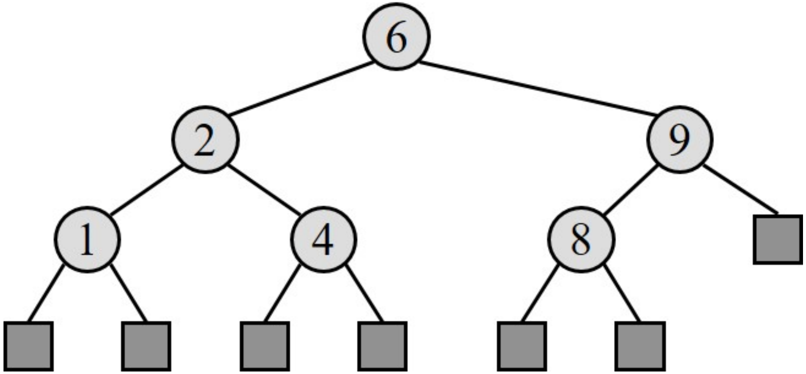
Computer Science 3A - CSC3A10

Lecture 10a: Search Trees

Academy of Computer Science and Software Engineering
University of Johannesburg

Binary Search Trees



Ordered Dictionaries

Keys are assumed to come from a total order.
New operations:

- **first()**: first entry in the dictionary ordering
- **last()**: last entry in the dictionary ordering
- **successors(k)**: iterator of entries with keys greater than or equal to k; increasing order
- **predecessors(k)**: iterator of entries with keys less than or equal to k; decreasing order

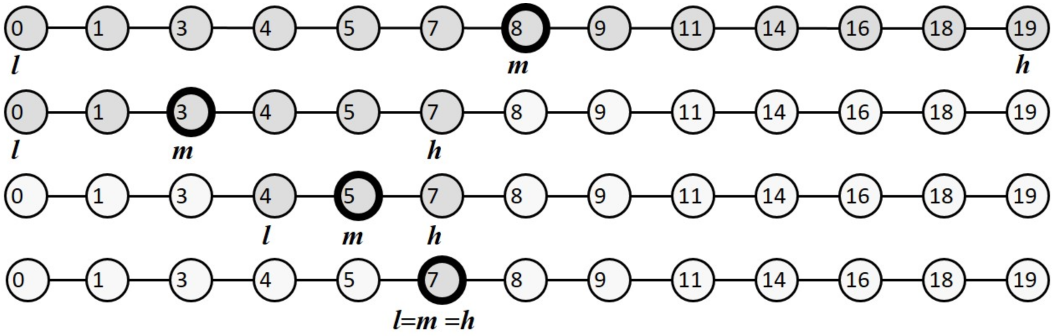
Binary Search

Binary search can perform operation find(k) on a dictionary implemented by means of an array-based sequence, sorted by key

- similar to the high-low game
- at each step, the number of candidate items is halved
- terminates after $O(logn)$ steps

Binary Search II

Example: find(7)



Search Tables

A search table is a dictionary implemented by means of a sorted sequence

- We store the items of the dictionary in an array-based sequence, sorted by key
- We use an external comparator for the keys

Search Tables II

Performance:

- find takes $O(logn)$ time, using binary search
- insert takes $O(n)$ time since in the worst case we have to shift n/2 items to make room for the new item
- remove take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal

The lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Trees

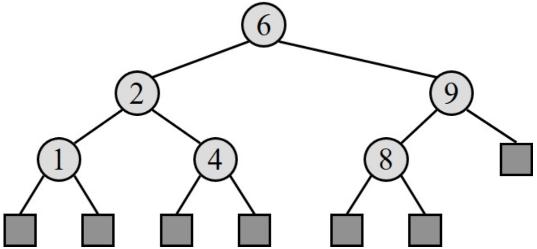
A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:

- 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) $\leq$ key(v) $\leq$ key(w)

External nodes do not store items!

