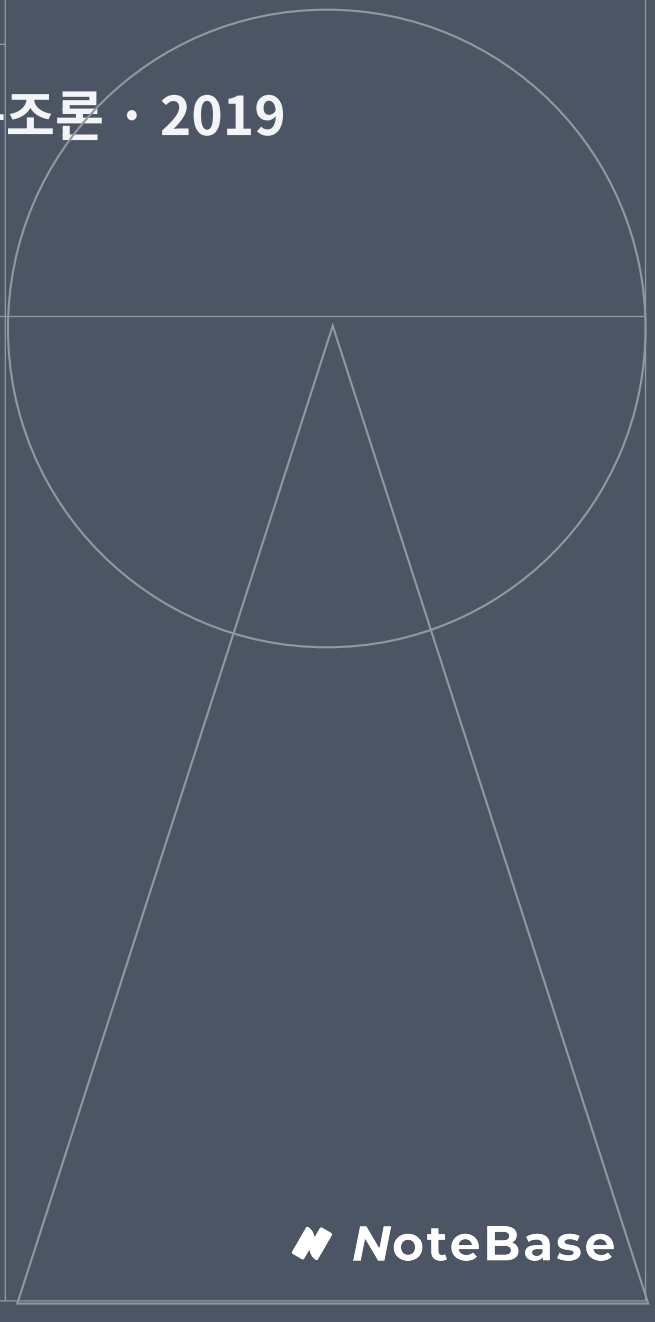
기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2019



NoteBase

기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2019

공무원/국가직/7급 · 자료구조론 · 2019

문 1. m원 탐색 트리(m-way search tree)에 대한 설명으로 옳지 않은 것은?

① 공백(empty)이 아닌 m원 탐색 트리의 루트(root) 노드는 최대 m개의 서브-트리(sub-tree)를 가질 수 있다.

② 공백이 아닌 m원 탐색 트리의 루트 노드에 저장된 키(key)들은 정렬되어 있어야 한다.

③ 공백이 아닌 m원 탐색 트리의 각 노드가 가지는 서브-트리 역시 m원 탐색 트리이다.

④ 차수(degree)가 m이고 높이(height)가 h인 다원 탐색 트리가 가질 수 있는 최대 노드 수는 $(m^h - 1)$이다. (단, 루트만 있는 트리의 높이는 1로 한다)

정답 4번

출제 포인트

이 문제는 m원 탐색 트리의 기본 성질과 높이별 최대 노드 수 공식을 정확히 구분하는지를 묻는 문제이다.

핵심 개념

m원 탐색 트리는 한 노드가 최대 m개의 서브트리와 최대 $m-1$개의 키를 가지는 탐색 트리이다.
노드 안의 키들은 오름차순으로 정렬되고, 키 사이 구간에 대응하는 각 서브트리도 같은 m원 탐색 트리 성질을 만족한다.
높이를 루트만 있을 때 1로 보면 최대 노드 수는 각 레벨의 노드 수를 합한 $1 + m + m^2 + \cdots + m^{h-1}$이므로 $\frac{m^h - 1}{m - 1}$이다.

선지별 해설

① O: m원 탐색 트리에서 한 노드가 가질 수 있는 자식 서브트리 수의 최댓값은 m개이므로 루트도 최대 m개의 서브트리를 가질 수 있다.
② O: 탐색을 가능하게 하려면 한 노드 내부의 키들이 정렬되어 있어야 한다. 그래야 키 사이 구간별로 내려갈 서브트리가 정해진다.
③ O: m원 탐색 트리의 각 서브트리는 동일한 정렬 규칙과 자식 수 제한을 그대로 만족하므로 역시 m원 탐색 트리이다.
④ X: 틀린 부분은 최대 노드 수를 $m^h - 1$이라고 한 표현이다. 실제로는 각 레벨의 최대 노드 수를 모두 더해야 하므로 최대 노드 수는 $1 + m + \cdots + m^{h-1} = \frac{m^h - 1}{m - 1}$이다.

NoteBase

오답 포인트

④는 마지막 레벨의 크기 $m^h - 1$만 보고 등비수열의 합 공식을 빠뜨리면 쉽게 속는 선택지이다. m원 탐색 트리의 높이 문제는 한 레벨의 노드 수와 전체 노드 수를 구분해야 한다.

다시 나오면 이렇게 판단

트리의 성질 문제는 먼저 한 노드의 최대 자식 수, 노드 내부 키의 정렬 여부, 서브트리의 재귀적 동일성을 확인한다. 높이와 최대 개수를 묻는 문항은 각 레벨의 개수를 더한 값인지, 마지막 레벨 하나의 값인지를 분리해서 판단한다.

암기 포인트

m원 탐색 트리는 노드당 자식 최대 m개, 키는 정렬, 최대 노드 수는 $\frac{m^h - 1}{m - 1}$이다.

NoteBase

문 2. 다음은 C 언어로 구현한 원형 큐(circular queue) 삽입 알고리즘이다. ㉠, ㉡에 들어갈 내용으로 옳은 것은?

```
#define MAX_QUEUE_SIZE 10
int queue[MAX_QUEUE_SIZE];
int front = rear = -1;

void addq(int front, int *rear, int item) {
   _____㉠_____
   if (_____㉡_____) {
      printf("Queue is full!!\n");
      return -1;
   }
   queue[*rear] = item;
}
```

① ㉠: *rear = (*rear+1) % MAX_QUEUE_SIZE; ㉡: front == *rear

② ㉠: *rear = (front+1) % MAX_QUEUE_SIZE; ㉡: front == *rear + 1

③ ㉠: front = (front+1) % MAX_QUEUE_SIZE; ㉡: front == *rear

④ ㉠: front = (*rear+1) % MAX_QUEUE_SIZE; ㉡: front == *rear + 1

정답 1번

출제 포인트

이 문제는 원형 큐 삽입에서 rear를 어떻게 이동시키고, 그 직후 어떤 조건으로 포화를 판정하는지를 묻는 문제이다.

올바른 순서

1. rear를 한 칸 전진시킨다. `*rear = (*rear + 1) % MAX_QUEUE_SIZE`로 배열 끝 다음에 다시 0으로 돌아가게 한다.
2. 전진한 rear가 front와 같은지 검사한다. 같아지면 삽입할 빈 칸 없이 찬 상태로 본다.
3. 포화이면 오류를 내고 삽입을 중단한다. 더 진행하면 기존 원소와 위치가 충돌한다.
4. 포화가 아니면 `queue[*rear] = item;`으로 새 원소를 저장한다.

왜 이 순서: 포화 여부는 이동이 끝난 rear 위치를 기준으로 판정되므로 rear 갱신이 먼저 와야 한다.

NoteBase

선지별 해설

① O
ㄱ의 `*rear = (*rear+1) % MAX_QUEUE_SIZE`는 삽입 위치를 원형으로 한 칸 전진시키는 식이다. ㄴ의 `front == *rear`는 전진한 rear가 front와 만나면 포화로 보는 조건이므로 문항이 의도한 삽입 알고리즘과 맞는다.

② X
틀린 부분은 ㄱ의 `*rear = (front+1) % MAX_QUEUE_SIZE`이다. rear를 front 기준으로 계산하면 현재 rear의 다음 칸으로 이동하는 삽입 흐름이 깨진다. 올바르게는 `*rear = (*rear+1) % MAX_QUEUE_SIZE`이다. ㄴ도 이 문항의 의도와는 맞지 않는다. 포화 판정은 rear를 이동시킨 뒤 `front == *rear`로 본다.

③ X
틀린 부분은 ㄱ의 `front = (front+1) % MAX_QUEUE_SIZE`이다. 삽입 연산에서는 rear가 이동하고 front는 삭제 연산에서 이동한다. 올바르게는 `*rear = (*rear+1) % MAX_QUEUE_SIZE`이다. ㄴ의 `front == *rear` 자체는 포화 조건으로 맞지만, ㄱ이 잘못되어 전체 보기는 틀리다.

④ X
틀린 부분은 ㄱ의 `front = (*rear+1) % MAX_QUEUE_SIZE`와 ㄴ의 `front == *rear + 1`이다. 삽입 시 움직여야 할 포인터는 front가 아니라 rear이다. 또한 `front == *rear + 1`은 원형 순환을 고려한 modulo 형태가 아니어서 끝에서 0으로 돌아가는 경우를 제대로 표현하지 못한다. 올바르게는 `*rear = (*rear+1) % MAX_QUEUE_SIZE`로 이동시킨 뒤 `front == *rear`를 검사한다.

오답 포인트

삽입 연산인데 front를 움직이는 선택지를 고르거나, 포화 조건을 rear와 front의 충돌이 아니라 단순한 `+1` 비교로 외우면 틀리기 쉽다. 원형 큐에서는 배열 끝에서 다시 0으로 돌아가는 modulo 연산이 핵심 함정이다.

다시 나오면 이렇게 판단

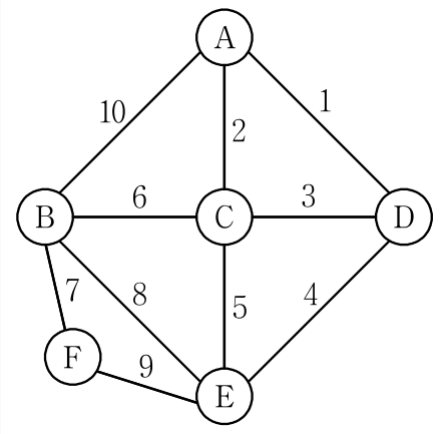
원형 큐 문제는 먼저 연산이 삽입인지 삭제인지 구분한다. 삽입이면 rear를, 삭제면 front를 움직인다고 보고, 그다음 포인터 이동 결과가 상대 포인터와 충돌하는지로 공백 여부나 포화를 판단한다.

