**北 京 林 业 大 学**

**18 学年— 19 学年第1 学期 数据结构A 实习报告书**

专 业： 计算机类 班 级： 计算机类17-2 .

姓 名： 庞 博 学 号： 171101215 .

实习地点： 学研 T09 辅导教师： 王少荣 李冬梅 .

实习题目： 北林自助导游及信息服务系统 .

实习环境： Visual Studio 2017 .

实习内容：

[一、实现方法 2](#_Toc534234223)

[1. 校园平面图 2](#_Toc534234224)

[2. 系统的功能模块图 3](#_Toc534234225)

[3. 最短路算法流程图 4](#_Toc534234226)

[二、实验结果 8](#_Toc534234227)

[三、结论分析 8](#_Toc534234228)

[1. 问题与解决方法 8](#_Toc534234229)

[2. 收获和体会 8](#_Toc534234230)

[3. 尚存在的问题 8](#_Toc534234231)

[四、附录 8](#_Toc534234232)

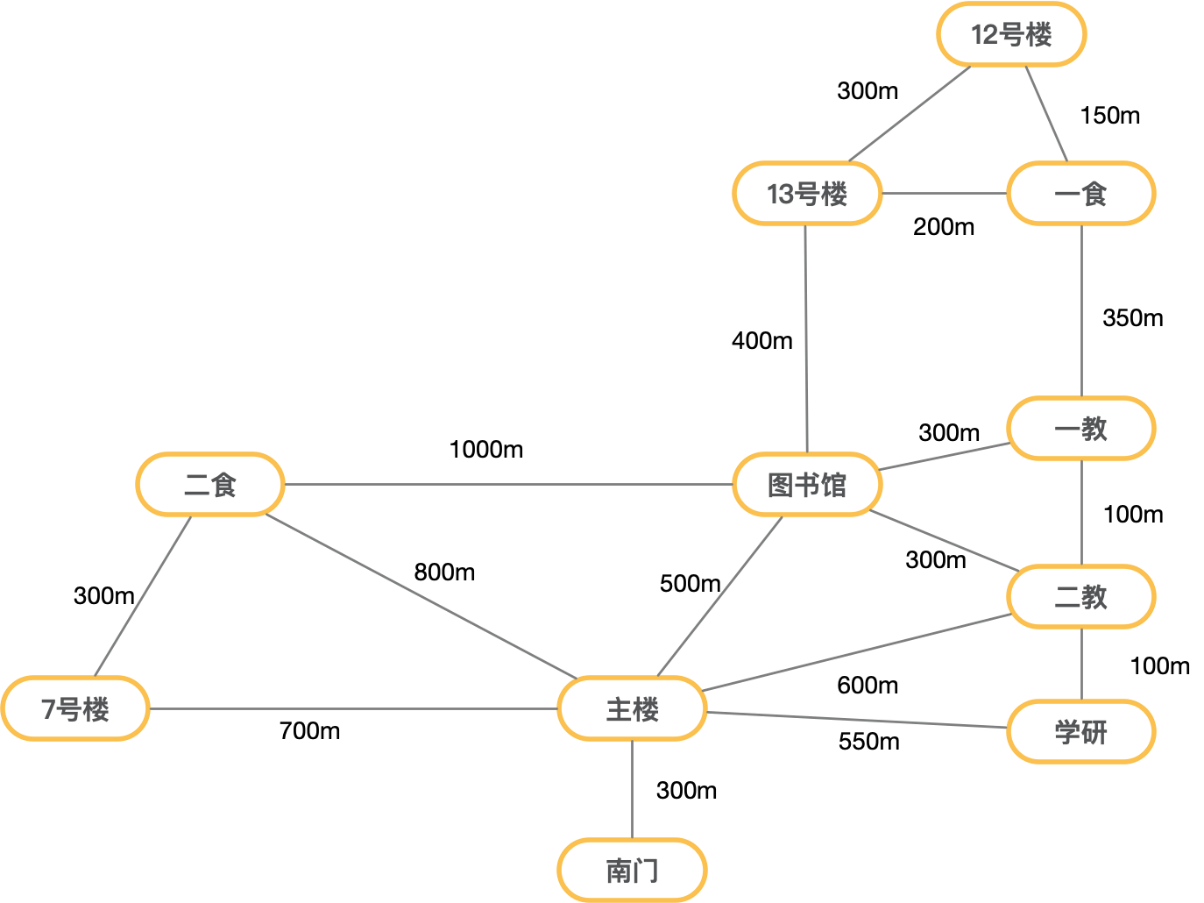
[附录一 Structure::Library API 8](#_Toc534234233)

[附录二 Util API 18](#_Toc534234234)

