Search Trees

- Last time
  - Binomial Heap
- Today
  - Review: Binary search tree, AVL tree
  - Splay tree

Motivation

- Efficient implementation of ordered dictionary
  - Methods
    - findElement(k)
    - insertItem(k)
    - RemoveItem(k)
  - Other methods
    - closestKeyBefore(k)
    - closestElemBefore(k)
    - closestKeyAfter(k)
    - closestElemAfter(k)

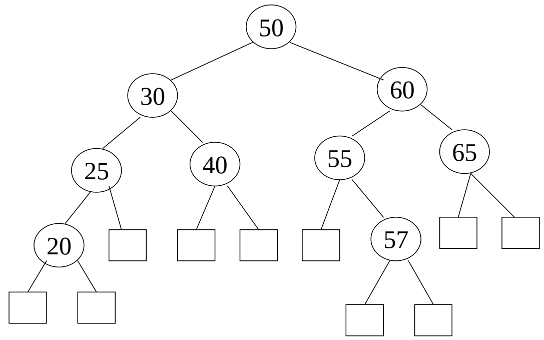
Sorted table

- Binary search: O(log n)
- Insertion: O(n)
- Deletion: O(n)
- ClosestKeyBefore: O(logn)

Binary search tree

- Definition:
  - A binary tree,
  - Where each internal node *v* stores an element *e*
  - The left subtree of v are <= e
  - The right subtree of v are >=e
- Assume all external nodes are empty
- The in-order traversal of binary search tree visits elements in non-decreasing order

A Binary Search Tree

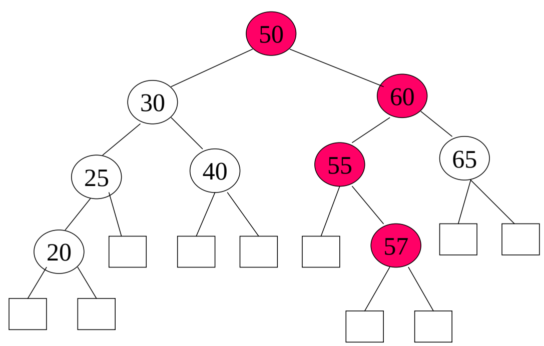


Search A Binary Search Tree

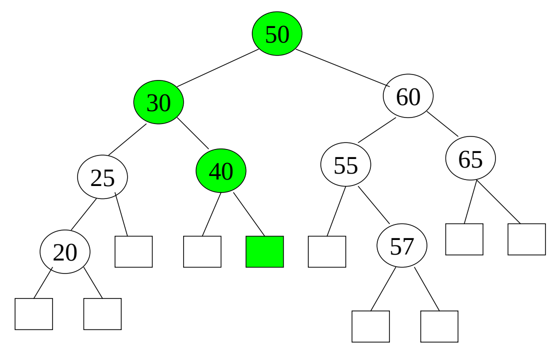
```
Node binaryTreeSearch(Key k, Node v)
// Parameters: k, key to search
//                    v, the root of the subtree to search
// return a node when found match key
// otherwise, return an external node
{
    if (v is an external node)
        return v;
    if (k == key(v))
        return v;
    else if (k < key(v))
        binaryTreeSearch(k, v.leftChild());
    else
        binaryTreeSearch(k, v.rightChild());
}
```

Cost? Best case? Worst Case?

