

《数据结构》课程设计总结



学 号 1651390

姓 名 刘思源

专 业 信息安全

2019 年 8 月

目录

- 第一部分 算法实现设计说明3
 - 1.1. 题目3
 - 1.2. 软件功能3
 - 1.3. 设计思想3
 - 1.4. 逻辑结构与物理结构8
 - 1.5. 开发平台 13
 - 1.6. 系统的运行结果分析说明 13
 - 1.7. 操作说明 17
- 第二部分 综合应用设计说明 18
 - 2.1 题目 18
 - 2.2 软件功能 18
 - 2.3 设计思想 19
 - 2.4 逻辑结构与物理结构 28
 - 2.5 开发平台 29
 - 2.6 系统的运行结果分析说明 30
 - 2.7 操作说明 32
- 第三部分 实践总结 34
 - 3.1 所做的工作 34
 - 3.2 总结与收获 34
- 第四部分 参考文献 35

第一部分 算法实现设计说明

1.1. 题目

试从空树出发构造一棵深度至少为 3(不包括失误结点)的 3 阶 B-树(又称 2-3 树),并可以随时进行查找、插入、删除等操作

要求:能够把构造和删除过程中的 B-树随时显示输出来,能给出查找是否成功的有关信息。

1.2. 软件功能

软件主要实现的是 B-树的增删查操作,并且要将整个过程可视化的呈现出来,具体的功能可以分为以下几个部分:

- 1、 **B-树的基本操作**。包括 B-树的构建、添加节点、删除节点、查找值。B-树的基本操作也是数据结构设计的一部分,以数据结构的设计为主,实现方法在介绍逻辑结构时介绍。
- 2、 **可视化图形显示**。包括 B-树的结构显示,以及前端后端的交互。在这里本文选择 Java 窗体程序完成前端,由于操作输入都不复杂,操作的数据量也不大,直接在程序内进行数据存储,不添加额外数据库。

1.3. 设计思想

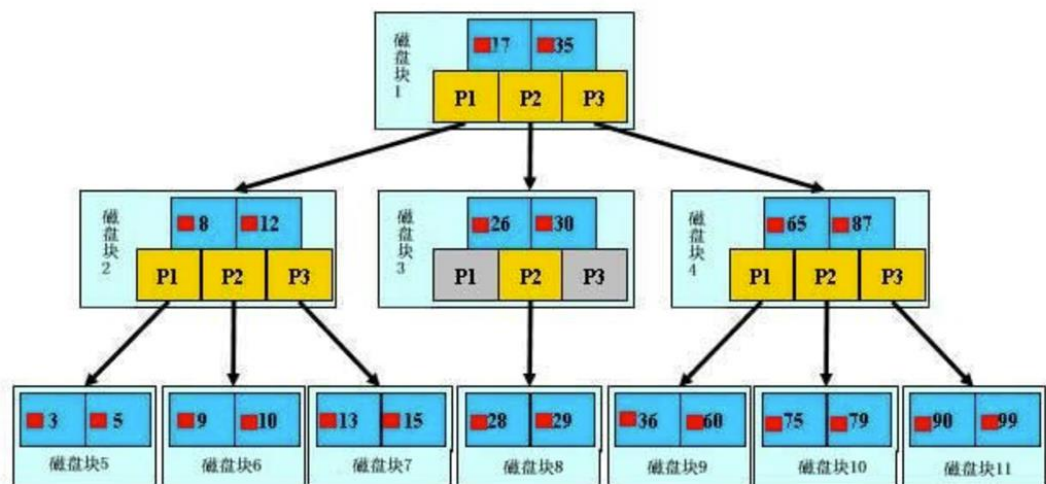
1、 基本原理

B 树是一种查找树,这一类树(比如二叉查找树,红黑树等等)最初生成的目的都是为了解决某种系统中,查找效率低的问题。B 树也是如此,一开始所使用的二叉查找树,二叉查找树的特点是每个非叶节点都只有两个孩子节点。然而这种做法会导致当数据量非常大时,二叉查找树的深度过深,搜索算法自根节点向下搜索时,需要访问的节点也就变的相当多。如果这些节点存储在外存储器中,每访问一个节点,相当于就是进行了一次 I/O 操作,随着树高度的增加,频繁的 I/O 操作一定会降低查询的效率。

对于外存储器的信息读取最大的时间消耗在于寻找磁盘页面。那么一个基本的想法就是能不能减少这种读取的次数,在一个磁盘页面上,多存储一些索引信息。B 树的基本逻辑就是这个思路,它要改二叉为多叉,每个节点存储更多的指针信息,以降低 I/O 操作数。

2、 搜索算法

这里使用一个例子作为解释:



<http://blog.csdn.net/guoziqing506>

图中的小红方块表示对应关键字所代表的文件的存储位置，实际上可以看做是一个地址，比如根节点中 17 旁边的小红块表示的就是关键字 17 所对应的文件在硬盘中的存储地址。P 是指针，需要注意的是：指针，关键字，以及关键字所代表的文件地址这三样东西合起来构成了 B 树的一个节点，这个节点存储在一个磁盘块上

下面，看看搜索关键字的 29 的文件的过程：

从根节点开始，读取根节点信息，根节点有 2 个关键字：17 和 35。因为 $17 < 29 < 35$ ，所以找到指针 P2 指向的子树，也就是磁盘块 3（1 次 I/O 操作）。

读取当前节点信息，当前节点有 2 个关键字：26 和 30。 $26 < 29 < 30$ ，找到指针 P2 指向的子树，也就是磁盘块 8（2 次 I/O 操作）。

读取当前节点信息，当前节点有 2 个关键字：28 和 29。找到了！（3 次 I/O 操作）。

由上面的过程可见，同样的操作，如果使用平衡二叉树，那么需要至少 4 次 I/O 操作，B 树比之二叉树的这种优势，还会随着节点数的增加而增加。另外，因为 B 树节点中的关键字都是排序好的，所以，在节点中的信息被读入内存之后，可以采用二分查找这种快速的查找方式，更进一步减少了读入内存之后的计算时间，由此更能说明对于外存数据结构来说，I/O 次数是其查找信息中最大的时间消耗，而我们要做的所有努力就是尽量在搜索过程中减少 I/O 操作的次数。

3、向 B 树插入关键字

向 B 树种插入关键字的过程与向二叉查找树中插入关键字的过程类似，但是要稍微复杂一点，因为根据上面 B 树的定义，我们可以看出，B 树每个节点中关键字的个数是有范围要求的，同时，B 树是平衡的，所以，如果像二叉查找树那样，直接找到相关的叶子，插入关键字，有可能导致 B 树的结构发生变化而这种变化会使得 B 树不再是 B 树。

所以，我们这样来设计 B 树种对新关键字的插入：首先找到要插入的关键字应该插入的叶子

节点（为方便描述，设这个叶子节点为 u ），如果 u 是满的（恰好有 $2t-1$ 个关键字），那么由于不能将一个关键字插入满的节点，我们需要对 u 按其当前排在中间关键字 $u.key_t$ 进行分裂，分裂成两个节点 u_1, u_2 ；同时，作为分裂标准的关键字 $u.key_t$ 会被上移到 u 的父节点中，在 $u.key_t$ 插入前，如果 u 的父节点未满，则直接插入即可；如果 u 的父节点已满，则按照上面的方法对 u 的父节点分裂，这个过程如果一直不停止的话，最终会导致 B 树的根节点分裂，B 树的高度增加一层。

```

public void insert(K key, E element) {
    Pair<K, E> pair = new Pair<K, E>(key, element);
    if (isEmpty()) {
        root = new BTreeNode<Pair<K, E>>(order);
        root.addKey(0, pair);
        treeSize++;
        root.setFather(nullBTreeNode);
        root.addChild(0, nullBTreeNode);
        root.addChild(1, nullBTreeNode);
        return;
    }

    BTreeNode<Pair<K, E>> currentNode = root;

    if (get(pair.first) != null) {
        replace(key, element);
        return;
    }

    while (!currentNode.isLastInternalNode()) {
        int i = 0;
        while (i < currentNode.getSize()) {
            if (currentNode.isLastInternalNode()) {
                i = currentNode.getSize();
            } else if (currentNode.getKey(i).first.compareTo(pair.first) > 0) {
                currentNode = currentNode.getChild(i);
                i = 0;
            } else {
                i++;
            }
        }
        if (!currentNode.isLastInternalNode())
            currentNode = currentNode.getChild(currentNode.getSize());
    }

    if (!currentNode.isFull()) {
        int i = 0;
        while (i < currentNode.getSize()) {
            if (currentNode.getKey(i).first.compareTo(pair.first) > 0) {
                currentNode.addKey(i, pair);
                currentNode.addChild(currentNode.getSize(), nullBTreeNode);
                treeSize++;
                return;
            } else {

