The Hong Kong University of Science and Technology
Department of Computer Science and Engineering
COMP2012: Object-Oriented Programming and Data Structures
(Spring 2014)

Final Examination (L1 & L2)

Date: 19th May, 2014

Time: 12:30 – 15:30 (3 hours)

Venue: LG4204

- This is a closed-book examination. However, you are allowed to bring with you a piece of A4-sized paper with notes written, drawn or typed on both sides for use in the examination.
- Your answers will be graded according to correctness, efficiency, precision, and clarity.
- During the examination, you must put aside your calculators, mobile phones, tablets and all other electronic devices. All mobile phones must be turned off.
- This booklet consists of single-sided pages. Please check that all pages are properly printed. You may use the reverse side of the pages for your rough work. If you decide to use the reverse side to continue your work, please clearly write down the question number.

Student Name	Solution
English nickname (if any)	
Student ID	
ITSC email	_____ @stu.ust.hk

I have not violated the Academic Honor Code in this examination (signature): _____

Question	**Score / Max. score**
Q1. Implementing a Map Class with Templates and Overloading	/39
Q2. Implementing Radix Sort Using STL	/15
Q3. An Ecosystem Using Inheritance and Polymorphism	/25
Q4. Manipulations on a Family Tree	/21
Total	**/100**

Q1 Implementing a Map Class with Templates and Overloading *[39 marks]*

In this question, you are to implement a template container class, ***Map***. A map stores elements, each of them consisting of a *key* and a mapped *value*. The *key* uniquely identifies the element, while the mapped *value* is a record or object associated with the *key* of the element. The ***Map*** class is an unordered container, i.e. the elements are not sorted in any order (not even according to their *keys* or mapped *values*). Each element of the ***Map*** container is an object of the ***KeyValuePair*** template class.

Below is an example of creating and using a ***Map*** object called ***students***, whose *keys* are integers and *values* are strings:

```
Map<int, string> students; // creating an empty students map
students.insert(19, "Mary Lee"); // Mary Lee (value) is of key 19
students.insert(21, "Kenny Wong"); // Kenny Wong (value) is of key 21
students.insert(21, "John Wong"); // Replacing Kenny Wong by John Wong
students[19] = "Ann Wong";  // Replacing Mary Lee by Ann Wong
students.insert(1, "Candy Choi"); // Adding Candy Choi of key 1
cout << "Capacity of students map: " << students.getCapacity() << endl;
cout << "Number of elements: " << students.size() << endl;
string temp = students[19]; // get the value with key 19
cout << "The mapped value for the key 19 is "  << temp << endl;
```

The students map should now consist of `Ann Wong (key 19)`, `John Wong (key 21)` and `Candy Choi (key 1)`. Therefore, the output of the above code is:

```
Capacity of students map: 4
Number of elements: 3
The mapped value for the key 19 is Ann Wong
```

Here is another example of creating and using a ***Map*** object called ***bookPrices***, whose *keys* are strings and *values* are doubles:

```
Map<string, double> bookPrices;
bookPrices.insert("C++ Programming", 240.5);
KeyValuePair<string, double> newBook;  // an element in map
newBook.key = "Object-oriented Programming";
newBook.value = 329.9;
bookPrices = bookPrices + newBook; // insert newBook
cout << bookPrices["Object-oriented Programming"] << endl;
```

The ***bookPrices*** should now consist of two elements (`"C++ Programming" and "Object-oriented Programming"`), and the output of the above code is:

```
329.9
```

Assuming that the key and the mapped value could be any object, below are the definitions of the two template classes, ***KeyValuePair*** and ***Map***:

```
// K is the type of key, V is the type of value


template <class K, class V>
class KeyValuePair {
  public:
    K key;
    V value;
};


template <class K, class V>
class Map {
  public:
    // constructor: constructs an empty map container
    Map();

    // copy constructor: constructs a map container with
    // a copy of each of the elements in other map
    Map(const Map& other) { copy(other); };

    // destructor
    ~Map();

    // returns the number of elements in the map container
    int size() const { return count; }

    // returns the capacity of the map container
    int getCapacity() const { return capacity; }

    // returns true if this map has no elements, false otherwise
    bool isEmpty() const { return (count == 0); }

    // assignment operator: copy the elements from other map
    // to this map container
    const Map & operator=(const Map& other) {
      if (this != &other) copy(other);
      return *this;
    };

    // inserting a key and a mapped value into the map container
    void insert(const K& key, const V& value);

    // + operator overloading: inserting the key and value pair, newPair,
    // into the map container
    Map operator+(const KeyValuePair<K, V>& newPair);

    // subscript operators: returns the associated value of the key
    // found in the map container
    V& operator[](const K& key);
    V operator[](const K& key) const;
```

```
  private:
    KeyValuePair<K, V>* elements; // a dynamic array of key-value
                                  // pairs
    int capacity; // the maximum size of the elements array
    int count;  // the existing number of key-value pairs in the array

    // Helper function to return the array index of the key-value pair
    // in the map given key. Returns -1 if key is not found
    int findKey(const K& key) const;

    // To double the capacity
    void doubleCapacity();

    // To copy the elements from another map called other to the current
    // map container
    void copy(const Map& other);
};
```

