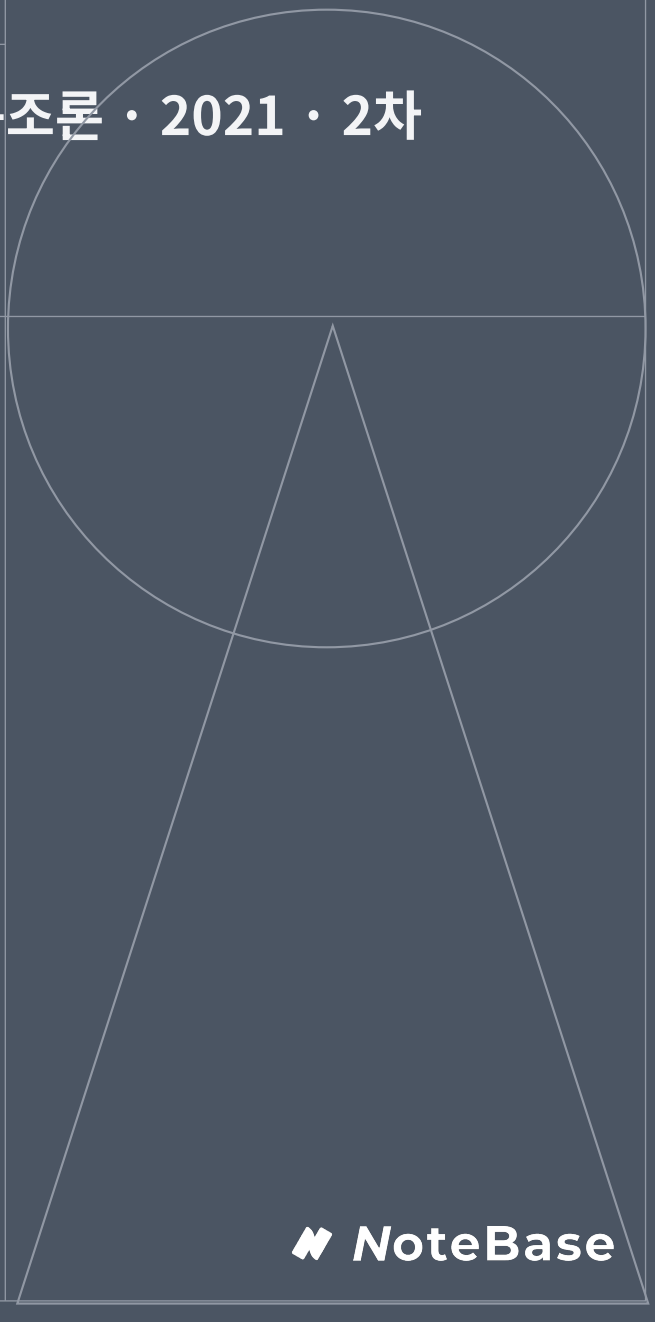
기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2021 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2021 · 2차

공무원/국가직/7급 · 자료구조론 · 2021 · 2차

문 1. 다음 탐색 알고리즘에 대한 설명으로 옳지 않은 것은?

```
int a_search(int list[], int key, int n) {
  int i;
  for(i=0; i<n; i++)
    if(list[i]==key) return i;
  return -1;
}
```

① 탐색에 성공하면 키 값의 인덱스를 반환한다.

② 최선의 경우, 시간 복잡도는 빅오 표기법으로 O(1)이다.

③ 최악의 경우, 시간 복잡도는 빅오 표기법으로 O(n)이다.

④ 평균적인 경우, 시간 복잡도는 빅오 표기법으로 O((n+1)/2)이다.

정답 4번

출제 포인트

제시된 a_search 함수는 배열을 처음부터 끝까지 순서대로 하나씩 비교하는 순차 탐색(선형 탐색, sequential search) 알고리즘이다. 이 문항은 순차 탐색의 최선·최악·평균 경우의 시간 복잡도를 정확히 알고 있는지, 그리고 빅오(Big-O) 표기법을 올바르게 사용할 수 있는지를 함께 묻는다.

조건 정리

for(i=0; i<n; i++) 반복문 안에서 list[i]==key 를 만족하면 그 즉시 인덱스 i를 반환하고 종료하며, 끝까지 못 찾으면 -1을 반환한다. 즉 비교 횟수는 key가 배열의 몇 번째 위치에 있는지(또는 없는지)에 따라 최소 1회에서 최대 n회까지 달라진다.

NoteBase

사용할 공식

순차 탐색의 평균 비교 횟수는 $\frac{1+2+\cdots+n}{n}=\frac{n+1}{2}$이다. 다만 빅오 표기법은 입력 크기가 커질 때의 증가 차수(order)만을 나타내는 표기이므로 상수 계수나 저차항을 제거하여 $O(n)$으로 단순화해서 써야 한다. 즉 $O\left(\frac{n+1}{2}\right)$이라는 표현 자체가 빅오 표기법의 정의에 어긋난다.

계산 과정

최선의 경우: key가 배열의 첫 번째 원소(list[0])와 일치하면 반복문이 1회만 실행되고 바로 반환되므로 시간 복잡도는 $O(1)$이다. 최악의 경우: key가 배열에 없거나 마지막 원소에 있으면 반복문이 n회 모두 실행되므로 시간 복잡도는 $O(n)$이다. 평균의 경우: 비교 횟수의 평균은 $\frac{n+1}{2}$회로 계산되지만, 빅오 표기법으로 옮길 때는 계수 $\frac{1}{2}$과 상수항 $\frac{1}{2}$을 무시하고 최고차항인 n만 남겨 $O(n)$으로 표기해야 한다.

선지 판단

① O: return i 문에 의해 탐색에 성공하면 key와 일치하는 원소의 인덱스 i를 반환한다.
② O: 최선의 경우는 i=0에서 바로 일치하는 경우이므로 반복이 1회로 끝나 O(1)이다.
③ O: 최악의 경우는 배열 끝까지 비교해야 하는 경우이므로 반복이 n회 실행되어 O(n)이다.
④ X: 평균 비교 횟수 자체는 $\frac{n+1}{2}$회로 맞게 계산했지만, 이를 빅오 표기법으로 쓸 때 계수와 상수항을 남긴 $O\left(\frac{n+1}{2}\right)$ 형태로 표기한 것이 틀렸다. 빅오 표기법의 정의상 평균의 경우도 최악의 경우와 같은 차수인 $O(n)$으로 표기해야 한다. → 올바른 표현은 "평균적인 경우, 시간 복잡도는 빅오 표기법으로 O(n)이다."

오답 포인트

실제 비교 횟수의 평균값이 $\frac{n+1}{2}$이라는 계산 자체는 맞기 때문에, 이 값을 그대로 빅오 표기법 안에 넣어도 된다고 착각하기 쉽다. 하지만 빅오 표기법은 정확한 횟수를 나타내는 식이 아니라 입력이 커질 때의 증가 추세(차수)만 나타내는 표기법이므로, 계수와 저차항·상수항을 모두 제거하고 최고차항만 남겨야 한다. 이 구분을 못 하면 ④를 옳은 설명으로 착각하게 된다.

다시 나오면 이렇게 판단

선지에서 빅오 표기법 안에 분수나 (n+1)/2 같은 구체적인 계수·상수가 그대로 남아 있으면 일단 표기법 오류를 의심한다. 정확한 평균 비교 횟수를 구하는 계산과, 그 결과를 빅오 표기법으로 옮기는 것은 별개의 단계이며, 후자에서는 항상 최고차항만 남기고 나머지는 버린다는 원칙을 적용한다.

암기 포인트

순차 탐색: 최선 O(1), 최악 O(n), 평균 O(n). 평균 비교 횟수의 실제 값은 (n+1)/2회이지만, 빅오 표기법으로는 계수·상수를 제거하여 O(n)으로 쓴다.

NoteBase

문 2. 다음 C 코드의 실행 결과는?

```
#include <stdio.h>
int func (int n) {
  if(n <= 1) return 2;
  return func (n-1) + n;
}
int main (int argc, char* argv[]) {
  printf("%d\n", func (10) );
  return 0;
}
```

① 55
② 56
③ 58
④ 60

정답 2번

출제 포인트

재귀 함수 func(n)의 종료 조건과 재귀호출 시 누적되는 값을 정확히 추적해 func(10)의 반환값을 계산할 수 있는지를 묻는다. 종료 조건이 n<=1일 때 0이 아니라 2를 반환한다는 점을 놓치면 오답이 나온다.

조건 정리

func(n)은 n<=1이면 2를 반환하고, 그렇지 않으면 func(n-1)+n을 반환한다. main에서 func(10)의 값을 출력한다.

사용할 공식

n<=1일 때 func(n) = 2, n>1일 때 func(n) = func(n-1) + n. 이를 풀어 쓰면 func(10) = func(1) + (2+3+4+\cdots+10) = 2 + \sum_{k=2}^{10} k 이다.

NoteBase

계산 과정

func(1) = 2
func(2) = func(1) + 2 = 2 + 2 = 4
func(3) = func(2) + 3 = 4 + 3 = 7
func(4) = func(3) + 4 = 7 + 4 = 11
func(5) = func(4) + 5 = 11 + 5 = 16
func(6) = func(5) + 6 = 16 + 6 = 22
func(7) = func(6) + 7 = 22 + 7 = 29
func(8) = func(7) + 8 = 29 + 8 = 37
func(9) = func(8) + 9 = 37 + 9 = 46
func(10) = func(9) + 10 = 46 + 10 = 56
따라서 printf가 출력하는 값은 56이다.

선지 판단

① 55 → X. func(1)의 종료값을 2가 아니라 1로 잘못 계산하거나 2부터 10까지의 합만 구했을 때(2+3+\cdots+10=54, 여기에 종료값 1을 더하면 55) 나오는 오답이다.
② 56 → O. func(1)=2에서 시작해 2부터 10까지를 차례로 더하면 2+2+3+4+5+6+7+8+9+10=56이 되어 계산 과정과 일치한다.
③ 58 → X. 종료 조건을 n<=1이 아니라 n<=0으로 잘못 적용해 func(1)까지 한 번 더 자기 자신을 더하는 식으로 계산하면 나올 수 있는 오답이다.
④ 60 → X. 2부터 10까지의 합 54에 잘못된 종료값(6)을 더하거나 등차수열 합 공식을 잘못 적용했을 때 나오는 오답이다.

오답 포인트

이 문제의 함정은 재귀의 종료값이 0이 아니라 2라는 점이다. 종료값을 0으로 착각하면 2~10의 합인 54가 나와 정답과 2만큼 차이가 생긴다. 또한 func(n-1)+n에서 더해지는 n의 범위(2부터 10까지)를 정확히 세지 않으면 계산이 어긋난다.

다시 나오면 이렇게 판단

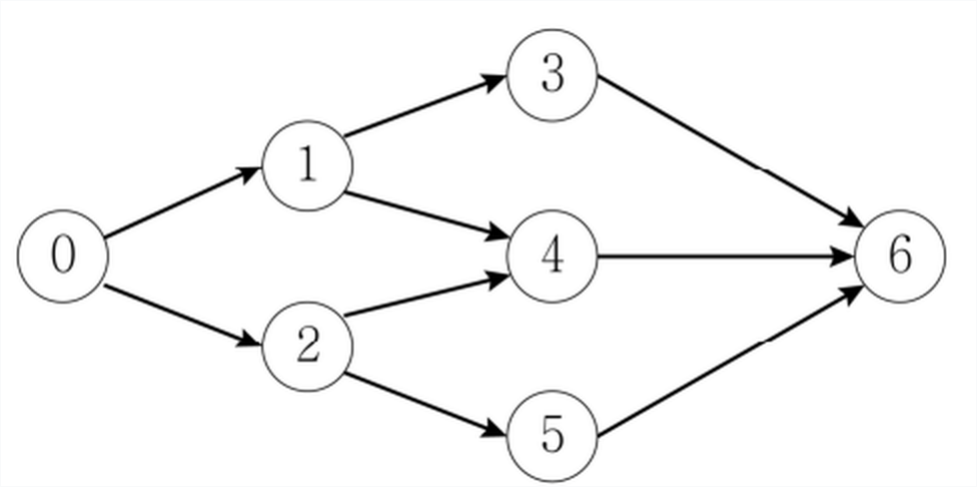
재귀 함수 문제는 먼저 종료 조건(base case)이 무엇을 반환하는지부터 확정한 다음, 재귀식이 어떤 값을 몇 번 누적하는지를 표로 나열해 하나씩 계산한다. n=10처럼 작지 않은 값이 주어지면 손으로 직접 func(1)부터 func(10)까지 순서대로 채워가는 것이 실수를 줄이는 방법이다.

암기 포인트

재귀 함수의 결과는 종료 조건의 반환값과 재귀식이 누적하는 항들의 합으로 이루어진다. func(10) = 종료값 2 + (2+3+\cdots+10)이라는 구조를 기억해 둔다.

NoteBase

문 3. 다음 그래프에 대해 위상 정렬(topological sort) 알고리즘을 수행했을 때 생성되는 위상 순서 중 옳지 않은 것은?



① 0, 1, 2, 3, 4, 5, 6

② 0, 1, 3, 2, 4, 5, 6

③ 0, 2, 4, 5, 1, 3, 6

④ 0, 2, 5, 1, 3, 4, 6

정답 3번

출제 포인트

위상 정렬(topological sort)은 방향 비순환 그래프(DAG)에서 모든 간선 $u \rightarrow v$에 대해 정점 u가 v보다 먼저 나오도록 정점을 나열하는 것이다. 이 문제는 주어진 4개의 순서 중 어느 것이 그래프의 간선 방향과 모순되는지, 즉 위상 정렬의 조건을 위반하는지를 판별하는 문제이다.

올바른 순서

그래프의 간선은 0→1, 0→2, 1→3, 1→4, 2→4, 2→5, 3→6, 4→6, 5→6이다. 어떤 나열이 위상 순서로 옳으려면 이 9개 간선 전부에 대해 화살표의 시작 정점이 나열에서 도착 정점보다 앞에 위치해야 한다. 특히 1→4 간선이 핵심 제약인데, 이는 1이 반드시 4보다 앞에 와야 함을 의미한다.

NoteBase

선지별 해설

① 0, 1, 2, 3, 4, 5, 6 → O. 각 정점의 위치는 0(1번째), 1(2번째), 2(3번째), 3(4번째), 4(5번째), 5(6번째), 6(7번째)이다. 1→3(2번째→4번째), 1→4(2번째→5번째), 2→4(3번째→5번째), 2→5(3번째→6번째) 등 모든 간선에서 시작 정점이 도착 정점보다 앞서므로 올바른 위상 순서이다.
② 0, 1, 3, 2, 4, 5, 6 → O. 위치는 0(1), 1(2), 3(3), 2(4), 4(5), 5(6), 6(7)이다. 1→4는 2번째→5번째로 순서가 맞고, 1→3은 2번째→3번째로 맞으며, 2→4(4번째→5번째), 2→5(4번째→6번째)도 모두 앞뒤 관계가 맞으므로 올바른 위상 순서이다.
③ 0, 2, 4, 5, 1, 3, 6 → X. 위치는 0(1), 2(2), 4(3), 5(4), 1(5), 3(6), 6(7)이다. 그런데 간선 1→4는 1이 4보다 먼저 나와야 하는데, 이 나열에서는 4가 3번째, 1이 5번째로 4가 1보다 먼저 나온다. 이는 1→4 간선의 방향과 정면으로 모순되므로 위상 정렬 결과로 나올 수 없는 순서이다.
④ 0, 2, 5, 1, 3, 4, 6 → O. 위치는 0(1), 2(2), 5(3), 1(4), 3(5), 4(6), 6(7)이다. 1→4는 4번째→6번째로 순서가 맞고, 1→3(4번째→5번째), 2→4(2번째→6번째), 2→5(2번째→3번째)도 모두 앞뒤 관계가 맞으므로 올바른 위상 순서이다.

오답 포인트

이 문제에서 학생들이 놓치기 쉬운 부분은 1과 4의 관계이다. 그래프 그림만 훑어보면 2에서 4로 가는 간선(2→4)만 눈에 띄어 4가 2 계열에만 속한다고 착각하기 쉬운데, 실제로는 1에서도 4로 가는 간선(1→4)이 존재한다. 즉 4는 1과 2 양쪽 모두로부터 간선을 받는 정점이므로, 위상 순서에서 4는 1보다도 뒤에, 2보다도 뒤에 와야 한다. ③번은 4가 1보다 먼저 나와 이 조건을 어긴 것이다.

다시 나오면 이렇게 판단

위상 정렬 순서의 옳고 그름을 판별할 때는 나열을 눈으로 훑기보다, 그래프의 모든 간선을 하나씩 짚어가며 나열에서의 위치(인덱스)를 확인하는 방식으로 접근한다. 각 정점에 나열 순서상의 번호를 매긴 뒤, 모든 간선에 대해 (시작 정점 번호) < (도착 정점 번호)가 성립하는지 전수 검사하면 실수 없이 답을 찾을 수 있다. 특히 하나의 정점으로 여러 방향에서 간선이 들어오는 경우(이 문제의 4처럼) 그 모든 선행 정점과의 관계를 빠짐없이 확인해야 한다.

암기 포인트

위상 정렬이 성립하려면 그래프의 모든 간선 $u \to v$에 대해 나열에서 u가 v보다 앞에 있어야 한다는 조건이 하나라도 깨지면 그 나열은 위상 정렬 결과가 될 수 없다. 어떤 정점이 여러 정점으로부터 간선을 받는다면, 그 정점은 반드시 그 모든 선행 정점들 뒤에 위치해야 한다는 점을 기억해 둔다.

NoteBase

문 4. 다음과 같이 배열을 이용하여 스택 자료구조를 정의하였다. ㉠, ㉡에 들어갈 내용을 바르게 연결한 것은?

```
#define MAX 10
int stack[MAX];
int top = -1;
int push(int t) {
  if (top >= MAX - 1) {
    printf("\n Stack overflow.");
    return -1;
  }
  stack[____㉠____] = t;
  return t;
}
int pop(void) {
  if (top < 0) {
    printf("\n Stack underflow.");
    return -1;
  }
  return stack[____㉡____];
}
```

① ㉠: ++top, ㉡: --top

② ㉠: ++top, ㉡: top--

③ ㉠: top++, ㉡: --top

④ ㉠: top++, ㉡: top--

정답 2번

출제 포인트

배열 기반 스택 구현에서 top 변수의 전위 증가(++top)와 후위 증가(top++), 전위 감소(--top)와 후위 감소(top--)가 push·pop 동작에서 실제로 어떤 순서로 연산되는지 구분할 수 있는지를 묻는다. top이 마지막으로 채워진 원소의 인덱스를 가리키는 방식(top = -1로 초기화)에서는 push할 때 인덱스를 먼저 올린 뒤 저장해야 하고, pop할 때는 현재 위치의 값을 먼저 꺼낸 뒤 인덱스를 내려야 한다는 점이 핵심이다.

NoteBase

핵심 개념

전위 연산자(++top, --top)는 값을 먼저 변경한 다음 그 변경된 값을 사용하고, 후위 연산자(top++, top--)는 현재 값을 먼저 사용한 다음 값을 변경한다. push 시점에는 top이 아직 빈 공간(-1 또는 마지막으로 채운 인덱스)을 가리키고 있으므로, 새 원소를 넣을 자리는 top을 하나 올린 위치이다. 따라서 stack[++top] = t 형태로 먼저 top을 증가시킨 뒤 그 인덱스에 저장해야 첫 push 때 top이 -1에서 0으로 바뀌어 stack[0]에 정상적으로 저장된다. 반대로 stack[top++] = t라면 첫 push 때 여전히 -1인 top으로 stack[-1]에 접근하려 하므로 배열 범위를 벗어나는 오류가 된다. pop 시점에는 top이 현재 채워진 마지막 원소의 인덱스를 정확히 가리키고 있으므로, 그 값을 먼저 반환한 다음 top을 하나 내려야 한다. 따라서 return stack[top--]처럼 현재 인덱스의 값을 먼저 사용하고 나서 top을 감소시켜야 다음 pop이 그 다음 원소를 올바르게 가리킨다. 만약 return stack[--top]으로 쓰면 top을 먼저 내린 뒤 그 값을 반환하게 되어, 실제로는 하나 아래 원소를 반환하고 가장 위에 있던 원소는 건너뛰게 된다.

선지별 해설

① ㉠ ++top, ㉡ --top → ㉠은 옳으나 ㉡이 틀렸다. pop에서 --top을 쓰면 top을 먼저 감소시킨 뒤 그 위치의 값을 반환하므로, 실제 스택의 맨 위 원소가 아니라 그 아래 원소를 반환하게 되어 가장 최근에 넣은 값을 잃어버린다.

② ㉠ ++top, ㉡ top-- → 옳다. push에서는 stack[++top] = t로 top을 먼저 1 증가시킨 뒤 그 인덱스에 값을 저장하여 -1에서 시작한 top이 첫 push 때 0이 되고 stack[0]에 정확히 저장된다. pop에서는 return stack[top--]로 현재 top이 가리키는 값을 먼저 반환한 뒤 top을 1 감소시켜, 나중에 넣은 값이 먼저 나오는 후입선출(LIFO) 구조가 그대로 지켜진다.

