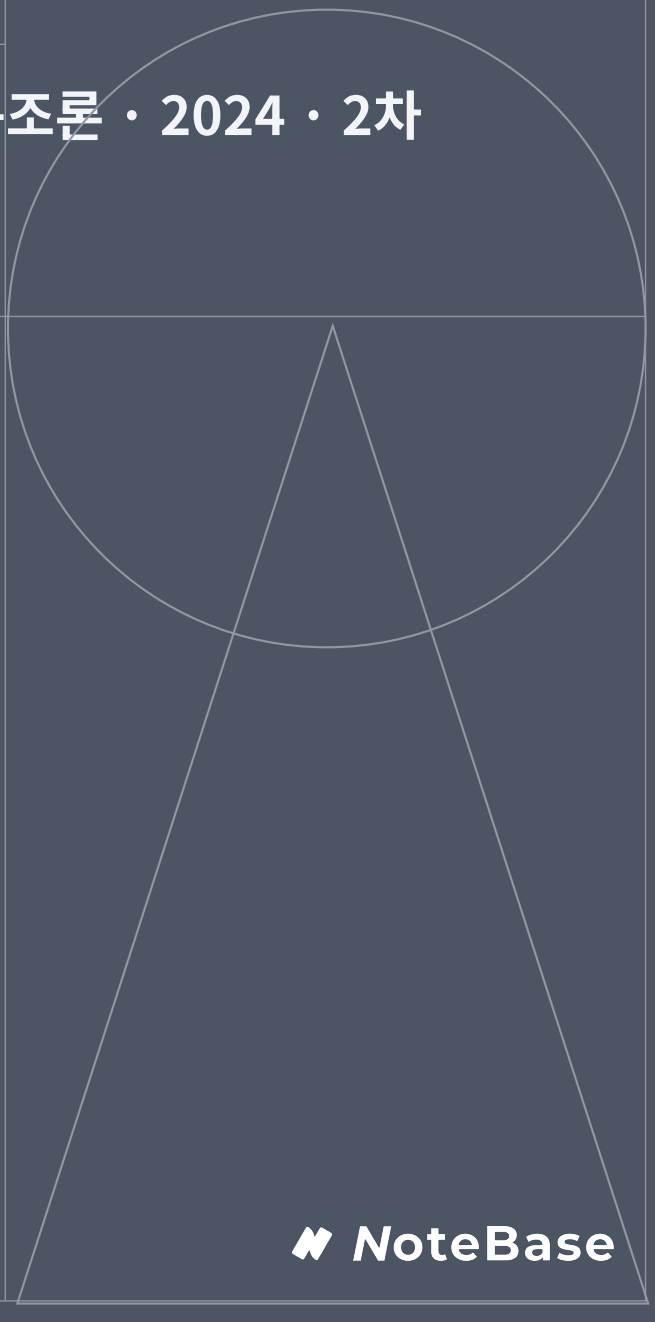
기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2024 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2024 · 2차

공무원/국가직/7급 · 자료구조론 · 2024 · 2차

문 1. 배열로 구현한 최대 힙(max heap)에 키 값 <15, 10, 12, 8, 9, 7, 20>을 차례대로 삽입하였을 때, 값 7이 저장되어 있는 색인(index) 번호는? (단, 배열에 저장되는 키 값의 첫 번째 저장위치 색인 번호는 1이다)

① 4

② 5

③ 6

④ 7

정답 3번

출제 포인트

이 문제는 배열로 구현한 최대 힙에서 원소를 순서대로 삽입할 때 마지막에 들어간 원소만 상향 이동한다는 점과 각 키의 최종 인덱스를 추적할 수 있는지를 묻는다.

조건 정리

주어진 삽입 순서는 15, 10, 12, 8, 9, 7, 20이다.
배열 인덱스는 1부터 시작한다.
배열 기반 최대 힙은 부모 인덱스가 i이면 왼쪽 자식은 $2i$, 오른쪽 자식은 $2i + 1$이다.
새 원소는 배열의 마지막 칸에 들어간 뒤, 부모보다 클 때만 위로 올라간다.

사용할 공식

힙 삽입 규칙은 마지막 위치 삽입 후 상향 이동이다.
부모와 자식의 인덱스 관계는 $i \rightarrow 2i$, $2i + 1$이다.
이 규칙을 쓰는 이유는 7의 위치가 마지막 삽입 원소 20의 상향 이동 경로에 포함되는지만 확인하면 최종 인덱스를 바로 판단할 수 있기 때문이다.

NoteBase

계산 과정

1. 15, 10, 12, 8, 9, 7을 차례대로 넣으면 최대 힙은 [15, 10, 12, 8, 9, 7]이 된다. 이때 7은 인덱스 6에 있다.
2. 20을 마지막 칸인 인덱스 7에 넣으면 [15, 10, 12, 8, 9, 7, 20]이 된다.
3. 20은 부모 12보다 크므로 인덱스 3과 교환되어 [15, 10, 20, 8, 9, 7, 12]가 된다.
4. 20은 다시 부모 15보다 크므로 루트와 교환되어 [20, 10, 15, 8, 9, 7, 12]가 된다.
5. 7은 상향 이동 경로에 없으므로 끝까지 인덱스 6에 남는다.

선지 판단

1번 X: 인덱스 4에는 8이 저장된다. 7은 여섯 번째로 삽입될 때 인덱스 6에 들어가고, 이후에도 부모 12보다 작아서 이동하지 않는다. 틀린 표현은 '7의 인덱스가 4이다'이고, 올바르게는 '7의 인덱스는 6이다'이다.
2번 X: 인덱스 5에는 9가 저장된다. 7과 9의 위치를 혼동한 경우이지만, 삽입 순서상 9가 먼저 인덱스 5를 차지한다. 틀린 표현은 '7의 인덱스가 5이다'이고, 올바르게는 '7의 인덱스는 6이다'이다.
3번 O: 7은 15, 10, 12, 8, 9 다음에 삽입되어 인덱스 6에 저장된다. 마지막에 20이 올라갈 때도 7이 있는 경로를 건드리지 않으므로 최종 인덱스는 그대로 6이다.
4번 X: 인덱스 7에는 마지막 상향 이동 뒤 12가 남는다. 20이 위로 올라가면서 12가 인덱스 7로 내려갈 뿐 7이 인덱스 7로 밀리는 일은 없다. 틀린 표현은 '7의 인덱스가 7이다'이고, 올바르게는 '7의 인덱스는 6이다'이다.

오답 포인트

마지막에 20이 삽입되면 힙 전체가 다시 정렬된다고 생각하면 틀린다. 힙 삽입에서는 새 원소가 지나가는 부모 경로만 바뀌므로, ③의 7은 그 경로 밖에 있어 그대로 인덱스 6에 남는다.

다시 나오면 이렇게 판단

배열 힙 문제는 '마지막 칸 삽입 -> 부모와 비교하며 상향 이동 -> 이동 경로에 없는 노드는 그대로 유지'의 순서로 판단한다. 찾는 값이 상향 이동 경로에 없으면 처음 배치된 인덱스가 최종 인덱스가 된다.

암기 포인트

배열 최대 힙 삽입은 끝에 넣고 위로만 올린다. 올라가는 것은 새 원소가 지나가는 경로뿐이므로 경로 밖 원소의 인덱스는 유지된다.

NoteBase

문 2. 비어 있는 이진 탐색 트리(binary search tree)에 키 값 <7, 5, 1, 8, 3, 6, 0, 9, 4, 2>를 순서대로 삽입한 후, 중위 순회(in-order traversal)한 결과는?

① 0 1 2 3 4 5 6 7 8 9

② 0 2 4 3 1 6 5 9 8 7

③ 7 5 1 0 3 2 4 6 8 9

④ 9 8 6 4 2 3 0 1 5 7

정답 1번

출제 포인트

이 문제는 이진 탐색 트리에 키를 순서대로 삽입한 뒤 중위 순회 결과가 어떤 방문 순서가 되는지를 묻는다.

올바른 순서

1. 루트 7을 만든다.
2. 각 키를 대소 비교에 따라 왼쪽 또는 오른쪽 자식 위치에 삽입해 이진 탐색 트리를 완성한다.
3. 중위 순회는 왼쪽 부분트리 -> 루트 -> 오른쪽 부분트리 순서로 방문한다.
4. 모든 키가 서로 다른 이진 탐색 트리의 중위 순회 결과는 오름차순이 되어 0 1 2 3 4 5 6 7 8 9가 된다.

왜 이 순서: 이진 탐색 트리에서는 왼쪽 부분트리의 모든 값이 루트보다 작고 오른쪽 부분트리의 모든 값이 루트보다 크므로, 왼쪽 -> 루트 -> 오른쪽으로 방문하면 값이 작은 것부터 큰 것까지 정렬된 순서가 된다.

선지별 해설

1번: O. 중위 순회는 왼쪽 부분트리 -> 루트 -> 오른쪽 부분트리 순서이므로, 서로 다른 키로 만든 이진 탐색 트리에서는 항상 오름차순이 나온다. 따라서 0 1 2 3 4 5 6 7 8 9가 맞다.

2번: X. "0 2 4 3 1 6 5 9 8 7"이 틀린 이유는 중위 순회 결과라면 1 앞에 3과 4가 올 수 없고, 마지막에 루트 7이 오는 형태도 성립하지 않기 때문이다. 올바르게는 0 1 2 3 4 5 6 7 8 9이다.

3번: X. "7 5 1 0 3 2 4 6 8 9"는 루트를 먼저 방문한 뒤 좌우 부분트리로 내려가는 전위 순회 결과에 가깝다. 중위 순회는 루트를 중간에 방문하므로 올바르게는 0 1 2 3 4 5 6 7 8 9이다.

4번: X. "9 8 6 4 2 3 0 1 5 7"은 큰 값이 먼저 나오고 중간에도 오름차순 규칙이 깨져 중위 순회 결과가 될 수 없다. 중위 순회라면 항상 0 1 2 3 4 5 6 7 8 9 순서가 된다.

오답 포인트

삽입 순서가 복잡하면 순회 결과도 뒤섞일 것이라고 착각하기 쉽다. 이 문제의 함정은 삽입 과정 자체보다 이진 탐색 트리의 성질과 중위 순회 규칙을 연결할 수 있는지를 보는 데 있다.

NoteBase

다시 나오면 이렇게 판단

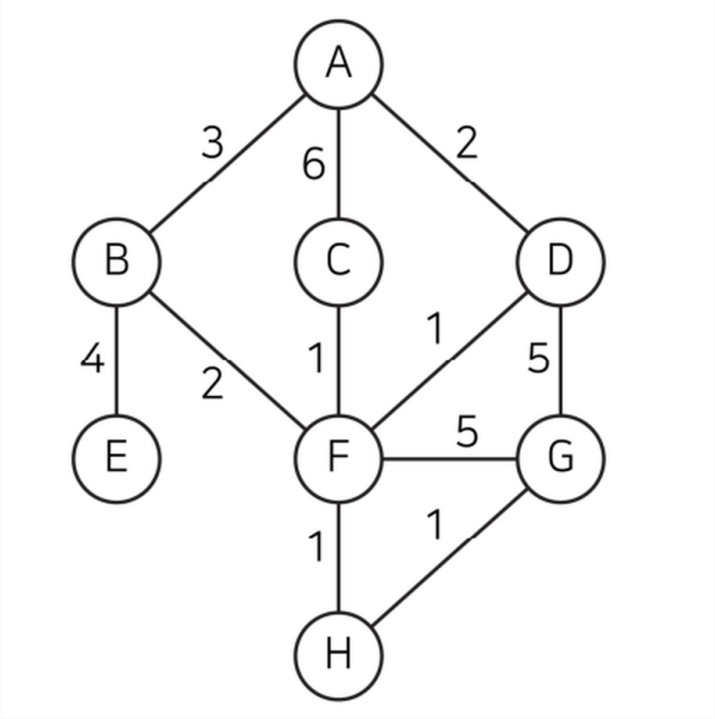
이진 탐색 트리와 중위 순회가 함께 나오면 먼저 왼쪽 부분트리 < 루트 < 오른쪽 부분트리 관계를 떠올린다.
그다음 중위 순회가 왼쪽 -> 루트 -> 오른쪽이라는 점을 적용해 결과가 오름차순인지부터 판단한다.

암기 포인트

이진 탐색 트리의 중위 순회 결과는 오름차순이다.

NoteBase

문 3. 다음 그래프의 정점 A에서 시작하여 다른 모든 정점까지의 최단 경로를 찾고자 한다.
다익스트라(Dijkstra) 알고리즘을 사용하여 찾은 정점 G까지의 최단 경로를 바르게 나열한 것은?



① A → B → F → G

② A → C → F → G

③ A → D → F → G

④ A → D → F → H → G

정답 4번

출제 포인트

이 문제는 가중치 그래프에서 시작 정점 A로부터 각 정점까지의 최단거리 갱신을 따라가며 정점 G까지의 실제 최단 경로를 판단하는 문제이다.

NoteBase

선지별 해설

① X
A→B→F→G의 총비용은 3+2+5=10이다. 더 짧은 경로가 이미 존재하므로 최단 경로가 아니다. 틀린 표현은 "최단 경로"이고, 올바르게는 "A→B→F→G는 비용 10의 경로이다"이다.

② X
A→C→F→G의 총비용은 6+1+5=12이다. C를 거치면 A→D를 먼저 거치는 경우보다 시작부터 비용이 크다. 틀린 표현은 "최단 경로"이고, 올바르게는 "A→C→F→G는 비용 12의 경로이다"이다.

③ X
A→D→F→G의 총비용은 2+1+5=8이다. D까지는 빠르지만 G로 바로 가는 간선의 가중치가 5라서 H를 거치는 경로보다 길다. 틀린 표현은 "최단 경로"이고, 올바르게는 "A→D→F→G는 비용 8의 경로이다"이다.

