Treaps and Skip Lists

COP3503 COMPUTER SCIENCE II

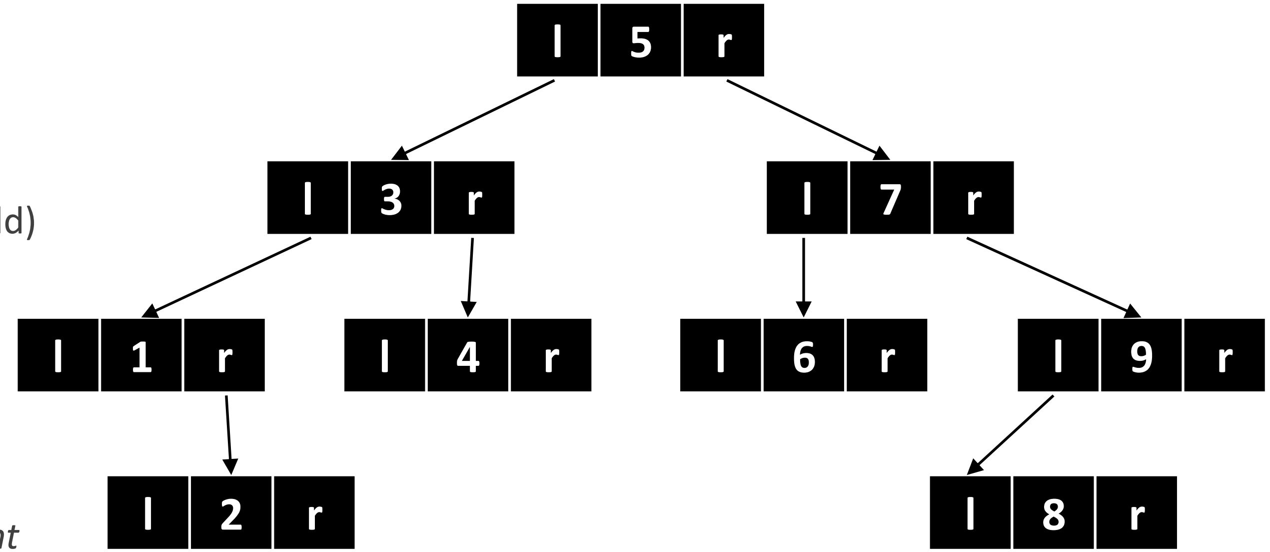
DR. MATTHEW B. GERBER

PORTIONS FROM SEAN SZUMLANSKI, MICHAEL MCALPIN, AND WIKIPEDIA

Review: Binary Search Trees

A *binary search tree* is a tree in which:

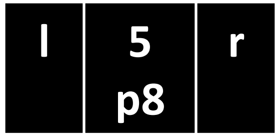
- Each node has at most two children
- Each node has at least the following data elements:
  - A **parent** pointer (NULL if the node is the root)
  - A **left** child pointer (NULL if the node has no left child)
  - A **right** child pointer (NULL if the node has no right child)
  - An indexing **key**
- The insertion and deletion behaviors guarantee that:
  - If node y is the left child of node x, or *any descendant* of the left child of x, then $y.key \leq x.key$
  - If node y is the right child of node x, or *any descendant* of the right child of x, then $y.key \geq x.key$



Treaps

A *treap* (tree-heap) has an additional property:

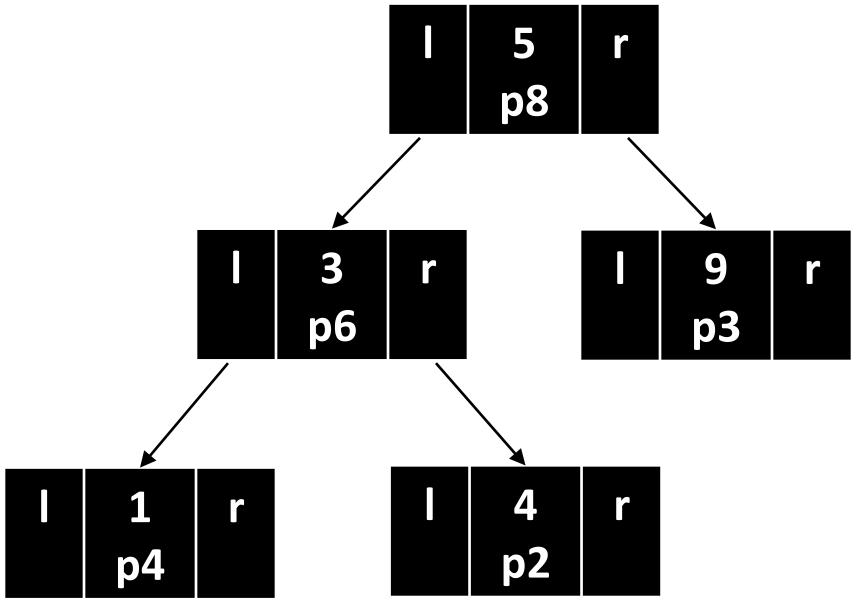
- Each node has an additional value, called *priority*



Treaps

A *treap* (tree-heap) has an additional property:

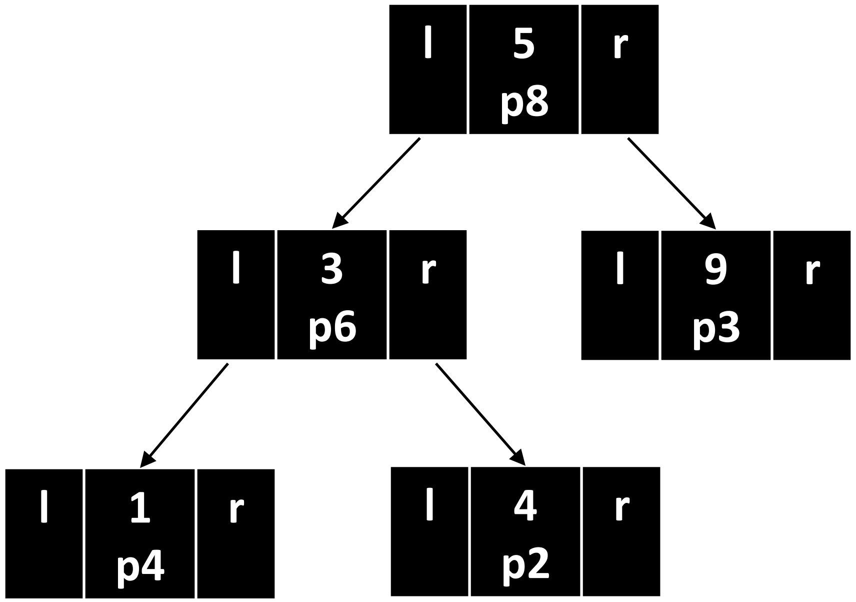
- Each node has an additional value, called *priority*
- The priority of a node must be *at least as great as the priority of its children*



