

파이썬으로 자료구조 잡기

DATA STRUCTURES WITH PYTHON

Chan-Su Shin

Divison of Computer Engineering
Hankuk University of Foreign Studies



chanhsu 4.15.2023

자료구조

Last updated, July 1 2025

DATA STRUCTURES WITH PYTHON¹



2



3

컴퓨터공학부

¹ 한국외대 컴퓨터공학부의 강의 내용을 포함해 다양한 자료구조를 정리한 것임.

² 신찬수, chansu@gmail.com, 이 자료는 한국외국어대학교 컴퓨터공학부 신찬수 교수의 강의교재 중 일부임. 저자의 허락을 받고, 출처를 밝히고, 수정하지 않고, 상업적 목적이 없는 개인 학습용으로만 사용해야 함.

³ 일부 내용은 신찬수의 유튜브 채널 ([Chan-Su Shin](#))에 동영상으로 올려져 있음.

* 이 자료에서 인용한 이미지와 일부 내용은 wikipedia, unsplash 등에 공개된 것임.

[저자 생각] 정식 출판을 염두에 두지 않고 정리한 내용으로 대충 그린 그림과 불친절한 설명에 5년 넘게 수정했지만 아직도 오타와 오류가 있습니다. 그럼에도 시간과 정성을 들여 5년 넘게 숙성해 얻은 결과물로 자료구조와 알고리즘에 조금이라도 다가오고 싶은 분들에게 작은 도움이 되길 바랍니다.

[표지 이미지 - Cover Image] King Fisher by Chansu

LIFO (Last In, First Out), called a "stack".

FIFO (First In, First Out), called a "queue".

FISH (First In, Still Here), called a "jam in printer".

FIGL (First In, Got Lost), called a "bureaucracy".

FILO (First In, Last Out), called a "_____".

somewhere on the internet

자료구조 목차

- 시작하기: 자료구조와 파이썬 관련 교재 소개 (p. 7)

- 1. Algorithm, Data Structure, Time Complexity (p. 14)
 - a. 가상 머신, 가상 언어, 가상 코드
 - b. 시간, 공간 복잡도
 - c. Big-O 표기법

- 2. Sequential Structures (배열 기반 자료구조) (p. 30)
 - a. 동적 배열 (dynamic array)
 - b. 스택 (stack), 큐 (queue), 디큐 (dequeue)

- 3. Linked List (연결 리스트) (p. 56)
 - a. 한방향 연결 리스트 (Singly Linked List)
 - b. 양방향 연결 리스트 (Circular Doubly Linked List)

- 4. Hash Table (해시 테이블) (p. 68)
 - a. 해시 함수 (Hash Function)
 - b. 충돌과 충돌 회피법 (Collision, Collision Resolution)
 - Open addressing vs. Chaining
 - Amortized 수행시간 분석: Charging 방법

- 5. Tree (트리) (p. 104)
 - a. 우선순위큐: 힙 (Priority Queue: Heap)
 - 힙 연산 및 힙 정렬 (Heap Sort)
 - b. [고급] 힙 모양의 이진트리
 - 범위 트리 (Segment Tree)
 - BIT (Binary Indexed Tree, Fenwick Tree)
 - c. 이진트리(Binary Tree)
 - 순회(traversal): preorder, inorder, postorder

- d. 이진탐색트리 (Binary Search Tree)
 - 삽입, 삭제 연산
- e. 균형이진탐색트리 (Balanced Binary Search Tree)
 - AVL 트리
 - Red-Black 트리와 2-3-4 트리
 - Splay 트리

6. Graph (그래프) (p. 174)

- a. 그래프 표현법과 기본 연산
 - 인접 행렬, 인접 리스트, 기본 연산의 복잡도
- b. 그래프 순회법(traversal)
 - DFS, BFS, 응용
- c. 최단 경로 알고리즘
 - Bellman-Ford, Dijkstra 알고리즘
- 다른 그래프 알고리즘은 알고리즘 과목에서 다룸!

7. [고급 자료구조] 기타 유용한 자료 구조 (p. 196)

- a. Union-Find 자료구조
 - 집합 연산 지원 (멤버십: find, 합집합: union)
- b. van Emde Boas 자료구조
 - 정수 key 값에 대한 매우 빠른 priority 큐 연산 제공
- c. Suffix Array 자료구조
 - 문자열 패턴 탐색 (string search)에 사용되는 자료구조
- d. Range 트리 자료구조
 - 점(point)을 다루는 (기하)문제에서 사용되는 자료구조

시작하기

1. 왜 파이썬으로 알고리즘을 코딩하면 좋을까?
 - a. 직관적이라 알고리즘의 pseudo 코드와 매우 유사해 생산성이 높다
 - b. 내부, 외부 모듈이 다양하고 확장성이 좋아 응용 분야의 프로그래밍에 유리하다
 - 데이터베이스 응용, 데이터 분석/처리/시각화, 머신러닝, 웹 프로그래밍
 - c. 학습에 도움이 되는 (무료/유료) 동영상 자료가 넘쳐난다 (아래 자료들 참고)
 2. 단점은 없나? (당연히 있다. 세상에 공짜 점심은 없다)
 - a. 인터프리터 언어이기에 출 단위로 컴파일하고 실행하기에 수행 시간이 느리다
 - C, C++: source code (xxx.c, xxx.cpp) → (compiler) → 실행 파일 (xxx.exe)
 1. CPU가 exe 파일을 직접 실행하기에 상대적으로 빠름
 - Python: source code (xxx.py) → (interpreter) → 출 단위로 interpreter가 실행
 1. CPU는 interpreter를 실행하고 interpreter가 출 단위로 실행하는 형식으로 상대적으로 느림
 - java: compiler 방식과 interpreter 방식을 혼용해 동작하기에 둘 사이에 위치
 - b. 비교 예: 2048 x 2048 크기의 2차원 행렬 두 개를 곱하는 3중 for 루프를 C, java, Python으로 구현해 CPU 시간을 비교한 내용
 - [medium.com에 게시된 글에서 인용](https://medium.com/@georgecushen/explaining-the-speed-difference-between-c-and-python-when-solving-the-same-problem-103f3a2a2a) (C 코드는 -O3 옵션 사용)

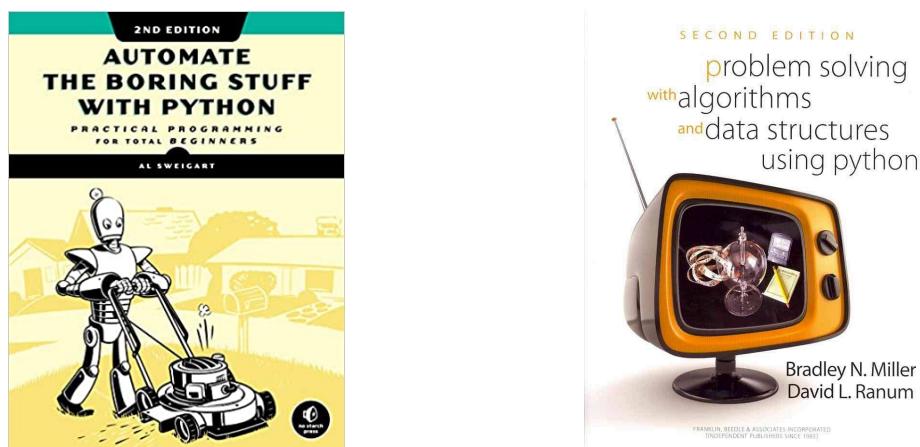
Language	Running time	Absolute Speed Up
Python	2821.108	1
Java	84.612	33.34
C	49.958	56.47
 - c. 그럼 빠르게 실행할 수 있는 파이썬 코드를 작성하는 건 어렵나?
 - Python3를 주로 사용하는데, 이 버전의 interpreter는 CPython인데, 새로운 interpreter인 PyPy3를 사용하면 많은 경우에 더 빠른 실행 시간을 얻을 수 있다
 - 그 이유는 자주 사용되는 코드 조각을 cache에 저장해서 다시 interpreter를 사용하지 않고 재사용하기 때문이다. (대신 메모리는 더 많이 사용) 따라서 간단한 코드는 Python3로, 반복되는 부분이 많은 복잡한 알고리즘 코드는 PyPy3로 작성하면 된다
 - PyPy3: <https://www.pypp.org/> 참조 .교재에 나오는 모든 코드는 Python3.6x를 기준으로 작성되었다

3. 충분히(!) 참고할 가치가 있는 자료들

- a. 공식 파이썬 홈페이지: www.python.org
- b. 동영상 무료 강의 사이트(기초):
 - 생활코딩 - Python/Ruby 강의: <https://opentutorials.org/course/1750>
 - programmers - Python 강의: <https://programmers.co.kr/learn/courses/2>
- c. 영어 튜토리얼 사이트:
 - tutorialspoint: <https://www.tutorialspoint.com/python3/index.htm>
 - diveintopython3: <http://www.diveintopython3.net/>

4. 참고용 교재 (Python + Data Structures)

- a. [Python] 점프 투 파이썬, 박응용 지음, 이지스퍼블리싱 (다음 링크에서 책의 내용을 모두 읽을 수 있음: <https://wikidocs.net/4321>)
- b. [Python] [Automate The Boring Stuff with Python](#), Al Sweigart (현존하는 가장 좋은 Python 책 중 하나)



- c. [자료구조] [Problem Solving with Algorithms and Data Structures Using Python](#), Bradley N. Miller and David L. Ranum (2013 version is available online [here](#), 웹 사이트 [here](#))
- d. [자료구조-한글책] [파이썬과 함께하는 자료구조의 이해](#), 양성봉지음, 생능출판사 (기초부터 중간 수준까지 비교적 잘 정리된 책)
- e. [자료구조] [Data Structures and Algorithms in Python](#), paperback, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser (2016) (산다면 paperback으로, hardcover는 너무 너무 비쌈)

f. [자료구조+알고리즘의 바이블 ] [Introduction to Algorithms](#), T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, MIT Press (3rd Edition) (기초부터 높은 수준까지 친절하게 정리된 알고리즘의 바이블! 국내 번역본도 출간되었지만 비쌈)

g. [코딩 인터뷰 문제 모음 ]

- [Cracking the Coding Interview: 189 Programming Questions and Solutions](#) (6th Edition), Gayle Laakmann McDowell
- [Elements of Programming Interviews in Python: The Insiders' Guide](#) 2016, Adnan Aziz, Tsung-Hsien Lee, Amit Prakash
- 구글, 아마존 등을 포함한 유명 해외 기업체 코딩 문제를 모아 단순한 알고리즘부터 가장 빠른 알고리즘까지 다양한 방법으로 해결하는 과정을 친절하게 설명해 놓은 책. 코드는 java로 되어 있음. 한글 번역본도 출간되어 있지만, 비쌈
- 이와 유사한 문제+정답 모음집 (무료) PDF 파일도 있음
 - <https://www.programcreek.com/wp-content/uploads/2012/11/coding-interview-in-java.pdf>

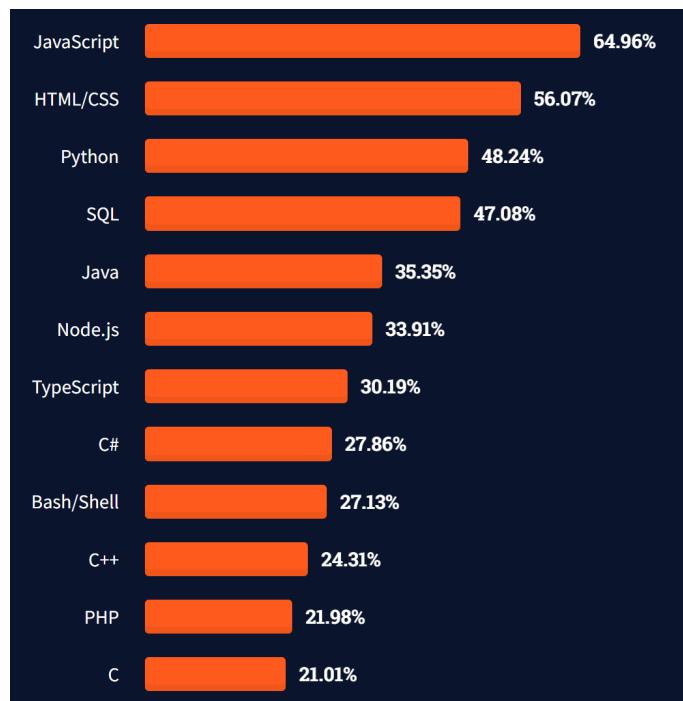
5. Python은?

- a. 1989년에 [Guido van Rossum](#)에 의해 설계, 제안
- b. high-level, general-purpose, interpreted, object-oriented, dynamic language
고수준 -- 범용성 -- 대화형 -- 객체지향 -- 동적타입을 지원하는 언어
- c. [Wikipedia](#)에서의 설명 참조
- d. 어느 정도 인기가 있을까?
 - TOIBE 프로그래밍 언어 랭킹, Stackoverflow Survey 2021, IEEE Spectrum 프로그래밍 언어 랭킹 참고

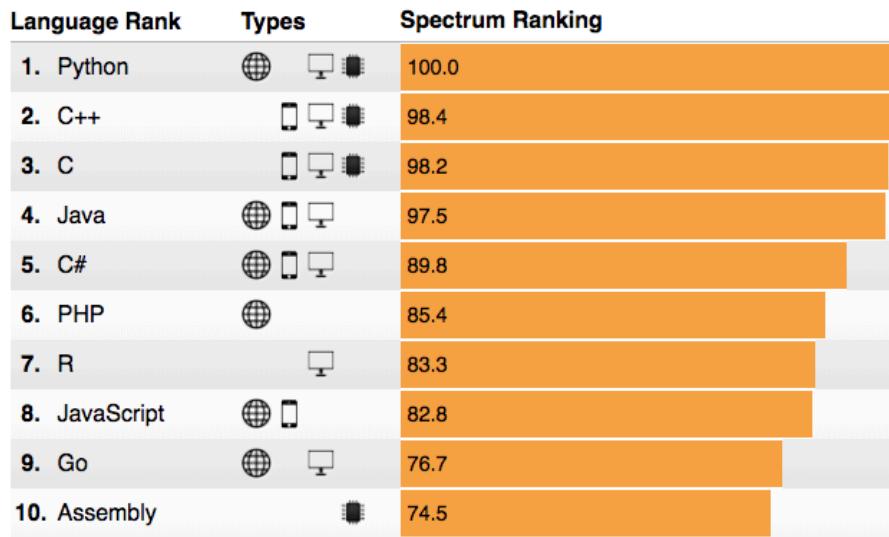
TOIBE Index for Programming Languages (2022-2023)

Dec 2023	Dec 2022	Change	Programming Language	Ratings	Change
1	1		Python	13.86%	-2.80%
2	2		C	11.44%	-5.12%
3	3		C++	10.01%	-1.92%
4	4		Java	7.99%	-3.83%
5	5		C#	7.30%	+2.38%
6	7	▲	JavaScript	2.90%	-0.30%
7	10	▲	PHP	2.01%	+0.39%
8	6	▼	Visual Basic	1.82%	-2.12%
9	8	▼	SQL	1.61%	-0.61%
10	9	▼	Assembly language	1.11%	-0.76%

Stackoverflow 2021 survey (전 세계 개발자 5만여 명을 포함한 8만여명의 조사 결과)



IEEE Spectrum 2018 조사



5. Python Version 3.6.x 이상 설치 (강의와 실습용 버전)

- a. Window 운영체제 기준으로 설명하고 실습함
- b. [python download site](#) - 2022년 3월 기준 3.10.2이 최신 버전임
- c. 다운 받아 설치하는데, Install Now 화면의 가장 아래 두 개의 체크박스 모두 체크함
 - Install launchers for all users (recommended)
 - Add Python 3.6 to PATH

6. 코딩 환경:

- a. 코드 실행을 위한 터미널: Powershell(추천), cmd shell (둘 다 Windows 기본 내장)
- b. 에디터 + 터미널 (IDE - 통합 개발 환경)
 - IDLE: python 패키지에서 제공하는 내장된 IDE
 - **Anaconda**: Jupyter Notebook 이란 웹 코드 에디터 제공
 - 1. Jupyter Notebook 이란 코드 에디터도 함께 제공
 - **vscord** (Visual Studio Code) [[download](#)] atom과 유사한 IDE (강력 추천!)
 - **atom** [[download](#)] (추천)

7. 강의노트 기호

- a. [💡] - 꽤 중요한 설명, [?] - 점검용 질문, [👉] - 강의내용에 따른 실습 또는 구현문제
- b. [⌚] - 알고리즘 관련 (매우 유용한) 퍼즐
- c. [🎙 인터뷰 문제] - IT 기업 사원 채용 인터뷰에서 등장하는 알고리즘 문제
- d. [💡 코딩 대회 문제] - 프로그래밍 대회에서 등장하는 꽤 난이도 있는 문제

8. [👉] [해보기 1]

- a. 윈도우 하단 왼쪽의 🔎에 Powershell 입력해서 Windows Powershell 실행
- b. Powershell 창에서 python 입력하고 엔터
- c. print("hello") 입력 후 엔터 (결과 확인)
- d. exit() 또는 Ctrl+C 입력 후 python 종료

9. [💪] [해보기 2]

- a. 적절한 장소에 **DataStructures**이라는 과목용 디렉토리 생성
- b. 디렉토리 창의 파일 메뉴에서 Windows Powershell 열기 클릭 (powershell이 열림)
- c. Powershell 창에서 notepad 등의 편집기 입력 후 실행
- d. notepad의 새 파일에 다음 두 줄을 입력후 hello.py 파일 이름으로 저장

```
import sys
print("hello", "python", sys.version)
```

- e. Powershell 창에서 python hello.py 입력 후 결과 확인

10. 만약, powershell에서 명령을 인식하지 못하는 경우 (해당 프로그램의 위치를 윈도우가 모름)

- a. 예를 들어, notepad를 제대로 인식하지 못하는 경우:
- b. 내 PC 우클릭 → 속성 → 고급 시스템 설정 → 환경변수 → 사용자 변수 리스트에서 path 더블 클릭 또는 새로 만들기 → notepad.exe가 있는 디렉토리 path 입력

11. IDE: **vscode** (강추:  [Download Visual Studio Code - Mac, Linux, Windows](#) 다운)

- a. extension에서 Python extension을 검색해서 install해도 되고, vscode가 필요한 extension을 추천할 때 설치해도 됨
- b. File → New File 선택해 hello.py 파일 새로 만든 후, 첫 줄에 print("hello") 입력
- c. Terminal → New Terminal 선택하면 cmd 또는 PowerShell 터미널 생성됨. 터미널에 명령 python hello.py 입력하여 실행해 봄
- d. 장점: 직관적이며 다양한 extension을 설치해서 사용할 수 있는 인터페이스, 거의 모든 언어 지원, git 명령 지원, Jupyter 에디터도 지원

13. IDE: Anaconda (<https://www.anaconda.com/download/>에서 다운)

- a. Jupyter Notebook 이란 코드 에디터도 함께 설치됨
 - interactive/script 모드 동시 제공
 - .ipynb 확장자로 save/load 가능

14. IDE: Atom (<https://atom.io/>에서 다운 받아 설치)

- a. File → Settings → Install
 - platformio-ide-terminal 과 script 두 패키지 검색 후 install
- b. platformio-ide-terminal 패키지:
 - 하단 왼쪽에 + 버튼을 누르면 command 창이 열림 (Powershell과 유사한 창)
 - python 코드를 에디터 창에서 작성한 후, command 창에서 실행하면 편리함
- c. script 패키지
 - Shift + Ctrl + b 를 누르면 현재 편집되는 python 코드를 실행시켜 줌

15. 기본 shell 명령들

- a. **dir** or **ls**: 디렉토리에 있는 다른 디렉토리와 파일을 화면에 출력함
- b. **cd A**: 현재 디렉토리에서 A 디렉토리로 이동함
 - . 현재 디렉토리, .. 부모 디렉토리, ~ 루트 디렉토리)
- c. **mkdir A**: 새로운 디렉토리 A를 현재 디렉토리 안에 새로 생성함
- d. **move A B**: 파일 A를 새로운 디렉토리 또는 새로운 파일 B로 바꿈 (rename/move)

- e. **copy** A B: 파일 또는 디렉토리 A를 파일 또는 디렉토리 B로 복사함
 - f. **del** A or **rm** A: 파일 A를 지움 (A가 디렉토리라면 A의 내용이 없어야 함)
 - g. **more** A: 파일 A의 내용(텍스트)을 화면에 출력함
 - h. reading:
<http://www.cs.princeton.edu/courses/archive/spr05/cos126/cmd-prompt.html>
 - i. 위/아래 화살표: 전/후 명령어 리스트 탐색, 텁+완성 기능
16. 강의 노트에 등장하는 외부 이미지나 자료 등은 wikipedia 등의 GFDL 또는 CC-BY-SA license 조건에서 인용된 것이다

자료구조

1. Data Structures, Algorithms, Time Complexity

- 🔥 알고리즘(algorithm)

- 알고리즘의 어원은 9세기 페르시아(이란-이라크) 수학자 Al-Khwarizmi의 라틴어 이름인 **algorismus**와 수를 나타내는 그리스어 **arithmos**가 섞여 만들어졌다는 게 정설이다[[Wiki](#)]
- 아래 사진은 사우디아라비아 Jeddah에 위치한 **KAUST** 대학의 컴퓨터과학과가 위치한 빌딩의 이름 (직접 촬영했음!)



- 알고리즘은 문제의 입력(input)을 수학적이고 논리적으로 정의된 연산과정을 거쳐 원하는 출력(output)으로 변환/계산(computation)하는 절차이고, 이 절차를 C나 Python과 같은 언어로 표현하면 프로그램(program) 또는 코드(code)가 된다
 - 입력은 배열(array(C) 또는 list(Python)), 연결리스트(linked list), 트리(tree), 해시테이블(hash table), 그래프(graph)와 같은 자료의 접근과 수정이 빠른 **자료구조**(data structure)에 저장된다
 - 자료구조에 저장된 입력 값을 기본적인 연산(primitive operation)을 차례로/반복적으로 적용하여 원하는 출력을 계산한다
- 알고리즘은 (1) 유한한 시간에 끝나야 하며 (2) 모든 입력에 대해 올바른 답을 출력해야 한다. 무한 루프에 빠지거나 어떤 입력에 대해서는 답을 출력하지 못하면 알고리즘이라 부를 수 없다

- 인류 최초의 알고리즘

- 그리스 수학자로 기하학의 아버지로 알려진 Euclid의 저서인 “Elements”(BC. 300)에 설명된 최대공약수 (Greatest Common Divisor: GCD)를 계산하는 알고리즘이 최초라고 알려져 있다
- 과학사 연구가들은 Euclid 이전의 Pythagoras 학파의 결과나 70여년 전의 다른 수학자의 결과가 더 앞서 있을 수도 있다고 주장하기도 한다
- Pseudo code (엄밀한 코드가 아닌 설명 중심의 코드란 뜻으로 실제로 실행되지 않는 코드)

```

algorithm gcd(a, b):      # 뺄셈 gcd
    while a*b != 0 do    # 두 수 중 하나가 0이 되기 전까지 빼기 연산 반복
        if a > b
            a = a - b
        else
            b = b - a
    return a+b           # 두 수 중 0이 아닌 수가 gcd이므로 합 자체가 gcd임

```

큰 수에서 작은 수를 빼는 과정을 큰 수가 작은 수보다 작아질때까지 반복하고 큰 수와 작은 수의 역할이 바뀌어 이 과정을 반복해서 작은 수가 0이 될 때까지 진행된다.

예1:

```

20 → 12 → 4 → 4 → 0
8 → 8 → 4 → 4 → 4

```

예2: a와 a-1 형식으로 주어지면 총 a번 while 문 반복하게 되는 매우 안 좋은 경우

```

6 → 1 → 1 → 1 → 1 → 1 → 0
5 → 5 → 4 → 3 → 2 → 1 → 1

```

여기서 큰 수가 작은 수보다 작을 때까지 작은 수만큼을 줄이게 된다는 것은 결국 큰 수를 작은 수로 나눈 나머지를 구하는 과정과 같다. 결국, 반복해서 뺄 것이 아니라 단순히 (큰 수 % 작은 수)를 하면 된다는 것을 알 수 있다

```

algorithm gcd(a, b):      # 나머지 gcd
    while a*b != 0 do
        if a > b
            a = a % b      # %는 나머지 연산자
        else
            b = b % a
    return a+b

```

예1:

```

20 → 4 → 4
8 → 0

```

예2: a와 a-1 형식으로 주어지더라도 나머지 방식은 매우 빠르게 계산됨

```

6 → 1 → 1
5 → 0

```

- iii. 위의 코드를 Python으로 변환해보자

```
algorithm gcd(a, b) → def gcd(a, b):
    while a*b != 0 do → while a*b != 0:
        if a > b → if a > b:
        else → else:
```

- iv. 만약 $a > b$ 라면, $\text{gcd}(a, b)$ 는 $\text{gcd}(a-b, b)$ 이 동일하다. 또한, $\text{gcd}(a, b)$ 는 $\text{gcd}(b, a\%b)$ 와 같다. (왜?) 이 점을 이용하면 gcd 함수를 재귀함수로도 작성할 수 있다. 예: $\text{gcd}(20, 12) \rightarrow \text{gcd}(12, 8) \rightarrow \text{gcd}(8, 4) \rightarrow \text{gcd}(4, 0) \rightarrow 4$

```
algorithm gcd(a, b): # 재귀 gcd
    if a*b == 0: # 바닥 조건 또는 탈출 조건
        return a+b
    if a > b: return gcd(b, a%b) # return gcd(b, a%b) 만 있으면 됨?
    else: return gcd(a, b%a)
```

- v. 결국, 두 수의 gcd 계산은 세 가지 방법 (알고리즘) - 뺄셈 계산 반복, 나머지 계산 반복, 재귀 계산 방법이 존재한다

두 방법 중에서 어떤 방법이 더 효율적(efficient)일까?

- b. 뺄셈 방법과 나머지 방법에 대한 수행 시간의 자세한 분석은 23 페이지를 참조한다

- 최초의 프로그래머는?

- a. Ada Lovelace (1817 - 1852)



Ada Lovelace

Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 of sep.)

Number of Operations.	Variables used.	Variables used in the results.	Indication of the value on my Working Variables.	Statement of Results.		Data.	Working Variables.	Results Variables.				
				$T_1 C$	$T_2 C$	$T_3 C$	$T_4 C$	$T_5 C$	$T_6 C$	$T_7 C$	$T_8 C$	$T_9 C$
1	$\times V_1$	$V_1 \cdot V_2$	$V_1 \cdot V_2 = V_3$	1	1	1	1	1	1	1	1	1
2	$- V_1$	$V_1 - V_1 = V_4$	$V_4 = 0$	1	1	1	1	1	1	1	1	1
3	$+ V_1$	$V_1 + V_1 = V_5$	$V_5 = 2$	1	1	1	1	1	1	1	1	1
4	$- V_1$	$V_1 - V_1 = V_6$	$V_6 = 0$	1	1	1	1	1	1	1	1	1
5	$+ V_1$	$V_1 + V_1 = V_7$	$V_7 = 2$	1	1	1	1	1	1	1	1	1
6	$- V_1$	$V_1 - V_1 = V_8$	$V_8 = 0$	1	1	1	1	1	1	1	1	1
7	$- V_1$	$V_1 - V_1 = V_9$	$V_9 = 0$	1	1	1	1	1	1	1	1	1
8	$+ V_1$	$V_1 + V_1 = V_{10}$	$V_{10} = 2$	1	1	1	1	1	1	1	1	1
9	$- V_1$	$V_1 - V_1 = V_{11}$	$V_{11} = 0$	1	1	1	1	1	1	1	1	1
10	$\times V_1$	$V_1 \cdot V_1 = V_{12}$	$V_{12} = 1$	1	1	1	1	1	1	1	1	1
11	$+ V_1$	$V_1 + V_1 = V_{13}$	$V_{13} = 2$	1	1	1	1	1	1	1	1	1
12	$- V_1$	$V_1 - V_1 = V_{14}$	$V_{14} = 0$	1	1	1	1	1	1	1	1	1
13	$+ V_1$	$V_1 + V_1 = V_{15}$	$V_{15} = 2$	1	1	1	1	1	1	1	1	1
14	$- V_1$	$V_1 - V_1 = V_{16}$	$V_{16} = 0$	1	1	1	1	1	1	1	1	1
15	$+ V_1$	$V_1 + V_1 = V_{17}$	$V_{17} = 2$	1	1	1	1	1	1	1	1	1
16	$- V_1$	$V_1 - V_1 = V_{18}$	$V_{18} = 0$	1	1	1	1	1	1	1	1	1
17	$+ V_1$	$V_1 + V_1 = V_{19}$	$V_{19} = 2$	1	1	1	1	1	1	1	1	1
18	$- V_1$	$V_1 - V_1 = V_{20}$	$V_{20} = 0$	1	1	1	1	1	1	1	1	1
19	$+ V_1$	$V_1 + V_1 = V_{21}$	$V_{21} = 2$	1	1	1	1	1	1	1	1	1
20	$- V_1$	$V_1 - V_1 = V_{22}$	$V_{22} = 0$	1	1	1	1	1	1	1	1	1
21	$+ V_1$	$V_1 + V_1 = V_{23}$	$V_{23} = 2$	1	1	1	1	1	1	1	1	1
22	$- V_1$	$V_1 - V_1 = V_{24}$	$V_{24} = 0$	1	1	1	1	1	1	1	1	1
23	$+ V_1$	$V_1 + V_1 = V_{25}$	$V_{25} = 2$	1	1	1	1	1	1	1	1	1
24	$- V_1$	$V_1 - V_1 = V_{26}$	$V_{26} = 0$	1	1	1	1	1	1	1	1	1
25	$+ V_1$	$V_1 + V_1 = V_{27}$	$V_{27} = 2$	1	1	1	1	1	1	1	1	1

Bernoulli number를 계산하는 코드

- i. Charles Babbage의 초기 컴퓨터 모델인 "Analytical Engine"에서 동작하는 프로그램을 작성하여 최초의 프로그래머라 불림
 - ii. Bernoulli number를 계산하는 코드였고, 현대 컴퓨터에서 구현된 최초의 프로그램 (또는 최초의 소프트웨어)으로 인정됨. 그러나 Analytical Engine이 끝내 완성되지 못해 본인의 프로그램이 구동되는 걸 보지 못함
 - iii. 프로그래밍 언어에서 사용되는 주요 개념인 루프, GOTO, IF 문 등의 제어 명령의 개념을 소개하고 서브루틴의 개념도 고안했다고 알려짐
 - iv. 프로그램의 실제 작성자가 아니라는 주장도 존재
 - v. 위 이미지와 내용은 위키피디아
https://en.wikipedia.org/wiki/Ada_Lovelace에서 발췌
 - vi. 핵심 업적은 문제를 해결하는 데 컴퓨터를 이용한 첫 사례라는 것!
- 최초의 컴퓨터 과학자는?
 - a. Alan Mathison Turing (1912 - 1954) - Imitation Game (관련 영화)
 - b. 컴퓨터(Turing Machine)에서의 Algorithm과 Computation의 수학적 토대를 완성한 수학자이자 컴퓨터 과학자
 - c. 핵심 업적은 컴퓨터로 해결할 수 없는 문제도 존재함을 증명한 것! (13장 계산 복잡도 등을 설명하는 절에서 추가 설명)

- **가상컴퓨터, 가상언어, 가상코드 (Virtual Machine, Pseudo Language, Pseudo Code)**

- 자료구조와 알고리즘의 성능(performance)은 대부분 **수행 시간**(시간복잡도: time complexity)으로 따지는 것이 일반적이다
- 이를 위해, 실제 코드(C, Java, Python 등)로 구현하여 실제 컴퓨터에서 실행한 후, 수행 시간을 측정할 수도 있지만, HW/SW 환경을 하나로 통일해야 하는 어려움이 있다
- 따라서, 가상언어로 작성된 가상코드를 가상컴퓨터에서 시뮬레이션하여 HW/SW에 독립적인 계산 환경(computational model)에서 측정해야 한다
- 가상컴퓨터**(virtual machine)
 - 현대 컴퓨터 구조는 Turing machine에 기초한 von Neumann 구조를 따른다
 - 현재 가장 많이 사용하는 현실적인 가상 컴퓨터 모델은 **RAM**(Random Access Machine) 모델이다. 본 과목에서 사용하는 RAM 모델은 classic RAM 모델이 아닌 다양한 현실적인 모델(Real RAM, word RAM 모델)을 조합해 사용한다
 - RAM 모델은 CPU + memory + register + primitive operation으로 정의되는 가상컴퓨터 모델 중 하나이다

CPU: 연산(operation, command)을 수행

메모리: 임의의 크기의 **실수** (즉, 정확한 실수 값을) 저장할 수 있는 무한한 워드(word)로 구성

레지스터(register): CPU의 계산에 활용되는 충분한 개수의 독립된 메모리

기본연산(primitive/atomic operation): 단위 시간에 수행할 수 있는 연산으로 정의

- 복사연산:** A = B (B 레지스터의 값을 A 레지스터에 복사 또는 B 레지스터의 값을 A 레지스터에 복사)
 - 산술연산:** +, -, *, / (나머지 % 연산은 허용 안되나, 본 강의에서는 포함한다) - 레지스터에 복사된 값에 대해 기본연산을 수행
 - 비교연산:** >, >=, <, <=, ==, !=
 - 논리연산:** AND, OR, NOT
 - *비트연산:** bit-AND, bit-OR, bit-NOT, bit-XOR, <<, >>
 - 메모리 접근:** R \leftarrow A (레지스터 A에 저장된 메모리 주소의 워드에 저장된 값을 레지스터 R로 읽기) R \rightarrow A (레지스터 R의 값을 A에 저장된 메모리 주소의 워드에 저장)
- 위의 모든 기본연산은 모두 레지스터의 값을 대상으로 이루어진다. 예를 들어, A = B + C라고 하면, B와 C라는 레지스터 값을 더 한 후 레지스터 A에 복사 (덮어쓰기)한다. 결국, 산술연산 + 한 번과 복사연산 한 번으로 구성된다. 총 두 번의 기본연산으로 구성되어 있다고 생각한다

- 단, 메모리 접근 연산은 따로 고려하지 않고, 다른 연산에 포함되어 있다고 가정한다. 예를 들어, $A = B + C$ 에서는 두 레지스터 사이의 덧셈한 결과를 다른 레지스터에 저장하는 것이지만, 메모리에 저장된 두 값을 읽어와 덧셈 후 다른 메모리에 저장하는 것으로 해석해도 된다는 의미이다. 이 경우에는 메모리 접근을 위해 추가 시간이 필요하지만, $+$ 과 $=$ 연산에 해당 시간이 포함되어 있다고 가정하기로 한다
- 본 과목에서는 설명의 편의를 위해, 레지스터와 메모리를 구분하지 않고 모두 메모리라고 가정한다. 그러면 $A = B + C$ 는 메모리 B와 C의 값을 더해 메모리 A에 저장하는 것으로 해석해도 된다는 의미이다. 당연히 $+$ 와 $=$ 의 두 번의 기본연산으로 구성되어 있다
- 워드의 bit 수도 얼마인지 크게 신경쓰지 않는다. 즉, 임의의 실수 값도 원하는 정밀도로 저장할 수 있다고 가정한다. 비현실적인 가정이지만 실제로 우리가 다루는 일반적인 알고리즘 문제에서는 정밀도가 그렇게 크지 않기에 충분히 감당할 수 있는 가정이다
- 본 과목의 RAM 모델은 아래의 두 RAM 모델의 내용을 섞어서 정의한 것이다**

Real RAM 모델은 `sqrt`, 삼각함수, `exp`, `log` 함수 등도 단위 시간에 수행된다고 가정하는 더 강력한 계산모델이다. 이 모델은 실수 연산이 중요한 **기하** 알고리즘의 계산 모델에서 주로 사용된다. (현대 컴퓨터와 프로그래밍 언어 모델에 가장 근접한 계산모델이다)

Word RAM 모델은 w -bit 정수(integer)만 저장할 수 있는 메모리로 구성된 계산 모델이다. 메모리는 w -bit 워드(word)로 구성된 배열이며 w 는 최소 $\log_2 n$ 이다. 이 모델은 일반 RAM 모델에서 제공하는 기본 연산을 그대로 단위 시간에 처리할 수 있다. 이 모델은 특히 정수를 다루는 자료구조와 알고리즘의 계산 모델로 널리 사용되고 있다

e. **가상언어**(Pseudo Language)

- 영어나 한국어와 같은 실제 언어보다 간단 명료하지만, C, Python 같은 프로그래밍 언어보다 융통성있는 언어로, Python과 유사하게 정의함. (수학적/논리적으로 모호함 없이 명령어가 정의되기만 하면 됨.)
- 기본 명령어:** (기본연산을 정의하는 명령어로 일반 프로그래밍 언어와 유사함)
 - 복사연산: $A = B$
 - 산술연산: $+$, $-$, $*$, $/$, $\%$
 - 비교연산: $>$, \geq , $<$, \leq , $=$, $!=$
 - 논리연산: AND, OR, NOT
 - 비트연산: bit-AND, bit-OR, bit-NOT, bit-XOR, $<<$, $>>$
 - 비교문: if, if else, if elseif ... else 문
 - 반복문: for 문, while 문
 - 함수정의, 함수호출, return 문

f. **가상코드**(Pseudo Code) - 가상언어로 작성된 코드

- i. 예: 배열 A의 n개의 정수 중에서 최대값을 계산하는 가상코드 (반드시 아래 형식을 따를 필요는 없음)

- ii. **algorithm arrayMax(A, n)**

```

input: n개의 정수를 저장한 배열 A ← 입력을 설명하는 주석의 역할
output: A의 수 중에서 최대값 ← 출력을 설명하는 주석의 역할
currentMax = A[0]
for i = 1 to n-1 do
    if currentMax < A[i]
        currentMax = A[i]
return currentMax

```

- iii. 위 코드에서 배정연산, 비교연산 등이 사용된다

- 알고리즘의 시간복잡도

- a. 가상컴퓨터에서 가상언어로 작성된 가상코드를 실행(시뮬레이션)한다고 가정한다
- b. 특정 입력에 대한 수행시간은 그 입력에 대해 수행되는 기본연산(primitive operation)의 횟수로 정의된다
- c. 문제는 입력의 종류가 무한하므로 모든 입력에 대해 수행시간을 측정하여 평균을 구하는 것은 현실적으로 매우 까다롭다는 점이다
- d. 따라서 **최악의 경우의 입력**(worst-case input)을 가정하여, 최악의 경우의 입력에 대한 알고리즘의 수행시간을 측정한다
- e. 최악의 경우의 입력에 대한 수행시간이 모든 입력에 대한 평균 수행시간보다는 정확하지는 않지만, 어떤 입력이 주어지더라도 최악의 경우의 수행시간보다 더 많은 시간이 걸리지 않는다는 점을 보장한다!

Checkpoint:

알고리즘의 수행시간 = 최악의 경우의 입력에 대한 기본연산의 수행 횟수

- f. 최악의 경우의 수행시간은 입력의 크기 n에 대한 함수 $T(n)$ 으로 표기 된다. $T(n)$ 의 수행시간을 갖는 알고리즘은 어떠한 입력에 대해서도 $T(n)$ 시간 이내에 종료됨을 보장한다
- g. 실시간 제어가 필요하고 절대 안전이 요구되는 분야(항공, 교통, 위성, 원자로 제어 등)에선 실제로 최악의 경우를 가정한 알고리즘 설계가 필요하기 때문에, 유효한 수행시간 측정방법으로 볼 수 있다
- h. 최악의 경우의 입력에 대해, 알고리즘의 기본연산(복사, 산술, 비교, 논리, 비트논리)의 횟수를 아래의 최대 값을 계산하는 예를 통해 자세히 세어보자
- i. 예: n개의 정수 중 최대값을 찾는 알고리즘

```

algorithm arrayMax(A, n)
    input: n개의 정수를 저장한 배열 A
    output: A의 수 중에서 최대값
    currentMax = A[0]
    for i = 1 to n-1 do
        if currentMax < A[i]
            currentMax = A[i]
    return currentMax

```

- **if** 문의 결과에 따라 `currentMax = A[i]`가 실행되든지 아니든지 결정된다
- 최악의 경우의 입력은 무조건 `currentMax = A[i]`을 실행해야 하므로 **if** 문을 계속 참(True)이 되도록 해야 한다. 이 같은 입력은 A의 저장된 값이 오름차순으로 정렬된 경우이다. (즉, 오름차순으로 정렬된 n개의 값이 저장된 배열 A가 최악의 경우의 입력이라는 의미이다)

j. 최악의 입력에 대한 횟수 분석

```

algorithm arrayMax(A, n)
    input: n개의 정수를 저장한 배열 A
    output: A의 수 중에서 최대값
    currentMax = A[0] (1번)
    for i = 1 to n-1 do
        if currentMax < A[i] (n-1번)
            currentMax = A[i] (n-1번)
    return currentMax

```

[주의 1] `A[i]`도 실제로 시간이 필요하다. i 번째 원소의 (메모리) 주소를 계산해 메모리에서 값을 읽어와야 하기 때문이다. 기본 연산 횟수 분석을 간단하게 하기 위해, 이 주소 계산을 위한 시간은 걸리지 않는다고 가정한다

[주의 2] **for** 루프의 제어에 관계하는 변수 i에 대한 기본 연산 역시 분석을 간단히 하기 위해 무시한다. ($i = 1$ 연산 1회, 루프 반복 때마다 $i \leq n-1$ 의 비교 1회, $i = i + 1$ 의 +, = 연산 2회가 필요하지만 대범하게 무시한다. 무시한 기본 연산 횟수는 루프의 반복 횟수에 비례하는 값이므로 결국 반복 횟수의 몇 배만 추가하는 정도의 영향만 미치게 된다. 이는 앞으로 살펴볼 Big-O로 표기된 최종 수행시간에 영향을 주지 않기에 무시해도 된다. 😊)

- 따라서 $T(n) = 2n - 1$ 이 된다
 - $n = 10$ 이면 $T(10) = 19$ 가 되어 19번 이내의 기본연산을 수행한다는 뜻이고, $n = 2000$ 이면, $T(2000) = 399$ 번의 기본연산을 수행한다는 의미이다

- [?] 질문: 아래 알고리즘의 최악의 입력에 대해 수행하는 기본연산의 횟수는?

```

algorithm arraySum(A, B, n)
    sum = 0           # 1번

```

```

for i = 0 to n-1 do
    for j = i to n-1 do      # 이중 for 루프는 n번 + (n-1)번 + ⋯ + 1번 반복
        sum = sum + A[i]*B[j] # 이 문장 자체는 *, +, = 3번의 기본연산
    return sum

```

결국, $T(n) = 1 + 3*(1+2+\dots+(n-1)+n) = 3n^2/2 + 3n/2 + 1$

- [GCD 계산 문제 - skip 가능] 앞에서 두 수 a, b의 최대공약수(gcd)를 계산하는 세 가지 방법에 대해 살펴봤다. 이 gcd 문제에 대한 다음 질문에 답해보자. ($a > b$ 라고 가정한다)
 - a. 먼저, 뺄셈 연산을 하는 gcd 계산 알고리즘부터 생각해보자. 이 알고리즘의 수행시간은 **while 루프의 반복 횟수에 비례**한다
 - b. **질문1:** $a = 100$, $b = 99$ 라고 할 때, while 루프의 반복 횟수는 얼마인가?
 - i. $a > b$ 이므로 $a = a - b = 1$ 이 되고, 그 이후엔 b 의 값이 1씩 감소해 두 값 모두 1이 되고, 한 번 더 루프를 수행하면 두 값 중 하나가 0이 되어 루프 반복이 끝난다. 결국, 100번의 반복이 필요하다
 - ii. a 와 b 의 값의 차이가 1인 경우 (예: $a = b - 1$ 인 경우)에 a 번 만큼 while 루프를 반복하게 된다!
 - c. **질문2:** 입력의 크기 n 은 얼마인가? 예를 들어,
 - i. 입력으로 주어진 값이 2개 뿐이므로 $n = 2$ 인가?
 - ii. 두 수 중 큰 수의 값이 100이므로 $n = 100$ 이라고 해야 할까?
 - d. $n = 2$ 라고 주장하는 경우
 - i. $a = b - 1$ 인 경우에는 루프는 a 번 반복된다. 따라서 gcd 알고리즘의 수행시간은 a 에 비례하는 식이 된다. 그러면 수행시간을 n 에 관한 식이 아닌 a 에 관한 식이 되어 제대로 정의되지 않는다. 문제는 a 값이 n 과 전혀 관계가 없으며 **매우 매우 클 수 있다**는 것이다. 즉, 입력의 크기는 매우 작은데 실제 수행시간은 **입력의 크기와 무관하게 매우 클 수 있다**는 게 문제다
 - e. $n = 100$ 이라고 (**무리하게**) 주장하는 경우 (즉, $n = a$ 라고 주장하는 경우)
 - i. 100번의 반복을 하기에 $n (= a)$ 번의 반복을 한다고 주장할 수 있다. 즉, 입력의 크기 n 에 관한 식으로 수행시간을 표현할 수 있다. 하지만 자연스럽지 않다!
 - f. 다음으로 나머지를 반복적으로 계산하는 gcd 알고리즘을 생각해보자
 - i. $a=100$, $b=99$ 라고 하면, 나머지 연산 (%)을 몇 번 하게 되나? **2번**
 - ii. 그럼 $a > b$ 인 경우에 최대 몇 번의 나머지 연산을 수행하게 될까?
 1. **사실1:** $a > b > 1$ 에 대해, $a \% b \leq a/2$ 임이 성립한다
 - a. 증명: 각자 해보기

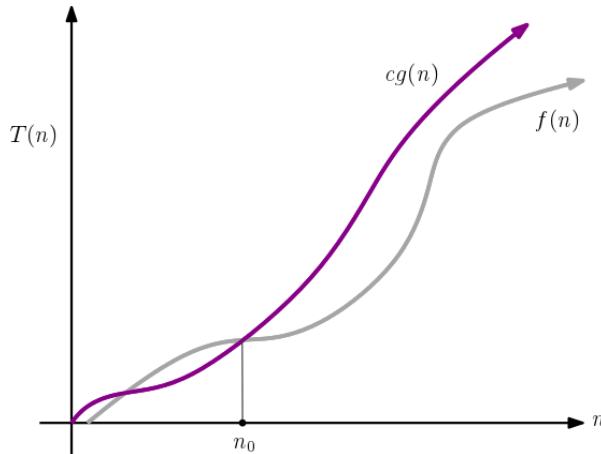
2. 사실2: $a > b > 1$ 에 대해, 나머지 % 연산은 최대 $2\log_2 a$ 번
이하만 수행된다
- 증명: 사실1을 이용하면 쉽게 증명 가능하다. 큰 수 a 가 반으로 줄어들기에, a 와 b 가 번갈아 반으로 줄어들어 0에 도달하는 것이므로 $\log_2 a + \log_2 b \leq 2\log_2 a$ 번 반복할 수 있다
- iii. 사실 2에 의해 gcd 알고리즘은 $2\log_2 a$ 에 비례하는 횟수의 기본연산을 수행한다
- $n = 2$ 라고 주장하는 경우에는 수행시간에 n 이 등장하지 않는다는 문제가 여전히 존재한다
 - $n = a$ 라고 주장하는 경우에는 결국 수행시간이 $\log_2 a$ 에 비례한다는 결론에 도달한다. 뺄셈 반복 알고리즘의 수행시간 a 보다는 매우 빨라졌지만, 여전히 부자연스럽다!
- iv. 입력으로 주어지는 수가 두 개이므로 $n = 2$ 라고 하는 게 일관성이 있지만, 실제 n 에 관한 식이 아닌 주어진 입력 값에 대한 식으로 수행시간이 표현된다는 문제가 있고, $n = a$ 로 하면 a 의 실제 값의 크기 자체가 입력의 크기가 된다는 부자연스러움이 있다. 두 가지 경우 모두 마음에 안든다는 게 결론!
- v. a 의 값이 컴퓨터에서는 $\log_2 a$ 비트로 표현된다. 산술, 비교 등의 기본 연산도 $\log_2 a$ 비트의 두 수에 대해 이루어지는 셈이다. 즉, $\log_2 a$ 비트 수가 일종의 입력의 크기라고 볼 수도 있다. a 의 값이 증가하면 그에 대한 비트 수도 증가한다는 직관에 맞다. 결국, 더 섬세하게 정의하면 gcd 문제에서의 입력의 크기 $n = \log_2 a$ 라고 정의할 수 있다. 이 경우를 해당 문제의 비트 복잡도(bit complexity)라고 부른다. (일반적인 경우는 비트 복잡도가 아닌 입력의 주어진 값의 개수를 입력의 크기 n 으로 정한다는 사실에 주의하자. 따라서 gcd 문제의 입력은 $n = 2$ 라고 말하는게 일반적이다)
- vi. 다만, gcd 문제에서는 입력을 비트 복잡도로 정의하는 것이 조금 더 자연스럽기 때문에 $n = \log_2 a$ 로도 정의해도 된다. 그러면 while 루프의 반복 횟수가 최악의 경우에 $2\log_2 a$ 를 넘지 않기에, gcd 알고리즘의 기본 연산 횟수는 $2n$ 을 넘지 않는다고 말할 수 있다. 다시 말하면, 입력의 비트 복잡도 크기 n 에 관한 식으로도 수행시간을 표현할 수 있다
- 반면에 뺄셈을 반복해 gcd를 계산하는 알고리즘은 최악의 경우 a 번 while 루프가 반복되므로 $a = 2^{\log_2 a}$ 가 되어 2^n 형식의 (입력 크기 $n = \log_2 a$ 에 대한) 지수 수행시간이 된다
- g. [중요] 그러나 본 교재에서는 비트 복잡도가 아닌 입력의 주어지는 값의 개수를 입력의 크기로 정의하겠다
- h. 질문3: 나머지 gcd 알고리즘의 수행시간이 가장 클 때의 두 수 a, b 는 어떤 값을 가질 때인가?

- i. 뱃셈 gcd 알고리즘에서는 두 수가 인접한 수일 때 두 수 중 큰 값만큼 루프를 반복했다
- ii. 나머지 gcd 알고리즘에서는 두 수가 _____ 때 루프 횟수가 최대가 된다
- iii. [힌트] 연속한 두 Fibonacci 수 F_{k+1} , F_k 에 대해 루프 횟수를 계산해보자
- iv. $F_{k+1} \% F_k = (F_k + F_{k-1}) \% F_k = (F_k \% F_k + F_{k-1} \% F_k) \% F_k = (0 + F_{k-1}) \% F_k = F_{k-1}$
- v. 결국 $\gcd(F_{k+1}, F_k) = \gcd(F_k, F_{k-1})$ 이 된다
- vi. 이 과정을 k번 반복하면 $\gcd(F_1, F_0)$ 에 도달하게 되고, 이 경우에는 $\gcd(1, 0)$ 이므로 gcd 값이 1이 되어 종료한다
- vii. 즉, k번 반복하게 된다
- i. 질문4: 그럼 k값은 얼마인가?
- i. 자료를 찾아보면, F_k 는 ϕ^k 에 비례한다. 여기서 ϕ 는 황금비율이라 불리며, $x^2 - x + 1 = 0$ 의 양의 실근이다
- ii. 결국, $n = F_k = \phi^k$ 이므로 $k = \log \phi$ 이 된다
- j. 결론적으로 나눗셈 반복 방법으로 gcd를 계산하는 알고리즘의 최악의 입력은 인접한 두 피보나치 수이며, 이 경우에 $\log \phi$ 정도 반복하게 된다
- k. [hard] 질문5: $\gcd(F_a, F_b) = F_{\gcd(a, b)}$ 이 성립함을 증명하라!

- Big-O 표기법

- 이제부터 입력의 크기 n 은 무조건 입력으로 주어지는 값의 개수로 정의한다
- 최악의 입력에 대한 기본연산의 횟수를 정확히 세는 건 일반적으로 귀찮고 까다롭다
- 정확한 횟수보다는 입력의 크기 n 이 커질 때, 수행시간 $T(n)$ 의 증가하는 정도(rate of the growth of $T(n)$ as n goes big)가 훨씬 중요하다
- 수행시간 함수 $T(n)$ 이 n 에 관한 여러 항(term)의 합으로 표현된다면, 함수 값의 증가율이 가장 큰 항 하나로 간략히 표기하는 게 시간 분석을 간단하게 하는 데 큰 도움이 된다
- 예를 들어, $T(n) = 2n + 5$ 이면, 상수항보다는 n 의 일차항이 $T(n)$ 의 값을 결정하게 되므로 상수항을 생략해도 큰 문제가 없다
- $T(n) = 3n^2 + 12n - 6$ 이면, n 값이 커짐에 따라 n^2 항이 $T(n)$ 의 값을 결정하게 되므로, 일차항과 상수항을 생략해도 큰 문제가 없다
- 이렇게 최고차 항만을 남기고 나머지는 생략하는 식으로 수행시간을 간략히 표기하는 방법을 **근사적 표기법**(Asymptotic Notation)이라고 부르고, Big-O(대문자 O)를 이용하여 다음의 예처럼 표기한다.
 - $T(n) = 2n + 5 \rightarrow T(n) = O(n)$
 - $T(n) = 3n^2 + 12n - 6 \rightarrow T(n) = O(n^2)$
 - [자료]
- Big-O 표기하기 위한 방법은 다음과 같다.
 - n 이 증가할 때 가장 빨리 증가하는 항(최고차 항)만 남기고 다른 항은 모두 생략한다
 - 가장 빨리 증가하는 항에 곱해진 상수 역시 생략한다
 - 남은 항을 $O(\)$ 안에 넣어 표기한다
- Big-O에 대한 엄밀한 수학적 정의
 - 예를 들어, $T(n) = 3n^2 + 12n - 6$ 이 있을 때, 최고차 항이 n^2 인 함수 클래스에 포함된다는 뜻은 $n > n_0$ 이상의 모든 값에 대해 $3n^2 + 12n - 6 \leq cn^2$ 이 성립하는 n_0 값과 c 값이 항상 존재한다는 뜻이다. (아래는 영어 정의)

$$O(f(n)) = \{g(n) : \text{there exists constant } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq cg(n) \text{ for all } n > n_0\}$$



iii. $T(n) = 3n^2 + 12n - 6 = O(n)$ 이라고 말할 수 없다. 이유는 위의 정의에 의하면, $3n^2 + 12n - 6 \leq cn$ 을 만족하는 n_0 값과 c값을 찾을 수 없기 때문이다

- n의 값이 작으면 위 부등식이 성립하지만 이차식의 증가율이 훨씬 커서 n이 상당히 커지면 c의 값을 어떻게 정하더라도 위의 부등식을 만족할 수 없기 때문이다

j. 예 1: $\log_a n = O(\log_2 n)$

- $\log_a n = \log_2 n / \log_2 a = (1/\log_2 a) \log_2 n = c \log_2 n = O(\log_2 n)$ (여기서 $c \geq 1/\log_2 a$ 인 임의의 상수로 정하면 됨)
- 따라서 밑의 값의 상관없이 $\log n$ 은 Big-O 표기법으로 밑이 2인 $\log_2 n$ 이라고 통일해도 상관없다

k. 예 2: (python 또는 pseudo code로 기술한 알고리즘의 수행 시간)

- O(1) 시간 알고리즘:** constant time algorithm: 값을 1 증가시킨 후 리턴

```
def increment_one(a):
    return a+1
```

- O(log n) 시간 알고리즘:** logarithmic time algorithm: log의 밑은 2라고 가정: n을 이진수로 표현했을 때의 비트수 계산 알고리즘

```
def number_of_bits(n):
    count = 0
    while n > 0:
        n = n // 2
        count += 1
    return count
```

- O(sqrt(n)) 시간 알고리즘:** n의 약수의 개수를 세는 알고리즘

```
def number_of_factors(n):
    count, k = 0, 1
    while k*k < n:
        if n%k == 0:
```

```

        count += 2
        k += 1      # need more?
    return count

```

- iv. **$O(n)$ 시간 알고리즘** : linear time algorithm: n개의 수 중에서 최대값 찾는 알고리즘
- v. **$O(n^2)$ 시간 알고리즘**: quadratic time algorithm: 두 배열 A, B의 모든 정수 쌍의 곱의 합을 계산하는 알고리즘

```

def array_sum(A, B, n)
    sum = 0
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            sum += A[i]*B[j]
    return sum

```

- vi. **$O(n^3)$ 시간 알고리즘**: cubic time algorithm: $n \times n$ 인 2차원 행렬 A와 B의 곱을 계산하여 결과 행렬 C를 리턴하는 알고리즘

```

def mult_matrices(A, B, n)
    input: n x n 2d matrices A, B
    output: C = A x B
    for i = 1 to n do
        for j = 1 to n do
            C[i][j] = 0
    for i = 1 to n do
        for j = 1 to n do
            for k = 1 to n do
                C[i][j] += A[i][k] * B[k][j]
    return C

```

- vii. **$O(2^n)$ 이상의 시간이 필요한 알고리즘**: exponential time algorithm: k번째 피보나치 수 계산하는 재귀 알고리즘 또는 n개의 원소를 갖는 집합의 모든 부분집합을 출력하는 알고리즘

- 아래 fibonacci 알고리즘은 $O(\phi^n)$ 시간 알고리즘이다

```

def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)

```

- $f(n)$ 은 $O(2^n)$ 시간 알고리즘이다

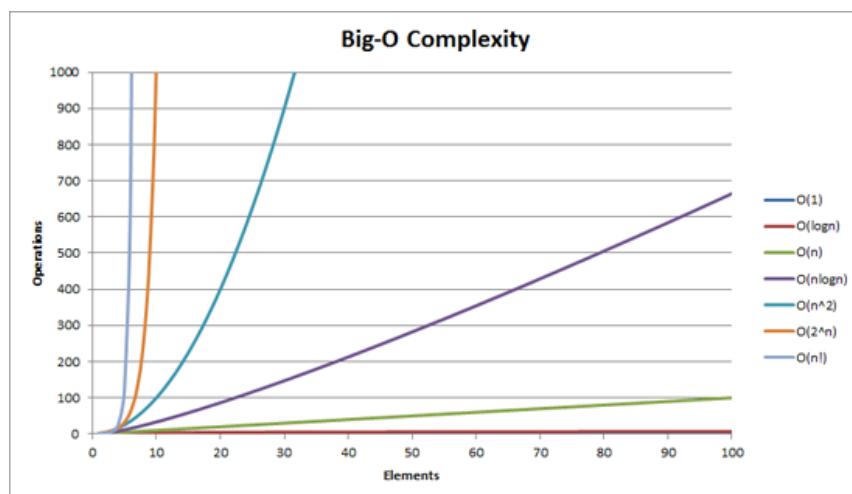
```

def f(n):
    if n == 1: return n
    return f(n-1) + f(n-1)

```

viii. $O(n!)$ 시간이 필요한 알고리즘: 1부터 n까지의 자연수에 대한 순열(permuation)을 모두 출력하는 알고리즘. 순열의 개수가 $n!$ 이므로 최소한 그 정도의 시간이 필요한 알고리즘

- i. **다항 시간 (polynomial time)** 알고리즘: 수행 시간이 $O(n^k)$ 인 알고리즘 (k 는 0보다 큰 실수)
- m. **지수 시간 (exponential time)** 알고리즘: 다항 시간이 아닌 지수 시간 알고리즘
- n. **비교 그림** [출처: [stackoverflow](#)]



- o. 더 자세한 비교 그림 및 설명 <http://bigocheatsheet.com/>
 - p. 자료구조와 기본 알고리즘의 작동 원리를 코드와 애니메이션으로 보여주는 사이트 소개
 - i. 싱가포르 대학 visualgo: <https://visualgo.net/ko>
-



[⌛ 알고리즘 퍼즐 1] 독살 음모를 밝혀라!

- 정답은 왼쪽의 QR 코드로 확인

왕이 마실 와인 8병 중 한 병에 강력한 독이 들어 있다. 한 방울만이라도 치명적이다. 정상 와인을 아무리 섞어도 치명적이다. 다행히 검사장비를 구할 수 있다. 단, 검사는 1시간이 필요하다. 왕은 무조건 1시간 후에 와인을 마실테니 독이 든 병을 찾아내라는 명령을 내렸다. 이를 위해, 최소 몇 대의 검사 장비가 있으면 충분할까? (한 번 검사할 때에는 여러 병의 와인을 섞어 검사 가능하다)



[⌛ 알고리즘 퍼즐 2] 가장 빠른 말 찾기 (구글 인터뷰 질문)

- 정답은 왼쪽의 QR 코드로 확인

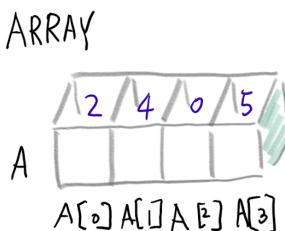
25마리의 말 중에서 가장 빠른 세 마리를 선택하고 싶다. 경기장엔 한번에 다섯 마리 이하로 경주를 할 수 있고, 시계가 없어 기록을 짤 수 없고, 대신 순위만 알 수 있다. 한 경주에서 같은 순위는 없다고 가정한다. 최소 몇 번의 경주로 가장 빠른 세 마리를 알 수 있는가?

2. Sequential Structures: Array (List), Stack, Queue, Dequeue

1. Array (배열), List (리스트)

- a. 데이터를 연속적인 메모리 공간에 저장하고, 저장된 곳의 주소(address, reference)를 통해 매우 빠른 시간에 접근할 수 있는 가장 많이 쓰이는 기본적인 자료구조
- b. C, C++, java, Python 등의 언어에서는 모두 array 자료구조 지원 (Python에서는 list, tuple 등의 자료구조가 배열 자료구조임)
- c. 예1: C 언어에서의 배열 A

i. `int A[4]; // 정수 4개를 저장할 수 있도록 연속적인 메모리 공간 할당`



- ii. 만약, 2번째 셀(cell)에 저장된 값을 알고 싶다면, 그 셀의 시작 주소는 아래와 같이 계산이 가능하다. (왜? 연속적으로 저장되어 있기 때문에)

A의 시작 주소 + 2*(값이 차지하는 byte 수)

- iii. 결국, 배열의 시작 주소, 저장된 값의 종류(바이트 개수), 몇 번째에 저장되어 있는지를 나타내는 인덱스 (index) 세 가지 정보만으로 값이 저장된 곳의 주소를 상수 시간에 (매우 빠르게) 계산할 수 있다 [매우 중요한 사실!]

`A[2]의 시작 주소 = A[0]의 시작 주소 + 2*4 (index=2, sizeof(int)=4)`

- iv. 읽기와 쓰기 연산: O(1) 시간

1. 읽기 예: `print(A[2]) # A[2]의 값을 읽는 연산`
2. 쓰기 예: `A[2] = 8 # A[2]의 메모리에 값 8을 저장하는 연산`

- v. C의 배열은 읽기/쓰기 연산만 제공하는 제한된 자료구조로 두 가지 방식으로 선언이 가능하다

1. **정적 선언**: 위의 예에서처럼 배열의 크기를 미리 정해 선언하는 경우. 미리 크기를 계산해야 하는 단점
2. **동적 선언**: `malloc`, `calloc` 등의 메모리의 동적 할당 함수를 호출해 코드 실행 중 선언하는 경우. 크기를 코드 실행 도중 입력 받거나 결정해야하는 단점

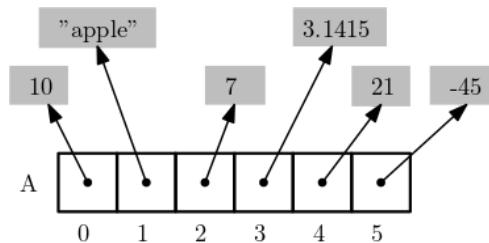
d. 예2: Python 언어에서의 list A (또는 tuple A)

- i. [중요] C 언어 배열의 셀에는 실제 값(data)이 저장된 형식이지만, Python 리스트의 셀에는 데이터가 아닌 데이터가 저장된 곳의 주소(address 또는 reference)가 저장된다. 즉, 데이터가 저장된 곳(객체)이 데이터를 나타낸다

이 방법의 장점은 실제 데이터의 타입(type)이 하나로 통일할 필요가 없다는 것이다. 즉, 아래 예처럼 다양한 타입의 데이터를 한 리스트에 저장할 수 있다

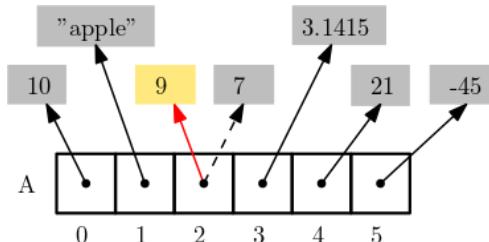
[사실] 파이썬의 모든 것은 객체(object)이다. 심지어 함수 자체도 객체이다

- ii. 항상 객체의 주소만 저장하기 때문에, 리스트의 셀의 크기를 메모리의 주소를 표현할 수 있는 (4 바이트 또는) 8 바이트로 고정하면 된다. 모든 셀의 크기가 같기 때문에 index에 의해 $O(1)$ 시간 접근이 가능하다
- iii. 예: `A = [10, "apple", 7, 3.1415, 21, -45]`인 경우에 다양한 리스트 연산



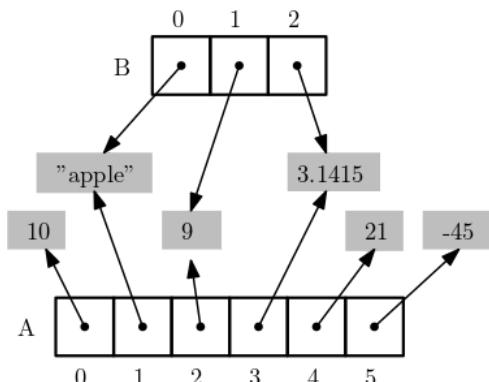
사칙연산:

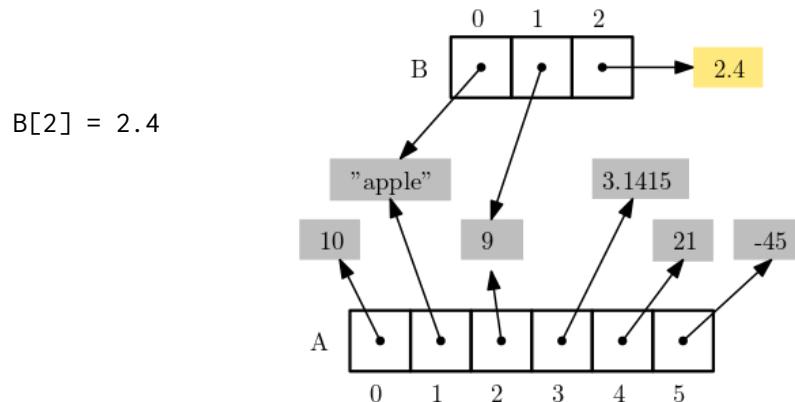
`A[2] = A[2] + 2`



Slicing (복사) 연산:

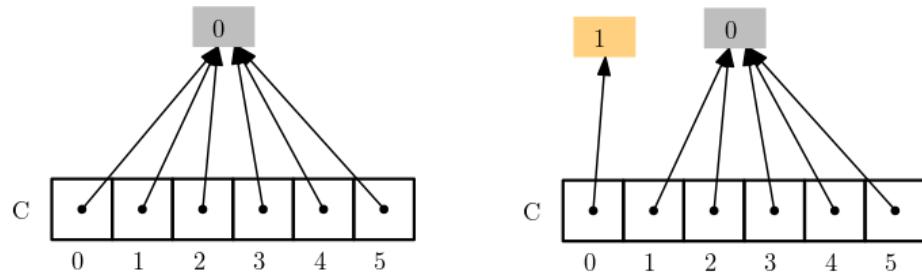
`B = A[1:4]`





초기화 연산:

```
C = [0] * 6    # 객체 0의 주소를 모두 저장한다 (객체 공유)
C[0] = 1       # 정수 객체는 immutable이라 새로운 값 1에 대한 객체
                # 생성됨
```



여기서 알아야 할 파이썬 지식: **변경 불가능** 타입 vs. **변경 가능** 타입

변경 불가능 타입: int, float, str, bool, tuple 등
변경 가능 타입: list, dict, set 등

iv. 파이썬의 리스트는 읽기/쓰기 이외에 훨씬 더 유연하고 강력한 연산을 지원한다

1. A.append(value): 맨 오른쪽 (뒤)에 새로운 값 value를 삽입
2. A.pop(i): A[i] 값을 지운 후 리턴 (A[i]의 오른쪽 값들은 왼쪽으로 한 칸씩 당겨 A[i]의 빈 칸을 채운다. 따라서, 리스트의 길이가 하나 감소)
 - a. pop()은 맨 오른쪽 값을 지움
3. A.insert(i, value): A[i] = value 연산 (단, A[i], A[i+1], ... 값들은 오른쪽으로 한 칸씩 이동해 A[i]를 비운 후, value 값 저장)

4. A.**remove**(value): value를 찾아 제거 (value 값이 여러 개라면 가장 왼쪽 값을 지우고, 오른쪽의 값들은 왼쪽으로 한 칸씩 이동해 빈 공간을 메꿈)
5. A.**index**(value): value 값이 처음으로 등장하는 index 리턴
6. A.**count**(value): value 값이 몇 번 등장하는지 횟수를 세어 리턴
7. A[i:j] : A[i], ..., A[j-1]까지를 복사해 새로운 리스트 생성하여 리턴
 - a. **slicing** 연산이라 부름
 - b. B = A[i:j]라고 하면 A[i], ..., A[j-1]까지가 복사되어 B가 됨. A는 **전혀 변화가 없음**에 유의!
8. value **in** A: 멤버십 연산자 - A에 value가 있으면 **True**, 없으면 **False**

v. 예를 통한 연산 이해

```
>>> A = [2, 1, 3, 2, 8]
>>> A.append(5)
>>> A
[2, 1, 3, 2, 8, 5]
>>> A.pop(1)
1
>>> A
[2, 3, 2, 8, 5]
>>> A.insert(2, 7)
>>> A
[2, 3, 7, 2, 8, 5]
>>> A.remove(2)
>>> A
[3, 7, 2, 8, 5]
>>> A.index(8)
3
>>> A.append(3)
>>> A.count(3)
2
>>> B = A[1:4]
>>> B
[7, 2, 8]
```

vi. 리스트는 동적배열이다!

