# 자료구조

## L06 Binary Trees (2)

2022년 1학기

국민대학교 소프트웨어학부

# In this lecture

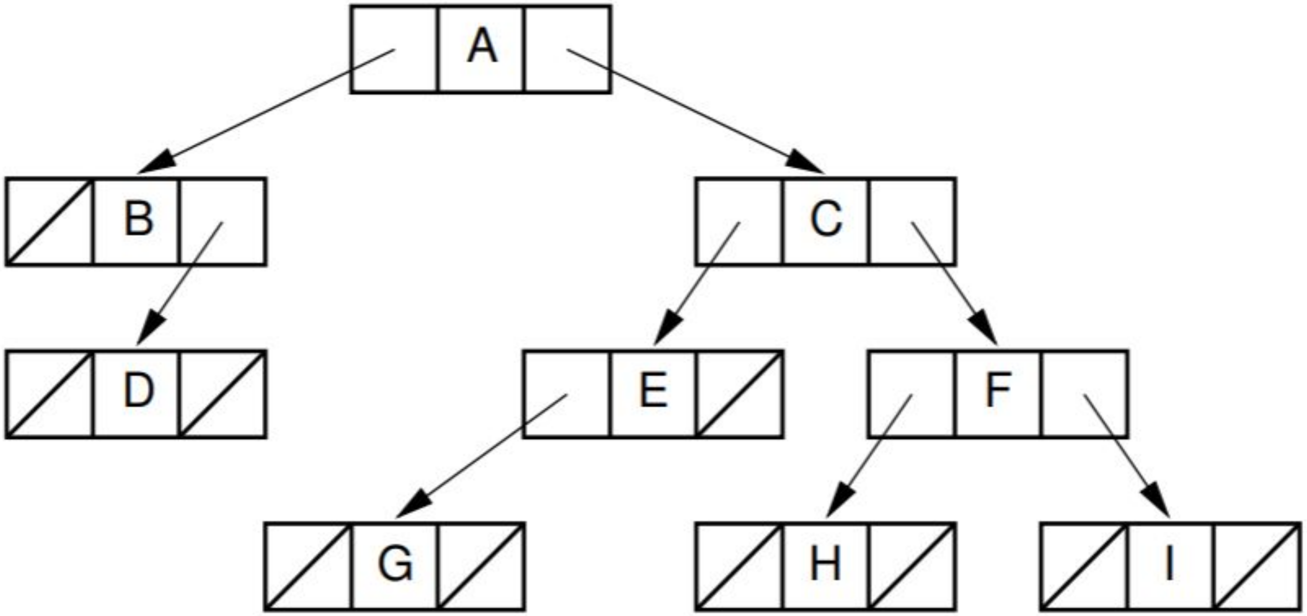
- Implementation and space overhead of binary tree
- Main idea and operations for BST (Binary Search Tree)
- Complexity and implementations for BST

# Outline

- **Implementation of Binary Tree**
- Binary Search Trees

# Binary Tree Implementation (1)

- 방법1: 리프 노드와 내부 노드를 **똑같이** 구현한다.



# Space Overhead

- Space overhead = total space - data space
  - Overhead fraction = (space overhead) / (total space)

- 정 이진 트리 정리로 부터:
  - 포인터의 절반은 **null**이다.

- 리프 노드만 데이터를 저장한다면? Space overhead는 트리가 full인지 아닌지에 따라 달라진다.
  - 예) 내부노드는 많은데 리프노드가 하나라면?

