CSC263 Week 3

Announcements

- PS1 marks out, average: 90%
  - re-marking requests can be submitted on MarkUS.
- **Assignment 1** is out, due Feb 10
  - more challenging than PS! Start early!
  - work in groups of up to 4.

NOT EVERY GROUP PROJECT

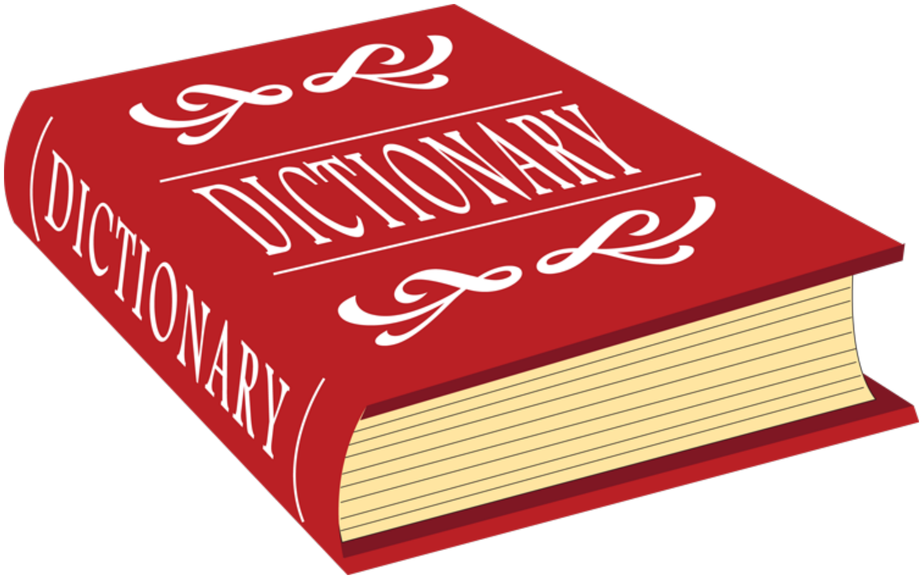
IN SCHOOL YOU HAVE EVER DONE

This week

- ➔ ADT: Dictionary

- ➔ Data structure:
  - ◆ Binary search tree (BST)
  - ◆ Balanced BST - AVL tree

Dictionary



What's stored:

- → words

Supported operations

- → Search for a word
- → Insert a word
- → Delete a word

Dictionary, more precisely

What's stored

- A set **S** where each node **x** has a field **x.key**
  (assumption: keys are distinct, unless o.w. specified)

Supported operations

- **Search(S, k)**: return **x** in **S**, s.t., **x.key = k**
  - return NIL if no such **x**
- **Insert(S, x):** insert node **x** into **S**
  - if already exists node **y** with same key , replace **y** with **x**
- **Delete(S, x):** delete a given **node** **x** from **S**

A thing to note: k is a key, x is a node.

More on Delete

Why Delete(S, **x**) instead of Delete(S, **k**)?

Delete(S, **k**) can be implemented by:

1. x = Search(S, k)
2. Delete(S, x)

We want separate different operations, i.e., each operation focuses on only one job.

Implement a Dictionary using **simple** data structures

40 -> 33 -> 18 -> 65 -> 24 -> 25

Unsorted (doubly) linked list

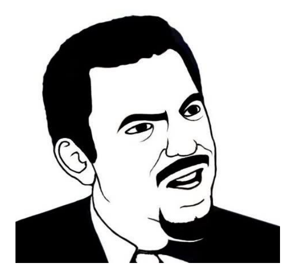
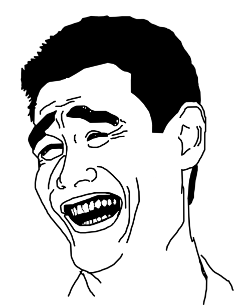
- **Search(S, k)**
  - **O(n)** worst case
  - go through the list to find the key
- **Insert(S, x)**
  - **O(n)** worst case
  - need to check if **x.key** is already in the list
- **Delete(S, x)**
  - **O(1)** worst case
  - Just delete, O(1) in a doubly linked list

Sorted array [18 , 24 , 25 , 33 , 40 , 65]

- **Search(S, k)**
  - **O(log n)** worst case
  - binary search!
- **Insert(S, x)**
  - **O(n)** worst case
  - insert at front, everything has to shift to back
- **Delete(S, x)**
  - **O(n)** worst case
  - Delete at front, everything has to shift to front

We can do better using smarter data structures, of course

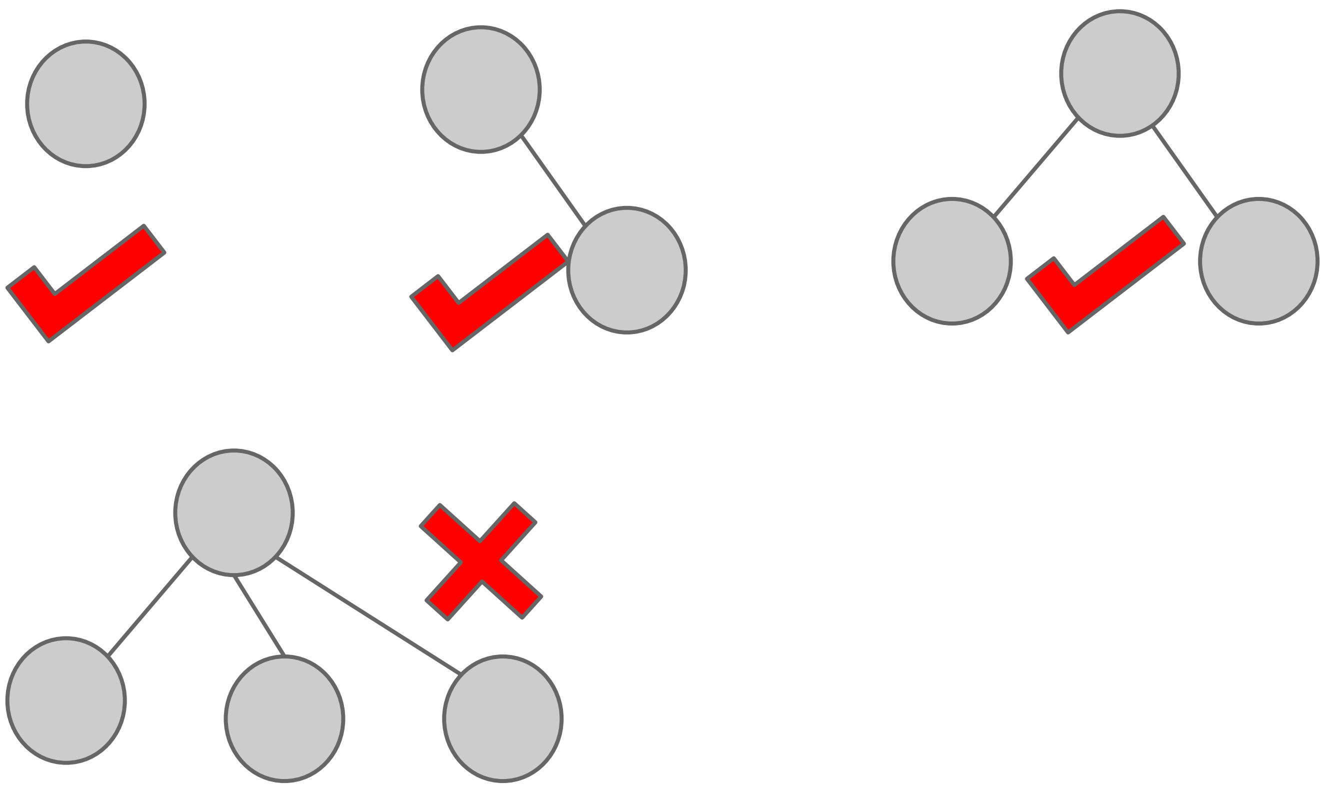
	unsorted list	sorted array	**BST**	**Balanced BST**
Search(S, k)	O(n)	O(log n)	**O(n)**	**O(log n)**
Insert(S, x)	O(n)	O(n)	**O(n)**	**O(log n)**
Delete(S, x)	O(1)	O(n)	**O(n)**	**O(log n)**

