Lecture 4

- Binary search tree average cost analysis
- The importance of being balanced
- AVL trees and AVL rotations
- Insert in AVL trees

Reading: Weiss Ch 4, sections 1-4

Best, worst, and average case time costs in data structures

- Often we would like to analyze time costs for data structure operations
- Typical operations we would care about would be: successful find, unsuccessful find, insert new data, update existing data, delete data...

- We are looking for time cost *functions* that are functions of the size of the problem
  - For data structure operations, the size of the problem is: N, the number of data items stored in the structure

- We can consider the time cost of performing one of those operations...
- ... and we will want to think about the *best*, *worst*, or *average* case for that operation

- Whether analyzing best, worst, or average case, there are two aspects to the question:
  - What is the internal state of the data structure?
  - What data element is the operation being performed on?

- You have to think about both of these...

Data structure state and data element choice

- For each of the following, give an example of an internal state of the data structure, and a choice of data element, that gives the correct answer

- What is the worst-case number of nodes visited in an unsuccessful find operation in a binary tree with N nodes?

- What is the best-case number of nodes visited in an unsuccessful find operation in a binary tree with N nodes?

- What is the worst-case number of nodes visited in a successful find operation in a completely filled binary tree with $N = 2^{H+1} - 1$ nodes?

Analyzing Find and Insert in binary search trees

- Define the "depth" of a node x_i in the tree as the number of nodes on the path from the root to x_i inclusive (thus the depth of x_i is equal to the zero-based level of x_i , plus 1)
- In the worst case, the number of comparisons in the Find and Insert operations is equal to the depth of deepest leaf of the tree (and the total number of steps required for any of these operations is proportional to the number of comparisons)
- In a binary search tree with N nodes, this can be as large as:________
- This worst-case performance is not very good, and it very easily can occur: for example, when items are inserted in the BST in sorted, or reverse sorted, order!
  - and various partially sorted insertion orders are as bad, or almost as bad
- So, overall worst-case time cost of BST operations is not very good at all.
- But what about average-case?...

Analysis of average-case successful find in a BST

- We will do an average case analysis of the "successful find" operation: how many steps are required, on average, for the find algorithm to find a key that's in the tree?
  - (Analyzing unsuccessful find, insert, and delete would differ in some details, but would have the same asymptotic big-Θ result)
- Suppose you have a BST with N nodes $x_1, \ldots, x_N$, holding keys $k_1, \ldots, k_N$, such that x_i is the node holding key k_i
- Let the depth of node x_i be $d(x_i)$, so the number of comparisons required to find key k_i in this tree is $d(x_i)$
- Let the probability that you will search for key k_i be p_i
- Then the average number of comparisons needed in a successful find operation is:

$$D_{avg}(N) = \sum_{i=1}^{N} p_i d(x_i)$$

- So we need to know the p_i and the $d(x_i)$ ...

Probabilistic assumption #1

- Whenever you do average case analysis, you need to be clear about the probabilistic assumptions you are making

- Then when using the results of the analysis, be aware that the results may not be relevant in cases where the assumptions do not hold!

- Here our first assumption is:
- **Probabilistic Assumption #1**: all keys in the tree are equally likely to be searched for.

- This means that $p_1 = \ldots = p_N = \frac{1}{N}$, and so the average number of comparisons in a successful find is

$$D_{avg}(N) = \sum_{i=1}^{N} \frac{1}{N} d(x_i) = \left(\sum_{i=1}^{N} d(x_i) \right) / N$$

- Here the quantity $\sum_{i=1}^{N} d(x_i)$ is the *total node depth* of the tree; so with Assumption #1, the depth of the average node in a tree is just the total node depth divided by the number of nodes in the tree

The need for an additional assumption

- With Assumption #1, the average number of comparisons in a successful find is equal to the average depth of the nodes in the tree. But obviously this in turn depends on the shape of the tree; for example:
  - If the N-node BST is a linked list, then the average node depth is

$$\frac{1}{N}\sum_{i=1}^{N} d(x_i) = \frac{1}{N}\sum_{i=1}^{N} i = \frac{N+1}{2} = O(N)$$

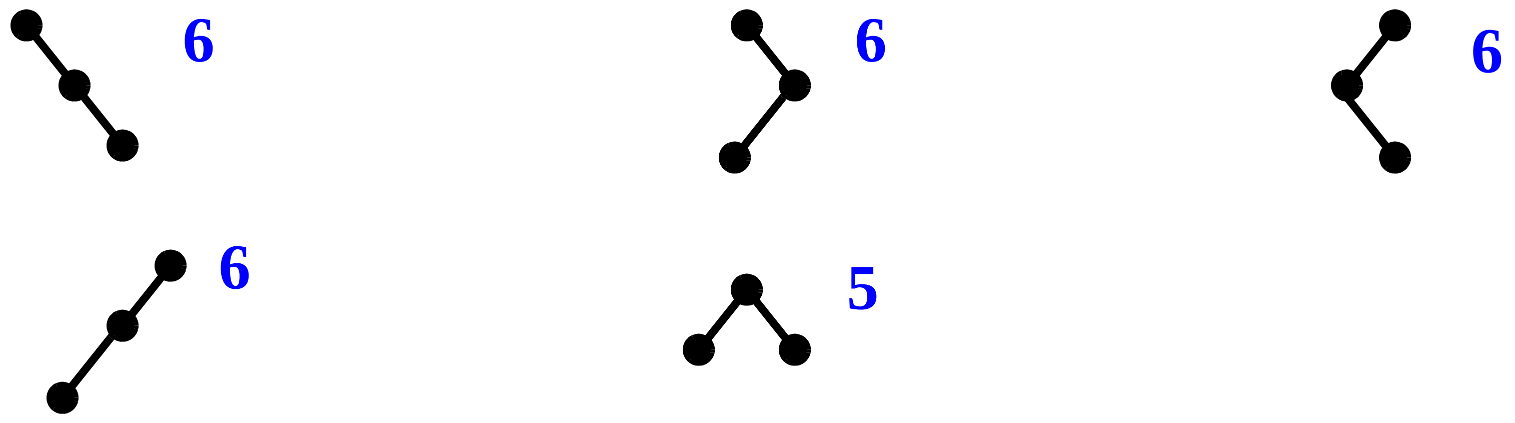
  - If the BST is a completely filled binary tree, then the average node depth is

$$\frac{1}{N}\sum_{i=1}^{N} d(x_i) = \frac{1}{N}\sum_{i=1}^{\log(N+1)} i2^{(i-1)} < \log(N+1) = O(\log N)$$

- This gives us the average number of comparisons needed for a successful find, *in a particular BST with N nodes*
- But in a complete analysis of the average case, we need to average over all the different possible BST's with *N* nodes...
  - ... and to do that we need to make an assumption about what are the probabilities of occurrence of each of these different BST's

