CS121 Data Structures A, C
Binary Search Trees

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

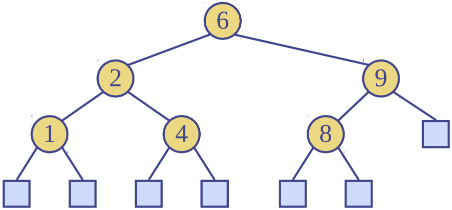
Binary Search Trees

A **binary search tree** (BST) is a *proper* binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have

$$key(u) < key(v) < key(w)$$

External nodes do not store items

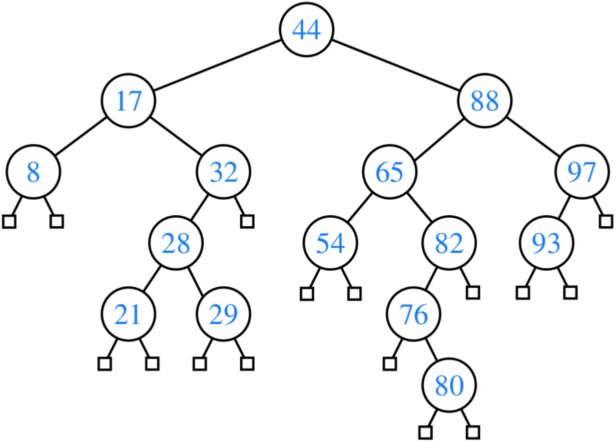
An inorder traversal of a BST visits the keys in increasing order



Binary Search Tree Example

We need to have an order relation defined on the keys

The use of sentinel external nodes simplifies the presentation of several of our search and update algorithms



Search

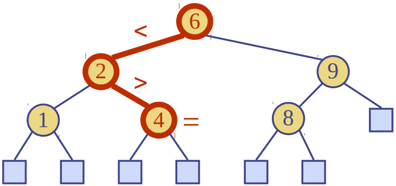
To search for a key k, we trace a downward path starting at the root

The next node visited depends on the comparison of k with the key of the current node

If we reach a leaf, the key is not found

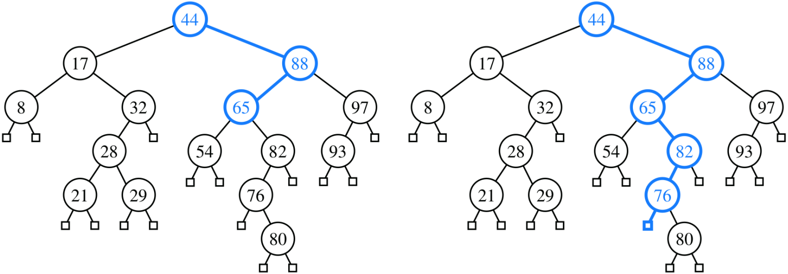
Example: `get(4)` calls `TreeSearch(root, 4)`

```
Algorithm TreeSearch(p, k):
  if T.isExternal(p) then
    return p
  if k < key(p) then
    return TreeSearch(left(p), k)
  else if k = key(p) then
    return p
  else                               ▷k > key(p)
    return TreeSearch(right(p), k)
```



Examples of Search in a BST

Searches for key 65 (on the left) and key 68 (on the right)



(a) (b)

Analysis of Binary Tree Searching

Algorithm `TreeSearch` is recursive and executes a constant number of primitive operations for each recursive call

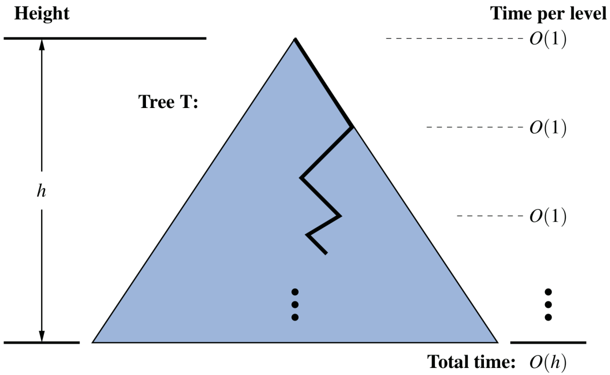
Each recursive call of `TreeSearch` is made on a child of the previous position

