CSC2710 Analysis of Algorithms — Fall 2022
Unit 7 - Transform and Conquer - Presorting and Instance Simplification

Book: §9.1 – §9.2; §13.1 – §13.4

7.1 Transform and Conquer Overview

- We are going to start looking at methods for algorithms based on the idea of transformations.
- Transform-and-Conquer algorithms consist of two stages
  1. **Transform**: problem instance is modified to be more amenable to a particular solution.
  2. **Conquer**: the transformed problem is solved yielding a solution to the original problem.
- There are three variants of Transform-and-Conquer
  1. **Instance Simplification**: Transform to a simpler or more convenient instance of the same problem.
  2. **Representation Change**: Transform to a different representation of the same instance.
  3. **Problem Reduction**: Transformation to an instance of a different problem for which an algorithm is already available.

7.2 Presorting

- Some algorithmic problems are *a lot* simpler if the input to the problem is sorted.
  - This makes sorting our transformation (specifically instance simplification).
- Naturally, algorithms that utilize presorting have to pay the costs of the sorting algorithm that is being used.
- Item we have seen two optimally efficient sorting algorithms.
  1. Mergesort - $\Theta(n \lg n)$
  2. Quicksort - $\Theta(n \lg n)$ on average, $\Theta(n^2)$ in the worst case.
- There are several examples where presorting an array will be useful in solving the problem.

7.2.1 Example: Checking element uniqueness in an array

- Recall the element uniqueness problem is:

  Definition 1 (Element Uniqueness Problem)**.**
  Input: *A set of n elements $A = \{a_1, a_2, \ldots, a_n\}$.*
  Output: *True if every element in A is unique. An element a_i is unique if $a_i \neq a_j$ for all $j \in \{1, 2, \ldots, n\} \setminus \{i\}$.*

- Our brute force solution to this algorithm required $\Theta(n^2)$ time.
- What benefits can we gain if we assume the sequence is sorted increasing order?
  - Observation: If the array is sorted non-unique elements will occur next to each other in the list.

- The algorithm we use would be as follows

```
# Input: Array A of integers in sorted increasing order
# Output: True is returned if the array consists only of unique
#     elements, False otherwise
def IsUnique( A ):
    for i in range (len(A) - 1):              # Line 1
        if( A[i] == A[i+1] ):                 # Line 2
            return False                      # Line 3
    return True                               # Line 4
```

- What's the running time of IsUnique?
  - If we use an optimal sort we spend $\Theta(n \lg n)$ time presorting.
  - IsUnique looks at $n-1$ elements in the array
  - Our total cost is therefore,
$$cn \lg n + n - 1 \in \Theta(n \lg n) .$$
- Thus, we have shown that presorting helps in solving the element uniqueness problem.

7.2.2 Example: Computing the Mode

- We now consider how presorting will help us determine the mode of a sequence of numbers.
- The *mode* of a sequence is the number that occurs with the greatest frequency in the sequence.
  - Example: the sequence $\langle 3, 5, 7, 2, 3, 1, 3, 2, 3, 7 \rangle$ has mode 3 as three occurs most frequently in the sequence (i.e., there are four occurrences).
- How would we naïvely solve the problem using brute force?
  1. Scan the entire list counting the frequency of each distinct element.
  2. display the element with the largest frequency.
- To use the brute force solution we must use an auxiliary array of size n that holds a tuple of the key and count.
- When we check the i-th element of the original list we must traverse the values already encountered in the auxiliary array to determine if we need to
  - insert a new element with count 1
  - increment the count for a different element.
- What is the worst case time for this brute force approach?
  - In the worst case, each element requires us to check $i-1$ elements in the auxiliary array.

  - This gives us the running time

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} (i-1) \\ &= \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1 \\ &= \sum_{i=0}^{n-1} i - n \\ &= \frac{(n-1)\,n}{2} - n \\ &\in \Theta\left(n^2\right) \end{aligned}$$

