线性与图型数据结构可视化模拟器

学　　号_____20027106_____

姓　　名_____刘海天_____

指导教师_____杜永萍_____

2022 年 11 月

目录

1、 需求分析

1.1 程序功能需求分析

本软件利用图形可视化的方法，从数据结构的可视化和算法执行过程的可视化两个方面实现了一个可视化系统。用户可以通过绘制图形和符号来展现数据结构中数据元素以及数据元素之间的关系，也可以依托软件中提供的输入功能数据读入系统，软件能够根据用户的需求展现相关算法的执行过程。为实现如上需求，本软件的设计分为三个主题层面：数据层，图形层，交互层。

1.1.1 数据层

1） 基于面向对象的理论，对于单链表、循环队列、二叉搜索树建立其对应的类并进行封装，在处理数据集的时候依照其不同的数据结构实现基础的增删改操作。
2） 对于单链表、循环队列、二叉搜索树实现相关的 2~3 个算法。

1.1.2 图形层

1） 能够与数据层的内容形成一一对应，包括点集、边集等信息的连接
2） 能够在用户界面形成可视化图形，能够响应数据层的变化。
3） 能够通过动画的方式显示可视化图形的变化，从而展现算法的实现过程

1.1.3 交互层

1） 用户可以在交互页面准确找到不同的功能分区并执行想要的操作
2） 用户可以在交互界面自由的添加、删除、修改和拖拽节点，实现对数据层和图形层内容的修改。

1.2 数据处理需求分析

本软件一共需要处理三种类型的数据，分别为：数据层、图形曾数据、交互层数据。数据层数据即单链表、循环队列、二叉搜索树的类的定义，存储数据并提供算法函数供其他层调用。数据层的数据与图形层数据相连。

图形层数据主要是依托绘图库，数据完全来源于数据层数据，并且实现一一绑定，在数据层数据发生变化的时候能够实时响应。除此之外，图形层数据还包含可视化图形布局元素，需要保证可视化视图元素间互不影响、互不重叠、显示清晰。

交互层数据主要通过用户在软件交互界面获取，交互数据的改变将直接修改数据层数据，用户在交互页面的每一种“添加”“删除”等操作都会直接写入数据层进行修改。同时功能按钮的触发，将按照设定的功能进行数据的修改以完成预设的不同功能。

1.3 程序开发的需求分析

本软件主要采用面向对象的开发方式。由于 JavaScript 是开发 Web 的常用编程语言，适合进行可视化操作，且 ES6 支持类的封装、继承的功能，符合面向对象的开发需求，故在此软件中主要使用 JavaScript 作为主要编程语言进行开发。

HTML + CSS + JavaScript 是当下流行的 WEB 开发组合。HTML 是一种用来制作超文本文档的简单标记语言，用 HTML 编写的超文本文档可以涵盖文字、图片、声音、动画等多种不同元素，并放置在浏览器上产生相同的页面效果，与操作系统或浏览器的具体选择无关。CSS 是一种表现 HTML 或 XML 等文件样式的计算机语言，可以静态修饰网页，并对网页中各元素的样式与排版进行控制，可以极大程度的美化页面。故本软件最终选择 HTML5 + CSS3 + JavaScript 进行开发。

Antv-G6 是一个简单、易用、完备的图可视化引擎，它在高定制能力的基础上，提供了一系列设计优雅、便于使用的图可视化解决方案。能帮助开发者搭建属于自己的图 、图分析应用或是图编辑器应用。因此本软件引入 antv-G6 作为可视化支持。

除此之外，为了方便 JavaScript 的调试，引入了 node-JS 作为调试工具。

1.4 用户页面设计需求分析

从功能需求分析中可以得知，本软件需要提供单链表、循环队列、二叉搜索树的可视化功能并实现其相关的算法。为了能够实现三种不同数据结构内容的切换，主页上能够进行对三个不同页面的跳转，并依托浏览器的功能实现返回和切换，从而方便了用户在三种不同数据结构（三种不同功能）的页面的选择和切换。

1.4.1 主页设计

在程序主页（index.html）中可以看到三个超链接，超链接上有对应的文字提示，用户可以点击文字进行超链接至对应的功能页

1.4.2 功能页设计

本软件对于单链表、循环队列、二叉搜索树三种不同数据结构的可视化页面采用了相同的设计语言。页面分为三个部分：标题、功能栏、可视化内容显示区（画布 canvas），用户可以在功能栏按照按键以及输入栏的提示进行数据的读入，从而实现软件的交互并在显示区看到对应的功能可视化实现。设计示意图如下。

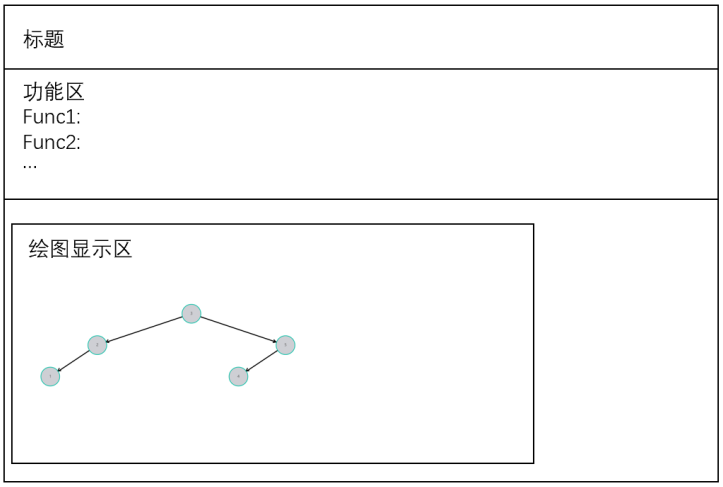


图 1：功能页页面设计

2、 数据结构设计

2.1 程序整体结构

2.1.1 程序流程和数据流交互

如下图所示，本软件的数据流分为两种情况。当用户在交互界面执行的功能对数据结构内部数据有修改时，会将修改的信息以数据流的方式传送至数据层进行修改，数据层修改后同步至图形层进行图实例的修改，同时选取当前合适的动画效果并一同返回图形可视化界面。当用户在交互界面执行的功能对数据结构内部数据无影响时，例如执行算法时，数据流将不经过数据层，直接与图形层相连，执行相关动画效果渲染后显示在图形可视化界面。图中“功能”特指代对数据有影响的操作（例如增、删、改），“算法”特指代对数据无影响的操作（例如查询、遍历等）。

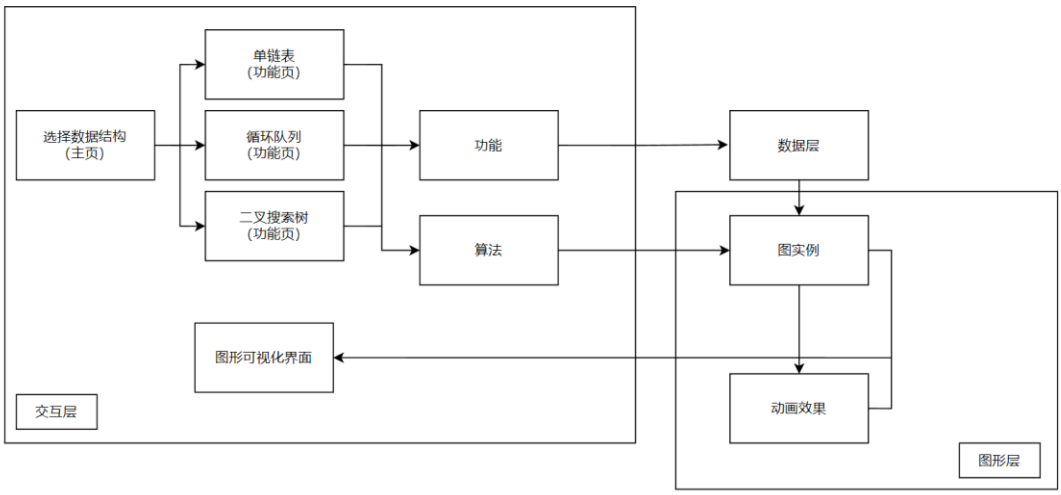


图 2：数据流交互

2.1.2 程序文件组成

本软件在开发的时候引用了 antv-G6 图形化引擎和 nodeJS 调试工具，这些结构在目录中不做赘述，所有的程序都放置在 src 目录下，具体结构如下图所示。Js 文件夹下时所有 JavaScript 代码，包括三种数据结构的数据层构建、canvas 画布属性以及将数据层与图形层、交互层相互绑定连接的脚本。根目录下的 html 文件分别为主页和功能页的网页文件，是用户交互页面的入口。用户可以通过访问 index.html 进入软件主页进行操作。

