24차시

Graph

Graph Review

- Graph: 정점(Node, Vertex)과 간선(Edge)으로 이루어진 자료구조
- Directed Graph, 유향 그래프, 방향 그래프

  : 간선의 방향성이 존재
- Undirected Graph, 무향 그래프, 양방향 그래프

  : 간선의 방향성이 존재하지 않다, 간선이 양방향
- Weight Graph, 가중치 그래프

  : 간선에 가중치가 존재하는 그래프

Graph Review

- Simple Path, 단순 경로: 경로 중에서 정점을 중복없이 방문하는 경로
- Cycle, 사이클: 시작 노드로 다시 돌아오는 경로
- Cycle Graph, 순환 그래프: 사이클이 존재하는 그래프
- Acyclic Graph, 비순환 그래프: 사이클이 존재하지 않는 그래프

Graph Review

- Connected Component, 연결 요소: 간선들로 연결된 노드들의 집합

  방향 그래프에서는 Strongly Connected Component, Weakly Connected Component로 구분

- Degree, 차수: 노드와 연결된 간선의 수
- In Degree, 진입 차수: 노드로 들어오는 간선의 수
- Out Degree, 진출 차수: 노드에서 나가는 간선의 수

Graph Review

- 인접 리스트: 노드마다 연결된 간선을 리스트로 저장하는 방식
- 인접 행렬: 2차원 배열, 행렬로 간선을 저장하는 방식

Graph Review

- Traversal, 탐색: 시작 노드와 연결된 모든 정점을 탐색하는 것

- DFS, 깊이 우선 탐색: 이동할 수 있는 노드가 존재한다면 먼저 이동 후 탐색 하는 것
- BFS, 너비 우선 탐색: 시작노드와 가까운 노드부터 탐색을 하는 것
- DFS는 방문 이후 방문 여부를 기록해도 되지만 **BFS는 큐에 넣는 순간 노드 방문을 기록**해야 한다

문제

- DFS와 BFS BOJ 1260
- 죽음의 게임 BOJ 17204
- ~~거리가 k이하인 트리 노드에서 사과 수확하기 BOJ 25516~~
- 섬의 개수 BOJ 4963
- 안전 영역 BOJ 2468

DFS와 BFS BOJ 1260

- DFS와 BFS를 구현
- 노드를 방문하는 것은 노드 번호가 작은 순서대로 방문해야 한다
- 인접 리스트를 사용한다면 정렬 후 순회, 인접 행렬을 사용하면 작은 번호부터 반복문 사용

DFS와 BFS BOJ 1260

```cpp
#define MAX_NODE 1001

// Graph
vector<int> edges[MAX_NODE];
int matrix[MAX_NODE][MAX_NODE];

// DFS
bool dfs_visited[MAX_NODE];

// BFS
bool bfs_visited[MAX_NODE];
queue<int> bfs_queue;
```

DFS와 BFS BOJ 1260

```
int n, m, v;
int src, dst;

cin >> n >> m >> v;
while (m--) {
cin >> src >> dst;
    edges[src].push_back(dst);
    edges[dst].push_back(src);
}
for (int i = 1; i <= n; i++)
    sort(edges[i].begin(), edges[i].end());

dfs(v);
bfs(v);
```

DFS와 BFS BOJ 1260

```
int n, m, v;
int src, dst;

cin >> n >> m >> v;
while (m--) {
    cin >> src >> dst;
    matrix[src][dst] = true;
    matrix[dst][src] = true;
}

dfs(v);
bfs(v);
```

DFS와 BFS BOJ 1260

```
void dfs(int cur) {
    dfs_visited[cur] = true;
    cout << cur << ' ‘;
    for (auto next : edges[cur]) {
        if (dfs_visited[next])
            continue;
        dfs(next);
    }
}
```

DFS와 BFS BOJ 1260

```
void dfs(int cur) {
    dfs_visited[cur] = true;
    cout << cur << ' ‘;
    for (int next = 1; next <= MAX_NODE; next++) {
        if (dfs_visited[next] || !matrix[cur][next])
            continue;
        dfs(next);
    }
}
```

DFS와 BFS BOJ 1260

```
void bfs(int start) {
    bfs_queue.push(start);
    bfs_visited[start] = true;

    while (!bfs_queue.empty()) {
        int cur = bfs_queue.front();
        bfs_queue.pop();
        cout << cur << ' ';
        for (auto next : edges[cur]) {
            if (!bfs_visited[next]) {
                bfs_queue.push(next);
                bfs_visited[next] = true;
            }
        }
    }
}
```

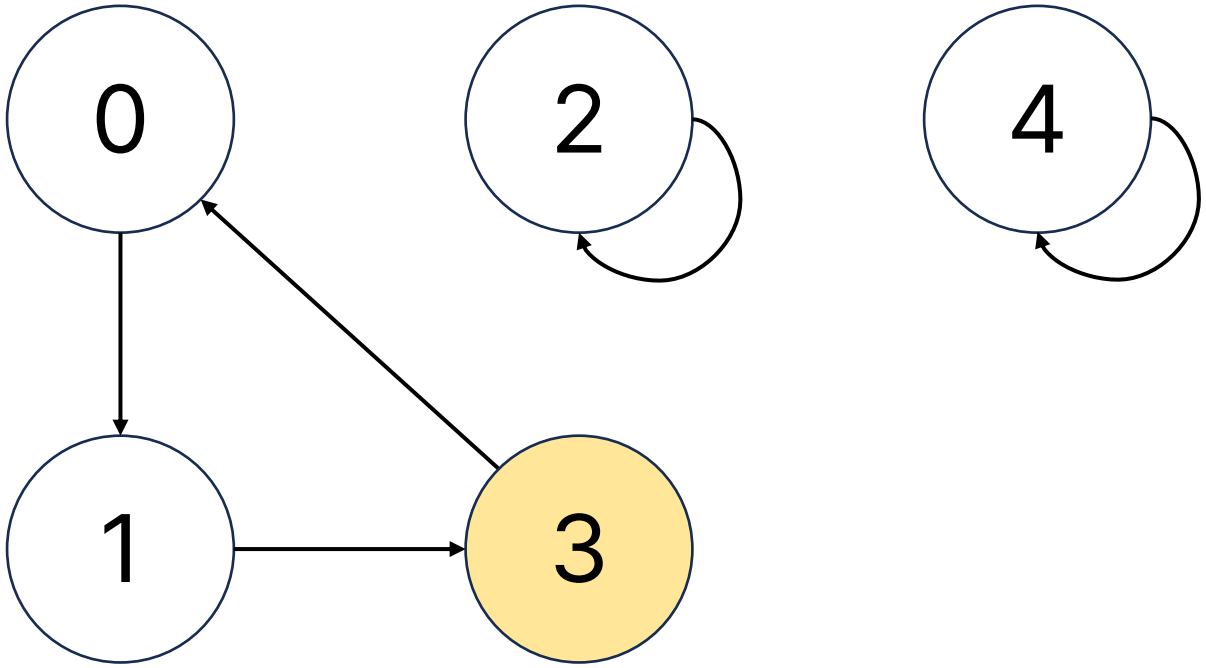
DFS와 BFS BOJ 1260

```
void bfs(int start) {
    bfs_queue.push(start);
    bfs_visited[start] = true;

    while (!bfs_queue.empty()) {
        int cur = bfs_queue.front();
        bfs_queue.pop();
        cout << cur << ' ';
        for (int next = 1; next <= MAX_NODE; next++) {
            if (!bfs_visited[next] and matrix[cur][next]) {
                bfs_queue.push(next);
                bfs_visited[next] = true;
            }
        }
    }
}
```

죽음의 게임 BOJ 17204

- 각 사람은 가르키는 사람이 존재한다
- 0번에서 시작해 1번에 1명씩 가르키는 사람으로 넘어간다
- 주어진 숫자만큼 넘어갔을 때, 보성이에서 끝나야 한다
- 이 때, 가장 작은 숫자를 출력해라

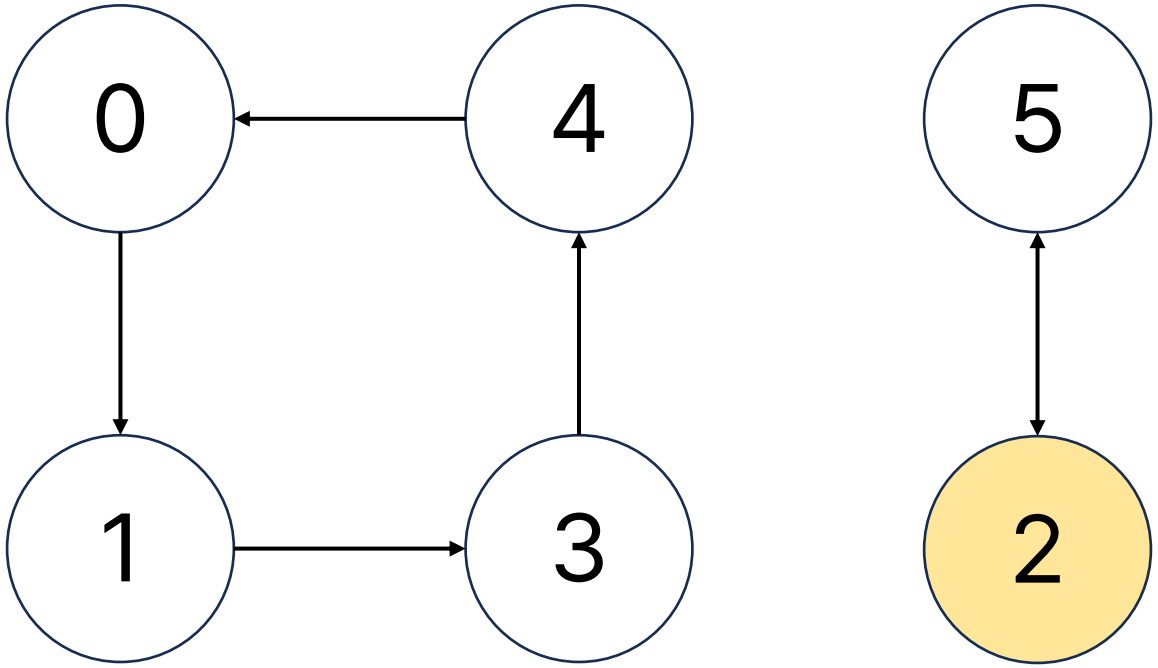
죽음의 게임 BOJ 17204



죽음의 게임 BOJ 17204

- 다음 사람으로 넘어가다가 보성이를 만나는 경우, 그 때가 가장 최소이다
- 가장 처음 보성이를 만났을 때, 몇 번 넘어갔는지 확인

죽음의 게임 BOJ 17204



죽음의 게임 BOJ 17204

- 만일 보성이를 만나기 전에 이미 방문한 사람이 나오는 경우 하나의 사이클(순환)이 만들어지게 된다
- 즉, 이 이후로는 해당 사이클에서 계속 돌고 절대 보성이에게 도달할 수 없게 된다
- 이러한 경우 -1을 출력한다

