자료구조 실습

11/12

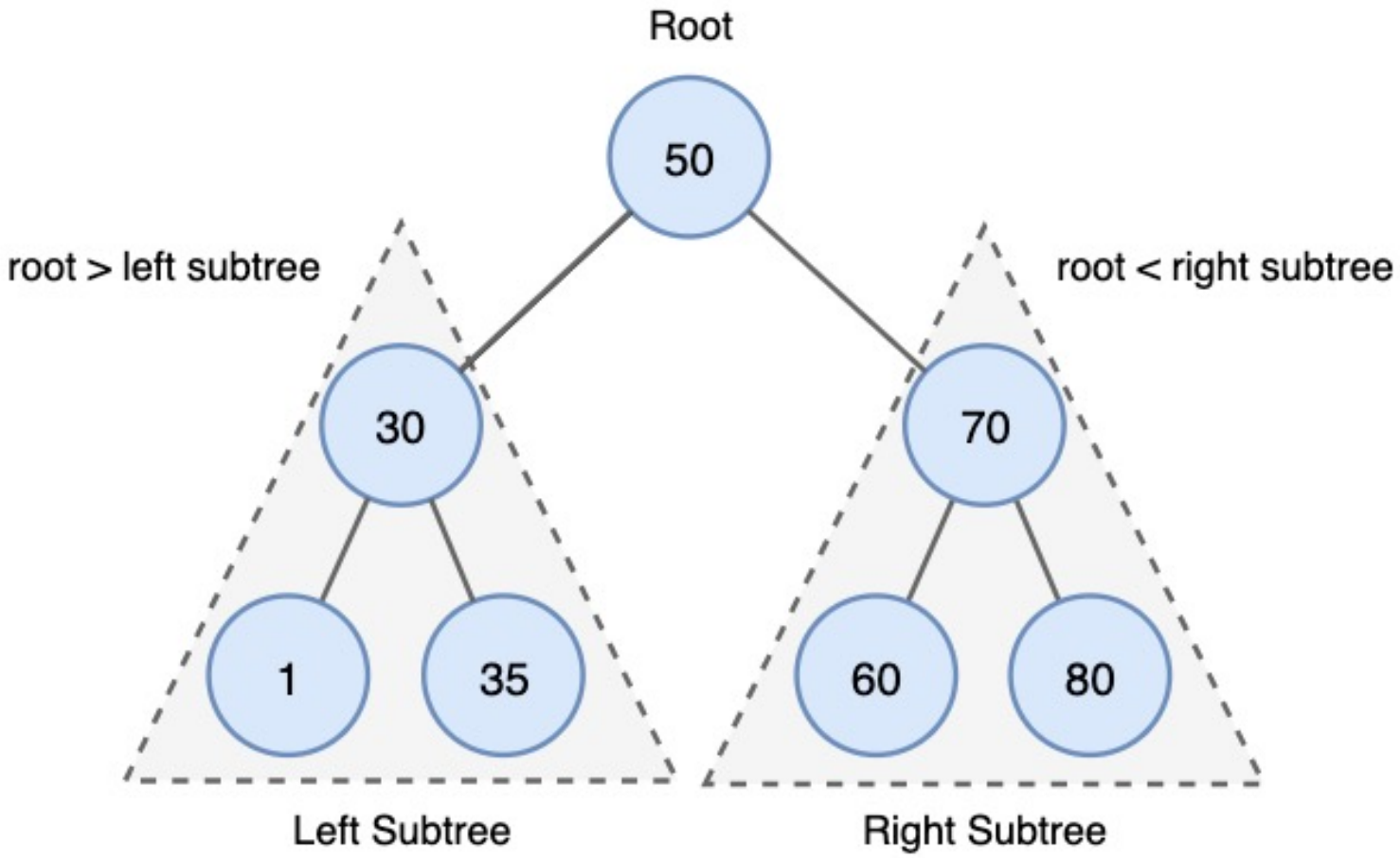
김민서 (imkmsh@khu.ac.kr)

목차

- BST (Binary Search Tree) 란?
- 노드 검색
- 노드 삽입
- 노드 삭제

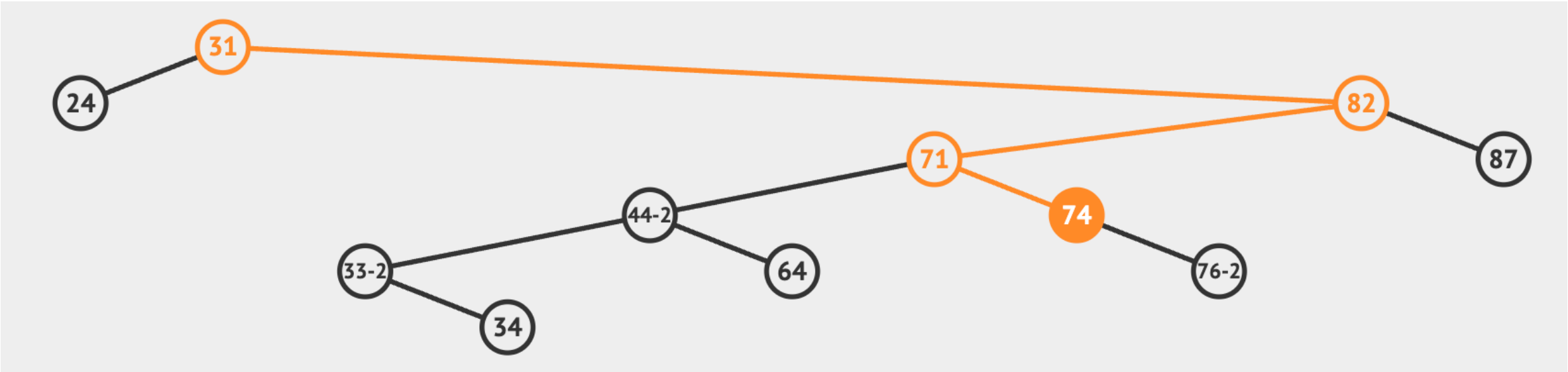
김민서 (imkmsh@khu.ac.kr)

BST (Binary Search Tree) 란?



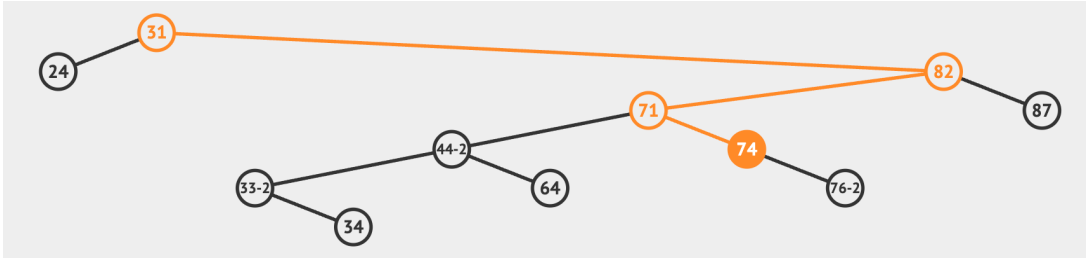
김민서 (imkmsh@khu.ac.kr)

노드 검색



김민서 (imkmsh@khu.ac.kr)

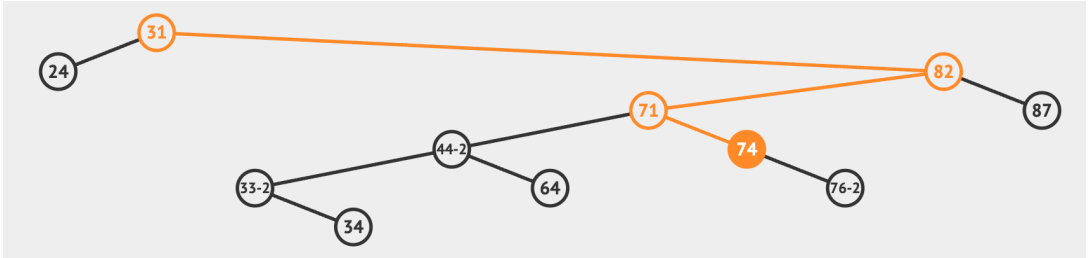
노드 검색




```c
/*------------------------------------------------------------
  탐색 함수: bst_search
  - 주어진 키(x)를 트리에서 찾아서 해당 노드 포인터를 반환
  - 존재하지 않으면 NULL 반환
  - 원리: BST는 '왼쪽 < 루트 < 오른쪽' 규칙을 이용
------------------------------------------------------------*/
Node* bst_search(Node* root, Key x) {
    Node* cur = root;                       // 탐색은 루트 노드부터 시작
    while (cur) {                           // NULL에 도달할 때까지 반복
        if (x == cur->[redacted] return [redacted]   // 키가 일치하면 탐색 성공 → 해당 노드 반환
        if (x < cur->key)                   // 찾는 값이 현재 키보다 작으면
            cur = cur->[redacted]           // 왼쪽 서브트리로 이동
        else                                // 찾는 값이 현재 키보다 크면
            cur = cur->[redacted]           // 오른쪽 서브트리로 이동
    }
    return NULL;                            // 루프 종료 시 NULL이면 해당 키는 트리에 없음
}
```

김민서 (imkmsh@khu.ac.kr)

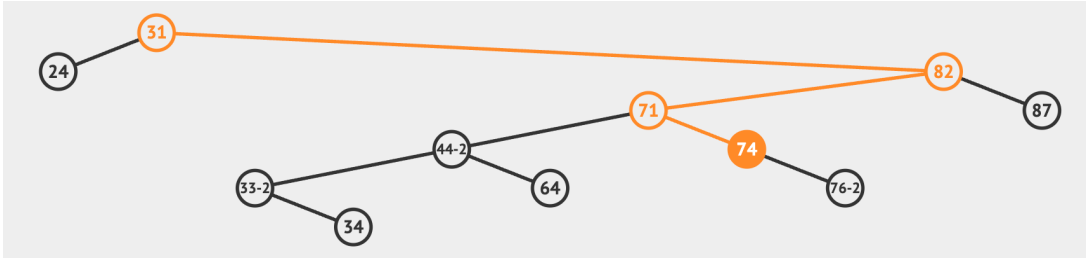
노드 검색




```c
/*------------------------------------------------------------
  탐색 함수: bst_search
  - 주어진 키(x)를 트리에서 찾아서 해당 노드 포인터를 반환
  - 존재하지 않으면 NULL 반환
  - 원리: BST는 '왼쪽 < 루트 < 오른쪽' 규칙을 이용
------------------------------------------------------------*/
Node* bst_search(Node* root, Key x) {
    Node* cur = root;                    // 탐색은 루트 노드부터 시작
    while (cur) {                        // NULL에 도달할 때까지 반복
        if (x == cur->key) return [redacted]    // 키가 일치하면 탐색 성공 → 해당 노드 반환
        if (x < cur->key)                // 찾는 값이 현재 키보다 작으면
            cur = cur->[redacted]        // 왼쪽 서브트리로 이동
        else                             // 찾는 값이 현재 키보다 크면
            cur = cur->[redacted]        // 오른쪽 서브트리로 이동
    }
    return NULL;                         // 루프 종료 시 NULL이면 해당 키는 트리에 없음
}
```

김민서 (imkmsh@khu.ac.kr)

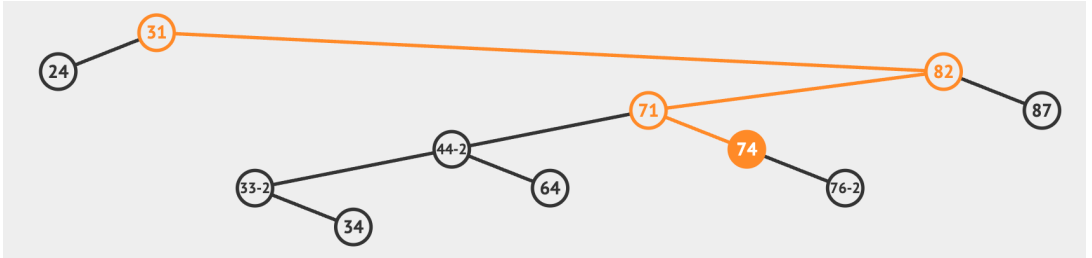
노드 검색




```c
/*------------------------------------------------------------
  탐색 함수: bst_search
  - 주어진 키(x)를 트리에서 찾아서 해당 노드 포인터를 반환
  - 존재하지 않으면 NULL 반환
  - 원리: BST는 '왼쪽 < 루트 < 오른쪽' 규칙을 이용
------------------------------------------------------------*/
Node* bst_search(Node* root, Key x) {
    Node* cur = root;                       // 탐색은 루트 노드부터 시작
    while (cur) {                           // NULL에 도달할 때까지 반복
        if (x == cur->key) return cur;      // 키가 일치하면 탐색 성공 → 해당 노드 반환
        if (x < cur->key)                   // 찾는 값이 현재 키보다 작으면
            cur = cur->[redacted]           // 왼쪽 서브트리로 이동
        else                                // 찾는 값이 현재 키보다 크면
            cur = cur->[redacted]           // 오른쪽 서브트리로 이동
    }
    return NULL;                            // 루프 종료 시 NULL이면 해당 키는 트리에 없음
}
```

김민서 (imkmsh@khu.ac.kr)

