27th

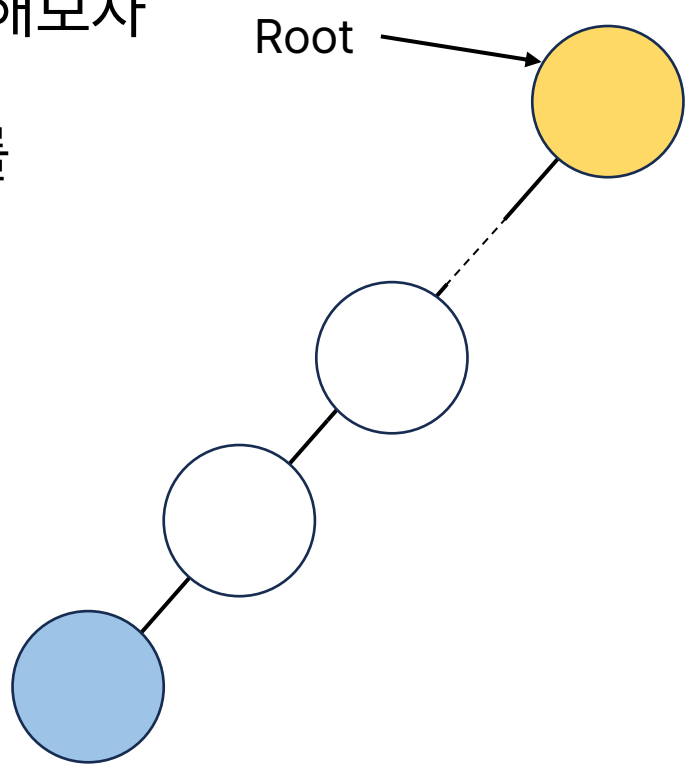
LCA

LCA

- 우선 순회를 하며 각 노드의 깊이와 부모를 알아낸다
- 두 노드의 높이를 맞추기 위해 낮은 노드를 높이 있는 노드의 위치까지 올린다
- 동일한 조상이 나올 때까지 두 노드를 하나씩 올린다

LCA

- 노란색 노드와 파란색 노드의 LCA를 찾는다 해보자
- 파란색 노드의 조상을 살펴보려면 모든 노드를 보아야 한다



LCA

- 최악의 경우 모든 노드를 탐색해야 할 수 있다
- 따라서 노드의 개수가 N개 일 때, $O(N)$의 시간 복잡도를 따른다

LCA

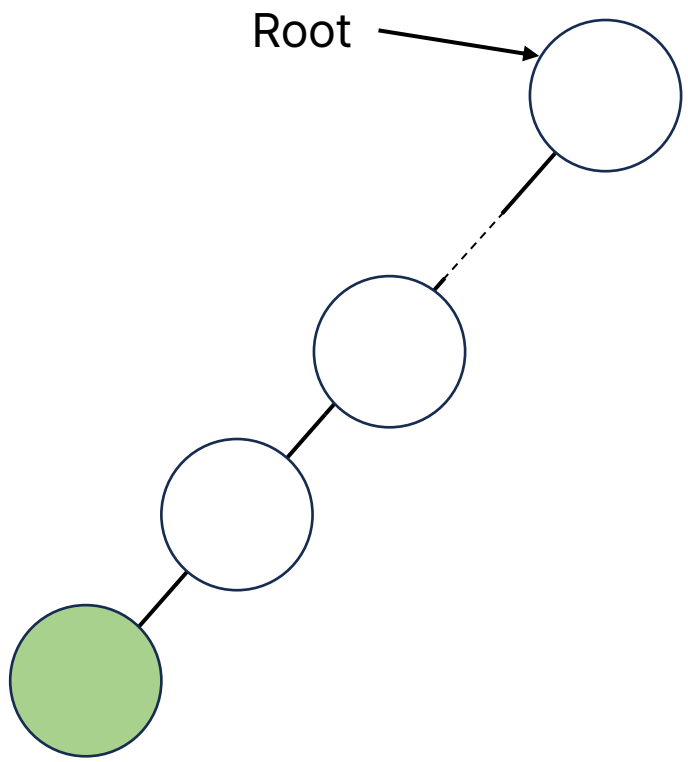
- 시간을 줄이려면 추가적인 정보를 기억하고 있어야 한다
- 전처리 과정이 필요하다
- 우리는 모든 노드의 조상을 기억하는 표를 만들어 기억할 것이다
- 모든 조상을 기억하는 것은 너무 많은 메모리를 사용한다

LCA

- 시간을 줄이려면 추가적인 정보를 기억하고 있어야 한다
- 전처리를 통하여 미리 몇가지 정보를 기억하고 있는다
- 우리는 모든 노드의 조상을 기억하는 표를 만들어 기억할 것이다
- 모든 조상을 기억하는 것은 너무 많은 메모리를 사용한다

LCA

- 다음과 같은 모양의 트리가 존재한다고 생각하자
- 모든 조상을 기억한다면 초록색 노드는 모든 노드를 저장할 공간이 필요하다
- $O(N^2)$의 공간 복잡도를 따르게 된다
- 따라서 모든 조상을 기억할 수 없다
- Sparse Table을 사용한다



Sparse Table

- 희소 테이블, 모든 데이터가 아닌 일부의 데이터만 저장하는 표이다
- 코드로만 생각하면 2차원 배열이다(int table[][])
- table[i][j]는 i번 노드의 2^j번째 조상을 저장한다
- 부모는 i번 노드의 1번째 조상이므로 table[i][0]에 들어간다($2^0 = 1$)

Sparse Table

- 노드가 N개 있다고 생각할 때 트리의 높이가 제일 큰 경우는 모든 노드가 일렬로 연결된 경우이다
- 따라서 노드가 N개 있다면 제일 아래에 있는 노드는 최악의 경우 약 N개의 조상이 존재한다
- 일렬로 모든 노드가 존재한다고 생각하고 제일 아래에 있는 노드가 모든 조상노드를 표현하기 위한 공간을 생각해보자

Sparse Table

- 2의 거듭제곱으로 올라가므로 int table[MAX_NODE][J]에서 J가 2라면 4번째 조상까지, 3이라면 8번째 조상까지 표현이 가능하다
- 반대로 노드의 개수가 5~8개면 J가 최소 3만큼 필요하고 9~16이면 J가 최소 4만큼 필요하다
- 따라서 J는 최소 $[\ log_2 N\ ]$만큼의 공간만 필요하다

Sparse Table

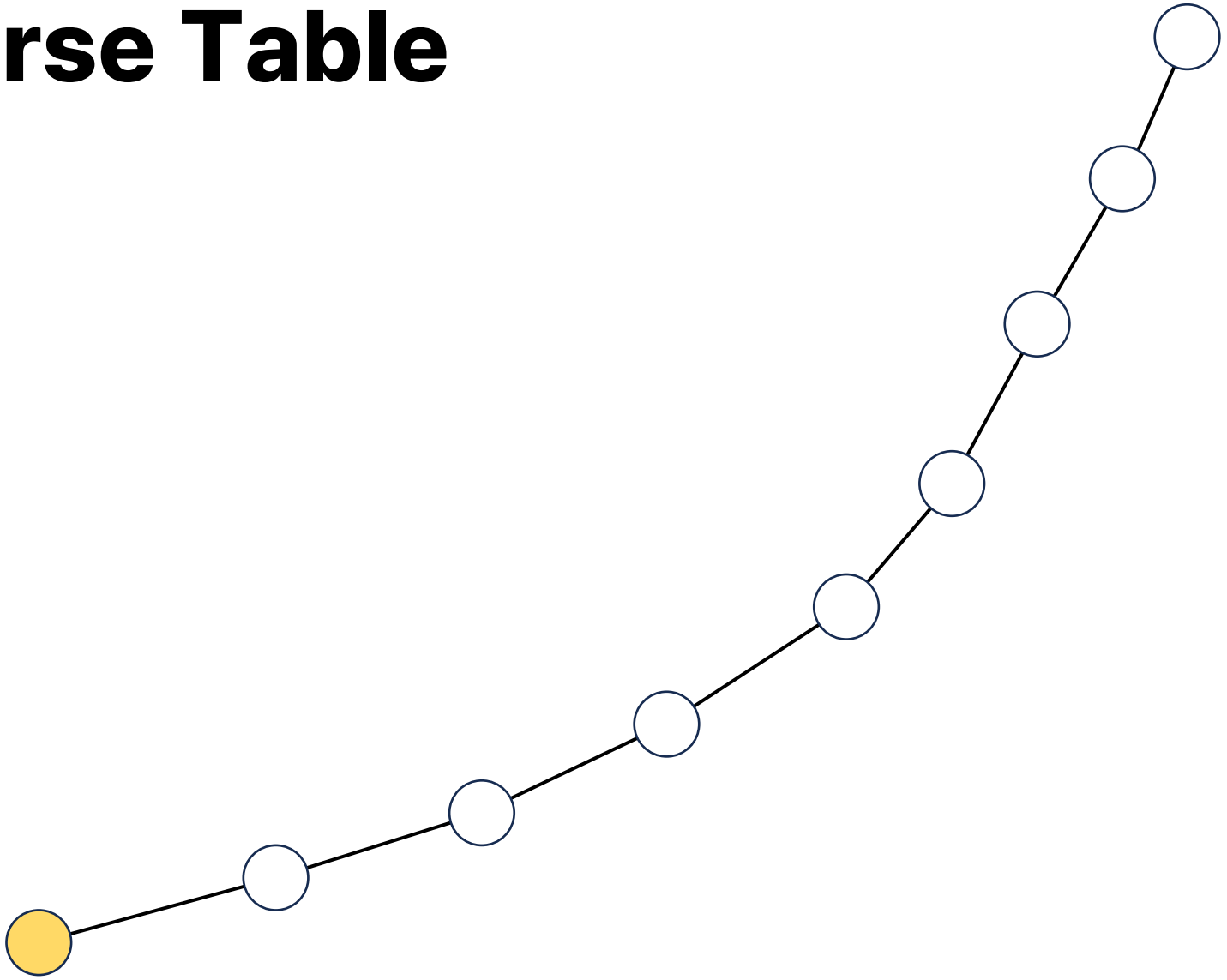
- Sparse Table을 채워보자
- 트리의 부모와 자식 관계가 주어진 경우 table[node][0]에 부모 노드를 채우고 그렇지 않다면 순회를 하며 각 노드의 부모를 찾아서 채우자
- 그 이후에는 table[i][j + 1] = table[table[i][j]][j]로 나머지를 채울 수 있다

Sparse Table

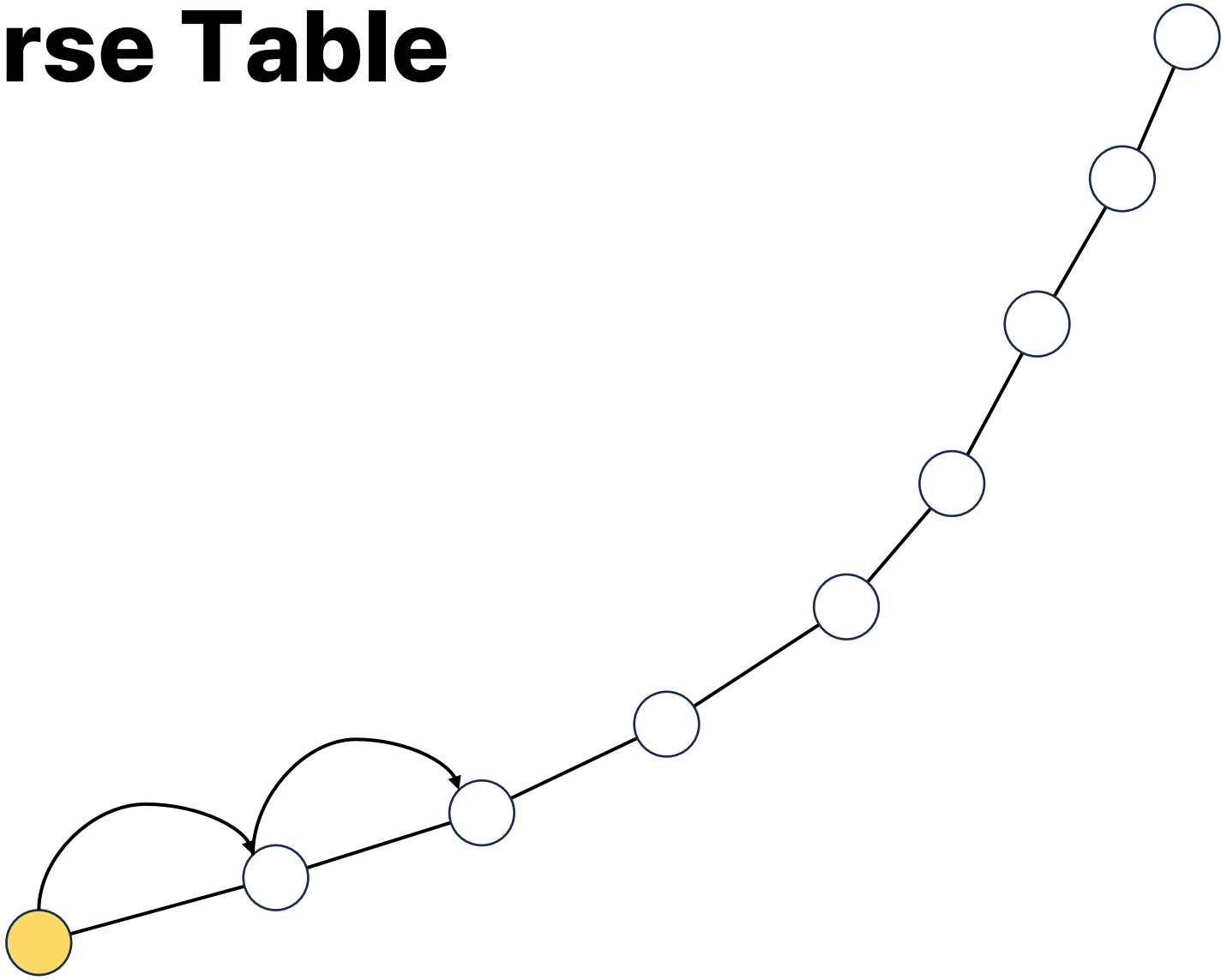
- 직접 숫자를 넣어 생각해보자

- i번 노드의 2번째 조상은 i번 노드의 부모의 부모이다
- i번 노드의 4번째 조상은 i번 노드의 조부모(2번째 조상)의 조부모이다
- i번 노드의 8번째 조상은 i번 노드의 4번째 조상의 4번째 조상이다

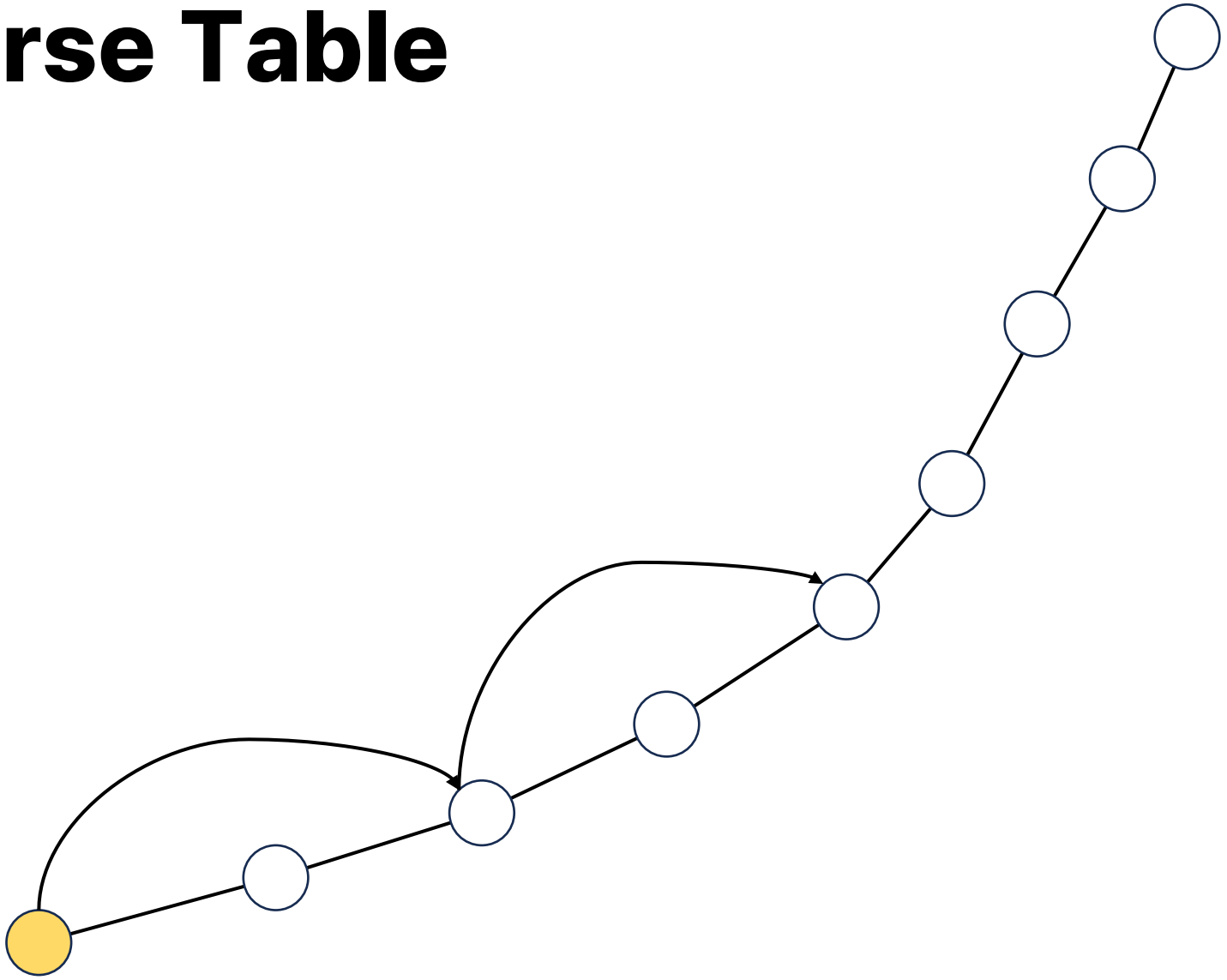
Sparse Table



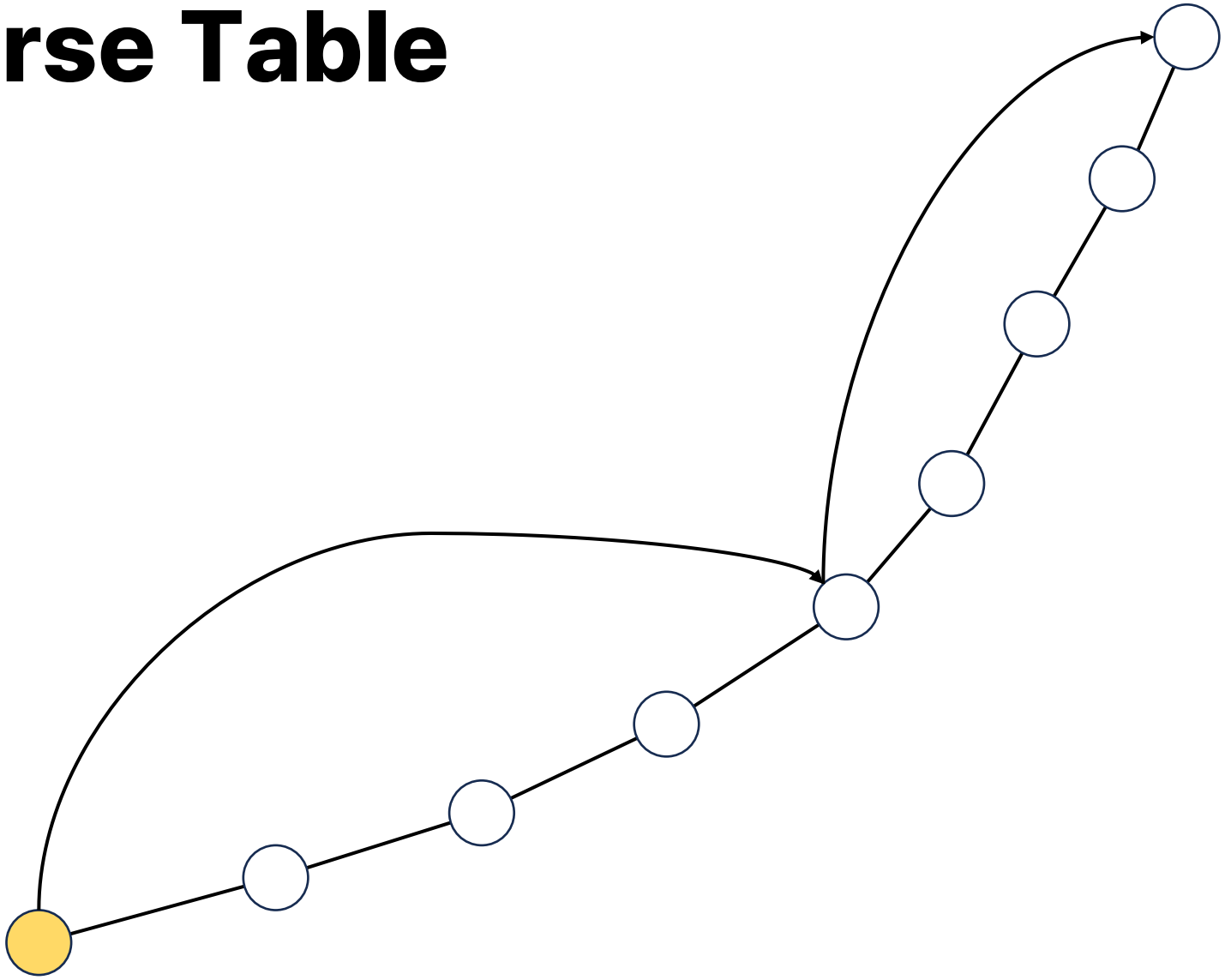
Sparse Table



Sparse Table



Sparse Table



Sparse Table

- 우리는 i번 노드의 k번째 조상을 직접 구하는 것이 아닌 k/2번째 조상의 k/2번째 조상을 구하면 i번째 노드의 k번째 조상을 구할 수 있다

