CS106X   Instructor: Cynthia Lee

Autumn 2015   Practice Exam

PRACTICE FINAL EXAM 3

NAME (LAST, FIRST): _____

SUNET ID:_____ @stanford.edu

Problem	Sorting	Heap	BST	Trees	Graphs, Classes	TOTAL
	1	2	3	4	6	
Score						
Possible						

Instructions:

- The time for this exam is **3 hours**.
- Use of anything other than a pencil, eraser, pen, one 8.5x11 page (one side) of notes, and the official textbook is prohibited. In particular, **no** computers or digital devices of any kind are permitted. Blank scratch paper may be provided by proctors and does not need to be turned in.
- PLEASE rip off the pages of library reference in the back of the exam and do not turn them in.
- **Please do NOT insert new pages** into the exam. Changing the number of pages in the exam confuses our automatic scanning system. You may use the back sides of the exam pages for more space. Thanks.
- SCPD and OAE: Please call or text 760-845-7489 if you have a question.

Please sign *before* you begin:

I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

_____ _____ _____

(Signature)   (Date)   (Start time - HH:MM zone)

1. Sorting. Show each step of insertion sort on the vector of integers shown below. The pseudocode for insertion sort is given as a reminder.

```
print(vector) // Initial state means the values at this point in the code
for (i = 1; i<vector size; i++) {
      j = i
      while (j > 0 and vector[j-1] > vector[j]) {
            temp = vector[j]
            vector[j] and vector[j-1]
            vector[j-1] = temp
            j—
      }
      print(vector) // You write vector's values at this point in the code
}
```

Below is the initial state of the vector of values to sort. Fill in the rest of the steps for each time the "print" executes. You <u>may not need</u> all the provided blank vectors to complete your solution.

9	4	8	5	1	2	3	7	6

2. **Heaps.** Draw the tree structure that results from interpreting the data in the array below as a binary heap. Then answer the question about your drawing.

30	20	12	9	13	10	2	7	5	1

DRAW:

CIRCLE ONE: Is this a valid max-heap?   YES   NO

3. **BSTs .** We have implemented the Map ADT using a plain Binary Search Tree (no special balancing measures). Our node structure is defined as "`struct node { int key; int value; node* left; node* right; };`" Draw a diagram of the BST that results from inserting the following (key, value) pairs in the order given. Please label each BST node with both the key and value. (5,2), (5,5), (9,3), (6,12), (7,5), (8,1)

Diagram after inserting (5,2):	Diagram after inserting (5,5):
(5,2) *This one is completed for you as a node formatting example.*	
Diagram after inserting (9,3):	Diagram after inserting (6,12):
Diagram after inserting (7,5):	Diagram after inserting (8,1)

4. **Trees.** Recall that one application of a Stack is to process an arithmetic expression written in postfix notation. We can also represent arithmetic expressions as a tree. Your job is to write a function that will convert a Stack of singleton expression nodes (where all the `left` and `right` pointers are initially set to NULL) into the correct tree structure. Note that you do not need to actually calculate the "answer" to the arithmetic expression. Here is an example:
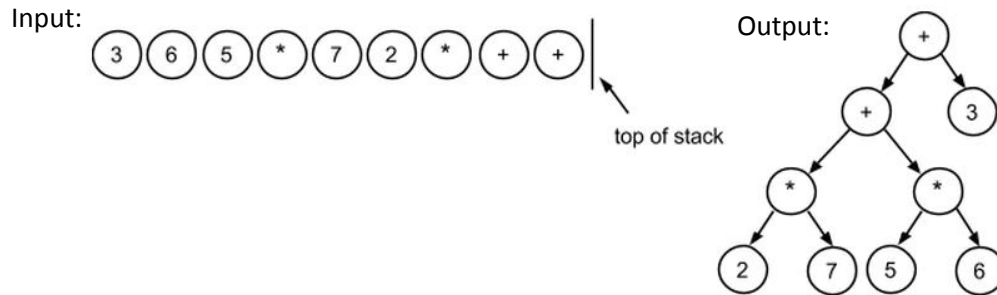
The struct for our expression tree nodes is as follows:

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

- The `operator` field can have three values: `'+'`, `'*'`, and `'N'`, where `'+'` and `'*'` represent the addition and multiplication operators, and `'N'` means that this node represents a number in the expression, not an operator.
- When the operator is `'N'`, the `value` field contains the value of the number.
- The `left` and `right` fields point to the left and right children in the expression tree.

Your function should have the following signature:

```
Node* makeTree(Stack<Node*>& expression);
```

- Return a pointer to the root of your newly constructed tree (if `expression` is empty, return NULL).
- When your function returns, `expression` should be empty (all contents popped).
- You may assume that the provided `expression` generates a valid tree (so the Stack should always have something remaining to pop when you need it to).

