**(7)** [3+17=20 points]

In this problem, you will be implementing a data structure for a "forgetful brain". The way a forgetful brain works is as follows: it has a fixed and limited capacity for facts (which for our purposes are just strings). Initially, your brain is empty. As you learn more facts, they are added to the brain. When the brain is full, any newly added fact displaces one that was previously there, meaning that you forget the previous fact. Which fact gets displaced? The one that was used least recently. There are two ways in which your brain can use a fact: (1) Learning a new fact is using it; that is, you remember the things you learned recently. (2) You can deliberately recall a fact.

As an example, suppose that the brain has a capacity of 3 facts, and you learn A, B, C in order. Then, you recall A, and then you learn D. At this point, B is the least recently used fact, so you forget B in order to learn D. When you learn E, you next forget C, and if you next learn F, you forget A. If instead, you recalled A again before learning F, then you would next forget D. The `Brain` class thus looks as follows:

```
class Brain
{
  public:
    Brain (int capacity);
      // create a new Brain with the given fixed capacity.
    void remember (const string & fact);
      /* access the fact, i.e., mark it as freshly remembered.
         We will never ask you to remember a fact that you haven't learned. */
    void learn (const string & fact);
      /* add the given fact to the brain, and mark it as freshly remembered.
         If the brain is full, throw out the least recently used fact
         to make room for the newly added fact. */
};
```

In order to implement this `Brain` class, you should use the following modified version of a `List`, which a friend has already written for you, and which you cannot modify. We guarantee that you will never be asked to learn the same fact twice, so you don't need to worry about duplicates in your brain or list.

Your friend's modified list class is called `LimitedList`; it's basically a `List` which will never resize, even if you reached the capacity. Instead, it will throw an exception if you exceed its capacity. Here is your friend's header file.

```
class LimitedList {
  public:
    LimitedList (int capacity);
     /* creates a list fixed to this capacity. It can still grow and
        shrink with insert/remove, but if an insert would make the
        size exceed the capacity, it will throw an exception. */

    void set (int i, const string & item); // exactly the same as standard set
    const string & get (int i) const; // exactly the same as standard get
    void insert (int i, const string & item);
      /* almost the same as standard insert, except if the list is
         full, it will throw an exception rather than resizing.
         In the case of the exception being thrown, it will not alter
         the list. */
    void remove (int i); // exactly the same as standard remove

  protected:
    int find (const string & item) const;
      /* returns the first location at which item is stored in the
         list. Returns -1 if the item isn't in the list. */
    int size () const;
      // returns the number of items currently stored in the list

  private:
    // the actual variables used to store stuff
};
```

(a) How should you use `LimitedList` to build `Brain`? Inheritance (if so, what type), composition, or other? Why?

(b) Give an implementation of the `Brain` class, by adding your code in the following piece of code.

```
class Brain
// relevant code here if you want
{
  public:
    Brain (int capacity)
     {









      }

    void remember (const string & fact) {










      }

    void learn (const string & fact) {










      }

    private:   // any data or methods you would like to add





};
```