- 이것을 2의 거듭제곱 형태로 표현해보자

Sparse Table

- 희소 테이블에서 표에 채운 값은 2^j번째 조상이다
- 2^j번째 조상을 이용해서 $2^j + 2^j = 2^{j+1}$ 번째 조상을 구할 수 있다

- 따라서 모든 i,모든 노드에 대하여 table[i][j]를 알고 있다면 table[i][j+1]을 알 수 있다

Sparse Table

- 2^j번째 조상의 2^j번째 조상이 2^{j+1}번째 조상이다

➔ 2^j번째 조상의 2^j번째 조상이 2^{j+1}번째 조상이다

- table[i][j + 1] = table[table[i][j]][j]

Sparse Table

- 제일 처음에 2^0 = 1번째 조상인 부모 노드를 모두 표에 채웠다

	0	1	2	3	4	5	6
0		-	-	-	-	-	-
1		-	-	-	-	-	-
2		-	-	-	-	-	-
3		-	-	-	-	-	-

Sparse Table

- 2^0번째 조상들을 모두 알고 있으므로 모든 노드의 2^1번째 조상을 알 수 있다

	0	1	2	3	4	5	6
0			-	-	-	-	-
1			-	-	-	-	-
2			-	-	-	-	-
3			-	-	-	-	-

Sparse Table

- 이후 재귀적으로 모든 표를 채울 수 있다

	0	1	2	3	4	5	6
0							
1							
2							
3							

Sparse Table

```
int p[MAX_NODE][18];

void get_sparse_table() {
    for (int j = 1; j < 18; j++)
        for (int i = 1; i <= MAX_NODE; i++)
            p[i][j] = p[p[i][j - 1]][j - 1];
}
```

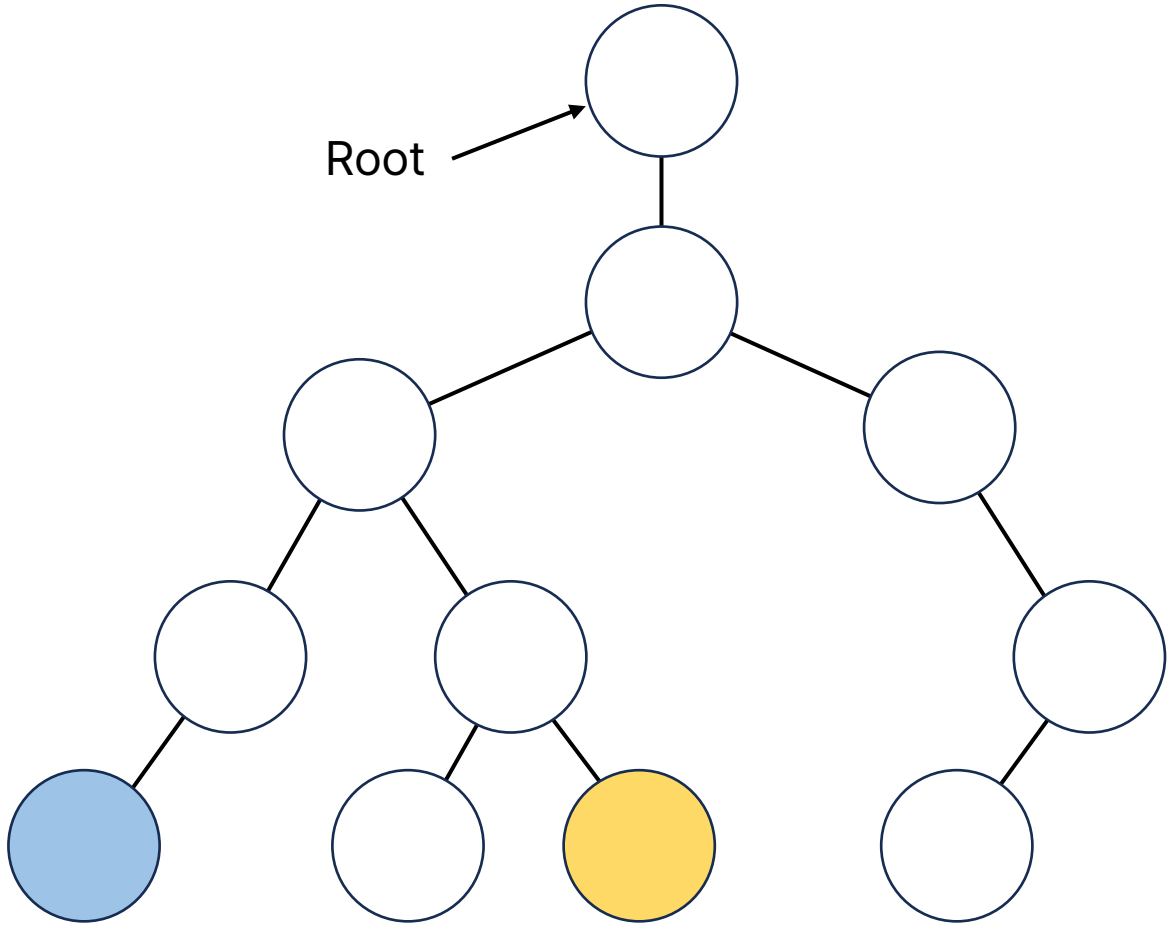
Sparse Table

```
void dfs(int cur) {
    for (auto child : edges[cur]) {
        if (depth[child] != -1)
            continue;
        depth[child] = depth[cur] + 1;
        p[child][0] = cur;
        dfs(child);
    }
}
```

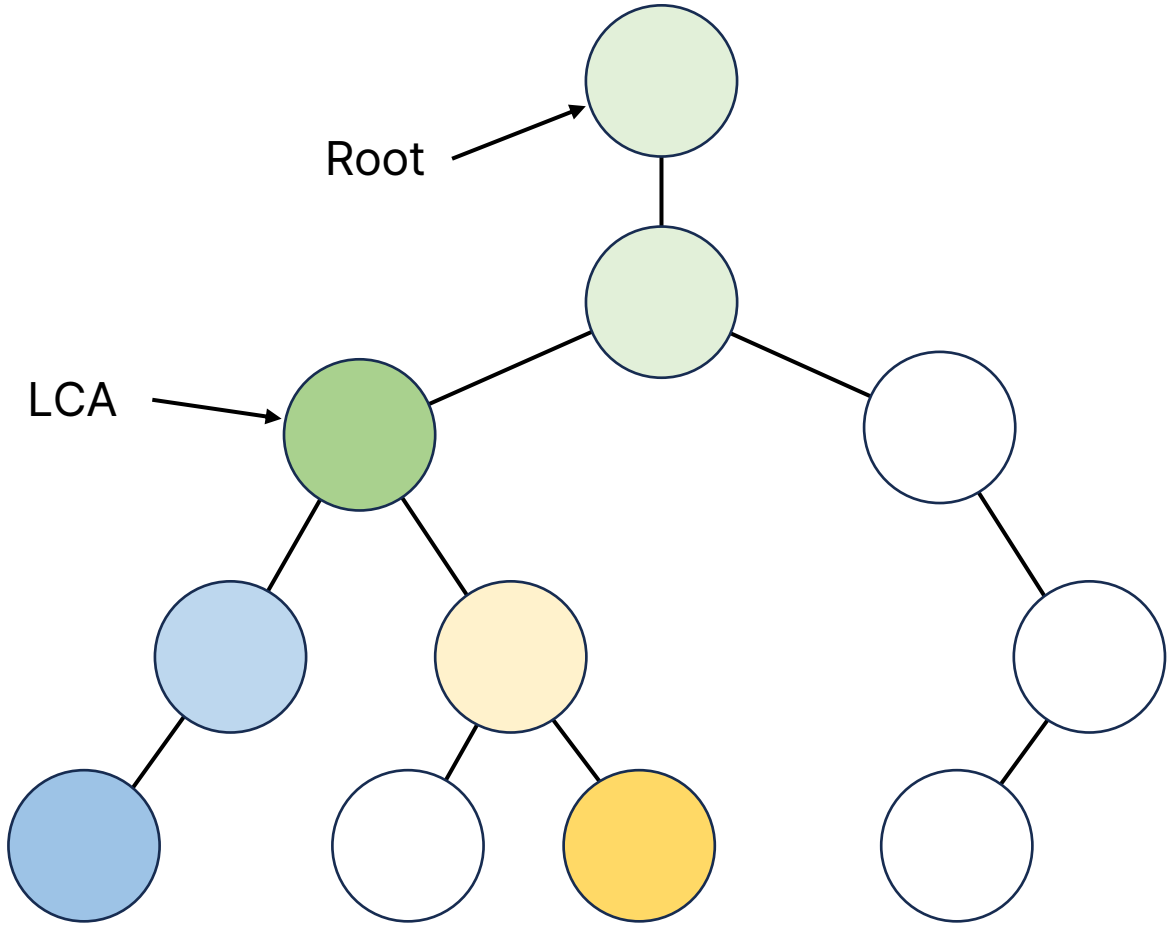
LCA

- LCA를 찾는 첫 과정은 우선 두 노드를 동일한 높이까지 올리는 것이다
- 최소 공통 조상이므로 두 노드가 동일한 높이에 있지 않다면 두 노드 사이에는 최소 공통 조상이 존재하지 않는다

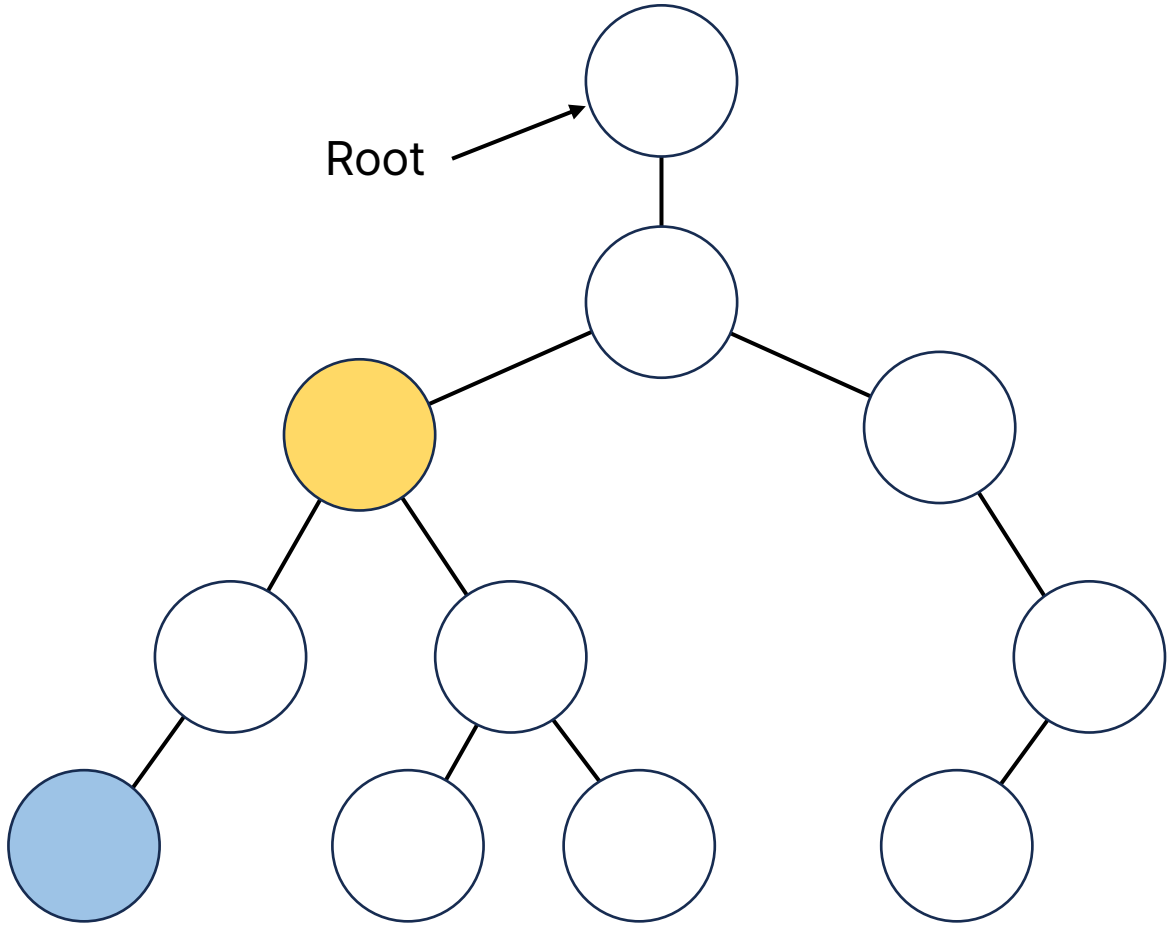
LCA



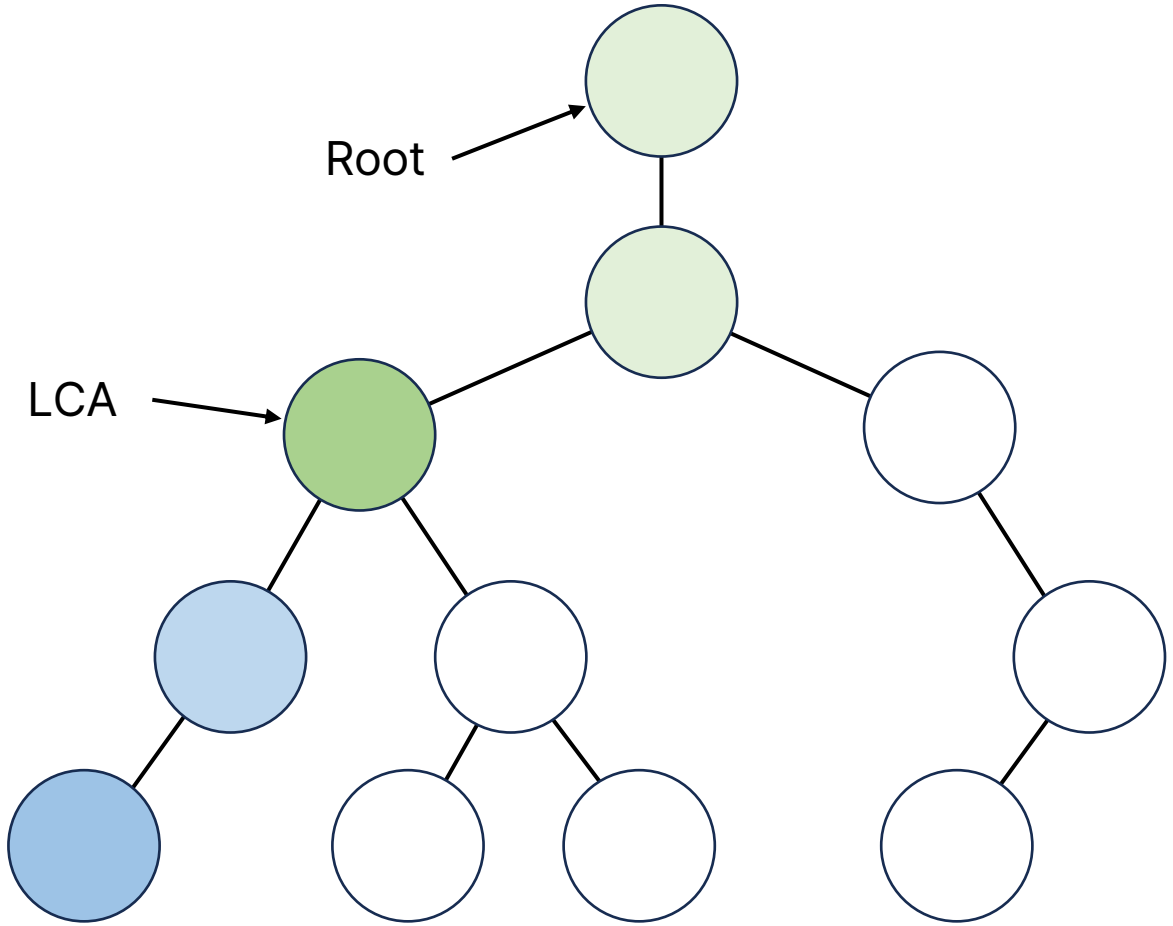
LCA



LCA



LCA



LCA

- 따라서 두 노드의 높이가 다르다면 더 낮은 노드를 높은 높이에 있는 노드 위치까지 끌어 올리는 것이다

- 한 칸씩 올리는 것은 매우 비효율적이다
- Sparse Table과 비트 연산을 이용하면 빠르게 올릴 수 있다

LCA

- 제일 처음 DFS를 수행하면서 각 노드의 깊이를 알아낸다
- 두 노드의 깊이의 차만큼 더 낮은 노드를 올려야 한다

- ex) a노드의 깊이가 4이고 b노드의 깊이가 7이라면 b노드를 3칸 올려야 한다

LCA

- 3칸 올리는 것을 예시로 생각해보자
- 3을 이진수로 표현하면 0011_2이다
- 3 = 2+1로 표현할 수 있다
- Sparse Table에서는 2의 거듭제곱에 해당하는 조상을 기억하고 있다
- Sparse Table에서 표를 채우는 것처럼 3칸 올리는 것을 b노드의 1번째 조상의 2번째 조상으로 해결할 수 있다

LCA

- 더 큰 수에 대해서 생각해보자
- 127칸 만큼 차이난다고 가정하는 경우 01111111_2만큼 차이가 난다
- 127은 64 + 32 + 16 + 8 + 4 + 2 + 1로 나타낼 수 있다
- 이는 각 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0번째 조상이므로 Sparse Table에서 이미 구해놨다
- 이를 table[b][0]의 table[?][1]의 table[?][2]의 ... table[?][6]으로 구할 수 있다
- 따라서 127번 올라가는 것을 7번으로 줄일 수 있다

