Lecture 10/17/16

Lecturer: Xiaodi Wu

Reading: Chapter 2.3, Chapter 3.1
Properties about Binary Trees

Theorem
Let \( T \) be a (proper) binary tree with \( n \) nodes, \( h \) the height of \( T \). We have

- \# external nodes of \( T \) is between \( h + 1 \) and \( 2^h \).
- \# internal nodes of \( T \) is between \( h \) and \( 2^h - 1 \).
- The height of \( T \) is between \( \log(n + 1) - 1 \) and \( (n - 1)/2 \).
Theorem (Theorem 2.9, on page 85)

In a (proper) binary tree $T$, the number of external nodes is 1 more than the number of internal nodes.
Properties about Binary Trees

Theorem (Theorem 2.9, on page 85)

In a (proper) binary tree $T$, the number of external nodes is 1 more than the number of internal nodes.

Proof.

By induction,

- If $T$ only has one node, it must be external. Thus, no internal node. The statement holds.
Properties about Binary Trees

Theorem (Theorem 2.9, on page 85)

In a (proper) binary tree $T$, the number of external nodes is 1 more than the number of internal nodes.

Proof.
By induction,

- If $T$ only has one node, it must be external. Thus, no internal node. The statement holds.

- Otherwise, $T$ has at least one external node with its parent. Remove any external node $w$ and its parent $v$, then connect $w$’s sibling to $v$’s parent. The tree remains proper and binary, but smaller.
Properties about Binary Trees

Let $e$, $i$ be external/internal nodes of a (proper) binary tree.

$e = i + 1$ and $e + i = n$. 

$n \geq 2^h + 1$. What is this case?

$n \leq 2^h + 1 - 1$. What is this case?

$(n - 1)/2 \leq h \leq \log(n + 1) - 1$.

$h + 1 \leq e \leq 2h$.

$h \leq i \leq 2h - 1$. 


Properties about Binary Trees

Let \( \#e, \#i \) be \# external/internal nodes of a (proper) binary tree.

- \( \#e = \#i + 1 \) and \( \#e + \#i = n \).
- \( n \geq 2h + 1 \). What is this case?
Properties about Binary Trees

Let \( \#e, \#i \) be \# external/internal nodes of a (proper) binary tree.

- \( \#e = \#i + 1 \) and \( \#e + \#i = n \).
- \( n \geq 2h + 1 \). What is this case?
- \( n \leq 2^{h+1} - 1 \). What is this case?
Properties about Binary Trees

Let \#e, \#i be \# external/internal nodes of a (proper) binary tree.

- \#e = \#i + 1 and \#e + \#i = n.
- \( n \geq 2h + 1 \). What is this case?
- \( n \leq 2^{h+1} - 1 \). What is this case?
- \( (n - 1)/2 \leq h \leq \log(n + 1) - 1 \).
- \( h + 1 \leq \#e \leq 2^h \).
- \( h \leq \#i \leq 2^h - 1 \).
Implementation of Binary Trees

Vector-based Structure

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$. 

Methods: leftChild(), rightChild(), root(), parent(), children(), $O(1)$ time.

Space could be as large as $O(2^{n+1}/2)$. 

Implementation of Binary Trees

Vector-based Structure

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$.

- methods: leftChild(), rightChild(), root(), parent(), children(), $O(1)$ time.
- Space could be as large as $O(2^{(n+1)/2})$. 
Implementation of Binary Trees

Linked Structure: similar to doubly linked list

▶ Each node: pointers to parent, leftChild, rightChild, and the element stored.
▶ methods: leftChild(), rightChild(), root(), parent(), children(), $O(1)$ time.
▶ Space usage $O(n)$. 
Search in a sorted table

- Search a key $k$ in a table of size $n$. Trivial $O(n)$. 
  - Maintain three pointers: low, high, and mid = \( \frac{\text{low} + \text{high}}{2} \).
  - Compare $k$ with the key of the mid. If $k = \text{key(mid)}$, return mid.
  - If $k < \text{key(mid)}$, then update the pointer low ← low, high ← mid − 1.
  - If $k > \text{key(mid)}$, then update the pointer low ← mid + 1, high ← high.
Search in a sorted table

- Search a key $k$ in a table of size $n$. Trivial $O(n)$.
- In a sorted table (non-decreasing order): $O(\log(n))$. **Binary Search!**
Search in a sorted table

- Search a key \( k \) in a table of size \( n \). Trivial \( O(n) \).
- In a sorted table (non-decreasing order): \( O(\log(n)) \). **Binary Search!**

How?

