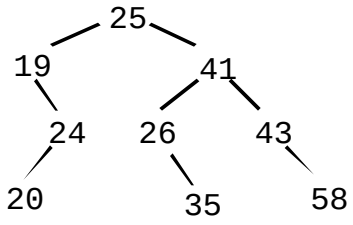
Practice Solution

Review Session: Sunday, June 4th, 7-9pm, 380-380C (Math corner of quad)
Final Exam: Tuesday, June 6th 12:15 –3:15pm Kresge Auditorium

Be sure to bring your text with you to the exam. We won't repeat the standard ADT definitions on the exam paper, so you'll want your text for references on whatever interfaces are needed.

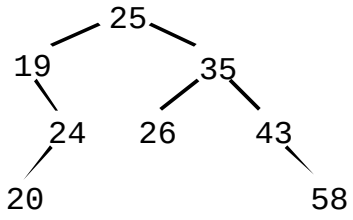
Problem 1: Short answer

a) After all insertions:



Pre-order traversal: **25 19 24 20 41 26 35 43 58**

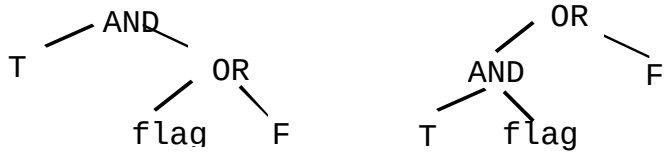
After deleting 41 (alternatively could shift up 43 and leave 35 where it was)



b) A polymorphic ADT is unrestricted in type (i.e. a stack that can store integers or strings or structures) and can be used in a variety of situations, encouraging code-reuse. However, it has significantly reduced type-checking—if we mistakenly enter a string into a stack of int pointers, the compiler cannot recognize our error, since string matches the void * prototype. Working with pointers also introduces more opportunity for memory management errors (incorrect casting, using unallocated storage, leaking memory, accessing freed data, etc.)

c) The variable **b** takes up 4 bytes. The added line is **`b.s = "winky";`**

d) The expression **`T AND flag OR F`** :can parse to either of these two trees:



One means to resolve the ambiguity is to force left or right associativity (without enforcing any precedence). Alternatively precedence could be introduced into the grammar or parsing to to bind AND or OR at a higher precedence than the other.

e) A long, skinny Huffman tree is indicative of a significant compression opportunity because some chars appear very frequently and others that are much more rare, allows to use fewer bits for the common characters at the expense of the rare ones. A bushy, balanced tree gives little or no compression since all chars have relatively equal frequency and all will use the same number of bits.

f)
```
int ModBy4(int num) {
    return (num & 3);  // mask off all but lower 2 bits, 3 is 0000011
 }
```

Problem 2: Function Pointers

a)
```
string RemoveAllMs(string word)
{
   string result;
   int i, pos;

   pos = 0;
   result = (string)GetBlock(strlen(word)+1);      // add one for null char
   for (i = 0; word[i] != '\0'; i++)
      if (tolower(word[i]) != 'm')
         result[pos++] = word[i];
   result[pos] = '\0';
   return result;
}
```

b)

Function typedef:
```
typedef bool (*predFn)(char);
```

Change function header:
```
string RemoveIf(string word, predFn pred)
```

Change test:
```
if (!pred(word[i]))
```

c) Create helper which takes additional boolean parameters and compares fn result to it:
```
void RemoveHelper(string word, predfn fn, bool toRemove)
{
      . . .
      if (fn(word[i]) != toRemove) {
            . . .
      }
      . . .
}

void RemoveIf(string word, predfn test) {
      RemoveHelper(word, test, TRUE);
}

void RemoveIfNot(string word, predfn test) {
      RemoveHelper(word, test, FALSE);
}
```

Problem 3: Binary Trees

```
bool IsSearchTree(tree t)
{
   return RecIsSearchTree(t, -10000, 10000);
}



bool RecIsSearchTree (tree t, int lower, int upper)
{
   if (t == NULL)
         return TRUE;
   if ((t->key > upper) || (t->key < lower))
         return FALSE;

   return (RecIsSearchTree (t->left, lower, t->key)
        && RecIsSearchTree (t->right, t->key, upper));
}
```

Problem 4: Recursion

```
static int CountFifteens(cardADT hand[], int n)
{
    return (CountSums(hand, n, 15));
}


static int CountSums(cardADT hand[], int n, int sum)
{
    if (sum == 0) return (1);
    if (sum < 0 || n == 0) return (0);
    return (CountSums(hand + 1, n - 1, sum)
            + CountSums(hand + 1, n - 1, sum - CardValue(hand[0])));
}

static int CardValue(cardADT card)
{
    switch (Rank(card)) {
      case Ace: return (1);
      case Jack: case Queen: case King: return (10);
      default: return (Rank(card));
    }
}
```

Problem 5: Parsing and expressions

```
 expressionADT SimplifyConstants(expressionADT exp)
 {
     expressionADT lhs, rhs;

     switch (ExpType(exp)) {
       case IntegerType: case IdentifierType:
         return (exp);
       case CompoundType:
         lhs = SimplifyConstants(ExpLHS(exp));
         rhs = SimplifyConstants(ExpRHS(exp));
         if (ExpType(lhs) == IntegerType
                && ExpType(rhs) == IntegerType) {
             return (NewIntegerExp(EvalExp(exp)));
         }
         return (NewCompoundExp(ExpOperator(exp), lhs, rhs));
     }
}
```

