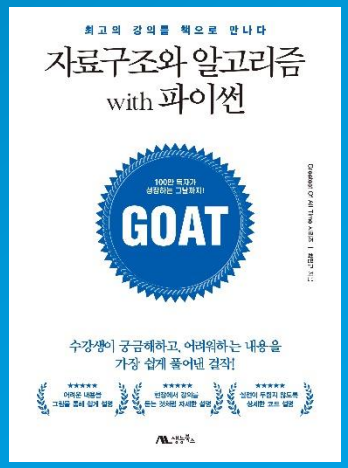


CHAPTER

07 탐색

SW알고리즘개발
11주차

7장. 탐색

07-1 탐색이란?

07-2 순차 탐색

07-3 이진 탐색

07-4 이진 탐색 트리

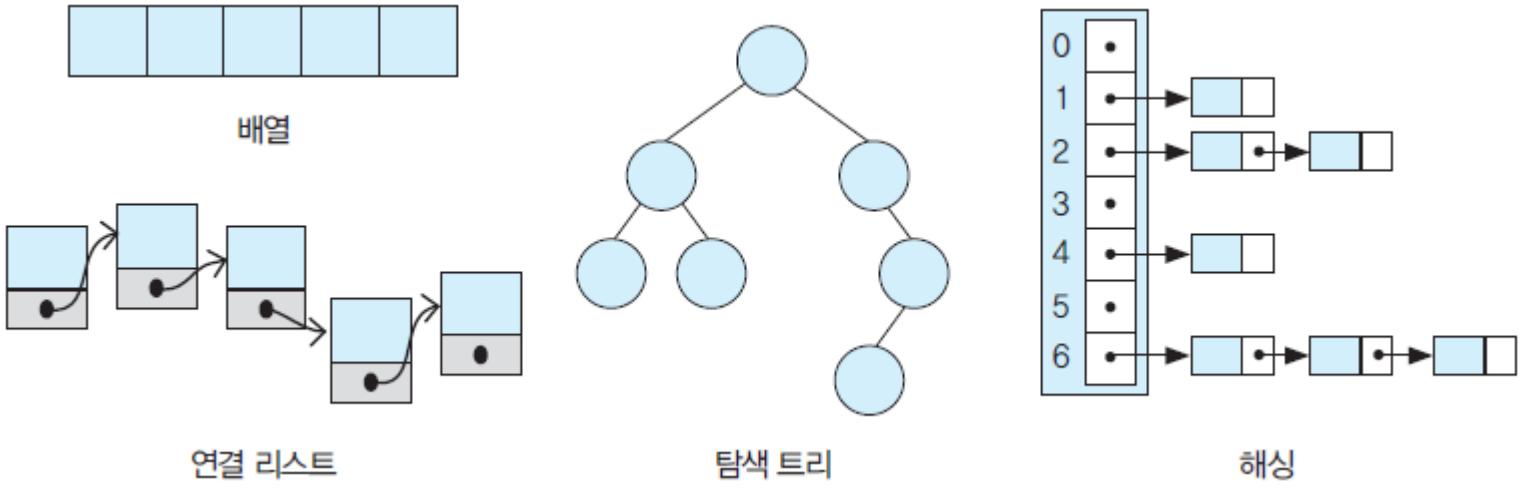
7.1 탐색이란?

- 데이터의 집합에서 원하는 조건을 만족하는 데이터(레코드, record)를 찾는 작업

- 테이블(Table) : 레코드의 집합
- 레코드(record) : 여러 개의 필드로 구성
- 키(key, 탐색키) : 탐색의 기준이 되는 필드
- 탐색
  - 테이블에서 원하는 탐색키를 가진 레코드를 찾는 작업

탐색을 위한 테이블

- 테이블을 구성하는 방법에 따라 효율이 달라짐
  - 배열, 연결 리스트
  - 이진 탐색 트리, 해싱(12장) 등



연결 리스트 탐색 트리 해싱

- 탐색 방법 선택 시 고려할 점
  - 탐색 연산 성능, 삽입 성능, 삭제 성능

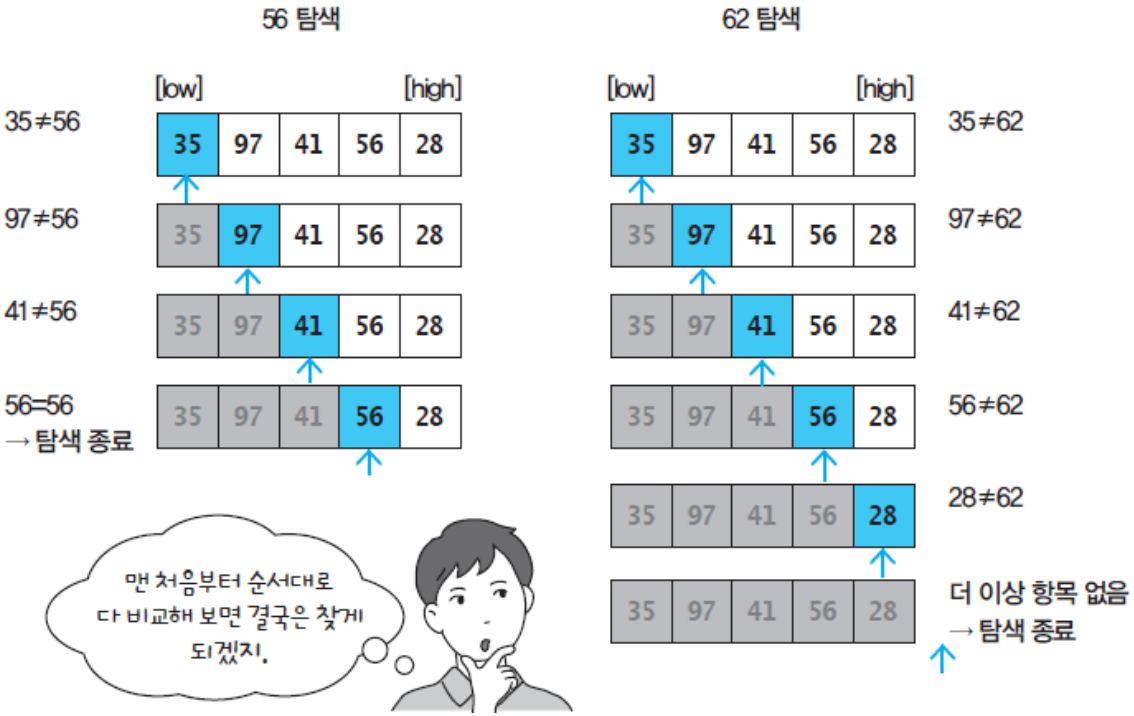
데이터 구조 기반 탐색 알고리즘의 효율성

- 탐색 알고리즘에서 사용되는 데이터 구조인 배열, 연결 리스트, 탐색 트리, 해싱의 효율성은 각 데이터 구조가 특정 연산을 어떻게 처리하는지에 따라 다름.

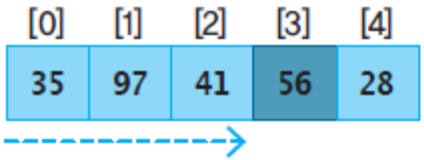
연산	배열 (Array)	연결 리스트(Linked List)	탐색 트리(BST)	해싱(Hashing)
탐색(Search)	O(n)	O(n)	O(log n) (평균), O(n) (최악)	O(1) (평균), O(n) (최악)
삽입(Insertion)	O(n)	O(1)	O(log n) (평균), O(n) (최악)	O(1) (평균), O(n) (최악)
삭제(Deletion)	O(n)	O(1) (노드 알고 있을 때)	O(log n) (평균), O(n) (최악)	O(1) (평균), O(n) (최악)
임의 접근(Random Access)	O(1)	O(n)	O(n)	O(1)

7.2 순차 탐색(sequential search, 선형 탐색(linear)