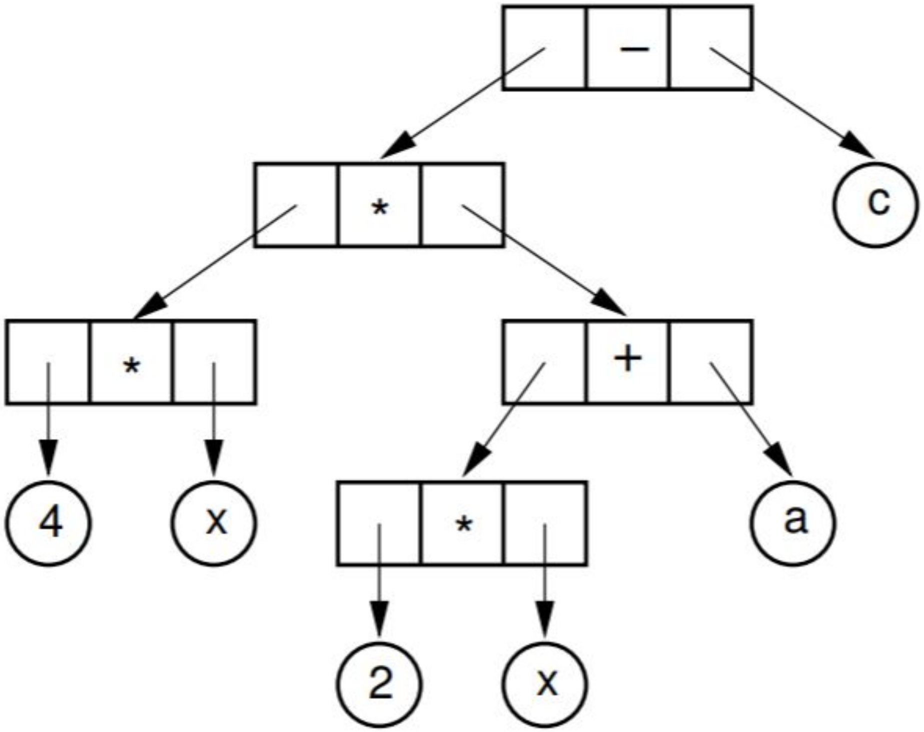
# Space Overhead

- 예) full tree이고, 내부 노드와 리프노드를 똑같이 구현하는 경우 (2개의 포인터 1개의 데이터)
  - Total space: (2P + D)n
  - Overhead: 2Pn
  - P = D인 경우, overhead fraction= 2P/(2P+D) = 2/3

- Overhead를 어떻게 줄일까?
  - Idea: 리프노드에서 child pointer를 제거한다.
  - Overhead fraction: P/(P+D) (Why?)
  - P = D인 경우 overhead fraction: 1/2

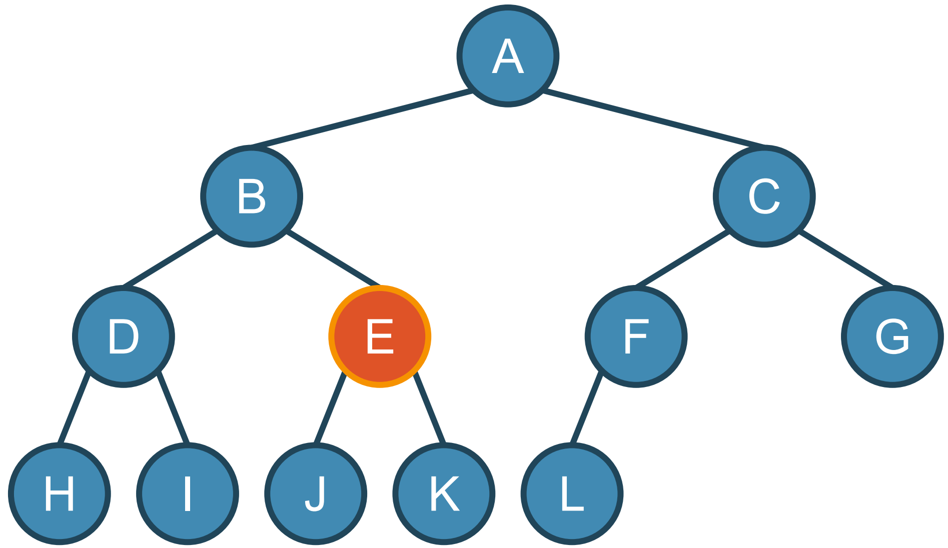
# Binary Tree Implementation (2)

- 방법2: 리프 노드와 내부 노드를 **다르게** 구현한다.




Between implementations (1) and (2), which is better? Why?

