COMP 761: Lecture 28 – Binary Search Trees I

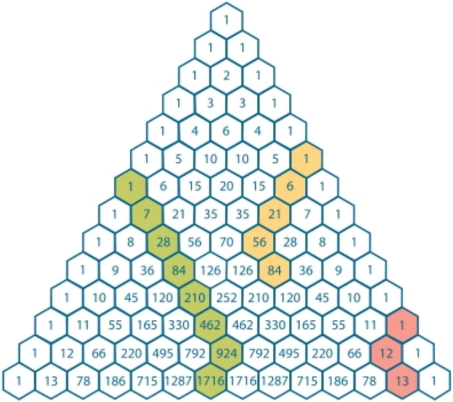
David Rolnick

November 9, 2020

Problem

Prove the Hockey Stick Identity:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$



Course Announcements

Course Announcements

- Office hours right after class.

Course Announcements

- Office hours right after class.
- Problem Set 3 grades out, let Vincent and me know if you think something should be reconsidered.

Binary search trees

Binary search trees

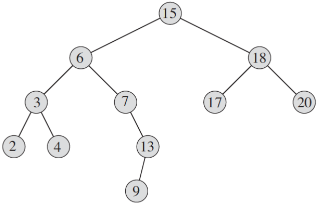
- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.

Binary search trees

- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.
- Likewise for the *right subtree*.

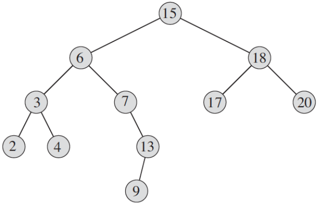
Binary search trees

- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.
- Likewise for the *right subtree*.
- A *binary search tree* is a binary tree, each node storing a *key*.



Binary search trees

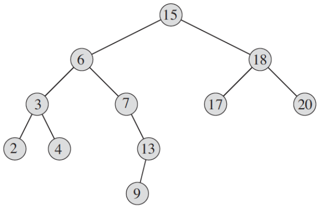
- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.
- Likewise for the *right subtree*.
- A *binary search tree* is a binary tree, each node storing a *key*.



- We require that for every node v:

Binary search trees

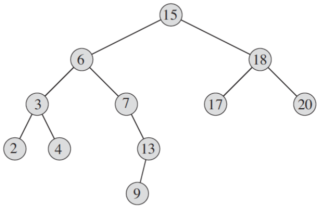
- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.
- Likewise for the *right subtree*.
- A *binary search tree* is a binary tree, each node storing a *key*.



- We require that for every node v:
  - The left subtree has all nodes less than or equal to v.

Binary search trees

- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.
- Likewise for the *right subtree*.
- A *binary search tree* is a binary tree, each node storing a *key*.



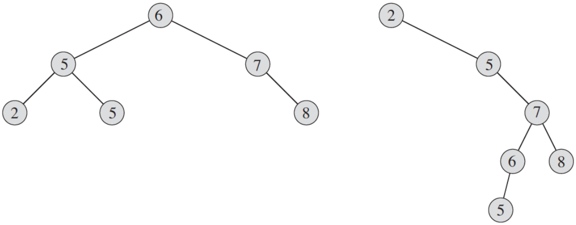
- We require that for every node v:
  - The left subtree has all nodes less than or equal to v.
  - The right subtree has all nodes greater than or equal to v.

Binary search trees

- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.
- Likewise for the *right subtree*.
- A *binary search tree* is a binary tree, each node storing a *key*.
- We require that for every node v:
  - The left subtree has all nodes less than or equal to v.
  - The right subtree has all nodes greater than or equal to v.
- Can there be more than one binary search tree for a given set of keys?

Binary search trees

- In a binary tree, we say the *left subtree* of a node v is the left child (if it exists) and the rest of the subtree rooted at the left child.
- Likewise for the *right subtree*.
- A *binary search tree* is a binary tree, each node storing a *key*.
- We require that for every node v:
  - The left subtree has all nodes less than or equal to v.
  - The right subtree has all nodes greater than or equal to v.
- Can there be more than one binary search tree for a given set of keys?
- Yes!



