

Extended Introduction to Computer Science

CS1001.py

Chapter G
Lecture 15



Data Structures 2:
Binary Search Trees

Michal Kleinbort, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
<http://tau-cs1001-py.wikidot.com>

* Slides based on a course designed by Prof. Benny Chor

Data Structures

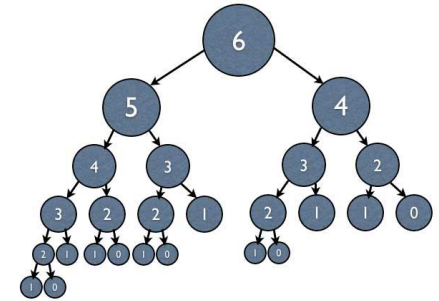
1. Linked Lists 
2. Binary Search Trees 
3. Hash Tables
4. Generators

Lecture Plan

- Trees, binary trees, and binary search Trees
- Operations: `Insert` and `Lookup (search)`
- Implementation of `class Binary_search_tree`
- Additional operations: `minimum`, `depth`

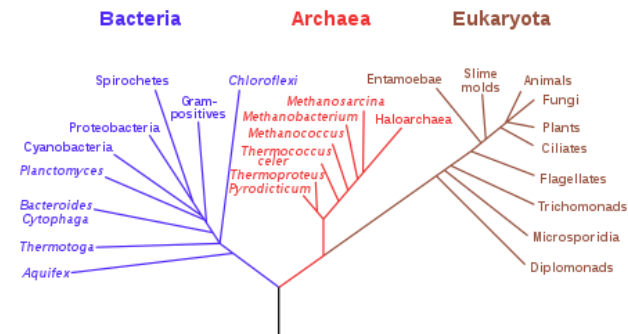
Applications of Trees

- We have seen trees earlier in the course, illustrating the execution of recursive functions.

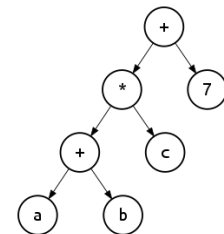


- Trees are extensively used for many applications, such as:

- Illustration of **program flow** (as above)
- Representation of **arithmetic expressions**
- Illustration of **games**
- **Evolutionary** processes (e.g. tree of life)
- ...



- We will now explore how trees can be used to **store** and **search data**

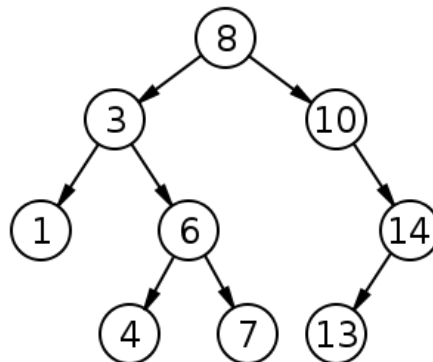


Linked Data Structures

- **Linked lists** are just the simplest form of linked data structures, in which pointers are used to link objects **linearly**.



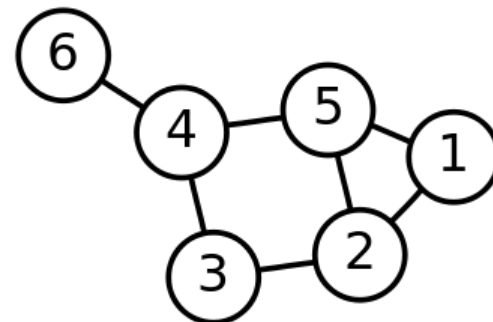
- Another linked structure is a **binary search tree**, where each element points to its **left** and **right** child, corresponding to **smaller** and **larger** elements, respectively.



