Today

- AVL Trees (cont)

Assignments

- HW 10 out (due by end of semester)
- Extra Credit out (also due by end of semester)
(From Last Time): AVL Trees


- Calculate a “balance factor” for each node (left subtree – right subtree height)
  - Note: just store the height of ea. subtree root and calculate balance factor
- After every insert and delete, do “tree rotations” if needed
  - Either a single or double rotation may be needed
- Apply rotations (deletion) and update balance factors “up the tree”

Note: parent becomes 4’s right-child, and 4’s right-child becomes parent’s left-child
General case for single rotation (left–left case)

- Insertion in the left subtree of the left child
- Assumed that the tree is balanced before the rotation
- And becomes unbalanced after the rotation

Requires a “right” rotation:

After insert and rotation, done with rotations

- Subtree now rooted at $k_1$ is back to original height
- Since subtree is now balanced, no other rotations needed
Exercise: Fill in the following helper function:

```c
// return new root after left-left rotation
Node* single_rotate_right(Node* k2)
{
    Node* k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    return k1;
}
```
General case for single rotation (*right–right case*)

- Insertion in the *right* subtree of the *right* child
- The "mirror" image of the left–left case

Requires a *left* rotation:

Exercise: Fill in the following helper function:

```c
// return new root after right–right rotation
Node* single_rotate_left(Node* k2)
{
    Node* k1 = k2->right;
    k2->right = k1->left;
    k1->left = k2;
    return k1;
}
```

A single (right or left) rotation might *not* rebalance the tree ...
In this case, the right rotation didn’t help!!!

This is because we inserted into the left child’s right subtree
To deal with this, do two rotations (left-right case)

Start with a left rotation (at $k_2$):

Followed by a right rotation (at $k_3$):
Exercise: Fill in the following helper function:

- it is okay to reuse the previous functions as needed

```c
// return new subtree root of double rotation
Node* double_rotate_with_left_child(Node* k3)
{
    k3->left = single_rotate_left(k3->left);
    return single_rotate_right(k3);
}
```
The “mirror” image – right-left case

Rotate left then right:

Exercise: Fill in the following helper function:

- it is okay to reuse the previous functions as needed

```c
// return new subtree root of double rotation
Node* double_rotate_with_right_child(Node* k3)
{
    k3->right = single_rotate_right(k3->right);
    return single_rotate_left(k3);
}
```
Basic insertion algorithm (see book for more details):

1. Recursively traverse path for insertion
2. Insert node
3. During backtracking, update heights and determine balance factors
4. Rebalance (rotate) if node’s balance factor less than $-1$ or greater than $1$

Only need to perform (single or double) rotation once on insert

- After rotation performed, rebalancing is complete
- Height changes are “local” to insertion path
- Because inserting on a path doesn’t change height of other paths

Cost of insertion

- Cost for a single or double rotation is $O(1)$
- Traversing “down” the path is $O(\log n)$ since tree is balanced
- Traversing “up” the path for rotations and height update is $O(\log n)$
- Thus insertion is $O(\log n)$
Unlike for insertion, deletion may require multiple rebalance steps

- may need to keep applying rotations as we navigate back “up” the path

Note that with AVL trees ...

- Rotations and traversals are tricky to get right
- In practice, traversals (down and up tree) create overhead
- Red-black trees deal with this overhead (by only balancing down tree)
- Interestingly, they further relax balance constraints (but still $O(\log n)$)
Hints for HW10: Where each node stores its own height

1. The work is focused on add and remove (trivially in height() function)
2. Start with getting add to work, then do remove
3. Start by modifying add to:
   (a) work recursively
   (b) get height to be updated
   (c) call stubbed out rebalance function (just returning subtree_root)
4. Then work on the rebalance function
5. Then move on to remove
   (a) recursively find node to remove
   (b) find inorder successor interatively
   (c) copy key and value to subtree_root
   (d) call remove from subtree_root->right to remove inorder successor (via its key)
   (e) adjust/extend rebalance as needed (small issues not accounted for)
Basic idea of “backtracking”

- Backtracking occurs after a recursive call ...

```c++
Node* add(Node* subtree_root, const K& a_key, const V& a_val)
{
    if (!subtree_root) {
        ... create the node with a_key and a_val ...
    }
    else {
        if (a_key < subtree_root->key)
            subtree_root->left = add(subtree_root->left, a_key, a_val);
        else
            subtree_root->right = add(subtree_root->right, a_key, a_val);
        // backtracking:
        ... update height of subtree_root as needed ...
    }
    // backtracking: rebalance at the subtree_root (as needed)
    return rebalance(subtree_root);
}
```
Making sense of the rebalance step ... 

- calls the `rotate_left` and `rotate_right` helpers
- updates heights due to the rotations
- trick is to determine which rotation to call and when (based on heights)

```c
Node* rebalance(Node* subtree_root) {
    if (!subtree_root) // nothing to balance
        return subtree_root;

    Node* lptr = subtree_root->left;
    Node* rptr = subtree_root->right;

    // left but no right pointer (special case)
    if (lptr && !rptr && lptr->height > 1) {
        // ... check if need double rotation ...
        // ... do rotate right ...
        // ... adjust height ...
    }

    // right but no left pointer (special case)
    else if (... similar to above ...) {
        // ... similar to above ...
    }

    // left and right pointer, left "heavy"
    else if (...) {
        // ... similar to cases above ...
    }

    // left and right pointer, right "heavy"
    else if (...) {
        // ... similar to cases above ...
    }

    return subtree_root;
}
```