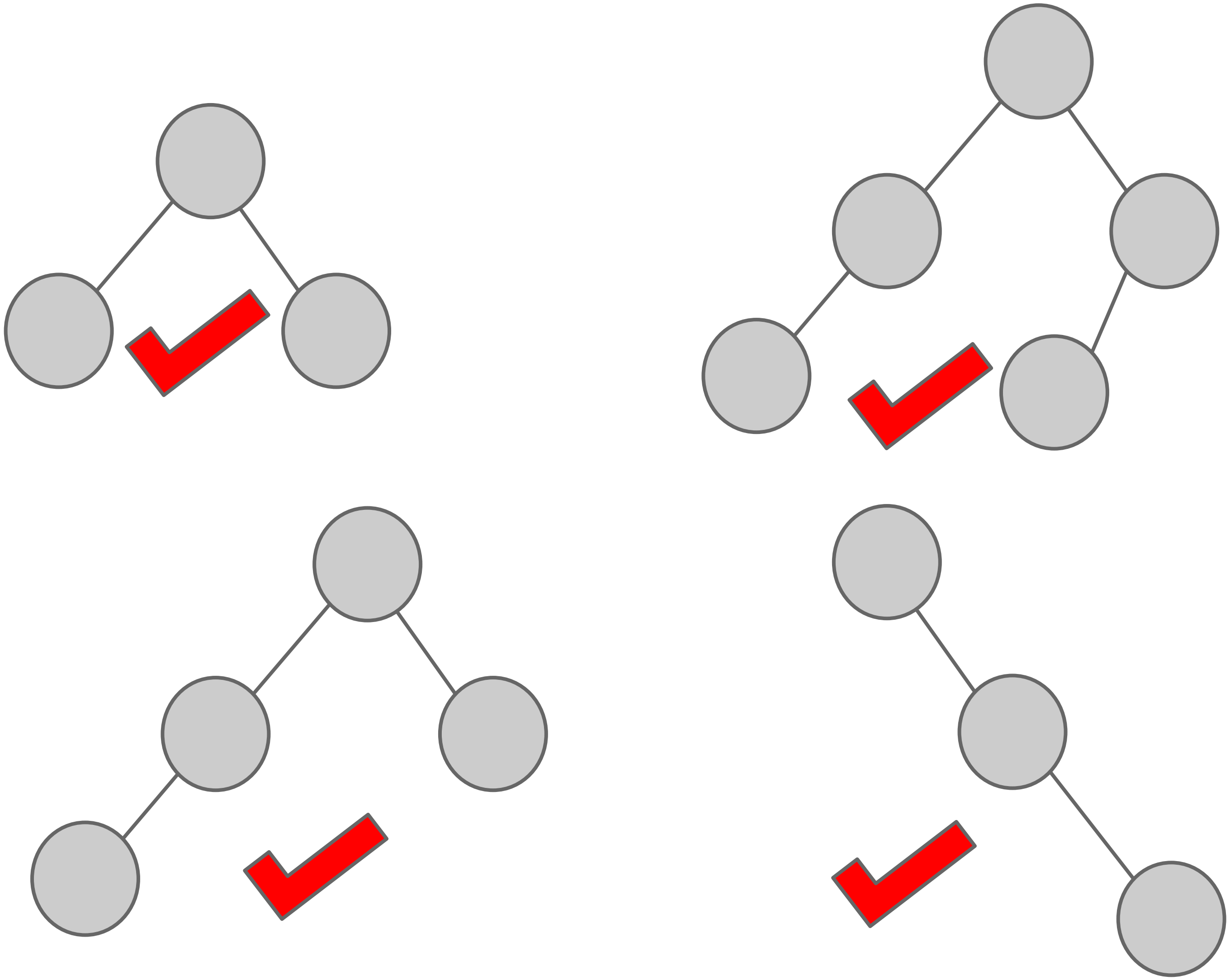
Binary Search Tree

It's a binary tree, like binary heap

Each node has **at most** 2 children

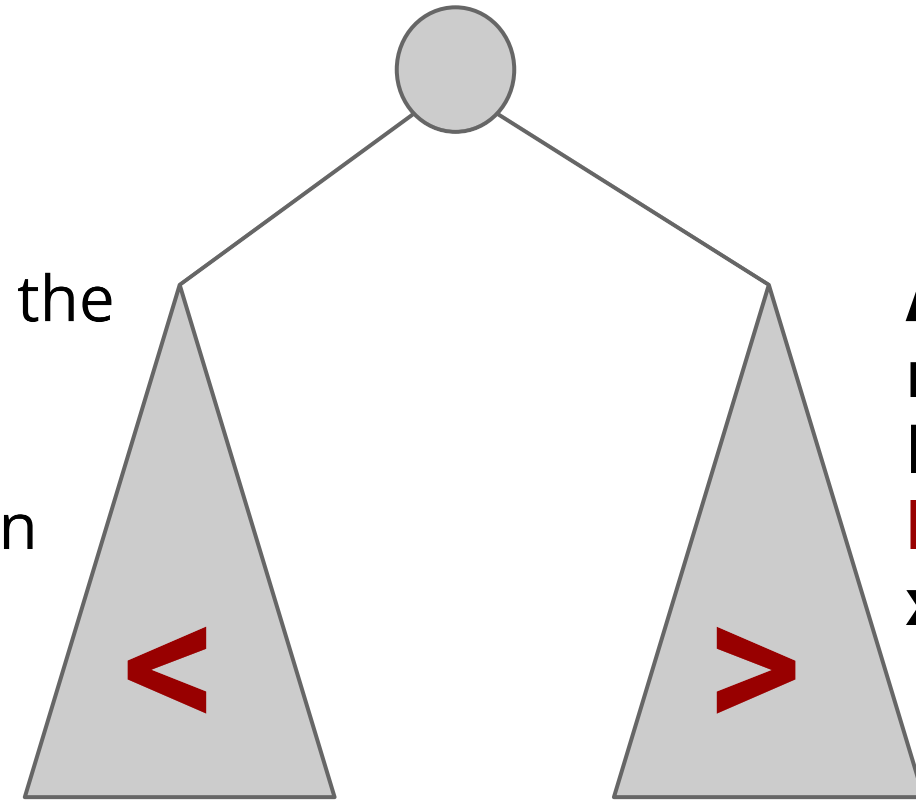


need NOT be nearly-complete, unlike binary heap



It has the BST property

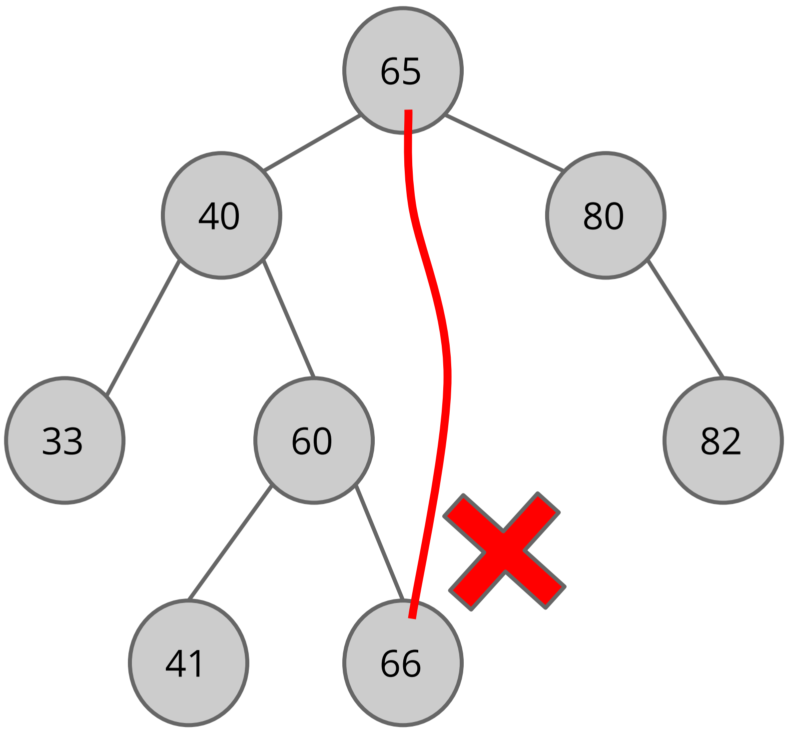
For **every** node **x** in the tree

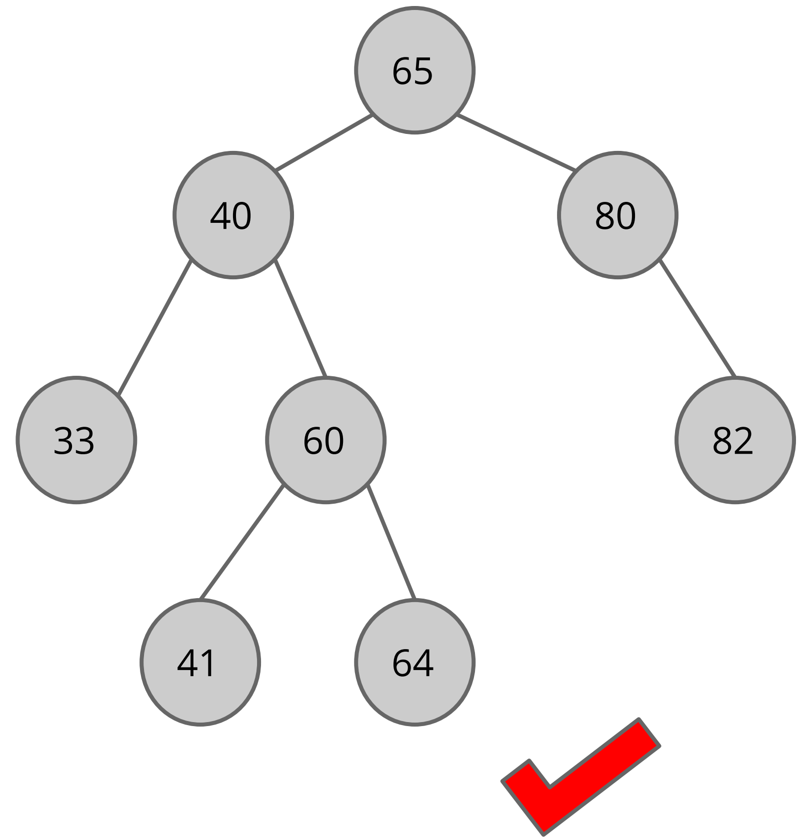


All nodes in the **left** subtree have keys **smaller** than **x.key**

All nodes in the **right** subtree have keys **larger** than **x.key**

BST or NOT?





Because of BST property, we can say that the keys in a BST are sorted.

CSC148 Quiz: How to obtain a sorted list from a BST?

Perform an inorder traversal.

We pass a BST to a function by passing its **root** node.

```
InorderTraversal(x):
# print all keys in BST rooted at x in ascending order

   if x ≠ NIL:
      InorderTraversal(x.left)
      print x.key
      InorderTraversal(x.right)
```

Worst case running time of InorderTraversal:
O(n), because visit each node exactly once.

Operations on a BST

First, information at each node x

- ➔ x.key: the key
- ➔ x.left: the left child (node)
- ➔ x.right: the right child (node)
- ➔ x.p: the parent (node)

Operations on a BST

read-only operations

- ➔ TreeSearch(root, k)
- ➔ TreeMinimum(x) / TreeMaximum(x)
- ➔ Successor(x) / Predecessor(x)

modifying operations

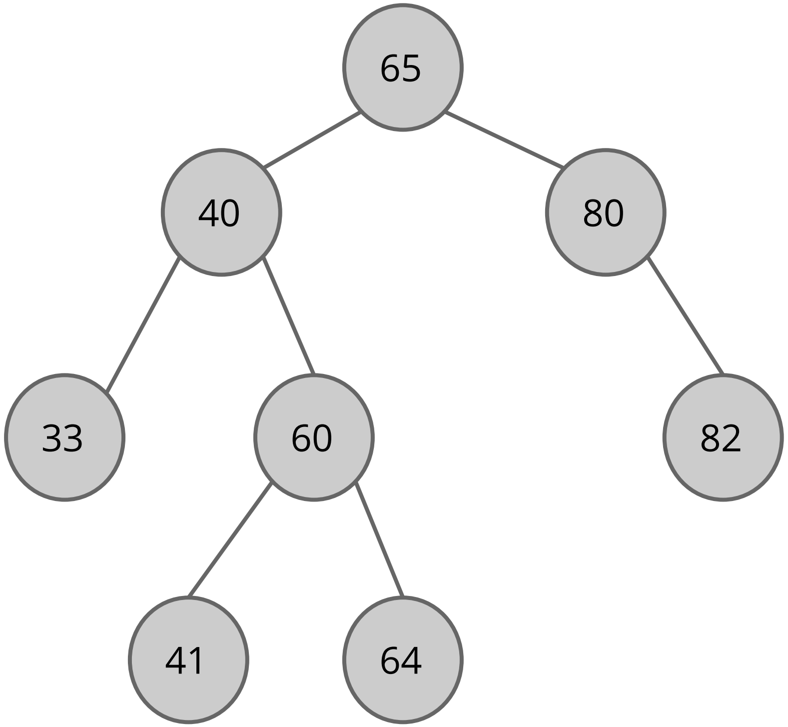
- ➔ TreeInsert(root, x)
- ➔ TreeDelete(root, x)

TreeSearch(root, k)

Search the BST rooted at root, return the node with key k; return NIL if not exist.

