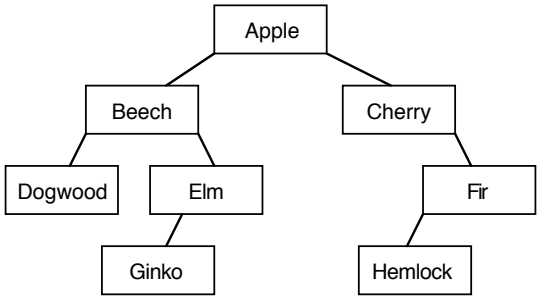
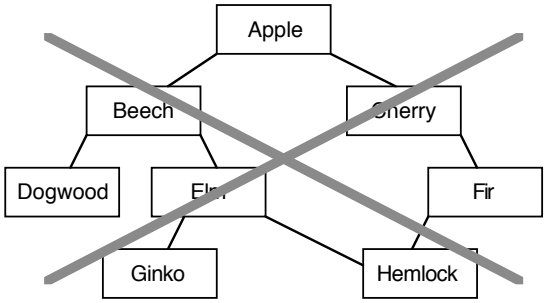
Chapter 15 Trees, Searching & Recursion

15.1 Trees

In computer science, a tree is a structure similar to a linked list with more than one link per node. The diagram is meant to illustrate a tree with names (of actual woody trees) as the data in its nodes. Most computer trees are *binary trees*, meaning that there are two links in each node. Either one or both of the links in a node may be *null*.



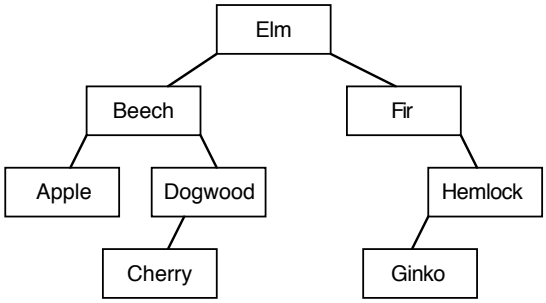
Various parts of a tree have names. The nodes immediately below a given node are called children (the children of *Beech* are *Dogwood* and *Elm*). The node above a given node is the parent (the parent of *Hemlock* is *Fir*). The node at the top is called the *root* node. Nodes with no children are called *leaves*. Yeah, computer trees are upside down, root on the top, leaves on the bottom.



Not all possible structures that can be built with nodes having more than one link are trees. A tree must have exactly one path from the root to each other node. The second illustration does not show a tree because there are two ways to get from *Apple* to *Hemlock* (via *Beech* and *Elm* or via *Cherry* and *Fir*).

Binary Search Trees

A binary search tree is a tree containing data sorted in a particular way. The illustration at the right shows words placed correctly for a *binary search tree*. The rule is that nodes to the left of a particular node have names earlier in the alphabet than the node itself and nodes to the right have names later in the alphabet. In standard illustrations the names read from left to right. More importantly, a search for an item in the tree can start at the root and proceed down a single path, turning left or right depending upon the alphabetical position of the item being searched for.



The advantage of a binary search tree over a linked list is that items in a binary search tree can be found without searching a very large portion of the nodes — a binary tree with a root to leaf length of 10 links can hold more than 1,000 nodes, a binary tree with a root to leaf length of 20 links can hold more than 1,000,000 nodes.

The classes below implement a binary search tree (real implementations include a routine for deleting a node from the tree — this is much more complex than *add*, *find*, or *toString*.

```
class SearchTree
   {
   private TreeNode head;

   public SearchTree()
      {
      head = null;
      }

   public void add(String data)
      {
      if (head == null)
         head = new TreeNode(data);
      else
         head.add(data);
      }

   public boolean isInTree(String data)
      {
      if (head == null)
         return false;
      return head.isInTree(data);
      }

   public String toString()
      {
      if (head == null)
         return "empty tree";
      return head.toString();
      }
   }
```

Nodes are always added to the bottom of the tree.

To find something, look until you find it or you find a *null* link.

To add something, look until you find it (in which case it is already there and you don't need to add it) or you find a *null* link. The *null* link you found is the place to put it (you can find it there since searching for it found that spot).

