Chapter 9

Recursion With Data Structures

What is in This Chapter ?

In the last course, we discussed recursion at a simple level. This chapter explains how to do more complex ***recursion*** using various data structures. You should understand recursion more thoroughly after this chapter.

9.1 Recursive Efficiency

You should already be familiar with recursion at this point, having taken it last term in COMP1405. Although recursion is a powerful problem solving tool, it has some drawbacks. A non-recursive (or iterative) method may be **more** efficient than a recursive one for two reasons:

1. there is an overhead associated with large number of method calls
2. some algorithms are inherently inefficient.

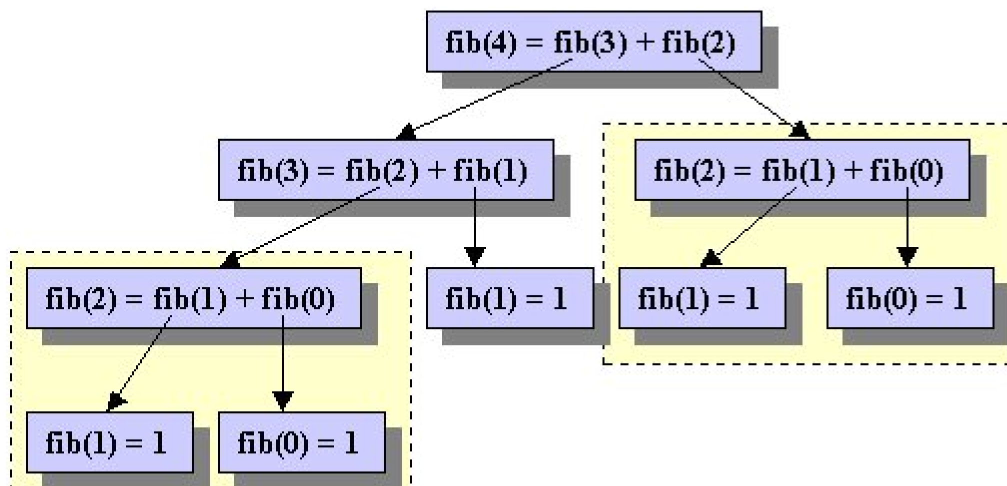
For example, computing the **n**th Fibonacci number can be written as:

1	if **n = 0**
fib(n) = 1	if **n = 1**
fib(n-1) + fib(n-2)	if **n > 1**

A straight-forward recursive solution to solving this problem would be as follows:

```
public static int fibonacci(int  n) {
      if (n <= 1)
            return 1;
      return fibonacci(n-1) + fibonacci(n-2);
}
```

However, notice what is happening here:



In the above computation, some problems (e.g., **fibonacci(2)**) are being solved more than once, even though we presumably know the answer after doing it the first time. This is an example where recursion can be inefficient if we do not do it carefully.

The following iterative solution avoids the re-computation:

```
public static int fibonacci2(int  n) {
      int first = 1;
      int second = 1;
      int third = 1;
      for (int i=2; i<=n; i++) {
            third = first + second; // compute new value
            first = second;         // shift the others to the right
            second = third;
      }
      return third;
}
```

Notice how the previous two results are always stored in the **first** and **second** variables. So the computation need not be duplicated. We can do this recursively ... we just need to keep track of previous computations.

```
public static int fibonacci(int  n, int prev, int prevPrev) {
      if (n <= 0)
            return prev + prevPrev;
      else
            return fibonacci(n-1, prev + prevPrev, prev);

}
```

We essentially use the same idea of keeping track of the last two computations and passing them along to the next recursive call. However, we would have to start this off with some values for these parameters. For example to find the 20th Fibonacci number, we could do this:

```
fibonacci(18, 1, 1);
```

The first two numbers are 1 and then there are 18 more to find.
However, this is not a nice solution because the user of the method must know what the proper values are in order to call this method. It would be wise to make this method **private** and then provide a **public** one that passes in the correct initial parameters as follows:

```
public static int fibonacci3(int  n) {
      if (n <= 1)
            return 1;
      return fibonacci(n-2, 1, 1);  // calls above method
}
```

This method in itself is not recursive, as it does not call itself. However, indirectly, it does call the 3-parameter **fibonacci()** method ... which is recursive. We call this kind of method indirectly recursive.

*An **indirectly recursive** function is one that does not call itself, but it does call a recursive method.*

Indirect recursion is mainly used to supply the initial parameters to a recursive function. It is the one that the user interacts with. It is often beneficial to use recursion to improve efficiency as well as to create non-destructive functions.

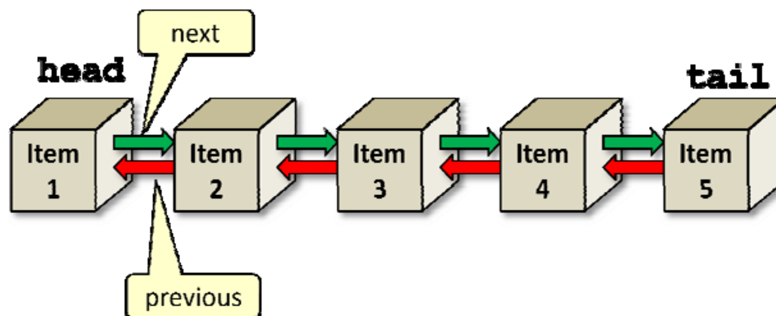
9.2 Examples With Self-Referencing Data Structures

Until now, the kinds of problems that you solved recursively likely did not involve the use of data structures. We will now look at using recursion to solve problems that make use of a couple of simple data structures.

First, recall the linked-list data structure that we created in the last chapter. It is a self-referencing data structure since each **Item** object points to two other **Item** objects:

```
public class LinkedList {
      Item  head;
      Item  tail;
      ...
}
```

```
public class Item {
      byte  data;
      Item  previous;
      Item  next;
      ...
}
```



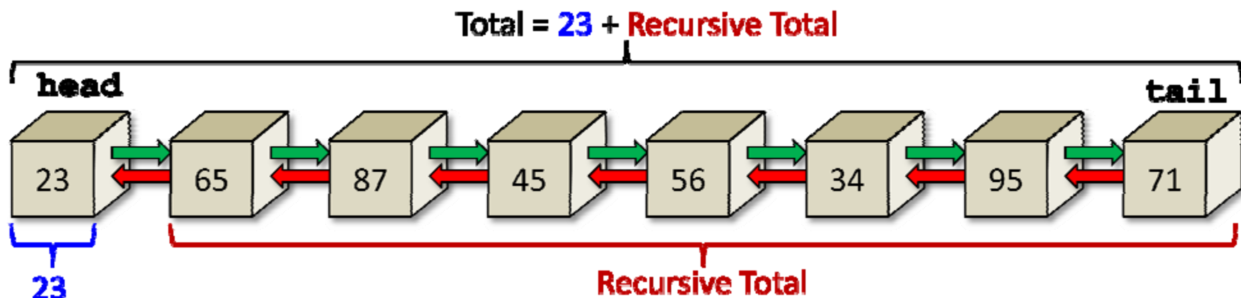
We can write some interesting recursive methods for this data structure.

Example:

Recall the following method:

```
public int totalData() {
    if (head == null)
        return 0;

    int     total = 0;
    Item    currentItem = head;
    while (currentItem != null) {
        total += currentItem.data;
        currentItem = currentItem.next;
    }
    return total;
}
```

Let us see how we can write this method recursively without using a **while** loop. As before, we need to consider the base case. If the **head** of the list is **null**, then the answer is **0** as in the above code. Otherwise, we will need to break the problem down recursively. To do this, we can simply break off the first data item from the list and add it to the recursive result of the remainder of the list ... this will be the solution:



However, we do not want to destroy the list, so we will simply "pretend" to break off a piece by traversing through the **next** pointers of the items, starting at the **head**. The solution is straightforward as long as we are allowed to pass in a parameter representing the item in the list from which to start counting from (e.g., the head to begin). Recall, that we always begin with a "base case" ... which is the stopping condition for the recursion. It always represents the simplest situation for the data structure. In this case, the simplest case is a **null** Item. If the item is null, there are no numbers to add, so the result is clearly 0. Otherwise, we just need to "tear off" the first number and continue with the remainder of the list (i.e., continue adding ... but starting with the **next** item in the list):

```
private int totalDataRecursive(Item startItem) {
    if (startItem == null)
        return 0;

    return startItem.data + totalDataRecursive(startItem.next);
}
```

Notice the simplicity of the recursion. It is quite straight forward and logical. As it turns out, writing recursive methods for most self-referencing data structures is quite natural and often produces simple/elegant code.

