Reminder: test 2 in lab this week!

TODAY:
- removing from a BST
- traversing a BST
- rebalancing discussion begins

REFRESHER ON OUR CONTEXT & MOTIVATION:

Dictionary ADT
- maps keys → values
- assumes keys are unique
- dictionaries are behind-the-scenes of many applications
- operations insert, get, update, remove must be fast
- previous data structures like List will be too slow

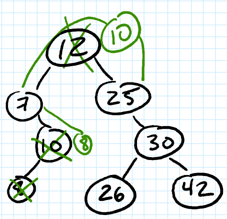
Plan: implement dictionary ADT as BST
- operations on BST are $O(\text{height})$
- as long as BST is "balanced" (fully packed) the height of a size-n BST is $O(\log_2 n)$
- ... and $O(\log_2 n)$ is really fast! (recall $\log_2(1\text{ billion}) \approx 30$)

BST: binary search tree
- binary tree (every node has left & right subtrees which can possibly be empty)
- binary search property invariant is true at every node
  - → All keys in the left subtree of a node must be less than the key at that node. All keys in the right subtree of a node must be greater than the key at that node.

Last week we discussed how to implement "get" and "insert".

the "remove" method

remove (12)
- find predecessor which is 10
- replace(key, value) of node with predecessor's (key, value)
- remove predecessor



remove (12)
replaced with 10
remove (10)
replaced with 8
remove (8)

Depending where the node is in the tree, remove should:

✓ ① if it's a leaf (just delete it)

② if it has only 1 child (replace (key, value) with that child's (key, value), then remove the child)

③ if it has 2 children (find the $\overset{\text{successor}}{\text{predecessor}}$, replace the node's (key, value) with $\overset{\text{successor}}{\text{predecessor's}}$ (key, value), then remove predecessor)

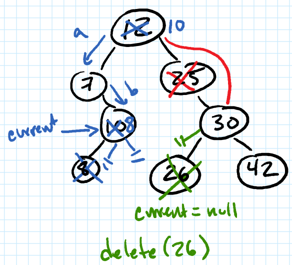
✓ ④ if it's NOT in the tree, throw error

How do we find the **predecessor** key of a node in a BST?

- the **predecessor** must be the **largest** key in the node's **left** subtree
- this is the **rightmost** node in the **left** subtree

How do we find the **successor** key of a node in a BST?

- the **successor** must be the **smallest** key in the node's **right** subtree
- this is the **leftmost** node in the **right** subtree



predecessor of 12? 10

successor of 12? 25

remove (25)

remove(12):

- found predecessor, 10

left child of root to a

remove from subtree(left of root , 10)

remove from subtree(b , 10)

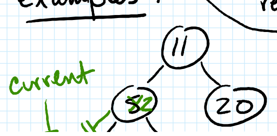
pseudocode for remove

```
void remove (K key) // public method
    // remove changes tree structure, so update root
    root = removeFromSubtree(root, key)
    decrement size

node* removeFromSubtree(current, key) // private helper
    if current is null
        throw exception "key not found"
    if key < current's key
        current's left = removeFromSubtree(current's left, key)
        return current
    else if key > current's key
        current's right = removeFromSubtree(current's right, key)
        return current
    else // this must mean we found the key in the BST at current node
        if current's left is null and current's right is null // current is a leaf
            delete current
            set current to null
            return current
        if current's left is not null and current's right is not null // current has 2 children
            get the predecessor of current
            set current's key to predecessor's key
            set current's value to predecessor's value
            current's left = removeFromSubtree(current's left, predecessor's key)
            return current
        if current's left is null // current has 1 child, right only
            temp = current's right
            delete current
            return temp
        if current's right is null // current has 1 child, left only
            temp = current's left
            delete current
            return temp
```

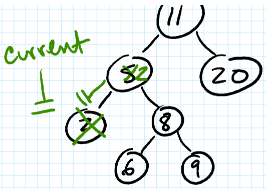
we're looking for key still

examples:

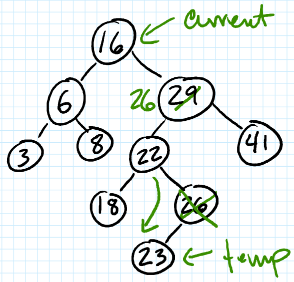


remove(5) this calls removeFromSubtree(→11 , 5) return → 11

removeFromSubtree(→5 , 5) return → 2



remove(5) this calls remove from subtree(→11, 5) return →11
remove From Subtree(→5, 5) return → 2
remove from subtree(→2, 2) return →11



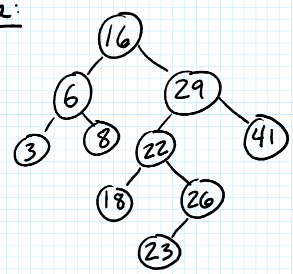
remove(29) this calls removeFromSubtree(→16, 29) return → 16
removeFromSubtree(→29, 29) return → updated 26 node
removeFromSubtree(→22, 26) return → 22
removeFromSubtree(→26, 26) return temp

TRAVERSING A TREE

Goal: visit every node in a BST and generate a list of (key, value) pairs in a particular order.

types of traversal:

example:



pre order: visit root, left subtree, right subtree
16, 6, 3, 8, 29, 22, 18, 26, 23, 41

post order: visit left subtree, right subtree, root
3, 8, 6, 18, 23, 26, 22, 41, 29, 16

in order: visit left subtree, root, right subtree
3, 6, 8, 16, 18, 22, 23, 26, 29, 41

level order: use BFS to go level-by-level
16, 6, 29, 3, 8, 22, 41, 18, 26, 23

implementing traversals:

```
vector<pair<K,V>> traversePreOrder()    // public method
    make a list to store result
    buildPreOrderTraversal(root, list)
    convert list to vector
    return vector

void buildPreOrderTraversal(current, list)  // private method which
                                            // adds to the list
    // base case
    if current is null
        return
    // recursive case
    else:
        add current's key and value to the list
        buildPreOrderTraversal(current's left, list)
        buildPreOrderTraversal(current's right, list)
```

return

Q: What would we change to implement:

1. post order? move the adding of current to AFTER the recursive calls
2. in order? move the adding of current to BETWEEN the recursive calls