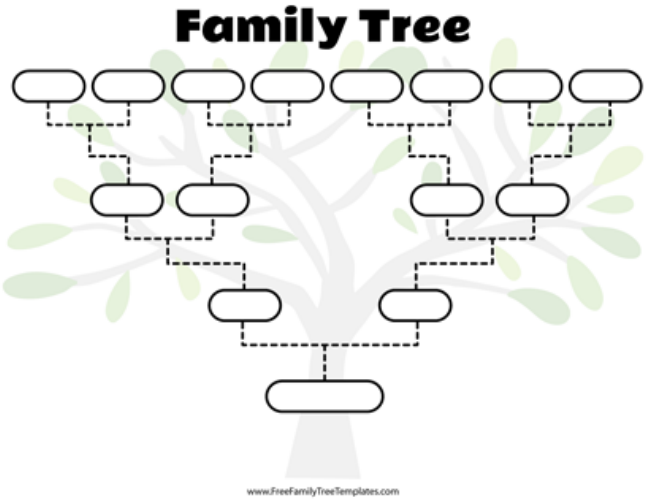
스터디 4주차 (엄태호)

트리 (Tree)



- 트리 구조는 그래프 자료구조의 일종으로 비선형 구조의 계층적인 자료구조입니다.
- 1개 이상의 유한한 개수의 노드의 집합으로 트리내에서 노드와 노드 사이의 경로는 반드시 존재해야 합니다.
- 트리구조는 실생활에서 회사의 조직도, 가계도, 파일 시스템 등에서 주로 사용됩니다.

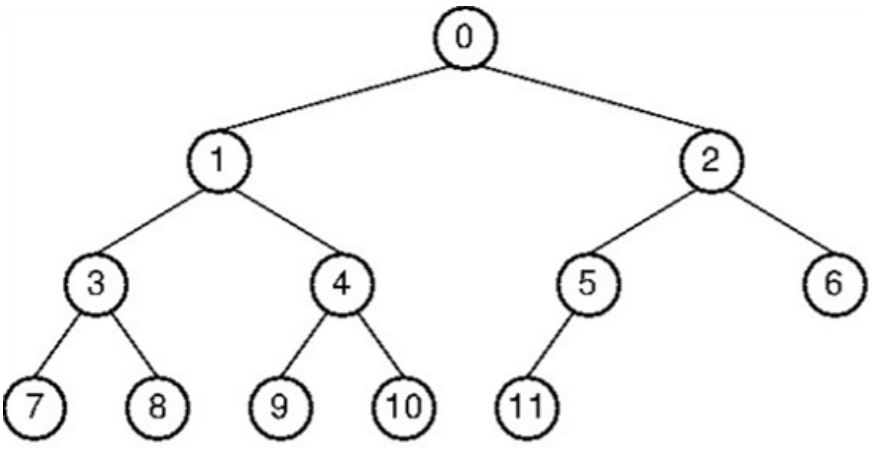
트리에서 등장하는 용어들

- **정점 (= 노드, Vertex)**
  데이터를 담고 있는 트리의 구성요소로 트리 구조에서 정점의 개수는 간선의 개수+1입니다.
  `N(V) = N(E) + 1`
  - **루트 노드 (Root Node)**
    노드 중 가장 최상단에 위치한 노드로 부모가 없는 노드를 말합니다.
    루트 노드를 기준으로 트리는 뻗어나가는데 대부분의 트리 구조에서 루트 노드는 한개만 존재합니다.
  - **잎 노드 (= 말단 노드, Leaf Node)**
    자식이 없는 노드로 일반적으로 가장 하위에 위치한 노드입니다.
  - **내부 노드 (Inner Node)**
    루트노드와 잎 노드 사이에 위치한 노드를 말합니다.
  - **형제 노드 (Sibling Node)**
    같은 부모 노드를 가진 자식 노드로 같은 레벨에 위치해 있습니다.
- **간선 (Edge)**
  노드와 노드를 이어주는 선으로 단방향, 양방향으로 나누어집니다.
  특정 노드 n에 대해 n의 바깥으로 나가는 간선은 outdegree, 들어오는 간선은 indegree라고 부릅니다.

