28th

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

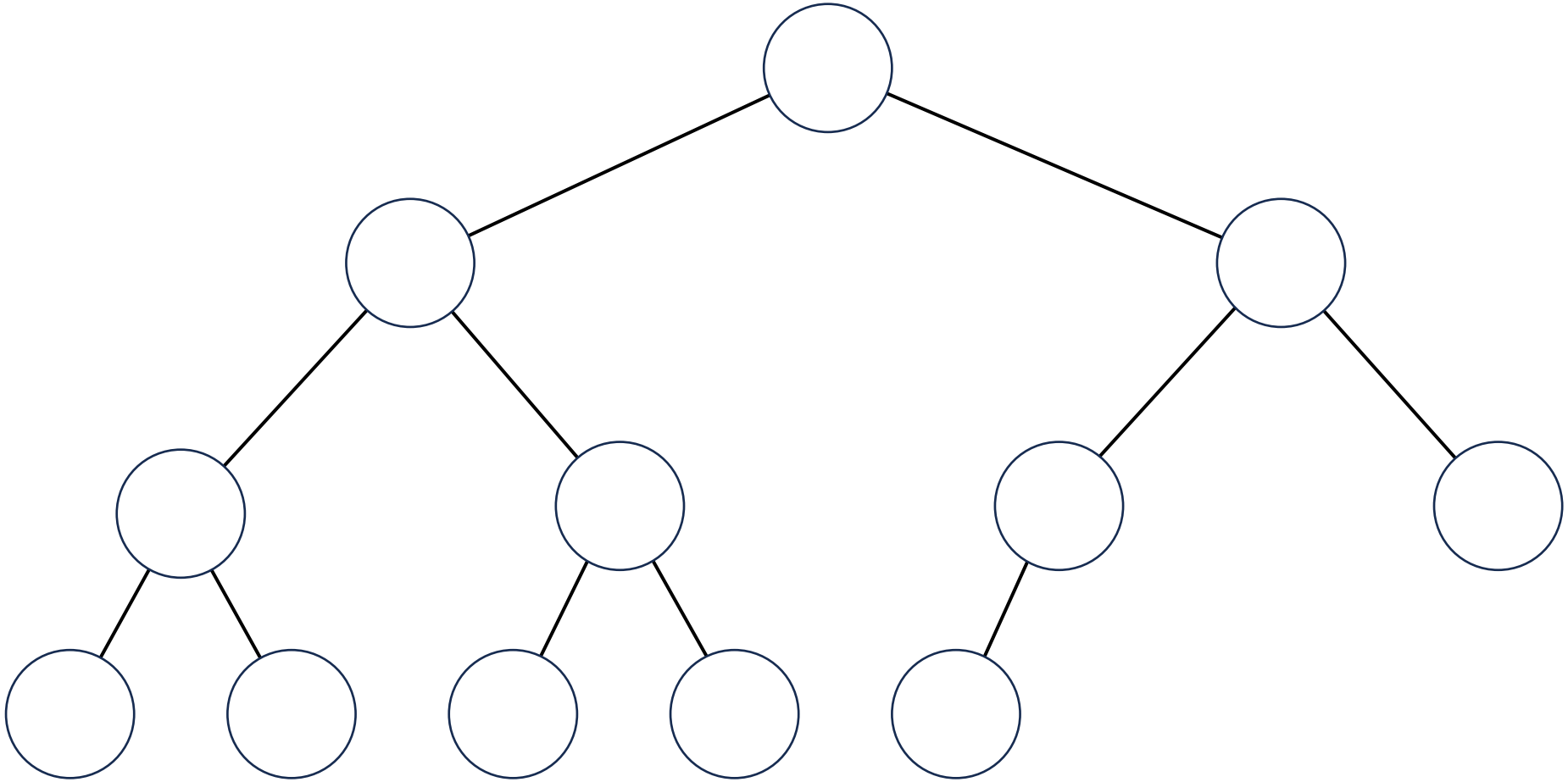
- 이진 트리 구조의 자료구조
- 제일 아래에 있는 레벨을 제외한 나머지 노드는 모두 채워져 있다
- 마지막 레벨은 왼쪽에서부터 채워져 있다

Heap

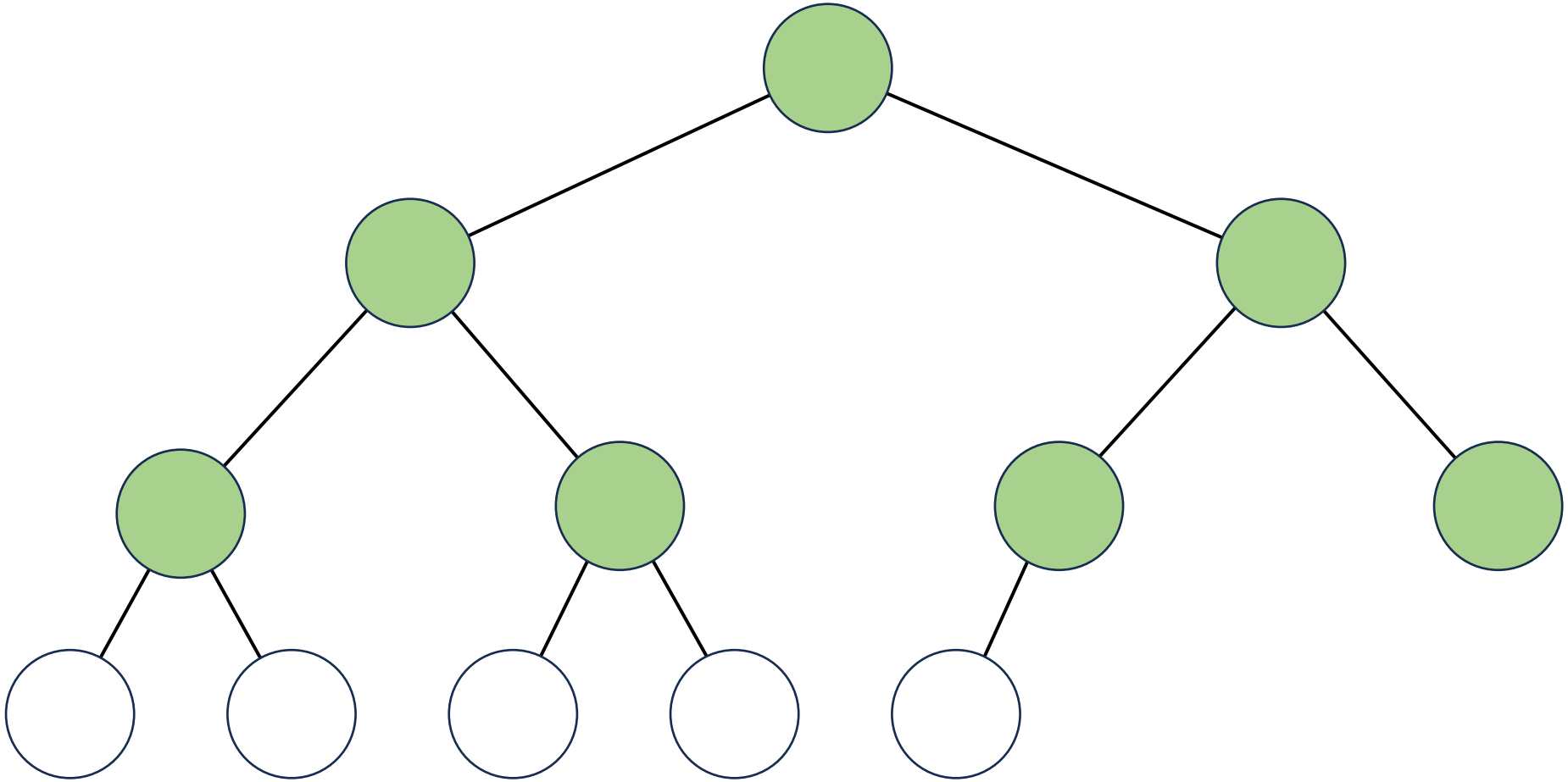
- 전체 정렬이 아닌 ~ 중에 최대값 또는 최소값을 빠르게 찾도록 만들어진 자료구조이다
- 모든 값이 정렬된 형태가 아닌 부모와 자식 간에 정렬된 형태를 보인다
- 최대값을 찾고 싶은 경우 부모가 자식들보다 큰 값을 유지하는 형태로 만든다(최대 힙)

