CSE221 Data Structures
Lecture 18: Binary Search Trees and AVL Trees

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

November 17, 2021

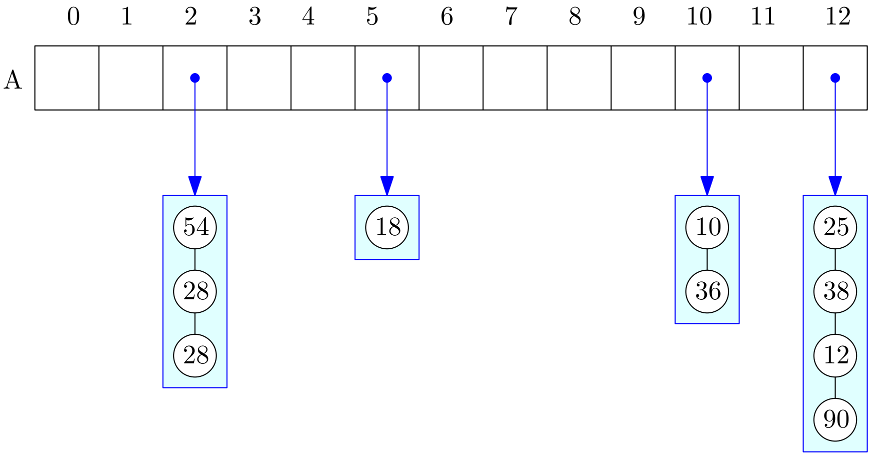
Introduction

- Final exam is on Wednesday 15 December, 20:00–22:00.
- Assignment 3 is due on Thursday next week.
- Reference for this lecture: Textbook Chapter 9.5, 10.1 and 10.2.

Dictionaries

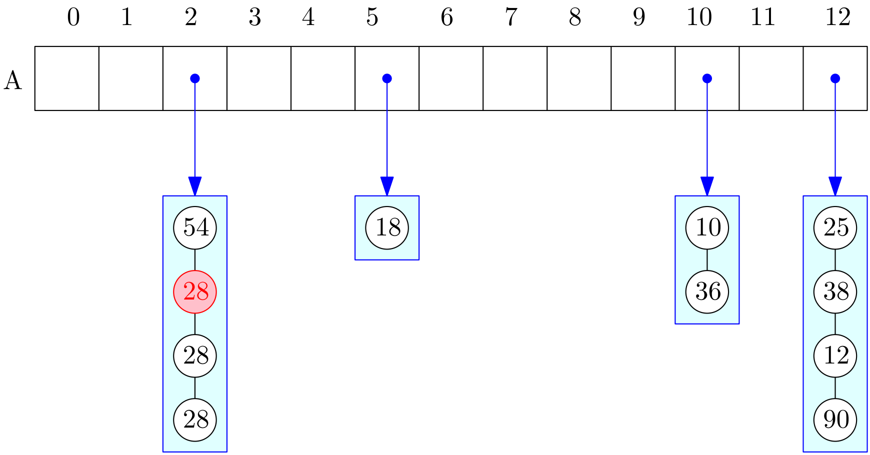
- A *dictionary* ADT stores key-value pairs (k, v) called *entries*.
- The keys stored in a dictionary are *not* necessarily unique.
- So a dictionary can store two entries (k, v) and (k, v').
- This is the main difference with a map, in which keys are unique.
- Dictionary operations are the same as for maps, except for the differences below:
  - **put**$(k,\ v)$ is replaced with **insert**(k, v) which inserts a new entry with key k. It does *not* overwrite a previous entry (k, v') if there was one.
  - **find**(k) returns an iterator referring to an entry (k, v) if there is (at least) one in the dictionary.
  - **findAll**(k) returns a pair of iterators (b, e), such that all the entries with key value k lie in the range $[b, e)$.
  - **erase**(k) Remove from D an arbitrary entry with key equal to k.

Dictionaries



- How to insert a new pair $(28, v)$?

Dictionaries

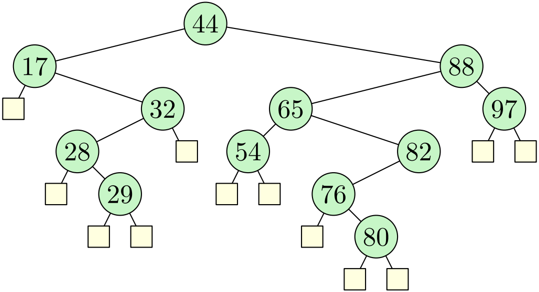


- Insert it *before* the first such pair.

