Extended Introduction to Computer Science CS1001.py

Lecture 17A:
Binary Search Trees

Instructors: Benny Chor, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Amir Gilad

School of Computer Science
Tel-Aviv University
Spring Semester 2017
http://tau-cs1001-py.wikidot.com

Lecture 15-16-17-18-19 : Highlights

- OOP (lec 15)

- Data Structures
  1. Linked Lists (lec 16)
  2. Binary Search Trees (lec 17)
  3. Hash tables (lec 17-18)
  4. Iterators and generators (lec 19)

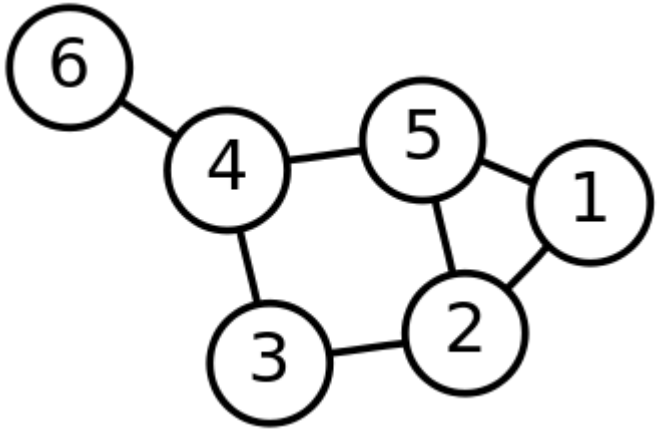
Linked data structures

- Linked lists are just the simplest form of linked data structures, in which pointers are used to link objects.
- We can create structures of other forms.
- For example doubly-linked lists, whose nodes include a pointer from each item to the preceding one, in addition to the pointer to the next item.
- Another linked structure is binary trees, where each node points to its left and right child. We will see it now and also how it may be used to store and search data.
- Another linked structure is graphs (probably not in this course).

Graphs

- A graph is a structure with Nodes (or vertices) and Edges. An edge connects two nodes.
- In directed graphs, edges have a direction (go from one node to another).
- In undirected graphs, the edges have no direction.

Example: undirected graph.
Drawing from wikipedia



Trees

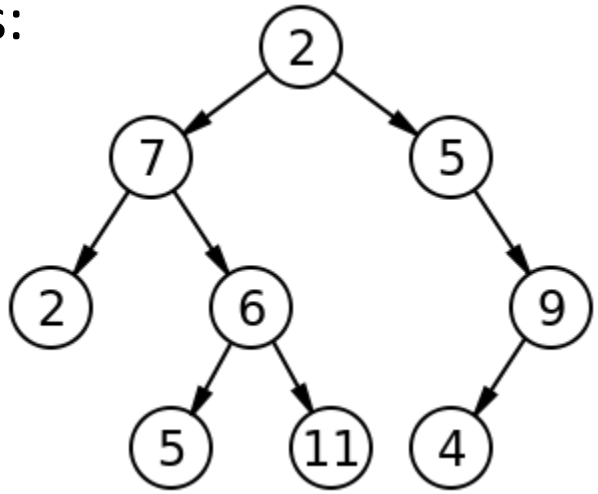
- Trees are useful models for representing different physical or abstract structures.

- Trees may be defined as a special case of graphs. The properties of graphs and trees are discussed in the course Discrete Mathematics (and used in many courses).

- We will only discuss a common form of so called (rooted) trees.

Rooted Binary Trees

- Definition: a rooted binary tree
  - contains no nodes (empty tree), or
  - is comprised of three disjoint sets of nodes:
    - a root node,
    - a binary tree called its left subtree, and
    - a binary tree called its right subtree



Adapted from wikipedia

- Note that this is a recursive definition.

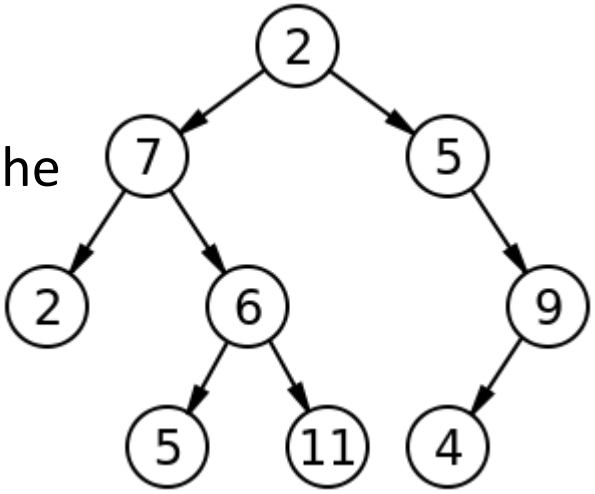
- Rooted binary trees are a special case of rooted trees, in which each node has at most 2 children.

