

2주차 : 트리, 힙, 우선순위 큐, 이진 트리 순회

* 용어 설명

- 그래프란? $G = (V, E)$

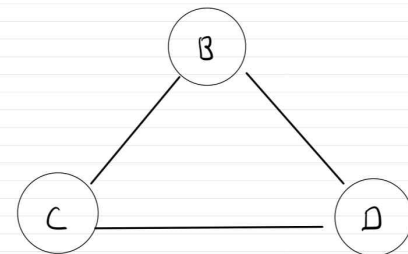
- 정점(vertex, V), 간선(edge, E)의 조합

=> 애도 그래프다

ex. 오른쪽 그래프에서

$V = \{A, B, C, D\}$

$E = \{(B, C), (B, D), (C, D)\}$



- 인접 : 두 정점 사이에 간선이 있을 경우

- walk : 인접한 정점들을 따라 이동한 족적

- 닫힌 walk : 시작점과 끝점이 동일한 walk

- 연결(connected) 그래프 : 임의의 두 정점 사이에 walk가 적어도 하나 존재하는 것

=> 애는 비연결 그래프

- 연결 그래프에서는, 항상 $E = V - 1$ 임을 알 수 있다.

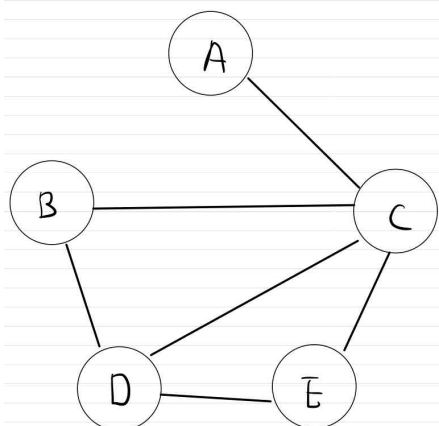
- path : walk 중 중복된 꼭짓점이 존재하지 않는 것

ex. 아래 그래프에서 A-C-D-B-C-E는 walk(o), path(x)

- cycle : 닫힌 walk와 path의 조건을 모두 만족하는 경우

(= 출발했던 점으로 돌아올 수 있는데, 중복된 점이 (시작점&종점) 제외하곤 없는 경우)

ex. 아래 그래프에서의 cycle?



- B-C-D-B

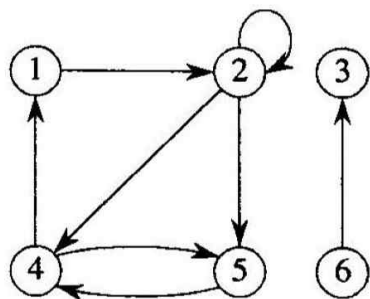
- D-C-E-D

- B-C-E-D-B

- Directed graph : V 가 공집합이 아닌 유한 집합이고, E 는 V 사이의 관계(relation)이 있는 경우

= 간선이 화살표인 경우

ex.



(앞선 개념들을 배운 이유 : 모든 트리는 그래프이다!)

- 트리(Tree) : 아래의 세 조건을 만족시키는 특수한 그래프

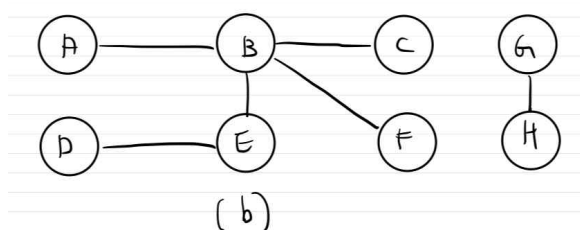
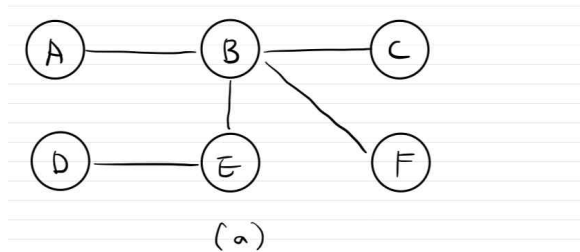
1) Undirected graph

