COMP2521 24T1
AVL Trees
Kevin Luxa
cs2521@cse.unsw.edu.au
Invented by Georgy Adelson-Velsky and Evgenii Landis in 1962
Approach:

- Keep tree height-balanced
- Repair balance as soon as imbalance occurs
  - During insertion or deletion
- Repairs are done locally, not by restructuring entire tree
Height of an AVL tree

Since AVL trees are always height-balanced, the height of an AVL tree is guaranteed to be at most
\[ \log_\phi (n + 2) - 0.3277 \text{ (where } \phi \text{ is the golden ratio)} \]
\[ \approx 1.4404 \log_2(n + 2) - 0.3277 = O(\log n) \]
Note:

AVL trees are not necessarily size-balanced. For example, the following is a perfectly valid AVL tree:
AVL Tree Insertion

Method:

• Insert item recursively
• Check balance at each node along the insertion path *in reverse*
  • i.e., from bottom to top
• Fix imbalances as they are found
Example: Insert 5 into this tree

```
Example: Insert 5 into this tree

Balance must be checked at 4, then at 2, then at 6
```
Example: Insert 5 into this tree

Balance must be checked at 4, then at 2, then at 6
AVL Tree Insertion

How to check balance along insertion path *in reverse*?

- Perform balance checking as a *postorder* operation in the insertion function
  - In other words - add balance checking code *below* recursive calls
Outline of insertion process:

1. if the tree is empty:
   - return new node
2. insert recursively
3. check (and fix) balance
4. return root of updated tree
avlInsert(t, v):
    Input: AVL tree t, item v
    Output: t with v inserted

    if t is empty:
        return new node containing v
    else if v < t->item:
        t->left = avlInsert(t->left, v)
    else if v > t->item:
        t->right = avlInsert(t->right, v)
    else:
        return t

    return avlRebalance(t)
avlRebalance(t):
   \textbf{Input:} possibly unbalanced tree \( t \)
   \textbf{Output:} balanced \( t \)

   bal = balance(t)
   \textbf{if} bal > 1:
       \textbf{if} balance(t->left) < 0:
           t->left = rotateLeft(t->left)
           t = rotateRight(t)
   \textbf{else if} bal < -1:
       \textbf{if} balance(t->right) > 0:
           t->right = rotateRight(t->right)
           t = rotateLeft(t)

   \textbf{return} t

balance(t):
   \textbf{Input:} tree \( t \)
   \textbf{Output:} balance factor of \( t \)

   \textbf{return} height(t->left) - height(t->right)
There are 4 rebalancing cases:

- Left Left
- Left Right
- Right Left
- Right Right
AVL Tree Insertion
Rebalancing

Left Left

\[
\text{bal} = \text{balance}(t) \\
\text{if} \ \text{bal} > 1: \ (\text{true}) \\
\quad \text{if} \ \text{balance}(t\rightarrow\text{left}) < 0: \ (\text{false}) \\
\quad \quad t\rightarrow\text{left} = \text{rotateLeft}(t\rightarrow\text{left}) \\
\quad \quad t = \text{rotateRight}(t) \\
\text{else if} \ \text{bal} < -1: \\
\quad \text{if} \ \text{balance}(t\rightarrow\text{right}) > 0: \\
\quad \quad t\rightarrow\text{right} = \text{rotateRight}(t\rightarrow\text{right}) \\
\quad \quad t = \text{rotateLeft}(t)
\]
AVL Tree Insertion

Rebalancing

bal = balance(t)
if bal > 1: (true)
    if balance(t->left) < 0: (true)
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if bal < -1:
    if balance(t->right) > 0:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)
AVL Tree Insertion

Rebalancing

\[
\text{bal} = \text{balance}(t)
\]

\[
\text{if bal} > 1: \quad \text{(false)}
\]

\[
\text{if balance}(t->left) < 0:
\quad t->left = \text{rotateLeft}(t->left)
\quad t = \text{rotateRight}(t)
\]

\[
\text{else if bal} < -1: \quad \text{(true)}
\]

\[
\text{if balance}(t->right) > 0: \quad \text{(true)}
\quad t->right = \text{rotateRight}(t->right)
\quad t = \text{rotateLeft}(t)
\]
AVL Tree Insertion

