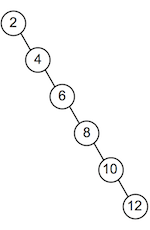
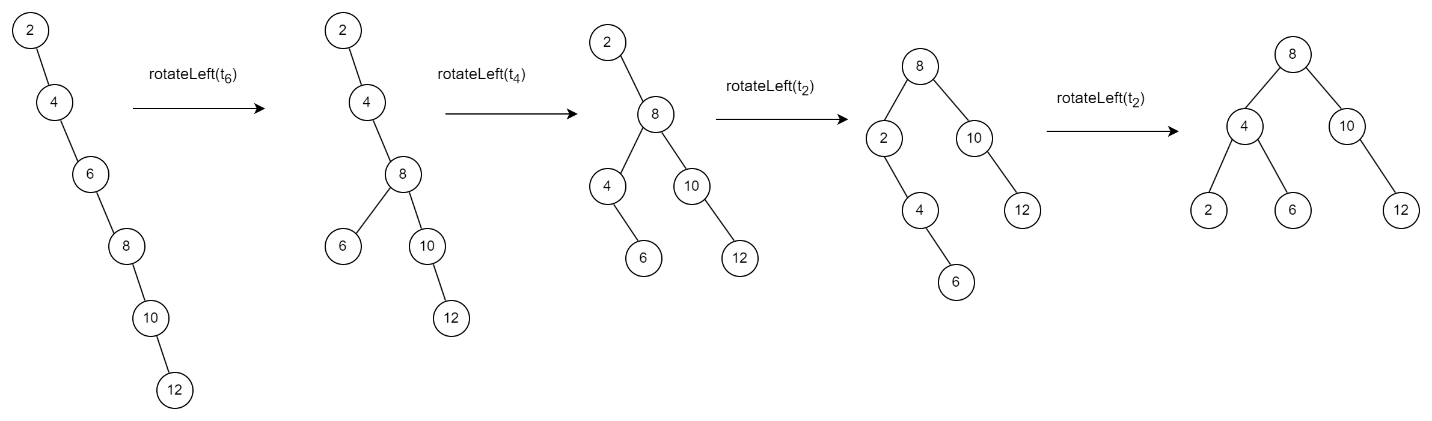
1. (Rebalancing)

Trace the execution of rebalance(t) on the following tree. Show the tree after each rotate operation.



Let be a tree rooted at node , and be the number of nodes in the tree . An operation to rebalance the tree begins by and successive recursive similar calls until the tree achieves a balanced state. In this case, a call to partition will involve a rotate operation that results into the following.

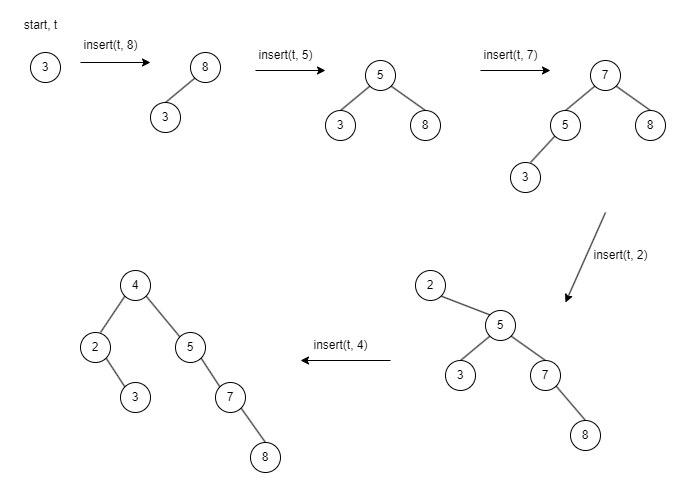


The first call to rotateLeft(t2) results into a sub-tree rooted at 2 with three nodes. This will require another rotation at root node 2 to achieve a balanced tree.

2. (Splay trees)

* 1. Show how a Splay tree would be constructed if the following values were inserted into an initially empty tree in the order given:

3 8 5 7 2 4



* 1. Let t be your answer to question a., and consider the following sequence of operations:

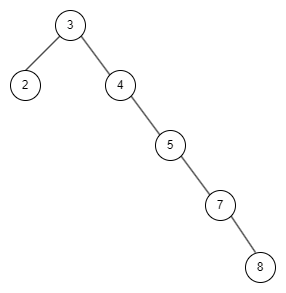
SearchSplay(t,3)

SearchSplay(t,5)

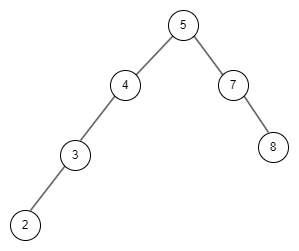
SearchSplay(t,6)

Show the tree after each operation.