죽음의 게임 BOJ 17204

- 모든 노드는 출발 간선이 하나이므로 리스트 대신 배열을 사용

```
int n, k;
int arr[150];
bool visited[150];

cin >> n >> k;
for (int i = 0; i < n; i++)
    cin >> arr[i];
```

죽음의 게임 BOJ 17204

```
int dfs(int x, int d) {
    if (x == k)
        return d;
    if (visited[x])
        return -1;
    visited[x] = true;
    return dfs(arr[x], d + 1);
}

cout << dfs(0, 0);
```

섬의 개수 BOJ 4963

- 하나의 섬은 모두 연결되어 있다
- 하나의 연결요소는 하나의 섬이다
- 연결요소의 개수가 곧 섬의 개수다

- 여러 개의 테스트 케이스로 이루어져 있으므로 초기화에 유의하자

섬의 개수 BOJ 4963

- 하나의 섬은 모두 연결되어 있다
- 하나의 연결요소는 하나의 섬이다
- 연결요소의 개수가 곧 섬의 개수다

- 여러 개의 테스트 케이스로 이루어져 있으므로 초기화에 유의하자

섬의 개수 BOJ 4963

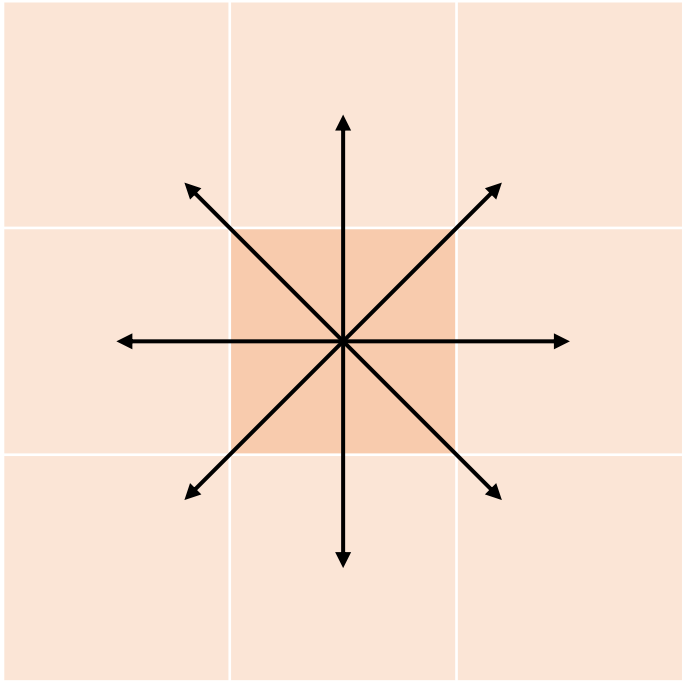
```
int arr[50][50];
bool visited[50][50];
int w, h;

while (true) {
    cin >> w >> h;
    if (w == 0 and h == 0)
        break;
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            cin >> arr[i][j];
            visited[i][j] = false;
        }
    }
// Traversal
}
```

섬의 개수 BOJ 4963

```
while (true) {
    int cnt = 0;
    for (int i = 0; i < h; i++)
        for (int j = 0; j < w; j++)
            if (arr[i][j] and !visited[i][j]) {
                cnt++;
                dfs(i, j);
            }
    cout << cnt << '\n';
}
```

섬의 개수 BOJ 4963

- 가로, 세로, 대각선 방향으로 이어져 있으므로 간선이 존재한다고 생각할 수 있다



섬의 개수 BOJ 4963

```
void dfs(int x, int y) {
visited[x][y] = true;
    if (x + 1 < h and arr[x + 1][y] and !visited[x + 1][y])
        dfs(x + 1, y);
    if (y + 1 < w and arr[x][y + 1] and !visited[x][y + 1])
        dfs(x, y + 1);
// ...
}
```

섬의 개수 BOJ 4963

- 굉장히 복잡해진 코드가 나온다
- 모든 노드가 이동하는 방향은 동일하다
- 좌표 평면에서는 상하좌우 또는 상하좌우대각선으로 이동하는 경우가 많다
- 이를 이용해 dx, dy배열로 반복문으로 순회를 할 수 있다

섬의 개수 BOJ 4963

```
int dx[] = {1, -1, 0, 0, 1, -1, 1, -1};
int dy[] = {0, 0, 1, -1, -1, -1, 1, 1};

void dfs(int x, int y) {
    visited[x][y] = true;

    for (int i = 0; i < 8; i++) {
        if (valid(x + dx[i], y + dy[i]))
            dfs(x + dx[i], y + dy[i]);
    }
}
```

섬의 개수 BOJ 4963

```
bool valid(int x, int y) {
    if (x < 0 or h <= x)
        return false;
    if (y < 0 or w <= y)
        return false;
    return arr[x][y] and !visited[x][y];
}
```

안전 영역 BOJ 2468

- 섬의 개수와 매우 유사한 문제
- 연결 요소의 개수 중 최대를 출력하는 문제
- 섬의 개수에서는 땅이 명확하게 주어졌지만 안전 영역에서는 물의 높이에 따라 땅의 여부가 정해진다
- 물에 높이를 바꿔가면서 연결 요소의 개수를 센다
- 그 중 최대값을 출력한다

안전 영역 BOJ 2468

- 물에 잠기지 않은 경우는 현재 물의 높이보다 땅이 높은 경우이다
- 땅의 높이 <= 물의 높이인 경우 땅이 존재하지 않는다
- valid함수를 수정해서 이동 가능 여부를 수정해준다

안전 영역 BOJ 2468

- 땅은 상하좌우로 붙어있을 때 하나로 취급하므로, dx, dy 배열도 수정해야한다

안전 영역 BOJ 2468

- 여러번 순회를 해야하므로 꾸준히 visited 배열을 초기화 해줘야한다
- 꾸준히 초기화 하는 방법 외에도 각 순회마다 번호를 부여할 수 있다
- visited배열을 int로 선언하고 각 순회마다 특정한 번호를 이용해 초기화 과정을 대체 할 수 있다

안전 영역 BOJ 2468

```
int n, water_h;
int arr[100][100];
int visited[100][100];
int dx[] = {1, -1, 0, 0};
int dy[] = {0, 0, 1, -1};

cin >> n;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        cin >> arr[i][j];
```

안전 영역 BOJ 2468

```
int ans = 1; // 모든 땅이 한 덩어리인 경우
    while (++water_h <= 100) { // 물의 높이를 점차 증가 시킴
        int cnt = 0;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (valid(i, j)) {
                    cnt++;
                    dfs(i, j);
                }
        ans = max(cnt, ans);
    }
```

안전 영역 BOJ 2468

```
void dfs(int x, int y) {
    visited[x][y] = water_h;
    for (int i = 0; i < 4; i++) {
        if (!valid(x + dx[i], y + dy[i]))
            continue;
        dfs(x + dx[i], y + dy[i]);
    }
}
```

안전 영역 BOJ 2468

```
bool valid(int x, int y) {
    if (x < 0 or n <= x)
        return false;
    if (y < 0 or n <= y)
        return false;
    return arr[x][y] > water_h and visited[x][y] < water_h;
}
```

DFS

- DFS는 왜 재귀 함수 형태로 구현이 가능한가?
- DFS를 다시 살펴보자

DFS

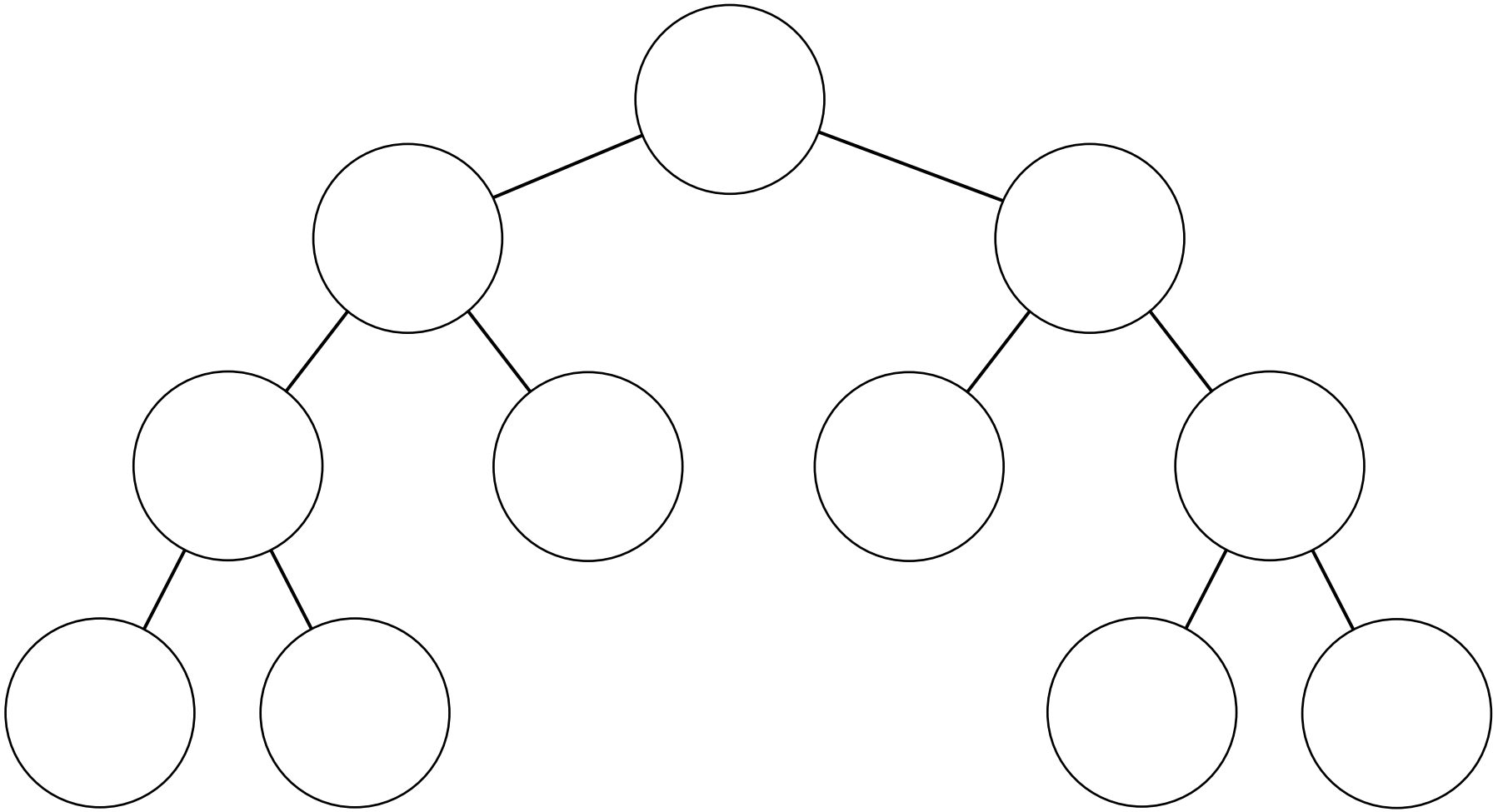
- 현재 노드에서 이동 가능한 노드가 존재하면 해당 노드로 이동한다
- 더 이상 방문 가능한 노드가 없으면 이전 노드로 돌아간다