Graphs and Trees

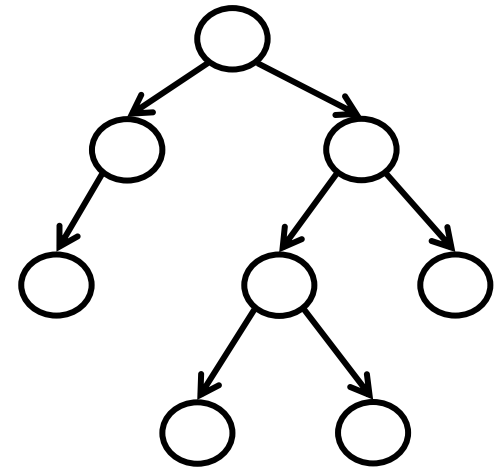
- A **graph** is a structure containing **nodes** (or vertices) and **edges**. An edge connects two nodes.
- In **directed graphs**, edges have a direction (go from one node to another). In **undirected graphs**, the edges have no direction.
- **Trees** may be defined as a special case of **graphs**. This is discussed in the course **Discrete Mathematics** (and used in many other, most notable **Algorithms**).
- Here, we will only discuss a common form of trees called **rooted binary trees**, which will be defined next, using recursion.
- From now on we will simply use the term **tree** instead.

Example: undirected graph.
Drawing from wikipedia



Rooted Binary Trees - Definition

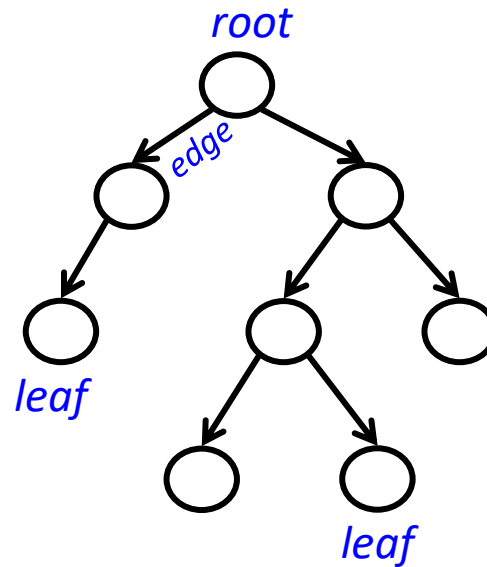
- A **rooted binary tree**
 - contains no nodes (empty tree), or
 - is comprised of three **disjoint** sets of nodes:
 - a **root** node,
 - a binary tree called the **left subtree**, and
 - a binary tree called the **right subtree**
- Note that this is a **recursive definition**.



- Rooted binary trees are a special case of the more general **rooted trees**, in which each node may have **more than just 2 children**.

Rooted Binary Trees – Basic Notions

- An **edge** refers to the directed link from parent to 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**.



