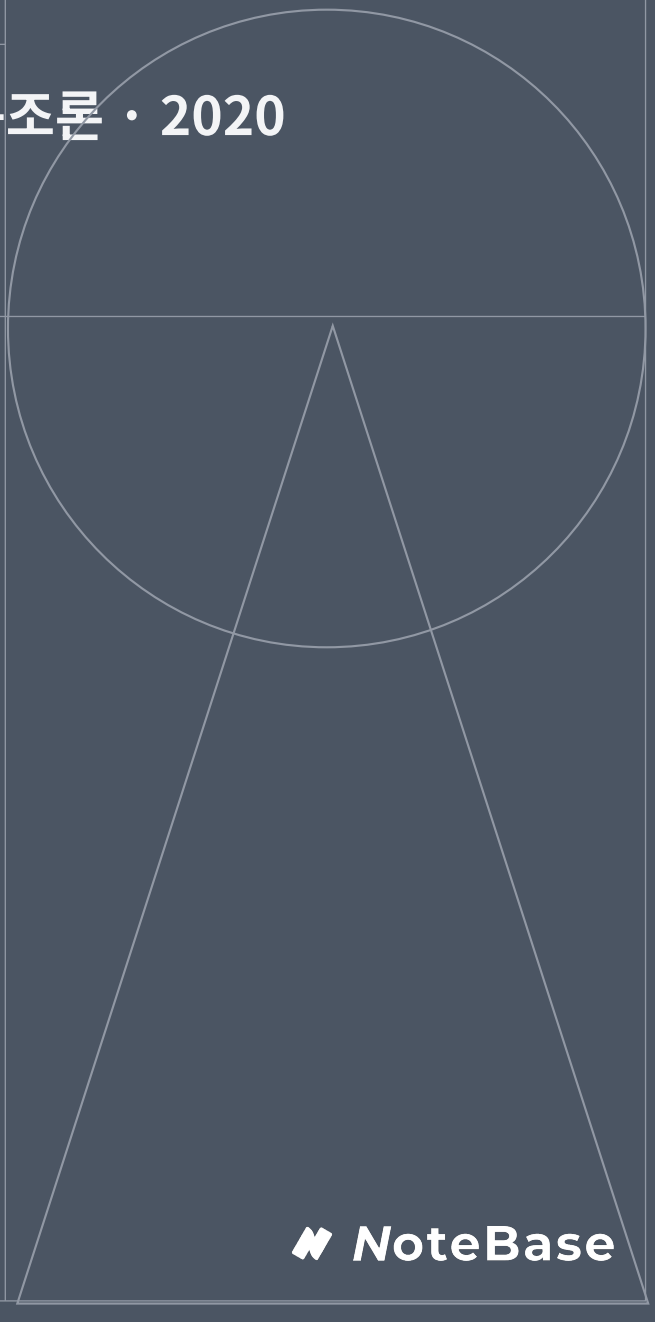
기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2020



NoteBase

기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2020

공무원/국가직/7급 · 자료구조론 · 2020

문 1. 다음의 D1 ~ D4에서 제시된 키(key) 값의 순서대로 공백(empty) 이진 탐색 트리(binary search tree)에 데이터를 삽입하여 T1 ~ T4를 만들었을 때, 모양이 다른 트리는? (단, 모양은 노드의 키 값과 간선의 연결 구조를 포함한다)

D1: 4, 2, 3, 6, 8, 5, 7 → 이진 탐색 트리 T1
D2: 4, 2, 6, 8, 5, 3, 7 → 이진 탐색 트리 T2
D3: 4, 6, 3, 8, 2, 5, 7 → 이진 탐색 트리 T3
D4: 4, 6, 8, 5, 7, 2, 3 → 이진 탐색 트리 T4

① T1
② T2
③ T3
④ T4

정답 3번

출제 포인트

이 문제는 이진 탐색 트리에 키를 삽입할 때 삽입 순서가 부모-자식 관계를 어떻게 바꾸는지 묻는 문제이다.

올바른 순서

1. 첫 번째 키 4가 루트가 된다.
2. 이후 키는 현재 노드보다 작으면 왼쪽, 크면 오른쪽으로 내려가 빈 자리에 삽입된다.
3. D1, D2, D4는 왼쪽 부분트리에서 2가 먼저 들어가고 3이 그 오른쪽 자식으로 붙어 왼쪽 모양이 같아진다.
4. D3는 3이 먼저 들어가고 2가 그 왼쪽 자식으로 붙어 왼쪽 부분트리 모양이 달라진다.

왜 이 순서: BST에서는 먼저 삽입된 키가 나중에 삽입되는 비교 대상의 조상이 되므로, 2와 3의 선후가 바뀌면 왼쪽 부분트리 구조가 바뀐다.

NoteBase

선지별 해설

① X: T1의 구조는 루트 4, 왼쪽 2-오른쪽 3, 오른쪽 6-왼쪽 5-오른쪽 8-왼쪽 7이다. D2와 D4도 같은 구조이므로 모양이 다른 트리가 아니다.
틀린 표현 → 올바르게는: "T1이 다르다"가 아니라 "T1은 T2, T4와 같은 모양이다."
② X: T2는 삽입 순서만 다를 뿐 2가 먼저 왼쪽 자식이 되고 3이 그 오른쪽 자식이 되어 T1, T4와 같은 모양이 된다. 따라서 다른 트리가 아니다.
틀린 표현 → 올바르게는: "T2가 다르다"가 아니라 "T2는 T1, T4와 같은 모양이다."
③ O: T3는 4의 왼쪽에 3이 먼저 들어간 뒤 2가 3의 왼쪽 자식으로 붙는다. 그래서 왼쪽 부분트리가 2-오른쪽 3 형태가 아니라 3-왼쪽 2 형태가 되어 다른 세 트리와 모양이 다르다.
④ X: T4는 오른쪽 부분트리를 먼저 만든 뒤 왼쪽에 2, 그 오른쪽에 3을 넣지만 최종 구조는 T1, T2와 같다. 삽입 시점이 늦어도 부모-자식 관계가 같으면 최종 모양도 같다.
틀린 표현 → 올바르게는: "T4가 다르다"가 아니라 "T4는 T1, T2와 같은 모양이다."

오답 포인트

많이 틀리는 지점은 삽입된 값의 집합만 보고 같은 트리라고 판단하는 것이다. 이 문제는 값이 같은지가 아니라 2와 3 중 누가 먼저 들어가 왼쪽 부분트리의 부모가 되는지를 봐야 한다.

다시 나오면 이렇게 판단

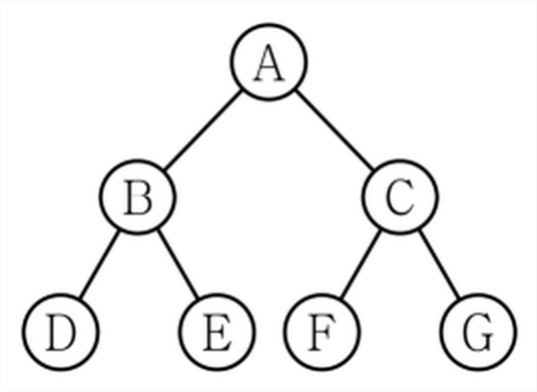
BST 모양 비교 문제는 루트부터 잡고, 각 부분트리에서 먼저 삽입되는 값이 그 부분트리의 부모가 된다는 점을 추적하면 된다. 특히 같은 두 값의 선후가 바뀌는 지점을 먼저 찾으면 정답이 빨리 갈린다.

암기 포인트

BST는 먼저 들어간 값이 위, 나중에 들어간 값이 아래에 놓인다. 2→3이면 2의 오른쪽 3, 3→2이면 3의 왼쪽 2이다.

NoteBase

문 2. 다음 최소 힙(min heap)에 대한 설명으로 옳지 않은 것은?



① 완전 이진 트리(complete binary tree)이다.

② 노드 A는 가장 작은 키 값을 가져야 한다.

③ 노드 B와 노드 C가 같은 키 값을 가질 수 있다.

④ 노드 C는 노드 D보다 작은 키 값을 가져야 한다.

정답 4번

출제 포인트

이 문제는 도식으로 주어진 최소 힙에서 완전 이진 트리 조건과 부모-자식 사이의 키 대소관계만으로 각 선지를 판단할 수 있는지를 묻는다.

선지별 해설

1번 O: 제시된 트리는 마지막 레벨이 왼쪽부터 빈칸 없이 채워진 형태이므로 완전 이진 트리이다.
2번 O: 최소 힙에서는 모든 부모의 키가 자식의 키보다 작거나 같으므로 루트 A는 전체 노드 중 가장 작은 키를 가진다.
3번 O: 형제 노드 B와 C 사이에는 대소 제약이 없으므로 같은 키 값을 가져도 최소 힙 조건에 어긋나지 않는다.
4번 X: 최소 힙은 부모와 자식 사이의 대소관계만 보장하므로 C와 D처럼 서로 다른 서브트리에 있는 노드 사이의 크기 관계는 정해지지 않는다. '노드 C는 노드 D보다 작은 키 값을 가져야 한다'가 아니라 '노드 C와 노드 D의 대소관계는 단정할 수 없다'가 옳다.

오답 포인트

힙을 전체 정렬된 트리로 착각하면 4번처럼 형제 서브트리 사이에도 순서가 정해진다고 오해한다. 최소 힙은 각 부모가 자기 자식보다 작거나 같다는 국소 조건만 만족한다.

NoteBase

다시 나오면 이렇게 판단

힙 문제는 먼저 완전 이진 트리인지 확인하고, 다음으로 부모-자식 간 비교만 적용한다. 조상·후손 관계가 아닌 두 노드의 대소를 단정하는 선지는 틀릴 가능성이 크다.

암기 포인트

최소 힙은 루트 최소, 부모는 자식 이하, 형제와 다른 서브트리끼리는 순서 보장이 없다는 구조이다.

NoteBase

문 3. 다음 인접 행렬(adjacency matrix)에 의해 표현되는 그래프에 대한 설명으로 옳은 것은? (단, 인접 행렬에서 [i]는 정점 i를 의미한다)

	[0]	[1]	[2]	[3]	[4]
[0]	0	1	0	0	0
[1]	0	0	1	1	0
[2]	0	0	0	0	1
[3]	0	0	0	0	1
[4]	0	1	0	0	0

① 간선의 개수가 5개인 무방향 그래프(undirected graph)이다.

② 정점 0에서 정점 4까지의 단순 경로(simple path) 길이는 2이다.

③ 정점 3에 도달하기 위한 경로가 없는 정점이 있다.

④ 사이클(cycle)이 존재하는 방향 그래프(directed graph)이다.

정답 4번

출제 포인트

이 문제는 인접 행렬의 각 행과 열을 읽어 방향 그래프의 간선 수, 경로 길이, 도달 가능성, 사이클 존재를 판단하는 문제이다.

