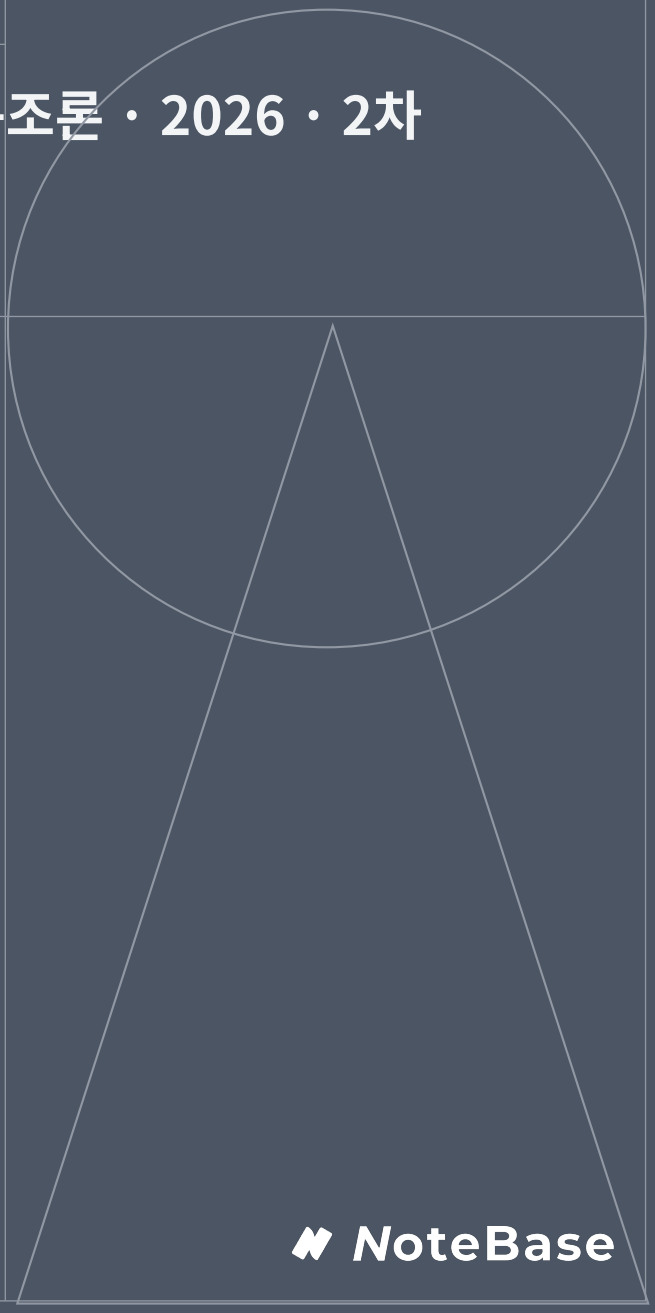
기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2026 · 2차



NoteBase

기출OX 해설집

공무원/국가직/7급 · 자료구조론 · 2026 · 2차

공무원/국가직/7급 · 자료구조론 · 2026 · 2차

문 1. 포인터 변수 L이 단순 연결 리스트의 첫 번째 노드를 가리킬 때, (가) ~ (다)를 수행하는 각 알고리즘의 시간 복잡도를 바르게 연결한 것은? (단, n은 리스트의 노드 개수를 나타낸다)

(가) L이 가리키는 리스트의 노드 순서를 역순으로 변환
(나) 새로운 노드를 L이 가리키는 리스트의 첫 번째 노드가 되도록 삽입
(다) L이 가리키는 리스트의 마지막 노드 삭제

① (가): $O(1)$, (나): $O(1)$, (다): $O(1)$
② (가): $O(1)$, (나): $O(n)$, (다): $O(n)$
③ (가): $O(n)$, (나): $O(1)$, (다): $O(n)$
④ (가): $O(n)$, (나): $O(n)$, (다): $O(1)$

정답 3번

간단한 해설

첫 노드 앞에 삽입할 때는 **정해진 수의 포인터 변경**만 필요하지만, 역순 변환과 마지막 노드 삭제에는 **리스트 순회**가 필요하다.

NoteBase

대응 관계 해설

단순 연결 리스트의 각 노드는 다음 노드를 가리키는 링크를 가지며, L은 첫 번째 노드를 가리킨다. 시간 복잡도는 노드 수 n이 늘어날 때 필요한 연산 횟수가 어떻게 증가하는지로 판단한다.

(가) 역순 변환: $O(n)$. 각 노드의 링크가 이전 노드를 가리키도록 바꾸어야 하므로 **모든 노드를 한 번씩 방문**한다. 이때 다음에 방문할 노드를 미리 저장한 뒤 현재 노드의 링크 방향을 바꾸면 된다. 예를 들어 A → B → C를 C → B → A로 바꾸고 L이 C를 가리키게 한다. 노드 하나당 일정한 수의 연산을 n번 수행하므로 전체 시간은 n에 비례한다.

(나) 첫 노드로 삽입: $O(1)$. 새 노드의 링크가 기존 L이 가리키던 노드를 가리키게 한 다음, L이 새 노드를 가리키도록 바꾼다. **기존 노드를 순회할 필요가 없고** 노드 수와 관계없이 일정한 수의 포인터만 변경한다.

(다) 마지막 노드 삭제: $O(n)$. 마지막 노드를 제거하려면 **마지막 노드의 바로 앞 노드**를 찾아 그 링크를 null로 바꾸어야 한다. 단순 연결 리스트에서는 이전 노드로 되돌아갈 수 없으므로 L에서 시작해 링크를 따라가며 바로 앞 노드를 찾는다. 이 탐색에 노드 수에 비례하는 시간이 걸리며, 이후 링크 변경과 노드 제거 자체는 일정한 시간이 걸린다.

따라서 (가) $O(n)$, (나) $O(1)$, (다) $O(n)$의 대응이며, 이는 ③과 일치한다.

NoteBase

문 2. 다음은 같은 기능을 수행하는 두 함수를 구현한 C 코드이다. 각 함수의 시간 복잡도를 바르게 연결한 것은?

```
double p1(double x, int n) {
   int i;
   double result = 1.0;

   for(i = 0; i < n; i++)
      result = result * x;
   return result;
}

double p2(double x, int n) {
   if(n == 0)
      return 1;
   else if((n % 2) == 0)
      return p2(x * x, n/2);
   else
      return x * p2(x * x, (n - 1)/2);
}
```

① p1 함수: $O(n)$, p2 함수: $O(1)$

② p1 함수: $O(n)$, p2 함수: $O(\log n)$

③ p1 함수: $O(2^n)$, p2 함수: $O(n^2)$

④ p1 함수: $O(\log n)$, p2 함수: $O(n\log n)$

정답 2번

간단한 해설

p1은 곱셈을 **n번 반복**하므로 시간 복잡도가 $O(n)$이다. p2는 매 단계에서 **재귀 호출을 한 번만 하면서 n을 절반으로 줄이므로** 시간 복잡도가 $O(\log n)$이다.

NoteBase

대응 관계 해설

n이 0 이상의 정수인 경우를 대상으로, 곱셈·나눗셈·비교 등 기본 연산 한 번의 시간을 상수로 보고 분석한다. 시간 복잡도는 함수가 계산하는 값의 크기가 아니라 n이 커질 때 필요한 연산 횟수의 증가 정도를 나타낸다.

p1 → $O(n)$: i가 0부터 n보다 작은 동안 1씩 증가하므로 반복문 본문은 정확히 n번 실행된다. 매번 result에 x를 한 번 곱하며, 초기화와 반환은 상수 시간이 걸린다. 따라서 **전체 실행 시간은 반복 횟수 n에 비례**한다.

p2 → $O(\log n)$: n이 0이면 바로 반환한다. n이 양의 짝수이면 다음 호출의 n은 $\frac{n}{2}$, 홀수이면 $\frac{n-1}{2}$이므로, 어느 경우든 다음 n은 $\lfloor \frac{n}{2} \rfloor$이다. 두 분기는 조건에 따라 하나만 실행되므로 **각 단계에서 재귀 호출은 한 번뿐**이다. x를 제곱하거나 반환값에 x를 곱하는 작업은 단계마다 상수 시간이 걸린다. 실행 시간을 $T(n)$이라 하면 $T(n) = T(\lfloor \frac{n}{2} \rfloor) + O(1)$이다. 예를 들어 n이 8이면 $8 \to 4 \to 2 \to 1 \to 0$으로 줄어든다. 이처럼 0에 도달할 때까지 절반으로 줄이는 단계 수가 로그 규모이므로 전체 시간 복잡도는 $O(\log n)$이다.

따라서 p1의 $O(n)$과 p2의 $O(\log n)$을 연결한 ②에 해당한다.

NoteBase

문 3. 다음은 이중 연결 리스트(doubly linked list)를 구현한 C 코드이다. 노드 삭제 함수의 빈칸에 들어갈 내용으로 옳은 것은?

```
typedef int element;
typedef struct DListNode {
   element data;
   struct DListNode *llink;
   struct DListNode *rlink;
} DListNode;
DListNode *head = NULL;

void init() {
   head = (DListNode*)malloc(sizeof(DListNode));
   head->llink = head->rlink = head;
}

void dinsert(DListNode *inode, DListNode *inserted) {
   inserted->llink = inode;
   inserted->rlink = inode->rlink;
   inode->rlink->llink = inserted;
   inode->rlink = inserted;
}

void ddelete(DListNode *removed) {
   if(removed == head) return;
   removed->llink->rlink = removed->rlink;
   [                    ]
   free(removed);
}
```

① removed->rlink->llink = removed->llink;

② removed->rlink->llink = removed->rlink;

③ removed->llink = removed->rlink->llink;

④ removed->rlink = removed->rlink->llink;

정답 1번

간단한 해설

삭제할 노드의 앞뒤 노드를 직접 연결하려면 **후행 노드의 llink가 선행 노드를 가리키게** 해야 한다. 앞선 문장이 선행 노드의 rlink를 이미 바꾸었으므로, 빈칸에서는 반대 방향의 연결을 고친다.

NoteBase

공통 해설

llink는 선행 노드를, rlink는 후행 노드를 가리킨다. 삭제할 노드를 B, 그 선행 노드를 A, 후행 노드를 C라고 하면, B를 제거한 뒤에는 A의 rlink가 C를 가리키고 C의 llink가 A를 가리켜야 한다. 제시된 코드는 head를 삭제하려는 경우 바로 반환하므로, 나머지 경우에는 이러한 연결을 고친 뒤 removed의 메모리를 해제한다.

빈칸 앞의 removed->llink->rlink = removed->rlink;는 A의 rlink를 C로 바꾸는 문장이다. 이 문장은 A의 필드만 수정하므로 removed->llink와 removed->rlink에는 여전히 각각 A와 C의 주소가 남아 있다. 따라서 빈칸은 **C의 llink에 A의 주소를 대입**하는 removed->rlink->llink = removed->llink;가 된다.

