CS 240 – Data Structures and Data Management

Module 4: Dictionaries

T. Biedl    E. Schost    O. Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2021
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.

Examples: symbol table, license plate database
Elementary Implementations

Common assumptions:

- Dictionary has \( n \) KVPs
- Each KVP uses constant space (if not, the “value” could be a pointer)
- 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** \( \Theta(n) \)

\[
[5, 1, 3, 2, 7]
\]
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

(key = 15, <other info>)
BST as realization of ADT Dictionary

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

Example: \texttt{BST::search}(24)
BST as realization of ADT Dictionary

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

Example: `BST::search(24)`
BST as realization of ADT Dictionary

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

Example: **BST::search** (24)
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.

Example: \textit{BST::search}(24)
**BST as realization of ADT Dictionary**

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

**BST::insert**\((k, v)\) Search for \(k\), then insert \((k, v)\) as new node

Example: **BST::insert**\((24, v)\)
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.
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.
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
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
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.
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
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**

*BST::search, BST::insert, 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:**
  
  - $n = 7$
  
  - $\{1, 2, 3, \ldots, 7\}$
**Height of a BST**

*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:**

$$\Theta(\log(n))$$
Height of a BST

`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:
**Height of a BST**

*BST::search, BST::insert, BST::delete* all have cost $\Theta(h)$, where $h =$ 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

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:

\[ \text{The heights of the left and right subtree differ by at most 1.} \]

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

Rephrase: If node \( v \) has left subtree \( L \) and right subtree \( R \), then

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

\[
\text{balance}(v) = -1 \text{ means } v \text{ is left-heavy}
\]

\[
\text{balance}(v) = +1 \text{ means } v \text{ is right-heavy}
\]

\[
\text{balance (root)} = -1 - 1 = -2
\]
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.
Height of an AVL tree

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

\( \Rightarrow \) *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?

**Claim:** the height of any binary search tree with \( n \) keys

is \( \Omega(\log n) \).
Proof: Let $h$ be the height of such a tree.

The number $n$ of keys in the tree is less than or equal to the number of keys in the tree of height $h$:

$$n \leq 2^{h+1} - 1 \quad (h=3 \Rightarrow 2^{h+1} - 1 = 15)$$

$$\Rightarrow n \leq 2^{h+1} - 1 \Rightarrow n + 1 \leq 2^{h+1}$$

$$\Rightarrow \log (n+1) - 1 \leq h$$
Proof of \( \theta \)

Fix \( h \), let \( N(h) \) be the minimum number of nodes in an AVL tree of height \( h \).

\[ N(-1) = 0, \quad N(0) = 1, \quad N(1) = 2 \]

\[ N(2) = 4 \]

\[ N(3) = 4 + 2 + 1 = 7 \]
\[ N(h) = N(h-1) + N(h-2) + 1 \]

\[ \Rightarrow N(h) \in \Theta(\psi^h), \quad \psi = \frac{1 + \sqrt{5}}{2} \]

**Claim** \[ N(h) \geq \sqrt{2}^h - 1, \quad h \geq -1. \]

- \[ h = -1 \quad N(-1) = 0 \quad \sqrt{2}^{-1} = 0 \quad \checkmark \]
- \[ h = 0 \quad N(0) = 1 \quad \sqrt{2}^0 = 1 \quad \checkmark \]

Assume true for \(-1, 0, \ldots, h-2, h-1\). Prove for \(h\).

\[ N(h) = N(h-1) + N(h-2) + 1 \geq 2N(h-2) + 1 \geq 2(\sqrt{2}^{h-2} - 1) + 1 \]
\[ = \sqrt{2}^h - 2 + 1 \]
\[ = \sqrt{2}^h - 1 \]
Take an AVL tree with \( n \) nodes and height \( h \).

\[
n \geq N(k) \geq \sqrt{2}^h - 1
\]

\[
n + 1 \geq \sqrt{2}^h
\]

\[
\log_{\sqrt{2}} (n + 1) \geq h \Rightarrow h \in O(\log n).
\]
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 insertion

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

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

Example: $AVL::insert(8)$
AVL Insertion Example

Example: $AVL::insert(8)$
AVL Insertion Example

Example: \textit{AVL::insert}(8)
AVL Insertion Example

Example: $AVL::insert(8)$
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 $z$:

```
rotate-right(z)
1. $y \leftarrow z.'left$, $z.'left \leftarrow y.'right$, $y.'right \leftarrow z$
2. setHeightFromSubtrees(z), setHeightFromSubtrees(y)
3. return $y$ // returns new root of subtree
```
Why do we call this a rotation?
Why do we call this a rotation?
Why do we call this a rotation?
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 $y$. 
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 $z$:

![Diagrams showing double left rotation](image)

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. \textbf{case}\\
   \begin{itemize}
   \item \( z \) : // Right rotation\\
   \hspace{1cm} \textbf{return} \hspace{0.5cm} \text{rotate-right}(z)
   \item \( y \) : // Double-right rotation\\
   \hspace{1cm} z.left \leftarrow \text{rotate-left}(y)\\
   \hspace{1cm} \textbf{return} \hspace{0.5cm} \text{rotate-right}(z)
   \item \( x \) : // Double-left rotation\\
   \hspace{1cm} z.right \leftarrow \text{rotate-right}(y)\\
   \hspace{1cm} \textbf{return} \hspace{0.5cm} \text{rotate-left}(z)
   \end{itemize}

\textbf{Rule}: The middle key of \( x, y, z \) becomes the new root.
AVL Insertion Example revisited

Example: AVL::insert(8)
AVL Insertion Example revisited

Example: $AVL::insert(8)$
AVL Insertion: Second example

Example: $AVL::insert(45)$
AVL Insertion: Second example

Example: \texttt{AVL::insert(45)}
AVL Insertion: Second example

Example: \textit{AVL::insert}(45)
AVL Insertion: Second example

Example: $AVL::insert(45)$
AVL Insertion: Second example

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

Remove the key $k$ with \texttt{BST::delete}.
Find node where \textit{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{algorithm}
\textbf{AVL::delete($k$)}
\begin{enumerate}
\item $z \leftarrow \texttt{BST::delete($k$)}$
\item // Assume $z$ is the parent of the BST node that was removed
\item \textbf{while} ($z$ is not NIL)
\item \hspace{1em} if ($|z.left.height - z.right.height| > 1$) then \checkmark
\item \hspace{1em} Let $y$ be taller child of $z$
\item \hspace{1em} Let $x$ be taller child of $y$ (break ties to prefer single rotation)
\item \hspace{1em} $z \leftarrow \texttt{restructure($x, y, z$)}$ \checkmark
\item // \textit{Always} continue up the path and fix if needed.
\item $\texttt{setHeightFromSubtrees($z$)}$ \checkmark
\item $z \leftarrow z.parent$
\end{enumerate}
\end{algorithm}
Example: `AVL::delete(22)`
AVL Deletion Example

Example: `AVL::delete(22)`
AVL Deletion Example

Example: `AVL::delete(22)`

```
  28
   4?
  /   \
10    31
   /   /  \
  6    14  37
 /   /     / \
4     13   16  46
 /   /   /  / \ \
4     8   18  16  0
 /   /   /   /   /   / \
0     0    1    0    0  0
```

AVL Deletion Example

Example: \texttt{AVL::delete}(22)

![AVL tree diagram](image)
AVL Deletion Example

Example: \texttt{AVL::delete(22)}
AVL Deletion Example

Example: `AVL::delete(22)`
AVL Deletion Example

Example: `AVL::delete(22)`

```
14
 / \  \
10   28
 / \   / \ \
6   2 37 2  \
 /   17 1 0  \
4   13 16 1  \
0   0 0 0 0
```

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 $\Theta(height)$

`restructure` restores the height of the subtree to what it was,
- so `restructure` will be called *at most once*.

**delete:** `BST::delete`, then check & update along path to deleted node
- total cost $\Theta(height)$
- `restructure` 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.
Claim: Let $z$ be the first non-balanced node we meet after insert.

We call $T$ the tree rooted at $z$.

We call $T'$ the tree after restructure.

Then: (1) All nodes in $T'$ are balanced.

(2) $\text{height}(T') = \text{height}(T \text{ before insert})$
Proof for right rotation

Let $h$ be the height at $z$.
(after insert)

<table>
<thead>
<tr>
<th>Proof for right rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z$</td>
</tr>
<tr>
<td>y h-1 D h-3</td>
</tr>
<tr>
<td>x h-2 C h-2 or h-3</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

after insert
Proof for right rotation

Let h be the height at z. (after insert)

before insert
0) all nodes in $T'$ are balanced

2) $\text{height}(T') = h - 1 = \text{height (before insert)}$

after restructure $(T')$

Fin.