Binary Search Trees II

An inorder traversal of a binary search trees visits the keys in increasing order



Binary Search Tree: Search

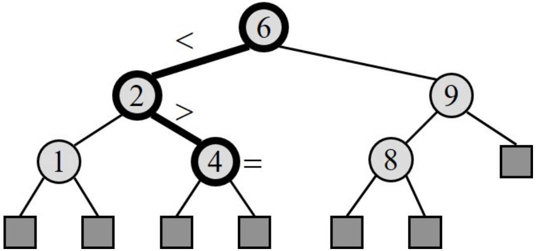
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return *null*

Binary Search Tree: Search II

```
Algorithm TreeSearch(k, v)
  if T.isExternal (v)
  return v
if $k < key(v)$
  return TreeSearch(k, T.left(v))
else if $k = key(v)$
  return v
else { $k > key(v)$ }
  return TreeSearch(k, T.right(v))
```

Binary Search Tree: Search III

Example: find(4): Call TreeSearch(4,root)

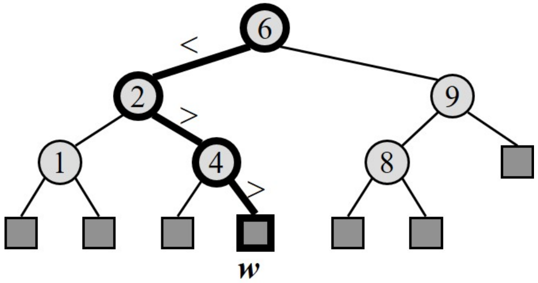


Binary Search Tree: Insert

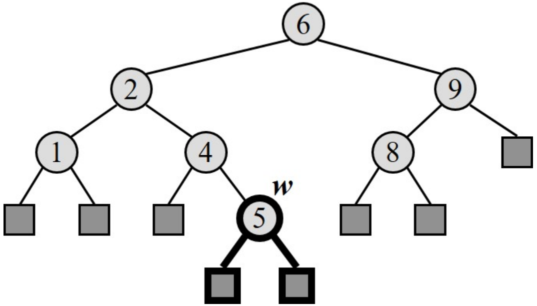
- To perform operation inser(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

Binary Search Tree: Insert II

Example: insert 5



Binary Search Tree: Insert III

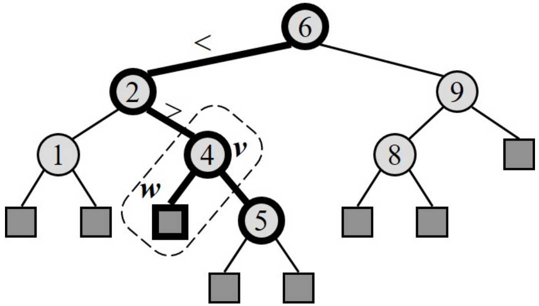


Binary Search Tree: Remove

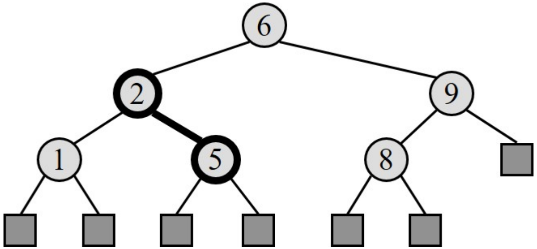
- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent

Binary Search Tree: Remove II

Example: remove 4



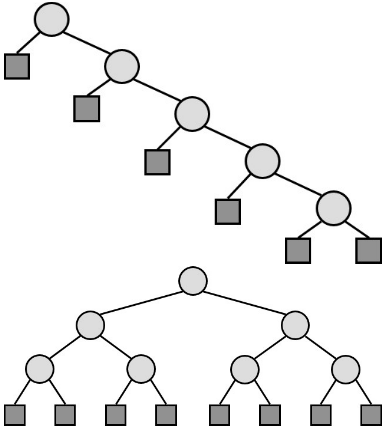
Binary Search Tree: Remove III



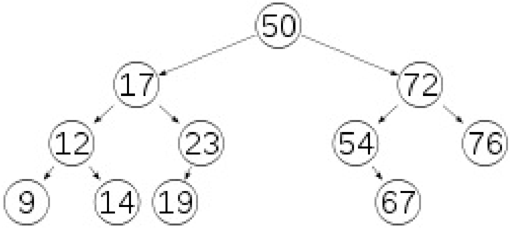
Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
- the space used is O(n)
- methods find, insert and remove take O(h) time
- The height h is $O(n)$ in the worst case and $O(logn)$ in the best case

