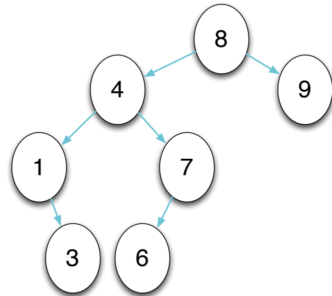
CS 237 Meeting 19 — 10/24/12

Announcements

1. Midterm: New date: Oct 29th. In class open book/notes.
2. Try to complete the linear feedback shift register lab in one sitting (and please put all the equipment away when you are done so it will not get in the way of others who need to use the Macs in that lab).

Structs, Pointers, and Recursive Structures

1. Last time, we were considering how to write a C program that would sort numbers by first organizing them into a binary tree so that all the numbers in the left subtree of a node are less than the number in the node and all the numbers in the right subtree of a node are greater than (or equal to) the number in the node.



   Then, the program could extract the numbers in order by completing a pre-order traversal of the tree.

2. We started by looking at a Java program that would sort in this manner using a recursive class to represent trees:

```
public class IntTree {

    private int value;
    private IntTree left;
    private IntTree right;

    public IntTree( int value ) {
        this.value = value;
    }

    public void preOrder( ) {
        if( left != null ) {
            left.preOrder();
        }<
        System.out.print( value + " " );
        if ( right != null ) {
            right.preOrder();
        }
    }

    public void insert( int value ) {
        if ( value < this.value ) {
            if ( left == null ) {
                left = new IntTree( value );
            } else {
                left.insert( value );
            }
        } else {
            if ( right == null ) {
                right = new IntTree( value );
            } else {
                right.insert( value );
            }
        }
    }
}
```

along with a main method to drive the process:

```
import java.util.*;

public class Sort {
```

```
    public static void main( String argv[] ) {
        Scanner input = new Scanner( System.in );

        int value = input.nextInt();
        IntTree tree = new IntTree( value );
        while ( input.hasNextInt() ) {
            tree.insert( input.nextInt() );
        }
        tree.preOrder();
    }
}
```

3. We saw that in C, we need to use pointer types to provide a way to terminate the recursion in a recursive type. For example, to define an intTree structure in C we would say:

```
typedef struct intTree {
  int value;
  struct intTree *left;
  struct intTree *right;
} *intTree;
```

   That is, we will have each structure that represents a tree hold two (possibly null) pointers to its subtrees.

4. Note that we also defined intTree as a pointer type. Most of the time, it will be the pointer type that we need to refer to rather than the structure type. When and if we do need to refer to the structure type we can say "struct intTree".

Separate Compilation of C Programs

1. Even this program is quite small, we have broken it up into several smaller files to illustrate how this is done in C.

   - First, we will first create a file named `intTree.h` containing the code

```
typedef struct intTree {
  int value;
  struct intTree *left;
  struct intTree *right;
} * intTree;

// Insert a node for the number value in the
// appropriate place in a tree
void insert( intTree *rootPtr, int value );

// Visit and print the values of all nodes in root
void preOrder( intTree root );
```

   - The typedef in this file describes the struct type used to represent tree nodes. The function headers provide enough information to use the functions associated with the type in the definition of main even though it might be in a different file from the function definitions themselves.
   - In the file containing main, we include a `#include` for this .h file.

```
#include <stdio.h>
#include "intTree.h"

intTree root = NULL;

// Sort an input file full of integers
int main( int argc, char *argv[] ) {

  // Put all the numbers from the file into a tree
  int newValue;
  while( scanf( "%d", &newValue ) == 1 ) {
    insert( &root, newValue );
  }

  // Traverse the tree while printing its contents
  preOrder( root );

  // Free all the tree nodes to clean up nicely
```

```
    freeTree( root );
  }
```

