Data structure [A10]

김종규, PhD

2017-05-10

Review

- C99 standard

```
for(int i = 0; i < 10; i++) {
  printf("%d\n", i);
}

int i;
for(i = 0; i < 10; i++) {
  printf("%d\n", i);
}
```

⟶ Old (C90 or C80) standard

Remarks on C semantics

```
for(int i = 0; i < 10; i++) {
   ...
}
printf("%d\n", i);
```

$\longrightarrow$ Error: ‘i’ undeclared

Remarks on C semantics

```
{
   int i;
   for(i = 0; i < 10; i++) {
      ...
   }
}
printf("%d\n", i);
```

$\longrightarrow$ Error: ‘i’ undeclared

Review: successor

```
TREE-SUCCESSOR(x)
1   if x.right ≠ NIL
2         return TREE-MINIMUM(x.right)
3   y = x.p
4   while y ≠ NIL and x == y.right
5         x = y
6         y = y.p
7   return y
```

그림: Successor

⟶ parent pointer

Review: Doubly Linked list

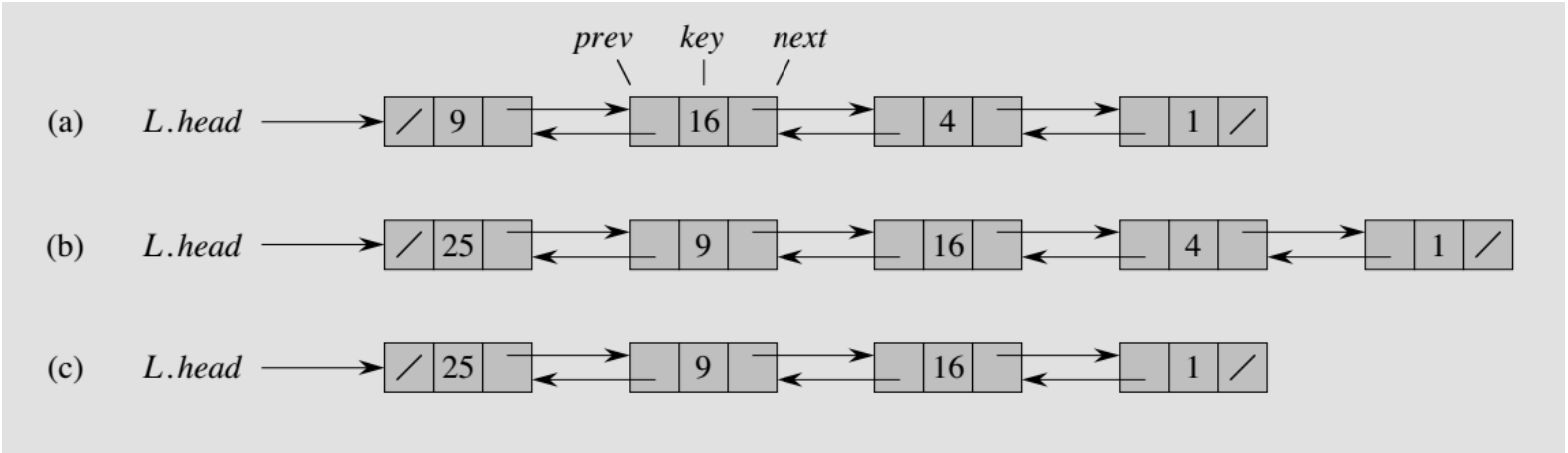

그림: Concept of doubly linked list

Review: Search a linked list

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

그림: Searching a value

- $O(n)$

Review: Insert a node to a linked list

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3        L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

그림: Inserting a node

- $O(1)$

Review: Delete a node from a linked list

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

그림: Deleting a node

- $O(1)$

Outline

- Deleting a node from a binary search tree
  - Many cases

Deleting a node

- Is it difficult to insert a node in a linked list?
  - No, it's quite straightforward
- Is it difficult to insert a node in a binary search tree?
  - No, but it's a bit complicated (BST property)
- Is it difficult to delete a node in a linked list?
  - No, but it's a bit complicated (maintain previous pointer)
- What about deleting a node in a binary search tree?
  - It's a bit more complicated than insert
  - We need to find successor and predecessor

