

CS 240E – Data Structures and Data Management (Enriched)

Module 5: Other Dictionary Implementations

Tom Iagovet

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version 2026-06-09 01:33

Outline

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - Biased Search Requests
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - MTF-heuristic in a BST
 - Splay Trees

Outline

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - Biased Search Requests
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - MTF-heuristic in a BST
 - Splay Trees

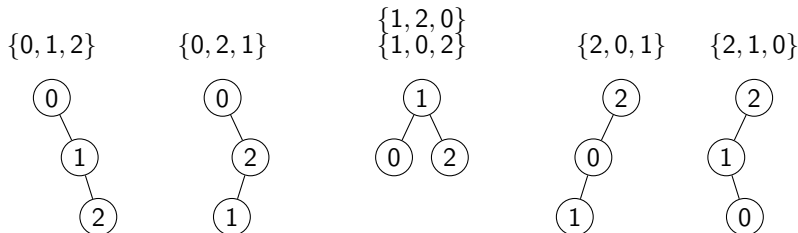
Expected height of BSTs

Goal: If the keys are nicely distributed, then using a BST without any extras will do.

Expected height of BSTs

Goal: If the keys are nicely distributed, then using a BST without any extras will do.

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



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

Proof:

Outline

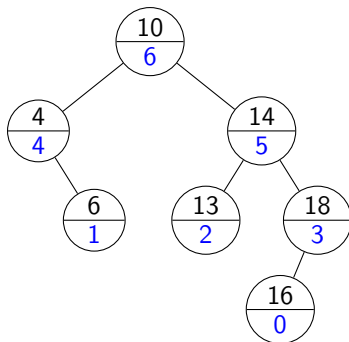
- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps**
 - Skip Lists
 - Biased Search Requests
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - MTF-heuristic in a BST
 - Splay Trees

Treaps

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

Idea: Use binary search tree, but associate a priority with each key.

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



We call this a **treap** (= tree + heap).

Treaps

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

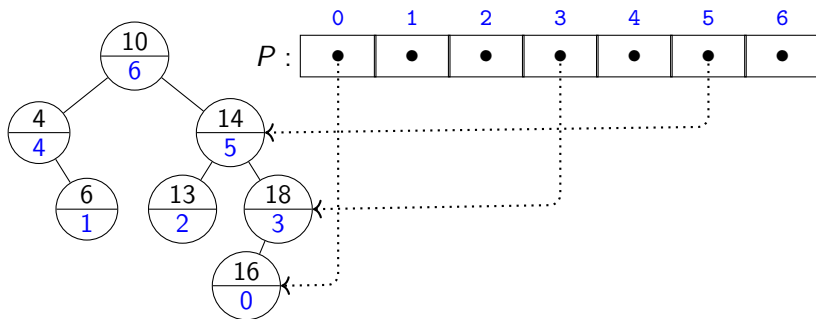
- Root-item has priority $n - 1$.
- This is picked randomly, so proof for expected height of BST applies.

Treaps

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

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

For updates, we need an array P where $P[i]$ stores node with priority i .



Treaps: Insertion

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

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

Treaps: Insertion

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

- We need a random permutation of $\{0, \dots, n-1\}$
 - ▶ Currently we had a random permutation of $\{0, \dots, n-2\}$.
- Recall: *shuffle* creates a random permutation:

shuffle(A)

A: array of size n stores $\langle 0, \dots, n-1 \rangle$

1. **for** $i \leftarrow 1$ to $n-1$ **do**
2. $\text{swap}(A[i], A[\text{random}(i+1)])$

Treaps: Insertion

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

- We need a random permutation of $\{0, \dots, n-1\}$
 - ▶ Currently we had a random permutation of $\{0, \dots, n-2\}$.
- Recall: *shuffle* creates a random permutation:

shuffle(A)

A : array of size n stores $\langle 0, \dots, n-1 \rangle$

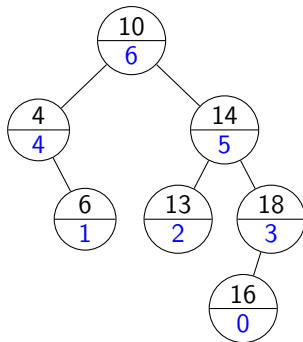
1. **for** $i \leftarrow 1$ to $n-1$ **do**
2. $\text{swap}(A[i], A[\text{random}(i+1)])$

We imitate *shuffle*'s behaviour by *randomly* picking priority for new item.

- $p \leftarrow \text{random}(n)$ is in $\{0, \dots, n-1\}$
- The item that previously had priority p now gets priority $n-1$.
- If this violates the heap-property, then rotate to fix it.

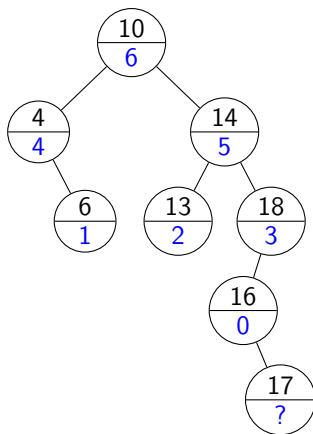
Treaps insertion: Example

Example: *treap::insert*(17)



Treaps insertion: Example

Example: *treap::insert*(17)

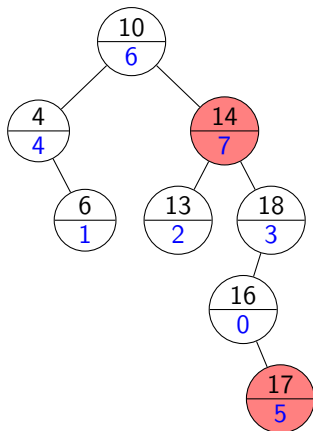


Treaps insertion: Example

Example: *treap::insert*(17)

Randomly pick priority 5 $\in \{0, \dots, 7\}$, and change priority of $P[5]$ to 7.

These priorities violate order-property.

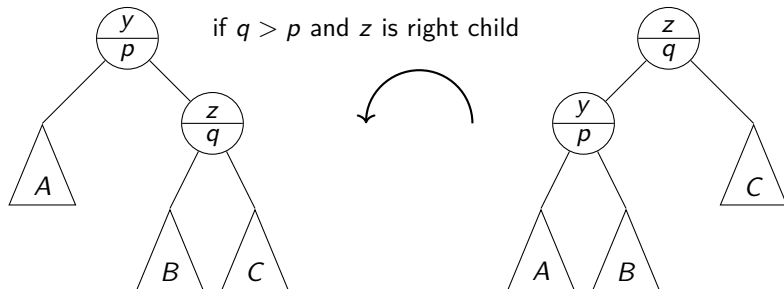


Fixing incorrect priorities with rotations

- In binary heaps, we fixed increased priorities via *fix-up* (swaps)
- Does not work here! We must also maintain the BST-order.
- Idea: Rotations maintain the BST-order and fix priorities.

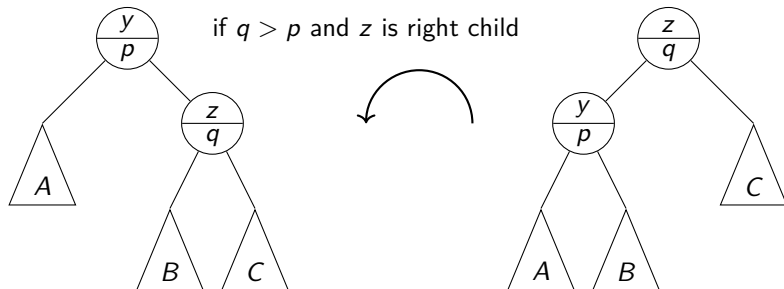
Fixing incorrect priorities with rotations

- In binary heaps, we fixed increased priorities via *fix-up* (swaps)
- Does not work here! We must also maintain the BST-order.
- Idea: Rotations maintain the BST-order and fix priorities.



Fixing incorrect priorities with rotations

- In binary heaps, we fixed increased priorities via *fix-up* (swaps)
- Does not work here! We must also maintain the BST-order.
- Idea: Rotations maintain the BST-order and fix priorities.