Algorithm hints:

- Pop a Node* from the Stack.
  - If the popped Node is an operator, you'll need to go build its left subtree, and then, when that is complete, go build its right subtree. (There is a reason you are given the nodes in a Stack: if your algorithm is structured correctly, you will get them in just the right order.)
  - If the popped Node is a number, then it will have no children (it will be a leaf node). So you should stop building children along this branch of the tree when you get to a number.

```
Node* makeTree(Stack<Node*>& expression) {
```

5. **Hashing.** Imagine we want to be able to store binary trees in a hash table. Recall that this means that we need a hash function that takes as input a binary tree and outputs an integer hash key in the range 0 to number-of-buckets. We have designed the following recursive formula as the pseudocode for our hash function (it is parametrized to a simple integer hash function h^1, k buckets, and constant primes P and Q):

   - $hash(T) = (P * hash(T\text{->}left) * h(T\text{->}val) + Q * hash(t\text{->}right)) \pmod{k}$

   Write a function `simulateTreeHash` that, given a `Vector<node*>` of binary trees (each vector entry is a pointer to a root), returns a `Map<int, Vector<node*>>` showing, for each integer bucket index, which trees would fall into that bucket when using the above formula. Assume you are provided the function h with this signature: "`int h(int val);`" but you will write all the rest of the necessary code (helper/wrapper functions are ok).

   ```
   struct node {
       int val;
       node* left;
       node* right;
   };
   const int P = 715827883;
   const int Q = 2796203;

   Map<int, Vector<node*>> simulateTreeHash(Vector<node*> trees, int k) {
   ```

---

[1] In other words, **h** is a hash function that goes from integer input to integer output. We will use it as a helper for the hash function that we are writing that goes from binary tree input to integer output. Assume it is already defined and you can just call it.

// more space for Hashing problem (you should not need this much space)

6. **Graphs and Classes.** For this problem, you will implement a variant of the Game of Life[2] that is sometimes given as the first assignment in CS106B/X. The Game of Life represents cells in a petri dish, where the number of cells is constant, but in each "generation," a given cell can be alive or dead. The cells start in some initial configuration of alive or dead, and then simulate a new "generation" by doing a pass over all of them to update their alive/dead status. When doing this pass over each cell, we determine if a given cell will be alive in the next generation by examining its **friends** and seeing how many are alive or dead—**a "friend" is like a customized definition of "neighbor" in a graph.** Once you've counted the number of alive friends (for our special definition of "friend"), the rules are as follows:

- If 0-1 friends of a cell are alive, the cell dies.
- If 2 friends of a cell are alive, the cell is stable (if alive, stays alive; if dead, stays dead).
- If 3 friends are alive, the cell will be alive.
- If 4 or more friends are alive, the cell dies of overcrowding.
- All the changes happen simultaneously, so if a previously dead cell becomes alive, do not then count it as alive when you count up the number of alive friends of one of its friends (at least not until the next round).

Our definition of a cell's "friends" are defined by weighted, undirected graph edges, so a cell can have any (non-negative) number of friends. Note that the nodes and their edge relationships do **not** change during the course of the game—the only change is whether the cell represented by a given node is alive or dead. The friends of a cell c are as follows:

- **If a path of total weight no more than 20 exists between c and d, then d is c's friend.**
- **c is not its own friend.**

Example: In the graph below, cell 1 has 3 neighbors (2, 3, and 4).
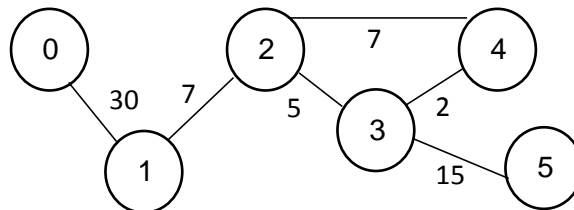
Write a class that computes this new version of the Game of Life. Your class should have exactly three public member functions, as follows:

- `void initializeCells(string filename)`
    - Randomly initializes all cells to be alive or dead (with equal probability). Suggestion: "`bool alive = getRandomInteger(0,1);`"
    - Reads the configuration of the graph edges from a file. The file format is as follows: (1) the first line of the file is the number of cells, (2) then follows one line per cell (starting with cell 0, cell 1, etc), with space-separated integers giving the weights of the edges for that cell to every other cell. **A weight of 0 means there is no edge between those two**

