CS 240 – Data Structures and Data Management

Module 5: Other Dictionary Implementations - Enriched

T. Biedl E. Kondratovsky M. Petrick O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2022

version 2022−01−31 10:09

Outline

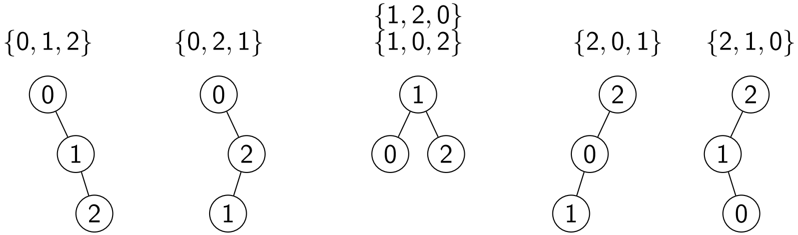
1. Even more Dictionary implementations
   - Expected height of a BST
   - Treaps
   - Optimal static binary search trees
   - MTF-heuristic in a BST
   - Splay Trees

Outline

1. Even more Dictionary implementations
   - Expected height of a BST
   - Treaps
   - Optimal static binary search trees
   - MTF-heuristic in a BST
   - Splay Trees

Expected height of BSTs

Assume we *randomly* choose a permutation of $\{0, \ldots, n-1\}$ and build a binary search tree in this order:



Theorem: The expected height of the tree is $O(\log n)$.
Proof:

Expected height vs. average height

This does *not* imply that the average height of a BST is $O(\log n)$.

- Can show: Average height is $\Theta(\sqrt{n})$ (no details).
- Average height (over all BSTs)
  $\neq$ expected height (over all randomly built BSTs)

Expected height vs. average height

This does *not* imply that the average height of a BST is $O(\log n)$.

- Can show: Average height is $\Theta(\sqrt{n})$ (no details).
- Average height (over all BSTs)
  $\neq$ expected height (over all randomly built BSTs)
- Difference already obvious for $n = 3$:
  - Expected height is $\frac{1}{6}(2+2+1+1+2+2) \approx 1.66$.
    6 possible permutations.
  - Average height is $\frac{1}{5}(2+2+1+2+2) = 1.8$.
    5 possible binary search trees.

- Message: Randomization does *not* automatically imply an average-case bound.
  (It depends on what we average over and how we randomize.)

Outline

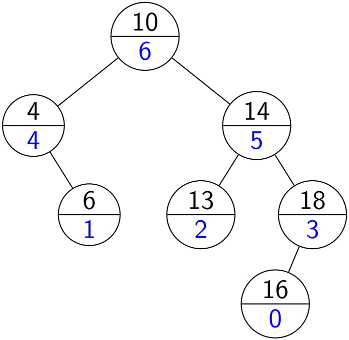
1. Even more Dictionary implementations
   - Expected height of a BST
   - Treaps
   - Optimal static binary search trees
   - MTF-heuristic in a BST
   - Splay Trees

Treaps

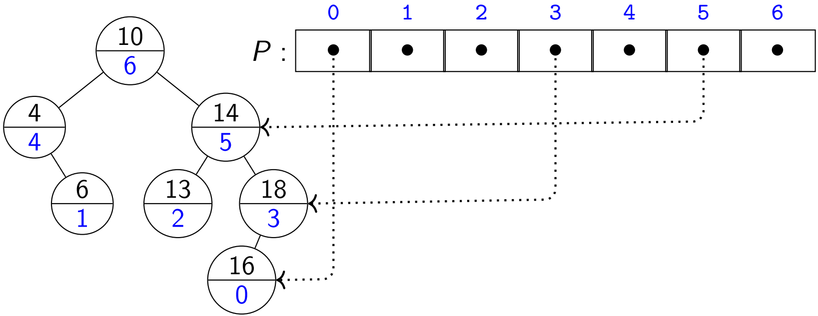
Goal: Build a binary search tree that acts as if it had been build in randomly picked insertion order.

Idea: Use binary search tree, but store a priority with each node.

- Priorities are a permutation of $\{0, \ldots, n-1\}$.
- Permutation has been picked *randomly*
- All permutations should be equally likely.
- Priorities are *decreasing* when going downwards (similar to a heap).

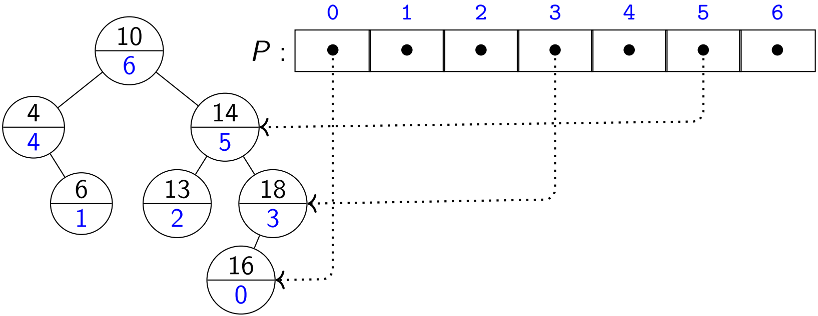


Treaps



- We will also need an array P where $P[i]$ stores node with priority i.
- We call this a **treap** (= tree + heap).

