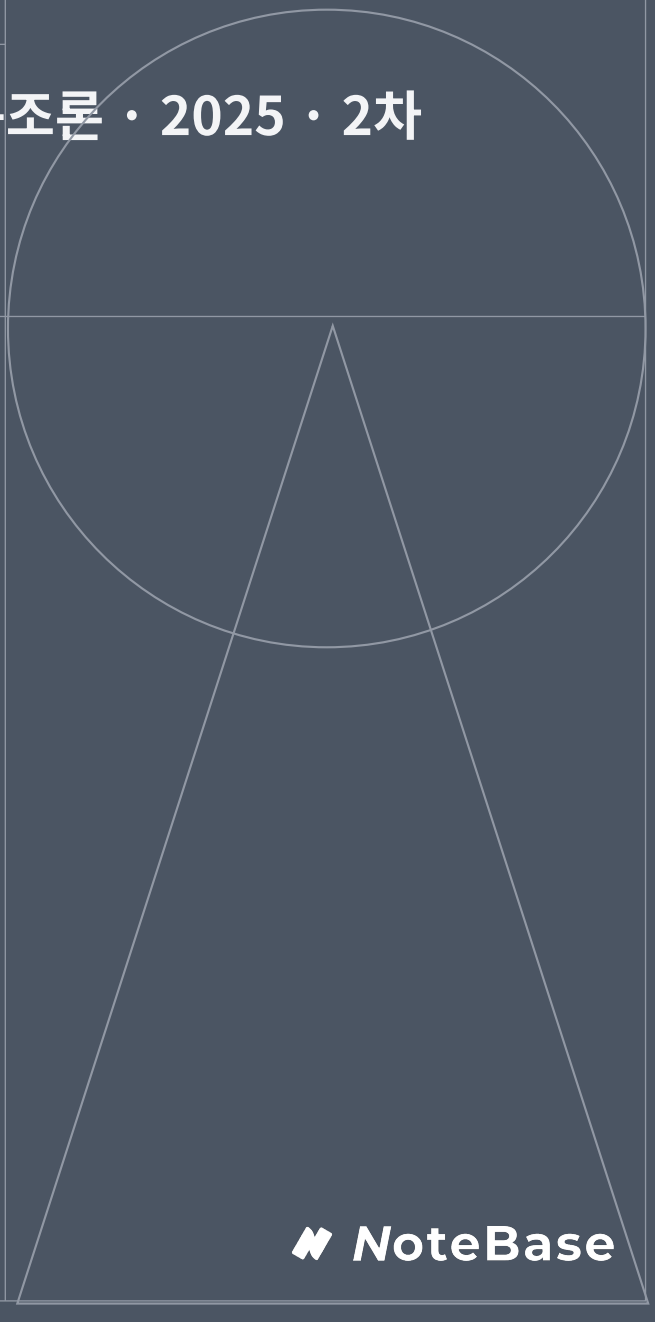
기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2025 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2025 · 2차

공무원/국가직/7급 · 자료구조론 · 2025 · 2차

문 1. 55개 간선(edge)을 갖는 무방향 완전 그래프의 정점(vertex) 개수는?

① 8

② 9

③ 10

④ 11

정답 4번

출제 포인트

이 문제는 무방향 완전 그래프의 간선 수 공식을 이용해 정점 수를 역산할 수 있는지를 묻는 문제이다.

조건 정리

주어진 값: 간선 수는 55개이다.
구해야 할 값: 정점 수이다.
주의할 조건: 무방향 완전 그래프이므로 모든 정점 쌍이 간선 1개로 연결된다.

사용할 공식

무방향 완전 그래프 K_n의 간선 수는 $\frac{n(n-1)}{2}$이다.
이 공식을 쓰는 이유: 완전 그래프에서는 정점 n개 중 서로 다른 두 정점을 고르는 모든 경우가 곧 간선의 수가 되기 때문이다.

계산 과정

간선 수가 55이므로 $\frac{n(n-1)}{2} = 55$이다.
양변에 2를 곱하면 $n(n-1) = 110$이다.
이를 만족하는 연속한 두 자연수는 10과 11이므로 $n = 11$이다.

NoteBase

선지 판단

① X: 8개의 정점이면 간선 수는 $\frac{8\times 7}{2}=28$이다. 55가 아니므로 틀리다. 틀린 표현은 정점 수 8이라는 판단이며, 올바르게는 정점 수가 11이어야 한다.
② X: 9개의 정점이면 간선 수는 $\frac{9\times 8}{2}=36$이다. 55가 아니므로 틀리다. 틀린 표현은 정점 수 9라는 판단이며, 올바르게는 정점 수가 11이어야 한다.
③ X: 10개의 정점이면 간선 수는 $\frac{10\times 9}{2}=45$이다. 55가 아니므로 틀리다. 틀린 표현은 정점 수 10이라는 판단이며, 올바르게는 정점 수가 11이어야 한다.
④ O: 11개의 정점이면 간선 수는 $\frac{11\times 10}{2}=55$이다. 주어진 조건과 일치하므로 옳다.

오답 포인트

완전 그래프의 간선 수를 $n(n-1)$로만 계산하고 2로 나누지 않으면 오답이 된다. 무방향 그래프에서는 정점 쌍 (A,B)와 (B,A)를 같은 간선으로 보므로 반드시 2로 나누어야 한다.

다시 나오면 이렇게 판단

완전 그래프 문제는 먼저 $\frac{n(n-1)}{2}$ 공식을 떠올리고, 주어진 간선 수를 대입해 연속한 두 자연수의 곱으로 바꿔 정점 수를 판단한다.

암기 포인트

무방향 완전 그래프의 간선 수는 정점 쌍의 개수이므로 $\frac{n(n-1)}{2}$이다.

NoteBase

문 2. 초기 데이터에 대하여 오름차순 정렬을 2번 연산 수행한 후의 결과가 다음과 같을 때, 사용한 정렬 방법은?

◯ 초기 데이터: 21, 3, 18, 33, 9, 15
◯ 2번 연산 수행 결과: 3, 18, 21, 33, 9, 15

① 삽입 정렬
② 선택 정렬
③ 버블 정렬
④ 기수 정렬

정답 1번

출제 포인트

이 문제는 정렬 과정의 중간 결과를 보고 어떤 정렬이 어떤 방식으로 진행되는지 식별할 수 있는지를 묻는다.

올바른 순서

1. 첫 번째 원소 21을 정렬 완료 구간으로 둔다.
2. 두 번째 원소 3을 꺼내 21 앞에 삽입하여 3, 21, 18, 33, 9, 15가 된다.
3. 세 번째 원소 18을 꺼내 정렬 구간 3, 21 사이에 삽입하여 3, 18, 21, 33, 9, 15가 된다.
4. 두 번 연산 후 결과가 제시 배열과 정확히 일치한다.

왜 이 순서: 삽입 정렬은 앞 단계에서 만들어진 정렬 구간을 다음 단계의 입력으로 사용하므로, 두 번째 삽입은 첫 번째 삽입 결과가 전제이다.

NoteBase

선지별 해설

① O: 삽입 정렬은 앞쪽의 정렬된 부분에 다음 원소를 알맞은 위치로 끼워 넣는 방식이다. 1회전 후 3, 21, 18, 33, 9, 15가 되고, 2회전 후 18이 21 앞에 삽입되어 3, 18, 21, 33, 9, 15가 되므로 제시 결과와 일치한다.
② X: 선택 정렬은 각 회전마다 남은 구간의 최솟값을 앞자리로 보낸다. 1회전 후에는 3이 맨 앞으로 오지만 2회전 후에는 9가 두 번째 자리에 와서 3, 9, 18, 33, 21, 15가 되어야 한다. 틀린 표현은 '제시 결과가 선택 정렬의 2회 연산 결과'라는 점이며, 올바르게는 '선택 정렬이었다면 두 번째 위치는 18이 아니라 9이다.'이다.
③ X: 버블 정렬은 인접 원소를 교환하면서 큰 값이 뒤로 밀려난다. 1회전만 끝나도 최댓값 33이 맨 뒤로 이동해야 하므로 제시 결과처럼 네 번째 자리에 남아 있지 않는다. 틀린 표현은 '제시 결과가 버블 정렬의 중간 상태'라는 점이며, 올바르게는 '버블 정렬이었다면 33은 이미 뒤쪽 끝으로 이동해 있어야 한다.'이다.
④ X: 기수 정렬은 자릿수별로 분배와 수집을 반복하는 방식이므로 앞쪽 일부만 정렬된 삽입 정렬식 중간 상태와 패턴이 다르다. 일의 자리 기준 첫 패스 결과부터 제시 배열과 같은 형태가 되지 않는다. 틀린 표현은 '제시 결과가 기수 정렬 2회 연산 결과'라는 점이며, 올바르게는 '제시 결과는 앞부분 정렬 구간이 단계적으로 확장되는 삽입 정렬의 중간 상태이다.'이다.

오답 포인트

많이 틀리는 지점은 '2번 연산'을 비교 두 번으로 읽는 점과, 앞부분만 정렬된 모양을 선택 정렬로 착각하는 점이다. 삽입 정렬은 앞의 정렬 구간이 한 칸씩 넓어지고, 선택 정렬은 매 회전 최솟값이 앞자리로 고정되며, 버블 정렬은 최댓값이 뒷자리로 밀린다는 차이를 구분해야 한다.

다시 나오면 이렇게 판단

중간 결과에서 앞쪽 일부만 정렬되어 있고 새 원소가 그 구간 안으로 끼어든 흔적이 보이면 삽입 정렬로 판단한다. 매 회전마다 최솟값이 맨 앞으로 오면 선택 정렬이고, 최댓값이 맨 뒤로 이동하면 버블 정렬로 판단한다.

암기 포인트

삽입 정렬은 정렬된 앞구간에 다음 원소를 끼워 넣는 방식으로 기억한다.

NoteBase

문 3. 다음 C 코드로 표현한 알고리즘의 시간 복잡도는?

```
Algorithm(int n) {
    int i, j, k;
    int sum = 0;
    for(i = 0; i < 1000; i++)
        for(j = 1; j < (n - 500); j++)
            for(k = (n - 100); k < (n + 100); k++)
                sum = sum + i + j + k;
}
```

① $O(1)$

② $O(n)$

③ $O(n^2)$

④ $O(n^3)$

정답 2번

출제 포인트

이 문제는 중첩 반복문에서 각 반복문의 실제 반복 횟수를 분리해 전체 시간 복잡도를 구할 수 있는지를 묻는다.

조건 정리

주어진 반복:

- i: 0부터 999까지이므로 1000회 반복이다.
- j: 1부터 $j < n - 500$까지이므로 약 $n - 501$회 반복이다.
- k: $n - 100$부터 $k < n + 100$까지이므로 200회 반복이다.

구해야 할 값: 전체 수행 횟수의 점근적 차수이다.

주의할 조건: n에 따라 변하는 반복문은 j뿐이고, i와 k는 상수 반복이다.

NoteBase

사용할 공식

전체 반복 횟수는 완전히 중첩된 각 반복문의 반복 횟수를 곱해 구한다. 따라서
$T(n) = 1000 \times (n - 501) \times 200$ 이고, 상수배와 상수항을 제거하면 $O(n)$이다.
이 공식을 쓰는 이유:
세 반복문이 완전히 중첩되어 있지만 n에 따라 커지는 반복문은 j 하나뿐이기 때문이다.

계산 과정

1. 바깥쪽 i 반복문은 $i = 0$부터 999까지이므로 1000회 반복된다.
2. 가운데 j 반복문은 $j = 1$부터 $j < n - 500$까지이므로 약 $n - 501$회 반복되어 $O(n)$이다.
3. 안쪽 k 반복문은 $k = n - 100$부터 $k < n + 100$까지이므로 정확히 200회 반복되어 $O(1)$이다.
4. 전체 수행 횟수는 $T(n) = 1000 \times (n - 501) \times 200 = 200000n - 100200000$ 이다.
5. 점근 표기에서는 상수배와 상수항을 제거하므로 시간 복잡도는 $O(n)$이다.

선지 판단

① X: 틀린 표현은 "O(1)"이다. 가운데 j 반복문이 n에 비례해 늘어나므로 전체는 상수 시간이 아니다. 올바르게는 $O(n)$이다.
② O: i와 k 반복문은 각각 1000회, 200회로 상수이고 j 반복문만 n에 비례하므로 전체 시간 복잡도는 $O(n)$이다.
③ X: 틀린 표현은 "O(n^2)"이다. n에 비례하는 반복문이 두 개가 아니라 하나뿐이므로 제곱 시간이 되지 않는다. 올바르게는 $O(n)$이다.
④ X: 틀린 표현은 "O(n^3)"이다. 반복문이 세 겹이라는 모양만 보고 모두 n회라고 판단하면 안 된다. 두 반복문은 상수 회수이므로 올바르게는 $O(n)$이다.

