

Data Structures
with
Python

Chan-Su Shin @hufs

Data Structures with Python¹



[컴퓨터공학부](#)

¹ 한국외대 컴퓨터전자시스템공학부의 <자료구조및실습>과 융복합SW 이중전공의 <자료구조> 등의 강의 내용을 정리한 것임.

² 신찬수, chansu@gmail.com, 이 자료는 한국외국어대학교 컴퓨터전자시스템공학부 신찬수 교수의 강의노트 중 일부임. 저자의 허락을 받고, 출처를 밝히고, 수정하지 않고, 상업적 목적이 없는 교육 활동에만 사용해야 함. 이 자료에서 인용한 이미지와 일부 내용은 참고용 교재와 wikipedia 등에 공개된 것임.

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

* 이 자료에서 인용한 이미지와 일부 내용은 wikipedia 등에 공개된 것임. 본 교재의 2019년 버전은 **Algorithms in Python** 이라는 이름의 책 (ISBN: 979-11-968591-3-8)으로도 출판되었음.

[저자 생각] 정식 출판을 염두에 두지 않고 정리한 내용으로 대충 그린 그림과 불친절한 설명에 다양한 형태의 오타와 오류가 양념으로 범벅된 책입니다. 그럼에도 시간과 정성을 들여 2년 넘게 숙성해 얻은 결과물로 자료구조와 알고리즘에 조금이라도 다가오고 싶은 분들에게 작은 도움이 되길 바랍니다.

[표지 이미지 - Cover Image] Photo by [Carl Raw](#) on [Unsplash](#)

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

0: 자료구조 + 파이썬 목차

- 시작하기: 자료구조와 파이썬 관련 교재 소개
- Python 기본 문법 (별책)

1. Algorithm, Data Structure, Time Complexity

- a. 가상 머신, 가상 언어, 가상 코드
- b. 시간, 공간 복잡도
- c. Big-O 표기법

2. Sequential Structures

- a. 동적 배열 (dynamic array)
- b. 스택 (stack), 큐 (queue), 디큐 (dequeue)

3. Linked List (연결 리스트)

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

4. Hash Table (해시 테이블)

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

5. Tree (트리)

- a. 우선순위 큐: 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)

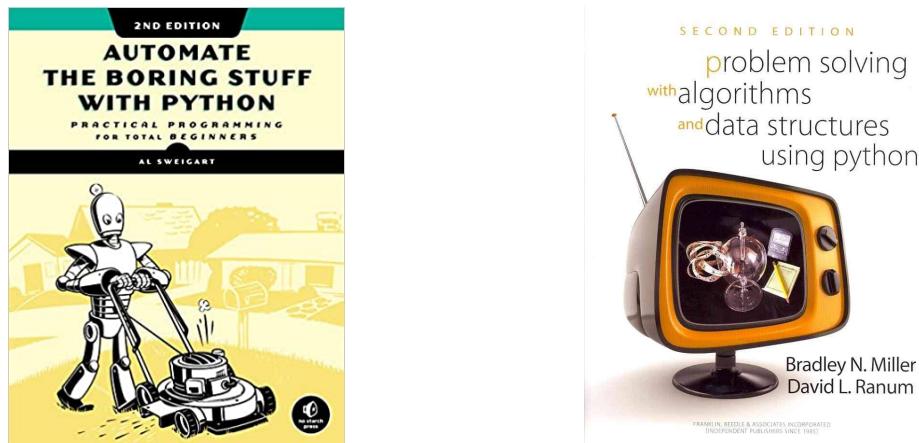
- a. 그래프 표현법과 기본 연산
 - 인접 행렬, 인접 리스트, 기본 연산의 복잡도
- b. 그래프 순회법(traversal)
 - DFS, BFS, 응용
- c. 최단 경로 알고리즘
 - Bellman-Ford, Dijkstra 알고리즘
- 다른 그래프 알고리즘은 알고리즘 과목에서 다룸!

7. [고급 자료구조] 기타 유용한 자료 구조

- 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

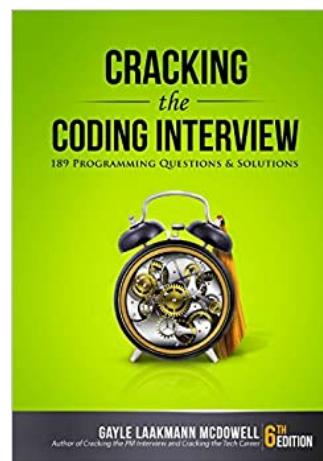
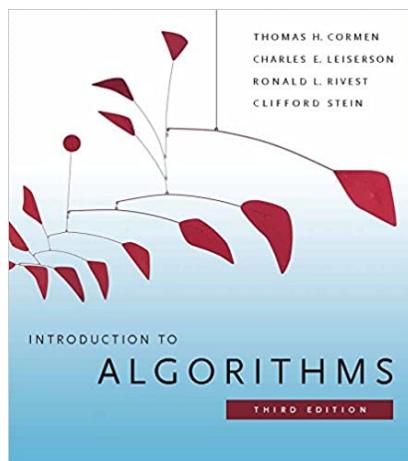
1: 시작하기

1. 왜 파이썬일까?
 - a. 직관적이라 배우기 쉽다
 - b. 내부, 외부 모듈이 다양하고 확장성이 좋아 응용 분야의 프로그래밍에 유리하다
 - 데이터베이스 응용, 데이터 분석/처리/시각화, 머신러닝/딥러닝, 웹 프로그래밍
 - c. 학습에 도움이 되는 (무료 동영상) 자료가 넘쳐난다 (아래 자료들 참고)
2. 충분히(!) 참고할 가치가 있는 자료들
 - a. 공식 파이썬 홈페이지: www.python.org (가장 정확한 정보 제공 및 공식 다운로드 사이트)
 - b. 동영상 무료 강의 사이트(기초):
 - 구름(goorm) 시스템 제공 Python 강의: [here](#)
 - 생활코딩 - Python/Ruby 강의: <https://opentutorials.org/course/1750>
 - programmers - Python 강의: <https://programmers.co.kr/learn/courses/2>
 - c. 영문 튜토리얼 사이트:
 - Corey Schafer 강의(강추): [here](#)
 - tutorialspoint <https://www.tutorialspoint.com/python3/index.htm>
 - diveintopython3 <http://www.diveintopython3.net/>
3. 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 (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 출간됨)
 - 이와 유사한 문제+정답 모음집 (무료) PDF 파일도 있음
<https://www.programcreek.com/wp-content/uploads/2012/11/coding-interview-in-java.pdf>

4. Python은?

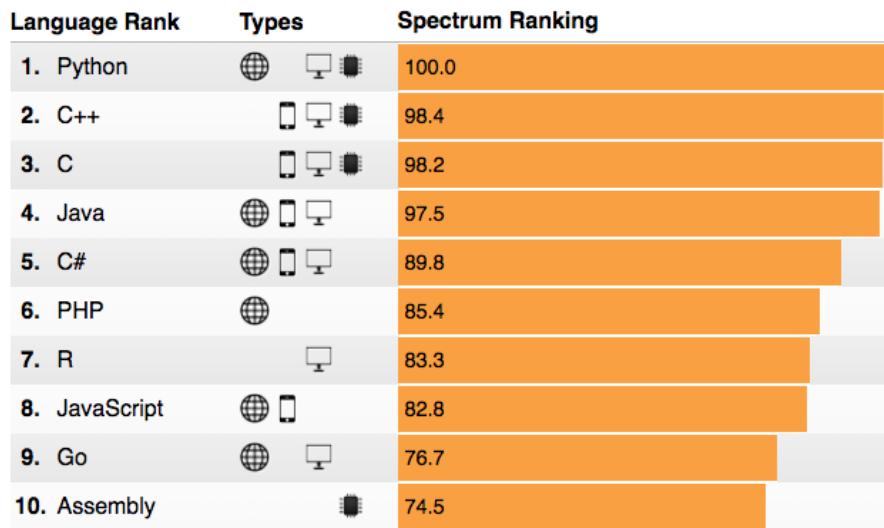
- a. 1989년에 [Guido van Rossum](#)에 의해 설계, 제안
- b. high-level, general-purpose, interpreted, object-oriented, dynamic language
고수준 -- 범용성 -- 대화형 -- 객체지향 -- 동적타입을 지원하는 언어
- c. [Wikipedia](#)에서의 설명 참조
- d. 어느 정도 인기가 있을까?
 - TOIBE 프로그래밍 언어 랭킹, IEEE Spectrum 프로그래밍 언어 랭킹

TOIBE Index for Programming Languages (2018-2020)

<u>Jul 2020</u>	<u>Jul 2019</u>	<u>Change</u>	<u>Programming Language</u>	<u>Ratings</u>	<u>Change</u>
<u>1</u>	<u>2</u>		<u>C</u>	<u>16.45%</u>	<u>+2.24%</u>
<u>2</u>	<u>1</u>		<u>Java</u>	<u>15.10%</u>	<u>+0.04%</u>
<u>3</u>	<u>3</u>		<u>Python</u>	<u>9.09%</u>	<u>-0.17%</u>
<u>4</u>	<u>4</u>		<u>C++</u>	<u>6.21%</u>	<u>-0.49%</u>
<u>5</u>	<u>5</u>		<u>C#</u>	<u>5.25%</u>	<u>+0.88%</u>
<u>6</u>	<u>6</u>		<u>Visual Basic</u>	<u>5.23%</u>	<u>+1.03%</u>
<u>7</u>	<u>7</u>		<u>JavaScript</u>	<u>2.48%</u>	<u>+0.18%</u>
<u>8</u>	<u>20</u>		<u>R</u>	<u>2.41%</u>	<u>+1.57%</u>
<u>9</u>	<u>8</u>		<u>PHP</u>	<u>1.90%</u>	<u>-0.27%</u>
<u>10</u>	<u>13</u>		<u>Swift</u>	<u>1.43%</u>	<u>+0.31%</u>
<u>11</u>	<u>9</u>		<u>SQL</u>	<u>1.40%</u>	<u>-0.58%</u>
<u>12</u>	<u>16</u>		<u>Go</u>	<u>1.21%</u>	<u>+0.19%</u>

Jul 2019	Jul 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.058%	-1.08%
2	2		C	14.211%	-0.45%
3	4		Python	9.260%	+2.90%
4	3		C++	6.705%	-0.91%
5	6		C#	4.365%	+0.57%
6	5		Visual Basic .NET	4.208%	-0.04%
7	8		JavaScript	2.304%	-0.53%

8	7	▼	PHP	2.167%	-0.67%
9	9		SQL	1.977%	-0.36%
10	10		Objective-C	1.686%	+0.23%
11	12	▲	Ruby	1.636%	+0.43%



출처: [IEEE Spectrum 2018](#)

5. Python Version 3.7.x 또는 3.8.x 설치 (강의와 실습용 버전)

- Window 운영체제 기준으로 설명하고 실습함
- [python download site](#) - 2020년 3월 기준 3.8.1이 최신 버전임
- 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:

- 코드 작성을 위한 에디터: notepad++ (추천, 기본설치된 메모판도 가능) [[download](#)]
- 코드 실행을 위한 터미널: Powershell(추천), cmd shell (둘 다 Windows 기본 내장)
- 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

- [!] - 꽤 중요한 설명, [?] - 점검용 질문, [?] - 강의내용에 따른 실습 또는 구현문제
- [?] - 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++ 입력 후 실행
 - d. notepad++의 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, notepad++를 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: Anaconda (추천: <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
12. IDE: **vscode** (추천:  [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: 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

14. 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, 텭+완성 기능

15. 강의 노트에 등장하는 외부 이미지나 자료 등은 wikipedia 등의 GFDL 또는 CC-BY-SA license 조건에서 인용된 것이다

자료구조

A: Data Structures, Algorithms, Time Complexity

1. 알고리즘(algorithm)

- a. 알고리즘의 어원은 9세기 페르시아(이란-이라크) 수학자 Al-Khwarizmi의 라틴어 이름인 algorismus와 수를 나타내는 그리스어 arithmos가 섞여 만들어졌다는 게 정설이다
[\[Wiki\]](#)
- b. 아래는 사우디아라비아 Jeddah에 위치한 KAUST 대학의 컴퓨터과학과가 위치한 건물 사진 (직접 촬영했음 ^^!)



- c. 알고리즘은 문제의 입력(input)을 수학적이고 논리적으로 정의된 연산과정을 거쳐 원하는 출력(output)을 계산하는 절차이고, 이 절차를 C나 Python과 같은 언어로 표현한 것이 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 적용하여 원하는 출력을 계산한다

2. 인류 최초의 알고리즘

- a. 그리스 수학자로 기하학의 아버지로 알려진 Euclid의 유명한 저서인 “Elements”(BC. 300)에 설명된 최대공약수 (Greatest Common Divisor: GCD)를 계산하는 알고리즘이 최초라고 알려져 있다
- b. 과학사 연구가들은 Euclid 이전의 Pythagoras 학파의 결과나 70여년 전의 다른 수학자의 결과가 더 앞서 있을 수도 있다고 주장한다
 - https://en.wikipedia.org/wiki/Euclidean_algorithm#Historical_development

- c. Pseudo code [[wiki](#)] (pseudo code는 엄밀한 코드가 아닌 설명 중심의 코드)

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

- d. 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. 여기서 큰 수가 작은 수보다 작을 때까지 작은 수만큼 줄이게 된다는 것은 결국 큰 수를 작은 수로 나눈 나머지를 구하는 과정과 같다. 결국 반복해서 뺄 것이 아니라 단순히 큰수 % 작은수를 하면 된다는 것을 알 수 있다

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

- e. 위의 코드를 Python으로 변환해보자

- 예: algorithm gcd(a, b) → def gcd(a, b):

- f. 만약 **a > b**라면, **gcd(a, b)**는 **gcd(a-b, b)**이고 동시에 **gcd(b, a%b)**가 된다. (왜?)
이 점을 이용하면 gcd 함수를 재귀함수로도 작성할 수 있다. 예: **gcd(16, 6) -> gcd(6, 4) -> gcd(4, 2) -> gcd(2, 0)**

- g. [구현] 직접 구현해보자: Python 함수 복습 겸!

- 3. 가상컴퓨터, 가상언어, 가상코드 (Virtual Machine, Pseudo Language, Pseudo Code)**
- a. 자료구조와 알고리즘의 성능(performance)은 대부분 수행 시간(시간복잡도: time complexity)으로 정의되는 것이 일반적이다
 - b. 이를 위해, 실제 코드(C, Java, Python 등)로 구현하여 실제 컴퓨터에서 실행한 후, 수행시간을 측정할 수도 있지만, HW/SW 환경을 하나로 통일해야 하는 어려움이 있다
 - c. 따라서, 가상언어로 작성된 가상코드를 가상컴퓨터에서 시뮬레이션하여 HW/SW에 독립적인 계산 환경(Computational Model)에서 측정해야 한다
 - d. **가상컴퓨터**(virtual machine)
 - 현대 컴퓨터 구조는 Turing machine에 기초한 von Neumann 구조를 따른다
 - 현재 가장 많이 사용하는 가상컴퓨터 모델은 (real) RAM(Random Access Machine) 모델이다
 - (real) RAM 모델은 CPU + memory + primitive operation으로 정의된다
 - 연산(operation, command)을 수행하는 CPU
 - 임의의 크기의 실수도 저장할 수 있는 무한한 개수의 레지스터(register 또는 word)로 구성된 memory
 - 단위 시간에 수행할 수 있는 기본연산(primitive operation)의 집합
 - a. A = B (대입 또는 복사 연산: B의 값을 A 레지스터에 복사)
 - b. 산술연산: +, -, *, / (나머지 % 연산은 허용 안되나, 본 강의에서는 포함한다)
 - c. 비교연산: >, >=, <, <=, ==, !=
 - d. 논리연산: AND, OR, NOT
 - e. 비트연산: bit-AND, bit-OR, bit-NOT, bit-XOR, <<, >>
 - e. **가상언어**(Pseudo Language)
 - 영어나 한국어와 같은 실제 언어보다 간단 명료하지만, C, Python 같은 프로그래밍 언어보다 융통성 있는 언어로, Python 유사하게 정의함. (수학적/논리적으로 모호함 없이 명령어가 정의되기만 하면 됨.)
 - 기본 명령어
 - A = B (배정, 복사 연산)
 - 산술연산: +, -, *, /, %
 - 비교연산: >, >=, <, <=, ==, !=
 - 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
 - 예: 배열 A의 n개의 정수 중에서 최대값을 계산하는 가상코드 (반드시 아래 형식을 따를 필요는 없음)

```
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

```

- 위 코드에서 배정연산, 비교연산 등이 사용된다

4. 알고리즘의 시간복잡도

- 가상컴퓨터에서 가상언어로 작성된 가상코드를 실행(시뮬레이션)한다고 가정한다
- 특정 입력에 대해 수행되는 알고리즘의 기본연산(primitive operation)의 횟수로 수행시간을 정의한다
- 문제는 입력의 종류가 무한하므로 모든 입력에 대해 수행시간을 측정하여 평균을 구하는 것은 현실적으로 가능하지 않다는 점이다
- 따라서 최악의 경우의 입력(worst-case input)을 가정하여, 최악의 경우의 입력에 대한 알고리즘의 수행시간을 측정한다

Checkpoint

알고리즘의 수행시간 = 최악의 경우의 입력에 대한 기본연산의 수행 횟수

- 최악의 경우의 수행시간은 입력의 크기 n 에 대한 함수 $T(n)$ 으로 표기 된다.
- $T(n)$ 의 수행시간을 갖는 알고리즘은 어떠한 입력에 대해서도 $T(n)$ 시간 이내에 종료됨을 보장한다
- 실시간 제어가 필요하고 절대 안전이 요구되는 분야(항공, 교통, 위성, 원자로 제어 등)에선 실제로 최악의 경우를 가정한 알고리즘 설계가 필요하기 때문에, 유효한 수행시간 측정방법이다
- 최악의 경우의 입력에 대해, 알고리즘의 기본연산(복사, 산술, 비교, 논리, 비트논리)의 횟수를 센다

- 예: n 개의 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 **A**의 저장된

값이 오름차순으로 정렬된 경우이다. (즉, 오름차순으로 정렬된 n개의 값이 저장된 배열 A가 최악의 경우의 입력이라는 의미)

■ 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
```

- 따라서 $T(n) = 2n - 1$ 이 된다
- $n = 10$ 이면 $T(10) = 19$ 가 되어 19번 이내의 기본연산을 수행한다는 뜻이고, $n = 200$ 이면, $T(200) = 399$ 번의 기본연산을 수행한다는 의미이다

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

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

5. Big-O 표기법

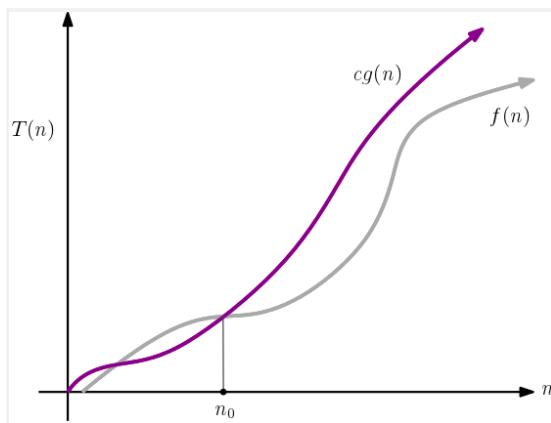
- 최악의 입력에 대한 기본연산의 횟수를 정확히 세는 건 일반적으로 귀찮고 까다롭다
- 정확한 횟수보다는 입력의 크기 n이 커질 때, 수행시간의 증가하는 정도(rate of the growth of $T(n)$ as n goes big)가 훨씬 중요하다
- 수행시간 함수 $T(n)$ 이 n에 관한 여러 항(term)의 합으로 표현된다면, 함수 값의 증가율이 가장 큰 항 하나로 간략히 표기하는 게 시간 분석을 간단하게 하는 데 큰 도움이 된다
- 예를 들어, $T(n) = 2n + 5$ 이면, 상수항보다는 n의 일차항이 $T(n)$ 의 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.
 - $T(n) = 2n + 5 \rightarrow T(n) = O(n)$
 - $T(n) = 3n^2 + 12n - 6 \rightarrow T(n) = O(n^2)$
 - [\[자료\]](#)

g. 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

h. [skip 가능] 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)
- $O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq cg(n) \text{ for all } n > n_0 \}$



- $T(n) = 3n^2 + 12n - 6 = O(n)$ 이라고 말할 수 없다. 이유는 위의 정의에 의하면, $3n^2 + 12n - 6 \leq cn$ 을 만족하는 n_0 값과 c 값을 찾을 수 없기 때문이다
 - n 의 값이 크지 않으면 위 부등식이 성립하지만 이차식의 증가율이 훨씬 커서 n 이 상당히 커지면 c 의 값을 어떻게 정하더라도 위의 부등식이 성립하지 않게 된다

i. 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

j. 예 2: (python 또는 pseudo code로 기술한 코드의 수행시간)

- **$O(1)$ 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: \log 의 밑은 2라고 가정: n 을 이진수로 표현했을 때의 비트수 계산 알고리즘


```
def number_of_bits(n):
    count = 0
    while n > 0:
```

```

n = n // 2
count += 1
return count

```

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

```

algorithm 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

```

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

```

algorithm 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

```

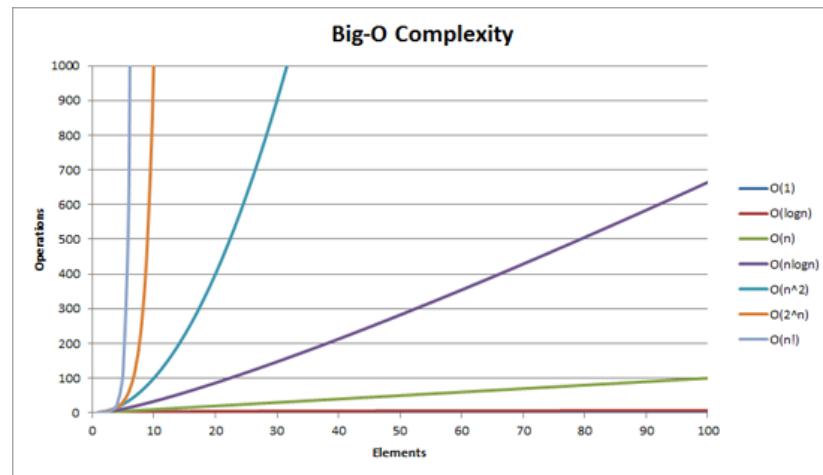
- **$O(2^n)$ 이상의 시간이 필요한 알고리즘:** exponential time algorithm: k번째 피보나치 수 계산하는 재귀 알고리즘

```

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

```

k. 비교 그림 [출처: [stackoverflow](#)]



I. 더 자세한 비교 그림 및 설명 [\[here\]](#)

m. Python에서 시간 측정법: time 모듈, timeit 모듈 이용 (파이썬 내용 참고)

n. 자료구조 작동 원리를 코드와 애니메이션으로 보여주는 사이트 소개

- 싱가포르 대학 visualgo: <https://visualgo.net/ko>

- [▣ 알고리즘 퍼즐 1] 독살 음모를 밝혀라!

왕이 마실 와인 8병 중 하나에 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, 몇 대의 검사 장비가 있으면 충분할까? (한 번 검사할 때에는 여러 병의 와인을 섞어 검사 가능하다)

- [▣ 알고리즘 퍼즐 2] 25마리의 말 중에서 가장 빠른 세 마리를 선택하고 싶다. 경기장엔 한 번에 다섯 마리 이하로 경주를 할 수 있고, 시계가 없어 기록을 잴 수 없고, 대신 순위만 알 수 있다. 한 경주에서 같은 순위는 없다고 가정한다. 최소 몇 번의 경주로 가장 빠른 세 마리를 알 수 있는가? (구글 인터뷰 질문)

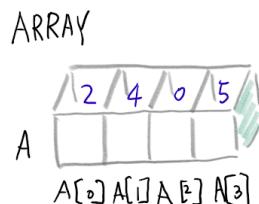
B: Array (List), Stack, Queue, Dequeue

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, array 등의 자료구조가 배열 자료구조임)

c. 예1: C 언어에서의 배열 A

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

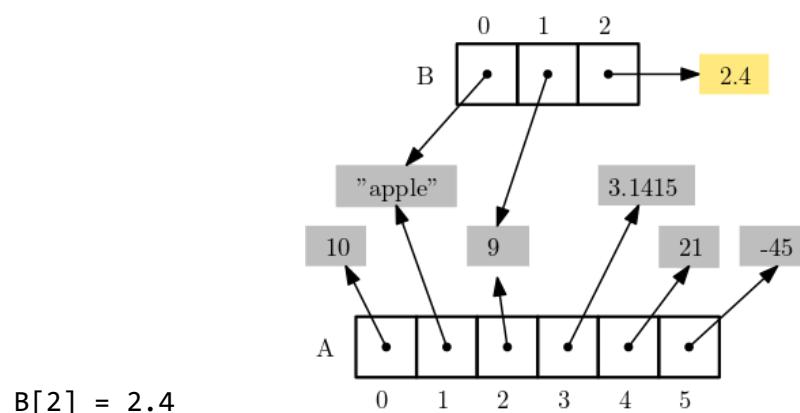
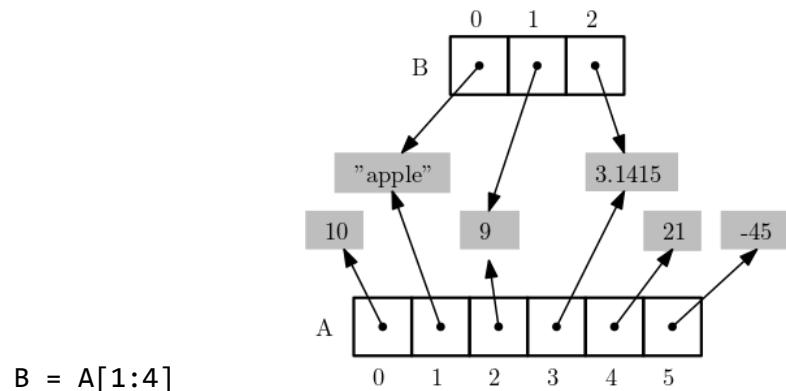
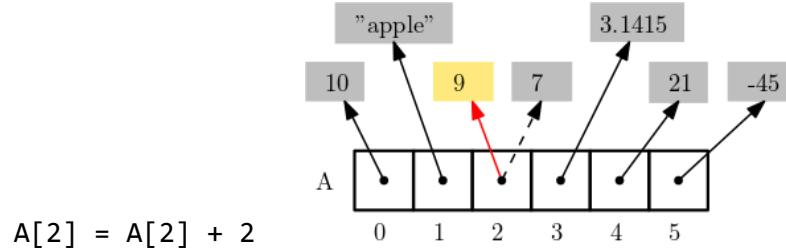
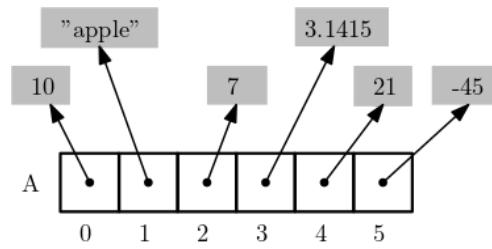


- ii. 만약, 2번째 셀(cell)에 저장된 값을 알고 싶다면, 그 셀의 시작 주소는 A의 시작 주소 + $2 * (\text{값이 차지하는 byte 수})$ 로 계산이 가능하다. (왜? 연속적으로 저장되어 있기 때문에)
- iii. 결국, 배열의 시작 주소, 저장된 값의 종류(바이트 개수), 몇 번째에 저장되어 있는지를 나타내는 인덱스 (index) 세 가지 정보만으로 값이 저장된 곳의 주소를 계산할 수 있다 [매우 중요한 사실!]
- iv. $A[2]$ 의 시작 주소 = $A[0]$ 의 시작 주소 + $2 * 4$ ($\text{index}=2, \text{sizeof(int)}=4$)
- v. 메모리 주소가 주어지면 단위 시간에 값을 읽거나 저장할 수 있음
- vi. 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

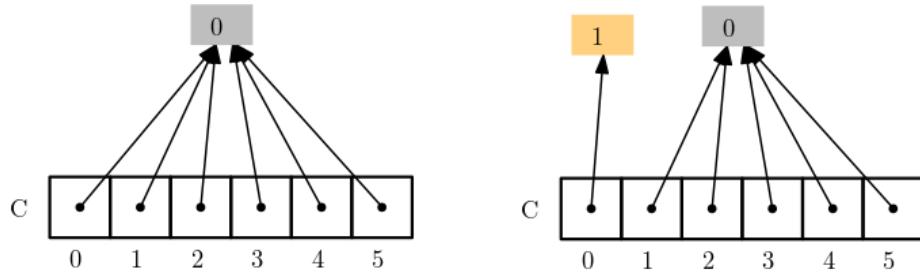
d. 예2: Python 언어에서의 list A (또는 tuple A)

- i. [중요] C 언어 배열의 셀에는 실제 값(데이터)이 저장된 형식이지만, Python 리스트의 셀에는 데이터가 아닌 데이터가 저장된 곳의 주소(address 또는 reference)가 저장된다
- 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: $A = [10, "apple", 7, 3.1415, 21, -45]$



$C = [0] * 6$
 $C[0] = 1$



- 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(i번째 오른쪽 값들은 왼쪽으로 한 칸씩 당겨져, cell의 수가 하나 감소)
 - o 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 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 (**slicing** 연산)
 8. value in A: 멤버십 연산자 - A에 value가 있으면 True, 없으면 False

v. 리스트는 동적 배열이다!

1. C 언어의 배열과의 또 다른 중요한 차이점은 list의 크기(셀의 개수)가 필요에 따라 자동으로 증가, 감소한다는 것이다

```
>>> A = []
>>> sys.getsizeof(A)
64 # 빈 리스트도 64 바이트 할당
>>> 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 할당 받아 새로운 리스트를 만들고 이전 리스트의 값을 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)라 부른다 (C에서의 배열은 동적 배열이 아니므로 사용자가 직접 상황에

맞게 처리해야 한다) 따라서 사용자는 배열의 크기를 전혀 신경쓰지 않아도 된다 [매우 중요]

4. 동적 배열인 list를 위해선 python 내부적으로 현재 list의 크기(capacity)와 list에 저장된 실제 값의 개수(n)를 항상 알고 있어야 한다 (이를 위한 내부 변수가 필요함) 이런 추가 정보를 위한 메모리가 필요하므로 빈 리스트 A는 0바이트보다 클 수 밖에 없다 (초기값 예: capacity = 1, n = 0)

5. 추가 메모리 이외에 연산 시간에도 영향을 준다. 특정 append 연산에서 메모리를 늘려야 한다면 이전 리스트의 값을 새로운 리스트로 일일이 이동해야 한다 (아래와 같은 방식으로)

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

6. 단계 1.b에서 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)

vi. 수행시간:

1. A[i] = A[j] + 1
 - o A[j] 값을 읽은 후 (단위시간) 1을 더하고 (단위시간) A[i]에 쓴다 (단위시간) (읽기/쓰기: O(1) 시간)
2. A.append(5)
 - o 맨 오른쪽에 삽입. 평균적으로 O(1) 시간
3. A.pop()
 - o 맨 오른쪽 값을 삭제 후 리턴. 평균적으로 O(1) 시간
4. A.pop(3)
 - o A[4], ..., A[len(A)-1] 값이 왼쪽으로 한 칸씩 이동하는 시간 필요
 - o 최악의 경우가 A.pop(0)일 때이므로 O(len(A)) 시간

5. A.insert(0, 10)
 - A[0], ..., A[len(A)-1]을 모두 오른쪽으로 이동하므로 A의 크기만큼 시간 필요
 - 최악의 경우 (+ 평균적인 경우)에 $O(len(A))$ 시간
6. A.remove(9)
 - 처음으로 등장하는 9를 찾아 제거. 최악의 경우 (+ 평균적인 경우)에 $O(n)$ 시간 필요
7. A.index(9), A.count(9)
 - 9가 처음으로 등장하는 index와 총 횟수를 계산해야 하므로 최악의 경우 $O(len(A))$ 시간 필요

vii. 리스트는 메모리 인심이 좋아 메모리 낭비가 큼

```
>>> A = []
>>> sys.getsizeof(A)
64                                     # 빈 리스트도 64 바이트 무조건 할당
>>> 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
```

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

ix. [?] 나만의 동적 배열을 구현해보자 (리스트 클래스를 설계해보자)

e. 예3: Python의 numpy 모듈에서 제공하는 array 자료구조

i. 대용량 데이터 처리에 효율적 → 데이터 사이언스, 머신러닝 코딩에 적합

ii. numpy 모듈이 필요함: 보통 import numpy as np로 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 접근가능
```

5. 배열 단위의 연산이 매우 빠름 (장점)

- o 2차원 array (행렬)에 대한 연산 등의 alebra 연산에 최적화되어 설계 → **빅 데이터, 머신러인 데이터 처리에 적합**

iv. 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)
>>> 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]])
```

2. Stack: LIFO, Queue: FIFO, Dequeue; Stack + Queue

- a. 데이터 값을 저장하는 기본적인 구조로 일차원의 선형(linear) 자료구조이다
- b. (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
- c. Stack은 가장 최근에 저장된 값 다음에 저장되며, 가장 최근에 저장된 값이 먼저 나간다: **LIFO**(Last In First Out) 원칙
- d. Queue는 가장 최근에 저장된 값 다음에 저장되지만(stack과 동일) 가장 오래전에 저장된 된 값부터 나간다: **FIFO**(First In First Out, 선착순) 원칙
- e. Dequeue는 stack과 queue의 연산을 모두 지원하는 자료구조이다

