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

Module 5: Other Dictionary Implementations

Therese Biedl

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version *2025−01−29 10:28*

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - 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

- Dictionary ADT: Implementations thus far
- 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

Dictionary ADT: Implementations thus far

A *dictionary* is a collection of key-value pairs (KVPs), supporting operations *search*, *insert*, and *delete*.

Realizations we have seen so far:

- **Unordered array or list**: $\Theta(1)$ insert, $\Theta(n)$ search and delete
- **Ordered array**: $\Theta(\log n)$ search, $\Theta(n)$ insert and delete
- **Binary search trees**: $\Theta(\mathit{height})$ search, insert and delete
- **Balanced Binary Search trees** (AVL trees):
  $\Theta(\log n)$ search, insert, and delete

Dictionary ADT: Implementations thus far

A *dictionary* is a collection of key-value pairs (KVPs), supporting operations *search*, *insert*, and *delete*.

Realizations we have seen so far:

- **Unordered array or list**: $\Theta(1)$ insert, $\Theta(n)$ search and delete
- **Ordered array**: $\Theta(\log n)$ search, $\Theta(n)$ insert and delete
- **Binary search trees**: $\Theta(height)$ search, insert and delete
- **Balanced Binary Search trees** (AVL trees):
  $\Theta(\log n)$ search, insert, and delete

Outlook:

- **We will see:** If the KVPs were inserted in random order, then the expected height of the binary search tree would be $O(\log n)$.
- **Then study:** How can we use randomization within the data structure to mirror what would happen on random input?

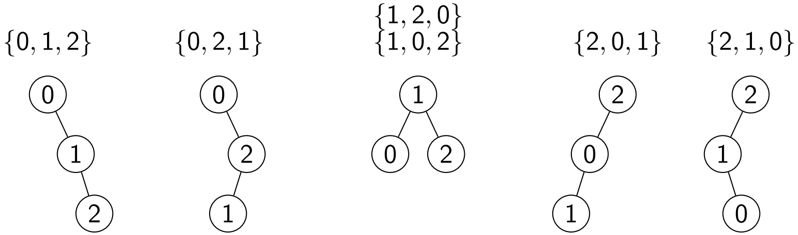
Outline

5 Dictionaries with Lists revisited

- Dictionary ADT: Implementations thus far
- 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

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

5 Dictionaries with Lists revisited

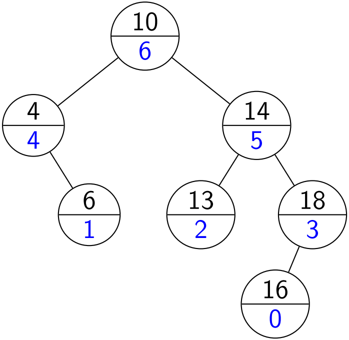
- Dictionary ADT: Implementations thus far
- Expected height of a BST
- Treaps
- Skip Lists
- Biased Search Requests
- Optimal Static Ordering
- Optimal Static Binary Search Trees
- Dynamic Ordering: MTF
- MTF-heuristic in a BST
- Splay Trees

Treaps

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

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

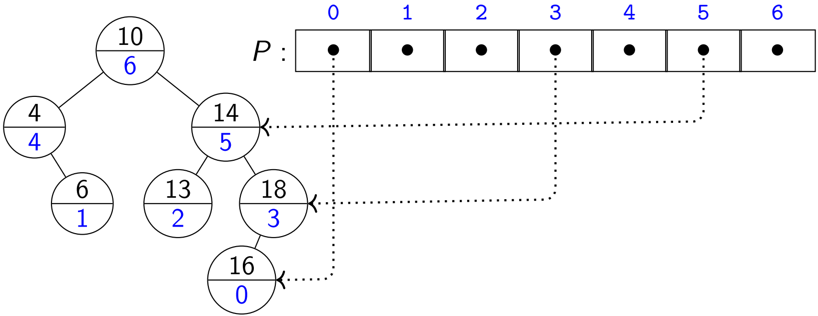
- Priorities are a permutation of $\{0, \ldots, n-1\}$.
- Permutation has been picked *randomly*
- All permutations should be equally likely.
- Priorities are *decreasing* when going downwards (similar to a heap).



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

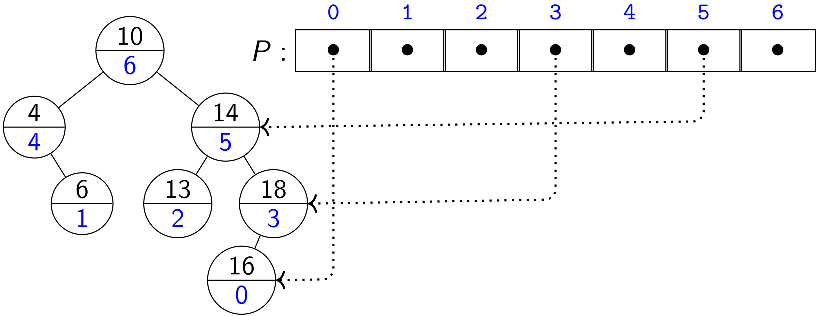
Treaps

We also need an array P where $P[i]$ stores node with priority i.



Treaps

We also need an array P where $P[i]$ stores node with priority i.



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.

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* creates a random permutation:

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

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* creates a random permutation:

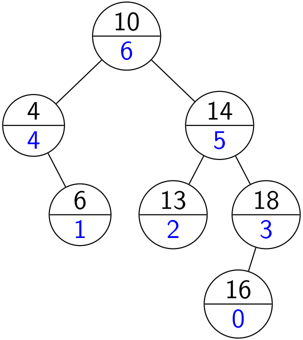
> *shuffle*(A)
> A: array of size n stores $\langle 0, \ldots n{-}1 \rangle$
> **for** $i \leftarrow 1$ to $n-1$ **do**
>   *swap*($A[i], A[\mathit{random}(i+1)]$)

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

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

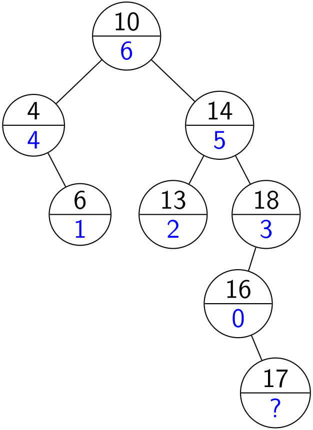
Treap Insertions Example

Example: *treap::insert*(17)



Treap Insertions Example

Example: *treap::insert*(17)

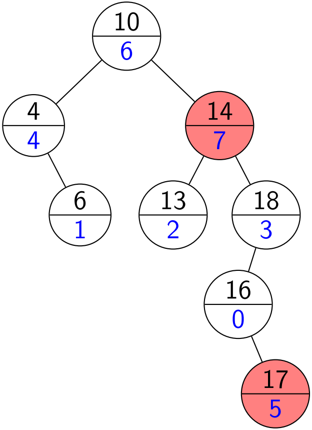


Treap Insertions Example

Example: *treap::insert*(17)

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

These priorities violate order-property.