Dictionaries

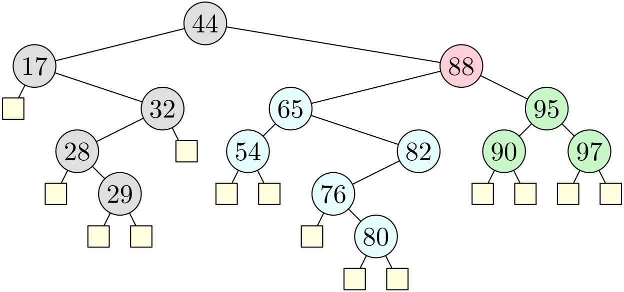
- We can implement a dictionary ADT using a hash table with separate chaining.
- As shown in the previous slide, we make sure that all the entries with the same key k are contiguous, by inserting any new entry (k, v) at the position before the first such entry.
- Then all operations take $O(1)$ expected time, except for `findAll` which take time $O(1 + s)$ where s is the number of items that are found.
- We will now present *binary search trees*, which allow us to perform in $O(\log n)$ time any ordered map or dictionary operation.

Binary Search Trees



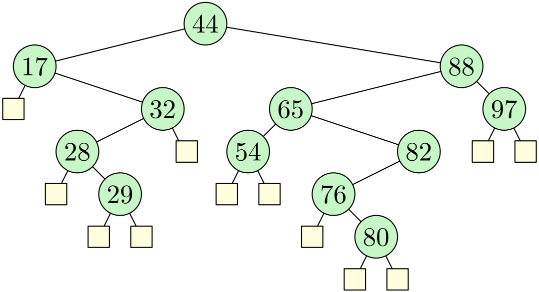
- A *binary search tree* (BST)is a *full binary tree*, i.e. each internal node has exactly 2 children.
- Each internal node records an entry (k, x). We only represent k in the figures.

Binary Search Trees



- For any node v storing (k, x):
  - All the keys in the left subtree are $\leqslant k$.
  - All the keys in the right subtree are $\geqslant k$.

Binary Search Trees

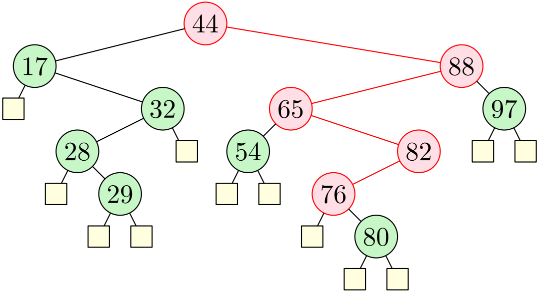


- What is the *inorder* traversal of this tree? Answer:
  □ 17 □ 28 □ 29 □ 32 □ 44 □ 54 □ 65 □ 76 □ 80 □ 82 □ 88 □ 97 □

Proposition

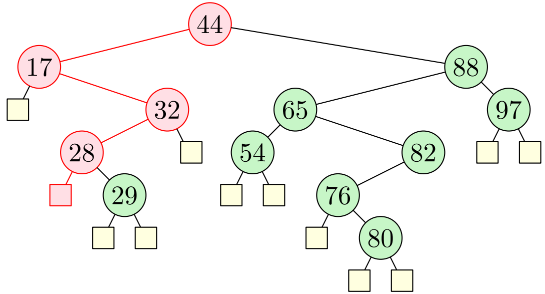
In the inorder traversal of a BST, the keys appear in non-decreasing order. Leaves and internal nodes alternate in this sequence.

Searching in a BST



- Nodes visited during the execution of `find`(76).
- The search was successful: We return the node containing 76.

Searching in a BST



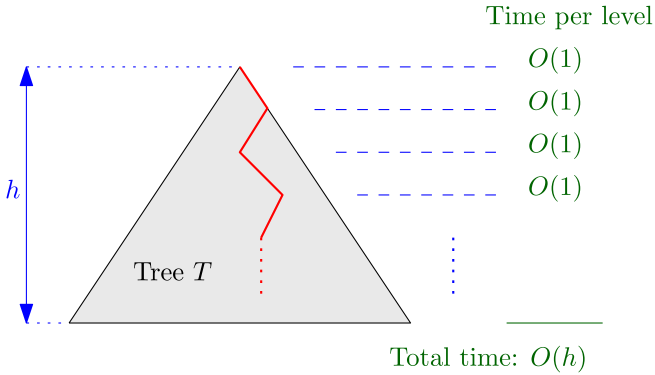
- Nodes visited during the execution of `find`(25).
- The search was unsuccessful: We return the leaf node corresponding to the position of 25 in the inorder traversal.