Rooted Trees – Basic Notions

- A **directed edge** refers to the edge from the parent to the child (the arrows in the picture of the tree)
- The root node of a tree is the (unique) node with no parents (usually drawn on top).
- A leaf node has no children.
- Non leaf nodes are called internal nodes.

- The depth (or height) of a tree is the length of the longest path from the root to a node in the tree. A (rooted) tree with only one node (the root) has a depth of zero.

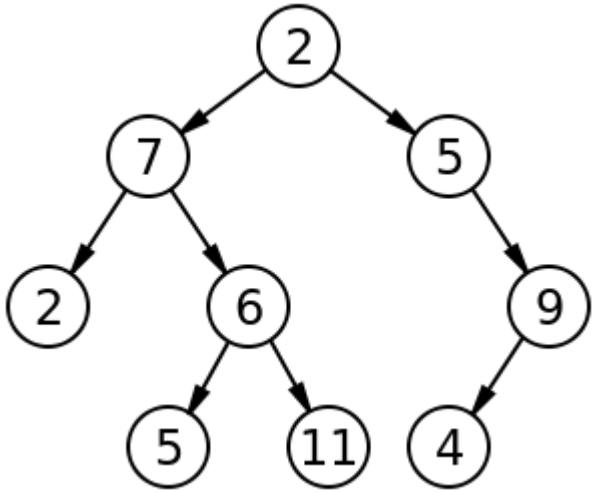
- A node *p* is an ancestor of a node *q* if *p* exists on the path from the root node to node *q*.
- The node *q* is then termed as a descendant of *p*.
- The out-degree of a node is the number of edges
- leaving that node.
- All the leaf nodes have an out-degree of 0.



Adapted from wikipedia

Example Binary Tree

- Here the root is labeled 2. the depth of the tree is 3. Node 11 is a descendent of 7, but not of (either of the two nodes labeled) 5.



Drawing from wikipedia

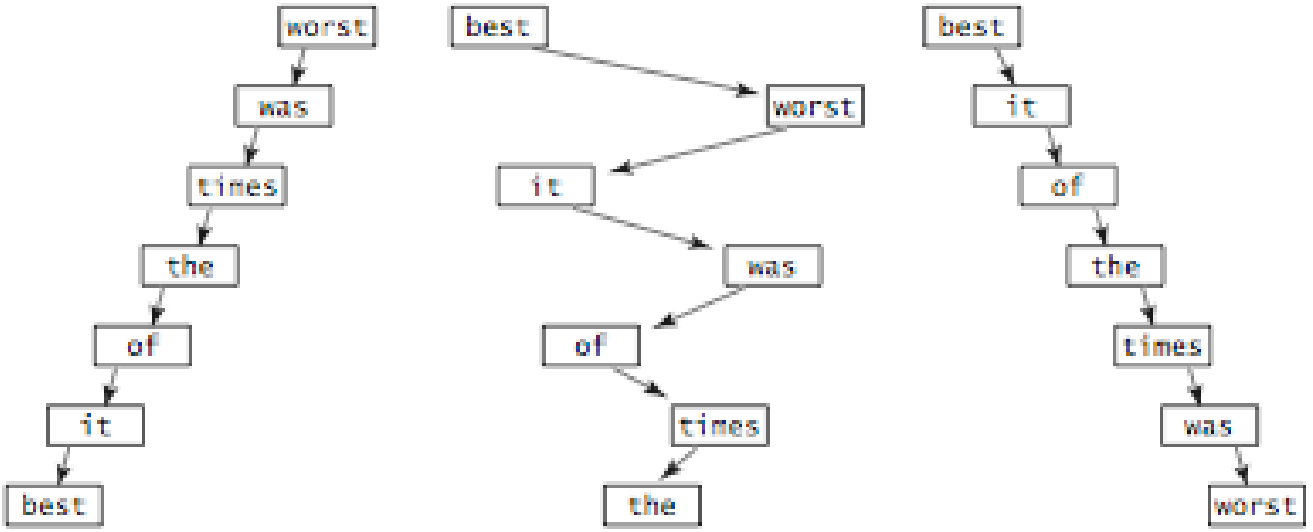
Properties of Binary Trees

- The number of nodes n in a binary tree of depth d satisfies $d + 1 \leq n \leq 2^{d+1} - 1$

- We will see the two extreme cases in the next slides.