(a) (b)

Binary search trees

Binary search trees

- In storing binary search trees, we generally store at each node v:
  - The key
  - Pointers to the left and right children (or null if they don't exist)
  - Pointer to the parent

Binary search trees

- In storing binary search trees, we generally store at each node v:
  - The key
  - Pointers to the left and right children (or null if they don't exist)
  - Pointer to the parent
- This allows us to move around the tree easily.

Binary search trees

- In storing binary search trees, we generally store at each node v:
  - The key
  - Pointers to the left and right children (or null if they don't exist)
  - Pointer to the parent
- This allows us to move around the tree easily.
- We will want the following operations within a binary search tree:

Binary search trees

- In storing binary search trees, we generally store at each node v:
  - The key
  - Pointers to the left and right children (or null if they don't exist)
  - Pointer to the parent
- This allows us to move around the tree easily.
- We will want the following operations within a binary search tree:
  - Search (find if a given key is in the tree)

Binary search trees

- In storing binary search trees, we generally store at each node v:
  - The key
  - Pointers to the left and right children (or null if they don't exist)
  - Pointer to the parent
- This allows us to move around the tree easily.
- We will want the following operations within a binary search tree:
  - Search (find if a given key is in the tree)
  - Maximum and minimum (find the max/min keys)

Binary search trees

- In storing binary search trees, we generally store at each node v:
  - The key
  - Pointers to the left and right children (or null if they don't exist)
  - Pointer to the parent
- This allows us to move around the tree easily.
- We will want the following operations within a binary search tree:
  - Search (find if a given key is in the tree)
  - Maximum and minimum (find the max/min keys)
  - Successor and predecessor (given a key in the tree, find the keys immediately greater and less than it)

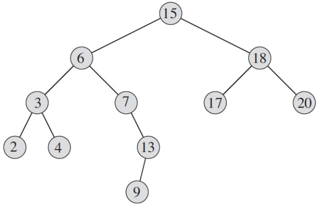
Binary search trees

- In storing binary search trees, we generally store at each node v:
  - The key
  - Pointers to the left and right children (or null if they don't exist)
  - Pointer to the parent
- This allows us to move around the tree easily.
- We will want the following operations within a binary search tree:
  - Search (find if a given key is in the tree)
  - Maximum and minimum (find the max/min keys)
  - Successor and predecessor (given a key in the tree, find the keys immediately greater and less than it)
  - Insert and delete (add or remove a new key)

Search

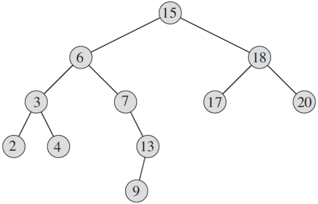
Search

- Suppose we are given a value k and a binary search tree.



Search

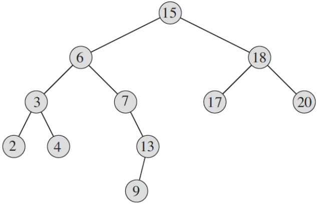
- Suppose we are given a value k and a binary search tree.



- How can we check if k is stored in the tree?

Search

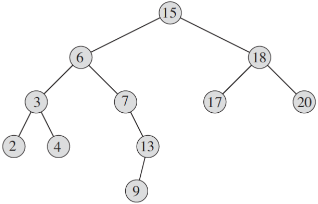
- Suppose we are given a value k and a binary search tree.



- How can we check if k is stored in the tree?
- If the root has key k_x, go left if $k < k_x$ and go right if $k > k_x$.

Search

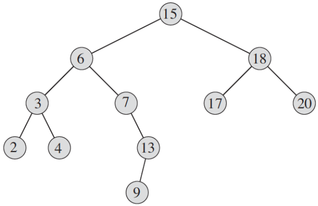
- Suppose we are given a value k and a binary search tree.