```

```

        i++;
    }
}
currentNode.addKey(currentNode.getSize(), pair);
currentNode.addChild(currentNode.getSize(), nullBTNode);
treeSize++;
} else {
    BTNode<Pair<K, E>> newChildNode = getHalfKeys(pair, currentNode);
    for (int i = 0; i < halfNumber; i++) {
        newChildNode.addChild(i, currentNode.getChild(0));
        currentNode.removeChild(0);
    }
    newChildNode.addChild(halfNumber, nullBTNode);
    BTNode<Pair<K, E>> originalFatherNode = getRestOfHalfKeys(currentNode);
    currentNode.addChild(0, newChildNode);
    currentNode.addChild(1, originalFatherNode);
    originalFatherNode.setFather(currentNode);
    newChildNode.setFather(currentNode);
    treeSize++;

    if (!currentNode.getFather().equals(nullBTNode)) {
        while (!currentNode.getFather().isOverflow()
&& !currentNode.getFather().equals(nullBTNode)) {
            boolean flag = currentNode.getSize() == 1
&& !currentNode.getFather().isOverflow();
            if (currentNode.isOverflow() || flag) {
                mergeWithFatherNode(currentNode);
                currentNode = currentNode.getFather();
                if (currentNode.isOverflow()) {
                    processOverflow(currentNode);
                }
            } else {
                break;
            }
        }
    }
}
}
}
}
}

```

4、从 B 树删除关键字

删除操作的基本思想和插入操作是一样的，都是不能因为关键字的改变而改变 B 树的结构。插入操作主要防止的是某个节点中关键字的个数太多，所以采用了分裂；删除则是要防止某个节点中，因删除了关键字而导致这个节点的关键字个数太少，所以采用了合并操作。基本原则是不能破坏关键字个数的限制；

如果在当前节点中，找到了要删的关键字，且当前节点为内部节点。那么，如果有比较丰满的前驱或后继，借一个上来，再把要删的关键字降下去，在子树中递归删除；如果没有比较丰满的前驱或后继，则令前驱与后继合并，把要删的关键字降下去，递归删除；

如果在当前节点中，还未找到要删的关键字，且当前节点为内部节点。那么去找下一步应该扫描的孩子，并判断这个孩子是否丰满，如果丰满，继续扫描；如果不丰满，则看其有无丰满的兄弟，有的话，从父亲那里接一个，父亲再找其最丰满的兄弟借一个；如果没有丰满的兄弟，则合并，再令父亲下降，以保证 B 树的结构。

```
public void delete(K key) {
    BTreeNode<Pair<K, E>> node = getNode(key);
    BTreeNode<Pair<K, E>> deleteNode = null;
    if (node.equals(nullBTreeNode))
        return;

    if (node.equals(root) && node.getSize() == 1 && node.isLastInternalNode()) {
        root = null;
        treeSize--;
    } else {
        boolean flag = true;
        boolean isReplaced = false;
        if (!node.isLastInternalNode()) {
            node = replaceNode(node);
            deleteNode = node;
            isReplaced = true;
        }

        if (node.getSize() - 1 < halfNumber) {
            node = balanceDeletedNode(node);
            if (isReplaced) {
                for (int i = 0; i <= node.getSize(); i++) {
                    for (int j = 0; j < node.getChild(i).getSize(); j++) {
                        if (node.getChild(i).getKey(j).first.equals(key)) {
                            deleteNode = node.getChild(i);
                            break;
                        }
                    }
                }
            }
        }
    } else if (node.isLastInternalNode()) {
        node.removeChild(0);
    }

    while (!node.getChild(0).equals(root) && node.getSize() < halfNumber &&
flag) {
        if (node.equals(root)) {
```

```

        for (int i = 0; i <= root.getSize(); i++) {
            if (root.getChild(i).getSize() == 0) {
                flag = true;
                break;
            } else {
                flag = false;
            }
        }
    }
    if (flag) {
        node = balanceDeletedNode(node);
    }
}

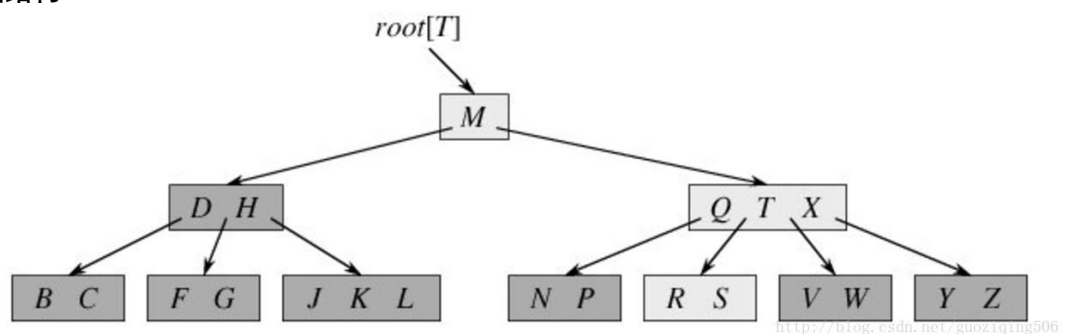
if (deleteNode == null) {
    node = getNode(key);
} else {
    node = deleteNode;
}

if (!node.equals(nullBTNode)) {
    for (int i = 0; i < node.getSize(); i++) {
        if (node.getKey(i).first == key) {
            node.removeKey(i);
        }
    }
    treeSize--;
}
}
}

```

1.4. 逻辑结构与物理结构

1、 数据结构



1. 根节点至少有两个孩子

2. 每个非根节点至少有 $M/2$ (上取整)个孩子,至多有 M 个孩子
3. 每个非根节点至少有 $M/2-1$ (上取整)个关键字,至多有 $M-1$ 个关键字, 并且以升序排列
4. $key[i]$ 和 $key[i+1]$ 之间的孩子节点的值介于 $key[i]$ 、 $key[i+1]$ 之间

采用的数据节点为 key-value 形式:

```
class Pair<A extends Comparable<A>, B> implements Comparable<Pair<A, B>>,
Serializable {
    private static final long serialVersionUID = -8914647164831651005L;

    A first;
    B second;

    Pair(A a, B b) {
        first = a;
        second = b;
    }

    public String toString() {
        if (first == null || second == null)
            return "(null, null)";
        return "(" + first.toString() + ", " + second.toString() + ")";
    }

    @Override
    public int compareTo(Pair<A, B> o) {
        return first.compareTo(o.first);
    }
}
```

```
class BTNode<E extends Comparable<E>> implements Serializable {
    private static final long serialVersionUID = 2631590509760908280L;

    private int fullNumber;
    private BTNode<E> father;
    private ArrayList<BTNode<E>> children = new ArrayList<BTNode<E>>();
    private ArrayList<E> keys = new ArrayList<>();
```

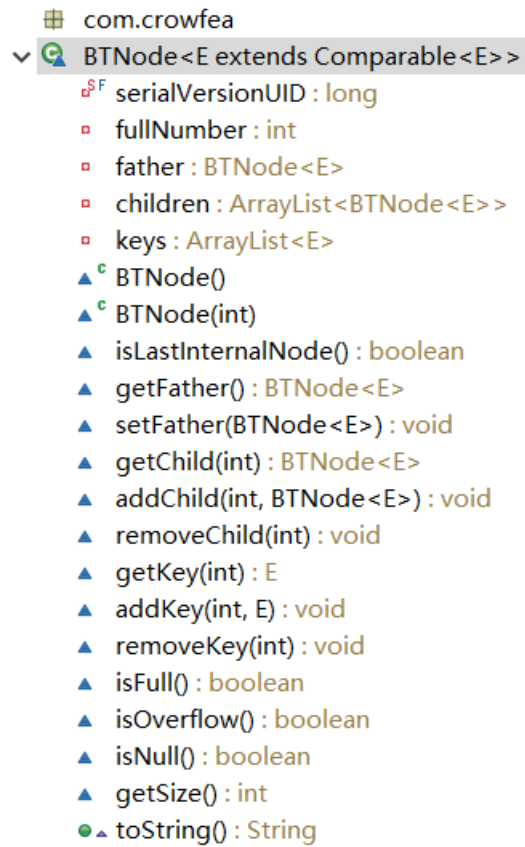
```
public class BTree<K extends Comparable<K>, E> implements Tree, Serializable {
    private static final long serialVersionUID = 1267293988171991494L;
    private BTNode<Pair<K, E>> root = null;
    private int order, index, treeSize;
    private boolean findres;
    private final int halfNumber;
```

```
private final BTNode<Pair<K, E>> nullBTNode = new BTNode<Pair<K, E>>();
```