1. C 언어의 배열과의 또 다른 **중요한 차이점**은 리스트의 크기 (셀의 개수)가 필요에 따라 자동으로 증가, 감소한다는 것이다

```
>>> import sys          # getssizeof 함수 사용위해
>>> A = []
>>> sys.getsizeof(A)
```

```

64          # 빈 리스트도 64 바이트 할당
          # 실제 컴퓨터마다 초기 할당 크기가 다름
>>> A.append("Python")    # append 후 96 바이트로 재 할당
>>> sys.getsizeof(A)
96

```

2. 그래서 `append` 또는 `insert` 연산을 위한 공간(메모리)이 부족하면 더 큰 메모리를 할당받아 새로운 리스트를 만들고 이전 리스트의 값을 새 리스트로 **모두** 이동한다. 반대로 `pop` 연산을 하면서 실제 저장된 값의 개수가 리스트 크기에 비해 충분히 작다면 더 작은 크기의 리스트를 만들고 모두 이동한다 ⇒ 마치, 식구가 많으면 큰 집으로 이사하고, 식구가 줄어들면 작은 집으로 이사하는 것과 같은 방식임
3. 이렇게 필요에 따라 크기가 변하는 배열을 **동적 배열 (dynamic array)**이라 부른다

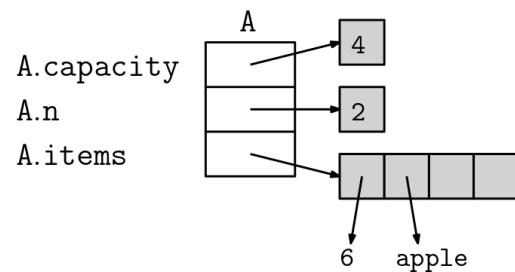
C에서의 배열은 동적 배열이 아니므로 사용자가 직접 상황에 맞게 처리해야 한다. 즉, `malloc` 함수를 호출해 더 큰 배열을 할당한 후 기존의 배열에 저장된 값을 이동하는 코드를 직접 작성해야 한다

따라서 파이썬 사용자는 리스트의 크기를 전혀 신경쓰지 않아도 된다

4. 동적 배열인 `list`를 위해선 python 내부적으로 현재 `list`의 크기(`capacity`)와 `list`에 저장된 실제 값의 개수(`n`)를 항상 알고 있어야 한다
 - a. 이를 위한 내부 변수가 필요하므로 빈 리스트 `A`를 위해 할당한 메모리의 크기는 0 바이트보다 클 수 밖에 없다

5. 추가 메모리 이외에 연산 시간에도 영향을 준다. 특정 `append` 연산에서 메모리를 늘려야 한다면 이전 리스트의 값을 새로운 리스트로 일일이 이동해야 한다 (아래의 코드와 유사한 방식으로 이루어진다)

리스트 클래스는 `capacity`, `n`, `items` 세 멤버를 갖는다고 가정 (아래 그림 참조) `capacity`는 리스트의 크기를, `n`은 현재 저장된 값의 개수를, `items`는 값이 저장되는 배열을 의미한다



```

def append(A, value):
    1. if A.capacity == A.n: # 꽉 참! resize 필요! 2배씩 증가
        a. allocate a new list B with larger memory
        b. update B.capacity and B.n
        c. for i in range(A.n):
            B.items[i] = A.items[i]
        d. dispose A # A의 메모리 해제
        e. A = B # B 이름을 A로 바꿈
    2. A.items[n] = value
    3. A.n += 1

```

6. 단계 1.c에서 A에 저장된 값의 개수만큼 시간이 걸림: **O(n)**

결국, resize가 일어나지 않는 append 연산은 **O(1)** 시간, 일어나는 경우엔 **O(n)** 시간이 걸림

7. 그러나 resize가 append 할 때마다 발생하는 것이 아니라 가끔 발생하는 것 (크기가 $2^1, 2^2, 2^3, \dots$ 일 때만 발생하는 것)이라 평균 시간을 계산해보면 **O(n)** 시간보다 훨씬 작다
8. 2배씩 크기를 증가하거나 반으로 감소하는 경우엔 append, pop 연산 시간은 **평균적으로 O(1)**임을 증명할 수 있다. (이에 대한 내용은 뒤의 **hash table** 자료구조에서 다시 자세히 언급)

vii. [중요] 수행시간:

1. A[i] = A[j] + 1
 - a. A[j] 값을 읽은 후 (단위시간) 1을 더하고 (단위시간) A[i]에 쓴다 (단위시간) (읽기/쓰기: **O(1)** 시간)
2. A.append(5)
 - a. 맨 오른쪽에 삽입. **평균적으로 O(1)** 시간
3. A.pop()
 - a. 맨 오른쪽 값을 삭제 후 리턴. **평균적으로 O(1)** 시간
4. A.pop(3)
 - a. A[4], ..., A[len(A)-1] 값이 왼쪽으로 한 칸씩 이동하는 시간 필요
 - b. 최악의 경우가 A.pop(0)일 때이므로 **O(len(A))** 시간
5. A.insert(0, 10)
 - a. A[0], ..., A[len(A)-1]을 모두 오른쪽으로 이동하므로 A의 크기만큼 시간 필요
 - b. 최악의 경우 (+ 평균적인 경우)에 **O(len(A))** 시간
6. A.remove(9)

- a. 처음으로 등장하는 9를 찾아 제거. 최악의 경우 (+ 평균적인 경우)엔 $O(n)$ 시간 필요

7. `A.index(9), A.count(9)`

- a. 9가 처음으로 등장하는 index와 총 횟수를 계산해야 하므로 최악의 경우 $O(\text{len}(A))$ 시간 필요

8. `A[i:j]`

- a. $A[i], \dots, A[j]$ 까지 slicing(복사)해야 하기에 $O(j-i)$ 시간이 필요함. 최악의 경우에 $O(n)$ 시간 필요. 상수시간이 아님에 유의!

9. `5 in A`

- a. 값 5가 리스트 A에 있는지 없는지를 검색하는 연산자 `in`은 $A[0], A[1], \dots$ 차례로 검사하기에 $O(\text{len}(A))$ 시간 필요. 상수시간이 아님에 유의!

viii. [단점 1] 리스트는 메모리 인심이 좋아 메모리 낭비가 큼

```
>>> A = []
>>> sys.getsizeof(A)
64                                     # 빈 리스트도 64 바이트 무조건 할당
                                         # 실제로는 컴퓨터마다 다름에 유의
>>> A.append("Python")      # append 연산 후 96 바이트로 재 할당
>>> sys.getsizeof(A)
96

>>> a = 1
>>> sys.getsizeof(a)          # 정수 변수: 4바이트가 아닌 28바이트
28
```

메모리 효율적인 리스트를 사용하려면, array 모듈에서 제공하는 compact array type을 사용할 것! (파이썬 교재 참조)

ix. [단점 2] `append`와 `pop`의 연산이 최악의 경우에 $O(1)$ 시간이 아니라 평균 $O(1)$ 시간이라는 것

x. [💪] 나만의 동적 배열을 구현해보자 (리스트 클래스를 설계해보자)

e. [이차원 리스트] C의 이차원 배열처럼 이차원 리스트도 생성할 수 있다

i. 아래처럼 리스트 안에 리스트를 원소로 정의하면 이차원 리스트 생성

```
>>> A = [[1,2], [3,4], [5,6,7]]
>>> A[0]
[1, 2]
>>> A[0][1]
2
```

```
>>> A[2]
[5, 6, 7]
>>> A[2][1]
6
```

- ii. 3차원 또는 그 이상의 리스트도 같은 방법으로 생성 가능
- iii. [주의] 3x3 2차원 리스트를 모두 0으로 초기화하여 생성한다고 하자

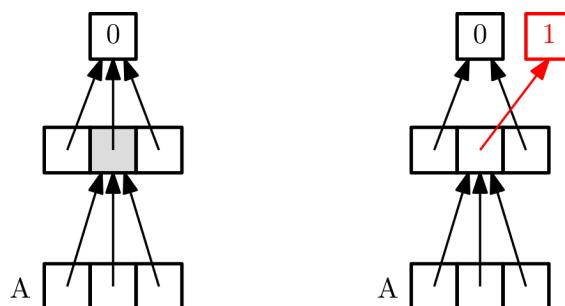
방법 1: 잘못된 방법임에 주의!

```
>>> A = [[0]*3]*3
>>> A
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

>>> A[1][1] = 1
>>> A          # A[0][1], A[1][1], A[2][1]이 모두 동시에 바뀐다!
[[0, 1, 0], [0, 1, 0], [0, 1, 0]]
```

1. 이유는? 아래 그림처럼 A[0], A[1], A[2] 모두 하나의 일차원 리스트 객체 [0, 0, 0]를 가리킨다. A[1][1] = 1로 수정하면 A[0][1], A[1][1], A[2][1]이 모두 1을 가리키게 되는 의도치 않는 현상이 발생한다

A[0], A[1], A[2]가 가리키는 1차원 리스트가 **mutable**이기에 발생하는 현상이다



2. 그러면 어떻게 초기화를 해야 하는가?

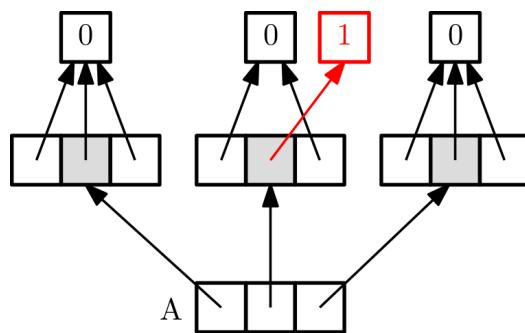
방법 2: 올바른 방법

```
>>> A = []
>>> for i in range(3):
...     A.append([0]*3)
>>> A
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> A[1][1] = 1
>>> A
[[0, 0, 0], [0, 1, 0], [0, 0, 0]]
```

또는 list comprehension을 이용 (이 장의 마지막 부분 참고)

```
>>> A = [[0]*3 for _ in range(3)] # comprehension
>>> A
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> A[1][1] = 1
>>> A
[[0, 0, 0], [0, 1, 0], [0, 0, 0]]
```

그림으로 표현하면 아래와 같다



f. 예3 [skip 가능]: Python의 numpy 모듈에서 제공하는 array 자료구조

- i. 대용량 데이터 처리에 효율적 → 데이터 사이언스, 머신러닝 코딩에 적합
- ii. numpy 모듈이 필요함: 보통 `import numpy as np`로 사용

```
>>> import numpy as np
>>> A = np.arange(10)      # range(start, stop, step) 유사
>>> A
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print(A)
[0 1 2 3 4 5 6 7 8 9]
```

iii. 특징

1. **A[k]**: k번째 원소 값을 인덱스로 O(1) 시간에 접근 (list와 같은 점)

2. **mutable** (list와 같은 점)

```
>>> A[2] += 10
>>> A
array([ 0,  1, 12,  3,  4,  5,  6,  7,  8,  9])
```

3. 한 종류 타입의 값만 저장 가능 (list와 다른 점)

```
>>> A[2] = 3.14    # 정수로 변환되어 저장!
>>> A
array([0, 1, 3, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> A[2] = "python" # error 발생
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
'python'
```

4. 메모리를 list보다 적게 사용 (**장점**)

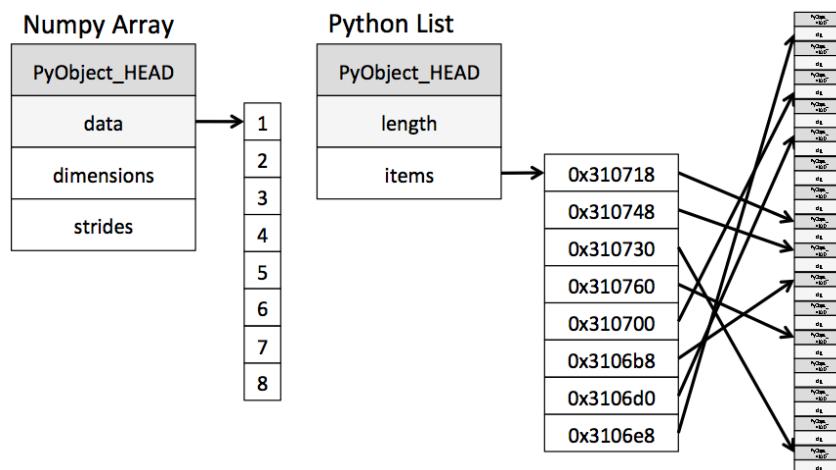
```
>>> B = [x for x in range(10)]
>>> import sys
>>> sys.getsizeof(B)      # B[k]에 값의 reference 저장
100
>>> sys.getsizeof(A)      # A[k]에 값을 직접 저장
88                         # 같은 타입이므로 O(1) 시간 접근 가능
```

5. 배열 단위의 연산이 매우 빠름 (**장점**)

- a. 2차원 array (행렬)에 대한 연산 등의 algebra 연산에
최적화되어 설계 → **빅 데이터, 머신러닝 데이터 처리에 적합**

iv. list 보다 빠르고 더 적은 메모리를 사용하는 이유

1. 아래 그림처럼 리스트는 2번의 단계를 거쳐 원소에 접근하지만,
numpy의 배열은 C의 array처럼 1번의 단계만 거쳐 원소에 접근
그림 출처: [Why Python is Slow: Looking Under the Hood](#)



2. 위의 그림에서 numpy 배열은 data에 원소 값들이 연속적으로 할당되어
1번 단계만에 접근할 수 있지만, list에선 items의 저장된 원소의
주소(reference)를 한번 더 따라가야 하므로 추가 단계가 필요하다.
그래서 연산 시간이 빠르고 메모리 사용량도 적다

v. 생성 및 기본 연산:

1. **arange(start, stop, step)** 사용하는 방법

```
>>> A = np.arange(0, 40, 2)
>>> A
```

```

array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24,
       26, 28, 30, 32, 34, 36, 38])
>>> A.shape # A의 차원 - 여기선 원소 20개의 1차원
(20,)
>>> A = A.reshape(4,5)    # 4x5 2차원 배열로 변환
>>> A
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18],
       [20, 22, 24, 26, 28],
       [30, 32, 34, 36, 38]])
>>> A.shape
(4, 5)
>>> A[1][2] = 2*A[3,2] # A[i][j] 또는 A[i,j] 가능
>>> A
array([[ 0,  2,  4,  6,  8],
       [10, 12, 68, 16, 18],
       [20, 22, 24, 26, 28],
       [30, 32, 34, 36, 38]])

```

2. list를 사용하는 방법

```

>>> A = np.array([1,2,3,4,5,6])
>>> A
array([1, 2, 3, 4, 5, 6])

```

3. 제공 함수를 사용하는 방법

```

>>> A = np.zeros((3,5))    # 0.0으로 초기화된 2x3 배열
>>> A
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
>>> A = np.ones((2,3))    # 1.0으로 초기화된 2x3 배열
>>> A = np.full((2,3), 5) # 5로 초기화된 2x3 배열
>>> A
array([[5, 5, 5],
       [5, 5, 5]])
>>> A = np.eye(3,3)       # 3x3 identity 배열 생성
>>> A
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

```

4. linspace와 random 함수를 이용하는 방법

```

>>> A = np.linspace(0, 10, 4) # [0,10] 구간을 4 등분
>>> A
array([ 0.,  3.33333333,  6.66666667, 10. ])
>>> A = np.random.random((2,2))

```

```
# 2x2 원소 값을 [0, 1) 구간에서 랜덤하게 생성해 초기화  
>>> A  
array([[0.06494227, 0.53270715],  
       [0.96106495, 0.61470327]])
```

가장 단순한 선형 자료구조 세가지: Stack, Queue, Dequeue

- 데이터 값을 저장하는 기본적인 구조로 일차원의 선형(linear) 자료구조이다
- (배열/리스트와 유사하게) 값을 저장(insert 또는 set)하는 연산과 저장된 값을 꺼내는(remove 또는 get) 연산이 제공된다. 그러나 **매우 제한적인 규칙 - LIFO, FIFO**-등을 따른다
- Stack은 가장 최근에 저장된 값 다음에 저장되며, 가장 최근에 저장된 값이 먼저 나간다: **LIFO**(Last In First Out) 원칙
- Queue는 가장 최근에 저장된 값 다음에 저장되지만(stack과 동일) 가장 오래전에 저장된 된 값부터 나간다: **FIFO**(First In First Out, 선착순) 원칙
- Dequeue는 stack과 queue의 연산을 모두 지원하는 자료구조이다

2. Stack

- a. 데이터는 파이썬 리스트에 저장 (append, pop 함수가 제공되므로)
- b. 연산은 push, pop, top, isEmpty, size(len) 등 5 가지 연산 제공
- c. 클래스 Stack 선언: stack_queue.py

```
class Stack:
    def __init__(self):
        self.items = []                      # 데이터 저장을 위한 리스트 준비

    def push(self, key):
        self.items.append(key)

    def pop(self):
        try:                                # pop할 아이템이 없으면
            return self.items.pop()
        except IndexError:                  # IndexError 발생
            print("Stack is empty")

    def top(self):
        try:
            return self.items[-1]
        except IndexError:
            print("Stack is empty")

    def __len__(self):                     # len()로 호출하면 stack의 item 수 반환
        return len(self.items)

    def isEmpty(self):
        return len(self) == 0
```

d. 예제:

```
from stack_queue import Stack    # Stack 클래스를 import
S = Stack()
S.push(10)
S.push(2)          # S = [10, 2]
print(S.top())    # 2, S = [10, 2]
print(S.pop())    # 2, S = [10]
print(len(S))     # 1
print(S.isEmpty()) # False
```

- e. 앞에서 정의한 Stack 클래스는 파이썬의 리스트 자료구조를 그대로 사용한 경우이다. 따라서 동적 배열인 리스트의 메모리 관리 방식에 따라 append (push), pop은 모두 평균 $O(1)$ 시간을 보장한다. 평균 상수 시간이 아닌 최악의 경우에도 상수 시간을 보장하고 싶다면, 미리 스택의 크기를 미리 정해서 할당한 후, push, pop 연산을 설계하면 된다.
- i. 스택의 크기가 정해져 있기에 모든 칸에 값이 저장되어 있다면, 새로운 값을 저장하기 위한 빈 공간이 없을 수도 있다. 결국, push 연산에서 빈 공간이 있는지를 항상 먼저 검사하고 빈 공간이 없다면 오류 메시지 "Stack is full"를 출력해야 한다
 - ii. 스택에 가장 최근에 저장된 값의 인덱스를 알아야 하기에 이를 위한 변수가 필요하다 (아래 예에서는 idx로 표기함)

`class Stack: # 스택의 크기를 max_size로 정한 경우`

```
def __init__(self, max_size=8):
    self.items = [None] * max_size # max_size 크기의 리스트 선언
    self.idx = -1                # index of the stack's top
                                # initially -1 (why?)

def push(self, key):
    if self.idx+1 == len(self): # no more available empty slot
        print("Stack is full")
        return
    self.idx += 1
    self.items[self.idx] = key

def pop(self):
    if self.idx == -1: # empty stack
        print("Stack is empty")
        return None # return None if stack is empty
    top_item = self.items[self.idx]
    self.idx -= 1
    return top_item

# top 함수도 pop 함수와 유사하게 작성
```

f. 스택의 사용 예 1: 괄호 맞추기

- i. 수식 $(2+5)*7-((3-1)/2+7)/4$ 라고 하면, 왼쪽 괄호와 오른쪽 괄호가 쌍을 이뤄 짹이 맞아야 한다.
 - ii. $((())$ 경우는 올바른 짹이지만, $(())()$ 경우는 왼쪽과 오른쪽 괄호갯수는 3개로 같음에도 짹이 맞지 않는다
 - iii. 왼쪽과 오른쪽 괄호가 섞인 괄호 시퀀스를 입력으로 받아 짹이 맞는 괄호라면 True를, 아니면 False를 출력하는 코드를 작성해보자
 - 1. 괄호 시퀀스 $s = \dots(\dots)\dots$ 처럼 중간에 (\dots) 가 등장한다고 해보자. 전체 시퀀스 짹이 맞기 위해선, 이 두 괄호 안에 포함된 짹은 시퀀스 \dots 역시 괄호 짹이 맞아야 함을 알 수 있다
 - 2. s 의 가장 왼쪽 괄호부터 차례대로 살펴보면서 현재까지 본 괄호들을 최대한 짹을 맞춘다고 가정하자
 - 3. 왼쪽 괄호 (이 등장하면 짹이 되는 오른쪽 괄호)이 나중에 반드시 등장해야 하고, 두 괄호안에 포함되는 괄호들도 짹이 맞아야 한다
 - 4. 짹이 맞는 괄호들은 즉시 시퀀스에서 빼버린다면, 오른쪽 괄호 (이 등장할 때, 자신의 왼쪽 괄호가 가장 최근에 짹을 못 맞춘 괄호로 기다리고 있어야 한다. 즉, 가장 최근에 짹을 못 맞춘 왼쪽 괄호가 가장 빨리 오른쪽 괄호와 짹을 맞추기 때문에 스택의 LIFO 원칙과 일치한다
 - iv. Pseudo 코드:
- ```
def parChecker(parSeq: str): # 매개변수: 타입 형식으로 타입 표기 가능
 S = Stack()
 for symbol in parSeq:
 if symbol == "(":
 S.push(symbol)
 else: # symbol == ")"
 if S.isEmpty(): # 짹이 모자람!
 return False
 else: # symbol == ")"이고,
 # 스택에 저장된 건 "(" 뿐이므로
 S.pop()

 if S.isEmpty(): # 최종적으로 스택에 남아 있는 게 없어야 함
 return True
 else:
 return False
```
- v. [실습문제] 만약, 중괄호 {}, 대괄호 []도 함께 섞여있는 일반적인 경우에 대해서도 위의 코드를 수정해서 짹 맞추기 코드를 어렵지 않게 작성할 수 있다
    1. symbol이 {{ 중 하나면 스택에 push한다
    2. symbol이 )}] 중 하나라면, 스택의 top에 있는 왼쪽 괄호가 같은 종류여야 하고 같다면 pop한다

### g. 스택의 사용 예 2: infix 수식을 postfix 수식으로 변환하기

i. infix, prefix, postfix 수식이란?

1. infix 수식은 일반적인 수식 작성법:  $2 + 3$  처럼 연산자가 가운데
2. prefix 수식은  $+ 2 3$  처럼 연산자가 앞에
3. postfix 수식은  $2 3 +$  처럼 연산자가 뒤에 오도록 작성

| Infix             | Prefix        | Postfix         |
|-------------------|---------------|-----------------|
| $A + B * C + D$   | $+ + A * B C$ | $A B C * + D +$ |
| $(A + B) * (C+D)$ | $* + A B + C$ | $A B + C D + *$ |
| $A * B + C * D$   | $+ * A B * C$ | $A B * C D * +$ |
| $A + B + C + D$   | $+ + + A B C$ | $A B + C + D +$ |

ii. infix를 postfix로 변환하면 수식 계산이 더 편해진다. 왜 그런가?

iii. infix → postfix로 변환하는 세 단계:

1. 연산자 우선순위에 따라 괄호를 삽입한다 → 연산자마다 괄호 한 쌍씩 할당
2. 연산자를 자신의 괄호 쌍중에서 오른쪽 괄호의 바로 오른쪽으로 이동
3. 괄호를 모두 지운다

예:  $A + B * C \rightarrow (A + (B * C)) \rightarrow A B C * +$

iv. **입력**: +, -, \*, /, (, )와 영문 대문자가 섞인 infix 수식

**출력**: postfix 수식

v. 위의 방법처럼 괄호를 모두 삽입한 후에 변환하지 않고, 입력을 왼쪽부터 차례로 검사하면서 연산자의 올바른 위치를 찾아 나가는 방식을 채택한다

vi. **관찰 1**:  $A + B * C \rightarrow A B C * +$

1. 피연산자 (operand)의 순서는 그대로 유지한다
2. 수식의 왼쪽부터 고려하기 때문에, +를 \*보다 먼저 고려한다
3. +의 위치는 다음에 만나는 연산자 (여기선 \*)에 의해 결정된다.
4. \*이 + 보다 연산 순위가 높기 때문에 \* 다음에 +가 위치해야 한다 ⇒ +연산자의 입장에서 해석하면, +가 스택에서 대기하면서 자신보다 연산 순위가 낮은 연산자가 등장하면 스택에서 나가면 된다

vii. **관찰 2**:  $(A + B) * C \rightarrow A B + C *$

1. +는 \*보다 먼저 )를 만나기 때문에, ) 다음에 위치해야 올바르다

2. 즉, )의 우선순위가 \*보다 높기 때문에 + 연산자는 ) 다음에 위치하게 된다
- viii. 결국, 어떤 연산자의 위치는 자신보다 우선순위가 **높거나 같은** 연산자가 오른쪽에서 나타나면 계속 스택에서 대기하면 된다. 자신을 스택에서 꺼내줄 수 있는 연산자는 자신보다 우선순위가 낮아야 한다
- ix. 정리하면, 현재 연산자의 우선순위보다 같거나 높은 우선순위를 갖는 스택에 저장된 연산자를 모두 꺼내 현재 연산자보다 먼저 오도록 해야 한다
- x. Pseudo 코드: infix → postfix
1. 괄호와 연산자를 저장하기 위한 스택 **opstack = Stack()** 준비
  2. Postfix 수식(결과)을 저장하기 위한 리스트 **outstack = []** 준비
  3. 리스트 exp에는 항과 연산자가 infix 수식 형태(문자열)임
  4. **for each token in exp:** # 연산자와 피연사자를 token이라 부름
    - a. **if** token == operand: **outstack.append(token)**
    - b. **if** token == '(': **opstack.push(token)**
    - c. **if** token == ')':
      - a. **opstack**에 저장된 연산자를 (를 만날 때까지 계속 pop한 연산자를 **outstack**에 append함
    - d. **if** token in '+-\*': # 사칙 연산자 중에 하나라면
      - a. **opstack**에 있는 자신보다 우선순위가 **높거나 같은** 연산자는 차례로 모두 pop한 연산자를 **outstack**에 append함
      - b. **opstack.push(token)**
  5. **opstack**에 남아 있는 연산자를 모두 pop한 후 **outstack**에 append함
  6. **print(outstack)** # output
- i. 예:  $exp = (A + B) * C - D$ 에 위 코드를 적용해보자
1. **token == '('** → **opstack = ['()']**
  2. **token == 'A'** (operand) → **outstack = ['A']**
  3. **token == '+'** → **opstack = ['()', '+']**
  4. **token == 'B'** → **outstack = ['A', 'B']**
  5. **token == ')'** → '()'까지 pop해서 **outstack**에 → **opstack = [], outstack = ['A', 'B', '+']**
  6. **token == '\*'** → **opstack = ['\*']**
  7. **token == 'C'** → **outstack = ['A', 'B', '+', 'C']**
  8. **token == '-'** → 우선순위 높은 피연산자 모두 pop해서 **outstack**에 → **opstack = ['-'], outstack = ['A', 'B', '+', 'C', '\*']**
  9. **token == 'D'** → **outstack = ['A', 'B', '+', 'C', '\*', 'D']**
  10. 단계 5 → **opstack = [], outstack = ['A', 'B', '+', 'C', '\*', 'D', '-']**  
**postfix = A B + C \* D -**

ii. [직접 해보기] 예  $exp = (A + B) * (C - D) + E$  를 위의 예처럼 해보기

b. 스택의 사용 예 3: postfix 수식을 실제로 계산해보기

- i. 이제 postfix 수식이 주어졌을 때, 수식 결과를 실제로 계산해보자
- ii. 예를 들어,  $3 \ 2 + 4 *$  가 주어지면, 어떻게 해야할까?
  1. 왼쪽부터 보면서, 숫자(operand)가 나오면 스택에 저장한다
  2. (이항) 연산자(operator)가 나오면 스택에 있는 두 수를 꺼내 연산을 수행하고 그 결과를 다시 스택에 넣는다 (**왜?**)
  3. (단항) 연산자가 나오면 스택에 있는 하나의 수를 꺼내 연산을 수행하고 그 결과를 다시 스택에 넣는다
  4. 이 과정을 마친 후, 스택에 남아 있는 값이 수식의 계산 결과이다
- iii. 위의 예에 대해, 스택의 변화를 보면 다음과 같다
  1.  $[] \rightarrow [3] \rightarrow [3, 2] \rightarrow + \rightarrow [5] \rightarrow [5, 4] \rightarrow * \rightarrow [20]$

c. 이제 infix 수식 입력  $\rightarrow$  postfix 수식으로 변환  $\rightarrow$  계산으로 이어지는 전체 계산기 코드를 완성해보자

아래와 같은 세 개의 함수를 작성합니다.

```
get_token_list(expr):
 ○ expr은 문자열로 수식을 나타낸다
 ○ expr을 연산자와 피연산자 토큰들로 나눈 후 리스트에 담아 리턴한다. (주의:
 연산자, 피연산자 모두 문자열 형식으로...)
 ○ [주의 1] 연산자는 +, -, *, /, ^ 5가지 이항 연산자만 다루고, 연산자와
 피연산자 사이에 공백이 올 수도 있고 공백이 없을 수도 있음에 유의하자!
 ○ [주의 2] 피연산자는 float로 변환한다
 ○ [주의 3] 한 자리 이상의 실수가 피연산자가 등장할 수 있다!
```

```
infix_to_postfix(token_list):
 ○ token_list는 수식의 연산자 피연산자가 infix 수식의 순서대로 저장된
 리스트이다
 ○ 이 token_list를 postfix 수식으로 변환하고 그 결과를 리스트에 담아
 리턴한다
```

```
compute_postfix(token_list):
 ○ postfix 형식의 token_list에 대한 계산 값을 리턴한다
```

**입력:** infix 수식의 문자열을 입력받는다

**출력:** 입력 수식을 계산한 (실수) 값을 출력한다

#### d. 스택 예 4: Nearest Smaller Element 계산하기 (NSE 문제)

- e. 리스트 A에 n개의 정수가 주어진다. 각 A[i]에 대해, A[0] ... A[i-1] 중에서 A[i]보다 작은 값 중에서 A[i]에 가장 가까운 값을 구해 B[i]에 저장하라 (즉, 각 A[i]에 대해 left-nearest smaller element를 계산하라는 의미)

i. 단, B[0] = None (*not defined*)

- f. 단순 방법:  $O(n^2)$  시간 - A[i]의 왼쪽의 값을 차례로 방문하면서 처음으로 A[i]보다 작은 값을 찾아 B[i]에 저장함

```
for i in range(n):
 j = i - 1
 while j >= 0:
 if A[j] < A[i]:
 break
 j = j - 1
 B[i] = A[j]
```

- g. 빠른 방법:  $O(n)$  시간, 스택 S를 마련해서 아래 단계를 수행. S[0] = None으로 초기화 한 후 사용

i.  $S[-1] < A[i]$  이면  $B[i] = S[-1]$ , S.push(A[i]) 수행

ii.  $S[-1] > A[i]$  이면, A[i]보다 작은 값이 나올 때까지 S.pop()한 후,  $B[i] = S[-1]$ , S.push(A[i]) 수행

iii. 이 과정을 반복하면 해를 정확히 구할 수 있는 이유는 무엇인가?

iv. 예: A = [1, 3, 4, 2, 5, 3, 4, 2]

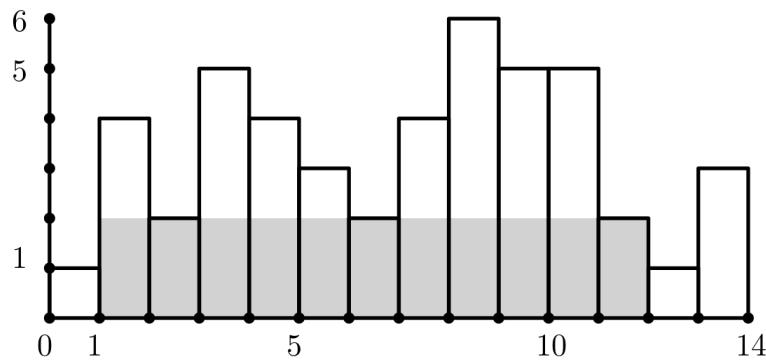
$S=[1] \rightarrow S=[\cancel{1}, 3] \rightarrow S=[\cancel{1}, \cancel{3}, 4] \rightarrow (2) \rightarrow S=[\cancel{1}, 2] \rightarrow S=[1, \cancel{2}, 5] \rightarrow (3) \rightarrow S=[1, \cancel{2}, 3] \rightarrow S=[1, 2, \cancel{3}, 4] \rightarrow (2) \rightarrow S=[\cancel{1}, 2] \rightarrow B = [\text{None}, 1, 3, 1, 2, 2, 3, 1]$

v. 수행시간은?  $O(n)$

1. amortized 수행시간 분석의 간단한 예

#### h. 스택 예 5: 스카이라인 아래의 가장 큰 직사각형 찾기

- i. 아래 그림처럼 폭이 1인 빌딩의 높이 n개가 차례로 주어진다. 빌딩들의 스카이라인 (위쪽 경계를 모아 놓은 것) 아래에 포함된 여러 직사각형 중에서 면적이 가장 큰 직사각형을 찾는 문제



- ii. 위의 그림에서는 왼쪽 끝에서 두 번째 빌딩부터 오른쪽 끝에서 세 번째 빌딩의 스카이라인 아래에 있는 색칠된 직사각형의 면적이 가장 크다. 이 직사각형의 높이는 2가되고 폭은 11이므로 면적은 22가 된다
- iii. 앞에서 설명한 NSE 문제와 본질적으로 다르지 않다!
- iv. 가장 왼쪽 빌딩부터 오른쪽으로 진행하면서 현재 빌딩을 (마지막으로) 포함하는 직사각형의 최대 면적을 계산하고 현재까지 계산된 가장 큰 면적을 유지하는 방식으로 진행하면 된다
- v. 위의 그림을 보면, 가장 왼쪽 빌딩을 0번이라고 하면, 현재는 **11번 빌딩을 포함하는** 최대 직사각형 면적을 계산하는 단계이다. 11번 빌딩에서 직사각형이 끝나야 하므로 직사각형이 관통하는 왼쪽에 있는 빌딩들은 모두 11번 빌딩의 높이보다 같거나 작아야만 한다. 결국, 11번 빌딩 높이보다 큰 빌딩은 고려할 필요가 없다

## 7. [🎤] 스택과 관련된 인터뷰 문제 두 가지

- a. [질문 1] 스택을 하나 또는 두 개를 사용해서 push, pop, min 세 연산 모두  $O(1)$  시간에 수행되도록 하려면? 여기서 min 연산은 현재 push된 값 중에서 최소 값을 리턴하는 연산을 의미한다
  - i. [힌트] push 할 때마다 현재까지의 최소값을 다른 스택에 저장해서 기억!
  - ii. 가능하면 사용하는 메모리의 양을 최소화하는 방법을 마련해 볼 것!
- b. [질문 2] 스택을 정확히 2개 이용해서 큐를 구현하라. 즉, 스택 두 개만 사용해서 enqueue, dequeue를 구현해야 한다. 단, 연산 시간은 최소화해야 한다
  - i. [힌트] 연산시간은 파이썬 리스트의 append 시간 분석처럼 평균 시간으로  $O(1)$  시간에 가능하도록 하면 어떨까?

### 3. Queue

- a. Queue 자료구조는 가장 먼저 들어온 값이 먼저 나가는 원칙 - First-In First-Out 원칙에 따라 삽입(enqueue)과 삭제(dequeue) 연산이 이루어진다
- b. 지원 연산은 enqueue, dequeue, isEmpty, front, len이며 모두 O(1) 시간에 수행되어야 한다
  - i. enqueue(key): 값 key를 큐의 오른쪽 끝 빈 칸에 삽입 (스택에서의 push와 완전히 같은 연산임)
  - ii. dequeue(): 가장 왼쪽에 저장된 (= 가장 오래된) 값을 삭제 후 리턴. 가장 오래된 값이 저장된 인덱스를 알고 있어야 한다. 이 인덱스를 front\_idx라고 하자
  - iii. front(): 가장 왼쪽에 저장된 (= 가장 오래된) 값을 (삭제하지 않고) 리턴

c. 코드:

```

class Queue:
 def __init__(self):
 self.items = [] # 데이터 저장을 위한 리스트 준비
 self.front_idx = 0 # 다음 dequeue될 값의 인덱스 기억

 def __len__(self):
 return len(self.items) - self.front_index

 def enqueue(self, key):
 self.items.append(key)

 def dequeue(self):
 if self.front_idx == len(self.items): # 이 기준이 왜 동작?
 print("Queue is empty")
 return None # dequeue할 아이템이 없음을 의미

 else: # dequeue는 front_idx를 조정해서 삭제 효과를 얻음
 x = self.items[self.front_idx]
 self.front_idx += 1 # 다음에 dequeue될 값의 인덱스 조정
 return x

 def front(self):
 # dequeue 코드와 동일 (단, front_index를 1 증가하는 코드만 삭제)

 if self.front_idx == len(self.items): # 이 기준이 왜 동작?
 print("Queue is empty")
 return None # dequeue할 아이템이 없음을 의미
 else:
 return self.items[self.front_idx]
```

- d. `dequeue`를 상수시간에 실행하기 위해선, `dequeue`가 될 값의 인덱스(index)를 저장하고 관리해야 한다 (위의 코드에서 `front_idx` 변수) → `dequeue`가 되면, 인덱스 값이 하나 증가하여 다음 `dequeue` 될 예정인 값의 인덱스를 가리키도록 관리한다 → `self.front_idx += 1` 코드
- e. [?] 위의 다섯 개의 연산의 수행시간을 Big-O로 표기하면 어떻게 되는가? 모두 (평균)  $O(1)$  시간이면 충분함
- f. **주의1:** `Queue` 클래스에서 값을 저장하는 자료구조로 리스트를 사용했는데, `dequeue` 연산을 위해 `pop(0)`라는 리스트 연산을 사용할 수도 있었다. 그런데, 왜 사용하지 않았을까? `pop(0)` 연산은 0번째 값을 삭제한 후, 1번째, 2번째, … 값을 차례대로 왼쪽으로 한 칸씩 이동해야 한다. 당연히  $n$ 개 값이 큐에 저장되어 있다면  $O(n)$  시간이 필요하다. 따라서 가장 먼저 들어와 있는 값이 저장된 인덱스를 위한 `front_idx`라는 멤버를 사용해야 한다. `front_idx`의 값은 `dequeue` 연산이 이루어질 때마다 1씩 증가하면 된다

큐를 리스트로 구현했고 `enqueue` 연산을 리스트의 `append` 연산을 사용했기 때문에 `enqueue` 연산이 평균  $O(1)$  시간이 걸린다. 만약 **최악의 경우**  $O(n)$  시간을 보장하고 싶다면, 리스트의 크기를 미리 정해서 아래와 같이 생성 함수를 부르면 된다. 단, 다음에 `enqueue`될 빈 칸의 인덱스 (보통 `rear_idx`라 부름)가 필요하고 `enqueue`에서 관리해야 한다. 이 경우에는 큐의 빈 칸이 없을 수 있기 때문에 먼저 빈 칸이 있는지 검사하는 조건문이 와야 한다

```
def __init__(self, size=8):
 self.items = [None]*size
 self.front_idx = 0
 self.rear_idx = 0
```

- g. **주의2:** `front_idx` 멤버를 사용하면  $O(1)$  시간에 `dequeue` 연산이 가능하지만, 단점도 존재한다. 어떤 단점일까? (잠시 생각을…) 예를 들어, 100개의 값이 `enqueue` 된 후, 다시 100번의 `dequeue` 되었다고 해보자. 그러면 `items`의 100개 칸이 이후에는 다시 사용되지 않고 낭비된다. 메모리 낭비를 줄이려면 원형 큐 (circular queue)를 사용할 수 있다

이 경우에는 `enqueue`와 `dequeue`에서 각각 `front_idx`와 `rear_idx`를 1씩 증가할 때, `size`로 나눈 나머지로 계산해야 한다

메모리 부족을 신경쓰지 않고 사용하려면 연결리스트 (linked list)로 큐를 구현하면 된다. 연결리스트에 대한 내용은 다음 장에서 자세히 설명되어 있다

### h. 큐 사용 예 1: Josephus game

- i. 1, 2, ... n번까지 원형 테이블에 앉아있다. 1번부터 시작해서 k번째 사람이 탈락하는 게임을 한다
- ii. 예: n = 6이고, k = 2인 경우, 탈락하는 순서가 2 → 4 → 6 → 3 → 1가 되어 최종적으로 5이 생존한다
  - 첫 번째 라운드에서 홀수 번호가 모두 탈락한다
  - 두 번째 라운드에서는 대략 절반의 사람만 참여하는 게임이 된다. (n이 짹수면 1번 부터 시작하고, n이 홀수면 3번부터 시작하는 식이다)
  - $W(n) = n$ 명이 게임할 때 우승자의 번호
    - $n = 2, 3, \dots, 16$ 까지  $W(n)$ 을 작성하면서 패턴을 찾아보자
  - n이 짹수인 경우와 n이 홀수인 경우로 나눠서  $W(n)$ 에 관한 점화식을 세워보자 (답은 아래와 같다. 왜 그럴까?)
    - $W(2k) = 2W(k)$
    - $W(2k+1) = 2W(k) + 1$
  - 자세한 설명: <https://www.youtube.com/watch?v=uCsD3ZGzMgE>
- iii.  $W(n, k) = n$ 명이 일반적인  $k \geq 2$ 에 대해 게임을 하는 경우의 우승자 번호라고 정의하면, 점화식을 어떻게 세울 수 있을까? 점화식을 계산하는 데 필요한 시간은?
  - 자세한 답은 위키피디아 [Josephus problem](#)을 참조하기 바란다
- iv. 큐를 이용해 아래와 같은  $O(n^2)$  시간 알고리즘으로  $W(n, k)$ 를 계산할 수 있다
  - 매 라운드마다 숫자를 꺼내서(dequeue)하고 다시 넣는(enqueue)하는 과정을  $(k-1)$ 번 반복한 후, 그 다음 번 수인 k번째 수를 제거(dequeue)하면 된다

### v. 코드:

```
from stack_queue import Queue
def Josephus(n, k):
 Q = Queue()
 for v in range(1, n+1):
 Q.enqueue(v)

 while len(Q) > 1:
 for i in range(1, k):
 Q.enqueue(Q.dequeue())
 Q.dequeue() # k-th number is deleted

 return Q.dequeue() # len(Q) == 1
```

i. 큐 사용 예 2: Radix 정렬

- i. Radix 정렬에서는 0부터 9까지 값을 대표하는 10개의 큐를 이용한다. 입력 값의 첫 번째 자리 값에 해당하는 큐에 enqueue를 하고, 다시 dequeue를 해서 모은 후 두 번째 자리 값에 대해 enqueue, dequeue 과정을 반복한다
- ii. Stack을 사용할 수도 있지만, 그렇게 하면 **stable** 정렬 성질을 만족하지 못한다
- iii. 자세한 내용은 알고리즘 교재의 정렬 편을 참고하기 바란다

j. 큐 사용 예 3: BFS (너비 우선 탐색)

- i. 그래프의 순회(traversal) 방법 중 하나로, 현재 방문 중인 노드의 인접한 노드를 차례대로 방문해야 하는데, 자기 순서가 올 때까지 큐에 enqueue되어 대기한다. dequeue가 되는 순간 방문이 되는 것이며, 다시 인접한 노드들을 큐에 enqueue하게 된다
- ii. 자세한 내용은 알고리즘 교재의 그래프 > 순회 편을 참고하기 바란다

#### 4. Dequeue (double ended queue)

- a. 원쪽과 오른쪽에서 모두 삽입과 삭제가 가능한 큐 - 두 가지 버전의 push와 pop 연산을 구현하면 되고, 나머지 연산은 Stack, Queue 클래스와 유사하게 구현한다 (구체적인 클래스 구현은 연습문제로)
- b. Python에서는 collections라는 모듈에 deque란 클래스로 dequeue가 이미 구현됨
  - i. 덱(deck)으로 발음
  - ii. 오른쪽 push = **append**, 왼쪽 push = **appendleft**
  - iii. 오른쪽 pop = **pop**, 왼쪽 pop = **popleft**

```
from collections import deque
>>> d = deque('ghi') # make a new deque with three items
>>> d.append('j') # add a new entry to the right side
>>> d.appendleft('f') # add a new entry to the left side
>>> d # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop() # return and remove the rightmost item
'j'
>>> d.popleft() # return and remove the leftmost item
'f'
>>> list(d) # list the contents of the deque
['g', 'h', 'i']
>>> print(d[0], d[-1]) # peek at leftmost, rightmost items
```

#### c. deque 사용 예 1: Palindrome 검사 코드

- i. Palindrome은 왼쪽부터 읽어도 오른쪽부터 읽어도 같은 문자열을 말한다
  - 예: 기러기, radar, madam 등
- ii. 방법 1: 문자열 s와 s를 reverse한 문자열이 같다면 palindrome임
  - reversed(s)는 문자열 s를 거꾸로 한 iterable 객체임
 

```
>>> s == ''.join(reversed(s))
```
- iii. 방법 2: i = 0, j = n-1부터 시작해 s[i] == s[j]이면 i는 하나 감소시키고, j는 하나 증가하는 방식으로 i == j이거나 i > j가 될 때까지 반복하면 된다
- iv. 방법 3: dequeue를 사용함
  - dequeue에 문자열을 저장한 후 양쪽에서 하나씩 빼면서(pop과 popleft 이용) 같은지 비교하는 것을 반복함. 다르다면 계속할 필요없이 palindrome이 아님
  - Pseudo 코드:

```
from collections import deque
def check_palindrome(s):
```

```

dq = deque(s)
palindrome = True
while len(dq) > 1:
 if dq.popleft() != dq.pop():
 palindrome = False
return palindrome

```

#### d. deque 사용 예 2: Sliding Window Minimum

- i. 리스트 A에 저장된 n개의 정수 값에 대해, 길이가 k인 윈도우 (window)를 왼쪽에서 오른쪽으로 이동하면서 해당 윈도우에 포함된 값 중에서 최소 값을 새로운 리스트 B에 저장하라
  - ii.  $A = [2, 1, 4, 5, 3, 4, 6, 2], k = 4$  ( $B[0]=B[1]=B[2]=None$ )
 

|       |     |                  |            |
|-------|-----|------------------|------------|
| ----- | → 1 | $dQ = [1, 4, 5]$ | $B[3] = 1$ |
| ----- | → 1 | $dQ = [1, 3]$    | $B[4] = 1$ |
| ----- | → 3 | $dQ = [3, 4]$    | $B[5] = 3$ |
| ----- | → 3 | $dQ = [3, 4, 6]$ | $B[6] = 3$ |
| ----- | → 2 | $dQ = [2]$       | $B[7] = 2$ |
  - iii. 윈도우를 왼쪽에서 오른쪽으로 한 칸 이동하면, 왼쪽 수가 하나 빠지고 오른쪽 수가 하나 윈도우에 들어온다. 윈도우에 포함된 수들을 관리하는 deque dQ를 마련한 후, 아래 과정을 반복 실행한다
    - dQ에는 윈도우에 포함된 값부터 오름차순으로 저장함.
    - dQ.front의 값이 윈도우를 떠나면 dQ.popleft()
    - 현재  $A[i]$  값보다 크거나 같은 수를 모두 dQ.pop() 실행 (이유는?)
    - dQ.append( $A[i]$ ),  $B[i] = dQ.front()$
  - iv. 위의 과정을 거치면 정답을 제대로 계산할까? 그 이유는?
- e. [Just-for-fun: Python 응용] deque를 이용한 기타 음표(note) 만들어보기
  - i. [참고 도서] Python Playground, 4장, Mahesh Venkitachalam
  - ii. 예를 들어, D 노트는 146.83 Hz(헤르쯔)의 주파수(초당 진동수)에 의해 결정된다. 각 노트는 고유의 주파수가 있다
 

C4: 261.6, E-flat: 311.1, F: 349.2, G: 392.0, B-flat: 466.2
  - iii. 주파수가 주어지면 해당 노트를 만드는 방법 중에 Karplus-Strong 알고리즘을 소개한다
    - 입력: frequency (실수, float)
    - 출력: wave 파일
  - iv. Karplus-Strong 알고리즘
    - 변수 소개

- sampleRate = 44100 (보통 CD의 값이 44,100Hz)
- nSamples = sampling\_rate
- buf = [-0.5, 0.5] 구간의 랜덤 값으로 초기화된 크기가 (sampleRate/frequency)인 deque
- samples = 실제 재생될 값이 저장될 리스트 (크기는 nSamples)

- **for i in range(nSamples):**

```
samples[i] = buf[0] # 0번째 값을 sample로 복사
avg = (buf[0]+buf[1])/2
buf.append(avg*0.996) # 조금씩 작은 값이 append
buf.popleft() # buf[0]를 제거
```
- buf[0]이 삭제되는 대신  $0.996 \times (\text{buf}[0]+\text{buf}[1])/2$  값이 오른쪽 끝에 삽입되기에 buf의 값의 개수는 변함이 없음!

v. 코드: [출처: 위의 참고 도서 p.62]

```
import numpy as np
import random, wave
from collections import deque
def generateNote(freq):
 nSamples = 44100
 sampleRate = 44100
 N = int(sampleRate/freq)
 # initialize ring buffer
 buf = deque([random.random() - 0.5 for i in range(N)])
 # init sample buffer
 samples = np.array([0]*nSamples, 'float32')
 for i in range(nSamples):
 samples[i] = buf[0]
 avg = 0.995*0.5*(buf[0] + buf[1])
 buf.append(avg)
 buf.popleft()
 # samples to 16-bit to string
 # max value is 32767 for 16-bit
 samples = np.array(samples * 32767, 'int16')
 return samples.tobytes()

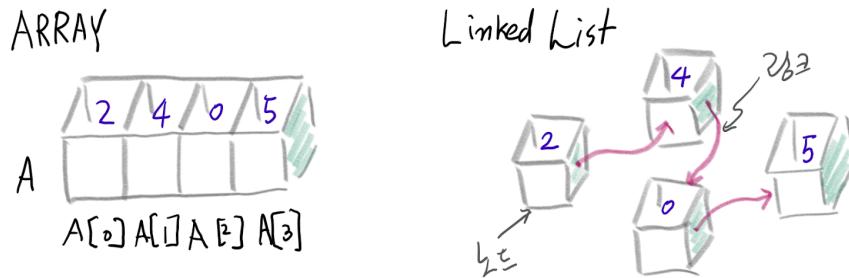
def writeWAVE(fname, data):
 # open file
 file = wave.open(fname, 'wb')
 # WAV file parameters
 nChannels = 1 # mono channel
 sampleWidth = 2 # 2바이트 샘플 = 16 bits
 frameRate = 44100
 nFrames = 44100
 # set parameters
```

```
file.setparams((nChannels, sampleWidth, frameRate,
 nFrames, 'NONE', 'noncompressed'))
file.writeframes(data)
file.close()

generate D note
writeWAVE("D.wav", generateNote(146.83))
```

### 3. Linked List

1. 연결리스트(linked list)는 파이썬의 리스트(list)와는 이름만 비슷하지 전혀 다른 개념이다
  - a. 파이썬의 list는 C의 array와 유사한 개념으로 index를 통해 접근하고 수정한다
  - b. 연결리스트는 노드(node)가 링크(link)에 의해 기차처럼 연결된 순차(sequential) 자료구조로 링크를 따라 원하는 노드의 데이터를 접근하고 수정한다



2. 노드(node): 실제 값을 위한 data 정보 (보통 key 값을 저장)와 인접 노드로 향하는 link 정보로 구성된 클래스
3. 배열은 크기가 미리 정해지고 배열에 저장된 값을 반복적으로 참조(read/write)하는 경우에 적합한 순차적 자료구조이고 연결리스트는 크기가 정해지지 않고 삽입과 삭제가 자주 반복되는 경우에 적합한 자료구조라고 생각하자

### 4. 한방향 연결리스트(Singly Linked List)

- a. 노드들이 한쪽 방향으로만(next 링크를 따라) 연결된 리스트
  - i. 가장 앞에 있는 노드를 특별히 **head** 노드라 부르고, head 노드를 통해 리스트의 노드를 접근한다 (head 노드부터 시작해 링크를 계속 따라가면 모든 노드를 접근 할 수 있으므로, head 노드가 연결리스트를 대표한다고 말할 수 있음)
  - ii. 가장 뒤에 있는 노드는 다음 노드가 없기 때문에 그 노드의 head 링크는 None을 저장함. 즉, next 링크가 None이라면 그 노드가 마지막 노드임
- b. 노드 클래스

```
class Node:
 def __init__(self, key=None, value=None):
 self.key = key # 노드를 다른 노드와 구분하는 key 값
 self.value = value # 필요한 경우 추가 데이터 value
 self.next = None # 다음 노드로의 링크 (초기값은 None)

 def __str__(self): # print(node)인 경우 출력할 문자열
 return str(self.key, self.value))
```

### c. 한방향 연결 리스트 클래스

```

class SinglyLinkedList:
 def __init__(self):
 self.head = None # 연결리스트의 가장 앞의 노드 (head)
 # 초기값은 None
 self.size = 0 # 리스트의 노드 개수 (필요하다면)

 def __iter__(self): # generator 정의
 # 연결리스트의 노드를 for 문 형식으로 접근
 # 위해
 v = self.head
 while v != None:
 yield v
 v = v.next

 def __str__(self): # 연결 리스트의 값을 print 출력
 return " -> ".join(str(v) for v in self)
 # generator를 이용해 for 문으로 노드를 순서대로
 # 접근해서 join 함 - key 값 사이에 -> 넣어 출력
 # 위 문장의 의미를 해석했다면, 최소 파이썬 중급 이상

 def __len__(self):
 return self.size # len(A) = A의 노드 개수 리턴

```

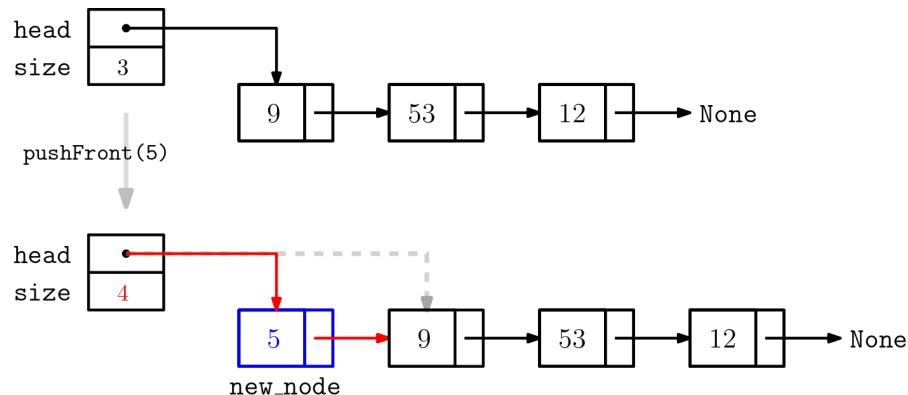
### d. 지원 연산: 삽입, 삭제, 탐색 등의 연결 리스트를 수정할 수 있는 연산 제공

L = SinglyLinkedList()

- i. L.pushFront(key): key 값을 갖는 새 노드를 L의 가장 앞에 삽입
  - L.head가 변경되어야 함
- ii. L.pushBack(key): key 값을 갖는 새 노드를 L의 가장 뒤에 삽입
- iii. L.insertAfter(nkey, xkey): xkey 값을 저장한 노드 다음에 새로운 값 nkey를 삽입한다. (search 연산 활용)
 

[주의] 새로운 키 값을 삽입하는 위의 세 함수는 모두 새로운 노드를 생성해 리스트에 삽입하게 되는데, 마지막에 이 새로운 노드를 리턴한다고 가정한다
- iv. L.popFront(): L의 첫 노드(head 노드)를 삭제한 후 key 값을 리턴. 빈 리스트라면 None 리턴
- v. L.popBack(): L의 마지막 노드를 삭제한 후 key 값을 리턴. 빈 리스트라면 None 리턴
- vi. L.search(key): L에서 key 값을 갖는 노드를 찾아 리턴
- vii. L.removeKey(key): key 값을 찾아 삭제
- viii. L.remove(v): L에서 노드 v를 제거. removeKey는 key 값을, remove는 노드를 제거함에 유의

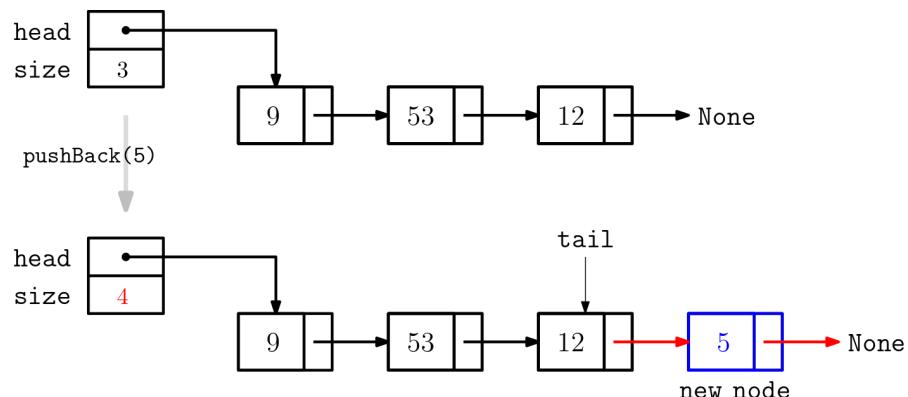
e. 삽입 연산: pushFront vs. pushBack



- i. pushFront는 현재 head 노드 앞에 새로운 노드를 생성해 삽입한다. 삽입된 노드가 새로운 head 노드가 되어야 한다

```
def pushFront(self, key, value=None):
 new_node = Node(key, value) # 새 노드 생성
 new_node.next = self.head # head 노드 앞에 삽입
 self.head = new_node # 새로운 head 노드 지정
 self.size += 1 # 노드 개수 증가
 return new_node # 삽입된 노드 리턴
```

- ii. pushBack은 그림에서 보듯 마지막 노드(tail)를 다음에 삽입이 되므로 tail 노드의 next 링크가 새로운 노드로 변경되어야 한다
- [주의] tail 노드가 None인 경우라면, 즉, 빈 리스트라면 새로운 노드가 리스트의 head 노드가 된다



```
def pushBack(self, key, value=None):
 new_node = Node(key, value)
 if self.size == 0: # empty list
 self.head = new_node # new_node가 head가 됨
 else:
 tail = self.head
 while tail.next != None: # tail 노드 찾기
 tail = tail.next
 tail.next = new_node
```

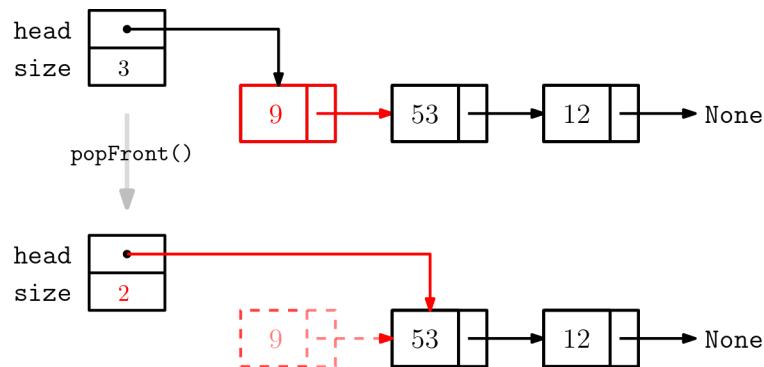
```
 self.size += 1
 return new_node
```

- iii. `insertAfter(nkey, xkey)`는 `xkey`를 포함한 노드의 다음에 `nkey`를 갖는 노드를 삽입하는 함수이다. 이를 위해서 우선 `xkey`의 노드를 `search`해야 한다. `search` 함수를 호출해 `xkey`를 포함한 노드 `x`를 찾고 `nkey`를 포함한 노드 `new_node`를 다음 노드로 연결하면 된다. `xkey`가 리스트에 없을 수도 있는데, 이 경우에는 `head` 노드에 삽입(= `pushFront(nkey)`)하는 것으로 처리한다

```
def insertAfter(self, nkey, xkey):
 x = self.search(xkey)
 if x == None:
 return self.pushFront(nkey)
 else:
 new_node = Node(nkey)
 new_node.next = x.next # (1)
 x.next = new_node # (2)
 self.size += 1
 return new_node
```

#### f. 삭제 연산: `popFront` vs. `popBack`

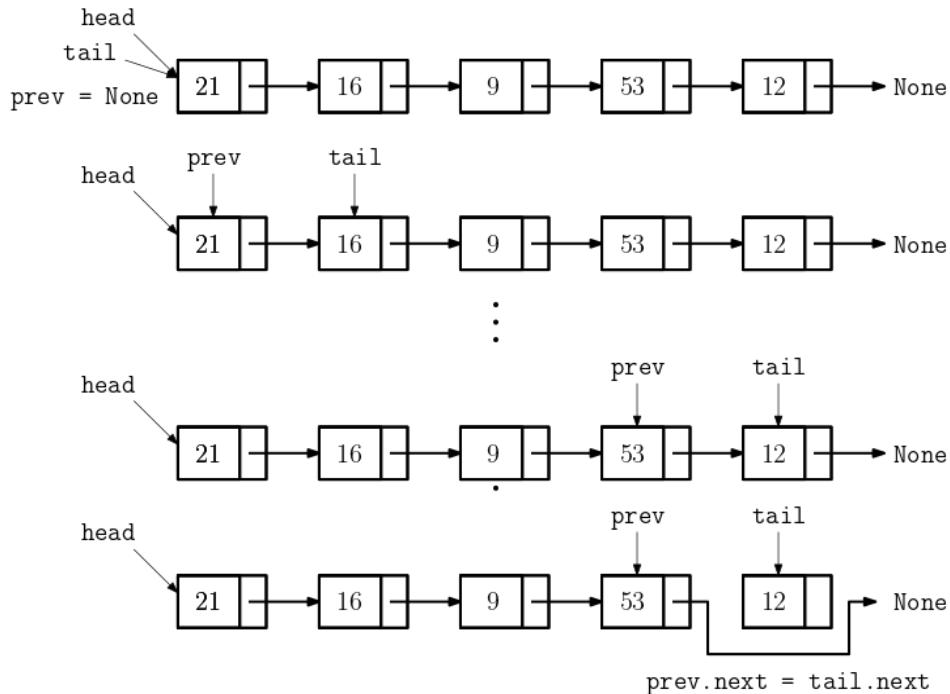
- i. `popFront`은 리스트의 `head` 노드를 삭제하고 `head` 노드의 `key`값을 반환하는 함수로, 두 가지로 나눈다: (1) 빈 리스트라 지울 `head`가 없는 경우와 (2) 최소 하나 이상의 노드가 있는 경우



```
def popFront(self):
 if self.size == 0: # 경우 (1)
 return None # return None if it's empty
 else: # 경우 (2)
 x = self.head
 key = x.key # value = x.value
 self.head = x.next
 self.size = self.size - 1
 del x # delete x from memory
 return key # or return (key, value)
```

- ii. `popBack`은 `tail` 노드를 찾아 지우고 `key` 값을 반환하는 데, `tail` 노드의 전 노드의 링크를 수정해야 하므로 그 노드를 알아야 한다 (`tail` 노드의 전 노드를 `prev` 노드라 부르기로 하자)

- 아래 그림은 `prev` 노드와 `tail` 노드 (서로 인접한 두 노드)를 `prev = None`, `tail = head`으로 시작하여 `tail` 노드가 가장 마지막 노드에 도달할 때까지 움직인다. 그 때 `prev` 노드가 `tail` 직전 노드가 된다



```

prev, tail = None, self.head
while tail.next != None: # what if current.next == None
 prev = tail
 tail = tail.next

```

- iii. 세 가지 경우로 나눠서, (1) 빈 리스트인 경우, (2) 리스트에 노드가 하나만 있는 경우 (3) 리스트에 두 개 이상의 노드가 있는 경우를 각각 처리해야 한다

```

def popBack(self):
 if self.size == 0: # 경우 (1)
 return None
 else:
 prev, tail = None, self.head
 while tail.next != None:
 prev = tail
 tail = tail.next
 if prev == None: # 경우 (2)
 self.head = None
 else: # 경우 (3)
 prev.next = tail.next
 key = tail.key

```

```
def tail
 self.size -= 1
 return key # or return (key, value)
```

g. search(key): key값을 저장한 노드를 찾아 리턴하고 없으면 None 리턴

i. **방법 1:** head: head 노드부터 next 링크를 따라 가면서 뒤지는 방법

```
def search(self, key):
 v = self.head
 while v != None:
 if v.key == key:
 return v
 v = v.next
 return v
```

ii. **방법 2:** for 루프를 이용하는 방법 ← `__iter__(self)`에 의해 가능!

```
def search(self, key):
 for v in self:
 if v.key == key:
 return v
 return None
```

h. insertAfter(key, x): key 값을 갖는 새로운 노드를 노드 x 뒤에 삽입노드 리턴

```
def insertAfter(self, key, x):
 new_node = Node(key)
 if x:
 new_node.next = x.next
 x.next = new_node
 else: # insert at head if x == None
 new_node.next = self.head
 self.head = new_node
 self.size += 1
 return new_node
```

예를 들어, key 값이 3인 노드 다음에 key 값이 5인 새로운 노드를 삽입하고 싶다면 다음처럼 처리하면 된다

```
x = search(3)
insertAfter(5, x)
```

i. `remove(v)`: 노드  $v$ 를 리스트에서 제거. 성공하면 `True`, 실패하면 `False`를 리턴

- i. 경우 (1): 리스트가 비어 있거나 노드  $v$ 가 `None`인 경우 → do nothing
- ii. 경우 (2): 노드  $v$ 가 head 노드인 경우 → `popFront` 호출하여 처리
- iii. 경우 (3): 노드  $v$ 의 전 노드  $w$ 를 찾은 후 (`popBack`의 경우처럼)  $w.next = v.next$ 로  $w$ 의 링크를 수정한다

```
def remove(self, v):
 if self.size == 0 or v == None: # 경우 (1)
 return False
 elif v == self.head: # 경우 (2)
 self.popFront()
 return True
 else: # 경우 (3)
 w = self.head # v != self.head임에 유의
 while w.next != v:
 w = w.next
 w.next = v.next
 self.size -= 1
 return True
```

j. `removeKey(key)`: key 값을 찾아 리스트에서 제거하는 함수로 `remove` 함수를 이용

```
def removeKey(self, key):
 return self.remove(self.search(key))
```

k. [?] 연산의 시간복잡도 - 마우스로 해당 칸을 긁어보세요~

i. 대상이 되는 노드가 head 노드로부터  $k$ 번째 떨어진 노드라고 가정

|                          |        |                        |        |
|--------------------------|--------|------------------------|--------|
| <code>pushFront</code>   | $O(1)$ | <code>pushBack</code>  | $O(k)$ |
| <code>popFront</code>    | $O(1)$ | <code>popBack</code>   | $O(k)$ |
| <code>insertAfter</code> | $O(k)$ | <code>remove</code>    | $O(k)$ |
| <code>search</code>      | $O(k)$ | <code>removeKey</code> | $O(k)$ |

I. [?] 한방향 연결 리스트와 배열의 장단점은 무엇인가?

i. 장점:

- 어떤 노드  $x$ 의 다음에 새로운 노드를 삽입하는 연산 시간은  $O(1)$  시간이면 된다. 단, 노드  $x$ 를 알고 있어야 한다. 배열의 경우에는 삽입할 빈 칸을 만들기 위해, 그 이후의 값들을 한 칸씩 이동해야 해서 최악의 경우에는  $O(n)$  시간이 필요하다

ii. 단점:

- $k$ 번째 값을 배열에서는  $O(1)$ 시간에 알 수 있지만, 연결리스트에서는  $O(k)$  시간이 필요하고 최악의 경우에는  $O(n)$  시간이 걸린다
- 바로 전 노드 (prev 노드)로의 링크가 없어 삽입과 삭제의 시간이 많이 걸릴 수 있다 (예: popBack 연산)

m. [🎤 인터뷰 문제: 연결리스트 반대로 바꾸기] 한방향 연결리스트의 연결을 반대방향으로 바꾸는 함수 reverse() 구현하기

i. 연결리스트를 반대방향으로 연결한 후, 새로운 head 노드를 리턴

ii. 방법 1: 변수 몇 개만 추가로 사용해서 작성해보자

- 예: 연결리스트가  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ 에 대해 아래 코드를 적용해보자. 한 단계씩 따라가며 올바르게 동작하는지 살펴보자

```
def reverse1(self):
 a, b = None, self.head # a follows b one node behind
 while b:
 if b:
 c = b.next
 b.next = a
 a = b
 b = c
 self.head = a # 왜?
```

iii. 방법 2: 재귀 함수로 구현해보자 (가능하다면)

```
def reverse2(self,):
```

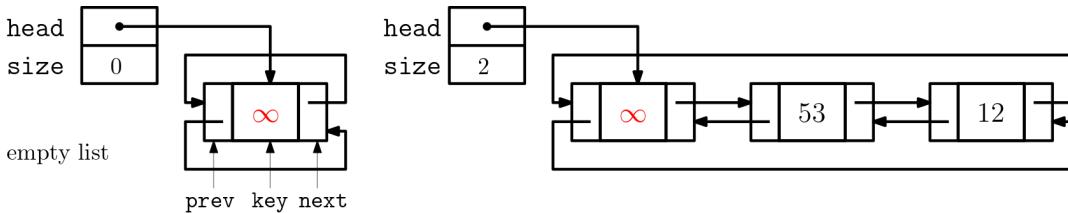
- n. [🎤 인터뷰 문제: Running technique]: 한방향 연결리스트에서 tail 노드와 prev 노드를 찾는 방법에 쓰인 기법으로 연결리스트의 전체 노드 개수를 모른다는 가정하에 두 개의 (포인터) 변수를 사용해 선형 탐색으로 원하는 위치의 노드를 계산하는 방법
- i. 인터뷰 문제로 많이 등장!
  - ii. `prev = None, tail = L.head`로 prev가 tail의 한 노드 뒤에서 따라가면서 tail이 실제 tail 노드에 도착하면, prev는 tail 노드 전 노드를 가르킴!
  - iii. [🎤 문제 1] `find_kth_node_from_tail(L, k)`:
    - tail 노드로부터 k번째 전에 있는 노드를 찾아라. (단, 리스트 L의 노드 개수는 모른다고 가정한다)

[힌트] prev, tail 노드 찾는 방법은  $k = 1$ 인 경우이므로 그 방법을 그대로 적용하면 된다
  - iv. [🎤 문제 2] `find_middle_node(L)`:
    - 리스트 L의 노드 개수를 모른다고 가정하고, L의 중간에 위치한 노드를 찾아라! (L의 노드 개수가 짝수면 중간의 두 노드 중 아무 노드라도 정답)

[힌트] 두 포인터의 거리를 항상 2배를 유지하도록 하면 된다. 어떻게?