3. Stack

- a. 데이터는 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, val):
        self.items.append(val)

    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)
print(S.top())
print(S.pop())
print(len(S))
print(S.isEmpty())
```

e. [?] 위의 5개의 연산의 수행시간을 Big-O로 표기하면 어떻게 되는가?

f. 스택의 사용 예 1: 괄호 맞추기

- i. 수식 $(2+5)*7 - ((3-1)/2+7)/4$ 라고 하면, 왼쪽괄호와 오른쪽괄호가 쌍을 이루 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 일치함

iv. Pseudo 코드:

```
def parChecker(parSeq):
    S = Stack()
    for each symbol in parSeq:
        if symbol is "(":
            S.push(symbol)
        else: # symbol == ")"
            if S is empty: # if nothing matched
                return False
            else: # symbol == ")"이고,
                  # 스택에 저장된 건 "(" 뿐이므로
                S.pop()
```

```

if S is empty: # 스택에 남아 있는 게 없어야 짹짓기 완성
    return True
else:
    return False

```

- v. (연습문제) 만약, 중괄호 {}, 대괄호 []도 함께 섞여있는 일반적인 경우에 대해서도 위의 코드를 수정해서 짹 맞추기 코드를 작성할 수 있다

1. symbol이 {[중 하나면 스택 push
2. symbol이]) 중 하나라면, 스택의 top에 있는 왼쪽 괄호가 같은 종류여야 함 (같다면 pop)

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

- i. 참고용 교재 b: [3.9](#) 참조
- ii. 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 Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

iii. infix → postfix 방법

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

예: A + B * C → (A + (B * C)) → 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 찾아나가는 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 \Rightarrow 우선순위가 높은 연산자부터 차례로 나열
- vii. **관찰 2:** $(A + B) * C \rightarrow A B + C *$
 1. +는 *보다 먼저)를 만나기 때문에,) 다음에 위치해야 옮바르다
 2. 즉,)의 우선순위가 *보다 높기 때문에 + 연산자는) 다음에 위치하게 된다
- viii. 결국, 어떤 연산자의 위치는 자신보다 우선순위가 높거나 같은 연산자가 오른쪽에서 나타날 때까지 기다린 후, 그 연산자 다음에 위치하면 된다. 이를 위해 연산자는 자료구조에 저장된 채로 대기해야 한다
- ix. 거꾸로 말하면, 현재의 연산자보다 (자료구조에 이미 저장된) 우선순위가 높은 연산자는 자신보다 왼쪽에 위치해야 한다
- x. **Pseudo 코드:** infix \rightarrow postfix
 1. 괄호와 연산자를 저장하기 위한 스택 `opstack = Stack()` 준비
 2. Postfix 수식(결과)을 저장하기 위한 리스트 `outstack = []` 준비
 4. 리스트 `exp`에는 항과 연산자가 infix 수식 형태로 입력되어 있음
 5. **for each token in exp:** # 연산자 피연사자를 `token`이라 부름
 - a. **if** `token == operand`: `outstack.append(token)`
 - b. **if** `token == '('`: `opstack.push(token)`
 - c. **if** `token == ')'`:
 - o `opstack`에 저장된 연산자를 (를 만날 때까지 계속 pop한 후 `outstack.push(token)`)
 - d. **if** `token in '+-*/'`:
 - o `opstack`에 있는 자신보다 우선순위가 높거나 같은 연산자는 차례로 모두 pop한 후 `outstack`에 append함
 - o `opstack.push(token)`
 6. `opstack`에 남아 있는 연산자를 모두 pop한 후 `outstack`에 append함
 7. `print(outstack)`
- i. [구현 과제] 코드로 구현하는 것은 연습문제로 해 보길
- b. **스택의 사용 예 3: postfix 수식을 실제로 계산해 보기**

- 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. [infix → postfix → computation] 계산기 코드를 완성해보자

아래와 같은 세 개의 함수를 작성합니다.

`get_token_list(expr):`

- `expr`은 문자열로 수식을 나타낸다
- `expr`을 연산자와 피연산자로 나눈 후 리스트에 담아 리턴한다. (주의: 연산자, 피연산자 모두 문자열 형식으로...)
- [주의1] 연산자는 `+, -, *, /, ^` 5가지 이항 연산자만 다루고, **연산자와 피연산자 사이에 공백이 올 수도 있고 공백이 없을 수도 있음**에 유의하자!
- [주의2] 피연산자는 `float`로 변환한다
- [주의3] **한 자리 이상의 실수**가 피연산자가 등장할 수 있다!

`infix_to_postfix(token_list):`

- `token_list`는 수식의 연산자 피연산자가 `infix` 수식의 순서대로 저장된 리스트이다
- 이 `token_list`를 `postfix` 수식으로 변환하고 그 결과를 리스트에 담아 리턴한다

`compute_postfix(token_list):`

- `postfix` 형식의 `token_list`에 대한 계산 값을 리턴한다

- **입력:** `infix` 수식의 문자열을 입력받는다

- **출력:** 입력 수식을 계산한 (실수) 값을 출력한다

- 스택 예 4: nearest smaller element 계산하기

- 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)
 - 단, B[0] = None (not defined)
- 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]
```

- 빠른 방법: $O(n)$ 시간, 스택 S를 마련해서 아래 단계를 수행

- S[-1] < A[i] 이면 B[i] = S[-1], S.push(A[i]) 수행
- 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
- 이 과정을 반복하면 해를 정확히 구할 수 있는지 살펴보자
- 예: A = [1, 3, 4, 2, 5, 3, 4, 2]
 $\begin{aligned} S=[1] \rightarrow S=[\textcolor{red}{1}, 3] \rightarrow S=[1, \textcolor{red}{3}, 4] \rightarrow S=[\textcolor{red}{1}, 2] \rightarrow S=[1, \textcolor{red}{2}, 5] \rightarrow S=[1, \textcolor{red}{2}, 3] \rightarrow \\ S=[1, 2, \textcolor{red}{3}, 4] \rightarrow S=[\textcolor{red}{1}, 2] \\ \rightarrow B = [\text{None}, 1, 3, 1, 2, 2, 3, 1] \end{aligned}$
- 수행시간은?
 - Amortized 수행시간 분석의 간단한 예

- [マイ크] 스택과 관련된 인터뷰 문제 몇 가지

- [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
 - [힌트] push 할 때마다 현재까지의 최소값을 유지하고 기억!
 - 가능하면 사용하는 메모리의 양을 최소화하는 방법을 마련해 볼 것!
- [질문 2] 스택을 정확히 2개 이용해서 큐를 구현하라. 즉, 스택 2개만 사용해서 enqueue, dequeue를 구현해야 한다. 단, 연산 시간은 최소화해야 한다
 - [힌트] 연산시간은 파이썬 리스트의 append 시간 분석처럼 평균 시간으로 $O(1)$ 시간에 가능하도록 하면 어떨까?

3. Queue

- a. Queue 클래스를 정의한다 (Stack 클래스와 매우 유사 - LIFO vs. FIFO)
- b. 지원 연산은 enqueue, dequeue, isEmpty, front, len
 - i. enqueue(val): value를 큐의 오른쪽(rear)에 삽입 (push와 같음)
 - ii. dequeue(): 가장 왼쪽에 저장된 값을 삭제 후 리턴
 - iii. front(): 가장 왼쪽에 저장된 값을 (삭제하지 않고) 리턴
- c. 이 구현은 연습문제로 남김
- d. [?] 위의 5개의 연산의 수행시간을 Big-O로 표기하면 어떻게 되는가?
 - i. 상수시간보다 더 걸리는 연산이 존재하는가? 그렇다면 그 이유는?
- e. dequeue를 상수시간에 실행하기 위해선, dequeue가 될 값의 인덱스(index)를 저장하고 관리한다 → dequeue가 되면, 인덱스 값이 하나 증가하여 다음 dequeue 될 예정인 값의 인덱스를 가리키도록 관리한다!

- f. 코드:

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

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

    def dequeue(self):
        if self.front_index == len(self.items):
            print("Queue is empty")
            return None          # dequeue할 아이템이 없음을 의미
        else:
            x = self.items[self.front_index]
            self.front_index += 1
            return x
```

- g. 큐 사용 예1: 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
 - 1. 자세한 설명: <https://www.youtube.com/watch?v=uCsD3ZGzMgE>
- iii. 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.
- iv. 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

```

4. Dequeue

- a. 왼쪽과 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
'g' 'i'

```

- c. Dequeue의 사용 예: Palindrome 검사 코드
 - i. Palindrome은 왼쪽부터 읽어도 오른쪽부터 읽어도 같은 문자열을 말한다
 - 1. 예: 기러기, radar, madam 등
 - ii. 방법 1: 문자열 s와 s를 reverse한 문자열이 같다면 palindrome임
 - 1. 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 증가시는 방식으로 i == j이거나 i > j가 될 때까지 반복하면 된다
 - iv. 방법 3: dequeue를 사용함

1. `dequeue`에 문자열을 저장한 후 양쪽에서 하나씩 빼면서(`pop`과 `popleft` 이용) 같은지 비교하는 것을 반복함. 다르다면 계속할 필요없이 `palindrome`이 아님

2. 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. $A = [2, 1, 4, 5, 3, 4, 6, 2], k = 4$ ($B[0] \dots B[2] = \text{None}$)

----- → 1	$Q = [1, 4, 5]$	$B[3] = 1$
----- → 1	$Q = [1, 3]$	$B[4] = 1$
----- → 3	$Q = [3, 4]$	$B[5] = 3$
----- → 3	$Q = [3, 4, 6]$	$B[6] = 3$
----- → 2	$Q = [2]$	$B[7] = 2$
- iii. 윈도우를 왼쪽에서 오른쪽으로 한 칸 이동하면, 왼쪽 수가 하나 빠지고 오른쪽 수가 하나 윈도우에 들어온다. 윈도우에 포함된 수들을 관리하는 `deque dQ`를 마련한 후, 아래 과정을 반복 실행한다
 1. `dQ`에는 윈도우에 포함된 값부터 오름차순으로 저장함.
 2. `dQ.front`의 값이 윈도우를 떠나면 `dQ.popleft()`
 3. 현재 $A[i]$ 값보다 크거나 같은 수를 모두 `dQ.pop()` 실행 (이유는?)
 4. `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
 1. 입력: frequency
 2. 출력: wave 파일
 - iv. Karplus-Strong 알고리즘
 1. 변수 소개
 - sampleRate = 44100 (보통 CD의 값이 44,100 Hz)
 - 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)
 2. 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
```
 3. buf[0]이 삭제되는 대신 $0.996 * (\text{buf}[0]+\text{buf}[1])/2$ 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.tostring()

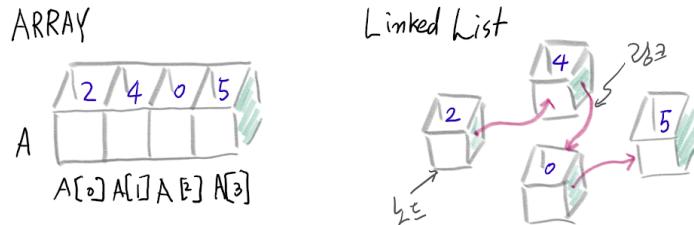
def writeWAVE(fname, data):
    # open file
    file = wave.open(fname, 'wb')
    # WAV file parameters
    nChannels = 1      # mono channel
    sampleWidth = 2     # 2바이트 샘플 = 16 bits
    frameRate = 44100
    nFrames = 44100
    # set parameters
    file.setparams((nChannels, sampleWidth, frameRate,
                    nFrames, 'NONE', 'noncompressed'))
    file.writeframes(data)
    file.close()

```

vi. 위의 두 함수를 호출해서 원하는 노트들을 만들어보고, 생성된 wave 파일을 재생해보자!

C: Linked List

1. 연결리스트(**linked list**)는 파이썬의 리스트(list)와는 이름만 비슷하지 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. 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 next 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) # 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
        v = self.head   # 코드 이해했다면 파이썬 증급 이상!
        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 ->를 넣어 출력)
# 위 문장의 의미를 해석했다면, 최소 파이썬 증급 이상
```

```
def __len__(self):
```

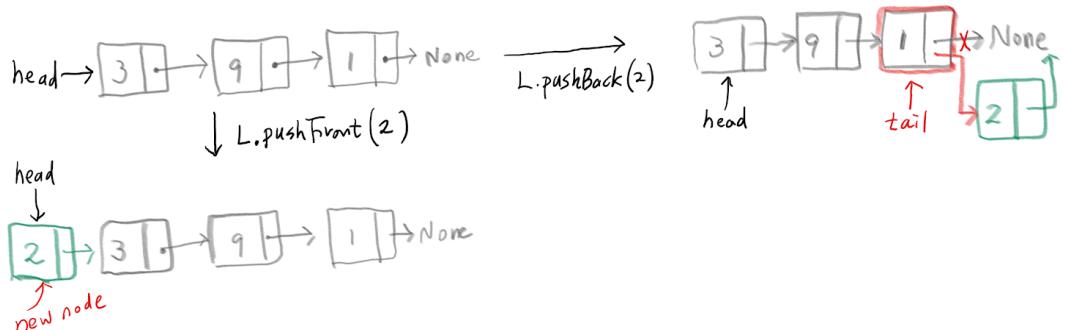
```
    return self.size # len(A) = A의 노드 개수 리턴
```

- 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.popFront(): L의 첫 노드(head 노드)를 삭제한 후 key값을 리턴
- iv. L.popBack(): L의 마지막 노드를 삭제한 후 key 값을 리턴
- v. L.search(key): L에서 key 값을 갖는 노드를 찾아 리턴
- vi. L.remove(v) : L에서 노드 v를 제거

- e. 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
    self.head = new_node
    self.size += 1
```

- ii. pushBack은 그림에서 보듯 마지막 노드(tail)를 다음에 삽입이 되므로 tail 노드의 next 링크가 새로운 노드로 변경되어야 한다. [주의] tail 노드가 None 이라면, 즉 빈 리스트라면 새로운 노드가 리스트의 head 노드가 된다

```
def pushBack(self, key, value=None):
```

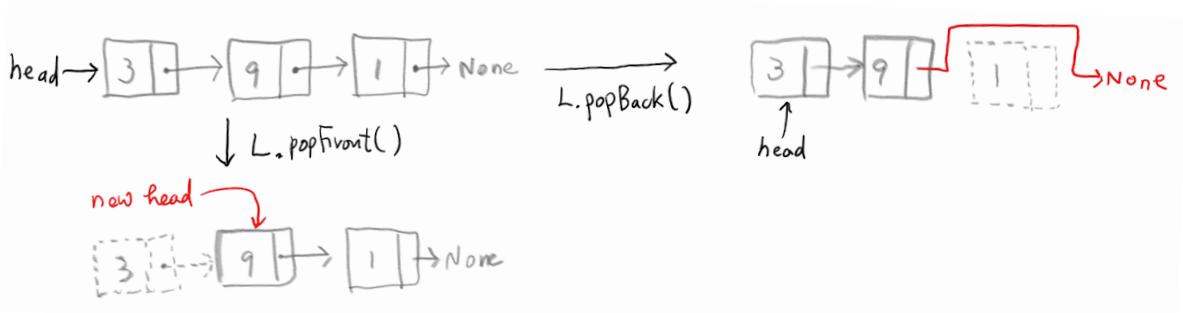
```
    new_node = Node(key, value)
    if self.size == 0:          # empty list
```

```

        self.head = new_node # new_node 가 head가 됨
    else:
        tail = self.head
        while tail.next != None: # tail 노드 찾기
            tail = tail.next
        tail.next = new_node
    self.size += 1

```

f. popFront vs. popBack



- i. popFront는 리스트의 head 노드를 삭제하고 key 값을 반환하는 함수로, 두 가지로 나눈다: 빈 리스트라 지울 head가 없는 경우와 그렇지 않은 경우

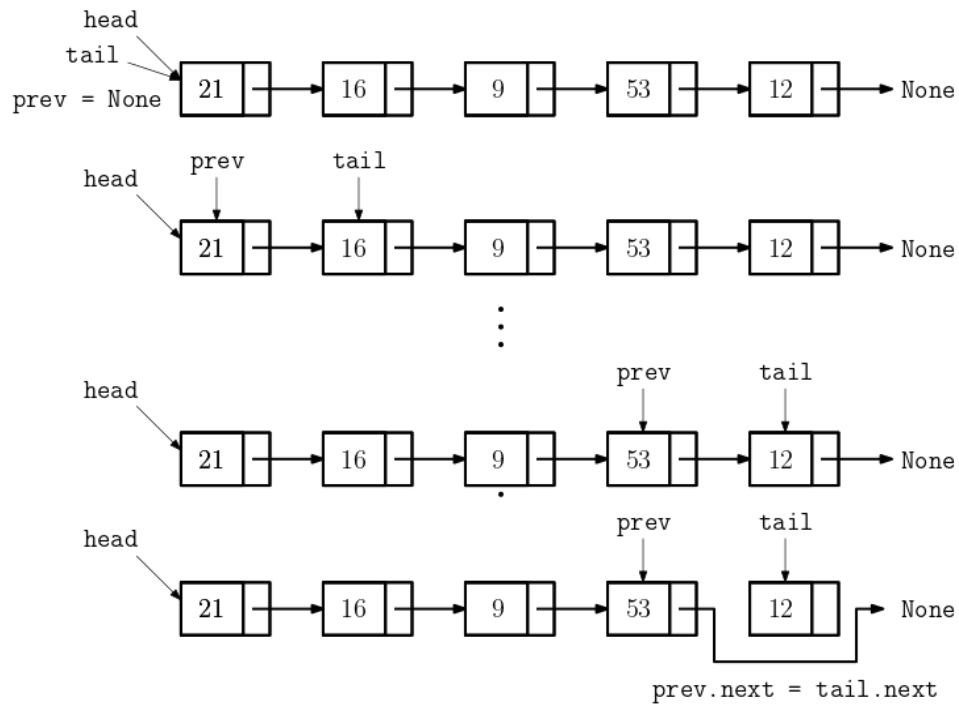
```

def popFront(self):
    if self.size == 0: # or len(self)==0
        return None # return None if it's empty
    else:
        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. popBack은 tail 노드를 찾아 지우고 key 값을 반환하는 데, tail 노드의 전 노드를 알아야 한다 (전 노드의 링크를 수정해야 하므로)

- 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



```

previous, tail = None, self.head
while tail.next != None: # what if current.next == None
    previous = tail
    current = 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
            del 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. `remove(v)`: 노드 v를 리스트에서 제거하는 함수

- i. 경우 (1): 리스트가 비어 있거나 노드 v가 None인 경우 → do nothing
- ii. 경우 (2): 노드 v가 head 노드인 경우 → `popFront` 호출하여 처리
- iii. 경우 (3): 노드 v의 전 노드 w를 찾는 후 (`popBack`의 경우처럼) `w.next = v.next`로 w의 링크를 수정한다
- iv. [?] 실제 코드를 작성해 볼 것

- i. [?] 연산의 시간복잡도

- i. 대상이 되는 노드가 `head` 노드로부터 k번째 떨어진 노드라고 가정

<code>pushFront</code>		<code>pushBack</code>	
<code>popFront</code>		<code>popBack</code>	
<code>search</code>		<code>remove</code>	

- j. [?] 한방향 연결 리스트와 배열의 장단점은 무엇인가?

- i. 장점:
- ii. 단점:

- k. [?] [해보기-easy-medium] 한방향이중연결리스트의 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
    while b:
        if b:
            c = b.next
            b.next = a
        a = b
        b = c
    self.head = a # 왜?
```

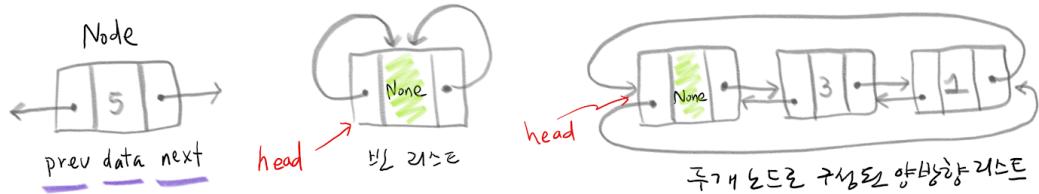
- iii. **방법 2:** 재귀 함수로 구현해보자 (가능하다면)

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

- I. **Running technique:** 한방향 연결리스트에서 `tail` 노드와 `prev` 노드를 찾는 방법에 쓰인 기법으로 두 개의 (포인터) 변수를 사용해 원하는 위치의 노드를 계산하는 방법
- i. `prev = None, tail = L.head`로 `prev`가 `tail`의 한 노드 뒤에서 따라가면서 `tail`이 실제 `tail` 노드에 도착하면, `prev`는 `tail` 노드 전 노드를 가르킴!
 - ii. **[?] 인터뷰 문제 1]** `find_kth_node_from_tail(L, k)`:
 - `tail` 노드로부터 k 번째 전에 있는 노드를 찾아라. (단, 리스트 `L`의 노드 개수는 모른다고 가정한다)
 - iii. **[?] 인터뷰 문제 2]** `find_middle_node(L)`:
 - 리스트 `L`의 노드 개수를 모른다고 가정하고, `L`의 중간에 위치한 노드를 찾아라! (`L`의 노드 개수가 짝수면 중간의 두 노드 중 아무 노드라도 정답)

4. 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 포함해 왼쪽으로도 이동 가능하도록 한다
 - 마지막 노드와 첫 노드가 서로 연결된 원형(circular) 리스트를 가정한다
 - 첫 노드, 즉 **head 노드는 항상 dummy 노드**가 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. 노드 클래스:

```
class Node:
    def __init__(self, key=None):
        self.key = key # 노드에 저장되는 key 값
        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

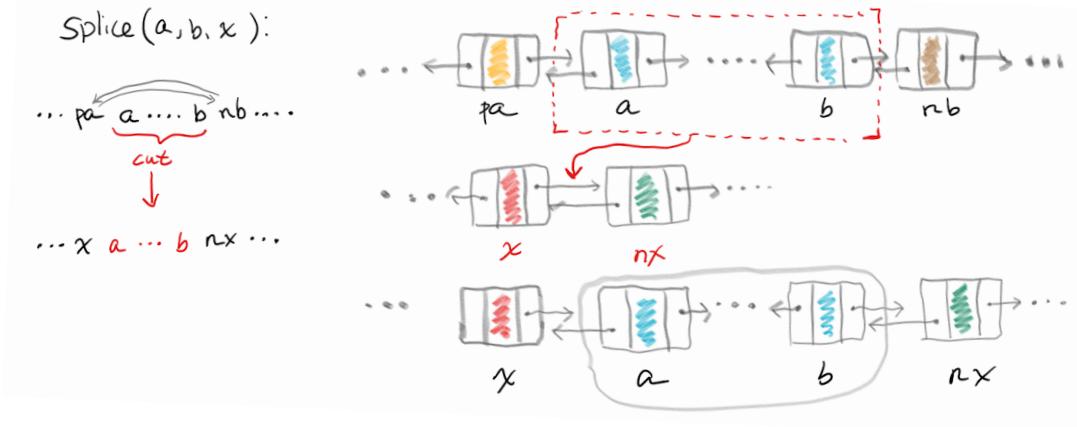
```
class DoublyLinkedList:
    def __init__(self):
        self.head = Node() # 빈 리스트는 dummy 노드만으로 표현
        self.size = 0 # 필요하다면

    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



```

def splice(self, a, b, x):
    if a == None or b == None or x == None:
        return

    ap = a.prev      # ap is previous node of a
    bn = b.next      # bn is next node of b

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

    # insert [a..b] after x
    xn = x.next
    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 `self`는 생략함 (`splice` 함수를 호출하여 가능)
- `moveAfter(a, x)`: 노드 `a`를 노드 `x` 뒤로 이동
 - `splice(a, a, x)`와 같음! (왜?)
 - `moveBefore(a, x)`: 노드 `a`를 노드 `x` 앞으로 이동
 - `splice(a, a, x.prev)`와 같음!
 - `insertAfter(x, key)`: 노드 `x` 뒤에 데이터가 `key`인 새 노드를 생성해 삽입
 - `moveAfter(Node(key), x)`와 같음!
 - `insertBefore(x, key)`: 노드 `x` 앞에 데이터가 `key`인 새 노드를 생성해 삽입
 - `moveBefore(Node(key), x)`와 같음!
 - `pushFront(key)`: 데이터가 `key`인 새 노드를 생성해 `head` 다음(front)에 삽입
 - `insertAfter(self.head, key)`와 같음!
 - `pushBack(key)`: 데이터가 `key`인 새 노드를 생성해 `head` 이전(back)에 삽입
 - `insertBefore(self.head, key)`와 같음!
- h. 삭제 연산: 매개변수 `self`는 생략함
- `remove(x)` 또는 `deleteNode(x)`: 노드 `x`를 제거


```
def remove(self, x):
    if x == None or x == self.head: # 조건 체크
        return
    x.prev.next, x.next.prev = x.next, x.prev
    # x를 떼어냄
```
 - `popFront()`: `head` 다음에 있는 노드의 데이터 값 리턴. 빈 리스트면 `None` 리턴


```
def popFront(self):
    if self.isEmpty(): return None
    key = head.next.key
    self.remove(head.next)
    return key
```
 - `popBack()`: `head` 이전에 있는 노드의 데이터 값 리턴. 빈 리스트면 `None` 리턴 : [?] 직접 해볼 것
 - i. 기타 연산: `join`, `split`
 - `join(another_list)`: `self` 뒤에 `another_list`를 연결함
 - `split(x)`: `self`를 노드 `x` 이전과 `x` 이후의 노드들로 구성된 두 개의 리스트로 분할
 - `self = self.head → ... → x.prev까지이고, new_list = x → x.next → ... → self.head.prev까지가 됨`
 - j. [?] 연산의 시간복잡도

moveAfter/Before		insertAfter/Before
------------------	--	--------------------

pushFront/Back		popFront/Back	
remove		search	

k. 배열에 같은 연산을 적용한 경우의 시간복잡도 (이중연결리스트와 비교)

moveAfter/Before		insertAfter/Before	
pushFront/Back		popFront/Back	
remove		search	

l. [?] 양방향 연결 리스트와 배열의 장단점은 무엇인가?

i. 장점:

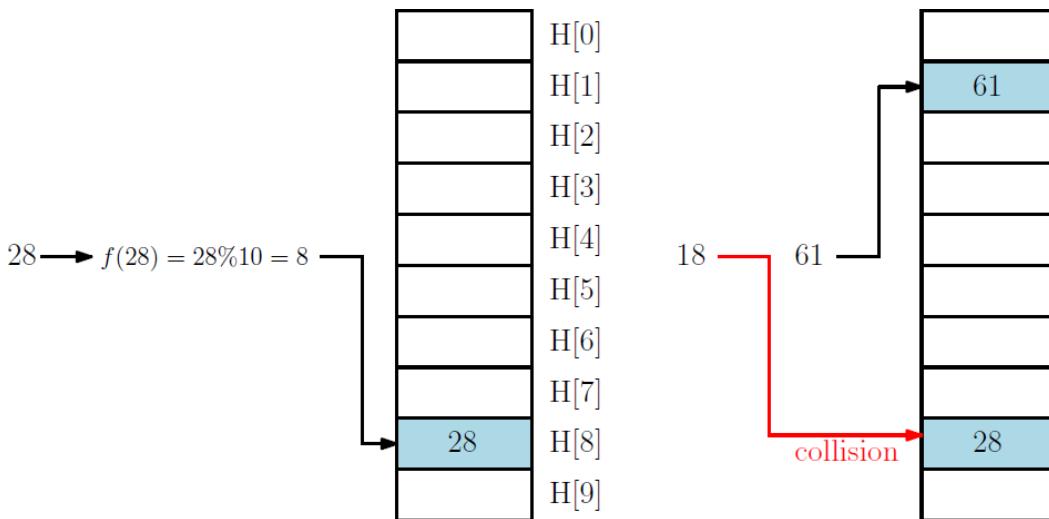
ii. 단점:

m. [?] [해보기]-easy-medium] 원형이중연결리스트를 이용해 Josephus 문제 구현하기

- i. Josephus 문제에 대해선 Queue 설명을 참고
- ii. 입력으로 n과 k를 받아 한명씩 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 남을때까지 반복
 - 남은 노드의 번호를 출력함

D: 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 overwrite함
 - b. `remove(key)` # (key, value)가 해시 테이블에 있다면 제거(delete)
 - c. `search(key)` # (key, value)를 찾아 리턴, 없다면 없다고 알려줌
2. 해시 테이블이 사용되는 분야는 매우 광범위한 실용성이 뛰어난 자료구조 중 하나
 - a. database, compilers & interpreters, document similarity, network router, substring search, file/directory synchronization, cryptography
3. 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. 그러면 수건을 찾고 싶다면, 수건이 저장된 서랍 번호를 먼저 (계산하여) 알아낸 후, 네 번째 서랍에 들어 있는 아이템들을 하나씩 비교해 수건을 찾아내면 된다
 - c. 가장 핵심적인 과정은 각 정보를 몇 번째 서랍에 넣을지를 결정하는 것이다.
4. 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 사용
 - b. 해시 함수는 `f(K) = K % 10`로 정의된다고 하자.
 - 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 출동해결방법(collision resolution method)이라 부른다



5. 해시 함수 (hash function)

- [?] key 값이 정수가 아니라면? 실수이거나 문자열이라면?
 - (실수, 문자열) key 값을 정수에 대응시키는 prehash 함수를 먼저 사용해 변환
 - Python: `hash(x)` 함수는 x 를 정수로 맵핑하는 prehash 함수임
 - `__hash__` 함수로 지정할 수 있음
- e, f, g번을 꼼꼼히 읽어보고, c번과 d번은 심화 내용으로 건너뛰어도 됨
 - 앞으로 mod라 함은 modulo 연산을 의미하고, % 연산자와 같다
- 완전(perfect)** 해시 함수: 충돌없이 1-to-1 맵핑하는 해시 함수
 - 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.
 - 따라서, 완전 해시 함수를 사용하는 건 비효율적임
- c-universal** 해시 함수: 서로 다른 임의의 두 key 값 x, y 에 대해 $\text{prob}(f(x) == f(y)) = c/\text{size}(H)$ 이 성립하는 해시 함수
 - 여기서 c 는 0보다 큰 실수 상수임
 - 비교적 골고루 맵핑하고 완전해시 함수보다 계산하기가 쉬워 현실적임
 - 예: $m = |H|$ 은 해시 테이블의 크기이고 소수라 가정, $w = \log m$
 - key 값 x 를 w 비트씩 분할해, $x = (x_1, x_2, \dots, x_k)$ 로 표현
 - 각 a_i 를 w 비트 정수라 하고, $a = (a_1, a_2, \dots, a_k)$ 로 정의
 - $f_a(x) = (a \cdot x \bmod p) \bmod m = ((a_1 x_1 + \dots + a_k x_k) \bmod p) \bmod m$
 - 랜덤 값 a 를 골라 해시 함수 $f_a(x)$ 를 정의하면, 이 함수가 1-universal 해시 함수가 됨!
- 현실에서 자주 쓰이는 해시 함수들 (1)
 - Division:** $f(k) = (k \bmod p) \bmod m$ (p : 소수)
 - key 값들의 성질이 잘 알려져 있지 않은 경우에 유용
 - Multiplication:** $f(k) = ((ak) \bmod 2^w) \gg (w-r)$

- 1. **a**: 랜덤 값, $w = \log k$, $r = \log m$
- iii. **Folding**: key 값의 digit를 나눠 연산하는 형식
 - 1. **shift folding**: 예: 계좌번호 $k = 1254-387-601 \rightarrow$ 두 digit씩 나눠 모두 더한 후 $\text{mod } m \rightarrow (12 + 54 + 38 + 76 + 01) \text{ mod } m$
 - 2. **boundary folding**: 여러 digit로 나눈 후, 더하는 데, 짹수번 조각은 거꾸로 해서 더함, 예: $(12 + 45 + 38 + 67 + 01) \text{ mod } m$
- iv. **Mid-Square**: key 값을 적당히 연산한 후, 그 결과의 중간 부분을 떼어내 주소로 이용
 - 1. 예: $m = 1000$ 이라면, $k = 3121$ 이라면, $3121^2 = 9740641$ 이 되고 중간에 3 digit를 떼어낸 406이 주소가 됨.
- v. **Extraction**: key 값의 각 파트마다 임의의 digit을 떼어내 연결해 계산
 - 1. 예: 계좌번호가 1254-387-601라면, 1254에서 12, 601에서 1을 떼어낸 후 서로 붙여 121을 만듬. 121이 주소가 됨
- f. 현실에서 자주 쓰이는 해시 함수들 (2): key 값이 문자열(string)일 때
 - i. **Additive hash**: $\text{key}[i]$ 의 단순 합
 - ii. **Rotating hash**: $<<$, $>>$ (비트 쉬프트) 연산과 \wedge (exclusive or) 연산을 반복
 - iii. **Universal hash**:
 - 1. Bernstein hash: `initial_value = 5381, a = 33`
 - 2. STLPort 4.6.2 hash: `initial_value = 0, a = 5`
 - 3. java.lang.String.hashCode(): `initial_value = 0, a = 31`
 - iv. **Pseudo 코드 예제**:

```

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

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

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

```
- g. 좋은 해시 함수란?
 - i. 되도록 빠르게 계산되어야 한다
 - ii. 충돌이 되도록 적어야 한다

6. 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.
- d. **Open addressing: linear probing**
 - i. 해시 테이블 H 의 slot에 값 하나만 저장할 수 있다고 가정하자.
 - ii. 아래 그림에서는 key 값 $A5, A2, A3$ 가 저장되고, 다음으로 $B5, A9, B2, B9$ 가 입력된다.
 - iii. 각 값이 저장되는 슬롯의 번호는 알파벳 다음의 숫자라고 가정하자.
 - iv. $A2, A3, A5$ 가 저장된 후, $B5$ 가 저장될 차례라고 하자. $B5$ 는 5번째에 저장되어야 하는데, 이미 $A5$ 가 저장되어 있다. 결국 다른 곳에 저장해야 한다.
 - v. open addressing 방법은 아래쪽으로 슬롯을 차례로 탐색하면서 가장 먼저 발견된 빈 슬롯에 저장하는 것이다.
 - vi. 이에 따라 $B5$ 는 $H[6]$ 에 저장된다.
 - vii. 다음의 $B2$ 에 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.
 - viii. $B9$ 에 대해서 $H[9]$ 가 선점되어 있으니 다음 슬롯을 점검한다. $H[9]$ 이 마지막 슬롯이므로 다음 슬롯은 한바퀴 돌아서 $H[0]$ 가 된다. 따라서 $B9$ 은 $H[0]$ 에 저장된다.

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

- ix. [?] $C2$ 가 입력되었다. Open addressing에 의해 충돌해결이 이루어진다면, $C2$ 는 어느 슬롯에 저장되나?
- x. $m = |H| =$ 해시 테이블의 슬롯의 개수
- xi. **삽입 연산: `set(key, value)`**
 1. 해시 테이블 H 의 각 슬롯에는 하나의 아이템(item)을 저장한다.

2. 아이템은 (key, value) 쌍으로 정의된다.
3. key는 아이템들끼리 구분하므로 아이템마다 서로 달라야 한다.
4. value는 해당 아이템의 다양한 정보를 의미한다.

`set(key, value)`

5. key 값을 갖는 아이템이 이미 테이블에 있다면, 해당 아이템의 value를 매개변수 value 값으로 수정하고, 없다면 새 아이템 (key, value)를 삽입하는 연산
6. 정상적으로 수정 또는 삽입이 이루어졌다면, key값을 그대로 리턴하고, 테이블에 빈 슬롯이 없어 삽입을 하지 못했다면 FULL리턴
7. 이를 위해, open addressing 방법에 따라 key 값을 갖는 아이템을 찾거나 빈 슬롯을 찾아 H의 인덱스를 리턴하는 `find_slot(key)` 함수 필요

`find_slot(key)`

8. key 값을 갖는 아이템을 찾아 슬롯 번호(index)를 리턴한다. 단, 해당 슬롯에 아이템이 있다면 key 값을 갖는 아이템이며, 슬롯이 비어 있다면 해당 아이템이 해시테이블에 없다는 뜻이다.
9. 만약 key 값을 갖는 슬롯이 존재하지도 않고, 빈 슬롯도 없다면 FULL을 리턴한다.

Pseudo 코드: (from https://en.wikipedia.org/wiki/Open_addressing)

```
def set(key, value=None):
    i = find_slot(key)
    if i == FULL:      # 빈 슬롯이 없는 경우의 대책은?
                      # 더 큰 테이블이 필요하다!
        return None
    if H[i] is occupied: # key값이 존재하면 기존 값 수정
        H[i].value = value # value 값 update 후 리턴
    else: # H[i]가 비어있는 경우, 즉 key가 없다면 새로 저장
        H[i].key = key
        H[i].value = value
    return key
```

```
def find_slot(key):
    i = f(key)
    start = i
    while ( H[i] is occupied ) and ( H[i].key != key ):
        i = (i + 1) % m
        if i == start: return FULL
    return i
```

xii. 삭제 연산: `remove(key)`

1. `key` 값을 갖는 아이템을 `find_slot`을 이용해 찾는다. `i = find_slot(key)`라 하자
2. `H[i]`가 비었다면 삭제할 아이템이 실제로 존재하지 않는 경우이므로 `None` 리턴
3. `H[i]`가 존재한다면, 이 아이템때문에 아래쪽으로 밀려서 저장된 아이템들을 연쇄적으로 위로 옮려 이동한 후, 성공적인 삭제가 수행되었다는 의미에서 `key` 값 자체를 리턴
 - 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 `k`라 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?**
 - c. 또한 해시테이블이 원형 리스트와 같기 때문에 `i > j`일 수도 있으므로, `...j..i..k..` 순서라거나, `..k..j..i..`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 반복한다

4. 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)
            # 아래 세 가지 경우의 k인 경우 이동
            # | i..k..j |
            # |...j...i...k..| or |..k..j..i..|
            if not (i < k <= j or j < i < k or
                    k <= j < i):
                break # [?] if 조건문의 의미?
            H[i] = H[j]      # H[j]를 H[i]로 이동
            i = j

```

5. 연쇄적인 이동을 하지 않고 제거하는 방법은 없을까?

