CS106B — Instructor: Cynthia Lee

Spring 2016 — June 1, 2016

PRACTICE FINAL EXAM #4 (DRESS REHEARSAL)

NAME (LAST, FIRST): ____

SUNET ID:____ @stanford.edu

Problem	1	2	3	4	5	6	7	TOTAL
Topic	MST	Linked List	Heap	Graphs	BSTs	Inheritance	ADTs	
Score								
Possible	3	5	10	12	10	8	12	60

Instructions:

- The time for this exam is **3 hours**. There are 60 points for the exam, or about 3 minutes per point.
- Use of anything other than a pencil, eraser, pen, two 8.5x11 pages (two sides) 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 but should not 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.

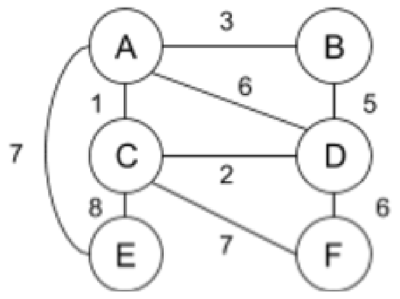
Please sign *before* you begin:

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

____ ______________

(Signature) (Date)

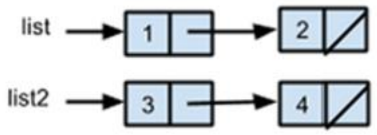
1. **Kruskal's (3pts).** For this problem, you will simulate Kruskal's minimum spanning tree algorithm. This is the algorithm you coded for the Maze-generating part of Trailblazer, and we also simulated it in class. (Recall it uses a priority queue and where you keep track of groups of vertices that are connected to each other but not to other groups.) Write the edges of the Minimum Spanning Tree, **in the order** that Kruskal's selects them (from first to last). Name edges according to their start and end points (e.g., either "AB" or "BA").



`MST edges:` ____

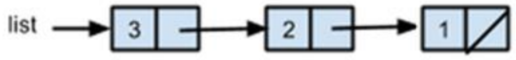
2. **Linked Lists (5pts).** Write code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown and/or creating new nodes as needed. This is not a general algorithm—you are writing code for this example **only**, using the variable names shown. You are NOT allowed to change any existing node's data field value, nor create new ListNode objects, but you may create a single `ListNode*` pointer variable to point to existing nodes. **Your code should NOT leak memory.** If a variable does not appear in the "after" picture, it doesn't matter what value it has after changes are made.

Before:



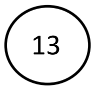
```
struct ListNode {
    int data;
    ListNode* next;
};
```

After:



3. **Heaps (10pts).**

(a) We have implemented the Priority Queue ADT using a **binary <u>min</u>-heap**. Draw a diagram of the heap's tree structure that results from inserting the following priority values in the order given: 13, 4, 12, 8, 11, 3.

Diagram after inserting 13:  *This one is completed for you.*	Diagram after inserting 4:
Diagram after inserting 12:	Diagram after inserting 8:
Diagram after inserting 11:	Diagram after inserting 3:

(b) Continuing from Part (a), draw the array-based heap representation of the last drawing you made in Part (a) (after inserting 3). The array index labels are provided for you. You may use 0-based or 1-based indexing for your heap, as shown in class/homework (full credit for either even though I have a favorite ☺).

0	1	2	3	4	5	6	7	8	9

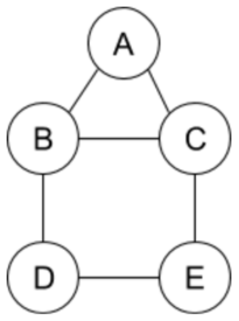
(c) Continuing with the last drawing you made in Part (a) (after inserting 3), perform three dequeue (*i.e.,* extract-min) operations.

Copy your last diagram from Part (a).	Diagram after dequeue: Value dequeued: ________
Diagram after dequeue: Value dequeued: ________	Diagram after dequeue: Value dequeued: ________

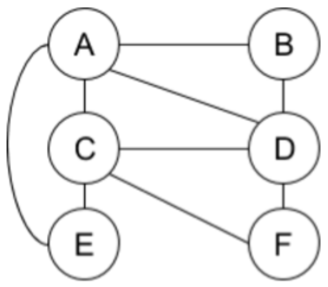
4. **Graphs (12pts).** Write a function named `findEulerPath` that accepts as a parameter a reference to a `BasicGraph`, and tries to find an Euler path in the graph, returning the path as a `Vector` of `Vertex` pointers. *(A real-world example where you might want an Euler path: A snow plow wants to remove the snow from all the roads in the neighborhood without unnecessarily driving over the same road twice.)*

An Euler path is a path in an undirected graph that uses every edge in the graph exactly once. For example, in Graph 1 (below left), you can form an Euler path by starting at vertex B, and following this sequence of vertices: {B, D, E, C, A, B, C}. You'll find that this path uses every edge in the graph exactly once. It revisits some vertices more than once, which is fine--it is only the edges that we may not reuse. Graph 2 (below right) also contains an Euler path. Start at F and follow this sequence of vertices: {F, C, D, A, C, E, A, B, D, F}. This path happens to start and end at the same vertex, F. Again, that is fine and what matters is that this path uses every edge exactly once. If the graph contains multiple valid Euler paths, you may return any one of them.