(a) ***[3 marks]*** Implement the constructor which constructs an empty map container (i.e., with zero capacity).

```
template <class K, class V>
Map<K, V>::Map(){



   capacity = 0;
   elements = NULL;
   count = 0;




}
```

(b) ***[3 marks]*** Implement the destructor.

```
template <class K, class V>
Map<K, V>::~Map() {
  if (elements!= NULL){
    delete[] elements;
    elements = NULL;
    capacity = 0;
    count = 0;
    }



}
```

(c) ***[4 marks]*** Implement the function ***copy***, which duplicates by making a deep copy of all the elements from another map object, named ***other***, to the current map container. Note that all the elements of the current map container are erased.

```
template <class K, class V>
void Map<K, V>::copy(const Map& other){



      if ((capacity != 0) && (elements!=NULL))
      {
        delete[] elements;
        elements = NULL;
      }
      capacity = other.capacity;
      count = other.count;
      if( other.capacity == 0 )
        return;
      elements = new KeyValuePair<K, V>[capacity];
      for (int i=0; i<other.count; i++)
          elements[i] = other.elements[i];
          // Or by the following two statements
          // elements[i].key = other.elements[i].key;
          // elements[i].value = other.elements[i].value;




}
```

(d) ***[5 marks]*** Implement the private member function ***doubleCapacity***, which expands the container by doubling its capacity with a reallocation of storage space and elements, i.e., copying all the elements over to the new space.

```
template <class K, class V>
void Map<K, V>::doubleCapacity() {

   capacity*=2;

   KeyValuePair<K, V> *newArr = new KeyValuePair<K, V>[capacity];
   for (int i=0; i<count; i++)
      newArr[i] = elements [i];
   delete[]elements;
   elements = newArr;
}
```

(e) ***[5 marks]*** Implement the helper ***findKey*** function which returns the array index of the key-value pair corresponding to a given key value ***key***. If the key does not exist, return -1.

```
template <class K, class V>
int Map<K, V>::findKey(const K& key) const {

   for (int i=0; i<count; i++)
      if (elements[i].key == key)
         return i;
   return -1;
}
```

(f) ***[6 marks]*** Implement the ***insert*** function, which inserts a key, called ***key***, and a mapped value, called ***value***, into the current map container. If the key already exists in the array, overwrite the mapped value; otherwise, add the ***key*** and ***value*** into the current map. If there is not enough capacity, expand the capacity of the array by invoking the ***doubleCapacity*** function.

```
template <class K, class V>
void Map<K, V>::insert(const K& key, const V& value) {

  if (isEmpty()){  // empty map
    capacity = 1;
    elements = new KeyValuePair< K, V >;
    elements[0].key = key;
    elements[0].value = value;
    count = 1;
    return;
  }

  int index = findKey(key);
  if (index == -1) {
    if (count == capacity) // not enough capacity to insert
       doubleCapacity();
    index = count++;
    elements[index].key = key;
  }
  elements[index].value = value;
}
```

(g) ***[3 marks]*** Implement ***operator+*** which extends the container by **inserting** a given key-value pair called ***newPair*** into the current Map container. If the key already exists in the array, overwrite the mapped value; otherwise, add the key-value pair and double the capacity if necessary.
The ***operator+*** should support concatenation, i.e. `a+b+c`, where `a` is a ***Map*** object, `b` and `c` are ***KeyValuePair*** objects.

```
template <class K, class V>
Map<K, V> Map<K, V>::operator+(const KeyValuePair<K, V>& newPair) {

  insert(newPair.key, newPair.value);
  return (*this);

}
```

(h) ***[4 marks]*** Implement the two subscript operator functions which overloads [] by returning the mapped *value* given the *key* named ***key***. You may assume that ***key*** always exists in the map in this problem.

```
template <class K, class V>
V Map<K, V>::operator[](const K& key) const {

   int index = findKey(key);
   return (elements[index].value);



}


template <class K, class V>
V& Map<K, V>::operator[](const K& key) {

   int index = findKey(key);
   return (elements[index].value);



}
```

(i) ***[2 marks]*** Why do we need the two subscript operator functions above?