Different BST's with N nodes

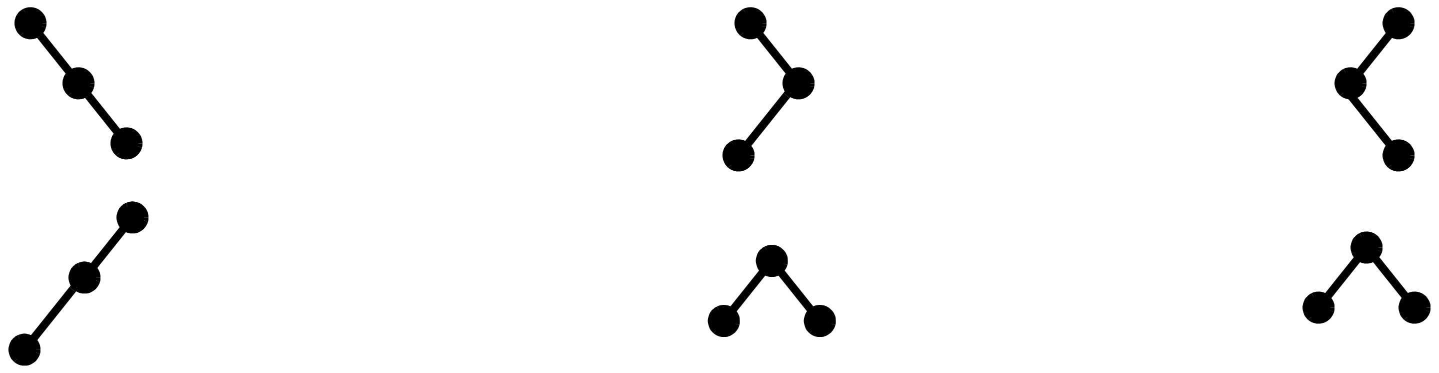
- As a (small) example, let $N = 3$. There are 3 distinct keys in the tree: k_1, k_2, k_3
- What are the possible shapes of a BST holding 3 keys?
- Here are the possibilities, each with their total node depth:



- And so the average node depths in these trees (with Assumption #1) are 2, 2, 2, 2, and 5/3. But what is the probability of each of those trees occurring?
- Once we know that, we can compute the average depth of a node (and so the average number of comparisons for a successful find) in a 3-node BST
- (Of course we want to solve this in general, for any N)

Probabilistic assumption #2

- Given a set of N keys, and the usual BST insert algorithm, the structure of a BST holding those keys is completely determined by the order of insertion of the keys
  - For example, if the keys $k_1, \ldots, k_N$ are inserted in sorted order, a linked list results. If instead the “middle”key is inserted first, and then recursively the “middle” keys of the remaining subsets, the tree will be nicely balanced
- But in the absence of any other information, let’s assume that each key of $k_1, \ldots, k_N$ is equally likely to be the first key inserted; each remaining key is equally likely to be the next one inserted; etc. Another way to put this assumption is:
- **Probabilistic Assumption #2**: Any order of insertion (i.e. any permutation) of the keys $k_1, \ldots, k_N$ is equally likely to be used to build the BST
- Example: N=3. There are N! = 3! = 6 different orders of insertion of the 3 keys. Here are the trees that result, and we will assume each has probability 1/6 of occuring:



Average total depth of a BST

- Recall the "total depth" of a BST is the sum of the depths of its nodes, that is:

$$\text{total depth of BST} = \sum_{i=1}^{N} d(x_i)$$

- Let $D(N)$ be the *average* total depth of BST's with N nodes, averaging over all the $N!$ possible N-node BST's assuming Probabilistic Assumption #2 holds. That is:

$$D(N) = \sum_{\text{all BSTs with N nodes}} (\text{probability of the BST})(\text{total depth of the BST})$$

$$D(N) = \sum_{\text{all BSTs with N nodes}} \left(\frac{1}{N!}\right)\left(\sum_{i=1}^{N} d(x_i)\right)$$

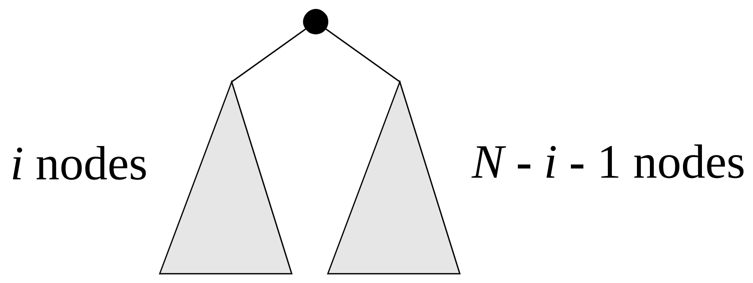
- Then, assuming Assumption #1 also holds, the average number of comparisons needed for a successful find in an N-node BST is just

$$D_{avg}(N) = D(N)/N$$

- And that's what we're looking for! So let's first find what $D(N)$ is. (Instead of finding the total depth for each tree, we will solve a recurrence relation for $D(N)$ directly)

Towards a recurrence relation for average BST total depth

- Imagine that we know the average total BST depth $D(i)$ for all $i < N$
- For an intermediate step, let $D(N|i)$ be the average total depth of a binary search tree with *N* nodes, given that its left subtree has *i* nodes (and so its right subtree has *N* - *i* - 1)



- Now the average total number of comparisons needed to find all the keys in the left subtree is $D(i) + i$, since each of the i nodes node there has its path length from the root increased by 1 by having to go through the root; average total number of comparisons needed to find all the keys in the right subtree is $D(N-i-1) + N-i-1$, for the same reason; and key in the root has depth 1.
- So, the following equation holds:

$$D(N|i) = [D(i) + i] + [D(N-i-1) + N-i-1] + 1$$

$$= D(i) + D(N-i-1) + N$$

Probability of subtree sizes

- Let $P_N(i)$ be the probability that the left subtree of the root of a BST with N nodes has i nodes
- Then $D(N)$, the quantity we seek, is

$$D(N) = \sum_{i=0}^{N-1} P_N(i) D(N|i)$$

- How many nodes the left subtree has is determined by which key $k_1, \ldots, k_N$ is inserted in the tree first. If the smallest was inserted first, the left subtree has 0 nodes; if the second smallest is inserted first, the left subtree has 1 node; etc.
- But by Assumption #2, any of the N keys are equally likely to be inserted first, so all left subtree sizes from 0 through N-1 are equally likely: $P_N(i) = 1/N$
- Therefore, the average total depth of a binary search tree with N nodes is

$$D(N) = \sum_{i=0}^{N-1} \frac{1}{N}[D(i) + D(N-i-1) + N]$$

$$= \frac{1}{N}\sum_{i=0}^{N-1} D(i) + \frac{1}{N}\sum_{i=0}^{N-1} D(N-i-1) + N$$