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

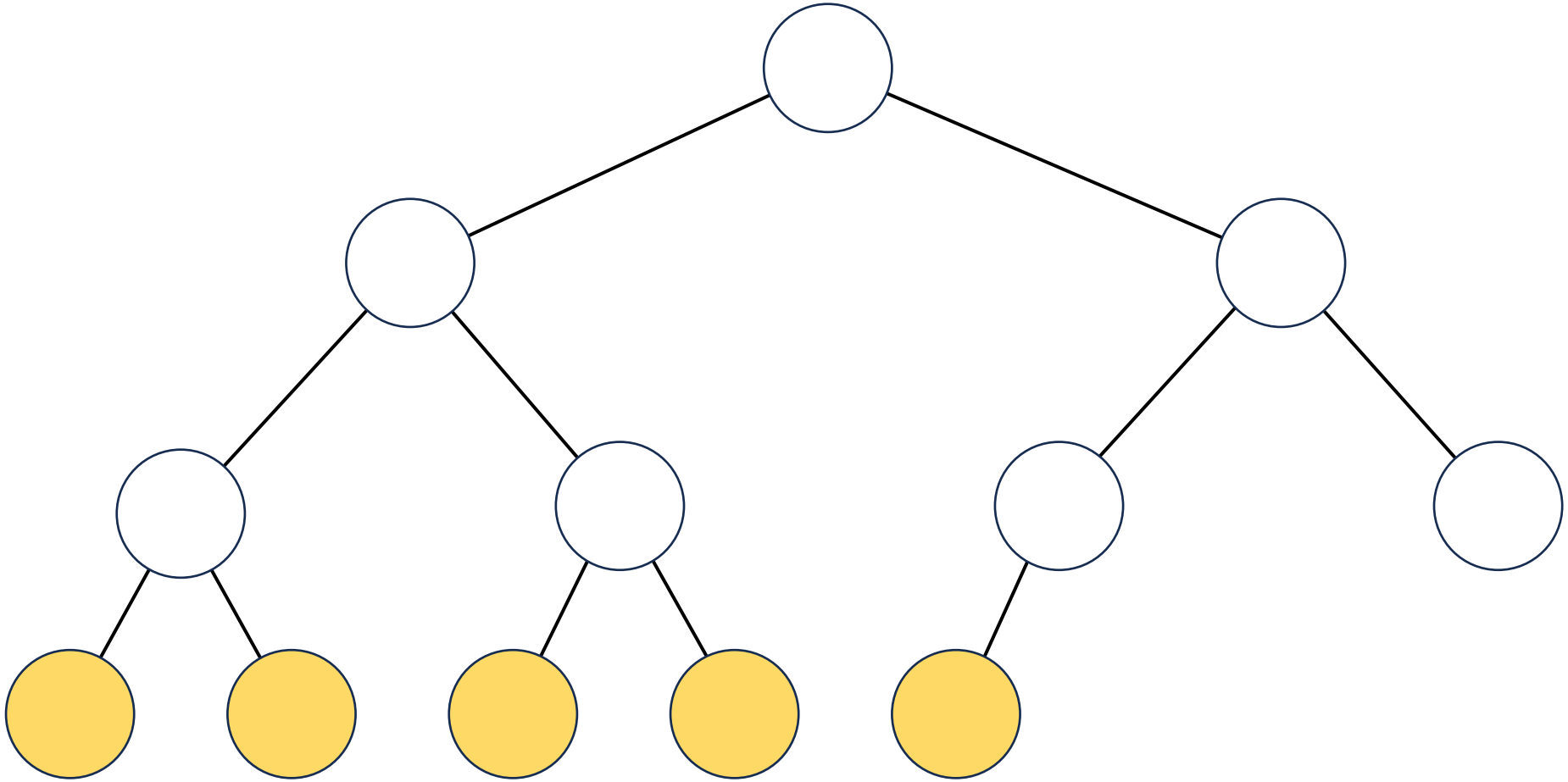
Heap



Heap



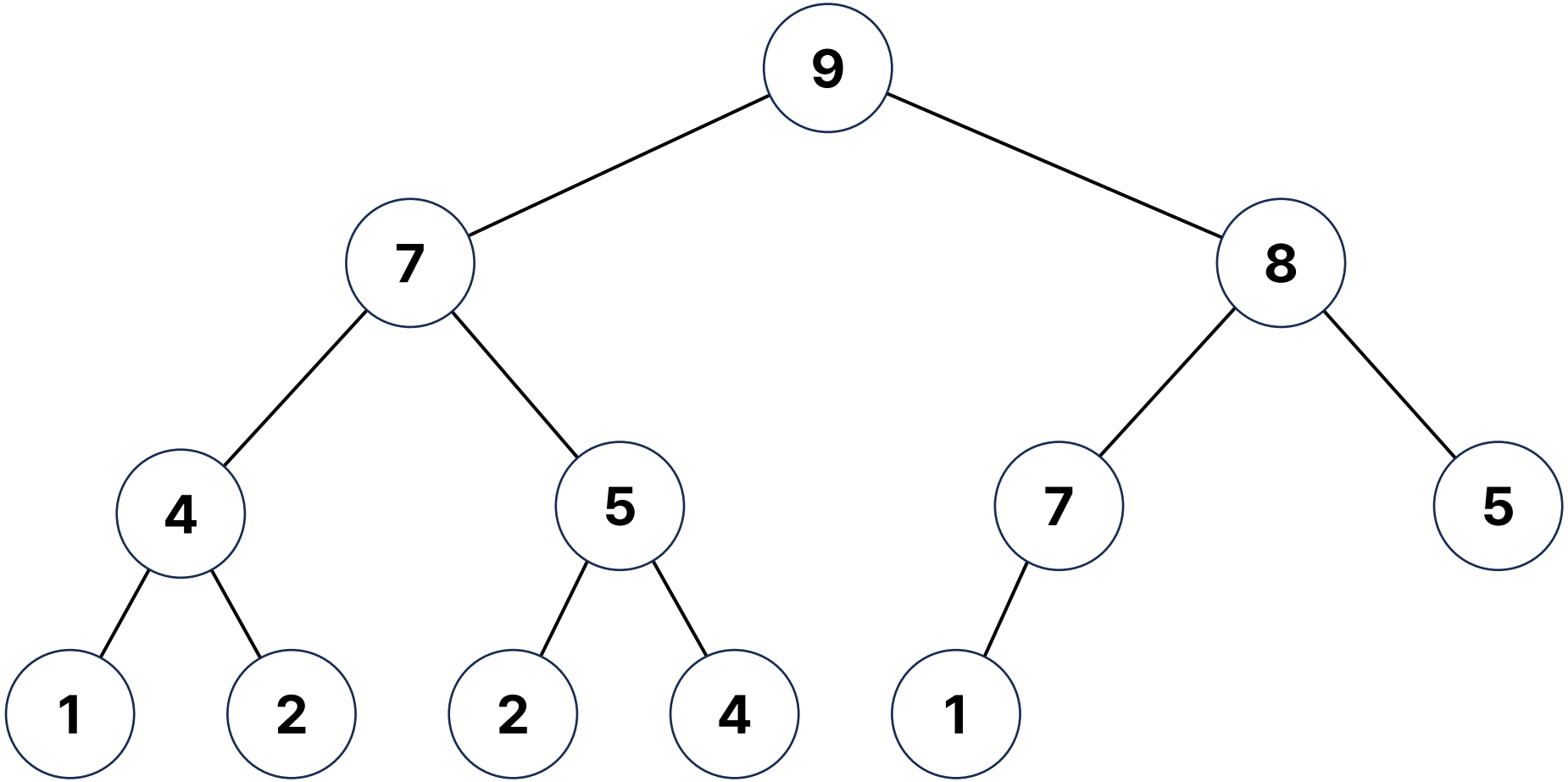
Heap



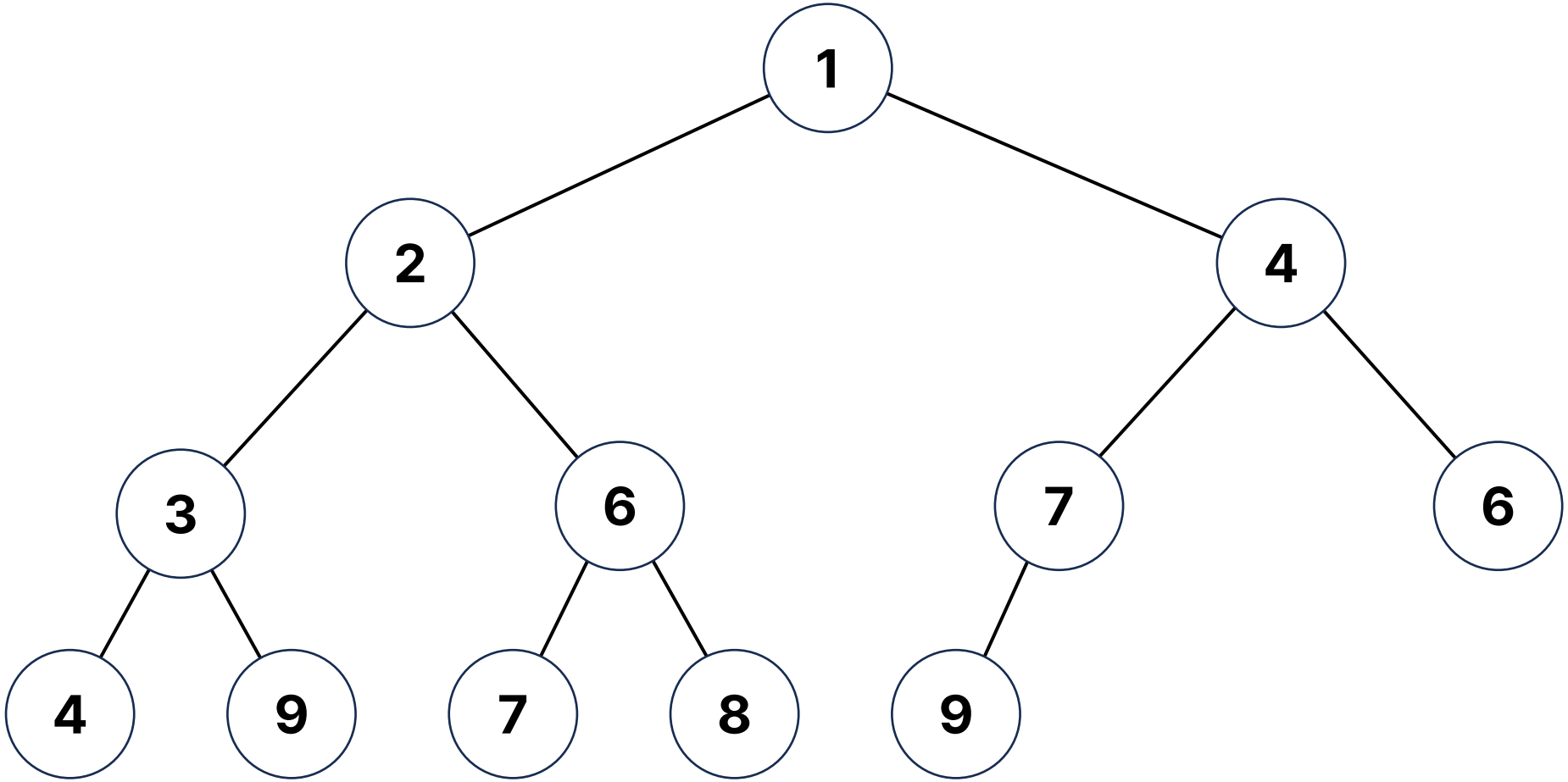
Heap

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

Max Heap



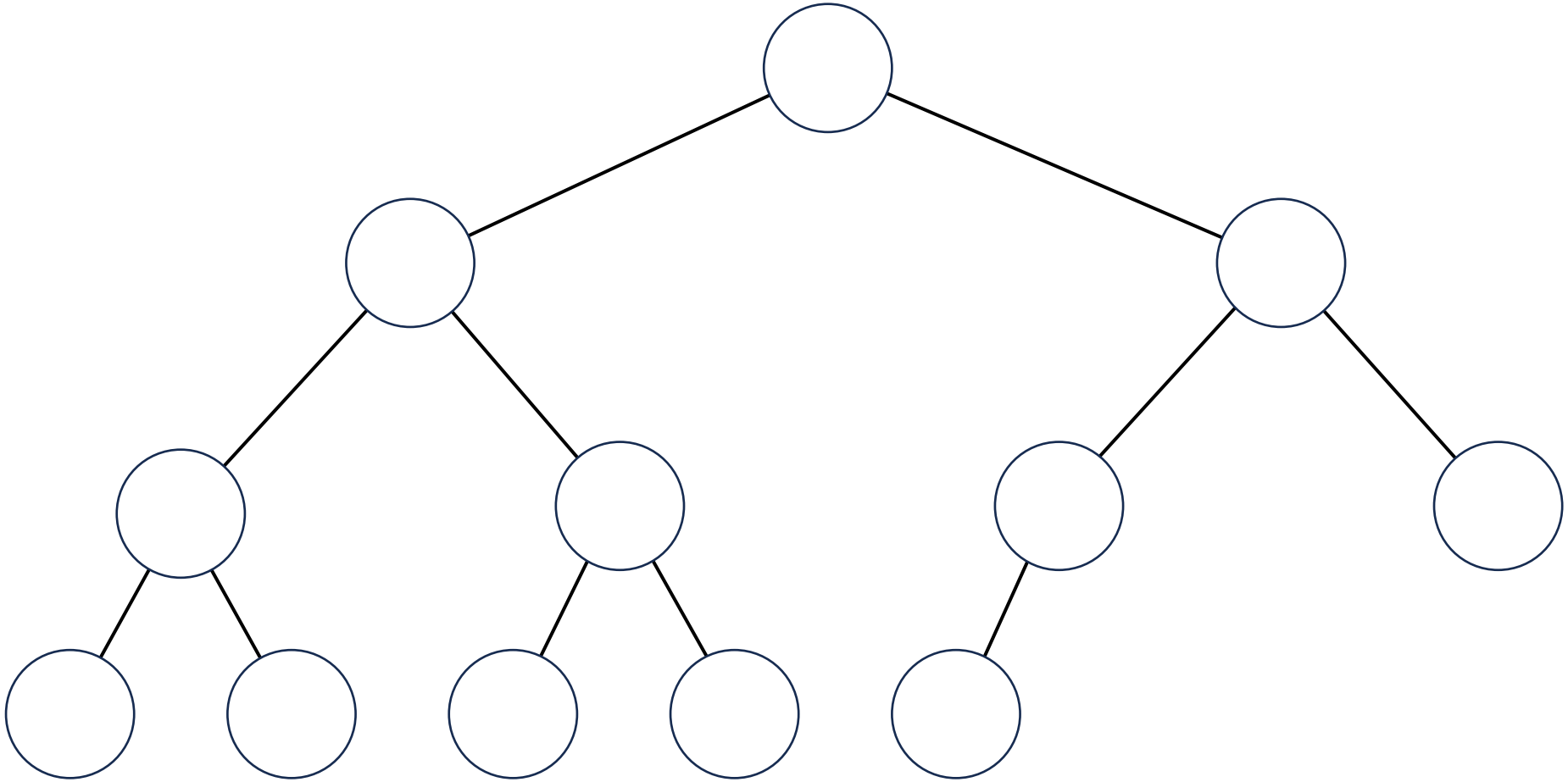
