# COMP2521 19T0

## Week 7, Tuesday: A Question of Balance

Jashank Jeremy
jashank.jeremy@unsw.edu.au

radix sort
balanced trees



# Sorting



## Radix Sorting

Can we decompose our keys?
Radix sorts let us deal with this case.

Keys are values in some base-$R$ number system.
*e.g.*, binary, $R = 2$; decimal, $R = 10$;
ASCII, $R = 128$ or $R = 256$; Unicode, $R = 2^{16}$

Sorting individually on each part of the key at a time:
digit-by-digit, character-by-character, rune-by-rune, *etc.*

# Radix Sorting, Most-Significant-Digit First

Consider characters, digits, bits, runes, *etc.*,
from left to right;
partitioning input into $R$ pieces according to `key.0`;
recurse into each piece, using succesive keys —
`key.1`, `key.2`, ..., `key.`$w$

| | | | |
|---|---|---|---|
| 1019 | 2301 | 3129 | 2122 |
| **1**019 | **2**301 | **3**129 | **2**122 |
| **10**19 | **23**01 | **21**22 | **31**29 |
| 1019 | 2122 | 2301 | 3129 |

with $R = 2$, roughly a quicksort.



# Radix Sorting, Least-Significant-Digit First

Consider characters, digits, bits, runes, *etc.*,
from right to left;
use a **stable** sort using the $d$th digit as key,
using (*e.g.*,) key-indexed counting sort.

| | | | |
|---|---|---|---|
| 1019 | 2301 | 3129 | 2122 |
| 101**9** | 230**1** | 312**9** | 212**2** |
| 23**01** | 21**22** | 10**19** | 31**29** |
| 2**301** | 1**019** | 2**122** | 3**129** |
| **1019** | **2122** | **3129** | **2301** |
| 1019 | 2122 | 2301 | 3129 |

this *will not work* if the sort is not stable!



# Radix Sort

## Analysis; Summary

Complexity: $O\left(w\left(n+R\right)\right) \approx O(n)$,
where $w$ is the 'width' of data;
the algorithm makes $w$ passes over $n$ keys

LSD

*Not* in-place: $O(n+R)$ extra space required.
May be stable! Usable on variable length data.

MSD

*Not* in-place: $O(n+DR)$ extra space required.
($D$ is the recursion depth.)
May be stable! Usable on variable length data.
Can complete *before* examining all of all keys.

# Balanced Trees



## Recap: The Search Problem

**input** a key value
**output** item(s) containing that key

Common variations:

- keys are unique; key value matches 0 or 1 items
- multiple keys in search, items containing any key
- multiple keys in search/item, items containing all keys

We assume: keys are unique, each item has one key.



## Recap: Trees; Binary Trees

Trees are branched data structures,
consisting of nodes and edges, with no cycles.

Each node contains a value.
Each node has edges to $\leq k$ other nodes.
For now, $k = 2$ — binary trees

Trees can be viewed as a set of nested structures:
each node has $k$ (possibly empty) subtrees.

## Recap: Binary Search Trees

For all nodes in the tree:

the values in the left subtree are less than the node value the

values in the right subtree are greater than the node value

A binary tree of $n$ nodes is degenerate if its height is at most $n - 1$.

A binary tree of $n$ nodes is balanced if its height is at least $\lfloor \log_2 n \rfloor$.

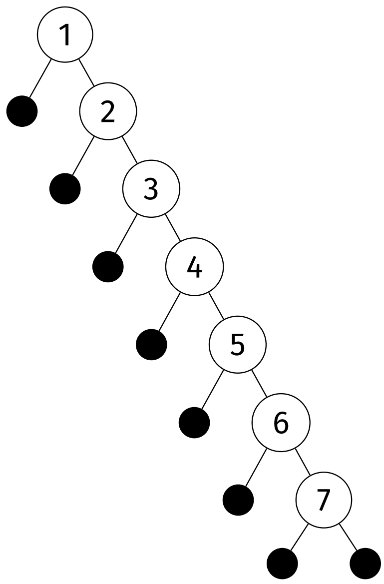
Structure tends to be determined by order of insertion:
$[4, 2, 1, 3, 6, 5, 7]$ vs $[6, 5, 2, 1, 3, 4, 7]$

COMP2521 19T0 lec12
cs2521@ jashankj@
Sorting
Balanced Trees
Recap
Searching
Trees, BTrees
Search Trees
Properties
Primitives
Rotation
Partition
Simple Approaches
Global
Root Insert
Random Trees
Complex Approaches
Splay

## Binary Search Trees

### The Worst Case

Ascending-ordered or descending-ordered data is a pathological case: we always right- or left-insert along the spine of the tree.




