

算法与数据结构设计报告

（ 2021 / 2022学年 第 一 学期）

题 目： A.AVL树的判定问题

B.火车调度

|  |  |
| --- | --- |
| **专 业** | **信息安全** |
| **学 生 姓 名** |  |
| **班 级 学 号** |  |
| **指 导 教 师** |  |
| **指 导 单 位** | **计算机科学与技术系** |
| **日 期** | **2021.10.25-2021.11.5** |

|  |  |  |
| --- | --- | --- |
| **支撑指标点** | **评价准则** | **计分（每项10分）** |
| 课程目标1：文献调研与资料收集能力，问题发现、研究、分析与解决能力（20分） | 1、能够掌握算法与数据结构设计的相关基础知识，并能够针对求解的工程问题，收集资料进行合理的分析与设计 |  |
| 2、通过调研，能够选择合适的程序设计语言与编程开发平台，对求解的工程问题进行编程实现 |  |
| 课程目标2：通过课程设计，培养学生综合应用算法和数据结构等知识解决工程问题的实践能力（20分） | 3、能够给出数据结构和算法的设计描述，给出关键算法的流程图或伪代码，并给出各算法之间的结构关系描述 |  |
| 4、具备一定的人机交互设计意识，人机交互设计合理、友好，操作简便 |  |
| 课程目标3：培养解决工程问题的开发工具运用能力，能够利用程序设计软件或系统对问题求解进行模拟和实现，能够设计测试数据验证问题解决方法的正确性，并能够对问题解决方法的性能和效率进行分析（40分） | 5、具备一定的算法与数据结构设计分析能力，能够完成课题要求的各项任务和指标 |  |
| 6、能够结合计算机软硬件资源，合理选用算法、数据结构、数据存储方式等技术手段，对求解的工程问题进行有效建模和求解 |  |
| 7、具备一定自学能力与探索创新意识，能够充分利用教科书及其资源（如网络等）自学新知识与新技能 |  |
| 8、掌握调试方法与工具，对程序开发过程中出现的问题进行分析、跟踪与调试，并能够进行充分测试 |  |
| 课程目标4：选择同类课题的学生能够通过讨论和交流解决课程设计中的难题，能在实验报告中准确阐述课程设计的内容，能够清晰陈述观点和回答问题（20分） | 9、能够正确、完整地回答指导教师关于课题的问询，反映其对课题内容，以及相关的工程基础知识具有较好的理解和掌握 |  |
| 10、具备一定的语言表达能力与文字处理能力，能够结合复杂工程问题撰写报告，报告内容和实验数据详实，格式规范 |  |
| 算法与数据结构设计能力测评总分 | |  |
| **指导教师： 年 月 日** | | |
| **备注：** | | |

**AVL树的判定问题**

**一、课题内容和要求**

本课题目标系统“AVL树的判定系统”的功能框架图如图1-1所示。

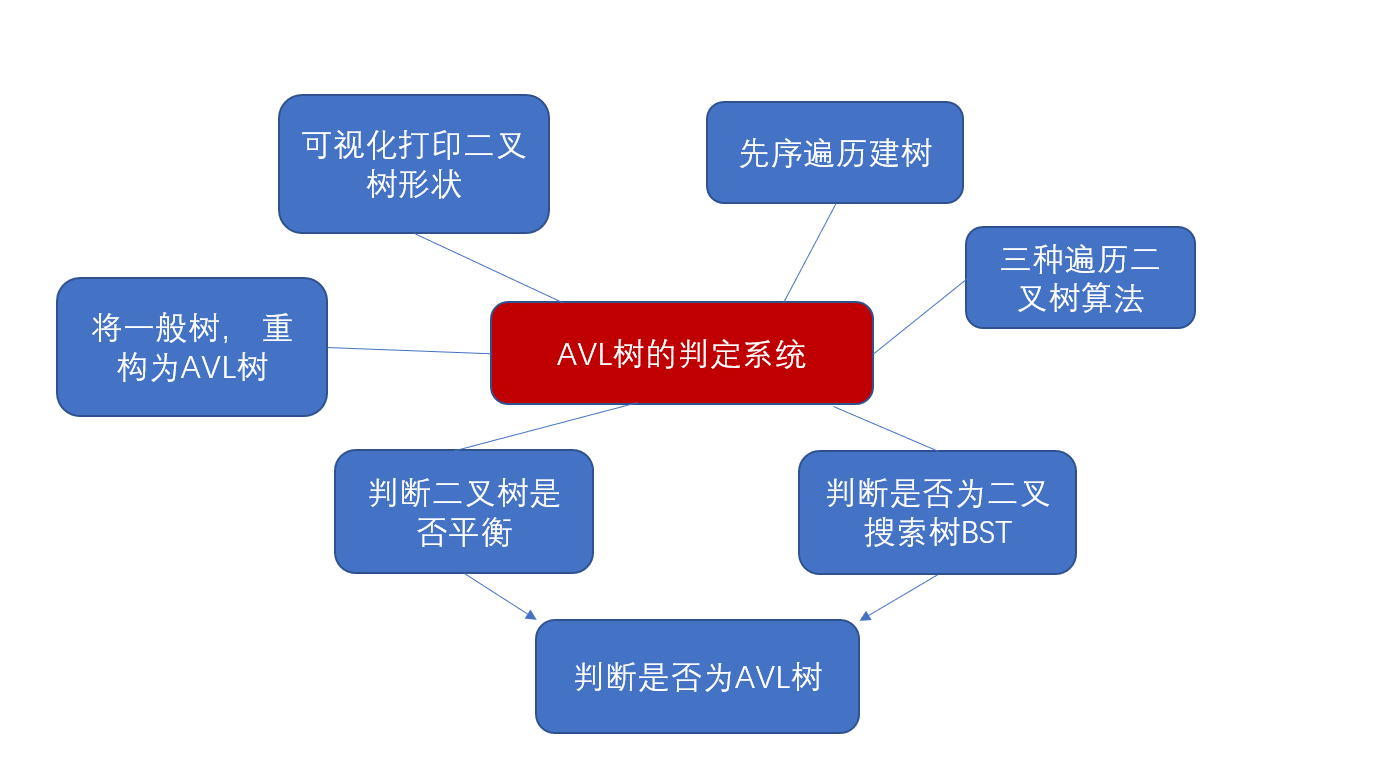


图1-1 功能框架图

本系统所支持以下功能：

1.支持链表形式的二叉树的先序遍历建树。

2.支持先序遍历，中序遍历，后序遍历所建立的二叉树。

3.支持判断所建立的二叉树，是否为二叉搜索树BST。

4.支持判断所建立的二叉树，是否平衡。

5.支持根据(2)和(3)，判断所建立的二叉树，是否为AVL树。

6.支持将不满足AVL树性质的二叉树，调整为AVL树。

7.支持可视化打印一颗二叉树的结构

**二、数据结构说明**

1.二叉树结构体

typedef struct BinaryTreeNode{

int value; //二叉树节点的值

struct BinaryTreeNode \*LChild, \*RChild; //二叉树当前节点的左右孩子

}BinaryTreeNode;

2.几个全局变量

int isBST=1;//是否为BST

int node\_arr[100]={0};//树中非空节点的值存入该数组

int node\_num=0;//树中节点的数量

3.第三方库

C++ STL库中实现的queue队列，queue<BinaryTreeNode\*> q\_node和 queue<int> q\_value，用于层次遍历

**三、算法设计**

1.先序遍历创建二叉树算法(同时判断是否是BST)

**实现函数：**

BinaryTreeNode \*PreCreateBt\_IsBST(BinaryTreeNode \*t,int parent\_value,int flag)

