NAME (print): _____

Honor Acknowledgment (signature): _____

DO NOT SPEND MORE THAN 10 MINUTES ON ANY OF THE OTHER QUESTIONS! If you don't see the solution to a problem right away, move on to another problem and come back to it later.

Before starting, make sure your test contains 11 pages.

If you think there is a syntax error, then ask. You may assume any include statements are provided.

	value	grade
Problem 1	12 pts.	
Problem 2	12 pts.	
Problem 3	16 pts.	
Problem 4	16 pts.	
Problem 5	12 pts.	
TOTAL:	68 pts.	

Whenever you see references to *TreeNode* in this test, the following definition is assumed.

```
template <class T>
struct TreeNode
{
    T info;
    TreeNode * left;
    TreeNode * right;

    TreeNode(T newInfo, TreeNode * newLeft=NULL, TreeNode * newRight=NULL);
};

TreeNode::TreeNode(T newInfo, TreeNode * newLeft, TreeNode * newRight)
    : left(newLeft), right(newRight), info(newInfo)
{}
```

```
//  class for simulating a die (object "rolled" to generate
//                              a random number)
//
//  Dice(int sides) -- constructor, sides specifies number of "sides"
//               for the die, e.g., 2 is a coin, 6 is a 'regular' die
//
//  int Roll() -- returns the random "roll" of the die, a uniformly
//                distributed random number between 1 and # sides
//
//  int NumSides() -- access function, returns # of sides
//
//  int NumRolls() -- access function, returns # of times Roll called
//                    for an instance of the class

// abstract base class for a Map class
// (sometimes called Table or Dictionary or Associative Array class)
//
// The class Map supports a mapping of Keys -> Values
// for example: string -> int
//
// Abstract functions
//
// bool includes(key)        -- returns true if key stored in map
//
//    use: if (map.includes("apple")) ...
//
// Value & getValue(key)        -- returns reference to Value associated
//                                 with key, error if key not in table
//                                 (const version exists too)
//
//    use: value = map.getValue("apple")
//
// getValue(pair);          -- returns a pair with given key
//                             (and associated value)
//    use: map.getValue(p);
//
// insert(Key,Value)           -- add new key,value pair
// insert(pair)                -- add new pair (with key) to map
//                             if key in map this is a no-op
//
//    use: map.insert(key,value);
//    use: map.insert(p);
//
// MakeIterator()           -- returns pointer to a usable iterator
//                             the iterator returns (Key,Value) pairs
//
```

PROBLEM 1 : (*It depends ...* (12 points))

Part A: (6 points)
Consider the following data structures:

- vector with data in random order
- binary search tree (not necessarily balanced)

and the following operations on the data structures:

- find(key) returns true if key is present, otherwise returns false
- insert(key) adds key to collection (at the end for vectors, in correct place for trees) if not already present

For each data structure, give the big-Oh expression that describes the worst case running time of the operation. You should "justify" your answer by drawing a picture of the state of the data structure that represents the worst case.

Part B: (3 points)

A map will be used to store student records for a large university. Once a year a new batch of students will be added to the map. But during the rest of the year, quick access to student's records will be of primary importance. Why is a binary search tree a better choice to implement the map than an unsorted vector?

Part C: (3 points)

A map will be used to catalogue the inventory of a pack-rat. New data will be added to the map frequently, but it will rarely be accessed again. Why is an unsorted vector a better choice to implement the map than a binary search tree?

PROBLEM 2 : (*Search and ye shall find.* (12 points))

Part A (8 points)

Draw the single binary search tree from which the following traversals result:

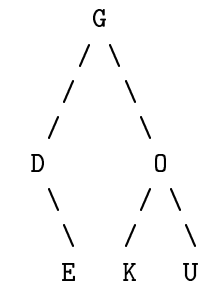
```
pre:  3 1 0 2 7 5 4 6 8

in:   0 1 2 3 4 5 6 7 8

post: 0 2 1 4 6 5 8 7 3
```

Part B (4 points)

Give a possible order of insertion that would result in the following binary search tree:



PROBLEM 3 : (*Foul is foul* (16 points))

Part A (8 points)

Write the function *SuitableForViewers* that scans a TV program script for "profane" words. The words in the script are organized in a binary tree; your function should check the tree to see if it contains any profane words. Since if even one word is considered profane, the script is sent back to be edited, your function can stop after it finds one.