- How can we check if k is stored in the tree?
- If the root has key k_x, go left if $k < k_x$ and go right if $k > k_x$.
- Continue, if at key k_y, go left if $k < k_y$ and go right if $k > k_y$.

Search

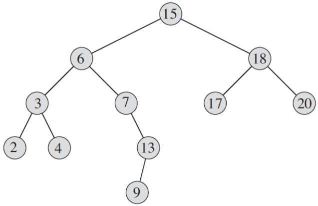
- Suppose we are given a value k and a binary search tree.



- How can we check if k is stored in the tree?
- If the root has key k_x, go left if $k < k_x$ and go right if $k > k_x$.
- Continue, if at key k_y, go left if $k < k_y$ and go right if $k > k_y$.
- Stop if ever have key = k or if no left/right child to move to.

Search

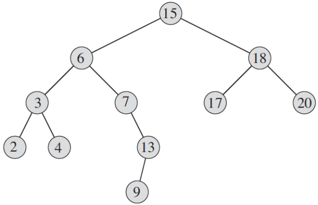
- Suppose we are given a value k and a binary search tree.



- How can we check if k is stored in the tree?
- If the root has key k_x, go left if $k < k_x$ and go right if $k > k_x$.
- Continue, if at key k_y, go left if $k < k_y$ and go right if $k > k_y$.
- Stop if ever have key = k or if no left/right child to move to.
- How long does this take?

Search

- Suppose we are given a value k and a binary search tree.

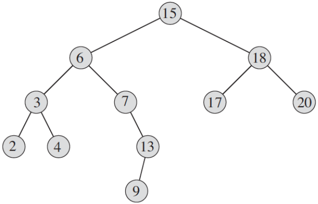


- How can we check if k is stored in the tree?
- If the root has key k_x, go left if $k < k_x$ and go right if $k > k_x$.
- Continue, if at key k_y, go left if $k < k_y$ and go right if $k > k_y$.
- Stop if ever have key = k or if no left/right child to move to.
- How long does this take?
- The time is $O(h)$, where h is the *height* of the tree (=maximum depth of all nodes).

Maximum and minimum

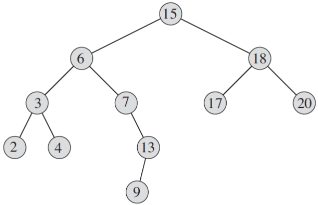
Maximum and minimum

- How to find the max key in the tree?



Maximum and minimum

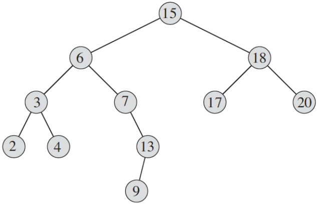
- How to find the max key in the tree?



- Keep going right in the tree until not possible anymore.

Maximum and minimum

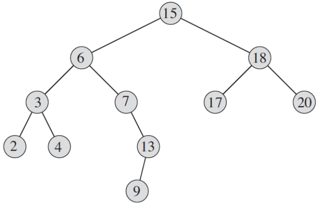
- How to find the max key in the tree?



- Keep going right in the tree until not possible anymore.
- Similarly with the min, go left.

Maximum and minimum

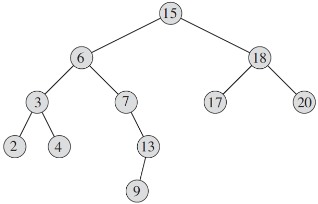
- How to find the max key in the tree?



- Keep going right in the tree until not possible anymore.
- Similarly with the min, go left.
- How long does this take?

Maximum and minimum

- How to find the max key in the tree?

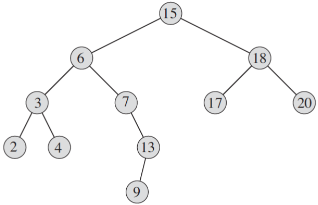


- Keep going right in the tree until not possible anymore.
- Similarly with the min, go left.
- How long does this take?
- Again, time is $O(h)$, where h is the height.