- What happens if we presort?
  - Observation: The similar elements are grouped together in “runs”.
  - Our goal becomes to find the element in the sorted array with the longest run.
- The formal algorithm is given below:

```python
# Input: A is a sorted array of integers
# Output: The mode of A
def FindMode( A ):
    i = 0                                                           # Line 1
    mfreq = 1       # The current mode's frequency                    Line 2
    mode = A[0]     # The current mode                                Line 3
    while( i < len(A) ):                                            # Line 4
        rlen = 1    # The length of the current run                   Line 5
        rval = A[i] # The value of the current run                    Line 6
        while( i + rlen < len(A) and A[i + rlen] == rval):          # Line 7
            rlen = rlen + 1  # Increase the count of the run          Line 8
            # If it's a longer run, update the current mode
            if( rlen > mfreq ):                                     # Line 9
                mfreq = rlen                                        # Line 10
                mode = rval                                         # Line 11
        i = i + rlen                                                # Line 12
    return mode                                                     # Line 13
```

- **Practice**: What is the optimal running time for the presorting?
- What is the running time of FindMode?
  - FindMode runs through the entire array exactly once. This results in $\Theta(n)$ time.
- The complete running time is $T(n) = \Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$.
  - Does everyone see why?

7.2.3 Example: Search Problem

- **Practice**: What is the search problem?
- **Practice**: What is the running time of sequential search?
- Do you think presorting an array and then searching with say, binary search will give us a faster algorithm?
  - How many say Yes? No's?
- It turns out that we won't be able to do better here's why.
  - What is the fastest we can sort an array?
    - In time $\Theta(n \lg n)$[1]
  - What is the running time of binary search?
    - In the worst case, $\Theta(\lg n)$.
  - What's the total running time?
    - $T(n) = \Theta(n \lg n) + \Theta(\lg n) = \Theta(n \lg n)$
- We can conclude that presorting doesn't make sense for the search problem.
- Presorting does make sense for many more problems then the ones we've seen. In particular:
  - Computational geometry problems.
  - Some important problems in graph theory.
  - It is also applicable with other techniques we will see later.

7.3 Challenge Problems

1. Consider the problem of finding the distance between the two closest numbers in an array of n numbers. (The distance between two numbers x and y is computed as $|x - y|$).
   - Determine a *brute force* algorithm to find the two closest elements without altering the data in any way. What is the running time of your algorithm?
   - Determine a *transform-and-conquer* algorithm to find the two closest elements. What is the running time of your algorithm? Does presorting save time?
   - Now determine a *transform-and-conquer* algorithm to find the two farthest elements. What is the running time of your algorithm? Does presorting save time?
2. Given two strings, s_1 and s_2 of length n, create a $\Theta(n \lg n)$ algorithm to determine if s_1 and s_2 are anagrams (contain the same characters).
   - What is the running time of your algorithm?
3. Recall the *convex hull* and *closest pair* problems for a group of 2-dimensional points in space.
   - Divise a sub-quadratic (less than $O(n^2)$) algorithm to find the convex hull of the points. Does pre-sorting help? How can you sort the points?
   - Is there a pre-sorting algorithm that helps find the closest pair of points? Does it require extra space? If so, how much?

[1]This is an *extremely* important bound if you haven't noticed yet.

7.4 Instance Simplification - Balanced Binary Search Trees

- An example of a representation change (transformation) from a set to a dictionary data structure. However, if you are already in a bianry search tree, balancing it can be considered an instance simplification.
  - A *dictionary* data structure is a data structure that store key-value pairs (k, v) and provides three operations over this collection[2].
    1. Insert: add a new value with key k to the data structure
    2. Delete: a value with key k from the data structure
    3. Search (sometimes called Query): Determine the value v associated with a key k in the data structure.
  - This is a little different from the dictionary data structure definition you are probably used to.
  - We will only talk about keys here but you can safely assume each entry in the dictionary is an object.
  - Examples: linked lists, BSTs, arrays, hash tables, etc.
- Why is this transformation important?
  - We gain significant time efficencies in insert, delete, and search operations.
  - The BST is a good dictionary data structure in the average case.
  - On average,
    - Insert takes time $\Theta(\lg n)$
    - Search takes time $\Theta(\lg n)$
    - Delete takes time $\Theta(\lg n)$
- **Practice**: In the worst case how long will it take to insert an element into a BST?
- **Practice**: What about the worst case for search?
- Our goal of transforming a set to a BST is to capitalize on average case performance. Ideally, we want the worst case to be the average case.
- The average case performance occurs when the the tree is either balanced or almost balanced.
  - A tree is called balanced if for every internal node the height of its left and right subtree are approximately equal.
  - Sometimes this is called height balanced. If you see balanced in the literature it *always* means height balanced.
  - You can also think about the balance property as the length of the longest root-to-leaf path in the tree is $\lg n$ where n is the number of nodes in the tree.
    - Recall the length of the longest root-to-leaf path in a tree is called it's height.
    - We also have the height of the tree is $\lg n$
- There are two solutions that we will look at today. They are both transform and conquer designs.
  - Instance simplification: Transform an unbalanced tree into a balanced one. These trees are sometimes called *self-balancing*. They are normally balanced on both insertions and deletions.
    - Examples: AVL trees, Red-Black trees[3], splay trees, etc.

