1 Balanced Binary Search Tree

In the review of binary search trees, we notices that it is possible that a binary search tree becomes unbalanced and degenerates into a linked list. For instance, if we insert 1, 2, 3, ..., n into an empty tree. Now the time to insert n elements becomes quadratic in n and we have some idea about the quadratic run time in our earlier project.

One idea to overcome the unbalance is to make sure that elements are randomly inserted into a binary search tree and hope that the randomness could make the tree more balanced. Since we may not be able to control the way elements are inserted, another idea is to dynamically adjust the binary search tree so that certain balanced condition is maintained. We want to make sure the time spent in adjusting the tree is in proportion to the height of the tree.

We have used term balanced without defining it and mainly appeal to our intuition. Formally, a tree is balanced if \( h = \Theta(\log n) \), where \( h \) is the height of the tree and \( n \) is the number of nodes in the tree. Since for a binary tree that is completely filled at every level, \( h > \log n \), we only need to show \( h = O(\log n) \).

We will consider two well known balanced binary search trees: AVL trees and Red-Black trees.

1.1 AVL Trees

Let us state the definition of an AVL tree first.

**AVL Tree:** An AVL tree is a binary search tree with the additional properties:

- **Basis:** An empty binary search tree is an AVL tree.
- **Recursion (induction):** For a nonempty binary search tree, it is an AVL tree provided \(| h(S_L) - h(S_R) | \leq 1\), where \( h(S_L) \) and \( h(S_R) \) are heights of left subtree, \( S_L \), and right subtree, \( S_R \), of the root respectively; and both subtrees are AVL trees.

It is instructive to consider a few binary search trees to see if they are AVL trees or not. We have done that in our lectures.

The next question could be if this definition really makes the tree balanced or not. To answer this question, we will ask ourselves another question:

1.1.1 Why is an AVL tree balance?

*What is the minimum number of nodes an AVL tree must have in order to reach height \( h \)?
Let \( N_i \) denote the minimum number of nodes an AVL tree must have in order to reach height \( i \).

Try to verify that \( N_0 = 0 \), \( N_1 = 1 \), \( N_2 = 2 \), \( N_3 = 4 \), \( N_4 = 7 \), and \( N_5 = 12 \).

If we follow the pattern that we observe in verifying \( N_0 \) to \( N_5 \), we have the following integer sequence:

\[
0, 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, \ldots
\]

What is the next value? As it turns out, \( N_i \) can be defined recursively by

\[
N_i = 1 + N_{i-1} + N_{i-2}
\]

for \( i \geq 2 \) with initial values \( N_0 = 0 \) and \( N_1 = 1 \).

We may wonder what is the purpose of study \( N_i \). The reason is to establish that an AVL tree is balanced. Suppose we have an AVL tree with \( n \) nodes. Then there exists \( i \) such that \( n \leq N_i \). Then the height of this tree is no more than \( i \). For instance, suppose an AVL tree has 138 nodes. Since 138 < 143, the height of this tree is less than 10. The reason follows from the definition that \( N_i \) is the minimum number of nodes an AVL tree must have to reach height of \( i \). If an AVL tree has nodes that is no more than \( N_i \), then its height cannot be more that \( i \).

To really answer the balanced question, we need to establish the relationship between \( i \) and \( N_i \). To do that let us consider another sequence below:

\[
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots
\]

Do we recognize the sequence? This is the Fibonacci sequence, which can be defined recursively by

\[
F_i = F_{i-1} + F_{i-2}
\]

for \( i \geq 2 \) with initial values \( F_0 = 0 \) and \( F_1 = 1 \).

By observing the two sequences and their recursive definitions, we establish that, \( i \geq 0 \), \( N_i = F_{i+2} - 1 \). By knowing the the relationship between \( i \) and \( F_i \), we may find the relationship between \( i \) and \( N_i \). One technique to solve \( F_i \) in terms of \( i \) is generating function, which can be used to solve \( N_i \) in terms of \( i \) directly.

