Your Name: 孙 [illegible] **Your ID:** 52511592 **HW3** Due date: 11:00 PM, October 21, 2018

1: (3') Complexity relations

$\lim_{n\to\infty} \frac{n^{100}}{2^n} = 0.$

Indicate, for each pair of functions (A,B) in the table below, whether A is O, o, Ω or Θ of B. Your answer should be "yes" or "no" written in each box.

	A	B	O	o	Ω	Θ
1	n^{100}	2^n	Yes	Yes	no	no
2	$\sqrt{n}$	$(lgn)^{12}$	Yes	no	Yes	Yes
3	n^{lgn}	$(lgn)^n$	Yes	no	Yes	Yes
4	2^n	$n^{n/2}$	Yes	no	Yes	Yes
5	$n^{1/2}$	5^{lgn}	Yes	No	Yes	Yes
6	$lg(n!)$	$lg(n^n)$	Yes	Yes	no	no

2: (3') Complexity comparison

Prove that $\log(n!) = O((\log n)^{\log n})$. (Hint: Use the limit properties of the Landau symbols and the knowledge you learn from mathematical analysis.)

$$\lim_{n\to\infty} \frac{\log n!}{(\log n)^{\log n}} = \lim_{n\to\infty} \frac{\log n + \log(n-1) + \cdots + \log 1}{(\log n)^{\log n}}$$

$$= \lim_{n\to\infty} \frac{n \log n}{(\log n)^{\log n}}$$

$$= \lim_{n\to\infty} \frac{n}{(\log n)^{\log n}} = \lim_{n\to\infty} \frac{\log n}{\log n (\log\log n)}$$

$$= \lim_{n\to\infty} \frac{1}{\log(\log n)} = 0.$$

3: (3') Complexity calculation

Solve the following recurrence relation using substitution method (You cannot use the master theorem.). Your solution should be in the form of big theta notation.

$$T(n) = 2T(n/3) + n, T(n) = ?$$

we guess $T(n) = \theta(n)$

$T(n) = \theta(3n)$
$= \theta(n)$

for $k < n$. $T(k) = Cn$

$T(n) = 2T(\frac{n}{3}) + n$

$= 2C \cdot \frac{n}{3} + n$

$= (\frac{2}{3}C + 1) n$

$= Cn$

$\therefore \frac{2}{3}C + 1 = C$

$C = 3.$

4: (3') Complexity analysis

Calculate the **worst-case** time complexity of the following function. The input of this function is an array of n integers.

```
void func(int arr[], int n) {
    int gap, i, j;
    int temp;
    for (gap = n >> 1; gap > 0; gap >>= 1) {
        for (i = gap; i < n; ++i) {
            temp = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > temp; j -= gap)
                arr[j + gap] = arr[j];
            arr[j + gap] = temp;
        }
    }
}
```

$\frac{n}{2}$

$\div 2$ — $\cdot \log_2 \frac{n}{2}$

$\frac{n}{2} \sim n.$

$i: \frac{n}{2}, \frac{n}{2}+1, \cdots n.$

$gap: \frac{n}{2}, \frac{n}{4}, \cdots 0$

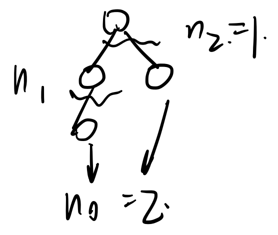
$j: 0, \frac{n}{4}-1, \frac{n}{2}+2-\frac{n}{8}$

$\log_2 \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2}$

$= O(n^2)$

5: (2') Binary tree property

In a rooted tree, the degree of a node is defined as the number of children of the node. For a binary tree, the degree is at least 0 and at most 2. We denote n_0 to be the number of nodes with degree 0, n_1 to be the number of nodes with degree 1 and n_2 to be the number of nodes with degree 2. Justify whether $n_0 = n_2 + 1$.



this is true.

suppose the numbers of nodes is n

$$n = n_1 + n_2 + n_0$$

suppose the total degree is N

$$N = 2n_2 + n_1$$

and we can know that $N+1 = n$

then we get $2n_2 + \cancel{n_1} + 1 = \cancel{n_1} + n_2 + n_0$

So: $n_0 = n_2 + 1$

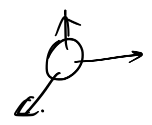
6: (3') Sibling Implementation

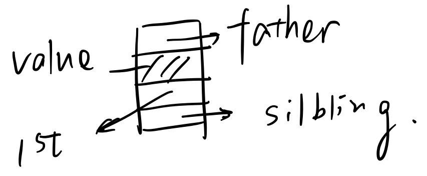
Recall that in the slides, professor mentioned an alternative tree implementation - the left-child right-sibling tree. In this problem, you need to implement the insertions and a breadth-first traversal of the tree. Insertions include inserting a sibling and a child. The traversal part is implemented using queue. The parameter *root* may be empty. For the root node, you can assume that it does not have siblings. The definitions of *TreeNode* and *Queue* has been provided in the template for you.