암기 포인트

원형 큐 삽입은 rear 전진 후 충돌 검사이고, 삭제는 front 전진이다.

NoteBase

문 3. 다음 그래프를 Kruskal 알고리즘을 이용하여 최소 비용 신장 트리로 만들 때 선택되지 않는 간선은? (단, 알고리즘은 노드 A에서 시작한다)



① (A, C)

② (B, C)

③ (B, F)

④ (C, D)

정답 4번

출제 포인트

이 문제는 가중치가 작은 간선부터 보되 사이클이 생기면 제외하는 Kruskal 알고리즘의 선택 기준을 그래프에서 적용할 수 있는지를 묻는다.

선지별 해설

1번 (A, C): O. 가중치 2인 간선으로, 1순위 간선 (A, D) 다음에 사이클 없이 추가된다.
2번 (B, C): O. 앞에서 선택된 간선은 (A, D), (A, C), (D, E)이고 B는 아직 분리되어 있으므로 가중치 6인 (B, C)는 사이클 없이 추가된다.
3번 (B, F): O. (B, C)까지 선택한 뒤 F는 아직 연결되지 않았으므로 가중치 7인 (B, F)는 최소 비용 신장 트리를 완성하는 데 선택된다.
4번 (C, D): X. 가중치 3인 간선이지만, 그보다 앞서 (A, D)와 (A, C)가 이미 선택되어 C-A-D 경로가 만들어진 상태이므로 (C, D)를 넣으면 사이클이 생겨 제외된다. 틀린 표현은 '선택된다'이고, 올바르게는 '사이클을 만들므로 선택되지 않는다'이다.

NoteBase

오답 포인트

가중치가 작다고 해서 무조건 선택되는 것은 아니다. 이 문항은 가중치 3인 (C, D)가 작지만 이미 연결된 정점 사이를 이어 사이클을 만든다는 점을 놓치기 쉽다. 또한 '노드 A에서 시작한다'는 문구는 Prim 알고리즘에서 중요한 조건이고 Kruskal에서는 선택 기준이 아니다.

다시 나오면 이렇게 판단

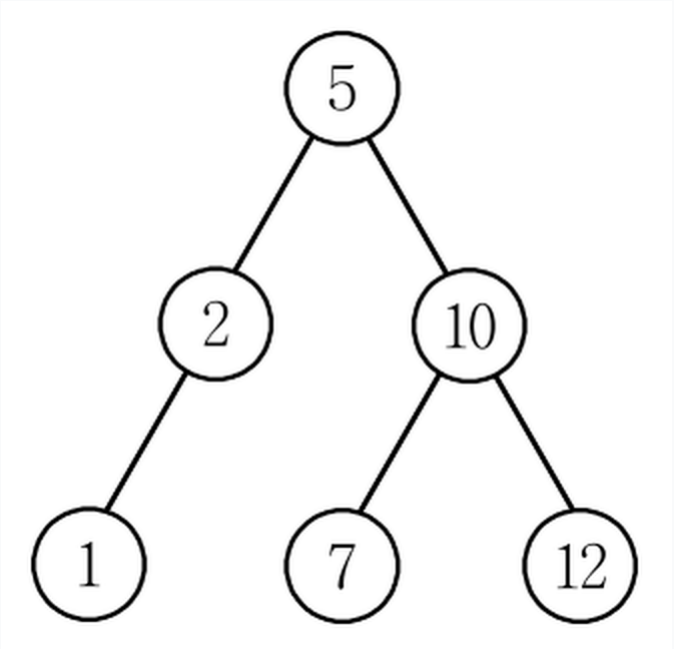
Kruskal 문제는 간선을 가중치 오름차순으로 나열한 뒤, 각 단계에서 현재 선택된 간선들과 사이클을 만드는지만 확인하면 된다.

암기 포인트

Kruskal은 시작 정점이 아니라 가중치 오름차순과 사이클 배제가 핵심이다.

NoteBase

문 4. 다음은 이진 트리(binary tree)이다. 매개변수 ptr로 함수 func를 호출하였을 때, 결과 값은? (단, ptr은 루트 노드에 대한 포인터이다)




```
struct node {
    int data;
    struct node *left;
    struct node *right;
};
int func(struct node *ptr) {
    if (ptr == NULL)
       return 0;
    else if (ptr->left == NULL
                && ptr->right == NULL)
       return 1;
    else
       return (ptr->data + func(ptr->left)
                   + func(ptr->right));
}
```

① 10

② 17

③ 20

④ 37

NoteBase

정답 3번

출제 포인트

이 문제는 이진 트리 도식과 재귀 함수의 종료 조건을 함께 읽어, 각 부분트리의 반환값이 어떻게 누적되는지를 계산하는 문제이다.

선지별 해설

① X
10은 노드 10의 data일 뿐 전체 트리에 대해 func(ptr)를 호출한 결과가 아니다. 루트 5에서 시작한 호출은 왼쪽·오른쪽 부분트리 반환값까지 모두 더하므로 올바른 결과값은 20이다.

② X
17은 오른쪽 부분트리 값 12에 루트 5만 더하고, 왼쪽 부분트리의 반환값 3을 누락한 값이다. 실제 계산은 func(2)=3, func(10)=12, func(5)=20이다. 올바른 결과값은 20이다.

③ O
리프 노드 1, 7, 12는 모두 left와 right가 NULL이므로 각각 1을 반환한다. 따라서 func(2)=2+1+0=3, func(10)=10+1+1=12, func(5)=5+3+12=20이므로 20이 맞다.

④ X
37은 5, 2, 1, 10, 7, 12를 그대로 모두 더한 값이다. 그러나 이 함수는 리프 노드에서 data를 더하지 않고 1을 바로 반환한다. 따라서 리프 1, 7, 12는 각각 1로 처리되어야 하므로 올바른 결과값은 20이다.

오답 포인트

리프 노드의 data까지 합산한다고 착각하면 37로 흐르기 쉽다. 이 함수의 핵심 함정은 리프에서 값을 반환하는 조건이 `ptr->data`가 아니라 `1`이라는 점이다.

다시 나오면 이렇게 판단

재귀 함수 문제는 먼저 종료 조건이 무엇을 반환하는지 본다. 그다음 리프, 한 단계 위 노드, 루트 순서로 부분결과를 아래에서 위로 누적하면 전체 반환값을 안정적으로 구할 수 있다.

암기 포인트

재귀 트리 계산은 리프 반환값부터 확정하고 부모로 올라간다.

NoteBase

문 5. $n \times n$ 하삼각 행렬은 총 $n(n+1)/2$개의 원소를 갖는다. 원소 $a_{ij}(1 \le i \le j \le n)$가 최소 공간을 사용하도록 1차원 배열 $b[k]$에 행우선순서(row-major order)로 저장했을 때, 하삼각 행렬의 원소 a_{ij}가 저장될 배열 $b[k]$의 색인 k를 계산하는 식으로 옳은 것은? (단, i는 행 색인 값, j는 열 색인 값으로 하고, 1차원 배열 b에는 하삼각 행렬의 0 값은 저장되지 않으며, b의 색인 k의 값은 0부터 시작한다)

정방 행렬(square matrix) 중에서 대각선보다 위의 모든 원소가 0인 경우를 특별히 하삼각 행렬(lower triangular matrix)이라 하며, 다음은 4×4 하삼각 행렬의 예이다. (단, a_{ij}는 0이 아닌 어떤 실수 값을 의미하고, 0은 반드시 0이어야 함을 의미한다)

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

① $(i-1) \times n + (j-1)$

② $j \times (j-1)/2 + (i-1)$

③ $i \times (i-1)/2 + (j-1)$

④ $(j-1) \times n + (i-1)$

정답 3번

출제 포인트

이 문제는 하삼각 행렬을 행우선순서로 압축 저장할 때 원소 a_{ij}가 1차원 배열에서 몇 번째 위치에 들어가는지를 구하는 문제이다.

조건 정리

하삼각 행렬이므로 저장 대상은 예시 행렬 기준으로 $j \le i$인 원소이다.
배열 b에는 0인 원소를 저장하지 않으므로 대각선 위의 칸은 모두 건너뛴다.
행우선순서이므로 현재 원소 앞에는 1행부터 $(i-1)$행까지의 저장 원소가 먼저 온다.

사용할 공식

1행에는 1개, 2행에는 2개, ..., $(i-1)$행에는 $(i-1)$개의 원소가 저장된다. 따라서 i행 앞까지의 누적 개수는 $1 + 2 + \cdots + (i-1) = i(i-1)/2$이다.
여기에 i행 안에서 a_{i1}부터 a_{ij} 직전까지의 개수 $(j-1)$을 더하면 인덱스는 $k = i(i-1)/2 + (j-1)$이다.

NoteBase

계산 과정

1. 행우선순서에서는 먼저 같은 행보다 앞선 모든 행의 저장 원소 수를 센다.
2. 하삼각 행렬의 앞선 행 원소 수는 $1 + 2 + \cdots + (i-1) = i(i-1)/2$이다.
3. 현재 i행에서 j열까지 가려면 왼쪽에 있는 저장 원소 수 $(j-1)$을 더한다.
4. 따라서 $k = i(i-1)/2 + (j-1)$이므로 3번이 맞다.

선지별 해설

1번: X. $(i-1)n + (j-1)$은 $n \times n$ 정방행렬 전체를 0까지 포함해 행우선순서로 저장할 때의 식이다. 이 문제는 0을 저장하지 않는 압축 저장이므로 앞행 누적은 n의 배수가 아니라 $1 + 2 + \cdots + (i-1)$이어야 한다. 틀린 표현은 전체 행 길이 n을 그대로 곱한 부분이고, 올바르게는 $i(i-1)/2 + (j-1)$이다.
2번: X. 이 식은 앞선 위치의 개수를 열 인덱스 j 기준으로 누적한 형태라서 행우선순서와 맞지 않는다. 행우선순서 압축 저장에서는 몇 개의 행이 먼저 저장되었는지가 핵심이므로 누적항은 j가 아니라 i로 만들어져야 한다. 올바르게는 $i(i-1)/2 + (j-1)$이다.
3번: O. i행 앞에는 총 $i(i-1)/2$개의 원소가 저장되고, 같은 행 안에서 j열 원소 앞에는 $(j-1)$개의 원소가 있으므로 $k = i(i-1)/2 + (j-1)$가 된다.
4번: X. $(j-1)n + (i-1)$은 행과 열의 역할을 바꿔 정방행렬 전체를 열 기준으로 세는 형태이다. 하삼각 압축의 행우선 저장에서는 열이 아니라 행이 먼저 기준이 된다. 틀린 표현은 $(j-1)n$으로 앞부분을 세는 방식이고, 올바르게는 $i(i-1)/2 + (j-1)$이다.

오답 포인트

0을 저장하지 않는 압축 배열이라는 조건을 놓치고 일반 정방행렬의 행우선 또는 열우선 인덱스 식을 그대로 대입하면 1번과 4번에 끌리기 쉽다. 또 행우선순서인데 누적 기준을 열 j로 잡으면 2번처럼 틀리게 된다.