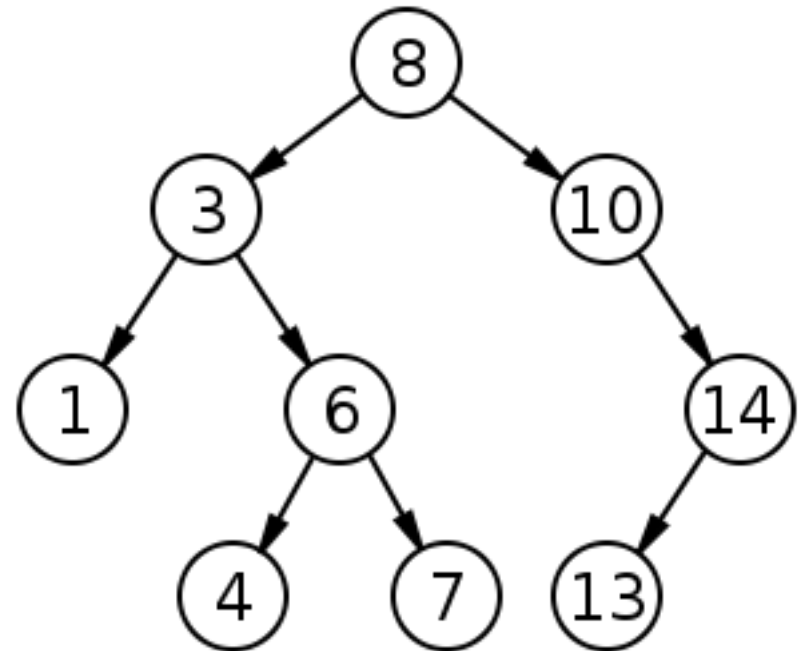
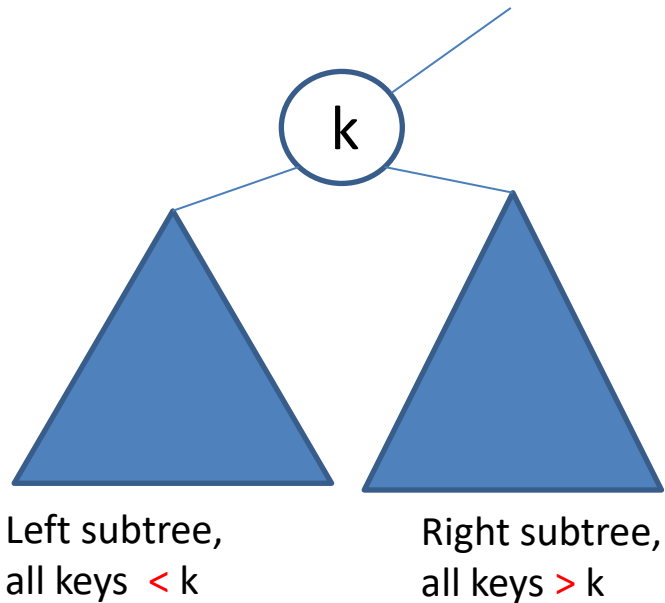
- A node p is an **ancestor** of a node q if p exists on the path from the root node to node q . The node q is then termed as a **descendant** of p .
- A **subtree** of a tree is a tree rooted at a child of the root.

Binary Search Trees

- Binary **search** trees are data structures used support operations like **insert**, **search**, **delete**, and other operations.
- Like in a linked list, each node in a binary search tree contains a **single data record**.
- We assume each data record has a **unique identifier**, called the **key**. Two nodes cannot have the same key.
- The keys are organized so that every node satisfies the **order property** shown in the next slide.

Binary Search Tree Property

- For each node, all the keys in the **left/right** subtrees are **smaller/larger** than the key in the current node, respectively.
- Recall we assume **keys are unique** (no repetitions)

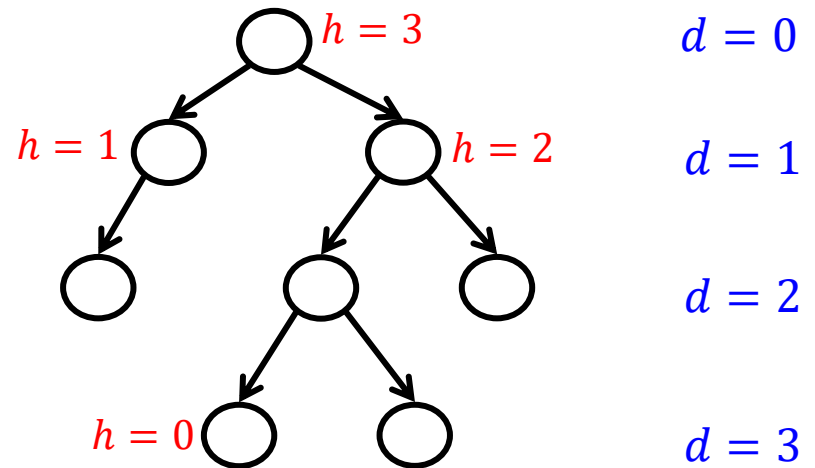


Demos of Insertion and Search

- Simulation:
<https://www.cs.usfca.edu/~galles/visualization/BST.html>
- Gif:

Depth/ Height

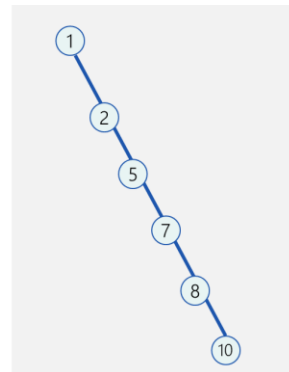
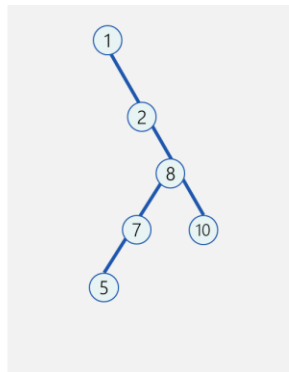
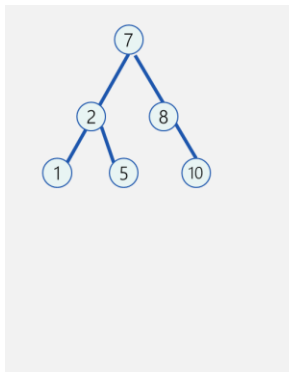
- The **depth of a node** (denoted d) is the number of edges from the root to it.
- The **depth of a tree** is the depth of its **deepest leaf**
 - Thus, a tree with only a **single node** has **depth=0**



- Instead of talking about depths, we may also talk in terms of **heights**:
The **height of a node** (denoted h) is the maximal number of edges from it to a **descendant leaf**.
- The **height of a tree** is the height of its root
- Note: for a tree, depth = height

Time Complexity

- In both insertion and lookup, time complexity is the **length** of the **path** we take from the root to the final node. This depends on two factors:
 - 1) The **shape of the tree**, and particularly its **depth**.
 - In the **worst case**, may have to traverse the **whole depth** of the tree.
 - 2) The **location in the tree** of the searched node / place of insertion
 - Even when the depth is large, this location may be close to the root



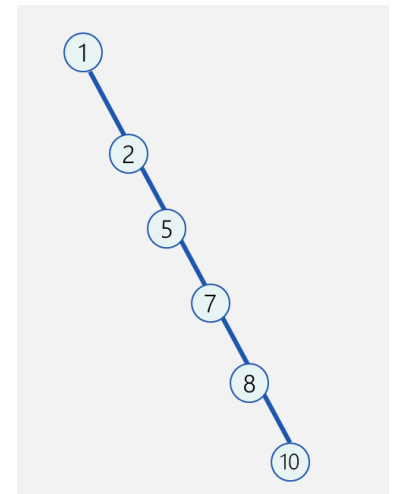
Three tree shapes containing the same set of keys.

Lookup and Insertion: Time Complexity

- Both lookup and insertion follow a **path from the root to some node** (the searched node or the place of insertion). At **each node** they “spend” **$O(1)$ time**. So, The time complexity of both is the length of that path.
- The **best case** occurs when that path ends “near” the root, and takes **$O(1)$ time**, regardless of the tree shape and size.
- The **worst case** occurs when we have to traverse a path from the root to the **farthest (=deepest) leaf** in the tree. In other words, we need to **descent the full depth** of the tree.
 - In a totally **unbalanced** tree, this yields **$O(n)$ time**
 - However, if the tree is **balanced**, this takes **$O(\log n)$ time**.

Totally **Unbalanced** Binary Tree

- Each node has only **one non-empty subtree**.
- The depth d of such a tree with n nodes is
$$d = n - 1 = O(n).$$
- What **insertion order** yields this tree?