자료 읽는 순서

1. 각 행에서 값이 1인 열을 찾아 정점에서 나가는 간선을 정리한다.
2. 1의 개수를 모두 세어 전체 간선 수와 방향 그래프 여부를 확인한다.
3. 정점 0에서 4까지의 경로와 정점 3으로의 도달 가능성, 1-2-4-1 같은 되돌아오는 경로가 있는지를 보기와 연결한다.

NoteBase

선지별 해설

1번: X. 행렬에서 1의 개수는 [0]행 1개, [1]행 2개, [2]행 1개, [3]행 1개, [4]행 1개로 총 6개이므로 간선은 6개이다. 또한 [0][1]은 1이지만 [1][0]은 0이므로 무방향 그래프가 아니라 방향 그래프이다. 틀린 표현은 '간선의 개수가 5개인 무방향 그래프'이고, 올바르게는 '간선의 개수가 6개인 방향 그래프'이다.
2번: X. 정점 0에서 4로 가려면 0→1→2→4 또는 0→1→3→4를 따라야 하므로 단순 경로 길이는 3이다. 틀린 표현은 '길이는 2'이고, 올바르게는 '길이는 3'이다.
3번: X. 정점 0은 0→1→3, 정점 1은 1→3, 정점 2는 2→4→1→3, 정점 4는 4→1→3으로 정점 3에 도달할 수 있다. 정점 3은 길이 0의 경로로 자기 자신에 도달한다. 따라서 정점 3에 도달하지 못하는 정점은 없다. 틀린 표현은 '경로가 없는 정점이 있다'이고, 올바르게는 '모든 정점에서 정점 3에 도달할 수 있다'이다.
4번: O. 1→2→4→1이 성립하므로 사이클이 존재한다. [1][2], [2][4], [4][1]이 모두 1인 것이 근거이다. 따라서 사이클이 존재하는 방향 그래프라는 설명이 옳다.

오답 포인트

인접 행렬에서 1의 개수를 대충 세거나, [i][j]와 [j][i]를 같은 것으로 보고 무방향 그래프로 착각하면 1번에서 흔들린다. 또 경로의 길이를 방문 정점 수로 세면 2번을 2로 착각하기 쉽다.

다시 나오면 이렇게 판단

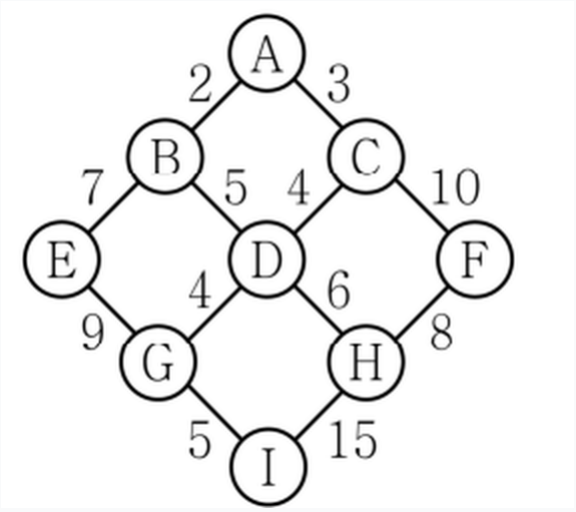
인접 행렬 문제는 먼저 각 1을 간선 하나로 바꾸어 적고, 그다음 대칭 여부로 방향성, 연속된 간선 수로 경로 길이, 출발점으로 되돌아오는 경로 존재로 사이클을 순서대로 판단한다.

암기 포인트

인접 행렬의 1 개수는 간선 수이고, 대칭이 아니면 방향 그래프이며, 출발점으로 돌아오면 사이클이다.

NoteBase

문 4. 다음 가중 그래프(weighted graph)에 Kruskal 알고리즘을 적용하여 구성한 최소 비용 신장 트리(minimum cost spanning tree)의 최소 비용은?



① 36

② 39

③ 44

④ 46

정답 2번

출제 포인트

이 문제는 가중 그래프에서 Kruskal 알고리즘으로 사이클을 만들지 않는 가장 작은 간선 8개를 골라 최소 신장 트리의 총비용을 구할 수 있는지를 묻는다.

조건 정리

주어진 값: 간선 가중치는 AB=2, AC=3, CD=4, DG=4, BD=5, GI=5, DH=6, BE=7, FH=8, EG=9, CF=10, HI=15이다.
구해야 할 값: 9개 정점을 모두 연결하는 최소 비용 신장 트리의 총비용이다.
주의할 조건: Kruskal 알고리즘은 가중치가 작은 순서대로 보되, 이미 연결된 정점끼리라 사이클이 생기는 간선은 제외한다.

NoteBase

사용할 공식

Kruskal 알고리즘: 가중치가 작은 간선부터 확인하여 사이클이 생기지 않으면 채택하고, 정점이 9개이면 간선 8개가 될 때까지 반복한다.
이 원리를 쓰는 이유: 문제에서 최소 비용 신장 트리를 직접 Kruskal 알고리즘으로 구성하라고 했으므로 간선 선택 순서와 사이클 제외 여부가 정답을 결정한다.

계산 과정

1. 가중치가 작은 순서로 AB(2), AC(3), CD(4), DG(4), BD(5), GI(5), DH(6), BE(7), FH(8), EG(9), CF(10), HI(15)를 본다.
2. AB, AC, CD, DG는 사이클이 없으므로 채택한다. 이어서 BD는 B와 D가 B-A-C-D 경로로 이미 연결되어 있어 사이클이 생기므로 제외한다.
3. GI, DH, BE, FH를 채택하면 9개 정점이 모두 연결되고 간선 수가 8개가 된다.
4. 선택된 간선은 AB, AC, CD, DG, GI, DH, BE, FH이고 총비용은 2+3+4+4+5+6+7+8=39이다.

선지 판단

1번 X: 36은 최소 비용 신장 트리로는 만들 수 없다. E를 연결하는 최소 간선이 7, F를 연결하는 최소 간선이 8이므로 다른 필수 간선들까지 합치면 총비용이 39보다 작아질 수 없다. 최소 비용은 36이 아니라 39이다.
2번 O: Kruskal 알고리즘으로 AB, AC, CD, DG, GI, DH, BE, FH를 선택하면 사이클 없이 모든 정점을 연결하고 총비용이 39가 된다.
3번 X: 44는 BD(5) 같은 사이클 간선을 잘못 포함했을 때 나올 수 있는 값이다. B와 D는 AB, AC, CD만으로 이미 연결되므로 BD는 제외해야 한다. 최소 비용은 44가 아니라 39이다.
4번 X: 46은 FH(8)보다 큰 CF(10)이나 BE(7)보다 큰 EG(9) 같은 간선을 불필요하게 택한 경우의 큰 값이다. Kruskal 알고리즘은 더 작은 비순환 간선을 먼저 택하므로 최소 비용은 46이 아니라 39이다.

오답 포인트

가중치가 작은 간선을 보는 것만으로 끝내면 틀린다. 특히 BD(5)는 값이 작아 보여도 이미 B-A-C-D 경로가 만들어진 뒤에는 사이클을 만들기 때문에 제외해야 하며, 이 검사를 놓치면 44 같은 오답이 나온다.

다시 나오면 이렇게 판단

최소 신장 트리 문제는 간선을 오름차순으로 정렬한 뒤 작은 간선부터 사이클 여부만 검사하면서 채택한다. 정점이 9개이면 간선이 8개가 되는 순간 합을 멈추면 된다.

암기 포인트

Kruskal 알고리즘의 핵심은 오름차순 선택, 사이클 제외, 간선 수 = 정점 수 - 1이다.

NoteBase

문 5. 키 값이 1, 2, 3, 4, 5, 6, 7, 8인 8개의 노드로 구성된 이진 탐색 트리를 전위 순회(preorder traversal)하면, 키 값 6, 2, 1, 4, 3, 5, 8, 7 순서로 노드를 방문한다. 이 트리를 후위 순회(postorder traversal)할 경우 방문 노드의 키 값을 방문 순서대로 바르게 나열한 것은?

① 1, 2, 3, 4, 5, 6, 7, 8

② 1, 3, 2, 6, 5, 8, 7, 4

③ 1, 3, 5, 4, 2, 7, 8, 6

④ 8, 7, 6, 5, 4, 3, 2, 1

정답 3번

출제 포인트

이 문제는 이진 탐색트리의 preorder 결과로 트리 구조를 복원한 뒤 postorder 방문 순서를 구하는 절차를 묻는 문제이다.

올바른 순서

1. 루트 결정 — preorder의 첫 원소 6이 전체 트리의 루트이다.
2. 서브트리 분리 — 6보다 작은 2, 1, 4, 3, 5는 왼쪽 서브트리이고 6보다 큰 8, 7은 오른쪽 서브트리이다.
3. 왼쪽 서브트리 복원 — 2가 루트이고 왼쪽 자식은 1, 오른쪽 자식은 4이며 4의 왼쪽 자식은 3, 오른쪽 자식은 5이다.
4. 후위 순회 적용 — 왼쪽 서브트리는 1, 3, 5, 4, 2이고 오른쪽 서브트리는 7, 8이므로 전체 postorder는 1, 3, 5, 4, 2, 7, 8, 6이다.

왜 이 순서: 후위 순회는 항상 왼쪽 서브트리, 오른쪽 서브트리, 루트 순서이므로 루트 6은 반드시 마지막에 와야 한다.

선지별 해설

① X: 오름차순 1, 2, 3, 4, 5, 6, 7, 8은 이진 탐색트리의 중위 순회 결과에 가깝고 후위 순회 결과가 아니다. 틀린 표현은 전체 수열이 후위 순회라는 점이다. 올바르게는 1, 3, 5, 4, 2, 7, 8, 6이다.

② X: 6이 중간에 나타나므로 후위 순회의 기본 규칙에 어긋난다. 틀린 표현은 루트 6을 마지막보다 앞에 둔 점과 왼쪽 서브트리 순서를 1, 3, 2로만 처리한 점이다. 올바르게는 왼쪽 서브트리가 1, 3, 5, 4, 2, 오른쪽 서브트리가 7, 8, 마지막 루트가 6이다.

③ O: 왼쪽 서브트리의 후위 순회가 1, 3, 5, 4, 2이고 오른쪽 서브트리의 후위 순회가 7, 8이므로 전체 결과는 1, 3, 5, 4, 2, 7, 8, 6이다.

④ X: 8, 7, 6, 5, 4, 3, 2, 1은 단순 역순 나열일 뿐 후위 순회와 무관하다. 틀린 표현은 오른쪽 서브트리의 루트 8과 전체 루트 6의 위치를 모두 앞쪽에 둔 점이다. 올바르게는 각 서브트리를 먼저 후위 순회한 뒤 마지막에 루트를 두어 1, 3, 5, 4, 2, 7, 8, 6이 된다.