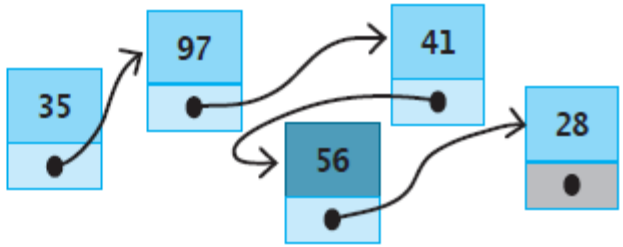
- 탐색을 위한 레코드는 리스트에 저장
- 탐색의 범위는 low에서 high까지 가정
- 리스트의 처음부터(low) 하나씩 순서대로 레코드를 탐색 키와 비교하여, 만약 같으면 그 레코드의 위치를 반환. 만약 high까지도 원하는 레코드가 나타나지 않으면 탐색 실패이므로 -1을 반환



순차 탐색 알고리즘



배열 구조의 테이블(56 탐색)



연결된 구조의 테이블(56 탐색)

- 테이블이 배열인 경우

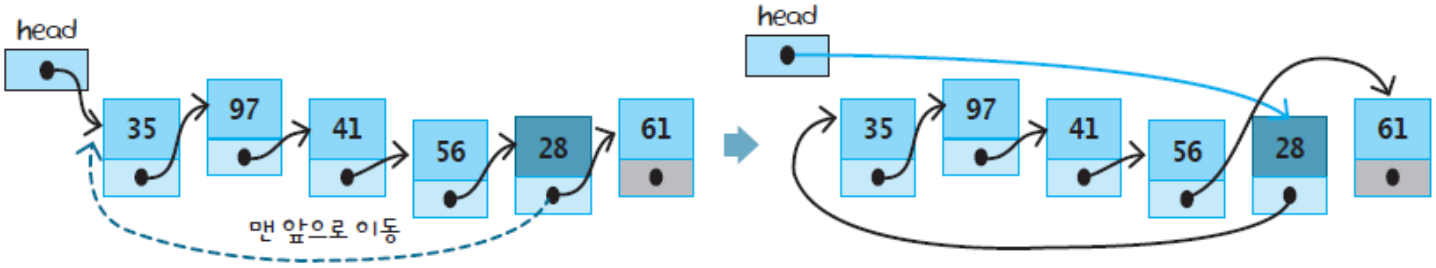
```
01:  def sequential_search(A, key, low, high) :
02:      for i in range(low, high+1) :          # i : low, low+1, ... high
03:          if A[i] == key :                   # 탐색 성공하면
04:              return i                       # 인덱스 반환
05:      return -1                              # 탐색에 실패하면 -1 반환
```

- 탐색 성능 : 테이블의 크기가 n이라면
  - 최선의 경우: 찾는 자료가 맨 앞에 있는 경우 – $O(1)$,
  - 최악또는 평균적 경우: 찾는 자료가 테이블의 맨 뒤에 있거나 리스트에 없는 키를 찾는 경우 – $O(n)$
  - 테이블이 정렬되어 있지 않다면 별다른 대안은 없음

(1) 순차 탐색을 개선하는 방법은? 맨 앞으로 보내기

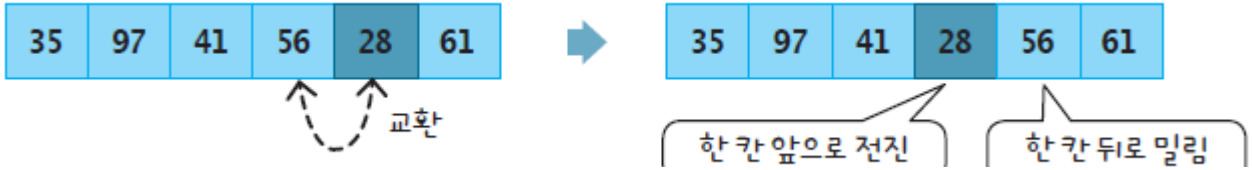
- **자기 구성(self-organizing) 순차 탐색**
  - 자주 사용되는 레코드를 앞쪽으로 옮기는 방법
  - 자기 구성 리스트
- **맨 앞으로 보내기(move to front)**
  - 탐색에 성공한 레코드를 **리스트의 맨 앞으로** 보내는 방법
  - 한번 탐색된 레코드가 바로 이어서 다시 탐색 될 가능성이 많은 응용
  - 자주 찾는 레코드의 위치 최적화: 반복 탐색 시 평균 탐색 성능이 개선
    - $O(1)$에 가까운 평균 시간 복잡도
  - 배열의 경우

  - 연결 리스트의 경우



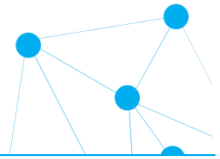
(2) 순차 탐색을 개선하는 방법은?교환하기

- **교환하기(transpose)**
  - 탐색된 레코드를 바로 앞의 레코드와 교환하는 전략
  - 자주 탐색 되는 레코드는 점진적으로 앞으로 이동하고, 그렇지 않은 레코드는 점진적으로 뒤로 밀리는 효과
  - '지금까지 더 많이 탐색된 레코드가 앞으로도 더 많이 탐색될 가능성이 큰' 응용에 만 적용
  - 자주 찾는 레코드의 위치 최적화: 반복 탐색 시 평균 탐색 성능이 개선
    - $O(1)$에 가까운 평균 시간 복잡도




```
01:  def sequential_search_transpose(A, key, low, high) :
02:      for i in range(low, high+1) :
03:          if A[i] == key :
04:              if i > low :            # 맨 처음 요소가 아니면
05:                  A[i], A[i-1] = A[i-1], A[i] # 교환하기(transpose)
06:                  i = i-1             # 한 칸 앞으로 왔음
07:              return i                # 탐색에 성공하면 키 값의 인덱스 반환
08:      return -1                   # 탐색에 실패하면 -1 반환
```

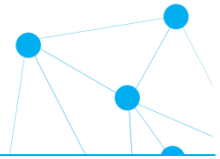
Quiz 2:



- 자기 구성 리스트에서 탐색에 성공한 레코드를 리스트의 맨 앞으로 보내는 작업을 구현하려 합니다. 이 리스트를 배열 구조와 연결된 구조로 각각 구현했을 때, 이 작업의 시간 복잡도는 각각 어떻게 될까요?

- **배열 구조** : 배열에서 탐색에 성공한 요소를 찾은 후, 그 요소를 맨 앞으로 이동시키려면, 해당 요소의 위치에서 시작하여 맨 앞까지 모든 요소들을 **한 칸씩 뒤로 밀어야 함**
  - **평균 탐색 시간 복잡도의 감소**:
    - 최악의 경우 리스트의 끝까지 탐색해야 하므로 $\boldsymbol{O(n)}$
    - **자주 탐색하는 요소에 대해 평균 시간 복잡도는 $\boldsymbol{O(1)}$에 가까워지도록 자기 리스트 구성**
  - **요소 이동 시간 복잡도**: 요소를 맨 앞으로 이동시키기 위해 최악의 경우 $n-1$개의 요소를 한 칸씩 이동하므로 $\boldsymbol{O(n)}$
    - **총 최악 시간 복잡도 : $\boldsymbol{O(n)}$ + $\boldsymbol{O(n)}$ = $\boldsymbol{O(n)}$**
    - **총 평균 탐색 시간 복잡도 : $\boldsymbol{O(1)}$**

Quiz 3:

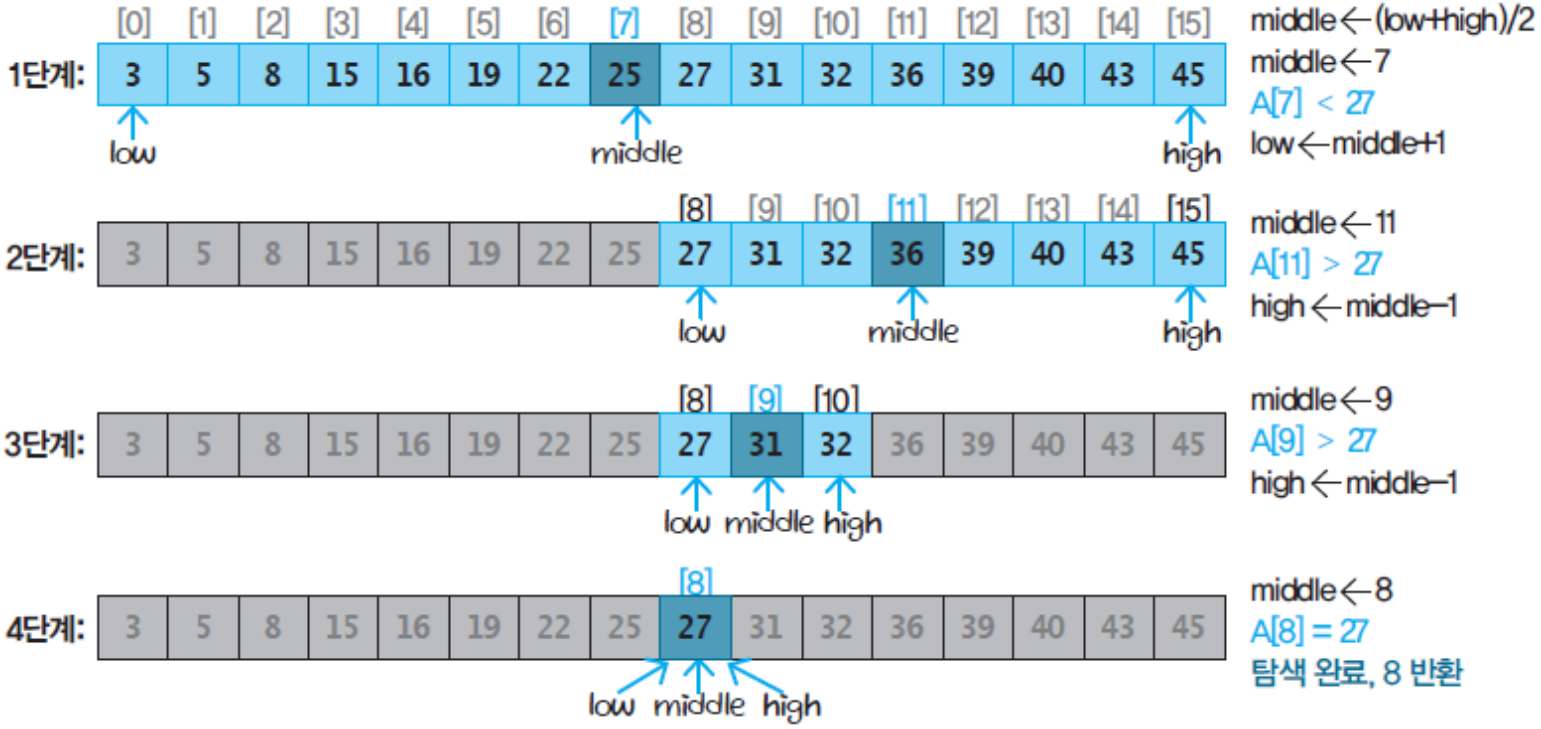


- 자기 구성 리스트에서 탐색에 성공한 레코드를 교환하기 전략을 사용한다면 배열 구조와 연결된 구조에서 시간 복잡도는 각각 어떻게 될까요?

- 배열 구조 :
  - **평균 탐색 시간 복잡도의 감소**:
    - 최악의 경우 배열에서 key 값을 찾기 위해서 선형 탐색을 해야 하므로 최악의 경우 $\boldsymbol{O(n)}$ 시간이 소요
    - **자주 탐색하는 요소에 대해 평균 시간 복잡도는 $\boldsymbol{O(1)}$에 가까워지도록 자기 구성 리스트**
  - **교환 시간 복잡도**: key 값이 맨 앞에 있는 경우에는 교환이 필요하지 않으므로 $\boldsymbol{O(1)}$. Key가 맨 앞이 아닌 경우, key 값의 인덱스i와 바로 앞의 인덱스 i-1을 교환하는 작업은 상수 시간 $\boldsymbol{O(1)}$ 소요
    - **총 최악 시간 복잡도 : $\boldsymbol{O(n)}$ + $\boldsymbol{O(1)}$ = $\boldsymbol{O(n)}$**
    - **총 평균 탐색 시간 복잡도 : $\boldsymbol{O(1)}$**

7.3 이진 탐색(binary search)

- 가정: 배열의 모든 레코드가 킷값을 기준으로 오름차순으로 정렬
- 매번 탐색의 범위를 절반으로 줄이면서, 타겟이 중간값보다 크거나 작은지에 따라 탐색의 범위를 좁혀가는 방식
- 사례: 사전에서 단어를 찾는 과정. 사전을 펼쳐 찾고자 하는 단어가 현재 페이지보다 앞에 있는지 뒤에 있는지를 확인하고 단어가 있는 부분만을 다시 탐색해 탐색 범위를 줄임

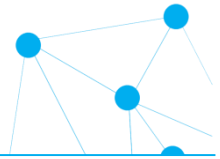


이진 탐색 알고리즘(순환 구조 & 반복 구조)

```
01: def binary_search(A, key, low, high) :
02:     if (low <= high) :              # 항목들이 남아 있으면(종료 조건)
03:         middle = (low + high)//2    # middle 계산
04:         if key == A[middle] :       # 탐색 성공
05:             return middle           # 중앙 레코드의 인덱스 반환
06:         elif (key<A[middle]) :      # 왼쪽 부분리스트 탐색 -> 순환호출
07:             return binary_search(A, key, low, middle - 1)
08:         else :                      # 오른쪽 부분리스트 탐색 -> 순환호출
09:             return binary_search(A, key, middle + 1, high)
10:     return -1                       # 탐색 실패 -1 반환
```

```
01: def binary_search_iter(A, key, low, high) :
02:     while (low <= high) :           # 항목들이 남아 있으면(종료 조건)
03:         middle = (low + high)//2    # middle 계산
04:         if key == A[middle]:        # 탐색 성공
05:             return middle
06:         elif (key > A[middle]):     # key가 middle의 값보다 크면
07:             low = middle + 1        # middle+1 ~ high 사이 검색
08:         else:                       # key가 middle의 값보다 작으면
09:             high = middle - 1       # low ~ middle-1 사이 검색
10:     return -1                       # 탐색 실패 -1 반환
```

이진 탐색의 특징



- 탐색 범위: 테이블의 크기 $n = 2^k$
  - 순환 호출을 한 번할 때마다 탐색의 범위는 절반으로 줄어든다.

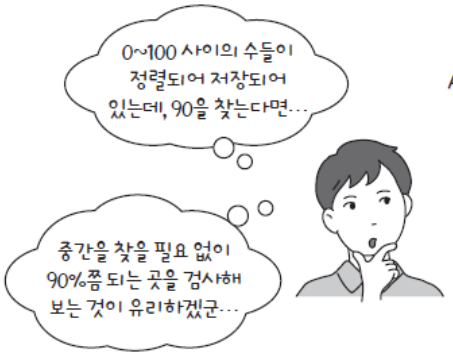
$$2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \cdots \rightarrow 2^1 \rightarrow 2^0$$

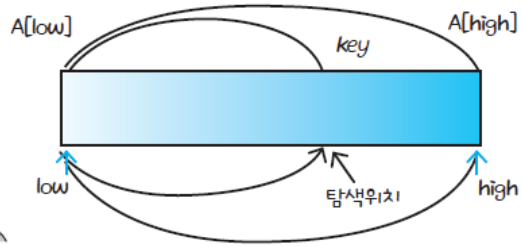
    - 반복되는 횟수 : $k = log_2 n$
- 효율적인 탐색 방법:
  - 최선 시간 복잡도 : $O(1)$
    - 키 값이 처음 중간 위치에서 바로 발견되는 경우
  - 평균 및 최악 시간 복잡도 : $O(log_2 n)$
    - 키 값이 배열에 없거나, 탐색 범위를 끝까지 좁혀야 할 경우
    - 매번 범위를 절반으로 줄이기 때문에 탐색이 반복되는 횟수는 k
- 반드시 배열이 정렬되어 있어야 사용할 수 있음
- 탐색은 효율적이지만 삽입/삭제는 효율적이지 않음
  - 테이블이 한번 만들어지면 이후로 변경되지 않고 탐색 연산만 처리한다면 이진 탐색이 최고의 선택
  - **삽입과 삭제가 빈번하다면?** 이진탐색트리 등 다른 방법을 고려

보간 탐색(interpolation search)