Successor and predecessor

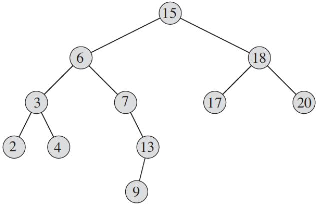
Successor and predecessor

- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



Successor and predecessor

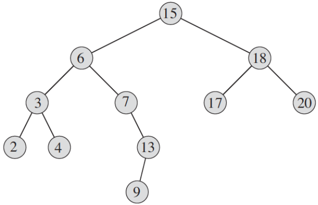
- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- Any two nodes in the tree have a lowest common ancestor (LCA) = the deepest node in the tree that is an ancestor to both.

Successor and predecessor

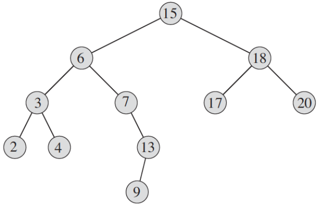
- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- Any two nodes in the tree have a lowest common ancestor (LCA) = the deepest node in the tree that is an ancestor to both.
- Let x be the LCA for v and w.

Successor and predecessor

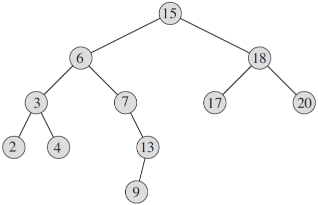
- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- Any two nodes in the tree have a lowest common ancestor (LCA) = the deepest node in the tree that is an ancestor to both.
- Let x be the LCA for v and w.
- v and w must be on different subtrees from x since it is deepest.

Successor and predecessor

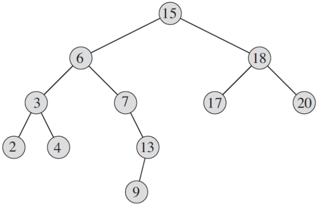
- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- Any two nodes in the tree have a lowest common ancestor (LCA) = the deepest node in the tree that is an ancestor to both.
- Let x be the LCA for v and w.
- v and w must be on different subtrees from x since it is deepest.
- Is v on the left or the right subtree?

Successor and predecessor

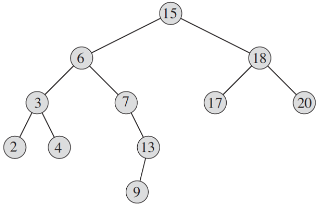
- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- Any two nodes in the tree have a lowest common ancestor (LCA) = the deepest node in the tree that is an ancestor to both.
- Let x be the LCA for v and w.
- v and w must be on different subtrees from x since it is deepest.
- Is v on the left or the right subtree?
- v must be on the left subtree, w on the right, since $k_v < k_w$.

Successor and predecessor

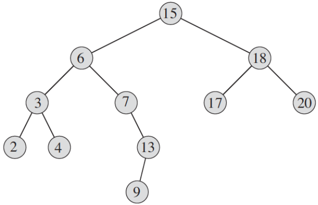
- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- Any two nodes in the tree have a lowest common ancestor (LCA) = the deepest node in the tree that is an ancestor to both.
- Let x be the LCA for v and w.
- v and w must be on different subtrees from x since it is deepest.
- Is v on the left or the right subtree?
- v must be on the left subtree, w on the right, since $k_v < k_w$.
- But then $k_v < k_x < k_w$, so w isn't the successor to v....unless what?

Successor and predecessor

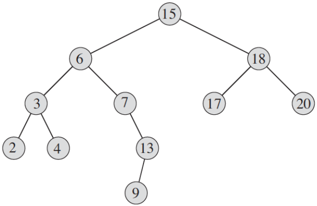
- Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- Any two nodes in the tree have a lowest common ancestor (LCA) = the deepest node in the tree that is an ancestor to both.
- Let x be the LCA for v and w.
- v and w must be on different subtrees from x since it is deepest.
- Is v on the left or the right subtree?
- v must be on the left subtree, w on the right, since $k_v < k_w$.
- But then $k_v < k_x < k_w$, so w isn't the successor to v....unless what?
- Unless x equals v or w.