There are two sides to an assignment: the left side and the right side. Therefore, the return value could be either lvalue (left side) or rvalue (right side).

If the caller of the subscrip operator is a const object, then we need the const subscript operator as the rvalue. It has to return a value (or constant reference). This is for a constant object.

On the other hand, if the caller of the subscript operator is not a const object, then the non-const subscript operator is invoked as either the lvalue or rvalue depending on which side the caller is at. A constant object cannot call this non-constant subscript operator.

(j) ***[2 marks]*** Given the following statements, which of the above two subscript operator [] functions in (i) is called?

```
Map<string, int> imap;
imap.insert("A", 101);
cout << imap["A"] << endl;
```

```
Non-constant [] function
```
Or
```
V& Map<K, V>::operator[] (K key);
```

(k) ***[2 marks]*** Suppose you would like to overload the **output operator (<<)** for the ***Map*** class to support printing the key-value pairs of the current ***Map*** object. To allow direct access to the private data of the ***Map*** object, the operator overloading function should be a friend to the ***Map*** class. Show the friend declaration or function prototype *within* the ***Map*** class definition to achieve this.

Note that the output operator should support concatenation, i.e.

```
cout << a << 10 << b;
```

where `a` and `b` are map objects.

```
template <class M, class N>
friend ostream& operator<<(ostream &, const Map<M, N>&);
```

Q2 Implementing Radix Sort Using STL *[15 marks]*

You have learned radix sort to sort integers in class. In this problem, you are to use it to sort strings stored in a vector in lexicographical order (i.e., ordering words as in a dictionary).

Assumptions:

- The strings contain only the lower-case English letters (i.e., 'a', 'b'… 'z') with no space in between.
- The strings should be sorted in lexicographical (dictionary) order. For example, the strings "abc", "ab", "bca", "cb", "a" should be ordered as "a", "ab", "abc", "bca", "cb" after sorting.

Sketch of the radix sort algorithm for strings:

Suppose we need to sort a vector of strings: "abc", "ab", "bca", "cb" and "a".

Step 1. Initialization

- We use a vector of queues to facilitate the sorting. There are 27 queues, where queue 0 corresponds to an empty letter, queue 1 corresponds to the letter 'a', queue 2 corresponds to the letter 'b', …, and queue 26 corresponds to the letter 'z'.

Step 2. Placing strings into queues

- Align strings at the left. Let *L* be the maximum length of the strings in the vector. Starting from letter position *L-1*, check the letter (may be empty) on the rightmost unexamined position of each string and place the string into the queue corresponding to that letter (shown in the figures below after first examining the third letter of each string).

Letter Position / String Index	0	1	2
0	a	b	c
1	a	b	
2	b	c	a
3	c	b	
4	a		

Queue Index	Corresponding Letter	Content
0	Empty	a, cb, ab
1	a	bca
2	b	
3	c	abc
…	…	…
26	z	

Step 3. Placing strings back to the vector

- Place the strings in the queues back into the original vector. (For this example, after this first iteration the strings are ordered in the vector as "ab", "cb", "a", "bca", "abc.")

Step 4. Repeat Steps 2 and 3 by examining the next lower letter index until letters on all the positions have been checked.