③ ㉠ top++, ㉡ --top → 둘 다 틀렸다. push에서 stack[top++] = t를 쓰면 top이 -1인 상태에서 먼저 stack[-1]에 접근한 뒤 top을 0으로 올리게 되어 배열 인덱스 범위(음수)를 벗어나는 오류가 발생한다. pop의 --top도 ①과 같은 이유로 맨 위 원소를 건너뛰는 오류를 일으킨다.

④ ㉠ top++, ㉡ top-- → ㉡은 옳으나 ㉠이 틀렸다. push에서 top++를 쓰면 첫 호출 시 top이 여전히 -1인 채로 stack[-1]에 접근하게 되어 초기 push부터 잘못된 위치에 값을 쓰게 된다.

오답 포인트

전위·후위 연산자를 단순히 '증가냐 감소냐'로만 구분하지 말고, 연산이 대괄호 안의 인덱스 계산에 언제 반영되는지를 따져야 한다. push는 top이 아직 빈 자리를 가리키고 있다가 새 값을 넣을 자리로 바뀌어야 하므로 먼저 증가(전위)가 맞고, pop은 top이 이미 채워진 자리를 정확히 가리키고 있으므로 그 값을 먼저 쓰고 나중에 감소(후위)해야 한다는 점을 혼동하지 않아야 한다. top++와 ++top의 결과값 차이를 헷갈려 push에서 후위 증가를 고르는 실수가 가장 흔하다.

NoteBase

다시 나오면 이렇게 판단

top = -1로 초기화된 배열 스택 문제가 나오면, push는 '먼저 올리고 저장'(전위 증가), pop은 '먼저 꺼내고 내림'(후위 감소)이라는 대응 관계를 기준점으로 삼아 빠르게 판별한다. 만약 초기화가 top = 0(다음에 채울 빈 자리를 가리키는 방식)이라면 반대로 push는 후위 증가, pop은 전위 감소가 맞는 경우도 있으므로, 항상 초기 top 값과 첫 push 이후 값이 어디를 가리켜야 하는지부터 직접 손으로 추적해 본다.

암기 포인트

top = -1 초기화 스택은 'push는 ++top으로 먼저 올리고 넣는다, pop은 top--로 먼저 꺼내고 내린다'로 짝지어 기억해 둔다.

NoteBase

문 5. 차수가 4이고 높이가 4인 m원 탐색 트리가 가질 수 있는 노드의 최대 수는? (단, 루트 노드만 있는 경우 트리의 높이는 1이다)

① 21

② 51

③ 85

④ 255

정답 3번

출제 포인트

차수(order)가 4인 m원 탐색 트리에서 각 레벨(depth)마다 가질 수 있는 노드 수의 상한을 이용해, 높이 4인 트리가 가질 수 있는 전체 노드 수의 최댓값을 구하는 문제다. m원 탐색 트리는 하나의 노드가 최대 $m-1$개의 키와 최대 m개의 자식 포인터를 갖는 구조이므로, 각 노드가 자식을 최대로 채웠을 때 레벨별 노드 수가 m의 거듭제곱으로 늘어난다는 점이 핵심이다.

조건 정리

차수 $m=4$이므로 각 노드는 자식을 최대 4개까지 가질 수 있다. 트리의 높이는 4이며, 문제 조건에 따라 루트 노드만 있을 때 높이를 1로 세므로 루트가 1레벨, 그 아래로 2레벨·3레벨·4레벨까지 존재하는 4개 레벨 구조가 된다. 각 레벨의 최대 노드 수는 그 레벨의 모든 노드가 자식을 최대로(4개씩) 채웠을 때 나온다.

사용할 공식

차수 m인 트리에서 레벨 i(루트를 레벨 1로 둘 때)의 최대 노드 수는 m^{i-1}이다. 높이 h인 트리 전체의 최대 노드 수는 각 레벨의 최대 노드 수를 모두 더한 등비수열의 합이므로 $\sum_{i=0}^{h-1} m^i = \frac{m^h-1}{m-1}$로 계산한다.

계산 과정

레벨별 최대 노드 수를 직접 나열하면 1레벨(루트) 1개, 2레벨 $4^1=4$개, 3레벨 $4^2=16$개, 4레벨 $4^3=64$개다. 이를 모두 더하면 $1+4+16+64=85$이다. 공식으로 확인하면 $m=4,\ h=4$일 때 $\frac{4^4-1}{4-1}=\frac{256-1}{3}=\frac{255}{3}=85$로 동일한 값이 나온다. 따라서 이 트리가 가질 수 있는 노드의 최대 수는 85개다.

NoteBase

선지 판단

① 21 → 오답. 이 값은 높이가 4가 아니라 높이 3인 트리의 최대 노드 수($1 + 4 + 16 = 21$)에 해당하며, 높이 4의 마지막 레벨(4레벨의 64개)을 빠뜨린 계산이다.
② 51 → 오답. 레벨별 노드 수를 등비수열이 아닌 다른 방식(예: 일부 레벨을 절반만 채우거나 잘못된 밑수)으로 합산했을 때 나오는 값으로, 각 레벨을 최대로 채운 값이 아니다.
③ 85 → 정답. 1+4+16+64=85, 즉 $\frac{4^4-1}{4-1} = 85$로 차수 4·높이 4인 m원 탐색 트리가 가질 수 있는 노드의 최대 수와 정확히 일치한다.
④ 255 → 오답. $4^4 - 1 = 255$는 분모 $(m - 1) = 3$으로 나누지 않은 값이다. 즉 등비수열의 합 공식에서 나눗셈을 빠뜨린 계산 실수로 나오는 값이다.

오답 포인트

가장 흔한 실수는 등비수열 합 공식의 분모 $(m - 1)$을 빠뜨리고 $m^h - 1$만 계산해 255를 답으로 고르는 것이다. 또한 트리의 높이를 세는 기준(루트만 있을 때 높이 1)을 놓쳐 레벨 수를 하나 적게 잡으면 21이 나온다. 이 문제는 이진 탐색 트리(차수 2)의 노드 수 공식 $2^h - 1$과 혼동하지 않고, 차수가 4인 m원 탐색 트리에 맞는 밑수 4를 정확히 적용하는 것이 중요하다.

다시 나오면 이렇게 판단

문제에서 차수(m)와 높이(h)가 주어지면, 우선 문제가 정의한 높이 기준(루트만 있을 때 높이가 몇인지)을 확인해 레벨 수를 정확히 파악한다. 그다음 레벨별 최대 노드 수를 $1, m, m^2, \ldots, m^{h-1}$로 나열하거나 등비수열 합 공식 $\frac{m^h-1}{m-1}$에 대입해 계산하면 빠르고 정확하게 답을 구할 수 있다.

암기 포인트

차수 m인 m원 탐색 트리, 높이 h일 때 최대 노드 수 공식은 $\frac{m^h-1}{m-1}$로 암기해 둔다. 이진 탐색 트리는 이 공식에서 $m = 2$를 대입한 $2^h - 1$과 같은 형태임을 함께 기억해 두면 헷갈리지 않는다.

NoteBase

문 6. 다음은 이진트리를 전위 순회하여 얻어진 전위표기식이다. 이 트리에 대한 설명으로 옳지 않은 것은?

+ / * 6 2 - 3 1 * + 5 8 - 7 4

① 연산식의 최종 결과는 45이다.

② 트리로 재구성하면 루트 노드는 +이다.

③ 트리로 재구성하여 중위 순회하면 6*2/3-1+5+8*7-4이다.

④ 트리로 재구성하여 후위 순회하면 62*31/-58+74-*+이다.

정답 4번

출제 포인트

전위표기식(prefix expression)으로부터 이진트리를 복원하고, 그 트리를 중위·후위 순회했을 때 나오는 문자열과 연산식의 값을 정확히 계산할 수 있는지 확인하는 문제이다. 전위표기식은 연산자, 왼쪽 부분식, 오른쪽 부분식 순서로 나열되므로, 앞에서부터 하나씩 읽으며 연산자를 만나면 그 아래에 두 개의 하위 트리가 온다고 보고 재귀적으로 트리를 만들어야 한다.

조건 정리

주어진 전위표기식은 다음과 같다.
+ / * 6 2 - 3 1 * + 5 8 - 7 4
전위 순회는 "노드 → 왼쪽 서브트리 → 오른쪽 서브트리" 순서로 방문한 결과이므로, 이 나열 순서를 그대로 재귀 파싱하면 원래의 이진트리를 유일하게 복원할 수 있다.

사용할 공식

전위표기식 파싱 규칙: 토큰을 하나 읽어 그것이 피연산자(숫자)이면 리프 노드로 만들고, 연산자이면 그 노드를 만든 뒤 다시 같은 규칙으로 왼쪽 자식을 먼저 만들고 이어서 오른쪽 자식을 만든다.
중위 순회: 왼쪽 서브트리 → 노드 → 오른쪽 서브트리 순서로 나열한다.
후위 순회: 왼쪽 서브트리 → 오른쪽 서브트리 → 노드 순서로 나열한다.

NoteBase

계산 과정

전위표기식 + / * 6 2 - 3 1 * + 5 8 - 7 4 를 앞에서부터 순서대로 읽어 재귀적으로 트리를 복원한다.

최상위 노드는 첫 토큰인 +이다. 그 왼쪽 서브트리를 만들기 위해 다음 토큰들을 읽는다.
다음 토큰 /가 왼쪽 서브트리의 루트가 되고, 그 아래에서 다시 왼쪽·오른쪽을 읽는다.
/의 왼쪽: 다음 토큰 *이 루트, 그 아래 왼쪽=6, 오른쪽=2 → (6*2)
/의 오른쪽: 다음 토큰 -가 루트, 그 아래 왼쪽=3, 오른쪽=1 → (3-1)
따라서 +의 왼쪽 서브트리는 ((6*2)/(3-1))이다.

이어서 +의 오른쪽 서브트리를 읽는다. 다음 토큰 *이 루트가 되고, 그 아래에서 다시 왼쪽·오른쪽을 읽는다.
*의 왼쪽: 다음 토큰 +가 루트, 그 아래 왼쪽=5, 오른쪽=8 → (5+8)
*의 오른쪽: 다음 토큰 -가 루트, 그 아래 왼쪽=7, 오른쪽=4 → (7-4)
따라서 +의 오른쪽 서브트리는 ((5+8)*(7-4))이다.

전체 트리는 다음과 같다.
루트: +
왼쪽: / (왼쪽: * → 6, 2 / 오른쪽: - → 3, 1)
오른쪽: * (왼쪽: + → 5, 8 / 오른쪽: - → 7, 4)

① 연산식의 최종 값을 계산하면, 왼쪽 서브트리 값 = (6*2)/(3-1) = 12/2 = 6, 오른쪽 서브트리 값 = (5+8)*(7-4) = 13*3 = 39이다. 전체 값 = 6+39 = 45이다. 선지 ①의 45와 일치한다.

② 트리를 재구성했을 때 최상위(루트) 노드는 첫 번째로 읽은 토큰인 +이다. 선지 ②와 일치한다.

③ 중위 순회 결과를 구하면, 왼쪽 서브트리 /(6*2, 3-1)의 중위 순회는 "6*2" + "/" + "3-1" = 6*2/3-1이고, 오른쪽 서브트리 *(5+8, 7-4)의 중위 순회는 "5+8" + "*" + "7-4" = 5+8*7-4이다. 전체 중위 순회는 왼쪽 서브트리 + 루트(+) + 오른쪽 서브트리 = 6*2/3-1 + 5+8*7-4이다. 선지 ③의 6*2/3-1+5+8*7-4와 일치한다.

④ 후위 순회 결과를 구하면, 왼쪽 서브트리 /(6*2, 3-1)의 후위 순회는 "62*" + "31-" + "/" = 62*31-/이고, 오른쪽 서브트리 *(5+8, 7-4)의 후위 순회는 "58+" + "74-" + "*" = 58+74-*이다. 전체 후위 순회는 왼쪽 서브트리 + 오른쪽 서브트리 + 루트(+) = 62*31-/ + 58+74-* + + = 62*31-/58+74-*+이다.
선지 ④에 제시된 문자열은 62*31/-58+74-*+로, 올바른 후위 순회 결과인 62*31-/58+74-*+와 비교하면 다섯 번째~여섯 번째 기호의 순서가 - 와 / 가 뒤바뀌어 있다(정확히는 "31-/" 이어야 할 자리가 "31/-"로 되어 있다).
즉 선지 ④는 올바른 후위 순회 결과가 아니다.

NoteBase

선지 판단

① O — 연산식의 최종 결과는 (6*2/(3-1)) + ((5+8)*(7-4)) = 6+39 = 45로 계산되어 옳은 설명이다.
② O — 전위표기식의 첫 토큰이 +이므로 트리로 재구성했을 때 루트 노드는 +가 되어 옳은 설명이다.
③ O — 트리를 재구성해 중위 순회하면 6*2/3-1+5+8*7-4가 되어 옳은 설명이다.
④ X — 트리를 재구성해 후위 순회하면 62*31-/58+74-*+가 되어야 하는데, 선지에 제시된 62*31/-58+74-*+는 왼쪽 서브트리 부분의 -와 / 순서가 뒤바뀌어 있어 옳지 않은 설명이다.

오답 포인트

후위 순회를 구할 때는 반드시 "왼쪽 서브트리 전체 → 오른쪽 서브트리 전체 → 노드" 순서를 지켜야 하며, 하위 서브트리 안에서도 같은 규칙이 재귀적으로 적용된다. 이 문제에서 /노드의 왼쪽 서브트리(6*2)를 후위 순회하면 62*가 되고 오른쪽 서브트리(3-1)를 후위 순회하면 31-가 되므로, /노드 전체의 후위 순회는 이 둘을 이어 붙인 뒤 마지막에 /를 붙여 62*31-/가 되어야 한다. 이를 62*31/-처럼 서브트리 내부의 연산자 순서를 바꾸어 쓰면 잘못된 결과가 된다. 전위표기식만 보고 직관적으로 순서를 재배열하려 하지 말고, 반드시 트리를 먼저 복원한 뒤 그 트리에서 각 순회 규칙을 순서대로 적용해야 실수를 막을 수 있다.

다시 나오면 이렇게 판단

전위표기식이 주어지면 항상 맨 앞 토큰을 루트로 잡고, 재귀적으로 왼쪽 서브트리와 오른쪽 서브트리를 순서대로 읽어 트리를 완전히 복원한 다음에 순회를 진행한다. 트리를 그림으로 그려 두면 중위·후위 순회 결과를 조합할 때 서브트리 단위로 헷갈리지 않는다. 최종적으로 계산이나 순회 문자열을 구한 뒤에는 각 선지와 한 글자씩 대조하여 순서가 뒤바뀐 곳이 없는지 확인한다.

암기 포인트

전위(preorder) = 노드-왼쪽-오른쪽, 중위(inorder) = 왼쪽-노드-오른쪽, 후위(postorder) = 왼쪽-오른쪽-노드. 세 순회 모두 서브트리 단위로 재귀 적용되며, 하나의 서브트리를 완전히 끝내고 나서 다음 부분으로 넘어간다는 순서를 반드시 지켜야 한다.

NoteBase

문 7. 이진트리의 성질에 대한 설명으로 옳은 것은? (단, 루트 노드의 레벨은 0이고, 차수가 0인 노드의 개수는 n_0, 차수가 1인 노드의 개수는 n_1, 차수가 2인 노드의 개수는 n_2로 한다)

① 이진트리에서 단말노드의 개수를 n_0이라 할 때, n_2와의 관계식 $n_0 = n_2 + 1$을 만족한다.

② 깊이가 k인 이진트리가 가질 수 있는 노드의 최대 개수는 2^{k+1}이다($k \geq 0$).

③ 이진트리의 레벨 j에 있는 노드의 최대 개수는 2^{j-1}개이다($j \geq 0$).

④ 이진트리의 전체 노드 개수는 $n_0 + n_1 + n_2 - 1$개이다.

정답 1번

출제 포인트

이진트리의 노드 개수와 관련된 기본 성질(차수별 노드 개수 관계, 레벨별 최대 노드 개수, 깊이별 최대 노드 개수)을 정확히 알고 있는지 묻는 문항이다. 특히 차수가 0인 노드(단말노드)와 차수가 2인 노드 사이의 관계식을 유도할 수 있어야 한다.

조건 정리

루트 노드의 레벨은 0으로 정의한다. 이진트리 전체 노드 수를 n이라 하면 $n = n_0 + n_1 + n_2$이다. 이진트리는 루트를 제외한 모든 노드가 부모와 자신을 잇는 간선을 정확히 하나씩 가지므로 전체 간선 수는 $n - 1$이다.

사용할 공식

간선 수를 차수 기준으로 다시 세면, 차수가 1인 노드는 자식을 1개 가지므로 간선 1개씩을, 차수가 2인 노드는 자식을 2개 가지므로 간선 2개씩을 만든다. 따라서 전체 간선 수는 $n_1 + 2n_2$로도 표현된다. 두 식을 등호로 연결하면 $n_1 + 2n_2 = n_0 + n_1 + n_2 - 1$이다. 또한 깊이가 k인 포화이진트리(full binary tree)의 전체 노드 최대 개수는 $2^{(k+1)} - 1$이고, 레벨 j에 있는 노드의 최대 개수는 2^j이다.

계산 과정

$n_1 + 2n_2 = n_0 + n_1 + n_2 - 1$에서 양변의 n_1을 소거하면 $2n_2 = n_0 + n_2 - 1$이 되고, 이를 정리하면 $n_2 = n_0 - 1$, 즉 $n_0 = n_2 + 1$이 성립한다. 이는 이진트리의 단말노드 수가 차수 2인 노드 수보다 항상 1개 많다는 성질이며, 트리의 형태(균형/불균형, 완전/불완전)와 무관하게 항상 성립한다.

NoteBase

선지 판단

① 단말노드의 개수를 n_0이라 할 때 $n_0 = n_2 + 1$이 성립함을 위 계산 과정에서 확인했으므로 옳다.
② 깊이가 k인 이진트리가 가질 수 있는 노드의 최대 개수는 $2^{(k+1)}$이 아니라 $2^{(k+1)} - 1$이다 → 틀린 표현 $2^{(k+1)}$ → 올바른 표현 $2^{(k+1)} - 1$.
③ 레벨 j에 있는 노드의 최대 개수는 $2^{(j-1)}$이 아니라 2^j이다(예: 루트가 있는 레벨 0에는 최대 1개, 즉 $2^0 = 1$개가 있어야 하는데 $2^{(0-1)} = 2^{-1}$이 되어 성립하지 않는다) → 틀린 표현 $2^{(j-1)}$ → 올바른 표현 2^j.
④ 이진트리의 전체 노드 개수는 $n_0 + n_1 + n_2 - 1$이 아니라 $n_0 + n_1 + n_2$이다(정의상 모든 노드는 차수 0, 1, 2 중 하나에 속하므로 단순히 셋을 더하면 전체 개수가 된다) → 틀린 표현 $n_0 + n_1 + n_2 - 1$ → 올바른 표현 $n_0 + n_1 + n_2$.

오답 포인트

②는 '깊이 k일 때 전체 노드 최대 개수' 공식에서 -1을 빠뜨리는 실수를 유도한다(등비수열 합 공식 $1 + 2 + 4 + \cdots + 2^k = 2^{(k+1)} - 1$을 떠올려야 한다). ③은 '레벨'과 '깊이'를 혼동하거나 지수 계산에서 1을 잘못 빼는 실수를 유도한다(레벨 j의 최대 노드 수는 2^j이지 $2^{(j-1)}$이 아니다). ④는 간선 수 공식과 노드 수 공식을 혼동하여 불필요하게 -1을 붙이는 실수이다(간선 수는 $n - 1$이지만 노드 수는 그냥 $n_0 + n_1 + n_2$이다).

다시 나오면 이렇게 판단

이진트리 성질 문제에서는 먼저 '노드 수 공식'과 '간선 수 공식'을 분리해서 생각한다. 노드 수는 항상 $n_0 + n_1 + n_2$이고, 간선 수는 항상 노드 수보다 1개 적은 $n - 1$이며 동시에 $n_1 + 2n_2$로도 표현된다. 이 두 식을 연립하면 $n_0 = n_2 + 1$이 바로 유도되므로 외울 필요 없이 그 자리에서 재유도할 수 있다. 또한 '레벨 j의 최대 노드 수 2^j'와 '깊이 k까지의 전체 최대 노드 수 $2^{(k+1)} - 1$'은 서로 다른 개념이므로 절대 혼동하지 않는다.

암기 포인트

이진트리 단말노드 관계식: $n_0 = n_2 + 1$(간선 수 $n_1 + 2n_2 = n - 1$에서 유도). 레벨 j의 최대 노드 수: 2^j. 깊이 k까지의 전체 최대 노드 수: $2^{(k+1)} - 1$. 전체 노드 수: $n_0 + n_1 + n_2$(빼기 없음).

