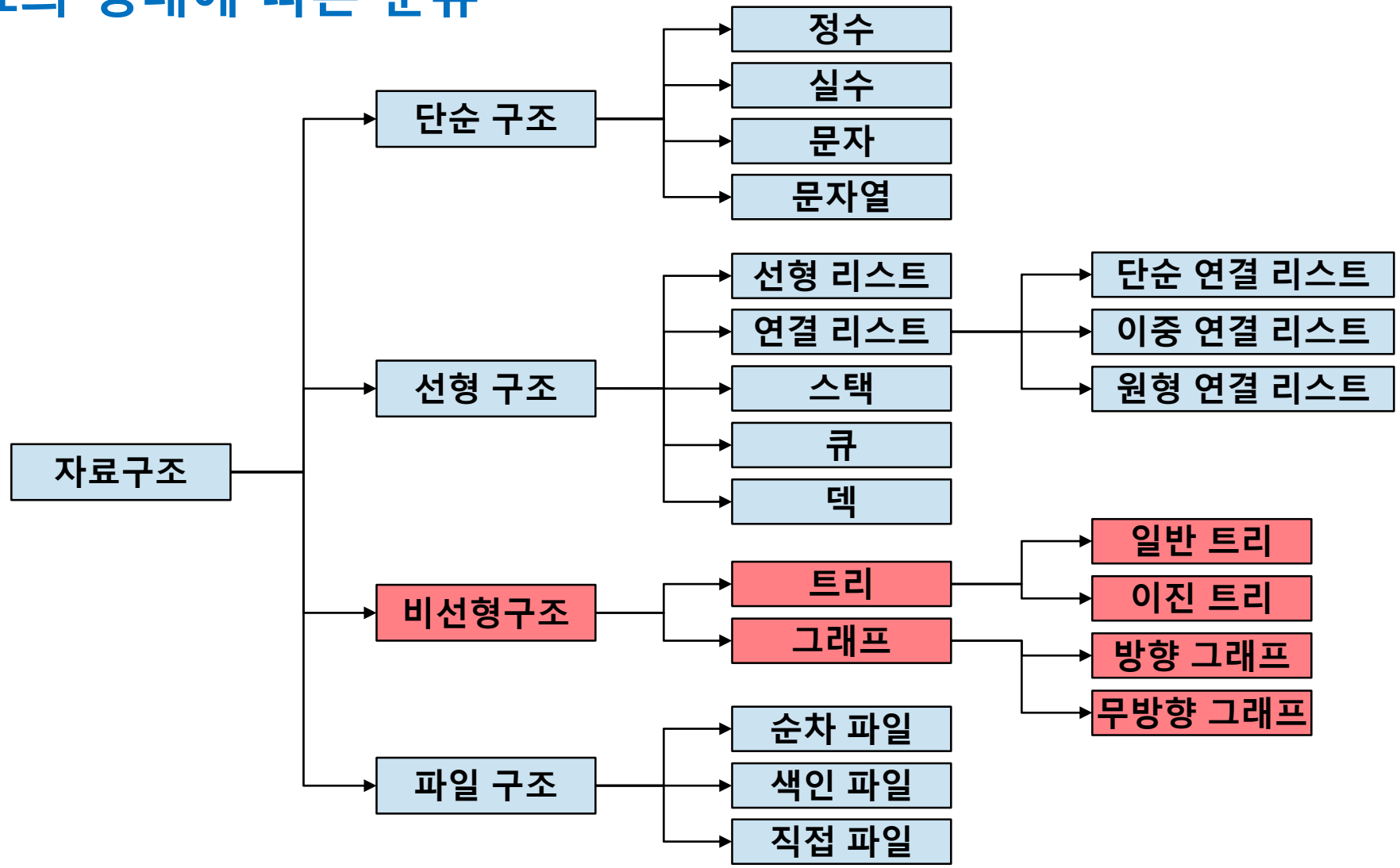
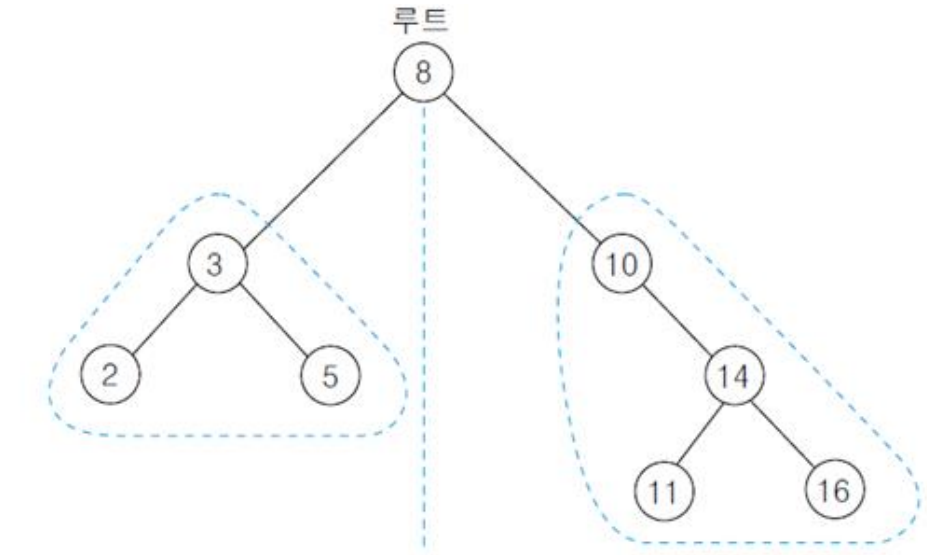
Tree

- **자료의 형태에 따른 분류**



Tree

- **이진 탐색 트리(binary search tree)**
  - 이진 트리에 탐색을 위한 조건을 추가하여 정의한 자료구조
  - 이진 탐색 트리의 정의
    (1) 모든 원소는 서로 다른 **유일한 키**를 갖는다.
    (2) **왼쪽** 서브트리에 있는 원소의 키들은 **그 루트의 키보다 작다**.
    (3) **오른쪽** 서브트리에 있는 원소의 키들은 **그 루트의 키보다 크다**.
    (4) 왼쪽 서브트리와 오른쪽 서브트리도 이진 탐색 트리다.