# Array Implementation for Complete Binary Tree




| A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

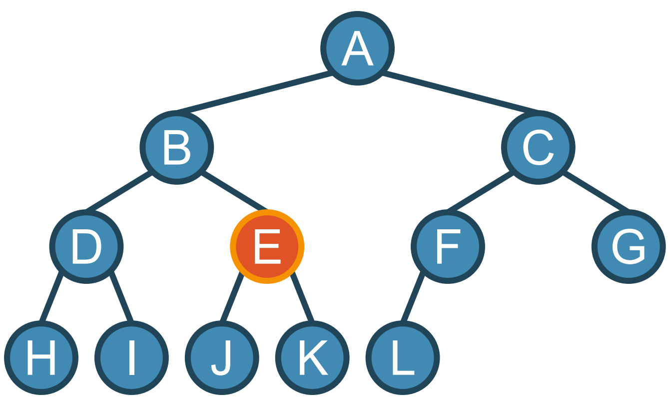
Parent

Left Child

Right Child

# Array Implementation for Complete Binary Tree

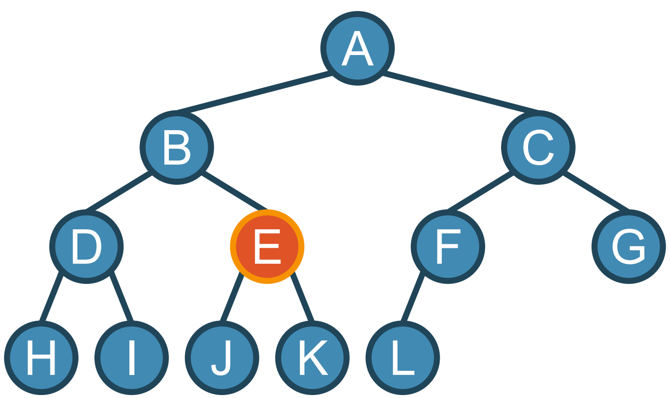
- `parent(r)        =`
- `leftchild(r)     =`
- `rightchild(r)    =`
- `leftsibling(r)   =`
- `rightsibling(r)  =`




| A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

# Array Implementation for Complete Binary Tree

- parent(r) = $\lfloor (r-1)/2 \rfloor$ (if $r > 0$)
- leftchild(r) = $2r+1$ (if $2r+1 < n$)
- rightchild(r) = $2r+2$ (if $2r+2 < n$)
- leftsibling(r) = $r-1$ (if r is even)
- rightsibling(r) = $r+1$ (if r is odd & $r+1 < n$ )




| A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

# Outline

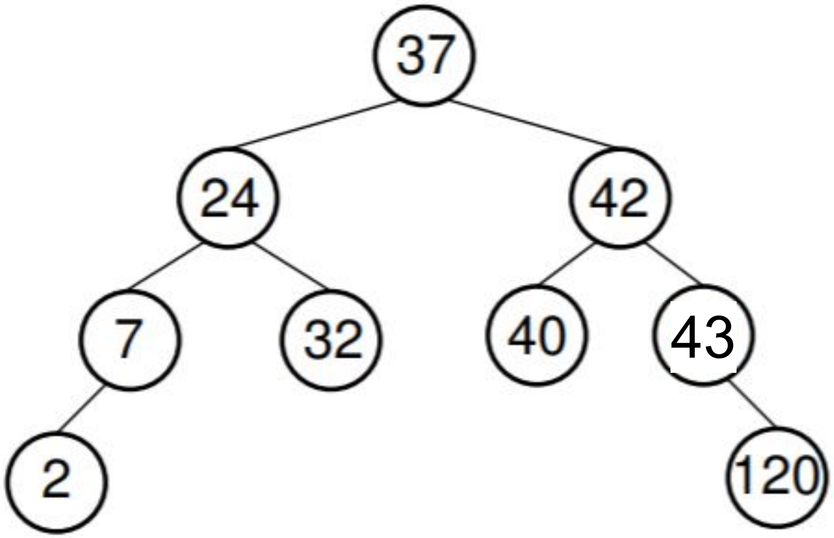
- Implementation of Binary Tree
- **Binary Search Trees**