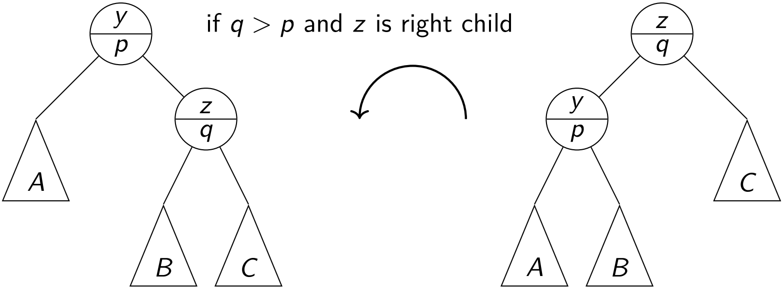


Fixing incorrect priorities with rotations

- 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.

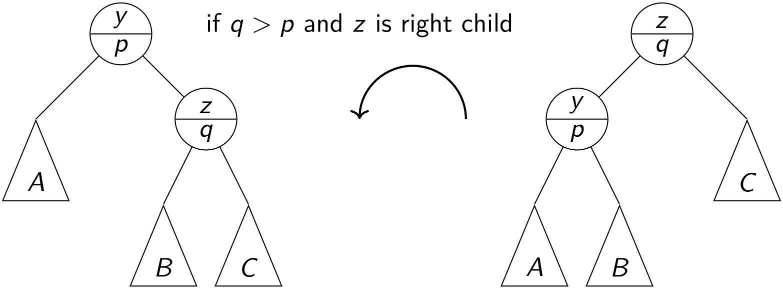
Fixing incorrect priorities with rotations

- 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

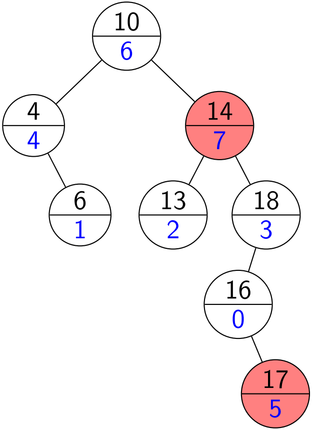
- In binary heaps, we fixed increased priorities via *fix-up* (swaps)
- Does not work here! We must also maintain the BST-order.
- Idea: Rotations maintain the BST-order and fix priorities.



treap::fix-up-with-rotations(z)
1. **while** ($y \leftarrow z.$*parent* is not `NULL` and $z.$*priority* $> y.$*priority*) **do**
2. **if** z is the left child of y **do** *rotate-right*(y)
3. **else** *rotate-left*(y)

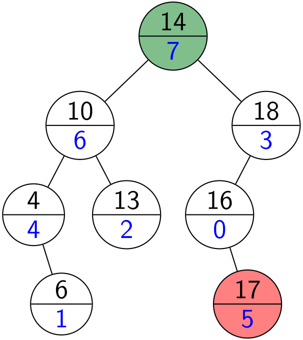
Treap Insertions Example

Example: *treap::insert*(17)



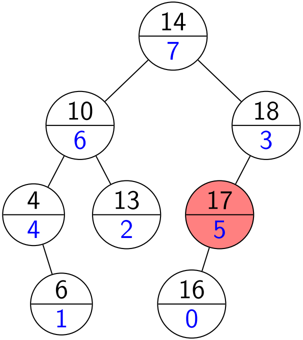
Treap Insertions Example

Example: *treap::insert*(17)
Fix *upper* violation first (why?)



Treap Insertions Example

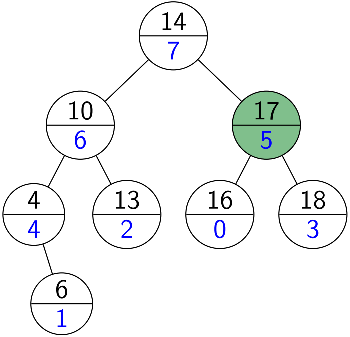
Example: *treap::insert*(17)
Fix *upper* violation first (why?)



Treap Insertions Example

Example: *treap::insert*(17)

Fix *upper* violation first (why?)



Treap Insertion Code

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

treap::insert(k, v)

$n \leftarrow P.\mathit{size}$ `// current size`

$z \leftarrow \mathit{BST::insert}(k, v)$; n++

$p \leftarrow \mathit{random}(n)$

if $p < n - 1$ **do**

  $z' \leftarrow P[p]$, $z'.\mathit{priority} \leftarrow n - 1, P[n-1] \leftarrow z'$

  fix-up-with-rotations(z')

$z.\mathit{priority} \leftarrow p$; $P[p] \leftarrow z$

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)

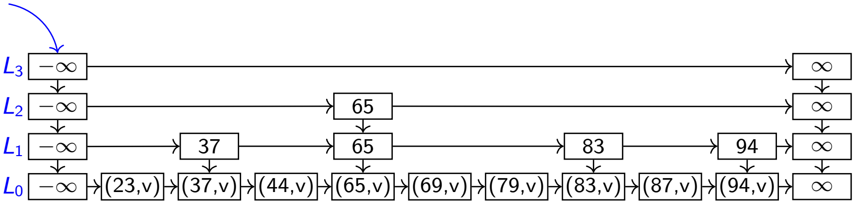
Outline

5 Dictionaries with Lists revisited

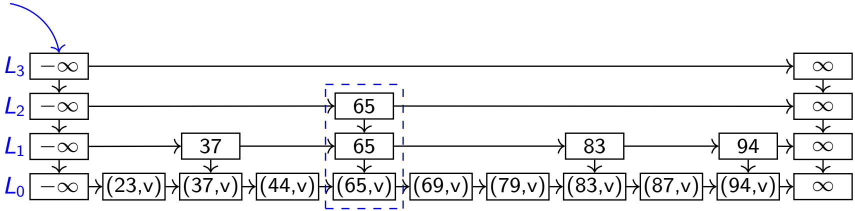
Skip Lists

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

- 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.,
  $L_0 \supseteq L_1 \supseteq \cdots \supseteq L_h$
- List L_h contains only the sentinels



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

Search in Skip Lists

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

get-predecessors (k)

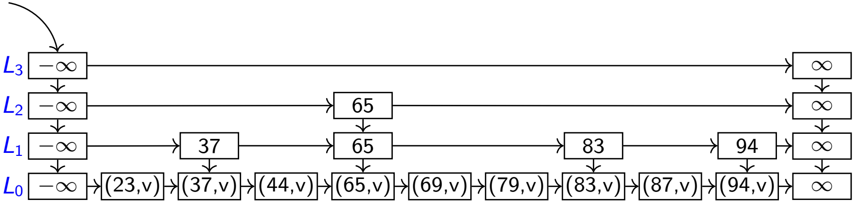
```
p ← root
P ← stack of nodes, initially containing p
while p.below ≠ NULL do
      p ← p.below
      while p.after.key < k do p ← p.after
      P.push(p)
return P
```

skipList::search (k)

```
P ← get-predecessors(k)
p0 ← P.top() // predecessor of k in L0
if p0.after.key = k    return KVP at p0.after
else return "not found, but would be after p0"
```