Tree

- **이진 탐색 트리의 탐색 연산**
  - 루트에서 시작한다.
  - 탐색할 key값 x를 루트 노드의 key값과 비교한다.
    - **(key값 x = 루트노드의 key값)인 경우** :
      ☞ 원하는 원소를 찾았으므로 탐색연산 성공
    - **(key값 x < 루트노드의 key값)인 경우** :
      ☞ 루트노드의 **왼쪽** 서브트리에 대해서 탐색연산 수행
    - **(key값 x > 루트노드의 key값)인 경우** :
      ☞ 루트노드의 **오른쪽** 서브트리에 대해서 탐색연산 수행
  - 서브트리에 대해서 순환적으로 탐색 연산을 반복한다

Tree

- **탐색 연산 알고리즘**

알고리즘 8-4 이진 탐색 트리에서의 탐색 연산 알고리즘

```
searchBST(bsT, x)
    p ← bsT;
    if (p=null) then
        return null;
    if (x = p.key) then
        return p;
    if (x < p.key) then
        return searchBST(p.left, x);
    else return searchBST(p.right, x);
end searchBST( )
```

Tree

- **이진 탐색 트리의 삽입 연산**

  1) 먼저 탐색 연산을 수행
     - 삽입할 원소와 같은 원소가 트리에 있으면 삽입할 수 없으므로, 같은 원소가 트리에 있는지 탐색하여 확인한다.
     - 탐색에서 탐색 실패가 결정되는 위치가 삽입 위치가 된다.

  2) 탐색 실패한 위치에 원소를 삽입한다.

Tree

- **이진 탐색 트리 삽입 알고리즘**

알고리즘 8-5 이진 탐색 트리에서의 삽입 연산 알고리즘

```
insertBST(bsT, x)
    p ← bsT;
    while (p≠null) do {
        if (x = p.key) then return;
        q ← p;
        if (x < p.key) then p ← p.left;
        else p ← p.right;
    }

    new ← getNode();
    new.key ← x;
    new.left ← null;
    new.right ← null;

    if (bsT = null) then bsT←new;
    else if (x < q.key) then q.left ← new;
    else q.right ← new;
    return;
end insertBST()
```