Because what is profane changes often, the function *IsProfane* is provided for your convenience to determine if a given word is acceptable. You do **not** have to implement it.

```
bool IsProfane(const string & word)
// post: return true if word is not suitable for young viewers,
//       false otherwise
{
   // assume implemented correctly
}

bool SuitableForViewers(TreeNode<string> * t)
// post: returns true is t contains no profane words,
//       false otherwise
{
```

Part B (8 points)

Hollywood programmers like to play tricks. One of their favorite is to print a copy of a TV program script with the order of some of the words, phrases, or sentences reversed. To accomplish this trick, they randomly change the word's order while making a copy of the script.

Write the function *TrickCopy* that makes a copy of a TV script but randomly switches the order of the children. In other words, each time a tree node is copied, it is equally likely the children will be in the correct order as in reverse order (i.e., left as right and right as left).

You should use the *Dice* class to generate random values, let a roll of zero cause a proper copy of the sub-tree to be made and a roll of one copies the node but reverses it's children.

```
TreeNode<string> * TrickCopy(TreeNode<string> * t)
// post: returns a copy of t, possibly with some of its links reversed
{
```

PROBLEM 4 : (*eeorrst* (16 points))

In this problem the functions `VectorToTree` and `VectorToTreeII` convert a sorted vector into a binary search tree (different algorithms are used in the different functions). Assume that the function `Insert` correctly inserts a value into a binary search tree (we went over this function in class).

```
TreeNode * VectorToTree(const Vector<string> & a, int first, int last)
// precondition: a[first] <= a[first+1] <= ... <= a[last]
// postcondition: returns balanced search tree containing a[first]...a[last]
{
    if (first <= last)
    {
        int mid = (first + last)/2;

        return new TreeNode(a[mid], VectorToTree(a,first,mid-1),
                        VectorToTree(a,mid+1,last));
    }
    return NULL;
}
```

Part A: (5 points)

Note that in both cases the vector `a` is sorted.

What is the complexity (using big-Oh) of the function `VectorToTree` for an n element vector? Briefly justify your answer.

```
TreeNode * VectorToTreeII(const Vector<string> & a, int first, int last)
// precondition: a[first] <= a[first+1] <= ... <= a[last]
// postcondition: returns search tree containing a[first]...a[last]
{
    TreeNode * t = NULL;
    int k;
    for (k=first; k <= last; k++)
    {
        Insert(t,a[k]);
    }
    return t;
}
```

Part B: (5 points)

What is the complexity of the function `VectorToTreeII` assuming that `Insert` does NOT do any re-balancing? Briefly justify your answer.

Part C: (6 points)

Does the answer to either question above change if the basic algorithm remains the same but the parameter `a` is a linked-list instead of a vector? Briefly justify your answer.

PROBLEM 5 : (*2, 4, 6, 8, I don't want to evaluate* (12 points))

For this problem, you will add variables to the expression trees described in class so that, in addition to combining constants, a tree may contain variables whose values can change between evaluations of the expression. To implement these variables, you will build a class that extends an expression tree node to access a map that associates variable names with a value.

Below is an example program that show a variable, *k*. node:

```
int main()
{
   UVMap<string, int> variables;
   variables.insert("k", 24);

   Node * exp = new Times(new Variable("k", variables),
                          new Number(2));

   cout << exp << " = " << exp->evaluate() << endl;
   variables.getValue("k") = 13;
   cout << exp << " = " << exp->evaluate() << endl;

   return EXIT_SUCCESS;
}
```

which generates the following output:

```
   k * 2 = 48
   k * 2 = 26
```

Part A: (4 points)

Show the new output if the following lines are added to the end of the example program above:

```
   variables.getValue("k") = 32;
   cout << exp << " = " << exp->evaluate() << endl;
```

Show the new output if the operator node in the expression tree, *exp*, is changed from *Times* to *Divide*:

Part B: (8 points)

Recall that all nodes in the expression tree derive from a common superclass, *Node*, given below:

```
class Node
{
  public:
    virtual ~Node() {}
    virtual int evaluate() = 0;
    virtual void print(ostream & out) = 0;
};
```

Write a class *Variable* that inherits from *Node* and redefines the virtual member functions *print* and *evaluate* so they function to produce the output shown in the example program (as specified in their respective postconditions). The constructor has been written for you.

```
class Variable : public Node
{
  public:
    Variable(const string & name, const Map<string, int> & values)
      : myName(name), myValue(values)
    {}

    virtual int evaluate()
    // post: returns value of variable as retrieved from map
    {




    }

    virtual void print(ostream & out)
    // post: name of variable is inserted into out
    {




    }

  private:
    string myName;
    Map<string, int> & myValue;
};
```

If the map declared in the function main is changed from a `UVMap<string, int>` to a `BSTMap<string, int>` the code written for the class Variable will not need to change. Briefly explain why.