2、程序组织

数据结构类

BTNode 类：完成树的节点的定义



```
com.crowfea
BTNode<E extends Comparable<E>>
  serialVersionUID : long
  fullNumber : int
  father : BTNode<E>
  children : ArrayList<BTNode<E>>
  keys : ArrayList<E>
  BTNode()
  BTNode(int)
  isLastInternalNode() : boolean
  getFather() : BTNode<E>
  setFather(BTNode<E>) : void
  getChild(int) : BTNode<E>
  addChild(int, BTNode<E>) : void
  removeChild(int) : void
  getKey(int) : E
  addKey(int, E) : void
  removeKey(int) : void
  isFull() : boolean
  isOverflow() : boolean
  isNull() : boolean
  getSize() : int
  toString() : String
```

BTree 类：完成树的定义，以及基本的增删改查功能

```

com.crowfea
└─ BTree<K extends Comparable<K>, E>
    ├── serialVersionUID : long
    ├── root : BTreeNode<Pair<K, E>>
    ├── order : int
    ├── index : int
    ├── treeSize : int
    ├── findres : boolean
    ├── halfNumber : int
    ├── nullBTreeNode : BTreeNode<Pair<K, E>>
    ├── BTree(int)
    ├── isEmpty() : boolean
    ├── getRoot() : BTreeNode<Pair<K, E>>
    ├── getTreeSize() : int
    ├── getFindRes() : boolean
    ├── getHeight() : int
    ├── getHeight(BTreeNode<Pair<K, E>>) : int
    ├── get(K) : Pair<K, E>
    ├── setFind(boolean) : void
    ├── getNode(K) : BTreeNode<Pair<K, E>>
    ├── replace(K, E) : void
    ├── getHalfKeys(Pair<K, E>, BTreeNode<Pair<K, E>>) : BTreeNode<Pair<K, E>>
    ├── getHalfKeys(BTreeNode<Pair<K, E>>) : BTreeNode<Pair<K, E>>
    ├── getRestOfHalfKeys(BTreeNode<Pair<K, E>>) : BTreeNode<Pair<K, E>>
    ├── mergeWithFatherNode(BTreeNode<Pair<K, E>>, int) : void
    ├── mergeWithFatherNode(BTreeNode<Pair<K, E>>) : void
    ├── setSplitFatherNode(BTreeNode<Pair<K, E>>) : void
    ├── processOverflow(BTreeNode<Pair<K, E>>) : void
    ├── insert(K, E) : void
    ├── findChild(BTreeNode<Pair<K, E>>) : int
    ├── balanceDeletedNode(BTreeNode<Pair<K, E>>) : BTreeNode<Pair<K, E>>
    ├── replaceNode(BTreeNode<Pair<K, E>>) : BTreeNode<Pair<K, E>>
    ├── delete(K) : void
    └─ toString() : String

```

前端可视化类

MyCanvas 类：进行前端可视化的作图

```

com.crowfea
└─ MyCanvas
   ├── bTree : BTree<Integer, Double>
   ├── width : int
   ├── height : int
   ├── fontSize : int
   ├── rectangleWidth : int
   ├── MyCanvas(int, int, BTree<Integer, Double>)
   ├── paint(Graphics) : void
   ├── updateCanvas(BTree<Integer, Double>) : void
   ├── DrawNode(Graphics, String, int, int) : void
   └─ DrawBTree(Graphics) : void

```

DrawTree 类：绘制 B-树，显示绘图信息

```

com.crowfea
└─ DrawBTree
   ├── key : int
   ├── canvas : MyCanvas
   ├── keyText : JTextField
   ├── elementText : JTextField
   ├── previousButton : JButton
   ├── nextButton : JButton
   ├── index : int
   ├── bTreeLinkedList : LinkedList<BTree<Integer, Double>>
   ├── bTree : BTree<Integer, Double>
   ├── DrawBTree(BTree<Integer, Double>)
   ├── checkValid() : void
   ├── deleteList() : void
   ├── insertValue() : void
   ├── deleteValue() : void
   ├── findValue() : void
   ├── goPrevious() : void
   └─ goNext() : void

```

组件类

Tree 类：作为 B-树的父类

Pair 类：提供一个 key-value 结构的数据作为节点

1.5. 开发平台

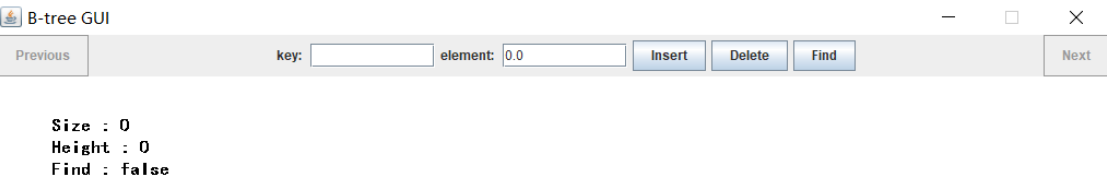
开发平台：Eclipse-2018-12
环境：jre 8
程序支持 jre 1.8 以上

程序 export 出 jar 包，通过 exe4j 编译为可执行程序，其中添加了 jre8 环境。


1.6. 系统的运行结果分析说明

启动程序

此时显示树的规模为 0，高度为 0，没有进行过查询所以 find 缺省为 false。



插入操作

 B-tree GUI

Previous

key:

element:

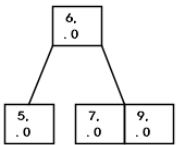
Insert


Delete

Find

Next

Size : 4
Height : 2
Find : false



 B-tree GUI

Previous

key:

element:

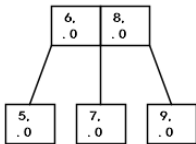
Insert


Delete

Find

Next

Size : 5
Height : 2
Find : false



 B-tree GUI

Previous

key:

element:

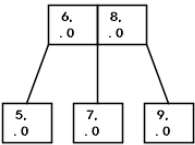
Insert


Delete

Find

Next

Size : 5
Height : 2
Find : false



 B-tree GUI

Previous

key:

element:

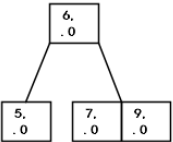
Insert

Delete

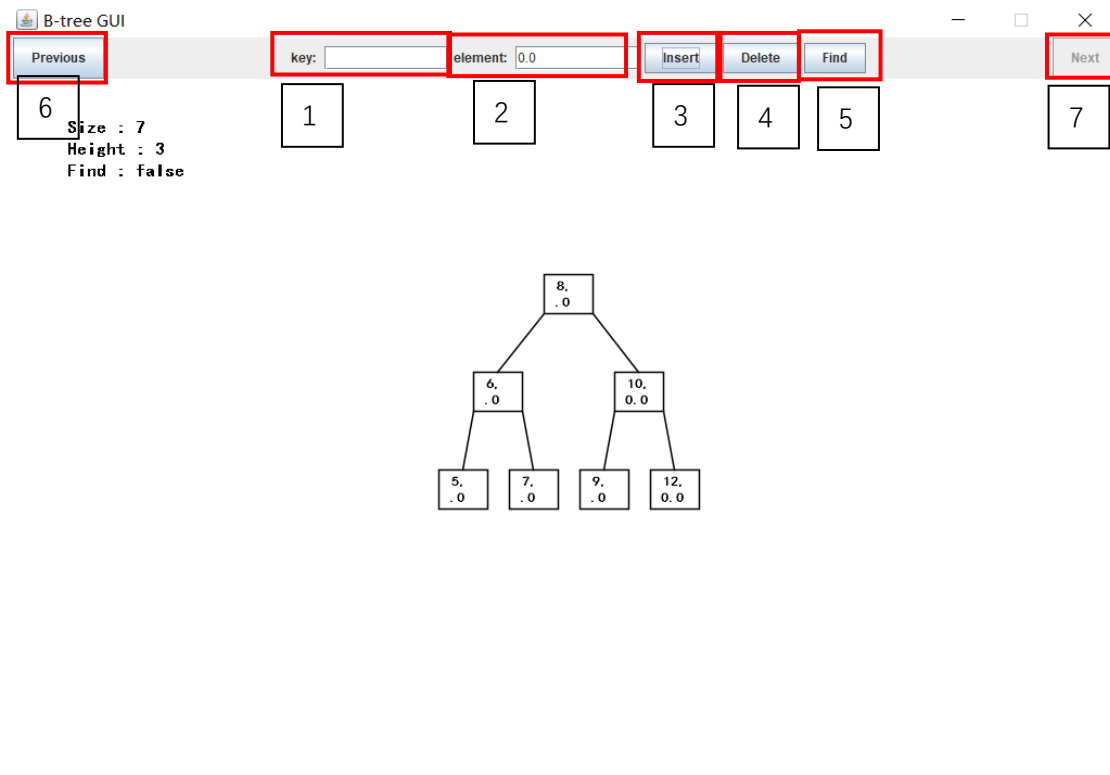
Find

Next

Size : 4
Height : 2
Find : false



1.7. 操作说明



- 1、输入框 key：输入节点的 key 值。节点为 key-value 形式，key 为浮点数。
- 2、输入框 element：输入节点的 element 值。浮点数。
- 3、插入按钮：输入数据后点击 insert 进行插入操作。
- 4、删除按钮：输入 key 值后可以检索完成删除。
- 5、查询按钮：输入 key 值后可以查询该值是否存在，结果显示在左侧的 Find 结果中。
- 6、前一步：可以回调查看操作的前一步树的结构。
- 7、后一步：在回调中查看操作的下一步结果。

