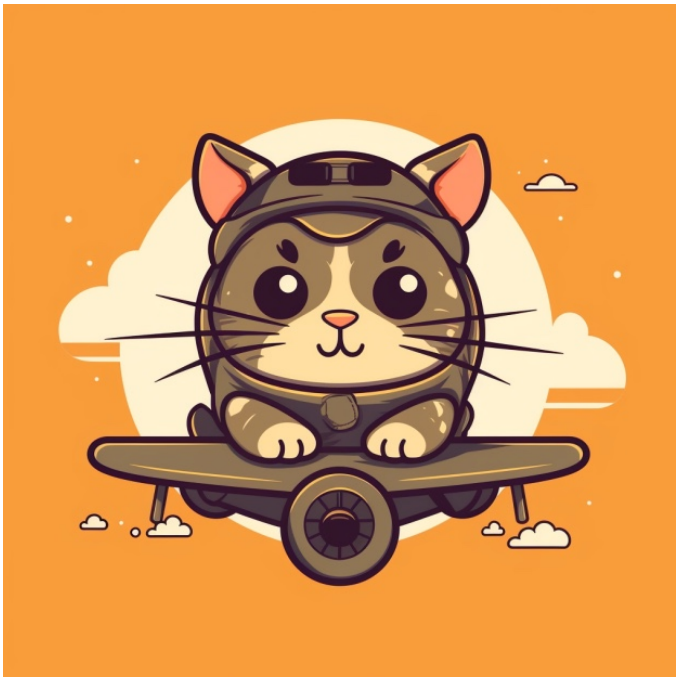
🥷

# [취업폭격기 Zeromini 위클리 개념폭격 #32]

## 📕 과목 : 자료구조론

## 🔥 참고문제 : 2021년 서울시 7급

## 🌝 문제 수정 버전 : V 1.0 (자료구조내용만 포함)



1. **위상 정렬(Topological Sorting)**
    - 문제: 위상 정렬의 개념을 설명하고, 어떤 상황에서 사용되는지 예를 들어 설명하세요.
    - 해설: 위상 정렬은 방향 그래프에서 모든 노드(정점)를 방향성에 반하지 않도록 순서대로 나열하는 방법입니다. 이 정렬 방식은 순환(cycle)이 없는 방향 그래프(DAG, Directed Acyclic Graph)에만 적용됩니다. 예를 들어, 여러 작업이 서로 의존성을 가지고 있을 때, 어떤 순서로 작업을 진행해야 하는지 결정하는 데 사용됩니다. 컴파일러가 소스 코드를 컴파일할 때 의존성 있는 모듈의 컴파일 순서를 결정하거나, 프

로젝트 관리에서 작업의 순서를 결정하는 데에도 활용됩니다. 위상 정렬은 여러 가능한 순서 중 하나를 제공하며, 이는 그래프의 구조에 따라 다를 수 있습니다. 이 정렬 방식은 DFS(깊이 우선 탐색)나 큐를 사용하여 구현할 수 있으며, 각 노드의 '진입 차수'를 계산하여 순서를 결정합니다.

2. **이중 연결 리스트(Doubly Linked List)**
    - 문제: 이중 연결 리스트의 구조와 특징을 설명하고, 삽입 연산의 과정을 간략히 설명하세요.
    - 해설: 이중 연결 리스트는 각 노드가 앞뒤 노드를 가리키는 두 개의 포인터(prev, next)를 가지고 있는 자료구조입니다. 이 구조는 단일 연결 리스트에 비해 노드의 삽입과 삭제가 더 유연하며, 양방향으로 탐색이 가능합니다. 삽입 연산 시, 새 노드의 prev 포인터는 이전 노드를, next 포인터는 다음 노드를 가리키도록 설정합니다. 이후 이전 노드의 next 포인터와 다음 노드의 prev 포인터를 새 노드로 업데이트하여 연결합니다. 이중 연결 리스트는 탐색이 빈번하거나 중간에 자주 삽입, 삭제가 일어나는 경우에 유용하게 사용됩니다.
3. **회문(Palindrome)**
    - 문제: 회문을 판별하는 재귀 함수의 작동 원리를 설명하세요.
    - 해설: 회문 판별 재귀 함수는 문자열의 양 끝 문자를 비교하여 같으면 양 끝을 제외한 부분 문자열에 대해 같은 함수를 재귀적으로 호출합니다. 이 과정은 문자열의 길이가 0이나 1이 될 때까지 반복됩니다. 길이가 0이나 1이면 회문으로 판단하고, 중간에 양 끝 문자가 다르면 회문이 아님을 반환합니다. 이 방법은 문자열의 길이에 따라 시간 복잡도가 O(n/2)이며, 간단한 회문 검사에 효과적입니다. 예를 들어, "abba"나 "racecar"는 회문으로 판별됩니다.
4. **후위 표기식(Postfix Expression)**
    - 문제: 중위 표기식을 후위 표기식으로 변환하는 과정을 설명하세요.
    - 해설: 중위 표기식에서 연산자는 피연산자 사이에 위치하지만, 후위 표기식에서는 연산자가 피연산자 뒤에 위치합니다. 변환 과정에서는 스택을 사용하여 연산자의 우선순위를 관리합니다. 피연산자를 만나면 바로 출력하고, 연산자를 만나면 스택에 있는 연산자와 우선순위를 비교합니다. 현재 연산자가 스택의 연산자보다 높은 우선순위를 가지면 스택에 푸시하고, 그렇지 않으면 스택의 연산자를 팝하여 출력한 후 현재 연산자를 스택에 푸시합니다. 이 과정을 모든 표현식에 대해 반복하고, 마지막에 스택에 남은 연산자를 모두 팝하여 출력합니다.
5. **시간 복잡도(Time Complexity)**
    - 문제: 점근 표기법을 사용하여 시간 복잡도를 표현하는 방법을 설명하고, O(n), O(log n), O(n^2)의 차이점을 예를 들어 설명하세요.