Max Heap



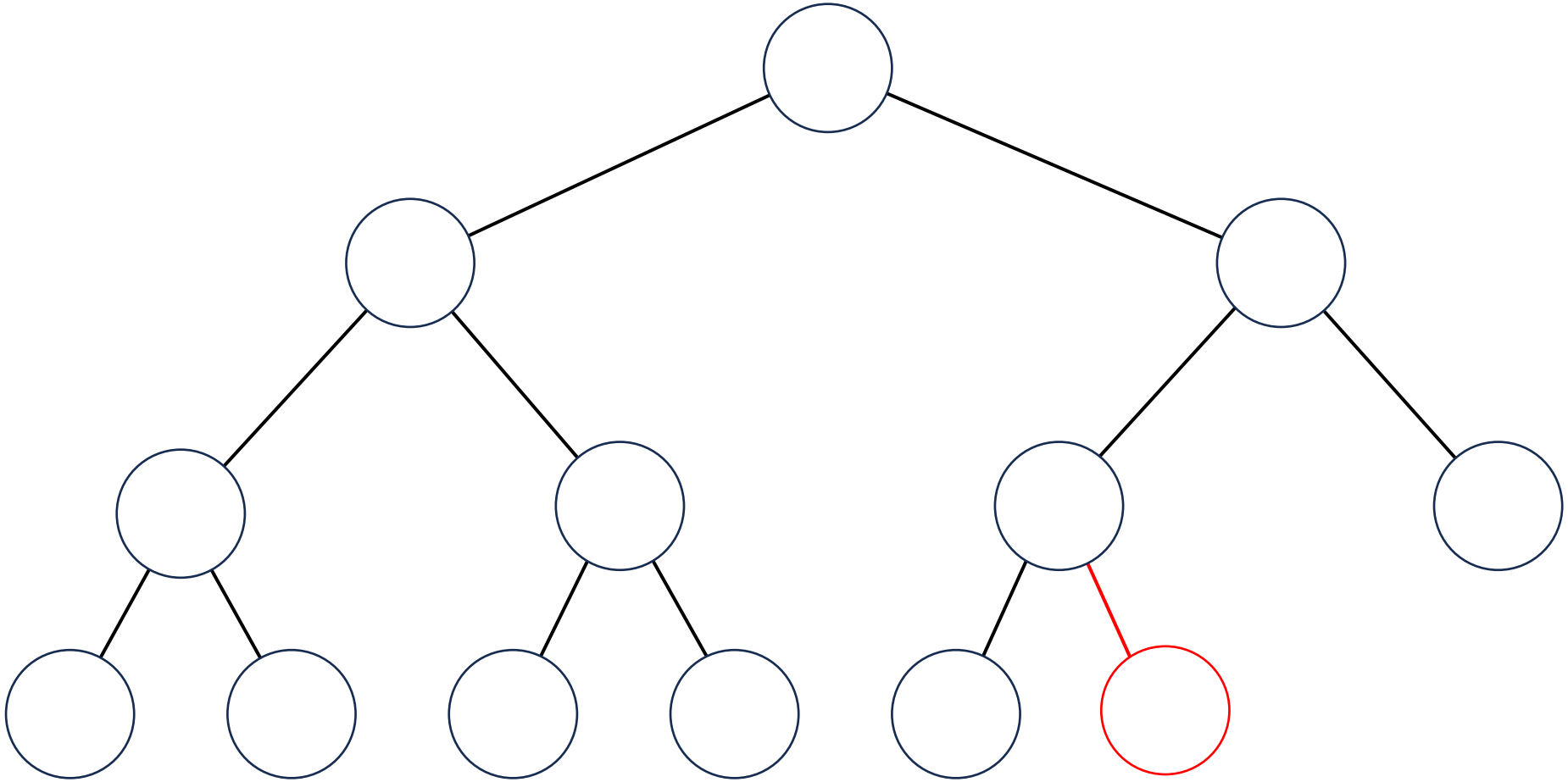
Heap Insert

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

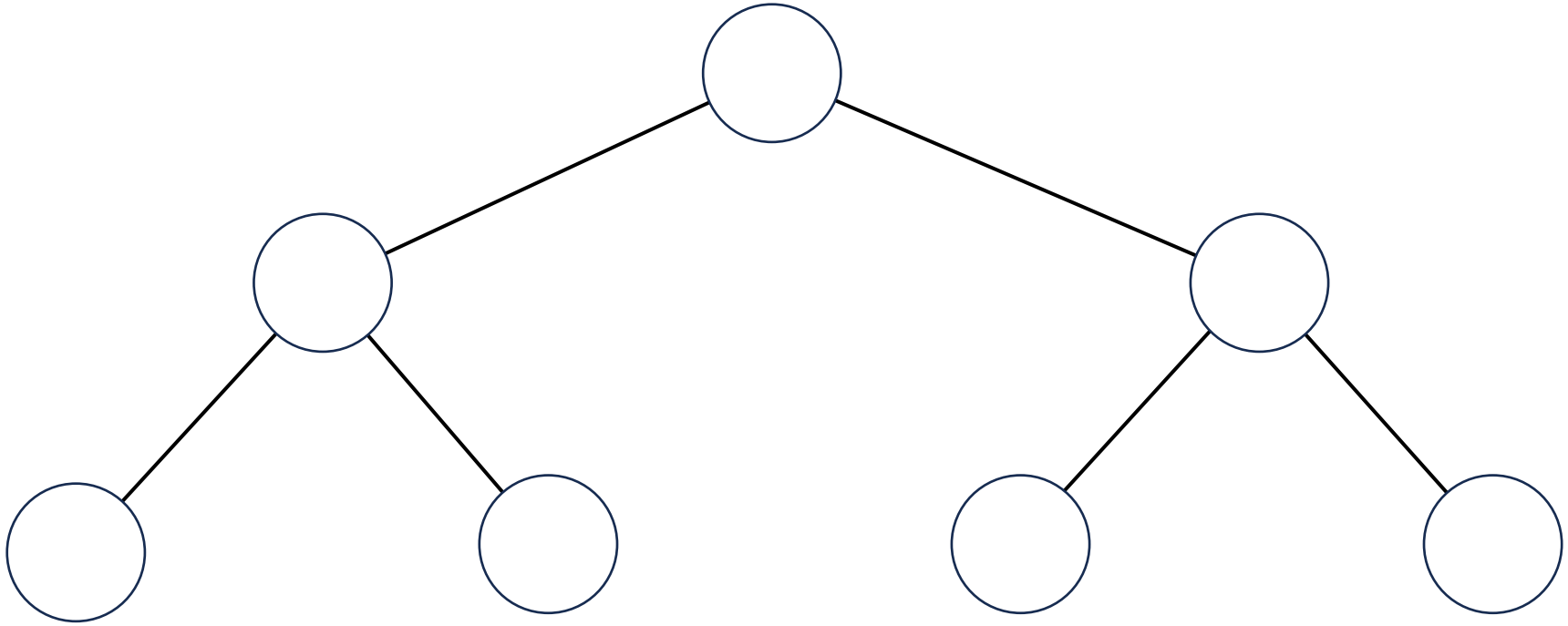
Heap Insert



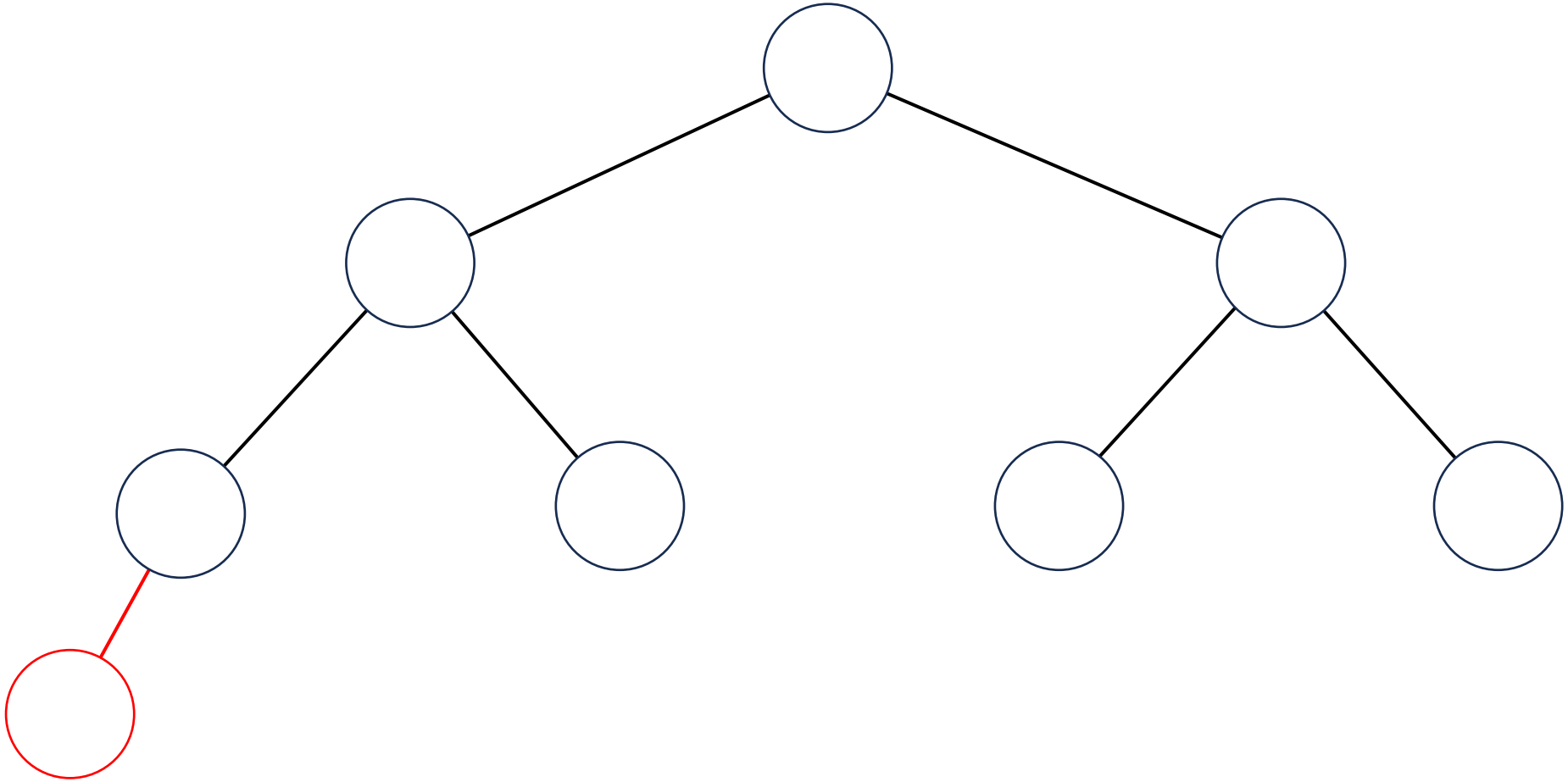
Heap Insert



Heap Insert



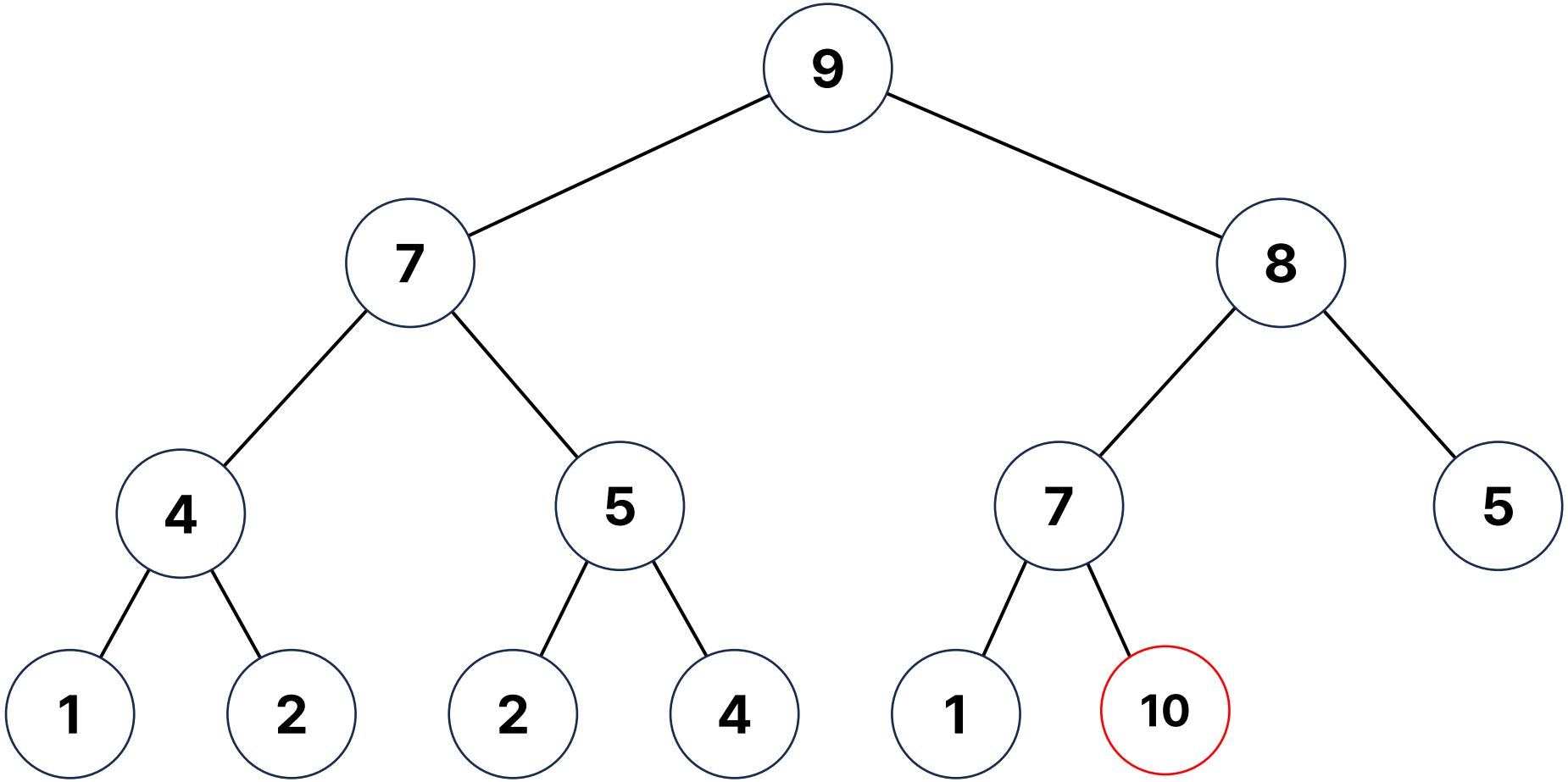
Heap Insert



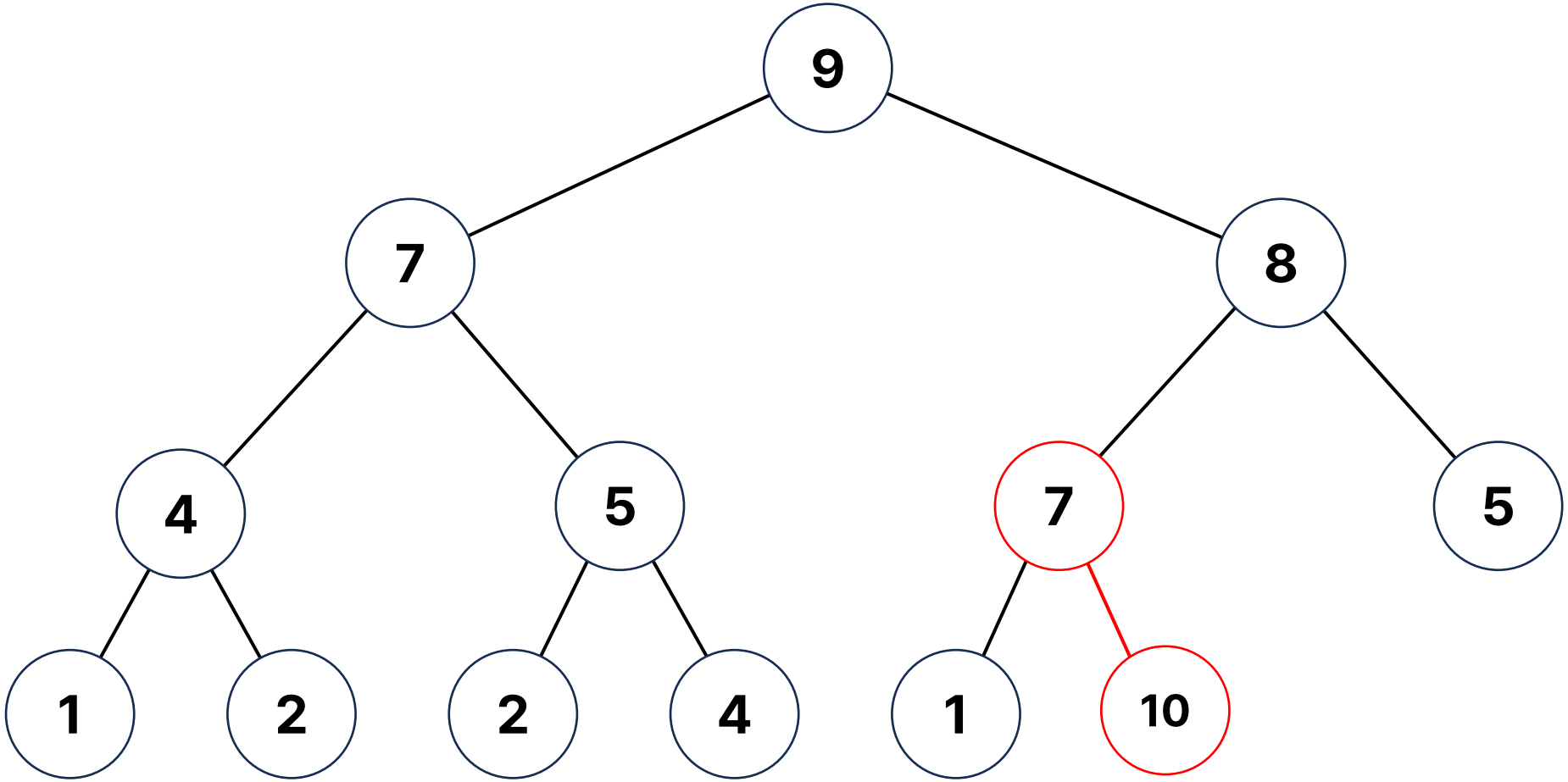