오답 포인트

중첩 반복문이라는 이유만으로 곧바로 $O(n^2)$나 $O(n^3)$로 올리는 실수가 대표적 함정이다. 이 문항에서는 k의 시작값과 끝값에 n이 들어가 있어도 구간 길이가 200으로 고정되어 있다는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

중첩 반복문은 각 반복문의 실제 반복 횟수가 상수인지 n에 의존하는지부터 분리하고, 그다음 상수 반복은 $O(1)$로 두고 n이 들어간 반복문만 곱해 최종 차수를 정한다.

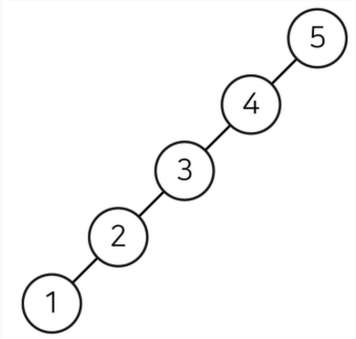
암기 포인트

반복문의 겹수보다 중요한 것은 각 반복문의 실제 반복 횟수이며, 상수 반복은 모두 흡수되고 n이 들어간 반복문만 시간 복잡도를 결정한다.

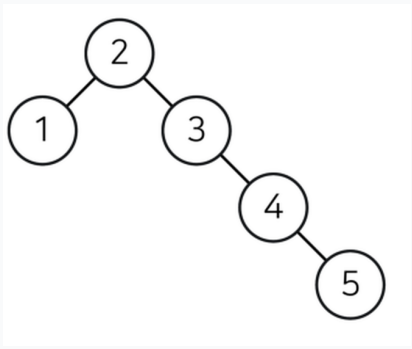
NoteBase

문 4. 이진 트리(binary tree) 중에서 전위 순회(preorder traversal)와 중위 순회(inorder traversal)한 결과가 동일한 것은?

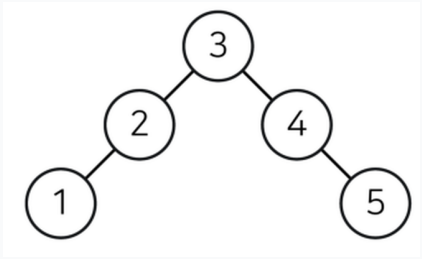
①



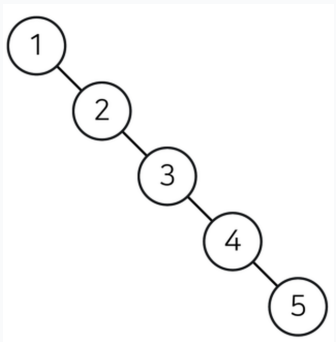
②



③



④



정답 4번

출제 포인트

이 문제는 전위 순회가 루트를 먼저 방문하고 중위 순회가 왼쪽 서브트리를 먼저 방문한다는 차이로, 두 결과가 같아지려면 어떤 트리 구조가 필요한지를 묻는 문제이다.

NoteBase

선지별 해설

1번: X. 이 트리는 5에서 1까지 왼쪽 자식만 이어진다. 전위 순회 결과는 5-4-3-2-1이고 중위 순회 결과는 1-2-3-4-5이므로 서로 다르다. 틀린 지점은 모든 노드에 왼쪽 자식이 있다는 점이다. 올바르게는 모든 노드의 왼쪽 자식이 없어야 전위와 중위가 같아진다.
2번: X. 루트 2의 왼쪽 자식 1 때문에 중위 순회에서는 1이 2보다 먼저 나오지만 전위 순회에서는 2가 먼저 나온다. 전위 순회 결과는 2-1-3-4-5이고 중위 순회 결과는 1-2-3-4-5이다. 틀린 지점은 왼쪽 자식 1이 존재한다는 점이다. 올바르게는 루트를 포함한 모든 노드에서 왼쪽 자식이 없어야 한다.
3번: X. 루트 3의 왼쪽 서브트리 2-1 때문에 중위 순회는 1-2-3-4-5가 되고 전위 순회는 3-2-1-4-5가 된다. 틀린 지점은 왼쪽 서브트리가 존재하여 중위 순회가 루트보다 먼저 왼쪽 노드들을 출력한다는 점이다. 올바르게는 왼쪽 서브트리 없이 오른쪽으로만 이어지는 구조여야 한다.
4번: O. 이 트리는 1-2-3-4-5가 오른쪽 자식으로만 이어진 우편향 트리이다. 전위 순회는 1-2-3-4-5이고 중위 순회도 왼쪽 자식이 없으므로 1-2-3-4-5가 되어 동일하다.

오답 포인트

왼쪽으로 길게 늘어진 1번과 오른쪽으로 길게 늘어진 4번을 대칭처럼 보고 같은 성질이라고 착각하기 쉽다. 하지만 중위 순회는 왼쪽 자식이 있으면 반드시 그 왼쪽 노드가 루트보다 먼저 출력되므로, 왼쪽 자식의 존재 여부가 핵심 함정이다.

다시 나오면 이렇게 판단

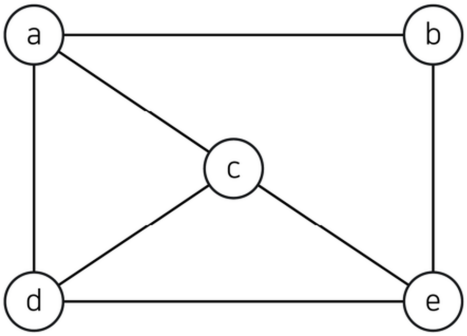
각 선택지에서 왼쪽 자식이 하나라도 있는지 먼저 확인하면 된다. 왼쪽 자식이 보이는 순간 전위 순회와 중위 순회는 순서가 갈라지고, 왼쪽 자식이 전혀 없을 때만 두 순회 결과가 같아진다.

암기 포인트

전위와 중위가 같으려면 왼쪽 자식이 없는 오른쪽 편향 트리이다.

NoteBase

문 5. 다음 그래프의 인접 행렬이 아래와 같을 때, (가) ~ (사)에 들어갈 값을 모두 더한 결과는?



	a	b	c	d	e
a	0	(가)	1	1	0
b	(나)	0	0	0	(다)
c	1	0	0	(라)	1
d	1	0	1	0	1
e	0	(마)	1	(바)	(사)

① 4

② 5

③ 6

④ 7

정답 3번

출제 포인트

이 문제는 무방향 단순 그래프의 인접 행렬이 대칭이고 자기 자신으로의 간선이 없으면 대각 원소가 0이라는 점을 묻는 문제이다.

NoteBase

선지별 해설

① X: 4는 틀리다. 그림에서 a-b, b-a, b-e, c-d, e-b, e-d는 모두 간선이 있으므로 (가), (나), (다), (라), (마), (바)는 1이고, 자기 루프가 없으므로 (사)는 0이다. 합은 1+1+1+1+1+1+0=6이다. 틀린 값 4 → 올바르게는 6이다.
② X: 5는 틀리다. 빠지기 쉬운 값은 대칭 위치 중 하나인데, 무방향 그래프의 인접 행렬은 대칭이므로 a-b와 b-a, b-e와 e-b를 따로 채워야 한다. 합은 6이다. 틀린 값 5 → 올바르게는 6이다.
③ O: 6이 맞다. (가)=1, (나)=1, (다)=1, (라)=1, (마)=1, (바)=1, (사)=0이므로 총합은 6이다.
④ X: 7은 틀리다. 7이 되려면 (사)까지 1이어야 하는데 e에서 e로 돌아오는 자기 루프가 그림에 없다. 따라서 대각 원소인 (사)는 0이다. 틀린 값 7 → 올바르게는 6이다.

오답 포인트

무방향 그래프인데도 대칭 위치를 하나만 세거나, 대각 원소를 자동으로 1로 착각하면 오답이 된다. 특히 (사)는 e의 차수가 아니라 e-e 자기 연결 여부를 뜻한다는 점이 함정이다.

다시 나오면 이렇게 판단

먼저 그래프가 방향 그래프인지 무방향 그래프인지 확인하고, 그다음 빈칸을 간선 유무로 채운다. 마지막에는 대칭 위치와 대각 원소 0 여부를 한 번 더 점검하면 된다.

암기 포인트

무방향 단순 그래프의 인접 행렬은 대칭이고 대각선은 0이다.

NoteBase

문 6. 다음 C 코드에서 power(2.0, 20, 0)를 호출할 때, 변수 c의 마지막 출력값은?

```
double power(double x, int n, int c)
{
    printf("c : %d\n", c);
    if(n == 0) return 1;
    else if(n % 2 == 0) {
        c++;
        return power(x * x, n / 2, c);
    }
    else {
        c++;
        return x * power(x * x, (n - 1) / 2, c);
    }
}
```

① 5
② 6
③ 7
④ 8

정답 1번

출제 포인트

이 문제는 재귀 함수에서 인자 c가 언제 출력되고 언제 1 증가한 값으로 다음 호출에 전달되는지를 추적해 마지막 출력값을 구하는 문제이다.

조건 정리

주어진 호출은 power(2.0, 20, 0)이다.
각 호출의 시작에서 먼저 printf("c : %d
", c)로 현재 c를 출력한다.
그 뒤 n이 0이 아니면 c를 1 증가시키고 다음 재귀 호출로 넘긴다.
따라서 마지막 출력값은 마지막 호출의 인자 c 값이다.

NoteBase

사용할 공식

공식이라기보다 호출 순서를 따라간다.
짝수 n이면 power(x * x, n / 2, c + 1)로 진행되고, 홀수 n이면 power(x * x, (n - 1) / 2, c + 1)로 진행된다.
이 원리를 쓰는 이유는 매 호출마다 c가 정확히 1씩 증가하고, 출력은 증가 전의 값으로 이루어지기 때문이다.

계산 과정

1. power(2.0, 20, 0)에서 먼저 0을 출력하고, n=20은 짝수이므로 다음 호출 인자는 c=1이 된다.
2. power(4, 10, 1)에서 1을 출력하고, n=10은 짝수이므로 다음 호출 인자는 c=2가 된다.
3. power(16, 5, 2)에서 2를 출력하고, n=5는 홀수이므로 다음 호출 인자는 c=3이 된다.
4. power(256, 2, 3)에서 3을 출력하고, n=2는 짝수이므로 다음 호출 인자는 c=4가 된다.
5. power(65536, 1, 4)에서 4를 출력하고, n=1은 홀수이므로 다음 호출 인자는 c=5가 된다.
6. power(4294967296, 0, 5)에서 5를 출력한 뒤 n=0이므로 종료된다.

따라서 마지막 출력값은 5이다.

선지 판단

① O: 호출 순서를 끝까지 따라가면 출력되는 c 값은 0, 1, 2, 3, 4, 5이고 마지막 값은 5이다.
② X: 6은 마지막 호출에서 c를 한 번 더 증가시킨 값처럼 착각한 경우이다. 틀린 표현은 마지막 출력값이 6이라는 점이고, 올바르게는 n=0인 호출에서 이미 5를 출력한 뒤 바로 종료된다.
③ X: 7은 재귀 깊이를 실제보다 더 많이 센 경우이다. 틀린 표현은 호출이 7까지 진행된다는 점이고, 올바르게는 n이 20→10→5→2→1→0으로 줄어 총 6번 호출된다.
④ X: 8은 c가 매 분기마다 여러 번 증가한다고 오해한 경우이다. 틀린 표현은 마지막 출력값이 8이라는 점이고, 올바르게는 각 호출마다 c는 한 번만 증가하여 마지막 출력값은 5이다.

오답 포인트

printf가 c++보다 먼저 실행된다는 순서를 놓치면 6으로 잘못 보기 쉽다. 또한 c가 참조로 누적된다고 막연히 생각하거나 호출 횟수를 과대계산하면 7이나 8 같은 값으로 흔들린다.

다시 나오면 이렇게 판단

재귀 추적 문제는 현재 호출에서 먼저 실행되는 문장인지, 다음 호출 직전에 바뀌는 값인지 순서를 분리해서 본다. 그다음 n의 변화만 따라가 호출 횟수와 각 단계의 인자 값을 표처럼 적으면 마지막 출력값을 안정적으로 판단할 수 있다.

암기 포인트

재귀 추적의 핵심은 출력 시점과 값 변경 시점을 분리하는 것이다. 이 문제에서는 출력 후 c 증가이므로 마지막 출력값은 마지막 호출 인자 5이다.

