ECE 2574
Introduction to Data Structures and Algorithms

35: Balancing Trees: Treaps

Chris Wyatt
Electrical and Computer Engineering
Virginia Tech

Today we will see a way to approximately balance a binary search tree using *Treaps*, or a tree of heaps.

Review BST implementation

Subtree rotations

Treap nodes

Treap insert

Treap remove

Treap search

Program 5

The complexity of the Binary Search Tree is obviously important in applications.

Basic operations:

$\left.\begin{array}{l}\text{insert}\\ \text{delete}\end{array}\right\}$ These use search

search

How fast can we search a binary search tree for a key?

What is the best case ?

What is the worst case ?

What is the average case ?

What is the organization for searching a BST,

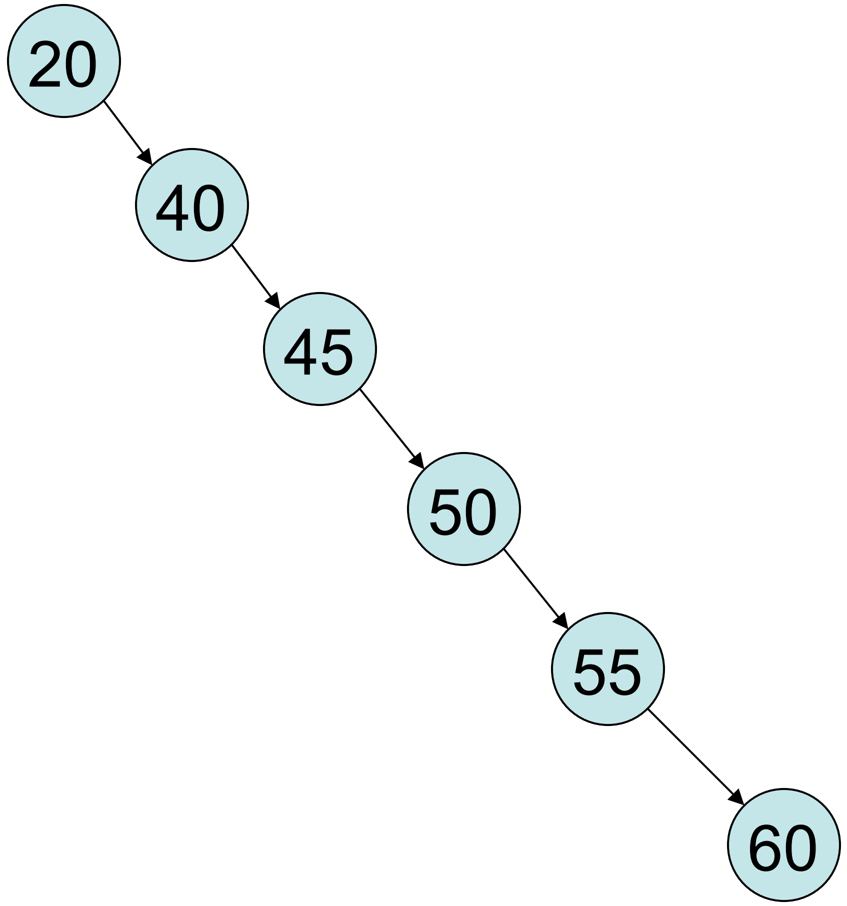
that has the smallest number of comparisons in the best case?

that has the smallest number of comparisons in the worst case?

that has the smallest number of comparisons in the average case?

What is the worst order of insertion into a binary search tree we could have?

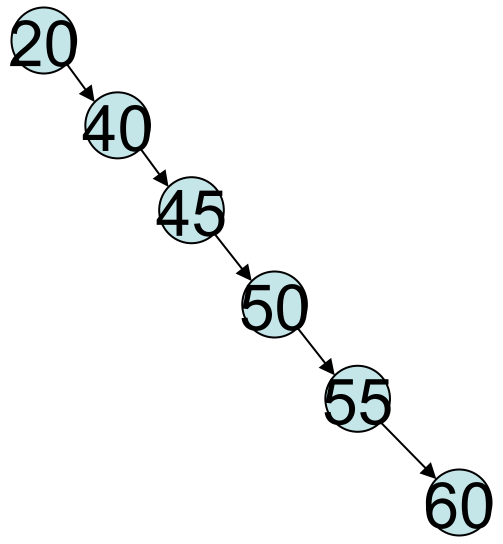
What if we inserted 20, 40, 45, 50, 55, 60 ?

