Question 1 (3 Marks)

Catalan numbers are defined for non-negative integers as follows:

```
Catalan(0) = 1
Catalan(n) = Catalan(n-1)*(4n+2)/(n+1), for n>0
```

Write a **recursive** function that calculates `Catalan(n)`. Make sure that your function does something reasonable for **all** integer arguments.

Question 2 (4 Marks)

Write a function, `getMean`, which given the `head` pointer to a linked list of integers and the `Node` class defined below, returns the **mean** of the values stored in the list. Note that the mean (sometimes called the average) is defined to be the sum of the integers divided by the number of integers in the list. If the list is empty, your function should throw an exception. Your function may be iterative **or** recursive. If you choose to call any other functions, you must also write their implementations.

```
class Node {
   public:
      int value;
      Node *next;
}
```

Question 3 (15 Marks)

a) Assuming that we start with an empty stack of integers, `s`, give the output of the following code fragment: (2 marks)

```
s.push(4);
s.push(5);
cout << s.pop() << endl;
s.push(12);
s.push(3);
s.pop();
cout << s.pop() << endl;
```

b) i) Assuming that we start with an empty sorted binary tree, `t`. Draw the resulting tree after the following statements are executed. Note that this is a regular binary tree, **not** a self-balancing tree (i.e. no rotations will be performed). (3 marks)

```
t.insert(10);
t.insert(5);
t.insert(8);
t.insert(12);
```

ii) Redraw your tree after the following statement is executed. (2 marks)

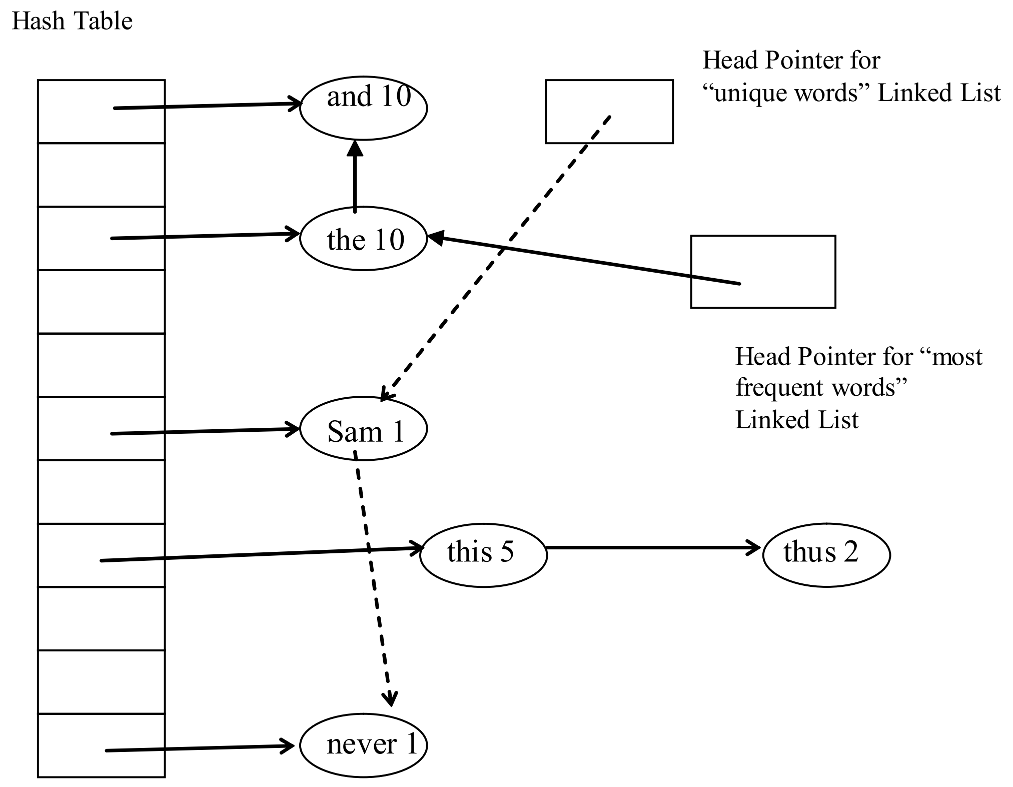
```
t.delete(10);
```

iii) Redo part i) assuming that the sorted binary tree is **self-balancing** (i.e. rotations will be performed during insert, if necessary). Draw a tree for each step, i.e. one tree for every insertion that doesn't involve a rotation, and two for each insertion that does involve a rotation (one showing the tree before the insertion, and the other showing the tree after the insertion). For each tree, give the balance factor of every node. For each rotation, indicate the type of rotation performed (i.e. LL, LR, RL, RR). (5 marks)

c) Draw the expression tree for the following: (((10*5)+(3-2))/7) (3 marks)