To list the tree in alphabetical order:

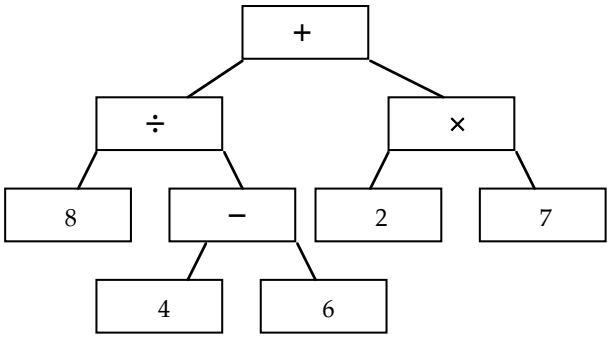
- each node asks its left child to contribute.
- the node's own name is added.
- each node asks its right child to contribute.

```
class TreeNode
   {
   private TreeNode left, right;
   private String data;

   public TreeNode(String s)
      {
      data = s;
      left = right = null;
      }

   public boolean isInTree(String s)
      {
      if (s.compareTo(data) < 0)
         if (left == null)
            return false;
         else
            return left.isInTree(s);
      if (s.compareTo(data) > 0)
         if (right == null)
            return false;
         else
            return right.isInTree(s);
      return true;
      }

   public void add(String s)
      {
      if (s.compareTo(data) < 0)
         if (left == null)
            left = new TreeNode(s);
         else
            left.add(s);
      if (s.compareTo(data) > 0)
         if (right == null)
            right = new TreeNode(s);
         else
            right.add(s);
      }

   public String toString()
      {
      String s = "";
      if (left != null)
         s += left.toString();
      s += data;
      if (right != null)
         s += right.toString();
      return s;
      }
   }
```

Expression Evaluation trees

The tree at right is meant to contain the arithmetic expression (8 ÷ (4 – 6)) + (2 × 7). A *toString* method for a binary search tree would print this tree out as 8 ÷ 4 – 6 + 2 × 7, which is almost good enough (the lack of parentheses allows confusion, a *toString* method for an expression evaluation tree should put in parentheses. The *toString* method below puts in parentheses (including a few more than necessary).

```
public String toString()
    {
    String s = "";
    if (left != null || right != null)
        s += "(";
    if (left != null)
        s += left.toString();
    s += data;
    if (right == null)
        s += right.toString();
    if (left != null || right != null)
        s += ")";
    return s;
    }
```



Alternate listing methods for expressions

One way to analyze the listing method (*toString*) is to think about the additions a node makes to the string.

left: The node requests that everything to its left be put in the string .
data: The node puts its own data into the string
right: The node requests that everything to its right be put into the string.

There are 3 standard ways of listing elements in a tree. The listings are created by changing the order in which a node makes additions to the string:

First method: If we order things as data–left–right, the listing comes out: + ÷ 8 – 4 6 × 2 7
Second method: If we order things as left–data–right, the listing comes out: 8 ÷ 4 – 6 + 2 × 7
Third method: If we order things as left–right–data, the listing comes out: 8 4 6 – ÷ 2 7 × +

The first method produces a well known way of printing out an expression of this sort. It is called *prefix* or *Polish* notation. To read the expression and see what operation refers to what number, start at the right and go left by the rules:

1. If you encounter a number, save it (for later).
2. If you encounter a operator, retrieve the last two numbers saved, perform the operation and save the result. Be especially careful with ÷ and –. These operations give wrong answers if you rearrange the order of the numbers. Be sure to keep the numbers in their original order.
3. When you run out of items, retrieve the only remaining number you have saved (it is the answer).

The second method (with parentheses thrown in) produces the method used in algebra books. It is called *infix* or *algebraic* notation.

The third method also produces a well known way of printing out an expression of this sort. It is called *postfix* or *reverse Polish* notation. To read the expression and see what operation refers to what number, start at the left and go right by the same rules as for *Polish* notation.

An expression in either *Polish*, or *reverse Polish* notation can be quickly and easily interpreted by a computer. Having a computer interpret an expression in algebraic notation is far more complex.