From Fibonacci sequence results we have:

\[
F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)
\]

with

\[
\phi = \frac{1 + \sqrt{5}}{2} = 1.61803...
\]

\[
\hat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803...
\]

Notice that \( \hat{\phi} \) is a negative number. Hence, \( \hat{\phi}^n \) could be positive or negative depending on \( n \) being even or odd. Notice also that \( \phi \) is strictly less than 1. Hence, \( |\hat{\phi}^n| < 1 \) and \( \hat{\phi}^n \) gets arbitrarily small as \( n \) gets arbitrarily large. We have:

\[
\frac{1}{\sqrt{5}} (\phi^n - 1) < F_n < \frac{1}{\sqrt{5}} (\phi^n + 1)
\]

\[
(\phi^n - 1) < \sqrt{5} F_n < (\phi^n + 1)
\]
or

\[ \sqrt{5} F_n \approx \phi^n \]

By taking the log with base \( \phi \), we obtain:

\[ \log_\phi \sqrt{5} + \log_\phi F_n = n \]

by replacing \( n \) with \( n + 2 \)

\[ \log_\phi \sqrt{5} + \log_\phi F_{n+2} = n + 2 \]

or

\[ \log_\phi F_{n+2} = n + 2 - \log_\phi \sqrt{5} \]

Since \( N_n = F_{n+2} - 1 \), we have \( N_n = F_{n+2} - 1 \approx F_{n+2} \). Hence:

\[ \log_\phi N_n = \log_\phi F_{n+2} = n + 2 - \log_\phi \sqrt{5} \]

Note that \( \log_\phi \sqrt{5} = 1.672... \). After changing base to 2, we obtain

\[ 1.44 \log_2 N_n \approx \log_\phi F_{n+2} = n + 2 - \log_\phi \sqrt{5} > n \]

Recall that \( n \) is the height and \( N_n \) is the minimum number of nodes to achieve such height. The height is bounded from above by \( 1.44 \log_2 N_n \). Hence, an AVL tree is a balanced tree.

We will skip the search operation for an AVL tree since the same search for basic binary search tree works here. We will consider the insert operation next.

### 1.1.2 Insertion Operation

Let us define the balance factor of each node in a binary tree as the difference between the height of the left subtree and the height of the right subtree. For an AVL tree, from the definition, the balance factor of each node takes three possible values: 1, 0, -1. Value 1 means left subtree height is one more than that of the right subtree, 0 means the same, and -1 means right is one more.

When a node is inserted into an AVL tree, it follows the same logic as inserting into a basic binary search tree. After the insert, the balance factor of its ancestors could be changed. Try to come up with a few AVL trees. Then perform one insert and observe the changes in the balance factor.

Now consider the problem of adjusting each node’s balance factor after an insert into an AVL tree. Here is the logic:

1. The balance factor of the inserted node is 0.

2. In the return from recursive insert, two possibilities exist: the height is increase by 1 or remains the same.
(a) If the height remains the same, from this point on in the return no ancestors’ balance factor changes as a result of insert

(b) If the height increases by 1, then this could be the left subtree or the right subtree. Since the two cases are symmetric, we will consider the left case. The structure of the recursive code tells us which case it is.

i. the node’s current balance factor is 1: new value is 2 and its ancestors should reexamine their balance factor.

ii. the node’s current balance factor is 0: new value is 1 and its ancestors should reexamine their balance factor.

iii. the node’s current balance factor is -1: new value is 0 and its ancestors need not reexamine their balance factor.

Realizing that two cases exist with regard to adjusting the balance factor, we need to communicate the information when returning from a recursive call. In C++, we can use pass by reference flag for that. We add a new field balance_factor to the tree node:

```cpp
struct Node {
    int data;
    Node* left;
    Node* right;
    char balance_factor; // why char here?
};
```