대입문의 왼쪽은 수정할 필드이므로 후행 노드의 llink여야 하고, 오른쪽은 새로 가리킬 선행 노드의 주소여야 한다. 후행 노드 자신의 주소를 넣으면 C의 llink가 C 자신을 가리키며, removed의 llink나 rlink만 바꾸면 C에서 B로 향하는 연결이 남는다. 양쪽 연결을 모두 고친 다음 free(removed)를 실행해야 남은 노드가 해제된 B를 가리키지 않는다.

NoteBase

문 4. 2차원 배열 A[1:15][1:20]의 원소 A[10][8]에 대하여, 행 우선 순서 주소와 열 우선 순서 주소를 바르게 연결한 것은? (단, 첫 번째 원소 A[1][1]의 주소는 2000이고, 각 원소의 크기는 1바이트이다)

① 행 우선 순서 주소: 2114, 열 우선 순서 주소: 2265

② 행 우선 순서 주소: 2182, 열 우선 순서 주소: 2187

③ 행 우선 순서 주소: 2187, 열 우선 순서 주소: 2114

④ 행 우선 순서 주소: 2265, 열 우선 순서 주소: 2182

정답 3번

간단한 해설

행 우선 순서는 **한 행의 20개 원소**를, 열 우선 순서는 **한 열의 15개 원소**를 먼저 연속 저장한다. A[10][8] 앞에는 각각 187개와 114개의 원소가 있으므로, 원소당 1바이트일 때 주소는 각각 2187과 2114이다.

계산 과정

배열 A[1:15][1:20]은 15행 20열이며, 행과 열의 첨자는 모두 1부터 시작한다. A[1][1]의 주소가 2000이고 원소 하나가 1바이트이므로, 목표 원소 앞에 저장되는 원소 수에 1바이트를 곱한 값을 시작 주소에 더한다.

행 우선 순서: **한 행의 모든 열을 저장한 뒤 다음 행**으로 넘어간다. A[10][8] 앞에는 1~9행의 원소 $9 \times 20 = 180$개와 10행의 1~7열 원소 7개가 있다. 따라서 앞선 원소는 $180 + 7 = 187$개이고, 주소는 $2000 + 187 \times 1 = 2187$이다.

열 우선 순서: **한 열의 모든 행을 저장한 뒤 다음 열**로 넘어간다. A[10][8] 앞에는 1~7열의 원소 $7 \times 15 = 105$개와 8열의 1~9행 원소 9개가 있다. 따라서 앞선 원소는 $105 + 9 = 114$개이고, 주소는 $2000 + 114 \times 1 = 2114$이다.

행 우선 순서 주소 2187, 열 우선 순서 주소 2114의 연결은 ③과 일치한다.

NoteBase

문 5. 다음은 연결 리스트를 이용하여 스택을 구현한 C 코드이다. (가), (나)에 들어갈 내용을 바르게 연결한 것은?

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

node *head = NULL;
node *tail = NULL;

void init() {
   head = (node*)malloc(sizeof(node));
   tail = (node*)malloc(sizeof(node));
   head->next = tail;
   tail->next = tail;
}

int push(int k) {
   node *t;
   if((t = (node*)malloc(sizeof(node))) == NULL) {
      printf("\n Out of memory");
      return -1;
   }
   t->key = k;
   [          (가)          ]
   head->next = t;
   return k;
}

int pop() {
   node *t;
   int i;
   if(head->next == tail) {
      printf("\n Stack underflow");
      return -1;
   }
   t = head->next;
   i = t->key;
   [          (나)          ]
   free(t);
   return i;
}
```

① (가): t->next = head;, (나): head->next = t->next;

② (가): t->next = head->next;, (나): head->next = t->next;

③ (가): t->next = head->next;, (나): head = t->next;

④ (가): t->next = head->next;, (나): head->next = t;

정답 2번

간단한 해설

이 코드는 **head의 다음 노드**를 스택의 맨 위로 사용한다. 삽입할 때는 새 노드가 기존 맨 위 노드를 가리키게 하고, 삭제할 때는 head의 다음 링크가 삭제할 노드의 다음 노드를 가리키게 한다.

NoteBase

빈칸별 해설

init()에서 head와 tail을 별도로 만들고 head->next = tail;로 연결하므로, 두 노드는 자료의 앞뒤 경계를 표시하는 노드이다. 실제 자료가 있으면 head->next가 스택의 맨 위 노드를 가리키고, head->next == tail이면 빈 스택이다. 삽입과 삭제는 머리 노드 head를 유지하면서 그 다음 링크를 바꿔 수행한다.

(가): t->next = head->next;를 넣는다. 새 노드 t의 다음 링크에 **기존 맨 위 노드의 주소를 먼저 저장**한 뒤, 이어지는 head->next = t;로 새 노드를 맨 위에 연결한다. 예를 들어 기존 연결이 head → A → tail이면 삽입 후에는 head → t → A → tail이 된다. t->next = head;를 넣으면 새 노드가 머리 노드를 가리켜 head와 t 사이에 순환이 생기고, 기존 자료로 이어지는 연결도 보존되지 않는다.

(나): head->next = t->next;를 넣는다. 앞의 t = head->next;에서 t는 삭제할 맨 위 노드를 가리키고, i = t->key;에서 반환할 값을 보관한다. 이제 **head의 다음 링크를 t 다음 노드로 변경**하여 t를 연결에서 제외한 뒤 free(t);로 메모리를 해제한다. head → B → A → tail이었다면 head → A → tail이 되어 B가 먼저 꺼내진다. head = t->next;는 머리 노드 포인터 자체를 바꾸므로 이 구조를 유지하지 못한다. head->next = t;는 삭제할 노드를 계속 가리키게 하므로, free(t); 이후 해제된 메모리를 가리키는 링크가 남는다.

(가)의 t->next = head->next;와 (나)의 head->next = t->next;를 함께 제시한 선택지는 ②이다.

NoteBase

문 6. 다음 C 언어로 작성된 함수들의 시간 복잡도에 대한 설명으로 옳은 것은?

```
void func1(int n) {
   for(int i = 0 ; i < 2 ; i++)
      for(int j = 0 ; j < n ; j++)
         printf("%d\n", j);
}

void func2(int n) {
   for(int i = 0 ; i < 100 ; i++)
      printf("%d\n", i);
}

void func3(int n) {
   if(n == 0)
      return;
   printf("%d\n", n);
      func3(n/2); ····· (가)
}
```

① func1 함수는 이중 for 반복문으로 구성되어 시간 복잡도는 $O(n^2)$이다.

② func2 함수는 for 반복문으로 구성되어 시간 복잡도는 O(n)이다.

③ func3 함수는 for 반복문이 없어 시간 복잡도는 O(1)이다.

④ (가) 문장을 func2(n/2);으로 수정할 경우, func3 함수의 시간 복잡도는 O(1)이다.

정답 4번

간단한 해설

④처럼 (가)를 func2(n/2);로 바꾸면 **자기 자신을 다시 호출하지 않는다**. func2는 전달받은 n의 값과 관계없이 100번만 반복하므로, 수정된 func3의 시간 복잡도는 $O(1)$이다.

NoteBase

선지별 해설

시간 복잡도는 반복문의 개수보다 입력 크기 n에 따른 전체 실행 횟수의 증가 양상으로 판단한다. 여기서는 n을 0 이상의 정수로 보고, 기본 연산과 한 번의 출력에 드는 비용을 상수로 본다.

① X: func1의 바깥쪽 반복문은 i가 0, 1일 때 **항상 2번만 반복**한다. 안쪽 반복문은 매번 n번 실행되므로 출력 횟수는 $2n$이고, 시간 복잡도는 $O(n)$이다. 이중 반복문이라도 두 반복 횟수가 모두 n에 비례하는 것은 아니므로 $O(n^2)$으로 판단할 수 없다.

② X: func2는 i가 0부터 99까지 변하면서 **입력 크기와 무관하게 100번 반복**한다. 매개변수 n은 반복 조건에 쓰이지 않으므로, 시간 복잡도는 $O(n)$이 아니라 $O(1)$이다.

③ X: 원래 func3는 (가)에서 **인자를 절반으로 줄여 자기 자신을 호출**한다. C 언어의 정수 나눗셈에 따라 소수 부분이 버려지며, 인자가 0이 되면 종료한다. 예를 들어 n이 8이면 인자는 $8 \to 4 \to 2 \to 1 \to 0$으로 변한다. 각 호출에서 상수 시간의 작업을 하고 호출 횟수는 n의 로그에 비례하므로, 시간 복잡도는 $O(1)$이 아니라 $O(\log n)$이다. for문이 없어도 재귀 호출에 드는 시간을 포함해야 한다.

④ O: (가)를 func2(n/2);로 바꾸면 func3는 **재귀 호출 없이 func2를 한 번 호출**한다. n이 0이면 바로 종료하고, 0이 아니면 n을 한 번 출력한 뒤 func2를 실행한다. func2는 인자로 전달된 값과 무관하게 100번만 반복하므로, 수정된 func3의 전체 실행 횟수는 n이 커져도 일정하며 시간 복잡도는 $O(1)$이다.

NoteBase

문 7. 다음은 원형 연결 리스트의 메모리 구성이다. 메모리 주소 1500인 위치에 data와 link 항목 내용이 각각 F와 1400인 새 노드를 삽입할 때, 변경되는 내용으로 옳은 것은?

메모리 주소	data	link
1000	A	1300
1100	B	1400
1200	C	1000
1300	D	1100
1400	E	1200

① data 값이 A인 노드의 link 값을 1400으로 변경

② data 값이 B인 노드의 link 값을 1500으로 변경

③ data 값이 C인 노드의 link 값을 1400으로 변경

④ data 값이 E인 노드의 link 값을 1500으로 변경

정답 2번

간단한 해설

새 노드 F의 link가 1400이므로 F는 E 바로 앞에 삽입된다. 기존에 E를 가리키던 **B의 link를 1500으로 변경**해야 B → F → E로 연결된다.

공통 해설

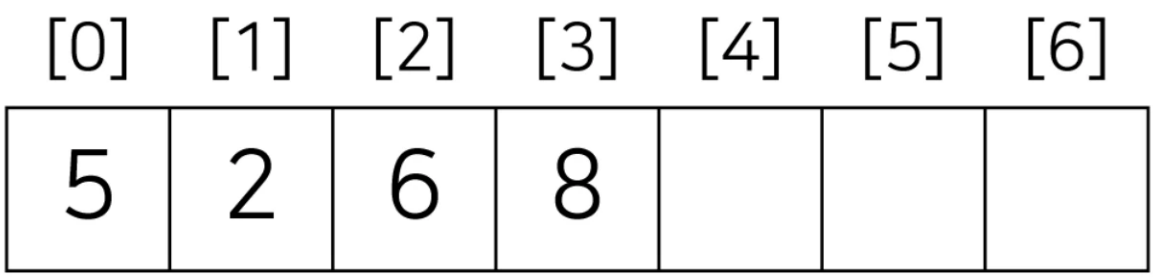
연결 리스트의 link는 다음 노드의 메모리 주소를 저장한다. 새 노드를 두 노드 사이에 삽입할 때는 새 노드가 뒤 노드를 가리키게 하고, **앞 노드의 link가 새 노드를 가리키도록** 바꾼다. 노드의 연결 순서는 메모리 주소의 크기순으로 정해지지 않는다.

표에서 주소 1400의 노드는 E이고, B의 link가 1400이므로 현재 B → E로 연결되어 있다. 주소 1500에 놓이는 새 노드 F의 link가 1400으로 주어졌으므로 F는 E를 가리킨다. 따라서 B의 link를 1400에서 1500으로 바꾸면 기존의 B → E 연결이 B → F → E로 바뀐다.

삽입 후 A에서 link를 따라가면 A → D → B → F → E → C → A로 돌아와 원형 연결이 유지된다. 삽입 위치의 앞 노드인 B만 기존 link를 변경하므로, A·C·E의 link는 각각 1300·1000·1200을 그대로 유지한다.

NoteBase

문 8. 다음은 크기가 7인 배열로 구현된 원형 덱(deque)의 상태이다. 현재 상태에서 연산 1부터 연산 5까지 순서대로 수행하였을 때, 수행이 완료된 후 원형 덱의 상태는? (단, 현재 상태는 front = 0, rear = 3이다)



연산 1: 맨 뒤에 위치한 데이터 삭제
연산 2: 맨 앞에 위치한 데이터 삭제
연산 3: 맨 뒤에 데이터 3을 삽입
연산 4: 맨 앞에 데이터 1을 삽입
연산 5: 맨 앞에 데이터 4를 삽입

① [0] [1] [2] [3] [4] [5] [6]
1 2 6 3 4

② [0] [1] [2] [3] [4] [5] [6]
4 1 2 6 3

③ [0] [1] [2] [3] [4] [5] [6]
1 2 6 3 4

④ [0] [1] [2] [3] [4] [5] [6]
4 1 2 6 3

정답 1번

간단한 해설

앞쪽에 1을 넣으면 front가 0이 되고, 다시 앞쪽에 4를 넣을 때는 **인덱스 0에서 6으로 순환**한다. 따라서 4는 [6]에, 1은 [0]에 저장되며 rear는 3이다.

NoteBase

계산 과정

초기 상태: 덱은 양쪽 끝에서 삽입과 삭제가 가능한 자료구조이다. 그림에서 front = 0은 첫 데이터 5의 위치이고, rear = 3은 마지막 데이터 8의 위치이다. 앞에서 뒤로 읽은 데이터는 5 → 2 → 6 → 8이다. 배열 크기가 7이므로 인덱스 범위는 0~6이며, **0의 앞은 6, 6의 뒤는 0**으로 이어진다. 삽입·삭제 시 나머지 데이터를 옮기지 않고 해당 끝의 위치를 갱신한다.

연산 1: 맨 뒤의 데이터 8을 [3]에서 삭제한다. rear는 3에서 2로 이동하고 front는 0을 유지한다. 남은 데이터는 5 → 2 → 6이다.

연산 2: 맨 앞의 데이터 5를 [0]에서 삭제한다. front는 0에서 1로 이동하고 rear는 2를 유지한다. 남은 데이터 2와 6은 각각 **원래 위치 [1]과 [2]**에 그대로 있다.

연산 3: 맨 뒤에 데이터 3을 넣으므로 rear를 2에서 3으로 이동시킨 뒤 [3]에 3을 저장한다. front는 1이며, 데이터 순서는 2 → 6 → 3이다.

연산 4: 맨 앞에 데이터 1을 넣으므로 front를 1에서 0으로 이동시킨 뒤 [0]에 1을 저장한다. rear는 3이며, 데이터 순서는 1 → 2 → 6 → 3이다.

연산 5: 맨 앞에 데이터 4를 넣으려면 front = 0보다 한 칸 앞을 사용해야 한다. 원형 배열이므로 **[0]의 앞인 [6]**에 4를 저장하고 front를 6으로 갱신한다. rear는 3을 유지하며, 앞에서 뒤로 읽은 데이터는 4 → 1 → 2 → 6 → 3이다.

최종 상태: 배열의 [0], [1], [2], [3], [6]에는 각각 1, 2, 6, 3, 4가 저장되고 [4], [5]는 비어 있다. front = 6, rear = 3인 이 상태는 ①의 그림과 일치한다.

NoteBase

문 9. 다음 C 프로그램의 실행 결과는?