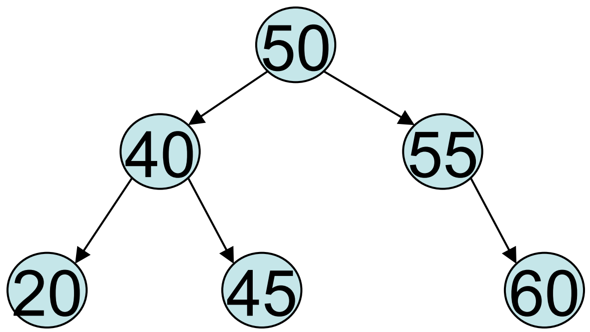


To summarize,

The average complexity (number of comparisons) when searching a BST is best when the tree is balanced.

So the question naturally arises, can we make a BST balanced?

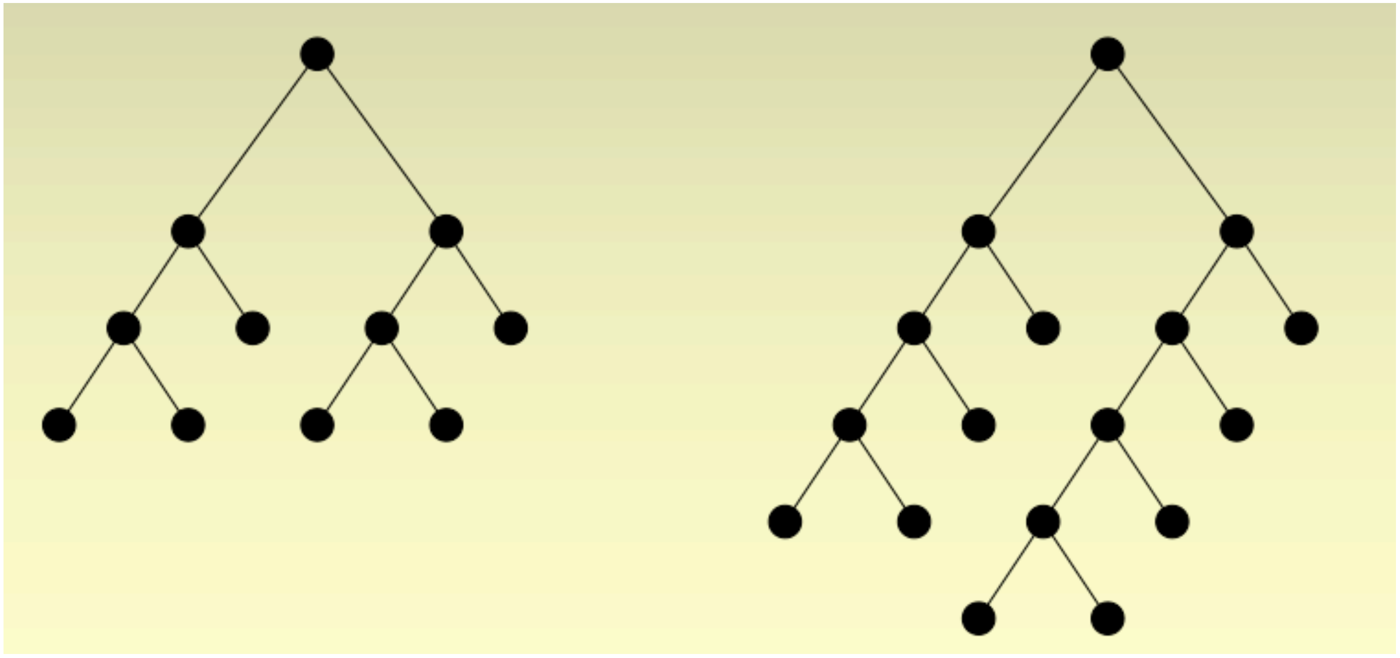




Balanced Trees

Recall, a tree of height h is balanced if it is full down to level h-1;

and the depth of a tree was the number of nodes from the root to a leaf.



Basic approach to making a balanced binary tree.

1. Insert/Delete a node
2. Restore the balance of the tree.

The primary tool used to restore balance is called a rotation.