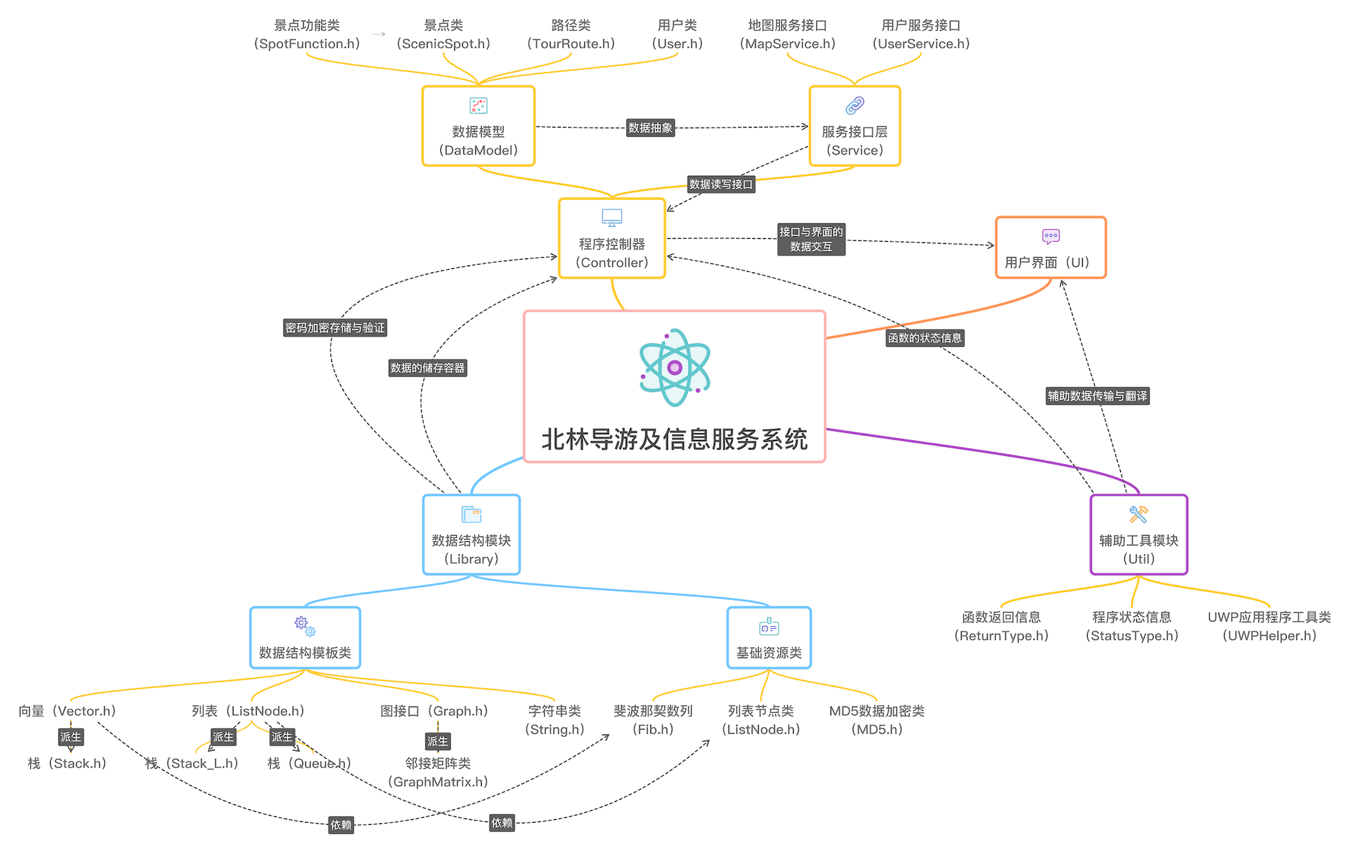
[附录三 Service API 19](#_Toc534234235)

