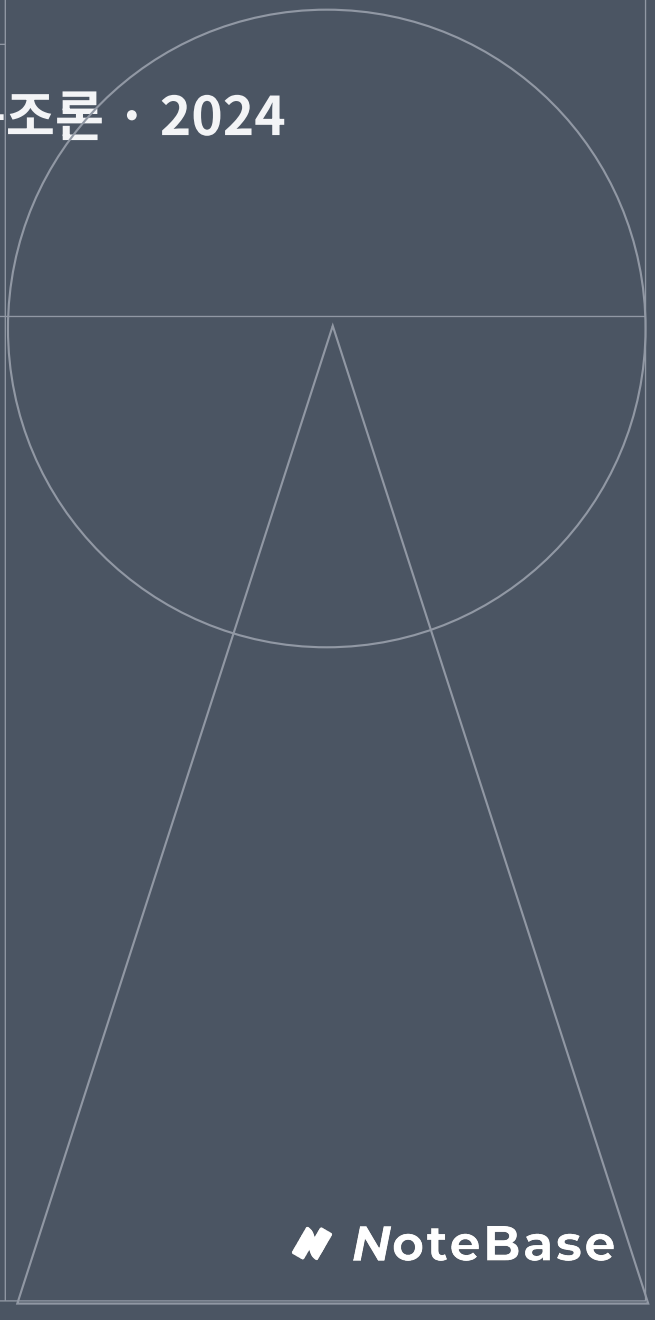
기출OX 해설집

공무원/서울시/7급 · 자료구조론 · 2024



NoteBase

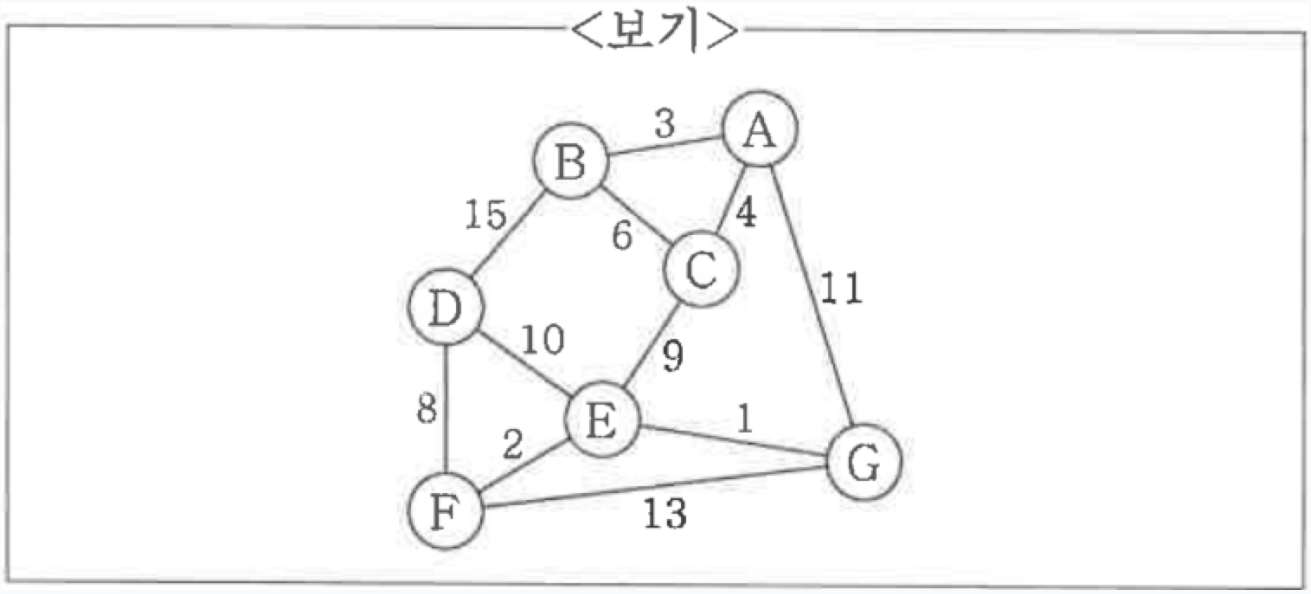
기출OX 해설집

공무원/서울시/7급 · 자료구조론 · 2024

공무원/서울시/7급 · 자료구조론 · 2024

문 1. <보기>의 그래프에서 가중치가 낮은 간선부터 삽입해 나가는 크루스칼(Kruskal) 알고리즘을 이용하여 최소 비용 신장 트리를 구성하고자 할 때, 5번째로 삽입되는 간선의 가중치는?

<보기>



① 8

② 9

③ 10

④ 11

정답 1번

출제 포인트

크루스칼(Kruskal) 알고리즘으로 최소 비용 신장 트리(MST)를 구성할 때 간선이 삽입되는 순서를 추적하는 문제이다. 단순히 MST를 완성하는 것이 아니라 'N번째로 삽입되는 간선'을 묻는다는 점에서 사이클 검사 과정을 단계별로 정확히 따라가야 한다.

조건 정리

노드 7개(A, B, C, D, E, F, G), 간선 11개. 각 간선의 가중치: E-G=1, E-F=2, A-B=3, A-C=4, B-C=6, D-F=8, C-E=9, D-E=10, A-G=11, F-G=13, B-D=15. MST 완성에는 노드 수 - 1 = 6개의 간선이 필요하다.

NoteBase

사용할 공식

크루스칼 알고리즘: 모든 간선을 가중치 오름차순으로 정렬한다. 가장 작은 간선을 선택하여 사이클이 형성되면 건너뛰고, 사이클이 없으면 MST에 삽입한다. 사이클 판정은 Union-Find로 두 끝 노드가 같은 집합에 속하는지 확인한다. 삽입된 간선 수가 노드 수 - 1이 되면 완성된다.

계산 과정

가중치 오름차순 정렬: E-G(1), E-F(2), A-B(3), A-C(4), B-C(6), D-F(8), C-E(9), D-E(10), A-G(11), F-G(13), B-D(15).

1번째: E-G(1) — 사이클 없음 — 삽입. 집합: {E,G} / {A} / {B} / {C} / {D} / {F}
2번째: E-F(2) — 사이클 없음 — 삽입. 집합: {E,G,F} / {A} / {B} / {C} / {D}
3번째: A-B(3) — 사이클 없음 — 삽입. 집합: {E,G,F} / {A,B} / {C} / {D}
4번째: A-C(4) — 사이클 없음 — 삽입. 집합: {E,G,F} / {A,B,C} / {D}
(건너뜀) B-C(6) — B와 C가 이미 같은 집합 {A,B,C} — 사이클 형성 — 건너뜀
5번째: D-F(8) — D={D}, F∈{E,G,F} — 사이클 없음 — 삽입. 집합: {D,E,G,F} / {A,B,C}

5번째로 삽입되는 간선은 D-F이고 가중치는 8이다.

선지 판단

① 8 — 5번째로 삽입되는 간선 D-F의 가중치이다. (O)
② 9 — C-E의 가중치로, D-F(8)보다 크다. C-E는 D-F 다음인 6번째로 삽입된다. (X)
③ 10 — D-E의 가중치이다. D-E는 C-E보다도 크고, C-E 삽입 후 D와 E가 이미 같은 집합이 되어 사이클로 건너뛰어진다. (X)
④ 11 — A-G의 가중치이다. A-G는 정렬 순서상 9번째 후보이며, 삽입 시점에 A와 G가 이미 같은 집합에 속해 사이클로 건너뛰어진다. (X)

오답 포인트

B-C(6)를 5번째로 착각하기 쉽다. 그러나 3번째(A-B)와 4번째(A-C) 삽입으로 이미 A, B, C가 같은 집합에 속해 있으므로 B-C는 사이클을 형성한다. 건너뜀 처리 후 그다음 후보인 D-F(8)가 실제 5번째 삽입 간선이 된다. 삽입 횟수와 후보 선택 횟수를 혼동하지 않도록 주의한다.

다시 나오면 이렇게 판단

간선 정렬 후 Union-Find 집합을 단계별로 추적한다. 'N번째 삽입'을 물으면 사이클로 건너뛴 간선은 카운트에 포함시키지 않고 실제로 MST에 추가된 간선만 번호를 센다.

NoteBase

암기 포인트

크루스칼 알고리즘 3단계: 가중치 오름차순 정렬 → 사이클 검사(Union-Find) → 삽입 또는 건너뜀. MST 완성 조건은 삽입된 간선 수 = 노드 수 - 1이다. 사이클 판정은 두 끝 노드가 같은 집합인지 여부로 결정된다.

NoteBase

문 2. 배열로 표현한 이진 탐색 트리에 자연수를 저장할 때, 낭비되는 저장 공간이 가장 적은 입력 순서는?

① 6 - 7 - 12 - 19 - 3 - 22 - 17

② 12 - 19 - 17 - 6 - 3 - 7 - 22

③ 3 - 6 - 7 - 12 - 17 - 19 - 22

④ 22 - 19 - 17 - 12 - 7 - 6 - 3

정답 2번

출제 포인트

이 문제는 이진 탐색 트리를 배열로 표현할 때 삽입 순서가 트리의 모양을 어떻게 바꾸고, 그 모양이 배열 인덱스의 낭비 공간을 얼마나 키우는지를 묻는 문제이다.

올바른 순서

1. 첫 원소를 루트로 둔다.
2. 이후 원소를 이진 탐색 트리 규칙에 따라 작은 값은 왼쪽, 큰 값은 오른쪽에 삽입한다.
3. 이렇게 만들어진 트리의 각 노드를 배열 인덱스 규칙인 루트 1, 왼쪽 자식 $2i$, 오른쪽 자식 $2i + 1$에 대응시킨다.
4. 사용한 노드 수보다 가장 큰 인덱스가 얼마나 커지는지를 비교하면 낭비 공간을 판단할 수 있다.

왜 이 순서: 먼저 삽입 순서로 트리 모양이 결정되고, 그다음 그 모양이 배열 인덱스를 결정하므로 트리 구조를 보지 않고는 낭비 공간을 비교할 수 없다.