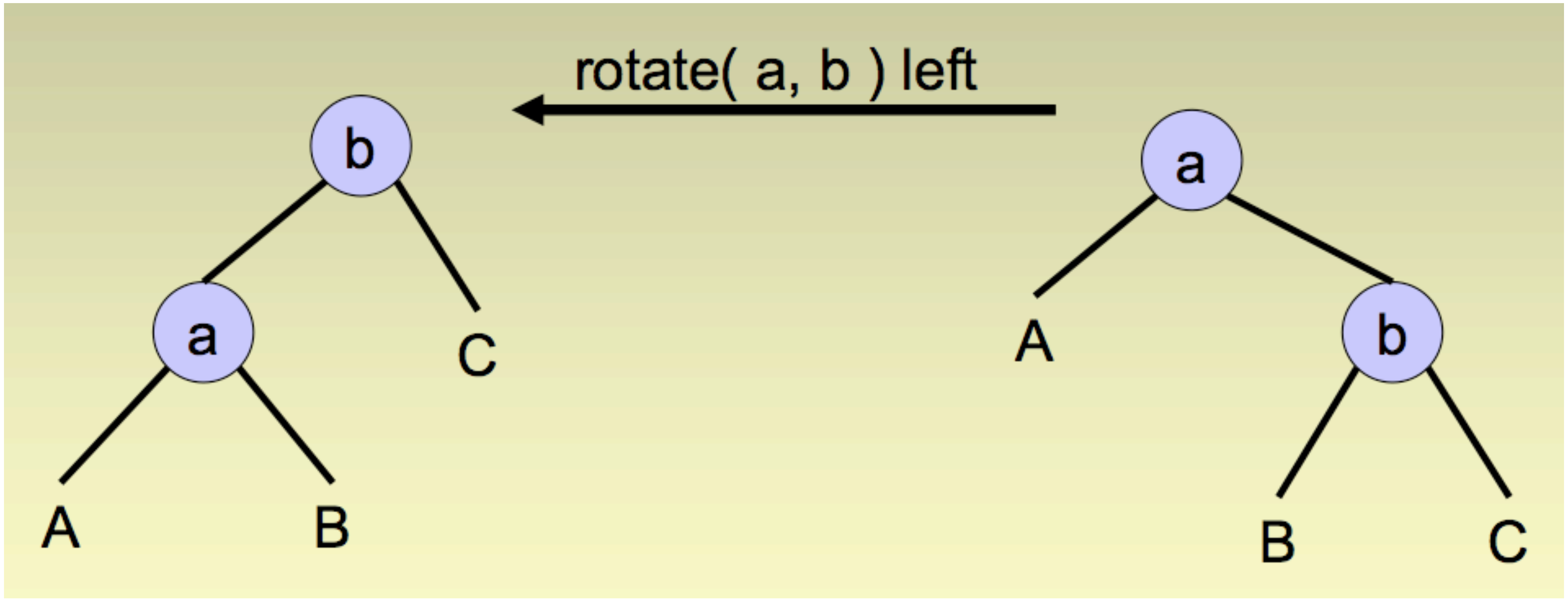
There are left and right rotations

and

the rotation should not violate the binary tree property.