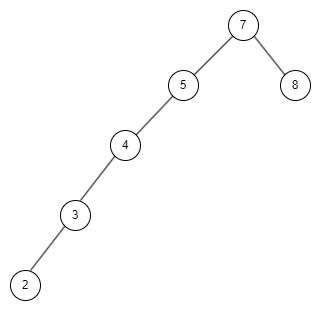
Below is the tree after the operation SearchSplay(t, 3)



Below is the tree after the operation SearchSplay(t, 5)



Below is the tree after operation SearchSplay(t, 6)



3. (AVL trees)

1. *Note: You should answer the following question without the help of the*treeLab*program from the lecture.*

Show how an AVL tree would be constructed if the following values were inserted into an initially empty tree in the order given:

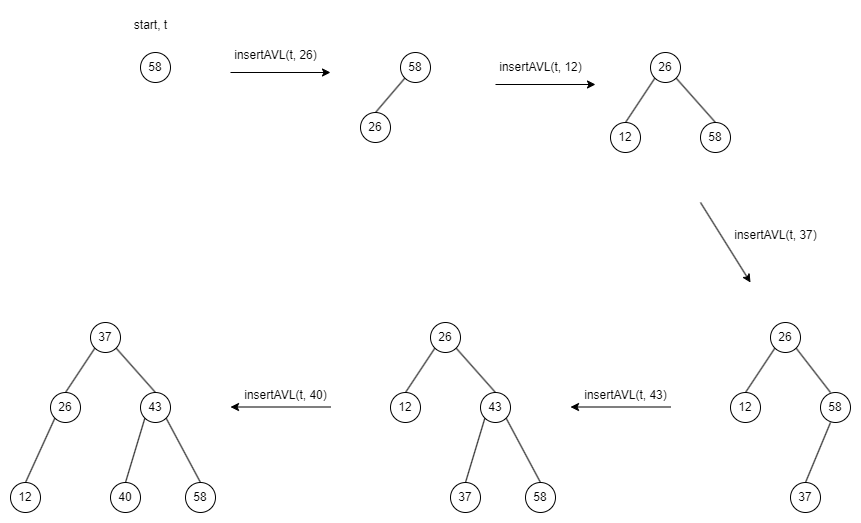
58 26 12 37 43 40

Let be the procedure that inserts into the tree t, the node n. Then following insertions will trigger rotations:

– A right rotation at the root, currently 58, to achieve balance.

- A left rotation at 37 then a right rotation at 58, in that order.

- A a right rotation at 43 followed by a left rotation at the root.



The above rotations solve imbalances that would otherwise lead to an unbalanced tree.

1. Extend the BST ADT from the lecture ([BST.h](http://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week8/progs/BST.h), [BST.c](http://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week8/progs/BST.c)) by an implementation of the function

Tree deleteAVL(Tree t, Item it)

to delete an element from an AVL tree.

----- Attached -----

1. Extend the [treeLab.c](http://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week8/progs/treeLab.c) program from the lecture by the command  'D'  to AVL-delete an element.

----- Attached -----

1. Run your program for Exercise 3c with the following sequence of commands:

a2 a3 a5 a1 a4 a6 D3

How many "repair" rotations are necessary when 3 is deleted?

One repair operation is necessary to achieve a balance in the tree after deleting the node with value 3.

How many rotations would have been necessary if you had implemented standard deletion with **version 2 for case 4** according to slides [44](https://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week7/slide044.html) and [45](https://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week7/slide045.html) (week 7)?

Similarly, one rotation would be enough for the second version of deletion when used with the AVL tree.

Which version is likely the more efficient of the two for AVL deletion?

As a result, both versions have a similar efficiency.

4. (Lazy deletion)

There are (at least) two approaches to dealing with deletions from binary search trees. The first, as used in lectures, is to remove the tree node containing the deleted value and re-arrange the pointers within the tree. The second is to not remove nodes, but simply to mark them as being "deleted".

For this question, assume that we are going to re-implement Binary Search Trees so that they use mark-as-deleted rather than deleting any nodes. Under this scheme, no nodes are ever removed from the tree; instead, when a value is deleted, its node remains (and continues to retain the same value) but is marked so that it can be recognised as deleted.

