CS106B Instructor: Cynthia Lee

Autumn 2017

PRACTICE FINAL #5

NAME (LAST, FIRST): _____

SUNET ID:_____ @stanford.edu

Problem	1	2	3	4	5	6	7	TOTAL
Topic	Linked List	Big-O	Heap	BST	Graphs	Trees	Inheritance	
Score								
Possible	5	6	8	10	12	11	8	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:

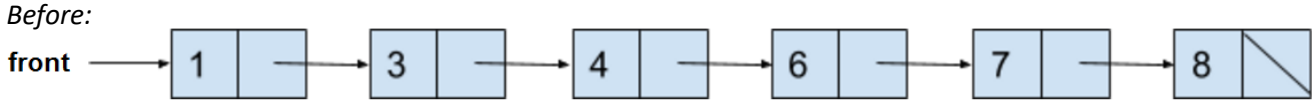
I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

_____

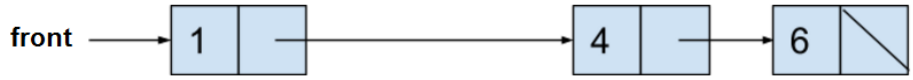
(Signature)

1. **Linked Lists (5pts).** Write code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown as needed. This is not a general algorithm—you are writing code for this example **only**, using the variable name (`front`) 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*` variable to point to existing nodes**. **Your code should not leak memory**.

Before:



After:




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

2. **Big-O (6pts).** Give a tight bound of the nearest runtime complexity class (worst-case) for each of the following code fragments in Big-O notation, in terms of variable N (**the variable N appears in the code** so use that value). As a reminder, when doing Big-O analysis, we write a simple expression that gives only a power of N, such as $O(N^2)$ or O(log N), *not* an exact calculation. Write your answer in the blanks on the right side. It may be helpful to remember this math identity: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. **Use the specific implementation of vector's <u>remove</u> that is shown below** for your analysis in parts (b) and (c). It is the same dynamically allocated array implementation that we used in ArrayList, so no surprises there (just provided for your reference).

(a)-(c) Questions (2pts each)	Answers

`bool search(Vector<int>& myvec, int key) {`
`  int N = myvec.size();`
`  return recursiveSearch(myvec, key, 0);`
`}`
 	
`bool recursiveSearch(Vector<int>& myvec, int key, int i) {`
`  if (i == myvec.size()) return false;`
`  if (myvec[i] == key) return true;`
`  return recursiveSearch(myvec, key, i + 1);`
`}`
	O( )

`bool search(Vector<int>& myvec, int key) {`
`  int N = myvec.size();`
`  return recursiveSearch(myvec, key);`
`}`
 	
`bool recursiveSearch(Vector<int>& myvec, int key) {`
`  if (myvec.size() == 0) return false;`
`  if (myvec[0] == key) return true;`
`  myvec.`**`remove`**`(0); // see code below`
`  return recursiveSearch(myvec, key);`
`}`
	O( )

`bool search(Vector<int>& myvec, int key) {`
`  int N = myvec.size();`
`  return recursiveSearch(myvec, key);`
`}`
 	
`bool recursiveSearch(Vector<int>& myvec, int key) {`
`  if (myvec.size() == 0) return false;`
`  if (myvec[myvec.size()-1] == key) return true;`
`  myvec.`**`remove`**`(myvec.size()-1); // see code below`
`  return recursiveSearch(myvec, key);`
`}`
	O( )

```
… void Vector<ValueType>::remove(int index) { //refer to this implementation
    for (int i = index; i < mySize; i++) {
        myElements[i] = myElements[i + 1];
    }
    mySize--;
}
```

3. **Heaps (8pts).**

(a) **Enqueue (6pts).** We have implemented the Priority Queue ADT using a **binary min-heap** stored in a dynamically-allocated array (0-based). Below is the state of the array after enqueuing the priority values 44 and then 55:

Before: `Size: 2` `Capacity: 10`

0	1	2	3	4	5	6	7	8	9
44	55								

After: `Size: _____` `Capacity: _____`

In the array spaces above, write what the binary min-heap looks like after enqueuing the following additional priority values, in this order: 50, 25, 60, 75, 15. **You may cross out and overwrite** existing 44 and 55 values if needed. In the indicated spaces above, also update the size and capacity as needed. You may use the space below as scratch space to draw the tree structure *(not graded)*.