- 해설: 점근 표기법은 알고리즘의 성능을 표현하는 방법으로, 입력 크기가 무한히 커질 때 알고리즘의 실행 시간이나 공간 요구사항의 상한 또는 하한을 나타냅니다. O(n)은 선형 시간 복잡도를 나타내며, 입력 크기에 비례하여 시간이 증가합니다. 예를 들어, 배열에서 특정 요소를 찾는 선형 검색이 여기에 해당합니다. O(log n)은 로그 시간 복잡도로, 이진 검색과 같이 입력 크기가 커져도 성능이 로그 함수에 따라 증가합니다. O(n^2)은 이차 시간 복잡도로, 입력 크기의 제곱에 비례하여 시간이 증가합니다. 예를 들어, 버블 정렬이나 선택 정렬이 이에 해당합니다. 이러한 표기법은 알고리즘을 비교하고 적절한 알고리즘을 선택하는 데 중요한 기준이 됩니다.

6. **정렬 알고리즘(Sorting Algorithm)**
   - 문제: 삽입 정렬(Insertion Sort)의 작동 원리와 특징을 설명하세요.
   - 해설: 삽입 정렬은 각 반복에서 하나의 입력 요소를 적절한 위치에 삽입하여 정렬된 배열을 만드는 방식입니다. 이 과정은 카드 게임에서 손에 든 카드를 정렬하는 방식과 유사합니다. 첫 번째 요소를 정렬된 것으로 간주하고, 다음 요소를 이미 정렬된 배열 부분과 비교하여 적절한 위치에 삽입합니다. 이 과정을 모든 요소에 대해 반복합니다. 삽입 정렬은 평균과 최악의 경우 시간 복잡도가 O(n^2)이지만, 작은 데이터 세트나 거의 정렬된 데이터에 대해서는 매우 효율적입니다. 또한, 안정적인 정렬 방법이며, 제자리 정렬(in-place sorting) 방식으로 추가 메모리를 거의 사용하지 않습니다.

7. **그래프 탐색(Graph Traversal)**
   - 문제: 너비 우선 탐색(Breadth-First Search, BFS)의 개념과 특징을 설명하세요.
   - 해설: 너비 우선 탐색(BFS)은 그래프의 노드를 방문할 때, 시작 노드에 인접한 노드를 먼저 방문하고, 그 다음으로 인접한 노드들의 인접 노드를 차례로 방문하는 방식입니다. 이 방법은 큐(Queue)를 사용하여 구현되며, 시작 노드를 큐에 넣고, 큐에서 노드를 하나씩 꺼내며 해당 노드의 인접 노드를 모두 큐에 넣는 과정을 반복합니다. BFS는 최단 경로 문제나 그래프의 연결성을 확인하는 데 유용하며, 각 노드는 한 번씩만 방문하므로 시간 복잡도는 O(V+E)입니다(V는 정점의 수, E는 간선의 수).