To implement this idea of "lazy" deletion, consider the following modification to the BST data structure:

typedef struct Node {

bool deleted;

int data;

Tree left, right;

} Node;

* 1. Modify the [search algorithm](https://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week7/slide023.html) (week 7 lecture) for a "conventional" binary search tree to take into account deleted nodes.

TreeSearch(tree,item):

   Input  tree, item

   Output true if item found in tree, false otherwise

   if tree is empty then

        return false

   else if item < data(tree) then

        return TreeSearch(left(tree),item)

   else if item > data(tree) then

        return TreeSearch(right(tree),item)

   else // found

        if item is marked deleted then // item is deleted

            return false

        else  // item is NOT marked as deleted so its found

            return false

        end if

   end if

* 1. One significant advantage of deletion-by-marking is that it makes the deletion operation simpler. All that deletion needs to do is search for a node containing the value to be deleted. If it finds such a node, it simply "marks" it as deleted. If it does not find such a node, the tree is unchanged.

Write a deletion algorithm that takes a BST t and a value v and returns a new tree which does not contain an undeleted node with value v.

Tree deleteNode(Tree tree, Item v)

    Input  tree, v

    Output new tree which does not contain an undeleted node with value v

    if t is empty then

        return NULL;

    if v < data(tree) then

        left(tree) = deleteNode(left(tree), v);

    else if v > data(tree) then

        right(tree) = deleteNode(right(tree), v);

    else // Found the node with value v

        tree->deleted = true; // Mark the node deleted

    end if

    return t;

* 1. The most problematic aspect of deletion-by-marking is insertion. If handled naively, the tree grows as if it contains *n+d* values, where *n* is the number of nodes containing undeleted values and *d* is the number of nodes containing deleted values. If many values are deleted, then the tree becomes significantly larger than necessary.

A more careful approach to insertion can help to limit the growth of the tree by re-using nodes containing deleted values. Modify the algorithm for [AVL tree insertion](https://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week8/slide041.html) from the lecture to re-use deleted nodes where possible without causing an imbalance.

insertAVL(tree, item):

    Input  tree, item

    Output tree with item AVL-inserted

    if tree is empty then

        return new node containing item

    else if item = data(tree) and tree is marked as deleted then

        mark the tree as undeleted

        return tree

    else

        if item < data(tree) then

            left(tree) = insertAVL(left(tree), item)

        else if item > data(tree) then

            right(tree) = insertAVL(right(tree), item)

        end if

        if height(left(tree)) - height(right(tree)) > 1 then

            if item > data(left(tree)) then

                left(tree) = rotateLeft(left(tree))

            end if

            tree = rotateRight(tree)

        else if height(right(tree)) - height(left(tree)) > 1 then

            if item < data(right(tree)) then

                right(tree) = rotateRight(right(tree))

            end if

            tree = rotateLeft(tree)

        end if

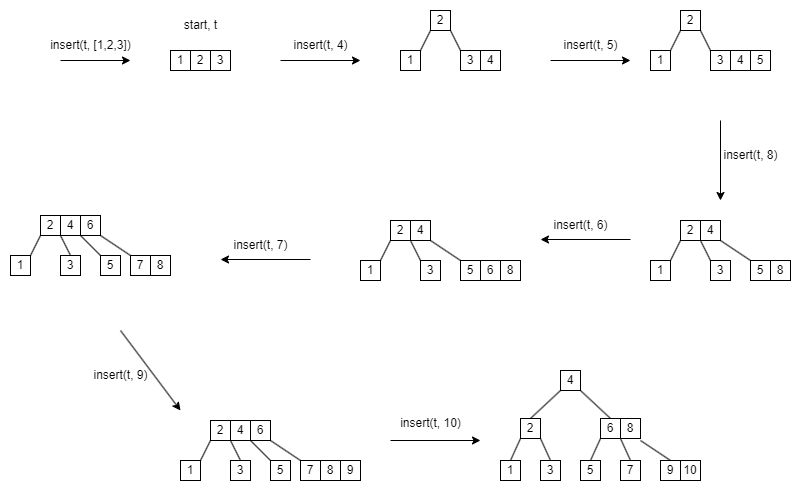
        return the updated tree

    end if

1. (2-3-4 trees)

Show how a 2-3-4 tree would be constructed if the following values were inserted into an initially empty tree in the order given:

1 2 3 4 5 8 6 7 9 10



Once you have built the tree, count the number of comparisons needed to search for each of the following values in the tree:

1 7 9 13

Let cmp(key, v) be a comparison between the search key and other keys in the tree. Thus, the above searches can be completed as follows with the resulting costs.