## 一、实现方法

### 1. 校园平面图

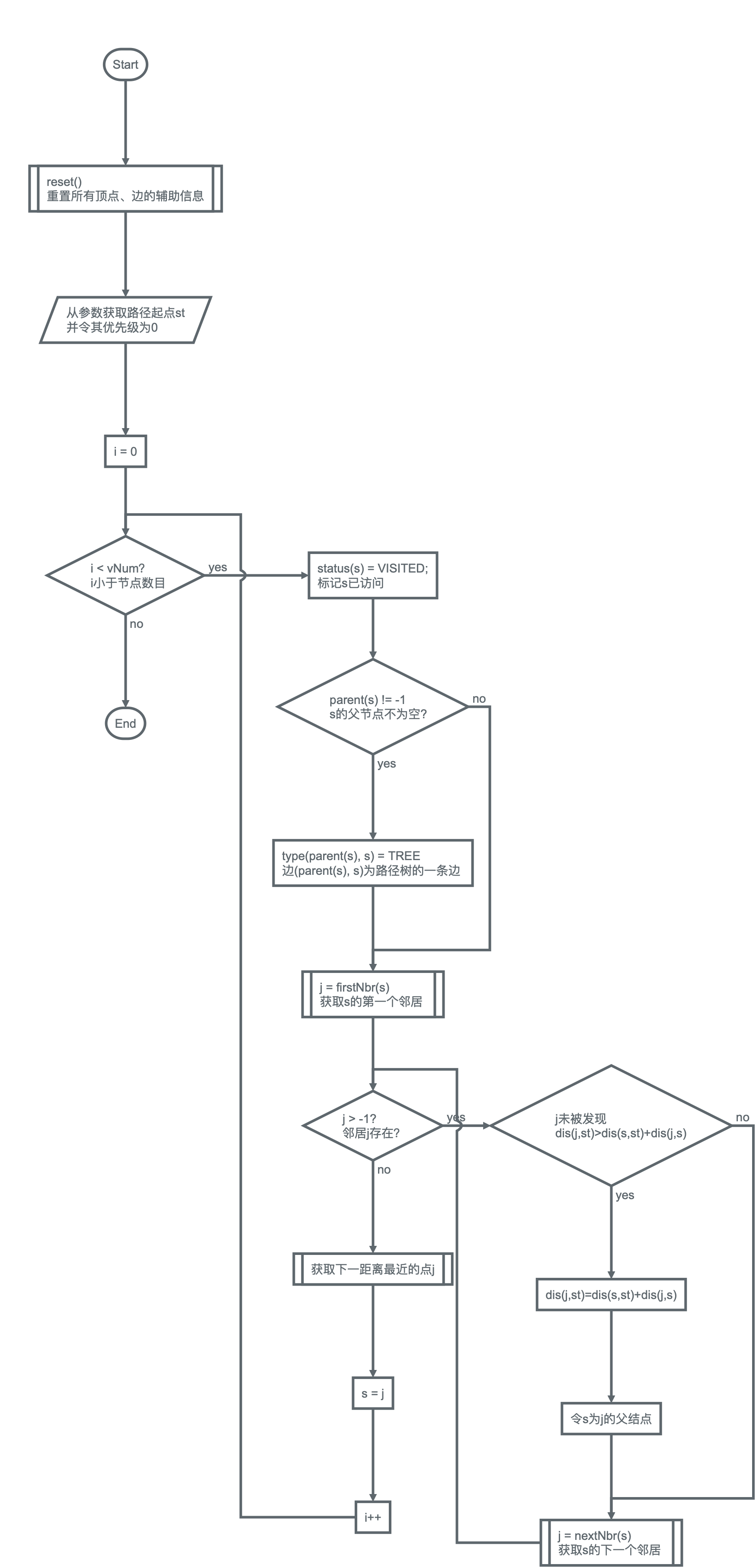


### 2. 系统的功能模块图



* 模块功能说明
  + 数据结构模块（Library）
  + 该模块实现了基本的数据结构，为上层模块提供数据容器，同时实现了快速排序、折半查找、斐波那契查找、BFS算法、Dijkstra算法等重要算法。在本次实习中，我对数据结构层进行了细致的设计。在设计过程中基于C++11标准，使用面向对象的设计思想，应用了泛型编程，实现了一次编写多次运用，加强了代码的可重用性。为了防止命名空间污染，及命名重复引入了Structure命名空间。（Structure::Library API详见 附录一）
  + 辅助工具模块（Util）
  + 该模块包含函数返回和状态信息的枚举类、及用于和Windows API交互的UWP辅助工具类。设计函数返回值和状态信息的模块是为了能够清晰准确语意化的描述程序的状态信息，同时弥补C++异常处理不完善的部分。而UWPHelper主要是进行标准C++与C++/CX语言的数据交换（Util API详见 附录二）
  + 程序控制器模块（Controller）
  + 该模块仅包含两个静态成员变量（基于服务接口模块），用于在UWP中进行全局访问。
    - 数据模型模块（DataModel）
    - 该模块对于项目中如景点、路径、用户等信息进行了数据抽象和严格封装。
    - 服务接口模块（Service）
    - 该模块实现了程序的主要功能，在数据结构模块和数据模型模块的基础上，对于程序的所有功能进行了具体实现，利用辅助工具模块中的枚举类，为上层模块提供程序运行的详细状态信息。Service层不包含任何交互逻辑，仅执行数据操作，修改用户界面不影响整体的操作逻辑。（Service API详见 附录三）
  + 用户界面模块（UI）
  + 该模块本次使用了Microsoft提供的XAML及Windows API实现简单的用户界面，由于程序的底层抽象比较完整，可以很容易的移植到不同的用户界面而不影响程序的整体功能

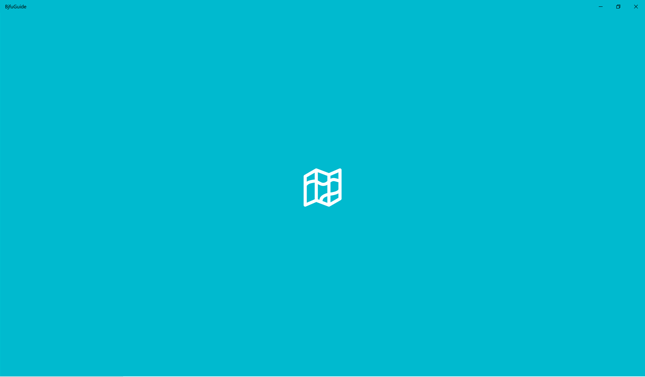
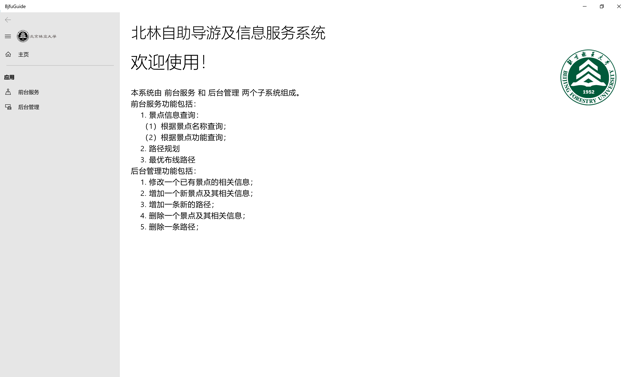
### 3. 最短路算法流程图