8. **리스트(List)**
   - 문제: 단순 연결 리스트(Singly Linked List)와 이중 연결 리스트(Doubly Linked List)의 차이점을 설명하세요.
   - 해설: 단순 연결 리스트는 각 노드가 다음 노드만을 가리키는 포인터를 가지고 있는 반면, 이중 연결 리스트는 각 노드가 이전 노드와 다음 노드를 가리키는 두 개의 포인터를 가집니다. 이중 연결 리스트는 양방향으로 탐색이 가능하고, 노드의 삽입과 삭제가 더 유연하지만, 추가적인 메모리 공간을 필요로 합니다. 반면, 단순 연결 리스트는 메모리 사용이 더 적지만, 노드의 삭제나 역방향 탐색이 비효율적일 수 있습니다.

9. **해싱(Hashing)과 체이닝(Chaining)**
    - 문제: 해싱과 체이닝 기법을 설명하고, 해싱에서 체이닝이 어떻게 오버플로우 문제를 해결하는지 설명하세요.
    - 해설: 해싱은 키를 해시 함수를 통해 해시 테이블의 주소로 변환하여 데이터를 저장하고 검색하는 기법입니다. 체이닝은 해시 충돌이 발생했을 때, 같은 해시 주소를 가진 요소들을 연결 리스트로 연결하여 충돌 문제를 해결하는 방법입니다. 이 방식은 해시 테이블의 각 버킷에 연결 리스트를 두어, 충돌이 발생하면 해당 리스트에 요소를 추가합니다. 체이닝은 해시 테이블의 크기에 비해 많은 데이터를 효율적으로 관리할 수 있게 해주며, 해시 테이블의 확장이 필요 없어 관리가 용이합니다.
10. **다익스트라 알고리즘(Dijkstra's Algorithm)**
    - 문제: 다익스트라 알고리즘의 원리와 특징을 설명하고, 어떤 유형의 문제에 적합한지 설명하세요.
    - 해설: 다익스트라 알고리즘은 가중치가 있는 그래프에서 한 정점에서 다른 모든 정점까지의 최단 경로를 찾는 알고리즘입니다. 이 알고리즘은 우선순위 큐를 사용하여 가장 가까운 정점을 선택하고, 이 정점을 통해 다른 정점까지의 거리를 업데이트합니다. 이 과정을 반복하여 모든 정점의 최단 거리를 찾습니다. 다익스트라 알고리즘은 네트워크 라우팅 프로토콜이나 지도에서의 경로 찾기 등에 사용됩니다. 그러나 음수 가중치가 있는 그래프에는 적합하지 않으며, 이 경우 벨만-포드 알고리즘 같은 다른 알고리즘이 사용됩니다.
11. **단순 연결 리스트(Singly Linked List)의 연산**
    - 문제: 단순 연결 리스트에서 add_last 연산과 delete_last 연산의 수행 시간 복잡도를 설명하세요.
    - 해설: 단순 연결 리스트에서 add_last 연산은 리스트의 끝에 새로운 노드를 추가하는 연산입니다. tail 포인터를 사용하면 이 연산은 O(1) 시간에 수행됩니다. 반면, delete_last 연산은 리스트의 마지막 노드를 삭제하는 연산으로, tail 포인터가 마지막 노드의 바로 이전 노드를 가리키지 않는 한, 전체 리스트를 순회해야 하므로 O(n) 시간이 소요됩니다. 이러한 특성 때문에 단순 연결 리스트는 끝 부분에 대한 연산이 빈번한 경우 비효율적일 수 있습니다.
12. **히프(Heap)와 히프 정렬(Heap Sort)**
    - 문제: 최소 히프(min heap)의 특성과 히프 정렬의 과정을 설명하세요.
    - 해설: 최소 히프는 부모 노드의 값이 자식 노드의 값보다 항상 작거나 같은 완전 이진 트리입니다. 이 특성으로 인해 트리의 루트에는 항상 가장 작은 요소가 위치합니다. 히프 정렬은 이러한 히프의 특성을 이용하여 정렬을 수행합니다. 먼저 모든 요소를 최소 히프로 구성한 후, 가장 작은 요소(루트)를 히프의 마지막 요소와 교환하고,

히프의 크기를 하나 줄입니다. 이후 히프를 재구성하고, 이 과정을 반복하여 정렬을 완료합니다. 히프 정렬의 시간 복잡도는 O(nlogn)이며, 추가 메모리를 거의 사용하지 않는 제자리 정렬 방식입니다.