Treaps



- We will also need an array P where $P[i]$ stores node with priority i.
- We call this a **treap** (= tree + heap).

Theorem: The expected height of a treap is $O(\log n)$.

Proof: Root-item has priority $n-1$. This is picked randomly, so proof for expected height of BST applies.

Treap Insertion

Consider adding a new KVP. What priority should it get?

- We need a random permutation of $\{0, \ldots, n-1\}$
  - Currently we had a random permutation of $\{0, \ldots, n-2\}$.

Treap Insertion

Consider adding a new KVP. What priority should it get?

- We need a random permutation of $\{0, \ldots, n-1\}$
  - Currently we had a random permutation of $\{0, \ldots, n-2\}$.
- Recall *shuffle* from long ago:

```
shuffle(A)
A: array of size n stores ⟨0, ...n−1⟩
1.    for i ← 1 to n − 1 do
2.        swap( A[i], A[random(i + 1)] )
```

- In ith round,
  - have random permutation of $\{0, \ldots, i-1\}$
  - build random permutation of $\{0, \ldots, i\}$ in $O(1)$ time
  - key insight: swap with randomly chosen item

Treap Insertion

Consider adding a new KVP. What priority should it get?

- We need a random permutation of $\{0, \ldots, n-1\}$
  - Currently we had a random permutation of $\{0, \ldots, n-2\}$.
- Recall *shuffle* from long ago:

shuffle(A)
A: array of size n stores $\langle 0, \ldots n{-}1 \rangle$
    for $i \leftarrow 1$ to $n-1$ **do**
        swap($A[i], A[random(i+1)]$)

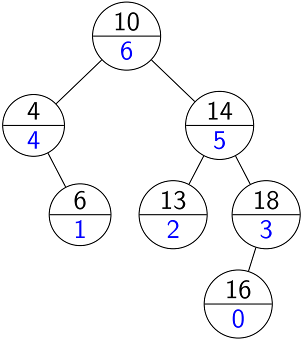
- In ith round,
  - have random permutation of $\{0, \ldots, i-1\}$
  - build random permutation of $\{0, \ldots, i\}$ in $O(1)$ time
  - key insight: swap with randomly chosen item

We can do the same by *randomly* picking priority p for new item.

- The item that had priority p previously now has priority $n-1$.
- If this violates the heap-property, then rotate to fix it.

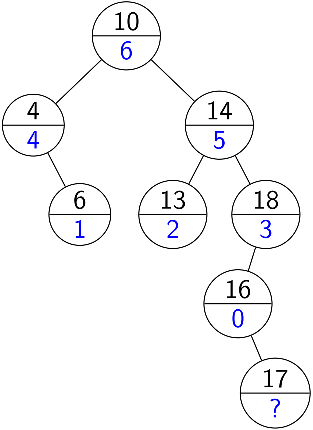
Treap Insertions Example

Example: *treap::insert*(17)



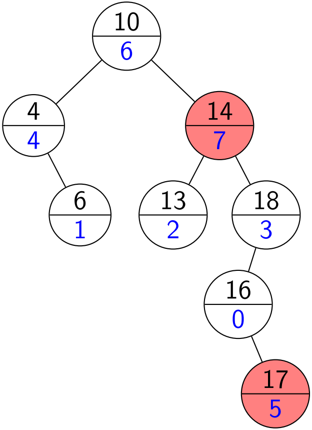
Treap Insertions Example

Example: *treap::insert*(17)



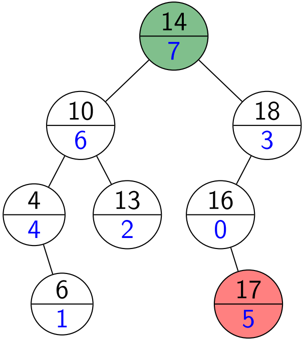
Treap Insertions Example

Example: *treap::insert*(17)
Randomly pick priority $5 \in \{0, \ldots, 7\}$



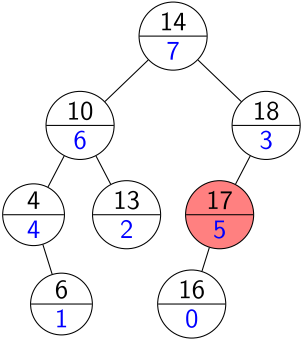
Treap Insertions Example

Example: *treap::insert*(17)
Randomly pick priority $5 \in \{0, \ldots, 7\}$



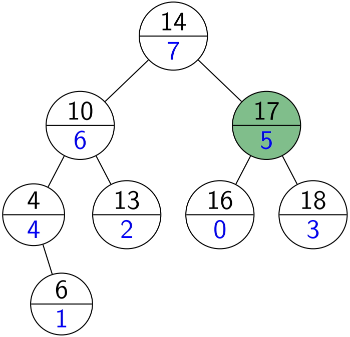
Treap Insertions Example

Example: *treap::insert*(17)
Randomly pick priority $5 \in \{0, \ldots, 7\}$



Treap Insertions Example

Example: *treap::insert*(17)
Randomly pick priority $5 \in \{0, \ldots, 7\}$



Treap Insertion Code

We assume that the treap stores array where $P[i]$ = node with priority i.

