Lecture 25

Splay Trees

- Splay Trees
  - Intuition and Basic Ideas
  - Rotations and Splays
  - History
  - Iterator Issues
  - Summary
- Lab 8 – Preview (Part 1)
  - Darwin

Splay Trees

Splay Trees

Binary search trees are very common in both the theory and practice of Computer Science.

For this reason, there are many variations that are studied and used in industry.

In the last lecture, we briefly introduced several variations, including splay trees.

In this lecture, we'll look more closely at splay trees, which are discussed in §14.5–14.6.

Intuition and Basic Ideas

Intuition: Rising to the Top

In many applications, there will be values that are accessed more frequently than others. Furthermore, we likely won't know in advance which of the values will be accessed more often, and moreover, the distribution of accesses may change over the life of the data structure.

The intuition behind a splay tree is the following:

The most recently accessed values are more likely to be accessed again in the near future, so we can improve performance by dynamically moving these values upward in the tree.

In other words, the data structure reacts to each `find` by moving the queried value up the tree. In fact, it takes this approach to the extreme: The value is moved all the way up to the root. Furthermore, it moves values to the root during the other operations (e.g., `insert`, `delete`).

This idea makes common sense, but it raises a number of questions and concerns.

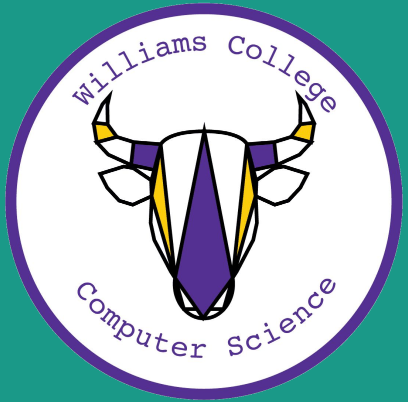
- How can we move a value upward while maintaining the subtree condition?
  We must maintain this condition otherwise the tree is not a binary search tree and the log run-time of `find` will be lost.
- Moving a value upward will cause other values to move downward.
  This means that the run-time of some subsequent `find` operations will be increased.

Class Discussion: (Temporal) Locality and Data Assumptions

The tendency of recent values being accessed again in the near future is known as *temporal locality*.
When does this occur in practice? In other words, when does the intuition behind a splay tree hold?
It will depend on the specific type of data being stored.

Let's discuss these specific cases:

- Patient records at a hospital.
- Student records at a college.



Sticker design by Iris Howley

Advanced data structures are often designed to take advantage of various types of locality (e.g., temporal, spatial, memory, branch, equidistant).

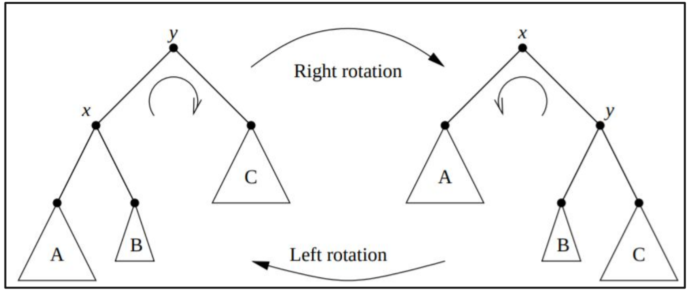
As a computer scientist, it is important to note that different applications will have different data assumptions, and to consider which data structures will perform better under these assumptions.

Basic Ideas: Rotations

How can we move a value upward in a binary search tree while maintaining the subtree condition?
One approach is a *left* or *right tree rotation*, which makes the following modifications to the tree.

- A parent and child exchange levels.
  - The child moves up one level into its parent's position. One of the child's subtrees moves upward with the child.
  - The parent moves down one level into the position of its other child. One of the parent's subtrees moves downward.
- One of the subtrees of the child becomes a subtree of the parent after the exchange.

A rotation can be specified by stating the parent node, and which direction it will be rotated.
A rotation can also be specified by stating the edge to rotate, or by stating the node to move upward.



When read left-to-right, Figure 14.4 shows a right-rotation on node y, or a rotation on edge xy, or a rotation that moves x up. Reading right-to-left, it is a left-rotation on x, a rotation on edge xy, or move y up. Note: Rotations can also be done on non-roots.

We'll focus on implementations that change the links of the tree, and not the individual values.
In other words, we adjust `.left` and `.right`, and not the `.value` property of a node.

Basic Ideas: Splay Operations

When we perform operations in a binary search tree (e.g., `find`, `insert`, `delete`) we traverse the tree from the root to a specific node of interest.

In a splay tree, we repeat each traversal in reverse, going from the node of interest up to the root. At each level, we will perform tree rotations that move the node of interest to the root. Collectively, these rotations are referred to as a *splay* operation.

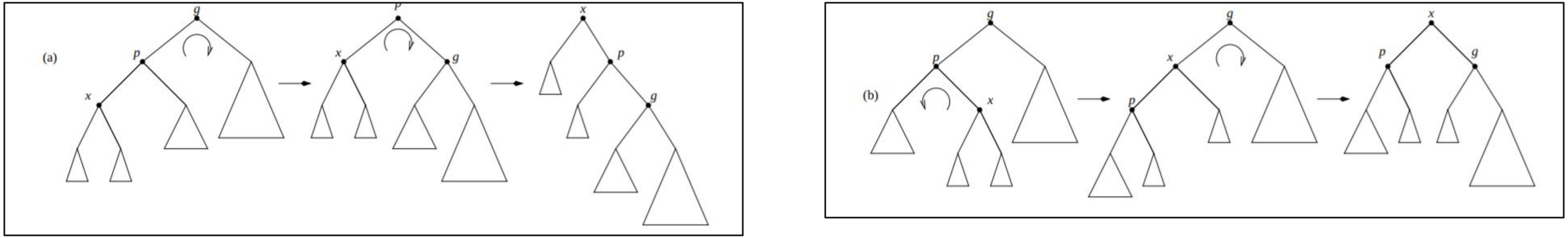


Figure 14.5 shows two cases for a single step of the splay operation (with two other cases being mirror copies).
Note: These illustrations are again focused on the root; the splay operation will often start lower in the tree.

The definition of the splay operation will require close attention.

- Each step performs two tree rotations. (Except when the node is at level 0 or level 1.)
- The order of the two tree rotations depends on what type of grandchild the node is.