- H[i]를 지워야 한다고 하면, 이 슬롯을 unoccupied로 표시하지 않고, 다른 표시를 해두면 어떨까?
- 예를 들어, H[i].key = DUMMY 처럼 일종의 DUMMY 객체를 만들어 '삭제했음'을 표시해 보자
- 이렇게 하면 find_slot 연산에서 H[i].key 값이 DUMMY이더라도 탐색을 (멈추지 않고) 계속해야 한다. find_slot는 원하는 key 값을 찾거나 빈 (unoccupied) 슬롯을 찾을 때까지 계속하면 된다
- 이런 방법을 사용하면, remove 연산은 상수시간에 가능하지만, find_slot의 시간은 DUMMY로 표시된 슬롯도 계속 따라가야 하므로 더 오래 걸린다. (공짜 점심은 없다!)
- DUMMY는 다음과 같은 클래스의 객체로 처리 가능

```

class DummyValueClass():
    pass
DUMMY = DummyValueClass()

```

- 참조

https://just-taking-a-ride.com/inside_python_dict/chapter2.html

xiii. 탐색 연산: **search(key)**

1. key 값을 갖는 아이템을 찾아 value 값을 리턴하고, 없다면 None을 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. HashOpenAddr 클래스 구현

- i. size = 해시테이블의 슬롯 개수
- ii. keys = 슬롯의 키(key)를 저장하는 리스트
- iii. values = 슬롯의 값(value)을 저장하는 리스트 (optional)
- iv. 위의 set, find_slot, remove, search는 클래스 구조에 맞게 수정
 - 1. H[i].key → self.keys[i], H[i].value → self.values[i]
 - 2. 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 means "unoccupied"
        self.values = [None]*self.size

    def set(self, key, value):
        ...
    def find_slot(self, key):
        ...
    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
        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. 이 연산의 수행시간은 해시 함수의 성능에 좌우된다
- ii. 수행시간 분석을 쉽게 하기 위해, 임의의 key 값이 특정 슬롯으로 해시될 확률이 모두 $1/m$ 으로 같다고 가정한다.
- iii. 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
- iv. * 테이블 크기 조정은 후반부에 자세히 설명한다 *
- v. search 함수의 성능이 다른 두 함수 (set, remove)의 성능을 결정하기 때문에, search(key)의 수행시간만을 분석한다
- vi. search 시간을 좌우하는 건 key 값이 속한 cluster의 크기에 좌우된다
 1. 클러스터(cluster)란? 아래 그림에서처럼 해시 테이블에 아이템들이 계속 삽입되면서 국지적으로 모여있게 된다. 처음엔 (A2, A3), (A5) 두 클러스터, 두 번째 테이블 그림에선 (A2, A3), (A5, B5), (A9) 세 클러스터가 만들어진다

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

vii. 특정 슬롯 i 에서 시작하는 길이 k 인 cluster:

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

1. 이 cluster가 발생할 확률 p 를 계산해보자!
2. 이 cluster에 속한 key 값들은 $i, i+1, \dots, i+k-1$ 슬롯으로 해시되어야 한다. 특정 슬롯으로 해시 될 확률은 가정에 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, 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:

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

3. 위의 식은 $m = 2n$ 인 경우가 가장 최대가 됨을 알 수 있다. 따라서,

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

4. 위의 식에서, $r!$ 는 대략 $(r/e)^r$ 으로 근사되고, $c < 0.8243$ 인 상수임이 알려져 있다

5. 결국 k 가 커질수록 길이가 k 인 cluster가 발생할 확률이 지수적으로 **매우 작아진다 (매우 중요한 사실)**

viii. 이제, **search(key)** 함수의 **평균 수행시간**을 계산해보자

1. 만약, $i = f(key)$ 라면, i 를 포함하는 cluster를 탐색하게 되고, 최악의 경우 cluster의 길이 k 만큼의 시간이 필요하다
 - a. 당연히 필요한 시간은 $O(1 + k)$
2. 평균이므로 모든 i 에 대해, 모든 길이 k 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 $k^2 c^k$ 는 충분히 큰 상수 k 에 대해선 1보다 작게 되어 상수로 수렴한다

3. 결국 **search** 함수 (**set**, **remove** 역시) **O(1)** 시간에 수행된다
4. $m \geq 2n$ 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
5. 실험에 의하면, $m \geq 1.25n$ (즉, 25% 이상은 빈 슬롯을 유지하는 게 가장 현실적이고 효율적인 기준이라고 밝혀졌다. 실제 Python의 dict와 set을 위한 해시 테이블에서도 이 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
 1. $f(key) + g(key) \rightarrow f(key) + 2g(key) \rightarrow \dots \rightarrow f(key) + kg(key) \rightarrow \dots$ 순서로 슬롯을 검색

- iii. linear probing과 이 두 방법의 구체적인 성능 비교는 후반부에 그림으로 자세히 설명

h. Chaining

- i. H의 slot에 값 하나만 저장하도록 하는 게 아니라, 각 slot마다 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)]
```

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

2. find_slot 은 key 값에 대한 해시 함수 값(슬롯 인덱스)을 단순 리턴

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

3. 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)
```

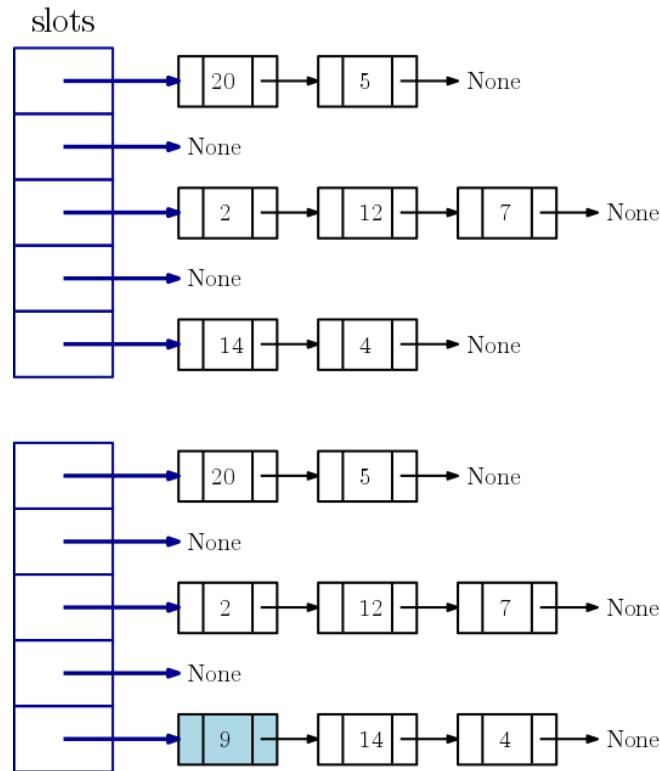
```
    else: # 기존의 key값을 있으므로 value값 수정연산
        v.value = value
```

4. search(key): H[hash_function(key)] 리스트를 탐색하여 있다면, value 값을 없다면 None을 리턴한다

5. 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) # 효율적인 코드 아님
```

- iii. 아래 그림에서는 해시 테이블의 크기 `size = 5`인 경우, `hash_function(key) = key % size`로 정의했을 때, `set(9, value)`를 실행한 경우의 변화를 나타낸다



- iv. [복습] `__setitem__(self, key, value)`와 `__getitem__(self, key)` 함수를 각각 구현해보자

- v. [고급] Chaining 연산 수행시간: search 시간 분석
1. c-universal hash 함수를 가정한다. (즉, 다른 두 key 값의 해시 값이 같은 확률이 c/m 이하이다.)
 2. 특정 key 값을 search하는 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!
 3. $C(x, y) =$ 두 key의 hash 값이 같으면 1, 아니면 0을 나타내는 랜덤변수 (random variable)
 4. $\text{Prob}(C(x, y) = 1) = c/m$ (c-universal hash 함수이므로)
 5. $\text{len}(x) =$ (x 와 같은 해시 값을 갖는 y 의 개수) 이므로

$$\text{len}(x) = \sum_y \text{Prob}(C(x, y) = 1)$$
 6. 정리하면,

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

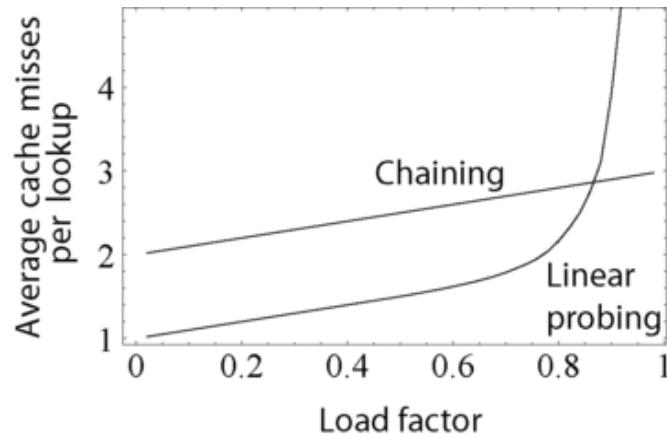
7. 결국 평균 $O(1 + n/m)$ 시간이 되고 $m \geq 2n$ 정도로 운영하면, $O(1)$ 시간에 세가지 연산을 모두 수행 가능하다!
-

i. 해시 자료구조의 성능은 어떻게 평가할까?

- i. 앞에서 가정한대로, 해시테이블의 크기(슬롯의 갯수)를 m 이라 하고, 테이블에 저장된 아이템의 갯수를 n 이라 하자
- ii. **load factor:** $LF = n/m$
 1. 테이블 크기에 비해 아이템이 많을 수록 load factor가 커진다. 그러면 충돌 횟수도 비례하여 증가하게 됨
- iii. **collision ratio:** $(\text{number of collisions})/n$
 1. 비율이 작을 수록 연산 시간이 작아짐
- iv. **search 성능**
 1. unsuccessful search는 찾고자 하는 key 값이 테이블에 없는 경우이고, 이 경우가 successful search보다 시간이 많이 걸림
 2. 결국 unsuccessful search의 수행시간이 search의 수행시간이 됨 (최악의 경우 가정)
- v. 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
 1. 예를 들어, linear probing인 경우, LF가 클수록 $1-LF$ 가 작아지고, 따라서 전체 값은 커지게 되어 탐색을 위한 평균 비교 횟수가 커짐.
(unsuccessful인 경우는 $(1-LF)^2$ 이므로 더 빠른 속도로 탐색시간이 증가함)
 2. 반면에, quadratic probing 방법은 \log 식에 따라 커지게 되어 linear probing보다는 탐색 횟수가 훨씬 작음
 3. 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.



7. 테이블 크기 조정: $n >> m$ 처럼 너무 많은 아이템이 해시 테이블에 저장된다면, 현재의 해시 테이블이 너무 작다. 그러면 연산시간이 커진다. 어떻게 해야 할까?
- a. 더 큰 해시 테이블을 할당받아 ($m \rightarrow m'$), 기존의 해시 테이블의 값을 모두 옮긴다
 - i. n 개의 아이템을 새 테이블로 옮겨야 하므로 $O(n)$ 시간 필요 → 꽤 큰 비용아닌가?
 - b. 이슈
 - i. 언제, 얼마나 크게 할당받아야 하나?
 - 1. $n = m$ 이 될 때, $m' = m + 1$ 로 크기 조정 → 계속 삽입이되면 크기 조정과 테이블 이동을 해야하는 단점
 - 2. [?] $n \geq m/2$ 이 될 때, $m' = 2m$ 로 테이블 크기를 2배 늘리는 게 일반적인 방법!
 - ii. [?] 새로운 해시 함수가 필요한가?
 - iii. 만약, n 번의 삽입이 연속적으로 일어나면서 테이블 크기 조정이 여러 번 발생하는 최악의 상황을 생각해보자. 이를 위해 필요한 총 시간은?
 - 1. $m = 2 \rightarrow 2^2 \rightarrow 2^3 \rightarrow \dots \rightarrow 2^k$

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

2. k번의 크기 조정이 일어난다. 그 때마다 비용 c는 다음과 같다

$$c = 1 \rightarrow 2^1 \rightarrow 2^2 \rightarrow \dots \rightarrow 2^{k-1}$$

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

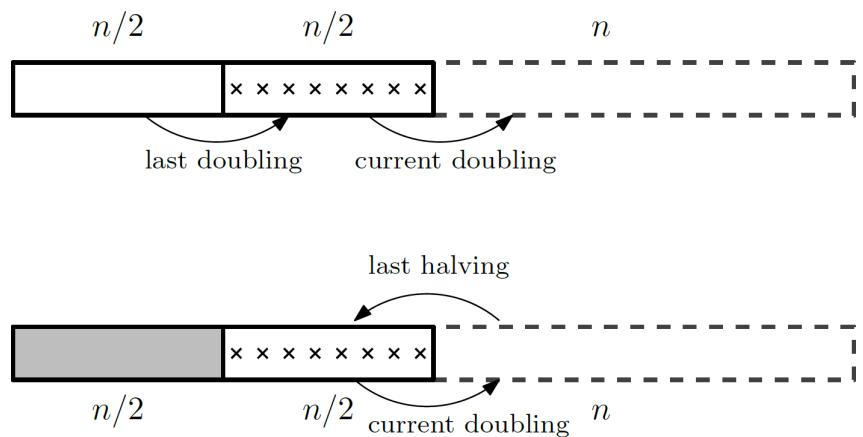
4. [!] $n = 2^{k-1}$ 이므로 평균 비용은 $(2^k - 1)/2^{k-1} < 2$ 이 된다. 즉, 평균적으로 따지면 한 번의 삽입 연산의 평균 비용이 O(1)이라는 의미이다!

- iv. 여기에서는 한 연산의 최악의 경우의 수행시간보다 여러 번의 연산에 대한 평균 수행시간을 정의하는 게 더 합리적이다. 이렇게 연산의 평균 시간을 분석하는 것을 **amortized analysis**라 하고, 평균시간을 **amortized time**이라 부른다

8. [고급] 더 자세한 Amortized 시간 복잡도 분석 방법: **Charging 방법** 소개

- a. 다음과 같은 해시 테이블의 resize 연산을 고려해보자: $n = m$ 이 되면 **doubling**이 발생하고, $n < m/4$ 가 되면 **halving**이 발생하는 해시 테이블 모델을 가정함
 - i. set 연산이 계속 이루어져 $n > m$ 이 되면, 더 이상 빈 slot이 없게 된다. 이 경우에 보통 **doubling**이란 방법을 통해 $m \rightarrow 2m$ 으로 slot의 개수를 두 배 늘린 새로운 H' 을 만들고 기존의 H 에 있는 모든 item을 새 해시 함수를 이용해 H' 으로 이동해야 한다
 - ii. 반대로 delete 연산이 훨씬 많이 발생해 H 에 저장된 item 개수가 $n < m/4$ 로 줄어든다면, $m \rightarrow m/2$ 로 반으로 줄이는 **halving** 과정을 수행하게 된다
 - b. doubling, halving 모두 기존의 item을 새로운 테이블로 옮겨야 하기 때문에 $O(n)$ 시간이 필요한 매우 비싼 연산이다. 현재 set이 마지막 빈 슬롯에 삽입된다면 ($n = m$ 이 되는 경우) doubling이 발생하여 $O(n)$ 의 시간이 필요하다. 결국, 이 set 연산의 최악의 경우의 수행시간은 $O(1)$ 이 아니라 $O(n)$ 이다. halving이 발생하는 경우의 delete 연산도 $O(n)$ 시간이 걸리게 된다
 - c. 해시 테이블 자료구조를 사용할 때에는 빈번한 set과 delete 연산을 처리하는 것이 일반적이다. 매 연산마다 doubling/halving이 발생하는 게 아니므로, 최악의 경우에 대한 연산 시간이 아니라 전체 연산 횟수에 대한 평균 시간을 분석하는 게 합리적이다. 이처럼 전체 연산 횟수에 대한 개별 연산의 평균 시간을 **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 식"
 - i. **set (insertion) 연산**
 - 1. $n/2 \rightarrow (\text{직전 doubling}) \rightarrow n/2+n/2 \rightarrow (\text{현재 doubling}) \rightarrow n+n$

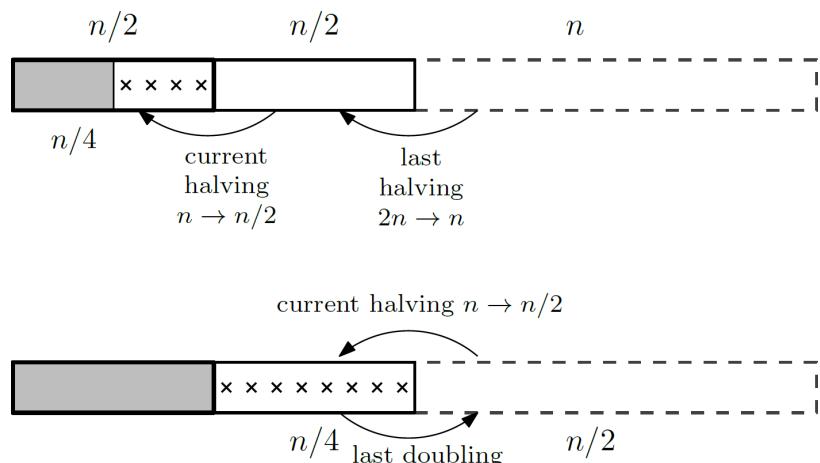
2. doubling의 비용을 바로 전 doubling 이후에 발생한 set 연산들에 떠 넘겨보자!
3. $n \rightarrow 2n$ doubling의 비용은 n 개의 item을 새로운 테이블로 옮기는 비용이므로 cn 시간이 필요하다고 볼 수 있다. 이 cn 비용을 바로 직전 doubling ($n/2 \rightarrow n$: $n/2$ 개는 full, 나머지 $n/2$ 개는 empty)이므로 $n/2$ 개가 모두 채워져야 현재 doubling 발생!) 또는 halving ($2n \rightarrow n$: $n/2$ 개는 full 나머지 $n/2$ 개는 empty, $n/2$ 개가 채워져야 현재 doubling 발생!) 이후에 발생한 $n/2$ 개의 set 연산에 전가하는 것이다. 비용 cn 을 $n/2$ 개의 set 연산에 전가하므로 하나의 set 연산에 대해선 $2c = \mathbf{O(1)}$ 정도만 감당하면 된다!



4. 여기서 중요한 사실은 이 set 연산은 오직 $n \rightarrow 2n$ doubling 연산에 대한 비용만 감당하면 된다. (다른 set 연산의 비용을 감당하지 않는다는 사실이 매우 중요! 왜?)
5. 결국, 하나의 set 연산은 doubling에서 전가된 비용을 포함해 $O(1)$ 의 시간 비용만 지불하면 된다: **set 연산 = 평균(amortized) $O(1)$ 시간**

ii. delete (remove) 연산:

1. halving은 $n \rightarrow (halving \text{ when } H \text{ has } n/4 \text{ items}) \rightarrow n/2$ 에서 발생한다.
2. 바로 직전 doubling 또는 halving 작업을 생각해보자.



3. 바로 직전에 halving이었다면, $2n \rightarrow n$ halving이므로, $n \rightarrow n/2$ halving이 발생하려면 $n/4$ 번의 deletion이 반드시 있어야 한다. 따라서 이 deletion에 현재 halving의 비용을 전가한다
4. 바로 직전에 doubling이었다면, $n/2 \rightarrow n$ doubling이므로 역시 $n/4$ 번의 deletion이 있어야 현재 halving이 발생한다. 따라서 $n/4$ 개의 deletion 연산에 현재 halving 비용을 전가한다
5. 결과적으로 $O(n)$ 의 비용이 드는 halving 작업을 직전의 $n/4$ 번의 deletion 연산에 전가할 수 있고, 이 deletion 연산은 다른 연산으로부터 전가받은 비용은 존재하지 않으므로 결국 평균적으로 각 deletion 연산이 $O(1)$ 씩의 비용을 나눠서 지불하는 셈이다
6. 결론: doubling/halving이 없는 set(insertion)과 deletion은 자체는 $O(1)$ 이고, doubling/halving이 필요한 set/deletion은 비용을 직전 set/deletion에 $O(1)$ 씩 분담할 수 있으므로 다음이 성립한다

set/deletion under table resize = $O(1)$ amortized 수행시간

9. [고급: Python 활용 예] Python의 `dict`는 해시 테이블로 관리되는 매우 효율적인 자료구조로 해시 테이블과 지원하는 연산이 같음
 - 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 slot으로 시작, 테이블의 슬롯이 $\frac{m}{3}$ 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
 - c. 해시 함수
 - i. 해시 테이블 사이즈가 $m = 2^k$ 이라면, 우선 key 값의 하위 k 비트를 그대로 기본 해시 함수값으로 사용한다 (예: $m = 2^4$, $f(0) = 0$, $f(15) = 15$, $f(16) = 0$)
 - ii. key 값이 k 비트보다 크다면, 상위 비트가 함수 값에 반영이 안될 수 있지만, collision resolution에서 반영되도록 설계되어 있다
 - 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 아님!)
- ii. **질문:** 이렇게 위치를 검사하면 테이블의 모든 슬롯을 검사할 수 있을까?
 - 1. 예: $m = 2^3$, $i = 0$ 이라면 아래 순서대로 슬롯을 검사하므로 빈 슬롯이 존재하면 결국 찾게 된다!

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

- 2. 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
i = i % 2k # as the next slot index to be checked
```

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

- iv. 실험 결과 매우 효율적이고 빠르다고 알려짐!

- 1. <https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c#l33>
- 2. https://just-taking-a-ride.com/inside_python_dict/chapter4.html

10. 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 매핑됨에 주의하자. 따라서 이 hash 함수를 그대로 사용하지 말고 목적에 맞게 수정해서 사용하는 게 좋다
- c. 사용자가 정의한 class를 hashable하도록 하고 싶다면, class의 특별 메쏘드로 __hash__ 함수를 구현하면 된다

11. [☞ 인터뷰 문제1] **Uniqueness problem**: n개의 수가 입력으로 주어질 때, 이 수들이 모두 다른 수인지 판별하는 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!

12. [☞ 인터뷰 문제 2 - pair sum] n개의 정수 값이 저장된 리스트 A와 정수 값 K가 입력으로 주어진다. 더해 K가 되는 A의 두 수를 찾아라

- a. 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
- b. 더 빠르게 할 수 없을까?
- c. [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
- d. 만약, $A[i] + A[j] < K$ 인 경우엔 i를 늘려야 할까, j를 줄여야 할까? $A[i] + A[j] > K$ 인 경우에는?
- e. 이 알고리즘의 수행시간은?
- f. 이 보다 더 빠르게 가능할까?
 - i. 힌트: 삽입과 검색이 평균 $O(1)$ 시간에 가능한 해시 테이블 사용?

13. [☞ 인터뷰 문제 3 - swap sum] 두 리스트 A와 B에 각각 n개와 m개의 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?

- a. 역시, 가장 단순한 방법은 모든 쌍을 고려해 교환해보는 방법이지만, 더 빠른 방법이 존재한다.
- b. 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. $c = \text{sum}(A) - \text{sum}(B)$ 라고 하면, c는 상수이다. 즉, $(a-b)$ 의 2배가 상수 c인 쌍 (a, b)를 찾으면 된다!
 - i. 힌트: 바로 앞의 pair sum 문제와 유사하지 않나?

- d. 알고리즘을 완성해보자. 이 알고리즘의 수행시간은?
14. [☞ 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
- [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
 - [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
 - [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
 - [가장 빠른 방법] 평균이 아닌 최악의 경우에 $O(n)$ 시간에 가능 (어떻게?)

15. [☞ 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: $S = AABABBAAABABBAAB$

$a = 1223334566777899$ # $a[i] = S[0] \dots S[i]$ 의 A의 개수

$b = 0011233334456667$ # $b[i] = S[0] \dots S[i]$ 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? $S[i] \dots S[j-1]$ 의 두 문자 개수는?

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

iv. 이 인덱스 쌍은 어떻게 구해야 할까?

1. 해시 테이블을 이용하는 방법?

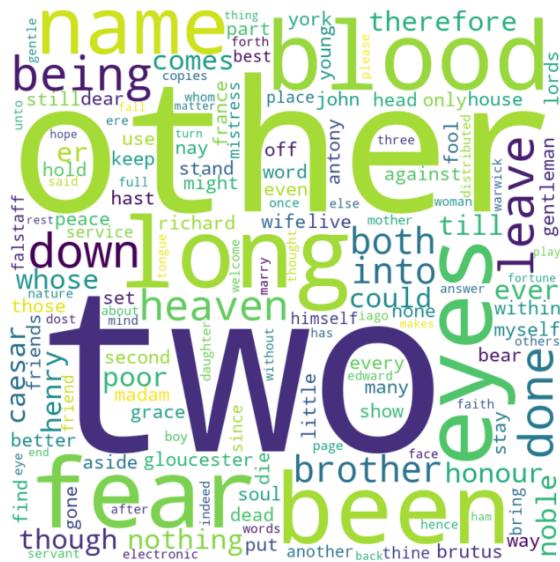
2. 제 3의 리스트 c를 이용하는 방법? [힌트: d의 값의 범위가 제한적이라는 사실을 이용해보자]

16. [복습] [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 `H0, HC`를 준비한다:
 - i. `m = the size of hash table = 10,000`
 - ii. `H0` 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 LF가 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 값을 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. matplotlib는 python 교재와 무료 설명 동영상 참고
 - ii. 라이브 데이터 그래프 그리기 동영상 추천: <https://youtu.be/Ercd-lp5PfQ>

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

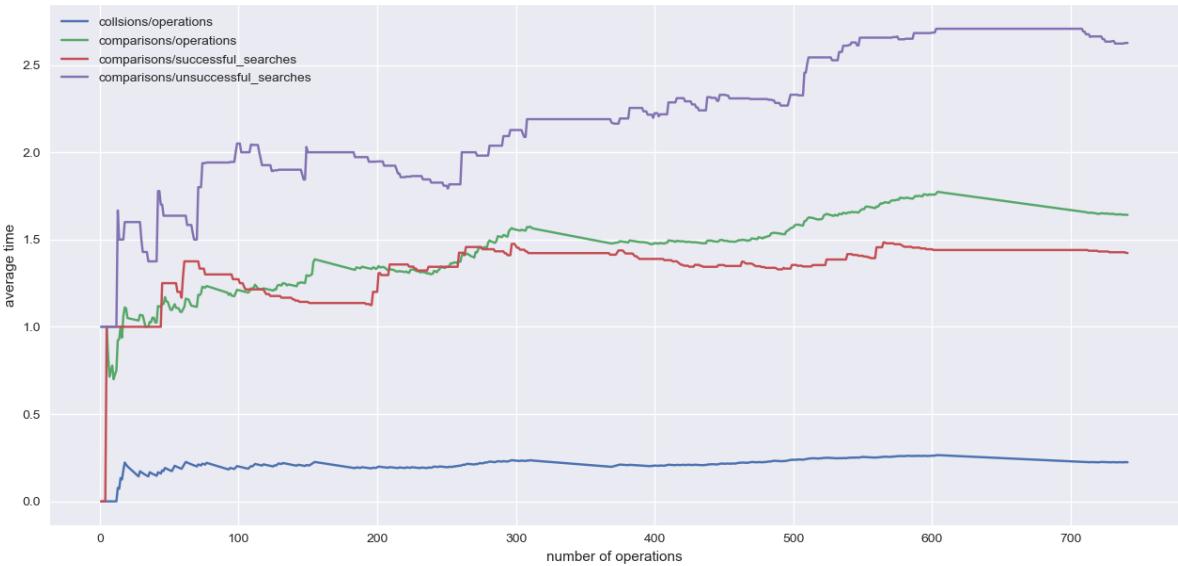
- a. 셰익스피어의 희곡이나 산문은 인터넷에서 text 파일 형식으로 쉽게 구할 수 있다. 이 파일을 읽어, 각 단어들이 몇 번 등장하는지 횟수를 세는 문제를 풀어보자
 - b. 먼저 줄 단위로 읽어 영단어로 split해야 한다. 이를 위해 regular expression 모듈의.findall 함수를 이용한다 (`import re` 필요)
 - i. `words = re.findall('\w+', line)` # line의에 포함된 모든 영어 단어를 찾아 리스트로 만들어 리턴함
 - c. 해당 단어의 등장 횟수를 기록하기 위한 방법은 다양하지만, 여기서는 hash table 자료구조를 사용해보자. 어떤 단어가 해시 테이블 H에 있다면, 해당 단어의 value (counter 역할) 값을 하나 증가시키고, 없다면 H에 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
 - f. 해시 테이블에 있는 단어의 횟수를 내림차순으로 정렬하면, 첫 번째 단어가 가장 많이 등장하는 단어이다. 이 중 원하는 랭킹에 속하는 단어들을 word cloud로 가시화한다 (아래 예는 빈도수 랭킹이 150위부터 300위까지를 파이썬의 wordcloud 모듈과 matplotlib를 이용해 가시화한 것이다)



- g. wordcloud 모듈 설치 필요: `pip install wordcloud`
 - i. 사용법은 https://lovit.github.io/nlp/2018/04/17/word_cloud/ 사이트에 잘 정리되어 있으니 참조 바람 (한글 wordcloud도 가능)

