# COMP2521 24T3
## Balancing Binary Search Trees

Hao Xue

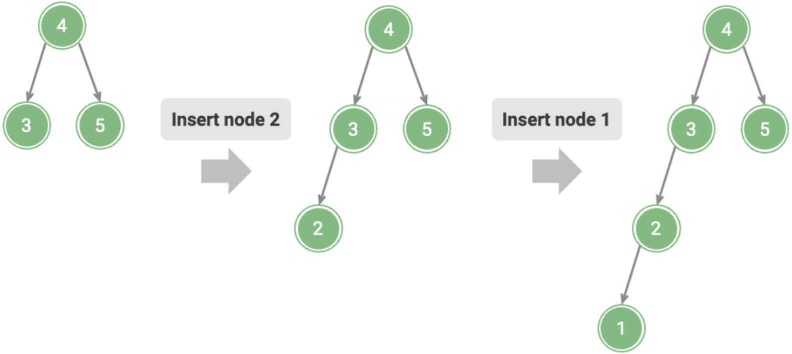
cs2521@cse.unsw.edu.au

balancing operations
balancing methods

# Binary Search Trees



The structure, height, and hence
performance
of a binary search tree
depends on the order of insertion.



# Example

Case 1 Items: 30, 40, 10, 50, 20, 5, 35

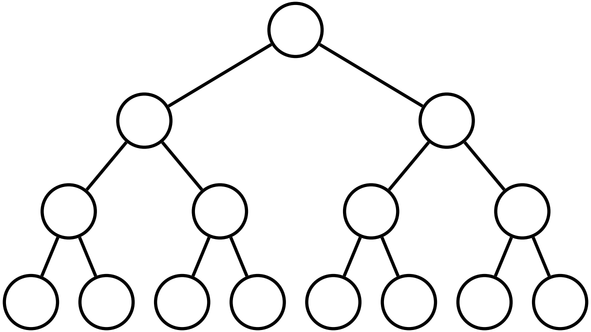
Case 2 Items: 50, 40, 35, 30, 20, 10, 5

# Binary Search Trees

## The Best Case

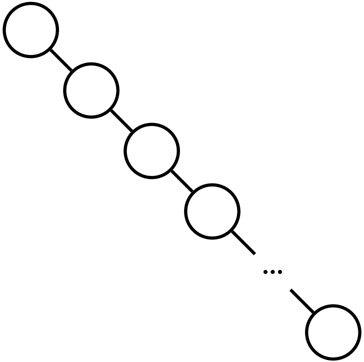
Best case

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

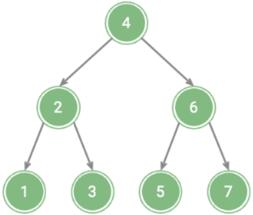


## Worst case

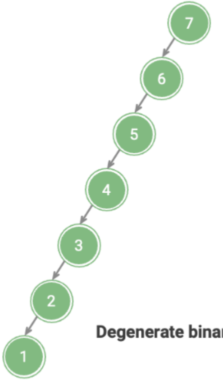
Items are inserted in ascending or descending order
such that tree consists of a single branch
Height of tree will be $O(n)$



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 ($O(n)$).




Balanced binary search tree




Degenerate binary search tree

# Types of Balance

**SIZE-BALANCED**

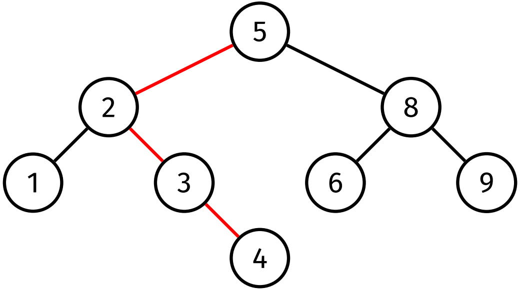
a *size-balanced* tree has,
for every node,
$$|\text{SIZE}(l) - \text{SIZE}(r)| \leq 1$$

**HEIGHT-BALANCED**

a *height-balanced* tree has,
for every node,
$$|\text{HEIGHT}(l) - \text{HEIGHT}(r)| \leq 1$$