NoteBase

문 8. 다음 C 언어 코드를 실행한 후 5번째 및 7번째 줄에 출력되는 문장으로 옳은 것은?

```c
#include <stdio.h>

void hanoi(int n, char from, char tmp, char to) {
  if( n==1 )
    printf("disk 1 from %c to %c\n", from, to);
  else {
    hanoi(n-1, from, to, tmp);
    printf("disk %d from %c to %c\n", n, from, to);
    hanoi(n-1, tmp, from, to);
  }
}

int main(int argc, char* argv[]) {
  hanoi(3, 'A', 'B', 'C');
}
```

① 5번째 줄: disk 1 from B to A, 7번째 줄: disk 1 from A to C

② 5번째 줄: disk 1 from C to B, 7번째 줄: disk 2 from B to C

③ 5번째 줄: disk 2 from A to B, 7번째 줄: disk 3 from A to C

④ 5번째 줄: disk 3 from A to C, 7번째 줄: disk 2 from B to C

정답 1번

출제 포인트

재귀 함수 hanoi(n, from, tmp, to)의 호출 순서를 정확히 추적해 출력 문장의 순서를 맞히는 문제이다. 재귀 호출이 언제 먼저 실행되고 printf가 언제 실행되는지, 그리고 재귀 호출마다 인자(from, tmp, to)가 어떻게 뒤바뀌어 전달되는지를 정확히 따라가는 것이 핵심이다.

조건 정리

함수 정의: hanoi(int n, char from, char tmp, char to). n==1이면 "disk 1 from %c to %c"(from, to)를 출력하고 종료한다. n>1이면 순서대로 (1) hanoi(n-1, from, to, tmp) 호출 (2) "disk %d from %c to %c"(n, from, to) 출력 (3) hanoi(n-1, tmp, from, to) 호출을 실행한다. main에서 hanoi(3, 'A', 'B', 'C')로 최초 호출하므로 최상위 호출은 n=3, from=A, tmp=B, to=C이다.

NoteBase

사용할 공식

별도의 수식은 없으며, 재귀 호출 시 인자가 자리를 바꿔 전달되는 규칙만 적용하면 된다. hanoi(n-1, from, to, tmp)로 호출하면 새 호출의 (from, tmp, to)는 각각 (원래 from, 원래 to, 원래 tmp)가 되고, hanoi(n-1, tmp, from, to)로 호출하면 새 호출의 (from, tmp, to)는 각각 (원래 tmp, 원래 from, 원래 to)가 된다.

계산 과정

최상위 호출 H(n=3, from=A, tmp=B, to=C)를 기준으로 실행 순서를 그대로 펼치면 다음과 같다.
1) H(3,A,B,C) → 첫 재귀호출 H(2, from=A, to=C, tmp=B) 실행
1-1) H(2,A,C,B) → 첫 재귀호출 H(1, from=A, to=B, tmp=C) 실행
1-1-1) H(1,A,B,C): n==1이므로 출력 → 1번째 줄: disk 1 from A to C
1-2) H(2,A,C,B) 자신의 printf 실행 → 2번째 줄: disk 2 from A to B
1-3) H(2,A,C,B) → 두 번째 재귀호출 H(1, tmp=C, from=A, to=B) 실행
1-3-1) H(1,C,A,B): n==1이므로 출력 → 3번째 줄: disk 1 from C to B
2) H(3,A,B,C) 자신의 printf 실행 → 4번째 줄: disk 3 from A to C
3) H(3,A,B,C) → 두 번째 재귀호출 H(2, from=B, to=C, tmp=A) 실행
3-1) H(2,B,A,C) → 첫 재귀호출 H(1, from=B, to=A, tmp=C) 실행
3-1-1) H(1,B,C,A): n==1이므로 출력 → 5번째 줄: disk 1 from B to A
3-2) H(2,B,A,C) 자신의 printf 실행 → 6번째 줄: disk 2 from B to C
3-3) H(2,B,A,C) → 두 번째 재귀호출 H(1, tmp=A, from=B, to=C) 실행
3-3-1) H(1,A,B,C): n==1이므로 출력 → 7번째 줄: disk 1 from A to C
따라서 전체 출력은 1행부터 순서대로 disk 1 from A to C / disk 2 from A to B / disk 1 from C to B / disk 3 from A to C / disk 1 from B to A / disk 2 from B to C / disk 1 from A to C가 된다. 5번째 줄은 disk 1 from B to A이고, 7번째 줄은 disk 1 from A to C이다.

선지 판단

① 5번째 줄 disk 1 from B to A, 7번째 줄 disk 1 from A to C — 추적 결과와 정확히 일치하므로 옳다.
② 5번째 줄 disk 1 from C to B, 7번째 줄 disk 2 from B to C — 실제 3번째 줄과 6번째 줄의 내용을 각각 5번째, 7번째로 잘못 대응시킨 것이므로 틀리다.
③ 5번째 줄 disk 2 from A to B, 7번째 줄 disk 3 from A to C — 실제 2번째 줄과 4번째 줄의 내용이며 줄 번호가 앞당겨진 오답이므로 틀리다.
④ 5번째 줄 disk 3 from A to C, 7번째 줄 disk 2 from B to C — 실제 4번째 줄과 6번째 줄의 내용이며 줄 번호가 밀린 오답이므로 틀리다.

NoteBase

오답 포인트

재귀 호출 두 번(첫 번째 hanoi 호출과 두 번째 hanoi 호출) 사이에 printf가 끼어 있다는 순서를 놓치면 줄 번호가 어긋난다. 또한 두 번째 재귀호출 hanoi(n-1, tmp, from, to)에서는 인자 이름이 아니라 자리(순서)가 그대로 전달됨을 유의해야 한다. 즉 새 호출의 from 자리에는 원래의 tmp 값이, tmp 자리에는 원래의 from 값이, to 자리에는 원래의 to 값이 들어간다. 이 자리바꿈을 혼동하면 B와 A, C의 위치를 뒤바꿔 잘못된 문자를 쓰게 된다.

다시 나오면 이렇게 판단

hanoi류 재귀 출력 문제는 트리를 직접 그리기보다, 각 호출을 H(n, from, tmp, to) 형태로 적어두고 (첫 재귀호출) → (자신의 출력) → (두 번째 재귀호출) 순서를 기계적으로 펼쳐 가며 번호를 매기면 실수 없이 풀 수 있다. 인자 전달 시에는 항상 변수 이름이 아니라 '몇 번째 자리에 어떤 값이 들어가는지'를 확인한다.

암기 포인트

재귀함수에서 printf가 재귀호출 사이에 있으면, 출력 순서는 (왼쪽 재귀호출의 모든 출력) → (현재 호출의 출력) → (오른쪽 재귀호출의 모든 출력) 순서로 결정된다는 전위/중위 순회와 유사한 구조를 기억해 둔다.

NoteBase

문 9. 다음과 같이 이중연결리스트를 이용하여 큐 자료구조를 정의하였다. add는 큐의 삽입 알고리즘이고, delete는 큐의 삭제 알고리즘이다. ㉠, ㉡에 들어갈 내용을 바르게 연결한 것은?

```
typedef struct _dnode {
  int key;
  struct _dnode *prev, *next;
} dnode;
dnode *head = (dnode*)malloc(sizeof(dnode));
dnode *tail = (dnode*)malloc(sizeof(dnode));
head->prev = head;   head->next = tail;
tail->prev = head;   tail->next = tail;

int add(int k) {
  dnode *t = (dnode*)malloc(sizeof(dnode));
  if (t == NULL) {
    printf("\n Out of memory.");
    return -1;
  }
  t->key = k;
  _________㉠_________
  tail->prev = t;  t->next = tail;
  return k;
}

int delete(void) {
  dnode *t; int i;
  t = head->next;
  if (t == tail) {
    printf("\n Queue underflow.");
    return -1;
  }
  i = t->key;
  _________㉡_________
  free(t);  return i;
}
```

① ㉠ tail->prev = t; t->prev = tail->prev;
㉡ head->next = t->next; t->next->prev = head;

② ㉠ tail->prev->next = t; t->prev = tail->prev->next;
㉡ head->next = t->next; t->next->prev = head;

③ ㉠ tail->prev->next = t; t->prev = tail->prev;
㉡ head->next = t->next; t->next = head;

④ ㉠ tail->prev->next = t; t->prev = tail->prev;
㉡ head->next = t->next; t->next->prev = head;

정답 4번

출제 포인트

head와 tail을 더미 노드로 두는 원형 이중연결리스트로 큐를 구현할 때, 삽입(add)과 삭제(delete)에서 포인터를 어떤 순서로 갱신해야 하는지를 묻는다. 대입문의 실행 순서를 바꾸면 나중 문장이 이미 덮어써진 값을 참조하게 되어 자기 자신을 가리키는 오류나 끊어진 포인터가 생긴다는 점이 핵심이다.

NoteBase

올바른 순서

ㄱ(삽입, 새 노드 t를 tail 앞에 연결): t->key = k; 다음으로 먼저 tail->prev->next = t;를 실행해 현재 마지막 노드(이하 L)의 next가 t를 가리키도록 한다. 이 시점까지는 tail->prev가 아직 L 그대로이므로, 이어서 t->prev = tail->prev;를 실행하면 t->prev가 정확히 L이 된다. 이후 코드에 이미 주어진 tail->prev = t; t->next = tail;로 마무리되어 t가 L과 tail 사이에 온전히 끼워진다.
ㄴ(삭제, head 다음 첫 노드 t 제거): i = t->key;로 값을 먼저 저장한다. 다음으로 head->next = t->next;를 실행해 head가 t 다음 노드(이하 F2)를 가리키게 한다. 이 대입은 t->next 자체를 바꾸지 않으므로, 이어서 t->next->prev = head;를 실행하면 F2->prev가 정확히 head가 된다. 마지막으로 free(t)로 메모리를 해제하고 i를 반환한다.

선지별 해설

① ㄱ: tail->prev = t; t->prev = tail->prev; → 틀린 표현: 첫 문장에서 tail->prev를 곧바로 t로 덮어써 버리므로, 두 번째 문장의 tail->prev는 이미 t 자신이 되어 t->prev = t라는 자기참조 오류가 발생한다 → 올바른 표현: tail->prev->next = t; t->prev = tail->prev; (원래의 tail->prev, 즉 마지막 노드 L을 먼저 이용하고 나중에 tail->prev를 바꾸어야 한다)
② ㄱ: tail->prev->next = t; t->prev = tail->prev->next; → 틀린 표현: 첫 문장 실행 직후 tail->prev->next는 이미 t로 바뀌어 있으므로, 두 번째 문장의 tail->prev->next는 t 자신을 가리켜 t->prev = t라는 자기참조 오류가 발생한다 → 올바른 표현: t->prev = tail->prev; (next를 한 번 더 거치지 말고 tail->prev를 바로 읽어야 한다)
③ ㄱ은 tail->prev->next = t; t->prev = tail->prev;로 옳지만, ㄴ: head->next = t->next; t->next = head;에서 두 번째 문장이 틀렸다 → 틀린 표현: t->next = head;는 삭제되는 노드 t 자신의 next만 head로 바꿀 뿐, 리스트에 남아야 할 다음 노드(F2)의 prev를 head로 갱신하지 못한다. 그 결과 F2->prev는 여전히 곧 free될 t를 가리키는 끊어진 포인터로 남는다 → 올바른 표현: t->next->prev = head;
④ ㄱ: tail->prev->next = t; t->prev = tail->prev; / ㄴ: head->next = t->next; t->next->prev = head; → 둘 다 옳다. 삽입에서는 마지막 노드의 next와 새 노드의 prev를 순서대로 안전하게 연결하고, 삭제에서는 head의 next와 다음 노드의 prev를 순서대로 안전하게 갱신한다. 정답.

오답 포인트

포인터를 대입하는 두 문장의 순서를 바꾸면, 원래 값을 먼저 다른 변수에 옮겨 적기 전에 그 값을 덮어써 버려 이후 문장이 자기 자신을 가리키는 오류가 생긴다(①, ②의 경우). 또한 삭제에서는 head->next와 다음 노드의 prev 두 방향을 모두 갱신해야 하는데, 반대 방향 포인터를 남는 노드가 아니라 삭제되는 노드 자신에 대입해 버리면(③) 남은 노드의 역방향 포인터가 이미 해제된 노드를 계속 가리키는 상태가 된다.

NoteBase

다시 나오면 이렇게 판단

이중연결리스트의 삽입·삭제 코드를 볼 때는 각 대입문을 순서대로 손으로 따라가면서, 어떤 값을 참조로 쓰기 전에 그 값이 이미 다른 문장에 의해 덮어써지지 않았는지를 확인한다. 삽입은 새 노드의 prev·next와 이웃 노드(마지막 노드, tail)의 next·prev까지 4개의 링크가, 삭제는 남는 두 노드(head, 다음 노드)의 next·prev 2개의 링크가 빠짐없이 갱신되는지를 기준으로 판단한다.

암기 포인트

옛 값을 다른 곳에 먼저 옮겨 적은 뒤에만 그 값을 덮어쓴다는 순서 원칙과, 원형 이중연결리스트에서는 삽입·삭제 모두 관련된 앞뒤 링크를 남김없이 갱신해야 리스트의 무결성이 유지된다는 점을 기억해 둔다.

NoteBase

문 10. 다항식에서 하나의 항을 표현하기 위해 다음과 같은 구조체 poly를 정의하였다. 이 구조체를 이용하여 $A(x) = 3x^{14} + 2x^8 + 1$과 $B(x) = 8x^{14} - 4x^9 + 10x^3$을 저장하는 데 필요한 최소의 메모리 용량(byte)은? (단, 정수(int)를 표현하는 데 2byte를 사용하고, 실수(float)를 표현하는 데 4byte를 사용하며, 다항식의 모든 항은 poly 구조체를 이용하여 저장한다)

```
struct poly{
    int exp;
    float coef;
};
```

① 22

② 24

③ 36

④ 38

정답 3번

출제 포인트

다항식의 각 항을 구조체 배열(또는 연결리스트)로 표현할 때, 구조체 하나의 크기와 저장해야 할 항의 개수를 정확히 세어 전체 메모리 용량을 계산할 수 있는지 묻는 문제이다. 구조체 크기 자체는 단순하지만, 다항식에서 계수가 0인 항(즉, 존재하지 않는 차수)을 항으로 세지 않는지가 실수하기 쉬운 지점이다.

조건 정리

구조체 poly는 int exp(지수, 2byte)와 float coef(계수, 4byte)로 구성되므로 항 하나를 저장하는 데 $2 + 4 = 6$byte가 필요하다. $A(x) = 3x^{14} + 2x^8 + 1$은 항이 3개이다(차수 14, 8, 0인 항 각각 하나씩, 상수항 1은 $1 \cdot x^0$이므로 하나의 항으로 취급). $B(x) = 8x^{14} - 4x^9 + 10x^3$도 항이 3개이다(차수 14, 9, 3). 두 다항식 모두 계수가 0인 차수는 저장하지 않으므로, 실제로 저장되는 항의 총 개수는 $3 + 3 = 6$개이다.

사용할 공식

전체 메모리 용량 = (구조체 하나의 크기) × (저장할 항의 총 개수) = (int 크기 + float 크기) × (항의 개수).

NoteBase

계산 과정

구조체 하나의 크기: 2byte(exp) + 4byte(coef) = 6byte.
A(x)의 항 개수: 3개 ($3x^{14}$, $2x^{8}$, 1).
B(x)의 항 개수: 3개 ($8x^{14}$, $-4x^{9}$, $10x^{3}$).
총 항 개수: 3 + 3 = 6개.
필요한 최소 메모리 용량: 6byte × 6 = 36byte.

선지 판단

① 22 → X. 구조체 크기(6byte)와 항 개수(6개)를 곱한 값과 맞지 않는다. 예를 들어 항 개수를 잘못 세거나 exp를 누락하는 등의 계산으로는 22가 나오지만 올바른 계산 결과가 아니다.
② 24 → X. 만약 항 하나의 크기를 4byte(즉 exp를 누락하고 coef만 계산)로 잘못 보고 6항을 곱하면 24가 되므로, exp 필드를 빠뜨린 오답에 해당한다.
③ 36 → O. 구조체 하나의 크기 6byte에 실제 저장되는 항의 총 개수 6개를 곱한 값으로, 계산 과정에서 도출한 값과 일치한다.
④ 38 → X. 6byte와 6개를 곱한 값(36byte)에 오차가 더해진 값으로, 항의 개수나 구조체 크기 중 하나를 잘못 계산했을 때 나올 수 있는 값이다.

오답 포인트

이 문제의 핵심 함정은 두 가지이다. 첫째, 다항식의 차수 범위(0~14차)를 보고 15개의 항이 모두 있다고 착각해 전체 배열 크기로 계산하면 안 되고, 계수가 0이 아닌 항만 저장 대상이 된다. 둘째, 상수항 1(x^{0} 항)도 계수가 0이 아니므로 반드시 하나의 항으로 세어야 하며, 이를 빠뜨리면 항 개수가 하나 줄어 잘못된 값이 나온다.

다시 나오면 이렇게 판단

구조체 기반으로 다항식을 저장하는 문제가 나오면, 먼저 구조체 필드별 크기를 더해 항 하나의 크기를 구하고, 그다음 다항식에서 계수가 0이 아닌 항의 개수를 하나씩 세어 곱하는 순서로 접근한다. 차수의 최댓값이나 범위에 현혹되지 않고, 실제로 표현된 항만 개수에 포함시킨다.

암기 포인트

희소 다항식(sparse polynomial)을 구조체 배열로 표현할 때 필요한 메모리는 (구조체 하나의 크기) × (0이 아닌 항의 개수)로 계산하며, 최고차항 지수의 크기와는 무관하다.

NoteBase

문 11. 다음은 행렬 연산을 하는 C 코드와 transpose함수 호출 후 행렬의 상태이다. ㉠에 들어갈 코드로 옳은 것은?

```
#include <stdio.h>
#define R 3
#define C 3
void transpose(int A[R][C], int B[R][C]) {
  for(int r = 0; r < R; r++)
   for(int c = 0; c < C; c++)
      ㉠
}
int main(int argc, char* argv[]) {
  int array1[R][C] = {{1,2,3},{4,5,6},{7,8,9}};
  int array2[R][C] = {0};
  transpose(array1, array2);
  return 0;
}
transpose함수 호출 후 행렬의 상태:
          1 2 3                    1 4 7
array1:   4 5 6          array2:   2 5 8
          7 8 9                    3 6 9
```

① A[c][r] = B[c][r];

② B[c][r] = A[c][r];

③ A[c][r] = B[r][c];

④ B[c][r] = A[r][c];

정답 4번

출제 포인트

2차원 배열(행렬)을 이용해 전치행렬(transpose)을 만드는 C 코드에서, 이중 반복문 안에 들어갈 배열 원소 대입식을 정확히 채울 수 있는지 묻는다. 반복 변수 r, c의 의미와 배열 첨자의 순서(행, 열)를 정확히 구분하는 것이 핵심이다.

조건 정리

R=3, C=3이고 A는 array1, B는 array2이다. array1은 1행 {1,2,3}, 2행 {4,5,6}, 3행 {7,8,9}로 채워진 3×3 행렬이다. transpose(array1, array2) 호출 후 array2(=B)는 1행 {1,4,7}, 2행 {2,5,8}, 3행 {3,6,9}가 되어야 한다. 즉 B는 A의 행과 열을 서로 바꾼 전치행렬이다. 반복문은 바깥이 r(0~2), 안쪽이 c(0~2)로 도는 이중 for문이다.

NoteBase

사용할 공식

전치행렬의 정의: 원래 행렬 A의 i행 j열 원소 $A[i][j]$는 전치행렬 B에서 j행 i열 자리, 즉 $B[j][i]$로 옮겨진다. 이 문제의 반복 변수로 바꾸면, 바깥 루프의 r을 A의 행 번호로, 안쪽 루프의 c를 A의 열 번호로 사용하고 있으므로 $B[c][r] = A[r][c]$가 되어야 한다.

계산 과정

반복문은 r=0,1,2 각각에 대해 c=0,1,2를 순회한다. r=0일 때 c=0,1,2를 돌면서 A[0][0]=1, A[0][1]=2, A[0][2]=3을 각각 B[0][0], B[1][0], B[2][0]에 넣어야 array2의 1열(세로)이 1,2,3이 되어 결과적으로 array2 1행이 1,4,7이 되는 구조가 맞물린다. 이를 일반화하면 A[r][c]의 값을 B[c][r] 자리에 넣는 식, 즉 B[c][r] = A[r][c];가 되어야 A의 행 값이 B의 열 값으로 옮겨 가면서 전치가 완성된다. 실제로 r=0,c=0→B[0][0]=A[0][0]=1, r=0,c=1→B[1][0]=A[0][1]=2, r=0,c=2→B[2][0]=A[0][2]=3, r=1,c=0→B[0][1]=A[1][0]=4 … 순서로 채워 넣으면 최종 array2가 1행 1,4,7 / 2행 2,5,8 / 3행 3,6,9가 되어 제시된 결과와 일치한다.

