COMP2521 23T3
Balanced Binary Search Trees

Kevin Luxa
cs2521@cse.unsw.edu.au

balanced trees
avl trees
The structure, height, and hence performance of a binary search tree depends on the order of insertion.
**Best case**

Items are inserted evenly on the left and right throughout the tree. Height of tree will be $O(\log n)$.
Worst case

Items are inserted in ascending or descending order such that tree consists of a single branch. Height of tree will be $O(n)$. 

![Diagram of a tree with height $O(n)$]
A binary tree of $n$ nodes is said to be balanced if it has (close to) minimal height ($O(\log n)$), and degenerate if it has (close to) maximal height ($O(n)$).

We want to build balanced trees.
Types of Balance

**SIZE BALANCED**

A *weight-balanced* or *size-balanced* tree has, for every node,

\[|\text{SIZE}(l) - \text{SIZE}(r)| \leq 1\]

**HEIGHT BALANCED**

A *height-balanced* tree has, for every node,

\[|\text{HEIGHT}(l) - \text{HEIGHT}(r)| \leq 1\]
Balance Examples
Balancing Operations
Balancing Methods
AVL Trees

Balanced or Not?
(I)
Balanced or Not?

(I)

Balance

Examples

Balancing

Operations

Balancing

Methods

AVL Trees

\[ \text{SIZE} \left( \tau_4 \right) = 5 \]
\[ \text{SIZE} \left( \tau_2 \right) = 2 \]
\[ \text{SIZE} \left( \tau_5 \right) = 2 \]
\[ \text{SIZE} \left( \tau_3 \right) = 1 \]
\[ \text{SIZE} \left( \tau_6 \right) = 1 \]
\[ \text{SIZE} \left( \tau_\emptyset \right) = 0 \]

\[ \text{HEIGHT} \left( \tau_4 \right) = 2 \]
\[ \text{HEIGHT} \left( \tau_2 \right) = 1 \]
\[ \text{HEIGHT} \left( \tau_5 \right) = 1 \]
\[ \text{HEIGHT} \left( \tau_3 \right) = 0 \]
\[ \text{HEIGHT} \left( \tau_6 \right) = 0 \]
\[ \text{HEIGHT} \left( \tau_\emptyset \right) = -1 \]

SIZE BALANCED

HEIGHT BALANCED
Balanced or Not?

\[
\begin{align*}
\text{size}(\tau_4) &= 5 \\
\text{size}(\tau_2) &= 3 \\
\text{size}(\tau_5) &= 1 \\
\text{size}(\tau_1) &= 1 \\
\text{size}(\tau_3) &= 1 \\
|\text{size}(\tau_2) - \text{size}(\tau_5)| &= 2
\end{align*}
\]

not size balanced.

\[
\begin{align*}
\text{height}(\tau_4) &= 2 \\
\text{height}(\tau_2) &= 1 \\
\text{height}(\tau_5) &= 0 \\
\text{height}(\tau_1) &= 0 \\
\text{height}(\tau_3) &= 0 \\
|\text{height}(\tau_2) - \text{height}(\tau_5)| &= 1
\end{align*}
\]

height balanced.
Balanced or Not?

\[
\begin{align*}
\text{SIZE}(\tau_4) &= 5 \\
\text{SIZE}(\tau_2) &= 3 \\
\text{SIZE}(\tau_5) &= 1 \\
\text{SIZE}(\tau_1) &= 1 \\
\text{SIZE}(\tau_3) &= 1
\end{align*}
\]
\[
\begin{align*}
|\text{SIZE}(\tau_2) - \text{SIZE}(\tau_5)| &= 2 \\
\text{NOT SIZE BALANCED}
\end{align*}
\]

\[
\begin{align*}
\text{HEIGHT}(\tau_4) &= 2 \\
\text{HEIGHT}(\tau_2) &= 1 \\
\text{HEIGHT}(\tau_5) &= 0 \\
\text{HEIGHT}(\tau_1) &= 0 \\
\text{HEIGHT}(\tau_3) &= 0
\end{align*}
\]
\[
\begin{align*}
|\text{HEIGHT}(\tau_2) - \text{HEIGHT}(\tau_5)| &= 1 \\
\text{HEIGHT BALANCED}
\end{align*}
\]
Let's look at $\tau_3$.