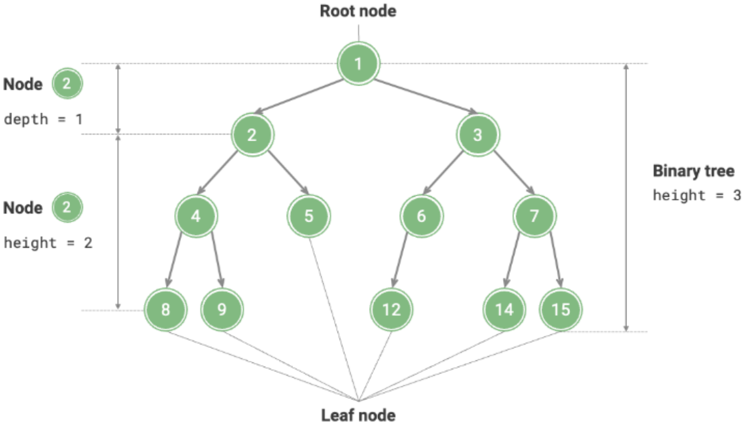
Height of a tree: Maximum path length from the root node to a leaf

- The height of an empty tree is considered to be -1
- The height of the following tree is 3



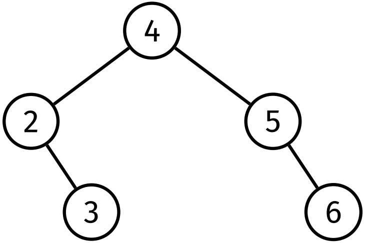
# Recap

## BST Height



# Types of Balance

## Example

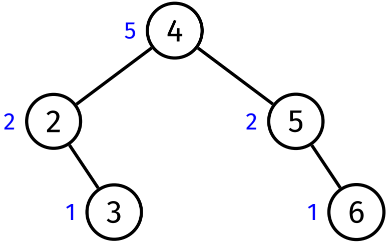



Size-balanced?

Height-balanced?

# Types of Balance

## Example

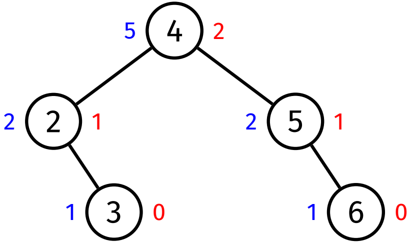



Size-balanced?
Yes

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

Height-balanced?

# Types of Balance
## Example



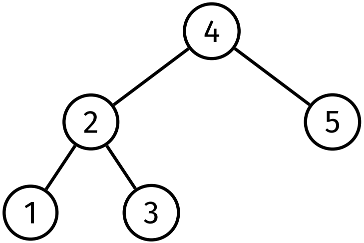

Size-balanced?
Yes

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

Height-balanced?
Yes

For every node,
$|\text{HEIGHT}(l) - \text{HEIGHT}(r)| \leq 1$

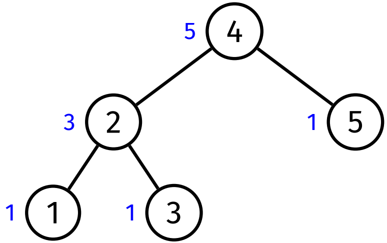
# Types of Balance
## Example




Size-balanced?

Height-balanced?

# Types of Balance
## Example




Size-balanced?

No

At node 4,

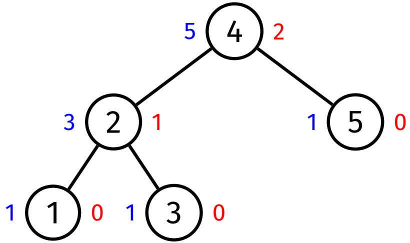
$|\text{SIZE}(l) - \text{SIZE}(r)|$

$= |3 - 1| = 2 > 1$

Height-balanced?

# Types of Balance

## Example




Size-balanced?
No

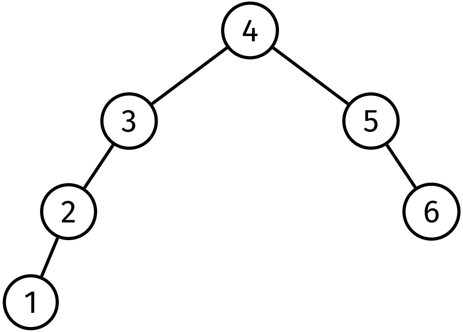
At node 4,
$$|\text{SIZE}(l) - \text{SIZE}(r)|$$
$$= |3 - 1| = 2 > 1$$

Height-balanced?
Yes

For every node,
$$|\text{HEIGHT}(l) - \text{HEIGHT}(r)| \leq 1$$

# Types of Balance

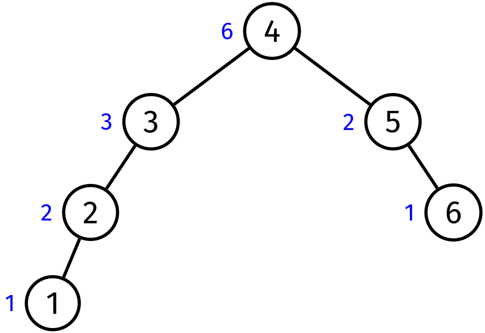
## Example




Size-balanced?