Rotations and Splays

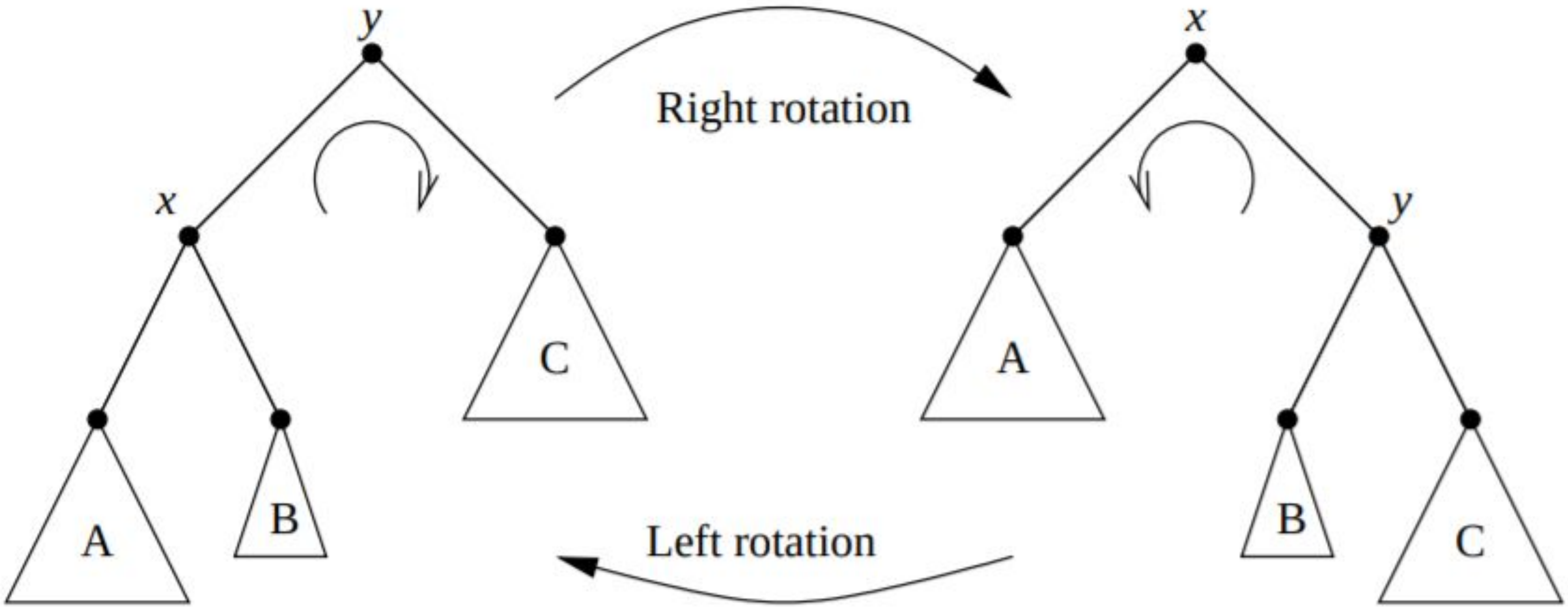
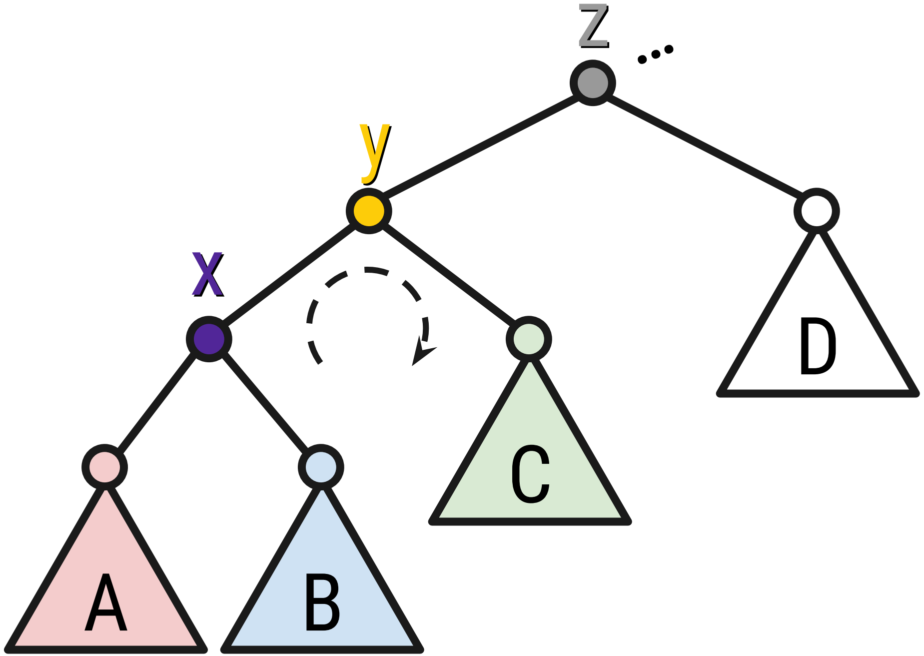
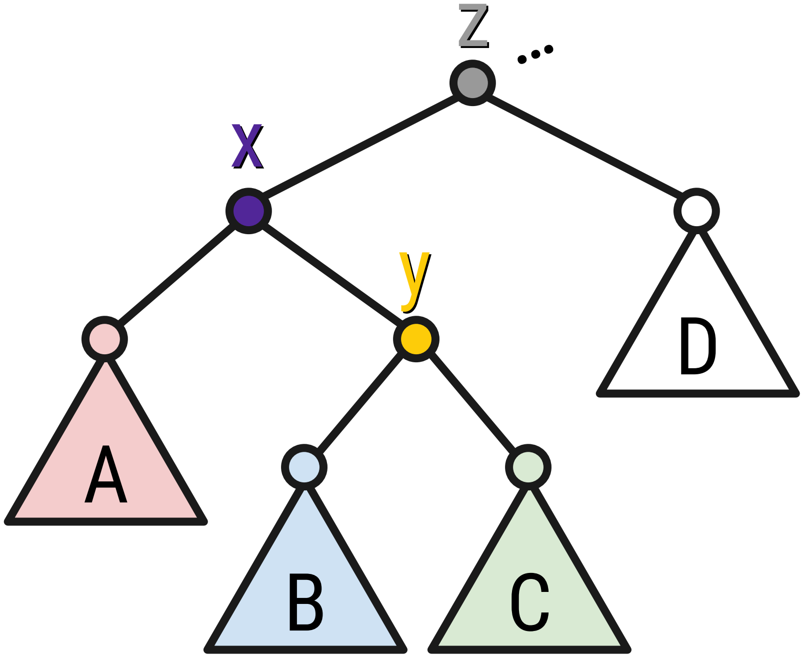


Figure 14.4 The relation between rotated subtrees.

The textbook's illustration of tree rotations. (No assumptions are made about the sizes of A, B, C.)
In the next slide, we'll illustrate the before and after of a right-rotation with nodes added above y.



Before the right rotation on `y`.



After the rotation.

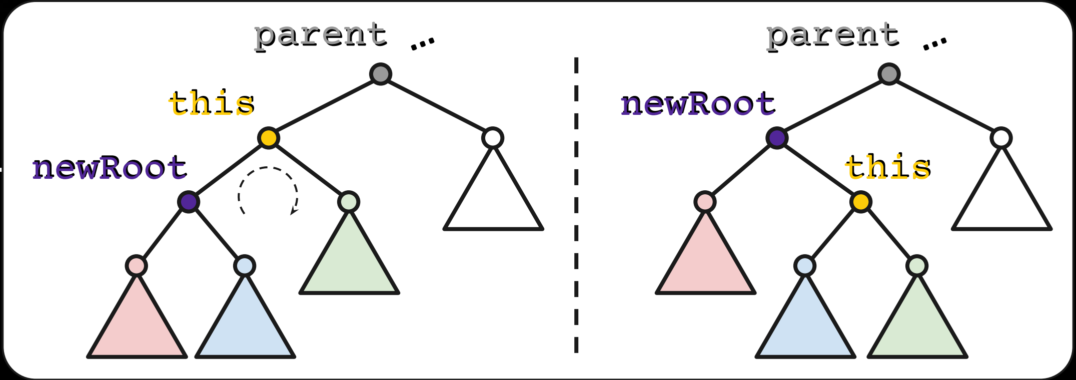
A right-rotation on node `y`. (A subsequent left-rotation on `x` would return the tree to its prior state.)

- Why does the subtree condition hold after the rotation? If value `b` is in subtree B, then `x` ≤ `b` ≤ `y`.
- In the `structure` package, this is `y.rotateRight()` (or `rotateRight()` from in `y`). What are all of the references that will change? How do we properly change `z`'s references?

```java
 * Method to perform a right rotation of tree about this node
 * Node must have a left child.  Relation between left child and node
 * are reversed
 *
 * @pre This node has a left subtree
 * @post Rotates local portion of tree so left child is root
 */
protected void rotateRight()
{
    BinaryTree<E> parent = parent();
    BinaryTree<E> newRoot = left();
    boolean wasChild = parent != null;
    boolean wasLeftChild = isLeftChild();

    // hook in new root (sets newRoot's parent, as well)
    setLeft(newRoot.right());

    // puts pivot below it (sets this's parent, as well)
    newRoot.setRight(this);

    if (wasChild) {
        if (wasLeftChild) parent.setLeft(newRoot);
        else              parent.setRight(newRoot);
    }
}
```

```java
 * Update the right subtree of this node.  Parent of the right subtree
 * is updated consistently.  Existing subtree is detached
 *
 * @post Sets left subtree to newRight
 *       re-parents newRight if not null
 *
 * @param newRight A reference to the new right subtree of this node
 */
public void setRight(BinaryTree<E> newRight)
{
    if (isEmpty()) return;
    if (right != null && right.parent() == this) right.setParent(null);
    right = newRight;
    right.setParent(this);
}
```

```java
 * Determine if this node is a left child
 *
 * @post Returns true if this is a left child of parent
 *
 * @return True iff this node is a left child of parent
 */
public boolean isLeftChild()
{
    if (parent() == null) return false;
    return this == parent().left();
}
```

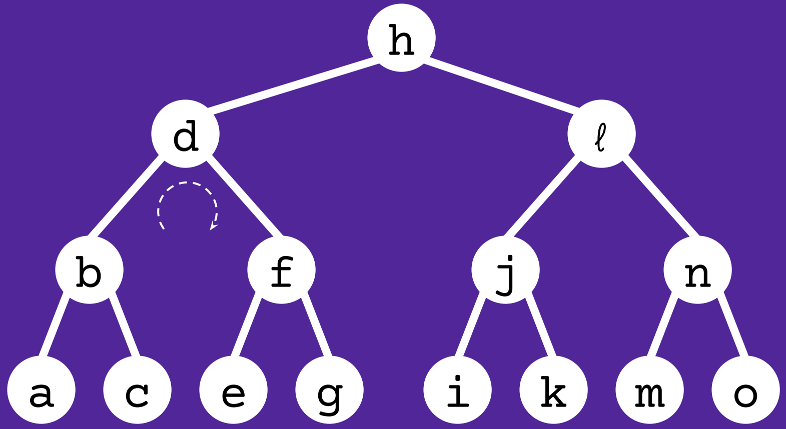


The method `rotateRight()` in `BinaryTree.java` and two helper functions.
The rotation acts on itself (i.e., the `this` instance of `BinaryTree`) so there are no arguments.
Note that `left` and `right` properties are changed and not the `value` property.