LCA

```
int lca(int a, int b) {
    if (dep[a] < dep[b])
        swap(a, b);
    int diff = dep[a] - dep[b];
    for (int i = 0; diff; i++) {
        if (diff & 1)
            a = p[a][i];
        diff >>= 1;
    }
}
```

LCA

- a의 깊이가 1이고 b의 깊이가 101이라고 해보자
- 우선 깊이가 더 큰 것을 a에 두기 위해 if문과 swap을 이용한다

```
if (dep[a] < dep[b])
        swap(a, b);
```

LCA

- 둘의 깊이 차이를 구하자
- 그리고 이진수로 살펴보기 위해 비트 연산을 사용하자

```
 int diff = dep[a] - dep[b];
for (int i = 0; diff; i++) {
    if (diff & 1)
        a = p[a][i];
    diff >>= 1;
}
```

LCA

- 둘의 차이는 100이다
- 100을 이진수로 나타내면 다음과 같다

0	1	1	0	0	1	0	0

LCA

- a를 b와 같은 높이로 만들기 위해서는 a를 $2^6+2^5+2^2$조상으로 올라가야 한다
- Sparse Table을 이용하여 a = p[a][2]; a = p[a][5]; a = p[a][6];으로 b와 같은 높이로 올라갈 수 있다

0	1	1	0	0	1	0	0

LCA

- for문에서 i값이 증가하면서 오른쪽으로 한 칸씩 shift된다
- i값에 따라서 제일 오른쪽 끝에 어떤 비트가 와 있는지 알 수 있다

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

LCA

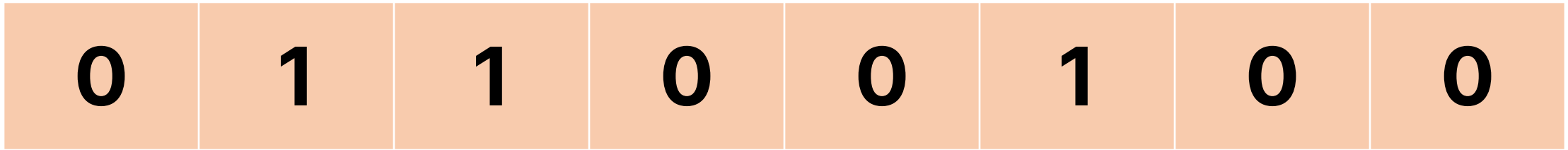
- i가 4라면 제일 오른쪽 비트는 0임을 알 수 있다

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

LCA

- 이 내용을 이용해 i번 shift 했을 때 제일 우측 비트가 1이라면 Sparse Table을 이용해 올라간다

```
for (int i = 0; diff; i++) {
    if (diff & 1)
        a = p[a][i];
    diff >>= 1;
}
```



LCA

- i가 0일 때, 가장 우측 비트가 0이므로 올라가지 않는다

0	1	1	0	0	1	0	0

LCA

- i가 1일 때, 가장 우측 비트가 0이므로 올라가지 않는다

0	0	1	1	0	0	1	0

LCA

- i가 2일 때는 가장 우측 비트가 1이다
- a = p[a][2];로 2^2만큼 올라간다

0	0	0	1	1	0	0	1

LCA

- i가 5일 때는 가장 우측 비트가 1이다
- a = p[a][5];로 2^5만큼 올라간다
- $2^5 + 2^2$ = 36만큼 올라왔다

0	0	0	0	0	0	1	1

LCA

- i가 6일 때는 가장 우측 비트가 1이다
- a = p[a][6];로 2^6만큼 올라간다
- $2^6 + 2^5 + 2^2$ = 100만큼 올라왔다

0	0	0	0	0	0	0	1

LCA

- 한 번 더 Shift하게 되면 diff값이 0이 된다
- 더 이상 올라가지 않아도 된다는 것을 의미하므로 반복문이 종료된다

0	0	0	0	0	0	0	0

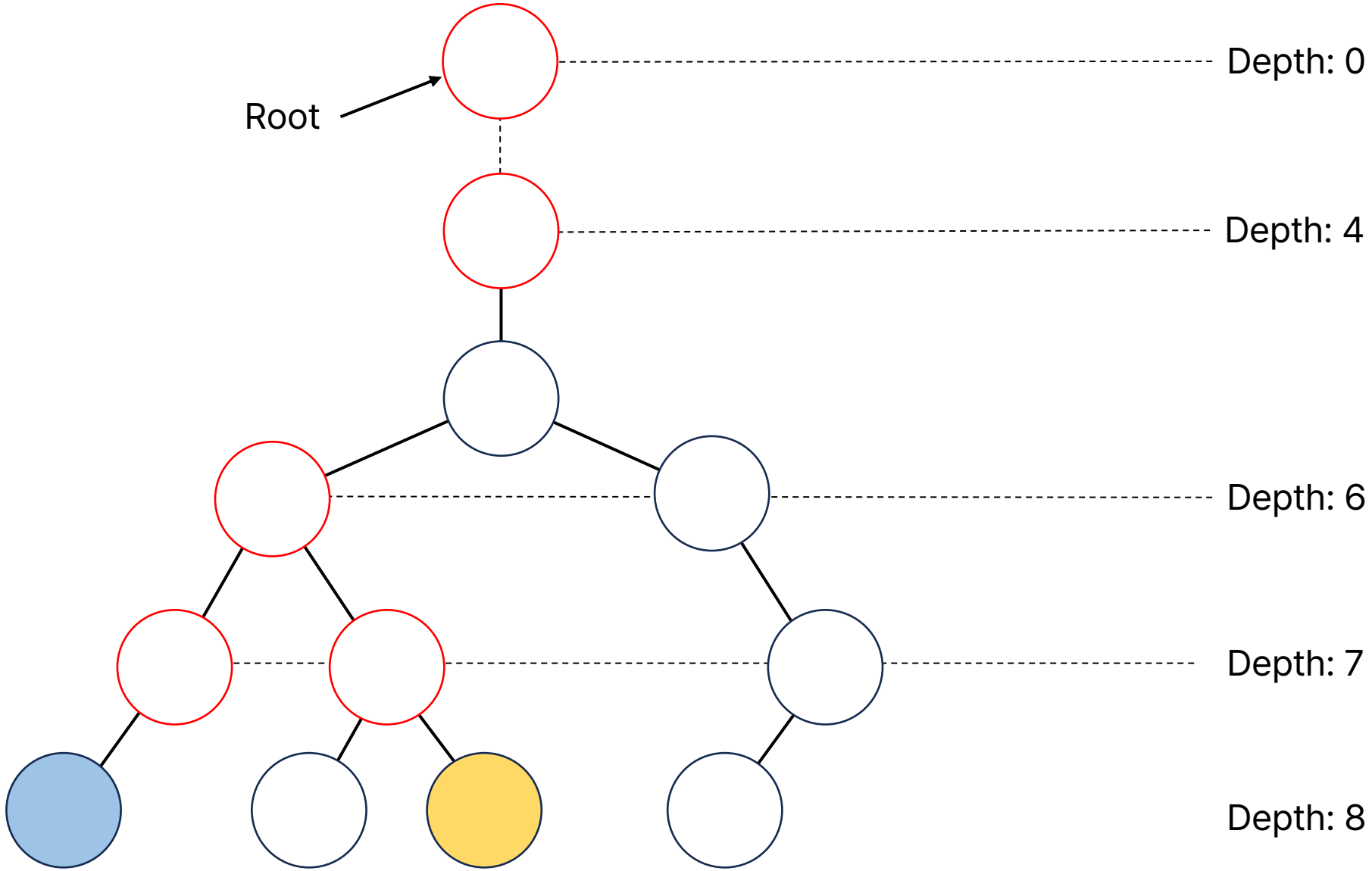
LCA

- 높이를 맞춘 이후를 생각하자
- 마찬가지로 하나씩 올라가는 것은 비효율적이다
- Sparse Table의 정보를 이용하자

LCA

- Sparse Table의 저장된 정보는 각 노드의 2^k번째 조상이 저장되어 있다
- 이를 이용해 각 노드의 조상들을 트리에서 살펴볼 수 있다

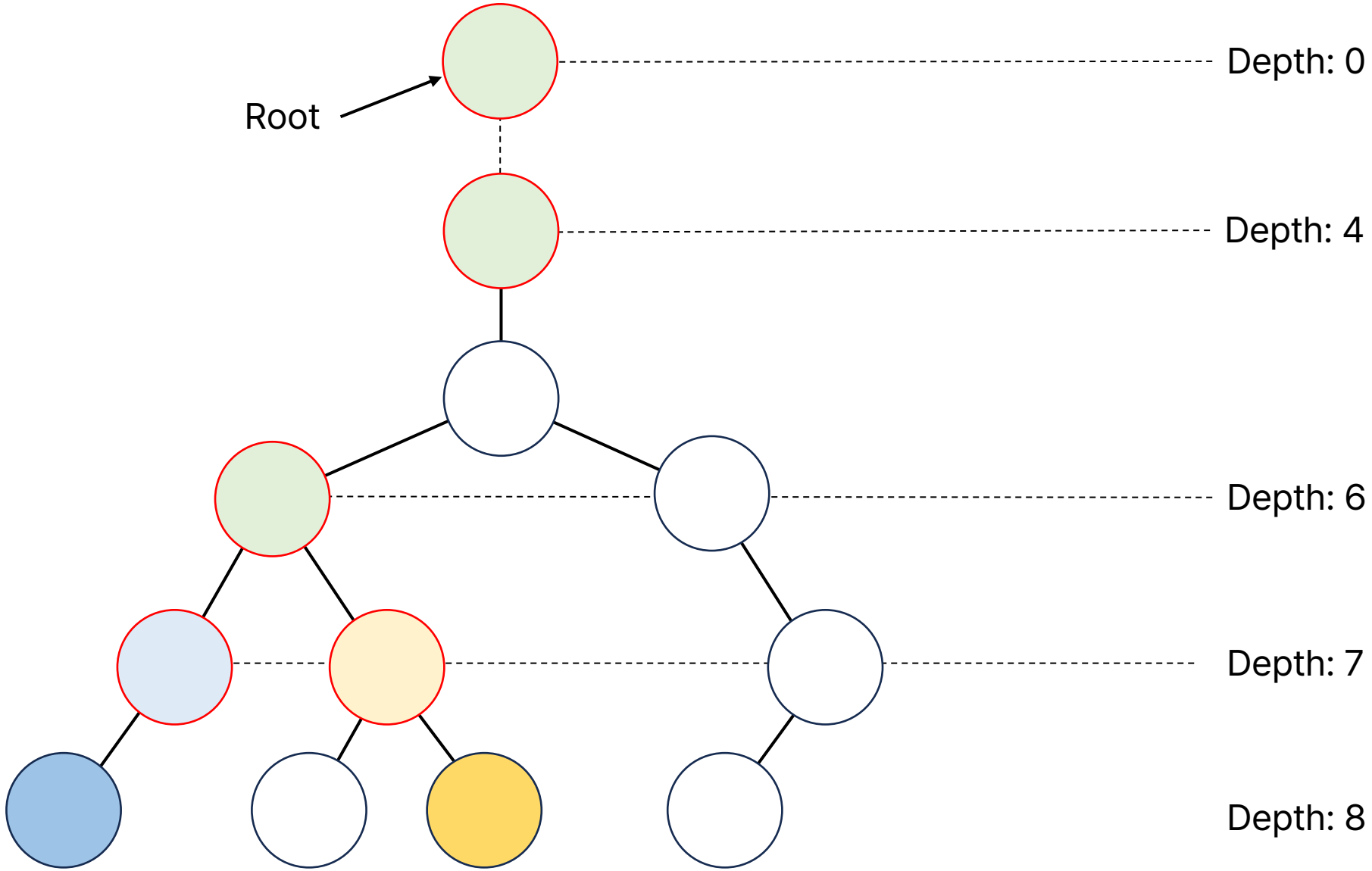
LCA



LCA

- 두 노드의 깊이가 8이라고 했을 때
- table[노드][1]에는 깊이 7의 조상이 저장되어 있다
- table[노드][2]에는 깊이 6의 조상이 저장되어 있다
- table[노드][3]에는 깊이 4의 조상이 저장되어 있다
- table[노드][4]에는 깊이 0의 조상이 저장되어 있다

LCA

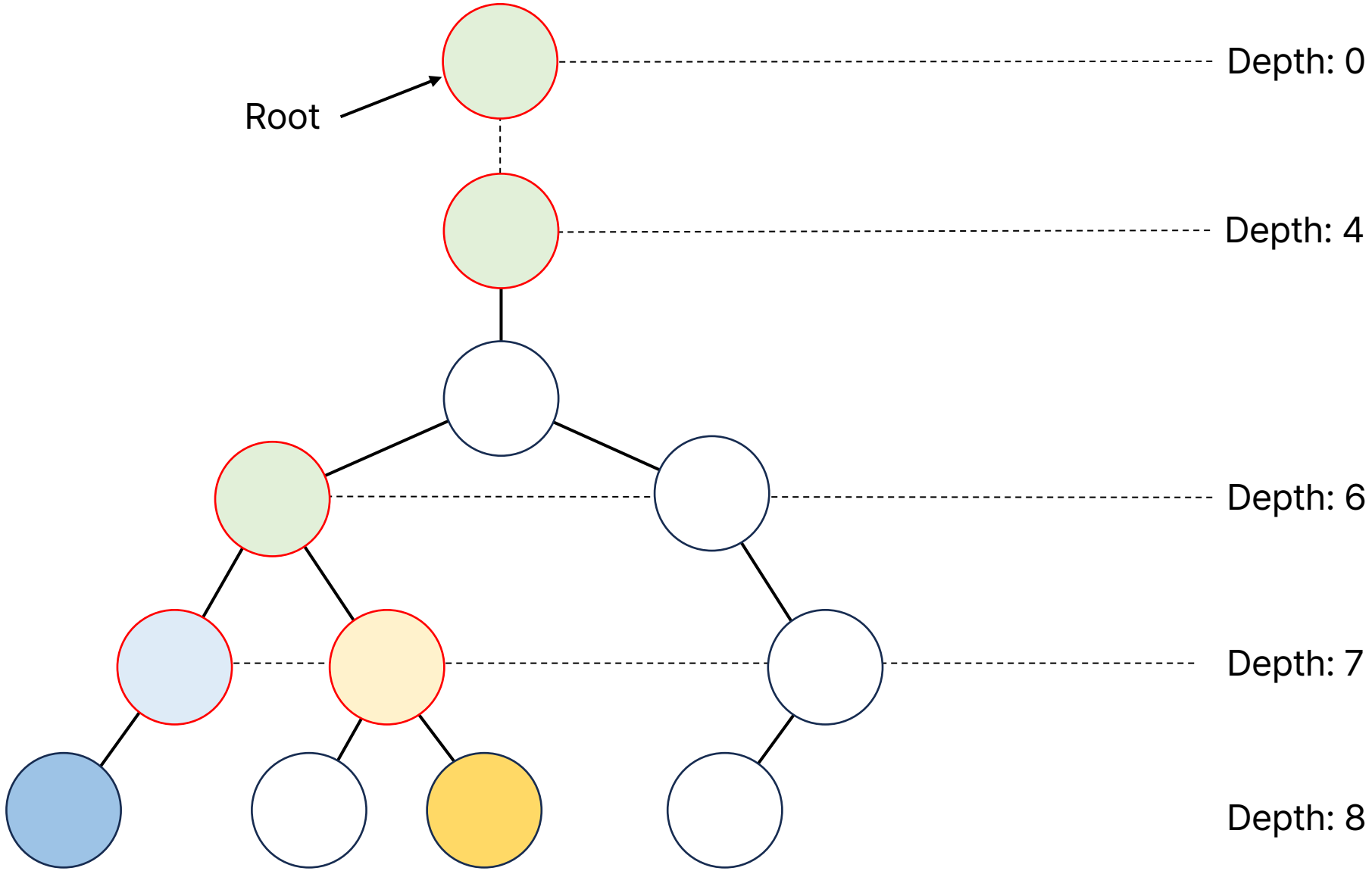


LCA

- 조상을 살펴봤을 때 LCA의 높이보다 높거나 같다면 두 노드의 조상은 동일할 것이다
- 반대로 LCA보다 낮은 높이의 조상이라면 두 노드의 조상은 다를 것이다

- 이 정보를 살펴보면 어느 구간에 LCA가 존재하는지 살펴볼 수 있다

LCA



LCA

- Depth 0, 4, 6에서 두 노드의 조상이 같으므로 Depth 6, 7, ... 에 LCA가 존재하는 것을 알 수 있다
- Depth 7에서 두 노드의 조상이 다르므로 Depth 6에 LCA가 존재하는 것을 알 수 있다

LCA

- Sparse Table로 나타내면 table[a][j]와 table[b][j]는 다르지만 table[a][j+1], table[b][j+1]이 동일한 경우 이 두 노드의 깊이 사이에 LCA가 존재하는 것이다
- 따라서 두 값이 달라지는 지점을 찾은 다음 해당 범위에서 LCA를 찾으면 된다

LCA

- 제일 위, 루트 또는 그 이상은 동일하다
- 따라서 위에서부터 내려오면서 값을 탐색하다 달라지는 시점을 확인한다
- 달라지는 시점부터 부모로 계속 올라가면서 동일해지는 시점을 찾는다
- Sparse Table은 2의 거듭제곱의 조상관계를 찾으므로 저장된 조상의 차이 중 제일 큰 것은 log N이다
- 따라서 최대 log N개만 살펴보면 된다

LCA

```
for (int i = 17; i >= 0; i--) {
    if (p[a][i] != p[b][i]) {
        a = p[a][i];
        b = p[b][i];
        break;
    }
}
while (a != b) {
    a = p[a][0];
    b = p[b][0];
}
```

LCA

- 하지만 달라지는 시점에서 멈출 필요 없이 Sparse Table을 계속 이용할 수 있다
- i에서 달라졌다는 것은 2^i번째 조상은 다르지만 2^{i+1} 조상은 같다는 뜻이다
- 따라서 node = p[node][i]로 올라간다면 내 1번째 조상부터 2^i번째 조상 사이에 LCA가 존재하는 것을 알 수 있다
- 반복문을 멈추지 말고 계속 사용해서 LCA를 찾을 수 있다

LCA

```
if (a == b)
    return a;

for (int i = 17; i >= 0; i--) {
    if (p[a][i] != p[b][i]) {
        a = p[a][i];
        b = p[b][i];
    }
}

return p[a][0];
```

LCA

```
int lca(int a, int b) {
    if (dep[a] < dep[b])
        swap(a, b);
    int diff = dep[a] - dep[b];
    for (int i = 0; diff; i++) {
        if (diff & 1)
            a = p[a][i];
        diff >>= 1;
    }
    if (a == b)
        return a;
    for (int i = 17; i >= 0; i--) {
        if (p[a][i] != p[b][i]) {
            a = p[a][i];
            b = p[b][i];
        }
    }
    return p[a][0];
}
```

문제

- LCA 2 BOJ 11438
- 정점들의 거리 BOJ 1761
- 도로 네트워크 BOJ 3176

Priority Queue
Heap

Priority Queue

- 스택과 큐는 들어오는 순서에 따라서 나가는 순서가 정해졌다면, 우선순위 큐는 데이터 값에 따라서 나가는 순서가 정해진다
- 제일 기본적인 우선순위 큐는 값이 큰 데이터부터 꺼낸다
- 일반적으로 힙을 통하여 구현되어 있다