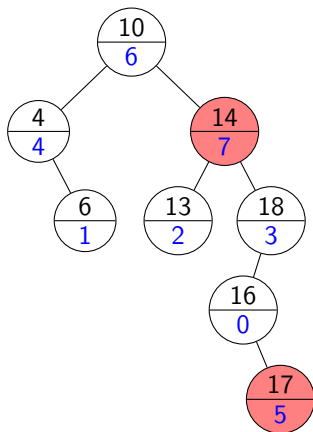


treap::fix-up-with-rotations(z)

1. **while** ($y \leftarrow z.\text{parent}$ is not NULL and $z.\text{priority} > y.\text{priority}$) **do**
2. **if** z is the left child of y **do** *rotate-right*(y)
3. **else** *rotate-left*(y)

Treaps insertions: Example

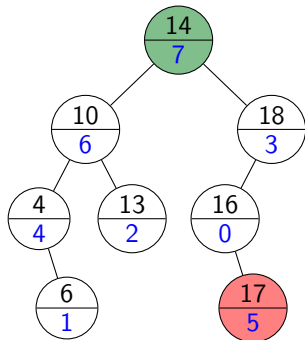
Example: *treap::insert*(17)



Treaps insertions: Example

Example: *treap::insert*(17)

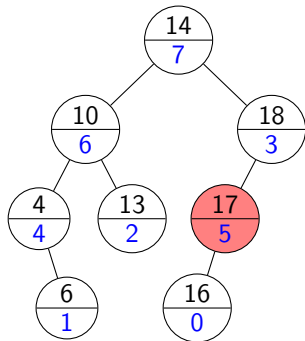
Fix *upper* violation first (why?)



Treaps insertions: Example

Example: *treap::insert*(17)

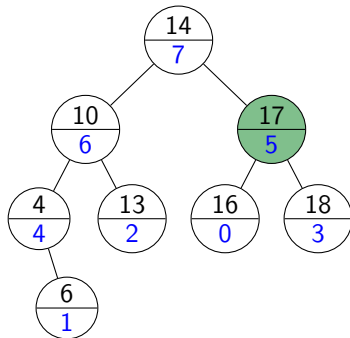
Fix *upper* violation first (why?)



Treaps insertions: Example

Example: *treap::insert*(17)

Fix *upper* violation first (why?)



Treaps insertion: Code

Recall: $P[i]$ = node with priority i .

```
treap::insert( $k, v$ )  
1.  $n \leftarrow P.size$            // current size  
2.  $z \leftarrow BST::insert(k, v); n++$   
3.  $p \leftarrow random(n)$   
4. if  $p < n - 1$  do  
5.      $z' \leftarrow P[p], z'.priority \leftarrow n - 1, P[n - 1] \leftarrow z'$   
6.     fix-up-with-rotations( $z'$ )  
7.  $z.priority \leftarrow p; P[p] \leftarrow z$   
8. fix-up-with-rotations( $z$ )
```

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)

But not particularly useful in practice

(except when priorities have meaning $\rightsquigarrow$ later)

- Large space overhead (parent-pointers, priorities, P)
- There are ways to avoid some of the space overhead, but in general randomized binary search trees are rarely used.
- We will now see a randomization that works better (but is not a binary search tree)

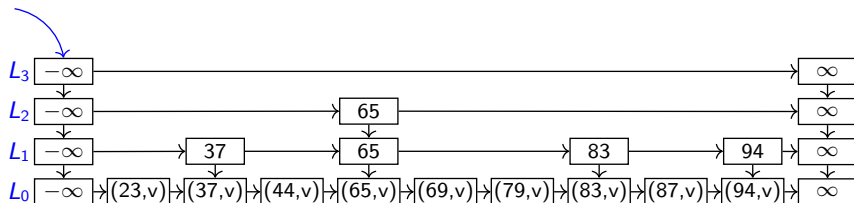
Outline

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - **Skip Lists**
 - Biased Search Requests
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - MTF-heuristic in a BST
 - Splay Trees

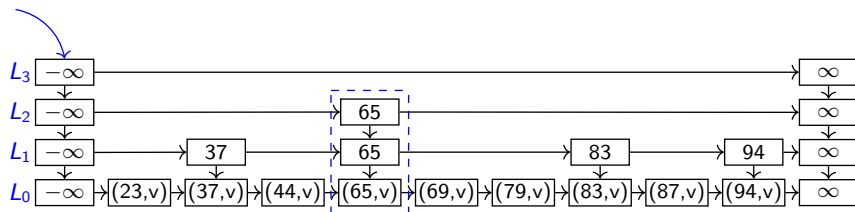
Skip lists

A hierarchy of ordered linked lists (*levels*) $L_0, L_1, \dots, L_h$:

- Each list L_i contains the special keys $-\infty$ and $+\infty$ (**sentinels**)
- List L_0 contains the KVPs of S in non-decreasing order.
(The other lists store only keys and references.)
- Each list is a subsequence of the previous one, i.e.,
 $L_0 \supseteq L_1 \supseteq \dots \supseteq L_h$
- List L_h contains only the sentinels, all other lists contain at least one non-sentinel.



Skip lists



A few more definitions:

- **node** = entry in one list vs. **KVP** = one non-sentinel entry in L_0
- There are (usually) more **nodes** than **KVPs**
Here $\#$ (non-sentinel) nodes = 14 vs. $n \leftarrow \#$ KVPs = 9.
- **root** = topmost left sentinel is the only field of the skip list.
- Each node p has references $p.after$ and $p.below$
- Each key k belongs to a **tower** of nodes
 - ▶ Height of tower of k : maximal index i such that $k \in L_i$
 - ▶ Height of skip list: maximal index h such that L_h exists

Skip lists: Search

For each list, find **predecessor** (node before where k would be).
This will also be useful for *insert/delete*.

get-predecessors (k)

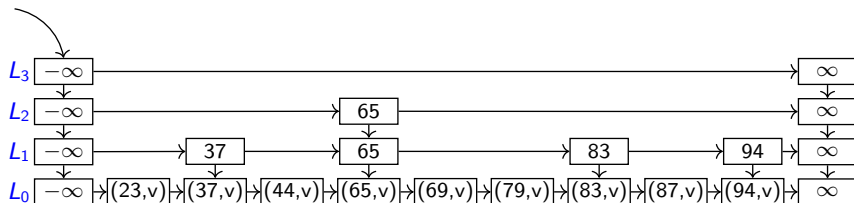
1. $p \leftarrow \text{root}$
2. $P \leftarrow$ stack of nodes, initially containing p
3. **while** $p.\text{below} \neq \text{NULL}$ **do**
4. $p \leftarrow p.\text{below}$
5. **while** $p.\text{after.key} < k$ **do** $p \leftarrow p.\text{after}$
6. $P.\text{push}(p)$
7. **return** P

skipList::search (k)

1. $P \leftarrow \text{get-predecessors}(k)$
2. $p_0 \leftarrow P.\text{top}()$ // predecessor of k in L_0
3. **if** $p_0.\text{after.key} = k$ **return** KVP at $p_0.\text{after}$
4. **else return** "not found, but would be after p_0 "

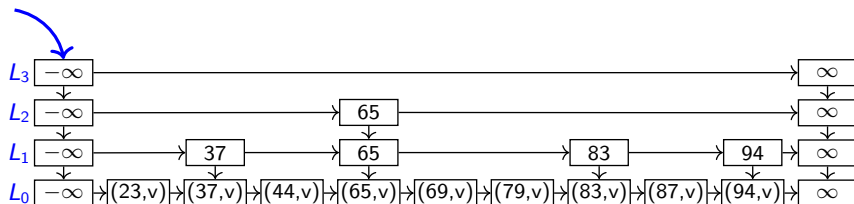
Skip list search: Example

Example: *search*(87)



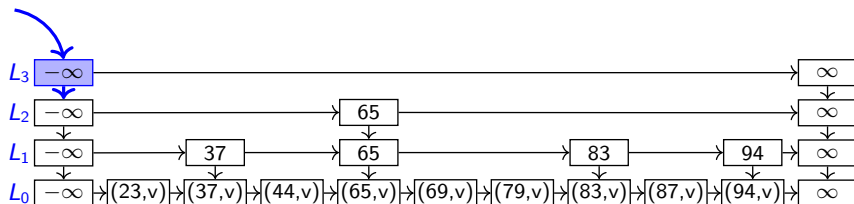
Skip list search: Example

Example: *search*(87)



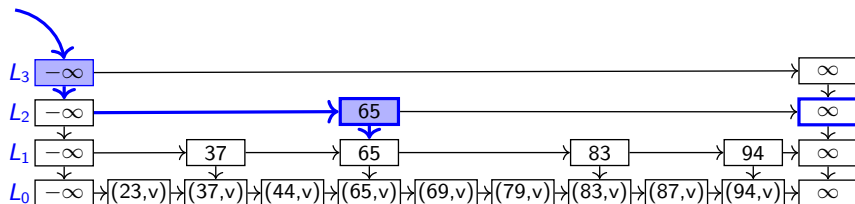
Skip list search: Example

Example: *search*(87)



Skip list search: Example

Example: *search*(87)