Totally unbalanced binary tree

Each node has only one non empty subtree. The depth of such a totally unbalanced tree with n nodes is $d = n - 1$

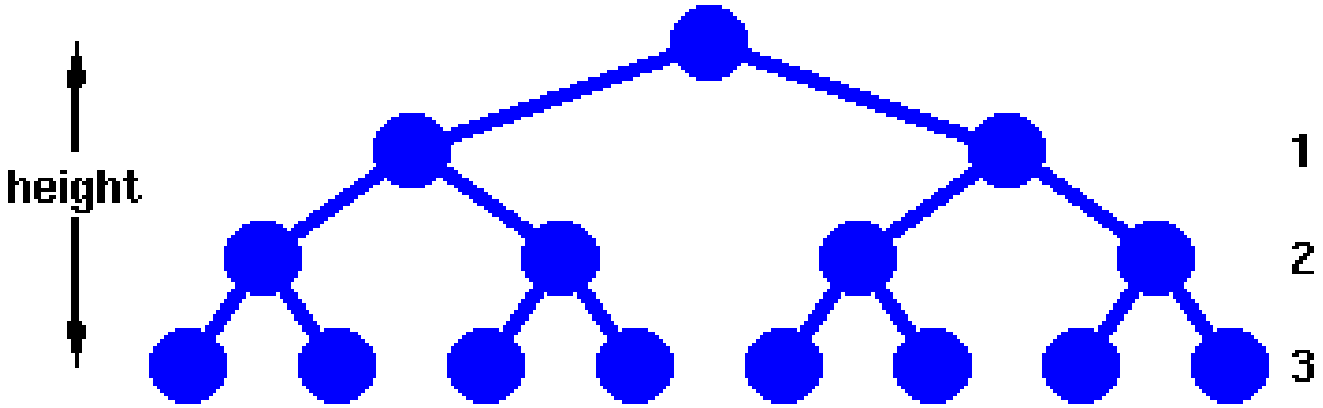


Worst case (totally unbalanced) BSTs

Drawing from
http://introcs.cs.princeton.edu/java/44st/images/bst-worst.png

Complete (Rooted) Binary Trees

- When all the leaves are in the same level, and all internal nodes have degree exactly 2, we call the tree complete.
- $n = \sum_{i=0}^{d} 2^i = 2^{d+1} - 1$
- how many leaves? internal nodes?



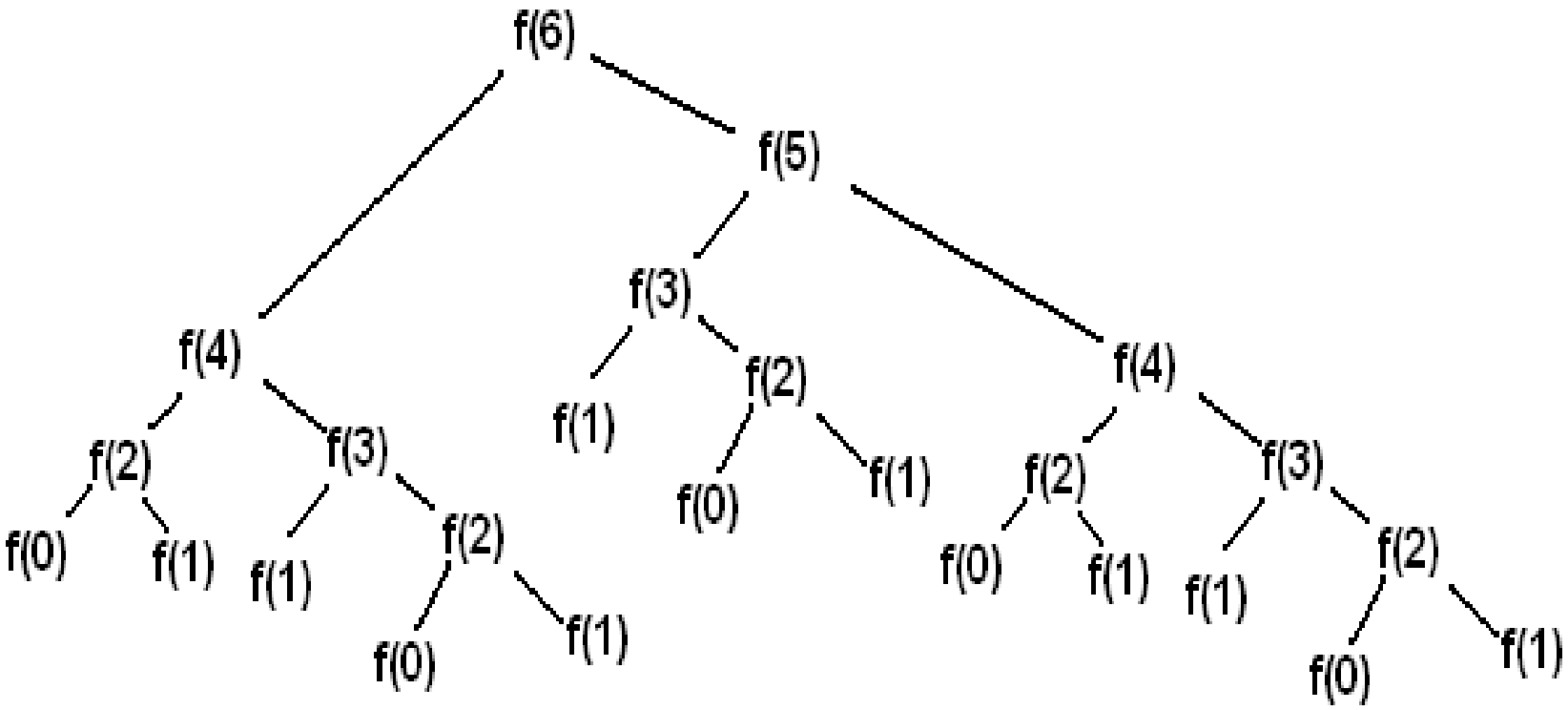
Drawing from
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Gifs/completeTree.gif