Rebalancing

**Right Right**

```plaintext
bal = balance(t)
if bal > 1: (false)
    if balance(t->left) < 0:
        t->left = rotateLeft(t->left)
        t = rotateRight(t)
else if bal < -1: (true)
    if balance(t->right) > 0: (false)
        t->right = rotateRight(t->right)
        t = rotateLeft(t)
```

![Diagram of Right Right rebalancing](image)
Insert 7 into this tree:

```
  6
 / \
2  9
|  |
1 8
 | |
3 5
```

Insert 7 into this tree:
AVL Tree Insertion
Rebalancing Example 1 - Left Left

Check for balance at 8, then at 9, then at 6.

9 is unbalanced.
AVL Tree Insertion

Rebalancing Example 1 - Left Left
Insert 4 into this tree:
AVL Tree Insertion
Rebalancing Example 2 - Left Right

Check for balance at 3, then at 5, then at 2, then at 6.

5 is unbalanced.
AVL Tree Insertion

Rebalancing Example 2 - Left Right

Diagram showing the insertion of a node into an AVL tree and the resulting rebalancing operations to maintain the AVL property.
AVL tree insertion requires balance checking at each node on the insertion path...

...which requires the height of many subtrees to be computed

In an ordinary binary search tree, computing the height is $O(n)!$ (need to traverse whole (sub)tree)
Solution:

For each node, store the height of its subtree in the node itself:

```c
struct node {
    int item;
    struct node *left;
    struct node *right;
    int height;
};
```
Height of each node's subtree is stored in the node itself
When does height data need to be maintained?

- Whenever a node is inserted
  - Heights of all ancestors may be affected
- Whenever a rotation is performed
  - Heights of original root and new root may be affected
Whenever a node is inserted...
...heights of all ancestors may be affected

Example: Insert 4 into this tree
Recompute height of each ancestor (from bottom to top) using the heights stored in its children.
The heights of 5’s children are 0 and -1 (empty tree).

Thus, the height of 5 is \( \max(0, -1) + 1 = 1 \).
AVL Tree Insertion
Maintaining Height Data - Insertions

The heights of 2's children are 0 and 1.

Thus, the height of 2 is \( \max(0, 1) + 1 = 2 \).
The heights of 6's children are 2 and 1.

Thus, the height of 6 is $\max(2, 1) + 1 = 3$. 
AVL Tree Insertion
Maintaining Height Data - Insertions

Done.

Note that recomputing the height of each node was done in $O(1)$ time.
Whenever a rotation is performed...
...heights of original root and new root may be affected
Example: Perform a right rotation at 7
Recompute height of original root then recompute height of new root using the heights stored in their children.
The height of 7’s children are 0 and 0.

Thus, the height of 7 is \( \max(0, 0) + 1 = 1 \).
The height of 4’s children are 1 and 1.

Thus, the height of 4 is $\max(1, 1) + 1 = 2$. 
AVL Tree Insertion

Maintaining Height Data - Rotations

Every rotation, two height updates are performed, each in $O(1)$ time.

Done.
Analysis:

- Height of an AVL tree is $O(\log n)$
- In the worst case, length of insertion path is $O(\log n)$
- Have to maintain height data and check/fix balance at each node on insertion path
  - This is $O(1)$ per node
- Therefore, worst-case time complexity of AVL tree insertion is $O(\log n)$
AVL Tree Search

Exactly the same as for regular BSTs.