Graph 1:



Graph 2:



Implementation notes:

- If the graph **does not contain an Euler path**, or if the graph does not contain any vertices, your function should return an empty vector. For example, in the first graph above at left, if the edge D-E were absent, then the graph would not contain an Euler path. If the graph does not contain any edges, you may return either an empty vector or a one-element vector containing any single vertex from the graph.
- Although there are other ways to do this, **you must write a solution that explores possible paths in the graph using <u>recursive backtracking</u>** (*i.e.*, explore the current node's neighbors one at a time, recursively exploring their neighbors, etc.). As usual with recursive backtracking, you should detect when your exploration leads to "dead-end" options that cannot be viable, and then backtrack.
- Since the **graph is undirected**, it takes two `BasicGraph` directed edges to represent what is conceptually one undirected edge (*e.g.*, an edge V1 to V2 and an edge V2 to V1). You may assume the input graph is a valid undirected graph. You may also assume the graph contains no self-edges (*e.g.*, from V1 to V1).
- As usual, you must match the provided function signature, but you may define helper functions and additional structs as needed. You should **<u>not</u>** modify the contents of the graph such as by adding or removing vertices or edges from the graph, though you may modify the state variables inside individual vertices/edges, such as visited, cost, and color.

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

```
Vector<Vertex*> findEulerPath(BasicGraph& graph) {









}
```

5. **BSTs (10pts).** Write a function `partitionTree` that takes a Binary Search Tree (BST) with one mystery node, whose key has been replaced by the sentinel value -1 (but which has been left in place in the BST), and partitions all the values in the tree into two lists: those values that must be less than the mystery node's original key and those that must be greater than the mystery node's original key. Moreover, each list should be returned in **sorted order**. The BST is implemented using this `BSTnode` struct:

```
struct BSTnode { int key; BSTnode * left; BSTnode * right; };
```

Your solution must use this function signature:

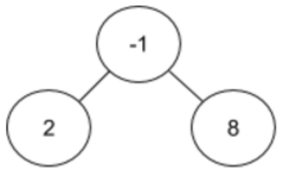
```
void partitionTree(BSTnode *tree, Vector<int>& lessThan, Vector<int>& greaterThan);
```

Implementation notes:

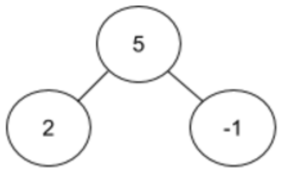
- **Your solution <u>must</u> use recursion.** Ideally, that's all it would use (though any correct solution can earn points). In other words, the core of solving it can be a simple, elegant recursive tree traversal algorithm, with no need for any "post-processing" such as a separate sorting or dividing of the list after the fact. That the two vectors should be returned in sorted order is a clue to the algorithm.
- As usual, you must match the function signature provided, but you may add helper methods.
- You may assume that all **keys in the tree are <u>unique</u> and <u>non-negative</u>**, except for one node with a key of -1, marking it as the one special "mystery" node.
- You may assume that `tree` points to the root of a valid BST (except the mystery node, which can appear anywhere in the BST with key -1).
- You may assume the Vector parameters `lessThan` and `greaterThan` are **initially <u>empty</u>**. When the `function` returns, they should be populated with the values that must be less than and greater than the mystery node's original key, respectively, and each **<u>sorted</u>** in ascending order.

Examples:

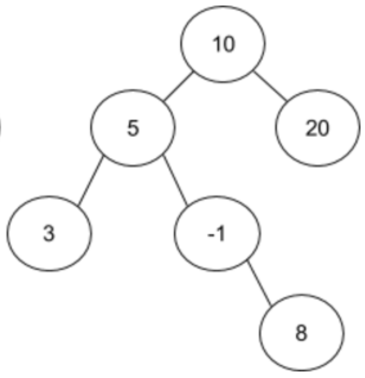
BST 1:



BST 2:



BST 3:



- For **BST 1**, you should return with `lessThan = {2}` and `greaterThan = {8}`.
- For **BST 2**, you should return with `lessThan = {2, 5}` and `greaterThan = {}`.
- For **BST 3**, you should return with `lessThan = {3, 5}` and `greaterThan = {8, 10, 20}`.