Performance II



AVL Tree ADT

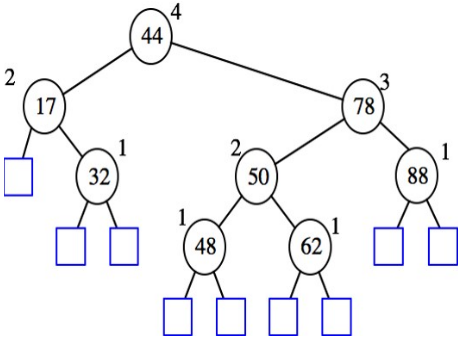


AVL Tree Properties

- AVL trees are balanced.
- An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1.

AVL Tree Properties II

An example of an AVL tree where the heights are shown next to the nodes:



Height of an AVL Tree

Fact: The height of an AVL tree storing n keys is O(log n). **Proof:**

- Let us bound $n(h)$: the minimum number of internal nodes of an AVL tree of height h.
- We easily see that $n(1) = 1$ and $n(2) = 2$
- For n $>$ 2, an AVL tree of height h contains the root node, one AVL subtree of height $n - 1$ and another of height $n - 2$.
- That is, $n(h) = 1 + n(h - 1) + n(h - 2)$

Height of an AVL Tree II

- Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. So
  - $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(n-6)$, ... (by induction),
  - $n(h) > 2^i n(h-2^i)$
- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: $h < 2logn(h) + 2$
- Thus the height of an AVL tree is $O(logn)$

Insertion in an AVL Tree

Insertion is as in a binary search tree. Always done by expanding an external node.

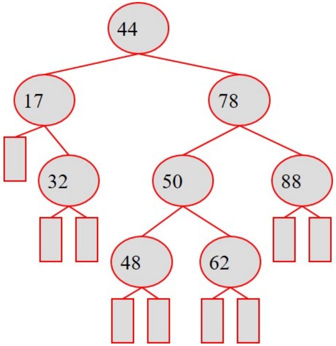


Figure: Before insertion

Insertion in an AVL Tree II

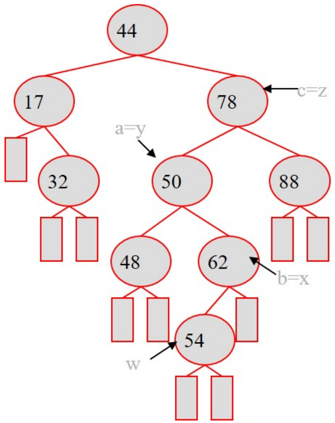
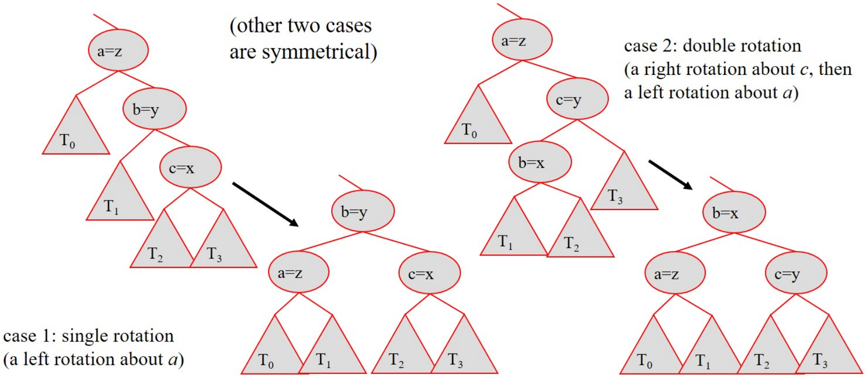