Up-heap Bubbling

- 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

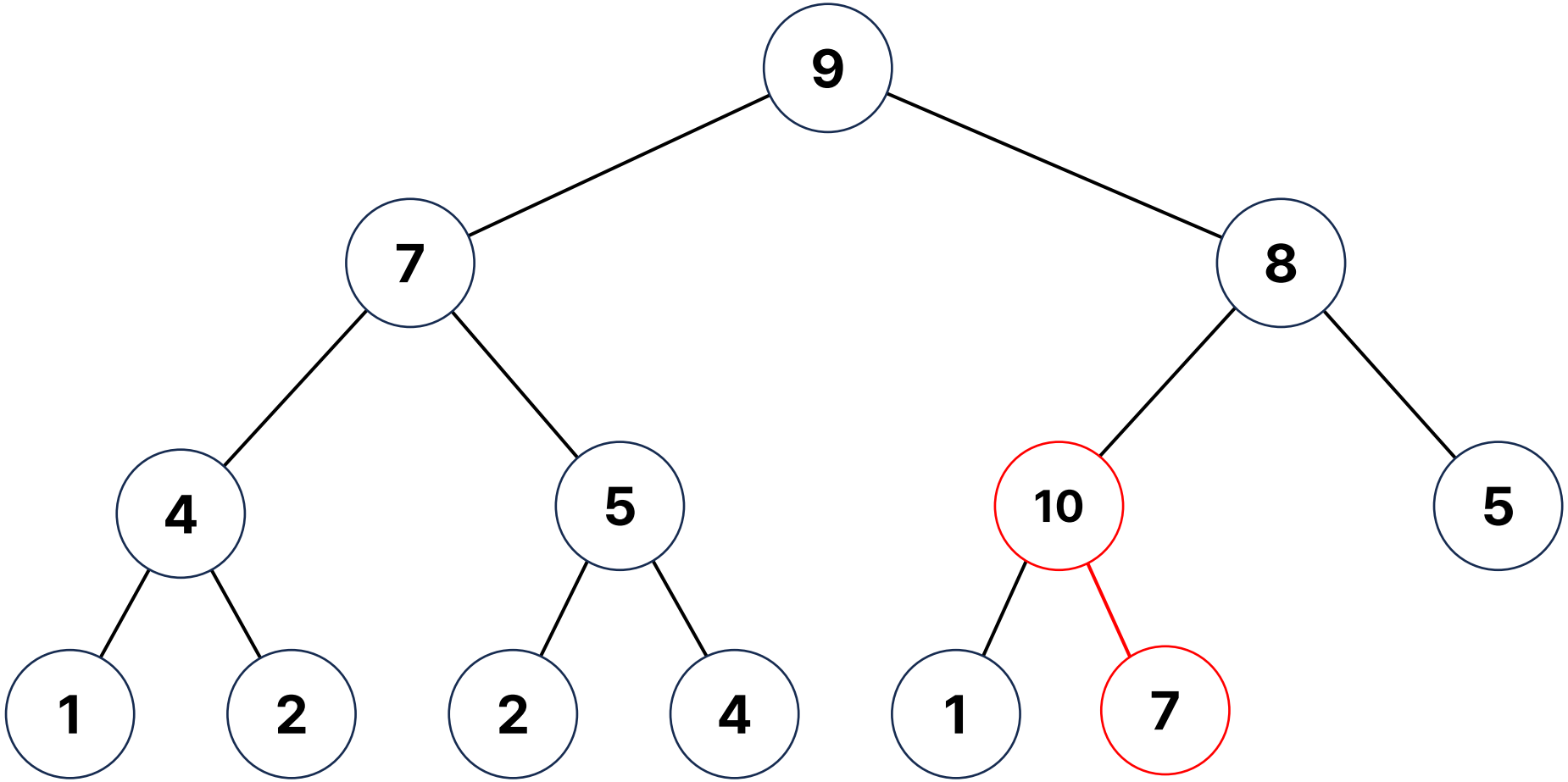
Max Heap Insert



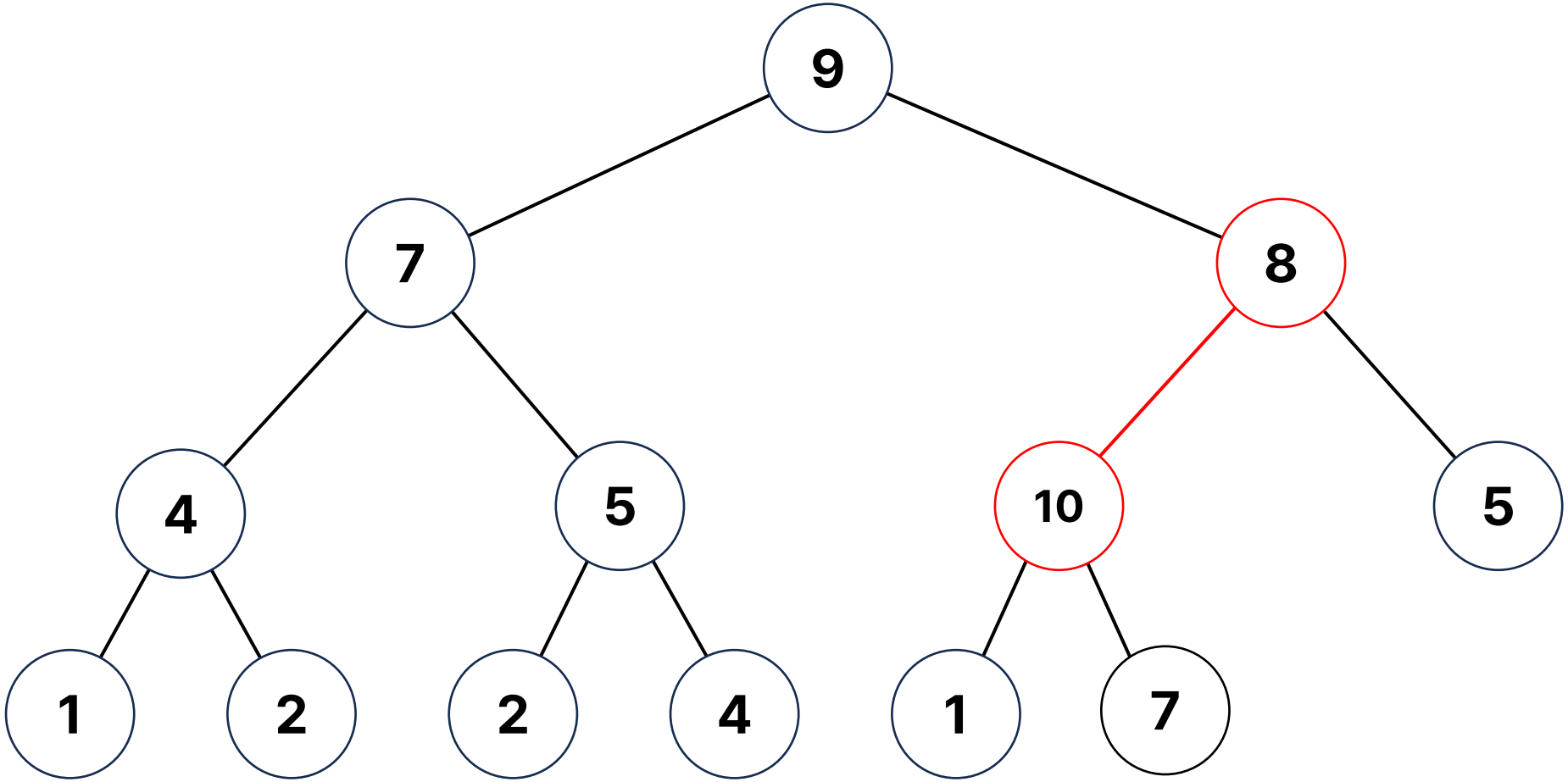
Max Heap Insert



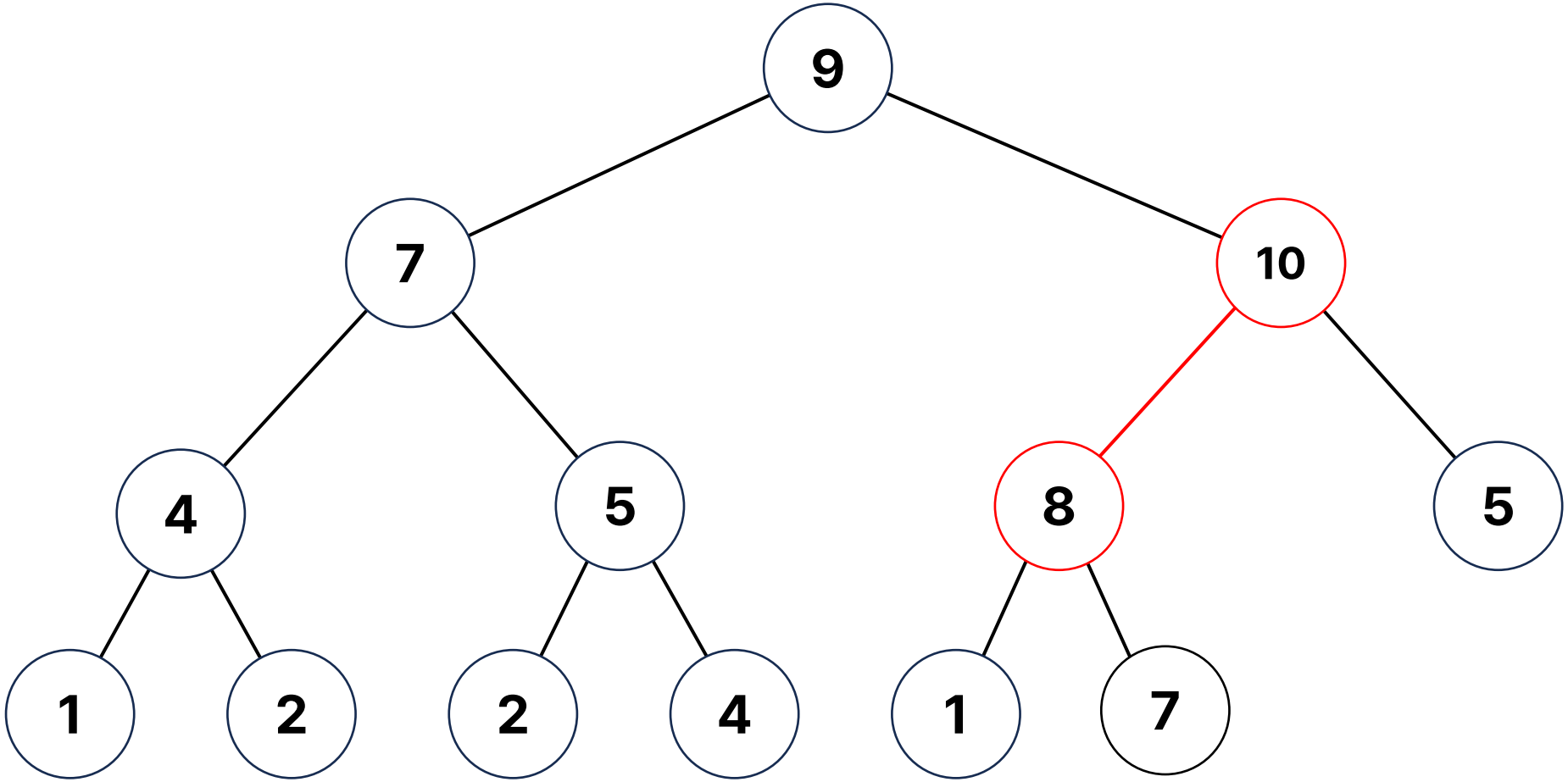
Max Heap Insert



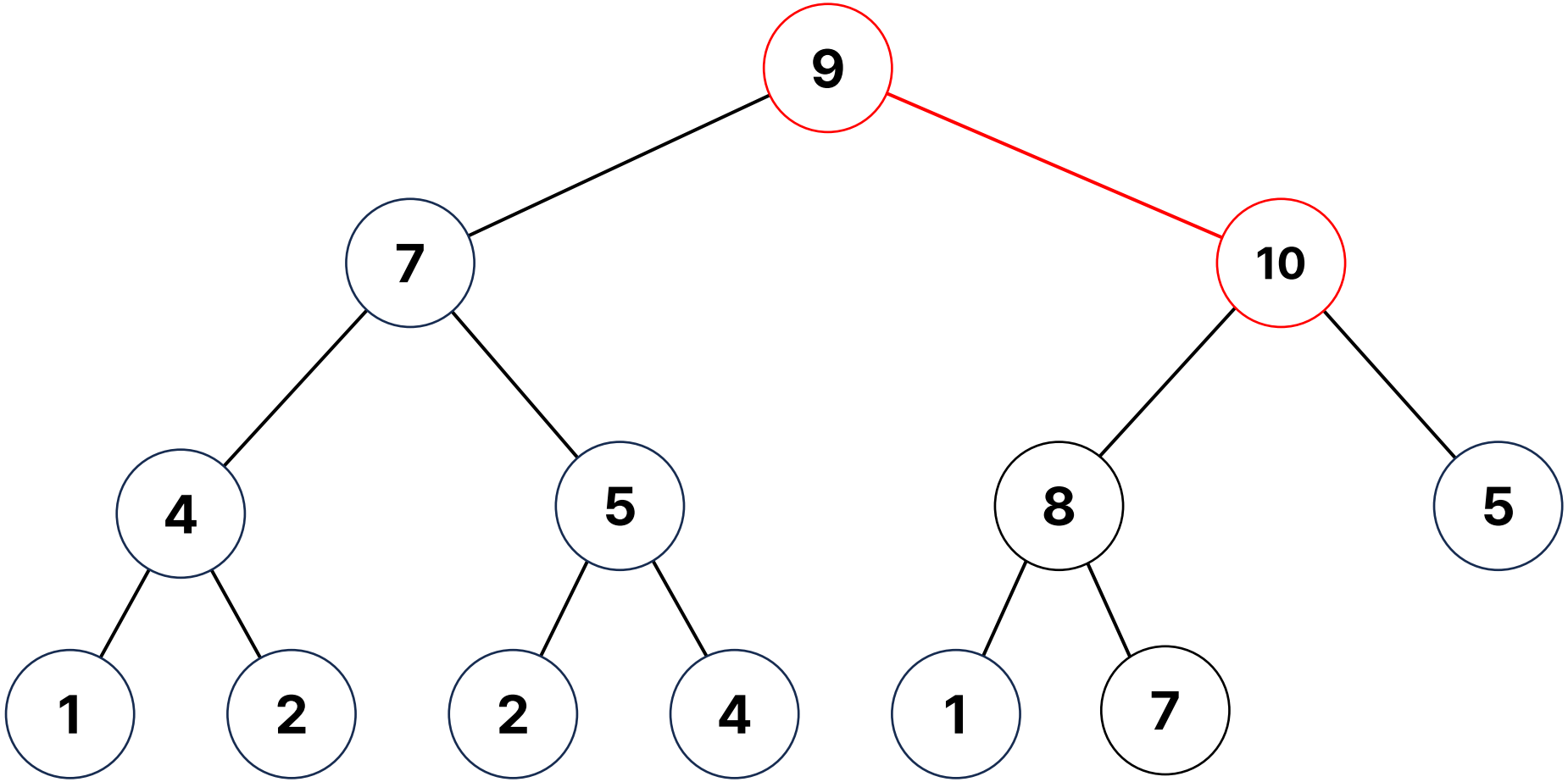