13. **함수의 시간 복잡도(Time Complexity)**
    - 문제: 다양한 C언어 함수의 시간 복잡도를 비교하고, 가장 효율적인 함수를 결정하세요.
    - 해설: 함수의 시간 복잡도는 알고리즘의 성능을 결정하는 중요한 요소입니다. 예를 들어, 두 중첩된 for 루프로 구성된 함수는 일반적으로 O(n^2)의 시간 복잡도를 가집니다. 반면, for 루프의 반복 횟수가 입력 크기에 로그 스케일로 증가하는 경우, 함수의 시간 복잡도는 O(nlogn)이 됩니다. 가장 효율적인 함수는 반복 횟수가 가장 적고, 불필요한 계산을 최소화하는 함수입니다. 예를 들어, 선형 시간 복잡도(O(n)) 또는 로그 시간 복잡도(O(logn))를 가지는 함수는 이차 시간 복잡도(O(n^2))를 가지는 함수보다 효율적입니다.
14. **연결 리스트(Linked List)의 순회**
    - 문제: 주어진 단순 연결 리스트의 메모리 구조를 바탕으로, 리스트의 원소를 순서대로 나열하세요.
    - 해설: 연결 리스트의 순회는 리스트의 첫 번째 노드부터 시작하여, 각 노드의 '다음' 포인터를 따라 다음 노드로 이동하는 과정을 반복하는 것입니다. 이 과정은 마지막 노드(보통 '다음' 포인터가 NULL인 노드)에 도달할 때까지 계속됩니다. 순회 과정에서 각 노드의 데이터를 읽거나 처리할 수 있으며, 이를 통해 리스트의 모든 원소를 순서대로 접근할 수 있습니다. 연결 리스트의 순회는 O(n) 시간 복잡도를 가지며, 리스트의 모든 원소를 방문하거나 검색하는 데 사용됩니다.
15. **스레드 이진트리(Threaded Binary Tree)**
    - 문제: 스레드 이진트리의 개념과 중위 순회(inorder traversal) 시의 장점을 설명하세요.
    - 해설: 스레드 이진트리는 이진트리의 노드 중 자식 노드가 없는 부분에 다음 순회할 노드를 가리키는 포인터(스레드)를 추가한 트리입니다. 이 구조는 중위 순회 시 스택이나 재귀 호출 없이 트리를 순회할 수 있게 해줍니다. 스레드 이진트리에서는 각 노드가 다음에 순회할 노드를 직접 가리키므로, 순회 과정이 더 빠르고 메모리 효율적입니다. 특히, 중위 순회를 자주 수행해야 하는 경우에 이점이 있으며, 추가적인 스택 공간이나 재귀 호출의 오버헤드 없이 순회가 가능합니다. 이러한 특성은 트리의 순회와 검색 작업을 효율적으로 만들어줍니다.
16. **다차원 배열(Multidimensional Array)**

- 문제: C언어에서 3차원 배열의 메모리 구조를 설명하고, 특정 원소에 접근하는 방법을 설명하세요.
- 해설: C언어에서 다차원 배열은 연속된 메모리 공간에 저장됩니다. 3차원 배열 A[5][6][7]의 경우, 각 차원은 5개의 2차원 배열, 각 2차원 배열은 6개의 1차원 배열로 구성되며, 각 1차원 배열은 7개의 원소를 가집니다. 배열의 원소는 행우선(row-major) 방식으로 저장되어, A[0][0][0], A[0][0][1], ..., A[0][0][6], A[0][1][0], ..., A[4][5][6] 순으로 메모리에 배치됩니다. 특정 원소에 접근하기 위해서는 배열 이름과 인덱스를 사용합니다. 예를 들어, A[2][3][4]는 배열의 세 번째 "페이지"의 네 번째 "행"의 다섯 번째 "열"에 위치한 원소를 가리킵니다.