* 流程图
* 
* 代码实现
* template <typename Tv, typename Te>  
  void Graph<Tv, Te>::dijkstra(int s)  
  { //assert: 0 <= s < vNum  
   reset();  
   priority(s) = 0;  
   for (int i = 0; i < vNum; i++) { //共需引入n个顶点和n-1条边  
   status(s) = VISITED;  
   if (-1 != parent(s)) {  
   type(parent(s), s) = TREE; //引入当前的s  
   }  
   for (int j = firstNbr(s); - 1 < j; j = nextNbr(s, j)) { //枚举s的所有邻居j  
   if ((status(j) == UNDISCOVERED) && (priority(j) > priority(s) + weight(s, j)))  
   {  
   //对邻接顶点j做松弛  
   priority(j) = priority(s) + weight(s, j);  
   parent(j) = s;  
   }  
   }  
   for (int shortest = INT\_MAX, j = 0; j < vNum; j++) { //选出下一最近顶点  
   if ((status(j) == UNDISCOVERED) && (shortest > priority(j)))  
   {  
   shortest = priority(j);  
   s = j;  
   }  
   }  
   }  
  }
* Structure::Stack<ScenicSpot> MapService::getShortestPath(Rank i,   
   Rank j,   
   int &shortestLength) {  
   Structure::Stack<ScenicSpot> path;  
   \_bjfuMap.dijkstra(i);  
   shortestLength = \_bjfuMap.priority(j);  
   while (i != j) {  
   path.push(\_bjfuMap.vertex(j));  
   j = \_bjfuMap.parent(j);  
   }  
   path.push(\_bjfuMap.vertex(i));  
   return path;  
  }
* Markdown流程图绘制代码
* ```flow  
  st=>start: Start  
  e=>end: End  
    
  io1=>inputoutput: 从参数获取路径起点st  
   并令其优先级为0  
    
  op2=>operation: i = 0  
  op3=>operation: status(s) = VISITED;  
   标记s已访问  
  op4=>operation: type(parent(s), s) = TREE  
   边(parent(s), s)为路径树的一条边  
  op5=>operation: dis(j,st)=dis(s,st)+dis(j,s)  
  op6=>operation: 令s为j的父结点  
  op7=>operation: s = j  
  op8=>operation: i++  
    
  sub1=>subroutine: reset()   
   重置所有顶点、边的辅助信息  
  sub2=>subroutine: j = firstNbr(s)  
   获取s的第一个邻居  
  sub3=>subroutine: j = nextNbr(s)  
   获取s的下一个邻居  
  sub4=>subroutine: 获取下一距离最近的点j  
    
  cond1=>condition: i < vNum？  
   i小于节点数目  
  cond2=>condition: parent(s) != -1  
   s的父节点不为空?  
  cond3=>condition: j > -1?  
   邻居j存在？  
  cond4=>condition: j未被发现  
   dis(j,st)>dis(s,st)+dis(j,s)  
    
  st->sub1->io1  
  io1->op2  
  op2->cond1  
  cond1(yes,right)->op3->cond2  
  cond1(no)->e  
  cond2(yes)->op4->sub2  
  cond2(no)->sub2->cond3  
  cond3(yes,right)->cond4  
  cond3(no)->sub4->op7->op8(left)->cond1  
  cond4(yes)->op5->op6->sub3->cond3  
  cond4(no)->sub3(left)->cond3  
  ```

## 

## 二、实验结果

* 页面展示

****

启动页面 欢迎页面

## 

前台服务页面 后台管理页面

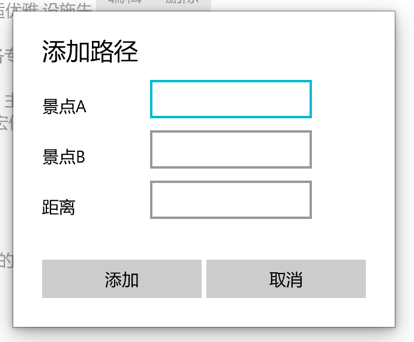
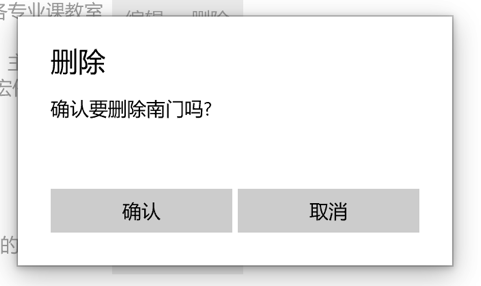
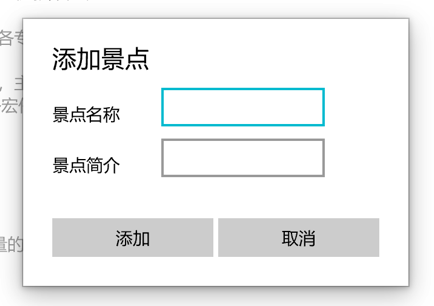
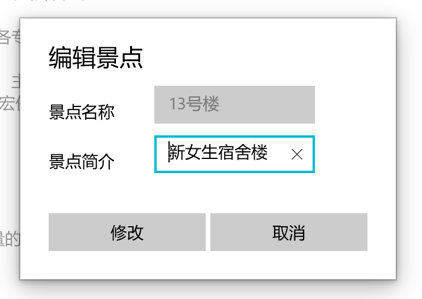
* 功能测试

## 

## 

****

## 

****

## 三、结论分析

### 1. 问题与解决方法

* 问题：
  + 所有类均为.h，Visual Studio提示链接错误：多个类的函数重复声明。
  + 拆分为.h和.cpp，Visual Studio提示编译错误：模板类函数声明找不到
  + 解决方法：
    - C++中普通类声明和实现需要分开写，分别存为.h和.cpp
    - C++中模板声明和实现需要写在一起，如果一定要分开需要在.h中引入.cpp

### 2. 收获和体会

* 提升了C++的运用水平，对于封装、多态、继承、泛型、内存管理、文件读取等C++知识等运用更加的灵活。
* 对本学期学习的各种数据结构和算法有了更深刻的认识。
* 我通过书籍和网络学习到了简单的UWP应用程序的开发方式。

### 3. 尚存在的问题

* 查询所有路径时的DFS算法的设计问题，目前仅此算法在GraphMartix.cpp中实现，暂时无法通过调用Graph.h中已经实现的DFS算法进行所有路径查找
* 用户管理服务类中已经实现了新建用户、修改密码等功能，但由于时间问题UI框架暂未搭建完毕。
* 用户登陆部分，后台进行了密码校验以及异常处理，但密码错误时前台界面暂时无法再次弹出登录框。
  + 根据逻辑要求需要接受对话框的返回结果，登陆界面为Windows::UI::Xaml::Controls::ContentDialog ^类型，Windows UWP应用程序弹出ContentDialog需要进行异步操作，UWP官方推荐的开发语言为C#，以上操作用c#代码表示为：await dialog.ShowAsync();，但是标准C++语言无类似关键字，若要用C++实现异步等待需要使用微软创建的C++/CX，或者在代码中引入C#的dll库。