- 즉, 본인을 제외하고 **마지막** 노드로 돌아간다
- 노드를 데이터로 생각한다면 마지막에 넣은 데이터를 꺼내는 LIFO 구조이다
- 스택을 사용하여 구현한다

DFS

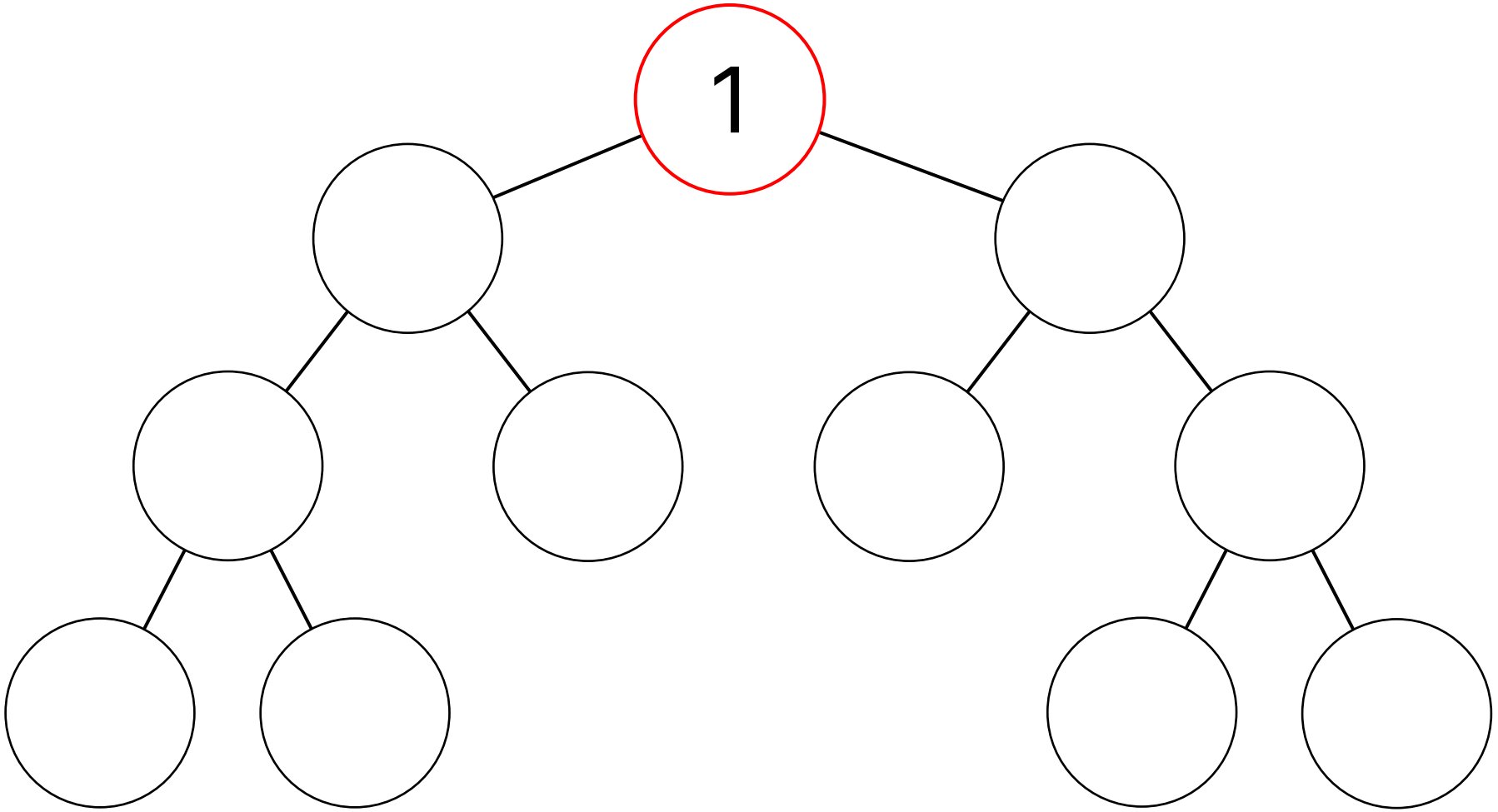
- 반복문과 Stack을 이용해 DFS를 구현할 수 있다

DFS



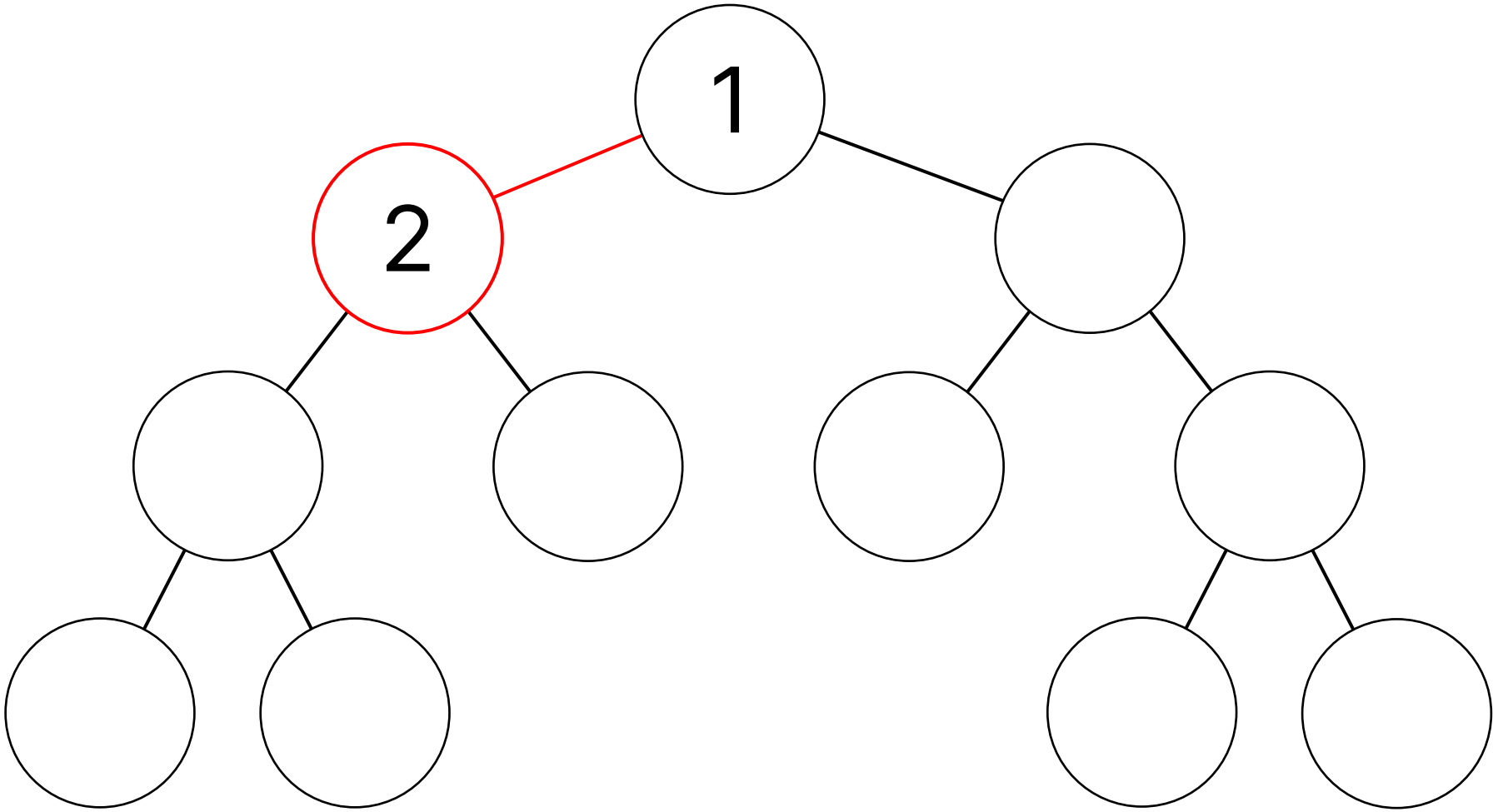
DFS

1				



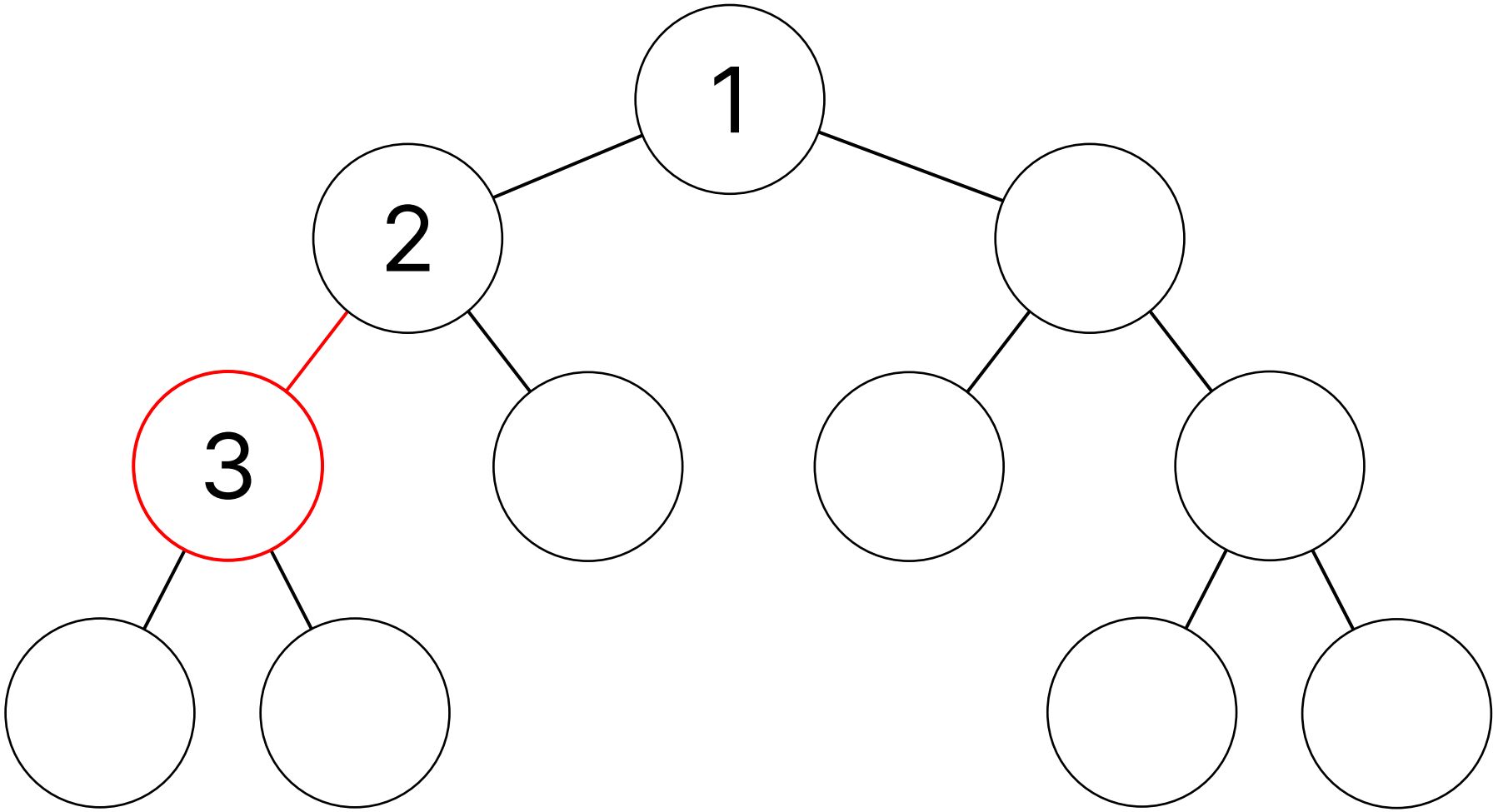
DFS

1	2			