선지별 해설

① X: 6이 루트가 되면 7-12-19가 연속해서 오른쪽으로 내려가고, 19의 자식으로 17과 22가 붙는다. 배열 인덱스는 6이 1, 7이 3, 12가 7, 19가 15, 17이 30, 22가 31이 되어 가장 큰 인덱스가 31이다. 노드는 7개뿐이므로 빈 칸이 매우 많이 생긴다. 틀린 표현은 최소라는 판단이며, 올바르게는 깊이를 고르게 만드는 입력이 최소가 된다.

② O: 12를 루트로 하고 6과 19를 좌우에 둔 뒤 3, 7, 17, 22가 각각 그 아래를 채우면 완전이진트리 형태가 된다. 배열 인덱스가 1부터 7까지 빈칸 없이 채워져 낭비 공간이 0이므로 가장 적다.

③ X: 오름차순 입력이라 모든 노드가 오른쪽 자식으로만 이어진다. 인덱스가 1, 3, 7, 15, 31, 63, 127까지 커져 낭비 공간이 가장 크다. 틀린 표현은 최소라는 판단이며, 올바르게는 오름차순 입력은 오른쪽 편향 트리를 만들어 낭비 공간을 크게 만든다.

④ X: 내림차순 입력이라 모든 노드가 왼쪽 자식으로만 이어진다. 인덱스가 1, 2, 4, 8, 16, 32, 64가 되어 역시 낭비가 크다. 다만 ③의 오른쪽 편향보다 최댓값 인덱스가 작으므로 최소가 아니다. 틀린 표현은 최소라는 판단이며, 올바르게는 내림차순 입력도 낭비가 크지만 ②보다 훨씬 비효율적이다.

NoteBase

오답 포인트

③과 ④를 둘 다 편향 트리로 보고 같은 정도의 비효율이라고 처리하면 틀리기 쉽다. 배열 표현에서는 왼쪽 자식이 $2i$, 오른쪽 자식이 $2i + 1$이므로 같은 높이의 편향이라도 오른쪽으로만 내려가는 ③이 더 큰 인덱스를 만든다. 또한 노드 수만 세지 말고 가장 큰 인덱스를 함께 봐야 낭비 공간을 정확히 비교할 수 있다.

다시 나오면 이렇게 판단

배열 표현 BST 문제는 입력 순서대로 트리 모양을 먼저 그린 뒤, 루트 1, 왼쪽 $2i$, 오른쪽 $2i + 1$ 규칙으로 몇 개의 깊은 인덱스가 생기는지를 비교하면 된다. 균형에 가까우면 유리하고, 정렬된 입력처럼 한쪽으로 치우치면 불리하다.

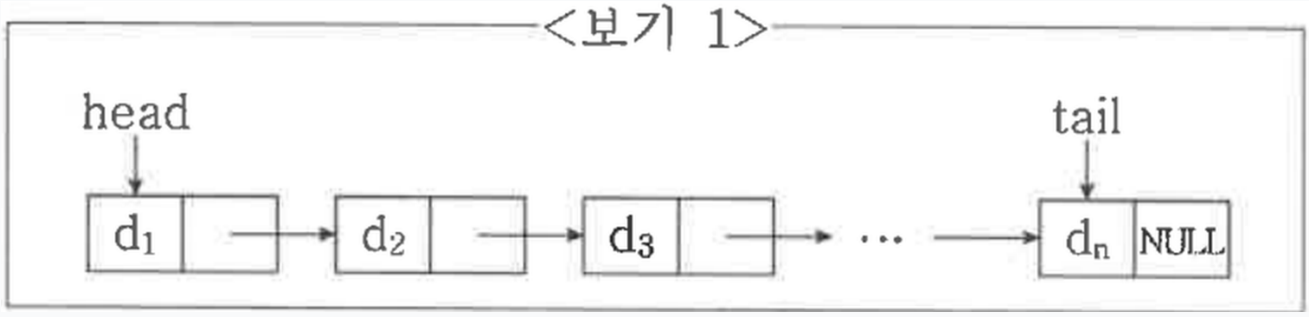
암기 포인트

BST 배열 낭비는 균형이면 작고 편향이면 크다. 특히 오른쪽 편향은 인덱스가 1, 3, 7, 15처럼 커져 가장 불리하다.

NoteBase

문 3. <보기 1>과 같은 형태로 연결 리스트가 구성되어 있을 때, O(1) 내에 수행할 수 없는 연산을 <보기 2>에서 모두 고른 것은? (단, head와 tail은 각각 첫 번째 원소와 마지막 원소를 가리키는 포인터이다.)

<보기 1>



<보기 2>

ㄱ. 첫 번째 위치에 새로운 원소 삽입

ㄴ. 마지막 위치에 새로운 원소 삽입

ㄷ. 마지막 위치의 원소 삭제

ㄹ. d_1부터 d_n 중에 특정 값이 저장되어 있는지 확인

① ㄱ, ㄴ

② ㄱ, ㄹ

③ ㄴ, ㄷ

④ ㄷ, ㄹ

정답 4번

출제 포인트

이 문제는 head와 tail만 주어진 단순 연결 리스트에서 각 연산이 추가 탐색 없이 가능한지, 즉 마지막 노드의 이전 노드나 특정 값 탐색이 필요한지를 묻는다.

NoteBase

선지별 해설

1번 X: 'ㄱ, ㄴ'을 O(1) 이내에 수행할 수 없다고 본 선택지이다. ㄱ 첫 번째 위치 삽입은 새 노드의 링크를 기존 head에 연결하고 head만 바꾸면 되므로 O(1)이다. ㄴ 마지막 위치 삽입도 tail이 마지막 노드를 이미 가리키므로 tail의 링크를 새 노드로 연결한 뒤 tail만 갱신하면 O(1)이다. 틀린 표현은 'ㄱ, ㄴ'이고, 올바르게는 'ㄷ, ㄹ'이다.
2번 X: ㄹ 특정 값 존재 확인은 맞게 골랐지만 ㄱ이 틀렸다. 특정 값 확인은 head부터 순차 탐색해야 하므로 최악의 경우 모든 노드를 확인해야 해서 O(n)이다. 반면 ㄱ 첫 번째 위치 삽입은 O(1)이다. 틀린 표현은 'ㄱ, ㄹ'이고, 올바르게는 'ㄷ, ㄹ'이다.
3번 X: ㄷ 마지막 위치 원소 삭제는 O(1) 이내에 불가능하지만 ㄴ이 틀렸다. tail이 있어도 삭제하려면 마지막 노드의 이전 노드를 찾아 그 링크를 NULL로 바꾸고 tail을 그 이전 노드로 옮겨야 한다. 단순 연결 리스트에는 이전 포인터가 없으므로 head부터 따라가며 찾아야 해서 O(n)이다. 반면 ㄴ 마지막 위치 삽입은 O(1)이다. 틀린 표현은 'ㄴ, ㄷ'이고, 올바르게는 'ㄷ, ㄹ'이다.
4번 O: ㄷ 마지막 위치 원소 삭제는 마지막 노드의 이전 노드를 찾아야 하므로 O(1)에 할 수 없고, ㄹ 특정 값 존재 확인도 순차 탐색이 필요하므로 O(1)에 할 수 없다. 따라서 O(1) 이내에 수행할 수 없는 연산을 모두 고른 선택지는 4번이다.

오답 포인트

tail 포인터가 있으니 마지막 위치 관련 연산이 모두 O(1)이라고 오해하기 쉽다. 그러나 마지막 위치 삽입은 tail만 알면 되지만 마지막 위치 삭제는 tail의 이전 노드를 알아야 하므로 성격이 다르다. 또한 값 존재 확인은 head와 tail을 알아도 중간 노드들을 건너뛸 수 없다는 점을 놓치면 틀린다.

다시 나오면 이렇게 판단

단순 연결 리스트 문제는 필요한 노드가 head나 tail이 직접 가리키는 대상만으로 충분한지, 아니면 그 이전 노드나 중간 값 탐색이 필요한지를 먼저 본다. 직접 연결만 바꾸면 되는 앞삽입과 뒤삽입은 O(1)이고, 순차 이동이 필요한 뒤삭제와 값검색은 O(n)이다.

암기 포인트

단순 연결 리스트는 앞삽입 O(1), tail이 있으면 뒤삽입 O(1), 뒤삭제와 값검색은 이전 노드 또는 중간 탐색이 필요해 O(n)이다.

NoteBase

문 4. <보기>의 배열 A를 버블 정렬과 삽입 정렬을 이용하여 오름차순으로 정렬하려고 한다. (가) 버블 정렬과 (나) 삽입 정렬에 의한 원소 교환 횟수를 가장 옳게 짝지은 것은? (단, 삽입 정렬에서 배열 원소의 이동은 인접한 원소 교환에 의해서만 수행된다. 예를 들어, A[5]를 A[2]와 A[3] 사이에 삽입하기 위해서는 두 번의 교환 연산 A[5]↔A[4], A[4]↔A[3]을 수행한다.)

<보기>

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
A	5	8	3	1	4	6	7	2

① (가): 7번, (나): 7번

② (가): 7번, (나): 15번

③ (가): 15번, (나): 7번

④ (가): 15번, (나): 15번

정답 4번

출제 포인트

이 문제는 버블 정렬과 삽입 정렬에서 인접한 원소 교환 횟수가 초기 배열의 역순쌍 수와 어떻게 대응되는지를 묻는 문제이다.

조건 정리

주어진 배열은 5, 8, 3, 1, 4, 6, 7, 2이다. 버블 정렬은 인접한 두 원소를 교환하면서 큰 값을 뒤로 보낸다. 삽입 정렬도 이 문제에서는 원소 이동을 인접 교환으로만 수행하므로, 한 원소가 왼쪽으로 이동한 칸 수만큼 교환 횟수가 증가한다.

사용할 원리

인접 교환만 사용하는 정렬에서 총 교환 횟수는 초기 배열의 역순쌍 수와 같다. 즉 swap 수 = inversion 수이다. 버블 정렬은 한 번의 인접 교환으로 역순쌍을 하나씩 없애고, 삽입 정렬도 원소를 제자리로 보내기 위해 필요한 인접 교환 수가 그 원소가 만드는 역순쌍 수와 일치한다.

NoteBase

계산 과정

버블 정렬의 교환 횟수는 역순쌍을 세면 된다. 5는 3, 1, 4, 2와 역순이어서 4회, 8은 3, 1, 4, 6, 7, 2와 역순이어서 6회, 3은 1, 2와 역순이어서 2회, 4는 2와 역순이어서 1회, 6은 2와 역순이어서 1회, 7은 2와 역순이어서 1회이다. 합계는 4+6+2+1+1+1=15회이다. 삽입 정렬도 인접 교환 기준으로 3은 2회, 1은 3회, 4는 2회, 6은 1회, 7은 1회, 2는 6회 이동하므로 총 2+3+2+1+1+6=15회이다.

선지별 해설

1번 X. (가)를 7번, (나)를 7번으로 본 것은 역순쌍을 일부만 세었거나 한 번의 패스 횟수와 총 교환 횟수를 혼동한 경우이다. 올바르게는 (가) 15번, (나) 15번이다.
2번 X. 삽입 정렬의 15번은 맞지만 버블 정렬의 총 교환 횟수도 15번이다. 올바르게는 (가) 15번, (나) 15번이다.
3번 X. 버블 정렬의 15번은 맞지만 삽입 정렬도 인접 교환 기준에서는 역순쌍 수만큼 교환하므로 15번이다. 올바르게는 (가) 15번, (나) 15번이다.
4번 O. 버블 정렬과 삽입 정렬 모두 인접 교환 횟수가 15번이므로 (가) 15번, (나) 15번이 맞다.

