CS106B Practice Final #2

This is an open-note, open-book exam. You can refer to any course handouts, textbooks, handwritten lecture notes, and printouts of any code relevant to any CS106B assignment. You may not use any laptops, cell phones, or internet devices of any sort. You will be graded on functionality—but good style helps graders understand what you were attempting. You do not need to #include any libraries and you do not need to forward declare any functions. You have 3 hours. We hope this exam is an exciting journey.

Last Name: _____

First Name: _____

Section Leader: _____

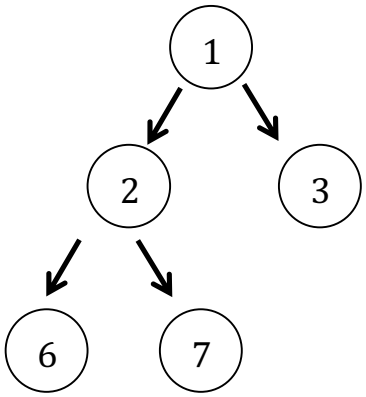
I accept the letter and spirit of the honor code. I've neither given nor received aid on this exam. I pledge to write more neatly than I ever have in my entire life.

(signed) _____

		Score	Grader
1. Short Answer	[10]	_____	_____
2. Rubik's Cube	[10]	_____	_____
3. Sparse Grid	[10]	_____	_____
4. Salty Passwords	[10]	_____	_____
5. Narcissistic Trees	[10]	_____	_____
6. Seam Carving	[12]	_____	_____
7. Eenvoudig Netwerk [1]	[8]	_____	_____
Total	**[70]**	**_____**	

[1] This problem really is a challenge.

Problem 1: Short Answer (10 points)



[a] Is the tree above a binary search tree? _____

[b] What is the output of the following code if passed in the root of the tree above?

```
void explore(Tree * tree) {
    if(tree == NULL) return;
    explore(tree->left);
    explore(tree->right);
    cout << tree->value << ", ";
}
```

Output: _____

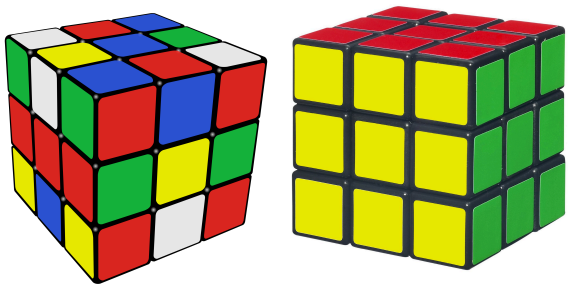
[c] If N is the number of nodes in the tree, what is the bigO of explore? _____

[d] Fill in the array so that it is a binary heap array (as in the heap in pQueue) that corresponds to the tree above:

0	1	2	3	4

Problem 1: Rubik's Cube (10 points)

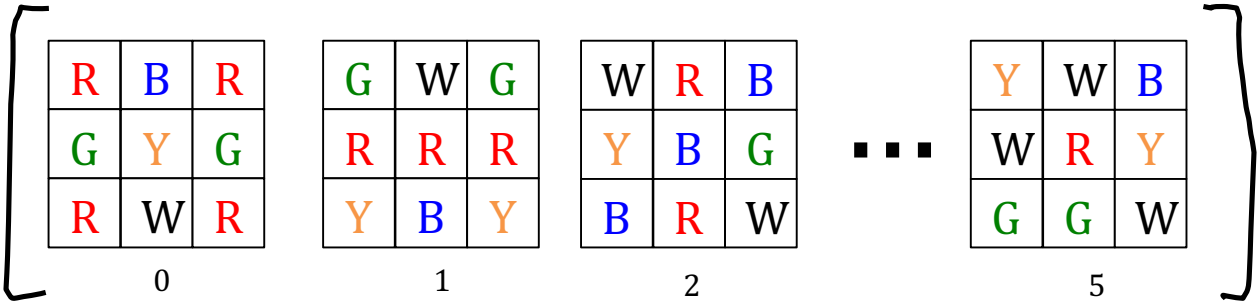
Rubik's cubes can be hard for humans to solve. Using the power of recursion we can program a computer to help us. Write a function that checks whether or not a Rubik's cube is solvable.



