COMP2521 24T1
Balancing Binary Search Trees

Kevin Luxa
cs2521@cse.unsw.edu.au

balancing operations
balancing methods
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 $O(n)$. 

```
  •
  •
  •
  •
  •
  ...
  •
```
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 ($O(n)$).
Sizes of Balance

**SIZE-BALANCED**

A 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,

$$|\text{HEIGHT}(l) - \text{HEIGHT}(r)| \leq 1$$
Types of Balance

Example

Size-balanced?

Height-balanced?
**Types of Balance**

**Example**

Size-balanced?

Yes

For every node,

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

Height-balanced?

Yes
Size-balanced?
Yes

For every node,
\[|\text{SIZE}(l) - \text{SIZE}(r)| \leq 1\]

Height-balanced?
Yes

For every node,
\[|\text{HEIGHT}(l) - \text{HEIGHT}(r)| \leq 1\]
Size-balanced?

Height-balanced?

At node 4, \( |\text{size}(l) - \text{size}(r)| = |3 - 1| = 2 > 1 \) so it is not size-balanced.

For every node, \( |\text{height}(l) - \text{height}(r)| \leq 1 \) so it is height-balanced.
Size-balanced?  

No

At node 4,

\[ |\text{SIZE}(l) - \text{SIZE}(r)| = |3 - 1| = 2 > 1 \]

Height-balanced?

Yes

For every node,

\[ |\text{height}(l) - \text{height}(r)| \leq 1 \]
Types of Balance

Example

Size-balanced?
No

At node 4,
\[|\text{SIZE}(l) - \text{SIZE}(r)| = |3 - 1| = 2 > 1\]

Height-balanced?
Yes

For every node,
\[|\text{HEIGHT}(l) - \text{HEIGHT}(r)| \leq 1\]
Types of Balance

Example

Size-balanced?

Height-balanced?
Size-balanced?
No

At node 3,
\[|\text{SIZE}(l) - \text{SIZE}(r)| = |2 - 0| = 2 > 1\]

Height-balanced?
No

At node 3,
\[|\text{height}(l) - \text{height}(r)| = |1 - (-1)| = 2 > 1\]
Size-balanced?
No

At node 3,
\[ |\text{SIZE}(l) - \text{SIZE}(r)| = |2 - 0| = 2 > 1 \]

Height-balanced?
No

At node 3,
\[ |\text{HEIGHT}(l) - \text{HEIGHT}(r)| = |1 - (-1)| = 2 > 1 \]
Rotation
- Left rotation
- Right rotation

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

```
  n1
 /   \
|     |
|     |
|     |
|     |   \n|     |    t3
|     |     |
|     |     |
|     |     |
|   n2   |
|       |
|       |
|       |
|       |   t1
|       |     |
|       |     |
|       |     |
|       |     |
|       |     |   t2
|       |     |     |
|       |     |     |
|       |     |     |
|       |     |     |
|       |     |     |   t3
```

Right rotation

```
  n2
 /   \
|     |
|     |
|     |
|     |   \n|     |    t3
|     |     |
|     |     |
|     |     |
|   n1   |
|       |
|       |
|       |
|       |   t1
|       |     |
|       |     |
|       |     |
|       |     |
|       |     |   t2
|       |     |     |
|       |     |     |
|       |     |     |
|       |     |     |
|       |     |     |   t3
```

Left rotation
Rotations maintain the order of a search tree:

(all values in $t_1$) $< n_2$ $<$ (all values in $t_2$) $< n_1$ $<$ (all values in $t_3$)
Rotate right at 5
Rotate right at 5

Before rotation:

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

After rotation:

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

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

```
<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>
```
Rotate right at 23
Rotate right at 23

Before rotation:

```
    23
   /  
  7   30
 /   /  
3    16  35
 /     /     
11     28     10
```

After rotation:

```
     7
    / 
   3   23
  /   /  
16   11  20
 /       /  
10      28  35
```


```c
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;
}
```
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;
}
```c
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 *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;
}

rotations
implementation

struct node
struct node
struct node
struct node
struct node
struct node
struct node
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 *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;
}

newRoot

4

1   7

5  9
Time complexity: $O(1)$

- Rotation requires only a few localised pointer re-arrangements
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:

![Tree Diagram]

Partition Example
Partition this tree around index 3:
After right rotation at 30:
After left rotation at 14:
After left rotation at 10:
partition(t, i):

Input: tree t, index i
Output: tree with i-th item moved to root

m = size(t->left)
if i < m:
    t->left = partition(t->left, i)
    t = rotateRight(t)
else if i > m:
    t->right = partition(t->right, i - m - 1)
    t = 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;
    struct node *left;
    struct node *right;
    int size;
};
```
Balancing Methods

- 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 $SIZE(t)/2$, then rebalance both subtrees (recursively)
Global Rebalancing

First, partition on index \( n/2 \)...

...then rebalance both subtrees
**rebalance(t):**

*Input:* tree $t$

*Output:* rebalanced $t$

```pseudocode
if size($t$) < 3:
    return $t$

$t$ = partition($t$, size($t$) / 2)
$t$->left = rebalance($t$->left)
$t$->right = rebalance($t$->right)
return $t$
```
Global Rebalancing

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):

**Input:** tree t, value v

**Output:** t with v inserted

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

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

\[ \text{return } t \]
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

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
Root Insertion

Rotate right at 30

```
      10
     /  
    5   14
   /    |
  30    24
 /     /  
24   29   32
```

```
      10
     /  
    5   14
   /    |
  30    24
 /     /  
30   29   32
```
Rotate left at 14
Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion

Rotate left at 10

Before:
- 10
- 5
- 14
- 30
- 29
- 32

After:
- 24
- 10
- 14
- 30
- 5
- 14
- 29
- 32
- 30
insertAtRoot\( (t, v) \):

**Input:** tree \( t \), value \( v \)

**Output:** \( t \) with \( v \) inserted at the root

\[
\text{if } t \text{ is empty:} \\
\quad \text{return new node containing } v \\
\text{else if } v < t->\text{item:} \\
\quad t->\text{left} = \text{insertAtRoot}(t->\text{left}, v) \\
\quad t = \text{rotateRight}(t) \\
\text{else if } v > t->\text{item:} \\
\quad t->\text{right} = \text{insertAtRoot}(t->\text{right}, v) \\
\quad t = \text{rotateLeft}(t) \\
\]

\[
\text{return } t
\]
Analysis:

- Same time complexity as normal insertion: $O(h)$
- Tree is more likely to be balanced, but no guarantee
- Root insertion 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.

\[\text{insertRandom}(t, v):\]

\[\text{Input: tree } t, \text{ value } v\]
\[\text{Output: } t \text{ with } v \text{ inserted}\]

\[\text{if } t \text{ is empty:}\]
\[\quad \text{return new node containing } v\]

\[// \ p/q \text{ chance of inserting at root}\]
\[\text{if } \text{random()} \mod q < p: \]
\[\quad \text{return insertAtRoot}(t, v)\]
\[\text{else:}\]
\[\quad \text{return insertAtLeaf}(t, v)\]

Note: \text{random()} \text{ 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)
The balancing methods we have covered are either inefficient (global rebalancing), or don’t guarantee a balanced tree (root/randomised insertion)
https://forms.office.com/r/5c0fb4tvMb