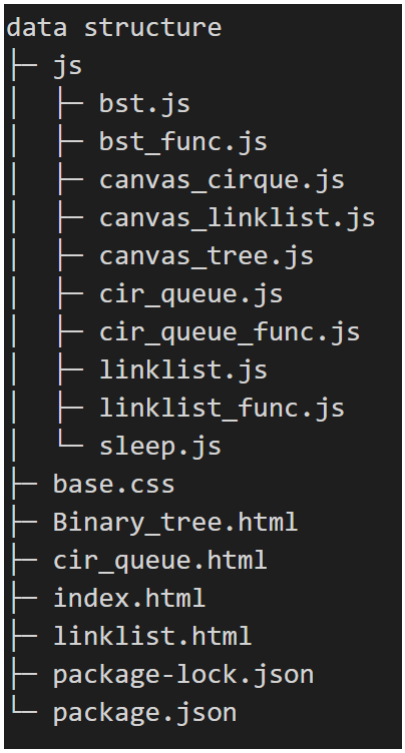


图 3：程序文件树

2.2 数据结构设计

2.2.1 单链表

单链表的定义如下图 4 所示。首先定义了结点，每个结点具有值 element 和 next 指针两个属性；并且定义了单链表数据结构，单链表的构造函数中声明了头指针和链表的长度。

```
class Node {
  constructor(element) {
    this.element = element
    this.next = null
  }
}

export class LinkList {
  constructor() {
    this.size = 0
    this.head = null
  }
```

图 4：单链表类定义

如图 5 所示，单链表 Linklist 包含了 5 个基础的成员函数，用于对链表进行基础操作。

图 5：单链表成员函数

(1) add(element):将值为 element 的节点添加到单链表的末尾。
首先判断链表是否为空，如果为空，该节点成为链表的头指针节点，否则从头节点顺次遍历至末尾节点，然后让末尾节点的 next 指针执行该节点，实现将节点添加至单链表的末尾，此时，链表的 size 属性+1。

(2) remove(position):将链表中位置为 position 的节点删除。
首先判断该位置是否为 0，即该节点是否是头指针节点，如果是头指针节点，则将头指针节点 next 指向的节点作为链表的头节点。如果该位置非头指针节点，则将 position-1 位置的节点的 next 指针指向 position+1 位置的节点实现节点删除，同时 size--。

(3) insert(position,element):在链表 position 的位置插入值为 element 的节点。
首先判断该位置是否为 0，即该节点是否是头指针节点，如果是头指针节点，则需要新建节点作为头指针节点，然后将该新建的节点的 next 指针指向原头指针节点。否则需要从头指针遍历找到该位置的前驱节点，然后令前驱节点的 next 指针指向新节点，新节点的 next 指针指向原 position 位置的节点实现插入，此时 size+。

(4) getNode(index):找到 index 位置的节点
从头指针节点开始遍历，直到找到 index 位置的节点并返回这个节点。

(5) clear():将链表清空
仅需将链表的头指针指向 null，并将 size 属性清零。

2.2.2 循环链表

循环队列的定义如下图 6 所示。循环队列中利用数组 Q 存储数据，然后分别设置头索引 headIdx 和尾索引 tailBackIdx，并设置循环队列的容量 capacity。

```
export class CircularQueue{
    constructor(k)
    {
        this.capacity = k;//容量
        this.headIdx = 0;
        this.tailBackIdx = 0;
        this.Q = [];
    }
```

图 6：循环链表类定义

如图 7 所示，循环队列 CircularQueue 中包含了 8 个基本的成员函数，可以对循环队列进行基本操作。

```
export class CircularQueue{
    constructor(k)
    {
        this.capacity = k;//容量
        this.headIdx = 0;
        this.tailBackIdx = 0;
        this.Q = [];
    }
    enqueue(el) {…
    }

    dequeue() {…
    };

    isEmpty() {…
    };

    isFull() {…
    };

    getFront() {…
    };

    getRear() {…
    };
    size(){…
    };
    clear() {…
    };
};
```

图 7：循环链表成员函数

(1) enqueue(el)：将值为 el 的节点插入循环队列

首先计算出尾指针在数组中实际的位置（tailBackIdx % capacity），然后将值 el 存入数组 Q 中尾指针的位置后，将尾指针 tailBackIdx++

(2) dequeue：将队首元素弹出

首先判断队列是否为空，如果为空则返回 false，当不为空时，头指针 headIdx++即是将队首元素弹出

(3) inEmpty()：判断队列是否为空

当头指针 headIdx 和 tailBackIdx 值相等时，队列为空

(4) inFull()：判断队列是否为满

当尾指针和头指针之间的差值为队列的容量时，队列为满

(5) getFront()：获得队首元素
返回 Q[headIdx % capacity]的值即可获得队首元素的值

(6) getRear()：获得队尾元素
返回 Q[(tailBackIdx-1) % capacity]的值即可获得队尾元素的值

(7) size：获得队列的长度
返回(tailBackIdx+capacity-headIdx)%capacity 的值即为队列的长度

(8) clear：清空队列
将头指针尾指针同时置零即可清空队列

2.2.3 二叉搜索树

二叉搜索树的定义如下图所示。二叉搜索树在数据层的表示本质上与一般二叉树并没有区别。首先定义树上节点，构造函数中定义节点的值、左子指针、右子指针、父节点指针、深度。图中的 x,y,cnt,d 等属性与绘图库中渲染函数相关，与数据层无关。定义二叉搜索树时，将根节点置为 null。

```
export class Node {
    constructor(key){
      this.key = key;
      this.fa = null;
      this.left = null;
      this.right = null;
      this.x=0;
      this.y=0;
      this.d=300;
      this.cnt=0;
      this.height=0;
    }
  }

export class BinarySearchTree {
    constructor() {
      this.root = null; // Node 类型的根节点
    }
```

图 8：二叉搜索树类定义

如图 9 所示，共有 15 个成员函数，其中一些成员函数为了辅助其他成员函数进行递归调用而设立，也有一些函数在类定义时集成实现了算法的功能。在此仅介绍二叉搜索树中的 5 个基础功能。

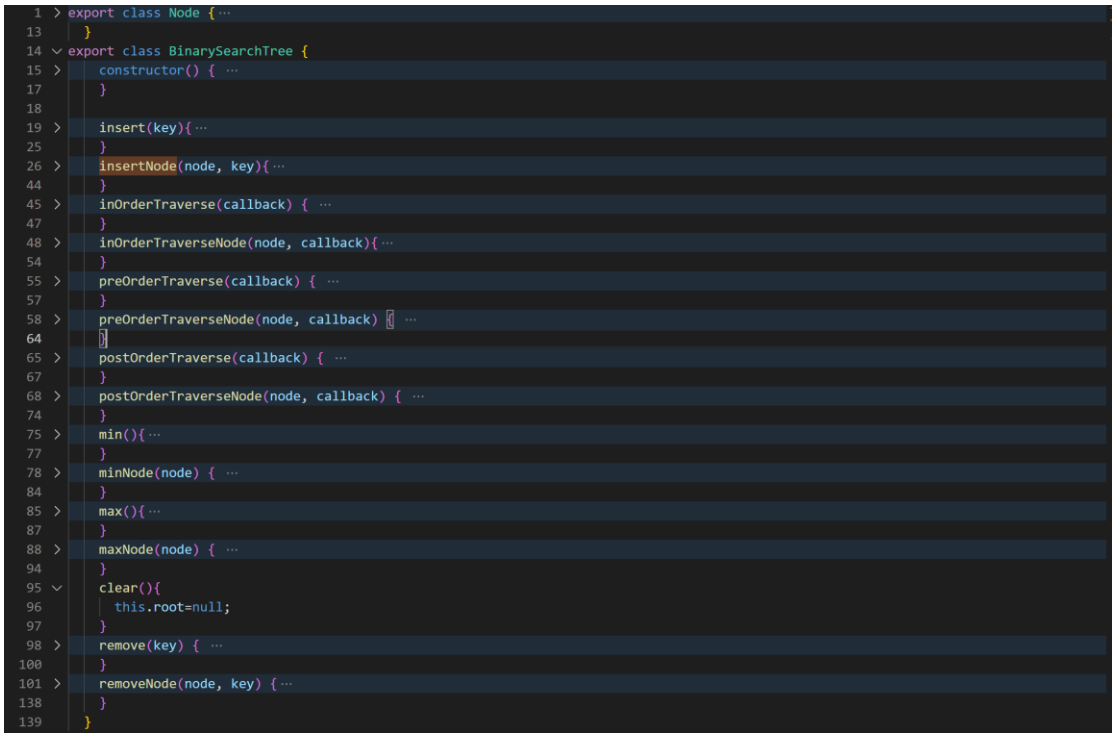


图 9：二叉搜索树成员函数

(1) insert(key)：向二叉搜索树中插入值为 key 的节点

insertNode(Node,key)函数为实现 insert 功能的递归函数，从树根开始遍历起，若当前节点的值大于 key，则判断其左子是否为空，如果为空，则建立左子并赋值为 key，否则将递归查找左子树可以插入的位置。若当前节点的值小于 key，则仿照如上规则，判断其右子节点，并递归查询其右子树，直到找到相应的位置插入。