2) acyclic

3) 연결 그래프

cf) Forest : 위 조건 중 1), 2) 조건을 만족하는 경우

ex.



| | (a) | (b) |
|-------------|-----|-----|
| undirected? | O | O |
| acyclic? | O | O |
| connected? | O | X |

=> (a) : 트리, (b) : Forest

- node : 정점을 트리에선 node라고 부름

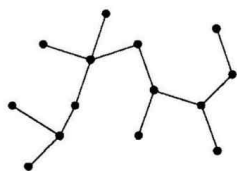
- rooted tree : 아래의 두 조건을 만족시키는 그래프

(1) 트리

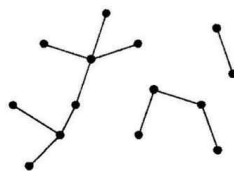
(2) root node : 부모 노드가 없는 노드

- root로 인해 '위계'가 생기므로, 부모 node와 자식 node가 있음.

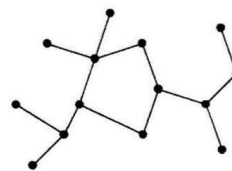
ex.



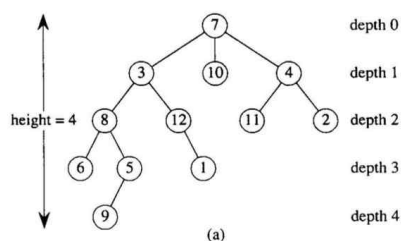
Not a rooted tree



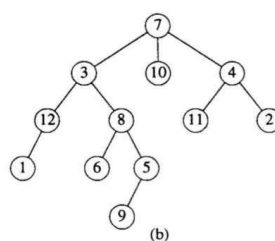
forest



Undirected graph



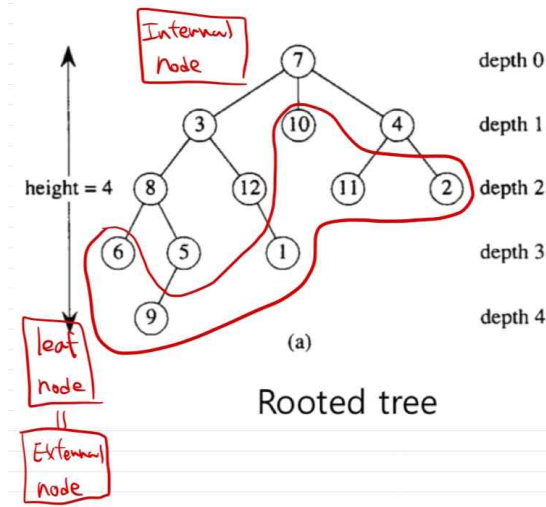
Rooted tree



(a) And (b) are different rooted trees

- leaf node : 자식 노드가 없는 노드

ex.



- 이진 트리(binary tree) : 아래의 두 조건을 만족시키는 그래프

(1) rooted tree

(2) 각 노드는 최대 2개의 자식 노드를 가짐

- 이진 트리의 구조

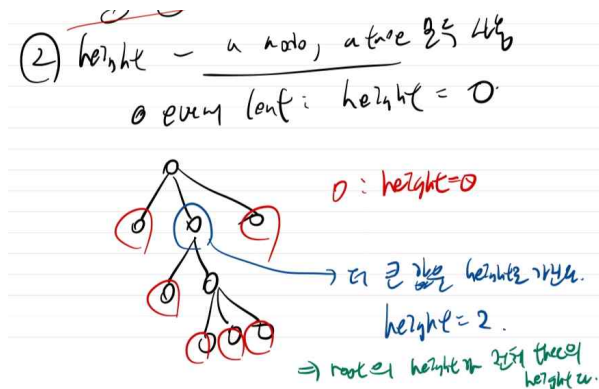
| 포화 이진 트리(Full Binary Tree) | 완전 이진 트리(Complete Binary Tree) |
|--|---|
| <p>Perfect Binary Tree</p> <ul style="list-style-type: none"> ✓ 모든 곳이 빈틈없이 채워진 이진 Tree를 일컫음 ✓ 모든 내부 Node의 자식이 둘이며, 모든 Leaf Node의 깊이가 동일한 이진 Tree | <p>Complete Binary Tree</p> <ul style="list-style-type: none"> ✓ Perfect Binary Tree에서 Node 일부를 적당히 제거한다면? ✓ Tree의 최하단을 제외한 영역은 Perfect Binary Tree를 이루며, 최하단의 모든 Node는 좌측에 물려 있는 경우 완전 이진 Tree |

- Depth, Height란?

