Binary Search Tree

Chapter 12 from textbook

Dynamic Sets

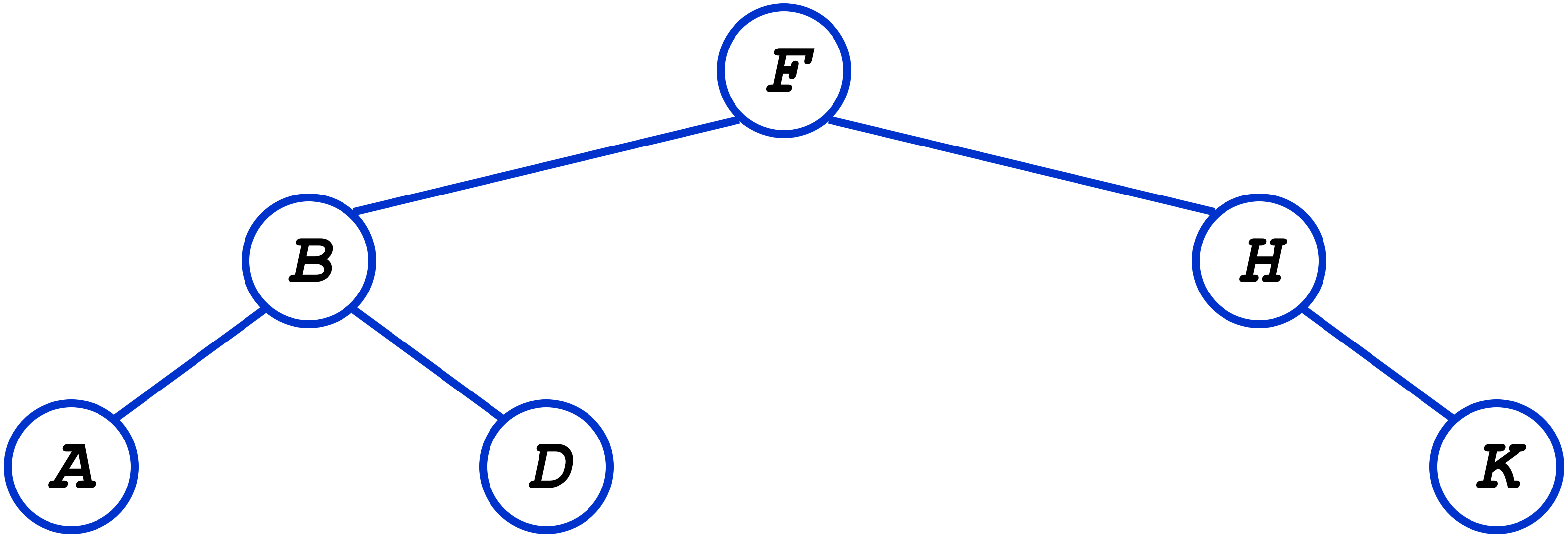
- Next 1-2 lectures we will focus on data structures rather than straight algorithms
- In particular, structures for *dynamic sets*
  - Elements have a *key* and *satellite data*
  - Dynamic sets support *queries* such as:
    - ***Search(S, k), Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x)***
  - They may also support *modifying operations* like:
    - ***Insert(S, x), Delete(S, x)***

Review: Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, eleements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

Review: Binary Search Trees

- BST property:
  key[leftSubtree(x)] ≤ key[x] ≤ key[rightSubtree(x)]
- Example:



Inorder Tree Walk

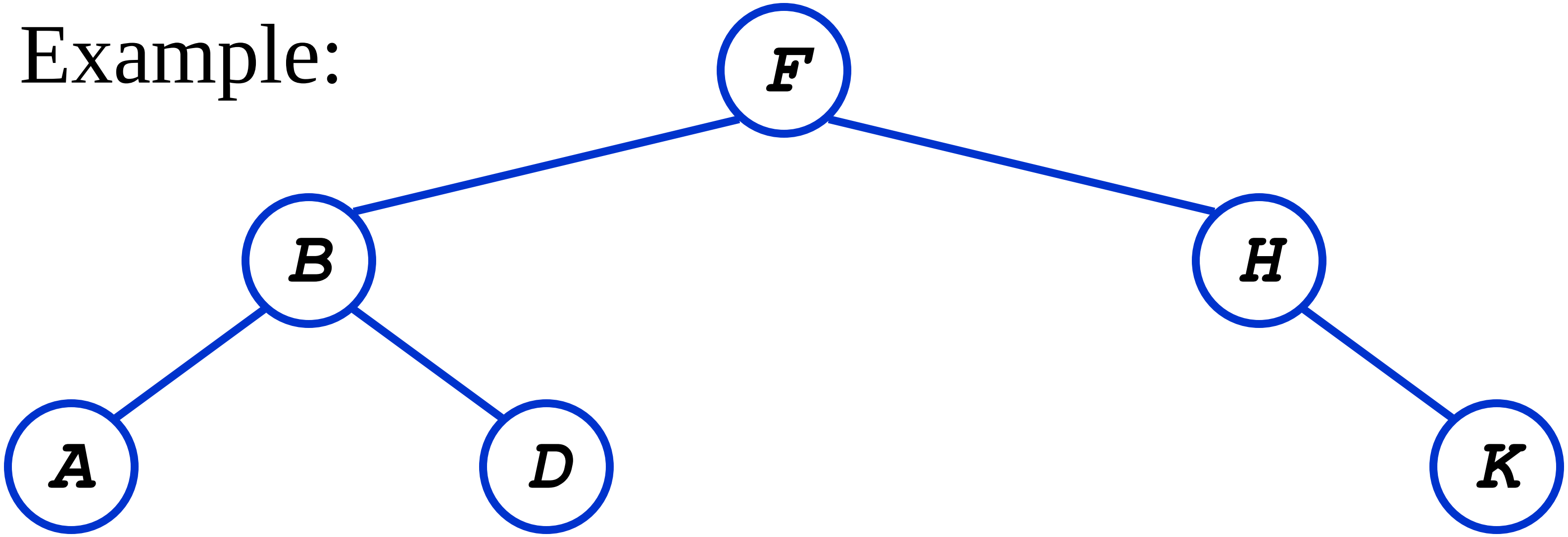
- *What does the following code do?*

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

- A: prints elements in sorted (increasing) order
- This is called an *inorder tree walk*
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Inorder Tree Walk

- Example:



- *How long will a tree walk take?*
- *Inorder walk prints in monotonically increasing order*

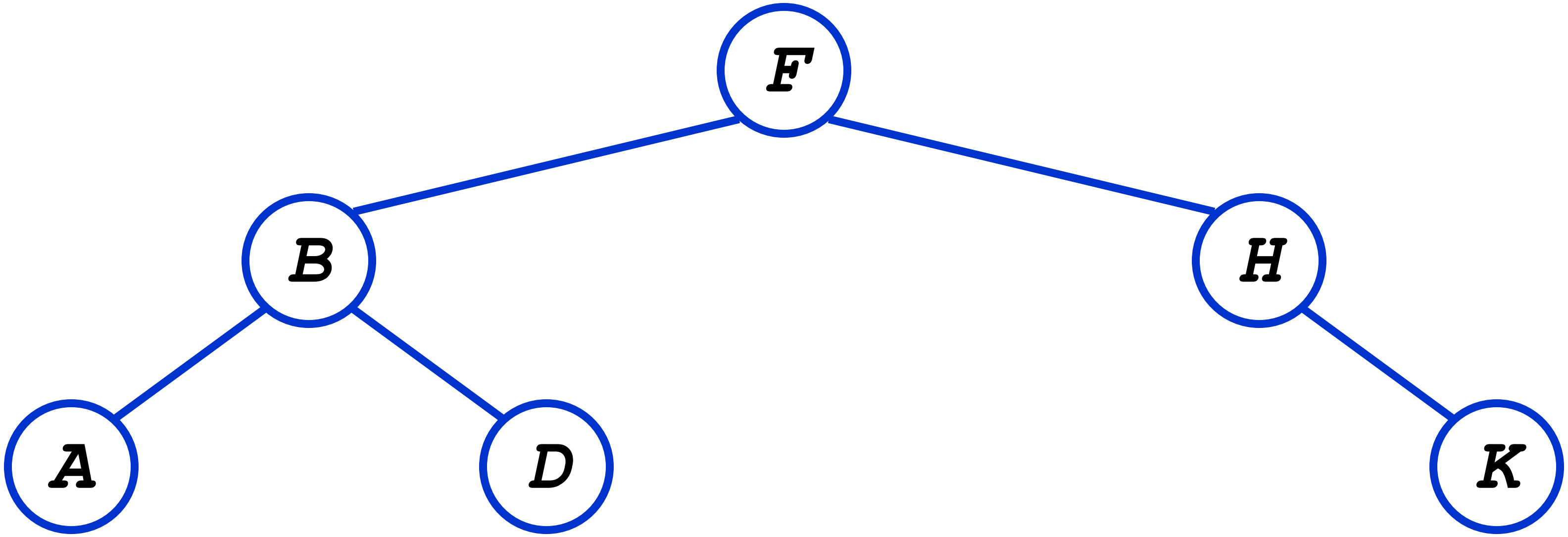
Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