Applications of Trees

- Trees may be used to represent:
  - arithmetic expressions
  - games
  - evolution
  - recursion
  - and much more.

- They can also represent the execution of programs: the nodes represent individual function executions, and the edges show the calls invoked from a given execution. A recursive function will appear multiple times in the same graph. It is a useful tool for visualizing algorithms, and also useful in the context of compilers.

Example: Call Tree for Computing Fibonacci *f*(n)



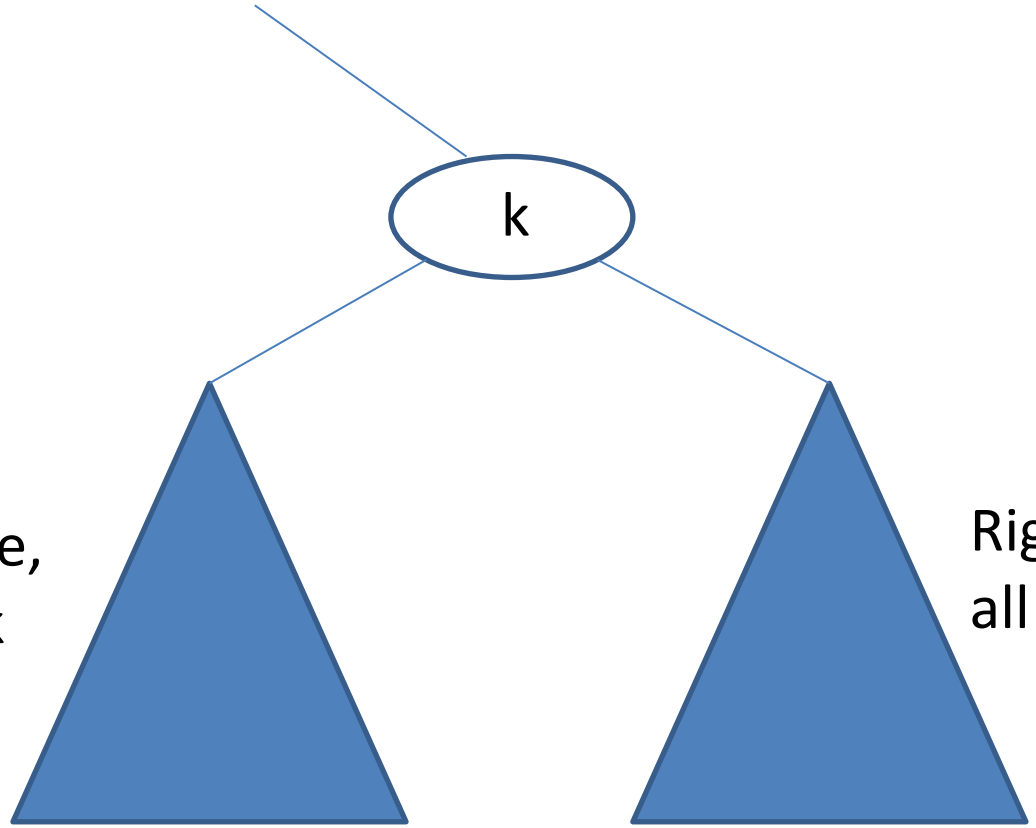
Drawing from http://i.stack.imgur.com/oPTFd.png

Binary Search Trees

- Binary search trees are data structures used to represent collections of data items.
- They support operations like insert, search, delete, etc.
- Each node in a binary search contains a single data record. We will assume each record consists of a key and value (for example, keys are IDs, and values are names.)
- A node will also include pointers to its left and right subtrees.
- The keys in the binary search tree are organized so that every node satisfies the property shown in the next slide.