Exercise: Tree Rotation

Perform a single right rotation around node `d` in the following binary tree.
Note: This is an exercise on rotation, not on splaying.

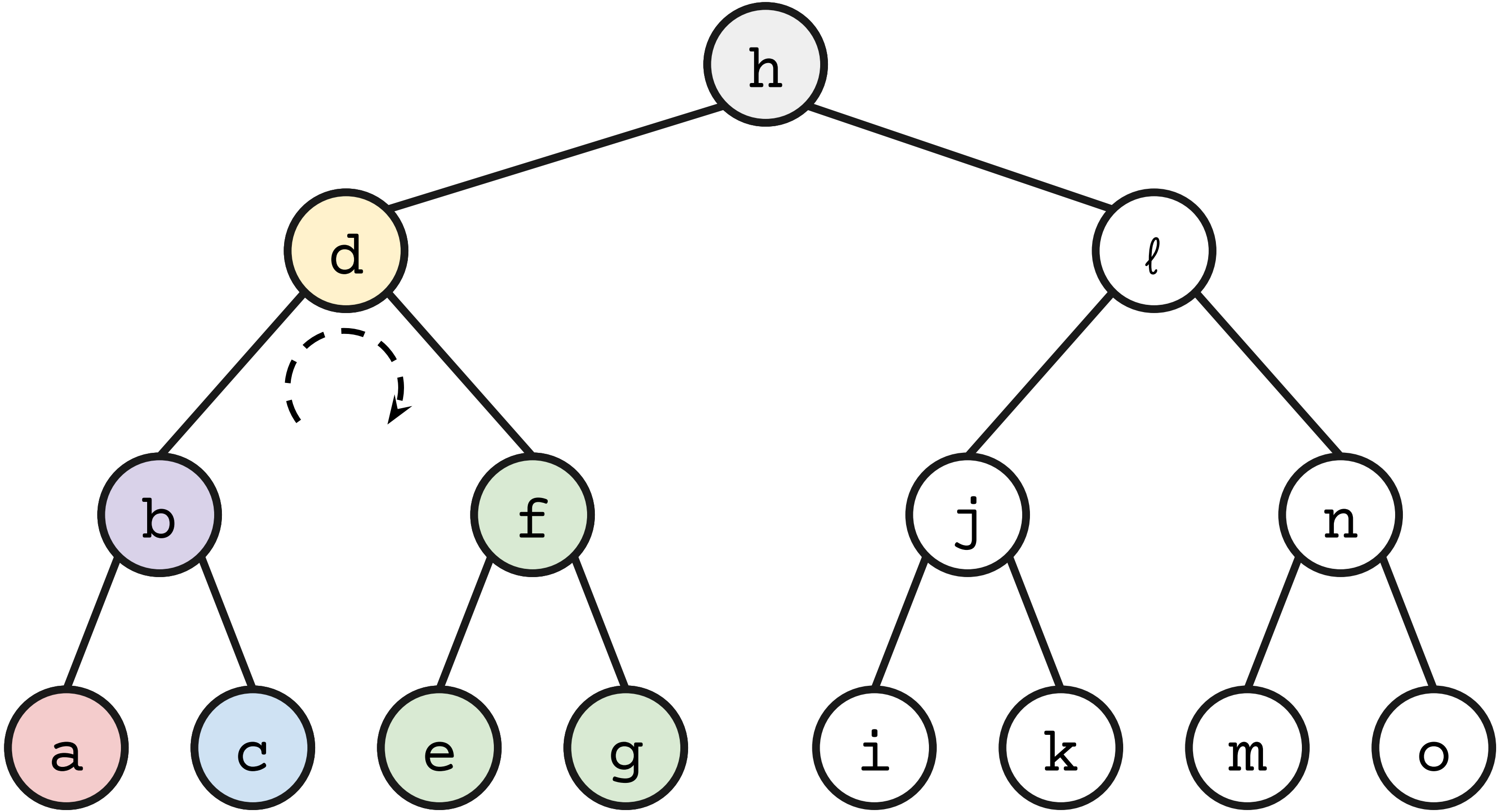
Write your answer for 2 minutes.
Then trade notes with a neighbor for 1 minute.



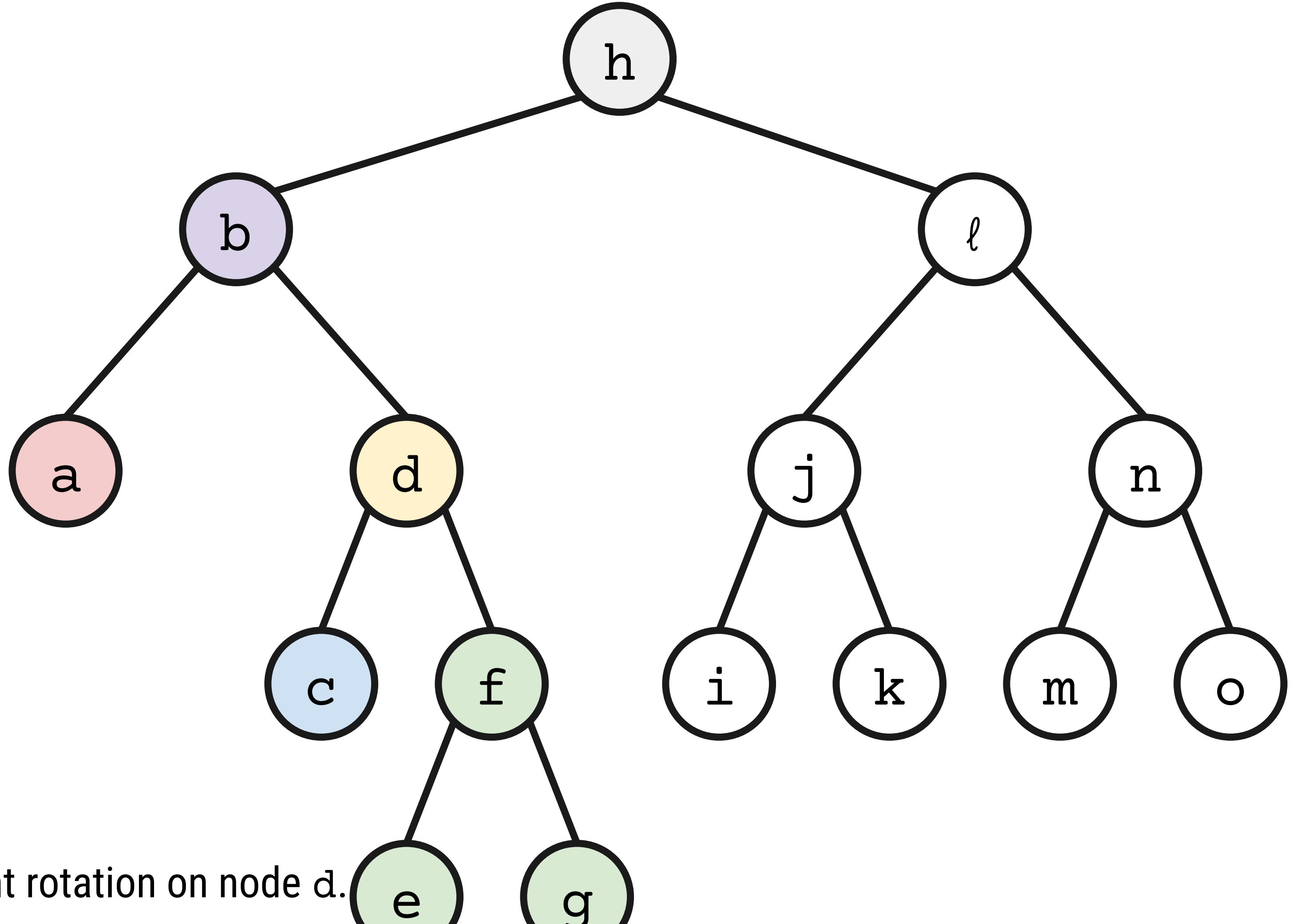
A binary search tree.
Perform a right rotation on node d.

Questions:

- Which subtree moves?
- Does your tree still satisfy the subtree condition?



Before: Right rotation on node `d`.



After: Right rotation on node d.