Priority Queue

1	2	3	4	5	6

1	2	9	14	3	12

Heap

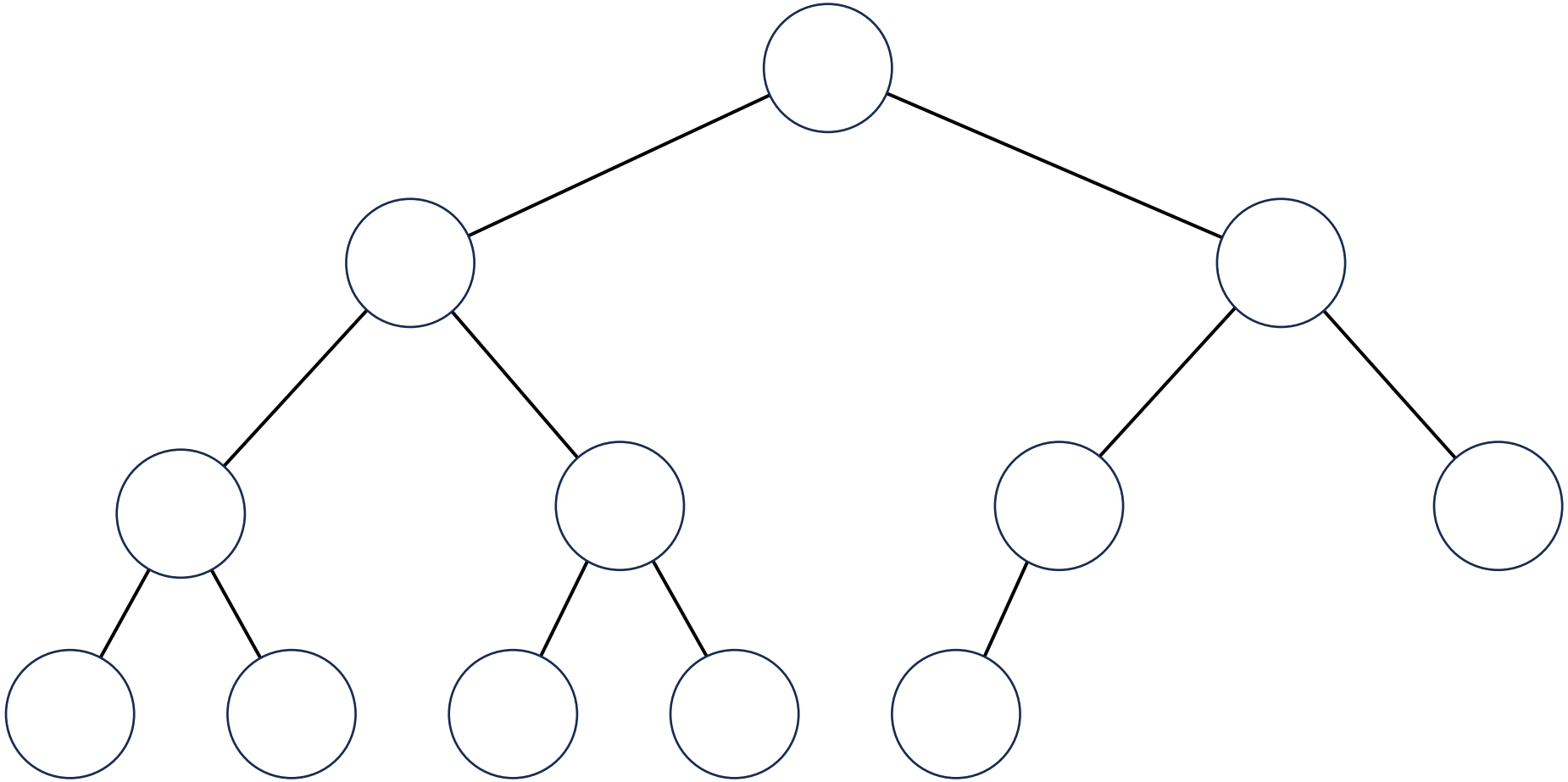
- 이진 트리 구조의 자료구조
- 제일 아래에 있는 레벨을 제외한 나머지 노드는 모두 채워져 있다
- 마지막 레벨은 왼쪽에서부터 채워져 있다

Heap

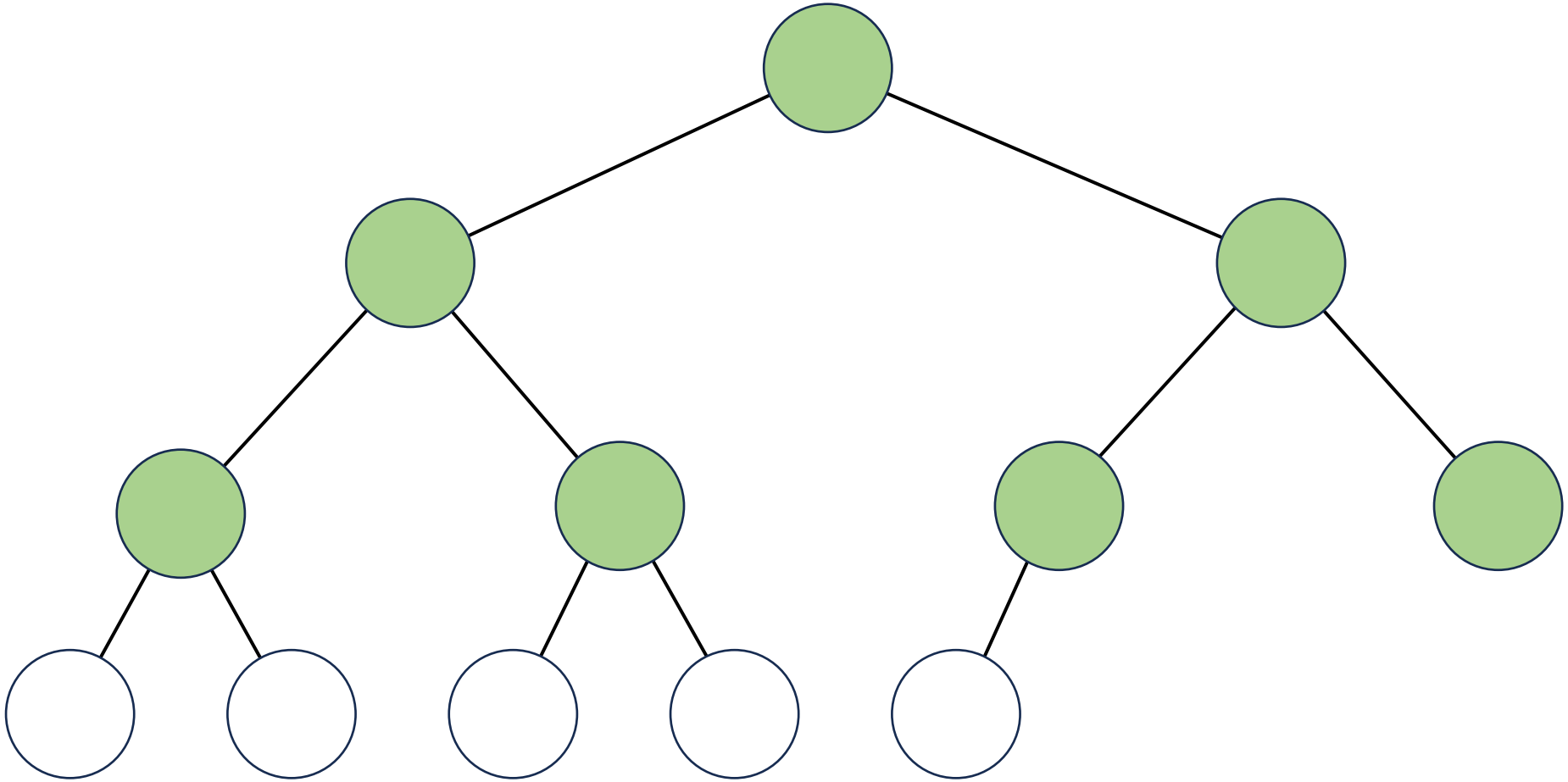
- 전체 정렬이 아닌 ~ 중에 최대값 또는 최소값을 빠르게 찾도록 만들어진 자료구조이다
- 모든 값이 정렬된 형태가 아닌 부모와 자식 간에 정렬된 형태를 보인다
- 최대값을 찾고 싶은 경우 부모가 자식들보다 큰 값을 유지하는 형태로 만든다(최대 힙)