```
#include <stdio.h>

void function(int array[], int idx, int size) {
   for(int i = idx; i < size - 1; i++)
      array[i] = array[i + 1];
}

int main() {
   int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
   function(array, 5, 10);
   function(array, 5, 10);

   for(int i = 0; i < 10; i++)
      printf("%d ", array[i]);
   printf("\n");

   return 0;
}
```

① 0 1 2 3 4 5 6 7 8 9

② 0 1 2 3 4 7 8 9 9 9

③ 0 1 2 3 5 5 6 7 8 9

④ 0 1 2 3 5 6 7 8 9 9

정답 2번

간단한 해설

함수를 호출할 때마다 인덱스 5부터 8까지 **오른쪽 원소를 한 칸씩 왼쪽으로 복사**한다. 마지막 원소는 그대로 남고 배열 크기도 줄지 않으므로, 두 번 호출한 뒤에는 끝의 세 원소가 모두 9가 된다.

NoteBase

계산 과정

함수의 동작: function(array, 5, 10)에서 idx는 5, size는 10이다. 반복 조건은 i < 9이므로 i는 5, 6, 7, 8 순서로 변하고, array[i] = array[i + 1]에 따라 각 위치에 오른쪽 원소의 값을 복사한다. 작은 인덱스부터 처리하므로 복사할 오른쪽 값은 아직 덮어쓰이지 않은 상태이다. **인덱스 0~4와 마지막 인덱스 9의 값은 바뀌지 않는다.** 함수 안에서 배열 원소를 바꾸면 main의 원래 배열에도 그 변경이 반영된다.

첫 번째 호출: 처음 배열은 0 1 2 3 4 5 6 7 8 9이다. 인덱스 5~8에 각각 인덱스 6~9의 값인 6, 7, 8, 9를 복사한다. 따라서 첫 번째 호출 후 배열은 0 1 2 3 4 6 7 8 9 9가 된다. 마지막 위치의 9를 지우는 연산은 없으므로 9가 두 개 남는다.

두 번째 호출: **첫 번째 호출로 바뀐 배열**에 같은 연산을 다시 적용한다. 이때 인덱스 6~9의 값은 7, 8, 9, 9이므로 이를 인덱스 5~8에 차례로 복사한다. 마지막 인덱스 9의 값은 여전히 9이므로 배열은 0 1 2 3 4 7 8 9 9 9가 된다.

출력: 출력 반복문은 i가 0부터 9까지인 동안 실행되므로 배열의 10개 원소를 모두 출력한다. 출력되는 수열은 0 1 2 3 4 7 8 9 9 9이며, 이는 ②의 수열과 일치한다. 각 수 뒤에는 공백이 붙고, 반복문이 끝나면 줄을 바꾼다.

NoteBase

문 10. 이진 탐색 트리의 모든 키를 오름차순으로 정렬한 결과를 일차원 배열 형식으로 출력하기 위해 사용할 수 있는 순회 알고리즘은? (단, 순회 알고리즘은 1회만 수행하고 별도의 정렬 알고리즘은 수행하지 않는다)

① 중위 순회

② 전위 순회

③ 후위 순회

④ 레벨 순회

정답 1번

간단한 해설

이진 탐색 트리는 각 노드의 왼쪽 서브트리에 작은 키, 오른쪽 서브트리에 큰 키를 둔다. **중위 순회**는 왼쪽 서브트리 → 현재 노드 → 오른쪽 서브트리 순서로 방문하므로 키가 오름차순으로 출력된다.

공통 해설

이진 탐색 트리에서는 각 노드를 기준으로 왼쪽 서브트리의 모든 키가 그 노드의 키보다 작고, 오른쪽 서브트리의 모든 키가 크다. 따라서 작은 키들, 현재 키, 큰 키들 순서로 출력하려면 **왼쪽 서브트리 → 현재 노드 → 오른쪽 서브트리** 순서로 방문해야 한다. 이 순서를 각 서브트리에도 재귀적으로 적용하는 알고리즘이 중위 순회이다.

예를 들어 루트의 키가 2이고 왼쪽 자식이 1, 오른쪽 자식이 3인 이진 탐색 트리를 중위 순회하면 1 → 2 → 3으로 출력된다. 같은 트리의 전위 순회는 2 → 1 → 3, 후위 순회는 1 → 3 → 2, 레벨 순회는 2 → 1 → 3이므로, 이들은 이진 탐색 트리의 키를 항상 오름차순으로 출력하지는 않는다.

중위 순회에서 각 노드를 방문할 때 그 키를 배열의 다음 칸에 차례로 저장하면 된다. 모든 노드를 한 번씩 방문하는 **한 차례의 순회만으로** 오름차순 배열을 얻으므로 별도의 정렬 알고리즘이 필요하지 않다.

NoteBase

문 11. 정점의 개수가 n인 단순 그래프가 트리이기 위한 필요충분조건으로 옳지 않은 것은? (단, n은 3 이상의 정수이다)

① 사이클이 없고 연결요소의 개수가 1이다.

② 임의의 두 정점 사이에 존재하는 경로의 개수는 1이다.

③ 간선의 개수가 n−1이고, 차수가 1인 정점의 개수가 2 이상이다.

④ 사이클이 없지만, 인접하지 않은 임의의 두 정점 사이에 간선을 추가하면 사이클의 개수가 1이 된다.

정답 3번

간단한 해설

③의 조건은 트리의 **필요조건이지만 충분조건은 아니다**. 간선이 n−1개이고 차수가 1인 정점이 2개 이상이어도, 그래프가 여러 연결요소로 나뉘고 사이클을 포함할 수 있다.

NoteBase

선지별 해설

① O: 트리는 연결되어 있으면서 사이클이 없는 단순 그래프이다. 연결요소는 서로 경로로 연결되는 정점들의 묶음이므로, **연결요소가 1개**라는 것은 그래프 전체가 연결되어 있다는 뜻이다. 따라서 두 조건을 함께 만족하는 것은 트리의 정의와 일치한다.

② O: 여기서 경로는 정점을 중복해서 지나지 않는 단순 경로를 뜻한다. 임의의 두 정점 사이에 경로가 존재하므로 그래프는 연결되어 있다. 사이클이 있다면 그 위의 두 정점 사이에는 사이클을 양방향으로 따라가는 서로 다른 경로가 생긴다. 따라서 **경로가 정확히 하나**라는 조건은 사이클이 없다는 것도 보장한다. 반대로 트리에서 두 정점 사이에 서로 다른 경로가 있다면 사이클이 생기므로 경로는 유일하다.

③ X: 정점이 n개인 트리의 간선은 n−1개이고, n이 3 이상이면 차수가 1인 정점인 잎이 적어도 2개 있다. 트리의 가장 긴 단순 경로에서 양 끝 정점에 다른 이웃이 있으면 경로를 더 늘리거나 사이클을 만들게 되므로 두 끝은 모두 잎이다. 그러나 이 두 성질만으로는 **연결성이 보장되지 않는다**. 예를 들어 A·B·C가 삼각형을 이루고, 이들과 떨어진 D·E가 간선 하나로 연결된 그래프를 생각할 수 있다. 정점은 5개, 간선은 3 + 1 = 4개이고 차수가 1인 정점은 D·E의 2개이므로 제시된 조건을 모두 만족한다. 하지만 연결요소가 2개이고 삼각형 사이클도 있어 트리가 아니다. 필요충분조건으로 고치려면 간선이 n−1개라는 조건과 함께 그래프가 연결되어 있다는 조건을 두면 된다.

④ O: 트리에서는 두 정점 사이의 경로가 유일하다. 인접하지 않은 두 정점 사이에 간선을 하나 추가하면 **기존의 유일한 경로와 새 간선**이 결합하여 사이클이 정확히 하나 생긴다. 반대로 사이클이 없는 그래프가 연결되어 있지 않다면, 서로 다른 연결요소의 두 정점 사이에 간선을 추가해도 두 요소가 연결될 뿐 사이클은 생기지 않는다. 따라서 어떤 비인접 정점 쌍에 간선을 추가하더라도 사이클이 하나 생긴다는 조건은 연결성을 보장하며, 원래 사이클이 없다는 조건과 합쳐져 트리의 필요충분조건이 된다.

NoteBase

문 12. 다음과 같이 6개의 키값으로 구성된 4개의 입력 배열 데이터 (가)~(라)를 각각 순서대로 삽입하여 4개의 AVL 트리를 생성하였다. 생성된 모든 트리에 공통으로 존재하는 리프 노드 키값으로 옳은 것은?

(가)	4	1	2	8	5	9
(나)	1	2	8	5	9	4
(다)	9	8	4	2	1	5
(라)	9	8	1	5	4	2

① 2

② 4

③ 5

④ 9

정답 4번

간단한 해설

삽입 후 회전까지 반영하면 **9는 네 트리 모두에서 자식이 없는 리프 노드**로 남는다. 반면 4는 (다)·(라)에서 루트가 되고, 5는 (가)·(나)에서 루트가 되므로 공통 리프가 될 수 없다.

NoteBase

계산 과정

AVL 트리는 이진 탐색 트리의 순서를 유지하면서 모든 노드의 **왼쪽·오른쪽 부분 트리 높이 차이를 1 이하**로 유지한다. 키를 하나씩 삽입하고, 삽입 위치에서 루트 방향으로 올라가며 처음 불균형해진 노드를 기준으로 회전한다. LL형은 우회전, RR형은 좌회전으로 고친다. LR형은 왼쪽 자식을 좌회전한 뒤 해당 노드를 우회전하고, RL형은 오른쪽 자식을 우회전한 뒤 해당 노드를 좌회전한다. 리프 노드는 왼쪽·오른쪽 자식이 모두 없는 노드이다.

(가) 4 → 1 → 2 → 8 → 5 → 9: 2를 삽입하면 4에서 LR형 불균형이 생겨 2가 루트가 된다. 5를 삽입하면 4에서 RL형 회전이 일어나 5의 왼쪽 자식이 4, 오른쪽 자식이 8이 된다. 마지막 9를 삽입하면 루트 2에서 RR형 불균형이 생겨 좌회전한다. 최종 루트는 5이고, 왼쪽 자식 2의 두 자식은 1·4, 오른쪽 자식 8의 오른쪽 자식은 9이다. 따라서 리프 키값은 **{1, 4, 9}**이다.

(나) 1 → 2 → 8 → 5 → 9 → 4: 8을 삽입하면 1에서 RR형 불균형이 생겨 좌회전하고, 2가 루트가 된다. 5와 9는 각각 8의 왼쪽·오른쪽 자식이 된다. 마지막 4는 5의 왼쪽에 들어가므로 루트 2에서 RL형 불균형이 생긴다. 8을 우회전한 뒤 2를 좌회전하면 최종 구조는 (가)와 같아진다. 루트는 5이고, 2의 두 자식은 1·4, 8의 오른쪽 자식은 9이므로 리프 키값은 **{1, 4, 9}**이다.

(다) 9 → 8 → 4 → 2 → 1 → 5: 4를 삽입하면 9에서 LL형 불균형이 생겨 우회전하고, 8이 루트가 된다. 1을 삽입하면 4에서 LL형 회전이 일어나 2의 두 자식이 1·4가 된다. 마지막 5는 4의 오른쪽에 들어가므로 루트 8에서 LR형 불균형이 생긴다. 2를 좌회전한 뒤 8을 우회전하면 4가 최종 루트가 된다. 왼쪽 자식 2의 왼쪽 자식은 1이고, 오른쪽 자식 8의 두 자식은 5·9이므로 리프 키값은 **{1, 5, 9}**이다.

(라) 9 → 8 → 1 → 5 → 4 → 2: 1을 삽입하면 9에서 LL형 회전이 일어나 8이 루트가 된다. 4를 삽입하면 1에서 RL형 불균형이 생겨 4의 두 자식이 1·5가 된다. 마지막 2는 1의 오른쪽에 들어가므로 루트 8에서 LL형 불균형이 생겨 우회전한다. 최종 루트는 4이고, 왼쪽 자식 1의 오른쪽 자식은 2, 오른쪽 자식 8의 두 자식은 5·9이다. 따라서 리프 키값은 **{2, 5, 9}**이다.

네 리프 집합에 모두 들어 있는 키값은 9뿐이다. 2는 (가)에서, 4는 (다)에서, 5는 (가)에서 자식을 가지므로 공통 리프가 아니다. 공통 리프 키값 9는 ④에 해당한다.

NoteBase

문 13. 선형 조사법(linear probing)을 사용하는 해시 테이블에서 버킷(bucket)을 삭제할 때, 빈 버킷(empty bucket)으로 비워 두지 않고 삭제했음을 표시하는 이유는? (단, 버킷 내 슬롯(slot) 수는 1이다)

① 1차 군집 현상이 발생하지 않게 한다.

② 해시 테이블의 전체 크기를 자동으로 기존 크기의 두 배로 확장시킨다.

③ 충돌로 인해 원래 해시 주소와는 다른 주소에 저장된 레코드에 대한 탐색이 가능하게 한다.

④ 해시 테이블에 저장된 모든 레코드의 키값이 항상 오름차순으로 정렬된 상태가 되도록 유지시킨다.

정답 3번

간단한 해설

선형 조사법에서는 빈 버킷을 만나면 해당 키가 없다고 보고 탐색을 끝낸다. 삭제한 자리를 빈 버킷으로 만들면 그 뒤로 밀려 저장된 레코드를 놓칠 수 있으므로, **삭제 표시를 만나면 탐색을 계속**하도록 한다.

선지별 해설

선형 조사법은 해시 주소의 버킷이 차 있으면 다음 버킷을 차례로 조사하며, 테이블 끝에서는 처음으로 돌아가는 충돌 처리 방법이다. 이 문제에서는 버킷당 슬롯이 하나이므로 한 버킷에 레코드 하나만 저장한다. 탐색할 때도 같은 경로를 따라가며, 찾는 키를 만나면 성공하고 빈 버킷을 만나면 실패로 끝낸다.

① X: 1차 군집은 충돌한 레코드들이 연속된 버킷에 모이고, 그 구간으로 해시되는 레코드들이 뒤에 추가되면서 군집이 커지는 현상이다. 삭제 표시는 **1차 군집의 발생을 막는 장치가 아니다**. 그 목적은 삭제된 위치에서도 탐색 경로가 끊기지 않게 하는 것이다.

② X: 삭제 표시는 해당 버킷의 상태를 기록할 뿐, **테이블 크기를 변경하지 않는다**. 크기를 두 배로 늘리는 것은 별도의 확장 정책에 따른 재해싱으로 처리할 수 있으며, 삭제 표시 자체의 기능은 아니다.

③ O: 삭제 표시는 '레코드는 삭제되었지만 뒤에 찾는 키가 있을 수 있다'는 뜻이므로 **다음 버킷의 탐색을 계속**하게 한다. 예를 들어 A와 B의 해시 주소가 모두 2이고, A가 2번 버킷에, 충돌한 B가 3번 버킷에 저장되었다고 하자. A를 삭제하면서 2번을 빈 버킷으로 바꾸면 B를 찾을 때 2번에서 탐색이 끝나 B를 놓친다. 2번에 삭제 표시를 남기면 3번까지 조사하여 B를 찾을 수 있다.

④ X: 해시 테이블은 키의 대소 관계가 아니라 **해시값과 충돌 처리 규칙**에 따라 저장 위치를 정한다. 따라서 키값의 오름차순을 보장하지 않으며, 삭제 표시도 정렬을 유지하는 기능이 아니다. 삭제 표시는 충돌 때문에 원래 해시 주소 뒤에 저장된 레코드를 계속 찾게 하는 기능이다.

NoteBase

문 14. 다음 C 언어로 구현한 sort 함수에서 (가) 문장이 수행되는 횟수가 가장 작은 배열 arr의 입력내용은? (단, n = 5이다)