- 키 값의 예상 위치를 계산하여 해당 위치에서 이진 탐색 방식으로 탐색하는 방법
- 탐색 위치를 탐색 범위 가장자리(low, high)에 있는 레코드의 킷값과 탐색 키의 비율을 고려하여 계산
- 최선의 경우: 찾으려는 값이 첫 번째 추정 위치에 정확히 위치한 경우 – $O(1)$
- 평균 및 최악의 경우 :
  - 배열이 균등하게 분포된 경우 예상 위치가 유효하게 작용하므로 이진탐색보다 효율적: $O(log_2(log_2 n))$
  - 배열이 균등하게 분포되어 있지 않은 경우 탐색의 범위를 좁히는 데 한계 : $O(n)$

$$\text{탐색위치} = low + (high - low) \times \frac{key - A[low]}{A[high] - A[low]}$$





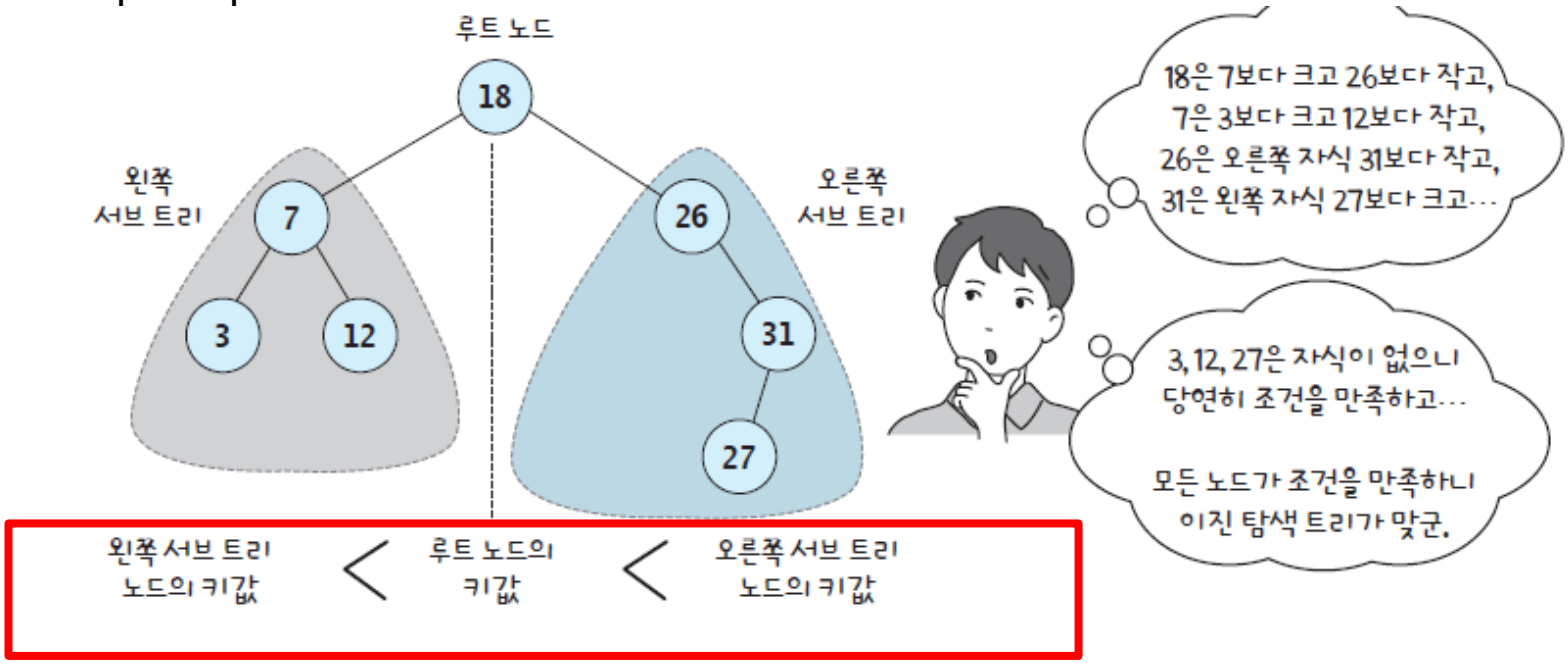

```
middle = int(low + (high-low) * (key-A[low].key) / (A[high].key-A[low].key))
```

이진탐색 대신 이진탐색트리를 사용하는 이유

- 모두 정렬된 배열에서는 빠른 탐색을 제공
- 동적 데이터 삽입과 삭제의 효율성
  - 이진 탐색: 정렬된 배열에서만 사용
    - 삽입이나 삭제가 필요할 때마다 배열을 재정렬이나 이동 비용이 발생 : $O(n)$
  - 이진탐색트리: 동적 데이터에 대해 효율적
    - 삽입과 삭제 연산 : $O(log_2 n)$
- 자동 정렬
  - 이진 탐색: 정렬된 배열에서만 사용
    - 삽입연산은 배열을 재정렬해야 하므로 실시간 변경되는 데이터에 부적합
  - 이진탐색트리:
    - 새 데이터를 삽입할 때마다 자동으로 정렬 유지하므로 전체 트리를 다시 정렬할 필요가 없음
- 비 균등하게 분포된 데이터 처리
  - 이진 탐색:
    - 균등하게 분포된 정렬된 배열에서 빠른 검색
    - 비균등 분포인 경우 탐색 효율이 저하
  - 이진탐색트리: 데이터의 분포에 크게 영향을 받지 않음

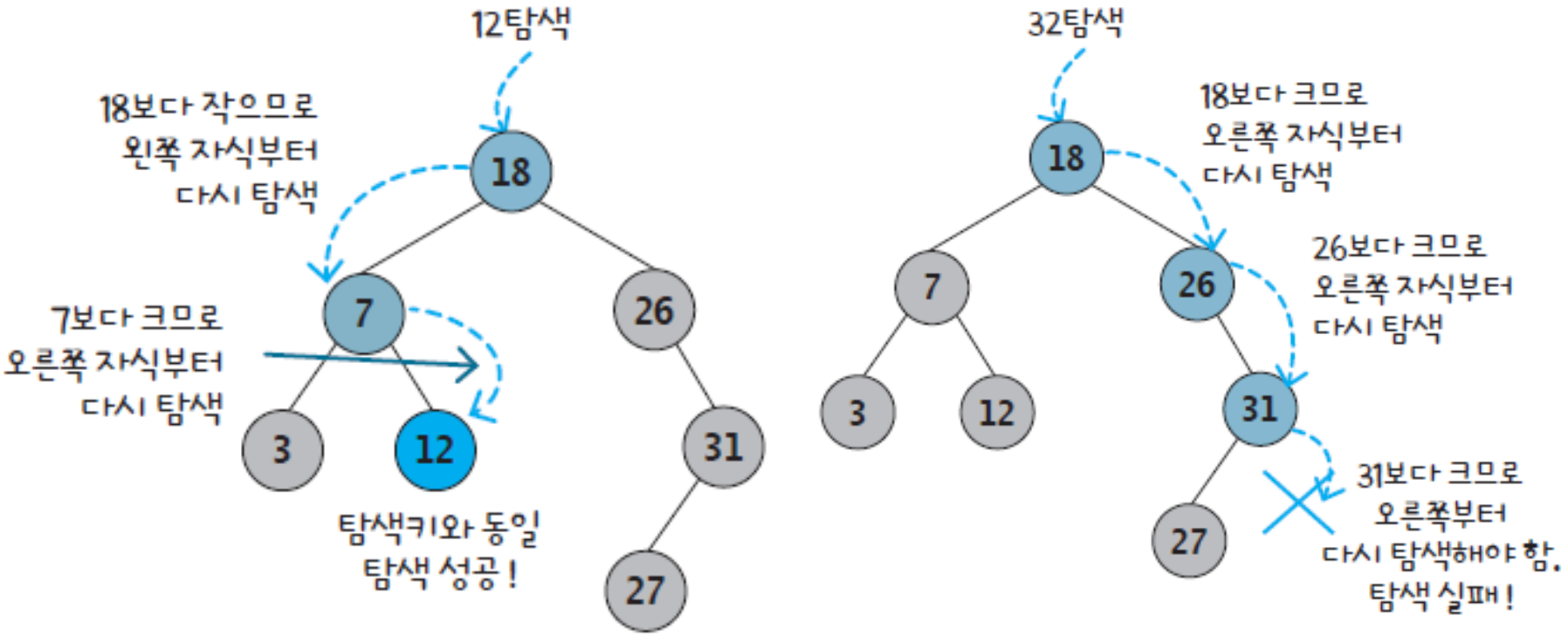
7.4 이진 탐색 트리(BST:Binary Search Tree)