❶ 삽입할 노드 탐색

❷ 삽입할 노드 생성

❸ 노드 연결

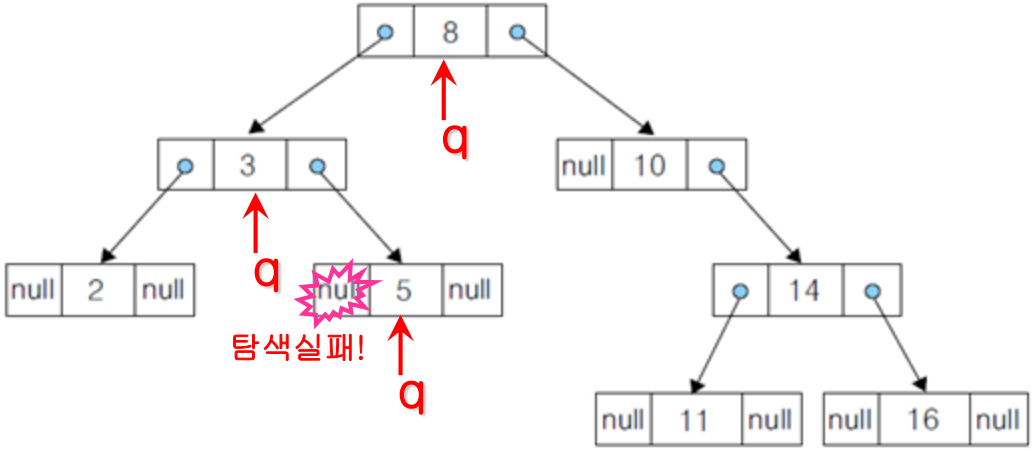
삽입할 자리 탐색

삽입할 노드 만들기

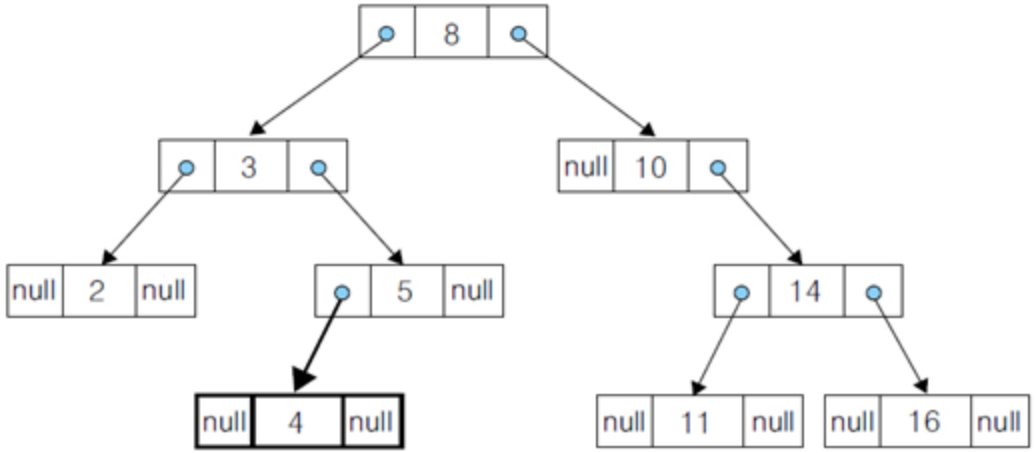
탐색한 자리에 노드연결

Tree

– 단순 연결 리스트로 표현한 이진트리에서의 원소 4 삽입하기



(1) 이진 탐색 트리의 연결 표현 : 삽입 전



2) 이진 탐색 트리의 연결 표현 : 삽입 후

Tree

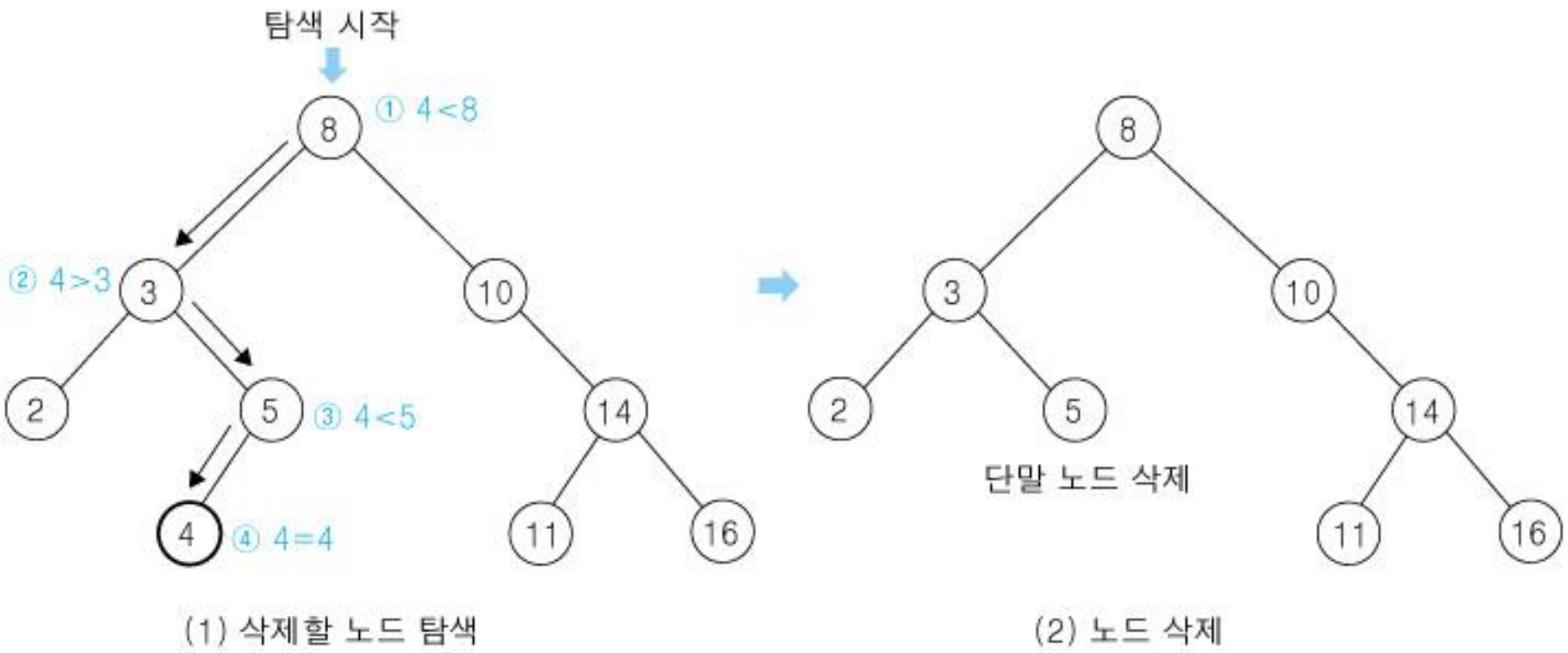
- **이진 탐색 트리의 삭제 연산**

  1) 먼저 탐색 연산을 수행
     - 삭제할 노드의 위치를 알아야 하므로 트리를 탐색한다.

  2) 탐색하여 찾은 노드를 삭제한다.
     - 노드의 삭제 후에도 이진 탐색 트리를 유지해야 하므로 삭제 노드의 경우에 대한 후속 처리(이진 탐색 트리의 재구성 작업)가 필요하다.
       - 삭제할 노드의 경우
         - » 삭제할 노드가 단말노드인 경우 : 차수가 0인 경우
         - » 삭제할 노드가 하나의 자식노드를 가진 경우 : 차수가 1인 경우
         - » 삭제할 노드가 두개의 자식노드를 가진 경우 : 차수가 2인 경우