다시 나오면 이렇게 판단

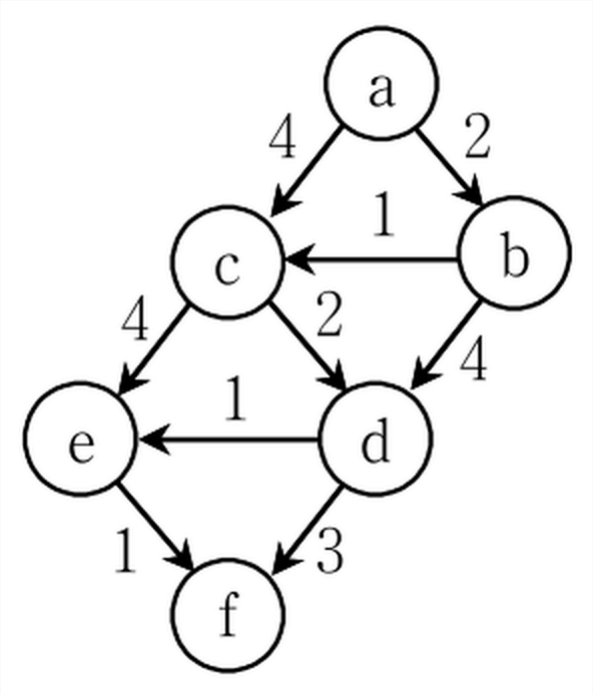
압축 저장 인덱스 문제는 먼저 0을 저장하는지부터 확인하고, 그다음 현재 원소 앞에 완전히 저장된 행 또는 열의 개수를 누적한 뒤 같은 행 또는 열 내부의 이동 칸수를 더하는 순서로 판단한다.

암기 포인트

하삼각 행우선 압축 인덱스는 앞행 누적 $i(i-1)/2$에 같은 행 왼쪽 칸수 $(j-1)$을 더한 값이다.

NoteBase

문 6. 다음과 같은 방향 그래프에 대하여 Dijkstra의 알고리즘을 적용하여 최단경로를 구하고자 한다. 노드 a에서 시작하여 노드 f로 가는 최단 경로를 찾아 가는 과정에서 노드 a에서부터 다른 노드까지 최단 경로를 차례로 알게 된다. 이 과정에서 알게 되는 최단 경로의 노드 순서로 옳은 것은?



① a - b - c - d - e - f

② a - b - d - f

③ a - c - d - f

④ a - c - d - e - f

정답 1번

출제 포인트

이 문제는 방향 그래프의 간선 방향과 가중치를 따라 Dijkstra 알고리즘에서 각 노드의 최단거리가 확정되는 순서를 판단하는 문제이다.

NoteBase

선지별 해설

1번: O. 시작 노드 a의 거리는 0이다. 먼저 b=2, c=4가 되고 b가 확정된다. b를 거치면 c는 3, d는 6이 되므로 다음은 c이다. c를 거치면 d는 5, e는 7이 되므로 다음은 d이다. d를 거치면 e는 6, f는 8이 되고, e를 거치면 f는 7로 줄어든다. 따라서 확정 순서는 a-b-c-d-e-f이다.
2번: X. d가 c보다 먼저 확정될 수 없다는 점이 틀렸다. a에서 b를 확정한 직후 c=3, d=6이므로 세 번째 확정 노드는 d가 아니라 c이다. 틀린 표현은 "a-b-d-f"이고, 올바르게는 "a-b-c-d-e-f"이다.
3번: X. b를 건너뛰고 c가 먼저 확정된다고 본 점이 틀렸다. a에서 직접 c로 가면 4이지만 a-b-c는 2+1=3이므로 b가 먼저 확정되고 그 다음에 c가 확정된다. 틀린 표현은 "a-c-d-f"이고, 올바르게는 "a-b-c-d-e-f"이다.
4번: X. e가 최종적으로 d 다음에 확정되는 점은 맞지만, 그 전에 b와 c가 확정되는 순서를 빠뜨린 점이 틀렸다. Dijkstra의 확정 순서는 일부 노드만 뽑아 적는 방식이 아니라 모든 미확정 노드 중 최솟값이 차례대로 확정되는 순서이다. 틀린 표현은 "a-c-d-e-f"이고, 올바르게는 "a-b-c-d-e-f"이다.

오답 포인트

a에서 c로 가는 직접 간선의 가중치 4만 보고 c를 먼저 확정하거나, 최종 최단경로 a-b-c-d-e-f와 각 단계의 확정 순서를 혼동하면 틀리기 쉽다. 이 문제의 함정은 b를 거친 후 c의 값이 3으로 더 작아진다는 점과, f에 도달하기 전에도 e의 최단거리가 먼저 확정된다는 점이다.

다시 나오면 이렇게 판단

시작 노드에서 초기 거리표를 만든 뒤, 미확정 노드 중 현재 거리가 가장 작은 노드를 하나씩 확정한다. 그 노드의 출발 간선만 완화하여 거리표를 갱신하고 다시 최솟값을 고르면 확정 순서를 안정적으로 구할 수 있다.

암기 포인트

Dijkstra는 "현재 최솟값 확정 -> 인접 간선 완화"의 반복이므로 직접 간선보다 우회 누적값이 더 작아지면 확정 순서가 바뀐다.

NoteBase

문 7. 다음과 같이 배열에 저장된 데이터에 대하여 내림차순으로 히프 정렬(heap sort)을 수행할 때, 첫 번째 데이터를 출력하고 히프를 재구성 한 후에 배열의 여섯 번째에 있는 값은? (단, 배열의 첫 번째 색인 값은 1이다)

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
21	37	22	17	11	30	5	10	9

① 5

② 9

③ 10

④ 11

정답 2번

출제 포인트

내림차순 힙 정렬에서 최대 힙을 만든 뒤 첫 삭제와 재구성을 수행하면 배열 상태가 어떻게 바뀌는지를 묻는 문제이다.

올바른 순서

1. 주어진 배열을 최대 힙으로 구성하면 [37, 21, 30, 17, 11, 22, 5, 10, 9]가 된다.
2. 첫 번째 출력값은 루트인 37이며, 마지막 원소 9를 루트로 올린다.
3. 루트 9는 자식 21, 30 중 큰 값인 30과 교환되어 [30, 21, 9, 17, 11, 22, 5, 10]이 된다.
4. 인덱스 3의 9는 다시 자식 22, 5 중 큰 값인 22와 교환되어 [30, 21, 22, 17, 11, 9, 5, 10]이 된다.

왜 이 순서인가: 내림차순 힙 정렬은 최대 힙을 사용하며, 루트로 올라온 값은 최대 힙 조건을 만족할 때까지 아래로 내려가야 한다.

NoteBase

선지별 해설

① X
5는 내림차순이 아니라 오름차순용 최소 힙을 떠올렸을 때 나오는 값이다. 이 문항은 최대 힙으로 재구성하므로 여섯 번째 값은 5가 아니라 9이다.

② O
최대 힙 재구성 결과가 [30, 21, 22, 17, 11, 9, 5, 10]이므로 여섯 번째 값은 9이다.

③ X
10은 루트로 올라온 9를 한 단계만 내리거나 마지막 레벨 배치를 혼동했을 때 나오는 값이다. 9는 22와 한 번 더 교환되므로 여섯 번째 값은 10이 아니라 9이다.

④ X
11은 재구성 뒤 인덱스 5의 값을 인덱스 6과 혼동한 값이다. 최대 힙을 끝까지 복구하면 인덱스 6에는 9가 놓이므로 11이 아니라 9가 맞다.

오답 포인트

내림차순이라는 조건을 보고도 최소 힙을 떠올리는 경우가 많고, 루트에 마지막 원소를 올린 뒤 하향 재조정을 한 번만 하고 멈추는 경우도 많다. 이 문항의 함정은 힙 종류와 sift-down의 반복 여부를 동시에 묻는 데 있다.

다시 나오면 이렇게 판단

힙 정렬 배열 문제는 먼저 정렬 방향에 맞는 힙 종류를 정한 뒤, 루트 삭제 후 올라온 원소를 자식과 비교하여 힙 조건을 만족할 때까지 끝까지 내려보면 된다.

암기 포인트

내림차순 힙 정렬은 최대 힙을 사용하고, 삭제 뒤 루트로 올라온 값은 sift-down으로 끝까지 내려간다.

NoteBase

문 8. 6개 외부 노드의 가중치가 각각 $q_1 = 3$, $q_2 = 4$, $q_3 = 6$, $q_4 = 8$, $q_5 = 10$, $q_6 = 15$인 허프만 트리의 설명으로 옳은 것은? (단, 허프만 트리는 $\sum_{1 \le i \le n} q_i d_i$가 최소가 되는 트리로서, q_i는 외부 노드의 가중치이고 d_i는 루트 노드에서부터 외부 노드까지의 거리이며, n은 외부 노드의 수이다)

① 외부 노드에서 루트까지의 거리와 외부 노드의 가중치 곱의 총합 ($\sum_{1 \le i \le n} q_i d_i$)은 112이다.

② 루트까지의 거리가 가장 긴 외부 노드에서 루트까지의 거리는 5이다.

③ 루트까지의 거리가 가장 긴 외부 노드는 q_1, q_2, q_3이다.

④ 루트까지의 거리가 가장 짧은 외부 노드는 q_6이고 그 다음으로 짧은 외부 노드는 q_4와 q_5이다.

정답 1번

출제 포인트

이 문제는 허프만 트리에서 가장 작은 가중치 두 개씩 결합하는 과정을 통해 $WPL = \sum q_i d_i$와 각 외부 노드의 깊이를 구할 수 있는지를 묻는다.

조건 정리

주어진 가중치는 $q_1 = 3, q_2 = 4, q_3 = 6, q_4 = 8, q_5 = 10, q_6 = 15$이다.
구해야 할 값은 WPL과 각 외부 노드의 루트까지 거리이다.
허프만 트리는 매 단계에서 가장 작은 가중치 두 개를 결합하며, 외부 노드의 깊이는 그 노드가 결합에 포함된 횟수와 같다.

사용할 공식

허프만 트리의 가중 경로 길이는
$WPL = \sum_{i=1}^{n} q_i d_i$
이며, 허프만 트리에서는 각 결합 단계에서 새로 만들어진 내부 노드의 가중치 합을 모두 더한 값과 같다.
이 공식을 쓰는 이유는 두 노드를 한 번 결합할 때마다 그 부분트리에 속한 모든 외부 노드의 깊이가 1씩 증가하기 때문이다.

계산 과정

1단계: 3과 4를 결합하여 7을 만든다.
2단계: 6과 7을 결합하여 13을 만든다.
3단계: 8과 10을 결합하여 18을 만든다.
4단계: 13과 15를 결합하여 28을 만든다.
5단계: 18과 28을 결합하여 루트 46을 만든다.
따라서 $WPL = 7 + 13 + 18 + 28 + 46 = 112$이다.
각 외부 노드의 깊이는 $d_1 = d_2 = 4, d_3 = 3, d_4 = d_5 = d_6 = 2$이다.