④ O
A→D→F→H→G의 총비용은 2+1+1+1=5이다. A에서 D까지 2, D에서 F까지 1, F에서 H까지 1, H에서 G까지 1로 누적하면 5가 되어 보기들 중 가장 짧다.

오답 포인트

③이 가장 흔한 함정이다. A→D→F까지의 누적거리 3이 매우 작아서 곧바로 F→G를 붙이기 쉽지만, F→G의 가중치 5보다 F→H→G의 1+1이 더 작다. 다익스트라는 중간 정점까지의 최단거리만 보지 않고 마지막 연결 간선까지 포함한 전체 누적거리를 비교해야 한다.

다시 나오면 이렇게 판단

시작 정점에서 가까운 정점부터 최단거리를 확정하고, 각 단계에서 인접 정점을 거쳐 가는 새로운 누적거리와 기존 값을 비교하면 된다. 목적지로 바로 가는 간선이 보여도 우회 경로의 누적합이 더 작을 수 있으므로 끝 간선의 가중치를 반드시 다시 합산해야 한다.

암기 포인트

다익스트라는 "현재 최단거리 + 다음 간선"의 누적합으로 판단하며, 직행 간선보다 짧은 우회 경로가 나올 수 있다.

NoteBase

문 4. 다음 C 코드에서 밑줄 친 'x=x+3;'의 수행 빈도수는? (단, n은 0 이상의 정수이다)

```
void complexity(int n){
        int i, j, x=0;
        for (i=0; i<=n; i++) {
                for (j=i; j<=n; j++)
                        x=x+3;
        }
}
```

① $\frac{n(n-1)}{2}$
② $\frac{n(n+1)}{2}$
③ $\frac{(n-1)(n+1)}{2}$
④ $\frac{(n+1)(n+2)}{2}$

정답 4번

출제 포인트

이 문제는 중첩 for문의 각 반복 구간 길이를 합해 문장 x=x+3;의 총 수행 횟수를 구하는 문제이다.

조건 정리

주어진 값: 바깥 반복문은 $i = 0$부터 n까지이므로 총 $n + 1$번 돈다.
구해야 할 값: 문장 x=x+3;의 총 수행 횟수이다.
주의할 조건: 안쪽 반복문은 각 i마다 $j = i$부터 n까지이므로 횟수가 $n - i + 1$번이며, 시작값과 끝값이 모두 포함된다.

NoteBase

사용할 공식

각 i에서 안쪽 반복 횟수는 $n-i+1$이므로 전체 횟수는
$\sum_{i=0}^{n}(n-i+1)=\sum_{k=1}^{n+1}k=\frac{(n+1)(n+2)}{2}$
이다.

이 공식을 쓰는 이유는 안쪽 반복 길이가 $n+1, n, \dots, 1$로 1씩 줄어드는 합이기 때문이다.

계산 과정

1. $i=0$일 때 안쪽 반복은 $j=0$부터 $j=n$까지이므로 $n+1$번 수행된다.
2. $i=1$일 때는 n번, 계속 줄어들어 $i=n$일 때는 1번 수행된다.
3. 따라서 총 수행 횟수는 $(n+1)+n+\cdots+1$이다.
4. 이를 합하면 $\frac{(n+1)(n+2)}{2}$이므로 정답은 4번이 된다.

선지 판단

1번 X: $\frac{n(n-1)}{2}$는 항의 개수를 너무 적게 본 값이다. 이 코드의 반복 길이는 $n+1, n, \dots, 1$이므로 올바르게는 $\frac{(n+1)(n+2)}{2}$가 된다.
2번 X: $\frac{n(n+1)}{2}$는 1부터 n까지의 합이라서 $i=0$일 때의 $n+1$번 수행을 빠뜨린 값이다. 올바르게는 1부터 $n+1$까지를 더해야 한다.
3번 X: $\frac{(n-1)(n+1)}{2}$는 양 끝 포함 조건을 놓쳐 반복 횟수를 줄여 잡은 값이다. 안쪽 반복은 $j=i$와 $j=n$이 모두 포함되므로 각 구간 길이는 $n-i+1$이 되어야 한다.
4번 O: 전체 수행 횟수는 $\sum_{i=0}^{n}(n-i+1)=\frac{(n+1)(n+2)}{2}$이므로 맞는 선지이다.

오답 포인트

안쪽 반복문의 시작값이 $j=i$라는 점을 무시하고 단순히 1부터 n까지의 합으로 처리하면 2번 같은 오답이 나온다. 또한 $i=0, j=n$이 모두 포함되는 양 끝 포함 범위를 빠뜨리면 1번이나 3번처럼 항 수를 잘못 세게 된다.

다시 나오면 이렇게 판단

중첩 반복문의 실행 횟수는 바깥 변수를 하나 고정한 뒤 안쪽 반복 길이를 식으로 적고, 그 길이들을 처음값부터 끝값까지 모두 합해 판단한다. 반복문의 시작값과 끝값이 포함되는지도 함께 확인한다.

암기 포인트

중첩 반복문은 바깥 변수별 안쪽 길이의 합으로 계산한다. for문에 $\leq$가 나오면 양 끝이 모두 포함된다.

NoteBase

문 5. 다음 C 코드로 표현한 알고리즘의 시간 복잡도는?

```
for (i=1; i<n; i++) {
      MaxData=arr[i];
      for (j=i-1; j>=0; j--) {
            if (arr[j]>MaxData)
                  arr[j+1]=arr[j];
            else
                  break;
      }
      arr[j+1]=MaxData;
}
```

① $O(\log n)$

② $O(n\log n)$

③ $O(n^2)$

④ $O(n^3)$

정답 3번

출제 포인트

이 문제는 주어진 C 코드가 삽입 정렬의 내부 동작이라는 점을 읽고, 중첩 반복문의 최악 수행 횟수로 시간 복잡도를 판단하는 문제이다.

조건 정리

외부 반복문은 $i=1$부터 $n-1$까지 돈다.
내부 반복문은 $j=i-1$부터 0까지 감소하며, 현재 값보다 큰 원소를 뒤로 한 칸씩 민다.
이 구조는 정렬된 앞부분에 현재 원소를 끼워 넣는 삽입 정렬이다.

NoteBase

사용할 공식

최악의 경우 내부 반복문의 총 실행 횟수는 $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$이다.
이 공식을 쓰는 이유는 역순 배열에서는 i번째 단계마다 현재 원소가 맨 앞까지 이동하여 내부 반복문이 최대 i번 수행되기 때문이다.

계산 과정

1. 바깥 반복은 총 $n-1$번 수행된다.
2. 최악의 경우 $i=1$일 때 1번, $i=2$일 때 2번, ..., $i=n-1$일 때 $n-1$번 비교·이동이 일어난다.
3. 전체 수행 횟수는 $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$이므로 차수는 n^2이고 시간 복잡도는 $O(n^2)$이다.

선지별 해설

1번 X: $O(\log n)$은 탐색 범위가 절반씩 줄어드는 이진 탐색류에서 나오는 복잡도이다. 이 코드는 각 단계에서 앞쪽 원소들을 선형으로 밀 수 있으므로 해당하지 않는다. 틀린 표현은 "$O(\log n)$"이다. 올바르게는 최악 시간 복잡도는 $O(n^2)$이다.
2번 X: $O(n \log n)$은 분할 정복이나 힙 구조처럼 각 단계 비용과 단계 수가 곱해지는 정렬에서 주로 나온다. 이 코드는 삽입 정렬이라서 원소 이동이 누적되어 선형 로그형이 아니다. 틀린 표현은 "$O(n \log n)$"이다.
올바르게는 최악 시간 복잡도는 $O(n^2)$이다.
3번 O: 이 코드는 삽입 정렬이다. 역순 입력에서는 i번째 단계마다 최대 i번 비교·이동이 발생하므로 전체가 $\sum_{i=1}^{n-1} i$가 되고, 따라서 시간 복잡도는 $O(n^2)$이다.
4번 X: $O(n^3)$이 되려면 삼중 중첩에 가까운 누적 반복이 필요하다. 이 코드는 실질적으로 이중 반복 구조이며 내부 반복의 총합이 $1+2+\cdots+(n-1)$ 수준이어서 세제곱까지 커지지 않는다. 틀린 표현은 "$O(n^3)$"이다.
올바르게는 최악 시간 복잡도는 $O(n^2)$이다.

오답 포인트

코드에 break가 있으므로 항상 빠르게 끝난다고 보고 $O(n \log n)$ 이하로 낮춰 잡기 쉽다. 그러나 시간 복잡도는 보통 최악의 경우를 기준으로 보며, 역순 배열에서는 break가 거의 작동하지 않아 삽입 정렬의 $O(n^2)$가 그대로 드러난다.

다시 나오면 이렇게 판단

정렬 코드의 시간 복잡도는 먼저 어떤 정렬인지 식별하고, 그다음 최악 입력에서 각 반복문이 몇 번 도는지를 합산해 판단한다. 바깥 n번과 안쪽 최대 i번이 누적되면 $1+2+\cdots+n$ 꼴인지부터 확인하면 된다.

암기 포인트

삽입 정렬은 앞부분에 끼워 넣는 정렬이며, 최악은 역순 입력에서 $O(n^2)$, 최선은 이미 정렬된 경우 $O(n)$이다.

NoteBase

문 6. 다음 자료에 대하여 퀵 정렬(quick sort)로 오름차순 정렬을 하고자 한다. 기준값인 피벗(pivot)으로 정렬 대상 자료의 첫 번째 원소를 사용할 때, 두 번째 분할(partition) 연산 수행 후의 결과는?

< 27, 5, 37, 4, 61, 11, 59, 15, 48, 20 >

① 4 5 11 20 15 27 59 61 48 37

② 4 5 11 15 20 27 59 61 48 37

③ 4 5 11 15 20 27 59 37 48 61

④ 4 5 11 20 15 27 59 37 48 61

정답 1번

출제 포인트

첫 원소를 피벗으로 쓰는 퀵 정렬에서 partition이 끝날 때 피벗이 제자리를 찾고, 두 번째 partition이 첫 partition의 왼쪽 부분배열에 적용된다는 점을 묻는 문제이다.

올바른 순서

1. 첫 partition에서 피벗 27을 기준으로 작은 값과 큰 값을 양쪽으로 교환하여 피벗 27을 제자리에 둔다.
2. 첫 partition 결과는 11 5 20 4 15 27 59 61 48 37이 된다.
3. 두 번째 partition은 피벗 27의 왼쪽 부분배열 11 5 20 4 15에 대해 첫 원소 11을 피벗으로 다시 수행한다.
4. 11보다 작은 값 5와 4는 왼쪽으로, 큰 값 20과 15는 오른쪽으로 가서 4 5 11 20 15가 되고, 전체 배열은 4 5 11 20 15 27 59 61 48 37이 된다.

왜 이 순서인가: 두 번째 partition의 입력은 전체 배열이 아니라 첫 partition으로 분리된 왼쪽 부분배열이기 때문이다.

선지별 해설

1번: O. 첫 partition 뒤 배열은 11 5 20 4 15 27 59 61 48 37이 된다. 여기서 왼쪽 부분배열 11 5 20 4 15만 다시 partition하면 4 5 11 20 15가 되므로 전체 결과는 4 5 11 20 15 27 59 61 48 37이 된다.

2번: X. 틀린 부분은 20 15를 15 20으로 완전히 정렬된 것처럼 적은 점이다. 올바르게는 두 번째 partition까지는 피벗 11의 위치만 확정되고 오른쪽 부분 20 15의 내부 순서는 아직 유지되므로 4 5 11 20 15 27 59 61 48 37이 된다.

3번: X. 틀린 부분은 오른쪽 부분배열을 59 37 48 61로 바꾸어 오른쪽 partition 결과를 미리 반영한 점이다. 올바르게는 두 번째 partition 직후 오른쪽 부분배열은 첫 partition 결과 그대로 59 61 48 37이다.

4번: X. 틀린 부분은 왼쪽 부분배열의 두 번째 partition 결과와 오른쪽 부분배열의 추가 partition 결과를 한 배열에 섞은 점이다. 올바르게는 두 번째 partition까지만 수행하면 4 5 11 20 15 27 59 61 48 37이 된다.

NoteBase

오답 포인트

퀵 정렬의 partition을 한 번 수행했다고 부분배열 전체가 정렬된다고 보면 2번처럼 20 15를 15 20으로 바꾸는 오답이 나온다. 또한 두 번째 partition을 왼쪽과 오른쪽에 모두 적용한 것으로 착각하면 3번이나 4번처럼 아직 수행하지 않은 오른쪽 부분배열 결과를 섞게 된다.

다시 나오면 이렇게 판단

partition 횟수는 피벗 하나가 제자리를 찾는 한 번의 연산으로 센다. 각 회차마다 어떤 부분배열이 현재 입력인지 먼저 표시하고, 그 부분만 갱신한 뒤 나머지 구간은 그대로 둔 상태로 다음 선지를 판단하면 된다.

암기 포인트

퀵 정렬의 partition은 전체 정렬이 아니라 피벗 한 개의 자리 확정이다. 다음 partition은 분리된 한 부분배열에만 적용된다.

NoteBase

문 7. 다음 재귀함수를 호출하는 C 프로그램의 실행 결과는?

```
#include <stdio.h>
void RecursiveFunc(int n) {
	if (n <= 0)  {
		printf("R-%d ", n);
		return;
	}
	printf("R-%d ", n);
	if (n%2 == 0)
		RecursiveFunc(n-2);
	else
		RecursiveFunc(n-1);

	printf("R-%d ", n);
}

int main(void)
{
	RecursiveFunc(3);
	return 0;
}
```

① R-3 R-2 R-0

② R-3 R-1 R-0 R-2 R-3

③ R-3 R-2 R-0 R-2 R-3

④ R-3 R-2 R-1 R-2 R-3

정답 3번

출제 포인트

이 문제는 재귀 함수에서 호출 직전 출력과 호출 복귀 직후 출력을 구분하여 전체 실행 순서를 추적할 수 있는지를 묻는다.