- 어떠한 대소 관계를 정했을 때(오름차순, 내림차순, ...) 부모 노드와 자식 노드들의 관계를 모두 유지시킨 구조를 가진다

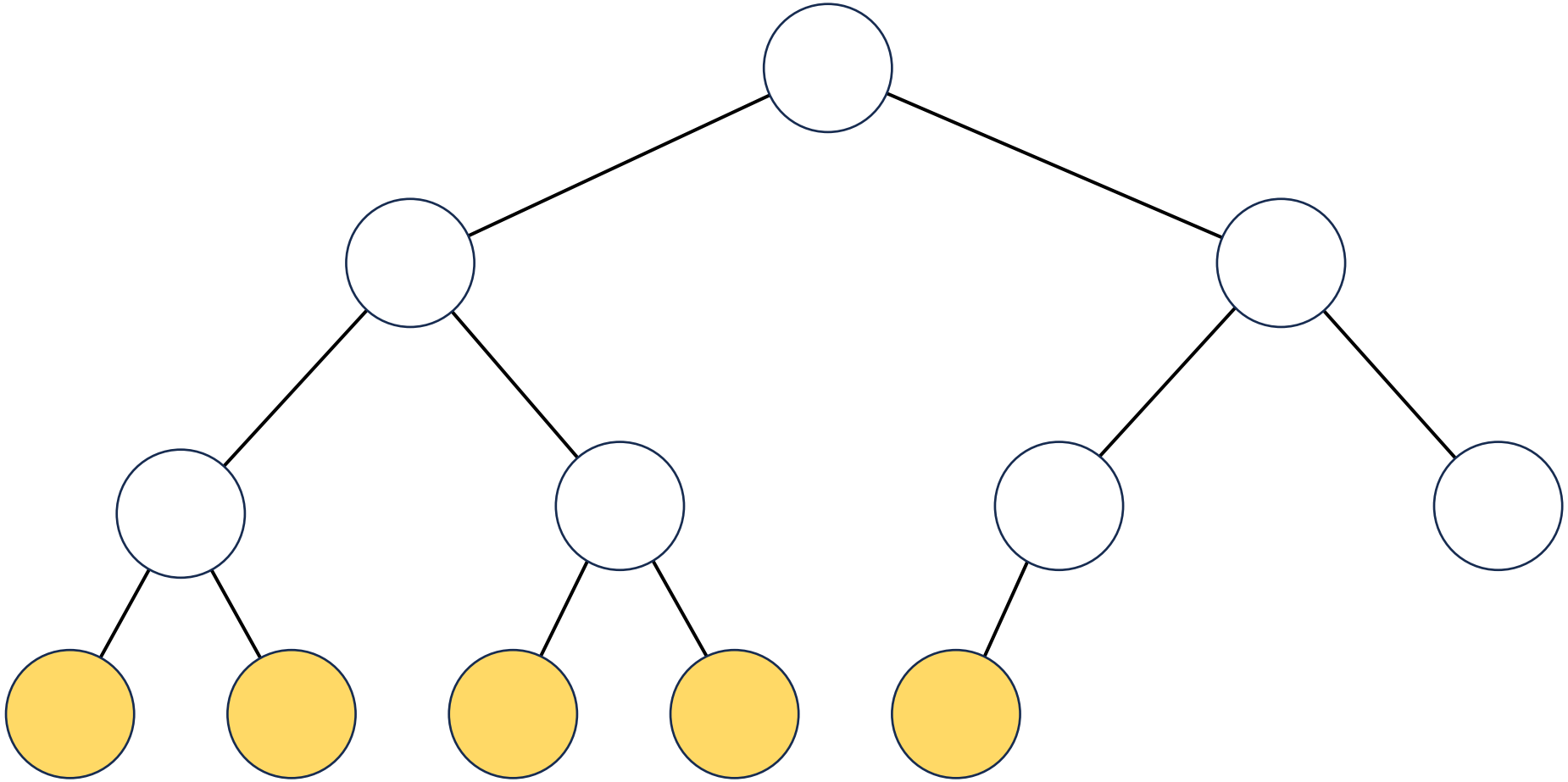
Heap



Heap



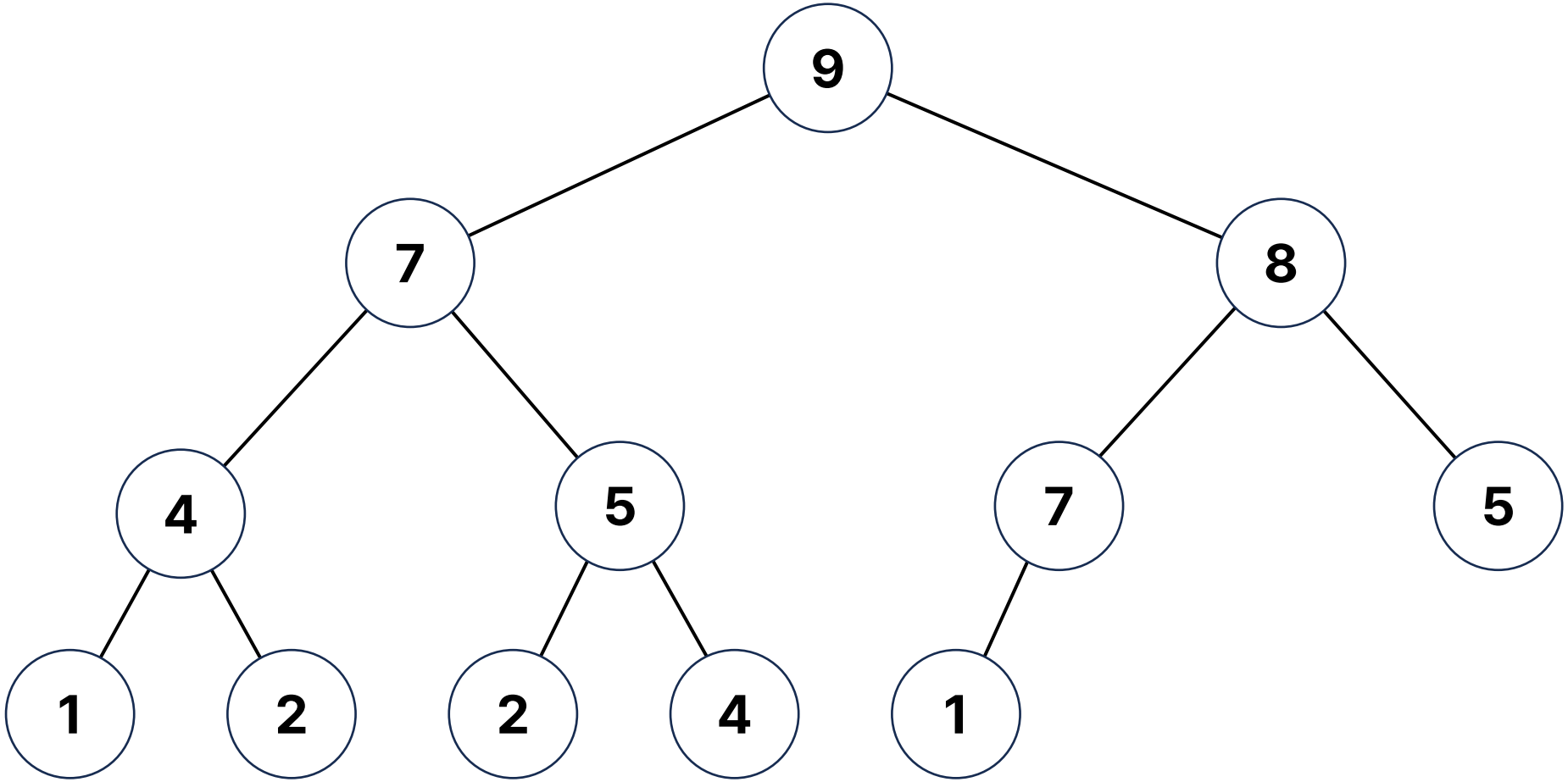
Heap



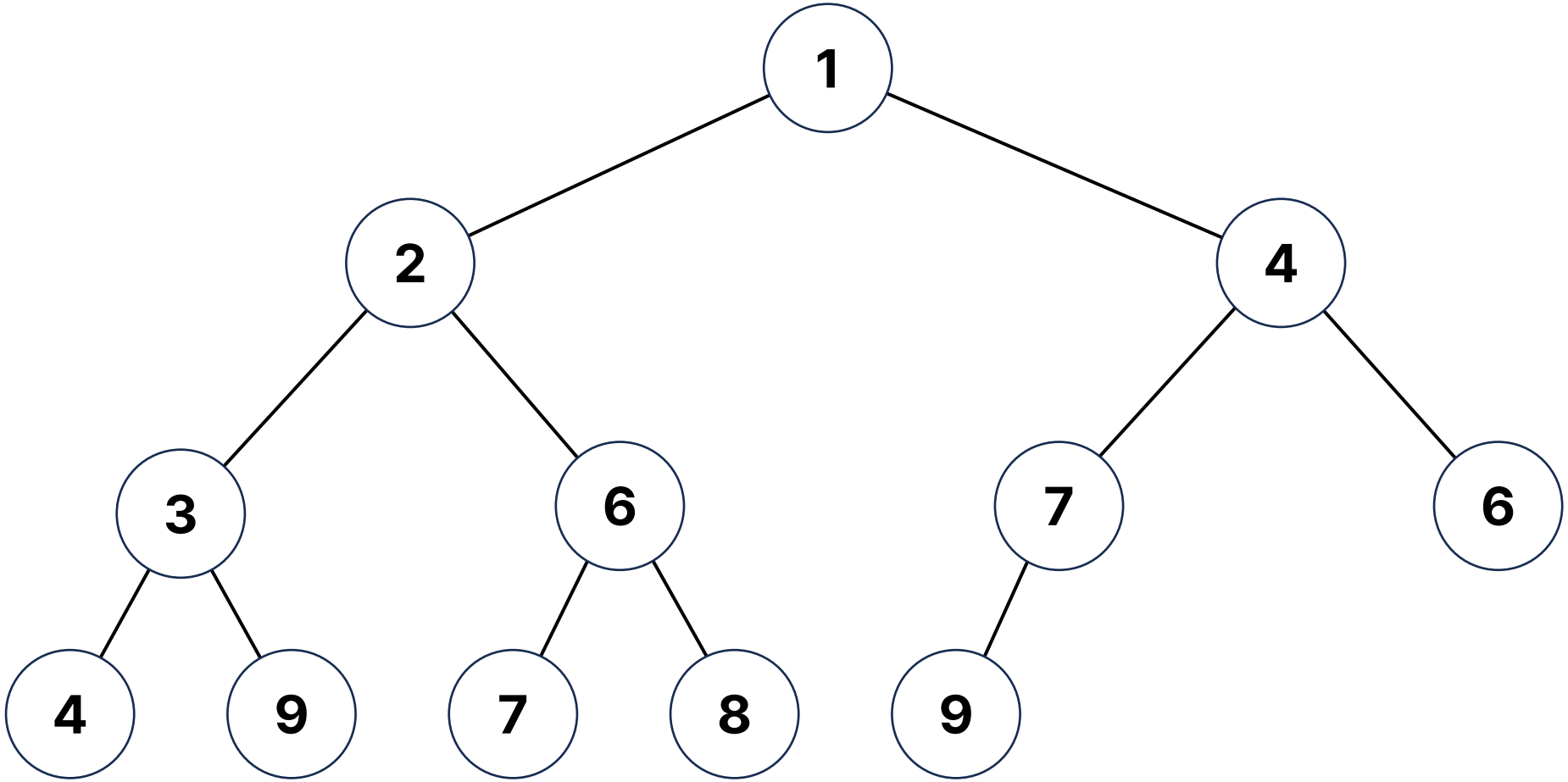
Heap

- 최대 힙: 부모 노드가 자식 노드들보다 큰 값을 유지하는 힙
- 최소 힙: 부모 노드가 자식 노드들보다 작은 값을 유지하는 힙

Max Heap



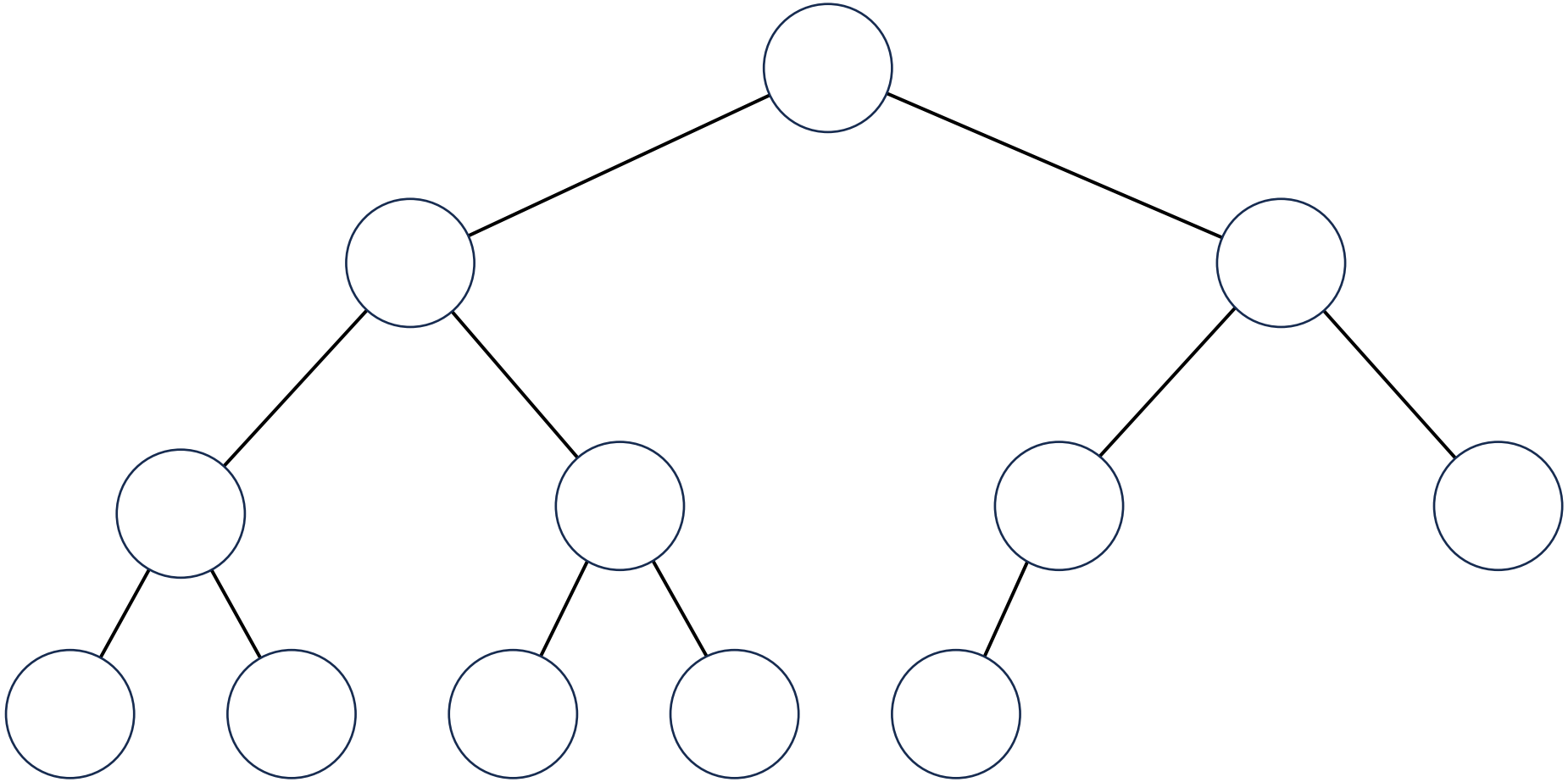
Max Heap



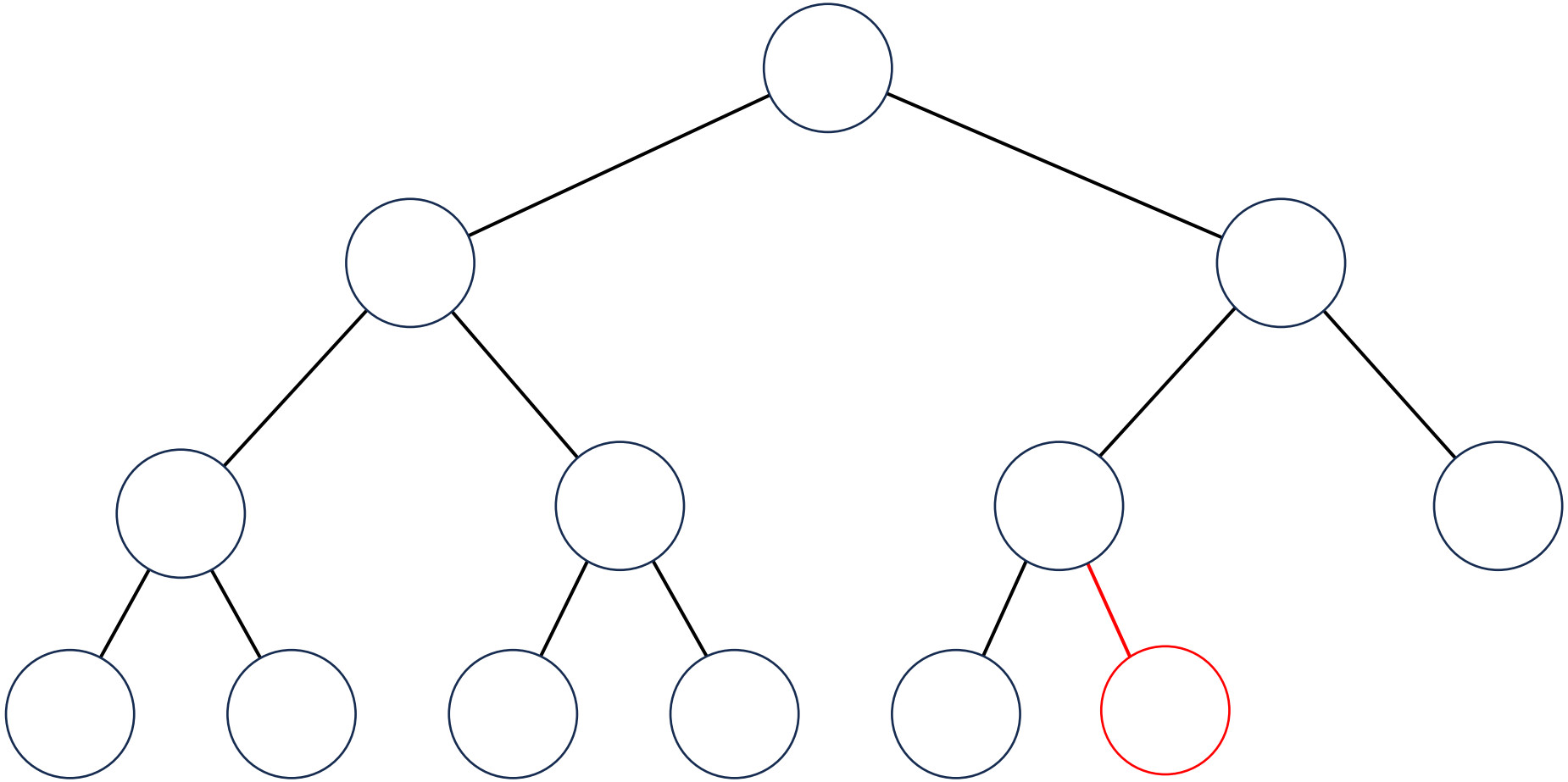
Heap Insert

- 힙 구조에 맞춰서 삽입은 마지막 레벨에 왼쪽부터 노드를 채운다
- 노드를 삽입한 이후에는 Up-heap Bubbling 과정을 통하여 힙의 구조(부모와 자식 관계)를 유지한다

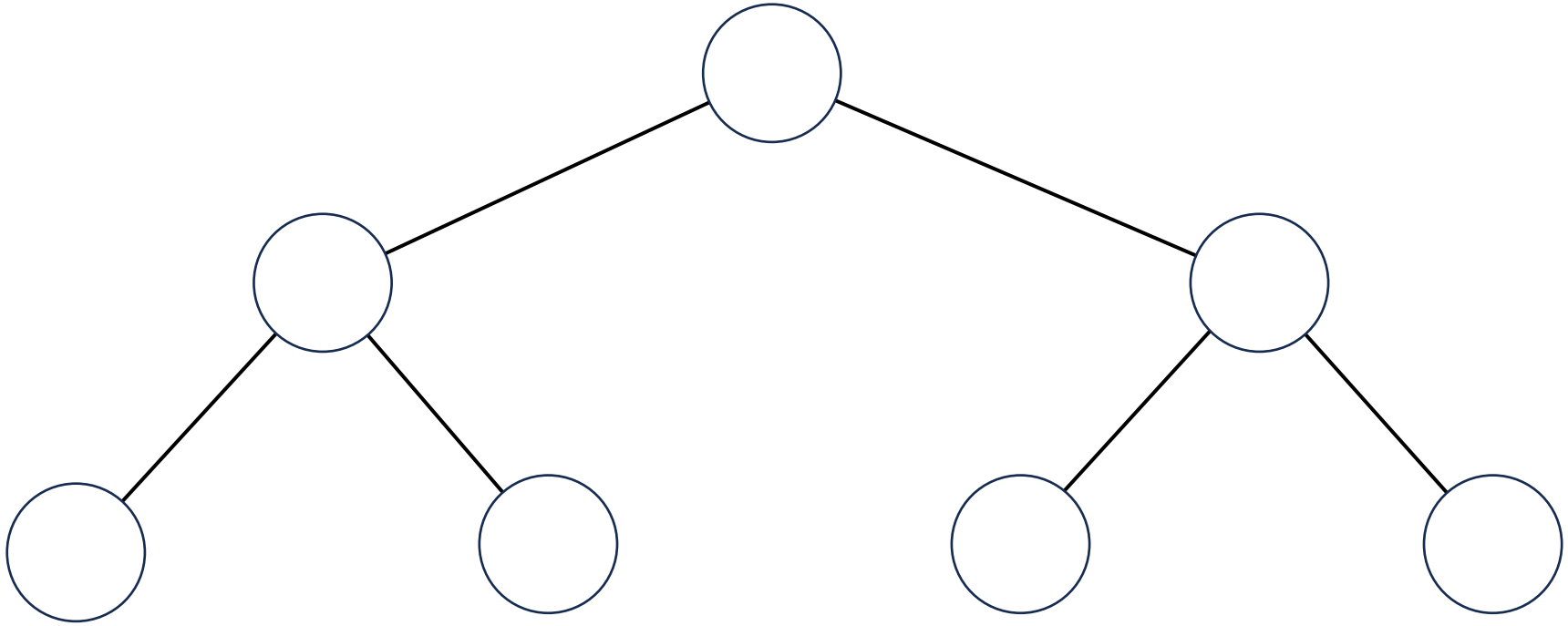
Heap Insert



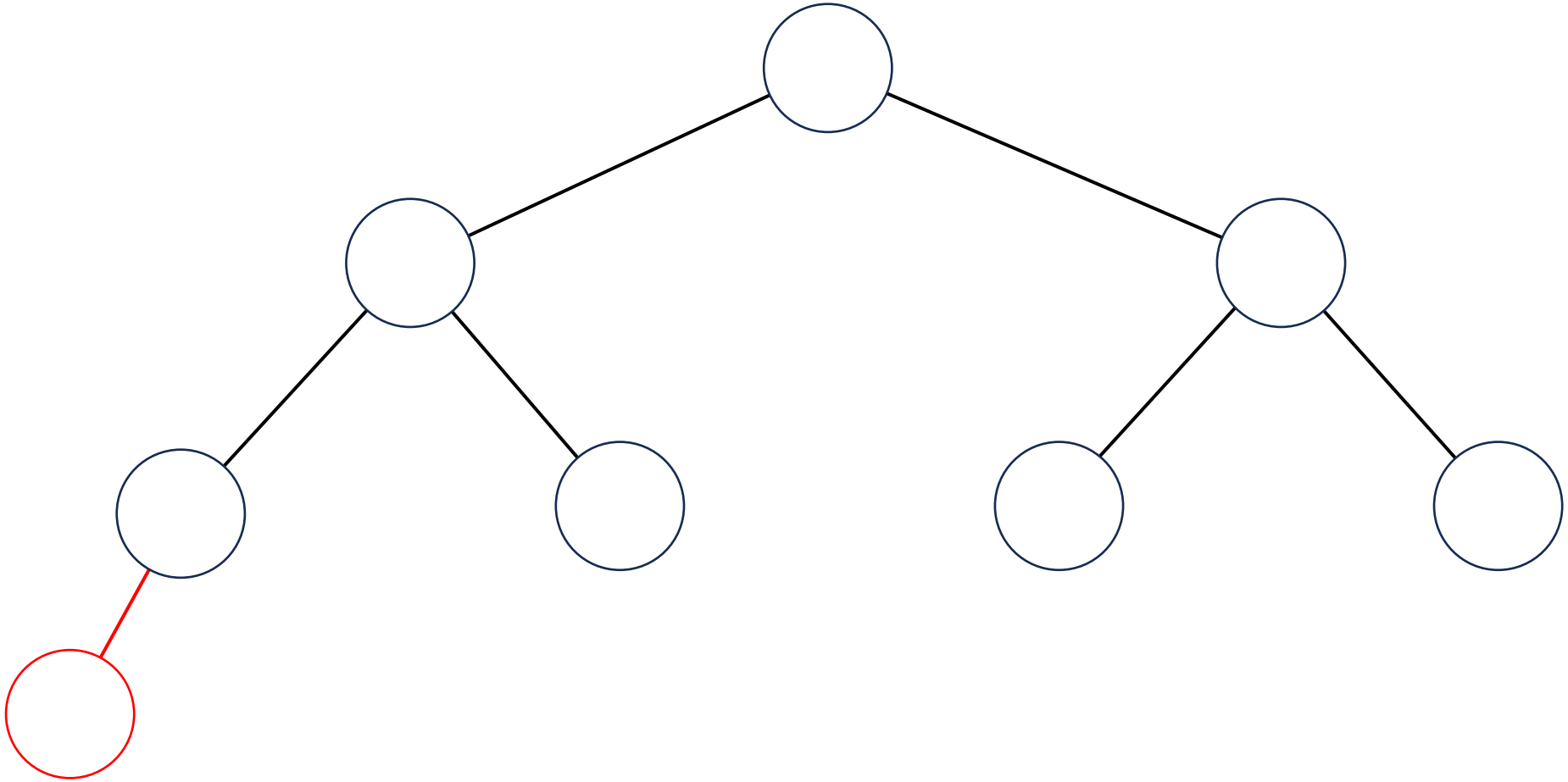
Heap Insert



Heap Insert



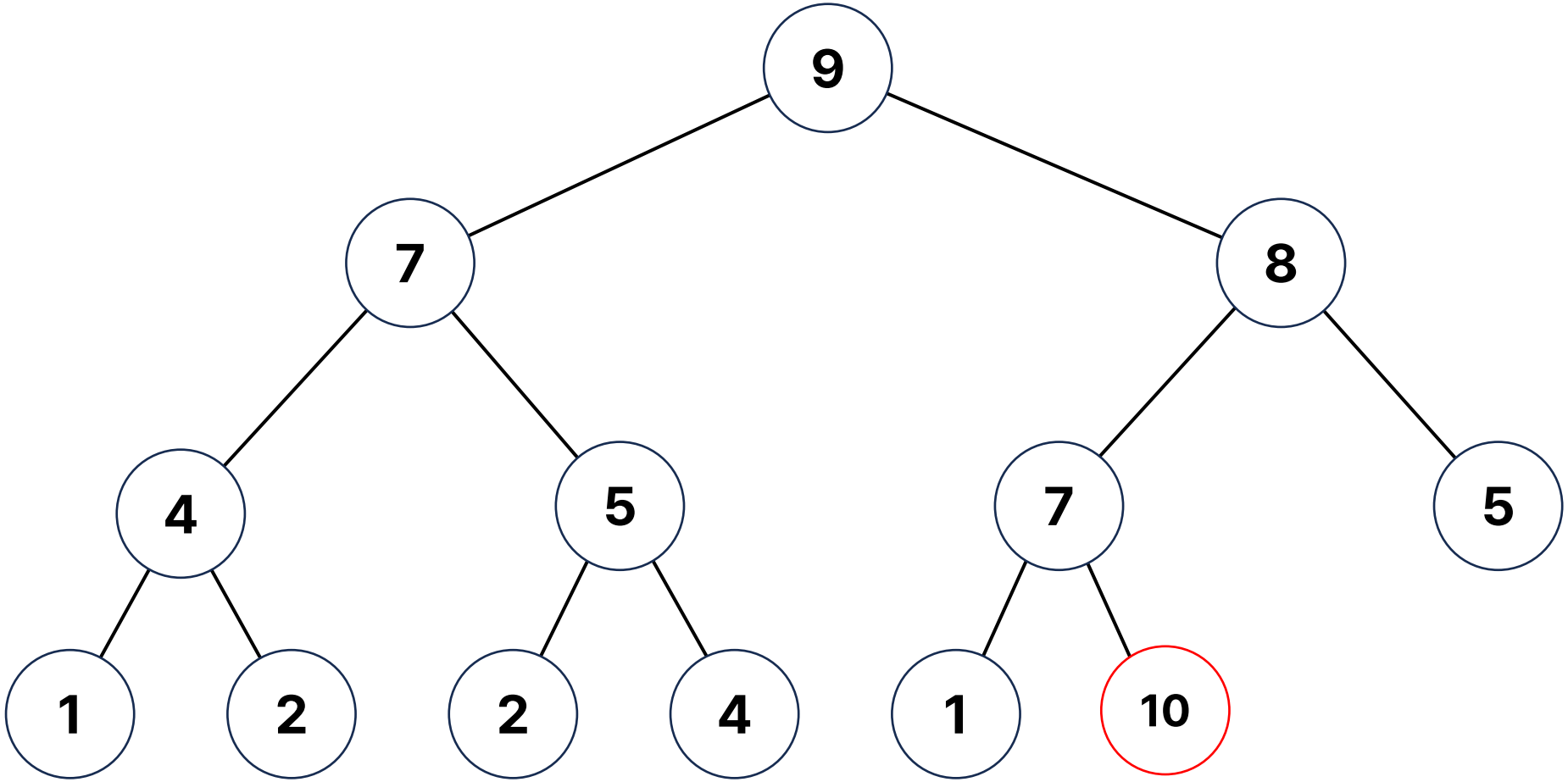
Heap Insert



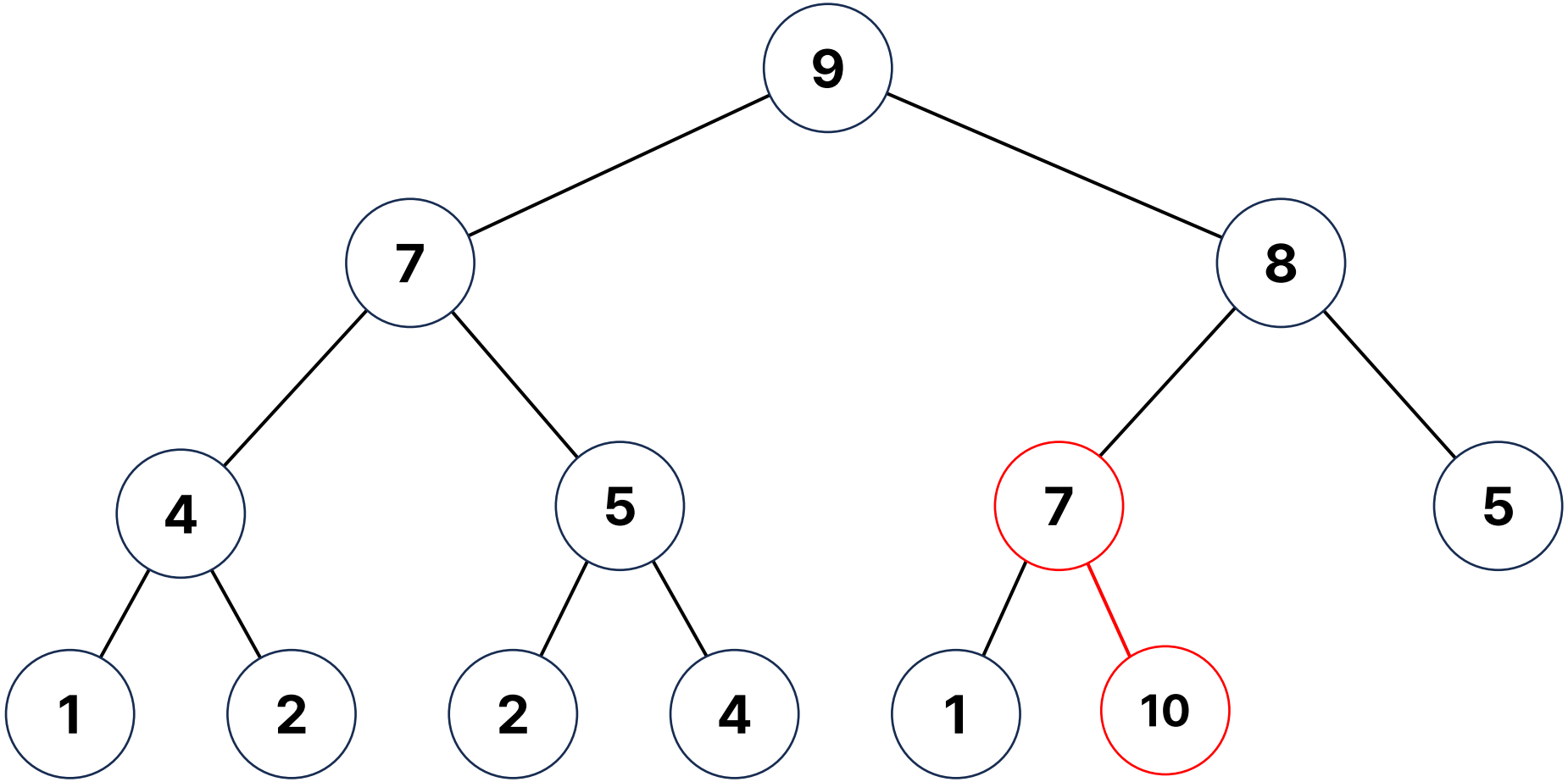
Up-heap Bubbling

- 힙 형태에 맞춰서 노드를 삽입한 이후에 부모-자식간 대소 관계를 유지하기 위한 과정
- 새로 삽입한 노드를 부모 노드와 값을 비교해 대소 관계가 올바른지 확인한다
- 반대로 되어 있다면 부모 노드와 값을 교환한다
- 루트 노드까지 또는 더 이상 교환이 일어나지 않을 때까지 반복한다