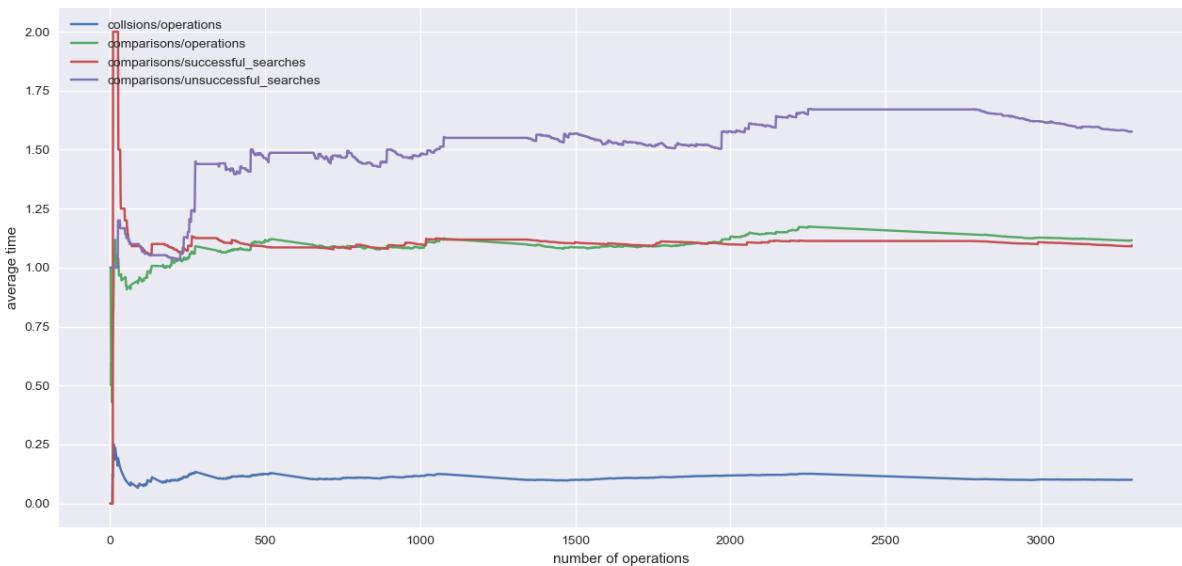
18. [LF에 따른 해시 테이블 성능 실험] 해시 테이블 크기 $m = 8$ 로 시작해서

- 해시 함수: $f(k) = k \% m$ 사용
- open addressing - linear probing으로 충돌 해결
- set, remove, search을 랜덤하게 호출해, 비교 횟수, 충돌 횟수 등을 측정함
- LF 값을 $LF = 0.5, 0.8$ 로 달리하면서 측정함



y-축은 연산 하나당 평균 비교 또는 충돌 횟수

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

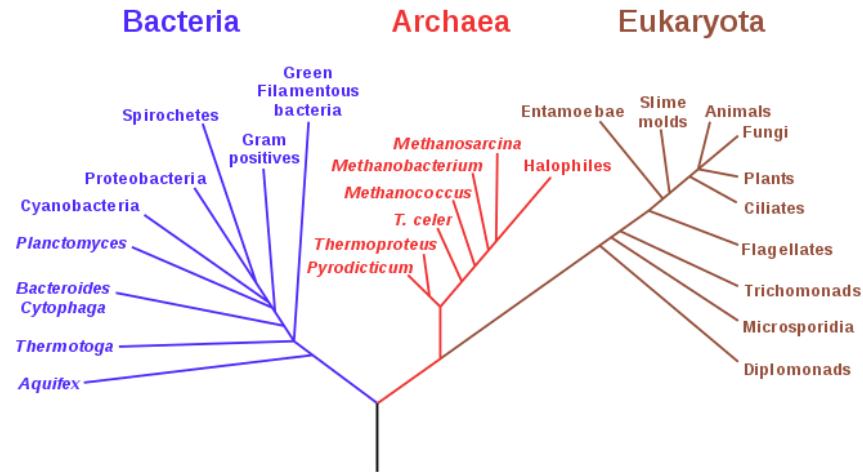


첫 슬롯의 개수를 $m=8$ 개부터 시작해 $LF \geq 0.5$ 이면 resize하도록 지정

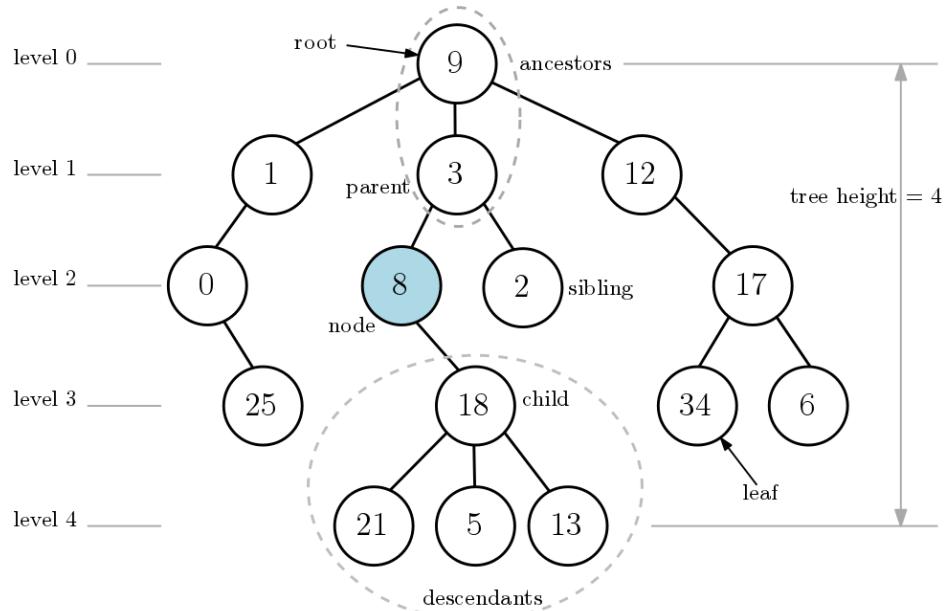
E: Tree (+ heap)

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

- a. 예 1: 유전 트리 (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 표현한다는 것



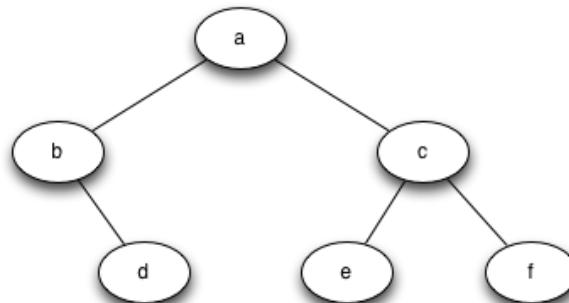
- c. 용어정리

- 부모노드/자식노드: 3은 8, 2의 부모노드, 8, 2는 3의 자식 노드
- 조상노드/자손노드: 9는 8의 조상노드, 8은 9의 자손노드
- 루트(root) 노드: 모든 노드의 조상 노드: 9
- 리프(leaf) 노드: 자식이 없는 노드
- 레벨(level): 루트 노드의 레벨 0를 시작으로 한 세대씩 내려가면서 1씩 증가
- 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)

2. Binary tree(이진트리)와 Heap (힙)

- 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



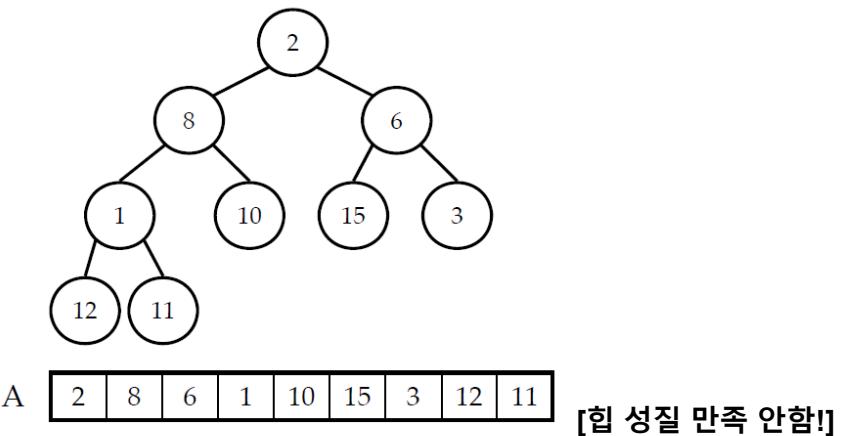
- c. 표현법 1: 하나의 리스트만으로 표현


```
[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

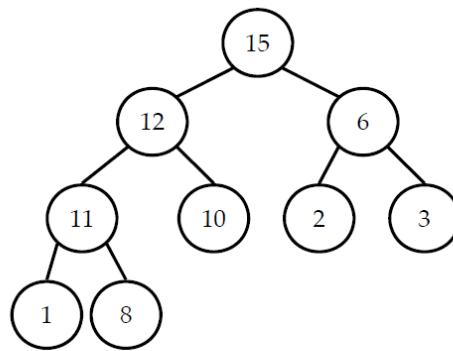

```
[a, [b, [], [d, [], []]], [c, [e, [], []], [f, [], []]]]
```

 - 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[]에, 오른쪽 자식노드는 A[]에 저장된다
- iv. [?] A[k]의 부모 노드는 A[]에 저장된다
- v. [?] 이 표현법의 문제는 빈 칸이 많을 경우 메모리 낭비가 심하는 것이다. 이런 일이 발생하는 극단적인 이진트리 모양은 어떤걸까?
- vi. 빈 칸을 만들지 않기 위해선, 레벨마다 노드를 빼지 않고 가득 채우고, 마지막 레벨엔 왼쪽부터 노드를 채워나간 트리모양이면 된다 (아래 그림 참조) 다른 의미로는 리스트에 차례대로 저장된 값을 이진트리로 해석하면 레벨마다 가득 찬 모양이 된다는 의미다



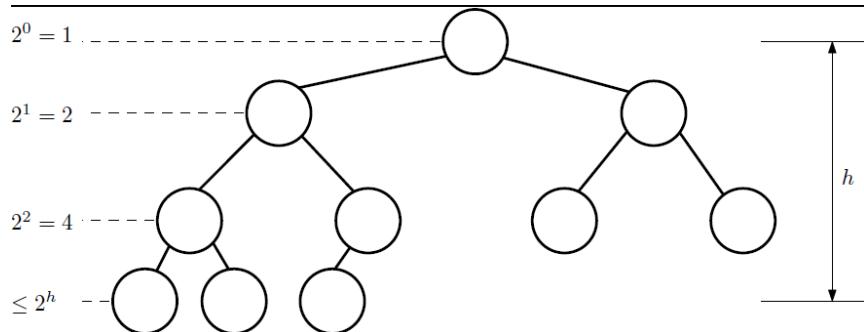
- g. 힙(heap): 다음의 모양과 힙 성질을 만족하는 리스트에 저장된 값의 시퀀스
 - i. (모양 성질) 다음과 같은 이진트리여야 한다:
 - 마지막 레벨을 제외한 각 레벨의 노드는 모두 채워져 있어야 한다
 - 마지막 레벨에선 노드들이 왼쪽부터 채워져야 한다
 - ii. (힙 성질) 루트 노드를 제외한 모든 노드에 저장된 값(key)은 자신의 부모 노드의 값보다 크면 안된다
 - 위의 그림의 이진트리는 모양성질은 만족하지만 힙 성질은 만족하지 않는다. (8의 부모가 2가 되어 성질 위배)
 - 반면에 아래 그림은 모양과 힙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



A [15 | 12 | 6 | 11 | 10 | 2 | 3 | 1 | 8]

[힙 성질을 만족함!]

- 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. $h = O(\log n)$ 이다

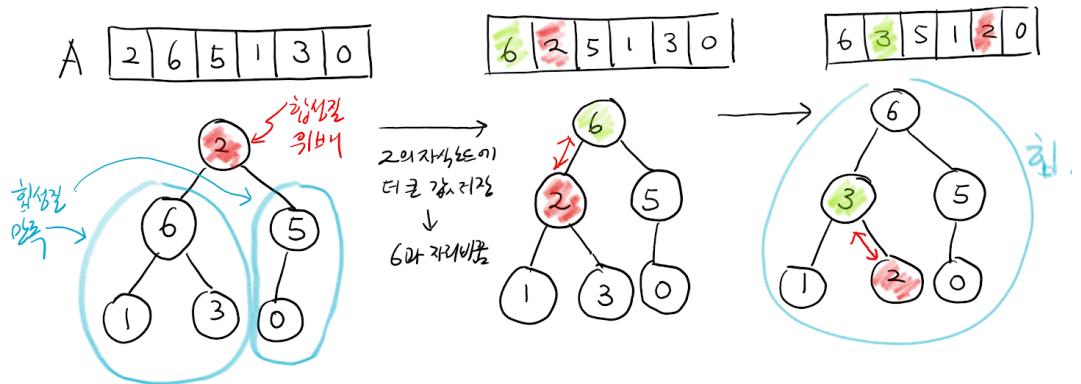


- vi. Heap 클래스:


```
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. 위의 두 번째 경우가 그런 예이다

- viii. 힙 성질을 만족하지 않으면, 값들을 재배열해서 힙을 만들 수 있다 → `make_heap` 함수 → 이를 위해 `heapify_down` 함수가 필요
- ix. `heapify_down(k)` 함수:
- $A[k]$ 의 자손 노드들은 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 어떤 값?
        if self.A[L] > self.A[k]:
            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_down`의 수행시간을 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 자리바꿈을 하는 것이므로 $O(1)$ 시간이면 충분하다.
 - 따라서, 최악의 경우에 힙의 높이에 비례하는 시간이 필요하다. 즉, $O(\log n)$ 시간이 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?

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

- `range(n-1, -1, -1) → range(n//2, -1, -1)`로 변경해도 문제 없음
(왜 그럴까? [?])
- `make_heap`의 수행시간을 분석해보자
 - a. **분석 1:** `for` 반복문은 n번 반복되고, 반복할 때마다 `heapify_down`이 한번씩 호출된다. `heapify_down`의 수행시간이 $O(\log n)$ 이므로 $O(n \log n)$ 시간이면 충분하다
 - b. **[고급] 분석 2:** 엄밀하게 분석하면 $O(n)$ 시간이면 충분하다 (아래 분석 참조. 건너 뛰어도 됨)
 - 레벨 i에 있는 노드 수는 (최대) _____ 이다
 - 레벨 i에 있는 노드가 `heapify_down`에서 움직이는 레벨 수는 (최대) _____ 이다.
 - 결국 레벨 i에 있는 노드들이 움직인 총 횟수는 = 레벨 i의 노드 수 x 레벨 i의 노드가 움직이는 레벨 수 = $\sum_{i=0}^h 2^i \times (h-i)$
 - 모든 레벨 i = 0, 1, ..., h에 대해 횟수를 더한 값이 `make_heap`을 위해 이동한 총 횟수 s이다

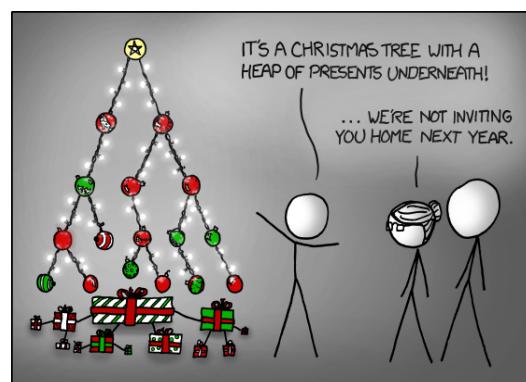
- 이 식을 전개해보자! (어떻게?) [hint: 역급수]

- xi. `heap_sort()`: 힙이 아니라면, 우선 `make_heap`으로 힙을 만든 후, `heapify_down` 함수를 반복적용하여 값들을 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_sort`의 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 힙이 됨
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 값을 힙 성질이 만족하도록 위치를 재조정해야 한다
- 이 경우엔 x가 힙의 리프에 위치하므로, 루트 노드 방향으로 올라가면서 자신의 위치를 조정하면 된다 → `heapify_up` 함수 구현

```
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 → $O(\log n)$

xiii. 삭제 연산 `delete_max()`: 임의의 값을 삭제하는 것이 아닌 루트 노드에 저장된 가장 큰 값을 삭제후 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 $O(\log n)$ 시간에 수행

xiv. 연산 및 수행시간 정리 (n개의 값을 저장한 리스트와 힙에 대해)

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

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)

```

- heapfu_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:
 - a. heappush(h, key): 힙 h에 key 값을 삽입 (= insert와 동일)
 - i. heappush(h, (key, value))처럼 튜플 삽입 가능
 - b. heappop(h): 최소값을 지우고 리턴 (delete_min의 역할)
 - c. heapify(A): 리스트 A를 힙 성질이 만족되도록 변환
 - i. make_heap()과 동일 (단, min-heap으로 변환)
 - d. h[0]: 힙의 최소값을 알고 싶다면

3. 적응형 힙 (Adaptive Heap)

- 앞에서 살펴본 힙 제공 연산은 delete_max, insert, heap_sort 등이다
- 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
- 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
- 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
- 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 index를 알아야 한다. 어떻게?

- f. **해결책**: 각 key 값이 저장된 위치 (index)를 기억하고 있도록 함!
- g. **Locator 클래스**: key 값과 key 값이 저장된 index를 쌍을 담는 클래스 선언

```
class Locator:
    def __init__(self, key, value, j):
        self.key = key      # key 값
        self.value = value # value 값 (optional)
        self.index = j      # key 값이 저장된 index j
```

- h. **AdapedHeap 클래스**:

```
class AdapedHeap:
    # Locator 클래스를 여기서 선언 (클래스 안에 다른 클래스 선언 가능)
    def __init__(self):
        self.A = [] # 여기선 빈 리스트로 초기화

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

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

    def insert(self, key, value=None):
        loc = self.Locator(key, value, len(self.A))
        # 마지막에 (key, index) Locator 객체 삽입
        self.A.append(loc) # 힙 리스트에 item 삽입됨!
        self.heapify_up(loc.index)
        return loc

    def heapify_up(self, i): # 인덱스 i에 저장된 item을 up!
        p = (i - 1)//2
        if p > 0 and self[p].key < self[i].key:
            # key swap!
            self.A[i].key, self.A[p].key = \
                self.A[p].key, self.A[i].key
            # index swap!
            self.A[i].index = i # correct?
            self.A[p].index = k # correct?
            heapify_up(p)

    def heapify_down(self, i):
        # heapify_up과 유사하게 작성 가능!
```

- i. **remove** 함수를 작성해보자

```
def remove(self, loc): # key 값이 아닌 Locator 객체가 전달됨
```

```

k = loc.index # self.A[k]에 있는 item을 지우면 됨!

if not (0 <= k < len(self) and self.A[k] == loc):
    # loc이 힙에 저장된 것이 아니라면 error
    raise ValueError('Invalid locator')

# 1. A[k]와 A[-1]을 swap!
swap self.A[k] and self.A[-1]
swap indices of A[k] and A[-1]
# 2. A[-1]을 pop!
self.A.pop()
# 3. A[k]를 heapify_down해서 재 배치
self.heapify_down(k)

```

j. delete_min (또는 delete_max) 함수를 작성해보자: remove 함수 이용

k. update_key 함수를 작성해보자

```

def update_key(self, loc, new_key):

    # loc.index에 저장된 key 값을 new_key 값으로 대체

    k = loc.index
    if not (0 <= k < len(self) and self.A[k] == loc):
        # loc이 힙에 저장된 것이 아니라면 error
        raise ValueError('Invalid locator')

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

```

l. remove, delete_min/delete_max, update_key의 수행시간은?

i. 왜?

4. 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 큐라고 부른다
 - i. priority 값에 따라 데이터를 관리할 필요가 있는 경우에 필수적
- b. 대표적인 우선순위 큐
 - i. stack, queue, dequeue (삽입 시간을 일종의 priority로 간주)
 - ii. heap, adaptive heap
 - iii. 아래 표는 wikipedia에서 정리한 여러 종류의 우선순위 큐
 - Binary가 heap을 의미!

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

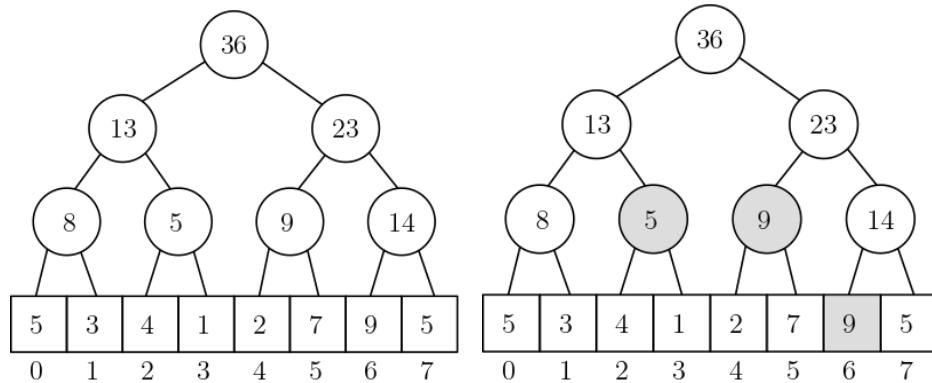
5. [고급] 힙 모양 이진트리 1: 범위 트리 (Segment Tree)

- a. 아래 그림처럼 8개의 수가 리스트에 저장되어 있다고 하자. 인덱스 범위 $[3, 7]$ 이 주어지고 이 인덱스 범위에 있는 값의 합 $\text{sum}(2, 8) = A[2] + \dots + A[6] = 23$ 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

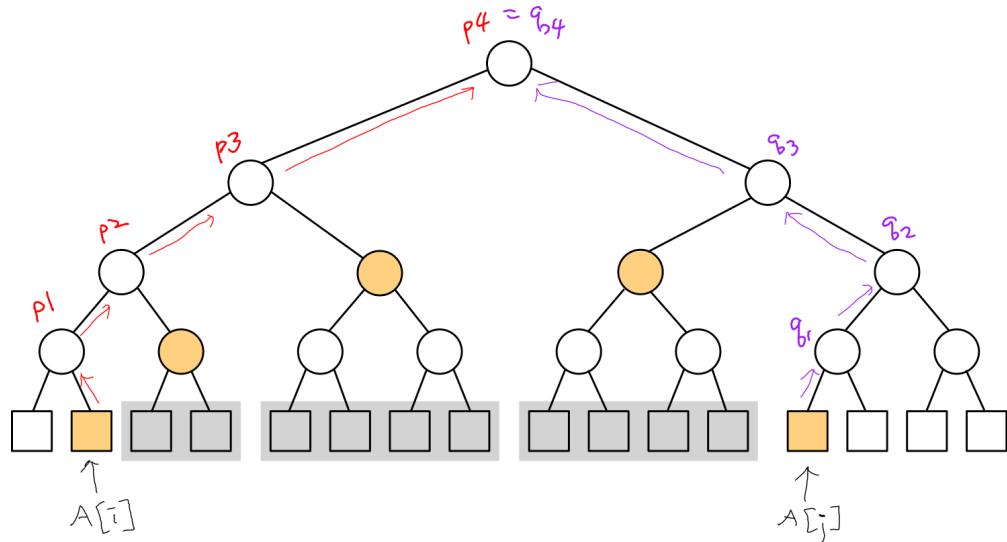
5	3	4	1	2	7	9	5
0	1	2	3	4	5	6	7

- c. 여러 개의 질의를 처리한다는 가정하에 리스트의 값을 미리 전처리 (preprocessing) 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 합을 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 합으로 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. $\text{sum}(1, 7) = (13) + (9) + A[6]$ # 3개의 노드로 표현 가능
 - iii. $\text{sum}(4, 7) = (23)$ # 노드 1개로 표현 가능
 - iv. $\text{sum}(i, j) = A$ 에 n개의 값이 있다면 최대 $2\log n + 2$ 개의 노드로 표현 가능!

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



v. 노란색 노드는 어떻게 찾을 수 있을까?

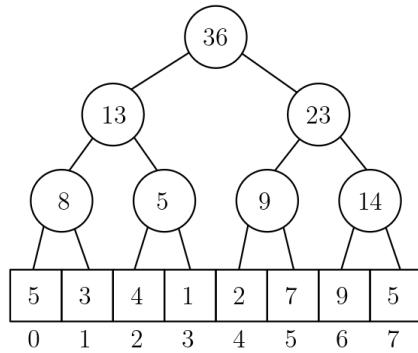
- $A[i]$ 와 $A[j]$ 에서 루트노드를 향해 동시에 올라가보자. 동시에 한 레벨씩 올라가기 때문에 언제가는 같은 노드 ($p4 = q4$)에 도착하게 된다 (이 노드를 $A[i]$ 와 $A[j]$ 의 **lowest common ancestor (LCA)** 노드라 부른다)
- $A[i] \rightarrow p1 \rightarrow p2 \rightarrow p3$ 에서 각 노드의 **오른쪽 자식 노드**가 노란색 노드가 되고, $A[j] \rightarrow q1 \rightarrow q2 \rightarrow q3$ 에서 각 노드의 **왼쪽 자식 노드**가 노란색 노드가 된다

vi. 노란색 노드는 몇 개일까?

- $A[i]$ 와 $A[j]$ 에서 한 레벨씩 올라가면서 각각 레벨당 최대 하나의 노드가 노란색 노드가 되기에 **트리 높이의 2배**를 넘지 않게 된다
- 트리를 힙 모양의 이진트리로 관리하기 때문에 $O(\log n)$ 개 정도가 된다

vii. 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
- i. $A[i]$ 에서 부모노드로 올라가면서 오른쪽 자식노드의 값을 계속 더하고, $A[j]$ 에서 부모노드로 올라가면서 왼쪽 자식노드의 값을 더한다

```

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

```

- ii. 문제는 입력 리스트 A 의 크기가 2^k 의 형태면 입력 값이 힙 구조 트리의 가장 아래 레벨에 놓이게 되어 위의 sum 함수가 잘 동작하지만, 그렇지 않은 경우엔 올바르게 작동하지 않을 수 있다 → 만약 $2^{k-1} < n < 2^k$ 라면, A 에 $2^k - n$ 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? tree 의 값을 역시 부분적으로 수정되어야 한다
- i. $A[k]$ 부터 루트 노드를 향해 올라가면서 방문하는 노드의 sum 값을 갱신하면 된다. 따라서 $O(\log n)$ 시간에 가능

```

def update(A, tree, k, x): # A[k]의 값을 x로 update
    k += n-1      # n = len(A)

```

```

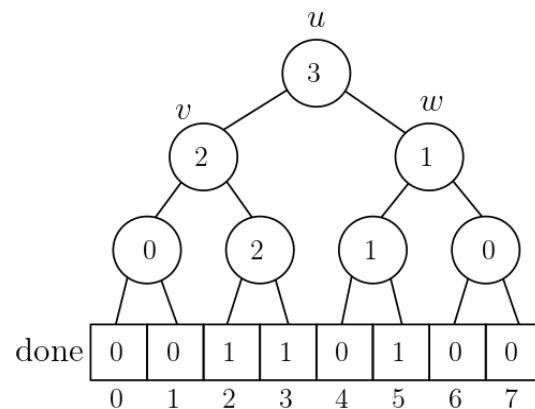
A[k] = x
while k >= 0:
    k = (k-1)//2
    A[k] = A[2*k+1] + A[2*k+2]

```

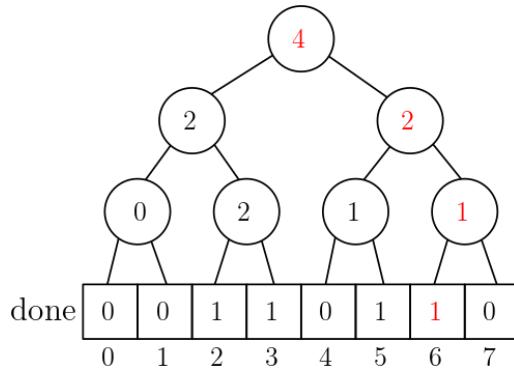
h. 응용 예: 순열복원 문제

- i. 0부터 $n-1$ 까지 서로 다른 수로 구성된 순열 (permutation) A에 대해, 새로운 리스트 S를 준비했다
 - $S[i] = A[0]$ 부터 $A[i-1]$ 까지 $A[i]$ 보다 작은 수의 개수
 - $S[0] = 0$
 - 즉, $A[i]$ 의 왼쪽에 있는 값 중에서 $A[i]$ 보다 작은 값의 개수를 $S[i]$ 에 계산해 저장했다
- ii. 실수로 리스트 A를 잃어버려서 S만 가지고 있다. S를 이용해 A를 재구성하는 문제
 - **입력:** 첫 줄에 리스트 B의 n개의 값 (n의 값은 1 이상 1000이하)
 - **출력:** `print(A)`
- iii. $A[i]$ 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 $L[i]$ 를 모른다는 것이 문제임
- iv. 예: $S = [0, 0, 2, 2, 3, 2, 4, 2]$, $n = 8$
 - $S[n-1]=S[7]=2$ 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 $A[7]$ 위치는 쉽게 결정 가능!
 - 새로운 리스트 done을 정의하고 0으로 초기화 한다. 이 done에는 현재까지 복원한 A의 값을 1로 표시해 마크한다. 2가 복원되었으므로 $done[2] = 1$ 로 변경함
 - $S[6] = 4$ 에 대해, $A[6]$ 의 오른쪽에 있는 $A[6]$ 보다 작은 값의 개수는 done의 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
 - 현재 $done = [0, 0, 1, 0, 0, 1, 0, 0]$
 - $S[5] = 2$ 이고, $done[0..2]$ 까지의 1의 개수가 1이므로 $A[5] = 2+1 = 3$ 이 되고, $done=[0, 0, 1, 1, 0, 1, 0, 0]$ 이 됨
 - $S[4] = 3$ 이고, $done[0..3]$ 까지의 1의 개수가 두 개고 $A[4] = 3+2 = 5$ 이 가능하지만, $done[5] = 1$ 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 done 리스트의 값을 prefix sum 계산처럼 하나씩 더하면서 결정할 수 있다.