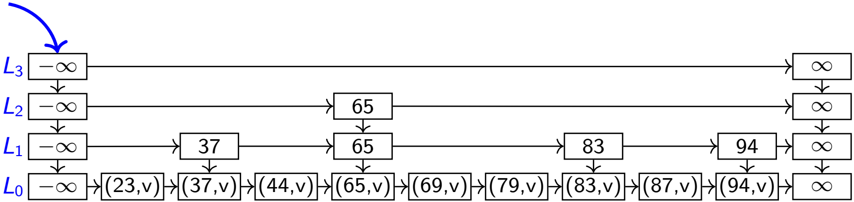
Example: Search in Skip Lists

Example: *search*(87)



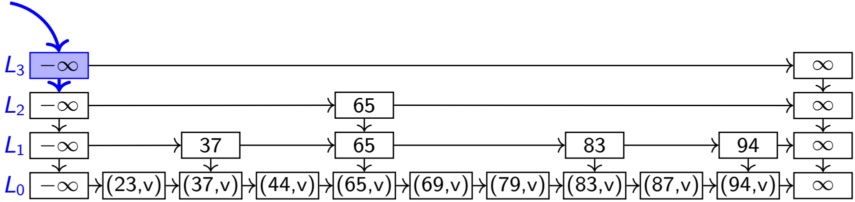
Example: Search in Skip Lists

Example: *search*(87)



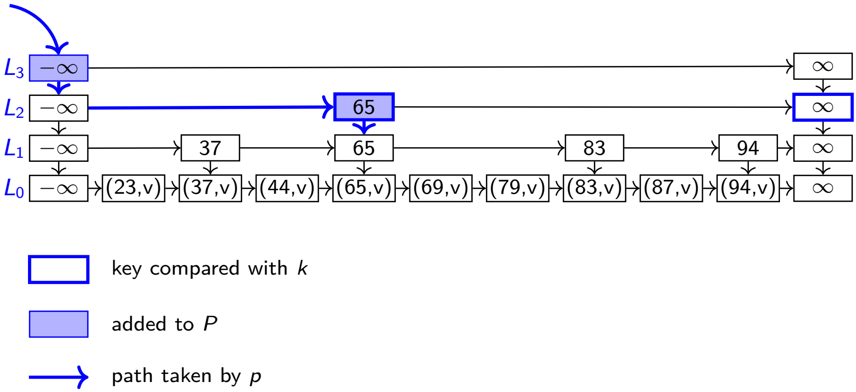
Example: Search in Skip Lists

Example: *search*(87)



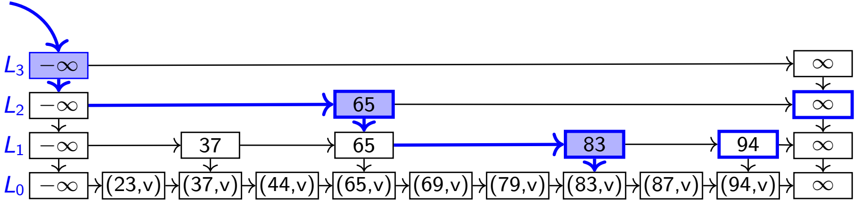
Example: Search in Skip Lists

Example: *search*(87)



Example: Search in Skip Lists

Example: *search*(87)



key compared with k

added to P

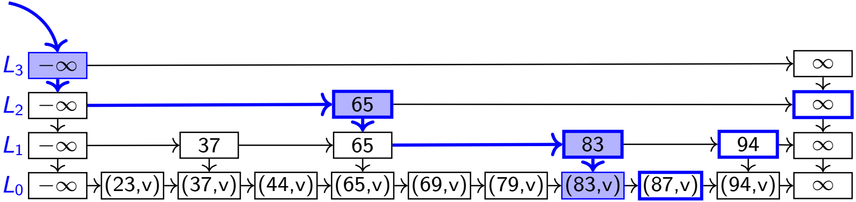
path taken by p

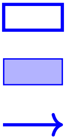
Final stack returned:

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

Example: Search in Skip Lists

Example: *search*(87)





key compared with k

added to P

path taken by p

Final stack returned:

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

Delete in Skip Lists

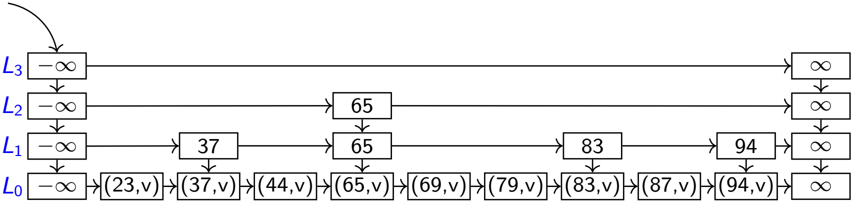
It is easy to remove a key since we can find all predecessors.
Then eliminate lists if there are multiple ones with only sentinels.

```
skipList::delete(k)
P ← get-predecessors(k)
while P is non-empty
    p ← P.pop()          // predecessor of k in some list
    if p.after.key = k
        p.after ← p.after.after
    else break           // no more copies of k

p ← left sentinel of the root-list
while p.below.after is the ∞-sentinel
    // top two lists have only sentinels, remove one
    p.below ← p.below.below
    p.after.below ← p.after.below.below
```

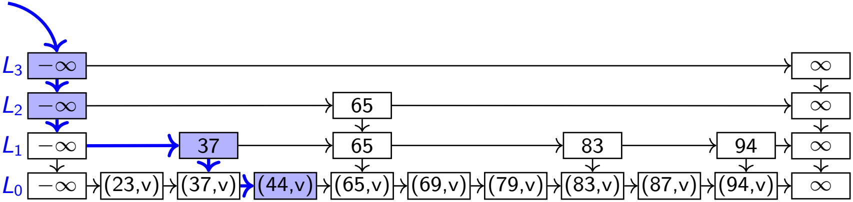
Example: Delete in Skip Lists

Example: *skipList::delete*(65)



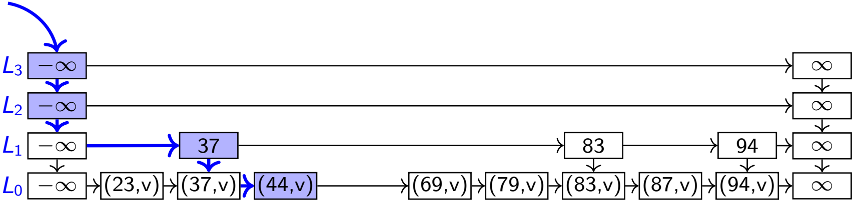
Example: Delete in Skip Lists

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



Example: Delete in Skip Lists

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

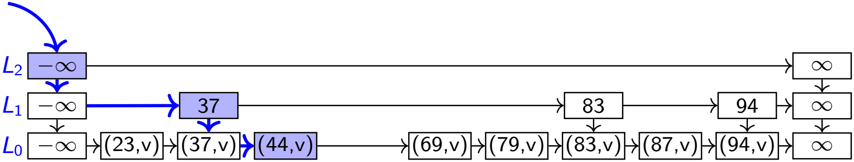


Example: Delete in Skip Lists

Example: *skipList::delete*(65)

get-predecessors(65)

Height decrease



Insert in Skip Lists

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 $L_0, \ldots, L_i$, so $i \to$ *height* of tower of k

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

Insert in Skip Lists

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 $L_0, \ldots, L_i$, so $i \to$ *height* of tower of k

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

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

Insert in Skip Lists

skipList::insert(k, v)