Height-balanced?

# Types of Balance
## Example



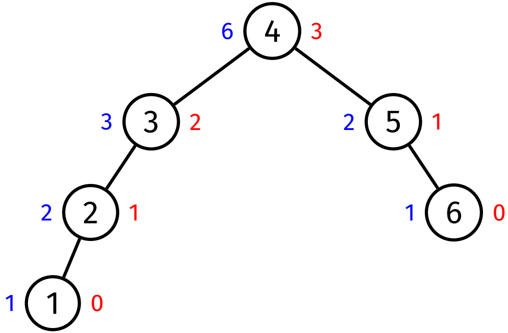

Size-balanced?
No

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

Height-balanced?

# Types of Balance

## Example




**Size-balanced?**

No

At node 3,

$$|\text{SIZE}(l) - \text{SIZE}(r)| = |2 - 0| = 2 > 1$$

**Height-balanced?**

No

At node 3,

$$|\text{HEIGHT}(l) - \text{HEIGHT}(r)| = |1 - (-1)| = 2 > 1$$

# Balancing Operations

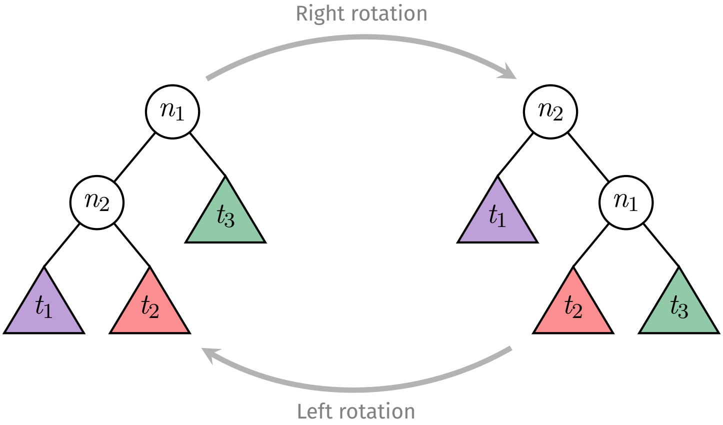
## Rotation

- Left rotation
- Right rotation

## Partition

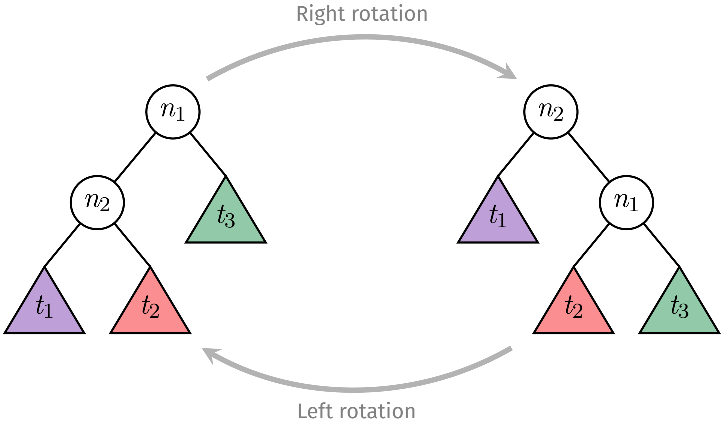
- Rearrange tree around a specified node by rotating it up to the root

LEFT ROTATION and RIGHT ROTATION:
a pair of operations
that change the balance of a tree



# Rotations

Rotations maintain the order of a search tree:



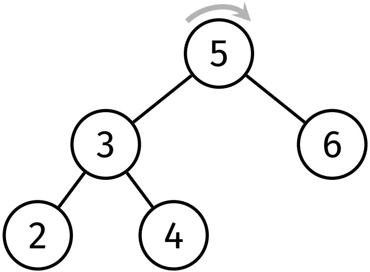

(all values in $t_1$) $< n_2 <$ (all values in $t_2$) $< n_1 <$ (all values in $t_3$)