선지 판단

① A[c][r] = B[c][r]; → A(원본 array1)에 값을 다시 대입하는 식이라 array1의 내용이 바뀌어 버린다. 문제에서 array1은 호출 후에도 1,2,3/4,5,6/7,8,9 그대로이므로 A를 변경하는 이 식은 틀린 표현이다. → 옳은 표현은 B(array2)에 값을 채워 넣는 식이다.
② B[c][r] = A[c][r]; → 좌변과 우변의 첨자 순서가 같아서 그냥 A를 그대로 복사하는 식이 된다. 이 경우 array2는 array1과 동일한 1,2,3/4,5,6/7,8,9가 되어 버려 실제 결과(1,4,7/2,5,8/3,6,9)와 다르다. → 틀린 표현 B[c][r]=A[c][r] → 올바른 표현 B[c][r]=A[r][c].
③ A[c][r] = B[r][c]; → 이 역시 A에 값을 대입하는 식이라 array1이 변경되며, 우변 B도 아직 초기화 전(0)이므로 결과가 맞지 않는다. → 틀린 표현이다.
④ B[c][r] = A[r][c]; → A의 r행 c열 값을 B의 c행 r열 자리에 옮겨 넣는 식으로, 계산 과정에서 확인한 것과 정확히 일치하며 제시된 array2 결과를 그대로 만들어 낸다. → 옳은 표현이다.

오답 포인트

①과 ③은 좌변이 A(원본 array1)로 되어 있어 원본 배열을 수정하는 식이라는 점에서 바로 제외할 수 있다. 문제 조건상 array1은 호출 후에도 그대로 남아 있어야 하기 때문이다. ②는 좌변 우변 배열은 맞게 B, A로 되어 있지만 첨자 순서를 바꾸지 않아 전치가 아니라 단순 복사가 되어 버리는 것이 함정이다. 반복 변수 r, c의 순서가 A와 B에서 서로 뒤바뀌어야 전치가 이루어진다는 점을 놓치면 ②를 고르기 쉽다.

NoteBase

다시 나오면 이렇게 판단

전치행렬 코드 문제가 나오면 먼저 반복문의 바깥/안쪽 변수(r, c)가 각각 행, 열 중 무엇을 가리키는지부터 정리한다. 그다음 원본 배열 A는 절대 좌변에 오면 안 되고(원본 보존), B의 첨자 순서가 A의 첨자 순서와 뒤바뀌어야(B[c][r]=A[r][c]) 전치가 성립한다는 원칙만 적용하면 빠르게 답을 고를 수 있다. 실제 숫자로 한두 스텝만 대입해 보면 검산이 쉽다.

암기 포인트

전치행렬 대입식은 B[열][행] = A[행][열] 형태, 즉 좌변과 우변의 첨자 순서가 서로 반대로 뒤집힌다는 것을 기억해 둔다. 좌변이 원본 배열이면 원본이 훼손되므로 항상 제외 대상이다.

NoteBase

문 12. 열 우선(column major) 순서로 저장되는 3차원 배열 student[2][3][4]가 있을 때, 첫 번째 원소인 student[0][0][0]의 주소가 1024이면 student[1][2][2]의 주소는? (단, 배열 student의 각 원소의 크기는 24byte이고, 저장 순서는 [0][0][0], · · ·, [0][2][3], [1][0][0], · · ·, [1][2][3] 순서로 저장한다)

① 1168

② 1192

③ 1504

④ 1528

정답 3번

출제 포인트

3차원 배열의 원소 주소 계산 문제이다. 배열 student[2][3][4]는 첫 번째 첨자가 2개의 '페이지'를 나누고, 각 페이지는 3행 4열의 2차원 배열로 구성된다. 이 문제의 핵심은 '열 우선(column major)'이 무엇을 기준으로 적용되는지를 정확히 파악하는 것이다. 저장 순서 [0][0][0],…,[0][2][3],[1][0][0],…,[1][2][3]은 첫 번째 첨자(페이지) 0을 모두 채운 뒤 1로 넘어간다는 블록 경계만 알려주며, 각 페이지 내부의 실제 저장 순서는 열 우선 규칙을 따른다.

조건 정리

배열 크기: student[2][3][4] (첫 번째 첨자 2개, 두 번째 첨자 3개, 세 번째 첨자 4개). 원소 크기: 24byte. 기준 원소: student[0][0][0]의 주소 = 1024. 구할 원소: student[1][2][2]의 주소. 저장 방식: 첫 번째 첨자 기준으로 페이지가 나뉘고, 각 페이지(3행×4열) 내부는 열 우선(같은 열에서 행이 먼저 증가하고, 그 다음 열이 증가)으로 저장된다.

사용할 공식

페이지 안에서 열 우선 저장이므로, 페이지 내 오프셋은 (세 번째 첨자 × 행의 개수) + 두 번째 첨자로 계산된다. 전체 선형 인덱스는 다음과 같다. index(a, b, c) = $a \times (3 \times 4) + c \times 3 + b$ 여기서 a는 첫 번째 첨자, b는 두 번째 첨자(행), c는 세 번째 첨자(열)이다. 주소는 address = base + index(a, b, c) × size 로 구한다.

계산 과정

student[1][2][2]이므로 $a = 1$, $b = 2$, $c = 2$이다. 먼저 페이지 오프셋: $a \times 12 = 1 \times 12 = 12$. 다음 페이지 내부(열 우선) 오프셋: $c \times 3 + b = 2 \times 3 + 2 = 8$. 두 값을 더하면 전체 인덱스는 $12 + 8 = 20$이다. 검산: student[0][0][0]부터 student[0][2][3]까지가 인덱스 0~11(페이지 0, 12개 원소)이고, student[1][0][0]부터 인덱스 12가 시작되므로 일관된다. 주소는 $1024 + 20 \times 24 = 1024 + 480 = 1504$ 이다.

NoteBase

선지 판단

① 1168 → X. 인덱스로 환산하면 (1168 − 1024)/24 = 6에 해당하는 원소로, student[1][2][2]의 인덱스 20과 다르다.
② 1192 → X. 인덱스 (1192 − 1024)/24 = 7에 해당하며 목표 원소의 인덱스와 다르다.
③ 1504 → O. 계산 과정에서 구한 인덱스 20에 정확히 대응하며, 1024 + 20 × 24 = 1504로 student[1][2][2]의 주소와 일치한다.
④ 1528 → X. 인덱스 (1528 − 1024)/24 = 21에 해당하는 원소로, 세 번째 첨자를 하나 더 증가시킨 student[1][2][3]에 해당하며 목표 원소보다 한 칸 뒤이다.

오답 포인트

가장 흔한 실수는 '열 우선'이라는 표현을 무시하고 두 번째 첨자와 세 번째 첨자를 통상적인 행 우선(두 번째 첨자가 느리게, 세 번째 첨자가 빠르게 증가) 방식으로 계산하는 것이다. 이 경우 인덱스가 22로 계산되어 주소 1552가 나오는데, 이는 선지에 존재하지 않는 값이므로 그 자체로 계산 방식이 잘못되었음을 알아챌 단서가 된다. 반대로 열 우선을 첫 번째 첨자에까지 적용해 페이지 순서마저 뒤바꾸면 인덱스가 17이 되어 주소 1432가 나오는데, 이 역시 선지에 없으므로 오답이다. 열 우선은 페이지 내부의 행·열 관계에만 적용되고, 페이지(첫 번째 첨자) 순서 자체는 그대로 0, 1 순으로 진행된다는 점을 구분해야 한다.

다시 나오면 이렇게 판단

3차원 배열 주소 계산 문제를 만나면 먼저 배열을 '첫 번째 첨자로 나뉘는 여러 개의 2차원 페이지'로 단순화한다. 그 다음 문제에서 지정한 저장 방식(행 우선/열 우선)이 페이지 내부의 2차원 배열에 적용되는 규칙임을 확인하고, 페이지 오프셋(첫 번째 첨자 × 페이지당 원소 수)과 페이지 내부 오프셋(열 우선이면 세 번째 첨자 × 행의 개수 + 두 번째 첨자)을 따로 구해 더한다. 계산한 주소가 선지 중 하나와 정확히 일치하는지로 검산하고, 일치하지 않으면 행 우선/열 우선 적용 범위를 다시 점검한다.

암기 포인트

2차원 배열 A[R][C]에서 행 우선 저장의 원소 A[i][j] 오프셋은 $i \times C + j$이고, 열 우선 저장의 오프셋은 $j \times R + i$이다. 3차원 배열은 첫 번째 첨자로 나뉜 페이지 오프셋(첫 번째 첨자 × 한 페이지의 원소 수)에, 페이지 내부 2차원 배열의 행 우선/열 우선 오프셋을 더한 값이라는 구조로 기억해 둔다.

NoteBase

문 13. 다음 C 코드에서 함수 A와 B는 최대 공약수를 구하는 함수이다. 함수 B가 함수 A와 같은 결과를 출력하도록 ㉠, ㉡에 들어갈 내용을 바르게 연결한 것은?

```
int A(int x, int y){
  int r;
  while (y != 0){
    r = x % y;  x = y;  y = r;
  }
  return x;
}

int B(int x, int y) {
  int r;
  if (_____㉠_____) return x;
  else {
    r = x % y;
    return _____㉡_____
  }
}
```

① ㉠: y==0, ㉡: B(y, r);

② ㉠: y==0, ㉡: B(r, y);

③ ㉠: y!=0, ㉡: B(y, r);

④ ㉠: y!=0, ㉡: B(r, y);

정답 1번

출제 포인트

반복문(while)으로 작성된 유클리드 호제법 함수 A를 재귀 함수 B로 바꿀 때, 반복문의 종료 조건이 재귀의 기저 조건(base case)으로, 반복문 한 번 실행 후의 변수 갱신이 재귀 호출의 인자 전달로 각각 대응됨을 이해하는지 확인하는 문항이다.

조건 정리

함수 A는 while (y != 0) 동안 r = x % y; x = y; y = r; 를 반복하다가 y가 0이 되면 반복을 멈추고 x를 반환한다. 함수 B는 이를 재귀로 표현해야 하므로, if (㉠) return x; 는 A의 반복 종료 조건과 같은 상황에서 실행되어야 하고, else 블록의 return ㉡; 은 A의 한 번의 반복(x = y; y = r; 로 갱신 후 다시 조건 검사)에 대응해야 한다.

NoteBase

사용할 공식

A의 반복 갱신 규칙은 r = x % y 를 구한 뒤 (x, y)를 (y, r)로 갱신하는 것이다. 이를 재귀 형태로 쓰면 다음 호출의 인자는 (y, r)이 되어야 한다.

계산 과정

먼저 ㉠을 정한다. A는 y != 0 인 동안 반복하고 y == 0 이 되는 순간 반복을 빠져나와 x를 반환한다. B의 if (㉠) return x; 문도 정확히 이 순간, 즉 y == 0 일 때 x를 반환해야 하므로 ㉠은 y == 0 이다.
다음으로 ㉡을 정한다. A의 반복문 한 번을 풀어 쓰면 r = x % y; 를 구한 뒤 x는 이전의 y 값으로, y는 방금 구한 r 값으로 갱신되고, 그 상태로 다시 while 조건 검사(사실상 다음 단계의 A(y, r) 호출)로 넘어간다. B는 else 블록에서 이미 r = x % y; 를 계산해 두었으므로, 다음 단계로 넘어가는 재귀 호출은 새 x 자리에 y를, 새 y 자리에 r을 넣은 B(y, r) 이어야 한다. 따라서 ㉡은 B(y, r) 이다.

선지 판단

① ㉠: y==0, ㉡: B(y, r) — 기저 조건과 재귀 호출 인자 순서가 모두 A의 동작과 일치한다. (O)
② ㉠: y==0, ㉡: B(r, y) — 기저 조건은 맞지만 재귀 호출의 인자 순서가 뒤바뀌어 있다. B(r, y)로 호출하면 다음 단계에서 x 자리에 r이, y 자리에 원래의 y가 들어가 A의 갱신 순서(x=y, y=r)와 어긋난다. (X)
③ ㉠: y!=0, ㉡: B(y, r) — 인자 순서는 맞지만 기저 조건이 반대이다. y != 0 일 때 return x를 하면 아직 나눗셈이 끝나지 않은 상태에서 결과를 반환해 버려 잘못된 값을 돌려준다. (X)
④ ㉠: y!=0, ㉡: B(r, y) — 기저 조건과 인자 순서가 모두 반대이다. (X)

오답 포인트

이 문제에서 자주 하는 실수는 두 가지이다. 하나는 if 문의 조건을 반복문의 진행 조건(y != 0)과 그대로 같다고 착각하는 것인데, if는 반복문을 빠져나오는 종료 조건(y == 0)에 대응해야 한다. 다른 하나는 재귀 호출의 인자 순서를 x, y 갱신 순서와 반대로 쓰는 것으로, A에서 x = y; y = r; 로 갱신되므로 재귀 호출도 반드시 B(y, r) 순서여야 하며 B(r, y)는 오답이다.

다시 나오면 이렇게 판단

반복문을 재귀로 바꾸는 문제가 나오면, 먼저 반복문의 종료 조건을 그대로 재귀의 기저 조건(반환 조건)으로 옮기고, 그다음 반복문 한 번의 내부 갱신 순서를 그대로 재귀 호출의 인자 순서로 옮긴다고 판단한다. 조건의 부정(!= ↔ ==)과 인자의 순서를 각각 따로 점검하면 헷갈리지 않는다.

암기 포인트

while(조건) 반복 → 재귀에서는 조건의 반대가 종료(반환) 조건이 되고, 반복문 내부의 변수 갱신 순서 그대로가 재귀 호출의 인자 순서가 된다는 대응 관계를 기억해 둔다.

NoteBase

문 14. C 언어에서 배열과 연결리스트에 대한 설명으로 옳지 않은 것은?

① 일반적으로 두 개의 정수형 데이터 세트 {1,2,3}과 {4,5,6,7,8}을 합병하여 {1,2,3,4,5,6,7,8}로 만드는 연산을 수행할 경우, 두 데이터 세트의 자료구조를 연결리스트로 만들면, 정수형 배열로 만드는 것보다 시간 복잡도를 낮게 구현할 수 있다.

② 일반적으로 정수형 데이터 세트 {1,2,3,4}를 {4,1,2,3}으로 변경하는 연산을 수행할 경우, 해당 데이터 세트의 자료구조를 연결리스트로 만들면, 정수형 배열로 만드는 것보다 더 빠르게 수행시킬 수 있다.

③ 일반적으로 정수형 데이터 세트 {1,2,3,4}를 {2,3,4,5}로 각 요소에 +1 연산을 수행할 경우, 데이터 세트의 자료구조를 연결리스트로 만들면, 정수형 배열로 만드는 것보다 메모리를 적게 차지한다.

④ 일반적으로 정수형 데이터 세트 {1,2,3,4}를 {0,1,2,3,4}로 새로운 값을 추가하는 연산을 수행할 경우, 데이터 세트의 자료구조를 연결리스트로 만들면, 정수형 배열로 만드는 것보다 더 빠르게 연산을 수행시킬 수 있다.

정답 3번

출제 포인트

배열과 연결리스트가 각각 어떤 연산에서 유리한지를 시간 복잡도와 공간(메모리) 복잡도 두 축으로 구분해서 아는지 묻는 문제다. 특히 '자료구조를 바꾸는(원소 추가·삭제·순서 변경) 연산'과 '기존 원소의 값만 갱신하는 연산'을 구분하는 것이 핵심이다.

핵심 개념

배열은 각 원소가 메모리에 연속으로 저장되어 인덱스로 즉시 접근할 수 있지만, 맨 앞이나 중간에 원소를 삽입·삭제하거나 두 배열을 합치려면 뒤따르는 원소들을 한 칸씩 옮겨야 하므로 시간 복잡도가 $O(n)$이다. 연결리스트는 각 노드가 다음 노드를 가리키는 포인터(참조)를 함께 저장하는 구조라서, 맨 앞에 노드를 추가하거나 두 리스트를 이어 붙일 때는 포인터 몇 개만 바꾸면 되므로 $O(1)$로 처리된다. 다만 연결리스트는 각 노드마다 데이터 값 외에 다음 노드를 가리키는 포인터를 추가로 저장해야 하므로, 같은 개수의 값을 저장할 때 배열보다 노드 하나당 메모리를 더 차지한다. 반대로 이미 존재하는 각 원소의 값만 바꾸는 연산(예: 모든 원소에 +1)은 배열이든 연결리스트든 전체 원소를 한 번씩 방문해 값을 덮어써야 하므로 둘 다 시간 복잡도는 $O(n)$으로 같고, 이 경우 자료구조를 바꾼다고 해서 메모리 사용량이 줄어들 이유가 없다.

NoteBase

선지별 해설

① O - {1,2,3}과 {4,5,6,7,8}을 합쳐 {1,2,3,4,5,6,7,8}로 만드는 것은 두 데이터 세트를 하나로 잇는 연산이다. 연결리스트라면 첫 번째 리스트의 마지막 노드가 두 번째 리스트의 첫 노드를 가리키게만 하면 되어 $O(1)$에 끝난다. 배열이라면 두 배열을 담을 새 공간을 만들고 모든 원소를 복사해야 하므로 $O(n)$이 걸린다. 따라서 연결리스트로 만들면 시간 복잡도를 더 낮게 구현할 수 있다는 설명은 옳다.
② O - {1,2,3,4}를 {4,1,2,3}으로 바꾸는 것은 새 값 4를 맨 앞에 넣고 기존 원소들을 한 칸씩 뒤로 미는 것과 같은 결과다. 연결리스트는 새 노드 하나를 만들어 머리 포인터만 그 노드로 바꾸면 되어 $O(1)$이다. 배열은 기존 원소를 전부 한 칸씩 뒤로 옮겨야 하므로 $O(n)$이다. 따라서 연결리스트가 더 빠르다는 설명은 옳다.
③ X - {1,2,3,4}를 {2,3,4,5}로 만드는 것은 원소의 개수나 순서를 바꾸는 것이 아니라, 이미 존재하는 각 원소의 값에 1을 더하는 갱신 연산이다. 이 연산은 배열이든 연결리스트든 모든 원소를 한 번씩 방문해서 값을 고쳐 써야 하므로 시간 복잡도는 둘 다 $O(n)$으로 같고, 자료구조를 연결리스트로 바꾼다고 해서 이 연산 자체의 메모리 사용량이 줄어들 이유가 없다. 오히려 연결리스트는 각 노드마다 다음 노드를 가리키는 포인터를 추가로 저장해야 하므로, 같은 개수의 정수를 담을 때 배열보다 메모리를 더 차지한다. → '메모리를 적게 차지한다'는 '메모리를 더 차지한다'로 고쳐야 옳은 설명이 된다.
④ O - {1,2,3,4}를 {0,1,2,3,4}로 만드는 것은 새 값 0을 맨 앞에 추가하는 연산이다. 연결리스트는 새 노드를 만들어 머리 포인터만 갱신하면 되어 $O(1)$이고, 배열은 기존 원소를 전부 한 칸씩 뒤로 밀어야 하므로 $O(n)$이다. 따라서 연결리스트가 더 빠르게 수행된다는 설명은 옳다.

오답 포인트

①②④에 공통으로 등장하는 상황(합병, 맨 앞 삽입, 새 값 추가)은 모두 원소의 개수나 위치가 바뀌는 '구조 변경' 연산이라 연결리스트가 유리하지만, ③은 원소 개수·순서 변화 없이 값만 갱신하는 연산이라는 점에서 성격이 다르다. 이 차이를 놓치고 '연결리스트=항상 빠르고 메모리도 적게 든다'고 단순 암기하면 ③을 옳은 설명으로 착각하기 쉽다. 또한 연결리스트가 시간 면에서 유리한 경우가 많다고 해서 메모리 면에서도 항상 유리한 것은 아니라는 점(포인터 저장 오버헤드)을 구분해야 한다.

다시 나오면 이렇게 판단

선지에서 다루는 연산이 '원소의 개수나 위치를 바꾸는 연산(삽입·삭제·병합)'인지, 아니면 '기존 원소의 값만 그대로 갱신하는 연산'인지부터 구분한다. 앞의 경우는 연결리스트가 포인터 조작만으로 $O(1)$에 처리되어 배열의 $O(n)$보다 유리하다고 판단하고, 뒤의 경우는 두 자료구조 모두 전체 원소를 순회해야 하므로 시간 복잡도가 같고, 메모리 면에서는 오히려 연결리스트가 포인터만큼 더 든다고 판단한다.

암기 포인트

연결리스트가 유리한 것은 맨 앞/중간 삽입·삭제·병합처럼 구조가 바뀌는 연산의 시간 복잡도(포인터만 바꾸면 되어 $O(1)$)이지, 이미 있는 값을 전부 훑어 갱신하는 연산의 메모리 사용량이 아니다. 연결리스트는 노드마다 포인터를 추가로 저장하므로 같은 데이터를 담을 때 배열보다 메모리를 더 쓴다는 점을 기억해 둔다.