BST Search: Example

- Search for *D* and *C*:



Operations on BSTs: Search

- Here's another function that does the same:

```
TreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

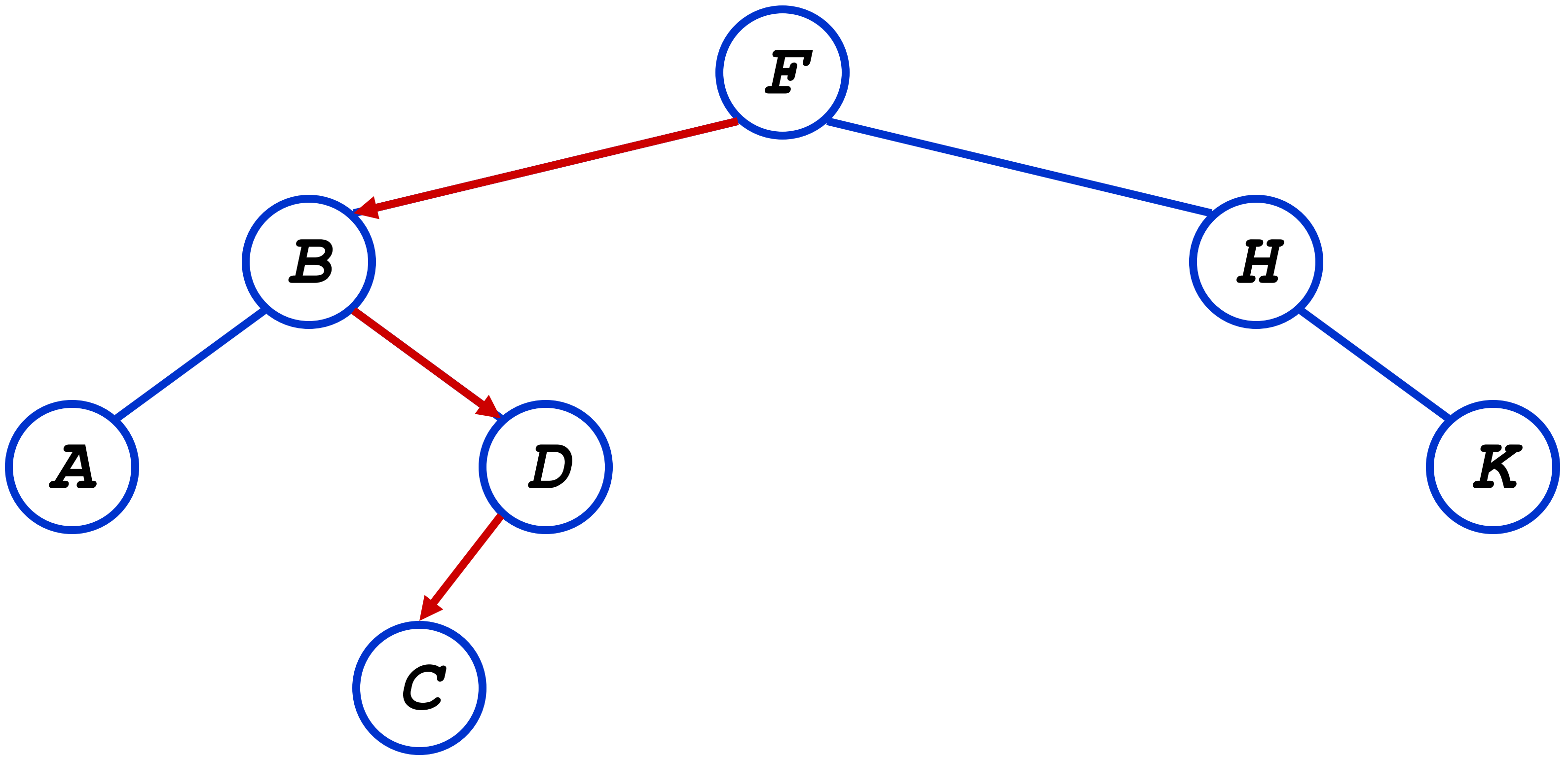
- *Which of these two functions is more efficient?*
  - Iterative version is more efficient on most computers

Operations of BSTs: Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)

BST Insert: Example

- Example: Insert *C*



BST Insert:

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

BST Search/Insert: Running Time

- *What is the running time of TreeSearch() or TreeInsert()?*
- A: $O(h)$, where h = height of tree
- *What is the height of a binary search tree?*
- A: worst case: $h = O(n)$ when tree is just a linear string of left or right children
  - We'll keep all analysis in terms of h for now
  - Later we'll see how to maintain $h = O(\lg n)$

Sorting With Binary Search Trees

- Informal code for sorting array A of length *n*:

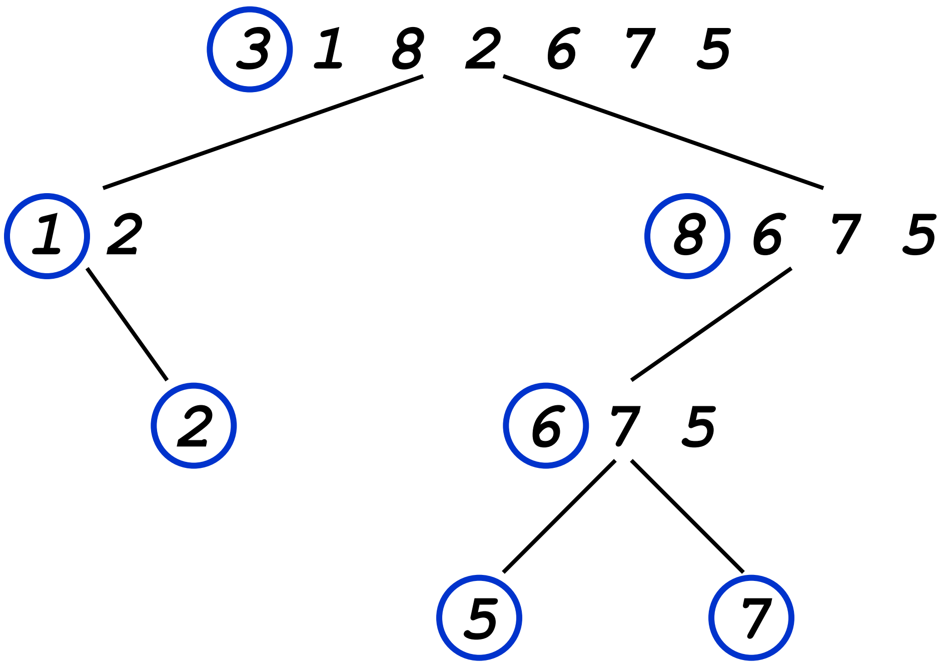
```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