## 5. 양방향 연결 리스트 (Doubly Linked List)

- a. 한방향 연결 리스트의 결정적인 단점은 다음 노드에 대한 링크(next)만 있어, 이전 노드를 알기 위해선 head 노드부터 차례로 탐색을 해야 한다는 것이다



- b. 한방향 리스트의 단점을 보완하는 다음과 같은 양방향 연결 리스트를 설계해보자

- 이전 노드로의 링크(prev)를 가지고 있어 왼쪽으로도 이동 가능하도록 한다
- 마지막 노드와 첫 노드가 서로 연결된 원형(circular) 리스트를 가정한다
- 첫 노드, 즉 head 노드는 항상 dummy 노드가 되도록 한다
  - dummy 노드**는 일종의 리스트의 처음을 구분할 수 있는 "marker"의 기능을 하는 특별한 노드이다. 따라서 빈 리스트는 위의 가운데 그림처럼 dummy 노드 하나로만 구성된다

- c. Node 클래스:

```
class Node:
 def __init__(self, key=None):
 self.key = key # 노드에 저장되는 key값
 self.next = self.prev = self # 자기로 향하는 링크

 def __str__(self): # print(node)인 경우 출력할 문자열
 return str(self.key)
```

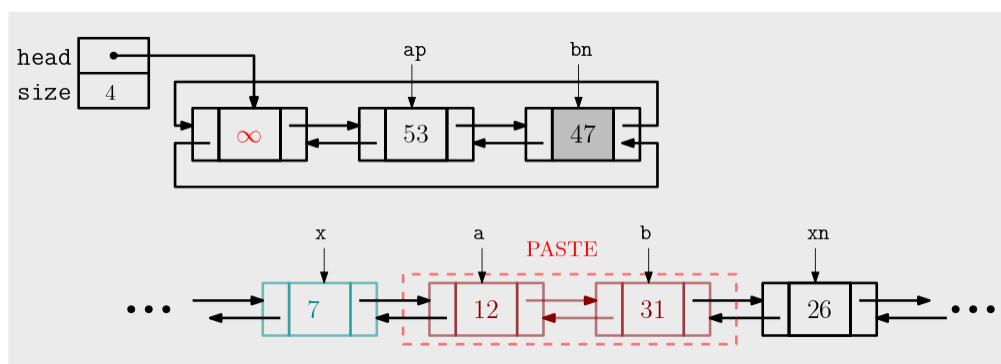
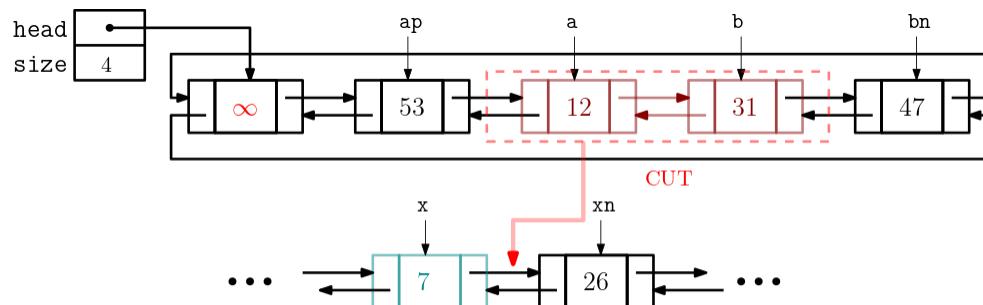
- d. 양방향 (원형) 연결 리스트 클래스

```
class DoublyLinkedList:
 def __init__(self):
 self.head = Node() # 빈 리스트는 dummy 노드만으로 표현
 self.size = 0 # 이 경우에는 size 값 관리가 필요

 def __iter__(self): # 한방향 리스트 참고하여 작성해보자
 ...
 def __str__(self): # 한방향 리스트 참고하여 작성해보자
 ...
 def __len__(self): # 한방향 리스트 참고하여 작성해보자
 ...
```

e. [중요] `splice(a, b, x)`: 다른 연산에 이용되는 **매우 중요한** 기본 연산

- i. 노드 a부터 노드 b까지를 떼어내(cut) 노드 x 뒤에 붙여(paste) 넣는 연산 (cut-and-paste 연산이라고 기억~)
  - 조건 1: a와 b가 동일하거나 a 다음에 b가 나타나야 함
  - 조건 2: head 노드와 x는 a와 b 사이에 포함되면 안 됨
- ii. [주의1] 노드 x가 a와 b의 노드일 수도 다른 리스트의 노드일 수도 있다
- iii. [주의2] DoublyLinkedList 클래스에 노드 개수를 관리하는 `size` 멤버 변수가 있다면, `splice` 이후의 노드 개수의 변화를 적절히 반영해야 한다



```

def splice(self, a, b, x):
 if a == None or b == None or x == None:
 return
 ap = a.prev # ap is the previous node of a
 bn = b.next # bn is next node of b
 xn = x.next

 # CUT [a..b]
 ap.next = bn
 bn.prev = ap

 # PASTE [a..b] after x
 xn.prev = b
 b.next = xn
 a.prev = x
 x.next = a

```

- f. 탐색 및 기본 연산 ([💪] 직접 구현해 볼 것)
- search(key): key 값 갖는 노드를 리턴하고, 없으면 None 리턴
  - isEmpty(): 빈 리스트면 True 아니면 False 리턴 ([❓] 어떻게 검사?)
  - first(), last(): 처음과 마지막 노드를 리턴. 빈 리스트면 None 리턴
- g. 이동과 삽입 연산: 매개변수 self는 생략함 (splice 함수를 호출하여 가능)
- moveAfter(a, x): 노드 a를 노드 x 뒤로 이동
    - splice(a, a, x)와 같음! (왜?)
  - moveBefore(a, x): 노드 a를 노드 x 앞으로 이동
    - splice(a, a, x.prev)와 같음!
  - insertAfter(key, x): 노드 x 뒤에 데이터가 key인 새 노드를 생성해 삽입
    - moveAfter(Node(key), x)와 같음!
  - insertBefore(key, x): 노드 x 앞에 데이터가 key인 새 노드를 생성해 삽입
    - moveBefore(Node(key), x)와 같음!
  - pushFront(key): 데이터가 key인 새 노드를 생성해 head 다음(front)에 삽입
    - insertAfter(key, self.head)와 같음!
  - pushBack(key): 데이터가 key인 새 노드를 생성해 head 이전(back)에 삽입
    - insertBefore(key, self.head)와 같음!
- h. 삭제 연산:
- remove(x): 노드 x를 제거
- ```
def remove(self, x):
    if x == None or x == self.head: # 조건 체크
        return False
    x.prev.next, x.next.prev = x.next, x.prev # x를 떼어냄
    return True
```
- removeKey(key): key를 찾아 제거. remove(search(key)) 형식으로 remove 함수를 이용함. 만약 같은 값이 여러 개라면 왼쪽 값을 제거함
 - popFront(): head 다음에 있는 노드의 데이터 값 리턴. 빈 리스트면 None 리턴
- ```
def popFront(self):
 if self.isEmpty():
 return None
 key = self.head.next.key
 self.remove(self.head.next)
 return key
```
- popBack(): head 이전에 있는 노드의 데이터 값 리턴. 빈 리스트면 None 리턴 : [💪] 직접 해볼 것

- i. 기타 연산: join, split
- i. join(another\_list): self 뒤에 another\_list를 연결함
  - ii. split(x): self를 노드 x 이전과 x 이후의 노드들로 구성된 두 개의 리스트로 분할
    - self = self.head  $\rightarrow \dots \rightarrow x.\text{prev}$ 까지이고, new\_list = x  $\rightarrow x.\text{next} \rightarrow \dots \rightarrow \text{self.head.prev}$ 까지가 됨
- j. 위의 연산 코드에서는 size (노드 개수를 저장하는 멤버 변수)에 대한 업데이트 부분은 생략되어 있음에 유의하자
- k. 양방향 연결리스트 연산의 최악의 경우의 시간복잡도 ( $n$ 개의 값이 저장되었다고 가정)

|                         |      |                           |          |
|-------------------------|------|---------------------------|----------|
| <b>moveAfter/Before</b> | 0(1) | <b>insertAfter/Before</b> | 0(1)     |
| <b>pushFront/Back</b>   | 0(1) | <b>popFront/Back</b>      | 0(1)     |
| <b>remove</b>           | 0(1) | <b>search</b>             | 0( $n$ ) |

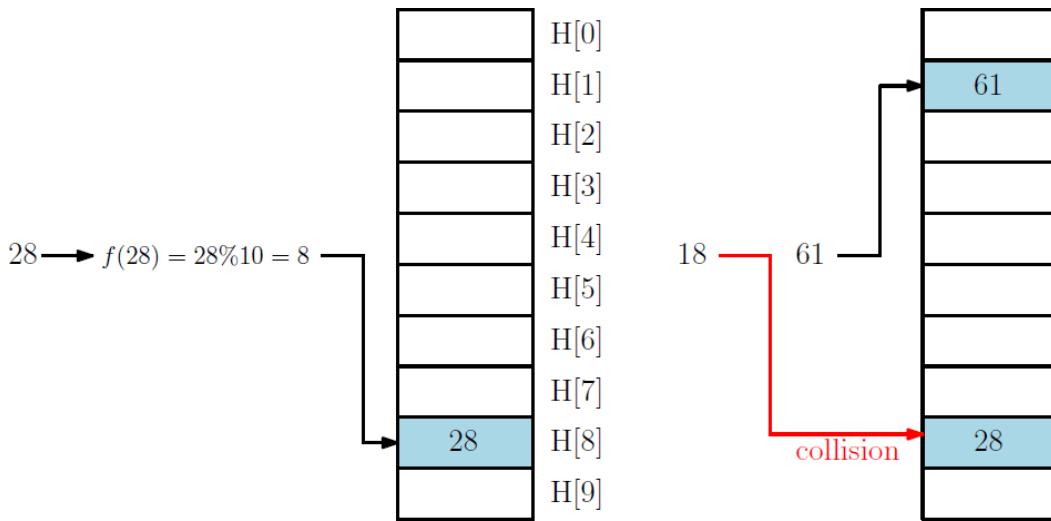
- l. 배열에 같은 연산을 적용한 경우의 시간복잡도 ( $n$ 개의 값이 저장되었다고 가정)

|                         |          |                           |               |
|-------------------------|----------|---------------------------|---------------|
| <b>moveAfter/Before</b> | 0( $n$ ) | <b>insertAfter/Before</b> | 0( $n$ )      |
| <b>pushFront/Back</b>   | 0(1)     | <b>popFront/Back</b>      | 0( $n$ )/0(1) |
| <b>remove</b>           | 0( $n$ ) | <b>search</b>             | 0( $n$ )      |

- m. [?] 양방향 연결 리스트와 배열의 장단점은 무엇인가?
- i. **장점**: 연산의 대상이 되는 노드가 결정된 후에는 대부분의 연산이 0(1) 시간에 수행된다. 위의 표 참조. (특정 값을 찾는 탐색 연산은 당연히 모든 노드를 검사할 수도 있으므로 0( $n$ ) 시간이 필요)
  - ii. **단점**: 두 링크 (prev, next)를 관리해야 하는 부담과 배열보다 많은 메모리 사용
- n. [💪] [해보기-easy-medium] 원형이중연결리스트를 이용해 Josephus 문제 구현하기
- i. Josephus 문제에 대해선 Queue 부분의 설명을 참고
  - ii. 입력으로 n과 k를 받아 한 명씩 탈락시킨 후 최종 생존자 번호를 출력함
    - 1번부터 n번까지의 key 값을 갖는 노드를 pushBack 함수를 써서 리스트 구성
    - k개의 링크를 따라간 노드를 remove하는 과정을 한 노드만 남을 때까지 반복한 후 남은 번호를 출력함

## 4. Hash Table

1. 해시 테이블은 일종의 사전(dictionary)이고, Python의 dict 자료구조처럼 다음 연산을 빠르게 지원한다 (가정: 데이터 아이템(item)은 key 값과 value 값의 쌍으로 구성된다고 가정)
  - a. `insert(key, value)` # (key, value)를 해시 테이블에 저장  
# 같은 key가 있다면 value 값을 업데이트(overwrite)함
  - b. `remove(key)` # (key, value)가 해시 테이블에 있다면 제거(delete)
  - c. `search(key)` # (key, value)를 찾아 리턴, 없다면 없다고 알려줌
2. 사실, 파이썬의 dict 자료구조도 실제 해시 테이블로 구현되어 있다
3. 해시 테이블이 사용되는 분야는 매우 넓어 실용성이 높은 자료구조 중 하나
  - a. database, compilers & interpreters, document similarity, network router, substring search, file/directory synchronization, cryptography 등의 분야
4. 해시 테이블은 보통 정보를 담아 저장할 수 있는 서랍장(테이블) 형태로 구현한다
  - a. 예를 들어, 정보 A는 3번째 서랍에 저장하기로 정하고, B는 0번째에, C는 다시 3번째에, D는 4번째에 저장하는 식이다. 예를 들어, 양말과 장갑은 두 번째 서랍에, 수건과 마스크는 네 번째 서랍에 넣는 식이다
  - b. 만약 수건을 찾고 싶다면, 수건이 저장된 서랍 번호를 먼저 (계산하여) 알아낸 후, 네 번째 서랍에 들어 있는 아이템들을 하나씩 비교해 원하는 수건을 찾아내면 된다
  - c. 가장 핵심적인 내용 중 하나는 주어진 정보를 몇 번째 서랍에 넣을지를 결정하는 것이다.
5. 정보 K(key 값)가 저장될 서랍장(슬롯, slot) 번호를 계산하는 함수 **f()**를 해시 함수(hash function)이라 한다.
  - a. 예를 들어, 해시 테이블 H를 일차원 배열 또는 리스트 H[10]으로 선언해 사용한다고 하자
  - b. 해시 함수는  $f(K) = K \% 10$ 로 정의해보자 (%는 나머지 연산임)
  - c.  $K = 28$ 을 저장하고 싶다면,  $H[f(28)]$  슬롯에 저장된다. 즉  $H[8]$ 에 저장된다
  - d.  $K = 61$ 은  $H[f(61)] = H[1]$ 에 저장된다
  - e.  $K = 18$ 도  $H[f(18)] = H[8]$ 에 저장되어야 한다. 그런데 이미  $H[8]$ 에는 28이 저장되어 있다. 이 경우를 **충돌(collision)**이 발생했다고 한다
  - f. 충돌이 발생한 경우에, 18을 저장할 공간이 더 있으면 저장하면 되지만, 지금 예처럼  $H[8]$ 에 값 하나만 저장할 수 있는 경우엔 18을 다른 곳에 저장해야 한다. 다른 곳을 정하는 방법을 **충돌해결방법(collision resolution method)**이라 부른다
  - g. 아래 그림을 참고하자



## 6. 해시 함수 (hash function)

- a. [?] key 값이 정수가 아니라면? 실수이거나 문자열이라면?
  - i. (실수, 문자열) key 값을 정수에 대응시키는 prehash 함수를 먼저 사용해 변환한다
  - ii. 파이썬의 `hash(x)` 함수는 `x`를 정수로 매핑하는 prehash 함수임
    - `__hash__` 특수 메소드로 지정해서 원하는 prehash 함수를 정의해 사용 가능
- b. e, f, g번을 꼼꼼히 읽어보고, c번과 d번은 심화 내용으로 건너뛰어도 됨
  - i. 앞으로 `mod`라 함은 `modulo` 연산을 의미하고, `%` 연산자와 같다
- c. 완전(perfect) 해시 함수: 충돌없이 1-to-1 매핑하는 해시 함수
  - i. 예를 들어, 100개의 슬롯을 갖고 있는 `H`에 50개의 값을 저장한다고 하면,  
 $100^{50}$  개 =  $10^{100}$  개의 함수 중에서  ${}_{100}C_{50} = 10^{94}$  완전 해시 함수가 존재함  
 따라서 임의의 함수가 완전 해시 함수가 될 확률은  $10^{-6}$ 으로 매우 작다.
  - ii. 문제는 입력에 대한 제한이 없기에 어떤 입력에 대해서도 충돌이 없는 완전 해시 함수를 찾는 것은 이론적으로 가능하지 않다
  - iii. 대신 삽입과 삭제가 없는 오직 탐색(search) 연산만 제공하는 환경에서는 완전 해시 함수를 알 수 있다. 해시 테이블의 슬롯에 해시된 값들을 다른 해시 테이블에 저장하는 방식의 2-레벨 해시 방법이다 (자세한 내용은 설명하고 잘 정리된 타대학 알고리즘 강의노트를 참조하기 바란다)
- d. 완전 랜덤(totally random) 해시 함수: 임의의 key 값  $x$ 가 특정 슬롯  $t$ 에 매핑될 확률이 (다른 key 값에 독립적으로)  $1/\text{size}(H)$ 인 해시 함수로 정의된다. 즉,  $\text{prob}(f(x) = t) = 1/m$ 이다. 완전 랜덤 해시 함수는 해시 테이블 자료구조 특성상 비현실적이다. 이유는 특정 key 값에 대한 연산이 여러 번 반복될 수 있고, 완전히

독립적으로 랜덤한 함수 값을 보장해야 하기에 같은 key 값에 대한 해시 함수 값이 다를 수 있기 때문이다

- e. **c-universal** 해시 함수 집합: 해시 함수 집합에서 랜덤하게 해시 함수  $f$ 를 선택해, 서로 다른 임의의 두 key 값  $x, y$ 에 대해  $\Pr(f(x) == f(y)) \leq c/\text{size}(H)$ 이 성립한다면, 해당 해시 함수를  $c$ -universal 해시 집합(family)이라 부른다.

즉, 서로 다른 두 key 값의 해시 함수 값이 같은 확률(충돌이 발생할 확률)이 해시 테이블의 크기에 반비례하는 해시 함수이다

- i. 여기서  $c$ 는 1보다 크거나 같은 실수 상수이고,  $m = \text{size}(H)$ 로  $H$ 의 슬롯 개수로 정의한다. 즉, 해시 테이블의 크기이다. 이 정의에서 중요한 것은 서로 다른 모든 두 key 값  $x, y$ 에 대해  $\Pr(f(x) == f(y)) \leq c/m$ 이 성립해야 한다는 점이다
- ii. 비교적 골고루 매핑하고 완전해시 함수보다 계산하기가 쉬워 실제 응용에서 많이 사용되는 해시 함수이다
- iii. 예1: 해시 함수 집합  $F(p, m) = \{f_{a,b} \mid a \in \{1, \dots, p-1\}, b \in \{0, 1, \dots, p-1\}\}$ 으로 정의하고,  $f_{a,b}(x) = ((ax+b) \bmod p) \bmod m$ 으로 정의한다. 그러면 이 집합에는 총  $p(p-1)$ 개의 함수가 포함된다

해시 함수 집합에서 랜덤으로 함수 하나를 선택한다는 것은 랜덤하게  $a$ 와  $b$ 를 선택한다는 것과 같다

$F(p, m)$ 에서 임의의 함수  $f$ 를 선택하면, 서로 다른 key 값  $x$ 와  $y$ 에 대해 다음의 부등식이 성립함을 보일 수 있다

$$\Pr(f_{a,b}(x) == f_{a,b}(y)) \leq 1/m$$

이 해시 함수는 **1-universal** 해시 함수임을 증명할 수 있다 (증명은 생략)

- iv. 예2:  $m = |H|$ 은 해시 테이블의 크기이고 소수라 가정하고, key 값은  $\{0, 1, \dots, m^k-1\}$ 의 (정수) 값이라고 가정한다
  - key 값  $x$ 를  $m$ -진수로 표현할 수 있다. 그러면  $k$  자리수의 수가 되고  $x = (x_1, x_2, \dots, x_k)$ 로 표현할 수 있다
  - $\{0, 1, \dots, m^k-1\}$ 에서 랜덤하게 정수  $a = (a_1, a_2, \dots, a_k)$ 를 선택한다
  - 그러면, 해시 함수  $f_a(x)$ 는 다음과 같이 정의된다

$$f_a(x) = (a \cdot x \bmod p) \bmod m = ((a_1x_1 + \dots + a_kx_k) \bmod p) \bmod m$$

여기서,  $p$  값은  $m$ 보다 더 큰 적당히 큰 소수라 가정한다. 그러면 이 함수는 **1-universal** 해시 함수임을 증명할 수 있다 (증명은 생략)

f. 현실에서 자주 쓰이는 해시 함수들 (1)

- i. Division:  $f(k) = (k \bmod p) \bmod m$  ( $p$ : 소수)
  - 매우 간단한 해시 함수로 key 값들의 성질이 잘 알려져 있지 않은 경우에 유용
- ii. Multiplication:  $f(k) = ((ak) \bmod 2^w) \gg (w-r)$ 
  - $a$ : 랜덤 값,  $w = \log k$ ,  $r = \log m$
- iii. Folding: key 값의 자리값들을 나눠서 연산하는 형식
  - **shift folding**: 예  
 $k = 1254-387-601 \rightarrow$  두 digit씩 나눠 모두 더한 후  $\bmod m \rightarrow (12 + 54 + 38 + 76 + 01) \bmod m$
  - **boundary folding**: 예  
 여러 digit로 나눈 후, 더하는 데, 짹수번 조각은 거꾸로 해서 더함, 예:  
 $(12 + \underline{45} + 38 + \underline{67} + 01) \bmod m$
- iv. Mid-Square: key 값을 적당히 연산한 후, 그 결과의 중간 부분을 떼어내 함수 값으로 리턴
  - 예:  $m = 1000$ 이라면,  $k = 3121$ 이라면,  $3121^2 = 9740641$ 이 되고 중간에 세 개의 digit를 떼어낸 406이 리턴됨.
- v. Extraction: key 값의 각 파트마다 임의의 digit을 떼어내 연결해 계산
  - 예: 계좌번호가 1254-387-601라면, 1254에서 12, 601에서 1을 떼어낸 후 서로 붙여 121을 만듬. 121이 주소가 됨

g. 현실에서 자주 쓰이는 해시 함수들 (2): key 값이 문자열(str)일 때

- i. Additive hash:  $key[i]$ 의 단순 합

```
def additive_hash(key, p, m): # string key, prime p
 h = initial_value
 for i in range(len(key)):
 h += key[i]
 return h % p % m
```

- ii. Rotating hash:  $<<$ ,  $>>$  (비트 쉬프트) 연산과  $\wedge$  (exclusive or) 연산을 반복

```
def rotating_hash(key, p, m):
 h = initial_value
 for i in range(len(key)):
 h = (h << 4) ^ (h >> 28) ^ key[i]
 return h % p % m
```

- iii. **Universal hash**: C++의 STL이나 Java에서 실제 사용되는 해시 함수로 c-universal 해시 함수이다

- Bernstein hash: `initial_value = 5381, a = 33`
- STLPort 4.6.2 hash: `initial_value = 0, a = 5`
- java.lang.String.hashCode(): `initial_value = 0, a = 31`

```
def U-hash(key, a, p, m):
 h = initial_value
 for i in range(len(key)):
 h = ((h*a) + key[i]) % p
 return h % m
```

#### h. 좋은 해시 함수란?

- i. 되도록 빠르게 계산되어야 한다 (함수 계산이 느리면 배보다 배꼽이 더 커짐)
- ii. 충돌이 되도록 적어야 한다 (충돌이 없을 수는 없지만, 가능하면 적어야 함)
  - 보통 c-universal 해시 함수를 사용한다. U-hash 함수가 대표적인 예

## 7. 충돌 해결 방법(collision resolution methods)

- a. 서로 다른 key 값  $x$ ,  $y$ 에 대해,  $f(x) = f(y)$ 가 된다면 두 key 값은 충돌이 발생했다고 정의한다
- b. 이 경우엔 두 값을 해쉬 테이블에 저장할 수 있는 방법 - 충돌해결방법이 필요하다
- c. **Open addressing**과 **Chaining** 두 가지 방법이 일반적이다
  - i. Open addressing은 충돌이 발생한 key 값을 다른 빈 슬롯을 찾아 그 슬롯에 저장하는 방법을 말한다. 빈 슬롯을 찾는 규칙에 따라 linear probing, quadratic probing, double hashing 등의 방법으로 더 세분화된다
  - ii. Chaining은 충돌이 발생하면 해당 슬롯에 연결된 연결 리스트에 pushFront 연산을 통해 삽입하는 식으로 해결한다. 즉, 해시 테이블의 슬롯마다 하나의 연결 리스트가 달려 있기 때문에 다른 빈 슬롯을 찾을 필요가 없다
  - iii. Open addressing은 슬롯에 하나의 key 값만 저장되는 방식이고 Chaining은 슬롯에 여러 개의 key 값이 연결 리스트의 노드에 차례로 저장되어 연결되어 있는 방식이다
  - iv. Open addressing의 linear probing 방법이 가장 단순하고 이해하기 쉽다. 이 방법을 자세히 설명하고 나머지 방법은 간단히 정리한다
- d. **Open addressing: linear probing**
  - i. 해시 테이블  $H$ 의 slot에 값 하나만 저장할 수 있다고 가정하자
  - ii. 아래 그림에서는 key 값 A5, A2, A3가 저장되고, 다음으로 B5, A9, B2, B9가 입력된다 (각 값이 저장되는 슬롯의 번호는 알파벳 다음의 숫자라고 가정하자)
  - iii. A2, A3, A5가 저장된 후, B5가 저장될 차례라고 하자. B5는 5번째 슬롯에 저장되어야 하는데, 이미 A5가 저장되어 있다. 결국 다른 곳에 저장해야 한다
  - iv. **linear probing 방법**은 아래쪽으로 슬롯을 차례로 탐색하면서 가장 먼저 발견된 빈 슬롯에 저장하는 것이다. 선형적으로 다음 슬롯을 연속해서 검사하는 방법이기에 linear probing이라는 명칭이 붙었다
  - v. 이에 따라 B5는 H[6]에 저장된다
  - vi. 다음의 B2에 대해서는 H[2]에 저장되어야 하나 이미 다른 값이 있으니 H[3]가 비었는지 점검한다. 다른 값이 있으니 H[4]가 비었는지 본다. H[4]가 비어 있으니 여기에 저장된다
  - vii. B9에 대해선 H[9]가 선점되어 있으니 다음 슬롯을 점검한다. H[9]이 마지막 슬롯이므로 다음 슬롯은 한바퀴 돌아서 H[0]가 된다. 따라서 B9는 H[0]에 저장된다

| $A5, A2, A3$ | $B5, A9$ | $B2$ | $B9$ | $C2$ |
|--------------|----------|------|------|------|
| 0            |          |      |      |      |
| 1            |          |      |      |      |
| 2            | $A2$     | $A2$ | $A2$ | $A2$ |
| 3            | $A3$     | $A3$ | $A3$ | $A3$ |
| 4            |          |      | $B2$ | $B2$ |
| 5            | $A5$     | $A5$ | $A5$ | $A5$ |
| 6            |          | $B5$ | $B5$ | $B5$ |
| 7            |          |      |      |      |
| 8            |          |      |      |      |
| 9            | $A9$     |      | $A9$ | $A9$ |

viii. [?]  $C2$ 가 입력되었다.  $C2$ 는 어느 슬롯에 저장되나? 답:  $H[7]$

ix.  $m = |H| =$  해시 테이블의 슬롯의 개수

#### x. 삽입 연산: `set(key, value)`

- 해시 테이블  $H$ 의 각 슬롯에는 하나의 아이템(item)을 저장한다.
- 아이템은  $(key, value)$  쌍으로 정의된다.
- $key$ 는 아이템들끼리 구분해야 하므로 아이템마다 서로 달라야 한다.
- $value$ 는 해당 아이템의 다양한 정보를 나타낸다

##### `set(key, value):`

- $key$  값을 갖는 아이템이 이미 테이블에 있다면, 해당 아이템의  $value$ 를 매개변수  $value$  값으로 수정하고, 없다면 새 아이템  $(key, value)$ 를 삽입하는 연산이다. 예를 들어,  $key$  값은 학번이고  $value$  값은 해당 학생의 이름, 학과, 전화번호 등의 개인 정보라고 한다면, 전화번호가 변경되는 경우에는 **set** 함수를 이용해 업데이트 해야 한다
- 정상적으로 수정 또는 삽입이 이루어졌다면,  $key$  값을 그대로 리턴하고, 테이블에 빈 슬롯이 없어 삽입을 하지 못했다면 **FULL**을 나타내는 특별한 값을 리턴
- 이를 위해, linear probing 방법에 따라  $key$  값을 갖는 아이템을 찾거나 빈 슬롯을 찾아  $H$ 의 인덱스를 리턴하는 `find_slot(key)` 함수 필요

##### `find_slot(key):`

- $key$  값을 갖는 아이템을 찾아 슬롯 번호(index)를 리턴하거나 **그런 아이템이 없다면** 아이템이 **삽입될 슬롯 번호**를 리턴한다
- 만약  $key$  값을 갖는 슬롯이 존재하지도 않고 빈 슬롯도 없다면 **FULL**을 리턴한다

Pseudo 코드: ([https://en.wikipedia.org/wiki/Open\\_addressing](https://en.wikipedia.org/wiki/Open_addressing))

```

def set(key, value=None):
 i = find_slot(key)
 if i == FULL: # 빈 슬롯이 없는 경우의 대책은?
 do something? # 더 큰 테이블이 필요하다!
 return None

 if H[i] is occupied: # key값이 존재하면 기존 값 수정
 H[i].value = value # value update 후 리턴
 else: # H[i]가 비어있는 경우, 즉 key가 없다면 새로 저장
 H[i].key = key
 H[i].value = value
 return i # 삽입된 슬롯 인덱스 리턴

def find_slot(key):
 i = f(key)
 start = i
 while (H[i] is occupied) and (H[i].key != key):
 i = (i + 1) % m # linear probing
 if i == start: # 한 바퀴 후에도 빈 슬롯 없음 → FULL
 return FULL
 return i

```

#### xi. 삭제 연산: remove(key)

- key 값을 갖는 아이템을 find\_slot을 이용해 찾는다. i = find\_slot(key)라 하자
- H[i]가 비었다면 삭제할 아이템이 실제로 존재하지 않는 경우이므로 처리할 내용이 없으므로 단순히 None 리턴
- H[i]가 존재한다면, 이 아이템을 지워야 한다. 문제는 이 아이템 때문에 아래쪽으로 밀려서 저장된 아이템들을 연쇄적으로 위로 옮겨 이동해야 한다. 왜, 그렇게 해야 할까? H[i]를 지우고 그대로 빈 칸으로 만들기만 하면, 나중에 H[i]에 밀려 아래쪽에 저장된 key 값을 탐색할 때, 빈 칸을 만나 탐색을 중단하게 되고 그런 key 값은 없다고 결정하게 된다
- 연쇄적인 이동을 완료한 후에는 성공적인 삭제가 수행되었다는 의미에서 key 값 자체를 리턴한다
  - a. H[i]는 현재 빈 슬롯이고, 아래쪽 H[j]에 있는 아이템을 H[i]로 이동할지를 결정해야 한다
  - b. H[j].key 값의 해시 함수 값을 k라 하자. 즉, k = f(H[j].key)이다. 이 k 값이 (i, j]에 있다면 (즉, ..i..k..j.. 순서라면) H[j]를 H[i]로 옮기면 안된다. 왜 안될까?
    - i. 옮긴다고 해보자. 그러면 H[j]가 H[i]로 이동하는 것이다. 나중에 H[j].key를 탐색하는 경우에는 H[k]부터 아래쪽으로 탐색을 시작한다. 그런데

$H[k]$ 보다 더 위에 있는  $i$  번째 슬롯에 실제 값이 있기 때문에 경우에 따라서는 해당 값이 없다고 결론 내릴 수도 있다

- c. 또한 해시 테이블이 원형 리스트와 같기 때문에  $i > j$  일 수도 있으므로,  $\dots j \dots i \dots k \dots$  순서라거나,  $\dots k \dots j \dots i \dots$ 인 경우에도 같은 이유로 옮기면 안된다
- d. 위의 경우가 아니라면  $H[j]$ 를  $H[i]$ 로 옮긴다. 그러면 이제  $H[j]$ 가 빈 슬롯이 되고, 같은 일을 반복하면 된다

### Pseudo 코드:

```
def remove(key):
 i = find_slot(key)
 if H[i] is unoccupied: # key가 없는 경우
 return None
 j = i
 while True:
 | mark H[i] as unoccupied
 | while True
 | | j = (j+1) % m
 | | if H[j] is unoccupied: # 이동완료
 | | return key
 | | k = f(H[j].key)
 | | # |...i..k..j...
 | | # |...j...i..k..| or |..k..j..i..| 경우 이동 안함
 | | if not (i < k <= j or j < i < k or k <= j < i):
 | | break # [?] 이동 결정! 그래서 break
 |
 | H[i] = H[j] # H[j]를 H[i]로 이동
 i = j
```

- 연쇄적인 이동을 하지 않고 제거하는 방법은 없을까?

- a.  $H[i]$ 를 지워야 한다고 하면, 이 슬롯을 unoccupied로 표시하지 않고, 다른 표시를 해두면 어떨까?
- b. 예를 들어,  $H[i].key = DUMMY$  처럼 일종의 DUMMY 객체를 만들어 '삭제했음'을 표시해 보자
- c. 이렇게 하면 `find_slot` 연산에서  $H[i].key$  값이 DUMMY이더라도 탐색을 (멈추지 않고) 계속해야 한다.

`find_slot`은 원하는 key 값을 찾거나 빈 (unoccupied) 슬롯을 찾을 때까지 계속하면 된다

- d. 이런 방법을 사용하면, `remove` 연산은 상수시간에 가능하지만, `find_slot`의 시간은 DUMMY로 표시된 슬롯도 계속 따라가야 하므로 더 오래 걸린다. (공짜 점심은 없다!)
- e. DUMMY는 다음과 같은 클래스의 객체로 처리 가능

```
class DummyValueClass():
 pass
DUMMY = DummyValueClass()
```

- f. 참조  
[https://just-taking-a-ride.com/inside\\_python\\_dict/chapter2.html](https://just-taking-a-ride.com/inside_python_dict/chapter2.html)

### xii. 탐색 연산: `search(key)`

- key 값을 갖는 아이템을 찾아 value 값을 리턴하고, 없다면 `None`을 리턴함

```
def search(key):
 i = find_slot(key)
 if H[i] is occupied: # key is in table
 return H[i].value
 else: # key is not in table
 return None # not found
```

- e. 해시 테이블의 클래스 `HashOpenAddr`를 구현해보자

- i. `size` = 해시테이블의 슬롯 개수
- ii. `keys` = 슬롯의 키(key)를 저장하는 리스트 (`None`으로 초기화)
- iii. `values` = 슬롯의 값(value)을 저장하는 리스트 (`optiona`, `None`으로 초기화)
- iv. 위의 `set`, `find_slot`, `remove`, `search`는 클래스 구조에 맞게 수정

- `H[i].key` → `self.keys[i]`, `H[i].value` → `self.values[i]`
- `if H[i] is unoccupied:` → `if self.keys[i] == None:`

```
class HashOpenAddr:
 def __init__(self, size=10):
 self.size = size # prime number
 self.keys = [None]*self.size
 # None → "unoccupied"
 self.values = [None]*self.size

 def find_slot(self, key):
 ...
```

```

def set(self, key, value):
 ...
def remove(self, key):
 ...
def search(self.key):
 ...
def hash_function(self, key): # f(key)
 ...
def __getitem__(self, key): # hashTable[key]의 형식으로
 return self.search(key) # value를 얻을 수 있도록 함

def __setitem__(self, key, value): # H[key] = value 가능
 self.set(key, value)

```

v. 예:

```

H = hashOpenAddr()
H[75] = "cat" # H.__setitem__(75, "cat") ⇒ H.set(75, "cat")
H[21] = "dog"
H[7] = "sparrow"
print(H[21]) # H.__getitem__(21) ⇒ H.search(21)
print(H[7]) # None이 출력

```

f. [고급] linear probing의 set, remove, search 수행시간은?

i. 수행시간 분석을 쉽게 하기 위해, 두 가지 가정을 더 한다:

- (가정 1) 임의의 key 값이 특정 슬롯으로 해시될 확률이 모두  $1/m$ 으로 같다고 가정한다 → **완전 랜덤 해시 해시 함수** (totally random hash function)를 사용한다는 의미이다. 현실성이 없는 해시 함수이지만 이론적 분석을 위해 사용한다
- (가정 2) 항상  $m \geq 2n$ 이라고 가정한다. 즉, 항상 빈 슬롯이 50% 이상 존재한다고 가정한다. (만약  $m < 2n$ 이라면 doubling을 수행해 2배 큰 해시 테이블을 새로 만들어 기존 테이블의 값을 이동시켜 항상  $m \geq 2n$ 을 유지할 수 있다) → load factor  $n/m \leq 1/2$ 이라는 의미임 (\* 테이블 크기 조정은 후반부에 자세히 설명한다 \*)

완전 랜덤 해시 함수는 현실성이 없기 때문에 **가정 1**은 오로지 분석을 위한 가정이다. 5-wise independent 해시 함수를 사용하면 비슷한 분석이 그대로 성립하는데, 이 주제는 교재 범위를 벗어나므로 여기서는 다루지 않는다

ii. search 함수의 성능이 다른 두 함수 set과 remove의 성능을 결정하기 때문에, search(key)의 수행 시간만을 분석한다. search 함수의 수행 시간은 결국

`find_slot` 함수의 수행 시간과 같다. 따라서 `find_slot` 함수의 수행시간을 분석한다

- iii. `find_slot`의 수행 시간을 좌우하는 건 key 값이 속한 클러스터(**cluster**)의 크기에 좌우된다

- 클러스터란? 아래 그림에서처럼 해시 테이블에 아이템들이 계속 삽입되면서 국지적으로 모여있게 된다. 처음엔 (A2, A3), (A5) 두 클러스터, 두 번째 테이블 그림에서는 (A2, A3), (A5, B5), (A9) 세 클러스터가 만들어진다

|   | A5, A2, A3 | B5, A9 | B2 | B9 | C2 |
|---|------------|--------|----|----|----|
| 0 |            |        |    |    | 0  |
| 1 |            |        |    |    | 1  |
| 2 | A2         | A2     | A2 | A2 | A2 |
| 3 | A3         | A3     | A3 | A3 | A3 |
| 4 |            |        | B2 | B2 | B2 |
| 5 | A5         | A5     | A5 | A5 | A5 |
| 6 |            | B5     | B5 | B5 | B5 |
| 7 |            |        |    |    | 7  |
| 8 |            |        |    |    | 8  |
| 9 |            | A9     | A9 | A9 | A9 |

- iv. 특정 슬롯  $i$ 에서 시작하는 길이  $k$ 인 cluster는 아래  $k$ 개의 슬롯으로 정의된다:

$$H[i], H[i+1], \dots, H[i+k-1]$$

- 이 cluster가 발생할 확률  $p$ 를 계산해보자!
- 이 cluster에 속한 key 값들은  $i, i+1, \dots, i+k-1$  슬롯으로 해시되어야 한다. 특정 슬롯으로 해시 될 확률은 가정 1에 의해 모두  $1/m$ 로 동일하므로, 동전을 던져서 해시 값이  $[i, i+k-1]$  구간에 떨어지면 성공, 아니면 실패로 해석하면 된다. 이 말은 총  $n$ 번의 동전을 던지는 데 ( $n$ 개의 값이 해시 테이블에 저장되는 데), 그 중  $k$ 번이 성공하면 길이가  $k$ 인 클러스터가 만들어진다는 뜻이다. 이는 성공 확률이  $k/m$ 인 이항분포 (binomial distribution)가 됨을 알 수 있다. 따라서, 확률  $p$ 는 다음과 같다:

$$p = \binom{n}{k} \left(\frac{k}{m}\right)^k \left(\frac{m-k}{m}\right)^{n-k}$$

- 최악의 경우를 분석해야 하므로, 위의 확률이 최대가 되는 경우를 분석해보자.  $m = 2n$ 인 경우가 가장 최대가 됨을 계산을 통해 알 수 있다. (그렇다고 우선 믿자 ^^) 따라서, 아래의 식 전개가 성립한다

$$\begin{aligned}
p &= \binom{n}{k} \left(\frac{k}{m}\right)^k \left(\frac{m-k}{m}\right)^{n-k} \\
&\leq \binom{n}{k} \left(\frac{k}{2n}\right)^k \left(\frac{2n-k}{2n}\right)^{n-k} \\
&= \left(\frac{n!}{(n-k)!k!}\right) \left(\frac{k}{2n}\right)^k \left(\frac{2n-k}{2n}\right)^{n-k} \\
&\approx \left(\frac{n^n}{(n-k)^{n-k} k^k}\right) \left(\frac{k}{2n}\right)^k \left(\frac{2n-k}{2n}\right)^{n-k} \\
&= \left(\frac{n^k n^{n-k}}{k^k (n-k)^{n-k}}\right) \left(\frac{k}{2n}\right)^k \left(\frac{2n-k}{2n}\right)^{n-k} \\
&= \left(\frac{nk}{2nk}\right)^k \left(\frac{n(2n-k)}{2n(n-k)}\right)^{n-k} \\
&= \left(\frac{1}{2}\right)^k \left(\frac{(2n-k)}{2(n-k)}\right)^{n-k} \\
&= \left(\frac{1}{2}\right)^k \left(1 + \frac{k}{2(n-k)}\right)^{n-k} \\
&\leq \left(\frac{1}{2}\right)^k e^{k/2} = \left(\frac{\sqrt{e}}{2}\right)^k \\
&= O(c^k).
\end{aligned}$$

- 위의 식에서,  $r!$ 는 대략  $(r/e)^r$ 으로 근사되고,  $c < 0.8243$ 인 상수임이 알려져 있다
- 결국  $k$ 가 커질수록 길이가  $k$ 인 cluster가 발생할 확률이 지수적으로 매우 작아진다는 뜻이다. (매우 중요한 사실)

#### v. 이제, **find\_slot(key)** 함수의 평균 수행시간을 계산해보자

- 만약,  $i = f(key)$ 라면,  $i$ 를 포함하는 cluster를 탐색하게 되고, 최악의 경우 cluster의 길이  $k$ 만큼의 시간이 필요하다
  - 해시 함수를  $O(1)$  시간에 계산하고 cluster를  $O(k)$  시간에 탐색하기 때문에 필요한 시간은  $O(1 + k)$ 이다.
- 평균이므로 모든  $i = 1, \dots, m$ 에 대해, 모든 길이  $k = 0, \dots, m$ 에 대해 평균을 구한다 (여기서 run은 cluster를 의미함)

$$\begin{aligned}
& O\left(1 + \left(\frac{1}{m}\right) \sum_{i=1}^m \sum_{k=0}^{\infty} k \Pr(i \text{ is a part of a run of length } k)\right) \\
& \leq O\left(1 + \left(\frac{1}{m}\right) \sum_{i=1}^m \sum_{k=0}^{\infty} k^2 \Pr(\text{run of length } k \text{ starts at } i)\right) \\
& = O\left(1 + \left(\frac{1}{m}\right) \sum_{i=1}^m \sum_{k=0}^{\infty} k^2 p\right) \\
& = O\left(1 + \sum_{k=0}^{\infty} k^2 p\right) = O\left(1 + \sum_{k=0}^{\infty} k^2 c^k\right) \\
& = O(1)
\end{aligned}$$

- a. 위의  $k^2c^k$ 는 충분히 큰 상수  $k$ 에 대해선 1보다 작게 되어 무한 급수가 상수로 수렴한다

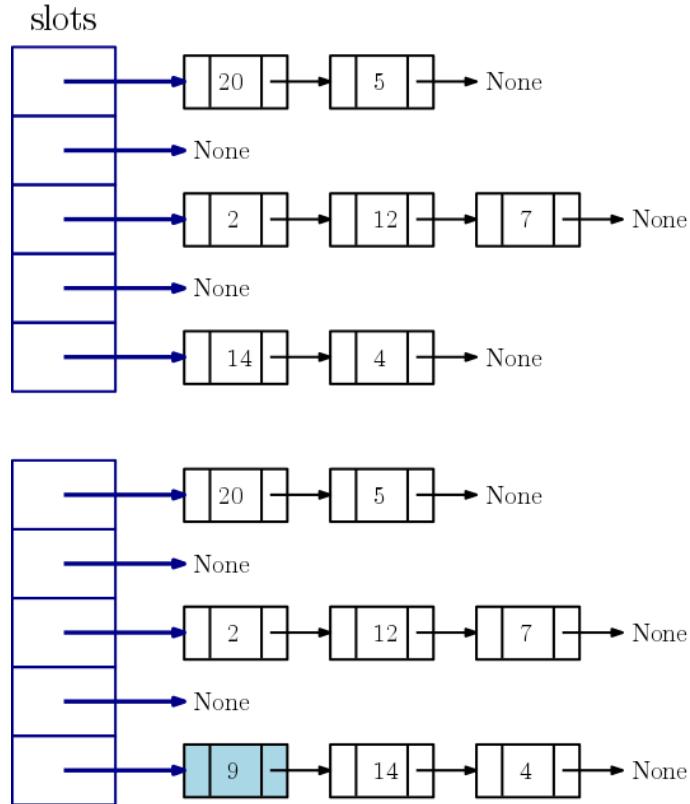
- 결국 `find_slot`, `search`, `set`, `remove` 모두 **O(1)** 시간에 수행된다
- $m \geq 2n$  조건처럼 해시 테이블의 일정 부분이 항상 비어 있도록 관리된다면, 해시 테이블의 탐색, 삽입, 삭제 연산은 (평균적으로) 상수시간에 수행이 가능하게 된다. 매우 효율적인 자료구조로 실제로 광범위한 분야에서 사용되고 있다
- 실험에 의하면,  $m \geq 1.25n$  (즉, 25% 이상 빈 슬롯을 유지하는 게) 기준이 가장 현실적이고 효율적인 기준이라고 밝혀졌다
- Python의 `dict`와 `set`을 위한 해시 테이블에서는  $m \geq 3n/2$  기준을 사용한다

#### g. Quadratic hashing, double hashing: 클러스터 사이즈를 줄이는 방법

- i. **quadratic probing**은 탐색하는 슬롯의 순서가  $f(key) \rightarrow f(key) + 1^2 \rightarrow f(key) + 2^2 \rightarrow f(key) + 3^2 \rightarrow \dots \rightarrow f(key) + k^2 \rightarrow \dots$  순서로 제곱한 간격으로 빈 슬롯을 탐색하는 방법
- ii. **double hashing**은 두 개의 해시 함수를 사용해서 빈 슬롯을 탐색
  - $f(key) + g(key) \rightarrow f(key) + 2g(key) \rightarrow \dots \rightarrow f(key) + kg(key) \rightarrow \dots$  순서로 슬롯을 검색
- iii. linear probing과 이 두 방법의 구체적인 성능 비교는 후반부에 그림으로 자세히 설명

## h. Chaining

- i. H의 슬롯에 값 하나만 저장하도록 하는 게 아니라, 각 슬롯마다 연결리스트를 연결해, 슬롯 하나당 이론적으로 무한히 많은 값을 저장하도록 하는 방법



- ii. 연결리스트 중에서, 간단한 구조의 한방향 연결리스트를 활용하는 것이 일반적이다.

```
class HashChain:
 def __init__(self, m):
 self.size = m
 self.H = [SinglyLinkedList() for _ in range(m)]

 • H[i]는 hash_function(key) = i인 key 값을 노드로 연결한
 한방향 연결리스트의 head 노드를 가리킨다 (한방향 연결리스트
 클래스를 import해서 사용하면 된다)

 • find_slot은 key 값에 대한 해시 함수 값(슬롯 인덱스)을 단순 리턴

 def find_slot(self, key):
 return self.hash_function(key)

 • set(key, value): H[hash_function(key)] 리스트를 탐색하여 key
 값이 없다면 연결리스트의 pushFront 함수를 호출하여 head 노드에
 삽입하고, 있다면 value 값을 수정
```

```

def set(self, key, value=None):
 i = self.find_slot(key)
 v = self.H[i].search(key)

 if v == None: # key 값 노드 없다면 삽입연산
 self.H[i].pushFront(key, value)

 else: # 기존의 key값을 있으므로 value값 수정 연산
 v.value = value

```

- **search(key)**:  $H[hash\_function(key)]$  리스트를 탐색하여 있다면, **value** 값을 리턴하고 없다면, **None**을 리턴한다
- **remove(key)**:  $H[hash\_function(key)]$  리스트를 탐색하여 **key** 값 노드를 연결리스트의 **remove**함수를 호출하여 삭제한다

```

def remove(self, key):
 i = self.find_slot(key)
 v = self.H[i].search(key)
 return self.H[i].remove(v)

```

위 코드가 **remove**에서 다시 **v**의 전 노드를 찾기 위해 탐색을 하기에 효율적이지 않다. 해결 방법은 **removeKey(key)** 함수를 따로 만들어 **search**와 제거 과정을 통합하든지, 한방향 연결리스트 대신 양방향 연결리스트를 사용하는 것이다

- 위의 그림에서는 해시 테이블의 크기 **size = 5**인 경우, **hash\_function(key) = key % size**로 정의했을 때, **set(9, value)**를 실행한 경우의 변화를 나타낸다
- [] **\_\_setitem\_\_(self, key, value)**와 **\_\_getitem\_\_(self, key)** 함수를 각각 구현해보자
- [**고급**] Chaining 연산 수행시간: **search** 시간 분석
  - c-universal hash 함수를 가정한다. (즉, 다른 두 key 값의 해시 값이 같을 확률이  $c/m$ 이하이다.)
  - 특정 key 값을 **search**하는 데 필요한 시간은  $f(key)$  슬롯의 연결리스트 길이  $\text{len}(H[f(key)])$ 에 의해 결정된다  $\rightarrow O(1 + \text{len}(H[f(key)]))$  시간 필요!
  - $C(x, y) =$  두 key의 hash 값이 같으면 1, 아니면 0을 나타내는 랜덤변수 (random variable)
  - $\text{Prob}(C(x, y) = 1) = c/m$  (c-universal 해시 함수이므로)
  - $\text{len}(x) = (x\text{와 같은 해시 값을 갖는 } y\text{의 개수})$  이므로  

$$\text{len}(x) = \sum_y \text{Pr}(C(x, y) = 1)$$

- 정리하면 다음과 같다

$$\begin{aligned}
 E[\text{len}(x)] &= E\left[\sum_y C(x, y)\right] \\
 &= \sum_y E[C(x, y)] \\
 &= \sum_y (1 \times \Pr(C(x, y) = 1) + 0 \times \Pr(C(x, y) = 0)) \\
 &= \sum_y \frac{c}{m} = \frac{cn}{m}
 \end{aligned}$$

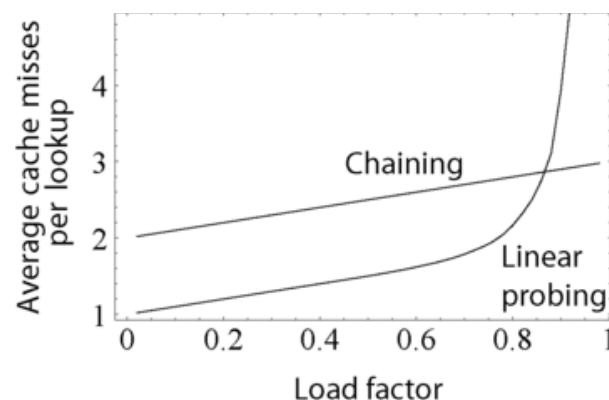
- 결국 평균  $O(1 + n/m)$  시간이 되고  $m \geq 2n$  정도로 운영하면, 평균  $O(1)$  시간에 세 가지 연산을 모두 수행가능하다! 즉, chaining 방법이나 open addressing 방법 모두  $n/m$ 이 상수로 보장되는 경우에는 평균  $O(1)$  시간을 보장한다
- 

### i. 해시 자료구조의 성능은 어떻게 평가할까?

- 앞에서 가정한대로, 해시테이블의 크기(슬롯의 갯수)를  $m$ 이라 하고, 테이블에 저장된 아이템의 개수를  $n$ 이라 하자
- load factor:**  $LF = n/m$ 
  - 테이블 크기에 비해 아이템이 많을 수록 load factor가 커진다. 그러면 충돌횟수도 비례하여 증가하게 됨
- collision ratio:**  $(\text{number of collisions})/n$ 
  - 비율이 작을 수록 연산 시간이 작아짐
- search 성능**
  - unsuccessful search는 찾고자 하는 key 값이 테이블에 없는 경우이고, 이 경우가 successful search보다 시간이 많이 걸림
  - 결국 unsuccessful search의 수행시간이 search의 수행시간이 됨 (최악의 경우 가정)
- 다음은 유명한 전산수학자 Donald Knuth(도날드 크누쓰)가 세운 유명한 식으로, open addressing을 채택했을 때, 주어진 key 값을 찾기 위해 필요한 평균 비교 횟수를 LF에 관해 정리한 것임
  - 예를 들어, linear probing인 경우, LF가 클수록 1-LF가 작아지고, 따라서 전체 값은 커지게 되어 탐색을 위한 평균 비교횟수가 커짐. (unsuccessful인 경우는  $(1-LF)^2$ 이므로 더 빠른 속도로 탐색시간이 증가함)
  - 반면에, quadratic probing 방법은  $\log$ 식에 따라 커지게 되어 linear probing보다는 탐색횟수가 훨씬 작음
  - double hashing은 두 개의 해시 함수를 사용해서 빈 슬롯을 탐색하는 방법

|                     | linear probing                                      | quadratic probing                   | double hashing                    |
|---------------------|-----------------------------------------------------|-------------------------------------|-----------------------------------|
| successful search   | $\frac{1}{2} \left( 1 + \frac{1}{1-LF} \right)$     | $1 - \ln(1 - LF) - \frac{LF}{2}$    | $\frac{1}{LF} \ln \frac{1}{1-LF}$ |
| unsuccessful search | $\frac{1}{2} \left( 1 + \frac{1}{(1-LF)^2} \right)$ | $\frac{1}{1-LF} - LF - \ln(1 - LF)$ | $\frac{1}{1-LF}$                  |

- vi. 아래 wikipedia에서 참조한 그래프는 open-addressing-linear probing 방법과 chaining 방법을 LF에 따른 탐색 시간을 비교해서 그림
- vii. LF가 0.8 정도까지는 linear probing이 더 빠르게 탐색을 하지만, 그 이후엔 탐색시간이 매우 느려짐을 알 수 있다.



## 8. 해시 테이블 크기 조정 (Resizing the hash table)

너무 많은 아이템이 해시 테이블에 저장된다면, 또는 load factor 기준을 만족하지 못한다면, 해시 테이블의 크기를 조정(resizing)해야 한다

- a. 더 큰 해시 테이블을 할당받아 ( $m \rightarrow m'$ ), 기존의 해시 테이블의 값을 모두 옮긴다

- i.  $n$ 개의 아이템을 새 테이블로 옮겨야 하므로  $O(n)$  이사 시간 필요 → 꽤 큰 비용아닌가?

### b. 이슈

- i. 언제, 얼마나 크게 할당받아야 하나?

- **방법 1:**  $n = m$ 이 될 때,  $m' = m + 1$ 로 크기 조정 → 계속 삽입이되면 크기 조정과 이사 비용이 계속 발생하게 된다. 즉,  $m = 1 \rightarrow 2 \rightarrow 3 \dots$  으로 증가하고 그 때마다 이사 비용이  $0 \rightarrow 1 \rightarrow 2 \dots$  발생하기 때문에 현재 상태까지의 총 이사 비용은  $1 + 2 + \dots + n-1 = O(n^2)$ 이 된다. 총  $n$ 번의 set 연산이 이루어졌기에 한 번의 set 연산당 평균 이사 시간은  $O(n^2)/n = O(n)$ 으로 상당히 크다
- **방법 2:**  $n \geq m/2$ 이 될 때,  $m' = 2m$ 로 테이블 크기를 2배 늘리는 게 일반적인 방법! 그러면 항상  $m \geq 2n$ 을 만족할 수 있다. 이렇게 테이블이 반이 채워지는 순간 테이블을 2배씩 키워서 이사하는 방법의 평균 이상 비용을 더 자세히 계산해보자

- ii. [?] 두 배 큰 해시 테이블로 이사하기 위해서 기존의 해시 테이블의 값을 일일이 새 해시 테이블로 set 연산을 통해 "이사"해야 한다. 이때, 새로운 해시 함수가 필요한가? Yes!

- iii. 만약,  $n$ 번의 삽입이 연속적으로 일어나면서 테이블 크기 조정이 여러 번 발생하는 최악의 상황을 생각해보자. 이를 위해 필요한 총 시간은? ( $m = 2, n = 1$ 부터 시작)

- $m = 2 \rightarrow 2^2 \rightarrow 2^3 \rightarrow \dots \rightarrow 2^k$   
 $n = 1 \rightarrow 2^1 \rightarrow 2^2 \rightarrow \dots \rightarrow 2^{k-1}$

- $k$ 번의 크기 조정이 일어난다. 그 때마다 이사 비용  $c$ 는 다음과 같다  
 $c = 1 \rightarrow 2^1 \rightarrow 2^2 \rightarrow \dots \rightarrow 2^{k-1}$

- 총 이사 비용은  $1 + 2^1 + \dots + 2^{k-1} = 2^k - 1$ 이 된다

- $n = 2^{k-1}$ 이므로 평균 비용은  $(2^k - 1)/2^{k-1} < 20$ 이 된다. 즉, 평균적으로 따지면 **한 번의 삽입 연산의 평균 비용이  $O(1)$** 이라는 의미이다!

- 방법 1의 평균 비용은  $O(n)$ 이지만, 방법 2는  $O(1)$ 로 매우 빠르다. 실제로 사용되는 대부분의 해시 테이블 resizing 전략은 방법 2와 유사하기 때문에 평균  $O(1)$  시간의 이사 비용만을 지불한다고 생각하면 된다

- iv. 이 경우처럼 한 연산의 최악의 경우의 수행 시간보다 여러 번의 연산에 대한 **평균 수행 시간을 정의하는 게 더 합리적인 경우가 있다.** 이렇게 연산의 평균

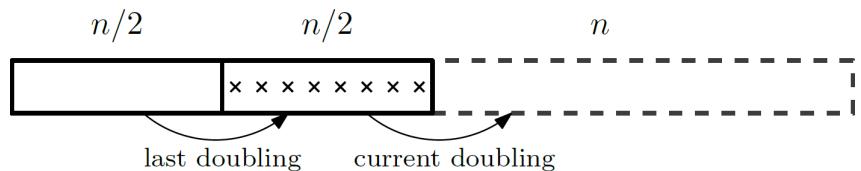
시간을 분석하는 것을 **amortized analysis**라 하고, 이렇게 분석된 평균시간을 **amortized time**이라 부른다

### 9. [고급] 더 자세한 amortized 시간 복잡도 분석 방법 중 **Charging 방법** 소개

- a. 다음과 같은 해시 테이블의 resizing 전략을 고려해보자:  $n = m$ 이 되면 **doubling**이 발생하고,  $n < m/4$ 가 되면 **halving**이 발생하는 해시 테이블 모델을 가정한다. (이 전략이 슬롯이 반 이상 채워지면 **doubling**을 하는 방법 2와는 다르다는 점을 유의하자)
  - i. **set** 연산이 계속 이루어져  $n > m$ 이 되면, 더 이상 빈 슬롯이 없게 된다. 이 경우에 보통 **doubling**이란 방법을 통해  $m \rightarrow 2m$ 으로 슬롯의 개수를 두 배 늘린 새로운  $H'$ 을 만들고 기존의  $H$ 에 있는 모든 아이템을 새 해시 함수를 이용해  $H'$ 으로 이동해야 한다
  - ii. 반대로 **remove** 연산이 훨씬 많이 발생해  $H$ 에 저장된 아이템의 개수가  $n < m/4$ 로 줄어든다면,  $m \rightarrow m/2$ 로 반으로 줄이는 **halving** 과정을 수행한다
- b. **doubling**, **halving** 모두 기존의 아이템을 새로운 테이블로 옮겨야 하기 때문에 최악의 경우에는  $O(n)$  시간이 필요한 매우 **비싸** 연산이다. 현재 **set**이 마지막 빈 슬롯에 삽입된다면 ( $n = m$ 이 되는 경우) **doubling**이 발생하여  $O(n)$ 의 시간의 이상 비용이 필요하다. 결국, 이 **set** 연산의 **최악의 경우**의 수행시간은  $O(1)$ 이 아니라  $O(n)$ 이다. **halving**이 발생하는 경우의 **remove** 연산도  $O(n)$  시간이 걸리게 된다
- c. 해시 테이블 자료구조를 사용할 때에는 빈번한 **set**과 **remove** 연산을 처리하기 때문에 매 연산마다 **doubling**/**halving**이 발생하는 게 아니다. 따라서 최악의 경우에 대한 연산 시간이 아니라 전체 연산 횟수에 대한 평균 시간을 분석하는 게 합리적이다. 이처럼 전체 연산 횟수에 대한 개별 연산의 **평균 시간**을 앞에서도 설명한 대로 **amortized 수행 시간**이라 부른다
- d. amortized 수행 시간 방법에는 aggregation, accounting, **charging**, potential function 방법 등 다양하다. 가장 직관적인 **charging** 방법을 소개한다
- e. **Charging analysis method**: "현재의 비용을 과거의 연산에 떠 넘기는 방식"
  - i. **set (insertion)** 연산
    - **현재**의 **doubling** 비용을 **직전** **doubling** 또는 **halving** 이후에 발생한 **set** 연산들에 떠 넘겨보자!
    - $n \rightarrow 2n$  이 되는 현재의 **doubling** 비용은  $n$ 개의 아이템을 새로운 테이블로 옮기는 비용이므로  $cn$  시간이 필요하다고 볼 수 있다. 이 비용을 바로 직전 **doubling** 또는 **halving** 이후에 발생한 **set** 연산에 "charging"해보자
    - **직전에 doubling이 발생했다면**, 아래 그림처럼,  $n/2$ 개의 슬롯이 2배인  $n$ 개의 슬롯으로 증가하게 된다. 이 상태에서는  $n/2$ 개의 슬롯이 비어 있기에 이 빈 슬롯을 다 채워야지 현재의 **doubling** 조건을 만족하게

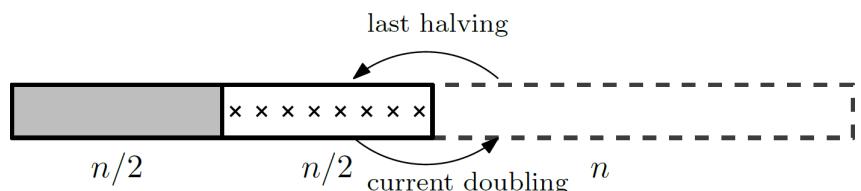
된다. 즉, 직전 doubling 이후에  $n/2$ 번의 set 연산이 있어야 현재의 doubling이 발생한다는 것이다. 현재의 doubling 비용이  $cn$ 이므로 이 비용을  $n/2$ 번의 set 연산에 골고루 나눠 이전하면 set 연산 하나당  $2c$  정도의 상수 비용을 추가로 부담하면 되는 셈이다

$$n/2 \rightarrow (\text{직전 doubling}) \rightarrow n \rightarrow (\text{현재 doubling}) \rightarrow 2n$$



- 직전에 halving이 발생했다면, 아래 그림처럼  $2n$ 개의 슬롯이 반으로 줄어  $n$ 개가 된다. 그런데 halving 발생 조건에 따르면,  $2n$ 개의 슬롯 중에서  $\frac{1}{4}$  이하인  $n/2$ 개만 사용되었기 때문에, 현재의 doubling을 위해서는 나머지  $n/2$ 개가 set 연산을 통해 채워져야 한다. 이 경우에도 역시  $n/2$ 개의 set 연산에 현재의 doubling 비용  $cn$ 을 골고루 나눠 이전하면, set 연산 하나당  $2c$ 의 상수 비용만 부담시키면 된다

$$2n \rightarrow (\text{직전 halving}) \rightarrow n \rightarrow (\text{현재 doubling}) \rightarrow 2n$$

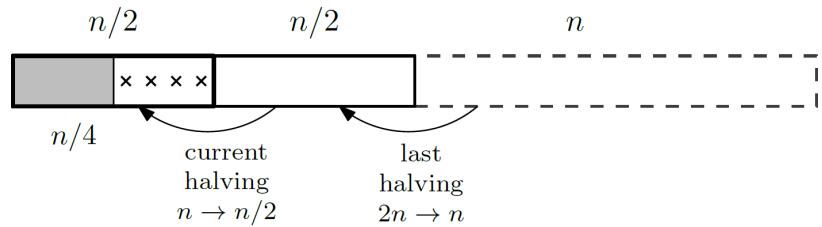


- 결국, 현재의 doubling 비용은 그 이전 발생한  $n/2$ 개의 set 연산에 중복없이  $2c = O(1)$  추가 비용을 "charging"할 수 있다는 것이다. set 연산 자체 수행 시간이  $O(1)$ 이고 resizing의 doubling에서 발생하는 추가 이전 비용 역시  $O(1)$ 이므로 set 연산의 수행 시간은 여전히 (평균)  $O(1)$ 이 된다
  - 여기서 중요한 사실은 이  $n/2$ 개의 set 연산들이 오직  $n \rightarrow 2n$  doubling 연산에 대한 비용만 추가로 부담하지 그 이전 또는 그 이후에 발생하는 doubling 연산의 추가 비용을 부담하지 않는다는 사실이다

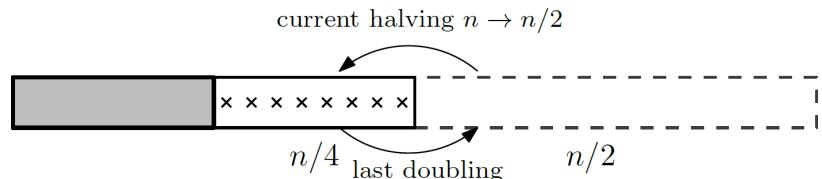
## ii. remove 연산:

- 이제 remove 연산의 amortized 시간 분석을 해보자
- halving은  $n/4$ 개 이하로 떨어졌을 때 발생하고 슬롯 개수는  $n \rightarrow n/2$ 이 된다. 바로 직전에 발생한 doubling 또는 halving 작업을 생각해보자
- 바로 직전에 halving이었다면,  $2n \rightarrow n$  halving이어야 하므로,  $n/2$ 개인 경우에 직전 halving이 발생한다. 그러면  $n/2$ 개 중에서 다시  $n/4$ 번의 remove 연산이 있어야 현재의 halving이 발생하므로  $n/4$ 번의 remove

연산에 현재 halving의 비용  $c_n$ 을 부과하면 된다. 그러면 각 remove 연산의 입장에서는  $O(1)$  정도의 추가 비용만 더 지불하면 된다



- 바로 직전에 doubling이었다면,  $n/2 \rightarrow n$  doubling이므로 역시  $n/4$ 번의 remove 연산이 있어야 현재 halving이 발생한다. 따라서  $n/4$ 개의 remove 연산에 현재 halving 비용을 전가할 수 있다



- 결과적으로  $O(n)$ 의 비용이 드는 halving 작업을 직전의  $n/4$ 번의 remove 연산에 전가할 수 있고, 이 remove 연산은 다른 (doubling이나 halving) 연산으로부터 전가받은 비용은 존재하지 않으므로 결국 평균적으로 각 remove 연산이  $O(1)$  씩의 비용을 나눠서 지불하는 셈이다
- 결론: set과 remove 연산 자체는 평균  $O(1)$ 이 수행 시간이면 충분하고, doubling/halving을 위해 필요한 이사 비용을 직전 set/remove에  $O(1)$ 씩 전가할 수 있으므로, **set과 remove 연산의 amortized 수행 시간은  $O(1)$ 이다**

## 10. 해시 테이블로 배열 자료구조 구현하기

해시 테이블 자료구조에서 제공하는 연산은 search, set, remove이고 모두 평균  $O(1)$  시간에 수행된다. 순차적 자료구조인 배열 자료구조에서는 다음과 같은 연산을 주로 제공한다. 아래 6개의 연산을 대응되는 python의 리스트 연산을 정리했다

1. `get_at(i): return A[i]` #  $O(1)$
2. `set_at(i, x): A[i] = x` #  $O(1)$
3. `insert_at(i, x): A.insert(i, x)` #  $O(n-i) = O(n)$
4. `delete_at(i): A.pop(i)` #  $O(n-i) = O(n)$
5. `push_back(x): A.append(x)` # 평균  $O(1)$
6. `push_front(x): A.insert(0, x)` #  $O(n)$
7. `pop_back(): A.pop()` # 평균  $O(1)$
8. `pop_front(): A.pop(0)` #  $O(n)$

- a. 배열 자료구조 (python의 `list` 자료구조)를 해시 테이블을 이용해 구현해보자. 목표하는 시간 복잡도는 다음과 같다

- i. `get_at, set_at`: 평균  $O(1)$  시간
- ii. `push_back, push_front, pop_back, pop_front`: 평균  $O(1)$  시간
- iii. `insert_at, delete_at`: 평균  $O(n)$  시간

Python의 리스트에서는 `push_back`은 평균  $O(1)$  시간이지만 `push_front`는 `pop(0)` 연산을 이용하기에  $O(n)$  시간이 최악의 경우에 필요하다. 해시 테이블을 이용해서 이 연산을 평균  $O(1)$  시간에 해결할 수 있다

- b. `H = built_hash(A)`:  $n$ 개의 값이 주어진 경우 해시 테이블 `H`에 모두 set하여 `H`에 저장한다. 이 경우 평균  $O(n)$  시간이면 된다

- i. 여기서 key 값은 `A[i]`의 인덱스  $i$  값이고 value 값은 `A[i]`가 된다. 즉,  $(i, A[i])$  쌍이 해시 테이블의 슬롯에 저장된다

c. `get_at(i):` # search 연산 수행 시간인 평균  $O(1)$  시간이면 충분  
`x = H.search(i)`  
`if x != None:`  
 `return x.value`  
`return None`

d. `set_at(i, x):` # set 연산 수행 시간인 평균  $O(1)$  시간이면 충분  
`H.set(i, x)`

e. `insert_at(i, x):`

- i.  $i$ 번째의 인덱스에 새로운 값  $x$ 를 삽입하기 위해서는  $i, i+1, \dots$ 에 저장된 값을 먼저 오른쪽으로 한 칸씩 이동해야 한다

- ii. 해시 테이블의 저장된  $(i, A[i]) \rightarrow (i+1, A[i])$ 로,  $(i+1, A[i]) \rightarrow (i+2, A[i+1]) \dots$  로 먼저 변경해야 한다. 이를 위해서는 해당 쌍을 해시 테이블에서 지우고 새로운 쌍을 삽입하는 방법으로 해결하면 된다

```
for k in range(i, len(H)):
 H.remove(k)
 H.set(k+1, A[k])
H.set(i, x)
```

수행시간은 평균 **O(n)**이 필요하다

- f. **delete\_at(i, x)**: `insert_at(i, x)`와 반대로  $(i, A[i])$  쌍을 지운 후, 오른쪽에 있는 쌍을 제거한 후 인덱스 값을 하나씩 줄인 후 다시 해시 테이블에 삽입하면 된다. 이 연산의 수행 시간 역시 평균 **O(n)**이면 충분하다
- g. **push\_back(x)**: # 평균 O(1) 시간  
`H.set(len(H), x)`
- h. **push\_front(x)**: 이 연산이 문제다! 가장 왼쪽에 삽입되기에 모든 값을 오른쪽으로 한 칸씩 이동해야 한다. 즉, 인덱스를 1씩 증가시켜야 한다. 그러면 **O(n)** 시간이 필요하다. 평균 O(1) 시간에 수행하기 위해서는 수 있을까?