NoteBase

선지 판단

1번 O: 계산 결과 $WPL = 112$이므로 옳다.
2번 X: 가장 긴 거리는 5가 아니라 4이다. 틀린 표현은 '거리는 5'이고, 올바르게는 '가장 긴 거리는 4'이다.
3번 X: 가장 긴 외부 노드는 q_1, q_2뿐이다. 틀린 표현은 'q_1, q_2, q_3'이고, 올바르게는 'q_1, q_2'이다. q_3의 깊이는 3이다.
4번 X: 가장 짧은 외부 노드는 q_6만이 아니라 q_4, q_5, q_6가 모두 깊이 2로 같다. 틀린 표현은 'q_6이고 그 다음으로 짧은 외부 노드는 q_4와 q_5'이고, 올바르게는 'q_4, q_5, q_6가 모두 가장 짧다'이다.

오답 포인트

②는 허프만 트리의 최대 깊이를 감으로 크게 잡으면 틀리기 쉽다. ③은 q_3가 작은 가중치라서 가장 깊을 것처럼 보이지만 실제로는 q_1, q_2가 먼저 결합된 뒤 다시 한 번 더 묶여서 더 깊어진다. ④는 큰 가중치 q_6가 얕게 놓인다는 사실은 맞지만, q_4와 q_5도 같은 깊이에 놓인다는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

허프만 트리 문제는 작은 가중치 두 개씩 결합한 순서를 끝까지 적고, 각 결합합의 총합으로 WPL을 확인한 뒤, 각 외부 노드가 결합에 포함된 횟수를 세어 깊이를 판단한다.

암기 포인트

허프만 트리에서는 WPL이 결합합의 총합이고, 외부 노드의 깊이는 그 노드가 결합에 포함된 횟수이다.

NoteBase

문 9. 다음과 같이 배열에 저장된 입력 자료들을 퀵 정렬(quick sort)로 오름차순 정렬하고자 한다. 정렬과정에서 단계별 정렬 순서로 나타날 수 없는 것은? (단, 피봇(pivot)은 마지막 레코드로 선택한다)

15	3	1	35	20	4	18	26

① 15, 3, 1, 18, 20, 4, 26, 35

② 15, 3, 1, 4, 20, 18, 26, 35

③ 1, 3, 4, 18, 20, 15, 26, 35

④ 1, 3, 4, 15, 20, 18, 26, 35

정답 2번

출제 포인트

이 문제는 마지막 원소를 피벗으로 잡는 퀵 정렬에서 각 분할이 끝날 때 배열이 어떤 모습이 되는지를 묻는 문제이다.

올바른 순서

1. 피벗 26으로 1차 분할한다. [15, 3, 1, 35, 20, 4, 18, 26] → [15, 3, 1, 18, 20, 4, 26, 35]이다.
2. 왼쪽 부분배열 [15, 3, 1, 18, 20, 4]를 피벗 4로 분할한다. → [1, 3, 4, 18, 20, 15, 26, 35]이다.
3. 오른쪽 부분배열 [18, 20, 15]를 피벗 15로 분할한다. → [1, 3, 4, 15, 20, 18, 26, 35]이다.
4. 부분배열 [20, 18]을 피벗 18로 분할하면 최종적으로 [1, 3, 4, 15, 18, 20, 26, 35]가 된다.

왜 이 순서인가: 각 단계는 현재 부분배열의 마지막 원소를 피벗으로 잡아 그 피벗을 최종 위치에 고정하는 과정이므로, 한 단계가 끝나면 해당 피벗의 왼쪽에는 작은 값만, 오른쪽에는 큰 값만 남는다.

NoteBase

선지별 해설

① O
피벗 26으로 첫 분할을 마친 상태로 볼 수 있다. 35와 18이 교환된 뒤 26이 제자리로 들어가면 [15, 3, 1, 18, 20, 4, 26, 35]가 된다.

② X
틀린 부분: 4가 들어간 뒤에도 15가 4의 왼쪽에 남아 있는 배열이라는 점이 틀리다.
올바르게는: 피벗 4로 분할이 끝나면 4의 왼쪽에는 4보다 작은 1, 3만 있어야 하므로 [1, 3, 4, 18, 20, 15, 26, 35]가 되어야 한다.

③ O
첫 분할 뒤 왼쪽 부분배열을 피벗 4로 다시 분할하면 나올 수 있는 상태이다. 4가 세 번째 위치에 고정되고, 15는 오른쪽 부분배열로 밀린다.

④ O
③ 다음 단계에서 부분배열 [18, 20, 15]를 피벗 15로 분할하면 나올 수 있는 상태이다. 15가 제자리로 이동하고 [20, 18]만 마지막 정렬 대상으로 남는다.

오답 포인트

퀵 정렬의 중간 상태를 볼 때 단순히 값 몇 개가 바뀌었는지만 보면 헷갈리기 쉽다. 한 분할이 끝난 상태라면 피벗은 반드시 최종 위치에 있어야 하고, 그 왼쪽과 오른쪽이 피벗 기준으로 완전히 갈려 있어야 한다는 점을 놓치면 ② 같은 배열을 가능하다고 오판하게 된다.

다시 나오면 이렇게 판단

퀵 정렬 순서 문제는 각 단계의 피벗이 누구인지 먼저 찾고, 그 단계가 끝난 배열에서 피벗 왼쪽에는 작은 값만, 오른쪽에는 큰 값만 있는지를 확인하면 된다. 이 조건을 깨는 보기는 그 단계의 결과가 될 수 없다.

암기 포인트

퀵 정렬의 한 단계가 끝나면 피벗은 제자리, 왼쪽은 작은 값, 오른쪽은 큰 값이다.

NoteBase

문 10. 다음과 같이 중위 표기법(infix notation)으로 되어 있는 수식을 후위 표기법(postfix notation)으로 변환하기 위해 스택을 이용한다. 변환 과정에서 스택에 토큰이 가장 많이 쌓여 있을 때의 스택 내 연산자 토큰 수는?

a*(b-c*d)/e

① 3개
② 4개
③ 5개
④ 6개

정답 2번

출제 포인트

이 문제는 중위 표기식을 후위 표기식으로 바꿀 때 연산자 스택에 토큰이 쌓이는 순서와 최대 적재 시점을 추적할 수 있는지를 묻는다.

올바른 순서

1. a를 출력한 뒤 *를 스택에 넣는다.
2. (를 만나면 스택에 넣고, b를 출력한 뒤 -를 스택에 넣는다.
3. c를 출력한 뒤 *를 만나면 -보다 우선순위가 높으므로 pop하지 않고 스택에 추가로 넣고, d를 출력한다.
4. 이 시점의 스택은 바닥에서부터 *, (, -, * 순서가 되어 저장 토큰 수가 4개로 최대가 된다.
5.)를 만나면 (가 나올 때까지 *와 -를 pop하여 출력하고, 마지막 /를 처리할 때는 기존 *를 먼저 pop한 뒤 /를 넣는다.

왜 이 순서: 괄호 안의 -보다 c*d의 *가 먼저 계산되어야 하므로, 기존 - 위에 *가 한 번 더 쌓이는 순간이 최대 적재 시점이 된다.

선지별 해설

1번 X: 3개로 보면 괄호를 스택에 넣는 토큰에서 빼거나, 안쪽 *가 즉시 pop된다고 잘못 센 것이다. 올바르게는 a*(b-c*d)를 읽는 도중 스택에 *, (, -, *가 함께 있어 최대 4개가 된다.

2번 O: a*(b-c*d)/e를 왼쪽부터 처리하면 d를 읽기까지 스택에 *, (, -, *가 차례로 쌓인다. 따라서 변환 과정에서 스택 내 토큰 수의 최댓값은 4개이다.

3번 X: 5개가 되려면 실제 식에 없는 추가 연산자나 괄호가 하나 더 있어야 한다. 올바르게는)를 만나기 전 최대 적재 상태가 4개에서 멈춘다.

4번 X: 6개는 이 식의 연산자 수와 괄호 수를 넘어서는 값이다. 올바르게는 바깥 *, 왼쪽 괄호, -, 안쪽 *가 동시에 있는 4개가 최대이다.

NoteBase

오답 포인트

괄호를 연산자 스택의 저장 토큰에서 제외하거나, 우선순위가 높은 *가 보이는 즉시 출력된다고 오해하면 3개로 잘못 센다. 실제로는) 또는 더 낮거나 같은 우선순위 연산자를 만날 때까지 스택에 남아 있으므로 최대 적재 시점이 더 늦다.

다시 나오면 이렇게 판단

중위식 변환에서 스택 크기를 묻는 문제는 식을 왼쪽부터 한 토큰씩 따라가며 연산자와 괄호를 만날 때마다 현재 스택 상태를 적어 보면 된다. 최대치는 대개 pop이 일어나기 직전, 기존 연산자 위에 더 높은 우선순위 연산자가 한 번 더 쌓이는 순간에 나온다.

암기 포인트

스택 최대 적재는 pop 직전에 생긴다. 괄호 안에서 낮은 우선순위 연산자 위에 높은 우선순위 연산자가 겹쳐 쌓이면 최대치가 된다.

NoteBase

문 11. 그래프에 대한 설명으로 옳은 것만을 모두 고르면?

ㄱ. 무방향 그래프(undirected graph)를 인접행렬(adjacency matrix)로 표현하면 주대각선을 중심으로 상위 행렬과 하위 행렬이 대칭이 된다.
ㄴ. 사이클이 없고 $n-1$개의 간선(edge)을 갖는 연결 그래프(connected graph)를 트리(tree)라고 한다.
ㄷ. n개의 정점(vertex)을 가진 방향 그래프의 최대 간선 수는 $\frac{n(n+1)}{2}$개이다.
ㄹ. 경로를 구성하는 간선의 수를 길이라고 할 때, 방향 그래프에서 사이클(cycle)의 길이는 항상 3 이상이다.

① ㄱ, ㄴ
② ㄱ, ㄷ
③ ㄴ, ㄷ
④ ㄷ, ㄹ

정답 1번

출제 포인트

이 문제는 그래프의 기본 개념인 인접행렬의 성질, 트리의 정의, 방향 그래프의 최대 간선 수, 사이클 길이의 최소 조건을 구별하는 문제이다.

핵심 개념

무방향 그래프의 인접행렬은 정점 i에서 j로 가는 간선 존재 여부가 j에서 i로 가는 경우와 같아서 주대각선을 기준으로 대칭이다.
트리는 연결그래프이면서 사이클이 없는 그래프이며, 정점이 n개이면 간선 수는 항상 n-1개이다.
단순 방향 그래프에서 자기 루프를 허용하지 않으면 최대 간선 수는 각 정점쌍마다 방향별로 하나씩 둘 수 있으므로 n(n-1)개이다.
방향 그래프의 사이클은 자기 루프를 허용하면 길이 1도 가능하고, 단순 방향 그래프에서도 서로 반대 방향 간선 두 개로 길이 2가 가능하다.