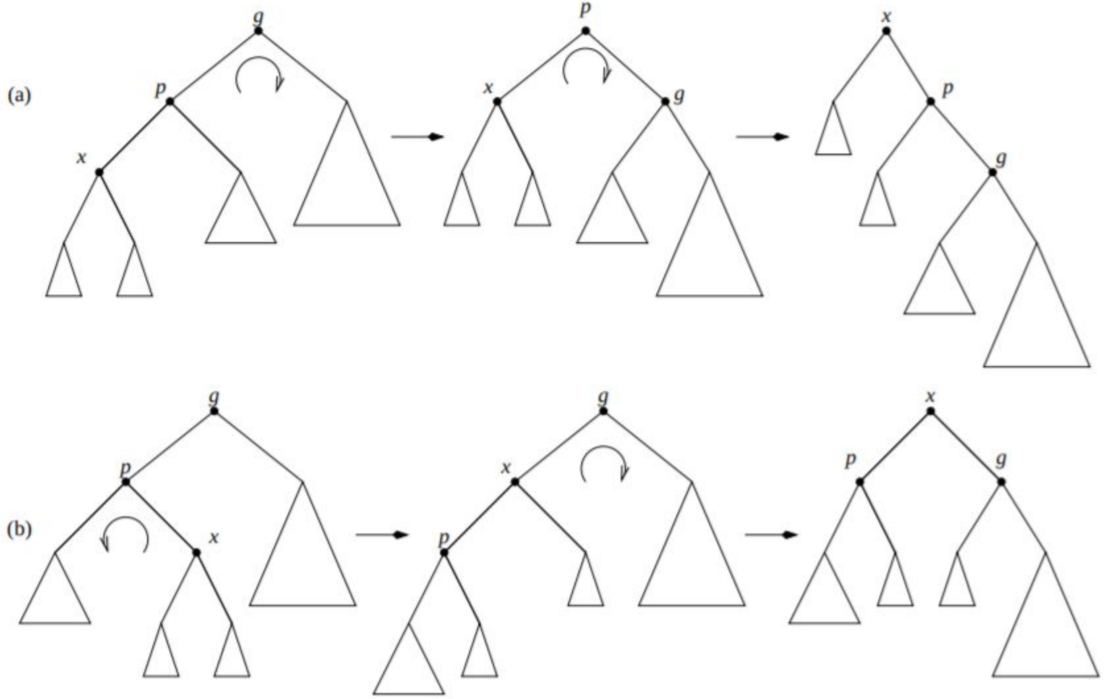


Figure 14.5 Two of the rotation pairs used in the splaying operation. The other cases are mirror images of those shown here.

A single *splay step* usually moves the node `x` upward two levels in the tree via two rotations.

- There are four cases depending on whether `x` and its parent are left or right children. This figure illustrates two of the four cases, and the other two cases mirror those above.

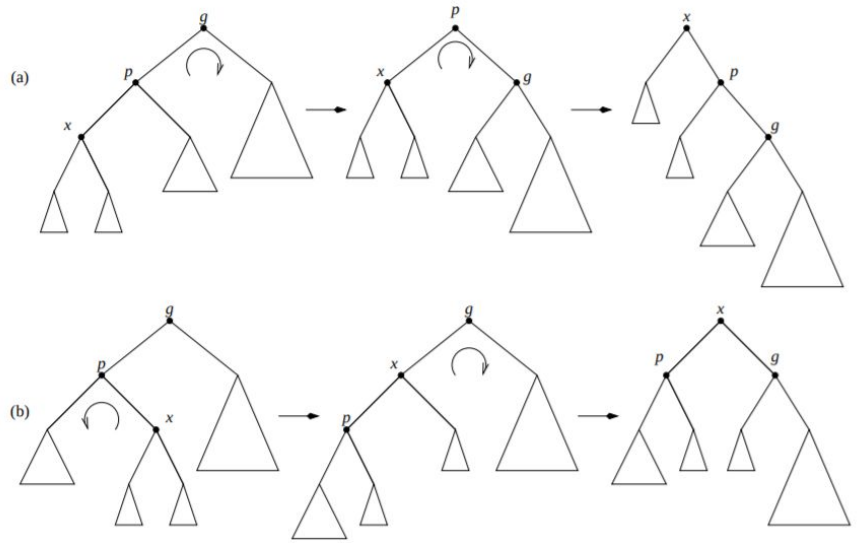


Figure 14.5 Two of the rotation pairs used in the splaying operation. The other cases are mirror images of those shown here.

- If x is the root, we are done.
- If x is a left (or right) child of the root, rotate the tree to the right (or left) about the root. x becomes the root and we are done.
- If x is the left child of its parent p, which is, in turn, the left child of its grandparent g, rotate right about g, followed by a right rotation about p (Figure 14.5a). A symmetric pair of rotations is possible if x is a left child of a left child. After double rotation, continue splay of tree at x with this new tree.
- If x is the right child of p, which is the left child of g, we rotate left about p, then right about g (Figure 14.5b). The method is similar if x is the left child of a right child. Again, continue the splay at x in the new tree.

The textbook's presentation of a single splay step, stated in terms of the node `x` that is moving up.

- The second bullet handles the cases where `x` is at level 1. These cases are often called the ***zig cases***.
- The third bullet handles the cases where `x` is a *left-left grandchild* or a *right-right grandchild*.
  The left-left case is in Figure 14.5 (a). There is a typo in this bullet: "left child of a left child" → "right child of a right child".
  Pay attention to the order of rotations. These are often called the ***zig-zig cases***.
- The fourth bullet handles the cases where `x` is a *left-right grandchild* or a *right-left grandchild*.
  The left-right case is in Figure 14.5 (b). Pay attention to the order of rotations. These are often called the ***zig-zag cases***.

```
protected void splay(BinaryTree<E> splayedNode)
{
    BinaryTree<E> parent,grandParent;

    while ((parent = splayedNode.parent()) != null)
    {
        if ((grandParent = parent.parent()) == null)
        {
            if (splayedNode.isLeftChild()) parent.rotateRight();
            else parent.rotateLeft();
        }
        else
        {
            if (parent.isLeftChild())
            {
                if (splayedNode.isLeftChild())
                {
                    // notice the order of this rotation.
                    // not doing this in order works, but not
                    // efficiently.
                    grandParent.rotateRight();
                    parent.rotateRight();
                }
                else
                {
                    parent.rotateLeft();
                    grandParent.rotateRight();
                }
            }
            else
            {
                if (splayedNode.isRightChild()) {
                    grandParent.rotateLeft();
                    parent.rotateLeft();
                }
                else
                {
                    parent.rotateRight();
                    grandParent.rotateLeft();
                }
            }
        }
    }
```

The method `splay()` in `SplayTree.java`.

- Unlike the tree `rotate` methods, this method has an argument, with `splayedNode` corresponding to node `x` in the previous figures.
- The `while` loop continues until `splayedNode` is the root (i.e., when its `parent` is `null`).
- The first `if` statement handles the zig cases. That is, `splayedNode` is at level 1 (i.e., its `grandparent` is null).
- The remaining `if` statements handle the zig-zig and zig-zag cases.

The `isLeftChild()` method is run on `parent` and on `splayedNode` to determine which specific zig / zig-zig / zig-zag case we are in.

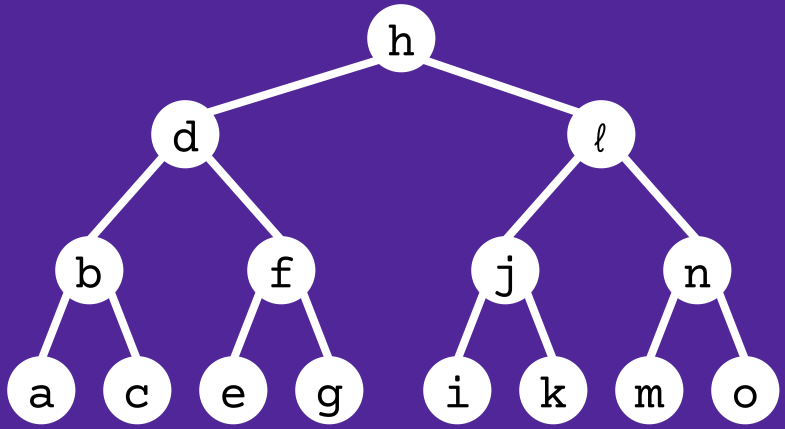
The comment gives insight into the rotation order. We'll look more into this in a subsequent slide.

Exercise: Splay Operation

Each splay operation consists of a sequence of tree rotations.
Provide the rotation sequence for the following cases.

- Splay on node `a`.
- Splay on node `c`.
- Splay on node `n`.