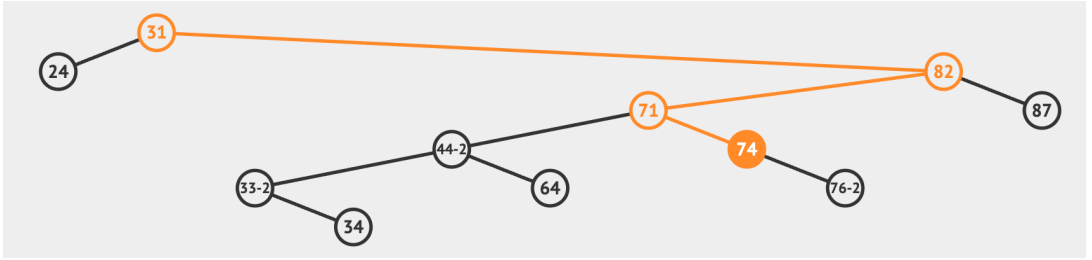
노드 검색




```c
/*------------------------------------------------------------
  탐색 함수: bst_search
  - 주어진 키(x)를 트리에서 찾아서 해당 노드 포인터를 반환
  - 존재하지 않으면 NULL 반환
  - 원리: BST는 '왼쪽 < 루트 < 오른쪽' 규칙을 이용
------------------------------------------------------------*/
Node* bst_search(Node* root, Key x) {
    Node* cur = root;                       // 탐색은 루트 노드부터 시작
    while (cur) {                           // NULL에 도달할 때까지 반복
        if (x == cur->key) return cur;      // 키가 일치하면 탐색 성공 → 해당 노드 반환
        if (x < cur->key)                   // 찾는 값이 현재 키보다 작으면
            cur = cur->left;                // 왼쪽 서브트리로 이동
        else                                // 찾는 값이 현재 키보다 크면
            cur = cur->[redacted]           // 오른쪽 서브트리로 이동
    }
    return NULL;                            // 루프 종료 시 NULL이면 해당 키는 트리에 없음
}
```

김민서 (imkmsh@khu.ac.kr)

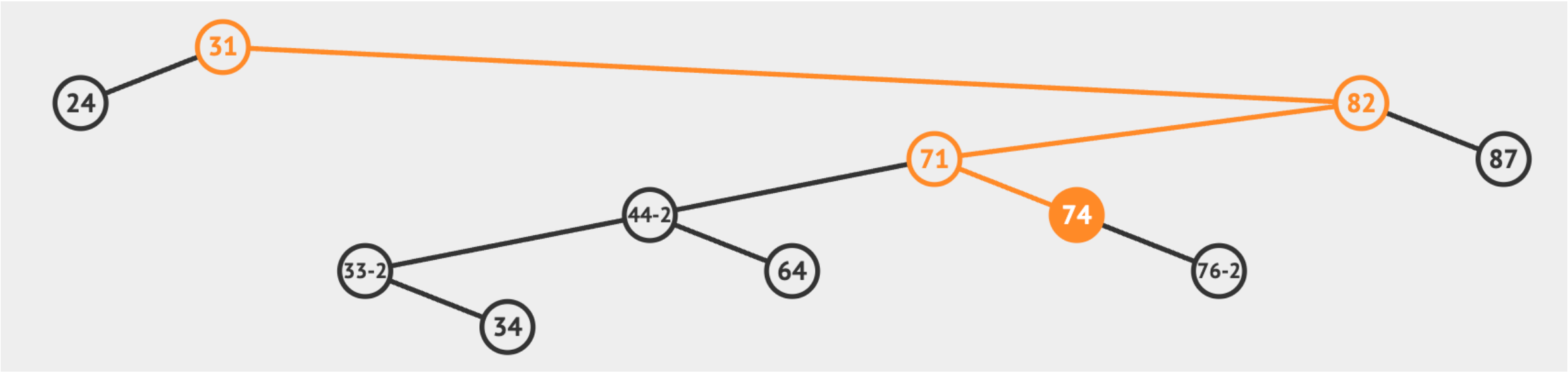
노드 검색




```c
/*------------------------------------------------------------
  탐색 함수: bst_search
  - 주어진 키(x)를 트리에서 찾아서 해당 노드 포인터를 반환
  - 존재하지 않으면 NULL 반환
  - 원리: BST는 '왼쪽 < 루트 < 오른쪽' 규칙을 이용
------------------------------------------------------------*/
Node* bst_search(Node* root, Key x) {
    Node* cur = root;                       // 탐색은 루트 노드부터 시작
    while (cur) {                           // NULL에 도달할 때까지 반복
        if (x == cur->key) return cur;      // 키가 일치하면 탐색 성공 → 해당 노드 반환
        if (x < cur->key)                   // 찾는 값이 현재 키보다 작으면
            cur = cur->left;                // 왼쪽 서브트리로 이동
        else                                // 찾는 값이 현재 키보다 크면
            cur = cur->right;               // 오른쪽 서브트리로 이동
    }
    return NULL;                            // 루프 종료 시 NULL이면 해당 키는 트리에 없음
}
```

김민서 (imkmsh@khu.ac.kr)

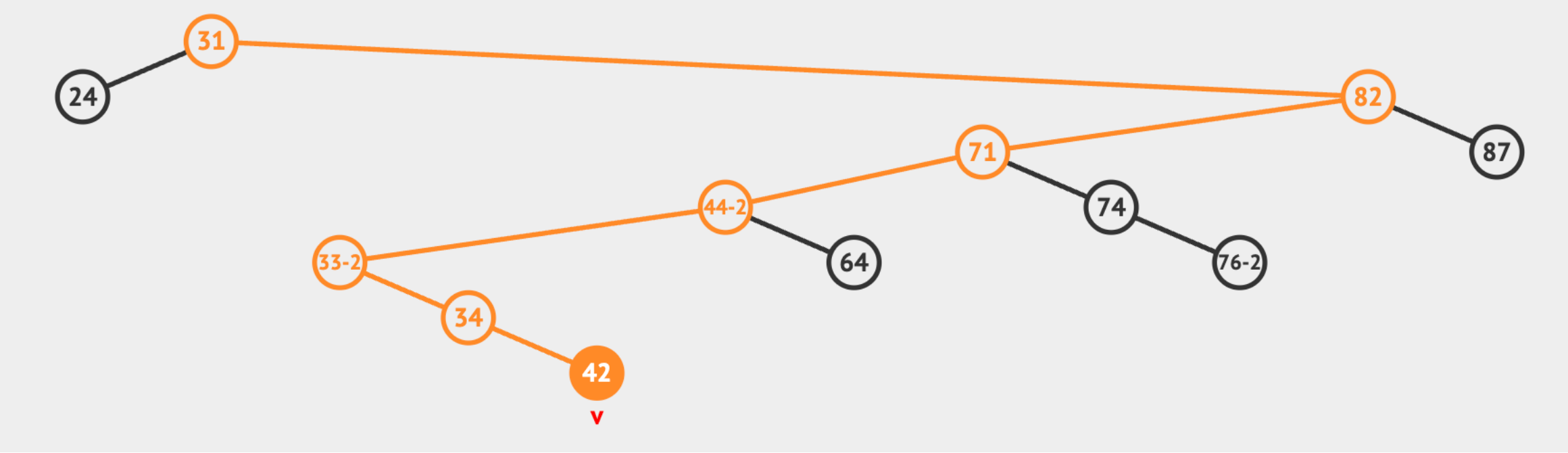
노드 삽입



Let’s insert 42

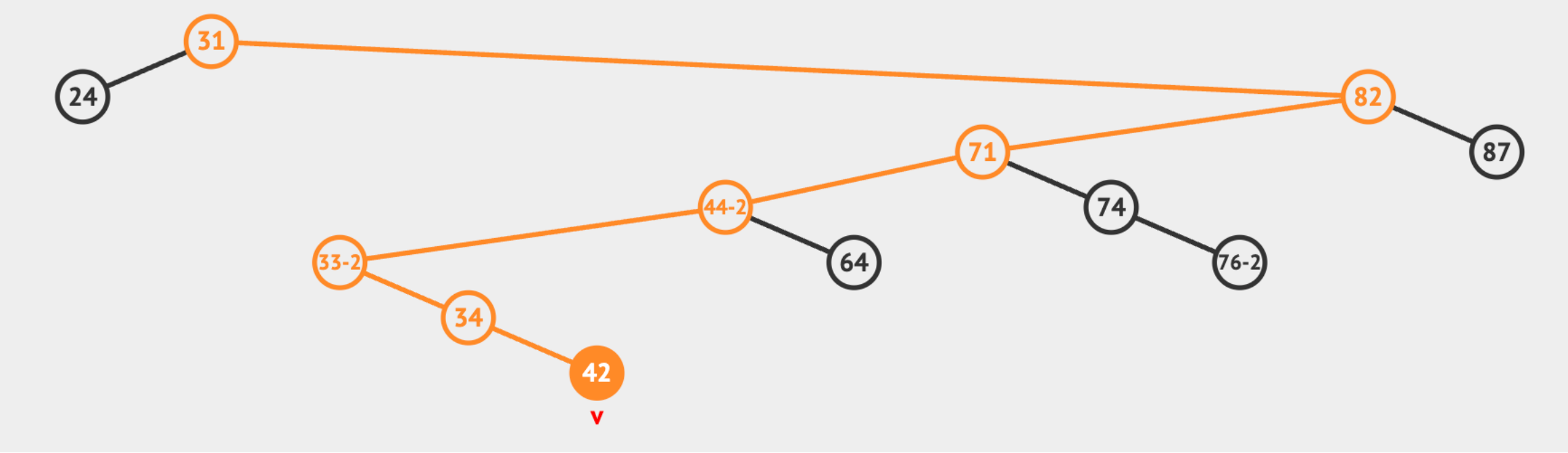
김민서 (imkmsh@khu.ac.kr)

노드 삽입



김민서 (imkmsh@khu.ac.kr)

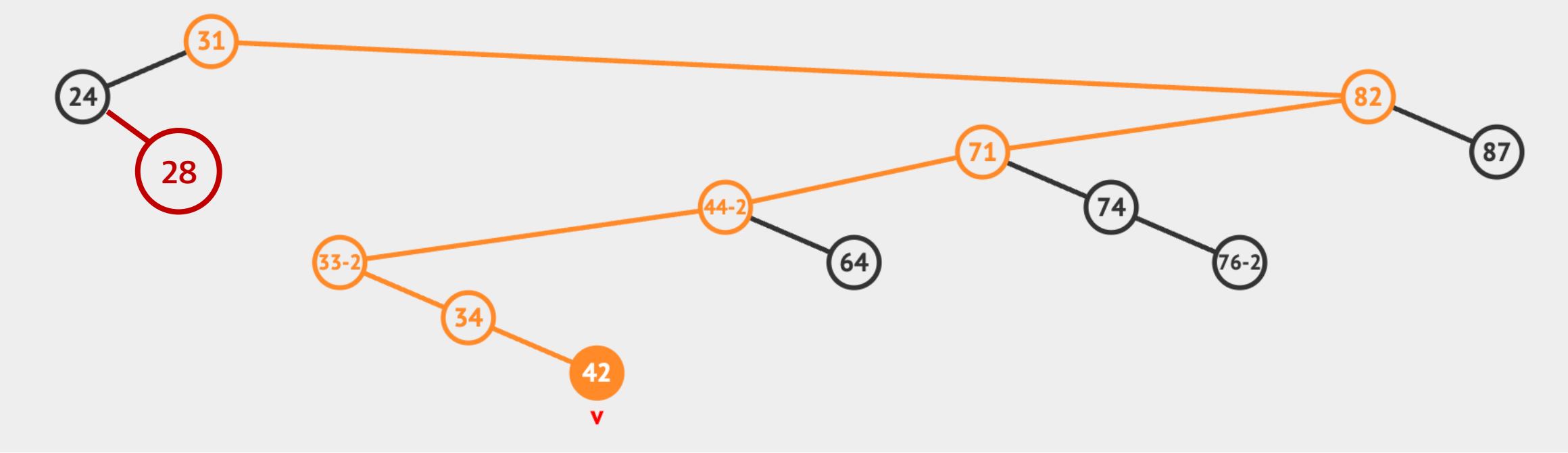
노드 삽입



Let’s insert 28

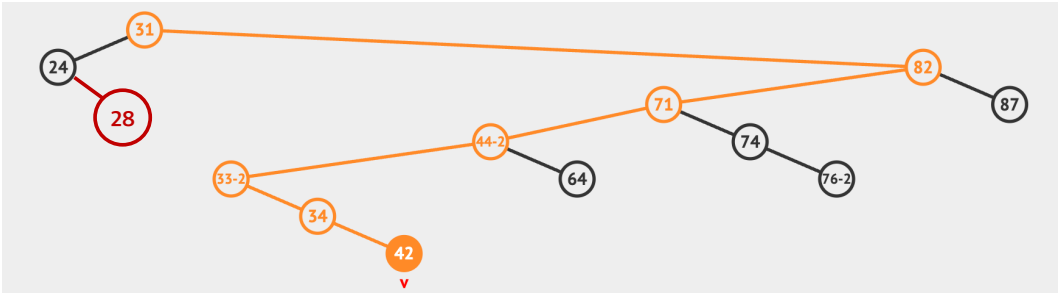
김민서 (imkmsh@khu.ac.kr)

노드 삽입



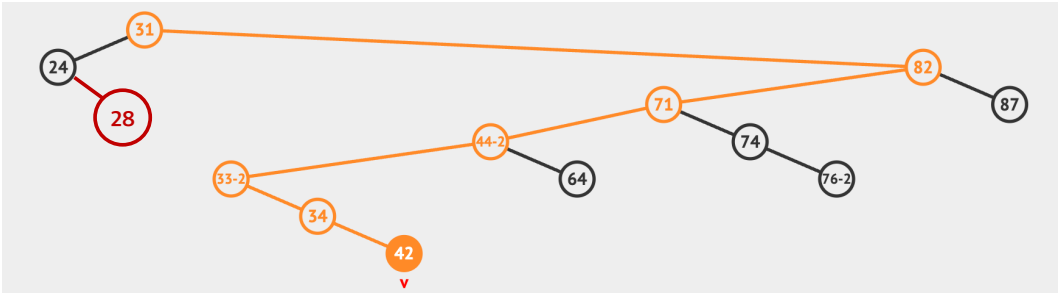
김민서 (imkmsh@khu.ac.kr)