- Similarly, we include the .h file in the .c file that defines insert and preOrder:

```
#include "intTree.h"
#include <stdlib.h>
#include <stdio.h>


// Visit and print the values of all of the nodes in root
void preOrder( intTree root ) {
  if ( root != NULL ) {
    preOrder( root->left );
    printf("%d ", root->value );
    preOrder( root->right );
  }
}

// Make a new, one-node tree containing a give value
intTree newTree( int value ) {
  intTree result =
        (intTree) malloc( sizeof( struct intTree ) );

  result->value = value;
  result->left = result->right = NULL;
  return result;
}

// Insert a node for the number value in the appropriate place
// in a tree
void insert( intTree *rootPtr, int value ) {
  if ( *rootPtr == NULL ) {
    *rootPtr = newTree( value );
  } else if ( value <= (*rootPtr)->value ) {
    insert( &(*rootPtr)->left, value );
  } else {
    insert( &(*rootPtr)->right, value );
  }
}

// Free up the memory allocated for a tree
void freeTree( intTree root ) {
  if ( root != NULL ) {
    freeTree( root->right );
    freeTree( root->left );
    free( root );
  }
}
```

Passing Pointers by Reference

1. The insert function in our tree sort code is an example of a C function that takes a pointer variable as a reference parameter.
   - If the method is passed a pointer variable that is NULL, it will update the variable to point to a newly created tree.
2. As in our swap example, we only need to do three things to pass a parameter by reference:
   - Add a star to the declaration of the parameter to indicate that it will hold a pointer to something of the parameter type rather than a value of the parameter type.
   - Throughout the body of the function, use "*x" instead of "x" to refer to the parameter (assuming its name is x).
   - When invoking the function, put an ampersand in front of the variable begin passed.
3. When pointers are passed by reference, however, things at least look a little more complicated syntactically.
   - If you were going to say "p->c" and try changing it to "*p->c" you will discover that the precedence rules are wrong (C tries to evaluate the arrow before the star) so you have to write "(*p)->c".

Memory Allocation with malloc

1. The intTree.c file contains a function named newTree that is not mentioned in the .h file. This is an implicit way of making newTree "somewhat private". In fact, it could be called from the file containing main, but some warnings would probably be generated since the compiler would not know what types of parameters it expected.

2. This function invokes a very interesting function named `malloc`. malloc is short for "memory allocate". This system function takes one int parameter specifying an amount of memory measured in bytes. It returns a pointer to an available block of memory of the requested size (or a null pointer if the memory requested is not available).

   - The malloc function is declared in the file stdlib.h. That is why the intTree.c file includes this header file.
   - malloc is usually used to allocate space for a value of some particular type. In such cases, the appropriate parameter value to pass to malloc is the number of bytes required to store a value of the type. C provides a "sizeof" operation (which looks deceptively like a function) that can be applied to a type name or a variable. It returns the size of a value of the type or the type of the variable in bytes.
   - This is why the parameter to malloc in our code is `sizeof( struct intTree )`.
   - The method is declared to return a value of type "void *". That is, it returns a pointer to nothing. Since pointers to nothing are not of much worth, almost all calls to malloc are preceded by a cast that tell C the type of value that the programmer actually plans to store in the allocated memory.
   - This is why the call to malloc in our code is preceded by "`(intTree)`".
   - Note: In our code intTree is the type of pointers to structures that represent intTrees. That is struct intTree and intTree are not the same type!

Free at Last

1. Unlike Java, C does not have a process that automatically reclaims memory space that has been dynamically allocated (i.e. there is not garbage collector).

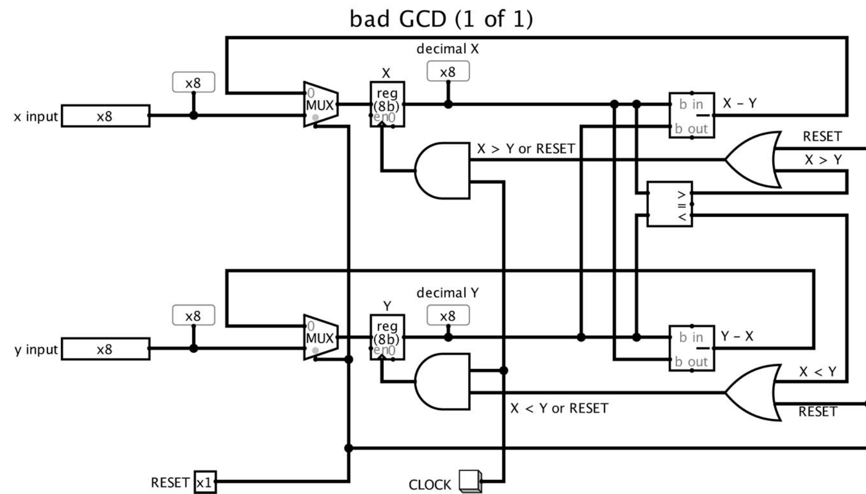
2. Instead, a C program must take responsibility for telling the system when it can reuse memory allocated using malloc by invoking a function named free with a pointer to any space that will no longer be used.