NoteBase

오답 포인트

preorder와 postorder를 직접 뒤집어 생각하거나, 이진 탐색트리의 중위 순회가 오름차순이라는 사실을 후위 순회에도 잘못 적용하면 ① 같은 오답을 고르기 쉽다. 또한 후위 순회에서 루트가 항상 마지막이라는 규칙을 놓치면 ②처럼 루트 6을 중간에 둔 수열도 그럴듯해 보인다.

다시 나오면 이렇게 판단

BST의 preorder가 주어지면 첫 원소를 루트로 잡고, 그보다 작은 값 묶음과 큰 값 묶음으로 왼쪽·오른쪽 서브트리를 나눈다. 그다음 postorder는 각 서브트리를 왼쪽-오른쪽-루트 순으로 붙이면 된다.

암기 포인트

BST는 값의 대소로 서브트리를 가르고, preorder는 루트가 먼저 나오며 postorder는 루트가 마지막에 나온다.

NoteBase

문 6. 차수(m)가 3인 공백 B-트리에 19, 18, 2, 15, 6, 13, 7을 순서대로 삽입하여 구성한 B-트리에 대한 설명으로 옳은 것은?

① 트리의 높이(height)가 1이다.

② 15는 루트(root) 노드에 저장되어 있다.

③ 3개의 자식(child) 노드를 가지는 노드가 존재한다.

④ 6이 저장된 노드와 18이 저장된 노드는 형제(sibling) 관계이다.

정답 4번

출제 포인트

차수 3 B-트리에 키를 순서대로 삽입할 때 분할과 승격이 어떻게 일어나며, 최종 트리에서 루트·자식·형제 관계가 어떻게 정리되는지를 묻는 문제이다. 높이(height)와 형제(sibling)의 정의를 정확히 알고, 삽입 도중의 중간 상태가 아닌 최종 트리를 기준으로 판단해야 한다.

올바른 순서

1. 19를 넣으면 노드는 [19], 18을 넣으면 같은 노드에 [18, 19]가 된다.
2. 2를 넣으면 [2, 18, 19]로 키가 3개가 되어 오버플로가 발생하고 가운데 키 18이 승격된다. 결과: 루트 [18], 자식 [2], [19].
3. 15를 넣으면 15 < 18이므로 왼쪽 자식 [2]로 들어가 [2, 15]가 된다. 이어 6을 넣으면 6 < 18이므로 역시 왼쪽 자식으로 가 [2, 6, 15]가 되어 오버플로가 나고 가운데 키 6이 승격되며, 왼쪽 자식은 [2]와 [15]로 분할된다. 결과: 루트 [6, 18], 자식 [2], [15], [19].
4. 13을 넣으면 6 < 13 < 18이므로 가운데 자식 [15]로 들어가 [13, 15]가 된다. 이어 7을 넣으면 6 < 7 < 18이므로 역시 가운데 자식으로 가 [7, 13, 15]가 되어 오버플로가 나고 가운데 키 13이 승격되며, 가운데 자식은 [7]과 [15]로 분할된다. 이때 루트가 [6, 13, 18]로 키가 3개가 되어 다시 오버플로가 나므로 가운데 키 13이 한 번 더 승격되어 루트는 [13]이 되고, 그 자식은 [6]과 [18]로 분할된다.

결과(최종): 루트 [13] / 내부 노드 [6], [18] / 리프 [2], [7], [15], [19]. 이때 [6]의 자식은 [2], [7]이고 [18]의 자식은 [15], [19]이다.

왜 이 순서: 차수 3 B-트리는 한 노드가 최대 2개의 키만 저장하므로 3번째 키가 들어오는 순간마다 분할이 일어나고 가운데 키가 부모로 승격된다.

NoteBase

선지별 해설

① X. 틀린 표현은 "높이(height)가 1이다"이다. 최종 트리는 루트 [13] 아래에 내부 노드 [6], [18]이 있고 그 아래에 리프 [2], [7], [15], [19]가 있는 3개 층 구조이다. B-트리의 높이는 루트에서 리프까지의 간선 수(루트를 레벨 0으로 둘 때 리프의 레벨)로 정의하므로, 이 트리의 높이는 2이다. 따라서 높이가 1이라는 설명은 옳지 않다.
② X. 틀린 표현은 "15는 루트(root) 노드에 저장되어 있다"이다. 15는 루트가 아니라, 루트 [13]의 오른쪽 자식인 내부 노드 [18]의 왼쪽 리프에 저장되어 있다. 루트에는 최종 분할 후 13이 저장된다. 올바르게는 루트 노드에는 13이 저장되어 있다.
③ X. 틀린 표현은 "3개의 자식(child) 노드를 가지는 노드가 존재한다"이다. 중간 과정에서는 루트 [6, 18]이 자식 3개를 가지지만, 모든 삽입이 끝난 최종 B-트리에서는 루트 [13]의 자식이 2개이고 [6], [18]도 각각 자식이 2개씩이다. 올바르게는 최종 트리에는 3개의 자식을 가진 노드가 존재하지 않는다.
④ O. 형제(sibling) 노드란 같은 부모를 공유하는 노드를 말한다. 최종 트리에서 [6]과 [18]은 모두 루트 [13]의 바로 아래 자식이므로 같은 부모를 공유하는 형제 노드이다. 따라서 6이 저장된 노드와 18이 저장된 노드는 형제 관계라는 설명은 옳다.

오답 포인트

이 문항의 함정은 삽입 도중의 중간 상태를 최종 상태로 착각하는 데 있다. 특히 ③은 루트가 잠시 [6, 18]이 되면서 자식 3개를 갖는 순간이 있으므로 맞는 것처럼 보이지만, 7까지 삽입한 뒤 루트가 한 번 더 분할된 최종 결과를 기준으로 보면 틀린 선지이다.

다시 나오면 이렇게 판단

B-트리 선지는 각 단계의 중간 그림을 섞어 놓는 경우가 많으므로, 삽입이 끝날 때까지 "삽입 -> 오버플로 확인 -> 가운데 키 승격 -> 필요하면 루트 재분할" 순서로 끝까지 추적한 뒤 최종 트리만 기준으로 판단한다.

암기 포인트

차수 3 B-트리는 한 노드에 최대 2키, 최대 3자식만 가진다. 3번째 키가 들어오면 가운데 키가 승격된다고 기억하면 된다. 높이는 루트에서 리프까지의 간선 수(루트=레벨 0)로, 형제는 같은 부모를 공유하는 노드로 정의된다.

NoteBase

문 7. 스택을 이용하여 다음 후위 표현식(postfix expression)의 연산을 수행하였다. 최종 결과 값과 연산 과정에서 스택 내 피연산자의 최대 개수를 바르게 연결한 것은? (단, *은 곱셈 연산자이고 /은 나눗셈 연산자이다)

9 1 - 2 / 1 2 1 + * - 1 -

① 최종 결과 값: 0, 스택 내 피연산자의 최대 개수: 3

② 최종 결과 값: 0, 스택 내 피연산자의 최대 개수: 4

③ 최종 결과 값: 1, 스택 내 피연산자의 최대 개수: 3

④ 최종 결과 값: 1, 스택 내 피연산자의 최대 개수: 4

정답 2번

출제 포인트

후위표현식을 왼쪽부터 읽으며 연산자를 만날 때마다 직전 두 피연산자를 꺼내 계산하고, 동시에 스택 안 피연산자 수의 최댓값을 추적하는 문제이다.

조건 정리

식은 9 1 - 2 / 1 2 1 + * - 1 - 이다.
구해야 할 값은 최종 계산 결과와 계산 과정에서 스택 안에 들어 있던 피연산자 수의 최댓값이다.
숫자를 읽으면 스택에 넣고, 연산자를 읽으면 위의 두 값을 꺼내 계산한 뒤 결과를 다시 넣는다.

계산 과정

9, 1 입력 후 스택은 [9, 1]이므로 크기는 2이다.
- 적용 후 스택은 [8]이다.
2 입력 후 스택은 [8, 2]이고, / 적용 후 [4]가 된다.
1, 2, 1을 차례로 입력하면 스택은 [4, 1, 2, 1]이 되어 크기 최댓값은 4가 된다.
+ 적용 후 [4, 1, 3], * 적용 후 [4, 3], - 적용 후 [1]이 된다.
마지막으로 1을 넣어 [1, 1]이 되고, -를 적용하면 [0]이 된다.
따라서 최종 결과값은 0이고 스택 내 피연산자의 최대 개수는 4이다.

NoteBase

선지별 해설

① X: 최종 결과값 0은 맞지만 스택 내 피연산자의 최대 개수를 3으로 본 점이 틀리다. 1, 2, 1까지 입력한 직후 스택은 [4, 1, 2, 1]로 4개이므로 올바르게는 최종 결과값 0, 최대 개수 4이다.
② O: 최종 결과값이 0이고, 계산 중 스택 크기의 최댓값도 4이므로 두 조건이 모두 맞다.
③ X: 최종 결과값 1이 틀리다. [4, 3]에서 -를 적용한 중간 결과는 1이지만, 뒤의 1과 마지막 -를 더 수행해야 하므로 최종 결과는 0이다. 올바르게는 최종 결과값 0, 최대 개수 4이다.
④ X: 스택 내 피연산자의 최대 개수 4는 맞지만 최종 결과값 1이 틀리다. 마지막 1 - 연산까지 끝내면 값은 0이므로 올바르게는 최종 결과값 0, 최대 개수 4이다.

오답 포인트

중간 계산값 1을 최종 결과로 착각하거나, 연산 직후의 스택 크기만 세고 숫자를 연속으로 넣는 구간의 최대 크기를 놓치기 쉽다. 이 문항에서는 [4, 3]에서 나온 1이 최종값이 아니고, [4, 1, 2, 1] 구간이 최대 크기라는 점이 함정이다.

다시 나오면 이렇게 판단

후위표현식은 숫자를 만나면 스택에 넣고 연산자를 만나면 두 값을 꺼내 계산 결과를 다시 넣는 순서로 한 칸씩 추적한다. 이때 마지막에 남는 한 값은 최종 결과로 보고, 각 단계의 스택 크기 최댓값은 별도로 기록해 판단한다.

암기 포인트

후위표현식에서 숫자는 스택 크기를 1 늘리고 연산자는 스택 크기를 1 줄인다고 보면 최댓값과 최종값을 함께 추적할 수 있다.

NoteBase

문 8. 다음의 표는 단순 연결 리스트(singly linked list)를 표현한 것으로 각 행은 각 노드의 주소, 데이터 및 다음 노드 주소로 구성되어 있다. <다음 노드 주소 값>은 주소가 102인 노드를 삭제한 후 다음 노드 주소의 값을 설명한 것이다. ㉠~㉣에 들어갈 값을 바르게 연결한 것은? (단, 특정 노드의 다음 노드가 없을 때 그 노드의 다음 노드 주소 값은 NULL이다)

주소	데이터	다음 노드 주소
100	LEE	NULL
101	PARK	104
102	KIM	101
103	JUNG	102
104	SEO	100

<다음 노드 주소 값>

◯ 주소가 100인 노드의 다음 노드 주소는 (㉠)이다.

