Lecture 02/03/17

Lecturer: Xiaodi Wu

Reading Assignment: Chapter 12.3
Binary Search Tree: Insertion

Insert key $k$ into a binary search tree $T$

- First, $w = \text{TreeSearch}(T.\text{root}(), k)$. 
Binary Search Tree: Insertion

Insert key \( k \) into a binary search tree \( T \)

- First, \( w = \text{TreeSearch}(T.\text{root}(), k) \).
- If \( k \) is not in \( T \), i.e., \( w \) is NIL. We replace \( w \) by a node storing \((k, e)\).
Binary Search Tree: Insertion

Insert key $k$ into a binary search tree $T$

- First, $w = \text{TreeSearch}(T.\text{root}(), k)$.
- If $k$ is not in $T$, i.e., $w$ is NIL. We replace $w$ by a node storing $(k, e)$.
- If $k$ is in $T$, i.e., $w$ is a node. Call $\text{TreeSearch}(\text{rightChild}(w), k)$ and apply the above algorithm recursively. (duplicate the key)
Algorithm Tree-Insert($k, e, v, T$)
Input: a search key-element $(k, e)$ and a node $v$ of a binary search tree $T$.
Output: a updated $T$.

$w \leftarrow \text{TreeSearch}(v, k)$

if $w$ is NIL then
    Replace $w$ by a node storing $(k, e)$. Return.
else
    Tree-Insert($k, e, w$.right, $T$).
end if
Insertion in binary search trees

**Algorithm** Tree-Insert\((k, e, v, T)\)
Input: a search key-element \((k, e)\) and a node \(v\) of a binary search tree \(T\).
Output: a updated \(T\).
\(w \leftarrow \text{TreeSearch}(v, k)\)
if \(w\) is NIL then
    Replace \(w\) by a node storing \((k, e)\). Return.
else
    Tree-Insert\((k, e, w.\text{right}, T)\).
end if

- Time: \(O(h)\) could from \(O(\log n)\) to \(O(n)\).
Algorithm Tree-Insert($k, e, v, T$)
Input: a search key-element $(k, e)$ and a node $v$ of a binary search tree $T$.
Output: a updated $T$.

$w \leftarrow$ TreeSearch($v, k$)

if $w$ is NIL then
    Replace $w$ by a node storing $(k, e)$. Return.
else
    Tree-Insert($k, e, w$.right, $T$).
end if

- Time: $O(h)$ could from $O(\log n)$ to $O(n)$.
- Correctness: rely on the correctness of TreeSearch.
Binary Search Trees: insert(30)
Binary Search Trees: \text{insert}(30)
Binary Search Trees: insert(29)
Binary Search Trees: insert(29)
Binary Search Trees: insert(29)
Binary Search Tree: Insertion

More Questions

▶ One can also call TreeSearch(w.left, k). Why?
More Questions

- One can also call TreeSearch(w.left, k). Why?
- Alternative way to handle duplication of the key?
Binary Search Tree: Insertion

More Questions

- One can also call TreeSearch(w.left, k). Why?
- Alternative way to handle duplication of the key? A counter at each node!
Binary Search Tree: Deletion

Delete key $k$ in a binary search tree $T$

▶ First, $z = \text{TreeSearch}(k, T.\text{root}())$. 
Delete key $k$ in a binary search tree $T$

- First, $z=$TreeSearch($k$, $T$.root()).
- If $k$ is not in $T$, i.e., $z = NIL$. We have nothing to remove. Done!
Delete key \( k \) in a binary search tree \( T \)

- First, \( z = \text{TreeSearch}(k, \ T.\text{root}()) \).
- If \( k \) is not in \( T \), i.e., \( z = \text{NIL} \). We have nothing to remove. Done!
- Otherwise, \( w \) is a node containing key \( k \). We distinguish the following three cases.
- (1) \( z \) has no child. Replace \( z \) with NIL.
Delete key $k$ in a binary search tree $T$

- First, $z = \text{TreeSearch}(k, T.\text{root}())$.
- If $k$ is not in $T$, i.e., $z = \text{NIL}$. We have nothing to remove. Done!
- Otherwise, $w$ is a node containing key $k$. We distinguish the following three cases.
  - (1) $z$ has no child. Replace $z$ with NIL.
  - (2) $w$ has exactly one child. Transplant($T$, $z$, $z.\text{left}/z.\text{right}$)
Binary Search Tree: Deletion

Delete key $k$ in a binary search tree $T$

- First, $z = \text{TreeSearch}(k, T.\text{root}())$.
- If $k$ is not in $T$, i.e., $z = \text{NIL}$. We have nothing to remove. Done!
- Otherwise, $w$ is a node containing key $k$. We distinguish the following three cases.
  - (1) $z$ has no child. Replace $z$ with $\text{NIL}$.
  - (2) $w$ has exactly one child. $\text{Transplant}(T, z, z.\text{left}/z.\text{right})$
  - (3) $w$ has two children. Find $w$’s successor and then $\text{Transplant}$. 
Binary Search Tree: Deletion