Tree

- 단말 노드의 삭제 연산
  - 노드 4를 삭제하는 경우

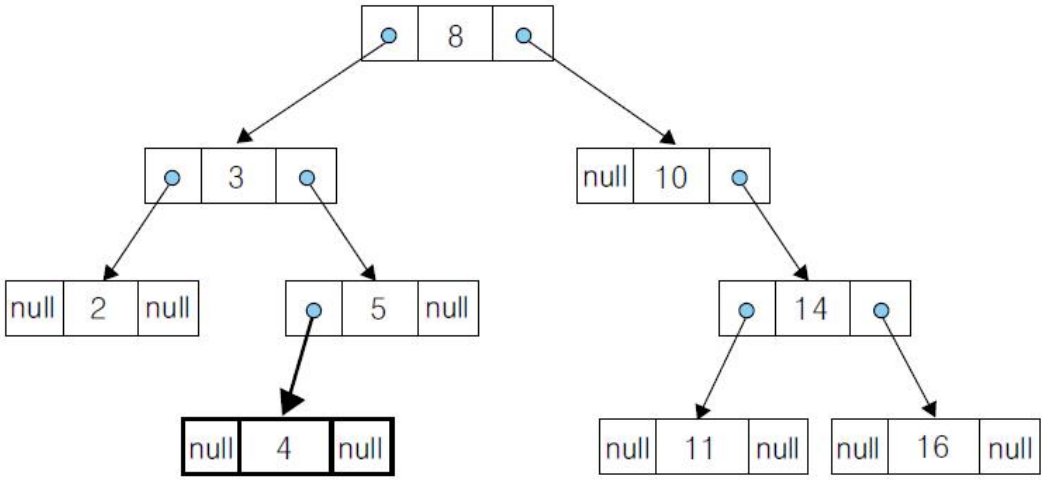


(1) 삭제할 노드 탐색

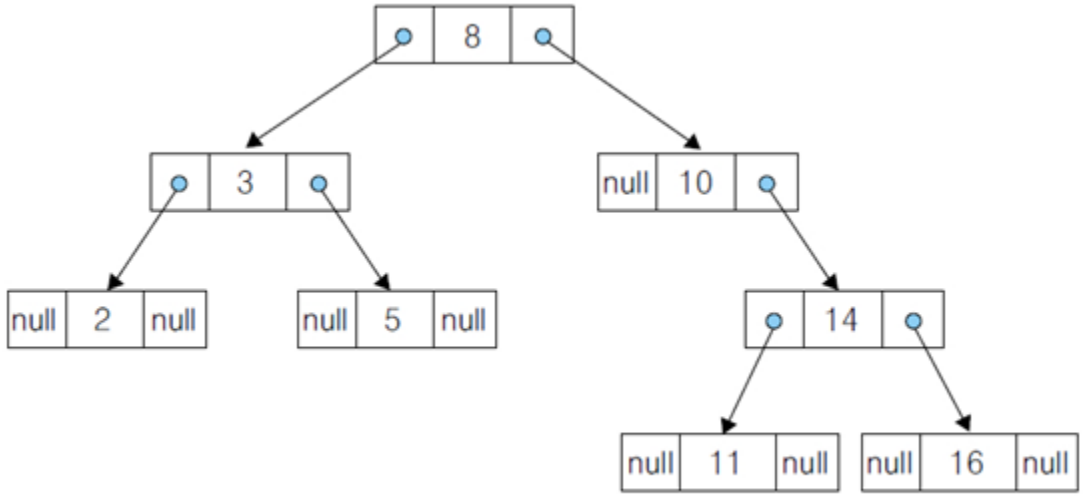
(2) 노드 삭제

Tree

- 노드 4를 삭제하는 경우에 대한 단순 연결 리스트 표현
  - 노드를 삭제하고, 삭제한 노드의 부모 노드의 링크 필드에 null 설정



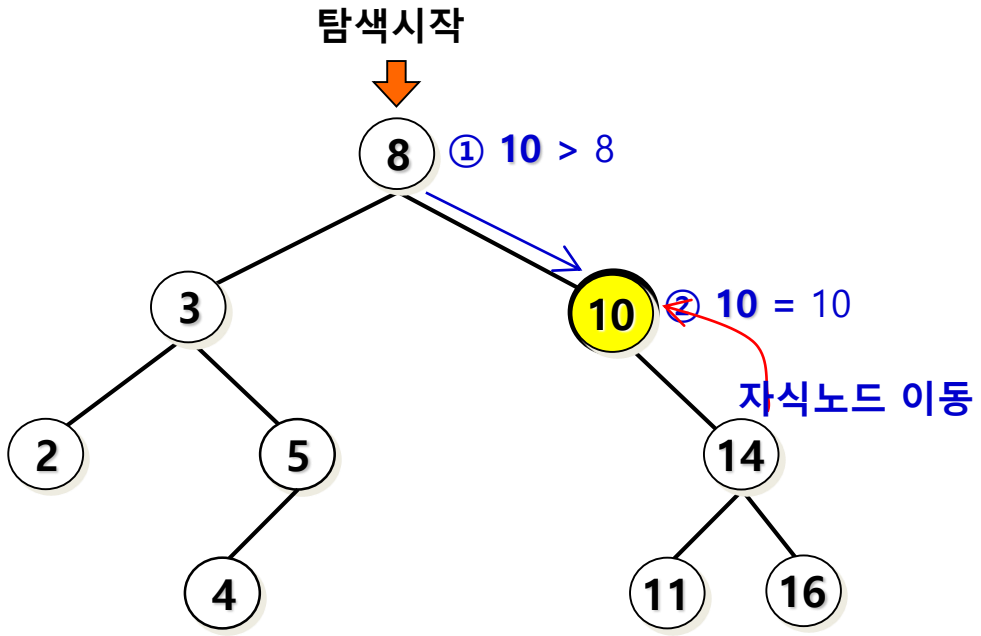
(1) 이진 탐색 트리의 연결 표현 : 삭제 전



(2) 이진 탐색 트리의 연결 표현 : 삭제 후

Tree

- 자식 노드가 하나인 노드, 즉 차수가 1인 노드의 삭제 연산
  - 노드를 삭제하면, 자식 노드는 트리에서 연결이 끊어져서 고아가 된다.
  - **후속 처리 : 이진 탐색 트리의 재구성**
    - **삭제한 부모노드의 자리를 자식노드에게 물려준다.**
  - 예) 노드 10을 삭제하는 경우