Deleting a node

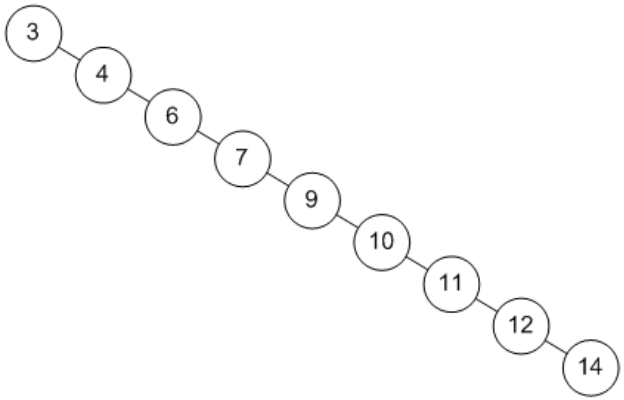

그림:

- Change the name of pointers: parent $\longrightarrow$ prev, right $\longrightarrow$ next, ignore left
- Is this different from doubly linked list? No

$\longrightarrow$ Whenever we delete a node where left or right is NIL, we could easily delete it as if it were a node from a doubly linked list

Simple case – 1

- Node to be removed has no children

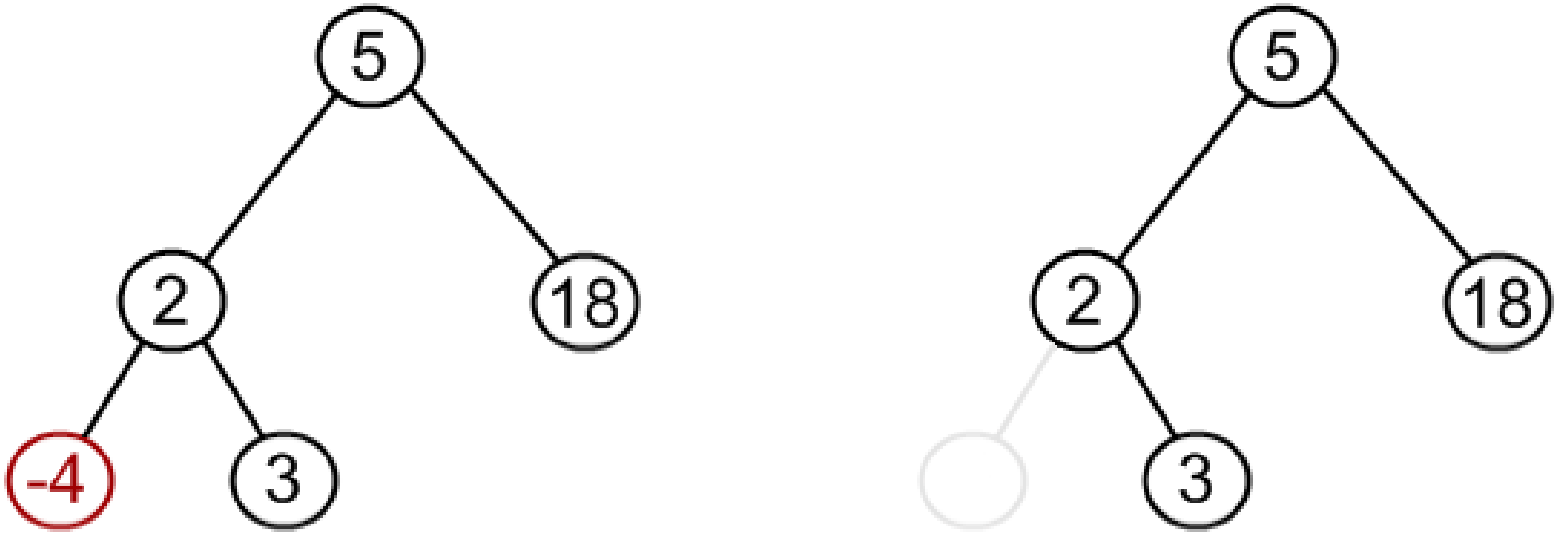

그림: remove

Simple case – 2

- No left or right child

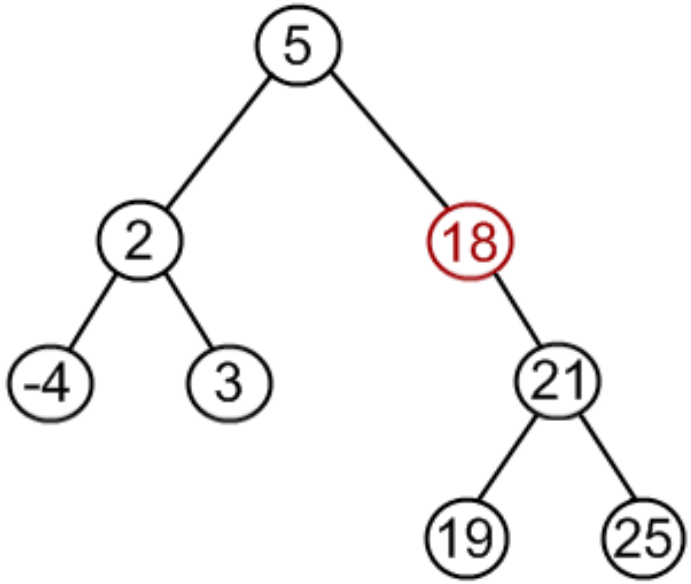

그림: remove

Simple case – 3

- Similar to deleting a node in a doubly linked list

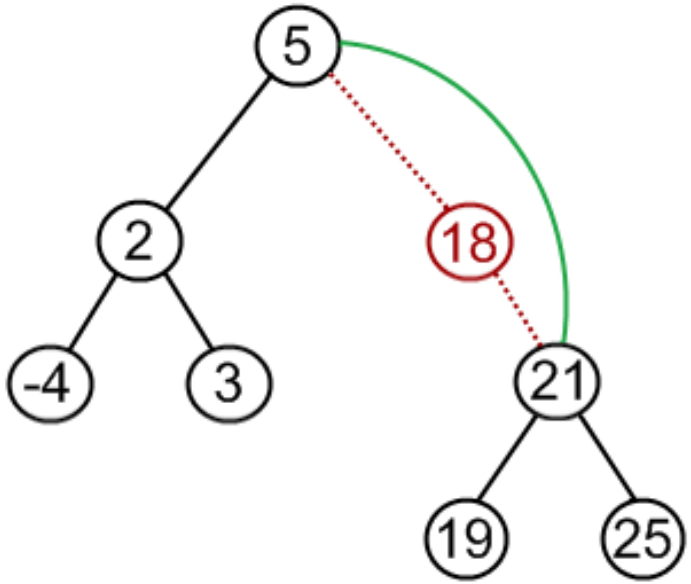

그림: remove

Skew from deletion – 1

- Deleting a subtree: left of 5

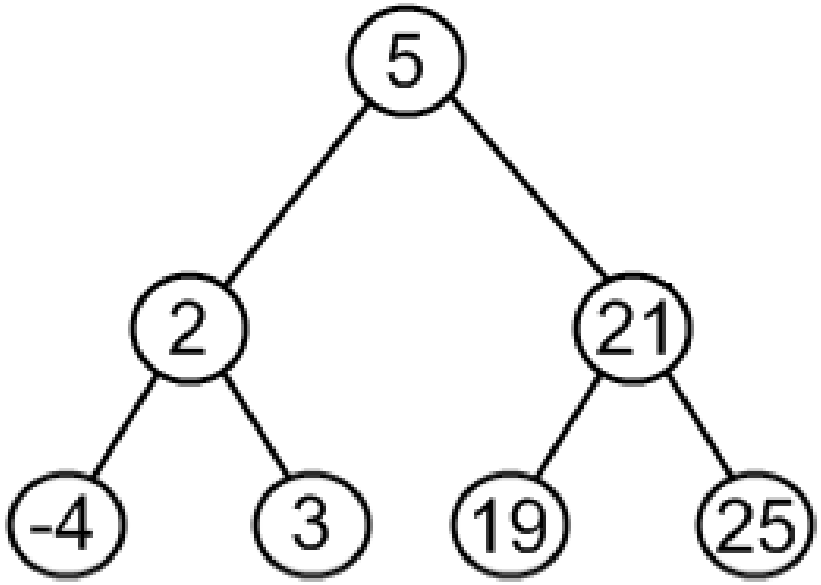

그림: remove

Skew from deletion – 2

- Removed all nodes less than 5?

