CS:3330 Algorithms
Midterm Exam (100 points)

Closed books and notes (except one sheet of notes)
None of the digital devices is allowed.
10/10/2019

1. (40 points) We insert the following numbers in the given order into an empty binary search tree. (**a)** If the tree is an AVL tree, for each rotation in an insertion, please display the tree before and after rotation. If no rotation happens during an insertion, simply draw the tree after the insertion. Please repeat the above task when the tree is (**b)** a Red-Black tree (use single circle for black nodes and double circle for red nodes) and (**c**) a splay tree:

   5, 1, 4, 3, 2.

2. (30 points) Given a binary search tree t (which is a tree node served as the root of the tree) and an integer x served as key, the predecessor of x in t is the maximal node among all nodes in t whose key is less than x. Please write a non-recursive algorithm (in pseudo-code) *predecessor*(t, x), which takes t and x and returns the predecessor of x in t; if no such node exists, returns null. The available methods include only *key*(n), *isNull*(n), *leftChild*(n), *rightChild*(n), *minimum*(n), and *maximum*(n), where n is a tree node and their meanings are given in the class.

Answer:

```
treeNode predecessor(treeNode t, int x)
    treeNode pred = null
    while true
        if isNull(t) return pred  // x doesn't appear in t
        if (key(t) < x)
            pred = t;            // pred remembers the last right turn
            t = rightChild(t)
        else if (key(t) > x)
            t = leftChild(t)
        else // key(t) = x
            if (isNull(leftChild(t))) return pred
            return maximum(leftChild(t))
```

The worst complexity of predecessor is O(h), where h is the height of t.

3. (30 points) Given a binary tree t (which is a tree node served as the root of the tree), a node n in t is called *core node* if the distance from n to any node of degree 1 (which has a single neighbor, either a child or a parent) is at least 2. For example, if the tree is a chain of 5 nodes, then only the middle node is a core node. Please design an efficient

algorithm, called markCores(t), which marks all the core nodes in the tree t. The available methods include only *isNull*(n), *leftChild*(n), *rightChild*(n), and setCore(n), where n is a tree node, and setCore(n) will mark n as a core node.

Answer:

The case when the root has a single child is special and we handle it in the beginning of markCores(t). The general case is when a node either has a parent or two children and we use a recursive procedure to handle them. The recursive procedure recMarkCore(t) will return true iff t is a leaf node.

```
markCores(treeNode t)
    boolean good = true // Is it good to set t as core code?
    if (isNull(t)) return   // special case: root t is null
    if (isNull(leftChild(t))) // special case: root's left child is null
        if (isNull(rightChild(t)) return // special case: the tree rooted by t is a single node
        t = rightChild(t)
        good = false
    else if (isNull(rightChild(t))) // special case: root's right child is null
        t = leftChild(t)
        good = false
    // general case: t has either a parent or two children; use a recursive procedure
    recMarkCore(t, good)

 boolean recMarkCore(treeNode t, boolean good)
   // Pre: t is not null and t has either a parent or two children.
   // Post: Visit each node in the subtree rooted by t;
            During the visit, mark t as core if neither t is a leaf nor one of its children is a leaf;
            Return true iff t is a leaf node.
   if (isNull(leftChild(t)) && isNull(rightChild(t))) return true // t is a leaf
   if (!isNull(leftChild(t)) && recMarkCore(leftChild(t), true)) good = false  // lefttChild(t) is a leaf
   if (!isNull(rightChild(t)) && recMarkCore(rightChild(t), true)) good = false  // rightChild(t) is a leaf
   if (good) setCore(t)
   return false  // t is not a leaf node
```

The complexity of markCores depends on that of recMarkCore. The complexity of recMarkCore is O(n) because it visits each node once and does a constant number of jobs at each node.

When we use setCore(t), we assume that the initial mark is non-core for every node. If the initial mark is core for every node, and we have a method called unsetCore(t), which mark t as non-core, the algorithm goes as follows:

```
markCores(treeNode t)
    if (isNull(t)) return   // special case: root t is null
    if (isNull(leftChild(t))) // special case: root's left child is null
        unsetCore(t)
        if (isNull(rightChild(t)) return // special case: the tree rooted by t is a single node
        t = rightChild(t)
        unsetCore(t)
    else if (isNull(rightChild(t))) // special case: root's right child is null
        unsetCore(t)
        t = leftChild(t)
        unsetCore(t)
```

```
     // general case: t has either a parent or two children; use a recursive procedure
     recMarkCore(t)

boolean recMarkCore(treeNode t)
    // Pre: t is not null and t has either a parent or two children.
    // Post: Visit each node in the subtree rooted by t;
              During the visit, mark t as core if either t is a leaf or one of its children is a leaf;
              Return true iff t is a leaf node.
    if (isNull(leftChild(t)) && isNull(rightChild(t))) // t is a leaf node
         unsetMark(t)
         return true  // t is a leaf node
    if (!isNull(leftChild(t)) && recMarkCore(leftChild(t))) unsetMark(t)  // lefttChild(t) is a leaf node
    if (!isNull(rightChild(t)) && recMarkCore(rightChild(t))) unsetMark(t)  // rightChild(t) is a leaf node
    return false  // t is not a leaf node
```