Searching in a BST

Pseudocode

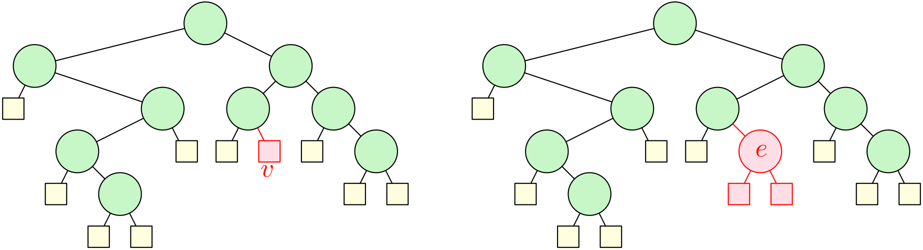
procedure $\textsc{TreeSearch}(k, v)$
  if T.isLeaf(v) **then**
    return v
  if $k <$ key(v) **then**
    return $\textsc{TreeSearch}(k, T.\text{left}(v))$
  if $k >$ key(v) **then**
    return $\textsc{TreeSearch}(k, T.\text{right}(v))$
  return v

Searching in a BST: Analysis



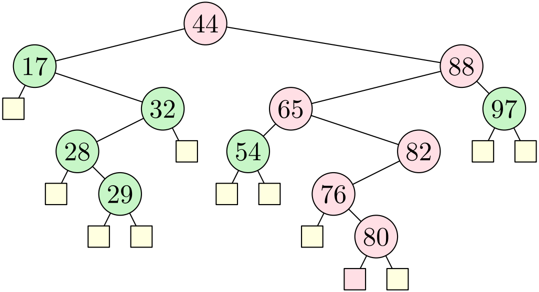
- So `find`(k) takes time $O(h)$, where h is the height of the tree.
- It can also be shown that `findAll`(k) takes $O(h+s)$ where s is the number of nodes that it returns.

Insertion



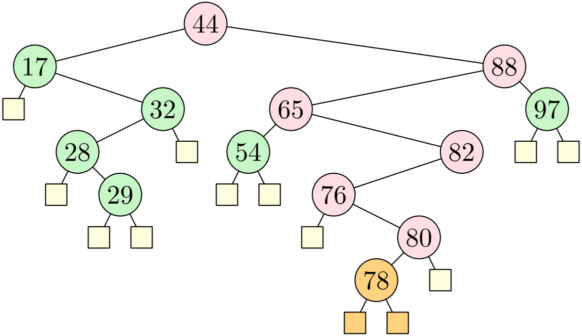
- We assume that we have a function **insertAtLeaf**(v, e) that expands a leaf into a subtree consisting of one internal node storing e and two leaves.

Insertion



- Inserting 78: We first find the position to insert.

Insertion



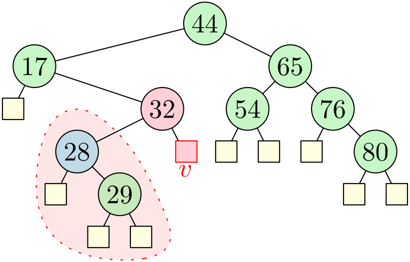
- Inserting 78.

Insertion

Pseudocode

procedure $\textsc{TreeInsert}(k,\ x,\ v)$
    $w \leftarrow \textsc{TreeSearch}(k, v)$
    if T.isInternal(w) **then**
        return $\textsc{TreeInsert}(k,\ x,\ T.\text{left}(w))$
    $T.\text{insertAtLeaf}(w, (k, x))$

Deletion

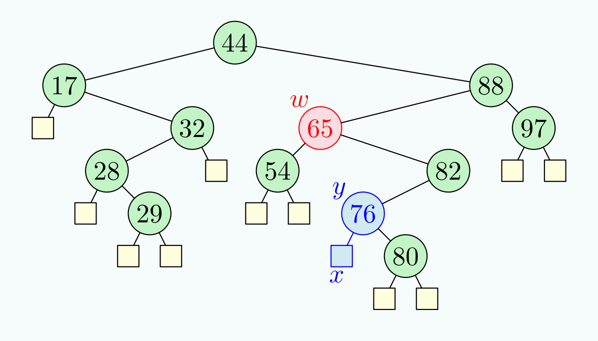


- **removeAboveLeaf**(v): Remove a leaf node v and its parent, replacing v's parent with v's sibling.