Treaps

A *treap* (tree-heap) has an additional property:

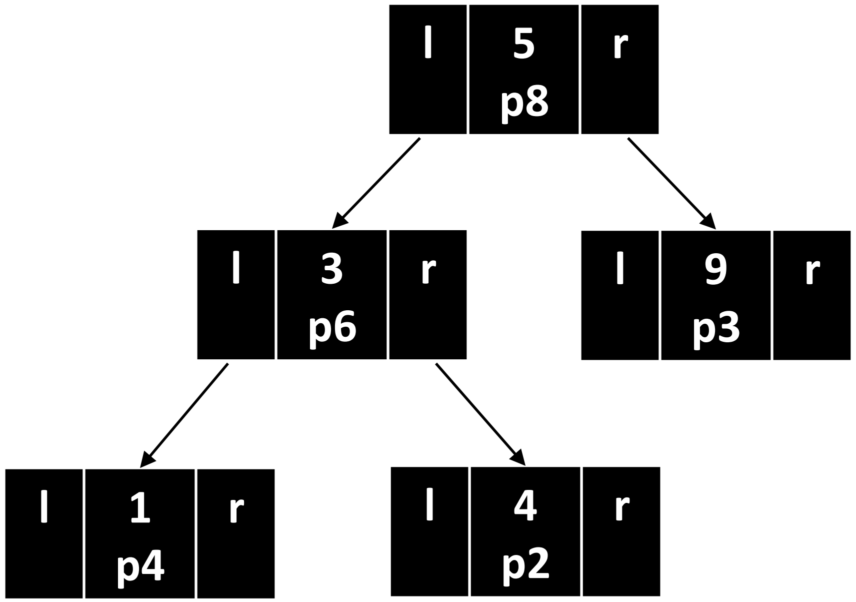
- Each node has an additional value, called *priority*
- The priority of a node must be *at least as great as the priority of its children*
- This means that the *effective order of insertion* into the tree is *random*



Treaps

A *treap* (tree-heap) has an additional property:

- Each node has an additional value, called *priority*
- The priority of a node must be *at least as great as the priority of its children*
- This means that the *effective order of insertion* into the tree is *random*
- Since a randomly-constructed binary search tree has an $\mathcal{O}(\log n)$ expected search length—



Expected Search Length in Randomly Constructed Binary Search Trees

We want to find node n. How many ancestors is n expected to have?

Expected Search Length in Randomly Constructed Binary Search Trees

We want to find node n. How many ancestors is n expected to have?

- Node m is an ancestor of n if:
  - It was inserted before n
  - It was inserted before any node with a value *between* node m's value and node n's value

Expected Search Length in Randomly Constructed Binary Search Trees

We want to find node n. How many ancestors is n expected to have?

- Node m is an ancestor of n if:
  - It was inserted before n
  - It was inserted before any node with a value *between* node m's value and node n's value
- So the nodes with next and previous values to n have a $\frac{1}{2}$ probability of being its ancestor…
- The next nodes out have a $\frac{1}{3}$ probability… and so on

Expected Search Length in Randomly Constructed Binary Search Trees

We want to find node n. How many ancestors is n expected to have?

- Node m is an ancestor of n if:
  - It was inserted before n
  - It was inserted before any node with a value *between* node m's value and node n's value
- So the nodes with next and previous values to n have a $\frac{1}{2}$ probability of being its ancestor…
- The next nodes out have a $\frac{1}{3}$ probability… and so on
- Then the upper bound for the expected value is the sum of two harmonic numbers H_k with $k < n$

Expected Search Length in Randomly Constructed Binary Search Trees

We want to find node n. How many ancestors is n expected to have?

- Node m is an ancestor of n if:
  - It was inserted before n
  - It was inserted before any node with a value *between* node m's value and node n's value
- So the nodes with next and previous values to n have a $\frac{1}{2}$ probability of being its ancestor…
- The next nodes out have a $\frac{1}{3}$ probability… and so on
- Then the upper bound for the expected value is the sum of two harmonic numbers H_k with $k < n$
- That's less than about $2(\gamma + \ln n)$…

Expected Search Length in Randomly Constructed Binary Search Trees

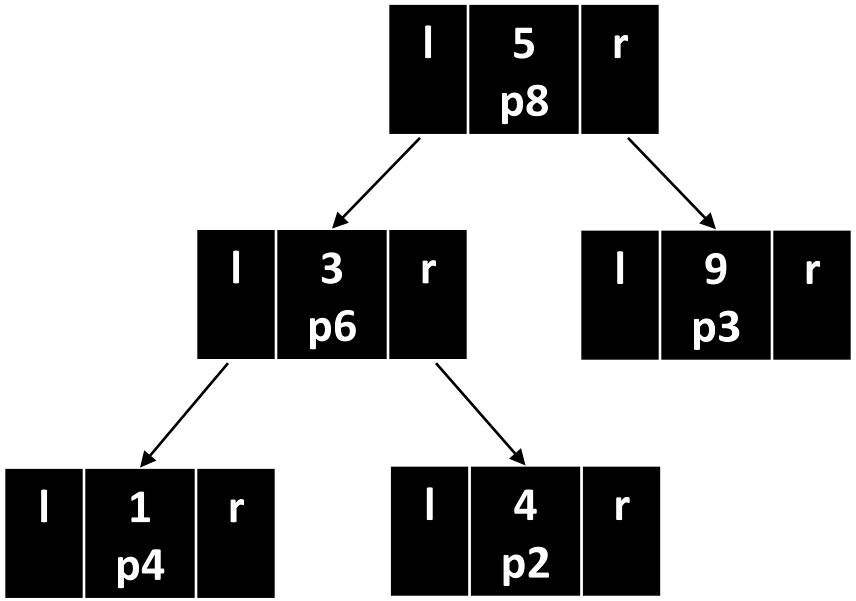
We want to find node n. How many ancestors is n expected to have?