(2) min()：查找二叉搜索树中最小的节点的值

minNode(node)为实现 min()功能的递归函数。从根节点起，递归查询左子节点，直到叶子节点，此时叶子节点的值为二叉搜索树中最小的值。

(3) max()：查找二叉搜索数中最大的节点的值

maxNode(node)为实现 max()功能的递归函数。从根节点起，递归查询右子节点直到叶子节点，此时叶子节点的值为二叉搜索树中最大的值。

(4) clear()：清空二叉搜索树

将根节点置为 null

(5) remove(key)：将值为 key 的节点删除

removeNode(node,key)函数为实现 remove 功能的递归函数，首先从树根开始遍历找到值为 key 的节点。然后分三种情况讨论：若该节点没有子节点，直接设置为 null；若该节点有一个子节点，就令其子节点作为其父节点的子节点；若该节点有两个子节点,找到右侧子树中最小节点的指向该节点，然后在删掉该节点。

3、 详细设计

3.1 单链表主要功能

3.1.1 尾部插入

```
var div = document.getElementById('func1b')    // 尾部插入
    div.addEventListener('click',function(){
    const tmp=document.getElementById("func1").value
    list.add(tmp);
    list_To_graph();
    console.log(list);
    console.log(data);
    console.log(graph);
    graph.data(data);
    graph.render();
    graph.fitView();
   })
```

图 10：尾部插入代码

如图 10 所示，在交互页面触发点击事件后，执行 add 函数，add 为 linklist 类的成员函数，在 2.2.1 中已经详细描述。然后进行数据层与图形层的绑定 list_To_graph()，然后利用 G6 引擎进行渲染

3.1.2 删除位置为 index 的节点

```
var div = document.getElementById('func2b')    // 删除
    div.addEventListener('click',function(){
    const tmp=document.getElementById("func2").value
    list.remove(tmp);
    list_To_graph();
    graph.data(data);
    graph.render();
    graph.fitView();
   })
```

图 11：删除节点代码

如图 11 所示，与 3.1.1 类似，此处为执行 remove 函数。

3.1.3 查询位置为 index 的节点值

```
var div = document.getElementById('func3b')    // 查询
    div.addEventListener('click',function(){
    const tmp=document.getElementById("func3").value
    alert(list.getNode(tmp).element);
   })
```

图 12：查询节点代码

如图 12 所示，与 3.1.1 类似，此处为执行 getNode 函数，然后通过 alert 语句形成弹出窗口实现查询。

3.1.4 在位置 index 插入值为 key 的节点

```
var div = document.getElementById('func4b')
   div.addEventListener('click',function(){
   const tmp1=document.getElementById("func4.1").value
   const tmp2=document.getElementById("func4.2").value
   list.insert(tmp1,tmp2);
   list_To_graph();
   graph.data(data);
   graph.render();
  })
```

图 13：指定位置插入代码

如图 13 所示，同 3.1.1 类似，此处为执行 insert 函数。

3.1.5 清空链表

```
var div = document.getElementById('func5b')    // 清空
    div.addEventListener('click',function(){
    data = {
      nodes: [],
      edges: []
    };
    list.clear();
    list_To_graph();
    graph.data(data);
    graph.render();
   })
```

图 14：清空链表代码

如图 14 所示，同 3.1.1 类似，此处为执行 clear 函数。

3.1.6 归并两个升序链表

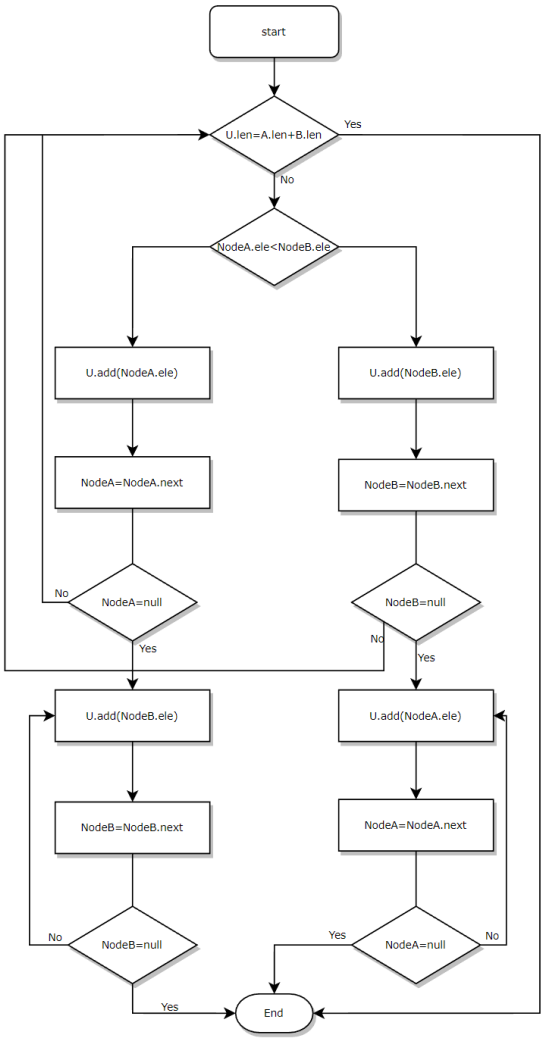


图 15: 归并链表算法流程图

如图 15 为合并两个升序序列的程序流程图，图中 U 为合并后的链表，A，B 分别为升序链表，NodeA 指 A 中节点，初始状态下 NodeA 为 A 的首节点，ele 为节点的值，next 是指向链表中下个节点的指针，len 为链表的长度。依照图中流程，可以将升序链表 A，B 按照数值归并形成新的升序链表 U。

```
var div = document.getElementById('func7b')
 div.addEventListener('click',function(){
  data = {
    nodes: [],
    edges: []
  };
  list.clear();
 let con1 = document.getElementById('func7A').value;
 var insert1 = con1.split(" ")
 let con2 = document.getElementById('func7B').value;
 var insert2 = con2.split(" ")
 var idx1=0,idx2=0;
 while(idx1<insert1.length && idx2<insert2.length)
  {
      if(insert1[idx1]*1<insert2[idx2]*1)  {
        list.add(insert1[idx1]*1);
        idx1++;
      }
      else {
        list.add(insert2[idx2]*1);
        idx2++;
      }
  }
  while(idx1<insert1.length)   {
    list.add(insert1[idx1]*1);
    idx1++;
  }
  while(idx2<insert2.length)   {
    list.add(insert2[idx2]*1);
    idx2++;
  }
  console.log(list);
 addedCount=0;
```

图 16：归并链表代码

如图 16 为代码实现过程，该归并过程可以表述为当 A 和 B 都不是空链表时，判断 A 和 B 哪一个链表的头节点的值更小，将较小值的节点添加到结果里，当一个节点被添加到结果里之后，将对应链表中的节点向后移一位。在循环终止的时候，A 和 B 至多有一个是非空的。由于输入的两个链表都是有序的，所以不管哪个链表是非空的，它包含的所有元素都比前面已经合并链表中的所有元素都要大。这意味着只需要简单地将非空链表接在合并链表 U 的后面，并返回合并链表 U 即可。

3.2 循环队列主要功能

3.2.1 尾部插入

```
var div = document.getElementById('func1b')     // 插入
      div.addEventListener('click',function(){
      const tmp=document.getElementById("func1").value
      //console.log(tmp);
      queue.enqueue(tmp);
      queue_To_graph();
      graph.data(data);
      graph.render();
      graph.fitView();
     })
```

图 17：尾部插入代码

如图 17 所示，在交互页面触发点击事件后，执行 enqueue 函数，enqueue 为 CircularQueue 类的成员函数，在 2.2.2 中已经详细描述。然后进行数据层与图形层的绑定 queue_To_graph()，然后利用 G6 引擎进行渲染。

3.2.2 删除队首

```
var div = document.getElementById('func2b')    // 删除
    div.addEventListener('click',function(){
    queue.dequeue();
    queue_To_graph();
    graph.data(data);
    graph.render();
    graph.fitView();
    })
```

图 18：删除队首代码

如图 18 所示，与 3.2.1 类似，此处为执行 dequeue 函数。

3.2.3 查询队首

```
var div = document.getElementById('func3b')
    div.addEventListener('click',function(){
    alert(queue.getFront());
   })
```

图 19：查询队首代码

如图 19 所示，与 3.2.1 类似，此处为执行 getFront 函数。

3.2.4 查询队尾

```
var div = document.getElementById('func4b')
    div.addEventListener('click',function(){
     alert(queue.getRear());
   })
```