- '이진 탐색'을 위한 '트리'
  - '탐색'의 성능은 유지하면서 '삽입'과 '삭제'도 효율적으로 처리
- 이진 탐색 트리의 조건
  - 이진 트리
  - 킷값의 중복을 허용하지 않음
  - 모든 노드가 '왼쪽 자식 노드는 나보다 작고, 오른쪽 자식 노드는 나보다 크다 '

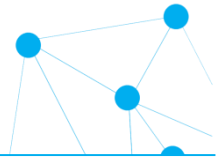


이진 탐색 트리의 탐색 연산

- 항상 루트 노드에서 시작해서 아래로 내려감
  - 예: 12의 탐색(성공)과 32의 탐색(실패)



탐색 알고리즘: 키를 이용한 탐색



- 이진 탐색 트리를 위한 노드 클래스

```
01:  class BSTNode:                              # 이진 탐색 트리를 위한 노드 클래스
02:      def __init__ (self, key, value):# 생성자: 키와 값을 받음
03:          self.key = key       # 키(key)
04:          self.value = value # 값(value): 키를 제외한 데이터 부분
05:          self.left = None     # 왼쪽 자식에 대한 링크
06:          self.right = None    # 오른쪽 자식에 대한 링크
```

- 이진 탐색 트리의 탐색 연산(순환 구조)

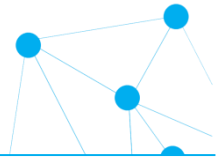
```
01: def search_bst(n, key) :
02:     if n == None :
03:         return None
04:     elif key == n.key:
05:         return n
06:     elif key < n.key:
07:         return search_bst(n.left, key)
08:     else:
09:         return search_bst(n.right, key)
```

← n을 루트로 갖는 이진 탐색 트리에서 키값이 key인 노드를 찾는 순환 함수

← n이 None이면 공백 트리이므로 None 반환
n의 key가 탐색키와 같으면 n 반환

← 탐색키가 n의 key보다 작으면 왼쪽 서브트리 탐색
아니면 오른쪽 서브트리 탐색

탐색 알고리즘: 값을 이용한 탐색



- 트리의 모든 노드를 검사해야 함
- 모든 노드를 방문하는 방법에는 제한이 없음
  - 전위 순회, 후위 순회, 중위 순회, 레벨 순회
- 최악의 경우 트리의 모든 노드를 검사해야 하므로 탐색의 성능은 키를 이용한 탐색에 비해 떨어짐
  - 전위 순회 탐색

```
01: def search_value_bst(n, value) :
02:     if n == None : return None
03:     elif value == n.value:
04:         return n
05:     res = search_value_bst(n.left, value)
06:     if res is not None :
07:         return res
08:     else :
09:         return search_value_bst(n.right, value)
```

← n을 루트로 갖는 이진 탐색 트리에서 키(key)가 아닌 값(value)으로 노드를 찾는 함수. 전위순회 이용.

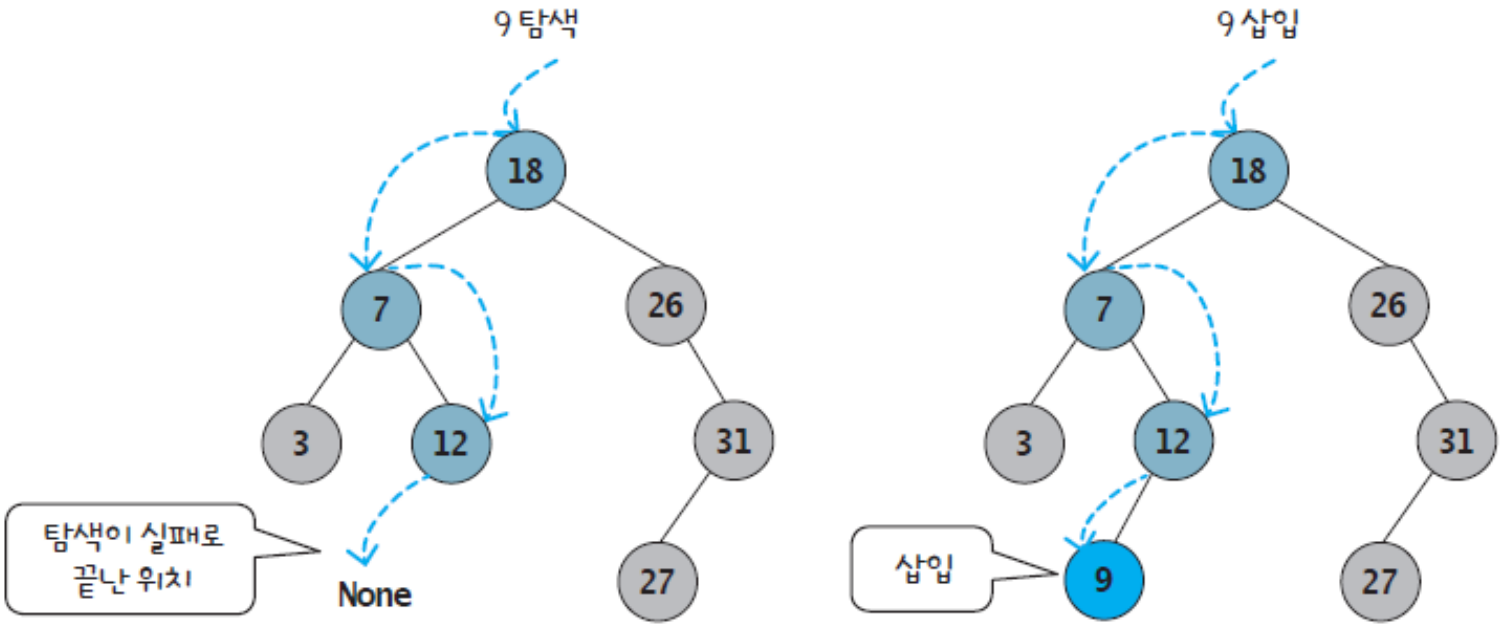
← 루트 탐색. 공백 트리이거나 탐색 성공이면 결과를 반환하고 종료.

← 왼쪽 서브트리 탐색. 왼쪽 서브트리에서 탐색 성공이면 결과를 바로 반환.

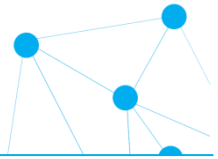
← 오른쪽 서브트리 탐색후 결과 반환.

이진 탐색 트리의 삽입 연산

1. 먼저, 삽입할 노드의 키를 이용한 탐색 과정을 수행
   - 탐색에 실패한 위치에 새로운 노드를 삽입
   - 탐색이 성공하면 중복된 킷값을 가진 노드가 이미 있는 상태이므로 노드를 삽입하지 않는다.
2. 노드를 삽입한 후에도 이진 탐색 트리의 특성이 유지되어야 함



삽입 알고리즘



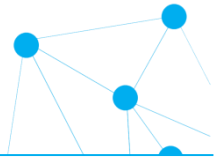
```
01:  def insert_bst(root, node):
02:      if root == None :              # 공백 노드에 도달하면, 이 위치에 삽입
03:          return node                # node를 반환(이 노드가 현재 root 위치에 감)
04:
05:      if node.key == root.key : # 동일한 키는 허용하지 않음
06:          return root                # root를 반환(root는 변화 없음)
07:
08:      # root의 서브 트리에 node 삽입
09:      if node.key < root.key :
10:          root.left = insert_bst(root.left, node)
11:
12:      else :
13:          root.right= insert_bst(root.right, node)
14:
15:      return root      # root를 반환(root는 변화 없음)
```