NoteBase

올바른 순서

1. RecursiveFunc(3) 진입 직후 R-3을 출력한다.
2. 3은 홀수이므로 RecursiveFunc(2)를 호출한다.
3. RecursiveFunc(2) 진입 직후 R-2를 출력한다.
4. 2는 짝수이므로 RecursiveFunc(0)을 호출하고, 기저 조건에서 R-0을 출력한 뒤 바로 복귀한다.
5. RecursiveFunc(2)로 돌아와 마지막 R-2를 출력하고 복귀한다.
6. RecursiveFunc(3)으로 돌아와 마지막 R-3을 출력한다.

왜 이 순서: 각 호출은 재귀 호출 전에도 한 번 출력하고, 하위 호출이 끝나 복귀한 뒤에도 같은 n을 다시 출력하기 때문이다.

선지별 해설

1번: X. R-3, R-2, R-0까지만 있고 복귀 후 출력되는 R-2, R-3이 빠져 있다. 틀린 표현은 출력이 한 번만 일어난다고 본 점이다. 올바르게는 R-3 R-2 R-0 R-2 R-3이다.
2번: X. 첫 호출이 3이므로 먼저 R-3을 출력한 뒤 홀수 분기에서 2로 내려간다. 틀린 표현은 두 번째 값이 R-1이라는 점이다. 올바르게는 3에서 바로 2로 호출되어 R-3 R-2 R-0 R-2 R-3이 된다.
3번: O. 3에서 시작해 호출 직전 R-3을 출력하고, 홀수이므로 2로 내려가 R-2를 출력한다. 2는 짝수이므로 0으로 내려가 R-0을 출력한 뒤 복귀하면서 R-2, R-3을 차례로 출력하므로 전체 결과가 R-3 R-2 R-0 R-2 R-3이 된다.
4번: X. 2는 짝수이므로 n-2가 호출되어 0으로 바로 간다. 틀린 표현은 중간에 R-1이 나온다고 본 점이다. 올바르게는 2 다음 호출이 1이 아니라 0이어서 R-3 R-2 R-0 R-2 R-3이다.

오답 포인트

홀수 분기와 짝수 분기를 한 번에 섞어 읽으면 3 다음에 1이 나온다고 착각하기 쉽다. 또 재귀 호출이 끝난 뒤 마지막 printf가 다시 실행된다는 점을 놓치면 뒤쪽의 R-2, R-3을 누락하게 된다.

다시 나오면 이렇게 판단

재귀 실행 결과 문제는 각 호출에서 수행되는 문장을 위에서 아래로 적고, 호출 전 출력과 복귀 후 출력을 분리해 추적하면 된다. 분기 조건이 있으면 현재 n에서 실제로 호출되는 다음 인자값을 먼저 확정한 뒤 내려가면 순서가 흔들리지 않는다.

암기 포인트

재귀 출력은 진입 출력, 하위 호출, 복귀 출력을 분리하면 된다. 홀수는 n-1, 짝수는 n-2라는 분기 기준이 이 문항의 핵심이다.

NoteBase

문 8. 다음은 병합 정렬(merge sort) 알고리즘을 구현한 파이썬 코드이다. (가), (나)에 들어갈 내용을 바르게 연결한 것은?

```
sorted = []

def merge_sort(arr, left, right):
    if left < right:
        middle = (left + right) // 2
        merge_sort(arr, left, middle)
        merge_sort(arr, middle + 1, right)
        merge(arr, left, middle, right)

def merge(arr, left, middle, right):
    global sorted
    i = left
    j = middle + 1
    k = (가)
    while i <= middle and (나):
        if arr[i] <= arr[j]:
            sorted.insert(k, arr[i])
            i = i + 1
        else:
            sorted.insert(k, arr[j])
            j = j + 1
        k = k + 1

    if i > middle:
        sorted[k:k + right - j + 1] = arr[j:right + 1]
    else:
        sorted[k:k + middle - i + 1] = arr[i:middle + 1]

    arr[left:right + 1] = sorted[left:right + 1]
```

① (가): left, (나): j<=right

② (가): left, (나): k<=right

③ (가): middle, (나): j<=right

④ (가): middle, (나): k<=right

정답 1번

출제 포인트

이 문제는 병합 정렬의 merge 단계에서 보조 배열의 시작 인덱스와 반복 조건을 정확히 연결할 수 있는지를 묻는다.

사례 정리

merge(arr, left, middle, right)는 이미 정렬된 두 부분배열 arr[left:middle+1]과 arr[middle+1:right+1]을 하나로 합치는 단계이다.
i는 왼쪽 부분배열의 현재 위치이고 j는 오른쪽 부분배열의 현재 위치이며 k는 보조 배열 sorted에 값을 써 넣는 위치이다.

NoteBase

핵심 개념

병합 과정은 두 부분배열의 현재 원소를 비교해 더 작은 값을 보조 배열의 현재 위치에 넣는 작업이다.
따라서 k는 병합 구간의 첫 위치인 left에서 시작해야 하고, while문은 두 부분배열이 모두 남아 있을 때만 돌아가므로 조건은 i<=middle and j<=right가 된다.

적용 기준

마지막에 arr[left:right+1] = sorted[left:right+1]로 복사하므로 sorted에서도 병합 결과가 left부터 채워져 있어야 한다.
또한 비교문 if arr[i] <= arr[j]를 수행하려면 왼쪽 포인터 i와 오른쪽 포인터 j가 각각 자신의 구간 안에 남아 있어야 한다.

선지별 해설

1번: O. (가)는 left이고 (나)는 j<=right이다. k는 병합 결과를 sorted[left:right+1] 구간에 채우는 인덱스이므로 left에서 시작해야 한다. while 조건도 왼쪽 구간과 오른쪽 구간에 원소가 모두 남아 있을 때만 비교를 계속해야 하므로 i<=middle and j<=right가 맞다.
2번: X. 틀린 표현은 (나)를 k<=right로 둔 부분이다. k는 출력 위치일 뿐 오른쪽 부분배열에 비교할 원소가 남아 있는지를 보장하지 못한다. 올바르게는 j<=right이다.
3번: X. 틀린 표현은 (가)를 middle로 둔 부분이다. k를 middle에서 시작하면 병합 결과가 구간의 중간부터 기록되어 left부터 right까지의 대응이 깨진다. 올바르게는 left이다.
4번: X. (가)를 middle로 둔 점과 (나)를 k<=right로 둔 점이 모두 틀리다. 병합은 left부터 채우고, 반복 지속 여부는 k가 아니라 오른쪽 포인터 j로 판단해야 한다.

오답 포인트

k를 단순히 보조 배열의 임의 위치라고 보면 middle이나 k<=right 같은 선택지에 끌리기 쉽다. 그러나 병합의 핵심은 원본 구간 left~right와 보조 배열 구간 left~right를 같은 위치에 대응시키는 데 있다.

다시 나오면 이렇게 판단

병합 코드 빈칸 문제는 먼저 각 포인터의 역할을 나누어 본다. i와 j는 두 부분배열의 탐색 포인터이고, k는 결과를 기록하는 포인터이므로 시작점은 병합 구간의 left이다. 반복 조건은 비교에 참여하는 두 포인터가 모두 유효한지로 판단한다.

암기 포인트

merge에서는 i는 왼쪽, j는 오른쪽, k는 결과 구간의 시작점 left에서 출발하고 비교 반복 조건은 i와 j가 모두 범위 안에 있을 때이다.

NoteBase

문 9. 다음은 C 언어 구조체를 이용하여 배열과 단순 연결 리스트로 큐를 표현한 것이다. 아래의 <조건>에 따라 (가), (나) 각각에 대해 Enqueue 함수를 세 번 호출하고 Dequeue 함수를 한 번 호출한 후, 큐에 할당된 메모리 공간의 크기(byte)를 계산한 값으로 바르게 연결한 것은?

(가) 배열을 이용한 큐 구조체

```
typedef struct _Queue {
        int front;
        int rear;
        int numOfData;
        int queArr[50];
} QType;
```

(나) 연결 리스트를 이용한 큐 구조체

```
typedef struct _node {
        int data;
        struct _node *next;
} Node;
typedef struct _Queue {
        Node *front;
        Node *rear;
        int numOfData;
} QType;
```

<조건>

○ 큐 변수는 QType q; 형태로 선언한다.

○ Enqueue 함수는 큐의 front에 데이터를 추가하는 함수로 함수 원형은 void Enqueue(QType *q, int

NoteBase

value)이다.
◯ Dequeue 함수는 큐의 rear에서 데이터를 삭제하는 함수이며, 함수 원형은 void Dequeue(QType *q)이다.
◯ int형 자료의 경우 4바이트, 포인터형 자료의 경우 8바이트의 메모리 공간을 사용한다.
◯ Enqueue와 Dequeue 함수에서는 배열 및 연결 리스트 특성에 맞게 필요한 메모리를 할당·회수하는 동작이 적절히 일어난다.
◯ 연결 리스트에서 삭제 연산은 메모리 회수를 포함한다.
◯ 구조체 변수를 메모리에 할당할 때 바이트 패딩 없이 할당한다.

① (가): 24, (나): 44
② (가): 24, (나): 56
③ (가): 212, (나): 44
④ (가): 212, (나): 56

정답 3번

출제 포인트

이 문제는 배열 기반 큐와 연결 리스트 기반 큐에서 구조체 자체 크기와 연산 후 남아 있는 동적 메모리 크기를 구분해 계산하는 문제이다.

조건 정리

(가) `QType`은 `int front`, `int rear`, `int numOfData`, `int queArr[50]`로 이루어진다.
(나) 큐 구조체는 포인터 2개와 `int` 1개로 이루어지고, 각 노드는 `int data`와 포인터 1개로 이루어진다.
`int`는 4바이트, 포인터는 8바이트이며, 바이트 패딩은 없다고 본다.
Enqueue 3회 후 Dequeue 1회이므로 연결 리스트 큐에는 최종적으로 노드 2개가 남는다.

계산 과정

(가) 배열 큐의 구조체 크기는 $4 + 4 + 4 + 4 \times 50 = 12 + 200 = 212$바이트이다.
배열 기반 큐는 저장 배열 `queArr[50]`가 구조체 안에 이미 포함되어 있으므로 Enqueue와 Dequeue를 해도 추가 동적 할당이 없다.

(나) 연결 리스트 큐의 헤더 구조체 크기는 $8 + 8 + 4 = 20$바이트이다.
노드 1개의 크기는 $4 + 8 = 12$바이트이다.
Enqueue 3회 후 노드 3개는 $12 \times 3 = 36$바이트이고, Dequeue 1회에서 삭제 노드 1개를 회수하므로 남은 노드 2개는 $12 \times 2 = 24$바이트이다.
따라서 연결 리스트 큐 전체 크기는 $20 + 24 = 44$바이트이다.

NoteBase

선지별 해설

① X: `(가) 24`가 틀리다. 배열 큐는 현재 들어 있는 원소 수만큼만 메모리를 쓰는 것이 아니라 `queArr[50]` 전체를 포함한 `QType` 구조체 전체가 할당되므로 212바이트이다. `(나) 44`는 맞다. 올바르게는 `(가) 212`, `(나) 44`이다.
② X: `(가) 24`가 틀리고 `(나) 56`도 틀리다. `(가)`는 배열 전체 공간을 빼먹은 값이고, `(나) 56`은 삭제된 노드를 회수하지 않고 3개 노드가 계속 남는다고 본 값이다. 올바르게는 `(가) 212`, `(나) 44`이다.
③ O: `(가) 212`, `(나) 44`로 둘 다 맞다. 배열 큐는 고정 크기 구조체 전체를 잡고, 연결 리스트 큐는 헤더 20바이트에 남아 있는 노드 2개의 24바이트를 더해 44바이트가 된다.
④ X: `(가) 212`는 맞지만 `(나) 56`이 틀리다. 연결 리스트에서 Dequeue는 삭제 연산과 함께 메모리 회수를 포함하므로 노드 3개를 계속 잡고 있지 않는다. 올바르게는 `(나) 44`이다.

오답 포인트

배열 큐를 현재 저장된 원소 수만큼만 메모리를 차지한다고 착각하면 `(가)`를 24처럼 작게 계산하게 된다. 연결 리스트 큐는 Dequeue 뒤에 삭제 노드를 바로 회수한다는 조건을 놓치면 `(나)`를 56으로 계산하게 된다.

다시 나오면 이렇게 판단

메모리 계산 문제는 먼저 구조체 자체 크기와 동적 할당되는 부분을 분리한다. 배열 기반은 선언 순간 전체 배열 공간이 이미 확보되고, 연결 리스트 기반은 `헤더 구조체 + 현재 남아 있는 노드 수`만 계산하면 된다.

암기 포인트

배열 큐는 고정 크기 전체를 한 번에 잡고, 연결 리스트 큐는 `큐 헤더 + 살아 있는 노드 수`만큼만 잡는다.

NoteBase

문 10. 다음 C 프로그램의 실행 결과는? (단, 2차원 배열의 경우 행 우선으로 메모리에 저장된다)

```
#include <stdio.h>
int main(void)
{
        int i, j, test[3][4];
        for ( i = 0; i < 3; i++ )
                for ( j = 0; j < 4; j++ )
                        test[i][3-j] = i*4 + j;
        printf("%d ", *(*(test + 1) + 5));
}
```

① 5
② 6
③ 9
④ 10

정답 4번

출제 포인트

이 문제는 2차원 배열의 행 우선 저장과 포인터 산술이 결합될 때 `*(*(test+1)+5)`가 어느 원소를 가리키는지를 묻는 문제이다.