## 四、附录

### 附录一 Structure::Library API

Fibonacci 数列类

namespace Structure {  
 /\*\*  
 \* Fibonacci数列类  
 \*/  
 class Fib {  
 private:  
 int f, g; //f = fib(k - 1), g = fib(k)。均为int型，很快就会数值溢出  
 public:  
 explicit Fib(int n); // 初始化为不小于n的最小Fibonacci项  
 int get(); // 获取当前Fibonacci项，O(1)时间  
 int next(); // 转至下一Fibonacci项，O(1)时间  
 int prev(); // 转至上一Fibonacci项，O(1)时间  
 }; // Fib  
} // namespace Structure

Vector 向量模板类

namespace Structure {  
 template<typename T>  
 class Vector {  
 protected:  
 Rank \_size; // 向量规模  
 int \_capacity; // 向量容量  
 T \*\_elem; // 数据区  
  
 void copyFrom(T const \*A, Rank lo, Rank hi); // 复制数组区间A[lo, hi)  
 void expand(); // 向量空间不足时扩容  
 void shrink(); //向量空间过大时压缩  
 Rank binSearch(T \*A, T const &e, Rank lo, Rank hi); // 二分查找  
 Rank fibSearch(T \*A, T const &e, Rank lo, Rank hi); // 斐波那契查找  
 Rank partition(Rank lo, Rank hi); // 构造轴点  
 void quickSort(Rank lo, Rank hi); // 快速排序  
  
 public:  
 //构造函数  
 explicit Vector(int c = DEFAULT\_CAPACITY);  
  
 explicit Vector(int c, int s, T v); // 构造函数）  
  
 explicit Vector(T const \*A, Rank n); // 构造函数（拷贝数组整体）  
  
 explicit Vector(T const \*A, Rank lo, Rank hi); // 构造函数（拷贝数组指定区间）  
  
 Vector(Vector<T> const &V); // 构造函数（拷贝向量整体）  
  
 explicit Vector(Vector<T> const &V, Rank lo, Rank hi); // 构造函数（拷贝向量指定区间）  
  
 // 析构函数  
 ~Vector(); // 析构函数  
  
 Rank size() const; // 向量规模  
 bool empty() const; // 判断是否为空  
 int disordered() const; // 有序性判别  
 Rank find(T const &e) const; // 无序向量的查找  
 Rank find(T const &e, Rank lo, Rank hi) const; // 无序向量指定区间的查找  
 Rank search(T const &e) const; // 有序向量的查找  
 Rank search(T const &e, Rank lo, Rank hi) const; // 有序向量指定区间的查找  
  
 T &operator[](Rank r) const; // 重载下标运算符  
 Vector<T> &operator=(Vector<T> const &V); // 重载赋值运算符  
 bool operator==(const Vector &rhs) const; // 重载比较运算符  
 bool operator!=(const Vector &rhs) const; // 重载比较运算符  
 T remove(Rank r); // 删除秩为r的元素  
 int remove(Rank lo, Rank hi); // 删除秩在指定区间的元素  
 Rank insert(Rank r, T const &e); // 在指定位置插入元素  
 Rank insert(T const &e); // 在向量最后插入元素插入元素  
 void sort(Rank lo, Rank hi); // 指定区间排序  
 void sort(); // 整体排序  
 int deduplicate(); // 无序向量的去重  
 int uniquify(); // 有序向量的去重  
  
 // 遍历  
 void traverse(void (\*visit)(T &m)); // 使用函数指针遍历  
 template<typename VST>  
 void traverse(VST &visit); // 使用函数对象遍历  
 }; // Vector  
} // namespace Structure

ListNode 列表节点模板类

namespace Structure {  
 template<typename T>  
 struct ListNode {  
 // 成员  
 T data; // 数据域  
 ListNode<T> \*pre; // 前驱  
 ListNode<T> \*pos; // 后继  
 // 构造函数  
 ListNode() = default; //针对header和trailer的构造  
 explicit ListNode(T e,   
 ListNode<T> \*p = nullptr,   
 ListNode<T> \*s = nullptr);//默认构造函数  
 // 操作接口  
 ListNode<T> \*insertAsPre(T const &e); //紧靠当前节点之前插入新节点  
 ListNode<T> \*insertAsPos(T const &e); //紧随当前节点之后插入新节点  
 }; // ListNode  
} // namespace Structure

List 列表模板类

namespace Structure {  
 class List {  
 private:  
 int \_size; // 规模  
 ListNode<T> \*header; // 头哨兵  
 ListNode<T> \*trailer; //尾哨兵  
  