노드 삽입



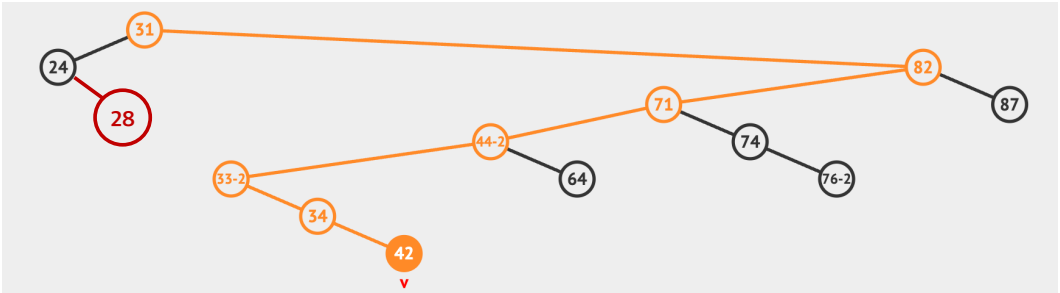

```c
/*------------------------------------------------------------
  삽입 함수: bst_insert
  - 새로운 키(x)를 BST에 삽입하여 새로운 루트를 반환
  - 중복 키는 삽입하지 않고 무시함
  - 원리: 빈 자리를 찾을 때까지 내려가서 새 노드를 붙임
------------------------------------------------------------*/
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면
        Node* n = (Node*)malloc(sizeof(Node));   // 새 노드를 동적 할당
        if (!n) { perror("malloc"); exit(1); }   // 메모리 부족 시 오류 처리
        n->key = [redacted]                      // 새 노드에 키 저장
        n->left = n->right = [redacted]          // 처음에는 자식이 없음
        return n;                                // 새 노드가 곧 루트가 됨
    }
```

노드 삽입



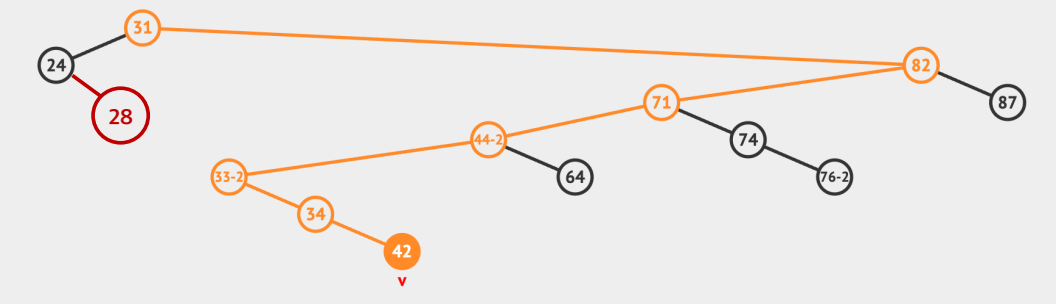

```
/*------------------------------------------------------------
  삽입 함수: bst_insert
  - 새로운 키(x)를 BST에 삽입하여 새로운 루트를 반환
  - 중복 키는 삽입하지 않고 무시함
  - 원리: 빈 자리를 찾을 때까지 내려가서 새 노드를 붙임
------------------------------------------------------------*/
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면
        Node* n = (Node*)malloc(sizeof(Node));   // 새 노드를 동적 할당
        if (!n) { perror("malloc"); exit(1); }   // 메모리 부족 시 오류 처리
        n->key = x;                              // 새 노드에 키 저장
        n->left = n->right = [redacted]          // 처음에는 자식이 없음
        return n;                                // 새 노드가 곧 루트가 됨
    }
```

노드 삽입




```c
/*------------------------------------------------------------
  삽입 함수: bst_insert
  - 새로운 키(x)를 BST에 삽입하여 새로운 루트를 반환
  - 중복 키는 삽입하지 않고 무시함
  - 원리: 빈 자리를 찾을 때까지 내려가서 새 노드를 붙임
------------------------------------------------------------*/
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면
        Node* n = (Node*)malloc(sizeof(Node));   // 새 노드를 동적 할당
        if (!n) { perror("malloc"); exit(1); }   // 메모리 부족 시 오류 처리
        n->key = x;                              // 새 노드에 키 저장
        n->left = n->right = NULL;               // 처음에는 자식이 없음
        return n;                                // 새 노드가 곧 루트가 됨
    }
```

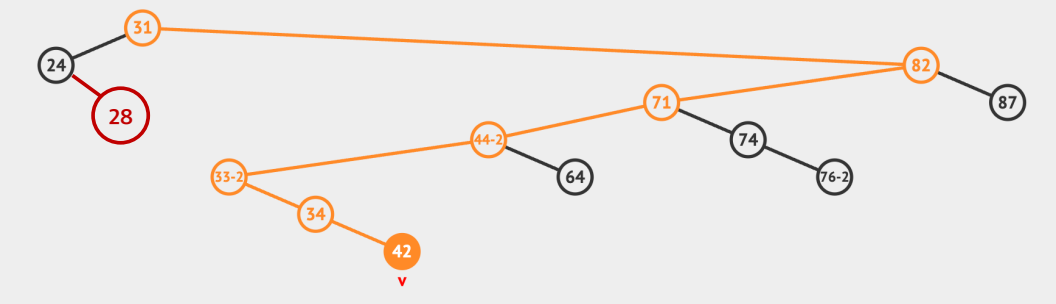
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복
        parent = [redacted]                      // 현재 노드를 부모로 기억
        if (x == cur->key) return [redacted]     // 이미 존재하는 키이면 삽입하지 않음
        if (x < cur->key) cur = [redacted]       // 작으면 왼쪽으로
        else              cur = [redacted]       // 크면 오른쪽으로
    }
```

김민서 (imkmsh@khu.ac.kr)

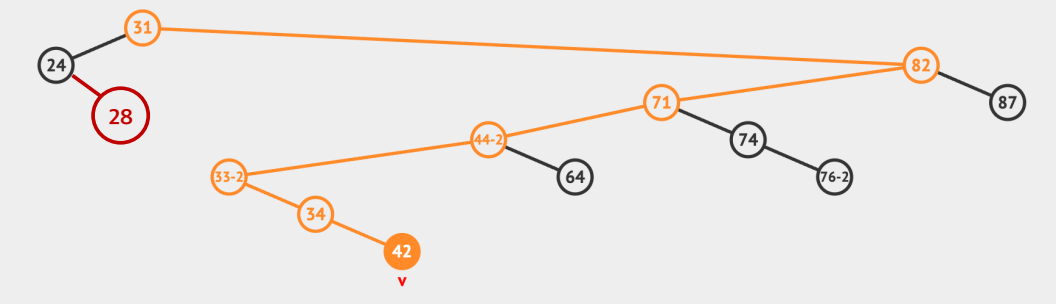
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복
        parent = cur;                            // 현재 노드를 부모로 기억
        if (x == cur->key) return [redacted]     // 이미 존재하는 키이면 삽입하지 않음
        if (x < cur->key) cur = [redacted]       // 작으면 왼쪽으로
        else              cur = [redacted]       // 크면 오른쪽으로
    }
```

김민서 (imkmsh@khu.ac.kr)

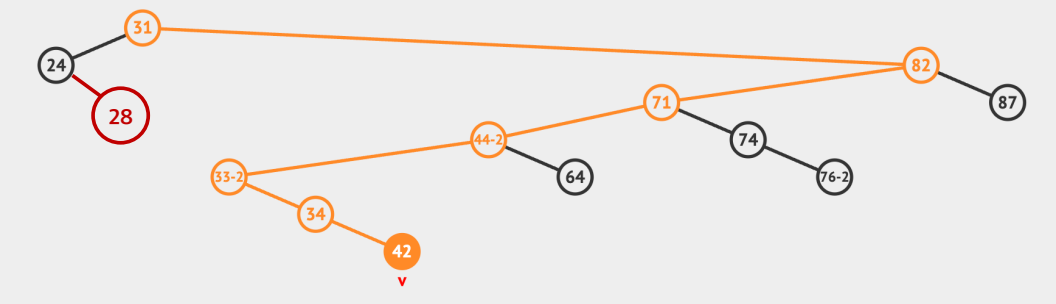
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복
        parent = cur;                            // 현재 노드를 부모로 기억
        if (x == cur->key) return root;          // 이미 존재하는 키이면 삽입하지 않음
        if (x < cur->key) cur =                  // 작으면 왼쪽으로
        else              cur =                  // 크면 오른쪽으로
    }
```

김민서 (imkmsh@khu.ac.kr)

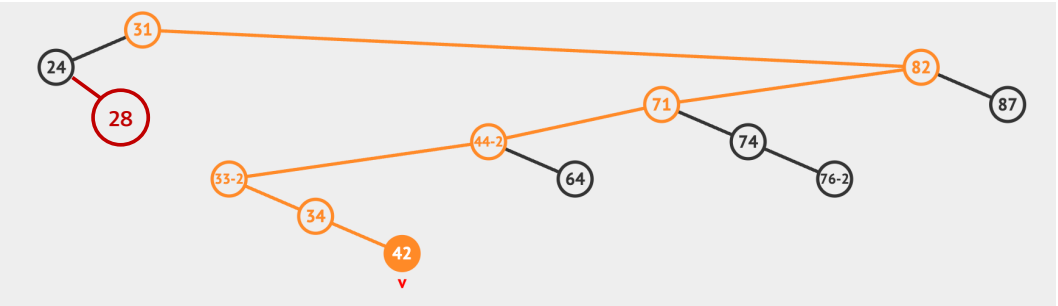
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복
        parent = cur;                            // 현재 노드를 부모로 기억
        if (x == cur->key) return root;          // 이미 존재하는 키이면 삽입하지 않음
        if (x < cur->key) cur = cur->left;       // 작으면 왼쪽으로
        else              cur =                  // 크면 오른쪽으로
    }
```

김민서 (imkmsh@khu.ac.kr)

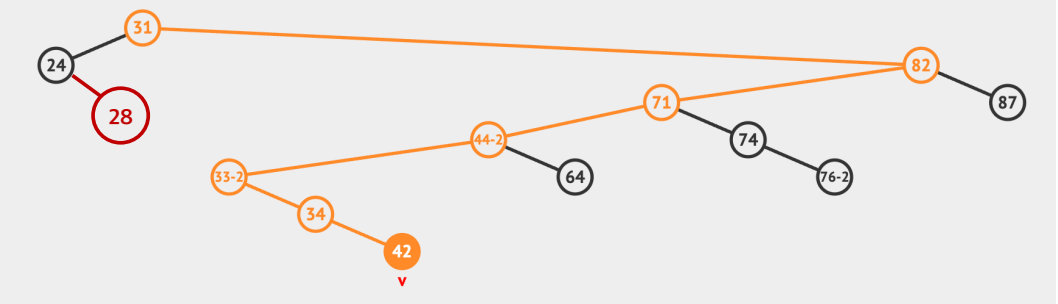
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복
        parent = cur;                            // 현재 노드를 부모로 기억
        if (x == cur->key) return root;          // 이미 존재하는 키이면 삽입하지 않음
        if (x < cur->key) cur = cur->left;       // 작으면 왼쪽으로
        else              cur = cur->right;      // 크면 오른쪽으로
    }
```

김민서 (imkmsh@khu.ac.kr)

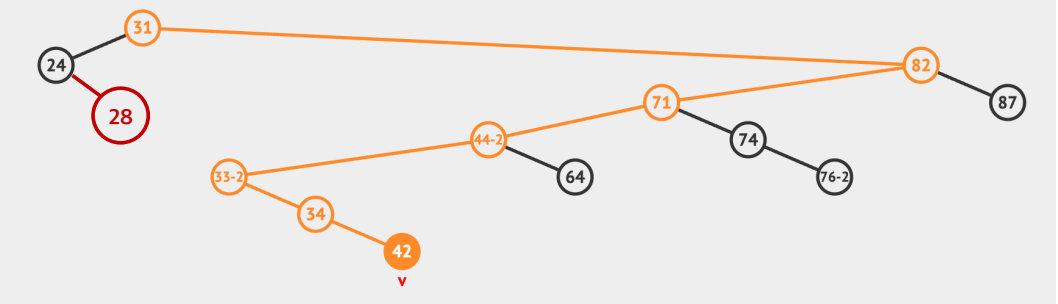
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복…
    }

    Node* n = (Node*)malloc(sizeof(Node));       // 새 노드를 생성
    if (!n) { perror("malloc"); exit(1); }       // 메모리 오류 처리
    n->key =                                     // 새 노드에 키 저장
    n->left = n->right =                         // 새 노드는 리프로 시작

    if (x < parent->key) parent->left =          // 삽입 위치가 부모보다 작으면 왼쪽에 연결
    else                 parent->right =         // 크면 오른쪽에 연결

    return root;                                 // 루트는 그대로 반환
}
```

김민서 (imkmsh@khu.ac.kr)