NoteBase

문 7. 다음 단정도(32비트) IEEE 754 표준 방식으로 표현된 2진수를 10진수 형식으로 옳게 변환한 것은?

0 9 31
11000001101001000000000000000000

① 10.5
② -10.75
③ -20.5
④ -20.75

정답 3번

출제 포인트

이 문제는 IEEE 754 단정도에서 부호 비트, 지수 비트, 가수 비트를 해석해 10진수 값을 구하는 문제이다.

조건 정리

주어진 값: 32비트 비트열 11000001101001000000000000000000이다.
구해야 할 값: 이 비트열의 10진수 값이다.
주의할 조건: 단정도는 부호 1비트, 지수 8비트, 가수 23비트이며 지수 bias는 127이다.

사용할 공식

단정도 실수값은 $(-1)^s \times 1.f \times 2^{E-127}$로 해석한다.
이 공식을 쓰는 이유: IEEE 754 단정도는 부호 비트 s, 지수 비트 E, 가수 비트 f를 이 규칙으로 조합해 실제 값을 나타내기 때문이다.

계산 과정

1. 부호 비트는 첫 비트 1이므로 음수이다.
2. 지수 비트는 10000011_2이므로 131이고, 실제 지수는 $131 - 127 = 4$이다.
3. 가수 비트는 01001000000000000000000이므로 유효숫자는 1.01001_2이다.
4. 따라서 값은 $-1.01001_2 \times 2^4$이고, 이는 $-10100.1_2 = -20.5$이다.

NoteBase

선지별 해설

① X: 10.5는 양수인데 이 비트열의 부호 비트는 1이므로 음수가 되어야 한다. 또한 지수도 4이므로 크기 역시 10.5가 아니다. 틀린 표현은 '10.5'이고, 올바르게는 '-20.5'이다.
② X: 음수라는 점은 맞지만 값이 다르다. -10.75가 되려면 지수가 3이거나 가수부가 1.01011_2 수준으로 달라져야 한다. 주어진 비트열은 지수 4, 가수 1.01001_2이므로 틀린 표현은 '-10.75'이고, 올바르게는 '-20.5'이다.
③ O: 부호 비트 1, 지수 $10000011_2 = 131$, 가수 1.01001_2를 적용하면 $-1.01001_2 \times 2^4 = -20.5$가 된다. 계산 결과와 일치하므로 옳다.
④ X: 부호와 지수는 맞지만 가수부가 다르다. -20.75는 -10100.11_2이므로 가수가 1.010011_2이어야 한다. 주어진 비트열의 가수는 1.01001_2이므로 틀린 표현은 '-20.75'이고, 올바르게는 '-20.5'이다.

오답 포인트

부호 비트만 보고 음수 여부만 맞춘 뒤 가수부를 대충 읽거나, 지수 bias 127을 빼지 않아 크기를 잘못 잡는 경우가 많다. 특히 ④처럼 소수 부분 0.1_2와 0.11_2를 혼동하면 -20.5를 -20.75로 잘못 고르기 쉽다.

다시 나오면 이렇게 판단

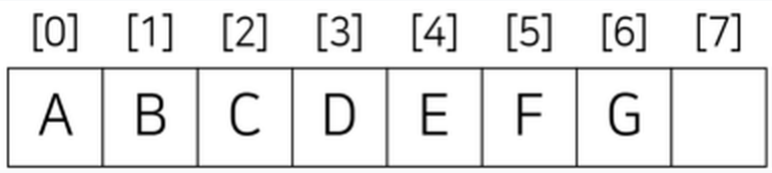
IEEE 754 변환 문제는 부호 확인, 지수에서 bias 차감, 가수에 숨은 1 복원, 마지막에 2진수를 10진수로 변환하는 순서로 판단하면 된다. 소수 부분은 $0.1_2 = 0.5$, $0.01_2 = 0.25$, $0.001_2 = 0.125$처럼 자릿값으로 다시 검산할 수 있다.

암기 포인트

단정도는 1비트 부호 + 8비트 지수 + 23비트 가수, 실제 지수는 $E - 127$, 정규화 수의 가수는 항상 숨은 1을 붙여 읽는다.

NoteBase

문 8. 다음 선형 리스트에서 연산 1과 연산 2가 수행된 후 리스트 상태와 두 연산에 사용된 원소의 총이동 횟수를 바르게 연결한 것은?



연산 1: 인덱스 1번에 원소 H 삽입

연산 2: 인덱스 3번 원소 삭제



정답 3번

출제 포인트

이 문제는 배열 기반 선형 리스트에서 삽입과 삭제가 연속 수행될 때 원소 이동 방향, 삭제 대상 인덱스, 총 이동 횟수를 함께 판단하는 문제이다.

올바른 순서

1. 연산 1에서 인덱스 1에 H를 넣기 위해 [1]~[6]의 B, C, D, E, F, G를 한 칸씩 오른쪽으로 이동한다.
2. H를 [1]에 삽입하면 리스트는 A, H, B, C, D, E, F, G가 된다.
3. 연산 2에서 현재 인덱스 3의 원소는 C이므로 C를 삭제한다.
4. 삭제 뒤 [4]~[7]의 D, E, F, G를 한 칸씩 왼쪽으로 이동하면 최종 리스트는 A, H, B, D, E, F, G가 된다.

왜 이 순서인가: 삽입이 먼저 일어나므로 삭제 대상 인덱스 3은 원래의 D가 아니라 삽입 후 밀린 C가 된다. 이동 횟수는 삽입 때 6회, 삭제 때 4회로 총 10회이다.

NoteBase

선지별 해설

① X: 제시된 상태는 B와 D가 사라지고 [3]에 빈칸이 남아 있어 배열 리스트의 삭제 결과와 맞지 않는다. 올바르게는 최종 리스트가 A, H, B, D, E, F, G가 되고 빈칸은 마지막 [7]에 남는다. 총이동 횟수도 3회가 아니라 10회이다.
② X: 제시된 상태는 B와 D가 함께 빠져 있어 인덱스 3의 삭제 대상을 잘못 잡은 경우이다. 올바르게는 삽입 후 인덱스 3의 C를 지워 A, H, B, D, E, F, G가 된다. 총이동 횟수도 3회가 아니라 10회이다.
③ O: 인덱스 1 삽입 후 리스트가 A, H, B, C, D, E, F, G가 되고, 이어서 인덱스 3의 C를 삭제하면 A, H, B, D, E, F, G가 된다. 이동 횟수도 오른쪽 이동 6회와 왼쪽 이동 4회를 더한 10회가 맞다.
④ X: 총이동 횟수 10회는 맞지만 리스트 상태가 틀렸다. 배열 기반 선형 리스트는 삭제 후 중간에 빈칸을 남기지 않으므로 [3]에는 D가 와야 하고 빈칸은 마지막 [7]에 있어야 한다.

오답 포인트

인덱스 3 삭제를 원래 리스트 기준으로 읽어 D를 지운다고 착각하거나, 배열 리스트 삭제 뒤 빈칸이 중간에 남는다고 생각하면 ④ 같은 오답을 고르게 된다. 삽입 후 현재 인덱스를 다시 계산해야 한다는 점이 함정이다.

다시 나오면 이렇게 판단

배열 리스트 연산은 매 단계의 결과를 먼저 갱신한 뒤 다음 연산의 인덱스를 읽는다. 삽입은 해당 위치 이후 원소를 오른쪽으로 밀고, 삭제는 뒤 원소를 왼쪽으로 당긴다고 보면 최종 상태와 이동 횟수를 함께 정리할 수 있다.

암기 포인트

배열 리스트는 삽입이면 오른쪽 밀기, 삭제면 왼쪽 당기기이며 중간 빈칸은 남지 않는다.

NoteBase

문 9. 다음은 힙 정렬(heap sort) 알고리즘을 구현한 C 코드이다. 리스트 L이 (61, 11, 59, 15, 48, 19)로 주어졌을 때, for 문에 관한 설명으로 옳은 것만을 모두 고르면? (단, 리스트 L을 배열로 표현할 때 색인 1부터 저장하고, n은 리스트의 크기를 나타낸다)

```
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )

void heap_sort(int L[], int n){
    int i, temp;
    for(i = n/2; i >= 1; i--)  ····· (가)
        adjust(L, i, n);
    for(i = n-1; i >= 1; i--) {  ····· (나)
        SWAP(L[1], L[i+1], temp);
        adjust(L, 1, i);
    }
}

void adjust(int L[], int root, int n) {
    int next = 2 * root;
    int X = L[root];
    while(next <= n) {
        if(next < n && L[next] < L[next + 1])
            next++;
        if(X > L[next]) break;
        L[next / 2] = L[next];
        next *= 2;
    }
    L[next / 2] = X;
}
```

ㄱ. (가) for 문을 수행한 후 리스트 L에는 (61, 48, 59, 15, 19, 11)이 저장된다. ㄴ. (나) for 문을 첫 1회 수행한 후 리스트 L에는 (59, 48, 19, 15, 11, 61)이 저장된다. ㄷ. (나) for 문의 시간 복잡도는 $O(n \log n)$이다.

① ㄱ, ㄴ

② ㄱ, ㄷ

③ ㄴ, ㄷ

④ ㄱ, ㄴ, ㄷ

정답 3번

출제 포인트

이 문제는 힙 정렬에서 최대 힙을 만드는 단계와 첫 정렬 반복 뒤 배열 상태, 전체 정렬 단계의 시간 복잡도를 절차적으로 판단하는 문제이다.

NoteBase

올바른 순서

1. (가) for문은 i=3, 2, 1 순서로 adjust를 호출하여 초기 배열을 최대 힙으로 만든다.
2. i=3에서는 자식이 하나뿐이어서 59와 19를 비교해 변화가 없고, i=2에서는 11이 더 큰 자식 48 아래로 내려가 (61, 48, 59, 15, 11, 19)가 된다.
3. i=1에서는 루트 61이 두 자식 48, 59보다 이미 크므로 추가 이동이 없어 (가) 종료 뒤 배열은 (61, 48, 59, 15, 11, 19)이다.
4. (나) for문은 루트와 마지막 원소를 교환한 뒤 남은 구간 1..i에 대해 adjust를 수행하여 최대값을 뒤에서부터 확정한다. 첫 반복에서는 61과 19를 바꾼 뒤 재조정하여 (59, 48, 19, 15, 11, 61)이 된다.

왜 이 순서: 최대 힙이 먼저 만들어져야 루트가 항상 현재 구간의 최댓값이 되고, 그다음에만 끝 원소와 교환하여 정렬 구간을 줄일 수 있기 때문이다.

선지별 해설

① X. ①은 ㄱ, ㄴ이 모두 참이어야 하는데 ㄱ이 틀리다. (가) 종료 뒤 배열의 마지막 두 값은 19, 11이 아니라 11, 19이다. 올바르게는 ㄱ을 빼고 ㄴ, ㄷ만 포함한 ③이어야 한다.

② X. ②도 ㄱ을 포함하므로 틀리다. ㄷ의 시간 복잡도 판단은 맞지만, 힙 구성 직후 배열을 잘못 적은 ㄱ 때문에 성립하지 않는다. 올바르게는 ㄱ 대신 ㄴ을 포함한 ③이어야 한다.

③ O. ㄴ은 첫 정렬 반복에서 61을 맨 뒤로 보낸 뒤 남은 5개에 대해 adjust를 수행하면 (59, 48, 19, 15, 11, 61)이 되므로 맞다. ㄷ도 (나) for문이 총 $n-1$번 돌고 각 adjust가 힙의 높이만큼인 $O(\log n)$이므로 전체가 $O(n \log n)$이어서 맞다.

④ X. ④는 ㄱ, ㄴ, ㄷ이 모두 참이어야 하지만 ㄱ이 틀리므로 성립하지 않는다. 올바르게는 거짓인 ㄱ을 제외한 ㄴ, ㄷ만 옳다.

오답 포인트

힙 구성 단계에서 i=2 조정 결과만 보고 다섯 번째와 여섯 번째 값의 위치를 뒤집어 적기 쉽다. 또 (나)의 첫 반복에서는 교환 직후 배열이 아니라 adjust까지 끝난 뒤 배열을 써야 한다는 점을 놓치면 ㄴ 판단이 흔들린다.

다시 나오면 이렇게 판단

힙 정렬 추적 문제는 먼저 힙 구성 단계에서 각 i의 adjust 결과를 순서대로 적고, 정렬 단계에서는 교환 직후와 재조정 직후를 분리해 확인하면 된다. 시간 복잡도는 바깥 반복 횟수와 adjust 한 번의 높이 비용을 곱해 판단한다.

암기 포인트

최대 힙 구성은 아래에서 위로, 정렬은 루트 교환 후 재조정, 전체 시간 복잡도는 $O(n \log n)$이다.