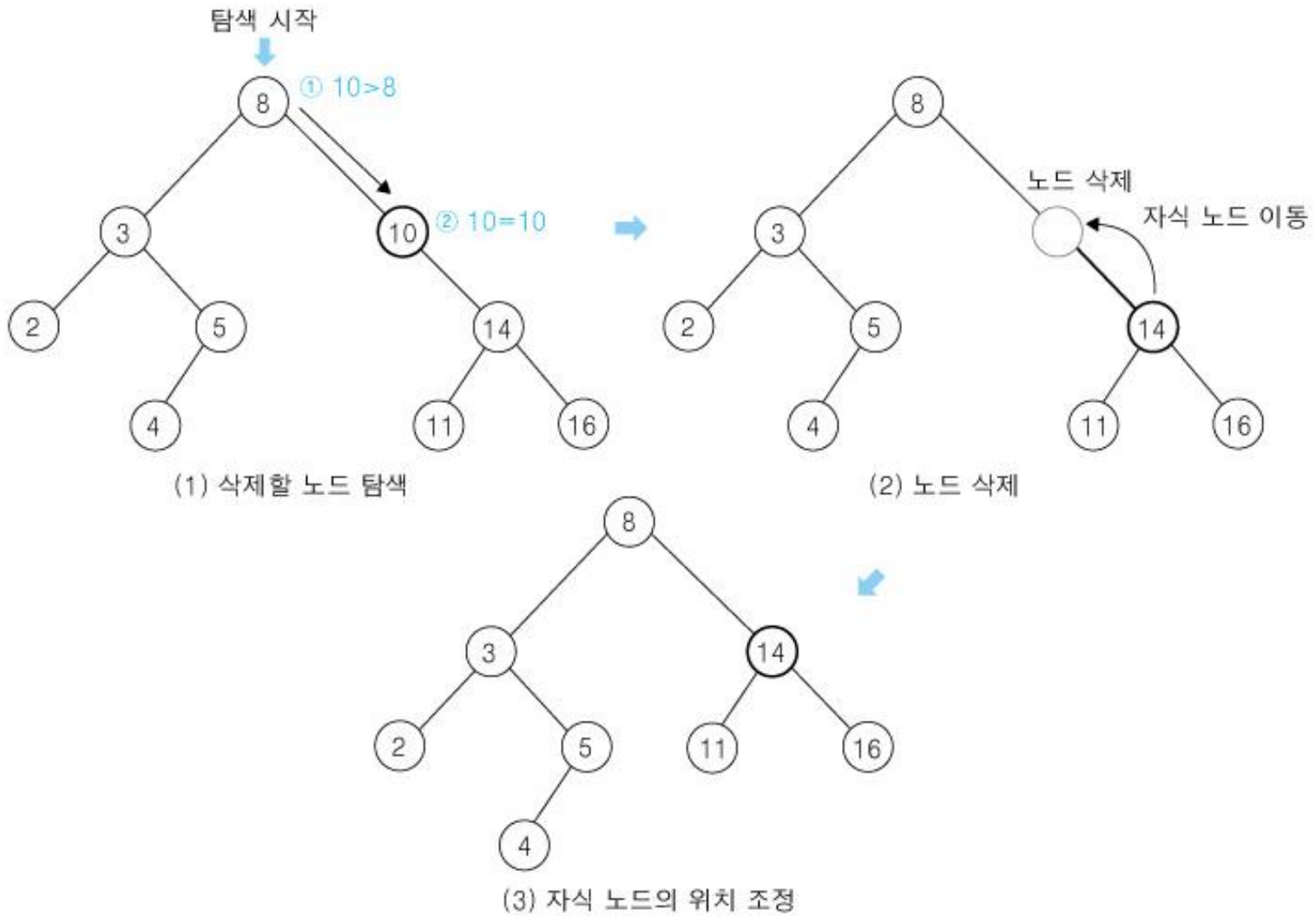
1단계: 삭제할 노드 탐색

2단계: 탐색한 노드 삭제

3단계: 후속처리

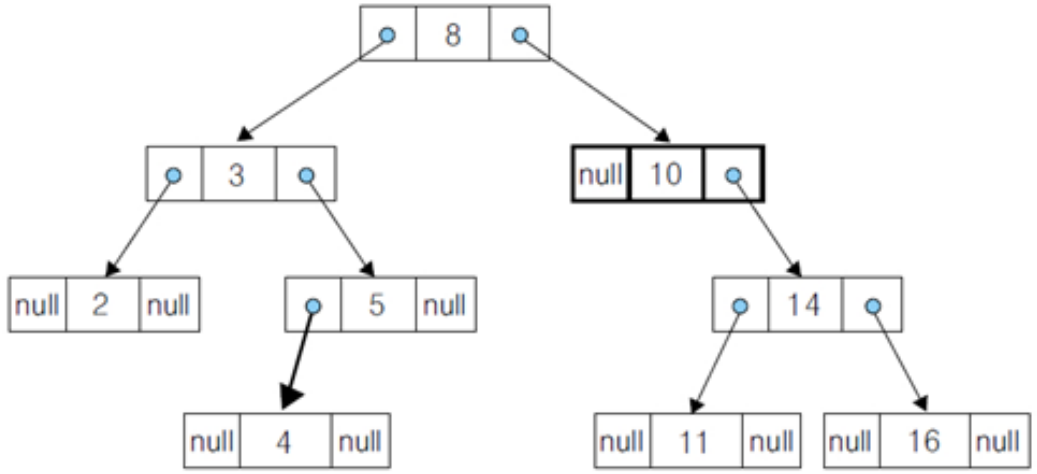
Tree

- 예) 노드 10을 삭제하는 경우

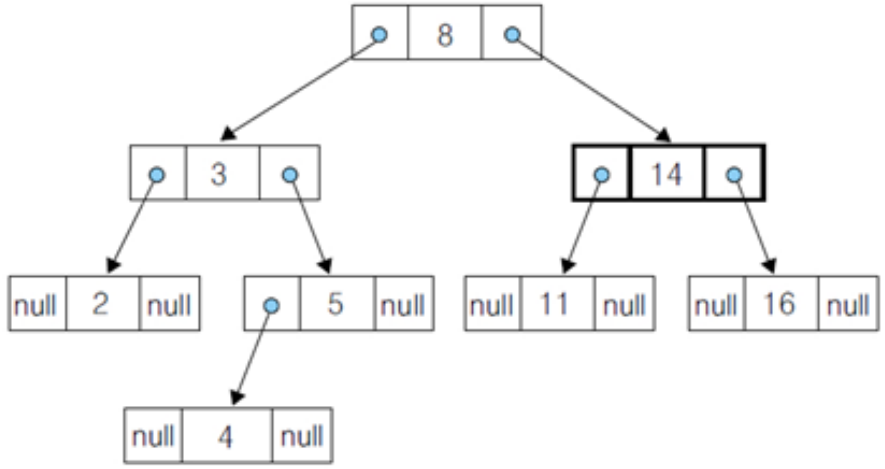


Tree

- 노드 10을 삭제하는 경우에 대한 단순 연결 리스트 표현



(1) 이진 탐색 트리의 연결 표현 : 삭제 전



(2) 이진 탐색 트리의 연결 표현 : 삭제 후

Tree

- 자식 노드가 둘인 노드, 즉 차수가 2인 노드의 삭제 연산
  - 노드를 삭제하면, 자식 노드들은 트리에서 연결이 끊어져서 고아가 된다.
  - **후속 처리 : 이진 탐색 트리의 재구성**
    - 삭제한 노드의 자리를 **자손 노드들 중에서 선택한 후계자에게 물려준다**.

