This practice exam is based on an actual final exam from CS106X (same topics coverage as CS106B, but somewhat higher expectations for mastery). The question types and mix of topics of our CS106B exam will be the same, but this practice may over-prepare you to some degree in terms of the difficulty of a couple of the individual problems. As my basketball coach would always say when we were conditioning—a practice that is harder than the game is a good thing! --Cynthia

CS106B — Instructor: Cynthia Lee

Spring 2016 — Practice Exam

PRACTICE FINAL EXAM 1

NAME (LAST, FIRST): ____

SUNET ID:____ @stanford.edu

Problem	1	2	3	4	5	6	TOTAL
Topic	Graphs	Pointers, Linked Lists	Recursion	BST, Heap	Inheritance	Algorithms	
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.

____ ____________ ____________________

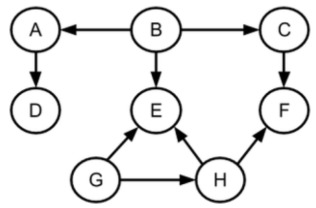
(Signature) (Date) (Start time - HH:MM zone)

1. **Graphs.** You are given a directed, unweighted graph as a `BasicGraph` (the same library class you used in the Trailblazer assignment). The graph is not necessarily connected. Your task is to find vertices in the graph that are actually a tree *(for a special definition of tree, given below)*. The function signature should be as follows:

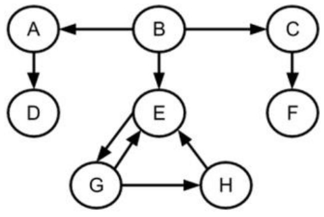
   ```
   Vertex* findLargestTree(BasicGraph& graph)
   ```

   - **`graph`** is the input graph. You should not change its structure (vertex set, edges), but you may change and use the vertex state (visited, color, cost/weight, etc.) in your algorithm.
   - **Return a pointer to the root of the largest tree in the graph**. If there is no such vertex in the graph (no valid trees), then return **`NULL`**. If there is a tie for largest tree, return the root of any one.
   - **What is a tree?** A tree in this case is defined as follows:
     - A tree consists of **a root node** (one vertex) and **all vertices reachable from that root**, and **all edges outgoing from the included vertices**.
     - No vertex in the tree may be reachable by more than one path from the root (this also means no cycles allowed).
     - Ignore edges that are **incoming** to a vertex in the tree **from a vertex not in the tree**. You may **not** ignore vertices that are reachable from any vertex included in the tree—in other words, all paths need to be followed to leaves/dead ends.

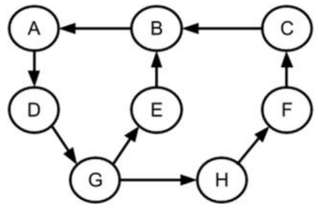
   Examples:



Return a pointer to Vertex B.
Vertex B is the root of a tree that includes vertices {A, B, C, D, E, F}. Note that G is **not** even under consideration by size, because it is not the root of a valid tree (E is reachable by more than one path: G->E and G->H->E).



Return a pointer to Vertex A *or* Vertex C.
Note that Vertex B is **not** the root of a valid tree because nodes reachable from it include a cycle.



Answer: Return NULL.

Write your solution on the next page. As usual, you are free to write helper function(s).

```
Vertex* findLargestTree(BasicGraph& graph) {











}
```

2. **Pointers and Linked Lists.** Write the **`contains`** function that given two linked lists will determine whether the second list is a subsequence of the first. To be a subsequence, every value of the second must appear within the first list and in the same order, but there may be additional values interspersed in the first list. A list contains itself; the NULL list is contained in any list.

Examples:

list	**sub**	**contains(list, sub)**
`1 -> 4 -> 2 -> 9`	`1 -> 4`	`true`
`1 -> 4 -> 2 -> 9`	`9 -> 4`	`false`
`1 -> 4 -> 2 -> 9`	`4 -> 9`	`true`
`1 -> 4 -> 2 -> 9`	`1 -> 1 -> 4`	`false`
`1 -> 4 -> 2 -> 9`	`2 -> 9 -> 10`	`false`

```
struct listnode {
   int val;
   listnode * next;
};

bool contains(listnode * list, listnode * sub) {
```

```
}
```

3. **Recursion.** A ternary (or 3-ary) tree is one where each node has up to three children. A node for this tree is defined as:

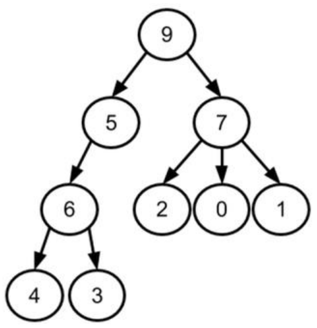
```
struct treenode {
   treenode(int k, treenode * l, treenode * m, treenode * r) {
       key= k; left = l; middle = m; right = r;
   };
   int key;
   treenode * left;
   treenode * middle;
   treenode * right;
};
```

Given a ternary tree, identify the subset of its nodes with the maximum sum of their **key** fields, subject to the constraint that no pair of the selected nodes may be ancestors/descendants of each other. Your function signature should be as follows:

```
Set<int> maxSumSubset (treenode * root);
```

- As the title of this problem suggests, you should solve this using recursion.
- You may assume that all keys in the tree are unique and non-negative.
- Unlike Heaps or BSTs, the keys have no ordering rule governing them.