treap::insert(k, v)
- $n \leftarrow P.size$ // current size
- $z \leftarrow$ *BST::insert*(k, v); $n{+}{+}$
- $p \leftarrow$ *random*(n)
- **if** $p < n-1$ **do**
  - $z' \leftarrow P[p]$, $z'.priority \leftarrow n-1$, $P[n-1] \leftarrow z'$
  - *fixUpWithRotations*(z')
- $z.priority \leftarrow p$; $P[p] \leftarrow z$
- *fixUpWithRotations*(z)

treap::fixUpWithRotations(z)
- **while** ($y \leftarrow z.parent$ is not `NIL` and $z.priority > y.priority$) **do**
  - **if** z is the left child of y **do** *rotate-right*(y)
  - **else** *rotate-left*(y)

Treaps summary

- Randomized binary search tree, so expected height is $O(\log n)$
- Achieves $O(\log n)$ expected time for *search* and *insert*
- *delete* can be handled similar (but even more exchanges)

Treaps summary

- Randomized binary search tree, so expected height is $O(\log n)$
- Achieves $O(\log n)$ expected time for *search* and *insert*
- *delete* can be handled similar (but even more exchanges)

- Large space overhead (parent-pointers, priorities, P)
- Not particularly efficient in practice
(except when priorities have meaning $\rightsquigarrow$ later)
- There are ways to avoid some of the space overhead, but in general randomized binary search trees are rarely used.
- We will soon see a randomization that works better (but is not a binary search tree)

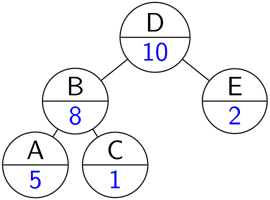
Outline

1. Even more Dictionary implementations
   - Expected height of a BST
   - Treaps
   - Optimal static binary search trees
   - MTF-heuristic in a BST
   - Splay Trees

Optimal static binary search trees

- Can we find the optimal static order for a binary search tree?

k_i	A	B	C	D	E
$P(k_i)$	$\frac{5}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{2}{26}$



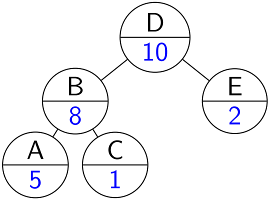
- Access-cost is now $\sum_k P(k) \cdot (1 + \text{depth of } k)$

  since we use $(1 + \text{depth of } k)$ comparisons to search for key k.

Optimal static binary search trees

- Can we find the optimal static order for a binary search tree?

k_i	A	B	C	D	E
$P(k_i)$	$\frac{5}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{2}{26}$



$1 \cdot \frac{10}{26} + 2 \cdot \frac{8}{26} + 2 \cdot \frac{2}{26} + 3 \cdot \frac{5}{26} + 3 \cdot \frac{1}{26} = \frac{48}{26}$

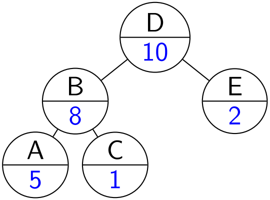
- Access-cost is now $\sum_k P(k) \cdot (1 + \text{depth of } k)$

  since we use $(1 + \text{depth of } k)$ comparisons to search for key k.

Optimal static binary search trees

- Can we find the optimal static order for a binary search tree?

k_i	A	B	C	D	E
$P(k_i)$	$\frac{5}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{2}{26}$



$1 \cdot \frac{10}{26} + 2 \cdot \frac{8}{26} + 2 \cdot \frac{2}{26} + 3 \cdot \frac{5}{26} + 3 \cdot \frac{1}{26} = \frac{48}{26}$

- Access-cost is now $\sum_k P(k) \cdot (1 + \text{depth of } k)$

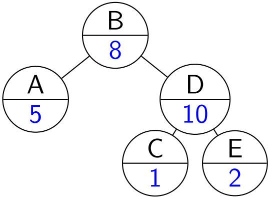
  since we use $(1 + \text{depth of } k)$ comparisons to search for key k.

- Natural greedy-algorithm:
  - Put item with highest access-probability at the root.
  - Split keys into left/right as dictated by the order-property.
  - Recurse in the subtree.