- You need *this* pointer in *C++*
- *insertSibling* inserts a new node as current object's *next_sibling*
- *insertChild* inserts a new node as current object's *first_child*
- You should exactly use up blank lines

Fill in the blanks of the following code.

```
template<class T>
class Queue
{
    // private members ...
public:
    Queue();
    ~Queue();
    void push(const T &x);
    T pop();
    bool empty() const;
};
template<class T>
class TreeNode
{
private:
    T value;
    TreeNode<T> *father, *first_child, *next_sibling;
public:
    TreeNode(const T &x = T())
        : value(x), father(nullptr),
          first_child(nullptr), next_sibling(nullptr) {}
    ~TreeNode() { /* Not required. */ }
    TreeNode<T> *insertSibling(const T &x)
    {
        TreeNode<T> *new_node = new TreeNode<T>(x);
        __new_node->father____ = __root________________;
        __root->first_child->sibling= __new-node_______;
        __new-node->value_____ = __x___________________;
        return new_node;
    }
```

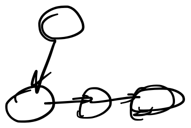




(cont.)

```
    TreeNode<T> *insertChild(const T &x)
    {
        TreeNode<T> *new_node = new TreeNode<T>(x);
        new_node->father = root;
        root->first_child = new_node;
        new_node->value = x;
        return new_node;
    }

    friend void traversePrint(TreeNode<T> *root)
    {
        Queue<TreeNode<T>*> q;
        if (q.empty())
            q.push(root);
        while (!q.empty()) {
            TreeNode<T> *cur = q.pop();
            std::cout << cur->value << '␣';
            for (TreeNode<T> *iter = cur->first_child;
                    iter != NULL;
                    iter = iter->next_sibling)
            {
                std::cout << iter->value << '␣';
            }
        }
    }
};
```



7: (3') Class Inheritance

Dr. Ming invented a new dynamic programming language called C−− and he is writing an interpreter for it. This language is very similar to the combination of C++ and Python, except that it does not allow multiple inheritance from the same base class. Thus, the inheritance graph is a tree.

In C++ and Python, when a function is not directly implemented in the derived class, the compiler or interpreter will search in all the base classes it inherits from. When the function still cannot be found in a base class, the compiler searches the base class's base classes, recursively. Thus, what the compiler or interpreter does looks like a depth-first search.

Dr. Ming would like to invite you to help him implement the member function searching in the inheritance tree of C−−. This search algorithm is implemented using pre-order. If there are multiple results available in the inheritance tree, you should return all of them using a linked list. In this implementation, the function is found when it is equal to the given *func* variable. Note that the equality of two functions (operator==) means they have the same name. The order of the elements in the linked list should be the reverse order of the occurrence in the search. In the search function, an initial list is passed as *start_list*. When the function is found in the search, create a new node and add it to the beginning of the linked list. You also need to return the head of the updated list on return.

In the code template, *functions* is an array whose entries are pointers pointing to *Function* objects. *base_classes* is an array whose entry are pointers pointing to the base *ClassNode* objects. Be careful with pointer operations.

The struct definition of *ClassNode* and *List* has been provided in the template below.

Extended reading material:

Multiple Inheritance in C++: https://www.geeksforgeeks.org/multiple-inheritance-in-c/
Python MRO algorithm: https://www.python.org/download/releases/2.3/mro/

Fill in the blanks of the following code.

```
struct Function {
    std::string name;
    // members omitted
    bool operator==(const Function &rhs) const { return name == rhs.name; }
};
struct ClassNode {
    Function **functions;
    int function_count;
    ClassNode **base_classes;
    int base_class_count;
};
struct List {
    Function *func;
    List *next;
};
```

(cont.)

```
List *find(ClassNode *root, const Function &func, List *start_list)
{
    if (root == NULL)
        return _______________;
    for (Function **iter = root->functions;
          ______________________________________________;
         ++iter)
    {
        if (____________________) {
            List *new_node = new List;
            new_node->func = _______________;
            new_node->next = _______________;
            start_list = _______________;
        }
    }
    for (ClassNode **iter = root->base_classes;
          ______________________________________________;
         ++iter)
    {
        start_list = find(_________________________);
    }
    return _______________;
}
```