Example:



Return a Set containing {4, 3, 7}.

Some of the reasoning involved is as follows: We would rather choose 4 and 3 than their parent 6 because 4+3>6. We would rather choose 7 than its children 2+0+1.

Write your solution on the next page. As usual, you are free to write helper function(s).

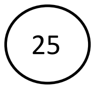
```
Set<int> maxSumSubset (treenode* root) {







}
```

4. **BSTs and Heaps.**

(a) 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 **<u>both</u> the key and value**. (25,2), (17,1), (29,0), (55,1), (45,7), (29,3)

Diagram after inserting (25,2): *This one is completed for you.*	Diagram after inserting (17,1):
Diagram after inserting (29,0):	Diagram after inserting (55,1):
Diagram after inserting (45,7):	Diagram after inserting (29,3):

(b) We have implemented the Priority Queue ADT using a binary min-heap. Draw a diagram of the heap's tree structure that results from inserting the following priority values in the order given: 25, 37, 28, 12, 30, 3

Diagram after inserting 25:  *This one is completed for you.*	Diagram after inserting 37:
Diagram after inserting 28:	Diagram after inserting 12:
Diagram after inserting 30:	Diagram after inserting 3:

5. **Inheritance.** Consider the classes on the left; assume that each is defined in its own file.

```
class Byron {
public:
    virtual void m3() {
        cout << "B 3" << endl;
        m1();
    }

    virtual void m1() {
        cout << "B 1" << endl;
    }
};

class Yeats : public Byron {
public:
    virtual void m3() {
        Byron::m3();
        cout << "Y 3" << endl;
    }

    virtual void m4() {
        cout << "Y 4" << endl;
    }
};

class Plath : public Yeats {
public:
    virtual void m1() {
        cout << "P 1" << endl;
        Yeats::m1();
    }

    void m3() {
        cout << "P 3" << endl;
    }
};

class Angelou : public Plath {
public:
    virtual void m4() {
        cout << "A 4" << endl;
        m3();
    }

    void m3() {
        cout << "A 3" << endl;
    }
};
```

Now assume that the following variables are defined:

```
Byron* var1 = new Plath();
Yeats* var2 = new Angelou();
Byron* var3 = new Byron();
Byron* var4 = new Yeats();
Yeats* var5 = new Plath();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the **line breaks with slashes** as in "x / y / z" to indicate three lines of output with "x" followed by "y" followed by "z".

If the statement does not compile, write "**compiler error**". If a statement would crash at runtime or cause unpredictable behavior, write "**crash**".

Statement	**Output**
`var4->m3();`	
`var4->m1();`	
`var4->m4();`	
`var2->m3();`	
`var2->m1();`	
`var2->m4();`	
`var1->m4();`	
`var1->m3();`	
`var1->m1();`	
`var5->m1();`	
`var5->m4();`	
`var5->m3();`	

Statement	**Output**
`((Yeats*) var4)->m3();`	
`((Yeats*) var4)->m4();`	
`((Angelou*) var3)->m4();`	
`((Byron*) var5)->m4();`	
`((Plath*) var2)->m3();`	
`((Angelou*) var2)->m3();`	

6. **Algorithms.** When asked for Big-O analysis, give a tight bound of the nearest runtime complexity class.

(a) The function **Binky** is defined as follows:

```
int Binky(int n)
{
    if (n <= 1) return 1;
    if ((n % 2) == 0)
       return 2 * Binky(n/2);
    else
       return Binky(n + 1);
}
```

Give the computational complexity of **Binky** expressed in big-O notation, where *N* is the value of the argument **n**, assumed to be a nonnegative integer. Briefly justify your answer.

(b) Removing a cell from a singly-linked list typically requires not only a pointer to the cell but also to its previous cell. You propose getting around this by overwriting the contents of the cell with the value of the cell that follows and then deleting the following cell instead when you need to delete a cell. Using this idea, you've written a new version of **deleteCell**:

```
void deleteCell(Cell *ptr) {
   Cell *toDelete = ptr->next;
   *(ptr) = *(ptr->next); // struct assignment copies over all fields
   delete toDelete;
}
```

Does this strategy work? YES NO (circle) Briefly explain why or why not:

(c) Given an unsorted input of N integers, you wish to print the median element. Assume N is odd. Three different algorithms are proposed to finding the median. Each is correct, but they have different performance profiles

1. Run the first N/2 passes of **`SelectionSort`** and print the lastmost element swapped.
2. Sort the array using **`MergeSort`** and print out the middlemost element.
3. Follow this algorithm starting with K = N/2 + 1. Choose the first element as the pivot and use the **`Quicksort`** partition function to divide into a left (all elements smaller than pivot) and a right section (all elements larger than pivot). Let L be the number of elements in left section. If L = K, print the pivot and you're done. If K < L, recursively apply algorithm to find Kth element in left section. If K > L, recursively apply the algorithm to find the (K - L)th element within right section.

Give the big-O running time (tight bound) of each algorithm in the worst case.

	Worst-case big-O
1.	
2.	
3.	

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);
}
```