Optimal static binary search trees

The greedy-algorithm does *not* give the optimum!

k_i	A	B	C	D	E
$P(k_i)$	$\frac{5}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{2}{26}$

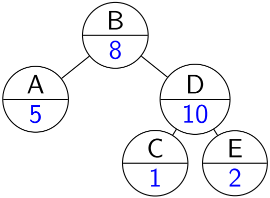


$$1 \cdot \frac{8}{26} + 2 \cdot \frac{5}{26} + 2 \cdot \frac{10}{26} + 3 \cdot \frac{1}{26} + 3 \cdot \frac{2}{26} = \frac{47}{26}$$

Optimal static binary search trees

The greedy-algorithm does *not* give the optimum!

k_i	A	B	C	D	E
$P(k_i)$	$\frac{5}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{2}{26}$



$1 \cdot \frac{8}{26} + 2 \cdot \frac{5}{26} + 2 \cdot \frac{10}{26} + 3 \cdot \frac{1}{26} + 3 \cdot \frac{2}{26} = \frac{47}{26}$

- To find the optimum, use "dynamic programming":
  - Effectively try *all* possible binary search trees
  - This would take exponential time if done in a straightfoward way.
  - Key idea: We can store and re-use solutions of subproblems to achieve polynomial run-time
- Many more details in cs341 (though not perhaps for this problem)

Outline

1. Even more Dictionary implementations
   - Expected height of a BST
   - Treaps
   - Optimal static binary search trees
   - MTF-heuristic in a BST
   - Splay Trees

MTF-heuristic for binary search trees

What does 'move-to-front' mean in a binary search tree?

- Front = the place that is easiest to access
- In a binary search tree, that's the root.

$\Rightarrow$ After every access, bring item to the root of BST

MTF-heuristic for binary search trees

What does ‘move-to-front’ mean in a binary search tree?

- Front = the place that is easiest to access
- In a binary search tree, that’s the root.

⇒ After every access, bring item to the root of BST

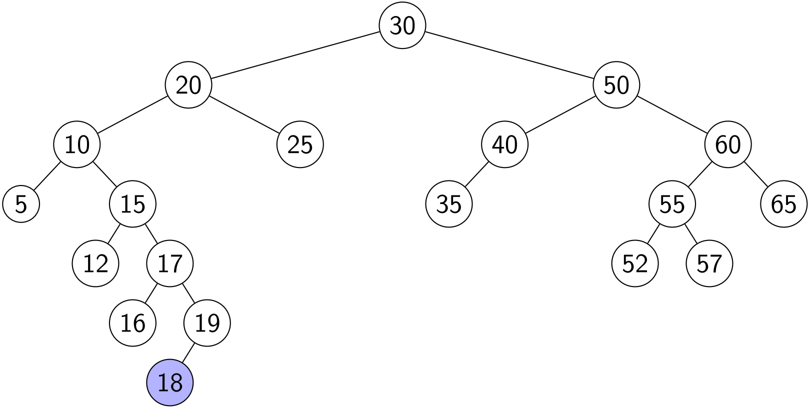
- But: order-property must be maintained!

⇒ Use *rotations*!

(This should remind you of treaps.)

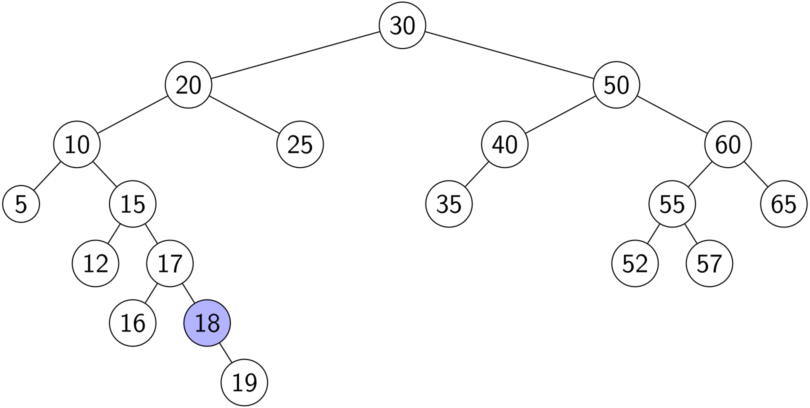
MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



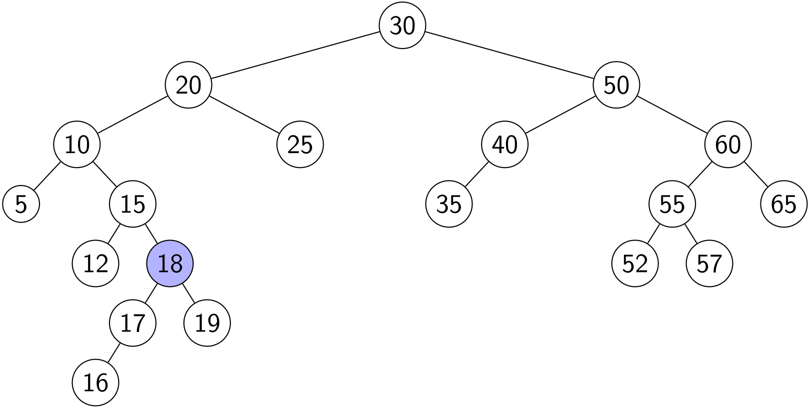
MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