Github 地址 (<https://github.com/CrowFea/DataStructureDesgin>)

第二部分 综合应用设计说明

2.1 题目

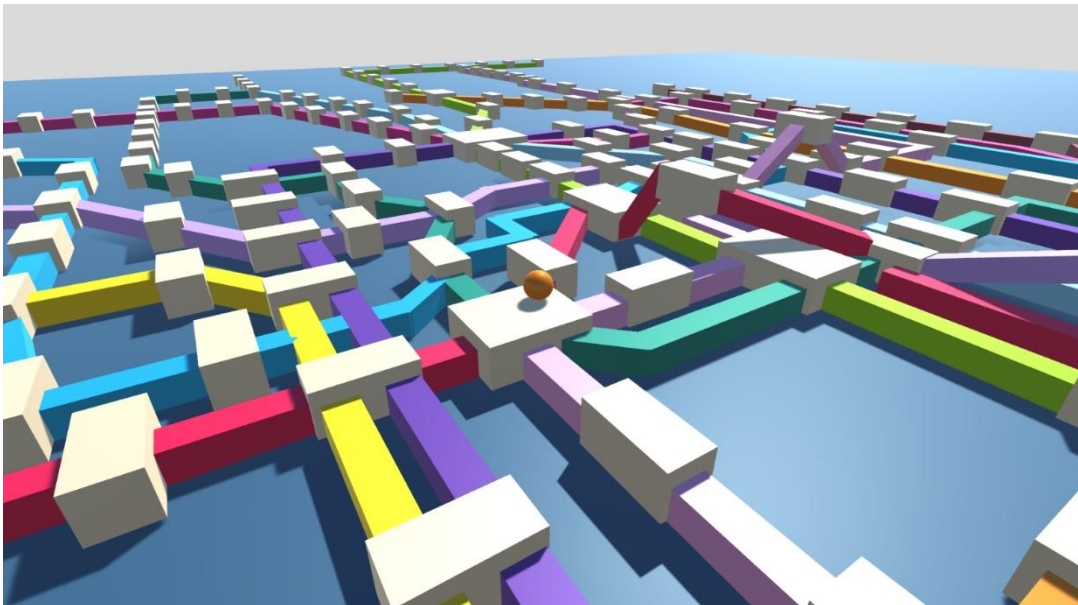
上海的地铁交通网络已基本成型上海的地铁交通网路已基本成型,建成的地铁线十多条,站点上百个,现需建立一个换乘指南打印系统,通过输入起点站和终点站,打印出地铁换乘指南,指南内容包括起点站,换乘站,终点站.

- (1)图形化显示地铁网络结构,能动态添加地铁线路和地铁站点
- (2)根据输入起点站和终点站,显示地铁换乘指南.
- (3)通过图形界面显示乘车路径

2.2 软件功能

软件可实现的功能如下:

- 1、**图形化显示上海地铁**, 本文选取了上海地铁的 1-13 号线。在 unity 中使用 ProBuilder 插件进行 level Design。制作了三维的上海地铁图。



- 2、**进行视角的转换和摄像机控制。**

Unity 采用摄像机作为视角入口, 由于场景复杂且略显庞大, 在使用程序时需要对视角进行调整和转换。需要编写脚本对摄像机进行控制。

- 3、**两种循迹模式。**

本程序完成了两种循迹模式:

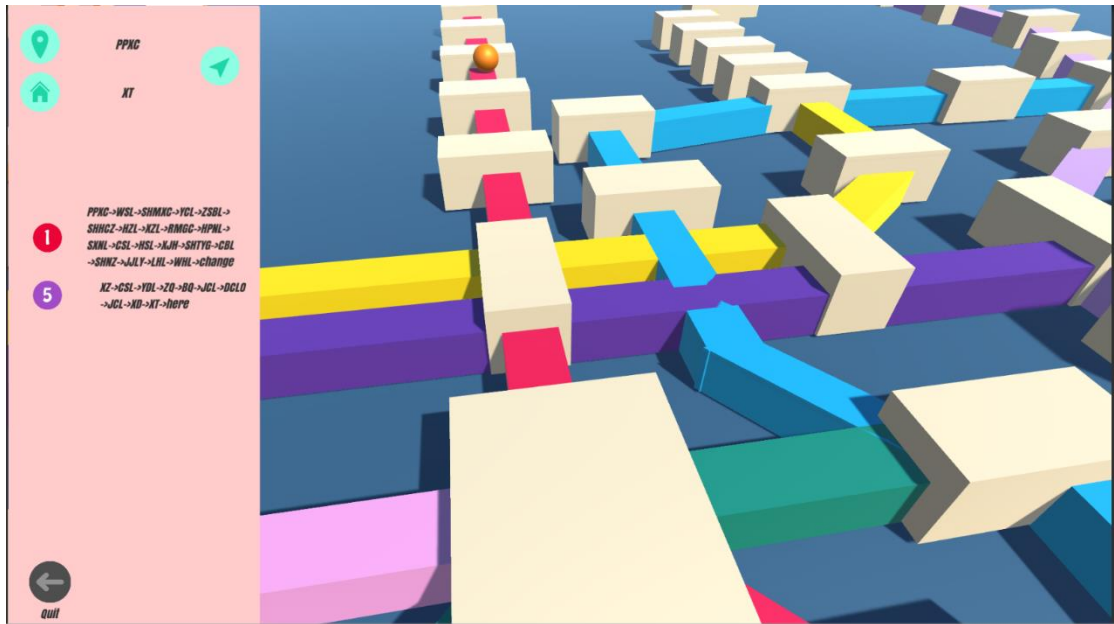
一种是通过在 UI 界面输入起点终点完成地铁换乘和循迹; 一种是鼠标直接点击, 玩家从当前位置寻找最短路径到达终点。

- 4、**最短路径循迹。**

无论在哪一种操作模式下, 玩家都需要找到最短的到达终点的路。因为在 level design 时比例尺不能完全和现实情况下的地铁相同。可视化中的图形长度仅为表示, 真正的站与站之间的长度为脚本程序中定义的长度。

5、简洁友好的 UI 界面。

玩家需要一个简洁友好，尽量美观的 UI 进行交互。UI 界面可以提供输入（包括起点终点），输出换乘策略等。



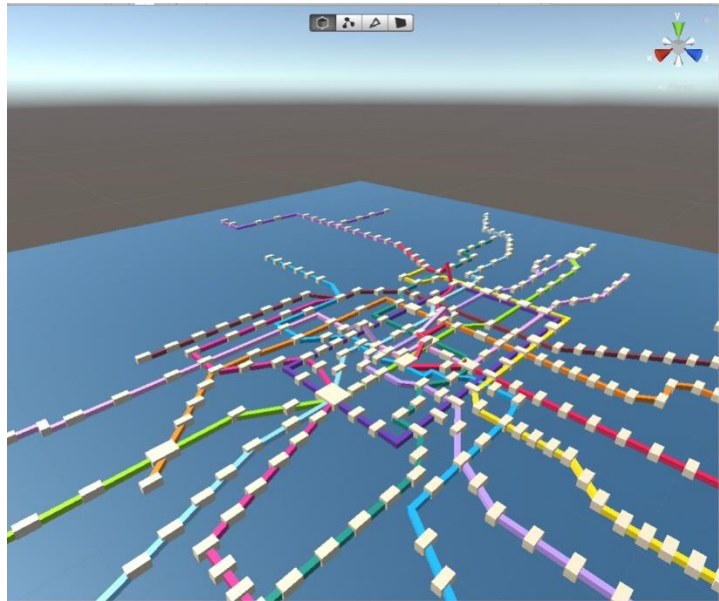
2.3 设计思想

本程序选择了Unity制作一款类似于地铁模拟的游戏。因为Unity引擎本身优秀的管线渲染，在小体量的程序中可以提供不错的交互与图形功能。因此本程序使用 unity 制作游戏本身，使用 visual studio 2017 编写 C#脚本控制游戏，部分 UI 素材来源于 unity asset store，其他素材均为使用 probuilder 绘制。