- Node m is an ancestor of n if:
  - It was inserted before n
  - It was inserted before any node with a value *between* node m's value and node n's value
- So the nodes with next and previous values to n have a $\frac{1}{2}$ probability of being its ancestor…
- The next nodes out have a $\frac{1}{3}$ probability… and so on
- Then the upper bound for the expected value is the sum of two harmonic numbers H_k with $k < n$
- That's less than about $2(\gamma + \ln n)$…
- Which is $\mathcal{O}(\log n)$

Treaps

A *treap* (tree-heap) has an additional property:

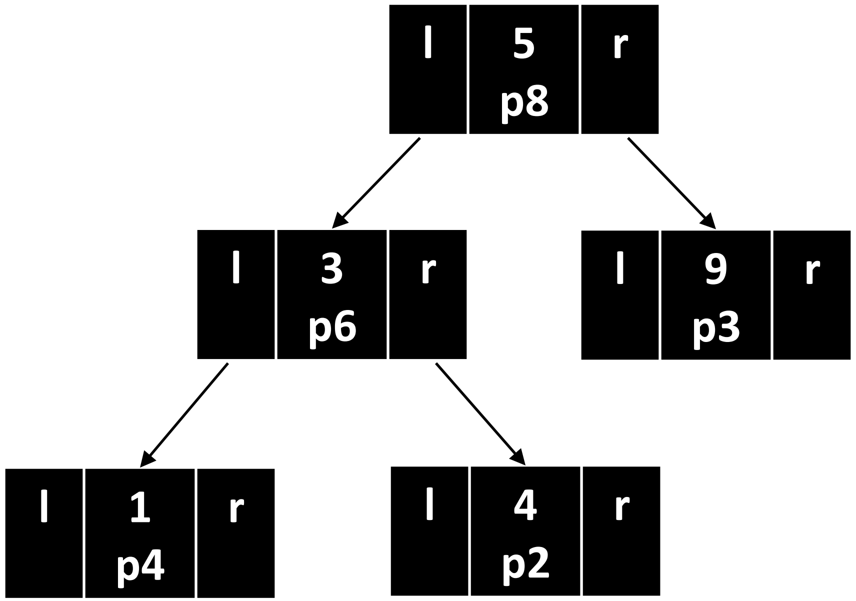
- Each node has an additional value, called *priority*
- The priority of a node must be *at least as great as the priority of its children*
- This means that the *effective order of insertion* into the tree is *random*
- Since a randomly-constructed binary search tree has an $\mathcal{O}(\log n)$ expected search length—so does a treap



Treaps

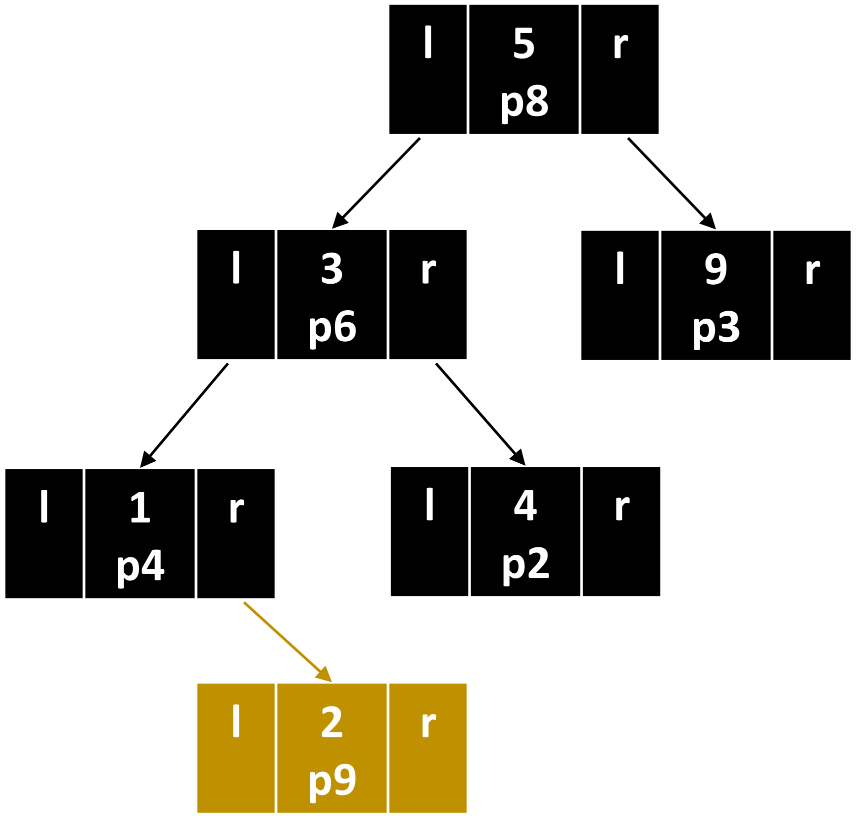
A *treap* (tree-heap) has an additional property:

- Each node has an additional value, called *priority*
- The priority of a node must be *at least as great as the priority of its children*
- This means that the *effective order of insertion* into the tree is *random*
- Since a randomly-constructed binary search tree has an $\mathcal{O}(\log n)$ expected search length—so does a treap
- So far, so good – but how do we maintain the heap property?