Deletion

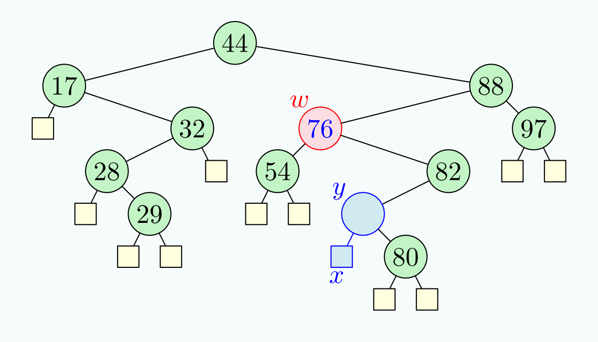
- We now show how to perform the operation `erase`(k), which delete a node with key k if there is one.
- We first perform a search to find a node w with key k.
- If at least one child of w is a leaf, we perform the operation from previous slide and we are done.
- Otherwise, we do as shown in the next slides:

Deletion



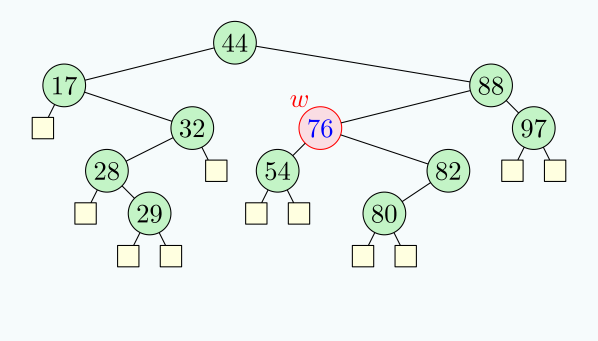
- Find the two nodes x and y that follow w in an inorder traversal.
- y is the leftmost internal node in the right subtree of w. It can be found by starting from the right child of w, and then following the left children.
- x is a leaf and y is its parent.

Deletion



- Move entry of y into w.

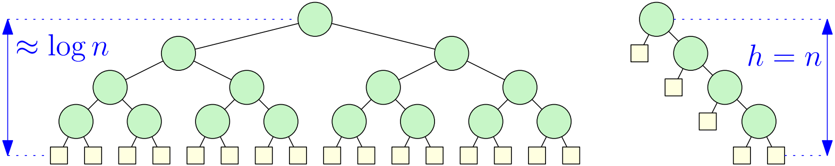
Deletion



- Remove x and y by doing `removeAboveLeaf(`x`)`.

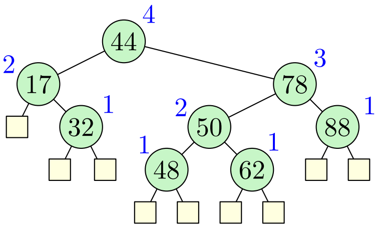
Performance of a Binary Search Tree

- `size` and `empty` take $O(1)$ time.
- `find`, `insert` and `erase` take $O(h)$ time.



- When T is a *complete* binary tree, we have $h = \lceil \log(n+1) \rceil$. This is the best case, the last three operations take $O(\log n)$ time.
- In the worst case, the internal nodes form a path, and $h = n$.
- We will now introduce *AVL trees*, which do not have this problem: Their worst-case height is $O(\log n)$.

AVL Trees



An AVL tree

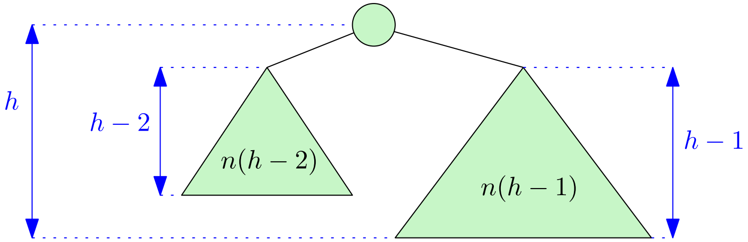
- A BST that satisfies the property below is called an *AVL Tree*.