NoteBase

문 15. 다음은 기수정렬 알고리즘이다. ㉠이 두 번 실행된 후의 결과로 옳은 것은?

〈조 건〉
○ LSD(Least Significant Digit)는 가장 낮은 자릿수이고, MSD(Most Significant Digit)는 가장 높은 자릿수이다. 135의 경우 LSD 위치의 값은 5이고 MSD 위치의 값은 1이다.
○ 버킷은 10개 큐로 구현한다.
○ list는 배열로 구현하며 초깃값은 (437, 125, 85, 7, 632, 194)이고 n은 6이다.

```
RadixSort(list, n) {
  for(d ← LSD의 위치에서 MSD위치로 한 자리씩 이동) {
    for(i ← list의 0번째 원소부터 n-1까지 반복) {
      list의 i번째 원소를 d의 위칫값에 따라 0번부터
      9번 사이의 버킷에 삽입한다.
    }
    for(b ← 0번 버킷부터 9번 버킷까지 반복) {
      b번째 버킷에서 원소들을 순차적으로 읽어서 list에
      재배정한다.
    }
    ㉠list의 원소를 순서대로 출력한다.
  }
}
```

① 7 85 125 194 437 632
② 7 125 632 437 85 194
③ 632 194 125 85 437 7
④ 632 194 125 437 85 7

정답 2번

출제 포인트

LSD 기수정렬(radix sort)의 한 자리(pass)를 직접 손으로 추적하는 문제이다. 버킷을 큐(FIFO)로 구현하므로, 같은 버킷에 들어간 원소들은 넣은 순서 그대로 꺼내야 하며, 버킷을 읽어오는 순서는 항상 0번 버킷부터 9번 버킷까지이다. ㉠이 두 번 실행되었다는 것은 for(d ← LSD에서 MSD로) 반복문이 두 바퀴 돌아 1의 자리 기준 정렬 1회, 10의 자리 기준 정렬 1회가 끝난 시점의 list 상태를 묻는 것이다.

NoteBase

조건 정리

초깃값 list = (437, 125, 85, 7, 632, 194), n = 6이다. 모든 수를 3자리로 보면 437, 125, 085, 007, 632, 194이며, 1의 자리(LSD)부터 시작해 10의 자리, 100의 자리(MSD) 순으로 자리를 옮겨가며 버킷에 분배한다. 버킷은 0번부터 9번까지 10개이고 각각 큐이므로 먼저 들어간 원소가 먼저 나온다.

사용할 공식

한 자리(pass)의 처리는 다음과 같다. i를 0부터 n-1까지 순서대로 돌면서 list[i]의 현재 자리 값 d에 해당하는 버킷(0~9번 중 하나)의 뒤쪽에 그 원소를 밀어 넣는다(enqueue). 이어서 b를 0번 버킷부터 9번 버킷까지 순서대로 돌면서 각 버킷의 앞쪽 원소부터 꺼내어(dequeue) list에 순서대로 다시 채운다. 이 두 단계가 끝난 뒤의 list가 그 pass의 출력이다.

계산 과정

1회차(㉠ 첫 번째 실행, 1의 자리 기준):
437의 1의 자리는 7, 125는 5, 85는 5, 7은 7, 632는 2, 194는 4이다.
i 순서대로 넣으면 버킷2에 632, 버킷4에 194, 버킷5에 125→85 순서로, 버킷7에 437→7 순서로 쌓인다.
버킷을 0번부터 9번까지 순서대로 꺼내면 632, 194, 125, 85, 437, 7이 되고, 이것이 ㉠이 첫 번째로 실행될 때 출력되는 list이다.

2회차(㉠ 두 번째 실행, 10의 자리 기준, 입력은 1회차 결과인 632, 194, 125, 85, 437, 7):
632의 10의 자리는 3, 194는 9, 125는 2, 85는 8(085이므로 10의 자리는 8), 437은 3, 7은 0(007이므로 10의 자리는 0)이다.
i 순서대로 넣으면 버킷0에 7, 버킷2에 125, 버킷3에 632→437 순서로, 버킷8에 85, 버킷9에 194가 쌓인다.
버킷을 0번부터 9번까지 순서대로 꺼내면 7, 125, 632, 437, 85, 194가 되고, 이것이 ㉠이 두 번째로 실행될 때 출력되는 list이다.

선지 판단

① 7 85 125 194 437 632 → 완전히 오름차순으로 정렬된 최종 결과이다. 이 문제는 자릿수가 최대 3자리이므로 100의 자리까지 처리해야 완전 정렬이 되는데, ㉠은 두 번째 실행(10의 자리까지만 처리)이므로 아직 완전 정렬 상태가 아니다. 틀린 표현 → 완전 정렬은 100의 자리(3회차)까지 끝나야 나오는 결과.
② 7 125 632 437 85 194 → 계산 과정에서 구한 2회차(10의 자리 기준) 결과와 정확히 일치한다. 정답.
③ 632 194 125 85 437 7 → 이것은 1회차(1의 자리 기준)만 처리된 결과이다. 틀린 표현 → 이는 ㉠의 첫 번째 실행 결과이지 두 번째 실행 결과가 아니다.
④ 632 194 125 437 85 7 → 1회차 결과와 순서가 다르다. 1회차에서는 버킷5(125, 85)가 버킷7(437, 7)보다 먼저 읽히므로 125, 85, 437, 7 순서가 되어야 하는데, 이 선지는 125, 437, 85, 7 순서로 되어 있어 버킷 순서와 맞지 않는다. 틀린 표현 → 버킷을 읽는 순서(0번→9번)와 버킷 내 삽입 순서를 혼동한 결과이다.

NoteBase

오답 포인트

가장 흔한 실수는 ㉠이 두 번 실행된 시점을 최종 정렬 완료 시점으로 착각하는 것이다(①). 이 문제의 수는 최대 3자리이므로 자리(pass)가 총 3번 필요하고, 두 번째 실행은 10의 자리까지만 처리된 중간 상태이다. 두 번째로 흔한 실수는 1회차 결과와 2회차 결과를 헷갈리는 것이다(③은 1회차 결과). 또한 같은 버킷에 여러 원소가 들어갔을 때 큐이므로 먼저 넣은 원소가 먼저 나온다는 점, 그리고 버킷을 읽는 순서는 항상 버킷 번호 0부터 9까지라는 점을 놓치면 원소 순서를 잘못 계산하게 된다(④).

다시 나오면 이렇게 판단

기수정렬 문제는 표를 그려 자리별로 각 수를 버킷 번호에 대응시키고, 같은 버킷에 여러 수가 들어가면 입력 순서대로 세로로 적어 두면 실수가 줄어든다. 버킷을 읽어올 때는 항상 0번부터 9번까지 순서대로, 각 버킷 안에서는 먼저 넣은 것부터(큐) 꺼낸다는 두 규칙만 지키면 어떤 자릿수, 어떤 pass 횟수를 묻든 동일한 방식으로 풀 수 있다.

암기 포인트

LSD 기수정렬은 낮은 자리부터 높은 자리 순으로 자릿수만큼 pass를 반복하며, 버킷은 큐(FIFO)로 구현해 안정 정렬(stable sort)을 보장한다. pass 횟수는 최댓값의 자릿수와 같으며, 이 문제처럼 3자리 수면 총 3번의 pass가 있어야 완전히 정렬된다.

NoteBase

문 16. 데이터 <1, 3, 5, 7, 8, 10, 9>를 차례대로 하나씩 입력받아 이진 탐색 트리와 AVL 트리를 각각 만들었을 때, 두 트리 간의 높이 차는?

① 1

② 2

③ 3

④ 4

정답 4번

출제 포인트

같은 입력 순서라도 이진 탐색 트리(BST)는 삽입 순서에 따라 한쪽으로 치우친 트리가 될 수 있는 반면, AVL 트리는 삽입 때마다 회전으로 균형을 맞추어 높이가 항상 로그 수준으로 유지된다는 점을 확인하는 문제이다. 두 트리를 각각 직접 구성해 높이를 세고 그 차이를 구하면 된다.

조건 정리

입력 순서는 1, 3, 5, 7, 8, 10, 9이며 이 순서 그대로 이진 탐색 트리와 AVL 트리에 각각 삽입한다. 높이는 루트를 0으로 놓고 가장 깊은 노드까지의 간선 수(레벨 수)로 센다. 이 기준을 두 트리에 동일하게 적용하면, 채택하는 높이 정의(간선 수 기준이든 노드 수 기준이든)와 무관하게 두 트리의 높이 차이 값은 같다.

사용할 공식

이진 탐색 트리 삽입: 새 값이 현재 노드보다 작으면 왼쪽, 크면 오른쪽으로 내려가며 빈 자리에 매단다(회전 없음). AVL 트리 삽입: 이진 탐색 트리와 같은 방식으로 삽입한 뒤, 루트 방향으로 올라가며 각 노드의 균형 인수(왼쪽 부분트리 높이 − 오른쪽 부분트리 높이)를 검사해 그 값이 -2 또는 $+2$가 되면 LL, RR, LR, RL 중 해당하는 회전을 적용해 균형을 다시 맞춘다.

NoteBase

계산 과정

[이진 탐색 트리] 1을 루트로 놓고 3, 5, 7, 8, 10을 차례로 삽입하면 모두 직전 노드보다 크므로 오른쪽으로만 이어져 1-3-5-7-8-10의 한 줄짜리 사슬이 만들어진다. 마지막으로 9를 삽입하면 9는 10보다 작으므로 10의 왼쪽 자식이 된다. 따라서 최종 형태는 1의 오른쪽에 3, 그 오른쪽에 5, 7, 8, 10이 차례로 이어지고 10의 왼쪽 자식이 9인 구조이며, 루트에서 9까지 내려가는 데 간선이 6개이므로 이 트리의 높이는 6이다.

[AVL 트리] 1, 3을 넣으면 1-3의 오른쪽 사슬(균형 인수 정상)이 된다. 5를 넣으면 1-3-5로 오른쪽으로 치우쳐 균형 인수가 무너지므로 RR 회전(1을 기준으로 왼쪽 회전)을 적용해 3을 새 루트로, 1을 왼쪽 자식, 5를 오른쪽 자식으로 놓는다. 7을 넣으면 5의 오른쪽 자식이 되어 3(1, 5(-,7)) 형태가 되고 아직 균형 인수 범위 안이라 회전이 없다. 8을 넣으면 7의 오른쪽에 붙으며 노드 5의 균형이 무너지므로(오른쪽으로 2 치우침, RR 형태) 5를 기준으로 왼쪽 회전을 하여 7을 새 부분트리 루트로, 5를 왼쪽 자식, 8을 오른쪽 자식으로 만든다. 이 결과 트리 전체는 3(1, 7(5, 8))이 된다. 10을 넣으면 8의 오른쪽에 붙는데, 이때 루트 3의 균형 인수가 무너진다(오른쪽 부분트리인 7 쪽이 왼쪽보다 2 높음, 그 안에서도 오른쪽 자식 8이 무거운 RR 형태)이므로 3을 기준으로 왼쪽 회전을 하여 7이 전체 루트가 되고, 3(1, 5)이 왼쪽 부분트리, 8(-, 10)이 오른쪽 부분트리가 된다. 마지막으로 9를 넣으면 9는 7보다 크고 8보다 크고 10보다 작으므로 10의 왼쪽 자식이 되어 노드 8의 균형이 무너진다(오른쪽 자식 10이 왼쪽으로 치우친 형태, 즉 RL 형태). 이 경우 먼저 10을 기준으로 오른쪽 회전을 해 9를 8의 오른쪽 자리로 올린 뒤(9는 왼쪽이 비고 오른쪽이 10), 다시 8을 기준으로 왼쪽 회전을 하면 9가 새 부분트리 루트가 되어 왼쪽 자식이 8, 오른쪽 자식이 10인 형태로 정리된다. 최종적으로 트리는 루트 7, 왼쪽 부분트리 3(1, 5), 오른쪽 부분트리 9(8, 10)의 완전히 균형 잡힌 모양이 되며, 루트에서 가장 깊은 노드까지 간선이 2개이므로 이 트리의 높이는 2이다.

선지 판단

① 1 — X. 두 트리의 높이 차는 6과 2의 차이인 4이며 1이 아니다.
② 2 — X. AVL 트리 자체의 높이가 2이지 두 트리의 차가 2인 것이 아니다.
③ 3 — X. 어떤 기준으로 세어도 두 트리의 높이 차가 3이 되지는 않는다.
④ 4 — O. 이진 탐색 트리의 높이 6에서 AVL 트리의 높이 2를 빼면 4이므로 이것이 두 트리 간의 높이 차이다.

오답 포인트

이 문제에서 가장 흔한 실수는 이진 탐색 트리를 구성할 때 9를 10의 오른쪽에 놓는 것이다. 입력 순서가 …, 10, 9이므로 9는 반드시 마지막에 10과 비교되어 10보다 작다는 이유로 10의 왼쪽 자식이 되어야 하며, 이 위치가 바로 이진 탐색 트리 전체에서 가장 깊은 자리(간선 6개 깊이)가 된다. 또 AVL 트리를 구성할 때 8 다음에 10을 넣은 직후에 회전을 빠뜨리면(루트 3의 균형이 무너진 상태를 그대로 두면) 높이를 3으로 잘못 세게 되므로, 노드를 하나 삽입할 때마다 삽입 경로에 있는 모든 조상 노드의 균형 인수를 반드시 확인해야 한다.

NoteBase

다시 나오면 이렇게 판단

오름차순이나 특정 순서로 계속 커지는 값들이 연이어 들어오는 입력 수열이 주어지면, 이진 탐색 트리는 회전이 없으므로 그 값들이 들어온 순서 그대로 한쪽으로 이어진 사슬이 되어 높이가 커지고, AVL 트리는 그때마다 회전으로 재조정되어 균형 잡힌 모양을 유지한다는 점부터 확인한다. 이후 두 트리를 각각 끝까지 그려 본 뒤 같은 기준(간선 수)으로 높이를 세어 차이를 계산하면 된다.

암기 포인트

이진 탐색 트리는 삽입 순서가 그대로 모양을 결정하며 최악의 경우(정렬된 순서로 계속 삽입) 높이가 노드 수만큼 커질 수 있다. AVL 트리는 삽입할 때마다 불균형이 생기는 즉시 LL, RR, LR, RL 중 하나의 회전으로 균형 인수를 $-1, 0, 1$ 범위로 되돌리므로 높이가 항상 노드 수의 로그에 비례하게 유지된다.

NoteBase

문 17. 초기 빈 상태의 트리에 다섯 개의 키 <1, 5, 3, 2, 6>를 차례대로 입력받아 디지털 탐색 트리를 만든 후, 전위 순회한 결과로 옳은 것은? (단, 각 키는 3비트 이진수로 표현한다)

① 1, 3, 2, 5, 6

② 1, 3, 5, 2, 6

③ 1, 5, 2, 6, 3

④ 1, 5, 3, 2, 6

정답 1번

출제 포인트

디지털 탐색 트리(digital search tree)는 이진 탐색 트리와 달리 키 값의 크기를 직접 비교하지 않고, 키를 이진수로 표현했을 때의 비트값에 따라 삽입 위치의 좌우 방향을 결정하는 트리이다. 이 문제는 주어진 삽입 순서대로 트리를 구성한 뒤 전위 순회(root → left → right) 결과를 구하는 문제이다.

조건 정리

키는 3비트 이진수로 표현한다. 1=001, 5=101, 3=011, 2=010, 6=110이다. 삽입 순서는 1, 5, 3, 2, 6이며, 각 키는 최상위 비트(첫째 자리)부터 순서대로 사용해 방향을 정한다.

사용할 공식

디지털 탐색 트리의 삽입 규칙은 다음과 같다. 트리가 비어 있으면 그 자리에 키를 저장한다. 이미 노드가 있으면 삽입할 키의 비트를 최상위 비트부터 차례로 확인하여, 비트가 0이면 왼쪽으로, 1이면 오른쪽으로 내려가고, 내려간 위치가 비어 있을 때까지 이 과정을 반복한 뒤 그 빈 자리에 키를 저장한다.

NoteBase

계산 과정

1(001) 삽입: 트리가 비어 있으므로 루트 자리에 1을 저장한다.
5(101) 삽입: 루트(1)가 차 있으므로 5의 첫째 비트 1을 보고 오른쪽으로 이동한다. 오른쪽이 비어 있으므로 그 자리에 5를 저장한다. (루트 1의 오른쪽 자식 = 5)
3(011) 삽입: 루트가 차 있으므로 3의 첫째 비트 0을 보고 왼쪽으로 이동한다. 왼쪽이 비어 있으므로 그 자리에 3을 저장한다. (루트 1의 왼쪽 자식 = 3)
2(010) 삽입: 루트가 차 있으므로 2의 첫째 비트 0을 보고 왼쪽(노드 3)으로 이동한다. 노드 3이 차 있으므로 2의 둘째 비트 1을 보고 오른쪽으로 이동한다. 그 자리가 비어 있으므로 2를 저장한다. (노드 3의 오른쪽 자식 = 2)
6(110) 삽입: 루트가 차 있으므로 6의 첫째 비트 1을 보고 오른쪽(노드 5)으로 이동한다. 노드 5가 차 있으므로 6의 둘째 비트 1을 보고 오른쪽으로 이동한다. 그 자리가 비어 있으므로 6을 저장한다. (노드 5의 오른쪽 자식 = 6)
완성된 트리 구조: 루트 = 1, 1의 왼쪽 = 3, 1의 오른쪽 = 5, 3의 오른쪽 = 2, 5의 오른쪽 = 6.
전위 순회(루트 → 왼쪽 서브트리 → 오른쪽 서브트리)를 적용하면, 루트 1을 방문하고, 왼쪽 서브트리(3, 그리고 3의 오른쪽 자식 2)를 방문해 3, 2를 얻고, 오른쪽 서브트리(5, 그리고 5의 오른쪽 자식 6)를 방문해 5, 6을 얻는다. 따라서 전위 순회 결과는 1, 3, 2, 5, 6이다.

선지 판단

① 1, 3, 2, 5, 6 → O. 계산 과정에서 구한 전위 순회 결과와 정확히 일치한다.
② 1, 3, 5, 2, 6 → X. 왼쪽 서브트리를 먼저 전부 방문해야 하는데 왼쪽 서브트리의 2보다 오른쪽 서브트리의 5를 먼저 방문한 순서이므로 전위 순회 규칙에 어긋난다.
③ 1, 5, 2, 6, 3 → X. 루트의 오른쪽 서브트리(5, 6)를 왼쪽 서브트리(3, 2)보다 먼저 방문한 순서이므로 왼쪽을 오른쪽보다 먼저 방문하는 전위 순회 규칙에 어긋난다.
④ 1, 5, 3, 2, 6 → X. 이는 삽입 순서 그대로를 나열한 것으로 실제 트리 구조를 전위 순회한 결과가 아니다.

오답 포인트

가장 흔한 실수는 디지털 탐색 트리를 이진 탐색 트리처럼 키 값의 크기 비교로 삽입한다고 착각하는 것이다. 디지털 탐색 트리는 크기 비교가 아니라 이진수로 표현한 키의 비트값(0이면 왼쪽, 1이면 오른쪽)으로 삽입 위치를 정한다는 점을 놓치면 ②나 ③처럼 잘못된 트리 구조를 만들게 된다. 또한 전위 순회를 삽입 순서와 혼동하면 ④를 고르는 오류가 생긴다.

다시 나오면 이렇게 판단

먼저 각 키를 문제에서 지정한 비트 수의 이진수로 변환한다. 이후 삽입할 때마다 루트부터 시작해 이미 노드가 있으면 현재 삽입 중인 키의 비트를 최상위 비트부터 하나씩 확인해 0이면 왼쪽, 1이면 오른쪽으로 내려가고, 빈 자리가 나오면 그 자리에 저장한다. 트리가 완성되면 전위(루트-왼쪽-오른쪽), 중위(왼쪽-루트-오른쪽), 후위(왼쪽-오른쪽-루트) 중 문제가 요구하는 순회 규칙에 맞게 정리한다.

NoteBase

암기 포인트

디지털 탐색 트리는 키의 크기가 아니라 이진수 비트값으로 좌(0)/우(1) 방향을 정해 삽입한다. 트라이와 달리 키를 리프뿐 아니라 내부 노드에도 저장한다. 전위 순회는 항상 루트를 먼저 방문한 뒤 왼쪽 서브트리 전체, 그다음 오른쪽 서브트리 전체를 방문한다는 순서를 지킨다.

