"If you can't beat your computer at chess, try kickboxing."
-Anon.

"Hardware: the parts of a computer that can be kicked."
-Jeff Pesis

"Hardware: the parts of a computer that can be kicked."
-Jeff Pesis

"The most overlooked advantage to owning a computer is that if they foul up, there's no law against whacking them around a little."
-Porterfield

"Never trust a computer you can't throw out a window."
-Steve Wozniak

Double buffer
Graphics buffer example

```
main program uses backPt

clearScreen (backgroundColor)
drawPixel(I,J,color)
outChar(letter,color)
switch()

unsigned char Buf1[8][16]; // [row][column]
unsigned char Buf2[8][16];
unsigned char *BackPt; // buffer used to build
unsigned char *FrontPt; // buffer used to display
```
GREEN  equ %10101010
RED    equ %01010101
YELLOW equ %11111111
maskTab fcb $C0,$30,$0C,$03

(This section contains a program code snippet for pixel manipulation in an embedded microcomputer system.)

* Inputs I,J,color passed on stack
mask  set  0
cmask set  1
I     set  4
J     set  5
color set  6

* Allocate masks

(pixel(I,J,color)
  addr = *backPt+16*I+J/4;  {backPt}+16+1
  mask = maskTab[J&0x03];     $30
  data = *addr;               read old
  data = data&(~mask);        ~$30 is $CF
  data = data|(mask&color);   $30&$55 is $10
  *addr = data;               write new

  Pixel(I,J,color) is a function that modifies a pixel at position (I,J) with the specified color (GREEN, RED, or YELLOW). The function uses a mask to determine which bits in the pixel are to be changed, and it updates the pixel with the new color.)
Introduction to Embedded Microcomputer Systems

10.9. Trees

Figure 10.11. Graphs and trees have nodes and are linked with pointers.
Figure 10.12. A tree can be constructed with only down arrows, and there is a unique path to each node.

Figure 10.13. A binary tree is constructed so that earlier elements are to the left and later ones to the right.

| Value equ 0 name of the node | #define NULL 0 |
| Data equ 1 data for this node | const struct Node{ |

Jonathan W. Valvano
**Program 10.20. Definition of a simple binary tree.**

<table>
<thead>
<tr>
<th>Left</th>
<th>equ</th>
<th>2</th>
<th>pointer to son</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>equ</td>
<td>4</td>
<td>pointer to son</td>
</tr>
<tr>
<td>ROOT</td>
<td>fdb</td>
<td>WS</td>
<td>Pointer to top</td>
</tr>
<tr>
<td>NULL</td>
<td>equ</td>
<td>0</td>
<td>undefined address</td>
</tr>
<tr>
<td>WS</td>
<td>fcb</td>
<td>'S',1 name, data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb WF</td>
<td>Left son</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb WV</td>
<td>Right son</td>
</tr>
<tr>
<td>WV</td>
<td>fcb</td>
<td>'V',2 name, data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb WT</td>
<td>WT is a left son</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb NULL</td>
<td>no right son</td>
</tr>
<tr>
<td>WT</td>
<td>fcb</td>
<td>'T',3 name, data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb NULL</td>
<td>no children</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb NULL</td>
<td>no right son</td>
</tr>
<tr>
<td>WF</td>
<td>fcb</td>
<td>'F',4 name, data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb WA</td>
<td>WA is a left son</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb NULL</td>
<td>no right son</td>
</tr>
<tr>
<td>WA</td>
<td>fcb</td>
<td>'A',5 name, data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fdb NULL</td>
<td>no children</td>
</tr>
</tbody>
</table>

```c
unsigned char Value;
unsigned char Data;
const struct Node *Left;
const struct Node *Right;};
typedef const struct Node NodeType;
typedef NodeType * NodePtr;
#define Root WS
#define WS &Tree[0]
#define WV &Tree[1]
#define WT &Tree[2]
#define WF &Tree[3]
#define WA &Tree[4]
NodeType Tree[5]={
    { 'S',1, WF, WV},
    { 'V',2, WT, NULL},
    { 'T',3, NULL, NULL},
    { 'F',4, WA, NULL},
    { 'A',5, NULL, NULL}};
```

**Program 10.21. Binary tree search functions.**

*Inputs: Reg A = look up letter*
*Outputs: Reg A=0 if not found, =data if found*

```c
int Look(unsigned char letter){
    NodePtr pt = Root;  /* top */
    while(pt!=NULL){ // done if null
        if(pt->Value == letter){
            return(pt->Data); /* good */
        }
        if(pt->Value < letter){
            pt = pt->Right;
        } else{
            pt = pt->Left;
        }
    }
    return NULL; /* not in tree */
}
```

In order to add and remove nodes at run time
the tree must be defined in RAM.
first search for the word (the search should fail),
change the null pointer to point to the new list.
Program 10.22. Program to add a node to a binary tree.

Figure 10.14 shows the binary tree as the nodes J, U, G are added to the dictionary.

Figure 10.14. Nodes are added to a binary tree such that the alphabetical order is maintained.

The search time for a binary tree increases as the log₂ of the size of the dictionary.

I suggest you do tutorial 10 before next week’s lab!!!