```
void sort(int arr[], int n) {
   int i, key, j;
   for(i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      while(j >= 0 && arr[j] < key) {
         arr[j + 1] = arr[j];  ……(가)
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
```

① 1, 2, 3, 4, 5

② 5, 3, 4, 7, 6

③ 8, 3, 10, 7, 2

④ 10, 9, 8, 7, 6

정답 4번

간단한 해설

while문의 조건은 arr[j] < key이므로 **key보다 작은 앞쪽 원소**만 오른쪽으로 이동한다. 이미 내림차순인 배열에서는 이 조건이 매번 거짓이 되어 **(가)가 0회** 수행된다.

NoteBase

선지별 해설

이 함수는 앞부분을 내림차순으로 유지하면서 key를 알맞은 위치에 넣는 삽입 정렬이다. n = 5이므로 i = 1, 2, 3, 4에서 삽입을 진행한다. 각 단계에서 **key보다 작은 앞쪽 원소를 한 칸씩 오른쪽으로 이동**할 때마다 (가)가 한 번 수행된다. while문 뒤의 arr[j + 1] = key는 (가)의 수행 횟수에 포함하지 않는다.

① 1, 2, 3, 4, 5: key가 2, 3, 4, 5일 때 앞부분의 모든 원소가 key보다 작다. 따라서 이동 횟수는 차례로 1회, 2회, 3회, 4회이고, (가)는 총 1 + 2 + 3 + 4 = 10회 수행된다.

② 5, 3, 4, 7, 6: key가 3일 때는 이동하지 않고, 4일 때는 3만 이동한다. 이어 key가 7일 때는 앞부분의 5, 4, 3을 이동하여 앞부분이 7, 5, 4, 3이 된다. 마지막 key인 6을 넣을 때는 5, 4, 3을 이동한다. 따라서 (가)는 총 0 + 1 + 3 + 3 = 7회 수행된다.

③ 8, 3, 10, 7, 2: key가 3일 때는 이동하지 않고, 10일 때는 8과 3을 이동하여 앞부분이 10, 8, 3이 된다. key가 7일 때는 3만 이동하고, 마지막 key인 2는 앞부분의 모든 원소보다 작으므로 이동이 없다. 따라서 (가)는 총 0 + 2 + 1 + 0 = 3회 수행된다.

④ 10, 9, 8, 7, 6: 이미 내림차순이므로 각 key 바로 앞의 원소가 key보다 크다. 매 단계에서 arr[j] < key가 처음부터 거짓이 되어 while문 본문에 들어가지 않는다. 따라서 (가)는 **총 0회** 수행된다.

(가)의 수행 횟수는 각각 10회, 7회, 3회, 0회이므로 가장 작은 입력은 ④이다.

NoteBase

문 15. 단순 그래프의 인접 행렬 표현이 인접 리스트 표현보다 점근적으로 빠른 연산으로 옳은 것은? (단, 인접 행렬에서 임의의 정점 v의 행 또는 열의 접근 시간과 인접 리스트에서 임의의 정점 v의 인접 리스트의 접근 시간은 모두 상수이다)

① 특정 정점의 차수 계산

② 그래프 전체의 간선 개수 세기

③ 특정 정점에서 시작하는 너비 우선 탐색

④ 특정 두 정점 사이의 간선 존재 여부 확인

정답 4번

간단한 해설

두 정점 사이의 간선은 인접 행렬의 **해당 원소 하나**만 읽으면 확인할 수 있어 $O(1)$ 시간이 걸린다. 인접 리스트는 한 정점의 이웃 목록에서 상대 정점을 찾아야 하므로, 최악에는 **목록 전체를 탐색**해야 한다.

NoteBase

선지별 해설

정점 수를 n, 간선 수를 m, 정점 v의 차수를 $d(v)$라 하자. 인접 행렬은 두 정점의 연결 여부를 행과 열이 만나는 원소에 저장하고, 인접 리스트는 각 정점에 연결된 이웃들을 목록으로 저장한다. 차수나 전체 간선 수를 별도로 저장하지 않은 기본 표현에서 최악의 경우를 비교한다. 문제의 상수 시간 접근 조건은 행이나 목록의 시작 위치를 얻는 데 적용되며, 그 안의 원소를 모두 읽는 시간까지 상수라는 뜻은 아니다.

① 특정 정점의 차수 계산: 인접 행렬에서는 해당 정점의 행에 있는 n개 원소를 조사하여 연결된 정점 수를 세므로 $O(n)$ 시간이 걸린다. 인접 리스트에서는 해당 목록의 항목만 세면 되므로 $O(1 + d(v))$ 시간이 걸린다. 단순 그래프에서 차수는 최대 $n - 1$이므로 인접 행렬이 점근적으로 더 빠르지 않다.

② 그래프 전체의 간선 개수 세기: 인접 행렬에서는 n^2개 원소를 조사하므로 $O(n^2)$, 인접 리스트에서는 정점별 목록과 그 항목들을 조사하므로 $O(n + m)$ 시간이 걸린다. 무방향 그래프에서는 같은 간선이 양쪽 정점에 저장되므로 중복을 제외해 세어야 하지만, 시간 복잡도의 차수는 달라지지 않는다. 희소 그래프에서는 **실제로 존재하는 간선만 저장하는 인접 리스트**가 유리하고, 간선이 많은 경우에도 인접 행렬이 점근적으로 더 빠른 것은 아니다.

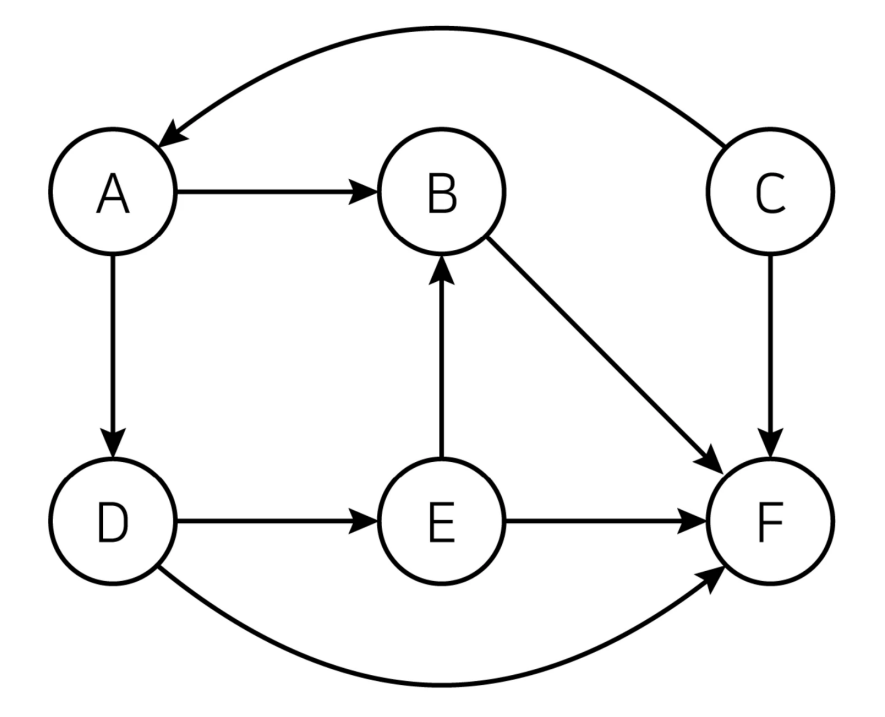
③ 특정 정점에서 시작하는 너비 우선 탐색: 너비 우선 탐색은 큐를 이용하여 발견한 정점의 이웃을 차례로 조사한다. 인접 행렬에서는 방문한 정점마다 **이웃이 아닌 정점의 칸까지** 행 전체를 확인하므로 최악에 $O(n^2)$ 시간이 걸린다. 인접 리스트에서는 각 정점의 실제 이웃 목록을 따라가므로 방문 표시 초기화를 포함해 $O(n + m)$ 시간이 걸린다. 따라서 인접 행렬이 점근적으로 더 빠르지 않다.

④ 특정 두 정점 사이의 간선 존재 여부 확인: 두 정점을 u, v라 하면 인접 행렬에서는 **행 u와 열 v가 만나는 원소 하나**로 연결 여부를 확인하므로 $O(1)$ 시간이 걸린다. 인접 리스트에서는 u의 이웃 목록에서 v를 찾아야 한다. v가 목록의 끝에 있거나 없으면 목록 전체를 조사하므로 $O(1 + d(u))$, 최악에는 $O(n)$ 시간이 걸린다.

따라서 두 표현의 최악 시간 복잡도를 비교할 때, 인접 행렬의 상수 시간 직접 접근이 점근적 이점이 되는 연산은 ④의 간선 존재 여부 확인이다.

NoteBase

문 16. 다음 그래프에 대한 깊이 우선 탐색(depth first search)을 수행할 때 가능한 정점들의 방문 순서는?