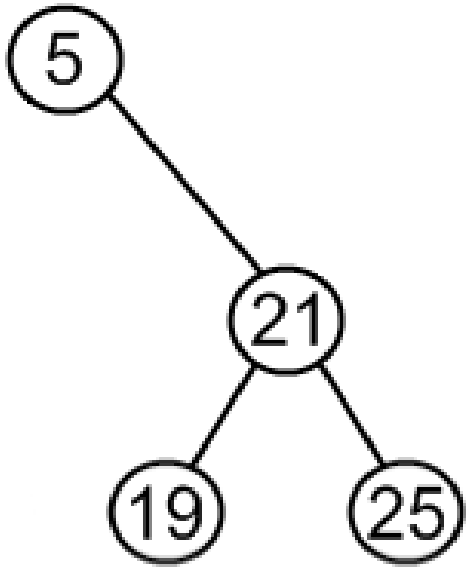

그림: remove

- What's different from list?

Skew from deletion – 3

- What if we rotate the tree?

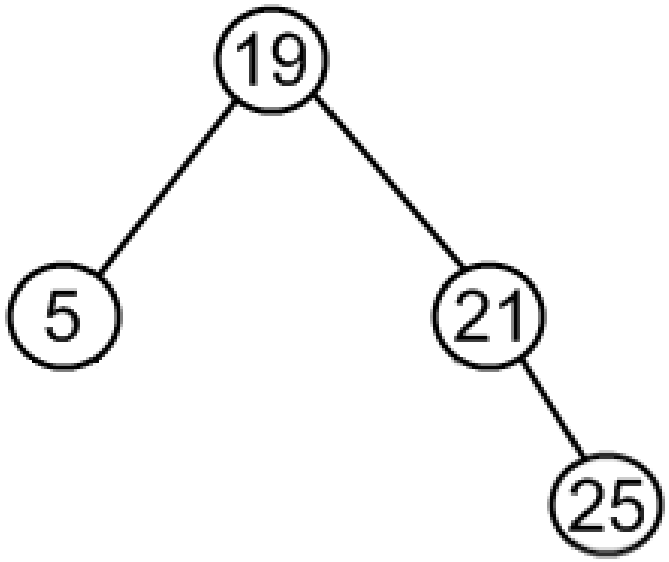

그림: remove

More complex case – 1

- Can we treat the node 12 as one from a doubly linked list?

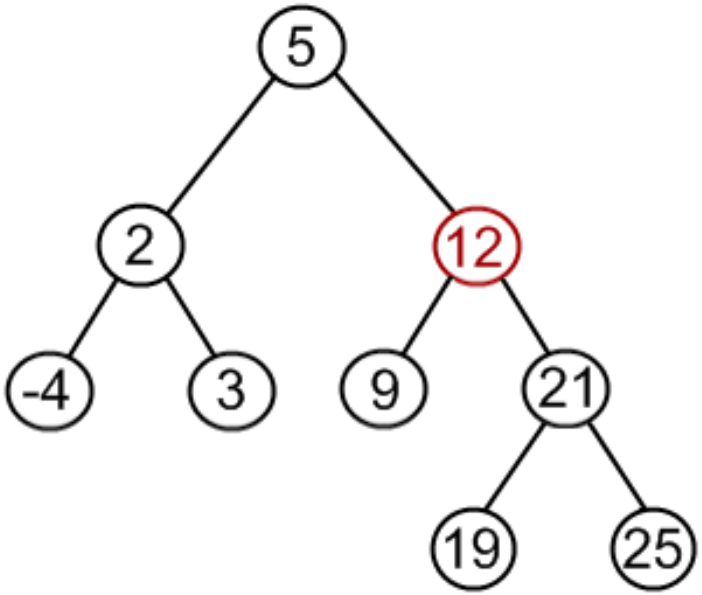

그림: remove

$\longrightarrow$ No. Neither left nor right empty.

More complex case – 2

- Suppose we deleted the node 12, what is a candidate node to be substituted? Note that we should maintain BST property.