$\text{size}(\tau_2) = 2$

$\text{size}(\tau_\emptyset) = 0$

$|2 - 0| = 2 > 1$

not size balanced

Let's look at $\tau_5$.

$\text{height}(\tau_\emptyset) = -1$

$\text{height}(\tau_6) = 1$

$|-1 - 1| = 2 > 1$

not height balanced
Let’s look at $\tau_3$.

- $\text{SIZE}(\tau_2) = 2$
- $\text{SIZE}(\tau_\emptyset) = 0$
- $|2 - 0 = 2| > 1$

**NOT SIZE BALANCED**

Let’s look at $\tau_5$.

- $\text{HEIGHT}(\tau_\emptyset) = -1$
- $\text{HEIGHT}(\tau_6) = 1$
- $|-1 - 1| = 2 > 1$

**NOT HEIGHT BALANCED**
Challenge:

Prove that every size-balanced tree is height-balanced.
Balancing Operations

Rotation
- Left rotation
  - Move right child to root, rearrange links to retain order
- Right rotation
  - Move left child to root, rearrange links to retain order

Partition
- Rearrange tree around a specified node by rotating it up to the root
LEFT ROTATION and RIGHT ROTATION: a pair of operations that change the balance of a tree
Rotations maintain the order of a search tree:

Right rotation

Left rotation

(all values in \(t_1\)) < \(n_2\) < (all values in \(t_2\)) < \(n_1\) < (all values in \(t_3\))
Method for right rotation:

- before the rotation: $n_1$ is original root, $n_2$ is left child of root
- $n_1$’s left subtree is now what was $n_2$’s right subtree
- $n_2$’s right child is now $n_1$
- $n_2$ is now the new root
- everything else is unchanged
Rotate right at 5
Rotate right at 5

Before rotation:
```
     5
   /   
  3     6
 /     /   
2     4     
```

After rotation:
```
     3
   /   
  2     6
 /     /   
4     5
```
Rotate left at 3

```
3
  / 
2   5
  |   |
4   6
```
Rotate left at 3

Before rotation:

```
      3
     / \  
  2    5
     /   /
  4    6
```

After rotation:

```
      5
     / \  
  3    6
     /   /
  2    4
```
Rotate right at 23
Rotate right at 23
struct node *rotateRight(struct node *root) {
    if (root == NULL || root->left == NULL) return root;
    struct node *newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    return newRoot;
}

struct node *rotateLeft(struct node *root) {
    if (root == NULL || root->right == NULL) return root;
    struct node *newRoot = root->right;
    root->right = newRoot->left;
    newRoot->left = root;
    return newRoot;
}
Analysis:

- Rotation is cheap - $O(1)$
- Rotation requires simple, localised pointer re-arrangements

Sometimes, rotation is applied along one branch, from leaf to root
- Cost of this is $O(h)$ where $h$ is the height of the tree
partition(tree, i)

Rearrange the tree so that the element with index \( i \) becomes the root
Method:

- Find element with index $i$
- Perform rotations to lift it to the root
  - If it is the left child of its parent, perform right rotation at its parent
  - If it is the right child of its parent, perform left rotation at its parent
  - Repeat until it is at the root of the tree
Partition this tree around index 3:

```
5  [0]
10 [1]
14 [2]
30 [4]
29 [3]
32 [5]
30
```

Partition this tree around index 3:
After right rotation at 30:

```
   10
   / \
  5   14
     /  \
    29   30
     /    \
    32    3
```
After left rotation at 14:
After left rotation at 10:

```
  29
 /   \
10    30
|     |
5 14  32
```
partition(t, i):

**Inputs:** tree t, index i

**Output:** tree with i-th item moved to root

\[ m = \text{size}(t->left) \]

**if** \( i < m \):

\[ t->left = \text{partition}(t->left, i) \]

\[ t = \text{rotateRight}(t) \]

**else if** \( i > m \):

\[ t->right = \text{partition}(t->right, i - m - 1) \]

\[ t = \text{rotateLeft}(t) \]

**return** \( t \)
Analysis:

• size() operation is expensive
  • needs to traverse whole subtree

• can cause partition to be $O(n^2)$ in the worst case

• to improve efficiency, can change node structure so that each node stores the size of its subtree in the node itself
  • however, this will require extra work in other functions to maintain

```c
struct node {
    int item;
    int size;
    struct node *left;
    struct node *right;
};
```
• Global Rebalancing
• Root Insertion
• Randomised Insertion
Global Rebalancing

Idea:
 Completely rebalance whole tree so it is size-balanced

Method:
Lift the median node to the root by partitioning on \( \text{SIZE}(t)/2 \), then rebalance both subtrees (recursively)
Global Rebalancing

First, partition on index $n/2$...