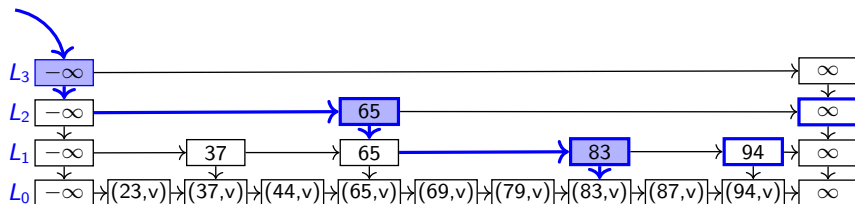
key compared with k

added to P

→ path taken by p

Skip list search: Example

Example: *search*(87)



key compared with k

 added to P

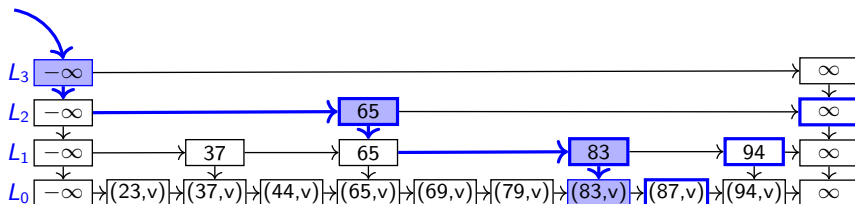
→ path taken by p

Final stack returned:

$(83, v)$
83
65
$-\infty$

Skip list search: Example

Example: *search*(87)



key compared with k

added to P

→ path taken by p

Final stack returned:

$(83,v)$
83
65
$-\infty$

Skip list: Deletion

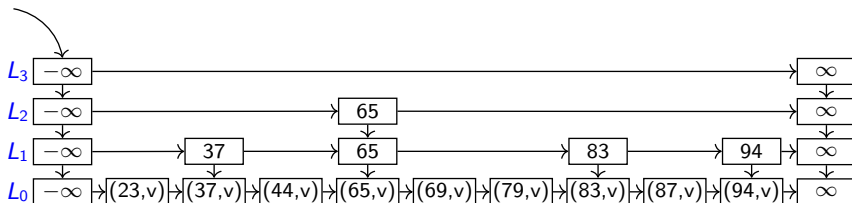
It is easy to remove a key since we can find all predecessors.
Then eliminate lists if there are multiple ones with only sentinels.

```
skipList::delete(k)
1.  $P \leftarrow \text{get-predecessors}(k)$ 
2. while  $P$  is non-empty
3.      $p \leftarrow P.\text{pop}()$  // predecessor of  $k$  in some list
4.     if  $p.\text{after.key} = k$ 
5.          $p.\text{after} \leftarrow p.\text{after.after}$ 
6.     else break // no more copies of  $k$ 

7.  $p \leftarrow$  left sentinel of the root-list
8. while  $p.\text{below.after}$  is the  $\infty$ -sentinel
    // top two lists have only sentinels, remove one
9.      $p.\text{below} \leftarrow p.\text{below.below}$ 
10.     $p.\text{after.below} \leftarrow p.\text{after.below.below}$ 
```

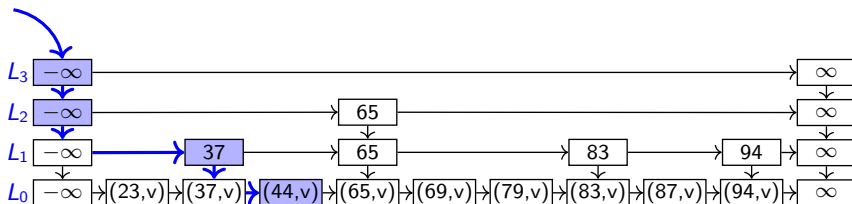
Skip list deletion: Example

Example: *skipList::delete*(65)



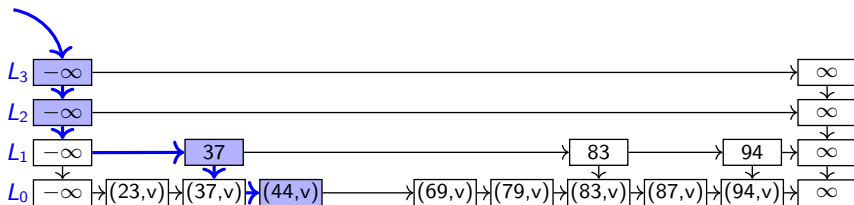
Skip list deletion: Example

Example: *skipList::delete*(65)
get-predecessors(65)



Skip list deletion: Example

Example: *skipList::delete*(65)
get-predecessors(65)

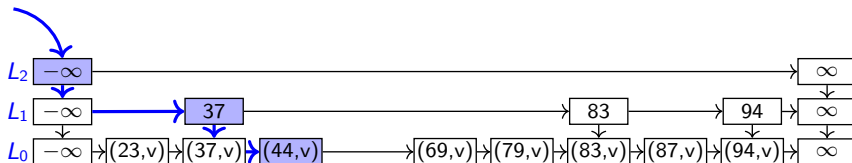


Skip list deletion: Example

Example: *skipList::delete*(65)

get-predecessors(65)

Height decrease



Skip lists: Insertion

skipList::insert(k, v)

- There is no choice as to where to put the tower of k .
- Only choice: how tall should we make the tower of k ?
 - ▶ Choose *randomly*! Repeatedly toss a coin until you get tails
 - ▶ Let i the number of times the coin came up heads
 - ▶ We want key k to be in lists $L_0, \dots, L_i$, so $i \rightarrow$ *height* of tower of k

$$\mathbb{P}(\text{tower of key } k \text{ has height } \geq i) = \left(\frac{1}{2}\right)^i$$

Skip lists: Insertion

skipList::insert(k, v)

- There is no choice as to where to put the tower of k .
- Only choice: how tall should we make the tower of k ?
 - ▶ Choose *randomly*! Repeatedly toss a coin until you get tails
 - ▶ Let i the number of times the coin came up heads
 - ▶ We want key k to be in lists $L_0, \dots, L_i$, so $i \rightarrow$ *height* of tower of k

$$\mathbb{P}(\text{tower of key } k \text{ has height } \geq i) = \left(\frac{1}{2}\right)^i$$

- Before we can insert, we must check that these lists exist.
 - ▶ Add sentinel-only lists, if needed, until height h satisfies $h > i$.

Skip lists: Insertion

skipList::insert(k, v)

- There is no choice as to where to put the tower of k .
- Only choice: how tall should we make the tower of k ?
 - ▶ Choose *randomly*! Repeatedly toss a coin until you get tails
 - ▶ Let i the number of times the coin came up heads
 - ▶ We want key k to be in lists $L_0, \dots, L_i$, so $i \rightarrow$ *height* of tower of k

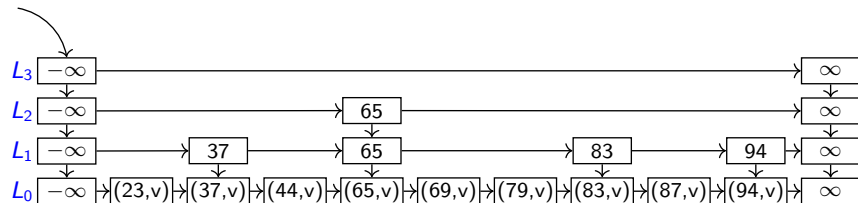
$$\mathbb{P}(\text{tower of key } k \text{ has height } \geq i) = \left(\frac{1}{2}\right)^i$$

- Before we can insert, we must check that these lists exist.
 - ▶ Add sentinel-only lists, if needed, until height h satisfies $h > i$.
- Then do the actual insertion.
 - ▶ Use *get-predecessors*(k) to get stack P .
 - ▶ The top i items of P are the predecessors $p_0, p_1, \dots, p_i$ of where k should be in each list $L_0, L_1, \dots, L_i$
 - ▶ Insert (k, v) after p_0 in L_0 , and k after p_j in L_j for $1 \leq j \leq i$

Skip list insertion: Example

Example: *skipList::insert*(52, v)

Coin tosses: H,T $\Rightarrow i = 1$

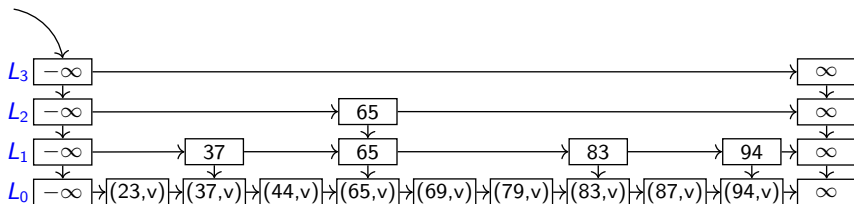


Skip list insertion: Example

Example: *skipList::insert*(52, v)

Coin tosses: H,T $\Rightarrow i = 1$

Have $h = 3 > i \Rightarrow$ no need to add lists



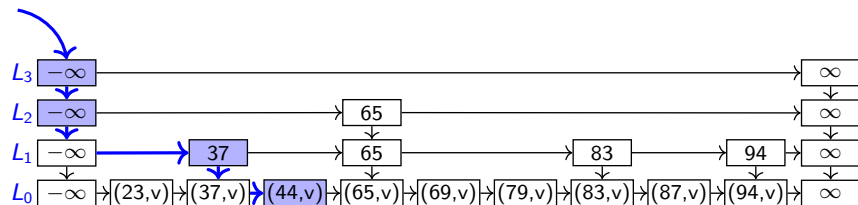
Skip list insertion: Example

Example: *skipList::insert*(52, v)