`insert_at` 함수처럼 물리적으로 이동하게 되면 **O(n)** 시간을 피할 수 없다. 그러나 가장 왼쪽에 삽입하는 경우에는 가장 왼쪽 값의 인덱스가 0으로 고정하지 말고 -1, -2, ... 처럼 음수를 허용해 삽입이 가능하다. 어떻게 보면 `dequeue`와 유사하다.

인덱스가 음수도 될 수 있기에 현재 가장 왼쪽의 인덱스 값을 저장하는 변수를 정해 관리해야 한다. 이 변수를 `first`라고 하자. 그러면 `push_front(x)` 연산은 `first`에서 1을 뺀 인덱스를 `key` 값으로 하면 된다

최초에 `first = 0`로 초기화한 후, `push_front(x)`가 호출될 때마다 아래의 두 문장이 수행되면 된다

```
first = first - 1
H.set(first, x)
```

`first` 변수를 사용했기 때문에 앞의 `get_at`, `set_at`, `insert_at`, `delete_at` 도 그에 맞춰 수정해줘야 한다. 예를 들어, `get_at(i)`인 경우에 `A[i]`의 실제 해시 테이블의 인덱스 `key` 값은 **i+first**가 된다. 다른 연산도 같은 방식으로 수정하면 된다

결국, `push_front(x)`연산은 평균 **O(1)** 시간에 가능하다

- i. `pop_back()` 연산과 `pop_front()` 연산도 함께 생각해보자. `pop_back` 연산은 가장 마지막 인덱스가 `len(H)+first-1` 이므로 `H.remove(len(H)+first-1)`로 처리할 수 있다

반면에 `pop_front()` 연산은 `H.remove(first)`를 수행한 후, `first = first + 1`을 통해 `first` 값을 1 증가시키면 된다

따라서 이 두 연산도 평균 O(1) 시간에 가능하다

- j. push\_back, push\_front, pop\_back, pop\_front 연산을 모두 평균  $O(1)$  시간에 제공하기 때문에 stack, queue, dequeu 자료구조로 사용할 수도 있다. 주의할 것은 최악의 경우의 시간이 아닌 평균 시간이라는 것이다
- k. 바로 다음에 설명하는 내용처럼 python의 자료구조인 dict, set 등은 모두 해시 테이블로 구현되어 있다

## 11. [고급] Python Dict 구조 내부 구현 방식

Python의 `dict`는 해시 테이블로 관리되는 매우 효율적인 자료구조로 해시 테이블과 지원하는 연산이 같다. 사전 자료구조는 내부적으로 해시 테이블로 구현되어 있다. 어떤 해시 함수와 충돌회피전략을 사용하는지 살펴보자

- a. `dict`는 key 값과 value 값의 쌍을 저장하는 자료구조

```
>>> D = {}
>>> sys.getsizeof(D)
240
>>> D[2019317] = "신찬수" # key는 학번, value는 이름인 경우
>>> D[2019209] = "홍길동"
>>> print(D)
{2019317 : "신찬수", 2019209 : "홍길동"}
```

- b. 해시 테이블 크기

- i. **8개 슬롯으로 시작**, 테이블의 슬롯이  $\frac{1}{3}$  이상 차면 2배씩 슬롯을 늘려 `resize` 함. 즉, 해시 테이블의  $\frac{1}{3}$  이상은 항상 비어 있도록 유지된다
- ii. 2배씩 증가하거나 감소하므로 해시 테이블의 크기는 항상  $2^k$  유지
- iii. 슬롯은 (hashcode, key, value) 세 개의 값이 저장된다

- c. 해시 함수: hashcode 값을 정의하는 함수

- i. `hash(x)` 함수는 파이썬에서 제공하는 함수로  $x$ 가 정수라면  $x$  값 그대로 리턴하고, 문자열이면 매우 긴 정수 값을 리턴한다 (아래 12번 항목에서 자세하게 설명함)
- ii. Python에서 해시 테이블의 슬롯 번호  $i$ 는  $\text{hash}(\text{key}) \% \text{slot\_size\_in\_dict}$ 로 단순하게 계산한다 (`division` 해시 함수)

- d. 충돌 회피 방법: open addressing 사용

- i. 해시 함수 값이  $i$ 인 경우, 충돌이 발생하면  $(5*i)+1$ 에 있는 슬롯을 검사하는 방식과 유사한 방법을 사용한다 (따라서 linear probing은 아니고 random probing에 가깝다)

- ii. **질문**: 이렇게 위치를 검사하면 테이블의 모든 슬롯을 검사할 수 있을까?

- 예:  $m = 2^3$ ,  $i = 0$ 이라면 아래 순서대로 슬롯을 검사하므로 빈 슬롯이 존재하면 결국 찾게 된다!

$0 \rightarrow 1 \rightarrow 6 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 0$

- 위의 식을 임의의  $2^k$ 에 적용하면  $0 \dots 2^k$  까지의 값이 빠짐없이 한번씩 등장한다고 이미 증명되어 있다

- iii. 충돌시 검사할 슬롯 번호를 계산하는 정확한 수식은 다음과 같다

```
i = (5*i) + 1 + perturb
perturb >= PERTURB_SHIFT # PERTURB_SHIFT = 5
```

```
i = i % m # the next slot index to be checked
```

- perturb의 초기 값은 key 값의 상위 비트 값에 의해 결정 (이 부분에서 상위 비트 값이 슬롯 결정에 관여!)
- PERTURB\_SHIFT 같은 실현을 거쳐 5로 결정 (오른쪽으로 5 비트만큼 이동한다는 의미는 값을  $2^5$ 으로 나눈다는 것)
- perturb는 0 이상의 정수 (unsigned int)이므로  $2^5$ 으로 나누다 보면 결국 값이 0이 된다. 그 이후부터 perturb 값은 계속 0을 유지하므로  $i = (5*i + 1) \% m$  이 되어, 궁극적으로 모든 슬롯을 검사하게 된다. 따라서, 빈 슬롯이 있다면 반드시 찾게 된다

#### iv. set 연산

- 첫 슬롯이 비어 있다면 저장하고, 다른 key가 이미 저장되어 있고 같은 key가 아니라면 빈 슬롯을 찾아 위에서 설명한 open addressing 방법을 적용한다. 이렇게 찾은 빈 슬롯에 저장한다

#### v. remove 연산: del 명령

- 해당 key 값을 set 연산과 유사하게 probing해서 찾는다
- key = DUMMY, value = EMPTY로 처리한다 (DUMMY와 EMPTY는 None처럼 비어 있는 슬롯을 의미하는 미리 정의된 특별한 객체)
- [주의] 앞의 linear probing 방법에서 remove 연산은 삭제한 아이템의 슬롯에 이 키 값에 의해 밀려 저장된 아이템을 찾아 메꾸는 방식을 사용했지만, 여기서는 단순히 삭제되었다는 표시만 하고 메꾸는 과정을 하지 않았다. 그러면 나중에 삭제된 값에 의해 밀린 아이템을 찾을 때, DUMMY 표시가 된 슬롯을 만나더라도 계속 탐색을 해야 한다

- vi. 아래 코드는 key 값을 탐색하는 전 과정을 나타낸 코드이고, [https://just-taking-a-ride.com/inside\\_python\\_dict/chapter4.html](https://just-taking-a-ride.com/inside_python_dict/chapter4.html)에서 인용

```

PERTURB_SHIFT = 5

def search(key, slots_count):
 hash_code = hash(key) # hash 함수는 아래 12번에서 설명
 if hash_code < 0:
 perturb = 2**64 + hash_code
 else:
 perturb = hash_code
 i = hash_code & mask # mask = (slot size - 1)
 visited = set()
 while len(visited) < slots_count:
 visited.add(i)
 if H[i] is unoccupied: # not found
 return None
 if H[i].key != DUMMY and H[i].key == key: # found!
 return H[i]
 i = (i * 5 + perturb + 1) % slots_count
 perturb >>= PERTURB_SHIFT

 return FULL # full and not found!

```

- vii. 실험 결과에 따르면 매우 효율적이고 빠르다고 알려짐!

- 실제 python의 dict 구현 코드  
<https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c#l33>
- dict의 해시 과정과 연산의 동작 과정을 상세하게 설명한 페이지  
[https://just-taking-a-ride.com/inside\\_python\\_dict/chapter4.html](https://just-taking-a-ride.com/inside_python_dict/chapter4.html)

## 12. Python에서 제공하는 hash 함수

- a. int, float, str, bool 값에 대해, hash 함수를 호출하면 다음과 같다

```

>>> hash("python")
-254091099
>>> hash("chansu")
-1902128559
>>> hash(17)

```

```
17
>>> hash(-29)
-29
>>> hash(True)
1
>>> hash(False)
0
>>> hash(3.141592)
1780711371
>>> hash([1, 2, 3])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

- b. 문자열에 대해선 음수 값이, 정수는 정수 값 그대로, 다른 값에 대해서는 매우 큰 정수 값으로 매팅됨에 주의하자

이 hash 함수를 이용해 정수 값을 얻은 후, 여러분이 설계한 해시 함수를 호출하는 식으로 활용하면 된다

- c. 사용자가 정의한 class를 hashable하도록 하고 싶다면, class의 magic 메쏘드로 \_\_hash\_\_ 함수를 구현하면 된다

## 해시 테이블 관련 인터뷰 문제 몇 가지

13. [マイク icon] 인터뷰 문제1] **Uniqueness problem**: n개의 수가 입력으로 주어질 때, 이 수들이 모두 다른 수인지 판별하는 (고전) 문제 (모두 다르면 True, 아니면 False로 판별)

- a. 입력된 수가 리스트 A에 저장되어 있다고 가정

- b. 알고리즘 1:  $O(n^2)$  시간 (각  $A[i]$ 에 대해  $A[i]$ 와 같은  $A[j]$ 가 있는지 검사)

```
for i in range(n):
 for j in range(i+1, n):
 if A[i] == A[j]: return False
return True
```

- c. 알고리즘 2:  $O(n \log n)$  시간

```
sort A # O(n log n) 시간 필요
for i in range(1, n):
 if A[i-1] == A[i]: # 왜 인접한 두 수만 점검하면 되나?
 return False
return True
```

- d. 알고리즘 3:  $O(n)$  시간 [단, 최악의 시간이 아니라 평균 시간임에 유의!]

```
H = Hash(2*n) # slot의 개수가 2n인 해시 테이블 생성
m = 2n으로 한 이유는?

for x in A:
 if H.search(x) != None: # 무슨 의미?
 return False
 else:
 H.set(x)
return True
```

- e. [고급] Uniqueness 문제의 (비교 모델에서의) 하한은  $\Omega(n \log n)$ 으로 증명됨!

- i. 이는 두 수를 비교해서 uniqueness를 판별하는 계산 모델인 linear decision tree 모델에서의 하한이다 (즉, 두 수를 비교해 결과가 -, 0, + 셋 중 하나이고, 그 결과에 따라 다음 비교를 반복하는 계산 모델)

- ii. 참고 자료:

<http://jeffe.cs.illinois.edu/teaching/497/06-algebraic-tree.pdf>

- iii. 방법 3의  $O(n)$  시간은 평균시간을 의미하므로 하한과 모순되지 않음에 유의하자

- f. 이와 유사한 문제로 두 집합 A, B가 주어지고 이 두 집합에 공통으로 속하는 원소가 있는지를 판별하는 Set Intersection 문제가 있다 → 세 가지 방법을 생각해 보자!

14. [🎤 인터뷰 문제 2 - pair sum]  $n$ 개의 정수 값이 저장된 리스트 A와 정수 값 K가 입력으로 주어진다. 더해 K가 되는 A의 두 수를 찾아라

- 가장 단순한 방법은 모든  $A[i]$ 와  $A[j]$  쌍에 대해, 합이 K가 되는지 검사하면 된다. 이중 for 루프를 사용하면  $O(n^2)$  시간에 가능
- 더 빠르게 할 수 없을까?
- [힌트] 오름차순으로 먼저 정렬을 하자.  $i = 0$ ,  $j = n-1$ 에서부터 i는 증가하고 j는 감소하면서  $A[i] + A[j] == K$ 인지 검사하는 식으로 선형 탐색을 해보자
- 만약,  $A[i] + A[j] < K$ 인 경우엔 i를 늘려야 할까, j를 줄여야 할까?  $A[i] + A[j] > K$ 인 경우에는?
- 이 알고리즘의 수행시간은?  $O(n \log n)$  시간 정렬 후  $O(n)$  시간 선형 스캔만 하면 되기에  $O(n \log n)$  시간이면 충분하다
- 이 보다 더 빠르게 가능할까?
  - 힌트: 삽입과 검색이 평균  $O(1)$  시간에 가능한 해시 테이블 사용?

15. [🎤 인터뷰 문제 3 - swap sum] 두 리스트 A와 B에 각각  $n$ 개와  $m$ 개의 정수 값이 저장되어 있다. A의 값 하나와 B의 값 하나를 서로 교환해서 A의 값의 합과 B의 값의 합이 같도록 만들고 싶다. 어떤 두 수를 선택해 교환해야 할까? 물론 그런 쌍이 없을 수도 있다

- 역시, 가장 단순한 방법은 모든 쌍을 고려해 교환해보는 방법이지만, 더 빠른 방법이 존재한다.
- A의 수 a와 B의 수 b를 서로 교환한다고 하자. 그러면 교환 후 합은 각각  $\text{sum}(A) - a + b$ ,  $\text{sum}(B) - b + a$ 가 된다. 두 합이 같아야 하므로  $\text{sum}(A) - \text{sum}(B) = 2(a - b)$ 가 된다
- $c = \text{sum}(A) - \text{sum}(B)$ 라고 하면, c는 상수이다. 즉,  $(a-b)$ 의 2배가 상수 c인 쌍 (a, b)를 찾으면 된다!
  - [힌트] 바로 앞의 pair sum 문제와 유사하지 않나? 그렇다. 결국  $2a + (-2b) = c$ 가 되는 (a, b) 쌍을 구하는 것이다. A의 값에 2배를 하고 B의 값에 -2배를 해서 그 합이 c ( $= \text{sum}(A) - \text{sum}(B)$ )가 되는 쌍을 찾으면 된다
- 이 알고리즘의 수행시간은? 정렬 후 스캔하는 알고리즘은  $O(n \log n)$  시간이면 충분하고, 해시 테이블을 사용하는 알고리즘은 평균  $O(n)$  시간이면 된다

16. [🎤 인터뷰 문제 4 - majority] 리스트 A에 양의 정수 n개가 저장되어 있다. 전체 개수의 절반 넘게 등장하는 수를 majority 수라고 정의한다. 예를 들어,  $A = [1, 4, 1, 2, 1]$ 이라면, 전체 5개의 수 중에서 1이 3개로  $5/2 = 2.5$ 개보다 많기에 majority 수이다. 그러나  $A = [1, 3, 1, 2]$ 라면, 1이 2번 등장하지만,  $4/2 = 2$ 개를 넘지 않기 때문에 1은 majority 수는 아니고 2와 3 역시 majority 수가 아니기에 이 리스트에는 majority 수가 없다. 여러분들은 주어진 A에서 majority 수가 있다면 출력하고 없다면 -1을 출력하는 코드를 작성해야 한다

- a. [느린 방법] 모든  $A[i]$ 가 몇 번씩 나타나는지  $O(n)$  시간에 검사한 후,  $n/2$ 번 보다 많이 등장하는 수가 있는지 본다 -  $O(n^2)$  시간
- b. [약간 빠른 방법]  $O(n \log n)$  시간에 오름 차순으로 정렬한다. 정렬된 값을 차례로 보면서 (linear scan) 각 수가 몇 번씩 나타나는지 세는 건 쉽다. 결국  $O(n \log n)$  시간이면 충분 (조금만 더 생각해보면 각 수를 셀 필요도 없다!)
- c. [평균적으로 빠른 방법] 삽입, 탐색에 평균  $O(1)$  시간이 걸리는 해시 테이블을 사용한다. 해시 테이블 아이템은  $(A[i], c_i)$ 로  $c_i$ 는  $A[i]$ 의 등장 횟수로 정의한다. 모든  $A[i]$ 를 삽입하는 데, 이미 테이블에 있다면  $c_i$ 를 1 증가시킨다. 평균  $O(n)$  시간에 해결 가능하다
- d. [최악의 경우에도 빠른 방법] 아래 예를 보자. A의 값을 왼쪽부터 하나씩 보면서 판단을 해보자.

$$A = [1, 2, 3, 1, 4, 1, 1, 1, 2, 3, 1]$$

- i.  $A[0] = 1, A[1] = 2$ 에서 알 수 있는 사실은? 현재까지 본 2개의 값만을 한정하면, 1은 majority 수가 아님을 알 수 있다. (물론, 나중에 1이 많이 등장해 majority 수로 판명될 수도 있다)
- ii. 같은 이유로 2 역시 현재까지는 majority 수는 아니다
- iii.  $A[2] = 3, A[3] = 1$ 로 서로 다르므로, 역시 두 수도 현재까지 고려한 수들만 고려할 때 majority 수가 아니고, 역시  $A[4] = 4, A[5] = 1$ 이므로 두 수 모두 majority 수가 아니다.
- iv.  $A[6] = 1, A[7] = 1$ 이 되어 처음으로 같은 수가 등장했다. 1이 현재까지 중에 majority 수가 될 수 있는 가능성이 있다.  $majority = 1, count = 2$ 로 기록한다
- v.  $A[8] = 2$ 이고, majority 수인 1과 다르므로 count를 1 감소
- vi.  $A[9] = 3$ 이고 majority 수인 1과 다르므로 count를 1 감소하면, 결국 count = 0이 된다. 그러면 1은 더 이상 majority 수가 아니다 (즉, 현재까지 1이 등장한 횟수만큼 1이 아닌 수가 등장했다는 의미이므로)
- vii.  $A[10] = 1$ 이 되었고, 현재 count = 0이므로 majority = 1로 수정한다. 더 이상 수가 없으므로 majority = 1이 된다!
- viii. 이 방법이 항상 올바른 답을 출력하는가? 더 필요한 과정은 없을까? 최종 알고리즘의 수행시간이  $O(n)$ 임을 확인해보자

17. [マイクアイコン 인터뷰 문제 5: longest equal number] A와 B가 섞인 문자열이 s가 주어진다. 그러면 A와 B가 같은 개수로 등장하는  $s[i] \dots s[j]$ 까지의 부문자열(substring) 중에서 가장 긴 길이의 부문자열을 찾아보자

a. 예:  $s = AABABBAABABAAB$

```
a = 1223334566777899 # a[i] = s[0]...s[i]의 A의 개수
b = 0011233334456667 # b[i] = s[0]...s[i]의 B의 개수
d = 1212101232321232 # d[i] = a[i] - b[i]
```

b. 리스트 d의 값은 무엇을 의미하나?

i.  $d[0] = d[2] = 1$ 로 두 값은 서로 같다. 무슨 의미일까?

ii. d의 값이 같은  $d[i]$ 와  $d[j]$ 에 대해,  $s[i+1] \dots s[j]$ 는 두 문자의 개수가 같은가? Yes!

iii. 그러면 d의 값이 같은 가장 멀리 떨어진 인덱스 쌍 ( $i, j$ )를 구하면 되지 않나?

- 위의 예에서 가장 멀리 떨어진 쌍 ( $i, j$ )는?

iv. 이 인덱스 쌍은 어떻게 구해야 할까?

- 해시 테이블을 이용하는 방법 : 평균  $O(n)$  시간

$d$ 의 값을 해시 테이블에 삽입하는 데, key 값은  $d$ 의 값이 되고, value는  $d$ 의 index의 최소 값과 최대 값을 유지하면 된다. 같은 key 값이 삽입되면, value 값인 (최소 인덱스와 최대 인덱스) 쌍을 현재 삽입된 값의 index와 비교해 업데이트하면 된다

- 제 3의 리스트 c를 이용하는 방법:  $O(n)$  시간

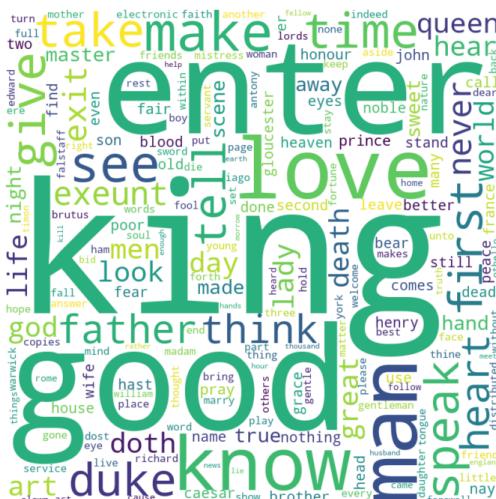
$d$ 의 값은 0부터  $n$ 까지 정수 값 중 하나이다. 따라서 새로운 배열  $c[0], \dots, c[n]$ 까지 ( $n+1$ )개의 값을 저장할 수 있는 리스트를 선언한다.  $c[k]$ 에는  $d$  값이  $k$ 인 (최소 인덱스, 최대 인덱스) 쌍을 유지한다.  $i = 0, \dots, n-1$ 에 대해,  $c[d[i]]$ 의 인덱스 쌍을 관리하면 된다

18. [💪] [해보기-medium] 파일 `simplemaps-worldcities-basic-refined.csv`은 전 세계 7322개 도시의 몇 가지 정보를 담고 있는 csv 포맷의 영문 텍스트 파일이다

- a. 도시 정보는 `city_name, latitude, longitude, population, country, iso3`(3글자 영문 약자)이다
- b. 두 개의 해시 테이블 `HO, HC`를 준비한다:
  - i. `m = the size of hash table = 10,000`
  - ii. `HO`는 충돌회피 방법으로 open addressing의 **linear probing**을 사용하고, `HC`는 한방향 연결리스트 클래스를 이용한 **chaining**을 사용한다
- c. 역시 두 개의 해시 함수 `f1` 과 `f2`를 직접 만들어야 한다. 함수를 위한 key는 위의 도시 정보 중에서 하나를 이용하거나 여러 개를 조합하여 자유롭게 설계하면 된다
- d. linear probing과 chaining을 위한 함수를 `find_slot, search, set, remove`를 각각 구현한다
- e. 두 해시 함수와 두 충돌회피방법의 성능을 측정해 비교 분석한 레포트도 제출한다. 측정할 요소는:
  - i. `n` = set, remove, search 의 총 연산 횟수
  - ii. `m` = 해시 테이블의 slot 개수
  - iii. load factor = `LF = n/m`
  - iv. resize 기준: load factor `LF`가 특정 값 이상이면 resize한다. 이때 특정 값을 0.5, 0.7, 0.9 등으로 다양하게 변화시키며 아래 값들을 측정해 보자
  - v. 평균 충돌 횟수 = collision ratio = `(number of collisions)/n`
  - vi. 평균 비교 횟수 = `(연산에서 실행한 총 비교 연산 횟수)/n`
  - vii. 평균 탐색(search) 시간: key가 테이블에 있는 경우(successful search)와 없는 경우(unsuccessful search)를 나눠서 별도로 평균 비교 횟수를 측정
- f. 위의 성능을 측정하기 위해서 임의로 충분한 횟수의 `set, search, remove` 등을 반복해서 호출하는 실험을 해야 함
- g. 이 성능 측정 값들을 csv 파일에 append 모드로 계속 저장하는 코드를 작성하고, 이 파일의 값을 읽어 파이썬의 `matplotlib` 패키지의 `pyplot` 모듈을 이용해 실시간으로 그래프를 그리는 코드를 별도로 작성해보자
  - i. `matplotlib`는 python 교재와 무료 설명 동영상 참고
  - ii. 라이브 데이터 그래프 그리기 동영상 추천: <https://youtu.be/Ercd-lp5PfO>

19. [해보기-medium] Shakespeare의 작품에 등장하는 단어의 빈도수 (frequency)를 계산해 관심있는 빈도수의 단어들을 word cloud를 이용해 가시화해 보자

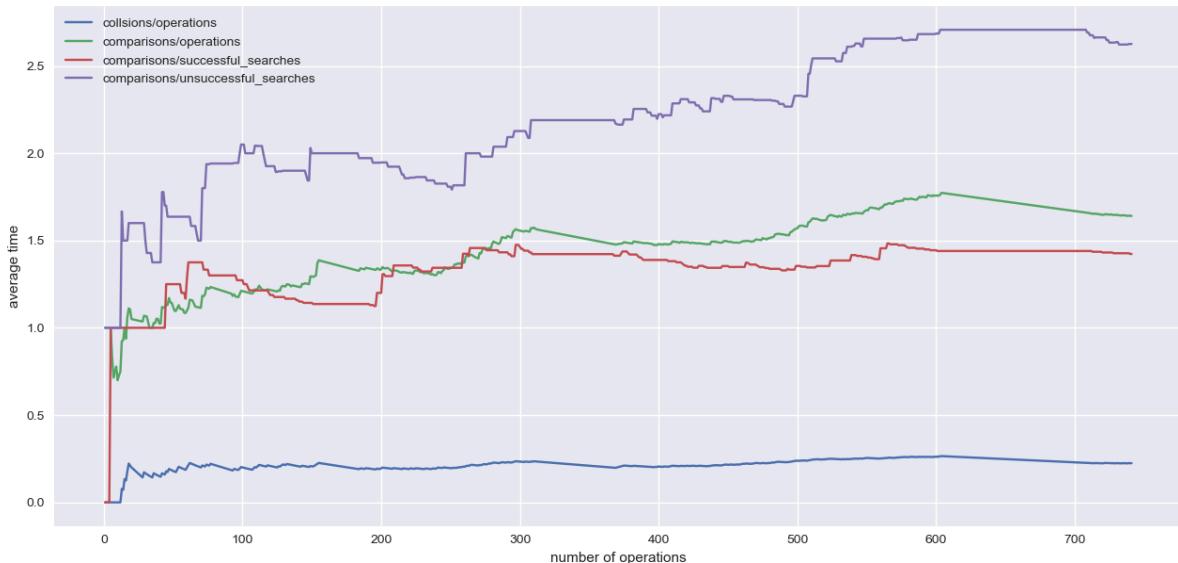
- a. 세이克斯피어의 희곡이나 산문은 인터넷에서 text 파일 형식으로 쉽게 구할 수 있다. 이 파일을 읽어, 각 단어들이 몇 번 등장하는지 횟수를 세는 문제를 풀어보자
    - i. <http://www.gutenberg.org/files/100/100-0.txt>에서 다운 받은 후, 앞 뒤 불필요한 설명 부분을 제거한 후 사용
  - b. 먼저 줄 단위로 읽어 영단어로 split해야 한다. 이를 위해 regular expression 모듈의.findall 함수를 이용한다 (import re 필요)
    - i. words = re.findall('\w+', line) # line에 포함된 모든 영어 단어를 찾아 리스트로 만들어 리턴함
  - c. 해당 단어의 등장 횟수를 기록하기 위한 방법은 다양하지만, 여기서는 hash table 자료구조를 사용해보자. 어떤 단어가 해시 테이블 H에 있다면, 해당 단어의 value (counter 역할) 값을 하나 증가시켜 set하고, 없다면 value = 1로 set하면 된다
  - d. resize 기준을  $n/m = 0.5$  정도로 정하면 총 단어의 개수에 비례하는 시간에 횟수를 계산할 수 있다
  - e. a, an, the, I, you, at, of 등과 같은 관사, 정관사, 전치사, 대명사처럼 빈도수 자체가 중요하지 않는 단어나 숫자로 구성된 단어 (numeric word) 등은 굳이 고려할 필요가 없으므로 stop words로 지정해 고려하지 않으면 된다
    - i. nltk (natural language toolkit) 고모모
    - ii. wordcloud 모듈의 wordcloud.STOPWORDS에도 정의되어 있으므로 이것을 stop words로 사용해도 된다
  - f. 해시 테이블에 있는 단어의 횟수를 내림차순으로 정렬하면, 첫 번째 단어가 가장 많이 등장하는 단어이다. 이 중 원하는 랭킹에 속하는 단어들을 word cloud로 가시화한다
    - i. wordcloud 모듈과 matplotlib를 모듈을 이용해 가시화할 수 있다
    - ii. 아래 예는 빈도수 랭킹이 1위부터 300위까지 가시화한 것이다



- iii. 사용법은 [https://lovit.github.io/nlp/2018/04/17/word\\_cloud/](https://lovit.github.io/nlp/2018/04/17/word_cloud/) 사이트에  
잘 정리되어 있으니 참조 바람 (한글 wordcloud도 가능)

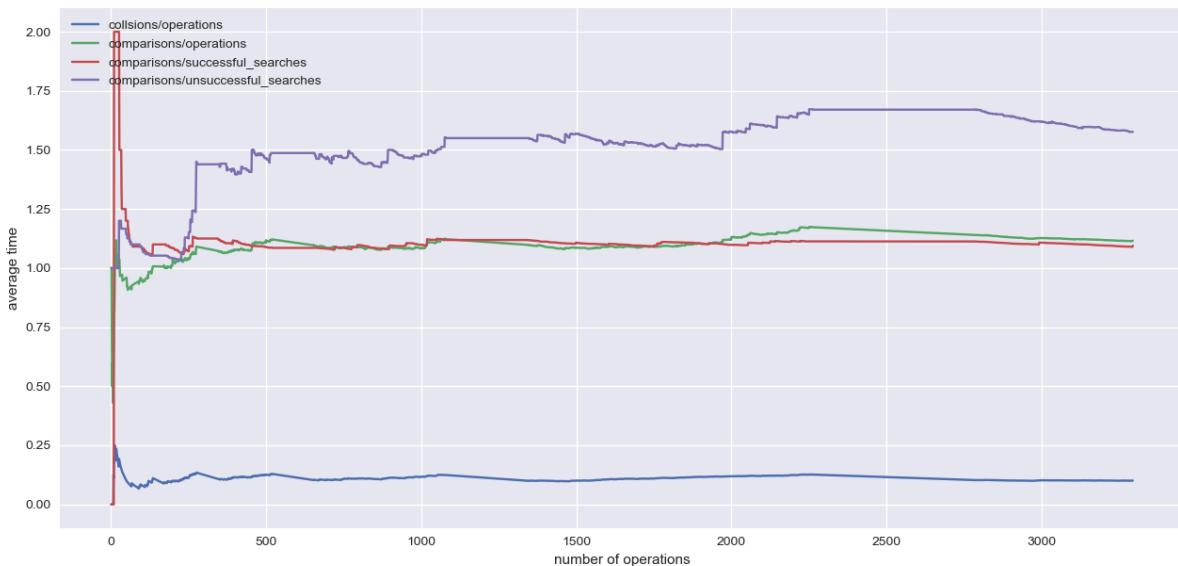
## 20. [LF에 따른 해시 테이블 성능 실험] 해시 테이블 크기 $m = 8$ 로 시작해서

- 해시 함수:  $f(k) = k \% m$  사용 (성능이 좋은 해시 함수는 절대 아님. 간단한 실험용으로)
- open addressing - linear probing으로 충돌 해결
- set, remove, search을 랜덤하게 호출해, 비교횟수, 충돌 횟수 등을 측정함
- LF 값을 LF = 0.5, 0.8로 달리하면서 측정함



y-축은 연산 하나당 평균 비교 또는 충돌 횟수

첫 슬롯의 개수를  $m=8$ 개부터 시작해  $LF \geq 0.8$ 이면 resize하도록 지정한 결과



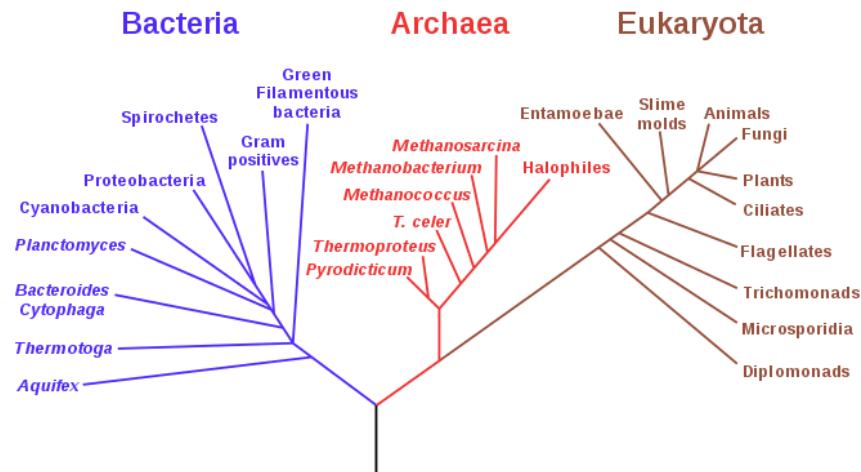
첫 슬롯의 개수를  $m = 8$ 개부터 시작해  $LF \geq 0.5$ 이면 resize하도록 지정했다.  
비교/충돌 횟수(연산 시간)이  $LF \geq 0.8$ 인 경우보다 상당히 빨라졌음을 확인할 수 있다

## 5. Tree (+ heap)

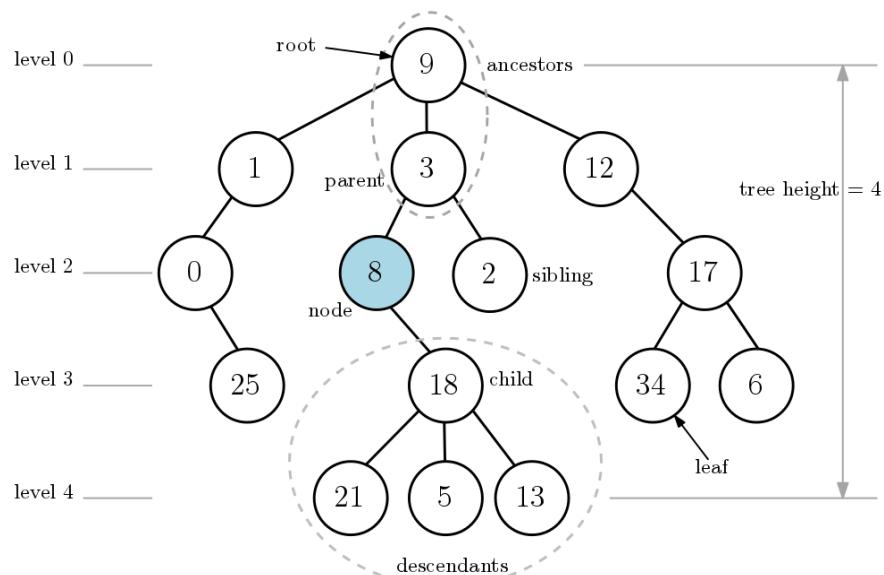
1. 매우 중요한 자료구조
2. 연결리스트는 노들들이 한 줄로 연결된 선형적인 자료구조인 반면 트리는 부모-자식 관계를 계층적으로 표현한 보다 일반적인 자료구조이다

a. 예 1: 유전 트리 (Phylogenetic tree)

[https://en.wikipedia.org/wiki/Phylogenetic\\_tree](https://en.wikipedia.org/wiki/Phylogenetic_tree)



b. 예 2: 추상적으로 표현하면 연결 리스트처럼 데이터를 저장하고 있는 노드(node)와 노드를 연결하는 에지(edge 또는 링크 link)로 구성된다. 연결 리스트와 차이점은 에지가 **부모-자식 관계를 표현**한다는 것



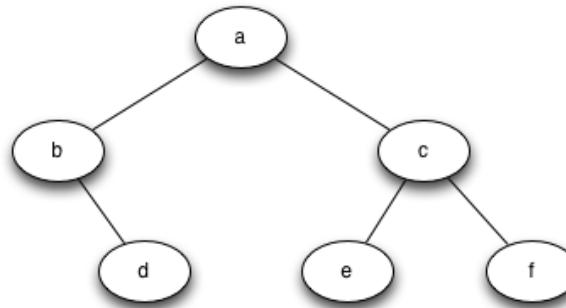
c. 용어정리

- i. 부모노드/자식노드: 3은 8, 2의 부모노드, 8, 2는 3의 자식 노드
- ii. 조상노드/자손노드: 9는 8의 조상노드, 8은 9의 자손노드
- iii. 루트(root) 노드: 모든 노드의 조상 노드: 9
- iv. 리프(leaf) 노드: 자식이 없는 노드

- v. 레벨(level): 루트 노드의 레벨 0를 시작으로 한 세대씩 내려가면서 1씩 증가
  - vi. 깊이(depth): 루트에서 다른 노드를 연결하는 에지 개수(노드 5의 깊이 = 4)
  - vii. 경로(path): 두 노드 사이를 연결하는 에지의 시퀀스
    - 18와 12 사이의 경로 =  $18 \rightarrow 8 \rightarrow 3 \rightarrow 9 \rightarrow 12$
    - 경로의 길이는 에지의 수, 위 경로의 길이는 4
  - viii. 높이(height): 노드의 높이는 자식 노드까지의 가장 큰 깊이
    - 노드 3의 높이는  $3 \leftarrow 3$ 에서 21, 5, 13이 가장 깊은 곳에 있으므로
  - ix. 트리 높이(tree height): 루트 노드의 높이가 트리 높이이며, 여기선 4임
  - x. 분지수(degree): 노드의 분지수는 자신의 자식 수이며, 트리의 분지수는 가장 큰 분지수로 정의된다. 18의 분지수는 3, 6의 분지수는 0, 트리의 분지수는 3
  - xi. 부트리(subtree): 어떤 노드와 그 노드의 자손노드들로 구성된 부분 트리
- d. (한방향) 연결 리스트는 head 노드가 루트 노드이며, tail 노드를 제외한 모든 노드가 자식이 모두 하나인 트리로 볼 수 있다 (매우 단순한 트리)

### 3. Binary tree(이진트리) 정의와 표현법

- a. 이진트리: 모든 노드의 자식이 2개를 넘지 않는 트리
  - i. 실제 자료구조에서 사용되는 트리는 이진트리가 대부분임
- b. 힙: 특별한 성질을 갖고 있는 이진트리 중 하나
  - i. 최대 값 (또는 최소 값)을 빠르게 찾을 수 있는 이진트리 자료구조



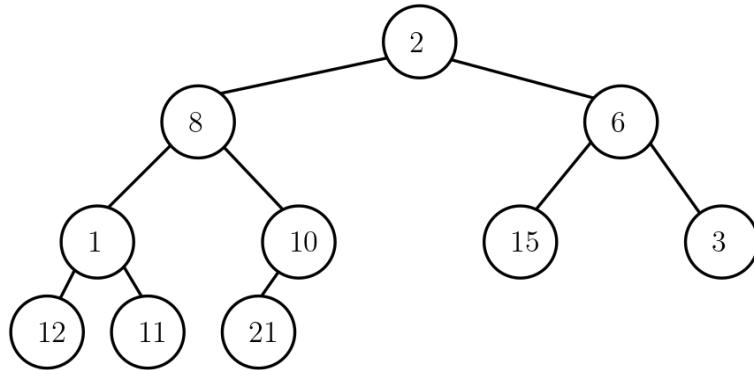
- c. 표현법 1: 하나의 리스트만으로 표현 → 일반적인 힙의 표현 방법이기도 함
 

```
[a, b, c, None, d, e, f] # 레벨 0부터 차례로, 왼쪽부터 오른쪽으로
 # 레벨을 색으로 구분했음
 # b의 왼쪽 자식노드가 없으니 None으로 표시
```
- d. 표현법 2: 리스트를 중복해서 표현
 

```
[a, [b, [], [d, [], []]], [c, [e, [], []], [f, [], []]]]
```

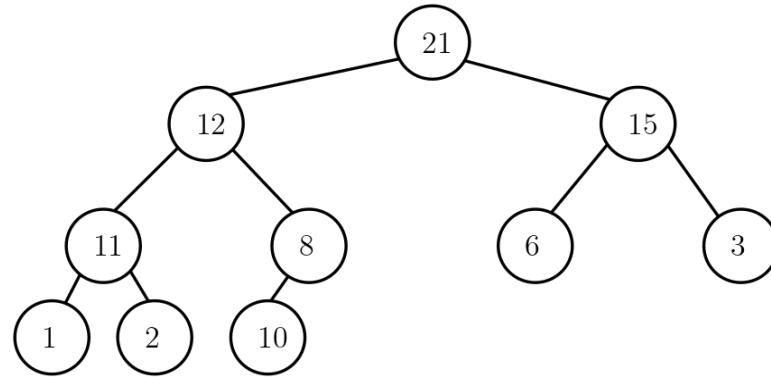
  - i. [루트 a, a의 왼쪽 부트리, a의 오른쪽 부트리] 형식으로 재귀적으로 정의
  - ii. b 노드 입장에서는 [b, [], [d, [], []]]이 되며, d 노드는 리프 노드이므로 [d, [], []]가 된다
- e. [?] 두 표현법의 장점과 단점은 무엇일까?
- f. [?] 표현법 1로 표현된 트리 리스트 A = [a, b, c, None, d, e, f]에서,

- i. 루트 a는 A[0]에 저장되어 있고, 왼쪽 자식노드는 A[1]에 오른쪽 자식노드는 A[2]에 저장되어 있다
- ii. 노드 c는 A[2]에, c의 왼쪽과 오른쪽 자식노드는 각각 A[5], A[6]에 저장되어 있다
- iii. [?] A[k]에 저장된 노드의 왼쪽 자식노드는 A[2\*k+1]에, 오른쪽 자식노드는 A[2\*k+2]에 저장된다
- iv. [?] A[k]의 부모 노드는 A[(k-1)//2]에 저장된다
- v. [?] 이 표현법의 문제는 빈 칸이 많을 경우 메모리 낭비가 심하는 것이다.  
이런 일이 발생하는 극단적인 이진트리 모양은 어떤걸까? (오른쪽 자식 노드 하나만을 갖는 트리 - 오른쪽 방향으로 극단적으로 기울어진 연결 리스트 형태의 트리)
- 예: A = [a, \_, b, \_, \_, \_, c, \_, \_, \_, \_, \_, \_, \_, d]는 노드 4개를 갖는 트리인데, 빈 칸은 11 칸이나 되어 메모리 낭비가 심하다. 이런 모양으로 노드 n개를 갖는 트리를 정의하면 A의 크기는 O(2^n)으로 매우 커지게 된다
- vi. 빈 칸을 만들지 않기 위해선, 레벨마다 노드를 빼지 않고 가득 채우고, 마지막 레벨엔 왼쪽부터 노드를 채워나간 트리 모양이면 된다 (아래 그림 참조) 다른 의미로는 리스트에 차례대로 저장된 값을 이진트리로 해석하면 레벨마다 가득 찬 모양이 된다는 의미다



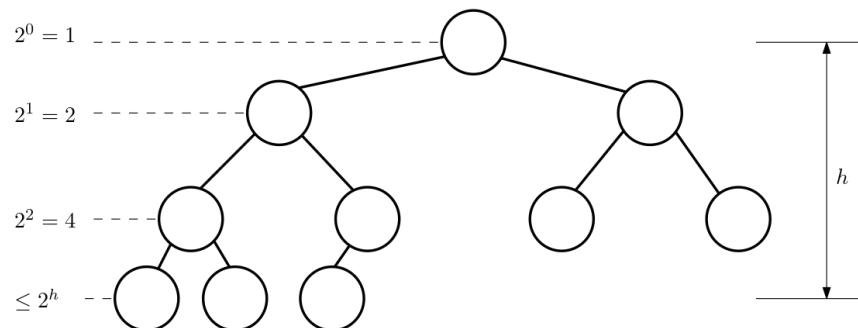
#### 4. 힙(heap): 다음의 모양과 힙 성질을 만족하는 리스트에 저장된 값의 시퀀스

- a. (모양 성질) 다음과 같은 모양의 이진트리여야 한다:
  - i. 마지막 레벨을 제외한 각 레벨의 노드는 모두 채워져 있어야 한다
  - ii. 마지막 레벨에선 노드들이 왼쪽부터 채워져야 한다
- b. (max-힙 성질) 루트 노드를 제외한 모든 노드에 저장된 값(key)은 자신의 부모 노드의 값보다 크면 안된다
  - i. 위의 그림의 이진트리는 모양 성질은 만족하지만 힙 성질은 만족하지 않는다. (8의 부모가 2가 되어 성질 위배)
  - ii. 반면에 아래 그림은 모양과 힙 성질 모두 만족한다



|    |    |    |    |   |   |   |   |   |    |
|----|----|----|----|---|---|---|---|---|----|
| 21 | 12 | 15 | 11 | 8 | 6 | 3 | 1 | 2 | 10 |
|----|----|----|----|---|---|---|---|---|----|

- iii. [매우 중요] 힙 성질에 따라 루트 노드에는 가장 큰 값이 저장되게 된다
- iv. 여기서 주의할 건, 실제 데이터 값은 리스트에 저장되어 있고 리스트의 값이 표현하는 (가상의) 이진트리가 모양 성질을 만족한다는 의미이다
- v. 힙의 높이  $h$ :
  - $n$ 개의 값으로 구성된 힙의 높이  $h$ 는 최대 어느 정도일까?
  - 모양 성질에 따르면, 레벨 0부터  $h-1$ 에 있는 노드는 모두 채워져 있어야 하므로, 노드 개수가 총  $1 + 2 + 2^2 + \dots + 2^{h-1} = 2^h - 1$ 이다
  - 레벨  $h$ 에는 하나 이상의 노드가 존재하므로 전체 노드 수  $n \geq 2^h$  이 성립한다
  - 양변에  $\log_2$ 를 취하면  $h \leq \log n$ 이 성립한다. 따라서,  **$h = O(\log n)$** 이다



- vi. Heap 클래스:

```
class Heap:
 def __init__(self, L=[]): # default: 빈 리스트
 self.A = L
 self.make_heap() # A의 값을 힙성질이 만족되도록
 # make_heap 함수 호출 (추후설명)

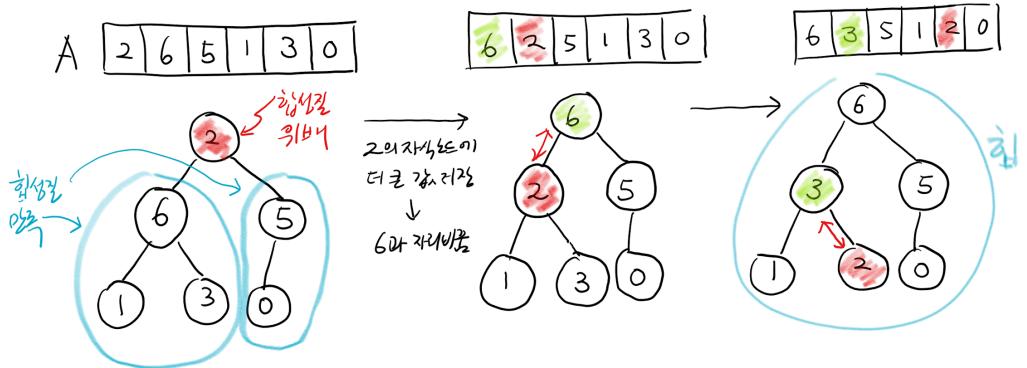
 def __str__(self):
 return str(self.A)
```

- vii. 리스트에 저장된 값을 이진트리로 해석하면 자동으로 모양 성질을 만족한다. 그러나 힙 성질을 만족하지 않을 수도 있다

- viii. 힙 성질을 만족하지 않으면, 값들을 재배열해서 힙을 만들 수 있다. 이 작업을 하는 함수를 `make_heap` 함수라 부른다. 이를 위해, **heapify\_down** 함수가 필요하다

ix. **heapify\_down(k)** 함수:

- $A[k]$ 를 제외하고  $A[k]$ 의 모든 자손 노드들이 모두 힙 성질을 만족한다고 가정할 때,  $A[k]$  값을 (필요하다면) 아래로 내려가면서 힙 성질을 만족하는 위치로 이동시키는 함수



```

def heapify_down(self, k, n):
 # n = 힙의 전체 노드수 [heap_sort를 위해 필요함]
 # A[k]를 힙 성질을 만족하는 위치로 내려가면서 재배치

 while 2*k+1 < n: # 조건문이 어떤 뜻인가?
 L, R = 2*k + 1, 2*k + 2 # L, R은 왼쪽, 오른쪽 자식노드
 if self.A[L] > self.A[k]: # 세 노드의 최대 값의 인덱스 찾기
 m = L
 else:
 m = k
 if R < n and self.A[R] > self.A[m]:
 m = R
 # m = A[k], A[L], A[R] 중 최대값의 인덱스

 if m != k: # A[k]가 최대값이 아니면 힙 성질 위배
 self.A[k], self.A[m] = self.A[m], self.A[k]
 k = m
 else:
 break # [?] 왜 break 할까?

```

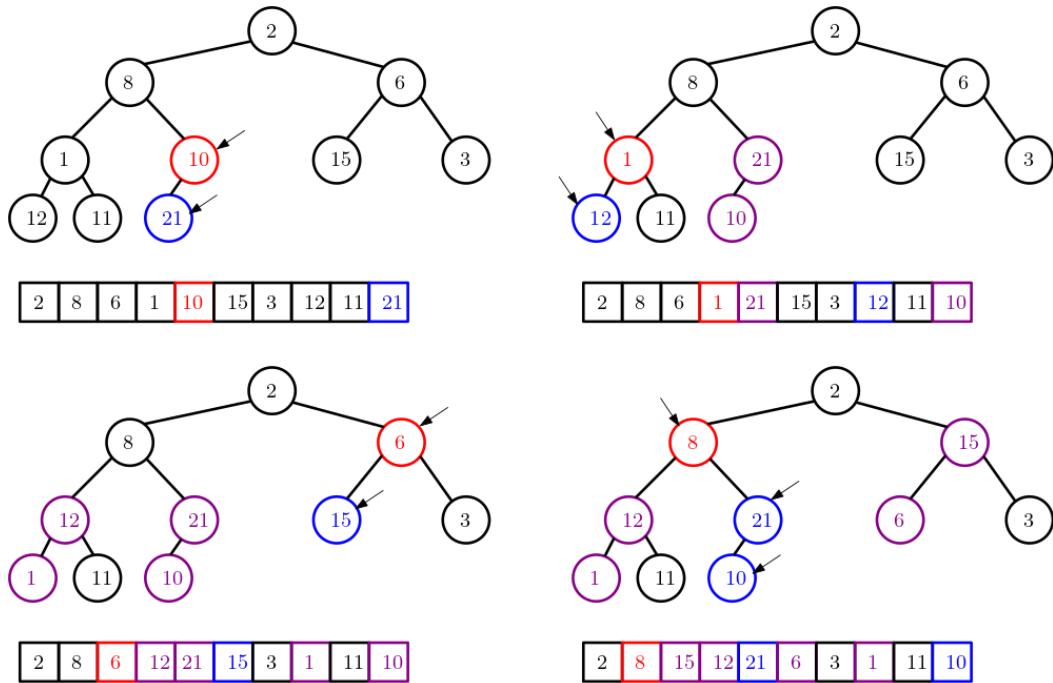
- **heapify\_down의 수행시간**을 알아보자
  - 수행시간은  $A[k]$ 가 내려온 레벨 수에 비례한다
  - 가장 많이 내려오려면 루트인  $A[0]$ 가 힙의 높이(가장 깊은 곳의 리프)까지 내려오는 것이다.
  - 한 레벨 내려올 때의 연산은 상수번의 비교를 하고 1번의 자리 바꿈을 하는 것이므로  $O(1)$  시간이면 충분하다.
  - 따라서, 최악의 경우에 힙의 높이에 비례하는 시간이 필요하다. 즉,  **$O(\log n)$  시간**이 필요하다

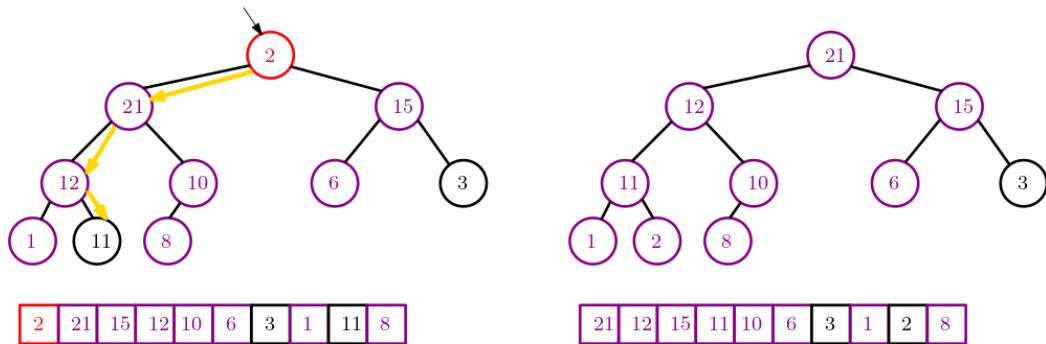
X. `make_heap`: 현재 리스트의 값들을 힙 성질을 만족하도록 재배열하는 함수

- 리스트 A의 각 값에 대해 `heapify_down`을 호출해 재배치한다
- 어떤 값부터 차례로 `heapify_down`을 호출해야 할까? 마지막 레벨의 가장 오른쪽에 위치한 수부터 루트 노드에 저장된 수까지 차례대로 호출하면 된다

```
def make_heap(self):
 n = len(self.A)
 for k in range(n-1, -1, -1): # A[n-1] → A[0] 순서로
 self.heapify_down(k, n)
```

- `range(n-1, -1, -1) → range(n//2, -1, -1)`로 변경해도 문제 없음 (왜 그럴까?)
  - 리프 노드는 자식 노드가 없기 때문에 `heapify_down`을 할 필요가 없다
  - 힙에는 반드시 리프 노드이고 나머지 반드시 리프가 아닌 노드이다.
  - 계산해보면,  $A[n//2+1], \dots, A[n-1]$ 은 모두 리프 노드이다 (쉽게 확인할 수 있다)
  - 따라서  $A[n//2], \dots, A[0]$  순서로 `heapify_down`을 하면 충분하다
  - 아래 그림은 `make_heap`의 과정을 차례대로 따라가면서 그린 것이다





- make\_heap의 수행시간을 분석해보자
  - **분석 1:** for 반복문은 n번 반복되고, 반복할 때마다 heapify\_down이 한번씩 호출된다. heapify\_down의 수행시간이  $O(\log n)$ 이므로  $O(n \log n)$  시간이면 충분하다
  - **[고급] 분석 2:** 엄밀하게 분석하면  $O(n)$  시간이면 충분하다 (아래 분석 참조. 건너 뛰어도 됨)
    - 레벨 i에 있는 노드 수는 (최대)  $2^i$  개이다 (루트 노드의 레벨은 0으로 정의)
    - 레벨 i에 있는 노드가 heapify\_down에서 움직이는 레벨 수는 (최대)  $h-i$ 이다.
    - 결국 레벨 i에 있는 노드들이 움직인 총 횟수는 =
 
$$\text{레벨 } i \text{의 노드 수} \times \text{레벨 } i \text{의 노드가 움직이는 레벨 수} = 2^i \times (h-i) = 2^i(h-i)$$
    - 모든 레벨  $i = 0, 1, \dots, h$ 에 대해 횟수를 더한 값이 make\_heap을 위해 이동한 총 횟수  $S$ 이다
 
$$S = \sum_{i=0}^h 2^i \times (h-i)$$
    - 이 식을 전개해보자! (어떻게?) [힌트: 멱급수]
    - 전개하면  $S = O(n)$ 이 된다

- xi. `heap_sort()`: 힙이 아니라면, 우선 `make_heap`으로 힙을 만든 후, `heapify_down` 함수를 반복적용하여 값을 오름차순으로 재배치하는 함수 → 힙 정렬(heap sort) 알고리즘이라 불림

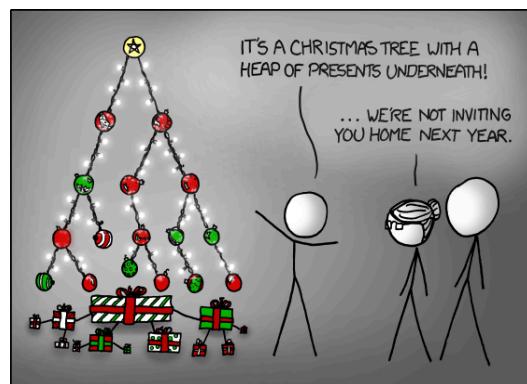
- **[핵심]** 가장 큰 값이 힙의 루트에 저장되어 있다. 이 값은 정렬 순서에 따라 리스트의 가장 마지막에 위치해야 한다
  - $A[0], A[n-1] = A[n-1], A[0]$  (두 값을 교환 - swap)
  - 새로운  $A[0]$ 의 값은 힙 성질을 만족하지 않을 수 있으므로 `heapify_down(n-1, 0)`를 호출해서 성질을 만족하는 자리로 이동
  - 이제 힙은  $(n-1)$ 개의 값( $A[0] \dots A[n-2]$ )으로 구성되므로  $n = n - 1$ 로 변경 후, 위의 두 과정을 다시 반복

```

def heap_sort(self):
 n = len(self.A)
 for k in range(len(self.A)-1, -1, -1):
 self.A[0],self.A[k] = \
 self.A[k],self.A[0]
 n = n - 1 # A[n-1]은 정렬되었으므로
 self.heapify_down(0, n)

```

- heap\_sort의 수행시간은?
  - make\_heap 시간 =  $O(n)$
  - ( $\text{swap} + \text{heapify\_down}$ )을  $(n-1)$ 번 반복하므로  $O(n\log n)$  시간이면 충분!
- 춤으로 표현한 힙 정렬 동영상: <https://youtu.be/Xw2D9aJRBY4>



출처: <https://github.com/aureooms/js-heap>

- [💪] [해보기] 위에서 설명한 Heap 클래스 구현한 후 다음을 수행해보자

```

heap = Heap([2,8,6,1,10,15,3,12,11])
Heap 객체 생성시 이미 make_heap이 호출되어 힙으로 구성 완료됨
print(heap) # [15, 12, 6, 11, 10, 2, 3, 1, 8]
heap.heap_sort()
print(heap) # [1, 2, 3, 6, 8, 10, 11, 12, 15]

```

#### xii. 삽입 연산 `insert(key)`: 기존의 힙에 새로운 key 값을 삽입하는 연산

- 힙 리스트 A의 가장 오른쪽에 새로운 값 x를 저장하고, 이 값이 힙 성질을 만족하도록 위치를 재조정해야 한다
- 이 경우엔 x가 힙의 리프 노드에 위치하므로, 루트 노드 방향으로 올라가면서 자신의 위치를 조정하면 된다 → 위치를 조정해 주는 `heapify_up` 함수를 구현한다

```

def heapify_up(self, k):# 올라가면서 A[k]를 재 배치
 while k>0 and self.A[(k-1)//2] < self.A[k] :
 self.A[k], self.A[(k-1)//2] = \
 self.A[(k-1)//2], self.A[k]
 k = (k-1)//2

def insert(self, key):
 self.A.append(key)
 self.heapify_up(len(self.A)-1)

```

- **수행시간**: 가장 마지막 레벨의 가장 오른쪽 빈 칸에 삽입되어 루트 노드까지 올라갈 수 있으므로 `heapify_up`과 `insert` 모두 힙의 높이만큼의 시간이 필요  
→ **O(logn)**

xiii. 삭제 연산 `delete_max()`: 임의의 값을 삭제하는 것이 아닌 루트 노드에 저장된 **가장 큰 값을 삭제**후 리턴하는 함수

- 루트 노드의 값을 삭제 후 리턴해야 하므로, `A[0]`를 `A[n-1]`의 값으로 대체하고, `heapify_down(0, n-1)`를 호출해 `A[0]`를 재배치한다

```

def delete_max(self):
 if len(self.A) == 0: return None
 key = self.A[0]
 self.A[0], self.A[len(self.A)-1] = \
 self.A[len(self.A)-1], self.A[0]
 self.A.pop() # 실제로 리스트에서 delete!
 heapify_down(0, len(self.A)) # len(A) = n-1
 return key

```

- **수행시간**: `heapify_down`이 1번 호출되고, 이 함수의 수행시간이 전체 시간을 결정하므로 **O(logn)** 시간에 수행

xiv. 연산 및 수행시간 정리 (n개의 값을 저장한 힙에 대해)

- `heapify_up, heapify_down: O(logn)`
- `make_heap = n times x heapify_down = O(nlogn) → O(n)`
- `insert = 1 x heapify_up: O(logn)`
- `delete_max = 1 x heapify_down: O(logn)`
- `heap sort = make_heap + n x delete_max = O(nlogn)`

xv. `heapify_up`과 `heapify_down`은 재귀적으로도 쉽게 작성할 수 있다!

- `def heapify_up(self, k):`  
`p = (k-1) // 2`  
`if k > 0 and self.A[p] < self.A[k] :`  
 `self.A[k], self.A[p] = \`  
 `self.A[p], self.A[k]`  
`self.heapify_up(p)`

- `heapify_down(k): # 각자 해보자!`

xvi. 현재까지 설명한 힙은 루트 노드에 최대 값이 저장되는 힙이었다. 이런 힙을 max-heap이라 부른다. 필요에 따라 min-heap 도 정의할 수 있고, 그에 따른 heapify\_down, heapify\_up 등의 함수를 max-heap과 대칭적으로 작성하여 이용하면 된다

xvii. [Python] heapq 모듈에서 heap 관련 함수를 그대로 제공한다

- 단, 부모노드에 자식노드보다 더 크지 않은 값이 저장되는 min-heap임에 유의하자. (노트에서 다룬 것은 max-heap임!)
- import heapq 필요
- 힙 자체는 리스트를 사용한다: h = []
- 지원연산:
  - heappush(h, key): 힙 h에 key 값을 삽입 (= insert와 동일)
    - heappush(h, (key, value))처럼 튜플 삽입 가능
  - heappop(h): 최소값을 지우고 리턴 (delete\_min의 역할)
  - heapify(A): 리스트 A를 힙 성질이 만족되도록 변환
    - make\_heap()과 동일 (단, min-heap으로 변환)
  - h[0]: 힙의 최소값을 알고 싶다면

## 5. 적응형 힙 (Adaptive Heap)

- a. 앞에서 살펴본 힙 제공 연산은 `delete_max`, `insert`, `heap_sort` 등이다
- b. 예1: 항공사에서 특정 승객의 좌석을 취소하고 싶다면 어떻게 할까? 승객을 key로 하는 힙에서는 특정 승객의 정보를 빠르게 찾을 수 없다. 애초에 빠른 탐색을 위해 설계된 자료구조가 아니라, 가장 크거나 작은 key 값을 빨리 찾을 수 있도록 설계된 자료구조이기 때문이다
- c. 예2: 어떤 승객이 갑자기 VIP 카드로 더 좋은 좌석을 요구한다면, 그 승객의 key 값 (일종의 priority 값)을 증가시켜야 한다. 반대로 key 값을 감소시켜야 할 수도 있다
- d. 새로운 두 연산이 필요하다:
  - i. `remove(key)`: key 값을 힙으로부터 제거
  - ii. `update_key(old_key, new_key)`: old\_key 값을 new\_key 값으로 대체
    - old\_key < new\_key 라면 `heapify_down` 호출 필요
    - old\_key > new\_key 라면 `heapify_up` 호출 필요
- e. 문제는 `remove(key)`를 수행하기 위해선, 힙에서 key 값이 저장된 위치 (`index`)를 알아야 한다. `update_key(old_key, new_key)`에 대해선, old\_key 값이 저장된 힙의 `index`를 알아야 한다. **어떻게?**
- f. 해결책: 각 key 값이 저장된 위치 (`index`)를 기억하고 있도록 함! dict 자료구조 (또는 해시테이블)에 (`key, index`)를 저장한다
- g. `AdaptedHeap` 클래스:
  - i. Locator 객체가 저장되는 힙 리스트 A가 필요하고, 각 key 값이 담긴 Locator 객체를 연결하는 사전(dict) D도 필요하다

