Outline

Dictionaries and Balanced Search Trees
- ADT Dictionary
- Review: Binary Search Trees
- AVL Trees
- Insertion in AVL Trees
- Restoring the AVL Property: Rotations
Outline

1. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Review: Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restoring the AVL Property: Rotations
Dictionary ADT

**Dictionary**: An ADT consisting of a collection of items, each of which contains
- a *key*
- some *data* (the “value”)

and is called a *key-value pair* (KVP). Keys can be compared and are (typically) unique.

Operations:
- `search(k)` (also called `findElement(k)`)  
- `insert(k, v)` (also called `insertItem(k, v)`)  
- `delete(k)` (also called `removeElement(k)`)
- optional: `closestKeyBefore`, `join`, `isEmpty`, `size`, etc.
Elementary Implementations

Common assumptions:

- Dictionary has $n$ KVPs
- Each KVP uses constant space
- Keys can be compared in constant time

**Unordered array or linked list**

- search $\Theta(n)$
- insert $\Theta(1)$ (except array occasionally needs to resize)
- delete $\Theta(n)$ (need to search)

**Ordered array**

- search $\Theta(\log n)$ (via binary search)
- insert $\Theta(n)$
- delete $\Theta(n)$
Outline

1. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Review: Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restoring the AVL Property: Rotations
Binary Search Trees (review)

Structure  Binary tree: all nodes have two (possibly empty) subtrees
          Every node stores a KVP
          Empty subtrees usually not shown

Ordering  Every key \( k \) in \( T.left \) is less than the root key.
          Every key \( k \) in \( T.right \) is greater than the root key.

In our examples we only show the keys, and we show them directly in the
node. A more accurate picture would be

\[
\text{key } = 15, \quad \text{<other info>}
\]
BST as realization of ADT Dictionary

\textit{BST::search}(k) Start at root, compare \( k \) to current node’s key. Stop if found or subtree is empty, else recurse at subtree.

\textit{BST::insert}(k, v) Search for \( k \), then insert \((k, v)\) as new node

Example:
Deletion in a BST

- First search for the node $x$ that contains the key.
- If $x$ is a leaf (both subtrees are empty), delete it.
- If $x$ has one non-empty subtree, move child up.
- Else, swap key at $x$ with key at successor or predecessor node and then delete that node.
Height of a BST

\[ \text{BST::search, BST::insert, BST::delete} \] all have cost \( \Theta(h) \), where \( h = \text{height of the tree} = \text{max. path length from root to leaf} \)

If \( n \) items are inserted one-at-a-time, how big is \( h \)?

- **Worst-case:** \( n - 1 = \Theta(n) \)
- **Best-case:** \( \Theta(\log n) \).

Any binary tree with \( n \) nodes has height \( \geq \log(n + 1) - 1 \)

- **Average-case:** Can show \( \Theta(\log n) \)
Outline

Dictionaries and Balanced Search Trees
- ADT Dictionary
- Review: Binary Search Trees
- AVL Trees
- Insertion in AVL Trees
- Restoring the AVL Property: Rotations
AVL Trees

Introduced by Adel’son-Vel’skiĭ and Landis in 1962, an AVL Tree is a BST with an additional **height-balance property** at every node:

*The heights of the left and right subtree differ by at most 1.*

(The height of an empty tree is defined to be $-1$.)

Rephrase: If node $v$ has left subtree $L$ and right subtree $R$, then

\[
\text{balance}(v) := \text{height}(R) - \text{height}(L) \quad \text{must be in }\{ -1, 0, 1 \}
\]

- $\text{balance}(v) = -1$ means $v$ is *left-heavy*
- $\text{balance}(v) = +1$ means $v$ is *right-heavy*

- Need to store at each node $v$ the height of the subtree rooted at it
- Can show: It suffices to store $\text{balance}(v)$ instead
  - uses fewer bits, but code gets more complicated
AVL tree example

(The lower numbers indicate the height of the subtree.)
AVL tree example

Alternative: store balance (instead of height) at each node.
Height of an AVL tree

**Theorem:** An AVL tree on $n$ nodes has $\Theta(\log n)$ height.

⇒ *search, insert, delete* all cost $\Theta(\log n)$ in the *worst case*!

**Proof:**

- Define $N(h)$ to be the *least* number of nodes in a height-$h$ AVL tree.
- What is a recurrence relation for $N(h)$?
- What does this recurrence relation resolve to?
Outline

1. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Review: Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restoring the AVL Property: Rotations
AVL insertion

To perform $AVL::insert(k, v)$:

- First, insert $(k, v)$ with the usual BST insertion.
- We assume that this returns the new leaf $z$ where the key was stored.
- Then, move up the tree from $z$, updating heights.
  - We assume for this that we have parent-links. This can be avoided if $BST::Insert$ returns the full path to $z$.
- If the height difference becomes $\pm 2$ at node $z$, then $z$ is unbalanced. Must re-structure the tree to rebalance.
AVL insertion

\[AVL::insert(k, v)\]
1. \[z \leftarrow BST::insert(k, v)\] // leaf where \(k\) is now stored
2. \textbf{while} (\(z\) is not \textsc{Nil})
3. \textbf{if} \(|z\cdot\text{left.height} - z\cdot\text{right.height}| > 1\) \textbf{then}
4. Let \(y\) be taller child of \(z\)
5. Let \(x\) be taller child of \(y\)
6. \[z \leftarrow \text{restructure}(x, y, z)\] // see later
7. \textbf{break} // can argue that we are done
8. \[\text{setHeightFromSubtrees}(z)\]
9. \[z \leftarrow z\cdot\text{parent}\]

\[\text{setHeightFromSubtrees}(u)\]
1. \[u\cdot\text{height} \leftarrow 1 + \max\{u\cdot\text{left.height}, u\cdot\text{right.height}\}\]
AVL Insertion Example

Example:

```
22
  4
  10
    3
    4
     1
      6
      13
      18
      16
      0

  31
    2
    28
      0

37
  1
  46

0
1
2
3
```

Biedl, Schost, Veksler  (SCS, UW)
Outline

1 Dictionaries and Balanced Search Trees
   • ADT Dictionary
   • Review: Binary Search Trees
   • AVL Trees
   • Insertion in AVL Trees
   • Restoring the AVL Property: Rotations
How to “fix” an unbalanced AVL tree

**Note:** there are many different BSTs with the same keys.

**Goal:** change the *structure* among three nodes without changing the *order* and such that the subtree becomes balanced.
Right Rotation

This is a right rotation on node z:

\[
\text{rotate-right}(z) \\
1. y \leftarrow z.\text{left}, \quad z.\text{left} \leftarrow y.\text{right}, \quad y.\text{right} \leftarrow z \\
2. \text{setHeightFromSubtrees}(z), \quad \text{setHeightFromSubtrees}(y) \\
3. \text{return } y \quad // \quad \text{returns new root of subtree}
\]
Why do we call this a rotation?
Left Rotation

Symmetrically, this is a **left rotation** on node $z$:

Again, only two links need to be changed and two heights updated. Useful to fix right-right imbalance.
Double Right Rotation

This is a double right rotation on node $z$:

First, a left rotation at $y$.
Second, a right rotation at $z$. 
Double Left Rotation

Symmetrically, there is a **double left rotation** on node $z$:

First, a right rotation at $y$.
Second, a left rotation at $z$. 
Fixing a slightly-unbalanced AVL tree

\[
\text{restructure}(x, y, z)
\]

node \(x\) has parent \(y\) and grandparent \(z\)

1. case

   \(\bullet z\):
   
   // Right rotation

   return \(\text{rotate-right}(z)\)

   \(\bullet y\):

   \(\bullet x\):

   // Double-right rotation

   \(\text{z} . \text{left} \leftarrow \text{rotate-left}(y)\)

   return \(\text{rotate-right}(z)\)

   \(\bullet y\):

   \(\bullet x\):

   // Double-left rotation

   \(\text{z} . \text{right} \leftarrow \text{rotate-right}(y)\)

   return \(\text{rotate-left}(z)\)

   \(\bullet y\):

   \(\bullet x\):

   // Left rotation

   return \(\text{rotate-left}(z)\)

\[
\text{Rule: The middle key of } x, y, z \text{ becomes the new root.}
\]
AVL Insertion Example revisited

Example:
AVL Insertion: Second example

Example: $AVL::insert(45)$
AVL Deletion

Remove the key $k$ with $\text{BST::delete}$. Find node where structural change happened. (This is not necessarily near the node that had $k$.) Go back up to root, update heights, and rotate if needed.

\[
\text{AVL::delete}(k)
\]

```
1. $z \leftarrow \text{BST::delete}(k)$
2. // Assume $z$ is the parent of the BST node that was removed
3. while ($z$ is not NIL)
4.     if ($|z.\text{left.height} - z.\text{right.height}| > 1$) then
5.         Let $y$ be taller child of $z$
6.         Let $x$ be taller child of $y$ (break ties to prefer single rotation)
7.         $z \leftarrow \text{restructure}(x, y, z)$
8.     // Always continue up the path and fix if needed.
9.     $\text{setHeightFromSubtrees}(z)$
10.    $z \leftarrow z.\text{parent}$
```
AVL Deletion Example

Example:
AVL Tree Operations Runtime

**search:** Just like in BSTs, costs $\Theta(\text{height})$

**insert:** $\text{BST}::\text{insert}$, then check & update along path to new leaf
- total cost $\Theta(\text{height})$
- $\text{restructure}$ restores the height of the subtree to what it was,
- so $\text{restructure}$ will be called *at most once*.

**delete:** $\text{BST}::\text{delete}$, then check & update along path to deleted node
- total cost $\Theta(\text{height})$
- $\text{restructure}$ may be called $\Theta(\text{height})$ times.

**Worst-case** cost for all operations is $\Theta(\text{height}) = \Theta(\log n)$.

But in practice, the constant is quite large.