  - 후계자 선택 방법

    방법1) 왼쪽 서브트리에서 가장 큰 자손노드 선택

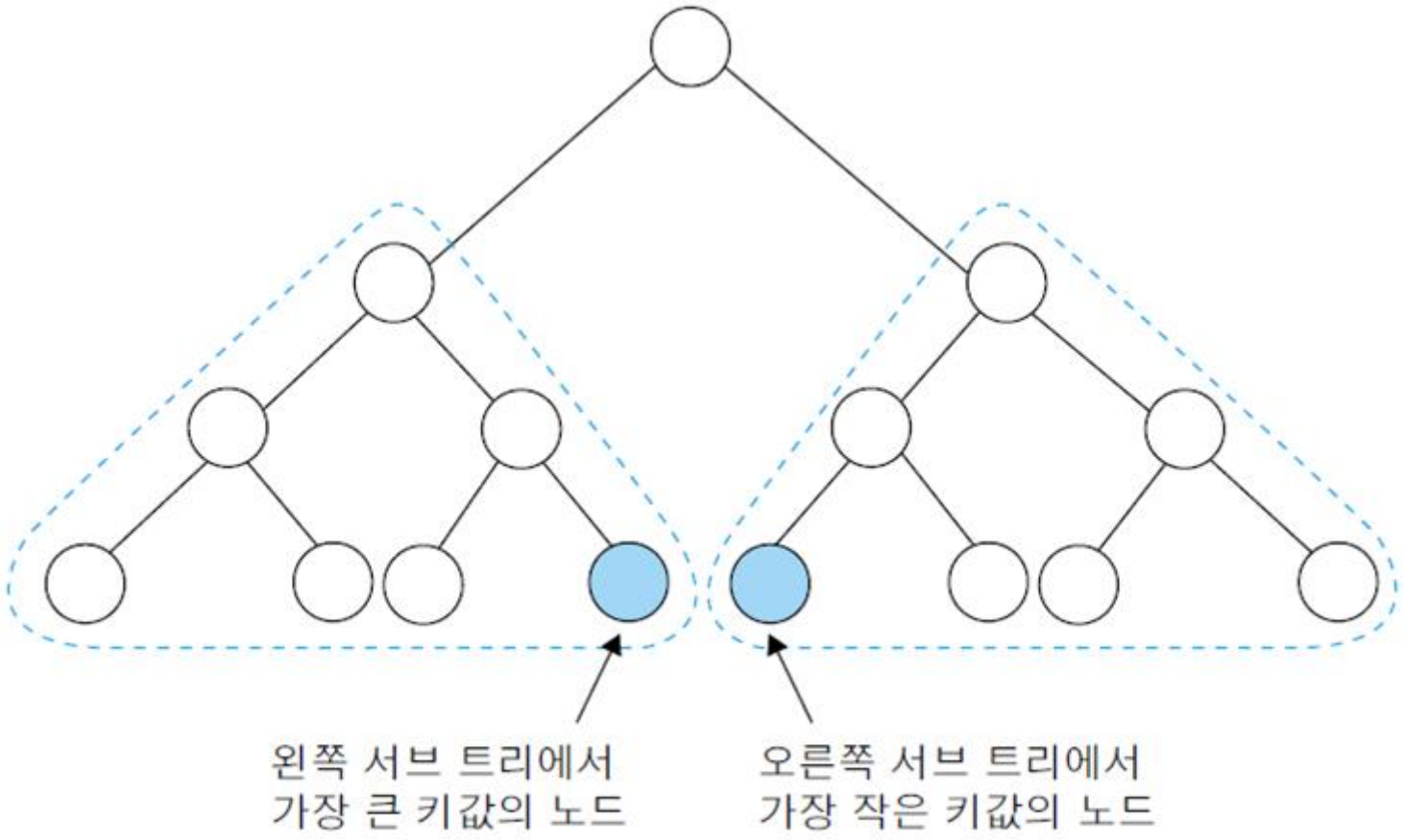
    » 왼쪽 서브트리의 오른쪽 링크를 따라 계속 이동하여 오른쪽 링크 필드가 NULL인 노드 즉, 가장 오른쪽에 있는 노드가 후계자가 된다.

    방법2) 오른쪽 서브트리에서 가장 작은 자손노드 선택

    » 오른쪽 서브트리에서 왼쪽 링크를 따라 계속 이동하여 왼쪽 링크 필드가 NULL인 노드 즉, 가장 왼쪽에 있는 노드가 후계자가 된다.