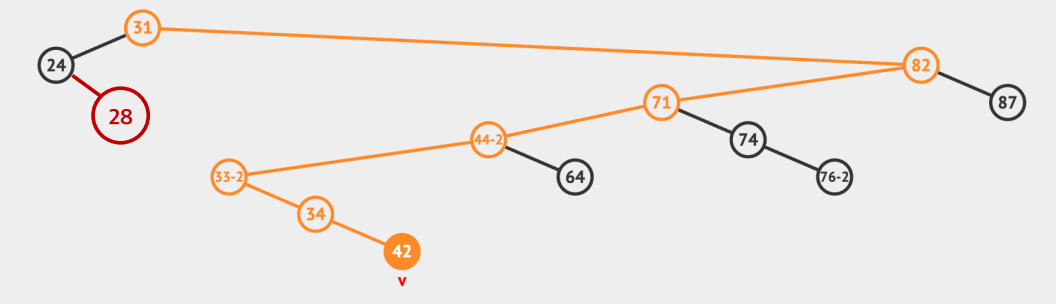
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복…
    }

    Node* n = (Node*)malloc(sizeof(Node));       // 새 노드를 생성
    if (!n) { perror("malloc"); exit(1); }       // 메모리 오류 처리
    n->key = x;                                  // 새 노드에 키 저장
    n->left = n->right =                         // 새 노드는 리프로 시작

    if (x < parent->key) parent->left =          // 삽입 위치가 부모보다 작으면 왼쪽에 연결
    else                 parent->right =         // 크면 오른쪽에 연결

    return root;                                 // 루트는 그대로 반환
}
```

김민서 (imkmsh@khu.ac.kr)

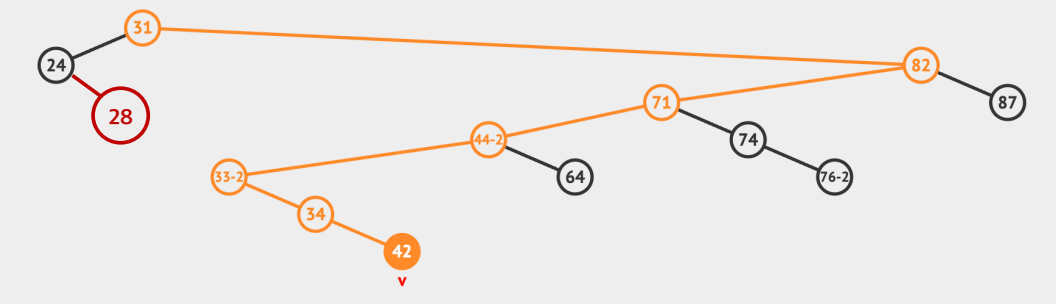
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복…
    }

    Node* n = (Node*)malloc(sizeof(Node));       // 새 노드를 생성
    if (!n) { perror("malloc"); exit(1); }       // 메모리 오류 처리
    n->key = x;                                  // 새 노드에 키 저장
    n->left = n->right = NULL;                   // 새 노드는 리프로 시작

    if (x < parent->key) parent->left =          // 삽입 위치가 부모보다 작으면 왼쪽에 연결
    else                 parent->right =         // 크면 오른쪽에 연결

    return root;                                 // 루트는 그대로 반환
}
```

김민서 (imkmsh@khu.ac.kr)

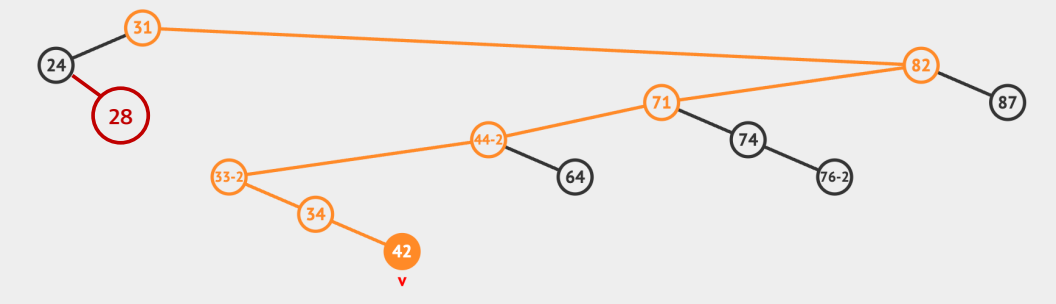
노드 삽입




```
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복…
    }

    Node* n = (Node*)malloc(sizeof(Node));       // 새 노드를 생성
    if (!n) { perror("malloc"); exit(1); }       // 메모리 오류 처리
    n->key = x;                                  // 새 노드에 키 저장
    n->left = n->right = NULL;                   // 새 노드는 리프로 시작

    if (x < parent->key) parent->left = n;       // 삽입 위치가 부모보다 작으면 왼쪽에 연결
    else                 parent->right = [redacted]  // 크면 오른쪽에 연결

    return root;                                 // 루트는 그대로 반환
}
```

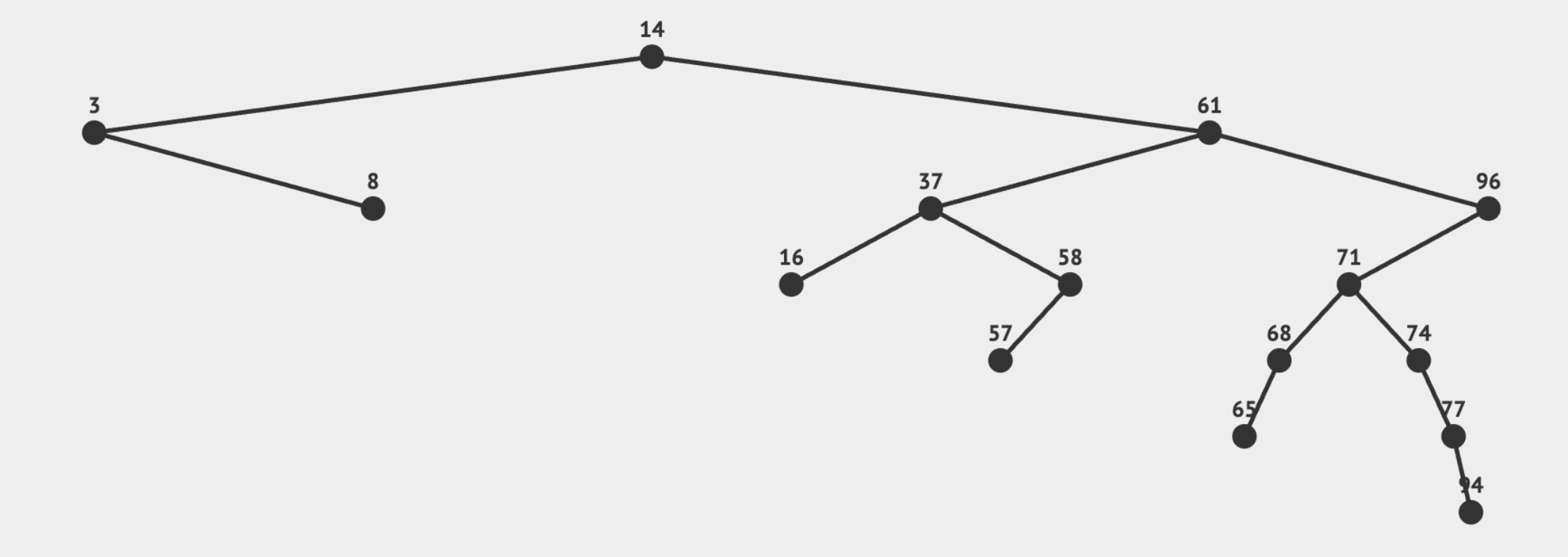
김민서 (imkmsh@khu.ac.kr)

노드 삽입



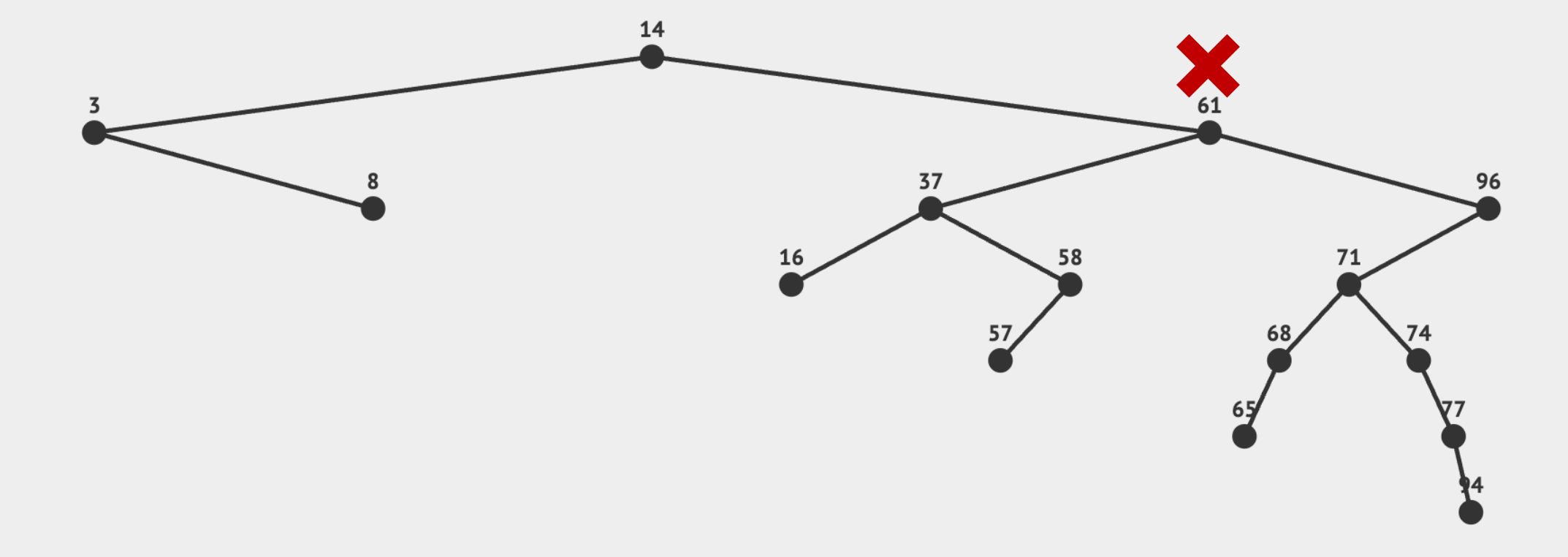

```c
Node* bst_insert(Node* root, Key x) {
    if (root == NULL) {                          // 트리가 비어 있으면…
    }

    Node* cur = root;                            // 현재 탐색 위치 (루트부터 시작)
    Node* parent = NULL;                         // 부모 노드를 저장할 변수
    while (cur) {                                // NULL을 만날 때까지 반복…
    }

    Node* n = (Node*)malloc(sizeof(Node));       // 새 노드를 생성
    if (!n) { perror("malloc"); exit(1); }       // 메모리 오류 처리
    n->key = x;                                  // 새 노드에 키 저장
    n->left = n->right = NULL;                   // 새 노드는 리프로 시작

    if (x < parent->key) parent->left = n;       // 삽입 위치가 부모보다 작으면 왼쪽에 연결
    else                 parent->right = n;      // 크면 오른쪽에 연결

    return root;                                 // 루트는 그대로 반환
}
```

노드 삭제



김민서 (imkmsh@khu.ac.kr)

노드 삭제

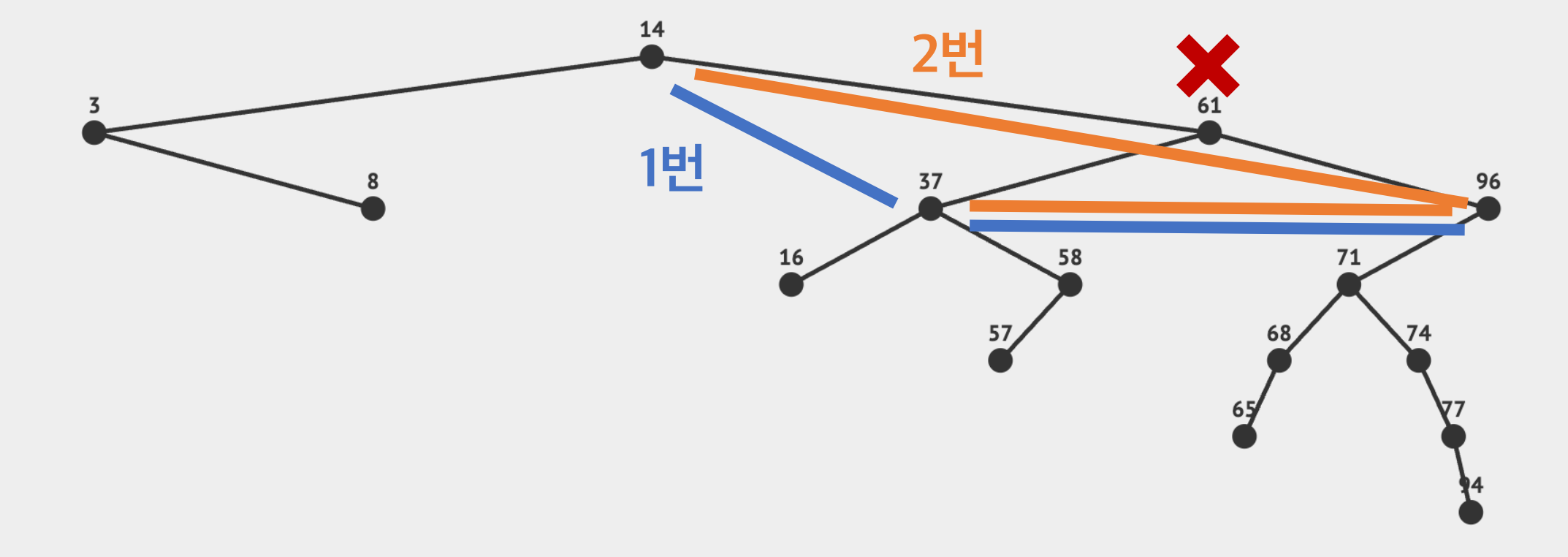


Let’s remove 61

김민서 (imkmsh@khu.ac.kr)