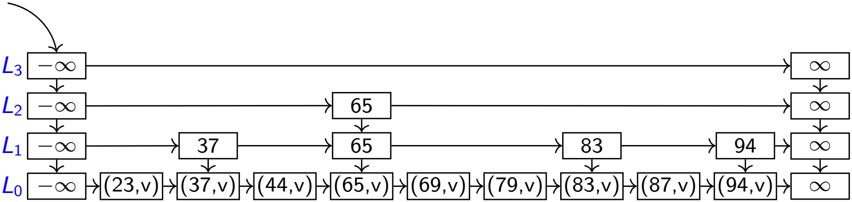
- There is no choice as to where to put the tower of k.
- Only choice: how tall should we make the tower of k?
  - Choose *randomly*! Repeatedly toss a coin until you get tails
  - Let i the number of times the coin came up heads
  - We want key k to be in lists $L_0, \dots, L_i$, so $i \to$ *height*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

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

- 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 $p_0, p_1, \cdots, p_i$ of where k should be in each list $L_0, L_1, \cdots, L_i$
  - Insert (k, v) after p_0 in L_0, and k after p_j in L_j for $1 \leq j \leq i$

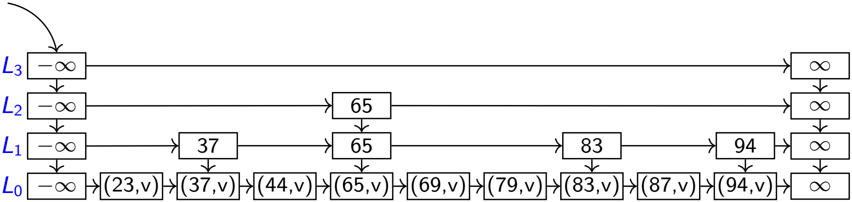
Example: Insert in Skip Lists

Example: *skipList::insert*$(52, v)$
Coin tosses: H,T $\Rightarrow i = 1$



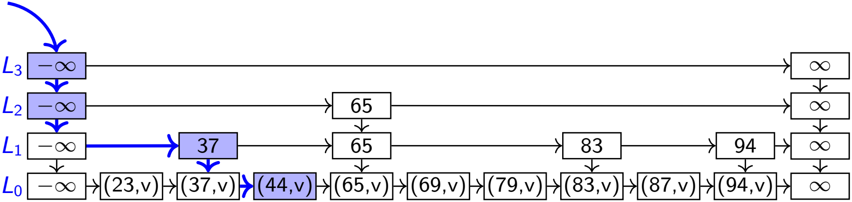
Example: Insert in Skip Lists

Example: *skipList::insert*$(52, v)$
Coin tosses: H,T $\Rightarrow i = 1$
Have $h = 3 > i \Rightarrow$ no need to add lists



Example: Insert in Skip Lists

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)



Example: Insert in Skip Lists

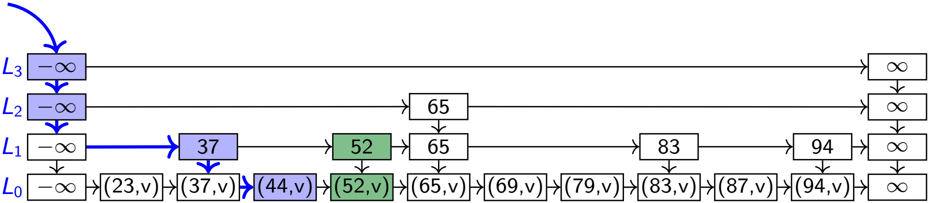
Example: *skipList::insert*$(52, v)$

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

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

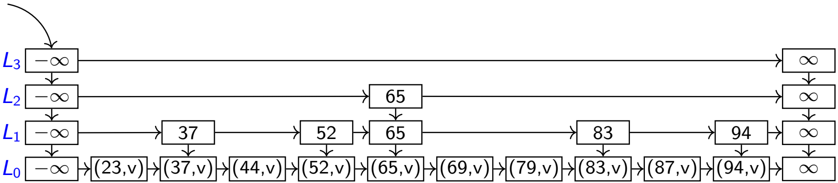
get-predecessors(52)

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



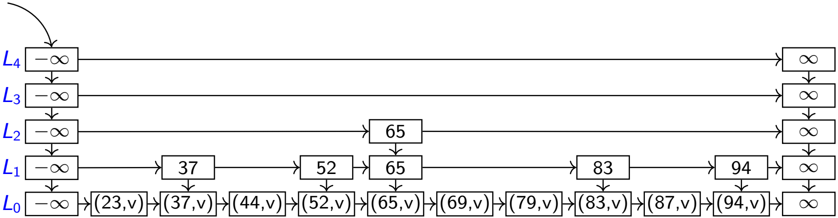
Example 2: Insert in Skip Lists

Example: *skipList::insert*$(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$



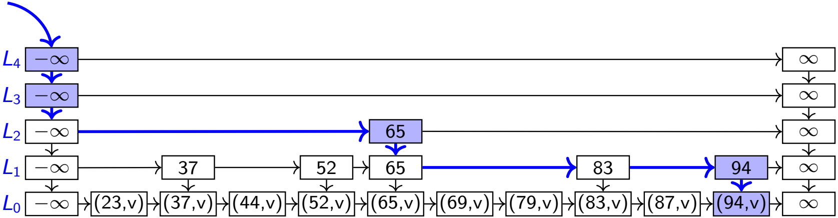
Example 2: Insert in Skip Lists

Example: *skipList::insert*$(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase



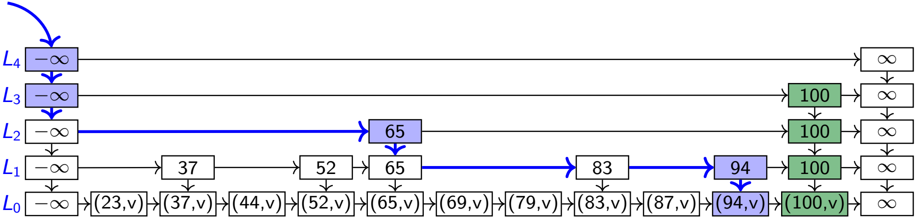
Example 2: Insert in Skip Lists

Example: *skipList::insert*$(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase
get-predecessors(100)



Example 2: Insert in Skip Lists

Example: *skipList::insert*$(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase
get-predecessors(100)
Insert 100 in lists $L_0, \ldots, L_i$



Skip Lists Space

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

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

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

Skip Lists Space

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

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

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

- 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 $\Pr(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
- Define $|L_i|$ = #non-sentinels in L_i. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.

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

- 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 is $O(\log n)$.

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

Skip Lists Height

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

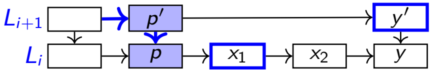
- Define $I_i = \chi_{(L_i \text{ has non-sentinels})} = \begin{cases} 1 & \text{if } |L_i| \geq 1 \\ 0 & \text{otherwise} \end{cases}$
- Observe: $I_i \leq \min\{1, |L_i|\}$, so $E[I_i] \leq \min\{1, E[|L_i|]\}$.
- Observe: height $h = \#\text{lists with non-sentinels} = \sum_{i \geq 0} I_i$.
- What 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 $p \leftarrow p.below$)? *height*.
- How often do we *step forward* (execute $p \leftarrow p.after$)?
  - We immediately drop down in L_h, so consider L_i for $i < h$.