◯ 주소가 101인 노드의 다음 노드 주소는 (㉡)이다.

◯ 주소가 103인 노드의 다음 노드 주소는 (㉢)이다.

◯ 주소가 104인 노드의 다음 노드 주소는 (㉣)이다.

① ㉠: NULL, ㉡: 104, ㉢: 101, ㉣: 100

② ㉠: NULL, ㉡: 103, ㉢: 101, ㉣: 104

③ ㉠: 101, ㉡: 103, ㉢: 102, ㉣: 104

④ ㉠: 101, ㉡: 104, ㉢: 102, ㉣: 100

정답 1번

출제 포인트

이 문제는 표에 주어진 단순 연결 리스트의 연결 순서를 복원하고, 특정 노드 삭제 시 선행 노드의 링크만 갱신된다는 점을 판단하는 문제이다.

NoteBase

자료 읽는 순서

1. 각 행의 다음 노드 주소를 따라 실제 연결 순서를 복원한다.
2. 주소 102 노드의 선행 노드가 103임을 찾는다.
3. 102를 삭제한 뒤에는 103의 다음 노드 주소만 101로 바꾸고 나머지 노드의 링크는 유지한다.

선지별 해설

1번 O: 표에서 연결 순서는 103 → 102 → 101 → 104 → 100 → NULL이다. 주소 102 노드를 삭제하면 102를 가리키던 103의 다음 노드 주소만 101로 바뀌고, 주소 100·101·104 노드의 다음 노드 주소는 그대로이므로 ㉠=NULL, ㉡=104, ㉢=101, ㉣=100이 된다.
2번 X: ㉡을 103으로 둔 부분이 틀리다. 주소 101 노드의 다음 노드 주소는 삭제 전후 모두 104이다. 올바르게는 ㉡=104이다.
3번 X: ㉠을 101로 둔 부분이 틀리다. 주소 100 노드는 원래 마지막 노드이므로 삭제 후에도 다음 노드 주소가 NULL이다. 올바르게는 ㉠=NULL이다.
4번 X: ㉠을 101로 둔 부분과 ㉢을 102로 둔 부분이 틀리다. 주소 100 노드는 여전히 NULL을 가리켜야 하고, 주소 103 노드는 삭제된 102를 건너뛰어 101을 가리켜야 한다. 올바르게는 ㉠=NULL, ㉢=101이다.

오답 포인트

삭제되는 노드 자신의 정보만 지우고 선행 노드의 링크 갱신을 놓치면 틀린다. 특히 주소 103이 102를 가리키고 있다는 점을 놓치면 ㉢을 102로 남기거나, 관련 없는 101의 링크를 바꾸는 오답에 빠지기 쉽다.

다시 나오면 이렇게 판단

단순 연결 리스트 삭제 문제는 먼저 다음 노드 주소를 따라 전체 연결 순서를 복원한 뒤, 삭제 대상의 선행 노드를 찾아 그 노드의 링크를 삭제 대상의 후속 노드로 바꾸면 된다.

암기 포인트

단순 연결 리스트에서 중간 노드 삭제는 삭제 노드가 아니라 선행 노드의 next를 후속 노드로 바꾸는 작업이다.

NoteBase

문 9. 다음 C 언어 프로그램의 출력 결과는? (단, 배열 h의 시작 주소는 6487296이고, 자료형 int와 float는 4바이트의 크기를 가진다)

```
#include <stdio.h>

int main()
{
     struct human {
            char sextype;
            union t {
                   char children[3];
                   char beard[3];
            } u;
            int age;
            struct date {
                   int month;
                   int day;
                   int year;
            } dob;
            int personalID;
            float salary;
     } h[10];
     printf("%d, %d, %d", sizeof(struct human),
            &h[3].u.children[0] - &h[0].sextype,
            &h[5].u.beard[1] - &h[0].sextype);
}
```

① 28, 85, 142

② 28, 86, 143

③ 31, 94, 157

④ 31, 95, 158

정답 1번

출제 포인트

이 문제는 구조체의 메모리 배치와 정렬, 그리고 char 포인터 뺄셈 결과를 정확히 계산할 수 있는지를 묻는다.

조건 정리

주어진 값은 int와 float의 크기가 4바이트라는 점이다.
struct human의 필드는 sextype 1바이트, union u 3바이트, age 4바이트, dob 12바이트, personalID 4바이트, salary 4바이트이다.
union u는 char 배열만 가지므로 정렬 기준이 1바이트이고, age부터는 4바이트 경계에 맞춰 배치된다.
구해야 할 값은 sizeof(struct human)과 두 char 포인터의 차이이다.

NoteBase

사용할 공식

구조체 크기는 필드 배치 뒤 정렬 패딩까지 포함해 계산한다.
char 포인터의 차이는 두 주소의 바이트 차이와 같다.
이 원리를 쓰는 이유는 printf의 둘째, 셋째 값이 모두 char형 원소의 주소 차이이기 때문이다.

계산 과정

sextype는 오프셋 0에 놓인다.
union u는 크기 3바이트이므로 오프셋 1부터 3까지 차지한다.
age는 int이므로 오프셋 4에 놓인다.
dob는 int 3개이므로 오프셋 8부터 19까지 차지한다.
personalID는 오프셋 20, salary는 오프셋 24에 놓인다.
따라서 struct human의 전체 크기는 28바이트이다.
&h[3].u.children[0]의 오프셋은 3×28+1=85이다.
&h[5].u.beard[1]의 오프셋은 5×28+1+1=142이다.

선지 판단

1번: O. 계산 결과가 28, 85, 142이므로 정확하다.
2번: X. 둘째 값 86과 셋째 값 143이 각각 1씩 크다. union u의 시작 위치와 배열 원소 위치를 한 바이트씩 더 크게 잡은 오차이다. 올바르게는 28, 85, 142이다.
3번: X. 첫째 값 31이 틀렸다. struct human은 4바이트 정렬을 고려해도 28바이트이지 31바이트가 아니다. 따라서 뒤의 94와 157도 잘못된 구조체 크기를 기준으로 계산한 값이다. 올바르게는 28, 85, 142이다.
4번: X. 첫째 값 31이 틀렸고, 둘째 값 95와 셋째 값 158도 구조체 크기와 멤버 오프셋을 모두 잘못 계산한 값이다. 올바르게는 28, 85, 142이다.

오답 포인트

union의 크기만 보고 바로 다음 int가 이어진다고 생각하면 맞지만, 전체 구조체 크기를 31처럼 단순 합으로 착각하면 이후 모든 주소 계산이 무너진다. 또한 char 포인터 차이는 바이트 단위라는 점을 놓치면 1바이트 오차가 생긴다.

다시 나오면 이렇게 판단

구조체 계산 문제는 필드 크기를 순서대로 적고, 각 필드의 정렬 기준에 맞춰 오프셋을 배치한 뒤, 마지막에 포인터의 기준형이 무엇인지 확인해 차이 단위를 결정하면 된다.

암기 포인트

구조체 크기는 단순 합이 아니라 정렬 포함 총크기이며, char 포인터 뺄셈은 바이트 차이이다.

NoteBase

문 10. 다음 C 언어 프로그램의 출력 결과는?

```c
#include <stdio.h>

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

void recursive(char *list, int i, int n) {
     int j, temp;
     if (i == n) {
            for (j = 0 ; j <= n ; j++)
                  printf("%c", list[j]);
            printf(" ");
     }
     else {
            for (j = i ; j <= n ; j++) {
                  SWAP(list[i], list[j], temp);
                  recursive(list, i + 1, n);
                  SWAP(list[i], list[j], temp);
            }
     }
}

int main()
{
     char list[3] = {'a', 'b', 'c'};
     recursive(list, 0, 2);
}
```

① ab ac bc

② ab bc ca

③ abc acb bac bca cba cab

④ abc acb bca bac cab cba

정답 3번

출제 포인트

이 문제는 swap으로 문자의 자리를 바꾼 뒤 재귀 호출을 하고 다시 원상 복구하는 실행 순서를 따라 순열의 출력 순서를 묻는 문제이다.

올바른 순서

1. i=0, j=0에서 첫 글자를 a로 둔 채 recursive(list, 1, 2)를 호출한다.
2. i=1에서 j=1, 2를 돌며 abc, acb를 출력한다.
3. 원상 복구 뒤 i=0, j=1에서 첫 글자를 b로 바꾼 뒤 i=1에서 bac, bca를 출력한다.
4. 다시 원상 복구 뒤 i=0, j=2에서 첫 글자를 c로 바꾼 뒤 i=1에서 cba, cab를 출력한다.

왜 이 순서: 바깥 반복문이 첫 글자를 a, b, c 순으로 고정하고, 안쪽 재귀가 그 상태에서 남은 두 글자의 순열을 모두 출력하기 때문이다.

NoteBase

선지별 해설

① X
코드는 for (j = 0; j <= n; j++)로 세 문자를 모두 출력하므로 결과가 두 글자 조합인 ab ac bc가 될 수 없다.
틀린 표현: 출력 결과를 두 글자 조합으로 본 점이다.
올바르게는: 항상 세 글자 순열 6개가 출력된다.

② X
이 코드는 조합이 아니라 위치 교환을 이용한 순열 생성 코드이므로 ab bc ca처럼 인접한 두 글자만 나열하지 않는다.
틀린 표현: 인접 문자쌍만 나온다고 본 점이다.
올바르게는: abc acb bac bca cba cab 순으로 출력된다.

③ O
i=0에서 첫 글자를 a, b, c 순으로 고정하고, 각 경우마다 i=1에서 남은 두 글자의 위치를 바꾸어 출력하므로 abc acb bac bca cba cab 순이 된다.

④ X
abc acb까지는 맞지만, 첫 글자가 b로 고정된 구간에서는 bac가 bca보다 먼저 나오고, 첫 글자가 c로 고정된 구간에서는 cba가 cab보다 먼저 나온다.
틀린 표현: 각 첫 글자 블록 내부의 재귀 출력 순서를 뒤집은 점이다.
올바르게는: abc acb bac bca cba cab이다.

오답 포인트

이 문제의 함정은 재귀 호출 자체보다 swap 후 호출, 호출 뒤 복원 swap의 순서를 놓치는 데 있다. 또한 순열 생성 코드를 조합 생성으로 착각하면 ①, ②처럼 출력 길이부터 잘못 판단하게 된다.

다시 나오면 이렇게 판단

재귀 출력 문제는 현재 호출이 무엇을 고정하는지와 다음 호출이 무엇을 순회하는지를 분리해서 본다. 각 단계에서 swap 직후의 배열 상태, 재귀 진입 시점, 복원 후 상태를 한 줄씩 적으면 출력 순서를 안정적으로 확정할 수 있다.

암기 포인트

이 유형은 앞자리를 하나씩 고정한 뒤 남은 자리의 순열을 전개하는 구조로 기억한다.