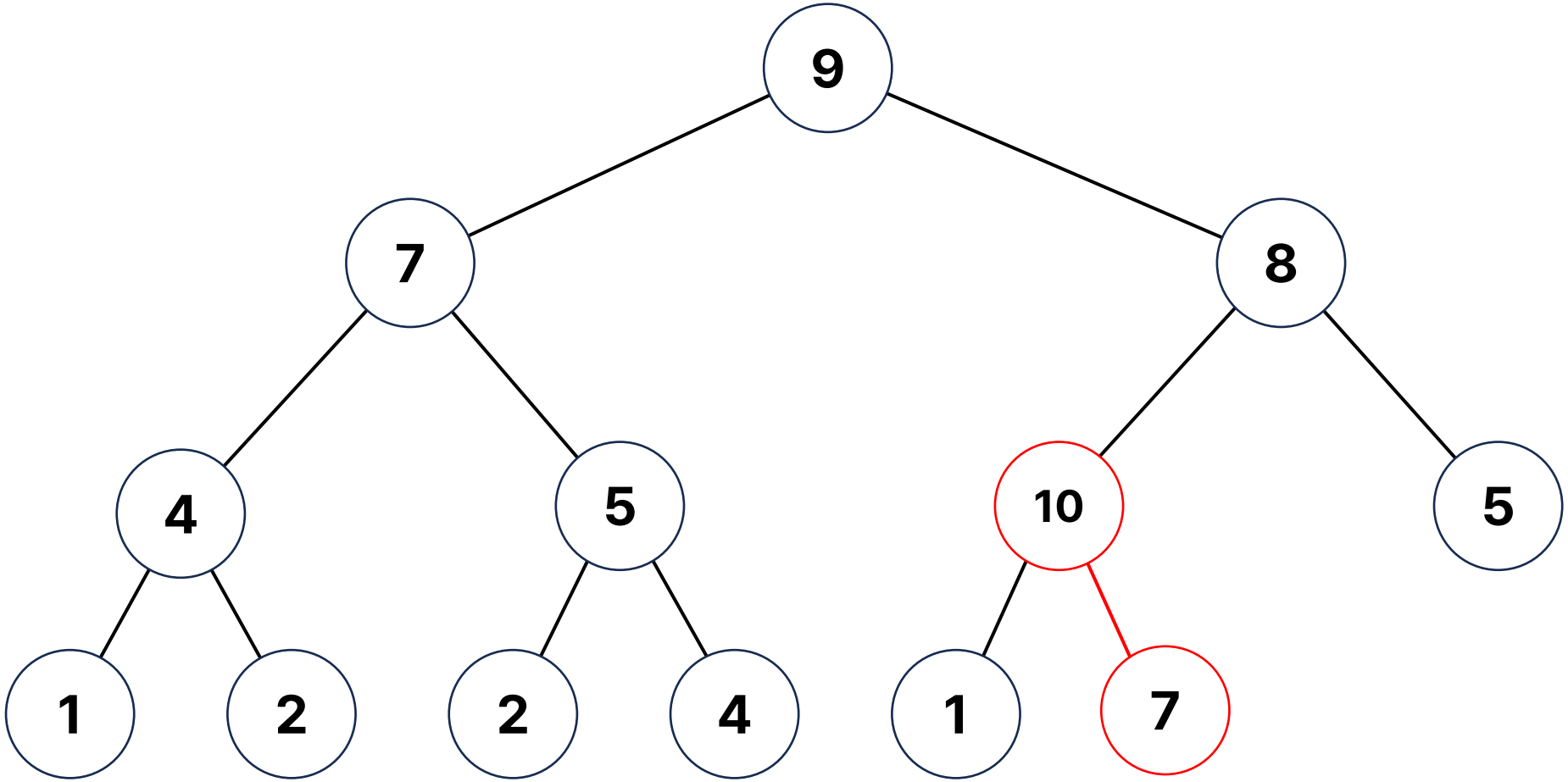
Max Heap Insert



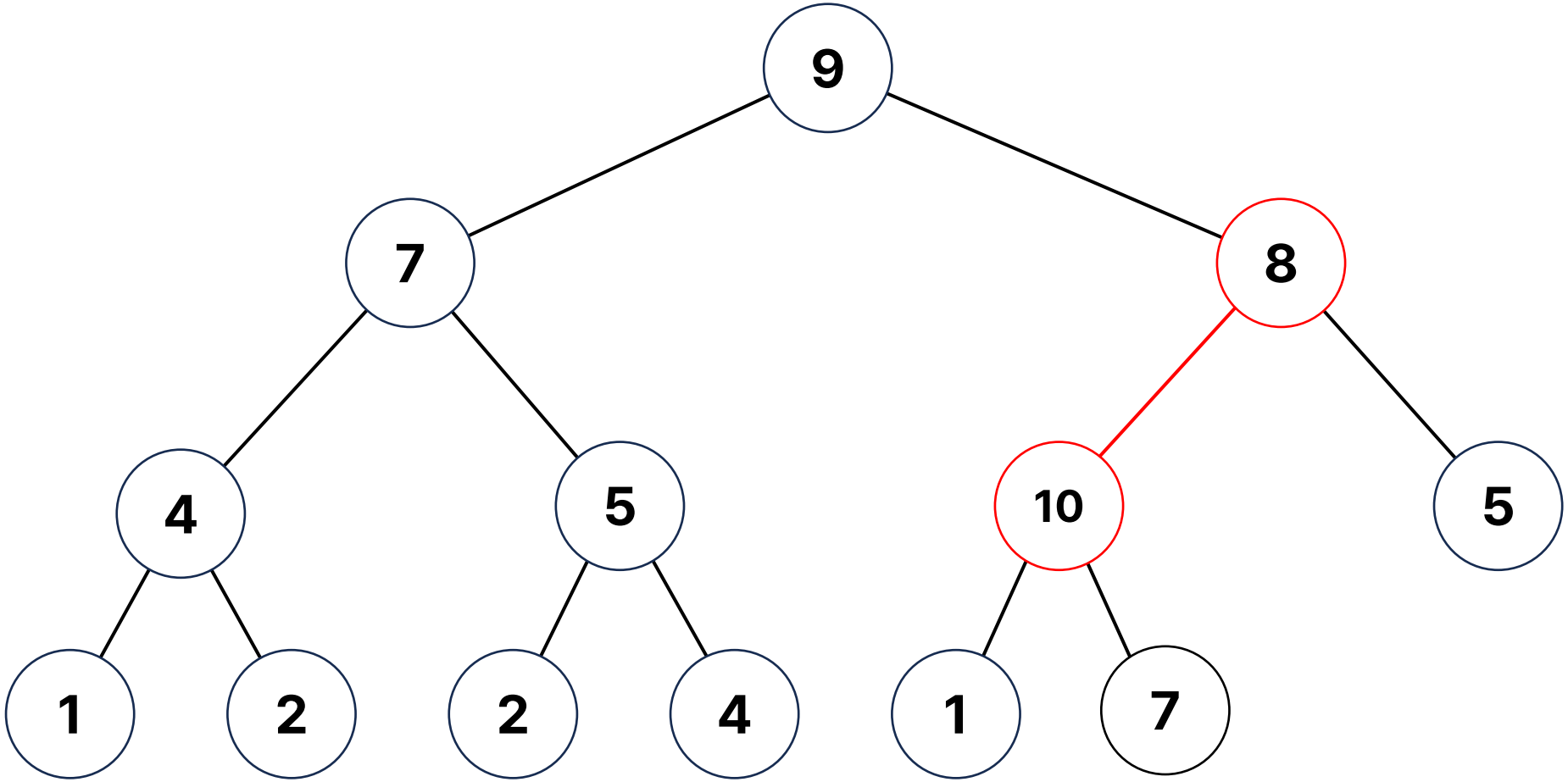
Max Heap Insert



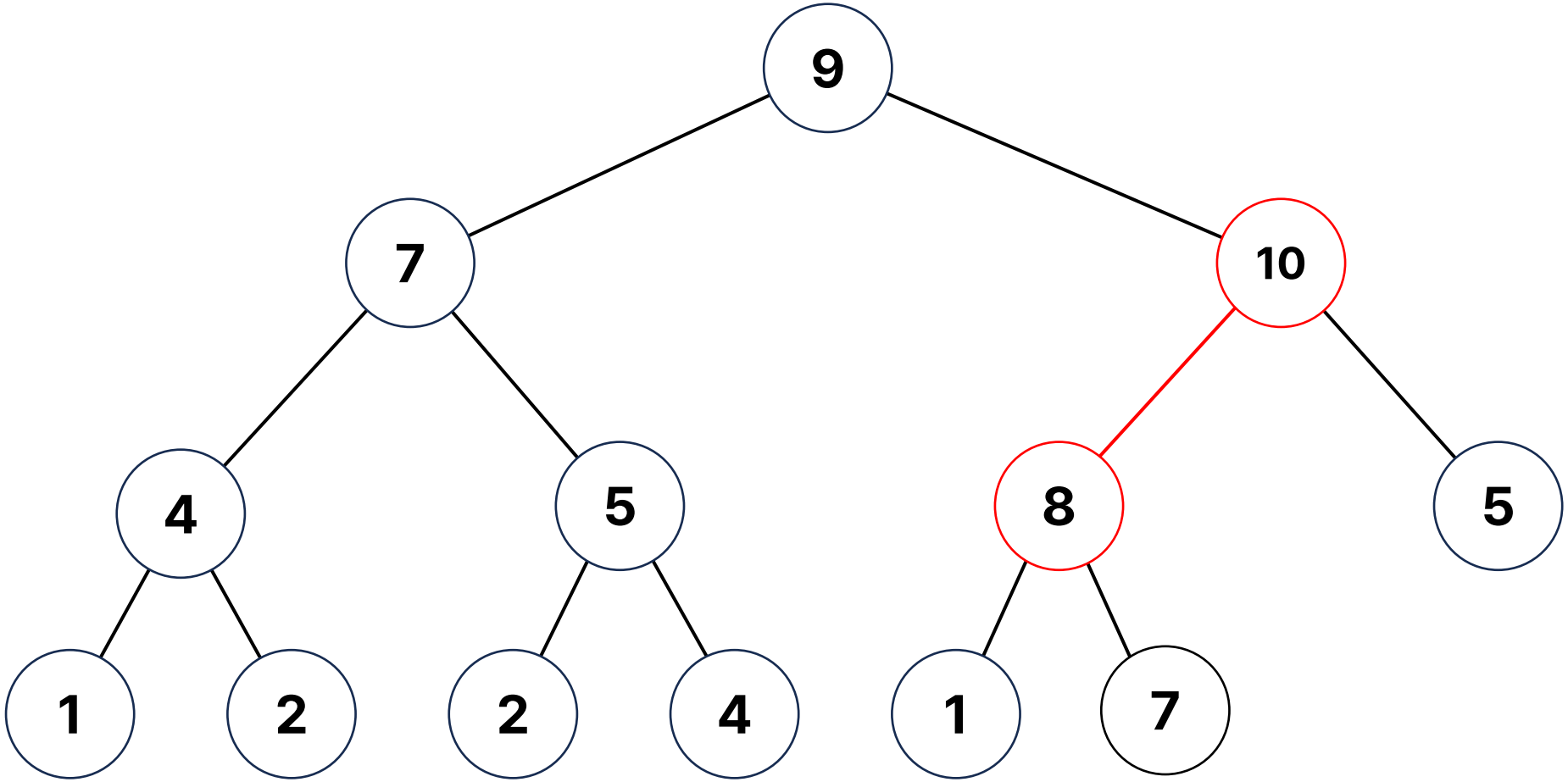
Max Heap Insert



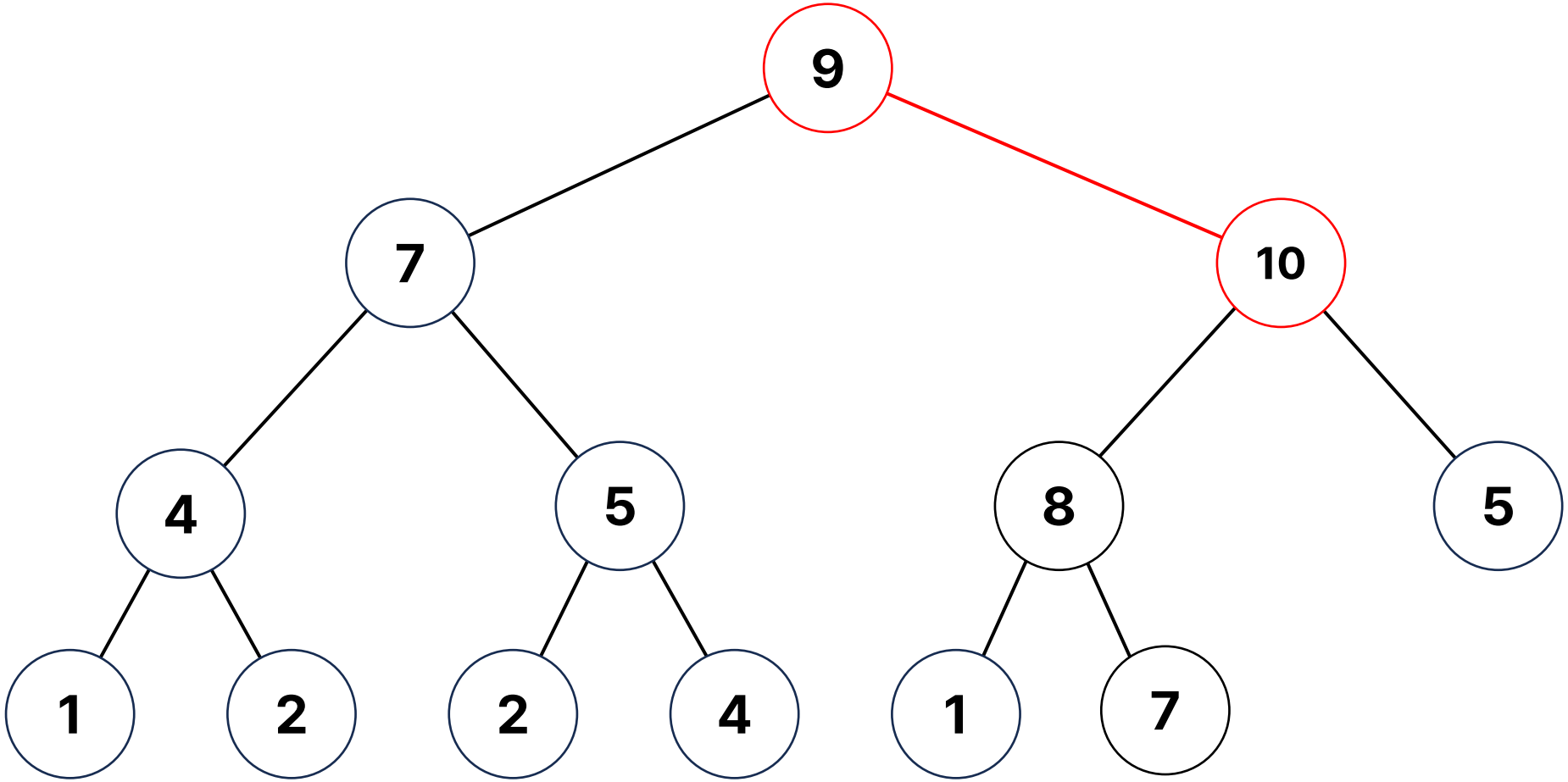
Max Heap Insert



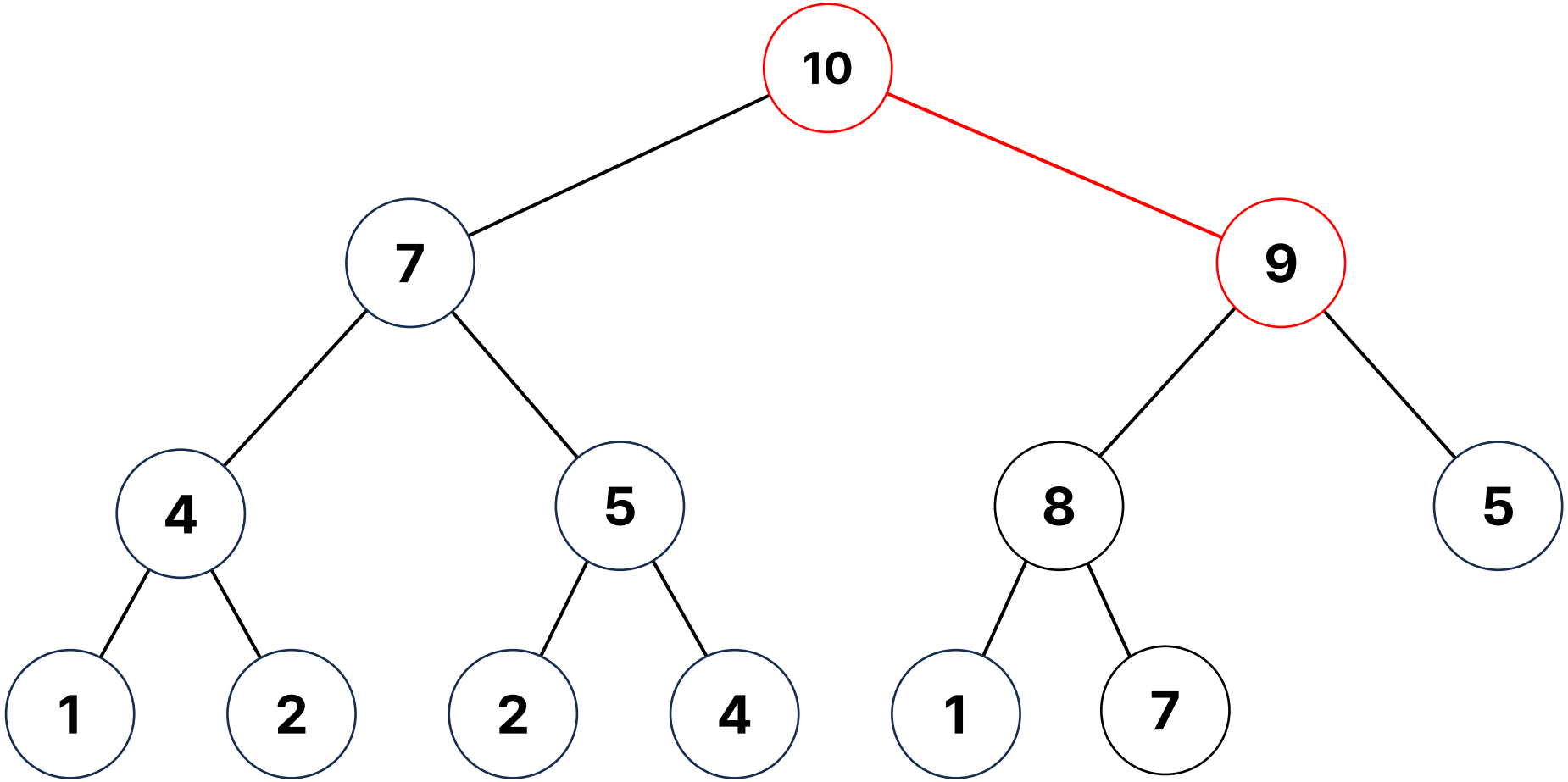
Max Heap Insert



Max Heap Insert



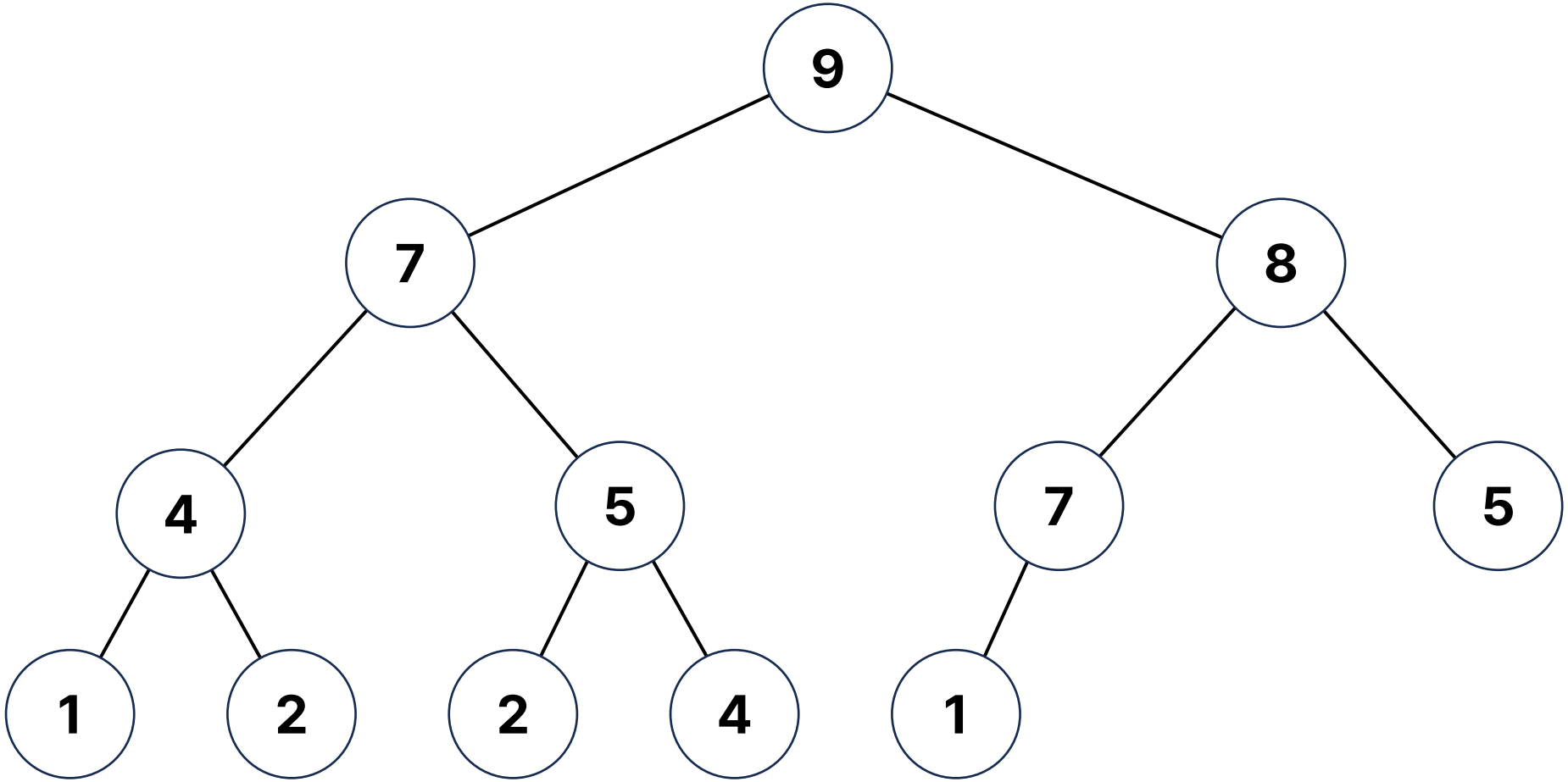
Max Heap Insert



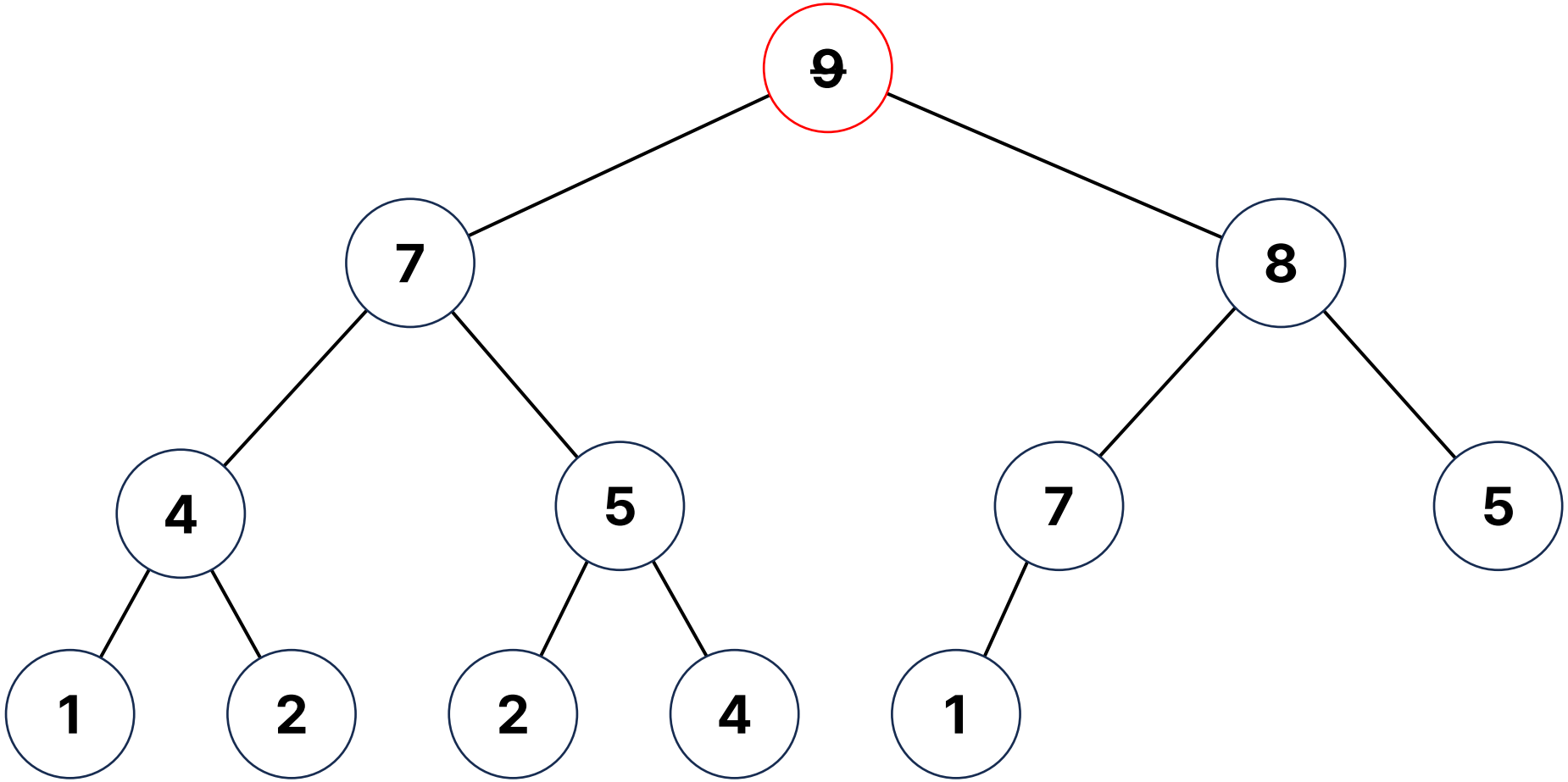
Heap Delete

- 루트에 있는 노드를 제거한다
- 제일 마지막 노드(제일 아래 레벨 중 제일 오른쪽 노드)를 루트로 가져온다