조건 정리

- 배열은 `test[3][4]`이다.
- 대입은 `test[i][3-j] = i*4 + j` 형태로 이루어진다.
- 2차원 배열은 행 우선으로 저장되므로 한 행의 길이는 4이다.
- 구해야 할 값은 `*(*(test+1)+5)`의 실제 정수값이다.

NoteBase

사용할 공식

행 우선 저장에서 `test[a][b]`의 선형 위치는 $a \times 4 + b$이다.
`test+1`은 두 번째 행을 가리키고, `*(test+1)`은 `test[1][0]`부터 시작하는 연속 메모리처럼 해석된다.
따라서 `+5`는 `test[1][0]`에서 다섯 칸 뒤 원소를 찾는 연산이다.

계산 과정

1. 반복문으로 각 행을 채우면 1행은 `[3, 2, 1, 0]`, 2행은 `[7, 6, 5, 4]`, 3행은 `[11, 10, 9, 8]`이 된다.
2. `test+1`은 두 번째 행 `test[1]`을 가리키므로 시작 원소는 `test[1][0] = 7`이다.
3. 여기서 다섯 칸 이동하면 순서대로 `7 -> 6 -> 5 -> 4 -> 11 -> 10`이므로 도착 원소는 `test[2][1] = 10`이다.
4. 따라서 출력 결과는 10이다.

선지별 해설

① X: 실제 출력값은 10이므로 5는 아니다. `출력 결과가 5이다`를 `출력 결과가 10이다`로 고쳐야 한다.
② X: `test[1][1]`이나 중간 이동 칸을 잘못 멈추면 6을 떠올릴 수 있으나, `+5`는 다섯 칸 뒤까지 이동하므로 6은 아니다. `출력 결과가 6이다`를 `출력 결과가 10이다`로 고쳐야 한다.
③ X: 행 우선 저장을 무시하고 열처럼 세거나 이동 칸 수를 하나 적게 세면 9를 고를 수 있으나, 실제 도착 원소는 `test[2][1]`이므로 9는 아니다. `출력 결과가 9이다`를 `출력 결과가 10이다`로 고쳐야 한다.
④ O: `test[1][0]`에서 다섯 칸 이동하면 `test[2][1]`에 도달하고 그 값이 10이므로 옳다.

오답 포인트

`test[i][3-j]` 때문에 각 행이 역순으로 채워진다는 점과, `*(test+1)`이 두 번째 행의 첫 원소부터 연속 메모리를 본다는 점을 동시에 놓치기 쉽다. 특히 `+5`를 다섯 칸 이동이 아니라 다섯 번째 값처럼 오해하면 6이나 9에서 멈추는 실수가 나온다.

다시 나오면 이렇게 판단

2차원 배열 포인터 문제는 먼저 각 행의 실제 저장값을 써 놓고, 그다음 시작 주소가 어느 원소인지 확인한 뒤 한 칸씩 세어 도착 원소를 찾으면 된다.

암기 포인트

행 우선 2차원 배열에서 `*(행 포인터 + k)`는 현재 행을 넘어 다음 행으로도 연속해서 이동한다.

NoteBase

문 11. 다음은 구조체 배열에서 최댓값을 찾아 출력하는 C 프로그램이다. (가), (나)에 들어갈 내용을 바르게 연결한 것은?

```
#include <stdio.h>
typedef struct _data {
      int value;
} DataType;
DataType *max(DataType *a) {
      int i, max = a[0].value, index = 0;
      for ( i = 0; i < 5; i++ )
            if ( max < a[i].value ) {
                  max = a[i].value;
                  index = i;
            }
      return [      (가)      ];
}

int main(void)
{
      int i;
      DataType data[5], *result;
      for ( i = 0; i < 5; i++ )
            scanf("%d", &data[i].value);

      result = max(data);
      printf("max = %d", [      (나)      ]);
      return 0;
}
```

① (가): &a[index], (나): result->value

② (가): &a[index], (나): result.value

③ (가): a[index], (나): result->value

④ (가): a[index], (나): (*result).value

정답 1번

출제 포인트

이 문제는 구조체 배열 원소의 주소를 반환하는 방법과 구조체 포인터의 멤버 접근 표기 차이를 묻는 문제이다.

핵심 개념

a[index]는 구조체 원소 그 자체인 DataType 값이다.
&a[index]는 그 원소의 주소인 DataType *이다.
result는 DataType *이므로 멤버 value에 접근할 때는 result->value 또는 (*result).value를 사용한다.

NoteBase

선지별 해설

1번: O. (가)의 &a[index]는 최대값을 가진 배열 원소의 주소를 반환하므로 함수의 반환형 DataType *과 일치한다. (나)의 result->value는 포인터 result가 가리키는 구조체의 value 멤버를 출력하는 올바른 표현이다.
2번: X. (가)의 &a[index]는 맞지만 (나)의 result.value가 틀리다. result는 구조체 변수가 아니라 구조체 포인터이므로 . 연산자를 쓸 수 없다. 올바르게는 result->value 또는 (*result).value이다.
3번: X. (가)의 a[index]가 틀리다. a[index]는 구조체 값 자체이므로 반환형 DataType *에 맞지 않는다. 올바르게는 최대값 원소의 주소를 돌려주는 &a[index]이다. (나)의 result->value 자체는 포인터 멤버 접근으로 맞다.
4번: X. (나)의 (*result).value는 포인터가 가리키는 구조체 멤버 접근으로 맞는 표현이지만 (가)의 a[index]가 틀리다. 이 함수는 구조체 값이 아니라 구조체 포인터를 반환해야 하므로 올바르게는 &a[index]이다.

오답 포인트

a[index]와 &a[index]를 같은 것으로 보면 반환형 불일치를 놓치게 된다. 또 result가 포인터라는 사실을 잊으면 2번처럼 . 연산자를 쓰는 실수를 하게 된다.

다시 나오면 이렇게 판단

함수 반환형이 포인터인지 값인지 먼저 본다. 반환형이 DataType *이면 식도 주소여야 한다. 그다음 변수가 포인터이면 ->, 실제 구조체 값이면 . 연산자를 쓴다고 판단하면 된다.

암기 포인트

값이면 ., 주소면 ->, 포인터 반환이면 &를 붙인다고 정리하면 된다.

NoteBase

문 12. 다음 전위 표기식을 후위 표기식으로 변환한 것은?

/ + a b - c d

① a b + c d - /
② a b c d + - /
③ a b / c d + -
④ a b c d / + -

정답 1번

출제 포인트

이 문제는 전위 표기식의 연산자-피연산자 결합 구조를 따라 후위 표기식으로 바꾸는 순서를 묻는 문제이다.

올바른 순서

1. 루트 연산자 `/`를 기준으로 식을 좌우 부분식으로 나눈다.
2. 왼쪽 부분식 `+ a b`를 후위식으로 바꾸면 `a b +`가 된다.
3. 오른쪽 부분식 `- c d`를 후위식으로 바꾸면 `c d -`가 된다.
4. 후위 표기식은 왼쪽 부분식, 오른쪽 부분식, 루트 연산자 순서로 붙이므로 `a b + c d - /`가 된다.

왜 이 순서: 후위 표기식은 각 부분식이 먼저 완성된 뒤 그 결과를 마지막 연산자가 받는 구조이므로, `/`는 두 부분식 `a b +`와 `c d -`가 모두 나온 뒤 맨 끝에 와야 한다.

선지별 해설

① O: `+ a b`를 `a b +`로, `- c d`를 `c d -`로 바꾼 뒤 마지막에 `/`를 붙인 형태이므로 정확하다.
② X: `c d +`가 되어 오른쪽 부분식의 연산자가 `-`에서 `+`로 바뀌고, 왼쪽 부분식 `+ a b`도 `a b +`로 정리되지 않았다. 올바르게는 `a b + c d - /`이다.
③ X: 맨 앞의 `/`가 너무 일찍 적용되어 `a b /`가 먼저 계산되는 형태가 되므로 원래 식의 루트 구조와 다르다. 올바르게는 두 부분식 `a b +`, `c d -`를 만든 뒤 `/`를 마지막에 둔다.
④ X: `c d /`가 먼저 만들어져 원래 오른쪽 부분식 `- c d`가 완전히 깨지고, 최종 연산자도 `/`가 아니라 `-`로 끝난다. 올바르게는 `a b + c d - /`이다.

오답 포인트

전위식을 후위식으로 바꿀 때 기호의 나열만 보고 앞에서부터 기계적으로 옮기면 루트 연산자와 부분식 경계가 무너진다. 이 문항에서는 `/`가 전체 식의 마지막에 와야 한다는 점과, 오른쪽 부분식이 `- c d`라는 점을 놓치면 ②, ③, ④처럼 다른 식이 된다.

NoteBase

다시 나오면 이렇게 판단

전위 표기식 변환 문제는 루트 연산자를 먼저 찾고, 그 아래 왼쪽 부분식과 오른쪽 부분식을 각각 독립적으로 후위식으로 만든 다음 `왼쪽-오른쪽-루트` 순서로 합치면 된다.

암기 포인트

전위는 루트가 먼저, 후위는 루트가 마지막이다.

NoteBase

문 13. 다음은 단순 연결 리스트를 이용하여 스택 자료구조를 구현한 파이썬 코드이다. (가), (나)에 들어갈 내용을 바르게 연결한 것은?