NoteBase

문 18. 입력받은 수식의 괄호쌍이 맞는지 스택 S를 이용하여 판단하는 알고리즘 DDD를 작성하였다. 수식에 존재하는 괄호는 소괄호 '()'이며, 다른 괄호는 수식에 존재하지 않는다. 수식의 길이는 n이며, 배열 X에 저장되어 입력된다. 알고리즘 DDD는 X를 입력받아 X에 저장된 수식의 괄호쌍이 맞으면 true, 그렇지 않으면 false를 반환하는 함수이다. ㉠~㉢에 들어갈 코드를 바르게 연결한 것은? (단, S.push(c)는 스택 S에 문자 c를 삽입하는 연산, S.pop()은 S에서 삭제하는 연산, S.empty()는 S가 비어 있으면 true, 아니면 false를 반환하는 함수이다)

```
Algorithm DDD(X, n)
Let S be an empty stack
for i←0 to n−1 do
  if X[i] = '(' then
    S.push(X[i])
  else if X[i] = ')' then
    if S.empty() then
      ㉠
    end if
    S.pop()
  end if
end for
if S.empty() then
  ㉡
else
  ㉢
end if
```

① ㉠ return true, ㉡ return true, ㉢ return false

② ㉠ return false, ㉡ return true, ㉢ return false

③ ㉠ return true, ㉡ return false, ㉢ return true

④ ㉠ return false, ㉡ return false, ㉢ return true

정답 2번

출제 포인트

스택을 이용한 괄호쌍 검사 알고리즘에서 세 갈래 분기(㉠ 여는 괄호 없이 닫는 괄호가 나온 경우, ㉡ 순회가 끝난 뒤 스택이 빈 경우, ㉢ 순회가 끝난 뒤 스택이 남아 있는 경우)에 어떤 값을 반환해야 하는지를 정확히 채워 넣을 수 있는지를 묻는다.

NoteBase

올바른 순서

알고리즘은 배열 X를 앞에서부터 훑으면서 '('를 만나면 스택 S에 push하고, ')'를 만나면 그 ')'에 대응하는 '('가 스택에 있는지를 먼저 확인한다.
㉠ 지점: ')'를 만났는데 S.empty()가 참이라는 것은 이 ')'와 짝을 이룰 '('가 스택에 하나도 없다는 뜻이므로, 이 시점에서 이미 괄호쌍이 어긋난 것이 확정된다. 따라서 더 진행할 필요 없이 곧바로 return false 해야 한다.
㉠에서 스택이 비어 있지 않다면(if 문을 그냥 지나치면) 바깥의 S.pop()이 실행되어, 방금 만난 ')'와 짝이 되는 '('를 스택에서 하나 제거한다.
for 문이 끝난 뒤에는 지금까지 push된 '(' 중 pop되지 않고 남은 것이 있는지를 S.empty()로 확인한다.
㉡ 지점: 순회가 끝난 시점에 S.empty()가 참이면, 지금까지 push된 모든 '('가 어떤 ')'와 짝을 이루어 pop되었다는 뜻이므로 괄호쌍이 완전히 맞는 것이다. 따라서 return true 해야 한다.
㉢ 지점: 순회가 끝났는데도 스택이 비어 있지 않다면, 끝까지 짝을 찾지 못한 '('가 남아 있다는 뜻이므로 괄호쌍이 맞지 않는 것이다. 따라서 return false 해야 한다.
결국 ㉠=return false, ㉡=return true, ㉢=return false가 되어야 알고리즘이 모든 경우(짝이 안 맞는 ')', 완전히 짝이 맞는 경우, 짝이 안 맞는 '('가 남는 경우)를 올바르게 판별한다.

선지별 해설

① ㉠ return true / ㉡ return true / ㉢ return false — X. ㉠에서 스택이 비어 있는데도 return true를 하면, 짝 없는 ')'가 있는 수식(예: ")(")도 참으로 오판하게 되어 틀리다.
② ㉠ return false / ㉡ return true / ㉢ return false — O. 짝 없는 ')'를 만나면 즉시 false, 순회 후 스택이 비면 true, 스택에 '('가 남아 있으면 false로 판단하는 것이 알고리즘의 의도와 정확히 일치한다.
③ ㉠ return true / ㉡ return false / ㉢ return true — X. ㉠과 ㉡·㉢의 논리가 모두 반대로 되어 있어, 짝 없는 ')'를 참으로 처리하고 오히려 완전히 짝이 맞는 정상 수식을 거짓으로 반환하게 되므로 틀리다.
④ ㉠ return false / ㉡ return false / ㉢ return true — X. ㉠은 맞지만 ㉡·㉢이 뒤바뀌어, 모든 괄호가 정상적으로 짝을 이룬 수식(스택이 빈 경우)을 오히려 false로, 짝이 안 맞아 스택이 남은 경우를 오히려 true로 반환하게 되어 틀리다.

오답 포인트

이 문제의 함정은 ㉡과 ㉢의 조건을 반대로 착각하는 것이다. if S.empty() then ㉡ else ㉢ 구조에서, '스택이 비어 있다(empty)'는 것은 남은 '('가 없다는 뜻이므로 성공(true) 조건이고, '스택이 비어 있지 않다'는 것은 남은 '('가 있다는 뜻이므로 실패(false) 조건이다. 이를 반대로 읽으면 ③·④처럼 오답을 고르게 된다. 또한 ㉠에서 return true를 고르면 짝 없는 닫는 괄호를 정상으로 처리하는 치명적 오류가 생긴다.

NoteBase

다시 나오면 이렇게 판단

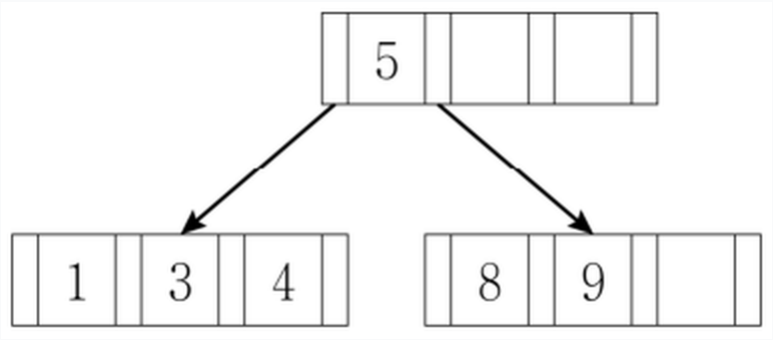
스택 기반 괄호 검사 알고리즘이 나오면 항상 두 가지 실패 조건을 분리해서 생각한다. 하나는 '닫는 괄호가 나왔는데 짝지을 여는 괄호가 스택에 없는 경우'(순회 도중 즉시 실패), 다른 하나는 '순회가 다 끝났는데 짝짓지 못한 여는 괄호가 스택에 남아 있는 경우'(순회 종료 후 실패)이다. 이 두 조건 중 하나라도 걸리면 false, 둘 다 걸리지 않으면 true라는 틀을 기억해 두면 빈칸의 방향을 헷갈리지 않는다.

암기 포인트

괄호 검사 스택 알고리즘의 3원칙: ① ')'를 만났을 때 스택이 비어 있으면 즉시 false, ② 순회가 끝난 후 스택이 비어 있으면 true, ③ 순회가 끝난 후 스택에 원소가 남아 있으면 false.

NoteBase

문 19. 다음과 같은 2-3-4 트리에 키 6과 7을 차례로 삽입한 이후의 트리 상태에 대한 설명으로 옳은 것은?



① 2-노드 1개, 3-노드 1개, 4-노드 2개로 구성된다.

② 루트 노드는 4-노드이다.

③ 가장 왼쪽의 리프 노드는 4-노드이다.

④ 가장 오른쪽의 리프 노드는 4-노드이다.

정답 3번

출제 포인트

2-3-4 트리(2-3-4 tree)는 한 노드가 1~3개의 키(2-노드~4-노드)를 가질 수 있는 균형 트리로, 이미 3개의 키(4-노드, 가득 찬 노드)를 가진 노드에 새 키를 넣어야 할 때는 그 노드를 먼저 분할(split)한 뒤 삽입을 이어간다. 이 문항은 주어진 트리에 6, 7을 차례로 삽입했을 때 노드 분할이 언제 일어나고 최종 트리 모양이 어떻게 되는지를 정확히 추적할 수 있는지를 묻는다.

조건 정리

삽입 전 트리: 루트=[5](2-노드), 왼쪽 자식=[1,3,4](4-노드, 이미 가득 참), 오른쪽 자식=[8,9](3-노드). 이 상태에서 키 6을 먼저 삽입하고, 이어서 키 7을 삽입한다.

사용할 공식

2-3-4 트리의 하향식(top-down) 삽입 규칙: 루트에서부터 내려가며 다음 자식으로 이동하기 전에 그 자식이 4-노드(키 3개, 가득 참)이면 먼저 분할한다. 분할 시 가운데 키는 부모로 올라가고, 나머지 두 키는 각각 좌우 자식(2-노드)으로 분리된다. 분할 후에도 부모 노드는 아직 4-노드가 아니라면 더 이상 분할할 필요가 없다. 자식이 가득 차 있지 않으면 그대로 내려가 리프에 키를 삽입한다.

NoteBase

계산 과정

① 키 6 삽입: 루트 [5]는 1개 키로 가득 차지 않았으므로 분할 없이 6>5 방향, 즉 오른쪽 자식 [8,9]로 내려간다. [8,9]도 2개 키로 가득 차지 않았으므로(4-노드가 아니므로) 그대로 6을 삽입해 [6,8,9]가 된다. 이 시점 트리: 루트 [5], 왼쪽 자식 [1,3,4], 오른쪽 자식 [6,8,9].
② 키 7 삽입: 루트 [5]에서 7>5이므로 오른쪽 자식으로 내려가려 하는데, 오른쪽 자식 [6,8,9]가 3개 키를 가진 4-노드(가득 참)이므로 내려가기 전에 먼저 분할한다. 가운데 키 8이 루트로 올라가 루트는 [5,8](3-노드, 아직 4-노드 아님, 추가 분할 불필요)이 되고, [6,8,9]는 [6]과 [9] 두 개의 2-노드로 나뉜다. 분할 후 트리: 루트 [5,8], 왼쪽 자식 [1,3,4], 가운데 자식 [6], 오른쪽 자식 [9].
③ 이어서 7을 삽입할 위치를 찾는다. 7은 5보다 크고 8보다 작으므로 가운데 자식 [6]으로 내려간다. [6]은 1개 키만 있어 가득 차지 않았으므로 그대로 7을 삽입해 [6,7]이 된다.
최종 트리: 루트 [5,8](3-노드), 왼쪽 리프 [1,3,4](4-노드), 가운데 리프 [6,7](3-노드), 오른쪽 리프 [9](2-노드).

선지 판단

① X — 최종 구성은 2-노드 1개(오른쪽 리프 [9]), 3-노드 2개(루트 [5,8], 가운데 리프 [6,7]), 4-노드 1개(왼쪽 리프 [1,3,4])이다. "2-노드 1개, 3-노드 1개, 4-노드 2개"라는 개수 조합과 일치하지 않는다.
② X — 루트는 삽입 과정에서 [5,8]로 바뀌어 키가 2개인 3-노드가 되었다. 4-노드(키 3개)가 아니다.
③ O — 가장 왼쪽의 리프 노드는 처음부터 끝까지 변하지 않은 [1,3,4]이며, 키가 3개이므로 4-노드가 맞다.
④ X — 가장 오른쪽의 리프 노드는 분할 이후 [9] 하나만 남아 키가 1개인 2-노드이다. 4-노드가 아니다.

오답 포인트

①과 ②는 삽입 도중 루트가 분할로 인해 [5]에서 [5,8]로 바뀐다는 점을 놓치면 나오는 오답이다. 특히 ②는 가득 찼던 오른쪽 자식 [6,8,9]가 분할되면서 그 가운데 키가 루트로 올라간다는 사실을 반영하지 못하면 고르기 쉽다. ④는 오른쪽 자식이 분할되어 키 개수가 3개에서 1개로 줄어든다는 것을 놓치고 삽입 전 상태([8,9]에 가까운 이미지)로 착각할 때 나오는 오답이다.

다시 나오면 이렇게 판단

2-3-4 트리 삽입 문제는 항상 "내려가기 전에 가득 찬(4-노드) 자식을 만나면 먼저 분할"이라는 하향식 규칙을 순서대로 적용한다. 삽입할 키마다 ⑴ 루트부터 시작해 다음에 내려갈 자식이 4-노드인지 확인 ⑵ 4-노드이면 가운데 키를 부모로 올리고 좌우로 분할 ⑶ 분할 후(또는 가득 차 있지 않았다면 그대로) 목표 위치까지 내려가 키를 삽입, 순으로 표를 그리듯 단계별로 적어 나가면 실수를 줄일 수 있다.

암기 포인트

2-노드=키 1개(자식 2개), 3-노드=키 2개(자식 3개), 4-노드=키 3개(자식 4개, 가득 참)이다. 2-3-4 트리의 하향식 삽입은 "가득 찬 노드는 지나가기 전에 미리 쪼갠다"는 원칙 하나로 전 과정을 설명할 수 있다.

NoteBase

문 20. 다음 행렬은 그래프(graph)를 인접 행렬(adjacency matrix)로 나타낸 것이다. 이 그래프에서 솔린 알고리즘(Sollin algorithm)을 이용하여 최소 비용 신장 트리(minimum spanning tree)를 찾고자 한다. 알고리즘 수행 과정에서 나타나는 부분 그래프에 해당하지 않는 것은? (단, 부분 그래프는 간선의 집합으로 표현하며, 간선은 그래프에 포함된 두 정점 v, k에 대응하여 (v, k)의 형태로 표현한다)

정점	0	1	2	3	4	5	6
0	0	20	15	∞	∞	8	∞
1	20	0	5	12	∞	∞	∞
2	15	5	0	25	35	28	6
3	∞	12	25	0	7	∞	∞
4	∞	∞	35	7	0	∞	11
5	8	∞	28	∞	∞	0	9
6	∞	∞	6	∞	11	9	0

① { (0, 5) }

② { (1, 3), (3, 4) }

③ { (3, 4) }

④ { (1, 2), (2, 6) }

정답 2번

출제 포인트

솔린(Sollin) 알고리즘, 즉 보르우브카(Bor□vka) 알고리즘의 수행 과정을 정확히 시뮬레이션할 수 있는지를 묻는 문제이다. 솔린 알고리즘은 크루스칼이나 프림처럼 정점을 하나씩 늘리거나 간선을 하나씩 추가하는 방식이 아니라, 매 라운드마다 '현재 존재하는 모든 컴포넌트(연결 요소)'가 동시에 자신과 연결된 간선 중 가중치가 가장 작은 것을 하나씩 선택하고, 그 간선들을 한꺼번에 부분 그래프에 추가하는 방식으로 진행된다. 따라서 알고리즘 수행 도중에 나타날 수 있는 부분 그래프는 '각 컴포넌트가 고른 최소 간선들의 모음'이어야 하며, 그렇지 않은 간선 조합은 중간 과정에 등장할 수 없다.

NoteBase

조건 정리

정점은 0부터 6까지 7개이며, 인접 행렬의 값이 간선의 가중치이다(∞는 간선이 없음을 의미). 간선과 가중치를 정리하면 다음과 같다: (0,1)=20, (0,2)=15, (0,5)=8, (1,2)=5, (1,3)=12, (2,3)=25, (2,4)=35, (2,5)=28, (2,6)=6, (3,4)=7, (4,6)=11, (5,6)=9. 솔린 알고리즘 초기에는 정점 하나하나가 각각 별개의 컴포넌트이다.

사용할 공식

솔린(보르우브카) 알고리즘의 각 라운드는 다음 규칙을 따른다. 현재 존재하는 모든 컴포넌트에 대해, 그 컴포넌트에 속한 정점과 다른 컴포넌트에 속한 정점을 잇는 간선 중 가중치가 최소인 간선을 하나씩 고른다. 이렇게 고른 간선들을 모두(동시에) 부분 그래프에 추가하고, 그 결과로 합쳐진 컴포넌트들을 다음 라운드의 새로운 컴포넌트로 삼는다. 이 과정을 컴포넌트가 하나만 남을 때까지 반복한다.

계산 과정

1라운드: 아직 모든 정점이 개별 컴포넌트이므로, 정점별로 자신에게 연결된 가장 작은 간선을 찾는다.
정점 0: (0,1)=20, (0,2)=15, (0,5)=8 중 최소는 (0,5)=8
정점 1: (1,0)=20, (1,2)=5, (1,3)=12 중 최소는 (1,2)=5
정점 2: (2,0)=15, (2,1)=5, (2,3)=25, (2,4)=35, (2,5)=28, (2,6)=6 중 최소는 (1,2)=5
정점 3: (3,1)=12, (3,2)=25, (3,4)=7 중 최소는 (3,4)=7
정점 4: (4,2)=35, (4,3)=7, (4,6)=11 중 최소는 (3,4)=7
정점 5: (5,0)=8, (5,2)=28, (5,6)=9 중 최소는 (0,5)=8
정점 6: (6,2)=6, (6,4)=11, (6,5)=9 중 최소는 (2,6)=6
따라서 1라운드에서 선택되는 간선의 합집합은 {(0,5), (1,2), (3,4), (2,6)}이다. 이 결과로 컴포넌트는 {0,5}, {1,2,6}, {3,4} 세 개로 합쳐진다.
2라운드: 세 컴포넌트 각각이 바깥으로 나가는 간선 중 최소를 찾는다.
컴포넌트 {0,5}: 바깥으로 나가는 간선 (0,1)=20, (0,2)=15, (5,2)=28, (5,6)=9 중 최소는 (5,6)=9
컴포넌트 {1,2,6}: 바깥으로 나가는 간선 (1,3)=12, (2,3)=25, (2,4)=35, (6,4)=11, (2,5)=28, (6,5)=9, (2,0)=15 중 최소는 (5,6)=9(=(6,5))
컴포넌트 {3,4}: 바깥으로 나가는 간선 (3,1)=12, (3,2)=25, (4,2)=35, (4,6)=11 중 최소는 (4,6)=11
따라서 2라운드에서 새로 추가되는 간선은 {(5,6), (4,6)}이며, 이로써 모든 정점이 하나의 컴포넌트로 합쳐져 알고리즘이 종료된다.
결국 알고리즘 진행 과정에서 실제로 등장(선택)하는 간선은 (0,5), (1,2), (2,6), (3,4), (5,6), (4,6) 여섯 개뿐이며, 그 부분집합들이 중간 단계의 부분 그래프로 나타날 수 있다. 이 목록에 (1,3)=12는 어느 라운드에서도 선택된 적이 없다. 정점 1이 속한 컴포넌트는 항상 (1,2)=5를 택하고, 정점 3이 속한 컴포넌트는 항상 (3,4)=7을 택하므로, 가중치가 더 큰 (1,3)=12는 최소 간선으로 뽑힐 기회가 전혀 없다.

NoteBase

선지 판단

① { (0,5) } — O, (0,5)는 1라운드에서 정점 0과 정점 5가 공통으로 선택하는 최소 간선이므로, 알고리즘 진행 중 나타나는 부분 그래프의 일부로 충분히 등장할 수 있다.
② { (1,3), (3,4) } — X, (3,4)=7은 실제로 선택되는 간선이 맞지만, (1,3)=12는 어떤 라운드에서도 최소 간선으로 선택된 적이 없다. 정점 1(또는 그 정점을 포함하는 컴포넌트)은 항상 가중치 5인 (1,2)를 택하므로, (1,3)이 포함된 조합은 솔린 알고리즘 수행 과정에 나타날 수 없다. → 부분 그래프에 해당하지 않는 것.
③ { (3,4) } — O, (3,4)=7은 1라운드에서 정점 3과 정점 4가 공통으로 선택하는 최소 간선이므로 나타날 수 있다.
④ { (1,2), (2,6) } — O, (1,2)=5는 정점 1과 정점 2가, (2,6)=6은 정점 6이 1라운드에서 선택하는 최소 간선이므로, 두 간선 모두 1라운드 결과에 포함되는 조합이다.

오답 포인트

이 문제는 최종 MST(최소 신장 트리)가 무엇인지를 묻는 것이 아니라, 알고리즘이 진행되는 '중간 과정'에서 나올 수 있는 부분 그래프를 묻는 것이다. 그래서 얼핏 보면 가중치가 작아 보이는 간선이면 다 맞는 답처럼 보일 수 있으나, 실제로는 '그 간선이 어느 라운드에서 어떤 컴포넌트의 최소 간선으로 뽑히는가'를 하나하나 확인해야 한다. (1,3)=12처럼 가중치가 작아 보여도, 그 정점이 속한 컴포넌트에 더 작은 간선이 이미 있으면 절대 선택되지 않는다는 점을 놓치기 쉽다.