Problem 6: Using ADTs

This solution uses the visited flag of the node to track where we've been. We also could have used a set of visited nodes and tested the cur against membership in that set to avoid repeating ourselves.

```
  static bool PathExists(nodeADT start, nodeADT finish)
  {
      stackADT stack = NewStack();
      nodeADT cur, neighbor;

      foreach (cur in Nodes(GetGraph(start)))
          ClearVisitedFlag(cur);

      Push(stack, start);
      while (!IsStackEmpty(stack)) {
          cur = Pop(stack);
          if (cur == finish) break;
          if (!HasBeenVisited(cur)) {
              SetVisitedFlag(cur);
              Foreach (neighbor in ConnectedNodes(cur))
                  Push(stack, neighbor);
          }
      }
      FreeStack(stack);
      return (cur == finish);
  }
```

Problem 7: Designing ADTs

In many respects, this question is an essay question, even though the answer takes the form of a C interface. There are many ways to design the package, and the answers will be judged on how well the design choices are defended. Here is one possibility.

```
/*
 * File: molecule.h
 * ----------------
 * This interface provides a simple abstraction for managing
 * the relationships between atoms in a molecule.
 */

#ifndef _molecule_h
#define _molecule_h

#include "genlib.h"

/*
 * Type: moleculeADT
 * -----------------
 * A molecule is a set of atoms and their bonds.
 */
typedef struct moleculeCDT *moleculeADT;

/*
 * Type: atomADT
 * -------------
 * This type acts as an abstraction for the individual atom.
 */
typedef struct atomCDT *atomADT;

/*
 * Type: bondT
 * -----------
 * This type defines the possible bonds between two atoms.
 */
typedef enum { NoBond, SingleBond, DoubleBond, TripleBond } bondT;

/*
 * Type: atomFn
 * ------------
 * This type defines the space of functions that map over atoms.
 */
typedef void (*atomFn)(atomADT atom, void *clientData);

/*
 * Function: NewMolecule
 * Usage: m = NewMolecule();
 * -------------------------
 * This function returns a new empty molecule with no atoms.
 */
moleculeADT NewMolecule(void);

/*
 * FreeMolecule
 * Usage: FreeMolecule(m);
 * -----------------------
 * This function frees the storage for the molecule and its atoms.
 */
void FreeMolecule(moleculeADT m);

/*
 * Function: AddAtom
 * Usage: atom = AddAtom(m, symbol);
 * ---------------------------------
```

```
 * This function creates a new atomADT with the specified chemical
 * symbol and adds it to the molecule m, initially with no bonds.
 */
atomADT AddAtom(moleculeADT m, string label);

/*
 * Function: GetMolecule
 * Usage: m = GetMolecule(atom);
 * -----------------------------
 * This function returns the molecule of which a atom is a part.
 */
moleculeADT GetMolecule(atomADT atom);

/*
 * Function: ChemicalSymbol
 * Usage: symbol = ChemicalSymbol(atom);
 * -------------------------------------
 * This function returns the chemical symbol of a atom.
 */
string ChemicalSymbol(atomADT atom);

/*
 * Function: CreateBond
 * Usage: CreateBond(a1, a2, bond);
 * --------------------------------
 * This function adds a bond of the indicated type between
 * the atoms a1 to a2.
 */
void CreateBond(atomADT a1, atomADT a2, bondT bond);

/*
 * Function: GetBond
 * Usage: bond = GetBond(a1, a2);
 * ------------------------------
 * This function returns the bond type between a1 and a2.  If
 * there is no bond, the function returns the constant NoBond.
 */
bondT GetBond(atomADT a1, atomADT a2);


/*
 * Function: MapMolecule
 * Usage: MapMolecule(fn, m, clientData);
 * --------------------------------------
 * This function calls fn(atom, clientData) for each atom
 * in the molecule m.
 */
void MapMolecule(atomFn fn, moleculeADT m, void *clientData);

/*
 * Function: MapBonds
 * Usage: MapBonds(fn, a0, clientData);
 * ------------------------------------
 * This function calls fn(atom, clientData) for each atom
 * connected to a0 in the molecule.
 */
void MapBonds(atomFn fn, atomADT a0, void *clientData);

#endif
```

The interface for molecules is copied almost directly from the preliminary interface for graphs shown in Figure 16-2 on page 699 of the text, with a few additional functions to manage the specific structures required for this application. The implementation of this interface will also track that of the graph package, and you can use an adjacency list approach to keep track of the atoms bonded to any particular atom. The significant differences between the interfaces are as follows:

1. The bonds in the molecule are always bidirectional. Thus, if you keep the same implementation strategy, **`CreateBond`** would have to add both the connection from **`a1`** to **`a2`** and the connection from **`a2`** to **`a1`**.
2. You need to store the bond type with each bond. Thus, instead of a simple linked list of atoms, you would need to store a list of records containing the target atom and the bond type.