① A, B, F, C, D, E

② C, A, B, D, E, F

③ D, F, E, B, A, C

④ E, F, A, B, D, C

정답 3번

간단한 해설

깊이 우선 탐색은 화살표 방향으로 미방문 정점을 따라가다가 더 갈 곳이 없으면 **이전 정점으로 되돌아간다**.
D에서 F를 먼저 방문하면 D로 돌아와 E와 B를 탐색하고, 그 탐색이 끝난 뒤 아직 방문하지 않은 A와 C를 차례로 시작점으로 삼을 수 있다.

NoteBase

올바른 순서

D → F → E → B → A → C

D: 깊이 우선 탐색은 현재 정점에서 나가는 간선을 따라 미방문 정점을 먼저 탐색하고, 더 진행할 수 없을 때 되돌아가는 방식이다. 시작 정점과 인접 정점의 선택 순서가 지정되지 않았으므로 D에서 시작하여, D가 가리키는 E와 F 중 F를 먼저 선택할 수 있다.

F: D → F 간선을 따라 방문한다. F에서 나가는 간선이 없으므로 탐색을 마치고 D로 되돌아간다. **되돌아가는 정점은 방문 순서에 다시 적지 않는다**.

E: D에는 아직 방문하지 않은 인접 정점 E가 남아 있으므로 D → E 간선을 따라 방문한다. E가 가리키는 B와 F 중 F는 이미 방문했으므로, 다음에는 미방문 정점 B를 탐색한다.

B: E → B 간선을 따라 방문한다. B가 가리키는 F는 이미 방문했으므로 더 진행하지 않고 E로 되돌아간다. E와 D에도 더 탐색할 미방문 인접 정점이 없으므로 D에서 시작한 탐색이 끝난다.

A: 아직 A와 C가 미방문 상태이므로, **남은 미방문 정점에서 탐색을 새로 시작**한다. A를 먼저 선택할 수 있으며, A가 가리키는 B와 D는 모두 이미 방문했으므로 A에서의 탐색을 마친다.

C: 마지막 미방문 정점 C에서 탐색을 시작한다. C가 가리키는 A와 F는 모두 이미 방문했으므로 추가 방문 없이 탐색을 마친다.

NoteBase

문 17. 2-3 트리에 대한 설명으로 옳은 것은?

① 2-3 트리가 항상 완전 균형 트리를 유지하는 것은 아니다.

② 2-3 트리의 삽입은 B 트리와 달리 리프 노드 외에서도 발생한다.

③ 3-노드에 저장된 키값들은 그 노드의 모든 자식 노드의 키값보다 크다.

④ 3-노드를 2개의 2-노드로 표현하여 2-3 트리를 레드-블랙 트리로 만들 수 있다.

정답 4번

간단한 해설

3-노드에는 키가 두 개 있으므로, 이를 키 하나씩을 가진 두 이진 노드로 나누고 **빨간 링크로 연결**하여 원래 하나의 노드였음을 나타낼 수 있다. 이 표현을 이용하면 2-3 트리를 레드-블랙 트리로 나타낼 수 있다.

선지별 해설

① X: 2-3 트리는 **모든 리프 노드의 깊이가 같은** 완전 균형 탐색 트리이다. 삽입이나 삭제로 노드의 분할·병합 등이 일어나더라도 이 성질을 유지하므로, 항상 완전 균형을 유지하는 것은 아니라는 설명은 틀리다.

② X: 2-3 트리에서 **새 키가 처음 들어가는 위치는 리프 노드**이며, 이는 B 트리의 삽입과 같다. 리프에 키가 넘치면 노드를 분할하고 가운데 키를 부모로 올리므로 내부 노드도 바뀔 수 있지만, 이러한 분할과 승격은 B 트리에서도 일어난다.

③ X: 3-노드는 크기순으로 정렬된 두 키를 저장한다. 내부 노드인 경우 **왼쪽 부분 트리의 키는 작은 키보다 작고, 가운데 부분 트리의 키는 두 키 사이이며, 오른쪽 부분 트리의 키는 큰 키보다 크다**. 따라서 노드에 저장된 키가 모든 자식 노드의 키보다 크다는 설명은 틀리다.

④ O: 3-노드의 두 키를 각각 하나의 키를 가진 이진 노드에 나누어 저장하고, **검은 부모 노드와 빨간 자식 노드**로 연결하면 원래의 3-노드를 표현할 수 있다. 예를 들어 키 10과 20을 가진 3-노드는 20을 검은 부모로, 10을 그 왼쪽의 빨간 자식으로 나타낼 수 있다. 이때 키의 대소 관계를 유지하면서 빨간 링크가 두 이진 노드를 하나의 3-노드에 대응시키므로, 2-3 트리를 레드-블랙 트리로 표현할 수 있다.

NoteBase

문 18. n개의 버킷(bucket)으로 구성된 비어있는 해시 테이블에 다음 <조건>에 따라 해시 함수 h(x)=x n을 사용하여 키값 <1, 20, 2, 35, 18>을 차례대로 삽입할 때, 발생하는 전체 충돌 횟수가 가장 많은 n값은?

<조건>
◯ n은 자연수이다.
◯ 버킷 내 슬롯(slot) 수는 1이다.
◯ 버킷의 인덱스는 0부터 시작한다.
◯ 충돌 해결 방법으로 선형 조사법(linear probing)을 사용한다.
◯ 각 키값의 삽입과정에서 발생하는 충돌 횟수가 k(k는 0 이상의 정수)일 때 전체 충돌 횟수가 k만큼 증가한다.

① 5
② 7
③ 13
④ 17

정답 4번

간단한 해설

선형 조사법에서는 **빈 버킷을 찾기 전까지 만나는 점유 버킷마다** 충돌을 센다. n = 17이면 35를 넣을 때 3회, 18을 넣을 때 4회 충돌하여 전체 충돌 횟수가 7회로 가장 많다.

NoteBase

선지별 해설

해시 함수 h(x)=x n은 키값을 n으로 나눈 나머지를 최초 버킷 인덱스로 정한다. 버킷당 슬롯이 1개이므로 이미 키가 있으면 다음 인덱스를 차례로 조사하여 빈 버킷에 넣는다. 이때 **조사 중 만나는 점유 버킷 하나마다 충돌 1회**를 더한다. 각 후보에서 빈 테이블로 시작하여 1, 20, 2, 35, 18 순서로 삽입한다.

① n = 5: 최초 해시값은 차례로 1, 0, 2, 0, 3이다. 먼저 1, 20, 2가 각각 인덱스 1, 0, 2에 들어간다. 35는 인덱스 0 → 1 → 2에서 3회 충돌한 뒤 인덱스 3에 들어간다. 이어 18은 **35가 들어간 인덱스 3**에서 1회 충돌하고 인덱스 4에 들어간다. 전체 충돌 횟수는 0 + 0 + 0 + 3 + 1 = 4회이다.

② n = 7: 최초 해시값은 차례로 1, 6, 2, 0, 4이다. 모두 서로 다른 인덱스이므로 각 키가 처음 조사한 빈 버킷에 바로 들어간다. 전체 충돌 횟수는 0회이다.

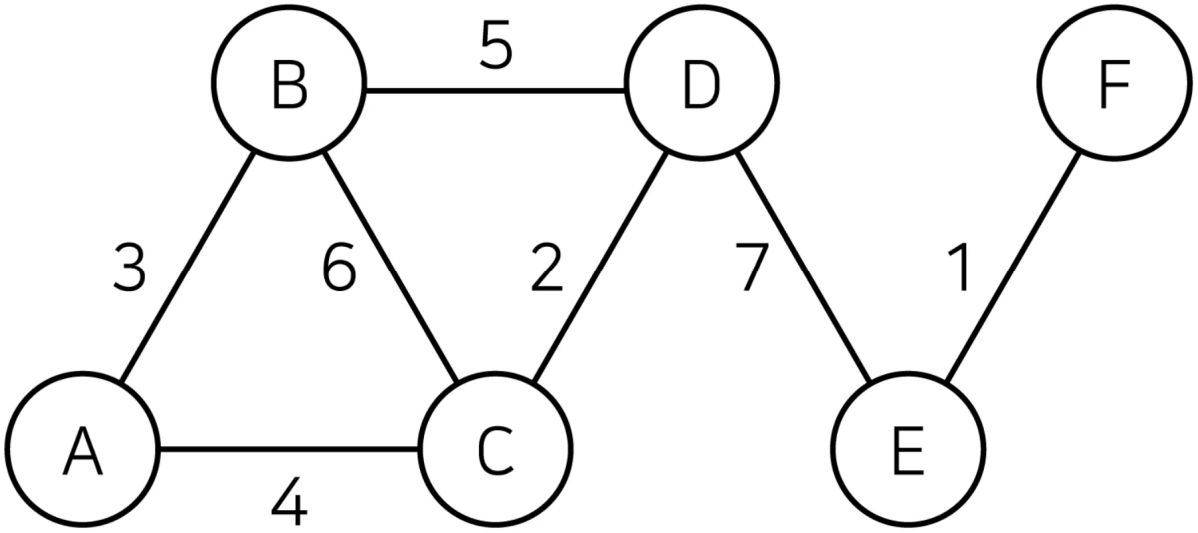
③ n = 13: 최초 해시값은 차례로 1, 7, 2, 9, 5이다. 이 경우에도 모두 서로 다른 인덱스이므로 충돌 없이 삽입된다. 전체 충돌 횟수는 0회이다.