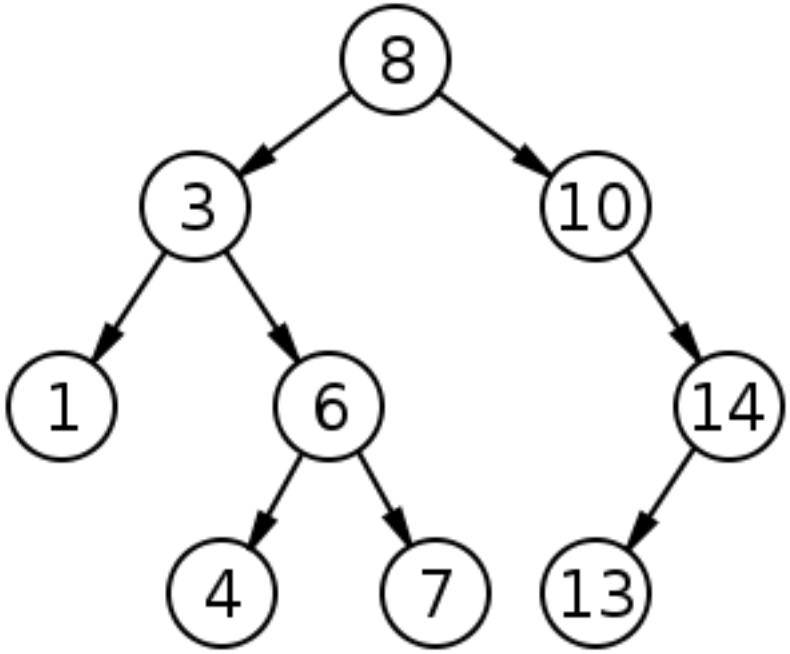
Binary search property

For each node, all the keys in the left/right subtrees are smaller/larger than the key in the current node, respectively.



Example binary search trees

Only the keys (and the left and right pointers) are shown. The nodes also contain the value associated with the key, but these are not shown here.

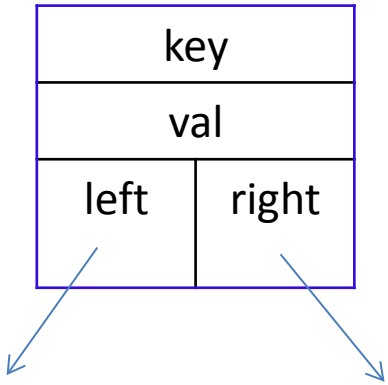


Drawing from wikipedia

Binary Search Tree: Python code

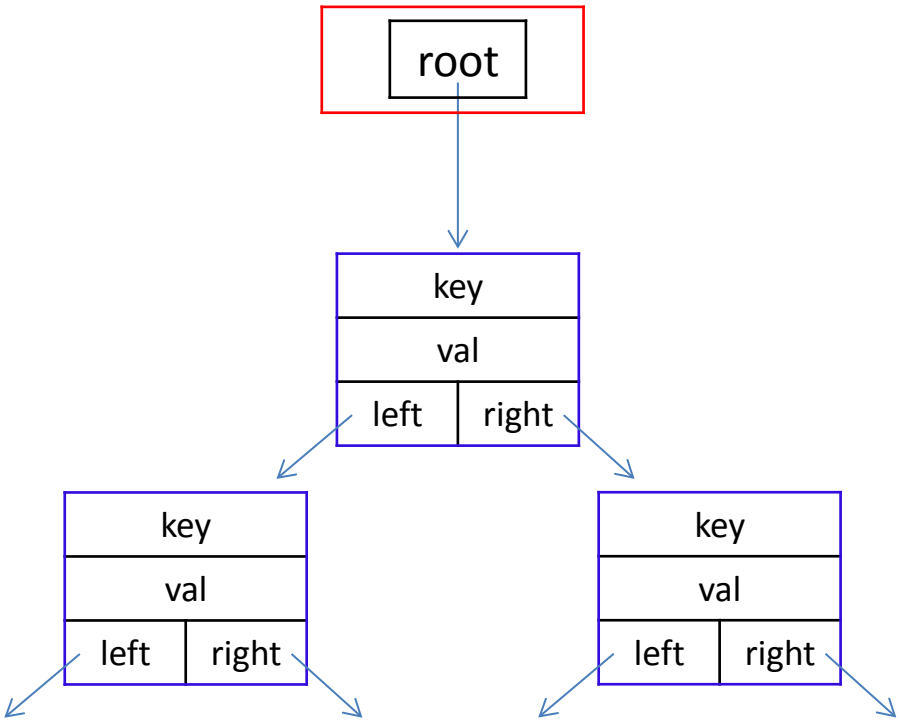
A tree node will be represented by a class Tree_node:

```
class Tree_node():
    def __init__(self, key, val):
        self.key    = key
        self.val    = val
        self.left   = None
        self.right = None

    def __repr__(self):
        return str(self.key) + ":" + str(self.val)
```