**设计思路：**

该算法改编自《数据结构》教材中的先序遍历建树，每个节点之间有空格，空节点用0表示。除此之外还加上了判断所建树是否为二叉搜索树BST的功能。

其中t为当前构建的节点的指针；parent\_value 为当前节点的父节点的值；flag 为当前节点的类型，flag=0代表当前构建的节点是左子节点 ，flag=1代表当前构建的节点是右子节点 ，flag=2代表当前构建的节点是根节点。最后返回指针t。

如果当前构建的节点是根节点，那么直接递归调用函数，继续构建其左右子节点；如果当前构建的节点是左子节点，那么先判断当前节点与父节点的大小，如果当前节点的值大于其父节点，则不满足二叉搜索树性质，将全局变量isBST置零，然后递归调用函数，继续构建其左右子节点；如果当前构建的节点是右子节点，那么先判断当前节点与父节点的大小，如果当前节点的值小于其父节点，则不满足二叉搜索树性质，将全局变量isBST置零，然后递归调用函数，继续构建其左右子节点。

虽然按照常规做法，应该将建树与判断是否为BST解耦，但我将它们合二为一，效率更高。

时间复杂度：O(n)，其中n是二叉树中的节点个数。

2. 先序遍历，中序遍历，后序遍历所构建的二叉树

**实现函数：**

void Transverse(BinaryTreeNode \*t)

**设计思路：**《数据结构》课本上的基础操作，采用递归思想，没有什么说的。

3. 判断所建立的二叉树，是否平衡

**实现函数：**

bool isBalanced(struct BinaryTreeNode\* root)和int Balanced\_height(struct BinaryTreeNode\* root)

**设计思路：**

isBalanced函数调用Balanced\_height函数。如果以root为根节点的二叉树平衡，则Balanced\_height函数返回该树的高度（只有一个根节点的情况算高度1），否则返回-1。isBalanced函数在Balanced\_height函数的基础上进行封装，返回以root为根的二叉树是否平衡，返回的是一个布尔值。

在Balanced\_height函数中，我采用了自底向上的思想，类似于后序遍历，对于当前遍历到的节点，先递归地判断其左右子树是否平衡，再判断以当前节点为根的树是否平衡。如果以当前节点为根的树是平衡的，则返回其高度（高度一定是非负整数），否则返回-1。如果一个二叉树中存在一个子树不平衡，则整个二叉树一定不平衡。

时间复杂度：O(n)，其中n是二叉树中的节点个数。使用自底向上的递归，每个节点的计算高度和判断是否平衡都只需要处理一次，最坏情况下需要遍历二叉树中的所有节点，因此时间复杂度是 O(n)。

4.判断是否为AVL树

AVL树又叫平衡二叉查找树，它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。根据1中求得的isBST标签和3中求得的isBalanced，判断是否为AVL树。

5.将不满足AVL树性质的二叉树，调整为AVL树

**实现函数：**

BinaryTreeNode\* sortedArrayToAVL(int\* nums, int numsSize)

BinaryTreeNode\* sortedArrayToAVL\_helper(int\* nums, int left, int right)

其中sortedArrayToAVL中的形式参数nums为待转化二叉树的节点数组的首地址，numsSize为待转化二叉树的节点数量。sortedArrayToAVL调用子函数sortedArrayToAVL\_helper，递归地重构AVL树。

**设计思路：**

首先在先序遍历创建二叉树的时候，把二叉树的节点保存进节点数组，然后从小到大排序。排过序后的节点数组其实就是二叉搜索树中序遍历的结果。接下来我们可以选择数组中间数字作为二叉搜索树的根节点，这样分给左右子树的数字个数相同或只相差 1，这样同时可以使得二叉搜索树保持平衡，如此重构的二叉树既满足搜索性质，又满足平衡性质，因此可以重构为AVL树。如果数组长度是奇数，则根节点的选择是唯一的，如果数组长度是偶数，则可以选择中间位置左边的数字作为根节点或者选择中间位置右边的数字作为根节点，选择不同的数字作为根节点则创建的AVL树也是不同的。确定AVL树的根节点之后，其余的数字分别位于AVL树的左子树和右子树中，左子树和右子树分别也是AVL树，因此可以通过递归的方式创建AVL树。

6.可视化打印一颗二叉树的结构

**实现函数：**

void Print\_Tree(BinaryTreeNode \*t)

输入一个二叉树的根节点t，即可可视化打印二叉树的结构

**设计思路：**

创建两个队列queue<BinaryTreeNode\*> q\_node，存放节点的指针和queue<int> q\_value，存放节点的值。首先将根节点和根节点的值放入两个队列，然后层次遍历二叉树。如果当前遍历节点的左孩子非空，则按层次遍历的正常流程处理，将左孩子的指针和值分别入队列；如果当前遍历节点的左孩子为空，则不能忽略，此时要创建一个“空”的二叉树节点，它的值我定为-1，左右孩子都为空，然后把这个“空”节点的指针和值分别入两个队列。我这样做的目的是完整的记录一个“满二叉树”，即把二叉树补成一个“满二叉树”，以便后续的可视化打印。针对右孩子的情况同理。

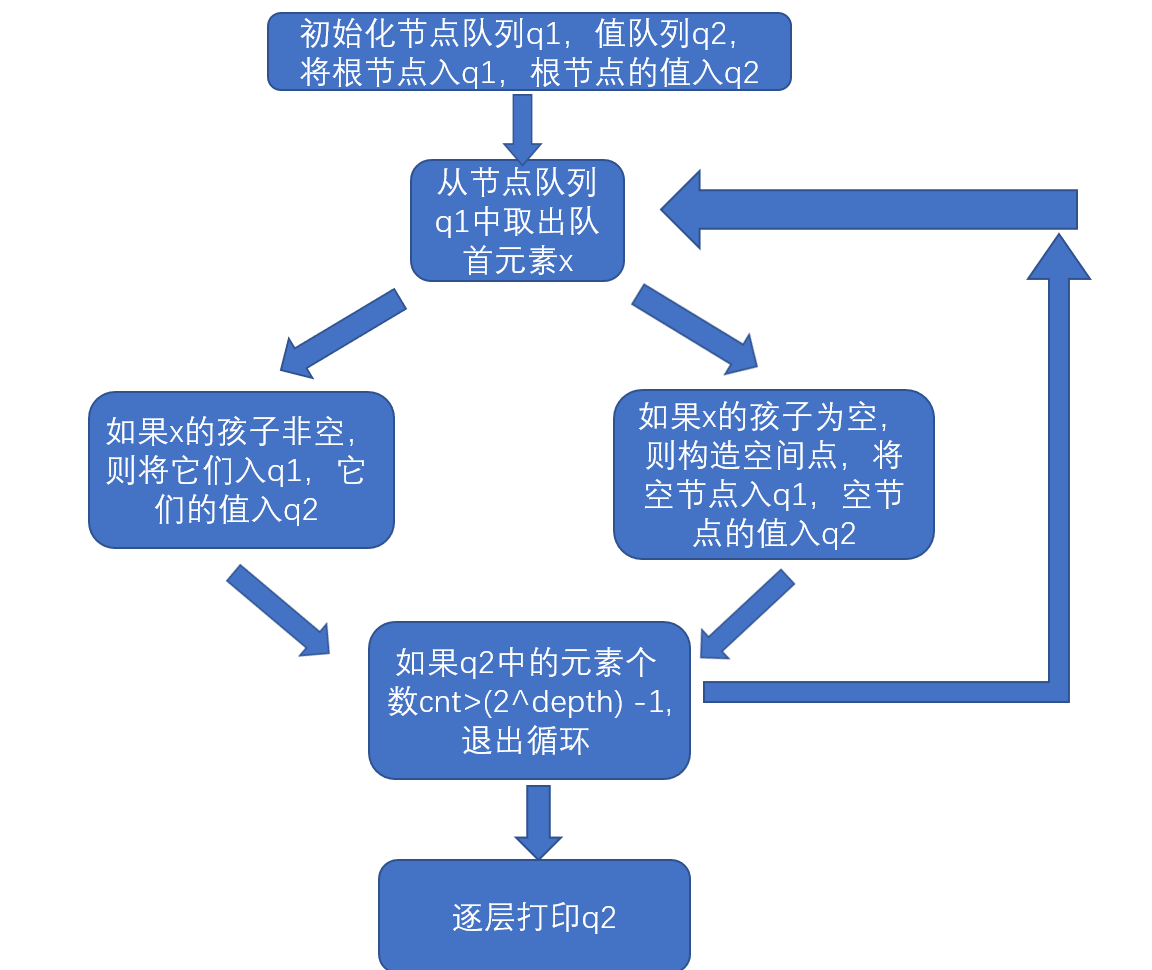


图1-2 算法流程图

另外要给层次遍历的循环再加一个限制条件：当前遍历到的二叉树深度depth大于事先求得的二叉树深度时，退出层次遍历，以防止无止尽地构造“空节点”。因为我们模拟了一个“满二叉树”，所以当前遍历到的深度可以通过已经入队列的节点数量求得，具体是当已经入队列的节点数量cnt >（2^depth）-1 时，depth++ 。