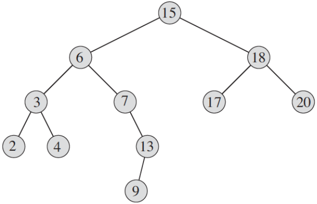
Successor and predecessor

Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



Successor and predecessor

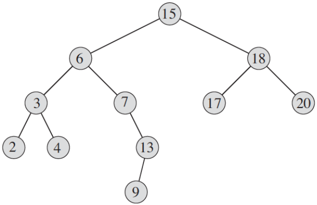
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.

Successor and predecessor

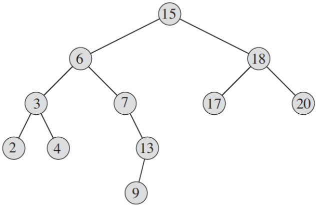
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- What is w here?

Successor and predecessor

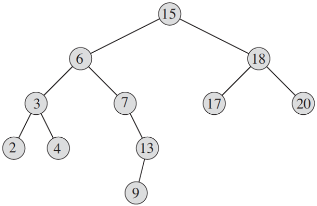
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.

Successor and predecessor

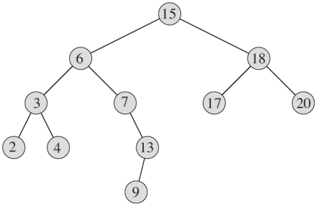
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.
- **Case 2:** x equals w, so w is ancestor to v.

Successor and predecessor

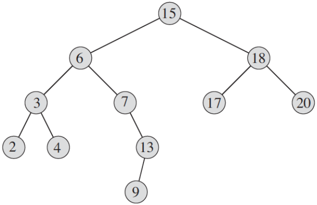
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.
- **Case 2:** x equals w, so w is ancestor to v.
- What is w here?

Successor and predecessor

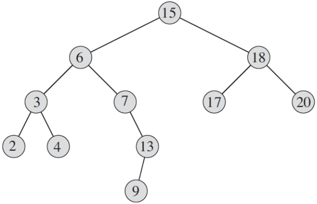
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.
- **Case 2:** x equals w, so w is ancestor to v.
- w must be the smallest ancestor of v with v on its left subtree.

Successor and predecessor

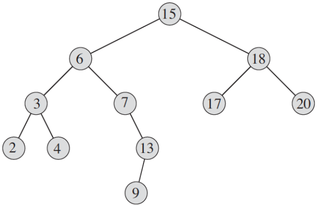
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.
- **Case 2:** x equals w, so w is ancestor to v.
- w must be the smallest ancestor of v with v on its left subtree.
- How do we know which case we are in?

Successor and predecessor

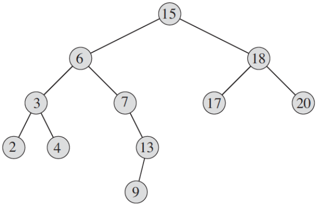
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.
- **Case 2:** x equals w, so w is ancestor to v.
- w must be the smallest ancestor of v with v on its left subtree.
- How do we know which case we are in?
- If v has a right child, then Case 1, otherwise Case 2.

Successor and predecessor

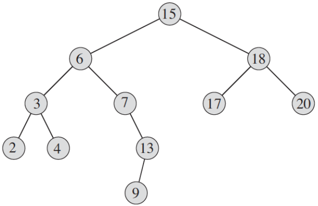
Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.
- **Case 2:** x equals w, so w is ancestor to v.
- w must be the smallest ancestor of v with v on its left subtree.
- How do we know which case we are in?
- If v has a right child, then Case 1, otherwise Case 2.
- Again, we have an algorithm in time $O(h)$.

Successor and predecessor

Given a node v with key k_v, how to find the *successor* w to v, i.e. with the smallest k_w that is larger than k_v? (Let's assume here all keys are different.)