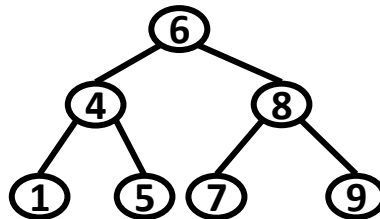


Complete (totally balanced) Binary Tree

- When all the leaves are in the **same level**, and all **internal nodes** have degree exactly 2, we have

$$n = \sum_{i=0}^d 2^i = 2^{d+1} - 1$$

- Such trees are called **complete**.
- Their depth satisfies $d = \lceil \log_2(n+1) \rceil - 1 = O(\log n)$.
- What **insertion orders** yield this tree?



Comment on **Balanced** Trees

- A tree is called **balanced** if its depth d satisfies $d = O(\log n)$ (need not be complete)
- In cases where we have control on the **order of insertion** of nodes to the tree, we can build the tree such that it becomes balanced.
- Furthermore, there are several types of **self-balancing trees**: trees that rebalance themselves to satisfy $d = O(\log n)$ at any time point, regardless of the order insertions/deletions
 - you will meet some in the **Data Structure** course
- In fact, even without such self-balancing mechanism, **random** order of insertion is **likely** to produce a balanced tree (albeit not a complete binary tree).

Self-balancing Trees (for reference only)

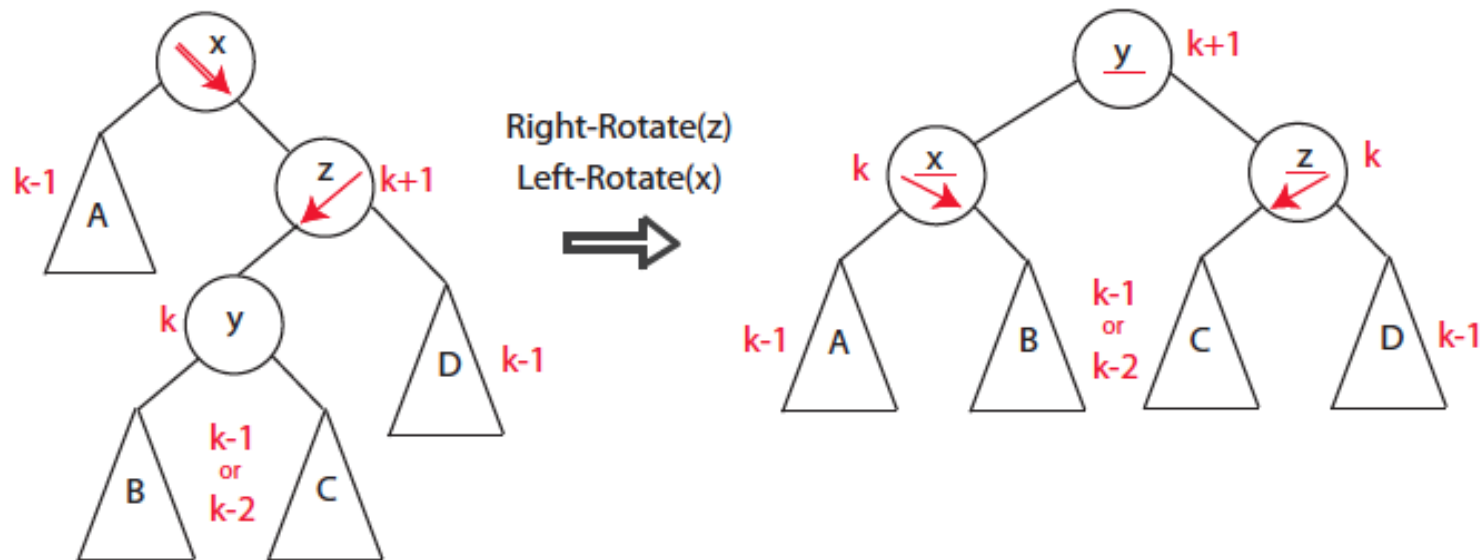


Figure from MIT algorithms course, 2008. Shows item insertion in an AVL tree.

