Why care about advanced implementations?

Same entries, different insertion sequence:

→ Not good! Would like to keep tree balanced.
Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as \( N-1 \)
- This means that the time needed to perform insertion and deletion and many other operations can be \( O(N) \) in the worst case
- We want a tree with small height
- A binary tree with \( N \) node has height at least \( \Theta(\log N) \)
- Thus, our goal is to keep the height of a binary search tree \( O(\log N) \)
- Such trees are called balanced binary search trees. Examples are AVL tree, and red-black tree.

Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting
Balancing Search Trees

- Many algorithms exist for keeping search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Red-black trees (black nodes balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - Each level of the tree is full except possibly in the bottom right
- This is expensive
  - For example, insert 2 and then rebuild as a complete tree
AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- **Balance factor** of a node
  - $\text{height(left subtree)} - \text{height(right subtree)}$
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Height of an AVL Tree

- $N(h) =$ **minimum** number of nodes in an AVL tree of height $h$.
- **Basic case:**
  - $N(0) = 1$, $N(1) = 2$
- **Inductive case:**
  - $N(h) = N(h-1) + N(h-2) + 1$
- **Theorem** (from Fibonacci analysis)
  - $N(h) \geq \phi^h$
  - where $\phi \approx 1.618$, the golden ratio.
Height of an AVL Tree

- $N(h) > \phi^h$ ($\phi \approx 1.618$)
- Suppose we have $n$ nodes in an AVL tree of height $h$.
  - $n \geq N(h)$ (because $N(h)$ was the minimum)
  - $n > \phi^h$ hence $\log_\phi n > h$ (relatively well balanced tree!!)
  - $h < 1.44 \log_2 n$ (i.e., Find takes $O(\log n)$)

Node Heights

<table>
<thead>
<tr>
<th>Tree A (AVL)</th>
<th>Tree B (AVL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>height=2 BF=1-0=1</td>
<td>height=2</td>
</tr>
<tr>
<td>4 6 0 9</td>
<td>6 2</td>
</tr>
<tr>
<td>1 5</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>5 8</td>
</tr>
</tbody>
</table>

height of node = $h$
balance factor = $h_{left}-h_{right}$
empty height = -1
Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference $h_{left} - h_{right}$) is 2 or –2, adjust tree by rotation around the node
Single Rotation in an AVL Tree

Let the node that needs rebalancing be $\alpha$.

There are 4 cases:

Outside Cases (require single rotation):
1. Insertion into left subtree of left child of $\alpha$. (left-left)
2. Insertion into right subtree of right child of $\alpha$. (right-right)

Inside Cases (require double rotation):
3. Insertion into right subtree of left child of $\alpha$. (left-right)
4. Insertion into left subtree of right child of $\alpha$. (right-left)

The rebalancing is performed through four separate rotation algorithms.
Consider a valid AVL subtree.

Inserting into X destroys the AVL property at node j.
AVL Insertion: Outside Case

Do a “rotation to right”

Single right rotation

Do a “right rotation”
Outside Case Completed

“Right rotation” done!
(“Left rotation” is mirror symmetric)

AVL property has been restored!

AVL Insertion: Inside Case

Consider a valid AVL subtree

AVL property has been restored!
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j

Does “right rotation” restore balance?

"One rotation" does not restore balance… now k is out of balance
AVL Insertion: Inside Case

Consider the structure of subtree Y...

Y = node i and subtrees V and W
AVL Insertion: Inside Case

We will do a left-right “double rotation” . . .

Double rotation : first rotation

left rotation complete
Double rotation: second rotation

Now do a right rotation.

Double rotation: second rotation

Right rotation complete.

Balance has been restored.
Implementation

Once you have performed a rotation (single or double) you won't need to go back up the tree.

Class BinaryNode
  KeyType: Key
  int: Height
  BinaryNode: LeftChild
  BinaryNode: RightChild

  Constructor(KeyType: key)
    Key = key
    Height = 0
  End Constructor

End Class

rotateToRight(G)

Relative to G, X is at left-left positions. rotateToRight(G) will exchange of roles between G and P, so P becomes G's parent.
After rotateToRight(G)

rotateToLeft(G) will handle the case when X is at right-right position relative to G.

Java-like Pseudo-Code

```java
rotateToRight(BinaryNode: x) {
    BinaryNode y = x.LeftChild;
    x.LeftChild = y.RightChild;
    y.RightChild = x;
    return y;
}
```
Java-like Pseudo-Code

rotateToLeft(BinaryNode: x) {
    BinaryNode y = x.rightChild;
    x.rightChild = y.leftChild;
    y.leftChild = x;
    return y;
}

Double Rotation

- Implement Double Rotation in two lines.