④ n = 17: 최초 해시값은 차례로 1, 3, 2, 1, 1이다. 먼저 1, 20, 2가 각각 인덱스 1, 3, 2에 들어가므로 인덱스 1~3이 연속으로 차 있다. 35는 인덱스 1 → 2 → 3에서 3회 충돌한 뒤 인덱스 4에 들어간다. 이어 18은 **인덱스 1~4가 모두 차 있으므로** 4회 충돌한 뒤 인덱스 5에 들어간다. 전체 충돌 횟수는 0 + 0 + 0 + 3 + 4 = 7회이다.

전체 충돌 횟수는 n = 5, 7, 13, 17에서 각각 4회, 0회, 0회, 7회이다. 따라서 충돌 횟수가 가장 많은 n값은 ④의 17이다.

NoteBase

문 19. 다음 그래프의 최소 비용 신장 트리(minimum cost spanning tree)에 대한 설명으로 옳은 것은?



① 사이클이 있어 최소 비용 신장 트리가 존재하지 않는다.

② 가중치가 가장 큰 간선은 최소 비용 신장 트리에 포함되지 않는다.

③ 모든 간선 가중치가 서로 달라 최소 비용 신장 트리의 개수는 1이다.

④ 두 정점 B와 D 사이의 최단 경로는 최소 비용 신장 트리에 포함된다.

정답 3번

간단한 해설

연결된 무방향 그래프에서 **모든 간선의 가중치가 서로 다르면** 최소 비용 신장 트리는 유일하다. 그림의 간선 가중치는 1부터 7까지 모두 다르므로 이 조건을 만족한다.

NoteBase

선지별 해설

최소 비용 신장 트리는 모든 정점을 연결하면서 사이클이 없고, 선택한 간선의 가중치 합이 가장 작은 부분 그래프이다. 이 그래프는 정점이 6개이므로 신장 트리는 간선 5개로 이루어진다.

① X: **원래 그래프에 사이클이 있어도** 연결 그래프라면 신장 트리가 존재한다. 예를 들어 A–B–C–A는 사이클이지만, 사이클을 이루는 간선 중 일부를 제외하면서 모든 정점을 연결할 수 있다. 사이클이 없어야 하는 대상은 원래 그래프가 아니라 선택한 신장 트리이다.

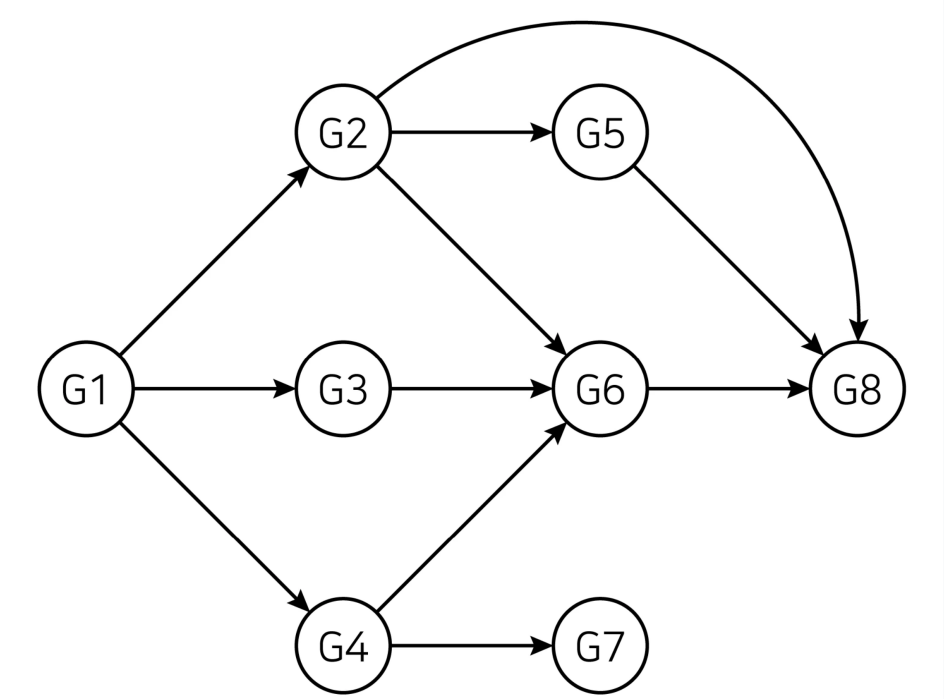
② X: 가장 큰 가중치 7의 간선 D–E는 **두 부분을 연결하는 유일한 간선**이다. 이를 제외하면 A·B·C·D와 E·F가 서로 단절되므로 모든 신장 트리에 D–E가 포함된다. 가중치가 크다는 이유만으로 제외할 수 없으며, 사이클 안에서 유일하게 가장 무거운 간선일 때 그 사이클의 나머지 경로로 대체하여 제외할 수 있다.

③ O: 연결된 무방향 그래프에서 **간선 가중치가 모두 다르면 최소 비용 신장 트리는 유일**하다. 이 그래프에서는 가중치가 작은 순서대로 사이클을 만들지 않는 간선을 선택하면 E–F(1), C–D(2), A–B(3), A–C(4)가 선택된다. B–D(5)와 B–C(6)는 이미 연결된 정점 사이에 사이클을 만들므로 제외하고, 마지막으로 D–E(7)를 선택한다. 따라서 최소 비용 신장 트리는 이 5개 간선으로 유일하게 정해지며, 비용은 1 + 2 + 3 + 4 + 7 = 17이다.

④ X: B와 D 사이의 최단 경로는 가중치가 5인 직접 간선 B–D이다. B–C–D의 비용은 6 + 2 = 8이고, B–A–C–D의 비용은 3 + 4 + 2 = 9이므로 직접 간선이 가장 짧다. 그러나 최소 비용 신장 트리에는 B–D가 포함되지 않고 B–A–C–D로 연결된다. **신장 트리 전체의 간선 비용 합을 최소화**하는 것과 특정 두 정점 사이의 경로 비용을 최소화하는 것은 서로 다른 목표이다.

NoteBase

문 20. 다음 그래프에 대해 위상 정렬(topological sort)을 수행할 때 생성되는 위상 순서로 옳지 않은 것은?



① G1, G2, G3, G4, G5, G6, G7, G8

② G1, G2, G3, G4, G5, G8, G6, G7

③ G1, G3, G2, G4, G6, G5, G8, G7

④ G1, G4, G3, G2, G7, G6, G5, G8

정답 2번

간단한 해설

그래프의 G6 → G8은 **G6이 G8보다 먼저 나와야 한다**는 뜻이다. ②는 G8을 G6보다 앞에 배치하여 이 선후 관계를 어긴다.

NoteBase

선지별 해설

위상 정렬은 방향 비순환 그래프의 모든 정점을 나열하되, 간선 A → B가 있으면 **A를 B보다 앞에 배치**하는 것이다. 매 단계에서 아직 나열하지 않은 정점들 사이의 진입차수가 0인 정점을 선택할 수 있다. 조건을 만족하는 정점이 여러 개이면 선택에 따라 서로 다른 위상 순서가 만들어진다.

그림에서 G2·G3·G4는 G1 뒤에, G5는 G2 뒤에, G6은 G2·G3·G4 뒤에, G7은 G4 뒤에 나와야 한다. G8로 들어오는 간선은 G2·G5·G6에서 시작하므로 G8은 이 세 정점이 모두 나온 뒤에 배치해야 한다.

① O: G1 뒤에 G2·G3·G4가 나오고, 이 세 정점 뒤에 G6이 나온다. G5는 G2 뒤에, G7은 G4 뒤에 있으며, G8도 G2·G5·G6 뒤에 있으므로 모든 간선의 선후 관계를 만족한다.

② X: **G6 → G8의 선후 관계를 거꾸로 배치**했다. G1·G2·G3·G4·G5까지 나열했더라도 G6이 아직 나오지 않았으므로 G8을 선택할 수 없다. G8을 G6 뒤로 옮기면 이 위반이 해소된다.

③ O: G3을 G2보다 먼저 배치해도 두 정점 사이에는 간선이 없어 문제가 없다. G6은 G2·G3·G4 뒤에, G5는 G2 뒤에 나오며, G8은 G2·G5·G6 뒤에 나온다. G7도 G4 뒤에 있으므로 모든 선후 조건을 만족한다.

④ O: G1 뒤의 G4·G3·G2는 서로의 선후를 강제하는 간선이 없어 이 순서로 나열할 수 있다. G4가 이미 나왔으므로 **G7을 G6보다 먼저 배치할 수 있다**. 이어지는 G6은 G2·G3·G4 뒤에, G5는 G2 뒤에, G8은 G2·G5·G6 뒤에 있어 모든 선후 조건을 만족한다.

NoteBase

문 21. 다음 C 언어로 구현한 arraySort 함수에서 재귀 호출 횟수의 최댓값이 가장 큰 배열 arr의 입력내용은? (단, arraySort 함수의 최초 호출 시 low 변수와 high 변수는 배열 arr의 첫 번째와 마지막 데이터의 인덱스이다)