Height-Balance Property

For every internal node v of T, the heights of the children of v differ by at most 1.

- It follows that a subtree of an AVL tree is also an AVL tree.

AVL Trees



Proposition

The height of h an AVL tree satisfies $h = O(\log n)$.

- We now prove this proposition. Let $n(h)$ denote the *minimum* number of nodes for an AVL tree with n internal nodes.

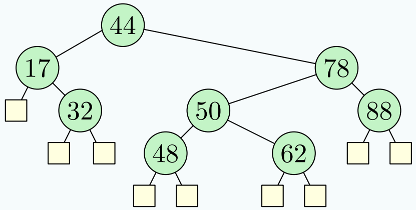
AVL Trees

- Then $n(h) = n(h-1) + n(h-2) + 1$.
- Base cases: $n(1) = 1$ and $n(2) = 2$.
- This is related to the Fibonacci sequence defined by $f_h = f_{h-1} + f(h-2)$, $f(1) = 1$ and $f(2) = 1$.
- From your calculus course,

$$f(h) = \Theta(\varphi^h) \quad \text{where } \varphi = (1+\sqrt{5})/2 \approx 1.618$$

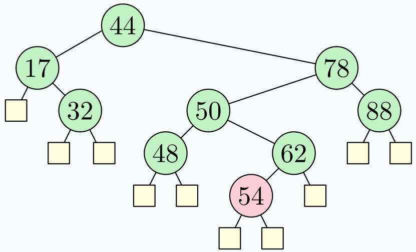
- We have $n(h) \geqslant f(h)$ for all h, so $n(h) = \Omega(\varphi^h)$.
- It follows that $n(h) \geqslant C\varphi^h$ for some constant C.
- Hence $\log n \geqslant \log(n(h)) \geqslant h \log(\varphi) + \log C$.
- Conclusion: $\log n = \Omega(h)$, which means that $h = O(\log n)$.

Insertion into an AVL Tree



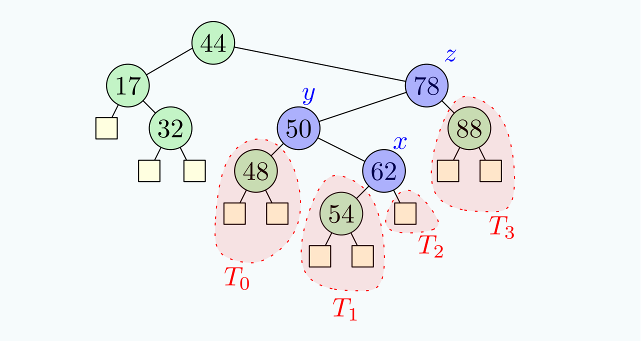
- AVL tree before insertion.

Insertion into an AVL Tree



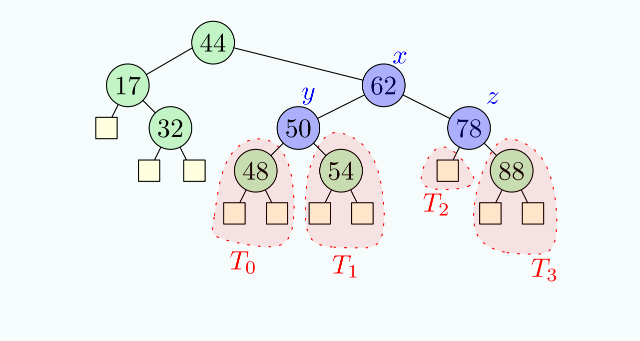
- Inserting 54. The tree is no longer an AVL tree. We need to fix it.

Insertion into an AVL Tree



- z is the lowest node in the insertion path that is unbalanced.

Insertion into an AVL Tree

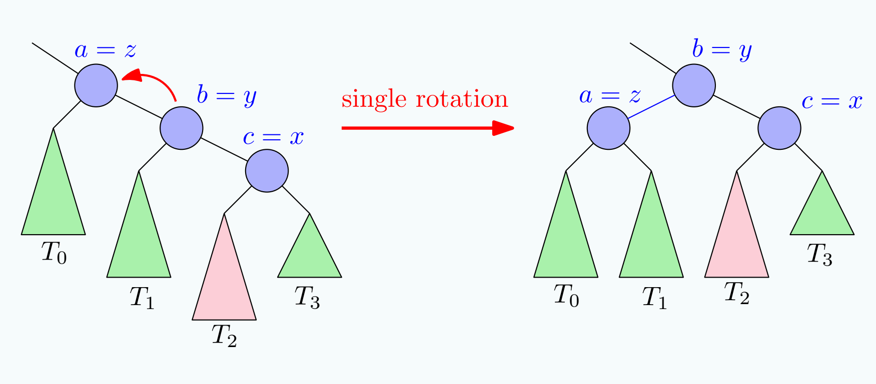


- The tree is now an AVL tree.