DoubleRotateToLeft(n : binaryNode) {
    rotateToRight(n.rightChild);
    rotateToLeft(n);
}

DoubleRotateToRight(n : binaryNode) {
    rotateToLeft(n.leftChild);
    rotateToRight(n);
}
Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference $h_{left} - h_{right}$) is 2 or −2, adjust tree by *rotation* around the node

Insert in ordinary BST

Algorithm `insert(k, v)`

- **input:** insert key $k$ into the tree rooted by $v$
- **output:** the tree root with $k$ adding to $v.$
- if `isNull(v)`
  - return `newInternalNode(k)`
- if $k \leq \text{key}(v)$  // duplicate keys are okay
  - $\text{leftChild}(v) \leftarrow \text{insert}(k, \text{leftChild}(v))$
- else if $k > \text{key}(v)$
  - $\text{rightChild}(v) \leftarrow \text{insert}(k, \text{rightChild}(v))$
- return $v$
Insert in AVL trees

Insert(v : binaryNode, x : element) : {
    if v = null then
        {v  new node; v.data  x; height  0;}
    else case
        v.data = x : ; //Duplicate do nothing
        v.data > x : v.leftChild  Insert(v.leftChild, x);
            // handle left-right and left-left cases
            if ((height(v.leftChild)- height(v.rightChild)) = 2)then
                    if (v.leftChild.data > x ) then //outside case
                        v = RotateToRight (v);
                    else //inside case
                        v = DoubleRotateToRightt (v);}
        v.data < x : v.rightChild  Insert(v.rightChild, x);
            // handle right-right and right-left cases
                    ...
    Endcase
        v.height  max(height(v.left),height(v.right)) +1;
    return v;
}

Example of Insertions in an AVL Tree

Insert 5, 40
Example of Insertions in an AVL Tree

Single rotation (outside case)
Double rotation (inside case)

AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.
Deletion

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent may have an imbalance.

- Example:

```
  44
  /   \
 17    62
  / \   / \ 
32  50 78 88
  / \   / \ 
48 54 48 54
```

before deletion of 32 after deletion

Rebalancing after a Removal

- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.

- We perform a rotateToRight to restore balance at z.

- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached.
Deletion in standard BST

Algorithm `remove(k, v)`

**input:** delete the node containing key $k$

**output:** the tree without $k$. 

if `isNull(v)`
    return v

if $k < \text{key}(v)$  // duplicate keys are okay
    leftChild(v) $\leftarrow$ `remove(k, leftChild(v))`
else if $k > \text{key}(v)$
    rightChild(v) $\leftarrow$ `remove(k, rightChild(v))`
else if `isNull(leftChild(v))`
    return rightChild(v)
else if `isNull(rightChild(v))`
    return leftChild(v)

node max $\leftarrow$ `treeMaximum(leftChild(v))`
key(v) $\leftarrow$ `key(min)`
rightChild(v) $\leftarrow$ `remove(key(min), rightChild(v))`

return v

Deletion in AVL Trees

Algorithm `remove(k, v)`

**input:** delete the node containing key $k$

**output:** the tree without $k$. 

if `isNull(v)`
    return v

if $k < \text{key}(v)$  // duplicate keys are okay
    leftChild(v) $\leftarrow$ `remove(k, leftChild(v))`
else if $k > \text{key}(v)$
    rightChild(v) $\leftarrow$ `remove(k, rightChild(v))`
else if `isNull(leftChild(v))`
    return rightChild(v)
else if `isNull(rightChild(v))`
    return leftChild(v)

node max $\leftarrow$ `treeMaximum(leftChild(v))`
key(v) $\leftarrow$ `key(max)`
leftChild(v) $\leftarrow$ `remove(key(max), leftChild(v))`

`AVLbalance(v)`

Assume the height is updated in rotations.

if (v.left.height > v.right.height+1) {
    y = v.left
    if (y.right.height > y.left.height) 
        DoubleRotateToRight(v)
    else rotateToRight(v)
}

if (v.right.height > v.left.height+1) {
    y = v.right
    if (y.left.height > y.right.height)
        DoubleRotateToLeft(v)
    else rotateToLeft(v)
}
AVL Tree Example:

• Now remove 53

AVL Tree Example:

• Now remove 53, unbalanced
AVL Tree Example:

- Balanced!

Now try Remove 11

AVL Tree Example:

- Remove 11, replace it with the largest, i.e., 8, in its left branch.

Now try Remove 8.
AVL Tree Example:

- Remove 8, unbalanced
AVL Tree Example:
• Balanced!!