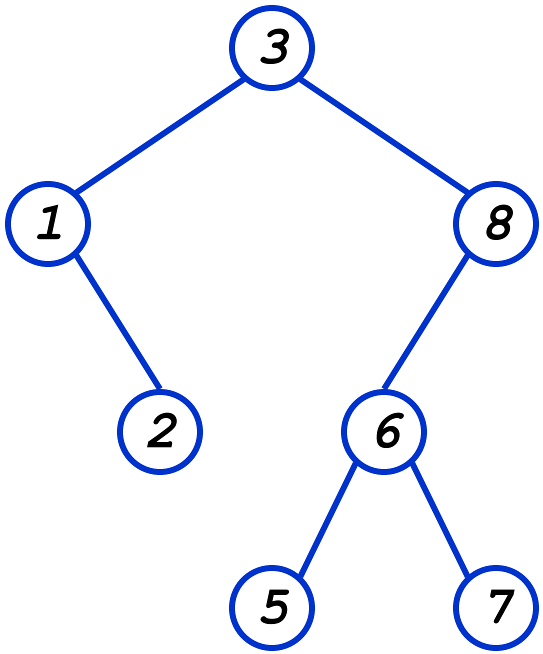
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs

- Average case analysis
  - It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```





Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as quicksort, different order!
    - Example: consider inserting 5

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTsort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTSort? Why?*
- A: quicksort
  - Better constants
  - Sorts in place
  - Doesn't need to build data structure

More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*
  - Insert
  - Minimum
  - Extract-Min