왼쪽서브 트리에 넣어야 하는 경우, 왼쪽자식을 루트로 삽입 연산을 순환 호출하고, 왼쪽자식 갱신

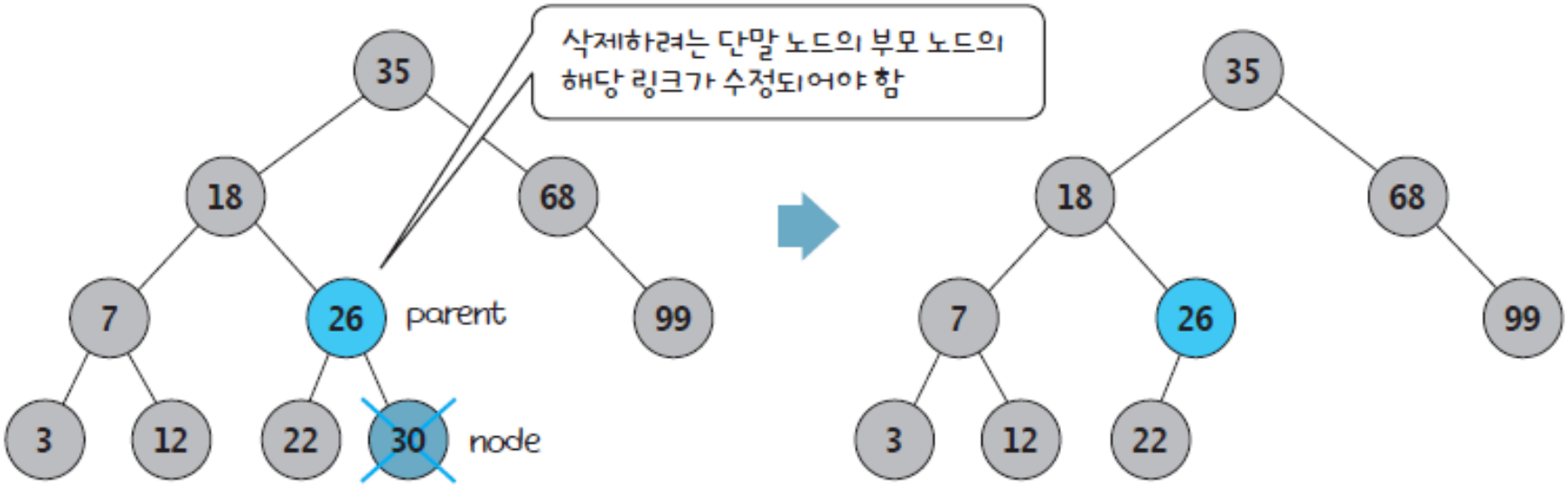
오른쪽서브 트리에 넣어야하는 경우, 오른쪽자식을 루트로 삽입 연산을 순환 호출하고, 오른쪽자식 갱신

- 순환 구조
- 항상 노드(root)를 반환하도록 구현됨

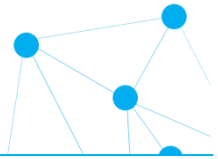
이진 탐색 트리의 삭제 연산



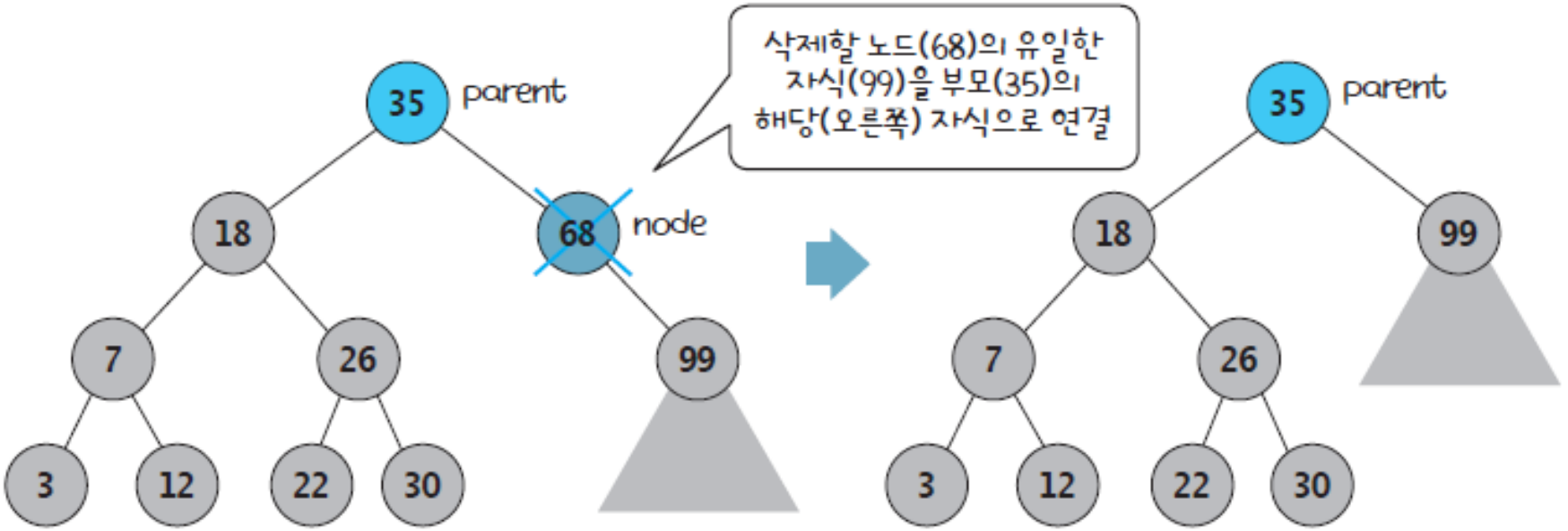
- 노드를 삭제한 후에도 이진 탐색 트리의 특성이 유지되어야 함
- 삭제할 노드의 자식 수에 따라 3가지 경우로 구분
- Case 1: 단말 노드의 삭제
  - 삭제 노드의 부모 노드의 해당 링크를 None으로 변경



이진 탐색 트리의 삭제 연산

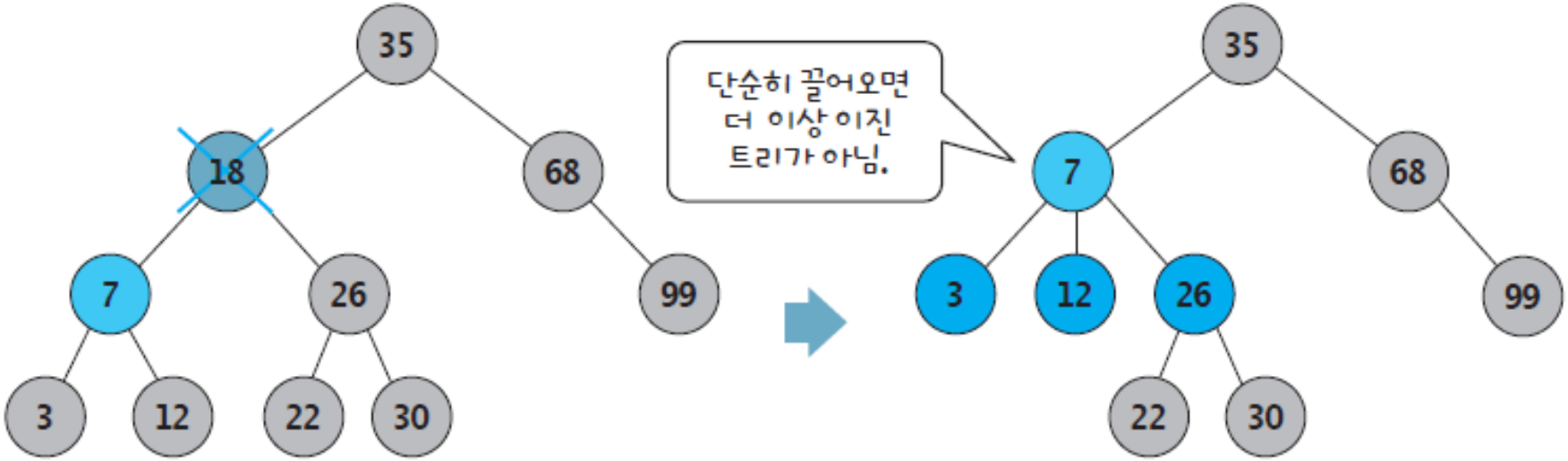


- Case 2: 자식이 하나인 노드의 삭제
  - 삭제할 노드가 하나의 자식을 갖는다면 그 자식을 자신을 대신해 부모 노드에 연결해준다.