- 노드를 삽입한 이후에는 Down-heap Bubbling 과정을 통하여 힙의 구조(부모와 자식 관계)를 유지한다

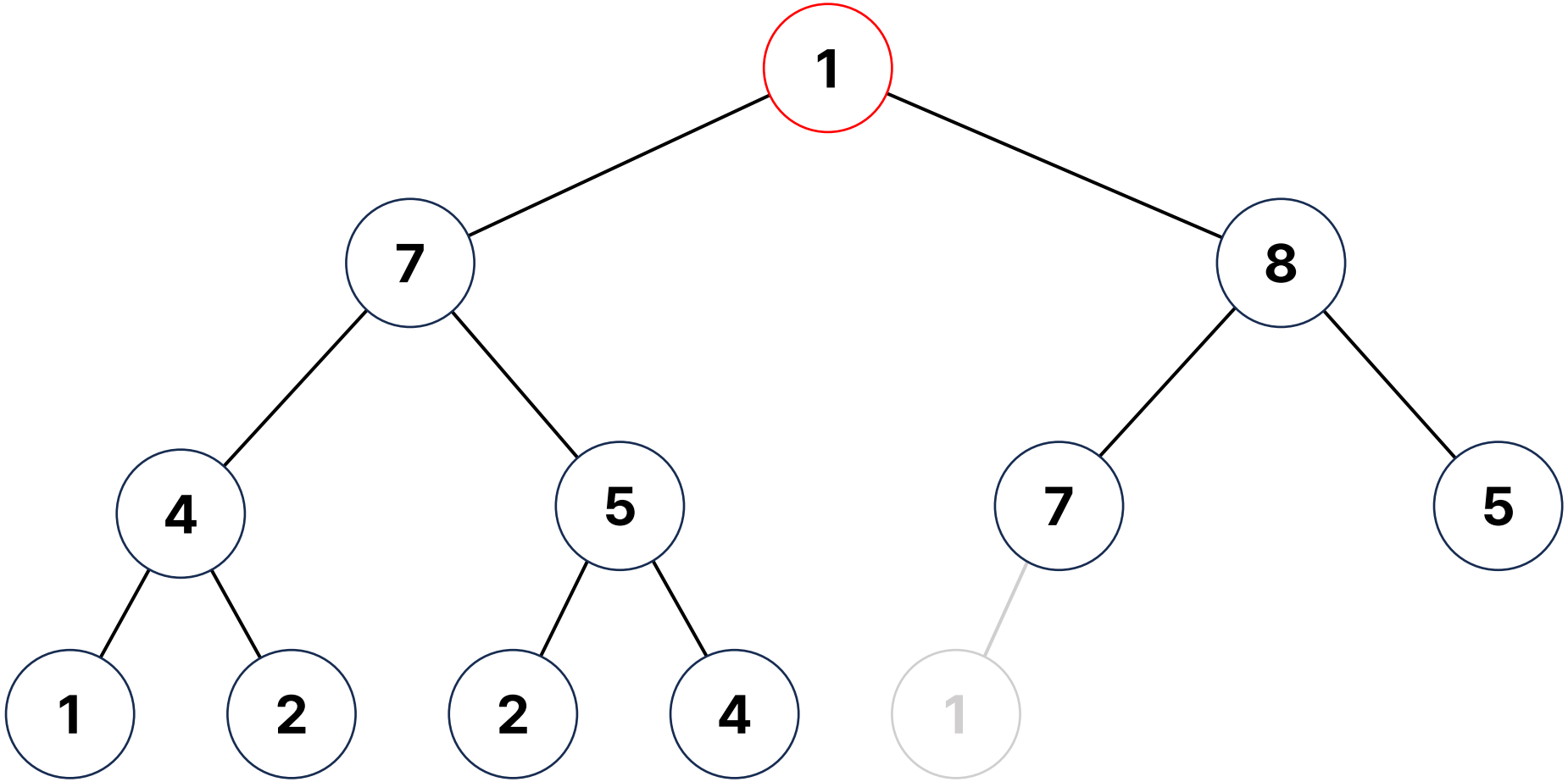
Max Heap Delete



Max Heap Delete



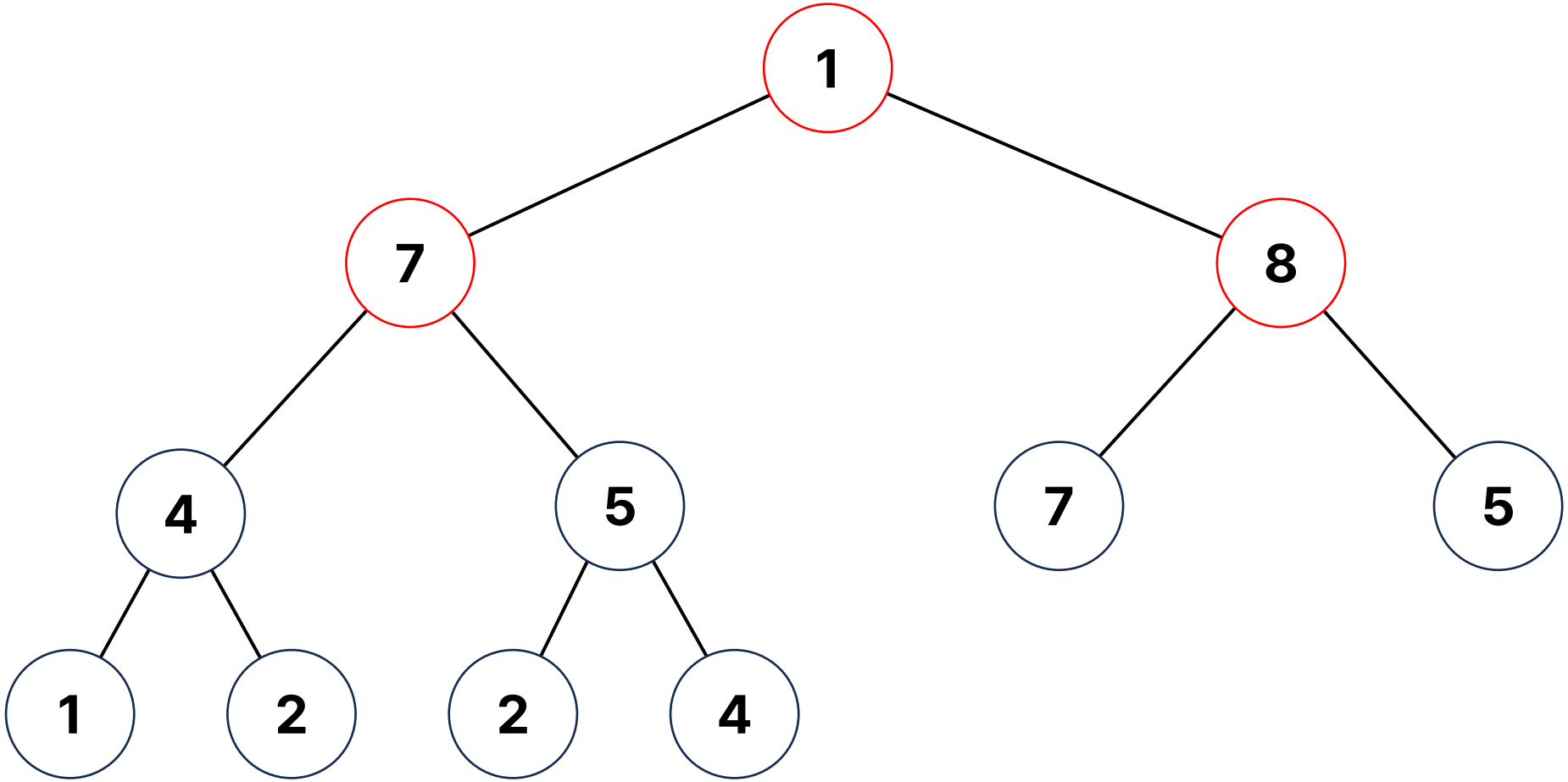
Max Heap Delete



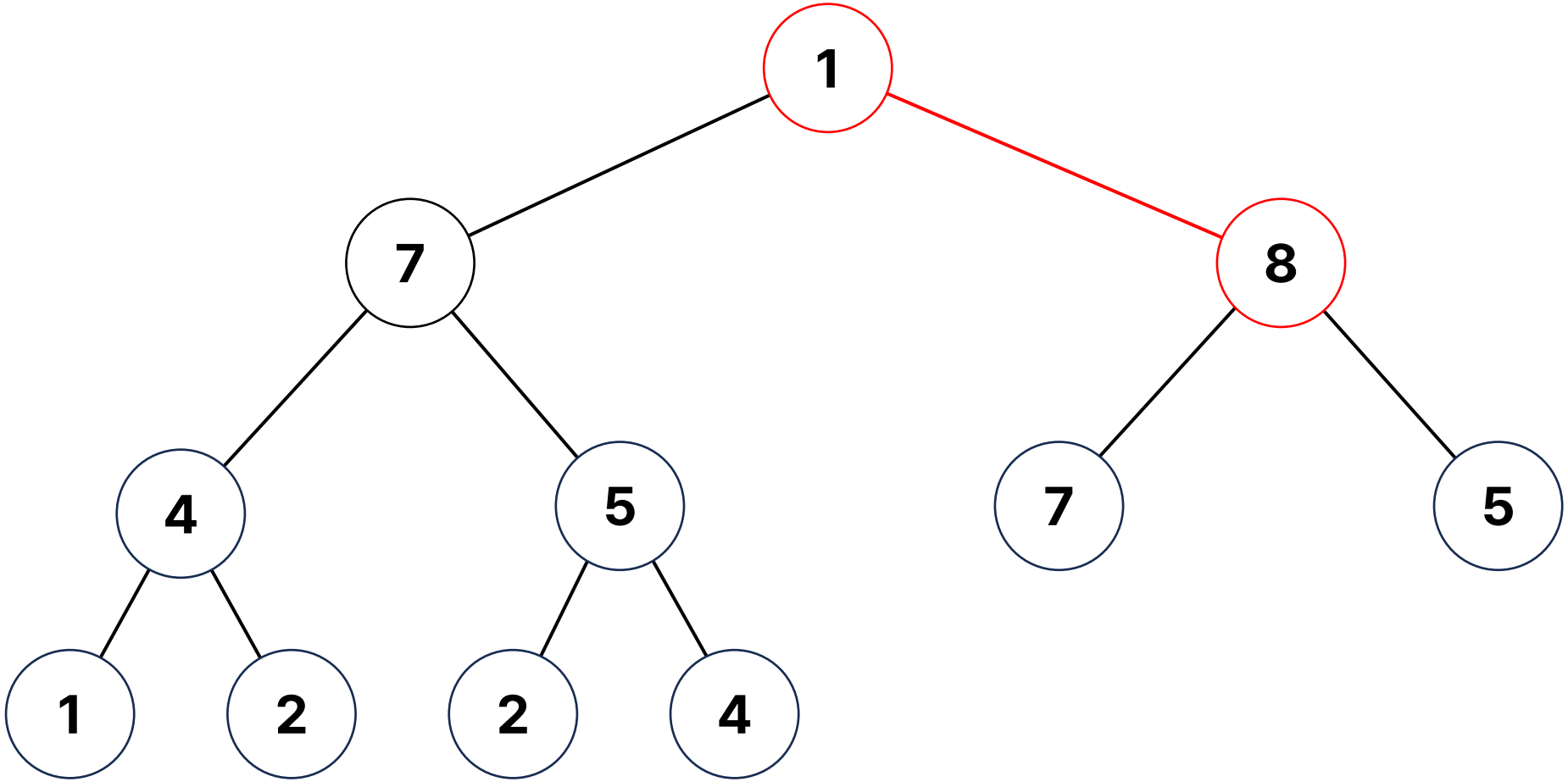
Down-heap Bubbling

- Up-heap Bubbling과 마찬가지로 힙 형태에 맞춰서 노드를 삽입한 이후에 부모-자식 간 대소 관계를 유지하기 위한 과정
- Up-heap Bubbling과 반대로 루트에서부터 내려온다
- 루트 노드보다 큰 자식 노드가 존재한다면 자식 노드 중 큰 값과 교환한다
- 리프 노드까지 더 이상 교환이 일어나지 않을 때까지 반복한다