COMP2521 19T0 lec12
cs2521@ jashankj@
Sorting
Balanced Trees
Recap
Searching
Trees, BTrees
Search Trees
Properties
Primitives
Rotation
Partition
Simple Approaches
Global
Root Insert
Random Trees
Complex Approaches
Splay

## Binary Search Trees

### Performance

Cost for insertion:
balanced $O(\log_2 n)$, degenerate $O(n)$
(we always traverse the height of the tree)

Cost for search/deletion:
balanced $O(\log_2 n)$, degenerate $O(n)$
(worst case, key $\notin \tau$; traverse the height)

We want to build balanced trees.

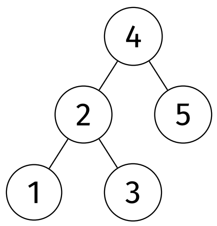
## What is 'Balance', then?

PERFECTLY BALANCED
a *weight-balanced* or *size-balanced* tree has, for every node,
$$|\text{SIZE}\,(l) - \text{SIZE}\,(r)| < 2$$

LESS STRINGENTLY
a *height-balanced* tree has, for every node,
$$|\text{HEIGHT}\,(l) - \text{HEIGHT}\,(r)| < 2$$



## Balanced or Not? (I)




$$\text{SIZE}\,(\tau_4) = 5$$
$$\text{SIZE}\,(\tau_2) = 3$$
$$\text{SIZE}\,(\tau_5) = 1$$
$$\text{SIZE}\,(\tau_1) = 1$$
$$\text{SIZE}\,(\tau_3) = 1$$
$$\text{SIZE}\,(\tau_2) - \text{SIZE}\,(\tau_5) = 2$$

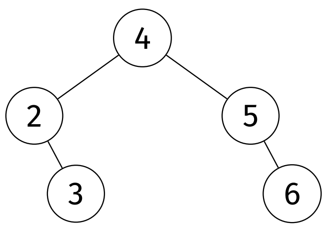
NOT SIZE BALANCED

$$\text{HEIGHT}\,(\tau_4) = 2$$
$$\text{HEIGHT}\,(\tau_2) = 1$$
$$\text{HEIGHT}\,(\tau_5) = 0$$
$$\text{HEIGHT}\,(\tau_1) = 0$$
$$\text{HEIGHT}\,(\tau_3) = 0$$
$$\text{HEIGHT}\,(\tau_2) - \text{HEIGHT}\,(\tau_5) = 1$$

HEIGHT BALANCED



## Balanced or Not? (II)



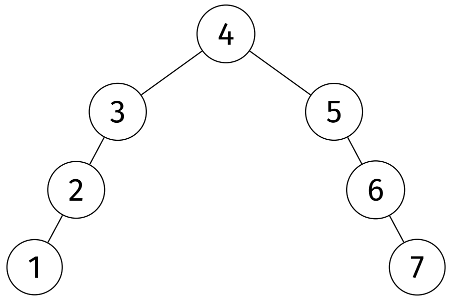

$$\text{SIZE}\,(\tau_4) = 5$$
$$\text{SIZE}\,(\tau_2) = 2$$
$$\text{SIZE}\,(\tau_5) = 2$$
$$\text{SIZE}\,(\tau_3) = 1$$
$$\text{SIZE}\,(\tau_6) = 1$$

SIZE BALANCED

$$\text{HEIGHT}\,(\tau_4) = 2$$
$$\text{HEIGHT}\,(\tau_2) = 1$$
$$\text{HEIGHT}\,(\tau_5) = 1$$
$$\text{HEIGHT}\,(\tau_3) = 0$$
$$\text{HEIGHT}\,(\tau_6) = 0$$

HEIGHT BALANCED

## Balanced or Not?
(III)



Let's look at $\tau_3$.

$\text{SIZE}\left(\tau_2\right) = 2$

$\text{SIZE}\left(\tau_\varnothing\right) = 0$

$2 - 0 = 2 \not< 2$

NOT SIZE BALANCED

Let's look at $\tau_5$.

$\text{HEIGHT}\left(\tau_\varnothing\right) = 0$

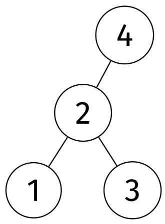
$\text{HEIGHT}\left(\tau_6\right) = 1$

$|0 - 1| = 1 < 2$

HEIGHT BALANCED



## Balanced or Not?
(IV)



Let's look at $\tau_4$.