- Depth : 루트 노드에서 현재 노드까지의 거리(즉 간선의 수)

- level : root 노드는 0, root 노드의 자식은 1, ... 으로 생각할 수 있다. (depth와 몹시 유사한 개념)

- Height : 현재 노드에서 Leaf 노드까지의 거리



- Degree(차수) : 현재 노드의 자식의 개수

- leaf node의 차수 = 0

cf) 트리의 높이와 전체 노드 개수 사이의 관계식 (height of tree = h , # node = n)

$$\Rightarrow h \leq \log_2 n < h+1$$

$$\text{즉, } h = \lfloor \log_2 n \rfloor$$

pf)

level 0(root) 에는 2^0 개의 노드,

level 1에는 2^1 개의 노드,

level 2에는 2^2 개의 노드

...

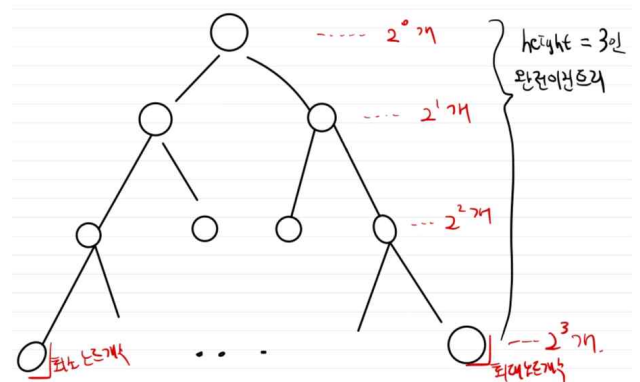
level h 에는 2^h 개의 노드가 있다.

- 이를 바탕으로 생각하면 $2^h \leq n < 2^{h+1}$ 이므로 $h \leq \log_2 n < h+1$

$$\text{즉 } \log_2 n - 1 < h \leq \log_2 n \text{이므로 } h \leq \log_2 n < h+1$$

$$\therefore h = \lfloor \log_2 n \rfloor \quad (\because h : \text{정수})$$

ex) $h=3$



$$\Rightarrow (2^0 + 2^1 + 2^2 + \dots + 2^h) \sim (2^0 + 2^1 + 2^2 + \dots + 2^h) \text{개의 노드를 가질 수 있다.}$$