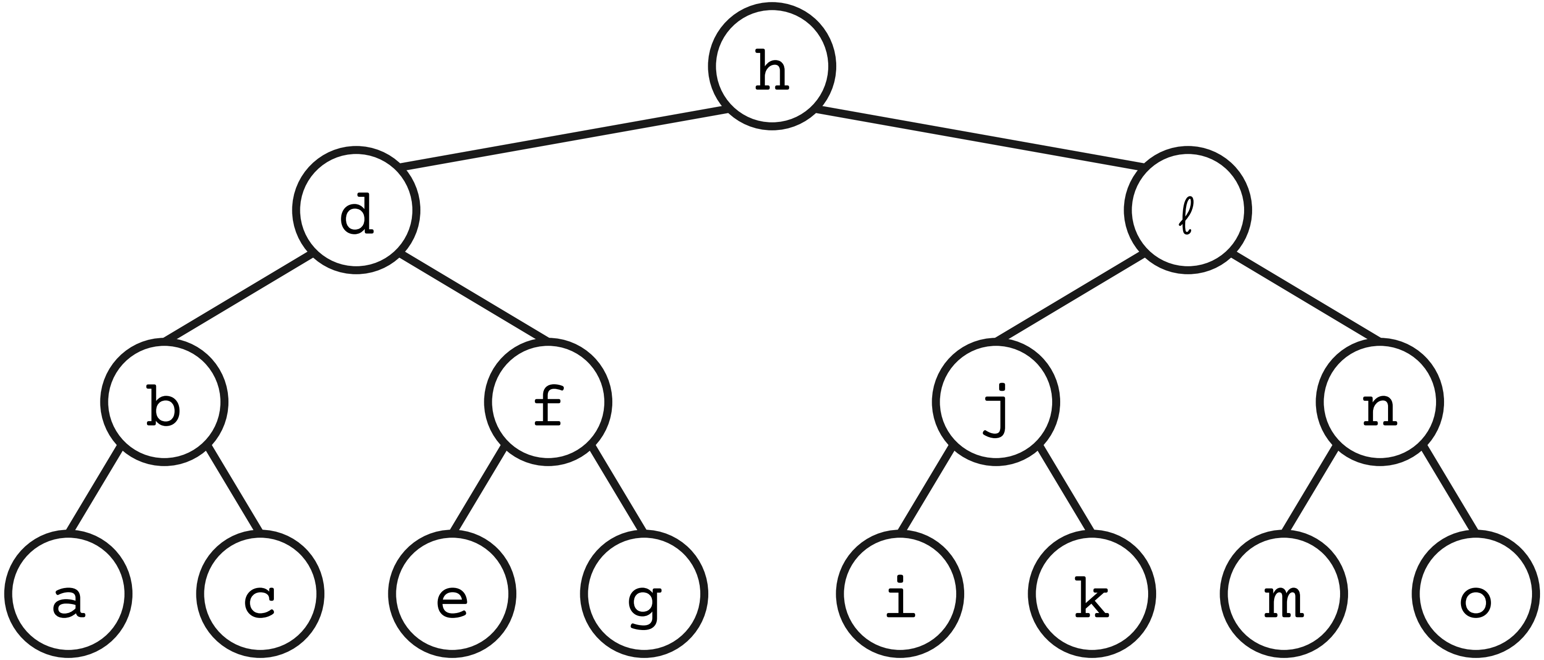
Determine the sequences independently for 1 minute.
Work on the additional points with a neighbor for 3 minutes.



A splay tree.

Additional points:

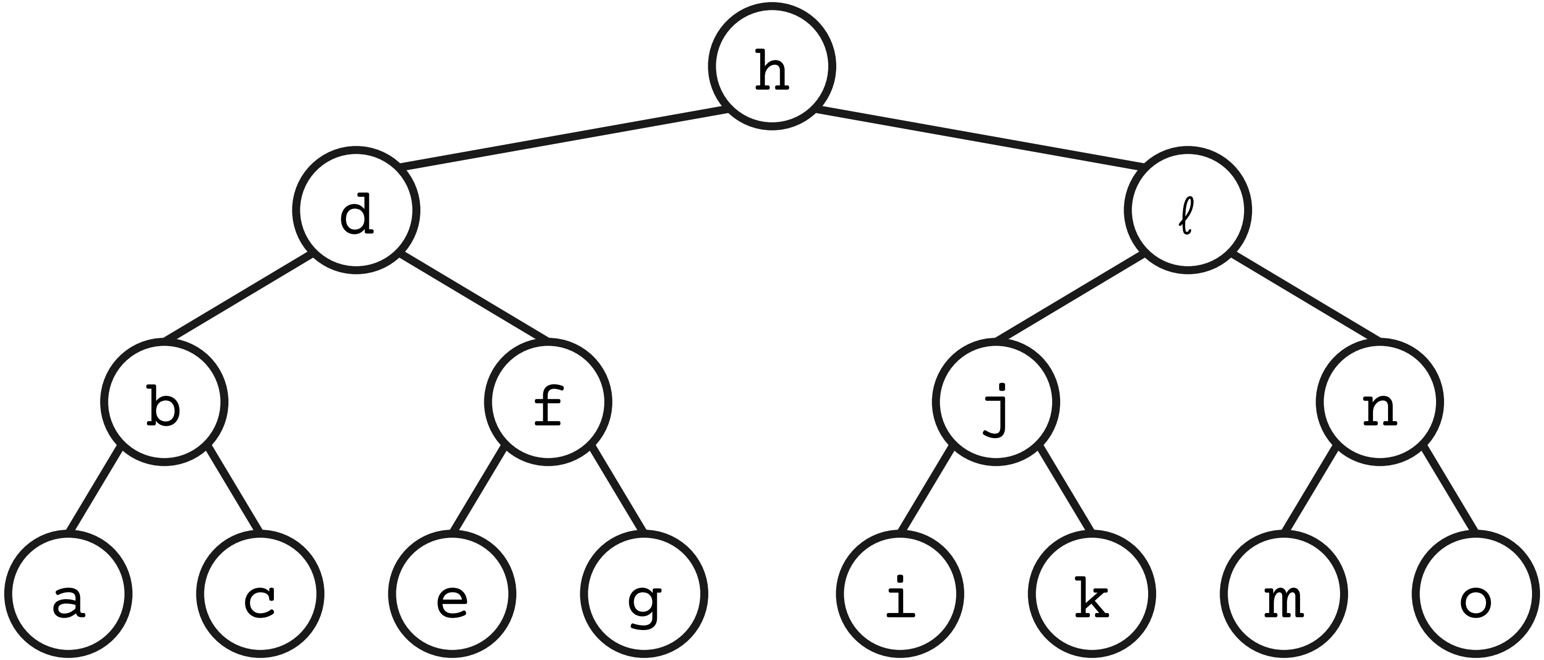
- Describe the first sequence in three ways: parent and direction; edge; node that moves up.
- Adjust the splay tree based on the first sequence. In other words, apply the first splay.



Splay `a`: Rotate `d` to the right, then rotate `b` to the right, then rotate `h` to the right.
Rotate edge `bd`, then edge `ab`, then edge `ah`.
Rotate `b` up, then `a` up, then `a` up.

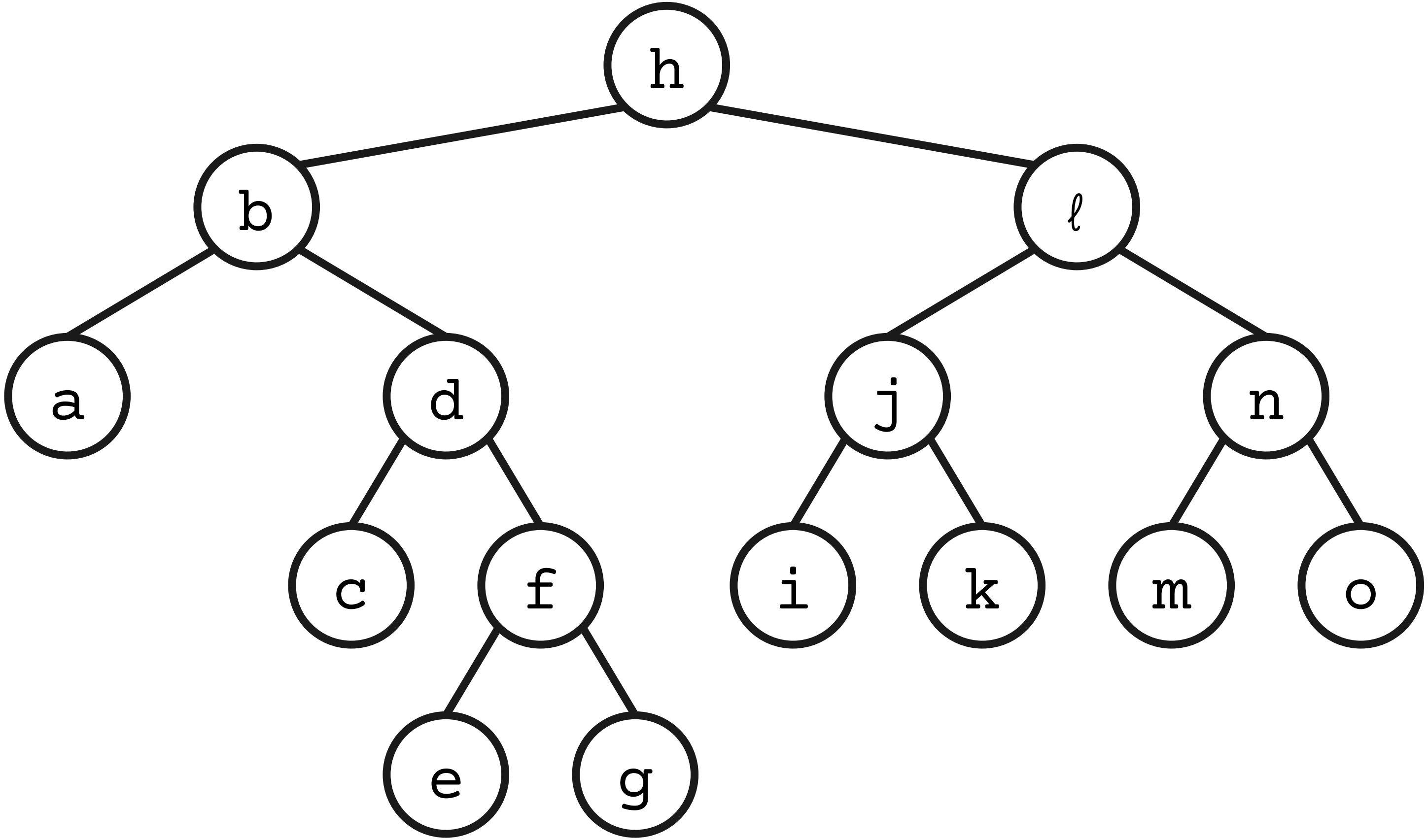
Splay `c`: Rotate `b` to the left, then `d` to the right, then `h` to the right.