← Solved Rubik's cube

A Rubik's cube is a 3D cube with 6 faces each of which has a 3x3 grid of colors. The goal of a Rubik's cube is to make each face a homogenous color. Here is a depiction of a Rubik's cube faces vector (it corresponds to the Rubik's cube above-left):

`faces:`



For any cube, the legal moves are to rotate one of the 6 faces either clockwise or counter clockwise. In total there are 12 different actions you can take for any given cube. We provide a helper method that rotates a face of a cube (passed by reference) for you. You do not need to know how the **`rotate`** function is implemented. It does modify the cube. We also provide a helper method **`isSolution`** which returns true, if all faces are homogenous color.

```
void rotate(Cube & cube, int faceIndex, bool clockwise);
bool isSolution(Cube & cube);
```

When searching for a Rubik's cube solution, you want to avoid repeating a search over the same cube configuration. Use a CubeSet "seen" to keep track of which states you have already visited. A CubeSet has two methods:

```
seen.add(Cube& cube);       // adds a cube to the set
seen.contains(Cube & cube); // checks if the cube has been added.
```

```
bool isSolvable(Cube & cube, CubeSet & seen) {
```

Problem 2: SparseGrid (10 points)

Sometimes you want to store a gigantic grid. One so big it couldn't fit in memory. In the special case where most of your values are 0 you can use a "SparseGrid."

Netflix famously uses a sparseGrid when calculating movie recommendations

Your job is to implement a **`SparseGrid`** class. From the users point of view it should behave just like a normal grid however, under the hood it should use much less memory. Specifically you should implement a constructor that takes in numRows and numCols as parameters and you should implement a **`setValue`** method and **`getValue`** method that set and get double values.

The central requirement is that the amount of memory used should be **proportional to the number of elements with non-zero values**, not numRows x numCols.

To satisfy this requirement store all non-zero values in a Map that associates (row, col) pairs with the value in that grid-cell. If a particular (row, col) pairs is not in the Map, its cell has value 0. There are several ways to make a (row, col) pair a key. One way is to write a function that turns a (row, col) pair into a string and to use the resulting string as your key.

If a user calls **`getValue`** and passes in a (row, col) that hasn't been set, you should return 0.
If a user passes in a (row, col) pair that is out of bounds you should throw "Out of bounds."

Fill in the private section of the SparseGrid header file:

SparseGrid.h

```
class SparseGrid {
public:

   /* Constructor
    * -----------
    * Creates a new SparseGrid with dimension numRows, numCols.
    * Initially all values are 0.
    */
    SparseGrid(int numRows, int numCols);

   /* Set Value
    * -----------
    * Sets the value at location row, col. Throws an error if (row, col)
    * are out of bounds.
    */
    void setValue(int row, int col, double value);

   /* Set Value
    * -----------
    * Gets the value at location row, col. Throws an error if (row, col)
    * are out of bounds. The default value for all locations is 0.
    */
    double getValue(int row, int col);

private:
```

Write the SparseGrid source file:

SparseGrid.cpp

```
SparseGrid::SparseGrid(int numRows, int numCols) {
```

```
void SparseGrid::setValue(int row, int col, double value) {
```

```
double SparseGrid::getValue(int row, int col) {
```

Problem 4: Salty Passwords (10 points)

Database break-ins have become common. If you run a website with usernames and passwords, you are responsible for preventing a user's password from becoming known. Thus it has become a convention that you should never store raw-passwords in a database. Instead people use hash values of passwords.