In Class Exercises
- Build an AVL tree with the following values:
  15, 20, 24, 10, 13, 7, 30, 36, 25
Deletion in AVL Trees

Algorithm `remove(k, v)`
- **input:** delete the node containing key `k`
- **output:** the tree without `k`

```plaintext
if isNull(v)
    return v
if k < key(v)  // duplicate keys are okay
    leftChild(v) ← remove(k, leftChild(v))
else if k > key(v)
    rightChild(v) ← remove(k, rightChild(v))
else if isNull(leftChild(v))
    return rightChild(v)
else if isNull(rightChild(v))
    return leftChild(v)
node max ← treeMaximum(leftChild(v))
key(v) ← key(max)
leftChild(v) ← remove(key(max), leftChild(v))
AVLbalance(v)
return v
```

`AVLbalance(v)`
Assume the height is updated in rotations.

```plaintext
if (v.left.height > v.right.height+1) {
    y = v.left
    if (y.right.height > y.left.height)
        DoubleRotateToRight(v)
    else  rotateToRight(v)
}
if (v.right.height > v.left.height+1) {
    y = v.right
    if (y.right.height > y.left.height)
        DoubleRotateToLeft(v)
    else  rotateToLeft(v)
}
```
Remove 24 and 20 from the AVL tree.

AVL Tree Performance

- AVL tree storing n items
  - The data structure uses $O(n)$ space
  - A single restructuring takes $O(1)$ time
    - using a linked-structure binary tree
  - Searching takes $O(\log n)$ time
    - height of tree is $O(\log n)$, no restructures needed
  - Insertion takes $O(\log n)$ time
    - initial find is $O(\log n)$
    - restructuring up the tree, maintaining heights is $O(\log n)$
  - Removal takes $O(\log n)$ time
    - initial find is $O(\log n)$
    - restructuring up the tree, maintaining heights is $O(\log n)$
Pros and Cons of AVL Trees

Arguments for AVL trees:
1. Search is $O(\log N)$ since AVL trees are always balanced.
2. Insertion and deletions are also $O(\log n)$.
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:
1. Difficult to program & debug; more space for height.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have $O(N)$ for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Red-Black Tree

- A ref-black tree is a binary search such that each node has a color of either red or black.
- The root is black.
- External (or null) nodes are black.
- Every path from a node to a leaf contains the same number of black nodes.
- If a node is red then its parent must be black.

Class BinaryNode

```
Class BinaryNode
    KeyType: Key
    Boolean: isRed
    BinaryNode: LeftChild
    BinaryNode: RightChild

Constructor(KeyType: key)
    Key = key
    isRed = true
End Constructor

End Class
```
Theorem: Any red-black tree with root \( x \), has \( n \geq 2^{h/2} - 1 \) nodes, where \( h \) is the height of tree rooted by \( x \).

Proof: We repeatedly replace the subtree rooted by a red node by one of its children.

Let the height of the new tree be \( h' \), then \( h' \geq h/2 \), because the number of red nodes in any path is no more than the number of black nodes.

The new tree is a perfect binary tree, because it has the same of nodes from the root to any leaf. It must have \( 2^{h' - 1} \) nodes.

So \( h \leq 2\log(n+1) \).
Maintain the Red Black Properties in a Tree

- **Insertions**
  - Must maintain rules of Red Black Tree.
  - New Node always added at leaf
  - can't be black or we will violate rule of the same # of blacks along any path
  - therefore the new node must be red
  - If parent is black, done (trivial case)
  - If parent red, things get interesting because a red node with a red parent violates no double red rule.

Algorithm: Insertion

A red-black tree is a particular binary search tree, so create a new node as red and insert it as in normal search tree.

Violation!

What property may be violated? The parent of a red node must be black.

Solution: (1) Rotate; (2) Switch colors.
Example of Inserting Sorted Numbers

- 1 2 3 4 5 6 7 8 9 10

Insert 1. A leaf is red. Realize it is root so recolor to black.

Insert 2

make 2 red. Parent is black so done.
**Insert 3**

Insert 3. Parent is red. 2’s uncle, i.e., the sibling of the parent of 2, is black (null). 3 is outside relative to grandparent. Rotate parent and grandparent.

**Insert 4**

When adding 4 parent is red. 4 has a red uncle (1). So switch the great parent (2)’s color with parent and uncle. 2 is set to black if it’s the root.
Insert 5

5's parent is red.
5's uncle is black (null).
5 is outside relative to grandparent (3) so rotate parent and grandparent then recolor.

Finish insert of 5

71

72
Insert 6

6 has a red uncle (3). So switch the grandparent (4)’s color with parent (5) and uncle (3).