$$2^3 \sim 2^4 - 1 \text{ 개의 노드.}$$

즉 height 3인 완전이진트리는

$$2^3 \leq n < 2^4 \text{ 개의 노드를 가질 수 있다.}$$

\Rightarrow 뒤에서 배울 Heap에서 push, pop 연산 속도가 $O(\log_n)$ 인 것과 연관됨

자료구조 Heap

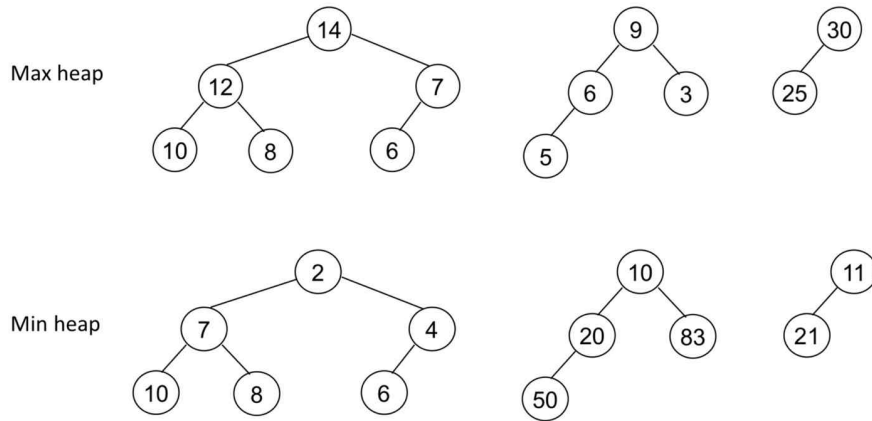
- **Heap**이란? 다음의 두 조건을 만족시키는 그래프

(1) 완전 이진 트리(Complete Binary Tree)

(2) heap property를 만족

- heap property란? : 다음을 만족하는 부모 노드와 자식 노드 사이의 관계
 - : 모든 부모 노드가 자신의 자식보다 큰 값을 가지거나(Max heap property)
 - : 모든 부모 노드가 자신의 자식보다 작은 값을 가지거나(min heap property)

ex.

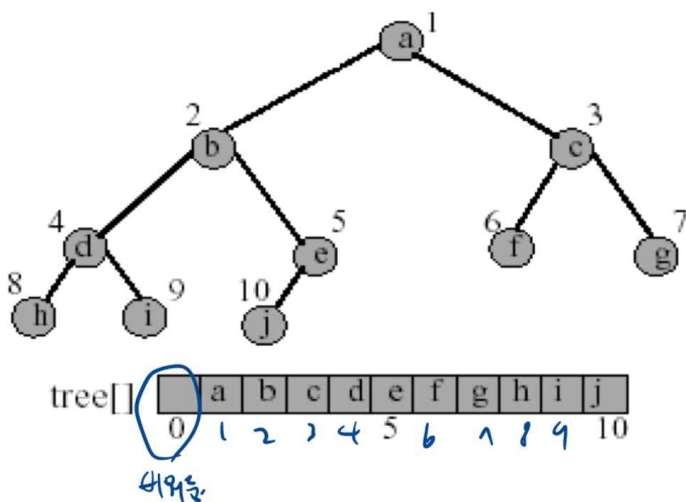


cf. 대부분의 강의에서 Max heap을 기준으로 개념 설명을 이어가며, 저희도 그렇게 하겠습니다.

- 이진 트리의 배열 표현 :

- 일반적으로 힙은 배열로 구현한다(\because 완전 이진 트리이므로 중간에 비어있는 요소가 없기 때문)
- root 노드를 배열의 0번 index에 넣고, 그 이후로 쪽 번호 매겨가며 배열에 대입한다.
- 왼쪽 자식 노드 index = (부모 노드 index) * 2
- 오른쪽 자식 노드 index = (부모 노드 index) * 2 + 1
- 부모 노드 index = (자식 노드 index) / 2