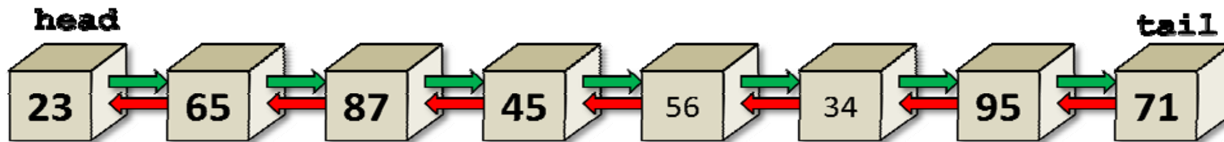
One downfall of the above method is that it requires a parameter which MUST be the **head** item of the list if it is to work properly! So then, we will want to make a **public** method that the user can call which will create the proper starting parameter (i.e., the **head**):

```
public int totalDataRecursive() {
    return totalDataRecursive(head);
}
```

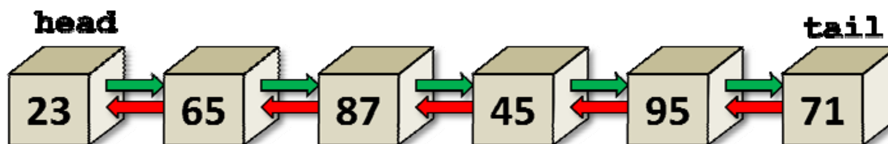
As you can see, the code is quite short. It simply supplies the list **head** to the single-parameter recursive method. This method is not itself recursive, but it does call the single-parameter method which IS recursive. We therefore call this an ***indirectly recursive*** method because although the method itself is not recursive, the solution that the method provides is recursive.

Example:

Now what about writing a recursive method that returns a new **LinkedList** that contains all the *odd* data from the list ?



The method should return a new **LinkedList**:



Hence, the return type for the method should be **LinkedList**.

```
public LinkedList oddItems() {
    // ...
}
```

The method code should begin with a "base case". What is the simplest list that we can have for use in the "base case" ? An empty one (i.e., headless), of course!

```
public LinkedList oddItems() {
    if (head == null)
        return new LinkedList();
    // ...
}
```

Otherwise, we will need to apply the same strategy of breaking off a piece of the problem. We can do this using indirect recursion again by starting with the head. It will be easiest, as well to have the new list passed in as a parameter that we can simply add to:

```
public LinkedList oddItems() {
    if (head == null)
        return new LinkedList();

    return oddItems(head, new LinkedList());
}
```

So then, the directly-recursive method will be defined as follows:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    // ...
}
```

Notice that it is private, because it is just a kind of *helper* method for the public one.

Since the **startItem** will eventually become **null**, we will need to check for that as our stopping condition (i.e., "base case"). In that case, we are done ... and we just need to return the **resultList**:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;
    // ...
}
```

As a side point, since we are checking for **null** here in this method as well, we can go back and simplify our indirectly-recursive method by removing that check:

```
public LinkedList oddItems() {
    return oddItems(head, new LinkedList());
}
```

Now, we need to check to see if the data of the **startItem** is indeed odd, and if so... then add it to the **resultList**:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;

    if (startItem.data %2 != 0)
        resultList.add(new Item(startItem.data));
    // ...
}
```

Notice that we used a constructor to create a new **Item** object before adding to the resulting list. What would have happened if we simply used `resultList.add(startItem)`? If we did this, then the same **Item** object would exist in both lists. This is bad because if we changed the **next** or **previous** pointer for this item, then it would affect both lists... and that would be disastrous.

Lastly, we continue by checking the remainder of the list:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;

    if (startItem.data %2 != 0)
        resultList.add(new Item(startItem.data));

    return oddItems(startItem.next, resultList);
}
```

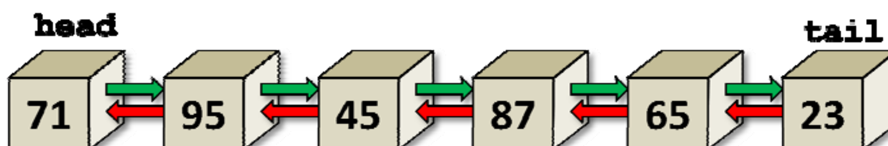
Do you understand why the **return** keyword is needed on the last line ? What gets returned ? Well ... remember that the method MUST return a **LinkedList**. If we leave off the **return** keyword, the compiler will complain because we are not returning any specific list. At the end of the recursive method calls, the "base case" will ensure that we return the **resultList** that we have built up with the odd numbers. Alternatively we could have just finished off the recursion and then specifically state that we want to return the **resultList**:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;

    if (startItem.data %2 != 0)
        resultList.add(new Item(startItem.data));

    oddItems(startItem.next, resultList);

    return resultList;
}
```

We can actually make this all work without that extra **resultList** parameter by creating the list when we reach the end of the list, and then adding items AFTER the recursion:

```
public LinkedList oddItems() {
    return oddItems(head);
}

private LinkedList oddItems(Item startItem) {
    if (startItem == null)
        return new LinkedList();

    LinkedList result = oddItems(startItem.next);

    if (startItem.data %2 != 0)
        result.add(new Item(startItem.data));

    return result;
}
```

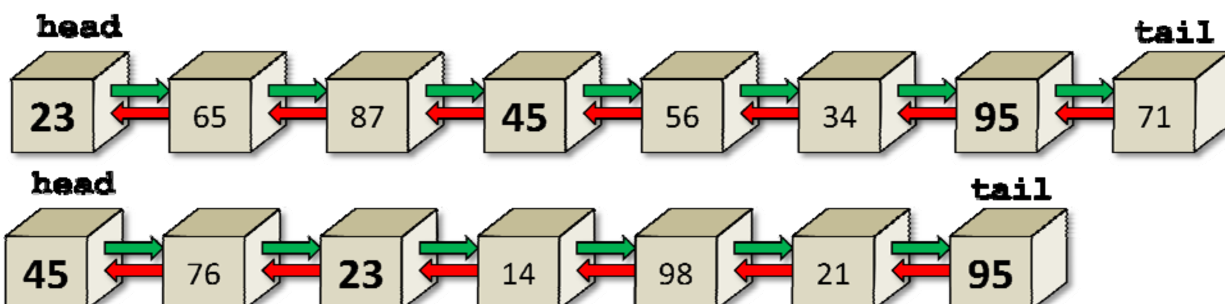
However, this solution will return the odd numbers in reverse order.



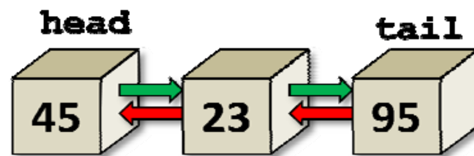
See if you can fix that problem. It can be done.

Example:

Now, let us find and return a list of all elements in common between 2 lists:



Again, the result will be a new **LinkedList**:



To do this, we will need a similar indirectly-recursive method, but one that takes the 2nd list as a parameter:

```
public LinkedList inCommon(LinkedList aList) {
    return inCommon(this.head, aList.head, new LinkedList());
}
```

Here, we see that the two list **heads** are passed in as well as a list onto which the common elements may be added. The directly-recursive method begins the same way as before ... quitting when we reach the end of one of the lists:

```
private LinkedList inCommon(Item start1, Item start2, LinkedList result) {
    if ((start1== null) || (start2 == null))
        return result;
    // ...
}
```

The remaining problem is a bit trickier. We need to check each item in one list with each item in the other list ... but just once. So we will need to recursively shrink one list until it has been checked against one particular item in the second list. Then, we need to move to the next item in the second list and check it against the entire first list again. It will be easier to do this if we had a *helper* method that simply checked whether or not a data item is in another list.

Consider a method called **contains()** which takes a starting item in the list and determines whether or not a specific data item is in the list by recursively iterating through the list. Can you write this ? It should be straight forward now:

```
public boolean contains(Item startItem, byte data) {
    if (startItem == null)
        return false;

    if (startItem.data == data)
        return true;

    return contains(startItem.next, data);
}
```

Now, how can we make use of this **contains()** method to solve our original problem ? Well, we can call it like any other function. We can simply iterate through the items of one list (as before) and check each item against the other list using this **contains()** function. If it IS contained, we add it to the solution. It is quite similar to the template for finding the odd numbers now:

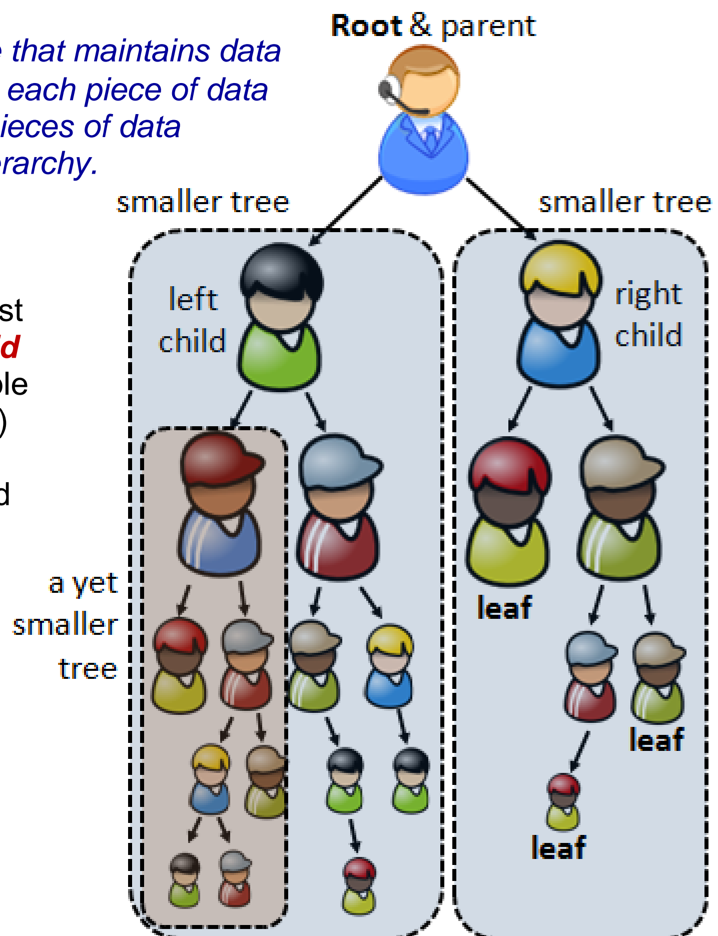
```
private LinkedList inCommon(Item list1, Item list2, LinkedList result) {
    if ((list1 == null) || (list2 == null))
        return result;

    // Check if the first list contains the data
    // at the beginning item of the 2nd list.
    if (contains(list1, list2.data))
        result.add(new Item(list2.data));

    // Now check the first list with the remainder of the 2nd list
    return inCommon(list1, list2.next, result);
}
```

As you can see, the call to **contains()** here will check all items of list 1 with the first item of list 2. Then, the recursive call will move on to the next item in list 2. Eventually, list 2 will be exhausted and we will have the solution!

Example:

Now let us try a different data structure. In computer science, we often store information in a ***binary tree***:

> *A **binary tree** is a data structure that maintains data in a hierarchical arrangement where each piece of data (called the parent) has exactly two pieces of data (called children) beneath it in the hierarchy.*



The binary tree is similar to the notion of a single gender (i.e., all males or all females) family tree in which every parent has at most 2 children, which are known as the ***leftChild*** and ***rightChild*** of their ***parent***. It is possible that a ***node*** in the tree (i.e., a piece of data) may have no children, or perhaps only one child. In this case, we say that the leftChild and/or rightChild is ***null*** (meaning that it is non-existent).

A node with no children is called a ***leaf*** of the tree.

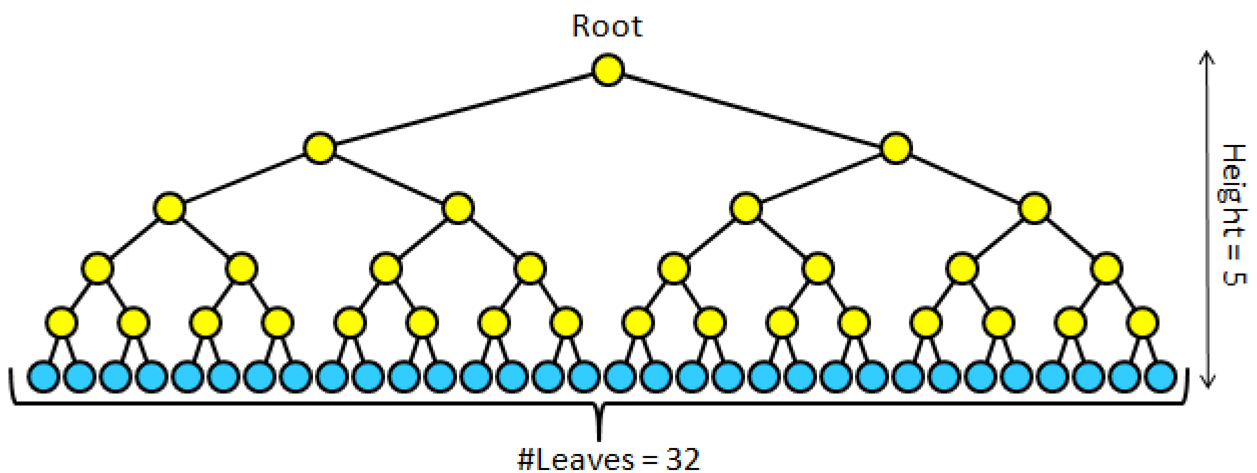
The ***root*** of the tree is the data that is at the top of the tree which has no parent.

Binary trees can be a very efficient data structure when searching for information. For example as we search for a particular item down the tree from the root, each time that we choose the left or right child to branch down to, we are potentially eliminating half of the remaining data that we need to search through.

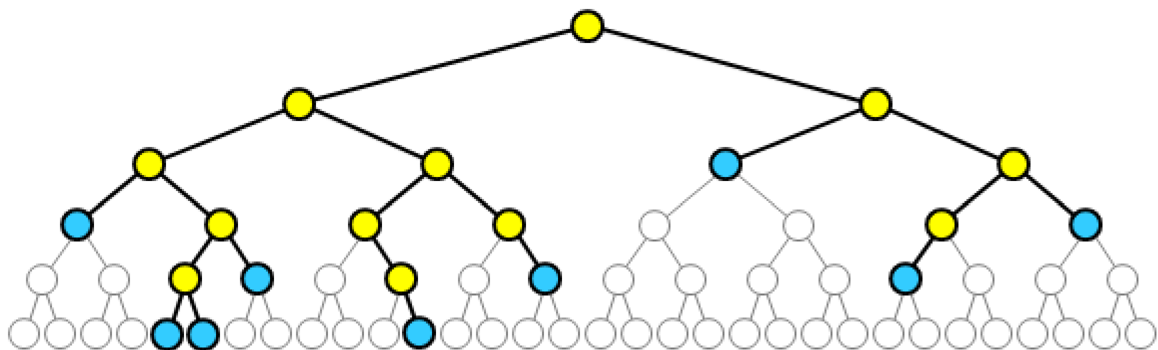
Typically, binary trees are represented as recursive data structures. That is, the tree itself is actually made up of other smaller trees. We can see this from the figure on the previous page, where each non-**null** child actually represents the root of a smaller tree.

In computer science, trees are often drawn with simple circles as nodes and lines as edges.

The ***height*** of a tree is the depth of the tree from root to the leaves. Here is an example of a ***complete*** tree (i.e., one that is completely filled with nodes) of height 5. A complete binary tree has $\mathbf{2^h}$ leaves (where **h** is the tree's height).



In our family tree picture (shown earlier), however, the tree was not complete ... there were nodes missing. Here is how we would draw the equivalent tree for that example:



Notice that it is basically the $\mathbf{2^5}$ binary tree with many of the nodes removed. Notice as well, that the leaves are not only at the bottom level of the tree but may appear at any level because any node that has no children is considered a leaf.

How can we represent a recursive data structure like this binary tree ? Well, remember, each time we break a branch off of a tree, we are left with two smaller trees. So a binary tree itself is made up of two smaller binary trees. Also, since trees are used to store data, each node of the tree should store some kind of information. Therefore, we can create a **BinaryTree** data structure as follows:

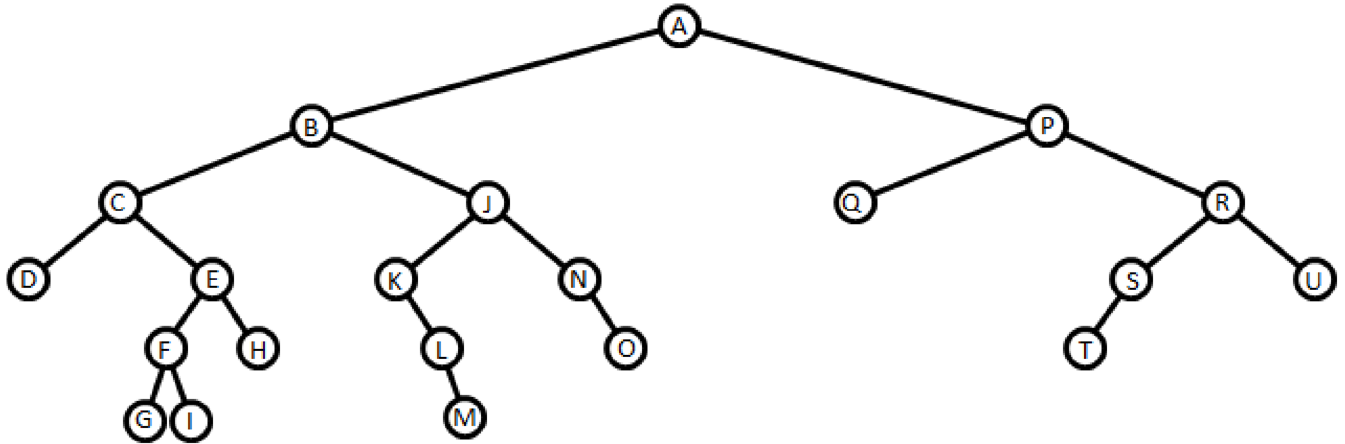
```
public class BinaryTree {
      private String          data;
      private BinaryTree      leftChild;
      private BinaryTree      rightChild;
}
```

Here, **data** represents the information being stored at that node in the tree ... it could be a String, a number, a Point, or any data structure (i.e., Object) with a bunch of information stored in it. Notice that the left **leftChild** and **rightChild** are actually binary trees themselves! A tree is therefore considered to be a ***self-referential*** (i.e., refers to itself) data structure and is thus a naturally recursive data structure.

Likely, we will also create some constructors as well as some get/set methods in the class:

```
public class BinaryTree {
      private String          data;
      private BinaryTree      leftChild;
      private BinaryTree      rightChild;

      // A constructor that takes root data only and
      // makes a tree with no children (i.e., a leaf)
      public BinaryTree(String d) {
            data = d;
            leftChild = null;
            rightChild = null;
      }

      // A constructor that takes root data as well as two subtrees
      // which then become children to this new larger tree.
      public BinaryTree(String d, BinaryTree left, BinaryTree right) {
            data = d;
            leftChild = left;
            rightChild = right;
      }

      // Get methods
      public String getData() { return data; }
      public BinaryTree getLeftChild() { return leftChild; }
      public BinaryTree getRightChild() { return rightChild; }

      // Set methods
      public void setData(String d) { data = d; }
      public void setLeftChild(BinaryTree left) { leftChild = left; }
      public void setRightChild(BinaryTree right) { rightChild = right; }
}
```

To create an instance of **BinaryTree**, we simply call the constructors. Consider a tree with the data for each node being a simple string with a letter character as follows:

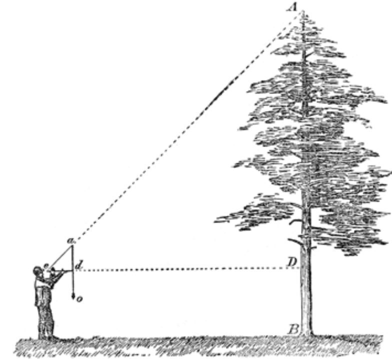


Here is a test program that creates this tree:

```
public class BinaryTreeTest {
      public static void main(String[] args) {
            BinaryTree  root;

            root = new BinaryTree("A",
                        new BinaryTree("B",
                              new BinaryTree("C",
                                    new BinaryTree("D"),
                                    new BinaryTree("E",
                                          new BinaryTree("F",
                                                new BinaryTree("G"),
                                                new BinaryTree("I")),
                                          new BinaryTree("H"))),
                              new BinaryTree("J",
                                    new BinaryTree("K",
                                          null,
                                          new BinaryTree("L",
                                                null,
                                                new BinaryTree("M"))),
                                    new BinaryTree("N",
                                          null,
                                          new BinaryTree("O")))),
                        new BinaryTree("P",
                              new BinaryTree("Q"),
                              new BinaryTree("R",
                                    new BinaryTree("S",
                                          new BinaryTree("T"),
                                          null),
                                    new BinaryTree("U"))));
      }
}
```

Example:

When we have such recursive data structures, it is VERY natural to develop recursive functions and procedures that work with them. For example, consider finding the height of this tree. It is not natural to use a **FOR** loop because we have no array or list to loop through.



How can we write a ***recursive*** function that determines the height of a binary tree ?

We need to determine the base case(s). What is the simplest tree ? It is one where the children are **null** (i.e., just a root). In this case, the height is 0.

Here is the code so far as an instance method in the **BinaryTree** class:

```
public int height()  {
      if ((leftChild == null) && (rightChild == null))
            return 0;
}
```

That was easy. Now, for the recursive step, we need to express the height of the tree in terms of the smaller trees (i.e., its children). So, if we knew the height of the **leftChild** and the height of the **rightChild**, how can we determine the height of the "whole" tree ?

Well, the height of the tree is one more than the trees beneath it. Assuming that the left and right sub-trees are equal in height, the recursive definition would be:

h(tree) = 1 + **h**(tree.leftChild)

However, as you can see from our family tree example, it is possible that the left and right children will have different heights (i.e., 4 and 3 respectively). So, to find the height of the whole tree, we need to take the largest of these sub-trees. So here is our recursive definition:

h(tree) = 1 + **maximum**(**h**(tree.leftChild), **h**(tree.rightChild))

Here is the code:

```
public int height()  {
    if ((leftChild == null) && (rightChild == null))
        return 0;
    return 1 + Math.max(leftChild.height(),
                        rightChild.height());
}
```

However, there is a slight problem. If one of the children is **null**, but not the other, then the code will likely try to find the **leftChild** or **rightChild** of a **null** tree ... and this will generate a **NullPointerException** in our program. We can fix this in one of two ways:

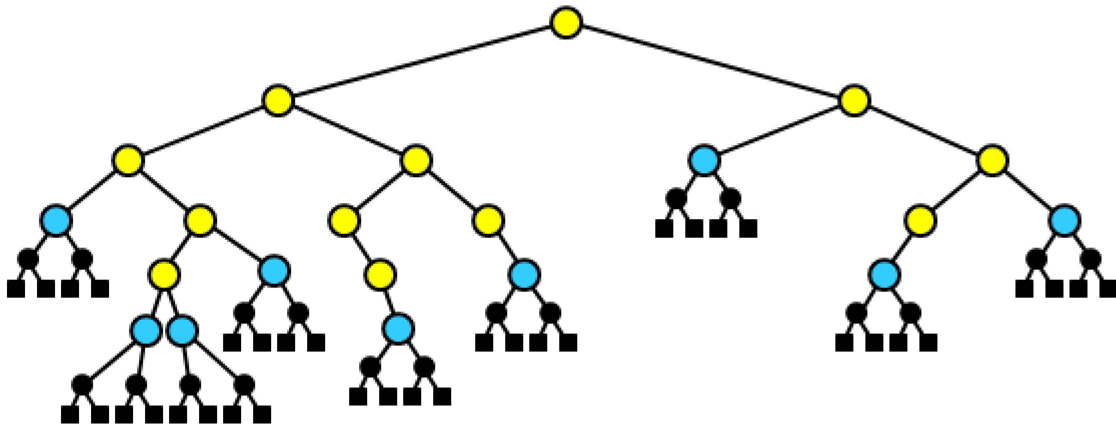
(1) check for the case where one child is **null** but not the other
(2) handle **null** trees as a base case.

Here is the solution for (1):

```
public int height()  {
    if (leftChild == null) {
        if (rightChild == null)
              return 0;
        else
              return 1 + rightChild.height();
    }
    else {
        if (rightChild == null)
              return 1 + leftChild.height();
         else
            return 1 + Math.max(leftChild.height(),
                                rightChild.height());
    }
}
```

The above code either checks down one side of the tree or the other when it encounters a tree with only one child. If there are no children, it returns **0**, and otherwise it takes the maximum of the two sub-trees as before.

In choice (2) for dealing with **null** children, it is simpler just to add a base-case for handling **null** tree roots. However this requires the addition of extra nodes. That is, instead of having a child set to **null**, we can have a special tree node that represents a **dummy** tree and simply have all leaves point to that special tree node. In a sense, then, these dummy tree nodes become the leaves of the tree:



In the above picture, the black circles are **BinaryTree** objects and the black boxes indicate that the values of the left and right children are **null**. So, the example above adds 18 dummy nodes to the tree. The dummy nodes are known as ...

A ***Sentinel Node (or Sentinel)*** *is a node that represents a path terminator. It is a specifically designated node that is not a data node of the data structure.*

Sentinels are used as an alternative over using **null** as the path terminator in order to get one or more of the following benefits: (1) Increased speed of operations; (2) Reduced algorithmic code size; (3) Increased data structure robustness (arguably).

How do we make a sentinel ? It is simply a regular **BinaryTree** but has its data value (and children) set to **null** as follows:

```
new BinaryTree(null, null, null)
```

If we decide to use sentinel tree nodes, then we need to add a constructor, perhaps a default constructor and then make changes to the other constructors as necessary to use sentinel nodes instead of **null**.