A Binary Search Tree: search 57



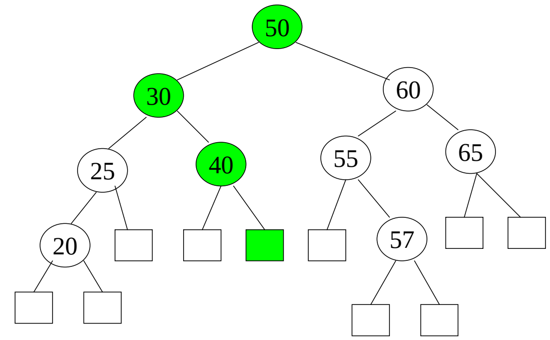
A Binary Search Tree: search 42



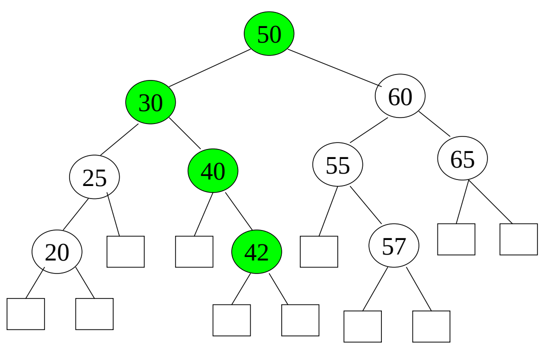
Insertion in a Binary Search Tree

- To insert element *e* with key *k*.
- Let *w* be the node returned by binaryTreeSearch()
  1. If *w* is an external node, replace it by an internal node with the key *k* and element *e*.
  2. If *w* is an internal node, continue to search its right subtree (or left subtree) until find an external node. Then apply case 1.

A Binary Search Tree: insert 42



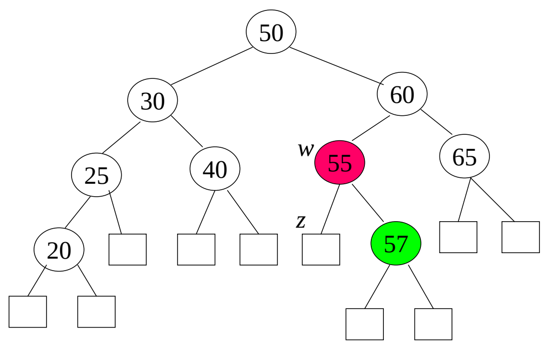
A Binary Search Tree: insert 42



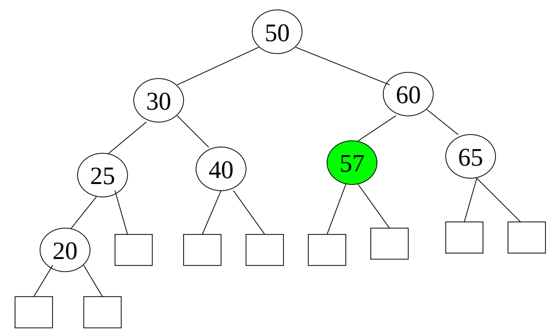
Removal in a Binary Search Tree

- To remove a node with key *k*, Let *w* be the node returned by binaryTreeSearch(k, root)
  1. If *w* is an external node, done!
  2. If *w* is an internal node
     a) One of *w*'s children is an external node, *z*. Remove *w* and *z*, and replace *w* by *z*'s sibling
     b) Both children of node *w* are internal nodes
        - Find internal node *y* that follows *w* in an inorder traversal
        - Replace *w*'s content by *y*'s.
        - Remove *y* using case (a).