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next

class MyStack:
    def __init__(self):
        self.top = None

    def push(self, data):
        (가)
        self.top = n

    def pop(self):
        if self.top != None:
            n = self.top
            (나)
            return n.data
```

① (가) n = Node(data, None), (나) self.top = n.next.next

② (가) n = Node(data, None), (나) self.top = self.top.next

③ (가) n = Node(data, self.top), (나) self.top = n.next.next

④ (가) n = Node(data, self.top), (나) self.top = self.top.next

정답 4번

출제 포인트

이 문제는 단순 연결 리스트로 스택의 push와 pop을 구현할 때 새 노드를 현재 top 앞에 연결하고, pop 후 top을 다음 노드로 이동시키는 원리를 적용할 수 있는지를 묻는다.

사례 정리

Node는 data와 next를 가진 단순 연결 리스트 노드이다.
stack의 top은 가장 최근에 들어간 노드를 가리킨다.
push는 새 원소를 top 앞에 붙여야 하고, pop은 현재 top을 제거한 뒤 다음 노드를 새 top으로 만들어야 한다.

NoteBase

핵심 개념

연결 리스트 기반 스택은 삽입과 삭제를 리스트의 머리에서 수행하는 구조이다.
push의 요건은 새 노드의 next가 기존 top을 가리키는 것이고, pop의 요건은 기존 top을 임시 저장한 뒤 top을 그 다음 노드로 옮기는 것이다.

적용 기준

이 코드에서는 top이 리스트의 머리이므로 (가)에서는 기존 top을 새 노드의 next로 연결해야 한다.
또한 pop에서 n=self.top으로 현재 머리를 저장했으므로 (나)에서는 self.top을 self.top.next로 갱신하면 된다.

선지별 해설

1번: X. (가)의 "n=Node(data, None)"가 틀렸다. 이렇게 하면 새 노드가 기존 스택과 끊겨 push 할 때 이전 원소들이 연결되지 않는다. 올바르게는 "n=Node(data, self.top)"이어야 한다. 또한 (나)의 "self.top=n.next.next"도 틀렸다. pop 뒤에는 한 칸만 내려가야 하므로 올바르게는 "self.top=self.top.next" 또는 "self.top=n.next"이다.
2번: X. (나)의 "self.top=self.top.next"는 맞지만, (가)의 "n=Node(data, None)"가 틀렸다. 새 top이 이전 top을 가리켜야 스택의 연결이 유지된다. 올바르게는 "n=Node(data, self.top)"이다.
3번: X. (가)의 "n=Node(data, self.top)"는 맞지만, (나)의 "self.top=n.next.next"가 틀렸다. 현재 top을 제거한 뒤에는 다음 노드 하나만 새 top이 된다. 두 칸 이동하면 중간 노드까지 건너뛰어 스택 구조가 깨진다. 올바르게는 "self.top=self.top.next" 또는 "self.top=n.next"이다.
4번: O. (가)의 "n=Node(data, self.top)"는 새 노드가 기존 top을 가리키게 하므로 push 구현에 맞다. 이어서 self.top=n으로 새 노드를 머리로 만든다. (나)의 "self.top=self.top.next"는 현재 top을 제거한 뒤 다음 노드를 새 top으로 두는 처리이므로 pop 구현에도 맞다.

오답 포인트

push에서 새 노드의 next를 None으로 두면 연결 리스트 기반 스택이 아니라 원소 하나짜리 노드만 계속 top에 올리는 꼴이 된다. pop에서 next.next로 두 칸 이동하는 실수도 자주 나오는데, 스택의 삭제는 머리 노드 하나만 제거하는 연산이라는 점을 놓치면 틀린다.

다시 나오면 이렇게 판단

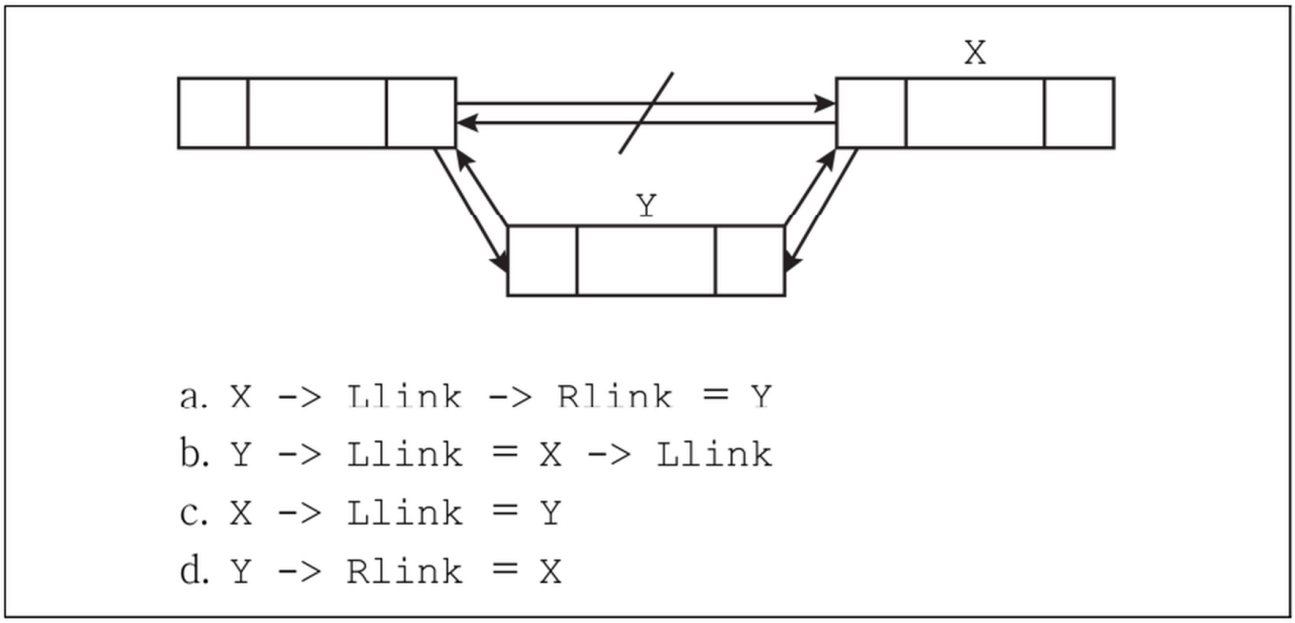
연결 리스트 스택 문제는 top이 어디를 가리키는지만 먼저 본다. push는 "새 노드 next=기존 top, 그다음 top=새 노드" 순서이고, pop은 "현재 top 저장, 그다음 top=기존 top의 next" 순서인지 확인하면 된다.

암기 포인트

연결 리스트 스택은 넣을 때 머리 앞에 붙고, 뺄 때 머리 한 칸만 내려간다.

NoteBase

문 14. 그림과 같은 이중 연결 리스트(doubly linked list)에서 노드 X의 왼쪽에 노드 Y를 삽입할 때, 연산의 순서를 바르게 나열한 것은? (단, Llink는 왼쪽 노드를 가리키는 포인터 변수이고, Rlink는 오른쪽 노드를 가리키는 포인터 변수이다)



① a → c → b → d

② a → c → d → b

③ b → c → a → d

④ b → d → a → c

정답 4번

출제 포인트

이 문제는 이중 연결 리스트에서 새 노드를 기존 두 노드 사이에 삽입할 때 포인터를 잃지 않도록 갱신하는 순서를 묻는 문제이다.

올바른 순서

1. b — Y->Llink를 기존 왼쪽 노드로 연결한다.
2. d — Y->Rlink를 X로 연결한다.
3. a — 기존 왼쪽 노드의 Rlink를 Y로 바꾼다.
4. c — 마지막으로 X의 Llink를 Y로 바꾼다.

왜 이 순서: c를 먼저 수행하면 X->Llink가 곧바로 Y를 가리켜 기존 왼쪽 노드에 접근할 수 없게 되므로, a는 반드시 c보다 먼저 수행되어야 한다.

NoteBase

선지별 해설

① X
- a를 먼저 수행하면 기존 왼쪽 노드의 Rlink는 Y로 바뀌지만, 아직 Y의 양쪽 링크가 정리되지 않아 삽입 상태가 완성되지 않는다.
- 틀린 표현: a가 c보다 앞선 채 시작하면 된다는 판단이다.
- 올바르게는: 기존 노드를 잃지 않도록 b와 d로 Y의 링크를 먼저 정리한 뒤 a와 c를 처리해야 한다.

② X
- a와 c의 상대적 순서는 맞지만, a를 수행하기 전에 Y의 오른쪽 링크를 먼저 X에 연결해 두는 편이 삽입 구조를 완전하게 만든다.
- 틀린 표현: a 직전에 d가 없어도 된다는 판단이다.
- 올바르게는: b → d → a → c 순서로 두 방향 링크를 먼저 확보한 뒤 주변 노드를 연결한다.

③ X
- c를 a보다 먼저 수행하면 X->Llink가 이미 Y로 바뀌므로 a의 X->Llink->Rlink는 기존 왼쪽 노드가 아니라 Y의 Rlink를 건드리게 된다.
- 틀린 표현: b 다음에 c를 둘 수 있다는 판단이다.
- 올바르게는: 기존 왼쪽 노드를 이용해야 하는 a를 먼저 수행하고, c는 마지막에 둔다.

④ O
- b로 Y의 왼쪽을, d로 Y의 오른쪽을 먼저 정한 뒤 a로 기존 왼쪽 노드의 Rlink를 Y에 연결하고, 마지막 c로 X의 Llink를 Y로 바꾸면 양방향 연결이 모두 올바르게 완성된다.

오답 포인트

가장 흔한 함정은 c를 너무 일찍 수행하는 경우이다. X->Llink는 기존 왼쪽 노드를 찾기 위한 기준 포인터인데 이를 먼저 Y로 바꾸면 a에서 수정해야 할 원래 노드에 더 이상 접근하지 못하게 된다.

다시 나오면 이렇게 판단

이중 연결 리스트 삽입 문제는 기존 포인터를 잃기 전에 새 노드의 링크를 먼저 채우고, 기존 노드를 따라가야 하는 갱신을 끝낸 뒤 기준 포인터를 마지막에 바꾸는 순서로 판단한다.

암기 포인트

새 노드 연결이 먼저이고 기준 포인터 교체가 마지막이다.

NoteBase

문 15. 최소 1개 이상의 노드로 이루어진 원형 연결 리스트(circular linked list)에서 포인터 L이 리스트의 첫 번째 노드를 가리킨다고 할 때, 데이터의 값이 x인 노드를 탐색하여 반환하는 C 코드는?

①
```
ListNode *searchCList(ListNode *L, int x) {
        ListNode *p=L;
        while(p!=L) {
                if(p->data==x) return p;
                p=p->link;
        }
        return NULL;
}
```

②
```
ListNode *searchCList(ListNode *L, int x) {
        ListNode *p=L;
        while(p->link!=L) {
                if(p->data==x) return p;
                p=p->link;
        }
        return NULL;
}
```

③
```
ListNode *searchCList(ListNode *L, int x) {
        ListNode *p=L;
        do {
                if(p->data==x) return p;
                p=p->link;
        } while(p!=L);
        return NULL;
}
```

④
```
ListNode *searchCList(ListNode *L, int x) {
        ListNode *p=L;
        do {
                if(p->data==x) return p;
                p=p->link;
        } while(p->link!=L);
        return NULL;
}
```

정답 3번

출제 포인트

이 문제는 원형 연결 리스트 탐색에서 첫 노드도 반드시 검사하면서 한 바퀴를 정확히 한 번만 도는 반복 조건을 고를 수 있는지를 묻는다.

핵심 개념

원형 연결 리스트는 마지막 노드의 link가 다시 첫 노드를 가리키는 구조이다. 따라서 시작 노드 *L*을 기준으로 첫 노드를 먼저 검사한 뒤, 포인터가 다시 *L*로 돌아올 때까지 반복해야 전체 노드를 한 번씩 검사할 수 있다.

NoteBase

선지별 해설

1번: X. `p`를 처음부터 `L`로 두고 `while (p != L)`를 검사하므로 반복문에 한 번도 들어가지 않는다. 틀린 표현은 `while (p != L)`이다. 실제로는 첫 노드도 검사할 수 있도록 `do ... while` 구조를 쓰거나, 첫 반복 전에 별도 검사가 있어야 한다.
2번: X. `while (p->link != L)`는 현재 노드의 다음 노드가 첫 노드가 아닐 때만 반복하므로 마지막 노드는 검사하지 못한다. 노드가 1개뿐인 경우에도 반복문에 들어가지 못한다. 틀린 표현은 `while (p->link != L)`이다. 실제로는 현재 노드가 다시 `L`이 될 때까지 순회해야 한다.
3번: O. `do ... while (p != L)`는 첫 노드를 먼저 검사하고, 이후 다음 노드로 이동한 뒤 시작점으로 돌아오기 전까지 계속 반복한다. 따라서 노드가 1개 이상인 원형 연결 리스트의 모든 노드를 정확히 한 번씩 검사할 수 있다.
4번: X. `do ... while`를 써서 첫 노드는 검사하지만, 반복 조건을 `p->link != L`로 두어 마지막 노드에 도달하면 그 마지막 노드를 검사하지 못하고 종료한다. 틀린 표현은 `while (p->link != L)`이다. 실제로는 마지막 노드까지 검사한 뒤 포인터가 다시 `L`이 되었는지로 종료를 판단해야 한다.

오답 포인트

출제자가 노린 함정은 `p != L`과 `p->link != L`의 차이, 그리고 `while`과 `do ... while`의 첫 검사 시점 차이이다. 2번과 4번은 얼핏 한 바퀴를 도는 것처럼 보이지만 마지막 노드 검사가 빠진다.

다시 나오면 이렇게 판단

원형 연결 리스트 순회 코드를 보면 먼저 첫 노드를 검사하는지, 그리고 종료 조건이 `다음 노드가 첫 노드인가`가 아니라 `현재 포인터가 다시 시작점으로 돌아왔는가`인지 확인하면 된다.

암기 포인트

원형 리스트 전체 탐색은 `첫 노드 검사 후 다시 시작점으로 돌아올 때 종료`가 핵심이다.

NoteBase

문 16. 0이 아닌 원소의 개수가 7개인 9×12 희소행렬에 대한 설명으로 옳지 않은 것은?

① 행렬을 그대로 선형 순차 리스트로 표현하여 2차원 배열로 구현하면, 108개의 배열 원소에 대한 공간이 필요하다.

② 0이 아닌 원소들에 대한 〈행 번호, 열 번호, 값〉의 쌍을 2차원 배열로 구현하면 14개의 배열 원소에 대한 공간이 필요하다.

③ 행렬을 그대로 선형 순차 리스트로 표현하여 2차원 배열로 구현하는 것은 실제로 사용하지 않는 공간이 많아 기억 공간의 활용도가 떨어진다.

④ 〈행 번호, 열 번호, 값〉의 쌍을 2차원 배열로 구현한 희소행렬에 대하여 전치 행렬을 구하는 방법의 시간 복잡도는 최악의 경우, O(열의 수^2 × 행의 수)이다.

정답 2번

출제 포인트

희소행렬의 일반 2차원 배열 표현과 3원소 표현에서 필요한 저장 공간, 그리고 전치 연산의 시간 복잡도를 비교하는 문제이다.

핵심 개념

희소행렬을 그대로 2차원 배열에 저장하면 전체 칸 수만큼 공간이 필요하므로 이 문제에서는 $9 \times 12 = 108$칸이 필요하다.
3원소 표현은 0이 아닌 원소마다 행 번호, 열 번호, 값의 3개 정보를 저장하므로 0이 아닌 원소가 7개이면 기본적으로 $7 \times 3 = 21$칸이 필요하고, 헤더까지 두는 구현이라면 $(7 + 1) \times 3 = 24$칸이 필요하다.

NoteBase

선지별 해설

① O
행렬을 일반 2차원 배열로 구현하면 0의 개수와 무관하게 전체 크기만큼 저장해야 하므로 $9 \times 12 = 108$개의 배열 원소가 필요하다.

② X
틀린 부분은 "14개의 배열 원소"이다.
0이 아닌 원소 7개를 <행 번호, 열 번호, 값>으로 저장하면 원소 하나당 3칸이 필요하므로 최소 21칸이 필요하다.
헤더 행까지 포함하는 교과서식 배열이면 24칸이 된다.
올바르게는 "3원소 표현에는 21개의 배열 원소가 필요하며, 헤더를 포함하면 24개가 필요하다"이다.

③ O
희소행렬을 일반 2차원 배열로 저장하면 대부분의 칸이 0인데도 공간을 차지하므로 실제로 사용하지 않는 공간이 많아 기억공간 활용도가 떨어진다.

④ O
3원소 표현으로 저장한 희소행렬의 단순 전치 방법은 각 열에 대해 모든 0이 아닌 원소를 검사하므로 시간 복잡도는 O(열의 수×0이 아닌 원소 수)이다. 최악의 경우 0이 아닌 원소 수를 행의 수×열의 수로 보면 O(열의 수^2×행의 수)가 된다.

오답 포인트

3원소 표현은 값만 저장하는 방식이 아니라 행 번호와 열 번호까지 함께 저장한다는 점을 놓치기 쉽다. ②는 0이 아닌 원소의 개수 7만 보고 공간을 과소계산하게 만드는 선지이다.

다시 나오면 이렇게 판단

희소행렬 저장 방식 문제는 먼저 전체 배열 저장인지, 0이 아닌 원소만 저장하는지부터 구분한다. 0이 아닌 원소만 저장하면 원소 하나당 몇 개의 정보가 필요한지를 곱해 공간을 계산하면 된다.

암기 포인트

일반 배열은 $m \times n$칸을 그대로 쓰고, 3원소 표현은 0이 아닌 원소 하나당 행·열·값의 3칸을 쓴다.

NoteBase

문 17. 다음 C 코드에서 heapify(a, 7)을 수행한 후 배열 a[]의 내용은? (단, 모든 배열 원소값은 양의 정수로 한정하고, -1은 연산 대상이 아니며, 0은 삭제된 값을 나타낸다)