Coin tosses: H,T $\Rightarrow i = 1$

Have $h = 3 > i \Rightarrow$ no need to add lists

get-predecessors(52)



Skip list insertion: Example

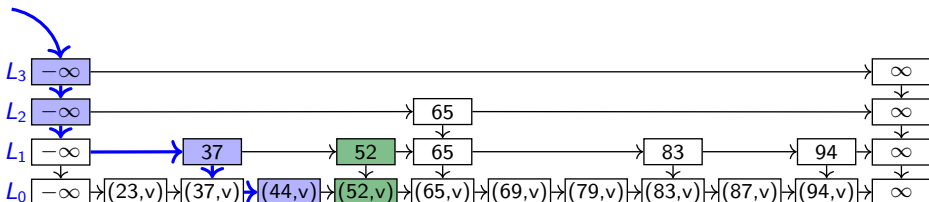
Example: *skipList::insert*(52, v)

Coin tosses: H,T $\Rightarrow i = 1$

Have $h = 3 > i \Rightarrow$ no need to add lists

get-predecessors(52)

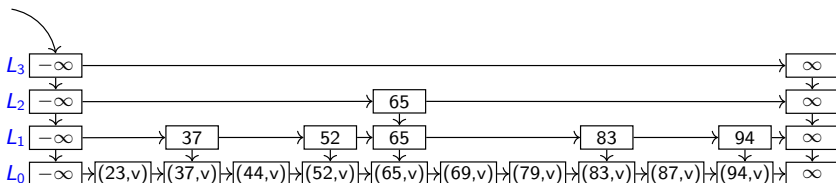
Insert 52 in lists $L_0, \dots, L_i$



Skip list insert: Example 2

Example: *skipList::insert*(100, v)

Coin tosses: H,H,H,T $\Rightarrow i = 3$

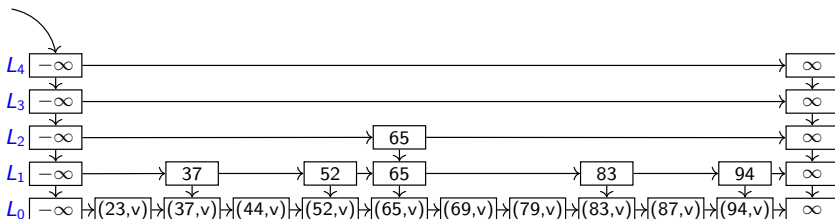


Skip list insert: Example 2

Example: *skipList::insert*(100, v)

Coin tosses: H,H,H,T $\Rightarrow i = 3$

Height increase



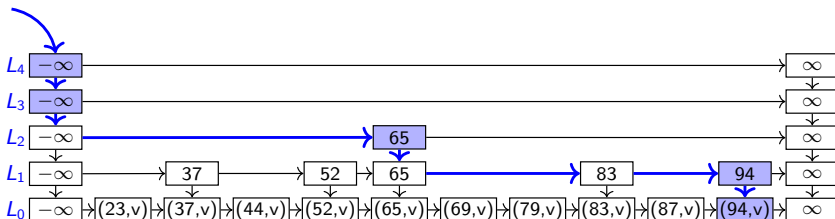
Skip list insert: Example 2

Example: *skipList::insert*(100, *v*)

Coin tosses: H,H,H,T $\Rightarrow i = 3$

Height increase

get-predecessors(100)



Skip list insert: Example 2

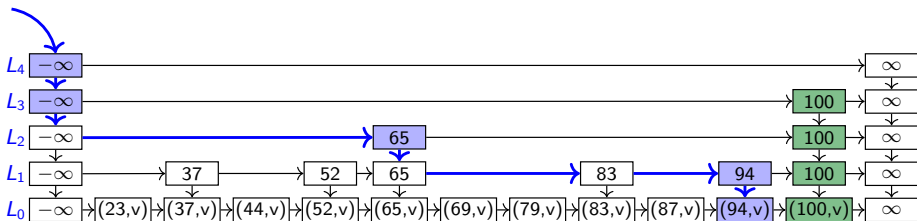
Example: *skipList::insert*(100, v)

Coin tosses: H,H,H,T $\Rightarrow i = 3$

Height increase

get-predecessors(100)

Insert 100 in lists $L_0, \dots, L_i$



Skip lists: Space

Claim: The expected number of non-sentinels is $O(n)$.

- Set $X_k =$ tower height of key k . Recall $\mathbb{P}(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
- Define $|L_i| = \# \text{non-sentinels in } L_i$. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.

(**Indicator-variable** $\chi_Z = \begin{cases} 1 & \text{if } Z \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ has $E[\chi_Z] = P(Z \text{ is true})$.)

Skip lists: Space

Claim: The expected number of non-sentinels is $O(n)$.

- Set $X_k =$ tower height of key k . Recall $\mathbb{P}(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
- Define $|L_i| = \# \text{non-sentinels in } L_i$. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.

(**Indicator-variable** $\chi_Z = \begin{cases} 1 & \text{if } Z \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ has $E[\chi_Z] = P(Z \text{ is true})$.)

- What is $E[|L_i|]$?

Skip lists: Space

Claim: The expected number of non-sentinels is $O(n)$.

- Set $X_k =$ tower height of key k . Recall $\mathbb{P}(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
- Define $|L_i| = \# \text{non-sentinels in } L_i$. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.

(**Indicator-variable** $\chi_Z = \begin{cases} 1 & \text{if } Z \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ has $E[\chi_Z] = P(Z \text{ is true})$.)

- What is $E[|L_i|]$?
- What is $E[\# \text{non-sentinels}] = \sum_{i=0}^h E[|L_i|]$?

Skip lists: Height

Claim: The expected height h is $O(\log n)$.

- Observe: $h = \sum_{i \geq 0} I_i$ where $I_i = \chi_{(|L_i| \geq 1)} = \begin{cases} 1 & \text{if } L_i \text{ has non-sentinels} \\ 0 & \text{otherwise} \end{cases}$
- Observe: $I_i \leq \min\{1, |L_i|\}$, so $E[I_i] \leq \min\{1, E[|L_i|]\}$.
- What is $E[h] = \sum_{i \geq 0} E[I_i]$?

Skip lists: Height

Claim: The expected height h is $O(\log n)$.

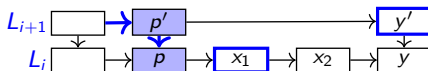
- Observe: $h = \sum_{i \geq 0} I_i$ where $I_i = \chi_{(|L_i| \geq 1)} = \begin{cases} 1 & \text{if } L_i \text{ has non-sentinels} \\ 0 & \text{otherwise} \end{cases}$
- Observe: $I_i \leq \min\{1, |L_i|\}$, so $E[I_i] \leq \min\{1, E[|L_i|]\}$.
- What is $E[h] = \sum_{i \geq 0} E[I_i]$?

Therefore the expected space is $O(n)$.

Skip lists: Getting predecessors

Claim: *skipList::get-predecessors* has $O(\log n)$ expected run-time.

- How often do we **drop down** (execute $p \leftarrow p.\text{below}$)? *height*.
- How often do we **step forward** (execute $p \leftarrow p.\text{after}$)?
 - ▶ We immediately drop down in L_h , so consider L_i for $i < h$.

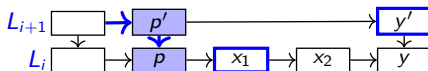


- ▶ Key insight: x_1 did **not** exist in L_{i+1} (else would go forward there)
- ▶ So the tower of x_1 **exactly** ended with L_i . What is the probability?

Skip lists: Getting predecessors

Claim: *skipList::get-predecessors* has $O(\log n)$ expected run-time.

- How often do we **drop down** (execute $p \leftarrow p.\text{below}$)? *height*.
- How often do we **step forward** (execute $p \leftarrow p.\text{after}$)?
 - ▶ We immediately drop down in L_h , so consider L_i for $i < h$.

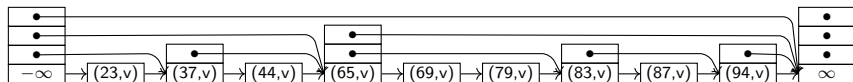


- ▶ Key insight: x_1 did **not** exist in L_{i+1} (else would go forward there)
- ▶ So the tower of x_1 **exactly** ended with L_i . What is the probability?

- So *search*, *insert*, *delete* have $O(\log n)$ expected run-time

Skip lists: Summary

- $O(n)$ expected space, all operations take $O(\log n)$ expected time.
- Lists make it easy to implement. We can also easily add more operations (e.g. *successor*, *merge*,...)
- As described they are no better than randomized binary search trees.
- But there are numerous improvements on the space:
 - ▶ Can save links (hence space) by implementing towers as array.



