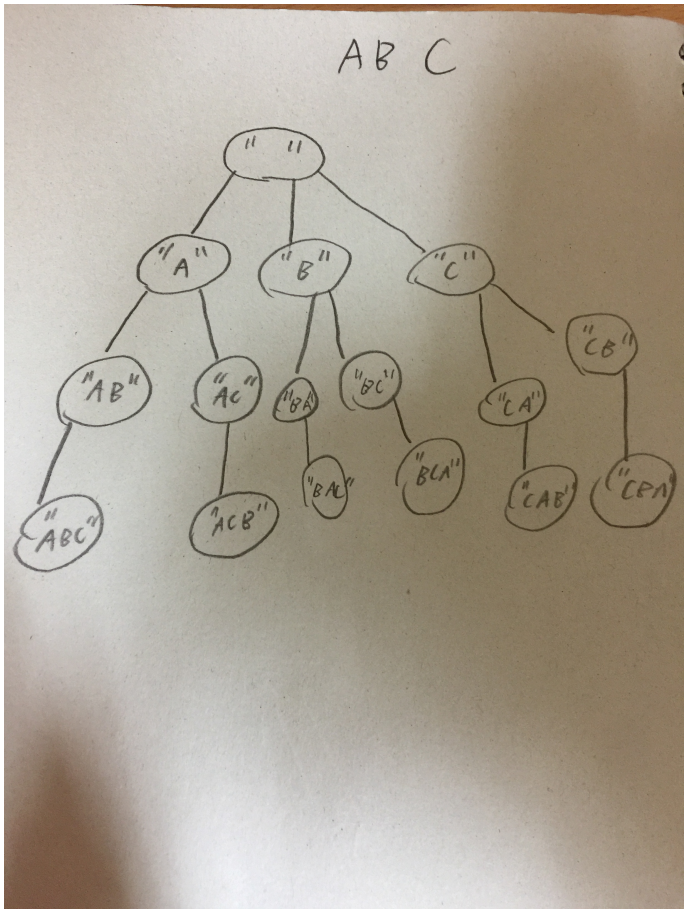
DFS/BFS

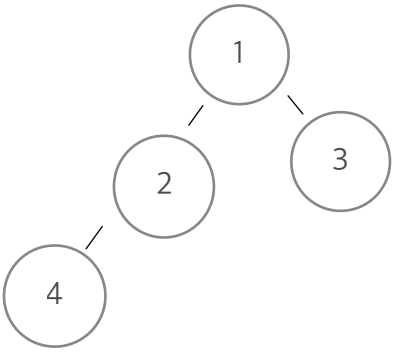
이번에 배울 내용은 DFS/BFS 입니다. DFS와 BFS는 보통 트리에 적용되는 탐색입니다. 물론 트리에 적용하는것을 배우고 나서 일반 그래프에서의 탐색 역시 볼것입니다.

DFS는 Depth First Search 의 줄임말로서 연결되어있는 자식 노드중 한개를 먼저 탐색하고 그 자식 노드에 달려있는 자식노드를 타고.. 이런식으로 리프 노드를 만날때 까지 탐색을 이어나가다가 리프 노드를 만나면 부모 노드로 퇴각한 뒤, 그 부모노드에 달린 자식노드를 다시 찾고.. 이것을 반복하는 것이 DFS 입니다. 백트래킹과 유사한 점이 많지요? 네, 백트래킹이 이러한 DFS의 원리를 차용한다고 생각하시면 편합니다. 다음 그림을 보시죠.



네, 저번 시간에 배운 백트래킹 문제 중, ABC의 순열을 구하는 문제입니다. 빈 문자열에서 시작하여 A를 고르고, A를 고른 **"상태"**에서 AB를 고르고, AB를 고른 **"상태"**에서 ABC를 고르고 더 고를 수 없으니 퇴각하고, 역시 더 고를 수 없으니 퇴각하고, 그 이제 AC를 고르고.. DFS가 이러한 백트래킹과 완전히 동일한 원리로 작동한다고 생각하시면 됩니다. 저번에 상태를 나타낼 때 Vertex를 사용한다고 했었죠? 이 경우가 그러한 경우입니다.

그렇다면 일반적인 경우에서의 DFS는 어떻게 구현할까요?



이 트리를 봐주세요. 1에 연결되어 있는 2를 제일 먼저 방문해 봅니다. 그 다음 2에 연결되어있는 4를 방문하여 봅니다. 그 다음 방문할 노드가 더 이상 없으니 퇴각합니다. 마찬가지로 2에도 없으니 퇴각합니다. 그 다음 3을 방문합니다. 이러한 과정을 백트래킹 처럼 재귀함수로 구현하면 편합니다.