Figure: After insertion

Trinode Restructuring

let (a,b,c) be an inorder listing of x, y, z
perform the rotations needed to make b the topmost node of the three


a=z
b=y
c=x
T_0
T_1
T_2
T_3
(other two cases are symmetrical)
case 1: single rotation (a left rotation about a)
b=y
a=z
c=x
T_0
T_1
T_2
T_3
a=z
c=y
b=x
T_0
T_1
T_2
T_3
case 2: double rotation (a right rotation about c, then a left rotation about a)
b=x
a=z
c=y
T_0
T_1
T_2
T_3

AVL Insertion Example

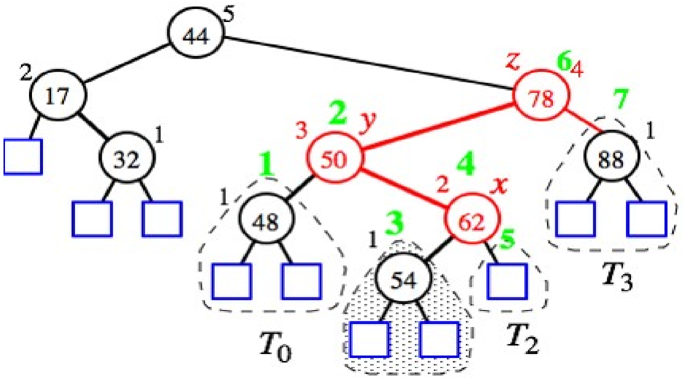


Figure: Unbalanced

AVL Insertion Example II

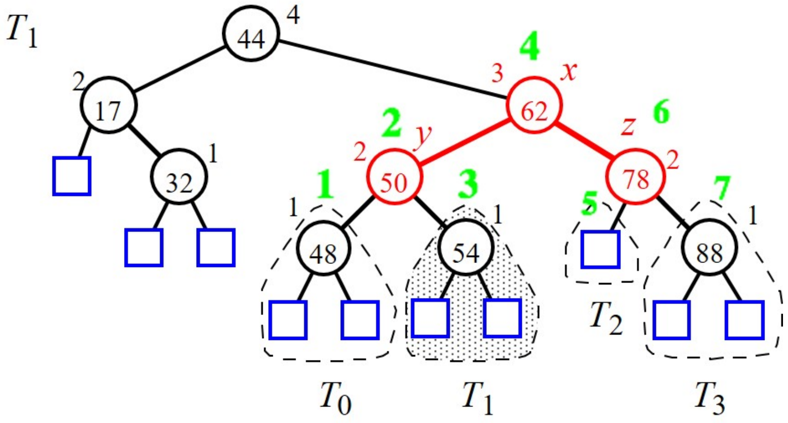
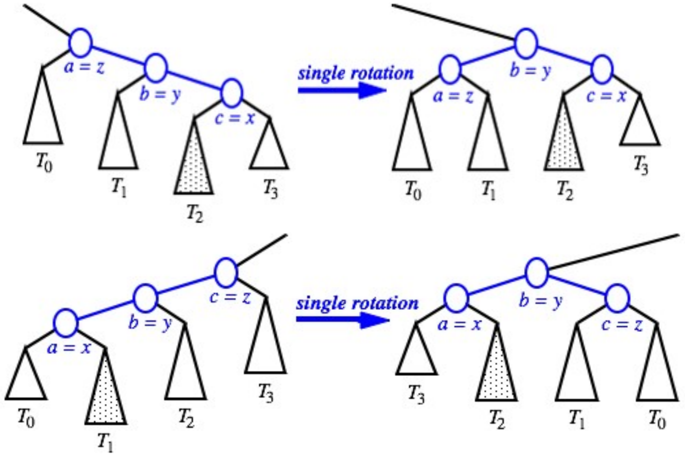
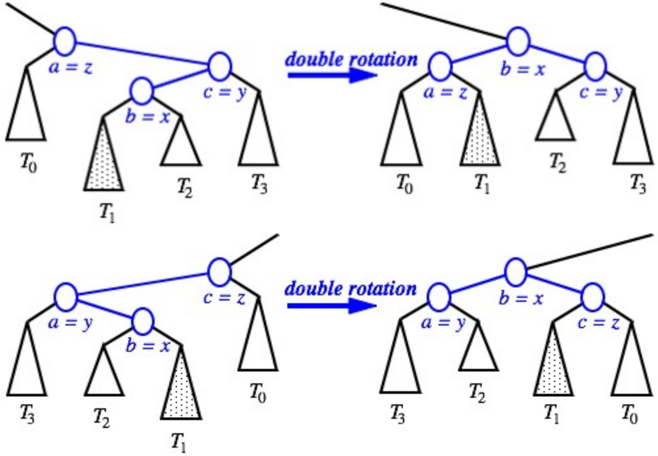