# Rotations

## Example



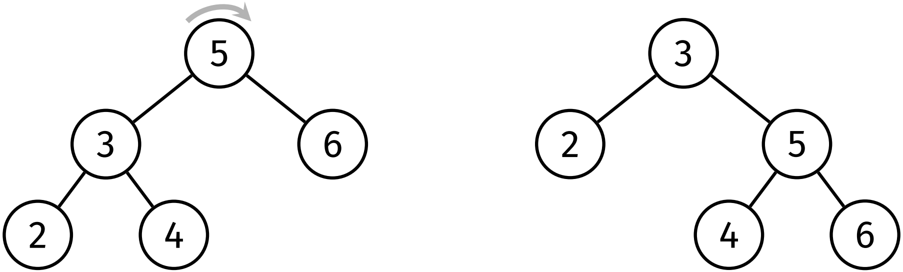
Rotate right at 5



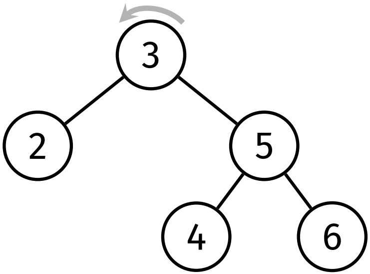
# Rotations

## Example

Rotate right at 5



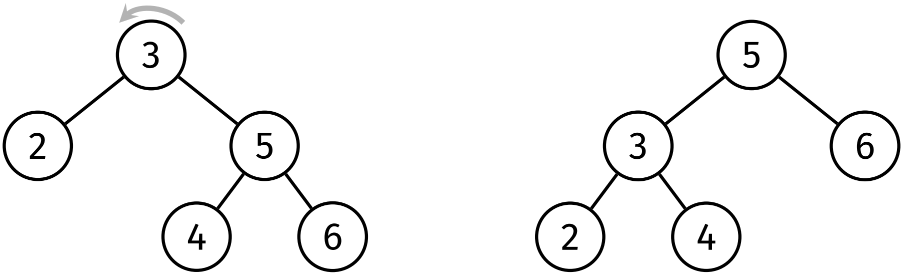
Rotate left at 3



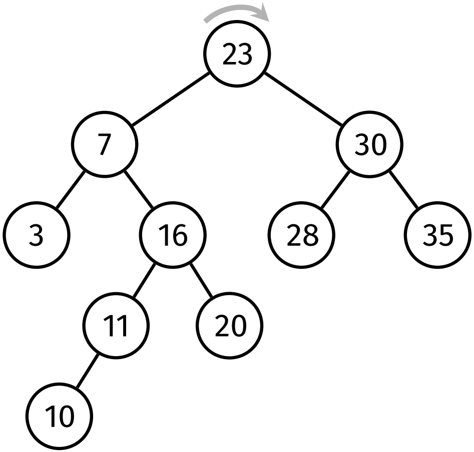
# Rotations

## Example

Rotate left at 3



Rotate right at 23

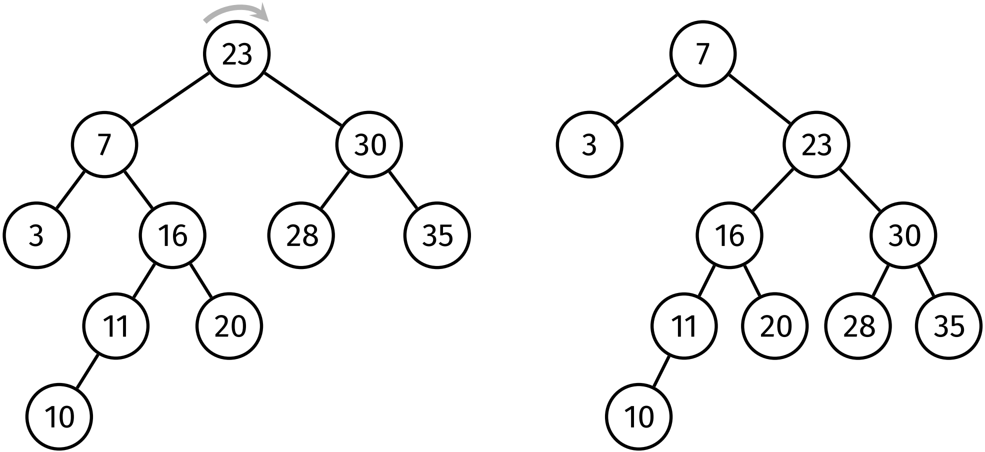


# Rotations

## Example



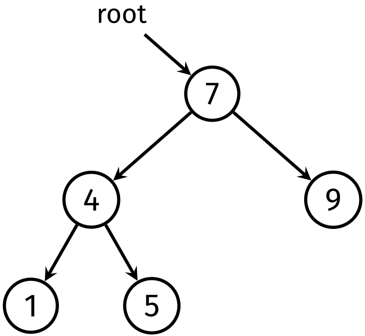
Rotate right at 23



```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;
}
```