Max Heap Insert



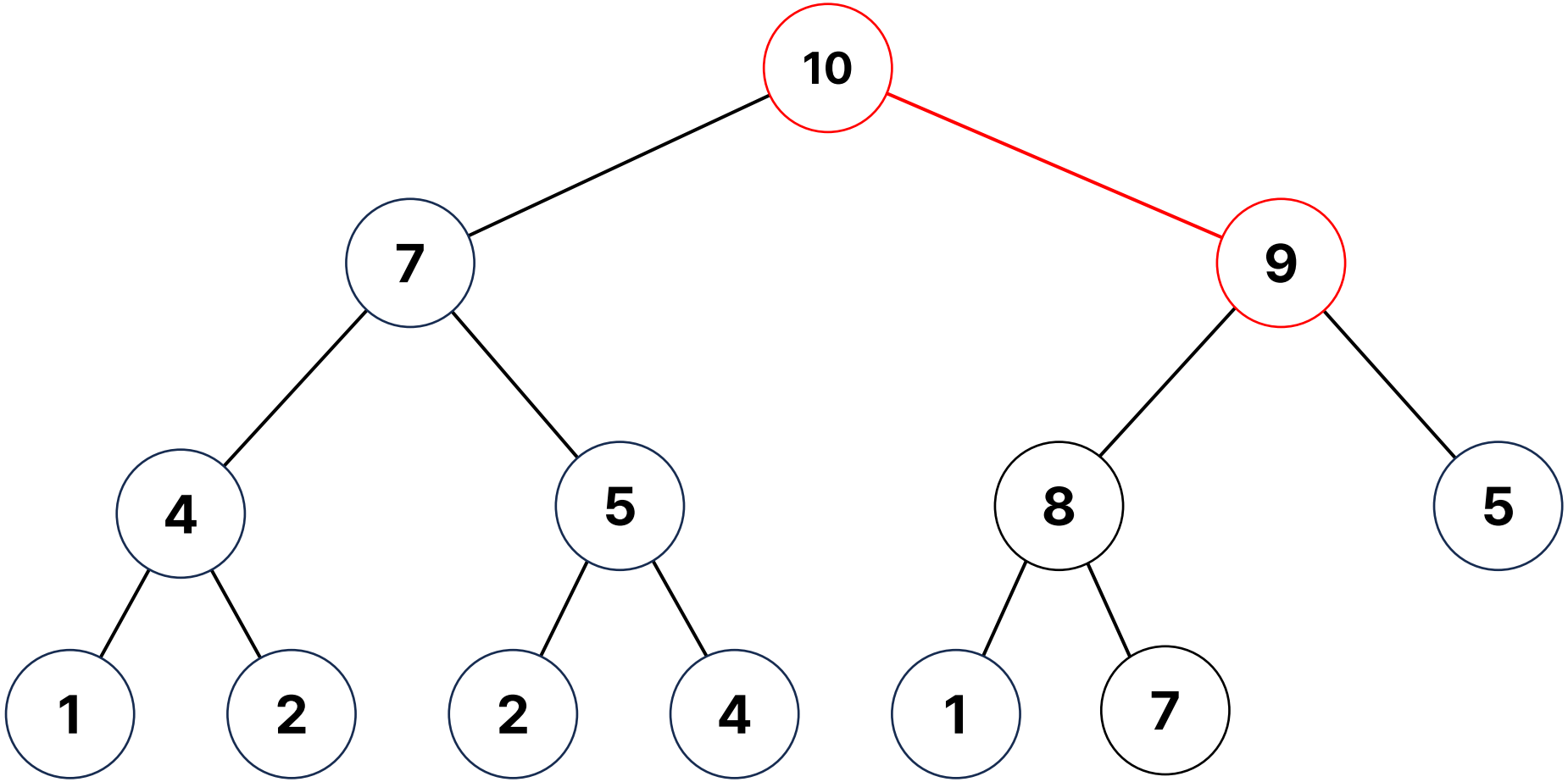
Max Heap Insert



Max Heap Insert



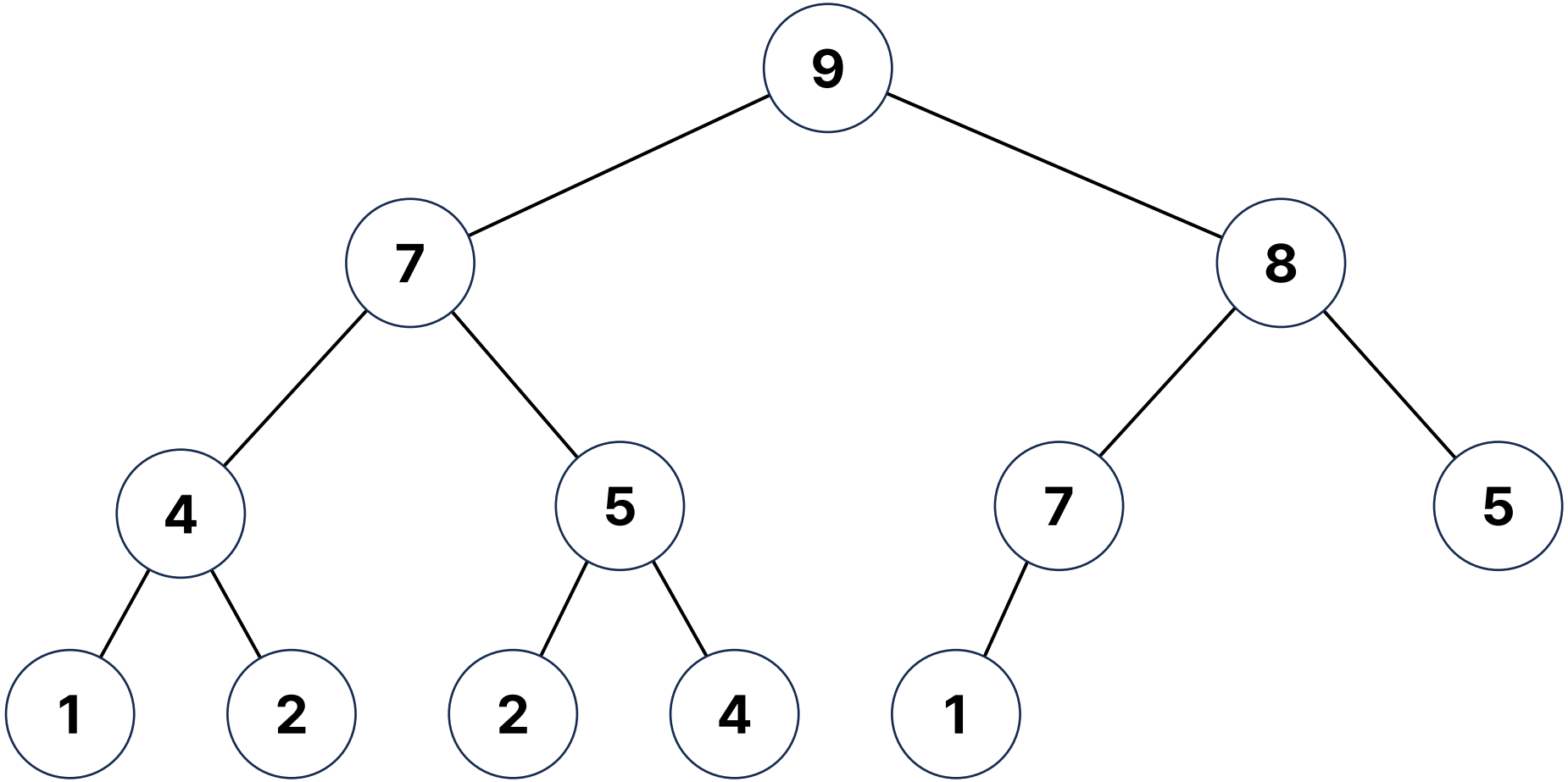
Max Heap Insert



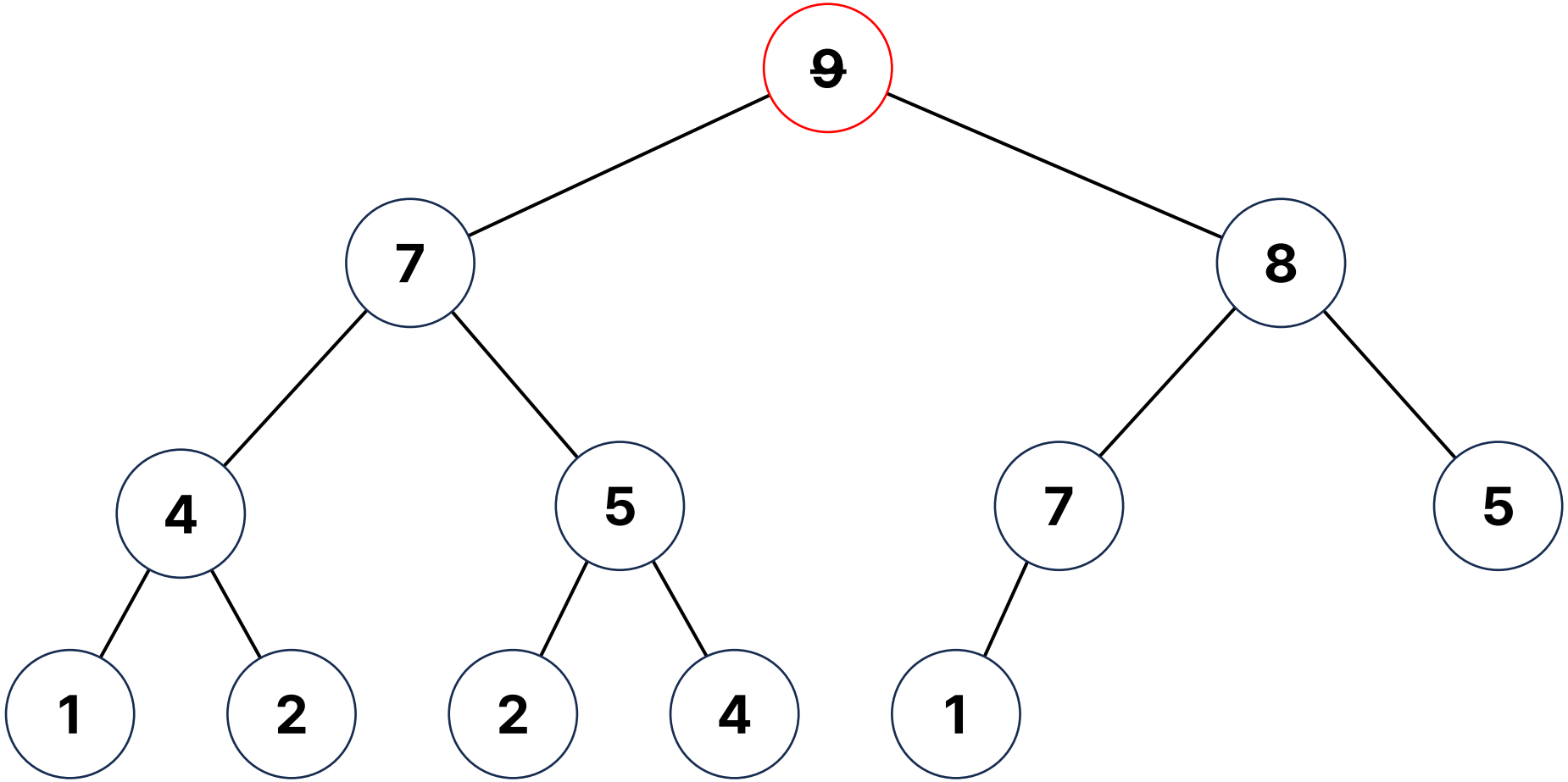
Heap Delete

- 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

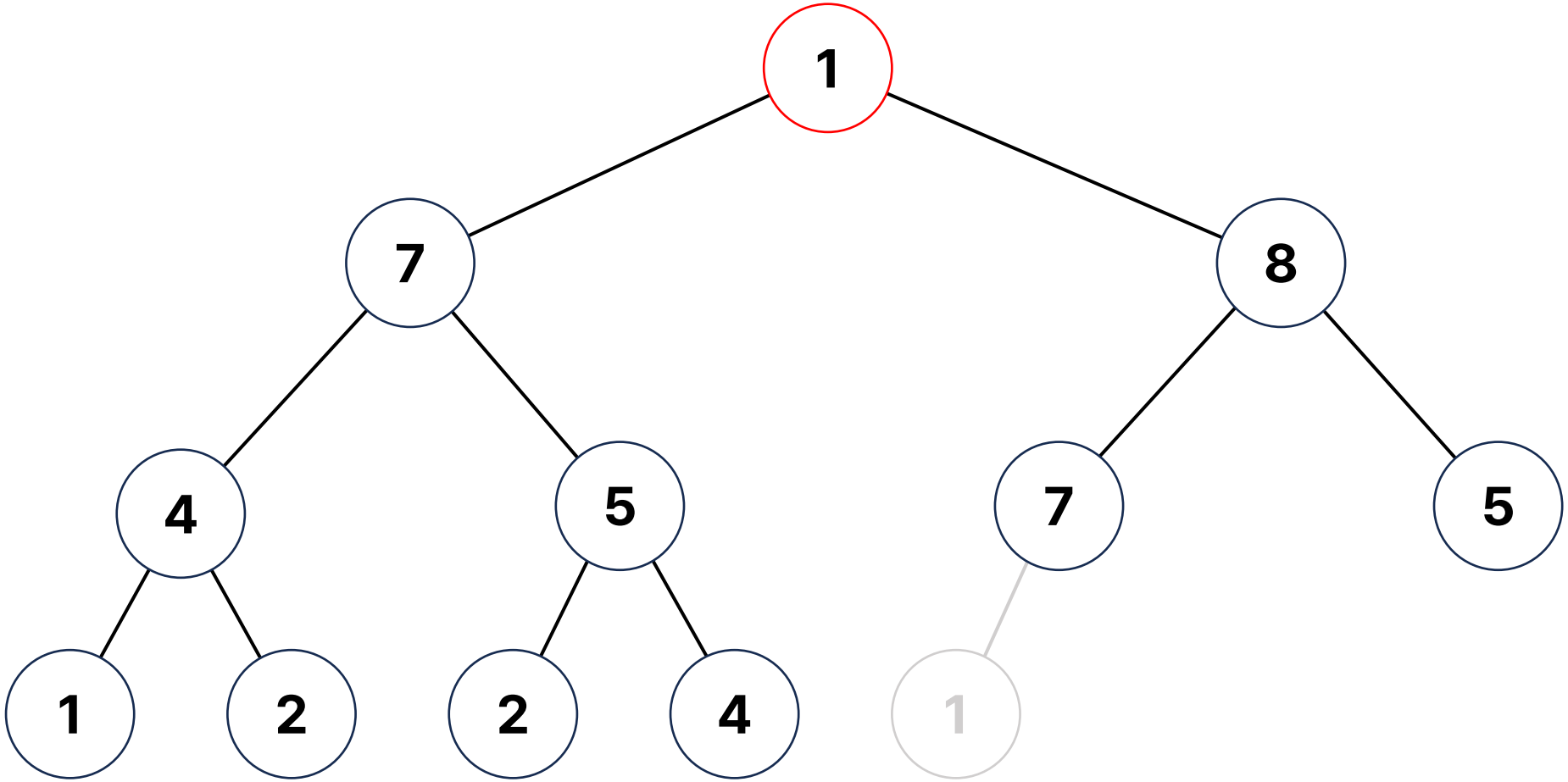
Max Heap Delete



