



알고리즘 e러닝

최적의 코딩을 결정하는 기본 알고리즘

이것이 코딩테스트다 강의와 순서만 다름

1. 가장 기본이 되는 자료구조: 스택과 큐

1. 스택

- 먼저 들어온 데이터가 나중에 나가는 형식. 선입후출. LIFO
- 입구와 출구가 동일한 형태로 스택을 시각화할 수 있다.
- 박스쌓기
- 동작 방법과 사용예시 알아둬야 할 필요가 있음. B?D?FS 등에서 자주 쓰임

-구현 by python

리스트 자료형 사용하면 됨

```
stack = []
#파이썬에서는 스택 구현에 리스트 사용

stack.append(5)
stack.append(2)
stack.append(3)
stack.append(7)
stack.pop()
stack.append(1)
stack.append(4)
stack.pop()

print(stack[::-1]) # 최상단 원소부터 출력
print(stack) # 최하단 원소부터 출력

# 파이썬에서는 별도로 표준라이브러리 이용할 필요 없이 리스트를 이용하면 된다.
```

파이썬에서는 별도로 표준라이브러리 이용할 필요 없이 리스트를 이용하면 된다.

자바에서는 push, pop으로 삽입삭제, 최상단부터 출력할 때는 peek사용!

2. 큐

- 먼저 들어온 데이터가 먼저 나가는 형식의 자료구조. 선입선출
- 큐는 입구와 출구가 모두 뚫려있는 터널과 같은 형태로 시각화 가능.
- 줄서있는 모습. 대기열. 먼저온사람 차례대로
- 데이터가 들어가는 쪽이 rear, 나오는 쪽이 front

구현 by python

```
from collections import deque

#큐 구현할 때는 덱 라이브러리 사용
queue = deque()

queue.append(5)
queue.append(2)
```

```

queue.append(3)
queue.append(7)
queue.popleft()
queue.append(1)
queue.append(4)
queue.popleft()
#삽입은 append, 삭제는 popleft

print(queue) # 먼저 들어온 순으로 출력
queue.reverse() # 역순으로 바꾸기
print(queue) # 나중에 들어온 순으로 출력

# 리스트를 이용해도 가능적으로는 큐를 구현할 수도 있지만
# 시간 복잡도가 높아서 비효율적. pop할시 꺼낸 다음 나머지를 앞으로 땡겨야해서 시간복잡도가 O(n)
# 엄밀히 말하면 덱 라이브러리는 스택과 큐의 장점을 합친 구조의 자료구조라고 볼 수 있음. list의 append와 동일하게 작동함. 오른쪽으로 삽입. 시간복잡도 상수시간. O(1)
# popleft도 가장 왼쪽에 있는 데이터 삭제니 O(1)
#출력결과
#deque([3, 7, 1, 4])
#deque([4, 1, 7, 3])

```

자바에서는 삽입offer, 삭제 poll 사용.

먼저 들어온 원소부터 추출하려면

System.out.print(q.poll() + " ");

2. 우선순위에 따라 데이터를 꺼내는 자료구조

우선순위가 가장 높은 데이터를 가장 먼저 삭제하는 자료구조.

우선순위 큐는 데이터를 우선순위에 따라 처리하고 싶을 때 사용

ex) 물건 데이터를 자료구조에 넣었다가 가치가 높은 물건부터 꺼내서 확인해야 하는 경우


스택과 큐는 데이터를 추출할 때 각각 가장 나중에 넣은 것, 가장 먼저 넣은 것을 추출함.

우선순위 큐는 가장 우선 순위가 높은 데이터를 추출하는 것!

우선순위 큐를 구현하는 방법

- 1) 리스트를 이용하여 구현
- 2) 힙(heap)을 이용하여 구현

- 데이터의 개수가 N개일 때, 구현 방식에 따라서 시간 복잡도를 비교한 내용은 다음과 같습니다.



우선순위 큐 구현 방식	삽입 시간	삭제 시간
리스트	$O(1)$	$O(N)$
힙(Heap)	$O(\log N)$	$O(\log N)$

- 단순히 N개의 데이터를 힙에 넣었다가 모두 꺼내는 작업은 정렬과 동일합니다. (힙 정렬)
 - 이 경우 시간 복잡도는 $O(N\log N)$ 입니다.