Finishing insert of 6

4’s parent is black so done.
7's parent is red. Parent's sibling is black (null). 7 is outside relative to grandparent (5) so rotate parent and grandparent then recolor

Finish insert of 7
8’s parent is red and its uncle (5) is also red. Switching the color of 6 with 5 and 7 creates a problem because 6’s parent, 4, is also red. Must handle the red-red violation at 6.

6’s uncle (1) is black. So rotate and recolor.

8’s parent is red and its uncle (5) is also red. Switching the color of 6 with 5 and 7 creates a problem because 6’s parent, 4, is also red. Must handle the red-red violation at 6.
Finish inserting 8

Insert 9
After rotations and recoloring 10 has a red uncle. Grandparent (8) switch colors with parent (9) and uncle (7).
8 has a red uncle (2). Grandparent (4) switch colors with parent (2) and uncle (6). 4 is recolored black as root.

Finishing Insert 10
Algorithm: Insertion

We have detected a need for balance when $X$ is red and its parent, too.

- If $X$ has a red uncle: colour the parent and uncle black, and grandparent red. Then replace $X$ by grandparent to see if new $X$’s parent is red.
Algorithm: Insertion

We have detected a need for balance when $X$ is red and his parent too.

- If $X$ has a red uncle: colour the parent and uncle black, and grandparent red. Then replace $X$ by grandparent to see if new $X$’s parent is red.
- If $X$ is a left child and has a black uncle: colour the parent black and the grandparent red, then rotateToRight($X$.parent.parent)

Algorithm: Insertion

We have detected a need for balance when $X$ is red and his parent too.

- If $X$ has a red uncle: colour the parent and uncle black, and grandparent red. Then replace $X$ by grandparent to see if $X$’s parent is red.
- If $X$ is a left child and has a black uncle: colour the parent black and the grandparent red, then rotateRight($X$.parent.parent)
Algorithm: Insertion

We have detected a need for balance when **X** is red and his parent too.

- If **X** has a red uncle: colour the parent and uncle black, and grandparent red. Then replace **X** by grandparent to see if **X**'s parent is red.
- If **X** is a left child and has a black uncle: colour the parent black and the grandparent red, then rotateRight(**X**.parent.parent)
- If **X** is a right child and has a black uncle, then rotateToLeft(**X**.parent) and **X**
**Algorithm: Insertion**

We have detected a need for balance when \( X \) is red and his parent too.

- If \( X \) has a red uncle: colour the parent and uncle black, and grandparent red. Then replace \( X \) by grandparent to see if \( X \)'s parent is red.
- If \( X \) is a right child and has a black uncle, then rotateLeft(\( X \).parent) and
- If \( X \) is a left child and has a black uncle: colour the parent black and the grandparent red, then rotateRight(\( X \).parent.parent)

**Double Rotation**

- What if \( X \) is at left right relative to \( G \)?
  - a single rotation will not work
- Must perform a double rotation
  - rotate \( X \) and \( P \)
  - rotate \( X \) and \( G \)
Double rotation is also needed when $X$ is at right left position relative to $G$. 

Properties of Red Black Trees

- If a Red node has any children, it must have two children and they must be black. (Why?)
- If a black node has only one child that child must be a Red leaf. (Why?)
- Due to the rules there are limits on how unbalanced a Red Black tree may become.
Motivation for Splay Trees

Problems with AVL Trees
- extra storage/complexity for height fields
- ugly delete code

Solution: splay trees
- blind adjusting version of AVL trees
- amortized time for all operations is $O(\log n)$
- worst case time is $O(n)$
- insert/find always rotates node to the root!
Splay Tree Idea

You’re forced to make a really deep access:

Since you’re down there anyway, fix up a lot of deep nodes!

Splaying Cases

Node n being accessed is:

- Root
- Child of root
- Has both parent (p) and grandparent (g)
  - Zig-zig pattern: g → p → n is left-left or right-right (outside nodes)
  - Zig-zag pattern: g → p → n is left-right or right-left (inside nodes)
Access root:
Do nothing (that was easy!)

Access child of root:
Zig (AVL single rotation)
Access (LR, RL) grandchild: Zig-Zag (AVL double rotation)

Access (LL, RR) grandchild: Zig-Zig

Rotate top-down – why?
Splaying Example:

Find(6)

... still splaying ...
... 6 splayed out!

Splay it Again, Sam!