Search(t, 1); cmp(1, 4), cmp(1, 2), then cmp(1, 1), totaling to a cost = 3 comparisons.

Search(t, 7); cmp(7,4), cmp(7,6), cmp(7,8), cmp(7,7), totaling to a cost = 4 comparisons.

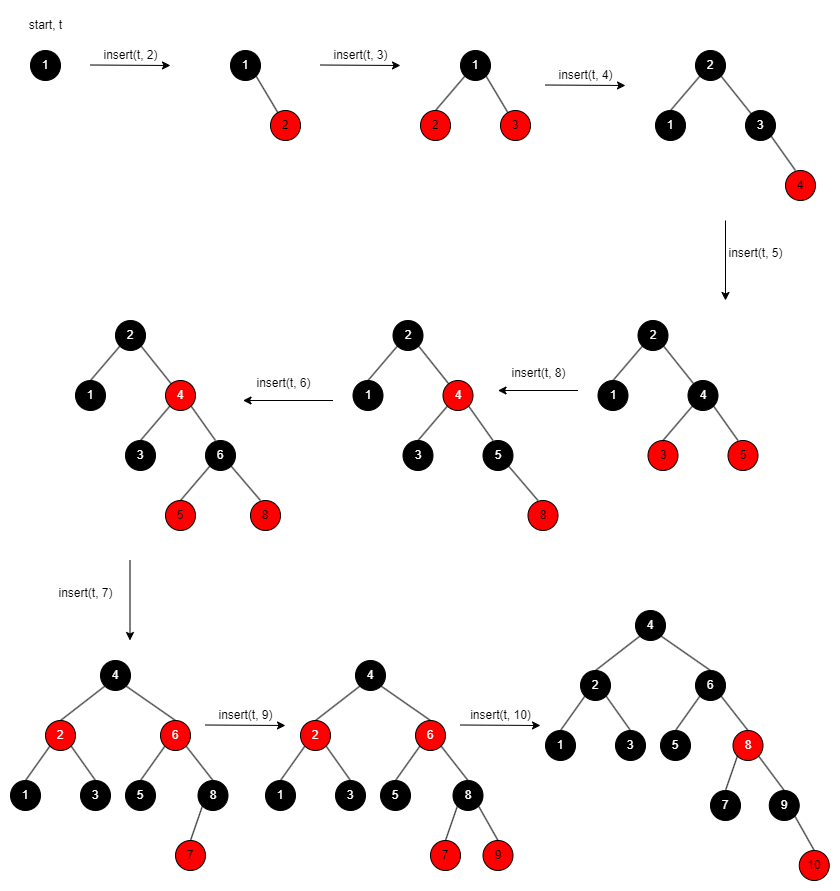
Search(t, 9): cmp(9,4), cmp(9,6), cmp(9,8), cmp(9,9), totaling to cost = 4 comparisons.

Search(t, 13); cmp(13, 4), cmp(13, 6), cmp(13, 8), cmp(13, 9), cmp(13, 10) resulting into cost = 5 comparisons.

6. (Red-black trees)

1. Show how a red-black tree would be constructed if the following values were inserted into an initially empty tree in the order given:

1 2 3 4 5 8 6 7 9 10



Once you have built the tree, compute the cost (#comparisons) of searching for each of the following values in the tree:

1 7 9 13

Search(t, 1); cmp(1, 4), cmp(1, 2), cmp(1, 1) => cost = 3 comparisons

Search(t, 7); cmp(7, 4), cmp(7, 6), cmp(7, 8), cmp(7, 7) => cost = 4 comparisons

Search(t, 9); cmp(9, 4), cmp(9, 6), cmp(9, 8), cmp(9, 9) => cost = 4 comparisons

Search(t, 13); cmp(13, 4), cmp(13, 6), cmp(13, 8), cmp(13, 9), cmp(13, 10) implies that cost is 5 comparisons.

1. Implement the pseudocode for Red-Black Tree Insertion from the lecture ([slides 70–74](http://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week8/slide070.html)) in our Red-Black Tree ADT ([RBTree.h](http://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week8/progs/RBTree.h), [RBTree.c](http://www.cse.unsw.edu.au/~cs9024/23T2/lecs/week8/progs/RBTree.c)) as the function:

Tree TreeInsert(Tree t, Item it) { ... }