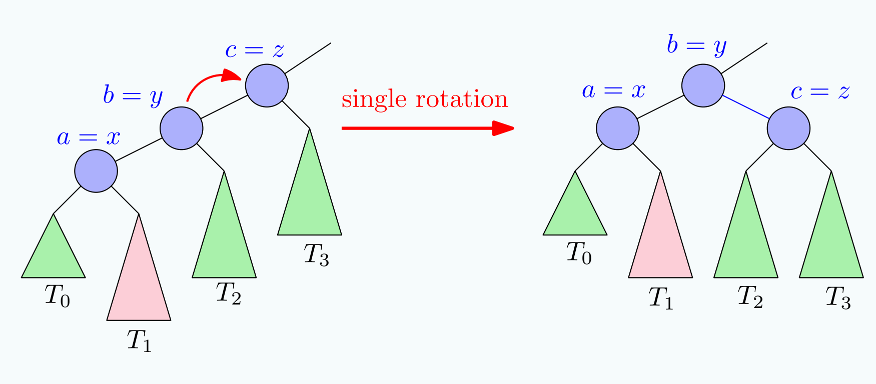
Insertion into an AVL Tree

- We first insert a node w in the same way as we did for ordinary BSTs.
- If the tree is still AVL, we are done. Otherwise, *restructure* the tree:
- Let z the first node on the path from w to the root that is unbalanced (i.e. heights of the two subtrees differ by at least 2).
- Let y be the child of z along this path, and x the child of y.
- Let $\{a, b, c\} = \{x, y, z\}$ such that $a < b < c$ in the inorder traversal.
- We partition the subtree rooted at z into nodes a, b, c and subtrees T_i such that T_0, a, T_1, b, T_2, c, T_3 appear in this order in the inorder traversal.
- Replace the subtree rooted at z with a subtree rooted at b, where a and c are the left and right child of b, respectively, and the T_i's are the subtrees rooted at the children of a and b.