Your job is to implement two functions:

```
void addUser(string user, string password, Map<string,string>&
database);
bool logIn(string user, string password, Map<string,string> &database);
```

Such that you never have to store raw-passwords

When a user creates a new account, **`addUser`** is called. You must save their username/password combination in the dataset so that you can later respond to **`logIn`** commands. However you must never put their "raw" password in the database (the password parameter for both addUser and logIn is a raw password).

Instead use the industry standard: salt and hash the password. To salt a password simply add a fixed string to the end of it. The one we will use is:

```
static const string SALT = "cs106brocks!";
```

After you have salted the password hash the salted result. The hash-value is safe to save in your database. We provide three hash-functions. Chose which one you would like to use (they aren't all correct).

Implement **`logIn`** so that it is compatible with your **`addUser`** command. If the raw-password passed in corresponds to what you have in your database for that user, return true. Otherwise return false. If the user doesn't exist you should return false.

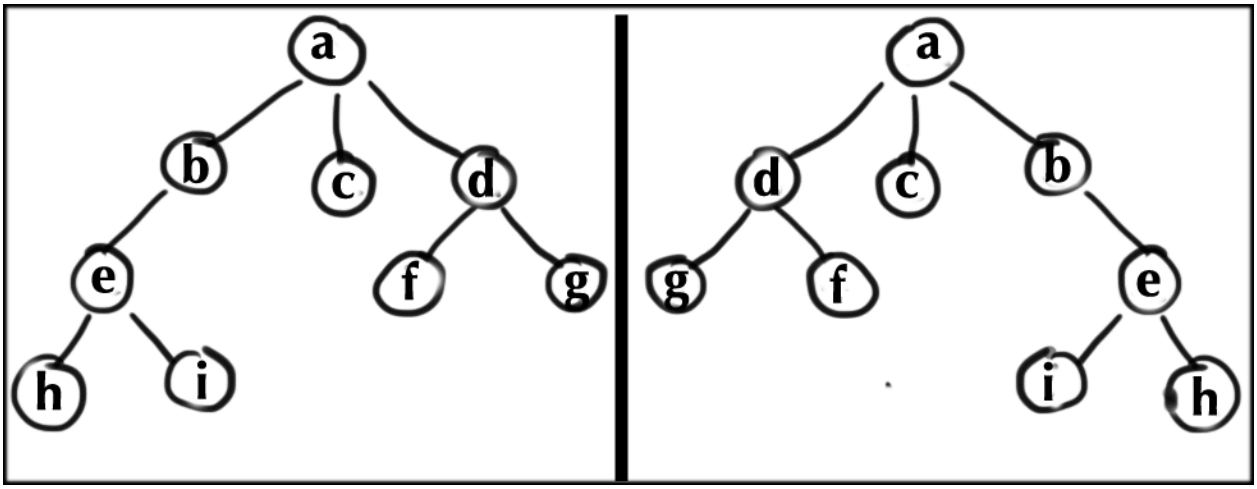
```
// You can use this hash function.
int hashSum(string input) {
   int sum = 0;
   for(int i = 0; i < input.length(); i++) {
      sum += pow(31, i) * input[i];
   }
   return sum;
}

// Or this one.
int hashRandom(string input) {
   return randomInteger(0, MAX_INT) * hash0(input);
}

// Or this one.
int hashLength(string input) {
   return input.length();
}


void addUser(string user, string password, Map<string,string>& db) {









bool logIn(string user, string password, Map<string,string>& db) {
```

Problem 5: Narcissistic Trees (10 points)

Did you know that trees like to look at themselves in the mirror? They're vain as the rest of us! Especially binary search trees… binary search trees think very highly of their looks.

Your task is to write a function **`reflect`** that takes an N-ary tree (i.e. arbitrary number of children) as input and returns a new N-ary tree—with all its nodes freshly allocated on the heap—that is the horizontal mirror reflection of the input.