(a) ***[2 marks]*** For the above example, what is the order of the strings in the vector after the second iteration (i.e., after one more round of queuing and putting the strings back to the vector)?

a, ab, cb, abc, bca

You are to implement the radix sort algorithm for strings. The body of the ***radixSort*** function is partially provided below:

```
inline int qIndex(char ch) {
  return (ch - 'a'+1);
}

void radixSort(vector<string> & strs) {
  vector< queue<string> > queues(27);
  int maxLen = maxLength(strs); // Question(b)
  for (int pos=maxLen-1; pos>=0; pos--) {
    VectorToQueues(strs, queues, pos); // Question(c)
    QueuesToVector(strs, queues);  // Question(d)
  }
}
```

You need to implement the three subroutine functions:

- ***maxLength***, to be implemented in Question (b)
- ***VectorToQueues,*** to be implemented in Question (c)
- ***QueuesToVector,*** to be implemented in Question (d)

(b) ***[3 marks]*** Implement the function ***maxLength***, which takes a vector of strings ***strs*** as input and returns the maximum length of the strings in this vector.

```
int maxLength(const vector<string> & strs) {

    int maxLength = 0;
    for (int i=0; i<strs.size(); i++) {
        if (strs[i].length() > maxLength) {
            maxLength = strs[i].length();
        }
    }
    return maxLength;

}
```

(c) ***[5 marks]*** Implement the function ***VectorToQueues***, which checks the letter at position ***pos*** for each string in the vector ***strs*** and places the strings into the appropriate queue in ***queues*** (Refer to Step 2).

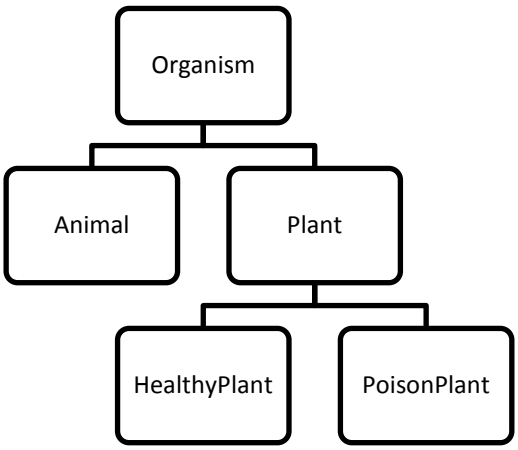
```
void VectorToQueues(vector<string> & strs,
                    vector< queue<string> > & queues, int pos) {

    for (int i=0; i<strs.size(); i++) {
        string str = strs[i];
        int qIdx = 0;
        if (str.length() >= pos+1) {
            qIdx = qIndex(str[pos]);
            // or qIdx = qIndex(str.at(pos))
            // or qIdx = qIndex(*(str.data()+pos));
        }
        queues[qIdx].push(str);
    }

    strs.clear(); //Optional. Case 1: cleared. Case 2: not cleared.



}
```

(d) ***[5 marks]*** Implement the function ***QueuesToVector***, which places strings in ***queues*** back into the vector ***strs*** (Refer to Step 3).

```
void QueuesToVector(vector<string> & strs,
                    vector< queue<string> > & queues) {



    // Case 1: cleared. Push into the vector.
    for (int i=0; i<queues.size(); i++) {
        while ( !queues[i].empty() ) {
            strs.push_back(queues[i].front());
            queues[i].pop();
        }
    }

    // Case 2: not cleared. Overwrite the vector.
    int j = 0;
    for (int i=0; i<queues.size(); i++) {
        while ( !queues[i].empty() ) {
            strs[j++] = queues[i].front();
            queues[i].pop();
        }
    }



}
```

Q3 An Ecosystem Using Inheritance and Polymorphism *[25 marks]*

You are to model an ecosystem consisting of plants and animals, where organisms may grow and become predators of each other (this may be an over-simplified implementation of a computer game). The figure below depicts the inheritance structure of the organism in the ecosystem:



Organism is an abstract class at the root of the hierarchy. Each organism object has its own life point (which is simply its energy or life value). Two member functions, ***setLifePoint*** and ***getLifePoint***, are used to set and get its life point, respectively. For simplicity, you may assume that the life point can be any value – positive, zero or negative. (In practice, there is a garbage collector running in the background removing all the organisms with non-positive life point.)

Organism has 2 subclasses: ***Animal (concrete class)*** and ***Plant (abstract class)***. ***Plant*** has 2 subclasses: ***HealthyPlant*** and ***PoisonPlant.*** Both ***Organism*** and ***Plant*** have a pure virtual function ***nutritionValue***. When an animal eats another organism (may be another animal , healthy or poison plant), the nutrition value of the organism will be added to the life point of the animal.