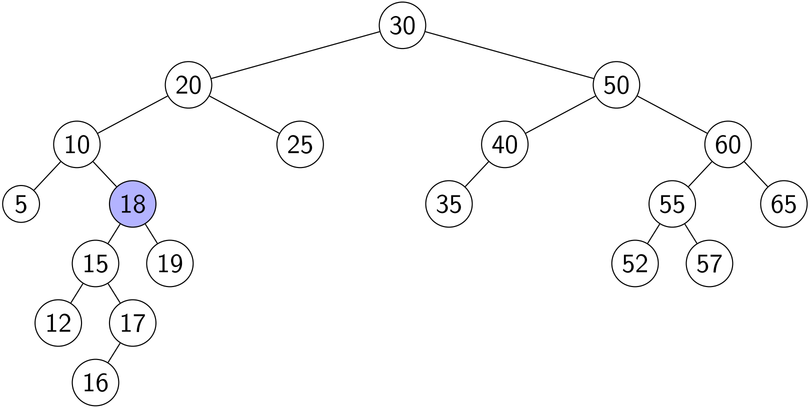
MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



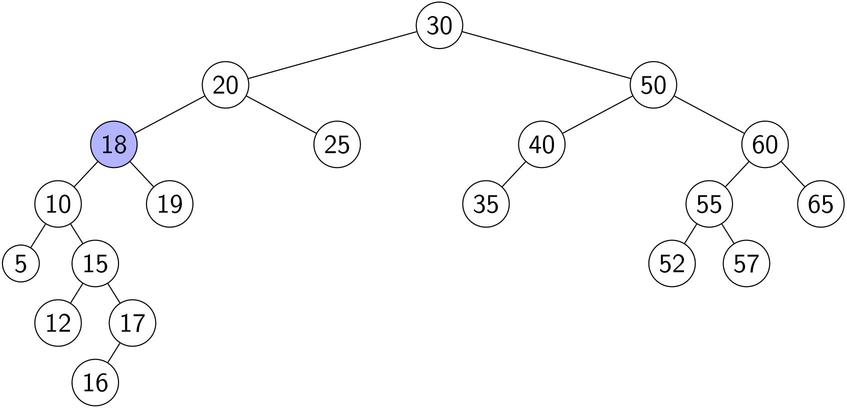
MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



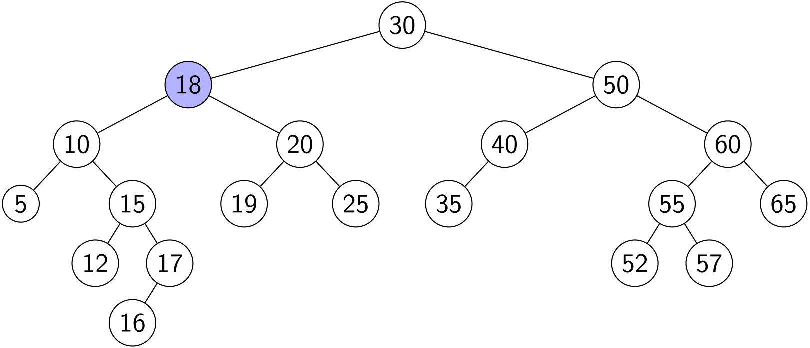
MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



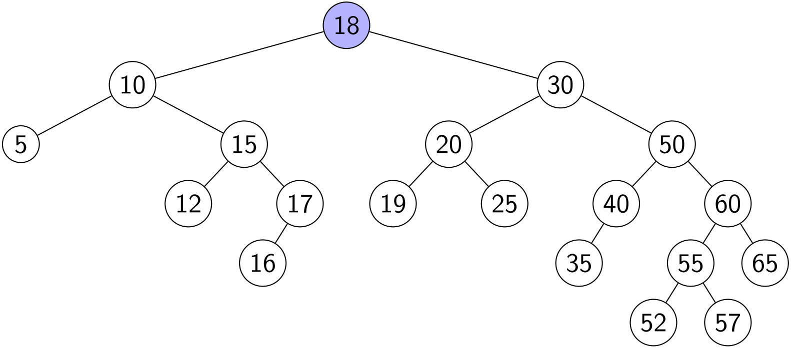
MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



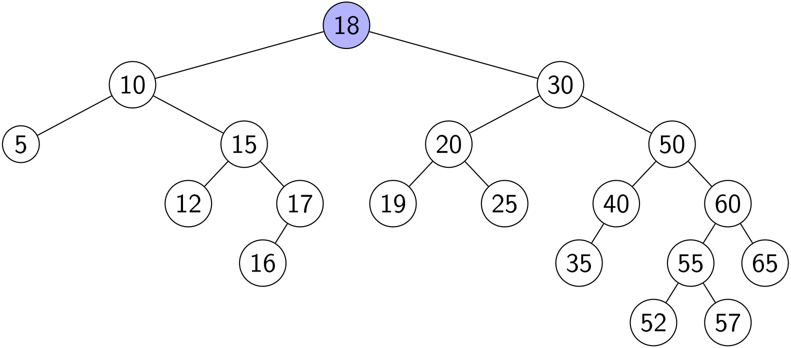
MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



MTF-heuristic for binary search trees

Example: *BST-MTF::search*(18)



This should work well, but we can do better by moving two level at a time.