- ▶ Biased coin-flips to determine tower-heights give smaller expected space
- ▶ With both ideas, expected space is $< 2n$ (less than for a BST).

Outline

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - **Biased Search Requests**
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - MTF-heuristic in a BST
 - Splay Trees

Biased search request

Thus far: All keys are assumed to be equal.

- Any key k is equally often the key used when searching.

Biased search request

Thus far: All keys are assumed to be equal.

- Any key k is equally often the key used when searching.

In reality: Some keys are more frequently accessed than others
(**access:** insertion or successful search)

- 80/20 rule: 80% of outcomes result from 20% of causes.
- Rule of **temporal locality**: A recently accessed item is likely to be accessed soon again.
- How can we handle such **biased search requests**?

Biased search request

Thus far: All keys are assumed to be equal.

- Any key k is equally often the key used when searching.

In reality: Some keys are more frequently accessed than others
(**access:** insertion or successful search)

- 80/20 rule: 80% of outcomes result from 20% of causes.
- Rule of **temporal locality**: A recently accessed item is likely to be accessed soon again.
- How can we handle such **biased search requests**?

Intuition: Frequently accessed items should be near the **front** (place where we first search in the data structure).

- Two scenarios: Do we know the access distribution beforehand or not?

Outline

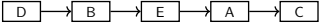
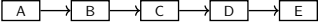
- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - Biased Search Requests
 - **Optimal Static Ordering**
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - MTF-heuristic in a BST
 - Splay Trees

Optimal static ordering

Scenario: We know access distribution, and want the best order of a list.

Example:

key	A	B	C	D	E
frequency of access	2	8	1	10	5

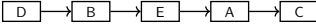
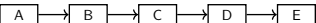
- Order  seems better than order 
- How do we formalize “better”?

Optimal static ordering

Scenario: We know access distribution, and want the best order of a list.

Example:

key	A	B	C	D	E
frequency of access	2	8	1	10	5

- Order  seems better than order 
- How do we formalize “better”?
- Define **access probability** of key $k = \frac{\text{frequency of } k}{\text{total number of accesses}}$
- Analyze (for any fixed order of keys) the

$$\text{expected access cost} = \sum_{i \geq 1} i \cdot (\text{access-probability of key at position } i)$$

(This is proportional to the (weighted) average-case time for *search*.)

Optimal static ordering

Example:

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

- Order $\boxed{A} \rightarrow \boxed{B} \rightarrow \boxed{C} \rightarrow \boxed{D} \rightarrow \boxed{E}$ has expected access cost $\frac{2}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{1}{26} \cdot 3 + \frac{10}{26} \cdot 4 + \frac{5}{26} \cdot 5 = \frac{86}{26} \approx 3.31$
- Order $\boxed{D} \rightarrow \boxed{B} \rightarrow \boxed{E} \rightarrow \boxed{A} \rightarrow \boxed{C}$ is better!
 $\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54$

Optimal static ordering

Example:

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

- Order $\boxed{A} \rightarrow \boxed{B} \rightarrow \boxed{C} \rightarrow \boxed{D} \rightarrow \boxed{E}$ has expected access cost $\frac{2}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{1}{26} \cdot 3 + \frac{10}{26} \cdot 4 + \frac{5}{26} \cdot 5 = \frac{86}{26} \approx 3.31$
- Order $\boxed{D} \rightarrow \boxed{B} \rightarrow \boxed{E} \rightarrow \boxed{A} \rightarrow \boxed{C}$ is better!
 $\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54$

Claim: Over all possible static orderings, we minimize the expected access cost by sorting by non-increasing access-probability.

Proof:

- Consider any other ordering. How can we improve its access cost?

Outline

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - Biased Search Requests
 - Optimal Static Ordering
 - **Optimal Static Binary Search Trees**
 - Dynamic Ordering: MTF
 - MTF-heuristic in a BST
 - Splay Trees

Optimal static BSTs

- Biased requests are very common in practice.
- We should re-visit all data structures in this light.
- Natural question:

*What is the best **binary search tree**
(presuming we know access probabilities)?*

Optimal static BSTs

- Biased requests are very common in practice.
- We should re-visit all data structures in this light.
- Natural question:

*What is the best **binary search tree**
(presuming we know access probabilities)?*

Can show:

- The “obvious” greedy-strategy
(put highest-frequency item at root, build subtrees recursively)
does **not** always give the best binary search tree.

Optimal static BSTs

- Biased requests are very common in practice.
- We should re-visit all data structures in this light.
- Natural question:

*What is the best **binary search tree**
(presuming we know access probabilities)?*

Can show:

- The “obvious” greedy-strategy
(put highest-frequency item at root, build subtrees recursively)
does **not** always give the best binary search tree.
- There **is** an algorithm to find the best binary search tree, but it uses an advanced algorithm design technique (“dynamic programming”) that will be seen in CS341.
- No details covered in CS240.

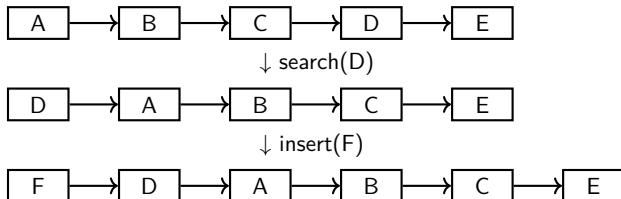
Outline

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - Biased Search Requests
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - **Dynamic Ordering: MTF**
 - MTF-heuristic in a BST
 - Splay Trees

Dynamic ordering: MTF

Scenario: We do *not know the access probabilities* ahead of time.

- **Idea:** modify the order dynamically, i.e., while we are accessing.
- Rule of thumb (**temporal locality**): A recently accessed item is likely to be used soon again.
- **Move-To-Front heuristic** (MTF): Upon a successful search, move the accessed item to the front of the list

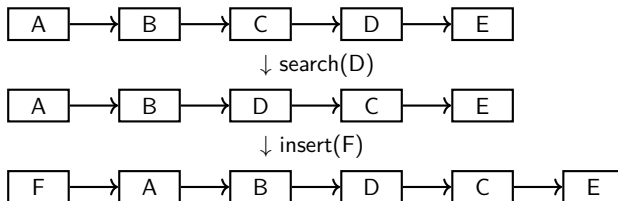


- We can also do MTF on an array, but should then insert and search from the *back* so that we have room to grow.

Dynamic ordering: Other ideas

There are other heuristics we could use:

- **Transpose heuristic:** Upon a successful search, swap the accessed item with the item immediately preceding it

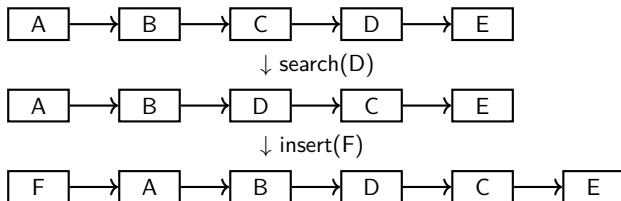


Here the changes are more gradual.

Dynamic ordering: Other ideas

There are other heuristics we could use:

- **Transpose heuristic:** Upon a successful search, swap the accessed item with the item immediately preceding it



Here the changes are more gradual.

- **Frequency-count heuristic:** Keep counters how often items were accessed, and sort in non-decreasing order.
Works well in practice, but requires auxiliary space.

Biased search requests: Summary

- We are unlikely to know the access-probabilities of items, so optimal static order is mostly of theoretical interest.
- For any dynamic reordering heuristic, some sequence will defeat it (have $\Theta(n)$ access-cost for each item).
- MTF and Frequency-count work well in practice.

Biased search requests: Summary

- We are unlikely to know the access-probabilities of items, so optimal static order is mostly of theoretical interest.
 - For any dynamic reordering heuristic, some sequence will defeat it (have $\Theta(n)$ access-cost for each item).
 - MTF and Frequency-count work well in practice.
 - For MTF, can also prove theoretical guarantees.
- $\left(\begin{array}{l} \text{▶ MTF is an } \textcolor{red}{\textit{online}} \text{ algorithm: Decide based on incomplete information.} \\ \text{▶ Compare it to the best } \textcolor{red}{\textit{offline}} \text{ algorithm (has complete information).} \\ \text{▶ Here, best offline-algorithm builds optimal static ordering.} \\ \text{▶ } \textcolor{violet}{\text{Can show:}} \text{ MTF is “2-competitive”}: \text{cost}(\text{MTF}) \leq 2 \cdot \text{cost}(\text{OPT}). \end{array} \right)$

Biased search requests: Summary

- We are unlikely to know the access-probabilities of items, so optimal static order is mostly of theoretical interest.
- For any dynamic reordering heuristic, some sequence will defeat it (have $\Theta(n)$ access-cost for each item).
- MTF and Frequency-count work well in practice.
- For MTF, can also prove theoretical guarantees.
 - ▶ MTF is an **online** algorithm: Decide based on incomplete information.
 - ▶ Compare it to the best **offline** algorithm (has complete information).
 - ▶ Here, best offline-algorithm builds optimal static ordering.
 - ▶ **Can show:** MTF is “2-competitive”: $\text{cost}(\text{MTF}) \leq 2 \cdot \text{cost}(\text{OPT})$.
- There is very little overhead for MTF and other strategies; they should be applied whenever unordered lists or arrays are used ($\rightarrow$ Hashing, text compression).