The relationship between life point and nutrition value is summarized in the following table:

Type of Organism	**Its nutrition value**
Animal	Its own life point
HealthyPlant	Its weight times its life point
PoisonPlant	The negative of the multiplication of its weight, its life point and its degree of poison

We will assume that only ***Anima***l can eat ***Plant*** (and not vice versa). ***Plant*** can conduct photosynthesis to gain its ***weight***. The change of ***weight*** is controlled by the input parameter ***rateOfChange*** (in percentage). For example, if the ***rateOfChange*** is equal to 1.5, the ***weight*** of plant

will be increased by 1.5% every time the photosynthesis function is called. You may assume that the ***rateOfChange*** is always greater than 0.

The following classes are defined in ***ecosystem.h***. Note that you are not allowed to add any other extra member functions:

```
// Class: Organism
const double INIT_LIFE_POINT = 100.0; // default life point = 100

class Organism {
  public:
    Organism(double lp = INIT_LIFE_POINT): lifePoint(lp) {}
    virtual double nutritionValue() const = 0;
    void setLifePoint(double lp) {lifePoint=lp;}
    double getLifePoint() const {return lifePoint; }
  private:
    double lifePoint;
};


// Class: Plant
class Plant: public Organism {
  public:
    Plant(double w, double lp = INIT_LIFE_POINT);
    virtual double nutritionValue() const = 0;
    void photosynthesis(double rateOfChange);
  protected:
    double weight;
};


// Class: HealthyPlant
class HealthyPlant : public Plant {
  public:
    HealthyPlant(double w, double lp = INIT_LIFE_POINT);
    virtual double nutritionValue() const ;
};

// Class: PoisonPlant
class PoisonPlant : public Plant {
  public:
    PoisonPlant(double degOfPoison, double w, double lp = INIT_LIFE_POINT);
    virtual double nutritionValue() const;
  private:
    double degreeOfPoison;
};


// Class: Animal
class Animal: public Organism {
  public:
    Animal(double lp = INIT_LIFE_POINT); // assume it is implemented
    virtual double nutritionValue() const ;
    void eat(const Organism* victim) ;
};
```

You are required to implement the above classes in ***ecosystem.cpp***

(a) ***[6 marks]*** The ***Plant*** class

(i) ***[3 marks]*** Implement the constructor of ***Plant***, which constructs a Plant object with a lifepoint ***lp*** and weight ***w***.

```
Plant::Plant(double w, double lp)
    : Organism(lp), weight(w)
{
}
```

(ii) ***[3 marks]*** Implement the member function ***photosynthesis*** of ***Plant***, which increases the Plant weight by ***rateOfChange***, where ***rateOfChange*** is in percentage.

```
void Plant::photosynthesis(double rateOfChange) {

    weight = weight * (1.0 + rateOfChange/100.0);
}
```

(b) ***[3 marks]*** In the ***HealthyPlant*** class, implement the member function ***nutritionValue*** which returns its nutrition value.

```
double HealthyPlant::nutritionValue() const {
    return getLifePoint()*weight;
}
```

(c) ***[3 marks]*** In the ***PoisonPlant*** class, implement the member function ***nutritionValue*** which returns its nutrition value.

```
double PoisonPlant::nutritionValue() const {
    return -degreeOfPoison*getLifePoint()*weight;
}
```

(d) ***[4 marks]*** An animal object eats a ***victim***, which may point to an animal, healthy or poison plant, by adding its life point with the nutrition value of the ***victim***.

Implement the member function ***eat*** of the Animal class. Note that if the input pointer is *NULL* or points to the animal itself, you should do nothing (an animal cannot eat itself).

```
void Animal::eat(const Organism* victim) {
    if (victim!= NULL && victim!= this) {
        double lp = getLifePoint();
        lp += victim->nutritionValue();
        setLifePoint(lp);
        victim->setLifePoint(0);  // may be set to any negative value
    }
}
```

(e) ***[9 marks]*** In this part of the problem, you are going to write a program applying the class functions.