...then rebalance both subtrees
Global Rebalancing
Pseudocode

rebalance(t):

Inputs: tree t
Output: rebalanced t

if size(t) < 3:
    return t

t = partition(t, size(t) / 2)
t->left = rebalance(t->left)
t->right = rebalance(t->right)
return t
Worst-case time complexity: $O(n \log n)$

- Assume nodes store the size of their subtrees
- First step: partition entire tree on index $n/2$
  - This takes at most $n$ recursive calls, $n$ rotations $\Rightarrow n$ steps
  - Result is two subtrees of size $\approx n/2$
- Then partition both subtrees
  - Partitioning these subtrees takes $n/2$ steps each $\Rightarrow n$ steps in total
  - Result is four subtrees of size $\approx n/4$
- ...and so on...
- About $\log_2 n$ levels of partitioning in total, each requiring $n$ steps
  $\Rightarrow O(n \log n)$
What if we insert more items?

- **Options:**
  - Rebalance on every insertion
    - Not feasible
  - Rebalance every $k$ insertions; what $k$ is good?
  - Rebalance when imbalance exceeds threshold.

- It’s a tradeoff...
  - We either have more costly insertions
  - Or we have degraded performance for periods of time
bstInsert(t, v):

**Inputs:** tree t, value v

**Output:** t with v inserted

\[ t = \text{insertAtLeaf}(t, v) \]

\[ \text{if size}(t) \mod k = 0: \]
\[ \quad t = \text{rebalance}(t) \]

\[ \text{return } t \]
Periodic Rebalancing
Remarks

- Good if tree is not modified very often
- Otherwise...
  - Insertion will be slow occasionally due to rebalancing
  - Performance will gradually degrade until next rebalance
GLOBAL REBALANCING
walks every node, balances its subtree;
⇒ perfectly balanced tree — at cost.

LOCAL REBALANCING
do small, incremental operations
to improve the overall balance of the tree
... at the cost of imperfect balance
Idea:

Rotations change the structure of a tree

If we perform some rotations every time we insert, that may restructure the tree randomly enough such that it is more balanced

One systematic way to perform these rotations: Insert new values at the root
**Method:**

Insert new value normally (at the leaf) ...
... and then rotate the new node up to the root.
Insert 24 at the root of this tree:
Insert 24 at the root of this tree:
Root Insertion

Example

Rotate right at 29

10

5 14 30

29 32

5 14 30

24 29 32

24

4

3

2

1

4

3

2
Root Insertion
Example

Rotate right at 30

10

5

14

30

24

32

29

10

5

14

30

24

32

29

30

29

32
Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Root Insertion Example

Rotate left at 14

```
<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>
```
Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Root Insertion Example

Rotate left at 10

Before rotation:

10
  5
   14
    30
     29
     32

After rotation:

10
  24
     5
    14
   30
     29
     32

24
  29
   32

30
  29
   32
insertAtRoot(t, v):

**Inputs:** tree t, value v

**Output:** t with v inserted at the root

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

return t
Analysis:

- Same complexity as normal insertion: $O(h)$
  - In reality, cost is doubled, as you need to traverse down and rotate up
- Tree is more likely to be balanced, but no guarantee
- Insert at root ensures recently inserted items are close to the root
  - Useful for applications where recently added items are more likely to be searched
- Major problem: ascending-ordered and descending-ordered data is still a worst case for root insertion
BSTs don’t have control over insertion order. worst cases — (partially) ordered data — are common.

**Idea:**
Introduce some randomness into insertion algorithm:
Randomly choose whether to insert normally or insert at root
insertRandom($t, v$):

**Inputs:** tree $t$, value $v$

**Output:** $t$ with $v$ inserted

if $t$ is empty:
    return new node containing $v$

// p/q chance of inserting at root
if random() mod q < p:
    return insertAtRoot($t, v$)
else:
    return insertAtLeaf($t, v$)

Note: random() is a pseudo-random number generator
30% chance of root insertion $\Rightarrow$ choose $p = 3$, $q = 10$
Randomised insertion creates similar results to inserting items in random order.