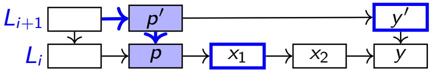


  - Key insight: x_1 did *not* exist in L_{i+1} (else would go forward there)
  - So the tower of x_1 *exactly* ended with L_i. What is the probability?

Skip Lists: getting predecessors

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

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

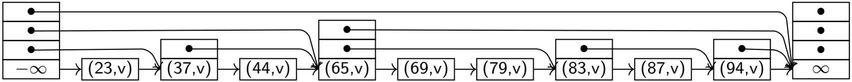


  - Key insight: x_1 did *not* exist in L_{i+1} (else would go forward there)
  - So the tower of x_1 *exactly* ended with L_i. What is the probability?

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

Summary of Skip Lists

- $O(n)$ 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
   - Dictionary ADT: Implementations thus far
   - 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

Improving unsorted lists/arrays

Recall *unsorted array* realization:

0	1	2	3	4
90	30	60	20	50

- *search*: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Very simple and popular. Can we do something to make search more effective in practice?

Improving unsorted lists/arrays

Recall *unsorted array* realization:

0	1	2	3	4
90	30	60	20	50

- *search*: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Very simple and popular. Can we do something to make search more effective in practice?
- No: if items are accessed equally likely.
  We can show that the average-case cost for *search* is then $\Theta(n)$.
- Yes: if the search requests are **biased**:
  some items are accessed much more frequently than others.
  - 80/20 rule: 80% of outcomes result from 20% of causes.
  - **access**: insertion or successful search
  - Intuition: Frequently accessed items should be in the front.
  - Two scenarios: Do we know the access distribution beforehand or not?

Outline

5 Dictionaries with Lists revisited

- Dictionary ADT: Implementations thus far
- 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

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
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

$$\begin{aligned} \text{Recall: } T^{avg}(n) &= \textstyle\sum_{I \in \mathcal{I}_n} T(I) \cdot (\text{relative frequency of } I) \\ &= \text{expected run-time on randomly chosen input} \\ &= \textstyle\sum_{I \in \mathcal{I}_n} T(I) \cdot \Pr(\text{randomly chosen instance is } I) \end{aligned}$$

- Count cost i if search-key ($=$ instance I) is at ith position ($i \geq 1$).
- So we analyze

$$\textbf{expected access cost} = \sum_{i \geq 1} i \cdot \underbrace{\Pr\left(\text{search for key at position } i\right)}_{\textbf{access-probability} \text{ of that key}}$$

Optimal Static Ordering

- Order $\boxed{A} \to \boxed{B} \to \boxed{C} \to \boxed{D} \to \boxed{E}$ 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 $\boxed{D} \to \boxed{B} \to \boxed{E} \to \boxed{A} \to \boxed{C}$ 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

- Order $\boxed{A} \longrightarrow \boxed{B} \longrightarrow \boxed{C} \longrightarrow \boxed{D} \longrightarrow \boxed{E}$ 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 $\boxed{D} \longrightarrow \boxed{B} \longrightarrow \boxed{E} \longrightarrow \boxed{A} \longrightarrow \boxed{C}$ 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?

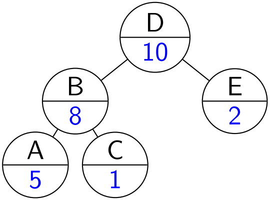
Outline

5 Dictionaries with Lists revisited
- Dictionary ADT: Implementations thus far
- 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 Binary Search Trees

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

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

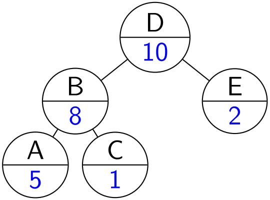


- The expected access-cost is now $\sum_k Pr(k) \cdot (1 + \text{depth of } k)$ since we use $(1 + \text{depth of } k)$ comparisons to search for key k.

Optimal Static Binary Search Trees

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

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



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

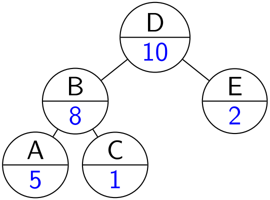
- The expected access-cost is now $\sum_k Pr(k) \cdot (1 + \text{depth of } k)$

  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
$Pr(k_i)$	$\frac{5}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{2}{26}$



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

- The expected access-cost is now $\sum_k Pr(k) \cdot (1 + \text{depth of } k)$

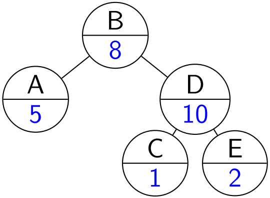
  since we use $(1 + \text{depth of } k)$ comparisons to search for key k.

- 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
$Pr(k_i)$	$\frac{5}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{2}{26}$

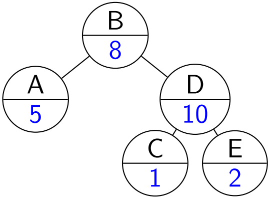


$1 \cdot \frac{8}{26} + 2 \cdot \frac{5}{26} + 2 \cdot \frac{10}{26} + 3 \cdot \frac{1}{26} + 3 \cdot \frac{2}{26} = \frac{47}{26}$

Optimal static binary search trees

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

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



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

- 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 perhaps not for this problem)

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - 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

$$A \to B \to C \to D \to E$$

$\downarrow$ search(D)

$$D \to A \to B \to C \to E$$

$\downarrow$ insert(F)

$$F \to D \to A \to B \to C \to E$$

- 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

A → B → C → D → E

↓ search(D)

A → B → D → C → E

↓ insert(F)

F → A → B → D → C → E

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

  A → B → C → D → E

  ↓ search(D)

  A → B → D → C → E

  ↓ insert(F)

  F → A → B → D → C → E

  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.

Summary of biased search requests

- 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.

Summary of biased search requests

- 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": $cost(MTF) \leq 2 \cdot cost(OPT)$.

)

Summary of biased search requests

- 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": $cost(MTF) \leq 2 \cdot cost(OPT)$.
- 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
   - Dictionary ADT: Implementations thus far
   - 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.

$\Rightarrow$ 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.

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

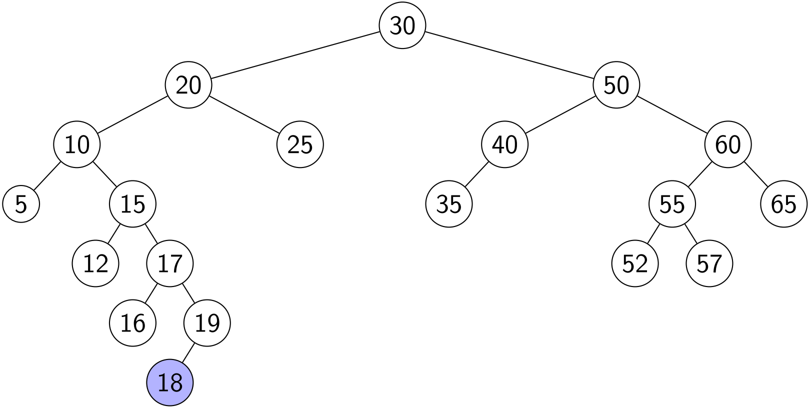
- But: order-property must be maintained!