 protected:  
 void init(); // 列表初始化  
 int clear(); // 清除所有节点  
 void copyNodes(ListNode<T> \*, int); // 复制列表中自位置p起的n项  
 void selectionSort(ListNode<T> \*, int); // 对从p开始连续的n个节点选择排序  
 public:  
 // 构造函数  
 List(); //默认构造函数  
 List(List<T> const &L); //拷贝构造函数（整体拷贝）  
 List(List<T> const &L, Rank r, int n); //拷贝构造函数（列表L中自第r项起的n项）  
 List(ListNode<T> \*p, int n); //拷贝构造函数（复制列表中自p起的n项）  
 // 析构函数  
 ~List(); //释放所有节点  
 // 只读访问接口  
 Rank size() const; // 规模  
 bool empty() const; // 判空  
 T &operator[](Rank r) const; // 重载下表运算符（效率低）  
 ListNode<T> \*first() const; // 首节点指针  
 ListNode<T> \*last() const; // 末节点指针  
 bool valid(ListNode<T> \*p); // 判断位置p是否对外合法  
 int disordered() const; // 判断列表是否已排序  
 ListNode<T> \*find(T const &e) const;// 无序列表查找  
 ListNode<T> \*find(T const &e, int n, ListNode<T> \*p) const; // 无序区间查找  
 ListNode<T> \*search(T const &e) const; // 有序列表查找  
 ListNode<T> \*search(T const &e, int n, ListNode<T> \*p) const; // 有序区间查找  
 ListNode<T> \*selectMax(ListNode<T> \*p, int n); // 在p及其n-1个后继中选出最大者  
 ListNode<T> \*selectMax(); // 选取最大者  
 // 可写访问接口  
 ListNode<T> \*insertAsFirst(T const &e); // 将e当作首节点插入  
 ListNode<T> \*insertAsLast(T const &e); // 将e当作末节点插入  
 ListNode<T> \*insertAfter(ListNode<T> \*p, T const &e); // 将e当作p的后继插入  
 ListNode<T> \*insertBefore(ListNode<T> \*p, T const &e); // 将e当作p的前驱插入  
 T remove(ListNode<T> \*p); // 删除合法位置p处的节点,返回被删除节点  
 void sort(ListNode<T> \*p, int n); // 列表区间排序  
 void sort(); // 列表整体排序  
 int deduplicate(); // 无序去重  
 int uniquify(); // 有序去重  
 void reverse(); // 前后倒置  
 // 遍历  
 void traverse(void (\* )(T &)); // 遍历，依次实施visit操作（函数指针，只读或局部性修改）  
 template<typename VST>  
 void traverse(VST &); // 遍历，依次实施visit操作（函数对象，可全局性修改）  
 }; // List  
} // namespace Structure

Stack 栈模板类（两种实现）

namespace Structure {  
 template<typename T>  
 class Stack : public Vector<T> { //将向量的首/末端作为栈底/顶  
 public:  
 void push(T const &e);// 入栈  
 T pop(); // 出栈  
 T &top(); //取顶  
 }; // Stack  
   
 template<typename T>  
 class Stack\_L : public List<T> { //将列表的首/末端作为栈顶/底  
 public:  
 void push(T const &e);// 入栈  
 T pop(); // 出栈  
 T &top(); //取顶  
 }; // Stack\_L  
} // namespace Structure

Queue 队列模板类

namespace Structure {  
 template <typename T>  
 class Queue : public List<T>  
 {  
 public:  
 void enqueue(T const &e); // 入队  
 T dequeue(); // 出队  
 T &front(); // 队首  
 }; // Queue  
} // namespace Structure

String 字符串类

namespace Structure {  
 class String {  
 protected:  
 char \*\_data; // 字符串内容  
 size\_t \_length; // 字符串长度  
 void copyFrom(const char \*str, size\_t lo, size\_t hi); // 复制字符串区间  
 static int \*buildNext(const char \*str); // 构造next数组  
  
 public:  
 // 构造函数  
 String(const char \*str = "\0"); // 默认构造函数  
 String(const std::string &str); // 拷贝构造函数  
 String(const String &str); // 拷贝构造函数  
 // 析构函数  
 ~String();  
 // 运算符重载  
 String operator+(const String &str) const;  
 String &operator=(const String &str);  
 String &operator+=(const String &str);  
 bool operator==(const String &rhs) const;  
 bool operator!=(const String &rhs) const;  
 bool operator<(const String &rhs) const;  
 bool operator>(const String &rhs) const;  
 bool operator<=(const String &rhs) const;  
 bool operator>=(const String &rhs) const;  
 char &operator[](int n) const;  
 friend std::istream &operator>>(std::istream &is, String &str);  
 friend std::ostream &operator<<(std::ostream &os, const String &str);  
 // 只读操作接口  
 size\_t length() const; // 获取长度  
 const char \*c\_str() const; // 获取C字符串  
 std::string cpp\_str() const; // 转C++字符串  
 int find(const char \*str) const; // 字符匹配  
 int find(const String &str) const; // 字符匹配  
 int find(char ch) const; // 字符匹配  
 char &at(size\_t pos);  
 // 可写操作接口  
 String &append(const String &str); // 后端插入  
 String &append(const char \*str); // 后端插入  
 };  
} // namespace Structure

Graph 图模板类

namespace Structure {  
 typedef enum {  
 UNDISCOVERED, DISCOVERED, VISITED  
 } VStatus; //顶点状态  
  
 typedef enum {  
 UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD  
 } EType; //边在遍历树中所属的类型  
  