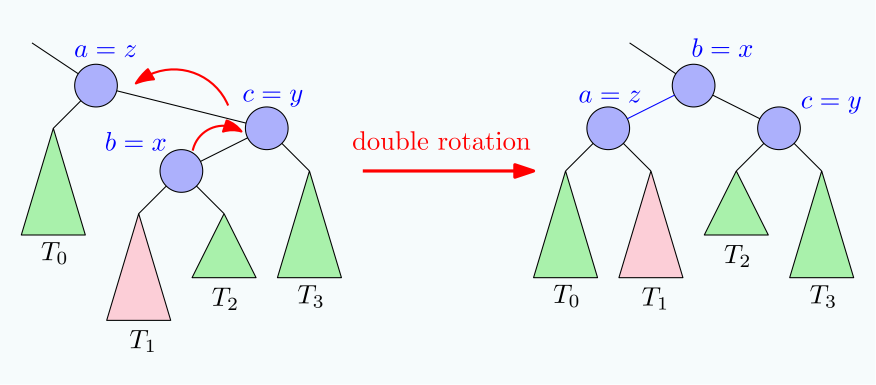
Restructuring Operations



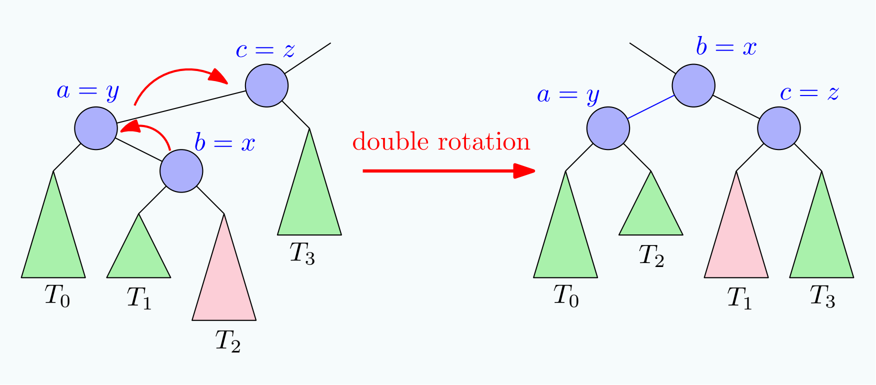
Restructuring Operations



Restructuring Operations



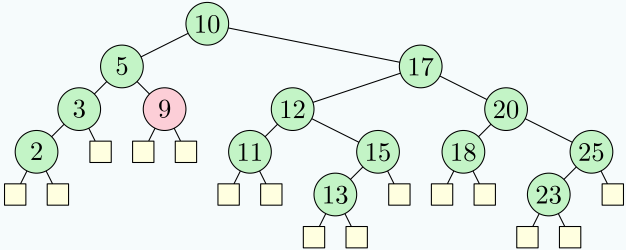
Restructuring Operations



Restructuring Operations

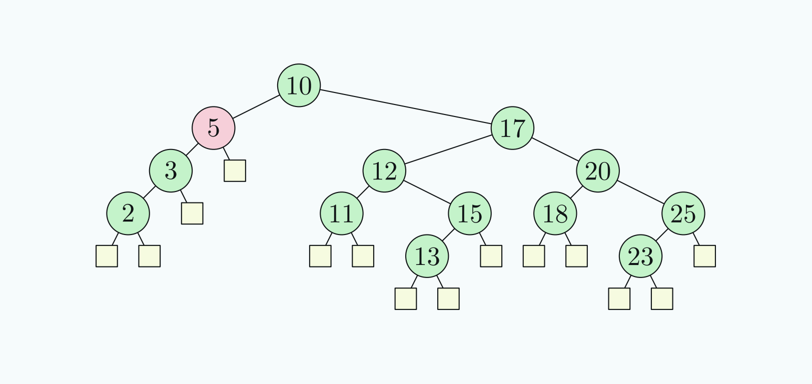
- After applying one of the 4 rebalancing operations above, all the AVL tree properties are restored.
- Same approach for deletion: First delete the node as we would do in an ordinary BST.
- Then rebalance the subtree rooted at z by performing one of the 4 operations above.
- Problem: The tree may become unbalanced at the parent of z. (See example in next slides.)
- So we rebalance at this parent node.
- We may have to do it at each node on the path from z to the root.
- It means $\Theta(\log n)$ restructuring operations in the worst case.

Restructuring Operations



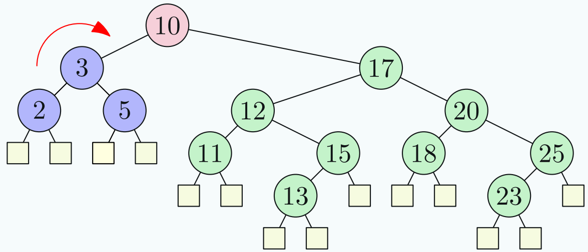
- An AVL tree.

Restructuring Operations



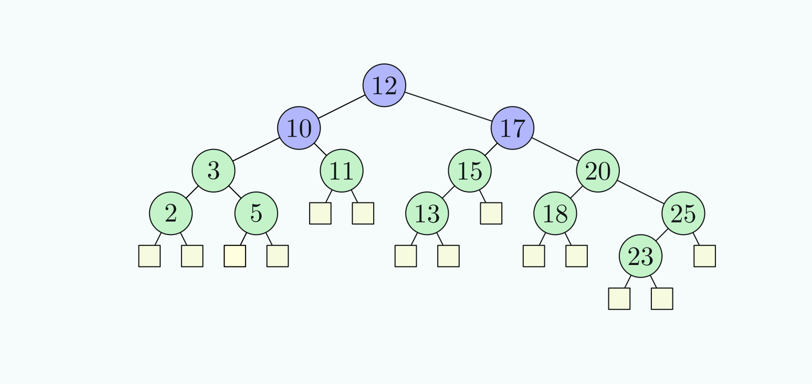
- After deleting 9, the tree is no longer AVL (at 5).

Restructuring Operations



- After a single rotation. The tree is still not AVL (at 10).

Restructuring Operations



- After a double rotation, it is an AVL tree.

AVL Tree Performance

- $O(1)$ time for empty, size.
- $O(\log n)$ time in the worst case for `find`, `insert` and `erase` because we perform at most one operation in constant time per level, and the height is $O(\log n)$.