3. Our sort program is not a very interesting example of this, because all of the memory is needed until the very end of the program. Nevertheless, our program includes a function that shows how one could free an entire tree of dynamically allocated structures:

```
// Free up the memory allocated for a tree
void freeTree( intTree root ) {
  if ( root != NULL ) {
    freeTree( root->right );
    freeTree( root->left );
    free( root );
  }
}
```

Synchronous Sequential Circuits

1. Consider the following attempt to solve the GCD circuit construction exercise included in last week's lab:



2. This circuit does not work as desired. It is an example of a sequential circuit with a bad synchronization problem.

   - The problems with the circuit are a result of the fact that the clock inputs to the two registers are not tied directly to the clock, but instead are connected to the output of gates whose inputs involve both the clock and circuit elements involving the outputs of the registers.
   - The intent of the circuits design is that the clock signal received by a register will only be 1 if the main clock has been set to 1 and the value in that register is currently greater than the value in the other register.
   - Assuming this is true when the clock first become 1, one register will see a rising clock pulse and load a new value.
   - This new value will quickly become the registers new output.
   - It will be smaller than the old output.
   - It will feed into the comparator used to determine which register should change.
   - If the decrease in value of the register that changed makes it less than the value in the other register, the output of the comparator will change.
   - The clock signal for the other register is the and of the output of the comparator that had been 0 and the main clock.
   - If the comparator changes it output while the main clock is still 1, the second register will see a rising clock pulse and load a new value.
   - TWO REGISTER UPDATES HAVE OCCURRED AT DIFFERENT TIMES AS A RESULT OF A SINGLE CLOCK PULSE.
   - This was probably not the intent of the designer of the circuit.

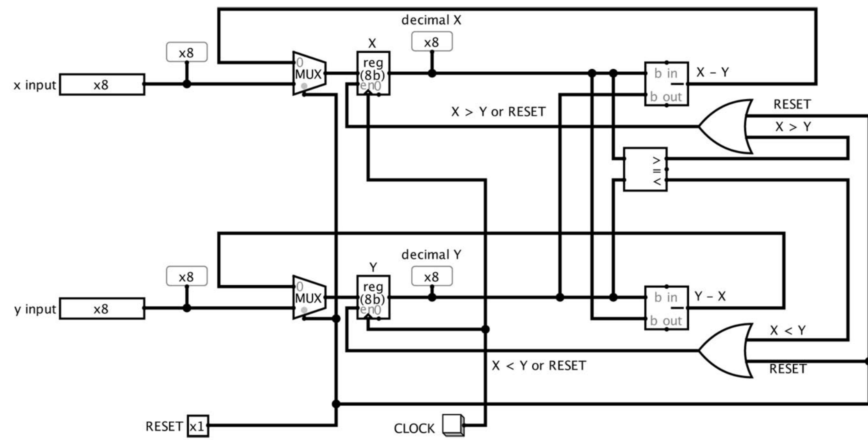
3. Things can get even worse. (In fact they were already). The updated comparator output may arrive before the updated output from the combination of the subtraction circuit and multiplexer that determines the input to the registers. In this case, the second register may actually load the wrong (a negative) value.

4. There is a simple design practice for avoiding such *race conditions*. In a *synchronous* sequential circuit, all registers have their clock inputs connected to the same signal. It won't arrive at all registers at precisely the same time because of differences in wire lengths, but it will be quite close.

5. This necessitates another way to arrange for some registers to change on a clock pulse when others stay the same.

   - This could be accomplished by placing a multiplexer in front of every register that would choose between a new value and the register's current value.
   - Better yet, real registers (and register in Logisim) usually include an enable input. The register only changes on a clock pulse if this input is 1.

6. A version of the GCD circuit that uses this approach is shown below:



decimal X
x8
x8
X
reg
(8b)
MUX
x input
x8
b in
X – Y
b out
X > Y or RESET
RESET
X > Y
decimal Y
x8
x8
Y
reg
(8b)
MUX
y input
x8
b in
Y – X
b out
X < Y
X < Y or RESET
RESET
RESET
x1
CLOCK