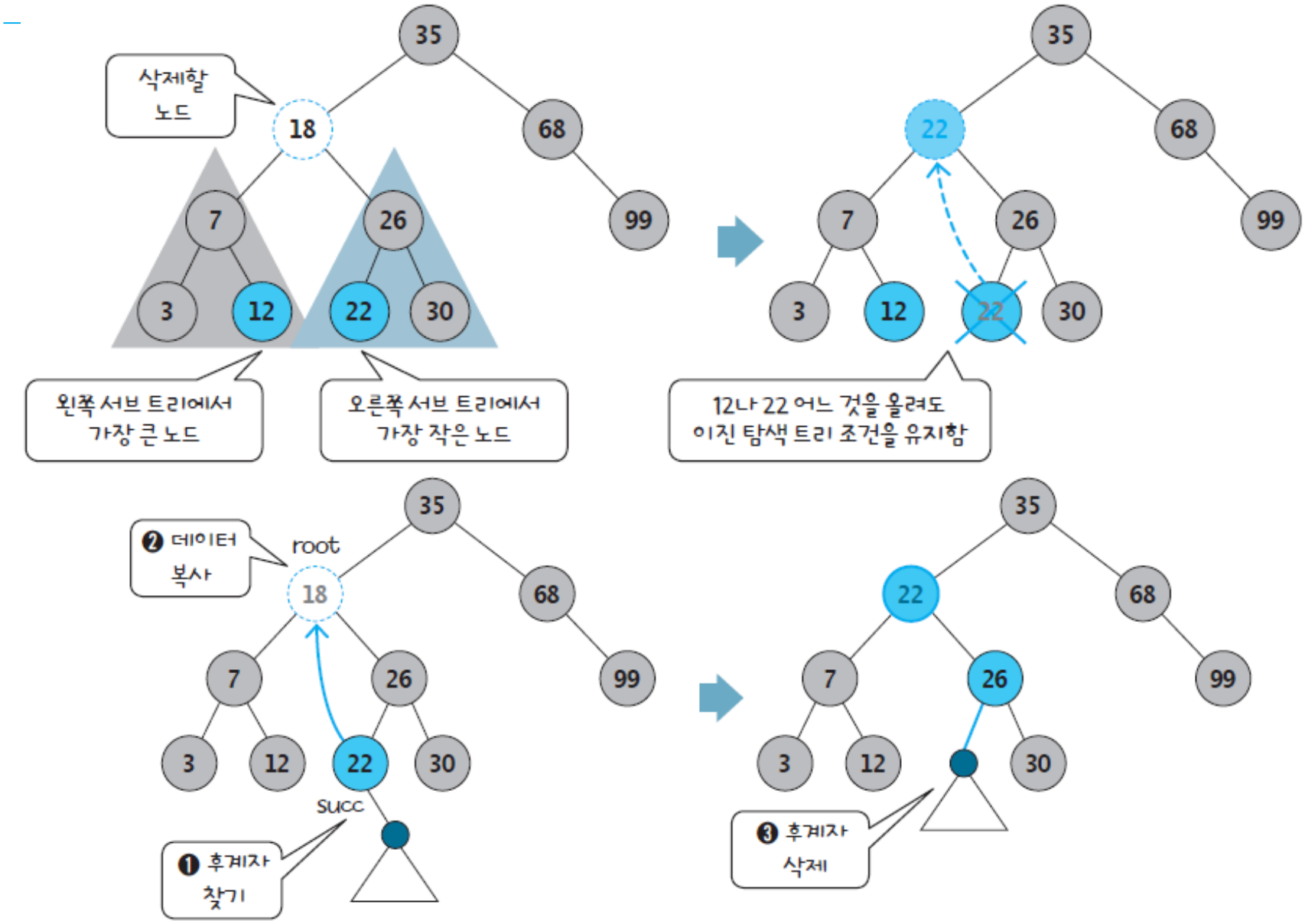


이진 탐색 트리의 삭제 연산

- Case 3: 2개의 자식을 모두 갖는 노드의 삭제
  1. **후계자 선택**
     - 삭제할 노드의 왼쪽 서브 트리에서 가장 큰 노드
     - 삭제할 노드의 오른쪽 서브 트리에서 가장 작은 노드
     - 후계자 후보 노드는 자식이 1 개 이하
     - 중위 순회했을 때, 삭제할 노드의 바로 앞과 뒤에 있는 노드
  2. **후계자를 삭제할 노드에 복사**
  3. **후계자를 삭제함**



이진 탐색 트리의 삭제 연산



삭제 알고리즘

```
01:  def delete_bst (root, key) :
02:      if root == None :  # 공백 트리
03:          return root
04:
05:      if key < root.key :
06:          root.left = delete_bst(root.left, key)
07:      elif key > root.key :
08:          root.right= delete_bst(root.right, key)
09:
10:      # key가 루트의 키와 같으면 root를 삭제
11:      else :
12:          # case1(단말 노드) 또는 case2(오른쪽 자식만 있는 경우)
13:          if root.left== None :
14:              return root.right
15:
16:          # case2(왼쪽 자식만 있는 경우)
17:          if root.right== None :
18:              return root.left
19:
20:          # case3(두 자식이 모두 있는 경우)
21:          succ = root.right
22:          while succ.left != None:
23:              succ = succ.left
24:
25:          root.key = succ.key
26:          root.value = succ.value
27:          root.right = delete_bst(root.right, succ.key)
28:
29:      return root
```

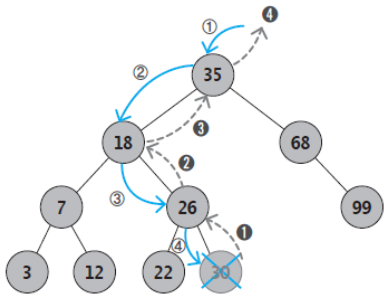
key가 루트보다 작거나 크면, 해당 자식이 루트인 서브트리에서 삭제를 계속 진행함(순환호출 이용) 이때, 자식이 변경될 수도 있으므로 반환된 값으로 자식을 갱신해야 함

삭제할 노드 위치에 오른쪽 자식을 끌어올림. 즉, 오른쪽 자식을 반환

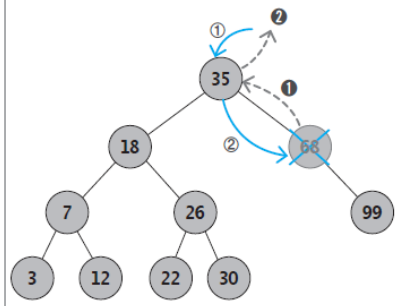
삭제할 노드 위치에 왼쪽 자식을 끌어올림. 즉, 왼쪽 자식을 반환

① 후계자를 찾고(오른쪽 서브트리 최소노드)
② 후계자의 데이터(key와 value)를 복사하고

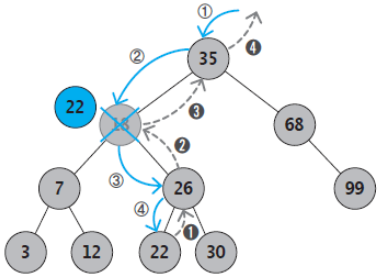
③ 마지막으로, 후계자 삭제(오른쪽 서브 트리에서 후계자 킷값을 가진 노드를 순환 호출로 삭제)



① delete_bst(35, 30) — 6행
② delete_bst(18, 30) — 8행
③ delete_bst(26, 30) — 8행
④ delete_bst(30, 30) → ❶ return None (14행)
❹ return 35 (29행)
❸ return 18 (29행)
❷ return 26 (29행)



① delete_bst(26, 68) — 8행
② delete_bst(68, 30) → ❶ return 99 (14행)
❷ return 35 (29행)



① delete_bst(35, 18) — 6행
② delete_bst(18, 18) — 27행
③ delete_bst(26, 22) — 6행
④ delete_bst(22, 22) → ❶ return None (14행)
❹ return 35 (29행)
❸ return 22 (29행)
❷ return 26 (29행)