- Note that those two summations just add the same terms in different order; so

$$D(N) = \frac{2}{N}\sum_{i=0}^{N-1} D(i) + N$$

- ... and multiplying by *N*,

$$ND(N) = 2\sum_{i=0}^{N-1} D(i) + N^2$$

- Now substituting *N*-1 for *N*,

$$(N-1)D(N-1) = 2\sum_{i=0}^{N-2} D(i) + (N-1)^2$$

- Subtracting that equation from the one before it gives

$$ND(N) - (N-1)D(N-1) = 2D(N-1) + N^2 - (N-1)^2$$

- ... and collecting terms finally gives this recurrence relation on *D*(*N*):

$$ND(N) = (N+1)D(N-1) + 2N - 1$$

Solving the recurrence relation

- To solve that recurrence relation, divide by $N(N+1)$ to get

$$\frac{D(N)}{N+1} = \frac{D(N-1)}{N} + \frac{2N-1}{N(N+1)}$$

- See that this telescopes nicely down to $N=1$:

$$\frac{D(N-1)}{N} = \frac{D(N-2)}{N-1} + \frac{2(N-1)-1}{(N-1)N}$$

$$\cdots$$

$$\frac{D(2)}{3} = \frac{D(1)}{2} + \frac{4-1}{2\cdot 3}$$

$$\frac{D(1)}{2} = \frac{D(0)}{1} + \frac{2-1}{1\cdot 2}$$

The solution

- Note that $D(1)$ is equal to 1. Collecting terms, we get

$$\frac{D(N)}{N+1} = \sum_{i=1}^{N} \frac{2i-1}{i(i+1)} = \sum_{i=1}^{N} \frac{2}{(i+1)} - \sum_{i=1}^{N} \frac{1}{i(i+1)}$$

- To simplify further, you can prove this identity for the second term (by induction):

$$\sum_{i=1}^{N} \frac{1}{i(i+1)} = \frac{N}{N+1}$$

- ... and rewrite the first term as

$$\sum_{i=1}^{N} \frac{2}{(i+1)} = 2\sum_{i=1}^{N} \frac{1}{i} + \frac{2}{N+1} - 2 = 2\sum_{i=1}^{N} \frac{1}{i} - \frac{2N}{N+1}$$

- And so

$$D(N) = 2(N+1)\sum_{i=1}^{N} \frac{1}{i} - 3N$$

- The summation can be computed exactly to determine $D(N)$ for any N (and then divide by N to get $D_{avg}(N)$). Or, we can seek a useful approximation...

Approximating D(N)

- The sum can be approximated within an additive constant by a definite integral (here ln is the natural logarithm):

$$\sum_{i=1}^{N} \frac{1}{i} \approx \int_{1}^{N} \frac{1}{x} dx = \ln(x)\Big|_1^N = \ln(N)$$

- And so the average-case number of comparisons for a successful find is approximately

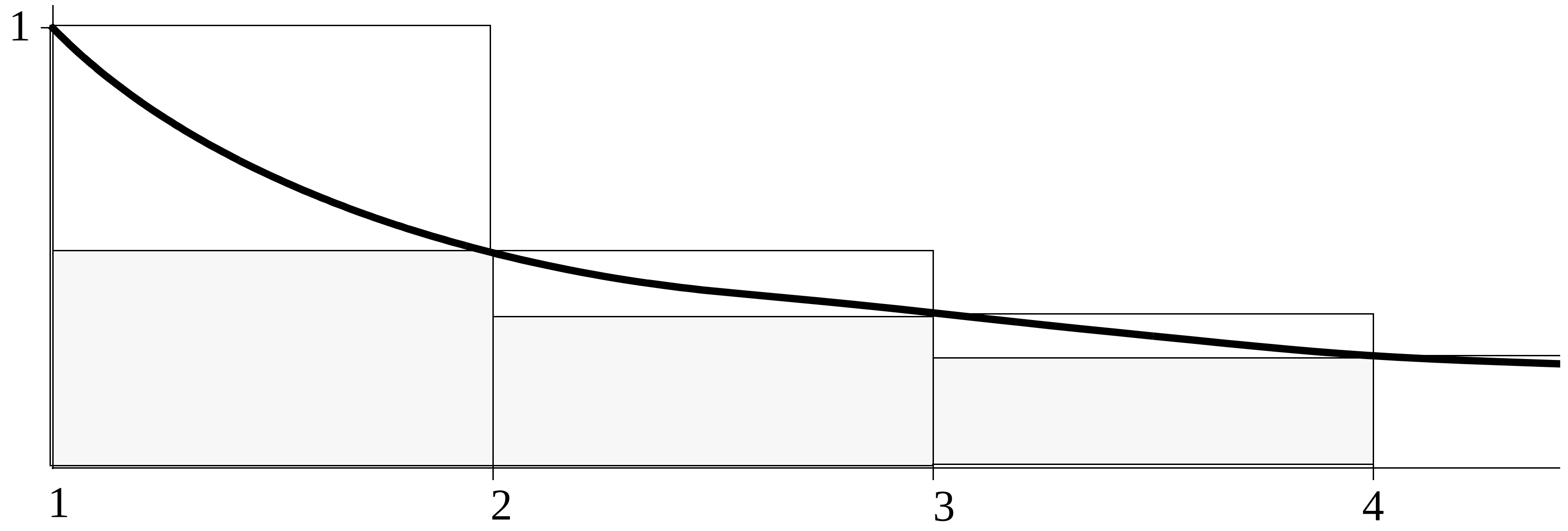
$$D_{avg}(N) \approx 2\frac{(N+1)}{N}\ln(N) - 3$$

- Which is of course $\Theta(\log N)$. In fact, since $\ln N = \ln 2 \cdot \log_2 N$ and $\ln 2 \approx 0.693$, we have, for large N

$$D_{avg}(N) \approx 1.386 \ \log_2 N$$

The approximate solution, more precisely

- We can consider how well the summation is approximated by the integral...
- By inspection of the graph of 1/x, you can see that the integral $\int_1^N (1/x)dx$ underestimates $\sum_{i=1}^N (1/i) - 1/N$, but overestimates $\sum_{i=1}^N (1/i) - 1$



- And therefore

$$\int_1^N \frac{1}{x}dx + 1 \quad > \quad \sum_{i=1}^{N} \frac{1}{i} \quad > \quad \int_1^N \frac{1}{x}dx + \frac{1}{N}$$