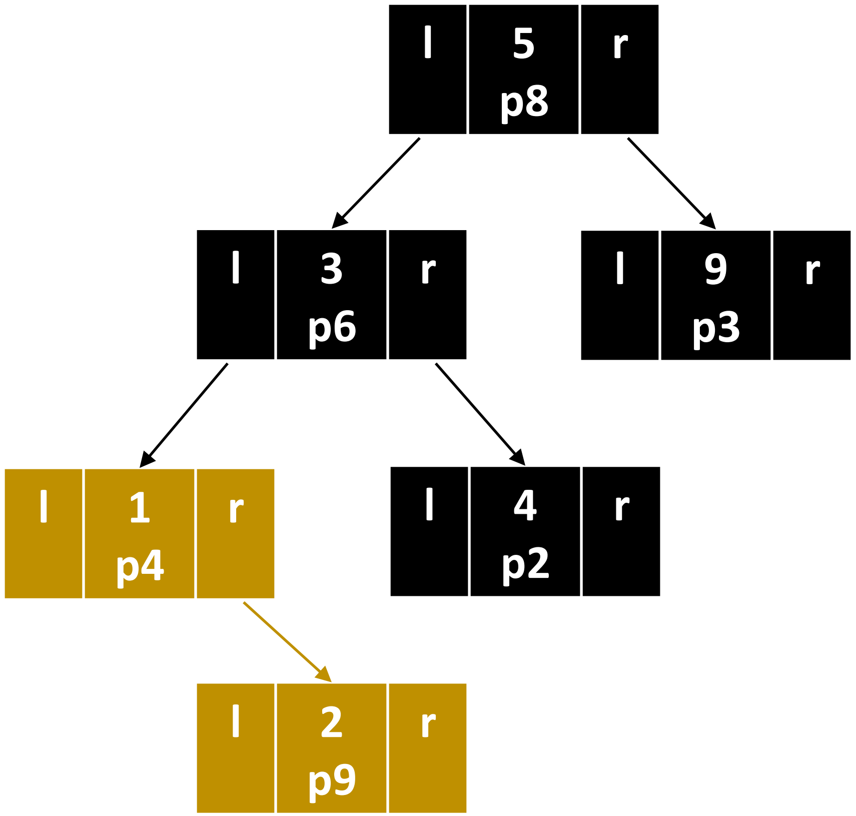
Insertion Into Treaps

- First, insert as though into a BST



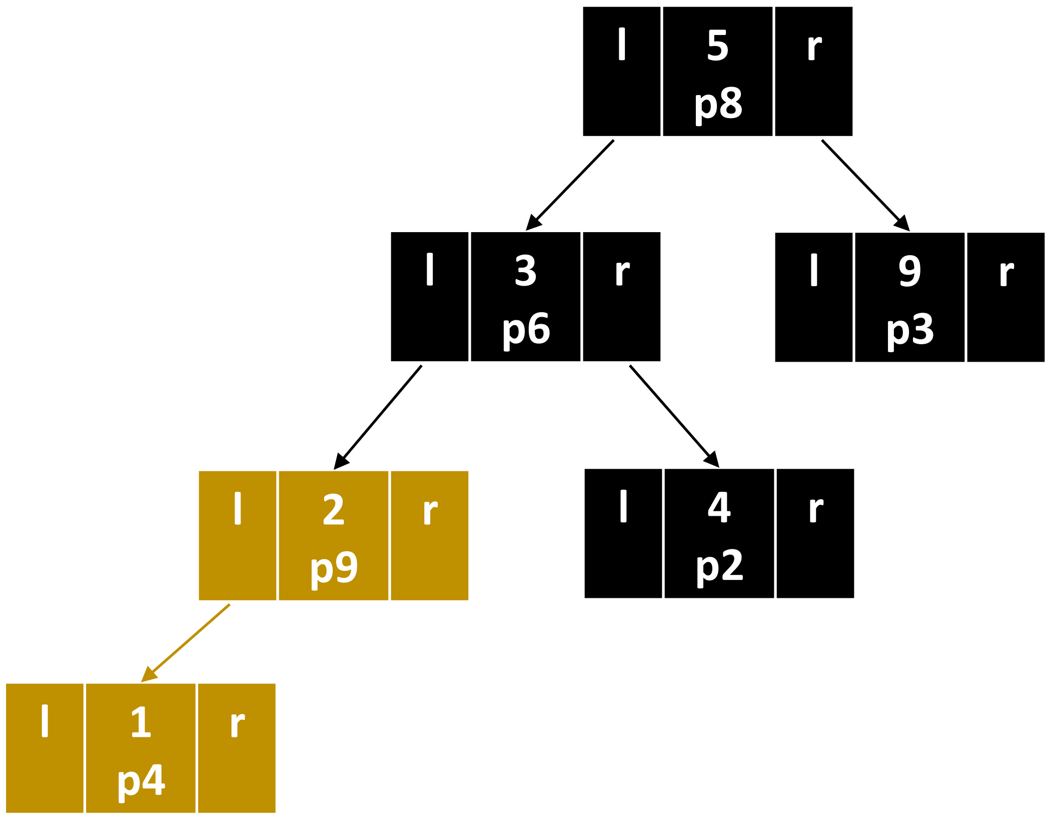
Insertion Into Treaps

- First, insert as though into a BST
- Then rotate (similarly to an AVL…)



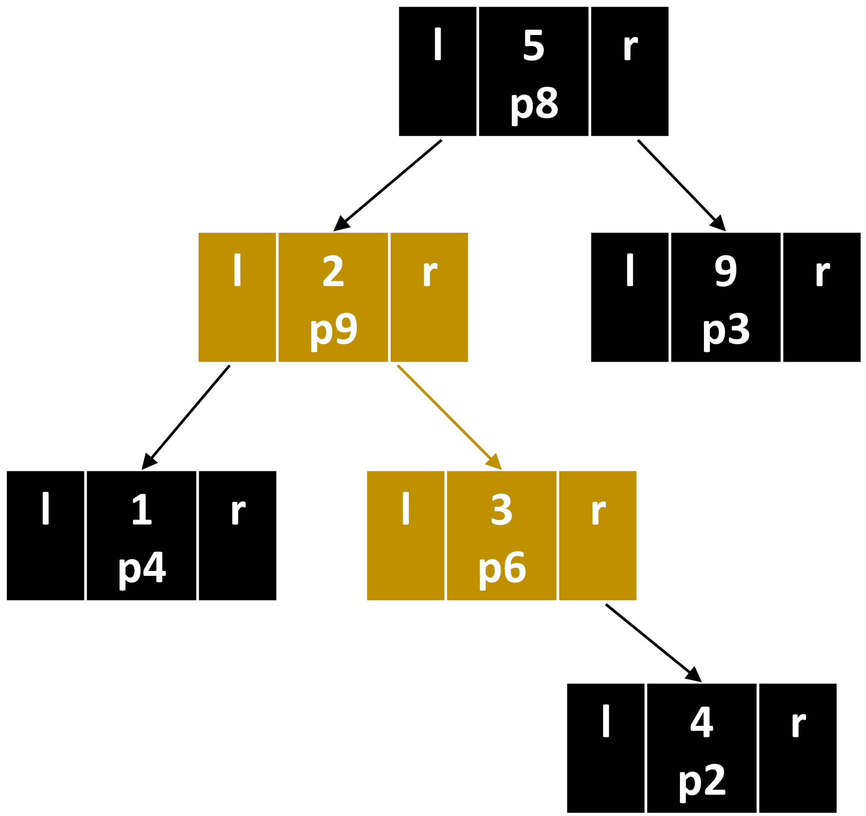
Insertion Into Treaps

- First, insert as though into a BST
- Then rotate (similarly to an AVL…)