NoteBase

선지별 해설

1번 O: ㄱ, ㄴ만 고른 보기이다. ㄱ은 무방향 그래프의 인접행렬이 대칭이라는 설명이므로 옳다. ㄴ은 연결되어 있고 사이클이 없으며 간선 수가 n-1개인 그래프를 트리라고 한 것으로 옳다.
2번 X: ㄷ이 틀려서 오답이다. n개의 정점을 가진 방향 그래프의 최대 간선 수를 n(n+1)/2라고 한 부분이 틀렸다. 올바르게는 자기 루프를 허용하지 않는 단순 방향 그래프의 최대 간선 수는 n(n-1)개이다.
3번 X: ㄷ이 포함되어 있으므로 오답이다. 틀린 표현은 최대 간선 수를 n(n+1)/2로 본 부분이다. 올바르게는 무방향 단순 그래프의 최대 간선 수가 n(n-1)/2이고, 방향 그래프는 n(n-1)개이다.
4번 X: ㄷ과 ㄹ이 모두 틀려서 오답이다. ㄷ은 방향 그래프의 최대 간선 수 공식을 무방향 그래프와 혼동한 것이다. ㄹ은 방향 그래프에서 사이클 길이가 항상 3 이상이라고 한 부분이 틀렸다. 올바르게는 방향 그래프에서는 길이 2의 사이클도 가능하고, 자기 루프를 허용하면 길이 1의 사이클도 가능하다.

오답 포인트

무방향 그래프의 최대 간선 수 n(n-1)/2와 방향 그래프의 최대 간선 수 n(n-1)을 서로 바꿔 기억하면 ㄷ을 맞는 진술로 오해하기 쉽다. 또 사이클의 최소 길이 3은 단순 무방향 그래프의 성질로 자주 배우므로, 이를 방향 그래프에 그대로 적용하면 ㄹ에 속기 쉽다.

다시 나오면 이렇게 판단

그래프 기본문제는 먼저 무방향인지 방향인지부터 구분한 뒤 인접행렬 대칭 여부, 최대 간선 수 공식, 사이클 최소 길이 조건을 각각 분리해서 판단하면 된다. 특히 최대 간선 수는 무방향이면 n(n-1)/2, 방향이면 n(n-1)로 바로 대조하면 된다.

암기 포인트

무방향은 대칭과 n(n-1)/2, 방향은 n(n-1), 트리는 연결 + 무사이클 + n-1이다.

NoteBase

문 12. 일반 리스트(general list) $L = (e_1, e_2, \cdots, e_n)$에 대해, 함수 $head(L)$은 첫 번째 원소인 e_1을 반환하고, 함수 $tail(L)$은 리스트 $(e_2, e_3, \cdots, e_n)$를 반환한다고 할 때, 다음과 같은 리스트 A에 대한 함수의 수행 결과 중 문자 'c'를 반환하는 것은?

$A = ((a, b), c, d)$

① $head(head(A))$

② $head(tail(A))$

③ $head(tail(head(A)))$

④ $head(tail(tail(A)))$

정답 2번

출제 포인트

이 문제는 일반 리스트에서 `head`는 첫 원소를, `tail`은 첫 원소를 제외한 나머지 리스트를 돌려준다는 정의를 중첩 적용할 수 있는지를 묻는 문제이다.

핵심 개념

`head(L)`는 리스트 `L`의 첫 번째 원소 하나를 반환한다.
`tail(L)`는 리스트 `L`에서 첫 번째 원소를 뺀 나머지 리스트를 반환한다.
주어진 `A=((a, b), c, d)`에서는 첫 원소가 `(a, b)`이고, 그 나머지 리스트가 `(c, d)`라는 점이 핵심이다.

NoteBase

선지별 해설

① X
`head(A)`는 `((a, b), c, d)`의 첫 원소이므로 `(a, b)`이다. 따라서 `head(head(A))`는 `(a, b)`의 첫 원소인 `a`를 반환한다.
틀린 부분은 결과가 `c`가 아니라는 점이다. 올바르게는 `a`를 반환한다.

② O
`tail(A)`는 첫 원소 `(a, b)`를 제외한 나머지 리스트 `(c, d)`이다. 따라서 `head(tail(A))`는 그 첫 원소인 `c`를 반환한다.

③ X
`head(A)`는 `(a, b)`이고, `tail(head(A))`는 `(a, b)`의 첫 원소 `a`를 제외한 나머지 리스트 `(b)`이다. 따라서 `head(tail(head(A)))`는 `b`를 반환한다.
틀린 부분은 결과가 `c`가 아니라는 점이다. 올바르게는 `b`를 반환한다.

④ X
`tail(A)`는 `(c, d)`이고, `tail(tail(A))`는 `(d)`이다. 따라서 `head(tail(tail(A)))`는 `d`를 반환한다.
틀린 부분은 결과가 `c`가 아니라는 점이다. 올바르게는 `d`를 반환한다.

오답 포인트

`head`가 원소 하나를 돌려준다는 점과 `tail`이 리스트를 돌려준다는 점을 섞어서 보면 헷갈리기 쉽다. 특히 `head(A)`의 결과 `(a, b)`를 다시 리스트로 보고 접근해야 하는데, 이를 `A`의 두 번째 원소 `c`와 혼동하면 ①이나 ③을 고르기 쉽다.

다시 나오면 이렇게 판단

중첩 함수 문제는 바깥에서 보지 말고 안쪽 함수부터 순서대로 계산하면 된다. 먼저 현재 대상이 리스트인지 원소인지 확인하고, `head`이면 첫 원소 하나를, `tail`이면 첫 원소를 뺀 나머지 리스트를 적어 내려가면 정답을 안정적으로 판단할 수 있다.

암기 포인트

`head`는 첫 원소, `tail`은 나머지 리스트이다. `A=((a, b), c, d)`이면 `head(A)=(a, b)`, `tail(A)=(c, d)`이다.

NoteBase

문 13. 다음은 단순 연결 리스트(singly linked list)에서 특정 값(value)을 가진 노드의 개수를 반환하는 C 함수이다. ㄱ, ㄴ에 들어갈 내용으로 옳은 것은?

```
struct node {
   int data;
   struct node *link;
};
int countNode(struct node *ptr, int value)
{
   int count = 0;
   while (_____㉠_____) {
      if ( ptr->data == value ) count++;
      _____㉡_____
   }
   return count;
}
```

	㉠	㉡
①	ptr == NULL	ptr->link = ptr;
②	ptr == NULL	ptr = ptr->link;
③	ptr != NULL	ptr->link = ptr;
④	ptr != NULL	ptr = ptr->link;

정답 4번

출제 포인트

이 문제는 단순 연결 리스트를 순회할 때의 종료 조건과 다음 노드로 이동하는 포인터 갱신을 정확히 아는지를 묻는 문제이다.

핵심 개념

연결 리스트 순회는 현재 포인터가 NULL이 아닐 동안 반복하면서, 각 노드를 검사한 뒤 ptr을 다음 노드로 이동시키는 방식으로 진행된다.
NULL은 더 이상 가리킬 노드가 없다는 뜻이고, ptr->link는 다음 노드의 주소이므로 순회 갱신식은 ptr = ptr->link이다.

NoteBase

선지별 해설

① X
- ㉠의 "ptr == NULL"이 틀린 표현이다. ptr이 NULL일 때 반복문에 들어가면 바로 ptr->data를 참조하므로 정상 순회가 되지 않는다.
- ㉡의 "ptr->link = ptr"도 틀린 표현이다. 다음 노드로 이동해야 하는데 현재 노드의 link를 자기 자신으로 바꾸어 리스트 구조를 훼손한다.
- 올바르게는 "ptr != NULL", "ptr = ptr->link"이다.

② X
- ㉠의 "ptr == NULL"이 틀린 표현이다. 순회는 노드가 존재하는 동안 수행되어야 하므로 종료 조건의 반대가 들어가야 한다.
- ㉡의 "ptr = ptr->link"는 맞는 표현이지만, ㉠이 틀려 전체 선지는 오답이다.
- 올바르게는 "ptr != NULL", "ptr = ptr->link"이다.

③ X
- ㉠의 "ptr != NULL"은 맞는 표현이다. 현재 노드가 있을 때만 data를 검사할 수 있다.
- ㉡의 "ptr->link = ptr"이 틀린 표현이다. 이는 포인터를 이동시키는 문장이 아니라 링크 필드를 덮어써 자기 자신을 가리키게 만드는 문장이다.
- 올바르게는 "ptr = ptr->link"이다.

④ O
- ㉠의 "ptr != NULL"은 현재 노드가 존재하는 동안만 반복하겠다는 뜻이므로 올바른 종료 조건이다.
- ㉡의 "ptr = ptr->link"는 현재 노드 처리 후 다음 노드로 이동시키는 순회 갱신식이므로 올바르다.

오답 포인트

이 문항의 함정은 "반복 조건"과 "링크 필드 수정"을 혼동하게 만드는 데 있다. "ptr == NULL"은 종료 시점인데 이를 반복 조건으로 넣으면 널 포인터를 역참조하게 되고, "ptr->link = ptr"은 이동문처럼 보이지만 실제로는 연결 구조 자체를 바꾸는 대입문이다.

다시 나오면 이렇게 판단

연결 리스트 순회 코드를 보면 먼저 "현재 포인터가 유효한가"를 확인하고, 본문 실행 뒤에는 "포인터 변수 자체를 다음 노드 주소로 갱신하는가"를 본다. 조건식이 NULL의 부정인지, 대입 대상이 ptr인지 ptr->link인지 구분하면 빠르게 판단할 수 있다.

암기 포인트

리스트 순회는 "ptr != NULL 동안 검사, ptr = ptr->link로 이동"이다.

NoteBase

문 14. 다음은 Fibonacci 수열을 계산하는 C 함수이다. 함수 fibonacci(4)를 호출하였을 때 화면에 출력되는 숫자의 순서로 옳은 것은?

```
int fibonacci( int n )
{
   printf("%d\n", n);

   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( fibonacci( n - 1 )
                + fibonacci( n - 2 ) );
}
```

① 4, 3, 2, 2, 1, 1, 0, 1, 0

② 4, 3, 2, 1, 0, 2, 1, 1, 0

③ 4, 3, 2, 1, 0, 1, 2, 1, 0

④ 4, 3, 2, 1, 0, 2, 1, 0, 1

정답 3번

출제 포인트

이 문제는 재귀 함수가 호출될 때마다 즉시 출력이 발생한다는 점과, fibonacci(n - 1)을 끝낸 뒤 fibonacci(n - 2)를 호출하는 깊이우선 실행 순서를 묻는 문제이다.

NoteBase

올바른 순서

1. fibonacci(4) 호출 직후 4가 출력된다.
2. 왼쪽 재귀호출 fibonacci(3)으로 내려가 3이 출력된다.
3. 다시 fibonacci(2)로 내려가 2가 출력된 뒤, fibonacci(1)과 fibonacci(0)이 차례로 호출되어 1, 0이 출력된다.
4. fibonacci(3)의 나머지 호출인 fibonacci(1)이 실행되어 1이 출력된다.
5. 마지막으로 fibonacci(4)의 오른쪽 재귀호출인 fibonacci(2)가 실행되어 2, 1, 0이 출력된다.

