Outline

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

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

A dictionary is a collection of items, each of which contains

- a key
- some data,

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

- $search \ \Theta(n)$
- $insert \ \Theta(1)$
- $delete \ \Theta(n)$ (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
   - Deletion in AVL Trees
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 Search and Insert**

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

Example:  **BST-search**($24$)

![Binary Search Tree Diagram](image)
BST Search and Insert

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

Example: **BST-search**(24)
BST Search and Insert

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

Example: **BST-search**(24)
**BST Search and Insert**

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

Example: **BST-search**\( (24) \)

![BST Search Example Diagram](image)
BST Search and Insert

**BST-search**\( (k) \) Start at root, compare \( k \) to current node.
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**\( (24, \ldots) \)
BST Delete

- First search for the node $x$ that contains the key.
- If $x$ is a leaf (both subtrees are empty), delete it.
BST Delete

- First search for the node $x$ that contains the key.
- If $x$ is a leaf (both subtrees are empty), delete it.
BST Delete

- 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
BST Delete

- 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
BST Delete

- 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
BST Delete

- 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
BST Delete

- 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

```
     23
    /  \
   10   25
  /  \   /  \
  8  14  23  29
       /  \
      24  50
```
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 *BST-inserted* one-at-a-time, how big is $h$?

- **Worst-case:**

\[ n - 1 = \Theta(n) \]

\[ \Theta(\log n) \]

Any binary tree with $n$ nodes has height $\geq \log(n+1) - 1$
**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 *BST-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 $h = \text{height of the tree} = \text{max. path length from root to leaf}$

If $n$ items are **BST-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 $=$ max. path length from root to leaf

If $n$ items are **BST-insert**ed 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
     - Deletion in AVL Trees
AVL Trees

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

The heights of the left subtree $L$ and right subtree $R$ differ by at most 1. (The height of an empty tree is defined to be $-1$.)

At each non-empty node, we require $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$:

- $-1$ means the tree is left-heavy
- $0$ means the tree is balanced
- $+1$ means the tree 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:

The heights of the left subtree $L$ and right subtree $R$ differ by at most 1. (The height of an empty tree is defined to be $-1$.)

At each non-empty node, we require $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$:

- $-1$ means the tree is left-heavy
- $0$ means the tree is balanced
- $+1$ means the tree is right-heavy

- Need to store at each node the height of the subtree rooted at it
- Can show: It suffices to store $\text{height}(R) - \text{height}(L)$ at each node.
  - uses fewer bits
  - code gets more complicated, especially for deleting
AVL tree example

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

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

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

$\Rightarrow$ **AVL-search, AVL-insert, AVL-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?

Caution, Goodrich & Tamassia uses a different height-definition, therefore their base cases are different from ours.
Outline

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

To perform \textit{AVL-insert}(T, k, v):

- First, insert \((k, v)\) into \(T\) 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 \textit{unbalanced}. Must re-structure the tree to rebalance.
AVL insertion

AVL-insert\((r, k, v)\)
1. \(z \leftarrow \text{BST-insert}(r, k, v)\)
2. \(z.\text{height} \leftarrow 0\)
3. \(\text{while } (z \text{ is not the root})\)
4. \(z \leftarrow \text{parent of } z\)
5. \(\text{if } (|z.\text{left.height} - z.\text{right.height}| > 1) \text{ then}\)
6. \(\text{Let } y \text{ be taller child of } z \text{ (break ties arbitrarily)}\)
7. \(\text{Let } x \text{ be taller child of } y\)
8. \((\text{break ties to prefer left-left or right-right})\)
9. \(z \leftarrow \text{restructure}(x) \quad // \text{ see later}\)
10. \(\text{break} \quad // \text{ can argue that we are done}\)
11. \(\text{setHeightFromSubtrees}(z)\)

setHeightFromSubtrees\((u)\)
1. \(\text{if } u \text{ is not an empty subtree}\)
2. \(u.\text{height} \leftarrow 1 + \max\{u.\text{left.height}, u.\text{right.height}\}\)
Example: $AVL\text{-}insert(8)$
AVL Insertion Example

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

Example: $AVL-insert(8)$
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
   - Deletion in AVL Trees
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 ← z.left, z.left ← y.right, y.right ← z
2.  setHeightFromSubtrees(z), setHeightFromSubtrees(y)
3.  return y  // returns new root of subtree
```
Right Rotation

This is a right rotation on node \( z \):

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

![Diagram showing left rotation on node z]

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

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

![Diagram showing double right rotation](image)

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.
Useful for left-right imbalance.
Double Left Rotation

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

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

\[ \text{restructure}(x) \]
\[ x: \text{node of BST that has a grandparent} \]
1. Let y and z be the parent and grandparent of x
2. \textbf{case}
   \[ z: \text{// Right rotation} \]
   \[ \begin{array}{l}
   y \\
   x
   \end{array} \]
   return \text{rotate-right}(z)

   \[ z: \text{// Double-right rotation} \]
   \[ \begin{array}{l}
   y \\
   x
   \end{array} \]
   z.left ← \text{rotate-left}(y)
   return \text{rotate-right}(z)

   \[ z: \text{// Double-left rotation} \]
   \[ \begin{array}{l}
   y \\
   x
   \end{array} \]
   z.right ← \text{rotate-right}(y)
   return \text{rotate-left}(z)

   \[ z: \text{// Left rotation} \]
   \[ \begin{array}{l}
   y \\
   x
   \end{array} \]
   return \text{rotate-left}(z)

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

Example: $AVL-insert(8)$
Example: \textit{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
   - Deletion in AVL Trees
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.

\[
\text{AVL-delete}(r, k) \\
1. \quad z \leftarrow \text{BST-delete}(r, k) \\
2. \quad // \text{ Assume } z \text{ is the child of the BST node that was removed} \\
3. \quad \text{setHeightFromSubtrees}(z) \\
4. \quad \textbf{while } (z \text{ is not the root}) \\
5. \quad \quad z \leftarrow \text{parent of } z \\
6. \quad \quad \textbf{if } (|z.\text{left.height} - z.\text{right.height}| > 1) \textbf{ then} \\
7. \quad \quad \quad \text{Let } y \text{ be taller child of } z \text{ (break ties arbitrarily)} \\
8. \quad \quad \quad \text{Let } x \text{ be taller child of } y \text{ (break ties as for Insert)} \\
9. \quad \quad \quad \quad z \leftarrow \text{restructure}(x) \\
10. \quad \quad \quad // \textbf{ Always } \text{ continue up the path and fix if needed.} \\
11. \quad \quad \quad \text{setHeightFromSubtrees}(z)
\]
AVL Deletion Example

Example: *AVL-delete*(22)
AVL Deletion Example

Example: $AVL\text{-}delete(22)$
Example: AVL-delete(22)
Example: *AVL-delete*(22)
AVL Deletion Example

Example: \textit{AVL-delete}(22)

\begin{itemize}
  \item AVL tree after deletion of vertex 22:
  \item The tree is balanced after the deletion.
\end{itemize}
AVL Deletion Example

Example: $AVL\text{-delete}(22)$
AVL Deletion Example

Example: \textit{AVL-delete}(22)
AVL Tree Operations Runtime

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

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

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

Total cost for all operations is $\Theta(\text{height}) = \Theta(\log n)$. 