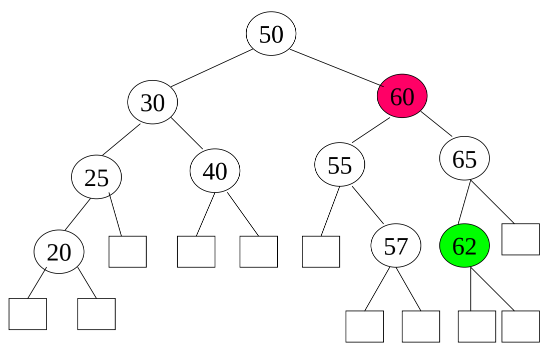
A Binary Search Tree: remove 55



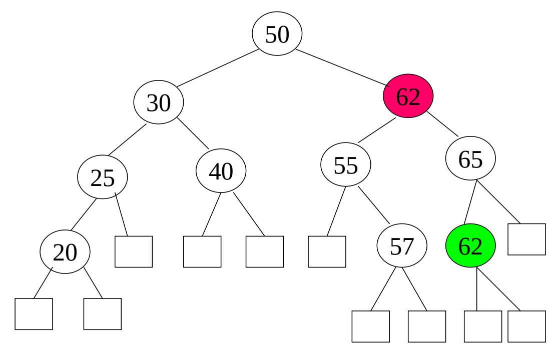
A Binary Search Tree: after removing 55



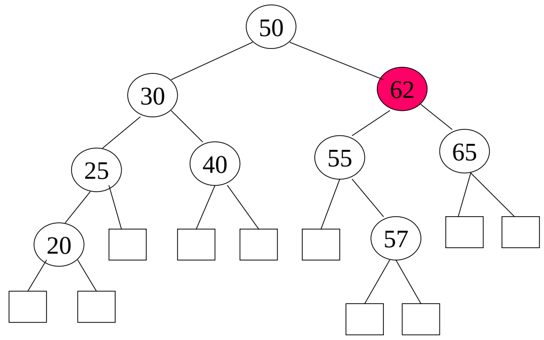
A Binary Search Tree: remove 60



A Binary Search Tree: remove 60



A Binary Search Tree: remove 60



AVL tree

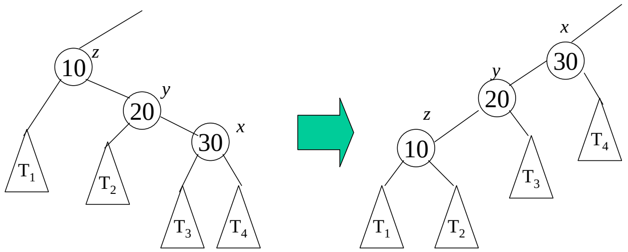
- Motivation
  - Worst case linear time for a binary search tree
  - Desire a height-balance tree so the height is in O(log n)
- AVL tree
  - A binary search tree that satisfies height-balance property
    - Height-balance property: for every internal node, the heights of the children can differ by at most 1
  - Height: O(log n)
- Need to maintain height-balance property when inserting or deleting

Splay Trees

- Apply *splaying* after every access to keep the search tree balanced in an amortized sense
- Splaying
  - Splay *x* by moving *x* to the root through a sequence of restructurings
  - One specific operation depends on the relative positions of *x*, its parent *y*, and its grandparent *z*
    - Zig-Zig
    - Zig-Zag
    - Zig

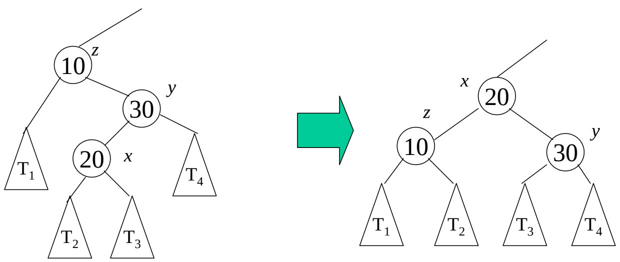
Splay *x*: zig-zig

The node *x* and its parent *y* are both left or right children



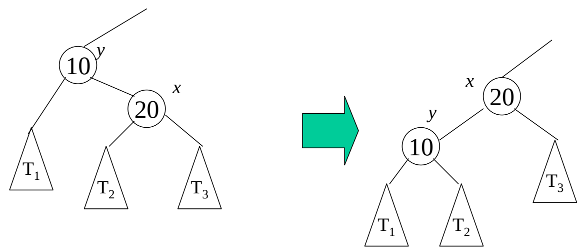
Splay x: zig-zag

One of x and y is a left child and the other is a right child.

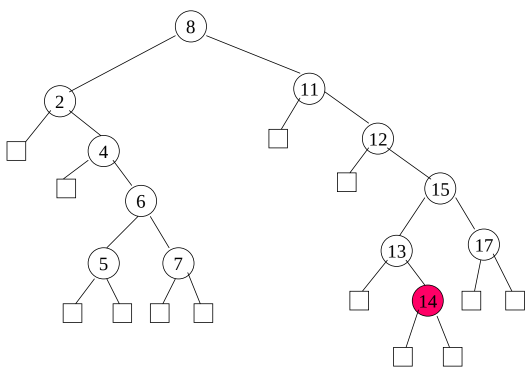


Splay x: zig

The node x does not have a grandparent (or the grandparent is not of our concern)