$\text{SIZE}\left(\tau_2\right) = 3$

$\text{SIZE}\left(\tau_\varnothing\right) = 0$

$3 - 0 = 3 \not< 2$

NOT SIZE BALANCED

Let's look at $\tau_4$.

$\text{HEIGHT}\left(\tau_2\right) = 1$ $\text{HEIGHT}\left(\tau_\varnothing\right) = 0$

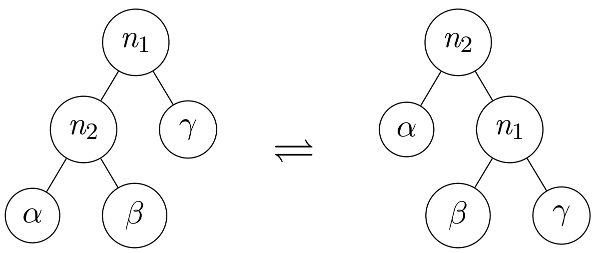
$1 - 0 = 1 < 2$

HEIGHT BALANCED



## Rotations
Rebalancing Primitives

LEFT ROTATION and RIGHT ROTATION:
a pair of 'primitive' operations
that change the balance of a tree
whilst maintaining a search tree.



$$\left(n_1, \left(n_2, \alpha, \beta\right), \gamma\right) \rightleftharpoons \left(n_2, \alpha, \left(n_1, \beta, \gamma\right)\right)$$

# Rotations

## Rotating Right

```
btree_node *btree_rotate_right (btree_node *n1)
{
    if (n1 == NULL) return NULL;
    btree_node *n2 = n1->left;
    if (n2 == NULL) return n1;
    n1->left = n2->right;
    n2->right = n1;
    return n2;
}
```

$n_1$ starts as the root of this subtree and is demoted;
$n_2$ starts as the left subtree of this tree, and is promoted.



# Rotations

## Rotating Left

```
btree_node *btree_rotate_left (btree_node *n2)
{
    if (n2 == NULL) return NULL;
    btree_node *n1 = n2->right;
    if (n1 == NULL) return n2;
    n2->right = n1->left;
    n1->left = n2;
    return n1;
}
```

$n_2$ starts as the root of this subtree and is demoted;
$n_1$ starts as the right subtree of this tree, and is promoted.



# Partition

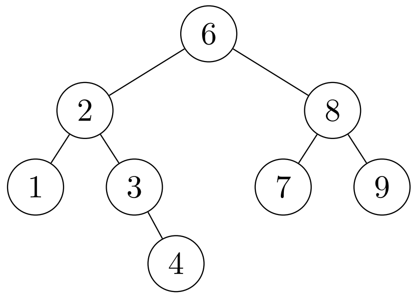
## Rotation in Context

A way to brute-force some balance into a tree:
lifting some $k$th index to the root.

$$
\begin{aligned}
&\textsc{partition} :: \text{BTree} \to \text{Word} \to \text{BTree} \\
&\textsc{partition}\ \text{Empty}\ k = \text{Empty} \\
&\textsc{partition}\ \ (\text{Node}\ n\ l\ r)\ \ k \\
&\quad |\ k < \textsc{size}\ l = \textsc{rotateR}\ (\text{Node}\ n\ \ (\textsc{partition}\ l\ k)\ \ r) \\
&\quad |\ \textsc{size}\ l < k = \textsc{rotateL}\ (\text{Node}\ n\ l\ \ (\textsc{partition}\ r\ \ (k - 1 - \textsc{size}\ l))) \\
&\quad |\ \text{otherwise} = \text{Node}\ n\ l\ r
\end{aligned}
$$

# Partition

## Partition in Context




What happens if we partition at index 3 (node $4$)?



# Partition

## C Implementation

```c
btree_node *btree_partition (btree_node *tree, size_t k)
{
    if (tree == NULL) return NULL;
    size_t lsize = size (tree->left);
    if (lsize > k) {
        tree->left = btree_partition (tree->left, k);
        tree = btree_rotate_right (tree);
    }
    if (lsize < k) {
        tree->right = btree_partition (tree->right, k - 1 - lsize);
        tree = btree_rotate_left (tree);
    }
    return tree;
}
```



# Primitives

With our primitive operations in hand —

$$\textsc{RotateL} :: \mathrm{BTree} \to \mathrm{BTree}$$
$$\textsc{RotateR} :: \mathrm{BTree} \to \mathrm{BTree}$$
$$\textsc{Partition} :: \mathrm{BTree} \to \mathrm{Word} \to \mathrm{BTree}$$

— let’s go balance some trees!