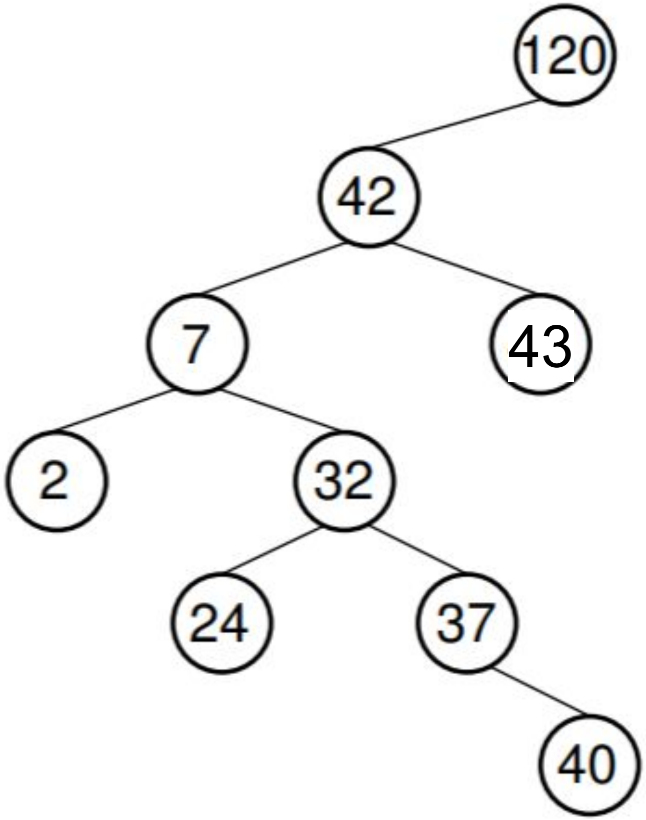
# Binary Search Trees

- 이진 탐색 트리(BST)는 다음 조건을 따르는 이진 트리이다.
  - 모든 노드는 **유일한 키**를 갖는다.
  - K라는 키를 갖는 노드의 **왼쪽** subtree에 있는 모든 노드는 **K보다 작은 키**를 갖는다.
  - K라는 키를 갖는 노드의 **오른쪽** subtree에 있는 모든 노드는 **K보다 큰 키**를 갖는다.
  - **모든 subtree는 BST**이다.

# Binary Search Trees

- BST examples



# BST as a Dictionary (1)

```java
public interface Dictionary<Key, E> {
    /** Reinitialize dictionary */
    public void clear();

    /** Insert a record */
    public void insert(Key k, E e);

    /** Remove and return a record (null if none exists). */
    public E remove(Key k);

    /** Remove and return an arbitrary record from dictionary. */
    public E removeAny();

    /** Return a record matching "k" (null if none exists). */
    public E find(Key k);

    /** @return The number of records in the dictionary. */
    public int size();
};
```

# BST as a Dictionary (2)

- Time Complexities of Dictionary implementations

| Implementation | Search | Insert/Remove |
| --- | --- | --- |
| Sorted list (array) | | |
| Unsorted list (array) | | |
| Binary Search Tree | | |

$n$: # items

$d$: depth of the tree

# BST as a Dictionary (2)

- Time Complexities of Dictionary implementations

| Implementation | Search | Insert/Remove |
|---|---|---|
| Sorted list (array) | $O(\log n)$ | $O(n)$ |
| Unsorted list (array) | $O(n)$ | $O(1)$ |
| Binary Search Tree | $O(d)$ | $O(d)$ |

