1. Balanced Search Trees

- Popular balanced search trees:

  AVL Trees – named after its inventors Adel’son-Vel’skii and Landis. It is a balanced binary search tree and was the first balanced trees to be proposed. Note: only need to handle insertion() and deletion()

  Red-Black Trees – Each node contains a color field (only 1 bit). It is a balanced binary search tree.

  2-3 Trees – each internal node has either two or three children, and all leaves are in the same level. It is not a binary tree. This is not covered here

2. Introduction to AVL Trees

- The subtrees of each node differ by at most 1 in their height.
- Maintain height of $O(\log n) \Rightarrow O(\log n)$ in searching
- Deletion and insertion take $O(\log n)$
- Each node maintain a balance factor ($= \text{height of right subtree} - \text{height of left subtree}$). Example:
Each node in AVL tree has a balance factor of −1, 0 or 1

Only insert() and delete() operations change the AVL tree structures.

Insert() : Node is inserted into AVL Trees in the same manner as an ordinary binary search tree, i.e. nodes are always inserted as leaf nodes.

After insertion, travels back (modify search strategy for this) along the path it took to find the point of insertion, and checks the balance at each node on the path.

If a node is found to be unbalanced (that is, it has a balance factor of either -2 or +2), then a rotation is performed to maintain it as AVL tree.

Note : Can be showed that only one (single or double ) rotation is needed.

delete() : Node is deleted from AVL Trees in the same manner as an ordinary binary search tree.

After deletion, travels back along the path it took to find the point of deletion, and checks the balance at each node on the path.

Note : Can be showed that multiple rotations may be needed.
There are four types of rotations:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Type of rotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>-1</td>
<td>-</td>
<td>Single right rotation</td>
</tr>
<tr>
<td>-2</td>
<td>+1</td>
<td>-</td>
<td>Double (left, then right)</td>
</tr>
<tr>
<td>+2</td>
<td>-</td>
<td>+1</td>
<td>Single left rotation</td>
</tr>
<tr>
<td>+2</td>
<td>-</td>
<td>-1</td>
<td>Double (right, then left)</td>
</tr>
</tbody>
</table>

**Right (at X) Rotation**

**Double Left (at Y) Rotation**
Example: Insert 55
We will need a double rotation to adjust the above tree.

1<sup>st</sup> rotation: left rotation at 30 (i.e. A=30, B=50)
2\textsuperscript{nd} rotation: right rotation at 80 (i.e. B=80, A=50)

More examples are given in class.....

- There are many websites with tutorial on AVL tree implementation. For example:


- Many websites with AVL tree animation. Example: (Need Java)

  http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html
3. AVL Trees: Insertion algorithm

- Each node has a balance factor of -1, 0, 1
- Rebalance tree immediately if it is unbalanced

Recursive Algorithm for insertion (modify from BST insertion)

// return True if there is a change in height
If the root is NULL
    Create a new tree with the item at the root and return True

Else if the item = root->data
    Throw exception // the data is not unique

Else if the item < root->data
    Recursive insert the item in the left subtree
    // main steps to rebalance tree ….
    If the height of the left subtree has increased (i.e. return True)
        Decrement balance // i.e. +1 → 0; 0 →-1; -1 → CRITICAL
    If balance is 0, return False // the height of this tree does not change
    If balance is -1, return True // the height of this tree increases by 1
    If balance CRITICAL: //i.e. -2, node balance is set in function
        Call Rebalance_Left() on this node
        Return False

Else // item > root->data
    Recursive insert the item in the right subtree
    // symmetric to left subtree case
// See table and figure in page 3
// Pre-conditions: values of balance in
// current root node X = -1;
// left subtree root node Y = ?;
// left right subtree root node W = ?
//
// Need to set proper balance values

Rebalance_Left(current_root_node)

If Y = +1 // Left-right case: Need double rotation

   // set proper balance values after rotations
   If W = -1
      X = +1; Y = 0; W = 0
   Else If W = +1
      X = 0; Y = -1; W = 0
   Else // W = 0
      X = 0; Y = 0; W = 0