$\Rightarrow$ Use *rotations*!

(This should remind you of treaps.)

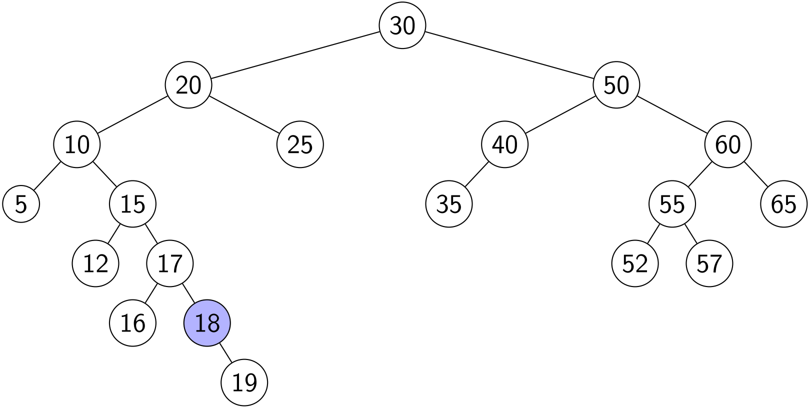
MTF-heuristic for binary search trees

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



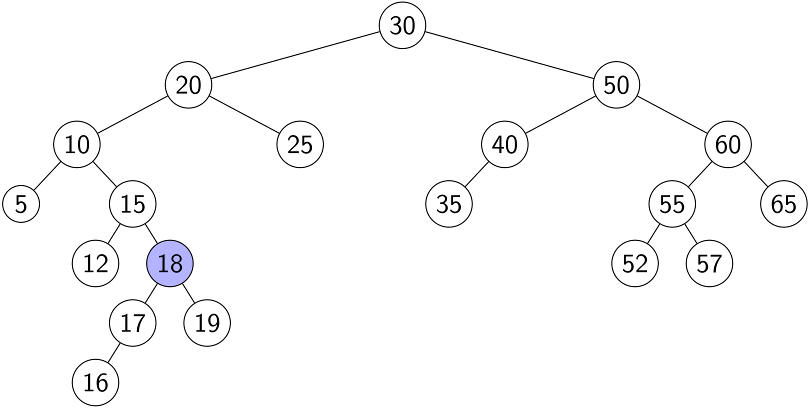
MTF-heuristic for binary search trees

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



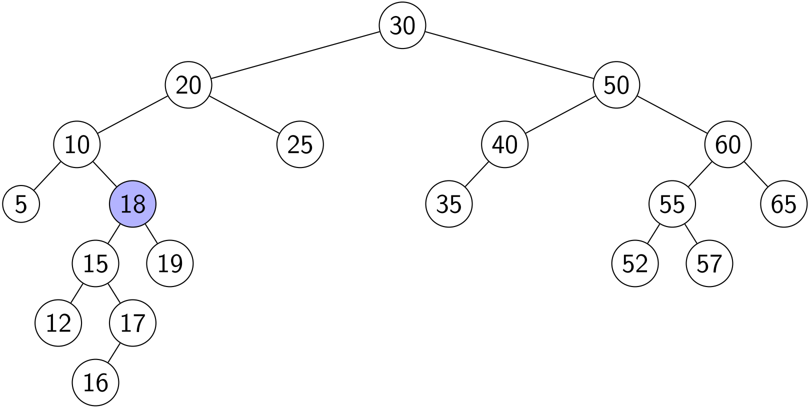
MTF-heuristic for binary search trees

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



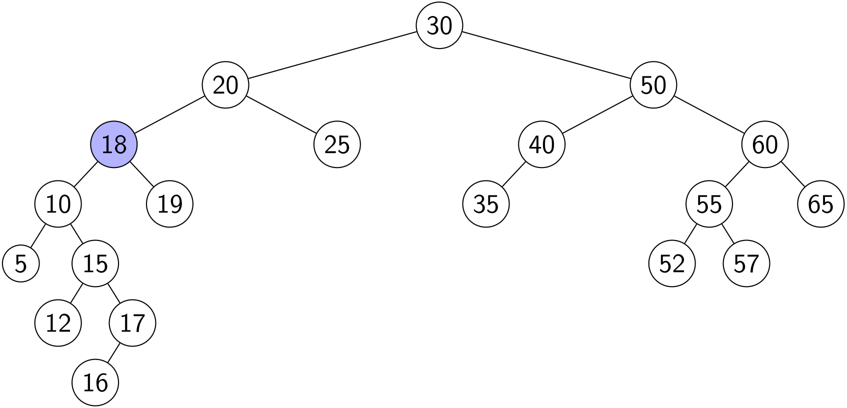
MTF-heuristic for binary search trees

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



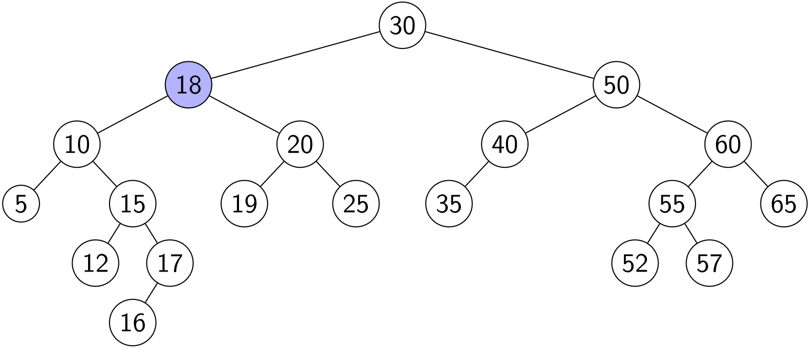
MTF-heuristic for binary search trees

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



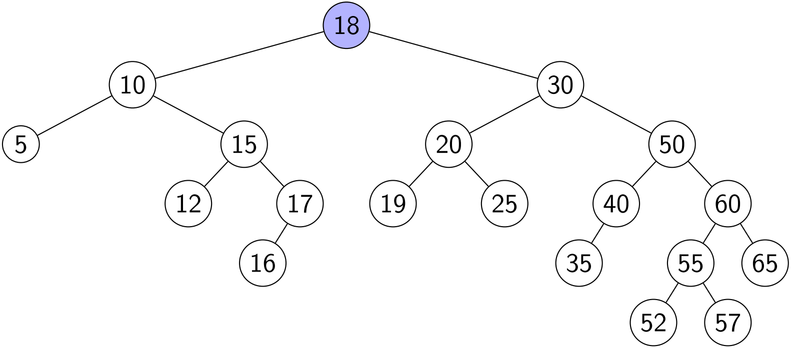
MTF-heuristic for binary search trees

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



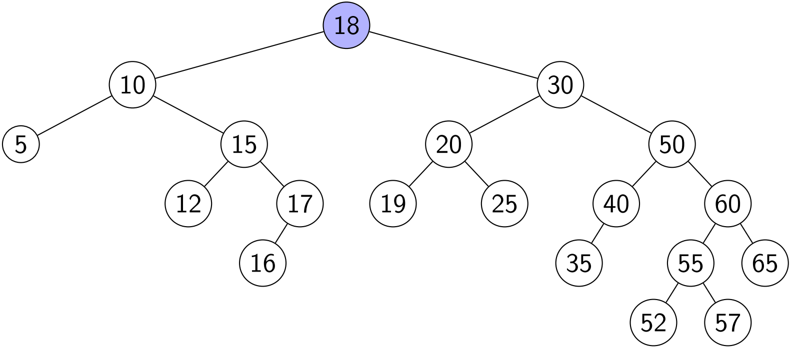
MTF-heuristic for binary search trees

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

Splay trees