TreeSearch(root, k)

- ➔ start from root
- ➔ if **k** is **smaller** than the key of the current node, go **left**
- ➔ if **k** is **larger** than the key of the current node, go **right**
- ➔ if equal, **found**
- ➔ if going to **NIL**, **not found**



TreeSearch(root, k): Pseudo-code

```
TreeSearch(root, k):

   if root = NIL or k = root.key:
      return root
   if k < root.key:
      return TreeSearch(root.left, k)
   else:
      return TreeSearch(root.right, k)
```

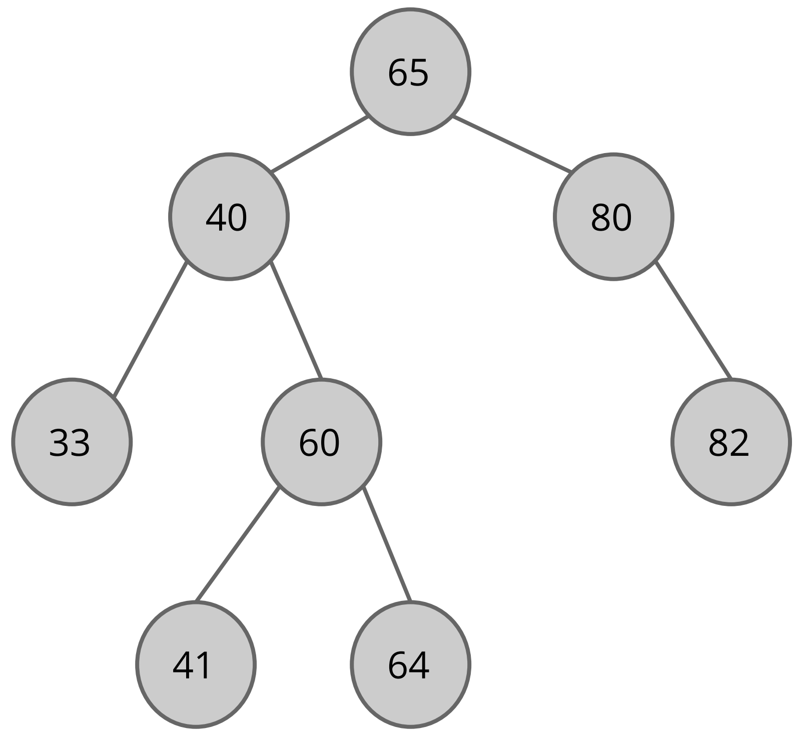
Worst case running time:
O(h), height of tree, going at most from root to leaf

TreeMinimum(x)

Return the node with the minimum key of the tree rooted at x

TreeMinimum(x)

- ➔ start from root
- ➔ keep going to the left, until cannot go anymore
- ➔ return that final node



TreeMinimum(x): pseudo-code

```
TreeMinimum(x):

    while x.left ≠ NIL:
        x ← x.left
    return x
```

Worst case running time:
O(h), height of tree, going at most from root to leaf

TreeMaximum(x) is exactly the same, except that it goes to the right instead of to the left.

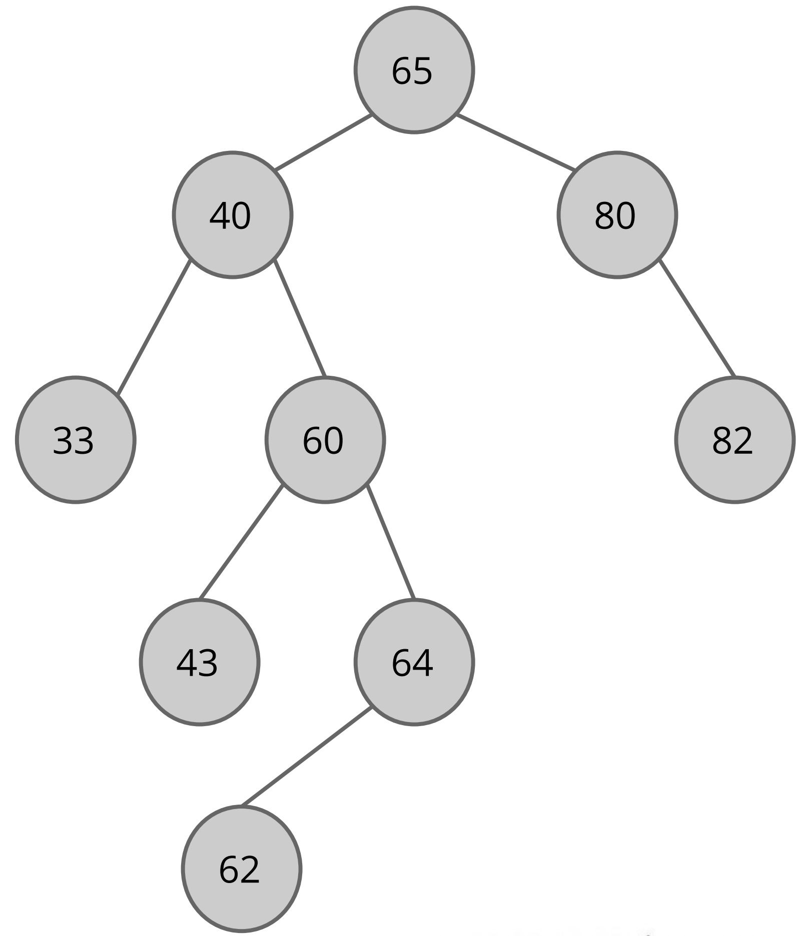
Successor(x)

Find the node which is the successor of x in the sorted list obtained by inorder traversal

or, node with the smallest key larger than x

Successor(x)

- The successor of 33 is…
  - 40
- The successor of 40 is…
  - 43
- The successor of 64 is…
  - 65
- The successor of 65 is …
  - 80



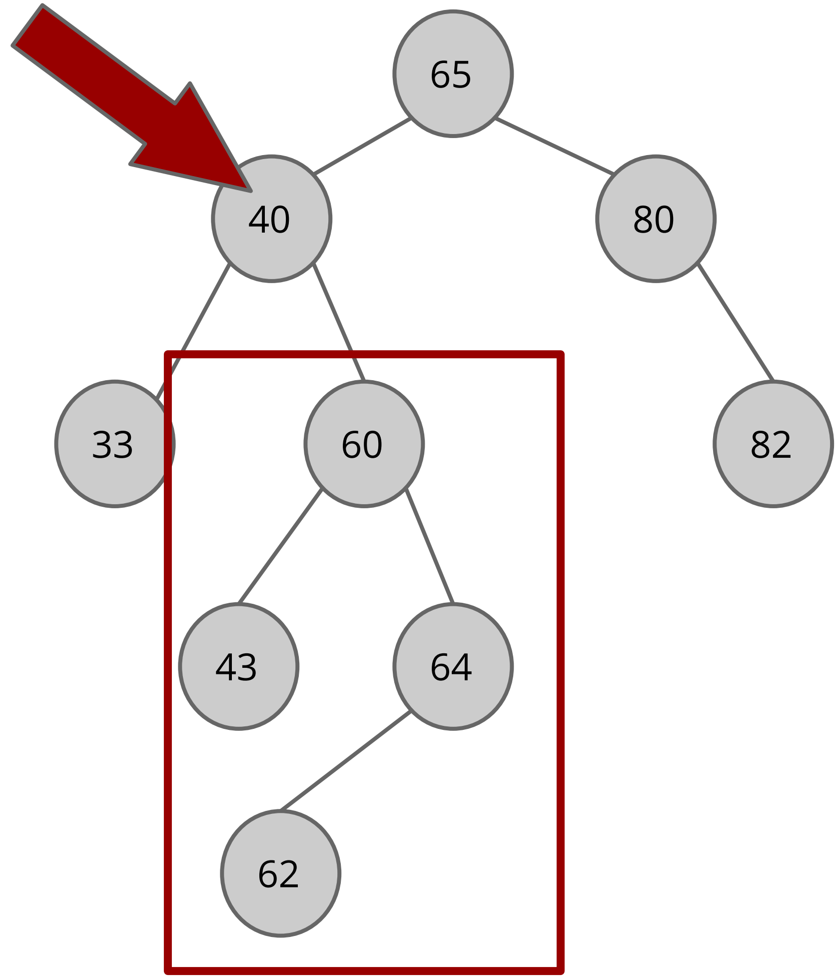
Successor(x):
Organize into two cases

- ➔ x has a right child
- ➔ x does not have a right child

x has a right child

Successor(x) must be in x’s **right subtree** (the nodes **right after x** in the inorder traversal)

It’s the **minimum** of x’s right subtree, i.e., TreeMinimum(x.right)



The first (smallest) node among what’s right after x.

x does not have a right child

Consider the **inorder traversal**

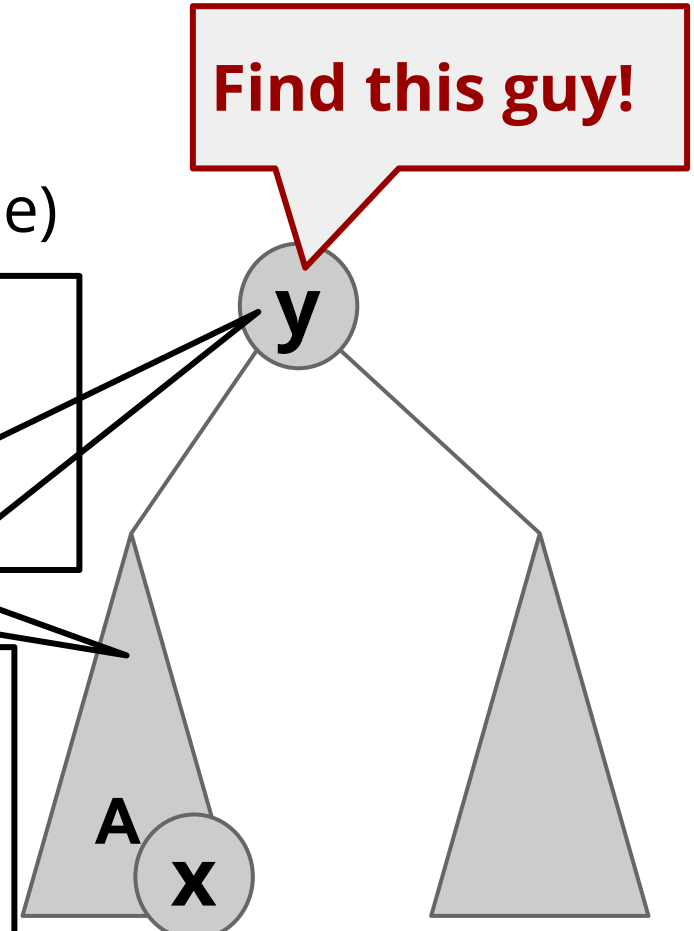
(left subtree -> root -> right subtree)

x is the **last one** visited in some **left subtree A**

(because no right child)

The successor **y** of x is the **lowest ancestor** of **x** whose **left subtree** contains **x**