(i) ***[4 marks]*** Write a function ***animalEatOrganisms***, which has 2 input parameters: ***animal*** and ***orgVec***. Inside this function, all organism objects stored in ***orgVec*** will be eaten by the

animal. Note that the organisms consumed by the ***animal*** should be deallocated/deleted from the memory. The function should clear (making empty) the vector ***orgVec*** before returning.

```
void animalEatOrganisms(Animal* animal, vector<Organism*>& orgVec) {

    for (int i=0; i<orgVec.size(); i++) {
        animal->eat(orgVec[i]);
        delete orgVec[i];
       // If no "delete" is applied
       //  orgVec[i]->setLifePoint(0); // set a zero or negative value

    }
    orgVec.clear();




}
```

(ii) ***[5 marks]*** A crazy monkey eats a banana, an opium and an ant. Their attributes are as follows:

Object Name	**Type**	**Description**
crazyMonkey	Animal	Its initial life point is 110 units

The list of victim is given below:

Object Name	**Type**	**Description**
banana	HealthyPlant	Its initial life point is 10 units and the weight is 5 units
Opium	PoisonPlant	Its initial life point is 6 units, the weight is 10 units and the degree of poison is 2.5 units. Photosynthesis should be conducted on Opium once and its weight should be increased by 10% before it is eaten by the crazy monkey.
ant	Animal	The initial life point is 10 units

The expected output is as follows:

```
crazyMonkey's initial life point : 110
crazyMonkey's final life point: 5
```

Please complete the following main function. Please note that you **MUST** call ***animalEatOrganisms*** defined above for the crazy monkey to eat each of the organisms.

```
int main() {

    vector<Organism*> orgVec;
    Animal *crazyMonkey = new Animal(110.0);

    HealthyPlant *banana = new Plant(5, 10);
    PoisonPlant *opium = new PoisonPlant(2.5, 10, 6);
    Animal *ant = new Animal(10.0);

    opium->photosynthesis(10.0); // +10% weight
    orgVec.push_back(banana);
    orgVec.push_back(opium);
    orgVec.push_back(ant);

    cout << "crazyMonkey's initial life point : "
         << crazyMonkey->getLifePoint() << endl;

    animalEatOrganisms(crazyMonkey, orgVec);

    cout << "crazyMonkey's final life point: "
          << crazyMonkey->getLifePoint() << endl ;









    delete crazyMonkey;
    return 0;
}
```

Q4 Manipulations on a Family Tree *[21 marks]*

A family tree represents descendant and ascendant relationships using a tree structure. We show in Figure 1 below an example of a family tree, where the children of a parent are ordered in descending seniority from left to right. In the figure, George has three children: Sarah, William and David. The most senior child, Sarah has two children, elder Edward and younger Annie. William has a child named Harry, and the youngest son of George, David, has one child named Charles.

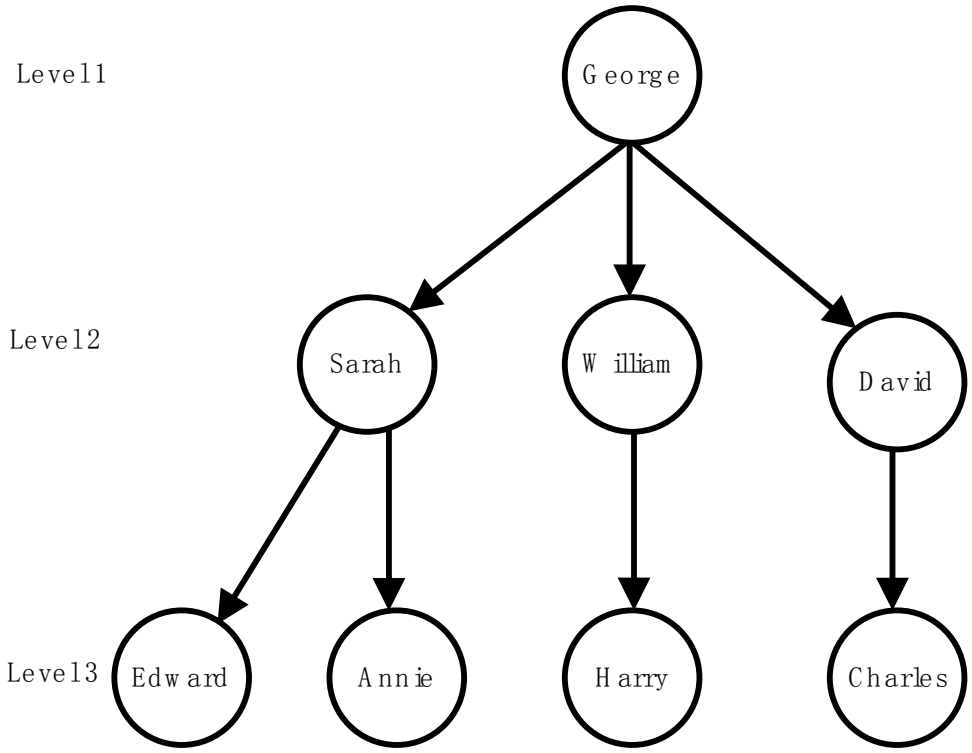


Figure 1 An example of family tree.