NoteBase

문 10. 다음 8개의 데이터를 순서대로 삽입하여 AVL 트리를 구성하였을 때, 이에 대한 설명으로 옳은 것은?

19, 7, 10, 15, 25, 32, 17, 5

① 루트 노드의 번호는 17이다.
② 중위 순회했을 때 네 번째 방문하는 노드의 번호는 15이다.
③ 전위 순회했을 때 세 번째 방문하는 노드의 번호는 10이다.
④ 후위 순회했을 때 다섯 번째 방문하는 노드의 번호는 32이다.

정답 2번

출제 포인트

이 문제는 AVL 트리에 데이터를 순서대로 삽입하면서 회전 결과를 확정한 뒤 최종 트리의 중위·전위·후위 순회 결과를 판단하는 문제이다.

올바른 순서

1. 19, 7, 10을 삽입하면 19에서 LR 불균형이 발생하여 회전 후 루트가 10이 된다.
2. 15, 25, 32를 차례로 삽입하면 10의 오른쪽이 깊어져 좌회전이 일어나고 루트가 19가 된다.
3. 17, 5를 삽입하면 추가 회전 없이 최종 트리는 루트 19, 왼쪽 10, 오른쪽 25 구조가 된다.
4. 최종 트리의 순회 결과는 중위 5, 7, 10, 15, 17, 19, 25, 32 / 전위 19, 10, 7, 5, 15, 17, 25, 32 / 후위 5, 7, 17, 15, 10, 32, 25, 19가 된다.

왜 이 순서: AVL은 삽입 직후 불균형이 생긴 조상에서 회전이 먼저 확정되어야 이후의 루트 위치와 순회 순서를 판단할 수 있다.

선지별 해설

① X: 최종 루트 노드는 19이다. 틀린 표현은 "17"이다. 올바르게는 32 삽입 뒤 10에서 좌회전이 일어나 루트가 19가 되고, 이후 17과 5를 넣어도 루트는 19로 유지된다.
② O: 중위 순회는 왼쪽-루트-오른쪽 순서이므로 AVL 여부와 무관하게 키 값이 오름차순으로 나온다. 최종 결과는 5, 7, 10, 15, 17, 19, 25, 32이므로 네 번째 방문 노드는 15이다.
③ X: 전위 순회는 루트-왼쪽-오른쪽 순서이다. 최종 전위 순회 결과는 19, 10, 7, 5, 15, 17, 25, 32이므로 세 번째 방문 노드는 7이다. 틀린 표현은 "10"이다. 올바르게는 세 번째 방문 노드는 7이다.
④ X: 후위 순회는 왼쪽-오른쪽-루트 순서이다. 최종 후위 순회 결과는 5, 7, 17, 15, 10, 32, 25, 19이므로 다섯 번째 방문 노드는 10이다. 틀린 표현은 "32"이다. 올바르게는 다섯 번째 방문 노드는 10이다.

NoteBase

오답 포인트

17이 뒤늦게 삽입되므로 17이 루트로 올라간다고 착각하거나, 중위·전위·후위 순회 순서를 서로 바꾸어 적용하면 오답이 된다. 이 문항은 먼저 최종 트리 모양을 확정하지 않으면 ③, ④를 연속해서 틀리기 쉽다.

다시 나오면 이렇게 판단

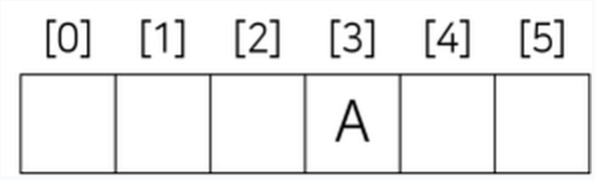
AVL 삽입 문제는 삽입할 때마다 불균형 노드와 회전 종류를 먼저 적고 최종 트리를 그린 뒤, 중위는 오름차순, 전위는 루트-좌-우, 후위는 좌-우-루트 순서로 대입하면 안정적으로 판단할 수 있다.

암기 포인트

AVL 문제는 회전으로 최종 트리를 먼저 확정하고, 중위는 오름차순이라는 기준으로 순회 선지를 빠르게 검산한다.

NoteBase

문 11. 다음 원형 큐에 대해 1부터 7까지 연산을 차례로 수행했을 때, 수행이 완료된 후 큐의 상태는? (단, 현재 상태는 front = 2, rear = 3이다)



연산 1: 원소 B 삽입
연산 2: 원소 한 개 삭제
연산 3: 원소 C 삽입
연산 4: 원소 D 삽입
연산 5: 원소 두 개 삭제
연산 6: 원소 E 삽입
연산 7: 원소 F 삽입

① [0] [1] [2] [3] [4] [5]
| | | | A | E | F |

② [0] [1] [2] [3] [4] [5]
| D | | | | F | E |

③ [0] [1] [2] [3] [4] [5]
| | F | E | A | | |

④ [0] [1] [2] [3] [4] [5]
| D | E | F | | | |

정답 4번

출제 포인트

이 문제는 원형 큐에서 front와 rear의 이동 규칙에 따라 삽입과 삭제를 순서대로 적용한 뒤 최종 저장 위치를 판단하는 문제이다.

NoteBase

올바른 순서

초기 상태는 front=2, rear=3이고 A는 [3]에 있다.
1. B 삽입: rear가 4가 되어 B가 [4]에 들어간다.
2. 원소 한 개 삭제: front가 3이 되어 A가 삭제되고 B만 [4]에 남는다.
3. C 삽입: rear가 5가 되어 C가 [5]에 들어간다.
4. D 삽입: rear가 원형으로 돌아가 0이 되어 D가 [0]에 들어간다.
5. 원소 두 개 삭제: front가 4, 5로 이동하면서 B와 C가 차례로 삭제되고 D만 [0]에 남는다.
6. E 삽입: rear가 1이 되어 E가 [1]에 들어간다.
7. F 삽입: rear가 2가 되어 F가 [2]에 들어간다.
따라서 최종 상태는 [0]=D, [1]=E, [2]=F이다.
왜 이 순서가 중요한가: 삭제는 front를 먼저 한 칸 이동시킨 뒤 그 위치의 원소를 꺼내므로, 초기의 A가 먼저 빠지고 이후 B, C가 차례로 삭제된다는 점이 정답을 가른다.

선지별 해설

① X: [3]=A, [4]=E, [5]=F 형태는 A가 끝까지 남아 있다고 본 경우이다. 하지만 연산 2에서 front가 3으로 이동하면서 A가 먼저 삭제된다. 틀린 표현은 A가 최종 상태에 남는다는 점이다. 올바르게는 최종 상태는 [0]=D, [1]=E, [2]=F이다.
② X: [0]=D, [4]=F, [5]=E 형태는 원형으로 돌아간 뒤 rear의 진행 순서를 거꾸로 본 경우이다. 연산 6에서 E가 [1], 연산 7에서 F가 [2]에 들어가므로 [4], [5]에 남을 수 없다. 틀린 표현은 E와 F의 저장 위치이다. 올바르게는 D, E, F가 차례로 [0], [1], [2]에 놓인다.
③ X: [1]=F, [2]=E, [3]=A 형태는 A를 삭제하지 않았고 E와 F의 삽입 순서도 뒤집은 경우이다. 연산 2에서 A는 삭제되고, 연산 6이 E 삽입, 연산 7이 F 삽입이므로 E가 F보다 먼저 저장된다. 틀린 표현은 A 잔존과 E/F 위치이다. 올바르게는 [0]=D, [1]=E, [2]=F이다.
④ O: 연산 4에서 D가 [0]으로 원형 이동하여 저장되고, 연산 5에서 B와 C가 삭제된 뒤 연산 6, 7에서 E와 F가 각각 [1], [2]에 들어가므로 [0]=D, [1]=E, [2]=F가 된다.

오답 포인트

가장 흔한 함정은 front가 현재 첫 원소를 가리킨다고 착각하는 점이다. 이 문제의 표기에서는 front가 첫 원소의 바로 앞을 가리키므로 연산 2에서 삭제되는 원소는 A이다. 또 rear가 5 다음에 0으로 순환한다는 점을 놓치면 D, E, F의 위치를 잘못 적게 된다.

다시 나오면 이렇게 판단

원형 큐 문제는 현재 front와 rear의 의미를 먼저 정한 뒤, 삽입이면 rear를 한 칸 이동시키고 저장 위치를 기록하고, 삭제면 front를 한 칸 이동시키며 빠지는 원소를 지우는 방식으로 한 단계씩 추적하면 된다. 끝까지 남아 있는 원소와 실제 배열 인덱스를 함께 적으면 실수를 줄일 수 있다.

NoteBase

암기 포인트

원형 큐는 삭제 때 front 이동 후 제거, 삽입 때 rear 이동 후 저장이 핵심이다.

NoteBase

문 12. 비어 있는 초기 상태의 이진 트리에 다섯 개의 키(1, 2, 3, 4, 5)를 순서대로 하나씩 삽입하면서 레드-블랙 트리를 만든 후, 중위 순회한 결과는?

① 1, 2, 3, 4, 5

② 1, 3, 5, 4, 2

③ 2, 1, 3, 5, 4

④ 2, 1, 4, 3, 5

정답 1번

출제 포인트

이 문제는 레드-블랙 트리가 이진 탐색 트리의 성질을 유지하므로 중위 순회 결과가 항상 키의 오름차순이라는 점을 묻는다.

핵심 개념

레드-블랙 트리는 균형을 맞추기 위해 회전과 색 변경을 하더라도 이진 탐색 트리의 대소 관계를 유지한다. 중위 순회는 왼쪽 서브트리, 루트, 오른쪽 서브트리 순서이므로 결과가 항상 정렬된 키 순서가 된다.

선지별 해설

① O. 키가 1, 2, 3, 4, 5이면 최종 트리의 모양과 무관하게 중위 순회 결과는 항상 1, 2, 3, 4, 5이다.
② X. 틀린 표현은 '1, 3, 5, 4, 2'라는 방문 순서이다. 중위 순회는 이진 탐색 트리에서 키를 오름차순으로 출력하므로 실제로는 1, 2, 3, 4, 5이다.
③ X. 틀린 표현은 '2, 1, 3, 5, 4'라는 방문 순서이다. 이는 왼쪽-루트-오른쪽 규칙을 만족하지 않으므로 중위 순회 결과가 아니며, 올바른 결과는 1, 2, 3, 4, 5이다.
④ X. 틀린 표현은 '2, 1, 4, 3, 5'라는 방문 순서이다. 이는 최종 트리 모양을 전위 순회한 결과처럼 보일 수 있지만 중위 순회 결과는 아니고, 올바르게는 1, 2, 3, 4, 5이다.

오답 포인트

최종 레드-블랙 트리의 모양을 떠올린 뒤 전위 순회나 레벨 순회를 중위 순회와 혼동하면 ④ 같은 선택지에 끌린다. 이 문항의 핵심은 균형 조정 방식이 아니라 이진 탐색 트리의 중위 순회 성질이다.

다시 나오면 이렇게 판단

균형 이진 탐색 트리에서 중위 순회를 묻는 문제는 먼저 그 구조가 이진 탐색 트리인지부터 확인한다. 이진 탐색 트리이면 회전이나 재균형 여부와 무관하게 중위 순회 결과를 키의 오름차순으로 판단할 수 있다.

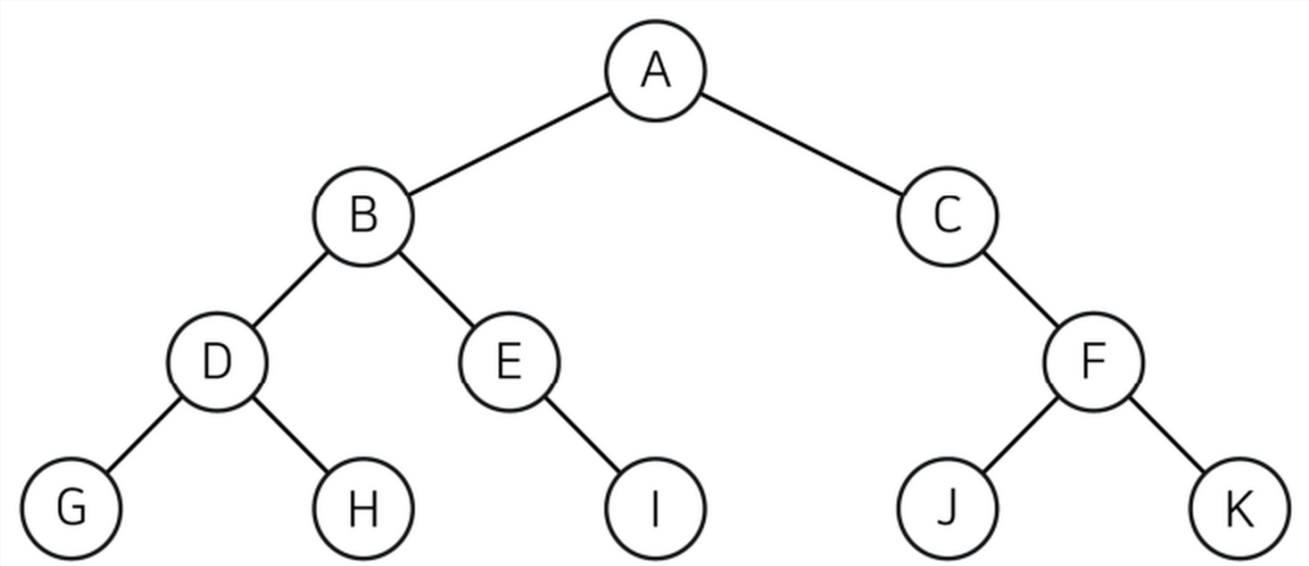
NoteBase

암기 포인트

레드-블랙 트리는 모양은 바뀌어도 이진 탐색 트리이고, 이진 탐색 트리의 중위 순회는 항상 오름차순이다.