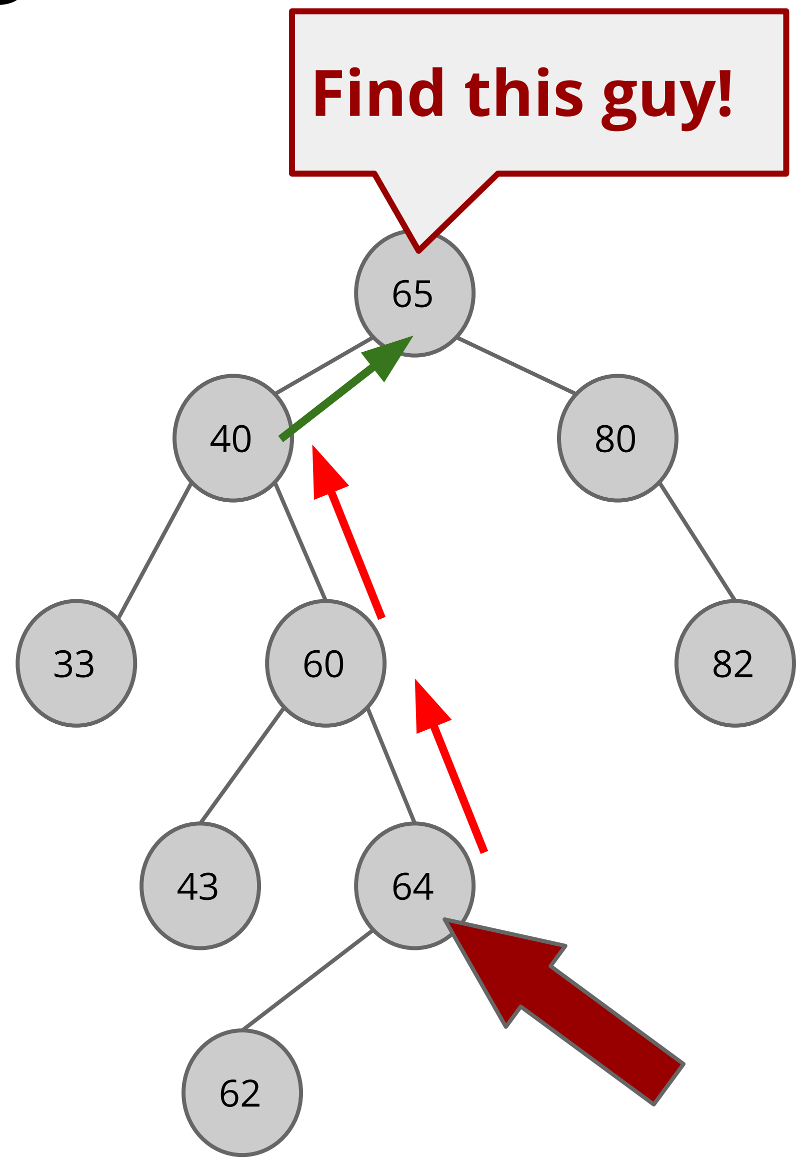
(**y** is visited right after finishing subtree **A** in inorder traversal)



x does not have a right child

How to find:

- ➔ go up to **x.p**
- ➔ if **x** is a **right** child of x.p, keep going up
- ➔ if x is a **left** child of **x.p**, stop, **x.p** is **the guy**!



Summarize the two cases of Successor(x)

- ➔ If x has a right child
  - ◆ return TreeMinimum(x.right)

- ➔ If x does not have a right child
  - ◆ keep going up to x.p while x is a right child, stop when x is a left child, then return x.p
  - ◆ if already gone up to the root and still not finding it, return NIL.

Successor(x): pseudo-code

```
Successor(x):
   if x.right ≠ NIL:
      return TreeMinimum(x.right)
   y ← x.p
   while y ≠ NIL and x = y.right: #x is right child
      x = y
      y = y.p # keep going up
   return y
```

Worst case running time

O(h), Case 1: TreeMin is O(log n); Case 2: at most leaf to root

Predecessor(x) works symmetrically the same way as Successor(x)

CSC263 Week 3

Thursday

Annoucement

- Problem Set 3 out

NEW feature! Exclusive for L0301!

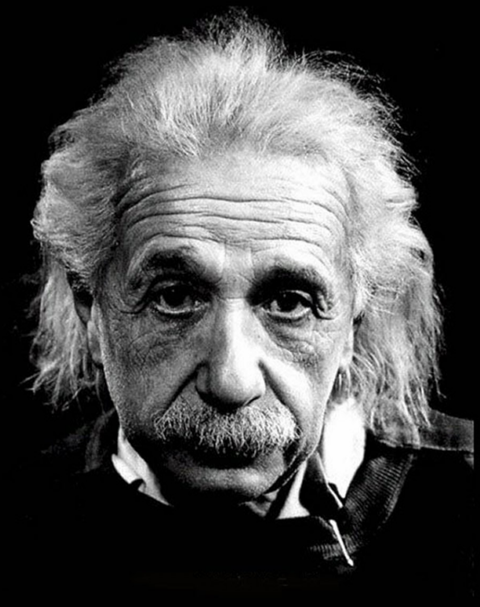
A weekly reflection & feedback system

2 minutes per week, let us know how things are going:

http://goo.gl/forms/S9yie3597B

Anonymous, short, topic-specific and potentially hugely helpful for improving learning experience.

Bonus: "**263 tips of the week**" shown upon form submission, updated **every Thursday** night.



Learn from yesterday, live for
today, hope for tomorrow.
The important thing is to
tell people how you feel,
once every week.

Recap of Tuesday

ADT: **Dictionary**

Data structure: **BST**

- ➔ read-only operations
  - ◆ TreeSearch(root, k)
  - ◆ TreeMinimum(x) / TreeMaximum(x)
  - ◆ Successor(x) / Predecessor(x)
- ➔ modifying operations
  - ◆ TreeInsert(root, x)
  - ◆ TreeDelete(root, x)

TreeInsert(root, x)

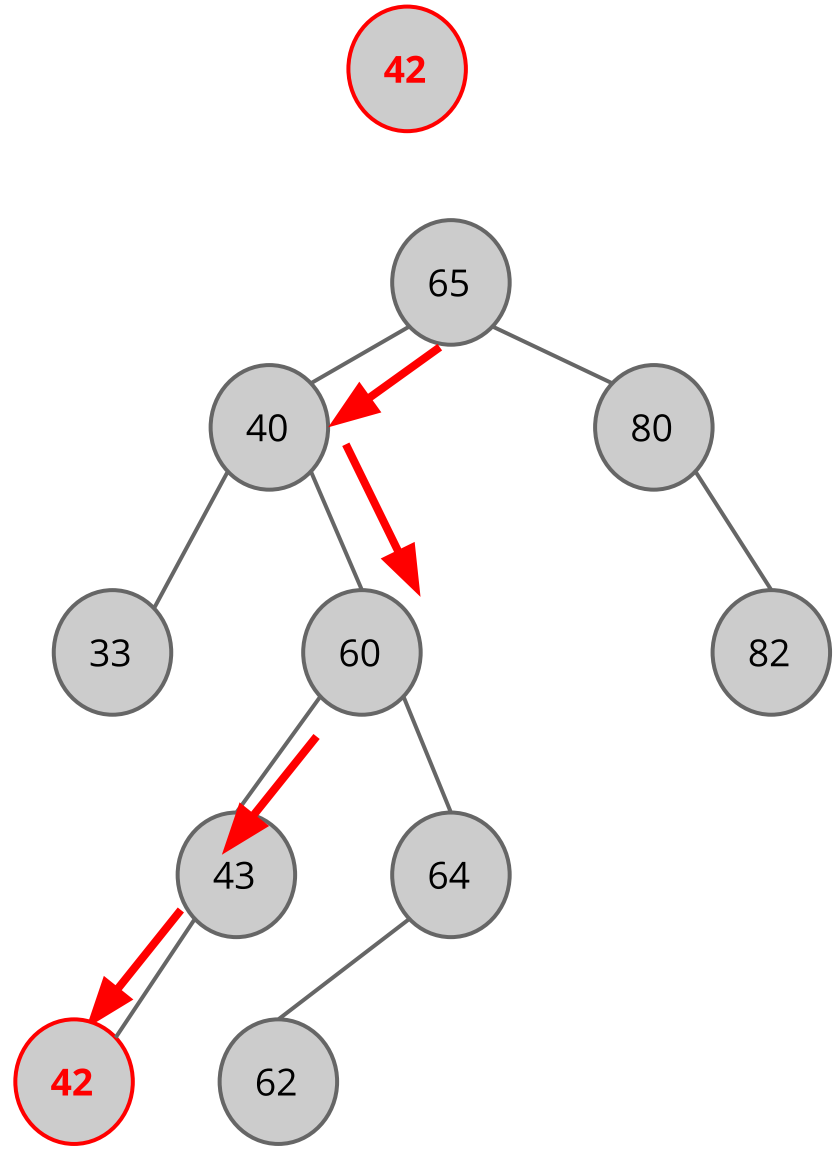
Insert node x into the BST rooted at root
return the new root of the modified tree
if exists y, s.t. y.key = x.key, replace y with x

TreeInsert(root, x)

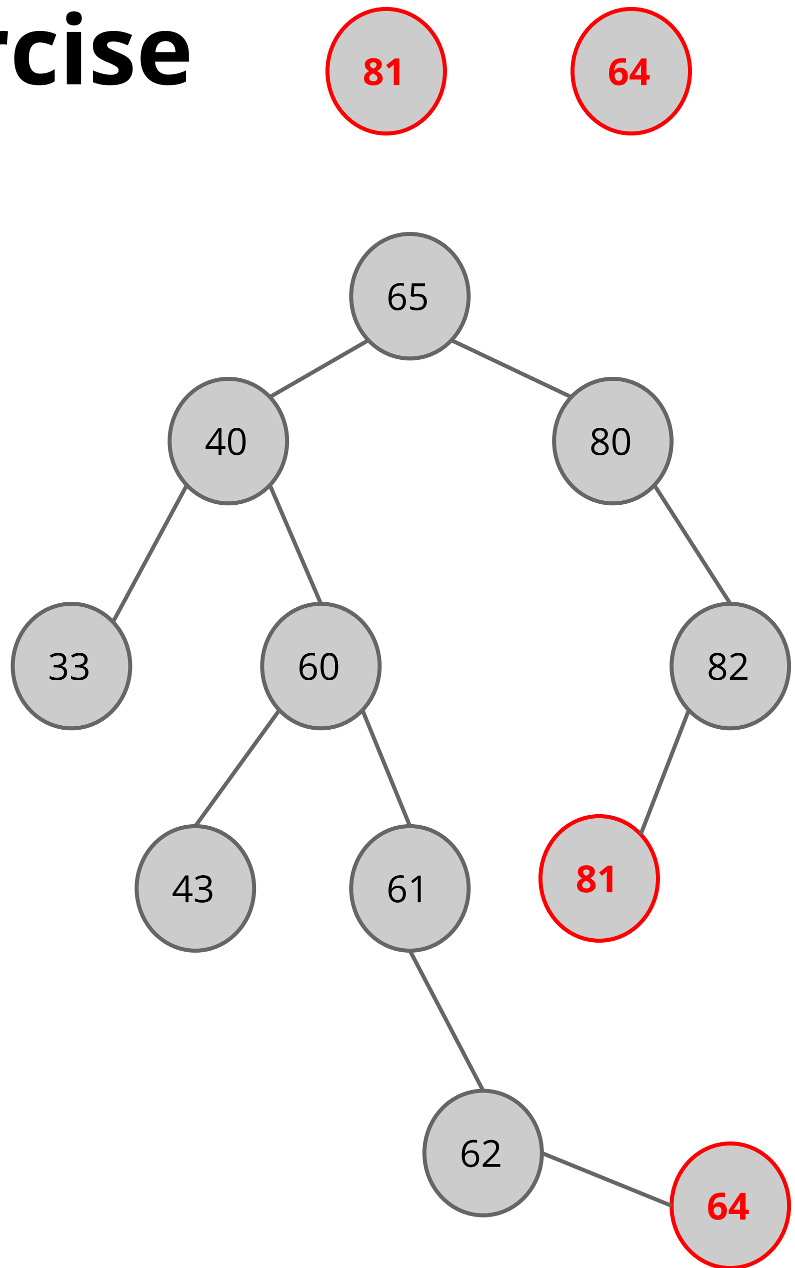
Go down, left and right like what we do in TreeSearch