The approximate solution, more precisely, cont'd

- And substituting in these bounds we can conclude that

$$D_{avg}(N) < \frac{2(N+1)}{N}(\ln N + 1) - 3$$

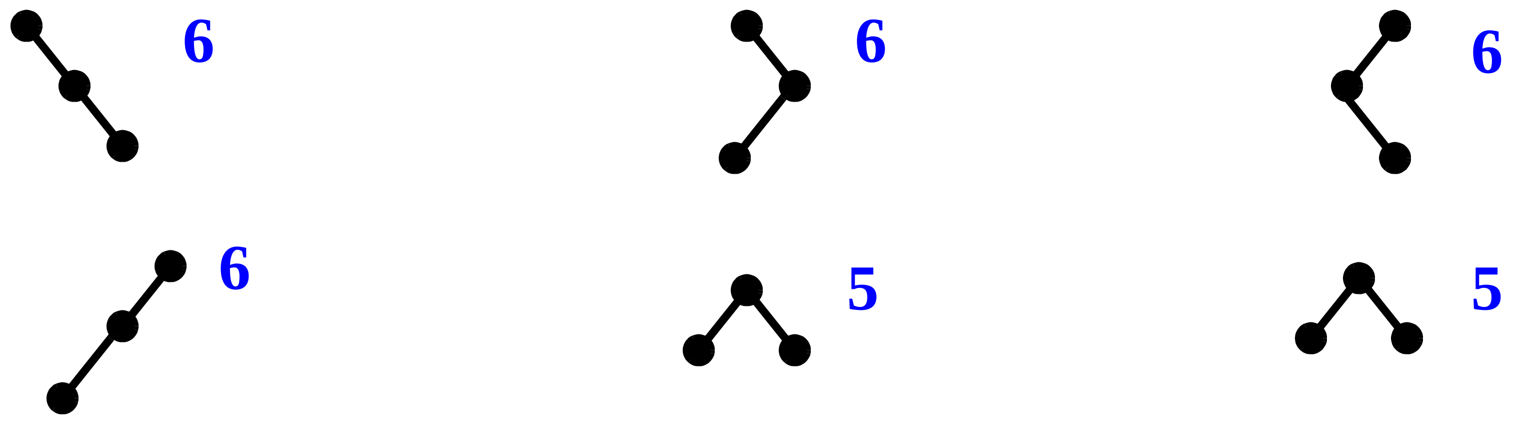
$$D_{avg}(N) > \frac{2(N+1)}{N}\left(\ln N + \frac{1}{N}\right) - 3$$

- These are fairly tight bounds, constraining the average case cost of successful search within an absolute range that approaches 2 as *N* gets large

- We can check these results by inspection for small *N*

Checking the result

- Let $N = 3$. There are $3! = 6$ permutations of the keys
- Here are the shapes of the 6 resulting BST's, each with their total node depth:



- If Assumption #2 holds, the average total node depth D(3) is exactly 34/6
  - (Compare to the summation formula given earlier in these notes)
- If Assumption #1 also holds, the average case number of comparisons for a successful find is $D_{avg}(3) = D(3)/3 = (34/6)/3 = 17/9$ which is about 1.89
- According to the bounds on the previous slide, $D(3)/3 < 2.6$ and $D(3)/3 > 0.81$, which agrees, at least for this small N.
- Yay!

The importance of being balanced

- A binary search tree has *average*-case time cost for Find = $\Theta(\log N)$, but the probabilistic assumptions leading to that result often do not hold in practice
  - For example Assumption #2 may not hold: Approximately sorted input is actually quite likely, leading to unblanced trees with worst-case cost closer to $\Omega(N)$
- But a *balanced* binary search tree has *worst*-case time cost for Find = $\Theta(\log N)$, which is much better than $\Omega(N)$ when N is large
- So, we would like our search trees to be balanced. How to achieve this?
- There are two kinds of approaches to this:
  - Deterministic methods
    - guarantee balance, but operations are somewhat complicated to implement
  - Randomized methods
    - operations are simpler to implement; balance not absolutely guaranteed, but achieved with high probability

Deterministic and randomized balancing

- Deterministic balancing:
  - Use a binary tree; change Insert and Delete to use “rotation” operations so that a balance condition always holds
    - AVL trees, red-black trees, AA trees, etc.
  - Allow nodes in the tree to have more than 2 children; change Insert and Delete to use “node splitting” or “node joining” operations so that a balance condition always holds
    - B trees and their variants, such as 2-3 trees
    - Deterministic skip lists
- Randomized balancing:
  - Use random numbers independent of keys to structure the tree. The result will almost certainly be balanced, no matter how the keys are inserted
    - Randomized search trees (treaps)
    - Skip lists
- We will look at AVL trees first, since the basic techniques there are used in many of the other approaches

AVL trees

- Named for its inventors Adelson-Velskii and Landis, this is the earliest balanced search tree approach [1962]

- An AVL tree is a binary search tree with this balance property:
  - For any node X in the tree, the heights of the left and right subtrees of X differ by at most 1

- From the AVL balance property, you can prove these facts:
  - The number of nodes on a path from the root to a leaf of an AVL tree with N nodes is at most 1.44 $\log_2(N+2)$, compared to at most N for an ordinary BST
  - The average level of a randomly constructed AVL tree with N nodes is asymptotically very close to $\log_2 N$, which is slightly better than the approximately $2 \ln N = 2 \ln 2 \log_2 N = 1.386 \log_2 N$ for a randomly constructed ordinary BST

- The trick is to implement efficient Insert and Delete operations that ensure that the BST ordering and AVL balance properties are *invariant* (they are both pre- and post-conditions of these operations)

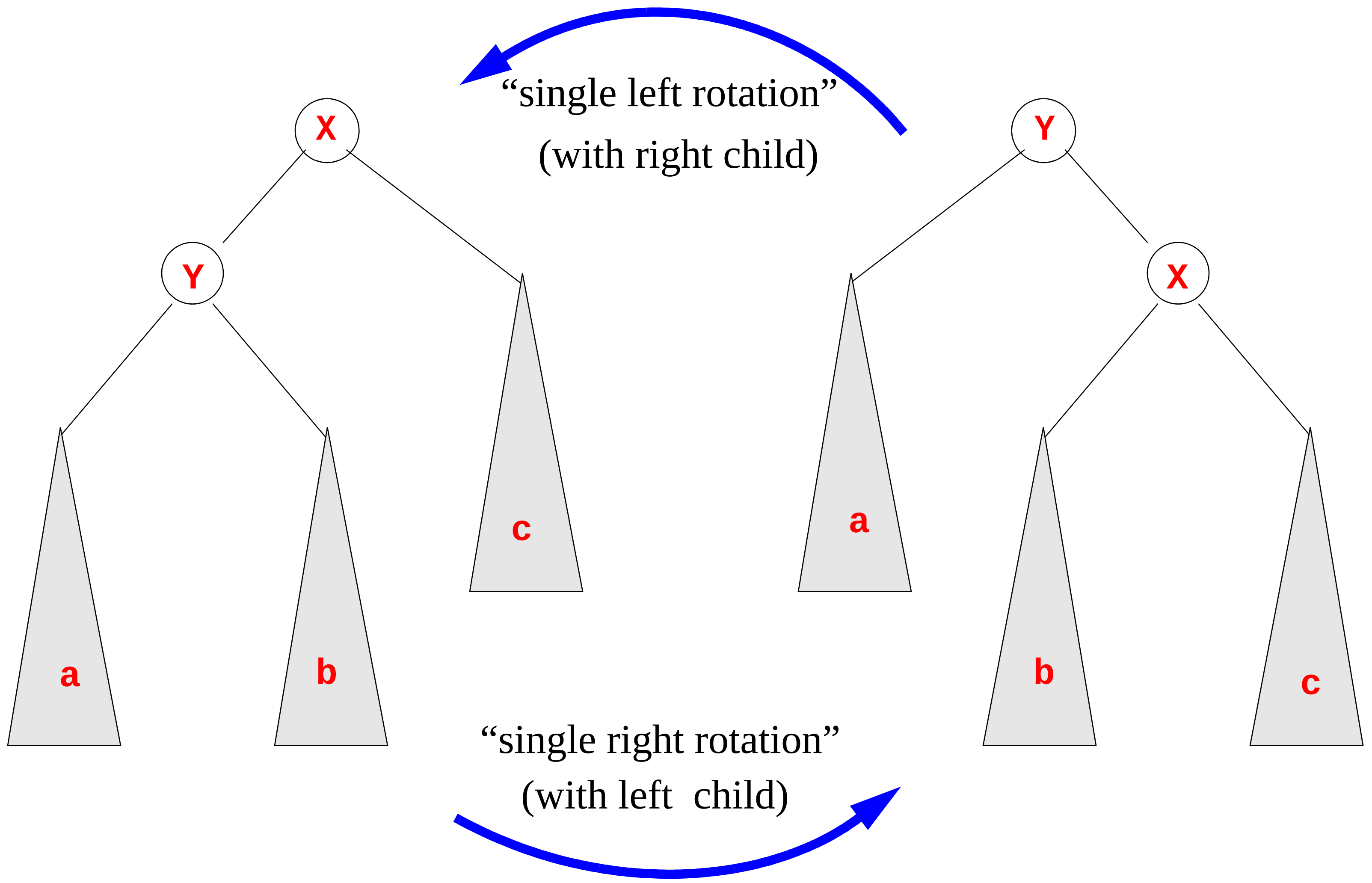
- We will look at what is required for the Insert operation...