$n$: # items

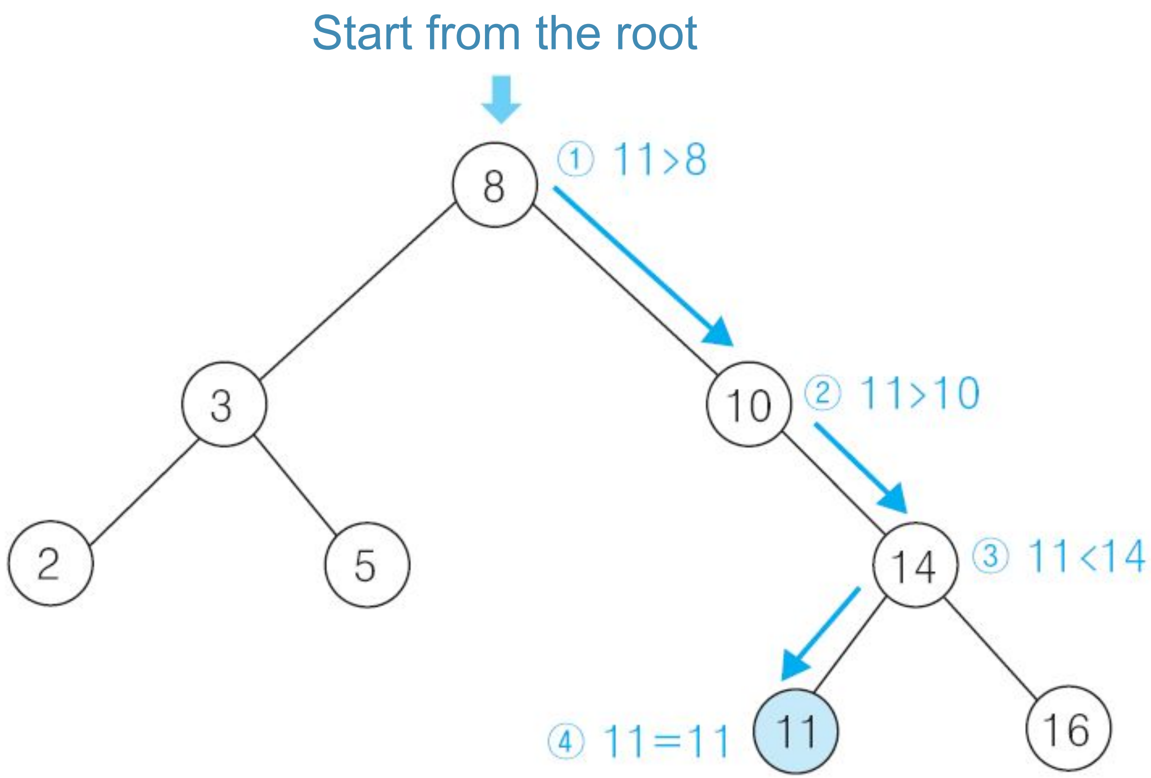
$d$: depth of the tree

# BST Search

- 키가 x인 노드 찾기
  - 루트노드 에서 시작
  - x와 루트노드의 키 rt.key와 비교해서
    - x = rt.key 이면 rt를 리턴
    - x < rt.key 이면 rt의 왼쪽 subtree에서 다시 찾기
    - x > rt.key 이면 rt의 오른쪽 subtree에서 다시 찾기