- **깊이 (Depth)**
  루트 노드부터 특정 노드까지의 거리를 말합니다.
  트리의 깊이는 모든 노드의 깊이 중 최댓값이며 트리의 높이와 같은 값을 가집니다.
  - **레벨 (Level)**
    일반적으로 깊이와 같은 의미를 가지지만 기준점에 따라 루트노드의 레벨이 0 혹은 1이 될 수 있습니다.
- **높이 (Height)**
  깊이와는 반대로 리프 노드를 기준으로 0부터 시작합니다.
  노드의 특성에 따라 특정 노드의 깊이와 높이가 다를수는 있지만 트리의 깊이와 높이는 같습니다.
- **차수 (Degree)**
  특정 노드의 자식 노드 개수를 말합니다.
- **가중치 (Weight)**
  정점 n부터 y까지 이동하는데 드는 비용을 말합니다.
  만약, n과 y사이의 정점이 없다면 가중치가 1이 되고, 정점이 1개가 있다면 가중치는 2가 됩니다.
  물론, 이는 간선에 가중치가 1이라는 전제입니다.
- **서브트리 (Sub Tree)**
  트리 내부에서 또 다른 트리의 형태를 가진 트리를 서브트리라고 합니다.

트리 순회 방식 (Tree traversal)

- 트리를 순회하는 방식은 크게 4가지(전위, 중위, 후위, 층별)로 구분할 수 있으며
  층별 순회를 제외하고 reverse를 포함 할 경우 총 7가지로 나눌 수 있습니다.
- 이진 트리에서 노드를 순회할 때 부모 노드, 왼쪽 자식노드, 오른쪽 자식노드 3가지를 방문하게 되는데 이 때, 부모 노드를 방문하는 순서를 어디에 두는지에 따라 전위, 중위, 후위로 나뉩니다.
- 일반적으로의 트리의 순회는 왼쪽 → 오른쪽 방향으로 순회하며
  reverse의 경우 오른쪽 → 왼쪽 방향으로 순회합니다.



출처 : https://m.blog.naver.com/rlakk11/60159303809

- 전위 순회 (Pre-order)

- 전위 순회는 루트를 시작으로 왼쪽 → 오른쪽 순으로 노드를 순회하는 방식입니다.
- 순서 : 0 - 1 - 3 - 7 - 8 - 4 - 9 - 10 - 2 - 5 - 11 - 6

- 중위 순회 (In-order)

- 중위 순회는 왼쪽 가장 하단에 위치한 리프 노드부터 부모노드, 오른쪽 노드 순서로 순회하는 방식입니다.
- 시작할 때 루트에서 왼쪽 리프노드를 찾을 때까지 먼저 쭉 내려간 뒤 해당 리프노드부터 순회를 시작합니다.
- 순서 : 7 - 3 - 8 - 1 - 9 - 4 - 10 - 0 - 11 - 5 - 2 - 6

- 후위 순회 (Post-order)

- 후위 순회는 왼쪽 노드 → 오른쪽 노드 → 부모노드 순으로 순회하는것을 말합니다.
- 순서 : 7 - 8 - 3 - 9 - 10 - 4 - 1 - 11 - 5 - 6 - 2 - 0

- 층별 순회

- 루트노드부터 왼쪽 노드 → 오른쪽 노드 방향으로 레벨별로 차례대로 순회하는 방식을 말합니다.
- 순서 : 0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11

이진트리 (Binary Tree)

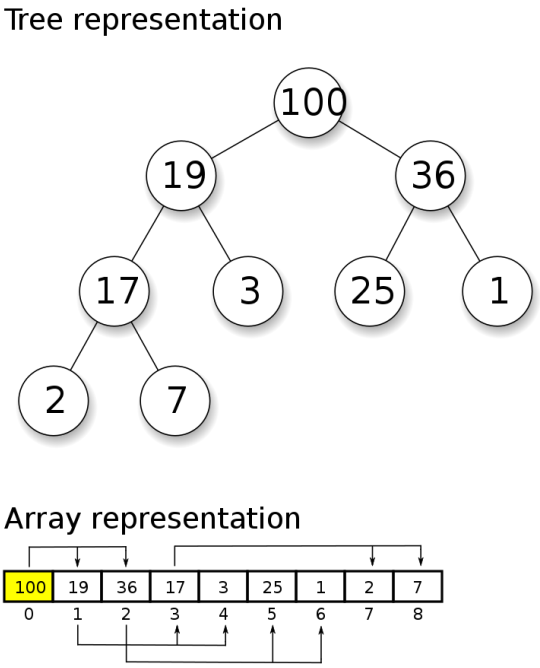
- 이진 트리는 효율적인 탐색을 위해 만들어진 자료구조로 탐색 시 평균 `O(log n)`의 시간복잡도를 가진다는 특징이 있습니다.
- 효율적인 탐색을 위해 차수(자식 노드의 개수)를 2로 제한하였으며 트리의 균형과 노드의 순서가 시간복잡도에 영향을 미치기 때문에 이를 유지하는게 중요합니다.
- 다만, 제한적인 구조를 가지고 있으며 노드당 2개의 포인터가 있어 트리의 구조가 커짐에 따라 메모리 사용량이 증가한다는 점, 삽입과 삭제과정이 다소 복잡하며 순회가 어렵다는 단점이 있습니다.
- 이진 트리는 노드를 삽입할 때 왼쪽과 오른쪽을 구분하며 **왼쪽부터 채워야합니다.**
- 또한, 트리의 차수가 2로 제한되어있다는것은 0~2개 까지의 자식노드를 가질 수 있다는것이며
  트리 내부에 노드가 없는 경우도 포함됩니다.