NoteBase

문 11. 행 우선(row major) 순서로 저장되는 C 언어 2차원 배열 A[5][5]에서 다른 값을 갖는 것은? (단, 임의의 배열 인덱스(index) i(0 ≤ i ≤ 4)와 j(0 ≤ j ≤ 4)에 대해 A[i][j]가 (i - j)의 값을 가진다)

① A[3][1]

② *(*(A + 3) + 1)

③ *(A[3] + 1)

④ **((A + 3) + 1)

정답 4번

출제 포인트

이 문제는 2차원 배열 이름, 행 포인터 이동, 역참조가 최종적으로 어느 원소를 가리키는지를 묻는 문제이다.

핵심 개념

A는 첫 번째 행을 가리키는 포인터처럼 동작하므로 A+3은 네 번째 행 A[3]을 가리킨다.
A[3]은 식에서 사용되면 첫 원소의 주소 &A[3][0]으로 변환되므로 A[3]+1은 &A[3][1]이 된다.
반면 (A+3)+1은 행 포인터를 한 번 더 이동한 A+4이므로 최종 대상 행이 A[4]로 바뀐다.

선지별 해설

1번 X: A[3][1]은 바로 A[3][1] 자체이므로 값은 3-1=2이다. 틀린 판단은 '다른 값을 갖는다'이고, 올바르게는 '기준값 2를 나타낸다'이다.
2번 X: *(*(A + 3) + 1)은 A+3으로 A[3]행에 간 뒤, 그 행의 첫 원소에서 한 칸 이동하여 A[3][1]을 읽는 식이다. 값은 2이므로 틀린 판단은 '다른 값을 갖는다'이고, 올바르게는 'A[3][1]과 같은 값 2를 갖는다'이다.
3번 X: *(A[3] + 1)은 A[3]이 &A[3][0]으로 변환된 뒤 한 칸 이동하여 A[3][1]을 읽는 식이다. 값은 2이므로 틀린 판단은 '다른 값을 갖는다'이고, 올바르게는 'A[3][1]과 같은 값 2를 갖는다'이다.
4번 O: **((A + 3) + 1)은 먼저 A+4가 되어 다섯 번째 행 A[4]를 가리키고, 이를 두 번 역참조하면 A[4][0]을 읽는다. 값은 4-0=4이므로 앞의 2와 달라서 정답이다.

오답 포인트

*(A + 3) + 1과 (A + 3) + 1을 같은 단계로 보면 행 이동과 열 이동을 혼동하게 된다. 이 문제의 함정은 +1이 어느 포인터에 적용되는지에 따라 A[3][1]이 될 수도 있고 A[4][0]이 될 수도 있다는 점이다.

다시 나오면 이렇게 판단

배열 포인터 식은 괄호 안에서 먼저 행 이동인지 열 이동인지 구분한 뒤, 최종적으로 A[i][j] 꼴로 환원해서 값을 비교하면 된다.

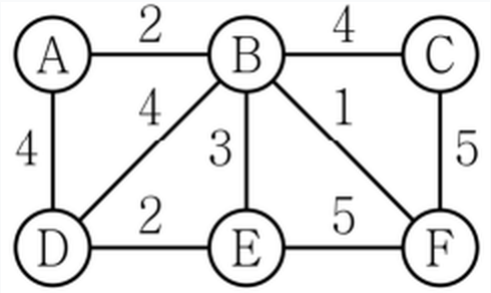
NoteBase

암기 포인트

행을 먼저 고르면 *(A+i)는 A[i]이고, 열까지 고르면 *(*(A+i)+j)는 A[i][j]이다. 행 포인터에 +1을 더하면 다음 행으로 이동한다.

NoteBase

문 12. 다음 가중치 그래프에 Prim 알고리즘을 적용하여 최소 비용 신장 트리(minimum cost spanning tree)를 만들 때, 최소 비용과 세 번째로 선택된 간선의 가중치의 합은? (단, A를 시작 정점으로 한다)



① 13

② 14

③ 15

④ 16

정답 3번

출제 포인트

이 문제는 가중치 그래프에서 Prim 알고리즘을 시작 정점 A부터 적용하여 매 단계의 최소 연결 간선과 최종 최소 신장 트리 비용을 추적할 수 있는지를 묻는다.

간선 선택 과정

시작 정점 A에서 가장 작은 간선은 A-B(2)이다.
다음에는 트리 {A, B}와 바깥 정점을 잇는 간선 중 가장 작은 B-F(1)을 선택한다.
그다음 트리 {A, B, F}와 바깥 정점을 잇는 간선 중 가장 작은 B-E(3)을 선택하므로 세 번째 간선의 가중치는 3이다.
이후 E-D(2), B-C(4)를 선택하면 최소 신장 트리의 총비용은 2+1+3+2+4=12이다.
따라서 최소 비용 12와 세 번째 간선의 가중치 3의 합은 15이다.

NoteBase

선지별 해설

1번 X: 13은 최소 비용 12에 세 번째 간선의 가중치를 1로 잘못 더한 값이다. 세 번째로 선택되는 간선은 B-F가 아니라 B-E이므로 올바른 값은 15이다.
2번 X: 14는 최소 비용 12에 두 번째 간선의 가중치 2를 섞거나 간선 선택 순서를 잘못 추적한 결과이다. 세 번째 간선의 가중치는 3이므로 올바른 값은 15이다.
3번 O: Prim 알고리즘을 A에서 시작하면 선택 순서는 A-B(2), B-F(1), B-E(3), E-D(2), B-C(4)이고 최소 신장 트리 비용은 12이다. 여기에 세 번째 간선의 가중치 3을 더하면 15가 된다.
4번 X: 16은 최소 신장 트리 비용이나 세 번째 간선의 가중치 중 하나를 크게 잡은 값이다. 이 그래프의 최소 신장 트리 비용은 12이고 세 번째 간선은 3이므로 올바른 값은 15이다.

오답 포인트

B-F의 가중치 1이 매우 작아서 이후 단계에서도 계속 먼저 선택된다고 착각하기 쉽다. Prim 알고리즘은 이미 트리에 포함된 정점 사이의 간선을 다시 고르는 방식이 아니므로, 각 단계마다 현재 트리와 바깥 정점을 잇는 간선만 비교해야 한다.

다시 나오면 이렇게 판단

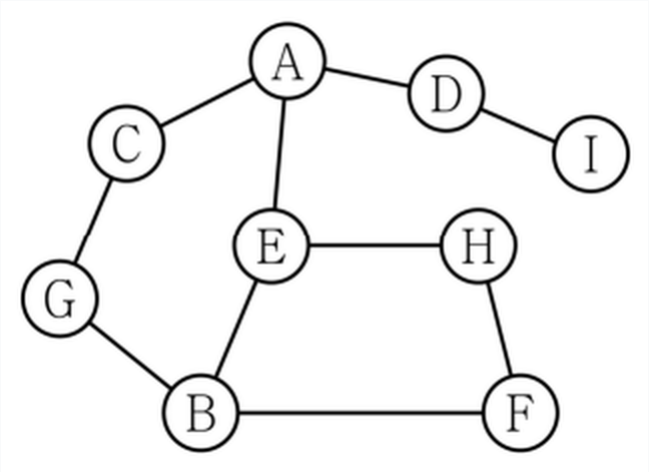
Prim 문제는 시작 정점을 기준으로 현재 트리에 포함된 정점 집합을 적어 두고, 매 단계마다 그 집합과 바깥 정점을 잇는 간선만 후보로 남겨 가장 작은 가중치를 선택하면 된다.

암기 포인트

Prim은 현재 트리 밖으로 나가는 최소 간선을 한 단계씩 붙이는 알고리즘이므로, 선택 순서는 전체 최소 간선 순서와 같지 않을 수 있다.

NoteBase

문 13. 다음 그래프에 대한 너비 우선 탐색(BFS: Breadth First Search)의 방문 순서는? (단, 시작 정점은 A이고, 인접한 정점들은 알파벳 순서로 방문한다)



① A, C, D, E, G, I, B, H, F

② A, C, D, E, I, G, H, B, F

③ A, C, D, G, B, E, F, H, I

④ A, C, D, G, I, B, E, H, F

정답 1번

출제 포인트

이 문제는 시작 정점 A에서 BFS를 수행할 때 큐에 먼저 들어간 정점이 먼저 방문된다는 원리와 인접 정점의 알파벳 순서 처리 규칙을 함께 적용할 수 있는지를 묻는다.

올바른 순서

방문 순서는 A, C, D, E, G, I, B, H, F이다.
1. A 방문 후 인접 정점 C, D, E를 알파벳 순서로 큐에 넣는다.
2. C를 방문하면 새 정점 G가 뒤에 추가된다.
3. D를 방문하면 새 정점 I가 뒤에 추가된다.
4. E를 방문하면 아직 방문되지 않은 B, H가 알파벳 순서로 추가된다.
5. 이후 큐 순서대로 G, I, B, H, F를 방문한다.
왜 이 순서인가: BFS는 먼저 발견된 정점을 큐의 앞에서부터 처리하므로 E에서 발견한 B와 H는 이미 큐에 들어가 있던 G, I보다 먼저 끼어들 수 없고, B에서 발견한 F도 H보다 앞설 수 없다.

NoteBase

선지별 해설

1번: O. A의 인접 정점 C, D, E를 먼저 큐에 넣고, 그다음 C에서 G, D에서 I, E에서 B와 H를 넣으면 방문 순서가 A, C, D, E, G, I, B, H, F가 된다. BFS의 큐 처리 순서와 정확히 일치한다.
2번: X. 틀린 표현은 I와 G의 순서이다. D는 C 다음에 방문되므로 D에서 발견한 I는 C에서 발견한 G보다 늦게 큐에 들어간다. 올바르게는 A, C, D, E, G, I, B, H, F이다.
3번: X. 틀린 표현은 E보다 G를 먼저 방문하고, I를 맨 뒤로 미룬 부분이다. BFS는 A의 인접 정점 C, D, E를 같은 레벨로 먼저 모두 처리하므로 E가 G보다 먼저 방문되어야 한다. 올바르게는 A, C, D, E, G, I, B, H, F이다.
4번: X. 틀린 표현은 G를 E보다 먼저 방문하고 I를 B보다 먼저 유지하지 못한 부분이다. G는 C를 처리한 뒤 큐에 추가된 정점이므로 이미 큐에 들어가 있던 D, E 다음에 방문된다. 또한 B는 E를 처리할 때, I는 D를 처리할 때 들어가므로 I가 B보다 먼저 방문된다. 올바르게는 A, C, D, E, G, I, B, H, F이다.

오답 포인트

DFS처럼 한 갈래를 깊게 따라가며 G나 B를 앞당기면 틀린다. 이 문제의 함정은 인접 정점의 알파벳 순서만 보고 판단하는 것이 아니라, 같은 레벨에서 먼저 큐에 들어간 정점이 누구인지를 끝까지 유지해야 한다는 점이다.