(b) **Dequeue (2pts).** Starting with the priority queue that you drew above (after the additional enqueue operations), you will now perform a dequeue operation.

0	1	2	3	4	5	6	7	8	9

After: `Size: _____` `Capacity: _____`

In the array spaces above, write what the binary min-heap looks like after performing one dequeue operation. In the indicated spaces above, also update the size and capacity, as needed. You may use the space below as scratch space to draw the tree structure *(not graded)*.

4. **BST (10pts).** 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**. As a reminder, inserting a (key, value) pair corresponds to the following code using Stanford Library Map: "`mymap[key] = value;`" or, equivalently: "`mymap.put(key, value);`" Please **label each BST node with the key <u>and</u> value**, as shown in the first example entry below.

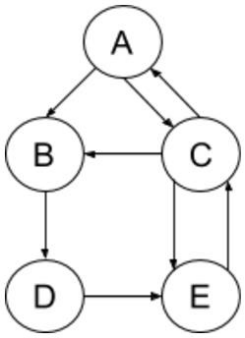
   Here are the (key, value) pairs to insert: (25,5), (4,5), (12,6), (4,10), (1,8), (8,1).

(a) Diagram after inserting (25,5):  *This one is completed for you.*	(b) Diagram after inserting (4,5):
(c) Diagram after inserting (12,6):	(d) Diagram after inserting (4,10):
(e) Diagram after inserting (1,8):	(f) Diagram after inserting (8,1):

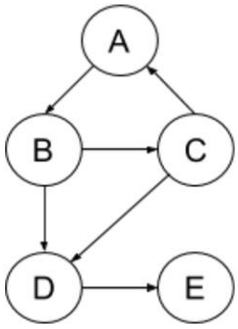
5. **Graphs (12pts).** Write a function named `findHamCycle` that accepts as a parameter a reference to a `BasicGraph`, and tries to find a Hamiltonian cycle in the graph, returning the path as a `Vector<Vertex*>`. **A Hamiltonian cycle[1] is a path in a directed graph that starts and ends at the same vertex and visits all other vertices in the graph exactly once.** *(The Traveling Salesperson/United States Map Problem that we talked about in class is a similar problem, except in Hamilton we have to start/end at the same point, and in Hamilton we aren't trying to find the minimum distance such path.)*

   Examples:

   Graph 1:

   Graph 2:

   - Graph 1 (above left) has several Hamiltonian cycles *(suggestion: consider why it is that it has several)*. One starts and ends at vertex A, so you could return this vector of vertices: `{A,B,D,E,C,A}`. Another is `{B,D,E,C,A,B}`. You'll find that these paths visits every vertex in the graph exactly once (except the starting/ending vertex, which is visited twice), but not all edges are used.
   - Graph 2 (above right) does **not** contain a Hamiltonian cycle, so you should return an empty vector: `{}`. Graph 2 does contain a cycle, but it does not visit every node. It also contains a path that visits every node, but the path is not a cycle (does not start and end at the same node).

   Implementation notes:

   - If the graph **does not contain a Hamiltonian cycle**, or if the graph does not contain any vertices, your function should return an empty vector.
   - If the graph **contains multiple Hamiltonian cycles**, you may return any one of them.
   - Although there are other ways to do this, **you must write a solution that explores possible paths in the graph using recursive backtracking** (*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.
   - As usual, you must match the provided function signature, but you may define helper functions as needed. You should **not** modify the structure 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. If you need more space, use the back of an exam page. As usual, you are free to write helper function(s).