When next position is NIL, insert there

If find equal key, replace the node



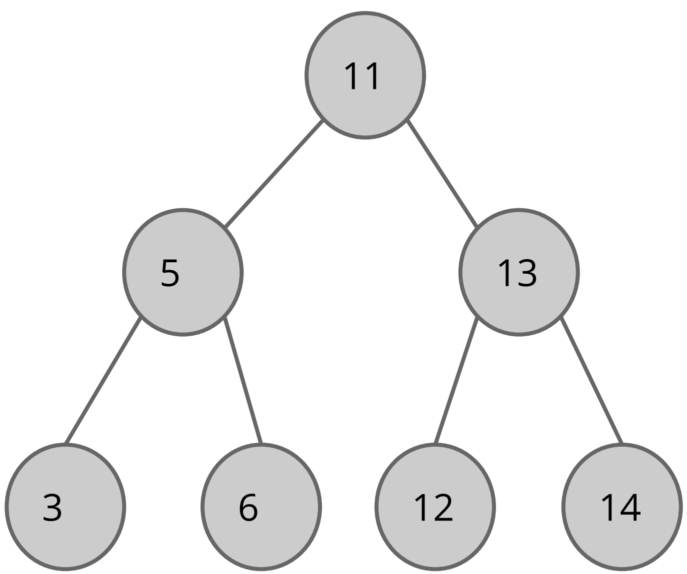
Exercise

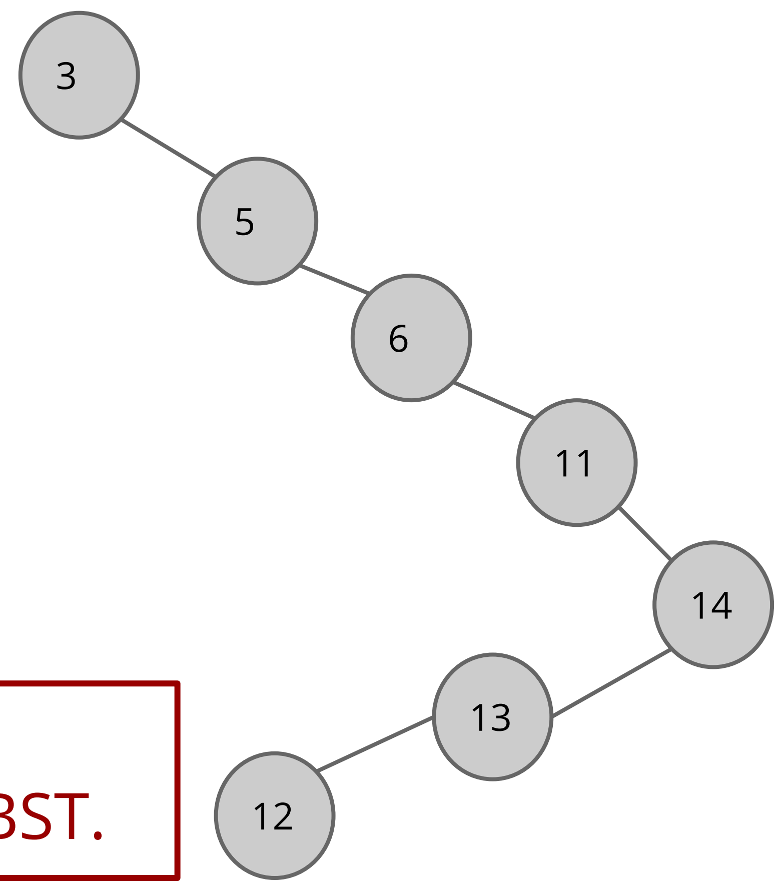


Ex 2: Insert sequence into an empty tree

Insert sequence 1:
11, 5, 13, 12, 6, 3, 14

Insert sequence 2:
3, 5, 6, 11, 14, 13, 12





Different insert sequences result in different "shapes" (heights) of the BST.

TreeInsert(root, x): Pseudo-code

```
TreeInsert(root, x):
#  insert and return the new root
    if root = NIL:
        root ← x
    elif x.key < root.key:
        root.left ← TreeInsert(root.left, x)
    elif x.key > root.key:
        root.right ← TreeInsert(root.right, x)
    else # x.key = root.key:
        replace root with x # update x.left, x.right
    return root
```

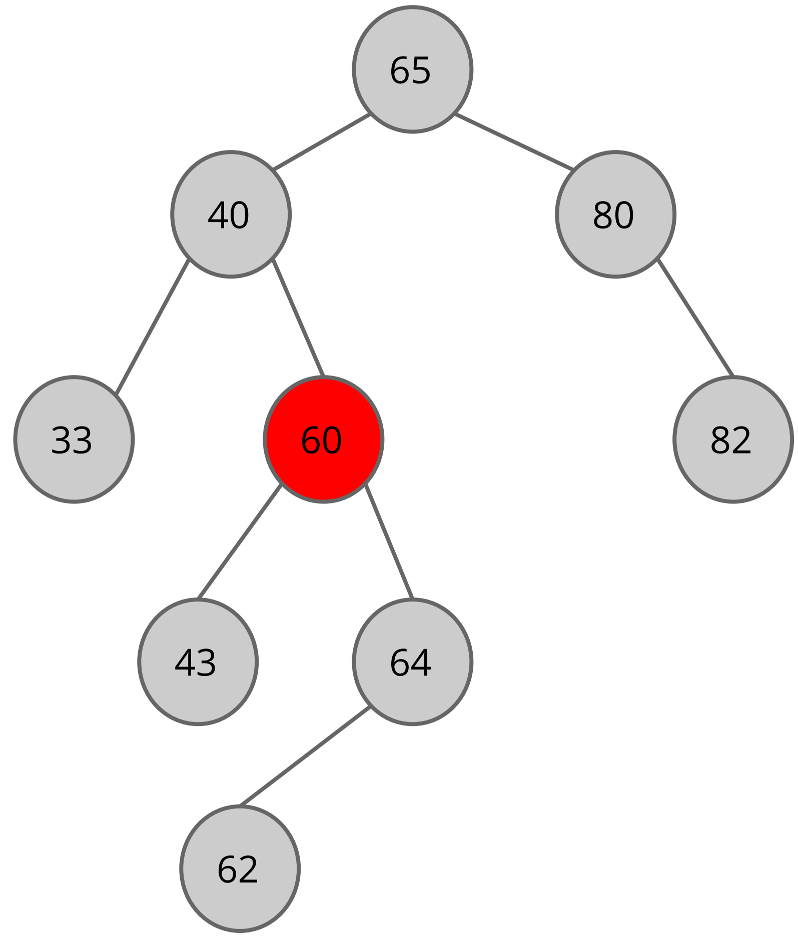
Worst case running time: **O(h)**

TreeDelete(root, x)

Delete node x from BST rooted at root
while maintaining BST property,
return the new root of the modified tree

What’s tricky about deletion?

Tree might be **disconnected** after deleting a node, need to **connect** them back together, while maintaining the **BST property**.



Delete(root, x): Organize into 3 cases

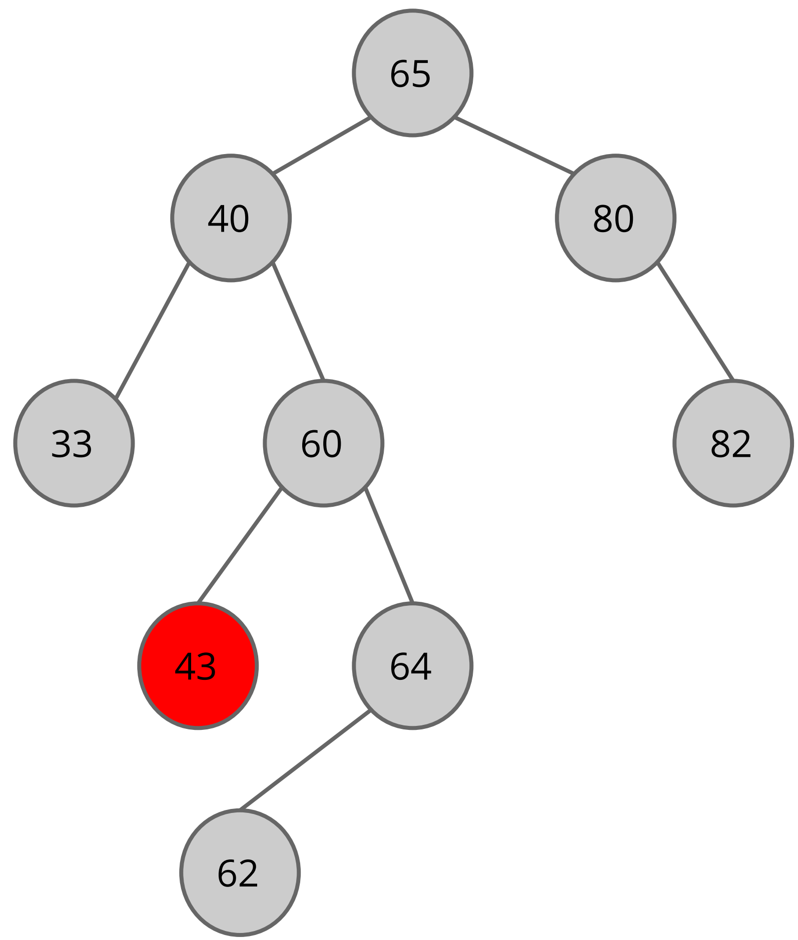
Case 1: **x** has **no** child — **Easy**

Case 2: **x** has **one** child — **Easy**

Case 3: **x** has **two** children — **less easy**

Case 1: x has no child

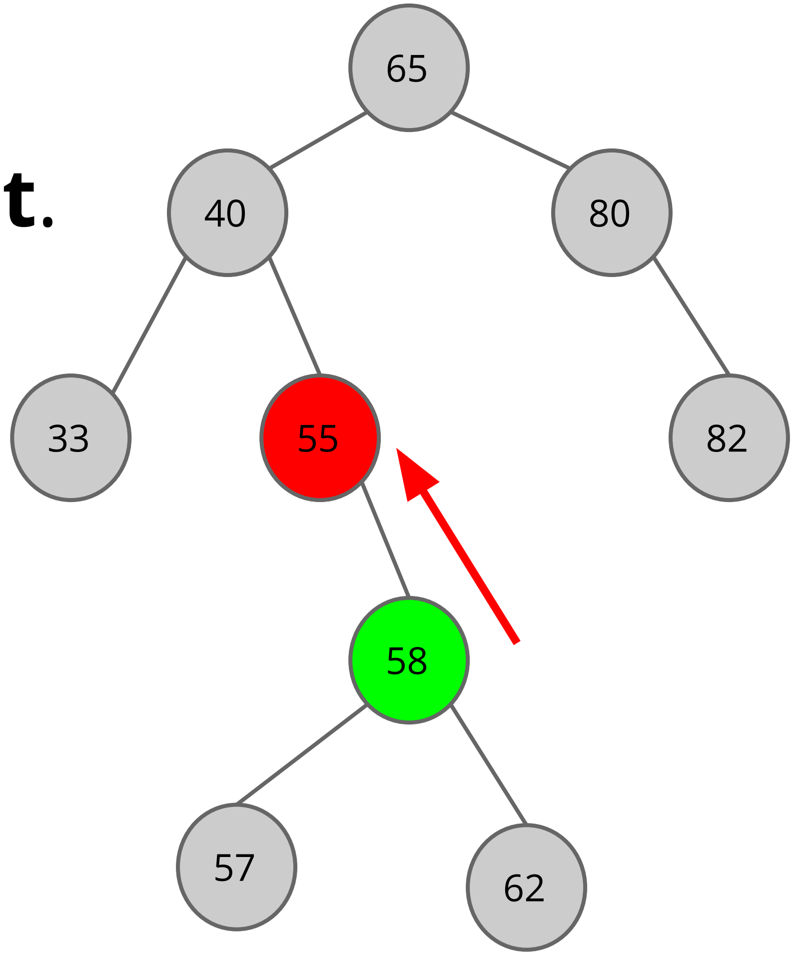
Just delete it, nothing else need to be changed.