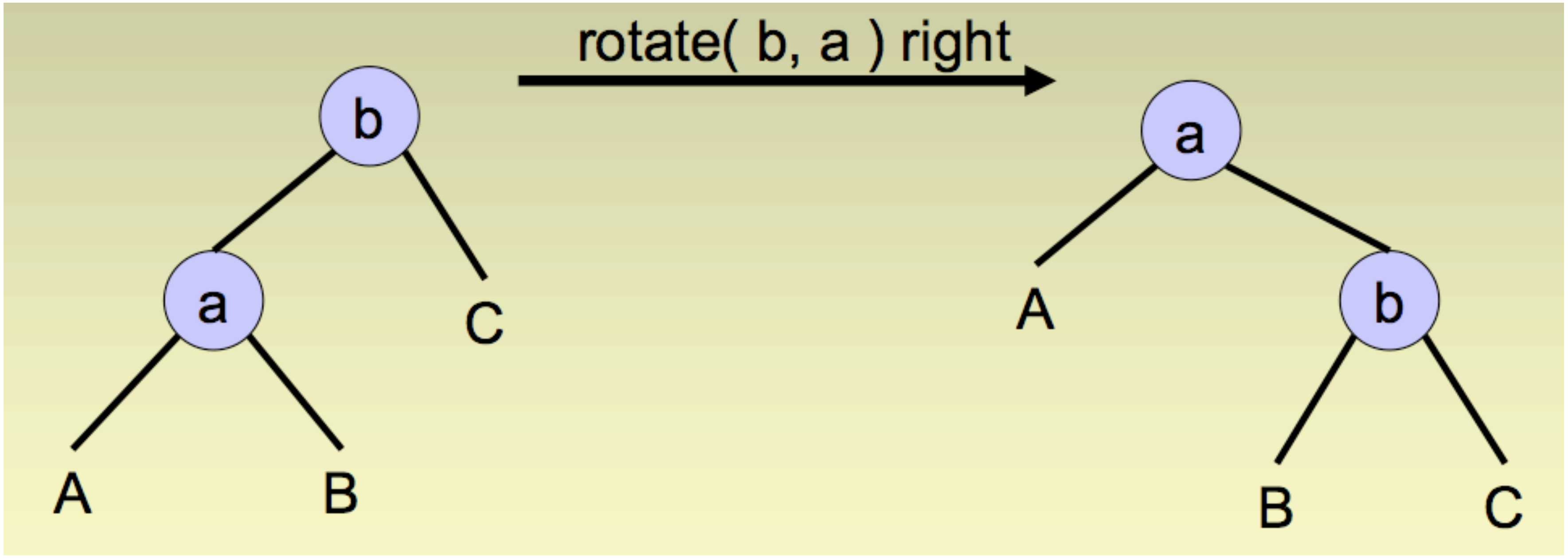
Left rotation

Let A, B, and C be subtrees and a, b nodes in the following tree.



Right rotation

Let A, B, and C be subtrees and a, b nodes in the following tree.

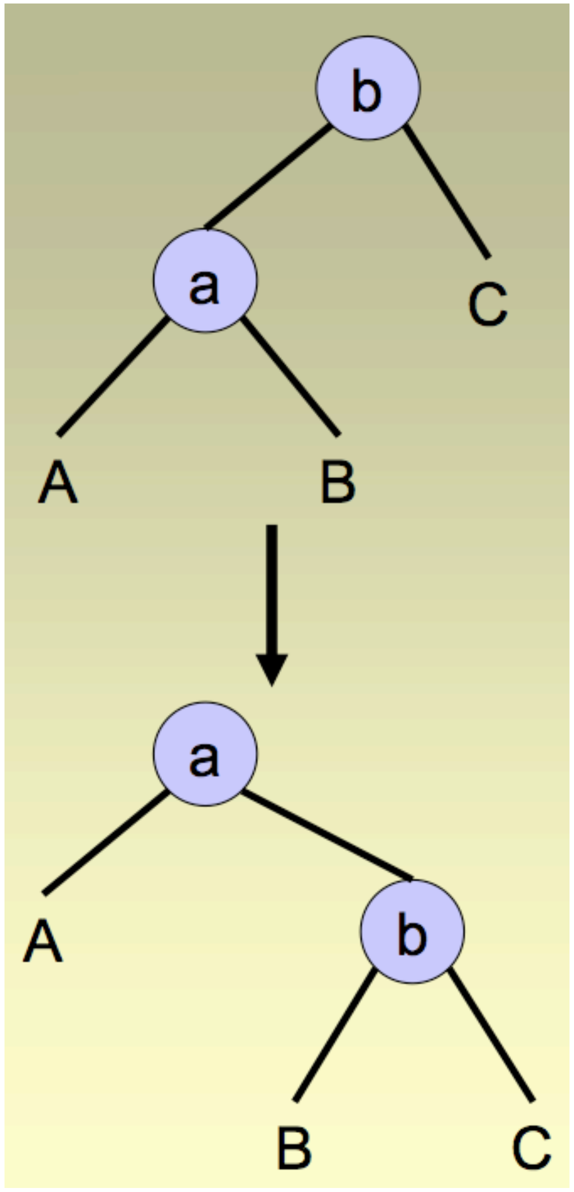


Pseudo-code for doing a rotate right

```
// rotate a tree rooted at node
rotateRight(in node:TreeNode)

b = node
a = node->left_child
// if b is a left child
if b = b->parent->left_child
    b->parent->left_child= a
else // b is a right child
    b->parent->right_child= a
endif

make B subtree left subtree of b
make b into a' s right child
```