Splay `n`: Rotate `h` to the left, then ℓ to the left.

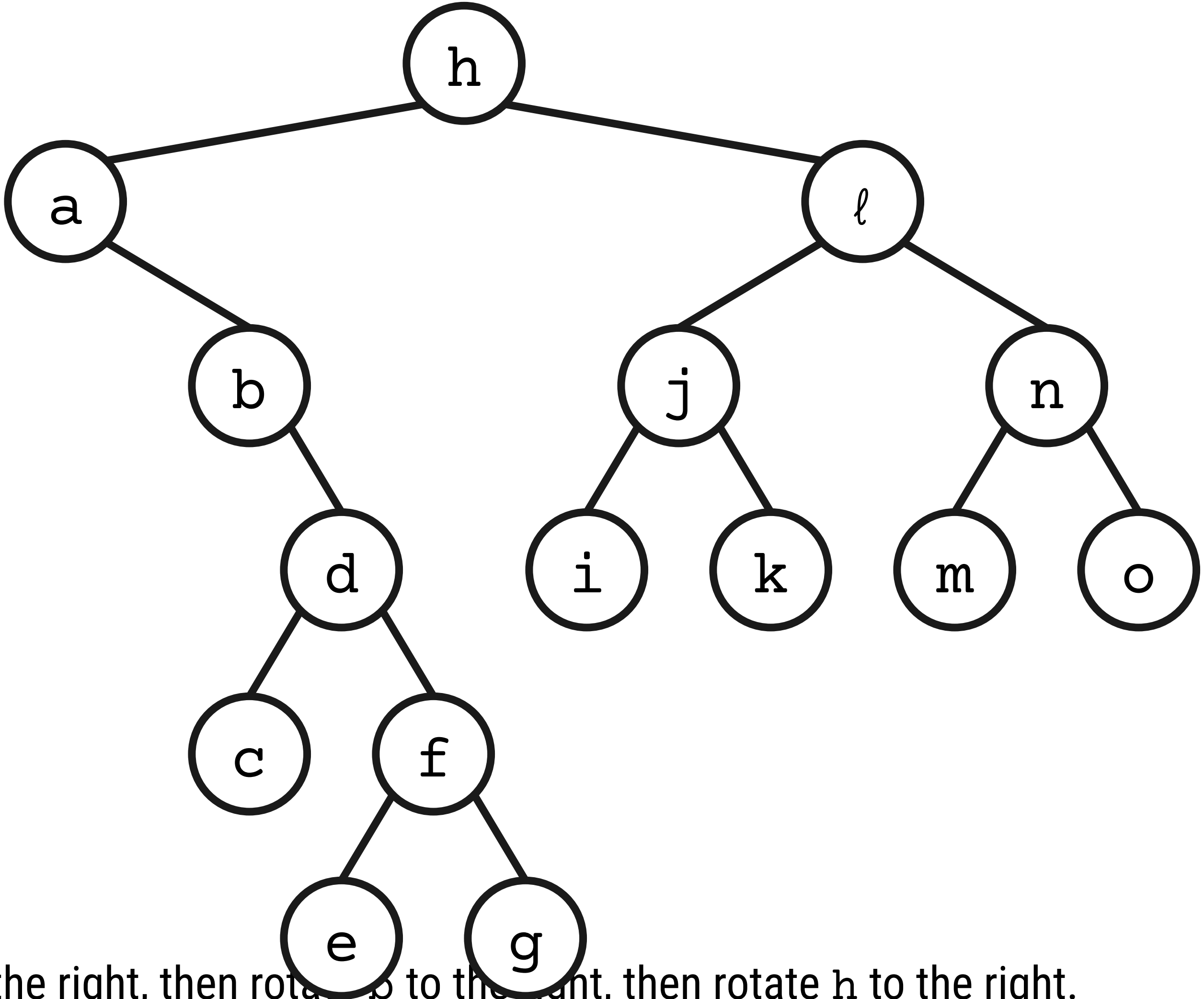


Now let's complete the three rotations that are involved in splaying node `a`.
Note that `a` is the smallest value in the tree, so this will be a worst-case example.

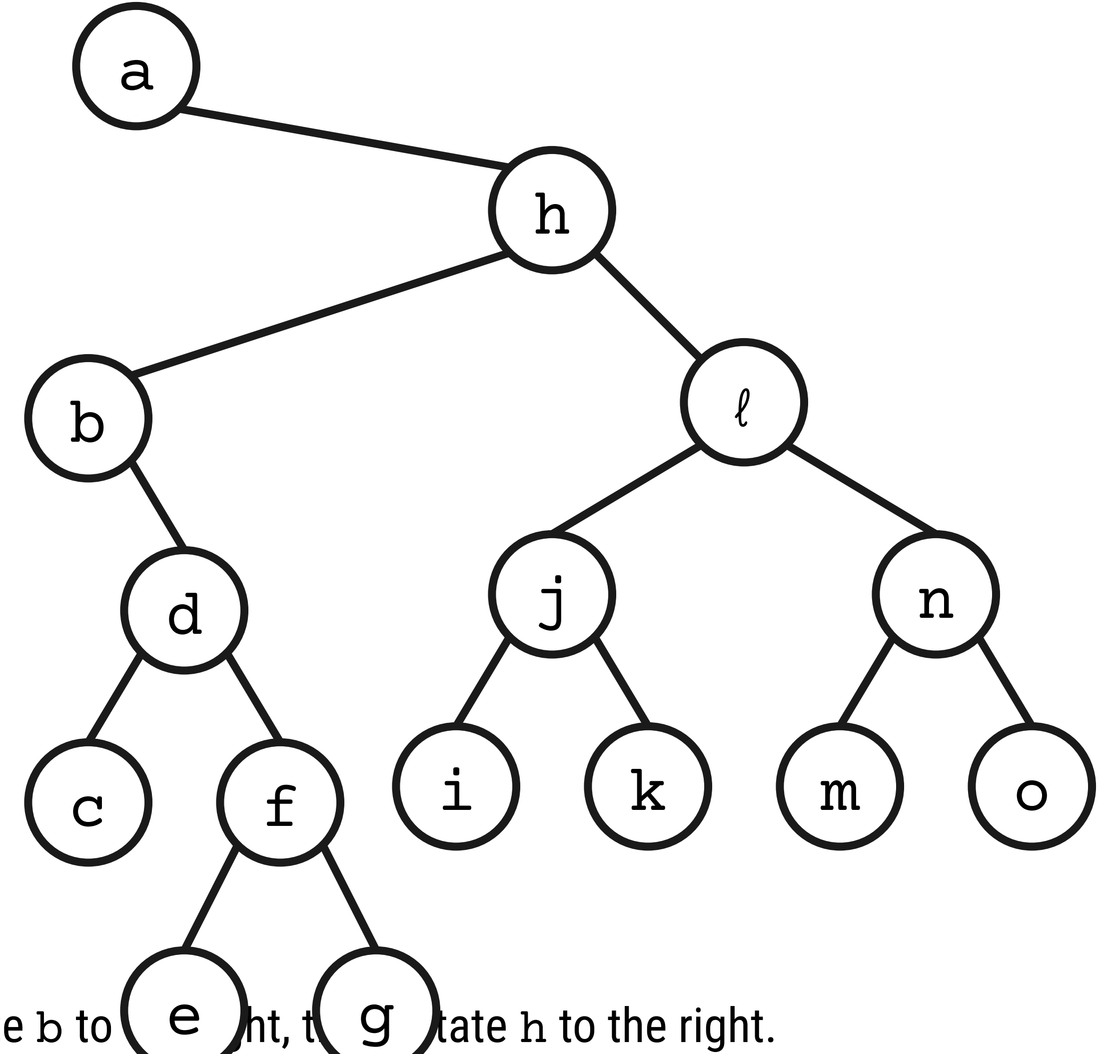
Splay `a`: Rotate `d` to the right, then rotate `b` to the right, then rotate `h` to the right.