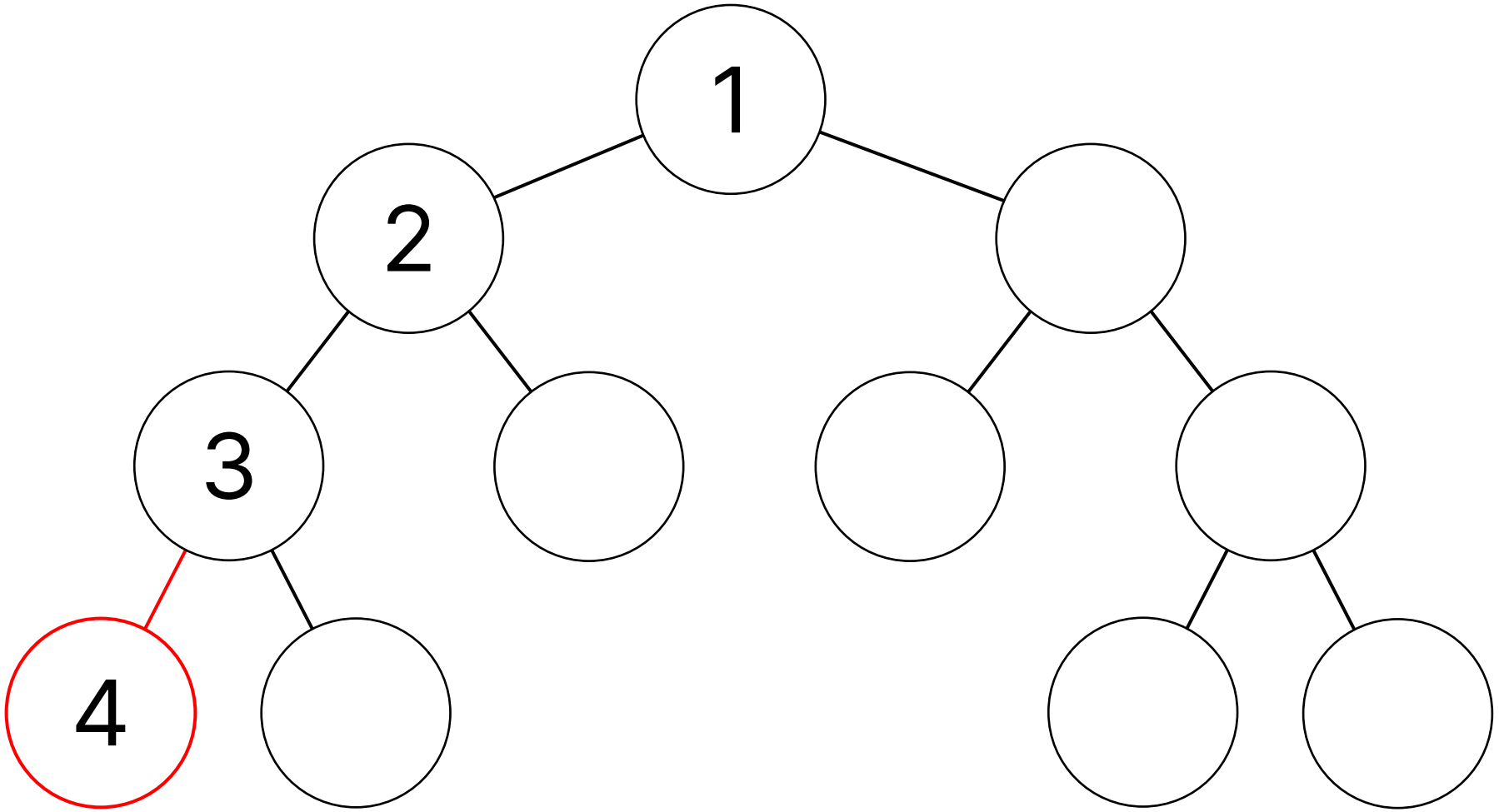
DFS

1	2	3		



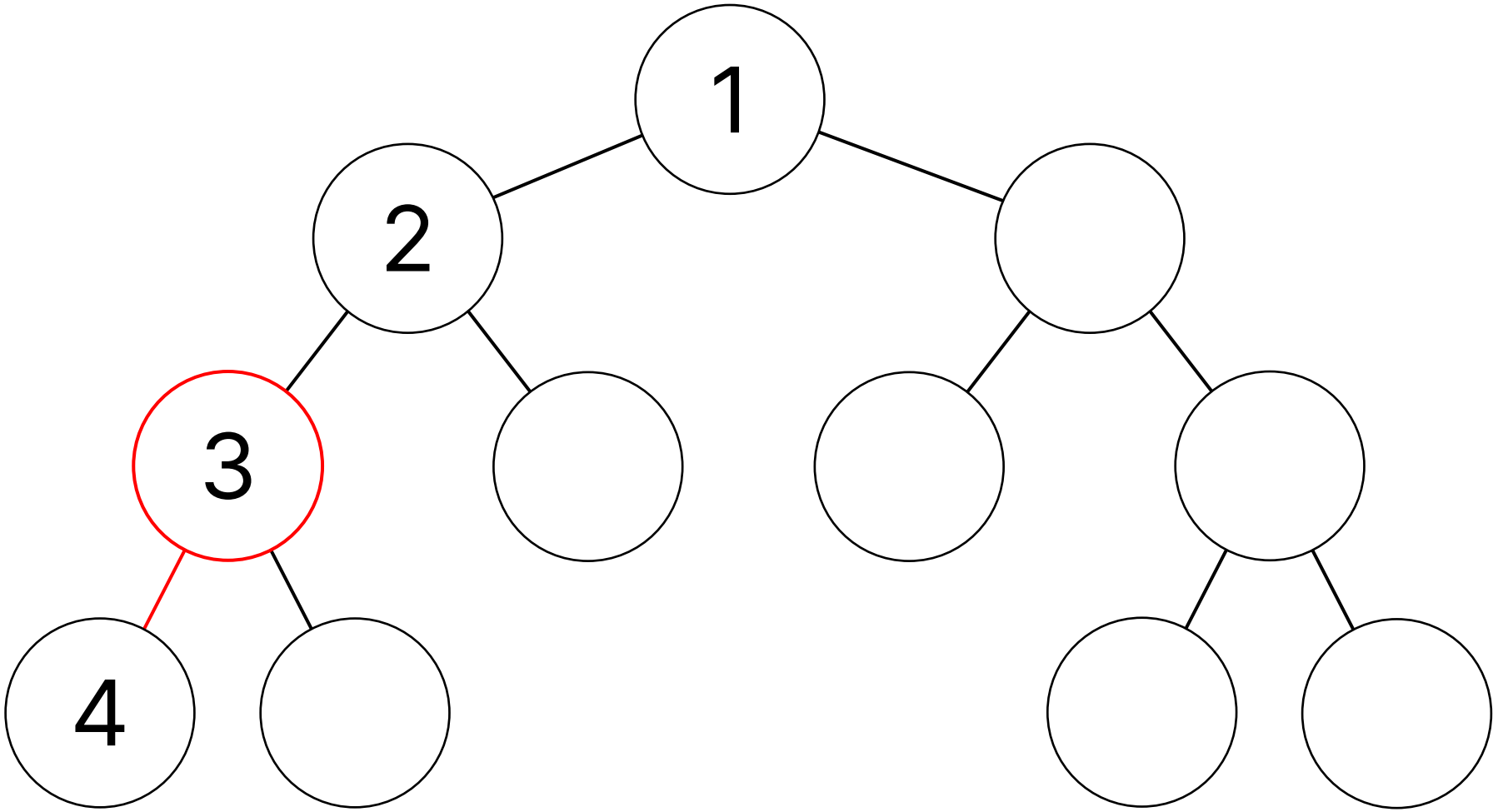
DFS

1	2	3	4	



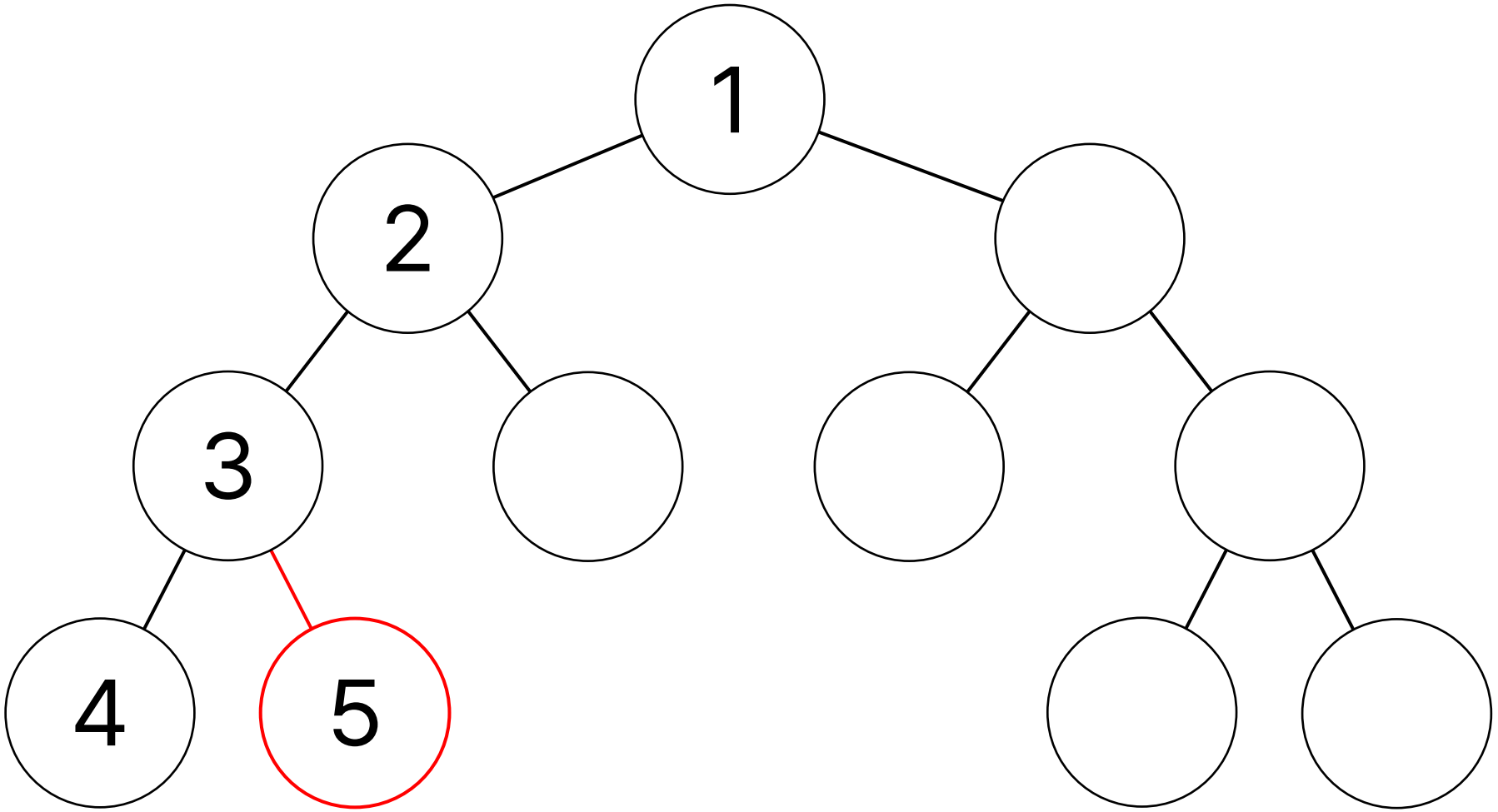
DFS

1	2	3	4	



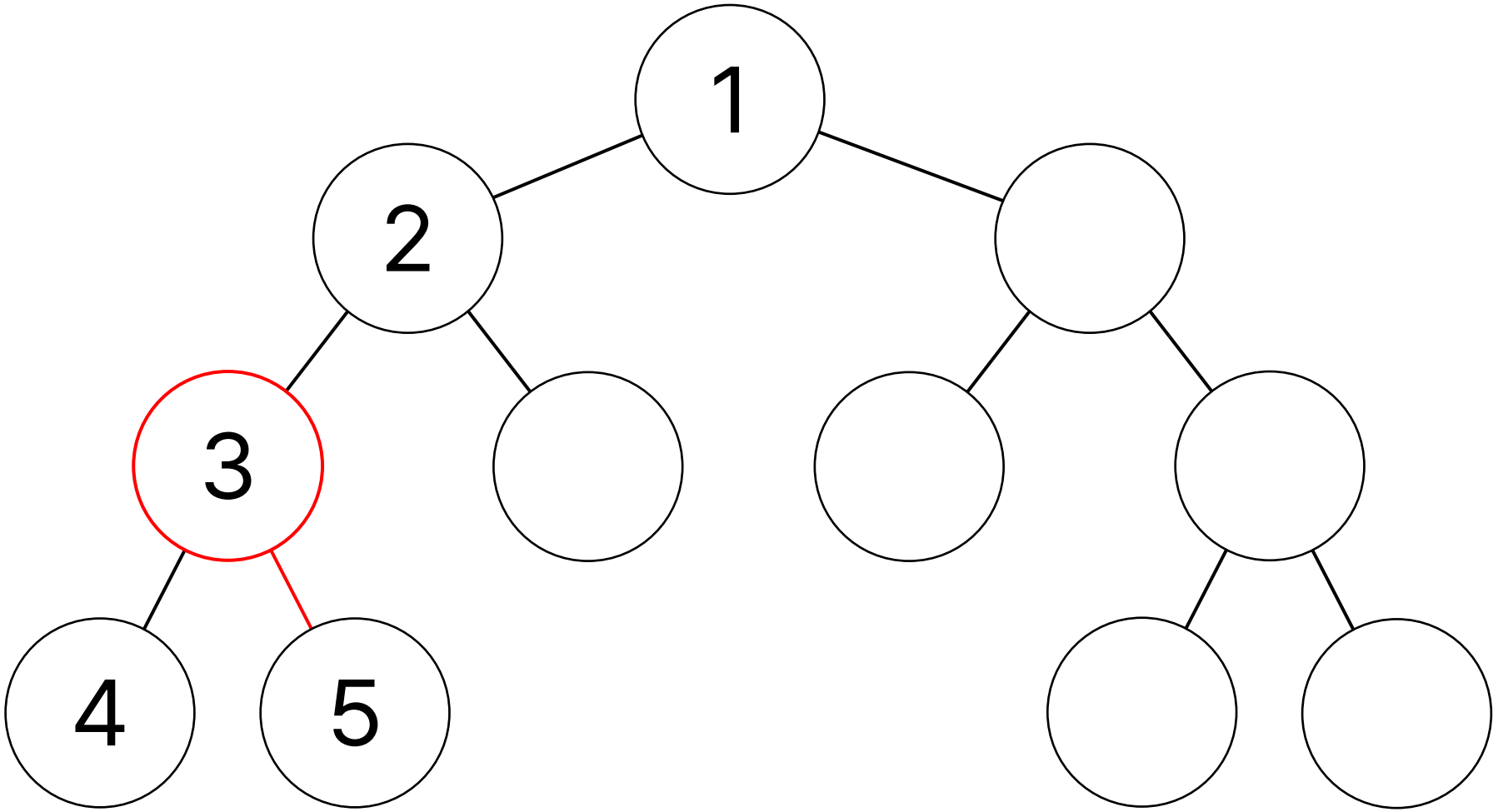
DFS

1	2	3	5	



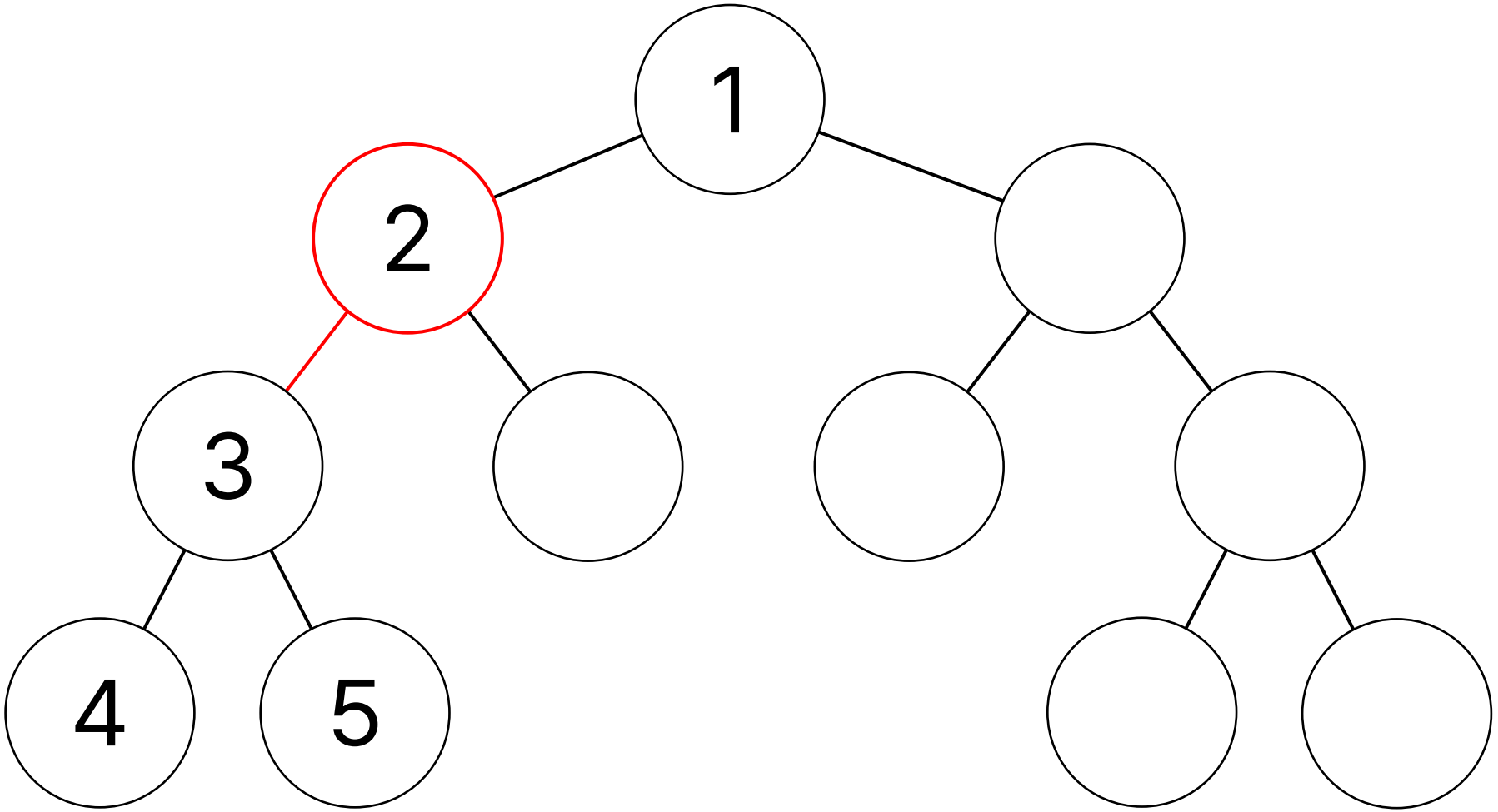
DFS

1	2	3	5	