리스트 - 삽입은 그냥 하면되지만 삭제의 경우 모든 요소들의 우선순위를 확인해서 삭제해야하기 때문에 $O(N)$

힙 - 데이터를 넣고 뺄 때 모두 최악의 경우에도 $O(\log N)$ 의 시간복잡도를 보장. 모두 삽입하고 삭제할 경우 힙정렬과 같은 작업으로 시간 복잡도 $O(N\log N)$. 병합정렬, 퀵정렬과 마찬가지로 빠른 정렬 방법!

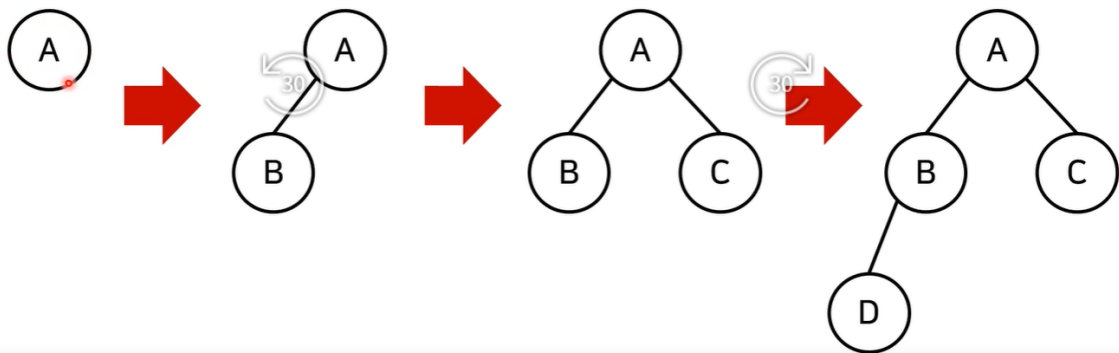
여기서 힙이란?

- 완전 이진트리 자료구조의 일종
- 힙에서는 항상 루트 노드를 제거한다
- 최소 힙(min heap)

- 루트 노드가 가장 작은 값을 가진다.
- 따라서 값이 작은 데이터가 우선적으로 제거된다.
- 최대 힙(max heap)
 - 루트 노드가 가장 큰 값을 가진다.
 - 값이 큰 데이터가 우선적으로 제거!

힙은 완전이진트리 형식을 따른다.

- 완전 이진 트리란 루트(root) 노드부터 시작하여 왼쪽 자식 노드, 오른쪽 자식 노드 순서대로 데이터가 차례대로 삽입되는 트리(tree)를 의미합니다.

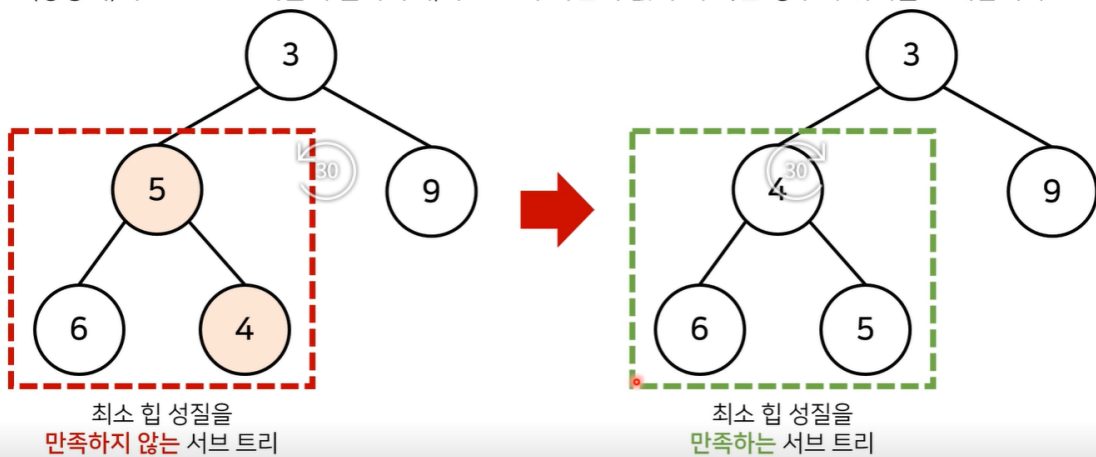


최소 힙 구성 함수 : Min-Heapify()