[2]Sometimes the values are called satellite data

[3]Coincidentally the balanced tree used in the gcc's C++ STL implementation of `std::map`.

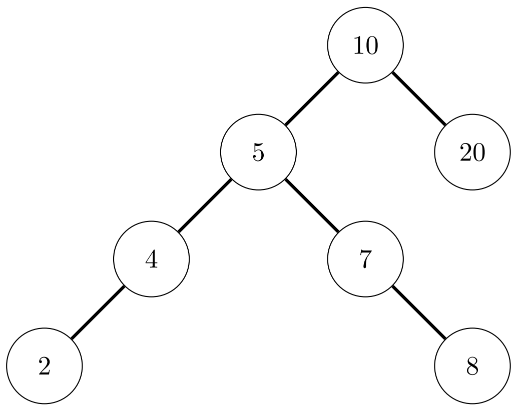
    - All balancing occurs through use of special operations called rotations.
    - We will look at AVL trees as they are the simplest and kickstarted the balanced BST research.
  - A representation change: Allow more than one element in the node of a search tree.
    - Examples: B-Trees (used in databases), etc.
    - All examples algorithms are perfectly balanced trees that differ in the number of elements allowed in a node.

7.4.1 AVL Trees

- First balanced BST structure due to Adelson-Velsky and Landis in 1962.
- In order to understand the definition of AVL trees we must be familiar with the term *balance factor*
  - The balance factor of a node is defined as the difference between the heights of the nodes left and right subtrees.
  - Example: If node v has left subtree $T_{v,l}$ and right subtree $T_{v,r}$ the balance factor is defined as:

$$B(v) = h(T_{v,l}) - h(T_{v,r})$$

    where h that takes a (sub)tree as input and returns the height of the tree.
- **Practice**: Let's compute the balance factor for each node in the following tree. You should assume that empty (sub)trees have a balance factor of -1.



- We define an AVL tree as follows:

  Definition 2 (AVL Tree). *A binary search tree in which the balance factor of every node is either 0, 1, or -1. The factor of -1 will be used to represent the height of an empty (sub)tree.*

- **Practice**: Is the above tree an AVL tree? Why or why not.
- We can prove that if we maintain the AVL property a tree is *always* balanced.
  - We start with fact collection (i.e. "How does that story go again?")
    - What does the tallest n node tree we can have that satisfies the AVL property look like?
      - Answer: A tree where every node in the tree has one branch one node longer than its other branch. WLOG we can say that every node in the tree has its right branch one node longer than its left branch.
      - Do you see why this still satisfies the AVL property?

    * Our goal is to show that the height of the tree is $\Theta(\lg n)$, otherwise the tree is not balanced.
    * It is hard to describe the height of the tree we discussed above but, relatively easy to describe the number of nodes in the tree of height h.
    * What recurrence relation describes the number of nodes, $N(h)$ in the tree of height h?
        · We have to count,
        1. the parent,
        2. the number of nodes in the right subtree, and
        3. the number nodes in the left subtree.
        · In terms of our worst case balancing we have 1 node as the parent (root), $N(h-1)$ nodes in the right subtree and $N(h-2)$ in the left subtree.
        · This gives us the recurrence $N(h) = 1 + N(h-1) + N(h-2)$.
        · What recurrence is this recurrence similar too?
    * It stands to reason that if we solve this recurrence and it comes out to $O(\lg n)$ we have proved that a tree that satisfies the AVL property is always balanced.

- Let's use our facts to prove that if a tree satisfies the AVL property it is always balanced.