Figure: Balanced

AVL Restructuring (as Single Rotations)

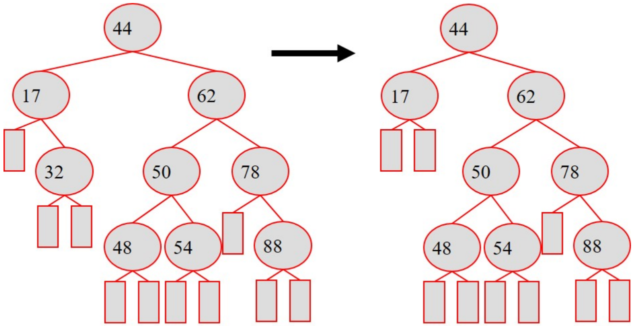


AVL Restructuring (as Double Rotations)



AVL Tree Removal

Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.

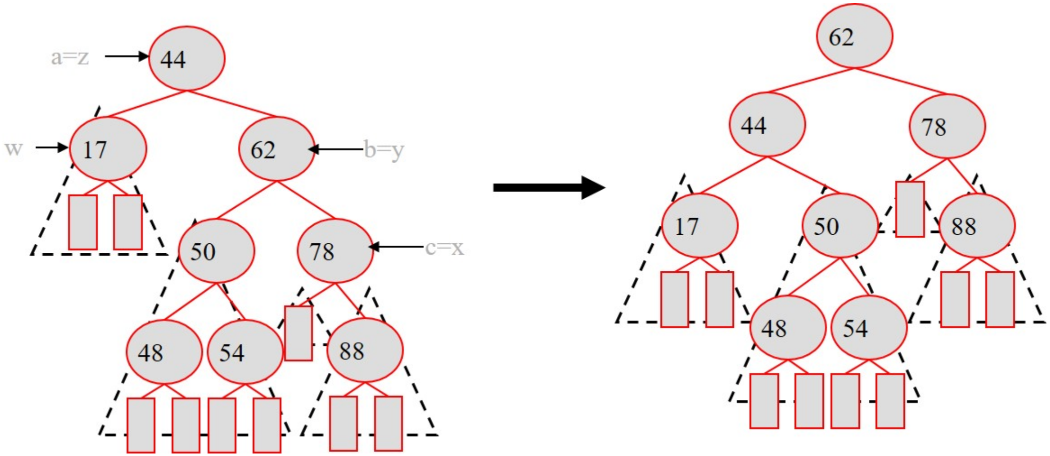


before deletion of 32 after deletion

AVL Tree Removal Rebalance II

- Let z be the first unbalanced node encountered while traveling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
- We perform restructure(x) to restore balance at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

AVL Tree Removal Rebalance II



Running Times for AVL Trees

- A **single restructure** is O(1) using a linked-structure binary tree
- **find** is O(log n) as the height of tree is O(log n), no restructures are needed
- **insert** is O(log n) as the initial find is O(log n). Restructuring up the tree, maintaining heights is O(log n)
- **remove** is O(log n), as the initial find is O(log n). Restructuring up the tree, maintaining heights is O(log n)

Important Notice:

Splay Trees will **NOT** form part of the scope for the exam.