오답 포인트

버블 정렬의 한 패스 수나 삽입 정렬의 한 원소 이동만 보고 전체 교환 횟수를 축소해서 세기 쉽다. 특히 삽입 정렬에서 대입 이동 횟수와 이 문제의 인접 교환 횟수를 같은 것으로 보지 않으면 (나)를 7번처럼 잘못 잡기 쉽다.

다시 나오면 이렇게 판단

인접 교환 횟수를 묻는 정렬 문제는 먼저 역순쌍 수로 환원해서 본다. 버블 정렬은 총 역순쌍 제거 횟수로, 삽입 정렬은 각 원소가 왼쪽으로 밀고 들어가는 칸 수의 합으로 판단하면 된다.

암기 포인트

인접 교환 기반 버블 정렬과 삽입 정렬의 총 교환 횟수는 초기 역순쌍 수와 같다.

NoteBase

문 5. 재귀를 이용한 이진 탐색(binary search)으로 3을 찾을 때, 함수 호출 횟수가 가장 많은 배열은? (만약, 이진 탐색할 범위의 길이가 2n+1이라면 n+1번째 원소가, 2n이라면 n번째 원소가 기준이 된다.)

① 1 2 3 4 5 6

② 2 3 4 5 6 7

③ 3 4 5 6 7 8

④ -1 0 1 2 3 4

정답 2번

출제 포인트

이 문제는 재귀 이진 탐색에서 중앙 원소 선택 규칙에 따라 목표값 3에 도달할 때의 호출 횟수를 비교하는 문제이다.

조건 정리

모든 배열의 길이는 6이므로 처음 기준 원소는 3번째 원소이다. 길이가 2인 구간에서는 1번째 원소가 기준이 된다. 재귀 호출은 기준 원소를 다시 잡을 때마다 1회씩 추가된다.

계산 과정

① [1, 2, 3, 4, 5, 6]은 처음 기준 원소가 3이므로 1회 만에 종료된다.
② [2, 3, 4, 5, 6, 7]은 처음 기준 원소가 4이므로 왼쪽 [2, 3]으로 들어가고, 그 구간의 기준 원소 2를 본 뒤 오른쪽 [3]으로 한 번 더 들어가서 총 3회 호출된다.
③ [3, 4, 5, 6, 7, 8]은 처음 기준 원소가 5이므로 왼쪽 [3, 4]로 들어가고, 그 구간의 기준 원소가 3이어서 총 2회 호출된다.
④ [-1, 0, 1, 2, 3, 4]는 처음 기준 원소가 1이므로 오른쪽 [2, 3, 4]로 들어가고, 그 구간의 기준 원소가 3이어서 총 2회 호출된다.

선지별 해설

① X: 호출 횟수는 1회이다. 처음 중앙 원소가 바로 3이므로 더 진행되지 않는다. 틀린 표현은 '함수 호출 횟수가 가장 많다'는 판단이며, 올바르게는 '가장 적은 편에 속한다'이다.
② O: 호출 횟수는 3회이다. 4와 비교한 뒤 왼쪽으로, 다시 2와 비교한 뒤 오른쪽으로 이동하므로 네 선택지 중 가장 깊게 내려간다.
③ X: 호출 횟수는 2회이다. 처음 5를 본 뒤 왼쪽 [3, 4]에서 바로 3을 만나므로 ②보다 적다. 틀린 표현은 '가장 많다'는 판단이며, 올바르게는 '2회 호출된다'이다.
④ X: 호출 횟수는 2회이다. 처음 1을 본 뒤 오른쪽 [2, 3, 4]의 중앙 3에서 찾으므로 ②보다 적다. 틀린 표현은 '가장 많다'는 판단이며, 올바르게는 '2회 호출된다'이다.

NoteBase

오답 포인트

짝수 길이에서 중앙을 4번째가 아니라 3번째로 잡는 조건을 놓치면 ②의 호출 경로를 잘못 계산하기 쉽다. 또한 호출 횟수는 비교 횟수와 같은 흐름으로 보아야 하므로, 구간이 한 번 더 쪼개지는 배열이 무엇인지 확인해야 한다.

다시 나오면 이렇게 판단

이진 탐색 호출 횟수 비교 문제는 각 선지마다 처음 중앙 원소를 먼저 적고, 목표값보다 큰지 작은지에 따라 남는 부분배열을 순서대로 써 내려가면 된다. 부분배열이 한 단계 더 생기는 선지가 호출 횟수가 더 많다.

암기 포인트

재귀 이진 탐색의 호출 횟수는 '중앙 선택 규칙 + 남는 부분배열의 단계 수'로 결정된다.

문 6. 제시된 인접 행렬(adjacency matrix) 중, 그래프로 구현하였을 때 사이클(cycle)이 생성되는 것은?

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

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

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

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

정답 1번

출제 포인트

이 문제는 6×6 인접 행렬을 무방향 그래프로 바꾸어 닫힌 경로가 생기는지 판별할 수 있는지를 묻는다.

선지별 해설

1번: O. 간선은 1-2, 1-3, 2-4, 3-4, 3-5, 4-6으로 읽힌다. 이때 1-2-4-3-1이 닫힌 경로를 이루므로 사이클이 생성된다.
2번: X. 간선은 1-2, 1-3, 3-4, 3-5, 5-6이다. 가지처럼만 뻗는 구조라 출발점으로 되돌아오는 경로가 없다. 올바르게는 사이클이 없는 트리이다.
3번: X. 간선은 1-2, 1-3, 3-4, 4-6이고 5번 정점은 고립되어 있다. 닫힌 경로가 전혀 없으므로 사이클이 생기지 않는다. 올바르게는 비연결 비순환 그래프이다.
4번: X. 간선은 1-3, 2-4, 3-4, 3-5, 4-6이다. 3-4를 중심으로 연결되지만 다시 시작점으로 돌아오는 별도의 연결이 없어 사이클이 없다. 올바르게는 연결된 트리이다.

오답 포인트

대칭행렬에서는 같은 간선이 행과 열에 두 번 나타나므로 1의 개수를 그대로 세면 오판하기 쉽다. 4번은 가운데 연결이 복잡해 보여 사이클처럼 보이지만 닫힌 경로가 없고, 3번은 정점 하나가 고립되어 있다는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

대각선 값으로 자기 루프 여부를 먼저 확인하고, 대칭행렬이면 윗삼각 부분의 1만 간선으로 옮겨 본다. 그다음 시작 정점으로 되돌아오는 닫힌 경로가 있는지 추적하고, 연결 그래프에서 간선 수가 정점 수보다 1 적으면 트리라고 판단한다.

NoteBase

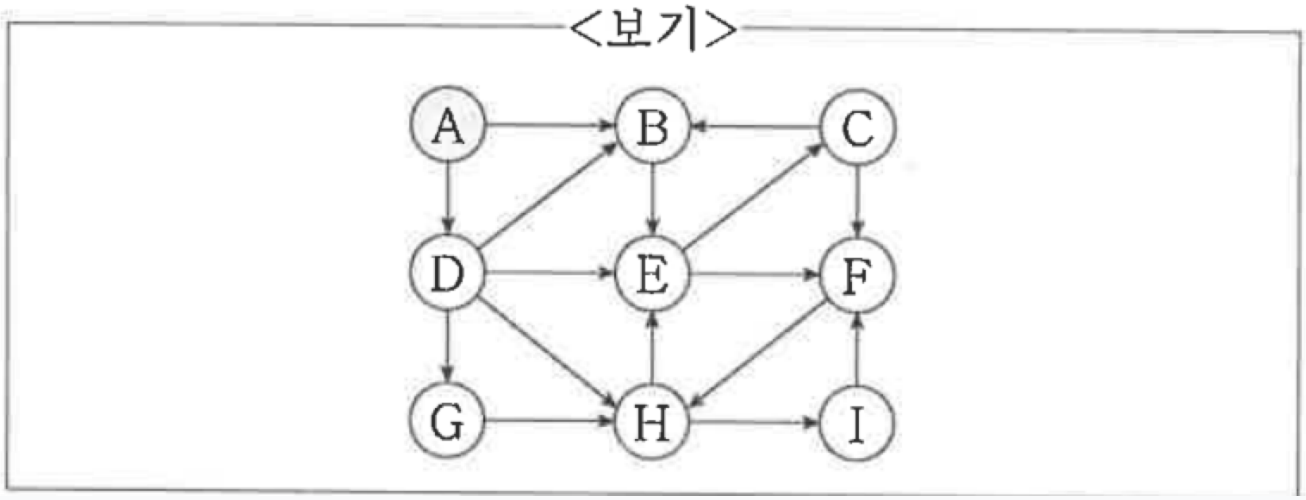
암기 포인트

무방향 그래프는 연결되고 간선 수가 정점 수보다 1 적으면 트리이고, 닫힌 경로가 하나라도 있으면 사이클이다.

NoteBase

문 7. <보기>의 그래프에서 정점 A를 출발점으로 너비 우선 탐색(breadth first search)을 수행하는 과정에서 나타날 수 있는 큐(queue)의 상태로 가장 옳지 않은 것은?

<보기>



① (앞) D - E (뒤)

② (앞) C - I - F (뒤)

③ (앞) B - E - G - H (뒤)

④ (앞) H - G - C - F (뒤)

정답 2번

출제 포인트

이 문제는 너비 우선 탐색에서 정점을 발견하는 순서와 큐에 들어가는 순서의 관계를 추적할 수 있는지를 묻는 문제이다.

올바른 순서

BFS에서는 현재 큐의 맨 앞 정점을 꺼내고, 그 정점에서 새로 발견한 인접 정점들을 큐의 뒤에 연속해서 넣는다. 이 그래프에서는 A에서 시작하면 먼저 B와 D가 큐에 들어가고, E는 B 또는 D를 처리하는 시점에 발견된다. C와 F는 모두 E를 처리하는 순간 함께 큐 뒤에 붙고, I는 H를 처리한 뒤에야 들어간다. 따라서 E의 자식인 C와 F 사이에 H의 자식인 I가 끼어드는 형태는 만들 수 없다.

NoteBase

선지별 해설

① O: A의 인접 정점을 B, D 순으로 넣으면 큐는 [B, D]가 된다. 이어서 B를 꺼내 E를 넣으면 큐는 [D, E]가 되므로 가능한 상태이다.
② X: C와 F는 E를 처리할 때 함께 enqueue되는 같은 묶음이다. I는 H를 처리한 뒤에야 enqueue되므로 [C, I, F]처럼 I가 C와 F 사이에 들어가는 상태는 나올 수 없다. 틀린 표현은 I가 C와 F 사이에 위치할 수 있다는 점이며, 올바르게는 [C, F, I] 또는 인접 정점 처리 순서에 따라 [F, C, I]처럼 C와 F가 먼저 연속해서 나타나야 한다.
③ O: A의 인접 정점을 D, B 순으로 넣으면 큐는 [D, B]가 된다. 그다음 D를 먼저 꺼내 E, G, H를 새로 발견하는 순서로 넣으면 큐가 [B, E, G, H]가 되므로 가능한 상태이다.
④ O: A의 인접 정점을 B, D 순으로 넣고 B를 처리하면 큐는 [D, E]가 된다. 이어서 D를 처리하면서 H, G 순으로 새 정점을 넣으면 [E, H, G]가 되고, 그다음 E를 처리하며 C, F를 넣으면 [H, G, C, F]가 되므로 가능한 상태이다.

오답 포인트

BFS에서는 같은 부모 정점에서 발견된 자식들이 큐 뒤에 한 묶음으로 들어간다는 점을 놓치기 쉽다. ②는 C와 F가 E의 자식이고 I는 H의 자식이라는 부모 관계를 흐리게 만들어, 나중에 발견된 I가 먼저 들어올 수 있는 것처럼 보이게 한 함정이다.