- Maintain three pointers: \( \text{low} \), \( \text{high} \), and \( \text{mid} = (\text{low} + \text{high})/2 \).
Search in a sorted table

- **Search a key \( k \) in a table of size \( n \). Trivial \( O(n) \).**
- **In a sorted table (non-decreasing order): \( O(\log(n)) \). **Binary Search!**

**How?**

- Maintain three pointers: \( \text{low, high, and mid= (low+high)/2} \).
- Compare \( k \) with the key of the mid. If \( k = \text{key(mid)} \), return mid.
- If \( k < \text{key(mid)} \), then update the pointer \( \text{low} \leftarrow \text{low}, \text{high} \leftarrow \text{mid} - 1 \).
- If \( k > \text{key(mid)} \), then update the pointer \( \text{low} \leftarrow \text{mid} + 1, \text{high} \leftarrow \text{high} \).
Binary Search: a recursive implementation

**Algorithm** BinarySearch($S, k, low, high$)

Input: an ordered vector $S$ storing $n$ items.
Output: an element with key $k$ within $[low, high]$; otherwise, NO_SUCH_KEY.

if $low > high$ then
  return NO_SUCH_KEY
else
  mid ← $(low + high)/2$
  if $k = key(mid)$ then
    return mid.
  else if $k < key(mid)$ then
    return BinarySearch($S, k, low, mid-1$).
  else
    return BinarySearch($S, k, mid+1, high$).
end if
Binary Search: time and correctness

Time

- Watch the difference between low and high. Shrink to half in each recursive call.

Correctness

- Maintain an invariant: the key is either within \([\text{low}, \text{high}]\) or does not exist.
- Invariant remains during recursive calls.
Binary Search: time and correctness

Time

- Watch the difference between low and high. Shrink to half in each recursive call.
- \( O(\log(high - low)) = O(\log(n)) \).
Binary Search: time and correctness

Time

- Watch the difference between low and high. Shrink to half in each recursive call.
- $O(\log(\text{high} - \text{low})) = O(\log(n))$.

Correctness

- Maintain an invariant: the key is either within $[\text{low}, \text{high}]$ or does not exist.
- Invariant remains during recursive calls.
Binary Search Tree

Definition

- **Binary Search Tree**: for every internal node $e$, the elements in the left subtree are $\leq e$, and the elements in the right subtree are $\geq e$.

- Goal: binary search on a tree data structure.
Binary Search Tree

Definition

- **Binary Search Tree**: for every internal node $e$, the elements in the left subtree are $\leq e$, and the elements in the right subtree are $\geq e$.
- Goal: binary search on a tree data structure.

Search

- Compare $k$ with the key of the root. If $k = \text{key}(\text{root})$, return root.
- If $k < \text{key}(\text{root})$, then search in the left subtree.
- If $k > \text{key}(\text{root})$, then search in the right subtree.
Search in binary search trees

**Algorithm** TreeSearch($k$, $v$)

Input: a search key $k$ and a node $v$ of a binary search tree $T$.

Output: the node with key $k$ or an external node, i.e., NO_SUCH_KEY.

if $v$ is external then
  return $v \Rightarrow$ NO_SUCH_KEY

else
  if $k = \text{key}(v)$ then
    return $v$.
  else if $k < \text{key}(v)$ then
    return TreeSearch($k$, $T$.leftChild($v$)).
  else
    return TreeSearch($k$, $T$.rightChild($v$)).
end if
end if