NoteBase

문 13. 다음 이진 트리에 대해 전위, 중위, 후위 순회(postorder traversal)를 수행할 때, 각 순회의 5번째 노드를 바르게 연결한 것은?



① 전위: E, 중위: H, 후위: E

② 전위: G, 중위: E, 후위: H

③ 전위: H, 중위: E, 후위: E

④ 전위: H, 중위: B, 후위: I

정답 3번

출제 포인트

이 문제는 이진트리의 전위, 중위, 후위 순회 규칙을 그림에 적용하여 각 순회의 5번째 방문 노드를 정확히 찾는지를 묻는 문제이다.

선지별 해설

① X: 전위를 E로 본 부분이 틀리다. 전위 순회 5번째 노드는 A, B, D, G 다음의 H이다. 틀린 표현은 '전위 E'이고, 올바르게는 '전위 H, 중위 H, 후위 E'가 아니라 '전위 H, 중위 E, 후위 E'이다. 또한 중위 5번째도 H가 아니라 E이다.

② X: 전위를 G로 본 부분이 틀리다. 전위 순회에서 G는 4번째이고 5번째는 H이다. 틀린 표현은 '전위 G'이고, 올바르게는 '전위 H'이다. 후위 H도 틀리다. 후위 순회에서 H는 2번째이고 5번째는 E이다.

③ O: 전위 5번째는 H, 중위 5번째는 E, 후위 5번째는 E로 모두 맞다. 세 순회를 직접 나열하면 선택지의 연결이 정확히 일치한다.

④ X: 중위를 B로 본 부분과 후위를 I로 본 부분이 틀리다. 중위 순회에서 B는 4번째이고 5번째는 E이다. 후위 순회에서 I는 4번째이고 5번째는 E이다. 틀린 표현은 '중위 B, 후위 I'이고, 올바르게는 '중위 E, 후위 E'이다.

NoteBase

오답 포인트

전위, 중위, 후위의 방문 규칙을 머릿속에서 섞으면 부모 노드와 자식 노드의 방문 시점이 한 칸씩 밀린다. 이 문항에서는 E, H, I처럼 인접한 위치의 노드를 5번째 후보로 착각하기 쉽다는 점이 함정이다.

다시 나오면 이렇게 판단

트리 순회 문제는 루트 방문 시점을 먼저 고정한 뒤 해당 규칙대로 실제 방문 순서를 끝까지 한 줄로 적어 본다. 그다음 묻는 번째 수만 세면 혼동이 줄어든다.

암기 포인트

전위는 루트 먼저, 중위는 루트 가운데, 후위는 루트 나중이라는 위치 규칙으로 기억한다.

NoteBase

문 14. 다음 C 코드를 사용하여 배열을 선언하였고 arr[0][0][0]의 주소를 a라고 할 때, arr[3][4][2]의 행 우선 순서 주소는?

```
char arr[6][5][4];
```

① a + 34

② a + 78

③ a + 96

④ a + 112

정답 2번

출제 포인트

이 문제는 3차원 배열의 행 우선 순서 주소 계산식으로 arr[3][4][2]의 오프셋을 구하는 문제이다.

조건 정리

배열은 char arr[6][5][4]이다. char형 원소의 크기는 1바이트이다. 기준 주소는 arr[0][0][0] = a이다. 구해야 할 값은 arr[3][4][2]의 행 우선 순서 주소이다.

사용할 공식

행 우선 순서에서 arr[i][j][k]의 오프셋은 $(i \times 5 \times 4) + (j \times 4) + k$이다. 마지막 첨자가 가장 빨리 변하고, 앞 첨자일수록 뒤 차원의 크기를 곱한 가중치를 가진다.

계산 과정

arr[3][4][2]의 오프셋은 $(3 \times 5 \times 4) + (4 \times 4) + 2$이다. 이를 계산하면 $60 + 16 + 2 = 78$이다. 따라서 주소는 a + 78이다.

NoteBase

선지별 해설

① X: 선택지의 잘못된 부분은 "a + 34"라는 오프셋 값이다. 이 값은 두 가중치를 뒤바꿔 (3 × 4) + (4 × 5) + 2 = 34처럼 i에는 4를, j에는 5를 곱했을 때 나오는 오답이다. 올바른 행 우선 가중치는 i에 5 × 4 = 20, j에 4이므로 정답은 "a + 78"이다.
② O: (3 × 5 × 4) + (4 × 4) + 2 = 78이므로 arr[3][4][2]의 주소는 a + 78이다. 행 우선 순서에서 마지막 차원 4가 바로 가중치로 쓰인다는 점까지 맞게 반영한 선택지이다.
③ X: 선택지의 잘못된 부분은 "a + 96"이라는 오프셋 값이다. 96은 각 차원의 가중치를 과대 계산한 값일 뿐, 실제 행 우선 계산 결과와 맞지 않는다. 올바르게는 "a + 78"이다.
④ X: 선택지의 잘못된 부분은 "a + 112"라는 오프셋 값이다. 이 값은 뒤 차원이 아니라 앞 차원의 크기를 가중치로 잘못 써서 i에 6 × 5 = 30, j에 5를 곱한 (3 × 6 × 5) + (4 × 5) + 2 = 112일 때 나오는 오답이다. 올바른 가중치는 자신보다 뒤 차원들의 곱이므로 정답은 "a + 78"이다.

오답 포인트

앞 첨자에 뒤 차원의 크기를 곱하는 가중치를 빠뜨리거나, 앞 차원의 크기(6, 5)를 가중치로 잘못 쓰면 오답이 된다. 특히 3차원 배열에서 첫 번째 첨자는 5 × 4, 두 번째 첨자는 4를 곱한다는 구조를 놓치면 숫자를 임의로 더한 오답으로 흐르기 쉽다.

다시 나오면 이렇게 판단

다차원 배열 주소 문제는 마지막 첨자가 가장 빨리 변하는 행 우선 순서인지 먼저 확인한 뒤, 앞 첨자부터 뒤 차원 크기를 차례로 곱해 오프셋을 만든다고 판단하면 된다.

암기 포인트

행 우선 주소는 앞 첨자일수록 뒤 차원들의 곱을 가중치로 가진다.

NoteBase

문 15. 다음은 정렬된 정수 배열에 대한 이진 탐색(binary search)을 구현한 C 코드이다. (가) ~ (다)에 들어갈 내용을 바르게 연결한 것은?

```
int BSearch(int data[], int size, int target) {
    int mid, left = 0, right = size - 1;
    while(left <= right) {
        mid = (가);
        if(target == data[mid]) return mid;
        else if(target > data[mid]) left = (나);
        else right = (다);
    }
    return -1;
}
```

① (가): (left+right) // 2, (나): mid-1, (다): mid+1

② (가): right-(left-right) // 2, (나): mid, (다): left

③ (가): (left+right) / 2, (나): mid+1, (다): mid-1

④ (가): left+(right-left) / 2, (나): right, (다): mid

정답 3번

출제 포인트

이 문제는 이진 탐색에서 중앙 인덱스 계산식과 비교 후 경계 갱신 규칙을 정확히 연결하는지를 묻는 문제이다.

핵심 개념

중앙 인덱스 계산: mid는 현재 탐색 구간 [left, right]의 가운데 인덱스이다 / 결정 차이축: mid가 항상 현재 구간 안에 있어야 한다 / 키워드: 가운데 인덱스

구간 축소: target이 data[mid]보다 크면 left를 mid+1로, 작으면 right를 mid-1로 옮겨 이미 비교한 mid를 구간에서 제외한다 / 결정 차이축: 비교가 끝난 mid를 다시 포함하는지 여부이다 / 키워드: mid 제외

NoteBase

선지별 해설

① X
(가)의 틀린 부분은 `(left+right) // 2`이다. C에서 `//`는 나눗셈 연산자가 아니라 주석 시작이므로 코드로 성립하지 않는다. 올바르게는 `(left+right) / 2` 또는 `left+(right-left)/2`이다.
또한 (나)와 (다)도 서로 뒤바뀌었다. `target > data[mid]`이면 오른쪽 절반을 탐색해야 하므로 `left = mid+1`이어야 하고, 그 반대이면 `right = mid-1`이어야 한다.

② X
(가)의 틀린 부분은 `right-(left-right) // 2`이다. 이 식은 가운데 인덱스를 계산하는 식이 아니고, 역시 `//`도 C 문법에 맞지 않는다. 올바르게는 `(left+right) / 2` 또는 `left+(right-left)/2`이다.
(나)의 틀린 부분은 `mid`이다. `left = mid`로 두면 이미 비교한 mid를 다시 포함해 구간이 줄지 않을 수 있다. 올바르게는 `left = mid+1`이다.
(다)의 틀린 부분은 `left`이다. 오른쪽 경계는 왼쪽 경계값이 아니라 `mid-1`로 줄여야 한다.

③ O
(가)의 `(left+right) / 2`는 C에서 정수 나눗셈으로 중앙 인덱스를 구하는 전형적인 식이다. `target > data[mid]`이면 왼쪽 경계를 `mid+1`로 옮기고, 그렇지 않으면 오른쪽 경계를 `mid-1`로 옮겨 이미 확인한 mid를 제외하므로 이진 탐색이 올바르게 진행된다.

④ X
(가)의 `left+(right-left) / 2` 자체는 오버플로를 줄이는 안전한 중앙 인덱스 계산식이므로 맞는 표현이다.
그러나 (나)의 틀린 부분은 `right`이다. `target > data[mid]`일 때 `left = right`로 두면 오른쪽 절반 전체를 제대로 남기지 못한다. 올바르게는 `left = mid+1`이다.
(다)의 틀린 부분은 `mid`이다. `right = mid`로 두면 이미 비교한 mid가 다시 포함되어 반복이 멈추지 않을 수 있다. 올바르게는 `right = mid-1`이다.

오답 포인트

④의 (가)처럼 중앙값 계산식만 보고 정답으로 고르기 쉽다. 그러나 이진 탐색은 mid 계산식과 함께 경계를 어떻게 줄이는지가 핵심이며, 비교한 mid를 다시 포함하면 무한 반복이나 잘못된 구간 축소가 발생한다.

다시 나오면 이렇게 판단

이진 탐색 코드를 보면 먼저 `mid`가 현재 구간의 가운데인지 확인하고, 다음으로 `target > data[mid]`이면 `left = mid+1`, `target < data[mid]`이면 `right = mid-1`처럼 비교한 mid를 반드시 제외하는지 확인하면 된다.

NoteBase

암기 포인트

이진 탐색은 가운데를 잡고, 비교가 끝난 mid는 버리므로 큰 쪽은 `left = mid+1`, 작은 쪽은 `right = mid-1`이다.

NoteBase

문 16. 다음 파이썬 프로그램의 실행 결과는?