- 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
 - 범위 트리를 사용하면 더 빠르게 할 수 있다
- v. 범위 트리를 사용해서 $L[i]$ 를 $O(\log n)$ 시간에 찾아보자. 그러면 전체 수행시간은 $O(n \log n)$ 이면 충분하다
- 현재 $A[4]$ 를 결정하는 순간이라고 하자
 - $\text{done} = [0, 0, 1, 1, 0, 1, 0, 0]$ 임
 - done 을 범위 트리로 표현하면 아래 그림과 같다
 - a. 내부노드에는 리프노드의 1의 개수가 저장
 - b. 동시에 리프노드 개수도 함께 저장



- 루트노드 u 부터 탐색을 시작하는 데, 루트노드의 왼쪽 자식노드 v 의 값 2와 $S[4]=3$ 을 더하면 5가 된다. 즉 $A[4]$ 는 자신보다 작은 수가 최소 5개 이상이어야 한다 (**등수로는 6등 이상**). 그런데 v 의 리프노드 개수는 4개뿐이므로, $A[4]$ 의 등수인 6등에 비해 2가 모자란다. 그래서 오른쪽 자식노드 w 를 따라 내려가 자신에 해당하는 리프노드 위치를 찾아야 한다
- 이 때, 오른쪽 자식노드로 내려가면서 찾아야 할 등수는 $6-4=2$ 가 되어야 한다 (왜냐하면 $A[4]$ 의 현재 등수는 6위인데, 왼쪽 리프노드 수는 4개뿐이므로)
- 오른쪽 자식 노드 w 로 내려와서, 다시 w 의 왼쪽 자식노드를 살펴본다. 왼쪽 서브트리에 있는 1의 개수가 하나 뿐이므로 현재 등수 2와 1을 더하면 3이 되지만 w 의 리프노드는 2개가 되어 여전히 1이 모자란다. 그래서 다시 w 의 오른쪽 자식노드로 수정된 등수 1 (= 2-1)을 가지고 내려간다.
- 이러한 과정을 거쳐 한 레벨 더 내려가면 리프노드 $\text{done}[6]$ 에 도달하고 $A[4] = 6$ 라고 결정할 수 있다
- 이 결정에 따라 $\text{done}[6] = 1$ 이 되어야 하고, 범위 트리도 그에 맞게 수정되어야 한다. (아래 그림 참조)



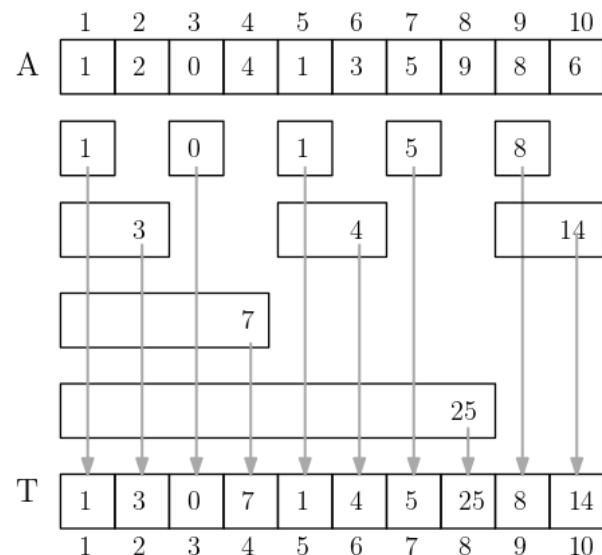
- $A[i]$ 를 결정하는 건 루트노드에서 리프노드로 탐색하는 과정이므로 $O(\log n)$ 시간이면 충분하고, $done$ 의 값을 업데이트하는 건 거꾸로 리프노드에서 루트노드로 올라가면서 수정하는 것이므로 역시 $O(\log n)$ 시간이면 된다. 총 $O(n \log n)$ 시간에 계산 가능하므로 전체적으로는 $O(n \log n)$ 시간에 순열을 복원할 수 있다

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

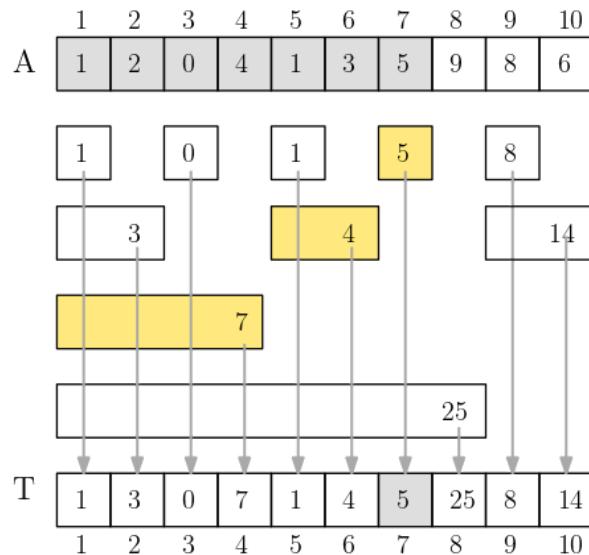
- Fenwick tree 라고도 불림 https://en.wikipedia.org/wiki/Fenwick_tree
- 리스트 A 에서 $\text{prefix_sum}(k) = A[1] + \dots + A[k]$ 로 정의된다
- n 개의 값이 저장된 리스트 A 를 전처리해 BIT 트리를 구성해 놓으면 $\text{prefix_sum}(k)$ 질의와 $\text{update}(k, x)$ 연산을 $O(\log n)$ 시간에 답할 수 있다
- BIT 트리 T 는 다음 두 연산을 지원한다
 - $\text{prefix_sum}(k)$ 연산: $O(\log n)$ 시간에 수행
 - $\text{update}(k, x)$ 연산: $A[k]$ 의 값을 x 로 $O(\log n)$ 시간에 업데이트
- $A = [\text{None}, 1, 2, 0, 4, 1, 3, 5, 9]$
 - $A[0]$ 는 사용하지 않고 $A[1]$ 부터 사용한다고 하자
 - 설명이 간단해지는 장점. $A[0]$ 부터 시작하는 경우로 쉽게 변경 가능
 - $k = 6$ 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
 - 6을 이진수로 표현하면 110이고, $110 = 100 + 010 = 4 + 2$ 가 되므로 4개의 합과 2개의 합으로 분할할 수 있음을 알 수 있다
 - 7의 경우에는 111이므로 $111 = 100 + 010 + 001 = 4 + 2 + 1$ 로 세 합으로 표현할 수 있다
 - 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 T 를 리스트로 저장하는데, $T[45]$ 에는 $A[45]$ 값을 저장하고, $T[44]$ 에는 $(A[41]+\dots+A[44])$ 을 저장하고, $T[40]$ 에는

$(A[33]+...+A[40])$ 을 저장하고, $T[32]$ 에는 $(A[1]+...+A[32])$ 를 저장한다. 그러면 k 를 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 T 에 저장한다 (트리가 리스트로 표현됨에 주의)



- $T[k] = k$ 의 이진수에서 1이 처음으로 등장하는 LSB(Least Significant Bit)의 값의 개수만큼 A 의 값을 더해 저장한다. 즉, $T[k] = A[k-LSB(k)+1]+...+A[k]$ 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. $T[8] = A[1]+...+A[8] = 25$



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

$$\begin{aligned}\text{prefix_sum}(7) &= T[111] + T[111-001] + T[110-010] \\ &= T[111] + T[111-\text{LSB}(111)] + T[110-\text{LSB}(110)]\end{aligned}$$

```
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

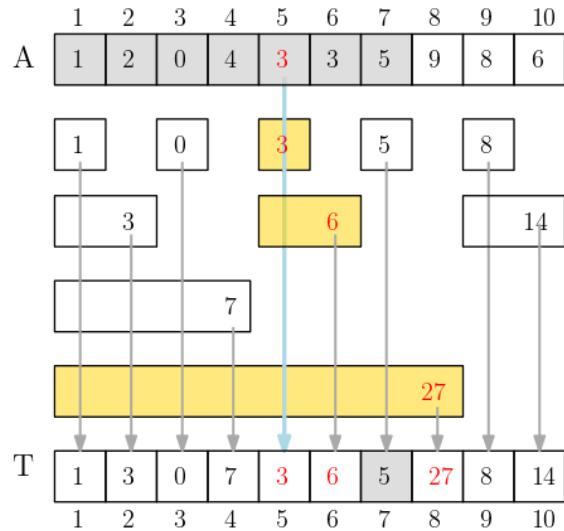
```
def LSB(k):
    return k & -k
```

- $-k$ 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. prefix_sum 의 수행시간: $O(\log n)$

- 길이가 $1, 2, 2^2, 2^3, \dots$ 인 구간들의 합으로 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 → 3으로 수정했다고 하자. 그러면 $T[5], T[6], T[8]$ 의 값이 영향을 받기 때문에 이 세 값을 바꿔야 한다



- 영향 받는 T의 값들을 어떻게 결정하면 될까? $A[5]$ 를 포함하는 $T[i]$ 를 결정하면 된다: 당연히 $i > k$ 이어야 한다. $5 = 101$ 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 과정에서의 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 트리 T를 구성하는 데 필요한 시간은? $O(n)$

- 어떻게 하면 선형시간에 구성할 수 있을까?

xii. BIT 트리를 이용한 예:

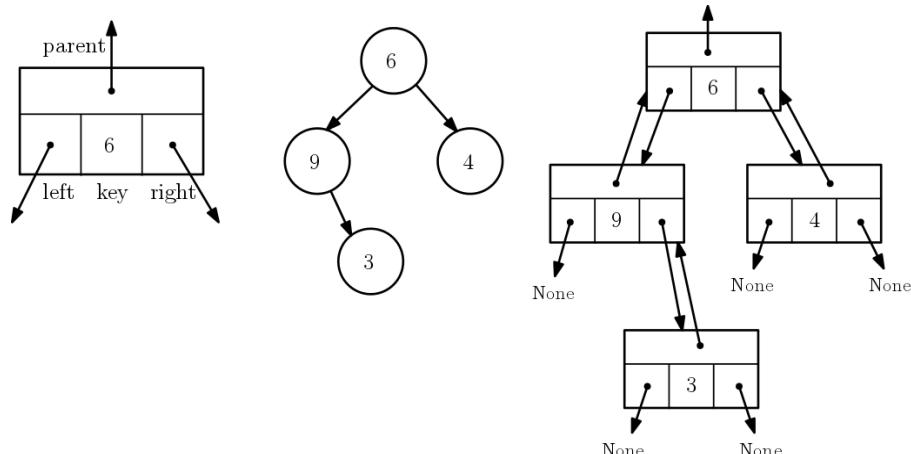
- prefix 구간을 질의로 주고, 해당 구간에 대한 합, 최소, 최대값 등을 $O(\log n)$ 시간에 답할 수 있다
- 인덱스 구간 $[i, j]$ 를 주고, 구간 합 $A[i] + \dots + A[j]$ 를 $O(\log n)$ 시간에 계산할 수 있다 (segment tree를 이용한 경우와 같은 시간)
 - a. $A[i] + \dots + A[j] = \text{prefix_sum}(j) - \text{prefix_sum}(i-1)$ 이므로 가능하다
- 순열복원 문제를 segment tree 대신 BIT 트리를 이용해서 해결 가능할까?

7. 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
        # 필요하면 추가 정보 - 예: self.height = 0
        self.parent = parent
        self.left = left
        self.right = right

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

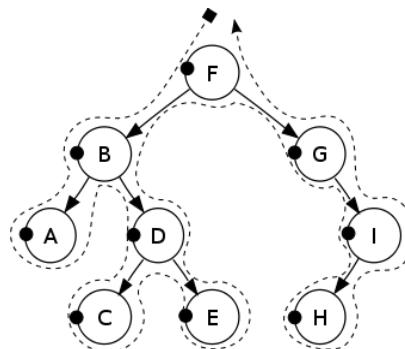


- iii. 위 그림의 가장 왼쪽은 treeNode 클래스의 멤버들을 도형으로 표현한 것이고, 가운데 트리를 실제 treeNode를 이용해 연결한 것이 가장 오른쪽 그림이다

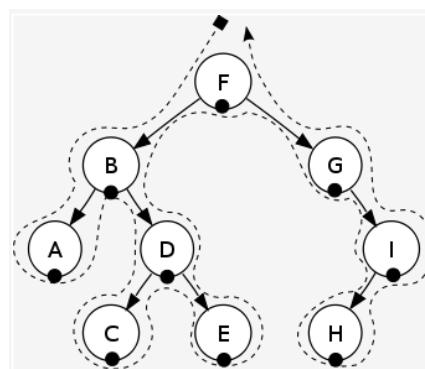
b. 순회(traversal)

- 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)
 - **Inorder:** LMR (Left subtree 순회 → Middle 노드 출력 → Right subtree 순회)
 - **Postorder:** LRM (Left subtree 순회 → Right subtree 순회 → Middle 노드 출력)

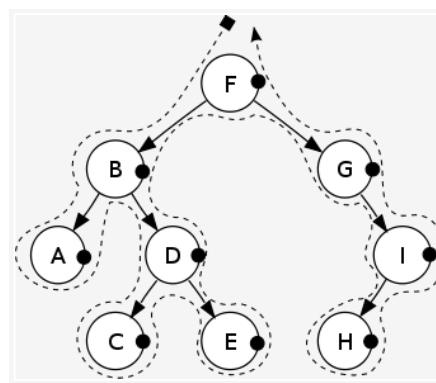
Pre-order: F, B, A, D, C, E, G, I, H



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)
        if self.left: self.left.preorder()
        if self.right: self.right.preorder()
```

예2: Tree/BST 클래스의 메소드로 preorder 정의

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

- iv. [?] 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
a. 앞에서 예를 든 이진트리
- 예 2: **preorder**: A,B,D,F,C,E **inorder**: B,F,D,A,E,C (직접 재구성해서 아래에 그려볼 것)

- postorder와 inorder가 주어지면 항상 복원가능한가?
- preorder와 postorder가 주어지면 항상 복원가능한가?

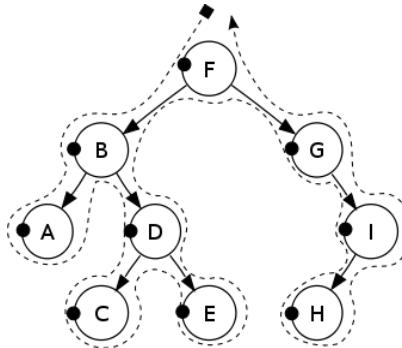
- v. [Python: 고급] iterator를 만들고 싶다면? 즉, `for key in v:` 라고 하면 v와 자손노드들을 특정 (pre-, in-, post-) order에 따라 반복하고 싶다면?

- 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 - 난이도 상] LCA (Lowest Common Ancestor) 문제:

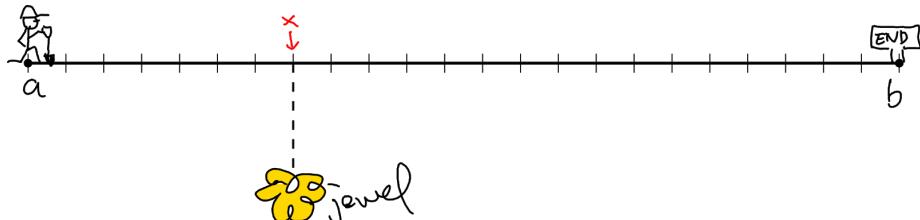
- i. 이진트리의 두 노드 u, v 에 대해, u 와 v 의 공통 조상 노드 중 가장 깊이가 깊은 노드를 LCA라 정의한다 (u 와 v 에 더 가까운 공통 조상 노드가 LCA가 된다)
 - 주의: 이진트리가 아니어도 상관없음!



- ii. 위의 트리 예에서 노드 A와 C의 공통 조상 노드는 B, F 두 노드가 존재한다. 그 중에서 B가 루트 노드로부터 더 깊이 있기 때문에 (A, C에 더 가깝기 때문에) LCA가 된다. 노드 B와 E에 대해선 B가 E의 조상 노드이기 때문에 B가 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. (1) 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
 - `preorder`를 통해 노드를 방문하는 시간을 기록해 `pre[v]`에 저장한다. 즉, 루트 노드 v 의 `pre[v] = 1`이고, 방문할 때마다 시간을 1씩 증가시킨다. `postorder`를 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 가능하다. 이 정보로부터 $\text{is_ancestor}(u, v)$ 의 답을 어떻게 $O(1)$ 시간에 알 수 있을까?

viii. (2) 다음으로 아래와 같은 퀴즈를 하나 풀어보자



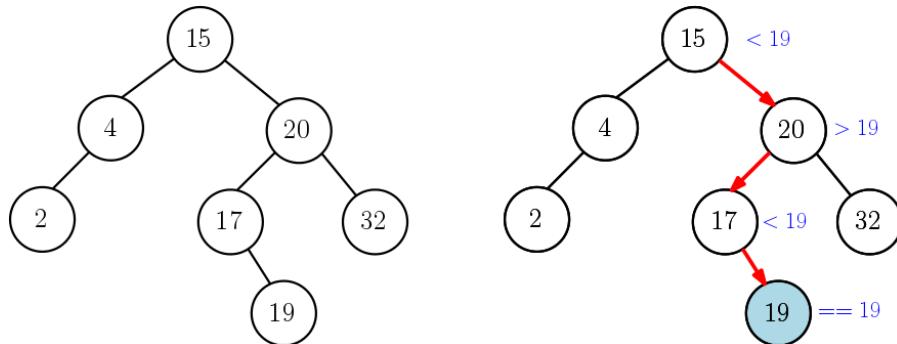
- 여러분은 a에 서 있고, a와 b 사이의 어느 지점 지하에 보물이 숨겨져 있다. 문제는 직접 파보지 않고는 보물이 있는지 알 수 없다는 것이다
- 파는 횟수를 최소로 하려면 어떤 전략이 필요할까? 단, a와 b 사이의 거리를 D라고 하고, 거리 단위는 1이라고 하자. 파는 횟수를 D에 관한 식으로 표현할 수 있을까?
- [힌트] 리스트의 인터뷰 문제 중 이와 비슷한 문제와 밀접한 관계가 있음
- [정답] $O(\log D)$ 번!

- ix. 앞의 퀴즈에서 여러분이 서 있는 처음 위치 a를 노드 u라고 하고, 보물을 v라고 하면, u와 v의 LCA는 바로 x 지점이 된다: $\text{LCA}(u, v) = x$
- 지점 x에서 땅을 판다는 건 x가 v의 조상인지를 아는 것과 같다. 이건 $\text{is_ancestor}(x, v)$ 연산으로 $O(1)$ 시간에 가능하다
 - 총 $O(\log D)$ 번 파야하고 매번 $O(1)$ 시간이 필요하므로 $O(\log D)$ 시간이면 보물을 찾을 수 있다
 - 여기서 END 표시된 지점은 트리의 루트 노드라고 한다면 D는 최대 트리의 높이 정도가 되므로 $O(\log D) = O(\log n)$ 이 된다. 따라서 $O(\log n)$ 시간에 보물을 알아낼 수 있다.
- x. 보물 찾기 문제를 트리 문제로 바꿔 생각해도 전혀 문제 없다. 문제는 파야할 지점에 해당하는 트리 노드 x를 전처리 단계의 작업을 통해 쉽게 알 수 있도록 처리를 하는 것이다.
- xi. 어떤 노드 u의 바로 1대 조상은 u의 부모 노드이다. 2대 조상은 u의 조부모 노드이다. 이런 식으로 따져서 k대 조상도 자연스럽게 정의할 수 있다. 물론, k대 조상이 존재하지 않을 수도 있다. 이 경우엔 루트 노드를 k대 조상으로 지정한다.
- xii. 그럼 트리의 각 노드 u에 대해, 이차원 리스트의 값 $\text{ancestor}[u][i]$ 를 노드 u의 2^i 대 조상 노드라고 정의하자. 이 ancestor 리스트만 전처리 단계에서 미리 계산해 놓으면, 위의 보물 퀴즈 해결법으로 $O(\log n)$ 시간에 u와 v의 LCA를 계산할 수 있다!

- xiii. 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] $O(n \log n)$ 에 가능하다
 - [힌트 2] preorder 순회를 하면서 계산 할 수 있다
 - $\text{ancestor}[u][0] = u.\text{parent}$ # 2^0 대 조상이니 부모노드
 - preorder로 노드 u 에 도착했다는 의미는 u 의 조상노드를 모두 방문한 후 u 에 도착했다는 의미
 - 일종의 DP(동적계획법) 식으로 계산 가능 (알고리즘 교재 참조)
 - $\text{ancestor}[u][i] = u$ 의 2^i 대 조상노드
 $= (u$ 의 2^{i-1} 대 조상노드) $\text{의 } 2^{i-1} \text{ 대 조상노드}$
 $= (\text{ancestor}[u][i-1])\text{의 } 2^{i-1} \text{ 대 조상노드}$
 $= \text{ancestor}[\text{ancestor}[u][i-1]][i-1]$
- xiv. $\text{ancestor}[u][i]$ 값처럼 다른 종류의 값들도 같은 방식으로 계산해서 저장할 수 있다. 예를 들어, 노드 u 에서 2^i 대 조상 노드까지의 경로에 있는 노드의 값(value) 중에서 최대값 또는 최소값을 저장할 수 있다. 즉, $\text{max_value}[u][i] = u$ 에서 2^i 대 조상노드의 경로의 특정 값처럼 정의하면 $O(n \log n)$ 시간에 2차원 리스트 max_value 를 계산할 수 있다 → 이러한 기법을 **binary lifting**이라 부른다. 알고리즘 교재의 Graph > MST(최소신장트리) 편의 second best MST 알고리즘에서 이 자료구조를 사용한다

8. Binary Search Tree(BST: 이진탐색트리)

- a. 이진탐색트리(BST)는 저장된 key 값들이 아래의 성질을 만족하는 이진트리
 - 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 search(key) 함수: 이진탐색트리 T 의 노드 v 와 v 의 자손 노드들중에서 key 값을 갖는 노드를 찾아 리턴하거나 없다면 None을 리턴함

- i. `find_loc(key)` 함수를 먼저 구현함 - key 값이 있다면 해당 노드 리턴, 없다면 그 값이 삽입될 곳의 부모 노드 리턴

```
def search(self, key):
    p = self.find_loc(key)
    if p and p.key == key: # key is in tree
        return p
```

```

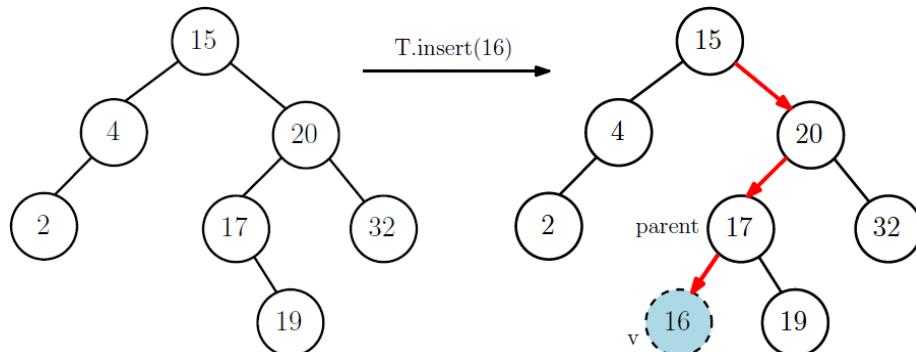
        else: return None      # key is not in tree
def find_loc(self, key):
    if self.size == 0: return None
    p = None    # p is parent of v
    v = self.root
    while v:    # while v != None
        if v.key == key: return v
        elif v.key < key:
            p = v
            v = v.right
        else:
            p = v
            v = v.left
    return p

```

e. 위의 그림의 오른쪽은 `find_loc(19)`, `search(19)`의 과정을 표현한 것이다.

f. 삽입 연산: `insert(key)`

- i. 탐색트리이기 때문에, key가 들어갈 위치가 정해진다.
- ii. 아래 그림의 왼쪽 트리에 `key = 16`을 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):
    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
        return v
    else:
        print("key is already in the tree!")
        return p # 중복 key를 허용하지 않으면 None 리턴

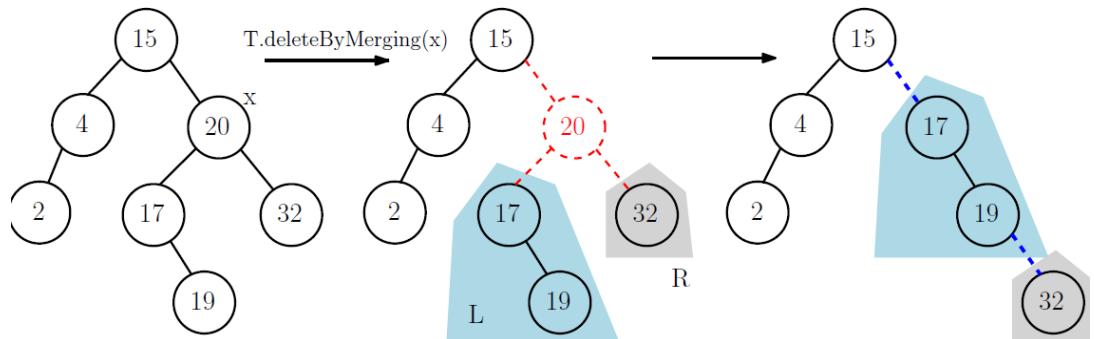
```

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)` 메소드 → `tree[key]`로 `value` 접근 가능
 - `__setitem__(self, key, value)` 메소드 → `tree[key] = value`로 삽입 가능

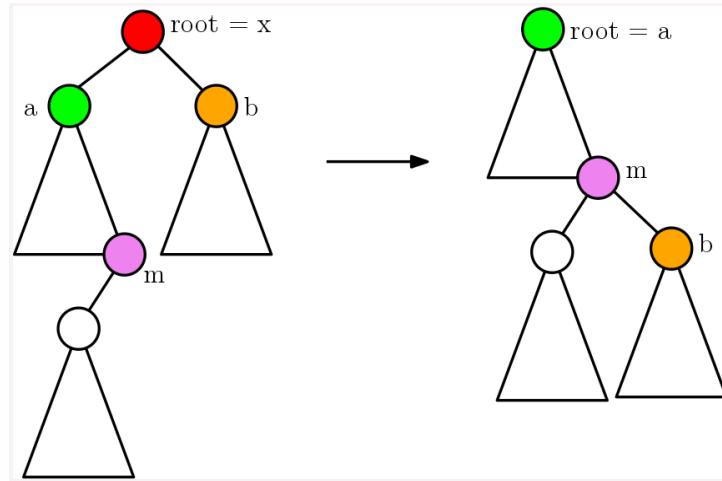
h. 삭제 연산: `delete`

- 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



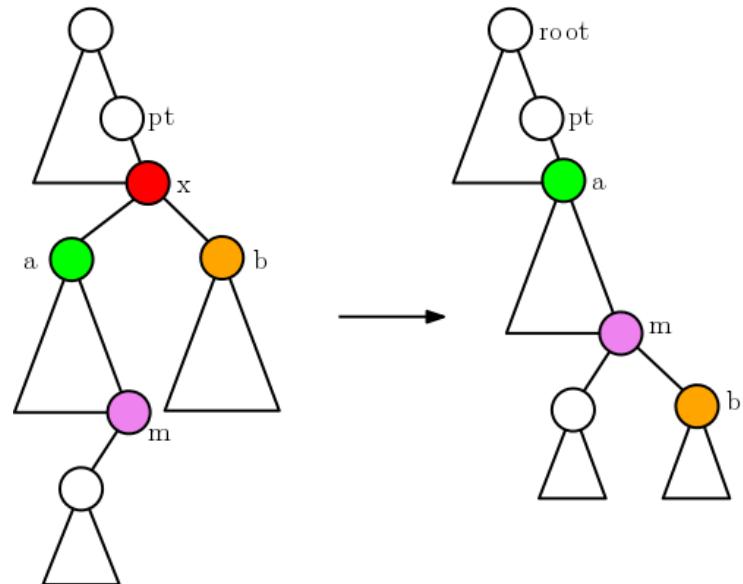
- 두 가지 경우로 나눈다: 삭제할 노드 `x`가 `root` 노드인 경우와 아닌 경우
 - `a = x.left, b = x.right`로 지정
- 1. `root == x` 인 경우 (즉, 삭제할 노드가 루트노드인 경우: 삭제 후 트리의 루트가 바뀜에 주의!) [아래 그림 참조]

- m 이 존재한다면, 즉 $a \neq None$ 라면, b 가 m 의 오른쪽 자식노드가 되도록 링크 수정한 후, `self.root = a`로 변경
- $a == None$ 이면, b 가 새로운 루트가 됨



2. `root != 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
- $a == None$ 이면, b 가 pt 의 자식노드가 되도록 함



- Pseudo 코드:

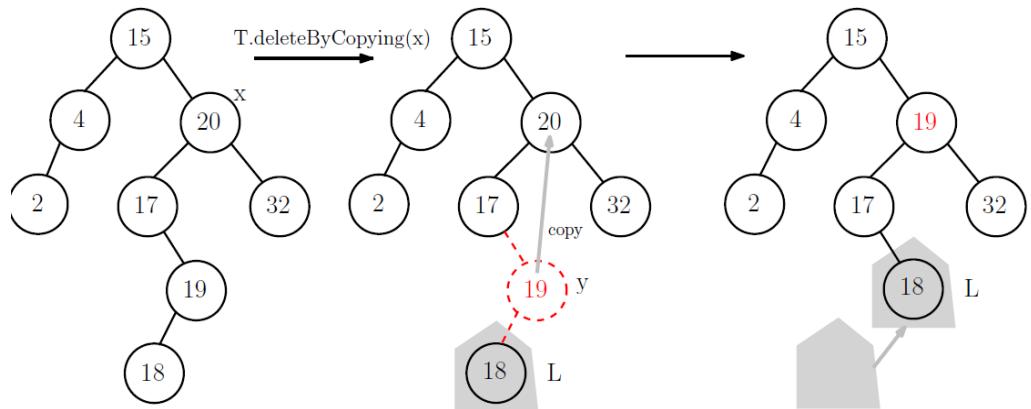
```

def deleteByMerging(self, x):
    # assume that x is not None
    a, b, pt = x.left, x.right, x.parent
    # c = node which will be at the position x
    # s = 균형이 깨진 첫 번째 노드 (균형이진탐색트리에 이용)
    if a == None:
        c = b
        s = pt
    else: # a != None
        c = m = a
        while m.right:
            m = m.right # find m
        # 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
    return s # first node that would be rebalanced

```

- iii. **deleteByCopying**

- 노드 x를 제거한다고 하면, x의 왼쪽 서브트리 L과 오른쪽 서브트리 R을 아래와 같이 조정한다.
 - L에서 가장 큰 값을 갖는 노드 y를 찾는다.
 - y의 key 값을 x의 key 값으로 카피한다.
 - y의 왼쪽 서브트리가 존재한다면, y의 위치로 옮긴다.

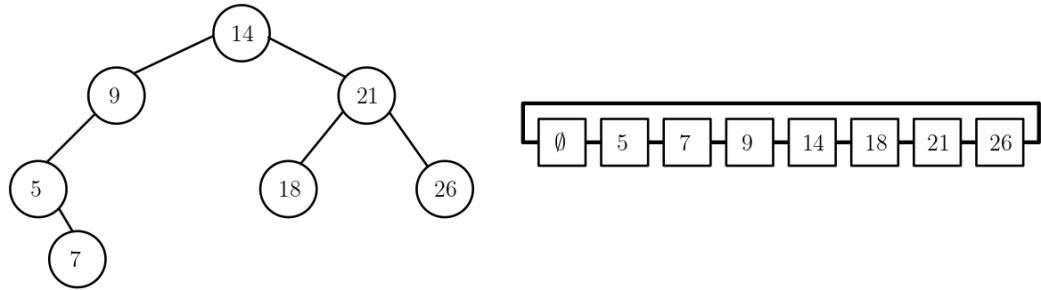


- 이 경우도 Merging 방법처럼 몇 가지 경우로 나눠 경우에 알맞은 방법으로 처리한다
- [?] deleteByCopying 함수는 각자 구현해볼 것!

i. 연산 수행시간

- i. `search`는 최악의 경우에 가장 깊은 곳의 노드까지 비교하면서 내려가야 하므로 트리의 높이에 비례하는 시간이 필요하다. 즉 트리의 높이를 h 라고 하면, $O(h)$ 시간이 걸린다
- ii. `insert` 역시 `search` 과정을 통해 새로운 노드가 삽입될 위치를 찾은 후, 몇 개 노드의 `parent`, `left`, `right`를 수정하는 것이므로 $O(h)$ 시간이 걸린다
- iii. `delete`도 m 을 찾기 위해 최악의 경우에 h 만큼 비교하면서 내려가기 때문에 $O(h)$ 시간이 걸린다
- iv. 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?
 - n 개의 노드를 갖는 이진트리의 높이는 최소 _____ 이상인가?

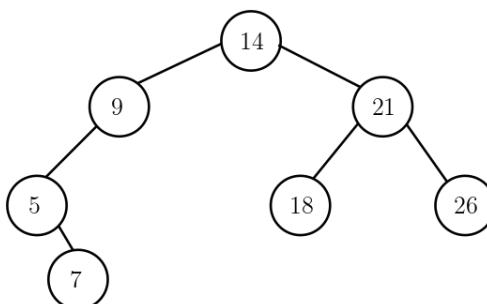
- k. [☞ 인터뷰 문제 1: BST를 양방향 연결리스트로 변환] 이진탐색트리 T 가 주어지면, inorder 순서가 되도록 양방향 연결리스트로 변환하는 함수를 작성하자. 예를 들어, 아래 왼쪽 그림의 BST T 를 오른쪽 그림의 양방향 연결리스트 L 로 변환하는 것이다. BST 클래스와 양방향 연결리스트 클래스를 활용해 구현해보자.



i. Pseudo 코드를 작성해보자

ii. 수행시간은?

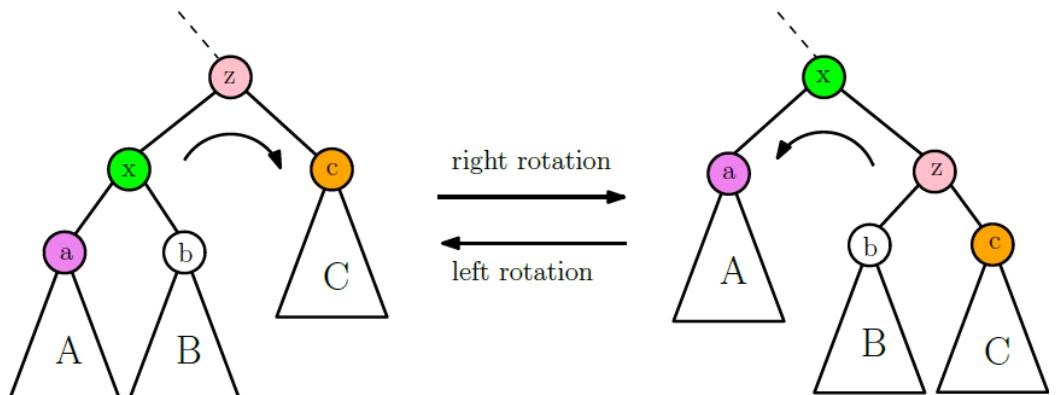
- l. [☞ 인터뷰 문제 2: BST를 양방향 연결리스트로 변환] 예를 들어, 노드가 없는 빈 BST에 insert 연산을 반복해서 아래와 같은 트리를 얻었다고 하자
- 가장 먼저 insert되는 key는 무엇인가? _____
 - 전체 key에 대한 insert 되는 순서는 유일하지 않다. 14,9,5,7,21,18,26 순서로 삽입해도 되지만, 14,5,9,7,21,26,18 순서도 가능하다. 아래 트리에 대해 이런 순서의 경우의 수는 얼마일까?



- m. [복습] [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

9. 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, 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 **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 AxBzC이고, 회전 후의 순서 역시 AxBzC이므로 같다. left rotation 전, 후의 순서도 같다. 따라서 rotation이 BST의 순서를 그대로 유지한다
 - iv. 수행시간: 상수 개의 링크 수정이면 충분하므로 **O(1)**



```

def rotateRight(self, z):          # rotateLeft도 유사하게 정의
    if not z: return              # check if z == None
    x = z.left
    if x == None: return          # if x == None: nothing changed
    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