You are given ***root***, the pointer to the root of the tree (the eldest ascendant of all nodes) in the family tree. The node structure is the following:

```
struct Node {
  string Name;
  vector<Node *> children;
  int numOfChildren;  // simply the size of children vector
};
```

That is, each node contains a ***Name*** string which is his/her name, an integer field ***numOfChildren*** which is the number of his/her children, and an array (vector) of Node pointers called ***children*** which points to his/her children (if any) in decreasing seniority with array index. For example, in Figure 1, the node George has field values ***Name*** = "George", ***numOfChildren*** = 3, and ***children[0], children[1]*** and ***children[2]*** point to Sarah node, William node and David node, respectively.

(a) ***[6 marks]*** Suppose that you are to transform the family tree above so that the children are ordered according to their *ascending* seniority. For example, Figure 2 shows the transformed family tree from its original one of Figure 1. It is clearly a mirror image of the previous tree:

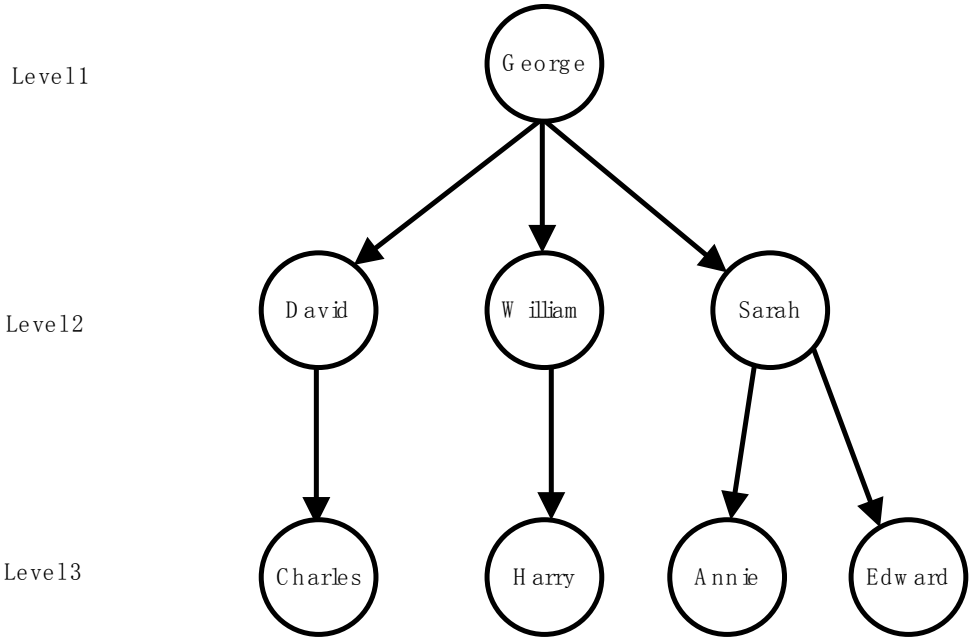


Figure 2 Mirrored Family Tree

Describe below an algorithm which takes in the ***root*** of the tree and changes it to a mirrored tree.

```
void treeMirror(Node* root)
{
   Node* temp;

   // 1pt, If tree is empty, terminate
   if (root == NULL)
      return;

   // 3pt, Recursive call for all children
   for (int i = 0; i < root->numOfChildren; i++)
      treeMirror(root->children[i]);

   // 2pt, Reverse all children in this node
   for (int i = 0, j = root->numOfChildren - 1; i < j; i++, j--)
   {
      temp = root->children[i];
      root->children[i] = root->children[j];
      root->children[j] = temp;
   }
}
```

(b) ***[7 marks]*** Given a family tree and a member's name (assumed valid), you are to print out all his/her ancestors, if any, in reverse chronological order.

For example, given a family tree in Figure 1 and a query name "Charles", the output should be "David George". If the query name is "George", the output should be empty.