다시 나오면 이렇게 판단

BFS 큐 문제는 큐의 맨 앞 정점을 하나 꺼낼 때마다 어떤 정점들이 새로 뒤에 붙는지를 부모별로 묶어서 추적하면 된다. 같은 정점에서 한 번에 발견된 자식들 사이에는 나중에 다른 부모에서 발견된 정점이 끼어들 수 없다는 규칙으로 판단하면 된다.

암기 포인트

BFS는 먼저 들어간 정점이 먼저 나오고, 한 부모의 자식들은 큐 뒤에 연속해서 붙는다.

NoteBase

문 8. <보기>에서 수식 트리와 수식 표기법에 대한 설명으로 옳은 것을 모두 고른 것은?

<보기>
ㄱ. 중위 표기식을 후위 표기식으로 변환할 때, 피연산자를 저장하기 위한 스택을 보조 자료구조로 사용한다.
ㄴ. 중위 표기식과 이에 대한 후위 표기식에서 피연산자가 나타나는 순서는 동일하다.
ㄷ. 수식 트리를 후위 순회하면 후위 표기식을 얻을 수 있다.
ㄹ. 후위 표기식을 계산할 때, 수식의 가장 끝에 있는 연산자가 가장 마지막에 계산되지 않을 수도 있다.

① ㄱ, ㄴ
② ㄱ, ㄹ
③ ㄴ, ㄷ
④ ㄷ, ㄹ

정답 3번

출제 포인트

이 문제는 중위 표기식과 후위 표기식의 변환 원리, 수식 트리의 순회 방식, 후위 표기식 계산 순서를 함께 묻는다.

핵심 개념

중위 표기식은 연산자가 피연산자 사이에 오는 표기이고, 후위 표기식은 연산자가 피연산자 뒤에 오는 표기이다.
중위 표기식을 후위 표기식으로 바꿀 때는 연산자 우선순위와 괄호를 처리하기 위해 스택을 사용하고,
피연산자는 읽는 순서대로 바로 출력한다.
수식 트리를 후위 순회하면 왼쪽 서브트리-오른쪽 서브트리-루트 순서가 되어 후위 표기식이 된다.

선지별 해설

1번: X. ㄱ, ㄴ의 조합이다. ㄱ은 틀리고 ㄴ만 옳으므로 오답이다. ㄱ의 '피연산자를 저장하기 위한 스택'이 틀린 표현이고, 실제로는 중위 표기식을 후위 표기식으로 변환할 때 연산자와 괄호 처리를 위해 스택을 사용한다.
2번: X. ㄱ, ㄹ의 조합이다. ㄱ과 ㄹ이 모두 틀리므로 오답이다. ㄹ의 '수식의 가장 끝에 있는 연산자가 가장 마지막에 계산되지 않을 수도 있다'가 틀린 표현이고, 실제로는 후위 표기식의 마지막 연산자가 전체 식의 최종 연산이므로 마지막에 계산된다.
3번: O. ㄴ, ㄷ의 조합이다. 중위 표기식과 그에 대응하는 후위 표기식에서는 피연산자가 나타나는 순서가 유지된다. 또한 수식 트리를 후위 순회하면 후위 표기식이 얻어지므로 옳다.
4번: X. ㄷ, ㄹ의 조합이다. ㄷ은 옳지만 ㄹ이 틀리므로 오답이다. ㄹ에서 잘못된 부분은 '가장 마지막에 계산되지 않을 수도 있다'이고, 올바르게는 후위 표기식의 맨 끝 연산자가 최종 계산을 담당한다.

NoteBase

오답 포인트

중위→후위 변환 단계와 후위 표기식 계산 단계를 섞으면 ㄱ과 ㄹ에서 흔들리기 쉽다. 변환 단계의 스택은 연산자 관리용이고, 계산 단계의 스택은 피연산자 저장용이라는 점을 분리해야 한다.

다시 나오면 이렇게 판단

변환, 순회, 계산을 따로 끊어 판단하면 된다. 변환에서는 스택에 무엇을 넣는지, 순회에서는 후위 순회 결과가 무엇인지, 계산에서는 마지막 연산자가 전체 결과를 만드는지를 확인하면 된다.

암기 포인트

중위→후위 변환은 연산자 스택, 수식 트리 후위 순회는 후위 표기식, 후위 표기식의 마지막 연산자는 최종 계산이다.

NoteBase

문 9. 크기가 13인 원형 큐(circular queue)를 배열로 구현할 때, 큐에 원소가 가장 많이 들어있다고 판단되는 경우는? (단, front와 rear는 배열의 인덱스 번호이며, front는 바로 다음에 삭제될 원소의 인덱스, rear는 새로 들어올 원소가 저장될 인덱스이다.)

① front: 5, rear: 3

② front: 3, rear: 5

③ front: 5, rear: 1

④ front: 1, rear: 5

정답 1번

출제 포인트

원형 큐에서 front가 삭제될 원소의 위치이고 rear가 새 원소가 들어갈 위치일 때 현재 원소 수를 계산하는 규칙을 묻는 문제이다.

조건 정리

배열 크기는 13이다. front는 바로 다음에 삭제될 원소의 인덱스이고 rear는 새 원소가 저장될 인덱스이다. 이 정의에서는 원소 수를 rear와 front의 차이로 계산하며, 공백 한 칸을 비워 두므로 최대 저장 가능 원소 수는 12개이다.

사용할 공식

현재 원소 수는 (rear - front + 13) 13이다. rear는 다음 삽입 위치이고 front는 현재 맨 앞 원소의 위치이므로, front에서 rear까지 차지한 칸 수를 원형으로 계산하면 현재 저장된 원소 수가 된다.

계산 과정

① (3 - 5 + 13) 13 = 11이다.
② (5 - 3 + 13) 13 = 2이다.
③ (1 - 5 + 13) 13 = 9이다.
④ (5 - 1 + 13) 13 = 4이다.
가장 큰 값은 11이므로 가장 많은 원소가 들어 있는 경우는 ①이다.

NoteBase

선지별 해설

① O: front=5, rear=3이면 원소 수는 11개이다. 네 선택지 중 가장 큰 값이므로 가장 많이 들어 있는 경우가 맞다.
② X: front=3, rear=5이면 원소 수는 2개이다. 가장 많은 경우가 아니라 거의 비어 있는 상태에 가깝다. 올바르게는 이 경우의 원소 수는 2개이다.
③ X: front=5, rear=1이면 원소 수는 9개이다. 원소가 적지 않지만 ①의 11개보다 작으므로 가장 많은 경우는 아니다. 올바르게는 ③은 9개가 저장된 경우이다.
④ X: front=1, rear=5이면 원소 수는 4개이다. ①보다 훨씬 적으므로 최다 저장 상태가 아니다. 올바르게는 ④는 4개가 저장된 경우이다.

오답 포인트

rear를 마지막 원소의 위치로 착각하면 한 칸씩 어긋난다. 또 원형 구조라는 점을 무시하고 단순히 rear-front만 보면 ①과 ③처럼 rear가 front보다 작은 경우를 잘못 판단하게 된다.

다시 나오면 이렇게 판단

원형 큐 문제는 먼저 front와 rear가 가리키는 의미를 확인한 뒤, rear가 다음 삽입 위치이면 원소 수를 (rear-front+배열크기) 배열크기로 계산해 비교하면 된다.

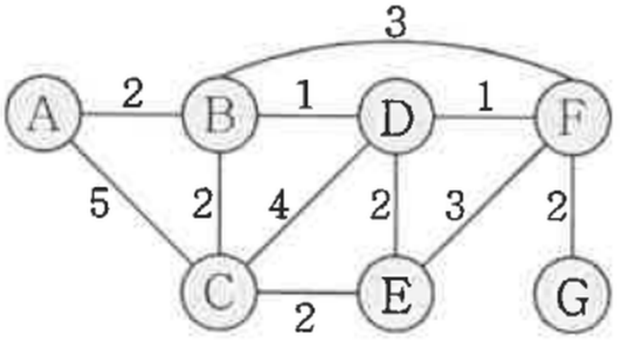
암기 포인트

원형 큐에서 front는 삭제 위치, rear는 삽입 위치이면 현재 원소 수는 rear-front를 원형으로 보정한 값이다.

NoteBase

문 10. 다익스트라(Dijkstra) 알고리즘을 이용하여 <보기> 그래프의 정점 A에서 각 정점으로의 최단 경로를 구할 때, 최단 경로를 발견하는 순서를 바르게 나열한 것은? (만약, 둘 이상의 정점에 대해 동시에 최단 경로를 찾을 수 있다면, 알파벳 순으로 정렬한다.)

<보기>



① A - B - D - F - C - E - G

② A - B - D - C - F - E - G

③ A - B - C - D - E - F - G

④ A - B - D - E - C - F - G

정답 2번

출제 포인트

이 문제는 다익스트라 알고리즘에서 시작 정점 A로부터 각 정점의 최단거리가 확정되는 순서와 동점일 때의 처리 기준을 묻는 문제이다.

올바른 순서

A에서 시작하면 처음 확정되는 정점은 A이다. 그다음은 A-B 가중치 2로 B가 확정된다. B를 거치면 D의 거리가 3, C의 거리가 4가 되므로 세 번째는 D이다. D를 거치면 F의 거리가 4, E의 거리가 5가 된다. 이 시점에서 C와 F가 모두 4이므로 조건에 따라 알파벳 순서인 C가 먼저, 다음이 F이다. 그 뒤 거리가 5인 E, 마지막으로 거리가 6인 G가 확정된다. 따라서 올바른 순서는 A-B-D-C-F-E-G이다.

NoteBase

선지별 해설

1번: X. D까지 확정한 뒤 C와 F의 현재 최단거리가 모두 4이다. 동점이면 알파벳 순서를 적용하므로 F가 아니라 C가 먼저 와야 한다. 틀린 표현은 '...D-F-C...'이고, 올바르게는 '...D-C-F...'이다.
2번: O. A(0) → B(2) → D(3)까지는 최단거리가 가장 작은 정점이 차례로 확정된다. 이후 C와 F가 모두 4이므로 C가 먼저, 다음이 F이며, 그 뒤 E(5), G(6)가 확정되어 A-B-D-C-F-E-G가 된다.
3번: X. B 다음에 C가 올 수 없다. B를 확정한 시점의 거리는 C=4, D=3이므로 D가 C보다 먼저 확정된다. 틀린 표현은 '...B-C-D...'이고, 올바르게는 '...B-D-C...'이다.
4번: X. E는 D를 거친 뒤에도 거리가 5이므로 거리 4인 C와 F보다 먼저 확정될 수 없다. 틀린 표현은 '...D-E-C-F...'이고, 올바르게는 '...D-C-F-E...'이다.

오답 포인트

이 문항의 함정은 그래프에서 눈에 띄는 경로 모양을 따라가며 방문 순서를 추정하는 데 있다. 다익스트라는 간선을 따라 이동한 순서가 아니라 현재까지 계산된 최단거리의 최소값이 확정되는 순서를 따르며, C와 F가 동시에 4가 되는 동점 상황에서 알파벳 순서를 적용해야 한다.

다시 나오면 이렇게 판단

다익스트라 순서 문제는 현재 확정되지 않은 정점들의 임시 거리표를 매 단계 갱신하면서 가장 작은 값을 고르면 된다. 동점이 있으면 문제에서 준 추가 규칙을 적용하고, 어떤 정점이 다른 정점보다 작아지는 순간이 언제인지 확인하면 순서를 안정적으로 판단할 수 있다.