# Approach #1: Global Rebalancing

Move the median node to the root,
by partitioning on $\text{SIZE}\ \tau/2$;
then, balance the left subtree,
and balance the right subtree.

```
btree_node *btree_balance_global (btree_node *tree)
{
    if (tree == NULL) return NULL;
    if (size (tree) < 2) return tree;
    tree = partition (tree, size (tree) / 2);
    tree->left = btree_balance_global (tree->left);
    tree->right = btree_balance_global (tree->right);
    return tree;
}
```



# Approach #1: Global Rebalancing

## Problems

- cost of rebalancing:
  for many trees, $O(n)$; for degenerate trees, $O(n \log n)$
- what if we insert more keys?
  - rebalance on every insertion
  - rebalance every $k$ insertions; what $k$ is good?
  - rebalance when imbalance exceeds threshold.

  we either have more costly instertions
  or degraded performance for (possibly unbounded) periods.
  … given a sufficiently dynamic tree, sadness.



# 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

## Local Rebalancing Approaches

amortisation: do (a small amount) more work now to avoid more work later
randomisation: use randomness to reduce impact of BST worst cases
optimisation: maintain structural information for performance



## Root Insertion

How do we insert a node at the root of a tree?
(Without having to rearrange all the nodes?)

We do a leaf insertion ...
... and rotate the new node up the tree.

More work? **No!**
Same complexity as leaf insertion,
but more actual work is done: amortisation.

(Side-effect: recently-inserted items are close to the root.
Depending on what you're doing, this might be very useful!)



## Root Insertion

### C Implementation

```c
btree_node *btree_insert_root (btree_node *tree, Item it)
{
    if (tree == NULL)
        return btree_node_new (it, NULL, NULL);
    if (less (it, tree->value)) {
        tree->left = btree_insert_root (tree->left, it);
        tree = btree_rotate_right (tree);
    } else {
        tree->right = btree_insert_root (tree->right, it);
        tree = btree_rotate_left (tree);
    }
    return tree;
}
```

# Randomised Insertion

BSTs don't have control over insertion order.
worst cases — (partially) ordered data — are common.

to minimise the likelihood of a degenerate tree,
we randomly choose which level to insert a node;
at each level, probability depends on remaining tree size.

do a 'normal' leaf insertion, most of the time.
randomly (with a certain probability),
do a root insertion of a value.



# Randomised Insertion

## C Implementation

```
btree_node *btree_insert_rand (btree_node *tree, Item it)
{
    if (tree == NULL)
        return btree_node_new (it, NULL, NULL);
    if (rand () < (RAND_MAX / size (tree)))
        return btree_insert_root (tree, it);
    else if (less (it, tree->value))
        tree->left = btree_insert_rand (tree->left, it);
    else
        tree->right = btree_insert_rand (tree->right, it);
    return tree;
}
```



# Randomised Insertion

## Properties

building a randomised BST is equivalent to
building a standard BST with
a random initial permutation of keys.

worst-case, best-case, average-case performance:
same as a standard BST —
but with no penalty for ordering!

# Randomised Deletion

We could do something similar for deletion:
when choosing a node to promote,
choose randomly from the
in-order predecessor or successor



# Splay Trees

Root insertion can still leave us
with a degenerate tree.

Splay trees vary root-insertion,
by considering *three* levels of the tree
— parent, child, grandchild —
and performing double-rotations based on p-c-g orientation;
the idea: double-rotations improve balance.

No guarantees, but *improved* performance.

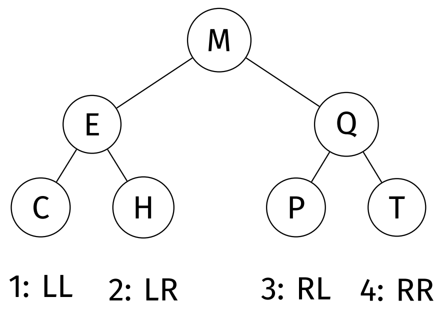
"... their performance is amortised by
the amount of effort required to understand them."
— me, 2016