다시 나오면 이렇게 판단

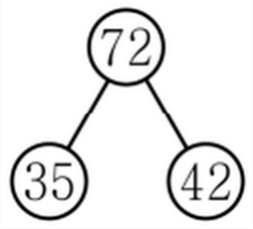
BFS 순서 문제는 현재 방문 정점의 인접 정점을 규칙대로 큐 뒤에 적고, 큐의 앞에서 빠지는 순서를 그대로 방문 순서로 옮기면 된다. 같은 레벨의 정점이 여러 개이면 발견 순서가 곧 방문 순서를 결정한다.

암기 포인트

BFS는 가까운 레벨부터, 같은 레벨 안에서는 먼저 큐에 들어간 정점부터 방문한다.

NoteBase

문 14. 다음에 주어진 최대 힙(max heap)에 키 값이 2, 14, 49, 3, 30, 41, 27, 17인 8개의 데이터를 순서대로 삽입하여 최대 힙을 구성하였다. 구성된 최대 힙에서 모든 리프(leaf) 노드의 키 값의 합은?



① 63

② 73

③ 108

④ 113

정답 3번

출제 포인트

이 문제는 초기 최대 힙에 데이터를 순서대로 삽입하면서 상향 재정렬이 어떻게 일어나는지 추적한 뒤 최종 리프 노드들의 합을 구하는 문제이다.

조건 정리

초기 최대 힙은 루트 72, 왼쪽 자식 35, 오른쪽 자식 42이다.
삽입 데이터는 2, 14, 49, 3, 30, 41, 27, 17의 순서이다.
최대 힙 삽입은 완전이진트리의 다음 빈자리에 넣고, 부모보다 크면 부모와 자리를 바꾸는 상향 재정렬로 처리한다.
리프 노드는 자식이 없는 노드이다.

사용할 원리

최대 힙에서는 새 원소를 마지막 자리에 넣은 뒤 부모보다 큰 동안 위로 올린다. 따라서 단순히 마지막에 들어간 값만 보는 것이 아니라, 각 삽입마다 위로 올라간 결과까지 반영한 최종 힙을 만들어야 한다.

NoteBase

계산 과정

초기 힙 배열은 [72, 35, 42]이다.
2 삽입 후 [72, 35, 42, 2]이다.
14 삽입 후 [72, 35, 42, 2, 14]이다.
49 삽입 후 부모 42와 교환되어 [72, 35, 49, 2, 14, 42]이다.
3 삽입 후 [72, 35, 49, 2, 14, 42, 3]이다.
30 삽입 후 부모 2와 교환되어 [72, 35, 49, 30, 14, 42, 3, 2]이다.
41 삽입 후 부모 30, 다시 35와 교환되어 [72, 41, 49, 35, 14, 42, 3, 2, 30]이다.
27 삽입 후 부모 14와 교환되어 [72, 41, 49, 35, 27, 42, 3, 2, 30, 14]이다.
17 삽입 후 [72, 41, 49, 35, 27, 42, 3, 2, 30, 14, 17]이다.
최종 리프 노드는 42, 3, 2, 30, 14, 17이므로 합은 108이다.

선지 판단

1번 X: 63은 최종 리프 합이 아니다. 41과 27의 상향 재정렬이 반영된 최종 리프 집합은 42, 3, 2, 30, 14, 17이므로 63이 아니라 108로 고치면 옳다.
2번 X: 73은 최종 리프 합이 아니다. 삽입 순서만 보고 중간 상태를 합치면 오답이 나오며, 최종 힙의 리프 합은 73이 아니라 108이다.
3번 O: 최종 힙 배열은 [72, 41, 49, 35, 27, 42, 3, 2, 30, 14, 17]이고 리프 노드의 합은 42 + 3 + 2 + 30 + 14 + 17 = 108이므로 옳다.
4번 X: 113은 최종 리프 합이 아니다. 내부 노드가 된 값을 리프로 잘못 포함하면 이런 오답이 나오기 쉽고, 올바른 값은 113이 아니라 108이다.

오답 포인트

이 문항의 함정은 삽입만 하고 끝내지 않고, 부모보다 큰 값이 위로 올라가는 상향 재정렬을 끝까지 추적해야 한다는 점이다. 특히 41이 30과 35를 지나 올라가고 27이 14 위로 올라가는 과정을 놓치면 최종 리프 집합을 잘못 잡게 된다.

다시 나오면 이렇게 판단

힙 삽입 문제는 배열 형태로 현재 상태를 적어 가면서 다음 빈칸 삽입, 부모와의 비교, 상향 교환, 최종 리프 확인의 순서로 판단하면 된다. 마지막에는 인덱스상 자식이 없는 노드만 다시 골라 합을 계산한다.

암기 포인트

최대 힙 삽입은 마지막 칸 삽입 후 부모보다 크면 위로 올리고, 리프 합은 최종 힙이 완성된 뒤 자식이 없는 노드만 더한 값이다.

NoteBase

문 15. <조건>을 만족하고 front와 rear의 값이 모두 0인 원형 큐(circular queue)에서 <동작>에 나열된 연산을 순서대로 모두 수행하였다. 연산 완료 후 원형 큐에 존재하는 모든 유효 데이터와 front 및 rear의 값을 바르게 연결한 것은? (단, enqueue(x)에 의해 삽입된 후 dequeue()에 의해 삭제되지 않은 데이터만 유효 데이터로 인정한다)

<조건>
◯ 원형 큐는 원소 개수가 6개인 1차원 배열로 구현되었다.
◯ front와 rear의 값이 같을 때 원형 큐는 공백 상태이다.
◯ front와 ((rear + 1) mod 6)의 값이 같을 때 원형 큐는 포화 상태이다.
◯ enqueue(x)에 의해 rear의 값을 ((rear + 1) mod 6)로 변경한 후 rear가 가리키는 위치에 데이터 x를 삽입한다. 단, 포화 상태일 때에는 데이터 삽입이 이루어지지 않아 원형 큐의 상태가 변하지 않는다.
◯ dequeue()에 의해 front의 값을 ((front + 1) mod 6)로 변경한 후 front가 가리키는 위치에서 데이터를 빼내어 삭제한다. 단, 공백 상태일 때에는 데이터 삭제가 이루어지지 않아 원형 큐의 상태가 변하지 않는다.
<동작>
enqueue(2) → enqueue(5) → enqueue(7) → enqueue(10) → dequeue() → enqueue(11) → enqueue(15) → enqueue(17) → dequeue() → dequeue() → enqueue(20)

① 모든 유효 데이터: 10, 11, 15, 17, 20, front: 3, rear: 1
② 모든 유효 데이터: 10, 11, 15, 20, front: 3, rear: 1
③ 모든 유효 데이터: 10, 11, 15, 17, 20, front: 4, rear: 2
④ 모든 유효 데이터: 10, 11, 15, 20, front: 4, rear: 2

정답 2번

출제 포인트

이 문제는 원형 큐의 공백·포화 조건과 enqueue, dequeue의 수행 순서에 따라 최종 유효 데이터와 front, rear 값을 추적할 수 있는지를 묻는다.

NoteBase

올바른 순서

초기 상태는 front=0, rear=0이다.
enqueue(2), enqueue(5), enqueue(7), enqueue(10)을 수행하면 rear가 4가 되고 유효 데이터는 2, 5, 7, 10이다.
dequeue()를 한 번 수행하면 front=1이 되고 2가 삭제되어 유효 데이터는 5, 7, 10이다.
enqueue(11), enqueue(15)를 수행하면 rear가 5, 0으로 이동하고 유효 데이터는 5, 7, 10, 11, 15가 되어 포화 상태가 된다.
포화 상태에서는 enqueue(17)이 수행되지 않으므로 큐 상태는 그대로 유지된다.
이후 dequeue(), dequeue()를 수행하면 front가 2, 3으로 이동하고 5, 7이 순서대로 삭제되어 유효 데이터는 10, 11, 15가 된다.
마지막으로 enqueue(20)을 수행하면 rear=1이 되고 최종 유효 데이터는 10, 11, 15, 20이다.
정답을 가르는 경계는 enqueue(17) 직전이 포화 상태라는 점과 dequeue가 두 번 일어나 front가 3이 된 뒤 enqueue(20)으로 rear가 1이 된다는 점이다.

선지별 해설

1번: X. 유효 데이터에 17이 포함된 부분이 틀렸다. enqueue(15) 직후 front=1, rear=0이므로 front와 ((rear + 1) mod 6)이 같아 포화 상태가 된다. 따라서 enqueue(17)은 무시되어야 하며 올바르게는 유효 데이터가 10, 11, 15, 20이고 front=3, rear=1이다.
2번: O. enqueue(17)이 포화 상태로 무시되고, 두 번의 dequeue로 5와 7이 삭제된 뒤 enqueue(20)이 rear=1 위치에 들어가므로 최종 상태는 유효 데이터 10, 11, 15, 20과 front=3, rear=1이 맞다.
3번: X. 유효 데이터에 17이 남아 있는 부분과 front, rear 값이 모두 틀렸다. 올바르게는 enqueue(17)이 실패하므로 17은 큐에 존재할 수 없고, 최종 포인터도 front=3, rear=1이다.
4번: X. 유효 데이터 자체는 맞지만 front와 rear가 틀렸다. 이 구현에서는 dequeue() 때 front를 먼저 한 칸 증가시키므로 두 번의 dequeue 후 front는 3이 되고, 마지막 enqueue(20) 후 rear는 1이 된다. 올바르게는 front=3, rear=1이다.

오답 포인트

가장 흔한 함정은 enqueue(15) 직후 이미 포화 상태가 되었다는 점을 놓쳐 17까지 삽입된다고 보는 실수와, dequeue에서 front가 언제 증가하는지를 반대로 기억해 최종 포인터를 4, 2로 계산하는 실수이다.

다시 나오면 이렇게 판단

원형 큐 추적 문제는 매 연산마다 먼저 공백·포화 여부를 판정하고, 그다음 포인터 변화와 실제 삽입·삭제된 데이터를 한 줄씩 기록하면 최종 상태를 안정적으로 판단할 수 있다.

NoteBase

암기 포인트

이 구현의 원형 큐는 포화면 삽입이 무시되고, enqueue는 rear를 먼저 이동한 뒤 저장하며 dequeue는 front를 먼저 이동한 뒤 삭제한다.

NoteBase

문 16. 키 값 1, 2, 3, 4, 5를 가지는 5개의 레코드로 이루어진 리스트를 퀵 정렬(quick sort)로 오름차순 정렬하고자 한다. 정렬 과정에서 총 비교 횟수가 가장 많은 리스트는? (단, 피봇(pivot)은 정렬하고자 하는 대상의 첫 번째 레코드로 선택한다)

① 1, 2, 3, 4, 5

② 2, 3, 1, 4, 5

③ 3, 1, 2, 4, 5

④ 3, 1, 2, 5, 4

정답 1번

출제 포인트

이 문제는 첫 번째 원소를 피벗으로 고정한 퀵 정렬에서 분할이 얼마나 불균형하게 반복되는지에 따라 전체 비교 횟수가 달라진다는 점을 묻는다.