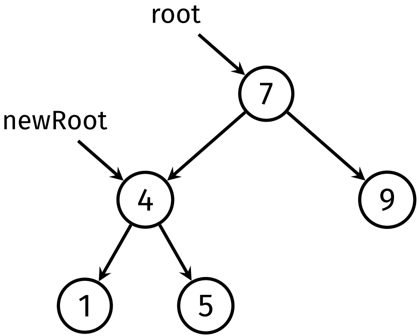
```c
struct node *rotateLeft(struct node *root) {
    if (root == NULL || root->right == NULL) return root;
    struct node *newRoot = root->right;
    root->right = newRoot->left;
    newRoot->left = root;
    return newRoot;
}
```

```
struct node *rotateRight(struct node *root) {
    if (root == NULL || root->left == NULL) return root;




}
```



# Rotations
## Implementation

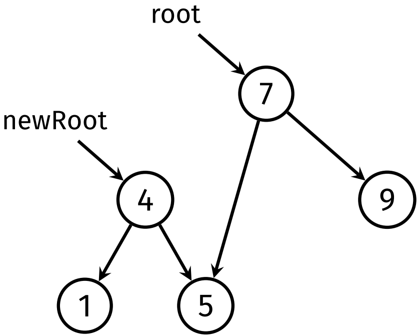
```
struct node *rotateRight(struct node *root) {
    if (root == NULL || root->left == NULL) return root;
    struct node *newRoot = root->left;



}
```



# Rotations

## Implementation

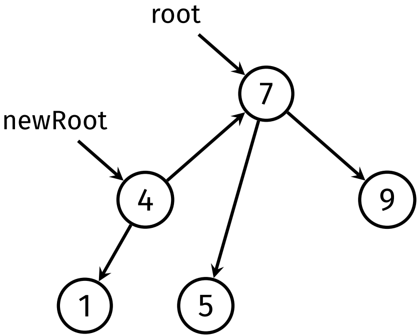
```
struct node *rotateRight(struct node *root) {
    if (root == NULL || root->left == NULL) return root;
    struct node *newRoot = root->left;
    root->left = newRoot->right;


}
```



# Rotations

## Implementation

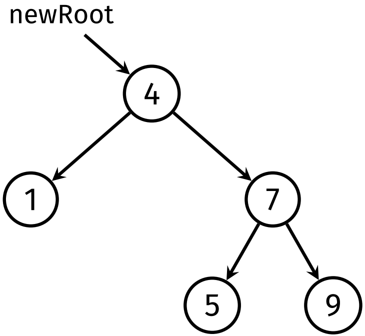
```c
struct node *rotateRight(struct node *root) {
    if (root == NULL || root->left == NULL) return root;
    struct node *newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;

}
```



# Rotations

## Implementation

```
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;
}
```

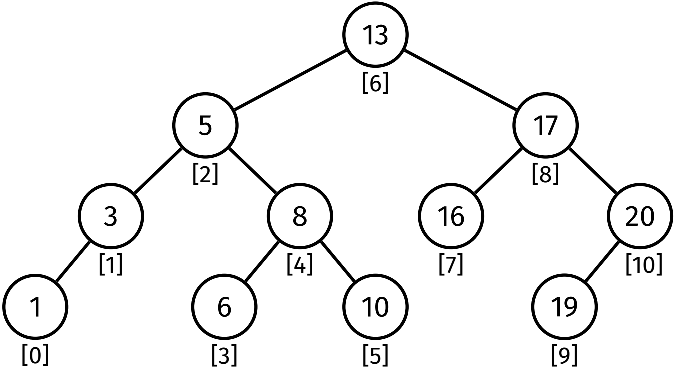


Time complexity: $O(1)$

- 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



# Partition

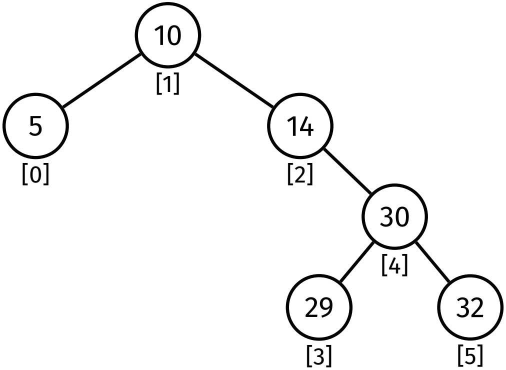
Method:

- 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

## Example

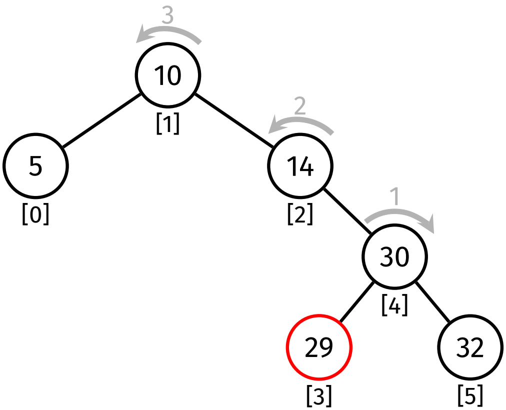
Partition this tree around index 3:



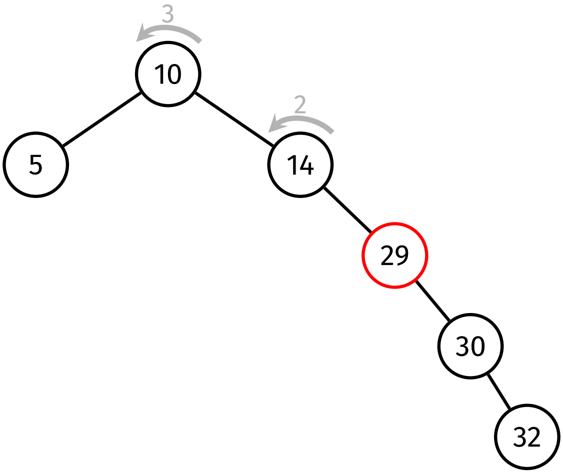
# Partition

## Example

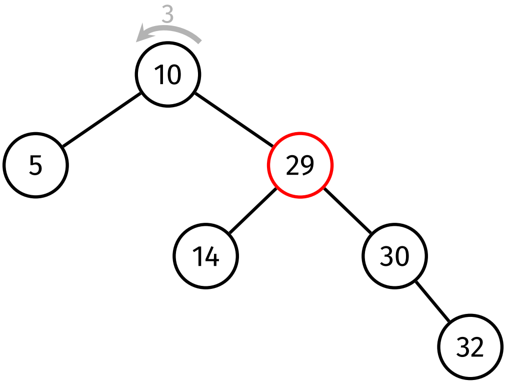
Partition this tree around index 3:



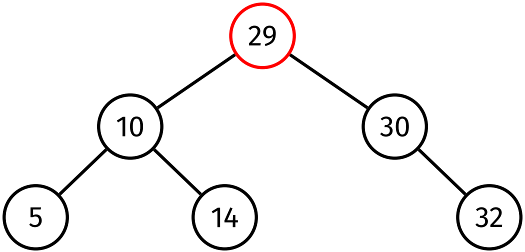
After right rotation at 30:



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
```

Assume $n$ nodes in our tree:
$[0, \cdots, m, m+1, \cdots, n-1]$
left tree: $[0, m]$
right tree: $[m+1, n-1]$

- $i < m$: index inside the left tree $\rightarrow$right rotate, remain $[0, \cdots, i, \cdots, m]$
- $i > m$: index inside the right tree $\rightarrow$left rotate, index $[m+1, \cdots, i, \cdots, n-1]$
- get the new index of $i$
- $[m+1-(m+1), \cdots, i-(m+1), \cdots, n-1-(m+1)]$
- Hence: $t$->right = partition($t$->right, $i$ - m - 1)

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

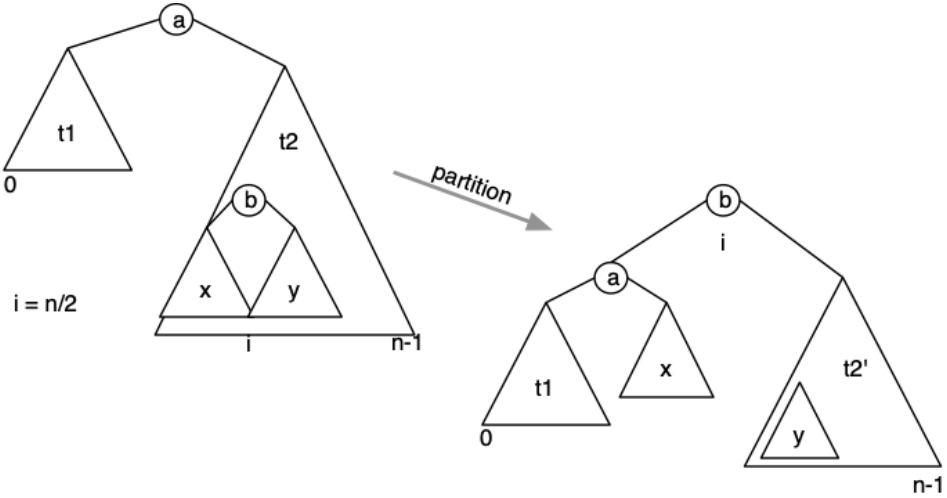
**Idea:**
Completely rebalance whole tree so it is size-balanced

**Method:**
Lift the median node to the root
by partitioning on $\textsc{size}(t)/2$,
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

    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

## Analysis

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 $\Rightarrow$ $O(n \log n)$

# Global Rebalancing

## Problems

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

# Periodic Rebalancing