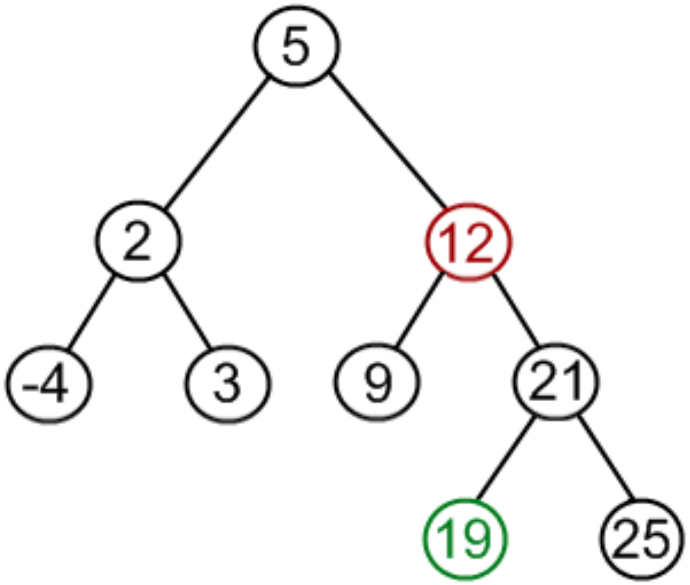

그림: remove

More complex case – 3

- To substitute the node 12, the substituted node should be deleted from the tree

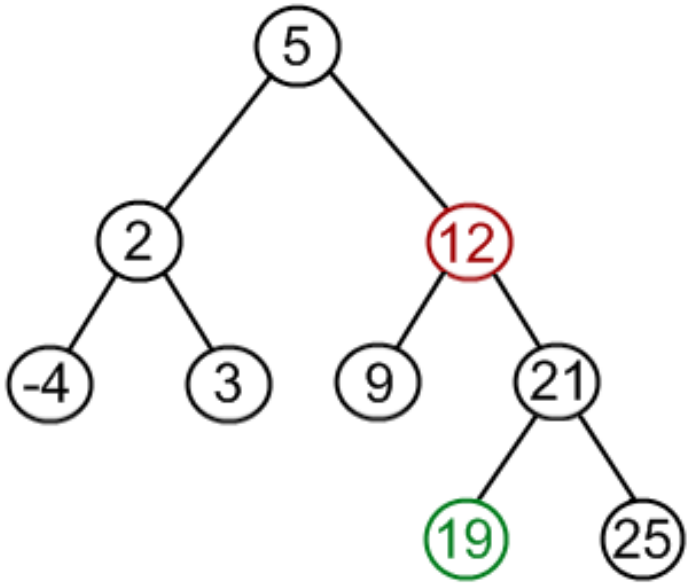

그림: remove

More complex case – 4

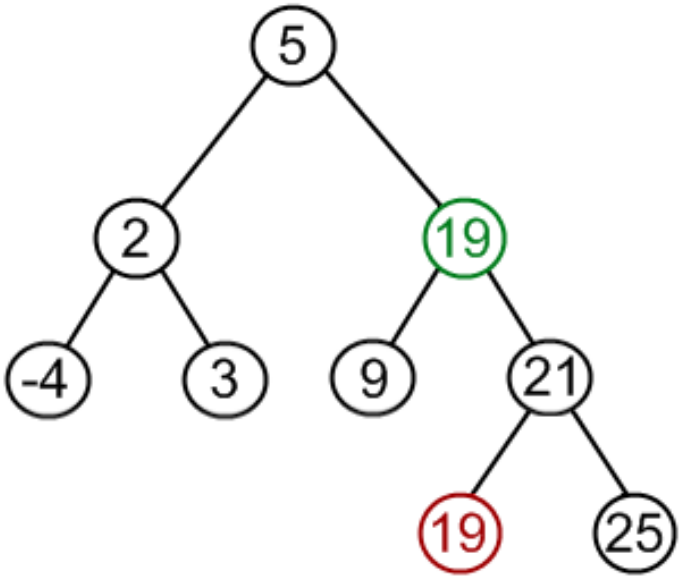

그림: remove

- In this case, the successor of 12 is the most desirable candidate
  - Easy to delete: leaf node
  - Satisfies the BST property

Complexity

- How long will it take?

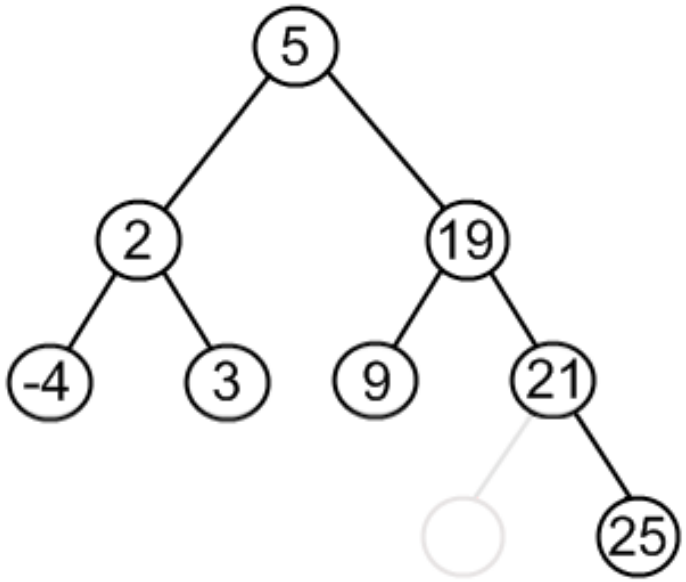

그림: remove

- Find successor: $O(h)$, where h is the height of the BST.