Write your solution on the next page.

```
void partitionTree(BSTnode *tree, Vector<int>& lessThan, Vector<int>& greaterThan) {

}
```

6. **Inheritance (8pts).** Consider the classes on the left; assume that each is defined in its own file.

```
class Hamilton {
public:
    virtual void m1() {
        cout << "H 1" << endl;
        m3();
    }

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

class Burr : public Hamilton {
public:
    virtual void m1() {
        Hamilton::m1();
        cout << "B 1" << endl;
    }

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

class Eliza : public Burr {
public:
    virtual void m2() {
        m2();
        m4();
        cout << "E 2" << endl;
    }

    void m4() {
        m3();
        cout << "E 4" << endl;
    }
};

class George : public Eliza {
public:
    virtual void m2() {
        cout << "G 2" << endl;
    }

    void m4() {
        cout << "G 4" << endl;
    }
};
```

Now assume that the following variables are defined:

```
Hamilton* var1 = new Eliza();
Burr*  var2 = new Burr();
Burr* var3 = new Eliza();
George* var4 = new George();
Eliza* var5 = new George();
```

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 (1/2pt each)**
`var1->m1();`	
`var1->m2();`	
`var2->m2();`	
`var3->m1();`	
`var3->m4();`	
`var4->m2();`	
`var4->m4();`	
`var4->m1();`	
`var5->m2();`	
`var5->m4();`	
`((Hamilton*) var4)->m2();`	
`((Eliza*) var4)->m4();`	
`((Eliza*) var1)->m1();`	
`((Eliza*) var2)->m2();`	
`((Hamilton*) var4)->m1();`	
`((Eliza*) var3)->m4();`	

7. **ADTs (12pts).** Tickets were scarce for the Pac-12 Championship game this past weekend. You pre-purchased `nTix` general admission tickets last month, but Saturday afternoon you realized you had too much final exam studying to do and you decided to sell your tickets—*but only* if you could get at least `minCash` total proceeds from the sale. After posting on Craigslist, you received several offers. Each offer comes from a customer with a **string name**, and each offer contains: (1) the **number of tickets** the customer wants (the customer will **not** buy unless they get exactly that number of tickets from you), and (2) the **total price** the customer will pay. Write a function that finds some combination of customers whose offers you can accept that will meet your `minCash` minimum total proceeds, while not selling more than the `nTix` you own.

```
struct offer {
    int seats;
    int price;
};

bool canSell(Map<string,offer>& customerOffers, int nTix, int minCash,
                 Vector<string>& acceptedOffers);
```

- `customerOffers` maps the customers' names (`string`) to their respective offers (`offer` struct).
- `acceptedOffers` is initially empty. When you return, it should contain the names of the customers whose offers you are accepting. If you were not able to find a set of customers whose offers met your constraints (and their constraints), `acceptedOffers` should be empty.
- Return true if you were able to find a set of customers whose offers met your constraints (and their constraints), otherwise return false.
- Return with the **first solution you find** (code does **not** look for the global maximum proceeds). You may find a solution that meets your minimum but does not sell all `nTix` tickets, and that's fine.

Implementation notes:

- **You must write a solution that explores possible combinations using recursive backtracking**. As usual with recursive backtracking, you should detect when your exploration leads to "dead-end" options that cannot be viable, and then backtrack.
- As usual, you must match the provided function signature, but you may define helper functions and additional structs as needed.

Write your solution on the next page.

}

```
bool canSell(Map<string,offer>& customerOffers, int nTix, int minCash,
             Vector<string>& acceptedOffers) {
```

```
}
```

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 Vertex  {
    std::string name;
    Set<Edge*> arcs;
    Set<Edge*>& edges;
    bool visited;
    Vertex* previous;
    Color getColor();
    setColor(int color);
};
```

```
struct Edge {
    Vertex* start;
    Vertex* end; //alias for finish, which also works
    double weight; //alias for cost, which also works
    bool visited;
};

class BasicGraph : public Graph<Vertex, Edge> {
public:
BasicGraph();
    bool isNeighbor(Vertex* v1, Vertex* v2) const;
    bool isNeighbor(string name1, string name2) const;
    const Set<Vertex*> getNeighbors(Vertex* vertex) const;
    const Set<Vertex*> getNeighbors(string vertex) const;
    Edge* getEdge(Vertex* v1, Vertex* v2) const;
    Edge* getEdge(std::string v1, std::string v2) const;
    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;
    bool containsEdge(Vertex* v1, Vertex* v2) const;
    bool containsEdge(string name1, string name2) const;
    bool containsEdge(Edge* edge) const;
    void resetData();

    /* For the graph problem, you are not allowed to edit the graph
     * structure, so we omitted addVertex/removeVertex, addEdge/removeEdge */
}
```