AVL rotations

- The key operation in many binary search tree balancing algorithms is the *AVL rotation*
  - Red-black trees, splay trees, treaps, and of course AVL trees all use this
- A crucial nice property of AVL rotations is that they leave the BST structural and ordering properties invariant: performing any AVL rotation on a BST will give you back a BST
- AVL rotations also move some nodes in the tree closer to the root, which can be used to improve balance, if you are careful about where to apply the rotations
- An AVL rotation is fast: constant time, requiring only following and changing a small constant number of pointers
- AVL tree insert and delete operations use AVL single rotations and also double rotations (the double rotations are just two AVL single rotations in sequence)
- We will look at these rotations, why they work, and how to implement them

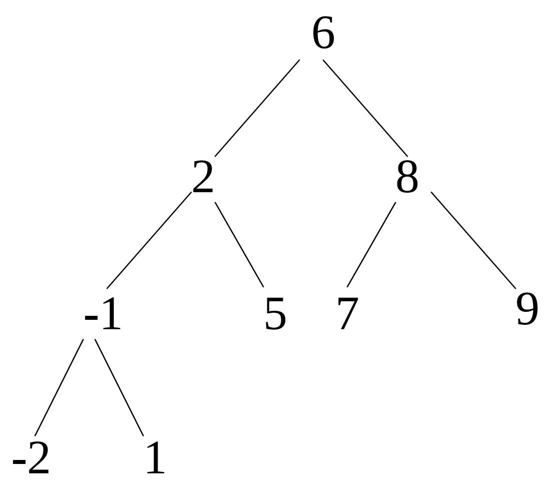
AVL rotations

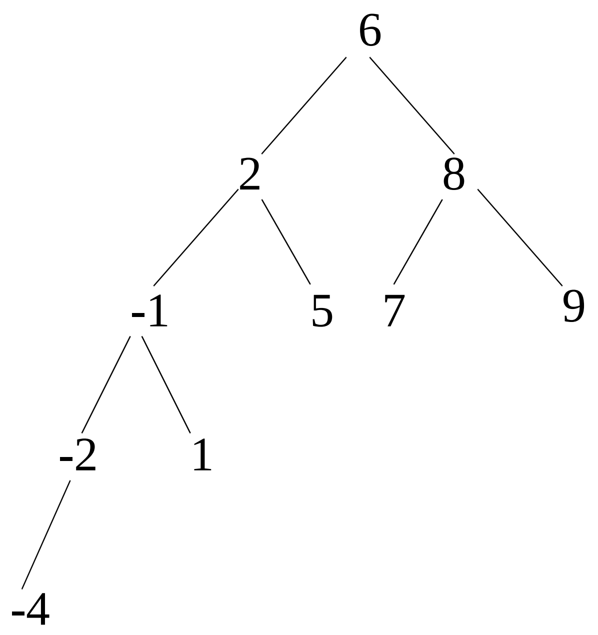
- AVL rotations move subtrees, while preserving BST ordering:



The AVL balance property

- The tree on the left has the AVL balance property
- The tree on the right has inserted a new node using the usual Insert algorithm; this has destroyed the AVL balance property
- Which node was inserted? And which nodes now no longer have the AVL property?





Preserving the AVL balance property

- Suppose you start with a binary search tree that has the AVL balance property
- First, perform an insertion in the usual way: this creates a new leaf node
- If that insertion destroyed the AVL property, the only nodes that no longer have the AVL property are on the path from the new leaf to the root (since those are the only nodes which have had any of their subtrees changed by the insertion)
- So, only nodes on that path need to be rebalanced to restore the tree's AVL property
- We will see that in fact, from that leaf toward the root, only the *first* node failing the AVL property needs to be rebalanced to restore the AVL property to the whole tree
- One of these operations will suffice to rebalance any AVL node after an insertion:
  - single left rotation
  - single right rotation
  - double left rotation
  - double right rotation

Restoring the AVL balance property

- Here’s the basic AVL insert algorithm:
  - Perform the insertion in the usual BST way
  - Moving back up from the newly inserted leaf toward the root, note the first node that no longer has the AVL balance property. (If all still have the property, you’re done!) Call this node X.
  - There are 4 possible places the insertion happened. Depending on the case, perform the appropriate rotation to restore the balance property:
    - (1) Insertion was in the left subtree of the left child of X: single rotation with left child (a.k.a. single right rotation)
    - (2) Insertion was in the right subtree of the left child of X: double rotation with left child (a.k.a double left-right rotation)
    - (3) Insertion was in the left subtree of the right child of X: double rotation with right child (a.k.a. double right-left rotation)
    - (4) Insertion was in the right subtree of the right child of X: single rotation with right child (a.k.a. single left rotation)
- We will look at case (1) and (2); case (3) is symmetric with (2), and (4) is symmetric with (1)

