北京郵電大學

《学生游学系统》项目开发文档

——数据结构说明报告

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

小组：第 09 小组

成员：张晨阳 2022211683

廖轩毅 2022211637

徐路 2022211644

2024 年 6 月 9 号

目录

1、Algorithms 类

```
class Algorithms {
   public:
    // 定义结构体存储 Dijkstra 算法的结果，包括路径和路径长度
    struct PathResult {
        std::vector<int> path;  // 存储最短路径的途径点序列
        double length;          // 存储最短路径的总长度
        double time;            // 存储最快路径的总时间

        PathResult()
            : length(0), time(0) {}
    };
};
```

2、Diary 类

```
class Diary {
   public:
    std::string title;              // 日记标题
    std::string author;             // 日记作者
    std::string destination;        // 日记描述对象/地点
    std::string content;            // 日记内容
    int popularity;                 // 日记热度
    int rating;                     // 日记评分
    int score;                      // 日记综合评分

    // 构造函数
    Diary(std::string title, std::string author, std::string destination, std::string
content);

    // 用于打印日记内容
    void DiaryPrint();

    // 用于压缩下载日记
    void DiaryWriteintoFile();
};
```

3、DiaryManager 类

```
class DiaryManager {
   private:
    std::vector<Diary> diaries;  //日记数组

   public:
    // 用于打印所有日记
    void printAllDiaries();

    // 用于添加日记
    void addDiary(Diary diary);

    // 用于获取日记的综合评分
    void getScore(int a, int b);

    // 用于通过综合评分快速排序日记
    void q_sort(int left, int right);

    // 用于搜索日记
    void diarySearch(std::string search_title, std::string search_author, std::string
search_destination, std::string search_content, int search_mode);

    // 用于下载日记
    void diaryDownload();

    // 用于解压日记
    void diaryUncompress(std::string path);

    // 用于增加日记的热度
    int up_popularity(std::string content);

    // 用于更新日记的评分
    int update_rate(std::string content, int new_rating);
};
```

4、Edge 类

```
class Edge {
   public:
    // 交通工具类型
    enum type {
        WALK = 1,           // 步行
        EBIKE = 2,       // 电动车
        BIKE = 3         // 自行车
    };

    Node* source;        // 边的起点节点
    Node* destination;   // 边的终点节点
    double distance;     // 边的长度
    double congestion;   // 边的拥挤度
    type transportMode;  // 边的交通工具类型
    double speed;        // 边的交通工具速度

    // 构造函数
    Edge(Node* source, Node* destination, double distance, double congestion, type
transportMode, double speed);

    // 获取边的起点节点
    Node* getFrom() const;

    // 获取边的终点节点
    Node* getTo() const;

    // 获取边的长度
    double getLength() const;

    // 获取边的拥挤度
    double getCongestion() const;

    // 获取边的交通工具速度
    double getSpeed() const;

    // 获取交通工具类型
    type gettype() const;
};
```

5、Facility 类

```
class Facility {
   private:
    Node location;  // 使用 Node 类型来代表位置
    std::string name;  // 设施名称
    std::string type;  // 设施类别

   public:
    // 构造函数
    Facility(const Node& loc, const std::string& name, const std::string& type);

    // 获取设施名称
    std::string getName() const;

    // 获取设施类别
    std::string getType() const;

    // 获取设施位置
    const Node& getLocation() const;
};
```

6、CodeInfo 结构体

```
// 定义一个名为 CodeInfo 的结构体，用于存储字符的编码和频率信息。
struct CodeInfo {
    // CodeInfo 的构造函数，初始化字符编码为默认值，字符出现次数为 0。
    CodeInfo()
        : code(), cnt(0) {}

    // 重载大于运算符，用于比较两个 CodeInfo 对象的频率 cnt。
    friend bool operator>(const CodeInfo& left, const CodeInfo& right);

    // 重载不等于运算符，用于比较两个 CodeInfo 对象的频率 cnt 是否不相等。
    friend bool operator!=(const CodeInfo& left, const CodeInfo& right);

    // 重载加法运算符，用于合并两个 CodeInfo 对象的频率 cnt。
    friend CodeInfo operator+(const CodeInfo& left, const CodeInfo& right);

    // 字符本身，使用 unsigned char 类型，可以存储 0 到 255 范围内的值。
    unsigned char ch;

    // 该字符的哈夫曼编码，使用 std::string 类型存储。
    std::string code;

    // 该字符出现的次数，使用 long long 类型存储，可以存储非常大的数值。
    long long cnt;
};
```