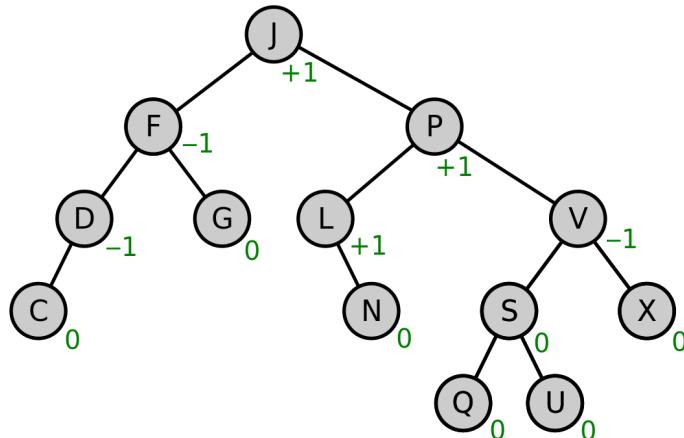
# z == self.root라면 x가 새로운 루트가 되어야 함!
if z == self.root:
    self.root = x
# [주의] height가 있다면 x와 z의 height 값 수정하는 코드 추가

```

f. AVL 트리

- i. 정의: 모든 노드에 대해서, 노드의 왼쪽 부트리와 오른쪽 부트리의 높이 차이가 1 이하인 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? 그 트리는 노드 수가 얼마인가?
 - [?] 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?)
- a. $N_h = N_{h-1} + N_{h-2} + 1$

b. $N(h)$ 계산방법 1 [정확하게~]

- i. $N_h + 1 = N_{h-1} + N_{h-2} + 1 + 1$
 $N_h + 1 = (N_{h-1} + 1) + (N_{h-2} + 1)$
- ii. $F_h = F_{h-1} + F_{h-2}$ ($\leftarrow F_h = N_h + 1$),
- iii. $F_0 = N_0 + 1 = 2$, $F_1 = N_1 + 1 = 3$ 이므로, 2와 3으로 시작하는 피보나치 수열 중 h 번째 값이 F_h 가 됨!
- iv. 일반적인 피보나치 수열은 0과 1로 시작하므로, F_h 는 일반 피보나치 수열의 $h+3$ 번째 피보나치 값!
- v. [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
- vi. $N_h + 1 = F_h \geq \phi^{h+3}/\sqrt{2}$
- vii. $n = N_h \geq \phi^{h+3}/\sqrt{2} - 1 \rightarrow h \leq c \log(n + 2) + b$
 $c = 1/\log 2, b \approx 0.328$

c. $N(h)$ 계산방법 2 [정확한 값이 아닌 유사한 값 계산]

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

- iv. 결론: n 개의 노드로 구성된 AVL 트리의 높이 h 는

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

임이 증명됨. 따라서 균형이진탐색트리이다

- v. 삽입과 삭제를 하게 되면, 어떤 노드의 두 서브트리의 높이 차가 1보다 크게 되는 경우가 있을 수 있음. 그 경우엔 특별한 규칙에 따라 (single/double) 회전을 통해 조건이 만족되도록 트리를 재조정(rebalancing)하게 됨:

- search: 트리 높이 h 가 $O(\log n)$ 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 클래스에 `self.height = 0`를 추가해 노드의 높이 정보를 저장

```
class AVL(BST): # BST 클래스를 부모 클래스로 지정
    # BST 클래스의 멤버, 메소드 상속받아 사용 가능
    # __init__가 없다면 부모 클래스 __init__ 함수 자동 호출
```

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

1. v = 보모클래스 BST의 insert 함수 호출 (트리에 삽입)
2. v에 의해 AVL 트리의 균형(-1, 0, 1)이 깨지면 rebalance!

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

...

- 주의: 노드의 height 값은 insert, delete 등을 통해 수정되어야 할 경우가 존재하므로 height 값을 변경되도록 각 연산에 적절한 코드를 삽입해야 한다!

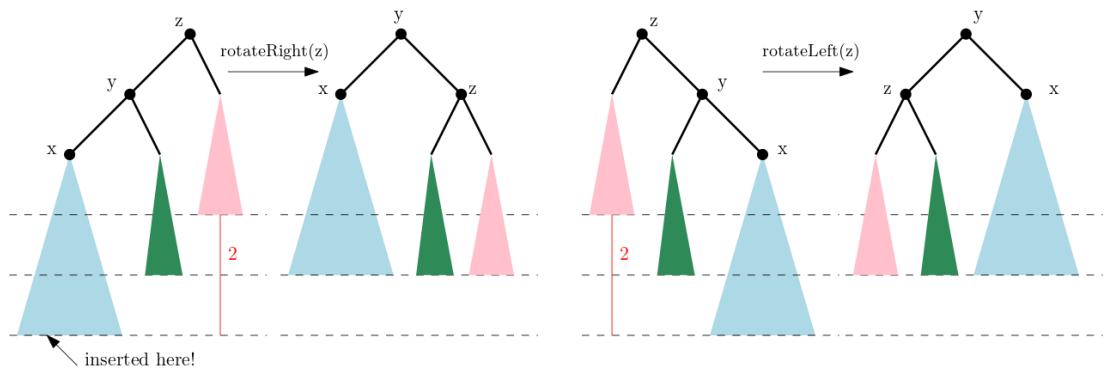
vii. 삽입: insert(key)

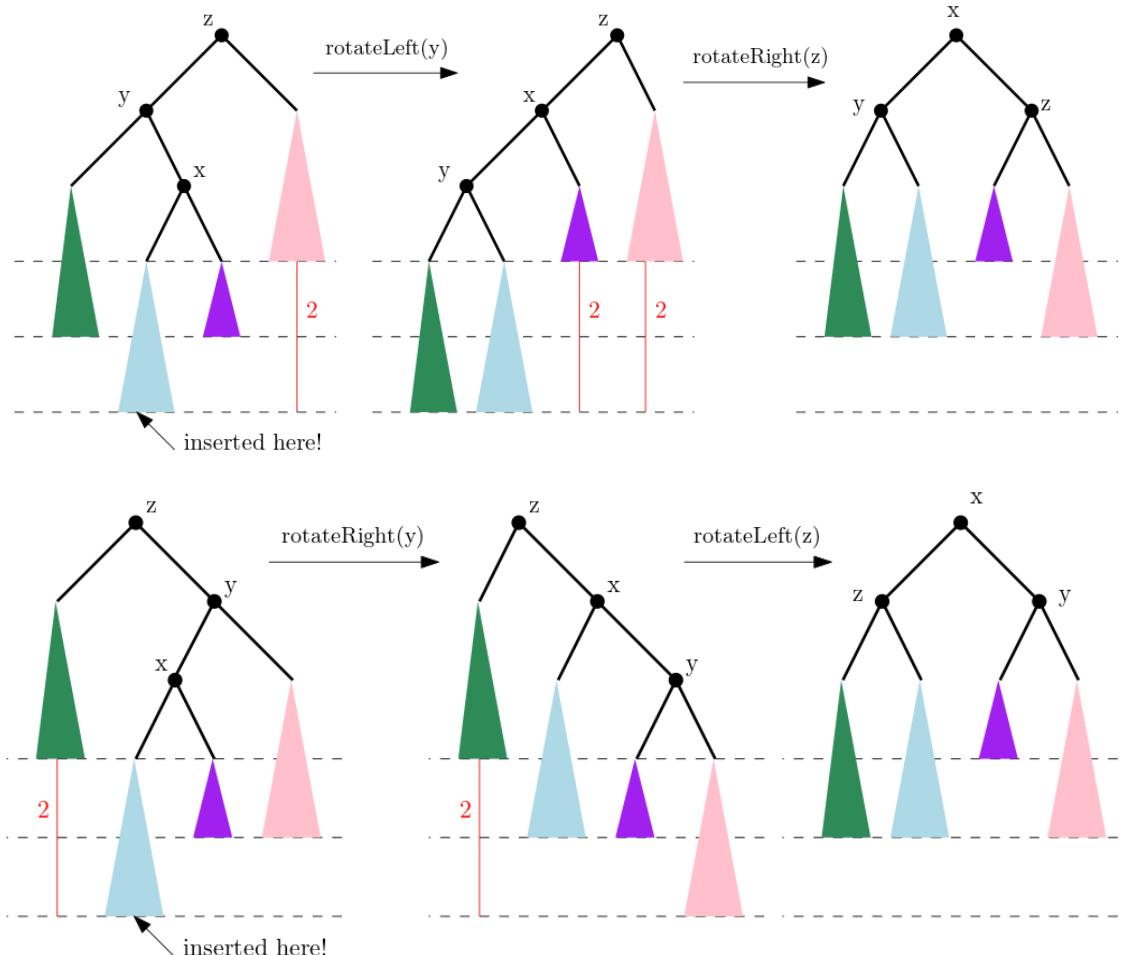
- [주의] BST 클래스의 insert 함수는 key 값을 갖는 새로운 노드 v를 리프 노드로 삽입한 후, v를 리턴함!
 - a. 부모(조상) 클래스의 같은 이름의 메소드를 호출하는 법:

```
v = super(AVL, self).insert(key)
```

super(현재_클래스_이름, self).method() 형식

- v로 인해 v의 조상노드의 균형이 깨질 수 있음 → **rebalancing** 필요!
 - a. BalanceFactor = -2 또는 2 인 조상 노드가 발생한다면, rotation을 통해 BalanceFactor를 조정해야함
- v에서 루트로 올라가면서 균형이 처음으로 깨진 노드를 z라 하자: (z의 BalanceFactor가 -2 또는 2라는 의미)
 - a. z → v 경로에서 z의 자식 노드를 y, y의 자식노드를 x라 하자
(x==v 일 수도 있음. x!=v라면, v는 x의 자손 노드 중 하나)
 - b. 균형을 맞추는 함수 rebalance(x, y, z)를 호출해 균형을 맞춤
- 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 수행함 (아래 그림)
- [?] 두 번보다 더 많은 rotation이 필요한 경우는 없을까?





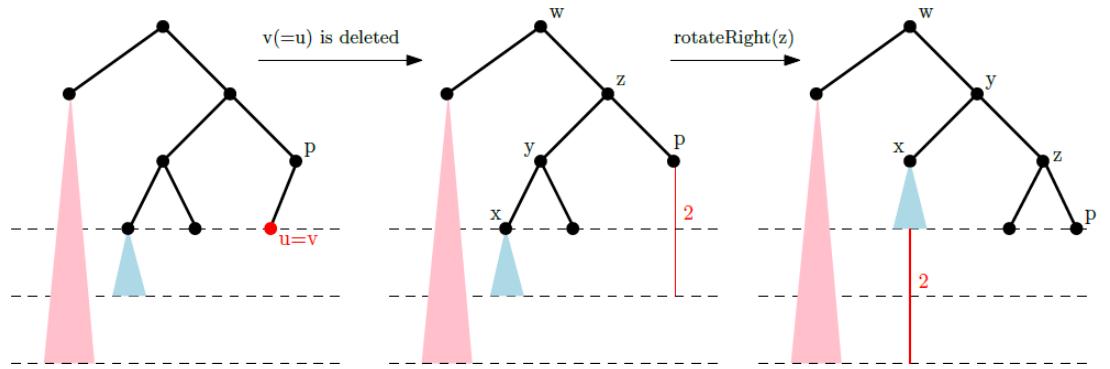
- [해보기-medium] rebalance(*x*, *y*, *z*)를 구현해보자!
 - rebalancing 한 후에 *z* 노드 위치의 노드 (top 노드)를 리턴하도록 구현해야 함 (경우에 따라서는 트리의 루트가 바뀔 수 있고, 새로운 루트를 알아야 기존의 루트를 update 할 수 있으므로)
- [해보기-medium] insert 함수의 pseudo code는?

```
def insert(self, key):
    # BST의 insert 함수는 실제 삽입된 노드가 리턴됨
    1. v = super(AVL, self).insert(key)
    2. find x, y, z # how?
    3. w = rebalance(x, y, z)
    4. if w.parent == None:
        self.root = w
```

viii. 삭제: delete(*u*)

- *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 *v*라 하자

- `deleteByMerging` 방법을 사용한다면, `u.left != None`이면 `m.right`에 `u.right`가 연결되기 때문에, `m`의 높이가 달라질 수 있다. 이 노드가 가장 깊은 곳의 노드이므로 `v = m`이 된다. 만약, `u.left == None`이면, `u.left`가 `u`를 대신하기 때문에, `v = u.left`로 해도 문제는 없다. (정확히 말하면, `u.left`의 높이는 변하지 않기 때문에, `v = u.parent`로 해야 한다)
 - a. `deletebyMerging` 함수에서 리턴 노드 `s`가 `v`임
- `deleteByCopying` 방법의 경우도 유사하게 높이에 영향을 받을 수 있는 가장 깊은 곳의 노드 `v`를 정하면 된다
- `insert` 경우처럼, `BalanceFactor = -2` 또는 `2`가 되는 `v`의 첫 조상 노드를 `z`라 하자. `z`에서 균형이 깨졌다라는 의미는, `u`가 삭제되면서 `z`의 부트리 중 `u`가 포함된 부트리의 높이가 1 감소 또는 증가하여 `BalanceFactor = -2` 또는 `2`가 되었다는 의미이다. 물론 `z = v`인 경우도 있을 수 있다!
- 균형을 맞추기 위해선, `v`가 속한 부트리의 높이를 증가시켜야 하고, 이를 위해선 `y`와 `x`를 `z`에서 `v`가 속하지 않은 (높이가 더 큰) 부트리에 속하는 노드(`y`는 `z`의 자식노드, `x`는 `y`의 자식노드)로 정의해야 한다 (이 부분이 `insert` 경우와 다른 점 중 하나)
 - 아래 그림을 예를 들어 설명
 - a. 가장 왼쪽 그림처럼 `v`가 삭제되면, 가운데 그림처럼 `z`의 `balanceFactor = -2`가 된다. 이를 조정하기 위해선 `z`의 왼쪽 자식노드를 `y`로, `y`의 자식 중 더 높이가 큰 부트리를 갖는 자식노드를 `x`로 정의해서, `z`에서 `rotateRight(z)`를 해야 한다 (회전한 결과가 가장 오른쪽 그림)
 - b. 회전을 통해 `z-y-x`의 균형을 맞췄지만, `y`의 새로운 부모 노드인 `w`에서 균형이 깨지게 된다. (이런 현상이 `insert`에서는 일어나지 않는다. [?] 왜?)
 - c. 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
 - d. 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:

```

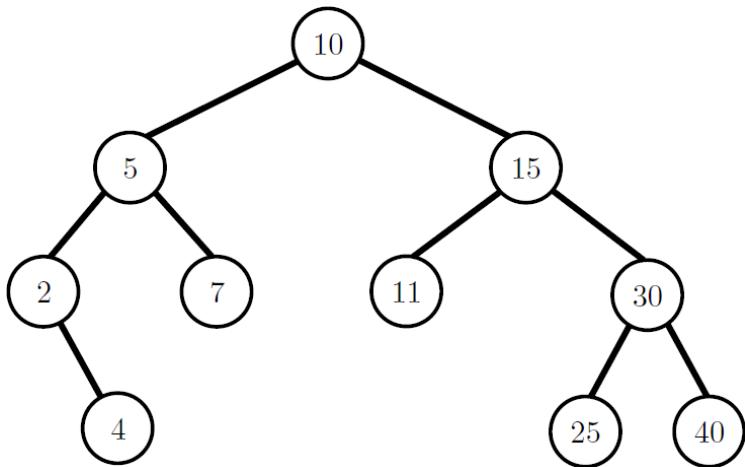
def delete(self, u):
    # deleteByMerging을 사용해도 됨
    # 노드의 삭제로 노드 높이에 영향을 받는
    # 첫 노드 v가 리턴된다 가정!
    v = super(AVL, self).deleteByCopying(u)
    while v != None:
        update v.height
        if v is not balanced: # z-y-x chain 존재!
            z = v
            if z.left.height >= z.right.height:
                y = z.left
            else:
                y = z.right
            if y.left.height >= y.right.height:
                x = y.left
            else:
                x = y.right
            v = rebalance(x, y, z)
            # rebalance는 rotation후 새로운 top 노드를 리턴
        w = v
        v = v.parent
    self.root = w # w could be a new root

```

- 자세한 삽입/삭제 알고리즘과 pseudo 코드는 [wikipedia](#) 등의 자료를 참조할 것

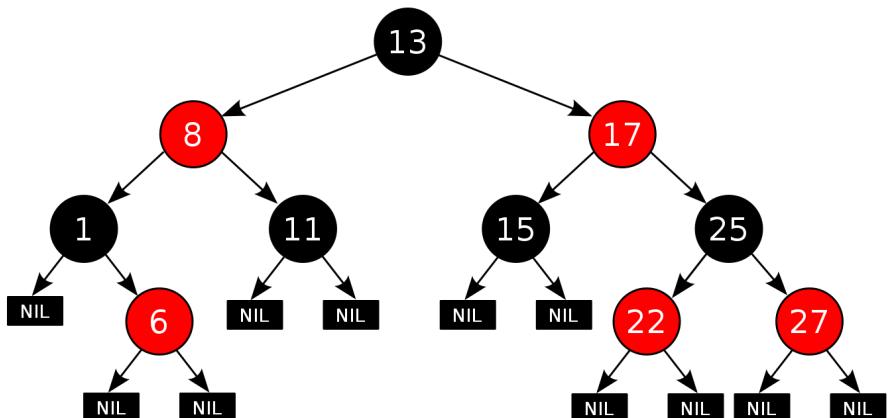
- ix. AVL 트리의 insert 경우의 회전은 최대 2번이면 항상 균형을 유지할 수 있고, delete 경우에는 최악의 경우에 루트 노드를 한 레벨씩 올라가면서 회전을 반복할 수도 있기에 **$O(\log n)$ 번의 회전이 가능**
- x. 결국, insert의 경우엔 key 값의 위치를 찾아야 하고, delete의 경우엔 (key 값을 갖는 노드의 탐색시간을 제외하더라도) rebalance을 위해 (최악의 경우에는) **$O(\log n)$ 번의 회전**할 수 있으므로 **$O(\log n)$ 시간이면 필요하다!**

- xi. [연습] 아래 AVL 트리에 `insert(27)`를 수행해본다. 다음으로 `insert(28)`를 수행해본다. 마지막으로 `delete(search(7))`을 수행해본다.



g. Red-Black 트리

- i. 가정: 리프노드의 두 자식노드인 None 노드는 NIL 노드라도 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 black의 색을 갖는다
 - 루트노드는 black이다
 - 리프노드인 NIL 노드의 색은 black으로 정의한다
 - 어떤 노드가 red라면, 두 자식노드는 모두 black이다
 - 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 black이고, red 노드의 두 자식노드는 모두 black임
 - 임의의 노드에서 서브트리의 각 NIL 노드까지의 경로에 포함된 black 노드 개수는 3으로 모두 같음
 - [사용 예]
 - a. C++의 STL(Standard Template Library)에서 사용하는 set, map, multiset, multimap은 red-black 트리로 구현됨
 - b. java의 Treemap, TreeSet 또한 red-black 트리로 구현됨



- iv. 1-5번 조건을 만족하는 경우, red-black 트리 T의 높이는 얼마일까? 답은 $O(\log n)$
 - $h(v)$: v의 높이 (또는 v의 서브트리 높이)
 - $bh(v)$: v에서 v의 서브트리의 NIL 노드까지의 경로에 포함된 black 노드 갯수 (black-height)
 - a. 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.

- a. $h(v)$ 에 관한 귀납법(induction)에 의해 증명
- b. $h(v) = 0$ 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
- c. $h(v) = 0$ 인 경우: 내부 노드가 없으므로 None 하나로 구성된 빈 트리가 되어, $bh(v) = 0$ 이 되어야 함
 - i. $2^{bh(v)} - 1 = 2^0 - 1 = 0$
- d. $h(v) \leq k$ 경우에는 v 의 자손 내부 노드의 최소 개수는 $2^{bh(v)} - 1$ 이하라고 가정함
- e. $h(v) = k+1$ 경우:
 - i. $h(v) > 0$ 이 성립함
 - ii. v 의 두 자식노드의 black height는 $bh(v)$ 이거나 $bh(v)-1$ 중 하나임. ([?] 왜?)
 - iii. v 의 두 자식 노드의 높이는 당연히 v 의 높이보다 작다 → 따라서 두 자식 노드에 대해서 가정이 성립한다
 - iv. 그럼 v 의 두 자식 서브트리에 포함된 대한 내부 노드 개수에 대한 최소 개수는

$$2^{bh(v)-1} - 1 + 2^{bh(v)-1} - 1 + 1 = 2^{bh(v)} - 1$$
이 되어 증명완료

- 사실 2: 루트에서 NIL까지의 임의의 경로에 포함된 black 노드 수는 경로에 포함된 노드 수의 반 이상이다
 - a. [?] 왜? [hint] Red-Black 트리의 4번째 성질 참조
- 사실 2로부터 루트노드 r 의 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 $O(\log n)$ 이 되어 [증명 끝]

v. insertion 전략 (자세한 코드는 생략)

- 새 노드 x 를 BST처럼 삽입한다
- x 의 색을 우선 **red**로 칠한다
- 4가지 경우로 나누어 처리한다

$p = x.parent$, $g = x.parent.parent$, $s = x.sibling$, $u = x.uncle$

경우 1: $x == T.root \rightarrow x.color = black$ (조건 1에 의해)

경우 2: $p.color == black \rightarrow$ do nothing!

경우 3: $p.color == red \rightarrow$

- 이 경우는 조건 4를 위반되기에 색 조정 필요!

- $s.color$ 는 무엇인가? 항상 **black**! (왜?)

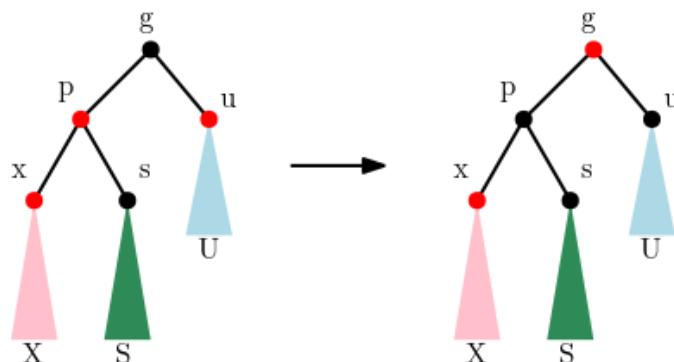
- $p.color == red$ 이므로 $g.color$ 는 항상 **black**!

경우 3.1: $u.color (u.color) == red \rightarrow$

- $g.color = red$ if $g != T.root$
- $p.color = u.color = black$

• X, S, U 의 노드들은 조건 5 변화 없음!

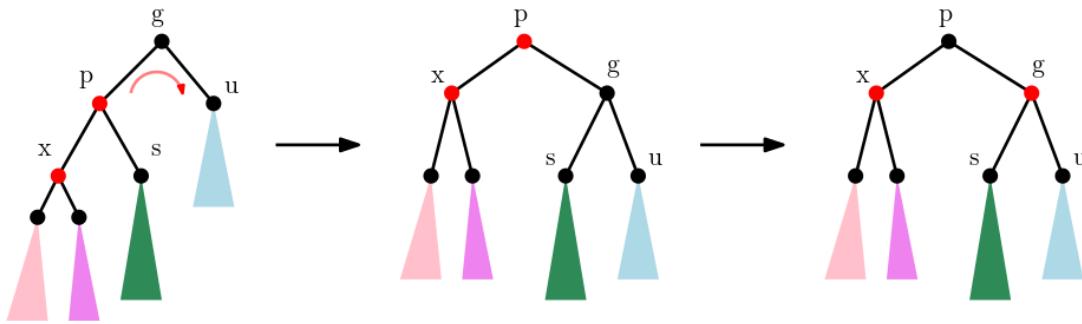
• g, p, u 역시 조건 5 만족함 (따져볼 것!)



경우 3.2: $u.color (u.color) == black \rightarrow$

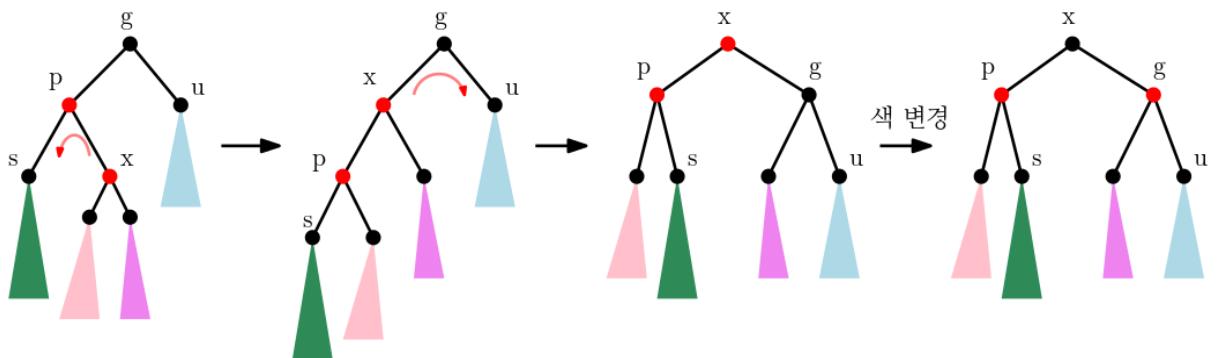
경우 3.2.1: $x - p - g$ 가 linear 모양인 경우

- rotate at g (한 번의 회전)
- $p.color = black$, $g.color = red$



경우 3.2.2: x - p - g가 triangle 모양인 경우

- rotate at p → rotate at g (두 번의 회전)
- x.color = **black**, g.color = **red**



vi. deletion (설명 생략)

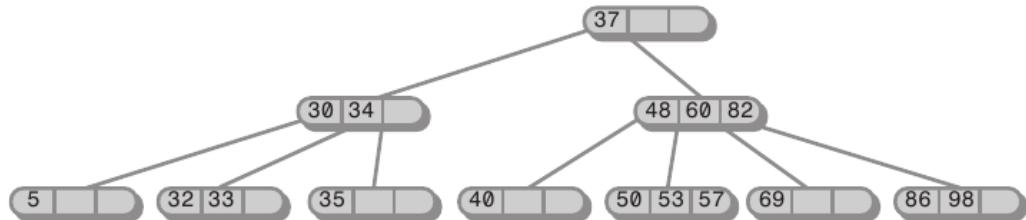
vii. 수행시간 및 필요 회전 횟수 정리

Operations	Worst Case	필요 회전 횟수
Space	$O(n)$	-
search	$O(\log n)$	0
insert	$O(\log n)$	2 (2 <- AVL)
delete	$O(\log n)$	3 ($\log n <- AVL$)

- AVL 트리와의 차이점은 delete 연산에 필요한 회전 횟수!

h. 2-3-4 트리 (red-black 트리의 쌍등이 트리)

- i. 2-3-4트리는 (1) 자식노드가 **최소 2개, 최대 4개** 이하이며, (2) 모든 리프노드는 마지막 레벨에 위치해야 하는 탐색트리로 정의된다
 - 단, 이진트리는 아니다
 - 2-노드, 3-노드, 4-노드로 구분 (자식 노드의 개수에 따라 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:

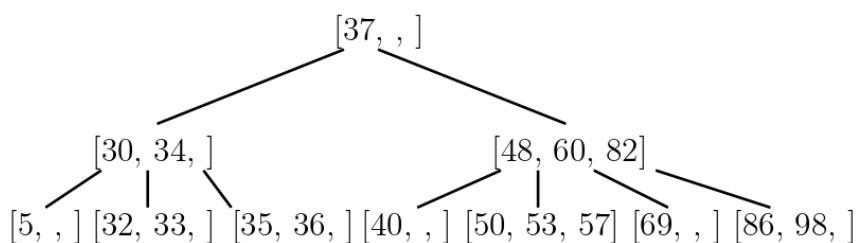
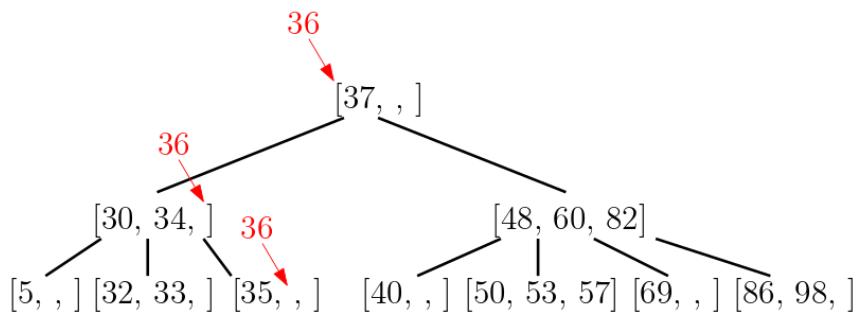
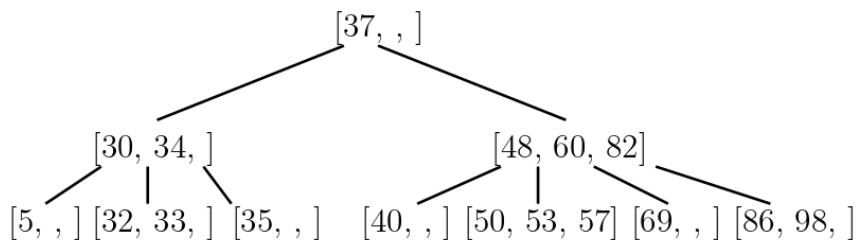
- 각 노드는 최대 4개의 자식 노드를 가질 수 있다.
- 위의 예에서는 [48, 60, 82] 노드가 네 개의 자식 노드를 가지고 있다. 이 노드에는 3개의 key가 정렬된 상태로 저장되어 있다. 첫 번째 자식 노드에는 48보다 작은 값이, 두 번째 자식 노드에는 48 이상 60 미만의 값들이, 세 번째 자식 노드에는 60 이상 82 미만의 값들이, 네 번째 자식 노드에는 82 이상의 값들이 저장되어 있다는 의미이다.
- 어떤 key 값을 탐색한다는 것은 루트 노드부터 해당 key 값이 속해 있을 자식 노드를 따라 내려가기만 하면 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:

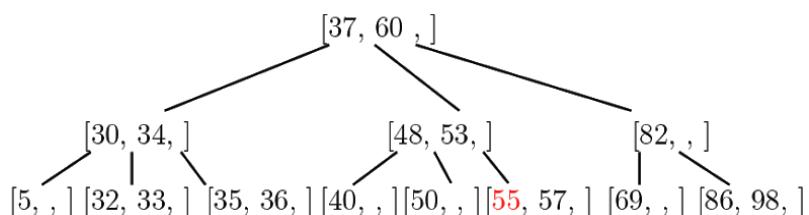
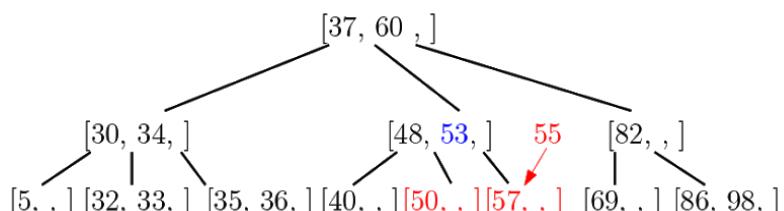
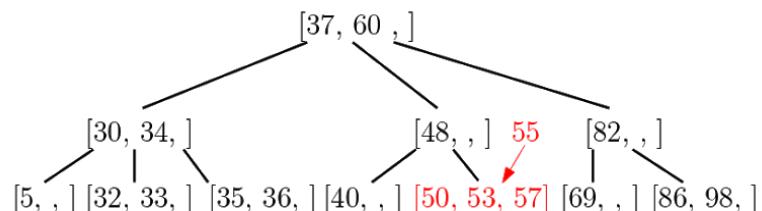
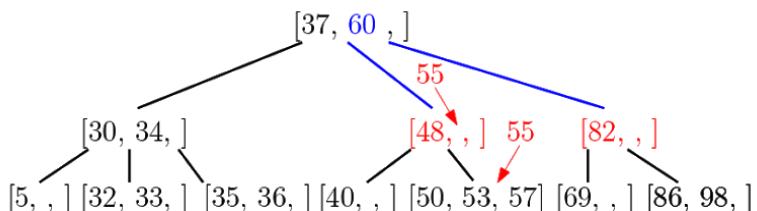
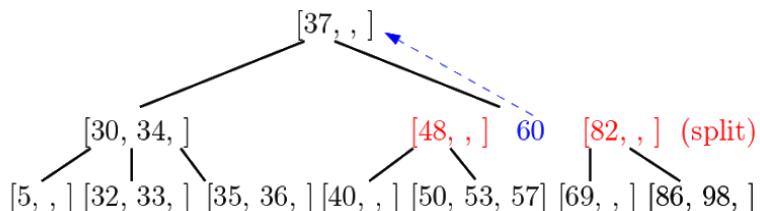
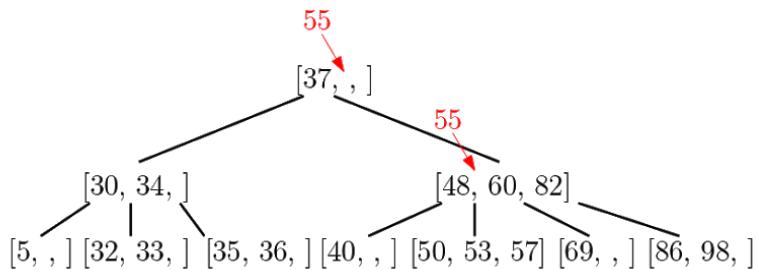
- 루트부터 시작해 삽입할 리프 노드까지 다음과 같은 방법으로 삽입될 리프 노드를 탐색한다 (**항상 리프 노드에 삽입됨**)
- 현재 방문 중인 노드 v가 4-노드 $[a, b, c]$ 라면 두 2-노드 $[a]$ 와 $[c]$ 로 "split"한다
 - 가운데 값 b 는 v 의 부모 노드에 삽입한다. v 의 부모노드는 b 를 위한 공간이 항상 있는가? (왜?)

- b. 만약 v 가 루트 노드였다면, b 만으로 이루어진 새로운 루트 노드 (2-노드 [b])를 만든다 (\leftarrow 높이가 1 증가되는 효과)
- c. 그 다음 자식 노드를 선택해 한 레벨 내려가 탐색을 계속한다
- 현재 방문 중인 노드 v 가 4-노드가 아닌 경우에는:
 - a. 리프 노드이면 key 값을 순서에 맞게 삽입하고 끝! (4-노드가 아니므로 삽입할 여유가 항상 있음에 유의)
 - b. 리프 노드 아니라면 다른 레벨의 자식 노드를 선택해 탐색을 계속한다
- 위의 그림에서

- a. 36을 삽입해보자



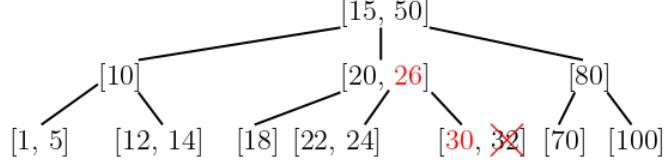
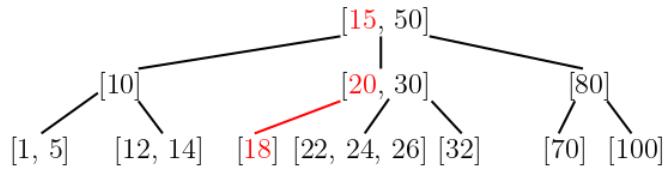
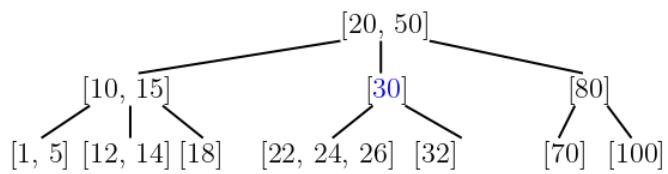
- 삽입될 리프 노드까지 내려오면서 만나는 노드 중에 4-노드가 없음. 따라서 split할 필요가 없었음!
- 결국 리프 노드 [35, ,]도 4-노드가 아니므로 36을 저장할 여유가 있음. 최종적으로 [35, 36,]의 형태로 저장됨
- 이제, 55를 삽입해보자.



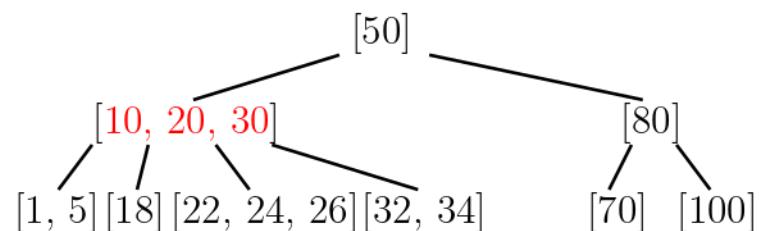
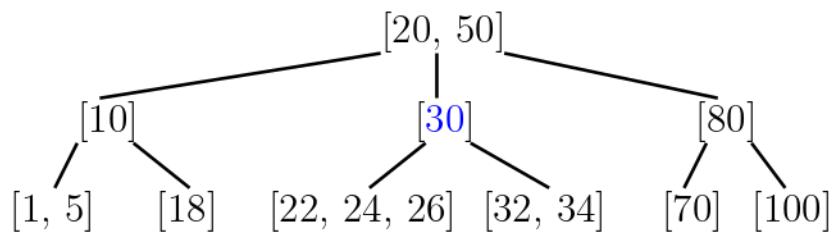
v. **delete** 연산:

- insert보다 복잡 (이유: 삽입은 항상 리프 노드에서 일어나지만, 삭제는 내부 노드의 key 값이 삭제될 때도 있어서)

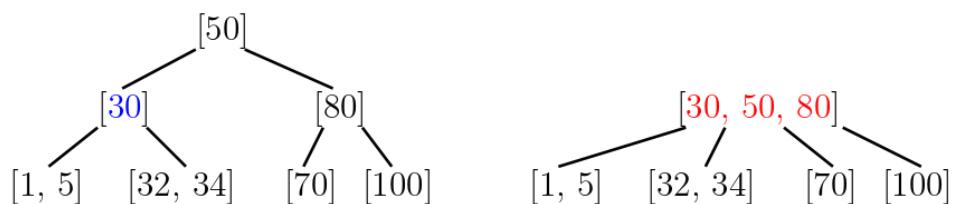
- key 값이 내부노드에 있는 경우
 - a. successor(key) 또는 predecessor(key)를 찾아 swap → 리프노드에 있는 값을 지우는 경우로 변경
- key 값이 리프노드에 있는 경우
 - a. 3-노드 또는 4-노드인 경우: 단순 삭제 가능
 - b. 2-노드인 경우: 지우면 underflow 발생. 이를 피하기 위해 루트 노드에서 리프 노드로 내려가면서 2-노드를 만나면 무조건 3-노드 또는 4-노드로 변경함 (단, 루트가 2-노드인 경우는 제외)
 - i. 그 과정에서 트리의 높이가 1 감소할 수 있음
 - ii. 결국, 삭제할 key 값이 저장된 리프 노드는 3-노드/4-노드가 되어 단순 삭제 가능
- 2-노드를 3-노드/4-노드로 변경
 - a. **Rotation:** 왼쪽 또는 오른쪽 3-노드/4-노드 형제에게서 값을 하나 빌려옴! (물리적인 rotation이 아니라, 값이 회전하듯 연쇄적으로 이동함) 그래서 3-노드가 됨!
 - b. 아래 그림에서 32를 제거한다고 하자. 2-노드 [30]에서 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)
 - c. [32] 리프노드에 도착해서도 같은 방식으로 값의 회전이 발생함: [32] → [30, 32]가 되어 32 제거 가능



- d. **Fusion:** 왼쪽, 오른쪽 형제가 모두 2-노드라면? 자신의 key 값과 왼쪽 (또는 오른쪽) 형제 노드 값 + 부모 노드의 값 하나를 fusion(merge)해서 하나의 4-노드로 만듬 (Q: 만약 부모가 루트고 2-노드라면? → shrink!)



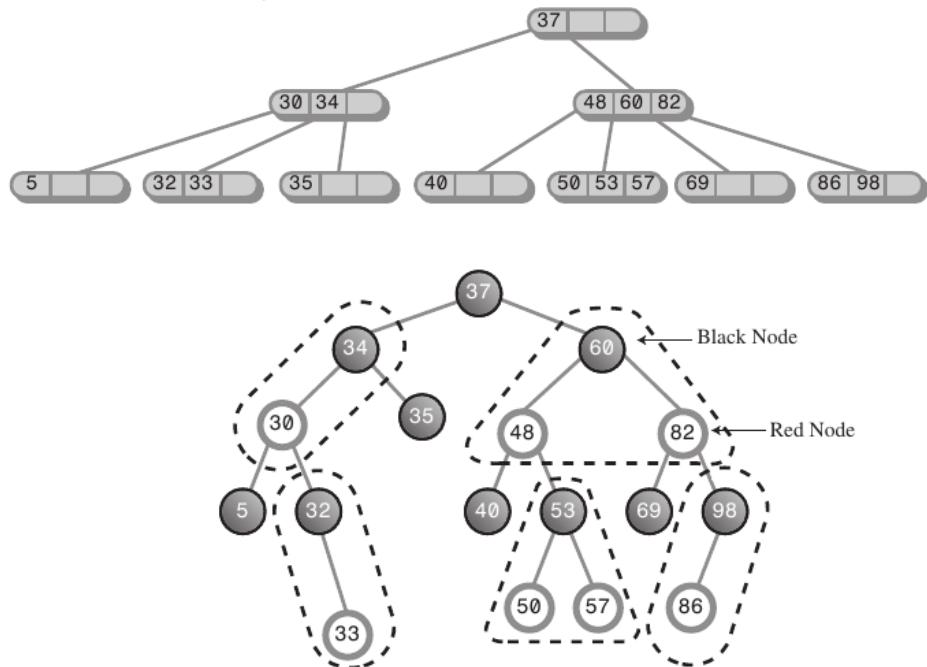
i. **Shrink:** 루트가 없어지고 높이가 1이 줄어드는 경우



- 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
 - a. [hint] 3-노드와 4-노드는 두 레벨에 걸쳐 두 개 또는 세 개의 red-black 노드로 변환! 노드 색 배정에 주의!
- 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!
 - a. [hint] 2-3-4 트리가 어떤 노드로 구성되는 게 h' 을 가장 크게 하는 경우일까?



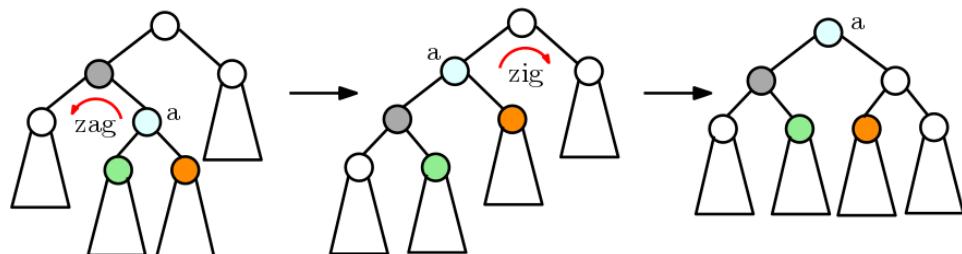
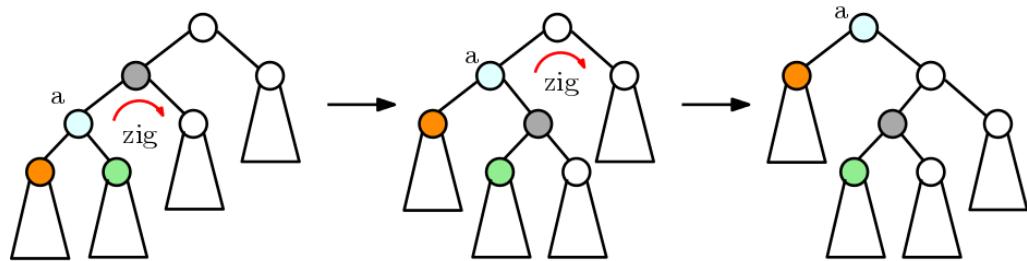
viii. red-black 트리가 2-3-4 트리보다 훨씬 광범위하게 사용된다. 이유는?

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)
 - a. zig/zag (a 의 부모가 루트인 경우)
 - b. zig-zig/zag-zag (a 와 a 의 부모가 각각 같은 방향 자식: linear 모양)
 - c. zig-zag/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. **Pseudo 코드:**
Splay 트리 클래스는 이진탐색트리를 부모 클래스로 지정해보자