Outline

1. Even more Dictionary implementations
   - Expected height of a BST
   - Treaps
   - Optimal static binary search trees
   - MTF-heuristic in a BST
   - Splay Trees

Splay trees

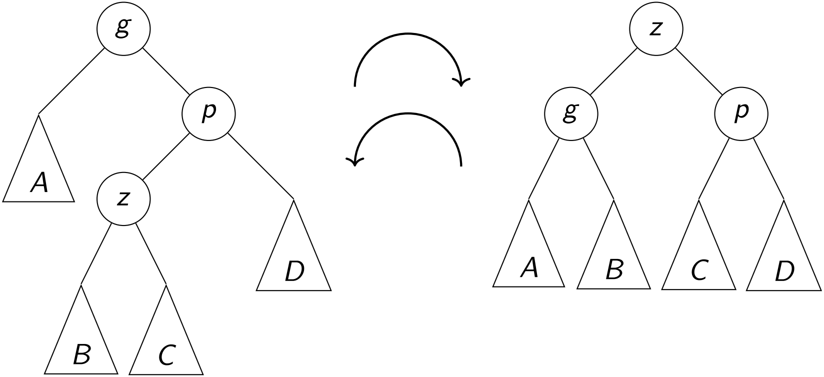
Splay tree overview:

- Binary search tree
- *No* extra information (such as height, balance, size) needed at nodes
- After search/insert, bring accessed node to the root with rotations
- Move node up two layers at a time (except when near root)
  - Use **zig-zig-rotation** or **zig-zag-rotation** to move up two levels.

Goal: This has amortized run-time $O(\log n)$.

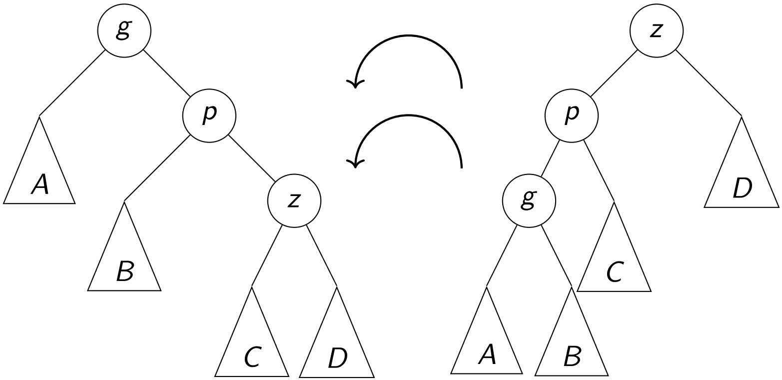
Zig-zag Rotation = Double Rotation

- Let z be the node that we want to move up.
- Let p and g be its parent and grandparent.
- If they are in zig-zag formation, apply a double-rotation.



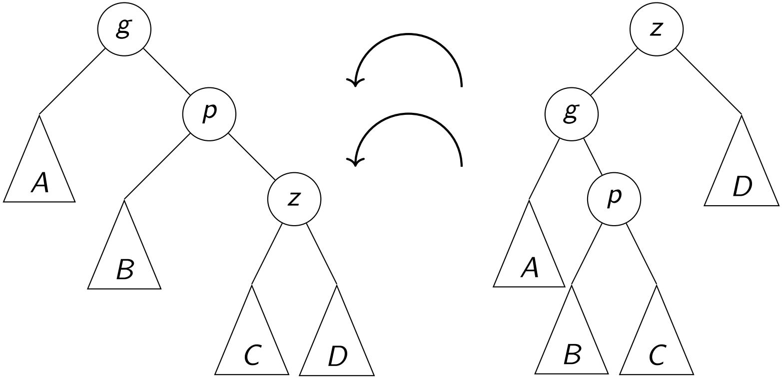
Zig-zig Rotation

- If they are in zig-zig formation, apply a new kind of rotation.



First, a left rotation at g. Second, a left rotation at p.

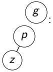
Compare to doing two single rotations

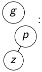


- Both operations bring z two levels higher.
- But using the zig-zig rotation allows to do amortized analysis.