조건 정리

- 레코드는 5개이다.
- 피벗은 매 단계에서 현재 부분리스트의 첫 번째 원소이다.
- 길이가 n인 부분리스트를 한 번 분할할 때 비교 횟수는 $n-1$회이다.

사용할 공식

전체 비교 횟수는 각 재귀 단계에서 발생하는 (현재 부분리스트의 길이 - 1)의 합이다.
이 원리를 쓰는 이유는 퀵 정렬의 분할 단계에서 피벗이 같은 부분리스트의 나머지 원소들과 각각 한 번씩 비교되기 때문이다.

계산 과정

1. ①은 [1, 2, 3, 4, 5]에서 피벗 1이 4회 비교되고, 이어 [2, 3, 4, 5]에서 3회, [3, 4, 5]에서 2회, [4, 5]에서 1회가 발생하여 총 10회이다.
2. ②는 [2, 3, 1, 4, 5]에서 4회 비교 후 [1]과 [3, 4, 5]로 나뉘고, 추가 비교는 0회와 2회, 다시 [4, 5]에서 1회가 더해져 총 7회이다.
3. ③은 [3, 1, 2, 4, 5]에서 4회 비교 후 [1, 2]에서 1회, [4, 5]에서 1회가 더해져 총 6회이다.
4. ④는 [3, 1, 2, 5, 4]에서 4회 비교 후 [1, 2]에서 1회, [5, 4]에서 1회가 더해져 총 6회이다.

NoteBase

선지별 해설

① O: 이미 오름차순인 상태라서 첫 원소 피벗이 매번 최솟값이 된다. 따라서 분할이 계속 한쪽으로만 치우치며 비교 횟수가 4+3+2+1=10회로 가장 많다.
② X: 첫 분할 뒤 [1]과 [3, 4, 5]로 나뉘어 한쪽은 바로 종료된다. 총 비교 횟수는 7회이므로 최대가 아니다. 틀린 부분은 '가장 많다'는 판단이며, 올바르게는 '①이 가장 많고 ②는 그보다 적다'이다.
③ X: 첫 피벗 3이 가운데 역할을 하여 [1, 2]와 [4, 5]로 비교적 고르게 갈라진다. 총 비교 횟수는 6회이므로 최대가 아니다. 틀린 부분은 '가장 많다'는 판단이며, 올바르게는 '③은 균형 분할에 가까워 비교 횟수가 더 적다'이다.
④ X: 첫 분할은 ③과 같은 구조이고 오른쪽 부분리스트 [5, 4]만 한 번 더 비교하면 끝난다. 총 비교 횟수는 6회이므로 최대가 아니다. 틀린 부분은 '가장 많다'는 판단이며, 올바르게는 '④도 ①보다 비교 횟수가 적다'이다.

오답 포인트

퀵 정렬의 평균 성능만 기억하고 이미 정렬된 입력이 빠를 것이라고 오해하기 쉽다. 그러나 이 문항은 피벗을 첫 번째 원소로 고정했으므로 이미 정렬된 리스트가 매 단계 최악의 분할을 만들어 비교 횟수가 가장 커진다.

다시 나오면 이렇게 판단

피벗 선택 방식부터 확인한 뒤 각 단계 비교 횟수를 '현재 부분리스트 길이 - 1'로 계산한다. 그다음 분할이 한쪽으로만 치우치는 입력인지, 양쪽으로 비교적 고르게 나뉘는 입력인지를 보면 최대 비교 횟수를 빠르게 판단할 수 있다.

암기 포인트

첫 원소 피벗 퀵 정렬에서는 이미 정렬된 입력이 최악 분할을 만들어 비교 횟수가 최대가 된다.

NoteBase

문 17. 다음의 키 값을 가지는 7개의 데이터를 순서대로 삽입하여 AVL 트리를 구성한 후 후위 순회를 수행하였다. 방문 노드들의 키 값을 방문 순서대로 바르게 나열한 것은?

10, 4, 7, 12, 3, 13, 15

① 3, 4, 10, 15, 13, 12, 7

② 3, 7, 4, 12, 15, 13, 10

③ 3, 7, 4, 15, 13, 12, 10

④ 3, 10, 13, 15, 4, 12, 7

정답 1번

출제 포인트

이 문제는 주어진 키들을 순서대로 삽입하면서 AVL 트리의 회전을 반영한 뒤 최종 트리의 후위 순회 결과를 묻는 문제이다.

올바른 순서

삽입 순서는 10 → 4 → 7 → 12 → 3 → 13 → 15이다.
10, 4, 7까지 삽입하면 4의 오른쪽에 7이 붙어 10에서 LR 불균형(왼쪽 자식의 오른쪽 방향)이 생긴다. 이때는 먼저 4를 축으로 왼쪽 회전한 뒤 10을 축으로 오른쪽 회전하는 LR 회전을 수행하며, 그 결과 7이 루트, 4가 왼쪽 자식, 10이 오른쪽 자식이 된다.
이어서 12는 10의 오른쪽 자식으로, 3은 4의 왼쪽 자식으로 삽입되며 불균형이 생기지 않는다.
13을 삽입하면 10 → 12 → 13이 오른쪽으로 이어져 10을 루트로 하는 부분트리가 RR 형태로 불균형이 되므로, 10을 축으로 왼쪽 회전하여 12를 그 부분트리의 루트로 올린다. 이때 12의 왼쪽 자식은 10, 오른쪽 자식은 13이 된다. 이 부분트리는 전체 트리에서 루트 7의 오른쪽에 위치한다.
15를 삽입하면 13의 오른쪽 자식이 되며, 경로상의 노드들(13, 12, 7)의 균형 인수가 모두 ±1 이내이므로 추가 회전은 일어나지 않는다.
따라서 최종 트리는 루트 7, 왼쪽 자식 4(그 왼쪽 자식 3), 오른쪽 자식 12(왼쪽 자식 10, 오른쪽 자식 13, 13의 오른쪽 자식 15)의 형태가 된다.
후위 순회는 왼쪽 부분트리 3, 4 → 오른쪽 부분트리 10, 15, 13, 12 → 루트 7 순서이므로 3, 4, 10, 15, 13, 12, 7이 된다.

NoteBase

선지별 해설

① O: 최종 AVL 트리의 후위 순회 결과 3, 4, 10, 15, 13, 12, 7과 정확히 일치한다. 왼쪽 부분트리 3, 4를 먼저 방문하고, 오른쪽 부분트리 10, 15, 13, 12를 방문한 뒤 마지막에 루트 7을 방문한다.
② X: 3, 7, 4, 12, 15, 13, 10은 전체 루트인 7이 두 번째 위치에 나온다는 점이 틀렸다. 후위 순회에서 루트 7은 가장 마지막에 방문해야 하고, 오른쪽 부분트리의 루트 12도 그 자식 10, 13, 15보다 뒤에 와야 한다. 올바른 결과는 3, 4, 10, 15, 13, 12, 7이다.
③ X: 3, 7, 4, 15, 13, 12, 10은 전체 루트 7이 두 번째 위치에 놓였고, 왼쪽 부분트리에 속하는 10이 맨 뒤에 나온다는 점이 틀렸다. 후위 순회에서는 왼쪽 부분트리 3, 4를 먼저 방문한 뒤 오른쪽 부분트리에서 10을 방문하고, 이어서 13의 오른쪽 자식 15와 13, 그다음 12, 마지막에 루트 7을 방문해야 한다. 올바른 결과는 3, 4, 10, 15, 13, 12, 7이다.
④ X: 3, 10, 13, 15, 4, 12, 7은 왼쪽 부분트리(3, 4)와 오른쪽 부분트리(10, 15, 13, 12)가 뒤섞여 있다는 점이 틀렸다. 3 다음에는 같은 왼쪽 부분트리의 4가 이어져야 하는데 오른쪽 부분트리의 10, 13, 15가 먼저 나온다. 후위 순회는 왼쪽 부분트리 전체를 먼저 방문하므로 4가 12보다 앞에 와야 한다. 올바른 결과는 3, 4, 10, 15, 13, 12, 7이다.

오답 포인트

삽입 순서와 순회 결과를 그대로 혼동하거나, 회전이 끝난 뒤의 최종 트리를 다시 그리지 않으면 오답이 나온다. 특히 ③처럼 13의 오른쪽 자식 15를 방문한 뒤 13, 12로 올라가는 후위 순회 규칙을 놓치기 쉽다.

다시 나오면 이렇게 판단

AVL 순회 문제는 삽입할 때마다 불균형이 생기는 지점을 바로 회전으로 고친 뒤 최종 트리를 다시 확정하고, 그다음 순회 규칙인 왼쪽 부분트리 → 오른쪽 부분트리 → 루트 순서만 적용하면 된다.

암기 포인트

AVL은 회전으로 최종 트리를 먼저 확정하고, 후위 순회는 좌우루트 순서로 읽는다.

NoteBase

문 18. 다음 C 언어 프로그램의 출력 결과는?

```
#include <stdio.h>

void func(char s[], int size) {
     if (size <= 0)       return;
     s[size-1] = s[size-1] + 1;
     func(s,size-2);
}

int main()
{
     char str[] = "window";
     func(str, 6);
     printf("%c", str[3]);
}
```

① d

② e

③ n

④ o

정답 2번

출제 포인트

이 문제는 재귀 함수가 문자열의 어느 인덱스를 어떤 순서로 바꾸는지를 추적해 최종 출력 문자를 판단하는 문제이다.

올바른 순서

1. `func(str, 6)`이 먼저 실행되어 `s[5]`가 `w`에서 `x`로 바뀐다.
2. 다음 호출 `func(str, 4)`에서 `s[3]`가 `d`에서 `e`로 바뀐다.
3. 다음 호출 `func(str, 2)`에서 `s[1]`이 `i`에서 `j`로 바뀐다.
4. `func(str, 0)`에서 종료한 뒤 `printf("%c", str[3]);`가 실행되어 `str[3]`인 `e`를 출력한다.

왜 이 순서인가: 함수는 현재 `s[size-1]`를 1 증가시킨 뒤 `size-2`로 재귀 호출하므로 접근 인덱스가 5, 3, 1 순으로 결정된다.

NoteBase

선지별 해설

① X: `d`는 원래 `str[3]`의 값일 뿐 최종 출력값이 아니다. `size=4` 호출에서 `s[3]`가 1 증가하므로 틀린 표현은 `d`이고 올바르게는 `e`이다.
② O: `size=4`일 때 `s[3]`가 `d`에서 `e`로 바뀌고, 마지막에 `printf`가 `str[3]`을 출력하므로 `e`가 맞다.
③ X: `n`은 `str[2]`의 값이다. 이 함수는 `size-1` 위치만 바꾸므로 실제로 수정되는 인덱스는 5, 3, 1뿐이며 2번 인덱스는 건드리지 않는다. 틀린 표현은 `n`이고 올바르게는 `e`이다.
④ X: `o`는 `str[4]`의 값이다. `str[4]`는 애초에 출력 대상도 아니고 재귀 호출에서 수정되는 위치도 아니다. 틀린 표현은 `o`이고 올바르게는 `e`이다.