```
public class BinaryTree2 {
      private String          data;
      private BinaryTree2     leftChild;
      private BinaryTree2     rightChild;

      // A constructor that makes a Sentinel node
      public BinaryTree2() {
            data = null;
            leftChild = null;
            rightChild = null;
      }

      // This constructor now uses sentinels for terminators instead of null
      public BinaryTree2(String d) {
            data = d;
            leftChild = new BinaryTree2();
            rightChild = new BinaryTree2();
      }

      // This constructor is unchanged
      public BinaryTree2(String d, BinaryTree2 left, BinaryTree2 right) {
            data = d;
            leftChild = left;
            rightChild = right;
      }

      // Get methods
      public String getData() { return data; }
      public BinaryTree2 getLeftChild() { return leftChild; }
      public BinaryTree2 getRightChild() { return rightChild; }

      // Set methods
      public void setData(String d) { data = d; }
      public void setLeftChild(BinaryTree2 left) { leftChild = left; }
      public void setRightChild(BinaryTree2 right) { rightChild = right; }
}
```

Now, we can re-write the test case to replace **null** with the new Sentinel nodes:

```
public class BinaryTreeTest2 {
      public static void main(String[] args) {
            BinaryTree2       root;

            root = new BinaryTree2("A",
                        new BinaryTree2("B",
                              new BinaryTree2("C",
                                    new BinaryTree2("D"),
                                    new BinaryTree2("E",
                                          new BinaryTree2("F",
                                                new BinaryTree2("G"),
                                                new BinaryTree2("I")),
                                          new BinaryTree2("H"))),
                              new BinaryTree2("J",
                                    new BinaryTree2("K",
                                          new BinaryTree2(),
                                          new BinaryTree2("L",
                                                new BinaryTree2(),
                                                new BinaryTree2("M"))),
                                    new BinaryTree2("N",
                                          new BinaryTree2(),
                                          new BinaryTree2("O")))),
                        new BinaryTree2("P",
                              new BinaryTree2("Q"),
                              new BinaryTree2("R",
                                    new BinaryTree2("S",
                                          new BinaryTree2("T"),
                                          new BinaryTree2()),
                                    new BinaryTree2("U"))));
      }
}
```

Now we will see the advantage of doing all this. We can re-write the **height()** method so that it does not need to check whether or not the children are **null**, but simply needs to stop the recursion if a sentinel node has been reached:

```
public int height()  {
    // Check if this is a sentinel node
    if (data == null)
        return -1;

    return 1 + Math.max(leftChild.height(),
                        rightChild.height());
}
```

Notice that since the sentinel nodes have added an extra level to the tree, when we reach a sentinel node, we can indicate a **-1** value as the height so that the path from the leaf to the sentinel does not get counted (i.e., it is essentially subtracted afterwards). The code is MUCH shorter and simpler. This is the advantage of using sentinels ... we do not have to keep checking for **null** values in our code.

Example:

How could we write code that gathers the leaves of a tree and returns them? Again, we will use recursion. In our example, we had 9 leaves. A leaf is identified as having no children, so whenever we find such a node we simply need to add it to a collection.

We can use an **ArrayList<String>** to store the node data. The base case is simple. If the **BinaryTree** is a leaf, return an **ArrayList** with the single piece of data in it:

```
public ArrayList<String> leafData()  {
      ArrayList<String> result = new ArrayList<String>();

      if (leftChild == null) {
            if (rightChild == null)
                  result.add(data);
      }

      return result;
}
```

Now what about the recursive part ? Well, we would have to check both sides of the root as we did before, provided that they are not **null**. Each time, we take the resulting collection of data and merge it with the result that we have so far. The merging can be done using the **list1.addAll(list2)** method in the **ArrayList** class. This method adds all the elements from **list2** to **list1**.

Here is the code:

```
public ArrayList<String> leafData()  {
   ArrayList<String>    result = new ArrayList<String>();

   if (leftChild == null) {
       if (rightChild == null)
          result.add(data);
       else
          result.addAll(rightChild.leafData());
   }
   else {
       result.addAll(leftChild.leafData());
       if (rightChild != null)
          result.addAll(rightChild.leafData());
      }
   return result;
}
```

What would this code look like with the Sentinel version of our tree ? Simpler ...

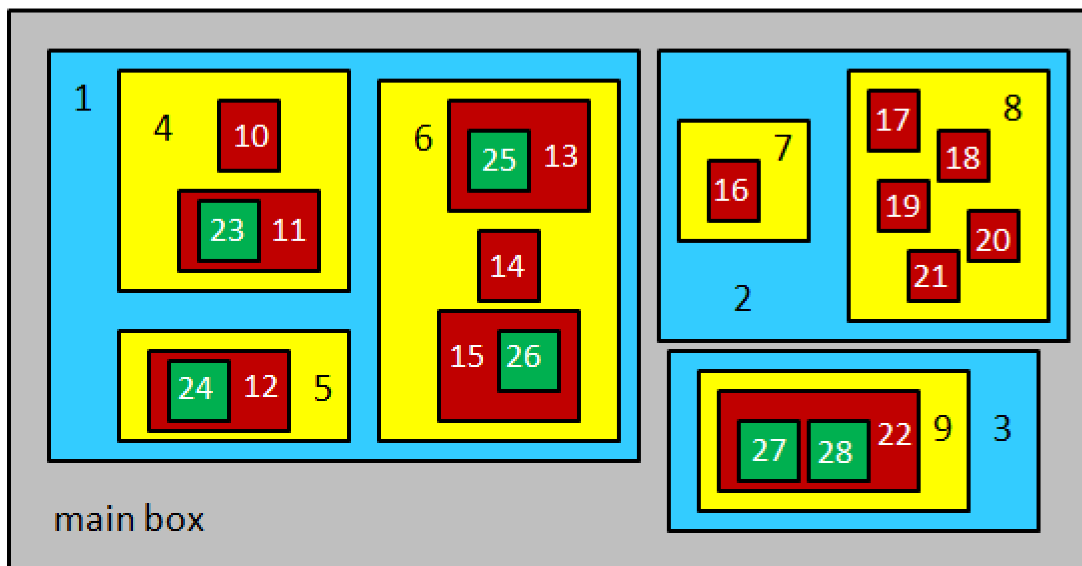
```
public ArrayList<String> leafData()  {
   ArrayList<String>    result = new ArrayList<String>();

   if (data != null) {
       if ((leftChild.data == null) && (rightChild.data == null))
          result.add(data);
       result.addAll(leftChild.leafData());
       result.addAll(rightChild.leafData());
   }
   return result;
}
```

There are many other interesting methods you could write for trees. You will learn more about this next year.

Example:

As another example, consider the following scenario. You wrap up your friends gift in a box ... but to be funny, you decide to wrap that box in a box and that one in yet another box. Also, to fool him/her you throw additional wrapped boxes inside the main box.

This boxes-within-boxes scenario is recursive. So, we have boxes that are completely contained within other boxes and we would like to count how many boxes are completely contained within any given box. Here is an example where the outer (gray) box has 28 internal boxes:

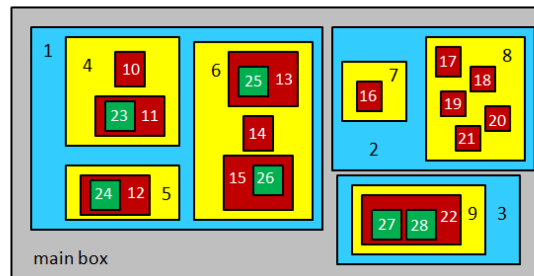


Assume that each box stores an **ArrayList** of the boxes inside of it. We would define a box then as follows:

```
import java.util.ArrayList;

public class Box {
      private ArrayList<Box>  internalBoxes;

      // A constructor that makes a box with no boxes in it
      public Box() {
            internalBoxes = new ArrayList<Box>();
      }

      // Get method
      public ArrayList<Box> getInternalBoxes() { return internalBoxes; }