```

class splayTree(BST):# BST becomes a parent class
    # BST의 모든 것(초기화함수포함)을 물려 받음
def splay(self, v):
    # make v a root by rotations
    ...
    return v    # return the root after splaying

def search(self, int key):
    # [REDACTED]
    v = super(splayTree, self).search(key)
    if v: # splay v
        self.root = self.splay(v)
    return v

def insert(self, key) {
    # [REDACTED]
    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 아님, 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 삼는다
            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!
-

[Summary] 현재까지 살펴본 자료구조들의 수행시간 비교 표

[출처: <http://bigocheatsheet.com/>]

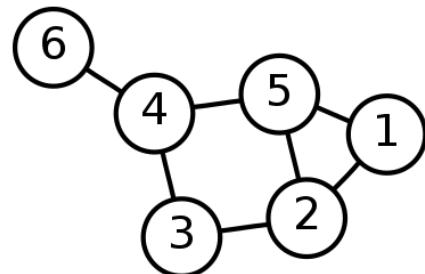
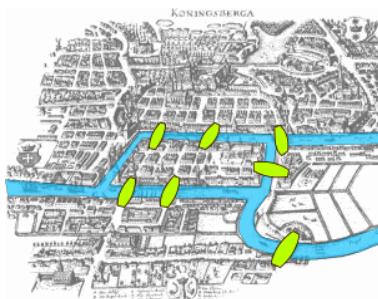
Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$	
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	

F: Graph 자료구조 및 기초 그래프 알고리즘

1. 두 노드 사이의 관계가 있는 경우 에지로 연결하여 표현하는 추상적이고 일반적인 자료구조

- a. 트리는 사이클이 없는 그래프!
- b. 연결리스트는 하나의 경로로 이루어진 트리이므로, 가장 단순한 형태의 그래프!



2. 그래프 $G = (V, E)$

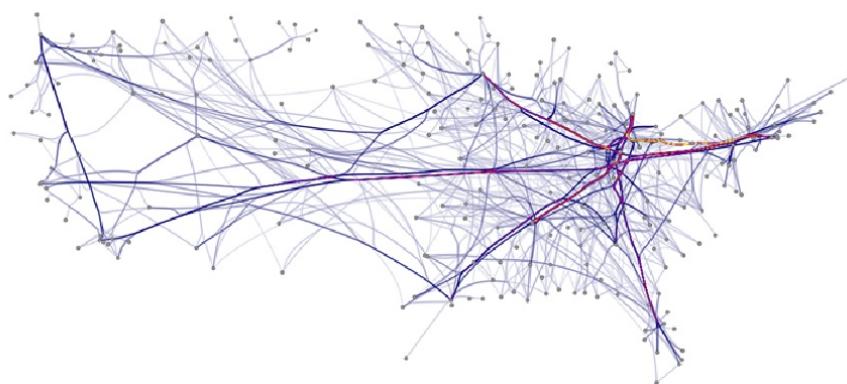
- a. V = 노드(node) 또는 정점(vertex) 집합
- b. E = 두 노드 쌍으로 정의. 만약 $(u, v) \in E$ 라면 노드 u 와 v 가 서로 (방향이 없는) 에지로 연결되어 있다고 한다
- c. 왼쪽 그림 예: (from Wikipedia/Graph)

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

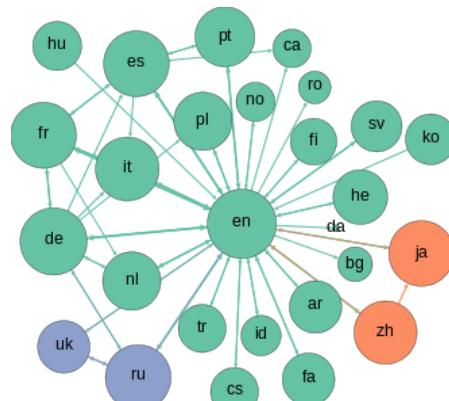
- d. 오른쪽 그림 예: $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1,2), (3,2), (1,5), (2,5), (4, 5), (6, 4), (3,4)\}$

3. 응용분야(applications)

- a. 주위의 거의 모든 관계를 그래프로 표현 가능하기에 무궁무진하다
- b. Computer Science, Linguistics, Physics and Chemistry, Biology, Social Science



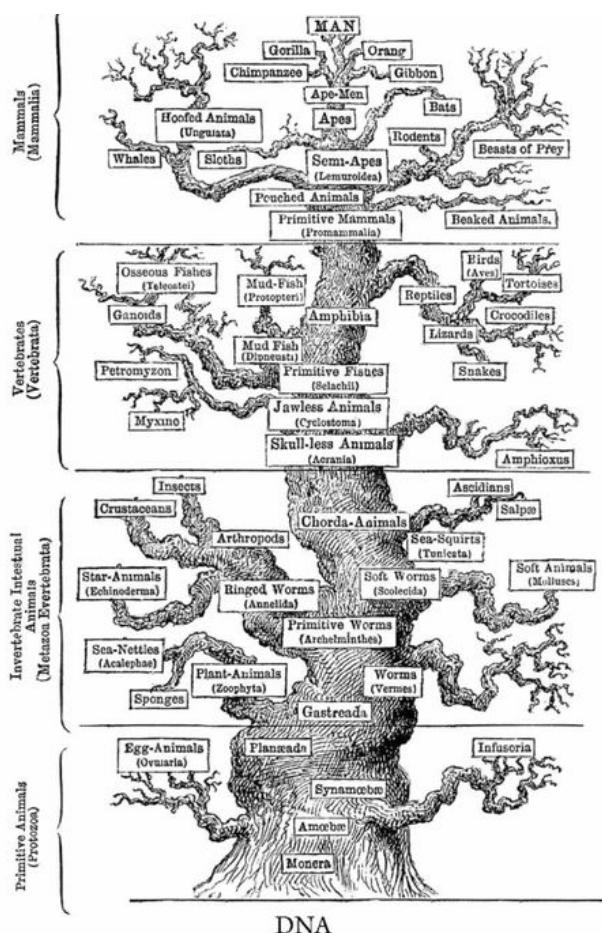
[US domestic-flights map from [Graphminator](#)]



[graph on web sites from Wikipedia]

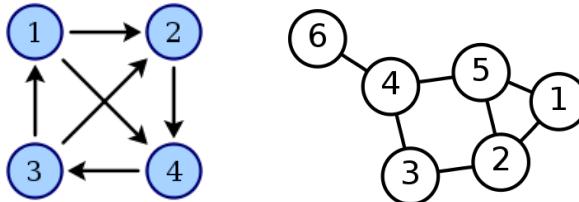


[subway map]

[Phylogenetic tree: [출처](#)]

4. 기본 용어 (basic graph terminology)

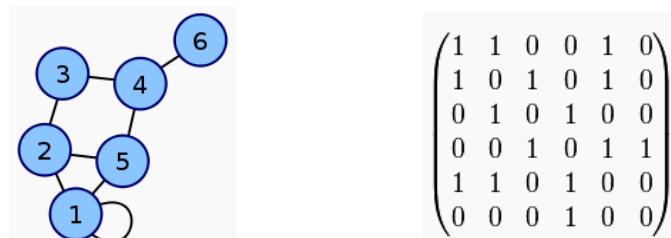
- a. 정점(vertex), 노드(node)
- b. 에지(edge), 링크(link)
 - i. directed/undirected edge
- c. 무방향 그래프, 방향 그래프

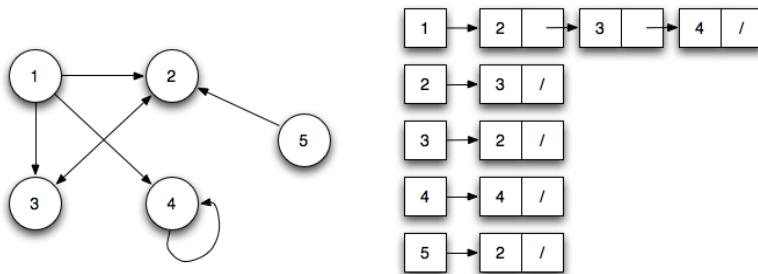


- d. 분지수(degree)
 - i. in-degree, out-degree, degree of vertex, degree of graph
 - ii. $\deg(u)$, $\text{in-deg}(u)$, $\text{out-deg}(u)$
- e. 인접성
 - i. 에지 (u, v) 가 존재하면, u 와 v 는 서로 인접(adjacent)
 - ii. 에지 $e = (u, v)$ 에 대해, e 는 u 와 v 에 인접(incident)
- f. 에지/정점 가중치(weight/cost)
- g. 경로(path)
 - i. 단순 경로(simple path)
 - ii. 경로의 길이 (에지에 가중치 값이 없는 경우/있는 경우)
- h. 사이클(cycle)
 - i. 트리(tree): 사이클이 없는 연결 그래프
 - ii. 포리스트(forest): 연결 트리의 컬렉션(집합)
- i. 연결성(connectedness)
- j. 부그래프(subgraph)
- k. 신장그래프(spanning subgraph)

5. 그래프 표현 방법 (graph representation)

- a. 인접행렬(adjacency matrix): 인접성을 행렬(2차원 배열/리스트)로 표현
 - i. 장단점:
- b. 인접리스트(adjacency list): 각 정점에 인접한 에지만을 연결리스트로 표현
 - i. 장단점:





c. 표현법에 따른 기본 연산 시간 비교

i. $V = \text{정점 개수}, E = \text{에지 개수}$

ii. Python 리스트

1. 방향그래프 예: $G = [[1, 3], [0, 2, 3], [], [4, 2], [3]]$

기본연산	인접행렬	인접리스트
(u, v) 가 에지인가?	$G[u][v] == 1$ $O(1)$	$G[u].search(v) != \text{None}$ $O(V)$
u 의 인접한 모든 에지 (u, v) 에 대해:	$G[u][v] \text{ for } v \text{ in}$ $\text{range}(n)$ $O(V)$	$\text{for edge in } G[u]$ $O(\text{deg}(u))/O(\text{out-deg}(u))$
새 에지 (u, v) 삽입	$G[u][v] \leftarrow 1$ $O(1)$	$G[u].pushFront(v)$ $O(1)$
에지 (u, v) 제거	$G[u][v] \leftarrow 0$ $O(1)$	$x \leftarrow G[u].search(v)$ $G[u].remove(x)$ $O(\text{deg}(u))/O(\text{out-deg}(u))$
G 를 위한 메모리	$O(V^2)$	$O(E)$

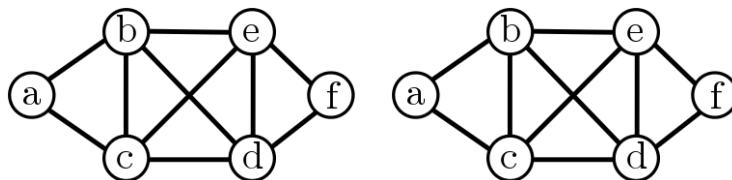
6. Traversal: DFS, BFS

a. DFS(Depth First Search: 깊이 우선 방문)

- i. 현재 방문 노드에서 방문하지 않은 이웃 노드가 있다면 방문하는 방식

```
RecursiveDFS(v):
    mark v as visited node
    for each edge (v, w):
        if w is unmarked:
            RecursiveDFS(w)
```

```
IterativeDFS(s):
    stack.push(s)
    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: _____

- iii. 첫 방문 시간과 최종 방문 시간 기록하기 + 방문 순서 (부모 노드) 기록하기

```
DFS(v):
    mark v as visited node
    pre[v] = curr_time # record the first visit time
    curr_time += 1
    for each edge (v, w):
        if w is unmarked:
            parent[w] = v
            DFS(w) # Recursive version
    post[v] = curr_time # record the finish time
    curr_time += 1
```

```
DFSAll(G):
    for all nodes v:
        unmark v
    for all nodes v:
        if v is unmarked:
            DFS(v)
```

비재귀 코드

```

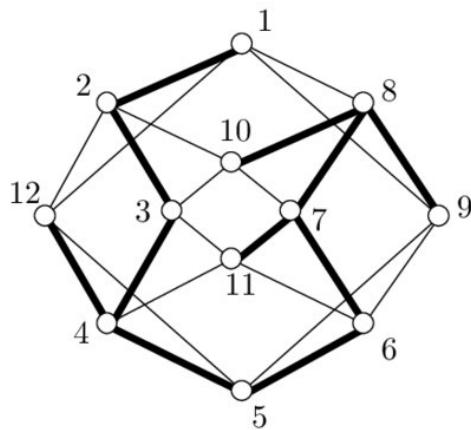
DFS(s):
    stack.push((None, s)) # tuple (parent, curr_node)
    while stack is not empty:
        p, v ← stack.pop()
        if v is unmarked:
            mark v as visited node
            parent[v] = p
            for each edge (v, w):
                if w is unmarked:
                    stack.push((v, w))

```

iv. DFS 트리

1. 신장트리
2. DFS 트리의 나타난 에지 (u, v) 의 $\text{pre}[u]$, $\text{pre}[v]$, $\text{post}[u]$, $\text{post}[v]$ 비교

v. 예제



vi. 응용

1. 연결 성분(connected component)을 구분해서 방문하기
 - o count 변수 사용, $\text{DFS}(v, \text{count})$, $\text{comp}[v] = \text{count}$ 기록
2. 미로 탈출: 이 경우엔 동, 서, 남, 북으로 모두 이동 가능!
 - o DFS는 backtracking 방법과 동일하다!!
 - o 미로는 이차원 리스트 M 에 저장되어 있음
 - o 입력 형식:
 - i. '1'은 장애물, '0'은 빈 칸 의미
 - ii. 바깥쪽 경계는 모두 '1'로 하여 외부로 나가지 못함
 - iii. 7×7 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 = 입구 표시, f = 출구 표시, *는 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 등쪽 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

M = 미로 입력

visited = 초기화

find_way_from_maze(1, 1) # 호출

vii. 사이클 찾기

1. 무방향 그래프에서는?

2. 방향 그래프에서는?

- 사이클이 없는 방향 그래프를 DAG(Directed Acyclic Graph)라 부른다
- DAG라면, outgoing 에지만 있는 노드와 incoming 에지만 있는 노드가 반드시 하나 이상 존재해야 한다. outgoing 에지만 있는 노드를 source 노드라 부르고, incoming 에지만 있는 노드를 sink 노드라 부른다

3. 주어진 그래프 G가 DAG인지 검사하는 알고리즘은?

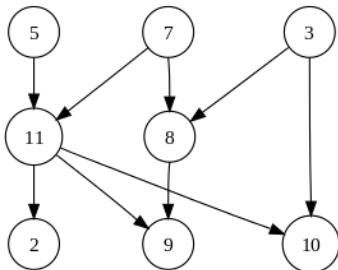
- **힌트:** DFS에서 stack에 있는 모든 노드들은 DFS 트리에서 현재 방문 중인 노드의 조상 노드이다 \Rightarrow stack의 노드를 차례로 방문해서 현재 방문 중인 노드까지의 경로가 존재한다
- 만약 현재 노드에서 다음 이웃 노드가 이미 방문한 노드이면서 stack에 들어 있는 노드라면 사이클이 존재한다는 의미이다!
- 우선 source 노드가 여러 개일 수 있으므로 알고리즘을 단순하게 만들기 위해 추가로 source 노드 s를 삽입하고, s에서 G의 모든 노드로 에지를 추가한다. 이렇게 정의한 그래프를 G'이라 하면, G가 DAG라면 G'도 DAG가 되며, 그 역도 성립함을 알 수 있다.
- 이제, G 대신 G'에 사이클이 있는지 검사한다
 - i. 각 노드의 상태(status)를 NEW, IN_STACK, DONE 세 가지로 나눈다 (IN_STACK은 해당 노드를 지나는 DFS를 아직 모두 끝내지 못했다는 의미로 여전히 recursion stack에 남아 있다는 것이다)
 - ii. 우선 모든 노드의 상태를 NEW로 초기화한다

```

isDAG(v): # call isDAG(s) initially
    status[v] = IN_STACK
    for each edge v → w:
        if status[w] == IN_STACK:
            return False
        elif status[w] == NEW:
            if isDAG(w) == False:
                return False
            status[v] = DONE
    return True
  
```

viii. DAG의 노드들을 순서에 따라 정렬해보자

1. 이 정렬을 Topological Sort 또는 Topological Ordering이라 부른다
2. source 노드가 가장 처음에 sink 노드가 가장 나중에 배열된다



5, 3, 7, 8, 11, 2, 9, 10

3, 5, 7, 8, 11, 9, 10, 2

...

3. Algorithm 1:

```

TopologicalSort(G):
    S = []
    for i in range(|V|):
        v ← any source node in G # how to find v?
        S.append(v)
        delete all outgoing edge from v
    return S
  
```

단점:

4. Algorithm 2:

- **힌트:** isDAG의 DFS에서 가장 처음에 DONE 상태인 노드는 무조건 sink 노드이다! → DONE의 가장 나중에 되는 노드부터 순서대로 나열하면 된다!

증명: 귀류법

TopologicalSort(G):

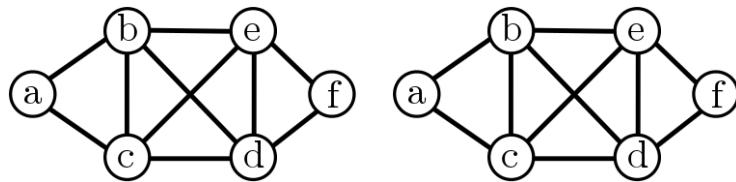
```

add a new source s with edge to all vertices
isDAG(s) # DONE 상태가 된 노드를 stack에 push함
t_order = []
for i in range(n):
    t_order.append(stack.pop())
print(t_order)
  
```

5. 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: a 노드부터 출발한다면, $a \rightarrow b$, $c \rightarrow e$, $d \rightarrow e$ 순으로 출발 노드부터의 거리 순으로 노드들을 방문하게 된다
- 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

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 source node s in G:
    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

```

- iv. 수행시간: _____

7. 최단 경로 문제(Shortest Path Problem)

- a. 일상생활에서 자주 경험하는 문제로, 문제 자체의 역사도 깊고 응용 가치도 커 많은 연구가 이루어진 문제
- b. 에지에 가중치(비용)가 주어진 방향 그래프를 대상으로 한다
 - i. 단, 가중치는 모두 양수라고 가정한다
- 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 Problem이라 부른다)
- d. 최단 경로의 기본 성질:
 - i. $u \rightarrow v$: 노드 u 에서 노드 v 로의 하나의 에지로 구성된 경로
 - ii. $s \rightsquigarrow v$: 노드 s 에서 v 로의 경로를 표시하고, 중간에 여러 노드가 존재 가능
 - iii. $s \rightsquigarrow u \rightarrow v$ 인 경우, 이 경로에서 u 는 v 의 predecessor 또는 parent이다
 - iv. [중요] 만약 $s \rightsquigarrow u \rightarrow v$ 가 s 에서 v 로의 최단 경로라면, $s \rightsquigarrow u$ 역시 s 에서 u 까지의 최단 경로가 된다
 1. s 에서 v 까지의 여러 경로 중에서 u 를 통해 v 에 도달하는 경로 중에 최단 경로가 있다는 의미이고,
 2. $s \rightsquigarrow u$ 를 먼저 계산하여 알고 있다면, $s \rightsquigarrow u \rightarrow v$ 는 쉽게 계산할 수 있다
 3. $\text{dist}[v] = s$ 에서 v 까지의 최단 경로의 길이로 정의하면,

$\text{dist}[v] = \min_{\text{each } u \mid u \rightarrow v} \{ \text{dist}[u] + \text{weight}(u \rightarrow v) \}$
 - v. 어라? 이건 DP 식인데... Yes!
 6. 그러면, $\text{dist}[v]$ 계산 전에 $\text{dist}[u]$ 가 먼저 계산되면 된다. 즉, v 의 predecessor u 를 먼저 계산하면 된다. 이런 논리를 계속 적용해나가면, s 에 인접한 노드까지 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. $\text{parent}[v] = (s \text{에서 } v \text{까지의 최단 경로에서 } v \text{ 바로 직전 노드를 저장})$
 - 최단 경로를 재구성하기 위해 parent 값 필요

v. 위의 DP 식을 함수 `relax`란 이름의 함수로 정리하면:

```
def relax(u → v):
    if  $\text{dist}[v] > \text{dist}[u] + \text{weight}(u \rightarrow v)$ :
         $\text{dist}[v] = \text{dist}[u] + \text{weight}(u \rightarrow v)$ 
```

e. 다음을 실행하면 어떻게 될까?

```
for i in range(n):
    for each edge u → v in G:
        relax(u, v)
```

1. 위의 알고리즘을 Bellman-Ford 알고리즘이라 부른다
2. 수행시간은 _____



(Bellman-Ford 알고리즘 설명 [youtube link](#))

f. 다음을 실행하면 어떤 걸 알 수 있을까?

```
for each edge s → u:
    relax(s, u)
```

1. s에 인접한 노드 u 중에서 $dist[u]$ 값이 가장 작은 u는 최단 경로의 길이가 $dist[u]$ 인가? _____ 왜?
2. u에 인접한 노드들에 대해서 다시 relax를 반복하면, $s \rightarrow u \rightarrow v$ 로 연결되는 최소 경로의 길이가 $dist[v]$ 에 저장된다

ii. 이 과정을 정리하면 다음 코드와 같다

```
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()
    for each edge u → v:
        relax(u, v)
        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 쌍을 알고 있어야 한다
        # 그래서 적응형 힙(Adapted Heap)을 쓰면 편리 (힙 자료구조 편 참조)

return dist, parent
```

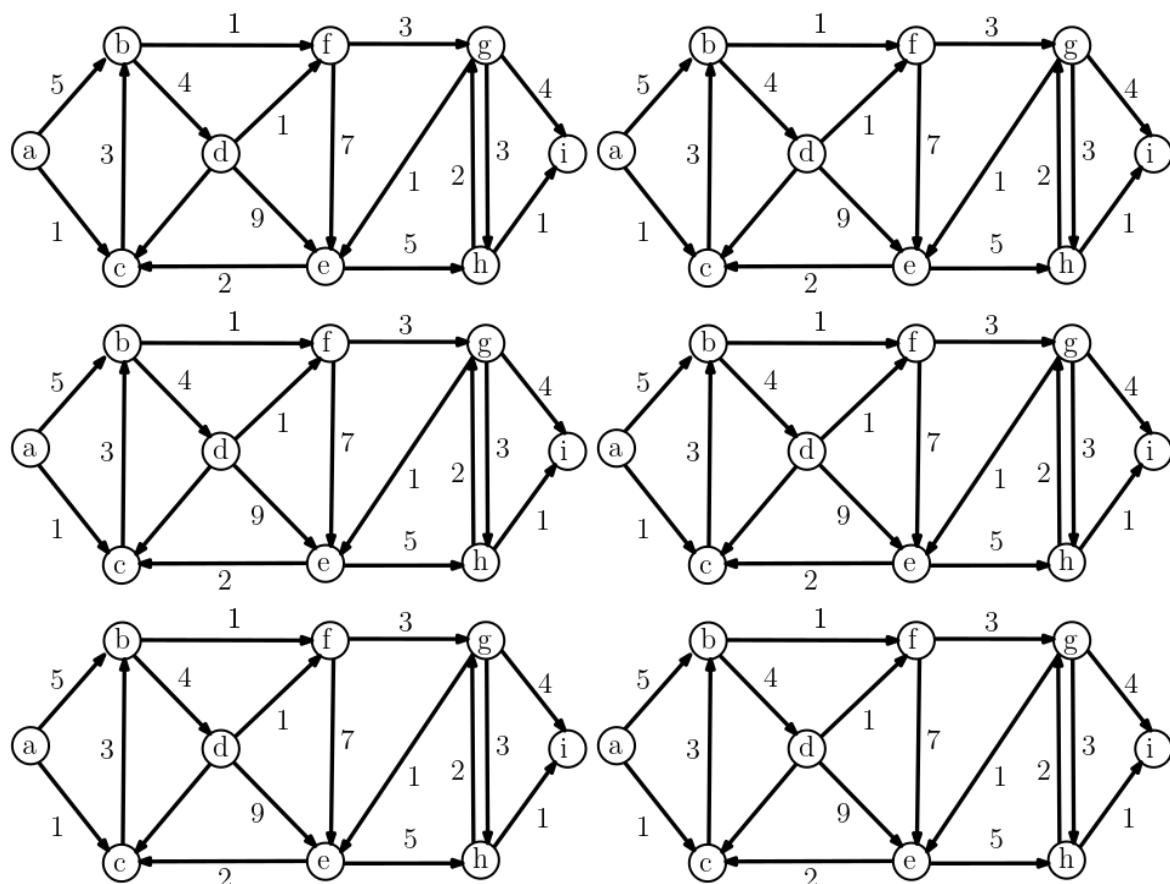
- g. 이 유명한 알고리즘을 **Dijkstra** 최단 경로 알고리즘이라 부른다. Dijkstra는 다익스트라로 발음한다



(Dijkstra 알고리즘 설명 [youtube link](#))



(Dijkstra 알고리즘 예제 설명 [youtube link](#))



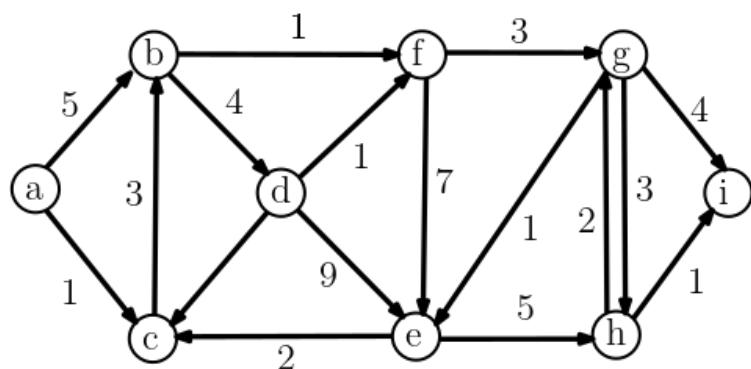
h. 구현이슈:

- i. 두 종류 힙 중 하나를 사용한다. 이 경우엔 연산의 수행시간이 다르다
(1) **binary heap** (2) **fibonacci heap** [고급 자료구조]

1. insert
2. delete_min
3. decrease_key

- ii. 수행시간: 힙의 종류에 따라 다르다!
-
-

i. 최단경로트리(Shortest Path Tree)



j. 만약, 가중치가 음수인 에지가 있는 경우라면?