노드 삭제

3번: 그 외

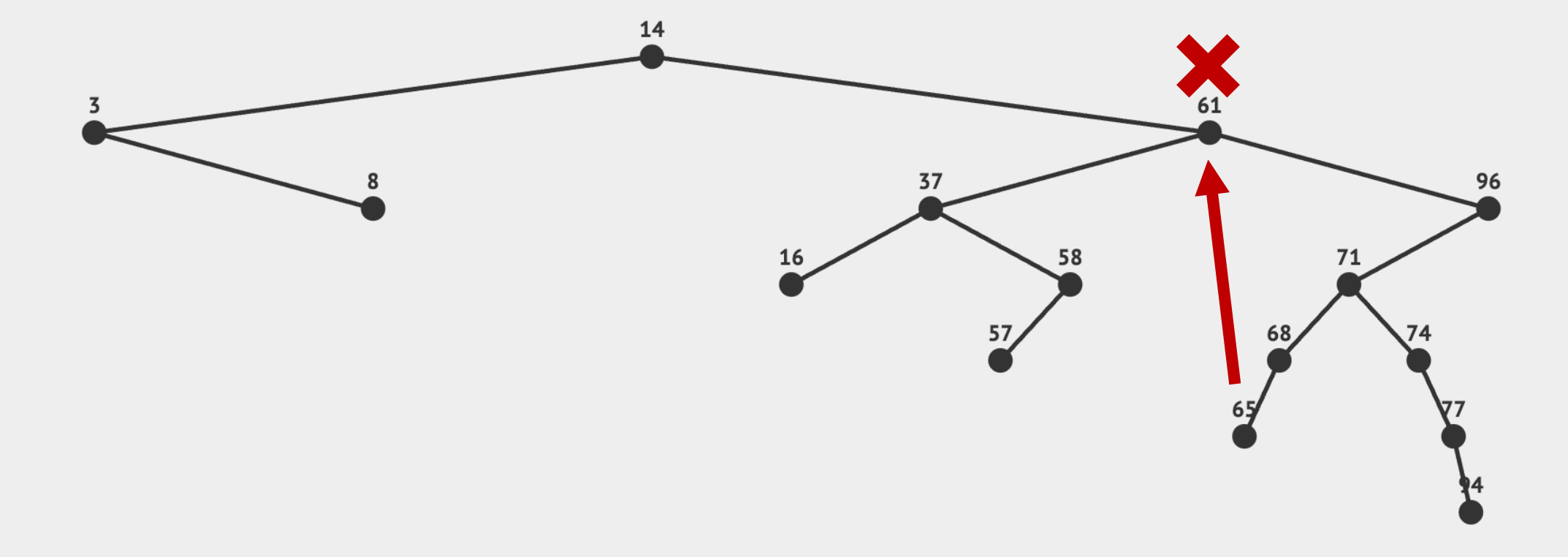


Let’s remove 61

김민서 (imkmsh@khu.ac.kr)

노드 삭제

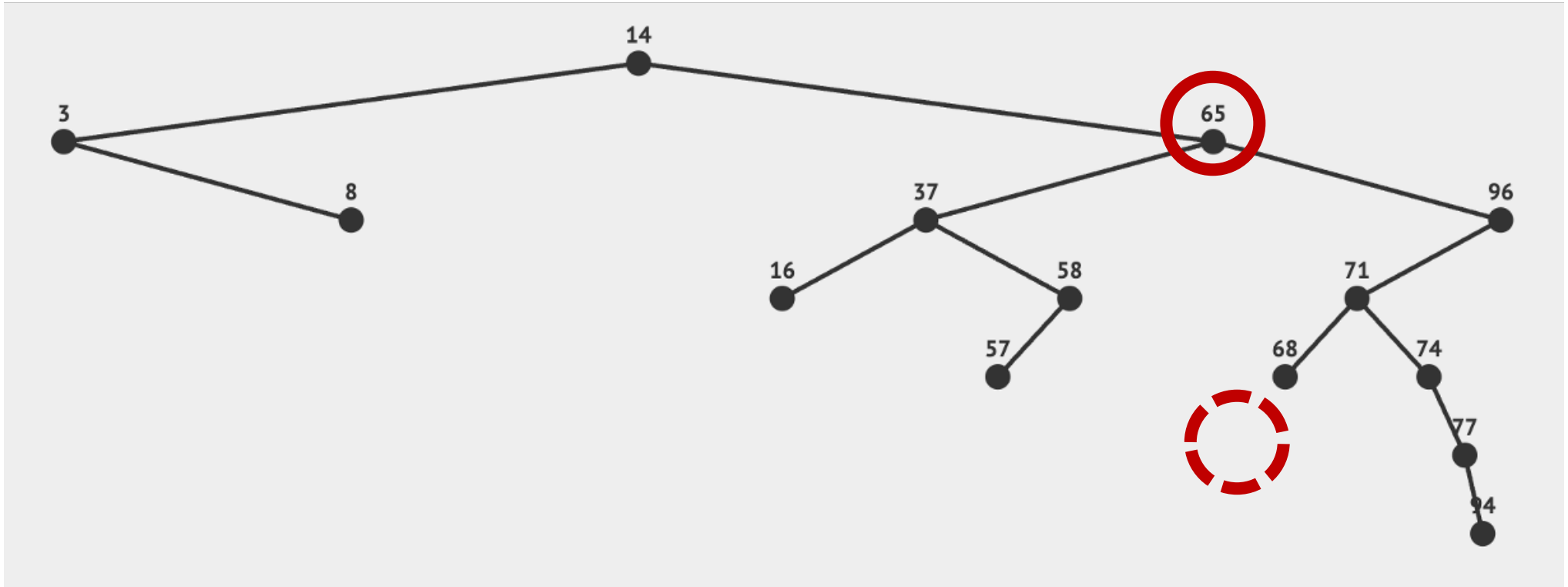
3번: 그 외



Let’s remove 61

김민서 (imkmsh@khu.ac.kr)

노드 삭제



Let’s remove 61

김민서 (imkmsh@khu.ac.kr)

노드 삭제 - 생각해 보기

- **Case 1. 자식이 없는 리프 노드**
  → 그냥 삭제 (연결 끊기만 하면 됨)

노드 삭제 - 생각해 보기

- **Case 1. 자식이 없는 리프 노드**
  → 그냥 삭제 (연결 끊기만 하면 됨)
- **Case 2. 자식이 1개인 노드**
  → **자식을 위로 올림**
  → 트리의 구조적/정렬적 성질이 깨지지 않음

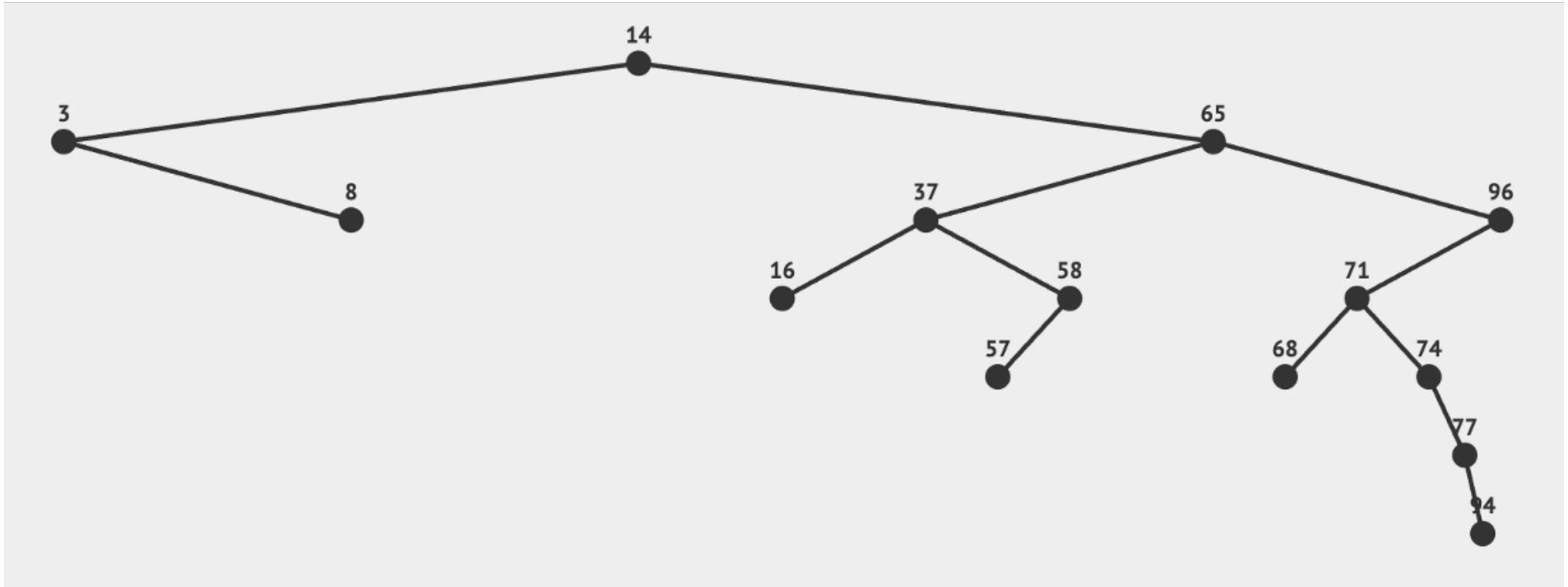
노드 삭제 - 생각해 보기

- **Case 1. 자식이 없는 리프 노드**
  → 그냥 삭제 (연결 끊기만 하면 됨)
- **Case 2. 자식이 1개인 노드**
  → **자식을 위로 올림**
  → 트리의 구조적/정렬적 성질이 깨지지 않음
- **Case 3. 자식이 2개인 노드**
  → 아무 자식이나 올릴 수 없음
  - 왼쪽 자식을 올리면, 오른쪽의 더 큰 값들이 작은 값 밑으로 들어가 버릴 수 있고,
  - 오른쪽 자식을 올리면, 왼쪽의 더 작은 값들이 큰 값 밑으로 들어가 버림
- 중간값 역할을 하는 노드를 찾아 교체해야 함

노드 삭제 - 생각해 보기

- **중간값 역할을 할 노드**
- **후계자 (Successor)**
  삭제할 노드의 **오른쪽 서브트리에서 가장 작은 값**
  → 삭제 노드보다 크고, 그 중 가장 작은 값
  → 트리 정렬 성질을 유지하면서 대체 가능
- **전임자 (Predecessor)**
  삭제할 노드의 **왼쪽 서브트리에서 가장 큰 값**
  → 삭제 노드보다 작고, 그 중 가장 큰 값

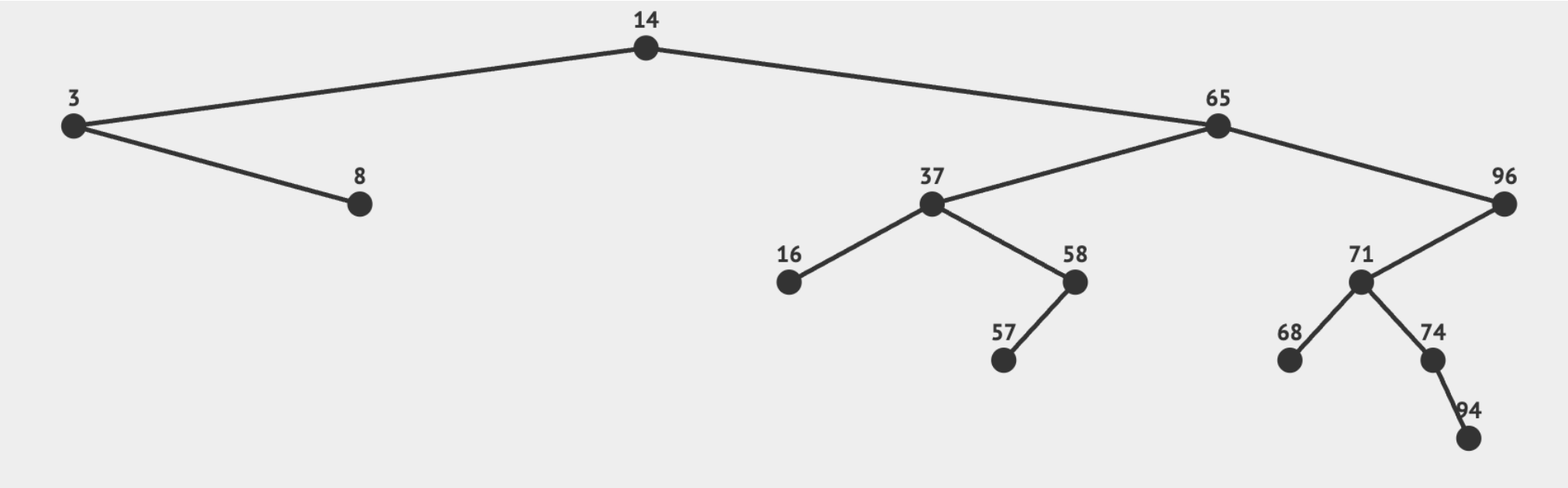
노드 삭제



Let’s remove 77

김민서 (imkmsh@khu.ac.kr)