Splay Tree Operations

SplayTree::insert(k, v)

$z \leftarrow$ *BST::insert*(k, v)
while (z is not the root)
  $p \leftarrow z.parent$
  if (z is the left child of p)
    if (p is the root) *rotate-right*(p)
    else $g \leftarrow p.parent$
    case 

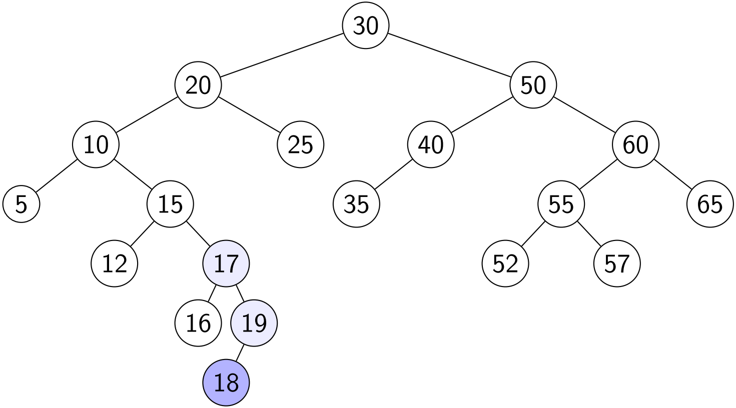
: // Zig-zig rotation
      rotate-right(g)
      rotate-right(p)
    

: // Zig-zag rotation
      rotate-right(p)
      rotate-left(g)
  else ... // symmetric case, z is right child

search and *delete* use corresponding BST-method
Then rotate the lowest visited node up.

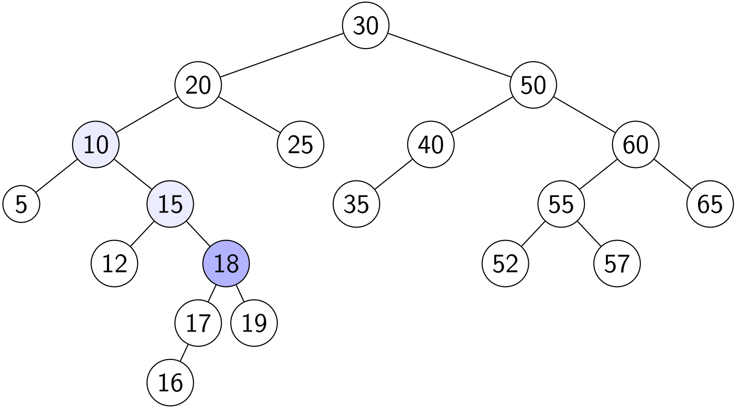
Splay Tree Insert

Example: *SplayTree::search*(18)



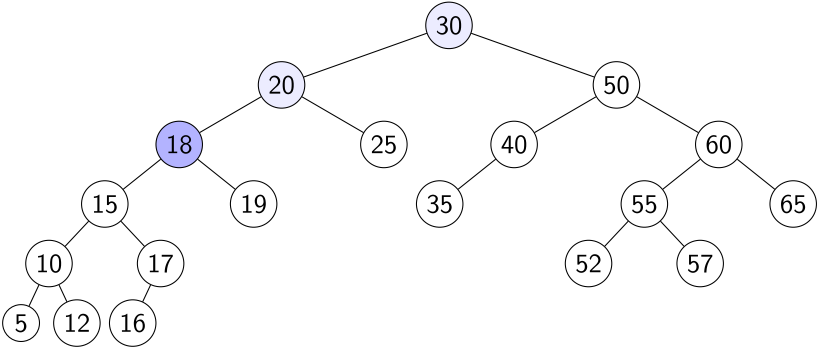
Splay Tree Insert

Example: *SplayTree::search*(18)



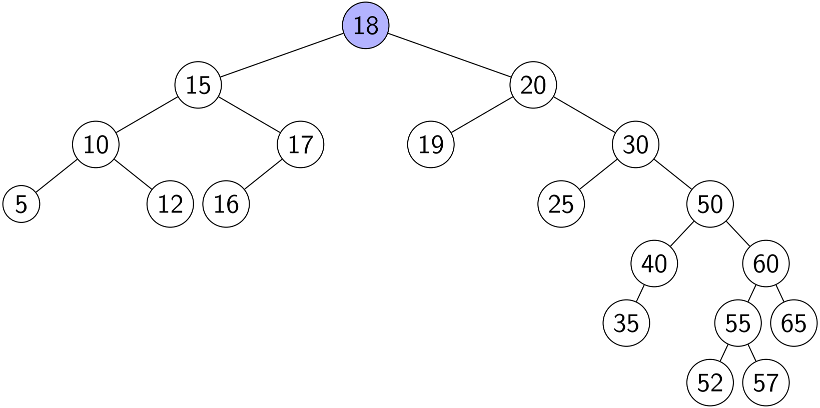
Splay Tree Insert

Example: *SplayTree::search*(18)