Use the above logic, we modify the recursive insert so that balance factor is adjusted.

```cpp
bool insert(Node *& T, int key, bool& flag)
{
    if (T == NULL){
        T = new Node;
        T -> data = key;
        T -> left = NULL;
        T -> right = NULL;
        T -> balance_factor = 0;
        flag = true;
        return true;
    }
    if (T->data == key){
        flag = false;
        return false;
    }
    else if (key < T->data){
        bool tmp = insert(T->left, key, flag);
        if (tmp && flag) { // left subtree height is 1 higher
            flag = false;
            return false;
        }
        return true;
    }
    else if (key > T->data){
        bool tmp = insert(T->right, key, flag);
        if (tmp && flag) { // right subtree height is 1 higher
            flag = false;
            return false;
        }
        return true;
    }
}
```
T->balance_factor += 1;
    if (T->balance_factor == 0)
        flag = false;
}
return tmp;
else // key > T->data
    // logic is symmetric to return from left;
}

In the lecture, we went over an example of inserting the following into an empty AVL tree: MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEPT.

Please review the lecture note for the details of this example. We will recap the main logic here. We need to first identify the nearest ancestor of which the balance factor becomes \( \pm 2 \). The example code of updating balance factor within the insert function shows how this could be done. If it is \(+2\), the return or insertion is from the left branch. In this case, either a LL or LR rotation needs to be performed. After that the resulting tree becomes AVL tree again. If it is \(-2\), the return is from the right subtree. In this case, either a RR or RL rotation needs to be performed.

Let \( p \) refers to the location of which the value points to the nearest ancestor having new balance factor \( 2 \). The code for LL rotation in C++ is as follows.

Node * save;
save = p->left->right; // save the right child of the new root
p->left->right = p; // old root becomes the right child of new root
p=p->left; // old root’s left child becomes new root
p->right->left=save; // the right child of the new root
    // becomes the left child of the old root

The code is not unique. Many other ways of coding may achieve the same logic. Recall different statements of inserting into a doubly linked list in the lecture, which is true in this case. For example, here is another code:

Node * save;
save = p->left;
p->left = p->left->right;
p->right =p;
p = save;

Notice that the key ideas of a rotation are: a rotation involves three nodes; the node whose key value is the median becomes the root; and only pointer values in the node fields are adjusted, as shown above.

The code for LR rotation in C++ is as follow.

Node * newR=p;
Node * newL=p->left;
Node * newP=p->left->right;
// the above keeps points to the three nodes involved in a rotation
newR->left = newP->right;
newL->right = newP->left;
newP->left = newL;
newP->right = newR;
p = newP;

Note the above involves 8 assignments and three new variables. It is possible to reduce the number of assignments and the new variables. It is instructive to figure that solutions out.

With adjustment to what p refers, the code in Java is similar and is left as problems. The cases for RR and RL rotations are symmetric and left as problems as well. In our project we will have a chance to work on all rotations and how to code them elegantly.

1.1.3 Deletion Operation

Similar to the deletion to a basic binary search tree, the deletion to an AVL tree is broken down into three cases. Notice that for the case that the deleted node has a single child, that branch contains a single node if the tree is an AVL tree. It is instructive to provide the argument for it. So from an algorithmic logic viewpoint, the only interesting case is deleting a leaf node. Does it make sense us? It is instructive to think really hard about how other cases reduce to this case.

When a leaf node is removed, the balance factor of its parent will change. This is similar to insert in reverse. We should be able to come up an algorithm to adjust the balance factors of those affected nodes as the result of the deletion. The algorithm is similar to what is presented earlier for adjusting balance factors due to insertion.

Suppose the deleted leaf node is the left child or in general the height of the left branch is one shorter than before the deletion. The case for the right child is symmetric. Depending on the balance factor of the parent, the following logic applies:

1. The parent node’s balance factor is 1 before the deletion: new balance factor is 0. The subtree remains an AVL tree, but the height is one shorter than before. Continue the process up the tree unless it is the root.

2. The parent node’s balance factor is 0 before the deletion: new balance factor is −1. The subtree remains an AVL tree, and the height remains the same as before. The current tree is an AVL tree and done.

3. The parent node’s balance factor is -1 before the deletion: new balance factor is −2. The subtree is no longer an AVL tree and a rotation is needed to make the subtree an AVL tree. The type of rotations depends on the balance factor of the right child of the parent:
(a) the balance factor is \(-1\): RR rotation. Adjust balance factors of new root to 0 and old root to 0. The height of the subtree is one shorter than before. Continue the process up the tree unless it is the root.