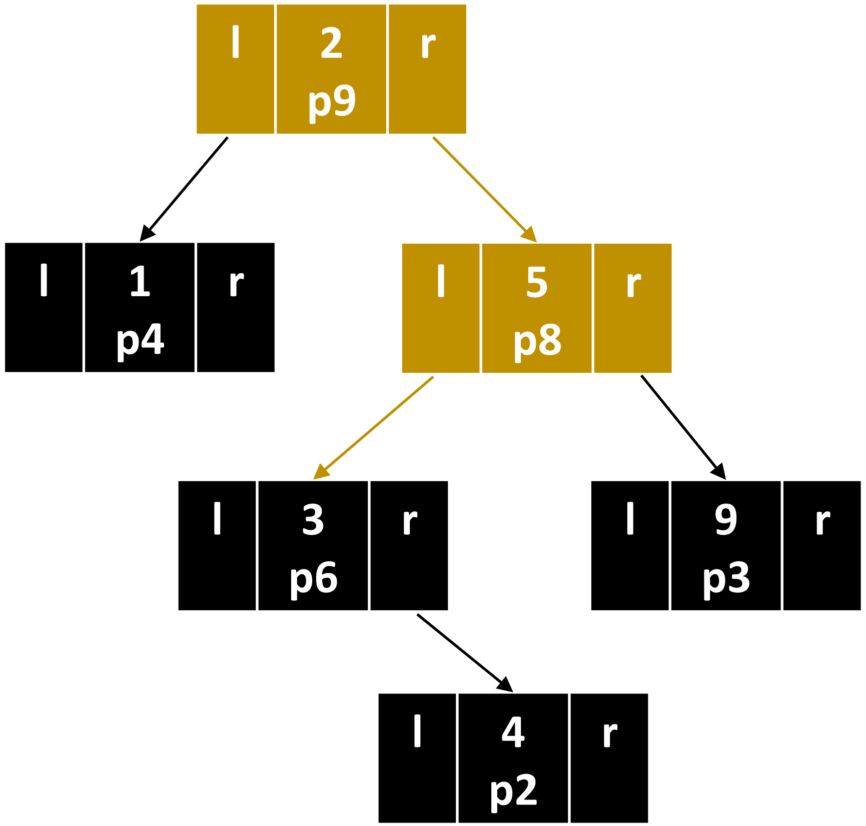
Insertion Into Treaps

- First, insert as though into a BST
- Then rotate (similarly to an AVL…)



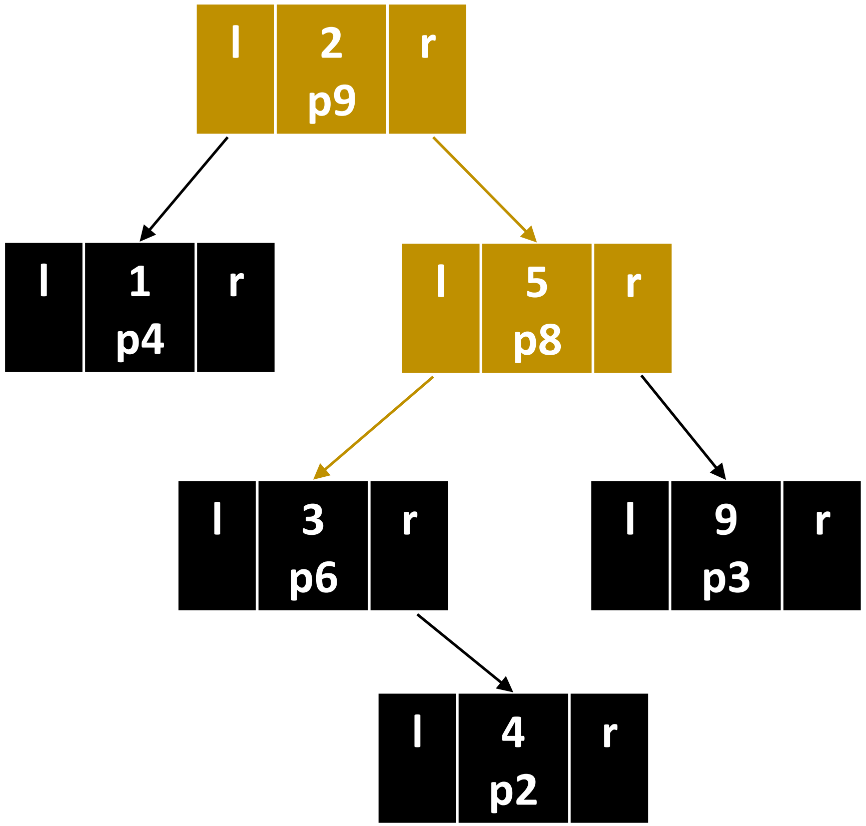
Insertion Into Treaps

- First, insert as though into a BST
- Then rotate (similarly to an AVL…)
- …until it's where it needs to be



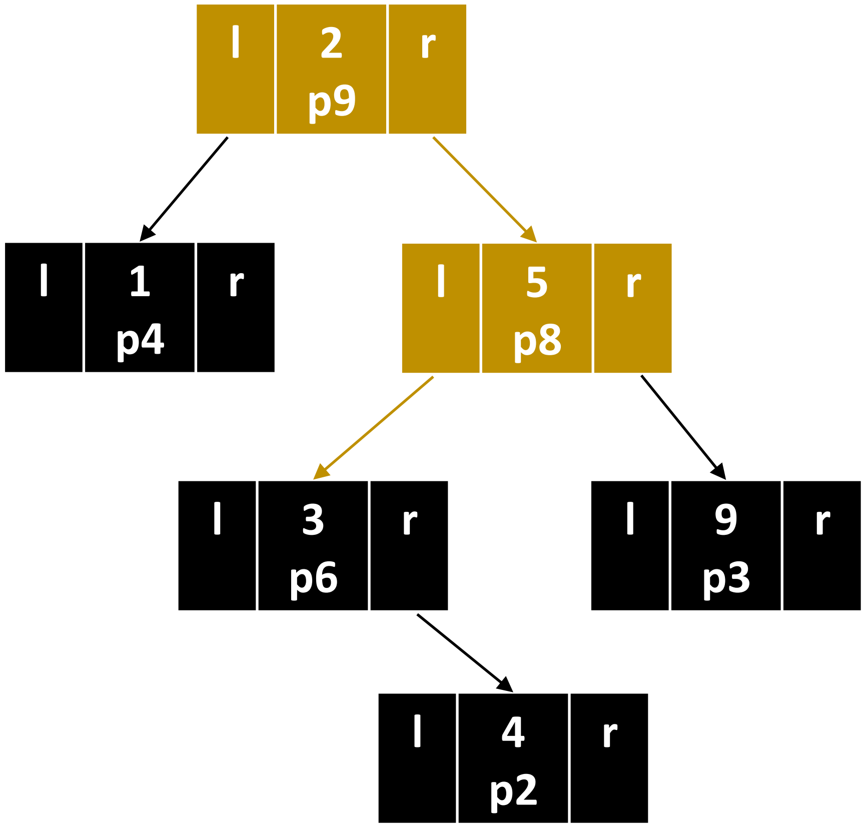
Insertion Into Treaps

- First, insert as though into a BST
- Then rotate (similarly to an AVL…)
- …until it's where it needs to be
- Notice that this *does not* guarantee a perfect result…
- Just, as per the expected value calculations, makes a good one very likely