오답 포인트

재귀 함수의 인자로 들어가는 `size`만 보고 문자 개수만 세면 오답이 나온다. 핵심 함정은 `s[size-1]`를 먼저 바꾸고 그다음 `size-2`로 내려간다는 점이며, 이 때문에 수정 위치가 5, 3, 1로 건너뛰어 결정된다.

다시 나오면 이렇게 판단

문자열 재귀 추적 문제는 `현재 호출의 수정 위치`와 `다음 호출의 인자 변화`를 먼저 적은 뒤, 실제로 출력하는 인덱스가 그 수정 목록에 포함되는지만 확인하면 된다.

암기 포인트

`s[size-1]`을 먼저 바꾸고 `size-2`로 내려가면 뒤에서부터 한 칸씩 건너뛴 인덱스만 수정된다.

NoteBase

문 19. 다음은 1차원 배열 arr에서 두 번째로 큰 수를 찾는 C 언어 함수이다. ㉠, ㉡에 들어갈 내용을 바르게 연결한 것은? (단, 배열 arr의 모든 원소는 서로 다른 양의 정수이고, n은 배열의 크기이며 2보다 크거나 같다)

```
void Largest2nd(int *arr, int n){
     int i, max1 = 0, max2 = 0;

     for (i = 0; i < n; i++){
          if (     ㉠     ){
               max2 = max1;
               max1 = arr[i];
          }
          else if (     ㉡     )
               max2 = arr[i];
     }

     printf("Second Largest Number = %d\n", max2);
}
```

① ㉠: arr[i] > max1, ㉡: arr[i] > max2 && arr[i] < max1

② ㉠: arr[i] > max1, ㉡: arr[i] < max2 && arr[i] > max1

③ ㉠: arr[i] < max1, ㉡: arr[i] > max2 && arr[i] < max1

④ ㉠: arr[i] < max1, ㉡: arr[i] < max2 && arr[i] > max1

정답 1번

출제 포인트

배열을 한 번 순회하면서 현재 최댓값 `max1`과 두 번째 최댓값 `max2`를 어떤 조건에서 갱신하는지 묻는 문제이다.

사례 정리

배열의 모든 원소는 서로 다른 양의 정수이다.
`max1`은 지금까지 본 값 중 가장 큰 값이고, `max2`는 그다음으로 큰 값이다.
새 원소 `arr[i]`가 `max1`보다 크면 기존 `max1`을 `max2`로 내리고 `arr[i]`를 새 `max1`로 올려야 한다.

NoteBase

핵심 개념

두 번째로 큰 수를 한 번의 순회로 찾으려면 최댓값과 차상위값을 함께 관리해야 한다.
새 값이 최댓값이면 `max1`과 `max2`가 함께 이동하고, 새 값이 최댓값은 아니더라도 `max2`보다 크고 `max1`보다 작으면 `max2`만 갱신된다.

적용 기준

㉠은 새로운 최댓값을 발견한 조건이어야 하므로 `arr[i] > max1`이어야 한다.
㉡은 최댓값은 아니지만 두 번째 최댓값 후보인 조건이어야 하므로 `arr[i] > max2 && arr[i] < max1`이어야 한다.

선지별 해설

① O: ㉠이 `arr[i] > max1`이므로 더 큰 값을 만나면 기존 `max1`을 `max2`로 옮기고 `max1`을 새 값으로 바꾼다. ㉡이 `arr[i] > max2 && arr[i] < max1`이므로 최댓값은 아니지만 두 번째로 큰 값 후보만 `max2`에 저장한다.
② X: ㉠은 맞지만 ㉡의 `arr[i] < max2 && arr[i] > max1`은 성립할 수 없는 조건이다. `max1`이 항상 `max2`보다 크므로 어떤 값도 동시에 `max2`보다 작으면서 `max1`보다 클 수 없다. 틀린 표현은 `arr[i] < max2 && arr[i] > max1`이고, 올바르게는 `arr[i] > max2 && arr[i] < max1`이다.
③ X: ㉡은 맞지만 ㉠의 `arr[i] < max1`은 더 작은 값을 만날 때마다 `max1`을 바꾸게 되어 최댓값 추적이 무너진다. 틀린 표현은 `arr[i] < max1`이고, 올바르게는 `arr[i] > max1`이다.
④ X: ㉠의 `arr[i] < max1`도 틀리고 ㉡의 `arr[i] < max2 && arr[i] > max1`도 틀리다. 이 조합은 최댓값 갱신 방향과 두 번째 최댓값 후보 조건을 모두 거꾸로 둔 경우이다. 올바르게는 ㉠이 `arr[i] > max1`, ㉡이 `arr[i] > max2 && arr[i] < max1`이다.

오답 포인트

이 문제를 틀렸다면 ㉠과 ㉡의 역할을 분리하지 못했을 가능성이 높다. ㉠은 새 최댓값 판단이고, ㉡은 최댓값과 두 번째 최댓값 사이에 들어오는 값 판단이다. 특히 ②처럼 `max1`과 `max2` 사이 부등호 방향을 뒤집으면 논리적으로 불가능한 조건이 된다.

다시 나오면 이렇게 판단

최댓값 두 개를 추적하는 코드는 먼저 새 값이 현재 최댓값을 넘는지 본다. 넘지 못하면 그다음에는 `max2`보다 크고 `max1`보다 작은 중간 구간에 들어오는지만 판단하면 된다.

암기 포인트

새 최댓값이면 기존 `max1`이 `max2`로 내려가고, 그렇지 않으면 `max2 < arr[i] < max1`일 때만 `max2`가 갱신된다.

NoteBase

문 20. 다음은 원형 연결 리스트(circular linked list)의 맨 앞에 insert_node가 지정하는 노드를 삽입하는 C 언어 함수이다. ㄱ, ㄴ에 들어갈 내용을 바르게 연결한 것은? (단, last_ptr은 마지막 노드를 지정하는 포인터 변수를 가리키는 주소 값을 가진다)

```
struct node {
     int data;
     struct node *link;
};
typedef struct node *node_ptr;

void InsertFront(node_ptr *last_ptr,
                  node_ptr insert_node){
     if (*last_ptr == NULL) {
          *last_ptr = insert_node;
          _____㉠_____
     }
     else {
          insert_node->link = (*last_ptr)->link;
          _____㉡_____
     }
}
```

① ㉠ insert_node = insert_node->link; ㉡ (*last_ptr) = insert_node;

② ㉠ insert_node = insert_node->link; ㉡ (*last_ptr)->link = insert_node;

③ ㉠ insert_node->link = insert_node; ㉡ (*last_ptr) = insert_node;

④ ㉠ insert_node->link = insert_node; ㉡ (*last_ptr)->link = insert_node;

정답 4번

출제 포인트

이 문제는 원형 연결 리스트에서 맨 앞 삽입을 할 때 빈 리스트와 비어 있지 않은 리스트의 포인터 연결 절차를 구분할 수 있는지를 묻는다.

NoteBase

올바른 순서

1. 빈 리스트이면 `*last_ptr = insert_node;`로 새 노드를 마지막 노드로 지정한다.
2. 빈 리스트이면 `insert_node->link = insert_node;`로 자기 자신을 가리키게 하여 원형을 만든다.
3. 비어 있지 않으면 `insert_node->link = (*last_ptr)->link;`로 새 노드가 기존 첫 노드를 먼저 가리키게 한다.
4. 마지막으로 `(*last_ptr)->link = insert_node;`로 마지막 노드의 링크를 새 노드로 바꾸어 새 노드를 맨 앞 노드로 만든다.

왜 이 순서인가: 기존 첫 노드를 잃지 않으려면 새 노드가 먼저 첫 노드를 가리킨 뒤 마지막 노드의 링크를 새 노드로 바꾸어야 원형 연결이 유지된다.

선지별 해설

1번: X. ㄱ의 `insert_node = insert_node->link;`는 새 노드의 링크를 설정하는 문장이 아니라 포인터 변수 `insert_node` 자체를 바꾸는 문장이라 빈 리스트의 자기 순환 연결을 만들지 못한다. ㄴ의 `(*last_ptr) = insert_node;`도 비어 있지 않은 경우 마지막 노드 포인터 자체를 새 노드로 옮겨 버려 맨 앞 삽입이 아니라 마지막 노드 갱신이 된다. 올바르게는 ㄱ `insert_node->link = insert_node;`, ㄴ `(*last_ptr)->link = insert_node;`이다.

2번: X. ㄴ의 `(*last_ptr)->link = insert_node;`는 비어 있지 않은 경우에 맞지만, ㄱ의 `insert_node = insert_node->link;`가 틀려서 빈 리스트를 원형으로 만들지 못한다. 올바르게는 ㄱ을 `insert_node->link = insert_node;`로 고쳐야 한다.

3번: X. ㄱ의 `insert_node->link = insert_node;`는 빈 리스트 처리로 맞다. 그러나 ㄴ의 `(*last_ptr) = insert_node;`는 마지막 노드를 새 노드로 바꾸는 동작이라 맨 앞 삽입 후에도 마지막 노드가 기존 마지막 노드로 유지되어야 하는 조건에 맞지 않는다. 올바르게는 `(*last_ptr)->link = insert_node;`로 마지막 노드의 링크만 새 노드로 바꾼다.

4번: O. ㄱ의 `insert_node->link = insert_node;`는 빈 리스트에서 새 노드 하나로 원형 연결을 만드는 처리이다. ㄴ의 `(*last_ptr)->link = insert_node;`는 새 노드를 기존 첫 노드 앞에 연결하는 처리이므로 맨 앞 삽입 절차에 맞는다.

오답 포인트

빈 리스트 처리에서 자기 자신을 가리키는 링크를 빠뜨리기 쉽고, 비어 있지 않은 경우에는 `last_ptr` 자체를 바꾸는 것과 `last_ptr`가 가리키는 마지막 노드의 `link`를 바꾸는 것을 혼동하기 쉽다. 맨 앞 삽입에서는 마지막 노드 포인터를 옮기지 않는다는 점이 함정이다.

다시 나오면 이렇게 판단

원형 연결 리스트 삽입 문제는 먼저 빈 리스트인지 여부를 나누고, 비어 있지 않으면 `마지막 노드 유지`, `새 노드가 기존 첫 노드를 먼저 가리킴`, `마지막 노드의 link만 새 노드로 변경`의 순서로 판단하면 된다.

NoteBase

암기 포인트

맨 앞 삽입은 `last`는 두고 `last->link`만 바꾸고, 빈 리스트는 `self-loop`로 시작한다.

NoteBase