Search in a sorted table

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

Correctness

- Maintain an invariant: the key is either within $[low, high]$ 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.
- \( O(\log(high - low)) = O(\log(n)) \).

Correctness

- Maintain an invariant: the key is either within \([low, high]\) 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.
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.

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)
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 $\iff$ binary search tree.
Binary Search Tree Properties

Inorder Traversal of Binary Search Trees

- Inorder Traversal leads to a nondecreasing sequence.

\[17, 28, 29, 32, 44, 54, 66, 76, 78, 80, 82, 88, 97\]

- Given a binary tree: inorder traversal nondecreasing \(\iff\) binary search tree.

Binary Search Tree vs Heap

- Binary Search Property vs Heap-Order Property
Binary Search Tree Properties

Inorder Traversal of Binary Search Trees

- Inorder Traversal leads to a nondecreasing sequence.

  
  \[17, 28, 29, 32, 44, 54, 66, 76, 78, 80, 82, 88, 97\]

- Given a binary tree: inorder traversal nondecreasing \(\iff\) binary search tree.

Binary Search Tree vs Heap

- Binary Search Property vs Heap-Order Property
- Any binary tree vs Complete Binary Tree
Binary Search Tree: Insertion

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

- First, $w =$ TreeSearch($k$, $T$.root()).
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 $w$. 

(duplicate the key)
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($k, e, T.\text{rightChild}(w), T$).

*end if*
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

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

- **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)