Case 2: x has one child

First delete that node, which makes an **open spot**.

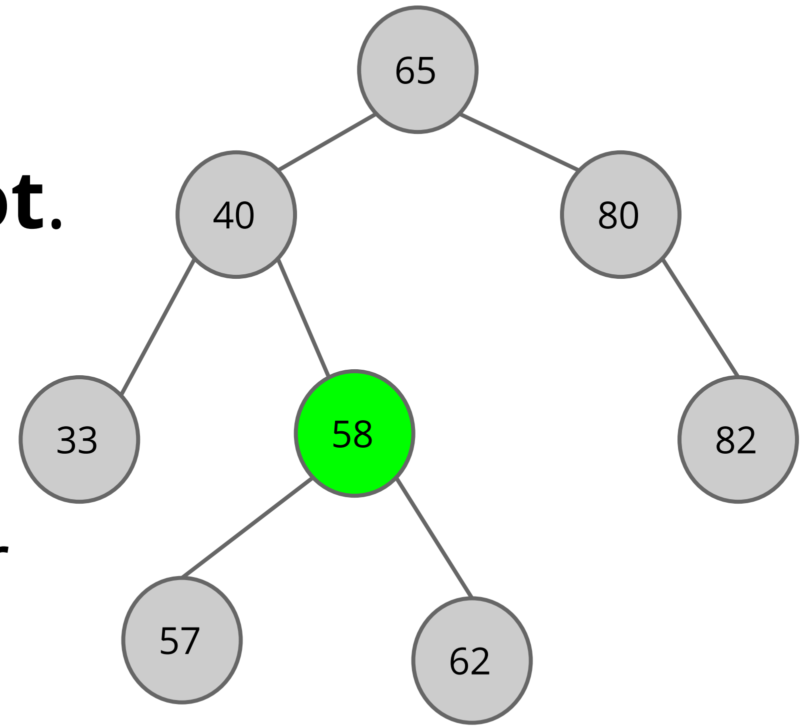
Then **promote** **x's only child** to the spot, together with the only child's subtree.



Case 2: x has one child

First delete that node, which makes an **open spot**.

Then **promote** **x's only child** to the spot, together with the only child's subtree.

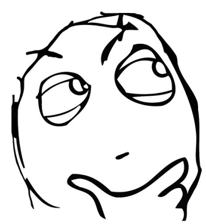


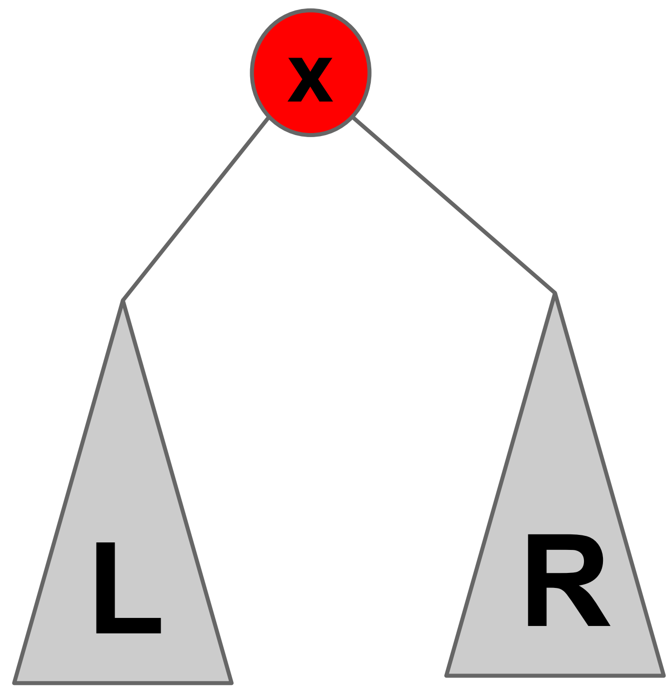
Case 3: x has two children

Delete **x**, which makes an open spot.

A node **y** should fill this spot, such that **L < y < R**,

Who should be y?





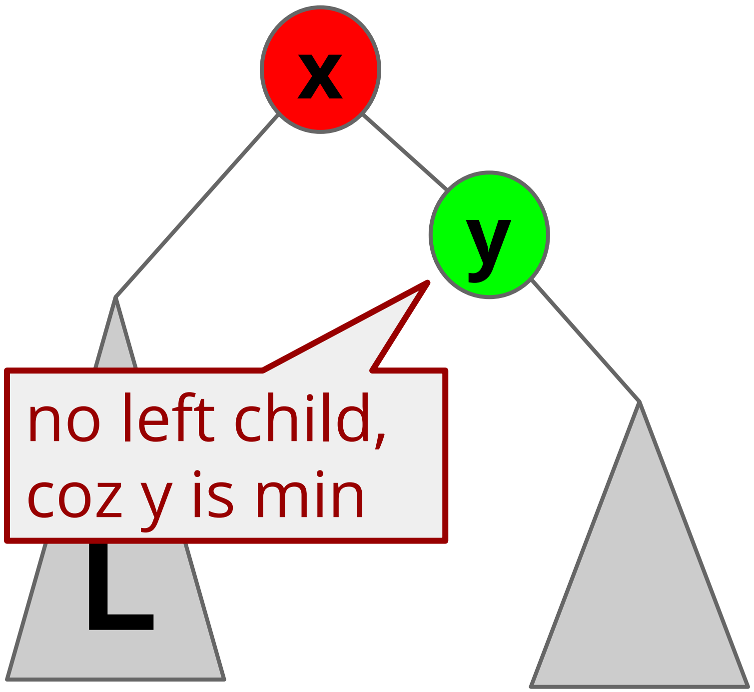
y ← the minimum of R, i.e., Successor(x)

L < y because y is in R, **y < R** because it's minimum

Further divide into two cases

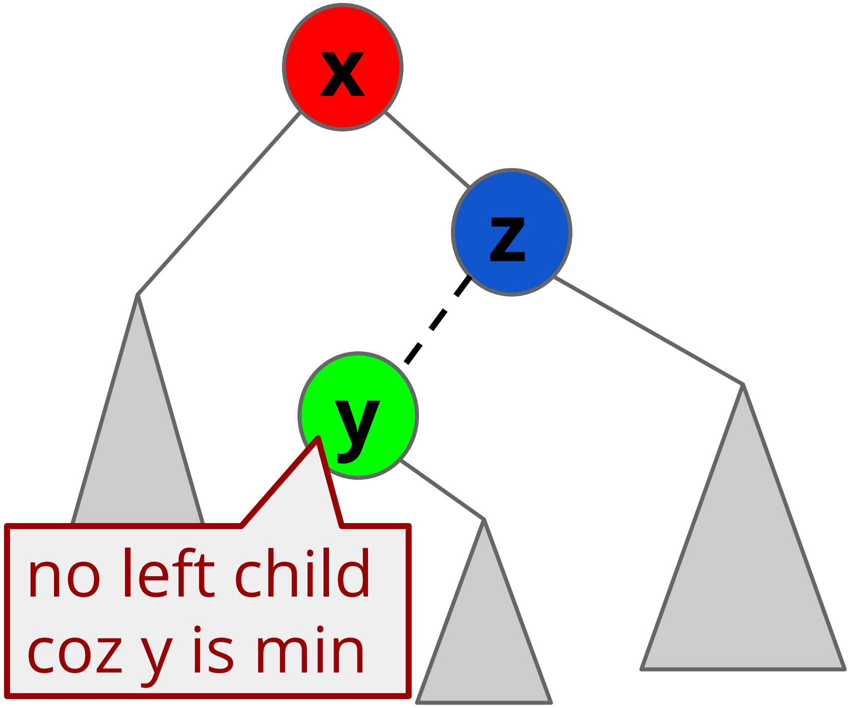
Case 3.1:

y happens to be the right child of **x**



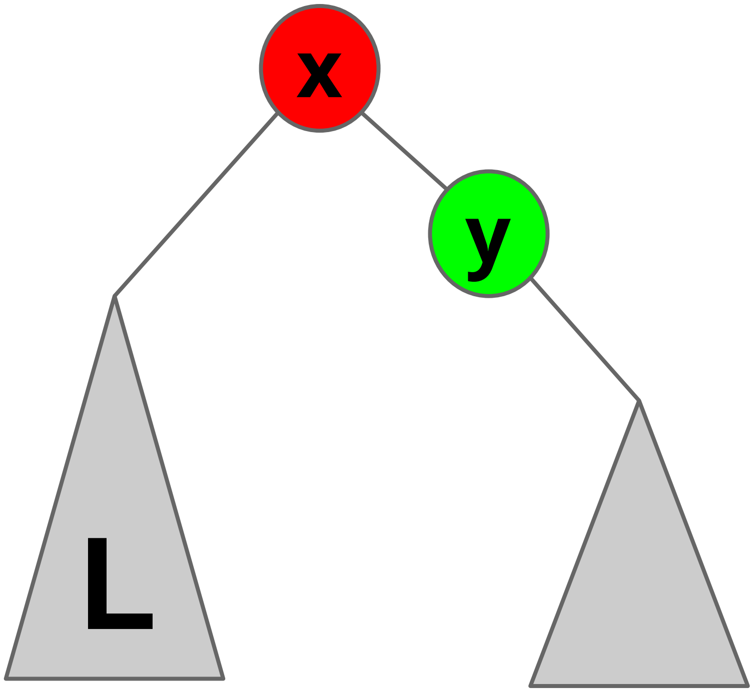
Case 3.2:

y is not the right child of **x**



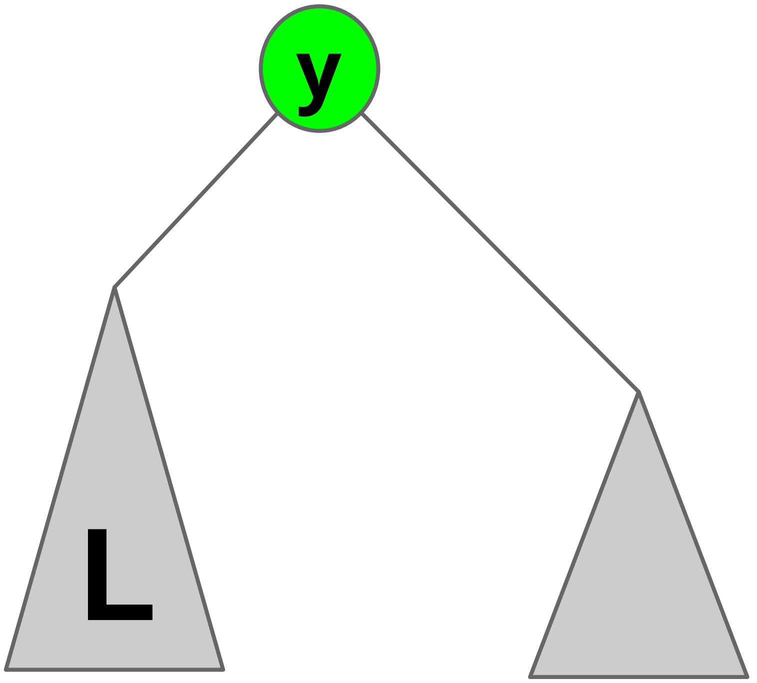
Case 3.1: y is x's right child

Easy, just **promote**
y to **x**'s spot



Case 3.1: y is x's right child

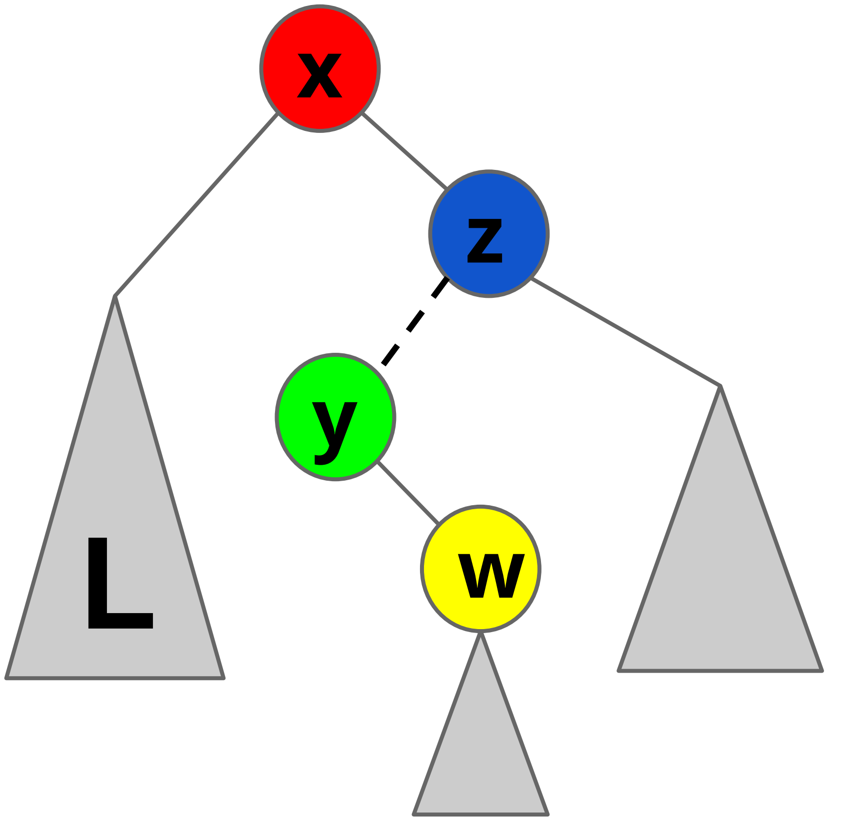
Easy, just **promote**
y to **x**'s spot



Case 3.2: y is NOT x's right child

1. Promote **w** to **y**'s spot, **y** becomes free.

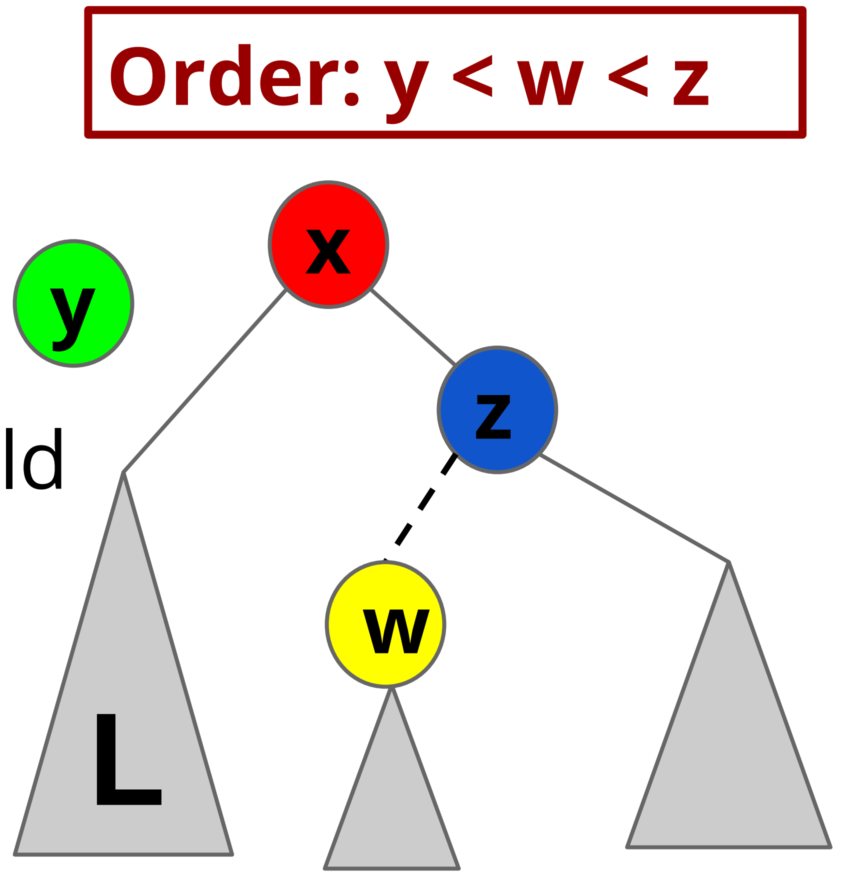
Order: y < w < z



Case 3.2: y is NOT x's right child

1. Promote **w** to **y**'s spot, **y** becomes free.

2. Make **z** be **y**'s right child (**y** adopts **z**)



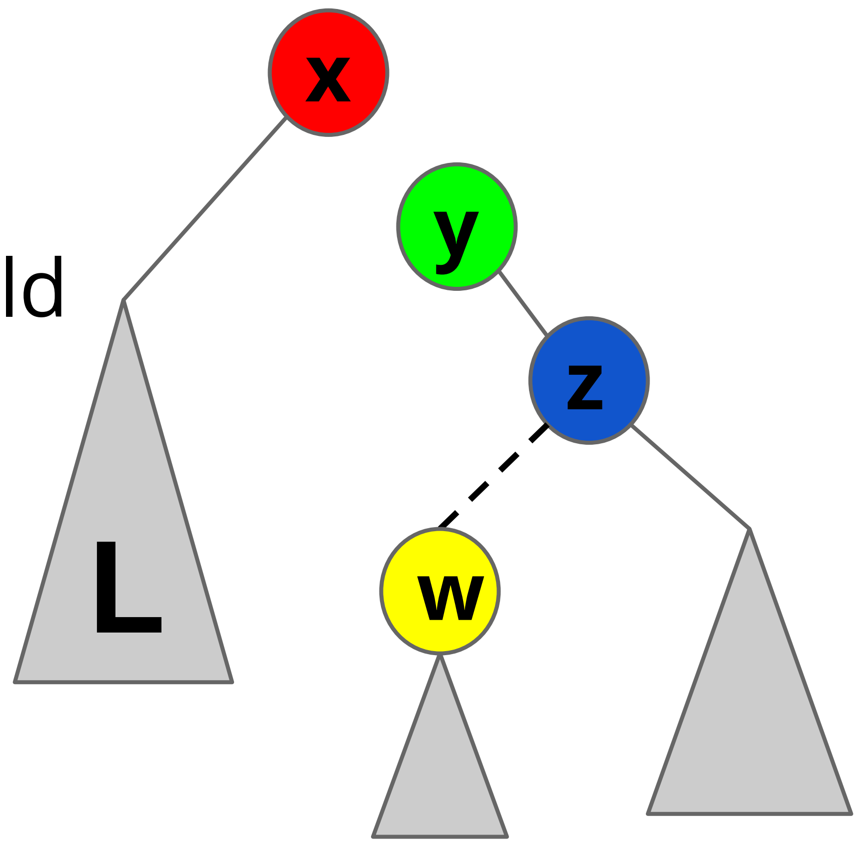
Case 3.2: y is NOT x's right child

1. Promote **w** to **y**'s spot, **y** becomes free.

2. Make **z** be **y**'s right child (**y** adopts **z**)

3. Promote **y** to **x**'s spot

Order: y < w < z



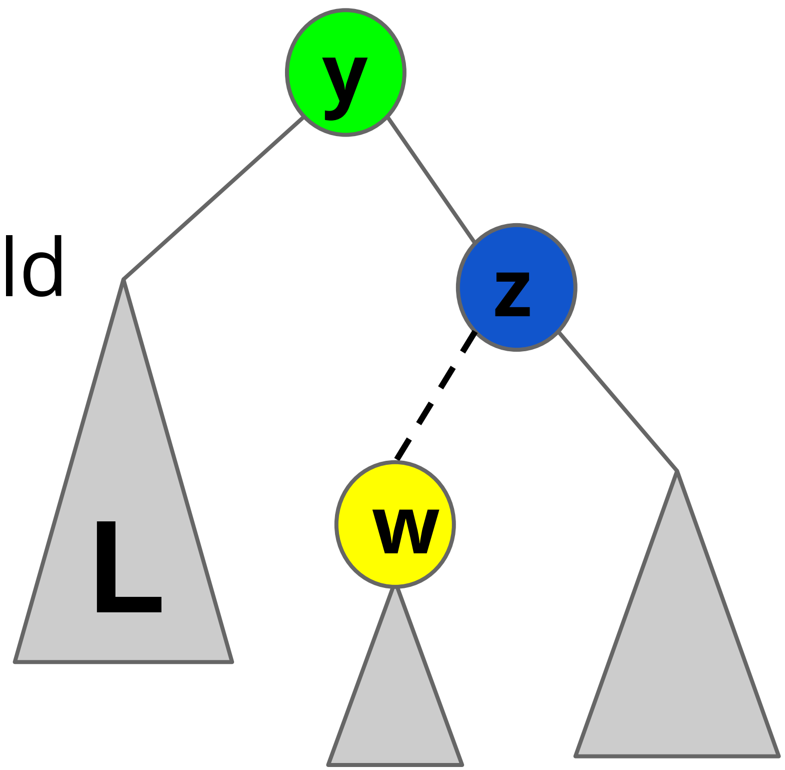
Case 3.2: y is NOT x's right child

1. Promote **w** to **y**'s spot, **y** becomes free.

2. Make **z** be **y**'s right child (**y** adopts **z**)

3. Promote **y** to **x**'s spot

Order: y < w < z



x deleted, BST order maintained, all is good.