Find (4)
Splay Tree Definition

- **A splay tree** is a binary search tree where a node is splayed after it is accessed (for a search or update)
  - deepest internal node accessed is splayed
  - splaying costs $O(h)$, where $h$ is height of the tree – which is still $O(n)$ worst-case
    - $O(h)$ rotations, each of which is $O(1)$
Splay Trees do Rotations after Every Operation (Even Search)

- new operation: **splay**
  - splaying moves a node to the root using rotations

  - **right rotation**
    - makes the left child $x$ of a node $y$ into $y$’s parent; $y$ becomes the right child of $x$

  - **left rotation**
    - makes the right child $y$ of a node $x$ into $x$’s parent; $x$ becomes the left child of $y$

Visualizing the Splaying Cases
Splaying:

- "x is a left-left grandchild" means x is a left child of its parent, which is itself a left child of its parent
- p is x’s parent; g is p’s parent

- is x the root?
  - yes: stop
  - no: is x a child of the root?
    - yes: is x the left child of the root?
      - yes: zig
      - no: zig
    - no: zig-zag
    - right-rotate about the root
- left-rotate about the root
- right-rotate about g, right-rotate about p
- zig-zig
- zig-zag
- left-rotate about g, left-rotate about p
- left-rotate about p, right-rotate about g
- zig-zag

Splay Tree Operations

- Which nodes are splayed after each operation?

<table>
<thead>
<tr>
<th>method</th>
<th>splay node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search for k</td>
<td>if key found, use that node</td>
</tr>
<tr>
<td></td>
<td>if key not found, use parent of ending external node</td>
</tr>
<tr>
<td>Insert (k,v)</td>
<td>use the new node containing the entry inserted</td>
</tr>
<tr>
<td>Remove item</td>
<td>use the predecessor of the node to be removed</td>
</tr>
<tr>
<td>with key k</td>
<td></td>
</tr>
</tbody>
</table>

111

112
Why Splaying Helps

- If a node \( n \) on the access path is at depth \( d \) before the splay, it’s at about depth \( d/2 \) after the splay
  - Exceptions are the root, the child of the root, and the node splayed

- Overall, nodes which are below nodes on the access path tend to move closer to the root

- Splaying gets amortized \( O(\log n) \) performance. (Maybe not now, but soon, and for the rest of the operations.)

Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root
Splay Operations: Insert

- Ideas?
- Can we just do BST insert?

Digression: Splitting

- Split(T, x) creates two BSTs L and R:
  - all elements of T are in either L or R \((T = L \cup R)\)
  - all elements in L are \(\leq x\)
  - all elements in R are \(\geq x\)
  - L and R share no elements \((L \cap R = \emptyset)\)
Splitting in Splay Trees

How can we split?
- We have the splay operation.
- We can find x or the parent of where x should be.
- We can splay it to the root.
- Now, what’s true about the left subtree of the root?
- And the right?

Split

\[
\text{split}(x)
\]

\[
\begin{array}{c}
T \\
L & R
\end{array}
\]

\[
\begin{array}{c}
L \\
\leq x
\end{array}
\quad
\begin{array}{c}
> x
\end{array}
\quad
\begin{array}{c}
l \\
< x
\end{array}
\quad
\begin{array}{c}
g \\
\geq x
\end{array}
\]
void insert(Node root, Object x) {
    <left, right> = split(root, x);
    root = new Node(x, left, right);
}

Splay Operations: Delete

Now what?
Join

Join(L, R): given two trees such that L < R, merge them

Splay on the maximum element in L, then attach R

Delete Completed

find(x) delete x Join(L, R)

T - x
Splay Tree Summary

Can be shown that any $M$ consecutive operations starting from an empty tree take at most $O(M \log(N))$

$\rightarrow$ All splay tree operations run in amortized $O(\log n)$ time

$O(N)$ operations can occur, but splaying makes them infrequent

Implements most-recently used (MRU) logic
- Splay tree structure is self-tuning

Splay Tree Summary (cont.)

Splaying can be done top-down; better because:
- only one pass
- no recursion or parent pointers necessary

There are alternatives to split/insert and join/delete

Splay trees are very effective search trees
- relatively simple: no extra fields required
- excellent locality properties:
  - frequently accessed keys are cheap to find (near top of tree)
  - infrequently accessed keys stay out of the way (near bottom of tree)
Amortized Analysis of Splay Trees

- Running time of each operation is proportional to time for splaying.
- Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v:
  - rank(v) = \log n(v) if null for external nodes
  - rank(v) = \log (2n(v)+1) if empty nodes for externals.
- Costs: zig = $1, zig-zig = $2, zig-zag = $2.
- Thus, cost for splaying a node at depth d = $d.
- Imagine that we store rank(v) cyber-dollars at each node v of the splay tree (just for the sake of analysis).
- The total counter values is rank(T) = sum of rank(v) for any node v in T.