当层次遍历结束时，q\_value队列中存放着完整的“满二叉树”层次遍历序列，其中空的节点的值我们用-1表示。接下来即可用循环逐层打印出二叉树的形状。这里有个小瑕疵：如果每个节点的值的位数不一样，比如有的时一位数，有的是三位数，打印出的二叉树会走形，所以尽量使用同样位数的值创建二叉树。

**四、详细设计**

下面详细展示一些关键模块的代码和注释

1.几个全局变量

int isBST=1;//是否为BST

int node\_arr[100]={0};//树中节点存入数组

int node\_num=0;//树中节点的数量

2.先序遍历创建二叉树算法(同时判断是否是BST)

BinaryTreeNode \* PreCreateBt\_IsBST(BinaryTreeNode \* t, int parent\_value, int flag) {

int ch;

cin >> ch;

getchar();

if (ch == -1) //输入为-1表示这里建立空二叉树，即遍历算法的空操作

{

t = NULL;

} else {

t = (BinaryTreeNode \* ) malloc(sizeof(BinaryTreeNode));

t - > value = ch; //构造根结点

node\_arr[node\_num++] = ch; //把新创建节点的值存入节点数组

if (flag == 2) //当前构建的节点是根节点

{

t - > LChild = PreCreateBt\_IsBST(t - > LChild, t - > value, 0); //构造左子树

t - > RChild = PreCreateBt\_IsBST(t - > RChild, t - > value, 1); //构造右子树

}

if (flag == 0) //当前构建的节点是左子节点

{

if (t - > value > parent\_value) // 当前构建的节点的值大于其父节点

{

isBST = 0; //不再是二叉搜索树

}

t - > LChild = PreCreateBt\_IsBST(t - > LChild, t - > value, 0); //构造左子树

t - > RChild = PreCreateBt\_IsBST(t - > RChild, t - > value, 1); //构造右子树

}

if (flag == 1) //当前构建的节点是右子节点

{

if (t - > value < parent\_value) // 当前构建的节点的值小于其父节点

{

isBST = 0; //不再是二叉搜索树

}

t - > LChild = PreCreateBt\_IsBST(t - > LChild, t - > value, 0); //构造左子树

t - > RChild = PreCreateBt\_IsBST(t - > RChild, t - > value, 1); //构造右子树

}

}

return t;

}

3. 判断所建立的二叉树，是否平衡

int Balanced\_height(BinaryTreeNode \* root) { //如果以root为根节点的二叉树平衡，返回以root为根的树的高度（只有一个根节点的情况算高度1）。否则返回-1

if (root == NULL) {

return 0;

}

int leftHeight = Balanced\_height(root - > LChild); //计算左子树高度

int rightHeight = Balanced\_height(root - > RChild); //计算右子树高度

//这里采用后序遍历，节约时间

if (leftHeight == -1 || rightHeight == -1 || fabs(leftHeight - rightHeight) > 1) { //leftHeight, rightHeight=-1代表 root的子树已经不平衡 ，这时就把不平衡信息传递上去

return -1; //返回-1既包含左右子树本身不平衡的情况，又包含左右子树因为高度不等导致当前节点不平衡的情况

} else {

return fmax(leftHeight, rightHeight) + 1;

}

}

bool isBalanced(BinaryTreeNode \* root) { //判断二叉树是否平衡

return Balanced\_height(root) >= 0;

}

4. 将不满足AVL树性质的二叉树，调整为AVL树

BinaryTreeNode \* sortedArrayToAVL\_helper(int \* nums, int left, int right) { //将nums数组的[left，right区间]重建成AVL

if (left > right) {

return NULL;

}

int mid = (left + right) / 2; // 总是选择中间位置左边的数字作为根节点

BinaryTreeNode \* root = (BinaryTreeNode \* ) malloc(sizeof(BinaryTreeNode));

root - > value = nums[mid];

root - > LChild = sortedArrayToAVL\_helper(nums, left, mid - 1);

root - > RChild = sortedArrayToAVL\_helper(nums, mid + 1, right);

return root;

}

BinaryTreeNode \* sortedArrayToAVL(int \* nums, int numsSize) {

return sortedArrayToAVL\_helper(nums, 0, numsSize - 1);

}

5. 可视化打印一颗二叉树的结构

void Print\_Tree(BinaryTreeNode \* t) //可视化打印二叉树结构

{

if (t == NULL) return;

queue < BinaryTreeNode \* > q\_node; //存放节点的指针

queue < int > q\_value; //存放节点的值

q\_node.push(t);

q\_value.push(t - > value); //这里先插入根节点

int depth = 1; //二叉树当前深度，根节点深度为1

int real\_depth = Depth(t); //二叉树真实深度

int cnt = 1; //插入节点的数量

while (!q\_node.empty() && depth <= real\_depth) //当前遍历了的二叉树深度小于等于实际深度

{

if (cnt > pow(2, depth) - 1) //满二叉树情况下，节点数=2^层数 -1

{

depth++; //根节点为第一层 ，不断更新深度

}

if (q\_node.front() - > LChild != NULL) //如果当前节点的左孩子非空，把左孩子入队列

{

q\_node.push(q\_node.front() - > LChild);

q\_value.push(q\_node.front() - > LChild - > value); //左孩子的值

cnt++; //遍历了的节点数+1

} else //如果当前节点的左孩子为空 ，构造一个空节点（value=0），入队列

{

BinaryTreeNode \* t = (BinaryTreeNode \* ) malloc(sizeof(BinaryTreeNode)); //构造空结点

t - > value = -1;

t - > LChild = NULL;

t - > RChild = NULL;

q\_node.push(t);

q\_value.push(t - > value);

cnt++;

}

if (q\_node.front() - > RChild != NULL) {

q\_node.push(q\_node.front() - > RChild);

q\_value.push(q\_node.front() - > RChild - > value);

cnt++;

} else //如果当前节点的右孩子为空 ，构造一个空节点（value=0），入队列

{

BinaryTreeNode \* t = (BinaryTreeNode \* ) malloc(sizeof(BinaryTreeNode)); //构造空结点

t - > value = -1;

t - > LChild = NULL;

t - > RChild = NULL;

q\_node.push(t);

q\_value.push(t - > value);

cnt++;

}

q\_node.pop(); //当前节点处理完，出队列

}

for (int i = 1; i <= real\_depth; i++) {

int layer\_node\_num = pow(2, i - 1); //第i层有几个节点

int layer\_margin\_num = pow(2, real\_depth - i) - 1; //第i层每个节点两边有几个空格

for (int j = 1; j <= layer\_node\_num; j++) //这一层有几个空格

{

for (int k = 1; k <= layer\_margin\_num; k++) {

cout << ' ';

cout << ' ';

