FINAL EXAM A

SECOND SEMESTER OF ACADEMIC YEAR 2021 - 2022

STUDENT ID: __________ NAME: __________ CLASS NAME: __________

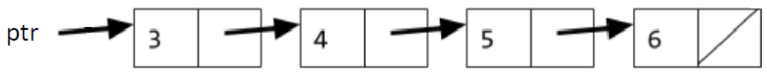
Problem	**LinkedList**	**Graphs**	**Big-O**	**ADT-1& Recursion**	**ADT-2& Recursion**	**Trees& Recursion**	**TOTAL**
	1	2	3	4	5	6	
Score	**17**	**15**	**18**	**15**	**17**	**18**	**100**

Instructions:

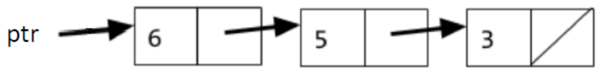
- The time for this exam is ***2.5 hours***. There are ***100 points*** for the exam.
- During the exam, it is forbidden to communicate and discuss with others. Use of anything other than a pencil, pen, notes, and the official textbook is prohibited.
- In particular, before the exam starts, you can download 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*** + 高级程序设计与实验答卷.docx. For example, if your student ID is **2021000123** and your name is "**李明**", you need to create a file "***2021000123_李明_高级程序设计与实验答卷.docx***" or "***2021000123_李明_高级程序设计与实验答卷.pdf***".
  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 (17pts).** 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 (`ptr`) 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:



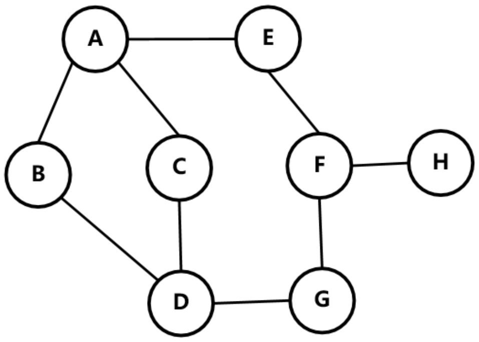
After:



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

2. **Graphs (15pts).**

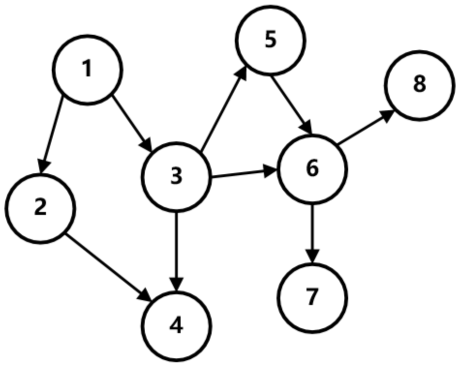
2.1 BFS (7pts). Show the order of nodes visited when running BFS, starting at the node labeled A on the right graph. If there is more than one correct solution, write **all correct solutions as possible**.



BFS:

A, ______________________________

2.2 DFS (8pts). Show the order of nodes visited when running DFS, starting at the node labeled 1 on the right graph. If there is more than one correct solution, write **all correct solutions as possible.**



DFS:

1, ______________________________

3. **Big-O (15pts).** Give a tight bound for the nearest runtime complexity class (worst-case) for each of the following code fragments. Your answer should use the Big-O notation in terms of variable N. (The variable N appears in the code.)
   As a reminder, when using the Big-O notation, we usually write a simple expression in terms of N, such as O(N), O(N^2) etc. We do not require an exact calculation. Write your answer in the blanks on the right side.
   The following identity may be helpful: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

	Three Questions (6pts each)	Answers
(a)	see code (a) below	O( )
(b)	see code (b) below	O( )
(c)	see code (c) below	O( )

(a)

```
int countEven(const Vector<int> & myvec){
  int N = myvec.size();
  int count = 0;
  countEvenRec(myvec, 0, count);
  return count;
}

void countEvenRec(const Vector<int> & myvec,
                  const int i, int & count){
  if (i == myvec.size())
     return;
  if (myvec[i] % 2 == 0) count++;
  countEvenRec(myvec, i + 1, count);
  return;
}
```

O(   )