# BST Search

- E.g.) To find the element of key 11

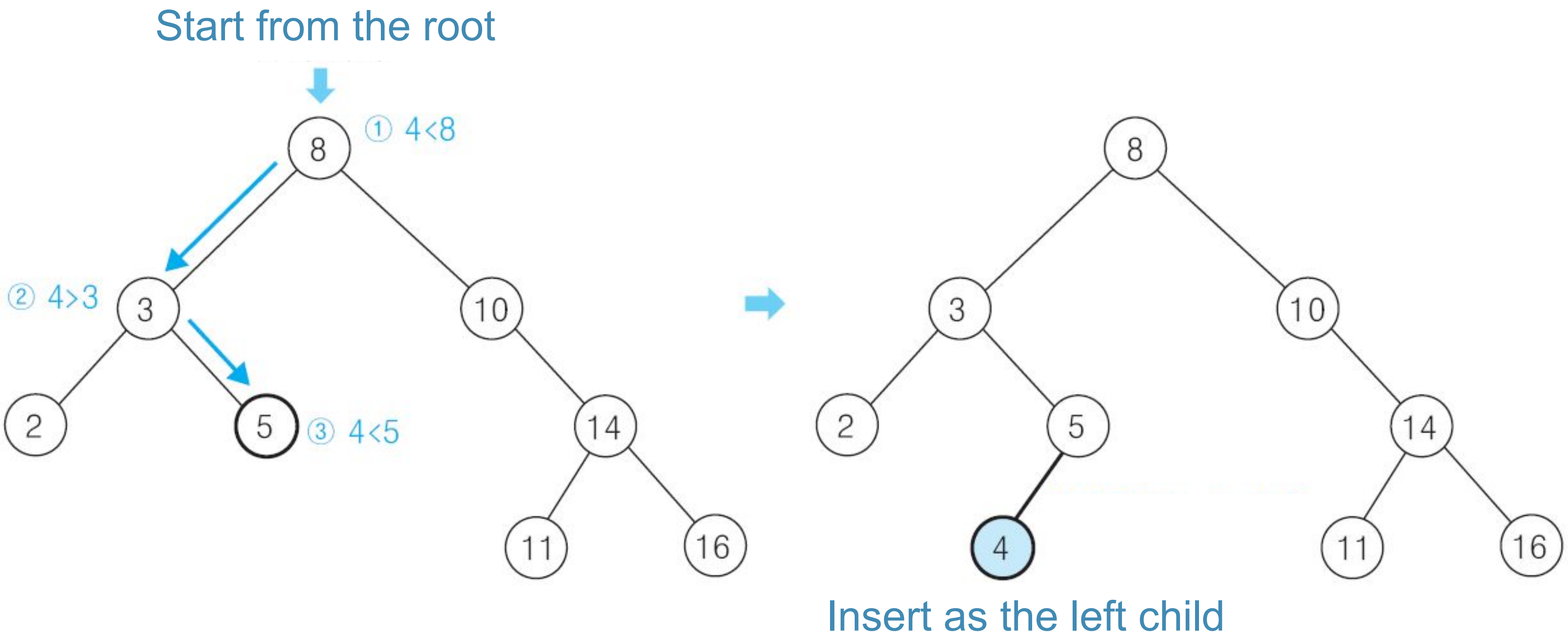


# BST Insert

- 키가 x인 노드 넣기
  - BST Search를 수행
    - 키가 x인 노드가 이미 있다면? 넣기 실패
  - 탐색이 실패한 위치에 키가 x인 노드를 넣음

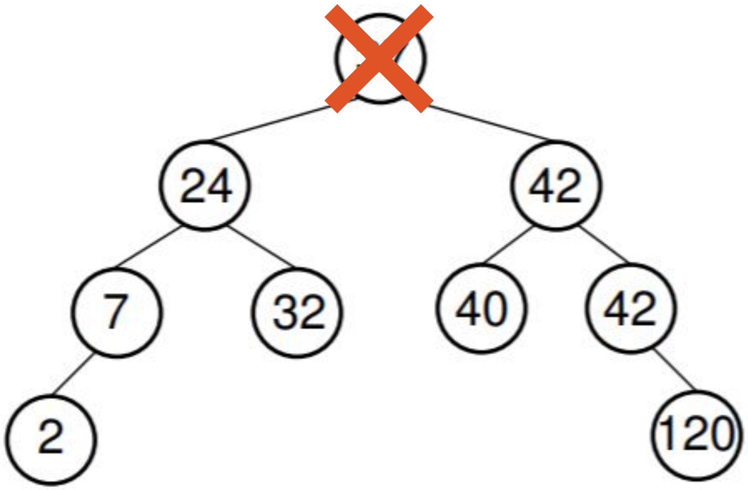
# BST Insert

- E.g.) To insert the element of key 4



# BST Remove

- 키가 x인 노드 지우기
  - BST Search를 수행
    - 키가 x인 노드가 없다면? 지우기 실패
  - 찾은 노드를 지움
  - 다른 노드를 지운 자리로 옮겨줌
    - 어떤 노드를 옮겨줘야할까?