다시 나오면 이렇게 판단

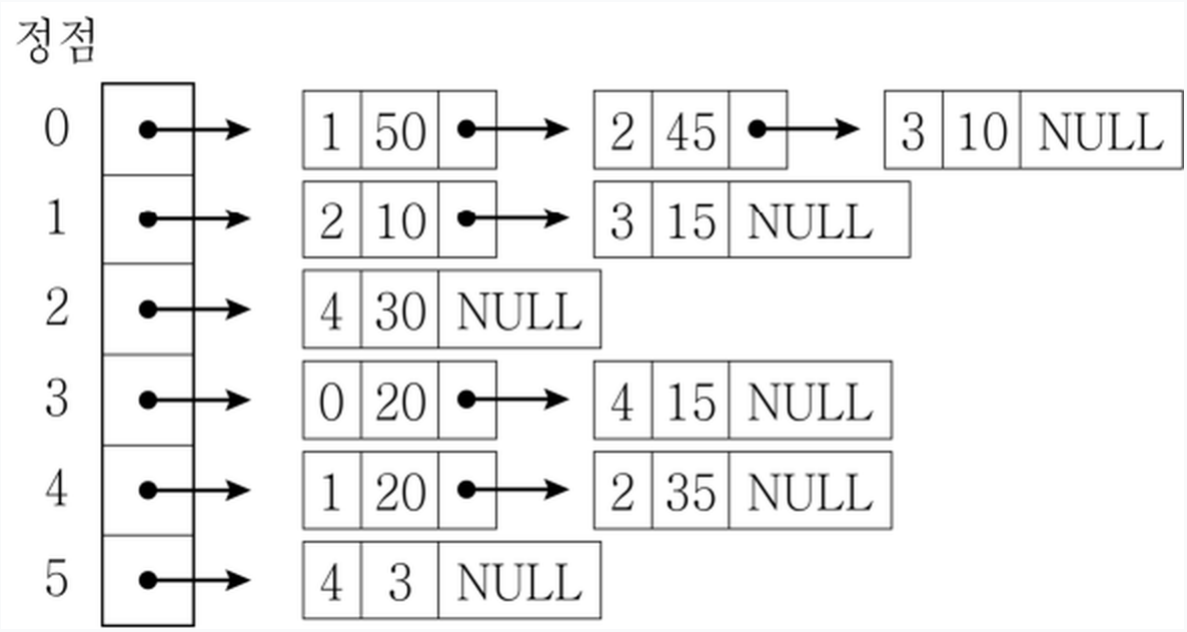
솔린(보르우브카) 알고리즘 문제가 나오면, 먼저 정점 하나하나를 개별 컴포넌트로 두고 각 정점이 이어진 간선 중 최솟값을 모두 찾아 1라운드 결과를 만든다. 그다음 합쳐진 컴포넌트 단위로 다시 바깥으로 나가는 최소 간선을 찾는 2라운드를 진행하며, 이 과정을 컴포넌트가 하나가 될 때까지 반복한다. 선택지에 있는 간선 집합이 실제 라운드 결과(또는 그 부분집합)에 포함되는지를 하나씩 대조하면 정답이 아닌 것을 쉽게 골라낼 수 있다.

암기 포인트

솔린(보르우브카) 알고리즘은 '정점 단위'가 아니라 '컴포넌트 단위'로 동시에 최소 간선을 고르는 것이 핵심이다. 크루스칼처럼 전체 간선을 가중치 순으로 정렬해 하나씩 보는 것도 아니고, 프림처럼 하나의 트리를 한 정점씩 확장하는 것도 아니다. 각 라운드에서 모든 컴포넌트가 동시에 자신의 최소 outgoing 간선을 고르고, 그 결과를 한꺼번에 합친다는 점을 기억해 둔다.

NoteBase

문 21. 다음은 방향그래프를 인접 리스트로 표현한 것이다. 이 그래프에 대한 설명으로 옳지 않은 것은?



① 모든 사이클 경로는 (0,3,0), (2,4,2), (0,1,3,0), (1,2,4,1), (1,3,4,1), (0,2,4,1,3,0)이고, 그 경로에 각각 대응하는 길이는 2, 2, 3, 3, 3, 5이다.

② 각 정점(0,1,2,3,4,5)에 각각 대응되는 진입차수는 1, 2, 3, 2, 3, 0이다.

③ 각 정점(0,1,2,3,4,5)에 각각 대응되는 인접한 정점의 집합은 {1,2}, {2,3}, {4}, {0,4}, {1,2}, {4,2}이다.

④ 각 정점(0,1,2,3,4,5)에서 각각 대응되는 진출차수는 3, 2, 1, 2, 2, 1이다.

정답 3번

출제 포인트

인접 리스트(배열+연결리스트)로 표현된 방향그래프를 보고 각 정점의 진출차수·진입차수·사이클·인접 정점 집합을 정확히 읽어낼 수 있는지를 묻는다. 정점 배열의 각 칸이 헤드 포인터이고, 거기서 뻗어나가는 연결리스트의 각 노드가 (인접 정점 번호, 간선 가중치)를 담고 있다는 구조를 이해해야 한다.

NoteBase

자료 읽는 순서

① 먼저 정점 배열의 각 행에서 시작해 연결리스트를 끝(NULL)까지 따라가며 간선을 모두 나열한다.
정점 0: 0→1(50), 0→2(45), 0→3(10)
정점 1: 1→2(10), 1→3(15)
정점 2: 2→4(30)
정점 3: 3→0(20), 3→4(15)
정점 4: 4→1(20), 4→2(35)
정점 5: 5→4(3)
② 각 정점에서 뻗어나가는 노드 개수를 세면 진출차수(out-degree), 다른 정점의 리스트에 자신이 몇 번 등장하는지를 세면 진입차수(in-degree)가 된다.
③ 인접한 정점의 집합은 해당 정점의 연결리스트에 있는 목적지 정점 번호를 모두 모은 것이다(가중치는 제외).

선지별 해설

① O — 그래프의 모든 간선을 나열해 보면 0→3→0(0→3, 3→0), 2→4→2(2→4, 4→2), 0→1→3→0, 1→2→4→1, 1→3→4→1, 0→2→4→1→3→0이 실제로 존재하는 사이클이며, 그 길이도 각각 2, 2, 3, 3, 3, 5로 정확히 대응된다.
② O — 진입차수는 각 정점을 목적지로 갖는 간선 수이다. 0은 3→0 하나(1), 1은 0→1·4→1 둘(2), 2는 0→2·1→2·4→2 셋(3), 3은 0→3·1→3 둘(2), 4는 2→4·3→4·5→4 셋(3), 5는 자신을 향하는 간선이 없어 0이다. 제시된 1, 2, 3, 2, 3, 0과 정확히 일치한다.
③ X — 정점 0의 연결리스트를 끝까지 따라가면 1, 2, 3 세 정점 모두로 나가므로 인접 정점 집합은 {1,2,3}이어야 하는데 선지는 {1,2}로 3을 빠뜨렸다. 또한 정점 5의 연결리스트에는 4 하나뿐인데 선지는 {4,2}로 존재하지 않는 2를 잘못 포함했다. 두 군데 모두 실제 인접 리스트 내용과 다르므로 옳지 않다. (참고로 1은 {2,3}, 2는 {4}, 3은 {0,4}, 4는 {1,2}로 이 넷은 맞게 제시되었다.)
④ O — 진출차수는 각 정점의 연결리스트에 매달린 노드 개수이다. 0은 3개(1,2,3), 1은 2개(2,3), 2는 1개(4), 3은 2개(0,4), 4는 2개(1,2), 5는 1개(4)이므로 제시된 3, 2, 1, 2, 2, 1과 정확히 일치한다.

오답 포인트

①②④는 각각 사이클 나열, 진입차수, 진출차수를 실제 간선과 하나씩 대조하면 모두 정확히 맞아떨어지므로 오답으로 고르기 쉽지 않다. 함정은 ③에서 인접 정점의 집합을 셀 때 연결리스트의 마지막 노드까지 끝까지 따라가지 않고 중간에서 멈추거나(정점 0에서 3을 빠뜨림), 반대로 다른 정점의 값을 섞어 넣는(정점 5에 2를 잘못 포함) 실수가 나온다는 점이다.

다시 나오면 이렇게 판단

인접 리스트 문제가 나오면 우선 각 정점의 연결리스트를 표 형태로 다시 옮겨 적어 간선 목록을 확정한 뒤, 그 표를 근거로 진출차수·진입차수·인접 집합·사이클을 각각 계산한다. 선지에 나열된 숫자나 집합을 자신이 만든 표와 하나씩 대조하며 빠지거나 더해진 원소가 없는지 확인한다.

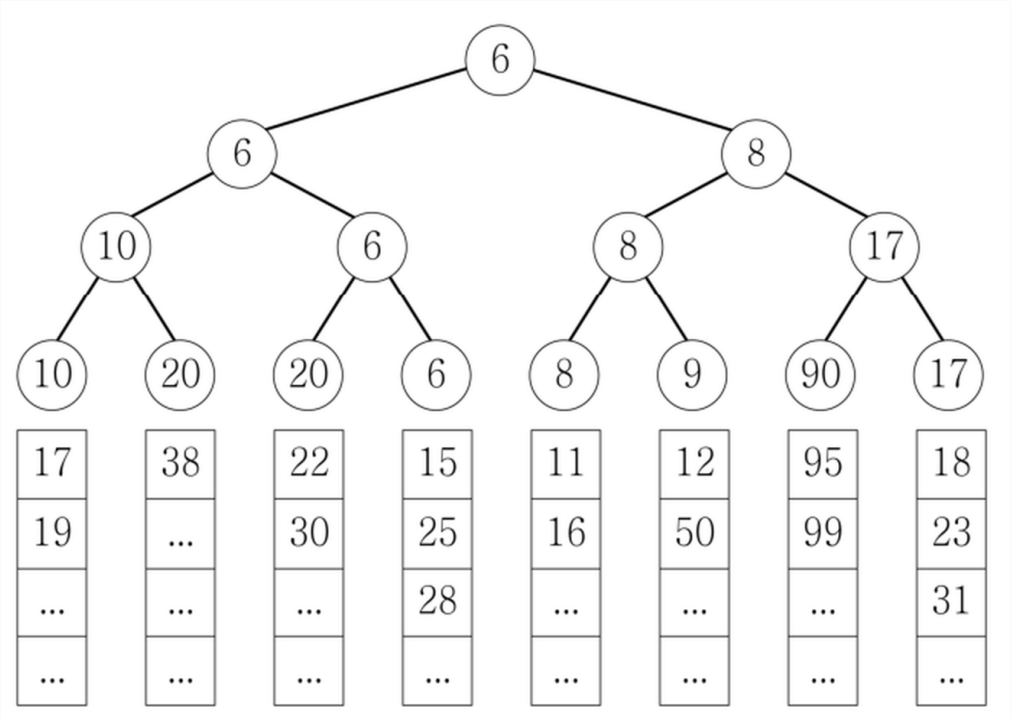
NoteBase

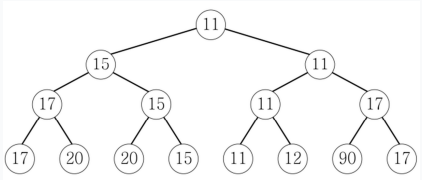
암기 포인트

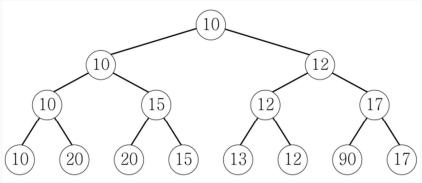
진출차수(out-degree)=해당 정점의 연결리스트 노드 개수, 진입차수(in-degree)=다른 정점들의 리스트에 그 정점이 등장한 총 횟수, 인접한 정점의 집합=해당 정점의 연결리스트를 끝까지 따라가며 얻은 목적지 정점 전체(가중치 제외)로 구분해 기억해 둔다.

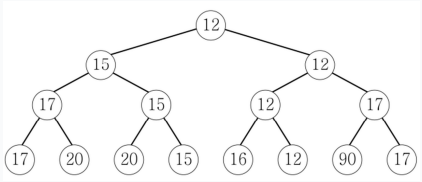
NoteBase

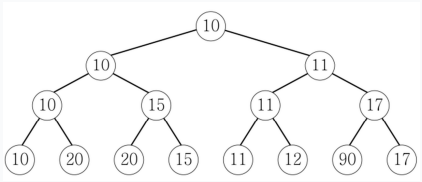
문 22. 승자 트리는 단말 노드(leaf node)에서부터 시작해서, 가장 작은 키 값을 갖는 노드(승자)를 찾아 부모로 올려서 루트 노드로부터 가장 작은 값을 찾아내는 방식이다. 승자 트리의 단말 노드에는 정렬된 배열(run)이 각각 달려 있고, 이 run에서 첫 번째 값을 읽어 가장 작은 값을 올려 내는 방식으로 모든 run을 비우면서 값을 정렬하게 된다. 8개의 run이 있는 아래 승자 트리에서 6, 8, 9 순서대로 3개의 레코드를 출력하고 난 뒤 run이 생략된 승자 트리의 형태로 옳은 것은?



① 

② 

③ 

④ 

NoteBase

정답 4번

출제 포인트

승자 트리(winner tree, 토너먼트 트리)에서 최솟값을 하나 뽑아 출력한 뒤, 그 값이 있던 단말(리프)에 같은 run의 다음 값을 채워 넣고 루트까지 다시 비교를 갱신하는 과정을 정확히 시뮬레이션할 수 있는지를 묻는다. 8개의 run이 각각 하나의 리프에 연결되어 있고, 리프 값이 출력되어 빠지면 그 리프 아래 run 배열의 맨 앞 값이 새 리프 값으로 올라온다는 규칙을 이해해야 한다.

조건 정리

리프(좌→우) L1~L8의 초기값과 그 아래 run의 나머지 값은 다음과 같다.
L1=10, run 나머지 17,19,…
L2=20, run 나머지 38,…
L3=20, run 나머지 22,30,…
L4=6, run 나머지 15,25,28,…
L5=8, run 나머지 11,16,…
L6=9, run 나머지 12,50,…
L7=90, run 나머지 95,99,…
L8=17, run 나머지 18,23,31,…
3단계 노드는 두 리프씩 묶은 최솟값이고(예: 3단계 첫 노드=min(L1,L2)), 2단계 노드는 3단계 두 노드씩 묶은 최솟값, 루트는 2단계 두 노드의 최솟값이다.

사용할 공식

승자 트리 갱신 규칙: 한 값이 출력(승자로 확정)되면 그 리프 자리에는 같은 run의 다음 값이 새로 채워진다. 이후 그 리프에서 루트까지 이어지는 경로 상의 모든 조상 노드를 부모 = (왼쪽 자식, 오른쪽 자식)으로 다시 계산한다.

NoteBase

계산 과정

1) 첫 출력: 전체 리프 중 최솟값은 L4=6이므로 6이 출력된다. L4 자리에는 run4의 다음 값인 15가 채워진다(L4=15). 이때 3단계 두 번째 노드=min(L3=20, L4=15)=15로 바뀐다.
2) 두 번째 출력: 갱신된 트리에서 최솟값은 L5=8이므로 8이 출력된다. L5 자리에는 run5의 다음 값인 11이 채워진다(L5=11). 3단계 세 번째 노드=min(L5=11, L6=9)=9로 갱신된다.
3) 세 번째 출력: 최솟값은 L6=9이므로 9가 출력된다. L6 자리에는 run6의 다음 값인 12가 채워진다(L6=12). 3단계 세 번째 노드=min(L5=11, L6=12)=11로 갱신된다.
최종 리프(좌→우): 10, 20, 20, 15, 11, 12, 90, 17.
3단계: min(10,20)=10, min(20,15)=15, min(11,12)=11, min(90,17)=17.
2단계: min(10,15)=10, min(11,17)=11.
루트: min(10,11)=10.
따라서 6, 8, 9를 출력한 뒤의 승자 트리는 루트 10 / 2단계 10,11 / 3단계 10,15,11,17 / 리프 10,20,20,15,11,12,90,17 이다.

선지 판단

① X: 루트가 11로 되어 있으나 계산 결과 루트는 10이어야 한다. 리프값도 17,20으로 시작해 첫 리프가 10이 아니므로 틀렸다.
② X: 루트 10, 2단계 10,12까지는 얼핏 비슷하지만 2단계 오른쪽 값이 12가 아니라 11이어야 하고, 세 번째 3단계 노드 리프가 13,12로 되어 있어 계산 결과(11,12)와 다르다.
③ X: 루트가 12로 되어 있어 계산 결과인 10과 다르다.
④ O: 루트 10, 2단계 10·11, 3단계 10·15·11·17, 리프 10,20,20,15,11,12,90,17로 계산 결과와 정확히 일치한다.

오답 포인트

가장 흔한 실수는 리프를 교체할 때 run의 '다음 값'을 잘못 짚는 것이다. 예를 들어 L6=9를 출력한 뒤 run6의 다음 값은 배열 맨 앞인 12인데, 이를 건너뛰고 그다음 값인 50을 채우면 3단계·2단계·루트가 모두 달라진다. 또한 리프 하나를 갱신한 뒤에는 그 리프에서 루트까지 이어지는 경로의 모든 조상 노드를 다시 계산해야 하는데, 일부만 갱신하고 멈추면 최종 트리가 틀어진다.

다시 나오면 이렇게 판단

승자 트리 문제는 표로 각 리프의 현재값과 그 아래 run의 남은 값을 먼저 정리한 뒤, 출력할 때마다 (1) 어느 리프에서 승자가 나왔는지 (2) 그 리프에 run의 다음 값이 무엇으로 채워지는지 (3) 그 리프의 조상 노드들을 루트까지 순서대로 다시 최솟값으로 갱신하는지, 이 세 단계를 매 출력마다 반복해서 표에 그대로 적어 나가면 실수를 줄일 수 있다.

NoteBase

암기 포인트

승자 트리는 승자가 출력되면 그 리프에 같은 run의 다음 값이 즉시 채워지고, 루트까지의 경로만 다시 비교하면 된다는 점을 기억해 둔다. 부모 노드 = 두 자식 중 최솟값이라는 규칙을 벗어나지 않는다.

NoteBase

문 23. 다음 코드는 C 언어를 이용하여 주어진 연결 리스트 구조의 데이터에 대한 삽입 정렬 함수를 구현한 것이다. ㉠ ~ ㉤에 들어갈 코드를 바르게 연결한 것은?

```
typedef struct _node {
  int data;
  struct _node *next;
}node;

void Insert(node **headRef, node *newNode) {
  if (*headRef == NULL ||
      (*headRef)->data >= newNode->data) {
    newNode->next = *headRef;
    *headRef = newNode;
  }
  else {
    node *current = *headRef;
    while (current->next != NULL &&
           current->next->data < newNode->data){
           _____㉠_____
    }
    _____㉡_____
    _____㉢_____
  }
}

void Sort(node **headRef) {
  node *result = NULL, *current = *headRef;
  node *next;
  while (current!=NULL) {
    _____㉣_____
    Insert(&result, current);
    _____㉤_____
  }
  *headRef = result;
}
```

```
A : current->next = current;
B : current = current->next;
C : current->next = newNode;
D : next = current->next;
E : current->next = next;
F : current = next;
G : newNode->next = current->next;
```

① ㉠: A, ㉡: D, ㉢: E, ㉣: F, ㉤: G

② ㉠: B, ㉡: G, ㉢: C, ㉣: D, ㉤: F

③ ㉠: C, ㉡: E, ㉢: B, ㉣: A, ㉤: D

④ ㉠: E, ㉡: G, ㉢: B, ㉣: F, ㉤: A

정답 2번

출제 포인트

정렬된 연결 리스트에 새 노드를 삽입하는 Insert 함수와, 이를 반복 호출해 전체 리스트를 삽입 정렬하는 Sort 함수의 빈칸을 채우는 문제이다. 포인터 next를 이용해 노드를 한 칸씩 이동하는 코드와, 노드 사이에 새 노드를 끼워 넣는 코드, 그리고 다음 순회를 위해 원래의 next 포인터를 미리 저장해 두는 코드를 정확히 구분할 수 있는지를 묻는다.

조건 정리

Insert 함수의 else 블록은 삽입 위치가 리스트의 맨 앞이 아닌 경우를 처리한다. while문은 current->next가 NULL이 아니고 current->next->data가 newNode->data보다 작은 동안 반복하며, 이 조건이 거짓이 되는 지점, 즉 current와 current->next 사이가 newNode를 끼워 넣을 위치이다. Sort 함수는 원래 리스트를 current로 순회하면서 각 노드를 결과 리스트 result에 Insert로 하나씩 삽입해 나가는 구조이다.

NoteBase

사용할 공식

연결 리스트에서 다음 노드로 이동할 때는 *current = current- > next* 형태의 대입을 사용한다. 두 노드 사이에 새 노드를 끼워 넣을 때는 먼저 새 노드의 next가 삽입 위치 뒤쪽 노드를 가리키게 한 다음(*newNode- > next = current- > next*), 앞쪽 노드의 next가 새 노드를 가리키도록 연결한다(*current- > next = newNode*). 이 순서를 바꾸면 뒤쪽 노드에 대한 참조를 잃어버리게 된다. 또한 리스트를 순회하며 현재 노드를 다른 함수(Insert)에 넘겨 그 노드의 next 값이 바뀔 수 있는 경우에는, 순회를 이어가기 위해 원래의 next 값을 미리 별도의 변수에 저장해 두어야 한다.