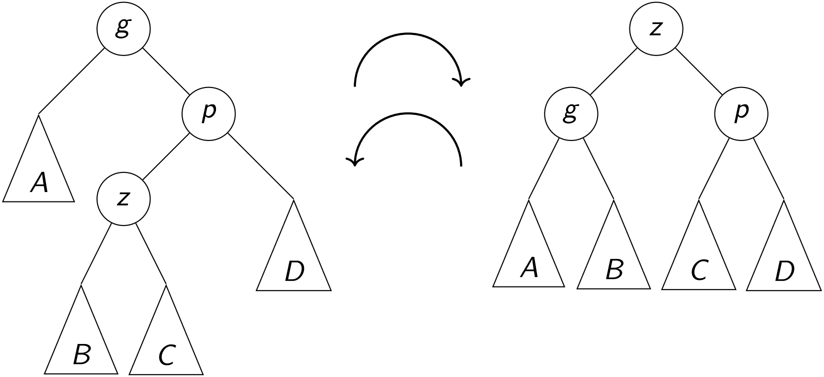
Splay tree overview:

- 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 $O(\log n)$.

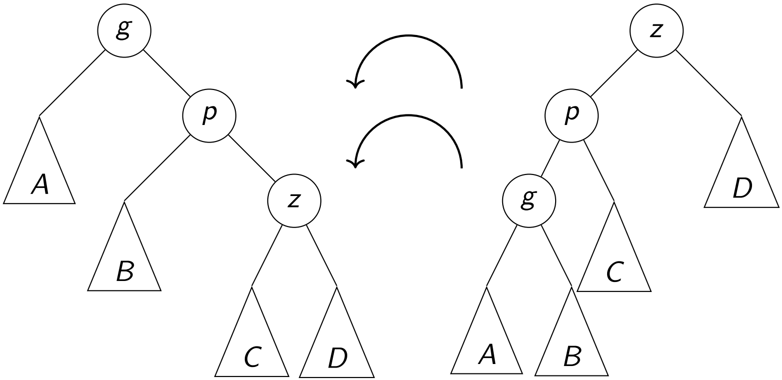
Zig-zag Rotation = Double Rotation

- 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

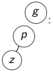
- If they are in zig-zig formation, apply a new kind of rotation.

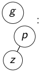


First, a left rotation at g. Second, a left rotation at p.

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 

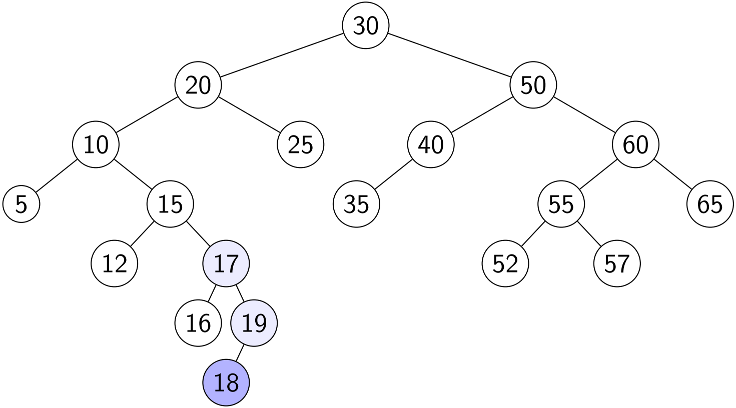
        : // Zig-zig rotation
            rotate-right(g)
            rotate-right(p)
        

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

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

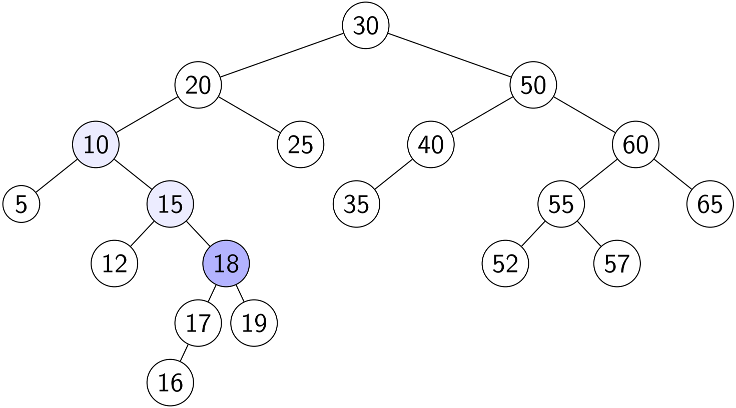
Splay Tree Insert

Example: *SplayTree::search*(18)



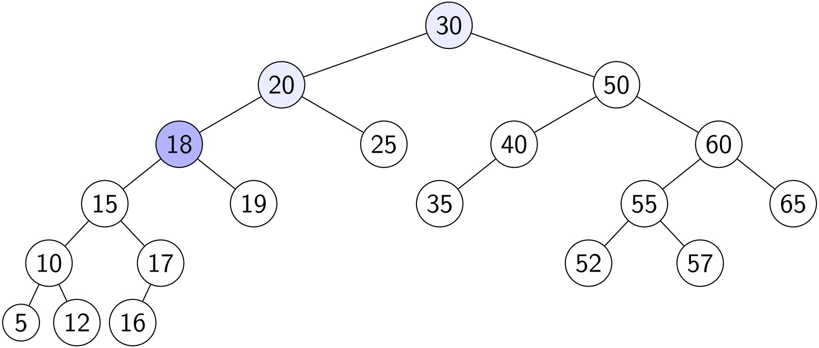
Splay Tree Insert

Example: *SplayTree::search*(18)



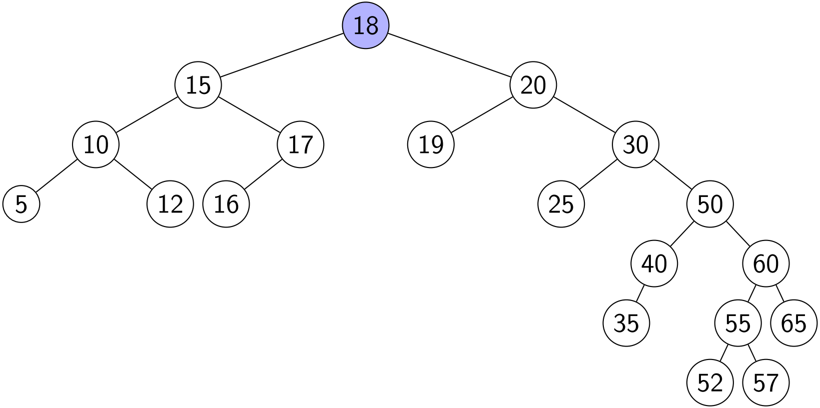
Splay Tree Insert

Example: *SplayTree::search*(18)