程序的核心功能在于：设置起始点，小球自动找到最短路径前往，并且将换乘方式打印在 UI 界面。因此核心设计在两个方面：如何进行最短路径循迹；如何打印信息。

1、准备工作：

Level Design: 首先建立基本的地形，在本程序中主要是地铁与走向的建立。站点使用固定大小的白色方块代表，每条线的地铁使用官方颜色作为代表。



UI 界面设计：本程序设置一个渲染层级最前的 Canvas 作为 UI 界面，包括两个输入框，循迹开始的交互按钮，显示换乘指南以及错误提醒的 text 区域，以及退出程序按钮。



相机控制：Unity 中摄像机作为程序视角的入口，但是通常为了方便玩家操作，需要对相机视角进行控制。在这里需要对相机的操作进行控制。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraController : MonoBehaviour
{
    // 模型
    public Transform model;

    // 旋转速度
    public float rotateSpeed = 32f;
```

```

public float rotateLerp = 8;
// 移动速度
public float moveSpeed = 1f;
public float moveLerp = 10f;
// 镜头拉伸速度
public float zoomSpeed = 10f;
public float zoomLerp = 4f;

// 计算移动
private Vector3 position, targetPosition;
// 计算旋转
private Quaternion rotation, targetRotation;
// 计算距离
private float distance, targetDistance;
// 默认距离
private const float default_distance = 5f;
// y 轴旋转范围
private const float min_angle_y = -89f;
private const float max_angle_y = 89f;

// Use this for initialization
void Start()
{
    // 旋转归零
    targetRotation = Quaternion.identity;
    // 初始位置是模型
    targetPosition = model.position;
    // 初始镜头拉伸
    targetDistance = default_distance;
}

// Update is called once per frame
void Update()
{
    float dx = Input.GetAxis("Mouse X");
    float dy = Input.GetAxis("Mouse Y");

    // 异常波动
    if (Mathf.Abs(dx) > 5f || Mathf.Abs(dy) > 5f)
    {
        return;
    }
}

```

```

float d_target_distance = targetDistance;
if (d_target_distance < 2f)
{
    d_target_distance = 2f;
}

// 鼠标左键移动
if (Input.GetMouseButton(0))
{
    dx *= moveSpeed * d_target_distance / default_distance;
    dy *= moveSpeed * d_target_distance / default_distance;
    targetPosition -= transform.up * dy + transform.right * dx;
}

// 鼠标右键旋转
if (Input.GetMouseButton(1))
{
    dx *= rotateSpeed;
    dy *= rotateSpeed;
    if (Mathf.Abs(dx) > 0 || Mathf.Abs(dy) > 0)
    {
        // 获取摄像机欧拉角
        Vector3 angles = transform.rotation.eulerAngles;
        // 欧拉角表示按照坐标顺序旋转, 比如 angles.x=30, 表示按 x 轴旋转 30°, dy 改变
        // 引起 x 轴的变化
        angles.x = Mathf.Repeat(angles.x + 180f, 360f) - 180f;
        angles.y += dx;
        angles.x -= dy;
        angles.x = ClampAngle(angles.x, min_angle_y, max_angle_y);
        // 计算摄像头旋转
        targetRotation.eulerAngles = new Vector3(angles.x, angles.y, 0);
        // 随着旋转, 摄像头位置自动恢复
        Vector3 temp_position =
            Vector3.Lerp(targetPosition, model.position, Time.deltaTime *
moveLerp);
        targetPosition = Vector3.Lerp(targetPosition, temp_position,
Time.deltaTime * moveLerp);
    }
}

// 上移
if (Input.GetKey(KeyCode.UpArrow))
{

```

```

        targetPosition -= transform.up * d_target_distance / (2f *
default_distance);
    }

    // 下移
    if (Input.GetKey(KeyCode.DownArrow))
    {
        targetPosition += transform.up * d_target_distance / (2f *
default_distance);
    }

    // 左移
    if (Input.GetKey(KeyCode.LeftArrow))
    {
        targetPosition += transform.right * d_target_distance / (2f *
default_distance);
    }

    // 右移
    if (Input.GetKey(KeyCode.RightArrow))
    {
        targetPosition -= transform.right * d_target_distance / (2f *
default_distance);
    }

    // 鼠标滚轮拉伸
    targetDistance -= Input.GetAxis("Mouse ScrollWheel") * zoomSpeed;
}

// 控制旋转角度范围: min max
float ClampAngle(float angle, float min, float max)
{
    // 控制旋转角度不超过 360
    if (angle < -360f) angle += 360f;
    if (angle > 360f) angle -= 360f;
    return Mathf.Clamp(angle, min, max);
}

private void FixedUpdate()
{
    rotation = Quaternion.Slerp(rotation, targetRotation, Time.deltaTime *
rotateLerp);
    position = Vector3.Lerp(position, targetPosition, Time.deltaTime *
moveLerp);

```

```

        distance = Mathf.Lerp(distance, targetDistance, Time.deltaTime * zoomLerp);
        // 设置摄像头旋转
        transform.rotation = rotation;
        // 设置摄像头位置
        transform.position = position - rotation * new Vector3(0, 0, distance);
    }
}

```

2、最短路径选择

主要流程如下所示：

1. 体素化。从源几何体构造实心的高度场，用来表示不可行走的空间。
2. 生成地区。将实心高度场的上表面中连续的区间合并为地区。
3. 生成轮廓。检测地区的轮廓，并构造成简单多边形。
4. 生成多边形网格。将轮廓分割成凸多边形。
5. 生成高度细节。将多边形网格三角化，得到高度细节。

体素化

在体素（体素是空间中的一个有大小的点）化阶段，源几何体被转换成高度场，用来表示不可行走的空间。一些不可走的表面在这个阶段会被剔除掉（比如坡度过大的面）。对于源几何体上的每个三角形，使用“保守体素化算法”（Conservative Voxelization）分割成体素，并加入到高度场中。保守体素化算法确保了每个三角形面，都会被生成的体素完全包围。

体素化阶段后，实心高度场(solid heightfield)包含了很多的区间(span)，覆盖了源几何体上的所有面。

生成地区

这一阶段的目标是，进一步定义实体表面上哪部分是可以行走的，以及将这些可行走的部分划分成连续的地区，这些地区可以最终构成简单多边形。

首先，将实心的高度场，转换成一个开放的高度场(open heightfield)，用来表示实体表面上那些可以行走的部分。一个开放的高度场，表示位于实体空间表面的地表部分。

然后，进一步剔除掉不可行走的区域。在计算完成的时候，开放区间中那些认为可以行走的部分，应该通过下面的测试：

该区域不能紧挨着障碍物（使用 WalkableRadius 作为距离阈值）

该区域在表面之上没有足够的开放空间(非碰撞区域)。（人在不碰撞到其他物体的情况下，能够合法的移动）（使用 WalkableHeight 作为高度阈值）

为剩下的所有区间生成邻接信息，用于把他们合并成一个大的面片。该算法使用一个最大垂直步长(WalkableStep4)来决定哪些区间是可以连在一起的。这允许一些特殊的结构能够被考虑进来。在该阶段后，这些相连的地区代表了可行走的面。

生成轮廓

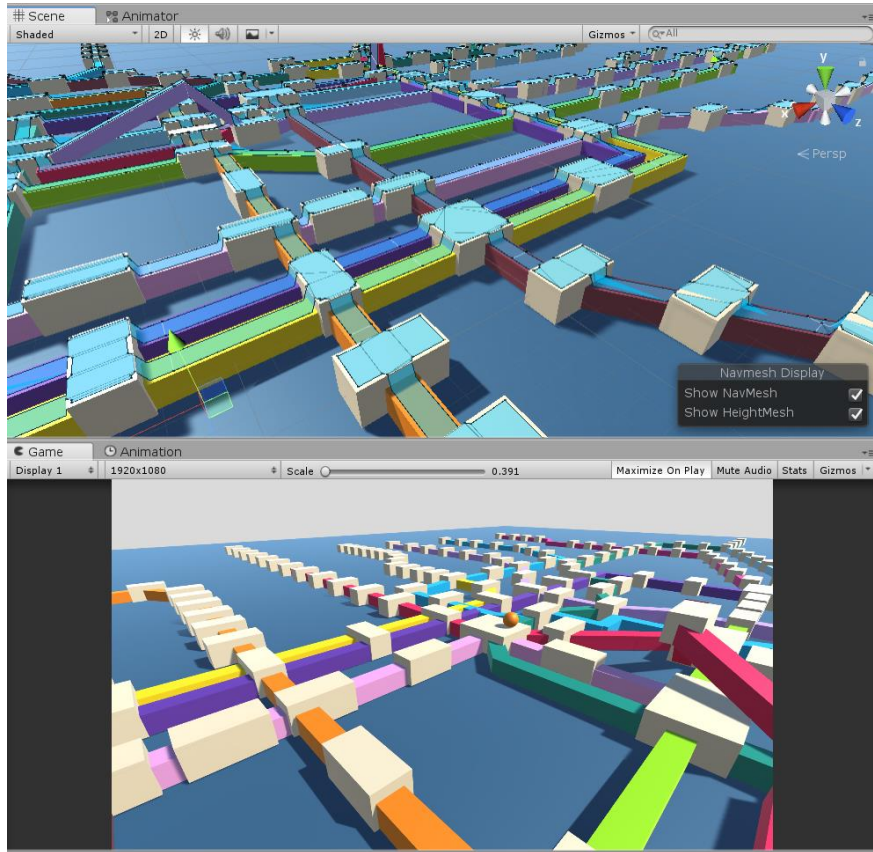
轮廓就是沿着地区边缘“行走”，构成简单多边形。这是从体素空间转换回向量空间的第一步处理。

首先，从地区生成非常精细的多边形。使用多种算法来完成下面的步骤：

简化相邻多边形的边缘（地区之间的部分）

简化边界(边界是没有邻接或邻接了障碍物的轮廓)(EdgeMaxDeviation)

优化边界的长度。（边界如果太长，不能得到最优的三角形）



上图中 Scene 页面中已经生成了可行走的 mesh 轮廓。

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Grid : MonoBehaviour {
    public GameObject NodeWall;
    public GameObject Node;

    public float NodeRadius = 0.5f;
    public LayerMask WhatLayer;
    public Transform player;
    public Transform destPos;

    public class NodeItem {
        public bool isWall;
        public Vector3 pos;
        public int x, y;

        public int gCost;
    }
}
```

```

    public int hCost;

    public int fCost {
        get {return gCost + hCost; }
    }

    public NodeItem parent;

    public NodeItem(bool isWall, Vector3 pos, int x, int y) {
        this.isWall = isWall;
        this.pos = pos;
        this.x = x;
        this.y = y;
    }
}

private NodeItem[,] grid;
private int w, h;

private GameObject WallRange, PathRange;
private List<GameObject> pathObj = new List<GameObject> ();

void Awake() {
    w = Mathf.RoundToInt(transform.localScale.x * 2);
    h = Mathf.RoundToInt(transform.localScale.y * 2);
    grid = new NodeItem[w, h];

    WallRange = new GameObject ("WallRange");
    PathRange = new GameObject ("PathRange");

    for (int x = 0; x < w; x++) {
        for (int y = 0; y < h; y++) {
            Vector3 pos = new Vector3 (x*0.5f, y*0.5f, -0.25f);

            bool isWall = Physics.CheckSphere (pos, NodeRadius, WhatLayer);
            grid[x, y] = new NodeItem (isWall, pos, x, y);

            if (isWall) {
                GameObject obj = GameObject.Instantiate (NodeWall, pos,
Quaternion.identity) as GameObject;
                obj.transform.SetParent (WallRange.transform);
            }
        }
    }
}