Splay Tree Insert

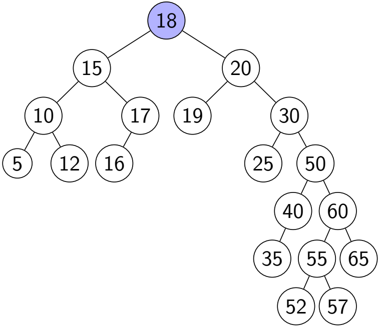
Example: *SplayTree::search*(18)



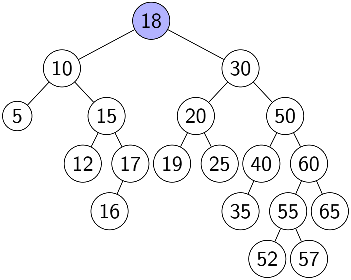
Zig-zig rotations vs. single rotations

Compare the resulting trees:

With zig-zig rotations:



With single rotations:

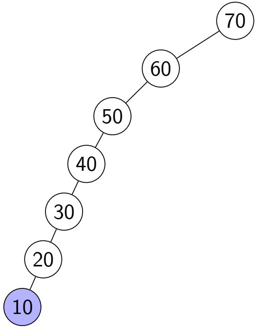


This is *not* more balanced, why do we apply zig-zig-rotations?

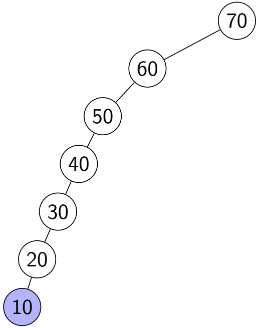
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



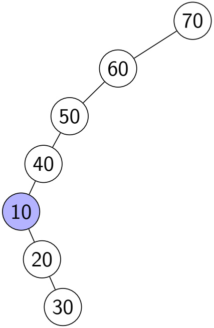
With single rotations:



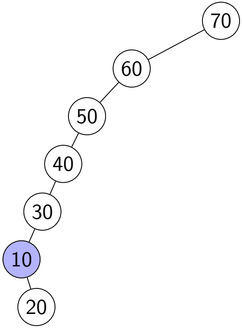
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



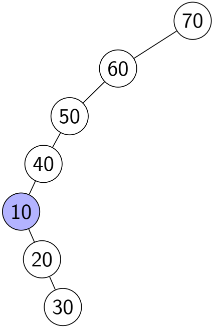
With single rotations:



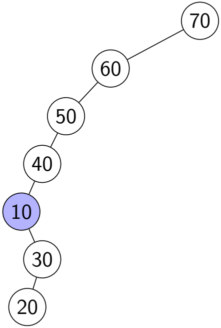
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



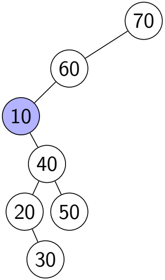
With single rotations:



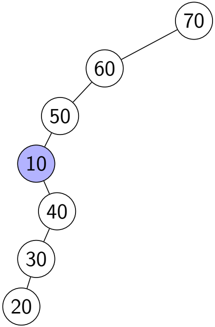
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



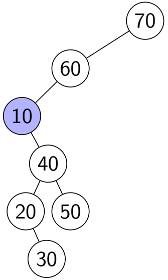
With single rotations:



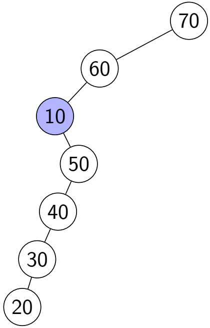
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



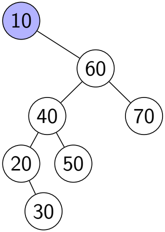
With single rotations:



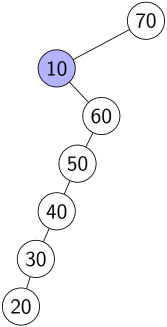
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



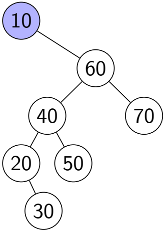
With single rotations:



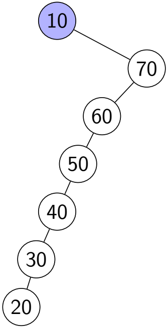
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



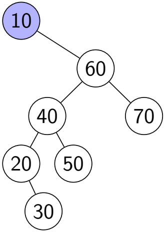
With single rotations:



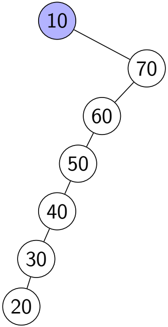
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

With zig-zig rotations:



With single rotations:



Splay tree intuition:

- For any node on search-path, the depth (roughly) halves
- For all nodes, the depth increases by at most 2

Splay tree summary

Theorem: In a splay tree, all operations take $O(\log n)$ amortized time.

(The formal proof does not follow the intuition and uses a potential function.)

In summary:

- Needs *no* extra information (such as height or size) needed at nodes
- Our pseudo-code assumed parent-references; this can be avoided by temporarily storing search-path.