Project… Printing a Heap

The purpose of this project will be to print a heap. The reason this is important is that in the next lesson we will implement a priority queue based on an array heap. During development and debugging it is important to be able to test the heap at intermediate points by viewing a printout of the heap.

Create a text file called *HeapData.in* with the following content.

A
C
M
G
D
S
Q
N
Z
W
P
U
T

Notice that the sequence of these items is in the index sequence of nodes in the heap in Fig. 55-3. Create a project called *PrintTree* with class *Tester* that will input the letters from the *HeapData.in* text file and print the heap in the following fashion.

A
C M
G D S Q
N Z W P U T

It is suggested that each letter be put into an array starting with index 1 as is conventional with heaps (index 0 in the array will be unused). The only real challenge here is in determining when to start a new line. Notice a new line is started after the character with index **1** is printed, after the character with index **3** is printed, after the character with index **7** is printed, after the character with index **15** is printed…. These indices form the following pattern:

2^1-1, 2^2-1, 2^3-1, 2^4-1, …

Use an *if* statement to detect when these character are printed and then start a new line.

Project… A Heap of Trouble (advanced)

The printout produced by the previous project is just barely better than no printout at all. What we need is a printout that looks closer to the real thing,…like the following:

```
                A
        C               M
    G       D       S       Q
  N   Z   W   P   U   T
```

Fig. 55-12 A more desirable printout. Even though there are no connecting lines we can still tell what a parent node's children are.

To produce this printout we need to notice several things about the horizontal spacing. In order to see some patterns our first step will be to place the printed characters in a grid as in Fig. 55-13.

← 7 spaces→							**A**						
← 3 sp→			**C**	← 7 spaces→							**M**		
1sp	**G**	← 3 sp→			**D**	← 3 sp→			**S**	← 3 sp→			**Q**
N	1sp	**Z**	1sp	**W**	1sp	**P**	1sp	**U**	1sp	**T**			

Fig. 55-13 Detail of desired horizontal spacing. As we move from row to row the number of spaces form a predictable pattern.

The spacing in the grid above falls into two different distinct patterns.

1. As we proceed down the rows the **leading spaces** (lighter shading above) form the following pattern where *n* (4 for this example) is the number of levels of the tree.
   $7 = 2^{n-1}-1$ level 1
   $3 = 2^{n-2}-1$ level 2
   $1 = 2^{n-3}-1$ level 3
   $0 = 2^{n-4}-1$ level 4

2. As we proceed down the rows the **inner spaces** (darker shading above) form the following pattern where *n* (4 for this example) is the number of levels of the tree.
   inner spaces do not apply to the first level
   $7 = 2^{n-1}-1$ level 2
   $3 = 2^{n-2}-1$ level 3
   $1 = 2^{n-3}-1$ level 4

The code you write for the *Tester* class in this project, *HeapOfTrouble*, should detect which row we are on as was suggested in the previous project and apply the appropriate formulas to generate the desired number of spaces. We should mention at this point that we are assuming that the console output of your IDE uses monospaced fonts. Most do. If your IDE lets you select fonts, choose one of the Courier fonts. They are monospaced which means each character takes up the same amount of horizontal space when printed. See Appendix AB for more on monospaced fonts.

Another critical aspect of this project is the ability to determine from the number of items to be printed just how many levels (*n*) there will be for the tree. We will use a logarithm base 2 to accomplish this. The following chart shows the relationships involved.

Index (i)	$Log_2(i)$	Value	(int) $Log_2(i)$	Tree Level (n)
1	$Log_2 1$	0.0	0	1
2	$Log_2 2$	1.0	1	2
3	$Log_2 3$	1.5849624872207642	1	2
4	$Log_2 4$	2.0	2	3
5	$Log_2 5$	2.321928024291992	2	3
6	$Log_2 6$	2.5849626064300537	2	3
7	$Log_2 7$	2.8073549270629883	2	3
8	$Log_2 8$	3.0	3	4
9	$Log_2 9$	3.1699249744415283	3	4
10	$Log_2 10$	3.321928024291992	3	4
11	$Log_2 11$	3.4594316482543945	3	4
12	$Log_2 12$	3.5849626064300537	3	4
13	$Log_2 13$	3.700439691543579	3	4
14	$Log_{21} 4$	3.8073549270629883	3	4
15	$Log_2 15$	3.906890630722046	3	4
16	$Log_2 16$	4.0	4	5
…	…	…	…	…

Table 55-1

Math.log(x) is given for base *e*, so to calculate log base *2* of *x* we must use the following standard formula for converting log bases:

```
Math.log(x) / Math.log(2)
```

Finally, the number of tree levels (*n*) required for *x* items is given by:

```
int n = (int)( Math.log(x) / Math.log(2) ) + 1;
```

Use portions of the last project to produce a new project, *HeapOfTrouble*, in which we have two classes, *Tester* and *HeapPrinter*. The *HeapPrinter* class will have a *static* method called *printHeap*. By putting this *static* method in a class by itself we will easily be able to use this class for testing our implementation of an array heap in the next lesson.

Use the *Tester* class to input the *HeapData.in* file from the last project. In *main* of *Tester* be sure to pack the incoming data into an array in which the first element of the file has index *1*. In the *printHeap* method determine the number of levels and the number of spaces needed as discussed above. Use these ideas to print the heap so as to look like Fig. 55-12.

Admittedly, the heap printer from this project is very simplistic in that it assumes that each node of the heap prints as just a single character. What if the nodes to be printed consist of differing numbers of characters? At first this might seem like a difficult task to tackle; however, all we need to do is determine the maximum number of digits to be printed and then set up print "fields" that always print this same width. When printing any *String* representing a node having a smaller number of characters than this maximum field width, we simply "pack" both sides of the *String* with the appropriate number of characters so as center the characters in the field.