   Rotate left at node Y // 1st rotation; Only need to modify 2 links

Else // Y = -1, Left-left case: need single rotation

   // set proper balance values after rotation
   Y = 0; X = 0

   // (i) 2nd rotation in Left-right case and
   // (ii) the only rotation in Left-left case
   Rotate right at X // only need to modify 2 links
4. Introduction to Red-Black Trees

- Each internal node has an additional field: color (1 bit)
- Each external node has 2 nil nodes (virtual nodes with no values)
- Now, all original nodes are internal nodes
- A binary search tree is a red-black tree if:

1. Every node is either red or black
2. Every leaf (nil) is black
3. If a node is red, then both its children are black
4. Every simple path from a node to a descendant leaf contains the same number of black nodes
5. The root is black

- Black-height, bh(x) = the number of black nodes on any path from x to a leaf, not counting x

![Red-Black Tree Diagram]

- Theorem: A red-black tree with n internal nodes has height at most $2\lg(n+1)$

  Proof: Not cover in this class

- So, Red-black trees have height $O(\log n)$
5. Red-Black Tree - Insertion Strategy

RedBlack_Tree_Insert(x)

- Set color[x] = Red
- Insert as in a regular BST
- Only property (3) might be violated, i.e. x’s parent is Red

- If case 1 can be applied, repeat case 1 until OK, then goto next step
- If case 2 can be applied, a left rotation is performed, then goto next step
- If case 3 can be applied, do a right rotation

- At the end, if color[root] = Red, set color[root] = Black

Assume current target node is indicated by x

- **Case 1**: x is node of interest, x's uncle is Red
12. Balanced Trees

- **Case 2**: x's uncle is Black, x is a Right child

- **Case 3**: x's uncle is Black, x is a Left child

Terminal case, tree is Red-Black tree

Note: Symmetric cases in right subtrees.

- Insertion takes $O(\log(n))$ time
- Requires at most two rotations
Example: insert 9 8 7 3 5 2 into a red-black tree in that order.

Insert 9: \((9) \rightarrow (9)\)

Insert 8: \[
\begin{array}{c}
(9) \\
/ \\
(8)*
\end{array}
\]

Insert 7: \[
\begin{array}{c}
(9) \text{ case 3} \\
/ \backslash \backslash \\
(8) \text{ nil} \rightarrow (7) (9) \\
/ \\
(7)*
\end{array}
\]

Insert 3: \[
\begin{array}{c}
(8) \text{ case 1} \\
/ \backslash \backslash \\
(7) (9) \rightarrow (7) (9) \\
/ \\
(3)*
\end{array}
\]

\[
\begin{array}{c}
(8)* \\
/ \backslash \backslash \\
(7) (9) \rightarrow (7) (9) \\
/ \\
(3)
\end{array}
\]

\[
\begin{array}{c}
(8) \\
/ \backslash \backslash \\
(7) (9) \rightarrow (7) (9) \\
/ \\
(3)
\end{array}
\]
1. Balanced Trees

Insert 5:

```
(8)         (8)        (8)
/   \       /   \       /   \          case 2       case 3          case 3
(7)    (9)   (7)    (9)   (7)    (9)           /        /          /    /
|barb2right|    |barb2right|                          /
(7)    (9)   (5)     (9)                              /            /
|                        |                        /
(3) nil      (5)      (3) (7)                      (3)         (3)*
|                        |                        /                    /
(5)*                          (3)*                (5)         (9)
```

Insert 2:

```
(8)         (8)        (8)
/   \       /   \       /   \          case 1      case 1          case 1
(5)    (9)   (5)    (9)   (5)    (9)           /        /          /    /
|barb2right|    |barb2right|                          /
(5)    (9)   (5)*   (9)                              /            /
|                        |                        /
(3) (7)      (3) (7)                          (3)         (7)
|                        |                        /                    /
(2)*                          (2)*                (2)         (2)
```