```
class AdaptedHeap:
 def __init__(self):
 self.A = [] # 여기선 빈 리스트로 초기화
 self.D = {} # dict: D[key] = index

 def __str__(self):
 return str(self.A)

 def __len__(self):
 return len(self.A)

 def find_index(self, key): # return index that A[index] == key
 return self.D.get(key) # None if key is not in D

 def insert(self, key, value=None):
 self.D[key] = len(A) # D[key] = len(A)
 self.A.append(key) # 힙에 key 삽입
 self.heapify_up(self.D[key]) # heapify_up
 return self.D[key]
```

```

def heapify_up(self, k): # 인덱스 k에 저장된 item을 up!
 p = (k - 1)//2 # p = parent index of A[k]

 while k > 0 and self.A[p] < self.A[k]:
 # index swap
 self.D[self.A[k]] = p
 self.D[self.A[p]] = k

 # key swap!
 self.A[k], self.A[p] = self.A[p], self.A[k]

 k = p
 p = (k - 1)//2

def heapify_down(self, k):
 n = len(self)
 while 2*k+1 < n:
 L, R = 2*k+1, 2*k+2
 if self.A[L] < self.A[k]:
 m = L
 else:
 m = k
 if R < n and self.A[R] < self.A[m]:
 m = R
 # A[m] is the minimum among A[k], A[L], and A[R]
 if m != k: # swap loc and index
 self.A[k], self.A[m] = self.A[m], self.A[k]
 self.D[self.A[k]] = k
 self.D[self.A[m]] = m
 k = m
 else:
 break

def remove(self, key):
 k = self.find_index(key)

 if not 0 <= k < len(self): # key가 힙에 없다면
 raise ValueError('Invalid index')

 # 1. A[k]와 A[-1]을 swap!
 self.A[k], self.A[-1] = self.A[-1], self.A[k]

 # 2. A[k]와 A[-1]의 인덱스 수정
 self.D[self.A[k]], self.D[self.A[-1]] = k, len(self)-1

```

```
3. A[-1] 제거
self.A.pop()

4. A[k]를 heapify_down해서 재 배치
self.heapify_down(k)
del self.D[key]
```

- h. find\_max와 delete\_max (또는 delete\_min) 함수를 작성해보자: remove 함수 이용

```
def find_max(self): # max key의 value도 함께 리턴해도 됨 (여기선 생략)
 max_key = None
 if len(self) > 0:
 max_key = self.A[0]
 return max_key

def delete_max(self):
 max_key = None
 if len(self) > 0:
 max_key = self.A[0]
 self.remove(max_key)
 return max_key
```

- i. update\_key 함수를 작성해보자

```
def update_key(self, old_key, new_key):
 k = self.A[old_key]

 if not 0 <= k < len(self):
 raise ValueError('Invalid locator')

 self.A[k] = new_key

 # update D
 self.D[new_key] = k
 del self.D[old_key]

 if old_key > new_key: # new_key가 더 작기 때문에 down!
 self.heapify_down(k)
 if old_key < new_key: # new_key가 더 크기 때문에 up!
 self.heapify_up(k)

 return self.D[new_key]
```

- j. remove, delete\_max/delete\_min, update\_key의 수행시간은 모두 **O(logn)**이다.  
이유는 쉽게 알 수 있다

## 6. 우선순위 큐 (Priority Queue)

- a. `delete_max` (또는 `delete_min`), `find_max` (또는 `find_min`), `insert`, `update_key` (optional) 연산을  $O(\log n)$  시간 이내에 제공하는 자료구조를 통칭해 **우선순위 큐**라고 부른다
  - i. 앞에서 설명한 힙 (binary heap이라 불림)은 가장 일반적인 우선순위 큐이다
  - ii. priority (우선순위) 값에 따라 데이터를 관리할 필요가 있는 경우에 필수적인 자료구조
  - iii. 그래프의 최소신장트리 (Minimum Spanning Tree)를 구하는 Prim 알고리즘이나 최단경로를 구하는 Dijkstra 알고리즘 등에 사용
- b. 대표적인 우선순위 큐
  - i. `stack`, `queue`, `dequeue` (삽입 시간을 일종의 priority로 간주)
  - ii. `heap`, `adaptive heap`: `max-heap`, `min-heap`
  - iii. 아래 표는 wikipedia에서 정리한 여러 종류의 우선순위 큐
    - Binary heap이 앞에서 설명한 heap이다
    - Fibonacci heap도 관심을 갖으면 좋다

| Operation                        | <code>find-min</code> | <code>delete-min</code> | <code>insert</code> | <code>decrease-key</code><br>( <code>update_key</code> ) | <code>meld</code> |
|----------------------------------|-----------------------|-------------------------|---------------------|----------------------------------------------------------|-------------------|
| Binary heap <sup>[5]</sup>       | $\Theta(1)$           | $\Theta(\log n)$        | $O(\log n)$         | $O(\log n)$                                              | $\Theta(n)$       |
| Leftist                          | $\Theta(1)$           | $\Theta(\log n)$        | $\Theta(\log n)$    | $O(\log n)$                                              | $\Theta(\log n)$  |
| Binomial <sup>[5][6]</sup>       | $\Theta(1)$           | $\Theta(\log n)$        | $\Theta(1)^{[a]}$   | $O(\log n)$                                              | $O(\log n)^{[b]}$ |
| Fibonacci <sup>[5][7]</sup>      | $\Theta(1)$           | $O(\log n)^{[a]}$       | $\Theta(1)$         | $\Theta(1)^{[a]}$                                        | $\Theta(1)$       |
| Pairing <sup>[8]</sup>           | $\Theta(1)$           | $O(\log n)^{[a]}$       | $\Theta(1)$         | $O(\log n)^{[a][c]}$                                     | $\Theta(1)$       |
| Brodal <sup>[11][d]</sup>        | $\Theta(1)$           | $O(\log n)$             | $\Theta(1)$         | $\Theta(1)$                                              | $\Theta(1)$       |
| Rank-pairing <sup>[13]</sup>     | $\Theta(1)$           | $O(\log n)^{[a]}$       | $\Theta(1)$         | $\Theta(1)^{[a]}$                                        | $\Theta(1)$       |
| Strict Fibonacci <sup>[14]</sup> | $\Theta(1)$           | $O(\log n)$             | $\Theta(1)$         | $\Theta(1)$                                              | $\Theta(1)$       |
| 2-3 heap <sup>[15]</sup>         | $O(\log n)$           | $O(\log n)^{[a]}$       | $O(\log n)^{[a]}$   | $\Theta(1)$                                              | ?                 |

## 7. [고급] 힙 모양 이진트리 1: 범위 트리 (Segment Tree)

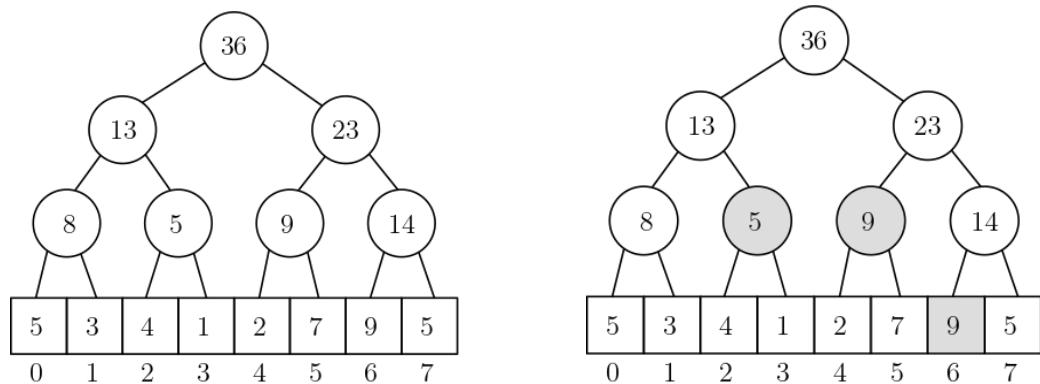
- a. 아래 그림처럼 8개의 수가 리스트에 저장되어 있다고 하자. 인덱스 범위 [2, 6]이 주어지고 이 인덱스 범위에 있는 값의 합  $\text{sum}(2, 6) = A[2] + \dots + A[6] = 23$ 을 알고 싶다고 하자
- b. 가장 쉬운 방법은 반복문을 통해  $A[2]$ 부터  $A[6]$ 까지 더하면 된다. 그러나 이 방법은 최악의 경우에  $O(n)$  시간이 필요하다. 만약, 이런 범위 질의 (query)를 여러 번 처리해야 한다면 이 방법은 너무 비효율적이다

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 1 | 2 | 7 | 9 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 4 | 1 | 2 | 7 | 9 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

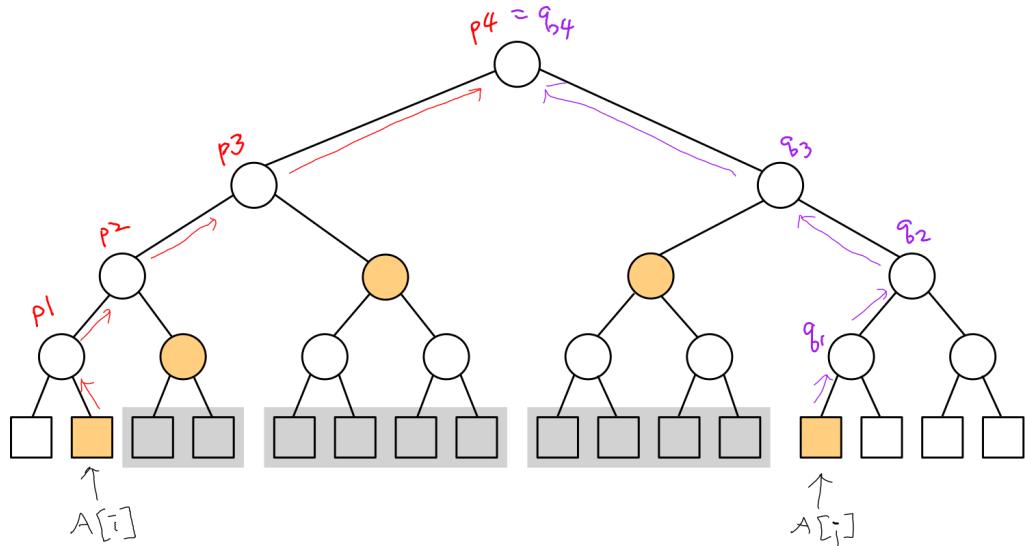
- c. 여러 개의 질의를 처리한다는 가정하에 리스트의 값을 미리 전처리 (pre-processing) 과정을 통해 질의에 대한 답을 빠르게 찾을 수 있도록 특별한 자료구조에 저장하면 어떨까?
- d. 아래 왼쪽 그림처럼 힙 모양으로 트리를 구성해보자
  - i. 모양이 힙과 같지 key 값이 저장된 규칙은 힙을 따르지 않음에 유의
  - ii. 리스트의 노드는 리프노드가 되고 내부 노드는 자신의 **리프노드들의 합**을 key 값으로 저장한다. 그러면 루트에는 모든 값의 합이 저장된다
  - iii. 그러면  $\text{sum}(2, 6)$ 은 아래 오른쪽 그림처럼 회색의 세 노드의 값을 더하면 된다. 5의 값을 갖는 노드는  $A[2]+A[3]$ 을 나타내고, 9 값을 갖는 노드는  $A[4]+A[5]$ 를 나타내기에 **5개의 값의 합을 3개의 값의 합**으로 표현할 수 있다



- e. 그러면 임의의 구간  $[i, j]$ 에 대해서  $A[i] + \dots + A[j]$ 를 위해 몇 개의 노드의 합으로 표현할 수 있을까?
  - i.  $\text{sum}(0, 7) = (36)$  # 루트 노드 1개로 표현 가능
  - ii.  $\text{sum}(1, 6) = A[1] + (5) + (9) + A[6]$  # 4개의 노드로 표현 가능
  - iii.  $\text{sum}(4, 7) = (23)$  # 노드 1개로 표현 가능
  - iv.  $\text{sum}(i, j) = A$ 에  $n$ 개의 값이 있다면 최대  $2\log n + 2$ 개의 노드로 표현 가능!

v. 아래 그림을 보면,  $A[i] + \dots + A[j]$  값을 계산하기 위해서 12개의 A의 값을 차례대로 더하지 않고 5개의 노란색 노드에 저장된 값만 더하면 된다

vi.  $A[i], A[j]$ 가 노란색 노드이고 내부노드 세 개가 노란색 노드에 해당된다. 내부노드는 리프노드 2개, 4개, 4개의 합이 각각 저장되어 있기에 노란색 노드의 값만 더하면 질의의 답을 계산할 수 있다



vii. 노란색 노드는 어떻게 찾을 수 있을까?

- $A[i]$ 와  $A[j]$ 에서 루트노드를 향해 동시에 올라가보자. 동시에 한 레벨씩 올라가기 때문에 언제가는 같은 노드 ( $p_4 = q_4$ )에 도착하게 된다 (이 노드를  $A[i]$ 와  $A[j]$ 의 Lowest Common Ancestor (LCA) 노드라 부른다)
- $A[i] \rightarrow p_1 \rightarrow p_2 \rightarrow p_3$ 에서 각 노드의 오른쪽 자식 노드가 노란색 노드가 되고,  $A[j] \rightarrow q_1 \rightarrow q_2 \rightarrow q_3$ 에서 각 노드의 왼쪽 자식 노드가 노란색 노드가 된다

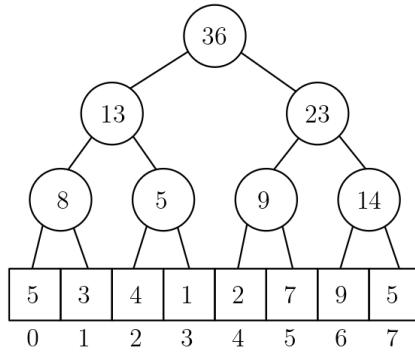
viii. 노란색 노드는 몇 개일까?

- $A[i]$ 와  $A[j]$ 에서 한 레벨씩 올라가면서 각각 레벨당 최대 하나의 노드가 노란색 노드가 되기에 레벨당 최대 2개씩의 노란색 노드가 존재한다. 따라서, 트리 높이의 2배를 넘지 않게 된다
- 트리를 힙 모양의 이진트리로 관리하기 때문에 높이가  $O(\log n)$ 이므로 노란색 노드는  $O(1 \log n)$ 개이다

ix. 힙 모양 형태의 트리 tree에 저장한다면 어떻게 하면 될까?

- A에 저장된 n개의 값이 힙의 리프노드가 된다. 힙의 내부노드는 정확히  $n-1$ 개가 되어야 한다. 그래서  $2n-1$ 개의 크기의 리스트에 저장하면 된다

- 예:



|    |    |    |   |   |   |    |   |   |   |    |    |    |    |    |
|----|----|----|---|---|---|----|---|---|---|----|----|----|----|----|
| 36 | 13 | 23 | 8 | 5 | 9 | 14 | 5 | 3 | 4 | 1  | 2  | 7  | 9  | 5  |
| 0  | 1  | 2  | 3 | 4 | 5 | 6  | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

- f. 질의 구간  $Q = [i, j]$ 가 주어지면  $\text{sum}(i, j)$ 를 이 트리를 이용해 계산해보자
- A[i]에서 부모노드로 올라가면서 오른쪽 자식노드의 값을 계속 더하고, A[j]에서 부모노드로 올라가면서 왼쪽 자식노드의 값을 더한다

```

def sum(A, tree, i, j):
 # n = len(A)
 i += n-1 # tree[i+n-1]이 A[i]에 해당
 j += n-1 # tree[j+n-1]이 A[j]에 해당
 s = tree[i] + tree[j]
 while parent(i) < parent(j): # (*)
 # LCA 노드에 도착할 때까지 반복
 i = parent(i) # parent(i) = (i-1)//2
 j = parent(j) # parent(j) = (j-1)//2
 if tree[i] is left child:
 s += tree[i+1] # 오른쪽 자식 값 더함
 if tree[j] is right child:
 s += tree[j-1] # 왼쪽 자식 값 더함
 return s

```

- 문제는 입력 리스트 A의 크기가  $2^k$ 의 형태면 입력 값이 힙 구조 트리의 가장 아래 레벨에 놓이게 되어 (즉, A[i]와 A[j] 모두 가장 아래 레벨에 위치해, while 루프를 반복할 때마다 한 레벨씩 동시에 올라가  $\text{parent}(i) == \text{parent}(j)$  인 경우에 LCA에 도달했음을 알 수 있어 while 루프 (\*)를 탈출할 수 있지만, 그렇지 않은 경우엔 올바르게 작동하지 않을 수 있다)
  - 만약  $2^{k-1} < n < 2^k$ 라면, A에  $2^k - n$ 개의 원소를 0으로 채우면 된다 (padding 단계). 이렇게 되더라도 수행시간의 Big-O 값은 변하지 않는다. 만약, padding을 하지 않고 싶다면, 리프노드가 마지막 두 레벨에 걸쳐 존재할 수 있기 때문에, while 문의 탈출조건을 적절히 수정해야 한다
- g. 만약 A[k]의 값을 수정해야 하는 경우라면? 다른 노드 값들 역시 부분적으로 수정되어야 한다

- i.  $A[k]$ 부터 루트 노드를 향해 올라가면서 방문하는 노드의  $sum$  값만 갱신하면 된다. 따라서  $O(\log n)$  시간에 가능

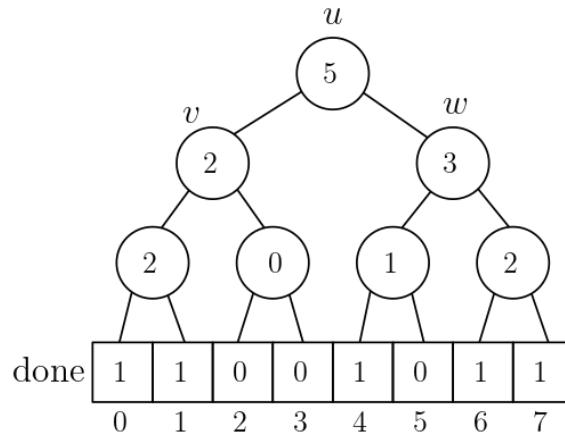
```
def update(A, tree, k, x): # A[k]의 값을 x로 update
 # A와 tree를 따로 관리한다면, A[k] = x로 update
 k += n-1 # n = len(A)
 tree[k] = x # update t[k] with value x
 while k >= 0:
 k = parent(k) # parent(k) = (k-1)//2
 tree[k] = tree[2*k+1] + tree[2*k+2]
```

- h. 트리의 내부 노드에 합을 저장하는 대신 최대값, 최소값 등을 저장해도 된다. 그러면  $A[i], \dots, A[j]$  중에서 최대값을 구하거나 최소값을 구하는 질의를 같은 방식으로 처리할 수 있다

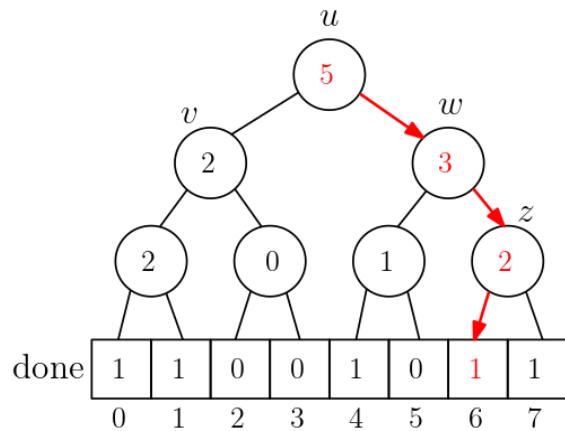
## 8. 응용 예: 순열복원 문제

- a. 0부터  $n-1$ 까지 서로 다른 수로 구성된 **순열** (permutation)  $A$ 에 대해, 새로운 리스트  $S$ 가 주어졌다
- i.  $S[i]$ 는  $A[0]$ 부터  $A[i-1]$ 까지 수 중에서  $A[i]$ 보다 작은 수의 개수 (단,  $S[0] = 0$ 으로 정의한다) 즉,  $A[i]$ 의 왼쪽에 있는 값 중에서  $A[i]$ 보다 작은 값의 개수를  $S[i]$ 에 계산해 저장했다
- b. **순열복원 문제**: 실수로 순열 리스트  $A$ 를 잃어버려서  $S$ 만 가지고 있다.  $S$ 를 이용해 순열  $A$ 를 재구성하라
- i. **입력**: 첫 줄에 리스트  $S$ 의  $n$ 개의 값 ( $n$ 의 값은 1 이상 1000이하)
  - ii. **출력**: `print(A)` #  $A$ 는 0부터  $n-1$ 까지의 순열임에 유의!
- c.  $A[i]$ 의 왼쪽에 있는 값 중에서 작은 값의 개수가  $S[i]$ 개라서, 오른쪽에 있는 값 중  $A[i]$ 보다 작은 값의 개수  $L[i]$ 를 알면  $A[i]$ 의 등수는  $S[i]+L[i]+1$ 가 된다. 0부터 시작하므로  $A[i] = S[i]+L[i]$ 이 된다. 그런데  $S[i]$ 만 알고  $L[i]$ 는 모른다는 것이 문제임
- d. 예:  $S = [0, 0, 2, 2, 3, 2, 4, 2]$ ,  $n = 8$
- i.  $S[n-1]=S[7]=2$ 라는 의미는  $A[7]$ 의 왼쪽에 작은 값이 2개 있다는 의미이고,  $A[7]$ 이 가장 오른쪽에 있는 값이라 결국  $L[i] = 0$ 이 되어  $A[7]$  등수 =  $S[i] + L[i] = 2$ 가 된다. 따라서  $A[7]$ 의 값은 쉽게 결정 가능!
  - ii. 새로운 리스트  $done$ 을 정의하고 0으로 초기화 한다. 이  $done$ 에는 현재까지 복원한  $A$ 의 값을 1로 표시해 마크한다. 2가 복원되었으므로  $done[2] = 1$ 로 변경함
  - iii.  $S[6] = 4$ 에 대해,  $A[6]$ 의 오른쪽에 있는  $A[6]$ 보다 작은 값의 개수는  $done$ 의 기록을 보고 유추할 수 있다. 현재는  $done[2]=1$ 이므로  $A[6]$ 의 오른쪽에 2가 있었다는 의미이다.  $L[6]=1$ 이다. 따라서  $A[6] = S[6]+L[6] = 4+1 = 5$ 가 된다.  $done[5] = 1$ 로 마킹한다

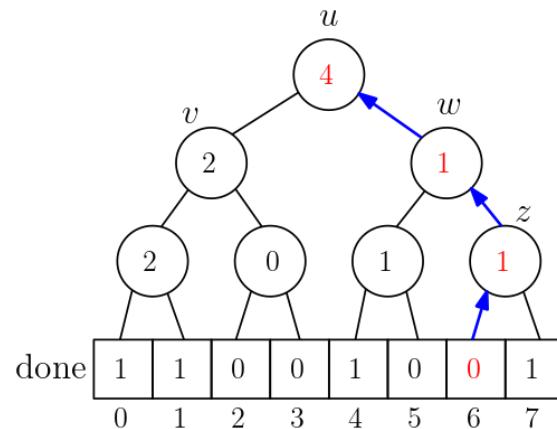
- iv. 현재  $\text{done} = [0, 0, 1, 0, 0, 1, 0, 0]$
  - v.  $S[5] = 2$ 이고,  $\text{done}[0..2]$ 까지의 1의 개수가 1이므로  $A[5] = 2+1 = 30$  되고,  $\text{done}=[0, 0, 1, \textcolor{red}{1}, 0, 1, 0, 0]$ 이 됨
  - vi.  $S[4] = 3$ 이고,  $\text{done}[0..3]$ 까지의 1의 개수가 두 개고  $A[4] = 3+2 = 50$  가능하지만,  $\text{done}[5] = 1$ 이라서 5가 이미 등장한 수임. 따라서 순위가 하나 더 밀려  $A[4] = 6$ 이 됨. 즉,  $A[4]$  오른쪽에 등장하는  $A[4]$ 보다 작은 수의 개수가 3개임을 알 수 있다. 이 개수는  $\text{done}$  리스트의 값을 prefix sum 계산처럼 하나씩 더하면서 결정할 수 있다.
  - vii. 이 과정을 반복하면  $A[n-1], A[n-2], \dots, A[0]$  순서로 결정할 수 있고, 하나를 결정할 때, prefix sum 값을 계산해야 하므로  $O(n)$  시간이 필요해 전체적으로는  $O(n^2)$  시간이면 충분하다
  - viii. 범위 트리를 사용하면 더 빠르게 할 수 있다
- e. 범위 트리를 사용해서  $L[i]$ 를  $O(\log n)$  시간에 찾아보자. 그러면 전체 수행시간은  $O(n \log n)$ 이면 충분하다
- i. 여기서는 앞에서 했던 것과 **반대로**  $\text{done}$ 의 값을 정의한다. 즉,  $\text{done}$ 은 모두 1로 초기화하고,  $A[i]$ 의 값이  $k$ 로 결정되면,  $\text{done}[k] = 0$ 으로 마크한다. 이렇게 반대로 하면 어떤 점이 좋을까? (이유는 자연스럽게 알수 있다 ^^)
  - ii. 현재  $A[4]$ 를 결정하는 순간이라고 하자. 그러면  $A[7], A[6], A[5]$ 의 값이 각각 2, 5, 3으로 이미 결정되었다. 따라서,  $\text{done}=[1, 1, 0, 0, 1, 0, 1, 1]$ 가 된다
  - iii.  $S[4] = 3$ 이므로 자기보다 작은 값이 왼쪽에 3개 존재한다. 이것은 왼쪽에 오는 수가 세 개이므로  $A[4]$ 는 네 번째 위치에 와야 한다는 것이다. 그러면  $\text{done}=[1, 1, 0, 0, 1, 0, 1, 1]$ 에서 왼쪽에서부터 **네 번째 1의 index** 값이  $A[4]$ 가 된다. 왜? 여기서 1은 아직 결정되지 않은 값을 나타내는 것이고 0은  $A[4]$ 의 오른쪽에 오는  $A[4]$ 보다 작은 값으로 이미 계산이 되어 해당 index에 0으로 마크되었기에 1의 개수만 세어 **네 번째 1의 위치**를 찾으면 되기 때문이다. 따라서  $\text{done}=[\textcolor{red}{1}, \textcolor{red}{1}, 0, 0, \textcolor{red}{1}, 0, \textcolor{brown}{1}, 1]$ 에서 네 번째 1의 index는 6이 되어,  $A[4] = 6$ 임을 알 수 있다. 그리고  $\text{done}[6] = 0$ 으로 지정한다
  - iv. 결국,  $S[i] = k$ 라면  $\text{done}$ 의 prefix sum이  $k+1$ 이 되는 위치의 index 값이  $A[i]$ 가 된다
  - v.  $\text{done}$ 을 범위 트리로 표현하면 아래 그림과 같다
    - 내부노드에는 자신의 **리프노드들에 저장된 1의 개수**가 저장



- vi. 루트 노드  $u$ 부터 탐색을 시작하는 데, 우리는 왼쪽부터  $s[4]+1 = 4$ 번째 1의 위치 (즉, 4번째 1이 저장된 리프 노드)를 찾는 것이 목표다
- vii. 루트 노드의 왼쪽 자식노드  $v$ 의 값 2라는 의미는  $v$ 의 리프 노드에는 1이 두 개가 있다는 뜻이므로 4번째 1은  $v$ 의 오른쪽 부트리에 있음을 알 수 있다. 따라서 오른쪽 자식 노드  $w$ 를 따라 내려가 자신에 해당하는 리프 노드 위치를 찾아야 한다
- viii. 이 때, 오른쪽 자식노드로 내려가면서 찾아야 할 1의 위치는  $v$ 의 리프 노드에 있는 1의 개수를 제외한  $4-2 = 2$  번째 1을 찾아야 한다
- ix. 오른쪽 자식 노드  $w$ 로 내려와서, 다시  $w$ 의 왼쪽 자식노드를 살펴본다. 왼쪽 서브트리에 있는 1의 개수가 하나 뿐이므로 다시  $w$ 의 오른쪽 자식 노드  $z$ 로 내려가야 한다. 이런 식으로 내려가면 결국 리프 노드  $done[6]$ 에 도착하게 된다. 그래서  $A[4] = 6$ 임을 알 수 있다 (아래 그림 참조)
- x. 한 마디로 요약해 말하면,  $done$ 의 prefix sum이 6이 되는 위치의 인덱스 값을 찾는 것과 같다! (이 말은 임의의 구간 합을 계산하는 범위 트리보다는 prefix 구간 합을 계산하는 Fenwick 트리가 더 알맞은 자료구조이다. 이 트리는 바로 다음에 설명!)



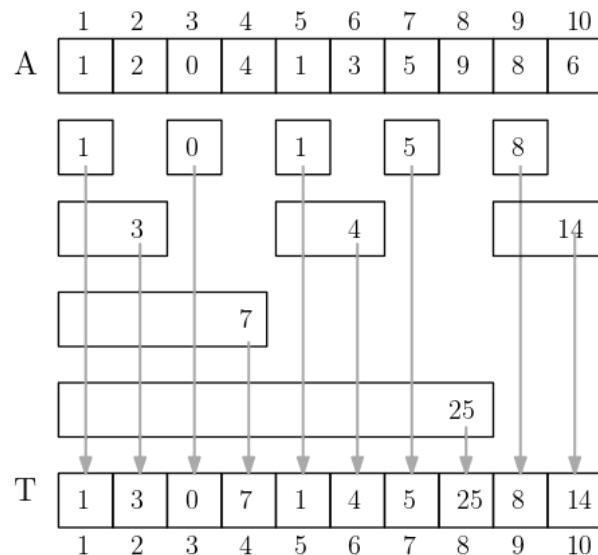
- xi. 이제  $done[6] = 0$ 으로 지정하고 범위 트리의 내부 노드의 값을 이에 맞게 업데이트하면 된다. 이 업데이트는  $done[6]$ 부터 루트 노드로 올라가면서 아래 그림처럼 1씩 줄이기만 하면 된다



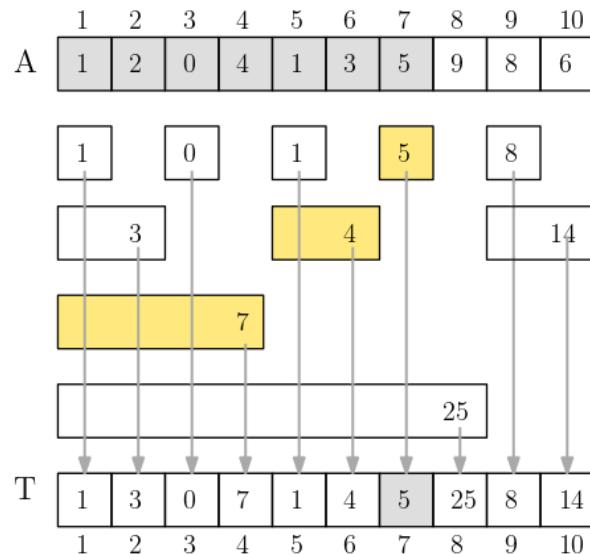
- xii.  $A[i]$ 를 결정하는 건 루트 노드에서 리프 노드로 탐색하는 과정이므로  $O(\log n)$  시간이면 충분하고,  $done$ 의 값을 업데이트하는 건 거꾸로 리프노드에서 루트노드로 올라가면서 수정하는 것이므로 역시  $O(\log n)$  시간이면 된다. 총  $O(\log n)$  시간에 계산 가능하므로 전체적으로는  $O(n \log n)$  시간에 순열을 복원할 수 있다

## 9. [고급] 힙 모양의 이진트리 2: Binary Indexed Tree (BIT)

- a. Fenwick tree 라고도 불림 [https://en.wikipedia.org/wiki/Fenwick\\_tree](https://en.wikipedia.org/wiki/Fenwick_tree)
- b. 리스트 A에서  $\text{prefix\_sum}(k) = A[1] + \dots + A[k]$ 로 정의된다
- c. n개의 값이 저장된 리스트 A를 전처리해 BIT 트리를 구성해 놓으면  $\text{prefix\_sum}(k)$  질의와  $\text{update}(k, x)$  연산을  $O(\log n)$  시간에 답할 수 있다
  - i.  $\text{prefix\_sum}(k)$  연산:  $A[1] + \dots + A[k]$ 을  $O(\log n)$  시간에 리턴
  - ii.  $\text{update}(k, x)$  연산:  $A[k]$ 의 값을  $x$ 로  $O(\log n)$  시간에 업데이트
- d. [주의]  $\text{prefix\_sum}(k)$  질의만 관심이 있다면, BIT 자료구조를 사용하지 않고, 모든 prefix sum을  $O(n)$  시간에 미리 구해 별도의 리스트에 저장해 놓은 후, 매 질의에 대해  $O(1)$  시간에 답을 할 수 있다.  $\text{update}$ 와  $\text{prefix\_sum}$  두 연산을 빠르게 지원하기 위해 특별한 형태의 자료구조가 필요한데, 그 자료구조가 BIT, Fenwick 트리 구조이다
- e.  $A = [\text{None}, 1, 2, 0, 4, 1, 3, 5, 9, 8, 6]$ 
  - i.  $A[0]$ 는 사용하지 않고  $A[1]$ 부터 사용한다고 하자
    - 설명이 간단해지는 장점.  $A[0]$ 부터 시작하는 경우로 쉽게 변경 가능
  - ii.  $k = 6$ 인 경우,  $\text{prefix\_sum}(k) = A[1] + \dots + A[6]$ 이 된다. 이 합을  $(A[1]+A[2]+A[3]+A[4])+(A[5]+A[6])$ 로 두 개의 작은 합으로 나누어 표현할 수 있다
  - iii. 6을 이진수로 표현하면 110이고,  $110 = 100 + 010 = 4 + 2$ 가 되므로 4개의 합과 2개의 합으로 분할할 수 있음을 알 수 있다
  - iv. 7의 경우에는 111이므로  $111 = 100 + 010 + 001 = 4 + 2 + 1$ 로 세 합의 합으로 표현할 수 있다
  - v. 45의 경우에는  $45 = 101101$ 이라면,  $100000 + 001000 + 000100 + 000001$ 으로 네 부분으로 나눌 수 있다. 즉,  $32+8+4+1 = 45$ 이 되어,  $(A[1]+\dots+A[32]) + (A[33]+\dots+A[40]) + (A[41]+\dots+A[44]) + (A[45])$ 으로 처음 32개의 합 + 8개의 합 + 4개의 합 + 1개의 합을 하면 원하는 prefix sum을 구할 수 있다
    - IDEA: BIT 트리 T를 리스트로 저장하는데,  $T[45]$ 에는  $A[45]$  값을 저장하고,  $T[44]$ 에는  $(A[41]+\dots+A[44])$ 을 저장하고,  $T[40]$ 에는  $(A[33]+\dots+A[40])$ 을 저장하고,  $T[32]$ 에는  $(A[1]+\dots+A[32])$ 를 저장한다. 그러면 k를 이진수로 표현해서  $T[45]+T[44]+T[40]+T[32]$ 를 계산한다 (그러면 4개의 T 값만 더하면 되므로 매우 빠르다)
  - vi. 이 아이디어를 이용해서 아래 그림과 같이 BIT 트리를 계산해 리스트 T에 저장한다 (트리가 리스트로 표현됨에 주의)



- $T[k] = k$ 의 이진수에서 1이 처음으로 등장하는 LSB(Least Significant Bit)의 값의 개수만큼 A의 값을 더해 저장한다. 즉,  $T[k] = A[k-LSB(k)+1] + \dots + A[k]$ 로 저장한다
- $T[1] = 1$ 의 이진수는 1이므로 오른쪽 첫 비트 1은 0번째 위치에 존재하므로 LSB의 값이  $2^0$ 이 되어 A[1] 값 하나만으로 구성된다. 즉,  $T[1] = A[1] = 1$
- $T[2] = 2$ 의 이진수는 10이므로 오른쪽에서 첫 1 비트가 1번째 위치에 등장하므로 LSB의 값이  $2^1 = 2$ 가 되어 2개의 값을 더한다. 즉,  $T[2] = A[1] + A[2] = 1+2 = 3$
- $T[4] = 4$ 의 이진수는 100이므로 LSB의 값이  $2^2$ 이므로 4개의 값을 더해서 구성.  $T[4] = 1+2+0+4 = 7$
- $T[8] = 8$ 의 이진수는 1000이므로 LSB의 값이  $2^3$ 이므로 8개의 값을 더해서 구성.  $T[8] = A[1] + \dots + A[8] = 25$



- vii.  $k = 7$ 인 경우에는 위의 그림처럼  $\text{prefix\_sum}(7) = T[7] + T[6] + T[4] = 5 + 4 + 7 = 16$ 으로 계산하면 된다

```
prefix_sum(7) = T[111] + T[111-001] + T[110-010]
 = T[111] + T[111-LSB(111)] + T[110-LSB(110)]
```

```
def prefix_sum(k):
 s = 0
 while k >= 1:
 s += T[k]
 k = k - LSB(k)
 return s
```

- viii.  $\text{LSB}(k)$  계산:  $k$ 의 오른쪽에서 첫 번째 1의 비트 위치가  $d$ 번째라면  $2^d$

```
def LSB(k):
 return k & -k
```

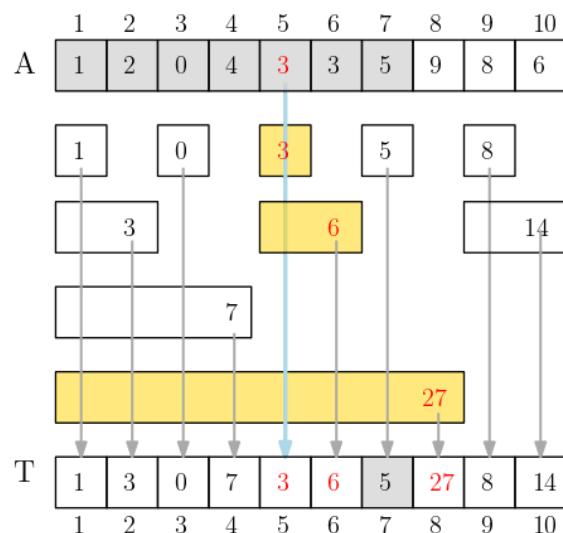
- $-k$ 는  $k$ 의 2의 보수임
- $k$ 와  $k$ 의 2의 보수를 비트 AND를 하면 어떤 일이 벌어지나? 이렇게 하면  $\text{LSB}(k)$ 가 계산되는 이유는?

- ix.  $\text{prefix\_sum}$ 의 수행시간:  $O(\log n)$

- 길이가  $1, 2, 2^2, 2^3, \dots$  인 구간들의 합으로  $\text{prefix\_sum}$ 을 계산하게 된다. 당연히 이러한 구간은 최대  $\log n$ 개 이므로  $O(\log n)$  시간이면 충분하다

- x.  $A[k]$ 의 값을 수정(update)하면  $T$ 의 몇 개의 값도 적절히 수정되어야 한다

- 예를 들어,  $A[5]$ 의 값을 현재의  $1 \rightarrow 3$ 으로 수정했다고 하자. 그러면  $T[5], T[6], T[8]$ 의 값이 영향을 받기 때문에 이 세 값을 바꿔야 한다



- 영향 받는  $T$ 의 값들을 어떻게 결정하면 될까?  $A[5]$ 를 포함하는  $T[i]$ 를 결정하면 된다: 당연히  $i > k$ 어야 한다.  $5 = 101_2$ 이므로  $T[5] (=$

$A[5]$ )는 당연히 수정되어야 함.  $101 + \text{LSB}(5) = 101+001 = 110 = 6$  이므로  $\text{LSB}(6) = 2$ 가 되어 2개의 값  $T[6] = A[5]+A[6]$ 이 되어  $A[5]$ 를 포함하게 된다. 따라서 수정되어야 한다.  $110 + \text{LSB}(6) = 110+010 = 1000 = 8$ 이고,  $T[8] = A[1]+\dots+A[8]$ 이 되어  $A[5]$ 를 포함한다. 따라서 수정되어야 한다

- 이 과정을 한 번 더 하면 10000이 되어 n보다 커지게 되면 더 이상 할 필요가 없게 된다
- 규칙을 정리하면 아래와 같다

```
def update(k, x): # A[k] ← x means A[k] += x-A[k]
 d = x - A[k]
 while k <= n:
 T[k] = d
 k = k + LSB(k)
```

- 그럼  $A[5]$ 를 포함하는  $T[i]$ 가 몇 개인가? 결국 비트당 하나씩 존재하므로  $\log n$ 개를 넘지 않는다
- 따라서 update의 수행시간:  $O(\log n)$  ( $\leftarrow$  prefix\_sum의 수행시간 분석과 동일)

xi. BIT 트리를 구성하는데 필요한 시간은?  $O(n)$

- 어떻게 하면 선형시간에 구성할 수 있을까?

xii. BIT 트리를 이용한 예:

- prefix 구간을 질의로 주고, 해당 구간에 대한 합, 최소, 최대값 등을  $O(\log n)$  시간에 답할 수 있다
- 인덱스 구간  $[i, j]$ 를 주고, 구간 합  $A[i]+\dots+A[j]$ 를  $O(\log n)$  시간에 계산할 수 있다 (범위 트리를 이용한 경우와 같은 시간)

$A[i]+\dots+A[j] = \text{prefix\_sum}(j) - \text{prefix\_sum}(i-1)$  이므로 가능하다

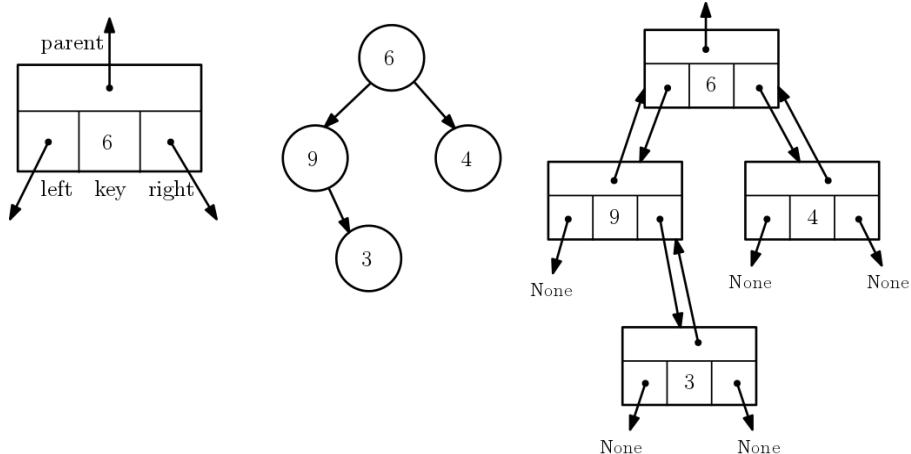
- 당연히, 순열복원 문제를 범위 트리 대신 BIT 트리를 이용해서 해결할 수 있다

## 10. 이진트리 (Binary Tree)

- a. 직접 표현법: 노드 클래스를 선언하여 모양대로 표현하는 가장 일반적인 방법
- i. key 값 (+ 필요하면 추가로 정보를 저장할 수 있는 다른 멤버 선언)
  - ii. left, right, parent 노드를 가리키는 멤버

```
class Node:
 def __init__(self, key=None, parent=None, left=None, right=None):
 self.key = key
 # 필요하면 추가 정보 - 예: self.value = value # 부가 정보
 # 필요하면 추가 정보 - 예: self.height = 0 # 노드의 높이
 self.parent = parent
 self.left = left
 self.right = right

 def __str__(self):
 return str(self.key)
```

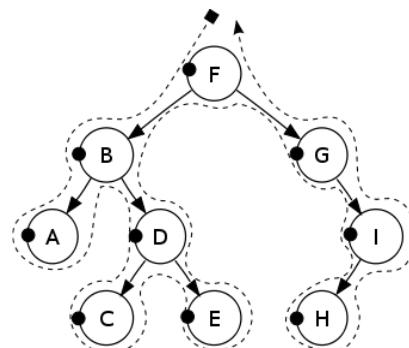


- iii. 위 그림의 가장 왼쪽은 Node 클래스의 멤버들을 도형으로 표현한 것이고, 가운데 트리를 실제 Node를 이용해 연결한 것이 가장 오른쪽 그림이다

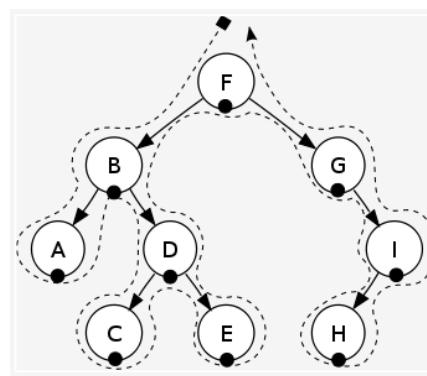
### b. 순회(traversal)

- i. 순회란 이진트리의 노드를 빠짐없이 방문하는 일정한 규칙
  - 예 1: 트리의 노드의 key 값을 빠짐없이 출력하고 싶을 때
  - 예 2: 노드의 key 값을 모두 일정한 값을 더하고 싶을 때
- ii. 일반적으로 세 가지 방법이 존재: preorder, inorder, postorder
  - 아래 그림은 Wikipedia > tree traversal에서 발췌
  - **Preorder:** **MLR** (Middle 노드 출력 → Left subtree 순회 → Right subtree 순회) → M이 앞에 (pre) 온다
  - **Inorder:** **LMR** (Left subtree 순회 → Middle 노드 출력 → Right subtree 순회) → M이 가운데(in) 온다
  - **Postorder:** **LRM** (Left subtree 순회 → Right subtree 순회 → Middle 노드 출력) → M이 뒤에 (post) 온다

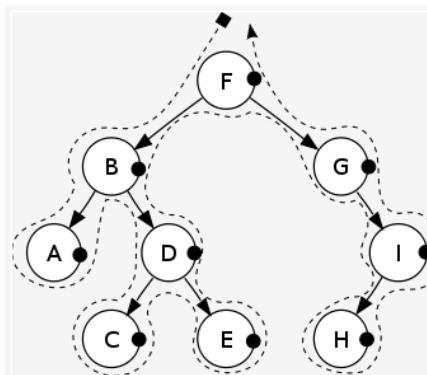
Pre-order: F, B, A, D, C, E, G, I, H



In-order: A, B, C, D, E, F, G, H, I



Post-order: A, C, E, D, B, H, I, G, F



iii. 코드는 매우 간단한 재귀 함수로 작성 가능!

예1: 노드 클래스의 메쏘드로 preorder 정의

```

def preorder(self): # 노드 self와 자손을 preorder로 방문
 if self != None:
 print(self.key) # M
 if self.left: self.left.preorder() # L
 if self.right: self.right.preorder() # R

```

### 예2: Tree/BST 클래스의 메소드로 preorder 정의할 수도

```
def preorder(self, v): # v부터 v의 자손노드를 preorder 방문
 if v != None:
 print(v.key) # M
 preorder(v.left) # L
 preorder(v.right) # R
```

- iv. inorder와 postorder도 같은 방식으로 메소드를 정의할 수 있다
- v. [?] 만약 어떤 이진트리의 preorder 시퀀스와 inorder 시퀀스만 주어진 경우, 이 두 시퀀스로부터 원래 이진트리를 복원할 수 있을까?
  - 예 1: **preorder**: F,B,A,D,C,E,G,I,H **inorder**: A,B,C,D,E,F,G,H,I
    - 앞에서 예를 든 이진트리
  - 예 2: **preorder**: A,B,D,F,C,E      **inorder**: B,F,D,A,E,C
 

|                                                                 |             |
|-----------------------------------------------------------------|-------------|
| <b>A</b> , <b>B</b> , <b>D</b> , <b>F</b> , <b>C</b> , <b>E</b> | B,F,D,A,E,C |
| M    L    R                                                     | L    M    R |

 A가 루트 노드가 되고, 왼쪽 부트리에 {B,D,F}가, 오른쪽 부트리에 {C,E}가 존재한다는 사실을 두 순서를 통해 쉽게 확인할 수 있다.  
 다음엔 왼쪽과 오른쪽 부트리에 대한 preorder와 inorder를 통해 재귀적으로 트리를 재구성할 수 있다
- postorder와 inorder가 주어지면 항상 복원가능한가? Yes!
  - preorder와 postorder가 주어지면 항상 복원가능한가? No! 반례?

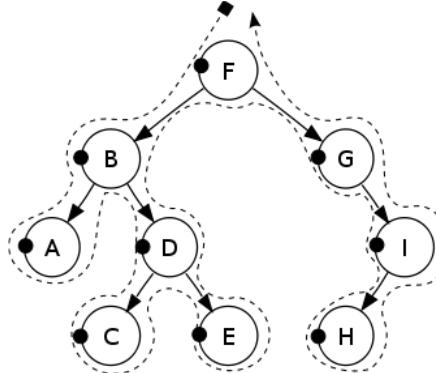
- vi. [Python: 고급] iterator를 만들고 싶다면? 즉, **for key in v:** 라고 하면 v와 자손노드들을 특정 (pre-, in-, post-) order에 따라 반복하고 싶다면?

- yield문을 사용해서 inorder에 따른 재귀적인 iterator 함수(정확하게는 generator 함수)를 만들어보자
- preorder, postorder도 유사하게 만들 수 있다  
 [0| iterator 함수를 이해할 수 있다면 파이썬 중-상급 이상!!]

```
def __iter__(self):
 if self != None: # 리프가 아니라면
 # L
 if self.left != None:
 for elm in self.left: # [?] 여기서 재귀
 yield elm
 # M
 yield self.key
 # R
 if self.right != None:
 for elm in self.right: # [?] 여기서 재귀
 yield elm
```

c. [💡 코딩 대회 문제] LCA (Lowest Common Ancestor) 문제:

- i. 이진트리의 두 노드  $u, v$ 에 대해,  $u$ 와  $v$ 의 공통 조상 노드 중 가장 깊이가 깊은 노드를 LCA라 정의한다 ( $u$ 와  $v$ 에 더 가까운 공통 조상 노드가 LCA가 된다)
  - 주의: 이진트리가 아니어도 상관없음!



- ii. 위의 트리 예에서 노드  $A$ 와  $C$ 의 공통 조상 노드는  $B$ ,  $F$  두 노드가 존재한다. 그 중에서  $B$ 가 루트 노드로부터 더 깊이 (lowest) 있기 때문에 (즉,  $A, C$ 에 더 가깝기 때문에) LCA가 된다. 노드  $B$ 와  $E$ 에 대해선  $B$ 가  $E$ 의 조상 노드이기 때문에  $B$  자체가 두 노드의 LCA가 된다
- iii. 입력으로 두 노드  $u, v$ 가 주어지면,  $LCA(u, v)$ 를 빠르게 찾는 게 목표다
- iv. 가장 간단한 방법은 몇 가지 경우로 나누어 트리의 노드를 순회하면  $O(n)$  시간에 쉽게 LCA를 찾을 수 있다. 그런데 한 번만 찾는 것이 아니라 여러 노드 쌍에 대해 LCA를 찾아야 될 수도 있다. 마치 은행의 계좌 조회를 하루에 여러 번 하는 것처럼 질문이 여러 번 주어지면 미리 필요한 자료구조를 만들어서 빠르게 질문에 답할 수 있도록 하는 게 중요하다
- v. 보통 질문을 질의(query)라고 부르고, 여러 질의를 빠르게 처리할 수 있도록 사전에 필요한 자료구조를 구축해 놓는 작업을 전처리 (preprocessing) 작업이라 한다
- vi. LCA 문제는 전처리를 통해 노드 쌍의 형식으로 주어지는 질의를 빠르게 답하는 문제로 정의된다

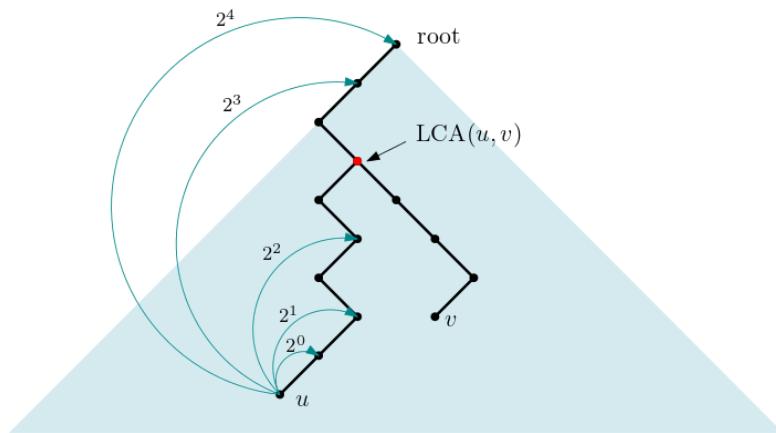
**결론:**  $O(n \log n)$  시간 전처리 단계를 거쳐 하나의 질의를 매번  $O(\log n)$  시간에 답할 수 있다!

- vii. 우선 `is_ancestor(u, v)`라는 연산을 생각해보자. 이 연산은  $u$ 가  $v$ 의 조상 노드이면 `True`, 아니면 `False`를 리턴한다.  $O(n)$  시간의 전처리를 해서 이 연산을  $O(1)$  시간에 할 수 있는 방법을 찾아보자
  - preorder를 통해 노드를 방문하는 시간을 기록해  $pre[v]$ 에 저장한다. 즉, 루트 노드  $v$ 의  $pre[v] = 1$ 이고, 노드를 방문할 때마다 시간을 1씩 증가시킨다. postorder를 통해 노드를 방문하는 시간을 같은 방법으로  $post[v]$ 에 기록한다. 이 경우엔 루트 노드의  $post[v]$ 의 값이 가장 크게 된다

- pre 값과 post 값은 모두  $O(n)$  시간에 계산 가능하다. 이 정보로부터  $\text{is\_ancestor}(u, v)$ 의 답을 어떻게  $O(1)$  시간에 알 수 있을까?
  - [힌트]  $u$ 가  $v$ 의 ancestor인 경우의 pre, post 값의 크기와 아닌 경우의 크기의 차이를 살펴보자
  - $u$ 가  $v$ 의 ancestor라면,  $u$ 가 루트 노드에 더 가깝기 때문에  $\text{pre}[u] < \text{pre}[v]$ 가 되며,  $v$ 의 자손을 모두 방문한 후에  $u$ 로 돌아가기에  $\text{post}[u] > \text{post}[v]$ 가 된다

viii. 전처리는 아래와 같은 단계를 따라 수행한다

- ix. 어떤 노드  $u$ 의 바로 1대 조상은  $u$ 의 부모 노드이다. 2대 조상은  $u$ 의 조부모 노드이다. 이런 식으로 따져서  $k$ 대 조상도 자연스럽게 정의할 수 있다. 물론,  $k$ 대 조상이 존재하지 않을 수도 있다. 이 경우엔 루트 노드를  $k$ 대 조상으로 지정한다 (아래 그림 참조 - 노드  $u$ 의  $2^0, 2^1, 2^2, 2^3$ 대 조상이 정의되고  $2^4$ 대 이상부터는 정의되지 않기에 루트 노드로 지정됨)



- x. 그럼 트리의 각 노드  $u$ 에 대해, 이차원 리스트의 값  $\text{ancestor}[u][i]$ 를 노드  $u$ 의  $2^i$ 대 조상 노드라고 정의하자. 이  $\text{ancestor}$  리스트만 전처리 단계에서  $O(n \log n)$  시간에 미리 계산해 놓고 이를 이용해  $O(\log n)$  시간에  $u$ 와  $v$ 의 LCA를 계산하려고 한다
- 노드  $u$ 는 최대  $\log n$  대 조상까지 갖을 수 있기에  $\text{ancestor}$ 는  $n \times \log n$  이차원 리스트임에 유의하자 → 당연히, 메모리 공간은  $O(n \log n)$ 이 필요함
- xi. 그럼 모든 노드  $u$ 와 모든  $i = 0, \dots, \log n$  값에 대해,  $\text{ancestor}[u][i]$ 를 계산하기만 하면 된다. 효율적인 방법은?
  - [힌트 1]  $O(n \log n)$  시간과 공간이면 충분하다
  - [힌트 2] preorder 순회를 하면서 계산 할 수 있다
    - $\text{ancestor}[u][0] = u.\text{parent}$  #  $2^0$  대 조상이니  $u$ 의 부모노드
    - preorder로 노드  $u$ 에 도착했다는 의미는  $u$ 의 조상노드를 모두 방문한 후  $u$ 에 도착했다는 의미로  $u$ 의 조상 노드에 대한

ancestor 리스트 값은 모두 계산되어 이미 저장되어 있다고 가정해도 된다

- 일종의 DP(동적계획법) 식으로 계산 가능 (DP 알고리즘 기법에 대해서는 알고리즘 교재 참조 바람)

```

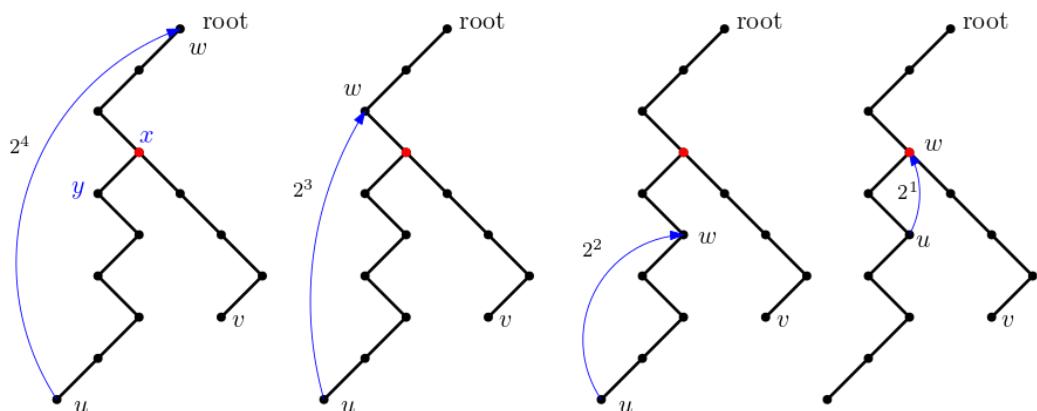
for each u in T:
 ancestor[u][1] = u.parent

for i = 2, 3, ..., logn:
 ancestor[u][i] = u의 2^i 대 조상노드
 = (u의 2^{i-1} 대 조상노드)의 2^{i-1} 대 조상노드
 = (ancestor[u][i-1])의 2^{i-1} 대 조상노드
 = ancestor[ancestor[u][i-1]][i-1]

```

xii.  $n \times \log n$  이차원 리스트 ancestor를  $O(n\log n)$  시간에 계산했다. 이제, 이 리스트를 이용해 질의로 주어진 두 노드  $u$ 와  $v$ 의 LCA를  $O(\log n)$  시간에 계산해보자

- LCA( $u, v$ )는  $u$ 에서부터 루트 노드까지의 경로  $P$ 에 있는 노드이다. 이 노드를  $x$ 라 하자. 당연히  $x$ 는  $u$ 와  $v$  모두의 조상이다. 그런데  $x$ 의 자식노드 중에서 경로  $P$ 의 노드를  $y$ 라고 하면,  $y$ 는  $u$ 의 조상이지만,  $v$ 의 조상은 아니다. (아래 그림의 가장 왼쪽 트리 참조)
- 결국,  $\text{is\_ancestor}(x, v) == \text{True}$ 지만  $\text{is\_ancestor}(y, v) == \text{False}$ 가 되는  $P$ 의 두 노드  $x, y$  ( $x$ 가  $y$ 의 부모 노드)를 찾으면 된다
- 찾는 방법은 아래 그림을 보고 설명한다
- $k = \log n$ 으로 초기화한다



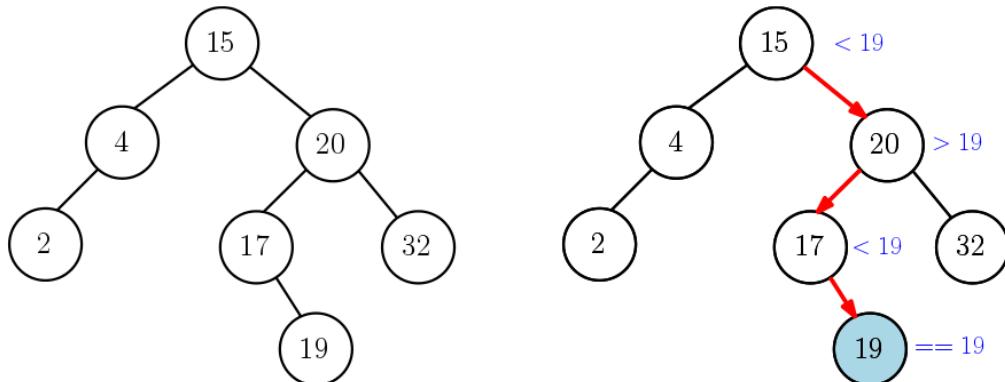
- $u$ 를 기준으로,  $2^k (= 2^4)$  대 조상을  $w$ 라 하자. 그림에서는  $2^4$  대 조상이 루트 노드를 넘어가기 때문에  $w = \text{ancestor}[u][2^4] = \text{root}$ 가 된다. 따라서,  $\text{is\_ancestor}(w, v) = \text{True}$ 이다.  $w$ 는  $u$ 와  $v$ 의 공통 조상이다. 문제는  $w$ 가 LCA인지는 아직 알 수가 없다. (LCA가  $w$ 일 수도 있지만,  $w$ 의 자손 노드 중 하나일 수 있다)

- $k = k-1$ 로 해서 다시  $2^k (= 2^3)$ 대 조상  $w$ 에 대해, `is_ancestor(w, v)`를 호출한다. 역시 (두 번째 그림에서처럼) `True`이므로, 다시  $k = k-1$ 을 해서  $2^k (= 2^2)$ 대 조상에 대해 `is_ancestor(w, v)`를 호출한다. 이 경우엔 (세 번째 그림에서처럼) `False`이다. 이 의미는 LCA가  $u$ 의  $2^2$ 대 조상과  $2^3$ 대 조상 사이의 노드라는 것이다! 그러나 아직 LCA를 확정할 수는 없고 LCA가 존재하는 구간을 결정한 상태이다
  - 최종 LCA 노드를 어떻게 찾을 수 있을까?
  - $2^2$ 대 조상노드를  $u$ 로 하여 같은 방법을 사용한다. 즉,  $u = (u\text{의 } 2^2\text{대 조상 노드}) = \text{ancestor}[u][2]$ 로 재정의한다. 그러면 **LCA는  $u$ 로부터  $2^2$ 대 조상 이내에 존재함을 알 수 있다.** (가장 오른쪽 그림 참조)
  - 이제 노드  $u$ 에서  $k = k-1$ 하고 (그러면  $k = 1$ 이 됨)  $2^k (= 2^1)$ 대 조상 노드를  $w$ 라 하고, `is_ancestor(w, v)`를 호출한다. `True`가 되므로,  $k = k-1$  ( $k = 0$ 이 됨)을 하고  $2^0 = 1$ 대 조상 노드를 보면 이 노드는  $v$ 의 조상 노드가 아니다. 즉,  **$2^1$ 대 노드 (노드  $x$ )는  $v$ 의 조상 노드이고,  $2^0$ 대 노드 (노드  $y$ )는  $v$ 의 조상 노드가 아니므로  $2^1$ 대 노드가 바로 LCA가 된다**
  - 결국, 노드  $u$ 부터  $2^k, 2^{k-1}, \dots, 2^0$ 대 조상 노드를 차례로 (총  $k$ 번 =  $\log n$ 번) 점검함으로써 LCA를 구할 수 있다.  $\log n$  번 `is_ancestor` 함수를 호출하게 되므로  $O(\log n)$  시간이면 충분하다!
- xiii. 결론:  $O(n \log n)$  시간 전처리 단계를 거쳐 하나의 질의를 매번  $O(\log n)$  시간에 답할 수 있다!
- xiv. `ancestor[u][i]` 값처럼 다른 종류의 값들도 같은 방식으로 계산해서 저장할 수 있다. 예를 들어, 노드  $u$ 에서  $2^i$  대 조상 노드까지의 경로에 있는 노드에 저장된 값(value) 중에서 최대값 또는 최소값을 저장할 수 있다. 즉, `max_value[u][i] = (u에서  $2^i$ 대 조상노드의 경로의 특정 값)처럼 정의하면  $O(n \log n)$  시간에 2차원 리스트 max_value를 계산할 수 있다`
- 이러한 기법을 **binary lifting**이라 부른다
  - 알고리즘 교재의 Graph > MST(최소신장트리) 편의 second best MST 알고리즘에서 이 자료구조를 사용한다
- xv. 다른 방법으로 해결할 수 있다. Segment 트리 (또는 Fenwick 트리)를 이용하는 방법으로  $O(n \log n)$  시간과  $O(n)$  메모리를 사용해 전처리를 한 후, LCA 질의를  $O(1)$  시간에 처리할 수 있다. 그런데, Sparse Table 자료구조를 이용하면, LCA 질의를  **$O(1)$**  시간에 처리할 수도 있다. 단, 이 때 필요한 메모리는  $O(n \log n)$ 이다

M. A. Bender and M. Farach-Colton. The LCA problem revisited. Latin American Symposium on Theoretical Informatics, 88-94, 2000

## 11. Binary Search Tree(BST: 이진탐색트리)

- a. 이진탐색트리(BST)는 저장된 key 값들이 아래의 성질을 만족하는 이진트리로 정의한다
  - i. None은 빈(empty) BST이다.
  - ii. BST의 노드  $v$ 의 key 값 ( $v.key$ )은  $v$ 의 왼쪽 자손 노드들의 key 값보다 작으면 안되고, 오른쪽 자손 노드들의 key 값보다 작아야 한다.
- b. 아래 그림의 왼쪽 트리가 BST이다.



- c. BST 클래스

