Outline

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

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** $\Theta(n)$
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.

(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

**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

**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) \; \text{Start at root, compare } k \text{ to current node’s key. Stop if found or subtree is empty, else recurse at subtree.}

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

Example: \textit{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 $h = \text{height of the tree} = \text{max. path length from root to leaf}$

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

- **Worst-case:**
Height of a BST

\texttt{BST::search}, \texttt{BST::insert}, \texttt{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:**
**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**: $n - 1 = \Theta(n)$
- **Best-case**: $\Theta(\log n)$.
  - Any binary tree with $n$ nodes has height $\geq \log(n + 1) - 1$
- **Average-case**: 

---

Biedl, Schost, Veksler (SCS, UW)  
CS240 – Module 4  
Winter 2021  
7 / 26
Height of a BST

\textit{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 $h$?

- \textbf{Worst-case:} $n - 1 = \Theta(n)$
- \textbf{Best-case:} $\Theta(\log n)$.
  - Any binary tree with $n$ nodes has height $\geq \log(n + 1) - 1$
- \textbf{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

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

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

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

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

- $\text{balance}(v) = -1$ 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  31
/  /  \
4 14 28 37
/ / / /  \
6 13 18 16 46
```

Biedl, Schost, Veksler (SCS, UW)
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
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

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

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

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

Example: $AVL::\text{insert}(8)$
AVL Insertion Example

Example: \texttt{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:

\[
\begin{align*}
\text{rotate-right}(z) \\
1. & \quad y \leftarrow z.\text{left}, \quad z.\text{left} \leftarrow y.\text{right}, \quad y.\text{right} \leftarrow z \\
2. & \quad \text{setHeightFromSubtrees}(z), \quad \text{setHeightFromSubtrees}(y) \\
3. & \quad \text{return } y \quad // \quad \text{returns new root of subtree}
\end{align*}
\]
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 $z$:

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

\[
\text{restructure}(x, y, z) \\
\text{node } x \text{ has parent } y \text{ and grandparent } z
\]

1. \textbf{case}
   - \textit{z}: \hspace{1em} \text{// Right rotation}
     \hspace{1em} \text{return } \text{rotate-right}(z)
   - \textit{y}: \hspace{1em} \text{// Double-right rotation}
     \hspace{1em} z.\text{left} \leftarrow \text{rotate-left}(y)
     \hspace{1em} \text{return } \text{rotate-right}(z)
   - \textit{y}: \hspace{1em} \text{// Double-left rotation}
     \hspace{1em} z.\text{right} \leftarrow \text{rotate-right}(y)
     \hspace{1em} \text{return } \text{rotate-left}(z)
   - \textit{x}: \hspace{1em} \text{// Left rotation}
     \hspace{1em} \text{return } \text{rotate-left}(z)

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

Example: $AVL::insert(8)$
Example: \texttt{AVL::insert}(8)

![AVL Insertion Diagram]

Biedl, Schost, Veksler (SCS, UW)  
CS240 – Module 4  
Winter 2021  
22 / 26
AVL Insertion: Second example

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

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

Example: \texttt{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 $\text{BST}::\text{delete}$. 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{algorithm}
\begin{algorithmic}[1]
\STATE $z \leftarrow \text{BST}::\text{delete}(k)$
\STATE // Assume $z$ is the parent of the BST node that was removed
\WHILE{$(z \text{ is not NIL})$}
\IF{$(|z.\text{left}.\text{height} - z.\text{right}.\text{height}| > 1)$}
\STATE Let $y$ be taller child of $z$
\STATE Let $x$ be taller child of $y$ (break ties to prefer single rotation)
\STATE $z \leftarrow \text{restructure}(x, y, z)$
\ENDIF
\STATE // Always continue up the path and fix if needed.
\STATE $\text{setHeightFromSubtrees}(z)$
\STATE $z \leftarrow z.\text{parent}$
\ENDWHILE
\end{algorithmic}
\end{algorithm}
AVL Deletion Example

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

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

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

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

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

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

Example: `AVL::delete(22)`
AVL Tree Operations Runtime

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

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

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