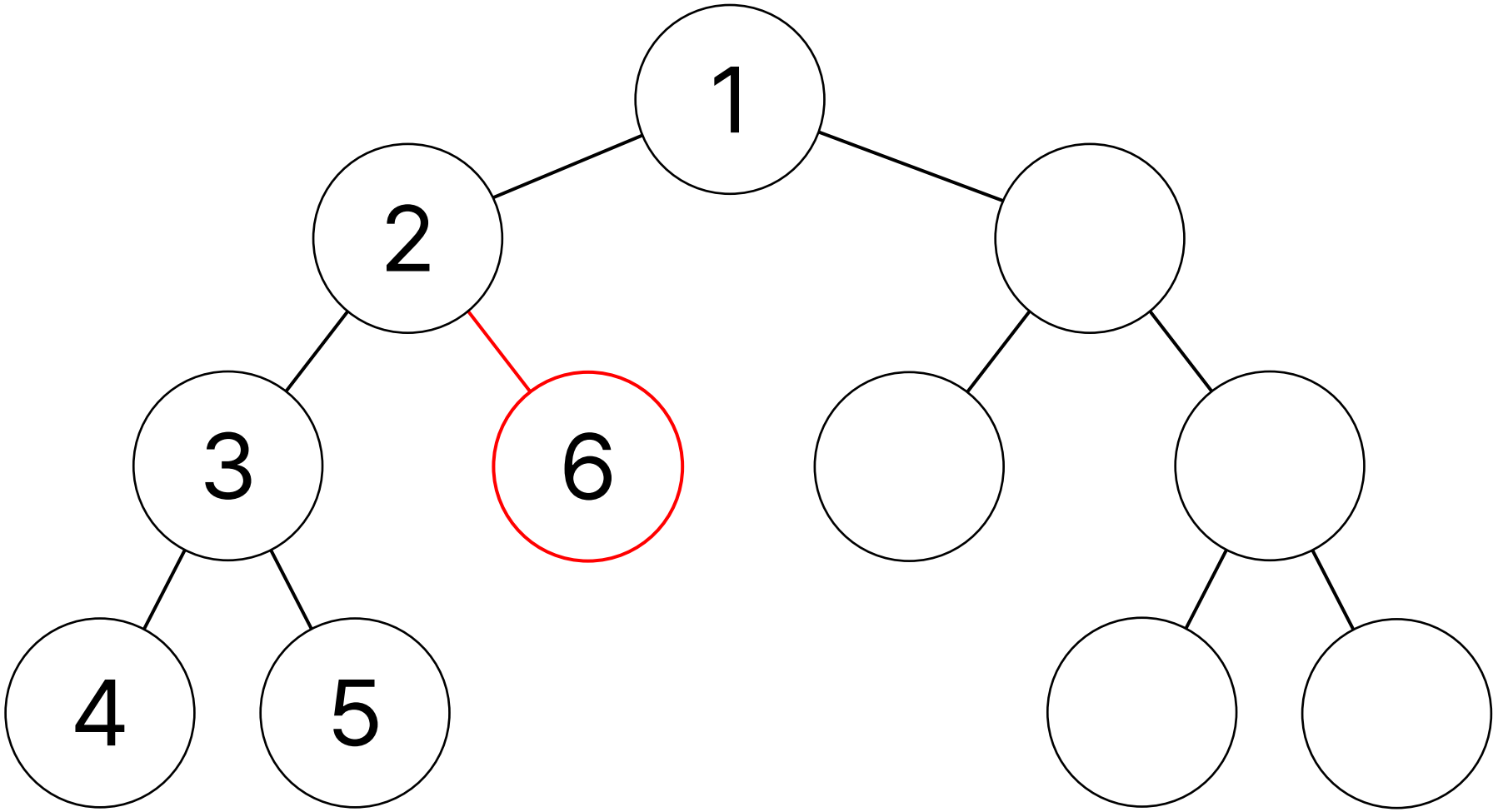
DFS

1	2	3		



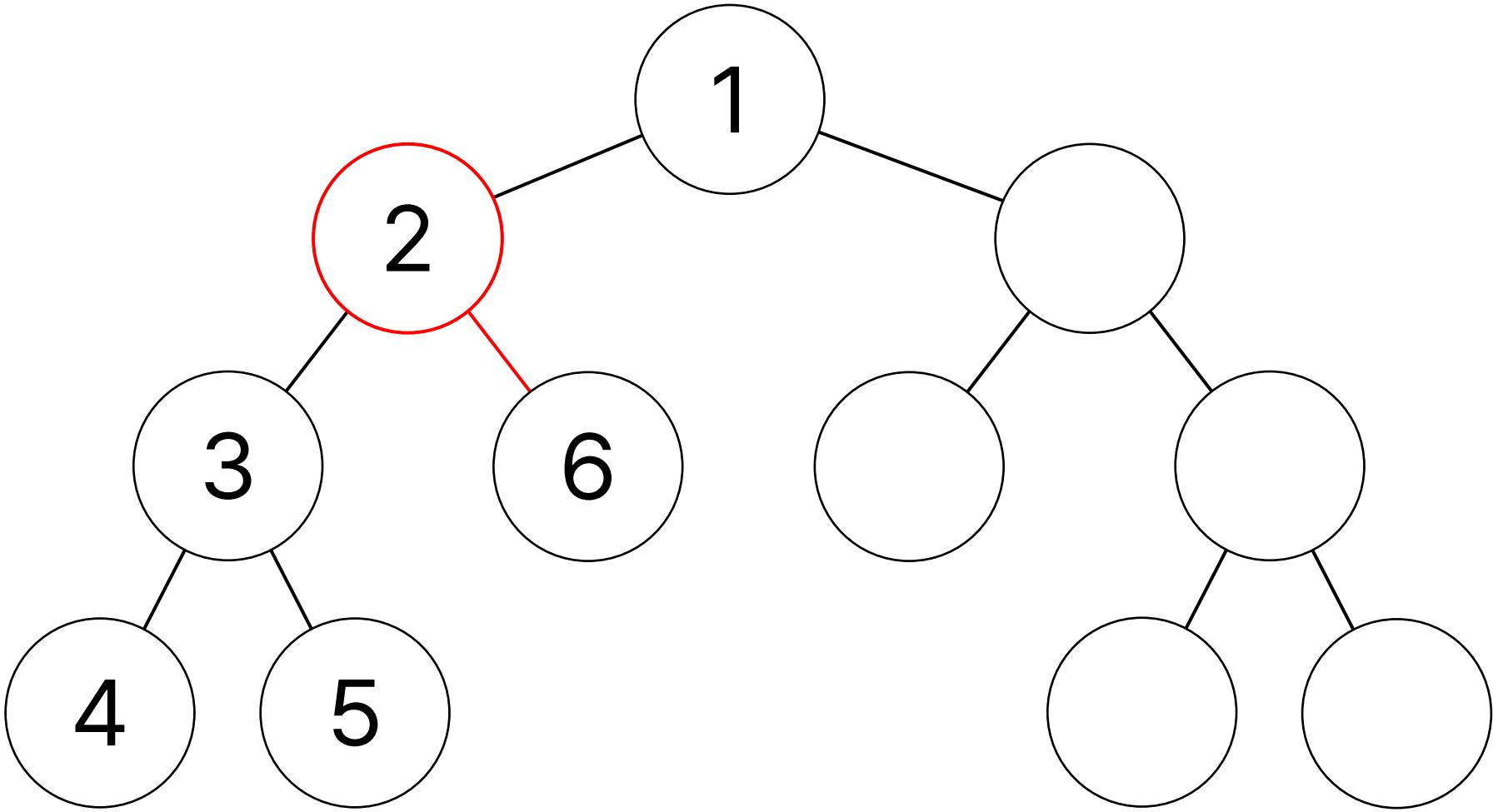
DFS

1	2	6		



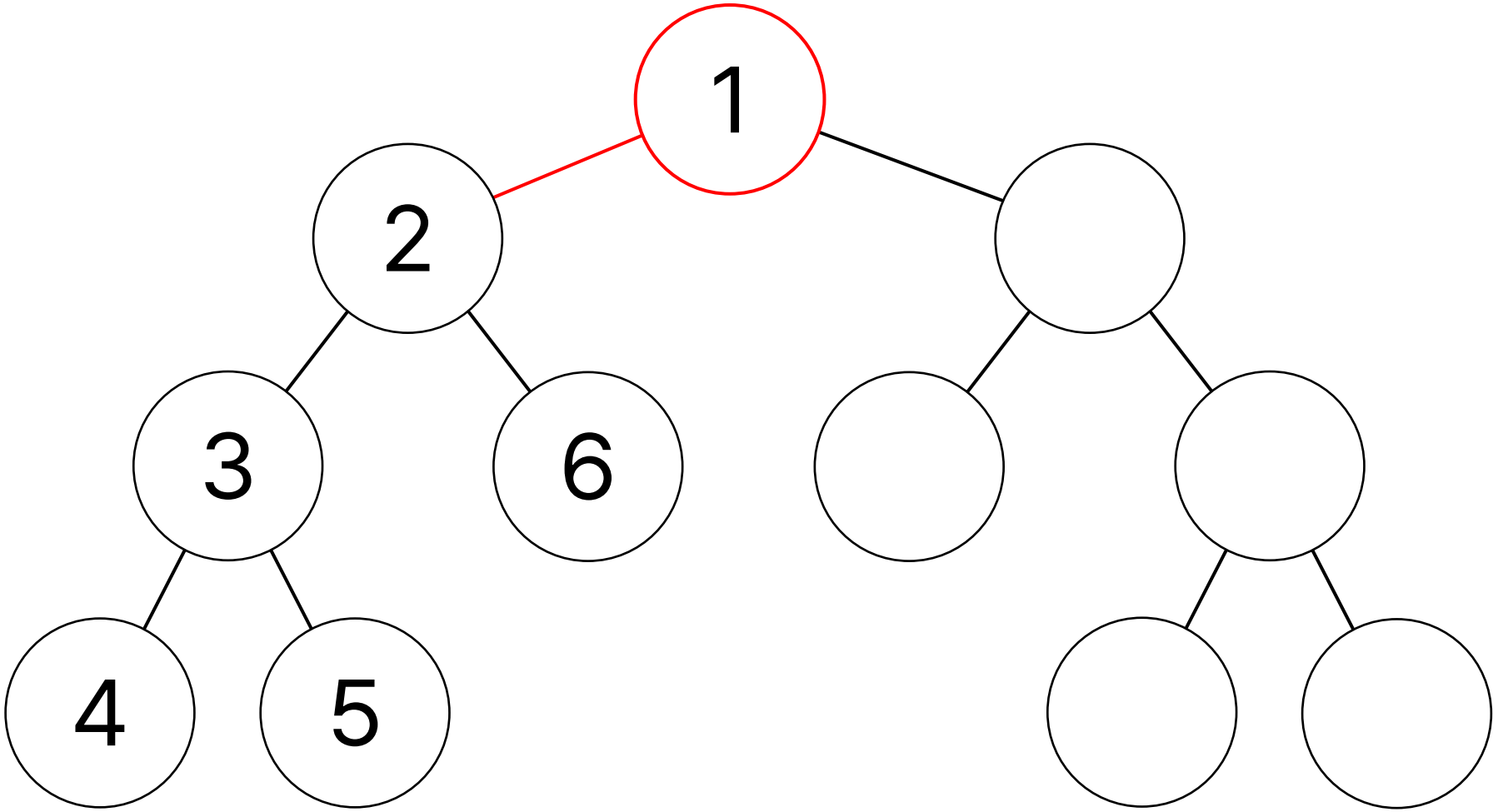
DFS

1	2	6		



DFS

1	2			



DFS

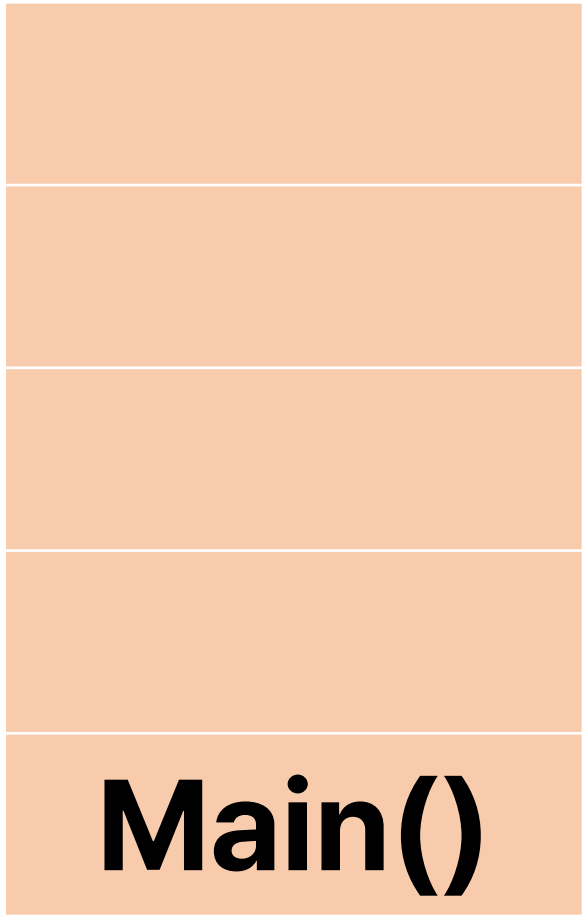
```
void dfs(int start) {
    st.push(start);