```
int a[8] = {-1, 1, 2, 3, 6, 4, 7, 5};

int heapify(int *a, int n) {
      int item = a[1], temp = a[n];
      int i = 1, j = 2;

      a[n] = 0,  n = n-1;
      while (j <= n) {
            if (j < n && a[j] > a[j+1]) j++;
            if (temp < a[j]) break;
            else {
                  a[i] = a[j];
                  i = j;
                  j = j*2;
            }
      }
      a[i] = temp;
      return item;
}
```

①

0	1	2	3	4	5	6	7
−1	2	3	4	6	5	7	0

②

0	1	2	3	4	5	6	7
−1	2	4	3	6	5	7	0

③

0	1	2	3	4	5	6	7
−1	2	3	4	5	6	7	0

④

0	1	2	3	4	5	6	7
−1	2	4	3	5	6	7	0

정답 2번

출제 포인트

최소 힙에서 루트를 삭제한 뒤 마지막 원소를 아래로 내려 보내며 더 작은 자식과 교환하는 재구성 절차를 묻는 문제이다.

올바른 순서

1. item=a[1]=1을 삭제값으로 저장하고 temp=a[7]=5를 꺼낸다.
2. a[7]=0으로 삭제 표시를 하고, 루트 위치에서는 자식 2와 3 중 더 작은 2를 올려 a[1]=2가 된다.
3. i=2로 내려가면 자식 6과 4 중 더 작은 4를 올려 a[2]=4가 된다.
4. 더 내려갈 자식이 없으므로 temp=5를 a[5]에 넣어 최종 배열이 -1, 2, 4, 3, 6, 5, 7, 0이 된다.

왜 이 순서: 각 단계에서 두 자식 중 더 작은 값을 먼저 끌어올려야 최소 힙 조건이 유지된다.

NoteBase

선지별 해설

① X: 루트에 2가 올라오는 것까지는 맞지만, 다음 단계에서 i=2의 자식 6과 4 중 더 작은 4가 올라와야 한다. 틀린 표현은 배열 인덱스 2가 3이고 인덱스 3이 4인 배치이다. 올바르게는 인덱스 2가 4이고 인덱스 3이 3이다.
② O: temp=5를 들고 내려가면 먼저 2가 루트로, 다음에는 4가 인덱스 2로 올라오고 5는 인덱스 5에 놓인다. 따라서 최종 배열은 -1, 2, 4, 3, 6, 5, 7, 0이 된다.
③ X: 힙 재구성은 전체 오름차순 정렬이 아니라 temp를 적절한 자리까지 내려 최소 힙 성질만 회복하는 과정이다. 틀린 표현은 인덱스 2가 3, 인덱스 3이 4, 인덱스 4가 5로 연속 정렬된 배치이다. 올바르게는 3은 인덱스 3에 그대로 남고 5는 인덱스 5에 들어간다.
④ X: 2와 4를 위로 올리는 흐름은 맞지만 temp=5가 들어갈 위치를 한 칸 위로 잘못 놓았다. 틀린 표현은 인덱스 4가 5이고 인덱스 5가 6인 부분이다. 올바르게는 인덱스 4가 6이고 인덱스 5가 5이다.

오답 포인트

마지막 원소 5를 넣은 뒤 배열이 거의 정렬된 모습이 된다고 착각하면 ③처럼 답하기 쉽다. 이 함수는 정렬이 아니라 삭제 후 최소 힙 성질을 회복하는 절차이므로, 각 단계에서 더 작은 자식만 위로 올린다는 점이 핵심이다.

다시 나오면 이렇게 판단

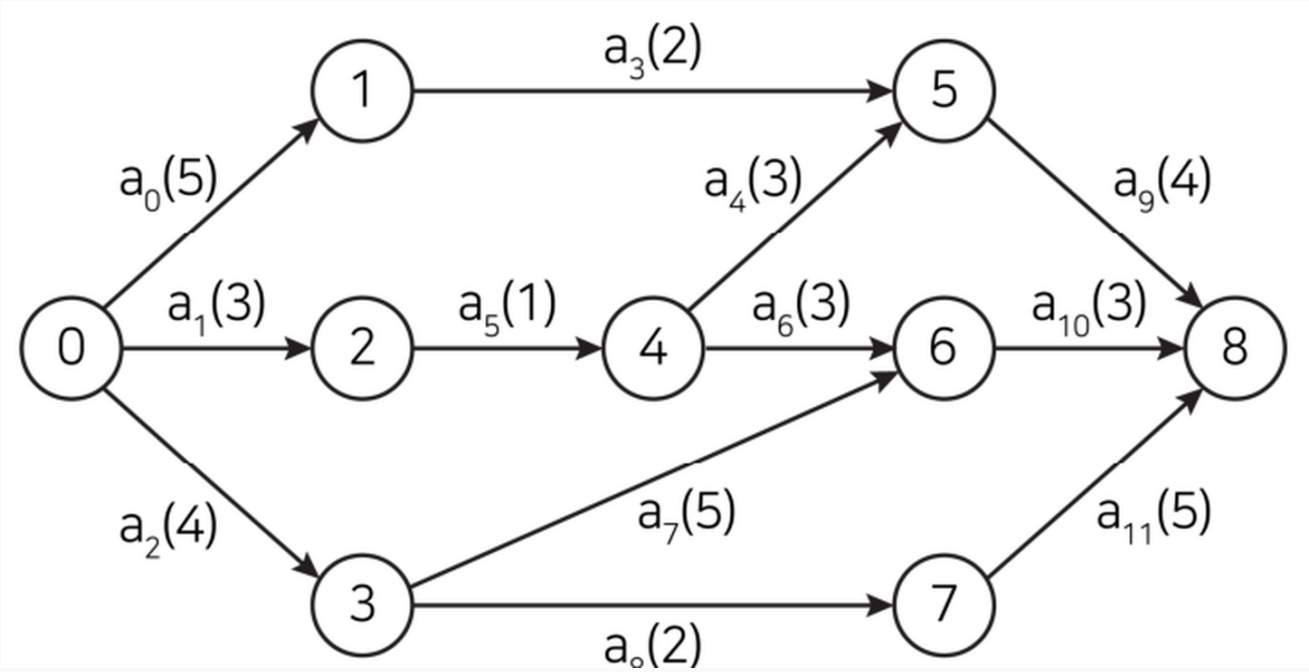
힙 삭제 문제는 마지막 원소를 temp로 잡고, 현재 위치의 두 자식 중 더 작은 값을 위로 올리면서 temp가 멈출 자리를 찾는 순서로 판단한다. 전체 배열을 정렬하려고 보면 오답으로 흐르기 쉽다.

암기 포인트

최소 힙 삭제는 마지막 원소 하향 이동, 작은 자식 상향 이동이라는 두 줄로 정리된다.

NoteBase

문 18. 다음 그래프는 AOE(Activity on Edge) 네트워크를 보여 준다. 정점의 숫자는 공정을 나타내고, 간선 위에 있는 a_i는 작업을, 괄호 안의 숫자는 그 작업을 수행하는 데 필요한 최소한의 시간을 나타낼 때, 임계 경로의 길이는?



① 10

② 11

③ 12

④ 13

정답 3번

출제 포인트

AOE 네트워크에서 각 정점의 최조 발생 시간을 구한 뒤 시작점에서 종점까지의 최장 경로 길이, 즉 임계 경로 길이를 계산하는 문제이다.

조건 정리

작업 시간은 a0=5, a1=3, a2=4, a3=2, a4=3, a5=1, a6=3, a7=5, a8=2, a9=4, a10=3, a11=5이다.
구해야 할 값은 정점 0에서 정점 8까지의 임계 경로 길이이다.
AOE 네트워크의 임계 경로는 가능한 경로들 중 총 소요 시간이 가장 긴 경로이다.

NoteBase

사용할 공식

정점의 최조 발생 시간은 $ve(j) = \max(ve(i) + w_{ij})$로 구한다. 임계 경로 길이는 종점의 최조 발생 시간 $ve(8)$과 같다.
이 공식을 쓰는 이유는 합류 정점에서 모든 선행 작업이 끝나야 다음 작업이 시작되므로, 가장 늦게 도착하는 경로가 전체 공정 시간을 결정하기 때문이다.

계산 과정

1. $ve(0) = 0$, $ve(1) = 5$, $ve(2) = 3$, $ve(3) = 4$이다.
2. $ve(4) = 3 + 1 = 4$, $ve(5) = \max(5 + 2, 4 + 3) = 7$, $ve(6) = \max(4 + 3, 4 + 5) = 9$, $ve(7) = 4 + 2 = 6$이다.
3. $ve(8) = \max(7 + 4, 9 + 3, 6 + 5) = \max(11, 12, 11) = 12$이다. 따라서 임계 경로는 0→3→6→8이고 길이는 12이다.

선지별 해설

① X: 10은 경로 0→2→4→6→8의 길이 3 + 1 + 3 + 3만 더한 값이다. 그러나 이 경로는 최장 경로가 아니다. 틀린 값 10 → 올바르게는 12이다.
② X: 11은 0→1→5→8, 0→2→4→5→8, 0→3→7→8 중 하나의 경로 길이이다. 가능한 경로 중 하나일 뿐 최장 경로는 아니다. 틀린 값 11 → 올바르게는 12이다.
③ O: 12는 경로 0→3→6→8의 길이 4 + 5 + 3이다. 종점 8의 최조 발생 시간도 12이므로 임계 경로 길이와 일치한다.
④ X: 13은 서로 다른 경로의 시간을 한 경로처럼 중복 합산할 때 나오는 값이다. AOE 네트워크의 임계 경로는 실제로 연결된 한 경로의 최장 길이여야 하므로 13은 성립하지 않는다. 틀린 값 13 → 올바르게는 12이다.

오답 포인트

합류 정점에서 최대값을 취하지 않고 보이는 경로 하나만 더하면 10이나 11에 멈춘다. 반대로 서로 다른 가지를 섞어 더하면 13처럼 실제로 존재하지 않는 경로 길이를 만들게 된다.

다시 나오면 이렇게 판단

AOE 네트워크가 나오면 시작 정점부터 각 정점의 최조 발생 시간을 순서대로 채우고, 합류 정점에서는 반드시 최대값을 취한다. 마지막 종점의 최조 발생 시간이 곧 임계 경로 길이이다.

암기 포인트

임계 경로 길이 = 시작점에서 종점까지의 최장 경로 길이 = 종점의 ve 값이다.

NoteBase

문 19. 키 값 <2, 6, 3, 4, 5, 1, 7, 8>을 순서대로 삽입해서 레드 블랙 트리를 생성할 때, 단일 회전과 이중 회전이 발생하는 횟수를 바르게 나열한 것은?

① 단일 회전: 1, 이중 회전: 1

② 단일 회전: 1, 이중 회전: 2

③ 단일 회전: 2, 이중 회전: 1

④ 단일 회전: 2, 이중 회전: 2

정답 2번

출제 포인트

레드블랙트리에 키를 순서대로 삽입할 때 어느 삽입에서 단일 회전과 이중 회전이 발생하는지를 추적하는 문제이다.

조건 정리

삽입 순서는 2, 6, 3, 4, 5, 1, 7, 8이다.
레드블랙트리 삽입에서는 부모와 자식이 모두 빨강이 되면 삼촌의 색을 먼저 확인한다.
삼촌이 검정이고 꺾인 형태이면 이중 회전, 일직선 형태이면 단일 회전이 발생한다.

계산 과정

3을 삽입하면 2-6-3이 오른쪽-왼쪽 형태가 되어 RL형 이중 회전이 1회 발생한다.
4를 삽입할 때는 삼촌이 빨강이어서 재색칠만 하고 회전은 없다.
5를 삽입하면 4-6-5가 왼쪽-오른쪽 형태가 되어 LR형 이중 회전이 1회 발생한다.
1과 7을 삽입할 때는 재색칠만 일어나고 회전은 없다.
8을 삽입하면 6-7-8이 오른쪽-오른쪽 형태가 되어 단일 좌회전이 1회 발생한다.
따라서 총합은 단일 회전 1회, 이중 회전 2회이다.

NoteBase

선지별 해설

① X
'단일 1회, 이중 1회'가 틀렸다. 3 삽입 때 한 번, 5 삽입 때 한 번의 이중 회전이 발생하므로 이중 회전은 2회이다. 올바르게는 '단일 1회, 이중 2회'이다.

② O
3 삽입에서 RL형 이중 회전 1회, 5 삽입에서 LR형 이중 회전 1회, 8 삽입에서 RR형 단일 회전 1회가 발생한다. 따라서 '단일 1회, 이중 2회'가 맞다.

③ X
'단일 2회, 이중 1회'가 틀렸다. 8 삽입에서만 일직선 형태에 따른 단일 회전이 일어나므로 단일 회전은 1회이다. 또한 3 삽입과 5 삽입에서 각각 이중 회전이 있으므로 올바르게는 '단일 1회, 이중 2회'이다.

④ X
'단일 2회, 이중 2회'가 틀렸다. 이중 회전 2회까지는 맞지만 단일 회전은 8 삽입에서 한 번뿐이다. 올바르게는 '단일 1회, 이중 2회'이다.

오답 포인트

이 문제의 함정은 재색칠만 일어나는 경우를 회전으로 세거나, 이중 회전을 구성하는 두 번의 단일 회전을 따로 세는 데 있다. 4, 1, 7 삽입은 재색칠만 있고, 3과 5 삽입은 각각 하나의 이중 회전으로 묶어서 세어야 한다.

다시 나오면 이렇게 판단

레드블랙트리 회전 횟수 문제는 각 삽입마다 부모-삼촌-조부의 색과 형태를 따로 기록하면 된다. 삼촌이 빨강이면 재색칠, 삼촌이 검정이면서 꺾인 형태이면 이중 회전 1회, 일직선 형태이면 단일 회전 1회로 분류하면 된다.

암기 포인트

레드블랙트리 삽입은 '빨강 삼촌이면 재색칠, 검정 삼촌이면 꺾임은 이중 회전, 일직선은 단일 회전'으로 정리된다.

NoteBase

문 20. 다음 이진 트리의 전체 노드 수를 계산하는 C 코드에서 (가)에 들어갈 내용은? (단, n은 최초에 이진 트리 내 루트 노드를 가리키는 포인터 변수이며, TNode는 이진 트리의 노드 구조를 나타낸다)