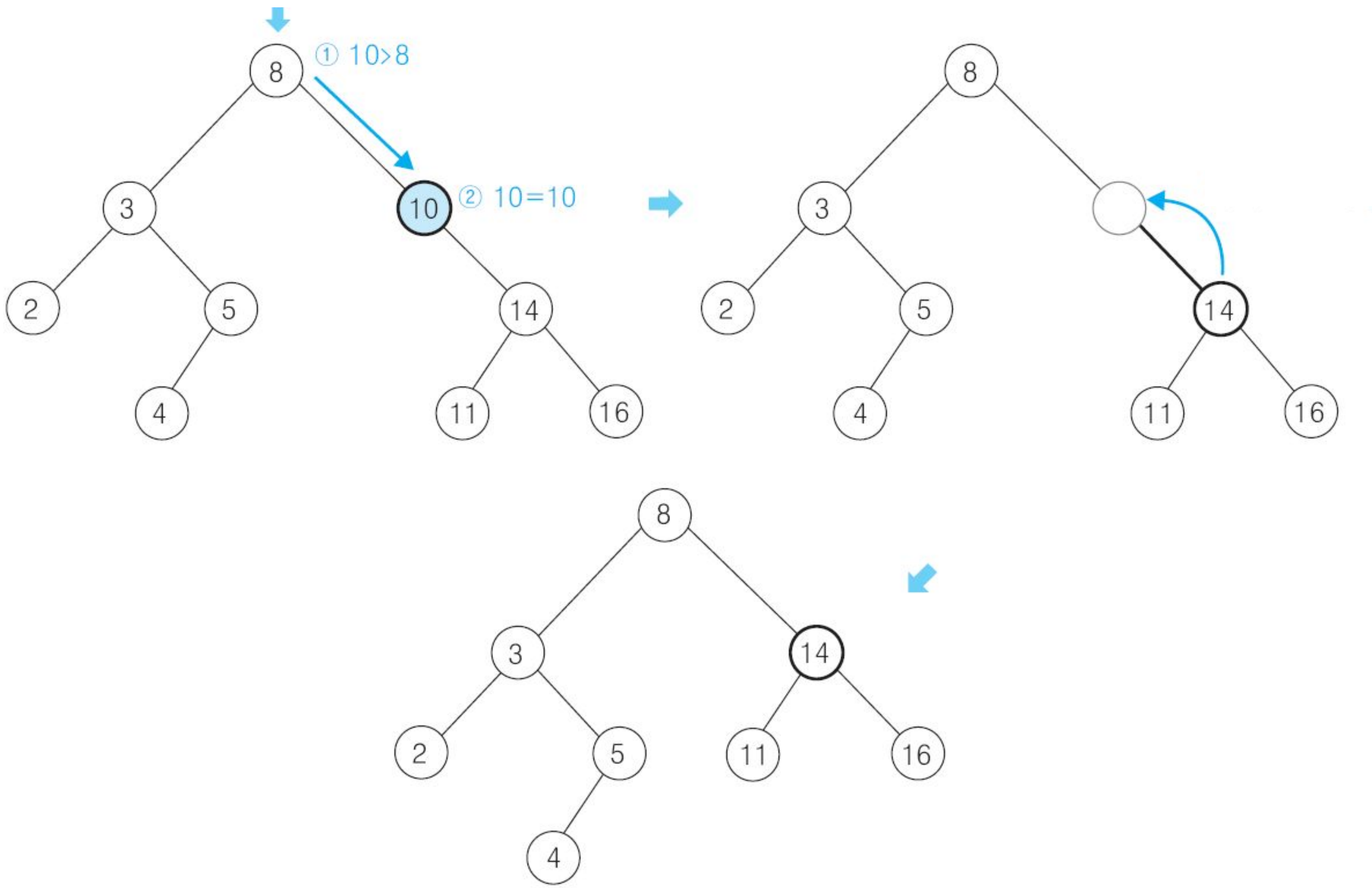


# BST Remove

- 키가 x인 노드 지우기 (계속)
  - Root에서 시작
  - x < rt.key 인 경우: x를 왼쪽 subtree에서 지움
  - x > rt.key 인 경우: x를 오른쪽 subtree에서 지움
  - x = rt.key 인 경우 rt를 지운다
    - 왼쪽 자식이 없다면, 오른쪽 subtree가 현재 subtree를 대체한다.
    - 오른쪽 자식이 없다면, 왼쪽 subtree가 현재 subtree를 대체한다.
    - 양쪽 자식이 모두 있다면, 오른쪽 subtree에서 가장 작은 키를 갖는 노드가 현재 subtree의 새로운 루트가 된다.
      - 가장 작은 키를 갖는 노드의 **오른쪽 subtree**가 가장 작은 키를 갖는 노드의 subtree를 대체한다.