Delete(z)

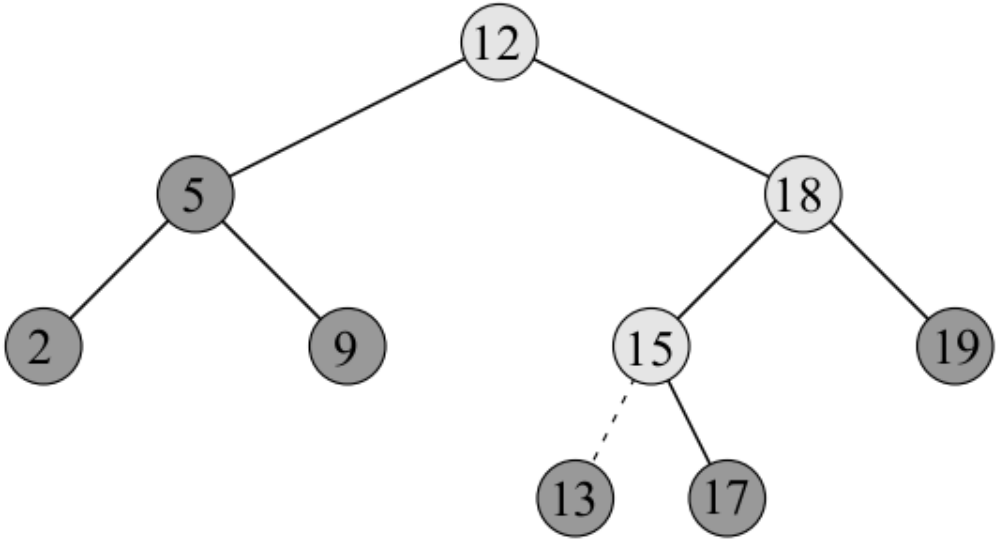

그림: Delete 13, 15, 18

Subtree substitution

- u is a subtree of T
- v is an independent tree
- substitute u with v $\longrightarrow$ modified tree T

$\longrightarrow$ Transplant

Transplant

- Time complexity? $O(1)$

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

그림: Transplant

Cases of delete

- z has no child
- z has one child
  - right child
  - left child
- z has two children