Outline

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - Biased Search Requests
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - **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.

⇒ 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

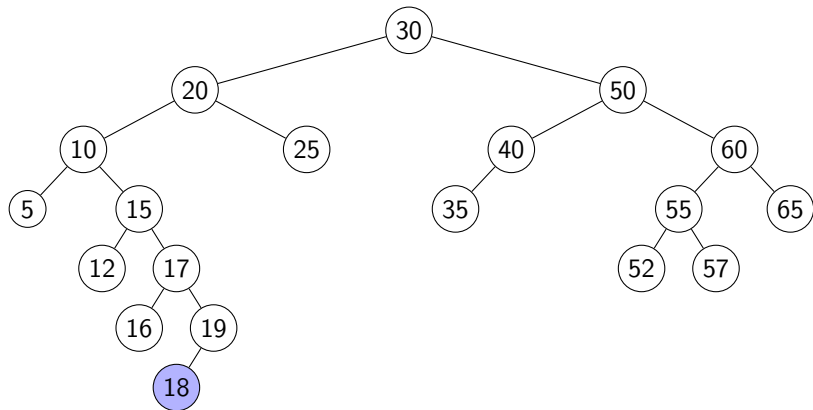
- But: order-property must be maintained!

⇒ Use *rotations*!

(This should remind you of treaps.)

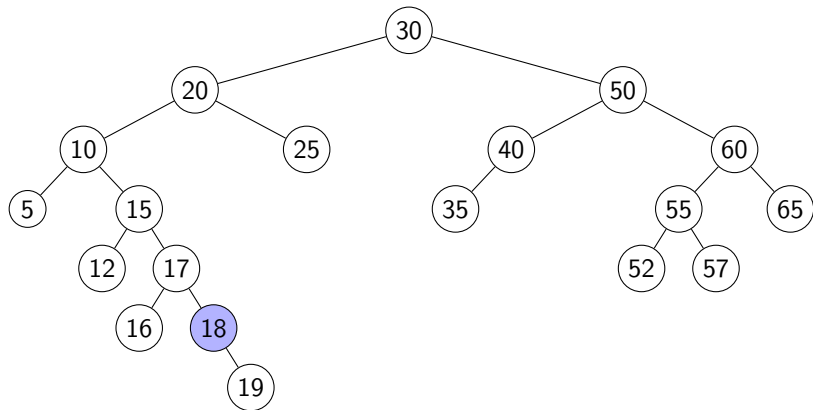
MTF-heuristic for binary search trees

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



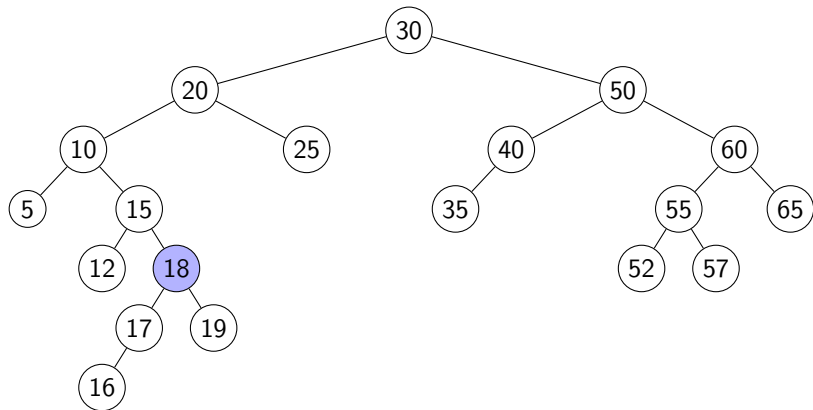
MTF-heuristic for binary search trees

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



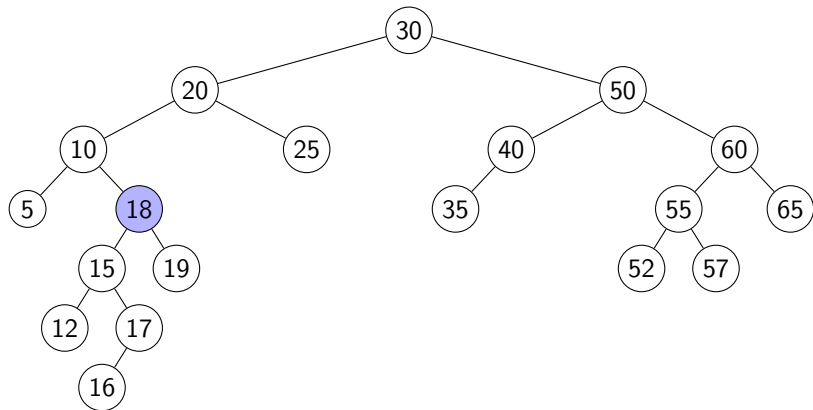
MTF-heuristic for binary search trees

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



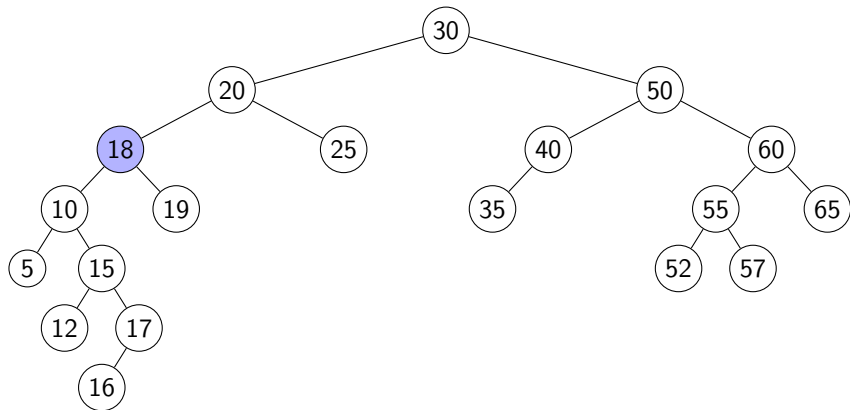
MTF-heuristic for binary search trees

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



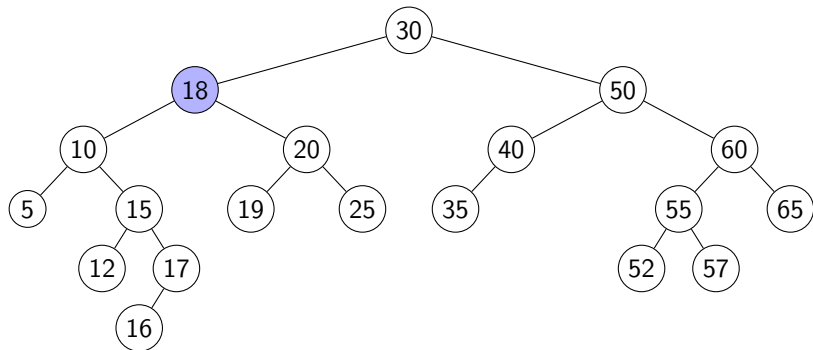
MTF-heuristic for binary search trees

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



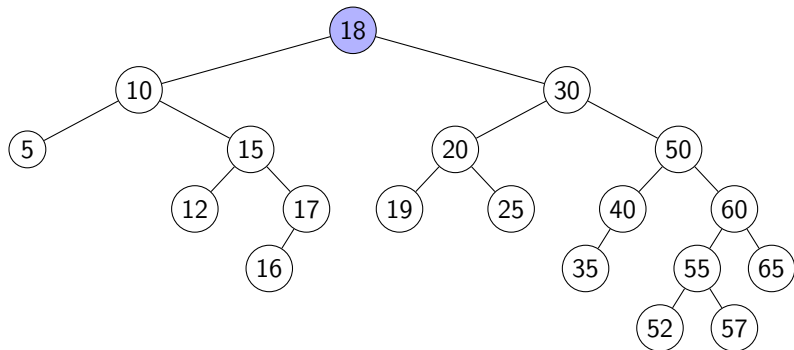
MTF-heuristic for binary search trees

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



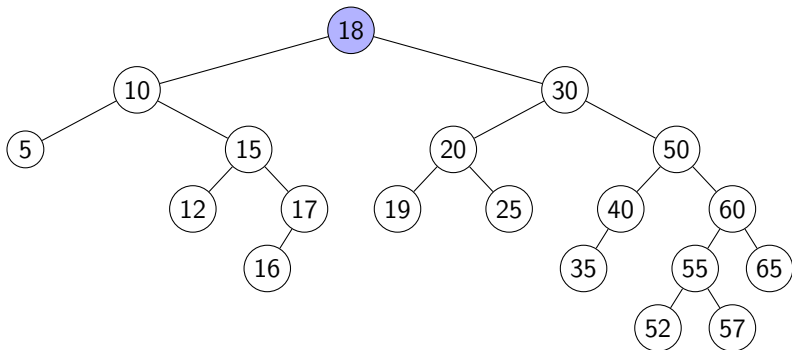
MTF-heuristic for binary search trees

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

- 5 Dictionaries with Lists revisited
 - Expected height of a BST
 - Treaps
 - Skip Lists
 - Biased Search Requests
 - Optimal Static Ordering
 - Optimal Static Binary Search Trees
 - Dynamic Ordering: MTF
 - 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
- As for MTF-heuristic:
After each operation, bring lowest accessed node z to the root.
- In contrast to MTF-heuristic:
move z up two layers at a time when possible.
 - ▶ Let p and g be parent and grand-parent of z .
 - ▶ Study relative order of g - p - z
 - ▶ If p is middle key: Use double-rotation = **zig-zag rotation**
 - ▶ Otherwise: Use **zig-zig rotation** (new!)