Splay `a`: Rotate `d` to the right, then rotate `b` to the right, then rotate `h` to the right.



Splay `a`: Rotate `d` to the right, then rotate `b` to the right, then rotate `h` to the right.



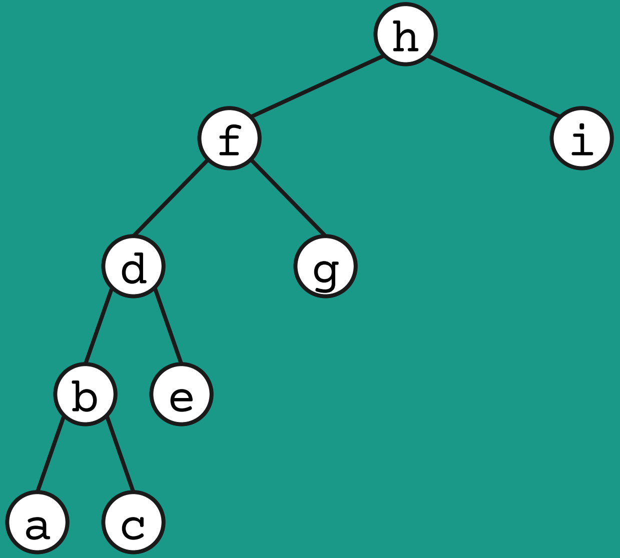
Splay `a`: Rotate `d` to the right, then rotate `b` to ... ht, t... tate `h` to the right.

What's Up: Zig-Zig Cases?

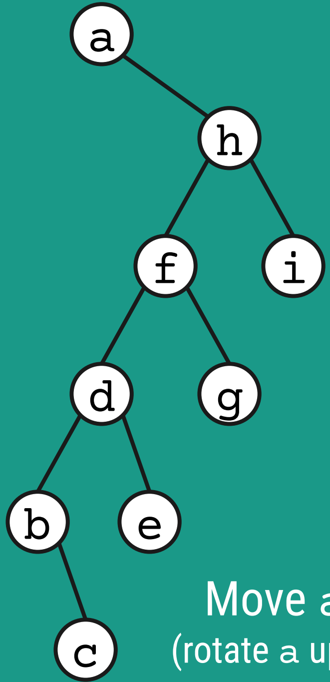
Question: Why does the splay operation rotate the grandparent then the parent in zig-zig cases?

Answer: It produces more balanced trees. (Intuitively, zig-zig cases are biased in one direction, while zig-zag cases are not.)

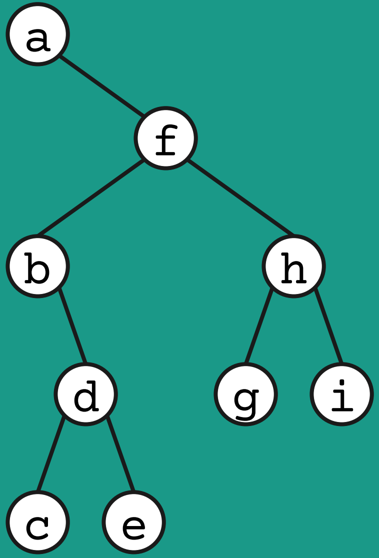
Below is a single zig-zig example to help illustrate this point. Think of the nodes `c`, `e`, `g`, `i` being subtrees.



Initial tree



Move `a` to root
(rotate `a` up four times)



Splay `a`

History

Brief History of Splay Trees

The splay tree was introduced in a paper

Self-Adjusting Binary Search Trees

The *splay operation* is a small adjustment to the *move-to-root* operation used in an earlier paper called

Self-Organizing Binary Search Trees

This adjustment allows the tree operations to run in amortized O(log n)-time instead of O(n)-time.

Research in Computer Science often advances through small insights like this.

Self-Adjusting Binary Search Trees

DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, NJ

Abstract. The *splay* tree, a self-adjusting form of binary search tree, is developed and analyzed. The binary search tree is a data structure for representing tables and lists so that accessing, inserting, and deleting items is easy. On an *n*-node splay tree, all the standard search tree operations have an amortized time bound of $O(\log n)$ per operation, where by "amortized time" is meant the time per operation averaged over a worst-case sequence of operations. Thus splay trees are as efficient as balanced trees when total running time is the measure of interest. In addition, for sufficiently long access sequences, splay trees are as efficient, to within a constant factor, as static optimum search trees. The efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called *splaying*, whenever the tree is accessed. Extensions of splaying give simplified forms of two other data structures: lexicographic or multidimensional search trees and link/cut trees.

Categories and Subject Descriptors: E.1 **[Data]**: Data Structures—*trees*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*sorting and searching*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Amortized complexity, balanced trees, multidimensional searching, network optimization, self-organizing data structures

Journal of the ACM (JACM), 1985

Self-Organizing Binary Search Trees

BRIAN ALLEN AND IAN MUNRO

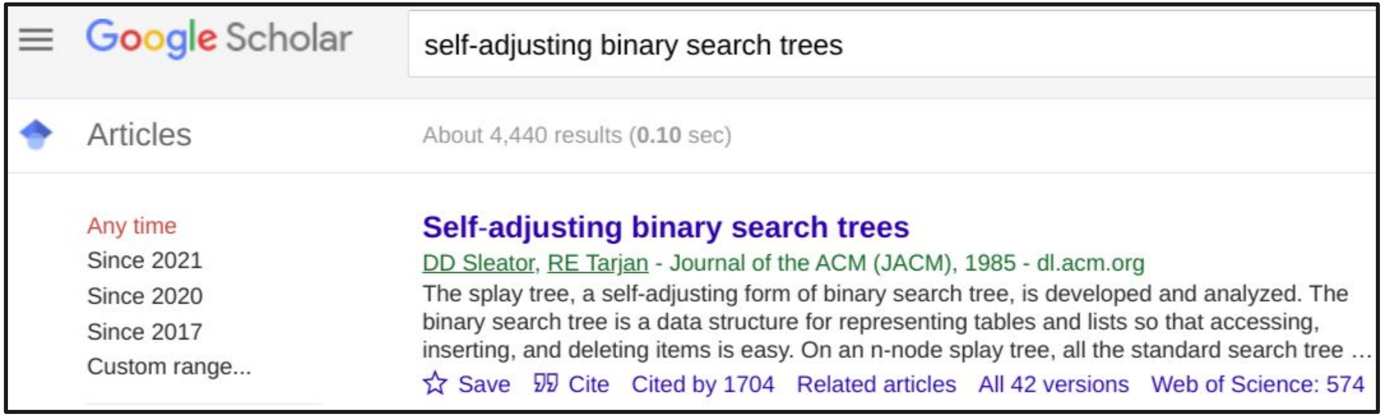
University of Waterloo, Waterloo, Ontario, Canada

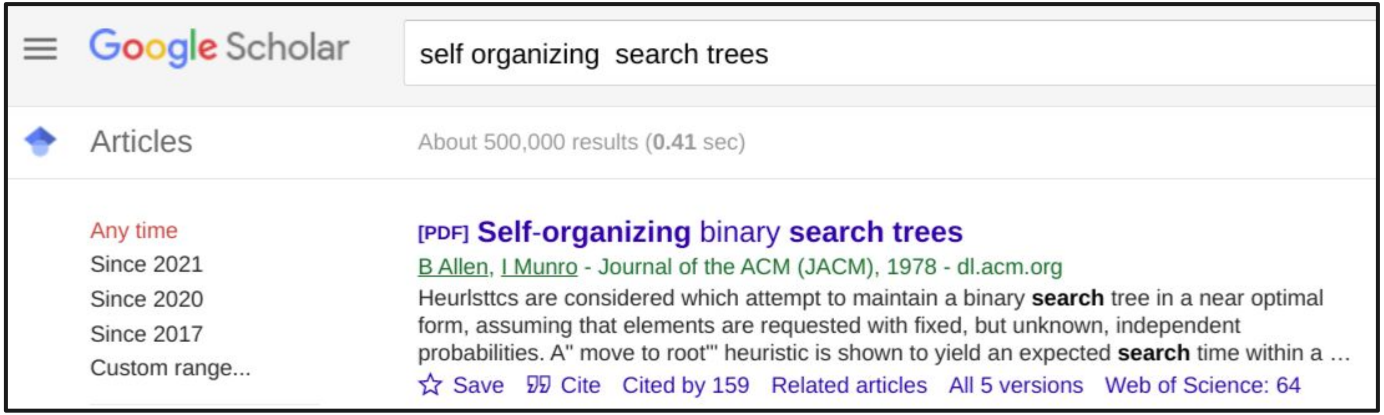
ABSTRACT Heuristics are considered which attempt to maintain a binary search tree in a near optimal form, assuming that elements are requested with fixed, but unknown, independent probabilities. A "move to root" heuristic is shown to yield an expected search time within a constant factor of that of an optimal static binary search tree. On the other hand, a closely related "simple exchange" technique is shown not to have this property. The rate of convergence of the move to root heuristic is discussed Also considered is the more general case in which elements not in the tree may have nonzero probability of being requested.