Splay Tree Insert

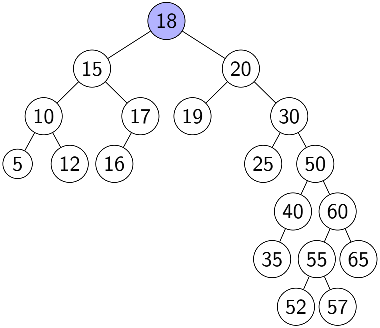
Example: *SplayTree::search*(18)



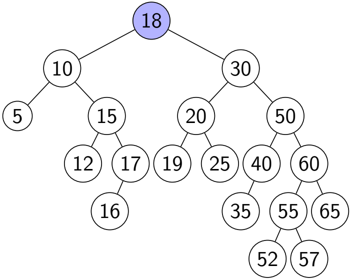
Zig-zig rotations vs. single rotations

Compare the resulting trees:

Splay trees (zig-zig rotations):



With MTF (single rotations):

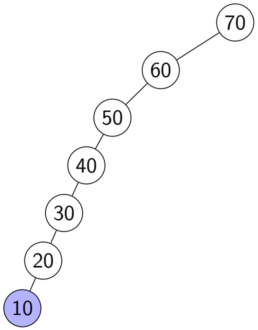


This is *not* more balanced, why do we apply zig-zig-rotations?

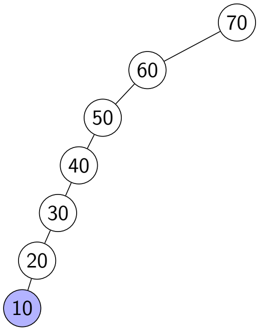
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



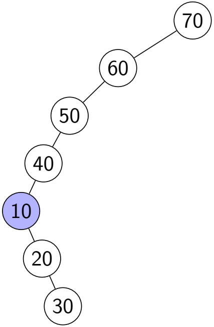
With MTF (single rotations):



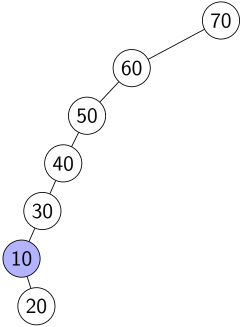
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



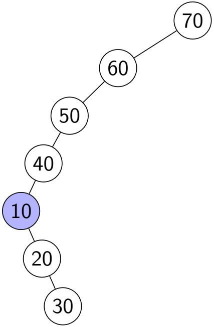
With MTF (single rotations):



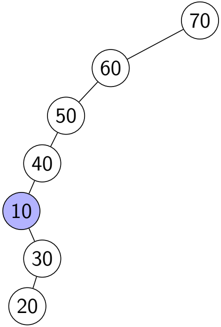
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



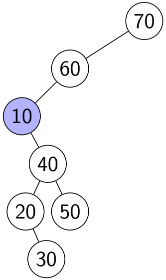
With MTF (single rotations):



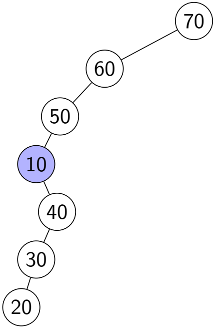
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



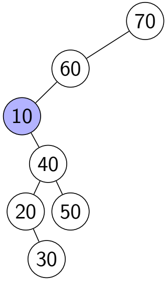
With MTF (single rotations):



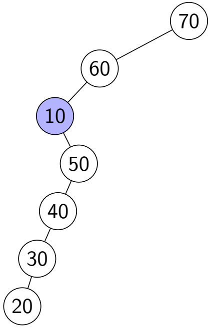
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



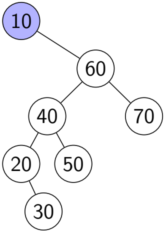
With MTF (single rotations):



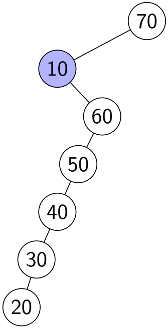
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



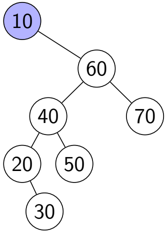
With MTF (single rotations):



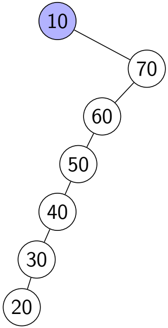
Zig-zig rotations vs. single rotations

Compare the result for a different initial tree:

Splay trees (zig-zig rotations):



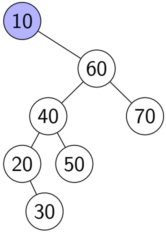
With MTF (single rotations):



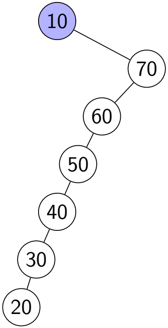
Zig-zig rotations vs. single rotations

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

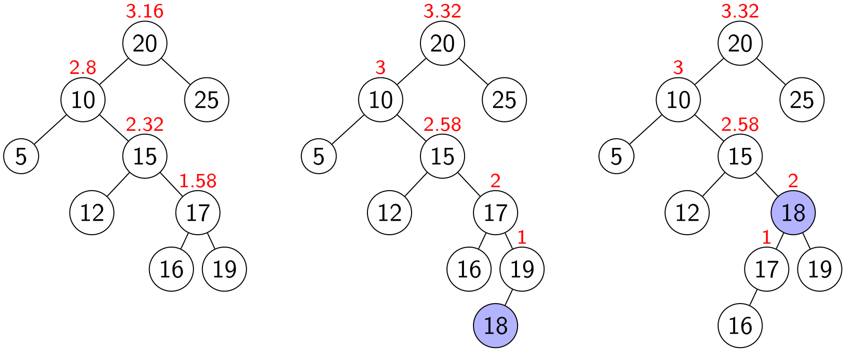
Splay tree analysis

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

Splay tree analysis

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

Proof: Use potential function $\Phi(t) = \sum_z \log\left(z.size()\right)$

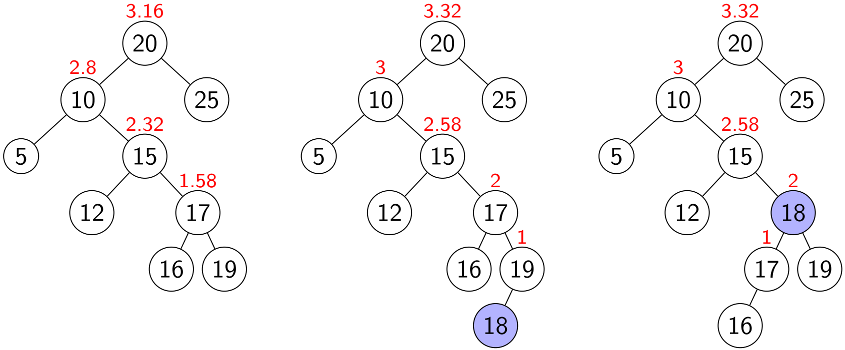


Now compute amortized time of operations (highly non-trivial).

Splay tree analysis

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

Proof: Use potential function $\Phi(t) = \sum_z \log\left(z.size()\right)$



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

Summary: Splay trees perform well, *without* extra information (such as height or size) at nodes.