FINAL EXAM A

SECOND SEMESTER OF ACADEMIC YEAR 2019 - 2020

STUDENT ID: _____ **NAME:** _____ **CLASS NAME:** _____

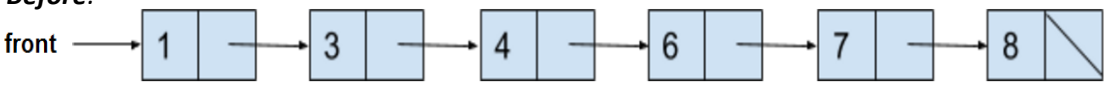
Problem	Linked List	Graphs	Sorting	Big-O	ADTs	Trees	TOTAL
	1	2	3	4	5	6	
Score	15	20	15	15	17	18	100

Instructions:

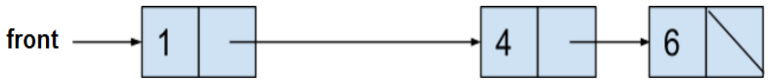
- The time for this exam is ***2.5 hours***. There are ***100 points*** for the exam.
- During the exam, it is forbidden to communicate and discuss with others. Use of anything other than a pencil, pen, notes, and the official textbook is prohibited.
- In particular, before the exam starts, you can download the electronic exam paper through the BB system; after the exam, you can submit the answer file through the BB system.
- During the exam, **in addition to reading the exam paper and asking questions through chat**, it is prohibited to use any Internet-connected devices, such as cell phones, laptops, PCs, and iPads. Otherwise, it will be considered cheating on the exam and will be punished according to the regulations of SHUFE.
- **Answer submission requirements**:
  1) Be sure to create an answer file on your machine**.** The name of the file is named after ***your student ID*** **+** ***your name*** **+** 高级程序设计和实验答卷.docx. For example, if your student ID is 2016000123 and your name is "李明", you need to create a file "***2016000123_李明_高级程序设计和实验答卷.docx***".
  2) You need to submit the answer file through the BB system within 10 minutes after the exam time ends.
- **Please do NOT insert new pages** into the exam. Changing the number of pages in the exam will confuse our examination paper review. You may use the back sides of the exam pages for more space. Thanks.

1、 Linked Lists (15pts). 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、Graphs (20pts).

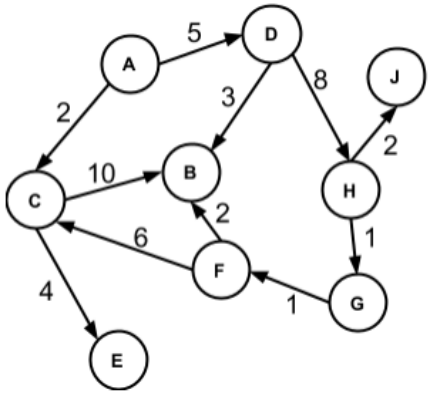
2.1 BFS/DFS (16pts). Show the order of nodes visited when running BFS and DFS, starting at the node labeled A on the right graph. If there is more than one correct solution, write **all correct solutions as possible**.

BFS:

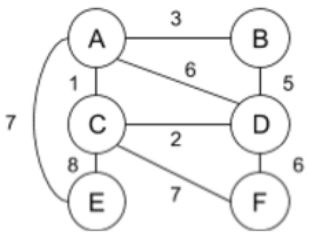
A,_____

DFS:

A,_____



2.2 Kruskal's Algorithm (4pts). You will simulate Kruskal's minimum spanning tree algorithm. 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:

_____

3、Sorting (15pts). 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] = 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 may not need all the provided blank vectors to complete your solution.

9	4	8	5	1	2	3	7	6

4、 Big-O (15pts). 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 remove 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).

Three Questions (5pts each)		Answers

(a)	`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( )

(b)	`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( )

(c)	`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--;
}
```

5、 ADTs (17pts). For this problem, you will write part of the game 2048. The game consists of a Grid of integer tiles and blank spaces (we will represent blank spaces with the integer 0), which we try to aggregate to the sum of 2048 by shifting them left, right, up and down. The function you are to write is ***moveLeft()***, which updates the game board when the user hits the left arrow key (or, on a phone, swipes left). The desired functionality is that number tiles "slide" left across blank spaces as far as they can before colliding into another number tile (or the left edge). If a tile collides with another tile that has the same value, the two are merged into a single tile with the sum of the values.

Example 1:

Before:

2	0	**2**	0	0
0	0	**8**	**8**	0
1	**4**	0	**4**	**1**

After:

4	0	0	0	0
16	0	0	0	0
1	**8**	**1**	0	0

Notes:

- Remember that we are only implementing ***moveLeft()***, so tiles will always slide left.
- In case of more than one possible merge, merges happen with the leftmost pair (see Example 2).
- Each number only merges once per "turn," and doesn't merge again if the new sum matches an adjacent number (see Example 3).
- You must use the function signature provided below.

Example 2:

Before:

2	0	0	**2**	**2**
0	0	0	0	0
0	**8**	**8**	**8**	0

After:

4	**2**	0	0	0
0	0	0	0	0
16	**8**	0	0	0

NOT THIS:

2	**4**	0	0	0
0	0	0	0	0
8	**16**	0	0	0

Example 3:

Before:

2	**2**	0	**2**	**2**
2	0	**2**	**4**	0
4	**0**	**2**	**2**	0

After:

4	**4**	0	0	0
4	**4**	0	0	0
4	**4**	0	0	0

NOT THIS:

8	0	0	0	0
8	0	0	0	0
8	0	0	0	0

You must use the following function signature. It should work for any size/dimensions board.

```
void moveLeft(Grid<int>& board) {


}
```

6、**Trees (18pts).** 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 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."

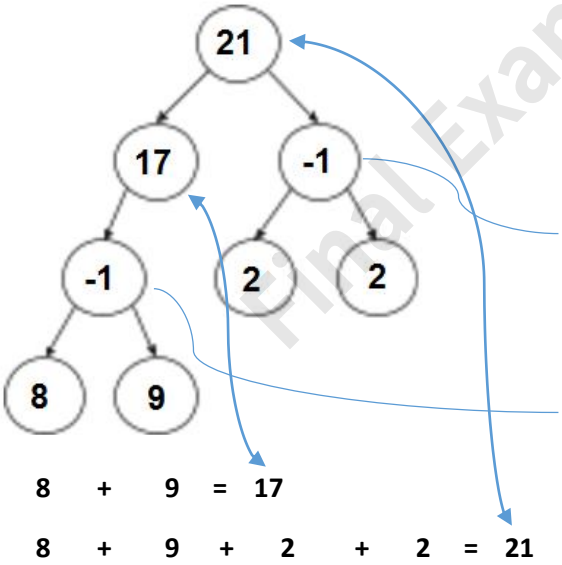
The descendant leaf nodes of a given node are all the leaves that are reachable as children, or grandchildren, etc., of the given node. Your function should have the following signature:

```
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.

```
bool isValidSumTree(SumNode * tree) {

}

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

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){…}
```