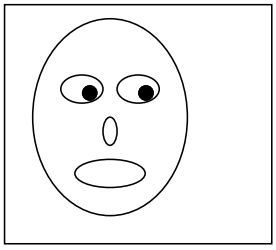
Expressions in *reverse Polish* are commonly used in computer programs. Expressions in *Polish* notation are neither harder nor easier to handle. It's likely that the left to right *reverse Polish* notation is used because early computers were mostly designed by English speakers (rather than Hebrew, Arabic, or Chinese speakers).

The name *Polish* notation comes from a paper by the Polish logician Jan Lukasiewicz — whose name English speakers can't seem to remember (or pronounce?).

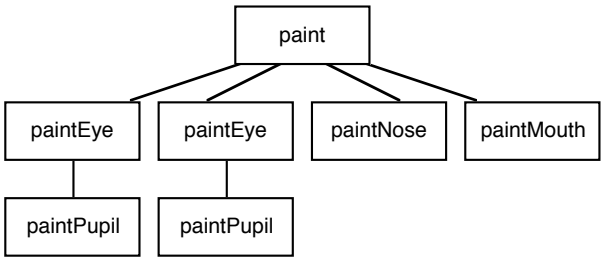
15.2 Making trees as you search them

Ephemeral trees

In one sense, programs have been searching trees since the first program was written with several methods. Consider a program that produces the image at the right. The *paint* method paints a blank face and then calls the methods *paintEye*, *paintNose* and *paintMouth*. The method *paintEye* calls the method *paintPupil*. There is a tree of sorts which is searched by the program as it draws the face.



The entire tree does not exist all at the same time. The program goes down the left-most branch to draw an eye. It then retreats to the paint method and goes down the branch that draws the second eye. Then, after a retreat to the paint method, it goes down the next branch to draw the nose. Finally, the last branch is used to draw the mouth.

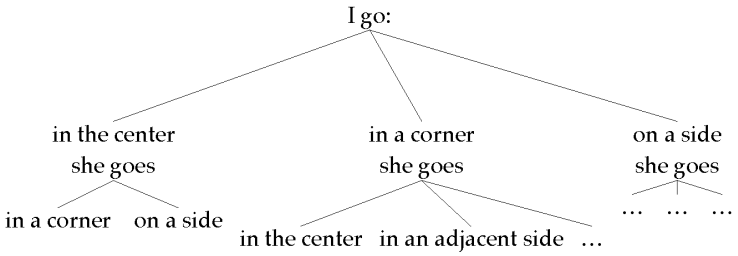


As the example above shows, a program can search through a 'tree' without creating the tree before hand, or even having the entire tree exist at any one time. The computer keeps track of the program's position in this tree with a structure known as the runtime stack. A list of the currently existing portion of the tree is called a stack trace.

Some problems contain patterned data that is naturally structured as a tree.

A program need not make the tree before searching the possibilities. Because of the pattern, the program can construct the branches of the tree as it searches them.

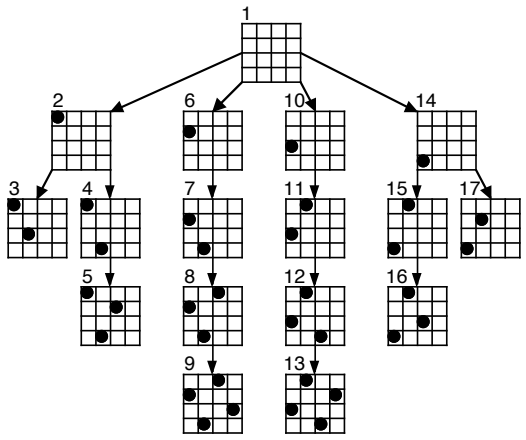
A game of tic-tac-toe:



An example — Queens on a Chess Board

One of the classic chess problems is to place 8 queens on a chess board without any of the queens being able to attack any of the others. A chess board is 8 × 8 squares. Queens can attack sideways to the edge of the board, up and down to the edge of the board and at either 45° angle to the edge of the board. Clearly you can't place more than 8 queens since each must go in a separate column.

The illustration at the right shows a search for the ways to put 4 queens on a 4×4 chess board. The diagram starts with an empty board at the top. Queens are placed on the board, column by column, starting at the left. When no more queens can be placed, the rightmost queen is removed — a queen is then placed lower on the board in the same column if possible. Two solutions exist for a 4×4 board, numbers 9 and 13 in the illustration. The solutions are mirror images of each other.