17. **2-3 트리(2-3 Tree)**
    - 문제: 2-3 트리의 구조와 키 삽입 과정을 설명하세요.
    - 해설: 2-3 트리는 각 노드가 최대 3개의 자식을 가질 수 있는 균형 이진 검색 트리입니다. 노드는 하나 또는 두 개의 키를 가질 수 있으며, 이에 따라 2-노드 또는 3-노드로 분류됩니다. 새로운 키를 삽입할 때, 트리는 먼저 삽입 위치를 찾은 후, 해당 위치에 키를 추가합니다. 만약 삽입으로 인해 노드에 키가 3개가 되면, 중간 값은 부모 노드로 올라가고, 나머지 두 키는 새로운 두 노드로 분할됩니다. 이 과정은 트리가 균형을 유지하도록 도와주며, 검색, 삽입, 삭제 연산의 시간 복잡도는 모두 O(logn)입니다.
18. **인접 행렬(Adjacency Matrix)**
    - 문제: 인접 행렬을 사용한 그래프 표현 방법을 설명하고, 그 특징에 대해 설명하세요.
    - 해설: 인접 행렬은 그래프의 노드 간 연결 관계를 행렬로 표현하는 방법입니다. 그래프의 각 노드는 행렬의 행과 열에 해당하며, 두 노드가 연결되어 있으면 해당 위치의 값이 1(또는 연결의 가중치), 연결되어 있지 않으면 0으로 표시됩니다. 인접 행렬은 노드 간 연결 정보를 직관적으로 표현할 수 있으며, 두 노드 간 연결 여부를 O(1) 시간에 확인할 수 있습니다. 그러나 모든 노드 쌍에 대한 정보를 저장해야 하므로, 메모리 사용량이 많고, 대부분의 공간이 불필요한 정보(0)로 채워질 수 있습니다. 이 방법은 노드 수가 적고, 밀집된 그래프에 적합합니다.
19. **트리 순회(Tree Traversal)**
    - 문제: 이진 트리의 중위 순회 방법을 설명하고, 주어진 트리 구조에 대해 중위 순회 결과를 제시하세요.
    - 해설: 이진 트리의 중위 순회는 왼쪽 자식 노드, 부모 노드, 오른쪽 자식 노드 순으로 노드를 방문하는 방법입니다. 이 순회 방식은 트리의 노드를 순서대로 방문하며, 특히 이진 검색 트리에서는 노드를 오름차순으로 방문합니다. 주어진 트리 구조에 대

해 중위 순회를 수행하면, 노드를 왼쪽 서브트리, 루트, 오른쪽 서브트리의 순서로 방문하게 됩니다. 이 방법은 트리의 구조를 이해하거나 정렬된 데이터를 추출하는 데 유용합니다.

20. **이진 탐색 트리(Binary Search Tree)**
    - 문제: 이진 탐색 트리의 생성 과정을 설명하고, 특정 입력 순서에 따른 트리의 형태를 설명하세요.
    - 해설: 이진 탐색 트리는 각 노드가 최대 두 개의 자식을 가지며, 왼쪽 자식의 값은 부모보다 작고, 오른쪽 자식의 값은 부모보다 큰 특성을 가진 트리입니다. 트리 생성 시, 새로운 값을 삽입할 때마다 이 규칙을 유지하면서 적절한 위치를 찾아 삽입합니다. 입력 순서에 따라 트리의 형태가 달라질 수 있으며, 균형이 잘 잡힌 트리는 검색, 삽입, 삭제 연산에서 좋은 성능을 보입니다. 그러나 일부 입력 순서에서는 트리가 한쪽으로 치우쳐 비효율적인 형태(예: 선형 리스트와 유사한 형태)가 될 수 있습니다.
21. **재귀 함수(Recursive Function)**
    - 문제: 재귀 함수의 개념을 설명하고, 재귀 함수를 사용할 때 고려해야 할 주요 사항들은 무엇인지 설명하세요.
    - 해설: 재귀 함수는 자기 자신을 호출하여 문제를 해결하는 함수입니다. 이 방식은 복잡한 문제를 간단하고 우아하게 해결할 수 있도록 도와줍니다. 재귀 함수를 사용할 때는 종료 조건을 명확히 정의해야 하며, 종료 조건이 없거나 잘못된 경우 무한 재귀에 빠질 위험이 있습니다. 또한, 재귀 호출의 깊이가 너무 깊어지면 스택 오버플로우가 발생할 수 있으므로, 이를 고려한 설계가 필요합니다. 재귀는 분할 정복 알고리즘, 트리나 그래프의 순회 등 다양한 문제에서 유용하게 사용됩니다.
22. **동적 프로그래밍(Dynamic Programming)**
    - 문제: 동적 프로그래밍의 기본 원리와 적용 사례를 설명하세요.
    - 해설: 동적 프로그래밍은 복잡한 문제를 작은 하위 문제로 나누고, 각 하위 문제의 해결 결과를 저장하여 중복 계산을 방지하는 방법입니다. 이 방식은 주로 최적화 문제에서 사용되며, 하위 문제의 중복이 많은 경우 효과적입니다. 예를 들어, 피보나치 수열 계산, 최단 경로 찾기, 배낭 문제 등에서 동적 프로그래밍이 널리 사용됩니다. 동적 프로그래밍을 사용할 때는 문제를 하위 문제로 나눌 수 있는지, 하위 문제의 해결 결과를 어떻게 저장할 것인지 고려해야 합니다.