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def func1(self, data):
        node = self.head
        self.head = Node(data, node)
    def func2(self, data):
        if self.head is None: self.func1(data)
        else :
            node = self.head
            while node.next is not None: node = node.next
            node.next = Node(data, None)
    def func3(self, pos):
        if pos < 0: return None
        node = self.head
        while pos > 0 and node != None:
            node = node.next
            pos -= 1
        return node
    def showList(self):
        node = self.head
        while node :
            print(node.data, end=' ')
            node = node.next

myList = SinglyLinkedList()
myList.func1(1); myList.func2(2)
myList.func1(3); myList.func2(4)
myList.func2(myList.func3(0).data)
myList.showList()
```

① 1 2 3 4

② 2 3 4 1

③ 3 1 2 4 3

④ 3 3 1 2 4

정답 3번

출제 포인트

이 문제는 단일 연결 리스트에서 머리 삽입, 꼬리 삽입, 0기반 위치 조회를 순서대로 추적해 최종 출력 순서를 판단하는 문제이다.

NoteBase

올바른 순서

1. `func1(1)`은 기존 head 앞에 1을 붙여 리스트를 `1`로 만든다.
2. `func2(2)`는 tail 뒤에 2를 붙여 리스트를 `1 2`로 만든다.
3. `func1(3)`은 다시 head 앞에 3을 붙여 리스트를 `3 1 2`로 만든다.
4. `func2(4)`는 tail 뒤에 4를 붙여 리스트를 `3 1 2 4`로 만든다.
5. `func3(0).data`는 현재 head 노드의 데이터인 3을 읽는다.
6. `func2(3)`은 읽은 3을 tail 뒤에 붙여 리스트를 `3 1 2 4 3`으로 만든다.

왜 이 순서인가: `func1`은 head를 바꾸는 앞삽입이고 `func2`는 마지막 노드 뒤에 붙는 뒤삽입이다. `func3(0)`은 현재 head의 값을 읽으므로 마지막 추가 값도 3이 된다.

선지별 해설

① X

`func1(3)`을 실행하는 순간 3은 맨 앞에 들어가므로 중간 리스트가 `1 2 3 4`처럼 이어지지 않는다.

틀린 표현: `1 2 3 4`

올바르게는: 중간 상태가 `3 1 2 4`가 되고, 마지막에 3이 한 번 더 뒤에 붙어 `3 1 2 4 3`이 된다.

② X

`func2`는 꼬리 삽입 함수이므로 2와 4가 앞쪽으로 이동하지 않는다. 마지막에 붙는 값도 현재 head의 데이터 3이지 1이 아니다.

틀린 표현: `2 3 4 1`

올바르게는: 순서는 `3 1 2 4 3`이다.

③ O

`func1` 두 번으로 head가 3이 되고, `func2` 두 번으로 2와 4가 차례로 tail에 붙는다. 이어서 `func3(0)`이 head 노드의 data인 3을 반환하므로 마지막에 3이 한 번 더 추가된다.

④ X

마지막 호출은 `func2(myList.func3(0).data)`이므로 현재 head 값 3을 tail에 추가하는 동작이다. 3이 맨 앞에 한 번 더 삽입되는 동작이 아니다.

틀린 표현: `3 3 1 2 4`

올바르게는: 마지막 3은 맨 뒤에 붙어서 `3 1 2 4 3`이 된다.

오답 포인트

오답의 핵심 함정은 `func1`과 `func2`의 삽입 방향을 뒤바꾸는 데 있다. 또 `func3(0)`을 처음 넣은 1로 착각하기 쉽지만, 위치 0은 항상 현재 head이므로 이 시점의 값은 3이다.

NoteBase

다시 나오면 이렇게 판단

연결 리스트 추적 문제는 각 함수가 head를 바꾸는지, tail 뒤에 붙는지, 값을 읽기만 하는지부터 구분한 뒤 매 호출마다 현재 리스트 상태를 한 줄로 갱신해 판단한다.

암기 포인트

`func1`은 앞삽입, `func2`는 뒤삽입, `func3(0)`은 현재 head 반환이라는 대응만 정확하면 최종 순서를 바로 복원할 수 있다.

NoteBase

문 17. 다음 C 프로그램의 실행 결과는?

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct ListNode {
   int value;
   struct ListNode* next;
} ListNode;

int func(ListNode* p, int n) {
   int i, sum = 0;
   for(i = 0; i < n; i++) {
      sum += p->value;
      p = p->next;
   }
   return sum;
}

int main() {
   int i, s, SIZE = 5;
   ListNode* p[SIZE];
   for(i = 0; i < SIZE; i++) {
      p[i] = (ListNode*)malloc(sizeof(ListNode));
      p[i]->value = i;
   }
   for(i = 0; i < SIZE-1; i++)
      p[i]->next = p[i+1];
   p[SIZE-1]->next = p[0];
   s = func(p[4], 7);
   printf("%d", s);

   return 0;
}
```

① 14

② 15

③ 16

④ 17

정답 1번

출제 포인트

이 문제는 원형 단일 연결 리스트를 따라 시작 노드에서 정해진 횟수만큼 이동하면서 누적합을 계산하는지를 묻는 문제이다.

조건 정리

각 노드의 value는 0, 1, 2, 3, 4이다.
연결은 0→1→2→3→4로 만든 뒤 마지막 노드 4의 next를 다시 0으로 연결하므로 원형 리스트가 된다.
func(p[4], 7)은 value가 4인 노드에서 시작하여 7번 반복하면서 현재 노드의 값을 sum에 더한 뒤 다음 노드로 이동한다.

NoteBase

계산 과정

더해지는 값의 순서는 4, 0, 1, 2, 3, 4, 0이다.
따라서 합은 4+0+1+2+3+4+0=14이다.
printf("%d", s);가 출력하는 값도 14이다.

선지별 해설

① O: 실제로 sum에 누적되는 값은 4, 0, 1, 2, 3, 4, 0이고 그 합은 14이므로 맞다.
② X: 틀린 표현은 출력값이 '15'라는 점이다. 올바르게는 시작 노드 4에서 7번 순회하면 마지막에 0까지 더해져 출력값은 '14'이다.
③ X: 틀린 표현은 출력값이 '16'이라는 점이다. 올바르게는 for문이 7번만 실행되므로 8개의 값을 더하지 않으며 결과는 '14'이다.
④ X: 틀린 표현은 출력값이 '17'이라는 점이다. 올바르게는 p[SIZE-1]->next = p[0] 때문에 4 다음 값이 다시 0이 되어 합은 '14'이다.

오답 포인트

원형 연결이라는 조건을 놓치면 4 다음에 더할 값을 잘못 상상하기 쉽다. 또한 for문의 반복 횟수 7회를 노드 개수 5와 섞어 세면 0이나 4를 한 번 덜 더하거나 더하게 되어 15, 16, 17 같은 오답이 나온다.

다시 나오면 이렇게 판단

연결 리스트 추적 문제는 시작 노드, next 연결 구조, 반복 횟수를 먼저 고정한 뒤 실제로 방문하는 값을 순서대로 적어 합을 계산하면 된다. 원형 연결이면 마지막 노드 다음에 어디로 돌아가는지를 반드시 다시 확인하게 된다.

암기 포인트

원형 리스트 추적은 '시작값 적기 → 방문 순서 적기 → 반복 횟수만큼만 더하기'로 정리된다.

NoteBase

문 18. 인접 행렬로 표현된 방향 그래프 중 위상 정렬이 불가능한 것은?

① bmatrix0 & 1 & 0 & 0 \ 0 & 0 & 1 & 0 \ 0 & 0 & 0 & 1 \ 0 & 0 & 0 & 0bmatrix

② bmatrix0 & 0 & 1 & 1 \ 0 & 0 & 1 & 1 \ 0 & 0 & 0 & 1 \ 0 & 0 & 0 & 0bmatrix

③ bmatrix0 & 1 & 1 & 1 \ 0 & 0 & 1 & 1 \ 0 & 0 & 0 & 1 \ 0 & 0 & 0 & 0bmatrix

④ bmatrix0 & 1 & 0 & 1 \ 0 & 0 & 1 & 0 \ 1 & 0 & 0 & 1 \ 0 & 0 & 0 & 0bmatrix

정답 4번

출제 포인트

이 문제는 인접 행렬에서 진입 차수 0인 정점이 계속 생기는지 확인하여 방향 그래프가 DAG인지, 즉 위상 정렬이 가능한지 판단하는 문제이다.

자료 읽는 순서

1. 각 행의 1을 간선으로 바꾸어 읽는다.
2. 순환이 있는지, 또는 진입 차수 0인 정점을 하나씩 제거할 수 있는지 본다.
3. 순환이 없으면 위상 정렬 가능, 순환이 있으면 위상 정렬 불가능으로 판단한다.

선지별 해설

① X: 1행 2열, 2행 3열, 3행 4열만 1이므로 간선은 1→2, 2→3, 3→4이다. 순환이 없는 DAG이어서 위상 정렬이 가능하다. 틀린 표현은 이 선택지를 '위상 정렬이 불가능한 것'으로 보는 점이고, 올바르게는 '위상 정렬이 가능한 그래프'이다.
② X: 1→3, 1→4, 2→3, 2→4, 3→4만 존재한다. 모든 간선이 앞 번호에서 뒤 번호로만 가므로 순환이 생기지 않는다. 틀린 표현은 이 선택지를 '위상 정렬이 불가능한 것'으로 보는 점이고, 올바르게는 '위상 정렬이 가능한 그래프'이다.
③ X: 1→2, 1→3, 1→4, 2→3, 2→4, 3→4로 읽힌다. 1, 2, 3, 4의 순서로 정렬할 수 있는 전형적인 DAG이다. 틀린 표현은 이 선택지를 '위상 정렬이 불가능한 것'으로 보는 점이고, 올바르게는 '위상 정렬이 가능한 그래프'이다.
④ O: 1행 2열이 1이어서 1→2, 2행 3열이 1이어서 2→3, 3행 1열이 1이어서 3→1이 된다. 따라서 1→2→3→1의 순환이 존재하므로 DAG가 아니고 위상 정렬이 불가능하다.

오답 포인트

행렬의 1을 개수만 보고 간선 방향을 놓치면 틀리기 쉽다. 특히 ④는 1→4, 3→4 때문에 끝점 4만 보고 단순한 DAG처럼 보이지만, 실제 함정은 1행 2열, 2행 3열, 3행 1열이 만드는 3정점 순환이다.

NoteBase

다시 나오면 이렇게 판단

인접 행렬 문제에서는 먼저 간선을 몇 개만 적어 보고, 진입 차수 0인 정점을 제거하는 과정이 끝까지 되는지 확인한다. 어느 단계에서라도 순환이 남아 진입 차수 0인 정점이 사라지면 위상 정렬은 불가능하다고 판단한다.

암기 포인트

위상 정렬 가능 조건은 DAG, 즉 순환이 없는 방향 그래프라는 점이다.

NoteBase

문 19. 다음 C 프로그램의 실행 결과는?

```
#include <stdio.h>
int func(int key, int low, int high, int* list) {
    list[high+1] = key;
    int i = 0;
    for(i = low; list[i] != key; i++);
    if(i == (high+1)) return -1;
    return i;
}

int main(int argc, char* argv[]) {
    int list[100] = {1, 2, 3, 4, 5, 6, 7};
    printf("%d\n", func(3, 4, 6, list));

    return 0;
}
```

① -1

② 0

③ 3

④ 4

정답 1번

출제 포인트

이 문제는 보초법을 사용한 순차 탐색 함수가 지정된 탐색 구간에서 key를 찾지 못할 때 어떤 값을 반환하는지를 추적하는 문제이다.

조건 정리

주어진 값은 key=3, low=4, high=6, list={1, 2, 3, 4, 5, 6, 7}이다.
구해야 할 값은 func(3, 4, 6, list)의 반환값이다.
주의할 조건은 함수가 배열 전체를 찾는 것이 아니라 인덱스 4부터 6까지만 찾는다는 점과, list[high+1]에 key를 넣어 보초를 세운다는 점이다.

NoteBase

사용할 원리

이 함수는 보초법 순차 탐색이다.
list[high+1] = key로 반복문의 종료를 보장한 뒤, for문이 high+1에서 멈추면 원래 탐색 구간 low~high 안에는 key가 없다고 보고 -1을 반환한다.

계산 과정

1. 호출 직후 list[7] = 3이 되어 보초가 세워진다.
2. for문은 i=4에서 시작하므로 list[4]=5, list[5]=6, list[6]=7을 차례로 확인한다.
3. 이 구간에는 3이 없으므로 i는 7까지 증가하고, list[7]=3인 보초에서 반복이 끝난다.
4. 반복 종료 뒤 i == high+1이 성립하므로 함수는 -1을 반환한다.

선지별 해설

① O: 반환값은 -1이다. 배열 안에 3이 있더라도 인덱스 2의 값은 low=4보다 앞에 있으므로 탐색 대상이 아니다. 실제 탐색 구간 4~6에서 찾지 못하고 보초 위치 7에서 멈추므로 실패를 뜻하는 -1이 반환된다.
② X: 0을 반환하는 문장은 코드에 없다. 올바르게는 지정된 구간에서 key를 찾지 못했으므로 -1을 반환한다.
③ X: 3은 찾으려는 key 값일 뿐 반환값이 아니다. 올바르게는 key를 원래 탐색 구간 4~6에서 찾지 못했으므로 -1을 반환한다.
④ X: 4는 탐색 시작 인덱스 low일 뿐 결과값이 아니다. 올바르게는 i가 7까지 진행한 뒤 실패 조건 i == high+1에 걸려 -1을 반환한다.