}

cout << q\_value.front();

for (int k = 1; k <= layer\_margin\_num; k++) {

cout << ' ';

cout << ' ';

}

cout << ' ';

cout << ' ';

q\_value.pop();

}

printf("\n");

}

}

**五、测试数据及其结果分析**

输入：11 12 13 -1 -1 14 15 -1 -1 -1 16 17 -1 18 -1 -1 19 20 -1 -1 -1

理想输出：不是AVL树

实际输出：

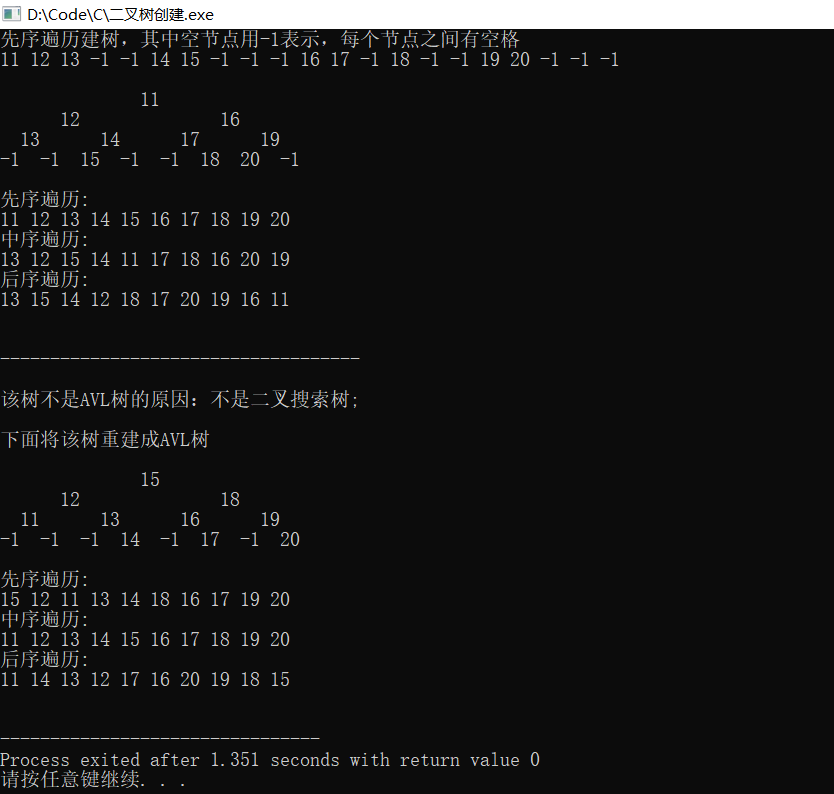


图1-3 运行结果图

输入：-1 （空树，边界情况）

理想输出：是AVL树

实际输出：

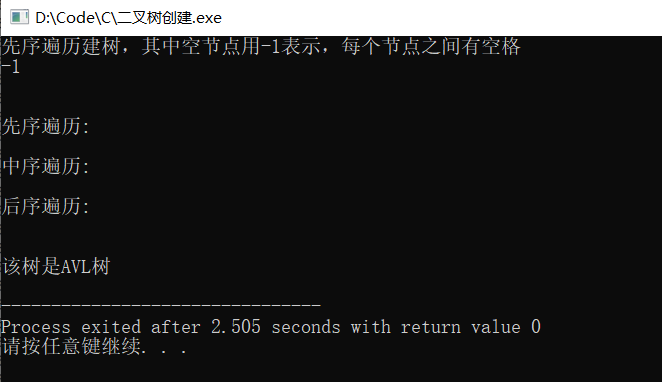


图1-4 运行结果图

输入：27 24 20 10 -1 -1 22 -1 -1 25 -1 -1 30 28 -1 -1 32 -1 -1

理想输出：是AVL树

实际输出：

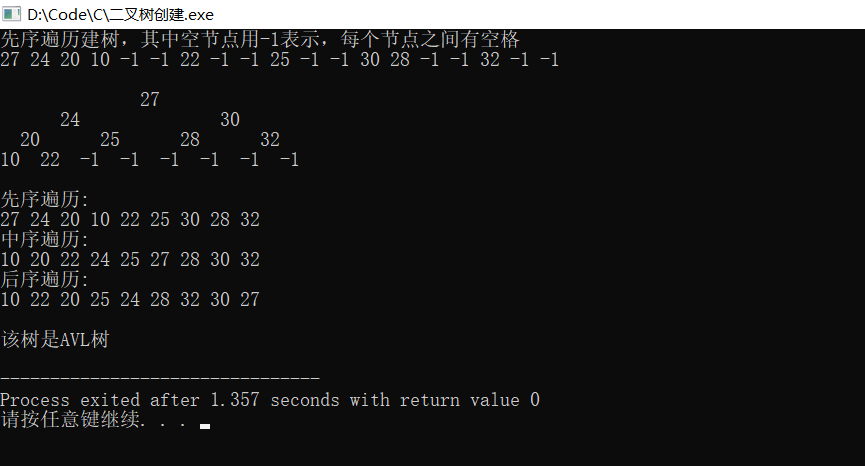


图1-5 运行结果图

输入：10 11 12 13 14 -1 -1 -1 -1 -1 -1(极端情况，退化成单链表)

理想输出：不是AVL树

实际输出：

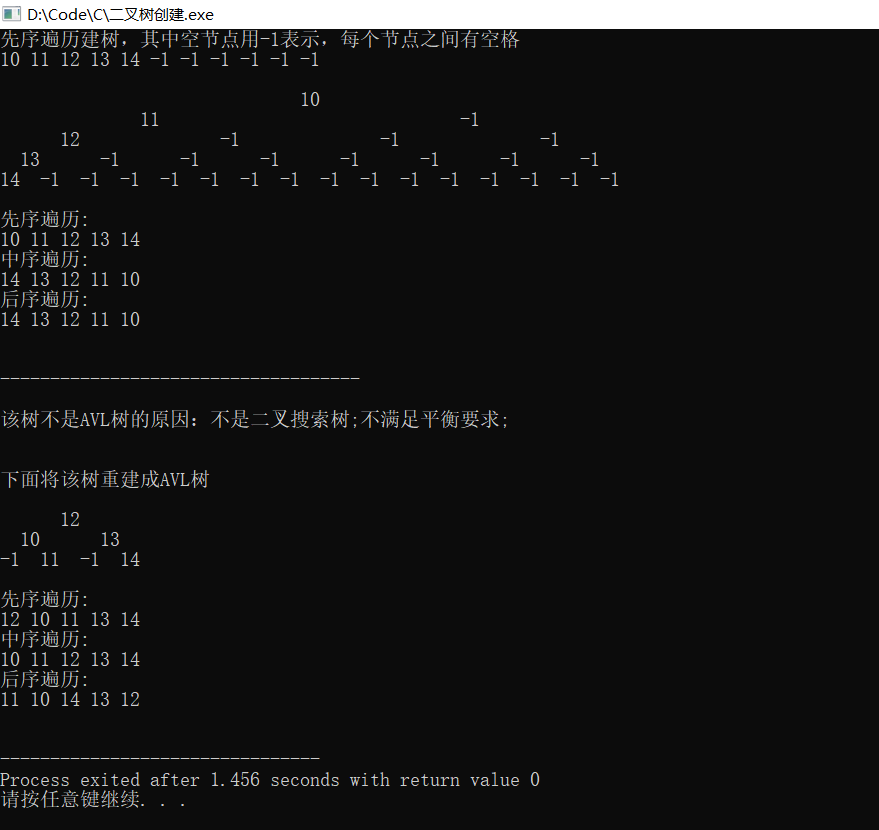


图1-6 运行结果图

**六、算法设计和程序调试过程中的问题**

**问题1**：

一开始我把判断二叉树是否满足二叉搜索树性质，另外用单独的递归遍历函数实现，感觉多做了很多功。

**解决方法**：

通过改造先序遍历建树函数，顺带实现了判断了是否为二叉搜索树的功能。详情见第三模块思路设计。

**问题2**：

在判断二叉树是否平衡函数中，一开始我采用自顶向下的遍历算法。即对于当前遍历到的节点，首先计算左右子树的高度，如果左右子树的高度差是否不超过 11，再分别递归地遍历左右子节点，并判断左子树和右子树是否平衡。这是一个自顶向下的递归的过程。但这样做对于同一个节点，函数Balanced\_height会被重复调用，导致时间复杂度较高。