```

class BST:
 def __init__(self):
 self.root = None
 self.size = 0
 self.height = 0 # 필요하다면, 높이 정보 저장

 def __len__(self):
 return self.size

 def __iter__(self): # [고급] 무슨 뜻? 건너 뛰어도 됨
 return self.root.__iter__()

 def __str__(self): # [고급] 한방향리스트 __str__와 유사 정의
 return " - ".join(str(k) for k in self)

 def preorder(self, v): # preorder print from node v
 ...
 def inorder(self, v): # inorder print from node v
 ...
 def postorder(self, v): # postorder print from node v
 ...

```

d. 탐색 연산: **search**

- i. 이진탐색트리 T의 노드 v와 v의 자손 노드들중에서 key 값을 갖는 노드를 찾아 리턴하거나 없다면 None을 리턴함
- ii. 해시 테이블 연산처럼 find\_loc(key) 함수를 먼저 구현함 - key 값이 있다면 해당 노드 리턴, 없다면 그 값이 삽입될 곳의 부모 노드 리턴

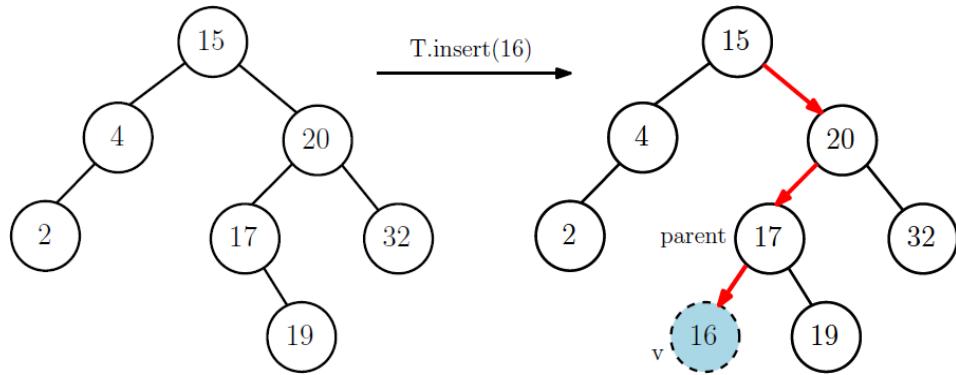
```
def find_loc(self, key):
 if self.size == 0: return None
 p = None # p is parent of v
 v = self.root
 while v: # while v != None
 if v.key == key:
 return v
 elif v.key < key:
 p = v
 v = v.right
 else:
 p = v
 v = v.left
 return p

def search(self, key):
 p = self.find_loc(key)
 if p and p.key == key: # key is in tree
 return p
 else: # key is not in tree
 return None
```

- e. 위의 그림의 오른쪽은 find\_loc(19), search(19)의 과정을 표현한 것이다.

f. 삽입 연산: **insert**

- i. 탐색트리이기 때문에, key가 들어갈 위치가 정해진다.
- ii. 아래 그림의 왼쪽 트리에 key = 16을 삽입하고 싶다면, 먼저 삽입될 위치를 find\_loc 함수를 이용해 찾는다. 값의 크기에 의해 17의 왼쪽에 삽입되어야 한다. find\_loc 함수는 16의 부모노드가 될 17 노드를 리턴한다



- iii. 17 노드의 어느 쪽 자식노드로 연결되는지는 key 값을 서로 비교해보면 쉽게 알 수 있으므로, 왼쪽 자식노드에 16을 연결한다 [주의] 새로 삽입한 노드를 마지막에 반환한다

```

def insert(self, key): # value도 추가로 받을 수 있음

 p = self.find_loc(key)

 if p == None or p.key != key:
 v = Node(key)
 if p == None:
 self.root = v
 else:
 v.parent = p
 if p.key >= key: # check if left/right
 p.left = v
 else:
 p.right = v

 self.size = self.size + 1

 # 이 곳에 height 정보 update하는 코드 또는 함수 삽입
 # update_height 함수를 준비해 호출하는 식으로 (밑의 코드)

 return v # 새로 삽입된 노드를 리턴함!
else:
 print("key is already in the tree!")
 return p # 중복 key를 허용하지 않으면 None 리턴

```

```

def update_node_height(self, v): # 노드 v의 높이 수정
 if v:
 l = v.left.height if v.left else -1
 r = v.right.height if v.right else -1
 v.height = max(l, r) + 1

def update_height(self, v): # v에서 root까지 올라가면서 높이 수정
 while v != None:
 self.update_node_height(v)
 v = v.parent

```

### g. [Python] 내장함수

- i. `__contains__(self, key)` 메소드 → if key **in** tree:처럼 **in** 연산자(membership)를 쓸 수 있게 됨

```

def __contains__(self, key): # 있다면 True, 아니면 False
 return self.search(key) != None

```

- ii. (key, value) 쌍으로 트리에 저장한다면: (해시테이블의 경우를 참조)
- `__getitem__(self, key)` # `tree[key]`로 value 접근 가능

```

def __getitem__(self, key):
 v = self.search(key)
 if v == None: # key is not found in tree
 raise ValueError("No keys in tree")
 else: # key is found
 return v.value

```

- `__setitem__(self, key, value):` # `tree[key]=value` 형식 가능

```

def __setitem__(self, key, value):
 v = self.search(key)
 if v == None:
 self.insert(key, value)
 else:
 v.value = value

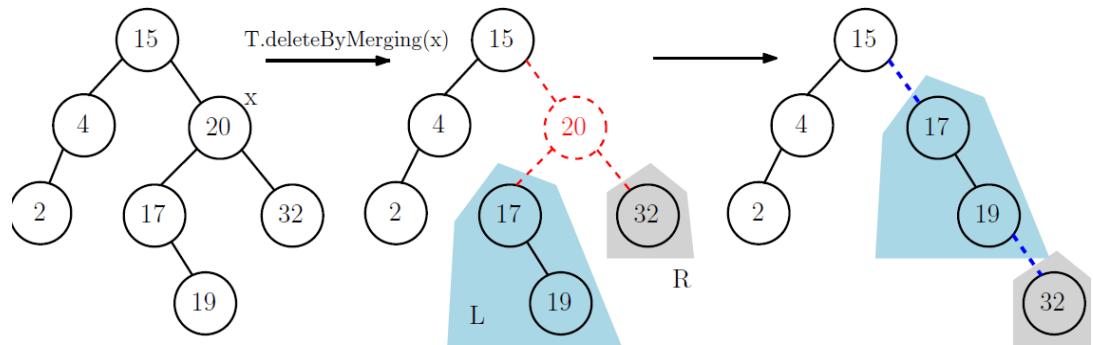
```

### h. 삭제 연산: `deleteByMerging`/`deleteByCopying`

#### i. 두 가지 버전: Merging과 Copying 방법

##### ii. `deleteByMerging`

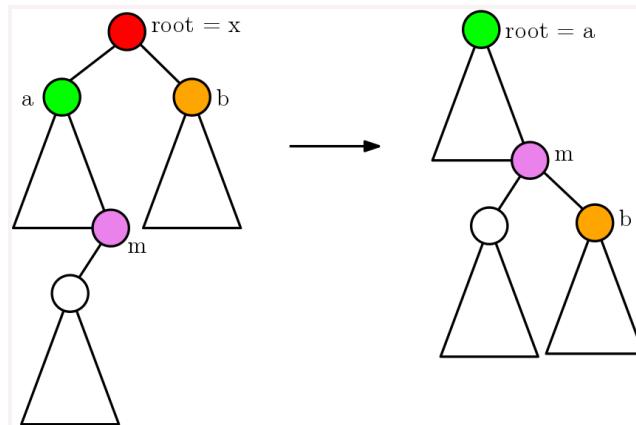
- 노드  $x$ 를 제거한다고 하면,  $x$ 의 왼쪽 서브트리  $L$ 과 오른쪽 서브트리  $R$ 을 아래와 같이 조정한다
- $L$ 을  $x$ 의 위치로 이동한다 ( $x$ 의 부모노드의 입장에서  $L$ 이  $x$  대신 자식 노드가 됨)
- $R$ 을  $L$ 에 있는 가장 큰 노드  $m$ 의 오른쪽 자식노드가 되도록 한다



- 두 가지 경우로 나눈다: 삭제할 노드  $x$ 가 root 노드인 경우와 아닌 경우

$a = x.left$ ,  $b = x.right$ 로 지정

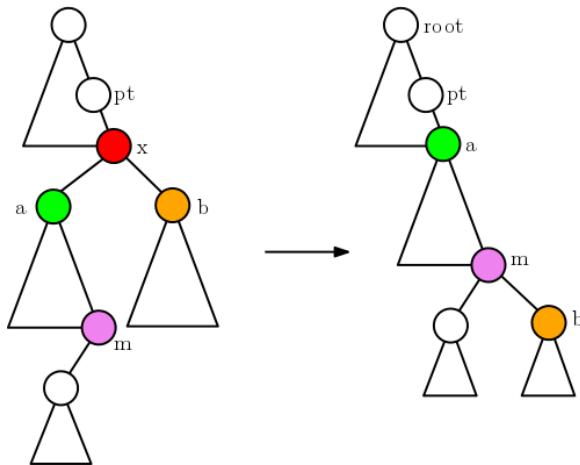
1.  $root == x$ 인 경우 (즉, 삭제할 노드가 루트노드인 경우: 삭제 후 트리의 루트가 바뀜에 주의!) [아래 그림 참조]
  - $a \neq None$ 라면 (즉,  $m$ 이 존재한다면),  $b$ 가  $m$ 의 오른쪽 자식노드가 되도록 링크 수정한 후,  $self.root = a$ 로 변경
  - $a == None$ 이면,  $b$ 를 새로운 루트로 변경하기만 하면 됨



2.  $root \neq x$ 인 경우 [아래 그림 참조]

- $pt$ 는  $x$ 의 부모노드를 의미한다
- $a \neq None$ 이면,  $m$ 이 존재하므로  $b$ 를  $m$ 의 오른쪽 자식노드로 만든 후,  $a$ 가  $pt$ 의 자식노드가 되도록 함

- $a == \text{None}$ 이면,  $b$ 가  $pt$ 의 자식노드가 되도록 함



- pseudo 코드:

```

def deleteByMerging(self, x):
 # assume that x is not None
 a, b, pt = x.left, x.right, x.parent

 # c = 노드 x 위치에 올 노드를 지정
 # s = 균형이 깨질 가능성이 있는 첫 번째 노드를 리턴함!
 # (균형이진탐색트리의 delete 연산에서 이용할 예정!)

 if a == None:
 c = b
 s = pt
 else: # a != None
 c = m = a
 while m.right: # find m
 m = m.right

 # make b as the right child of m
 m.right = b
 if b:
 b.parent = m
 s = m

 if self.root == x: # c becomes a new root
 if c:
 c.parent = None
 self.root = c

 else: # c becomes a child of pt (of x)
 if pt.left == x:
 pt.left = c
 else:

```

```

 pt.right = c
 if c:
 c.parent = pt

 self.size = self.size - 1

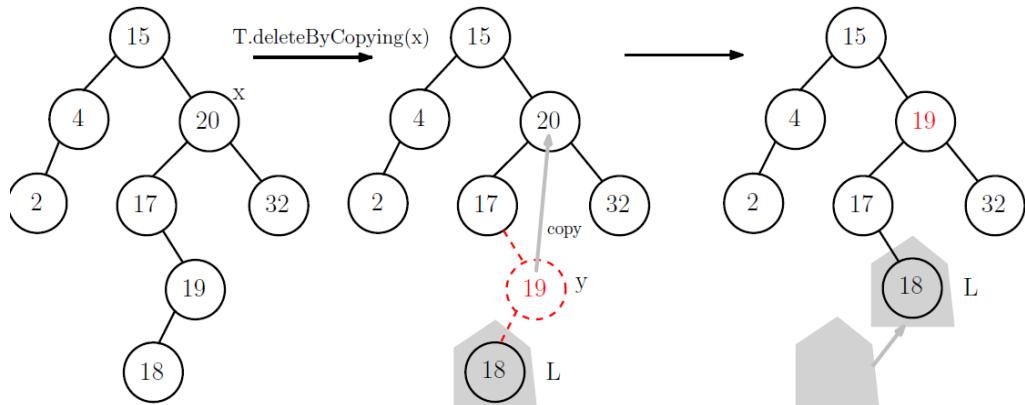
 self.update_height(s) # s부터 root까지 높이 수정

 return s # first node that would be rebalanced

```

### iii. deleteByCopying

- 노드  $x$ 를 제거한다고 하면,  $x$ 의 왼쪽 서브트리  $L$ 과 오른쪽 서브트리  $R$ 을 아래와 같이 조정한다.
  - $L$ 에서 가장 큰 값을 갖는 노드  $y$ 를 찾는다.
  - $y$ 의 key 값을  $x$ 의 key 값으로 카피한다.
  - $y$ 의 왼쪽 서브트리가 존재한다면,  $y$ 의 위치로 옮긴다.



- 이 경우도 Merging 방법처럼 몇 가지 경우로 나눠 경우에 알맞은 방법으로 처리한다. deleteByCopying 함수는 각자 구현해볼 것!

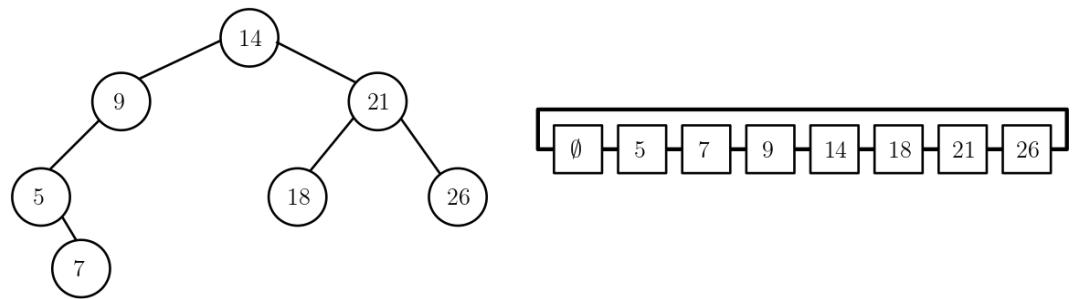
#### i. 연산 수행시간

- search는 최악의 경우에 가장 깊은 곳의 노드까지 비교하면서 내려가야 하므로 트리의 높이에 비례하는 시간이 필요하다. 즉 트리의 높이를  $h$ 라고 하면,  $O(h)$  시간이 걸린다
- insert 역시 search 과정을 통해 새로운 노드가 삽입될 위치를 찾은 후, 몇 개 노드의 parent, left, right를 수정하는 것이므로  $O(h)$  시간이 걸린다
- delete도 m을 찾기 위해 최악의 경우에  $h$  만큼 비교하면서 내려가기 때문에  $O(h)$  시간이 걸린다
- $n$ 개의 노드를 갖는 이진트리의 높이는 경우에 따라서  $n-1$ 까지 가능하므로 세 가지 연산 모두 최악의 경우에는  $O(h) = O(n)$  시간이 걸린다
  - $n$ 개의 노드를 갖는 이진트리의 높이가  $n-1$ 까지 증가하는 이진트리 모양은?

- key 값이 1, 2, ..., n이 이 순서로 insert가 된다면?
  
- n개의 노드를 갖는 이진트리의 높이는 최소 \_\_\_\_\_ 이상인가?
  - [힌트] 노드가 최대한 자식 노드를 많이 가져야 트리 높이가 작아진다. 그러면 heap 모양 아닌가?
  
- j. [💪] [해보기-BST클래스완성] search, insert, deleteByMerging, deleteByCopying 연산과 함께 아래 연산도 함께 구현해보자
  - i. **succ(self, x)**: key 값의 순서로 노드 x의 다음 노드(successor)를 리턴. x가 가장 큰 노드라면 None을 리턴
  - ii. **pred(self, x)**: key 값의 순서로 노드 x의 이전 노드(predecessor)를 리턴. x가 가장 작은 노드라면 None을 리턴
  - iii. 바로 위 트리에 대해, succ와 pred가 어디에 위치하는지 따져보자. succ(7) = 9, succ(5) = 7, pred(18) = 10 등이다. 경우를 적절히 나눠 코드를 작성하면 된다

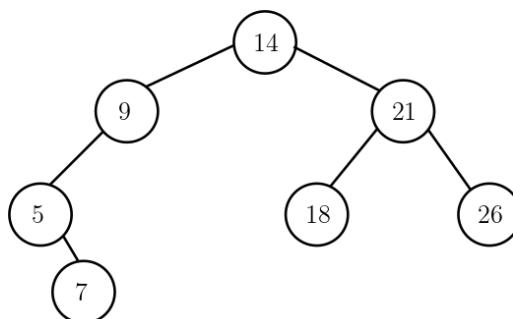
## 인터뷰 문제 몇 가지

- k. [🎤 인터뷰 문제 1: BST를 양방향 연결리스트로 변환] 이진탐색트리  $T$ 가 주어지면, inorder 순서가 되도록 양방향 연결리스트로 변환하는 함수를 작성하자. 예를 들어, 아래 왼쪽 그림의 BST  $T$ 를 오른쪽 그림의 양방향 연결리스트  $L$ 로 변환하는 것이다. BST 클래스와 양방향 연결리스트 클래스를 활용해 구현해보자.



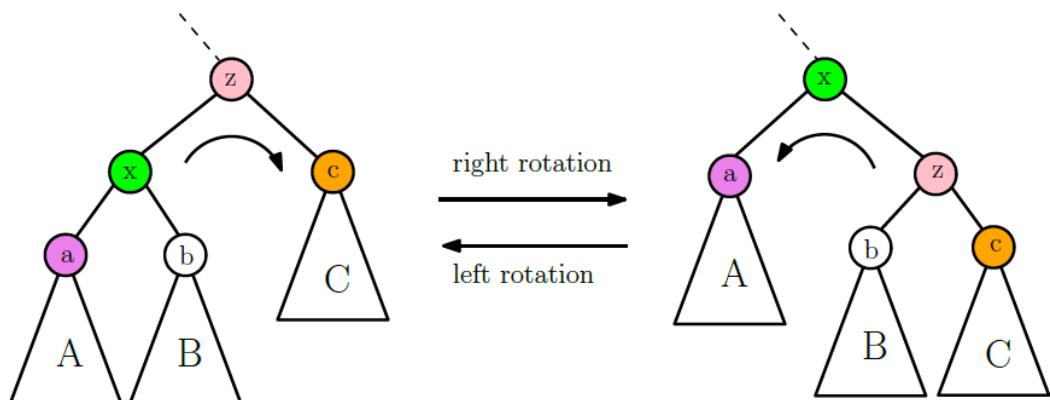
- I. [🎤 인터뷰 문제 2: BST를 양방향 연결리스트로 변환] 예를 들어, 노드가 없는 빈 BST에 insert 연산을 반복해서 아래와 같은 트리를 얻었다고 하자
- 가장 먼저 insert되는 key는 무엇인가? \_\_\_\_\_
  - 전체 key에 대한 insert 되는 순서는 유일하지 않다.  $14, 9, 5, 7, 21, 18, 26$  순서로 삽입해도 되지만,  $14, 9, 5, 7, 21, 26, 18$  순서도 가능하다. 아래 트리에 대해 이런 순서의 경우의 수는 얼마일까?

- 사실 1: 자식 노드보다는 부모 노드가 먼저 삽입되어야 한다 (예: 부트리  $\{9, 5, 7\}$ 에 대해서, 9가 가장 먼저, 다음이 5, 마지막으로 7이 입력되어야 한다)
- 사실 2: 겹치지 않는 두 부트리에 속한 노드들은 삽입 순서에 제한이 없다. (예: 두 부트리  $\{9, 5, 7\}, \{21, 18, 26\}$ 에 대해 두 부트리의 노드 사이에는 선후 관계가 전혀 없다)
- 아래 예는 몇 가지가 가능한가? 40가지 경우



## 12. Balanced Binary Search Tree (BBST: 균형이진탐색트리)

- a. 이진탐색트리의 연산 시간은 오직 트리 높이  $h$ 에 의해 결정되는데, 문제는 최악의 경우엔  $h = O(n)$ 이 되어 탐색, 삽입, 삭제 연산이 매우 느리다는 것이다
- b. 연산 속도를 빠르게 하기 위해선 트리 높이를 되도록 작게 유지하는 게 중요하다
- c. 삽입, 삭제 연산을 반복하더라도  $n$ 개의 노드를 갖는 이진트리의 높이를 항상  $O(\log n)$ 이 되도록 유지할 수 있다. 이렇게 유지할 수 있는 이진탐색트리를 균형이진탐색트리라 부른다
- d. 대표적인 균형이진탐색트리에는 AVL 트리, Red-Black 트리, 2-3-4 트리, Splay 트리 등이 있다
- e. 삽입과 삭제로 인해  $h = O(\log n)$ 이라는 높이 조건을 만족할 수 있도록 필요한 경우에는 이진트리의 모양을 (선제적으로) 변경해 높이를 줄이게 된다. 이러한 모양 변경은 주로 **rotation**이라는 기본 연산에 의해 이루어진다
  - i. left rotation과 right rotation이 있고, 서로 대칭적이다
  - ii. 회전 후에도 BST의 값의 순서가 그대로 유지되어야 한다
  - iii. 아래 그림에서, right rotation 전의 inorder 순서는  $AxBzC$ 이고, 회전 후의 순서 역시  $AxBzC$ 이므로 같다. left rotation 전, 후의 순서도 같다. 따라서 rotation이 BST의 순서를 그대로 유지한다
  - iv. **수행시간**: 상수 개의 링크 수정이면 충분하므로  **$O(1)$  시간 필요**



```
def rotateRight(self, z): # rotateLeft도 유사하게 작성
 if z == None: # do nothing
 return
 x = z.left
 if x == None: # if x == None: no rotate
 return
 b = x.right # b == None 인 경우도 가능
 x.parent = z.parent
 if z.parent:
 if z.parent.left == z:
 z.parent.left = x
 else:
 z.parent.right = x
 x.right = z
 z.parent = x
 z.left = b

 if b: b.parent = z

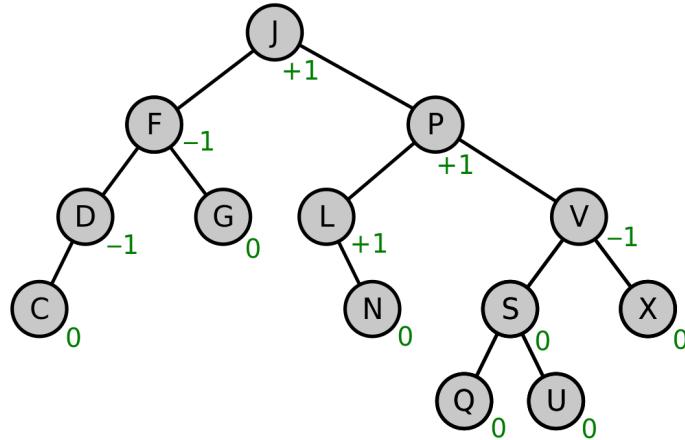
 # z == self.root라면 x가 새로운 루트가 되어야 함!
 if z == self.root:
 self.root = x

 # [주의] height를 관리한다면 z와 x의 height 수정하는 코드 추가
 self.update_node_height(z)
 self.update_node_height(x)
```

## f. AVL 트리

- i. 정의: 모든 노드에 대해서, 노드의 왼쪽 부트리와 오른쪽 부트리의 높이 차이가 1이하인 이진탐색트리

- 소련(Soviet Union)의 과학자 Georgy Adelson-Velsky와 Evgenii Landis가 1962년에 제안
- 아래 트리는 wikipedia에 등장하는 AVL 트리의 예:



- ii. 위의 그림에서 노드 옆의 -1, 0, +1은 **BalanceFactor**를 의미한다

- BalanceFactor는 오른쪽 부트리의 높이 - 왼쪽 부트리의 높이로 정의된다
- AVL 트리의 정의에 따라, 모든 노드의 BalanceFactor는 -1, 0, +1 중 하나임

- iii. 양쪽 부트리의 높이 차가 1이하면  $h = O(\log n)$ 을 보장하는가?

- [?] 높이  $h = 0$ 인 AVL 트리는 루트 노드 하나로만 구성되며,  $h = 1$ 인 AVL 트리 중 노드 수가 가장 작은 트리는 루트 노드가 하나의 자식 노드만 갖는 경우이다. 그럼  $h = 2$ 인 경우에 노드 수가 가장 작은 AVL 트리는 어떤 모양을 가질까? 루트 노드의 두 부트리를 생각해보면, 한 부트리는  $h = 1$ 인 최소 노드 수를 갖는 AVL 트리가 오고, 나머지 부트리는  $h = 0$ 인 최소 노드 수를 갖는 AVL 트리가 오는 게 최소 노드 수의  $h = 2$ 인 AVL 트리가 됨을 쉽게 확인할 수 있다. 따라서  $h = 1$ 인 경우와  $h = 0$ 인 경우의 AVL 트리의 최소 노드 수는 각각 2개, 1개이므로  $h = 2$ 인 경우에는 루트 노드 개수 1을 더해  $1 + 2 + 1 = 4$ 가 된다
- [?] 높이가  $h$ 인 AVL 트리 중에서 가장 작은 노드 수를  $N_h$ 라 하면,  $h = 0, 1, 2$ 의 경우에  $N_0, N_1, N_2$ 의 값을 구했다. 그러면 일반적인 높이  $h$ 에 대해서,  $N_h$ 는 얼마일까? 점화식으로 표현 가능(왜?)

$$\circ \quad N_h = N_{h-1} + N_{h-2} + 1$$

○ **N(h) 계산방법 1 [정확하게~]**

- $N_h + 1 = N_{h-1} + N_{h-2} + 1 + 1$   
 $N_h + 1 = (N_{h-1} + 1) + (N_{h-2} + 1)$

- $F_h = F_{h-1} + F_{h-2}$  ( $\leftarrow F_h = N_h + 1$ ),
- $F_0 = N_0 + 1 = 2$ ,  $F_1 = N_1 + 1 = 3$ 이므로, 2와 3으로 시작하는 피보나치 수열 중 h번째 값이  $F_h$ 가 됨!

- 일반적인 피보나치 수열은 0과 1로 시작하므로,  $F_h$ 는 일반 피보나치 수열의  $h+3$ 번째 피보나치 값!
- [Wikipedia]  $h+3$ 번째 Fibonacci 수는 대략  $\phi^{h+3}/\sqrt{2}$ ,  $\phi = (1 + \sqrt{5})/2 \approx 1.618$  황금률(golden ratio)이라 부름

- $N_h + 1 = F_h \geq \phi^{h+3}/\sqrt{2}$
- $n = N_h \geq \phi^{h+3}/\sqrt{2} - 1 \rightarrow h \leq c \log(n + 2) + b$   
 $c = 1/\log 2, b \approx 0.328$

○ **N(h) 계산방법 2 [정확한 값이 아닌 유사한 값 계산]**

- $N_h = N_{h-1} + N_{h-2} + 1 > 1 + 2N_{h-2} > 2N_{h-2}$
- $N_h > 2^2 N_{h-4} > \dots > 2^{h/2} N_0 = 2^{h/2}$
- $n = N_h > 2^{h/2} \rightarrow h < 2 \log n$

iv. 결론: n개의 노드로 구성된 AVL 트리의 높이 h는 다음과 같은 범위에 놓인다

$$\log(n + 1) \leq h < c \log n + b$$

임이 증명되었다. 따라서 AVL 트리는 **균형이진탐색트리**이다 (여기서 c는 2보다 작은 상수)

v. **삽입과 삭제를 하게 되면**, 어떤 노드의 두 서브트리의 높이 차가 1보다 크게 되는 경우가 있을 수 있다. 그 경우엔 특별한 규칙에 따라 한번 또는 여러 번의 회전을 통해 조건이 만족되도록 트리의 높이를 재조정(**rebalancing**)해야 한다. 단, 탐색(search)은 트리의 변화가 없으므로 재조정 불필요

- search: 트리 높이 h가  $O(\log n)$ 이므로  $O(\log n)$  시간에 가능
- insert: 삽입될 위치 탐색에  $O(\log n)$ , 재조정을 위한 회전을 상수번 실행한다고 증명되어 총  $O(\log n)$  시간에 가능
- delete: 삭제할 노드 탐색에  $O(\log n)$  시간, 삭제 후 재조정을 위한 회전을 최악의 경우에  $O(\log n)$ 번 실행할 수도 있다고 증명되어 총  $O(\log n)$  시간에 가능

vi. AVL 트리를 위한 클래스 마련

- Node class에 self.height = 0를 추가해 노드의 높이 정보를 저장

```
class AVL(BST): # BST 클래스를 부모 클래스로 지정
```

```
BST 클래스의 멤버, 메소드 상속받아 사용가능
__init__가 없으므로 부모 클래스 __init__ 함수가 자동 호출됨
```

```
def insert(self, key): # AVL 클래스의 search 함수
```

1. v = 부모클래스 BST의 insert 함수 호출 (트리에 삽입)한다. insert 함수는 새로 삽입된 노드 v를 리턴함에 유의한다
2. v부터 루트 노드 방향으로 올라가면서 균형이 깨지는 첫 번째 (가장 깊은) 노드 z를 찾는다
3. 균형이 깨지는 가장 깊은 노드 z가 존재하면, rebalance!

```
def delete(self, x): # AVL 클래스의 delete 함수
```

1. 부모클래스의 delete 함수 호출 (Merging 또는 Copying)
  - \* delete 함수는 **균형이 깨질 가능성이 있는 가장 깊은 노드 s**를 리턴
2. s부터 루프 노드 방향으로 올라가면서 균형이 깨지는 노드 z를 찾고, 그런 노드 z가 존재하면 균형을 맞추기 위해 rebalance!

vii. 삽입: insert(key)

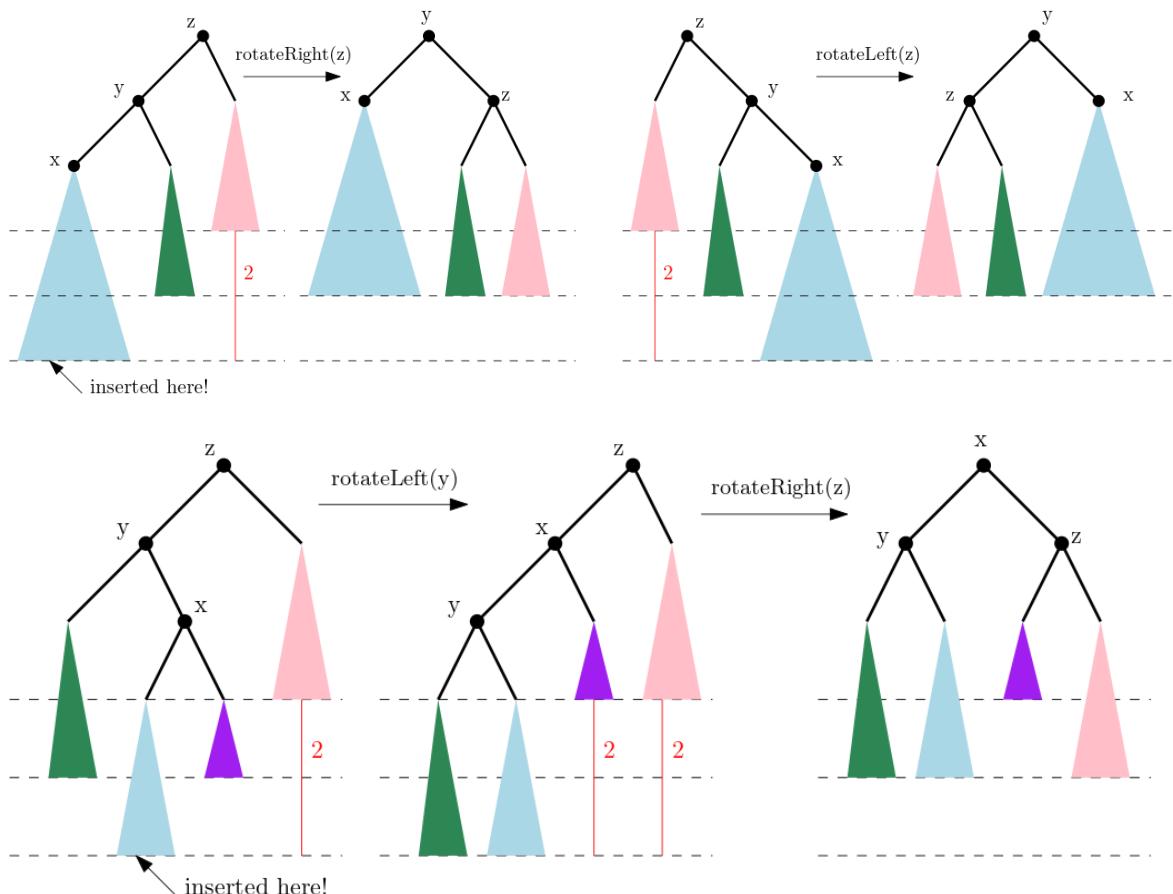
- [주의] BST 클래스의 insert 함수를 부른다. 이 함수에서는 key 값을 갖는 새로운 노드 v를 만들어 리프 노드로 삽입한 후, v를 리턴하게 된다

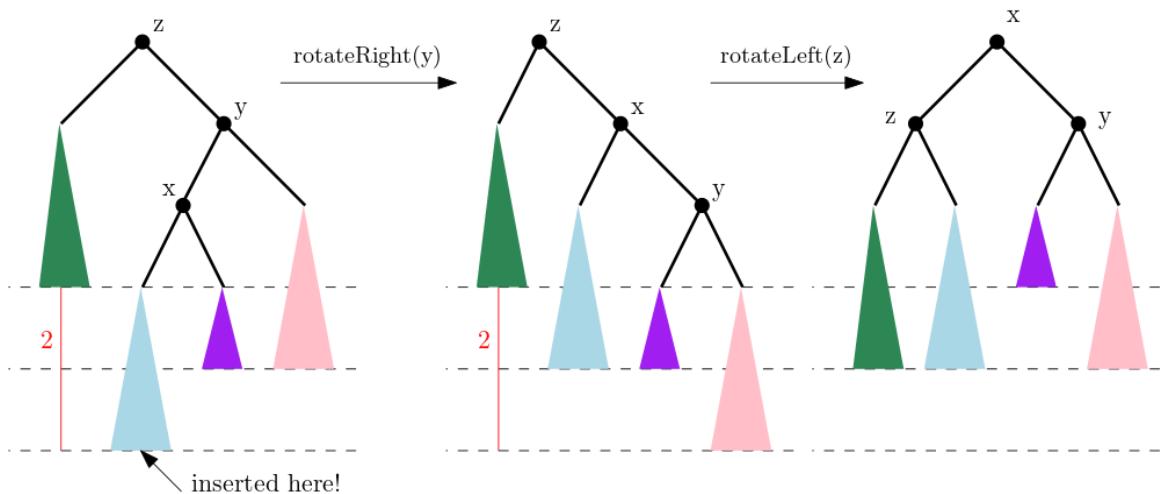
- 부모(조상) 클래스의 같은 이름의 메소드를 호출하는 법:

```
v = super().insert(key)
```

- v로 인해 v의 조상 노드의 균형이 깨질 수 있음 → rebalancing 필요!
  - BalanceFactor = -2 또는 2인 조상 노드가 발생한다면, rotation을 통해 BalanceFactor를 조정해야함
- v에서 루트로 올라가면서 균형이 처음으로 깨진 노드를 z라 하자: (z의 BalanceFactor가 -2 또는 2라는 의미)

- $z \rightarrow v$  경로에서  $z$ 의 자식 노드를  $y$ ,  $y$ 의 자식노드를  $x$ 라 하자  
( $x == v$  일 수도 있음)
  - 균형을 맞추는 함수  $rebalance(x, y, z)$ 를 호출해 균형을 맞춤
  - 함수  $rebalance(x, y, z)$ 는  $z-y-x$  세 노드 사이의 부모-자식 관계에 따라 네 가지 경우중 하나 ( $left-left$ ,  $right-right$ ,  $left-right$ ,  $right-left$ )가 되며, 경우에 따라 한 번 또는 두 번의 rotation을 수행한다. (아래 그림 참조)
- left-left, right-right인 경우에는 한 번의 회전으로 충분
- left-right, right-left인 경우에는 각각 왼쪽+오른쪽 회전과 오른쪽+왼쪽 회전을 해야 하기에 두 번의 회전이 필요
- [?] 두 번보다 더 많은 rotation이 필요한 경우는 없을까?





- insert 함수의 pseudo code는?

```

def insert(self, key):
 # BST의 insert 함수는 실제 삽입된 노드가 리턴됨

 1. v = super().insert(key) # v: 삽입된 노드

 2. # find x, y, z: 조상 노드 따라 올라가면서 찾기

 x, y, z = v, v.parent, None
 while y:
 z = y.parent
 if z and z is balanced:
 x, y = y, z
 else:
 break

 3. w = rebalance(x, y, z)

 4. self.update_height(w) # w의 조상 height 업데이트

 5. if w and w.parent == None: # root가 바뀐 경우

 self.root = w

```

viii. 이제, rebalance(x, y, z)를 구현해보자

- z - y - x 순서로 부모-자식 관계 성립한다. 위의 경우처럼 다음의 두 경우로 나눠 처리한다
  - z - y - x 가 일직선인 경우는 한 번의 회전이면 충분하고,
  - z - y - x 가 삼각형 모양인 경우는 두 번의 회전이 필요하다

- rebalance 한 후에 z 노드 위치에 오는 노드 (top 노드)를 리턴하도록 구현해야 한다 (트리의 루트가 바뀌는 경우인지를 검사해야 하고, 그런 경우에는 루트를 update해야 한다)
- 한 번 또는 두 번의 회전을 한 후에는 **이전의 높이 상태로 돌아가기** 때문에 더 이상 rebalance를 호출할 필요가 없다. 위의 회전 그림을 보면, 삽입으로 인해 높이가 1 증가해서 균형이 깨졌는데, rebalance를 통해 증가했던 높이 1이 다시 감소해 전체 트리의 균형이 맞춰진다. 이는 z 노드의 조상 노드에서 rebalance가 필요없다는 의미다

```
def rebalance(x, y, z):
 if z == None: # rebalance 불필요
 return
 if z.left == y and y.left == x: # 왼쪽 방향으로 일직선
 self.rotateRight(z)
 return y
 elif z.right == y and y.right == x: # 오른쪽 일직선
 self.rotateLeft(z)
 return y
 elif z.left == y and y.right == x: # 삼각형 경우 1
 self.rotateLeft(y)
 self.rotateRight(z)
 return x
 elif z.right == y and y.left == x: # 삼각형 경우 2
 self.rotateRight(y)
 self.rotateLeft(z)
 return x
```

#### ix. 삭제: delete(**u**) # 삭제할 key 값이 저장된 노드 **u**를 매개변수로 전달!

- **u**를 BST의 삭제 방법 중 하나를 이용해 삭제한다 (BST의 삭제 함수에서는 실제 **u**가 삭제되거나 다른 노드로 대체된다. 중요한 것은 삭제 또는 대체를 통해 **높이에 영향을 받을 수 있는 첫 노드** (가장 깊은 곳에 위치한 노드)를 파악해야 한다)

그러한 노드를 **s**라 하자 (이 노드 **s**는 deleteByMerging과 deleteByCopying 함수에서 리턴되는 노드로 정의되어 있기에 이 두 함수에서 구현되어야 한다)

- **deleteByMerging** 방법을 사용한다면, 리턴 노드 **s**부터 위로 올라가면서 높이 조건이 만족하지 않는 (균형이 깨지는) 첫 노드 **z**를 찾는다
- **deleteByCopying** 방법의 경우도 유사하게 높이에 영향을 받을 수 있는 가장 깊은 곳의 노드 **z**를 찾는다

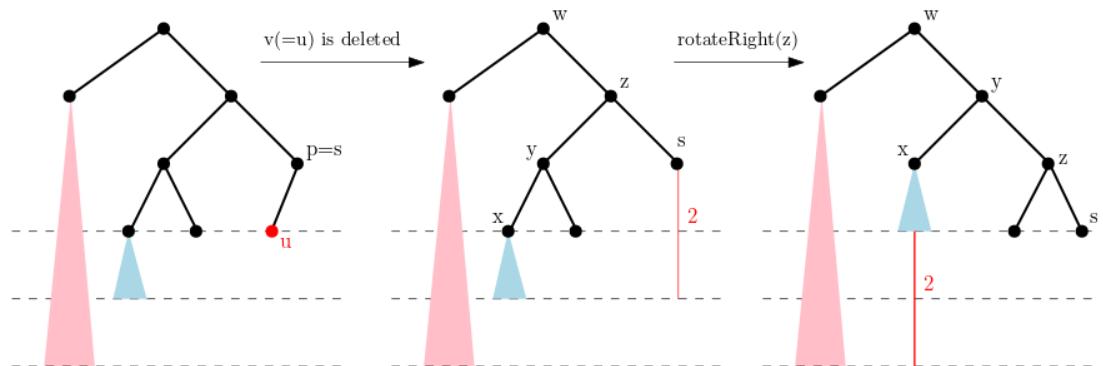
- $z$ 에서 균형을 맞추기 위해서는  $z$ 의 부트리 중에서 높이가 더 큰 부트리에 속하는  $z$ 의 자식 노드를  $y$ 로 정한다. 같은 방식으로  $y$ 의 부트리 중에서 높이가 더 큰 부트리에 속하는  $y$ 의 자식 노드를  $x$ 로 한다. 그러면 `insert` 함수의 경우와 같이  $z - y - x$  세 노드가 정의된다

바로 이 부분이 `insert` 경우와 다르다. (`insert`는 새로운 노드  $v$ 가 삽입되는 곳이 무거워져 높이 균형이 깨지게 되지만, `delete`에서는 노드  $u$ 가 삭제되면서 가벼워져 높이 균형이 깨지게 되기에 상황이 다를 수 밖에 없다)

이렇게  $z - y - x$ 를 정의하는 이유는 회전을 통해 무거운 (높이가 큰) 쪽의 일부분을 가벼운 (높이가 작은) 쪽으로 넘겨 높이 차이를 줄이고 싶기 때문이다

- 아래 그림을 예를 들어 설명해보자

- 가장 왼쪽 그림처럼 리프 노드인  $u$ 가 삭제되면, 가운데 그림처럼  $u$ 의 부모 노드인  $z$ 의 `balanceFactor` = -2가 될 수 있다
- 이를 조정하기 위해선  $z$ 의 왼쪽 자식 노드를  $y$ 로,  $y$ 의 자식 중 더 높이가 큰 부트리를 갖는 자식 노드를  $x$ 로 정의해서,  $z$ 에서 `rotateRight(z)`를 해야 한다 (회전한 결과가 가장 오른쪽 그림)
- 회전을 통해  $z-y-x$ 의 균형을 맞췄지만,  $y$ 의 새로운 부모 노드인  $w$ 에서 균형이 깨지게 된다. (이런 현상이 `insert`에서는 일어나지 않는다)
- 따라서 다시  $w \rightarrow z$ ,  $y \rightarrow y$ ,  $x \rightarrow x$ 로 지정하여 균형을 다시 맞춰야 한다. 경우에 따라선 루트 노드까지 올라가면서 계속 같은 작업을 해야 할 수도 있다
- 루트까지 올라가면서 균형을 맞추는 과정은 트리의 높이만큼만 반복하면 된다  $\rightarrow O(h) = O(\log n)$  회전이면 충분  $\rightarrow O(\log n)$  시간



- Pseudo code는 아래와 같다:

```

def delete(self, u):

 # deleteByMerging을 사용해도 됨
 # 노드의 삭제로 노드 높이에 영향을 받는 (불균형 가능성 있는)
 # 첫 노드 s가 리턴된다 가정!

 s = super().deleteByCopying(u)

 while s != None: # go up to the root
 self.update_node_height(s) # update s.height
 if s is not balanced: # z-y-x chain 존재!
 z = s
 # z.left, z.right가 None인 경우엔 height -1로 가정
 if z.left.height >= z.right.height:
 y = z.left
 else:
 y = z.right
 if y.left.height >= y.right.height:
 x = y.left
 else:
 x = y.right

 s = self.rebalance(x, y, z)

 # rebalance는 rotation후 새로운 top 노드를 리턴
 w = s
 s = s.parent

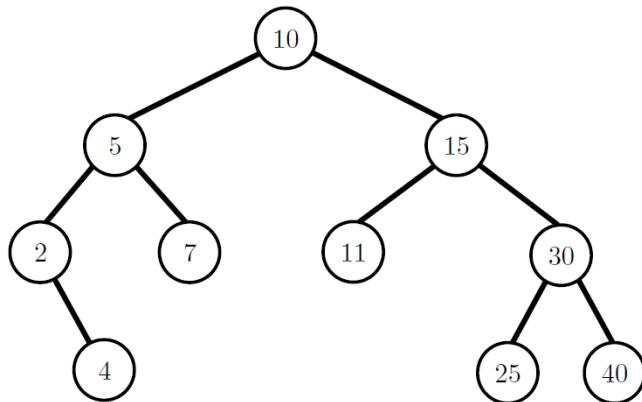
 self.root = w # w could be a new root

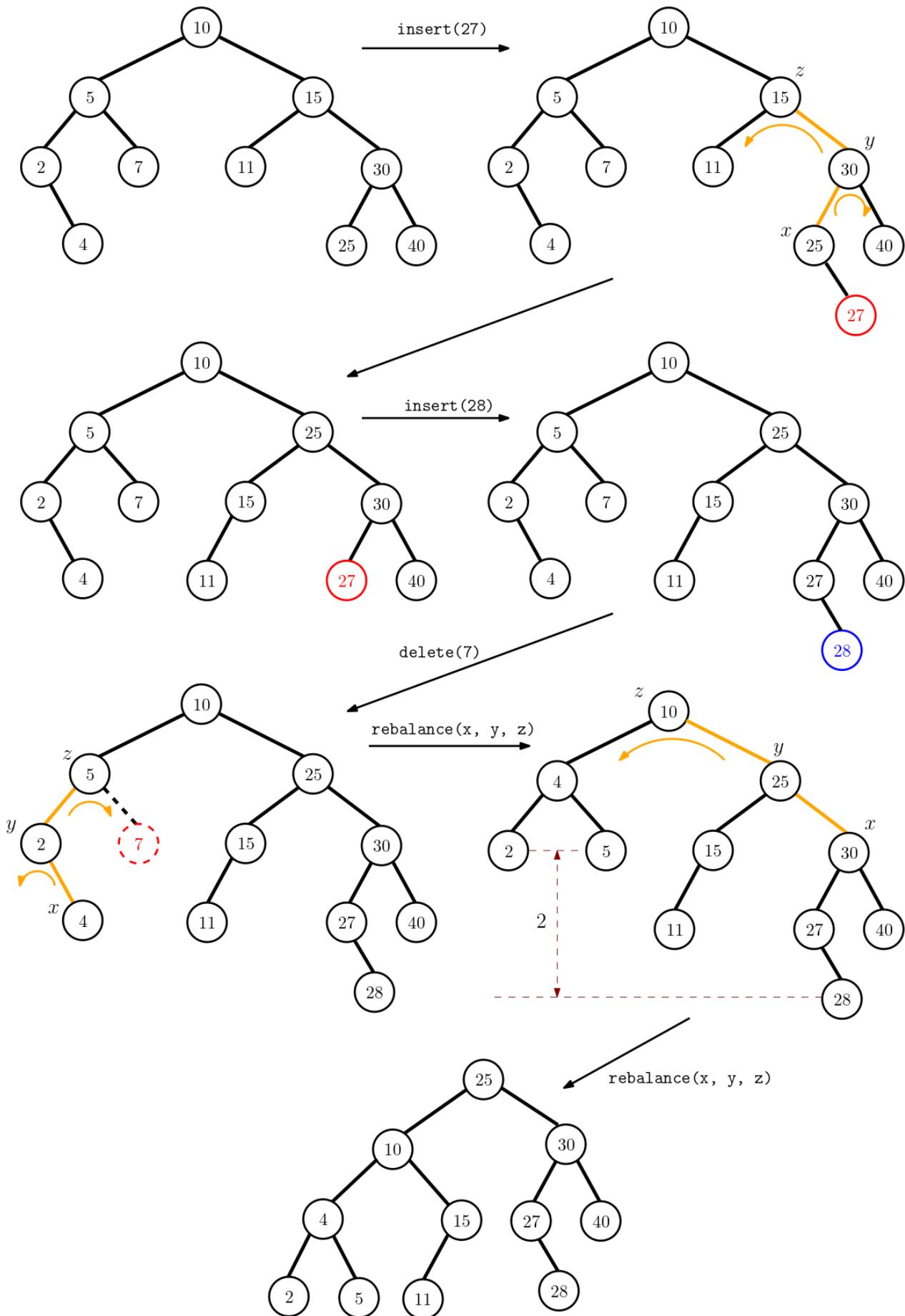
```

- 자세한 삽입/삭제 알고리즘과 pseudo 코드는 [wikipedia](#) 등의 자료를 참조할 것
- x. AVL 트리의 insert 경우의 회전은 최대 2번이면 항상 균형을 유지할 수 있고, delete 경우에는 최악의 경우에 루트 노드를 한 레벨씩 올라가면서 (최대 2번의) 회전을 반복할 수도 있기에 **0(logn) 번의 회전이 가능**
- xi. 결국, insert의 경우엔 key 값의 위치를 찾아야 하고, delete의 경우엔 (key 값을 갖는 노드의 탐색시간을 제외하더라도) rebalance을 위해 (최악의 경우에는) **0(logn) 번의 회전할 수 있으므로 두 연산 모두 0(logn) 시간이면 필요하다!**

- xii. [연습] 아래 AVL 트리에 `insert(27)`를 수행해본다. 다음으로 `insert(28)`를 수행해본다. 마지막으로 `delete(search(7))`을 수행해본다

- 답은 다음 페이지에서 확인





### g. Red-Black 트리

- i. **가정:** 리프노드의 두 자식노드인 None 노드는 NIL 노드라고도 불린다. Red-Black 트리를 정의하는 동안 이 NIL 노드를 **리프노드 또는 외부노드(external node)**라고 부름 (NIL 노드가 아닌 일반 노드를 내부 노드라 부름)

- ii. **정의:** 다음의 5가지 조건을 만족하는 이진탐색트리로 정의

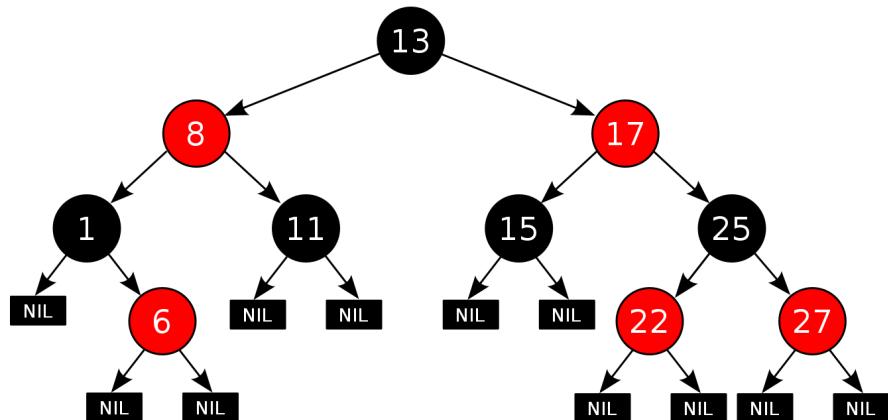
- 각 노드는 **red** 또는 **black**의 색을 갖는다
- 루트노드의 색은 **black**이다
- NIL 노드의 색은 **black**으로 정의한다

따라서 리프노드인 NIL 노드는 **black**이고, 루트노드의 부모노드도 NIL 노드이므로 루트의 부모노드는 **black**이다

- 어떤 노드가 **red**라면, 두 자식노드는 모두 **black**이다
- 어떤 노드에서 서브트리의 리프 NIL노드까지의 모든 경로에 포함된 **black** 노드의 개수는 같다. (이를 black-height라고 부름)

- iii. 아래 트리는 wikipedia에서 가져온 red-black 트리의 예이다

- 루트노드와 모든 NIL 노드는 **black**이고, **red** 노드의 두 자식노드는 모두 **black**임
- 임의의 노드에서 서브트리의 각 NIL 노드까지의 경로에 포함된 **black** 노드 개수는 **3**으로 모두 같음
- **사용 예**
  - C++의 STL(Standard Template Library)에서 사용하는 set, map, multiset, multimap은 red-black 트리로 구현됨
  - java의 TreeMap, TreeSet 또한 red-black 트리로 구현됨



iv. 1-5번 조건을 만족하는 경우, red-black 트리 T의 높이는 얼마일까? 답은 **0(logn)**

- $h(v)$ : v의 높이 (또는 v의 서브트리 높이)
- $bh(v)$ : v에서 v의 서브트리의 NIL 노드까지의 경로에 포함된 black 노드 갯수 (**black-height**)
  - 주의: v가 black이면 v는  $bh(v)$ 에 포함하지 않는다고 가정
- 사실 1: 노드 v의 서브트리가 가질 수 있는 내부 노드의 최소 개수는  $2^{bh(v)} - 1$  이다
  - $h(v)$ 에 관한 귀납법(induction)에 의해 증명
  - $h(v) = 0$ 인 기본 경우(base case)에 성립함을 증명한 후,  $h(v) \leq k$ 인 경우 성립한다고 가정(inductive hypothesis)을 한 후  $h(v) = k+1$ 인 경우 성립함을 증명하는 방식
    - $h(v) = 0$ 인 경우: 내부 노드가 없으므로 NIL 하나로 구성된 빈 트리가 되어,  $bh(v) = 0$ 이 되어야 함
      - $2^{bh(v)} - 1 = 2^0 - 1 = 0$
    - $h(v) \leq k$  경우에는 v의 자손 내부 노드의 최소 개수는  $2^{bh(v)} - 1$ 이라고 가정함
    - $h(v) = k+1$  경우:
      - $h(v) > 0$ 이 성립함
      - v의 두 자식노드의 black height는  $bh(v)$ 이거나  $bh(v)-1$  중 하나임. ([?] 왜?)
      - v의 두 자식 노드의 높이는 당연히 v의 높이보다 작다  
→ 따라서 두 자식 노드에 대해서 가정이 성립한다
      - 그럼 v의 두 자식 서브트리에 포함된 대한 내부 노드 개수에 대한 최소 개수는 다음과 같이 증명됨

$$2^{bh(v)-1} - 1 + 2^{bh(v)-1} - 1 + 1 = 2^{bh(v)} - 1$$

- 사실 2: 루트에서 NIL까지의 임의의 경로에 포함된 black 노드 수는 경로에 포함된 노드 수의 반 이상이다
  - [?] 왜? [hint] Red-Black 트리의 4번째 성질 참조
- 사실 2로부터 루트노드 r의 black height  $bh(r)$ 은  $bh(r) \geq h(r)/2$
- 사실 1에 의해 트리 노드 수는 최소  $2^{bh(r)} - 1 \geq 2^{h(r)/2} - 1$  이상이어야 함
- $n \geq 2^{h(r)/2} - 1 \leftrightarrow h(r) \leq 2 \log(n + 1)$ 이므로 트리의 높이는 **0(logn)**이 되어 [증명 끝]

#### v. insertion 전략 (삽입 후에 5개 조건을 만족하도록 회전+색 조정 필요)

- 새 노드 x를 BST처럼 삽입한다
- x의 색을 우선 **red**로 칠한다 (`x.color = red`)
- `x.parent.color == black`이 될 때까지 위로 올라가면서 아래처럼 경우를 나눠 색 배정 과정을 반복한다

```
while x.parent.color == red: # x가 루트라면, x.parent는 NIL노드
 # NIL 노드 색은 정의에 의해 black
```

# 주의: while 루프는 오직 2번만 반복하고 탈출함에 유의