왜 이 순서인가: return fibonacci(n - 1) + fibonacci(n - 2)이므로 왼쪽 호출 fibonacci(n - 1)의 전체 실행이 끝나야 오른쪽 호출 fibonacci(n - 2)가 시작된다.

선지별 해설

① X: 두 번째 2를 너무 일찍 배치하였다. 두 번째 2는 fibonacci(4)의 오른쪽 호출 fibonacci(2)가 시작될 때 출력되므로, fibonacci(3) 안의 마지막 fibonacci(1) 출력 뒤에 나와야 한다. 틀린 표현은 '4, 3, 2, 2, ...'이고, 올바르게는 '4, 3, 2, 1, 0, 1, 2, 1, 0'이다.

② X: 첫 번째 fibonacci(2)를 끝낸 뒤에는 fibonacci(3)의 두 번째 하위 호출인 fibonacci(1)이 먼저 실행된다. 따라서 0 다음에 바로 2가 올 수 없다. 틀린 표현은 '..., 1, 0, 2, ...'이고, 올바르게는 '..., 1, 0, 1, 2, ...'이다.

③ O: 호출될 때마다 먼저 현재 n을 출력하고, 왼쪽 재귀호출을 모두 끝낸 뒤 오른쪽 재귀호출로 넘어가므로 출력 순서는 4, 3, 2, 1, 0, 1, 2, 1, 0이 된다.

④ X: fibonacci(3) 내부의 마지막 fibonacci(1) 출력을 맨 뒤로 보내어 호출 트리를 거꾸로 읽었다. 틀린 표현은 '..., 1, 0, 2, 1, 0, 1'이고, 올바르게는 '..., 1, 0, 1, 2, 1, 0'이다.

오답 포인트

printf가 반환 직전에 실행된다고 착각하거나, 호출 트리를 좌우 대칭으로 훑듯이 읽으면 두 번째 2나 마지막 1의 위치를 잘못 놓치기 쉽다. 이 문제의 함정은 출력 시점이 반환 시점이 아니라 함수 진입 직후라는 점이다.

다시 나오면 이렇게 판단

재귀 호출 출력 순서 문제는 코드의 실행 위치를 먼저 본다. printf가 재귀 호출 앞에 있으면 전위 순회처럼 현재 값부터 출력되고, return 식의 왼쪽 호출이 모두 끝난 뒤 오른쪽 호출이 시작된다고 판단하면 된다.

암기 포인트

재귀 출력은 함수 몸체의 위에서 아래로 실행된다. 이 코드에서는 '출력 후 왼쪽 호출 전체, 그다음 오른쪽 호출 전체'의 순서가 유지된다.

NoteBase

문 15. 다음은 이진 탐색을 수행하는 C 함수이다. 이 함수를 사용하여 0부터 20까지의 정수가 차례대로 저장되어 있는 배열 b에 대한 이진 탐색을 수행할 때, key 값과 a[middle] 값을 비교하는 횟수가 다른 것은? (단, 배열의 색인은 0부터 시작한다)

```
int binarySearch(int *a, int key, int low,
                 int high)
{
   int middle;
   while (low <= high) {
      middle = (low + high) / 2;
      if (key == a[middle])
         return middle;
      else if (key > a[middle])
         low = middle + 1;
      else
         high = middle - 1;
   }
   return -1;
}
```

① binarySearch(b, 4, 1, 20)

② binarySearch(b, 7, 1, 20)

③ binarySearch(b, 9, 1, 20)

④ binarySearch(b, 14, 1, 20)

정답 2번

출제 포인트

이 문제는 이진 탐색에서 low와 high가 갱신될 때 middle이 어떤 순서로 바뀌는지와 key와 a[middle]의 비교 횟수를 정확히 추적할 수 있는지를 묻는다.

NoteBase

올바른 순서

각 호출의 비교 순서는 다음과 같다.
1번: 10 -> 5 -> 2 -> 3 -> 4로 진행되어 5회 비교이다.
2번: 10 -> 5 -> 7로 진행되어 3회 비교이다.
3번: 10 -> 5 -> 7 -> 8 -> 9로 진행되어 5회 비교이다.
4번: 10 -> 15 -> 12 -> 13 -> 14로 진행되어 5회 비교이다.
비교 횟수는 while 반복마다 새 middle을 계산한 뒤 key와 a[middle]를 대조할 때 1회씩 늘어난다.

선지별 해설

1번: X. middle 값이 10, 5, 2, 3, 4 순서로 바뀌므로 key=4는 5번째 비교에서 찾는다. 틀린 표현은 '다른 것'이다. 올바르게는 1번은 3번, 4번과 같은 5회 비교이므로 다른 것이 아니다.
2번: O. middle 값이 10, 5, 7 순서로 바뀌어 3번째 비교에서 찾는다. 2번만 나머지와 비교 횟수가 다르다.
3번: X. middle 값이 10, 5, 7, 8, 9 순서로 바뀌므로 5회 비교이다. 틀린 표현은 '다른 것'이다. 올바르게는 3번은 1번, 4번과 같은 5회 비교이므로 다른 것이 아니다.
4번: X. middle 값이 10, 15, 12, 13, 14 순서로 바뀌므로 5회 비교이다. 틀린 표현은 '다른 것'이다. 올바르게는 4번은 1번, 3번과 같은 5회 비교이므로 다른 것이 아니다.

오답 포인트

배열에 0부터 20까지 저장되어 있어도 실제 호출 구간은 low=1, high=20이다. 따라서 첫 비교는 a[10]과 이루어진다. low를 0부터로 잘못 두거나, low/high를 갱신한 뒤 새 middle을 다시 계산하는 과정을 빼먹으면 1번, 3번, 4번의 비교 횟수를 잘못 세기 쉽다.

다시 나오면 이렇게 판단

이진 탐색 비교 횟수 문제는 시작 구간을 먼저 적고, 각 단계마다 middle 인덱스와 그 값, 다음 low/high를 한 줄씩 기록하면 된다. 각 호출의 탐색 경로 길이를 비교하면 다른 선지를 바로 가릴 수 있다.

암기 포인트

이진 탐색의 비교 횟수는 key의 실제 값보다 middle 경로 길이로 결정되며, middle을 다시 계산할 때마다 비교 1회가 추가된다.

NoteBase

문 16. 시간 복잡도 함수에 대한 점근 표기법으로 옳은 것만을 모두 고르면?

ㄱ. $n^{2n} + 6 \cdot 2^n = O(n^{2n})$
ㄴ. $n^2/\log n = \Theta(n^2)$
ㄷ. $n^3 2^n + 6n^2 3^n = O(n^2 2^n)$
ㄹ. $6n^3 + \log n = O(n^3)$

① ㄱ, ㄷ
② ㄱ, ㄹ
③ ㄴ, ㄷ
④ ㄴ, ㄹ

정답 2번

출제 포인트

이 문제는 점근 표기법에서 O와 Θ의 의미를 구분하고, 합의 지배항과 지수함수의 성장률을 비교할 수 있는지를 묻는다.

조건 정리

판단 대상은 ㄱ~ㄹ 네 문장이다.
각 문장이 참인지 거짓인지를 먼저 판정한 뒤, 그 결과와 일치하는 묶음 보기를 고르면 된다.

사용할 기준

$f(n) = O(g(n))$는 충분히 큰 n에서 $f(n)$가 $g(n)$의 상수배 이하이면 성립한다.
$f(n) = \Theta(g(n))$는 $f(n)$와 $g(n)$가 같은 차수로 자라서 상한과 하한이 모두 성립할 때만 성립한다.
합은 더 빨리 자라는 항이 지배하고, 지수항끼리는 밑이 큰 쪽이 다항식 차이보다 더 크게 작용한다.

판단 과정

ㄱ: n^{2n}이 2^n보다 압도적으로 크므로 $n^{2n} + 6 \cdot 2^n$의 차수는 n^{2n}이다. 따라서 $O(n^{2n})$는 참이다.
ㄴ: $\frac{n^2}{\log n}$을 n^2와 비교하면 비율이 $\frac{1}{\log n} \to 0$이므로 같은 차수가 아니다. 따라서 $\Theta(n^2)$는 거짓이다.
ㄷ: $n^3 2^n$만 보아도 $\frac{n^3 2^n}{n^2 2^n} = n \to \infty$이므로 $O(n^2 2^n)$가 아니다. 또한 $6n^2 3^n$은 $\frac{6n^2 3^n}{n^2 2^n} = 6\left(\frac{3}{2}\right)^n \to \infty$이므로 더더욱 성립하지 않는다. 따라서 거짓이다.
ㄹ: $6n^3 + \log n$은 n^3이 지배하므로 $O(n^3)$가 성립한다. 따라서 참이다.

NoteBase

선지별 해설

1번 X: 'ㄱ, ㄷ'이라고 묶었지만 ㄷ은 거짓이다. $n^3 2^n$도 $n^2 2^n$보다 크고, 3^n은 2^n보다 더 빨리 커지므로 올바르게는 'ㄱ, ㄹ'이다.
2번 O: ㄱ과 ㄹ만 참이다. ㄱ은 지배항이 n^{2n}이고, ㄹ은 지배항이 n^3이므로 맞는 묶음이다.
3번 X: 'ㄴ, ㄷ'이라고 묶었지만 ㄴ과 ㄷ은 모두 거짓이다. ㄴ은 $O(n^2)$일 수는 있어도 $\Theta(n^2)$는 아니고, ㄷ도 $O(n^2 2^n)$가 아니다. 올바르게는 'ㄱ, ㄹ'이다.
4번 X: 'ㄴ, ㄹ'이라고 묶었지만 ㄴ이 거짓이다. $\frac{n^2}{\log n}$은 n^2보다 $\log n$만큼 작으므로 같은 차수로 볼 수 없고, 올바르게는 'ㄱ, ㄹ'이다.

오답 포인트

O와 Θ를 같은 뜻처럼 보는 실수가 잦다. 또한 지수항에서는 밑의 크기 차이인 3^n 대 2^n이 다항식 인수 n 하나보다 훨씬 크게 작용한다는 점을 놓치면 ㄷ을 참으로 오해하기 쉽다.