Case 1: insertion in the left subtree of the left child of X

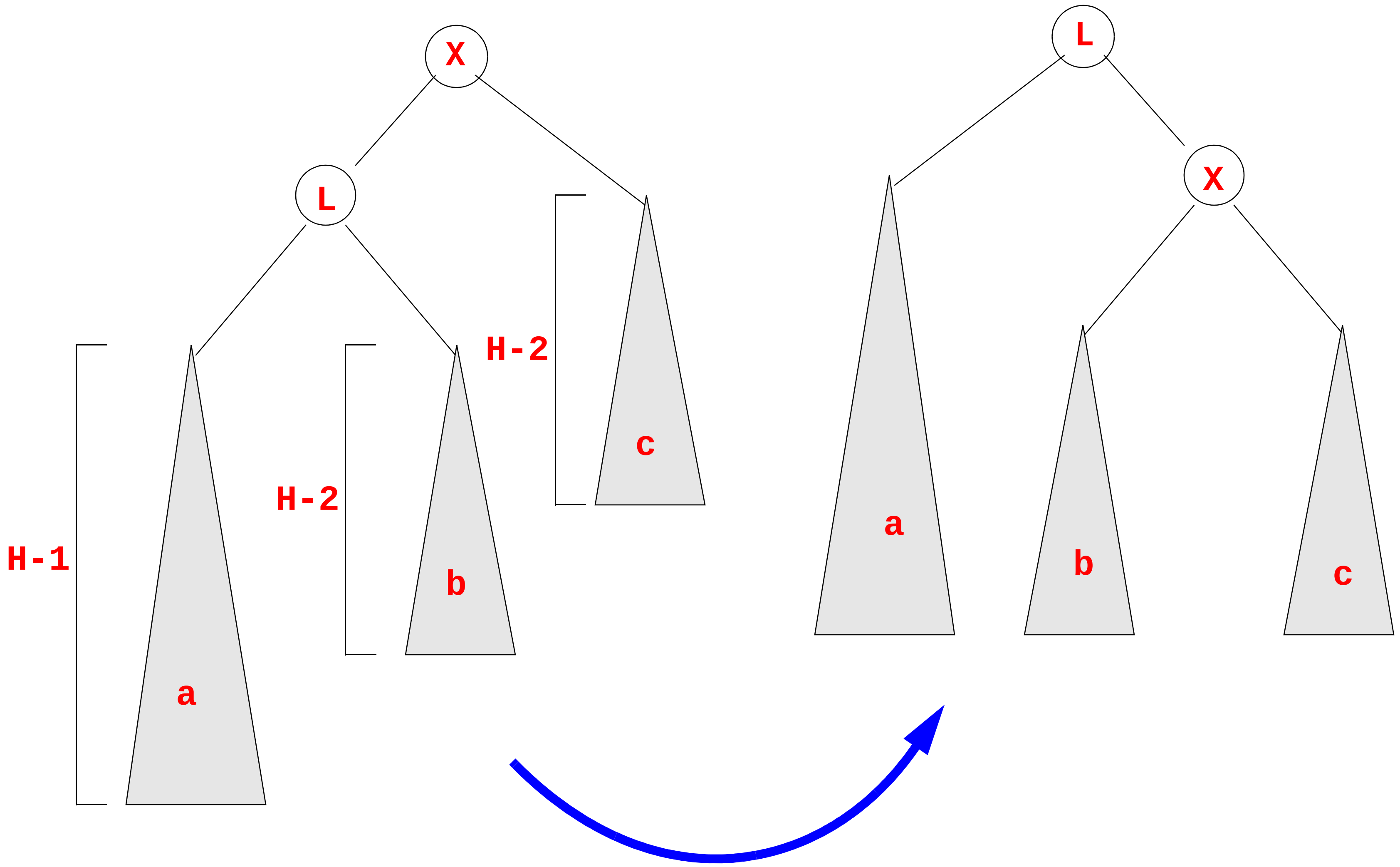
- The insertion occurred somewhere in the left subtree of the left child of X
- Before the insertion, the whole tree had the AVL balance property
- After the insertion, X is the first node on the path from the newly inserted leaf to the root of the tree that fails to have the AVL balance property

- We can conclude that:
  - Before the insertion, the height of X’s left subtree was one greater than the height of X’s right subtree (otherwise the insertion would not have destroyed the AVL property at X)
  - Before the insertion, the height of the 2 subtrees of X’s left child were the same (otherwise the insertion would have not destroyed the AVL property at X, or would have destroyed it at X’s left child instead), and these were also the same as the height of X’s right subtree
  - After the insertion, the height of the left subtree of X’s left child has increased by 1 (otherwise the AVL property would not have been destroyed at X)

Single rotation to handle Case 1

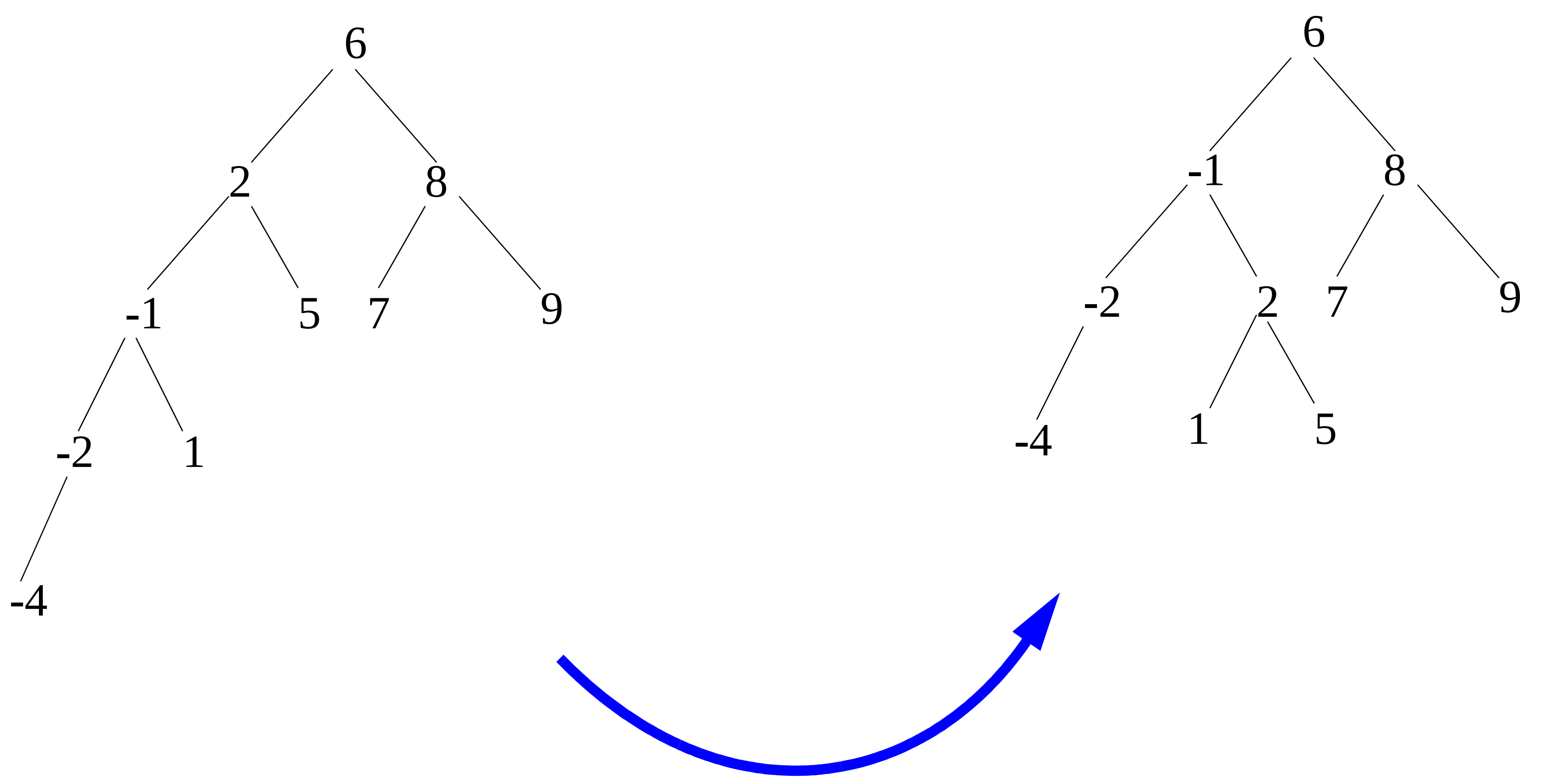
- Let H be the height of the subtree rooted at X before the insertion
- After the insertion:
  - the right subtree of X still has height H-2
  - the right subtree of the left child of X still has height H-2
  - the left subtree of the left child of X now has height H-1
  - and so the height of the subtree rooted at X now has height H+1
- The AVL property at X can be restored by a ‘single rotation with left child’:
  - the left child of X, along with the left child’s left subtree, is made the “new” X
  - former node X, along with its right subtree, is moved to be the right child of the new X
  - the right subtree of the left child of the former node X is moved to be the left subtree of the former node X
- The height of the subtree rooted at the new X is now again H, so no nodes above X need to be changed!
- This maneuver restores the AVL property, *and preserves the BST ordering property*