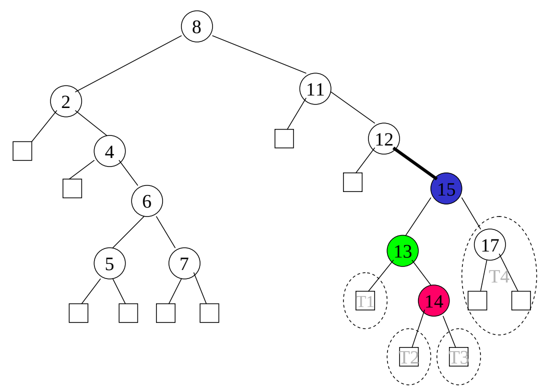


Example of Splaying a Node



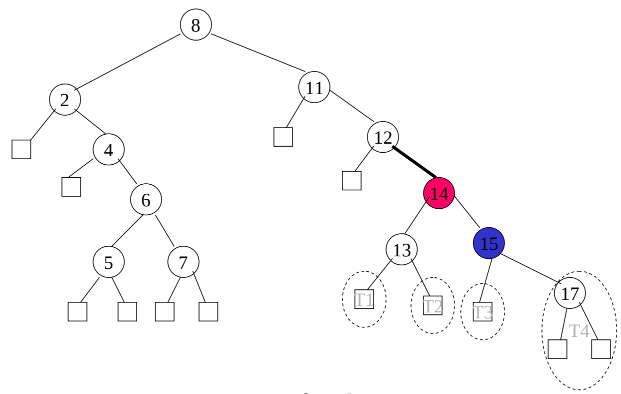
Splay 14

Example of Splaying a Node



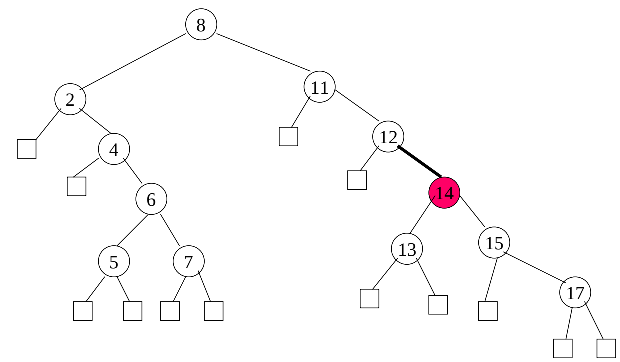
Zig-zag

Example of Splaying a Node



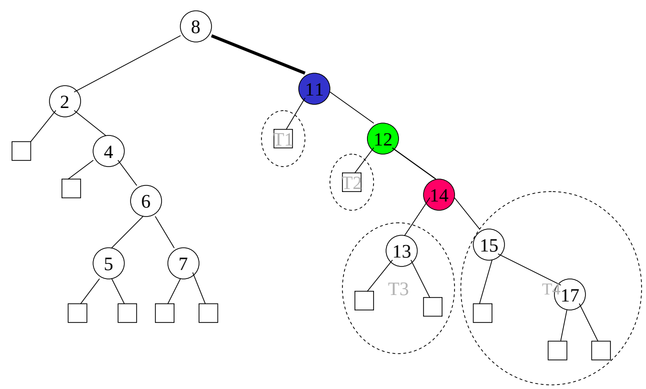
After the Zig-zag

Example of Splaying a Node



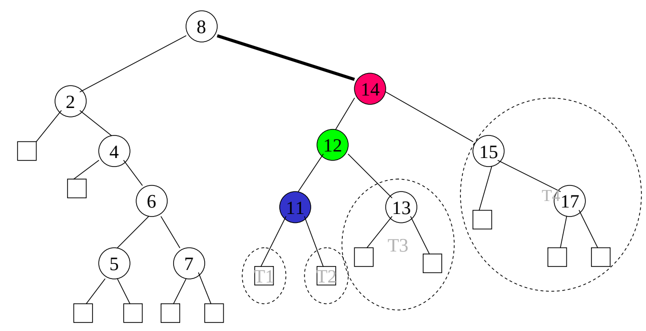
Splaying continues

Example of Splaying a Node



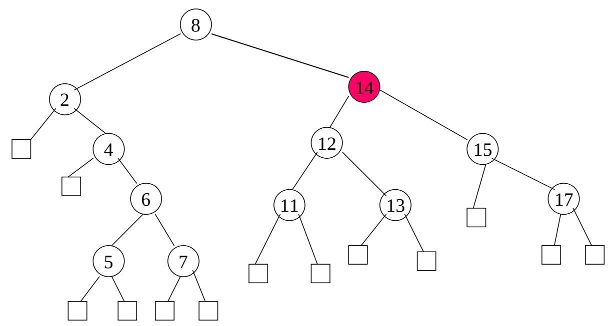
Zig-Zig

Example of Splaying a Node



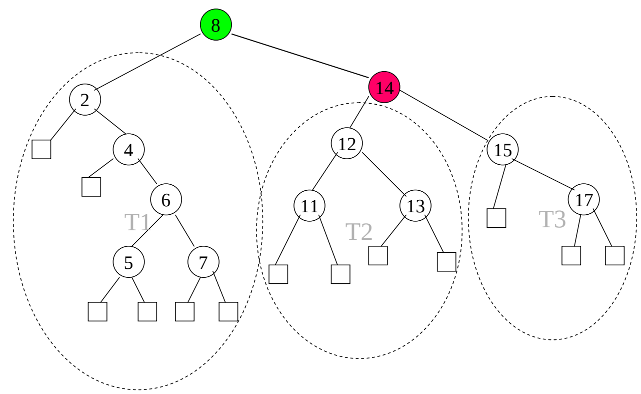
After the Zig-Zig

Example of Splaying a Node



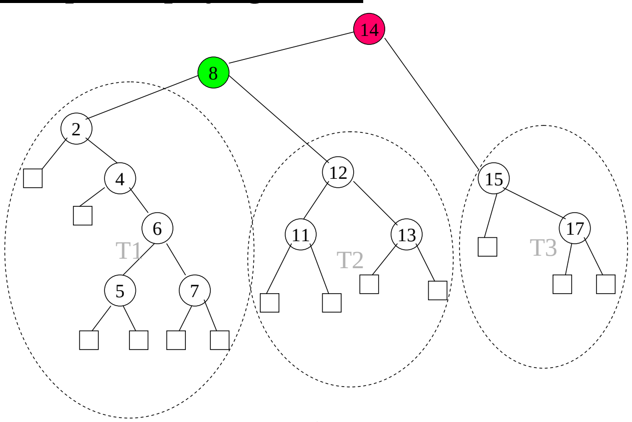
Splaying continues

Example of Splaying a Node



Zig

Example of Splaying a Node



After the Zig

When to Splay

- When searching for key *k*, splay the found internal node or the parent of the external node when search fails
- When inserting a key *k*, splay the newly created internal node
- When deleting a key *k*, splay the parent of the node that gets removed (See slide: Removal in a Binary Search Tree).

Properties of Splay Trees

- Linear depth when inserting keys in increasing order
  - What's the worst case cost for search, insertion, and deletion respectively?
- Consider a sequence of *m* operations on a splay tree, each a search, insertion, or deletion, starting from an empty tree with zero keys, also let n_i be the number of keys in the tree after operation *i*, and *n* be the total number of insertions. The total running time for performing the sequence of operations is

$$O(m + \sum_{i=1}^{m} \log n_i) = O(m \log n)$$

Properties of Splay Trees

- Consider a sequence of *m* operations on a splay tree, each a search, insertion, or deletion, starting from an empty tree with zero keys, also let *f(i)* be the number of times the item *i* is accessed in the splay tree, that is, its *frequency*, and let *n* be total number of items. Assuming that each item is accessed at least once, then the total running time for performing the sequence of operations is

$$O(m + \sum_{i=1}^{m} f(i) \log(m / f(i)))$$