Comic Relief*

*A **binary tree** with 16 leaves.
Courtesy of Dr. Shlomit Pinter,
photo taken in Kenya, 2005*



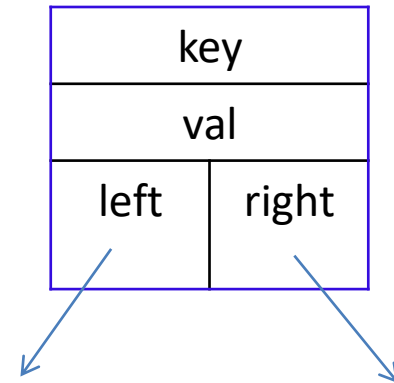
* אנו מזמינים אתכם לשלוח לנו הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Binary Search Tree: Python Code

- A tree node will be represented by `class Tree_node`.
- We allow for each record to hold some **value**, in addition to the **key** by which the tree is ordered (for example, keys are IDs, and values are names/addresses etc.)

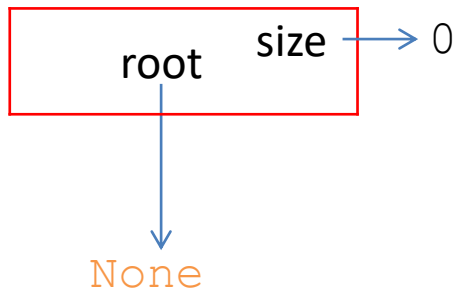
```
class Tree_node:
    def __init__(self, key, val):
        self.key    = key
        self.val     = val
        self.left   = None
        self.right  = None

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

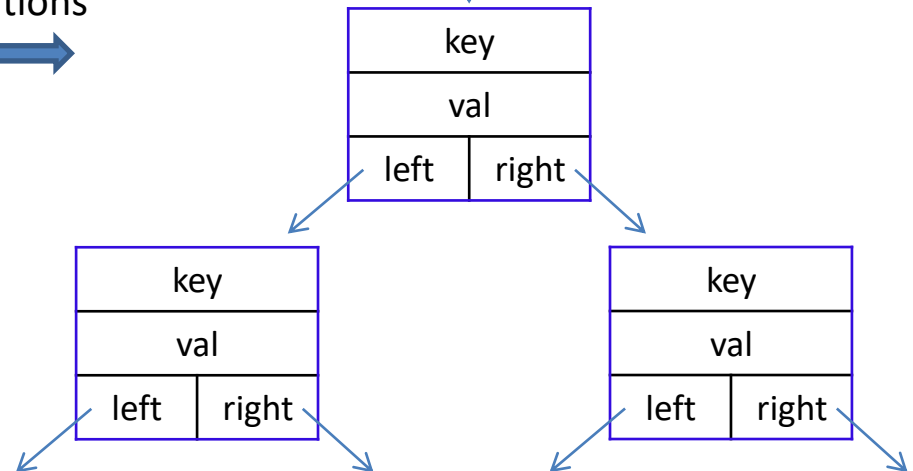
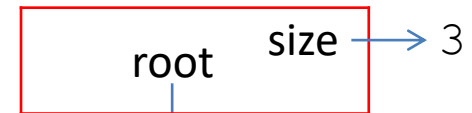


Binary Search Tree Class

```
class Binary_search_tree:  
    def __init__(self):  
        self.root = None  
        self.size = 0
```



after some insertions



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`.

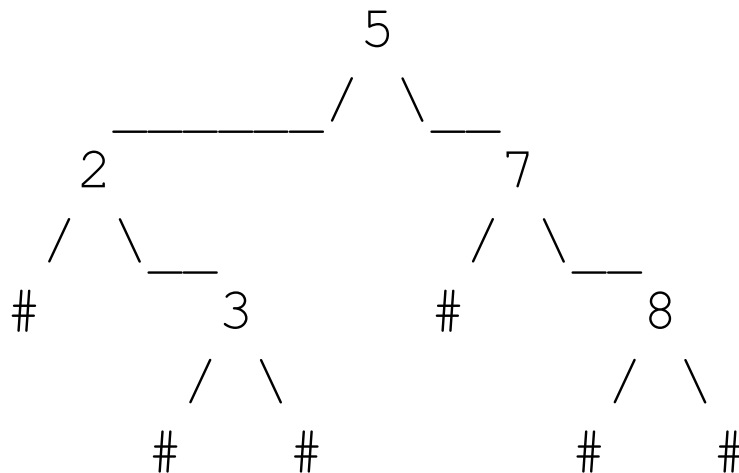
```
from printree import *