cf. 일반적으로는 root 노드를 배열의 1번 index에 넣는다고 하나, 유용재 교수님 강의에서는 0번 index부터 넣음



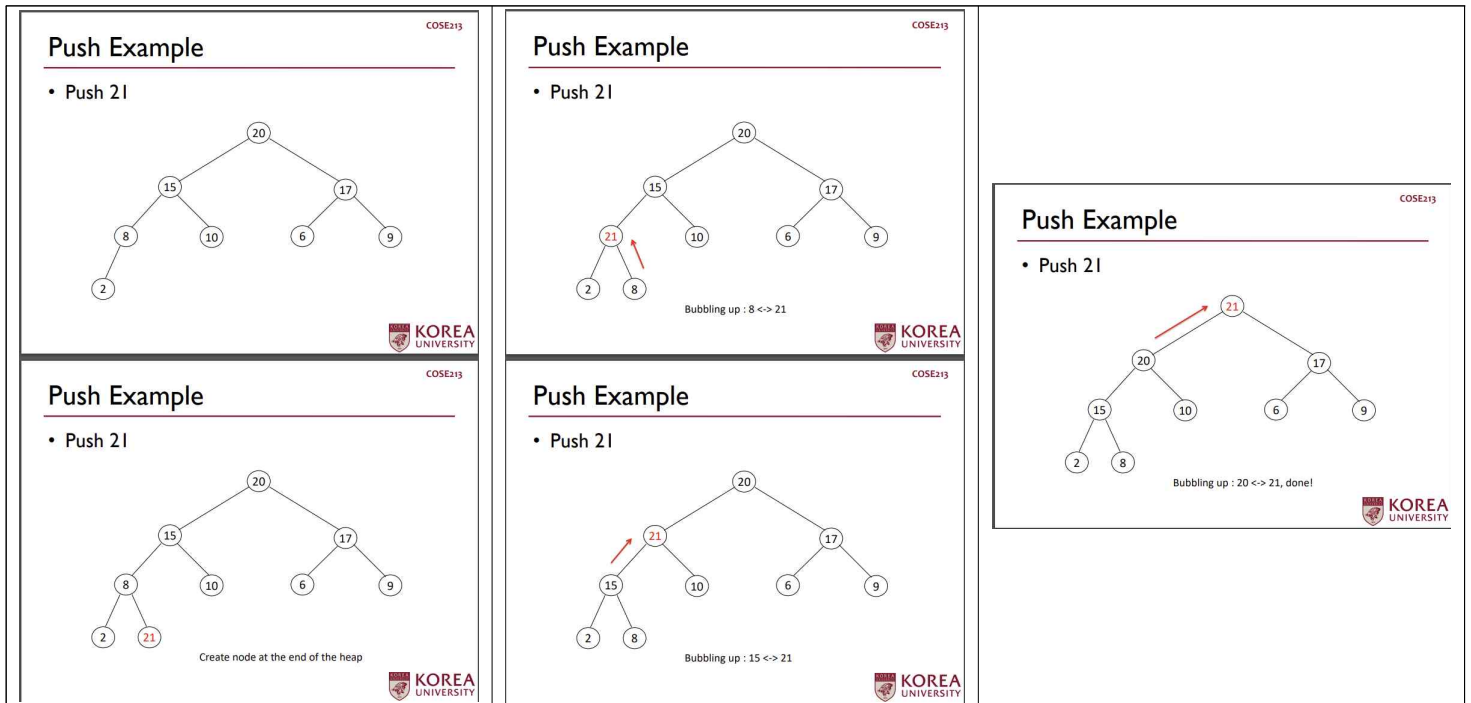
- Heap 자료구조의 주요 메소드 : push(), pop()

1) **push()** : 마치 물 속에서 공기방울이 떠오르듯 bubble up!(heap property 만족할 때까지)

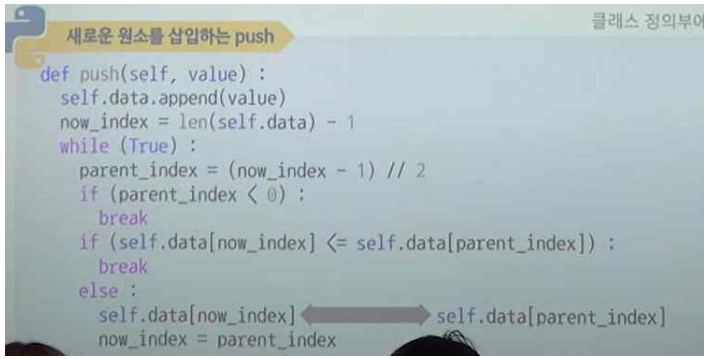
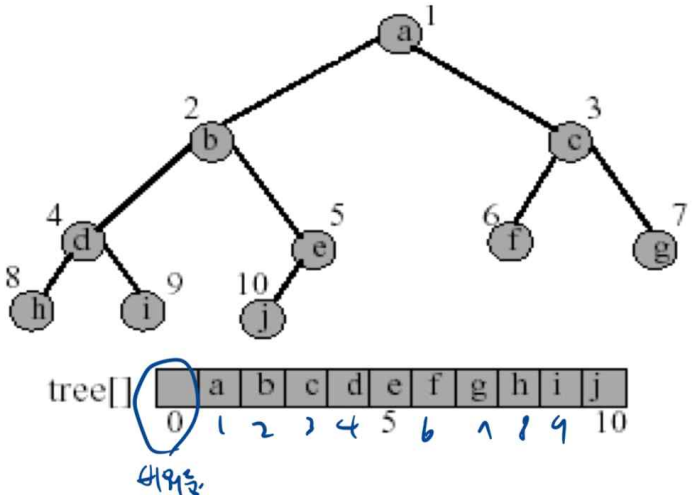
- 현재 노드와 부모 노드와의 관계를 살핀다.