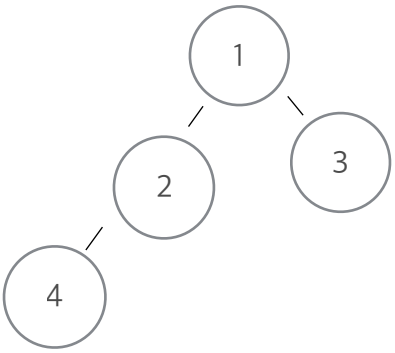
이러한 일련의 과정을 psedo code로 나타내면 다음과 같습니다.

```
dfs (u) :
    print u
    u is visited
    for v in AdjanceyList[u]
        if is v visited?:
            continue
        dfs(v)
```

앞으로 많이 쓰이게 될 코드이니 기억해 둡시다. 여기서 v를 방문했는지 체크하는 것이 있는데 이 체크하는 것이 매우 중요합니다. 한번 방문한 노드를 다시 방문하지 않는것이 DFS에서 매우 중요합니다. 만약 저 처리를 해주지 않는다면 무방향 그래프에서의 트리는 영원히 순회하게 됩니다. 물론 트리에서는 전에 방문했던 노드를 파라미터로 넘겨주는 방법으로 막을 순 있긴 합니다만, 후술할 일반적인 그래프 에서의 DFS에 적용하지 못한다는 치명적인 단점이 있습니다.

일반적인 그래프 에서도 DFS를 적용할 수 있습니다. 저 위의 psedo code 를 그대로 사용하면 됩니다. 그러면 모든 노드를 조사하게 되며 이러한 과정에서 추출되는 트리를 DFS Tree라고 합니다. 이러한 개념은 고급 알고리즘을 배울때 사용되므로 지금은 이런게 있다라는 정도로만 알아두시면 되겠습니다. 자 그러면, 이러한 탐색을 통해 알 수 있는 사실은 **어떤 임의의 노드 u에 연결된 모든 v를 알아낼 수 있다**는 것 입니다. 이러한 특성을 이용한 알고리즘 중에는 Flood-Fill 이라는 알고리즘이 있습니다. 이것은 나중에 단지번호 붙이기 문제를 풀면서 같이 알아봅시다.

그 다음은 BFS 입니다. BFS는 Breadth First Search의 줄임말로서 레벨 0 노드 즉, 루트노드를 먼저 탐색하고 루트에 연결된 노드 레벨 1 노드를 탐색하고, 그 다음 레벨 2 노드를 탐색하고.. 이러한 식으로 너비를 먼저 본다 해서 이러한 이름이 붙은것 같습니다. 이 BFS 탐색의 특징은 Root 로 부터 가장 가까운 노드를 먼저 본다는 점 입니다. 아까 트리를 다시한번 봐봅시다.



이 트리에서는 1을 먼저 탐색하고 그 다음 2, 3 을 탐색하고 그 다음 4를 탐색하게 됩니다. 이러한 일련의 과정을 수월하게 진행하기 위한 자료구조는 뭐가 있을까요? 바로 Queue 입니다. 1에 연결된 2, 3 을 Queue 에 넣고 그 다음 2에 연결된 4를 Q에 넣고.. 이러한 방식으로 말입니다.

이러한 일련의 과정을 psedo code로 나타내면 다음과 같습니다.

```
bfs (u) :
    Q <- Queue
    Q.push (u)
    while Queue is not empty:
        u <- Q.pop
        print u
        for v in AdjanceyList[u]
            if is v visited?:
                continue
            v is visited
            Q.push(v)
```

역시 자주 쓰이게 될 코드이니 기억해 둡시다. DFS와 상당히 유사한 구조를 띄고 있으나 while문으로 구현하며 Queue 자료구조를 써야함이 다릅니다. 또한 visit 처리는 DFS는 for문 위에서 하고 있지만 BFS는 for문 안쪽에서 해줍니다. 이것을 왜 이렇게 해야하는지는 한번 고민해 보시길 바랍니다. 마찬가지로 BFS역시 DFS 처럼 일반 그래프에도 적용할 수 있습니다.

일반 그래프에 적용하면 어떤 이점을 취할 수 있냐하면 **현재 위치, 혹은 상태에서 가장 가까운 지점을 순서대로 알아 낼 수 있다는 것 입니다.** 이것은 미로 찾기 등을 할때 매우 유용합니다. 현재 위치에서 가장 가까운 지점을 늘려나가다 보면 출구를 최단 경로로 찾아낼 수 있기 때문입니다. 가중치가 있는 경우에는 통하지 않는데 이것은 나중에 배울 Dijkstar Algorithm 에서 고민해 봅시다. 지금은 가중치가 전부 1이라고 가정할 수 있는 상황만 생각하시면 됩니다.

이렇게 간단한 DFS/BFS 개념을 배웠습니다. 이제 이것을 응용하여 문제를 풀어봅시다.

- https://www.acmicpc.net/problem/1260
- https://www.acmicpc.net/problem/2606
- https://www.acmicpc.net/problem/2667
- https://www.acmicpc.net/problem/7576