CS106B Midterm Exam #2

This midterm exam is open-book, open-note, but closed-computer. Answer each of the questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit. In all questions, you may include functions, classes, or other definitions that have been developed in the course giving the name of the function or class and the handout, chapter number, or lecture in which that definition appears. You do not need to prototype helper functions before using them, nor do you need to **`#include`** any Stanford library headers.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do. You do not need to worry about efficiency unless a problem specifically requires an efficient solution.

SUNetID: ______________________

Last Name: ______________________

First Name: ______________________

Section Leader: ______________________

I accept both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this test, nor will I give any. My answers represent my own work.

(signed) ______________________________________

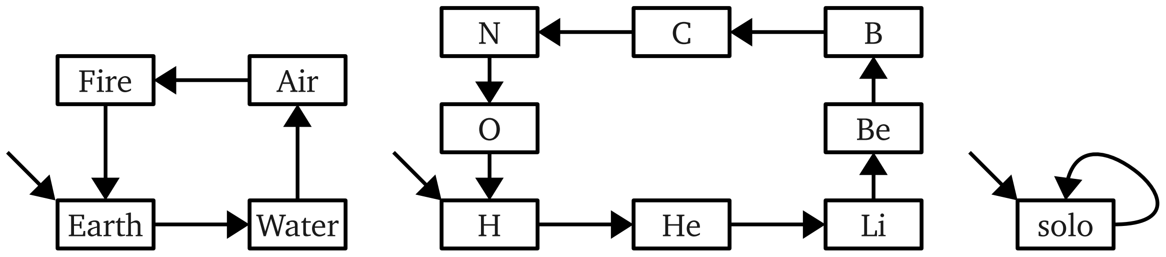
You have three hours to complete this midterm. There are 180 total points, and this midterm is worth 20% of your total grade in this course. As a rough measure of the relative difficulty of the problems, there is one point on this exam per minute of testing time.

Question		**Points**	**Grader**
(1) Linked Lists	30	/ 30	
(2) Hash Tables	40	/ 40	
(3) Binary Search Trees	30	/ 30	
(4) Grids	50	/ 50	
(5) Relatively Short Answer	30	/ 30	
	(180)	**/ 180**	

Good Luck!

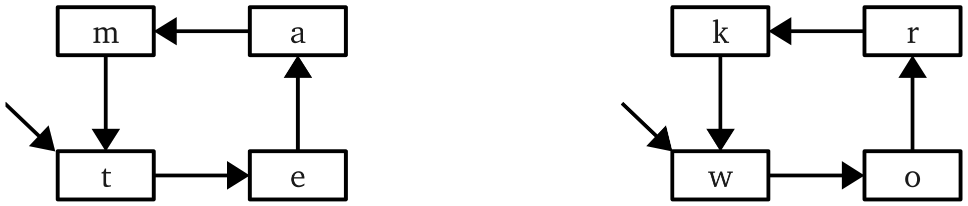
Problem One: Linked Lists (30 Points)

A *circularly-linked list* is a linked list where the very last cell, rather than pointing to **`NULL`**, instead points back to the very first node in the list. For example, the following are circularly-linked lists:

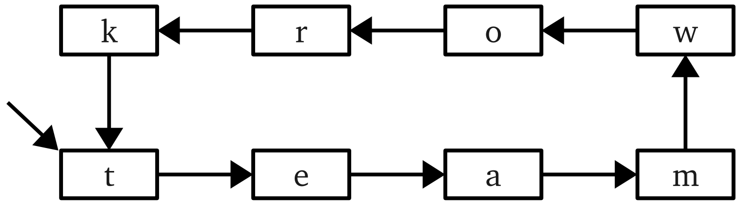


The leftmost list represents the four-element sequence ⟨Earth, Water, Air, Fire⟩, the middle list the eight-element sequence ⟨H, He, Li, Be, B, C, N, O⟩, and the rightmost list the one-element sequence ⟨solo⟩.