Single rotation with left child: a picture



CSE 100, UCSD: LEC 4

Single rotation, after insertion: an example



Single rotations: code

```
/** Perform single rotation to handle AVL case 1:
 * AVL violation due to insertion in left subtree of left child.
 * @return pointer to the root of the rotated subtree
 */
BSTNode* rotateWithLeftChild(BSTNode* X) {
   BSTNode* L = X->left;
   X->left = L->right;
   L->right = X;
   return L;
}
/** Perform single rotation to handle AVL case 4:
 * AVL violation due to insertion in right subtree of right child.
 * @return pointer to the root of the rotated subtree
 */
BSTNode* rotateWithRightChild(BSTNode* X) {
   BSTNode R = X->right;
   X->right = R->left;
   R->left = X;
   return R;
}
```

Case 2: insertion in the right subtree of the left child of X

- The insertion occurred somewhere in the right subtree of the left child of X
- Before the insertion, the whole tree had the AVL balance property
- After the insertion, X is the first node on the path from the newly inserted leaf to the root of the tree that fails to have the AVL balance property

- We can conclude that:
    - Before the insertion, the height of X's left subtree was one greater than the height of X's right subtree (otherwise the insertion would not have destroyed the AVL property at X)
    - Before the insertion, the height of the 2 subtrees of X's left child were the same (otherwise the insertion would have not destroyed the AVL property at X, or would have destroyed it at X's left child), and these were also the same as the height of X's right subtree
    - After the insertion, the height of the right subtree of X's left child has increased by 1 (that's what destroyed the AVL property at X)

Double rotation to handle Case 2

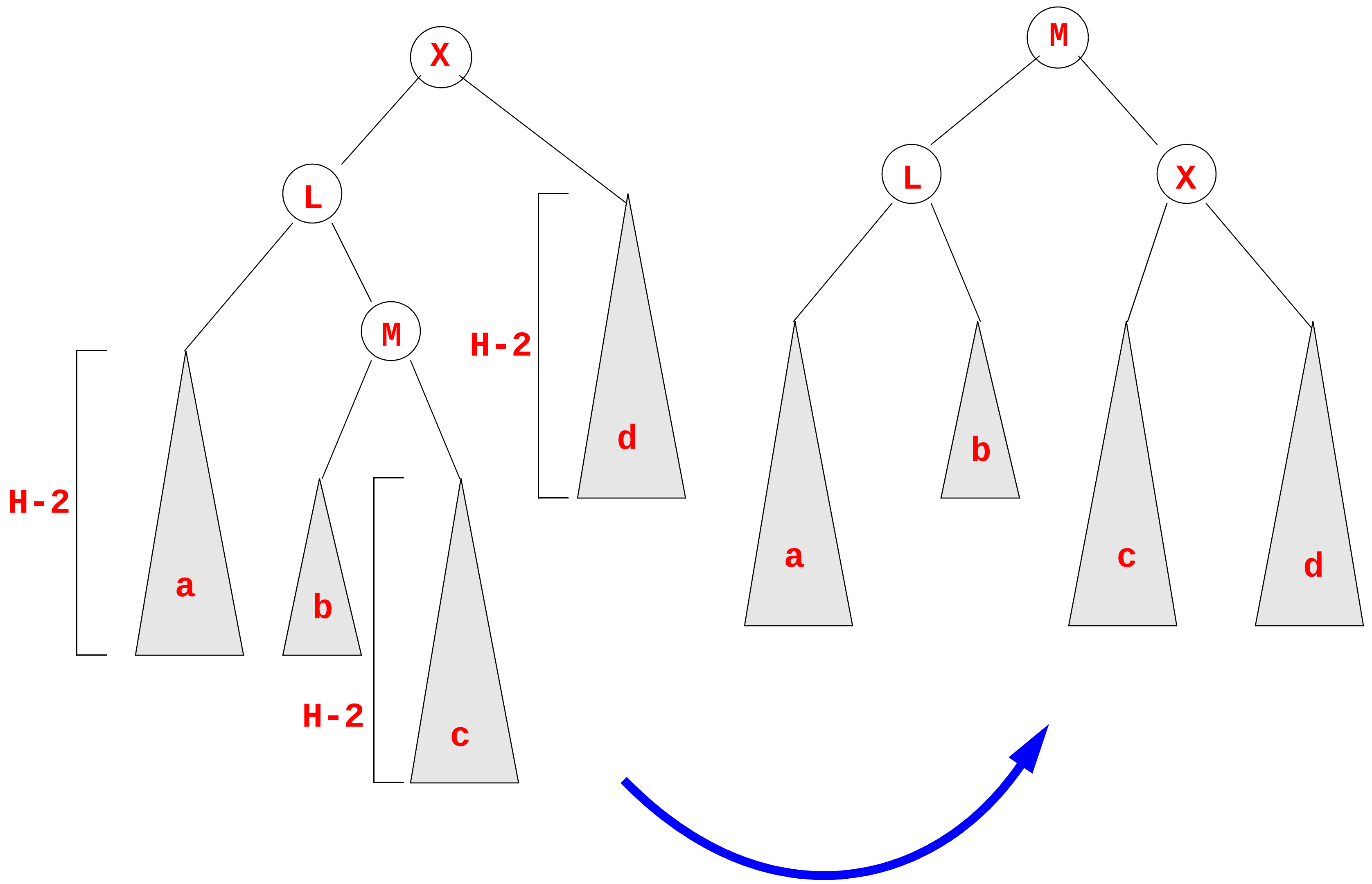
- Let H be the height of the subtree rooted at X before the insertion
- After the insertion:
    - the right subtree of X still has height H-2
    - the left subtree of the left child of X still has height H-2
    - the right subtree of the left child of X now has height H-1
    - one of the subtrees of the right subtree of the of the left child of X now has height H-2 (but it doesn’t matter which)
    - and so the height of the subtree rooted at X now has height H+1