(b)

```
void sort(Vector<int> & myvec){
  int N = myvec.size();
  sortRec(myvec, 0);
  return;
}

void sortRec(Vector<int> & myvec, int i){
  if (i == myvec.size()) return;
  for (int j = i + 1; j < myvec.size(); j++){
      if (myvec[i] > myvec[j]){
          int temp = myvec[i]; myvec[i] = myvec[j];
          myvec[j] = temp;
       }
  }
  sortRec(myvec, i + 1);
  return;
}
```

O(   )

(c)

```
int countFold(Vector<int> & myvec){
  int N = myvec.size();
  int count = 1;
  countFoldRec(myvec, count, 1);
  return count;
}
```

O(   )

```
void countFoldRec(Vector<int> & myvec, int& count,
                  int currentSize){
  if (currentSize >= myvec.size())
     return;
  count++;
  countFoldRec(myvec, count, currentSize * 3);
  return;
}
```

4. **ADT-1 & Recursion (15pts).** Write a program ***embeddedWords*** that finds all English words that can be formed by taking some subset of letters in order from a given starting word. For example, given the starting word happy, you can certainly produce the words "***a***", "***ha***", "***hap***", and "***happy***", in which the letters appear consecutively. You can also produce the words "***hay***" and "***ay***", because those letters appear in happy in the correct left-to-right order. You cannot, however, produce the words "***pa***" or "***pap***" because the letters—even though they appear in the word—don't appear in the correct order. The following function returns a set of English words formed by taking some subset of letters in order from a given word.

```
Set <string> embeddedWords(const string& word, const Lexicon& english)
{
     Set <string> solus;
     solus = embeddedWordsRec(word, "", english);
     return solus;
}
```

For example, call ***embeddedWords*****("*happy*", english)** would return a set of values: {"***a***", "***ha***", "***ay***", "***hap***", "***hay***","***happy***"}.

Please note that in this implementation function ***embeddedWords*** calls a helper function ***embeddedWordsRec*** that return a set of English words formed by taking the ***constructed*** part that has already been formed, and plus some subset of letters in order from a given ***remaining*** part. This helper function provides an additional parameter *constructed* which is useful in the following recursive calls.

Please write the function ***embeddedWordsRec*** with a **recursive strategy**.

```
Set <string> embeddedWordsRec(const string& remaining, const string& constructed,
                              const Lexicon& english){


}
```

5. **ADT-2 & Recursion (17pts).** In this problem you will write a function that decides if a given set of words can all be concatenated into a single string. A word can be concatenated with another if and only if the last letter of the first word is the same as the first letter of the second. For example, "***apple***" can be concatenated with "***enchant***", which in turn can be concatenated with "***tennis***". So these three words may be concatenated altogether, forming "***applenchantennis***". Obviously, we do not require the resulting string to be a proper English word.

Here are more examples:

Input: banana, orange, starfruit, berry

Answer: False

Input: map, lease, eraser, pencil

Answer: True

The concatenation is mapencileaseraser

Please write a function that decides, for a list of words given in the vector Words, whether there is a way to concatenate all of them.

a) You may assume each word provided has length at least two.
b) For a "true" instance, each word should be used exactly once in the concatenation.
c) Your function does not need to return the concatenation, nor the order in which the words appear in it.
d) You should feel free to write auxiliary functions.
e) You may use containers from either the Stanford library or C++ STL. You must use one of the two signatures provided.

```
bool concatenate(const vector<string> & Words);   or
bool concatenate(const Vector<string> & words);
```

6. **Trees & Recursion (18pts).** Write a function ***isBalanced*(*TreeNode* root, int tolerance*)** that checks whether the tree denoted by the root is balanced. We define that: a tree is balanced if and only if, for each node in the tree, the difference between **the number of nodes** in its left subtree and its right subtree is less than the given **tolerance(>= 0)**. Full marks require the tree to **be traversed only once**.

```
struct TreeNode
{
    int data;
    TreeNode* left;
    TreeNode* right;
};

bool isBalanced(TreeNode* root, int tolerance)
{


}
```

Please note that to claim a tree is balanced, it **requires all subtrees to be balanced**, not simply the root node. The number of nodes in a tree includes the root node. Having a difference that is equal to the tolerance is also regarded as unbalanced. You may write a helper function if you wish.

Tip: you may start by writing a simple function that counts the number of nodes in a tree.

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){…}`