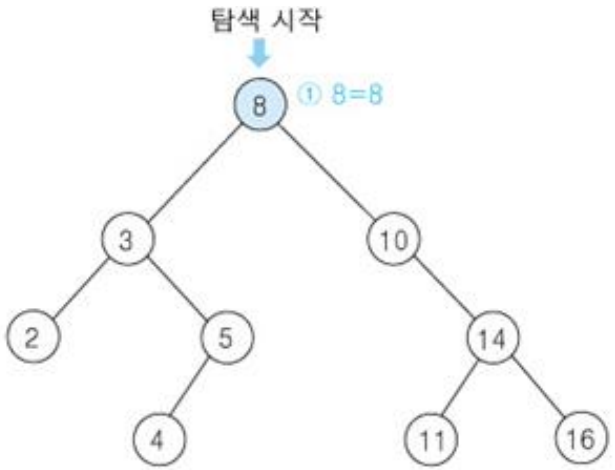
Tree

- 삭제한 노드의 자리를 물려받을 수 있는 후계자 노드

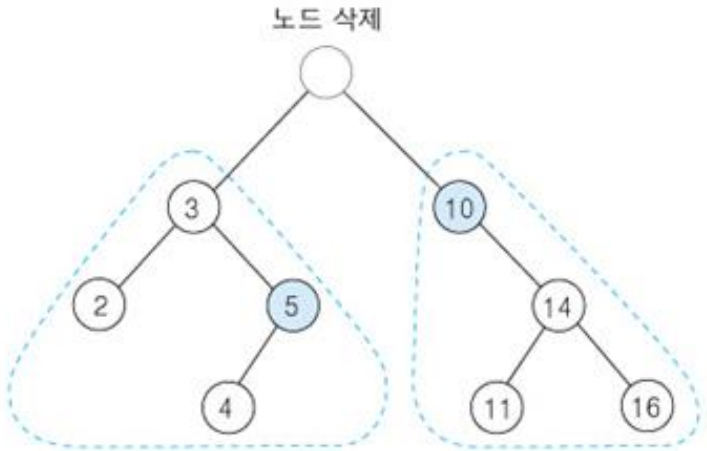


Tree

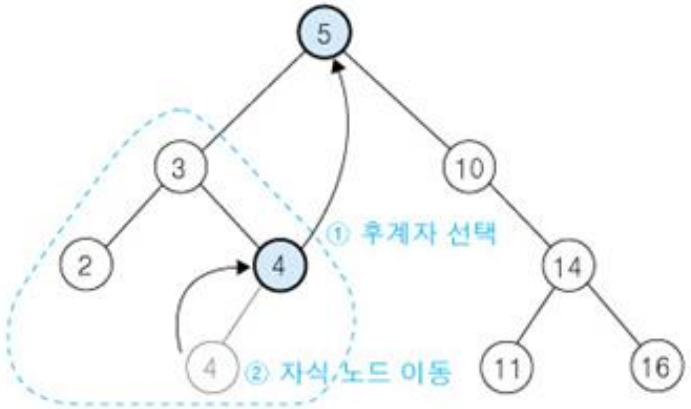
- 예) 노드 8을 삭제하는 경우



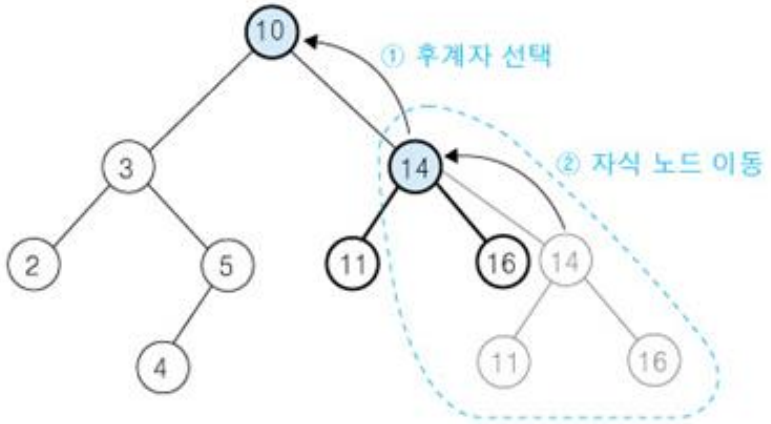
(1) 삭제할 노드 탐색



(2) 노드 삭제



(a) 트리 재구성_노드 5를 후계자로 선택한 경우



(b) 트리 재구성_노드 10을 후계자로 선택한 경우

Tree

- 노드 5를 후계자로 선택한 경우
  ① 후계자 노드 5를 원래자리에서 삭제하여, 삭제노드 8의 자리를 물려준다.
  ② 후계자 노드 5의 원래자리는 자식노드 4에게 물려주어 이진 탐색 트리를 재구성 한다. (☞ **자식노드가 하나인 노드 삭제 연산의 <u>후속처리 수행</u>!)**

- 노드 10을 후계자로 선택한 경우

  ① 후계자 노드 10을 원래자리에서 삭제하여, 삭제노드 8의 자리를 물려준다.
  ② 후계자 노드 10의 원래자리는 자식노드 14에게 물려주어 이진 탐색 트리를 재구성한다. (☞ **자식노드가 하나인 노드 삭제 연산의 <u>후속처리 수행</u>!)**

Tree

- **히프(heap)**
  - 완전 이진 트리에 있는 노드 중에서 키 값이 가장 큰 노드나 키 값이 가장 작은 노드를 찾기 위해서 만든 자료구조

  - **최대 히프(max heap)**
    - 키값이 가장 큰 노드를 찾기 위한 **완전 이진 트리**
    - {부모노드의 키값 ≥ 자식노드의 키값}
    - 루트 노드 : **키 값이 가장 큰 노드**
  - **최소 히프(min heap)**
    - 키값이 가장 작은 노드를 찾기 위한 **완전 이진 트리**
    - {부모노드의 키값 ≤ 자식노드의 키값}
    - 루트 노드 : **키 값이 가장 작은 노드**

Tree

- **히프(heap)가 추상 자료형**

```
ADT  Heap
   데이터 : n개의 원소로 구성된  완전 이진 트리로서 각 노드의 킷값은
            그의 자식 노드의 킷값보다 크거나 같다.
            (부모 노드의 킷값 ≥ 자식 노드의 킷값)
   연산 :
        heap∈Heap; item∈Element;

        createHeap() ::= create an empty heap;
                // 공백 히프의 생성 연산

        isEmpty(heap) ::= if (heap is empty) then  return  true;
                                  else return false;
                // 히프가 공백인지를 검사하는 연산

        insertHeap(heap, item) ::= insert item into heap;
                // 히프의 적당한 위치에 원소(item)를 삽입하는 연산

        deleteHeap(heap) ::= if (isEmpty(heap)) then return error;
                                  else {
                                          item ← 히프에서  가장 큰 원소;
                                          remove {히프에서 가장 큰 원소};
                                          return  item;
                                       }
                // 히프에서 킷값이 가장 큰 원소를 삭제하고 반환하는 연산
End Heap()
```

Tree

- **히프에서의 삽입 연산**
  - 1단계 : 완전 이진 트리를 유지하면서 확장한 노드에 삽입할 원소를 임시 저장
    - 노드가 n개인 완전 이진 트리에서 다음 노드의 확장 자리는 n+1번의 노드가 된다.
    - n+1번 자리에 노드를 확장하고, 그 자리에 삽입할 원소를 임시 저장한다.
  - 2단계 : 만들어진 완전 이진 트리 내에서 삽입 원소의 제자리를 찾는다.
    - 현재 위치에서 부모노드와 비교하여 크기 관계를 확인한다.
    - {현재 부모노드의 키값 ≥ 삽입 원소의 키값}의 관계가 성립하지 않으면, 현재 부모노드의 원소와 삽입 원소의 자리를 서로 바꾼다.