- 시작 위치 : 완전 이진 트리이므로, 위치는 고정

ex.



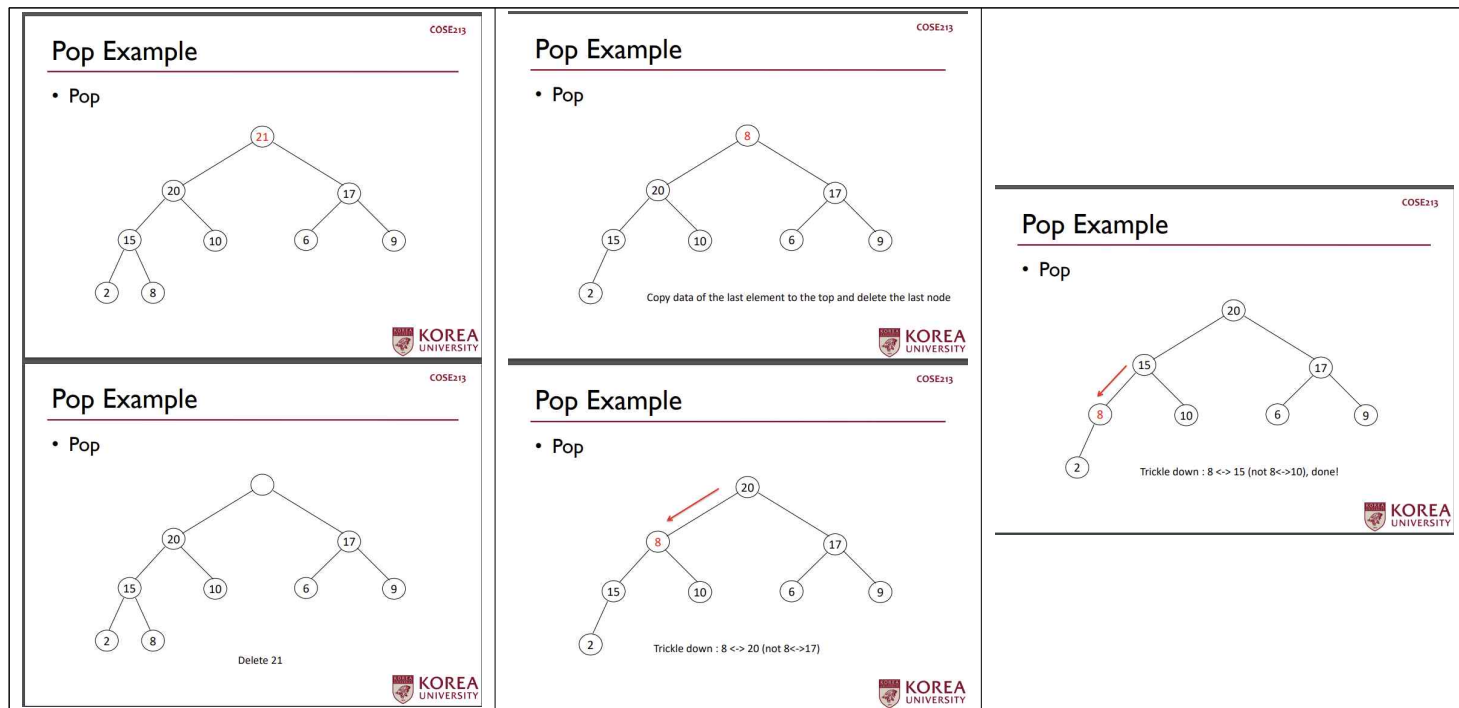
- push() 코드 구현

| python | C++ |
|---|--|
|  <pre> def push(self, value) : self.data.append(value) now_index = len(self.data) - 1 while (True) : parent_index = (now_index - 1) // 2 if (parent_index < 0) : break if (self.data[now_index] <= self.data[parent_index]) : break else : self.data[now_index] <-> self.data[parent_index] now_index = parent_index </pre> | <pre> template <class Type> void MaxHeap<Type>::Push(const Element<Type> &x) { if (n==MaxSize) { HeapFull(); return; } n++; for(int i=n; 1;) { if (i==1) break; // Root reached if (x.key <= heap[i/2].key) break; // move parent down heap[i] = heap[i/2]; i /= 2; } heap[i] = x; } </pre> |
| | <ul style="list-style-type: none"> - n : 현재 힙의 크기 - 최대크기에 도달하면 HeapFull()하고 종료 - n++ : 현재 힙의 크기를 1 증가 - for(int i=n; 1; ;) : i는 n이고, 1이므로 항상 참인 조건문(=무한루프) - break하는 경우 : 루트에 도달하거나, 현재 노드의 값이 부모 노드보다 작거나 같은 경우 - swap이 가능한 경우에는 swap하고, index를 절반으로 줄인다. <p>(해당 코드에서는 heap[0]이 다음 사진처럼 비워져있음)</p>  |

2) pop() : trickling down 하며

- 시작 위치 : root 노드를 삭제
- 비어있는 root 노드 자리에 가장 마지막 원소를 넣고(∵ 완전 이진트리 구조 유지해야하니까),
- trickling down!(heap property 만족할 때까지)
- 현재 노드와 자식 노드와의 관계를 살핀다

ex.



코드 구현

| python | C++ |
|---|---|