테스트 프로그램

```
01:  def print_node(msg, n) :                  # 노드 출력 함수
02:      print(msg, n if n != None else "탐색실패")
03:
04:  def print_tree(msg, r) :
05:      print(msg, end='')
06:      preorder(r)
07:      print()
08:
09:  data = [(6, "여섯"), (8, "여덟"), (2,"둘"), (4,"넷"),  (7,"일곱"),
     (5,"다섯"), (1,"하나"), (9,"아홉"), (3,"셋"), (0,"영")]
10:
11:  root = None                               # 루트 노드 초기화
12:  for i in range(0, len(data)):             # 노드 순서대로 추가하기
13:      root = insert_bst(root, BSTNode(data[i][0], data[i][1]))
14:
15:  print_tree("최초: ", root)                # 최초의 트리 출력
16:
17:  n = search_bst(root, 3);          print_node("srch 3: ", n)
18:  n = search_bst(root, 8);          print_node("srch 8: ", n)
19:  n = search_bst(root, 0);          print_node("srch 0: ", n)
20:  n = search_bst(root, 10);         print_node("srch10: ", n)
21:  n = search_value_bst(root,"둘");  print_node("srch둘: ", n)
22:  n = search_value_bst(root,"열");  print_node("srch열: ", n)
23:
24:  root = delete_bst(root, 7);       print_tree("del  7: ", root)
25:  root = delete_bst(root, 8);       print_tree("del  8: ", root)
26:  root = delete_bst(root, 2);       print_tree("del  2: ", root)
27:  root = delete_bst(root, 6);       print_tree("del  6: ", root)
```

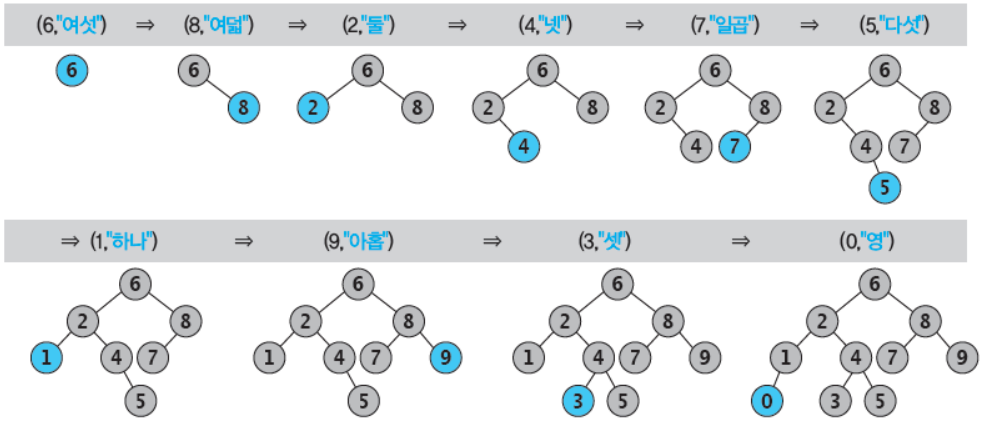
전위 순회를 이용한 트리 출력 함수

key

value

key를 이용한 탐색

값(value)을 이용한 탐색

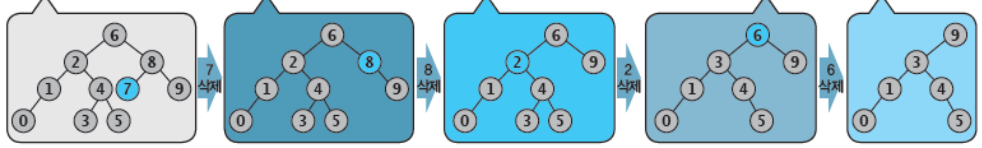


실행 결과

```
최초: {(6:여섯){(2:둘){(1:하나){(0:영)}}{(4:넷){(3:셋)}{(5:다섯)}}}{ {(8:여덟){(7:일곱)}{(9:아홉)}}}
srch 3: (3:셋)
srch 8: (8:여덟)
srch 0: (0:영)
srch10: 탐색실패
srch둘: (2:둘)
srch열: 탐색실패
del  7: {(6:여섯){(2:둘){(1:하나){(0:영)}}{(4:넷){(3:셋)}{(5:다섯)}}}{(8:여덟) (9:아홉)}}
del  8: {(6:여섯){(2:둘){(1:하나){(0:영)}}{(4:넷){(3:셋)}{(5:다섯)}}}{(9:아홉)}}
del  2: {(6:여섯){(3:셋){(1:하나){(0:영)}}{(4:넷){(5:다섯)}}}{(9:아홉)}}
del  6: {(9:아홉){(3:셋){(1:하나){(0:영)}}{(4:넷){(5:다섯)}}}}
```

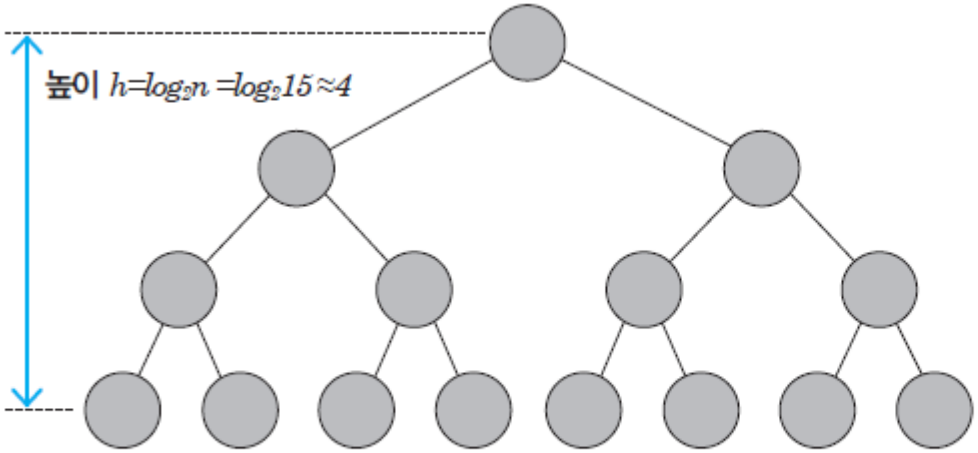
키를 이용한 탐색

값을 이용한 탐색

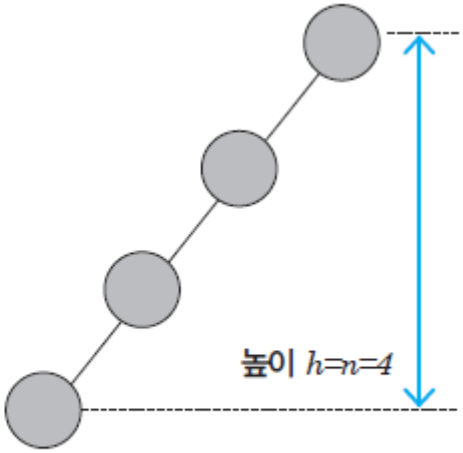


이진 탐색 트리의 성능

- 최선의 경우 삽입, 삭제, 탐색 성능 : $O(log_2 n)$
- 최악의 경우 삽입, 삭제, 탐색의 성능 : $O(n)$



(a) 균형 잡힌 이진 트리(포화 이진 트리)



(b) 경사 이진 트리

- 이진 탐색 트리의 효율을 높이기 위해 트리가 좌우 균형 유지
  - 이진 탐색 트리의 다양한 균형 기법들
    - AVL 트리, 2-3트리, 2-3-4 트리, B 트리, Red-Black 트리 등
  - 트리의 구조가 변경되는 삽입과 삭제 연산에서 불균형 상태가 발생하면 스스로 노드들을 재배치하여 균형 상태로 만드는 방법 사용

이진탐색트리 기반 탐색 알고리즘의 효율성

작업 유형	최선의 경우 시간 복잡도	균형 트리 시간 복잡도 $O(\log n)$	편향 트리 시간 복잡도 $O(n)$
탐색	$O(1)$	평균 및 최악: $O(\log n)$	모든 노드를 탐색해야 하므로 $O(n)$
삽입	$O(1)$	평균 및 최악: $O(\log n)$	모든 노드를 탐색해야 하므로 $O(n)$
삭제	$O(1)$	평균 및 최악: $O(\log n)$	삭제 대상 노드까지 순차적으로 탐색: $O(n)$