오답 포인트

배열 안에 3이 실제로 들어 있으므로 정답을 3이나 그 위치로 착각하기 쉽다. 그러나 이 함수는 low=4부터만 탐색하므로 인덱스 2의 3은 무시된다. 또한 list[high+1]에 넣은 3은 실제 데이터가 아니라 반복 종료용 보초라는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

탐색 코드 문제는 배열 전체를 보지 말고 시작 인덱스와 끝 인덱스를 먼저 확인한 뒤, 반복문의 시작값, 종료 조건, 보초값, 최종 반환 조건을 순서대로 추적하면 된다. 값이 배열 어딘가에 있는지보다 지정된 구간 안에 있는지가 먼저 판단 기준이 된다.

암기 포인트

보초법 순차 탐색은 high+1에서 처음 멈추면 원래 탐색 구간 실패이므로 -1을 반환한다.

NoteBase

문 20. 다음 수식을 후위 표기법(postfix notation)으로 변환한 후, 스택을 이용하여 계산하려고 한다. 계산 과정에서 스택에 일곱 번째로 삽입(push)하는 값은?

(A - B) * C - (D + E) / F

① C

② D

③ E

④ (D + E)

정답 3번

출제 포인트

이 문제는 중위식을 후위식으로 바꾼 뒤, 후위식 평가 과정에서 피연산자와 연산 결과가 스택에 어떤 순서로 push되는지를 묻는 문제이다.

올바른 순서

원식은 (A-B)*C-(D+E)/F 이고, 후위식은 AB-C*DE+F/- 이다.
중위식을 후위식으로 바꿀 때는 괄호 안과 연산자 우선순위가 높은 부분부터 "피연산자 두 개 다음에 연산자"로 옮긴다. 즉 (A-B)는 AB-, (D+E)는 DE+, *와 /는 -보다 우선순위가 높으므로 AB-C*와 DE+F/를 먼저 만든 뒤, 가장 마지막에 두 묶음을 -로 묶어 AB-C*DE+F/-가 된다.
push 순서는 1:A, 2:B, 3:(A-B), 4:C, 5:(A-B)*C, 6:D, 7:E, 8:(D+E), 9:F, 10:(D+E)/F, 11:(A-B)*C-(D+E)/F 이다.
연산자를 만나면 두 값을 pop한 뒤 계산 결과를 다시 push하므로, 피연산자만이 아니라 중간 연산 결과도 push 횟수에 포함된다.

선지별 해설

① X: C는 후위식에서 - 연산으로 (A-B)를 만든 뒤 들어가므로 4번째 push이다. 틀린 표현은 "일곱 번째가 C"라는 점이고, 올바르게는 "C는 네 번째로 push된다"이다.
② X: D는 앞부분 계산으로 (A-B)*C가 만들어진 다음 들어가므로 6번째 push이다. 틀린 표현은 "일곱 번째가 D"라는 점이고, 올바르게는 "D는 여섯 번째로 push된다"이다.
③ O: D가 6번째, E가 7번째로 push된다. 아직 +를 만나기 전이므로 이 시점에는 D와 E가 각각 따로 스택에 들어간다.
④ X: (D+E)는 D와 E를 push한 뒤 +를 만났을 때 계산되어 다시 push되는 값이므로 8번째 push이다. 틀린 표현은 "일곱 번째가 (D+E)"라는 점이고, 올바르게는 "(D+E)는 여덟 번째로 push된다"이다.

NoteBase

오답 포인트

이 문항의 함정은 push를 피연산자 입력만으로 세는지, 연산 결과 재삽입까지 포함해 세는지를 헷갈리게 하는 점이다. 정답 ③은 + 연산이 수행되기 직전까지 D와 E가 개별 값으로 push된다는 점을 잡아야 나온다.

다시 나오면 이렇게 판단

후위식 스택 문제는 후위식으로 먼저 바꾼 뒤, 토큰을 왼쪽부터 따라가며 피연산자는 즉시 push하고 연산자는 두 값을 pop하여 계산 결과를 다시 push하는 흐름으로 번호를 매기면 된다.

암기 포인트

후위식 평가는 피연산자 push, 연산 시 결과 재push의 반복이다.

NoteBase

문 21. 다음 C 프로그램의 출력 결과는?

```c
#include <stdio.h>
#include <stdlib.h>
void processArray(int* array, int size) {
    int i = 0;
    for(i = 0; i < size; i++) {
        array[i] += i * 2;
    }
}
void modifyArray(int* array) {
    *(array + 1) = 10;
    *(array + 3) = 20;
}
int main() {
    int i = 0;
    int sum = 0;
    int size = 5;
    int* array = (int*)malloc(size * sizeof(int));
    for(i = 0; i < size; i++) {
        array[i] = i;
    }
    processArray(array, size);
    modifyArray(array);
    for(i = 0; i < size; i++) {
        sum += array[i];
    }
    free(array);
    printf("Sum: %d\n", sum);
    return 0;
}
```

① Sum: 40

② Sum: 48

③ Sum: 50

④ Sum: 52

정답 2번

출제 포인트

이 문제는 동적 배열의 초기값을 함수 호출 순서대로 추적한 뒤 최종 합계를 계산하는 문제이다.

조건 정리

배열의 크기는 5이고 초기값은 `array[i] = i`이므로 `[0, 1, 2, 3, 4]`이다.
`processArray`는 각 원소에 `i * 2`를 더하므로 인덱스별 증가량은 `[0, 2, 4, 6, 8]`이다.
`modifyArray`는 두 번째 원소를 10으로, 네 번째 원소를 20으로 직접 덮어쓴다.

NoteBase

계산 과정

초기 배열은 `[0, 1, 2, 3, 4]`이다.
`processArray` 실행 후 배열은 `[0, 3, 6, 9, 12]`가 된다.
이후 `modifyArray`에서 `*(array + 1) = 10`, `*(array + 3) = 20`이므로 배열은 `[0, 10, 6, 20, 12]`가 된다.
최종 합계는 `0 + 10 + 6 + 20 + 12 = 48`이다.

선지별 해설

① X: 최종 합계는 40이 아니라 48이다. `modifyArray`가 일부 값을 더하는 것이 아니라 해당 위치 값을 10과 20으로 덮어쓴다는 점을 놓치면 오답이 된다. 틀린 표현 `Sum: 40` → 올바르게는 `Sum: 48`이다.
② O: `processArray` 후 `[0, 3, 6, 9, 12]`, `modifyArray` 후 `[0, 10, 6, 20, 12]`가 되므로 합계는 48이다.
③ X: 최종 합계는 50이 아니다. `modifyArray`에서 바뀐 두 위치만 보고 나머지 원소를 잘못 더하거나, `9`를 `12`로 잘못 읽으면 이런 값이 나온다. 틀린 표현 `Sum: 50` → 올바르게는 `Sum: 48`이다.
④ X: 최종 합계는 52가 아니다. `processArray`의 증가값과 `modifyArray`의 덮어쓰기를 동시에 더해 버리면 중복 계산이 생긴다. 틀린 표현 `Sum: 52` → 올바르게는 `Sum: 48`이다.

오답 포인트

`modifyArray`를 기존 값에 더하는 연산으로 착각하기 쉽다. 그러나 `*(array + 1) = 10`, `*(array + 3) = 20`은 해당 인덱스의 값을 새 값으로 대입하는 문장이다. 또한 포인터 표기 `*(array + k)`와 `array[k]`가 같은 위치를 가리킨다는 점을 놓치면 합계를 틀리기 쉽다.

다시 나오면 이렇게 판단

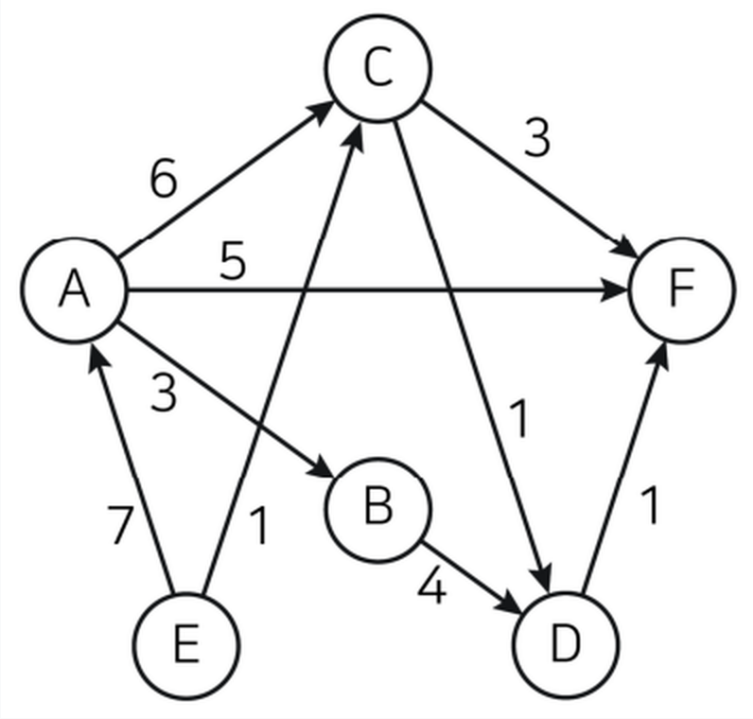
배열 추적 문제는 초기값을 먼저 적고, 함수가 호출되는 순서대로 배열 상태를 한 줄씩 갱신한 뒤 마지막 상태에서만 합계를 계산하면 된다. 대입문인지 누적문인지부터 구분하면 오답을 줄일 수 있다.

암기 포인트

`array[k]`와 `*(array + k)`는 같은 원소를 가리키며, `=`는 누적이 아니라 덮어쓰기라는 점이 핵심이다.

NoteBase

문 22. 다음은 정점 간의 거리를 표시한 방향 그래프이다. 경로의 길이와 상관없이 이동 가능한 모든 정점 사이의 최단 경로 합은?



① 57

② 58

③ 59

④ 60

정답 1번

출제 포인트

이 문제는 방향 가중치 그래프에서 도달 가능한 순서쌍만 골라 각 최단거리를 구한 뒤 그 합을 계산하는 문제이다.

조건 정리

정점은 A, B, C, D, E, F이다.
간선은 A→C=6, A→F=5, A→B=3, B→D=4, C→F=3, C→D=1, D→F=1, E→A=7, E→C=1이다.
구해야 할 값은 경로 길이와 무관하게 이동 가능한 모든 정점 순서쌍의 최단거리 합이다.
방향 그래프이므로 갈 수 없는 순서쌍은 합에서 제외하고, 자기 자신으로 가는 경우도 제외한다.

NoteBase

사용할 공식

각 시작 정점에서 도달 가능한 모든 도착 정점에 대해 최단거리만 합산한다.
최단거리는 직행 간선보다 더 짧은 우회 경로가 있으면 그 값으로 갱신한다.
이 원리를 쓰는 이유는 문제의 핵심이 모든 경로 길이의 합이 아니라 모든 도달 가능한 정점 사이의 최단 경로 합이기 때문이다.

계산 과정

A에서의 최단거리는 B=3, C=6, D=7, F=5이므로 합은 21이다.
B에서의 최단거리는 D=4, F=5이므로 합은 9이다.
C에서의 최단거리는 D=1, F=2이다. C→F의 직접 간선은 3이지만 C→D→F가 1+1=2이므로 더 짧다.
D에서의 최단거리는 F=1이므로 합은 1이다.
E에서의 최단거리는 A=7, B=10, C=1, D=2, F=3이다. 여기서 D는 E→C→D=2, F는 E→C→D→F=3으로 갱신되며, 합은 7+10+1+2+3=23이다.
따라서 전체 합은 21+9+3+1+23=57이다.

선지 판단

① O: 계산한 전체 최단거리 합이 57이므로 맞는 선지이다.
② X: 58은 최단경로 갱신 하나를 놓친 값이다. 대표적으로 C→F를 직접 간선 3으로 보면 1이 더해진다. 틀린 표현은 합이 58이라는 점이고, 올바르게는 57이다.
③ X: 59는 간접 경로 두 개를 놓친 값이다. 예를 들면 C→F를 3으로 보고 E→F도 4로 보면 2가 더해진다. 틀린 표현은 합이 59라는 점이고, 올바르게는 57이다.
④ X: 60은 여러 도달 쌍에서 직행 간선만 보거나 최단경로 갱신을 충분히 하지 않은 값이다. 틀린 표현은 합이 60이라는 점이고, 올바르게는 57이다.