(b) the balance factor is 0: RR rotation. Adjust balance factors of new root to 1 and old root to \(-1\). The height of the subtree remains the same. The current tree is an AVL tree and done.

(c) the balance factor is 1: RL rotation. Adjust balance factors of all tree nodes involved in the rotation (the values will depend on the old balance factor of the new root and it is instructive to figure it out). The height of the subtree is one shorter than before. Continue the process up the tree unless it is the root.

1.2 Red-Black Trees

Let us state the definition of a Red-Black tree first.

Red-Black Tree: A Red-Black tree is a binary search tree with the additional properties:

1. An empty binary search tree is a Red-Black tree.

2. Each node is either Red or Black, that is each node has a color that is either red or black.

3. Root is black so are the NULL pointer leaves.

4. If a node is Red, then both its children are Black.

5. All paths from any node to all its descendent leaves contain the same number of Black nodes.

It is instructive to consider a few binary search trees with nodes colored red and black to see if they are Red-Black trees or not. We have done that in our lectures. The above definition is from the introduction to algorithms textbook by Corman et al. Here is an equivalent definition:

Red-Black Tree: A Red-Black tree is a binary search tree with the additional properties:

- An empty binary search tree is a Red-Black tree.

- Each node is either Red or Black, that is each node has a color that is either red or black.

- Root is black.

- All paths from any node to all its descendent leaves (NULL pointers) contain the same number of Black nodes.
It is instructive to argue that both definitions are equivalent. Notice that the second definition is bit “simpler”, where it is not required that NULL pointer be a node in the tree and that a red node must have two children explicitly.

The next question could be if this definition really makes the tree balanced or not.

### 1.2.1 Why is a Red-Black tree balance?

Notice that for a binary tree when all paths from any node to all its descendent leaves contain the same number of nodes, the tree is a complete or full tree. For such trees the relation between the height of the tree, \( h \), and the number of nodes in the tree, \( N_h \), is \( N_h = 2^h - 1 \). Suppose \( h \) is the height of a red-black tree. Then there is a path from the root to a leaf node that has length \( h - 1 \). Such a path must have \( \lceil \frac{h}{2} \rceil \) many black nodes. So the number of black nodes is at least \( 2^\frac{h}{2} - 1 \), since all paths from any node to all its descendent leaves contain the same number of black nodes for a red-black tree. It is instructive to prove that statement. What is the minimum number of nodes for red-black trees of height \( h \) as a function of \( h \)? Since a binary tree of height \( h \) can have at most \( 2^h - 1 \) nodes, the number of nodes, \( N_h \), in a red-black tree is between \( 2^\frac{h}{2} - 1 \) and \( 2^h - 1 \). We have:

\[
2^\frac{h}{2} - 1 < N \leq 2^h - 1
\]

Or

\[
2^\frac{h}{2} < N + 1 \leq 2^h
\]

Taking the \( \log_2 \):

\[
\frac{h}{2} < \log_2(N + 1) \leq h
\]

From \( \frac{h}{2} < \log_2(N + 1) \), we get \( h < 2\log_2(N + 1) \). So the height of a red-black tree is no more than \( 2\log_2(N + 1) \). Therefore, red-black trees are balanced.

We will skip the search operation for a red-black tree since the same search for basic binary search tree works here. We will consider the insert operation next.

### 1.2.2 Insertion Operation

When a node is inserted into a red-black tree, the same logic, as inserting into a basic binary search tree, is followed. After the insert, the node is colored red. If the parent of the inserted node is a black node, which is a nice situation, we are done. It is instructive to argue that the resulting tree is indeed a red-black tree. If the the parent is red node, property 4) is violated. To correct the violation, rotation and recoloring will be needed. Keep in mind that property 4) and 5) must hold. Notice that we cannot simply make one of the red nodes black as this will violate property 5).