다시 나오면 이렇게 판단

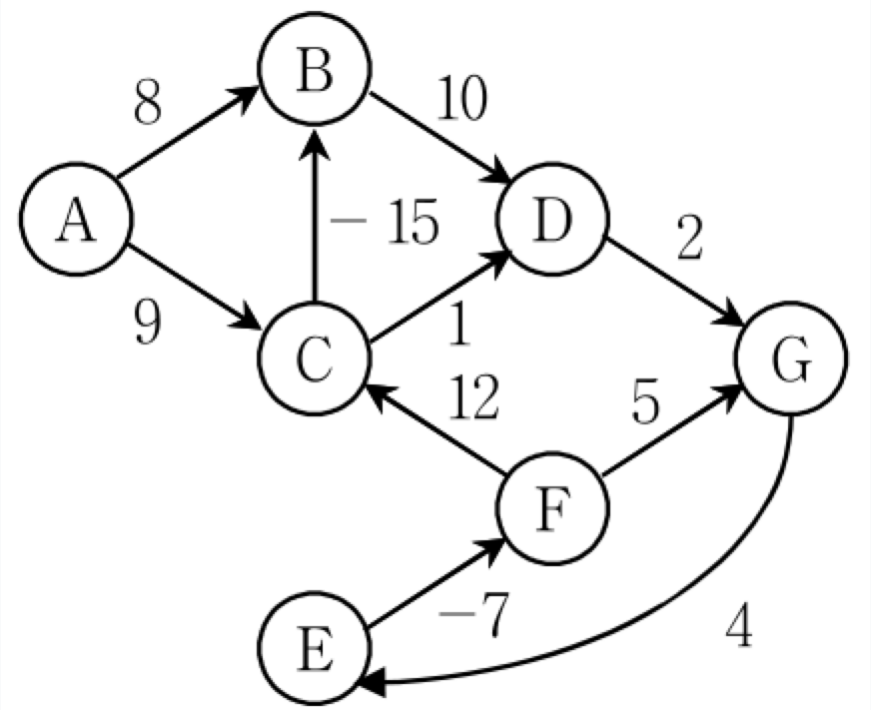
점근 표기 문제는 먼저 각 식의 지배항을 고르고, 그다음 O는 상한만 보는지, Θ는 같은 차수까지 요구하는지를 구분해 판단한다. 지수항 비교에서는 밑이 큰 쪽이 결국 이긴다고 보면 된다.

암기 포인트

합의 차수는 지배항이 정하고, O는 상한, Θ는 같은 차수, 지수 비교는 밑이 큰 쪽이 우세하다.

NoteBase

문 17. 다음 그래프에는 음의 가중치가 존재한다. 이 그래프에서, 노드 A에서 노드 F로의 최단 경로를 구하고자 할 때, 최소 비용은?



① 3

② 5

③ 7

④ 9

정답 1번

출제 포인트

이 문제는 방향 그래프에서 간선의 방향과 음의 가중치를 함께 읽어 A에서 F까지의 누적 비용 최솟값을 찾는 문제이다.

NoteBase

선지별 해설

1번(O): A→C→B→D→G→E→F의 비용은 9 + (-15) + 10 + 2 + 4 + (-7) = 3이므로 최소 비용은 3이다.
2번(X): 최소 비용을 5로 본 판단은 틀리다. C→B의 -15와 E→F의 -7을 반영하면 비용이 더 내려간다. 올바르게는 최소 비용이 3이다.
3번(X): 최소 비용을 7로 본 판단은 틀리다. A→C→D→G→E→F의 비용은 9이므로 7은 이 그래프의 최단 경로 비용이 아니다. 올바르게는 최소 비용이 3이다.
4번(X): 최소 비용을 9로 본 판단은 틀리다. 9는 A→C→D→G→E→F를 택했을 때의 비용일 뿐 최단 경로 비용이 아니다. 올바르게는 최소 비용이 3이다.

오답 포인트

C→B의 음의 가중치 -15와 E→F의 음의 가중치 -7을 빼먹거나, 화살표 방향을 무시하고 눈에 보이는 짧은 경로만 고르면 9 같은 값을 최단 경로로 오인하게 된다.

다시 나오면 이렇게 판단

시작점에서 도착점까지 도달 가능한 경로를 화살표 방향대로 적고, 각 경로의 가중치 합을 끝까지 계산한 뒤 가장 작은 값을 고르면 된다.

암기 포인트

방향 그래프 최단 경로는 화살표를 지키고, 음수 간선은 반드시 총합에 반영한다.

NoteBase

문 18. 다음은 단순 연결 리스트(singly linked list) L에서 리스트의 원소를 역순으로 바꾸는 함수이다. 함수에서 ㉠ 부분에 들어갈 프로그램 코드(code)로 옳은 것은?

```
typedef struct node{
   struct node *link;
   char data;
} listNode;
typedef struct{
   listNode *head;
} linkedList;

void reverse(linkedList *L)
{
   listNode *p, *q, *r;
   p = L->head;
   q = r = NULL;
   while ( p != NULL)  {
      r = q;
      q = p;
      p = p->link;
      ㉠
   }
   L->head = q;
}
```

① q = q->link;

② q->link = r;

③ p->link = q;

④ r->link = q;

정답 2번

출제 포인트

이 문제는 단순 연결 리스트를 역순으로 만들 때 현재 노드와 이전 노드의 연결 방향을 어느 시점에 어떻게 바꾸는지를 묻는다.

NoteBase

올바른 순서

1. r = q; 이전 노드를 임시로 보관한다.
2. q = p; 현재 노드를 새 기준 노드로 잡는다.
3. p = p->link; 다음 반복에 처리할 다음 노드로 이동한다.
4. q->link = r; 현재 노드가 이전 노드를 가리키게 하여 링크 방향을 뒤집는다.

왜 이 순서: 현재 노드 q의 링크를 뒤집을 때는 먼저 원래 다음 노드를 p에 저장해 두어야 나머지 리스트를 잃지 않는다.

선지별 해설

1번 X: q = q->link; 는 q를 현재 노드의 다음으로 이동시키는 문장인데, 이 시점의 q는 이미 현재 노드 p를 가리키도록 바뀌었으므로 역순 연결을 만들지 못한다. 틀린 표현은 링크 방향 변경 없이 q만 이동시키는 점이다. 올바르게는 q->link = r; 로 현재 노드가 이전 노드를 가리키게 해야 한다.

2번 O: q->link = r; 는 현재 노드 q의 링크를 직전 노드 r로 바꾸는 문장이다. 이 한 줄이 있어야 처리한 구간의 연결 방향이 뒤집히며, 반복이 끝나면 q가 새 머리 노드가 된다.

3번 X: p->link = q; 는 이미 p가 다음 노드로 이동한 뒤 그 다음 처리 대상의 링크를 바꾸는 문장이다. 틀린 표현은 현재 노드가 아니라 아직 처리하지 않은 노드의 링크를 수정하는 점이다. 올바르게는 현재 노드인 q의 링크를 이전 노드 r로 연결해야 한다.

4번 X: r->link = q; 는 이전 노드의 링크를 현재 노드로 향하게 만들어 역순이 아니라 원래 방향을 유지하는 쪽에 가깝다. 첫 반복에서는 r이 NULL이므로 성립할 수도 없다. 올바르게는 q->link = r; 이다.

오답 포인트

q와 r의 역할을 혼동하면 링크를 뒤집어야 할 대상이 현재 노드인지 이전 노드인지가 바뀐다. 특히 p = p->link; 뒤에는 p가 이미 다음 노드라는 점을 놓치면 3번을 고르기 쉽다.

다시 나오면 이렇게 판단

연결 리스트 역순 코드는 이전 노드 보관 -> 현재 노드 지정 -> 다음 노드 백업 -> 현재 노드의 링크 반전 순서로 본다. 링크 방향을 실제로 바꾸는 문장이 어느 포인터의 link를 수정하는지만 확인하면 정답을 고를 수 있다.

암기 포인트

역순 연결의 핵심은 현재 노드 q가 이전 노드 r를 가리키게 만드는 q->link = r; 이다.

NoteBase

문 19. 주어진 정수 배열 {26, 13, 77, 61, 35, 11, 8, 48, 15, 19}에 대한 합병 정렬(merge sort)의 순환 알고리즘(recursive algorithm)을 다음과 같이 구현하였다. 이 프로그램의 수행 과정에서 형성되는 배열의 순서로 옳지 않은 것은?

```
#include <stdio.h>
#define MAX 10
void merge( int list[], int temp[], int left,
               int middle, int right)
{
    int i=left, j=middle+1, k=left, t;
    for( ; i <= middle && j <= right ;  ) {
         if( list[i] <= list[j] )
            temp[k] = list[i++];
         else
            temp[k] = list[j++];
         k += 1;
     }
     if( i > middle ) {
        for( t = j ; t <= right ;)
            temp[k++] = list[t++];
     } else {
        for( t = i ; t <= middle ;)
            temp[k++] = list[t++];
     }
     for( t = left ; t <= right ; t++ )
         list[t] = temp[t];
}

void mergeSort( int list[], int temp[], int left,
                  int right)
{
     int middle = 0;
     if( left < right ) {
         middle = (left+right)/2;
         mergeSort(list, temp, left, middle);
         mergeSort(list, temp, middle+1, right);

         merge(list, temp, left, middle, right);
     }
}

int main(){
     int source[10]={26, 13, 77, 61, 35, 11, 8,
                     48, 15, 19};
     int temp[10] = {0x0,};

     mergeSort(source, temp, 0, 9);
}
```

① 13, 26, 35, 61, 77, 11, 8, 48, 15, 19

② 13, 26, 77, 61, 35, 11, 8, 48, 15, 19

③ 13, 26, 35, 61, 77, 8, 11, 15, 19, 48

④ 13, 26, 35, 61, 77, 8, 11, 15, 48, 19

정답 4번

출제 포인트

이 문제는 재귀적 병합 정렬에서 부분배열이 병합되는 순서와 각 병합 직후의 전체 배열 상태를 추적할 수 있는지를 묻는다.

NoteBase

올바른 순서

전체는 mergeSort(0,9)로 시작하며 middle=(0+9)/2=4이므로 인덱스 0~4(왼쪽 절반)를 먼저 처리한 뒤 인덱스 5~9(오른쪽 절반)를 처리한다.
1. 왼쪽 절반 mergeSort(0,4)는 middle=(0+4)/2=2로 다시 나뉘고, 그 안의 mergeSort(0,2)는 middle=1로 나뉘어 가장 안쪽의 인덱스 0~1의 26, 13이 먼저 병합되어 배열 앞부분이 13, 26으로 바뀐다.
2. 인덱스 0~2의 13, 26과 77이 병합되지만 13≤26≤77이라 이미 순서가 맞아 전체 배열 모습은 13, 26, 77, 61, 35, 11, 8, 48, 15, 19로 유지된다.
3. 인덱스 3~4의 61, 35가 병합되어 35, 61이 되고, 이어서 인덱스 0~4가 병합되어 왼쪽 절반이 13, 26, 35, 61, 77이 된다.
4. 오른쪽 절반 mergeSort(5,9)는 middle=(5+9)/2=7이므로 인덱스 5~7과 인덱스 8~9로 나뉜다. 인덱스 5~7은 11, 8, 48이 8, 11, 48로 정렬되고(그 안에서 인덱스 5~6의 11, 8이 먼저 8, 11이 된다), 인덱스 8~9의 15, 19는 이미 정렬되어 있어 그대로 유지된다.
5. 인덱스 5~9가 병합되면 8, 11, 48과 15, 19가 합쳐져 8, 11, 15, 19, 48이 되고, 마지막으로 인덱스 0~9 전체가 병합되어 최종 정렬된다.
왜 이 순서인가: 병합 정렬은 더 작은 구간이 먼저 정렬되어 있어야 그 상위 구간을 병합할 수 있으므로, 하위 부분배열의 병합 결과가 상위 병합의 입력이 된다.

