Module 4: Dictionaries

Mark Petrick
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2020

References: Goodrich & Tamassia 3.1, 4.1, 4.2
1. 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:**
- \( \text{search}(k) \) (also called \( \text{findElement}(k) \))
- \( \text{insert}(k, v) \) (also called \( \text{insertItem}(k, v) \))
- \( \text{delete}(k) \) (also called \( \text{removeElement}(k) \))
- optional: \( \text{closestKeyBefore} \), \( \text{join} \), \( \text{isEmpty}, \text{size} \), 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 $T . \text{left}$ is less than the root key.
             Every key $k$ in $T . \text{right}$ 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

$$\text{key} = 15, \langle\text{other info}\rangle$$
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**

\texttt{BST::search}, \texttt{BST::insert}, \texttt{BST::delete} all have cost $\Theta(h)$, where $h =$ height of the tree $=$ 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

1. 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’skiǐ 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

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

- $\text{balance}(v) = -1$ 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.

```
  22
 /  \
10 +1 31 +1
 /  \
  4 +1 14 +1
 /  \
 6 0 13 0
 /  \
16 0 18 -1
 /  \
28 0 37 +1
 /  \
46 0
```
Height of an AVL tree

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

\[ \Rightarrow \text{search, insert, delete all cost } \Theta(\log n) \text{ in the worst case!} \]

**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
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::insert \( (k, v) \)

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

\text{setHeightFromSubtrees}(u)

1. \( u.height \leftarrow 1 + \max\{u.left.height, u.right.height\} \)
AVL Insertion Example

Example:

```
22
/  
10   31
   /  
  4   28
 /  
6   13
```

```
18
/  
16
```

```
46
```

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. $y \leftarrow z\.\text{left}$, $z\.\text{left} \leftarrow y\.\text{right}$, $y\.\text{right} \leftarrow z$
2. $\text{setHeightFromSubtrees}(z)$, $\text{setHeightFromSubtrees}(y)$
3. return $y$ // 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-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 $z$. 

![Diagram](image-url)
Fixing a slightly-unbalanced AVL tree

restructure(x, y, z)
node x has parent y and grandparent z
1. case
   \[ z : \text{ // Right rotation} \]
   \[ y \quad x \]
   return rotate-right(z)
   \[ z : \text{ // Double-right rotation} \]
   \[ y \quad x \]
   z.left ← rotate-left(y)
   return rotate-right(z)
   \[ z : \text{ // Double-left rotation} \]
   \[ y \quad x \]
   z.right ← rotate-right(y)
   return rotate-left(z)
   \[ z : \text{ // Left rotation} \]
   \[ y \quad x \]
   return rotate-left(z)

**Rule:** The middle key of x, y, z becomes the new root.
AVL Insertion Example revisited

Example:
AVL Insertion: Second example

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

Remove the key $k$ with $BST::delete$.

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.

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

Example:

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>22</td>
<td>31</td>
<td>37</td>
<td>46</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>18</td>
<td>28</td>
<td>37</td>
<td>46</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
AVL Tree Operations Runtime

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

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

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

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

But in practice, the constant is quite large.