Max Heap Delete



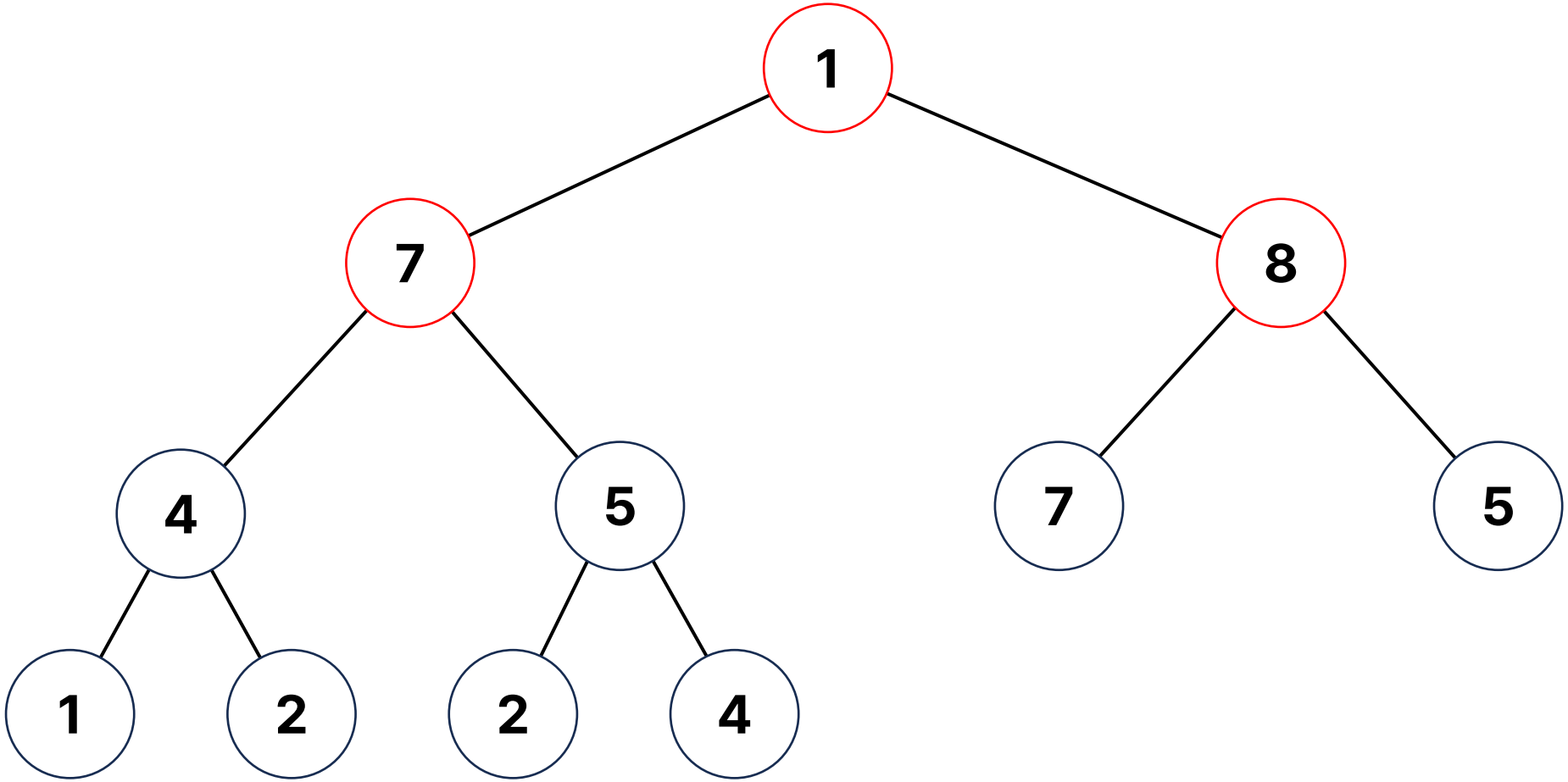
Max Heap Delete



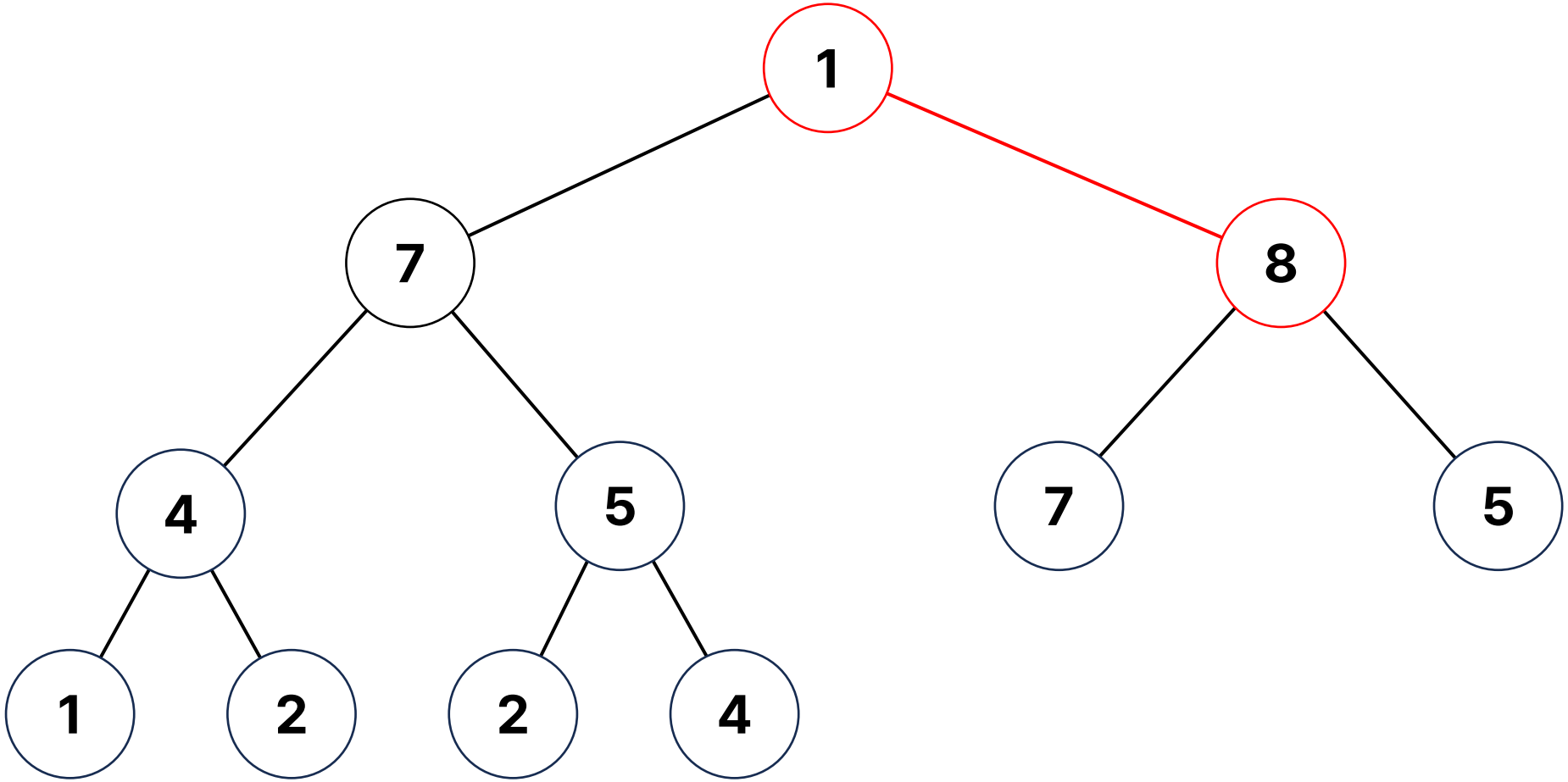
Down-heap Bubbling

- 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

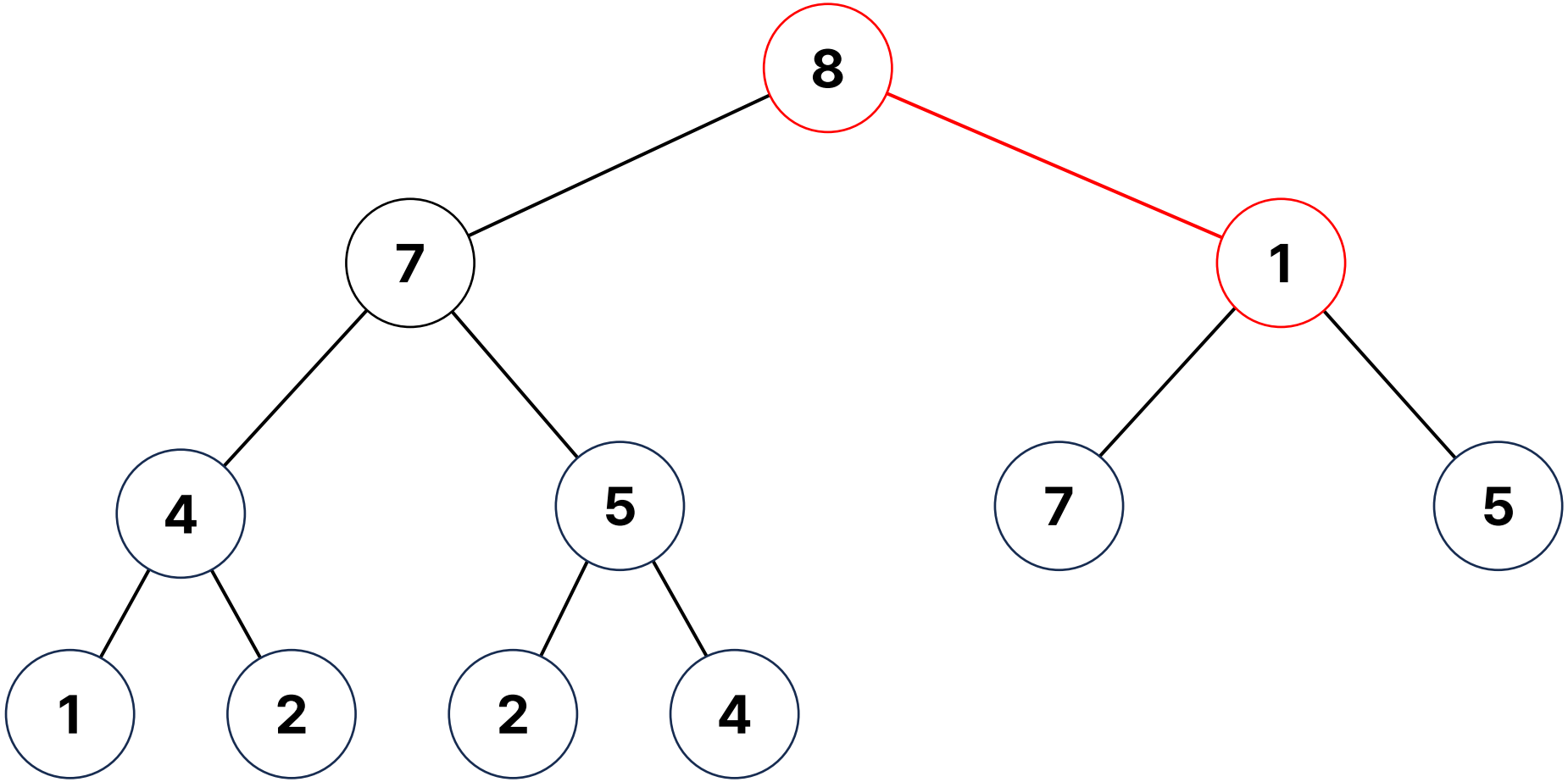
Max Heap Delete



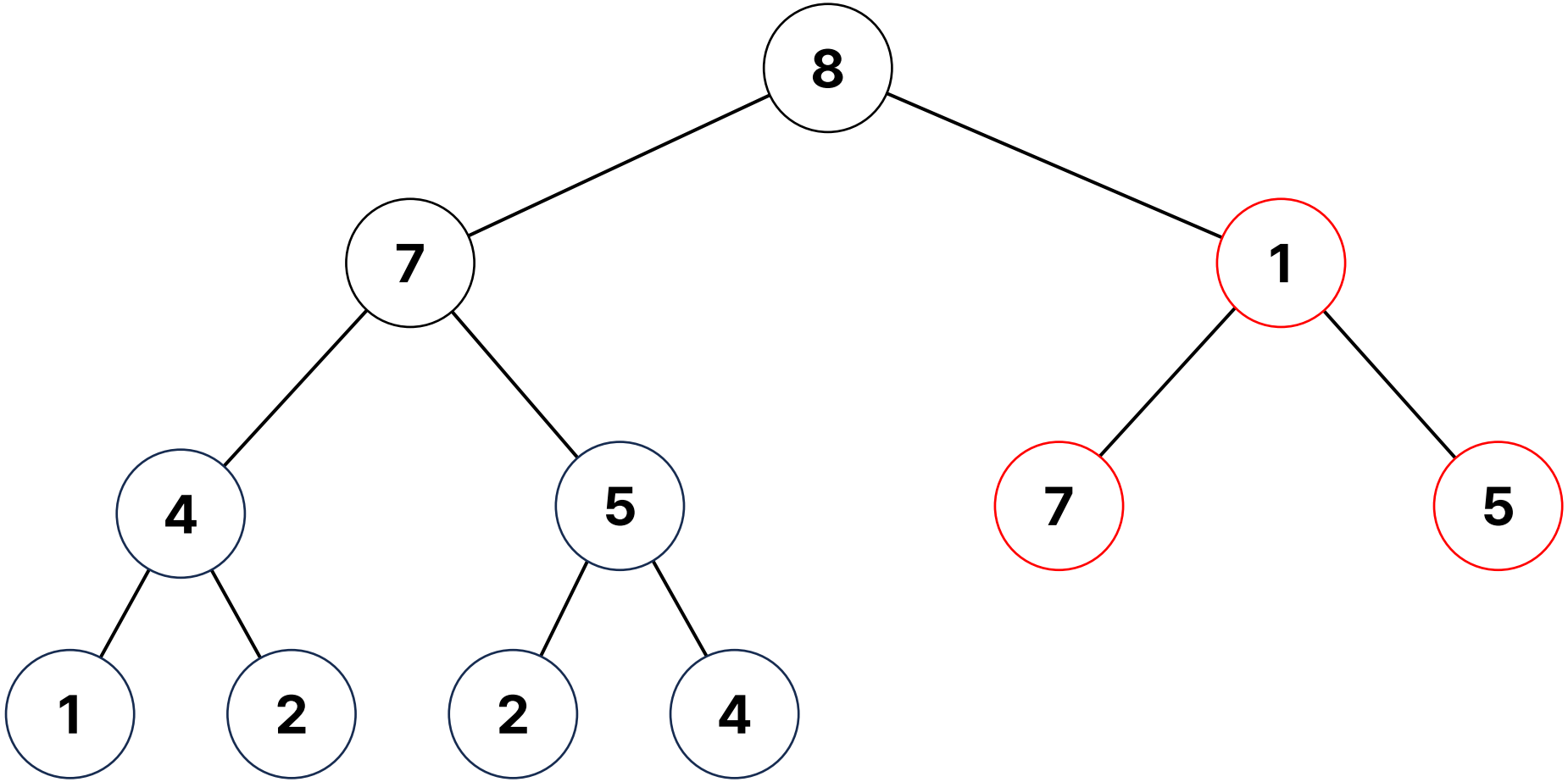
Max Heap Delete