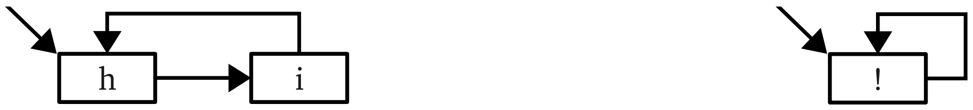
Suppose that you have two circularly-linked lists representing two sequences and wish to concatenate these sequences together. For example, given the following circularly-linked lists representing the sequences ⟨t, e, a, m⟩ and ⟨w, o, r, k⟩:



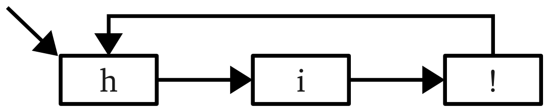
You would like to produce the following circularly-linked list for the sequence ⟨t, e, a, m, w, o, r, k⟩:



Similarly, given the following circularly-linked lists for the sequences ⟨h, i⟩ and ⟨!⟩:



You would want to produce the following circularly-linked list for the sequence ⟨h, i, !⟩:



Feel free to tear out this page as a reference. The question is on the next page.

Assuming that the nodes in the circularly-linked list are represented with the following **struct**:

```
struct Node {
    string value;
    Node* next;
};
```

Write a function

```
Node* concatenateCircularlyLinkedLists(Node* first, Node* second);
```

that accepts as input pointers to two different circularly-linked lists, then rewires them to form a new circularly-linked list representing the concatenation of those two lists. The function should then return a pointer to the very first cell in the resulting circularly-linked list.

Your implementation of **concatenateCircularlyLinkedLists** must not allocate any new heap storage and must instead change the pointers in the existing cells. Since all of the collections types we have seen so far (**Vector**, **Stack**, **Queue**, etc.) allocate heap storage, this precludes the usage of any of those types.

In implementing this function, you can assume the following:

- Neither **first** nor **second** will be **NULL**.
- The pointers **first** and **second** refer to the starts of different circularly-linked lists, meaning that you will not get two pointers to the same circularly-linked lists.

```
Node* concatenateCircularlyLinkedLists(Node* first, Node* second) {
```

(Extra space for Problem One, if you need it.)

Problem Two: Hash Tables (40 Points)

In last Monday's lecture, we implemented the basic functionality of a type called **OurHashMap**, which represented a map from **string**s to **int**s backed by a hash table. The **private** section of that class is reproduced here:

```
private:
      /* A struct representing a cell in a linked list that holds a key and
       * a value.
       */
      struct Cell {
            string key;
            int value;
            Cell* next;
      };

      /* Pointer to an array of buckets, each of which is in turn a pointer
       * to a linked list of the elements in that bucket.
       */
      Cell** buckets;

      /* The total number of buckets in the hash table. */
      int numBuckets;

      /* The total number of elements in the hash table. */
      int numElems;
```

Internally, the implementation used a function called **hashCode** to hash from arbitrary strings to non-negative integers. The implementation then determined the bucket a particular string belonged to by calling the **hashCode** function and modding by the number of buckets. For reference, the **hashCode** function has the following signature:

```
int hashCode(string key);
```

The version of **OurHashMap** we defined in lecture had a **put** function that could be used to insert key/value pairs into the map, but there was no way to *remove* a key from the map. It would be great to fix this!

Implement a member function

```
bool remove(string key);
```

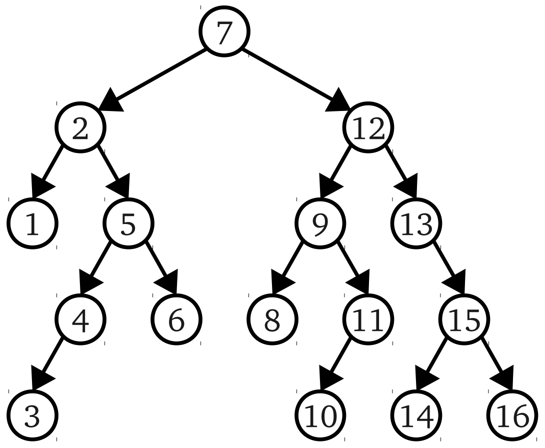
that accepts as input a key, then removes from the map the key/value pair that has that key (if such a key/value pair exists). The function should return a **bool** that is **true** if some key/value pair was removed and **false** otherwise. Your function should not leak any memory.

Feel free to tear this page out of the exam for reference, and write your solution on the next page.

```
bool OurHashMap::remove(string key) {
```

Problem Three: Binary Search Trees (30 Points)

In a binary search tree, a node *u* is called an *ancestor* of a node *v* if the path from *v* back up to the root of the tree passes through *u*. For example, consider this BST:



Here, the ancestors of 6 are 6, 5, 2, and 7; the ancestors of 10 are 10, 11, 9, 12, and 7; and the only ancestor of 7 is 7 itself. (Note that every node is always an ancestor of itself).