**解决方法**：

改用自底向上的类后序遍历思想。对于当前遍历到的节点，先递归地判断其左右子树是否平衡，再判断以当前节点为根的子树是否平衡。如果一棵子树是平衡的，则返回其高度（高度一定是非负整数），否则返回-1。如果存在一棵子树不平衡，则整个二叉树一定不平衡。

采用自底向上思想最坏情况下需要遍历二叉树中的所有节点，时间复杂度是 O(n)。

**问题3**：

可视化打印二叉树模块中，出现爆内存死循环异常

**解决方法**：

通过一步步调试，确定出错位置if(cnt>2^depth -1)。原来不同于MATLAB或PYTHON，C++中^运算符代表异或位运算。我想实现的幂次方运算应该改为if(cnt>pow(2,depth)-1)。

**七、课程设计总结**

1.算法进一步改进的设想

不足之处1：仅仅是重构了AVL树，关于重构后AVL树的增删改查还有待添加。

不足之处2：可视化打印二叉树，树中节点的值必须有相同的位数，不然打印出的二叉树会变形，目前还没想出好的解决办法。

2.课程设计过程的收获和感受

我的感受：

通过这次试验，我顺带复习了上学期数据结构的基础知识，同时又学习了很多新的知识。AVL树的判定问题可以简化为二叉搜索树的判定和二叉树平衡性的判定。这两个功能实现起来都不难，重要的是如何尽可能地简化程序，复用代码，提高效率。要解决一个大问题，往往是把大问题分解成小问题，再思考怎么把这些小模块分割和组合是最优的。往往开始的第一个想法不是最优的，但我通过思考和查阅资料，尽可能地完善和改进程序。除了基础部分的AVL树判断，我还实现了AVL树的重构和可视化打印二叉树形状等功能。AVL树的重构甚是巧妙，它的两个子问题BST重构和平衡二叉树重构可以通过节点排序+二分构造一石二鸟的完成。可视化打印二叉树有些复杂，虽然有些小瑕疵，但我通过改编学过的算法和数据结构基本完成了。纸上得来终觉浅，绝知此事要躬行，我们学习算法设计和数据结构，不能仅停留在理论上懂，而要多编程，多写代码，只有代码才是检验真理的唯一标准。我们更应该多思考，不局限于题目要求的功能，而应该发散思维，给自己更多的挑战。

我的收获：

写代码过程中不可避免的就是出错，这再一次体现了把一个大问题模块化的好处，遇到问题时可以分别测试几个模块，快速定位出错位置。

不要一上来就开始写代码必须要交流清楚，否则，后续将会有更多问题。

必须写注释，如果不写注释，时间久了，回过头来连你自己都看不懂。

不要心存侥幸：如果某个地方你感觉会出bug，那么，一定就是bug。千万不要心存侥幸，一定要把自己感觉会出bug的地方优化好，不留后患。

**火车调度**

**一、课题内容和要求**

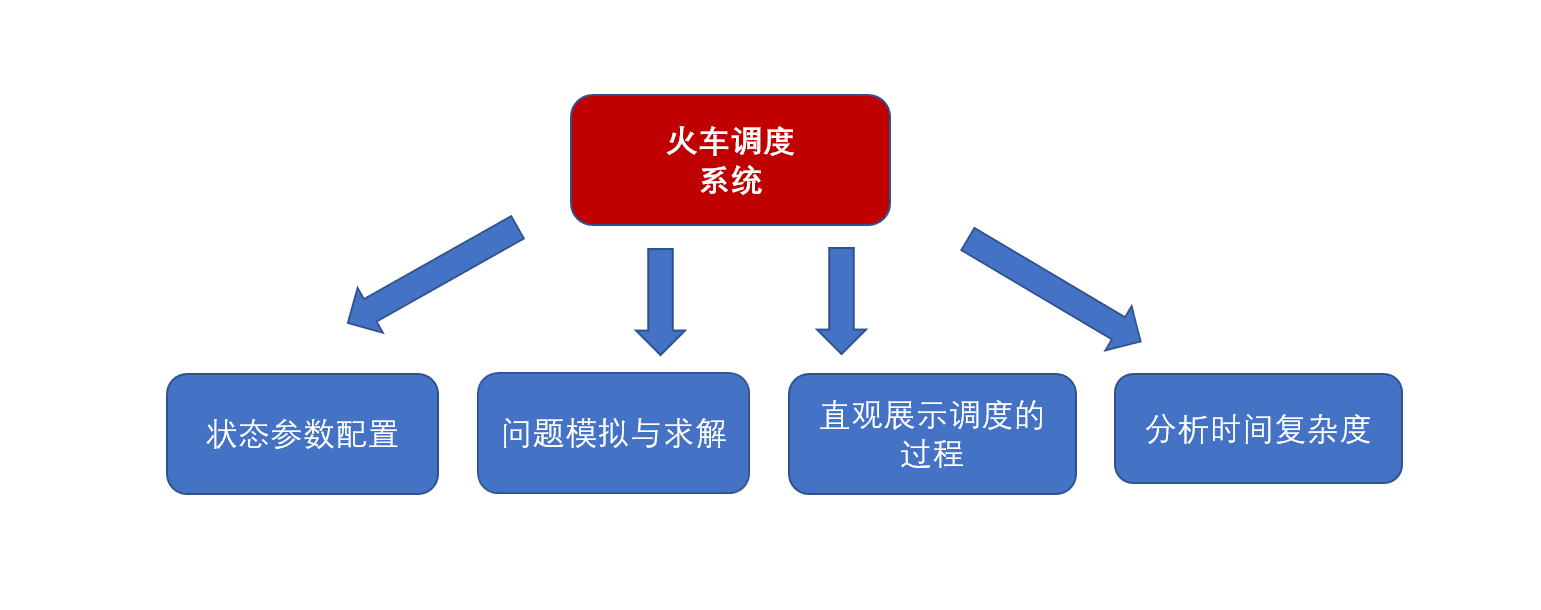


图2-1 功能框架图

（1）支持系统运行时的状态参数配置，如车厢数量和顺序，缓冲区数量等。

（2）利用堆栈，队列等数据结构进行问题模拟与求解。

（3）支持将求解过程和每一步各区域的变化直观展示。

（4）计算时间复杂度。

**二、数据结构说明**

程序里主要使用了C++ STL库中的数据结构：

1.程序利用train数组来储存车厢顺序。

2.程序利用queue<int>enter储存入轨区的情况。

3.程序利用vector< stack<int> > temp储存缓冲区的情况。

4.程序利用queue<int>ready 储存了出轨区的情况。

**三、算法设计**

1. 状态参数配置模块

可以自定义车厢数量，车厢顺序，缓冲区数量，拓展性强。

2．利用堆栈，队列等数据结构进行问题模拟与求解

**实现函数：**

int solution(int \*train,int num,int k)