# Splay Trees

## Double-Rotation Cases

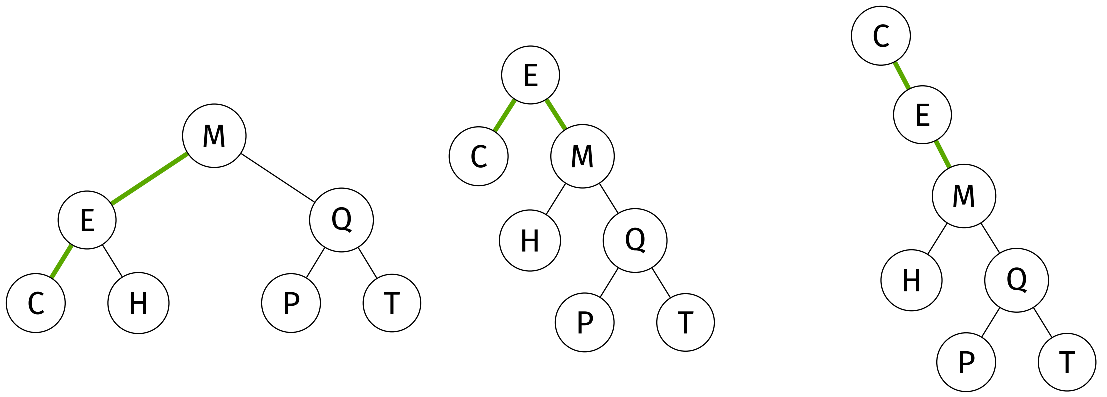
Four choices to consider for a double-rotation:



# Splay Rotations

## Double-Rotation: Left, Left

ROTATER $\tau_M$   ROTATER $\tau_E$

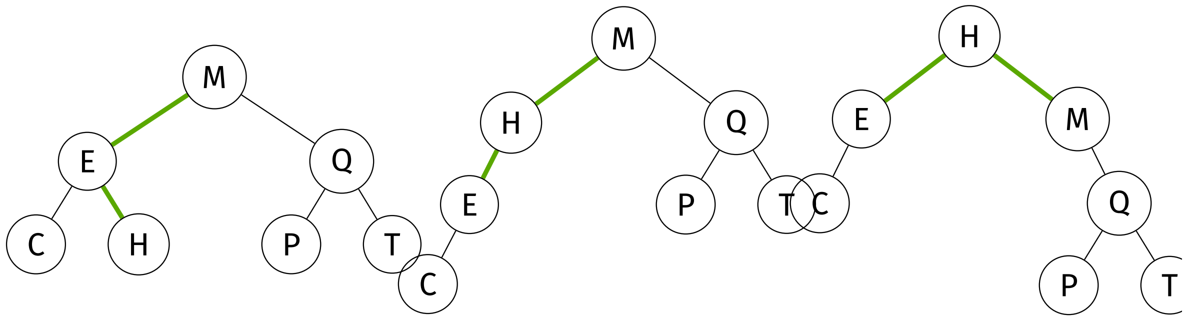





# Splay Rotations

## Double-Rotation: Left, Right

ROTATEL $\tau_E$   ROTATER $\tau_M$

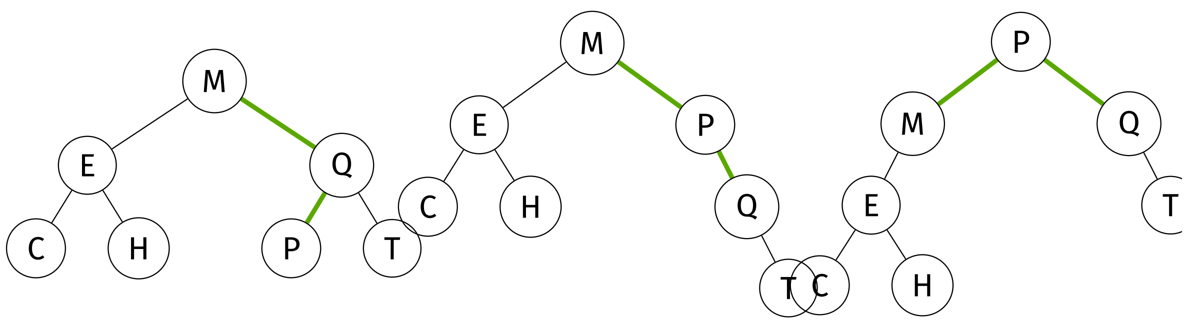





# Splay Rotations

## Double-Rotation: Right, Left

ROTATER $\tau_Q$   ROTATEL $\tau_M$



# Splay Rotations

## Double-Rotation: Right, Right

$\textsc{rotateL}\ \tau_{\mathsf{M}} \quad \textsc{rotateL}\ \tau_{\mathsf{Q}}$