First note that there are four possible configurations of the two adjacent red nodes and their immediate ancestor: LL, LR, RR, RL. Since RR and RL are symmetric to LL and LR, LL and LR will be considered here and the symmetric cases are left as problems.
Next notice that the uncle of the lower red node could be either red or black. So there are two cases to consider. Note that the grand parent of the lower red node must be black.

The logic of adjusting two LL red nodes:

- If the uncle is black, make a LL rotation with two red nodes and grand parent of the lower red node. After rotation, color the upper red node black and grand parent red, which is possible because the uncle is black. Done.

- If the uncle is red, make both parent and uncle black and grand parent red. This will make sure the number of black nodes from the grand parent to the leaves remain the same. If the grand parent is the root of the tree, make it black and we are done. If the grand parent’s parent is black, it is done as well. If the grand parent’s parent is red, we end up with two adjacent red node cases again, but move up one level in the tree. The process will be repeated for the two new red nodes.

The logic of adjusting two LR red nodes:

- If the uncle is black, make a LR rotation with two red nodes and grand parent of the lower red node. After rotation, color the lower red node (now root for the subtree) black and grand parent red. Done.

- If the uncle is red, make both parent and uncle black and grand parent red. This will make sure the number of black nodes from the grand parent to the leaves remain the same. If the grand parent is the root of the tree, make it black and we are done. If the grand parent’s parent is black, it is done as well. If the grand parent’s parent is red, we end up with two adjacent red node cases again, but move up one level in the tree. The process will be repeated for the two new red nodes.

Note that the second steps of both cases are the same. Note also once the rotation has taken place, similar to AVL tree insert, we are done. Unlike AVL tree insert, sometimes we have to recoloring and move up the tree, until we find a black uncle, where a rotation takes place.

Based on the above logic, in the lectures, we went over an example of inserting the following into an empty red-black tree: MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEPT. In this example, we have RR (black uncle) case, recoloring (red uncle) and root case, LL case, recoloring case, LL case, recoloring followed by LR case, LR case, and RR case. It is instructive to go through this example again from scratch on your own.

1.2.3 Deletion Operation

Similar to the deletion in a basic binary search tree, the deletion in a red-black tree is broken down into three cases. Since the case where the deleted note
having two children is reduced to either deleting a leaf node or deleting a single child node, we will consider the leaf node and single node cases.

If the deleted node is a red node, which is a nice situation, then after removing the node the resulting tree remains a red-black tree and we are done.

If the deleted node is a black node, let us assume the node is on the left branch (the right branch is symmetric and will be left as problems). For a leaf node being deleted, its parent’s left branch has one fewer black nodes than its right branch, the resulting tree violates red-black tree property 5). For a single child node being deleted, if its only child is a red node, recolor this node black and we are done. Otherwise, the situation is similar to the leaf node case, that is the left branch of the parent of the deleted node has one fewer black node than its right branch.

Similar to the development of insertion logic, we will break down various possibilities into different cases and handle each separately. Notice that some cases could be combined together and reduced to one another. Also we will provide one transformation to remedy the violation even though other transformations are available in general.

Let x be the root of the subtree, where its left branch has one fewer black node than its right branch. Let y be the right child of x and z be the left child of x after removing the deleted node. Note that the color of z is always black (Please verify that it is the case and provide the argument why it is so).

For the base case, z is NULL or null. We have the following cases:

- x is black and y is black with left and right NULL or null. Color Y red, move the branch having one fewer black node up to x. If x is the root of the tree done otherwise continue the process (see A(1)).
- x is black and y is red with two black nodes whose left and right are NULL or null. Color y black, y’s left child red, RR rotate with x and y and done (see E(1)(a)).
- x is red and y is black with left and right NULL or null. Color x black, y red, and done (see A(2)).
- x is red and y is black having a red left child (it may have red right child). RL rotate x, y, and the red left child of y; color x black; and done (see c(2))
- x is red and y is black having a red right child. RR rotate x, y, and the red right child of y; and done (see B(2)).