```
p, g, s, u 정의
p = x.parent, g = x.parent.parent
s = x.sibling, u = x.uncle
```

`p.color == red`이므로, 조건 4가 깨짐 → 색 조정 필요

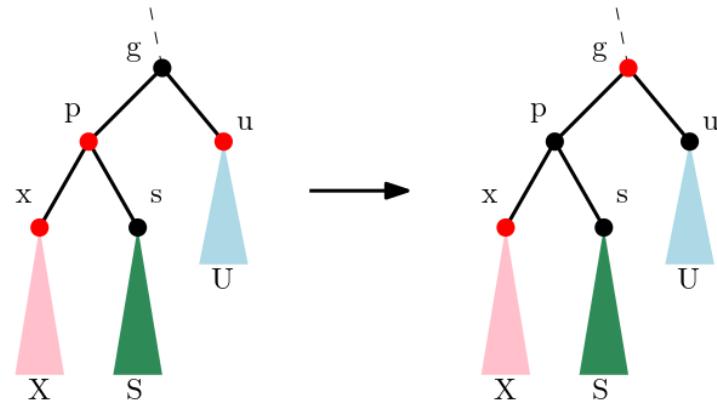
- `s.color`는 무엇인가? 항상 **black**! (왜?)
- `p.color == red`이므로 `g.color`는 항상 **black**!

경우 1: `u.color (u.color) == red` →

- `g.color = red`
- `p.color = u.color = black`

- `g`의 색이 **red**로 변경되었기에 두 자식 노드인 `p, u`는 **black**이어야 함. 그런데 `p, u`의 색이 **black**으로 조정되어 조건 4를 만족한다
- 이제 5번 조건이 만족하는지 따져보자. 색 조정 전에 노드 `g`를 통과해 `x, s, u`에 포함된 노드에 도착하는 경로는 `g`가 **black height**에 하나를 기여한다. (노드 `s`가 **black** 노드지만 조건 5에 의해 `g`를 통과해 `x, s, u`에 포함된 노드까지의 **black** 노드 개수는 같기에 `s`의 색을 따로 고려할 필요는 없다)

색 조정을 한 후에는 `g`의 **black height** 기여가 `p`와 `u`가 대신 기여하게 된다. 즉, `g`를 통과하는 경로는 `g`가 아닌 `p` 또는 `u`에서 **black** 노드가 카운트되어 동일한 **black** 노드 개수를 보장하게 된다. 따라서 5번 조건도 만족한다

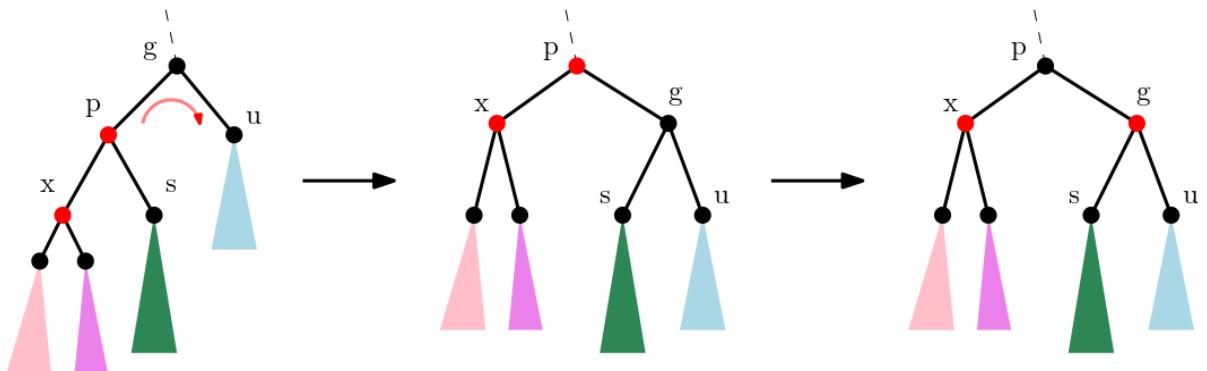


- g의 부모 노드가 **red** 노드일 수도 있다. 그렇다면, 다시 조건 4가 깨지게 되므로 색조정을 해야한다
- $x = g$ 로 하여 while 루프 다시 반복! (만약, g가 루트노드라면, g.parent가 NIL 노드, 즉 **black** 노드이므로 루프를 탈출함에 주의하자. g.color가 **red**인 채로 루프를 탈출하기 때문에, 조건 2번을 만족하기 위해서는 **루트 노드의 색을 무조건 black**으로 지정해야 한다. 그래서 코드의 가장 마지막에 T.root.color = **black** 실행한다)

**경우 2:  $u.\text{color} (u.\text{color}) == \text{black}$ :**

**경우 2.1:  $x - p - g$ 가 linear 모양인 경우:**

- 노드 g에서 한 번만 회전한 후, 색 조정 수행
- $p.\text{color} = \text{black}$ ,  $g.\text{color} = \text{red}$
- p가 **black**이 되므로 p의 부모노드의 색을 신경쓸 필요는 없음 따라서 더 이상 색 조정할 필요 없다
- while 루프의 조건문에서  $x.\text{parent}.\text{color} = p.\text{color} = \text{black}$ 이 되어 **루프를 탈출**하게 되어 재조정 작업이 끝난다
- 5번 조건의 만족 여부도 역시 경우 1처럼 따져볼 수 있다
- **한 번의 회전만으로 색 조정 완료!**

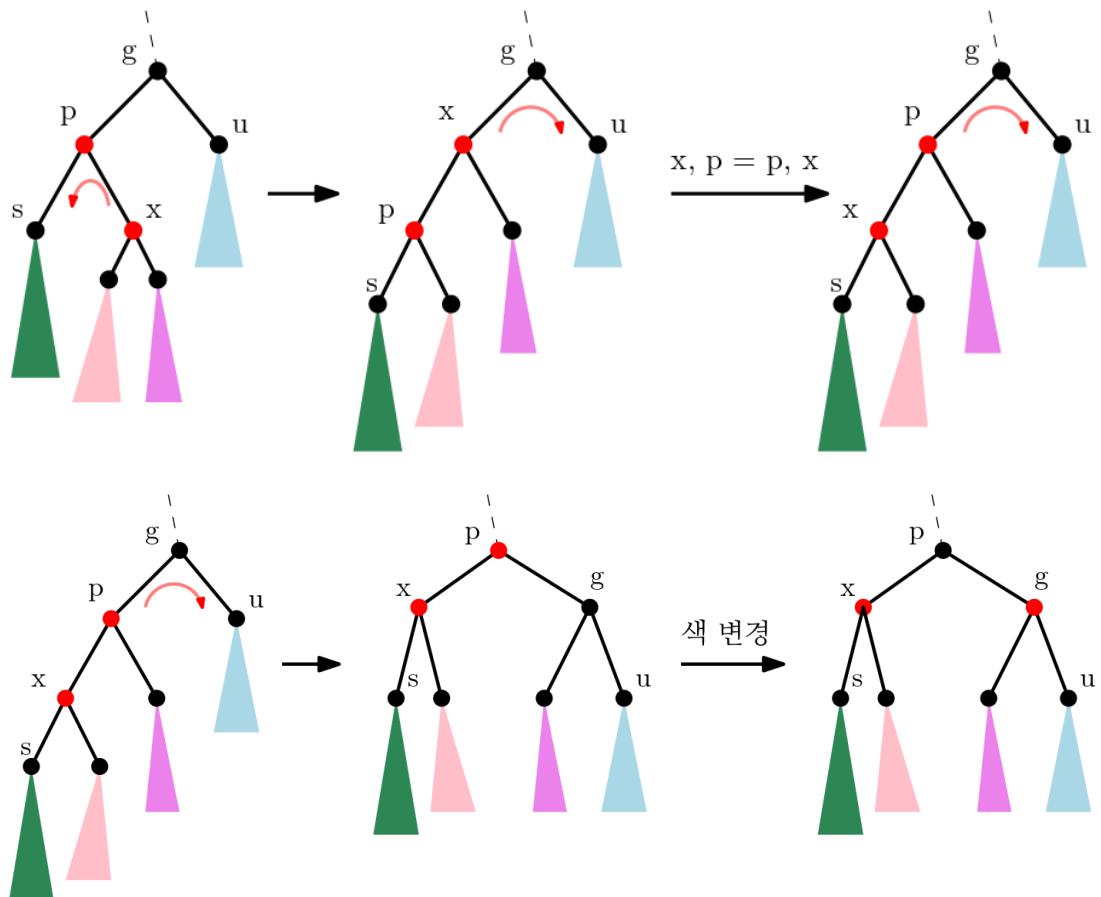


### 경우 2.2: x - p - g가 triangle 모양인 경우

- 노드 p에서 한 번 회전 후, x, p = p, x 수행. 즉, x와 p의 역할을 바꾼 후, while 문 반복한다. 그러면, x-p-g가 선형으로 늘어 서게 되어 경우 2.1로 분류되어 노드 g에서 한 번의 추가 회전을 하게 된다. 따라서 총 두 번의 회전이 발생한다

경우 2.1에서 아래처럼 색 조정을 수행한다

- `p.color = black`, `g.color = red`
- `p.color = black`이 되어 색 조정 완료! `while` 루프를 탈출



- 이렇게 변경하면 5번 조건이 항상 만족되는지 따져보자
- 결국 경우 2.2에서는 2번의 회전만으로 색조정 완료!

- while 문을 탈출한 직후: `T.root.color = black`

### vi. deletion (상당히 복잡하여 자세한 설명은 생략)

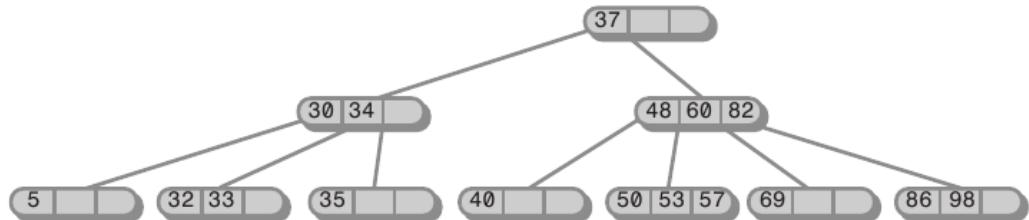
**vii. 수행시간 및 필요 회전 횟수 정리**

| Operations | Worst Case  | 필요 회전 횟수              |
|------------|-------------|-----------------------|
| Space      | $O(n)$      | -                     |
| search     | $O(\log n)$ | -                     |
| insert     | $O(\log n)$ | 2 (2 <- AVL)          |
| delete     | $O(\log n)$ | 3 ( $\log n <- AVL$ ) |

- AVL 트리와의 차이점은 delete 연산에 필요한 회전 횟수가 3번으로 매우 작다는 점

### h. 2-3-4 트리 (red-black 트리의 쌍둥이 트리)

- i. 2-3-4 트리는 (1) 자식노드가 **최소 2개, 최대 4개** 이하이며, (2) 모든 리프노드는 마지막 레벨에 위치해야 하는 탐색트리로 정의된다
  - 단, 이진트리는 아니다
  - 자식 노드의 개수에 따라 2-노드, 3-노드, 4-노드로 구분
  - 리프 노드들은 원형 이중 연결 리스트에 의해 좌-우로 연결되어 있음



#### ii. 트리의 높이는?

- 최대 높이는 노드가 모두 자식을 2개씩 가지는 경우이고, 최소 높이는 자식을 4개씩 가지는 경우이다
- 결국, 2-3-4 트리는  $\log_4 n$  이상,  $\log_2 n$  이하의 높이를 갖게 되어  $h = O(\log n)$ 이 됨을 쉽게 확인할 수 있다

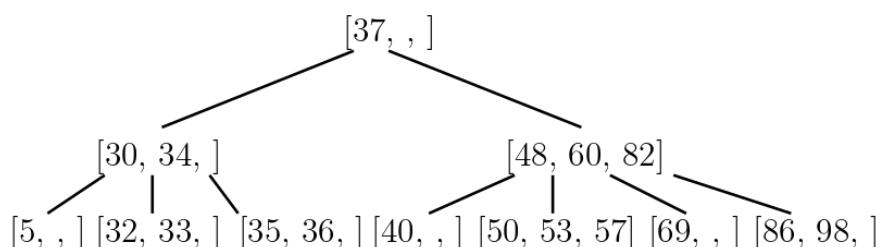
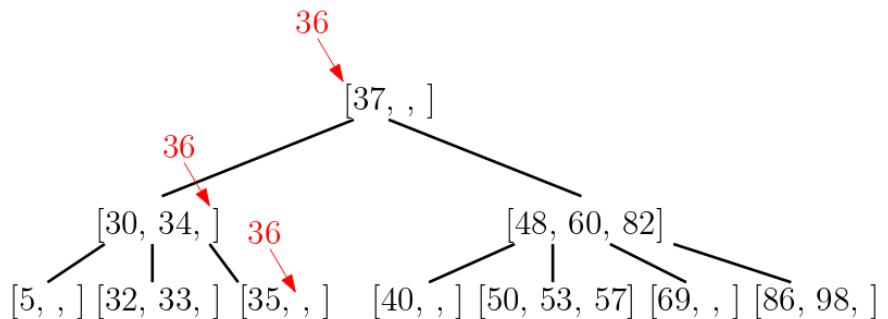
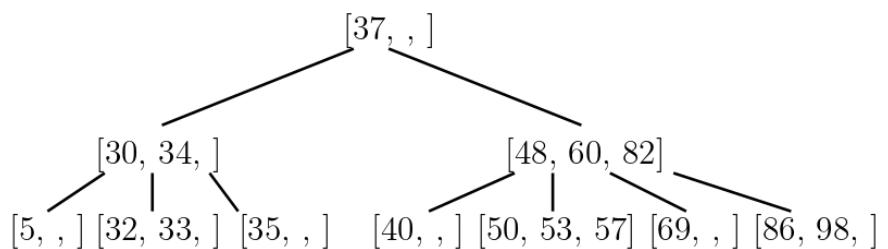
#### iii. search 연산:

- 각 노드는 최대 4개의 자식 노드를 가질 수 있다
- 위의 예에서는 [48, 60, 82] 노드가 네 개의 자식 노드를 가지고 있다. 이 노드에는 3개의 key가 정렬된 상태로 저장되어 있다. 첫 번째 자식 노드에는 48보다 작은 값이, 두 번째 자식 노드에는 48 이상 60 미만의 값들이, 세 번째 자식 노드에는 60 이상 82 미만의 값들이, 네 번째 자식 노드에는 82 이상의 값들이 저장되어 있다는 의미이다.
- 어떤 key 값을 탐색한다는 것은 루트 노드부터 해당 key 값이 속해 있을 자식 노드를 따라 내려가기만 하면 된다. 한 노드에서 어떤 자식 노드를 선택해 내려가는지는 상수 시간에 가능하므로 전체 탐색 시간은  $O(h)$  시간이면 충분하다.  $h = O(\log n)$  이므로  $O(\log n)$  시간에 탐색 가능하다

#### iv. insert 연산:

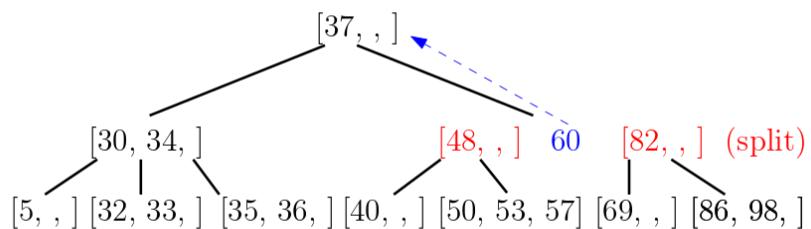
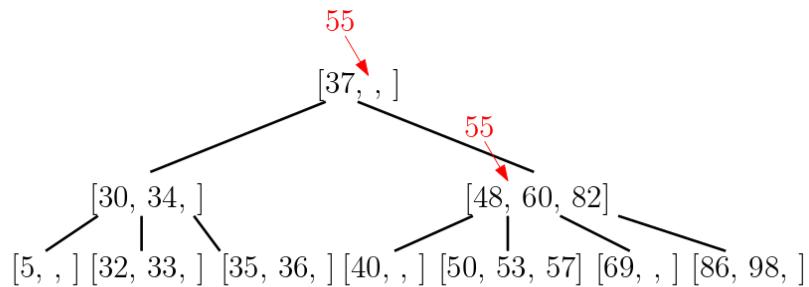
- 루트부터 시작해 삽입할 리프 노드까지 다음과 같은 방법으로 삽입될 리프 노드를 탐색한다 (**항상 리프 노드에 삽입됨**)
- 현재 방문 중인 노드  $v$ 가 4-노드  $[a, b, c]$ 라면 두 개의 2-노드  $[a]$ 와  $[c]$ 로 "split"한다
  - 가운데 값  $b$ 는  $v$ 의 부모 노드에 삽입한다.  $v$ 의 부모노드는  $b$ 를 위한 공간이 항상 있는가? (왜?)

- 만약  $v$ 가 루트 노드였다면,  $b$  만으로 이루어진 새로운 루트 노드 (2-노드 [ $b$ ])를 만든다 ( $\leftarrow$  높이가 1 증가되는 효과)
- 그 다음 자식 노드를 선택해 한 레벨 내려가 탐색을 계속한다
- 현재 방문 중인 노드  $v$ 가 4-노드가 아닌 경우에는:
  - 리프 노드이면 key 값을 순서에 맞게 삽입하고 끝! (4-노드가 아니므로 삽입할 여유가 항상 있음에 유의)
  - 리프 노드 아니라면 다른 레벨의 자식 노드를 선택해 탐색을 계속한다
- 위의 그림에서
  - 36을 삽입해보자

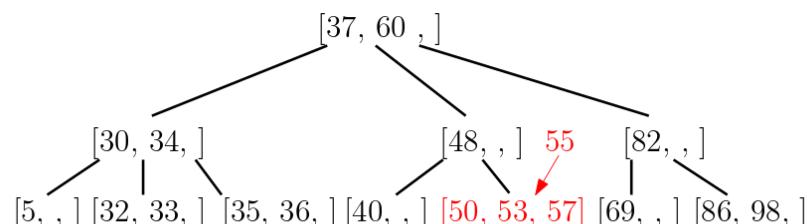
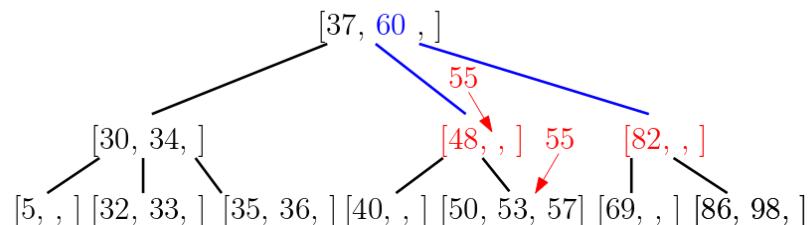


- 삽입될 리프 노드까지 내려오면서 만나는 노드 중에 4-노드가 없음. 따라서 **split**할 필요가 없었음!
- 결국 리프 노드 [35, , ]도 4-노드가 아니므로 36을 저장할 여유가 있음. 최종적으로 [35, 36, ]의 형태로 저장됨
- 이제, 55를 삽입해보자
- 루트 노드부터 리프노드를 향해 내려가다가 첫 4-노드인 [48, 60, 82]를 만나게 된다. 이 노드를 2-노드 두 개 [48]과 [82]로 split하고

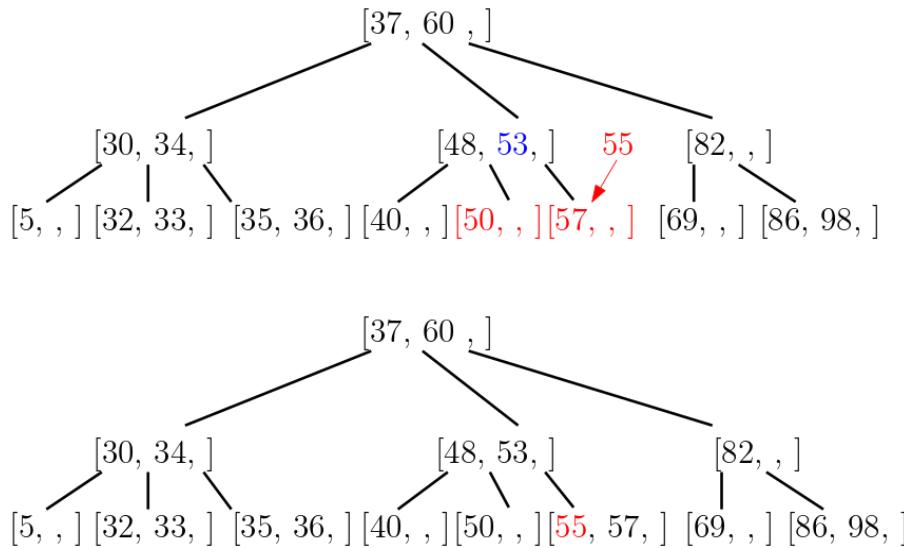
중간의 60은 부모 노드로 올린다 (부모 노드는 4-노드가 아니기 때문에 60의 자리가 항상 있다)



- 이제 [48]의 오른쪽 노드로 내려가면 55가 삽입될 [50, 53, 57] 리프 노드에 도착한다



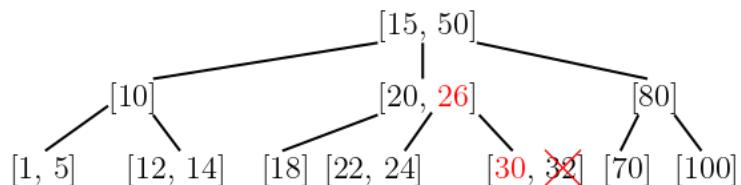
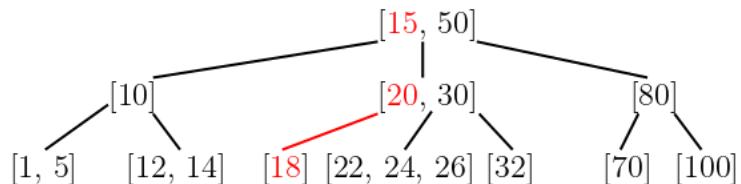
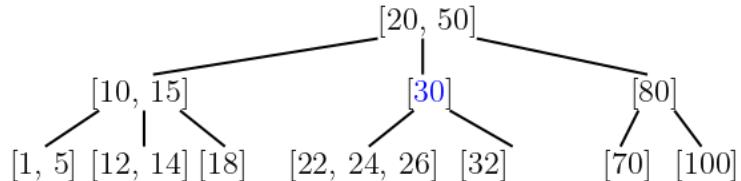
- 이 리프 노드가 4-노드이므로 다시 split해서 [50], (53은 부모 노드로 이동) [57] 두 노드로 분할한다
- 마지막으로 55는 [57] 노드에 삽입된다



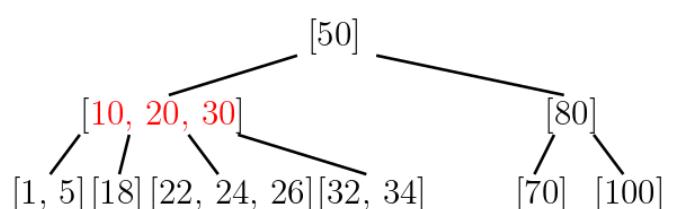
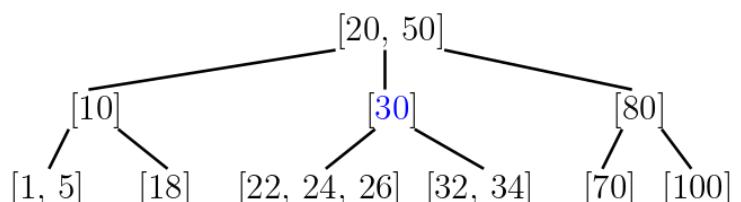
#### V. delete 연산:

- insert보다 복잡 (이유: 삽입은 항상 리프 노드에서 일어나지만, 삭제는 내부 노드의 key 값이 삭제될 때도 있어서)
- key 값이 내부노드에 있는 경우
  - succ(key) 또는 pred(key)를 찾아 swap하면 삭제할 노드가 리프노드가 되기 때문에, 리프노드에 있는 값을 지우는 경우로 변경된다. 따라서 리프노드에 있는 key 값을 지우는 경우만 살펴보면 됨
- key 값이 리프노드에 있는 경우
  - 3-노드 또는 4-노드인 경우: 삭제하더라도 최소 하나 이상의 키가 존재하므로 단순히 삭제만 하면 됨
  - 2-노드인 경우: 지우면 underflow 문제 발생
  - 이를 피하기 위해 루트노드에서 리프노드로 내려가면서 루트 노드가 아닌 2-노드를 만나면 무조건 3-노드 또는 4-노드로 변경함
    - 이 과정에서 트리의 높이가 1 감소할 수 있음
    - 결국, 삭제할 key 값이 저장된 리프노드는 3-노드/4-노드가 되어 단순 삭제 가능
- 2-노드를 3-노드/4-노드로 변경
  - Rotation: 왼쪽 또는 오른쪽 3-노드/4-노드 형제에게서 값을 하나 빌려옴! (물리적인 rotation이 아니라, 값이 회전하듯 연쇄적으로 이동함) 그래서 3-노드가 됨!

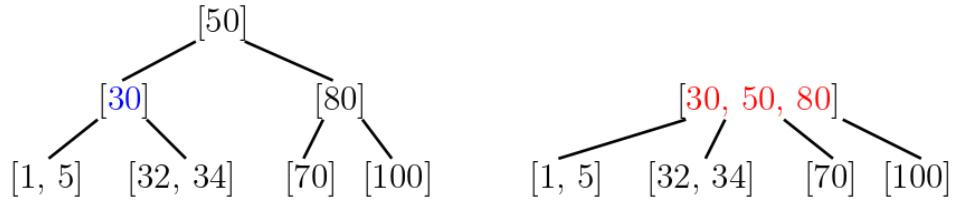
- 아래 그림에서 32를 제거한다고 하자. 루트 노드에서 2-노드인 [30]에 도착한다
- 왼쪽 3-노드 [10, 15]에서 15를 빌려옴! (15가 [20, 50]으로 이동, [20, 50]의 20이 [30]으로 내려감. 대신 [18]이 [20, 50]의 첫 번째 자식 노드가 됨)
- [32] 리프노드에 도착해서도 같은 방식으로 값의 회전이 발생함: [32] → [30, 32]가 되어 32 제거 가능



- Fusion:** 루트 노드가 아닌 2-노드의 왼쪽, 오른쪽 형제가 하필 모두 2-노드라면 회전을 할 수 없게 된다. 자신의 key 값과 왼쪽 (또는 오른쪽) 형제 노드 값 + 부모 노드의 값 하나를 fusion(merge)해서 하나의 4-노드로 만든다 (Q: 만약 부모가 루트 노드고 2-노드라면? → shrink!)



- **Shrink**: 형제 노드가 모두 2-노드이고 루트 노드 역시 2노드인 경우에는 루트가 없어지고 높이가 1이 줄어드는 shrink 연산이 이루어짐



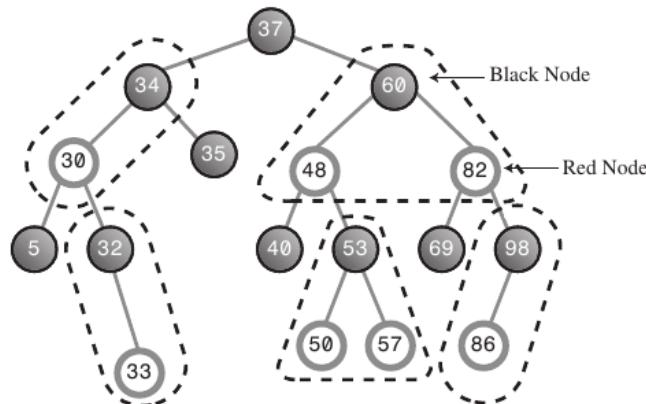
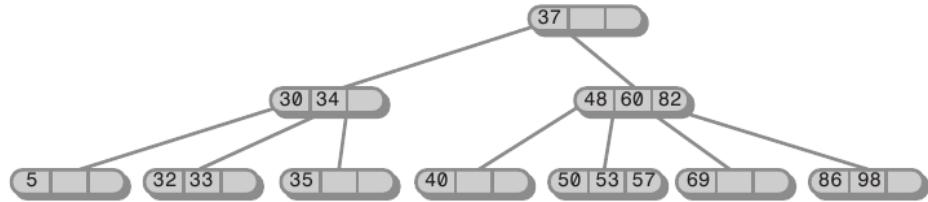
- 각자 예제 트리에서 여러 값을 삽입/삭제하는 연습을 해보자

- vi. 이 트리와 red-black 트리와의 관계는?
- 앞 장의 첫 2-3-4 트리를 red-black 트리로 만들어보자
    - 2-노드는 **black** 노드로 만든다 (예: 노드 37, 5, 40 등)
    - 3-노드는 인접한 두 레벨에 걸쳐 하나의 **black** 노드와 하나의 **red** 자식 노드로 만든다 (예: 노드 [30, 34] → [30] - [34])
    - 4-노드는 인접한 두 레벨에 걸쳐 하나의 **black** 노드와 두 개의 **red** 자식 노드로 만든다 (예: 노드 [48, 60, 82] → [48] - [60] - [82])

이렇게 변환하면 red-black 트리의 5개 조건을 모두 만족하나?

- 첫 레벨에 놓이는 노드가 **black** 노드이기에 루트 노드는 당연히 **black**이므로 2번 조건 만족한다
- 리프 (None) 노드는 **black** 노드로 가정하면 3번 조건도 만족한다
- **red** 노드의 자식 노드는 다음 레벨에 존재하는데, 해당 노드는 2-, 3-, 4-노드의 첫 레벨 노드이므로 무조건 **black**이다. 따라서 4번 조건 만족한다
- 2-, 3-, 4-노드의 첫 레벨 노드는 **black** 노드이고 2-3-4 트리의 정의상 모든 리프 노드가 같은 레벨에 있어야 하므로 임의의 트리에서 리프 노드까지의 **black** 노드의 개수는 같게 된다. 따라서 5번 조건도 만족한다

- 아래 red-black 트리를 2-3-4 트리로 만들어보자



vii. 결국, 2-3-4 트리와 red-black 트리는 1-1 대응 관계가 정의된다

- [복습]  $n$ 개의 노드를 갖는 2-3-4 트리의 높이  $h$ 의 최소, 최대 값은 각각 얼마일까?
- $n$ 개의 노드를 갖는 red-black 트리의 높이  $h'$ 은 최대 얼마일까?  
대응되는 2-3-4 트리의 높이  $h$ 로 표현가능하다!  
[힌트] 2-3-4 트리가 어떤 노드로 구성되는 게  $h'$ 을 가장 크게 하는 경우일까? (정답은 QR 코드에~)



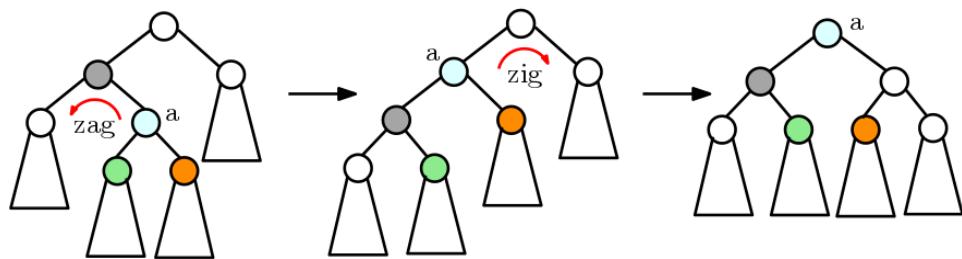
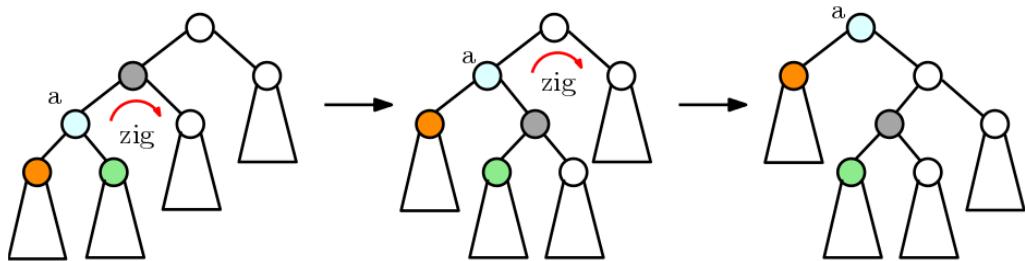
viii. red-black 트리가 2-3-4 트리보다 훨씬 광범위하게 사용된다

### i. Splay 트리

- i. AVL 트리와 red-black 트리에서 삽입과 삭제 연산을 수행하면 트리의 균형 조건이 깨지게 되는데, 회전 등의 방법을 적용하여 강제로 균형을 맞추게 됨
- ii. Splay 트리는 강제로 균형을 맞추지 않고, 한번 탐색되는 key 값이 앞으로도 탐색될 가능성이 높다는 성질(**locality of the access frequency**)을 활용하여 자주 탐색되는 key 값을 가능하면 루트 노드 (또는 루트 노드와 가까운 곳)에 위치시키는 전략을 사용하여, **평균적인 연산 수행 시간을  $O(\log n)$** 으로 유지함. (최악의 경우의 연산 수행시간은 매우 나쁠 수 있음)

#### iii. **Splaying** 연산 정의:

- 어떤 노드 **a**를 splaying 한다는 의미는 아래의 3가지 회전 연산을 반복적으로 적용하여 **a**를 루트노드로 만드는 연산
- 3가지 회전 종류 (zig: right rotation, zag: left rotation)
  - zig 또는 zag (**a**의 부모가 루트인 경우)
  - zig-zig 또는 zag-zag (**a**와 **a**의 부모가 각각 같은 방향 자식: linear 모양)
  - zig-zag 또는 zag-zig (**a**와 **a**의 부모가 서로 다른 방향 자식: triangle 모양)



- iv. **m**번의 search, insert, delete 연산을 섞어서 수행하고 **n**개의 노드가 splay 트리에 저장되었다면, 총  $O(m\log n)$  시간이면 충분하다는 증명이 존재
- v. 결국 1번의 연산당 평균  $O(\log n)$  시간이 필요. 이러한 시간 분석 방법을 **amortized time complexity** analysis라고 부른다
- vi. 세 연산의 기본 내용은 접근(access)한 값(노드)은 (이후에 참조될 가능성이 높기에) 항상 루트 노드로 splaying 하는 것이다. search(key)에서 key 값이

존재하면 해당 노드를 splaying한다. 만약, 없다면 key의 predecessor 또는 successor 노드를 splaying 할 수도 있다

#### vii. Pseudo 코드:

Splay 트리 클래스의 부모 클래스로 이진탐색트리 클래스를 지정하자

```

class splayTree(BST): # BST가 부모 클래스
 # BST의 모든 것(초기화함수포함)을 물려 받음

 def splay(self, v):
 while v and v.parent:
 p = v.parent
 g = p.parent

 # Case 1: p가 루트인 경우 (zig)
 if gp is None:
 if p.left is v:
 x = rotateRight(gp)
 else:
 x = rotateLeft(gp)

 # Case 2: zig-zig (v - p - gp가 모두 같은 방향)
 elif p.left is v and gp.left is p:
 rotateRight(p)
 x = rotateRight(g)
 elif p.right is v and gp.right is p:
 rotateLeft(p)
 x = rotateLeft(g)

 # Case 3: zig-zag (v - p, p - gp가 반대 방향)
 elif p.left is x and gp.right is p:
 rotateRight(p)
 x = rotateLeft(g)
 else: # p.right is x and gp.left is p
 rotateLeft(p)
 x = rotateRight(g)

 return x # 최종적으로 x가 루트

```

```

def search(self, int key):
 # 부모클래스 BST의 search 함수를 호출함 (왜?)
 v = super(splayTree, self).search(key)

 if v: # splay v
 self.root = self.splay(v)
 else: # 필요하다면
 w = pred(key) or succ(key)
 self.root = self.splay(w)
 return v

def insert(self, key) {
 # 부모클래스 BST의 insert 함수를 호출함 (왜?)
 1. v = super(splayTree, self).insert(key)
 2. self.root = self.splay(v)

def delete(self, x):
 1. splay(x)를 한다 → x가 루트가 됨!
 2. L과 R을 각각 x의 왼쪽 부트리, 오른쪽 부트리라 하자
 3. L이 empty 아니면, L에서 가장 큰 key 값의 노드 m을 탐색
 3.1 splay(m)을 한다 → m이 루트가 됨
 m.right = None
 3.2 R을 m의 오른쪽 자식으로 삽한다
 m.right = R, R.parent = m, self.root = m
 4. L이 empty라면, R을 루트로 한다
 R.parent = None, self.root = R

```

- 
- j. [💪] [해보기-medium] Splay 트리의 연산의 코딩을 완성해보자!

## [요약] 현재까지 살펴본 자료구조들의 수행시간 비교 표

[출처: <http://bigocheatsheet.com/>]

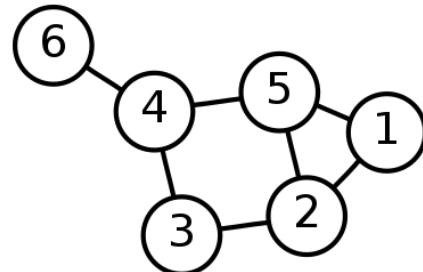
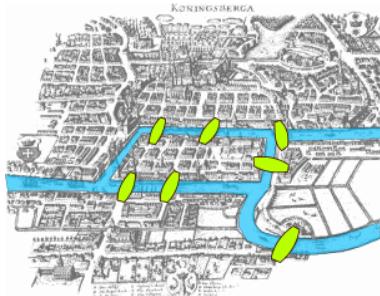
### Common Data Structure Operations

| Data Structure     | Time Complexity |           |           |           |           |           |           |           | Space Complexity |  |
|--------------------|-----------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|------------------|--|
|                    | Average         |           |           |           | Worst     |           |           |           |                  |  |
|                    | Access          | Search    | Insertion | Deletion  | Access    | Search    | Insertion | Deletion  |                  |  |
| Array              | O(1)            | O(n)      | O(n)      | O(n)      | O(1)      | O(n)      | O(n)      | O(n)      | O(n)             |  |
| Stack              | O(n)            | O(n)      | O(1)      | O(1)      | O(n)      | O(n)      | O(1)      | O(1)      | O(n)             |  |
| Queue              | O(n)            | O(n)      | O(1)      | O(1)      | O(n)      | O(n)      | O(1)      | O(1)      | O(n)             |  |
| Singly-Linked List | O(n)            | O(n)      | O(1)      | O(1)      | O(n)      | O(n)      | O(1)      | O(1)      | O(n)             |  |
| Doubly-Linked List | O(n)            | O(n)      | O(1)      | O(1)      | O(n)      | O(n)      | O(1)      | O(1)      | O(n)             |  |
| Skip List          | O(log(n))       | O(log(n)) | O(log(n)) | O(log(n)) | O(n)      | O(n)      | O(n)      | O(n)      | O(n log(n))      |  |
| Hash Table         | N/A             | O(1)      | O(1)      | O(1)      | N/A       | O(n)      | O(n)      | O(n)      | O(n)             |  |
| Binary Search Tree | O(log(n))       | O(log(n)) | O(log(n)) | O(log(n)) | O(n)      | O(n)      | O(n)      | O(n)      | O(n)             |  |
| Cartesian Tree     | N/A             | O(log(n)) | O(log(n)) | O(log(n)) | N/A       | O(n)      | O(n)      | O(n)      | O(n)             |  |
| B-Tree             | O(log(n))       | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n)             |  |
| Red-Black Tree     | O(log(n))       | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n)             |  |
| Splay Tree         | N/A             | O(log(n)) | O(log(n)) | O(log(n)) | N/A       | O(log(n)) | O(log(n)) | O(log(n)) | O(n)             |  |
| AVL Tree           | O(log(n))       | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n)             |  |
| KD Tree            | O(log(n))       | O(log(n)) | O(log(n)) | O(log(n)) | O(n)      | O(n)      | O(n)      | O(n)      | O(n)             |  |

## 6. Graph 자료구조와 기초 그래프 알고리즘

### 1. 두 노드 사이의 관계가 있는 경우 에지로 연결하여 표현하는 추상적이고 일반적인 자료구조

- a. 트리는 사이클이 없는 그래프이고, 연결리스트는 하나의 경로로 이루어진 트리이므로, 가장 단순한 형태의 그래프이다



### 2. 그래프 $G = (V, E)$

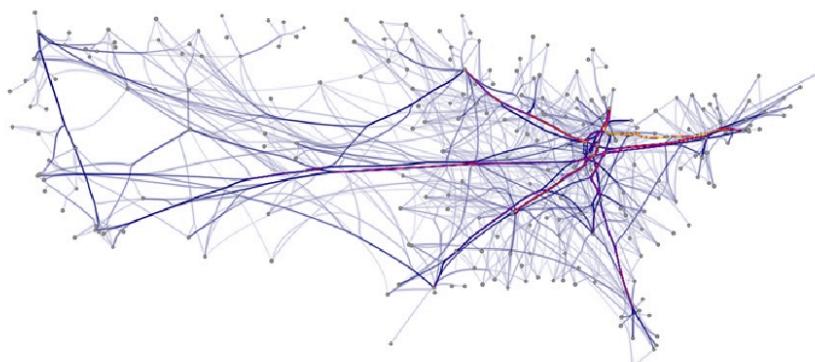
- a.  $V$  = 노드(node) 또는 정점(vertex) 집합을 의미. 그래프 이론 분야에서는 "정점" 선호
- b.  $E$  = 두 노드의 쌍으로 정의. 만약  $(u, v) \in E$ 라면 노드  $u$ 와  $v$ 가 서로 (방향이 없는) 에지로 연결되어 있다고 한다
- c. 위의 왼쪽 그림: (Wikipedia/Graph) Euler의 Königsberg의 다리로 유명

The paper written by [Leonhard Euler](#) on the [Seven Bridges of Königsberg](#) and published in 1736 is regarded as the first paper in the history of graph theory

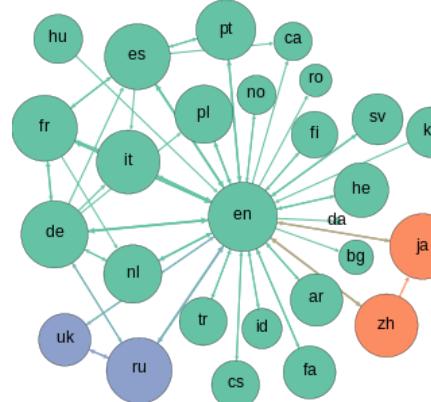
- d. 위의 오른쪽 그림: 지역 6곳을 노드로 정의하고 두 지역을 연결하는 다리가 존재하면 에지로 정의할 수 있다. 그러면 추상적인 "그래프"가 정의된다. 노드와 에지 집합은 아래와 같다
  - i.  $V = \{1, 2, 3, 4, 5, 6\}$  # 노드 번호를 0부터 시작해도 된다
  - ii.  $E = \{(1,2), (3,2), (1,5), (2,5), (4, 5), (6, 4), (3,4)\}$

### 3. 응용분야(applications)

- a. 주위의 거의 모든 관계를 그래프로 표현 가능하기에 응용 분야는 무궁무진하다
- b. Computer Science, Linguistics, Physics and Chemistry, Biology, Social Science



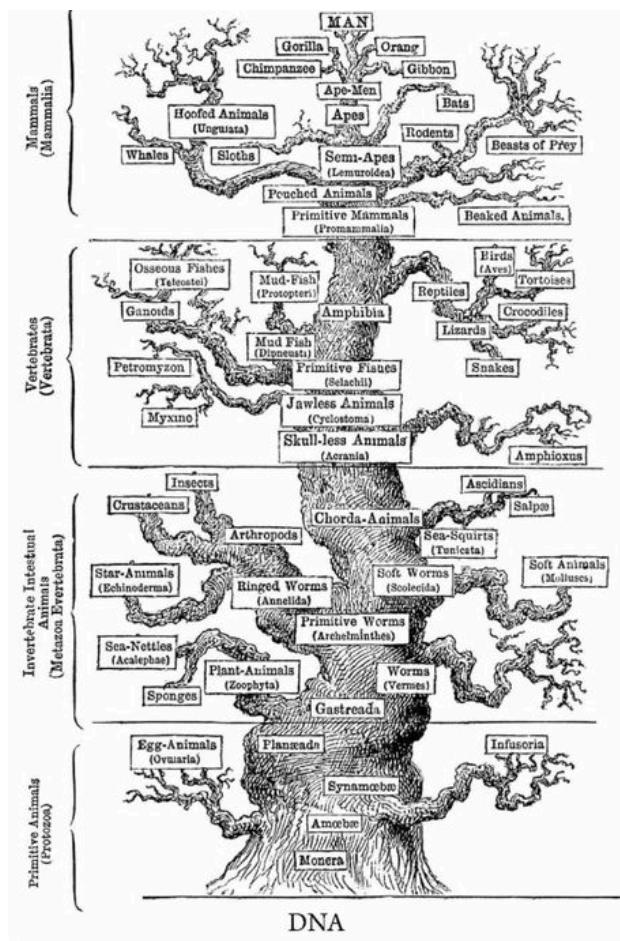
[US domestic-flights map from [Graphminator](#)]



[graph on web sites from Wikipedia]

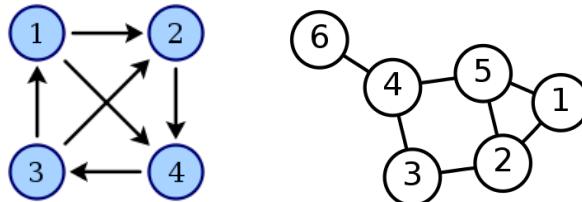


[subway map]

[Phylogenetic tree: [출처](#)]

#### 4. 기본 용어 (basic graph terminology)

- a. 정점(vertex), 노드(node)
- b. 에지(edge), 링크(link)
  - i. 방향/무방향 에지 (directed/undirected edge)
- c. 무방향 그래프, 방향 그래프
  - i. 무방향 에지로 구성된 그래프와 방향 에지로 구성된 그래프



- d. 분지수(degree)
  - i. in-degree:  $\text{in-deg}(u)$ : 노드  $u$ 로 들어오는 방향 에지의 개수
  - ii. out-degree:  $\text{out-deg}(u)$ : 노드  $u$ 에서 나가는 방향 에지의 개수
  - iii. degree of vertex:  $\deg(u)$ : 노드  $u$ 에 연결된 무방향 에지의 개수
  - iv. degree of graph:  $\deg(G)$ : 무방향 그래프  $G$ 의 최대 노드 분지수
- e. 인접성(adjacency, incidence)
  - i. 에지  $(u, v)$ 가 존재하면,  $u$ 와  $v$ 는 서로 인접(adjacent)하다고 말함
  - ii. 에지  $e = (u, v)$ 에 대해,  $e$ 는  $u$ 와  $v$ 에 인접(incident)하다고 말함
- f. 경로(path)
  - i. 한 노드에서 다른 노드로 연결되는 선형 경로로 경로에 포함된 노드는 모두 달라야 한다 (방향 그래프에서의 경로는 한 방향 - 일방 통해 경로를 의미한다)
  - ii. 경로의 길이는 에지에 가중치 값이 없는 경우는 경로의 에지의 개수로 정의되고, 가중치 값이 있는 경우는 에지의 가중치 합으로 정의된다
- g. 에지/정점 가중치(edge/vertex weight)
  - i. 가중치는 주로 비용(cost)의 역할을 한다. 예를 들어, 그래프의 두 노드를 연결하는 경로의 길이는 경로에 포함된 가중치의 합으로 정의한다. 최단 경로는 경로 길이가 가장 짧은 경로로 정의된다
- h. 사이클(cycle)
  - i. 출발 노드와 도착 노드가 동일한 경로로 정의된다. 즉, 한 노드에서 출발해 해당 노드로 도착하는 원형 경로를 의미한다
- i. 트리(tree): 사이클이 없는 연결 그래프
  - i. 포리스트(forest): 하나 이상의 연결 트리의 집합
- j. 부그래프(subgraph)
  - i. 그래프의 정점 집합의 부분 집합과 그 부분 집합에 속하는 정점 사이에 정의된 에지 집합으로 정의되는 부분 그래프이다. 그래프가 트리인 경우에는 부트리(subtree)라고 부른다
  - ii. 집합(set)과 부분집합(subset)의 관계와 유사하다

k. 연결성(connectedness)

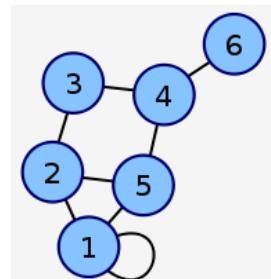
- i. 두 노드가 연결.connected) 되어 있다는 것은 두 노드를 연결하는 경로가 존재한다는 뜻이다
- ii. 그래프가 연결.connected) 되어 있다는 것은 그래프의 모든 노드 쌍이 연결되어 있다는 뜻이다
- iii. 그래프의 연결 성분(connected component)은 그래프의 연결된 부그래프를 의미한다. 그래프는 한 개 이상의 연결 성분으로 구성된다

l. 신장 그래프(spanning subgraph)

- i. 부그래프 중에서 그래프의 모든 노드가 등장하는 부그래프를 의미한다

5. 그래프 표현 방법 (graph representation)

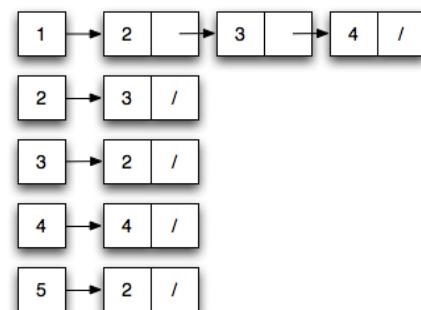
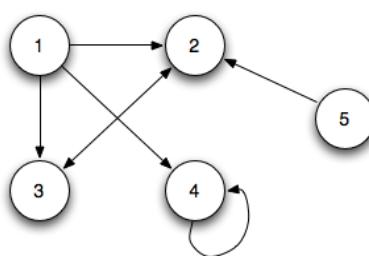
- a. 인접행렬(adjacency matrix): 인접성을 행렬(2차원 배열/리스트)로 표현
- b. 인접리스트(adjacency list): 정점에 인접한 에지만을 연결리스트로 표현



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Python 이차원 리스트 표현:

```
G = [[1,1,0,0,1,0], [1,0,1,0,1,0], [0,1,0,1,0,0],
 [0,0,1,0,1,1], [1,1,0,1,0,0], [0,0,0,1,0,0]]
```



Python 리스트로 표현:

```
G = [[], [2, 3, 4], [3], [2], [4], [2]] # 정점 번호가 1번부터
```

c. 표현법에 따른 기본 연산 시간 비교 (표현법에 따른 장단점 파악 필요)

i.  $n = |V|$  = 노드 개수,  $m = |E|$  = 에지 개수

ii. Python 리스트에 저장한다고 가정

| 기본연산                                  | 인접행렬                                                       | 인접리스트                                                                         |
|---------------------------------------|------------------------------------------------------------|-------------------------------------------------------------------------------|
| ( $u, v$ )가 에지인가?                     | $G[u][v] == 1$<br>$O(1)$                                   | $G[u].search(v) != None$<br>$O(n)$                                            |
| $u$ 의 인접한 모든<br>에지 ( $u, v$ )에<br>대해: | $G[u][v]$ <b>for</b> $v$ <b>in</b><br>$range(n)$<br>$O(n)$ | <b>for</b> edge <b>in</b> $G[u]$<br>$O(\deg(u))/O(\text{out-deg}(u))$         |
| 새 에지 ( $u, v$ )<br>삽입                 | $G[u][v] = 1$<br>$O(1)$                                    | $G[u].pushFront(v)$<br>$O(1)$                                                 |
| 에지 ( $u, v$ ) 제거                      | $G[u][v] = 0$<br>$O(1)$                                    | $x = G[u].search(v)$<br>$G[u].remove(x)$<br>$O(\deg(u))/O(\text{out-deg}(u))$ |
| $G$ 를 위한 메모리                          | $O(n^2)$                                                   | $O(n + m)$                                                                    |

d. 질문: 인접 리스트의 각 노드의 에지 리스트를 위해 연결 리스트 (또는 파이썬  
리스트)가 아닌 해시 테이블을 사용하면 기본 연산 시간은 어떻게 될까?

i. 해시 테이블의 삽입, 삭제, 탐색은 충돌 확률이 테이블 슬롯 수에 반비례하고  
빈 슬롯의 비율이 일정 수준으로 유지된다는 가정하에 평균  $O(1)$ 이다

## 6. Traversal: DFS, BFS



(1/2)



(2/2)

### a. DFS(Depth First Search: 깊이 우선 방문)

- i. 현재 방문 노드에서 방문하지 않은 이웃 노드가 있다면 방문하는 방식

#### 1. 재귀적인 방식

```
RecursiveDFS(v): # visiting v now!
 mark v as visited node
 for each edge (v, w): # 에지 순서는 임의로~
 if w is unmarked: # 인접한 노드 w가 미방문이면
 RecursiveDFS(w)
```

#### 2. 비재귀적인 (반복) 방식

##### **IterativeDFS(s):**

```
stack.push(s) # 나중에 방문할 노드를 스택에 대기시킴
while stack is not empty:
 v = stack.pop()
 if v is unmarked:
 mark v as visited node
 for each edge (v, w):
 if w is unmarked:
 stack.push(w)
```

- ii. **수행시간:** 각 노드 v를 방문할 때, v에 인접한 미방문 노드 w가 스택에 push되고 적당한 때에 다시 pop된다. 즉, 에지 (v, w)에 대해 한 번의 push와 한 번의 pop이 이루어진다고 해석해도 된다. push, pop 스택 연산은  $O(1)$  시간이 필요하므로  $O(m)$  시간이면 충분하다. 추가로 노드 개수만큼의 시간 역시 필요하므로 정리하면  $O(n + m)$  시간이 된다

- iii. [중요] DFS를 하면서 각 노드의 첫 방문 시간과 최종 방문 시간 기록하기 + 방문 순서 (부모 노드) 기록하기

#### 1. 세 개의 리스트를 준비

- o  $\text{pre}[v]$  = 노드 v를 첫 방문한 시간을 저장
- o  $\text{post}[v]$  = 노드 v를 통해 방문할 이웃 노드가 없는 시간, 즉 v의 입장에서 DFS가 완료되는 시간을 저장
- o  $\text{parent}[v]$  = 노드 v를 방문하기 직전의 노드 (v의 입장에서 보면, 부모 노드)를 저장 (예: 노드 u를 거쳐 v를 방문했다면  $\text{parent}[v] = u$ 가 됨)

```

DFS(v):
 # (*)
 visited[v] = True # mark v as a visited node
 pre[v] = curr_time # record the first visit time
 curr_time += 1
 for each edge (v, w):
 if visited[w] == False: # w is not visited yet
 parent[w] = v
 DFS(w) # Recursive version
 post[v] = curr_time # record the finish time
 curr_time += 1

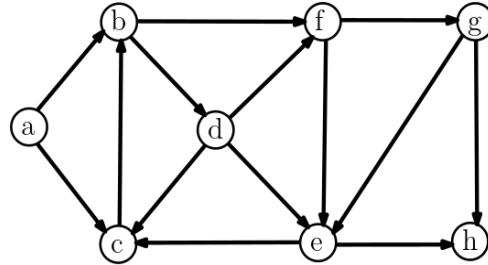
DFSAll(G):
 for all nodes v:
 visited[v] = False # initially not visited at all
 parent[v] = -1 # initially all parents are -1
 curr_time = 1 # time starts from 1
 count = 0 # 연결 성분 번호 나타내는 변수
 for all nodes v:
 if visited[v] == False: # v is not marked → DFS at
 v
 count += 1 # 연결 성분 번호 증가
 DFS(v)

비자규 코드: 아래 코드에서 pre와 post 값은 어떻게 지정해야 할까?
DFS(s):
 stack.push((None, s)) # tuple (parent, curr_node)
 while stack is not empty:
 p, v ← stack.pop()
 if visited[v] == False:
 visited[v] = True
 parent[v] = p
 for each edge (v, w):
 if visited[w] == False:
 stack.push((v, w))

```

- iv. 예제: 노드 a부터 시작하고, 노드 번호가 작은 인접 노드부터 방문한다고 가정하자

1. 아래 그래프 DFS 순서 (pre 값의 오름차순): a, b, d, c, e, h, f, g
2. pre, post 시간은 다음 페이지 DFS 트리와 함께 표시되어 있음



## v. DFS 트리

1. DFS 시작 노드가 루트 노드가 되며, 방문을 하면서 정의되는 부모 노드  
- 자식 노드 관계에 의해 정의되는 트리가 DFS 트리이다. 신장 트리(spanning tree)이다. 즉, 그래프의 모든 노드가 연결된 트리이다.
2. DFS 트리의 나타난 에지  $(u, v)$ 를 네 개의 타입으로 구별한다

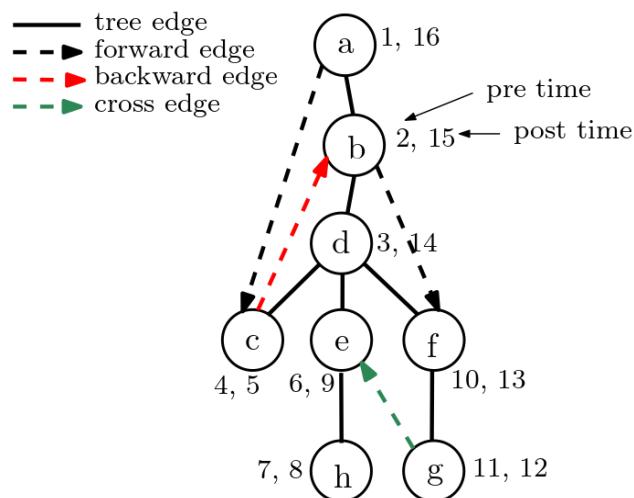
**tree edge** - DFS 트리를 구성하는 에지

**forward edge** - 트리 에지가 아닌 조상  $\rightarrow$  자손으로 향하는 에지

**backward edge** - 트리 에지가 아닌 자손  $\rightarrow$  조상으로 향하는 에지

**cross edge** - DFS 트리, forward, backward도 아닌 그래프의 에지

위의 왼쪽 그래프에 대한 DFS 트리와 에지 종류는 아래 그림과 같다  
(주의: 트리 에지는 모두 그렸지만 나머지 에지는 일부만 표시했음)



3. 무방향 그래프에서는 backward 에지와 forward 에지는 구별되지 않는다
4.  $\text{pre}[u]$ ,  $\text{pre}[v]$ ,  $\text{post}[u]$ ,  $\text{post}[v]$  값의 대소 관계를 통해, 구별 가능하다. 어떻게 하면 될까?

tree edge:

$\text{parent}[v] = u$

forward edge:

$\text{pre}[u] < \text{pre}[v] < \text{post}[v] < \text{pre}[u]$

backward edge:

$\text{pre}[v] < \text{pre}[u] < \text{post}[u] < \text{pre}[v]$

cross edge:

$\text{pre}[v] < \text{post}[v] < \text{pre}[u] < \text{post}[u]$

5. 결국, pre 값과 post 값의 대소 관계를 통해 상수 시간에 **forward**, **backward**, **cross** 에지를 구분할 수 있다
6. 이런 구분이 어떻게 유용하게 쓰일 수 있을까?
  - 무방향/방향 그래프의 사이클(cycle)이 있는지 검사하고 싶을 때에는 backward 에지가 존재하지만 검사하면 된다. 왜? backward 에지는 자손 노드에서 조상 노드로 향하는 에지이다. 그런데 조상 노드에서 자손 노드까지는 트리 에지로 연결된 경로가 존재한다. 결국 조상 노드와 자손 노드를 연결하는 사이클이 존재한다는 의미이다

## vi. DFS 응용

1. **연결 성분**(connected component)을 구분해서 방문하기
  - 연결된 부그래프를 의미하며 그래프는 하나 또는 그 이상의 연결 성분으로 구성된다
  - 연결 성분을 구별해서 구하는 문제는 기본적인 그래프 문제로 연결 성분의 연결 정도에 따라 다르게 정의하기도 한다 (예: 2-connected component 등)
  - DFS를 그대로 이용해서, 연결 성분의 번호를 count 전역 변수를 나타낸다. 각 노드가 속한 연결 성분의 번호를 기록하기 위해 comp라는 리스트를 준비한다
  - 재귀 DFS 코드의 첫 줄에 `comp[v] = count` 기록하면 된다
2. **미로 탈출**: 이 경우엔 동, 서, 남, 북으로 모두 이동 가능!
  - DFS는 backtracking 방법과 동일하다!!
  - 미로는 이차원 리스트 M에 저장되어 있음
  - 입력 형식:
    - i. 1은 장애물, 0은 빈 칸 의미
    - ii. 바깥쪽 경계는 모두 1로 하여 외부로 나가지 못함
    - iii.  $7 \times 7$  미로 예: (1, 1)이 입구, (5, 5)가 출구라는 의미

```

 7
 1 1 5 5
 1111111
 1000001
 1111101
 1000101
 1011101
 1000001
 1111111

```
  - 출력 형식:
    - i.  $s = \text{입구 표시}, f = \text{출구 표시}, * = \text{탈출 경로 표시}$

```

 1111111
 1s****1
 11111*1
 10001*1

```

```

10111*1
10000e1
1111111

find_way_from_maze(r, c) # 현재 칸 (r, c) 방문 중

 visited[r][c] = True
 if (r, c) == exit:
 return True

 if 동쪽 이웃 칸이 빈 칸이고 미방문이라면:
 if find_way_from_maze(r, c+1):
 M[r][c+1] = '*'
 return True
 if 남쪽 이웃 칸이 빈 칸이고, 미방문이라면:
 ...
 if 서쪽 이웃 칸이 빈 칸이고, 미방문이라면:
 ...
 if 북쪽 이웃 칸이 빈 칸이고, 미방문이라면:
 ...
 return False

n = 미로 크기 (가장 자리 경계 제외)
입구 (sx, sy), 출구 (ex, ey) 입력, 미로 M 입력
visited = 초기화
find_way_from_maze(1, 1) # 호출

```

### vii. 사이클 찾기

#### 1. 무방향 그래프에서는?

- backward 에지가 존재하는지 검사하면 된다
- **방법 1:** pre, post 시간을 구해서 backward 에지 존재하는지 검사
- **방법 2:** DFS에서  $v \rightarrow w$ 로 갈 때,  $\text{visited}[w] == \text{True}$ 라면  $w$ 를  $v$ 를 방문하기 전에 방문을 이미 한 후에  $v$ 에 도달했다는 의미다. 즉,  $w$ 에서  $v$ 까지의 경로가 존재한다는 것이고 다시  $v \rightarrow w$  에지가 있으므로 사이클이 존재함을 알 수 있다. (사실,  $v \rightarrow w$  에지는 backward 에지이다)

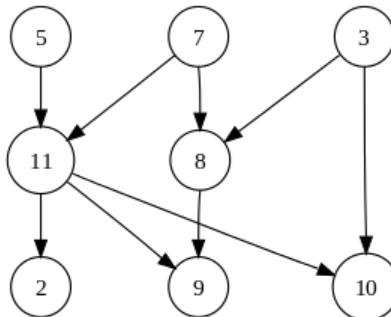
#### 2. 방향 그래프에서는?

- 무방향 그래프에서의 알고리즘과 원칙적으로 같다
- 사이클이 없는 방향 그래프를 DAG(Directed Acyclic Graph)라 부르고 매우 많이 분야에서 등장한다

- DAG라면, outgoing에지만 있는 노드와 incoming에지만 있는 노드가 반드시 하나 이상 존재해야 한다. outgoing에지만 있는 노드를 source 노드라 부르고, incoming에지만 있는 노드를 sink 노드라 부른다

viii. **위상 정렬** (Topological Sort): DAG의 노드들을 순서에 따라 정렬해보자

- DAG는 아래 그래프와 같이 사이클이 없는 방향 그래프이다. 이 DAG에는 세 개의 source 노드가 세 개의 sink 노드가 존재한다
- 부품을 조립하는 조립 공정을 그래프로 표현하면 DAG가 된다. 소스 노드에서 다음 노드로 부품을 전달하고, 해당 노드에서는 incoming에지를 통해 전달받은 부품을 조립해 outgoing에지를 통해 다음 노드들로 전달한다. 이런 공정에서는 사이클이 존재할 수 없다. 존재한다면 공정이 사이클에서 끝나지 않고 영원히 반복될 것이기 때문이다
  - 아래 그래프에서 노드 11은 노드 5와 7에서 부품을 받아 조립해 결과물을 노드 2, 9, 10에 전달한다
  - 노드 5, 7, 3은 들어오는 에지가 없으므로 소스 노드이고, 2, 9, 10은 나가는 에지가 없으므로 싱크 노드이다



5, 3, 7, 8, 11, 2, 9, 10

3, 5, 7, 8, 11, 9, 10, 2

...

- 조립 과정을 정렬을 해보자. 값의 대소 관계가 정의될 때 정렬이 정의된다. 정수와 실수 값은 정렬할 수 있는 건, 두 수의 대소가 정확히 정의되기 때문이다. 문자열도 사전순서에 따라 대소가 정의되므로 정렬이 가능하다. 조립 과정의 정렬 역시 두 노드의 대소가 정의되어야 한다. 노드 v에서 노드 w로 에지가 있다면 v의 부품이 먼저 완성되어야 그 부품을 w가 받아 조립을 시작할 수 있다. 따라서 순서적으로 v가 w가 앞선다. 즉 v < w의 대소 관계가 성립한다
- 문제는 모든 노드 쌍 사이에 대소 관계가 정의되지 않는다는 것이다. 예를 들어, 소스 노드 사이에는 대소 관계가 없다. 마찬가지로 싱크 노드 사이에도 없다. 노드 5와 8 사이에도 대소 관계가 정의되지 않는다. 이렇게 모든 쌍에 대소 관계가 정의되지 않고 일부에 대해서만

정의되는 관계를 partial order라 부른다. DAG은 일종의 partial order를 나타낸다고 말할 수 있다

5. Partial order에서도 정렬을 할 수 있다. 그러나 정렬의 결과가 하나 이상일 수 있다. 위의 그래프의 정렬 순서는 아래와 같다

$\{5, 3, 7\}, \{8, 11\}, \{2, 9, 10\}$

6.  $\{5, 3, 7\}$ 의 노드가  $\{8, 11\}$ 의 노드보다는 무조건 먼저 와야 하고,  $\{8, 11\}$ 의 노드가  $\{2, 9, 10\}$ 의 노드보다 먼저 와야 한다. 그러나  $\{5, 3, 7\}$ 의 노드 사이에는 선후관계가 없기 때문에 어떻게 나열되도 상관없다.  $\{8, 11\}, \{2, 9, 10\}$ 의 노드 사이에도 순서는 상관없다. 따라서 아래 정렬은 모두 올바른 순서이다

5, 3, 7, 8, 11, 2, 9, 10  
3, 5, 7, 8, 11, 9, 10, 2

...

7. DAG에서의 정렬을 위상 정렬(topological sort, topological ordering)이라 부른다

#### 8. Algorithm 1:

소스 노드들부터 차례대로 다음에 조립이 가능한 노드들을 찾아가는 방법: 가장 먼저 오는 노드는 in-deg가 0인 노드들이고 이 노드는 소스 노드들이다. 이 소스 노드들에서 나가는 에지(outgoing edge)를 지운 후, 노드들의 in-deg를 업데이트한다. 다시 in-deg가 0이 되는 노드들이 다음에 오면 된다. 이 방식으로 싱크 노드들이 남을 때까지 반복하면 된다

##### TopologicalSort(G):

```
S = []
for i in range(|V|):
 v ← any source node in G # how to find v?
 S.append(v)
 delete all outgoing edges from v
 # in-degrees of vertices are updated accordingly
return S
```

단점: "v의 outgoing 에지를 지운다"는 부분을 어떻게 구현해야 하나? in-deg를 일관성있게 update해야 하고, in-deg가 0이 되는 노드들의 리스트에 관리해야 한다

#### 9. Algorithm 2:

**힌트**: DFS에서 가장 처음으로 DFS가 완료되는 노드는 post 값이 가장 작은 노드이다. 나가는 에지가 없기 때문이다. 따라서 이 노드는 무조건

sink 노드이다! 결국, post 값이 작아지는 순서 (내림차순)로 나열하는 것이 위상 정렬 순서이다? YES!

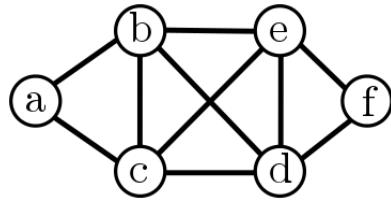
**TopologicalSort(G):**

1. add a new source node  $s$  with edge to all vertices
2.  $\text{DFS}(s)$  # post 값도 동시에 계산함
3. ( $s$ 를 제외하고) post 값의 내림차순으로 노드 번호를 출력

10. Algorithm 1 vs. Algorithm 2 (수행시간을 비교해보자)

b. **BFS**(Breadth First Search: 너비 우선 방문)

- i. 현재 방문 노드에서 방문하지 않은 이웃 노드를 차례대로 방문하는 방식



- ii. 위의 예: 노드 a부터 출발한다면,  $a \rightarrow \{b, c\} \rightarrow \{e, d\} \rightarrow e$  순으로 출발 노드부터의 거리 순으로 노드들을 방문하게 된다. 즉, 출발 노드로부터 거리가 1인 (에지 하나로 갈 수 있는) 노드를 모두 방문하고, 거리가 2인 노드를 (거리가 1인 노드들의 이웃 노드들)을 모두 방문하는 식으로 진행한다
- iii. 큐(queue)  $Q = \{a\}$ 를 준비하여,  $v = Q.dequeue()$ 하여 노드  $v$ 를 방문하고,  $v$ 의 인접한 미방문 노드  $w$ 를 모두  $Q.enqueue(w)$ 하는 단계를 반복하면 된다
- iv. 리스트  $dist[v]$ 는 출발 노드로부터 거리 (출발 노드와  $v$ 를 잇는 경로의 에지 개수)가 저장된다

**BFS(G):**

```

visited = [False] * n # BFS 중간에 방문했는지를 기록
parent = [-1] * n # BFS 트리에서의 parent 기록
dist = [0] * n # source 노드로부터의 최단 거리 기록
for all nodes s in G:
 if not visited[s]:
 Q.enqueue(s)
 visited[s] = True
 while Q is not empty:
 v = Q.dequeue()
 for each edge v → w:
 if not visited[w]:
 Q.enqueue(w)
 visited[w] = True
 parent[w] = v
 dist[w] = dist[v] + 1

```

- v. **수행시간**: DFS와 유사하게 각 에지  $(v, w)$ 에 대해,  $w$ 가 한 번씩 enqueue, dequeue되기에  $O(n + m)$  시간이면 충분하다

## 7. 최단 경로 문제(Shortest Path Problem)

- a. 일상생활에서 자주 경험하는 문제로, 문제 자체의 역사도 깊고 응용 가치도 커 많은 연구가 이루어진 그래프 분야의 고전 문제이다
- b. 에지에 가중치(비용)가 주어진 방향 그래프를 대상으로 한다
  - i. 단, 가중치는 모두 양수라고 가정한다. 가중치가 음수인 경우는 실제로 발생하는 경우가 적고, 음의 가중치를 갖는 에지가 있다면 이를 허용하는 최단 경로 알고리즘을 설계해야 한다
  - ii. 에지를  $u \rightarrow v$ 라고 하면, 가중치는  $\text{weight}(u \rightarrow v)$  또는  $\text{weight}(u, v)$ 로 표기한다
- c. 출발 노드  $s$ 가 입력으로 지정되고, 출발 노드  $s$ 에서 다른 모든 노드까지의 최단 경로를 찾는 문제를 다룬다 (이 문제를 **Single Source Shortest Path Problem**이라 부른다)
  - i. 출발 노드와 도착 노드를 지정해 하나의 최단 경로를 찾지 않는 이유는 출발 노드에서 다른 모든 노드로의 최단 경로를 찾는 복잡도와 다르지 않기 때문이다
- d. 최단 경로의 기본 성질:
  - i.  $u \rightarrow v$ : 노드  $u$ 에서 노드  $v$ 로의 하나의 에지로 구성된 경로
  - ii.  $s \rightarrow v$ : 노드  $s$ 에서  $v$ 로의 경로를 표시하고, 중간에 여러 노드가 존재 가능
  - iii.  $s \rightarrow u \rightarrow v$  인 경우, 이 경로에서  $u$ 는  $v$ 의 predecessor 또는 parent이다
  - iv. [중요한 사실] 만약  $s \rightarrow u \rightarrow v$  가  $s$ 에서  $v$ 로의 최단 경로 중 하나라면,  $s \rightarrow u$  역시  $s$ 에서  $u$ 까지의 최단 경로 중 하나이다
    - 1.  $s$ 에서  $v$ 까지의 여러 경로 중에서  $u$ 를 통해  $v$ 에 도달하는 경로 중에 최단 경로가 있다는 의미이고,  $s \rightarrow u$ 를 먼저 계산하여 알고 있다면,  $s \rightarrow u \rightarrow v$ 는 쉽게 계산할 수 있다는 뜻이다
    - 2.  $\text{dist}[v] = "s$ 에서  $v$ 까지의 최단 경로의 길이"라고 정의하면,
$$\text{dist}[v] = \min_{\text{for each } u \mid u \rightarrow v} \{ \text{dist}[u] + \text{weight}(u \rightarrow v) \}$$
  - 3. 즉,  $v$ 로 들어오는 각 에지  $(u, v)$ 에 대해,  $\text{dist}[u] + \text{weight}(u \rightarrow v)$ 를 계산하고 그 중에서 최소 값이  $\text{dist}[v]$ 가 된다는 의미이다
  - 4. 어라? 이건 DP 식인데… 맞다!
  - 5. 그러면,  $\text{dist}[v]$  계산 전에  $\text{dist}[u]$ 가 먼저 계산되면 된다. 즉,  $v$ 의 predecessor  $u$ 에 대한  $\text{dist}[u]$ 를 먼저 계산하면 된다.
  - 6. 이런 논리를 계속 적용해나가면, 노드  $s$ 까지 거슬러 올라가게 된다
  - 7. 당연히  $\text{dist}[s] = 0$ 이고,  $s$ 가 아닌 노드  $v$ 에 대해선,  $\text{dist}[v] = \infty$ 로 초기 값을 갖는다.

8. s에서 나가는 에지 중에서, 최소 가중치를 갖는 에지가  $s \rightarrow v$ 라면,  $\text{dist}[v] = \text{dist}[s] + \text{weight}(s \rightarrow v)$ 이 되는데, 여기서  $\text{dist}[s] = 0$ 이므로  $\text{dist}[v] = \text{weight}(s \rightarrow v)$ 가 된다

- 왜 그럴까?  $s \rightarrow u \rightarrow v$ 로 가는 경로가 v의 최단경로가 될 수도 있지 않을까? 만약 이게 사실이라고 하자.  $\text{weight}(s \rightarrow v)$ 가 s에서 나가는 에지 중에서 최소 가중치를 갖는다고 정의되었기에  $\text{weight}(s \rightarrow u) \geq \text{weight}(s \rightarrow v)$ 이다. 따라서  $s \rightarrow u \rightarrow v$ 의 경로는  $s \rightarrow v$ 보다 무조건 더 길어야 한다 (증명 끝)
- 따라서,  $\min_{s \rightarrow v} \text{weight}(s \rightarrow v)$ 인 에지를 따라가면 s에서 v까지의 최단 거리가 된다

9.  $\text{parent}[v] = (\text{s에서 } v\text{까지의 최단 경로에서 } v\text{ 바로 직전 노드를 저장})$

- 최단 경로를 재구성하기 위해  $\text{parent}$  값 필요 (DFS, BFS에서의  $\text{parent}$  개념과 같다)

v. 위의 DP식을 함수 **relax**란 이름의 함수로 정리하면:

```
def relax(u, v): # u를 통해 v로 오는 경로 길이를 dist[v]로 지정함
 dist[v] = dist[u] + weight(u → v)
 parent[v] = u
```

e. 다음을 실행하면 어떻게 될까?

```
for v in V: # 모든 노드 v에 대해, dist[v]를 매우 큰 값으로 초기화
 dist[v] = ∞

for i in range(1, n): # (n-1)번 반복
 for each edge u → v in G: # u를 통해 v로 오는 경로가 더 짧다면
 if dist[v] > dist[u] + weight(u → v):
 relax(u, v) # dist[v]를 update
```

i. 이 코드가 s에서 다른 모든 노드로의 최단 경로의 길이를 올바르게 계산하나?

1. 우선, 모든 에지는 정확히  $(n-1)$ 번 if-relax 검사를 받는다. 이 반복을 round라고 하면, round 1, round 2, ..., round  $(n-1)$ 이 실행된다
2. round 1을 마치고 나면, 어떤 노드의 최단 경로가 계산된다. 어떤 노드인가?
  - 출발 노드 s에서 나가는 에지 중에 weight가 가장 작은 에지의 다른 노드를 u라고 하면,  $\text{dist}[u]$ 는  $\text{weight}(s \rightarrow u)$  값으로 relax되고, 이 값이 s에서 u까지의 최단 경로 길이이다
  - 이 노드 u는 출발 노드로부터의 최단 경로가 예지 하나로 구성된 경우이다. u 이외에도 예지 하나로 구성된 최단 경로는

모두 round 1에서 최단 경로 길이가  $dist[]$ 에 제대로 update되고, 이후의 round에서는 이 값이 변하지 않는다

3. 이 논리를 round k로 확대 적용하면,  $(k-1)$ 개의 에지로 구성된 최단 경로의 길이는 round  $(k-1)$ 에서 제대로 update된다. round k에서는 여기에 한 에지를 더 연결해, k개의 에지로 구성된 최단 경로의 길이가 제대로 계산된다. 이유는 이 최단 경로가  $(k-1)$ 개의 에지와 마지막 한 개의 에지로 구성되는데, 이를  $s \rightsquigarrow u \rightarrow v$ 라고 하면,  $s \rightsquigarrow u$ 는  $(k-1)$ 개의 에지로 구성된 u까지의 최단 경로이고 round  $(k-1)$ 에서 제대로 계산된다. 따라서 round k에서  $u \rightarrow v$ 에 해당하는 에지가 relax되면서 s에서 u까지의 최단 경로가 계산되는 것이다
4. 경로에 포함된 노드는 모두 달라야 하기에, 아무리 긴 최단 경로도 에지가  $n-1$ 개 보다 더 많이 포함할 수는 없다. 따라서  $(n-1)$ 번만 반복하면 s에서 다른 모든 노드로 가는 최단 경로 길이가 올바르게 계산된다

- ii. 위의 알고리즘이 바로 유명한 [Bellman-Ford 알고리즘](#)이다. 자세한 내용은 오른쪽 유튜브 설명을 참고하기 바란다
- iii. 수행시간은  $relax$  함수가 상수 시간이므로 이중 반복문 회수에 비례하게 된다. 따라서  $O(nm)$ 이면 충분하다



- f. 다음을 실행하면 어떤 걸 알 수 있을까?