---

[2] This problem was originally given in a quarter when the Game of Life was given as an assignment, which of course was not the case for you this quarter. That may make this problem somewhat more challenging for you than it was for those students. On the other hand, some of those students may have had a hard time wrapping their mind around this strange version of the Game of Life, having grown accustomed to the assignment version, which could actually make it harder than seeing this version with fresh eyes. In any case, this should be solvable for you as a Practice Final question.

cells. All actual edge weights are > 0. Example file contents for the graph above (comments are not part of the file):

```
6                        // total number of cells
0 30 0 0 0 0             // adjacent to cell 0
30 0 7 0 0 0             // adjacent to cell 1
0 7 0 5 7 0              // etc.
0 0 5 0 2 15
0 0 7 2 0 0
0 0 0 15 0 0
```

- void updateCells()
  - Updates the board according to one generation of the game.
- void printCells()
  - Prints the dead/alive state of each cell to cout in the format "[cell number]: [0/1]". Here is an example with 3 cells:

    ```
    0: 1
    1: 0
    2: 1
    ```

Your class needs to have private data for representing the state of the board; the design is up to you. You might find that using your own Graph representation (using other container classes) rather than the full BasicGraph class is simpler. You may also wish to define a cell struct to hold the state of a cell. If you want to add any helper functions, they should be private.

Below is life.h. Fill in the private section according to your design. On the following pages, fill in life.cpp with the required public functions and any private ones you may have added.

```
class LifeGame {
public:
   void initializeCells(string filename);
   void updateCells();
   void printCells();
private:      //this may be more space than you need
```



```
};
```

```cpp
// life.cpp

#include <iostream>

void LifeGame::initializeCells(string filename){
```

```
//lifegame.cpp continued

void LifeGame::updateCells() {
```

```
//lifegame.cpp continued

void LifeGame::printCells() {
```

Summary of Relevant Data Types

We tried to include the most relevant member functions for the exam, but not all member functions are listed. You are free to use ones not listed here that you know exist. **You do <u>not</u> need #include.**

```
class string {
 bool empty() const;
 int size() const;
 int find(char ch) const;
 int find(char ch, int start) const;
 string substr(int start) const;
 string substr(int start, int length) const;
 char& operator[](int index);
 const char& operator[](int index) const;
};

class Vector {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem); // operator+= used similarly
 void insert(int pos, const Type& elem);
 void remove(int pos);
 Type& operator[](int pos);
};

class Grid {
 int numRows() const;
 int numCols() const;
 bool inBounds(int row, int col) const;
 Type get(int row, int col) const; // cascade of operator[] also works
 void set(int row, int col, const Type& elem);
};

class Stack {

 bool isEmpty() const;
 void push(const Type& elem);
 Type pop();
};

class Queue {
 bool isEmpty() const;
 void enqueue(const Type& elem);
 Type dequeue();
};
```

```
class Map {
 bool isEmpty() const;
 int size() const;
 void put(const Key& key, const Value& value);
 bool containsKey(const Key& key) const;
 Value get(const Key& key) const;
 Value& operator[](const Key& key);
};
```
Example range-based for: `for (Key key : mymap){…}`

```
class Set {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem);
 bool contains(const Type& elem) const;
};
```
Operators:
```
set + value // Returns the union of set set1 and individual value value
set += value // Adds the individual value value to the set set
set1 += set2 // Adds all the elements from set2 to set1
```
Example range-based for: `for (Type elem : mymap){…}`

```
class Lexicon {
  int size() const;
  bool isEmpty() const;
  void clear();
  void add(std::string word);
  bool contains(std::string word) const;
  bool containsPrefix(std::string prefix) const;
};
```
Example range-based for: `for (string str : english){…}`

```
struct Edge {
    Vertex* start;
    Vertex* finish;
    double cost;
    bool visited;
};
struct Vertex  {
    std::string name;
    Set<Edge*> arcs;
    Set<Edge*>& edges;

    bool visited;
    Vertex* previous;
};
```

```
class BasicGraph : public Graph<Vertex, Edge> {
public:
BasicGraph();
    Vertex* addVertex(Vertex* v);
    const Set<Edge*>& getEdgeSet() const;
    const Set<Edge*>& getEdgeSet(Vertex* v) const;
    const Set<Edge*>& getEdgeSet(std::string v) const;
    Vertex* getVertex(std::string name) const;
    const Set<Vertex*>& getVertexSet() const;
    void removeEdge(std::string v1, std::string v2, bool directed = true);
    void removeEdge(Vertex* v1, Vertex* v2, bool directed = true);
    void removeEdge(Edge* e, bool directed = true);
    void removeVertex(std::string name);
    void removeVertex(Vertex* v);
}
```