암기 포인트

다익스트라는 경로를 따라 걷는 순서가 아니라 최단거리 확정 순서이며, 동점이면 제시된 우선순위 규칙을 따른다.

NoteBase

문 11. <보기>의 항목들의 시간 복잡도를 낮은 것부터 높은 것 순서대로 바르게 나열한 것은?

<보기>

ㄱ. 임의의 자연수 n과 정수 a, b, c를 입력으로 받는 다음 함수

```
void hanoi(int n, int a, int b, int c) {
    if (n >= 1) {
        hanoi(n - 1, a, c, b);
        printf("%d to %d\n", a, b);
        hanoi(n - 1, c, b, a);
    }
}
```

ㄴ. 시간 복잡도 함수 $T(n)=5n^4+2n^3+n$

ㄷ. 배열로 표현한 스택(stack)에서의 삭제(pop) 연산 수행

ㄹ. 단순 연결 리스트에서 임의의 원소를 탐색하는 연산 수행 중 최악의 경우

① ㄱ - ㄴ - ㄹ - ㄷ

② ㄷ - ㄱ - ㄹ - ㄴ

③ ㄷ - ㄹ - ㄴ - ㄱ

④ ㄹ - ㄷ - ㄱ - ㄴ

정답 3번

출제 포인트

이 문제는 각 항목의 대표 시간 복잡도를 구한 뒤 낮은 차수에서 높은 차수 순으로 배열할 수 있는지를 묻는 문제이다.

올바른 순서

1. ㄷ — 배열로 구현한 스택의 pop은 top만 감소시키면 되어 시간 복잡도가 O(1)이다.
2. ㄹ — 단순 연결 리스트에서 임의의 원소를 최악의 경우 끝까지 탐색하므로 O(n)이다.
3. ㄴ — T(n)=5n^4+2n^3+n은 최고차항이 지배하므로 O(n^4)이다.
4. ㄱ — 하노이 함수는 T(n)=2T(n-1)+1 형태로 증가하므로 O(2^n)이다.

왜 이 순서: 상수 시간인 ㄷ이 가장 먼저 오고, 선형 탐색인 ㄹ이 그다음이며, 다항식 중 4차인 ㄴ보다 지수 시간인 ㄱ이 더 크게 증가한다.

NoteBase

선지별 해설

① X: ㄱ-ㄴ-ㄷ-ㄹ은 가장 큰 복잡도인 ㄱ을 맨 앞에 둔 것이 틀렸다. 틀린 부분은 "ㄱ이 ㄴ과 ㄷ보다 앞에 오는 순서"이다. 올바르게는 ㄷ-ㄹ-ㄴ-ㄱ이다.
② X: ㄷ-ㄱ-ㄹ-ㄴ은 ㄱ을 ㄹ과 ㄴ보다 앞에 둔 것이 틀렸다. 틀린 부분은 "지수 시간인 ㄱ이 선형 시간 ㄹ, 4차 다항 시간 ㄴ보다 앞에 오는 순서"이다. 올바르게는 ㄷ-ㄹ-ㄴ-ㄱ이다.
③ O: ㄷ-ㄹ-ㄴ-ㄱ은 O(1), O(n), O(n^4), O(2^n)를 낮은 것부터 높은 것 순서대로 정확히 배치한 순서이다.
④ X: ㄹ-ㄷ-ㄱ-ㄴ은 O(1)인 ㄷ보다 O(n)인 ㄹ을 앞에 둔 점과, O(2^n)인 ㄱ을 O(n^4)인 ㄴ보다 앞에 둔 점이 모두 틀렸다. 틀린 부분은 "ㄹ-ㄷ"와 "ㄱ-ㄴ"의 배치이다. 올바르게는 ㄷ-ㄹ-ㄴ-ㄱ이다.

오답 포인트

하노이 재귀를 단순 재귀 한 번으로 보고 O(n) 정도로 낮게 오해하거나, 다항식과 지수식의 크기 비교에서 n^4가 2^n보다 크다고 착각하면 순서가 뒤집힌다. 또한 배열 스택의 pop은 원소들을 이동하지 않으므로 O(1)이라는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

시간 복잡도 나열 문제는 각 항목을 먼저 O(1), O(log n), O(n), O(n log n), O(n^k), O(c^n) 같은 대표 등급으로 환산한 뒤 비교한다. 재귀는 점화식을 세워 보고, 자료구조 연산은 실제로 몇 개의 원소를 건드리는지를 기준으로 판단한다.

암기 포인트

배열 스택 pop은 O(1), 연결 리스트 탐색은 O(n), 다항식은 최고차항, 하노이 재귀는 O(2^n)으로 정리된다.

NoteBase

문 12. 배열로 구현된 이중 연결 리스트(doubly linked list)에서 배열 A는 원소의 키값인 정수를 저장하고 있고, 배열 prev, next는 각각 이중 연결 리스트에서 직전 원소와 다음 원소가 저장된 배열의 인덱스를 저장하고 있다. prev와 next에 0이 저장되어 있다면 이는 NULL값을 의미한다. 리스트를 순서대로 방문했을 때, 저장된 키값이 점점 커지는 것은?

①

인덱스	1	2	3	4	5	6
A	1	3	2	5	4	6
prev	0	3	1	5	2	4
next	3	5	2	6	4	0

②

인덱스	1	2	3	4	5	6
A	1	2	3	4	5	6
prev	2	3	4	5	6	0
next	0	1	2	3	4	5

③

인덱스	1	2	3	4	5	6
A	1	3	2	5	4	6
prev	3	5	2	6	4	0
next	0	3	1	5	2	4

④

인덱스	1	2	3	4	5	6
A	1	2	3	4	5	6
prev	2	4	5	3	6	0
next	0	1	4	2	3	5

정답 1번

출제 포인트

배열로 구현된 이중 연결 리스트에서 머리 노드를 찾고 `next`를 따라 실제 순회 순서를 복원할 수 있는지를 묻는 문제이다.

NoteBase

선지별 해설

1번: O. `prev=0`인 인덱스 1이 머리 노드이다. `next`를 따라가면 1→3→2→5→4→6이고, 이에 대응하는 `A` 값은 1→2→3→4→5→6이므로 점점 커진다.
2번: X. `prev=0`인 인덱스 6에서 시작해 `next`를 따라가면 6→5→4→3→2→1이고, `A` 값은 6→5→4→3→2→1로 점점 작아진다. 틀린 판단은 이 표가 오름차순 순회를 만든다고 보는 점이다. 올바르게는 내림차순 순회를 만든다.
3번: X. `prev=0`인 인덱스 6에서 시작해 `next`를 따라가면 6→4→5→2→3→1이고, `A` 값은 6→5→4→3→2→1로 역시 점점 작아진다. 틀린 판단은 연결 순서가 키 오름차순이라고 보는 점이다. 올바르게는 키 내림차순 연결이다.
4번: X. `prev=0`인 인덱스 6에서 시작해 `next`를 따라가면 6→5→3→4→2→1이고, `A` 값은 6→5→3→4→2→1이다. 중간에 3 다음 4가 나와 순서가 뒤섞이므로 점점 커지지 않는다. 틀린 판단은 이 표가 정렬된 순회를 표현한다고 보는 점이다. 올바르게는 오름차순도 내림차순도 아닌 비정렬 순회이다.

오답 포인트

`A` 배열 값만 보고 1, 2, 3, 4, 5, 6처럼 보인다고 착각하거나, `prev=0`과 `next=0`을 혼동하면 틀린다. 이 문제는 배열의 저장 위치가 아니라 연결 포인터가 만드는 실제 방문 순서를 묻는다.

다시 나오면 이렇게 판단

먼저 `prev=0`인 머리 노드를 찾고, 그 인덱스에서 `next`를 0이 나올 때까지 따라가며 인덱스 순서를 적는다. 그다음 그 인덱스들에 대응하는 `A` 값을 차례로 읽어 오름차순인지 확인하면 된다.

암기 포인트

배열 연결 리스트는 저장 위치가 아니라 `head(prev=0) → next`가 실제 순서이다.

NoteBase

문 13. <보기>는 어떤 이진 탐색 트리의 키값을 후위 순회 방문 순서대로 나열한 것이다. 이 트리에 대한 설명으로 가장 옳은 것은?

<보기>
21, 39, 50, 47, 32, 55, 52, 80, 91, 85, 79, 67

① 39는 내부 노드(비단말 노드)이다.
② 47의 왼쪽 자식은 32이다.
③ 79는 자식을 하나만 가진다.
④ 루트 노드의 왼쪽 자식은 55이다.

정답 3번

출제 포인트

이 문제는 이진 탐색 트리의 후위 순회 결과만 보고 루트와 각 서브트리의 부모·자식 관계를 복원할 수 있는지를 묻는다.

핵심 개념

이진 탐색 트리는 각 노드에서 왼쪽 서브트리의 모든 키가 더 작고 오른쪽 서브트리의 모든 키가 더 크다. 후위 순회는 왼쪽-오른쪽-루트 순서이므로 부분수열의 마지막 값이 그 서브트리의 루트가 된다. 이 수열을 복원하면 루트는 67, 루트의 왼쪽 자식은 52, 오른쪽 자식은 79이다.

선지별 해설

1번: X. 39는 47의 왼쪽 자식이며 자식이 전혀 없으므로 내부 노드가 아니라 단말 노드이다. 틀린 표현은 '내부 노드'이고, 올바르게는 '39는 단말 노드이다.'이다.
2번: X. 47의 왼쪽 자식은 39이고 오른쪽 자식은 50이다. 32는 52의 왼쪽 자식이다. 틀린 표현은 '47의 왼쪽 자식은 32'이고, 올바르게는 '47의 왼쪽 자식은 39이다.'이다.
3번: O. 67보다 큰 값들의 부분수열 80, 91, 85, 79에서 마지막 값 79가 오른쪽 서브트리의 루트이다. 그 앞의 값들은 모두 79보다 크므로 79의 왼쪽 자식은 없고, 오른쪽 자식 85만 가진다.
4번: X. 전체 수열의 마지막 값 67이 루트이고, 67보다 작은 값들의 부분수열 21, 39, 50, 47, 32, 55, 52의 마지막 값 52가 왼쪽 서브트리의 루트이다. 따라서 루트 노드의 왼쪽 자식은 55가 아니라 52이다.

오답 포인트

후위 순회 수열을 앞에서부터 단순히 연결해 부모·자식을 추정하면 32, 55, 79의 위치를 잘못 읽기 쉽다. 이 문제는 '마지막 값이 루트'라는 규칙과 '작은 값 묶음, 큰 값 묶음' 분할을 놓치면 오답이 된다.

NoteBase

다시 나오면 이렇게 판단

후위 순회 수열이 나오면 마지막 값을 먼저 루트로 잡고, 그보다 작은 연속 부분수열과 큰 연속 부분수열로 좌우 서브트리를 나눈다. 그다음 각 부분수열에서도 마지막 값을 다시 루트로 잡아 재귀적으로 복원하면 부모·자식 관계를 안정적으로 판단할 수 있다.