Transplant

**Algorithm** Transplant($T, u, v$) // subtree rooted at $v$ replaces the subtree rooted at $u$

if $u.p == \text{NIL}$ then
  $T.root = v$; // Handle the case when $u$ is the root
else if $u == u.p.left$ then
  $u.p.left = v$;
else
  $u.p.right = v$; // Assign the pointer in the parent of $u$.
endif

if $v != \text{NIL}$ then
  $v.p = u.p$; // Assign the pointer of $v$
endif

What happens when Transplant($T, z, z.left/z.right$)?
Binary Search Tree: Deletion

Transplant

Algorithm Transplant(T, u, v) // subtree rooted at v replaces the subtree rooted at u
if u.p == NIL then
    T.root = v; // Handle the case when u is the root
else if u == u.p.left then
    u.p.left = v;
else
    u.p.right = v; // Assign the pointer in the parent of u.
end if
if v != NIL then
    v.p = u.p; // Assign the pointer of v
end if

▶ What happens when Transplant(T, z, z.left/z.right)?
▶ Correctness: maintain the binary search tree property.
Binary Search Tree: Deletion

Transplant

**Algorithm** Transplant\((T, u, v)\) // subtree rooted at \(v\) replaces the subtree rooted at \(u\)

if \(u.\text{p}==\text{NIL}\) then
    \(T.\text{root}=v;\) //Handle the case when \(u\) is the root
else if \(u==u.\text{p}.\text{left}\) then
    \(u.\text{p}.\text{left}=v;\)
else
    \(u.\text{p}.\text{right}=v;\) // Assign the pointer in the parent of \(u\).
end if

if \(v\neq\text{NIL}\) then
    \(v.\text{p}=u.\text{p};\) // Assign the pointer of \(v\)
end if

▶ What happens when Transplant\((T, z, z.\text{left}/z.\text{right})?\)
▶ Correctness: maintain the binary search tree property.
▶ Time: \(O(h)\).
Binary Search Trees: Delete(32)
Binary Search Trees: Delete(32)
Case 3

- (3) z has two children.
Case 3

- (3) $z$ has two children.
- Find $y$: $z$’s successor. (must be the leftmost node in the right subtree, why?)
Case 3

- (3) $z$ has two children.
- Find $y$: $z$’s successor. (must be the leftmost node in the right subtree, why?)
- Such $y$ can only have at most one child. Why?
Case 3

- (3) $z$ has two children.
- Find $y$: $z$’s successor. (must be the leftmost node in the right subtree, why?)
- Such $y$ can only have at most one child. Why?
- Two Steps: (a) replace $z$ by $y$. (b) Delete the old $y$. 

Correctness: Step (a) by the inorder property.
Step (b) by the analysis in Case 1,2.
Case 3

- (3) \( z \) has two children.
- Find \( y \): \( z \)'s successor. (must be the leftmost node in the right subtree, why?)
- Such \( y \) can only have at most one child. Why?
- Two Steps: (a) replace \( z \) by \( y \). (b) Delete the old \( y \).
- Correctness: Step (a)?
Case 3

- (3) $z$ has two children.
- Find $y$: $z$’s successor. (must be the leftmost node in the right subtree, why?)
- Such $y$ can only have at most one child. Why?
- Two Steps: (a) replace $z$ by $y$. (b) Delete the old $y$.
- Correctness: Step (a)? by the inorder property.
Case 3

- (3) \( z \) has two children.
- Find \( y \): \( z \)'s successor. (must be the leftmost node in the right subtree, why?)
- Such \( y \) can only have at most one child. Why?
- Two Steps: (a) replace \( z \) by \( y \). (b) Delete the old \( y \).
- Correctness: Step (a)? by the inorder property.
- Step (b)?
Case 3

- (3) $z$ has two children.
- Find $y$: $z$’s successor. (must be the leftmost node in the right subtree, why?)
- Such $y$ can only have at most one child. Why?
- Two Steps: (a) replace $z$ by $y$. (b) Delete the old $y$.
- Correctness: Step (a)? by the inorder property.
- Step (b)? by the analysis in Case 1,2
Binary Search Tree: Deletion

Case 3

- (3) z has two children.
- Find y: z’s successor. (must be the leftmost node in the right subtree, why?)
- Such y can only have at most one child. Why?
- Two Steps: (a) replace z by y. (b) Delete the old y.
- Correctness: Step (a)? by the inorder property.
- Step (b)? by the analysis in Case 1,2.
- Time: $O(h)$.
Binary Search Trees: Delete(65)
Binary Search Trees: Delete(65)
Binary Search Trees: Delete(65)
Describe an algorithm that checks whether $T$ is a valid binary search tree. Analyze the worst-case complexity of your algorithm.

Assume $T$ is a binary search tree and let $k$ be another input. Describe an algorithm that finds one of the closest-to-$k$ keys in the binary tree $T$. Analyze the worst-case complexity of your algorithm. (Assume all the keys are integers and the distance between two keys is the absolute value of their difference.)