- 이진 트리의 종류

- **정 이진 트리 (Full Binary Tree, Strictly Binary Tree)**
  모든 노드의 자식 개수가 0 혹은 2인 트리로 자식이 1개인 노드가 없는 트리입니다.
- **완전 이진 트리 (Complete Binary Tree)**
  마지막 레벨을 제외하고 모두 노드가 채워져있는 트리로 왼쪽에서부터 채워집니다.
  만약, 왼쪽이 다 채워져있는데 오른쪽에 노드가 있는 경우 완전 이진트리라고 할 수 없습니다.
- **포화 이진 트리 (Perfect Binary Tree)**
  리프노드가 모두 꽉 차있어 새로운 노드를 추가할 때 깊이를 추가해야만 하는 경우를 포화 이진트리라고 한합니다.
- **변질 이진 트리 (Degenerate Tree, Pathological Tree)**
  모든 부모 노드가 하나의 자식만 가지고 있는 트리를 말하며 이 중에서도 자식노드가 한 쪽 방향으로만 쏠려있는 경우 편향 이진트리(Skewed Binary Tree)라고 합니다.

- **균형 이진 트리 (Balanced Binary Tree)**
  모든 노드의 왼쪽과 오른쪽 하위트리의 높이 차가 최대 1인 이진 트리를 말합니다.

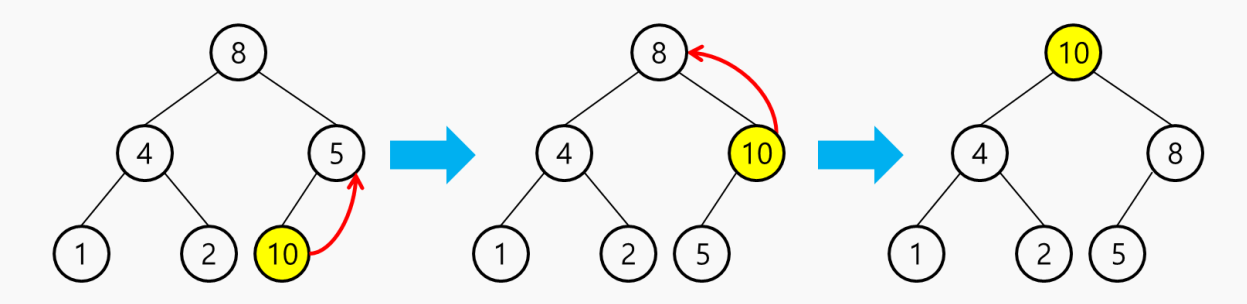
힙 (Heap)



- 힙은 이진 트리의 일종으로 **부모 노드는 항상 자식노드보다 크거나 작다는 특징**을 가지고 있어 최댓값 또는 최솟값을 구할 때 유용합니다.
- 최댓값 또는 최솟값에 접근할 때는 `O(1)`, 삽입 및 삭제시 `O(log n)`의 시간복잡도를 가집니다.
- 우선순위 큐를 구현하는데 유용한 자료구조로 루트노드의 값이 최솟값이냐 최댓값이냐에 따라 최소 힙, 최대 힙으로 나눌 수 있습니다.