```
#define SWAP(x,y) {int temp = x; x = y; y = temp;}

int splitArray(int arr[], int low, int high) {
   int pi = (low + high)/2;
   SWAP(arr[pi], arr[high]);
   int pivot = arr[high];

   int i = low - 1;
   for(int j = low; j < high; j++) {
      if(arr[j] < pivot) {
         i++;
         SWAP(arr[i], arr[j]);
      }
   }
   SWAP(arr[high], arr[i + 1]);
   return i + 1;
}

void arraySort(int arr[], int low, int high) {
   if(low < high) {
      int pi = splitArray(arr, low, high);
      arraySort(arr, low, pi - 1);
      arraySort(arr, pi + 1, high);
   }
}
```

① 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

② 2, 4, 6, 8, 10, 5, 3, 7, 1, 9

③ 3, 6, 2, 8, 5, 10, 7, 1, 9, 4

④ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

정답 2번

간단한 해설

②에서는 가운데 인덱스의 값이 매번 **현재 구간의 최댓값**이 된다. 피벗만 제외한 나머지가 모두 한쪽 구간에 남아, 처리할 원소 수가 10개에서 1개까지 하나씩 줄어들므로 재귀 호출이 가장 많이 발생한다.

NoteBase

선지별 해설

splitArray는 현재 구간의 가운데 인덱스에 있는 값을 피벗으로 선택한다. 가운데 인덱스는 low와 high의 평균에서 소수 부분을 버린 값이므로, 원소가 짝수 개이면 두 중앙 위치 중 왼쪽이다. 피벗을 구간 끝으로 옮긴 뒤 작은 값들을 왼쪽에 모으고, 피벗의 최종 위치를 기준으로 양쪽 구간을 재귀 처리한다. 가운데 '위치'를 고를 뿐 **크기순 가운데 값이 보장되지는 않는다**. 원소가 2개 이상인 구간에서만 분할하며, 분할할 때마다 재귀 호출문 두 개가 실행된다. 이하 호출 횟수는 최초 호출을 제외하고 빈 구간과 원소 하나인 구간에 대한 호출도 포함한다.

① 처음 피벗은 5이고, 좌우 구간의 원소 수는 4개와 5개이다. 왼쪽 구간에서는 2, 3을 피벗으로 분할하고, 오른쪽 구간에서는 8을 피벗으로 나눈 뒤 각각 6과 9를 피벗으로 분할한다. 전체 분할은 처음의 1회, 왼쪽의 2회, 오른쪽의 3회를 합쳐 6회이므로 재귀 호출은 6 × 2 = 12회이다. 오름차순 배열이어도 **첫 원소를 피벗으로 선택하는 코드가 아니므로** 한쪽으로만 분할되지 않는다.

② 처음 가운데 값 10이 피벗이다. 이를 마지막 값 9와 교환하면 앞의 9개는 2, 4, 6, 8, 9, 5, 3, 7, 1이 되고, 모두 10보다 작으므로 피벗 10의 위치가 그대로 확정된다. 남은 구간의 가운데 값은 9이며, 그다음에도 같은 방식으로 현재 구간의 최댓값이 선택된다. 피벗은 10 → 9 → 8 → 7 → 6 → 5 → 4 → 3 → 2 순으로 선택되어 **매번 한쪽 구간만 비게 된다**. 분할하는 구간의 크기는 10개부터 2개까지여서 분할은 9회, 재귀 호출은 9 × 2 = 18회이다.

③ 처음 피벗은 5이며, 분할 후 배열은 3, 2, 4, 1, 5, 10, 7, 8, 9, 6이 된다. 피벗을 제외한 좌우 구간은 4개와 5개이다. 왼쪽에서는 2, 4를 피벗으로 분할하고, 오른쪽에서는 8을 피벗으로 나눈 뒤 각각 7과 9를 피벗으로 분할한다. 전체 분할은 1 + 2 + 3 = 6회이므로 재귀 호출은 12회이다.

④ 처음 가운데 값 6을 피벗으로 분할하면 배열은 1, 5, 4, 3, 2, 6, 8, 7, 10, 9가 된다. 좌우 구간의 원소 수는 5개와 4개이다. 왼쪽에서는 4, 2를 피벗으로 분할하고, 오른쪽에서는 7, 10, 9를 피벗으로 분할한다. 전체 분할은 1 + 2 + 3 = 6회이므로 재귀 호출은 12회이다. 내림차순이라는 사실만으로 최악의 입력이 되는 것은 아니며, **실제 피벗 선택과 분할 결과**를 따라야 한다.

따라서 재귀 호출 횟수는 ①·③·④가 각각 12회, ②가 18회로, ②의 입력에서 가장 크다.

NoteBase

문 22. 다음은 후위 표기식을 스택을 이용하여 사칙연산을 하는 알고리즘을 구현한 C 프로그램이다. 출력 결과가 12인 프로그램의 실행에 대한 설명으로 옳은 것만을 모두 고르면?

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

#define MAX_STACK_SIZE  50

typedef int Item;
Item stack[MAX_STACK_SIZE];
int top;

void reset_stack() { top = -1; }

void push(Item elm) { stack[++top] = elm; }

Item pop() { return stack[top--]; }

Item calc_exp(char *exp) {
   int x, y, z, current, i = 0;
   reset_stack();
   while(exp[i] != '$') {
      current = exp[i++];
      if(current >= '0' && current <= '9')
         push(   (가)   );
      else if(current == '+' || current == '-' ||
              current == '*' || current == '/') {
         (나) = pop();
         (다) = pop();
         switch(current) {
            case '+': z = x + y; break;
            case '-': z = x - y; break;
            case '*': z = x * y; break;
            case '/': z = x / y;
         }
         push(z);
      }
   }
   return pop();
}

int main() {
   char exp[20] = "2 3 4 + * 6 3 / - $";
   printf("%d\n", calc_exp(exp));
   return 0;
}
```

ㄱ. (가)에 들어갈 내용은 current이다.

ㄴ. (나)에 들어갈 내용은 y이고 (다)에 들어갈 내용은 x이다.

ㄷ. 프로그램을 실행하는 동안 pop 함수의 총호출 횟수는 8회이다.

ㄹ. 프로그램을 실행하는 동안 push 함수의 총호출 횟수는 9회이다.

① ㄱ, ㄴ

② ㄱ, ㄷ

③ ㄴ, ㄹ

④ ㄱ, ㄴ, ㄹ

정답 3번

간단한 해설

후위 표기식에서는 **먼저 꺼낸 값이 오른쪽 피연산자**이므로 y에, 다음 값을 x에 저장한다. push는 숫자 5개를 넣을 때와 연산 결과 4개를 넣을 때 호출되어 총 9회이다.

NoteBase

보기별 해설

ㄱ. X: current에는 숫자 자체가 아니라 읽어 들인 문자의 코드값이 저장된다. 따라서 (가)에는 **current - '0'**을 넣어 숫자 문자에 해당하는 정수로 변환해야 한다. 예를 들어 문자 '3'에서 문자 '0'의 코드값을 빼면 정수 3이 된다. current를 그대로 넣으면 숫자의 코드값으로 연산하게 된다.

ㄴ. O: 스택은 나중에 넣은 값을 먼저 꺼내므로, 연산자를 만났을 때 **첫 번째 pop의 결과는 y, 두 번째는 x**에 저장한다. 그래야 x - y와 x / y에서 왼쪽·오른쪽 피연산자의 순서가 유지된다. 주어진 식에서는 3 + 4 = 7을 스택에 넣고, 이어 2 × 7 = 14를 넣는다. 다음으로 6 ÷ 3 = 2를 넣은 뒤 14 - 2 = 12를 계산한다. 따라서 (나)는 y, (다)는 x이다.