The program keeps track of its position in the tree with a single list of partial solutions. As the search goes down and up the branches of the tree, the list of partial solutions grows and shrinks.

The changing status of the list of partial solutions (as it grows and shrinks) is given below. For example 1→2→4→5 indicates that the search has progressed to partial solution 5, through 1, 2, and 4. At the next stage, 1→2→4, the program has retreated to partial solution 4…

1 1→2 1→2→3 1→2 1→2→4 1→2→4→5 1→2→4 1→2
1 1→6 1→6→7 1→6→7→8 1→6→7→8→9 1→6→7→8 1→6→7 1→6
1 1→10 1→10→11 1→10→11→12 1→10→11→12→13 1→10→11→12 1→10→11 1→10
1 1→14 1→14→15 1→14→15→16 1→14→15 1→14 1→14→17 1→14
1

To write a program to do the search, start with a board class:

`public Board()`	creates an empty board
`public boolean full()`	reports as to whether or not a board is full
`public Board makeFuller(int row)`	creates a board with another queen in the suggested row — when a fuller board is inappropriate, *null* is returned.
`public String toString()`	creates a string with the row numbers of the queens — board number 8 above would return *Board: 130*

The heart of a program is a method that accepts a board as input, displays a full board, builds a fuller board when possible and backs up when necessary.

```
private void makeBoards(Board b)
    {
    if (b != null)
        {
        if (b.full())                   // the board is full, print it out
            {
            count++;                    // count needs to be defined as a state variable
            System.out.println(count + "> " + b + " is full.");
            }
        else                            // the board is not full, add each possible queen to it
            {
            for (int r=0; r<Board.SIZE; r++)
                {
                makeBoards(b.makeFuller(r));
                }
            }
        }
    }
```

The method *makeBoards* should be started by *makeBoards(new Board())* which starts with an empty board. *Makeboards* will create an infinite loop if *makeFuller(r)* does not eventually return a full board or a *null* board. Try this program with *SIZE* equal to 4 and with a *println* command on the first line of *makeBoards*. The *makeFuller* method below only checks horizontally, not diagonally. To have *makeFuller* check diagonally, complete the *NWMatch* and *NEMatch* methods. If you don't you will get 24 answers.

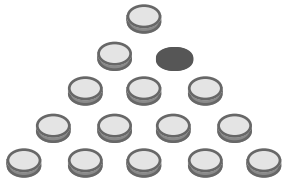
```
class Board
    {
    public static final int SIZE = 4;
    private int howMany;
    private int board[];

    public Board()
        {
        howMany = 0;
        board = new int[SIZE];
        }

    public boolean full()
        {
        return howMany >= SIZE;
        }

    public String toString()
        {
        String s = "Board: ";
        for (int c=0; c<howMany; c++)
            s += board[c];
        return s;
        }

    private boolean EWMatch(int aRow)
        {
        for (int c=0; c<howMany; c++)
            if (board[c] == aRow)
                return true;
        return false;
        }

    private boolean NWMatch(int aRow)
        {
        return false;
            // complete to catch queens on a diagonal
        }

    private boolean NEMatch(int aRow)
        {
        return false;
            // complete to check the opposite diagonal
        }

    public Board makeFuller(int aRow)
        {
        if (full()) return null;
        if (EWMatch(aRow)) return null;
        if (NWMatch(aRow)) return null;
        if (NEMatch(aRow)) return null;
        Board b = new Board();
        for (int c=0; c<howMany; c++)
            b.board[c] = board[c];
        b.board[howMany] = aRow;
        b.howMany = howMany + 1;
        return b;
        }
}
```

The program above, with the diagonals checked and the size changed to 8, will give 92 solutions. Only 12 of these are unique if rotations and mirror images are discarded.

Some other examples

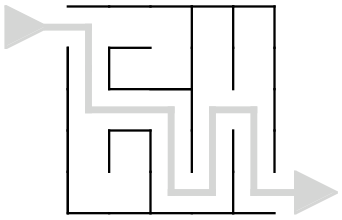
There are many other examples. Here are two.

Find a solution to the peg jumping problem posed by a triangular 15–hole puzzle. Pegs jump in six directions, horizontally and diagonally. Jumped pegs are removed. A solution is a series of 13 jumps (leaving only one peg).