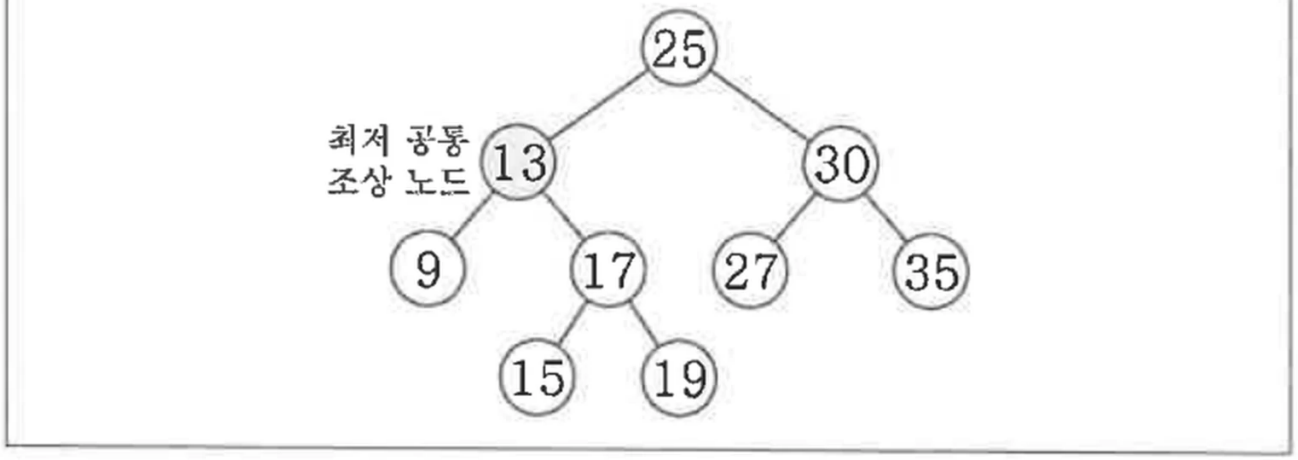
암기 포인트

BST 후위 순회는 마지막이 루트이고, 왼쪽 묶음은 작은 값, 오른쪽 묶음은 큰 값이다.

NoteBase

문 14. 이진 탐색 트리에 속한 두 개의 노드(node)가 주어질 때, 이들의 최저 공통 조상(lowest common ancestor) 노드를 찾는 코드를 작성하려고 한다. 이때, 최저 공통 조상 노드란 임의의 두 노드가 공통으로 갖는 조상 중, 가장 가까운 노드를 의미한다. 예를 들어, <보기 1>의 트리에서 노드 9와 19의 최저 공통 조상 노드는 13이다. <보기 2>의 (가)와 (나)에 작성해야 할 코드를 옳게 짝지은 것은? (단, 최저 공통 조상 노드를 찾는 두 개의 노드는 이미 트리에 존재한다고 가정한다.)

<보기 1>



<보기 2>

```
typedef struct Node {
        struct Node* left;
        struct Node* right;
        int value;
} Node;

Node* findLCA(Node* root, int val1, int val2) {
        while (root != NULL) {
          int value = root -> value;
          if ( _____(가)_____ ) {
               root = root -> left;
          } else if ( _____(나)_____ ) {
               root = root -> right;
          } else {
               return root;
          }
      }
      return NULL; // 트리가 비어있을 경우만
}
/* child1: 주어진 첫 번째 노드 (예시에서는 노드 9),
   child2: 주어진 두 번째 노드 (예시에서는 노드 19) */
Node* findLCAwithNodes(Node* root, Node* child1, Node* child2) {
      if (root == NULL || child1 == NULL || child2 == NULL) {
           return NULL;
      }

      return findLCA(root, child1->value, child2->value);
}
```

① (가) value < val1 && value < val2
(나) value > val1 && value > val2

NoteBase

② (가) value > val2 && value < val1
(나) value < val2 && value < val1

③ (가) value > val1 && value > val2
(나) value < val1 && value < val2

④ (가) value > val1 && value < val2
(나) value < val1 && value < val2

정답 3번

출제 포인트

이 문제는 이진 탐색 트리에서 두 값이 현재 노드보다 모두 작은지, 모두 큰지, 양쪽으로 갈리는지를 기준으로 최저 공통 조상 노드를 찾는 분기 조건을 묻는다.

핵심 개념

이진 탐색 트리는 왼쪽 서브트리의 값이 현재 노드보다 작고 오른쪽 서브트리의 값이 현재 노드보다 크다. 따라서 현재 값이 두 대상값보다 모두 크면 왼쪽으로 내려가고, 모두 작으면 오른쪽으로 내려간다. 현재 값이 두 값 사이에 끼이거나 둘 중 하나와 같으면 그 노드가 최저 공통 조상이다.

NoteBase

선지별 해설

1번: X. (가) `value < val1 && value < val2`이면 현재 노드값이 두 대상값보다 모두 작다는 뜻이므로 오른쪽 서브트리로 가야 한다. 그런데 코드에서는 왼쪽으로 이동하므로 방향이 반대이다. 틀린 표현은 (가)의 왼쪽 이동과 연결된 조건이다. 올바르게는 `(가) value > val1 && value > val2`이어야 한다. (나)도 `value > val1 && value > val2`이면 두 대상값이 모두 왼쪽에 있으므로 오른쪽이 아니라 왼쪽으로 가야 한다. 올바르게는 `(나) value < val1 && value < val2`이다.
2번: X. (가) `value > val2 && value < val1`은 현재 노드값이 두 값 사이에 있다는 뜻일 수 있으므로 이 경우는 왼쪽으로 내려갈 때가 아니라 현재 노드를 반환해야 하는 구간이다. 또한 `val1`과 `val2`의 대소관계가 이미 정해져 있다고 가정한 조건이라 일반적인 코드가 되지 못한다. 틀린 표현은 (가)의 사이 구간 비교이다. 올바르게는 `(가) value > val1 && value > val2`이다. (나) `value < val2 && value < val1` 자체는 두 값보다 모두 작은 경우이므로 오른쪽 이동 조건으로 맞다.
3번: O. (가) `value > val1 && value > val2`이면 두 대상 노드가 모두 현재 노드의 왼쪽에 있으므로 `root = root->left`가 맞다. (나) `value < val1 && value < val2`이면 두 대상 노드가 모두 현재 노드의 오른쪽에 있으므로 `root = root->right`가 맞다. 그 외의 경우는 현재 노드가 두 값 사이에 있으므로 최저 공통 조상으로 반환된다.
4번: X. (가) `value > val1 && value < val2`는 현재 노드값이 `val1`과 `val2` 사이에 있는 경우를 뜻하므로 왼쪽으로 내려가는 조건이 아니다. 이때는 현재 노드가 최저 공통 조상이 될 수 있으므로 반환 구간이다. 틀린 표현은 (가)의 사이 구간 비교이다. 올바르게는 `(가) value > val1 && value > val2`이다. (나) `value < val1 && value < val2`는 두 값보다 모두 작은 경우이므로 오른쪽 이동 조건으로는 맞다.

오답 포인트

출제자는 왼쪽과 오른쪽 이동 조건을 서로 뒤집거나, 현재 노드값이 두 값 사이에 있을 때도 계속 내려가게 만드는 함정을 넣었다. 특히 `value`가 두 값 사이에 있으면 분기 이동이 아니라 즉시 현재 노드를 반환한다는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

현재 노드값을 기준으로 두 대상값의 위치를 먼저 본다. 두 값이 모두 작으면 왼쪽, 두 값이 모두 크면 오른쪽, 하나는 작고 하나는 크거나 현재 노드와 같으면 그 자리에서 최저 공통 조상으로 판단한다.

암기 포인트

BST LCA의 핵심은 현재값 기준 비교이다. 현재값이 두 값보다 모두 크면 left, 모두 작으면 right, 두 값 사이면 return이다.

NoteBase

문 15. <보기>는 히프 정렬(heap sort) 알고리즘을 최대 히프(max heap)로 구현한 C언어 코드이다. (가), (나)에 들어갈 코드를 옳게 짝지은 것은? (단, 함수 swap()은 주어진 두 인자의 값을 서로 교환하는 사용자 정의 함수라고 가정한다.)

<보기>

```
void adjust(int a[], int root, int n) {
        int child, rootkey;
        int temp;
        temp = a[root];
        rootkey = a[root];
        child = 2 * root;
        while (child <= n) {
          if ((child < n) &&
             (a[child] < (a[child+1]))
                child++;
          if (__(가)__)
                break;
          else {
                (__(나)__);
                child *= 2;
          }
      }
      a[child/2] = temp;
}
void heapSort(int a[], int n) {
        int i;

        for (i = n/2; i > 0; i--)
                adjust(a, i, n);
        for (i = n; i > 1; i--) {
                swap(a[1], a[i]);
                adjust(a, 1, i-1);
        }
}
```

① (가): rootkey < a[child], (나): a[child] = a[child/2]

② (가): rootkey < a[child], (나): a[child/2] = a[child]

③ (가): rootkey > a[child], (나): a[child] = a[child/2]

④ (가): rootkey > a[child], (나): a[child/2] = a[child]

정답 4번

출제 포인트

이 문제는 최대 힙의 adjust 과정에서 부모와 더 큰 자식의 비교 조건, 그리고 자식을 부모 자리로 끌어올리는 대입 방향을 묻는다.

핵심 개념

최대 힙은 모든 부모 노드가 자식 노드보다 크거나 같은 힙이다.
adjust는 root 값을 temp에 보관한 뒤 두 자식 중 더 큰 자식을 골라 위로 올리면서 내려가고, 마지막 빈 자리에 temp를 넣어 최대 힙 성질을 회복한다.

NoteBase

선지별 해설

1번: X. `(가)`의 `rootkey < a[child]`는 부모가 자식보다 작을 때 중단하게 하므로 최대 힙 조건과 반대이다. 틀린 표현은 `(가)`의 비교 방향이다. 올바르게는 `rootkey > a[child]`일 때 중단한다.
2번: X. `(나)`의 `a[child/2] = a[child]`는 자식을 부모 자리로 올리는 방향은 맞지만, `(가)`가 `rootkey < a[child]`라서 중단 조건이 반대이다. 틀린 표현은 `(가)`의 비교 방향이다. 올바르게는 `rootkey > a[child]`이다.
3번: X. `(가)`의 `rootkey > a[child]`는 맞지만, `(나)`의 `a[child] = a[child/2]`는 부모 값을 자식 칸으로 내려 복사하는 형태라서 sift-down 과정과 맞지 않는다. 틀린 표현은 `(나)`의 대입 방향이다. 올바르게는 `a[child/2] = a[child]`이다.
4번: O. 더 큰 자식을 선택한 뒤 `rootkey > a[child]`이면 부모가 이미 그 자식보다 크므로 최대 힙 조건이 만족되어 `break`한다. 그렇지 않으면 `a[child/2] = a[child]`로 더 큰 자식을 부모 자리로 올리고, 반복이 끝난 뒤 `a[child/2] = temp`로 원래 루트 값을 넣는다.

오답 포인트

이 코드는 매 단계마다 swap을 하는 방식이 아니라 root 값을 temp로 보관하고 큰 자식만 위로 복사하는 방식이다. 그래서 `(가)`의 비교 부등호와 `(나)`의 대입 방향을 따로 보지 말고 한 흐름으로 연결해서 판단해야 한다.

다시 나오면 이렇게 판단

최대 힙 재조정 코드를 보면 먼저 두 자식 중 큰 자식을 고른다. 그다음 부모가 그 큰 자식보다 크면 종료하고, 작거나 같으면 큰 자식을 부모 자리로 올린다고 판단하면 된다.

암기 포인트

최대 힙 adjust는 `큰 자식 선택 -> 부모가 더 크면 종료 -> 큰 자식을 부모로 승격 -> 마지막에 temp 저장` 순서이다.

NoteBase

문 16. <보기 1>의 배열 A는 최소 힙(min heap)을 나타낸다. 이 힙에 <보기 2>의 연산을 차례로 수행했을 때, 배열 A의 최종 결과로 가장 옳은 것은?