```

```

    }

}

public NodeItem getItem(Vector3 position) {
    int x = Mathf.RoundToInt (position.x) * 2;
    int y = Mathf.RoundToInt (position.y) * 2;
    x = Mathf.Clamp (x, 0, w - 1);
    y = Mathf.Clamp (y, 0, h - 1);
    return grid [x, y];
}

public List<NodeItem> getNeighbourhood(NodeItem node) {
    List<NodeItem> list = new List<NodeItem> ();
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
            // 如果是自己, 则跳过
            if (i == 0 && j == 0)
                continue;
            int x = node.x + i;
            int y = node.y + j;
            // 判断是否越界, 如果没有, 加到列表中
            if (x < w && x >= 0 && y < h && y >= 0)
                list.Add (grid [x, y]);
        }
    }
    return list;
}

// 更新路径
public void updatePath(List<NodeItem> lines) {
    int curListSize = pathObj.Count;
    for (int i = 0, max = lines.Count; i < max; i++) {
        if (i < curListSize) {
            pathObj [i].transform.position = lines [i].pos;
            pathObj [i].SetActive (true);
        } else {
            GameObject obj = GameObject.Instantiate (Node, lines [i].pos,
Quaternion.identity) as GameObject;
            obj.transform.SetParent (PathRange.transform);
            pathObj.Add (obj);
        }
    }
    for (int i = lines.Count; i < curListSize; i++) {

```

```

        pathObj [i].SetActive (false);
    }
}

//曼哈顿估价法
private function manhattan(node:Node):Number
{
    return Math.abs(node.x - _endNode.x) * _straightCost + Math.abs(node.y +
_endNode.y) * _straightCost;
}

//几何估价法
private function euclidian(node:Node):Number
{
    var dx:Number=node.x - _endNode.x;
    var dy:Number=node.y - _endNode.y;
    return Math.sqrt(dx * dx + dy * dy) * _straightCost;
}

//对角线估价法
private function diagonal(node:Node):Number
{
    var dx:Number=Math.abs(node.x - _endNode.x);
    var dy:Number=Math.abs(node.y - _endNode.y);
    var diag:Number=Math.min(dx, dy);
    var straight:Number=dx + dy;
    return _diagCost * diag + _straightCost * (straight - 2 * diag);
}

```

3、换乘指南打印

在这里存在一个问题，小球的最短路径选择和换乘指南的打印是分离的。因为在最短路径的选择中，为了加速寻找的速度，以精简的图的形式存储。在换乘指南中，我们使用 key-value 形式的数据结构。首先在程序初始化的时候，将所有站点之间的换乘方式生成，此处采用的是求非空子集的做法。全部存储到 key-value 形式的数据中，在玩家输入后直接进行 $O(1)$ 的查找。

生成整个 key-value 的时间是 $O(n^2)$ 级别的，但是由于 Unity 的初始加载过程中会有读条页面出现，因此在完全进入游戏时已经生成结束。

2.4 逻辑结构与物理结构

1、数据结构

在计算最短路径是采用了图的数据结构。

在打印换乘指南时使用 key-value 结构。

```
map=new Dictionary<Tuple<string, string>, string>();  
map.Add(new Tuple<string, string>("FJL", "YYXL"), "#01FJL#YYXL#");
```

前面的二元组表示起点和终点的首字母大写，后面的 string 表示换成指南。在打印的时候需要解码。如上一句 string 表示 01 号线从 FJL（富锦路）到 YYXL（友谊西路）。