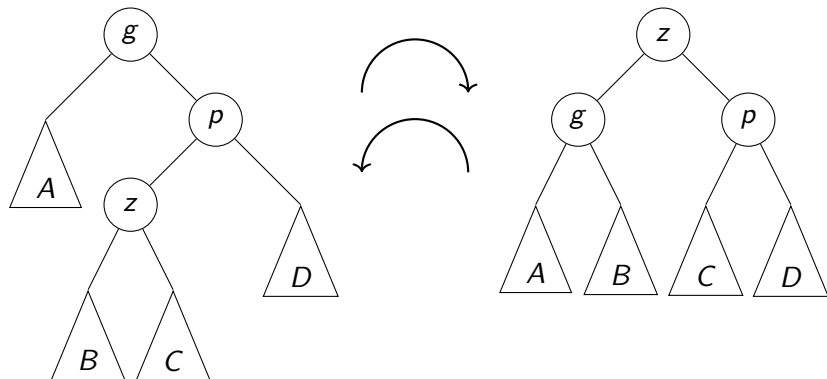
Splay trees

Splay tree overview:

- Binary search tree
- **No** extra information (such as height, balance, size) needed at nodes
- As for MTF-heuristic:
After each operation, bring lowest accessed node z to the root.
- In contrast to MTF-heuristic:
move z up two layers at a time when possible.
 - ▶ Let p and g be parent and grand-parent of z .
 - ▶ Study relative order of g - p - z
 - ▶ If p is middle key: Use double-rotation = **zig-zag rotation**
 - ▶ Otherwise: Use **zig-zig rotation** (new!)

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

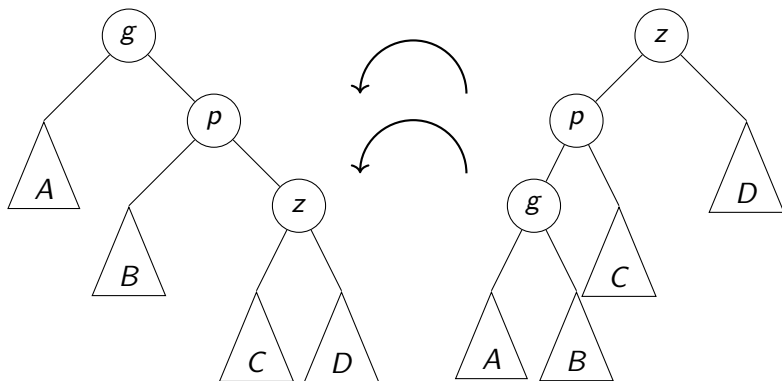
Zig-zag rotation = double rotation



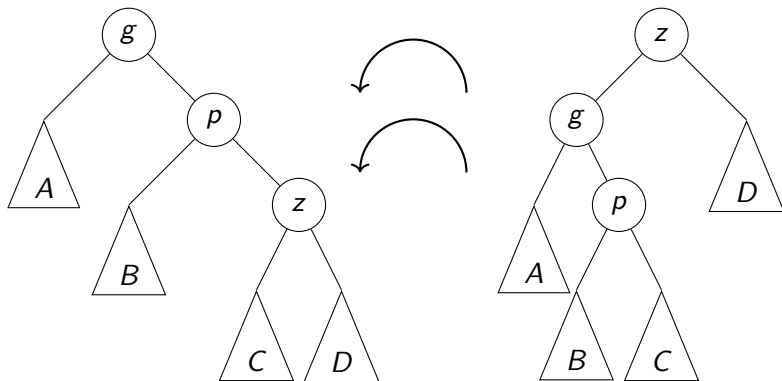
Exactly the same as the double rotation for AVL-trees; only the names of nodes are different.

Zig-zig rotation

New rotation: Rotate at the grand-parent *before* rotating at the parent.



Compare to what MTF does



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

Splay tree: Operations

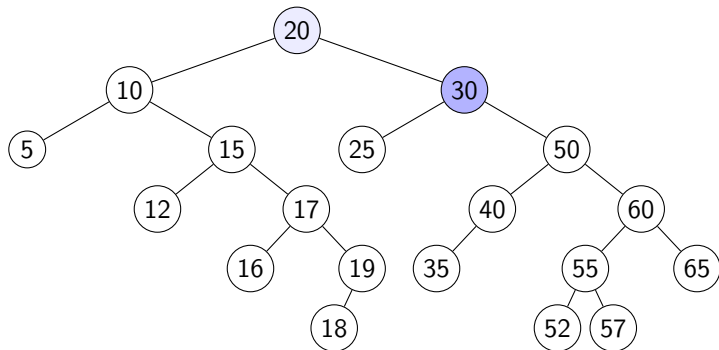
SplayTree::insert(k, v)

1. $z \leftarrow \text{BST::insert}(k, v)$
2. **while** (z is not the root)
3. $p \leftarrow z.\text{parent}$
4. **if** (z is the left child of p)
5. **if** (p is the root) *rotate-right*(p)
6. **else** $g \leftarrow p.\text{parent}$
7. **case**  : // Zig-zig rotation
 rotate-right(g)
 rotate-right(p)
8.  : // Zig-zag rotation
 rotate-right(p)
 rotate-left(g)
9. **else** ... // symmetric , z is right child

search and *delete* similar (rotate the lowest visited node up).

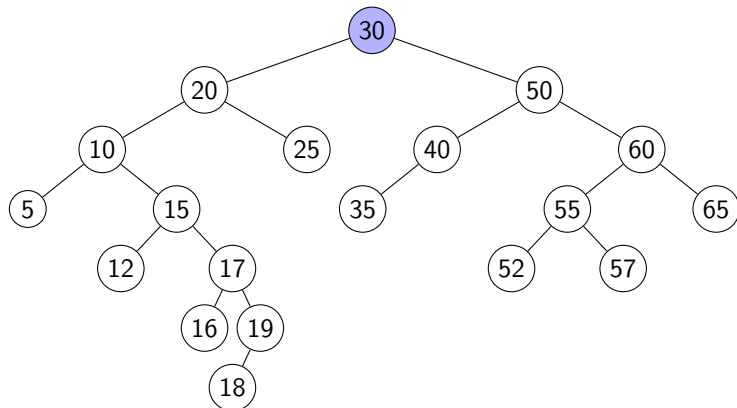
Splay tree: Search

Example: *SplayTree::search*(30)



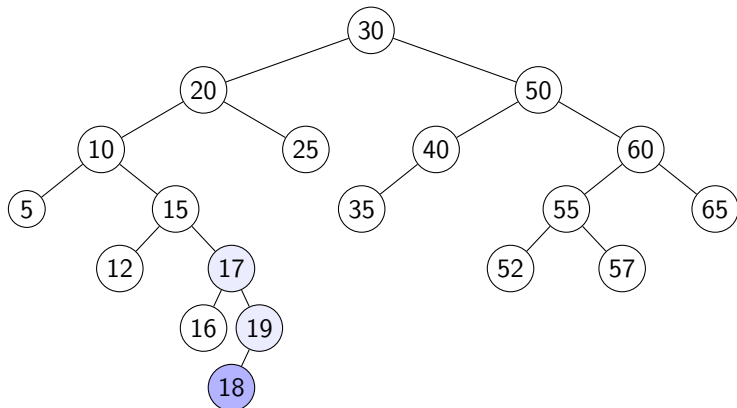
Splay tree: Search

Example: *SplayTree::search*(30)