  $\longrightarrow$ make right child (y) has no left child

  $\longrightarrow$ replace z with y

z has only right child

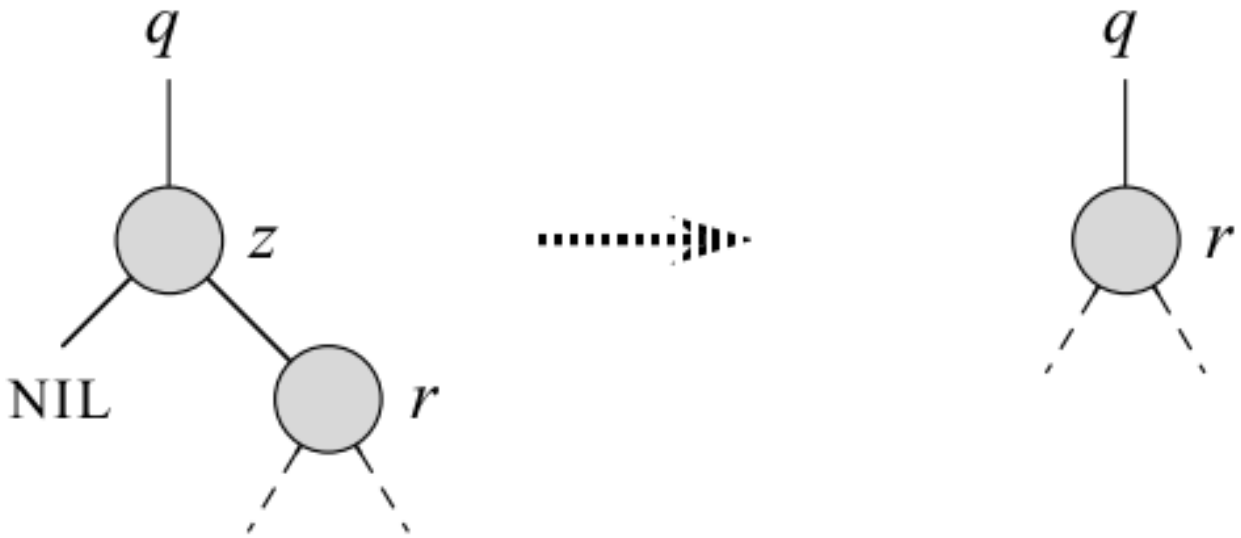

그림: One right child

z has only left child

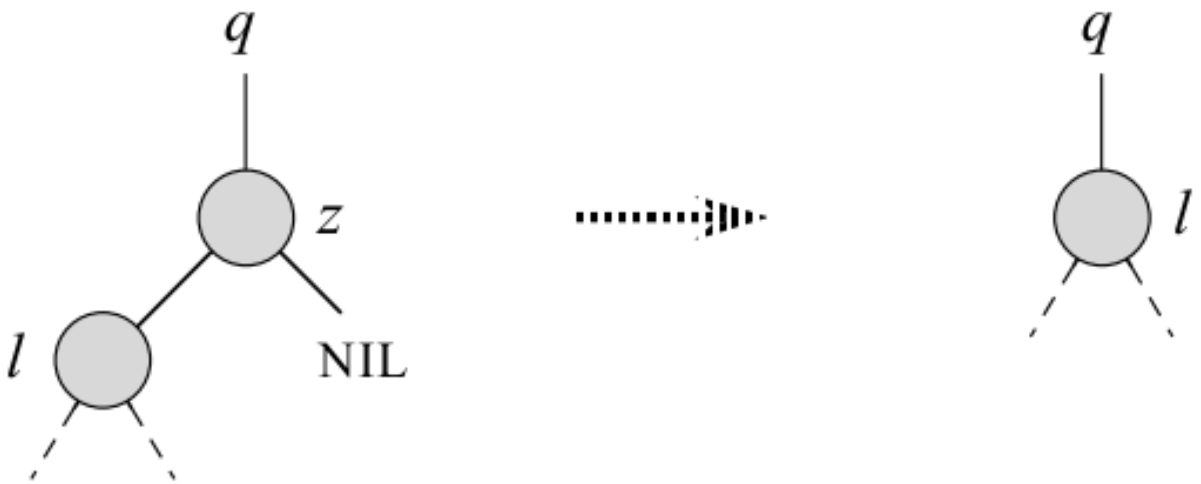

그림: One left child

z has two children but easy to handle

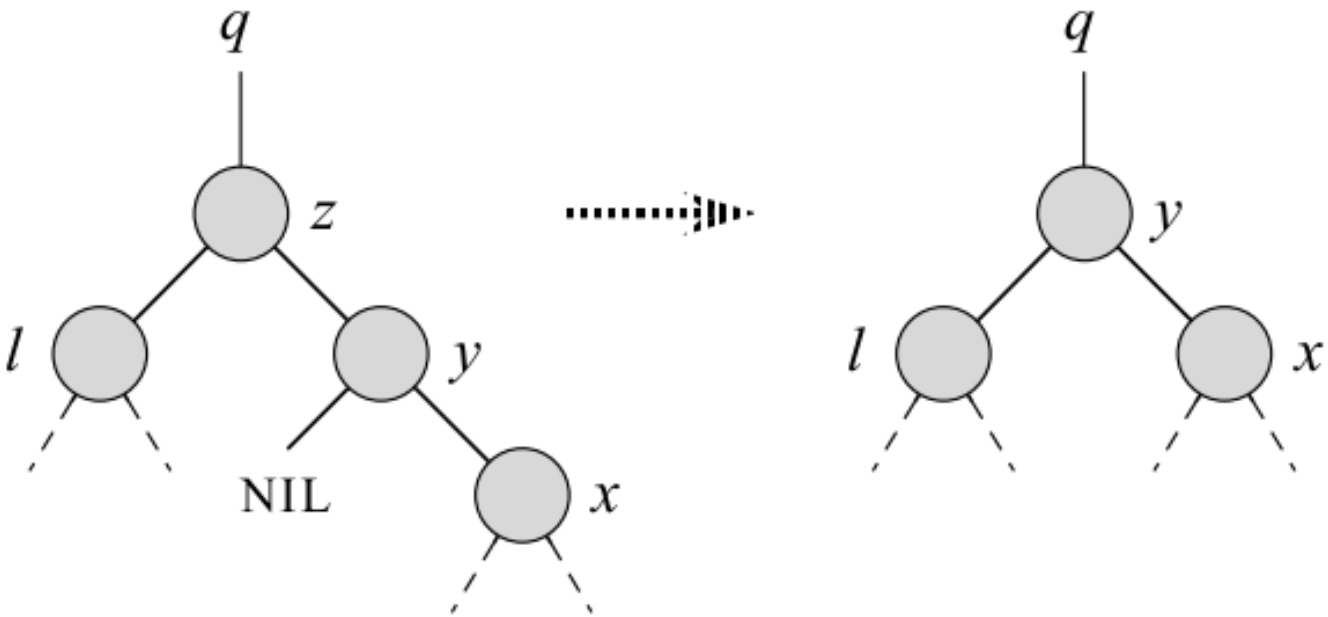

그림: Two children

General case

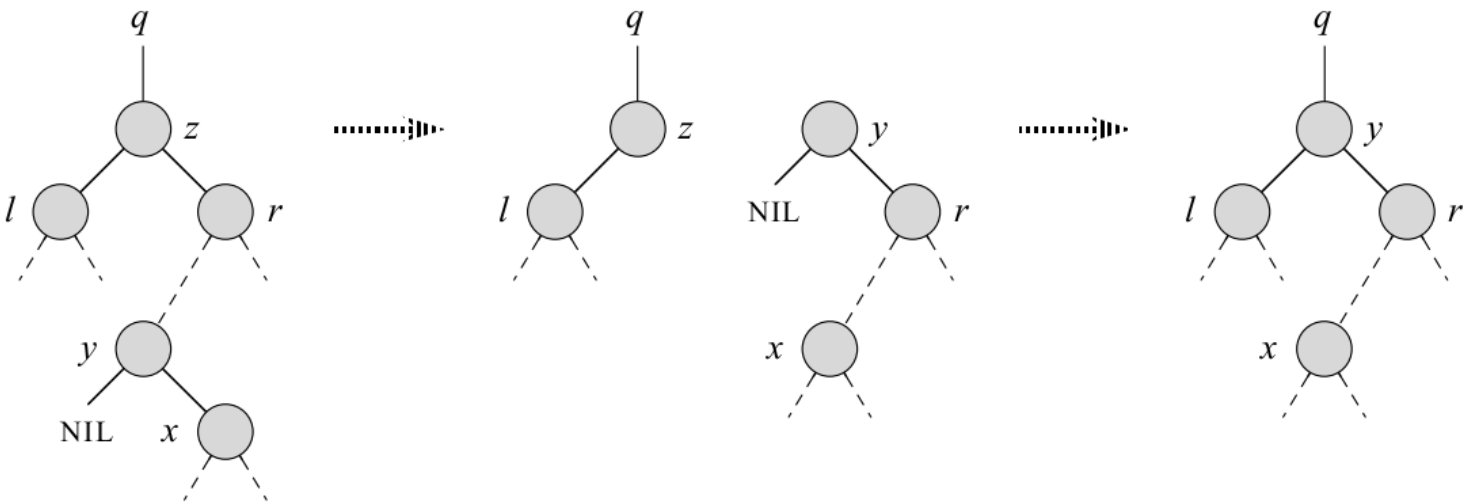

그림: General two children

Tree-delete

- Time complexity? $O(h)$

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

그림: Tree-delete

Homework

- Add parent node to `bst` sample code
- Implement insert and delete
- Write a simple test code
- Submit the source code

Wrap-up

- We reviewed the simple data structure doubly linked list
- Deletinig a node from a BST is a bit more complex than doubly linked list. But the basic idea is very similar
- Deleting binary search tree should preserve BST property
- We could delete a node from a BST with $O(h)$ where h is the height of the tree. To design the algorithm, we performed case by case analysis
- `Tree-delete()` uses `Transplant` which makes the algorithm easier to understand than without it