其中train为车厢顺序数组的首地址，num为车厢数，k为缓冲区个数。当调度成功时，函数返回1，否则返回0。

**设计思路：**

从前至后依次检查入轨上的所有车厢：如果正在检查的车厢就是下一个满足排列要求的车厢，可以直接把它放到出轨上去；如果不是，则把它移动到缓冲铁轨上，直到按输出次序要求轮到它时才将它放到出轨上；缓冲区里车厢的进和出只能在缓冲区的首部进行（相当于堆栈）。在缓冲区这我设置了几个局部变量：min\_id（缓冲区里的最小id，初始化为比最大编号更大的数），min\_k\_id（min\_id车厢对应的缓冲区号），Best\_id （缓冲区中最优插入位置），Best\_k\_id（最优插入位置在哪个缓冲区）。车厢进出缓冲区后都要更新对应标志的值。

总之，在重排车厢过程中，仅允许以下移动:

（1）车厢可以从入轨区移动到一个缓冲区的顶部或直接进入出轨区接在重排的列车后。

（2）车厢可以从一个缓冲铁轨的顶部移动到出轨接在重排的列车后。

新的车厢x 应送入这样的缓冲区：其首部的车厢编号 a 满足 a>x，且 a 是所有满足这种条件的缓冲区顶部车厢编号中最小的一个编号。优先进入有车厢的缓冲区，找不到符合条件的有车厢的缓冲区，才进入空缓冲区。有只有这样才能使后续的车厢重排所受到的限制最小。直观地来说，缓冲区里从顶部到底部车厢编号依次递增，且同一个缓冲区里编号的差值尽可能小，只有这样，才能充分利用空间。

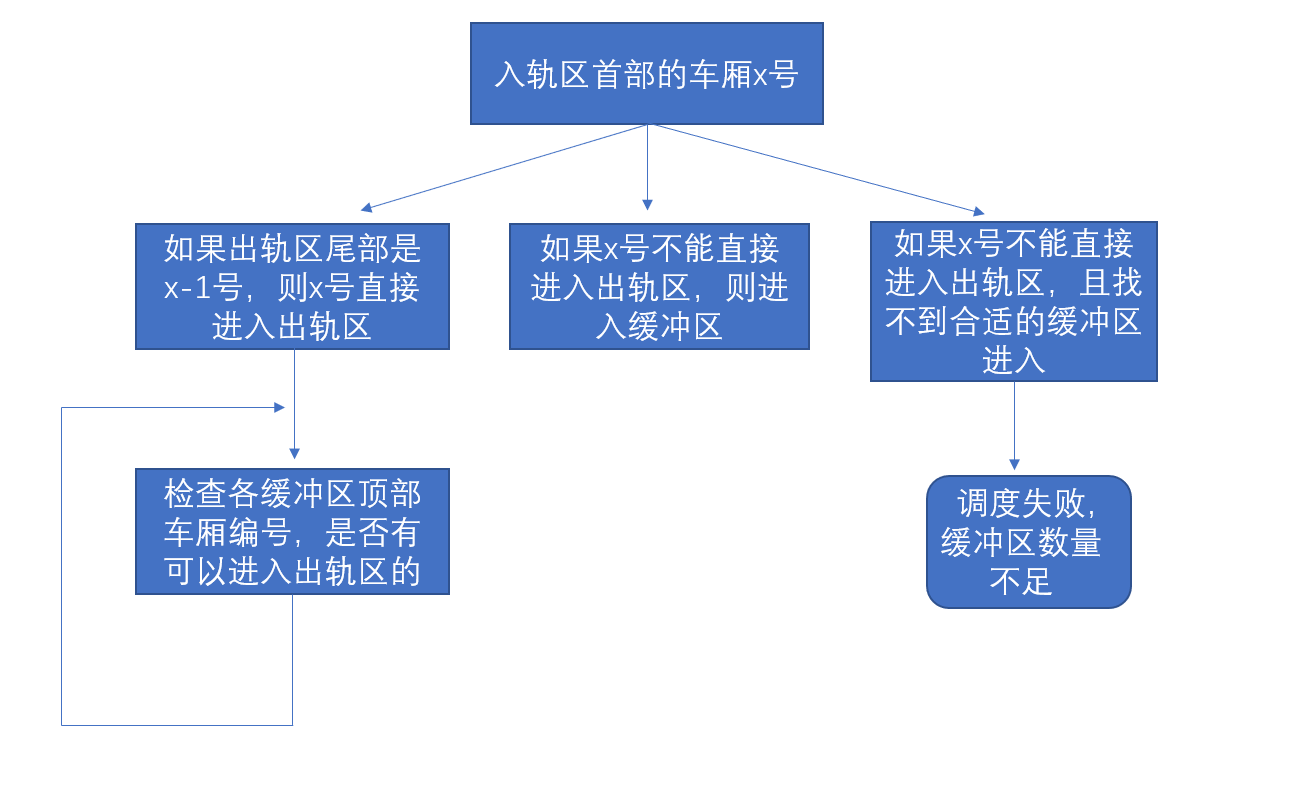


图2-2 算法流程图

3.直观展示求解过程

**实现函数：**

void print\_situation(queue<int> enter,vector< stack<int> > temp, queue<int> ready, int k)

其中enter为入轨区队列，temp为各缓冲区栈组成的向量，ready为出轨区队列，k为缓冲区数量。

**设计思路：**

通过传值方式将入轨区，缓冲区，出轨区传入，并遍历打印。因为C++ STL的特性是如果想遍历一个堆栈或队列，必须依次从堆栈或队列中把元素一个个pop()出来，如果使用传指针或传引用，则会破坏对应变量的结构。

4. 时间复杂度分析

求解函数solution中，有几个子操作：Ⅰ.把一个车厢从入轨区送入某个缓冲区；Ⅱ.把一个车厢从缓冲区送至出轨区 ；Ⅲ.把一个车厢从入轨区直接送至出轨区。下面分别讨论这三个子操作的时间复杂度。

Ⅰ.把一个车厢从入轨区送入某个缓冲区

首先要遍历k个缓冲区的顶部，找到最适合插入的位置。O（k）

移动车厢（出入轨区队列，入缓冲区栈）。O（1）

判断缓冲区加入新车厢后，是否要更新缓存区里存放的最小车厢编号。O（1）

子操作Ⅰ的综合时间复杂度为O（k）

Ⅱ.把一个车厢从缓冲区送至出轨区

首先要遍历k个缓冲区的顶部，有没有可以正好出栈接到出轨区的。O（k）

移动车厢（出缓冲区栈，入出轨区队列）。O（1）

子操作Ⅱ的综合时间复杂度为O（k）

Ⅲ.把一个车厢从入轨区直接送至出轨区

直接移动车厢（出入轨区队列，入出轨区队列）。O（1）

子操作Ⅲ的综合时间复杂度为O（1）

综合考虑整个solution过程，设一共有n个车厢。每一个车厢到出轨区，要么经历子过程Ⅰ和Ⅱ，要么经历子过程Ⅲ。所以最好情况下时间复杂度为O(n),最坏情况下时间复杂度为O(n\*k)。当然如果我把子过程Ⅰ和Ⅱ中改用AVL树存储缓冲区顶部的车厢编号，则最坏时间复杂度可降为O(n\*logk),但鉴于k的大小在实际情况中不会很大，O(n\*k)和O(n\*logk)的差距不大。

**四、详细设计**

下面详细展示一些关键模块的代码和注释

1.求解函数solution

int solution(int \*train,int num,int k)// num为车箱数， k为缓冲区个数

{ vector< stack<int> > temp;//缓冲区

for(int j=0;j<k;j++)

{ stack<int> s;

temp.push\_back(s);

}

queue<int>enter;//入轨区

queue<int>ready;//出轨区

for(int i=0;i<num;i++)

{

enter.push(train[i]);//初始化入轨区

}

int correct\_id=1;//按顺序下应该一个出轨车厢的id

int min\_id=100000;//待在缓冲区里的最小id，初始化为比最大编号更大的数

int min\_k\_id=-1;///min\_id车厢对应的缓冲区号（下标是从0开始的） ，这里初始化为一个不可能的数-1

print\_situation(enter,temp,ready,k);

for(int i=0;i<num;i++)

