CS106X
Autumn 2015 Cynthia Lee

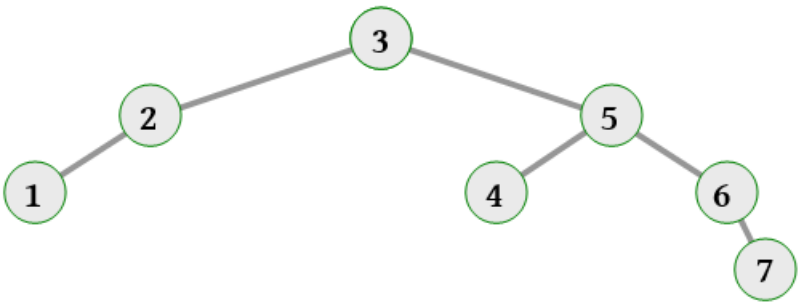
Section 6 (Week 7) Handout

Problem and solution authors include Jerry Cain and Cynthia Lee.

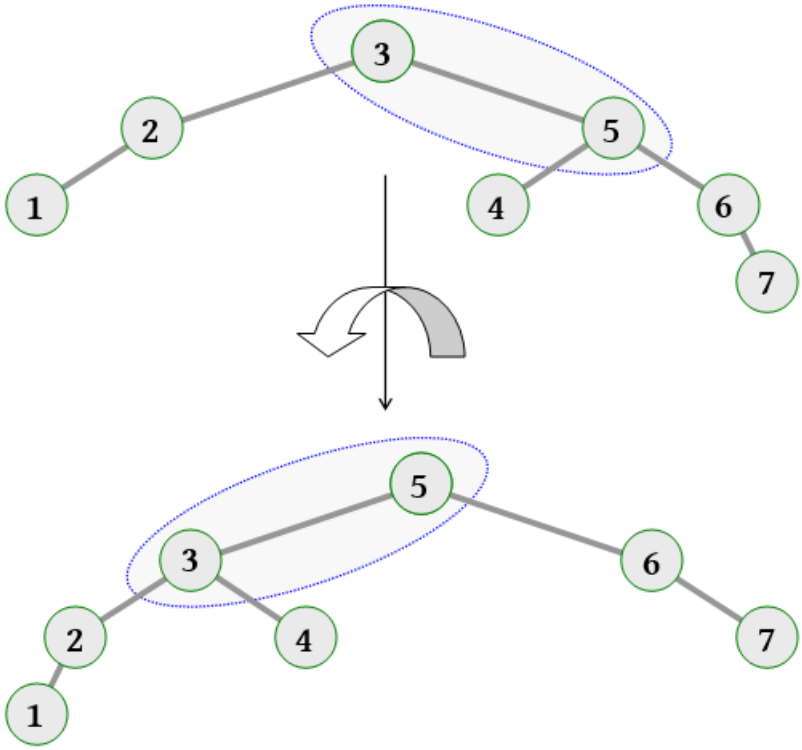
Problem 1: Tree Rotations

Recall in class we saw the video of the insertions to a **red-black tree**, which first followed the usual insert algorithm (go left if key being inserted is less than current, go right if greater than current), and then did additional "rotations" that rebalanced the tree. Now is your chance to explore the code that makes the rebalancing happen. Rotations have other uses; for example, in Splay trees, they can be used to migrate more commonly-used elements towards the root.

Suppose that a binary search tree has grown to store the first seven integers, as has the tree drawn to the right. Suppose that a program then searches for the 5. Clearly the search would succeed, but the standard implementation would leave the node in the same position, and later searches for the same element would take the same amount of time.