A *common ancestor* of two nodes v_1 and v_2 is a node *u* such that *u* is an ancestor of both v_1 and v_2. For example, in the above BST, 3 and 6 have 5 as a common ancestor, 9 and 16 have 7 as a common ancestor, and 2 and 15 have 7 as a common ancestor. The common ancestor of two nodes might be one of those nodes: for example, 12 is a common ancestor of 8 and 12.

Finally, the *lowest common ancestor* of two nodes v_1 and v_2 is the node *u* such that *u* is a common ancestor of v_1 and v_2 that is as deep as possible. For example, the common ancestors of 14 and 16 include 7, 12, 13, and 15, of which 15 is their lowest common ancestor. The lowest common ancestor of 1 and 6 is 2, and the lowest common ancestor of 10 and 16 is 12. The lowest common ancestor of two nodes might be one of those nodes: the lowest common ancestor of 13 and 16 is 13, for example.

Suppose that the nodes in a binary search tree are defined as follows:

```
struct Node {
    int value;
    Node* left;
    Node* right;
};
```

Your job is to write a function

```
Node* lowestCommonAncestorOf(Node* root, Node* v1, Node* v2);
```

that accepts as input a pointer **`root`** to the root of a BST, along with two pointers **`v1`** and **`v2`** to nodes in that BST, then returns a pointer to the lowest common ancestor of **`v1`** and **`v2`**.

In writing this function, you can assume the following:

- None of the input parameters will be **`NULL`**.
- **`v1`** and **`v2`** may refer to the same node.
- **`v1`** and **`v2`** are indeed nodes in the tree given by the **`root`** parameter.

Feel free to tear this page out as a reference, and write your solution on the next page.

```
Node* lowestCommonAncestorOf(Node* root, Node* v1, Node* v2) {
```

Problem Four: Grid (50 Points)

One container type we did not talk about implementing is the **`Grid`**, which represents a 2D array of values. This question explores one possible implementation of this type.

Unlike the **`Stack`**, **`Vector`**, **`Map`**, etc., the **`Grid`** type has its size specified when it is created and does not grow or shrink. One simple way to represent an *m* × *n* grid of values in C++ is as a multidimensional array of size *m* × *n*. When the grid is created, all of the elements in the grid are then initialized to an initial value (for **`int`**s, this is **`0`**). This requires O(*mn*) storage space.

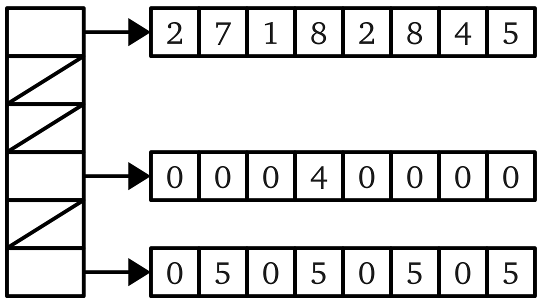
In many cases, though, the client of the **`Grid`** will not actually fill in all of the grid locations, leaving many of them holding their initial values. For example, suppose that only the top and bottom row of the grid are actually updated to hold new values. In that case, it's wasteful to store the middle rows of the grid, since we already know what values are stored there (namely, each cell holds the initial value). In the case where in advance it's known that not all rows of the **`Grid`** will be used, it's possible to reduce the amount of space the **`Grid`** will use by not actually allocating space for those rows until values are written to them.

Here is how this data structure will work. To represent a 2D array of size **`numRows`** × **`numCols`**, we will use an "array of arrays," with a top-level array of **`numRows`** pointers, each of which either points to **`NULL`** or to an array of **`numCols`** values. If none of the elements in row **`m`** have been written to, then the **`m`**th pointer in the top-level array will be **`NULL`**. Otherwise, if any element in row **`m`** is written to, then the **`m`**th pointer in the top-level array will point to an array of **`numCols`** elements that represents the values stored in that row.

For example, suppose that we want to store this 2D array of **`int`**s with our data structure:

2	7	1	8	2	8	4	5
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	4	0	0	0	0
0	0	0	0	0	0	0	0
0	5	0	5	0	5	0	5

Since **`0`** is the initial value for **`int`**s, we would represent this grid as



Notice how all of the rows that are all **`0`** are represented as **`NULL`** pointers, while the rows containing nonzero values actually have arrays allocated for them.