ㄷ. X: 연산자는 +, *, /, -의 4개이며, 각 연산자를 처리할 때 피연산자 두 개를 꺼내므로 pop은 4 × 2 = 8회 호출된다. 여기에 **마지막 return pop()의 호출 1회**가 더해진다. 이 호출은 스택에 남은 최종 결과 12를 꺼내 반환하므로, 프로그램 실행 중 pop의 총호출 횟수는 9회이다.

ㄹ. O: 숫자 2, 3, 4, 6, 3을 읽을 때 push가 각각 호출되어 5회이고, 연산자 4개를 처리한 뒤 결과 7, 14, 2, 12를 넣을 때도 각각 호출되어 4회이다. 따라서 **숫자 저장 5회와 연산 결과 저장 4회**를 합한 총호출 횟수는 9회이다. 공백은 어느 처리 조건에도 해당하지 않아 push를 호출하지 않고, '$'는 반복문을 종료하므로 처리하지 않는다.

옳은 보기는 ㄴ, ㄹ이며, 이 조합은 ③에 해당한다.

NoteBase

문 23. 다음 C 프로그램의 실행 결과는?

```c
#include <stdio.h>

typedef struct {
   int id;
   char name[10];
} Student;

void func(Student *dest, Student *src){
   dest = src;
}

int main() {
   Student st_a = {1, "AAA"}, st_b = {2, "BBB"};
   Student *st_arr[3] = {&st_a, &st_b, &st_b};

   func(st_arr[2], st_arr[0]);
   st_arr[2]->name[0] = 'X';

   for(int i = 0; i < 3; i++)
      printf("%s ", st_arr[i]->name);

   return 0;
}
```

① AAA XBB XBB

② AAA BBB XBB

③ AAA BBB XAA

④ XAA BBB XAA

정답 1번

간단한 해설

func의 dest = src는 **복사된 포인터 매개변수만 바꾸므로** st_arr[2]는 계속 st_b를 가리킨다. st_arr[1]과 st_arr[2]가 **같은 st_b를 가리키기 때문에** name[0]을 'X'로 바꾸면 두 위치에서 모두 XBB가 출력된다.

NoteBase

계산 과정

초기 상태: st_a의 name은 'AAA', st_b의 name은 'BBB'이다. 포인터 배열 st_arr의 0번 원소는 st_a를, 1번과 2번 원소는 모두 st_b를 가리킨다. 따라서 배열에는 구조체 세 개가 들어 있는 것이 아니라, 두 구조체의 주소가 저장되어 있다.

func 호출: func(st_arr[2], st_arr[0])에서 dest에는 st_b의 주소가, src에는 st_a의 주소가 복사된다. C의 함수 인수는 값으로 전달되므로 dest와 src는 배열 원소와 별개의 지역 매개변수이다. dest = src를 실행하면 함수 안의 dest만 st_a를 가리키게 되며, **st_arr[2] 자체의 주소 값이나 구조체 내용은 바뀌지 않는다**. 함수가 끝난 뒤에도 배열의 참조 관계는 초기 상태와 같다.

문자 변경: st_arr[2]->name[0] = 'X'는 st_arr[2]가 가리키는 st_b의 name 배열에서 첫 문자를 바꾼다. 따라서 st_b의 이름은 'BBB'에서 'XBB'가 된다. **st_arr[1]도 같은 구조체를 가리키므로** 두 원소를 통해 읽는 이름은 모두 'XBB'이다. st_a의 이름은 'AAA'로 유지된다.

출력: 반복문은 i가 0, 1, 2일 때 차례로 각 원소가 가리키는 구조체의 name을 출력한다. 따라서 출력 문자열은 'AAA XBB XBB'이며, ①의 결과와 일치한다.

NoteBase

문 24. 다음 C 프로그램의 실행 결과는?

```c
#include <stdio.h>

int count = 0;

void new_hanoi(int n, char from, char tmp, char to) {
   if (n <= 2)
      count += n;
   else {
      new_hanoi(n - 2, from, to, tmp);
      count += 2;
      new_hanoi(n - 2, tmp, from, to);
   }
}

int main() {
   new_hanoi(9, 'A', 'B', 'C');
   printf("%d", count);
   return 0;
}
```

① 31

② 46

③ 62

④ 511

정답 2번

간단한 해설

n이 2보다 크면 **n − 2에 대한 재귀 호출을 두 번** 수행하고 count에 2를 더한다. 종료 조건에서 더하는 값까지 누적하면 입력 9에 대한 출력은 **46**이다.

NoteBase

계산 과정

초기 상태: 전역 변수 count는 0이며 모든 재귀 호출이 같은 변수를 갱신한다. 한 번의 new_hanoi 호출이 하위 호출까지 모두 마칠 때 count에 더하는 총량을 $C(n)$이라 하자. from, tmp, to는 count의 증가량을 결정하는 조건이나 연산에 사용되지 않으므로, 증가량은 n으로 결정된다.

종료 조건: n이 2 이하이면 재귀 호출 없이 count += n만 실행하므로 $C(n) = n$이다. 최초 입력이 9이고 호출마다 2씩 줄어들기 때문에, 이 실행에서 도달하는 종료 값은 **n = 1**이며 $C(1) = 1$이다.

재귀 단계: n이 2보다 크면 n − 2를 인수로 하는 호출을 두 번 실행하고, 그 사이에 count += 2를 실행한다. 따라서 두 하위 호출의 증가량과 현재 호출에서 더하는 2를 합쳐 $C(n) = 2C(n-2) + 2$가 된다. **재귀 호출 한 번당 1을 세는 것이 아니라** 각 count 갱신문의 증가량을 누적해야 한다.

누적값 계산: 종료 값에서 시작해 상위 호출의 증가량을 차례로 구한다. 각 값은 해당 입력의 호출 한 번이 끝날 때까지 더해지는 총량이다.
$C(3) = 2 \cdot 1 + 2 = 4$
$C(5) = 2 \cdot 4 + 2 = 10$
$C(7) = 2 \cdot 10 + 2 = 22$
$C(9) = 2 \cdot 22 + 2 = 46$

출력: main에서 new_hanoi(9, 'A', 'B', 'C')가 끝나면 count는 0 + 46 = 46이다. printf는 이 값을 십진 정수로 출력하므로 실행 결과는 46이며, ②와 일치한다.

NoteBase

문 25. 다음은 입력 배열 데이터 data 내에 찾고자 하는 값 key와 같은 값이 저장된 모든 배열 원소의 인덱스를 배열로 반환하는 search 함수를 포함한 Java 프로그램이다. 출력 결과가 (12:2,13:0,23:3,43:1)인 프로그램의 실행에 대한 설명으로 옳은 것만을 모두 고르면?

```
public class SearchAlgorithm {
   private static int[] index_list;
   private static int size;
   private static void reset() {
      index_list = new int[1];
      size = 0;
   }
   private static void resize(int newsize) {
      int[] nlist = new int[newsize];
      for(int i = 0; i < size; i++)
         nlist[i] = index_list[i];
      index_list = nlist;
   }
   private static void insert_index(int index) {
      if(size == index_list.length) resize(2 * size);
      index_list[   (가)   ] = index;
   }
   public static int[] search(int[] data, int key) {
      (나) ;
      for(int i = 0; i < data.length; i++)
         if(data[i] == key) insert_index(i);
      int[] result_list = new int[size];
      for(int i = 0; i < result_list.length; i++)
         result_list[i] = index_list[i];
      return result_list;
   }
   public static void main(String[] args) {
      int[] data = {12, 45, 23, 43, 23, 54, 12, 23};
      int[] keys = {12, 13, 23, 43};
      String index_f = "(";
      for(int i = 0; i < keys.length; i++) {
         int key = keys[i];
         int[] indexs = search(data, key);
         index_f += key + ":" + indexs.length;
         if(i != keys.length - 1) index_f += ",";
      }
      System.out.println(index_f + ")");
      return;
   }
}
```

ㄱ. (가)에 들어갈 내용은 ++size이다.

ㄴ. (나)에 들어갈 내용은 reset()이다.

ㄷ. 프로그램을 실행하는 동안 resize 함수의 총호출 횟수는 4회이다.

ㄹ. 프로그램을 실행하는 동안 insert_index 함수의 총호출 횟수는 6회이다.

① ㄱ, ㄷ

② ㄴ, ㄷ

③ ㄴ, ㄹ

④ ㄱ, ㄴ, ㄹ

정답 3번

간단한 해설

search를 호출할 때마다 **reset()으로 저장 상태를 초기화**해야 검색값별 개수가 따로 집계된다. insert_index는 값이 일치할 때마다 한 번 호출되므로, 12·13·23·43의 발견 횟수를 합한 **총 6회** 호출된다.

NoteBase

보기별 해설

size는 저장된 인덱스의 개수이고, index_list.length는 인덱스를 저장할 수 있는 배열의 용량이다. insert_index는 배열이 가득 찼을 때 용량을 두 배로 늘린 뒤, 다음 빈 위치에 발견한 인덱스를 저장한다.

ㄱ. X: (가)에는 **size++**가 들어가야 한다. 배열 인덱스는 0부터 시작하므로, 현재 size를 저장 위치로 사용한 다음 size를 1 증가시켜야 한다. 첫 삽입에서는 index_list[0]에 저장하고 size가 1이 된다. ++size를 쓰면 size를 먼저 1로 늘려 길이가 1인 배열의 index_list[1]에 접근하므로 배열 범위를 벗어난다.

ㄴ. O: (나)에는 **reset()**이 들어간다. 이 함수는 index_list를 길이 1인 새 배열로 만들고 size를 0으로 설정한다. 각 검색 전에 이전 검색 결과를 지워야 12는 2개, 13은 0개, 23은 3개, 43은 1개라는 독립적인 결과를 반환한다. 특히 13은 발견되지 않으므로 size가 0인 상태를 유지하여 길이가 0인 결과 배열을 반환한다.

ㄷ. X: resize의 총호출 횟수는 **3회**이다. 검색할 때마다 용량이 1로 초기화된다. 12 검색에서는 첫 인덱스를 저장한 뒤 배열이 가득 차므로 두 번째 인덱스를 넣을 때 용량이 1 → 2로 늘어나 1회 호출된다. 23 검색에서는 두 번째와 세 번째 인덱스를 넣을 때 각각 용량이 1 → 2, 2 → 4로 늘어나 2회 호출된다. 13은 삽입이 없고, 43은 용량 1에 한 번만 삽입하므로 확장이 없다. 따라서 총호출 횟수는 1 + 0 + 2 + 0 = 3회이다.

ㄹ. O: insert_index는 **data[i]가 key와 같을 때마다 한 번** 호출된다. 12는 인덱스 0·6에서 발견되어 2회, 13은 발견되지 않아 0회, 23은 인덱스 2·4·7에서 발견되어 3회, 43은 인덱스 3에서 발견되어 1회 호출된다. 따라서 총호출 횟수는 2 + 0 + 3 + 1 = 6회이다.

옳은 보기는 ㄴ과 ㄹ이며, 이 조합은 ③에 해당한다.

NoteBase