A cleverer implementation would anticipate another search for 5 is likely, and would change the pointer structure in the vicinity of the 5 by performing what is called a **left-rotation**. A left rotation pivots around the link between a node and its parent, so that the child of the link rotates counterclockwise to become the parent of its parent. In our example above, a left-rotation would change the structure according to the figure on the right. Note that the restructuring is a local operation, in that only a constant number of pointers need to be updated. The key observation is that 5 is brought one level closer to the root, the



former parent of 5 becomes 5's left child and in the process inherits what **used** to be the left child of 5 as its right child. In general, these two nodes might occur anywhere in a binary search tree. Notice that the binary search tree order property is maintained.

a) Write a function `rotateLeft` which performs the left rotation operation, like the one shown in the figure above. It takes the parameter `toRotateAddr`, which is actually **the address of (!) a pointer to a node** you want to rotate left (for example, the *address of* a pointer to the node "3" in the first illustration above). You'll want to immediately change it to simply a pointer to the node you want to rotate by using dereference:

```
node *toRotate = *toRotateAddr;
```

You'll do most of your operations on `toRotate`, then in the very last step you'll need to use `toRotateAddr` to redirect the parent (or tree class "root" pointer, in the case of the root). For simplicity, you may assume that both the referenced node and its right child are both non-`NULL`.

```
struct node {
   int value;
   node *left;
   node *right;
};

static void rotateLeft(node **toRotateAddr);
```

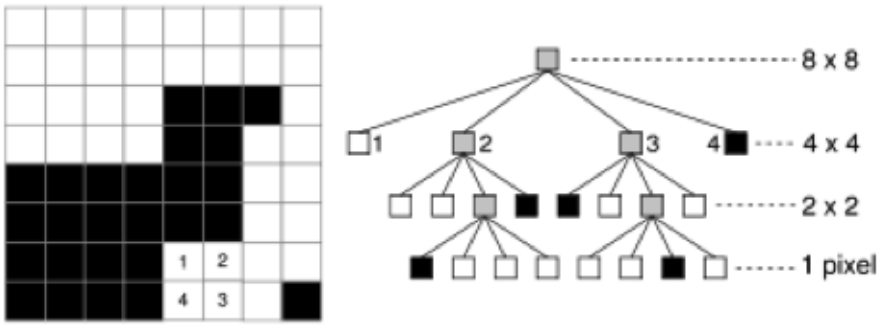
b) Now, using the `rotateLeft` operation you just wrote, along with its symmetric counterpart `rotateRight` (which you can assume works properly without actually writing it), implement `pullToRoot`, which takes the address of a pointer to the root of a binary search tree and bubbles the specified value up to the root. You may assume that the value is actually present, although a particularly clever implementation would leave the tree unaltered if the value is missing.

```
static void pullToRoot(node **rootAddr, int value);
```

Problem 2: Quadtrees

A quadtree is a rooted tree structure where each internal node has precisely four children. Every node in the tree represents a square, and if a node has children, each encodes one of that square's four quadrants.

Quadtrees have many applications in computer graphics, because they can be used as in-memory models of images. That they can be used as in-memory versions of black and white images is easily demonstrated via the following (borrowed from Wikipedia.org):



The 8 by 8 pixel image on the left is modeled by the quadtree on the right. Note that all leaf nodes are either black or white, and all internal nodes are shaded gray. The internal nodes are gray to reflect the fact that they contain both black **and** white pixels. When the pixels covered by a particular node are all the same color, the color is stored in the form of a Boolean and all four children are set to `NULL`. Otherwise, the node's sub-region is recursively subdivided into four sub-quadrants, each represented by one of four children.

Given a `Grid<bool>` representation of a black and white image, implement the `gridToQuadtree` function, which reads the image data, constructs the corresponding quadtree, and returns its root. Frame your implementation around the following data structure:

```
struct quadtree {
   int lowx, highx; // smallest and largest x value covered by node
   int lowy, highy; // smallest and largest y value covered by node
   bool isBlack; // entirely black? true.  Entirely white? False. Mixed? ignored
   quadtree *children[4]; // 0 is NW, 1 is NE, 2 is SE, 3 is SW
};
```

Assume the lower left corner of the image is the origin, and further assume the image is square and that the dimension is a perfect power of two.

```
static quadtree *gridToQuadtree(Grid<bool>& image);
```