Tree is more likely to be balanced (but no guarantee)
AVL Trees
Motivation:

- Previous balancing methods are either inefficient, or don’t guarantee a balanced tree ($O(\log n)$ height)
Invented by Georgy Adelson-Velsky and Evgenii Landis (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
Method:

- Insert item recursively
- Check balance at each node along the insertion path in reverse
  - i.e., from bottom to top
- As soon as an imbalance is found, fix it
Example: Insert 5 into this tree

```
6
/  \\
2    9
/    /
1    4
    /
    3
```

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?

- Simple - perform balance checking as a postorder operation in the insertion function
  - In other words - insert balance checking code below recursive calls to insert

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):

Inputs: 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

... continued on next slide ...
... continued from previous slide ...

leftHeight = height(t->left)
rightHeight = height(t->right)

if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
        t = rotateRight(t)
    else if (rightHeight - leftHeight) > 1:
        if v < t->right->item:
            t->right = rotateRight(t->right)
            t = rotateLeft(t)

return t
There are 4 rebalancing cases:

- Left Left
- Left Right
- Right Left
- Right Right
**Left Left**

```python
if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if (rightHeight - leftHeight) > 1:
    if v < t->right->item:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)
```

**Left Right**

```python
if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if (rightHeight - leftHeight) > 1:
    if v < t->right->item:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)
```
**AVL Tree Insertion**

### Right Left

\[
\text{if } (\text{leftHeight} - \text{rightHeight}) > 1:\n\quad \text{if } v > t->\text{left}->\text{item}: \quad \text{t->left = rotateLeft}(t->\text{left}) \quad t = \text{rotateRight}(t)\\
\quad \text{else if } (\text{rightHeight} - \text{leftHeight}) > 1:\n\quad \quad \text{if } v < t->\text{right}->\text{item}: \quad t->\text{right} = \text{rotateRight}(t->\text{right}) \quad t = \text{rotateLeft}(t)
\]

### Right Right

\[
\text{if } (\text{leftHeight} - \text{rightHeight}) > 1:\n\quad \text{if } v > t->\text{left}->\text{item}: \quad t->\text{left} = \text{rotateLeft}(t->\text{left}) \quad t = \text{rotateRight}(t)\\
\quad \text{else if } (\text{rightHeight} - \text{leftHeight}) > 1:\n\quad \quad \text{if } v < t->\text{right}->\text{item}: \quad t->\text{right} = \text{rotateRight}(t->\text{right}) \quad t = \text{rotateLeft}(t)
\]
Insert 7 into this tree:

```
  6
 / \
2   9
 / \ / \
1   5 8  1
    / \   
   0   1  0
```

AVL Tree Insertion
Rebalancing - Left Left
Check for balance at 8, then at 9, then at 6.

9 is unbalanced.
AVL Trees

Insertion

Rebalancing - Left Left
Insert 4 into this tree:
Check for balance at 3, then at 5, then at 2, then at 6.

5 is unbalanced.
AVL Tree Insertion
Rebalancing - Left Right
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;
    int height;
    struct node *left;
    struct node *right;
};
```

Extra effort is required to maintain this data whenever the tree is modified.
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
AVL Tree Insertion
Maintaining Height Data - Insertions

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 \( \max(0, -1) + 1 = 1 \).
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 $\max(2, 1) + 1 = 3$. 
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 $\max(1, 1) + 1 = 2$. 
Every rotation, two height updates are performed, each in $O(1)$ time.
Analysis:

- An AVL tree is always height balanced
  - So height of an AVL tree is $O(\log n)$
- Checking/fixing balance and maintaining height data is $O(1)$
- So checking/fixing balance adds $O(1)$ extra work for each node on insertion path
- Therefore, worst-case time complexity of AVL tree insertion is $O(\log n)$
Exactly the same as for regular BSTs.

Worst-case time complexity is $O(\log n)$, since AVL trees are height-balanced
• AVL trees are always height-balanced
• Worst-case time complexity of $O(\log n)$ for insertion, search, deletion
• AVL trees are not necessarily weight-balanced, for example:
https://forms.office.com/r/aPF09YHZ3X