Summarize TreeDelete(root, x)

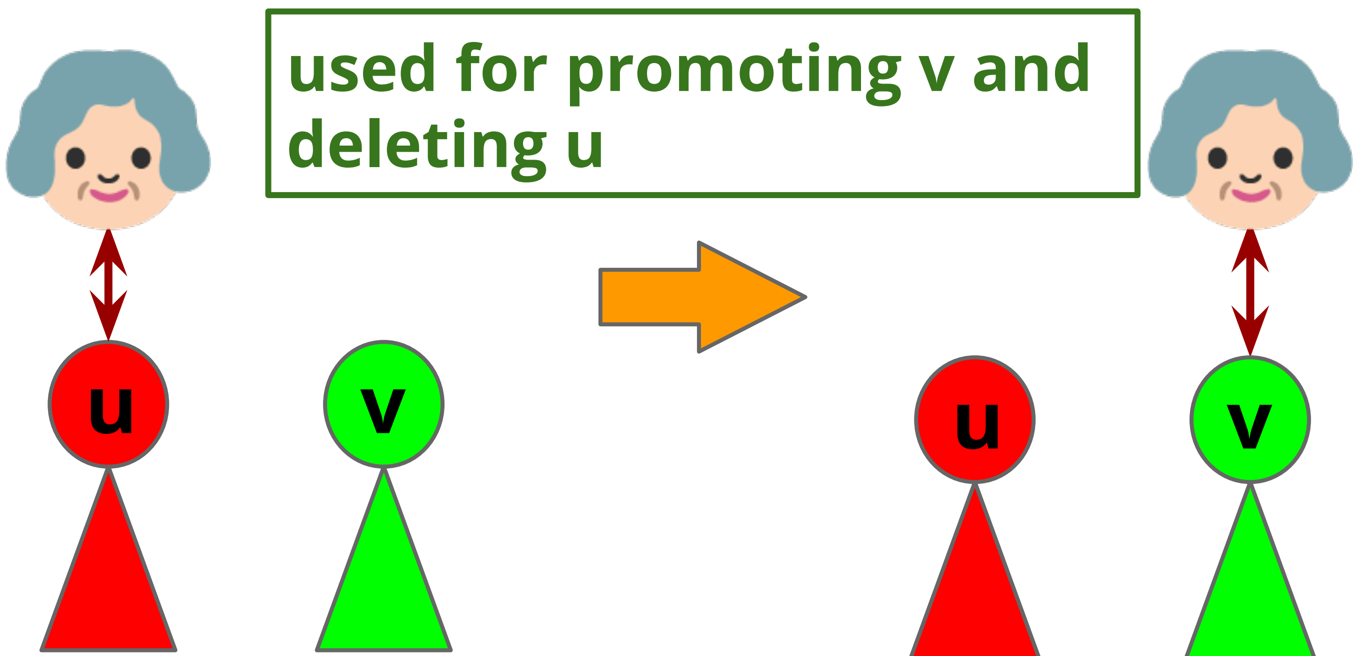
- Case 1: x has no child, **just delete**
- Case 2: x has one child, **promote**
- Case 3: x has two children, y = successor(x)
  - Case 3.1: y is x's right child, **promote**
  - Case 3.2: y is NOT x's right child
    - **promote** y's right child
    - y **adopt** x's right child
    - **promote** y

TreeDelete(root, x): pseudo-code

Textbook Chapter 12.3

Key: Understand **Transplant(root, u, v)**

 # v takes away u's parent



```
Transplant(root, u, v):
# v takes away u's parent
    if u.p = NIL:  # if u is root
        root ← v    # v replaces u as root
    elif u = u.p.left: # if u is mom's left child
        u.p.left ← v  #mom accepts v as left child
    else:  # if u is mom's right child
        u.p.right ← v  #mom accept v as right child
    if v ≠ NIL:
        v.p ← u.p  # v accepts new mom
                   # u can cry now...
```

```
TreeDelete(root, x):
  if x.left = NIL:
    Transplant(root, x, x.right)
  elif x.right = NIL:
    Transplant(root, x, x.left)
  else:
    y ← TreeMinimum(x.right)
    if y.p ≠ x:
      Transplant(root, y, y.right)
      y.right ← x.right
      y.right.p ← y
    Transplant(root, x, y)
    y.left ← x.left
    y.left.p ← y
  return root
```

Case 1 & 2

Case 3

Case 3.2

Promote right child

Promote left child

get successor(x)

y is not right child of x

promote **w**

y adopts **z**

promote y

update pointers

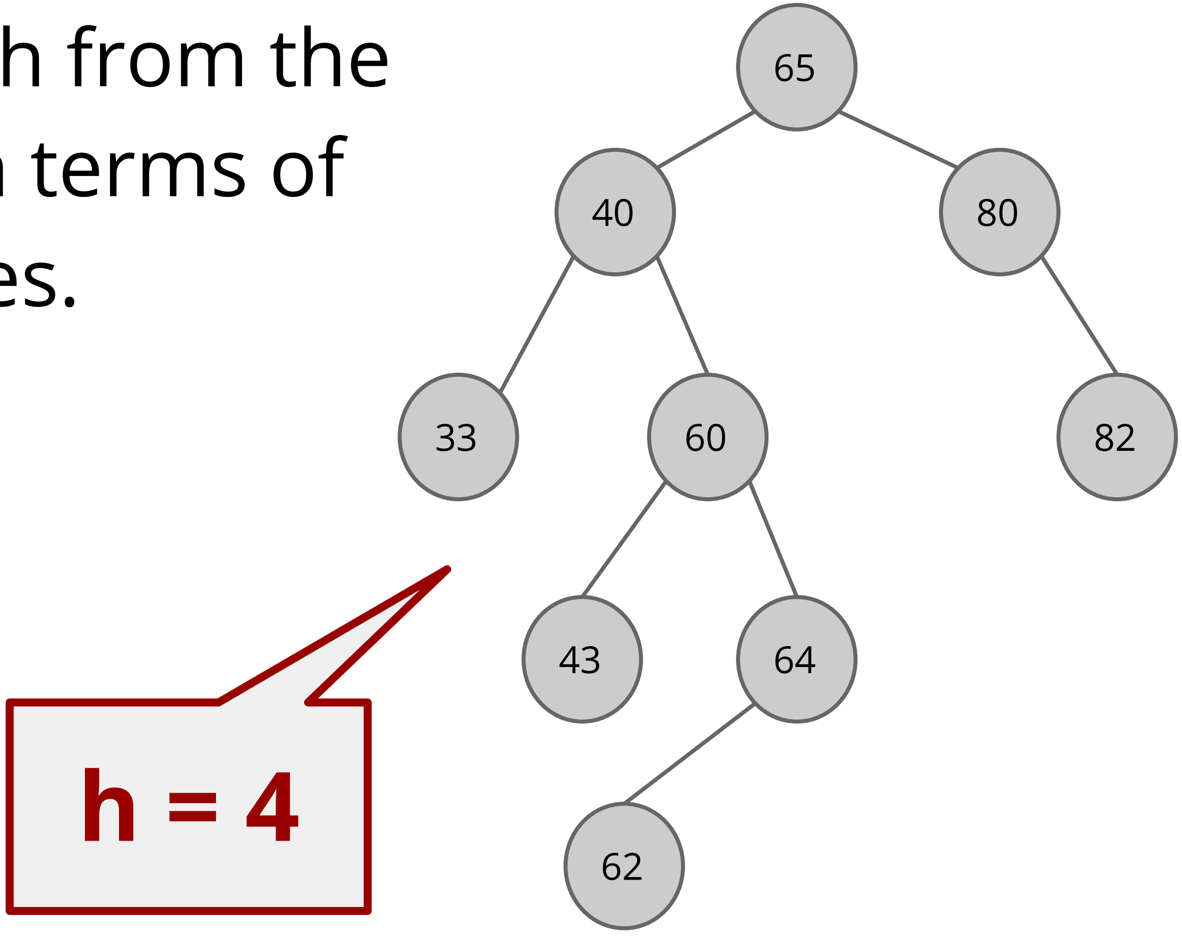
TreeDelete(root, x) worst case running time

O(h) (time spent on TreeMinimum)

Now, about that h (height of tree)

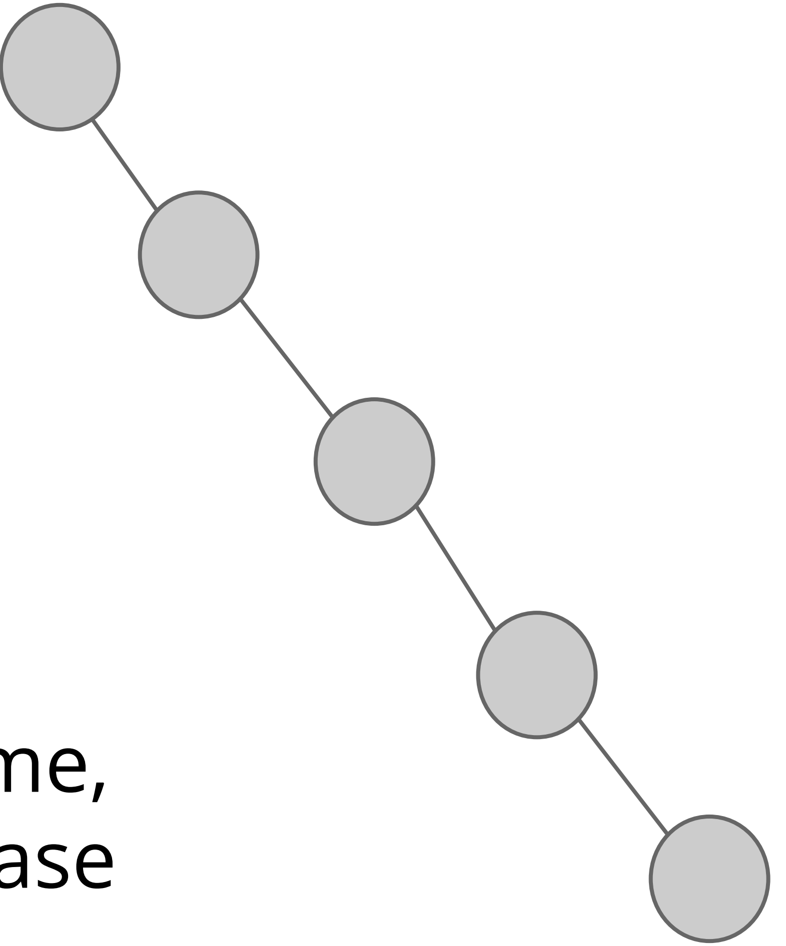
Definition: height of a tree

The longest path from the root to a leaf, in terms of number of edges.



Consider a BST with n nodes, what's the highest it can be?

h = n-1



i.e, in worst case

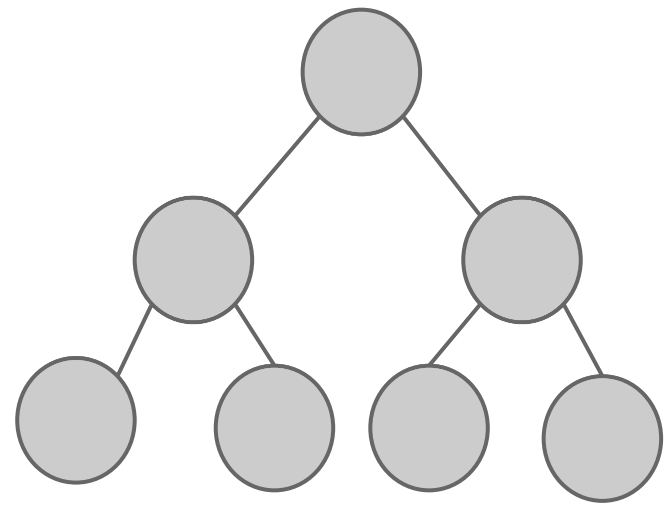
h ∈ Θ(n)

so all the operations we learned with **O(h)** runtime, they are **O(n)** in worst case

So, what's the best case for h ?

In best case, $h \in \Theta(\log n)$

A **Balanced BST**
guarantees to have height
in $\Theta(\log n)$



Therefore, all the **O(h)** become **O(log n)**

Next week

A Balance BST called **AVL tree**

[***http://goo.gl/forms/S9yie3597B***](http://goo.gl/forms/S9yie3597B)