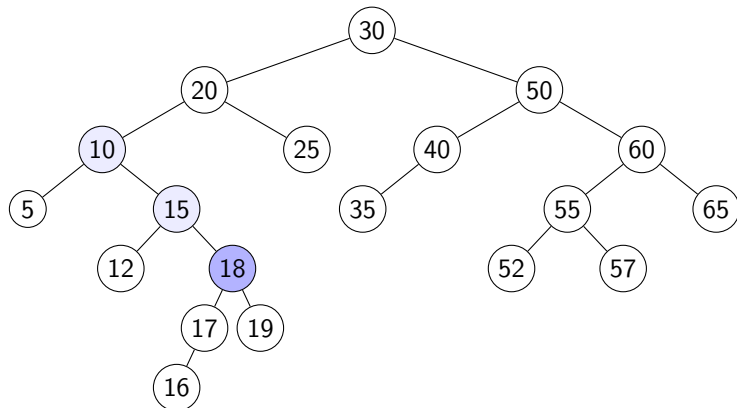
Splay tree: Search

Example: *SplayTree::search*(18)



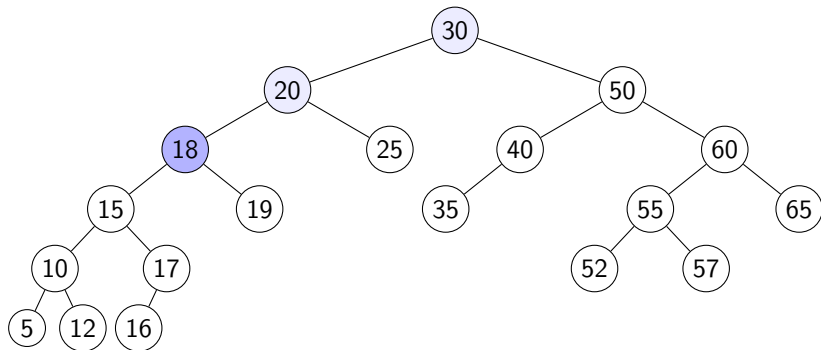
Splay tree: Search

Example: *SplayTree::search*(18)



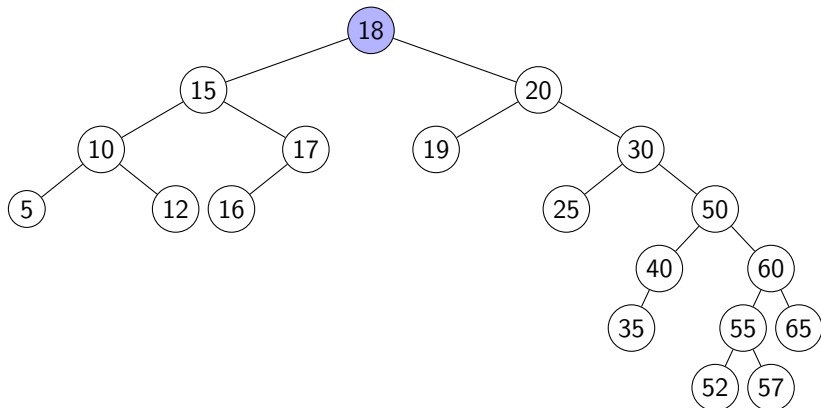
Splay tree: Search

Example: *SplayTree::search*(18)



Splay tree: Search

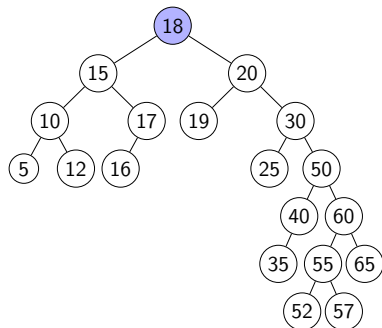
Example: *SplayTree::search*(18)



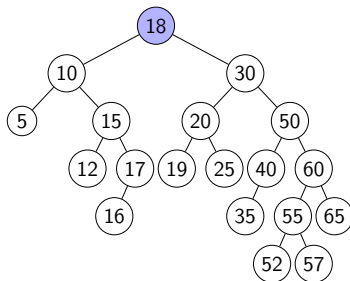
Zig-zig rotations vs. single rotations

Compare the resulting trees:

Splay trees (zig-zig rotations):



With MTF (single rotations):

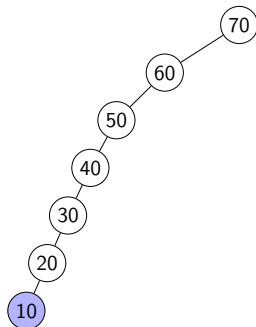


This is *not* more balanced, why do we apply zig-zig-rotations?

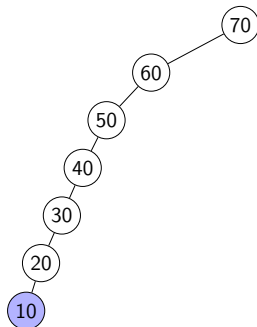
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



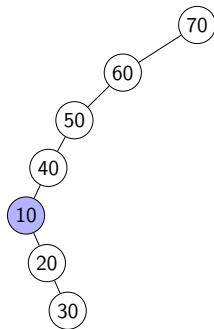
With MTF (single rotations):



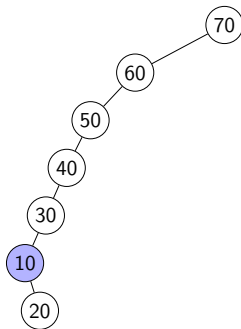
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



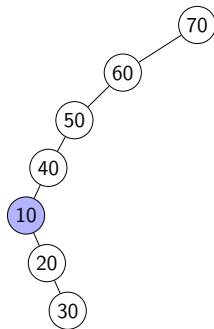
With MTF (single rotations):



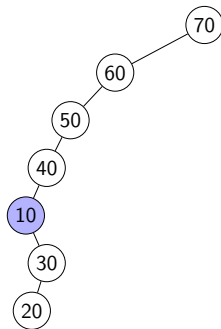
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



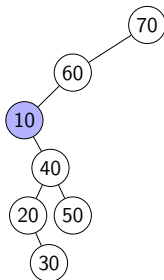
With MTF (single rotations):



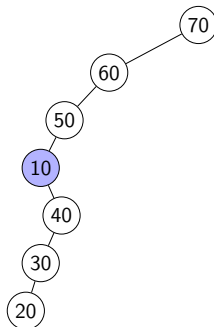
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



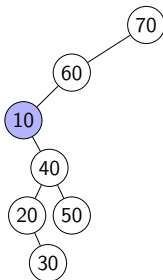
With MTF (single rotations):



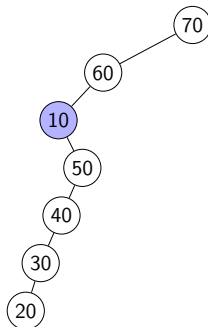
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



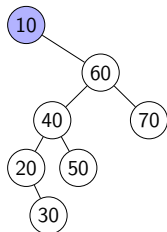
With MTF (single rotations):



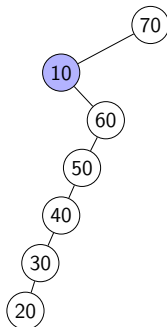
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



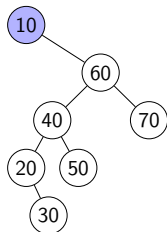
With MTF (single rotations):



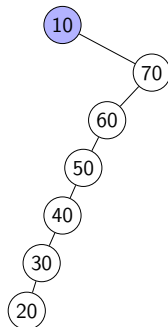
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



With MTF (single rotations):

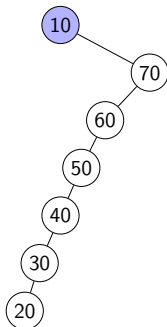
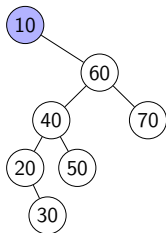


Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):

With MTF (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

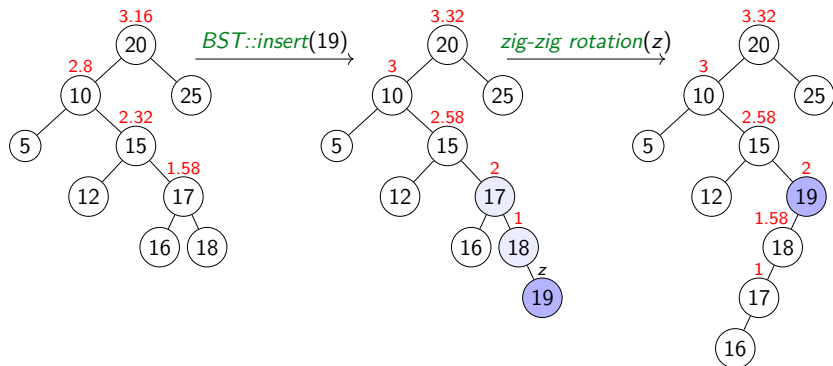
Splay trees: Analysis

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

Splay trees: Analysis

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

Proof: Use potential function $\Phi(t) = \sum_x \log(\text{size of subtree at } x)$



Now compute amortized time of operations (highly non-trivial).