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** (search)

```
def lookup(self, key):  
    ''' return val of node with key if exists, else None '''  
    node = self.root  
    while node != None:  
        if key == node.key:  
            return node.val # found!  
        elif key < node.key:  
            node = node.left  
        else:  
            node = node.right  
    return None
```


Binary Search Tree: insert

- We first look for the appropriate location for insertion, and then “hang” the new node as a left/right child.
- To that end, while descending we need to keep a pointer to the last node encountered.
- Note that the new node is always added as a leaf.
- If the user inserts an element whose key already exists in the tree, we assume that it should replace the one in the tree, that is, the value of the node should be updated and no new node is added to the tree.

Binary Search Tree: insert

```
def insert(self, key, val):
    ''' insert node with key,val into tree.
        if key already there, just update its value '''

    parent = None # this will be the parent of the new node
    node = self.root

    while node != None: # keep descending the tree
        if key == node.key:
            node.val = val      # update the val for this key
            return

        parent = node
        if key < node.key:
            node = node.left
        else:
            node = node.right

    if parent == None: # was empty tree, need to update root
        self.root = Tree_node(key, val)
    elif key < parent.key:
        parent.left = Tree_node(key, val) # "hang" new node as left child

    else:
        parent.right = Tree_node(key, val) # "hang" ... right child

    self.size += 1
    return None
```

Operation: `minimum`

- To compute the element with the **minimal key** in a binary search tree, we need to go **all the way to the left**:

```
def minimum(self):  
    ''' return value of node with minimal key '''  
    if self.root == None:  
        return None # empty tree has no minimum  
    node = self.root  
    while node.left != None:  
        node = node.left  
    return node.val
```

- Complexity (worst and best cases)?

Time Complexity of `minimum`

- 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 $n - 1$, and the time complexity is $O(n)$.
- The **worst case** in a `balanced tree` is $O(\log n)$.

Operation: `depth`

- To compute the depth, we use a **recursive** function, using the equation: $d(v) = 1 + \max\{d(v.\text{left}), d(v.\text{right})\}$
- Note that we need two recursive calls from each node.
- 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)
```

Time Complexity of `depth`

- Time complexity is linear in the size of the tree, $O(n)$, regardless of the tree shape (and depth).
- This follows from the observation that **every node** is visited **once**, with $O(1)$ time spent on each one.
- How does the **recursion tree** look like?

Binary Search Tree:

Complexity of our Implementation

	best case	worst case for any tree	worst case for balanced trees
Insert / lookup / minimum	$O(1)$	$O(n)$	$O(\log n)$
depth	$O(n)$	$O(n)$	$O(n)$

- Note that insert / lookup / minimum traverse a single path from the root, while depth traverse the whole tree.

Binary Search Tree:

Concluding Remarks

- We could implement lookup, insert and minimum with **recursion**. This, however, would **not improve** time complexity, and in fact would probably increase actual running time, and would also require **more memory** (why?).
- A function to **delete** a node is a little harder to write, and is omitted here.
- We can ask what the **average** time complexity of lookup and insert is. The average can be taken over which node we search (in a given tree), or over which tree shape we have (with a given element to search), or both.
- We observed that the shape of the tree depends on the sequence of inserts (and deletions) that generated the tree. If we are able to keep the tree **balanced** at all times, we will have an efficient way to store and search data in $O(\log n)$ time.
- You will encounter most of these issues (and more) in the **Data Structures** course.