    while (!st.empty()) {
    check:
        visited[st.top()] = true; // 방문한 경우 방문 여부 기록
        for (auto next : edges[st.top()]) {
            if (!visited[next]) {
                st.push(next);
                goto check;
            }
        }
        st.pop();
    }
}
```

DFS

- 함수를 호출하는 것 또한 스택의 과정이다
- 함수를 호출하면 호출한 함수로 넘어간다
- 호출한 함수가 종료되면 마지막으로 실행되던 함수, 본인을 호출한 함수로 돌아간다

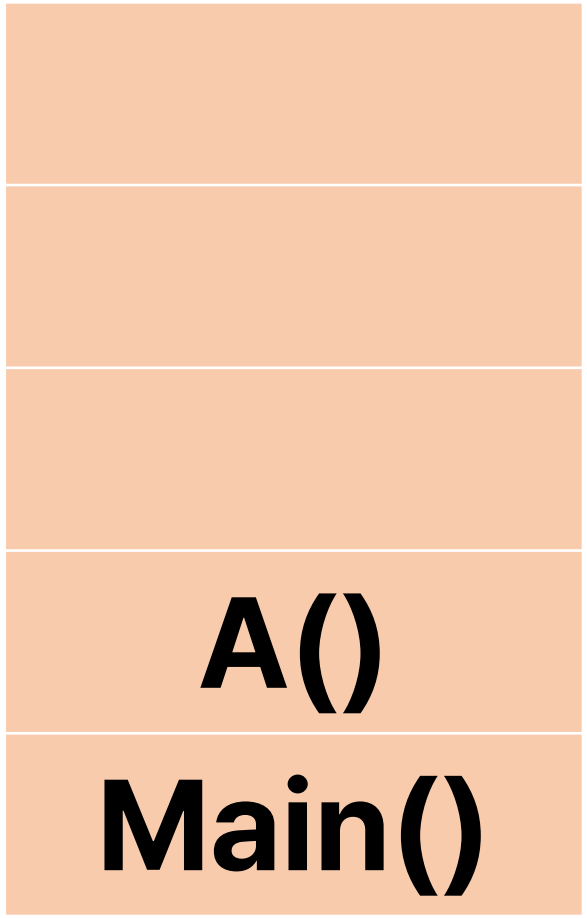
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



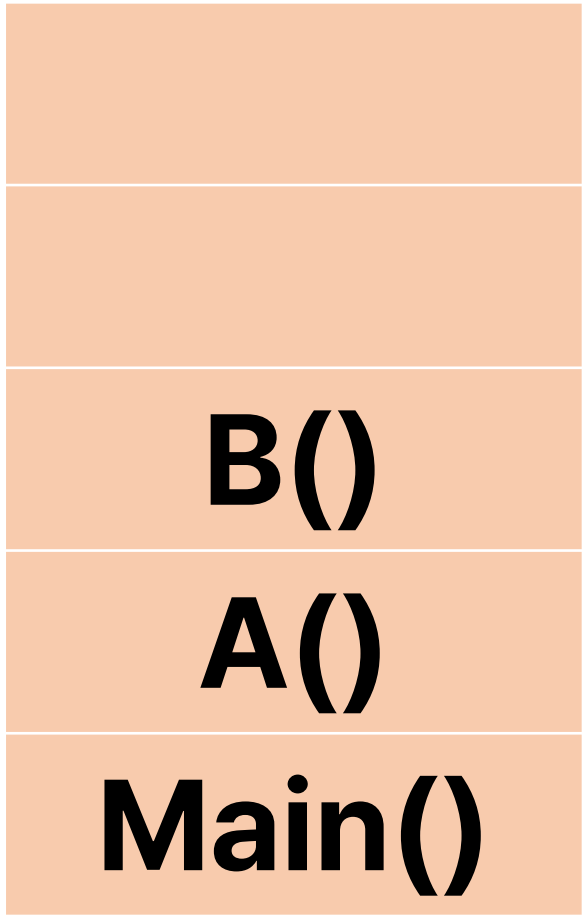
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