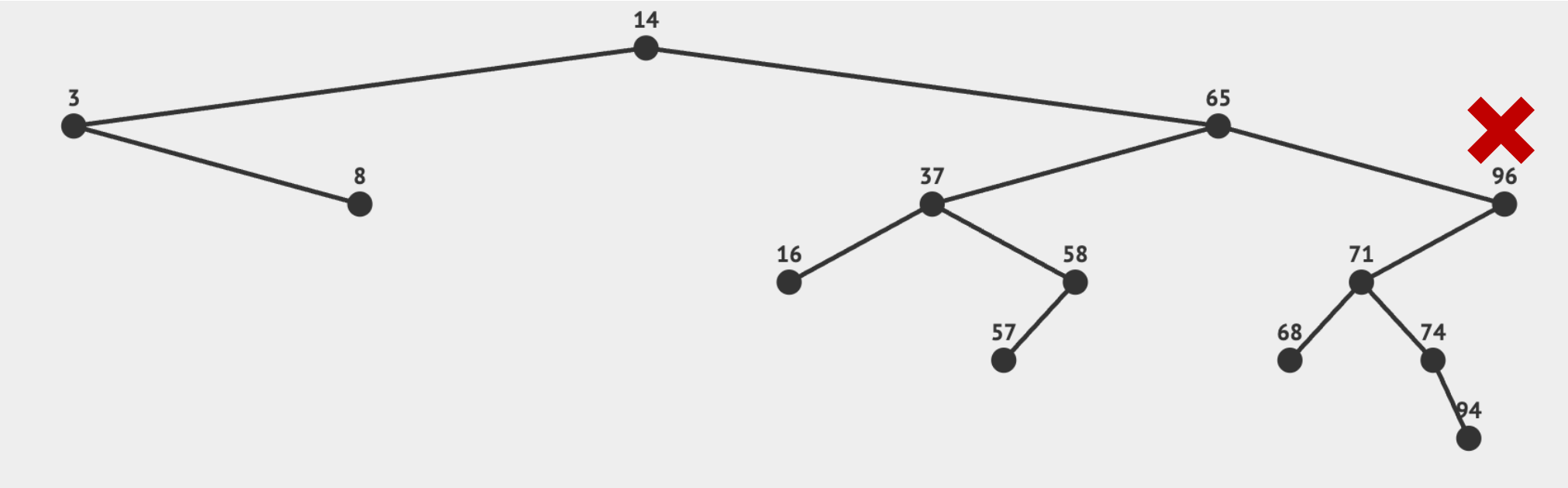
노드 삭제



Let’s remove 77

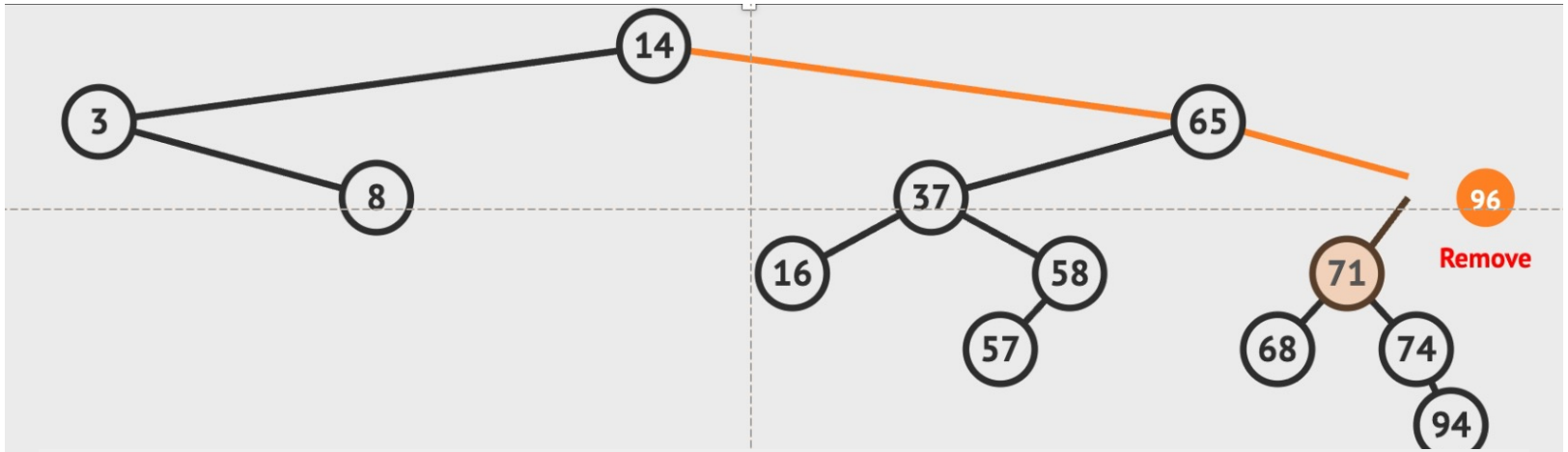
김민서 (imkmsh@khu.ac.kr)

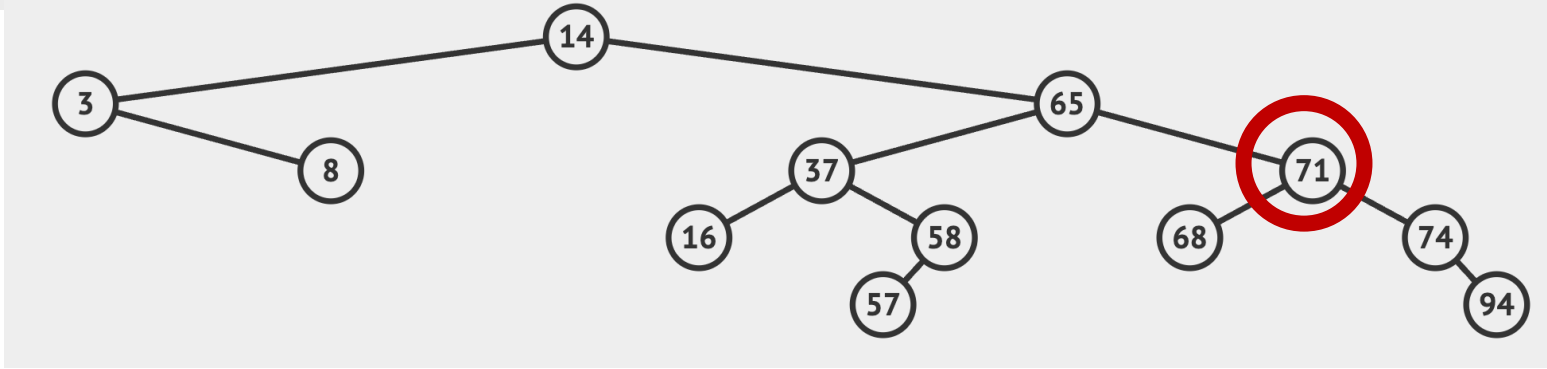
노드 삭제



Let’s remove 96

노드 삭제

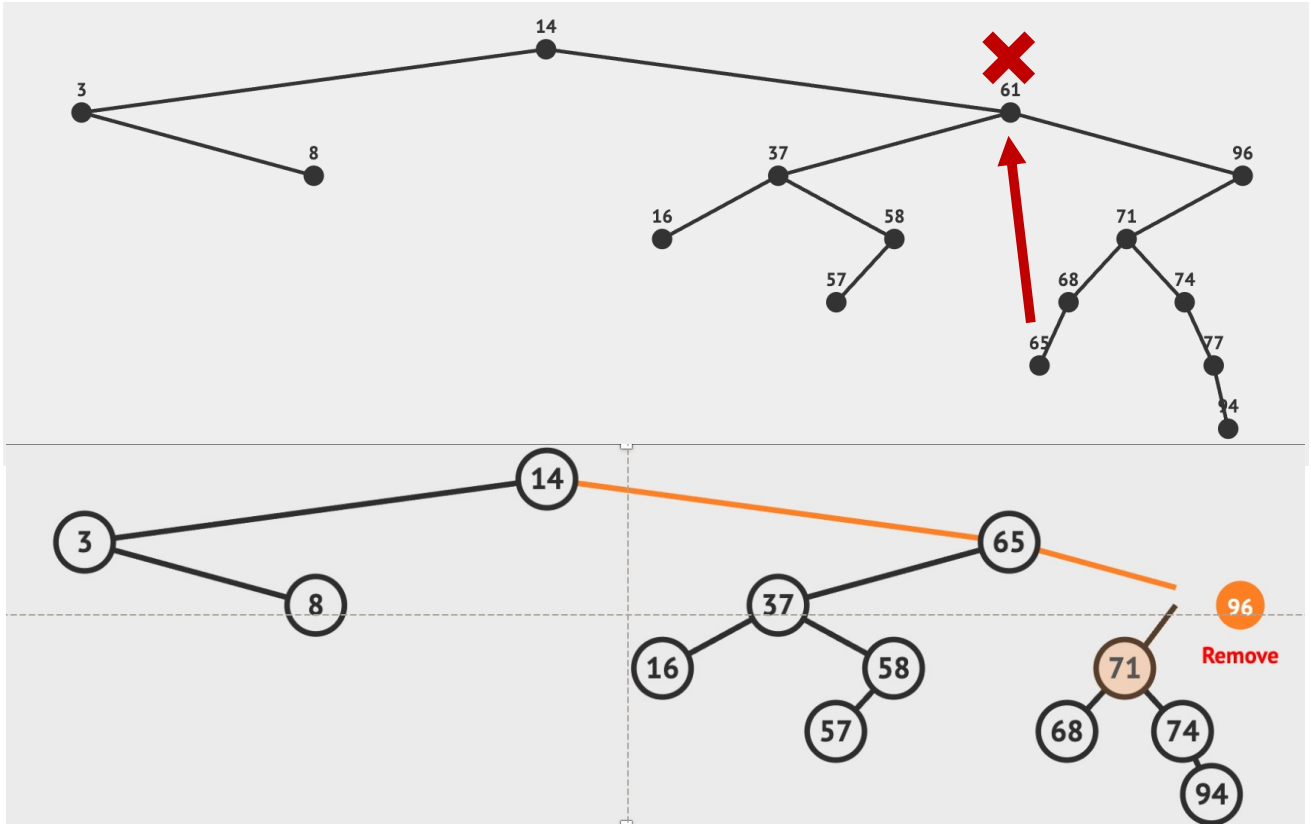




Let’s remove 96

김민서 (imkmsh@khu.ac.kr)

노드 삭제 - 생각해 보기



뭐가 다를까요?

김민서 (imkmsh@khu.ac.kr)

노드 삭제

```c
/*------------------------------------------------------------
  삭제 함수: bst_delete
  - 키(x)를 가진 노드를 트리에서 제거
  - 자식의 개수(0개, 1개, 2개)에 따라 세 가지로 분리
  - 원리: 후계자(successor) 노드로 키를 대체하여 구조 유지
------------------------------------------------------------*/
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드…
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우…
    }

    // (3) 자식이 2개인 경우 – 오른쪽 서브트리에서 가장 작은 키(후계자)를 찾아 키를 대체
    Node* succ = NEED_TO_SOLVE;                       // 오른쪽 서브트리의 루트로 시작
    while (succ->left) succ = NEED_TO_SOLVE;          // 왼쪽으로 계속 내려가 최소값 찾기
    root->key = succ->key;                          // 현재 노드의 키를 후계자의 키로 변경
    root->right = bst_delete(root->right, NEED_TO_SOLVE); // 오른쪽 서브트리에서 후계자 삭제
    return root;                                    // 루트 반환 (구조 유지)
}
```

김민서 (imkmsh@khu.ac.kr)

노드 삭제

```
/*------------------------------------------------------------
  삭제 함수: bst_delete
  - 키(x)를 가진 노드를 트리에서 제거
  - 자식의 개수(0개, 1개, 2개)에 따라 세 가지로 분리
  - 원리: 후계자(successor) 노드로 키를 대체하여 구조 유지
------------------------------------------------------------*/
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                    // 빈 트리면 바로 반환

    if (x < root->key) {                              // 삭제할 키가 루트보다 작으면
        root->left = [redacted](root->left, x);       // 왼쪽 서브트리에서 삭제 후 연결
        return root;                                  // 전체 루트는 그대로 유지
    }

    if (x > root->key) {                              // 삭제할 키가 루트보다 크면
        [redacted]                                    // 오른쪽 서브트리에서 삭제 후 연결
        return root;                                  // 전체 루트는 그대로 유지
    }
```

김민서 (imkmsh@khu.ac.kr)

노드 삭제

```
/*------------------------------------------------------------
  삭제 함수: bst_delete
  - 키(x)를 가진 노드를 트리에서 제거
  - 자식의 개수(0개, 1개, 2개)에 따라 세 가지로 분리
  - 원리: 후계자(successor) 노드로 키를 대체하여 구조 유지
------------------------------------------------------------*/
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                    // 빈 트리면 바로 반환

    if (x < root->key) {                              // 삭제할 키가 루트보다 작으면
        root->left = bst_delete(root->left, x);       // 왼쪽 서브트리에서 삭제 후 연결
        return root;                                  // 전체 루트는 그대로 유지
    }

    if (x > root->key) {                              // 삭제할 키가 루트보다 크면
        [redacted]                                    // 오른쪽 서브트리에서 삭제 후 연결
        return root;                                  // 전체 루트는 그대로 유지
    }
```

김민서 (imkmsh@khu.ac.kr)