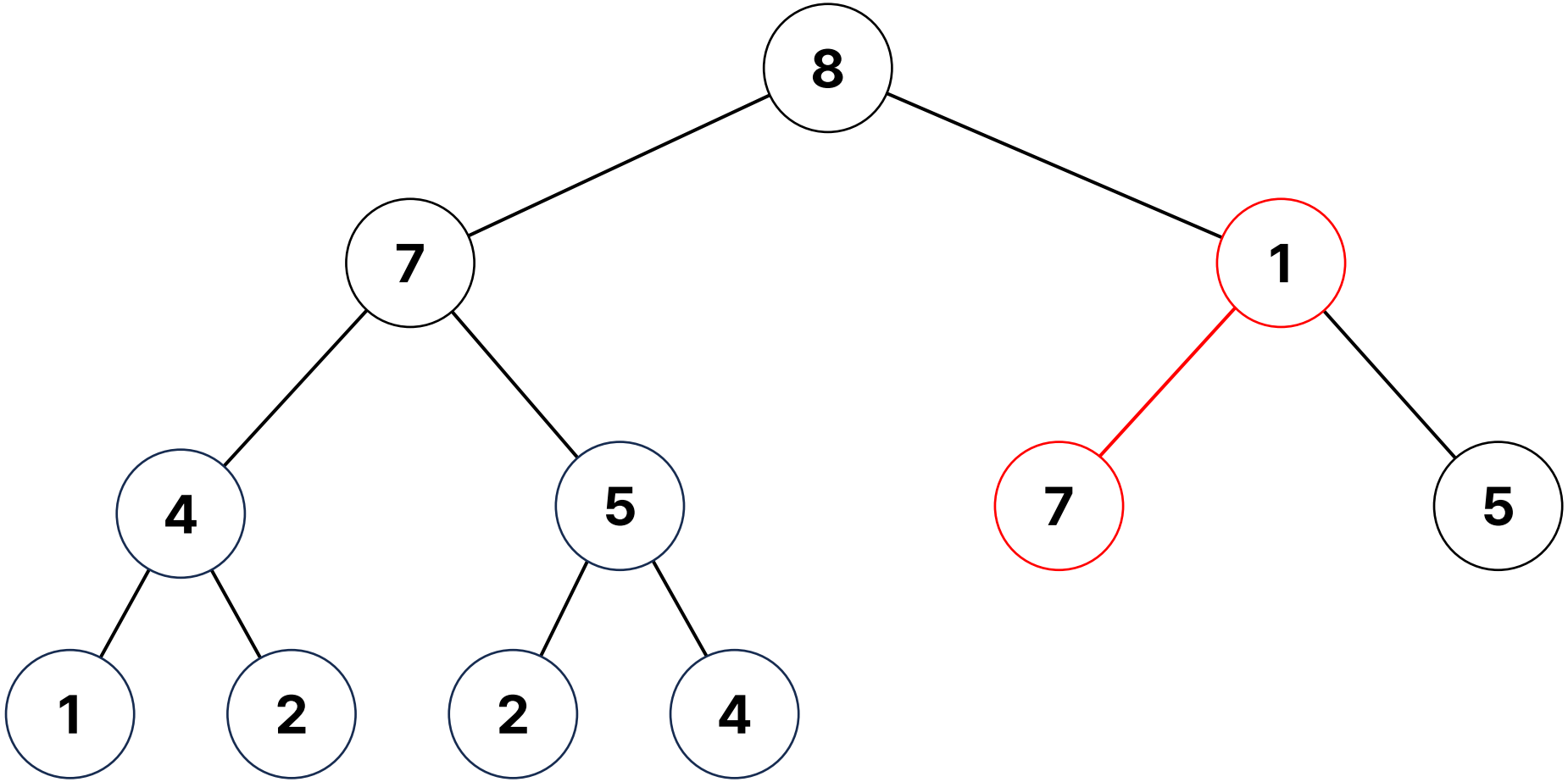
Max Heap Delete



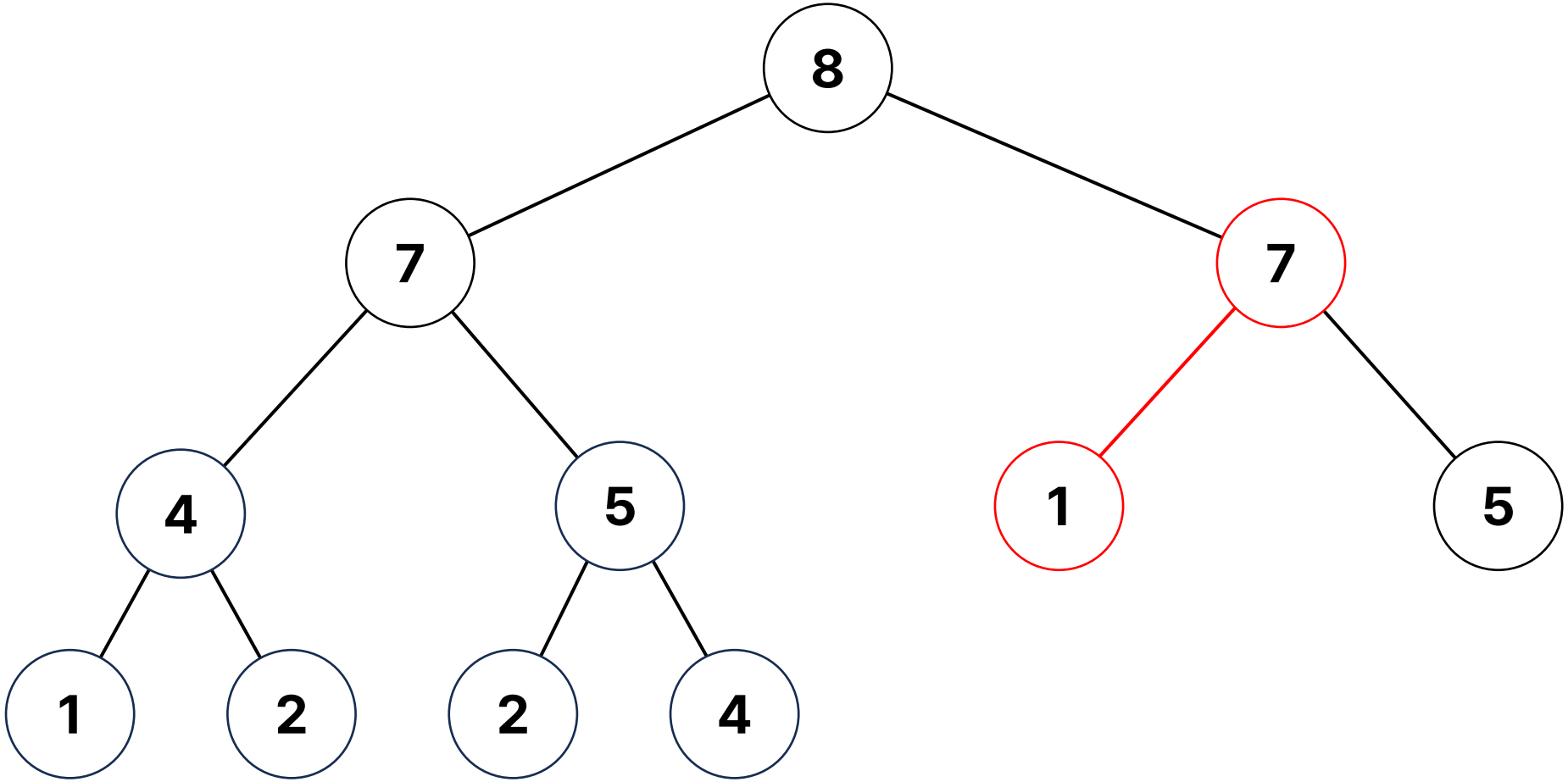
Max Heap Delete



Max Heap Delete



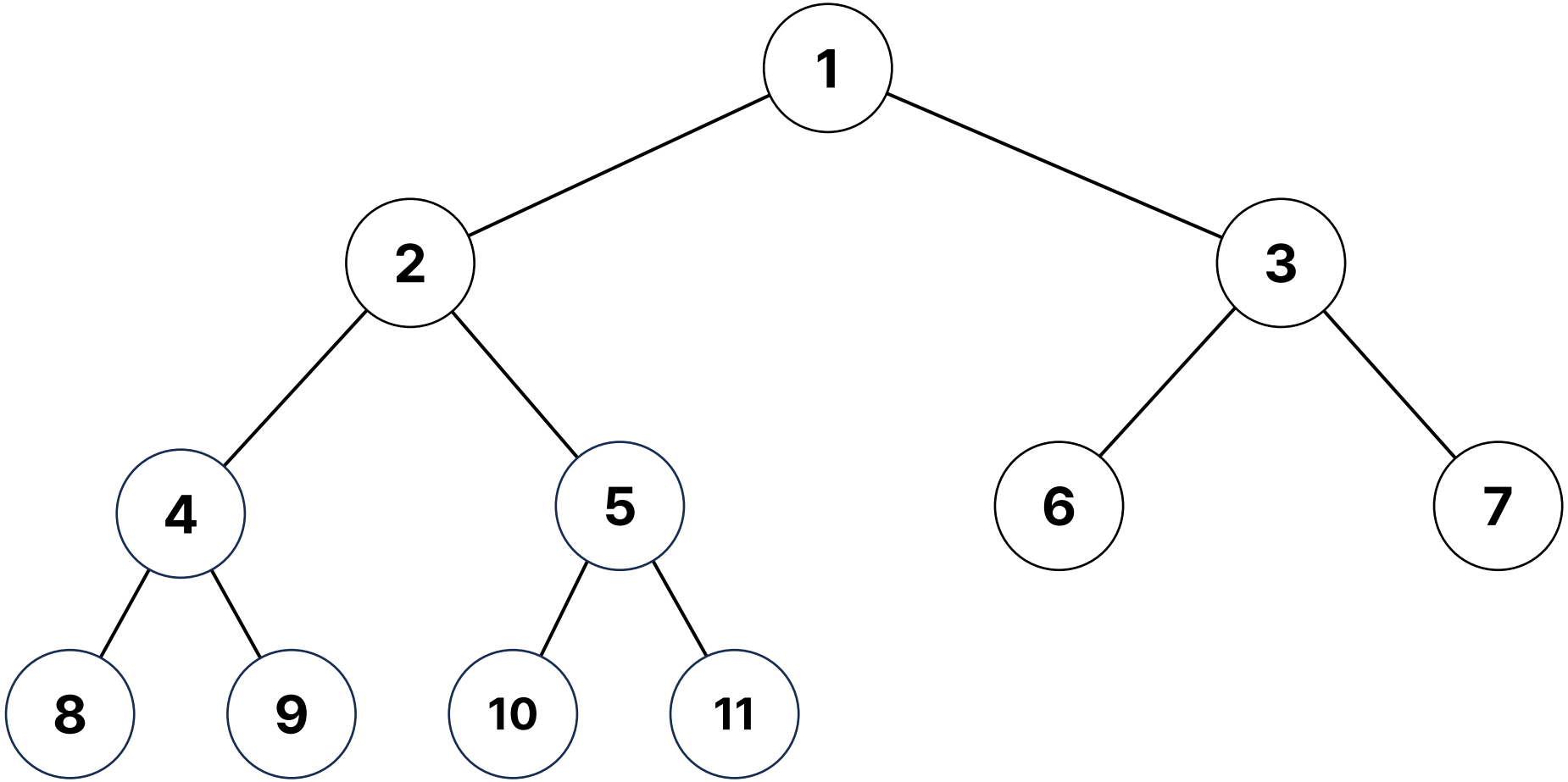
Max Heap Delete



Heap Implementation

- 배열을 이용하여 구현할 수 있다
- 트리 루트의 인덱스를 1로 설정한다
- 왼쪽 자식의 경우 부모 인덱스의 두 배, 오른쪽 자식의 경우 부모 인덱스의 두 배 + 1로 사용한다