선지별 해설

1번: O(수행 중 실제로 나타나는 배열). 인덱스 3~4가 35, 61로 정렬된 뒤 인덱스 0~4를 병합하면 왼쪽 절반이 13, 26, 35, 61, 77이 되어, 오른쪽 절반이 아직 그대로인 13, 26, 35, 61, 77, 11, 8, 48, 15, 19가 실제로 나타난다.
2번: O(수행 중 실제로 나타나는 배열). 인덱스 0~1의 26, 13을 병합한 직후 배열은 13, 26, 77, 61, 35, 11, 8, 48, 15, 19가 된다. 이어 인덱스 0~2를 병합할 때 13, 26에 77을 합치는데 13≤26≤77이라 순서가 바뀌지 않아 같은 배열 상태가 유지된다.
3번: O(수행 중 실제로 나타나는 배열). 오른쪽 절반 인덱스 5~9가 병합되면 8, 11, 15, 19, 48이 되므로, 왼쪽 절반이 정렬된 상태와 합쳐져 전체 배열은 13, 26, 35, 61, 77, 8, 11, 15, 19, 48이 된다.
4번: X(수행 중 나타나지 않는 배열). 오른쪽 절반 mergeSort(5,9)는 middle=7로 나뉘어 인덱스 5~7이 8, 11, 48로, 인덱스 8~9가 15, 19로 정렬된 뒤 병합되어 8, 11, 15, 19, 48이 된다. 따라서 오른쪽 절반이 완성되는 시점의 전체 배열은 13, 26, 35, 61, 77, 8, 11, 15, 19, 48이다. 선지처럼 끝부분이 15, 48, 19로 남아 48과 19가 뒤바뀐 배열은 병합 과정 어디에서도 나타나지 않는다.

오답 포인트

④는 오른쪽 절반이 8, 11, 15, 19, 48로 정렬되는 마지막 병합을 절반만 반영한 함정이다. 15와 19가 48보다 앞서야 한다는 점은 맞지만 병합이 끝나면 48이 맨 뒤로 이동하므로, 선지 ④처럼 끝부분이 15, 48, 19로 남아 48과 19가 뒤바뀐 배열은 어떤 병합 직후에도 나타나지 않는다.

NoteBase

다시 나오면 이렇게 판단

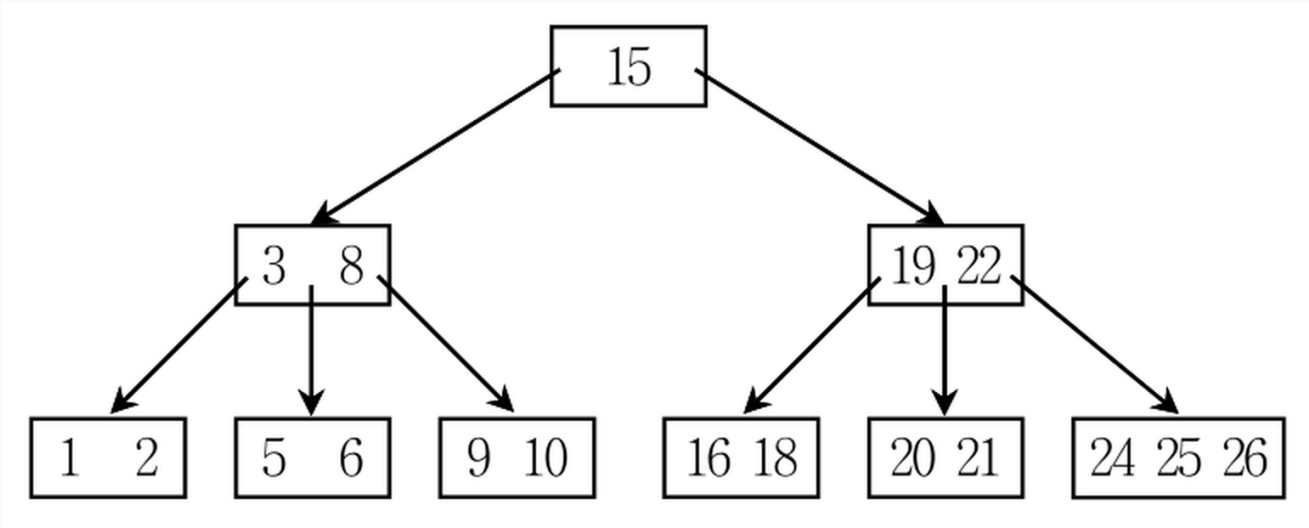
병합 정렬 추적 문제는 재귀 호출 순서를 따라 가장 작은 구간부터 병합 결과를 적어 나가면 된다. 각 단계에서 바뀌는 구간의 시작과 끝을 middle=(left+right)/2로 먼저 찾고, 그 구간만 정렬한 뒤 전체 배열에 다시 끼워 넣으면 실제로 나올 수 있는 배열인지 판단할 수 있다.

암기 포인트

병합 정렬의 중간 배열은 하위 구간 정렬 완료 후 상위 구간 병합 순서로 나타나며, 한 번의 병합이 끝난 구간 내부에는 다시 역전쌍이 남지 않는다.

NoteBase

문 20. 다음은 노드의 차수가 5인 B-트리의 현재 상태를 표현한 것이다. 현 상태에서 값 9를 삭제하였을 때, 결과 트리에 존재하는 노드 수를 종류별로 옳게 나타낸 것은? (단, n-노드란 $(n-1)$개의 키를 저장한 노드를 의미한다)



① 2-노드: 2, 3-노드: 6, 4-노드: 1, 5-노드: 0

② 2-노드: 2, 3-노드: 4, 4-노드: 1, 5-노드: 1

③ 2-노드: 0, 3-노드: 2, 4-노드: 3, 5-노드: 1

④ 2-노드: 0, 3-노드: 3, 4-노드: 1, 5-노드: 2

정답 4번

출제 포인트

이 문제는 차수(order) 5인 B-트리에서 삭제 후 최소 키 수 부족이 생기면 형제 노드와 부모 키를 이용해 병합하고, 최종 상태에서 2-노드부터 5-노드까지의 개수를 정확히 세는지를 묻는다. 차수 5 B-트리에서 각 노드는 최대 4개 키와 5개 자식을 가지며, 비루트 노드의 최소 키 수는 $\lceil 5/2 \rceil - 1 = 2$개이다. 따라서 삭제로 키가 1개만 남으면 언더플로가 되어 빌리기 또는 병합이 필요하다.

NoteBase

선지별 해설

① X: 선지 ①은 2-노드 2개, 3-노드 6개, 4-노드 1개, 5-노드 0개라고 제시한다. 리프 노드 [9 10]에서 9를 삭제하면 [10]이 되어 비루트 최소 키 수 2개를 만족하지 못한다. 왼쪽 형제 [5 6]도 키가 정확히 최소치인 2개뿐이라 빌릴 수 없고, 두 노드 사이의 부모 구분 키 8을 내려 [5 6]과 [10]을 합쳐 [5 6 8 10]으로 병합한다. 이때 부모 [3 8]은 키 8을 잃어 [3]이 되고, 키가 1개뿐이라 다시 언더플로가 된다. 부모 [3]의 형제 [19 22]도 최소치인 2개뿐이라 빌릴 수 없으므로, 두 노드 사이의 부모 구분 키(루트의 키) 15를 내려 [3]과 [19 22] 사이에 끼워 [3 15 19 22]로 병합한다. 이 과정에서 루트의 유일한 키 15가 내려가 루트가 비게 되므로, 병합된 [3 15 19 22]가 새 루트가 되고 트리 높이가 1 감소한다. 최종 개수는 2-노드 0개, 3-노드 3개, 4-노드 1개, 5-노드 2개이므로, ①의 2-노드 2개·3-노드 6개·5-노드 0개는 모두 틀리다.
② X: 선지 ②는 2-노드 2개, 3-노드 4개, 4-노드 1개, 5-노드 1개라고 제시한다. 위와 같은 병합을 끝내면 남는 노드는 새 루트 [3 15 19 22], 리프 [1 2], [5 6 8 10], [16 18], [20 21], [24 25 26]이다. 따라서 2-노드는 존재하지 않고(0개), 3-노드는 [1 2], [16 18], [20 21]의 3개이며, 5-노드는 새 루트 [3 15 19 22]와 [5 6 8 10]의 2개이다. ②의 2-노드 2개·3-노드 4개·5-노드 1개는 실제 값(0개, 3개, 2개)과 다르다.
③ X: 선지 ③은 2-노드 0개, 3-노드 2개, 4-노드 3개, 5-노드 1개라고 제시한다. 실제로 4-노드는 [24 25 26] 하나뿐(1개)이고, 5-노드는 새 루트 [3 15 19 22]와 병합된 리프 [5 6 8 10]의 2개이며, 3-노드는 [1 2], [16 18], [20 21]의 3개이다. ③의 3-노드 2개·4-노드 3개·5-노드 1개는 실제 값(3개, 1개, 2개)과 다르다.
④ O: 선지 ④는 2-노드 0개, 3-노드 3개, 4-노드 1개, 5-노드 2개라고 제시한다. 9 삭제 후 [9 10]이 [10]이 되어 언더플로가 발생하고, 형제 [5 6]도 최소치라 빌릴 수 없어 부모 키 8과 함께 [5 6 8 10]으로 병합한다. 이어서 키 1개만 남은 내부 노드 [3]의 언더플로는 형제 [19 22]와 부모 구분 키 15를 합쳐 [3 15 19 22]로 병합하며, 이때 루트가 비어 이 노드가 새 루트가 되고 높이가 1 줄어든다. 최종 트리는 루트 [3 15 19 22], 리프 [1 2], [5 6 8 10], [16 18], [20 21], [24 25 26]이므로 2-노드 0개, 3-노드 3개, 4-노드 1개, 5-노드 2개가 맞다.

오답 포인트

리프에서 한 번 병합한 중간 상태만 보고 개수를 세면 오답이 된다. 차수 5의 비루트 노드는 최소 키 2개를 가져야 하므로, [3]처럼 키 1개만 남은 내부 노드도 다시 언더플로 처리 대상이라는 점이 함정이다. 특히 병합이 위로 전파되어 루트의 유일한 키까지 내려가면 루트가 교체되고 트리 높이가 줄어드는데, 이 마지막 단계를 놓치면 노드 개수가 어긋난다.

다시 나오면 이렇게 판단

차수 5 B-트리 삭제 문제는 삭제한 노드가 최소 키 수 2개를 유지하는지부터 본다. 유지하지 못하면 형제가 여유 키를 갖는지 확인하고, 여유가 없으면 부모의 구분 키를 끼워 병합한다. 병합 뒤 부모도 최소 키 수를 못 맞추면 같은 검사를 위로 반복하며, 루트의 유일한 키까지 내려가 루트가 비면 병합 노드가 새 루트가 되어 높이가 감소한다. 마지막으로 완성된 트리에서 노드 종류를 센다.

암기 포인트

차수 5 B-트리의 비루트 최소 키 수는 $\lceil 5/2 \rceil - 1 = 2$개이며, 삭제 후 1개가 되면 빌리기 또는 병합이 필요하다.

NoteBase