      // Method to add a box to the internal boxes
      public void addBox(Box b) {
            internalBoxes.add(b);
      }
      // Method to remove a box from the internal boxes
      public void removeBox(Box b) {
            internalBoxes.remove(b);
      }
}
```

We could create a box with the internal boxes as shown in our picture above as follows:

```
public class BoxTest {
      public static void main(String[] args) {
            Box   mainBox, a, b, c, d, e, f, g, h, i, j;

            mainBox = new Box();

            // Create the left blue box and its contents
            a = new Box();             // box 10
            b = new Box();             // box 11
            b.addBox(new Box());       // box 23
            c = new Box();             // box 4
            c.addBox(a);
            c.addBox(b);
            d = new Box();             // box 12
            d.addBox(new Box());       // box 24
            e = new Box();             // box 5
            e.addBox(d);
            f = new Box();             // box 13
            f.addBox(new Box());       // box 25
            g = new Box();             // box 14
            h = new Box();             // box 15
            h.addBox(new Box());       // box 26
            i = new Box();             // box 6
            i.addBox(f);
            i.addBox(g);
            i.addBox(h);
            j = new Box();             // box 1
            j.addBox(c);
            j.addBox(e);
            j.addBox(i);
            mainBox.addBox(j);
```



```
            // Create the top right blue box and its contents
            a = new Box();              // box 7
            a.addBox(new Box());        // box 16
            b = new Box();              // box 8
            b.addBox(new Box());        // box 17
            b.addBox(new Box());        // box 18
            b.addBox(new Box());        // box 19
            b.addBox(new Box());        // box 20
            b.addBox(new Box());        // box 21
            c = new Box();              // box 2
            c.addBox(a);
            c.addBox(b);
            mainBox.addBox(c);

            // Create the bottom right blue box and its contents
            a = new Box();              // box 22
            a.addBox(new Box());        // box 27
            a.addBox(new Box());        // box 28
            b = new Box();              // box 9
            b.addBox(a);
            c = new Box();              // box 3
            c.addBox(b);
            mainBox.addBox(c);
      }
}
```

Now, how could we write a function to unwrap a box (as well as all boxes inside of it until there are no more) and return the number of boxes that were unwrapped in total (including the outer box) ?

Do you understand that this problem can be solved recursively, since a **Box** is made up of other **Boxes** ? The problem is solved similarly to the binary tree example since we can view the main box as the "root" of the tree while the boxes inside of it would be considered the children (possibly more than 2).

What is the base case(s) ? What is the simplest box ? Well, a box with no internal boxes would be easy to unwrap and then we are done and there is a total of 1 box:

```
public int unwrap() {
      if (numInternalBoxes == 0)
            return 1;
}
```

This is simple. However, what about the recursive case ? Well, we would have to recursively unwrap and count all inside boxes. So we could use a loop to go through the internal boxes, recursively unwrapping them one-by-one and totaling the result of each recursive unwrap call as follows:

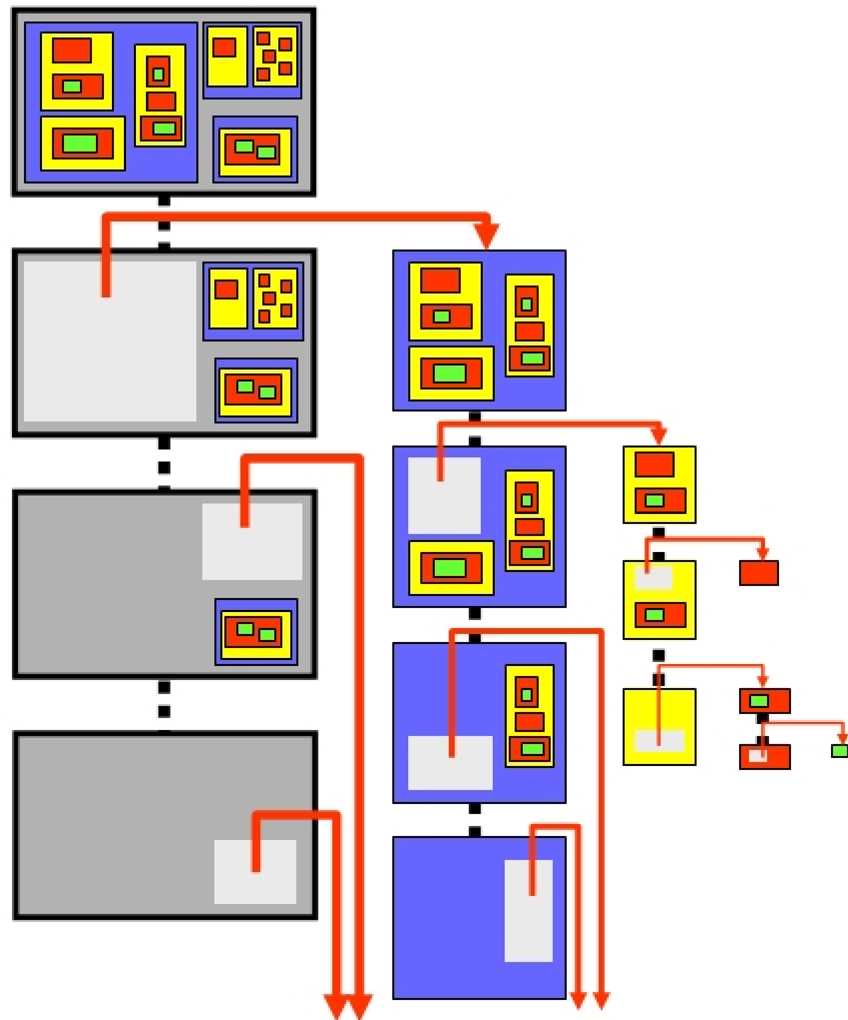
```
public int unwrap() {
	if (internalBoxes.size() == 0)
		return 1;
	// Count this box
	int count = 1;

	// Count each of the inner boxes
	for (Box b: internalBoxes)
		count = count + b.unwrap();
	return count;
}
```

Notice how we adjusted the base case. The **for** loop will attempt to unwrap all internal boxes recursively. If there are no internal boxes, then the method returns a value of 1 ... indicating this single box.

Each recursively-unwrapped box has a corresponding **count** representing the number of boxes that were inside of it (including itself). These are all added together to obtain the result.

The above code does not modify the **internalBoxes** list for any of the boxes. That is, after the function has completed, the box data structure remains intact and unmodified. This is known as a ***non-destructive*** solution because it does not destroy (or alter) the data structure. In real life however, the boxes are actually physically opened and the contents of each box is altered so that when completed, no box is contained in any other boxes (i.e., the list is modified/destroyed).

Alternatively, we can obtain the same solution without a **for** loop by allowing the arrayLists to be destroyed along the way. This would be called a ***destructive*** solution. Destructive solutions are often simpler to code and understand, but they have the disadvantage of a modified data structure, which can be undesirable in some situations. Here is the process depicting a portion of such a destructive solution: →



How does this simplify our code ? If we are not worried about keeping the **innerBoxes** lists intact, we can simply "bite-off" a piece of our problem by removing one box from the main box (i.e., taking it out of the list of internal boxes) and then we have two smaller problems:

(1) the original box with one less internal box in it, and

(2) the box that we took out that still needs to be unwrapped.

We can simply unwrap each of these recursively and add their totals together:

```
public int unwrap2() {
      if (internalBoxes.size() == 0)
            return 1;
      // Remove one internal box, if there is one
      Box  insideBox = internalBoxes.remove(0);

      // Unwrap the rest of this box as well as the one just removed
      return this.unwrap2() + insideBox.unwrap2();
}
```

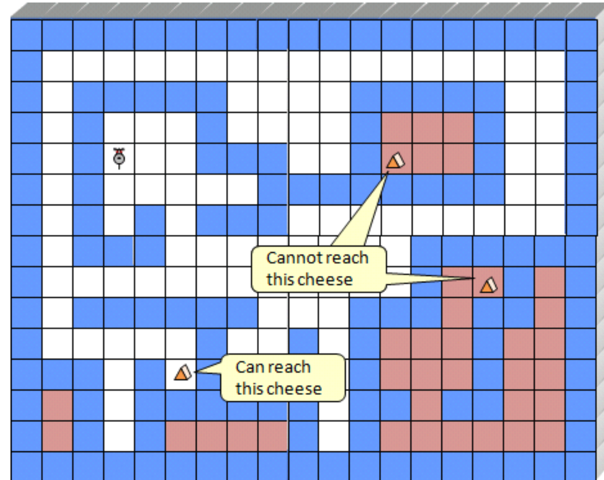
This is much smaller code now. It is also intuitive when you make the connection to the real-life strategy for unwrapping the boxes.

Of course, once the method completes ... the main box is empty ... since the boxes were removed from the array list along the way. If we wanted to ensure that the main box remained the same, we could put back the boxes after we counted them. This would have to be done AFTER the recursive calls ... but before the **return** statement:

```
public int unwrap3() {
      if (internalBoxes.size() == 0)
            return 1;
      // Remove one internal box, if there is one
      Box  insideBox = internalBoxes.remove(0);

      // Unwrap the rest of this box as well as the one just removed
      int result = this.unwrap3() + insideBox.unwrap3();

      // Put the box back in at position 0 (i.e., same order)
      internalBoxes.add(0,insideBox);

      return result;
}
```

This method is now considered ***non-destructive*** because the boxes are restored before the method completes.

9.3 A Maze Searching Example



Consider a program in which a rat follow the walls of a maze. The rat is able to travel repeatedly around the maze using the "right-hand rule". Some mazes may have areas that are unreachable (e.g., interior rooms of a building with closed doors). We would like to write a program that determines whether or not a rat can reach a piece of cheese that is somewhere else in the maze.

This problem cannot be solved by simply checking each maze location one time. It is necessary to trace out the steps of the rat to determine whether or not there exists a path from the rat to the cheese. To do this, we need to allow the rat to try all possible paths by propagating (i.e., spreading) outwards from its location in a manner similar to that of a fire spreading outwards from a single starting location.

Unlike a fire spreading scenario, we do not have to process the locations in order of their distance from the rat's start location. Instead, we can simply allow the rat to keep walking in some direction until it has to turn, and then choose which way to turn. When there are no more places to turn to (e.g., a dead end), then we can return to a previous "crossroad" in the maze and try a different path.

We will need to define a **Maze** class that the **Rat** can move in. Here is a basic class that allows methods for creating and displaying a maze as well as adding, removing and querying walls:

```
public class Maze {
      public static byte EMPTY = 0;
      public static byte WALL = 1;
      public static byte CHEESE = 2;

      private int        rows, columns;
      private byte[][]   grid;

      // A constructor that makes a maze of the given size
      public Maze(int r, int c) {
            rows = r;
            columns = c;
            grid = new byte[r][c];
      }
      // A constructor that makes a maze with the given byte array
      public Maze(byte[][] g) {
            rows = g.length;
            columns = g[0].length;
            grid = g;
      }
```

```
    // Return true if a wall is at the given location, otherwise false
    public boolean wallAt(int r, int c) {  return grid[r][c] == WALL;  }

    // Return true if a cheese is at the given location, otherwise false
    public boolean cheeseAt(int r, int c) {  return grid[r][c] == CHEESE;  }

    // Put a wall at the given location
    public void placeWallAt(int r, int c) {  grid[r][c] = WALL;  }

    // Remove a wall from the given location
    public void removeWallAt(int r, int c) {  grid[r][c] = EMPTY;  }

    // Put cheese at the given location
    public void placeCheeseAt(int r, int c) {  grid[r][c] = CHEESE;  }

    // Remove a cheese from the given location
    public void removeCheeseAt(int r, int c) {  grid[r][c] = EMPTY;  }

    // Display the maze in a format like this ------------>
    public void display() {
        for(int r=0; r<rows; r++) {
            for (int c = 0; c<columns; c++) {
                if (grid[r][c] == WALL)
                    System.out.print("W");
                else if (grid[r][c] == CHEESE)
                    System.out.print("c");
                else
                    System.out.print(" ");
            }
            System.out.println();
        }
    }

    // Return a sample maze corresponding to the one in the notes
    public static Maze sampleMaze() {
        byte[][] grid = { {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
                          {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
                          {1,0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,1},
                          {1,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1},
                          {1,0,1,0,0,0,1,1,1,0,0,1,0,0,0,1,0,0,1},
                          {1,0,1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,1},
                          {1,0,1,0,1,0,1,1,1,0,0,0,0,0,0,0,0,0,1},
                          {1,0,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,1},
                          {1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,1},
                          {1,0,1,1,1,1,1,1,0,0,0,1,1,1,0,1,1,0,1},
                          {1,0,0,0,0,0,1,0,0,1,0,1,0,0,0,1,0,0,1},
                          {1,1,1,0,1,0,1,0,0,1,0,1,0,0,0,1,0,0,1},
                          {1,0,1,0,1,1,1,1,1,1,0,1,1,0,1,1,0,0,1},
                          {1,0,1,0,1,0,0,0,0,1,0,1,0,0,0,0,0,0,1},
                          {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}};
        Maze m = new Maze(grid);
        m.placeCheeseAt(3,12);
        return m;
    }
}
```

```
WWWWWWWWWWWWWWWWWWW
W                 W
W WWWWW    WWWWW  W
W W   W    Wc  W  W
W W   WWW  W   W  W
W W     WWWWWWWW  W
W W W WWW         W
W WWW        WWWWWW
W            W  W W
W WWWWWW   WWW WW W
W     W  W W   W  W
WWW W W  W W   W  W
W W WWWWWW WW WW  W
W W W    W W      W
WWWWWWWWWWWWWWWWWWW
```

What does the **Rat** class look like ? It is very simple, for now:

```
public class Rat {
      private int row, col;

      // Move the Rat to the given position
      public void moveTo(int r, int c) {
            row = r; col = c;
      }
}
```

Let us see whether or not we can write a recursive function in the **Rat** class to solve this problem. The function should take as parameters the maze (i.e., a 2D array). It should return **true** or **false** indicating whether or not the cheese is reachable from the rat's location. Consider the method written in a **Rat** class as follows:

```
public boolean canFindCheeseIn(Maze m) {
      ...
}
```

What are the base cases for this problem ? What is the simplest scenario ? To make the code simpler, we will assume that the entire maze is enclosed with walls … that is … the first row, last row, first column and last column of the maze are completely filled with walls.

There are two simple cases:

1. If the cheese location is the same as the rat's location, we are done … the answer is true.

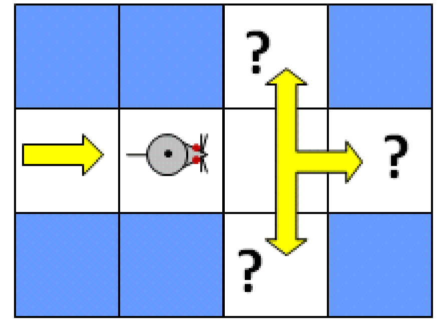
2. If the rat is on a wall, then it cannot move, so the cheese is not reachable. This is a kind of error-check, but as you will see later, it will simplify the code.

Here is the code so far:

```
public boolean canFindCheeseIn(Maze m) {
      // Return true if there is cheese at the rat's (row,col) in the maze
      if (m.cheeseAt(row, col))
            return true;