▶ Time: $O(h)$ could from $O(\log n)$ to $O(n)$. 
Search in binary search trees

**Algorithm** TreeSearch\((k, v)\)
Input: a search key \(k\) and a node \(v\) of a binary search tree \(T\).
Output: the node with key \(k\) or an external node, i.e., NO_SUCH_KEY.

```plaintext
if \(v\) is external then
    return \(v \Rightarrow \text{NO\_SUCH\_KEY}\)
else
    if \(k = \text{key}(v)\) then
        return \(v\).
    else if \(k < \text{key}(v)\) then
        return TreeSearch\((k, T.\text{leftChild}(v))\).
    else
        return TreeSearch\((k, T.\text{rightChild}(v))\).
end if
end if
```

- Time: \(O(h)\) could from \(O(\log n)\) to \(O(n)\).
Binary Search Trees: TreeSearch(78, root)
Binary Search Tree Properties

Inorder Traversal of Binary Search Trees

- Inorder Traversal leads to a nondecreasing sequence.

  \[17, 28, 29, 32, 44, 54, 66, 76, 78, 80, 82, 88, 97]\n
- Given a binary tree: inorder traversal nondecreasing \(\Leftrightarrow\) binary search tree.
Binary Search Tree: Insertion

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

- First, \( w = \text{TreeSearch}(k, T.\text{root}()) \).

(duplicate the key)
Binary Search Tree: Insertion

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

- First, $w = \text{TreeSearch}(k, T.\text{root}())$.
- If $k$ is not in $T$, i.e., $w$ is an external node. We replace $w$ by an internal node storing $(k, e)$ and add two external children to $w$. 
Insert key $k$ into a binary search tree $T$

- First, $w =$ TreeSearch($k$, $T$.root()).
- If $k$ is not in $T$, i.e., $w$ is an external node. We replace $w$ by an internal node storing $(k, e)$ and add two external children to $w$.
- If $k$ is in $T$, i.e., $w$ is an internal node. Call TreeSearch($k$, rightChild($w$)) and apply the above algorithm recursively. (duplicate the key)
Algorithm insertItem\((k, e, v, T)\)
Input: a search key-element \((k, e)\) and a node \(v\) of a binary search tree \(T\).
Output: a updated \(T\).
\(w \leftarrow TreeSearch(k, v)\)
if \(w\) is external then
    Replace \(w\) by an internal node storing \((k, e)\) with two external children. Return.
else
    insertItem\((k, e, T.rightChild(w), T)\).
end if
Insertion in binary search trees

**Algorithm** `insertItem(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`.

1. `w ← TreeSearch(k, v)`
2. **if** `w` is external **then**
   - Replace `w` by an internal node storing `(k, e)` with two external children. Return.
3. **else**
   - `insertItem(k, e, T.rightChild(w), T)`.
4. **end if**

- Time: $O(h)$ could from $O(\log n)$ to $O(n)$. 

Correctness: rely on the correctness of `TreeSearch`. 
Algorithm insertItem\((k, e, v, T)\)
Input: a search key-element \((k, e)\) and a node \(v\) of a binary search tree \(T\).
Output: a updated \(T\).
\(w \leftarrow TreeSearch\(k, v\)\)
\(\textbf{if } w \text{ is external } \textbf{then}\)
\hspace{1em} Replace \(w\) by an internal node storing \((k, e)\) with two external children. Return.
\(\textbf{else}\)
\hspace{1em} insertItem\((k, e, T.\text{rightChild}(w), T)\).
\(\textbf{end if}\)

- 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: 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(k, leftChild(w)). Why?
More Questions

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

More Questions

- One can also call TreeSearch(k, leftChild(w)). Why?
- Alternative way to handle duplication of the key? A counter at each node!
Remove key $k$ out of a binary search tree $T$

- First, $w$ = TreeSearch($k$, T.root()).
Binary Search Tree: Removal

Remove key $k$ out of a binary search tree $T$

- First, $w = \text{TreeSearch}(k, T.\text{root}())$.
- If $k$ is not in $T$, i.e., $w$ is an external node. We have nothing to remove. Done!
Binary Search Tree: Removal

Remove key $k$ out of a binary search tree $T$

- First, $w = \text{TreeSearch}(k, T.\text{root})$.
- If $k$ is not in $T$, i.e., $w$ is an external node. We have nothing to remove. Done!
- Otherwise, $w$ is an internal node. We distinguish the following two cases.
  - (1) at least one of the children of $w$ is an external node.
  - (2) both of the children of $w$ are internal nodes.
Binary Search Tree: Removal

Case 1

- (1) at least one of the children of \( w \) is an external node.
Binary Search Tree: Removal

Case 1

- (1) at least one of the children of $w$ is an external node.
- Let $z$ be the external child. Let $y$ be the other child.
- Remove $z$, $w$ and connect $y$ to $w$’s parent replacing $w$’s position.

