cnt = 0$

- 정수를 입력 받은 개수

- 반복

- 링 버퍼 $[Cnt \% n]$ = 입력값

- $Cnt + 1$ → $Cnt \geq n \Rightarrow$ 더이 쓰임

- If 입력 == N

- break

- 링버퍼에 남아있는 수 출력

- $i = Cnt - 1$

- if $i < 0$

- $i = 0$

- 반복 $i < Cnt$

- $print(i+1\text{번째} = \text{링버퍼}[i \% n])$

- $i + 1$

입력 순서대로

0부터 차례대로