노드 삭제

```c
/*------------------------------------------------------------
  삭제 함수: bst_delete
  - 키(x)를 가진 노드를 트리에서 제거
  - 자식의 개수(0개, 1개, 2개)에 따라 세 가지로 분리
  - 원리: 후계자(successor) 노드로 키를 대체하여 구조 유지
------------------------------------------------------------*/
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                    // 빈 트리면 바로 반환

    if (x < root->key) {                              // 삭제할 키가 루트보다 작으면
        root->left = bst_delete(root->left, x);       // 왼쪽 서브트리에서 삭제 후 연결
        return root;                                  // 전체 루트는 그대로 유지
    }

    if (x > root->key) {                              // 삭제할 키가 루트보다 크면
        root->right = bst_delete(root->right, x);     // 오른쪽 서브트리에서 삭제 후 연결
        return root;                                  // 전체 루트는 그대로 유지
    }
```

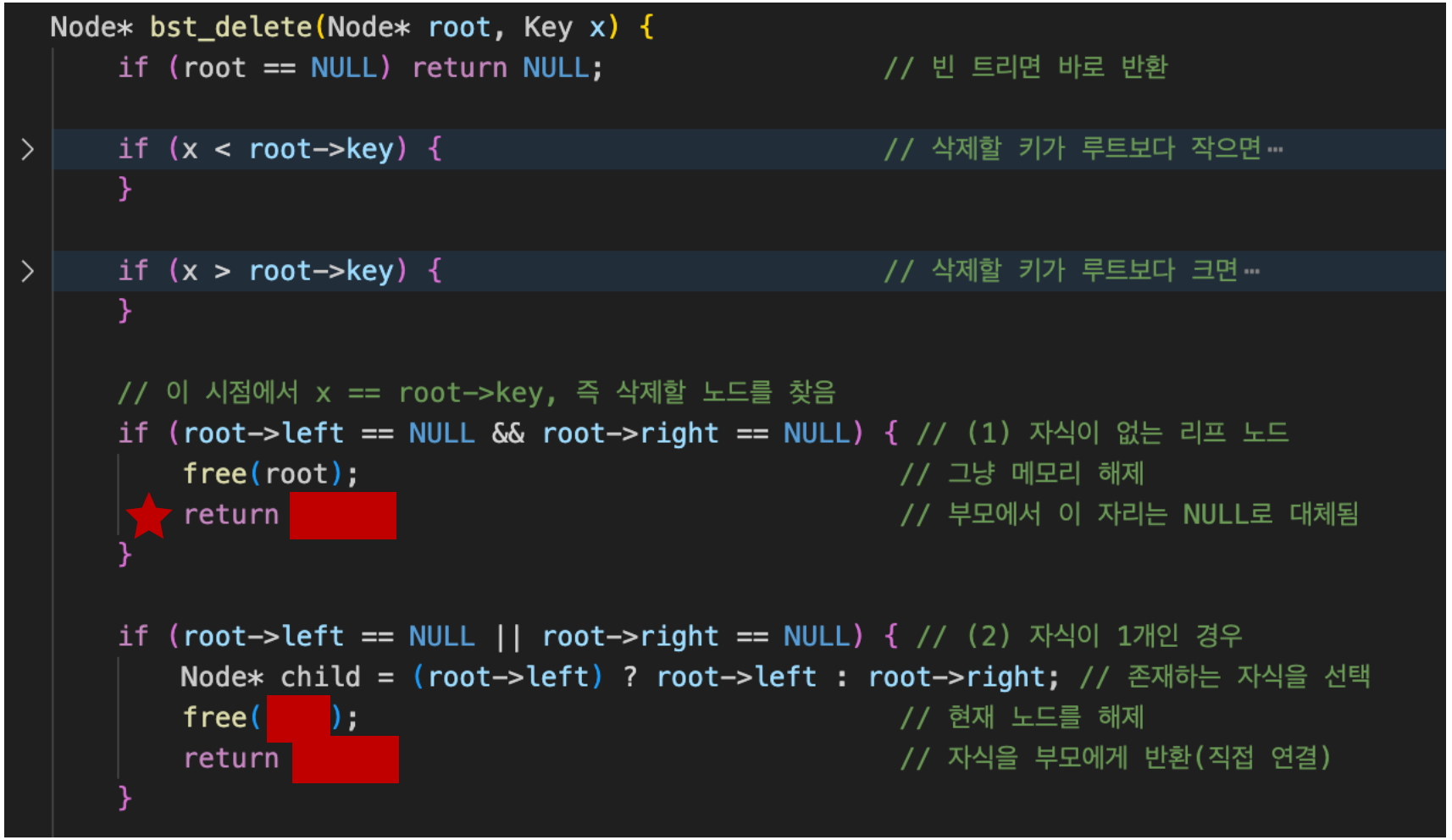
김민서 (imkmsh@khu.ac.kr)

노드 삭제

```
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드
        free( );                                     // 그냥 메모리 해제
        return                                       // 부모에서 이 자리는 NULL로 대체됨
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우
        Node* child = (root->left) ? root->left : root->right; // 존재하는 자식을 선택
        free( );                                     // 현재 노드를 해제
        return                                       // 자식을 부모에게 반환(직접 연결)
    }
```

김민서 (imkmsh@khu.ac.kr)

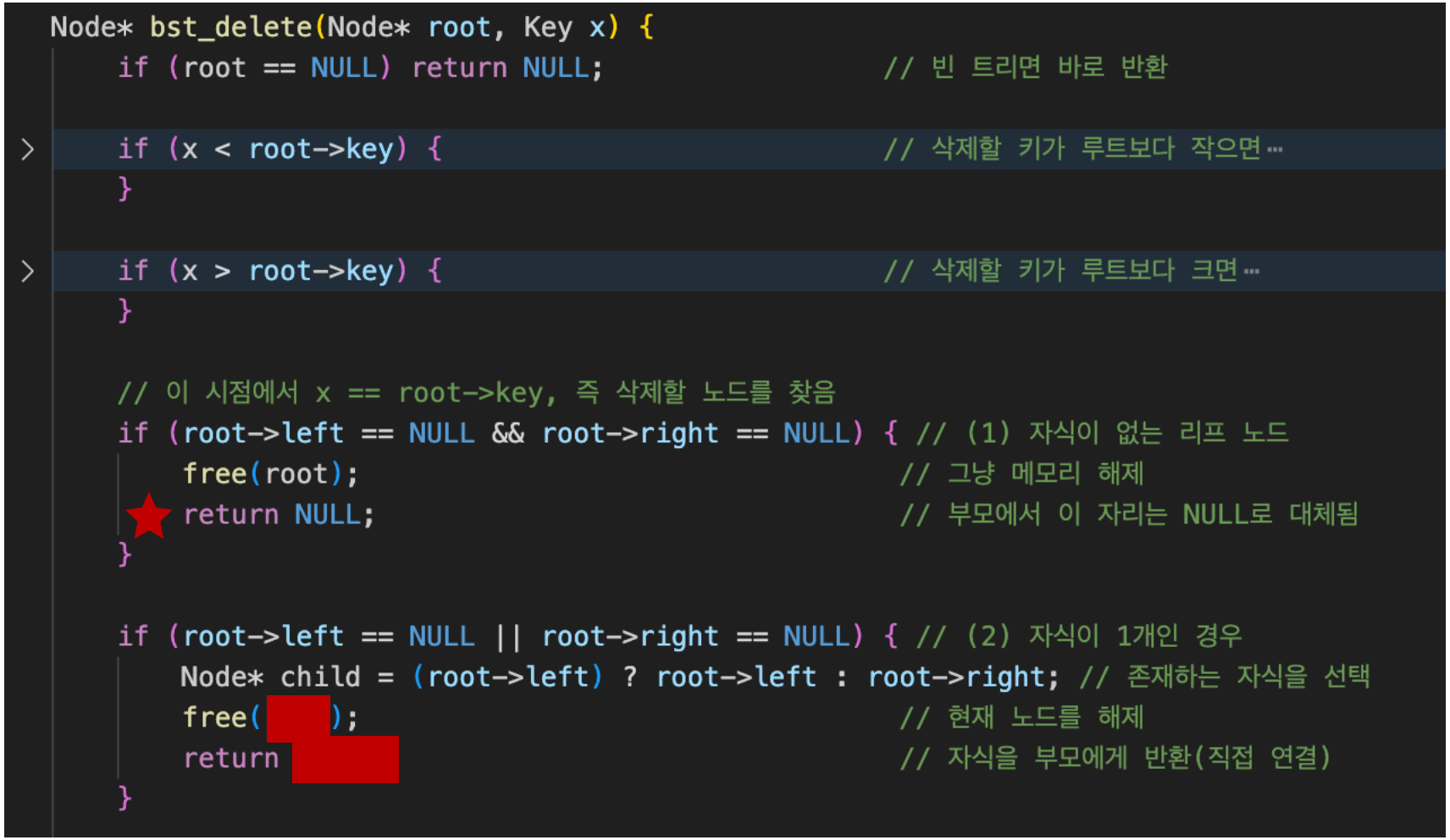
노드 삭제



```
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드
        free(root);                                  // 그냥 메모리 해제
        return                                       // 부모에서 이 자리는 NULL로 대체됨
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우
        Node* child = (root->left) ? root->left : root->right; // 존재하는 자식을 선택
        free(    );                                  // 현재 노드를 해제
        return                                       // 자식을 부모에게 반환(직접 연결)
    }
```

김민서 (imkmsh@khu.ac.kr)

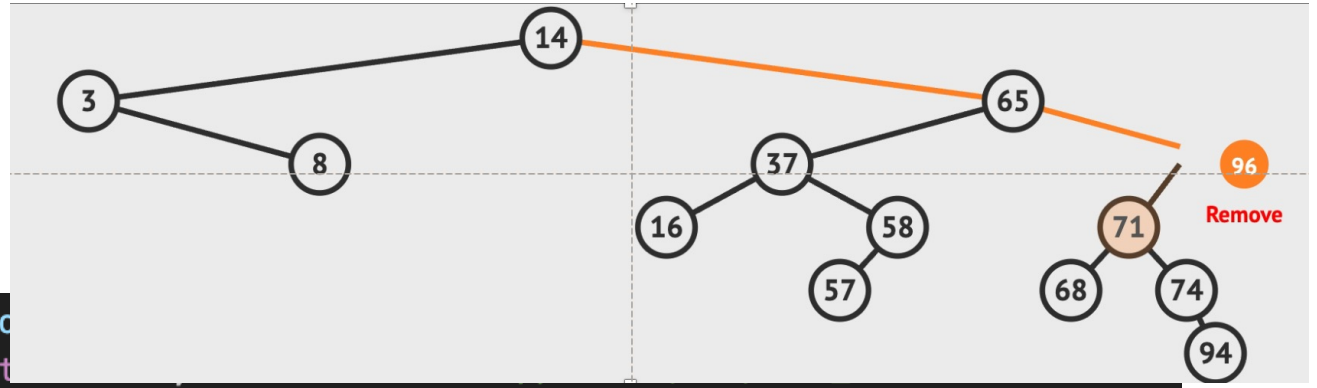
노드 삭제



```
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드
        free(root);                                  // 그냥 메모리 해제
        return NULL;                                 // 부모에서 이 자리는 NULL로 대체됨
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우
        Node* child = (root->left) ? root->left : root->right; // 존재하는 자식을 선택
        free(      );                                // 현재 노드를 해제
        return                                       // 자식을 부모에게 반환(직접 연결)
    }
```

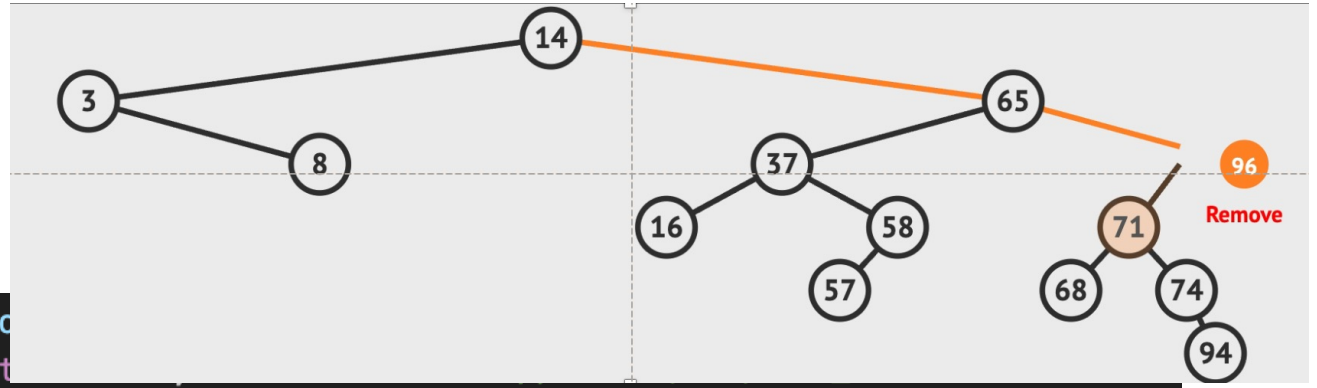
김민서 (imkmsh@khu.ac.kr)

노드 삭제




```
Node* bst_delete(Node* ro
    if (root == NULL) ret

    if (x < root->key) {                                    // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                                    // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드
        free(root);                                          // 그냥 메모리 해제
        return NULL;                                         // 부모에서 이 자리는 NULL로 대체됨
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우
        Node* child = (root->left) ? root->left : root->right; // 존재하는 자식을 선택
        free(    );                                          // 현재 노드를 해제
        return                                               // 자식을 부모에게 반환(직접 연결)
    }
```