图 20：查询队尾代码

如图 20 所示，与 3.2.1 类似，此处为执行 getRear 函数。

3.2.6 约瑟夫问题(N,M)

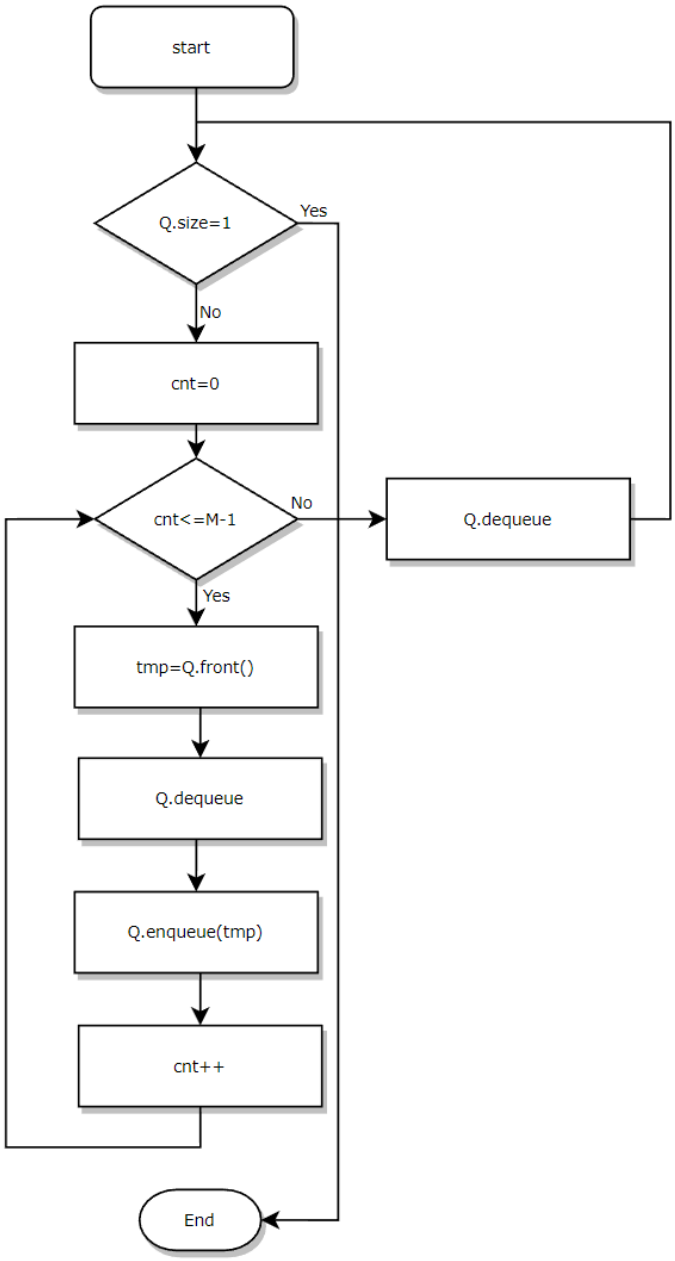


图 21：约瑟夫问题算法流程图

如图 21 为解决 N 个人报 M 数的约瑟夫问题的算法流程图。Q 为循环队列，front()为求出 Q 队首值的函数，dequeue 为队首出队，enqueue(key)为将 key 值入队。可以看到当 Q 的长度为 1 时，即仅剩最后一个值时程序结束，该值即为所求。

```
div.addEventListener('click',function(){
  queue.clear();
  data = {nodes: [],edges: []};
  const tmpn=document.getElementById("func6n").value;
  const tmpm=document.getElementById("func6m").value;
    async function w(){
      for(let i=1;i<=tmpn*1;i++)
      {
        queue.enqueue(i);
        queue_To_graph();
        graph.data(data);
        graph.render();
        graph.fitView();
      }
      await sleep(1000);
      for(let j=1;j<tmpn*1;j++)
      {
        for(let i=1;i<tmpm*1;i++)
        {
          var tmp=queue.getFront();
          queue.dequeue();
          queue.enqueue(tmp);
          queue_To_graph();
          graph.data(data);
          graph.render();
          graph.fitView();
          await sleep(1000);
        }
        queue.dequeue();
        queue_To_graph();
        graph.data(data);
        graph.render();
        graph.fitView();
        await sleep(1000);
      }
  }
  w();
})
```

图 22：约瑟夫问题代码

如图 22 为代码实现过程，整体流程思路为：先构成一个有 n 个结点的循环队列，从 1 开始计数，计到 m 时，对应结点从循环中删除被删除的下一个节点又从 1 开始计数，直到最后一个节点从队列中删除。

3.3 二叉搜索树主要功能

3.3.1 插入值为 key 的节点

```
var div = document.getElementById('func1b')
      div.addEventListener('click',function(){
      var tmp=document.getElementById("func1").value
      tree.insert(tmp*1);
      var Root=tree.root;
      addedCount=0;
      data = {
        nodes: [],
        edges: []
      };
     // console.log(tree);
      tree_To_graph(Root,-1);
      graph.data(data);
      graph.render();
      graph.fitView();
     })
```

图 23：插入节点代码

如图 23 所示，在交互页面触发点击事件后，执行 insert 函数，insert 为 BinarySearchTree 类的成员函数，在 2.2.3 中已经详细描述。然后进行数据层与图形层的绑定 tree_To_graph()，然后利用 G6 引擎进行渲染。

3.3.2 删除值为 key 的节点

```
var div = document.getElementById('func2b')
    div.addEventListener('click',function(){
    var tmp=document.getElementById("func2").value
    tree.remove(tmp*1);
    console.log(tree);
    var Root=tree.root;
    addedCount=0;
    data = {
     nodes: [],
     edges: []
   };
    tree_To_graph(Root,-1);
    console.log(data);
    graph.data(data);
    graph.render();
    graph.fitView();
   })
```

图 24：删除节点代码

如图 24 所示，与 3.2.1 类似，此处为执行 remove 函数。

3.3.3 查找值为 key 的节点

```
var div = document.getElementById('func3b')
div.addEventListener('click',function(){
for(let i=0;i<addedCount;i++)
{
   data.nodes[i].style={
     radius: 10,
      stroke: '#00C0A5',
      fill: '#92949F',
      fillOpacity: 0.45,
      lineWidth: 2,
    //fill: '#666',
     fontSize: 14,
     fontWeight: 'bold'
   }
}
var tmp=document.getElementById("func3").value
var root=tree.root;
search(root,(tmp*1));
change_target_color(tmp*1);
console.log(tree);
console.log(data);
graph.data(data);
graph.render();
graph.fitView();
)
```

图 25：查找结点代码

如图 25 所示，在交互页面触发点击事件后，执行 search 函数，通过 search 函数递归找到值为 key 的节点。然后改变该节点的颜色，渲染图形层呈现在画布上。

```
function search(node, key){
  if (node == null) {
    alert("No this number");
    return false;
  }
  if(key < node.key){
    search(node.left, key);
  } else if (key > node.key){
    search(node.right, key);
  } else {
    change_target_color(node.key);
   // alert("Finish");
    return true;
  }
  change_color(node.key);
}
```

图 26：search 函数

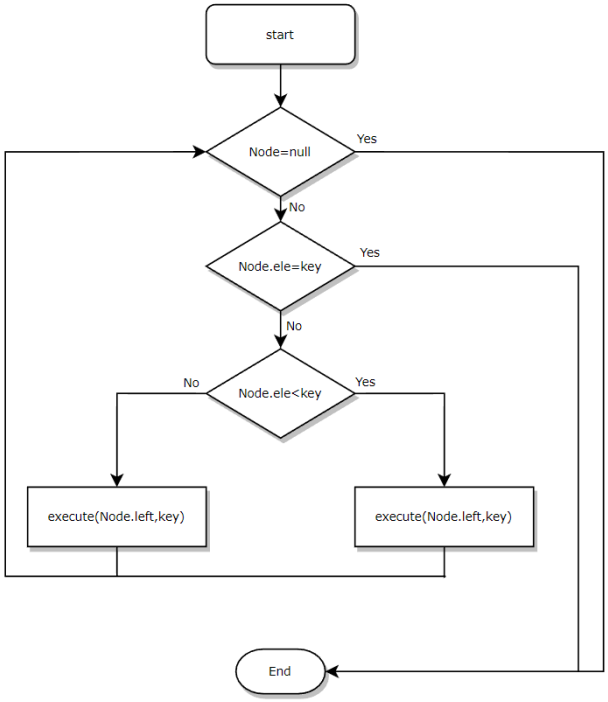


图 27：search 算法流程图

如图 26 所示为 search 函数的执行过程，在递归查找节点的过程中，如果当前节点的值大于 key，则递归查找左子树，反之查找右子树，直到找到值为 key 的节点。如果直到叶子节点仍找不到值为 key 的节点，则返回 false，并弹窗显示。右图为算法流程图。