 /\*\*  
 \* Graph template class  
 \* 图模板类  
 \* @tparam Tv 顶点类型  
 \* @tparam Te 边类型  
 \*/  
 template<typename Tv, typename Te>  
 class Graph {  
 private:  
 void BFS(int, int &); //（连通域）广度优先搜索算法  
 void DFS(int, int &); //（连通域）深度优先搜索算法  
 // void BCC(int, int &, Stack\_V<int> &); //（连通域）基于DFS的双连通分量分解算法  
 bool TSort(int, int &, Stack\_V<Tv> \*); //（连通域）基于DFS的拓扑排序算法  
 template<typename PU>  
 void PFS(int, PU); //（连通域）优先级搜索框架  
 public:  
 // 顶点  
 int vNum; //顶点总数  
 virtual int insert(Tv const &) = 0; //插入顶点，返回编号  
 virtual Tv remove(int) = 0; //删除顶点及其关联边，返回该顶点信息  
 virtual Tv &vertex(int) = 0; //顶点v的数据（该顶点的确存在）  
 virtual int inDegree(int) = 0; //顶点v的入度（该顶点的确存在）  
 virtual int outDegree(int) = 0; //顶点v的出度（该顶点的确存在）  
 virtual int firstNbr(int) = 0; //顶点v的首个邻接顶点  
 virtual int nextNbr(int, int) = 0; //顶点v的（相对于顶点j的）下一邻接顶点  
 virtual VStatus &status(int) = 0; //顶点v的状态  
 virtual int &dTime(int) = 0; //顶点v的时间标签dTime  
 virtual int &fTime(int) = 0; //顶点v的时间标签fTime  
 virtual int &parent(int) = 0; //顶点v在遍历树中的父亲  
 virtual int &priority(int) = 0; //顶点v在遍历树中的优先级数  
 // 边：这里约定，无向边均统一转化为方向互逆的一对有向边，从而将无向图视作有向图的特例  
 int eNum; //边总数  
 virtual bool exists(int, int) = 0; //边(v, u)是否存在  
 virtual void insert(Te const &, int, int, int) = 0; //在顶点v和u之间插入权重为w的边e  
 virtual Te remove(int, int) = 0; //删除顶点v和u之间的边e，返回该边信息  
 virtual EType &type(int, int) = 0; //边(v, u)的类型  
 virtual Te &edge(int, int) = 0; //边(v, u)的数据（该边的确存在）  
 virtual int &weight(int, int) = 0; //边(v, u)的权重  
   
 void reset(); // 所有顶点、边的辅助信息复位  
 // 算法  
 void bfs(int); //广度优先搜索算法  
 void dfs(int); //深度优先搜索算法  
 Stack\_V<Tv> \*tSort(int); //基于DFS的拓扑排序算法  
 void prim(int); //最小支撑树Prim算法  
 void dijkstra(int); //最短路径Dijkstra算法  
 template<typename PU>  
 void pfs(int, PU); //优先级搜索框架  
 };  
} // namespace Structure

GraphMatrix 邻接矩阵模板类

namespace Structure  
{  
 template <typename Tv>  
 struct Vertex  
 {  
 Tv data; // 数据  
 int inDegree, outDegree; // 出入度数  
 VStatus status; // 状态  
 int dTime, fTime; // 时间标签  
 int parent;  
 int priority; //在遍历树中的父节点、优先级数  
 // 构造函数  
 explicit Vertex(Tv const &d = (Tv)0);  
 };  
 template <typename Te>  
 struct Edge  
 {  
 Te data; // 数据  
 int weight; // 权重  
 EType type; // 类型  
 // 构造函数  
 Edge(Te const &d, int w);  
 };  
  
 template <typename Tv, typename Te>  
 class GraphMatrix : public Graph<Tv, Te> { //基于向量，以邻接矩阵形式实现的图  
 private:  
 Vector<Vertex<Tv>> V; //顶点集（向量）  
 Vector<Vector<Edge<Te> \*>> E; //边集（邻接矩阵）  
 public:  
 // 构造函数  
 GraphMatrix();   
 // 析构函数  
 ~GraphMatrix()；  
 GraphMatrix<Tv, Te> &operator=(const GraphMatrix<Tv, Te> &G)；  
 // 访问接口  
 const Vector<Vertex<Tv>> &getV() const;  
 const Vector<Vector<Edge<Te> \*>> &getE() const;  
 // 顶点的基本操作：查询第i个顶点（0 <= i < vNum）  
 virtual Tv &vertex(int i); // 返回顶点数据  
 virtual int inDegree(int i); // 返回入度  
 virtual int outDegree(int i); // 返回出度  
 virtual int firstNbr(int i); // 找到i的首个邻接顶点  
 virtual int nextNbr(int i, int j); // 逆向线性试探,相对于顶点j的下一邻接顶点  
 virtual VStatus &status(int i); // 返回第i个顶点的状态  
 virtual int &dTime(int i); // 时间标签dTime  
 virtual int &fTime(int i); // 时间标签fTime  
 virtual int &parent(int i); // 在遍历树中的父亲  
 virtual int &priority(int i) ; // 在遍历树中的优先级数  
 // 顶点的动态操作  
 virtual int insert(Tv const &vertex); //插入顶点，返回编号  
 virtual Tv remove(int i); //删除顶点及其关联边，返回该顶点信息  
 // 边的确认操作  
 virtual bool exists(int i, int j); //边(i, j)是否存在  
 // 边的基本操作：查询顶点i与j之间的联边（0 <= i, j < n且exists(i, j)）  
 virtual EType &type(int i, int j); //边(i, j)的类型  
 virtual Te &edge(int i, int j); //边(i, j)的数据（该边的确存在）  
 virtual int &weight(int i, int j); //边(i, j)的权重  
 // 边的动态操作  
 virtual void insert(Te const &edge, int w, int i, int j); //插入权重为w的边e = (i, j)  
 virtual Te remove(int i, int j); //删除顶点i和j之间的联边（exists(i, j)）  
   
   
 // 算法（本处为设计遗留问题，由于封装原因暂时无法在上层实现查询所有路径）   
 Vector<Stack<Tv>> &dfsFindAllPath(int i, int j);  
 void DFS(int i, int j, int &clock, Vector<Stack<Tv>> &allPath, Stack<Tv> &path);  
 }; // GraphMatrix  
} // namespace Structure