KEY WORDS AND PHRASES: binary search tree, adaptive structure, average behavior

CR CATEGORIES: 3.74, 5.25, 5.6

Journal of the ACM (JACM), 1978





Use Google Scholar (scholar.google.com) to view the articles that cite these article.

Iterator

Iteration Issues

In a binary search tree, the `find` operation is typically read-only in the sense that it doesn't change the underlying data. As a result, it is safe to run `find` operations while iterators are active.

This is no longer true with splay trees. However, it is possible to remedy this concern with some additional attention. The textbook's `structure` package creates a special class for this task in `SplayTreeIterator.java`.

Side note: Java's handling of iteration is strange for those more familiar with other languages. Upon reflection, one of Java's primary design goals appears to be the following:

- Container classes are *iterable* and can have multiple *iterators* acting on them simultaneously.

This helps motivate having both iterables and iterators. For example, consider a large source of data (the container classe) and multiple programs iterating over its contents.

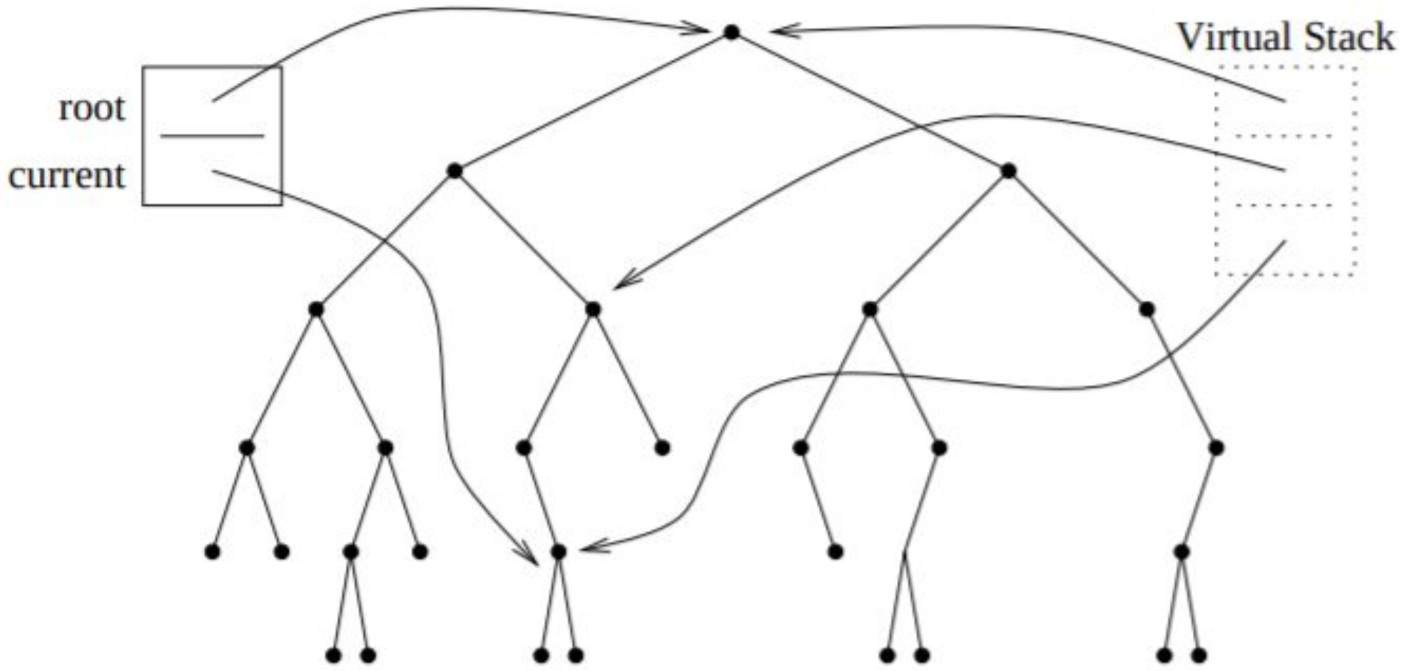


Figure 14.6 A splay tree iterator, the tree it references, and the contents of the virtual stack driving the iterator.

The textbook discusses how to build a safe iterator for splay trees.

Analysis

Analysis: Practical vs Theoretical

Splay trees perform very well when working with "normal" data.

A precise analysis is complicated, since we need to provide a model of temporal locality.

In fact, open problems still remain in terms of the analysis of this data structure.

On the other hand, splay trees tend not to perform well when the values are accessed randomly. Similarly, splay trees have better amortized or expected run-times than worst-case run-times.

Splay trees also have another practical benefit. They are able to "fix" degenerate trees that result from inserting values in increasing (or decreasing) order, which is a common case in practice.

On the other hand, splay trees have one disadvantage in practice.

- The run-times of each `find`, `insert`, `delete` is doubled due to the second traversal. (Each rotation takes constant time, but we perform one at each level going back up).

Summary

PROS

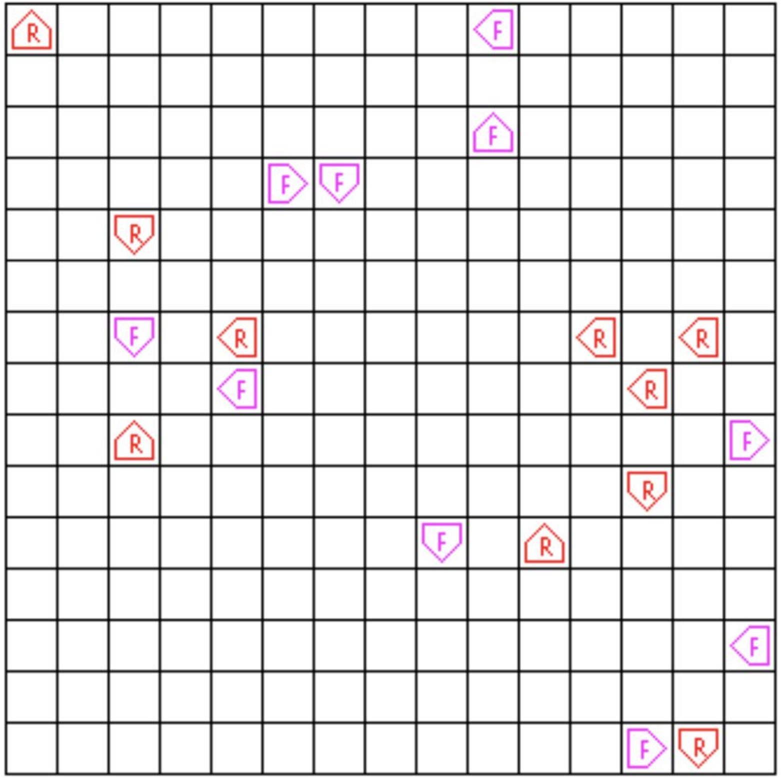
- Not too difficult to implement.
- Improves performance on the worst-case trees.
- Improves performance when the insertions are done in increasing or decreasing order, which is common in practice.
- Works well when the data is not random, especially if it has temporal locality.
- Interesting from educational perspective.
    - Illustrates tree operations.
    - Illustrates analysis issues.

CONS

- No guarantee of O(log n) worst-case performance.
- Makes every operation 2x slower.
- Difficult to analyze precisely.
- Doesn’t work well with random data and accesses.
- Removes the read-only property of `find`. This leads to more challenging iteration.

Lab 8 – Preview

(Part 1)



Step	Instruction	Comment
1	`ifenemy 4`	If there is an enemy ahead, go to step 4
2	`left`	Turn left
3	`go 1`	Go back to step 1
4	`infect`	Infect the adjacent creature
5	`go 1`	Go back to step 1

The Darwin lab.

- Game board is illustrated above with ***R**overs* and ***F**lytraps*.
- The (genetic) code for the Flytrap species is shown above.
- There will be a contest after Thanksgiving!