  Proof. It is sufficient to argue that tallest tree on n nodes, that satisfies the AVL property is balanced as all shorter trees will naturally be balanced with lower height. The tallest tree on n that satisfies the AVL property, has at every node, one subtree one node taller than its other subtree. WLOG, we can assume that at every node, the right subtree is one node taller than the left subtree. We can define the number of nodes in this tree in terms of its height using the recurrence:

$$N(h) = 1 + N(h-2) + N(h-1)$$

  The base case $N(1) = 1$. This recurrence is *very* similar to the recurrence that defines the Fibonacci sequence. We have $N(h) > F_h$ where F_h denotes the h Fibonacci number. From appendix B of our text, we know that $F_h = \frac{1}{\sqrt{5}}\left(\varphi^h - \hat{\varphi}^h\right) \approx \frac{1}{\sqrt{5}}\varphi^h$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is a constant called the Golden Ratio. This means we have

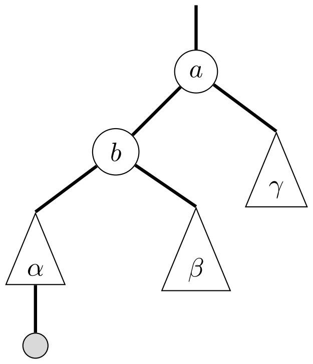
$$\begin{aligned}
& N(h) > \frac{1}{\sqrt{5}}\varphi^h \\
\Longrightarrow\ & \sqrt{5}N(h) > \varphi^h \\
\Longrightarrow\ & \log_\varphi\left(\sqrt{5}N(h)\right) > h \\
\Longrightarrow\ & \log_\varphi\sqrt{5} + \log_\varphi N(h) > h \\
\Longrightarrow\ & \frac{1}{\lg\varphi}\left(\lg\sqrt{5} + \lg N(h)\right) > h \\
\Longrightarrow\ & \frac{1}{\lg\varphi}\left(1.6019 + \lg N(h)\right) > h \\
\Longrightarrow\ & 2.3075 + 1.4405\lg N(h) > h \\
\Longrightarrow\ & h \in O(\lg n) && \text{Where } n = N(h).
\end{aligned}$$

  The proof of the lower bound is left as an exercise to the reader. □

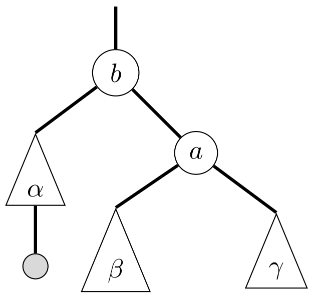
- If an insertion to an AVL tree causes a node to have balance factor greater than one, we must perform a rotation.
    - A *rotation* is a local transformation rooted a node whose balance factor has become 2 or -2.
        * Ties are broken by the unbalanced node that is *closest* to the newly inserted leaf.

  - In effect a rotation always decreases the height of one subtree by one and increases the height of the other subtree by one.

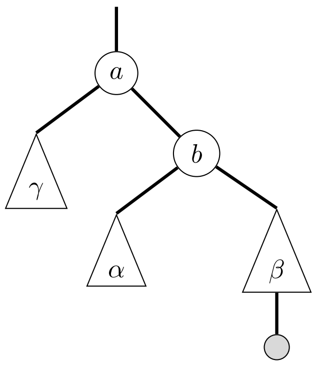
- The book outlines four types of rotations for AVL trees.
- I refer to two as basic rotations and two as compound rotations.
- Basic Rotations
  - **Right Rotation**: The node a's left child b is rotated up into the a's location causing a to be the right child of b .
    * Example Rotating node a, in the below (sub)tree, right



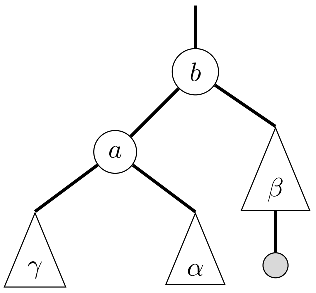
we obtain



    * Observe what happened to the subtrees α, β, γ. Does their placement make sense?
    * A right rotation only occurs if a leaf is inserted into the left subtree of a left child whose root had balance factor 1 *before* the insertion.
  - **Left Rotation**: The node a's right child b is rotated up into the a's location causing a to be the left child of b .
    * Example Rotating node a, in the below (sub)tree, left