2、在程序中 Assets 存储所有的游戏素材。程序组织

Frame：存储贴图 sprite 的框架

Materials：存储不同的渲染材质

Prefabs：存储游戏过程中的预设，以便在游戏中直接生成

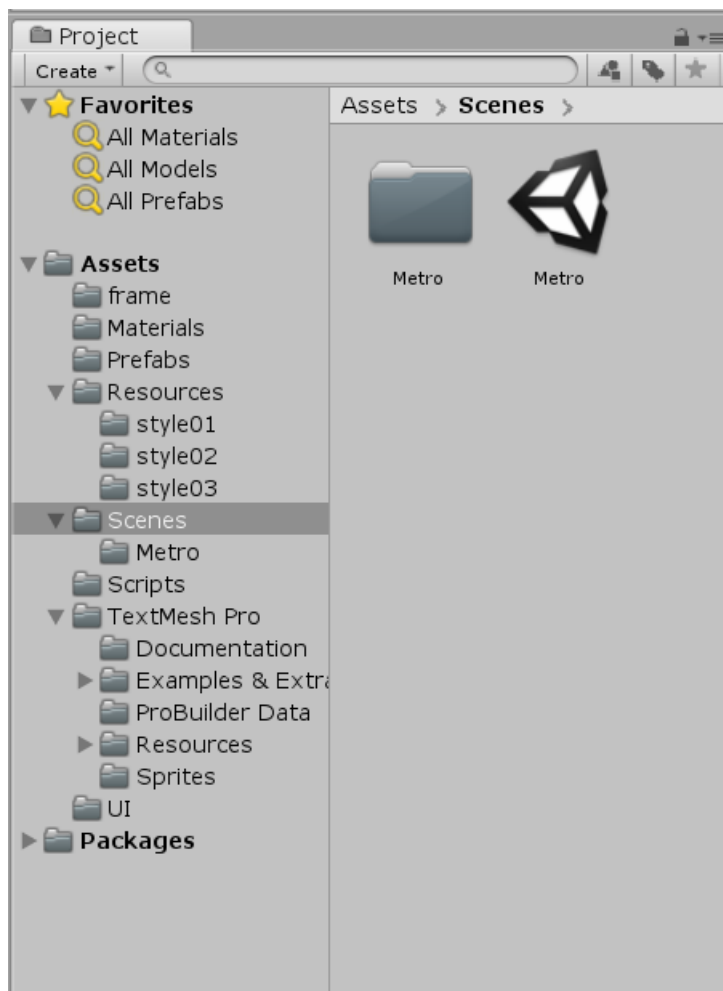
Resources：存储游戏中使用的所有美术素材

Scenes：存储场景

Scripts：存储控制脚本

TextMeshPro：存储 Level Design 过程中的素材与数据

UI：存储所有 UI 使用的素材



2.5 开发平台

开放平台：Unity5-2018.3.0f2, Visual Studio 2017

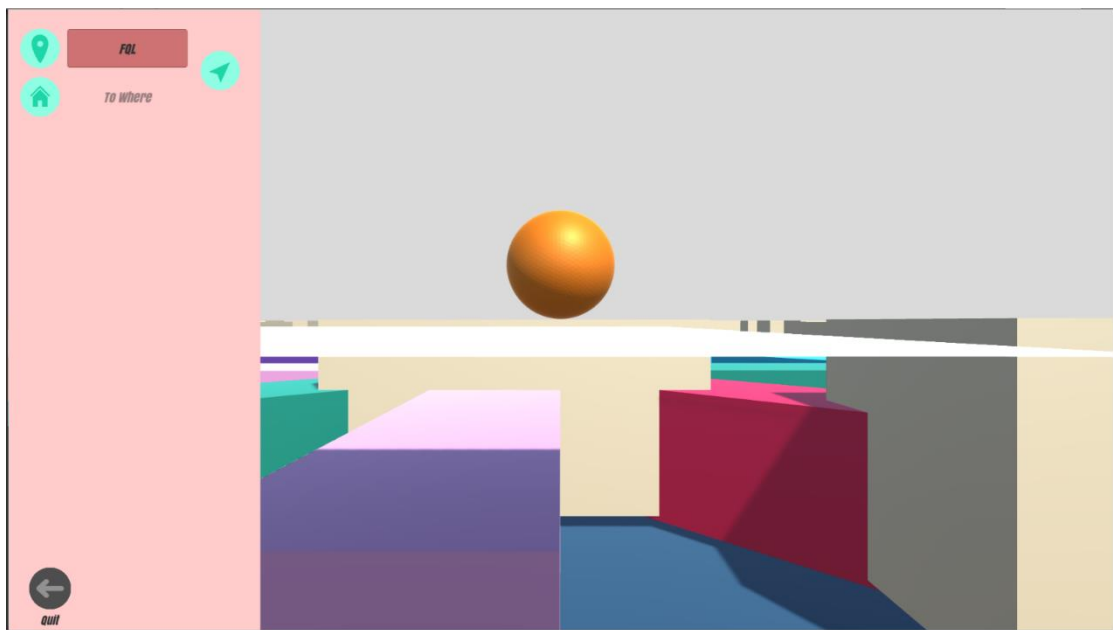
程序运行环境：Windows PC、linux、Mac os

素材来源：Unity Assets Store, adobe ai 自主绘制

2.6 系统的运行结果分析说明

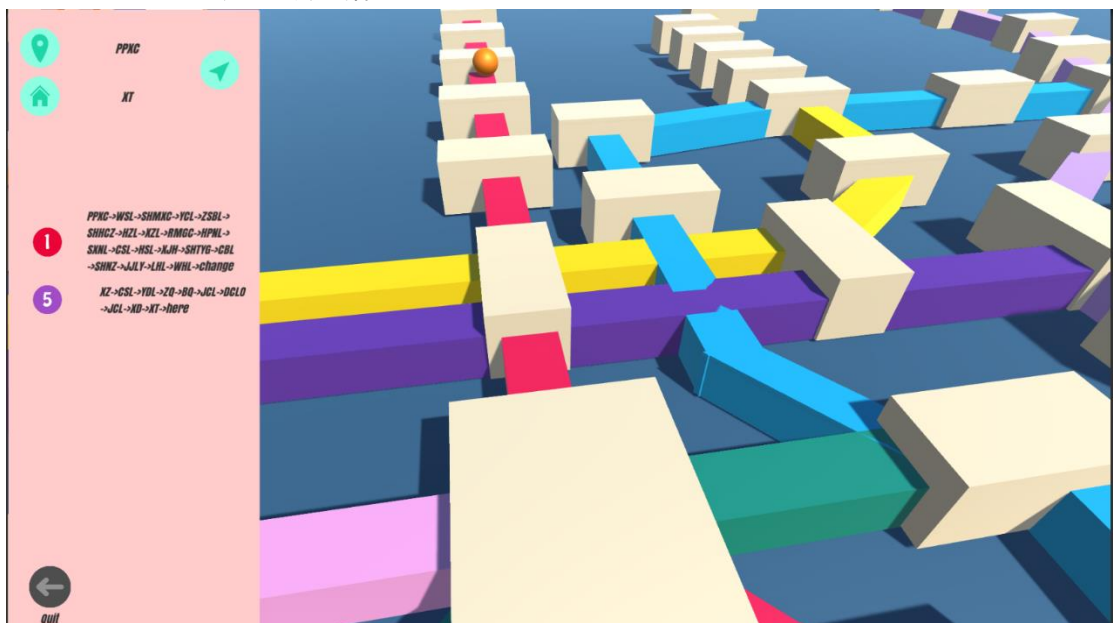
1、输入起点终点

输入时输入框会有颜色渐变的效果作为提示。输入完成后点击右侧的 Go 的按钮，同样会以颜色变化作为按下提示。



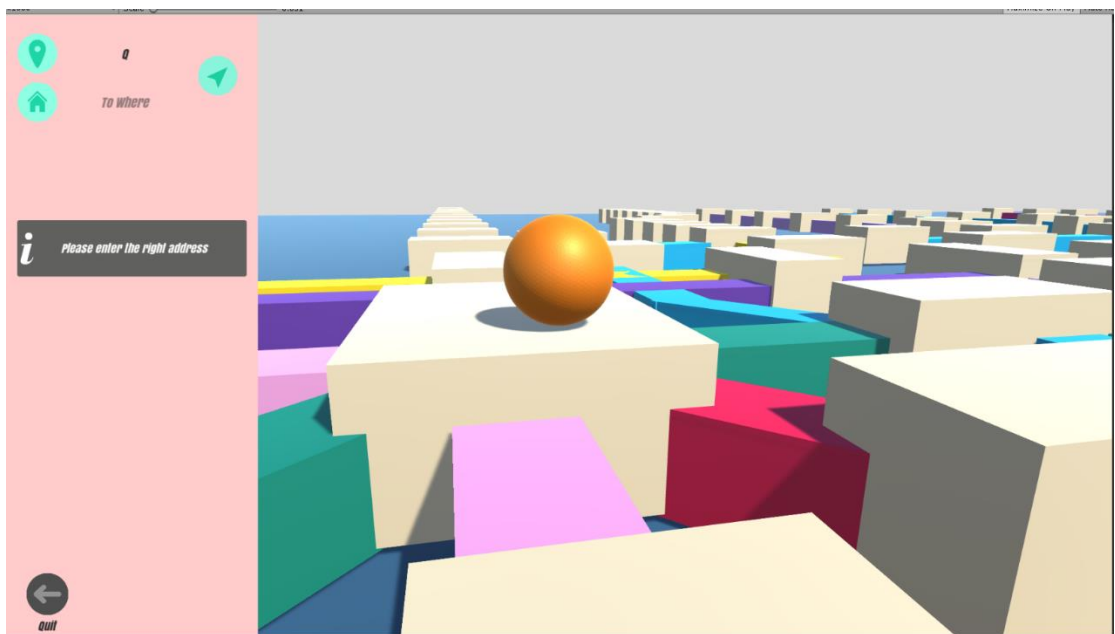
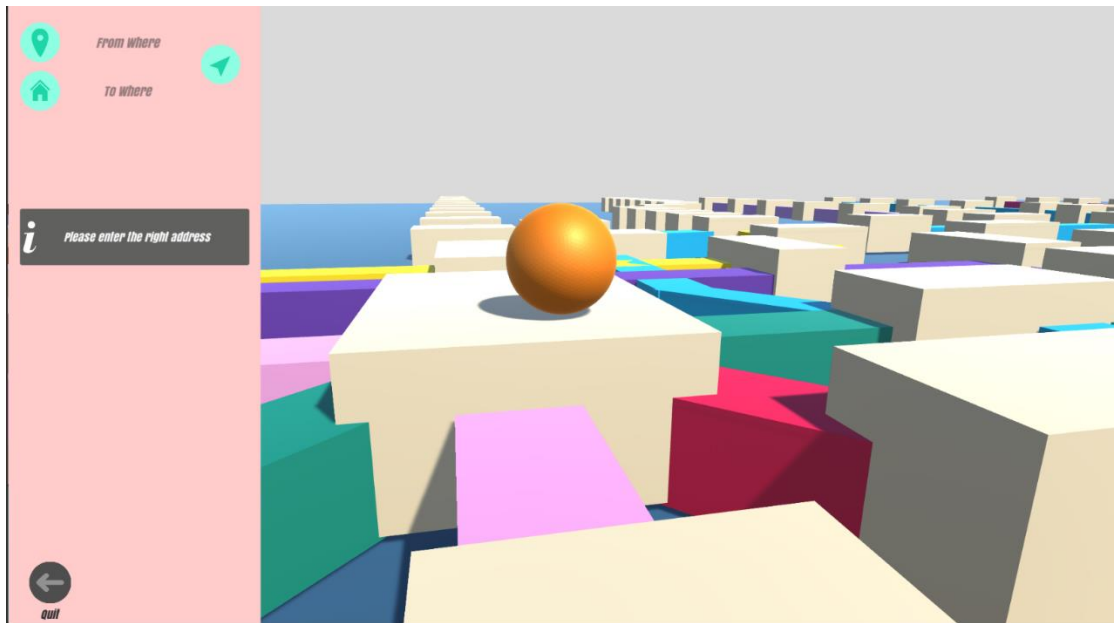
2、正常运行