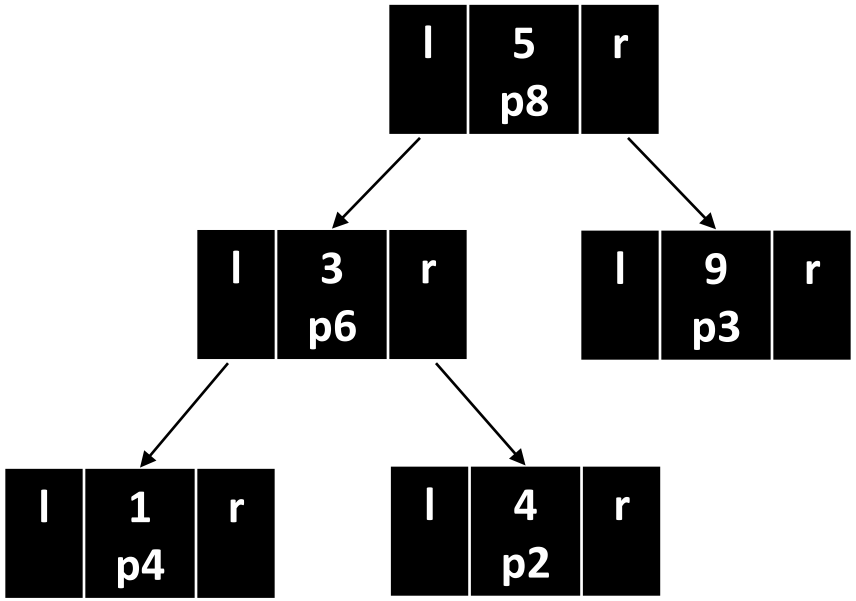
Insertion Into Treaps

- First, insert as though into a BST
- Then rotate (similarly to an AVL…)
- …until it's where it needs to be
- Notice that this *does not* guarantee a perfect result…
- Just, as per the expected value calculations, makes a good one very likely
- The rotation is maximum $\mathcal{O}(h)$, which—again, *on average*—is $\mathcal{O}(\log n)$



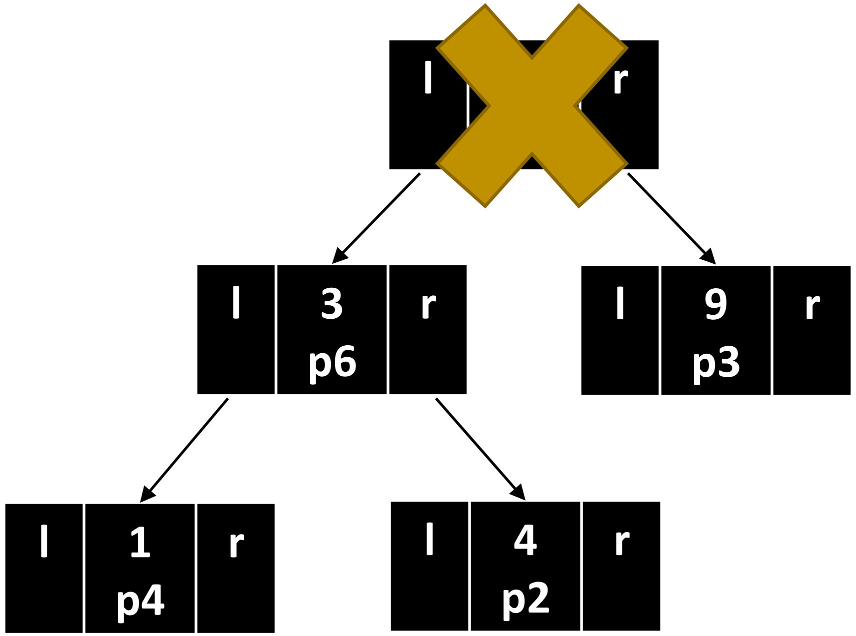
Deletion from Treaps

- As always, leaves and nodes with one child are easy
- If a node has two children, replace it with its adjacent successor or predecessor…



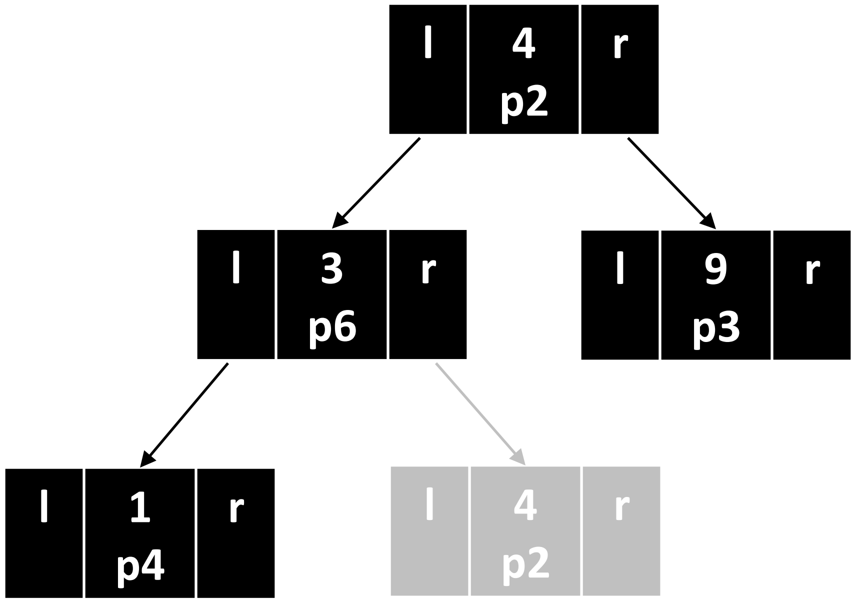
Deletion from Treaps

- As always, leaves and nodes with one child are easy
- If a node has two children, replace it with its adjacent successor or predecessor…