{

int id=train[i];

if(id==correct\_id)//如果当前车厢正好可以接上出轨区

{

enter.pop(); //驶离入轨区

ready.push(correct\_id);//更新出轨区

cout << "操作：将车厢 " << id << " 从入轨区 " "移至出轨区 " << endl;

print\_situation(enter,temp,ready,k);

correct\_id=id+1;//只要有车厢进入出轨取，correct\_id就要增加

while(min\_id==correct\_id)//如果缓冲区里最小的车厢id可以接上出轨区

{

//出栈，更新

temp[min\_k\_id].pop();//出栈(移入出轨区)

ready.push(correct\_id);//更新出轨区

cout << "操作：将车厢 " << min\_id << " 从缓冲区 " << min\_k\_id << " 移至出轨区" << endl;

print\_situation(enter,temp,ready,k);

min\_id=100000; //先假设 min\_id为比最大编号更大的数

for (int j = 0; j <k; j++)//遍历所有的缓冲区，更新出栈后的缓冲区

{

if (!temp[j].empty() ) //如果遍历到的不是空栈

{

if(temp[j].top() < min\_id)////如果栈顶元素比当前的min\_id小

{

min\_id = temp[j].top();

min\_k\_id = j;

}

}

}

correct\_id++; //只要有车厢进入出轨取，correct\_id就要增加

}

}

else//如果当前车厢不能正好接上出轨区，则要将其入栈

{ int Best\_id = 100000; // 缓冲区中最优插入位置是Best\_id车厢的上面(最小的大于id车厢).初始值设为比num还大

int Best\_k\_id = -1; //最优插入位置 （哪个缓冲区）

for(int j=0;j<k;j++)

{

if (!temp[j].empty() ) //如果遍历到的不是空栈

{

if (id < temp[j].top() && temp[j].top() < Best\_id) //要找出缓冲区中刚大于id的那个车厢和位置，然后把新车厢放在上面

{

Best\_id=temp[j].top();

Best\_k\_id=j;

}

}

else //遍历到空栈

{

if (Best\_k\_id==-1)//还没找到可以插入的位置

{

Best\_k\_id=j;

}

}

}

if(Best\_id==100000&&Best\_k\_id==-1)

{

cout<<"缓冲区数量不够"<<endl;

return 0;

}

else//缓冲区数量够

{

enter.pop(); //驶离入轨区

temp[Best\_k\_id].push(id);//把id车厢插入缓冲区最优位置

cout << "操作：将车厢 " << id << " 从入轨区 " "移至缓冲区 " << Best\_k\_id << endl;

print\_situation(enter,temp,ready,k);

if(id<min\_id)//入栈后更新缓冲区的最小车厢id和对应的缓冲区号

{

min\_id=id;

min\_k\_id=Best\_k\_id;

}

}

}

}

return 1;

}

2.打印各区域情况的函数print\_situation

void print\_situation(queue<int> enter,vector< stack<int> > temp, queue<int> ready,int k)//这里采用传值方式 ,打印各个区域的情况

{

cout<<endl;

cout<<"入轨区（左边为队列首）：";

while(!enter.empty())

{

cout<<enter.front();

enter.pop();

}

for(int j=0;j<k;j++)

{ stack<int> s=temp[j];

cout<<endl<<j+1<<"号缓冲区（左边为缓冲区顶部）：";

while(!s.empty())

{

cout<<s.top();

s.pop();

}

}

cout<<endl;

cout<<"出轨区（左边为队列首）：";

while(!ready.empty())

{

cout<<ready.front();

ready.pop();

}

cout<<endl<<"-----------------------------------------"<<endl<<endl;

