Name: ______________________________ Student #: ____________________

Engineering 4892 — Data Structures
Quiz 2

Dr. D. K. Peters

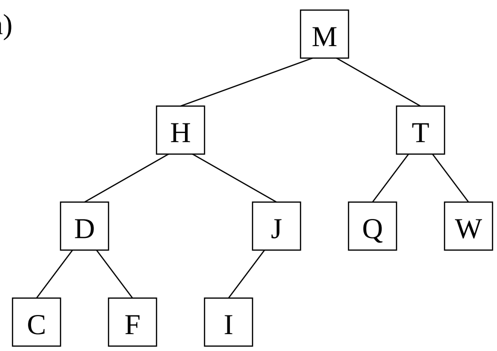
July 10, 2002

Instructions: Answer all questions. Write your answers on this paper. This is a closed book test, no textbooks, notes, calculators or other aides are permitted.
Total points: 50

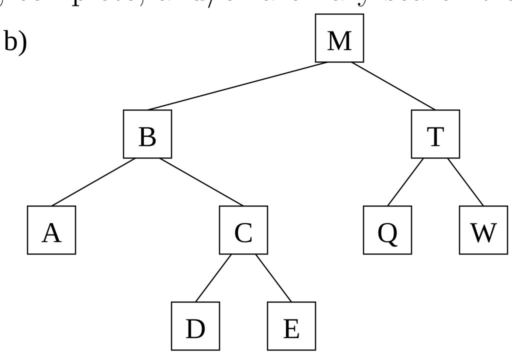
---

1. [12 points] For each of the following trees, fill in “true” or “false” in the appropriate cells of the table below to indicate if the tree is full, complete, and/or a binary search tree.

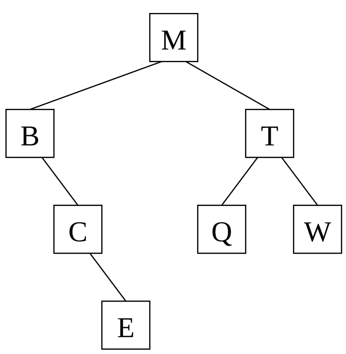
a)



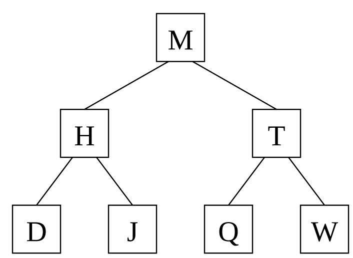
b)



c)



d)



	Full	Complete	Binary Search
(a)			
(b)			
(c)			
(d)			

2. [3 points] Consider a binary search tree that is initially empty. Using the insertion algorithm presented in class, which maintains the binary search tree property, but makes no effort to balance the tree, draw the tree that will result if the following numbers are inserted in the order given: 5, 8, 1, 7, 3, 2, 6, 9

Name: ______________________ Student #: ______________

3. [15 points] Consider a recursive function using backtracking to find a solution to the following puzzle. You are given two positive integers, `initial`, representing a number of marbles, and `steps`, representing the maximum number of moves permitted. The goal is to find a sequence of at most `steps` `Move`s so that you end up with 91 marbles. `Move` is defined as follows,

```
enum Move { More, GiveBack };
```

and these are interpreted as follows:

`More` — Request, and receive, 53 more marbles.

`GiveBack` — Give back half of the marbles you have. This move is only possible if the number of marbles you have is even.

If there is a solution, then `m` should contain the sequence of `Move`s that get from the initial value to 91. For example, if `initial` is 99 and `steps` is 4, then `m` should be { `More, GiveBack, GiveBack, More` }.

a) [1 points] What is (are) the base case(s) for the recursion?

b) [1 points] What is the variant expression for the recursion?

c) [13 points] Give an implementation of the function.

```
// Pre: initial >= 0 /\ steps >= 0
// Post: Result -> m is a sequece of moves to get to 91 from initial
//                  /\ |m| <= steps /\
//        !Result -> there is no sequence of moves to get to 91 from initial
//                  in less than steps steps
bool
goal(int initial, int steps, std::list<Move>& m)
{
```

4. [10 points] Consider the following partial implementation of a binary tree class. (Note: **Not** a binary *search* tree.) On the following page you are to implement the function `int rSize(BinaryNode<T>* r) const` that returns the number of nodes in the tree rooted at `r`.

```
template <class T>
class BinaryNode // The node of the binary tree
{
  public:
    T key;
    BinaryNode* left;
    BinaryNode* right;
```

```
    BinaryNode(BinaryNode *l = 0, BinaryNode* r = 0)
      : left(l), right(r) { }
    BinaryNode(const T& k, BinaryNode *l = 0, BinaryNode* r = 0)
      : key(k), left(l), right(r) { }
};

template <class T>
class BinaryTree
{
  public:
    enum Status { Ok, NewFail, NoSuchElement, DuplicateElement };
  // ...
    int size() const;
    // Post: result = number of nodes in the tree
    Status insert(const T& x);
    // Post: x is inserted in the tree keeping it optimally balanced

  protected:
    BinaryNode<T>* root;
    mutable Status err;   // Status of the last call.

  private:
    // ...
    int rSize(BinaryNode<T>* r) const;
    Status rInsert(BinaryNode<T>*& r, const T& x);

};

/*************************************************************
 * size -- return the number of nodes in the tree
 *************************************************************/
template <class T>
int
BinaryTree<T>::size() const
{
  return rSize(root);
}

/*************************************************************
 * insert -- insert an element in the tree, keeping it balanced
 *
 * Post:
 *************************************************************/
template <class T>
BinaryTree<T>::Status
```

```
BinaryTree<T>::insert(const T& x)
{
  return rInsert(root, x);
}
```

a) [7 points] Give the implementation for `rSize`.

```
template <class T>
int
BinaryTree<T>::rSize(BinaryNode<T>* r) const
{
```

b) [3 points] What is the time complexity of your algorithm? (Be sure to state what n represents).

5. [10 points] Consider the same partial binary tree implementation as given in question 4. In class I presented an implementation of `rInsert` that inserted in the shortest sub-tree. An alternative would be to choose to insert in the sub-tree containing the fewest nodes, or the left sub-tree if the sub-trees contain equal numbers of nodes.

a) [8 points] Give a *recursive* implementation of `rInsert` that inserts in the sub-tree containing the fewest nodes, or the left sub-tree if equal.

```
template <class T>
Status
BinaryTree<T>::rInsert(BinaryNode<T>* r, const T& x)
{
```

b) [2 points] Draw the tree that would result if the integers from 1 to 10 were inserted in increasing order into an initially empty tree using the above function.