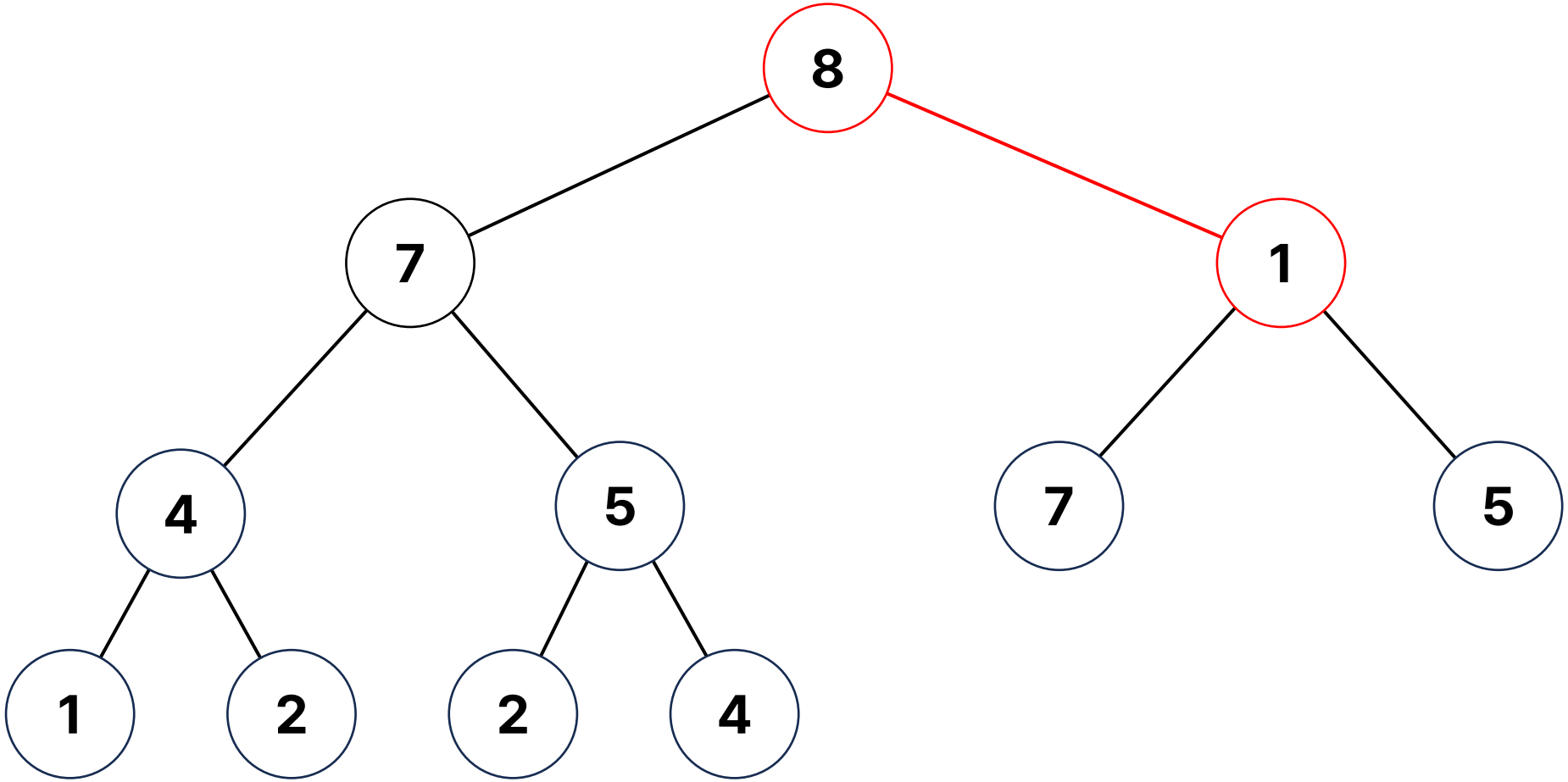
Max Heap Delete



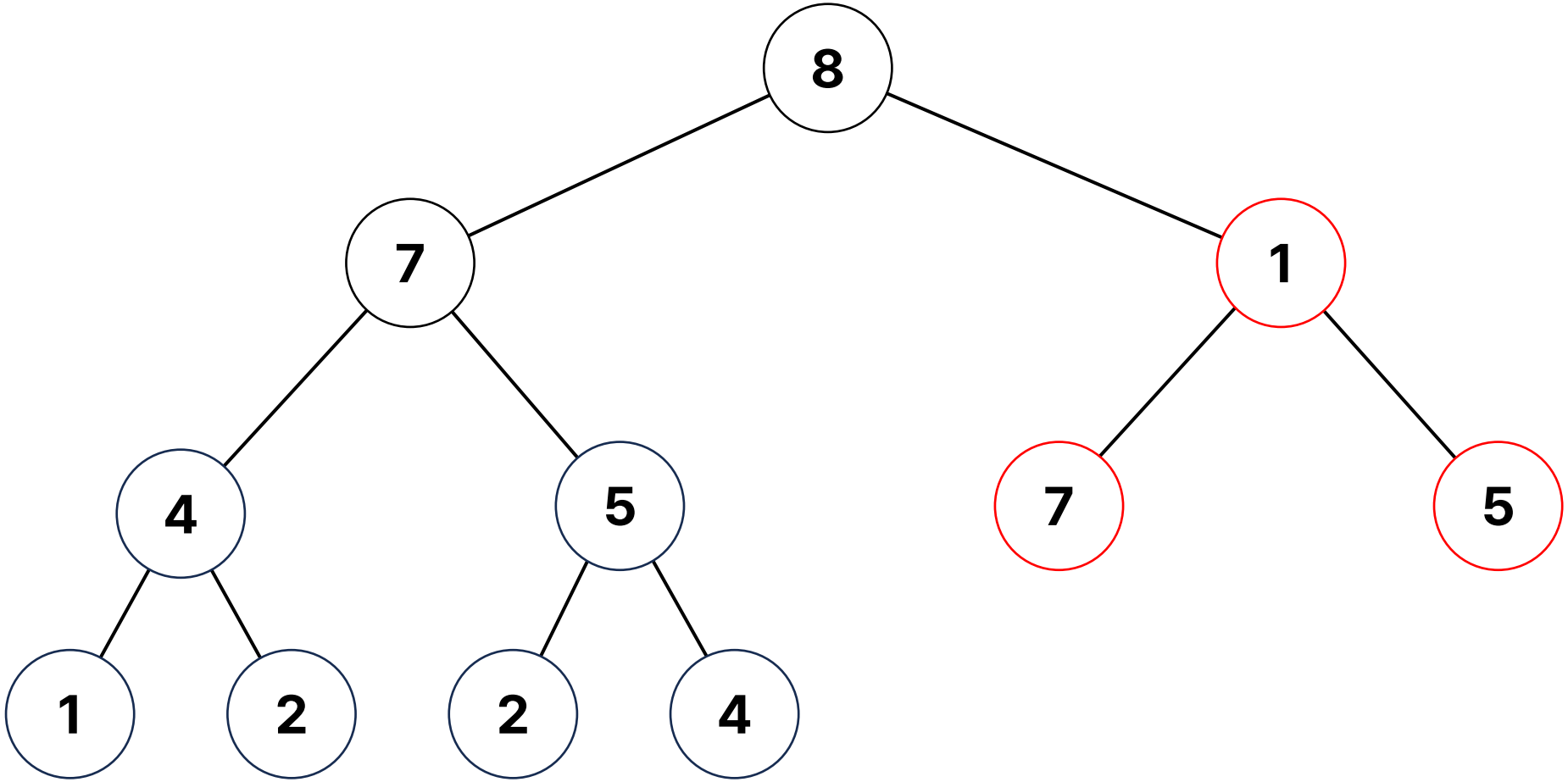
Max Heap Delete



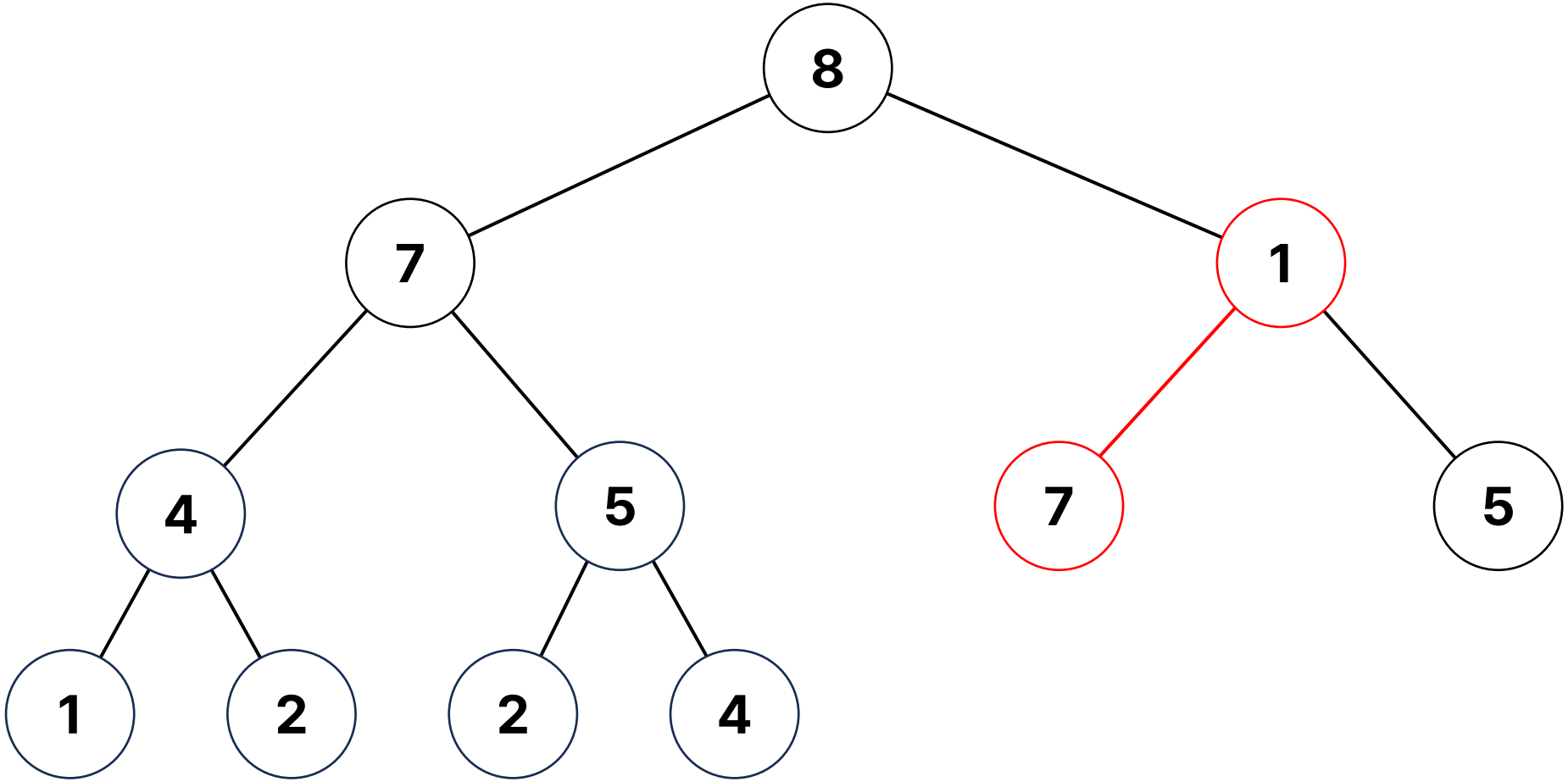
Max Heap Delete



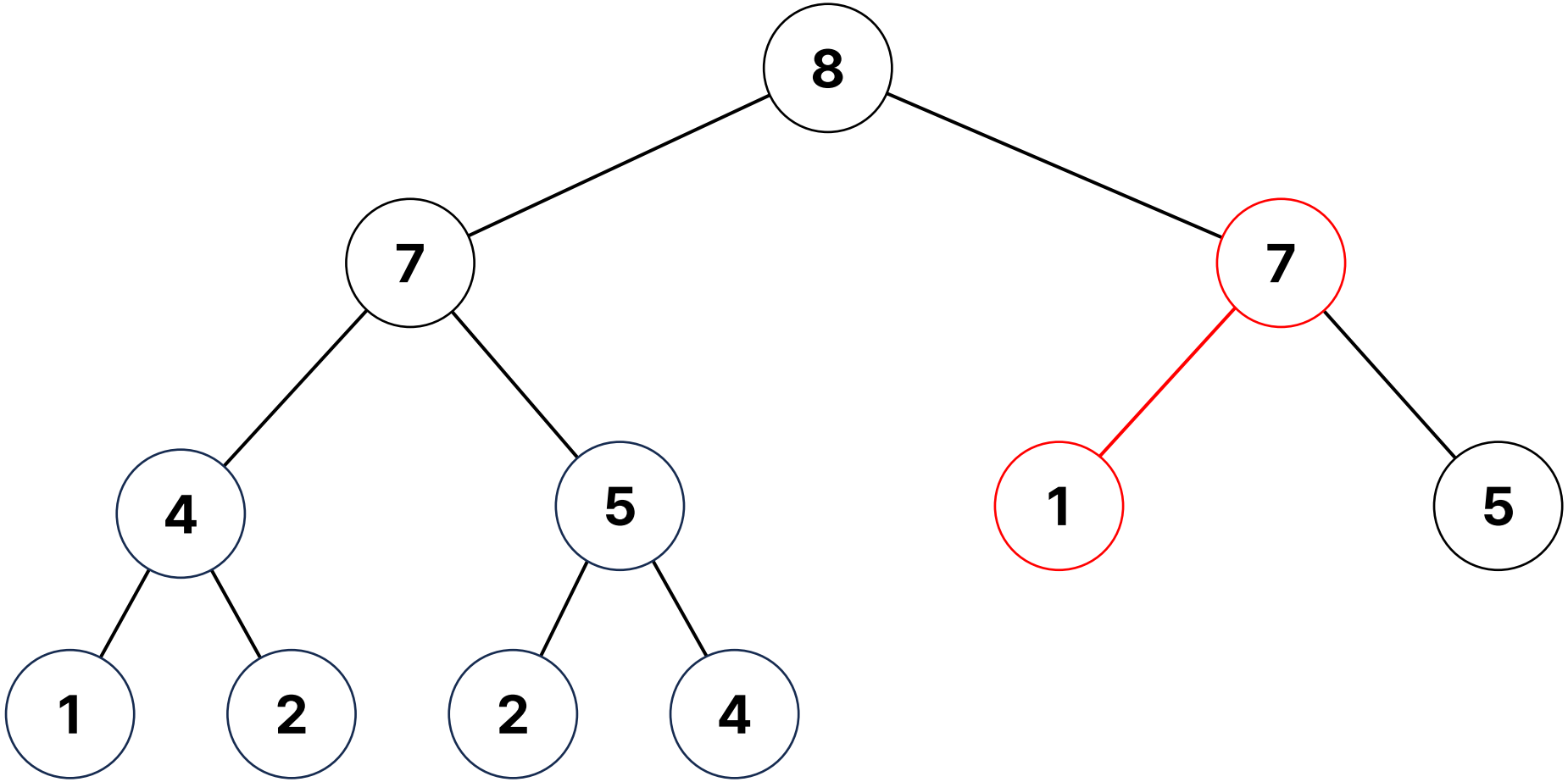
Max Heap Delete



Max Heap Delete



Max Heap Delete



Priority Queue

- 이러한 구조의 Heap을 이용해 삽입된 데이터 중 최대값/최소값을 꺼낼 수 있다

STL Priority Queue

- #include <queue>
- priority_queue<자료형> 변수명;

STL Priority Queue

- void pq.push(x): 우선순위 큐에 x를 삽입
- void pq.pop(): 우선순위 큐에서 첫번째 원소를 제거
- 자료형 pq.top(x): 우선순위 큐의 첫번째 원소를 리턴
- bool pq.empty(x): 우선순위 큐이 비어있으면 true, 아니면 false
- size_type pq.size(x): 우선순위 큐의 크기를 리턴

STL Priority Queue

- 기본적으로는 값이 크다는 것을 우선순위가 높다고 생각한다
- 정렬 순서를 내림차순으로 바꾸기 위해서는 부모와 자식 관계를 정할 함수를 전달하면 된다
- 두번째 원소의 경우 우선순위 큐에서 데이터를 저장하는데 사용할 자료형이다. 일반적으로 vector를 사용하면 된다
- priority_queue<int, vector<int>, greater<int>> pq;

STL Priority Queue

- 특정 구조체 또는 자료형을 사용하는 경우 대소 관계가 정해져 있지 않다
- 대소 관계를 정해줄 구조체를 만들거나 <연산자 오버로딩을 해주면 사용할 수 있다

STL Priority Queue

```
struct Node {
    int grade; // 등급, 낮을수록 우선순위가 높음
    int score; // 점수, 높을수록 우선순위가 높음
};

struct compare {
    bool operator()(Node a, Node b) {
        if (a.grade == b.grade)
            return a.grade > b.grade;
        return a.score > b.score;
    }
};

priority_queue<Node, vector<Node>, compare> pq;
```

STL Priority Queue

```
struct Node {
    int grade; // 등급, 낮을수록 우선순위가 높음
    int score; // 점수, 높을수록 우선순위가 높음

    bool operator<(const Node &t) const {
        if (grade == t.grade)
            return grade > t.grade;
        return score > t.score;
    }
};

priority_queue<Node> pq;
```

문제

- 최대 힙 BOJ 11279
- 최소 힙 BOJ 1927
- 절댓값 힙 BOJ 17286
- 카드 정렬하기 BOJ 1715
- 문제집 BOJ 1766