Describe an efficient algorithm which achieves the above given the ***root*** of the tree and the *query name*.

```
bool printAncestors(Node* root, string queryName)
{
   // 1pt, If tree is empty, terminate
   if (root == NULL)
      return false;

   // 1pt, Current node matches query name
   if (root->name == queryName)
      return true;

   // 1pt, Recursive call for all children
   for (int i = 0; i < root->numOfChildren; i++)
   {
      // 3pt, If return true, current node is the ancestor
      if (printAncestors(root->children[i], queryName))
      {
         cout << root->name << " ";
         return true;
      }
   }

   // 1pt, Current node is not the ancestor
   return false;
}
```

(c) ***[8 marks]*** We would like to print out the names in the family tree according to the order of the generations (or levels). The output should be in *spiral* order, where the names should be printed from the left to the right on the even levels, and in the opposite direction on the odd levels. We show an example in Figure 3, where the output sequence has been indicated as the dashed line. Accordingly, the output is

"George Sarah William David Charles Harry Annie Edward".

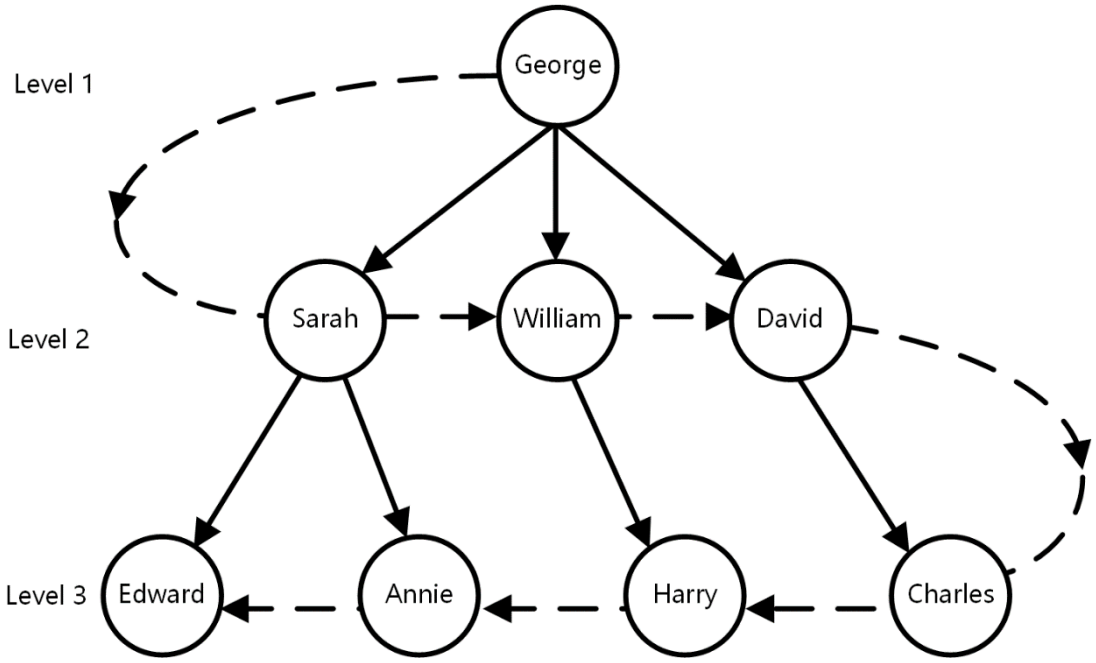


Figure 3 Spiral Traversal of a family tree.

Describe an efficient algorithm which prints out the names in spiral order given the ***root*** to the tree.

```
void spiralLevelOrderTraversal(Node* root)
{
   // 1pt, If tree is empty, terminate
   if(root == NULL)
      return;

   // 2pt, Use two stacks
   stack<Node*> stack1;
   stack<Node*> stack2;

   // 1pt, Insert root and start loop
   stack1.push(root);

   while (stack1.empty() == false || stack2.empty() == false)
   {
      // 2pt, Current odd level
      // Push their children from right to left into stack2
      while (stack1.empty() == false)
      {
         Node* node = stack1.top();
         stack1.pop();
         cout << node->name << " ";
         for (int i = node->numOfChildren - 1; i >= 0; i--)
            stack2.push(node->children[i]);
      }

      // 2pt, Current even level
      // Push their children from left to right into stack1
      while (stack2.empty() == false)
      {
         Node* node = stack2.top();
         stack2.pop();
         cout << node->name << " ";
         for (int i = 0; i < node->numOfChildren; i++)
            stack1.push(node->children[i]);
      }
   }
}
```

=== END OF PAPER ===