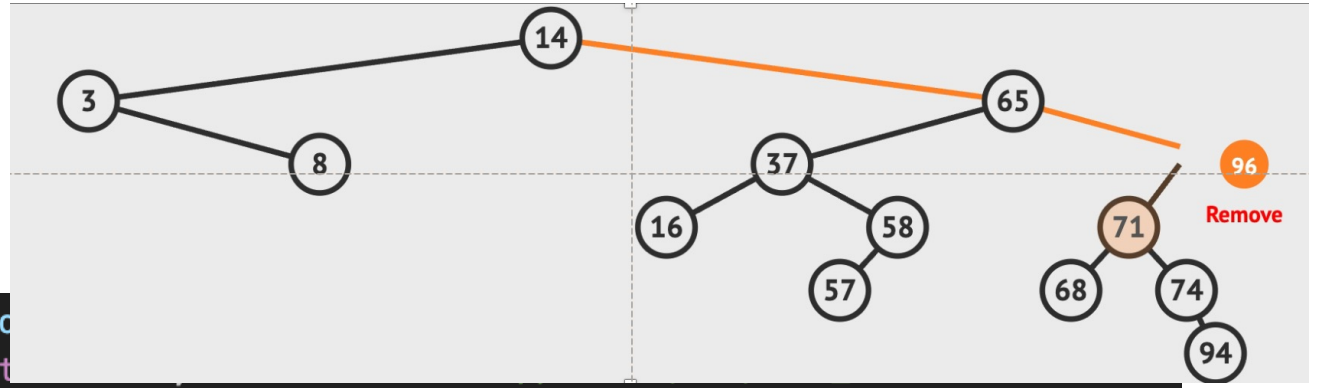
노드 삭제




```
Node* bst_delete(Node* ro
    if (root == NULL) ret

    if (x < root->key) {                          // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                          // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드
        free(root);                               // 그냥 메모리 해제
        return NULL;                              // 부모에서 이 자리는 NULL로 대체됨
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우
        Node* child = (root->left) ? root->left : root->right; // 존재하는 자식을 선택
        free(root);                               // 현재 노드를 해제
        return                                    // 자식을 부모에게 반환(직접 연결)
    }
```

김민서 (imkmsh@khu.ac.kr)

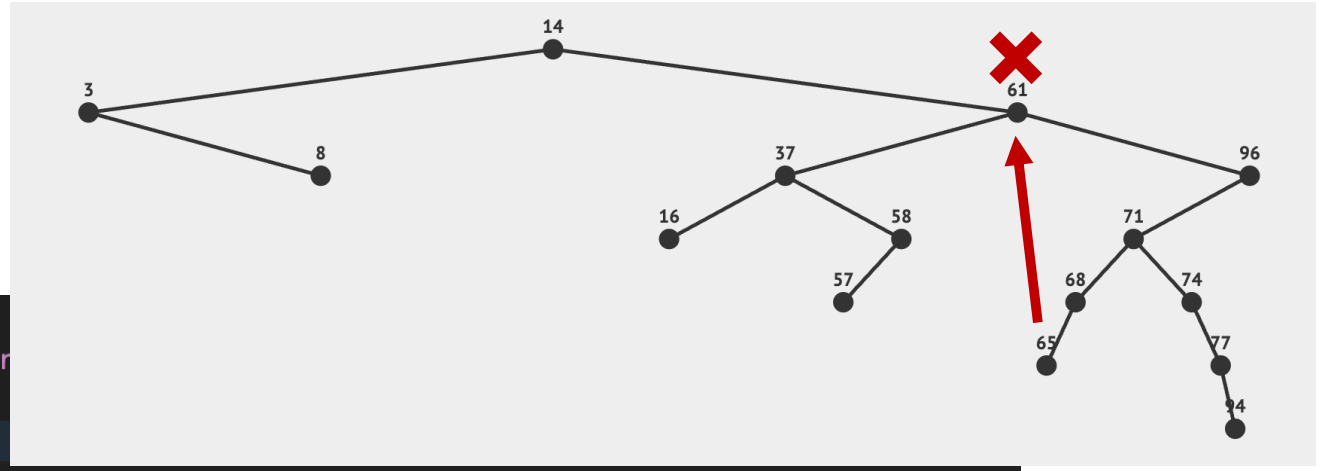
노드 삭제




```c
Node* bst_delete(Node* ro
    if (root == NULL) ret

    if (x < root->key) {                          // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                          // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드
        free(root);                               // 그냥 메모리 해제
        return NULL;                              // 부모에서 이 자리는 NULL로 대체됨
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우
        Node* child = (root->left) ? root->left : root->right; // 존재하는 자식을 선택
        free(root);                               // 현재 노드를 해제
        return child;                             // 자식을 부모에게 반환(직접 연결)
    }
```

김민서 (imkmsh@khu.ac.kr)

노드 삭제




```
Node* bst_delete(Node*
    if (root == NULL) r

    if (x < root->key)
    }

    if (x > root->key) {                          // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드…
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우…
    }

    // (3) 자식이 2개인 경우
    // 오른쪽 서브트리에서 가장 작은 키(후계자)를 찾아 키를 대체
    Node* succ =                                  // 오른쪽 서브트리의 루트로 시작
    while (succ->left) succ =                     // 왼쪽으로 계속 내려가 최소값 찾기
    root->key = succ->key;                        // 현재 노드의 키를 후계자의 키로 변경
    root->right =          (root->right,          ); // 오른쪽 서브트리에서 후계자 삭제
    return root;                                  // 루트 반환 (구조 유지)
}
```

김민서 (imkmsh@khu.ac.kr)

노드 삭제

```
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드…
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우…
    }

    // (3) 자식이 2개인 경우
    // 오른쪽 서브트리에서 가장 작은 키(후계자)를 찾아 키를 대체
    Node* succ = root->right;                       // 오른쪽 서브트리의 루트로 시작
    while (succ->left) succ =                       // 왼쪽으로 계속 내려가 최소값 찾기
    root->key = succ->key;                          // 현재 노드의 키를 후계자의 키로 변경
    root->right =          (root->right,          ); // 오른쪽 서브트리에서 후계자 삭제
    return root;                                    // 루트 반환 (구조 유지)
}
```

김민서 (imkmsh@khu.ac.kr)

노드 삭제

```
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드…
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우…
    }

    // (3) 자식이 2개인 경우
    // 오른쪽 서브트리에서 가장 작은 키(후계자)를 찾아 키를 대체
    Node* succ = root->right;                       // 오른쪽 서브트리의 루트로 시작
    while (succ->left) succ = succ->left;           // 왼쪽으로 계속 내려가 최소값 찾기
    root->key = succ->key;                          // 현재 노드의 키를 후계자의 키로 변경
    root->right = [redacted](root->right, [redacted]); // 오른쪽 서브트리에서 후계자 삭제
    return root;                                    // 루트 반환 (구조 유지)
}
```

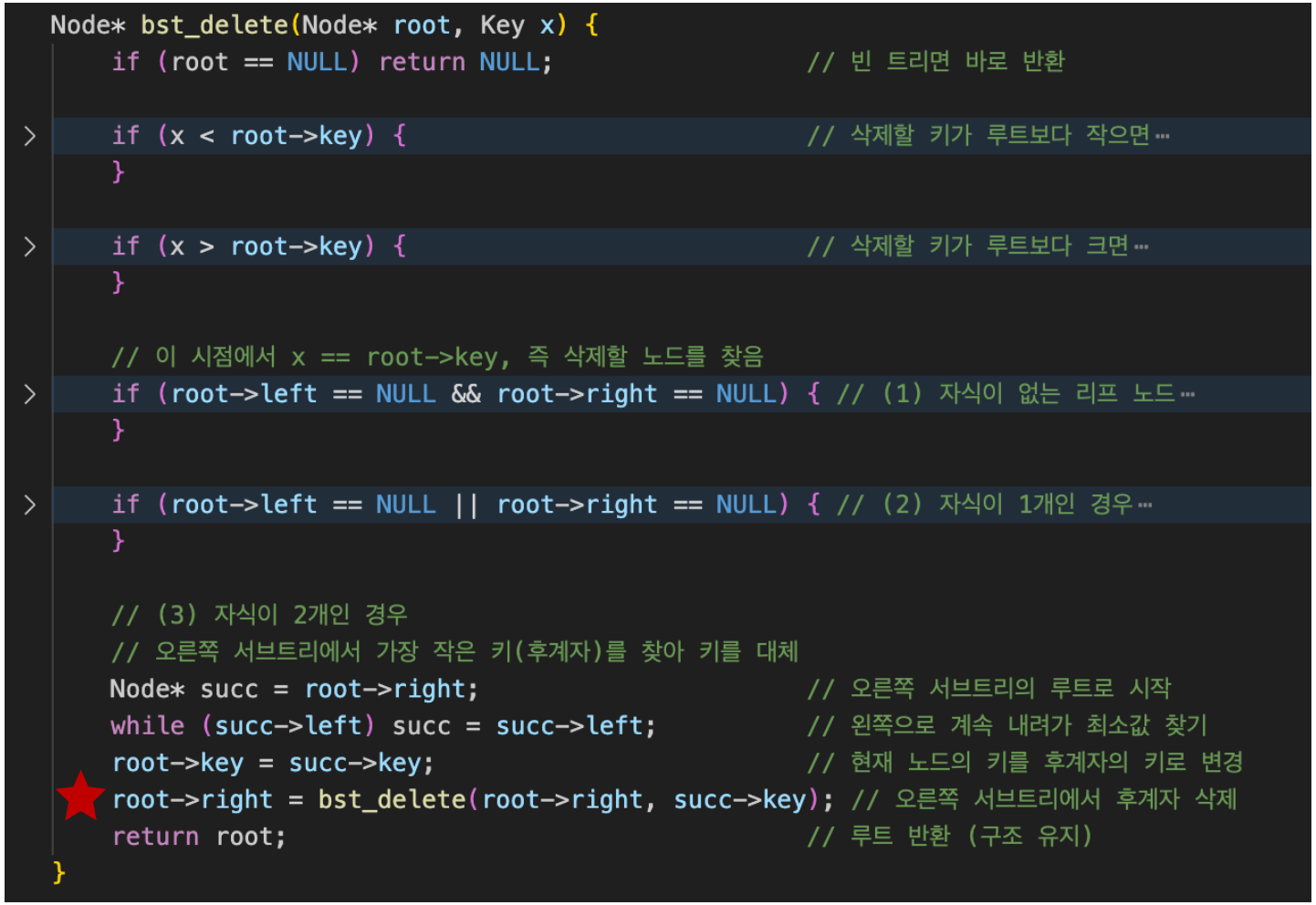
김민서 (imkmsh@khu.ac.kr)

노드 삭제

```
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드…
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우…
    }

    // (3) 자식이 2개인 경우
    // 오른쪽 서브트리에서 가장 작은 키(후계자)를 찾아 키를 대체
    Node* succ = root->right;                       // 오른쪽 서브트리의 루트로 시작
    while (succ->left) succ = succ->left;           // 왼쪽으로 계속 내려가 최소값 찾기
    root->key = succ->key;                          // 현재 노드의 키를 후계자의 키로 변경
    root->right = bst_delete(root->right, [redacted]); // 오른쪽 서브트리에서 후계자 삭제
    return root;                                    // 루트 반환 (구조 유지)
}
```

김민서 (imkmsh@khu.ac.kr)

노드 삭제



```
Node* bst_delete(Node* root, Key x) {
    if (root == NULL) return NULL;                  // 빈 트리면 바로 반환

    if (x < root->key) {                            // 삭제할 키가 루트보다 작으면…
    }

    if (x > root->key) {                            // 삭제할 키가 루트보다 크면…
    }

    // 이 시점에서 x == root->key, 즉 삭제할 노드를 찾음
    if (root->left == NULL && root->right == NULL) { // (1) 자식이 없는 리프 노드…
    }

    if (root->left == NULL || root->right == NULL) { // (2) 자식이 1개인 경우…
    }

    // (3) 자식이 2개인 경우
    // 오른쪽 서브트리에서 가장 작은 키(후계자)를 찾아 키를 대체
    Node* succ = root->right;                       // 오른쪽 서브트리의 루트로 시작
    while (succ->left) succ = succ->left;           // 왼쪽으로 계속 내려가 최소값 찾기
    root->key = succ->key;                          // 현재 노드의 키를 후계자의 키로 변경
    root->right = bst_delete(root->right, succ->key); // 오른쪽 서브트리에서 후계자 삭제
    return root;                                    // 루트 반환 (구조 유지)
}
```

Main.c 실행

```c
int main(void){
    Node* root = NULL;

    // 삽입
    int vals[] = { 50, 30, 70, 20, 40, 60, 80 };
    for(size_t i=0;i<sizeof(vals)/sizeof(vals[0]);++i)
        root = bst_insert(root, vals[i]);

    printf("중위순회(오름차순): ");
    bst_inorder(root); puts("");

    // 검색
    int q[] = {40, 25, 80};
    for(int i=0;i<3;i++){
        Node* found = bst_search(root, q[i]);
        printf("search %d -> %s\n", q[i], found ? "found" : "not found");
    }

    // 삭제: 자식 0, 1, 2 케이스 차례로
    root = bst_delete(root, 20);  // leaf
    root = bst_delete(root, 30);  // one-child
    root = bst_delete(root, 50);  // two-children (root)

    printf("삭제 후 중위순회: ");
    bst_inorder(root); puts("");

    bst_free(root);
    return 0;
}
```

```
중위순회(오름차순): 20 30 40 50 60 70 80
search 40 -> found
search 25 -> not found
search 80 -> found
삭제 후 중위순회: 40 60 70 80
```

값들을 직접 바꿔가며 실행해 보세요

김민서 (imkmsh@khu.ac.kr)

수고하셨습니다

자료구조 실습 11/12

EOF

김민서 (imkmsh@khu.ac.kr)