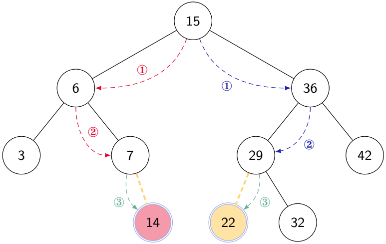
CPT108 Data Structures and Algorithms

Lecture 17

Trees

Insertion and Deletion

- Insert a new key into the binary search tree (BST)
  - `insert(22)`
  - `insert(14)`



TREE-INSERT(T, z)

$x = T.root$ // node being compare with z
$y =$ NIL // y will be parent of z
while $x \neq$ NIL
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
if $y ==$ NIL
    $T.root = z$ // tree T was empty
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$

- Observation
  - The new key is always inserted as a *new* leaf

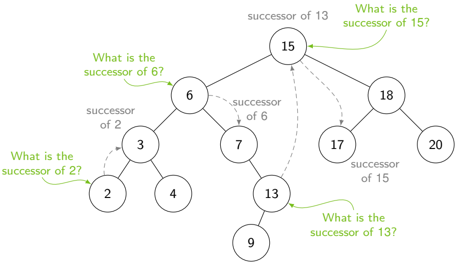
- When a node is deleted, we need to consider how we take care of the children of the deleted node
  - This has to be done such that the *property* of the binary search tree (BST) is maintained!

Binary search tree (BST)

Deletion (cont.)

Successor

- Given a BST, the successor of a node x is the node with the smallest key greater than $x.key$
- Or, in order word, it is the next node visited in an *inorder* tree walk (inorder traversal)
- Observation: A successor can have ***no*** children or only a ***right***-child

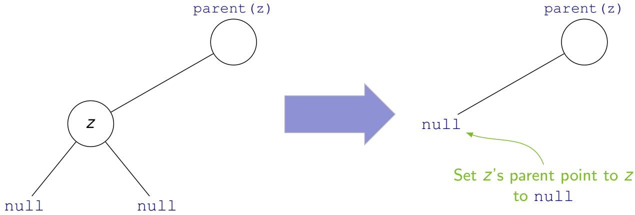



```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    // leftmost node in right subtree
    return TREE-MINIMUM(x.right)
else
    // find the lowest ancestor of x
    // whose left child is an ancestor of x
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
    return y
```

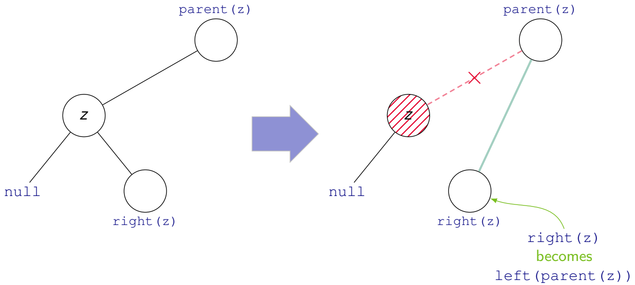
- What is the successor of 20?

Three cases for deletion

Case 1: Node *z* is a leaf



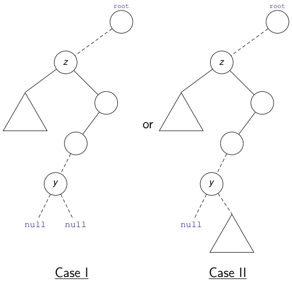
Case 2: Node *z* has exactly 1 (left or right) child



Modify appropriate `parent(z)` to point to
z's child (Parent adoption)

Case 3: Node z has 2 children

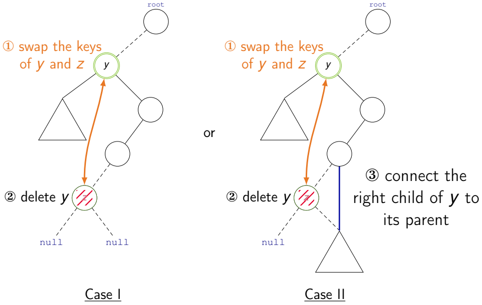
Step 1. Find successor y of z
(i.e., $y =$ `successor(z)`)