Question 4 (14 Marks)

An English Professor wishes to study the books written by a famous author to see how often various words appear in the text. The Professor is going to store this information in a data structure and as she needs to be able to find each word quickly, hashing is appropriate. For this application we have chosen to use **hashing with buckets**. In addition to finding the words quickly, the Professor also needs to be able to access the most commonly used words and those that appear only once in the text. Thus, along with our hash table, we have two linked lists: one contains all words that appear just once in the text, and the other contains the words that appear most often. See the example below. Note that some words will not be in either linked list. The lists are shown as singly linked lists below to keep the diagram simple, but they are actually **doubly linked lists** (each with a head and tail pointer).


Hash Table
Head Pointer for "unique words" Linked List
and 10
the 10
Head Pointer for "most frequent words" Linked List
Sam 1
this 5
thus 2
never 1

Here's a partial header file for this class:

```
class Concordance {
   private:
      class HNode {
         public:
            // this needs to be completed
      };

      HNode **table; // dynamically allocated array of pointers to HNodes
      HNode *uniqueWordsHead, *uniqueWordsTail;
      HNode *mostFrequentHead, *mostFrequentTail;
      int tableSize, count;

      int hash(const string &name); // returns the appropriate hash key

      // inserts the given node at the front of the unique words
      // doubly linked list, as the word appears just once in the text
      void insertUniqueWords(HNode *node);

      // removes the given node from the unique words doubly linked list,
      // as the word is now used more than once in the text
      void removeUniqueWords(HNode *node);

      // if this node belongs in the most frequent doubly linked
      // list, makes the necessary adjustments to that list
      void checkIfMostFrequent(HNode *node);

   public:
      Concordance(); // creates an empty hash table

      // Adds the occurrence of a word to the hash table.
      // - The words in each bucket are stored in alphabetical order.
      // - If it is a new word, the word is added to the appropriate bucket
      // of the hash table (see above point) with a frequency of 1, and the
      // node is also added to the front of the unique words doubly linked
      // list (i.e. uniqueWordsHead will point to this node).
      // - If it is an existing word the frequency is incremented.  If
      // required, the node is removed from the unique words list.
      // - If the word becomes the most or one of the most frequently used
      // words in the text (i.e. it now has the highest or tied for the
      // highest frequency), the most frequent linked list is adjusted.
      void insert(const string &name);
};
```

a) Complete the public portion of class `HNode`. Remember that you are using **hashing with buckets** along with **two doubly linked lists**. (3 marks)

b) Write the implementation of the `insert` method exactly as it would appear in `Concordance.cpp`. Hint: Your implementation should call all four methods defined in the private section (`hash`, `insertUniqueWords`, `removeUniqueWords`, and `checkIfMostFrequent`), but you do **not** need to write them here. If you choose to use any other methods or functions, you must also write their implementations. (7 marks)

c) Write the implementation of the `removeUniqueWords` private member method exactly as it would appear in `Concordance.cpp`. If you choose to use any other methods or functions, you must also write their implementations. (4 marks)

Question 5 (10 marks)

Given the following grammar:

<bool> ::= **true** | **false**
<binop> ::= **&&** | **||**
<exp> ::= **(** <goal> <binop> <goal> **)** | **(!** <goal> **)**
<goal> ::= <exp> | <bool>

a) Give a short description using examples and words as to the strings accepted by this grammar. (3 marks)

b) Draw the parse tree for the string "(true&&((!true)||false))". (3 marks)