- The AVL property at X can be restored by a ‘double rotation with left child’:
    - first, do a single left rotation rooted at X’s left child
    - then, do a single right rotation rooted at X

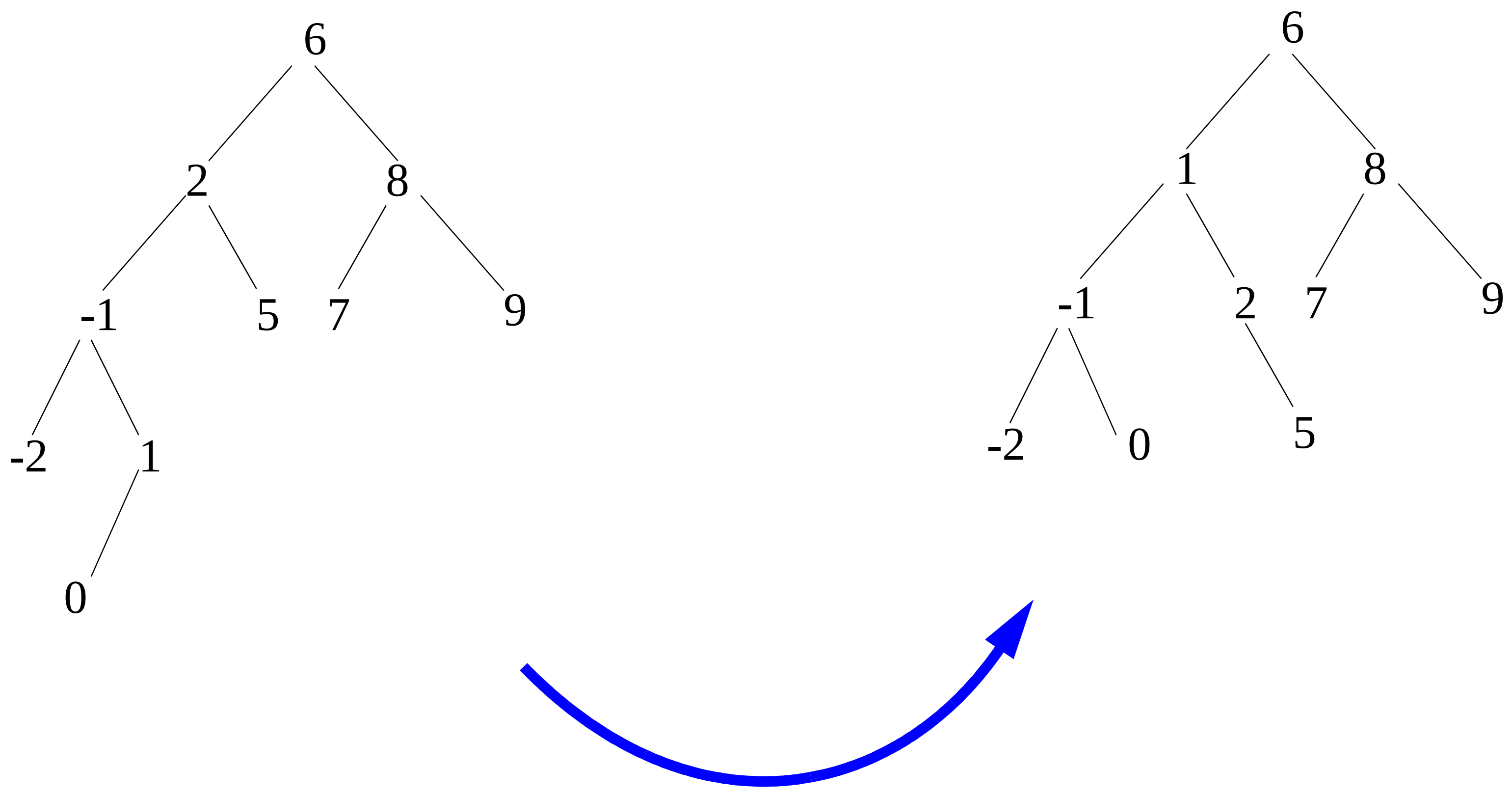
- The height of the subtree rooted at the new X is now again H, so no nodes above X need to be changed!

- This maneuver restores the AVL property, *and preserves the BST ordering property*

Double rotation with left child: a picture



Double rotation, after insertion: an example



Double rotations: code

```
/**
 * Perform double rotation to handle AVL case 2:
 * AVL violation due to insertion in right subtree of left child.
 * @return pointer to the root of the rotated subtree
 */
BSTNode* doubleWithLeftChild(BSTNode* X) {
   X->left = rotateWithRightChild(X->left);
   return rotateWithLeftChild(X);
}

/**
 * Perform double rotation to handle AVL case 3:
 * AVL violation due to insertion in left subtree of right child.
 * @return pointer to the root of the rotated subtree
 */
BSTNode* doubleWithRightChild(BSTNode* X) {
   X->right = rotateWithLeftChild(X->right);
   return rotateWithRightChild(X);
}
```

Implementing AVL operations

- To implement an AVL tree, each node needs to have access to “balance” information
  - each node can have a `height` field, holding the height of the subtree rooted there
  - or, can just hold the difference in height between left and right subtrees of the node

- It is easiest to implement Insert recursively:
  - descend the tree from the root, until reaching the location of the new node
  - create the new node, with height 0, and insert it as a leaf
  - when unwinding the recursion, update height fields of nodes on the path toward the root, and check if a node now violates the AVL balance property
  - if a node does violate the property, perform the appropriate rotation (and update the height fields of affected nodes)

- Implementing a Delete operation is similar: rotations are used to ensure that the AVL balance property is invariant

Next time

- Treaps
- Find, insert, delete, split, and join in treaps
- Randomized search trees
- Randomized search tree time costs

Reading: "Randomized Search Trees" by Aragon & Seidel, *Algorithmica* 1996, `http://sims.berkeley.edu/~aragon/pubs/rst96.pdf`; Weiss, Chapter 12 section 5