Success y of z will have no child
or only a right-child

Step 2. Swap the keys of z and y,
then delete node y (which now has value z!)
and connect its right-child of z to its parent

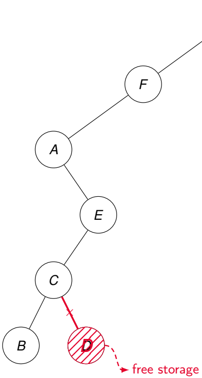
This *deletion* is either case I or case II



- If z is a node that we want to delete from a BST T
- Then, we have three cases:
  - If z has *no* children, then simply remove it by modifying its parent to replace z with `null` as its child
  - If z has just *one* child, then elevate that child to take z's position in the tree by modifying z's parent to replace z's child
  - If z has two children
    1. Find z's successor y — which must belong to z's right subtree
    2. Move y to take z's position in the tree
    3. The rest of z's original right subtree becomes y's new right subtree
    4. z's left subtree becomes y's new left subtree

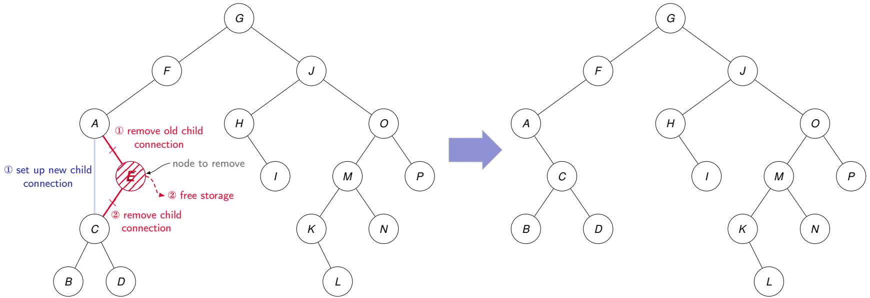
The node x is a leaf.

- `delete(`D`)`



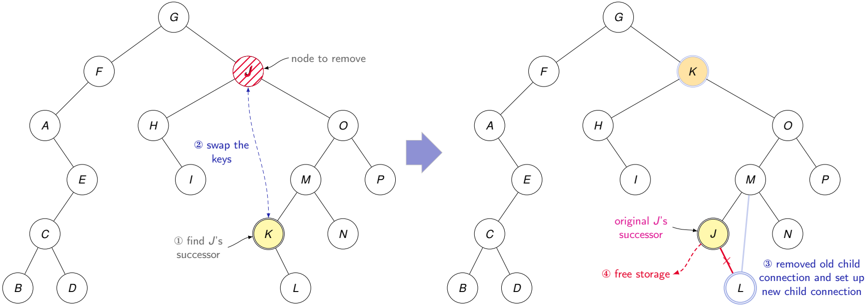
The node x has one child.

- `delete(`E`)`



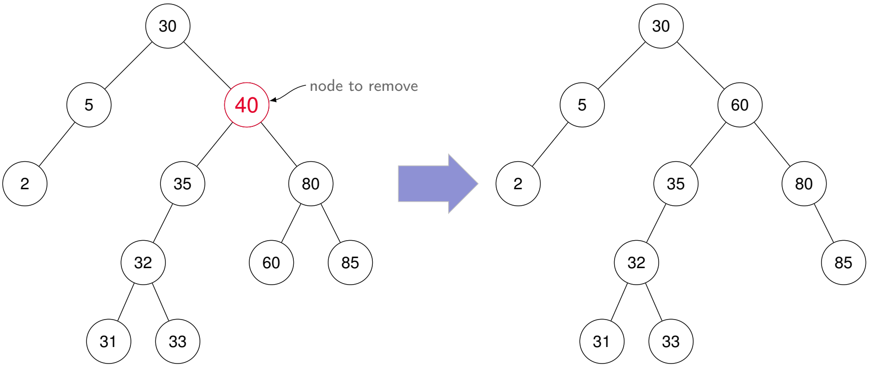
The node x has two children.