- **Case 1:** x equals v, so v is ancestor to w.
- w must be the minimum of the right subtree of v.
- **Case 2:** x equals w, so w is ancestor to v.
- w must be the smallest ancestor of v with v on its left subtree.
- How do we know which case we are in?
- If v has a right child, then Case 1, otherwise Case 2.
- Again, we have an algorithm in time $O(h)$.
- Similar process for finding the predecessor.

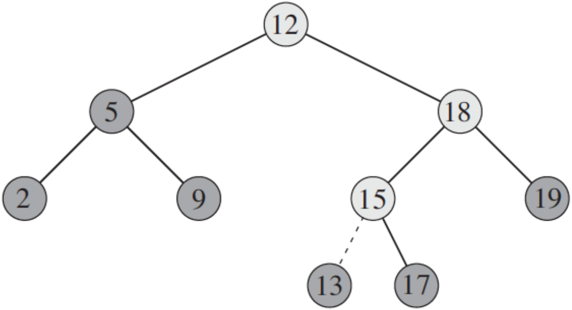
Insert

Insert

- How can we insert a new key k into a binary search tree?

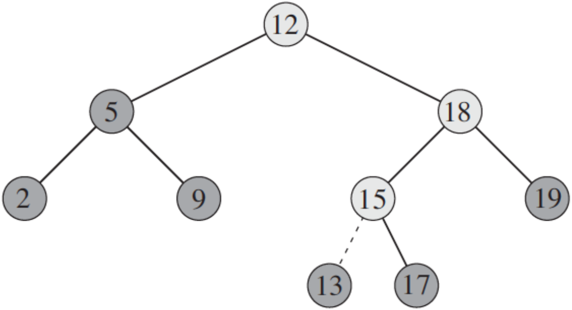
Insert

- How can we insert a new key k into a binary search tree?
- We can search for where it would be if it were there and then add it:



Insert

- How can we insert a new key *k* into a binary search tree?
- We can search for where it would be if it were there and then add it:



- This takes $O(h)$ time again.

Delete

Delete

- Let's try deleting a node z in the tree.

Delete

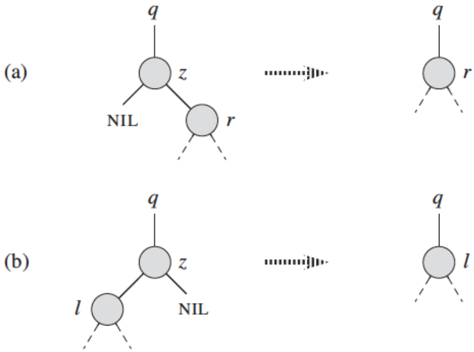
- Let's try deleting a node z in the tree.
- First suppose the node doesn't have a left child. What do we do?

Delete

- Let's try deleting a node z in the tree.
- First suppose the node doesn't have a left child. What do we do?
- We can just move the right subtree up, with its root taking z's place.

Delete

- Let's try deleting a node z in the tree.
- First suppose the node doesn't have a left child. What do we do?
- We can just move the right subtree up, with its root taking z's place.
- Similarly if z is missing a right child.



Delete

Delete

- If z has both left and right subtrees, we can try replacing it by a node in the right subtree.

Delete

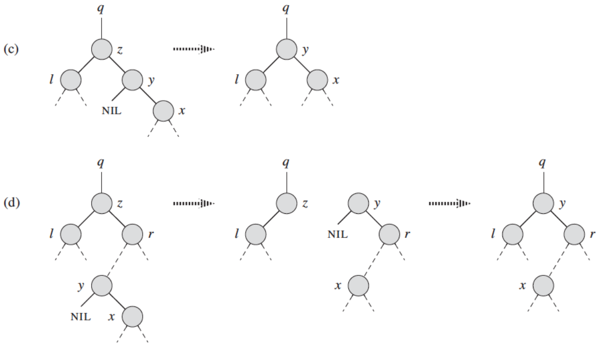
- If z has both left and right subtrees, we can try replacing it by a node in the right subtree.
- Which one do we need to pick?