The solution to this problem lies in the process of getting from the original, nearly full board, to the nearly empty one. A list of intermediate boards should be kept in a state variable (as the count was in the queens problem).

Find a solution to a maze. Start at the upper left and search in and out of the various blind alleys until a path to the opposite corner is found.



One method of constructing a maze is to start with the outside walls and then build up the inside walls a section at a time. As long as each new section connects to a wall at exactly one end, the original walls will grow towards each other without the path through the maze being closed off.

15.3 Recurrence Relations

A recurrence relation is a definition of something in terms of a simpler case of the same kind of thing. The simplest case(s) must be defined without reference to any other case. Two common mathematical examples are:

Mathematical examples

Example 1: The value of $n!$ (n factorial) is defined by:

1. 0! and 1! are both 1.
2. If $n > 1$, $n!$ is equal to $n \times (n–1)!$.

This defines the sequence: $0! = 1,\ 1! = 1,\ 2! = 2,\ 3! = 6,\ 4! = 24,\ 5! = 120,\ 6! = 720,\ 7! = 5040,\ \ldots$

This leads to a quick Java method:

```
public int factorial(int n)
   {
   if (n < 2)
      return 1;
   return n * factorial(n-1);
   }
```

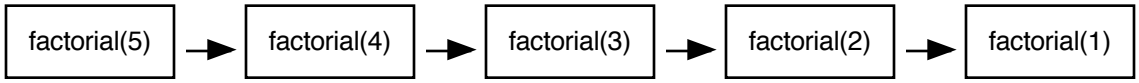
In this method, the recurrence relation is directly translated into Java. A method that includes the direct translation of a recurrence relation is called a *recursive* method.

Or this more involved method (with a loop):

```
public int factorialLoop(int n)
   {
   int fact = 1;
   for (int i=2; i<=n; i++)
      fact = fact * i;
   return fact;
   }
```

A method that uses a loop to build its result is called an *iterative* method.

Calling the *factorial* method with n equal to 5 searches the following tree – with only one branch, it's a list:



In the method *factorialLoop*, the result is built up as 1, 1×2, 1×2×3, 1×2×3×4, …

The methods *factorial* and *factorialLoop* do about the same amount of work. The iterative method will run somewhat faster since repeating a loop is faster than calling a method. With modern computer systems, the difference between these times is generally not excessive. However, with older system (20+ years), the difference could be very large. For this reason, older programming texts included whole sections on how to replace recursive methods with iterative ones.

Example 2: The Fibonacci sequence of integers is defined by:

1. The first two numbers are both 1.
2. After the first two numbers, each number is the sum of the two immediately preceding it.

This defines the sequence: $f_1 = 1,\ f_2 = 1, f_3 = 2,\ f_4 = 3,\ f_5 = 5,\ f_6 = 8,\ f_7 = 13,\ f_8 = 21,\ f_9 = 34,\ f_{10} = 55, \ldots$

This leads to a quick *recursive* method:

```
public int fibonacci(int n)
   {
   if (n < 3)
      return 1;
   return fibonacci(n-1) + fibonacci(n-2);
   }
```

The method *fibonacci* is *recursive*, the recurrence relation is directly translated into Java. ⇑

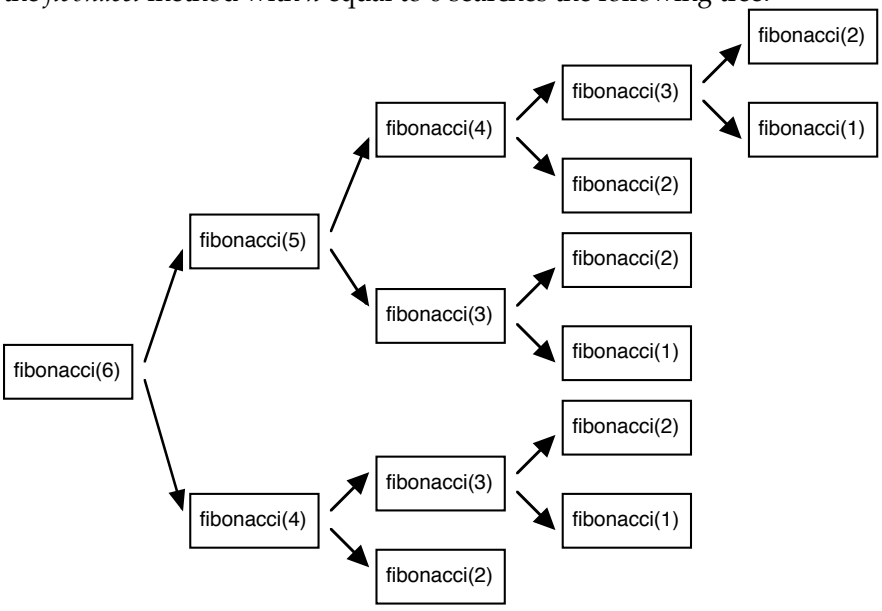
The method *fibonacciLoop* is *iterative*, the loop builds up the result. ⇒