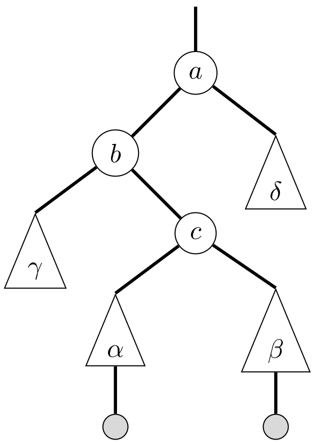


we obtain

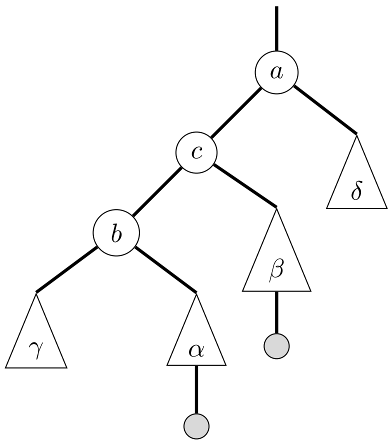


- Observe what happened to the subtrees α, β, γ. Does their placement make sense?
- A left rotation only occurs if a leaf is inserted into the right subtree of a right child whose root had balance factor 1 *before* the insertion.

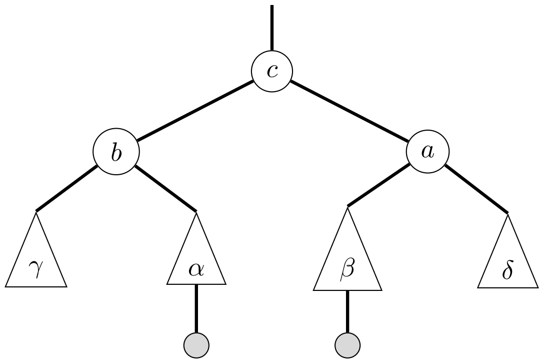
- Compound Rotations
  - If you understand your left and right rotations these compound operations are nothing more than applying that knowledge of rotations twice in succession.
  - Left-Right Rotation: This involves three nodes a, b, and c where c is the right child of b and b is the left child of a. The idea is to rotate b left and then rotate a right.
    - Note: the new leaf can be in either c's left or right subtree.
    - Example: Rotating node b, in the below (sub)tree, left



Which gives us:

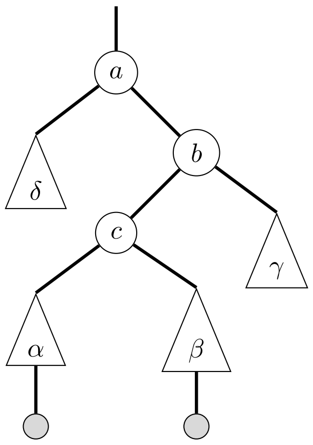


Notice we are still unbalanced but, this can be fixed by a right rotation of a. This give us:

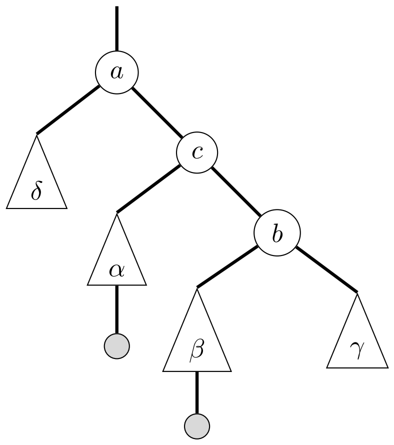


  * A left-right rotation is only performed if a new key is inserted into the right subtree of the left child of a (sub)tree whose root had the balance of 1 *before* insertion.

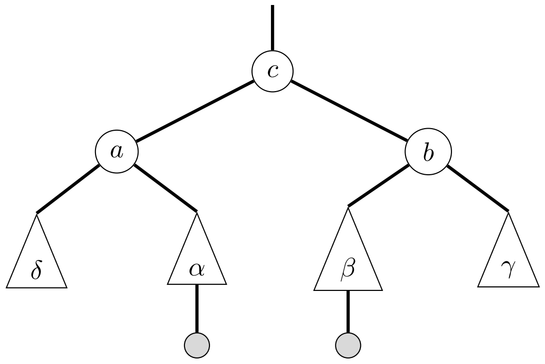
- Right-Left Rotation: This involves three nodes a, b, and c where c is the left child of b and b is the right child of a. The idea is to rotate b right and then rotate a left.
  * Example: Rotating node b, in the below (sub)tree, left