| <div data-bbox="92 1279 799 1666"> <p>Root Node를 제거하는 pop 1 / 2</p> <pre>def pop(self) : now_index = len(self.data) - 1 self.data[0] = self.data[now_index] del self.data[now_index] now_index = 0 while (True) : left_child_index = now_index * 2 + 1 right_child_index = now_index * 2 + 2 if left_child_index >= len(self.data) : break</pre> </div> <div data-bbox="92 1682 799 1951"> <p>Root Node를 제거하는 pop 2 / 2</p> <pre>if (self.data[now_index] >= self.data[left_child_index]) and (self.data[now_index] >= self.data[right_child_index]) : break else : if (self.data[left_child_index] > self.data[right_child_index]) : self.data[now_index] <=> self.data[left_child_index] now_index = left_child_index else : self.data[now_index] <=> self.data[right_child_index] now_index = right_child_index</pre> </div> | <pre>template <class Type> Element<Type> *MaxHeap<Type>::Pop (Element<Type> &x) { if (!n) { HeapEmpty(); return 0;} x = heap[1]; Element<Type> k = heap[n]; n--; // i : current node, j : child for (int i=1, j=2; j<=n;) { if (j<n) if (heap[j].key < heap[j+1].key) j++; // j is the larger child if (k.key >= heap[j].key) break; heap[i] = heap[j]; // move the child up i = j; j *= 2; // move down a level } heap[i] = k; return &x; }</pre> <div data-bbox="826 1720 1528 2049"> <ul style="list-style-type: none"> - n : 현재 heap 크기 - 힙이 비워져있는지 확인 - x에 루트 할당, k에 가장 끝 원소 할당하고 n-- - for문 : 루트에서 시작하여 자식 노드와 비교하며 k를 적절한 위치로 이동시킴 - 첫 번째 if문 : 더 큰 자식 노드를 선택 - 두 번째 if문 : 자식노드보다 현재 노드가 더 크면 이동할 필요가 없으므로 break </div> |

- pop, push 메소드 시간복잡도 : $O(\log_n)$ (∵ 트리의 높이만큼 실행되기 때문)

우선순위 큐(Priority Queue)