[1] Hamiltonian cycle is a real thing! I didn't make it up just because I am a Hamilton fan. ☺

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

---

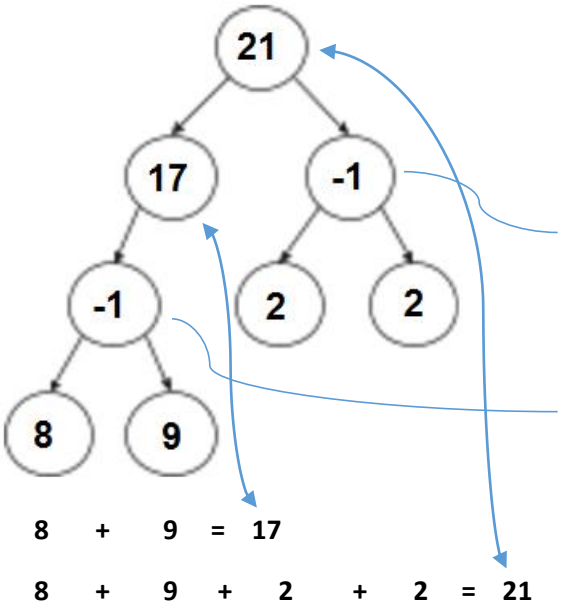
6. **Trees (11pts).** Write a function `isValidSumTree` that takes a binary tree and checks if it meets the requirements to be a valid Sum Tree. A Sum Tree (unlike Hamiltonian cycle, this is something I just made up) is a binary tree where each node holds an `int` key that must meet the following conditions:
   - Each **leaf node** must have **a non-negative number** as its key (no constraint other than non-negative).
   - Each **non-leaf node** (including the root, if it is not also a leaf) may have either: (1) a key that is the sum of the keys of all its descendant leaf nodes, or (2) a key of -1 that functions as a sentinel value meaning something like "I decline to state what my sum key would be."

   The descendant leaf nodes of a given node are all the leaves that are reachable as children, or grandchildren, etc., of the given node. Your function should have the following signature: `bool isValidSumTree(SumNode * tree);`

   - Notice that the function signature provided is **not returning the sum** itself. Rather it checks if the tree is a valid Sum Tree, and then returns **true** in the case that it is, or **false** in the case that it is not valid.
   - As usual, you must match the function signature provided, but **you are free to add helper function(s)**.
   - You may assume that the original input `tree` points to the root of a valid binary tree (*i.e.*, a tree where each node has 0, 1, or 2 children). But, of course, the keys stored in it may or may not be valid for a Sum Tree.
   - You **must use recursion** to solve the problem.

   The node is defined as: `struct SumNode { int key; SumNode * left; SumNode * right; };`

   Here is an example of a **valid** Sum Tree:



   - The key 17 could also be replaced by -1, but any value there other than 17 or -1 would be invalid. (The same is true for the key 21.)
   - The leaf nodes (8, 9, 2, 2) are allowed to have any non-negative number (as long as sums above match), but a -1 in a leaf node would be invalid.

```
// Write isValidSumTree here. There is space for a helper function on the next page,
// or you could write it here too if it fits.
bool isValidSumTree(SumNode * tree) {
```

```
// space for a helper function for isValidSumTree
```

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

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

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

class Eliza : public Angelica {
public:
    virtual void m1() {
        cout << "E 1" << endl;
    }

    virtual void m2() {
        cout << "E 2" << endl;
    }

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

class Peggy : public Eliza {
public:
    virtual void m1() {
        m2();
        m3();
    }
    void m2() {
        cout << "P 2" << endl;
    }
};

class Miranda : public Peggy {
public:
    virtual void m1() {
        Eliza::m1();
        cout << "M 1" << endl;
    }
};
```

Now assume that the following variables are defined:

```
Angelica* var1 = new Miranda();
Eliza*  var2 = new Miranda();
Eliza* var3 = new Peggy();
Peggy* var4 = new Peggy();
Peggy* var5 = new Miranda();
Angelica* var6 = new Angelica();
```

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();`	_____
`var1->m3();`	_____
`var2->m1();`	_____
`var2->m2();`	_____
`var3->m1();`	_____
`var3->m2();`	_____
`var4->m2();`	_____
`var4->m3();`	_____
`var5->m1();`	_____
`var5->m2();`	_____
`var5->m3();`	_____
`((Miranda*) var4)->m2();`	_____
`((Miranda*) var1)->m1();`	_____
`((Miranda*) var1)->m2();`	_____
`Peggy* var7 =new Eliza();`	Compiles? (yes/no) _____

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; // O(1)
  int size() const; // O(1)
  int find(char ch) const; // O(N)
  int find(char ch, int start) const; // O(N)
  string substr(int start) const; // O(N)
  string substr(int start, int length) const; // O(N)
  char& operator[](int index);  // O(1)
  const char& operator[](int index) const; // O(1)
};
string toUpperCase(string str);
string toLowerCase(string str);

class Vector {
  bool isEmpty() const; // O(1)
  int size() const; // O(1)
  void add(const Type& elem); // operator+= used similarly – O(1)
  void insert(int pos, const Type& elem); // O(N)
  void remove(int pos); // O(N)
  Type& operator[](int pos); // O(1)
};

class Grid {
  int numRows() const; // O(1)
  int numCols() const; // O(1)
  bool inBounds(int row, int col) const; // O(1)
  Type get(int row, int col) const; // or operator [][] also works –  O(1)
  void set(int row, int col, const Type& elem); // O(1)
};

class Stack {
  bool isEmpty() const; // O(1)
  void push(const Type& elem); // O(1)
  Type pop(); // O(1)
};

class Queue {
 bool isEmpty() const; // O(1)
 void enqueue(const Type& elem); // O(1)
 Type dequeue(); // O(1)
};
```

```
class Map {
 bool isEmpty() const; // O(1)
 int size() const; // O(1)
 void put(const Key& key, const Value& value); // O(logN)
 bool containsKey(const Key& key) const; // O(logN)
 Value get(const Key& key) const; // O(logN)
 Value& operator[](const Key& key); // O(logN)
};
```
Example for loop: for (Key key : mymap){…}

```
class HashMap {
 bool isEmpty() const; // O(1)
 int size() const; // O(1)
 void put(const Key& key, const Value& value); // O(1)
 bool containsKey(const Key& key) const; // O(1)
 Value get(const Key& key) const; // O(1)
 Value& operator[](const Key& key); // O(1)
};
```
Example for loop: for (Key key : mymap){…}

```
class Set {
 bool isEmpty() const; // O(1)
 int size() const; // O(1)
 void add(const Type& elem); // operator+= also adds elements – O(logN)
 bool contains(const Type& elem) const; // O(logN)
 void remove(ValueType value); // O(logN)
};
```
Example for loop: for (Type elem : mymap){…}

```
class Lexicon {
  int size() const; // O(1)
  bool isEmpty() const; // O(1)
  void clear(); // O(N)
  void add(string word); // O(W) where W is word.length()
  bool contains(string word) const; // O(W) where W is word.length()
  bool containsPrefix(string pre) const; // O(W) where W is pre.length()
};
```
Example for loop: for (string str : english){…}

```
/* GRAPH-RELATED FUNCTIONS */

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; // O(log V)
    bool isNeighbor(string name1, string name2) const; // O(log V)
    const Set<Vertex*> getNeighbors(Vertex* vertex) const; // O(log V)
    const Set<Vertex*> getNeighbors(string vertex) const; // O(log V)
    Edge* getEdge(Vertex* v1, Vertex* v2) const; // O(log V + log E)
    Edge* getEdge(std::string v1, std::string v2) const; // O(log V + log E)
    const Set<Edge*>& getEdgeSet() const; // O(log V)
    const Set<Edge*>& getEdgeSet(Vertex* v) const; // O(log V)
    const Set<Edge*>& getEdgeSet(std::string v) const; // O(log V)
    Vertex* getVertex(std::string name) const; // O(log V)
    const Set<Vertex*>& getVertexSet() const; // O(log V)
    bool containsEdge(Vertex* v1, Vertex* v2) const; // O(log E)
    bool containsEdge(string name1, string name2) const; // O(log E)
    bool containsEdge(Edge* edge) const; // O(log E)
    void resetData(); // O(V + E)

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