Which gives us:

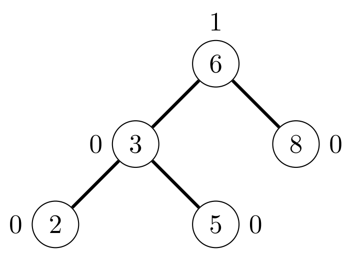


Notice we are still unbalanced but, this can be fixed by a right rotation of a. This give us:

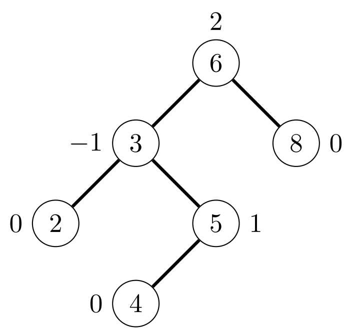


∗ A right-left rotation is only performed if a new key is inserted into the left subtree of the right child of a (sub)tree whose root had the balance of 1 *before* insertion.

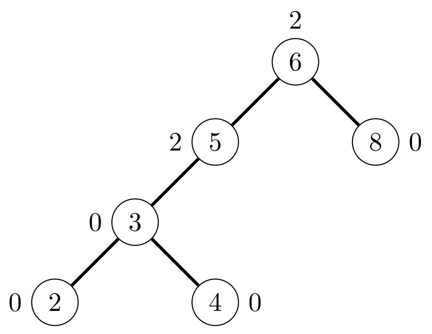
- The general operation of insertion is:
  1. Insert the element into the already balanced tree, this takes $\Theta(\lg n)$ time.
  2. Set that new elements balance factor to zero.
     - **Practice**: Why does a new element in the tree have a balance factor zero?
  3. In a bottom up traversal from the new leaf, update the balance factor for all nodes along the root-to-leaf path correcting the balance as you go using rotations. This takes $\Theta(\lg n)$ time.
     - Recall we apply the rotations based on the closest node in the the root to leaf path to the new leaf that has a "bad" balance factor.
     - Each rotation takes $\Theta(1)$ time.
- From the above we can see that the AVL insert algorithm is $\Theta(\lg n)$
- Let's look at an example of inserting the number 4 into the below tree. The number next to the nodes are the balance factors.



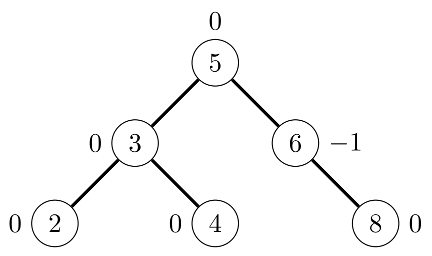
1. Insert the number 4 into the binary tree and update the balance factors.



2. Notice that we have a balance value of 2 in the root of the tree. We know our newly inserted node is a right child of a left child of the root and therefore requires a left-right rotation to restore balance.
   (a) We start with the left rotation of 4's grand parent and get:



   (b) We continue by performing a right rotation at 6. This gives us:



Our tree is now balanced and ready for another insertion.

7.4.2 Red-Black Trees

Check out more information here: https://www.programiz.com/dsa/red-black-tree

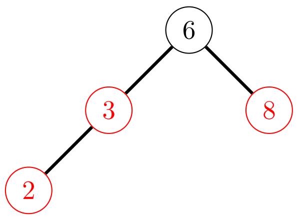
- Red-Black (RB) trees are by far the most common balanced binary search tree.
  - They appear in the underlying implementation of data structures in the C++ STL as well as in Java.
- The goal of RB trees is to reduce the number of rotations that appear in AVL trees.
  - In a sense, the balance factor in AVL trees is kept too tight.
- In a RB tree with n internal nodes, the height is at most $2\lg(n+1)$.
  - See proof of Lemma 13.1 in the book.
- In a RB tree nodes are given a color attribute. The are either *red* or *black*.
- There are parent-child relationships, based on the color attribute, that must be obeyed. They are:
  1. The root node of the tree must be black.
  2. If a node is red, both of its children are black.
  3. For each node, all simple paths from the node to the leaves contain the same number of black nodes.
- We use the following process to insert a new node in the RB tree T.
  1. Insert the node z in the tree as you would in any BST.
  2. Color the newly inserted node *red*.
  3. At this point the RB-Tree could be violating coloring requirements, so we must fix up the coloring. This is only done while z's parent is red. There are three cases.
     (a) Case 1: z's uncle is *red* and z's parent is a left or right child.
       - Color z's uncle black.
       - Color z's parent black.
       - Color z's grand parent red.
       - Set z to z's grandparent and repeat step 3 if z's parent is red.