Worst-case time complexity is $O(\log n)$, since AVL trees are height-balanced.
AVL Tree Deletion

**Method:**

- Delete item recursively
- Check balance at each node along the deletion path* in reverse
- Fix imbalances as they are found
Example: Delete 10 from this tree
Example: Delete 10 from this tree

Balance must be checked at 6, then at 13
Important:
If the item being deleted has two child nodes, the deletion path includes the path to its successor (the smallest value in its right subtree).
Example: Delete 13 from this tree

13 will be replaced by 15 (its in-order successor)
Example: Delete 13 from this tree

Balance must be checked at 17, then at 23, then at 15
avlDelete(t, v):
    Input: AVL tree t, item v
    Output: t with v deleted

    if t is empty:
        return empty tree
    else if v < t->item:
        t->left = avlDelete(t->left, v)
    else if v > t->item:
        t->right = avlDelete(t->right, v)
    else:
        if t->left is empty:
            temp = t->right
            free(t)
            return temp
        else if t->right is empty:
            temp = t->left
            free(t)
            return temp
        else:
            successor = minimum value in t->right
            t->item = successor
            t->right = avlDelete(t->right, successor)

    return avlRebalance(t)
Note: This is the same as in AVL tree insertion

```plaintext
avlRebalance(t):
  Input: possibly unbalanced tree t
  Output: balanced t

  bal = balance(t)
  if bal > 1:
    if balance(t->left) < 0:
      t->left = rotateLeft(t->left)
    t = rotateRight(t)
  else if bal < -1:
    if balance(t->right) > 0:
      t->right = rotateRight(t->right)
    t = rotateLeft(t)

  return t

balance(t):
  Input: tree t
  Output: balance factor of t

  return height(t->left) - height(t->right)
```
AVL tree deletion has the same rebalancing cases as AVL tree insertion.
Delete 2 from this tree:

```
  9
 / \
5   16
 /   / \
7   12 17
 /   /   /\
3   11 13 20
```

At node 2, the left child is 1.

```
 Delete 2:
  9
 /   \       (node 2 deleted)
5     16
 /     / \     (node 1 becomes the new root)
7     12 17
 /   /   /   /\
3   11 13 20 15
```

After deletion, node 5 is made the new root, and the tree is rebalanced.
Check for balance at 5 and 9
AVL Tree Deletion

Rebalancing Example 1 - Right Left

9 is unbalanced
AVL Tree Deletion

Rebalancing Example 1 - Right Left

Balanced
Delete 8 from this tree:
AVL Tree Deletion
Rebalancing Example 2 - Right Right

Check for balance at 13 and 9
AVL Trees

Insertion

Search

Deletion

Pseudocode

Rebalancing

Examples

Height data

Analysis

Summary

AVL Tree Deletion

Rebalancing Example 2 - Right Right

13 is unbalanced
AVL Tree Deletion

Rebalancing Example 2 - Right Right
Height data also needs to be maintained...

- Whenever a node is deleted
  - Heights of all nodes on deletion path may be affected
Example: Delete 6 from this tree
Recompute height of each node on the deletion path using the heights stored in its children.
The heights of 16’s children are 0 and 0.

Thus, the height of 16 is \( \max(0, 0) + 1 = 1 \).
The heights of 11’s children are 1 and 1.

Thus, the height of 11 is $\max(1, 1) + 1 = 2$. 
The heights of 19’s children are 2 and 2.

Thus, the height of 19 is \( \max(2, 2) + 1 = 3 \).
Note that recomputing the height of each node was done in $O(1)$ time.
Analysis:

• Height of an AVL tree is \( O(\log n) \)

• In the worst case, length of deletion path is \( O(\log n) \)

• Have to maintain height data and check/fix balance at each node on deletion path
  • This is \( O(1) \) per node

• Therefore, worst-case time complexity of AVL tree deletion is \( O(\log n) \)
• AVL trees are always height-balanced
  • This means the height of an AVL tree is $O(\log n)$
• Rotations are used to fix imbalances during insertion and deletion
• Balance is checked efficiently by storing height data in each node, which needs to be maintained
• Worst-case time complexity of $O(\log n)$ for insertion, search and deletion
https://forms.office.com/r/5c0fb4tvMb