```
typedef struct TreeNode {
        char data;
        struct TreeNode *left;
        struct TreeNode *right;
} TNode;

int count_node(TNode *n) {
        if (n == NULL) return 0;
        return (가) + count_node(n->right);
}
```

① 1

② n->data

③ count_node(n->left)

④ 1 + count_node(n->left)

정답 4번

출제 포인트

이 문제는 이진 트리의 전체 노드 수를 세는 재귀 함수에서 현재 노드 처리와 왼쪽·오른쪽 서브트리 재귀 호출을 어떻게 결합하는지를 묻는 문제이다.

핵심 개념

전체 노드 수 계산은 현재 노드 1개를 센 뒤 왼쪽 서브트리의 노드 수와 오른쪽 서브트리의 노드 수를 더하는 방식이다.
즉 노드 포인터가 NULL이면 0을 반환하고, NULL이 아니면
1 + left subtree count + right subtree count 구조가 된다.

NoteBase

선지별 해설

① X: (가)에 1만 넣으면 최종 반환식이 $1 + count_node(n->right)$가 되어 현재 노드와 오른쪽 서브트리만 센다. 틀린 표현은 왼쪽 서브트리 계산이 빠진 1이다. 올바르게는 1 + count_node(n->left)이어야 한다.
② X: n->data는 노드에 저장된 문자 데이터일 뿐 노드 수가 아니다. 틀린 표현은 n->data이다. 올바르게는 현재 노드 1개를 뜻하는 1과 왼쪽 서브트리 계산을 더한 1 + count_node(n->left)이어야 한다.
③ X: count_node(n->left)를 넣으면 최종 반환식이 $count_node(n->left) + count_node(n->right)$가 되어 현재 노드 자신 1개를 세지 못한다. 틀린 표현은 현재 노드 개수 1이 빠진 count_node(n->left)이다. 올바르게는 1 + count_node(n->left)이어야 한다.
④ O: 1 + count_node(n->left)를 넣으면 최종 반환식이 $1 + count_node(n->left) + count_node(n->right)$가 된다. 현재 노드 1개와 왼쪽·오른쪽 서브트리의 모든 노드를 합산하므로 전체 노드 수를 정확히 계산한다.

오답 포인트

재귀 함수에서 현재 노드 처리와 서브트리 처리를 분리해서 보지 않으면 ①과 ③처럼 한 항이 빠진 식을 고르기 쉽다. 특히 코드에 이미 + count_node(n->right)가 적혀 있으므로 (가)에는 왼쪽 서브트리와 현재 노드 1개가 함께 들어가야 한다는 점이 핵심 함정이다.

다시 나오면 이렇게 판단

트리 재귀 문제는 반환식에서 현재 노드, 왼쪽 서브트리, 오른쪽 서브트리의 세 요소가 모두 한 번씩 반영되는지부터 확인하면 된다. NULL일 때 0을 반환하는지 본 뒤, NULL이 아닐 때 현재 노드 1개를 세는 항이 빠졌는지를 점검하면 정답을 빠르게 가릴 수 있다.

암기 포인트

트리 노드 수 재귀는 현재 1개 + 왼쪽 재귀 + 오른쪽 재귀 구조이다.

NoteBase

문 21. 다음 탐색 알고리즘을 이용하여 아래 리스트에서 키(key) 34를 찾을 경우, 필요한 반복 연산 횟수는? (단, low 변수에는 0, high 변수에는 15가 전달된다)

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	6	11	13	18	20	22	27	29	30	34	38	41	42	45	47

① 3

② 4

③ 5

④ 6

정답 2번

출제 포인트

이 문제는 이진 탐색에서 low, high, middle의 변화를 따라가며 키를 찾을 때 while 반복이 몇 번 실행되는지를 계산하는 문제이다.

조건 정리

정렬된 리스트의 인덱스 범위는 0부터 15까지이다.
찾을 키는 34이다.
반복마다 middle은 (*low* + *high*)/2의 정수 몫으로 계산된다.

NoteBase

사용할 공식

반복마다 $middle = (low + high)/2$를 계산하고, key와 list[middle]를 비교한 뒤 key가 크면 low를 $middle + 1$로, 작으면 high를 $middle - 1$로 바꾼다.
이 공식을 쓰는 이유는 제시된 코드가 전형적인 이진 탐색 반복문이기 때문이다.

계산 과정

1회: low=0, high=15이므로 middle=7이고 list[7]=27이다. 34>27이므로 low=8이 된다.
2회: low=8, high=15이므로 middle=11이고 list[11]=38이다. 34<38이므로 high=10이 된다.
3회: low=8, high=10이므로 middle=9이고 list[9]=30이다. 34>30이므로 low=10이 된다.
4회: low=10, high=10이므로 middle=10이고 list[10]=34이다. 이때 key와 일치하므로 반환한다.
따라서 while 반복은 총 4번 실행된다.

선지별 해설

1번 X: 반복 횟수를 3번으로 보면 마지막으로 list[10]=34를 확인하는 비교를 빠뜨린 것이다. 올바르게는 4번째 반복에서 middle=10을 확인한 뒤 탐색이 끝난다.
2번 O: 1회에 27, 2회에 38, 3회에 30, 4회에 34를 차례로 비교하므로 반복 횟수는 4번이다.
3번 X: 반복 횟수를 5번으로 보면 정답을 찾은 뒤에도 반복이 한 번 더 돈다고 착각한 것이다. 올바르게는 4번째 반복에서 key==list[middle]가 성립하여 즉시 return한다.
4번 X: 반복 횟수를 6번으로 보는 것은 선형 탐색처럼 한 칸씩 검사한다고 오해한 것이다. 올바르게는 탐색 구간을 절반씩 줄이는 이진 탐색이므로 4번 만에 34를 찾는다.

오답 포인트

반복 횟수는 값의 위치만 보고 감으로 세면 틀리기 쉽다. 특히 정답을 찾는 마지막 비교도 while 반복 1회에 포함된다는 점과, middle 계산 후 비교가 끝나야 다음 low, high가 정해진다는 점이 함정이다.

다시 나오면 이렇게 판단

이진 탐색 반복 횟수 문제는 매 회차마다 low, $high$, $middle$을 직접 적고, 비교한 값까지 순서대로 기록하면 된다. key를 찾은 회차도 반복 1회로 센다.

암기 포인트

이진 탐색 횟수는 중간값 비교 횟수와 같고, 정답을 찾은 마지막 비교도 포함된다.

NoteBase

문 22. 버킷(bucket)이 13개이고 버킷당 슬롯(slot)이 1개인 해시 테이블에 키 값 <7, 20, 3, 19, 45>를 차례대로 저장하고자 한다. 다음 <조건>에 따라 아래 <해시 함수>를 이용하여 작동하는 이중 해싱(double hashing)에서, 마지막 키 값 45의 저장 위치는?

<조건>
◯ 버킷의 인덱스는 0부터 시작한다.
◯ $h(x)$는 첫 번째 조사 위치를 결정하는 기본 해시 함수이다.
◯ $f(x)$는 충돌 발생 시 조사 위치 간격을 결정하는 함수이다.
◯ 하나의 키 값에 대한 해시 함수 연산 중 충돌이 1번 이상 발생할 경우 i번째 충돌에 대응하여 $h_i(x)$를 적용한다.
<해시 함수>
h(x)=x 13
f(x)=x 11
h_i(x)=(h(x)+i × f(x)) 13

① 5
② 7
③ 8
④ 11

정답 3번

출제 포인트

이 문제는 이중 해싱에서 기본 해시값과 충돌 횟수별 재탐사 위치를 순서대로 계산해 최종 저장 버킷을 구하는 문제이다.

조건 정리

주어진 값: 버킷 수는 13개, 버킷 인덱스는 0부터 시작, 저장 순서는 7, 20, 3, 19, 45이다.
구해야 할 값: 마지막 키 45의 최종 저장 위치이다.
주의할 조건: 첫 위치는 h(x)=x 13, 충돌이 나면 h_i(x)=(h(x)+i× f(x)) 13을 충돌 횟수에 따라 순서대로 적용한다.

NoteBase

사용할 공식

기본 해시 함수는 h(x)=x 13이고, 이중 해싱의 재탐사 함수는 h_i(x)=(h(x)+i× f(x)) 13, 보조 해시 함수는 f(x)=x 11이다.
이 공식을 쓰는 이유: 문제에서 첫 조사 위치와 충돌 시 조사 간격을 각각 따로 주었으므로, 각 키를 저장할 때 기본 위치를 먼저 구하고 충돌 횟수만큼 재탐사 위치를 계산해야 한다.

계산 과정

1. 7은 $h(7) = 7$이므로 7번 버킷에 저장된다.
2. 20은 $h(20) = 7$이라 충돌하고, $f(20) = 9$이므로 h_1(20)=(7+9) 13=3이 되어 3번 버킷에 저장된다.
3. 3은 $h(3) = 3$이라 충돌하고, $f(3) = 3$이므로 h_1(3)=(3+3) 13=6이 되어 6번 버킷에 저장된다.
4. 19는 $h(19) = 6$이라 충돌하고, $f(19) = 8$이므로 h_1(19)=(6+8) 13=1이 되어 1번 버킷에 저장된다.
5. 45는 $h(45) = 6$이라 첫 충돌이 발생하고, $f(45) = 1$이므로 h_1(45)=(6+1) 13=7이다. 7번도 이미 차 있으므로 두 번째 충돌이 발생한다. 따라서 h_2(45)=(6+2× 1) 13=8이 되고, 8번 버킷이 비어 있으므로 최종 저장 위치는 8이다.

선지 판단

1번 X: 5번 버킷이라는 판단이 틀리다. 45는 $h(45) = 6$에서 시작해 7을 거쳐 $h_2(45) = 8$에 저장되므로, 올바르게는 8번 버킷이다.
2번 X: 7번 버킷이라는 판단은 첫 번째 재탐사 위치까지만 보고 멈춘 경우이다. 7번 버킷에는 이미 7이 저장되어 있으므로 한 번 더 충돌 처리해야 하고, 올바르게는 8번 버킷이다.
3번 O: 45는 6번에서 충돌한 뒤 7번에서도 다시 충돌하고, 두 번째 재탐사 결과 $h_2(45) = 8$이므로 8번 버킷에 저장된다.
4번 X: 11번 버킷이라는 판단은 보조 해시 함수값이나 모듈러 계산을 저장 위치와 혼동한 경우이다. 실제 저장 위치는 $f(45) = 1$ 자체가 아니라 $h_i(45)$로 구하며, 올바르게는 8번 버킷이다.

오답 포인트

첫 충돌 뒤의 $h_1(x)$만 계산하고 멈추면 7번을 고르게 된다. 이 문제의 함정은 45가 한 번이 아니라 두 번 충돌한다는 점과, 저장 위치가 $f(x)$가 아니라 $h_i(x)$라는 점이다.

다시 나오면 이렇게 판단

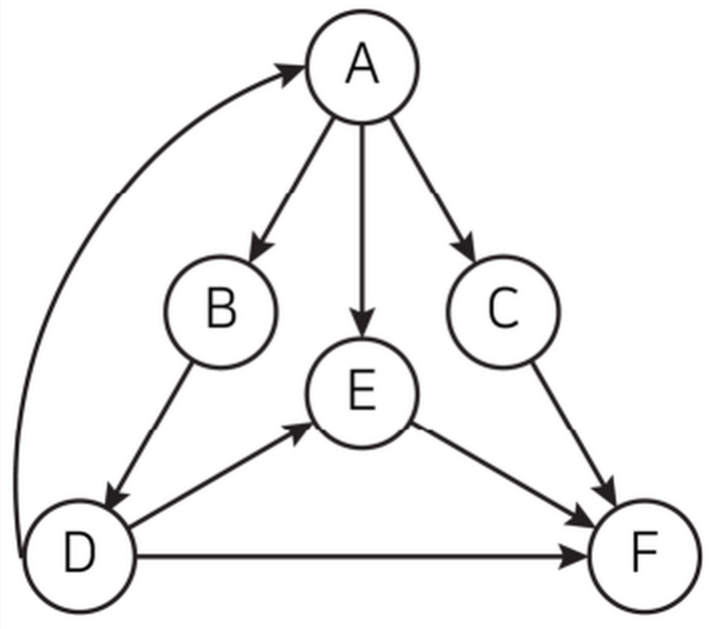
이중 해싱 문제는 먼저 앞선 키들로 해시 테이블 상태를 채운 뒤, 새 키에 대해 $h(x)$를 계산하고 충돌이 날 때마다 $h_1(x), h_2(x), \ldots$를 빈 버킷이 나올 때까지 순서대로 확인하는 방식으로 판단한다.

암기 포인트

이중 해싱의 최종 위치는 $f(x)$가 아니라 $h(x) + i \times f(x)$의 충돌 단계별 결과이다.

NoteBase

문 23. 다음 그래프에 대한 설명 중 옳은 것만을 모두 고르면?



ㄱ. 가장 긴 단순 경로의 길이는 4이다.
ㄴ. 진입차수가 진출차수보다 큰 정점은 1개이다.
ㄷ. 완전 그래프가 되기 위해 추가로 필요한 간선의 수는 10개이다.
ㄹ. 간선 <A, B>를 제거한 후 위상정렬을 수행하면 B, D, A, E, C, F 또는 B, D, A, C, E, F순으로 정점이 나열된다.

① ㄱ, ㄹ
② ㄴ, ㄷ
③ ㄱ, ㄴ, ㄹ
④ ㄴ, ㄷ, ㄹ

정답 1번

출제 포인트

이 문제는 방향 그래프에서 단순 경로의 길이, 각 정점의 진입차수와 진출차수, 완전 그래프 판단, 간선 제거 후 위상정렬 결과를 함께 판단하는 문제이다.

NoteBase

선지별 해설

① O: ㄱ은 B→D→A→C→F 또는 B→D→A→E→F처럼 길이 4의 단순 경로가 존재하므로 옳다. ㄹ은 간선 A→B를 제거하면 진입차수 0인 정점이 B뿐이어서 B, D, A까지는 고정되고, 그 뒤 E와 C의 순서만 바뀔 수 있으므로 B, D, A, E, C, F 또는 B, D, A, C, E, F가 된다.
② X: ㄴ이 틀렸으므로 이 조합은 성립하지 않는다. 진입차수가 진출차수보다 큰 정점은 E가 2대1, F가 3대0이므로 모두 2개이다. '1개'가 아니라 '2개'로 고쳐야 한다. ㄷ도 틀리므로 이 선택지는 이중으로 오답이다.
③ X: ㄱ은 옳지만 ㄴ이 틀렸으므로 이 조합은 성립하지 않는다. 진입차수가 진출차수보다 큰 정점 수는 1개가 아니라 2개이다. 또한 ㄹ은 옳으므로 정답 조합은 ㄱ, ㄴ, ㄹ이 아니라 ㄱ, ㄹ이다.
④ X: ㄴ이 틀리고 ㄷ도 틀리므로 이 조합은 성립하지 않는다. 이 그래프의 기저 무방향 그래프에는 이미 9개의 간선이 있으므로 정점 6개의 완전 그래프 K_6가 되려면 전체 15개 간선 중 부족한 6개만 추가하면 된다. '10개'가 아니라 '6개'로 고쳐야 한다.

오답 포인트

경로 길이를 방문 정점 수로 세거나 완전 그래프를 방향 간선 수 기준으로 혼동하면 틀리기 쉽다. 이 문항에서는 경로 길이를 간선 수로 세고, 완전 그래프는 정점 쌍이 모두 연결된 기저 무방향 그래프로 이해해야 한다.
진입차수와 진출차수도 각 정점마다 직접 세어야 한다.

다시 나오면 이렇게 판단

먼저 각 정점의 진입차수와 진출차수를 적고, 다음으로 가장 긴 단순 경로 후보를 실제로 이어 보면서 반복 정점이 생기는지 확인한다. 위상정렬 문제는 진입차수 0인 정점부터 차례로 제거하여 고정되는 부분과 순서가 바뀔 수 있는 부분을 나누면 판단할 수 있다.

암기 포인트

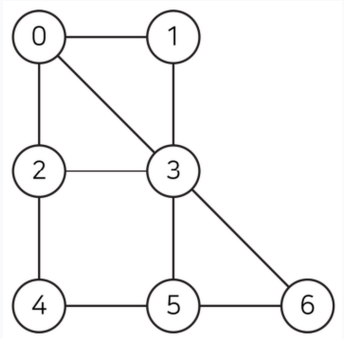
단순 경로의 길이는 간선 수이고, 위상정렬은 진입차수 0인 정점부터 시작하며, K_6의 간선 수는 15이다.

NoteBase

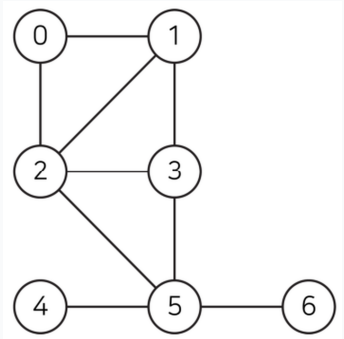
문 24. 무방향 그래프(undirected graph) G의 인접 행렬(adjacent matrix) 표현을 C 언어의 2차원 배열을 이용하여 다음과 같이 구현했다. 그래프 G의 정점(vertex)이 0, 1, 2, 3, 4, 5, 6의 숫자로 표현될 때, 무방향 그래프 G의 그림으로 옳은 것은?