在递归过程中调用了 change_color()和 change_target_color()函数，均为图形层渲染函数，在后文中会有详细描述。

3.3.4 清空

```
var div = document.getElementById('func4b')
  div.addEventListener('click',function(){
   tree.clear();
   console.log(tree);
   tree_To_graph(tree.root,-1);
   data = {
     nodes: [],
     edges: []
   };
   graph.data(data);
  graph.render();
  graph.fitView();
  })
```

图 28：clear 函数代码

如图 28 所示，与 3.2.1 类似，此处为执行 clear 函数。

3.3.5 前序遍历

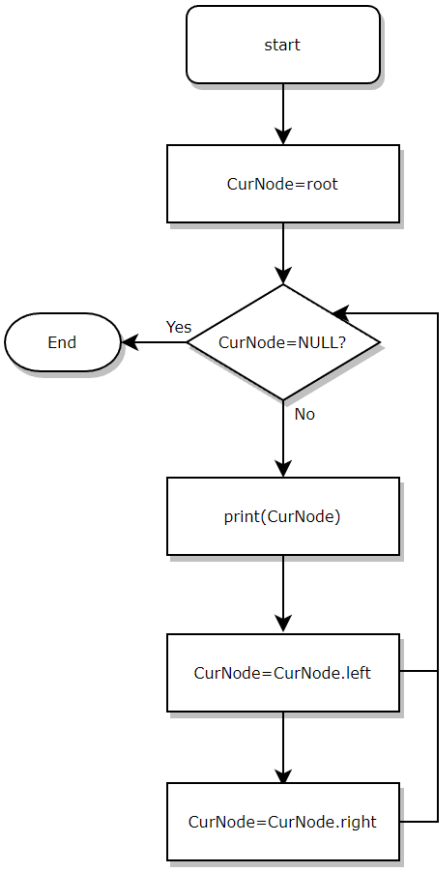


图 29：前序遍历算法流程图

如图 29 为二叉搜索树进行前序遍历的算法流程图，其中 CurNode 为当前节点，root 为二叉树根节点。二叉搜索树按照根、左、右的顺序进行递归遍历即为前序遍历。

```
preOrderTraverse(callback) {
  this.preOrderTraverseNode(this.root, callback);
}
preOrderTraverseNode(node, callback) {
  if (node != null) {
    callback(node.key);
    this.preOrderTraverseNode(node.left, callback);
    this.preOrderTraverseNode(node.right, callback);
  }
}
```

图 30：前序遍历实现代码

```
var div = document.getElementById('func5b')
div.addEventListener('click',function(){
      var list=[];
      const printNode = (value)=> list.push(value);
      tree.preOrderTraverse(printNode);
      console.log(list);
      console.log(tree);
      console.log(data);

        async function w(){
         for(let i=0;i<list.length;i++)
         {
         change_color(list[i]);
         await sleep(1000);
         }
        }
        w();
})
```

图 31：前序遍历实现代码

如图 30、图 31 为前序遍历的代码实现，当交互界面触发功能时，程序先执行 preOrderTravers 得到前序遍历序列，然后将这个序列中的节点与图形层中已经建立好的实例节点绑定，然后对其进行染色和动画渲染。其中 change_color()和 sleep 为动画渲染函数，将在后文中详细描述。

3.3.6 中序遍历

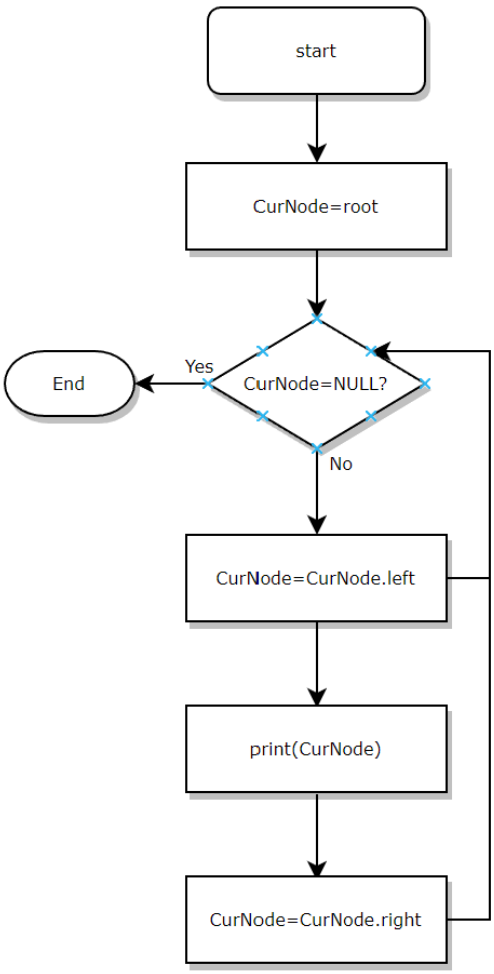


图 32：中序遍历算法流程图

如图 32 为二叉搜索树进行中序遍历的算法流程图，其中 CurNode 为当前节点，root 为二叉树根节点。二叉搜索树按照左、根、右的顺序进行递归遍历即为中序遍历。

```
inOrderTraverse(callback) {
  this.inOrderTraverseNode(this.root, callback);
}
inOrderTraverseNode(node, callback){
  if(node !== null){
    this.inOrderTraverseNode(node.left, callback);
    callback(node.key)
    this.inOrderTraverseNode(node.right, callback);
  }
}
```

图 33：中序遍历代码

如图 33 为中序遍历的代码实现，网页交互事件触发以及动画渲染与前序遍历相同，已在 3.3.5 中介绍。

3.3.7 后序遍历

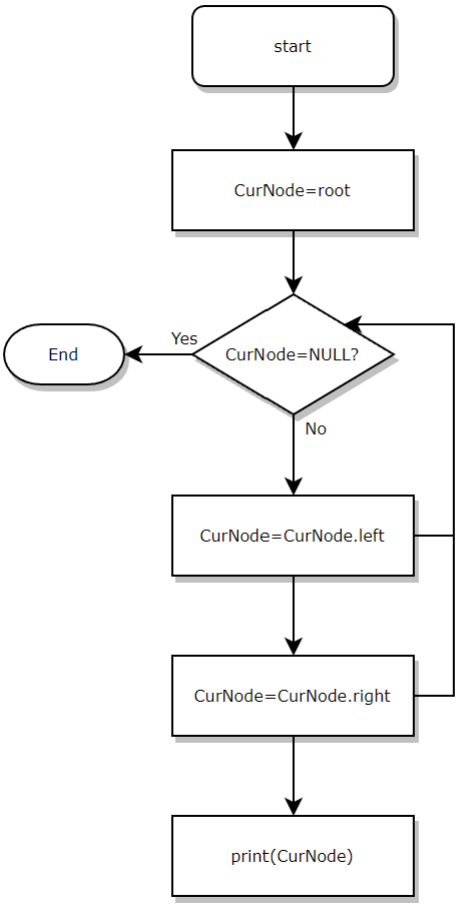


图 34：后序遍历算法流程图

如图 34 为二叉搜索树进行后序遍历的算法流程图，其中 CurNode 为当前节点，root 为二叉树根节点。二叉搜索树按照左、右、根的顺序进行递归遍历即为后序遍历。

```
postOrderTraverse(callback) {
  this.postOrderTraverseNode(this.root, callback);
}
postOrderTraverseNode(node, callback) {
  if (node != null) {
    this.postOrderTraverseNode(node.left, callback);
    this.postOrderTraverseNode(node.right, callback);
    callback(node.key);
  }
}
```

图 35：后序遍历代码

如图 35 为后序遍历的代码实现，网页交互事件触发以及动画渲染与前序遍历相同，已在 3.3.5 中介绍。

3.3.8 层序遍历

```
function levelTraverse(root) {
  let res = [];
    let queue = [root];
    while (queue.length != 0) {
        let node = queue.shift();
        res.push(node.key);
        if (node.left != null) queue.push(node.left);
        if (node.right != null) queue.push(node.right);
    }
    return res;
}
```

图 36：层序遍历代码

如图 36 所示为二叉树层序遍历的代码实现过程，res 数组为所求层序遍历结果数组。层序遍历的过程需要维护一个队列数据结构 queue，从树根开始，按照 BFS 的原理进行遍历。首先将树根入队，然后每次取出队首元素，将队首元素加入 res 结果数组，然后将队首元素的左子节点和右子节点分别入队，此时能够满足处在同一深度的节点在 res 结果数组中的位置相邻。重复此过程直到队列为空，此时 res 数组中的内容即为二叉树层序遍历的序列，同层元素按照左子节点在前，右子节点在后的顺序排列。