```
bstInsert(t, v):
    Input:  tree t, value v
    Output: t with v inserted

    t = insertAtLeaf(t, v)

    if size(t) mod k = 0:
        t = rebalance(t)

    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 vs Local Rebalancing



GLOBAL REBALANCING

walks every node, balances its subtree;
$\Rightarrow$ 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

# Root Insertion



**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

# Root Insertion

**Method:**

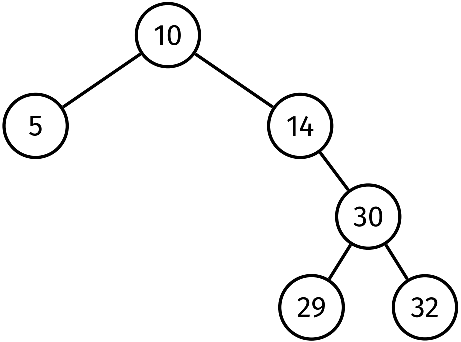
Insert new value normally (at the leaf) ...

... and then rotate the new node up to the root.

# Root Insertion

## Example

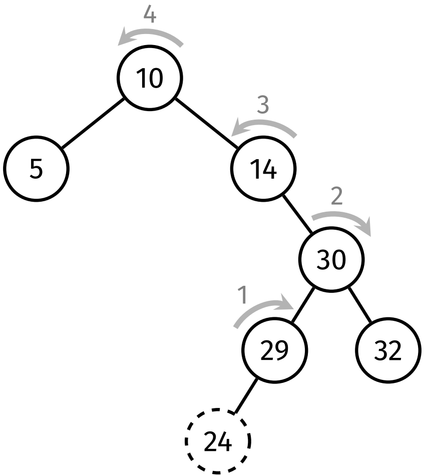
Insert 24 at the root of this tree:



# Root Insertion

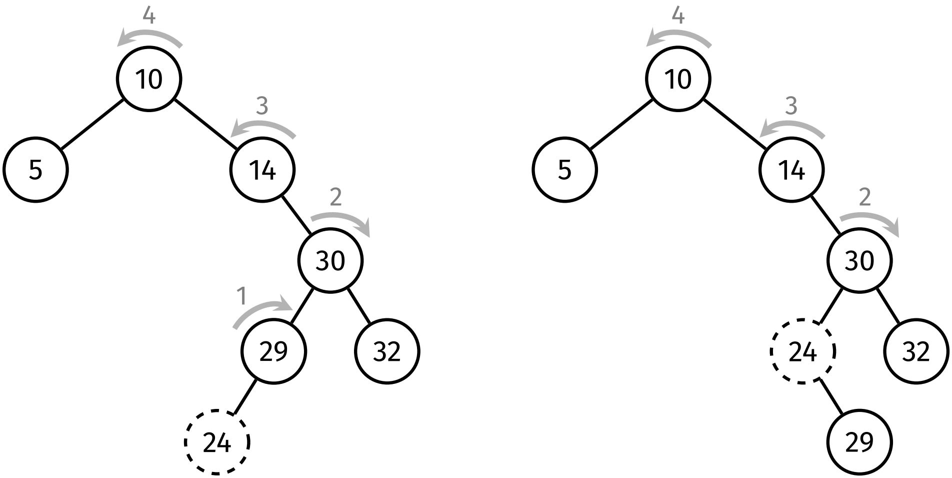
## Example

Insert 24 at the root of this tree:



# Root Insertion
## Example

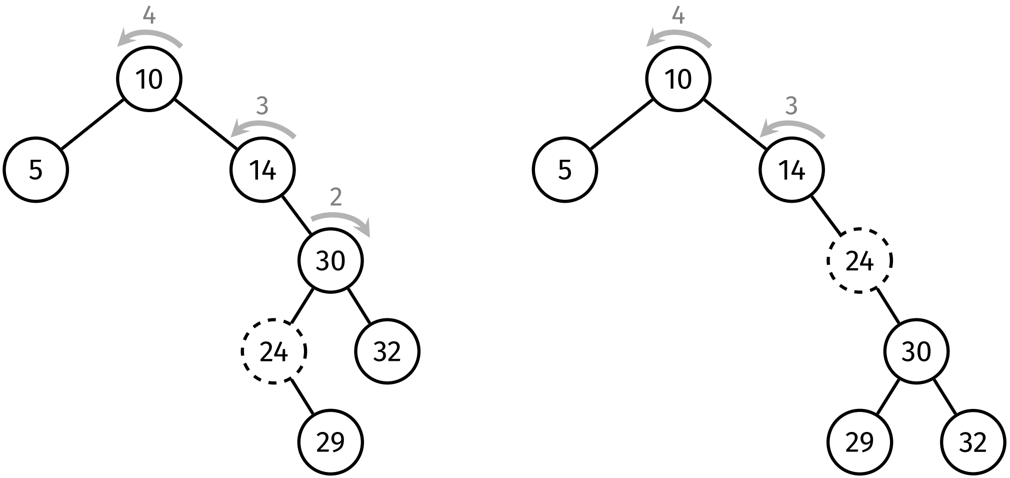
Rotate right at 29



# Root Insertion

## Example

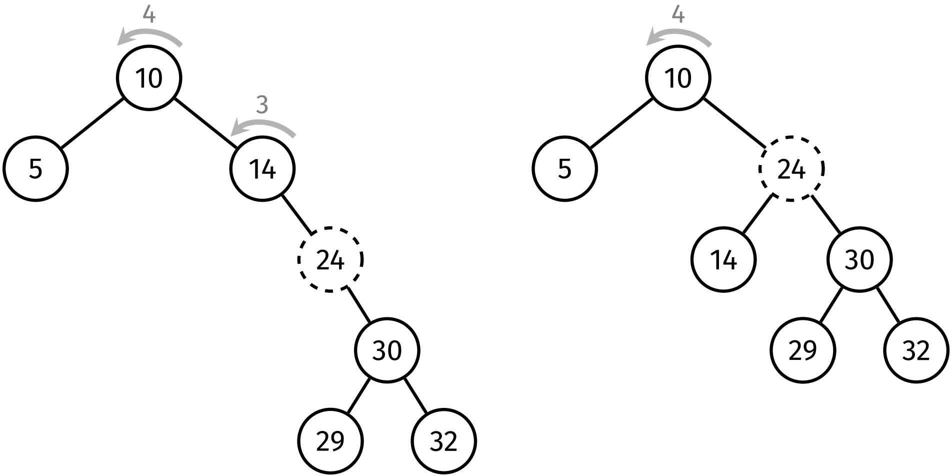
Rotate right at 30



# Root Insertion

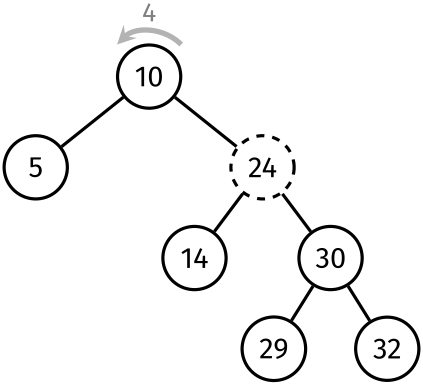
## Example

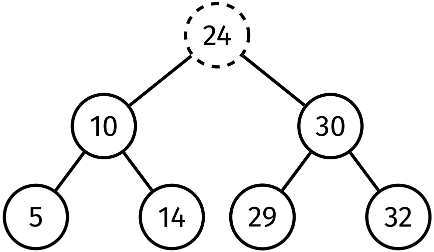
Rotate left at 14



# Root Insertion
## Example

Rotate left at 10



# Root Insertion

## Pseudocode

```
insertAtRoot(t, v):
    Input:  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
```

# Root Insertion

## Analysis

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

# Randomised 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

# Randomised Insertion

## Pseudocode

```
insertRandom(t, v):
    Input:  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 ⇒ choose p = 3, q = 10

# Randomised Insertion

## Remarks

Randomised insertion creates similar results to inserting items in random order.

Tree is more likely to be balanced (but no guarantee)

# Final Remarks

The balancing methods we have covered
are either inefficient (global rebalancing),
or don't guarantee a balanced tree (root/randomised insertion)

# Feedback

https://forms.office.com/r/zEqxUXvmLR