`TreeSearch` is called on the positions of a path of T that starts at the root and goes down one level at a time

Thus, the number of such positions is bounded by $h + 1$, where h is the height of T

The overall search runs in $O(h)$ time

Analysis of Binary Tree Searching (cont'd)



In the context of the sorted map ADT, the search will be used as a subroutine for implementing the `get` method, as well as for the `put` and `remove` methods

Each of these methods begins by trying to locate an existing entry with the given key

Insertion

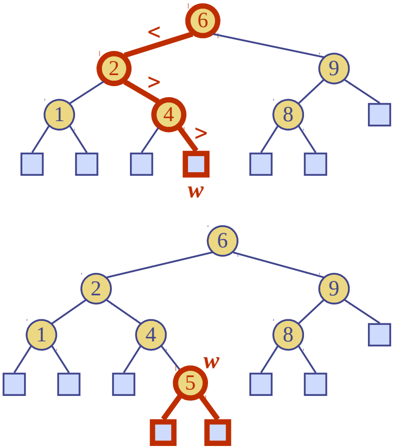
To perform operation `put(`k, v`)`, we search for key k (using `TreeSearch`)

If found, that entry's existing value is reassigned

Otherwise, let w be the leaf reached by the search

We insert k at node w and expand w into an internal node

Example: insert 5



Insertion Algorithm

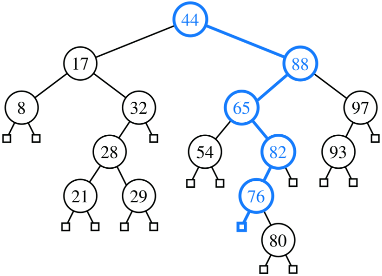
Let us assume a proper binary tree supports the following update operation:

expandExternal(p, e): Stores entry e at the external position p, and expands p to be internal, having two new leaves as children

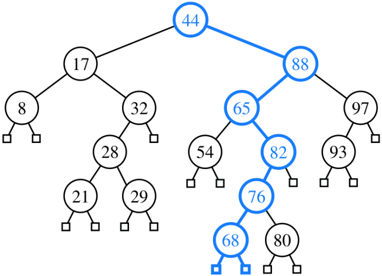
Algorithm `TreeInsert`(k, v):
Input: A search key k to be associated with value v
$p = \texttt{TreeSearch}(\texttt{root}(\ ),\ k)$
if $k == \texttt{key}(p)$ **then**
Change p's value to (v)
else
$\texttt{expandExternal}(p,\ (k, v))$

Example of Insertion into a BST

Insertion of an entry with key 68 into the search tree



(a)



(b)

Deletion

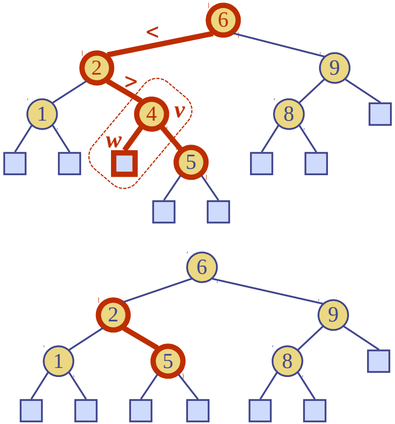
To perform operation `remove(`k`)`, we search for key k (using `TreeSearch`)

If an external node is returned, no entry to remove

Otherwise, if key k is found, let v be the node storing k

If node v has a leaf child w, we remove v and w from the tree, i.e. remove w and its parent