Or this more involved iterative method:

```
public int fibonacciLoop(int n)
   {
   if (n < 3)
      return 1;
   int backTwo = 1;
   int backOne = 1;
   int fib = 1;
   for (int i=3; i<=n; i++)
      {
      fib = backOne + backTwo;
      backTwo = backOne;
      backOne = fib;
      }
   return fib;
   }
```

The methods *fibonacci* and *fibonacciLoop* do not do the same amount of work. For numbers that are not close to the beginning of the sequence, the recursive method does a great deal more work.

The *fibonacci* method makes two recursive method calls. This produces a search of a binary tree. Calling the *fibonacci* method with *n* equal to 6 searches the following tree:



The tree above isn't that big, only 15 nodes. However, binary trees have a habit of roughly doubling in size with each increase in depth. When *n* is in the mid-twenties, the number of nodes is in the millions and the time required for the computation gets out of hand.

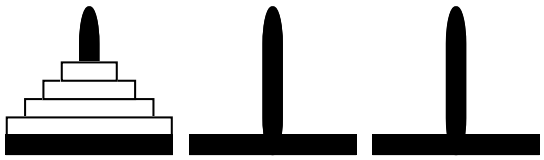
Even the small tree above has a lot of repetition. For example, the entire lower branch (starting with fibonacci(4)) is repeated at the upper right.

The *fibonacciLoop* method builds up the result with a loop that remembers the previous two results. Remembering the previous two results avoids the repetitious calculations of *fibonacci*, making the *fibonacciLoop* method quite quick.

A Puzzle Example — the Towers of Hanoi

This classic puzzle consists of 3 posts and a series of disks with holes in the center. No two disks have the same diameter. The disks are initially placed on the first post, in order with the largest on the bottom (the smallest on the top).

The idea is to move all of the disks onto the second post.



The rules are:

1. Only one disk may be moved at a time.
2. No disk may be placed on a smaller one.

A solution can be stated quickly with a recurrence relation:

1. If there is only one disk to move from post A to post B, just move it.
2. If there are *n* disks to move from post A to post B, move the top *n*–1 disks from post A to the third post, move the bottom disk from post A to post B and finally, move the *n*–1 disks from the third post to post B.

To print out a solution, the following methods will work. Call *moveStackOfDisks* with the number of disks in the problem as its parameter. Don't put in a number of disks above 4 or 5 unless you have a great amount of patience. The problem requires a stack of integers class (called *intStack*). Stacks are discussed in chapter 13. The stack of integers class should have a *String* instance variable to serve as its name.

```
public void moveStackOfDisks(int howManyDisks)
    {
    intStack a, b, c;
    a = new intStack("A");
    b = new intStack("B");
    c = new intStack("C");
    for (int n=howManyDisks; n>0; n--)
        a.push(n);
    moveDisks(a, b, c, howManyDisks);
    }

private void moveDisks(intStack from, intStack to, intStack extra, int howManyDisks)
    {
    if (howManyDisks > 1)
        moveDisks(from, extra, to, howManyDisks-1);
    int bottomDisk = from.pop();
    System.out.println("Move disk of size " + bottomDisk + " from "
                                + from.getName() + " to " + to.getName());
    to.push(bottomDisk);
    if (howManyDisks > 1)
        moveDisks(extra, to, from, howManyDisks-1);
    }
```