Delete

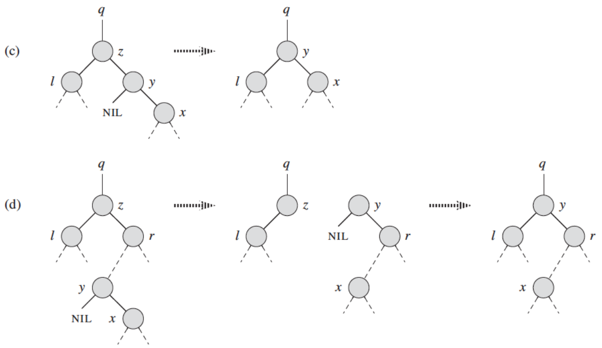
- If z has both left and right subtrees, we can try replacing it by a node in the right subtree.
- Which one do we need to pick?
- We want to pick the minimum in the right subtree, since everything in the right subtree has keys $\geq$ the key for z.

Delete

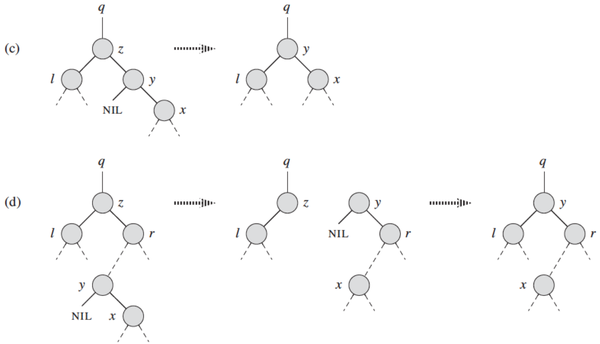
- If z has both left and right subtrees, we can try replacing it by a node in the right subtree.
- Which one do we need to pick?
- We want to pick the minimum in the right subtree, since everything in the right subtree has keys $\geq$ the key for z.
- Here is how we can do that:



Delete

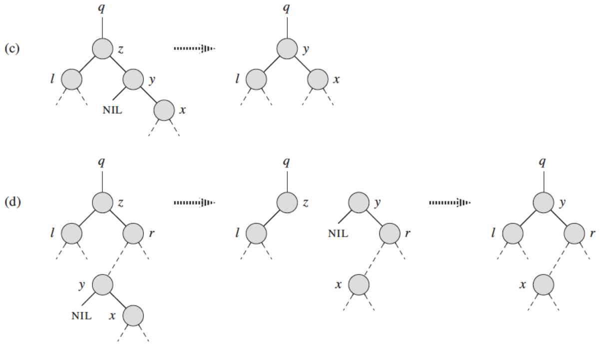


Delete



- How long does this process take?

Delete



- How long does this process take?
- As before, we might have to go all the way down the tree, so $O(h)$.

Expected height

Expected height

- We have a lot of algorithms running in $O(h)$.

Expected height

- We have a lot of algorithms running in $O(h)$.
- What is the maximum height with n keys?

Expected height

- We have a lot of algorithms running in $O(h)$.
- What is the maximum height with n keys?
- Worst case, $h = n - 1$.

Expected height

- We have a lot of algorithms running in $O(h)$.
- What is the maximum height with n keys?
- Worst case, $h = n - 1$.
- What is the minimum height with n keys?

Expected height

- We have a lot of algorithms running in $O(h)$.
- What is the maximum height with n keys?
- Worst case, $h = n - 1$.
- What is the minimum height with n keys?
- Best case, $h = O(\log n)$.

Expected height

- We have a lot of algorithms running in $O(h)$.
- What is the maximum height with n keys?
- Worst case, $h = n - 1$.
- What is the minimum height with n keys?
- Best case, $h = O(\log n)$.
- Let's consider a *typical* binary search tree.