     (b) Case 2: z's uncle is black or z has no uncle, and z is a right child.
       - Set z to the parent of z.
       - If z's parent is a left child, right rotate z; otherwise left rotate z.
       - Proceed to case 3.

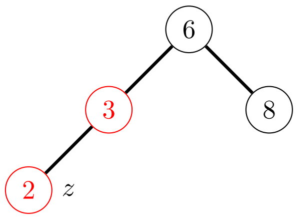
     (c) Case 3: z's uncle is black or z has no uncle, and z is a left child.
       - Color z's parent black
       - Color z's grandparent red
       - If z's parent is a left child, right rotate z's grandparent; otherwise left rotate z's grandparent.
       - Repeat step 3 if z's parent is red.
  4. Color the root node black.

- Let's look at and example of case 1.

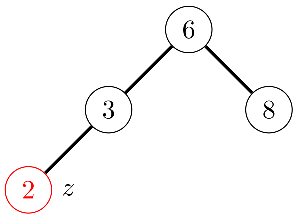
1. Insert node $z = 2$ with color red.



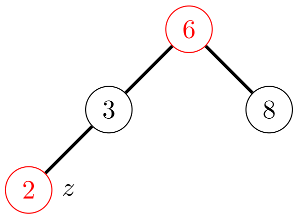
2. Color uncle black.



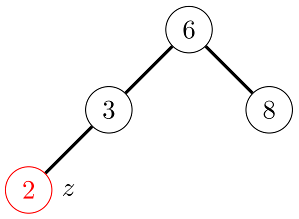
3. Color parent black.



4. Color grandparent red.

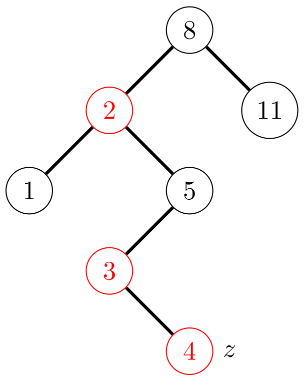


5. We've finished fixing up the tree, color the root black.

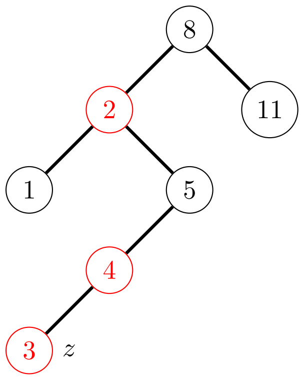


- Let's look at case 2 and 3 together.

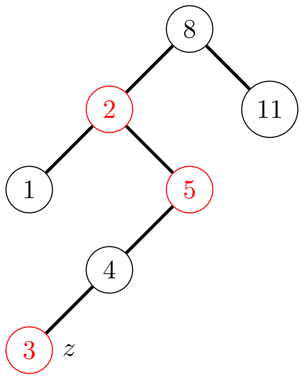
  1. Insert node $z = 4$.



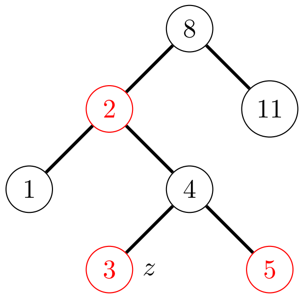
  2. Case 2 says to set z to the parent (in this case $z = 3$) and left rotate z. This give us:



  3. We're now in case 3 (and z's parent is red, so we continue), so color z's parent black and its grandparent red.



4. Right rotate z's grandparent to complete the case.



7.5 Challenge Problems

- Using the `rbtree.py` file on Classroom:
  - Insert: 4, 2, 6, 7, 8, 9, 10, 12, 14, 2, 3, 6, 23, 3, 9, 67, 4
  - Print the tree - is it balanced (according to the definition of 'balanced' discussed for AVL trees)?
  - Does it adhere to the three red-black properties?
- Create an AVL tree (on paper, not Python)
  - Insert the following: 7, 3, 11, 14, 17, 22 (updating and noting the balance factors for each insertion)