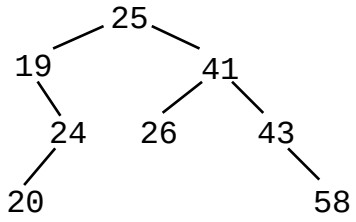
Practice Solution

Final exam: Monday, June 5th 8:30-11:30 am
Gates B01(next door to our usual room)

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 to refer to for whatever interfaces are needed.

Problem 1: Short answer

a)



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

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) Most Stanford phone numbers begin with the same two digits, thus they all hash to the same bucket. Having so many collisions means lookup will bog down when searching that particular bucket. The memory needs of the table don't change. You still have the same number of buckets (100) and the same number of cells (one for each customer), they are just arranged in one long list instead of many short lists.

d) Using **`InsertChar`** will be easier to implement, require less debugging, re-use code we already have written and work for all implementations without changes. An implementation-specific version can take advantage of internals to be more efficient.

e) Yes, but it will be slower -- computer search will do exhaustive traversal, following paths such as 'gtp' that don't lead to words without recognizing dead ends. However, the recursion isn't infinite since we hit our base case when all cubes are used.

f) If the pivot is the smallest or largest value, partitioning divides the array into an array of size 1 and one of size n-1. If this lopsided split happened at every level, QuickSort would run in $\mathbf{O(n^2)}$ time, its worst case. If using the first element as the pivot, data that is already sorted or reverse sorted will trigger this worst case performance. Although an element chosen at random does have a 2 in N chance of being the smallest or largest, the odds that we would repeatedly chose the worst element at every stage is exceedingly small.

g)
```
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: Recursion

```
static int ArithmeticCombinations(int arr[], int n, int result)
{
    return ACHelper(arr[0], arr + 1, n-1, result);
}

static int ACHelper(int sum, int arr[], int n, int result)
{
    if (n == 0)
        return (sum == result ? 1 : 0);
    return ACHelper(sum * arr[0], arr + 1, n-1, result)
         + ACHelper(sum + arr[0], arr + 1, n-1, result)
         + ACHelper(sum - arr[0], arr + 1, n-1, result);
}
```

Problem 4: 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 5: Designing and implementing ADTs

This implementation keeps each row in linked list, sorted by column order, it does not use a dummy header cell, but doing so would have meant a little more work to set up a new matrix and one less special case to handle in SetElement.

```
typedef struct _cell {
   int col;
   double val;
   struct _cell *next;
} Cell;

struct matrixCDT {
   Cell **rows;
   int numRows, numCols;
};


matrixADT NewMatrix(int numRows, int numCols)
{
   matrixADT m = New(matrixADT);
   int i;

   m->rows = NewArray(numRows, Cell *);
   for (i = 0; i < numRows; i++)
      m->rows[i] = NULL; // all rows start empty
   m->numRows = numRows;
   m->numCols = numCols;
   return m;
}

void SetElementAt(matrixADT m, double value, int row, int col)
{
   Cell *prev, *newOne;

   if (row < 0 || row >= m->numRows || col < 0 || col >= m->numCols)
      Error("SetElement out of bounds!");

   prev = FindPrevCell(m->rows[row], col);
   if (prev == NULL) {        // insert new cell at head of list
      newOne =  CreateCell(col, value);
      newOne->next = m->rows[row];
      m->rows[row] = newOne;
   } else if (prev->next != NULL && prev->next->col == col) {
      prev->next->value = value; // update existing cell to new val
   } else {             // insert new cell in middle of list
      newOne = CreateCell(col, value);
      newOne->next = prev->next;
      prev->next = newOne;
   }
}
```

```
// An internal helper to find the cell that would precede a given
// column in the list of cells for this row, this will be useful when
// trying to insert a new cell. It will return NULL of the col belongs
// at the head of the list
static Cell *FindPrevCell(Cell *head, int col)
{
   Cell *cur, *prev = NULL;

   for (cur = head; cur != NULL && cur->col < col; cur = cur->next)
      prev = cur;
   return prev;
}


// An internal helper to create a new cell
static Cell *CreateCell(int col, double value)
{
   Cell *newOne = New(Cell *);
   newOne->col = col;
   newOne->value = value;
   newOne->next = NULL;
   return newOne;
}


// In order to make this O(NR *NC) we have to walk the matrix in
// in order (ie we cannot just iterate through calling GetElementAt
// on each row/col combination, that would be O(NR*NC^2).
void PrintMatrix(matrixADT m)
{
   int row, col;
   Cell *cur;

   for (row = 0; row < m->numRows; row++) {
      cur = m->rows[row];
      for (col = 0; col < m->numCols; col++) {
         if (cur != NULL && cur->col == col) {
            printf("%g  ", cur->value);
            cur = cur->next;
         } else // no element in that col, assumed 0
            printf("0.0  ");
      }
      printf("\n");
   }
}
```

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);
}
```