Expected height

- We have a lot of algorithms running in $O(h)$.
- What is the maximum height with n keys?
- Worst case, $h = n - 1$.
- What is the minimum height with n keys?
- Best case, $h = O(\log n)$.
- Let's consider a *typical* binary search tree.
- Suppose that we insert $\{1, 2, \ldots, n\}$ into a binary search tree in random order. What is the expected height?

Hockey stick identity

Hockey stick identity

- We will use the hockey stick identity in our proof:

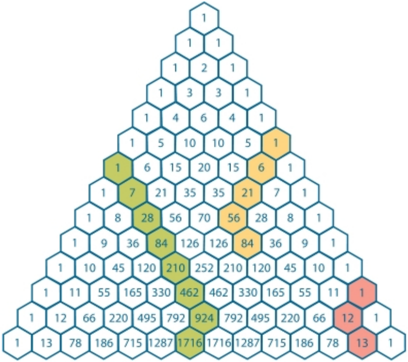
$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- How can we prove this?



Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- Let's use induction to prove it.

Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- Let's use induction to prove it.
- What is a good base case?

Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- Let's use induction to prove it.
- We can use $n = 1$ as a base case:

$$\sum_{i=0}^{0} \binom{i+k}{k} = \binom{k}{k} = \binom{k+1}{k+1}.$$

Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- Let's use induction to prove it.
- We can use $n = 1$ as a base case:

$$\sum_{i=0}^{0} \binom{i+k}{k} = \binom{k}{k} = \binom{k+1}{k+1}.$$

- Now suppose it's true for $n = m$, we have:

$$\sum_{i=0}^{m-1} \binom{i+k}{k} = \binom{m+k}{k+1}.$$

Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- Let's use induction to prove it.
- We can use $n = 1$ as a base case:

$$\sum_{i=0}^{0} \binom{i+k}{k} = \binom{k}{k} = \binom{k+1}{k+1}.$$

- Now suppose it's true for $n = m$, we have:

$$\sum_{i=0}^{m-1} \binom{i+k}{k} = \binom{m+k}{k+1}.$$

- How can we go from m to $m+1$?

Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- Let's use induction to prove it.
- We can use $n = 1$ as a base case:

$$\sum_{i=0}^{0} \binom{i+k}{k} = \binom{k}{k} = \binom{k+1}{k+1}.$$

- Now suppose it's true for $n = m$, we have:

$$\sum_{i=0}^{m-1} \binom{i+k}{k} = \binom{m+k}{k+1}.$$

- We can add $\binom{m+k}{k}$ to both sides:

$$\sum_{i=0}^{m} \binom{i+k}{k} = \binom{m+k}{k+1} + \binom{m+k}{k}$$

Hockey stick identity

- We will use the hockey stick identity in our proof:

$$\sum_{i=0}^{n-1} \binom{i+k}{k} = \binom{n+k}{k+1}.$$

- Let's use induction to prove it.
- We can use $n = 1$ as a base case:

$$\sum_{i=0}^{0} \binom{i+k}{k} = \binom{k}{k} = \binom{k+1}{k+1}.$$

- Now suppose it's true for $n = m$, we have:

$$\sum_{i=0}^{m-1} \binom{i+k}{k} = \binom{m+k}{k+1}.$$

- We can add $\binom{m+k}{k}$ to both sides:

$$\sum_{i=0}^{m} \binom{i+k}{k} = \binom{m+k}{k+1} + \binom{m+k}{k} = \binom{m+1+k}{k+1}.$$

- Where the last step is by Pascal's identity.

Jensen's inequality

Jensen's inequality

- We will also use another form of Jensen's inequality - if f is convex, then:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]).$$

Jensen's inequality

- We will also use another form of Jensen's inequality - if f is convex, then:

$$\mathbb{E}[f(x)] \geq f(\mathbb{E}[x]).$$

- This is essentially the same as the weighted form of Jensen's inequality we have already seen:

$$\sum_{i=1}^{n} p_i f(x_i) \geq f\left(\sum_{i=1}^{n} p_i x_i\right)$$

if p_i are nonnegative with $\sum_{i=1}^{n} p_i = 1$.