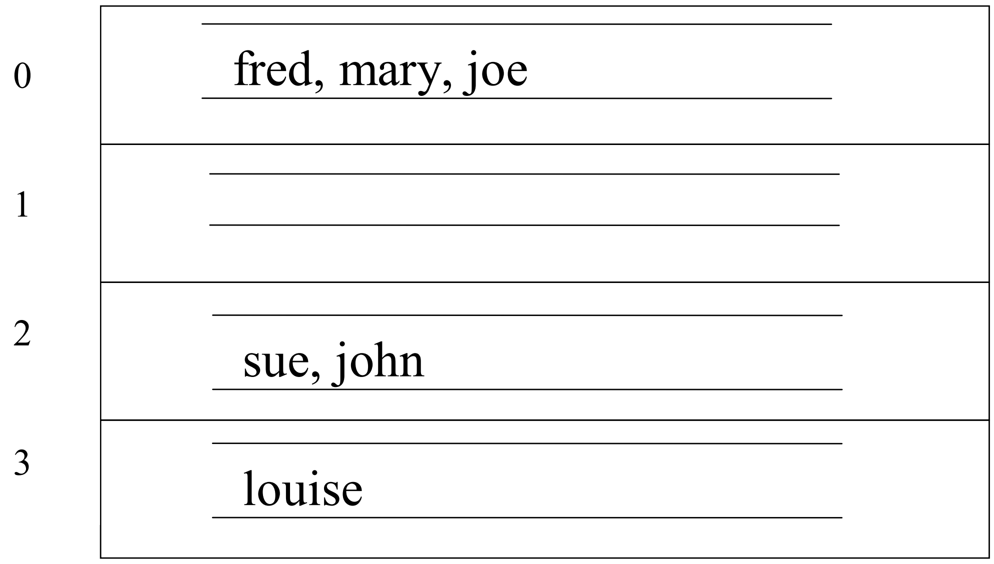
c) Assuming the usual "iterator" methods (`iterator.get()` and `iterator.look()`), and that somebody else has written "`recognizeGoal`" and "`recognizeBinop`", write method "`recognizeExp`". Have your method return `true` on success and `false` on failure. (4 marks)

Question 6 (14 Marks)

In this question you are implementing a class that models customers lining up and paying for their purchases at a grocery store. A grocery store has a number of lines, called "checkouts". Each one of these is a queue (FIFO). When a customer is ready to pay, he or she joins the checkout with the shortest queue.

For example, in this grocery store (below) we have four checkouts:

- checkout[0] has 3 customers: fred, mary and joe
- checkout[1] is empty (no customers)
- checkout[2] has 2 customers: sue and john
- checkout[3] has 1 customer: louise



The next customer would join the queue for checkout[1] (as it is the shortest).

Note that we do not know (or care) how the queues are implemented (e.g. they could be linked lists or arrays), so we have used our general queue notation (enqueue at the right, dequeue at the left).

Here's a partial header file for our Grocery Store:

```
class GroceryStore {
   private:
      // a dynamically allocated array of checkout queues (one queue per
      // checkout line).  Each queue is made up of customer names,
      // represented by strings.
      Queue<string> *checkout;

      int numCheckouts; // the number of checkouts

      // returns the array index of the checkout with the shortest queue.
      // (The shortest queue is the one with the fewest customers.)
      // If there is a tie for shortest queue, then, of the tied
      // checkouts, it returns the checkout with the lowest index.
      // For example, if checkout[1] and checkout[2] were the only two
      // empty queues, this method would return 1.
      int findShortestQueue();

   public:
      // creates a grocery store with "number" checkouts.  Throws an
      // invalid_argument exception if "number" is less than 1.
      GroceryStore(int number);

      // adds the given customer to the shortest available queue.
      void lineUp(const string &name);

      // serves the customer at the front of the given queue (i.e.
      // this customer buys his groceries and is removed from the queue).
      // Prints a message giving queue number and the customer name.
      // Throws an exception if this queue does not exist or is empty.
      void serve(int queue);
};
```

Here's everything that you need to know about the Queue template class:

```
    template <class T>
    class Queue {
    public:
        Queue (); // creates queue with infinite capacity
        void enqueue (T item); // adds item to the end of the queue
        T dequeue ();  // removes and returns item from the front of the
                       // queue.  Throws "overflow_error" if queue is empty
        int size ();   // returns number of items in queue
        T look ();     // as for dequeue but item is left in queue
   };
```

a) Write the `GroceryStore` constructor exactly as it would appear in `GroceryStore.cpp`. (3 marks)

b) Write the `findShortestQueue` private member method exactly as it would appear in `GroceryStore.cpp`. (5 marks)

c) Write the `lineUp` method exactly as it would appear in `GroceryStore.cpp`. Hint: This method should use `findShortestQueue`. (3 marks)

d) Give the output of this application program code fragment: (3 marks)

```
GroceryStore gs(3);
gs.lineUp("dan");
gs.lineUp("siva");
gs.lineUp("joe");
gs.serve(1);
gs.lineUp("lynn");
gs.lineUp("fran");
gs.serve(2);
gs.serve(0);
gs.serve(1);
```

Question 7 (2 Bonus Marks)

Identify the programming language pioneer who died in March 2007, as well as (at least) one of his claims to fame.