7、FileCompress 类

```
// 该类包含压缩和解压缩文件的方法
class FileCompress
{
   public:
    // 用于压缩文件
    void Compress(const std::string& FilePath);

    // 用于解压缩文件
    void UnCompress(const std::string& FilePath);

   private:
    // 用于从文件路径中获取文件名
    void GetFileName(const std::string& FilePath, std::string& output);

    // 用于获取扩展名（后缀）
    void GetPostfixName(const std::string& FilePath, std::string& output);

    // 用于填充 info 信息，读取源文件并填充字符频率信息
    void FillInfo(FILE* src);

    // 用于填充编码信息，根据哈夫曼编码填充字符的编码
    void FillCode(const HuffmanTreeNode<CodeInfo>* pRoot);

    // 核心压缩函数
    void CompressCore(FILE* src, FILE* dst, const std::string& FilePath);

    // 用于保存编码信息至压缩文件首部
    void SaveCode(FILE* dst, const std::string& FilePath);

    // 用于从文件中获取一行元素
    void GetLine(FILE* src, unsigned char* buf, int size);

    // 用于从解压缩文件中获取头部编码信息
    void GetHead(FILE* src, std::string& Postfix);

    // 核心解压函数
    void UnCompressCore(FILE* input, FILE* output, HuffmanTreeNode<CodeInfo>* pRoot);

   private:
    CodeInfo info[256];  // CodeInfo 类型数组
};
```

8、Graph 类

```
class Graph {
   public:
    int size;  // 图的大小
    HashMap<int, Node*, HashFunc> nodes;  // 存储图里的所有节点

    // 构造函数
    Graph(int size);

    // 用于添加节点
    void addNode(int id, Node::Type type, const std::string& name, const std::string&
description);

    // 用于添加边
    void addEdge(const int& from, const int& to, double distance, double congestion,
double speed, Edge::type transportMode);

    // 用于获取节点
    Node* getNode(int id);
};
```

9、HuffmanTreeNode 结构体

```
// 定义哈夫曼树节点结构体
template <typename T>
struct HuffmanTreeNode
{
    // 构造函数
    HuffmanTreeNode(const T& data);

    T _weight;      // 节点权重，通常表示字符出现频率
    HuffmanTreeNode* pLeft;  // 指向左子节点的指针
    HuffmanTreeNode* pRight; // 指向右子节点的指针
    HuffmanTreeNode* pParent; // 指向上父节点的指针
};
```

10、greateer 结构体

```
// 定义比较结构体，用于优先队列中比较节点权重
template <typename T>
struct greater
{
    bool operator()(const T& left, const T& right);
};
```

11、HuffmanTree 类

```
// 定义哈夫曼树类
template <typename T>
class HuffmanTree
{
public:
    // 构造函数
    HuffmanTree(const T* weight, int size, const T& invalid);

    // 析构函数，释放哈夫曼树占用的内存
    ~HuffmanTree();

    // 层序遍历哈夫曼树
    void LevelTraverse();

    // 获取哈夫曼树的根节点
    HuffmanTreeNode<T>* GetRoot();

private:
    // 销毁哈夫曼树的辅助函数
    void _Destroy(HuffmanTreeNode<T>*& pRoot);

    // 创建哈夫曼树的辅助函数
    void _Create(const T* weight, int size);

private:
    HuffmanTreeNode<T>* pRoot; // 哈夫曼树的根节点
    T _invalid;                // 无效的权重值，用于标记不使用的字符
};
```

12、LocationQuery 类

```
class LocationQuery {
   private:
    Graph& graph;  // 指向图的指针
    std::vector<Node*> facilities;  // 用于存储设施节点

   public:
    // 构造函数
    explicit LocationQuery(Graph& graph);

    void loadFacilities();
    std::vector<Node*> findNearbyFacilities(Node* location, double radius);
    std::vector<Node*> filterResultsByCategory(std::vector<Node*> results, std::string&
category);
    std::vector<Node*> sortFacilitiesByDistance(std::vector<Node*>& facilities, int low,
int high);
};
```

13、Node 类

```
class Node {
   public:
    // 节点类型枚举
    enum Type {
        BUILDING = 1,  // 建筑、景点、场所
        FACILITY = 2,  // 设施
        NONE = 0
    };

    int id;                    // 节点的唯一标识
    Type type;                 // 节点的类型
    std::string name;          // 节点的名称
    std::vector<Edge*> edges;  // 与该节点相连的边
    std::string description;   // 节点的描述信息
    double distance;           // 与当前搜索位置的距离

    // 构造函数
    Node(int id, Type type, const std::string& name, const std::string& description);

    // 析构函数
    ~Node();  // 释放 edges 中的边指针

    // 用于获取节点 ID
    int getId() const;

    // 用于获取节点名称
    const std::string& getName() const;

    // 用于获取节点类型
    Type getType() const;

    // 用于获取节点描述
    const std::string& getDescription() const;

    // 用于设置节点描述
    void setDescription(const std::string& description);

    // 用于获取与当前位置的距离
    double getDistance() const;

    // 用于设置与当前位置的距离
    void setDistance(double distance);
```

```
    // 用于添加边
    void addEdge(Edge* edge);

    // 静态方法，用于创建表示“空”的 Node 实例
    static Node emptyNode();
};
```

14、View 类

```
class View {
   public:
    int LocationID;    // 景点 ID
    std::string Name;  // 景点名字
    std::string Type;  // 景点类型
    int Popularity;    // 景点热度
    double Ratings;    // 景点评分
    int Score;         // 景点综合评分
};
```

15、ViewManager 类

```
class ViewManager {
   private:
    std::vector<View> views;  // 景区数组

   public:
    void getViews();
    void Recommendation(int obj, int quan, int mo, std::string s_s);
    void getScore(int a, int b);
    void q_sort(int left, int right);
    std::vector<View> selectSort(int length,int obj,std::string search_string);
};
```