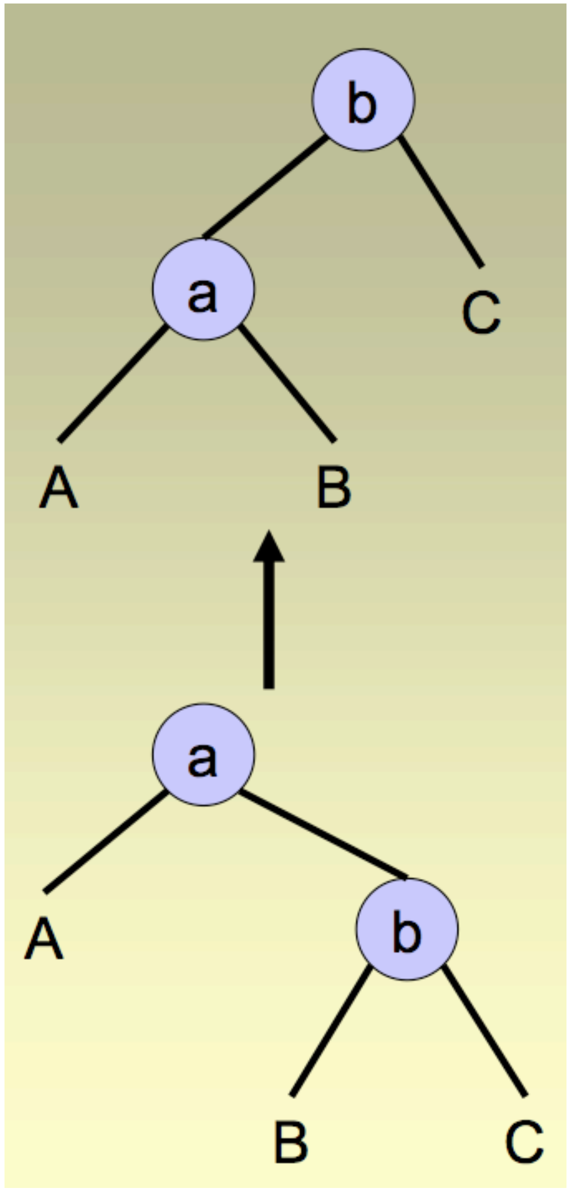


In class exercise: modify to perform a rotate left.

```
// rotate a tree rooted at node
rotateRight(in node:TreeNode)

b = node
a = node->left_child
if b = b->parent->left_child // if b is a left child
    b->parent->left_child = a
else // b is a right child
        b->parent->right_child = a
endif

make B subtree left subtree of b
make b into a's right child
```



The two most popular balanced Binary Trees are the AVL and the Red-Black Tree.

AVL trees (named after Adel'son-Velsk'ii and Landis who introduced them in 1962) require that the depths of the left and right subtrees of any node differ by at most one.

The rules to enforce the AVL property are complex.

The two most popular balanced Binary Trees are the AVL and the Red-Black Tree.

A related tree, introduced by Bayer in 1972, called red-black trees (symmetric binary B-trees) have the property: no path from the root to the leaf has length more than twice the length of any other path.

Red-black trees are much easier to implement, but still rather involved.

We will look at them in detail Friday.

A similar type of tree that is easier to program is the *Treap*.

A treap is a binary search tree that remains balanced in the sense that with high probability the height of the tree is proportional to the log of the number of nodes.

Each node is augmented with a randomly generated priority.

Like a normal BST the keys in the left/right subtrees are less/greater than the subtree root key, but they also form a heap with respect to the priority.

To implement the treap we add a priority member to the node structure.

When a node is created the priority is assigned a random value.

See `treap_bst.h` and `treap_bst.txx`

Treap Insert

1. Insert using binary search tree algorithm
2. Generate a random priority
3. Perform rotations to bubble up the node based on priority
   - If node is a left child, rotate right about parent
   - If node is a right child, rotate left about the parent
   - Stop when parent priority > node priority or node is root

Treap Insert Examples

Treap Remove

1. Search for the node
2. 3 cases:
    1. Node is a leaf, just remove it
    2. Node has a single child, remove node and replace with child
    3. Node has 2 children. Rotate about node with the direction determined by the relative ordering of the child priorities.

    If lchild priority < rchild priority rotate left

    If lchild priority > rchild priority rotate right

    Continue as long as node has two children.

Treap remove

Hints:

Do the work in case #3 first, this reduces to case #2

Then test for case #1 and #2.

Treap Remove Examples

Treap Search

This is just normal Binary Search Tree search.

Next Actions and Reminders

Read CH pp. 544-563 on Hash Tables

Program 5 is due 12/11