It is instructive and educational to make sure the above logic works and these are all the cases. Now consider the situation where z may not be NULL or null. We will have nine cases when we consider the coloring of y and y’s children (as well as the coloring of x):

A: y is black with two black children.
1. x is black: color y red, move the branch having one fewer black node up to x. If x is the root of the tree done otherwise continue the process.

2. x is red: color x black and y red. Done.

**B: y is black with a black left child and a red right child.**

1. x is black: RR rotate x and y, color the red right child black, and done.
2. x is red: RR rotate x and y, and done.

**C: y is black with a red left child and a black right child.**

1. x is black: RL rotate x, y, and the red left child; color the red left child (new root for the subtree) black; and done.
2. x is red: RL rotate x, y, and the red left child; color x black; and done.

**D: y is black with with two red children.**

1. x is black: could be treated the same as B(1) or C(1).
2. x is red: could be treated the same as C(2).

**E: y is red (with two black children).**

1. x is black:
   (a) y’s left child has two black children: Color y black and its left child red, RR rotate x, y, and done.
   (b) y’s left child has a black left child: RR rotate x, y; switch colors of x and y; perform B(2) on left subtree; and done.
   (c) y’s left child has a red left child: Color y black, RR rotate x, y; (the left subtree almost becomes C(1) case) RL rotate the left subtree (perform the first part of C(1) without recoloring the new root); and done.
2. x is red: This is impossible. Why?

## 2 Problems

After studying AVL trees and Red Black trees, we hope that we are able to solve the following problems.

1. Show that for any binary tree, \( h > \log n \), where \( h \) is the height of the tree and \( n \) is the number of nodes in the tree.

2. Show that when all paths from any node to all its descendent leaves contain the same number of nodes, the binary tree is full or complete.
3. Complete the insert function for the symmetric case.

4. Provide code in Java similar to the insert function, which adjusts the balance factors, in the subsection of Insert Operation of AVL trees.

5. Provide code in Java for LL and LR that is similar to the C++ code in the subsection of Insert Operation of AVL trees.

6. Develop the code (C++ or Java) for RR and RL rotations.

7. Complete the AVL tree delete logic for (3)(c), that is the new balance factor values of the nodes involved.

8. Provide the logic for Red Black tree insert for RR and RL cases similar to we did for LL and LR cases.

9. The basic idea of rotation is to maintain binary search tree property while locally modify the tree structure. Observe that LL and RR rotations involve only two nodes, that is apart from the parent of the subtree only two nodes’ fields are modified; and LR and RL rotations involves three nodes. Convince yourself that this is the case.

10. Let X (grand parent), Y (parent), Z (child) be the three nodes involved in a LR rotation. Show that the LR rotation can be achieved through a RR rotation involving Y and Z, followed by (on the tree resulted from the RR rotation) a LL rotation involving X and Z.

11. Let X (grand parent), Y (parent), Z (child) be the three nodes involved in a RL rotation. Show that the RL rotation can be achieved through a LL rotation involving Y and Z, followed by (on the tree resulted from the LL rotation) a RR rotation involving X and Z.

12. Make sure we know how to write code to implement rotations.

13. Develop an example AVL tree where a deletion of a leaf results 3 rotations all the way to the root by the deletion logic.

14. Explain why B(2) logic cannot be used for D(2) case.

15. Show that the two definitions of Red Black trees are equivalent.

16. What is the minimum number of black nodes for Red Black trees of height, $h = 0, 1, \ldots$, as a function of $h$?

3  Summary

We have introduce the notion or a measure for a binary search tree to be balanced. We have studied two kinds of binary search trees, namely AVL trees and Red Black trees, that are balanced according to the balanced notion. From the
definition of an AVL tree, we have shown AVL trees are balanced. We have done the same for Red Black trees.

The search operation for AVL trees and Red Black trees are identical to the search operation for basic binary search trees. However, the insert and delete operations are more complicated algorithmically even though the same bottom up (recursive) framework is used. The complications are due to maintaining the resulting tree after each insertion or deletion as an AVL tree (or a Red Black tree). The key idea is to use rotations (LL, LR, RR, RL) to make the resulting tree having the desired properties. Another mental tool used in developing the algorithmic idea is case enumeration and case reduction.