MD5加密类（[源自Github](https://github.com/JieweiWei)）

/\* Define of btye.\*/  
typedef unsigned char byte;  
/\* Define of byte. \*/  
typedef unsigned int bit32;  
namespace Structure {  
 class MD5 {  
 public:  
 /\* Construct a MD5 object with a string. \*/  
 MD5(const std::string &message);  
 /\* Generate md5 digest. \*/  
 const byte \*getDigest();  
 /\* Convert digest to string value \*/  
 std::string toStr();  
 private:  
 /\* Initialization the md5 object, processing another message block,  
 \* and updating the context.\*/  
 void init(const byte \*input, size\_t len);  
 /\* MD5 basic transformation. Transforms state based on block. \*/  
 void transform(const byte block[64]);  
 /\* Encodes input (usigned long) into output (byte). \*/  
 void encode(const bit32 \*input, byte \*output, size\_t length);  
 /\* Decodes input (byte) into output (usigned long). \*/  
 void decode(const byte \*input, bit32 \*output, size\_t length);  
 private:  
 /\* Flag for mark whether calculate finished. \*/  
 bool finished;  
 /\* state (ABCD). \*/  
 bit32 state[4];  
 /\* number of bits, low-order word first. \*/  
 bit32 count[2];  
 /\* input buffer. \*/  
 byte buffer[64];  
 /\* message digest. \*/  
 byte digest[16];  
 /\* padding for calculate. \*/  
 static const byte PADDING[64];  
 /\* Hex numbers. \*/  
 static const char HEX\_NUMBERS[16];  
 }; // MD5  
} // namespace Structure

### 

### 附录二 Util API

ReturnType

enum class UserReturnType {  
 SUCCESS, PASSWORD\_WRONG  
};  
  
enum class UserServiceReturnType {  
 SUCCESS, USER\_NOT\_EXIST, USER\_HAS\_EXISTED, PASSWORD\_WRONG  
};  
  
enum class MapServiceReturnType {  
 SUCCESS, SPOT\_NOT\_EXIST, SPOT\_HAS\_EXISTED, ROUTE\_NOT\_EXIST, ROUTE\_HAS\_EXIST  
};

StatusType

enum class UserServiceStatus {  
 DB\_SUCCESS, DB\_ERROR  
};  
  
enum class MapServiceStatus {  
 DB\_SUCCESS, DB\_ERROR  
};

UWPHelper

class UWPHelper {  
public:  
 // 字符串分割  
 static int StringSplit(std::vector<std::string>& dst,   
 const std::string& src,   
 const std::string& separator);  
 // std::wstring转std::string  
 static std::string ws2s(const std::wstring &ws);  
 // std::string转std::wstring  
 static std::wstring s2ws(const std::string &s);  
 // Platform::String转std::string  
 static std::string Platform\_str\_to\_std\_str(Platform::String^ ms);  
 // std::string转Platform::String  
 static Platform::String^ std\_str\_To\_Platform\_Str(const std::string & input);  
 // 获取应用AppData文件目录  
 static const std::wstring getFilePath(const std::string & input);  
};

### 

### 附录三 Service API

MapService

class MapService {  
private:  
 Structure::GraphMatrix<ScenicSpot, TourRoute> \_bjfuMap;  
 MapServiceStatus CurrentStatus;  
public:  
 MapService();  
 virtual ~MapService();  
 MapServiceStatus getCurrentStatus() const;  
// 前台服务  
 Structure::Vector<ScenicSpot> getSpotInfoByName(const Structure::String &spotName);  
   
 Structure::Vector<ScenicSpot> getSpotListByFunction(const Structure::String &function);  
   
 Structure::Stack<ScenicSpot> getShortestPath(const Structure::String &firstPointName,  
 const Structure::String &secondPointName,  
 int &shortestLength);  
   
 Structure::Vector<Structure::Stack<ScenicSpot>> getAllPath(const Structure::String &firstPointName,  
 const Structure::String &secondPointName);  
  
// 后台管理  
 const Structure::Vector<Structure::Vertex<ScenicSpot>> &getV() const;  
 const Structure::Vector<Structure::Vector<Structure::Edge<TourRoute> \*>> &getE() const;  
  
 Structure::GraphMatrix<ScenicSpot, TourRoute> \*get\_bjfuMap();  
 MapServiceReturnType createSpot(const Structure::String &spotName,  
 const Structure::String &spotInfo);  
   
 MapServiceReturnType editSpot(const Structure::String &spotName,  
 const Structure::String &newInfo);  
   
 MapServiceReturnType deleteSpot(const Structure::String &spotName);  
  
 MapServiceReturnType createRoute(const Structure::String &firstPointName,  
 const Structure::String &secondPointName,  
 int distance);  
  
 MapServiceReturnType deleteRoute(const Structure::String &firstPointName,  
 const Structure::String &secondPointName);  
  
 Structure::GraphMatrix<ScenicSpot, TourRoute> \*OptimalRouteDesign();  
  
protected:  
 void dataBaseUpdate();  
  
 Rank getVertexId(const Structure::String &name);  
  
 MapServiceReturnType InsertUndirectedEdges(const Structure::String &firstPointName,  
 const Structure::String &secondPointName,  
 TourRoute distance);  
  
 MapServiceReturnType RemoveUndirectedEdges(const Structure::String &firstPointName,  
 const Structure::String &secondPointName);  
  
 Structure::Stack<ScenicSpot> getShortestPath(Rank i, Rank j, int &shortestLength);  
  
};

UserService

class UserService {  
 private:  
 Structure::Vector<User> users;  
 UserServiceStatus CurrentStatus;  
  
 public:  
 UserService();  
  
 virtual ~UserService();  
  
 UserServiceReturnType login(const Structure::String &username,   
 const Structure::String &password);  
  
 void logout();  
  
 UserServiceReturnType createUser(const Structure::String &username,  
 const Structure::String &password);  
  
 UserServiceReturnType deleteUser(const Structure::String &username,   
 const Structure::String &password);  
  
 UserServiceReturnType changePassWord(const Structure::String &username,   
 const Structure::String &password,  
 Structure::String &newPassword);  
  
 UserServiceStatus getCurrentStatus() const;  
  
 protected:  
 void dataBaseUpdate();  
};