<보기 1>

	0	1	2	3	4	5	6	7	8
A		5	30	10	40	60	70	20	50

(0번 원소는 사용하지 않음)

<보기 2>

1. 최소값 삭제
2. 5 삽입

①

	0	1	2	3	4	5	6	7	8
A		5	10	20	30	40	50	60	70

②

	0	1	2	3	4	5	6	7	8
A		5	10	20	30	60	70	50	40

③

	0	1	2	3	4	5	6	7	8
A		5	30	10	40	60	70	20	50

④

	0	1	2	3	4	5	6	7	8
A		5	30	10	40	50	20	70	60

정답 2번

출제 포인트

이 문제는 최소 힙에서 최소값 삭제 후 하향 이동과 새 원소 삽입 후 상향 이동이 어떤 배열 상태를 만드는지를 묻는 문제이다.

올바른 순서

1. 루트 5를 삭제하고 마지막 원소 50을 루트로 올린다.
2. 50을 자식 30, 10과 비교하여 더 작은 10과 바꾸어 [10, 30, 50, 40, 60, 70, 20]이 된다.
3. 인덱스 3의 50을 자식 70, 20과 비교하여 더 작은 20과 바꾸어 [10, 30, 20, 40, 60, 70, 50]이 된다.
4. 값 5를 마지막에 삽입한 뒤 부모 40, 30, 10보다 계속 작으므로 위로 올라가 [5, 10, 20, 30, 60, 70, 50, 40]이 된다.

왜 이 순서: 삭제 연산은 마지막 원소를 루트로 올린 뒤 하향 이동으로 힙을 복구하고, 삽입 연산은 마지막 칸에 넣은 뒤 상향 이동으로 힙을 복구하기 때문이다.

NoteBase

선지별 해설

① X: [5, 10, 20, 30, 40, 50, 60, 70]은 값이 오름차순으로 정렬된 형태에 가깝고 힙의 레벨 순서 배열이 아니다. 삭제 후 50은 인덱스 7로 내려가고, 삽입한 5 때문에 40은 마지막 칸으로 밀려야 한다. 올바르게는 [5, 10, 20, 30, 60, 70, 50, 40]이다.
② O: 최소값 5를 삭제하면 [10, 30, 20, 40, 60, 70, 50]으로 복구되고, 다시 5를 삽입하면 부모보다 작은 동안 위로 올라가 최종 배열이 [5, 10, 20, 30, 60, 70, 50, 40]이 된다.
③ X: [5, 30, 10, 40, 60, 70, 20, 50]은 처음 상태와 같아서 삭제와 삽입에 따른 재배치가 전혀 반영되지 않았다. 삭제 후 루트는 10이 되어야 하고, 인덱스 3의 값도 20으로 바뀌어야 한다. 올바르게는 [5, 10, 20, 30, 60, 70, 50, 40]이다.
④ X: [5, 30, 10, 40, 50, 20, 70, 60]은 삽입된 5의 상향 이동을 반영하지 못해 10이 루트 아래에 그대로 남아 있고, 삭제 과정에서 내려간 50과 20의 위치도 잘못되었다. 올바르게는 [5, 10, 20, 30, 60, 70, 50, 40]이다.

오답 포인트

힙 배열을 정렬 결과로 착각하거나, 삭제 연산과 삽입 연산을 각각 한 번의 교환만으로 끝난다고 생각하면 틀리기 쉽다. 이 문항의 함정은 삭제에서는 하향 이동, 삽입에서는 상향 이동이 끝날 때까지 반복된다는 점이다.

다시 나오면 이렇게 판단

힙 연산 문제는 삭제이면 마지막 원소를 루트로 올린 뒤 더 작은 자식과 끝까지 비교하고, 삽입이면 마지막 칸에 넣은 뒤 부모와 끝까지 비교하는 순서로 판단하면 된다.

암기 포인트

최소 힙은 삭제 시 아래로 내려가며 복구하고, 삽입 시 위로 올라가며 복구한다.

NoteBase

문 17. 크기가 11인 배열 기반의 해시 테이블에 <보기>의 해시 함수 h1(k)와 h2(k)를 사용하는 이중 해싱(double hashing)으로 입력 키를 차례대로 저장할 때 39가 저장되는 테이블의 주소는?

<보기>
•기본 해시 함수 h1(k)=k mod 11
•2차 해시 함수 h2(k)=1 + (k mod 10)
•입력 키: 12, 44, 13, 88, 23, 94, 11, 39

① 2
② 3
③ 4
④ 9

정답 2번

출제 포인트

이 문제는 이중 해싱에서 $h_1(k)$로 기본 주소를 잡고, 충돌이 나면 $h_2(k)$만큼 건너뛰며 첫 빈 칸을 찾는 과정을 정확히 적용할 수 있는지를 묻는다.

조건 정리

해시 테이블 크기는 11이다.
기본 해시 함수는 h_1(k)=k 11이다.
2차 해시 함수는 h_2(k)=1+(k 10)이다.
입력 키는 12, 44, 13, 88, 23, 94, 11, 39의 순서로 저장한다.
구해야 할 값은 39가 저장되는 최종 주소이다.

사용할 공식

충돌이 없으면 $h_1(k)$에 저장한다.
충돌이 있으면 (h_1(k)+i · h_2(k)) 11 순서로 빈 칸을 찾는다.
이 공식을 쓰는 이유는 이중 해싱이 기본 주소와 보조 간격을 함께 사용해 충돌을 해결하는 방식이기 때문이다.

NoteBase

계산 과정

12는 $h_1(12) = 1$이므로 1번 주소에 저장된다.
44는 $h_1(44) = 0$이므로 0번 주소에 저장된다.
13은 $h_1(13) = 2$이므로 2번 주소에 저장된다.
88은 $h_1(88) = 0$에서 충돌하고 $h_2(88) = 9$이므로 다음 주소는 (0+9) 11=9가 되어 9번 주소에 저장된다.
23은 $h_1(23) = 1$에서 충돌하고 $h_2(23) = 4$이므로 다음 주소는 (1+4) 11=5가 되어 5번 주소에 저장된다.
94는 $h_1(94) = 6$이므로 6번 주소에 저장된다.
11은 $h_1(11) = 0$에서 충돌하고 $h_2(11) = 2$이므로 2번 주소도 충돌한 뒤 (0+2 · 2) 11=4가 되어 4번 주소에 저장된다.
39는 $h_1(39) = 6$에서 충돌하고 $h_2(39) = 10$이다.
탐사 순서는 6 -> (6+10) 11=5 -> (6+20) 11=4 -> (6+30) 11=3이다.
6, 5, 4는 이미 차 있으므로 처음 비어 있는 3번 주소에 39가 저장된다.

선지 판단

1번(X): 2는 39의 실제 첫 빈 칸이 아니다. 39의 탐사 순서에서 3번 주소가 먼저 비므로 2가 아니라 3이 맞다.
2번(O): 39는 6에서 시작해 5, 4를 거친 뒤 처음 비어 있는 3번 주소에 저장되므로 옳다.
3번(X): 4는 11이 이미 저장된 주소이다. 39는 4에서 멈추지 못하고 다음 탐사 주소인 3으로 이동하므로 4가 아니라 3이 맞다.
4번(X): 9는 88이 저장된 주소이며 39의 기본 주소도 아니다. 39의 탐사 간격은 10이므로 최종 저장 주소는 9가 아니라 3이다.

오답 포인트

39에서 충돌이 난 뒤 한 번만 더 이동하거나, 충돌한 칸도 후보로 착각하면 오답이 된다. 특히 4번 주소가 중간에 나오지만 이미 11이 들어 있으므로 첫 빈 칸 원칙을 끝까지 적용해야 한다.

다시 나오면 이렇게 판단

이중 해싱 문제는 각 키를 입력 순서대로 실제로 채워 넣으면서, 충돌이 난 키마다 기본 주소와 보조 간격을 구한 뒤 첫 빈 칸이 나올 때까지 탐사 순서를 끝까지 적어 보면 된다.

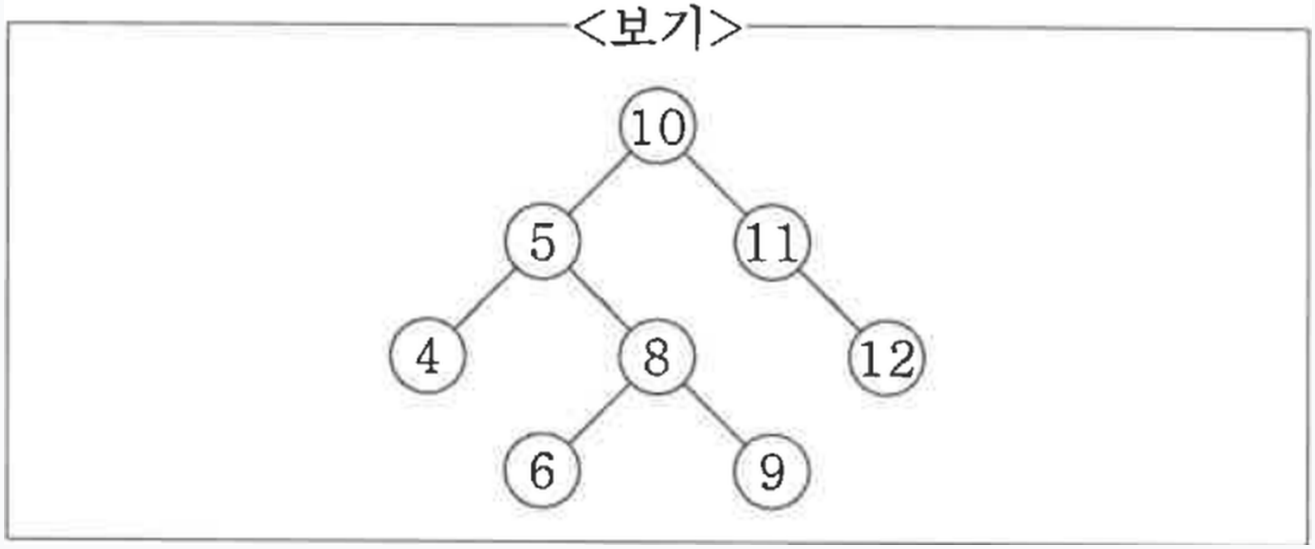
암기 포인트

이중 해싱은 기본 주소는 $h_1(k)$, 이동 간격은 $h_2(k)$, 정답은 첫 빈 칸이라는 순서로 판단한다.

NoteBase

문 18. <보기>의 AVL 트리에 7을 삽입하고, 높이 균형을 유지하기 위해 회전 연산을 수행하였다. 이때, 변화된 AVL 트리에 대한 층위 순회(level-order) 방문 순서는?

<보기>



① 5, 4, 6, 7, 8, 9, 10, 11, 12

② 5, 4, 6, 7, 9, 8, 12, 11, 10

③ 10, 6, 11, 4, 8, 7, 9, 5, 12

④ 10, 6, 11, 5, 8, 12, 4, 7, 9

정답 4번

출제 포인트

이 문제는 AVL 트리에 7을 삽입한 뒤 처음 불균형이 발생한 노드에서 어떤 회전이 일어나는지와, 회전 후 트리의 층위 순회 결과를 판단하는 문제이다.