오답 포인트

방향 그래프라는 점을 무시하거나, 직접 연결된 간선값만 보고 더 짧은 우회 경로를 갱신하지 않으면 오답이 된다. 특히 C→F는 3이 아니라 2이고, E→D와 E→F도 C와 D를 거치면 더 짧아진다.

다시 나오면 이렇게 판단

그래프 최단거리 합 문제는 방향 확인, 도달 가능한 순서쌍 정리, 각 쌍의 우회 경로 갱신, 시작 정점별 부분합 계산의 순서로 판단한다.

암기 포인트

최단거리 합은 보이는 간선의 합이 아니라 우회 경로까지 반영한 최소값들의 합이다.

NoteBase

문 23. data가 이중 연결 리스트(doubly linked list)에 다음과 같이 구성되어 있을 때, 데이터 A를 삭제한 후 변경된 내용으로 옳은 것은?

memory address	data	left_link	right_link
1000	A	1020	1030
1010	B	1030	NULL
1020	C	NULL	1000
1030	D	1000	1010

① B의 right_link → 1030, C의 left_link → 1020

② B의 right_link → 1030, D의 left_link → 1020

③ C의 right_link → 1020, D의 left_link → 1030

④ C의 right_link → 1030, D의 left_link → 1020

정답 4번

출제 포인트

이 문제는 이중 연결 리스트에서 특정 노드를 삭제할 때 앞 노드의 right_link와 뒤 노드의 left_link가 서로를 가리키도록 갱신되는지를 묻는다.

자료 읽는 순서

1. 표에서 각 노드의 left_link와 right_link를 따라 실제 연결 순서를 복원한다.
2. A(1000)의 이전 노드가 C(1020), 다음 노드가 D(1030)임을 확인한다.
3. 삭제 후에는 C의 right_link를 1030으로, D의 left_link를 1020으로 바꾸는지를 본다.

NoteBase

선지별 해설

① X: B는 원래 마지막 노드이므로 right_link는 NULL로 유지되고, C는 원래 첫 노드이므로 left_link는 NULL로 유지된다. 틀린 표현은 'B의 right_link → 1030'과 'C의 left_link → 1020'이다. 올바르게는 'B의 right_link → NULL', 'C의 left_link → NULL'이며, 삭제 후 변경되는 항목은 'C의 right_link → 1030', 'D의 left_link → 1020'이다.
② X: D의 left_link → 1020은 맞지만 B의 right_link → 1030은 틀리다. B의 right_link는 끝 노드이므로 NULL이어야 한다. 틀린 표현은 'B의 right_link → 1030'이고, 올바르게는 'B의 right_link → NULL'이다.
③ X: C의 right_link는 삭제되는 A를 건너뛰어 D의 주소 1030을 가리켜야 하므로 1020이 될 수 없다. D의 left_link도 자기 자신 쪽이 아니라 C의 주소 1020을 가리켜야 한다. 틀린 표현은 'C의 right_link → 1020, D의 left_link → 1030'이고, 올바르게는 'C의 right_link → 1030, D의 left_link → 1020'이다.
④ O: A의 왼쪽 노드 C와 오른쪽 노드 D를 직접 연결하면 C의 right_link는 1030, D의 left_link는 1020이 된다.

오답 포인트

삭제 대상 노드 A만 지우면 끝난다고 생각하면 틀린다. 이중 연결 리스트 삭제의 핵심은 A의 양옆 노드인 C와 D가 서로를 직접 가리키도록 링크를 다시 잇는 데 있다.

다시 나오면 이렇게 판단

이중 연결 리스트 삭제는 삭제 노드의 이전 노드와 다음 노드를 먼저 찾은 뒤, 이전 노드의 right_link를 다음 노드 주소로, 다음 노드의 left_link를 이전 노드 주소로 바꾸는 문제로 판단하면 된다.

암기 포인트

이중 연결 리스트 삭제는 앞노드의 right와 뒷노드의 left를 서로의 주소로 다시 연결하는 작업이다.

NoteBase

문 24. 다음 키값을 순서대로 삽입하여 B-트리를 구성할 때, 노드 오버플로(overflow)로 인한 노드 분할 횟수는? (단, 차수는 3이다)

1, 5, 7, 11, 13, 15, 17, 19

① 1
② 2
③ 3
④ 4

정답 4번

출제 포인트

이 문제는 차수 3인 B-트리에서 키를 순서대로 삽입할 때 오버플로로 발생하는 노드 분할 횟수를 누적해서 구하는 문제이다.

조건 정리

차수 3 B-트리는 한 노드가 최대 3개의 자식을 가지므로 한 노드에 저장될 수 있는 최대 키 수는 2개이다.
키가 3개가 되는 순간 오버플로가 발생하고 가운데 키가 부모로 올라가며 노드가 분할된다.
삽입 순서는 1, 5, 7, 11, 13, 15, 17, 19이다.

계산 과정

1과 5까지는 루트에 그대로 들어가므로 분할이 없다.
7을 넣으면 루트가 [1, 5, 7]이 되어 오버플로가 발생하고 [5]를 루트로 하는 분할 1회가 일어난다.
11은 오른쪽 리프 [7]에 들어가 [7, 11]이 된다.
13을 넣으면 오른쪽 리프 [7, 11, 13]이 오버플로가 되어 11이 부모로 올라가고 분할 1회가 추가된다. 누적 2회이다.
15는 오른쪽 리프 [13]에 들어가 [13, 15]가 된다.
17을 넣으면 오른쪽 리프 [13, 15, 17]이 오버플로가 되어 먼저 리프 분할 1회가 발생한다. 이때 올라간 15 때문에 루트가 [5, 11, 15]가 되어 다시 오버플로가 발생하므로 루트 분할 1회가 추가된다. 누적 4회이다.
19는 맨 오른쪽 리프에 들어가 [17, 19]가 되므로 추가 분할은 없다.

NoteBase

선지별 해설

① X: 분할 1회는 7을 삽입할 때의 첫 루트 분할만 센 값이다. 13 삽입 때 리프 분할 1회, 17 삽입 때 리프 분할 1회와 루트 분할 1회가 더 발생한다. 분할 횟수는 1회가 아니라 4회이다.
② X: 분할 2회는 7 삽입과 13 삽입까지만 센 값이다. 17 삽입에서 리프와 루트가 연쇄적으로 다시 분할된다. 분할 횟수는 2회가 아니라 4회이다.
③ X: 분할 3회는 리프 분할 3회만 세고 마지막 루트 분할 1회를 빠뜨린 값이다. 17 삽입 직후 루트 [5, 11, 15]도 오버플로가 되므로 한 번 더 분할된다. 분할 횟수는 3회가 아니라 4회이다.
④ O: 7 삽입 때 1회, 13 삽입 때 1회, 17 삽입 때 리프 분할 1회와 루트 분할 1회가 발생하므로 총 4회이다.

오답 포인트

17을 삽입할 때 오른쪽 리프 분할 뒤 가운데 키가 루트로 올라가면서 루트까지 오버플로가 된다는 점을 놓치기 쉽다. 리프 분할만 세고 부모의 연쇄 분할을 빼먹으면 3회로 오답 처리된다.

다시 나오면 이렇게 판단

B-트리 분할 횟수 문제는 각 삽입마다 도착한 리프의 키 수가 최대치를 넘는지 먼저 확인하고, 가운데 키 승격 뒤 부모 노드가 다시 최대치를 넘는지까지 연쇄적으로 추적하면 된다.

암기 포인트

차수 3 B-트리는 최대 자식 수가 3, 최대 키 수가 2이며 키가 3개가 되는 순간 분할된다고 기억하면 된다.

NoteBase

문 25. 다음 조건에서 데이터가 순서대로 해시 테이블에 입력된다고 할 때, 원래 계산된 버킷(bucket)에 저장되지 않는 데이터의 개수는? (단, 충돌(collision)로 인한 오버플로 발생 시, 선형 조사법(linear probing)으로 처리한다)

◯ 버킷 크기: 13
◯ 슬롯(slot) 크기: 2
◯ 해시 함수: h(k) = k mod 13

21, 33, 8, 19, 76, 58, 27, 83, 30, 44, 7, 73, 2, 66, 85

① 1
② 2
③ 3
④ 4

정답 2번

출제 포인트

해시 함수로 계산한 원래 버킷이 가득 찼을 때 선형 조사법으로 다음 버킷을 찾아 저장되는 데이터를 정확히 세는 문제이다.

조건 정리

버킷 크기는 13이고, 각 버킷의 슬롯 크기는 2이다. 해시 함수는 h(k)=k 13이며, 충돌로 오버플로가 발생하면 선형 조사법으로 다음 버킷을 차례로 확인한다. 입력 순서는 21, 33, 8, 19, 76, 58, 27, 83, 30, 44, 7, 73, 2, 66, 85이다.

사용할 공식

각 데이터의 원래 버킷은 k 13으로 계산한다. 한 버킷에는 2개까지 저장되므로, 이미 2개가 들어간 버킷에 다시 데이터가 들어오면 선형 조사법에 따라 다음 버킷 중 빈 슬롯이 있는 곳에 저장한다.

NoteBase

계산 과정

입력 순서대로 h(k)=k 13 값을 계산하고 슬롯이 2개씩인 버킷에 채워 나간다.
21 → 버킷 8(1번째), 33 → 버킷 7(1번째), 8 → 버킷 8(2번째, 버킷 8 가득 참), 19 → 버킷 6(1번째), 76 → 버킷 11(1번째), 58 → 버킷 6(2번째, 버킷 6 가득 참), 27 → 버킷 1(1번째), 83 → 버킷 5(1번째), 30 → 버킷 4(1번째), 44 → 버킷 5(2번째, 버킷 5 가득 참), 7 → 버킷 7(2번째, 버킷 7 가득 참).
여기까지 모든 데이터는 원래 계산된 버킷에 그대로 들어간다.
73은 73 13=8이지만 버킷 8(21, 8)이 이미 가득 차 있으므로 선형 조사로 다음 버킷 9의 빈 슬롯에 저장된다(원래 버킷 8이 아님).
2 → 버킷 2(1번째), 66 → 버킷 1(2번째, 버킷 1 가득 참)도 원래 버킷에 들어간다.
마지막 85는 85 13=7이지만 버킷 7(33, 7)이 가득 차 있고 다음 버킷 8(21, 8)도 가득 차 있어, 그다음 버킷 9의 남은 빈 슬롯(이미 73이 들어가 있음)에 저장된다(원래 버킷 7이 아님).
따라서 원래 계산된 버킷에 저장되지 않는 데이터는 73과 85, 모두 2개이다.

선지 판단

① 1: 원래 버킷에서 밀려난 데이터를 하나만 센 값이므로 틀리다.
② 2: 73과 85 두 개가 원래 계산된 버킷에 저장되지 않으므로 맞다.
③ 3: 73(버킷 8 가득)과 85(버킷 7·8 가득)에 더해, 충돌이 일어났지만 슬롯이 남아 원래 버킷에 정상 저장된 데이터(예: 버킷 7의 두 번째인 7, 또는 버킷 8의 두 번째인 8)까지 밀려난 것으로 잘못 세면 나오는 값이므로 틀리다. 충돌이 있어도 슬롯 2개 안에 들어가면 원래 버킷 저장이다.
④ 4: 원래 버킷에 저장되지 않은 데이터 개수가 아니라 충돌이 발생한 횟수를 센 값에 가깝다. 두 번째 슬롯에 채워질 때(8→버킷 8, 58→버킷 6, 44→버킷 5, 7→버킷 7, 66→버킷 1)와 다른 버킷으로 밀려날 때(73, 85)를 모두 충돌로 묶어 세는 식의 혼동에서 나오는 값이므로 틀리다. 실제로 원래 버킷을 벗어나 저장된 데이터는 73, 85뿐이다.

오답 포인트

충돌이 발생했다는 사실만으로 모두 원래 버킷 밖에 저장되는 것은 아니다. 슬롯 크기가 2이므로 같은 버킷에 두 개까지는 원래 버킷에 저장된다. 따라서 세 번째로 같은 버킷에 들어오는 데이터부터 밀려나는지 확인해야 한다.

다시 나오면 이렇게 판단

해시 테이블 문제는 입력 순서를 유지하면서 각 버킷의 현재 저장 개수를 갱신한다. 원래 버킷 번호, 실제 저장 버킷 번호를 나란히 적고 둘이 다른 경우만 센다.

암기 포인트

버킷당 슬롯이 2개이면 같은 해시값의 첫 두 데이터는 원래 버킷에 들어가고, 세 번째부터 선형 조사로 밀려난다.

NoteBase