Here's an example. Say you are given the tree on the left as input. The output of the function should be the tree on the right (newly allocated). In this example, the resulting image has a strong resemblance to a butterfly:



During the reflection a reflected-copy of the leftmost child from the input becomes the rightmost child in the output. A reflected-copy of the second leftmost child in the input becomes the second rightmost child in the output etc. Most importantly, the reflection happens through the entire tree, not just the first level of children.

Using the following definition of a tree, construct your solution

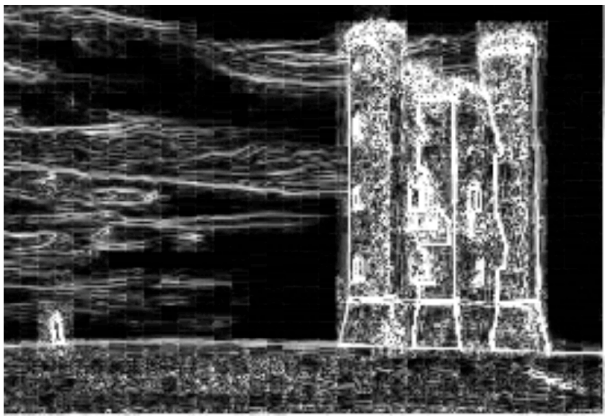
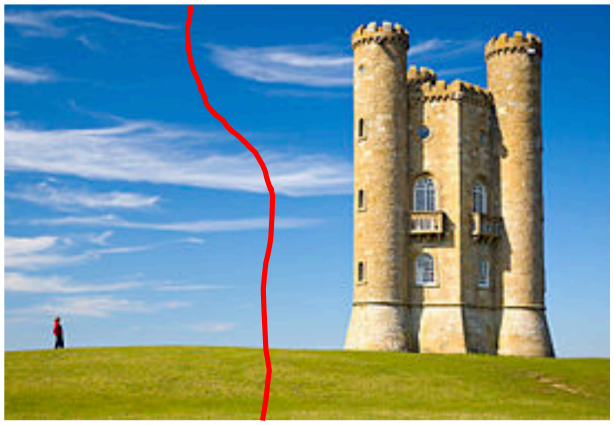
```
struct Tree{
    char value;
    Vector<Tree *> children
};

Tree * reflect(Tree * original) {
```

Problem 6: Seam Carving (12 points)

In this problem you will reduce the width of a picture by a single pixel without affecting how the image looks.


Consider the original image (on the left). It we want to resize it to a smaller image we could scale the image. However this would make the castle and person look distorted. Instead to preserve the castle and the person we have removed unimportant "seams" using Seam Carving.

Seam Carving reduces the width of an image pixel by pixel by repeating the process of removing the least important "seam" from a photo until the image is the desired width. A seam is a path of connected pixels from the top of the image to the bottom with exactly one pixel per row. A pixel is connected to its 8 direct neighbors.

The "importance" of a seam is the sum of the "importance" of its pixels. On the left is a visualization of pixel importance. On the right the least important seam is shown.

Write a function:

```
Vector<Pixel *> getLeastImportantSeam(Grid<Pixel *> & image)
```

That returns the *single* path of pixels that should be removed next. Hint: use graph search.

Each pixel is represented using the following struct:

```
struct Pixel{
    int row;         // the pixels row in the image
    int col;         // the pixels col in the image
    int importance;  // how substantial the pixel is to the picture
};
```

The image is a grid of pointers to pixels (already allocated) with their importance, row and col already set. The path you return is a Vector of pixel pointers from the image. The top of the image has `row == 0` and the bottom has `row == image.numRows() - 1`.

```
Vector<Pixel *> getBestSeam(Grid<Pixel *> & image) {
```

Problem 7: Eenvoudig Netwerk (8 points)

Heads up: this problem is very hard and not worth very many points. Make sure you have finished the rest of the exam before attempting it.