Binary Search Tree class

```
class Binary_search_tree():
    def __init__(self):
        self.root = None
```



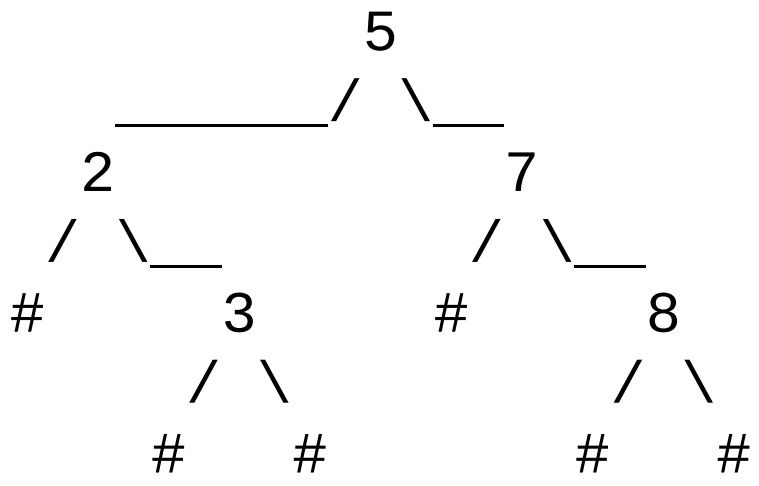
Binary Search Tree __repr__

- Representation of a tree can be done recursively.
- We will use a rather sophisticated implementation donated by a former student in our course – Amitai Cohen.
- You do not need to understand it.
- This implementation appears in the file printree.py.

```
from printree import *


class Binary_search_tree():
    def __repr__(self):
        out = ""
        for row in printree(self.root): #need printree.py file
            out = out + row + "\n"
        return out
```

Binary Search Tree __repr__

```
>>> t = Binary_search_tree()
>>> t.insert(5,"a")
>>> t.insert(2,"b")
>>> t.insert(3,"c")
>>> t.insert(7,"d")
>>> t.insert(8,"e")
>>> print(t)
```



Will see insert right away.

Ain't it cool?

Binary Search Tree: lookup

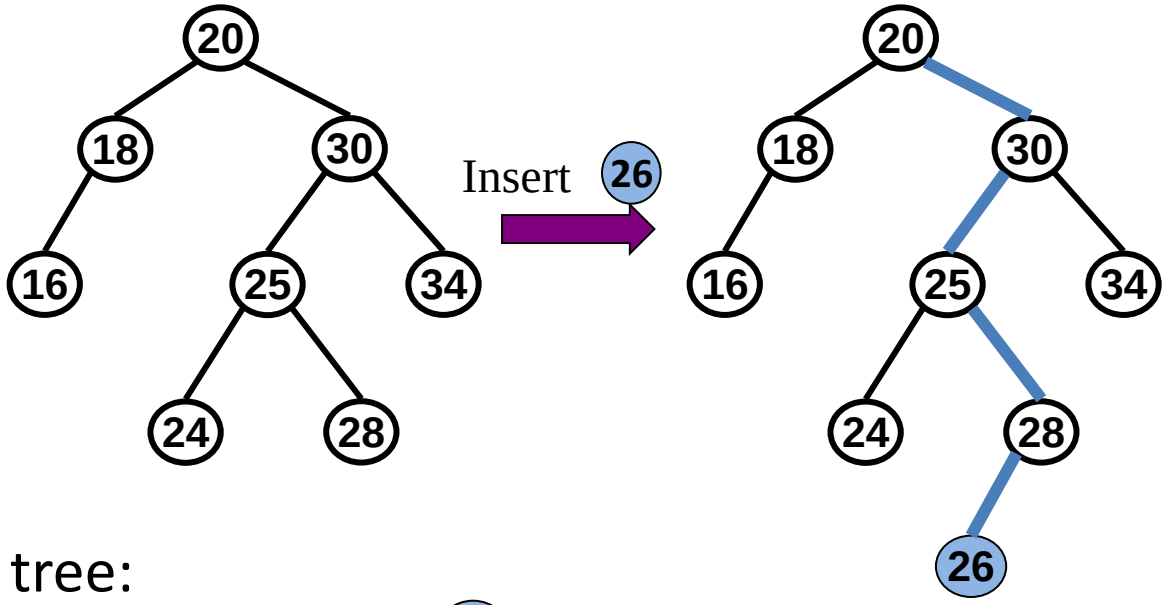
- lookup is written as an envelope function that calls a recursive function.

```
def lookup(self, key):
        ''' return node with key, uses recursion '''

        def lookup_rec(node, key):
            if node == None:
                return None
            elif key == node.key:
                return node
            elif key < node.key:
                return lookup_rec(node.left, key)
            else:
                return lookup_rec(node.right, key)



        return lookup_rec(self.root, key)
```