상황식 - 부모 노드로 거슬러 올라가며 부모보다 자식의 값이 작을 때 위치 교환하는 방식

최소 힙 구성 함수: Min-Heapify()

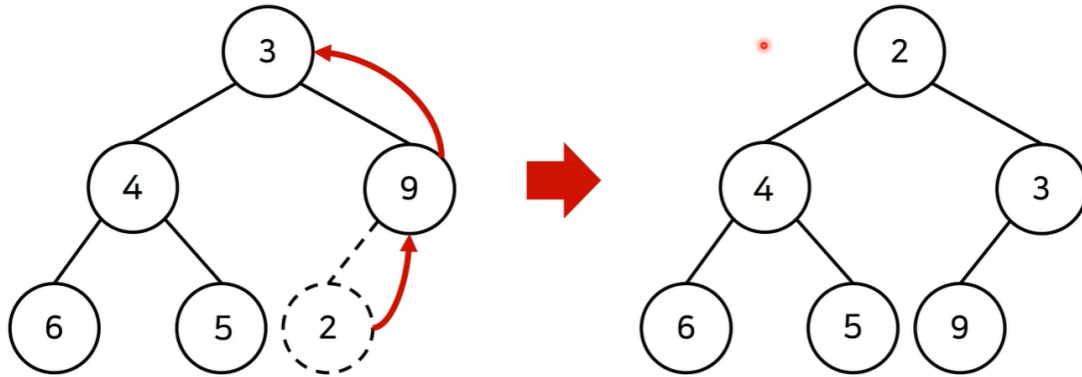
- (상황식) 부모 노드로 거슬러 올라가며, 부모보다 자신의 값이 더 작은 경우에 위치를 교체합니다.



heapify 함수를 통해 최소힙 성질을 만족하도록 변경할 수 있다

힉에 새로운 원소가 삽입될 때

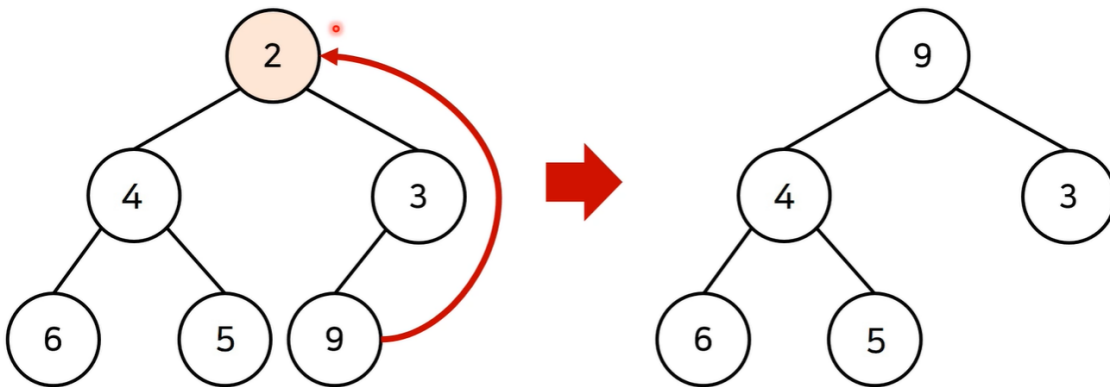
- 새로운 원소가 삽입되었을 때 $O(\log N)$ 의 시간 복잡도로 힉 성질을 유지하도록 할 수 있습니다.



제거할 때도 마찬가지!

힉에서 원소가 제거될 때

- 원소가 제거되었을 때 $O(\log N)$ 의 시간 복잡도로 힉 성질을 유지하도록 할 수 있습니다.
 - 원소를 제거할 때는 가장 마지막 노드가 루트 노드의 위치에 오도록 합니다.