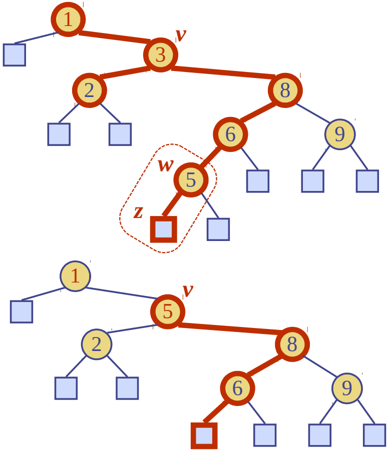
Example: remove 4



Deletion (cont'd)

We consider the case where the key k to be removed is stored at a node v whose children are both internal

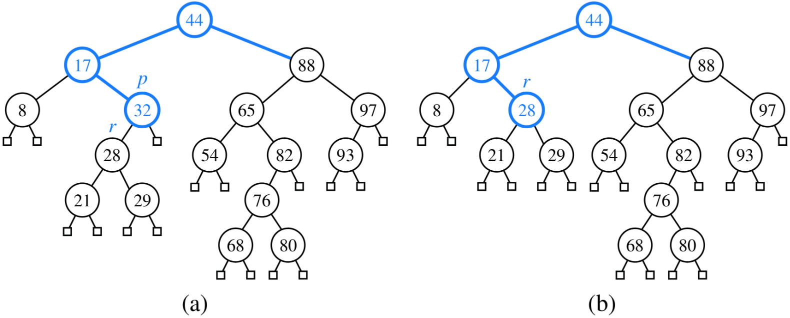
- we find the internal node w that follows v in an inorder traversal
- we copy $key(w)$ into node v
- we remove node w and its left child z (which must be a leaf) by means of removing z and its parent from the tree



Example: remove 3

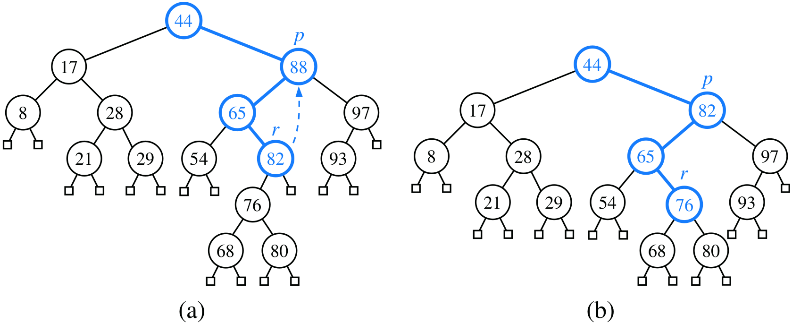
Example of Deletion from a BST: First Case

Deletion of an entry with key 32 from the search tree



Example of Deletion from a BST: Second Case

Deletion of an entry with key 88 from the search tree

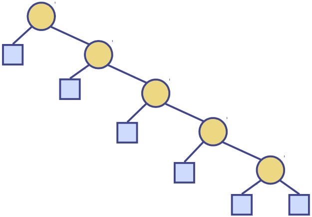


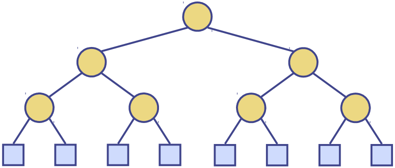
Performance

Consider a sorted map with n items implemented by means of a binary search tree of height h

- the space used is $O(n)$
- methods `get`, `put` and `remove` take $O(h)$ time (each relies on `TreeSearch`)

The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case





Sorted Map Implementation Using a BST

A slightly complicated implementation of the sorted map ADT using a BST is available in the textbook.

Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(h)$
firstEntry, lastEntry	$O(h)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(h)$
subMap	$O(s+h)$
entrySet, keySet, values	$O(n)$

Summary

Reading

Section 11.1 Binary Search Trees

Questions?