Correctness: maintain the binary search tree property.

Time: $O(h)$. 
Binary Search Tree: Removal

Case 1

- (1) at least one of the children of $w$ is an external node.
- Let $z$ be the external child. Let $y$ be the other child.
- Remove $z, w$ and connect $y$ to $w$’s parent replacing $w$’s position.
- Correctness: maintain the binary search tree property.
Binary Search Tree: Removal

Case 1

- (1) at least one of the children of \( w \) is an external node.
- Let \( z \) be the external child. Let \( y \) be the other child.
- Remove \( z, w \) and connect \( y \) to \( w \)’s parent replacing \( w \)’s position.
- Correctness: maintain the binary search tree property.
- Time: \( O(h) \).
Binary Search Trees: Remove(32)
Binary Search Trees: Remove(32)
Case 2

- (2) both of the children of \( w \) are internal nodes.
Binary Search Tree: Removal

Case 2

- (2) both of the children of $w$ are internal nodes.
- Find $y$: the first internal node that follows $w$ in an inorder traversal. How?
Binary Search Tree: Removal

Case 2

- (2) both of the children of $w$ are internal nodes.
- Find $y$: the first internal node that follows $w$ in an inorder traversal. How?
- Such $y$ must have an external left child. Why?
Binary Search Tree: Removal

Case 2

- (2) both of the children of $w$ are internal nodes.
- Find $y$: the first internal node that follows $w$ in an inorder traversal. How?
- Such $y$ must have an external left child. Why?
- Two Steps: (a) replace $w$ by $y$. (b) Remove($y$).
Binary Search Tree: Removal

Case 2

- (2) both of the children of \( w \) are internal nodes.
- Find \( y \): the first internal node that follows \( w \) in an inorder traversal. How?
- Such \( y \) must have an external left child. Why?
- Two Steps: (a) replace \( w \) by \( y \). (b) Remove(\( y \)).
- Correctness: Step (a)?
Case 2

- (2) both of the children of $w$ are internal nodes.
- Find $y$: the first internal node that follows $w$ in an inorder traversal. How?
- Such $y$ must have an external left child. Why?
- Two Steps: (a) replace $w$ by $y$. (b) Remove($y$).
- Correctness: Step (a)? by the inorder property.
Binary Search Tree: Removal

Case 2

- (2) both of the children of \( w \) are internal nodes.
- Find \( y \): the first internal node that follows \( w \) in an inorder traversal. How?
- Such \( y \) must have an external left child. Why?
- Two Steps: (a) replace \( w \) by \( y \). (b) Remove(\( y \)).
- Correctness: Step (a)? by the inorder property.
- Step (b)?
Case 2

- (2) both of the children of \( w \) are internal nodes.
- Find \( y \): the first internal node that follows \( w \) in an inorder traversal. How?
- Such \( y \) must have an external left child. Why?
- Two Steps: (a) replace \( w \) by \( y \). (b) Remove(\( y \)).
- Correctness: Step (a)? by the inorder property.
- Step (b)? by the analysis in Case 1
Case 2

- (2) both of the children of $w$ are internal nodes.
- Find $y$: the first internal node that follows $w$ in an inorder traversal. How?
- Such $y$ must have an external left child. Why?
- Two Steps: (a) replace $w$ by $y$. (b) Remove($y$).
- Correctness: Step (a)? by the inorder property.
- Step (b)? by the analysis in Case 1.
- Time: $O(h)$. 

Binary Search Tree: Removal
Binary Search Trees: Remove(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.)