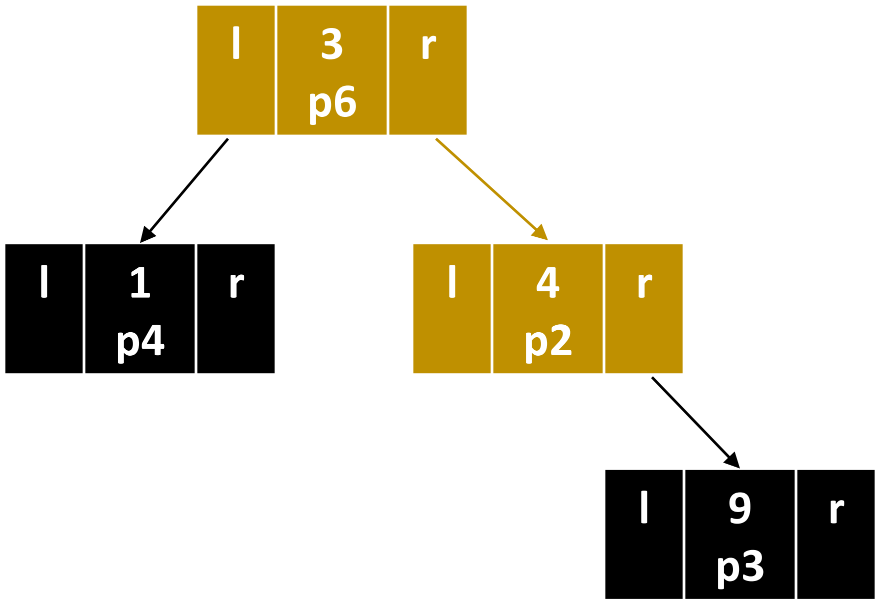
Deletion from Treaps

- As always, leaves and nodes with one child are easy
- If a node has two children, replace it with its adjacent successor or predecessor…



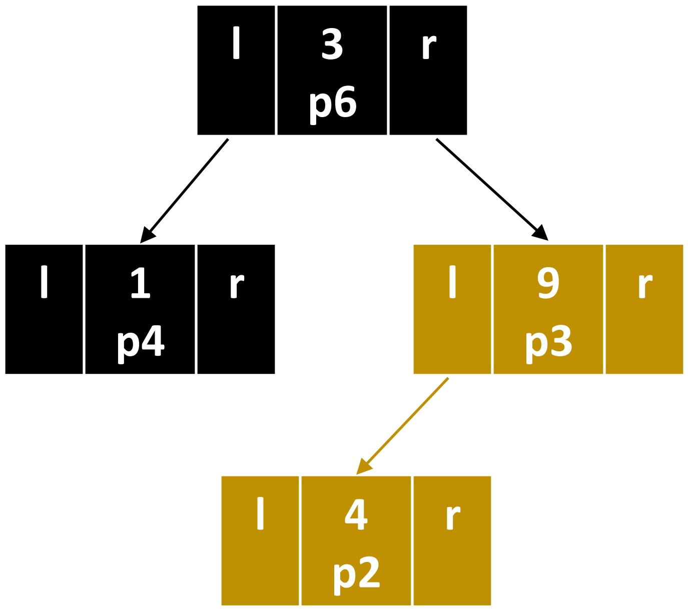
Deletion from Treaps

- As always, leaves and nodes with one child are easy
- If a node has two children, replace it with its adjacent successor or predecessor…
- …and, again, rotate it until it's in the right place



Deletion from Treaps

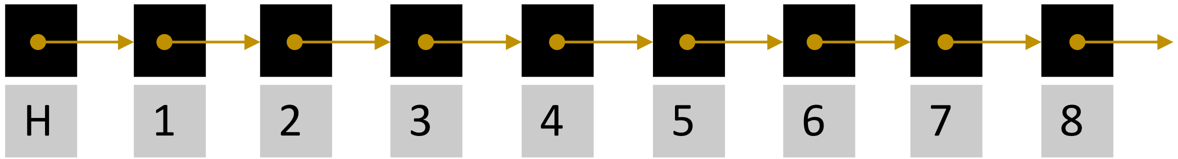
- As always, leaves and nodes with one child are easy
- If a node has two children, replace it with its adjacent successor or predecessor…
- …and, again, rotate it until it's in the right place



Skip Lists

Skip Lists

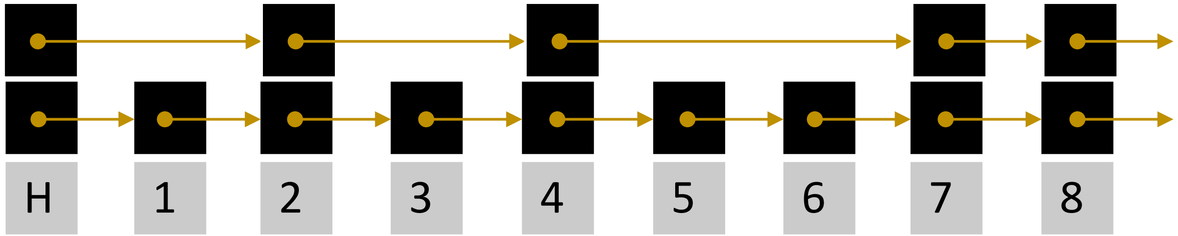
A skip list starts as an ordinary linked list…



Skip Lists

A skip list starts as an ordinary linked list…

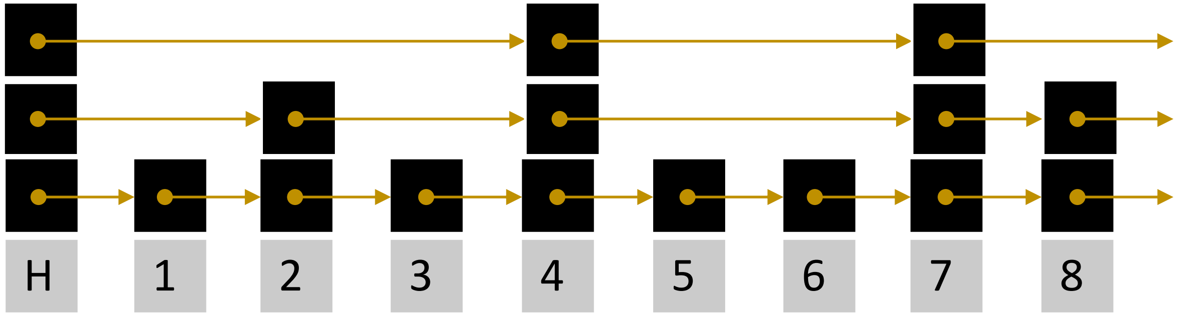
- But adds additional layers of pointers *on top* of it