3.3.9 二叉树转化为平衡树

如图 37 为二叉树转化为平衡树的算法流程图。inOrderTraverse Set 为二叉树的中序遍历集合，即上升序列；Node 时当前节点，execute 为递归函数。右图展示了递归函数的具体流程。

为解决上述问题，需要先求出二叉树的中序遍历，即所有元素从小到大排列的数组，然后每次选取数组下标为 length/2 的节点作为子树的根节点，然后以此节点为中心将 Set 划分为左右两个子 Set，以左子 Set 递归构建左子树，右子 Set 递归构建右子树。按照这个方式，首次将 Set[length/2]作为平衡树的根节点，再不断递归求解左子树和右子树，即可得到平衡二叉搜索树。由于最初的 Set 序列是严格递增的，因此递归建树的过程中，每棵子树都严格满足二叉搜索树的性质，因此算法的正确性得到证明。

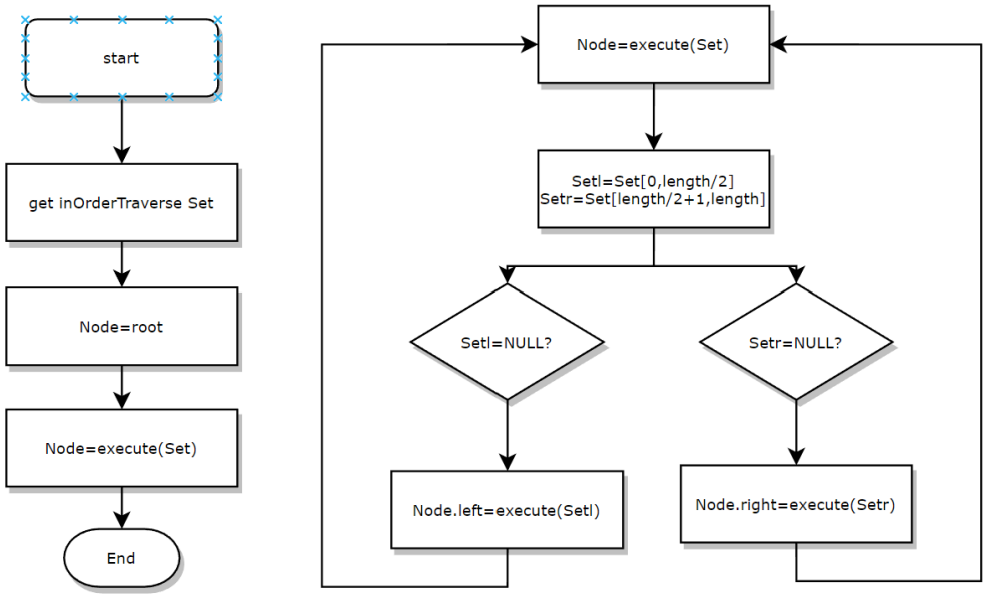


图 37：二叉树转化平衡树算法流程图

```
function  BSTtoAVL(list) {
  var len = list.length;
  if(list == null ) return null;
  var mid = Math.floor(len / 2);
  var root = new Node(list[mid]);
  var left = copyOfRange(list,0,mid);
  var right = copyOfRange(list,mid+1,len);
  // console.log(left);
  // console.log(right);
  if(left!=null){
    root.left=BSTtoAVL(left);
    root.left.fa=root;
  }
  if(right!=null){
    root.right=BSTtoAVL(right);
    root.right.fa=root;
  }
  return root;
}
function copyOfRange(list,start,end){
  if(start > end-1) return null;
  if(start == end-1)  return [list[start]];
  var tmp=[];
  for(let i=start;i<end;i++)
  {
    tmp.push(list[i]);
  }
  return tmp;
}
```

图 38：二叉树转化平衡树主要函数

如图 38 所示为该算法的主要函数代码实现，copyOfRange 函数是将数组分割的函数，能够实现划分左子树序列、右子树序列的功能。

3.4 图形层函数

3.4.1 数据层与图形层绑定

正如在 1.1 中所描述的，本软件中数据层和图形层间需要进行绑定，数据层的改变会响应到图形层。图形层的数据存储在 data 中，以点集和边集的形式储存，每个点有相应的 x,y 坐标，识别符 id，以及显示在图形上的内容 label 标签。因此将数据层中各个数据结构中存储的信息对应绑定到 data 中，然后自定义其在画布中的位置（x，y 坐标信息），然后根据 G6 引擎中的样式 API 调整图形样式，再利用 G6 引擎中 graph.data(data)读取数据，graph.render()渲染视图，就可以让图形在画布中指定位置呈现。

对于三种不同的数据结构，其数据层和 data 的绑定方式会有差异，但整体的思想相近。当数据层数据发生变化时，先将 data 清空，然后按照数据结构中点的顺序，将节点信息手动添加到 data，对于三种不同数据结构，点坐标的设定规则不同，进而能呈现不同的可视化效果。而通过 data 反向检索数据层数据的方式则是按照节点或边在 data 中定义的 id 和 label 检索其在数据结构中的对应，找到对应的数据后就可以进行数据层的修改。如下图 39，40，41，42 分别为链表、循环队列、二叉搜索树数据层和图形层的绑定函数的代码实现。

```
function list_To_graph()
{
    data = {
      nodes: [],
      edges: []
    };
    let h=list.head;
    if(h != null)
    {
        let addedCount=0;
        for(let i=0;i<list.size;i++) {
            data.nodes[i]={
              x: 50+addedCount*120,
              y: 200,
              id: `node-${addedCount}`,
              label: h.element};
            addedCount++;
            h=h.next;
        }
        for(let i=0;i<list.size-1;i++){
          data.edges[i] ={
            source: `node-${i}`,
            target: `node-${i+1}`,
            id:`edge-${i}`,
            style:{
              stroke:'#000000',
              lineWidth:3,
              endArrow:true,
              startArrow:false
            }
          };
        }
    }
}
```

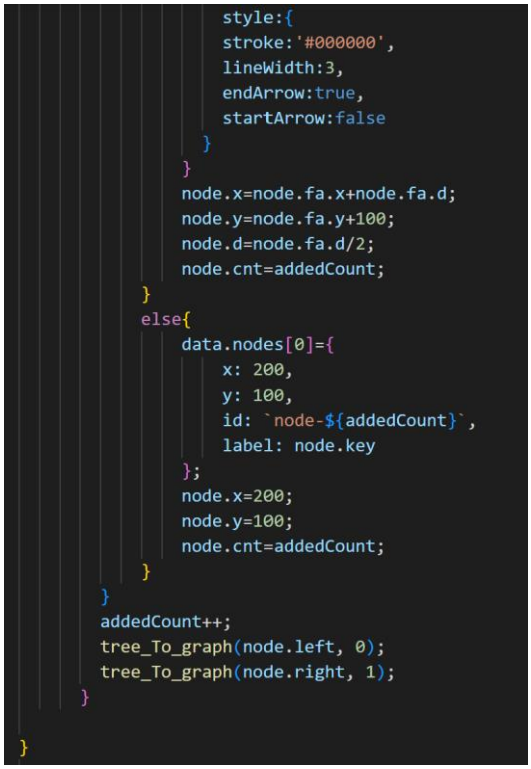
图 39：单链表数据绑定

```
function queue_To_graph()
{
    data = {
      nodes: [],
      edges: []
    };
    let addedCount=0;
    for(let i=queue.headIdx; i<queue.tailBackIdx; i++)
    {
        let h=queue.Q[i%queue.capacity];
        //console.log(h);
        data.nodes[addedCount]={
          x: 50+addedCount*120,
          y: 200,
          id: `node-${addedCount}`,
          label: h
        };
        addedCount++;
    }
    addedCount=0;
    for(let i=queue.headIdx; i<=queue.tailBackIdx; i++)
    {
        data.edges[addedCount] ={
          source: `node-${addedCount}`,
          target: `node-${addedCount+1}`,
          id:`edge-${addedCount}`,
          style:{
            stroke:'#000000',
            lineWidth:3,
            endArrow:true,
            startArrow:false
          }
        };
        addedCount++;
    }
}
```

图 40：循环队列数据绑定

```
function tree_To_graph(node,dir){
    if(node !== null){
        if( dir == 0 ){
            data.nodes[addedCount]={
                x: node.fa.x-node.fa.d,
                y: node.fa.y+100,
                id: `node-${addedCount}`,
                label: node.key
            };
            node.x=node.fa.x-node.fa.d;
            node.y=node.fa.y+100;
            node.d=node.fa.d/2;
            node.cnt=addedCount;
             data.edges[addedCount-1]={
                source: `node-${node.fa.cnt}`,
                target: `node-${addedCount}`,
                id:`edge-${addedCount-1}`,
                style:{
                stroke:'#000000',
                lineWidth:3,
                endArrow:true,
                startArrow:false
              }
            }
        }
        else{
            if( dir == 1)
            {
                data.nodes[addedCount]={
                    x: node.fa.x+node.fa.d,
                    y: node.fa.y+100,
                    id: `node-${addedCount}`,
                    label: node.key
                };
                data.edges[addedCount-1]={
                    source: `node-${node.fa.cnt}`,
                    target: `node-${addedCount}`,
                    id:`edge-${addedCount-1}`,
```