- `delete(`J`)`



The node *x* has two children.

- `delete(40)`



Remark

- There is also another approach of using the largest element in the left subtree, i.e., the predecessor, to replace the deleted node
  - $\Rightarrow$ refer to the reference book for details

Complexities

Operation	Complexity
`find(x)`	O(height of tree)
`findMin(x)`	O(height of tree)
`findMax(x)`	O(height of tree)
`insert(x)`	O(height of tree)
`delete(x)`	O(height of tree)
`traverse`	$O(n)$

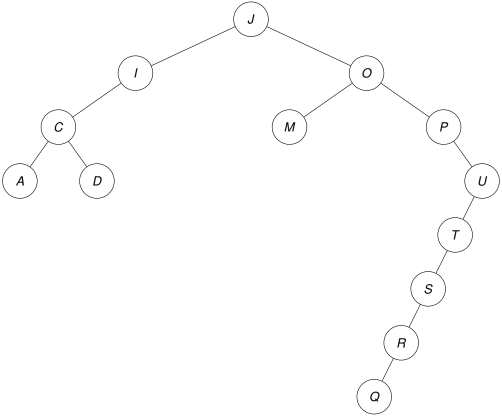
Problems with BSTs

- Problem
  - How can we predict the height of the tree?
- Many trees of different shapes can be composed of the same data
- How to control the tree shape?

Problems with BSTs (cont.)

Problem of Lopsidedness

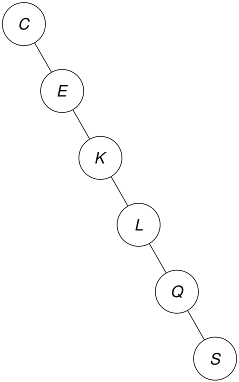
- Tree can be unbalanced
- Not all nodes have exactly 2 child nodes



Problems with BSTs (cont.)

Problem of Lopsidedness (cont.)

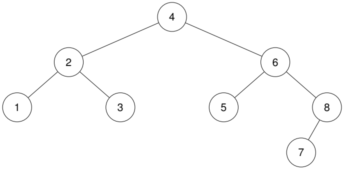
- Trees can be totally lopsided
- Suppose each node has a right child only
  - *Degenerates* into a *linked list*



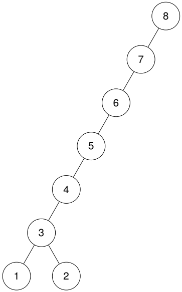
$\Rightarrow$ *Processing time* affected by the *shape* of the tree

Balanced vs Unbalanced Tree

- The trees below contain the same data



Tree$_1$: Balanced tree



Tree$_2$: Unbalanced tree

⇒ Which tree would you prefer to use?
And why?

How Fast is Sorting in BST?

- n elements (n is large) are to be sorted by first constructing a BST and then read them in inorder manner
- Bad case: the input is more or less sorted
  - A rather *linear* tree is constructed
  - Total steps in constructing a BST: $1 + 2 + \cdots + n = \dfrac{n(n+1)}{2} \sim n^2$
  - Total steps in traversing the tree: n

  $\Rightarrow$ Total: $O(n^2)$
- Best case: the BST is constructed in a *balanced* manner
  - Depth after adding i numbers: $\lg i$
  - Total steps in constructing a BST: $\lg 1 + \lg 2 + \cdots + \lg n < \lg n + \lg n + \cdots + \lg n = n \lg n$
  - Total steps in traversing the tree: n

  $\Rightarrow$ Total: $O(n \lg n)$ – much faster
- For any arbitrary input, one can indeed construct a rather balanced BST with some extra steps in insertion and deletion
  - E.g., an AVL tree (pp. 357–358, Cormen 2022)

Reading

- Chapter 12.3, Cormen (2022)