Skip Lists

A skip list starts as an ordinary linked list…

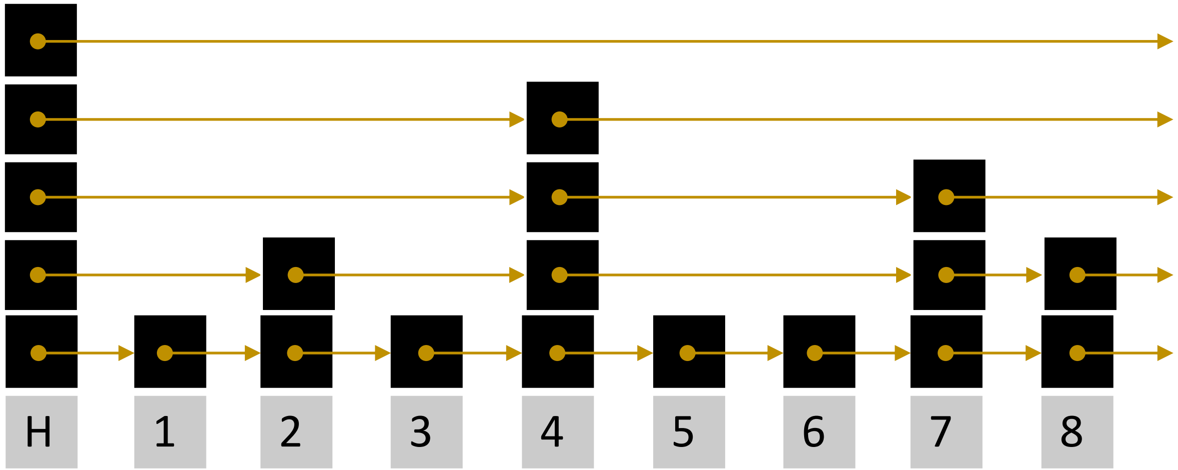
- But adds additional layers of pointers *on top* of it
- Each layer omits a certain number of the pointers from the previous later (usually about half)…



Skip Lists

A skip list starts as an ordinary linked list…

- But adds additional layers of pointers *on top* of it
- Each layer omits a certain number of the pointers from the previous later (usually about half)…
- …until all that's left is the head element
- Only needs about twice as many pointers as we had in the first place



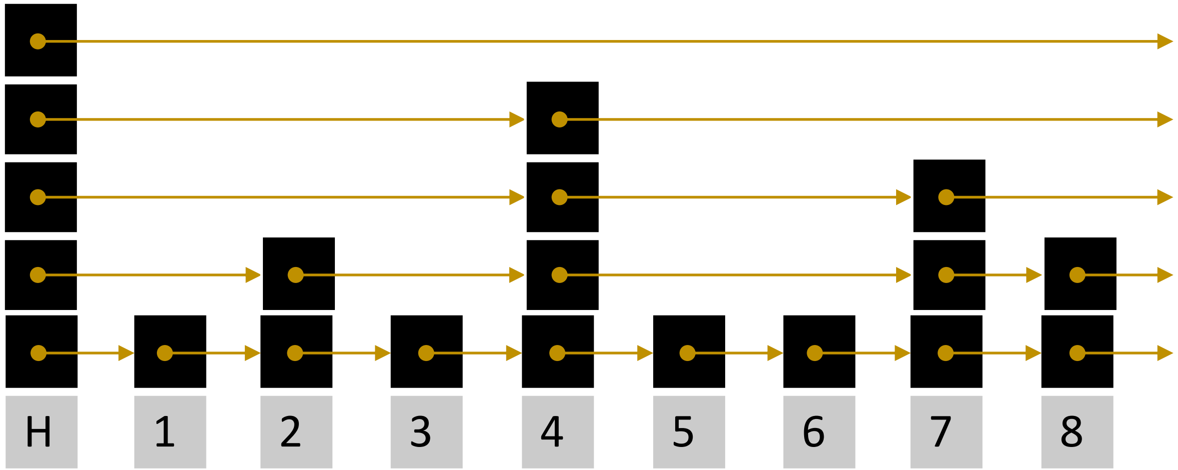
Searching Skip Lists

Searching for value v is easy:

- Let n be the node at the left of the top level
- Repeat
  - If $n.v = v$, succeed
  - If $n.v < v$, go right
  - If $n.v > v$ or $!n$, go left then down
  - If we've fallen off the bottom, fail

Obviously $\mathcal{O}(\log n)$ on average…

- …as long as we keep our cut-by-half property

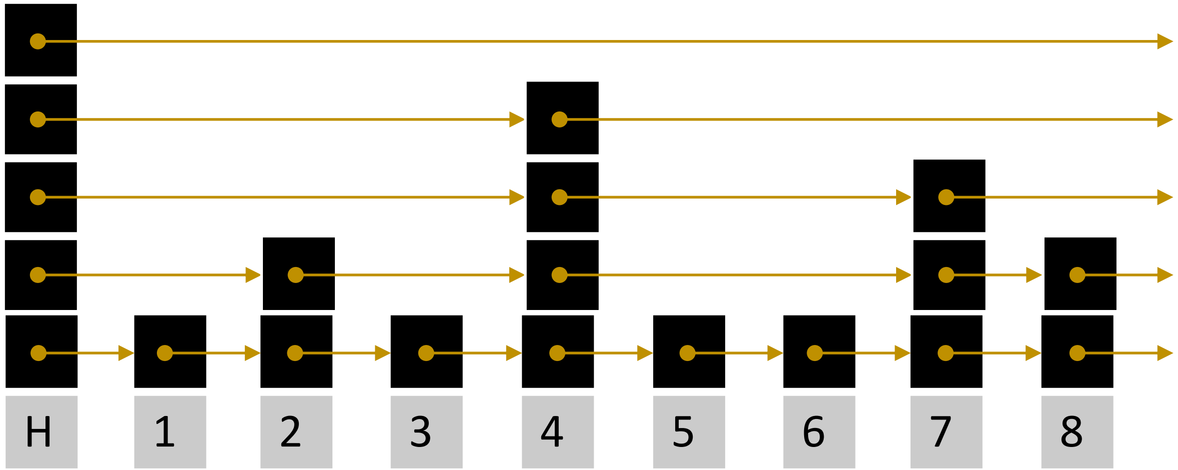


Insertion and Deletion

Insertion and deletion are *also* easy

Insertion:
- Find the spot the node should go
- Randomly choose the height of the node
- Insert it into the linked lists up to that level, as you normally would
- One search, plus $\log n$ insertions - $\mathcal{O}(\log n)$

Deletion:
- Find the node
- Delete it from all the linked lists it's in, as you normally would
- Dispose of the value
- One search, plus $\log n$ deletions - $\mathcal{O}(\log n)$



Re-Optimizing a Skip List

- We *could* theoretically end up deleting too many tall or short nodes
- We can fix this by just going through the list and re-balancing the height of each node
- Can use straight logarithmic boundaries if it's an internal structure…
- Or randomize it if we're worried about adversarial values
- $\mathcal{O}(n)$, but a fast $\mathcal{O}(n)$, since we're just doing *real* simple pointer operations
- Can attach this to list traversals that happen naturally (load/save, print, etc.)