图 41：二叉搜索树数据绑定



```
                    style:{
                    stroke:'#000000',
                    lineWidth:3,
                    endArrow:true,
                    startArrow:false
                  }
                }
                node.x=node.fa.x+node.fa.d;
                node.y=node.fa.y+100;
                node.d=node.fa.d/2;
                node.cnt=addedCount;
            }
            else{
                data.nodes[0]={
                    x: 200,
                    y: 100,
                    id: `node-${addedCount}`,
                    label: node.key
                };
                node.x=200;
                node.y=100;
                node.cnt=addedCount;
            }
        }
        addedCount++;
        tree_To_graph(node.left, 0);
        tree_To_graph(node.right, 1);
    }

}
```

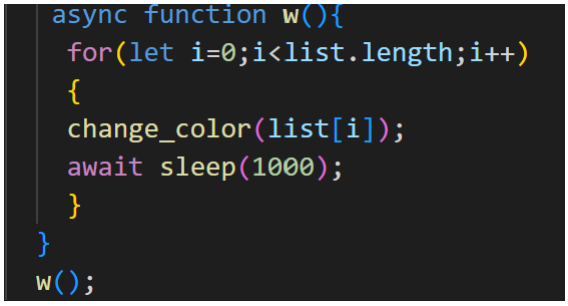
图 42：二叉搜索树数据绑定

3.4.2 染色&动画

为了可视化展现算法的执行流程，本程序中借助 G6 引擎设置了节点染色和动画的渲染。

由于染色不会改变数据，因此仅需要借助 3.4.1 中提到的检索一一对应方式，找到节点在图形层中的实例调整颜色样式即可。

动画则需要借助计时器和染色的功能，由于 JavaScript 单线程运行，因此利用 await 关键字，染色后在异步函数中暂停进程一定时间后重新执行染色，就可以形成动画效果。



```
async function w(){
 for(let i=0;i<list.length;i++)
 {
 change_color(list[i]);
 await sleep(1000);
 }
}
w();
```

图 43：异步染色+暂停函数

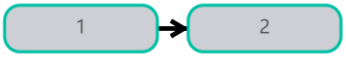
如上图 43 所示，w()为执行了染色和暂停的异步函数，能够实现对应的动画功能。在此软件中，进行动画渲染的内容很多，原理均与如上所述相同。

4、 测试

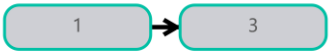
4.1 功能测试

4.1.1 单链表功能测试

1）尾部插入：在尾部插入“3”



2）删除位置为 1 的节点



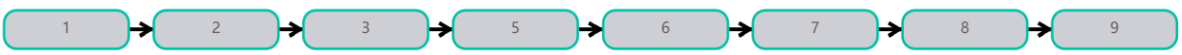
3）查询位置为 0 的节点

4）在位置 1 插入值为 4 的节点

5）清空链表

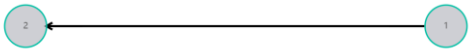
6）集合方式读入链表 1 2 3 4 5

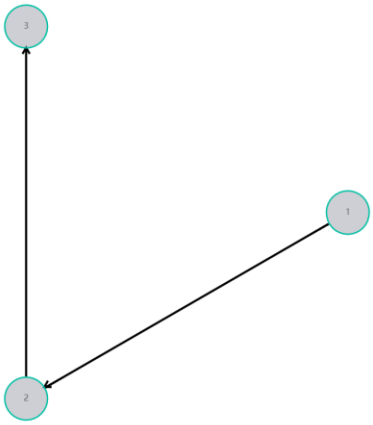
7）读入两个升序链表 A:1 5 6 9 B:2 3 7 8 进行归并



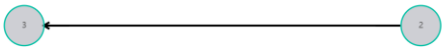
4.1.2 循环队列功能测试

1）队尾插入值为 2 的节点





2）删除队尾元素



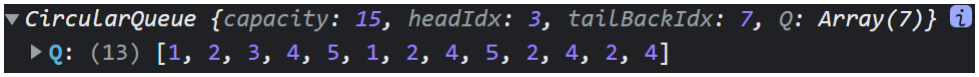
3）查询队首节点值

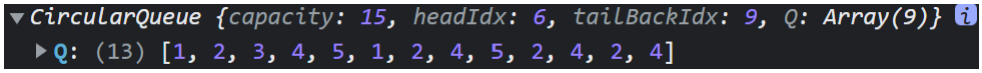
4）查询队尾节点值

5）清空队列

6）约瑟夫环问题

由于约瑟夫环问题的求解在可视化页面以动画形式演示不方便呈现在报告中，因此测试以控制台数据的方式进行。输入 N=5，M=3

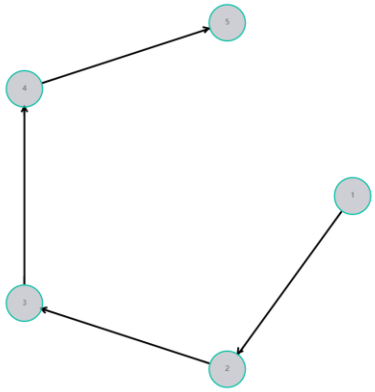




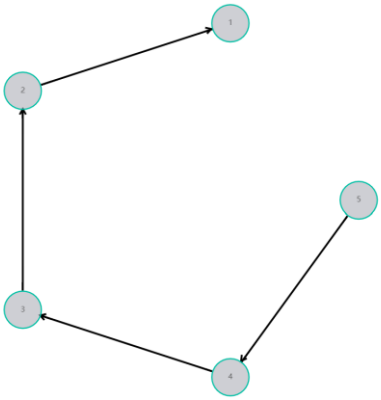
```
CircularQueue {capacity: 15, headIdx: 9, tailBackIdx: 11, Q: Array(11)}
  Q: (13) [1, 2, 3, 4, 5, 1, 2, 4, 5, 2, 4, 2, 4]
CircularQueue {capacity: 15, headIdx: 12, tailBackIdx: 13, Q: Array(13)}
  Q: (13) [1, 2, 3, 4, 5, 1, 2, 4, 5, 2, 4, 2, 4]
```

可以通过 headIdx 和 tailBackIdx 验证其结果正确，最后剩下的节点为 4。

此处仅展示可视化页面初态和末态

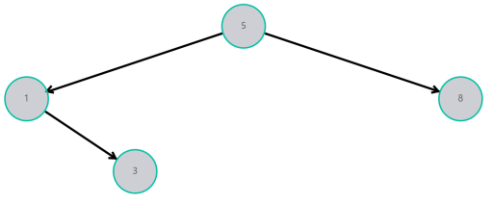


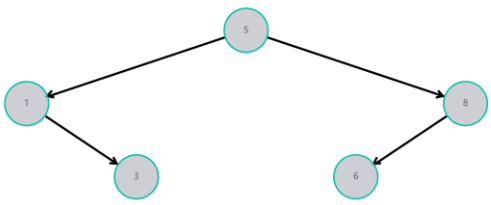
7）以集合方式读入 5 4 3 2 1



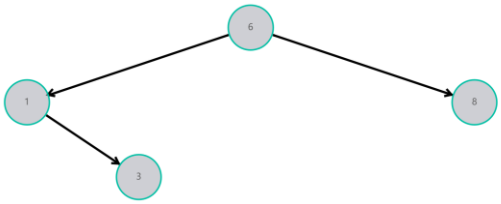
4.1.3 二叉搜索树功能测试

1）插入值为 6 的结点

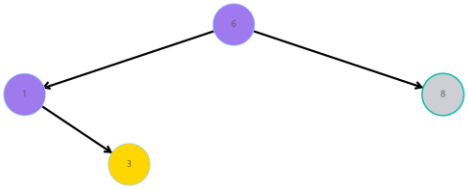




2）删除值为 5 的节点

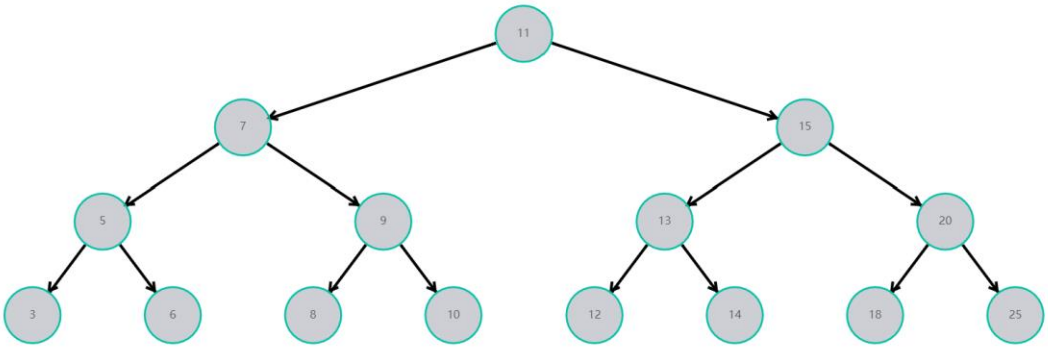


3）查找值为 3 的节点



4）清空二叉树

5）求前序遍历 二叉树为（11 7 15 5 3 9 8 10 13 12 14 20 18 25 6）



由于该功能的可视化过程需要动画，此处测试控制台显示前序遍历序列

```
▶ (15) [11, 7, 5, 3, 6, 9, 8, 10, 15, 13, 12, 14, 20, 18, 25]   bst_func.js:125
```