When the data structure is initially constructed, the top-level array of size **`numRows`** is allocated, but none of the arrays for any of the rows are allocated. Whenever a value is read from the grid, if the row containing the value has not been allocated yet, the implementation can just return **`0`**, since that's the default value for **int**s. There's no need to allocate the row in this case. Whenever a value is written to one of the rows in the grid, if the array for that row is already allocated, the new value is placed directly into that array. If a value is written and the array for that row has not yet been allocated, the array is then allocated, all of its elements are set to **0**, and the new value is then written. (In principle, you could deallocate a row if it ever becomes all **0**s, but for simplicity you don't need to do this.)

Your task in this problem is to implement a class **LazyGrid** that uses this representation to store a grid of **int**s, saving space on rows that are all **0**. The remainder of this page lists the interface of the class that you will be implementing, and the remaining pages give space for you to implement all of these member functions and specify the data members and helper member function you need in order to correctly implement the class.

Although it is possible to implement this structure by using a **Vector** of **Vector**s, for the purposes of this problem **you must dynamically allocate and deallocate your own memory.**

Feel free to tear out this and the preceding page for reference, and write your answer on the next pages.

```
/* A class representing a 2D array of integers. */
class LazyGrid {
public:
   /* Constructor: LazyGrid(int nRows, int nCols);
    * Usage: LazyGrid g(137, 42);
    * ---------------------------------------------------------------------------------
    * Constructs a new grid of size nRows × nCols.  This function will initialize
    * the top-level array, but will not fill in any of the rows.
    *
    * If either nRows or nCols is negative, this function should report an error.
    */
   LazyGrid(int nRows, int nCols);

   /* Destructor: ~LazyGrid();
    * Usage: (implicit)
    * ---------------------------------------------------------------------------------
    * Deallocates all memory allocated by this data structure.
    */
   ~LazyGrid();

   /* int getAt(int row, int col);
    * Usage: int value = g.getAt(0, 0);
    * ---------------------------------------------------------------------------------
    * Returns the value stored in the grid at position (row, col), reporting an
    * error if the index is out of range.
    */
   int getAt(int row, int col);

   /* void setAt(int row, int col, int value);
    * Usage: g.setAt(0, 0, 137); // grid[0][0] is now 137.
    * ---------------------------------------------------------------------------------
    * Sets the value at position (row, col) to be value, reporting an error if the
    * index is out of range.  This function may cause a row to be allocated and
    * initialized if this is the first value written to this row.
    */
   void setAt(int row, int col, int value);
private:
   /* Pointer to the top-level array-of-arrays used by this data structure.  Since
    * each element in this array is a pointer to an array (an int*), we need the
    * pointer to be an int** (a pointer to an array of int*'s).
    */
   int** elems;

   /* Add any private data members or member functions that you would like. */











};

/* Implement the constructor here. */
LazyGrid::LazyGrid(int nRows, int nCols) {
















}
```

```
/* Implement the destructor here. */
LazyGrid::~LazyGrid() {

















}
/* Implement getAt here. */
int LazyGrid::getAt(int row, int col) {

















}
```

```
/* Implement setAt here. */
void LazyGrid::setAt(int row, int col, int value) {

}
/* Implement any additional helper member functions here. */
```

Problem Five: Relatively Short Answer (30 Points)

Answer each of the following questions in the space provided.

(i) Dynamic Arrays and Linked Lists (10 Points)

There are two common ways to implement at **`Stack`**: using a dynamic array that doubles its size when extra space is needed, and using a linked list where elements are added to and removed from the front of the list.

Give one advantage of a **`Stack`** backed by a dynamic array over a **`Stack`** backed by a linked list and one advantage of a **`Stack`** backed by a linked list over a **`Stack`** backed by a dynamic array. Justify your answer.

(ii) Hash Tables and BSTs (10 Points)

There are two common strategies for implementing **`Set`**: as a hash table and as a balanced binary search tree.

Give one advantage of a **`Set`** backed by a hash table over a **`Set`** backed by a balanced binary search tree and one advantage of a **`Set`** backed by a balanced binary search tree over a **`Set`** backed by a hash table. Justify your answers.

(iii) Treesort **(7 Points)**

Consider the following sorting algorithm: given a list of values, insert all of the values into a binary search tree, then do an inorder traversal of the tree to get the list back in sorted order. This algorithm is called *treesort*.

Of the major sorting algorithms we discussed this quarter (selection sort, insertion sort, mergesort, quicksort), which one has behavior most similar to treesort? Justify your answer.

(iv) CS106B: The Sequel **(3 Points)**

If you could give a one-sentence suggestion to future CS106B students, what would it be? (We'll give you full credit regardless of what your answer is, as long as it's one sentence long!)