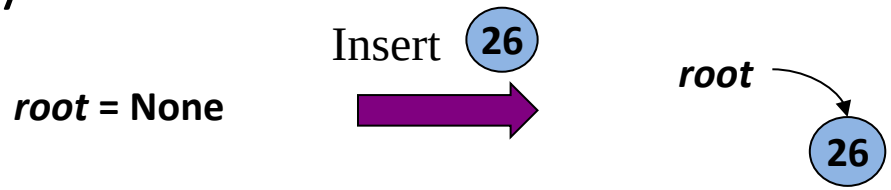
Binary Search Tree: insert

We first look for the appropriate location for insertion, and then hang the new node as a leaf.

For example:



Inserting into empty tree:



Binary Search Tree: insert

- Insert is written as an envelope function that calls a recursive function.
- The return value is used to update the left or right pointer that was previously None.
- If the user inserts an element whose key already exists in the tree, we assume that it should replace the one in the tree.

Binary Search Tree: insert

```
def insert(self, key, val):
        ''' insert node with key,val into tree, uses recursion '''

        def insert_rec(node, key, val):
            if node == None:
                return Tree_node(key, val)
            else:
                if key == node.key:
                    node.val = val     # update the val for this key
                elif key < node.key:
                    node.left =  insert_rec(node.left, key, val)
                elif key > node.key:
                    node.right = insert_rec(node.right, key, val)
                return node

        if self.root == None: #empty tree
            self.root = Tree_node(key, val)
        else:
            insert_rec(self.root, key, val)
```

Complexity for lookup / insert

- To analyze the time complexity of tree operations, we should first consider the shape of the tree.
- In the worst case, operations may have to traverse the whole depth of the tree. So the depth of the tree is an important factor in its performance.
- In particular, given the size of a tree n (the number of elements stored in the tree == the number of nodes), the question is what is the depth d of the tree as a function of n. The smaller the better.

- Worst case: when the tree is totally unbalanced, d = n-1 = O(n).
- Best case: we have seen that when the tree is perfectly balanced $n = 2^{d+1} - 1$, so d = O(log n).
- The depth is O(log n) for trees that are not perfectly balanced, but are close to it. We call them balanced trees (you will meet them again in the Data Structures course).

Binary Search Tree: lookup time complexity

- The lookup algorithm follows a path from the root to the node where the element is found, or (when the element is not found) to a leaf.
- The time complexity of lookup is the length of the path from the root to the element we are looking for.
- The best case occurs for example when the element we are looking for is in the root. So the best case time complexity is O(1) (and is not dependent on the shape of the tree).
- The worst case occurs when we have to traverse a path from the root to the farthest leaf. So the time complexity is the depth of the tree,
  - when the tree is totally unbalanced, it is O(n).
  - in balanced trees it is O(log n).

Binary Search Tree: insert time complexity

The insert algorithm is similar to lookup.

The best case is O(1) , for example when the element to be inserted is found in the root of the tree.

In a balanced tree, the worst case time complexity of insert is O(log n).

In arbitrary trees, the worst case time complexity of insert is O(n).

Binary Search Tree: insert and lookup time complexity

	best case	worst case for any tree	worst case for balanced trees
insert	O(1)	O(n)	O(logn)
lookup	O(1)	O(n)	O(logn)

Binary Search Tree: Executions

- in class

Binary Search Tree: minimum

- To compute the minimum key in a tree, we need to go all the way to the left. For this, we maintain two pointers. (Alternatively, we could write a recursive function)

```
def minimum(self):
        ''' return node with minimal key '''
        if self.root == None:
            return None
        node = self.root
        left = node.left
        while left != None:
            node = left
            left = node.left
        return node
```

- complexity?

Binary Search Tree: time complexity of min

The time complexity of min is the length of the path from the root to the leftmost node.

The best case occurs when the left subtree is empty (the left pointer in the root is None). In this case, the smallest item is at the root. The best case time complexity is O(1).

The worst case occurs in a totally unbalanced tree in which all right subtrees are empty, (the tree is a “left chain”) so the length of path to the minimum is n-1, so the time complexity is O(n). The worst case in a balanced tree is O(log n).