BST Operations: Minimum

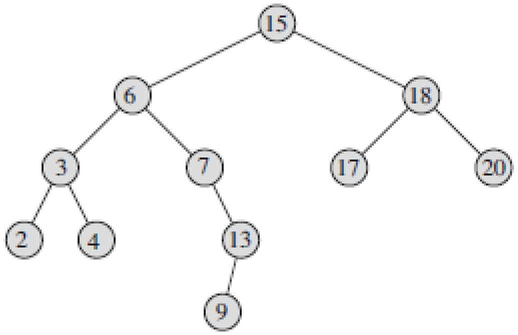
- *How can we implement a Minimum() query?*
- *What is the running time?*

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

BST Operations: Successor

- For deletion, we will need a Successor() operation



- *What is the successor of node 3? Node 15? Node 13?*
  - *Successor of a node x is the node with the smallest key greater than x.key*
- *What are the general rules for finding the successor of node x? (hint: two cases)*

BST Operations: Successor

- Two cases:
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- Predecessor: similar algorithm

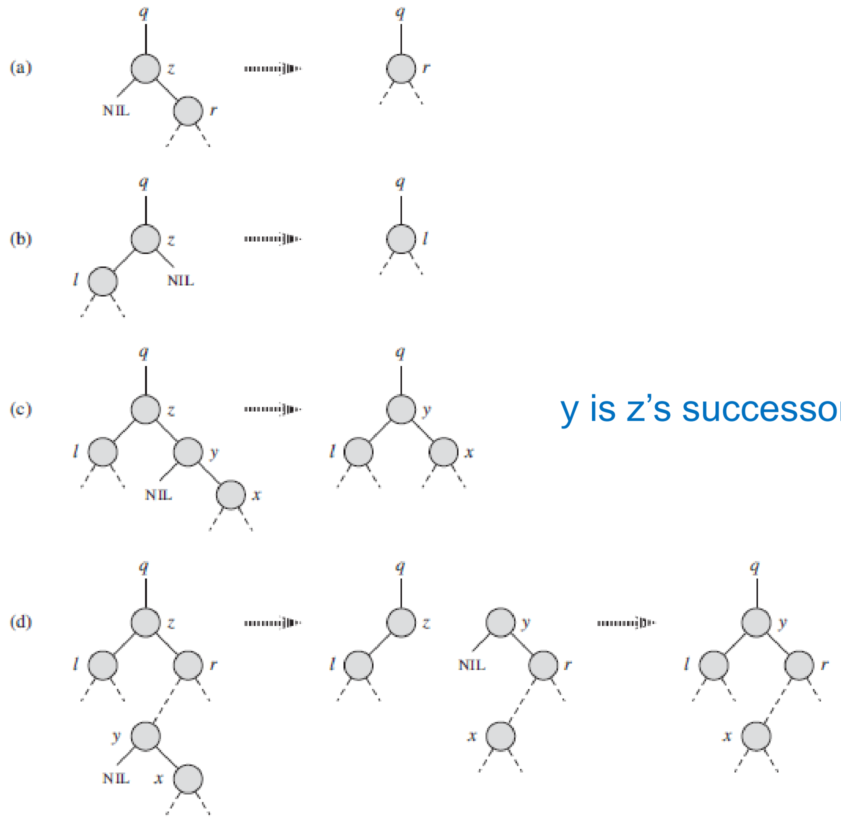
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

BST Operations: Delete

- Deletion is a bit tricky
- 3 cases:
  - z has no children:
    - Remove z
  - z has one child:
    - Elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child
  - z has two children:
    - Find z's successor y and swap z with y
    - Rest of z's original right subtree becomes y's new right subtree and z's left subtree becomes y's new left subtree
    - This can be tricky based on whether y is z's right child

BST Operations: Delete

- We want to delete node z



y is z's successor, x is y's right child

y is z's successor and rooted at subtree rooted at r, x is y's right child

We replace y by its own right child x, and set y to be r's parent. Then we set y to be q's child and parent of l

BST Operations: Delete

- Transplant replaces one subtree as a child of its parent with another subtree
  - When it replaces the subtree rooted at node u with a subtree rooted at node v
  - Node u's parent becomes node v's parent and u's parent end up having v as appropriate child

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

- Running time of tree delete is O(h), except Tree-Minimum everything is constant time

- Up next: guaranteeing a O(lg n) height tree
  - This ensures all operations on BST can be done in O(lg n) time