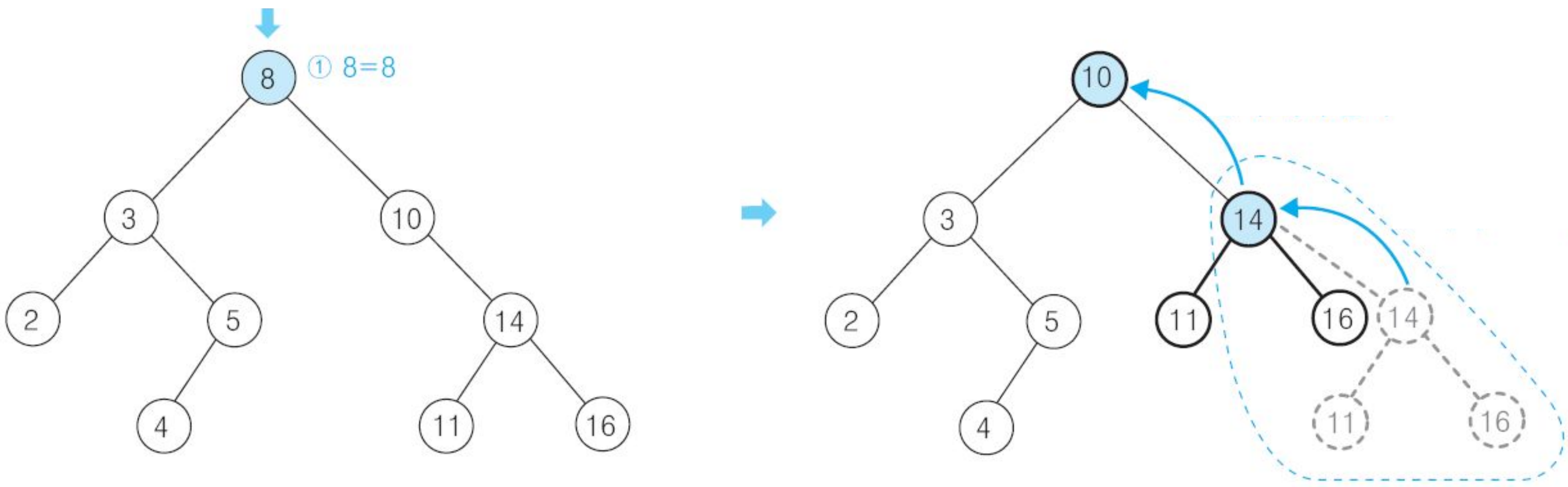
# BST Remove

- E.g.) When the removed node 10 has no left child



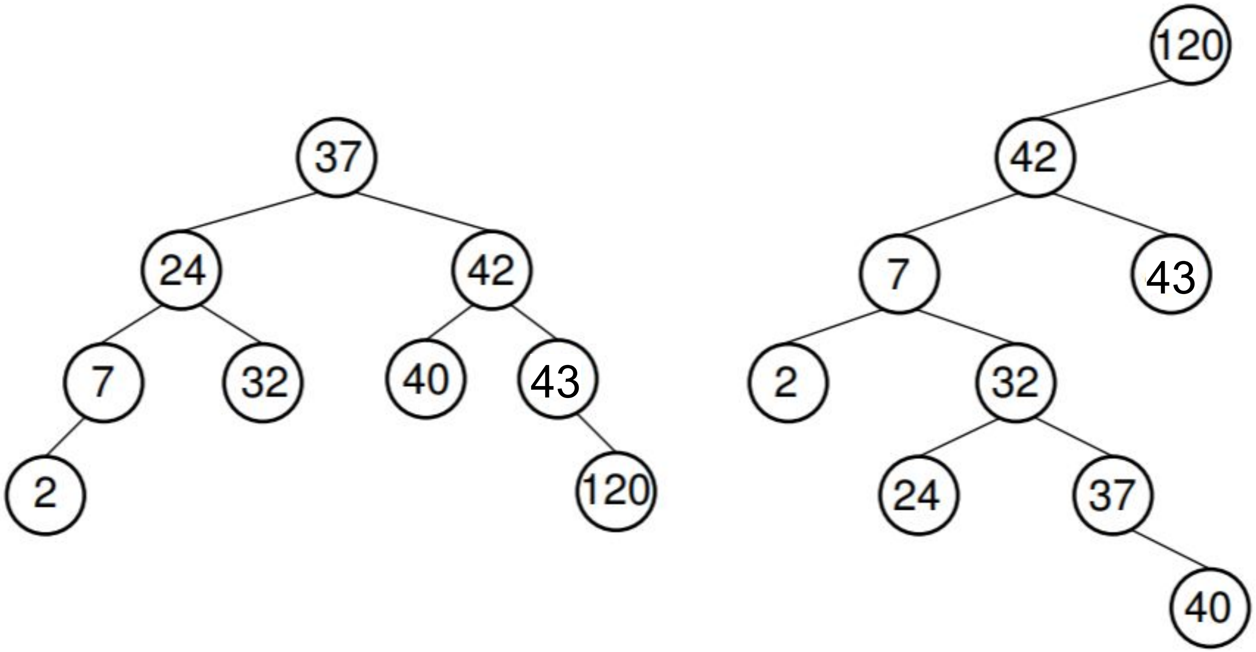
# BST Remove

- E.g.) When the removed node 8 has both children



# BST and Traversal

- BST에서 중위순회를 하면?
  - Sorted key values (in increasing order) !



# Time Complexity of BST Operations

- Find: $O(d)$
- Insert: $O(d)$
- Removal: $O(d)$
- $d$: depth of the tree
- $d$ is $O(\log n)$ if tree is balanced. What is the worst case?

# What You Need to Know

- Implementations and space overhead of binary tree
- Array-based implementations for complete binary tree
- How to implement link-based BST operation
    - Find, Insert, Remove …
    - Time complexity of BST operations

# Questions?