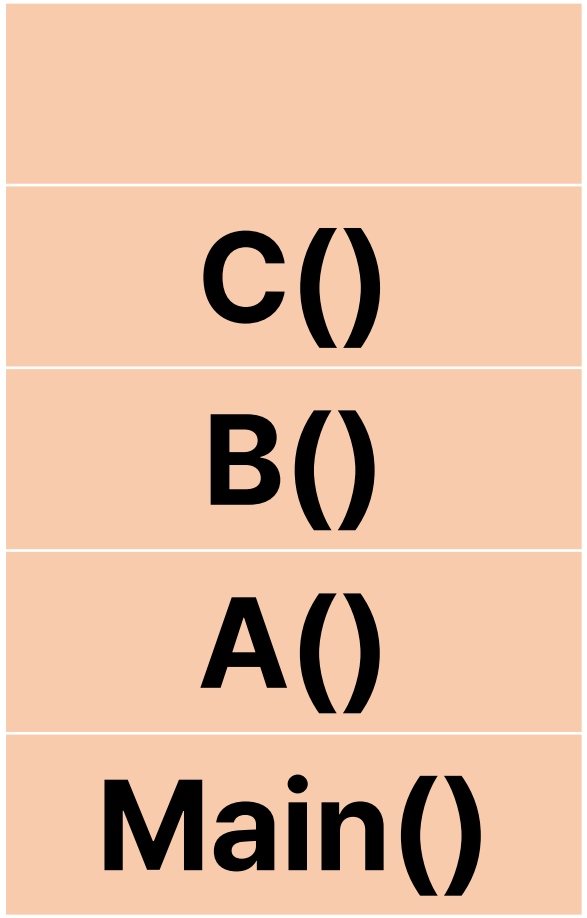
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



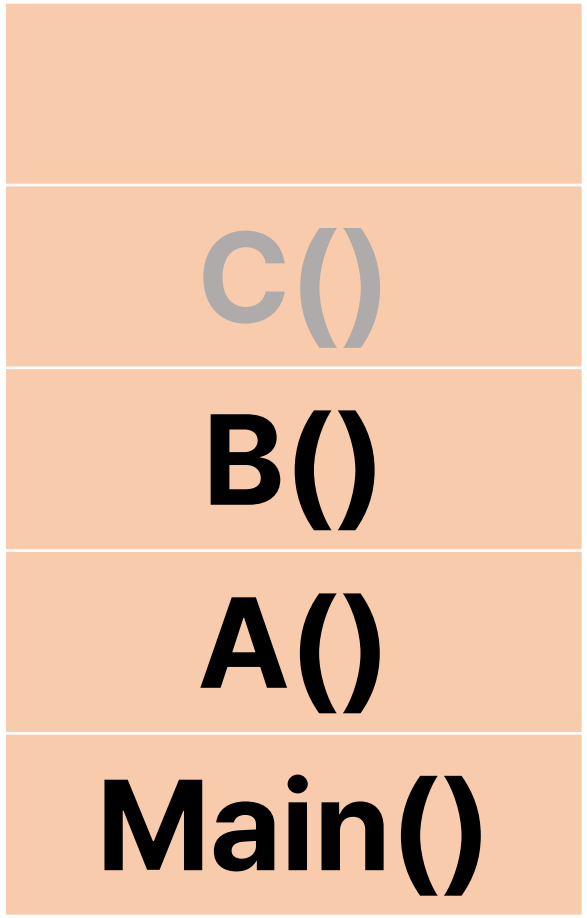
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



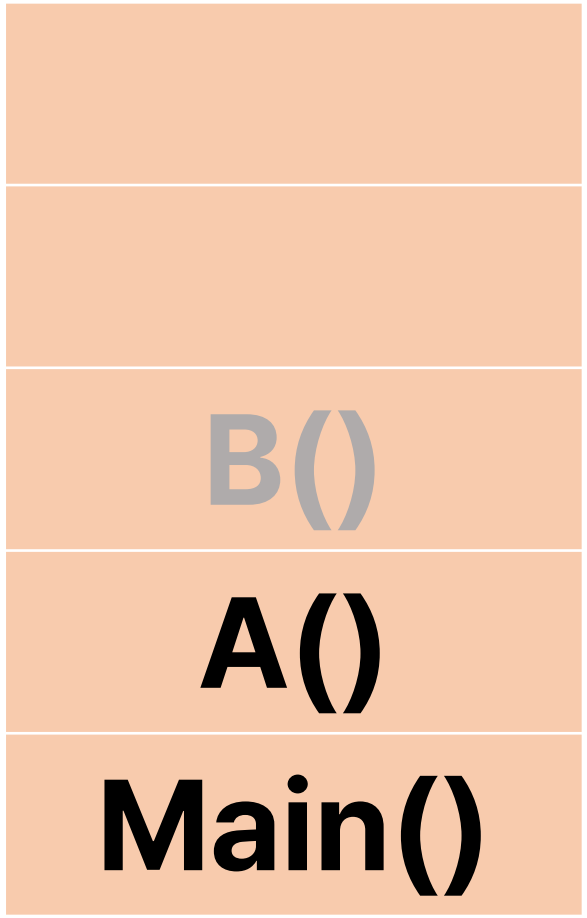
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



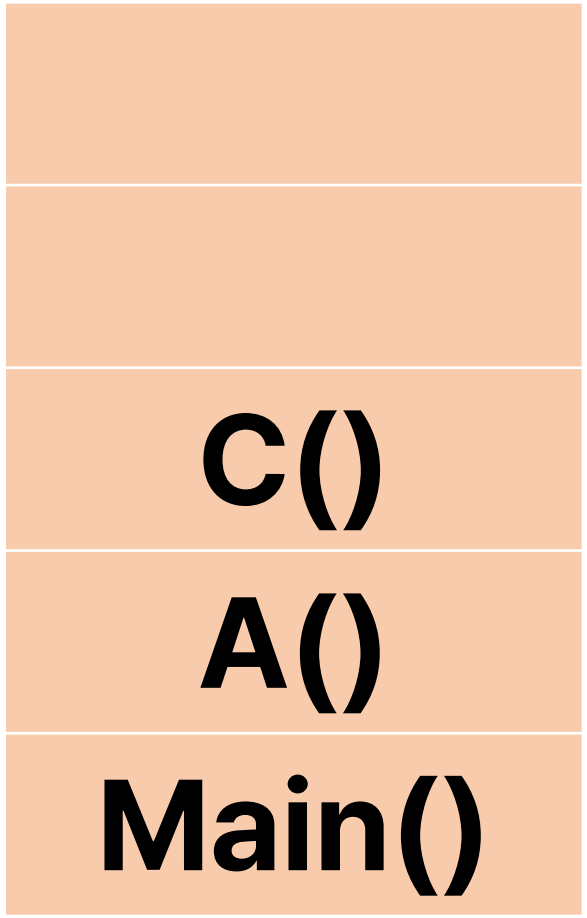
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



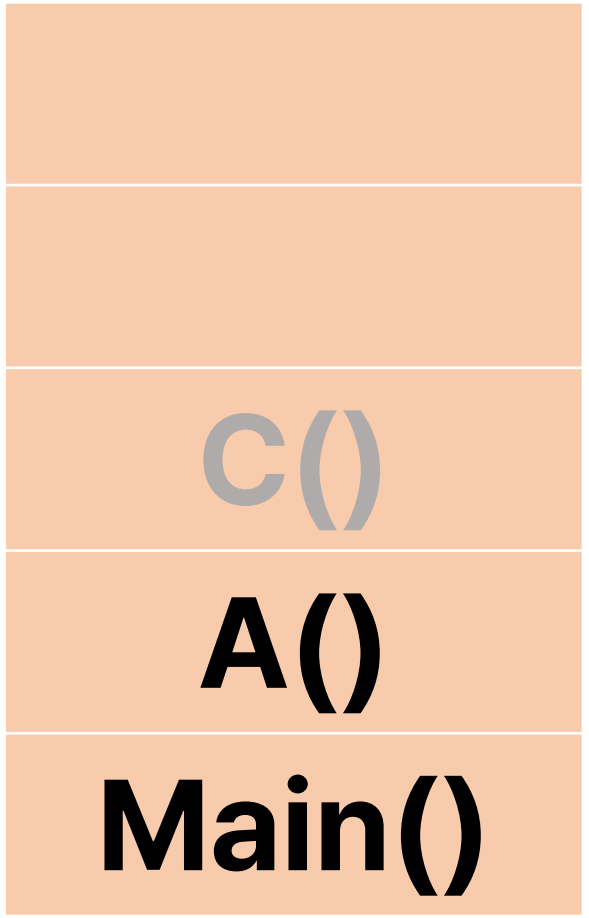
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



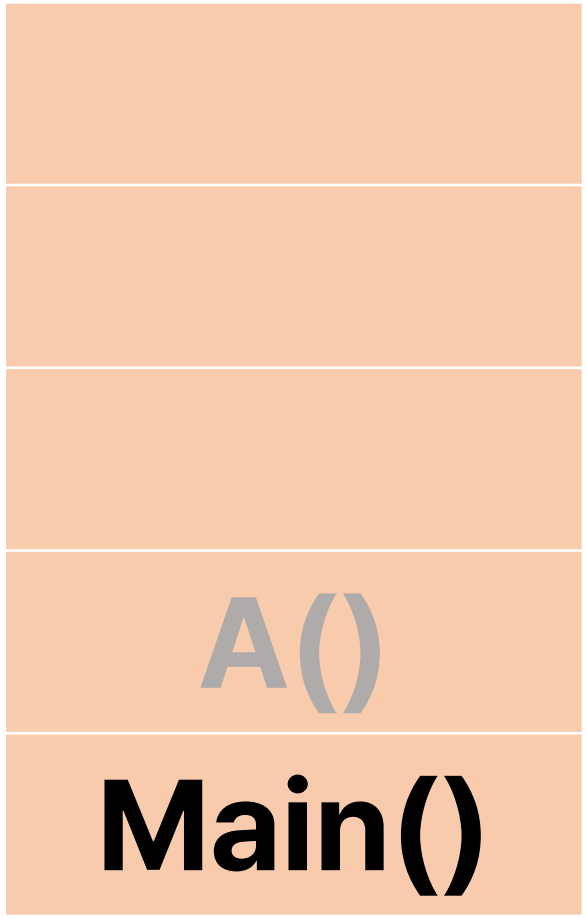
DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



DFS

```
void C() { return; }

void B() { C(); }

void A() {
    B();
    C();
}

int main(){
    A();
    return 0;
}
```



DFS

- DFS에서 스택을 사용할 필요 없이 DFS를 함수로 만드는 것은 그 자체로 스택을 이용하는 것이다
- 스택과 반복문을 사용하는 것과 함수로 구현하는 방법 두 가지 모두 가능하지만 함수의 구현이 훨씬 간단하다

문제

- 알고리즘 수업 - 깊이/너비 우선 탐색
- 숨바꼭질 BOJ 1697
- 토마토 BOJ 7576
- 토마토 BOJ 7569
- 달이 차오른다, 가자. BOJ 1194