계산 과정

㉠: while문 안에서 current->next->data가 newNode->data보다 작은 동안 계속 다음 자리를 확인해야 하므로 current를 한 칸 전진시켜야 한다. 이는 B(*current = current- > next*)에 해당한다.
㉡, ㉢: while문을 빠져나오면 current와 current->next 사이에 newNode를 끼워 넣어야 한다. 먼저 newNode->next가 current->next를 가리키게 해야 하므로 ㉡은 G(*newNode- > next = current- > next*)이고, 그다음 current->next가 newNode를 가리키게 연결해야 하므로 ㉢은 C(*current- > next = newNode*)이다. 순서가 바뀌면(C를 먼저 실행하면) current->next가 이미 newNode로 바뀐 뒤에 G를 실행하게 되어 newNode->next가 newNode 자신을 가리키는 오류가 생긴다.
㉣, ㉤: Sort 함수에서 Insert(&result, current)를 호출하면 current->next 값이 삽입 과정에서 바뀔 수 있다. 따라서 Insert를 호출하기 전에 원래 리스트에서의 다음 노드를 next 변수에 미리 저장해 두어야 하므로 ㉣은 D(*next = current- > next*)이고, Insert 호출 후에는 저장해 둔 next로 current를 이동시켜 순회를 이어가야 하므로 ㉤은 F(*current = next*)이다.
따라서 ㉠~㉤은 각각 B, G, C, D, F이며 이는 선택지 2번과 일치한다.

선지 판단

① A, D, E, F, G — X. ㉠에 A(current->next = current)를 넣으면 current가 자기 자신을 가리키게 되어 무한 루프 및 리스트 구조 파괴가 일어난다. 전진 코드가 아니므로 틀렸다.
② B, G, C, D, F — O. 위 계산 과정에서 확인한 것과 정확히 일치한다. current 전진(B), newNode 연결(G, C 순서), next 저장(D)과 current 이동(F)이 모두 올바르게 대응된다.
③ C, E, B, A, D — X. ㉠에 C(current->next = newNode)를 넣으면 while문 안에서 매 반복마다 리스트가 끊어져 버려 순회 자체가 불가능해진다. 전진 동작이 아니므로 틀렸다.
④ E, G, B, F, A — X. ㉠에 E(current->next = next)를 넣으면 아직 정의되지 않았거나 이전 값을 가진 next 변수를 사용하게 되어 current 전진이 이루어지지 않는다. 전진 코드가 아니므로 틀렸다.

NoteBase

오답 포인트

이 문제에서 가장 흔한 실수는 ㉡과 ㉢의 순서를 반대로 생각하는 것이다. current->next를 먼저 newNode로 바꿔 버리면(C를 먼저 실행) 그 뒤에 newNode->next에 대입할 current->next 값은 이미 newNode 자신이 되어버려 리스트가 끊어진다. 또 하나의 흔한 실수는 Sort 함수에서 next 변수를 저장하는 순서를 헷갈리는 것으로, Insert 호출 전에 미리 next를 저장해 두지 않으면 Insert가 current->next를 바꿔버린 뒤라서 원래 리스트의 다음 노드를 잃어버리게 된다.

다시 나오면 이렇게 판단

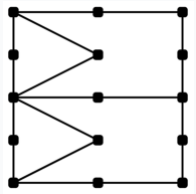
연결 리스트 코드에서 빈칸 유형이 나오면, 먼저 각 빈칸이 '노드 전진'인지 '노드 연결(포인터 재배치)'인지 '값 저장'인지부터 구분한다. 노드를 서로 연결할 때는 반드시 뒤쪽을 먼저 연결한 다음 앞쪽을 연결하는 순서를 지켜야 한다는 원칙을 적용하고, 순회 도중 다른 함수 호출로 포인터가 바뀔 위험이 있으면 호출 전에 다음 위치를 미리 변수에 저장해 두는 패턴인지 확인한다.

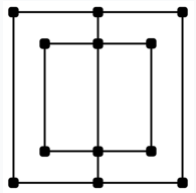
암기 포인트

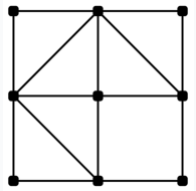
연결 리스트에 새 노드를 끼워 넣을 때는 '뒤쪽 연결 먼저, 앞쪽 연결 나중'(newNode->next 설정 후 current->next 설정) 순서를 지킨다. 순회 중인 노드를 다른 함수에 넘겨 그 노드의 next가 바뀔 수 있다면, 호출 전에 다음 노드를 별도 변수에 저장해 두어야 순회가 끊기지 않는다.

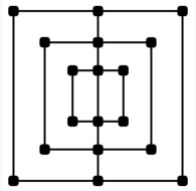
NoteBase

문 24. 단순 무방향 그래프(simple undirected graph) G가 주어졌을 때, 정점(vertex)의 중복 방문을 허용하되, 간선(edge)의 중복은 허용하지 않는 경로(path)를 트레일(trail)이라고 한다. 그래프의 모든 간선들을 단 한 번씩만 통과하되 출발점과 도착점이 일치하지 않는 것을 오일러 트레일(Euler trail)이라고 할 때, 다음 그래프 중에서 오일러 트레일이 존재하지 않는 그래프는?

① 

② 

③ 

④ 

정답 1번

출제 포인트

오일러 트레일(시작점과 끝점이 다른 오일러 경로)이 존재하려면 그래프가 연결되어 있어야 하고, 차수(degree)가 홀수인 정점의 개수가 정확히 2개여야 한다. 홀수차 정점이 0개이면 오일러 서킷(순환)은 존재하지만 출발점과 도착점이 다른 오일러 트레일은 존재하지 않고, 홀수차 정점이 4개 이상이면 오일러 트레일도 서킷도 존재하지 않는다. 이 문제는 각 선택지 그래프에서 정점별로 간선을 세어 홀수차 정점의 개수를 판정하는 문제이다.

NoteBase

선지별 해설

① 좌측 열의 세 분기점(맨 위·가운데·맨 아래)에서 각각 지그재그 대각선이 뻗어 나가는 구조이다. 가운데 분기점은 위쪽 대각선, 아래쪽 대각선, 좌측 위아래 연결, 가로선(우측 열 방향)까지 모두 이어져 차수가 5(홀수)이고, 맨 위 분기점과 맨 아래 분기점은 각각 대각선 1개, 세로선 1개, 가로선 1개로 차수가 3(홀수)이다. 여기에 우측 열 가운데 정점도 가로선·위아래 세로선이 모여 차수가 3(홀수)이 된다. 정리하면 홀수차 정점이 4개가 되어 짝이 맞지 않으므로, 오일러 트레일은커녕 오일러 서킷도 존재하지 않는다.
② 바깥 정사각형과 안쪽 작은 정사각형이 하나의 수직선으로 이어진 구조이다. 바깥 정사각형의 위쪽 중간 정점은 좌우 변과 안쪽으로 내려가는 수직선까지 합쳐 차수가 3(홀수)이고, 아래쪽 중간 정점도 같은 이유로 차수가 3(홀수)이다. 나머지 정점(모서리, 안쪽 정사각형의 중간 정점 등)은 모두 차수가 2 또는 4로 짝수이다. 따라서 홀수차 정점이 정확히 2개이므로 오일러 트레일이 존재한다.
③ 3x3 격자 기본선에 대각선 3개(상단 중앙-중단 좌측, 상단 중앙-중단 우측, 중단 좌측-하단 중앙)가 추가된 구조이다. 상단 중앙 정점은 좌우 격자선 2개, 아래 격자선 1개, 대각선 2개로 차수가 5(홀수)이고, 중단 좌측 정점은 위아래 격자선 2개, 오른쪽 격자선 1개, 대각선 2개로 차수가 5(홀수)이다. 나머지 정점은 모두 짝수 차수이다. 따라서 홀수차 정점이 정확히 2개이므로 오일러 트레일이 존재한다.
④ 바깥·중간·안쪽 세 정사각형이 하나의 수직 중심선으로 꿰어진 구조이다. 이 중심선은 각 정사각형의 위쪽 중간 정점과 아래쪽 중간 정점을 차례로 지나가는데, 중간 정사각형과 안쪽 정사각형의 위·아래 중간 정점은 자신이 속한 정사각형의 좌우 변과 중심선의 위아래 구간까지 합쳐 차수가 4(짝수)가 된다. 반면 가장 바깥 정사각형의 위쪽 중간 정점은 바깥 정사각형의 좌우 변 2개와 중심선 아래쪽 구간 1개만 가져 차수가 3(홀수)이고, 바깥 정사각형의 아래쪽 중간 정점도 같은 이유로 차수가 3(홀수)이다. 나머지 정점은 모두 짝수 차수이므로, 홀수차 정점이 정확히 2개가 되어 오일러 트레일이 존재한다.

오답 포인트

②, ③, ④를 오답으로 고르는 경우는 대개 대각선이나 중첩된 도형 때문에 정점 차수를 잘못 세어, 실제로는 홀수차 정점이 2개인데 이를 놓치거나 반대로 짝수라고 오판하는 경우이다. 반대로 ①을 그냥 지나치는 경우는 좌측 열 가운데 정점에서 대각선 두 개가 동시에 뻗어 나가는 것을 하나로 착각해 차수를 4로 잘못 세는 경우인데, 실제로는 위쪽 지그재그와 아래쪽 지그재그가 각각 별도의 간선이므로 차수 5가 된다는 점, 그리고 그 결과 좌측 열의 세 분기점과 우측 열 가운데 정점까지 총 4개의 홀수차 정점이 생긴다는 점을 놓치지 않아야 한다.

다시 나오면 이렇게 판단

그림으로 제시된 그래프에서 오일러 트레일의 존재 여부를 묻는 문제는, 그래프를 통째로 보지 말고 정점 하나하나에 모인 간선의 개수(차수)를 순서대로 세어보는 방식으로 접근한다. 특히 여러 선이 교차하거나 대각선이 추가된 정점, 중첩된 도형이 하나의 선으로 이어진 접점(중심선이 지나가는 정점)을 우선적으로 확인한다. 세어본 홀수차 정점의 개수가 0개면 오일러 서킷은 있지만 (출발·도착이 다른) 오일러 트레일은 없고, 2개면 오일러 트레일이 존재하며, 4개 이상이면 오일러 트레일도 서킷도 존재하지 않는다고 판단한다.

NoteBase

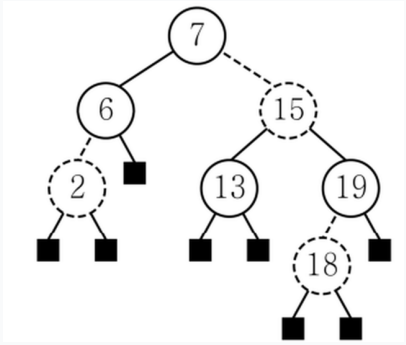
암기 포인트

오일러 트레일 존재 조건: 그래프가 연결되어 있고 홀수차 정점의 개수가 정확히 2개. 홀수차 정점 0개는 오일러 서킷(닫힌 경로)만 존재. 홀수차 정점이 1개이거나 3개 이상(짝수 개가 아닌 잘못된 경우)은 애초에 불가능하며, 4개 이상이면 오일러 트레일도 서킷도 없다. 모든 정점 차수의 합은 항상 짝수이므로 홀수차 정점의 개수는 항상 짝수 개(0, 2, 4, …)로 나온다.

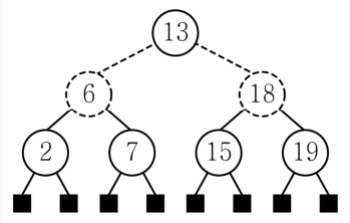
NoteBase

문 25. 비어 있는 레드-블랙(red-black) 트리에 <19, 18, 2, 15, 6, 13, 7>을 순서대로 입력한 결과 생성되는 레드-블랙 트리로 옳은 것은? (단, 점선 노드는 레드 노드, 점선 링크는 레드 링크이며, 실선 노드는 블랙 노드, 실선 링크는 블랙 링크이고, 리프 노드에 연결된 사각형의 단말은 외부 노드를 의미한다)

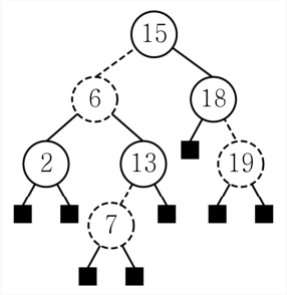
①



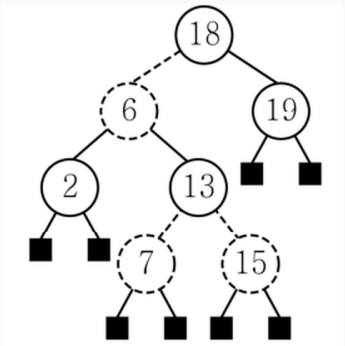
②



③



④



정답 4번

출제 포인트

레드-블랙 트리에 원소를 순서대로 삽입할 때 발생하는 이중 레드 위반(red-red violation)을 색상 재조정(recoloring)과 회전(rotation)으로 해소하는 표준 삽입 알고리즘을 정확히 적용할 수 있는지를 묻는다. 새 노드는 항상 레드로 삽입하며, 부모가 레드일 때만 위반이 발생하고 삼촌(uncle) 노드의 색에 따라 처리 방식(재색칠만 할지, 회전까지 할지)이 갈린다는 점이 핵심이다.

NoteBase

조건 정리

빈 트리에서 시작하여 19, 18, 2, 15, 6, 13, 7을 이 순서로 삽입한다. 각 삽입 시 새 노드는 레드로 만들고, 이진 탐색 트리 규칙(작으면 왼쪽, 크면 오른쪽)에 따라 위치를 정한 뒤 레드-블랙 성질(모든 리프에서 루트까지 블랙 높이 동일, 레드 노드의 자식은 모두 블랙)을 복구한다.

사용할 공식

삽입 위반 처리 규칙은 다음과 같다. 새 노드 z의 부모가 블랙이면 그대로 종료한다. 부모가 레드이면 삼촌 y를 확인한다. y가 레드이면 부모와 삼촌을 블랙으로, 조부모를 레드로 재색칠하고 조부모를 새로운 z로 삼아 위쪽으로 계속 검사한다(z가 루트에 닿으면 루트를 블랙으로 고정). y가 블랙(또는 외부 노드)이면 z와 부모, 조부모가 이루는 모양이 '일직선(zig)'인지 '꺾임(zig-zag)'인지에 따라 회전한다. 꺾임이면 부모를 기준으로 먼저 한 번 회전해 일직선으로 바꾸고, 그다음 조부모를 기준으로 반대 방향 회전을 하면서 새 서브트리 루트를 블랙으로, 원래 조부모를 레드로 바꾼다.

계산 과정

1) 19 삽입: 빈 트리이므로 루트, 블랙으로 확정. 트리: 19(B).
2) 18 삽입: 19의 왼쪽에 레드로 삽입. 부모 19가 블랙이므로 위반 없음. 트리: 19(B)-왼쪽 18(R).
3) 2 삽입: 18의 왼쪽에 레드로 삽입. 부모 18(R)-조부모 19(B)-삼촌(19의 오른쪽, 외부 노드=블랙). 일직선(왼쪽-왼쪽) 모양이므로 조부모 19를 기준으로 오른쪽 회전하고 18을 블랙, 19를 레드로 바꾼다. 트리: 18(B)-왼쪽 2(R), 오른쪽 19(R).
4) 15 삽입: 18→2(R)의 오른쪽에 레드로 삽입. 부모 2(R)-조부모 18(B)-삼촌 19(R). 삼촌이 레드이므로 부모 2와 삼촌 19를 블랙으로, 조부모 18을 레드로 재색칠하고 18을 새 z로 검사하는데 18이 루트이므로 루트를 블랙으로 고정. 트리: 18(B)-왼쪽 2(B, 오른쪽 자식 15(R))-오른쪽 19(B).
5) 6 삽입: 18→2→15(R)의 왼쪽에 레드로 삽입. 부모 15(R)-조부모 2(B)-삼촌(2의 왼쪽, 외부 노드=블랙). 15는 2의 오른쪽 자식인데 새 노드 6은 15의 왼쪽 자식이므로 꺾임(우-좌) 모양이다. 먼저 15를 기준으로 오른쪽 회전해 6이 위로 올라오게 하고, 이어서 2를 기준으로 왼쪽 회전하면서 6을 블랙, 2를 레드로 바꾼다. 트리: 18(B)-왼쪽 6(B, 왼쪽 2(R)·오른쪽 15(R))-오른쪽 19(B).
6) 13 삽입: 18→6→15(R)의 왼쪽에 레드로 삽입. 부모 15(R)-조부모 6(B)-삼촌(6의 왼쪽 2, 레드). 삼촌이 레드이므로 부모 15와 삼촌 2를 블랙으로, 조부모 6을 레드로 재색칠하고 6을 새 z로 검사한다. 6의 부모 18이 블랙이므로 종료. 트리: 18(B)-왼쪽 6(R, 왼쪽 2(B)·오른쪽 15(B, 왼쪽 자식 13(R)))-오른쪽 19(B).
7) 7 삽입: 18→6→15→13(R)의 왼쪽에 레드로 삽입. 부모 13(R)-조부모 15(B)-삼촌(15의 오른쪽, 외부 노드=블랙). 13은 15의 왼쪽 자식이고 새 노드 7도 13의 왼쪽 자식이므로 일직선(좌-좌) 모양이다. 조부모 15를 기준으로 오른쪽 회전하고 13을 블랙, 15를 레드로 바꾼다. 그 결과 13이 6의 오른쪽 자리로 올라오고, 13의 왼쪽 자식은 7(레드), 오른쪽 자식은 15(레드)가 된다.
최종 트리: 루트 18(블랙), 왼쪽 자식 6(레드), 오른쪽 자식 19(블랙, 외부 노드 2개). 6의 왼쪽 자식 2(블랙, 외부 노드 2개), 오른쪽 자식 13(블랙). 13의 왼쪽 자식 7(레드, 외부 노드 2개), 오른쪽 자식 15(레드, 외부 노드 2개).

NoteBase

선지 판단

① X → 루트가 7로 되어 있는데, 삽입 순서를 따라가면 루트는 마지막까지 18로 유지된다(7은 마지막에 삽입된 값으로 트리 맨 아래쪽에 위치해야 한다). 실제 계산 결과와 전혀 다른 구조이다.
② X → 루트가 13으로 되어 있으나, 계산 과정상 13은 6의 오른쪽 자식으로 트리의 세 번째 레벨에 위치해야 한다. 또한 6과 18이 모두 레드로 표시되어 있는데, 레드 노드가 부모-자식으로 연속(6이 13의 자식이 아니라 루트 자식이면서 동시에 18도 레드)될 경우 레드-레드 위반이 남아 레드-블랙 성질을 만족하지 못한다.
③ X → 루트가 15로 되어 있으나, 계산상 15는 마지막 회전에서 13의 오른쪽 자식(레드)으로 내려가야 한다. 삽입 순서를 그대로 따라가면 15가 루트가 되는 경우는 발생하지 않는다.
④ O → 계산 과정과 정확히 일치한다. 루트 18(블랙)-왼쪽 6(레드)-오른쪽 19(블랙, 외부 노드 2개), 6의 왼쪽 2(블랙, 외부 노드 2개)·오른쪽 13(블랙), 13의 왼쪽 7(레드, 외부 노드 2개)·오른쪽 15(레드, 외부 노드 2개)로 모든 노드 색과 위치가 삽입 순서를 그대로 시뮬레이션한 결과와 맞아떨어진다.

오답 포인트

레드-블랙 트리 문제에서 자주 틀리는 지점은 두 가지이다. 첫째, 삼촌 노드의 색을 확인하지 않고 무조건 회전부터 하거나, 반대로 삼촌이 블랙인데도 재색칠만으로 끝내는 실수이다. 둘째, 꺾임(zig-zag) 모양에서 회전을 한 번만 하고 끝내거나 회전 방향을 반대로 적용하는 실수이다. 이번 문항의 5단계(6 삽입)와 7단계(7 삽입)가 각각 꺾임과 일직선 회전을 요구하므로 이 두 유형을 혼동하면 오답에 이른다.

다시 나오면 이렇게 판단

삽입 순서가 주어지면 표로 만들지 말고 한 번에 한 노드씩, (1) BST 규칙으로 위치 찾기 → (2) 레드로 삽입 → (3) 부모 색 확인 → (4) 부모가 레드면 삼촌 색 확인 → (5) 삼촌 레드면 재색칠 후 조부모부터 재검사, 삼촌 블랙(외부 노드 포함)이면 모양(일직선/꺾임) 판별 후 회전, 순서로 기계적으로 진행한다. 마지막에는 반드시 루트를 블랙으로 고정했는지 확인한다.

암기 포인트

새 노드는 항상 레드로 삽입한다. 부모가 블랙이면 끝, 부모가 레드이고 삼촌이 레드이면 재색칠(부모·삼촌 블랙, 조부모 레드) 후 위로 재검사, 삼촌이 블랙이면 꺾임은 이중 회전, 일직선은 단일 회전 후 회전축 색을 블랙, 원래 조부모를 레드로 바꾼다. 루트는 항상 블랙이다.

NoteBase