Binary Search Tree: depth

- To compute the depth, we use a recursive function.
- Note that this is a non-linear recursion (two recursive calls).
- A tree with just a root node has depth 0, so that by convention an empty tree has depth -1

```
def depth(self):
    ''' return depth of tree, uses recursion'''
    def depth_rec(node):
        if node == None:
            return -1
        else:
            return 1 + max(depth_rec(node.left), depth_rec(node.right))

    return depth_rec(self.root)
```

Complexity: depth

- Time complexity is always the size of the tree O(n).

- This follows from the observation that every node is visited once, with O(1) time spent.

- We can also write a recurrence relation:

$$T(n) = 1 + T(n_l) + T(n_r)$$

where n_l and n_r are the sizes of the left and right trees, so that $n_l + n_r + 1 = n$, and $T(0) = 0$.

The solution for this formula is $T(n) = O(n)$ (no proof).

Binary Search Tree: size

- Computing the size is similar to the depth. Again a recursive function.

```
def size(self):
    ''' return number of nodes in tree, uses recursion '''
    def size_rec(node):
        if node == None:
            return 0
        else:
            return 1 + size_rec(node.left) + size_rec(node.right)

    return size_rec(self.root)
```

- Time complexity = O(n), same as depth.

Binary Search Tree: Concluding Remarks

- A function to delete a node is a little harder to write, and is omitted here.
- We can ask what the average case time complexity of lookup and insert is – this will be dealt with in the Data Structures course.
- If we are able to insure that the tree is always balanced, we will have an efficient way to store and search data. But we can observe that the shape of the tree depends on the sequence of inserts that generated the tree.
- It turns out that there are several variations of balanced binary search trees, such as AVL trees, and Red and Black trees, that insure that the tree remains balanced, by performing balancing operations each time an element is inserted (or deleted). This will also be taught in the Data Structures course.

Self-balancing trees (for reference only)

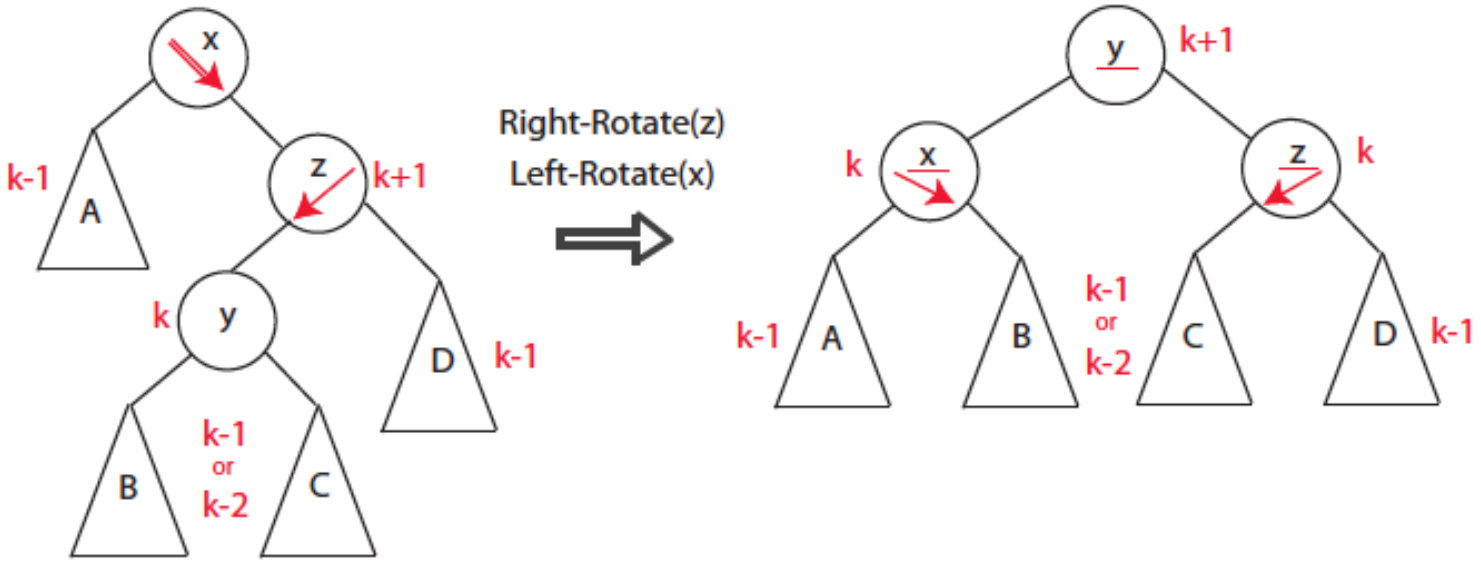


Figure from MIT algorithms course, 2008. Shows item insertion in an AVL tree.