```
int G[7][7] = {
                    { 0, 1, 1, 0, 0, 0, 0 },
                    { 1, 0, 1, 1, 0, 0, 0 },
                    { 1, 1, 0, 1, 0, 1, 0 },
                    { 0, 1, 1, 0, 0, 1, 0 },
                    { 0, 0, 0, 0, 0, 1, 0 },
                    { 0, 0, 1, 1, 1, 0, 1 },
                    { 0, 0, 0, 0, 0, 1, 0 }
              };
```

①

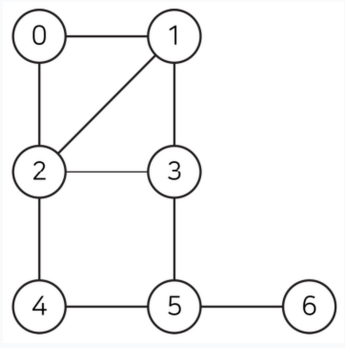


②

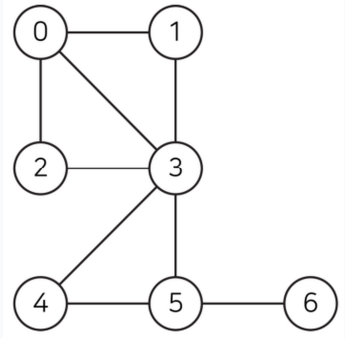


NoteBase

③



④



정답 2번

출제 포인트

이 문제는 무방향 그래프의 인접 행렬에서 값이 1인 위치만 간선으로 복원하여 전체 연결 구조를 정확히 읽을 수 있는지를 묻는다. 이 행렬에서 확인되는 간선은 0-1, 0-2, 1-2, 1-3, 2-3, 2-5, 3-5, 4-5, 5-6이다.

선지별 해설

1번: X. 그림에 0-3 간선과 2-4 간선, 3-6 간선이 추가되어 있는데 행렬의 해당 위치는 0이다. 반대로 행렬에서 1인 1-2 간선이 빠져 있다. 틀린 부분은 0-3, 2-4, 3-6 연결이라는 점이고, 올바르게는 1-2가 연결되고 0-3, 2-4, 3-6은 연결되지 않아야 한다.
2번: O. 0은 1, 2와만 연결되고 1은 0, 2, 3과 연결되며 2는 0, 1, 3, 5와 연결된다. 또한 3은 1, 2, 5와, 4는 5와, 5는 2, 3, 4, 6과, 6은 5와만 연결되어 행렬과 정확히 일치한다.
3번: X. 그림에는 2-4 간선이 있지만 행렬에서 2행 4열 값은 0이다. 대신 행렬에서 1인 2-5 간선이 그림에서 빠져 있다. 틀린 부분은 2가 4와 연결된다는 점이고, 올바르게는 2는 5와 연결되고 4와는 연결되지 않는다.
4번: X. 그림에는 0-3 간선과 3-4 간선이 보이지만 행렬에서는 둘 다 0이다. 또 행렬에서 1인 1-2 간선과 2-5 간선이 빠져 있다. 틀린 부분은 0-3, 3-4 연결이라는 점이고, 올바르게는 1-2와 2-5가 연결되어야 하며 0-3과 3-4는 연결되지 않는다.

NoteBase

오답 포인트

인접 행렬 문제에서는 눈에 띄는 대각선 간선을 먼저 따라가다 보면 없는 간선을 있다고 보거나, 실제로 필요한 간선을 하나 빼먹기 쉽다. 특히 이 문항은 0-3, 2-4, 3-4, 3-6처럼 그려 넣기 쉬운 간선과 1-2, 2-5처럼 놓치기 쉬운 간선을 함께 섞어 함정을 만든 문항이다.

다시 나오면 이렇게 판단

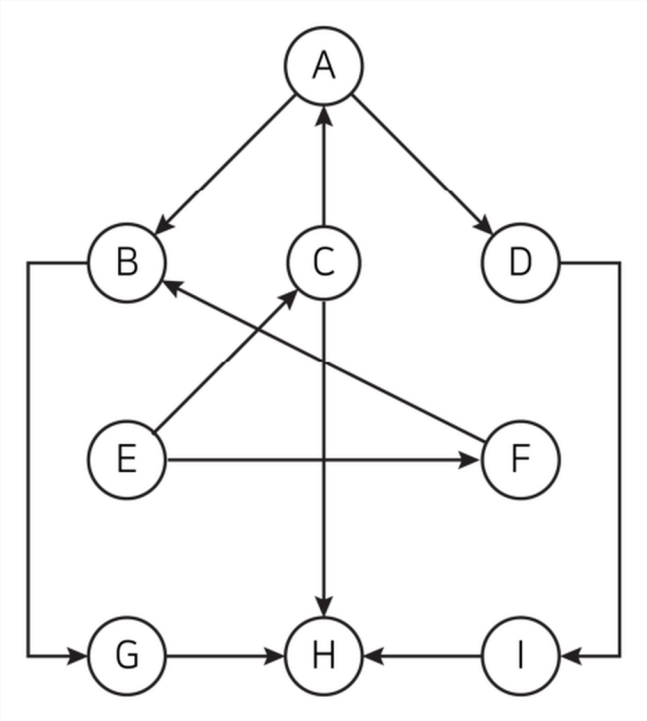
각 행에서 1이 있는 열만 적어 정점별 인접 리스트로 먼저 바꾼 뒤, 각 정점의 차수와 연결 상대를 그림과 대조하면 된다. 무방향 그래프이므로 한 번 확인한 간선은 중복 세지 않고, 없는 대각선 간선이 추가되지 않았는지를 끝에 다시 확인하면 된다.

암기 포인트

인접 행렬의 1은 간선 하나이고, 정답 그래프는 각 정점의 연결 상대 집합이 행별로 완전히 같아야 한다.

NoteBase

문 25. 다음 방향 그래프(directed graph)에서 노드 A부터 시작하여 깊이 우선 탐색과 너비 우선 탐색을 수행하는 경우, 여섯 번째로 방문하는 노드의 쌍을 바르게 연결한 것은? (단, 노드 A는 첫 번째 방문 노드이고, 인접 노드가 다수일 경우 방문 순서는 알파벳순으로 하며, 탐색 과정에서 도달하지 못하는 경로는 고려하지 않는다)



① 깊이 우선 탐색: C, 너비 우선 탐색: D

② 깊이 우선 탐색: D, 너비 우선 탐색: I

③ 깊이 우선 탐색: H, 너비 우선 탐색: H

④ 깊이 우선 탐색: I, 너비 우선 탐색: H

정답 4번

출제 포인트

이 문제는 방향 그래프에서 시작 노드 A로부터 도달 가능한 노드만 대상으로 하여 DFS와 BFS의 방문 순서를 정확히 구분하는지를 묻는 문제이다.

올바른 순서

DFS 방문 순서는 A(1) -> B(2) -> G(3) -> H(4) -> D(5) -> I(6)이다.
BFS 방문 순서는 A(1) -> B(2) -> D(3) -> G(4) -> I(5) -> H(6)이다.
왜 이 순서인가: 방향 그래프이므로 각 노드의 outgoing edge만 따라가며, A에서 C·E·F로 가는 경로는 없어서 방문 대상에서 제외된다.

NoteBase

선지별 해설

① X
- C와 D를 6번째 방문 노드로 본 것은 도달 가능성부터 잘못 본 것이다. A에서 C로 가는 간선이 없고 D는 BFS에서 3번째로 방문된다.
- 올바르게는 DFS 6번째는 I이고 BFS 6번째는 H이다.

② X
- DFS 6번째를 D로 본 것이 틀리다. DFS는 A -> B -> G -> H까지 한 갈래를 끝까지 내려간 뒤 되돌아와 D를 5번째, I를 6번째로 방문한다.
- 올바르게는 DFS 6번째는 I이고 BFS 6번째는 H이다.

③ X
- BFS 6번째를 H로 본 점은 맞지만 DFS 6번째를 H로 본 점이 틀리다. H는 DFS에서 G 다음으로 4번째에 방문된다.
- 올바르게는 DFS 6번째는 I이고 BFS 6번째는 H이다.

④ O
- DFS는 A의 인접 노드 중 알파벳순으로 B를 먼저 따라가 B -> G -> H를 방문한 뒤 되돌아와 D -> I로 진행하므로 6번째가 I이다.
- BFS는 A의 인접 노드 B, D를 먼저 큐에 넣고 다음 레벨의 G, I, H 순으로 방문하므로 6번째가 H이다.

오답 포인트

방향 그래프인데도 화살표 반대 방향을 따라가 C, E, F를 방문 대상에 넣거나, DFS와 BFS 모두 단순히 알파벳순 나열로 처리하는 착각이 자주 나온다. 이 문제는 먼저 A에서 실제로 도달 가능한 노드 집합을 확정한 뒤 그 안에서 DFS와 BFS의 규칙을 적용해야 한다.

다시 나오면 이렇게 판단

방향 그래프 탐색 문제는 먼저 시작 노드에서 나가는 간선만 따라 도달 가능한 노드를 추린 뒤, DFS는 한 갈래를 끝까지 내려가며 번호를 매기고 BFS는 같은 레벨의 노드를 큐 순서대로 세면 된다.

암기 포인트

DFS는 깊게 내려간 뒤 되돌아오고, BFS는 가까운 레벨부터 넓게 방문한다.

NoteBase