최소 힉에서는 최솟값인 루트가 먼저 제거되는데 원소가 제거되었을 때 가장 마지막 노드를 루트 노드로 옮긴 후에 루트 노드부터 **하향식**으로 Heapify()! → 상향식과 반대로 leaf까지 거슬러 내려가면서 비교! 부모가 더 클때 위치를 교환한다.

파이썬에서는

`import heapq` 를 통해 힉라이브러리 임포트해서 사용할 수 있음. 언어마다 기본 힉이 최소힉, 최대힉 으로 다를 수 있음. 파이썬에서는 최소힉으로 구현되어 있다. 따라서 힉정렬을 하게되면 오름차순으로 정렬하게 됨!

우선순위 큐 라이브러리를 활용한 힙 정렬 구현 예제 (Python)

```
import sys
import heapq
input = sys.stdin.readline

def heaport(iterable):
    h = []
    result = []
    # 모든 원소를 차례대로 힙에 삽입
    for value in iterable:
        heapq.heappush(h, value)
    # 힙에 삽입된 모든 원소를 차례대로 꺼내어 담기
    for i in range(len(h)):
        result.append(heapq.heappop(h))
    return result

n = int(input())
arr = list(map(int, input().split()))
res = heaport(arr)
for i in range(n):
    print(res[i])
```

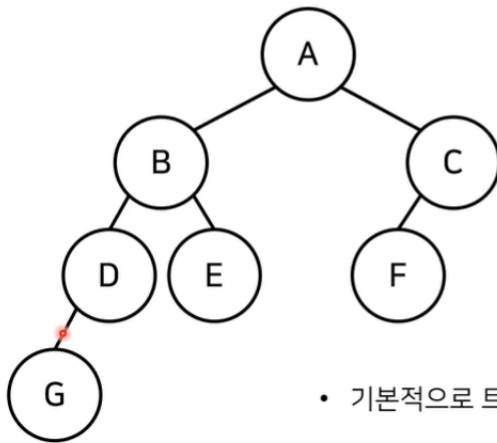
만약 내림차순으로 동작하도록 만들고 싶다면 데이터를 삽입할때와, 꺼낼때 -를 붙이면 맥스힙으로 만들 수 있다.

3. 활용도가 높은 자료구조: 트리 자료구조

트리 - 가계도와 같은 계층적인 구조를 표현할 때 사용할 수 있는 자료구조

트리 (Tree)

- 트리는 가계도와 같은 계층적인 구조를 표현할 때 사용할 수 있는 자료구조입니다.



[트리 관련 용어]

- 루트 노드(root node): 부모가 없는 최상위 노드
- 단말 노드(leaf node): 자식이 없는 노드
- 크기(size): 트리에 포함된 모든 노드의 개수
- 깊이(depth): 루트 노드부터의 거리
- 높이(height): 깊이 중 최댓값
- 차수(degree): 각 노드의 (자식 방향) 간선 개수

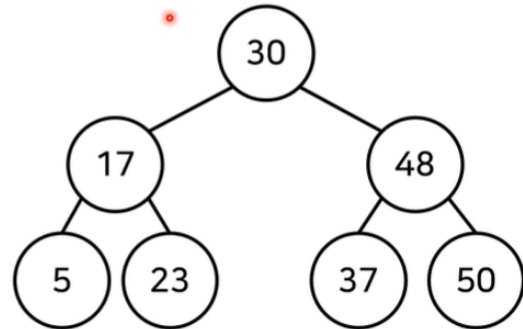
- 기본적으로 트리의 크기가 N 일 때, 전체 간선의 개수는 $N - 1$ 개입니다.

맨 마지막 줄의 의미는 트리에 포함되어 있는 노드가 N 개일 때 전체 간선의 개수는 $N-1$ 개라는 뜻. 기억해두면 좋다.

- 이진 탐색 트리(Binary Search Tree)

이진 탐색 트리 (Binary Search Tree)

- 이진 탐색이 동작할 수 있도록 고안된 효율적인 탐색이 가능한 자료구조의 일종입니다.
- 이진 탐색 트리의 특징: 왼쪽 자식 노드 < 부모 노드 < 오른쪽 자식 노드
 - 부모 노드보다 왼쪽 자식 노드가 작습니다.
 - 부모 노드보다 오른쪽 자식 노드가 큼니다.