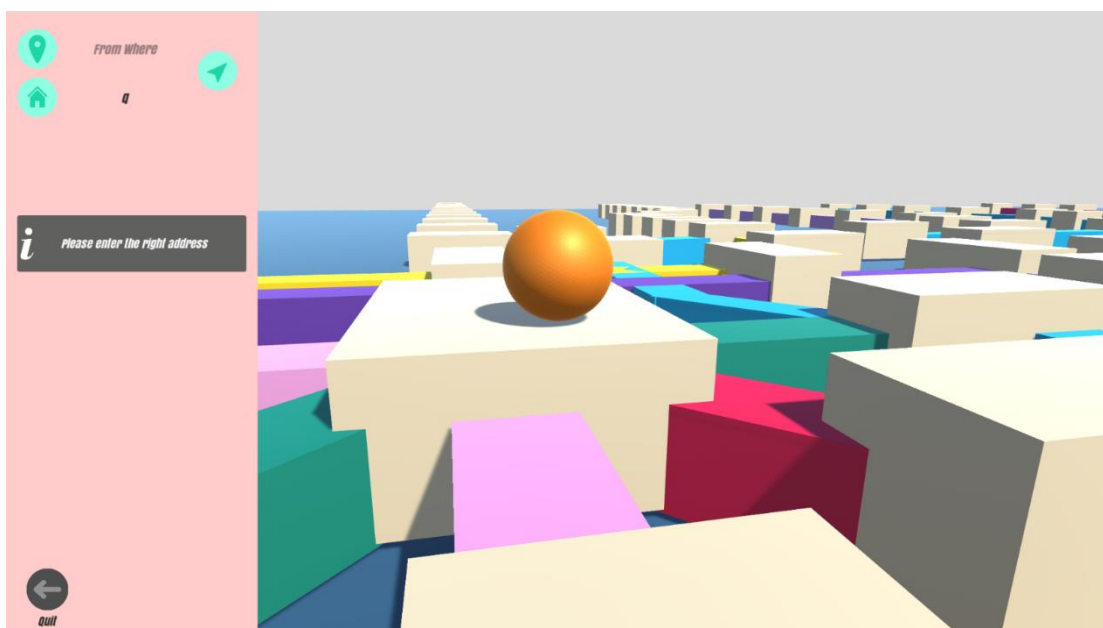
正常运行后下方会显示换成指南。



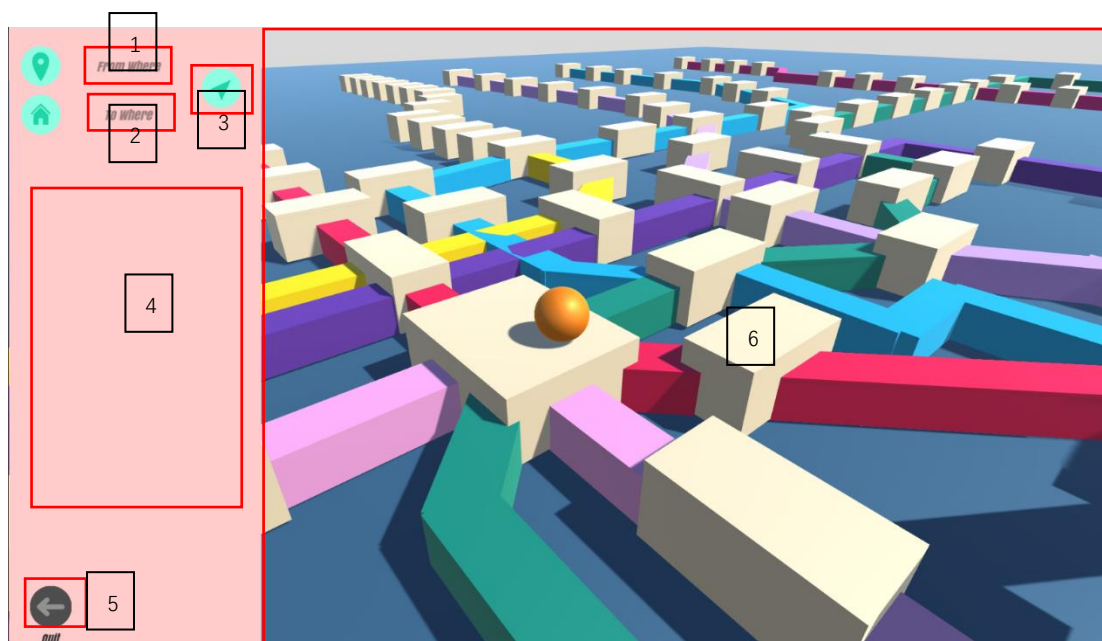
3、输入错误

在没有输入、输入不完整、输入错误情况下均会报错提示。





2.7 操作说明



相机控制：

- 1、 鼠标左键： 按住进行平移
- 2、 鼠标右键： 按钮进行以视角中点的旋转
- 3、 鼠标滚轮： 进行以视角中点的缩放
- 4、 方向键上下左右： 进行视角上下左右的平移

角色控制：

- 1、 输入起点：输入开始的地铁站。注意这里的输入输出均为地铁站首字母大写。如枫桥路即为（FQL）。
- 2、 输入终点：输入结束的地铁站。注意这里的输入输出均为地铁站首字母大写。如枫桥路即为（FQL）。

- 3、 循迹按钮：输入完成之后点击该按钮进行循迹。
- 4、 换乘输出：点击循迹按钮之后，此处会显示换乘的方式。
- 5、 退出程序：点击该按钮会退出程序。
- 6、 游戏世界：点击任意位置小球会从当前位置前往点击位置，在点击循迹按钮之后小球会自动沿着最短路径前进。

Github 地址 (<https://github.com/CrowFea/DataStructureDesgin>)

第三部分 实践总结

3.1 所做的工作

- 1、完成两项数据结构设计要求。基本完成了设计的要求。完成了一项 Java 窗体程序的制作，一项 Unity 游戏的制作。
- 2、亮点的部分：
在 java 窗体设计中，设计的 B-树可以进行前一步后一步的查看，整体的速度也较快。
在 Unity 设计中，优化了相机和管线的渲染，因为不需要进行复杂的渲染工作，简化后的渲染可以更快的完成工作；完成了可交互的自主探索，用户可直接点击就进行循迹；将换乘指南的打印和循迹分开，并行的工作加速了循迹速度。
- 3、不足的部分：
在 Unity 设计中，由于站点都是由站点首字母组成，不免有重复的部分。我们在这里将重叠的站点进行了标注，但在换乘指南的子集生成仍存在问题，导致有些站点不能正确打印换成指南，但是可以完成循迹。

3.2 总结与收获

- 1、在实践过程中，我对这门课程有了一个更进步的认识，在数据抽象能力、编程能力有了进步。对一些常用的基本数据结构（包括数组、顺序表、链表、栈与队列、广义表、树与森林、二叉树、图、索引结构、散列结构等）及其不同的存储实现，有了更深的理解。
- 2、实践中选取了两种不常用的语言与平台进行设计，对 Java、C#语言的使用更加熟悉。对 Unity 的使用也更熟练。

第四部分 参考文献

- [1]. Thomas H.Cormen、 Charles E.Leiserson, Introduction to Algorithms. The MIT Press, 2nd edition (September 1, 2001)
- [2]. Peter Shirley, Steve Marschner, Fundamentals Of Computer Graphics,
- [3]. Peter Shirley, Ray Tracing_ The Next Week
- [4]. Introduction to 3D Game Programming with DirectX 11