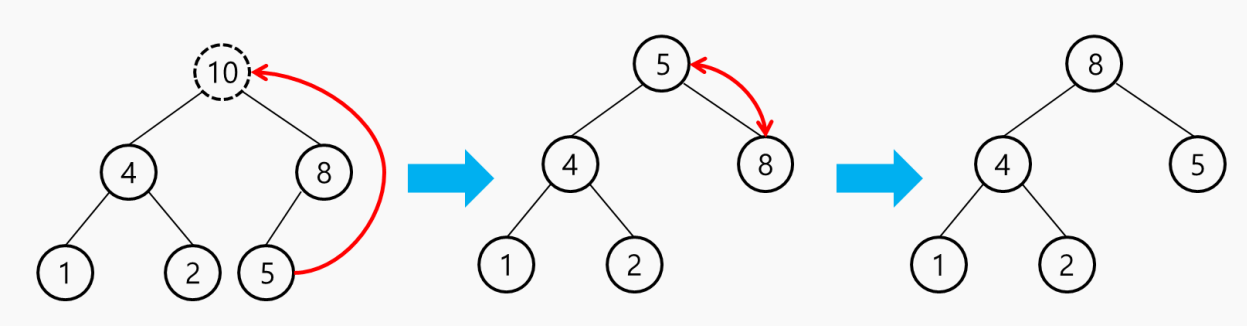
힙의 동작방식

- 삽입 과정 (Heapify Up)



- 힙 자료구조에 요소를 삽입 할 때는 가장 하위 레벨에서 왼쪽부터 삽입합니다.
- 이 때, 최대힙인 경우 자신의 부모노드와 값을 비교해 자신의 값이 더 크다면 위치를 변경해나가면서 만약 자신보다 큰 값의 부모를 만나거나 루트노드에 위치 할 경우 정렬과정이 종료됩니다.

- 삭제 과정 (Heapify Down)



- 삭제 과정에서는 대상이 되는 요소가 루트 노드이므로 먼저 루트 노드를 삭제하고 가장 오른쪽에 위치한 리프 노드를 루트 노드의 자리로 올립니다.
- 이 후, 자신의 자식 노드들과 값을 비교하며 정렬과정이 이루어집니다.

힙 구현해보기

- 일반적으로 힙은 배열로 구현하기 때문에 배열을 활용한 최소힙을 구현해보았습니다.

```
class MinHeap {
    #heap = [null];

    /* 요소가 없다면 null을 반환 */
    getMin() {
        return this.#heap[1] || null;
    }

    swap(fIndex, sIndex) {
        [this.#heap[fIndex], this.#heap[sIndex]] = [this.#heap[sIndex], this.#heap[fIndex]];
    }

    insert(node) {
        this.#heap.push(node);

        /* Heapify Up */
        if (this.#heap.length > 2) {
            let currIdx = this.#heap.length-1;
            let parentIdx = Math.floor(currIdx/2);

            while (currIdx > 1 && this.#heap[parentIdx] > this.#heap[currIdx]) {
                [this.#heap[parentIdx], this.#heap[currIdx]] = [this.#heap[currIdx], this.#heap[parentIdx]];
                currIdx = parentIdx;
                parentIdx = Math.floor(currIdx/2);
            }
        }
    }

    remove() {
        const minData = this.#heap[1];
```

```
        /* Heapify Down */
        if (this.#heap.length > 2) {
            this.#heap[1] = this.#heap.pop();

            let currIndex = 1, leftIndex = currIndex * 2, rightIndex = currIndex * 2 + 1;

            while (this.#heap[leftIndex] && this.#heap[rightIndex] &&
                    (this.#heap[currIndex] > this.#heap[leftIndex] || this.#heap[currIndex] > this.#heap[rightIndex])
            ) {
                if (this.#heap[leftIndex] < this.#heap[rightIndex]) {
                    this.swap(currIndex, leftIndex);
                    currIndex = leftIndex;
                } else {
                    this.swap(currIndex, rightIndex);
                    currIndex = rightIndex;
                }

                leftIndex = currIndex * 2, rightIndex = currIndex * 2 + 1;
            }
        } else if (this.#heap.length === 2) {
            // 데이터가 하나만 있는 경우
            this.#heap.pop();
        } else {
            // 어떤 데이터도 없는 경우
            return null;
        }

        return minData;
    }

    /**
    * 메소드 호출 시 힙이 어떻게 변화하는지를 확인하기 용이하도록 임의로 추가한 메소드로
    * 실제 힙 자료구조에 존재하지 않는 메소드입니다.
    */
    getData() {
        return this.#heap;
    }
}

const heap = new MinHeap();

/* Test Code */
heap.insert(4);
console.log(heap.getData()); // 4
heap.insert(1);
console.log(heap.getData()); // 1, 4
heap.insert(7);
console.log(heap.getData()); // 1, 4, 7
heap.insert(3);
console.log(heap.getData()); // 1, 3, 7, 4
heap.insert(2);
console.log(heap.getData()); // 1, 2, 7, 4, 3

heap.remove();
console.log(heap.getData()); // 2, 3, 7, 4
heap.remove();
console.log(heap.getData()); // 3, 4, 7
```