Tree

- **히프에서의 삽입 연산 알고리즘**

① 현재 히프의 크기를 하나 증가시켜서 노드 위치를 확장하고, 확장한 노드 번호가 현재의 삽입 위치 i가 된다.

② 삽입할 원소 item과 부모 노드 heap[⌊i/2⌋]를 비교하여 item이 부모 노드 보다 작거나 같으면 현재의 삽입 위치 i를 삽입 원소의 위치로 확정한다.

③ 만약 삽입할 원소 item이 부모 노드보다 크면, 부모 노드와 자식 노드의 자리를 바꾸어 최대 히프의 관계를 만들어야 하므로 부모 노드 heap[⌊i/2⌋]를 현재의 삽입 위치 heap[i]에 저장하고,

④ i/2를 삽입 위치 i로 하여, ②~④를 반복하면서 item을 삽입할 위치를 찾는다.

⑤ 찾은 위치에 삽입할 노드 item을 저장하면, 최대 히프의 재구성 작업이 완성되므로 삽입 연산을 종료한다.

Tree

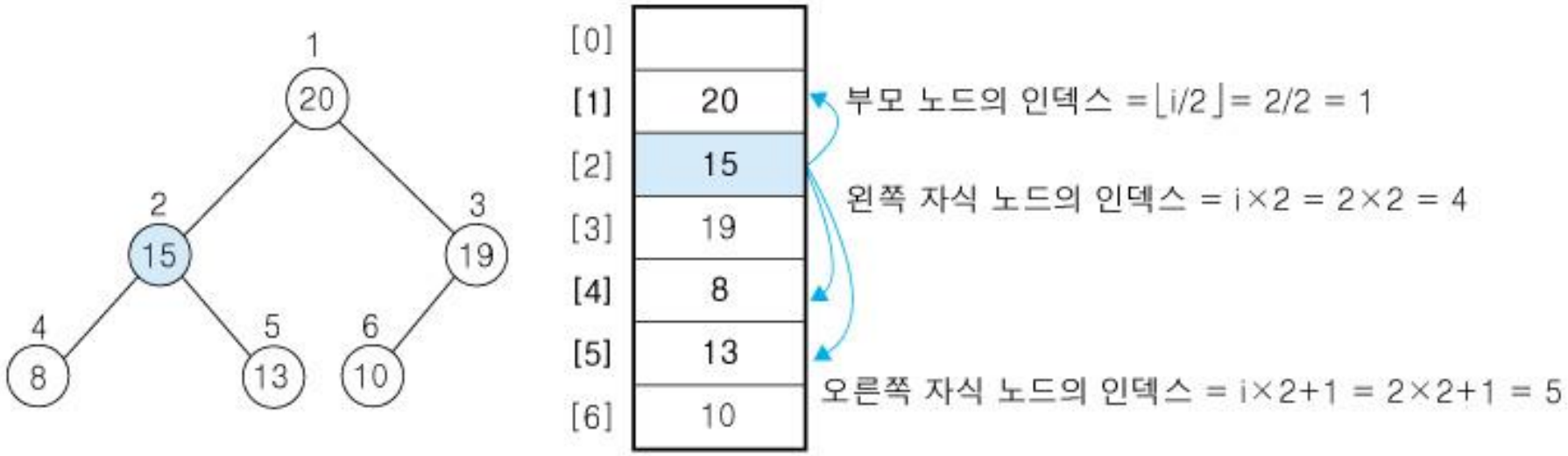
- **히프에서의 삭제 연산**
  - 히프에서는 루트 노드의 원소만을 삭제 할 수 있다.
  - 1단계 : 루트 노드의 원소를 삭제하여 반환한다.
  - 2단계 : 원소의 개수가 n-1개로 줄었으므로, 노드의 수가 n-1인 완전 이진 트리로 조정한다.
    - 노드가 n개인 완전 이진 트리에서 노드 수 n-1개의 완전 이진 트리가 되기 위해서 마지막 노드, 즉 n번 노드를 삭제한다.
    - 삭제된 n번 노드에 있던 원소는 비어있는 루트 노드에 임시 저장한다.
  - 3단계 : 완전 이진 트리 내에서 루트에 임시 저장된 원소의 제자리를 찾는다.
    - 현재 위치에서 자식노드와 비교하여 크기 관계를 확인한다.
    - {임시 저장 원소의 키값 ≥ 현재 자식 노드의 키값 }의 관계가 성립하지 않으면, 현재 자식 노드의 원소와 임시 저장 원소의 자리를 서로 바꾼다.

Tree

- **히프에서의 삭제 연산 알고리즘**
  - ① 루트노드 heap[1]을 변수 item에 저장하고,
  - ② 마지막 노드의 원소 heap[n]을 변수temp에 임시 저장한 후에,
  - ③ 마지막 노드를 삭제하고 히프배열의 원소 개수를 하나 감소한다.
  - ④ 마지막 노드의 원소였던 temp의 임시 저장위치 i는 루트노드의 자리인 1번이 된다.
  - ⑤ 현재 저장위치에서 왼쪽 자식 노드 heap[j]와 오른쪽 자식 노드 heap[j+1]이 있을 때, 둘 중에서 킷값이 큰 자식 노드의 킷값과 temp를 비교하여, temp가 크거나 같으면 현재 위치가 temp의 자리로 확정된다.
  - ⑥ 만약 temp가 자식노드보다 작으면, 자식노드와 자리를 바꾸고 다시 ⑤~⑥을 반복하면서 temp의 자리를 찾는다.
  - ⑦ 찾은 위치에 temp를 저장하여 최대 히프의 재구성 작업을 완성하고
  - ⑧ 루트노드를 저장한 item을 반환하는 것으로 삭제 연산을 종료한다.

Tree

- **순차 자료구조를 이용한 히프의 구현**
  - 부모노드와 자식노드를 찾기 쉬운 1차원 배열의 순차 자료구조 이용
  - 1차원 배열을 이용한 히프의 표현 예



[그림 8-33] 순차 자료구조를 이용한 히프의 표현 예