- i. Bellman-Ford 알고리즘은 여전히 올바르게 동작하는가?
- ii. Dijkstra 알고리즘이 동작하지 않는 예를 찾아보자

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 그래프인 경우:
 - ii. 방향 그래프인 경우:
- b. [난이도 높음] 두 노드 s 와 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, s 에서 최단 경로의 에지를 차례대로 따라 가다가 어떤 노드 u 에서 다음 노드 v 를 따라가지 않고 u 에서 w 로 다른 에지를 따라 가게 된다. w 에서 목적 노드 t 까지 $w-t$ 최단 경로를 따라 가야 한다. (왜?) 여기서 $u = s$ 일 수도 있음에 유의하자.
 - iv. 즉, 두 번째로 빠른 경로는 s 에서 u 까지 최단경로는 그대로 따라가고, u 에서 v 가 아닌 u 에서 w 로 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 t 로 가는 어떤 경로도 최단 경로의 길이와 같거나 더 커야 한다. 우리는 두 번째로 빠른 경로를 구하는 것이기에 w 에서 t 까지는 최단 경로로 가야 한다. 결국, 두 번째로 빠른 경로는

(s 에서 u 까지의 최단 경로) + ($u \rightarrow w$) + (w 에서 t 까지의 최단 경로)

의 모양이 될 것이다. 결국, s 에서 t 까지의 최단 경로 상에 있는 모든 u 에 대해, 최단 경로 상의 에지가 아닌 다른 에지를 이용해서 t 에 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?

G: Other Data Structures [Advanced Topics: skip 가능]

- Union-Find 자료구조
- van Emde Boas 자료구조
- Suffix array 자료구조
- Range tree 자료구조

1. Union-Find 자료구조

- a. 여러 개의 집합을 관리하는 자료구조로 파이썬의 `set` 자료구조와 유사
- b. 지원 연산
 - i. `make_set(x)`: x 를 원소로 하는 새로운 집합을 생성
 - ii. `union(x, y)`: x 가 속한 집합과 y 가 속한 집합이 다르면 `union` (합집합)
 - iii. `find(x)`: x 를 포함한 집합의 대표 원소를 리턴
- c. 가장 단순한 방법은 하나의 집합을 이중연결리스트로 관리하는 방법
 - i. `make_set(x)`: 노드 x 를 포함한 리스트 만들어 리턴
 - ii. `find(x)`: 노드 x 가 속한 리스트의 `head` 노드를 찾아 리턴 (`head` 노드가 집합을 대표)
 - iii. `union(x, y)`: x, y 에 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_set`이 n 번 호출되었다고 하면, 총 n 개의 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`, `parent`와 `rank` 세 멤버 값을 갖는다 (자식 정보는 없음!)
 - ii. 트리의 루트 노드가 집합을 대표함
 - iii. `rank`는 `union`할 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

    def make_set(key):
        return Node(key)    # 노드를 만들어 리턴함

    def find(x):
        while x.parent != x:  # root 노드에 도달할 때까지
            x = x.parent
```

```
return x
```

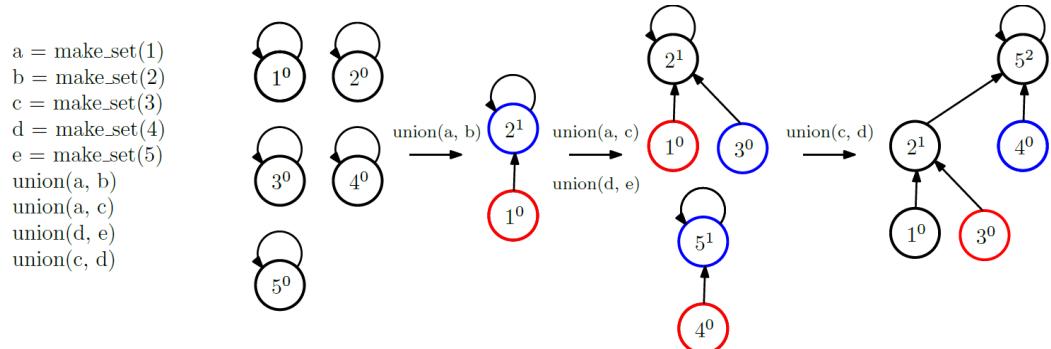
vi. `union(x, y):`

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 의 rank < w 의 rank라면 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, 작은 트리가 큰 트리의 자식으로 결합하는 형식으로 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

    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. h 와 rank의 관계는?

- b. rank 값에 따라 부모-자식 관계를 결정하는 이유가 있을 듯!
5. [?] rank = k라면 그 집합에는 최소 몇 개의 노드가 존재할까?
- a(k) = rank가 k 일 때 최소 노드 수로 정의
 - $a(0) = 1, a(1) = \underline{\hspace{2cm}}$
 - rank = $k-1$ 에서 k 가 될 때의 상황으로 $a(k)$ 점화식을
유도해보자 [hint: 두 rank가 같은 경우만 1씩 증가한다 → 같은
rank의 두 트리가 union 되므로 결합된 트리는 두 트리 중 작은
트리의 2배 이상이 된다]
 - 노드의 개수를 n 이라 하면 $a(k) \leq n$ 이 되어, k 값의 최대 값을 알
수 있게 된다 → 즉, 높이 h 를 알 수 있다?
- ix. **find** 함수를 **path compression**이라는 방법에 따라 수행하면 더 빠른 연산시간을 보장할 수 있다
- 구체적인 증명은 생략. 참고용 자료
[\[https://en.wikipedia.org/wiki/Disjoint-set_data_structure\]](https://en.wikipedia.org/wiki/Disjoint-set_data_structure)
 - find(x)**에서 x 노드부터 루트 노드로 이동하는 데, 이동 중에 방문하는 노드의 부모 노드를 모두 루트 노드로 변경하는 방법이 path compression이다: 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

```

3. 수행시간: find by path compression + union by rank

- a. find와 union 모두 (amortized)  $O(\alpha(n))$  시간이면 충분
- b.  $\alpha(n)$  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](#)]

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, 부분 트리를 특정하고 두 부분 트리를 합병하는 연산은 각각 find와 union 연산으로 처리할 수 있다
- f. Python의 set과 union-find 자료구조와의 차이점은?
  - i. set은 dict처럼 hash table로 구현되어 있어, find 함수는 **평균적**으로  $O(1)$  시간에 수행 가능하지만, **최악의 경우**에는  $O(n)$  시간이 필요함
  - ii. union-find 자료구조에서는 find 연산이 **최악의 경우**에도  $O(\log n)$  시간임

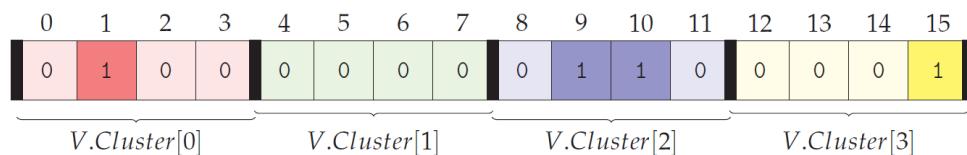
## 2. [상당히 복잡] van Emde Boas 트리

- 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
- c. 피보나치 힙(Fibonacci heap)은 더 복잡한 구조로 `deleteMax/deleteMin`만  $O(\log n)$ 이고 나머지 연산은 모두  $O(1)$  시간에 수행 가능
- d. 균형이진트리도 우선순위 큐에서 제공하는 모든 연산을  $O(\log n)$  시간에 제공한다
  - i. `deleteMax/deleteMin`, `decreaseKey`는 어떻게 처리할까?
- e. 이진 힙과 피보나치 힙에 저장되는 key 값으로는 실수 또는 문자열과 같이 대소를 비교할 수 있는 어떤 값도 가능하다
- f. 만약 key 값으로 정수 값만을 허용한다면 이 제한 조건을 이용해서 연산을 더 빠르게 할 수 있지 않을까?
  - i. 예: 임의의 값을 비교를 통해 정렬하는 데 최소  $O(n \log n)$  시간이 필요하지만, 최대  $d$ 자리의 수(정수 또는 실수)를 radix 정렬할 때에는  $O(dn)$  시간이면 충분한 것과 같은 이유
- g. 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 값으로 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 다른 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. 1975년, 네덜란드의 Peter van Emde Boas에 의해 제시된 자료구조!
  - iv. 응용: Network router (현재 노드에 도달한 패킷의 IP 주소가  $[a, b]$  구간에 속하면  $b$  노드로 routing 하는 방식이라면, 해당 IP 주소의 successor를 계산하면 된다)
- h. [youtube] <https://www.youtube.com/watch?v=hmReJCupbNU>
  - i. MIT Open Lecture: 이 강의 영상의 내용을 참고하여 설명함!
  - i. 그럼  $O(\log \log M)$ 은 어떻게 얻어졌을까?
    - i.  $T(k) = T(k/2) + O(1)$  인 경우  $T(k) = \underline{\hspace{10cm}}$
    - ii.  $T(\log M) = T(\log M/2) + O(1)$ 이면  $T(\log M) = \underline{\hspace{10cm}}$
    - iii.  $T(M) = T(\underline{\hspace{2cm}}) + O(1)$  일 때,  $T(M) = O(\log \log M)$ 이 될 때,  $T(\underline{\hspace{2cm}})$ 의 빈 칸은?
  - j. 알고리즘 1: Bit Vector

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 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 set하는 방식 → insert/delete 모두  $O(1)$  시간이면 충분
- 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
- iii. insert/delete:  $O(1)$ , successor:  $O(M)$  시간 (successor가 느림!)

### k. 알고리즘 2: $V$ 를 cluster로 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 위치는 무엇인가? \_\_\_\_\_
- iv.  $V$ 는 2차원 배열처럼 생각할 수 있음!  $x = 9$ 는  $V.cluster[ ] [ ]$ 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.  $x = i * \sqrt{M} + j$ 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 값
  2.  $i = \lfloor x / \sqrt{M} \rfloor = high(x)$
  3.  $j = x \bmod \sqrt{M} = low(x)$
  4. 왜 이렇게 계산이 가능할까?
- vii.  $index(i, j) = i * \sqrt{M} + j$ 로 정의
- viii. **insert( $V, x$ )**:
  1.  $V.cluster[high(x)][low(x)] = 1$
  2.  $V.summary[high(x)] = 1$
- ix. **successor( $V, x$ )**:
  1. if successor  $y$  is at  $j$  within  $V.cluster[high(x)]$ :

- a.  $y = \text{index}(\text{high}(x), j)$
  - 2. **else:**
    - a. find the next non-empty cluster  $i$
    - b. find the minimum entry  $j$  in  $V.\text{cluster}[i]$
    - c.  $y = \text{index}(i, j)$
  - 3. **return**  $y$
- x. **insert**는  $O(1)$  시간이면 충분
- xi. **successor**는
  - 1. 1번 단계에서  $y$ 가  $V.\text{cluster}[\text{high}(x)]$ 에 있는지 검사  $\rightarrow O(\sqrt{M})$
  - 2. 2.a 단계에서 다음 non-empty cluster는  $V.\text{summary}$ 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 존재함! (어떻게?)
  - 4. 결국,  $O(\sqrt{M})$  시간이 필요 (아직도 느림!)
- xii. **delete**는 여기선 고려하지 않음

### I. 알고리즘 3: Recursion (ver. 1)

- i. 알고리즘 2에서는 자료구조가 한 레벨만으로 구성된 자료구조임. (즉, 하나의  $V.\text{cluster}$ 와 하나의  $V.\text{summary}$ 만 존재)
- ii. 이 자료구조  $V$ 를 재귀적으로 여러 레벨에 걸쳐 구성함 [중요]
  - 1.  $V.\text{cluster}[i] = \text{size } \sqrt{M}$ 인 재귀 van Emde Boas 트리
    - a.  $V.\text{cluster}[0], \dots, V.\text{cluster}[\sqrt{M}-1]$ 로 구성
  - 2.  $V.\text{summary} = \text{size } \sqrt{M}$ 인 재귀 van Emde Boas 트리
- iii. **insert(V, x):**

```
insert(V.cluster[high(x)], low(x)) # insert x recursively
insert(V.summary, high(x)) # mark x recursively
```

  - $V.\text{cluster}$ 와  $V.\text{summary}$ 는 같은 크기의 vEB 트리 구조이므로, 수행시간을 위한 점화식은  $T(M) = 2T(\sqrt{M}) + O(1)$
  - 점화식을 풀면,  $T(M) = O(\log M)$  ( $\log \log M$ 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 == +infinity: # no successor in V.cluster[i]
 i = successor(V.summary, i) # (2)
 j = successor(V.cluster[i], -infinity) # (3)
return index(i, j)
```

- $T(M) = 3T(\sqrt{M}) + O(1)$ 이 된다.  $T(M) = \underline{\hspace{10mm}}$
- 우리가 원하는 수행시간이 아니다!

### m. 알고리즘 4: Recursion (ver. 2)

- 알고리즘 3의 successor 코드의 (3)에서 successor를 재귀적으로 호출한다. 이 호출은 V.cluster[i]에서 첫 번째 1의 offset 번호를 알기 위함이다. 만약 V.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 v에서 첫 번째 1의 offset 번호를 저장하는 V.min 멤버를 새로 정의해 저장한다 즉, V.min = v에서 첫 번째 1이 나타난 entry의 offset 번호 (다음 페이지의 그림 참조)
- 알고리즘 3의 코드에서 x의 successor가 V.cluster[i]에 있는 경우와 없는 경우를 (1)의 재귀호출의 리턴 값이 +infinity인지 비교해서 확인한다. 그런데 V.cluster[i]에서 마지막 1의 offset 번호를 V.cluster[i].max 변수에 저장해 놓았다면, low(x) < V.cluster[i].max이면 자신의 cluster 안에 successor가 있고, 아니면 다른 cluster에 successor가 있다는 뜻이다. 그러면, 상수 시간의 비교로 successor가 자신의 cluster 안에 있는지 다른 cluster에 있는지 구별할 수 있게 된다
- 모든 재귀적인 vEB 트리에 대해 min과 max 값을 저장함에 유의!
- insert(V, x):**
  1. **V.min = min(x, V.min) # V.min update**
  2. **V.max = max(x, V.max) # V.max update**
  3. **insert(V.cluster[high(x)], low(x)) # insert x recursively**
  4. **insert(V.summary, high(x)) # mark x recursively**
  - 수행시간에는 변화 없음!  $T(M) = O(\log M)$
  - 다음 그림에서 insert(V, 10)을 해보자

V.summary

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
|---|---|---|---|

V      V.cluster[0]      V.cluster[1]      V.cluster[2]      V.cluster[3]

|              |         |         |               |
|--------------|---------|---------|---------------|
| 0 0 1 0      | 0 1 0 1 | 0 0 0 0 | 0 1 0 0       |
| 0            | 4       | 8       | 12            |
| 2      V.min |         |         | 13      V.max |

vi. **successor(V, x):** # y를 x의 successor라 표기하면

```

1. i = high(x)
2. if low(x) < V.cluster[i].max: # y가 cluster[i]안에
 j = successor(V.cluster[i], low(x))
3. else: # y가 i 이후의 첫 non-empty cluster에 존재
 # 해당 cluster 번호는 V.summary에서의 successor!
 i = successor(V.summary, high(x))
 j = V.cluster[i].min (재귀 호출 불필요!)
4. return index(i, j)

```

- 위 그림에서 successor(V, 4)를 해보자
- successor(V, 9)를 해보자
- if 문의 결과에 따라 재귀함수가 정확히 한 번만 호출됨!
- 따라서 수행시간의 점화식은

$$T(M) = T(\sqrt{M}) + O(1) = O(\log \log M)$$

- 목표하는 수행시간!
- 그러나 insert가 아직  $O(\log M)$ 으로 느림!

a. 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  
 →  $O(\log M)$ 이 됨.  $O(\log \log M)$ 이 되기 위해선 한 번의 재귀호출만 해야 함  
 → V.cluster[high(x)]가 non-empty인 경우엔 V.summary에 기록할 필요가 없는데도 항상 재귀 호출(4번 라인)을 수행함!  
 → 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.min보다 더 작은 값 x가 삽입되면, (new min) x가 V.min에 저장되고 V.min에 저장되어 있었던 (old min) 값은 그제서야 cluster에 (재귀적으로) 저장한다 (즉, 재귀적인 저장 과정을 최대한 미루는(lazy) 방식으로 insert가 처리된다)
- iv. **insert(V, x):**
  1. if V.min == None:
 

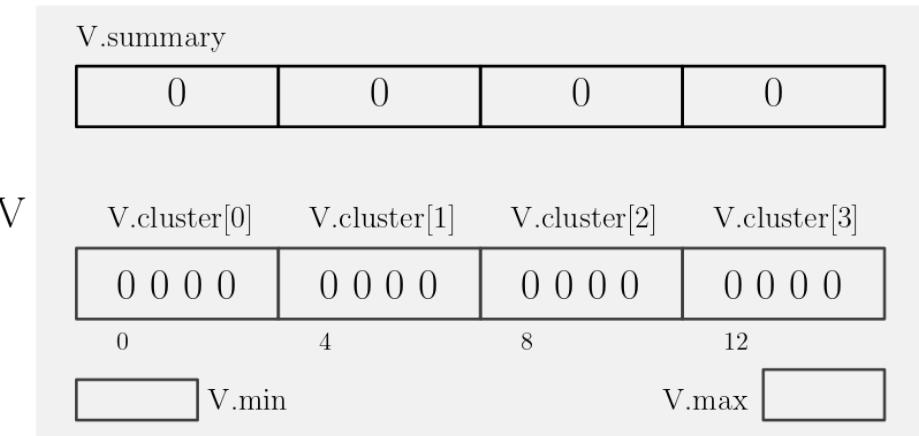
```
V.min = V.max = x # 최초 key값은 무조건 V.min에
return # O(1) time
```
  2. if x < V.min:
 

```
swap x and V.min # V.min에 저장된 key를 재귀 저장
```

```

3. if x > V.max: # V.max update
 V.max = x
4. if V.cluster[high(x)].min == None: # first key in cluster
 insert(V.summary, high(x)) # summary 재귀적 기록
5. insert(V.cluster[high(x)], low(x)) # 실제 cluster 삽입

```



- insert(V, 10), insert(V, 2), insert(V, 8)
- successor(V, 2)
- successor(V, 9)

v. 4번 라인의 **if** 문이 참이 되면 연속해서 두 개의 재귀호출이 이루어지는데?

- 그렇다면, 5번 라인의 insert 함수가 재귀 호출되어 1번 라인의 **if** 문을 만나면  $V.\min == \text{None}$ 이 되어 상수시간에 return되어 실제로 5번 라인에서의 재귀 호출은  $O(1)$  시간에 완료된다! 결국, **한 번의 재귀호출만 실질적으로 발생**하게 되어,  **$O(\log \log M)$**  시간을 보장한다

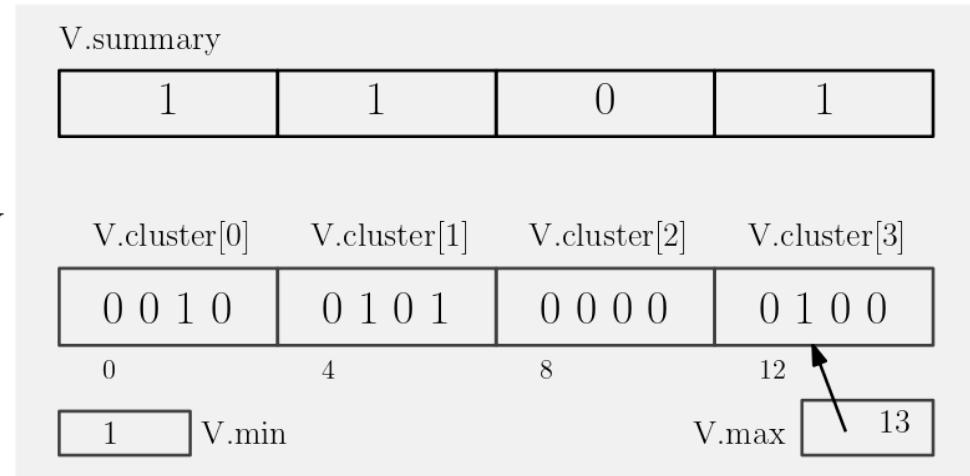
vi. 정리하면, V의 가장 작은 값은  $V.\min$ 에 저장하고, 실제 cluster와 summary 구조에는 저장하지 않는다. 대신  $V.\min$  보다 더 작은 값  $x$ 가 insert되면  $V.\min$ 의 값이 더 이상 min이 아니므로 cluster 또는 summary 구조에 (재귀적으로) 저장되고,  $x$ 는  $V.\min$ 에만 저장된다

vii. successor 코드도 아래 첫 줄을 추가해야 올바르게 동작한다 (왜?)

viii. **successor(V, x):**

- **if**  $x < V.\min$ : **return**  $V.\min$  # 추가됨!
- $i = \text{high}(x)$
- **if**  $\text{low}(x) < V.\text{cluster}[i].\max$ : # if  $y$  is in the cluster  
 $j = \text{successor}(V.\text{cluster}[i], \text{low}(x))$
- **else**: #  $y$  is in another cluster (= successor of  $V.\summary$  with  $i$ )  
 $i = \text{successor}(V.\summary, \text{high}(x))$   
 $j = V.\text{cluster}[i].\min$
- **return**  $\text{index}(i, j)$

ix. **Delete** 연산



```

delete(v, x):
 # Update V.min
 • if x == V.min: # 삭제할 key가 V.min인 경우:
 # min 값은 V.cluster가 아닌 V.min에만 저장되어 있음!

 i = V.summary.min # first non-empty cluster
 if i == None: # 현재 오직 하나의 key (v.min)만 존재
 V.min = V.max = None # 두 값만 None으로~
 return

 # V.min 이외의 key가 존재하므로 다음 V.min을 찾기
 # V.min으로 set하고, 동시에 x로 set하여 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 == None:
 # 지운 후 cluster가 empty가 되어 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 비용!

 delete(V.summary, high(x)) # second call

 # update V.max
 • if x == V.max: # V.max가 단계 3에서 지워진 경우
 if V.summary.max == None: # 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!

b. 이 자료구조가 사용하는 메모리 양:  $O(M)$

- i. 실제 저장된 key의 개수는  $n$ 이므로 메모리의 양도  $n$ 으로 표현되는 게 바람직
- ii. 만약,  $M \gg n$ 이라면 메모리 사용량이 매우 큰 자료구조가 됨
- iii. 이후, 연구자들에 의해, cluster를 연속된 배열에 저장하는 것이 아니라 non-empty cluster만 해시 테이블로 저장하는 방식으로  $O(n)$  메모리만 사용하면 되도록 하였음!

#### 4. Suffix array (suffix 배열)

- a. 문자열  $S = "banana"$ 에서 **suffix**는  $S[i:n]$ 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]

```

| i    | 0  | 1  | 2 | 3  | 4 | 5  | 6 |
|------|----|----|---|----|---|----|---|
| A[i] | 6  | 5  | 3 | 1  | 0 | 4  | 2 |
| 1    | \$ | a  | a | a  | b | n  | n |
| 2    |    | \$ | n | n  | a | a  | a |
| 3    |    |    | a | a  | n | \$ | n |
| 4    |    |    |   | \$ | n | a  |   |
| 5    |    |    |   |    | a | n  |   |
| 6    |    |    |   |    |   | \$ |   |
| 7    |    |    |   |    |   |    |   |

- 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. Hint:  $P$ 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:  $O(m \log n)$

## 5. 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)

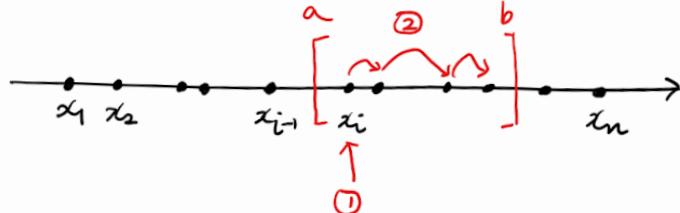
**목표:** 질의를 최대한 빠르게 처리할 수 있는  $P$ 에 대한 자료구조를 효율적으로 구성함

**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 a를 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 b보다 작거나 같은 점들을 차례대로 출력한다:  $O(k)$  ( $k = [a, b]$ 에 포함되는 점들의 개수)

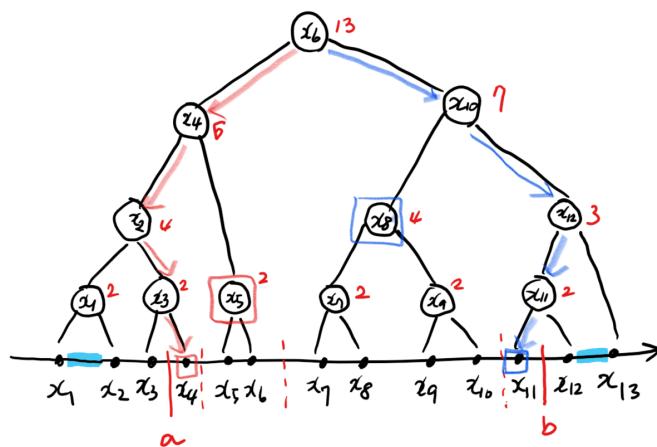
1D: sorted array

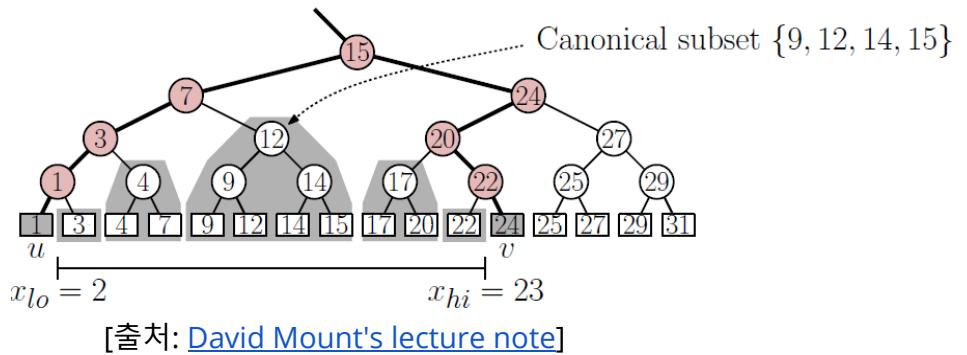


- preprocessing time =  
space =  
reporting (query) time =
- [질문] reporting이 아닌 counting (구간 안에 들어오는 점 개수만 세는 문제)을 위해 필요한 시간은?

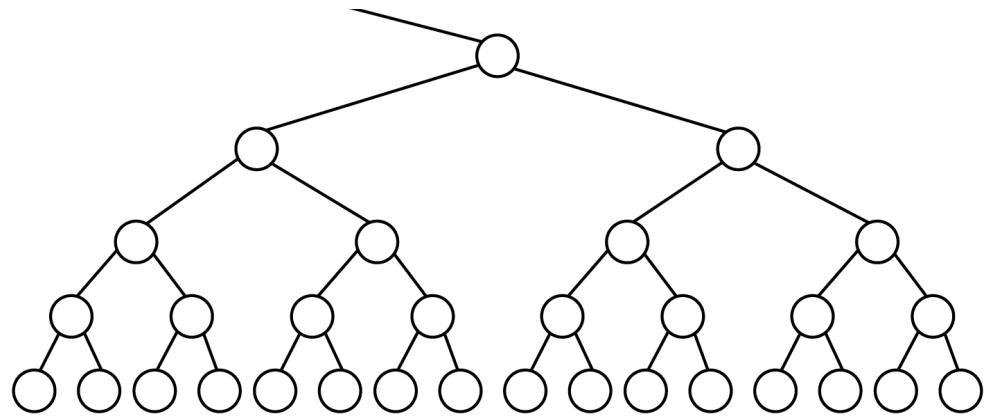
iii. **방법 2:** 1차원 range tree에 점들을 저장해보자

- 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?)



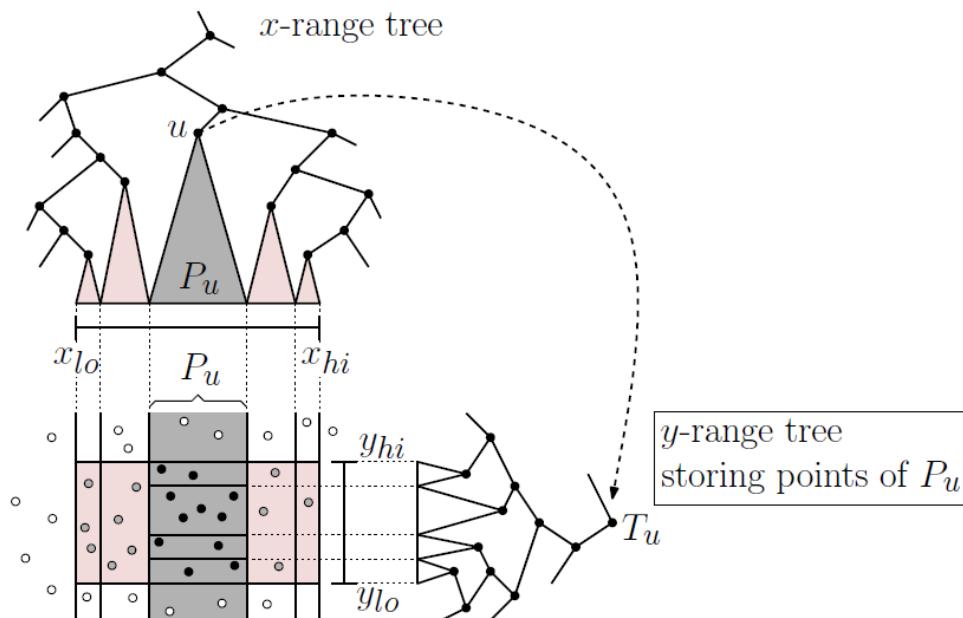


- 연습:



- preprocessing time =  
space =  
reporting time =
- [질문] counting 문제는 더 빠르게 해결할 수 있을까? 어떻게?

- i. **2차원 orthogonal range searching 문제:** 아래 그림을 참조할 것!
- 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
  - x-좌표 값을 기준으로 1차원 range tree  $T_x$ 를 만든다
  - 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
  - 각 노드는 부트리의 리프노드에 저장된 점들의 집합을 나타낸다
  - 이 점들 중에서 y-좌표 구간  $[c, d]$ 에 포함되는 점들만 골라 내야 한다. (어떻게?)
  - 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)
  - 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
  - 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 =  
space =  
query time =
- x. 2차원 range tree는 1차원 range tree의 각 노드에 1차원 range tree가 다시 저장된 재귀적인 자료구조이다
- xi. 같은 방식으로 **d차원 range tree**를 구성할 수 있다
- xii. preprocessing time =  
space =  
query time =