Heap Index



Heap Implementation

- 마지막 레벨의 왼쪽부터 채우므로 배열에서 사용되는 인덱스는 1씩 증가한다
- 새로운 노드가 삽입될 때 마지막 인덱스 + 1을 사용하면 된다

- 배열과 size변수를 통하여 트리 전체의 노드 개수와 새로운 노드가 삽입될 위치, 삭제 시 루트로 올릴 노드의 위치를 모두 알 수 있다

Heap Implementation

- Push를 구현하자
- 배열의 마지막에 새로운 노드를 넣는다
- 그 다음 Up-heap Bubbling을 해야한다
- 부모 노드가 자식 노드들보다 크도록 유지한다(최대힙)
- 부모를 찾는 것은 자식 노드의 인덱스를 2로 나누면 찾을 수 있다

Heap Implementation

```
#define MAX_NODE 100000

struct Heap {
    int node[MAX_NODE];
    int sz;

};
```

Heap Implementation

```
void push(int x) {
    node[++sz] = x;
    int cur = sz;

    while (cur > 1) {
        if (node[cur] > node[cur / 2]) {
            swap(node[cur], node[cur / 2]);
            cur /= 2;
        } else
            break;
    }
}
```

Heap Implementation

- Top을 구현하자
- 트리의 루트가 최대 또는 최소값이다
- 루트가 존재하는지 확인 후 리턴

Heap Implementation

```
#define ERROR -1

int top() {
    if (sz == 0)
        return ERROR;
    return node[1];
}
```

Heap Implementation

- Pop을 구현하자
- 트리에 데이터가 없는 경우 에러가 발생한다
- 트리의 루트를 제거하고 마지막 노드를 루트로 가져와야 한다
- 트리의 루트로 값을 가져온 후, Down-heap Bubbling을 해야한다
- 부모 노드가 자식 노드들보다 크도록 유지한다(최대힙)
- 자식 노드 중 큰 값을 확인해야한다

Heap Implementation

- 자식이 없는 경우, 자식이 왼쪽 자식만 존재하는 경우, 둘 다 존재하는 경우 3가지가 존재한다
- 자식이 없는 경우 Down-heap Bubbling은 멈춘다
- 자식이 왼쪽 자식만 존재하는 경우 왼쪽 자식과 값을 비교해 교환하거나 멈춘다
- 둘 다 존재하는 경우 왼쪽 자식과 오른쪽 자식을 모두 비교해 더 큰 값이 존재한다면 큰 값과 교환하고 그렇지 않은 경우 멈춘다

Heap Implementation

```
void pop() {
    if (sz == 0)
        return;
    node[1] = node[sz--];
    int cur = 1;
    while (sz >= cur * 2) {
    // 자식이 존재하는 동안 반복
        if (sz < cur * 2 + 1) {
        // 오른쪽 자식이 없는 경우
            if (node[cur] < node[cur * 2]) {
                swap(node[cur], node[cur * 2]);
                cur = cur * 2;
            } else
                break;
        }…
```

Heap Implementation

```
        else {
        // 양쪽 자식이 모두 있는 경우
            if (node[cur * 2] > node[cur] and node[cur * 2] >= node[cur * 2 + 1]) {
            // 왼쪽 자식이 제일 큰 경우
                swap(node[cur], node[cur * 2]);
                cur = cur * 2;
            }
            else if (node[cur * 2 + 1] > node[cur] and node[cur * 2 + 1] >= node[cur * 2]) {
            // 오른쪽 자식이 제일 큰 경우
                swap(node[cur], node[cur * 2 + 1]);
                cur = cur * 2 + 1;
            } else
                break;
        }
    }
}
```

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

Prefix Sum

Range Sum

- 배열이 주어지고 주어진 구간에 존재하는 수들을 전부 다 더해라
- [l, r] 구간이 주어진 경우 arr[l] + arr[l+1] + ... + arr[r]로 구할 수 있다
- 배열에 최대 N개의 원소가 있을 수 있으므로 O(N)의 시간 복잡도를 따른다

Range Sum

- 이러한 구간합을 Q번 주어진다면 어떨까?
- 단순히 더하는 것을 Q번 반복한다면 O(NQ)의 시간 복잡도를 따를 것이다

- 10만개의 원소가 주어지고 10만개의 쿼리가 주어진다면 100억번의 연산이 필요할 것이다

Range Sum

- 전처리를 통하여 시간을 줄여보자
- 미리 일정 구간을 계산해둔다면 계산하는 과정을 줄일 수 있다

- 그렇다면 어느 구간을 계산해야 하는가?

Prefix Sum

- 전처리를 통하여 i번째 칸에 1번째 원소부터 i번째 원소까지 더한 값을 저장한다
- sum[i] = arr[1] + arr[2] + ... + arr[i];

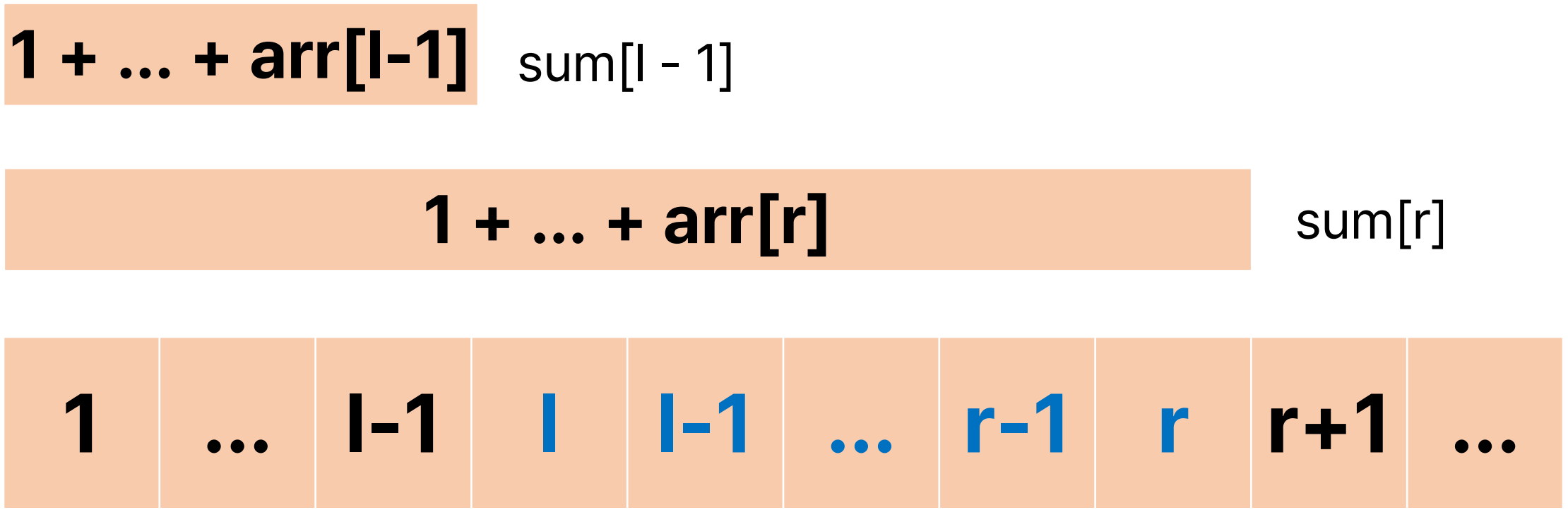
Prefix Sum

- i번째까지 더해둔 값을 사용해서 구간 합을 구해보자
- [l, r]구간의 합은 arr[l] + arr[l + 1] + … + arr[r]이었다
- 미리 구해둔 값을 사용한다면 [l, r]구간의 합을 sum[r] – arr[1] - … - arr[l-1]로 나타낼 수 있다

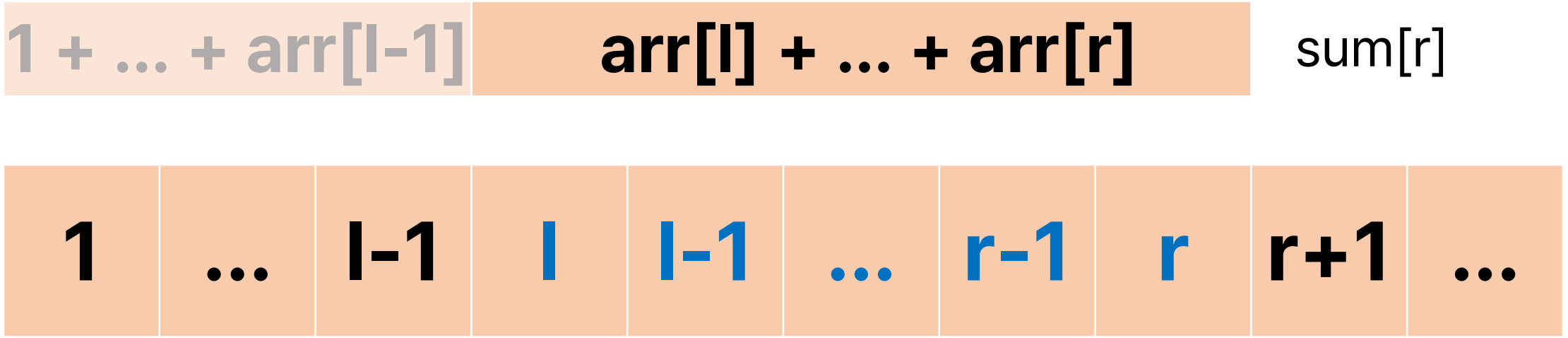
Prefix Sum

- sum[r] – arr[1] - … - arr[l - 1]을 다시 정리하면 sum[r] - (arr[1] + … + arr[l - 1])
  로 나타낼 수 있다
- 괄호 안의 식은 1부터 l - 1까지 더한 것이므로 sum으로 표현할 수 있다
- 따라서 [l, r]구간의 합은 sum[r] - sum[l - 1]로 나타낼 수 있다

Prefix Sum



Prefix Sum



Prefix Sum

- [1,1]을 구하는 것을 생각해보자
- 1번째 원소만 더하면 되므로 배열의 1번째의 인덱스를 0으로 사용한다면 arr[0]으로 구할 수 있다
- 하지만 앞선 식을 적용한다면 sum[0] - sum[-1]이므로 배열의 범위를 벗어난다
- 따라서 일반적으로 sum[0]을 0으로 놔두고 인덱스를 1부터 사용한다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n;

for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] + sum[i];
}
```

Prefix Sum

- sum배열과 arr배열을 따로 나눌 필요가 없다
- 기존 입력 숫자 하나는 [i, i] 구간의 구간합과 동일하다
- 따라서 구간 합을 구할 수 있다면 기존의 입력을 기억할 필요가 없다
- 따라서 배열 하나로 사용하면 된다

Prefix Sum

- l, r의 구간 합을 물어보는 경우 sum[r] - sum[l - 1]로 구할 수 있다
- 전처리 과정에서 O(N)의 시간 복잡도를 요구하고 쿼리는 빨셈 하나로 처리가 가능하므로 O(N + Q)의 시간 복잡도를 따른다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n, q;

cin >> n >> q;
for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] + sum[i];
}

int l, r;
while (q--) {
    cin >> l >> r;
    cout << sum[r] - sum[l - 1] << '\n';
}
```

Prefix Sum

- 구간 합 뿐만 아니라 구간의 복구가 가능하다면 다른 방식으로도 접근이 가능하다
- 구간 곱 또는 구간 XOR 등을 같은 방식으로 구현할 수 있다

Prefix Sum

- 구간 곱의 경우, 첫번째 원소부터 i번째 원소까지 모두 곱한 값을 기억하고 있는다
- [l, r]구간에 해당하는 구간 곱을 알고 싶은 경우 r까지 곱한 값에서 l-1까지 곱한 값을 나누면 된다
- 앞선 값에 계속해서 곱하므로 0번째 배열의 초기값은 1이어야 한다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n, q;

sum[0] = 1;
cin >> n >> q;
for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] * sum[i];
}

int l, r;
while (q--) {
    cin >> l >> r;
    cout << sum[r] / sum[l - 1] << '\n';
}
```

Prefix Sum

- XOR의 경우 1번 XOR한 경우 본인과 값이 동일하며, 2번 XOR한 경우 0으로 돌아간다
- 이것을 이용하여 곱셈에서 나눗셈을 해 원하는 구간의 값을 없애는 것처럼 다시 한번 XOR해 원하는 구간의 값을 없앨 수 있다
- [l, r]구간에 해당하는 구간 XOR을 알고 싶은 경우 r까지 XOR한 값에서 l-1까지 XOR한 값을 XOR하면 된다
- 0번째 배열의 초기 값은 0이다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n, q;

cin >> n >> q;
for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] ^ sum[i];
}

int l, r;
while (q--) {
    cin >> l >> r;
    cout << sum[r] ^ sum[l - 1] << '\n';
}
```

Prefix Sum

- 누적 합을 사용할 때는 자료형에 주의해야한다
- 마지막 값의 경우 1번째부터 N번째까지 모든 값을 더하거나 곱한 값을 저장하고 있다
- 즉, 값이 매우 커지므로 사용하려는 범위 안에 포함되는지 생각하고 변수를 선언해야한다

Prefix Sum

- 원소가 100,000개이며 각 원소는 최대 100,000의 값을 가진다면 최대 누적 합은 10,000,000,000이다
- int의 범위가 2,147,483,647이므로 int의 범위를 넘어간다
- 다음과 같은 경우 long long으로 sum배열을 만들어야 한다

문제

- 구간 합 구하기 4 BOJ 11659
- 구간 합 구하기 5 BOJ 11660
- 낚시 BOJ 30461