- not FIFO, 먼저 들어오는 순서가 아닌, 데이터의 우선 순위에 따라 pop한다.
- 만약 배열로 구현한다면?
 - push는 $O(1)$ 인데, pop은 $O(n)$ 이다. => $O(n)$
 - 배열삽입할 때 정렬해서 하면?
 - push $O(n)$ 이고 pop이 $O(1)$ 이므로, 여전히 $O(n)$
- 만약 힙으로 구현한다면? push는 $O(\log n)$, pop도 $O(\log n)$ => $O(\log n)$ 으로, 그냥 배열을 쓰는 것보다 더 효율적!
- 우선순위 큐 메서드
 - 1) insert(x) : heap에서 push()
 - 2) removeMax() : heap에서 pop() // min heap이면 removeMin()
 - 3) max() : 가장 우선순위가 큰 원소 찾을 // min heap이면 min()
 - 4) size()
 - 5) empty()

- 파이썬 : heapq 라이브러리 사용

<https://velog.io/@hsshin0602/%ED%8C%8C%EC%9D%B4%EC%8D%AC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84%ED%81%90priority-%ED%9E%99%ED%81%90heap>

Priority Queue 와 Heaq의 차이

queue 속 **PriorityQueue** 는 **Thread-Safe** 하고 **heapq**는 **Non-safe** 하기 때문이라고 한다.
Thread Safe 하다는 것은 반드시 확인 절차를 걸쳐야 하기 때문에 확인하는 작업때문에 더 느리다.

- Thread-Safe를 요구하는 상황에서는 PriorityQueue
- Thread-Non-Safe 해도 된다면 heapq

코딩테스트로 문제를 푸는 상황이라면 Thread-Safe를 요구하지 않기 때문에 heapq를 써야 시간초과를 면할 수 있을 것이다. 실무에서도 우선순위 큐는 대부분 heapq로 구현하고 있다.

- c++ : queue 라이브러리 사용

https://velog.io/@doorbals_512/C-priorityqueue-%EC%BB%A8%ED%85%8C%EC%9D%B4%EB%84%88-%EC%82%AC%EC%9A%A9%EB%B2%95

! push() 와 emplace() 의 차이

- 우선순위 큐에는 구조체를 많이 삽입하게 되는데, push 의 경우 오브젝트를 생성 후 삽입해야하기 때문에 불필요한 복사가 많이 발생한다.
- emplace 의 경우 오브젝트를 생성하지 않고 바로 값을 넣는다.

pair 구조체를 삽입 시 first의 값을 기준으로 정렬된다.

- 정렬 바꾸는 법

0) 우선순위 큐 선언 방법

```
priority_queue<'자료형', '구현체', '비교연산자'> 이름;
```

- 자료형 : int, double, pair 등 기본 자료형 및 구조체, 클래스 등을 사용한다.
- 구현체 : 보통 vector<자료형> 으로 구현한다. 지정하지 않으면 vector로 기본 적용.
- 비교연산자 : 비교를 위한 기준을 알려준다.

1) 내림차순 (큰 값 먼저)

```
priority_queue<int> pq;
```

- 구현체, 비교연산자를 지정하지 않고 자료형만 입력하면 기본적으로 내림차순으로 정렬이 된다.

2) 오름차순 (작은 값 먼저)

```
priority_queue<int, vector<int>, greater<int>> pq;
```

- 비교연산자에 greater<int> 를 사용하면 오름차순으로 정렬된다.

참고자료

박성빈, “알고리즘”, 고려대학교, 2024년 7월 19일

정원기, “자료구조”, 고려대학교, 2024년 7월 19일

유용재, “자료구조”, 고려대학교, 2024년 7월 19일

<https://velog.io/@hsshin0602/%ED%8C%8C%EC%9D%B4%EC%8D%AC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84%ED%81%90priority-%ED%9E%99%ED%81%90heap>
https://velog.io/@doorbals_512/C-priorityqueue-%EC%BB%A8%ED%85%8C%EC%9D%B4%EB%84%88-%EC%82%AC%EC%9A%A9%EB%B2%95