      // Return false if there is a wall at the rat's (row,col) in the maze
      if (m.wallAt(row, col))
            return false;
}
```

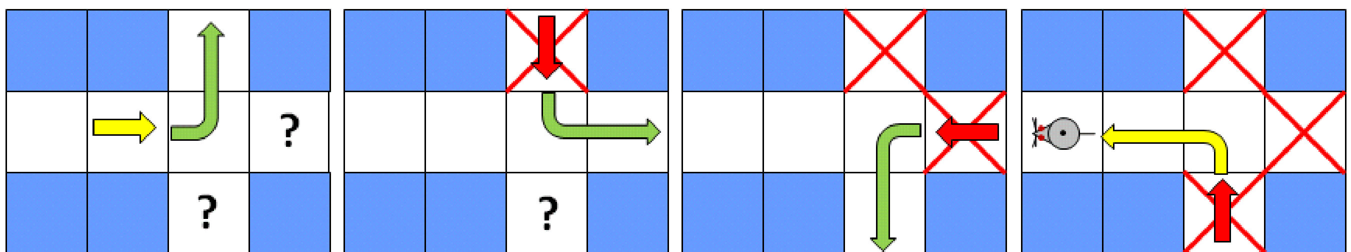
Notice that the **row** and **col** variables are the attributes of the **Rat** itself.

Now what about the recursion ? How do we "break off" a piece of the problem so that the problem becomes smaller and remains the same type of problem ? Well, how would you solve the problem if you were the rat looking for the cheese?



Likely, you would start walking in some direction looking for the cheese. If there was only one single path in the maze you would simply follow it. But what do you do when you come to a crossroads (i.e., a location where you have to make a decision as to which way to go) ?

Probably, you will choose one of these unexplored hallways, and then if you find the cheese … great. If you don't find the cheese down that hallway, you will likely come back to that spot and try a different hallway. If you find the cheese down any one of the hallways, your answer is **true**, otherwise…if all hallways "came up empty" with no cheese, then you have exhausted all possible routes and you must return **false** as an answer for this portion of the maze:



So the idea of "breaking off" a smaller piece of the problem is the same idea as "ruling out" one of the hallways as being a possible candidate for containing the cheese. That is, each time we check down a hallway for the cheese and come back, we have reduced the remaining maze locations that need to be searched.

This notion can be simplified even further through the realization that each time we take a step to the next location in the maze, we are actually reducing the problem since we will have already checked that location for the cheese and do not need to re-check it. That is, we can view each location around the rat as a kind of hallway that needs to be checked. So, the general idea for the recursive case is as follows:

> **if** (the cheese is found on the path to the left) **then return true**
> **otherwise if** (the cheese is found on the path straight ahead) **then return true**
> **otherwise if** (the cheese is found on the path to the right) **then return true**
> **otherwise return false**

There are only three possible cases, since we do not need to check behind the rat since we just came from that location. However, the actual code is a little more complicated. We need, for example, to determine the locations on the "left", "ahead" and "right" of the rat, but this depends on which way the rat is facing. There would be the three cases for each of the 4 possible rat-facing directions. A simpler strategy would simply be to check all 4 locations around the rat's current location, even though the rat just came from one of those locations. That way, we can simply check the 4 maze locations in the array around the rat's current location.

Here is the idea behind the recursive portion of the code:

move the rat up
if (**canFindCheese**(**maze**)) **then return true otherwise** move the rat back down

move the rat down
if (**canFindCheese**(**maze**)) **then return true otherwise** move the rat back up

move the rat left
if (**canFindCheese**(**maze**)) **then return true otherwise** move the rat back right

move the rat right
if (**canFindCheese**(**maze**)) **then return true otherwise** move the rat back left

return false

The code above has 4 recursive calls. It is possible that all 4 recursive calls are made and that none of them results in the cheese being found.

However, there is a problem in the above code. With the above code, the rat will walk back and forth over the same locations many times … in fact … the code will run forever … it will not stop. The problem is that each time we call the function recursively, we are not reducing the problem. In fact, each time, we are simply starting a brand new search from a different location.

The rat needs a way of "remembering" where it has been before so that it does not "walk in circles" and continue checking the same maze locations over and over again. To do this, we need to leave a kind of "breadcrumb trail" so that we can identify locations that have already been visited.

We can leave a "breadcrumb" at a maze location by changing the value in the array at that row and column with a non-zero & non-wall value such as **-1**. Then, we can treat all **-1** values as if they are walls by not going over those locations again.

We can add code to the **Maze** class to do this:

```
public static byte BREAD_CRUMB = -1;

// Mark the given location as visited
public void markVisited(int r, int c) {
      grid[r][c] = BREAD_CRUMB;
}

// Mark the given location as not having been visited
public void markUnVisited(int r, int c) {
      grid[r][c] = EMPTY;
}

// Return true of the location has been visited
public boolean hasBeenVisited(int r, int c) {
      return grid[r][c] == BREAD_CRUMB;
}
```

Now we can adjust our code to avoid going to any "visited locations" and to ensure that each location is visited. We can also put in the code to do the recursive checks now as follows:

```
public boolean canFindCheeseIn(Maze m) {
      // Return true if there is cheese at the rat's (row,col) in the maze
      if (m.cheeseAt(row, col))
            return true;

      // Return false if there is a wall at the rat's (row,col) in the maze
      if (m.wallAt(row, col) || m.hasBeenVisited(row, col))
            return false;

      // Mark this location as having been visited
      m.markVisited(row, col);

      // Move up in the maze and recursively check
      moveTo(row-1, col);
      if (canFindCheeseIn(m))
            return true;

      // Move back down and then below in the maze and recursively check
      moveTo(row+2, col);
      if (canFindCheeseIn(m))  return true;

      // Move back up and then left in the maze and recursively check
      moveTo(row-1, col-1);
      if (canFindCheeseIn(m))  return true;

      // Move back and then go right again in the maze and recursively check
      moveTo(row, col+2);
      if (canFindCheeseIn(m))  return true;

      // We tried all directions and did not find the cheese, so quit
      return false;
}
```

Notice that we are now returning with **false** if the location that the rat is at is a wall or if it is a location that has already been travelled on. Also, we are setting the rat's current maze location to **-1** so that we do not end up coming back there again.

After running this algorithm, the maze will contain many **-1** values. If we wanted to use the same maze and check for a different cheese location, we will need to go through the maze and replace all the **-1** values with **0** so that we can re-run the code. This recursive function is therefore considered to be destructive. Destructive functions are not always desirable since they affect the outcome of successive function calls.

However, there is a way to fix this right in the code itself. Notice that we are setting the maze location to **-1** just before the recursive calls. This is crucial for the algorithm to work. However, once the recursive calls have completed, we can simply restore the value to **0** again by placing the following just before each of the **return** calls:

```
m.markUnVisited(row, col);
```

For example:

```
// Move up in the maze and recursively check
moveTo(row-1, col);
if (canFindCheeseIn(m)) {
      moveTo(row+1, col);          // Move back down before marking
      m.markUnVisited(row, col);   // Unmark the visited location
      return true;
}
```

The code above should now do what we want it to do.

We need to test the code. To help debug the code it would be good to be able to display where the rat is and where the breadcrumbs are.

We can modify the **display()** method in the **Maze** class to take in the rat's location and do this as follows:

```
public void display(int ratRow, int ratCol) {
      for(int r=0; r<rows; r++) {
            for (int c = 0; c<columns; c++) {
                  if ((r == ratRow) && (c == ratCol))
                        System.out.print("r");
                  else if (grid[r][c] == WALL)
                        System.out.print("W");
                  else if (grid[r][c] == CHEESE)
                        System.out.print("c");
                  else if (grid[r][c] == BREAD_CRUMB)
                        System.out.print(".");
                  else
                        System.out.print(" ");
            }
            System.out.println();
      }
}
```

We can then use the following test program:

```
public class MazeTest {
      public static void main(String[] args) {
            Maze m = Maze.sampleMaze();
            Rat r = new Rat();
            r.moveTo(1,1);
            m.display(1,1);
            System.out.println("Can find cheese ... " +
                                    r.canFindCheeseIn(m));
      }
}
```

Also, by inserting **m.display(row,col);** at the top of your recursive method, you can watch as the rat moves through the maze:

```
WWWWWWWWWWWWWWWWWWW
W.                W
W.WWWWW    WWWWW  W
W.W   W    Wc  W  W
W.W   WWW  W   W  W
WrW     WWWWWWWW  W
W.W W WWW         W
W WWW        WWWWWW
W            W  W W
W WWWWWW   WWW WW W
W     W  W W   W  W
WWW W W  W W   W  W
W W WWWWWW WW WW  W
W W W    W W      W
WWWWWWWWWWWWWWWWWWW
```

```
WWWWWWWWWWWWWWWWWWW
W.                W
W.WWWWW    WWWWW  W
W.W...W    Wc  W  W
W.W...WWW  W   W  W
W.W.....WWWWWWWW  W
W.W.W.WWW         W
W.WWW....r   WWWWWW
W.........   W  W W
W.WWWWWW.. WWW WW W
W.....W..W W   W  W
WWW.W.W..W W   W  W
W W.WWWWWW WW WW  W
W W.W    W W      W
WWWWWWWWWWWWWWWWWWW
```

```
WWWWWWWWWWWWWWWWWWW
W.             r. W
W.WWWWW    WWWWW. W
W.W...W    Wc  W. W
W.W...WWW  W   W. W
W.W.....WWWWWWWW. W
W.W.W.WWW........ W
W.WWW........WWWWWW
W............W  W W
W.WWWWWW...WWW WW W
W.....W..W.W   W  W
WWW.W.W..W.W   W  W
W W.WWWWWW.WW WW  W
W W.W    W.W      W
WWWWWWWWWWWWWWWWWWW
```

```
WWWWWWWWWWWWWWWWWWW
W.r...............W
W.WWWWW....WWWWW..W
W.W...W....Wc  W..W
W.W...WWW..W   W..W
W.W.....WWWWWWWW..W
W.W.W.WWW.........W
W.WWW........WWWWWW
W............W  W W
W.WWWWWW...WWW WW W
W.....W..W.W   W  W
WWW.W.W..W.W   W  W
W W.WWWWWW.WW WW  W
W W.W    W.W      W
WWWWWWWWWWWWWWWWWWW
```

In the above case, the cheese was not found after an exhaustive search.

This page was intentionally left blank.