NoteBase

선지별 해설

① X: 수열이 5로 시작하므로 회전 후에도 왼쪽 부분트리의 루트를 5로 본 것이다. RL 회전 후 5의 자리는 6이 차지하므로 층위 순회의 시작은 10, 6, 11이 되어야 한다. 틀린 표현은 맨 앞의 시작 구조이며, 올바르게는 10, 6, 11, 5, 8, 12, 4, 7, 9이다.
② X: 이 수열도 5로 시작하여 루트 자체를 잘못 잡았고, 뒤의 9, 8, 12, 11, 10 순서도 층위 순회가 아니라 역순에 가깝다. 층위 순회는 위 레벨에서 아래 레벨로, 같은 레벨에서는 왼쪽에서 오른쪽으로 방문한다. 올바르게는 10, 6, 11, 5, 8, 12, 4, 7, 9이다.
③ X: 10, 6, 11까지는 맞지만 그다음에 4를 바로 방문한 것이 틀렸다. 4는 5의 자식이므로 5를 먼저 방문한 뒤 다음 레벨에서 나와야 한다. 틀린 표현은 4, 8, 7, 9, 5, 12 부분의 레벨 혼합이며, 올바르게는 10, 6, 11, 5, 8, 12, 4, 7, 9이다.
④ O: RL 회전 후 트리는 루트 10, 다음 레벨 6과 11, 그다음 레벨 5, 8, 12, 마지막 레벨 4, 7, 9 순서가 된다. 따라서 층위 순회 결과는 10, 6, 11, 5, 8, 12, 4, 7, 9가 맞다.

오답 포인트

이 문항의 함정은 7을 삽입한 뒤 바로 순회만 떠올리거나, 불균형 노드를 10으로 착각하는 데 있다. 실제로는 처음 불균형이 생기는 노드는 5이고, 유형도 RR이나 LL이 아니라 8의 왼쪽으로 들어간 RL 회전이다. 또한 층위 순회는 부모를 모두 방문한 뒤 자식 레벨로 내려간다는 점을 놓치면 ③처럼 4를 5보다 먼저 적게 된다.

다시 나오면 이렇게 판단

BST 삽입 위치를 먼저 찾고, 삽입 후 아래에서 위로 올라오면서 처음 균형 인수 절댓값이 2가 되는 노드를 찾으면 된다. 그 노드에서 삽입 경로가 오른쪽-왼쪽이면 RL, 왼쪽-오른쪽이면 LR로 판단하고, 회전이 끝난 뒤에는 레벨별로 왼쪽에서 오른쪽으로 다시 읽으면 된다.

암기 포인트

AVL 삽입은 처음 깨진 조상에서 회전하고, 층위 순회는 레벨 순서대로 읽는다. 이 문항은 5에서 RL 회전 후 6이 서브트리 루트가 된다.

NoteBase

문 19. <보기 1>은 인자로 들어온 두 집합의 교집합을 구하여 반환하는 재귀함수이다. (가)와 (나)에 들어갈 내용을 가장 옳게 짝지은 것은? (단, 인자로 들어오는 두 집합과 결과인 교집합 모두 <보기 2>의 예시와 같이 연결 리스트로 표현된다.)

<보기 1>

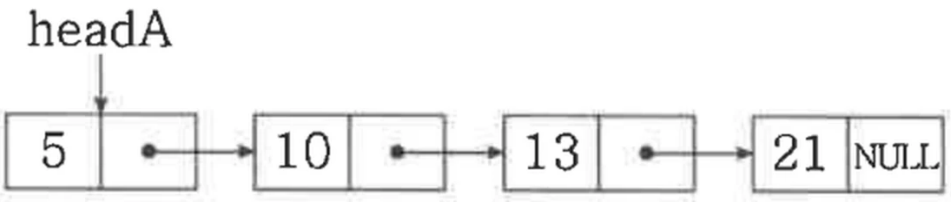
```
typedef struct Node {
  int data;
  struct Node *next;
} Node;        // 노드 정의

// headA와 headB: 두 집합을 표현하는 연결 리스트의 첫 노드의 주소
// 반환 값 p: 두 집합의 교집합을 표현하는 연결 리스트의 첫 노드의 주소
Node *intersect(Node *headA, Node *headB) {
    if (headA == NULL || headB == NULL) // 베이스 케이스
      return NULL;

    if (headA->data > headB->data)
        (가)
    else if (headA->data < headB->data)
        (나)
    else {
        Node *p = (Node *) malloc(sizeof(Node));
        p->data = headA->data;
        p->next = intersect(headA->next, headB->next);
        return p;
    }
}
```

<보기 2>

집합 A={5, 10, 13, 21}을 표현하는 연결 리스트
(원소는 오름차순으로 저장)



① (가) intersect(headA, headB->next);
(나) intersect(headA->next, headB);

② (가) intersect(headA->next, headB);
(나) intersect(headA, headB->next);

③ (가) return intersect(headA, headB->next);
(나) return intersect(headA->next, headB);

NoteBase

④ (가) return intersect(headA->next, headB);
(나) return intersect(headA, headB->next);

정답 3번

출제 포인트

오름차순으로 저장된 두 연결 리스트의 교집합을 재귀로 구할 때는 작은 값을 가진 쪽 포인터만 한 칸 전진시키고, 재귀 호출 결과를 현재 함수의 반환값으로 그대로 돌려주는지를 묻는 문제이다.

선지별 해설

① X: (가)와 (나)의 포인터 이동 방향 자체는 맞다. 그러나 `intersect(...)`만 쓰고 `return`이 없어서 재귀 호출의 결과 주소를 호출자에게 돌려주지 못한다. 틀린 표현은 `intersect(...)`이고, 올바르게는 `(가) return intersect(headA, headB->next);`, `(나) return intersect(headA->next, headB);`이다.
② X: `headA->data > headB->data`이면 더 작은 값은 `headB->data`이므로 `headB`를 전진시켜야 한다. 그런데 ②는 큰 값 쪽인 `headA`를 전진시키므로 아직 맞춰 보지도 않은 값을 버리게 된다. 틀린 표현은 `(가) intersect(headA->next, headB)`와 `(나) intersect(headA, headB->next)`의 배치이고, 올바르게는 두 분기가 서로 반대여야 한다.
③ O: `headA->data > headB->data`일 때는 `(가) return intersect(headA, headB->next);`, `headA->data < headB->data`일 때는 `(나) return intersect(headA->next, headB);`가 된다. 작은 값 쪽만 한 칸 넘기고, 재귀 결과를 그대로 반환하므로 교집합 리스트가 올바르게 이어진다.
④ X: ④는 ②와 마찬가지로 포인터 이동 방향을 반대로 적었다. `headA->data > headB->data`인데 `headA`를 넘기면 더 큰 값을 버리게 되어 이후에 만날 수 있는 공통 원소를 놓칠 수 있다. 틀린 표현은 `(가) return intersect(headA->next, headB);`, `(나) return intersect(headA, headB->next);`이고, 올바르게는 작은 값 쪽을 넘겨야 한다.

오답 포인트

이 문항의 함정은 두 가지이다. 하나는 재귀 호출문 앞의 `return` 유무를 놓치는 것이고, 다른 하나는 비교 결과가 `headA > headB`일 때 큰 값을 넘긴다고 착각하는 것이다. 교집합에서는 작은 값이 현재 상대 값과 같을 가능성이 사라졌을 때만 그 작은 값 쪽을 버린다.

다시 나오면 이렇게 판단

오름차순 두 리스트를 비교하는 재귀 문제는 먼저 `같으면 둘 다 이동`, `다르면 작은 값 쪽만 이동`이라는 규칙을 세운다. 그다음 분기문이 값을 어디서 버리는지 확인하고, 함수 반환형이 포인터이면 재귀 호출 앞에 `return`이 필요한지도 함께 점검하면 된다.

NoteBase

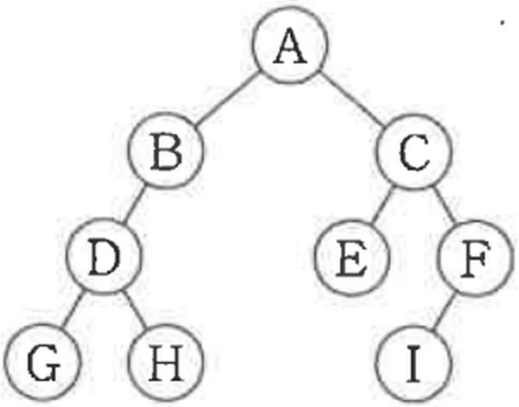
암기 포인트

교집합 재귀는 작은 값만 버리고, 재귀 결과 주소는 반드시 `return`한다.

NoteBase

문 20. <보기 1>의 이진 트리의 루트(root)가 A노드일 때, <보기 2>의 traverse_X(root)를 수행하였다. 노드의 출력 순서로 가장 옳은 것은?

<보기 1>



<보기 2>

```
alg traverse_X(노드 v) {
    if (v == NULL) return;
    노드 v 출력;
    traverse_Y(v의 왼쪽 자식);
    traverse_Y(v의 오른쪽 자식);
}
alg traverse_Y(노드 v) {
    if (v == NULL) return;
    traverse_X(v의 왼쪽 자식);
    traverse_X(v의 오른쪽 자식);
    노드 v 출력;
}
```

① A - B - D - G - H - C - E - F - I

② A - G - H - D - B - E - I - F - C

③ A - D - G - H - B - E - F - I - C

④ A - D - G - H - B - E - I - F - C

정답 3번

NoteBase

출제 포인트

이 문제는 traverse_X와 traverse_Y의 출력 시점이 서로 다를 때 이진 트리의 실제 방문 순서를 정확히 추적할 수 있는지를 묻는다.

선지별 해설

1번: X. A 다음에 B와 C를 바로 출력한 것이 틀리다. B는 traverse_Y(B)의 마지막에 출력되고, C도 traverse_Y(C)의 마지막에 출력된다. 틀린 표현은 "A-B-D-G-H-C-E-F-I"이고, 올바르게는 "A-D-G-H-B-E-F-I-C"이다.
2번: X. D에서 traverse_X가 시작되므로 D를 먼저 출력해야 하는데, G-H-D처럼 후위식으로 적은 것이 틀리다. 또한 F도 I보다 먼저 출력되어야 한다. 틀린 표현은 "G-H-D"와 "I-F"이고, 올바르게는 "D-G-H"와 "F-I"이다.
3번: O. traverse_X(A)에서 먼저 A를 출력하고, 왼쪽의 traverse_Y(B)는 D-G-H-B, 오른쪽의 traverse_Y(C)는 E-F-I-C 순서가 된다. 따라서 전체 출력은 A-D-G-H-B-E-F-I-C가 된다.
4번: X. 왼쪽 부분 A-D-G-H-B까지는 맞지만, F에서 traverse_X가 먼저 F를 출력한 뒤 traverse_Y(I)로 I를 출력하므로 오른쪽 끝은 I-F가 아니라 F-I가 된다. 틀린 표현은 "I-F"이고, 올바르게는 "F-I"이다.

오답 포인트

traverse_X를 전위순회, traverse_Y를 후위순회처럼 단순 치환하면 틀리기 쉽다. 이 문제의 함정은 부모 함수와 자식 함수가 서로 다르다는 점이며, 특히 B와 C는 Y의 마지막에, D와 F는 X의 시작에 나온다는 점을 놓치면 오답이 된다.

다시 나오면 이렇게 판단

현재 함수가 X인지 Y인지 먼저 표시한 뒤, X는 현재 노드 선출력, Y는 현재 노드 후출력으로 고정해서 부분트리 단위로 적어 나가면 된다. 각 노드에서 자식 호출이 X인지 Y인지까지 함께 써 두면 순서가 흔들리지 않는다.

암기 포인트

X는 자기 자신 먼저, Y는 자기 자신 나중이라는 기준만 잡으면 상호 재귀 순회도 출력 순서를 복원할 수 있다.

NoteBase