6）求中序遍历 二叉树同（5）

由于该功能的可视化过程需要动画，此处测试控制台显示中序遍历序列

```
▶ (15) [3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 18, 20, 25]   bst_func.js:144
```

7）求后序遍历 二叉树同（5）

由于该功能的可视化过程需要动画，此处测试控制台显示后序遍历序列

```
▶ (15) [3, 6, 5, 8, 10, 9, 7, 12, 14, 13, 18, 25, 20, 15, 11]   bst_func.js:163
```

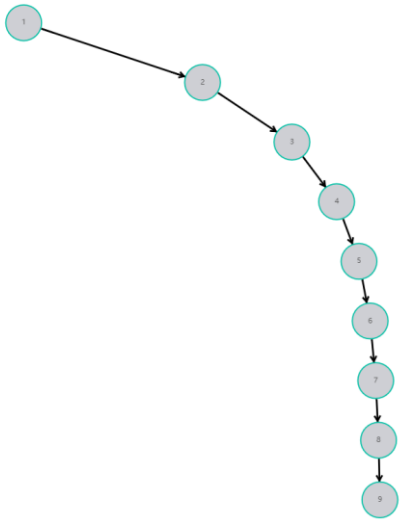
8）求层序遍历 二叉树同（5）

由于该功能的可视化过程需要动画，此处测试控制台显示层序遍历序列

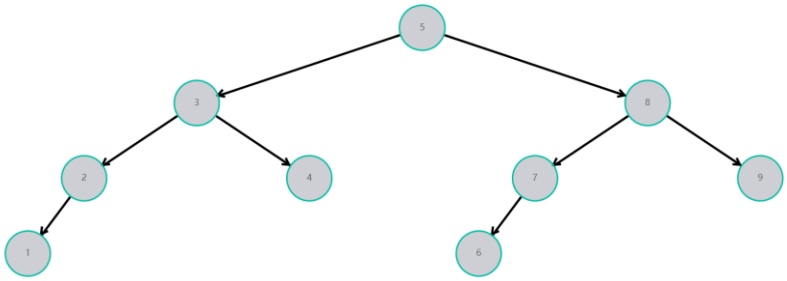
```
▶ (15) [11, 7, 15, 5, 9, 13, 20, 3, 6, 8, 10, 12, 14, 18, 25]   bst_func.js:181
```

9）二叉树转平衡树

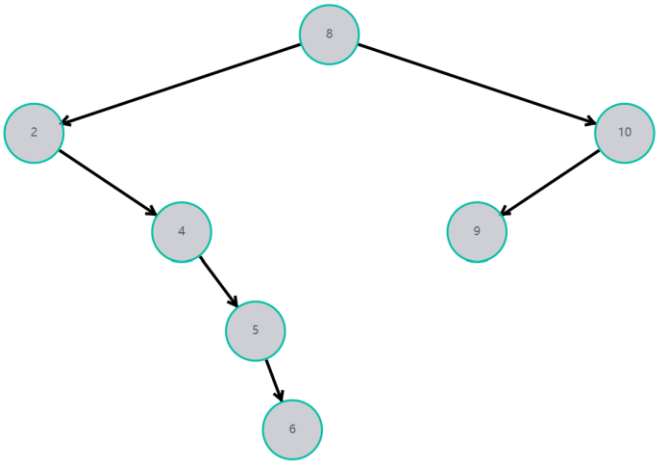
原二叉树：



平衡树：



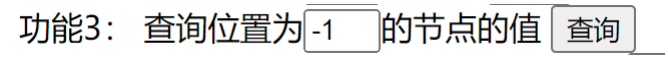
10）以集合方式读入（8 2 4 10 5 9 6）



4. 2 错误数据测试

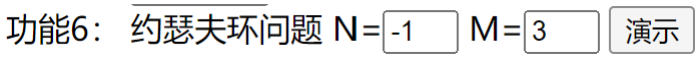
4.2.1 单链表错误数据测试

当查询位置为 index 的值，当 index 为负数时，系统会弹窗提示“非法查询”。



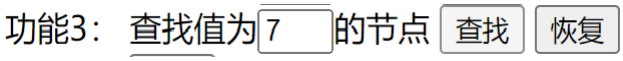
4.2.2 循环队列错误数据测试

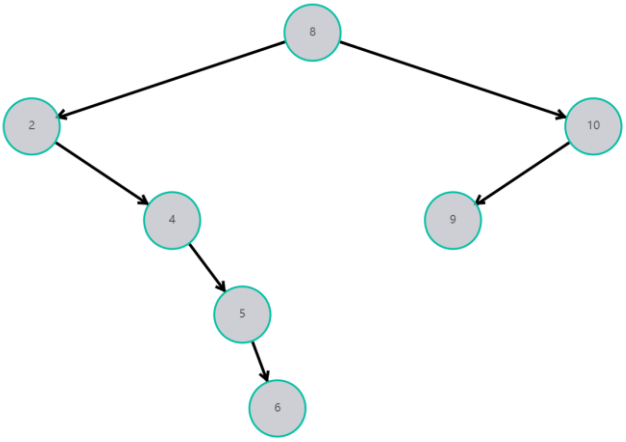
当读入约瑟夫问题的 N 和 M 时，若 N 或 M 小于 0 时，系统会弹窗提示“非法的约瑟夫问题”。

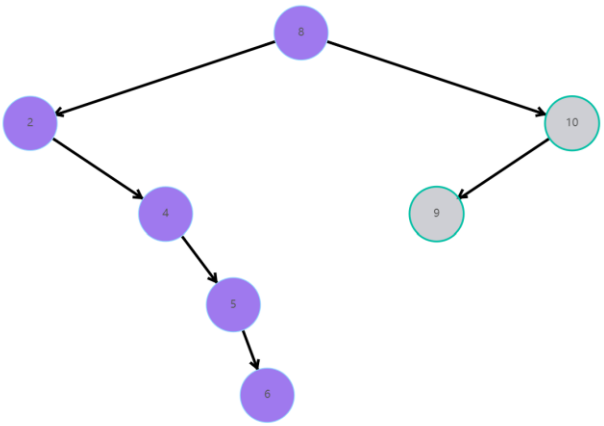


4.2.3 二叉搜索树错误数据测试

当查找二叉树中不存在的节点时，会以染色的方式显示出查找的路径，并且弹窗显示 No this number。







5、 总结与提高

此次数据结构课设课设是我真正意义上独立完成的第一个软件开发类项目，能够完整地完成此次数据结构课设让我的能力得到了全方位的提高。

此次数据结构课设的是一个非常完整的任务，需要进行设计，调查，选取方案，开发，测试，迭代改进这一系列完整的过程，这对于我来说有长足的锻炼意义。在此之前，我仅仅能够从数学层面理解数据结构，而经历这次完整开发流程，我理解到开发工作不仅仅是写代码，在写代码前的构思和工具选取，以及架构搭建都是非常重要的。这些工作的顺利开展能够确保实际在写代码时候的效率，并且能够避免出现难以调和的冲突。此外这次课设过程中，由于使用的是 WEB 开发组件，并不像 C 语言开发时有功能强大的 IDE 和非常成熟的 DEBUG 工具，WEB 开发中追踪每一步都需要手动进行，并且调试起来非常苦难，这就要求我进行高质量的编码，并且将代码解析细化，边写边调，确保已经写好的功能不出问题。由于这次数据结构课设是我完成的代码量最大的项目，我认识到代码量的提升能够加深开发者对开发工具的理解，大大地扩充开发手段，提升开发效率。此前我完全没有上手写过 HTML+CSS+JAVASCRIPT，这次课设之后，我对这些 WEB 开发组件有了更深入的了解，能够使用其完成其他的开发任务。

万丈高楼平地起，每一个优秀开发者都是经过各种历练才能够不断进步，数据结构课设的任务确实很大意义上让我非常头疼，但完整完成后受益匪浅。