```
for each edge $s \rightarrow u$:
 if $dist[s] > dist[s] + weight(s \rightarrow u)$:
 relax(s, u)
```

- i.  $s$ 에 인접한 노드  $u$ 에 대해,  $dist[u]$  값이 update된다. 그럼  $dist[u]$  값이 가장 작은  $u$ 는 최단 경로의 길이가 된다 (앞에서 이미 설명함)
- ii.  $u$ 에 인접한 노드들에 대해서 다시  $relax$ 를 반복하면,  $s \rightarrow u \rightarrow v$ 로 연결되는 최소 경로의 길이가  $dist[v]$ 에 저장된다
- iii. 이제는  $s$ 와  $u$ 에 인접한 에지들이 한 번 이상씩  $relax$ 가 된 상태다.  $s$ 와  $u$ 에 인접한 노드에 대해,  $dist[v]$ 가 가장 작은 값을 갖는 노드  $v$ 를 고려해보자.  $dist[v]$  값은  $s$ 에서  $v$ 로 오는 경로의 길이 중 현재까지 가장 작은 값이다. 그러면  $dist[v]$ 는  $s$ 에서  $v$ 까지의 최단 경로 길이임을 알 수 있다 (왜? 이보다 더 짧은 경로가 있었다면 그 전에 이미 고려되었어야 한다)
- iv. 결국, 현재의  $dist[u]$  값이 최소가 되는  $u$ 를 선택하면,  $dist[u]$ 는  $s$ 에서  $u$ 까지의 최단 경로의 길이가 되고 이 후에도  $relax$ 에 의해 변경되지 않는다.  $u$ 가 선택되면,  $u$ 의 인접한 노드  $v$ 에 대해 if-relax 문을 통해  $dist[v]$ 의 값을 (필요하면) update한다. 이 과정을 정리하면 다음 pseudo 코드와 같다

```

Dijkstra_SP_algorithm(G)
 s = 0 # 0 node is source node
 dist = [0, ∞, ..., ∞], parent = [None, ..., None]

 # 노드 v의 dist[v] 값을 key로 하는 힙 H
 # 여기서 힙은 당연히 min 힙이어야 한다!
 H ← all nodes v with key dist[v]

 while H is not empty:

 u = H.delete_min() # dist 값이 가장 작은 노드 선택
 # GREEDY choice!

 for each edge u → v:
 if dist[v] > dist[u] + weight(u → v):
 relax(u, v) # DP update!
 H.decrease_key(v, dist[v]) # decrease_key 연산

 # 주의점: 노드 v가 힙 H에 저장된 index를 알아야 한다! (왜?)
 # 즉, 노드 v는 자신의 key 값과 H에서의 index 정보 쌍을 알아야 한다
 # 그래서 적응형 힙(Adapted Heap)을 쓰면 편리 (힙 자료구조 참조)

 return dist, parent

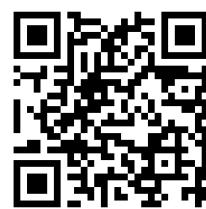
```

g. 이 유명한 알고리즘을 Dijkstra 최단 경로 알고리즘이라 부른다

- i. Dijkstra는 다익스트라로 발음한다
- ii. Dijkstra 알고리즘 = Greedy 선택 + DP 전략을 혼용한 알고리즘!!

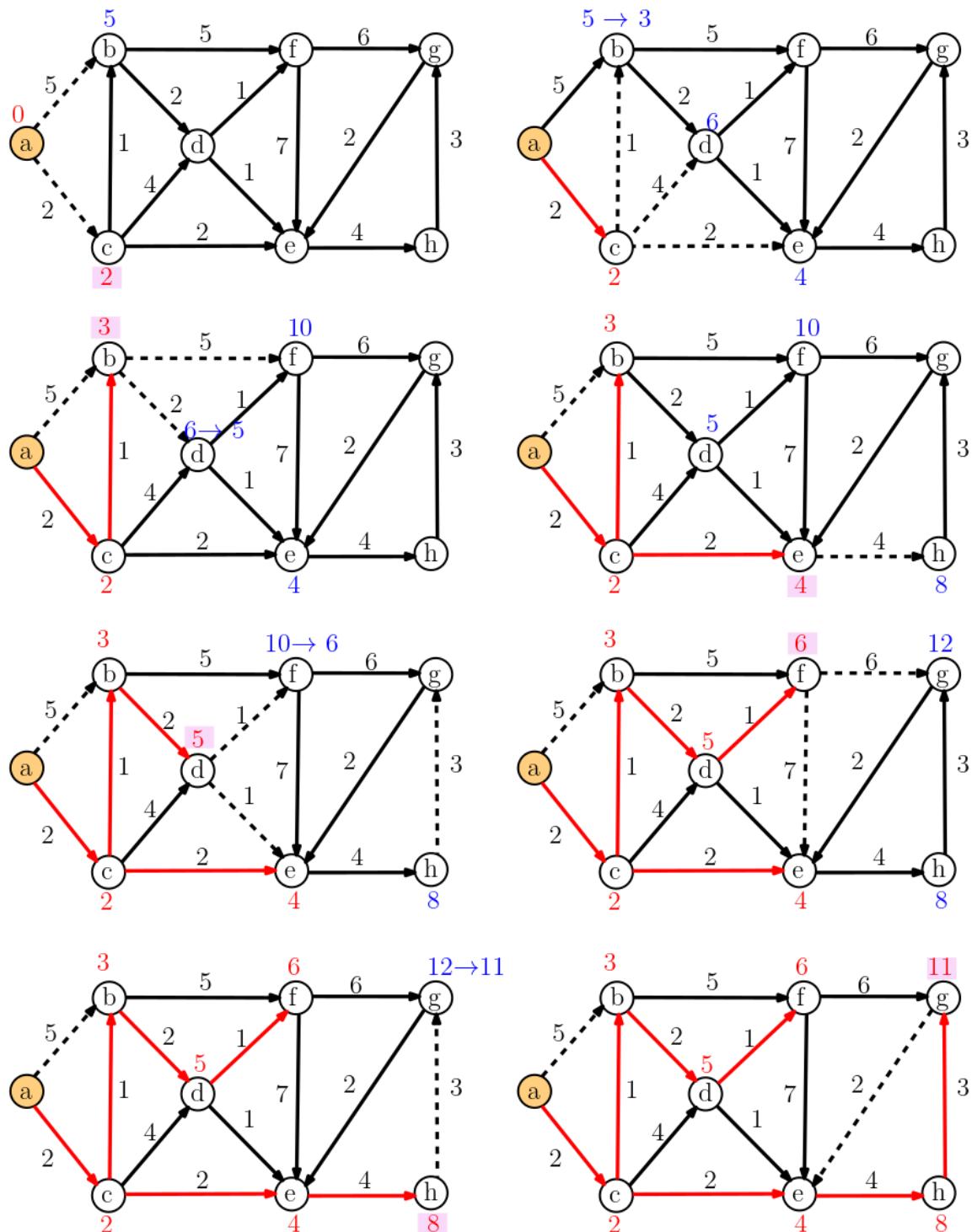


(알고리즘 설명)



(예제 설명)

아래 그래프에 Dijkstra 알고리즘을 적용해보자. 리스트 dist의 각 노드의 값은 출발 노드는 0, 다른 노드는 매우 큰 값(무한대)으로 초기화한다. 그림에서 노드 위에 표시된 값이 해당 노드의 dist 값이다. 무한대 값은 표시하지 않았다. 노드 a가 출발 노드이다. a의 인접한 노드들에 대해 relax를 하는 데, 해당 에지를 점선으로 표시했다. 현재 dist 값 중에 제일 작은 값을 heap에서 꺼내 (delete\_min) 해당 노드로의 에지를 최단 경로의 에지로 선택한다. 이 에지를 빨간색으로 표시했다. 이 과정을 모든 노드가 힙에서 delete\_min 될 때까지 반복한다



### h. 최단 경로 트리

- i. 최단 경로를 구성하는 빨간색 에지들을 모아보면 트리가 된다. 이 트리는 리스트 `parent`에 저장되어 있다. 이 트리를 최단 경로 트리라고 부른다
  1. 그런데 왜 트리 형태가 될까? 최단 경로의 부분 경로도 최단 경로라는 사실 때문이다

2. 물론 최단 경로 트리가 유일한 것은 아니다. 어떤 노드에 도달하는 최단 경로가 하나 이상일 수도 있기에, 그 중 하나의 경로만 최단 경로 트리에 등장한다
- ii. 최단 경로 트리 (즉, parent 정보)만 있으면 출발 노드에서 도착 노드까지의 최단 경로를 쉽게 재구성할 수 있다

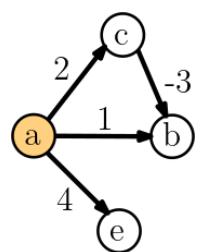
i. 구현 이슈:

- i. 개별 노드의 현재  $dist$  값을 힙에 넣어 관리해야 한다. 힙에는 ( $dist[v]$ ,  $v$ )의 정보가 저장된다. key 값은  $dist[v]$ 이고, 이 값이 노드  $s$ 에서  $v$ 까지의 거리임을 나타낸다. 이 힙에서는 두 가지 연산이 필수적인데,
  1. `delete_min` 연산: 최소  $dist$  값을 갖는 노드를 알아야 한다
  2. `decrease_key` 연산: 에지  $u \rightarrow v$ 에 대해 `relax` 연산이 수행되어  $dist[v]$  값이 감소하면 힙에서의 위치가 조정되어야 한다
- ii. 힙마다 두 연산의 수행시간이 다를 수 있다
  1. binary heap: 자료구조 시간에 배우는 일반적인 배열에 저장되는 힙
    - o `insert`, `delete_min`, `decrease_key` 모두  $O(\log n)$  시간 필요
  2. Fibonacci heap: binary heap보다 복잡한 형태의 힙
    - o `insert`, `decrease_key`는  **$O(1)$**  시간 필요 ( $\leftarrow$  차이점!)
    - o `delete_min`은  $O(\log n)$  시간 필요
    - o wikipedia: [https://en.wikipedia.org/wiki/Fibonacci\\_heap](https://en.wikipedia.org/wiki/Fibonacci_heap)
- iii. 수행시간: 사용하는 힙의 종류에 따라 다르다!
  1. 힙이  $dist$ 의 초기값을 가지고 구성된다 (`make_heap` 연산) -  $O(n \log n)$
  2. 노드는 한 번씩 힙에서 `delete_min`이 수행된다 -  $O(n \log n)$
  3. 에지마다 최대 한 번씩 `decrease_key`가 수행된다 -  $O(m \times d)$ 
    - o  $d$ 는 `decrease_key`의 수행시간
  4. 총 시간:  $O(n \log n + m d)$ 
    - o binary heap의 경우:  $d = O(\log n) \rightarrow O((n+m)\log n)$
    - o Fibonacci heap의 경우:  $d = O(1) \rightarrow O(n \log n + m)$

j. [생각해 볼 이슈] 가중치가 음수인 에지가 있다면?

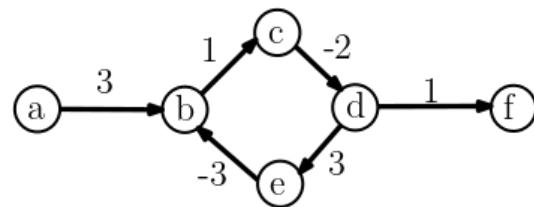
- i. 두 가지 경우로 나눠 생각해 볼 수 있다. 음수 가중치를 갖는 에지가 존재하지만, 사이클의 가중치 합 (사이클의 에지의 가중치 합)이 음수가 되는 경우와 그렇지 않은 일반적인 경우

1. 가중치가 음수인 에지가 있지만 사이클의 가중치 합은 모두 음수가 아니라면, Bellman-Ford 알고리즘은 문제없이 동작한다. 그러나 Dijkstra 알고리즘은 올바르게 동작하지 않을 수도 있다. 어떤 그래프일 때 제대로 동작하지 않을까? 오른쪽 그래프를 보면, 출발 노드  $a$ 의 세 에지 중에서 최소 가중치 값을 갖는 에지 ( $a, b$ )가 선택되어  $dist[b] = 1$ 로



고정된다. 즉, b까지의 최단 경로 길이는 1로 결정된다. 그러나 c를 거쳐 b로 가는 경로의 길이가 -1이 되어 더 짧은데 발견하지 못한다. 그러나 Bellman-Ford 알고리즘은 제대로 동작한다. 왜 그럴까? 위의 그래프를 보면, round 1에서는  $\text{dist}[b] = 1$ ,  $\text{dist}[c] = 2$ ,  $\text{dist}[e] = 4$ 로 relax되고, round 2에서는  $\text{dist}[b] = \text{dist}[c] + -3 = -1$ ,  $\text{dist}[c] = 2$ ,  $\text{dist}[e] = 4$ 이고, round 3에서는 변화가 없다.  $(n-1)$ 번 모두 에지에 대해 relax 연산을 하기에 음의 사이클이 없는 한 제대로 계산하게 된다

2. 사이클 가중치 합이 음수가 아래 그림과 같은 그래프처럼, 사이클 가중치 합이  $1+(-2)+3+(-3) = -1$ 이 되어 음수된다면, 사이클을 계속 반복할 수록 가중치 값이 감소하게 된다. 그러면 노드 a에서 노드 f로 가는 최단 경로는 정의할 수 없다



3. 이 상황에서는 최단 경로가 정의가 안된다는 걸 인식할 수 있어야 한다. Dijkstra 알고리즘은 음의 에지가 존재하는 경우에도 제대로 동작하지 않기 때문에, 이 상황을 제대로 인식하지 못한다. Bellman-Ford 알고리즘에서는 이 상황을 인식할 수 있을까?  $(n-1)$ 번의 round를 거친 이후에는 dist 값이 각 노드까지의 최단 경로 길이로 결정되어야 한다. 만약 한 번 더 반복을 해서, n번째 round를 추가로 실행해서 dist 값 중 일부가 더 작은 값으로 업데이트 된다면 음의 사이클이 존재한다는 의미가 된다. 따라서 Bellman-Ford 알고리즘은 추가로 한 번 round를 수행해 음의 사이클의 존재여부를 검사할 수 있다

8. [🎤 인터뷰 문제] 최단경로문제에서 한 걸음 더

- a. Dijkstra 알고리즘은 출발 노드에서 다른 모든 노드까지의 최단경로를 모두 계산해주는 One-to-All 알고리즘이다. 만약, 각 노드에서 목적 노드까지의 최단경로를 알고 싶다면 어떻게 해야 할까? 즉, All-to-One 문제는 어떻게 풀어야 하나?
  - i. **무방향 그래프**인 경우: 무방향 에지이기 때문에, 노드 a에서 노드 b로의 최단 경로가 b에서 a로의 최단 경로다. 따라서 One-to-All 경로를 거꾸로 출력하면 All-to-One 경로가 된다
  - ii. **방향 그래프**인 경우: 에지에 방향이 있기 때문에, 에지의 방향을 반대로 바꾼 그래프를 새로 정의해서 새 그래프에서 One-to-All 알고리즘을 수행한다. 그 결과가 All-to-One 경로가 된다
- b. [난이도 높음] Second Best Shortest Path 문제: 두 노드 s와 t 사이의 최단 경로가 유일할 수도 있지만, 두 개 이상 존재할 수도 있다. 만약, 두 번째로 빠른 경로를 알고 싶다면 어떻게 계산해야 할까? 최단 경로가 2개 이상이면 두 번째로 빠른 경로 역시 최단 경로가 되고, 최단 경로가 하나 뿐이라면, 두 번째로 빠른 경로는 최단 경로보다 더 길다. 두 번째로 빠른 경로의 길이를 계산하는 알고리즘을 생각해보자.
  - i. 네비 길찾기에서 가장 빠른 경로와 두 번째로 빠른 경로를 동시에 보여주고 사용자에게 선택하도록 하는 응용에 사용 가능하다
  - ii. 최단 경로와 두 번째로 빠른 경로의 차이를 분석하는 게 우선 할 일!
  - iii. 두 번째로 빠른 경로는 최단 경로와 같은 에지를 따라 가지 않고 다른 에지 하나는 사용할 것이다. 예를 들어, s에서 최단 경로의 에지를 차례대로 따라 가다가 어떤 노드 u에서 다음 노드 v를 따라가지 않고 u에서 w로 다른 에지를 따라 가게 된다. w에선 목적 노드 t까지 w-t 최단 경로를 따라 가야 한다. (왜?) 여기서 u = s일 수도 있음에 유의하자.
  - iv. 즉, 두 번째로 빠른 경로는 s에서 u까지 최단경로는 그대로 따라가고, u에서 v가 아닌 u에서 w로 가는 에지를 선택하는 순간이 반드시 존재해야 한다. (아니라면 s-t 최단 경로를 그대로 따라가게 되므로)
  - v. ( $u, v$ ) 에지가 아닌 ( $u, w$ ) 에지를 선택했기 때문에, 이 에지를 통해 t로 가는 어떤 경로도 최단 경로의 길이와 같거나 더 커야 한다. 우리는 두 번째로 빠른 경로를 구하는 것이기에 w에서 t까지는 최단 경로로 가야 한다. 결국, 두 번째로 빠른 경로는

$(s\text{에서 } u\text{까지의 최단 경로}) + (u \rightarrow w) + (w\text{에서 } t\text{까지의 최단 경로})$

의 모양이 될 것이다. 결국, s에서 t까지의 최단 경로 상에 있는 모든 u에 대해, 최단 경로 상의 에지가 아닌 다른 에지를 이용해서 t에 도착할 수 있는 길을 조사해서 그 중 길이가 제일 짧은 경로가 두 번째로 빠른 경로가 된다.

Dijkstra 알고리즘을 이용해서 구하고 싶다. 구체적으로 어떻게 해야 할까?

## 7. Other Data Structures [Advanced Topics]

1. Union-Find 자료구조
2. van Emde Boas 자료구조
3. Suffix array 자료구조
4. Range tree 자료구조

### 1. Union-Find 자료구조

- a. 여러 개의 집합을 관리하는 자료구조로 파이썬의 set 자료구조와 유사
- b. 지원 연산
  - i. make\_set(x): x를 원소로 하는 새로운 집합을 생성
  - ii. union(x, y): x가 속한 집합과 y가 속한 집합이 다르면 union (합집합)
  - iii. find(x): x를 포함한 집합의 대표 원소를 리턴
- c. 가장 단순한 방법은 하나의 집합을 이중연결리스트로 관리하는 방법
  - i. make\_set(x): 노드 x를 포함한 리스트 만들어 리턴
  - ii. find(x): 노드 x가 속한 리스트의 head 노드를 찾아 리턴 (head 노드가 집합을 대표)
  - iii. union(x, y): x, y에 대해 find 함수를 불러 head 노드를 찾은 후, 두 노드가 서로 다르다면 (다른 집합이라면) 두 리스트를 연결하여 하나의 리스트로 만든 후, 전체의 head 노드를 리턴
  - iv. 연산시간: make\_set이 n번 호출되었다고 하면, 총 n개의 원소가 존재
    - 1. find 연산은 최악의 경우에  $O(n)$  시간 필요
    - 2. union 연산도 find 연산을 두 번 호출하므로  $O(n)$  시간 필요
  - v. 더 빠른 방법은 없을까?
- d. 연결리스트가 아닌 트리 형태로 관리하는 방법
  - i. 트리의 노드는 key, parent와 rank 세 멤버 값을 갖는다 (자식 정보는 없음!)
  - ii. 트리의 루트 노드가 집합을 대표함
  - iii. rank는 union할 때, 두 집합에 대한 부모, 자식 관계를 결정할 때 사용함

```
class Node:
 def __init__(self, key):
 self.key = key
 self.parent = self # 자기 자신을 부모로 초기화
 self.rank = 0 # rank = 0 초기화
```

- iv. make\_set(key): return Node(key) # 노드를 만들어 리턴함

- v. `find(x)`: 노드  $x$ 의 부모 링크를 루트 노드까지 따라 올라가서 루트 노드 반환

```
def find(x):
 while x.parent != x: # root 노드에 도달할 때까지
 x = x.parent
 return x
```

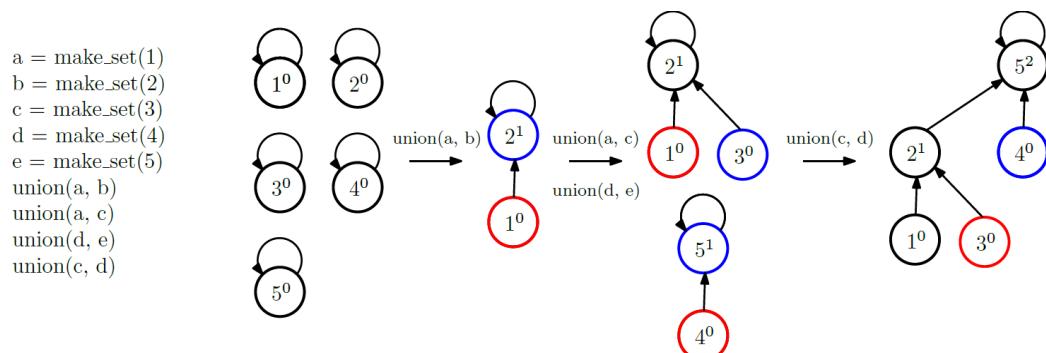
- vi. `union(x, y)`:

1. 대표 노드를 각각 찾음:  $v = \text{find}(x)$ ,  $w = \text{find}(y)$
2. rank가 작은 집합이 rank가 큰 집합을 부모로 연결해 결합
  - a. 예를 들어,  $v.\text{rank} < w.\text{rank}$ 라면  $w$ 가  $v$ 의 부모 노드가 됨
  - b. rank가 같다면  $w$ 의 rank를 1 증가 (union 후의 트리의 rank는 원래 두 집합의 rank가 같은 경우에만 1 증가하고, 그렇지 않은 경우엔 변하지 않음)
3. 이렇게 rank 정보를 이용해 union하는 방법을 "union by rank" 방법이라 부른다
4. "union by size" 방법도 있는데, 노드 수가 작은 트리가 노드 수가 더 큰 트리의 자식이 되는 결합 방식으로 union by rank와 본질적으로 동일하다

```
def union(x, y):
 v, w = find(x), find(y)
 if v.rank > w.rank:
 v, w = w, v # swap → 항상 w.rank >= v.rank

 v.parent = w # v → w
 if v.rank == w.rank:
 w.rank += 1
```

- vii. 예: (그림 출처: CLRS Introduction to algorithms에서)



- viii. 연산시간

1. `make_set`은 상수시간

2. `find(x)`는  $x$ 가 속한 집합의 대표인 루트노드로 따라 올라가야 하기 때문에 최악의 경우에 트리의 높이  $h$ 만큼의 시간이 필요함. 따라서  $O(h)$  시간 필요
3. `union(x, y)` 역시 두 번의 `find` 함수를 호출함으로  $O(h)$  시간 필요
4. [?] 트리의 높이는 최악의 경우에 어느 정도까지 커질까?
  - a. 높이  $h$ 와 `rank`의 관계는? **높이와 랭크는 결국 같은 값이다**
  - b. `rank` 값에 따라 부모-자식 관계를 결정하는 이유가 있을 듯!
5. [?]  $\text{rank} = k$ 라면 그 집합에는 최소 몇 개의 노드가 존재할까?
  - a.  $a(k) = \text{rank}$ 가  $h$ 일 때 최소 노드 수로 정의
  - b.  $a(0) = 1, a(1) = 2, a(2) = 4$
  - c.  $\text{rank} = h-1$ 에서  $h$ 가 될 때의 상황으로  $a(h)$  점화식을 유도해보자

[힌트] 두 `rank`가 같은 경우만 1씩 증가한다 → 같은 `rank`의 두 트리가 `union` 되므로 결합된 트리는 두 트리 중 작은 트리의 2배 이상이 된다

# 높이가  $h-1$ 로 같은 두 트리를 연결하기에 다음 점화식 성립

$$a(h) = 2a(h-1), a(0) = 1$$

- d. 이 점화식을 풀면,  $a(h) = 2^h a(0) = 2^h$  이 된다
- e. 노드의 개수를  $n$ 이라 하면  $a(h) = 2^h \leq n$ 이 되어,  **$h \leq \log n$** 임을 알 수 있다

6. 결국, `find`와 `union` 연산은  $O(h) = O(\log n)$  시간에 동작한다

- ix. `find` 함수를 **path compression**이라는 방법에 따라 수행하면 더 빠른 연산시간을 보장할 수 있다

1. 수행시간 분석이 조금 복잡해서, 구체적인 증명은 생략한다. 아래 위키 자료를 참고하기 바란다  
[\[https://en.wikipedia.org/wiki/Disjoint-set\\_data\\_structure\]](https://en.wikipedia.org/wiki/Disjoint-set_data_structure)
2. `find(x)`에서  $x$  노드부터 루트 노드로 이동하는 데, 이동 중에 방문하는 노드의 부모 노드를 모두 루트 노드로 변경하는 방법이 **path compression**이다
3.  $x$  노드부터 루트 노드까지의 경로가 해체되어 경로 위의 모든 노드가 바로 루트 노드를 가리키게 되어 다음 번 `find` 연산의 시간이 대폭 줄어드는 효과가 있다!

```
def find_pc(x): # non-recursive version
 root = x
 while root.parent != root:
```

```

 root = root.parent
 # root is found
 while x.parent != root:
 p = x.parent
 x.parent = root
 x = p
 return root

def find_pc(x): # recursive version
 if x.parent != x:
 x.parent = find(x.parent)
 return x

```

- x. **수행시간**: find by path compression + union by rank
1. find와 union 모두 (amortized)  $O(\alpha(n))$  시간이면 충분
  2.  $\alpha(n)$  함수는 매우 매우 천천히 증가하는 함수로  $O(\log n)$ 보다 훨씬 느리게 증가하는 함수임. 상수는 아니지만 상수 시간으로 간주해도 됨.  
**Inverse Ackermann** 함수라는 이름을 가짐 [참고: [Wikipedia](#)]

### xi. 구현 이슈

1. 굳이 노드 클래스를 선언해 구현할 필요 없이, 두 리스트 parent와 rank를 정의해 사용해도 된다. 값  $u$  ( $1, \dots, n$  중 하나)에 대해,  $parent[u]$ 는  $u$ 의 parent 값을 나타내고,  $rank[u]$ 는  $u$ 의 현재 rank 값을 나타낸다
2.  $parent[u] = u$ 로 초기화하고  $rank[u] = 0$ 으로 초기화하면 된다

### e. 활용

- i. 무방향 그래프에 노드 또는 에지가 새로 삽입(insert)되는 경우에 **연결 성분** (connected component)을 유지하는 문제에 활용 가능하다. 하나의 연결 성분을 union-find의 하나의 트리로 표현하면, 노드와 에지의 삽입으로 두 연결 성분이 하나의 연결 성분으로 합병되는 것을 find와 union 연산으로 처리할 수 있다
- ii. 그래프의 **최소 신장 트리** (MST: Minimum Spanning Tree)를 구하는 알고리즘 중에서 Kruskal 알고리즘에 사용: Kruskal 알고리즘은 신장 트리를 점진적으로 구성하는데, 작은 부분 트리들을 서로 합병하면서 신장트리를 만든다. 부분 트리를 union-find 자료구조로 표현하면, 두 부분 트리를 찾고 합병하는 연산은 각각 find와 union 연산으로 처리할 수 있다
- f. Python의 set과 union-find 자료구조와의 차이점은?

- i. `set`은 union-find 자료구조가 아닌 파이썬의 `dict`처럼 해시 테이블로 구현되어 있다. 따라서 `find` 함수는 평균적으로  $O(1)$  시간에 수행 가능하지만, 최악의 경우에는  $O(n)$  시간이 필요할 수도 있다. 반면에 union-find 자료구조에서는 `find` 연산이 최악의 경우에도  $O(\log n)$  시간임

## 2. [상당히 복잡] van Emde Boas 트리 - 정수 연산 (integer arithmetic) 자료구조

- a. 우선순위 큐(priority queue)에서 제공되어야 하는 기본 연산은 `findMin`, `findMax`, `insert`, `deleteMax`, `deleteMin`, `decreaseKey` 등이다

- b. 이진 힙(binary heap)에서는  $n$ 개의 key가 힙에 저장되어 있다면, `findMin/findMax`는  $O(1)$  시간에, 나머지 3개의 연산은 모두  $O(\log n)$  시간에 수행된다

피보나치 힙(Fibonacci heap)은 더 복잡한 구조로 `deleteMax/deleteMin`만  $O(\log n)$ 이고 나머지 연산은 모두  $O(1)$  시간에 수행 가능하다

균형이진트리도 우선순위 큐에서 제공하는 모든 연산을  $O(\log n)$  시간에 제공한다

- i. `deleteMax/deleteMin`, `decreaseKey`는 어떻게 처리할까?

- c. 이진 힙과 피보나치 힙에 저장되는 key 값으로는 실수 또는 문자열과 같이 대소를 비교할 수 있는 어떤 값도 가능하다

- d. 만약 key 값으로 정수 값만을 허용한다면 이 제한 조건을 이용해서 연산을 더 빠르게 할 수 있지 않을까?

**예:** 임의의 값을 비교를 통해 정렬하는 데 최소  $O(n \log n)$  시간이 필요하지만, 최대  $d$ 자리의 수를 radix 정렬할 때에는  $O(dn)$  시간이면 충분한 것과 같은 이유

- e. key 값이  $m$ -bit 정수 값인 경우에 van Emde Boas (vEB) 트리는  $M = 2^m$ 의 메모리를 사용하여 ( $m$ -bit 정수 값이므로, key 값은 **0부터  $M-1$  사이의 정수 값**으로 가정)

- i. `search(lookUp)`, `insert`, `delete`, `findMin`, `findMax`, `successor`, `predecessor` 연산을 제공하며, 모두  **$O(\log \log M) = O(\log m)$**  시간에 수행 가능하다

- ii. 보통 다루는 key 값이 32 비트 정수라면,  $[0, \dots, 2^{32}]$  범위에 해당한다. 이 경우에 세 연산은 모두  $O(\log \log 2^{32}) = O(\log 32) = O(5) = O(1)$ 가 되어 매우 빠르다

- iii. 1977년, 네덜란드의 Peter van Emde Boas에 의해 제시된 자료구조!

- f. [참고 영상] <https://www.youtube.com/watch?v=hmReJCupbNU>

- g. 그럼  $O(\log \log M)$ 은 어떻게 얻어졌을까?

- i.  $T(k) = T(k/2) + O(1)$ 인 경우  $T(k) = O(\log k)$

- ii.  $T(\log M) = T((\log M)/2) + O(1)$ 이면  $T(\log M) = O(\log \log M)$

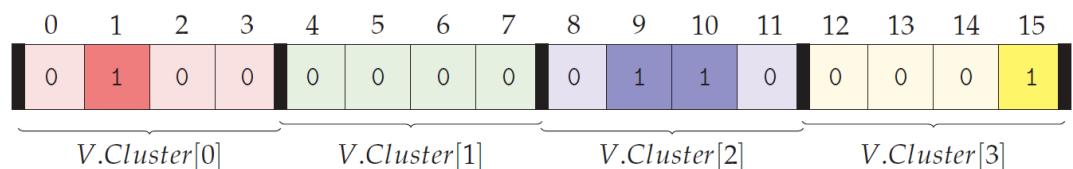
- iii.  $T(M) = T(\quad) + O(1)$  점화식이  $T(M) = O(\log \log M)$ 로 정리된다고 할 때, 빈 칸에 와야 할 내용은 무엇인가?  **$\sqrt{M}$**

- h. 알고리즘 1: Bit Vector 이용 방법

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1  | 0  | 0  | 0  | 0  | 1  |

- i. 배열  $V[0..M-1]$ 을 이용해, key  $x$ 가 insert되면  $V[x] = 1$ 을 set하고, 만약 delete 된다면  $V[x] = 0$ 으로 reset하는 직관적인 방법 → insert/delete 모두  $O(1)$  시간이면 충분하다
- ii. successor( $V, x$ ):  $V[x]$ 의 다음 entry를 차례대로 검사해 처음으로  $V[y] == 1$ 이 되는  $y$ 를 찾아 리턴함 → 최악의 경우엔  $O(|V|) = O(M)$  시간이 필요. predecessor( $V, x$ ) 역시  $O(M)$  시간이 필요하다
- iii. insert/delete:  $O(1)$  시간
- iv. successor, predecessor:  **$O(M)$**  시간
- v. Bit vector + 완전이진트리를 이용하는 경우: 모든 연산이  $O(\log M)$  시간 필요
- vi. 균형이진탐색트리를 이용하는 경우: 모든 연산이  $O(\log n)$  시간 필요

i. 알고리즘 2:  $V$ 를 cluster로 일률적으로 나누는 방법



- i. 그림처럼  $M = 16$ 이라면  $\sqrt{M} = 4$ 개의 그룹으로 나누고 각 그룹을 **cluster**라 부른다
- ii. 현재는  $x = 1, 9, 10, 15$  네 개의 key 값이 삽입되어 있는 상태이다
- iii. 만약  $x = 9$ 라면,  $x$ 가 저장된 cluster 번호와 해당 cluster에서의 위치(offset 번호)는 무엇인가?  
 $cluster$  번호 =  $x//4$ ,  $cluster$  안에서의 offset 번호 =  $x \% 4$
- iv.  $V$ 는 2차원 배열처럼 생각할 수 있기에,  $x = 9$ 는  $V.cluster[x//4][x \% 4]$ 에 저장됨!
- v. 추가로  $\sqrt{M}$ 개의 원소로 구성된  $V.summary$  배열을 준비해서,  $V.summary[i] == 1$  이면 cluster  $i$ 가 non-empty임을 표시함 (즉, 최소 하나 이상의 key 값이 cluster  $i$ 에 삽입되었음을 의미)
- vi.  $x$ 의 위치를 일차원 배열의 인덱스  $index(i, j)$ 로 가정하면,  $x = i * \sqrt{M} + j$ 로 표현할 수 있음.  $i$ 는 cluster 번호이고,  $j$ 는 cluster 내에서의 위치 (offset 번호)이다
  1.  $high(x)$ 는  $x$ 의 상위  $m/2$  비트 값,  $low(x)$ 는 하위  $m/2$  비트 값으로 정의하고 다음과 같다
    - a.  $i = x // \sqrt{M} = high(x)$
    - b.  $j = x \% \sqrt{M} = low(x)$

vii. 즉,  $\text{index}(i, j) = i * \sqrt{M} + j$  로 정의된다. (만약,  $\text{index}(i, j)$ 가  $[0, M-1]$  범위가 아니라면 `None`으로 정의)

viii. **insert(V, x):**

1.  $V.\text{cluster}[\text{high}(x)][\text{low}(x)] = 1$
2.  $V.\text{summary}[\text{high}(x)] = 1$

ix. **successor(V, x):**

1. **check** first whether successor  $y$  is within  $V.\text{cluster}[\text{high}(x)]$  by examining from  $V.\text{cluster}[\text{high}(x)][\text{low}(x)]$  to the cluster's end  
  - if** it is found at  $j$ :
  - a.  $y = \text{index}(\text{high}(x), j)$
2. **else:** # not found at the cluster
  - a. find the next non-empty cluster  $i$  in  $V.\text{summary}$
  - b. find the first non-empty entry  $j$  in  $V.\text{cluster}[i]$
  - c.  $y = \text{index}(i, j)$
3. **return**  $y$

예: `successor(V, 9)`, `successor(V, 1)`

x. `insert`는  $O(1)$  시간이면 충분

xi. `successor`는  $O(\sqrt{M})$  시간에 가능

1. 1번 단계에서  $y$ 가  $V.\text{cluster}[\text{high}(x)]$ 에 있는지 cluster의 값들을 오른쪽으로 가면서 하나씩 차례대로 검사  $\rightarrow O(\sqrt{M})$
2. 만약 없다면,  $V.\text{summary}[\text{high}(x)]$ 부터 오른쪽으로 가면서 처음으로 등장하는 non-empty cluster를 찾는다: 2.a 단계  $V.\text{summary}$ 의 entry를  $\text{high}(x)$  다음부터 하나씩 검사해서 찾음  $\rightarrow$  최악의 경우 cluster 개수  $\sqrt{M}$ 에 비례  $\rightarrow O(\sqrt{M})$
3. 2.b 단계에서 가장 처음으로 나오는 1을 찾아야 하는데, cluster  $i$ 의 가장 왼쪽 entry부터 하나씩 검사함  $\rightarrow O(\sqrt{M})$  ( $\rightarrow O(1)$  시간으로 줄일 수 있는 방법이 존재한다. **어떻게?**)
4. 결국, 모두 더하면  $O(\sqrt{M})$  시간이 되지만, 아직도 느림!

xii. `delete`는 여기선 고려하지 않음

### j. 알고리즘 3: Recursion (ver. 1)

i. 알고리즘 2에서는 자료구조가 한 레벨만으로 구성된 자료구조임. (즉, 하나의 `V.cluster`와 하나의 `V.summary`만 존재)

ii. 이 자료구조 `v`를 재귀적으로 여러 레벨에 걸쳐 구성함 [중요]

1. 설명의 편의를 위해,  $M = 2^{\{2^k\}}$ 의 형태라고 가정한다. 즉, 매 레벨 재귀적으로 내려갈 때마다  $2^{\{2^k\}} \rightarrow 2^{\{2^{\{k-1\}}\}} \rightarrow \dots \rightarrow 2^2 \rightarrow 2$ 로 정수 크기가 되도록 한다
2. `V.cluster[i]` = 크기가  $M^{1/2}$ 인 van Emde Boas 트리를 재귀적으로 구성한다. `V.cluster[i]`은 다시  $M^{1/4}$ 개의 cluster로 나뉘고, 이 cluster는  $M^{1/4}$  크기의 van Emde Boas 트리로 구성되는 식이다.
3. `V.summary` = 유사한 방식으로 크기  $M^{1/2}$ 인 van Emde Boas 트리를 재귀적으로 구성한다
4. `V.size` = 현재 vEB 구조의 크기 (현재 구조의  $M$  값)
5. 바닥 레벨 (`V.size = 2`인 경우) 크기가 2인 배열 `A`로 구성되고 `A`에 삽입된 key 값이 저장된다

#### iii. `insert(V, x):`

```
if V.size == 2: # 마지막 레벨
 V.A[x] = 1
else:
 insert(V.cluster[high(x)], low(x)) # 재귀삽입
 insert(V.summary, high(x)) # 재귀삽입
```

- 재귀 구조의 가장 윗 레벨부터 재귀적으로 찾아 내려가면서 가장 마지막 재귀 레벨 (`V.size == 2`)에 실제 삽입이 이루어진다
- `V.cluster`와 `V.summary`는 같은 크기의 vEB 트리 구조이므로, 수행시간을 위한 점화식은  $T(M) = 2T(M^{1/2}) + O(1)$ 이 되고, 이 점화식을 풀면,  $T(M) = O(\log M)$
- 아직까지 우리의 목표인  $O(\log \log M)$ 이 아님!

#### iv. `successor(V, x):` # $y$ 를 $x$ 의 successor라 표기하면

```
i = high(x)
j = successor(V.cluster[i], low(x)) # (1)

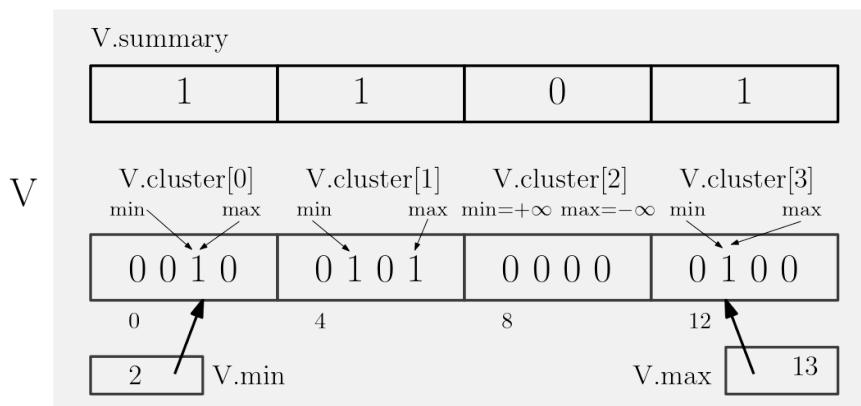
if j == None: # no successor in V.cluster[i]
 i = successor(V.summary, i) # (2)
 if i == None: # no cluster containing successor
 return None
 j = successor(V.cluster[i], -∞) # (3)

return index(i, j)
```

- (1)번:  $x$ 가 속한 cluster에서 successor가 있는지 먼저 재귀적으로 체크한다
- 만약 없다면  $j == \text{None}$ 이 리턴된다. 그렇다면 오른쪽의 cluster 중에서 처음으로 1이 등장하는 cluster를 (2)번 단계에서 찾는다. 이것은  $V.\text{summary}$ 에 대해 재귀적으로 successor 함수를 호출하면 된다
- 이렇게 찾은 cluster에서 처음 1을 다시 successor 함수를 호출해 찾으면 된다 → (3)번 단계
- 결론적으로 재귀 함수 successor를 3번 호출하게 되어, 수행시간 점화식은  $T(M) = 3T(M^{1/2}) + O(1)$ 이 된다. 이 점화식은 알고리즘 2보다 더 오래 걸린다.  $T(M) = O((\log M)^t)$  (여기서,  $t = \log_2 3$ ) 시간이 된다
- 더 빨리 처리할 수 있는 부분이 있는지 살펴보자

#### k. 알고리즘 4: Recursion (ver. 2)

- 알고리즘 3의 successor 코드의 (3)에서는 successor를 재귀적으로 호출한다. 이 호출은  $V.\text{cluster}[i]$ 에서 첫 번째 1의 offset 번호를 알기 위함이다. 만약,  $V.\text{cluster}[i]$ 의 첫 번째 1의 offset 번호를 미리 저장해 놓으면 재귀 호출 없이 상수 시간에 알 수 있다
- 이를 위해, van Emde Boas 트리  $v$ 에서 첫 번째 1의 offset 번호를 저장하는  $V.\text{min}$  멤버를 새로 정의해 저장한다. 즉,  $V.\text{min} = \underline{\text{현재 vEB인 } v\text{에서 첫 번째 1이 나타난 entry의 offset 번호}}$ 이다 (다음 그림 참조)



- 주의 1: 모든 재귀적인 vEB 트리에 대해 `min`과 `max` 값을 저장
- 주의 2: 바닥 레벨 ( $V.\text{size} == 2$ )에서는 배열 A 대신에 `min`과 `max` 두 변수로만 구성하면 된다
- 주의 3:  $\text{min} = +\infty$ ,  $\text{max} = -\infty$ 으로 초기화
- insert 함수는 `V.min`과 `V.max`를  $x$ 로 업데이트하는 코드만 앞에 삽입!

```
insert(V, x):

 1. V.min = min(x, V.min) # V.min update
 V.max = max(x, V.max) # V.max update

 2. if V.size > 2:
 a. insert(V.cluster[high(x)], low(x)) # insert x
 b. insert(V.summary, high(x)) # mark x
```

- 수행 시간에는 변화 없음!  $T(M) = O(\log M)$
- 위 그림에서 `insert(V, 10)`을 수행한다면, `V.min`과 `V.max`는 변하지 않는다. 10은 이진수로 1010이므로 `high(10) = 2`, `low(10) = 2`가 되어 `V.cluster[2]`에 해당하는 vEB에 `low(10) = 2`를 삽입하면 된다. 아래 그림에서는 재귀 자료구조가 없기에 `V.cluster[2][2] = 10` 된다. 마지막으로 `V.summary`에 `high(10) = 2`를 삽입하면 `V.summary[2] = 10`이 된다

vii. 이제 successor를 살펴보자

```
successor(V, x): # y를 x의 successor라 하면
 1. if V.size == 2: # 바닥 조건 처리
 if x == 0 and V.max == 1: return 1
 else: return None

 elif V.min < +∞ and x < V.min:
 return V.min

 2. else: # 일반적인 경우 처리
 i = high(x)

 # 삽입된 x보다 큰 값이 이미 삽입되어 있다는 의미임
 # 결국, y가 cluster[i]안에 존재한다는 뜻
 if low(x) < V.cluster[i].max:
 j = successor(V.cluster[i], low(x))
 else: # y가 첫 non-empty cluster에 존재
 # 해당 cluster는 V.summary에서의 successor!
 i = successor(V.summary, high(x))
 if i == None: return None
 j = V.cluster[i].min (재귀 호출 불필요!)
 return index(i, j)
```

viii. 알고리즘 3의 코드에서 x의 successor가 `V.cluster[i]`에 있는 경우와 없는 경우를 (1)의 재귀호출의 리턴 값이 `None`인지 비교해서 확인했다. 그런데 `V.cluster[i]`에서 마지막 1의 offset 번호를 `V.cluster[i].max` 변수에 저장해 놓았다면, `low(x) < V.cluster[i].max`이면 자신의 cluster 안에 자신보다 더 큰 값이 이미 삽입되어 있다는 의미이므로 cluster 안에 successor가 있다는 뜻이다

아니면, 다음 non-empty cluster에 successor가 있다는 뜻이 되어 V.summary에서 successor를 재귀호출하면 된다. 그러면, 해당 cluster의 min 값이 첫 1이 저장된 index이므로 우리가 찾는 successor가 된다

- 위 그림에서 successor(V, 4)를 해보자.  $x = 4 = 0100$ 이므로  $high(x) = 01 = 10$ 이고,  $low(x) = 00 = 0$ 이다. V.cluster[1].max가 V.cluster[1]의 가장 마지막 인덱스를 가리키는데, 이는  $low(x)$ 보다 크다. 즉,  $low(x)$  위치보다 오른쪽에, 같은 cluster 안에 1이 있다는 의미이므로 successor(x)는 같은 cluster안에 있다는 뜻이 된다. 따라서 재귀 호출하면 된다
  - 위 그림에서 successor(V, 9)를 해보자.  $x = 9 = 1001$ 이므로  $high(x) = 2$ ,  $low(x) = 10$ 이 된다. V.cluster[2]에는 1이 없으므로  $V.max = -\infty$ 이 되어  $low(x)$ 보다는 작다. 즉, successor(x)는 (존재한다면) 같은 cluster가 아닌 오른쪽의 다른 cluster에 있다. successor가 존재하는 cluster는 V.summary에서  $high(x)$ 의 successor를 찾는 재귀호출로 찾을 수 있다
  - if 문의 결과에 따라 재귀함수가 **정확히 한 번만 호출됨!**
  - 따라서 수행시간의 점화식은
- $$T(M) = T(M^{1/2}) + O(1) = O(\log \log M)$$
- **목표하는 수행시간!** 그러나 **insert가 아직  $O(\log M)$ 으로 느림!**

### 1. 알고리즘 5: Recursion (ver. 3) Recurse using lazy steps for min

- i. 알고리즘 4에서의 successor의 수행시간은 목표한 것과 동일
- ii. 알고리즘 4의 insert에서는 3번과 4번에서 두 번의 재귀호출을 함  
 $\rightarrow O(\log M)$ 이 됨.  $O(\log \log M)$ 이 되기 위해선 한 번의 재귀호출만 해야 함  
 $\rightarrow V.cluster[high(x)]$ 가 이미 non-empty인 경우에는 V.summary에 기록할 필요가 없는데도 항상 재귀 호출(4번 라인)을 수행함!  
 $\rightarrow$  이 불필요한 호출을 줄여야 함. (그러면 첫 번째 삽입될 때에는 재귀호출이 필요한데?)
- iii. 현재 레벨에서의  $V.min$  값은 cluster에 실제로 저장하지 않고  $V.min$  변수에만 저장한다! 나중에  $V.min$ 보다 더 작은 값  $x$ 가 삽입되면, 이 새 최소 값  $x$ 가  $V.min$ 에 저장되고  $V.min$ 에 저장되어 있었던 예전 최소 값은 **그제서야** cluster에 (재귀적으로) 저장한다 (즉, 재귀적인 저장 과정을 **최대한 미루는(lazy) 방식**으로 insert가 처리된다)

iv. `insert(V, x):`

```

1. if V.min == +∞:
 V.min = V.max = x # 최초 key 값은 무조건 V.min이
 return # O(1) time

2. if x < V.min:
 swap x and V.min # V.min에 저장된 key를 재귀 저장
3. if x > V.max:
 V.max = x # V.max update

4. if V.cluster[high(x)].min == +∞: # key in cluster
 insert(V.summary, high(x)) # summary 재귀적 기록
5. insert(V.cluster[high(x)], low(x)) # 실제 cluster 삽입

```

| V.summary |                      |              |              |                            |
|-----------|----------------------|--------------|--------------|----------------------------|
| V         | V.cluster[0]         | V.cluster[1] | V.cluster[2] | V.cluster[3]               |
|           | 0 0 0 0              | 0 0 0 0      | 0 0 0 0      | 0 0 0 0                    |
|           | 0                    | 4            | 8            | 12                         |
|           | <input type="text"/> | V.min        |              | <input type="text"/> V.max |

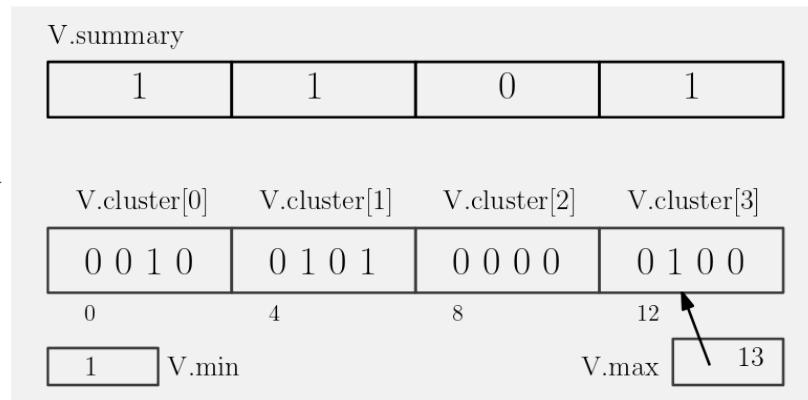
- `insert(V, 10), insert(V, 2), insert(V, 8)`
- `successor(V, 2)`
- `successor(V, 9)`

v. 4번 `if` 문이 참이 되면 연속해서 두 개의 재귀호출이 발생하는데?

- 그렇다면, 5번 라인의 `insert` 함수가 재귀 호출되어 1번 라인의 `if` 문을 만나면 `V.min == +∞`이 되어 상수시간에 `return`되어 실제로 5번 라인에서의 재귀 호출은  $O(1)$  시간에 완료된다! 결국, 한 번의 재귀호출만 실질적으로 발생하게 되어,  $O(\log \log M)$  시간을 보장한다

vi. 정리하면, `V`의 가장 작은 값은 `V.min`에 저장하고, 실제 cluster와 summary 구조에는 저장하지 않는다. 대신 `V.min`보다 더 작은 값 `x`가 `insert`되는 순간 `V.min`에 저장된 값이 더 이상 `min`이 아니므로 cluster 또는 summary 구조에 (재귀적으로) 저장되고, `x`는 `V.min`에만 저장된다

## m. Delete 연산



```

delete(V, x):
 if V.min == V.max:
 V.min = +∞
 V.max = -∞
 elif V.size == 2:
 if x == 0: V.min = 1
 else: V.min = 0
 V.max = V.min
 elif x == V.min: # 삭제할 key가 V.min인 경우
 # min 같은 V.cluster가 아닌 V.min에만 저장되어 있음!

 i = V.summary.min # first non-empty cluster
 # V.summary.min을 유지
 if i == +∞: # 현재 오직 하나의 key (V.min)만 존재
 V.min = +∞
 V.max = -∞ # 초기화~
 return

 # V.min 이외의 key가 존재하므로 (새로운) V.min을 찾기
 # V.min으로 set하고, 실제 cluster에서는 지움!
 x = V.min = index(i, V.cluster[i].min)

 delete(V.cluster[high(x)], low(x)) # first call
 # delete x

 if V.cluster[high(x)].min == +∞:
 # 지운 후 cluster가 empty가 되었다는 의미 → summary
 # entry 리셋 필요
 # None은 x가 cluster의 유일한 key 값이었다는 의미!
 # 위의 first call에서 x == V.min이 되고, i == None이 되어
 # 상수시간 update한 후, return 됨!
 # 결국, first call이 O(1) 시간에 수행되어
 # first + second call은 한 번의 재귀호출의 비용으로 가능!

 delete(V.summary, high(x)) # second call

```

```
update V.max
if x == V.max: # V.max가 단계 3에서 지워진 경우
 if V.summary.max == -∞: # x가 동시에 V.min임
 V.max = V.min
 else: # 새로운 v.max를 찾아서 update!
 i = V.summary.max
 V.max = index(i, V.cluster[i].max)
```

- 결국, 한 번만 재귀호출을 하는 셈이 되어 **O(log log M)**의 수행시간이면 충분!

n. 이 자료구조가 사용하는 메모리 양: **O(M)**

- i. 실제 저장된 key의 개수는 n이므로 메모리의 양도 n으로 표현되는 게 바람직하지만, M = O(n)인 경우에는 합리적이다
- ii. 만약, M >> n이라면 메모리 사용량이 매우 큰 자료구조가 됨

이후, 연구자들에 의해, cluster를 연속된 배열에 저장하는 것이 아니라 non-empty cluster만 해시 테이블로 저장하는 방식으로 O(n) 메모리만 사용하면 되도록 하였음!

### 3. Suffix array (suffix 배열)

- a. 문자열  $S = "banana"$ 에서 suffix는  $S[i:n]$ 을 의미한다. "a", "na", "ana", "nana", "anana", "banana"가 가능한 suffix이다. (반대는 prefix라 부름)
- b. 사전식 순서(lexicographic order): 사전(dictionary)의 나타난 문자열 순서
  - i. "ape"는 "apple"보다 사전에서 먼저 나타나므로 "ape" < "apple"
  - ii. "app"과 "apple"은 공통 prefix "app"을 가지고 있지만, 사전에선 더 짧은 "app"이 먼저 등장함. 따라서 "app" < "apple"
- c. suffix 배열 A는 문자열 S의 모든 suffix를 사전식 순서에 따라 정렬했을 때, 정렬에서의 순서를 저장한 배열로 정의 (가정: 문자열의 마지막을 구분하기 위해 \$문자를 추가함)

i.  $S = "banana"$

|                                                                          |                         |   |   |   |   |   |
|--------------------------------------------------------------------------|-------------------------|---|---|---|---|---|
| 0                                                                        | 1                       | 2 | 3 | 4 | 5 | 6 |
| suffixes = "banana\$", "anana\$", "nana\$", "ana\$", "na\$", "a\$", "\$" |                         |   |   |   |   |   |
| sort = "\$", "a\$", "ana\$", "anana\$", "banana\$", "na\$", "nana\$"     |                         |   |   |   |   |   |
| 6                                                                        | 5                       | 3 | 1 | 0 | 4 | 2 |
| A                                                                        | = [6, 5, 3, 1, 0, 4, 2] |   |   |   |   |   |

|             |    |    |   |    |   |    |   |
|-------------|----|----|---|----|---|----|---|
| <b>i</b>    | 0  | 1  | 2 | 3  | 4 | 5  | 6 |
| <b>A[i]</b> | 6  | 5  | 3 | 1  | 0 | 4  | 2 |
| <b>1</b>    | \$ | a  | a | a  | b | n  | n |
| <b>2</b>    |    | \$ | n | n  | a | a  | a |
| <b>3</b>    |    |    | a | a  | n | \$ | n |
| <b>4</b>    |    |    |   | \$ | n | a  |   |
| <b>5</b>    |    |    |   |    | a | n  |   |
| <b>6</b>    |    |    |   |    |   | \$ |   |
| <b>7</b>    |    |    |   |    |   |    |   |

d. 알고리즘 1: trivial method

- i. 모든 suffix를 만든 후, 가장 빠른 알고리즘으로 정렬하여 A를 계산한다
- ii. 정렬 할 때, 두 문자열 비교는  $O(n)$  시간 필요. 따라서  $O(n^2 \log n)$  시간

e. 알고리즘 2: optimal method

- i. suffix들은 한 글자씩 다르다는 성질을 이용하면 정렬을 이용하지 않아도  **$O(n)$**  시간이라는 매우 빠른 시간에 A를 계산할 있는 알고리즘이 제안됨!
- ii. 논문: Nong, Ge; Zhang, Sen; Chan, Wai Hong (2009). *Linear Suffix Array Construction by Almost Pure Induced-Sorting*. 2009 Data

Compression Conference. p. 193. doi:10.1109/DCC.2009.42. ISBN 978-0-7695-3592-0.

- iii. 구현: Yuta Mori: <https://sites.google.com/site/yuta256/sais>

f. 활용: 문자열 패턴 검색 (string pattern matching)

- i. 문자열  $S$ 에서 패턴 문자열  $P$ 가 나타나는 모든 곳을 찾기
- ii. 힌트:  $P$ 가 등장하는 곳을 찾는다는 건,  $P$ 로 시작하는 suffix를 찾는 것과 같기 때문에,  $A$ 를 이용해 이진탐색과 유사한 방식으로 검색할 수 있다

```
refined version from Wikipedia
input: S, P, A(suffix array)
output: (s, r) s.t S[s:n], S[s+1:n], ..., S[r:n] contain P

n = len(S), m = len(P)
def search(P):
 l = 0; r = n-1
 while l < r:
 mid = (l+r) / 2
 if P > S[A[mid]:n] : # 비교시간 O(m)
 l = mid + 1
 else:
 r = mid
 s = l; r = n-1
 while l < r:
 mid = (l+r) / 2
 if P < S[A[mid]:n] :
 r = mid
 else:
 l = mid + 1
 return (s, r)
```

- iii. 수행시간:  $O(m \log n)$

## 4. Range 트리

- a. 주로 orthogonal range searching 문제에 사용되는 트리 자료구조
- b. 예: 외대 글로벌 캠퍼스 남학생 중에 키가 170cm에서 179cm 구간(interval)에 있는 학생은 모두 몇 명인가? (또는 누구인가?)
  - i. 남학생의 키를 기준으로 오름차순으로 정렬한 후 170cm인 첫 학생을 이진탐색 (binary search)으로 검색한 후, 그 학생부터 차례대로 오른쪽 값을 보면서 179cm의 학생까지 세면 (선형 탐색, linear scan) 된다
  - ii. 정렬이 되어 있다고 하면, 이진탐색 =  $O(\log n)$  시간, 해당 키 구간에 있는 학생을 세는 시간 =  $O(k)$ 이다 (여기서  $k$ 는 해당 구간에 있는 학생 수임)
  - iii.  $k$ 는 전체 학생 수  $n$ 까지 커질 수 있으므로,  $O(k + \log n) = O(n)$  시간이 필요하다
  - iv. 그러나 한 번의 구간 질의(query)만 있는 것이 아니라 여러번 질의가 있다면, 질의 때마다 이진탐색과 선형탐색을 하는 건 매우 비효율적이다
  - v. 질의 응답 시간을 줄이기 위해서는 질의의 답을 쉽게 찾을 수 있도록 미리 전처리 과정을 통해 자료구조를 마련해 놓으면 된다
- c. 예 - 2차원 문제: 서울 시민 중에 월급이 200만원에서 300만원 사이를 받으면서 월 평균 기부액이 1만원에서 5만원 사이인 사람은 총 몇 명인가? (또는 누구인가?)
  - i. 입력 데이터는 2차원 평면의 점으로 표현 가능하다. x-축은 월급, y-축은 기부금액으로 정의하면 하나의 점은 한 사람의 (월급, 기부금액) 정보를 나타낸다. 그러면 전체 점 중에서 x-축의 구간 [200만원, 300만원]에 해당하는 사람들을 골라내고, 그 점들 중에서 다시 y-축 구간 [1만원, 5만원]에 들어가는 점만을 고르는 문제가 된다. 이 두 구간은 2차원에서 직사각형 [200만원, 300만원] x [1만원, 5만원]으로 정의되고, 이 직사각형이 질의 영역 (query region)이 된다
  - ii. 3, 4, … d차원 문제도 얼마든지 생각할 수 있음!
- d. Orthogonal range searching problem:
 

**입력:** d차원 공간에 주어진  $n$ 개의 점(point)으로 구성된 점 집합  $P$

**질의:** d차원의 orthogonal region (1차원: 구간, 2차원: 직사각형, 3차원: 직육면체 등)

**목표:** 질의를 최대한 빠르게 처리할 수 있는  $P$ 에 대한 자료구조를 효율적으로 구성함

**문제:** counting problem = 질의 영역의 점의 개수 출력  
reporting problem = 질의 영역의 점 자체를 출력 (\* 노트에서 설명 \*)
- e. **기준:** 아래 3가지 기준을 최소화 해야 함!
  - i. **preprocessing time** =  $P$ 에 대한 자료구조를 구성하는 시간
  - ii. **space** =  $P$ 를 위한 메모리 공간
  - iii. **query time** = 하나의 질의에 답하는 데 걸리는 시간
- f. (Orthogonal) range tree는 이를 위해 제안된 트리 자료구조
  - i. range tree는 균형이진탐색트리면 모두 사용 가능
  - ii. 트리 노드에 저장하는 정보의 내용이 달라지는 차이 뿐!

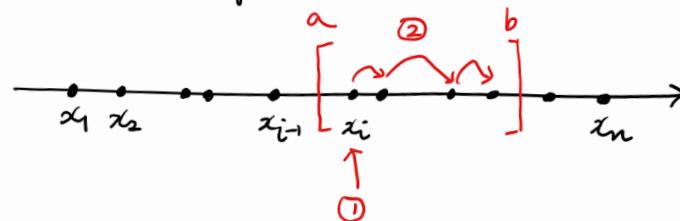
g. 1차원 orthogonal range searching 문제

- i.  $P = \{x_1, x_2, \dots, x_n\}$ , query 구간  $Q = [a, b]$

- ii. **방법 1:** sorted array에 점들을 저장해보자

- array에 저장한 후 x-좌표 값의 오름차순으로 정렬한다:  $O(n \log n)$
- \_\_\_\_\_ 방법으로 구간의 왼쪽 끝 점  $a$ 를 포함하는 두 점  $x_{i-1}$ 과  $x_i$ 를 탐색해 찾는다. (즉,  $x_{i-1} < a \leq x_i$ ):  $O(\log n)$
- $x_i$ 부터 오른쪽으로 가면서  $b$ 보다 작거나 같은 점들을 차례대로 출력한다:  $O(k)$  ( $k = [a, b]$ 에 포함되는 점들의 개수)

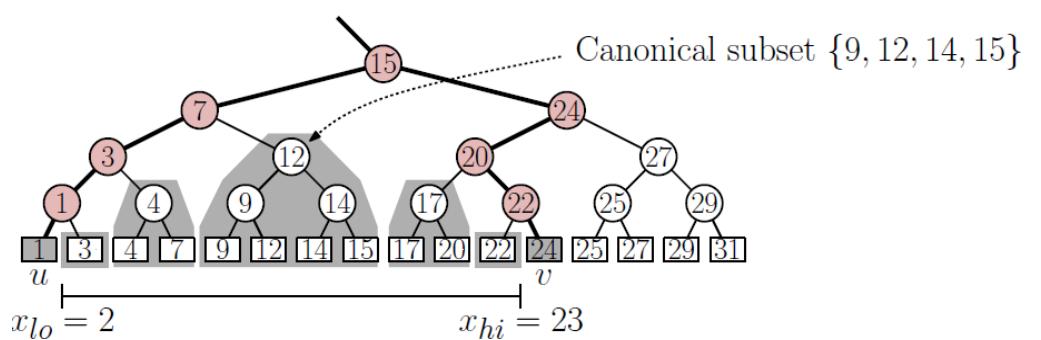
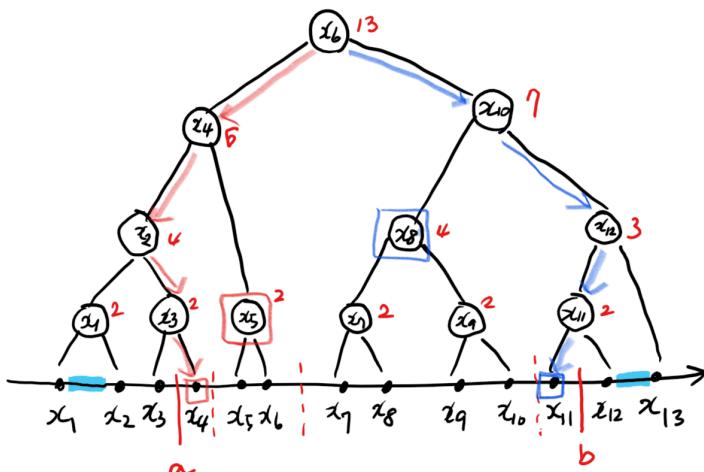
1D: sorted array



- preprocessing time =  
space =  
reporting (query) time =
- [질문] reporting이 아닌 counting (구간 안에 들어오는 점 개수만 세는 문제)을 위해 필요한 시간은?

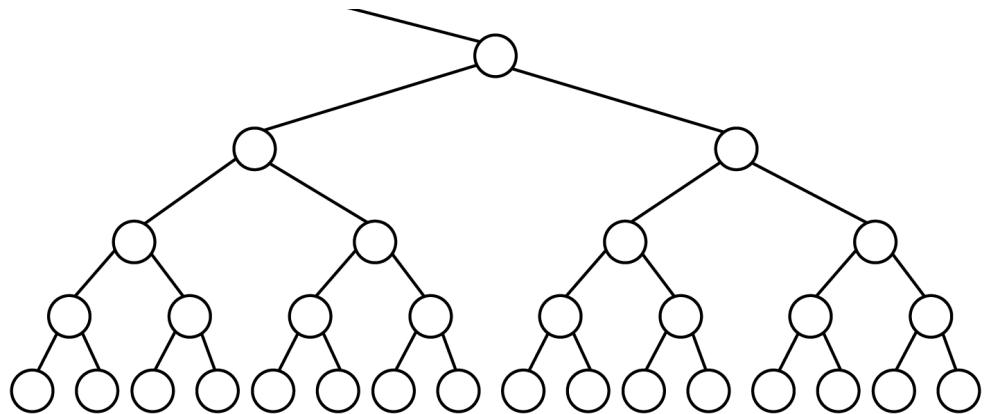
- iii. **방법 2:** 1차원 range tree에 점들을 저장해보자

- 완전이진트리 또는 균형이진탐색트리(예: AVL)의 리프 노드에 점들을 x-좌표 값을 key 값으로 저장한다:  $O(n) \sim O(n \log n)$
- a 값의 바로 오른쪽 리프 노드 x를 탐색한다:  $O(\log n)$
- b 값의 바로 왼쪽 리프 노드 y를 탐색한다:  $O(\log n)$
- x와 y 사이의 리프 노드에 저장된 값을 차례대로 출력한다:  $O(k)$   
어떻게? 두 가지 방법 존재:
  - 리프 노드들 사이를 연결리스트처럼 미리 연결해 놓으면 노드 x부터 y까지 차례대로 방문하면 됨:  $O(k)$
  - lca(x, y)에서 a까지의 경로 중에 오른쪽 부트리와 lca(x, y)에서 b까지의 경로 중에 왼쪽 부트리에 저장된 점들이 원하는 답이므로 이 부트리들의 노드를 모두 방문하면서 저장된 값을 출력함:  $O(k)$  (왜,  $O(k)$  시간이면 충분할까?)



[출처: [David Mount's lecture note](#)]

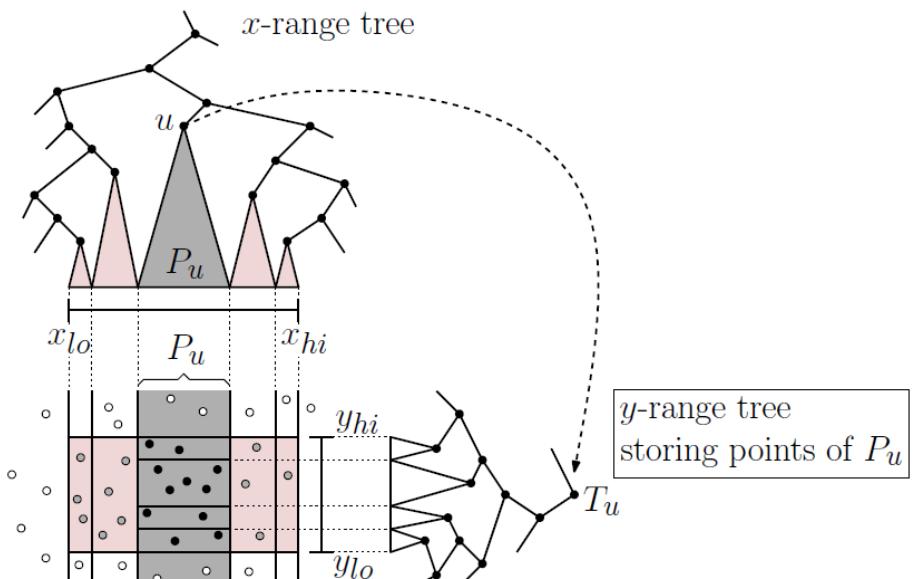
- 연습:



- preprocessing time =  $O(n \log n)$   
space =  $O(n)$   
reporting time =  $O(\log n + k)$
- [질문] counting 문제는 더 빠르게 해결할 수 있을까? 어떻게?

#### h. 2차원 orthogonal range searching 문제: 아래 그림을 참조할 것!

- i. 직사각형 질의 영역은  $Q = [a, b] \times [c, d]$ 이며, x-좌표 구간  $[a, b]$ 와 y-좌표 구간  $[c, d]$ 의 공통 영역으로 정의된다
- ii. x-좌표 값을 기준으로 1차원 range tree  $T_x$ 를 만든다
- iii. 1차원 문제에서처럼 x-좌표 구간  $[a, b]$ 에 포함되는  $T_x$ 의 노드  $O(\log n)$ 개를 선택한다. 이는  $[a, b]$  구간을  $O(\log n)$ 개의 구간으로 분할한 것을 나타낸다
- iv. 각 노드는 부트리의 리프노드에 저장된 점들의 집합을 나타낸다
- v. 이 점들 중에서 y-좌표 구간  $[c, d]$ 에 포함되는 점들만 골라 내야 한다. (어떻게?)
- vi. 이를 위해,  $T_x$ 의 각 노드  $v$ 에 1차원 range tree  $T_y(v)$ 를 독립적으로 하나씩 더 가지고 있게 한다. 이 트리는  $v$ 의 x-좌표 구간에 포함되는 점들의 y-좌표 값에 대한 range tree이다. (아래 그림 참조)
- vii. 그러면  $O(\log n)$ 개의 노드  $v$ 에 있는  $T_y(v)$ 에서 y-좌표 구간  $[c, d]$ 에 해당하는  $O(\log n)$ 개의 노드를 다시 선택한다
- viii. 결국,  $O(\log n) \times O(\log n) = O(\log^2 n)$ 개의 노드가 나타내는 점들을 모두 모으면  $Q$ 에 포함된 점들이 된다



[from [David Mount's lecture note](#)]

- ix. preprocessing time =  $O(n \log^2 n)$   
space =  $O(n \log n)$   
counting/reporting time =  $O(\log^2 n)/O(\log^2 n + k)$
- x. 2차원 range tree는 1차원 range tree의 각 노드에 1차원 range tree가 다시 저장된 재귀적인 자료구조이다
- xi. 같은 방식으로 d차원 range tree를 구성할 수 있다
- xii. preprocessing time =  $O(n \log^d n)$   
space =  $O(n \log^{d-1} n)$   
counting/reporting time =  $O(\log^d n)/O(\log^d n + k)$





**초판발행** - 2020년 3월 5일 - 자료구조 Python - Data Structures in Python  
**개정증보판** - 2024년 12월 1일 - 파이썬으로 자료구조 잡기 - Data Structures with Python  
**지은이** - 신찬수  
**펴낸곳** - 지어나눔, GeoNanum / 주소: (13611) 경기도 성남시 분당구 내정로 55 324-902  
**등록** - 2022년 4월 13일 제 2022-000036호

Published by GeoNanum

Copyright © 2022 신찬수, 지어나눔

이 책의 저작권은 신찬수와 지어나눔에 있습니다. 저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 전재를 금합니다.





스마트폰