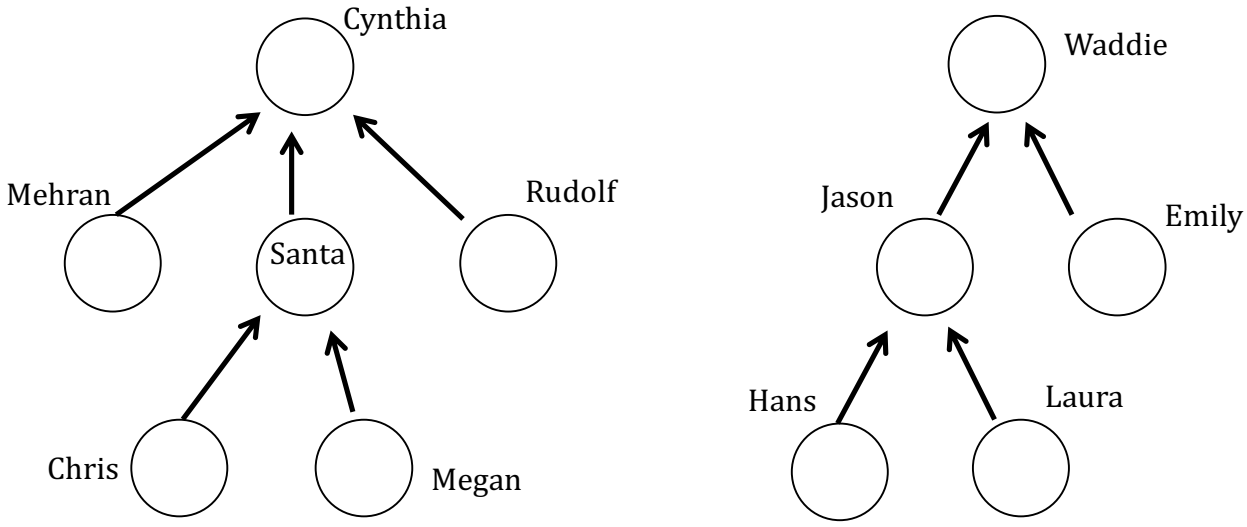
Eenvoudig Netwerk is a new Dutch minimalist social network that provides three functions.

1. **Nieuw**: Enter a username. The user exists.
2. **Betonen**: Enter two usernames. The users are now friends.
3. **Aangesloten**: Enter two usernames. The network returns true if there is a chain of user friendships that connect the users.

However, all commands **must run in O(log N)** time in the worst case. Write a class that implements all three functions. Your Nieuw, Betonen, and Aangesloten commands must be O(log N) to get credit.

Hint: you will need a non standard datastructure. Try thinking of each connected-set of users in the network as a tree, where nodes have pointers to their parents as opposed to parents having pointers to their children (the root of the tree would have a NULL pointer). That way, you can test if two nodes are in the same connected-set by following their parent pointers to the root, and testing if the two nodes have the same root. When connecting users that were previously unconnected you will have to merge their respective trees.

Here is a picture of such a datastructure:



You can quickly tell that Chris and Rudolf are connected – they share the same root: Cynthia. Chris and Emily are not connected. Chris has Cynthia as his root. Emily has Waddie.

When merging trees, make the smaller tree's root point to the larger tree's root.

Fill in the private section of the `Netwerk` header file:

Netwerk.h

```
class Netwerk {
public:

   /* Constructor
    * -----------
    * There are no users.
    */
    Netwerk();


   /* Nieuw
    * -----------
    * Adds a user. Now that user exists. They have no friends.
    */
    void nieuw(string user);


   /* Betonen
    * -----------
    * Connects two users. They are now friends.
    */
    void betonen(string userA, string userB);


   /* Aangesloten
    * -----------
    * Returns true if there is a chain of friendships between the users.
    */
    bool aangesloten (string userA, string userB);

private:
```

Write the Network source file:

`Netwerk.cpp`