이진 탐색 트리의 특징 기억!

왼쪽 자식 < 부모 < 오른쪽 자식

이진탐색트리에서 데이터를 조회하는 방법

1. 루트 노드부터 방문하여 탐색 진행. 현재 노드와 찾고자 하는 원소를 비교. 찾는 원소가 더 크다면 오른쪽, 더 작다면 왼쪽으로 이어서 진행
2. 현재 노드와 찾는 원소를 비교. 원소를 찾을 때까지 1,2 번 방법을 진행해 나가다가 찾으면 종료

이진 탐색 트리의 특징에 따라 나머지 반대쪽은 탐색할 필요가 없어진다 → 이상적인 경우(왼쪽과 오른쪽의 노드 숫자가 비슷하게 존재하고 있을 때) 탐색 횟수가 절반가량씩 줄어든다! $O(\log N)$

- 트리의 순회

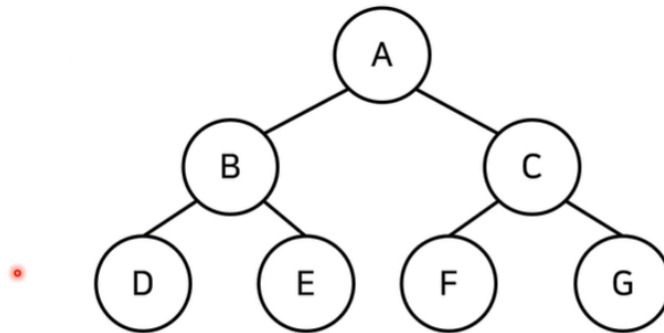
트리 자료구조에 포함된 노드를 특정한 방법으로 한 번씩 방문하는 방법. 트리의 정보를 시각적으로 확인 가능.

대표적인 방법으로는

1. 전위 순회: 루트 방문 → 왼쪽 자식 → 오른쪽 자식
2. 중위 순회: 왼쪽 자식 → 루트 방문 → 오른쪽 방문
3. 후위 순회: 왼쪽 자식 → 오른쪽 자식 → 루트 방문

일반적으로 트리를 구현할 때 순회 방법까지 구현함

트리의 순회 (Tree Traversal)



[입력 예시]

```

7
A B C
B D E
C F G
D None None
E None None
F None None
G None None
  
```

- 전위 순회(pre-order traverse): A → B → D → E → C → F → G
- 중위 순회(in-order traverse): D → B → E → A → F → C → G
- 후위 순회(post-order traverse): D → E → B → F → G → C → A

구현은 자바와 마찬가지로 Node라는 클래스 정의해서 가능. 트리는 딕셔너리 사용

트리의 순회 (Tree Traversal) 구현 예제

```

class Node:
    def __init__(self, data, left_node, right_node):
        self.data = data
        self.left_node = left_node
        self.right_node = right_node

# 전위 순회(Preorder Traversal)
def pre_order(node):
    print(node.data, end=' ')
    if node.left_node != None:
        pre_order(tree[node.left_node])
    if node.right_node != None:
        pre_order(tree[node.right_node])

# 중위 순회(Inorder Traversal)
def in_order(node):
    if node.left_node != None:
        in_order(tree[node.left_node])
    print(node.data, end=' ')
    if node.right_node != None:
        in_order(tree[node.right_node])
  
```

```

# 후위 순회(Postorder Traversal)
def post_order(node):
    if node.left_node != None:
        post_order(tree[node.left_node])
    if node.right_node != None:
        post_order(tree[node.right_node])
    print(node.data, end=' ')

n = int(input())
tree = {}

for i in range(n):
    data, left_node, right_node = input().split()
    if left_node == "None":
        left_node = None
    if right_node == "None":
        right_node = None
    tree[data] = Node(data, left_node, right_node)

pre_order(tree['A'])
print()
in_order(tree['A'])
print()
post_order(tree['A'])
  
```

4. 특수한 목적의 자료구조: 바이너리 인덱스 트리

비트..연산을 이해할 수가 없다. 안배웠어서 이해가안된다.