}

**五、测试数据及其结果分析**

输入：k=0

理想输出：缓冲区不够，失败

实际输出：

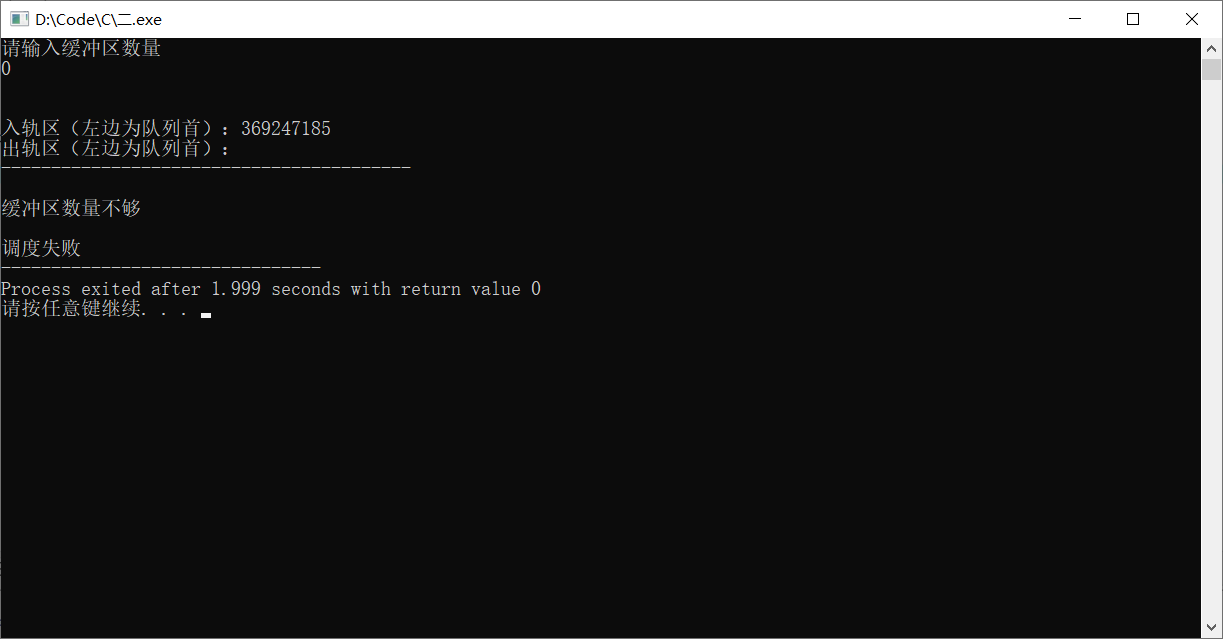


图2-3 运行结果图

输入：k=2

理想输出：缓冲区不够，失败

实际输出：

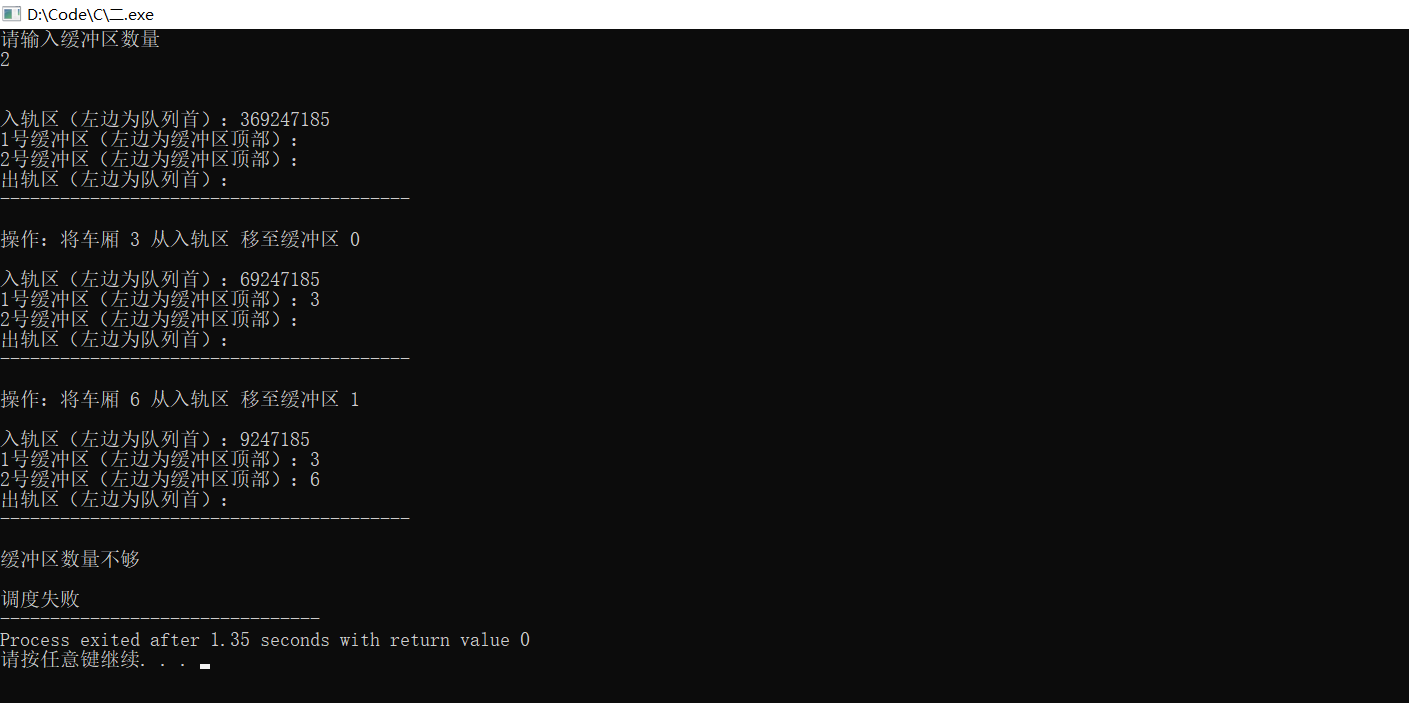


图2-4 运行结果图

输入：k=3

理想输出：调度成功

实际输出：（图片太长，这里只截取部分运行结果）

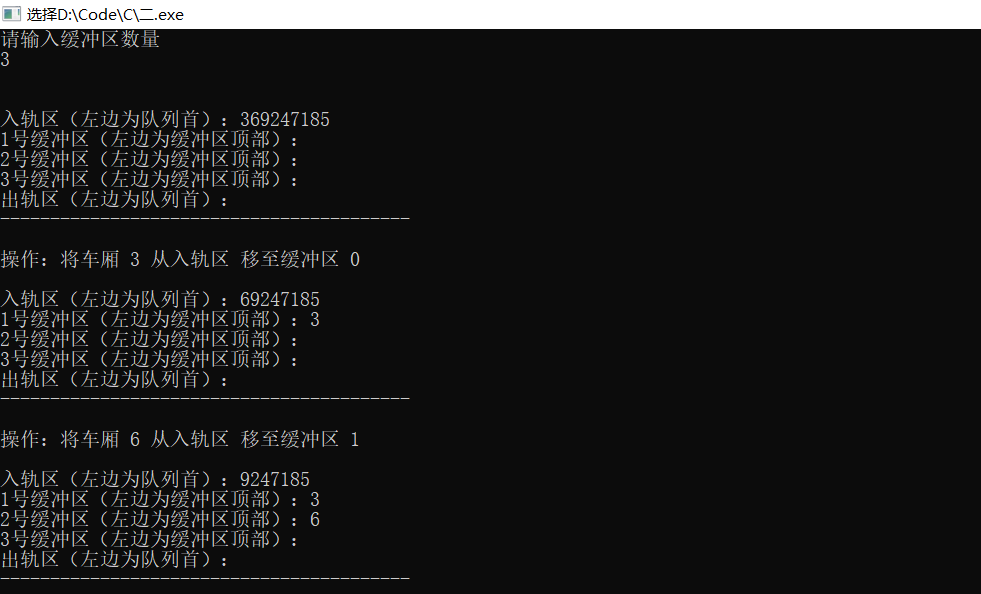


图2-5 运行结果图

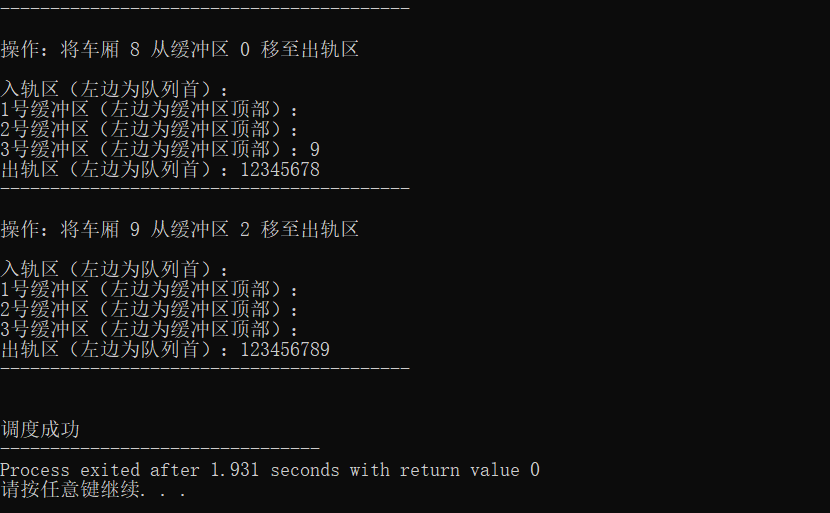


图2-6 运行结果图

**六、算法设计和程序调试过程中的问题**

**问题1**：print\_situation中，找不到能遍历队列或堆栈，同时又不破坏它们结构的方法。

**解决方法**：函数由传引用改为传值。

**问题2**：设定的几个标志变量 min\_id（缓冲区里的最小id，初始化为比最大编号更大的数），min\_k\_id（min\_id车厢对应的缓冲区号），Best\_id （缓冲区中最优插入位置），Best\_k\_id（最优插入位置在哪个缓冲区）没有设定好初始值，导致程序不能按照预期效果执行。

**解决方法**：将min\_id初始化一个比最大编号更大的数，我程序中用了100000。将min\_k\_id初始化一个不可能的缓冲区编号，我程序中用了-1。将Best\_id初始化一个比最大编号更大的数，我程序中用了100000。将Best\_k\_id初始化一个不可能的缓冲区编号，我程序中用了-1。

**问题3**：我想达到程序分步骤显示的效果

**解决方法**：通过getchar()函数阻塞进程，等待从键盘输入才继续进行。

**七、课程设计总结**

1.算法进一步改进的设想

不足之处1：如果把solution函数子过程Ⅰ和Ⅱ中改用AVL树存储缓冲区顶部的车厢编号，则最坏时间复杂度可降为O(n\*logk)。但由于时间紧迫和实际意义不大，我没有实现。

不足之处2：没有做到动态图形可视化界面。理论上用QT或MFC可以绘制GUI窗口，但由于正值期中考试周，来不及实现，我只做了黑框程序。

2.课程设计过程的收获和感受

我的感受：

通过这次试验，我深刻理解了队列和堆栈这种数据结构在实际算法设计中的应用。火车调度问题可以简化为几个子操作。这些子操作实现起来都不难，重点是如何设计数据结构储存各区域的情况并给予各变量合适的初始值。要解决一个大问题，往往是把大问题分解成小问题，再思考怎么把这些小模块分割和组合是最优的。一开始我想在每次移动操作后直接打印各队列和堆栈的情况，但通过网上查阅资料发现要想遍历队列和堆栈，必须使用pop()操作，但这势必会破坏现有数据的结构。所以我选择将打印队列和堆栈这一操作抽象出来，另成一个函数，采用传值的方式将队列和堆栈传入函数并遍历打印。纸上得来终觉浅，绝知此事要躬行，有些时候思路在脑子里想觉得很简单，但在实际上手写时才发现有很多边界限定，稍微思考地不全面整个程序就会出错。编写的程序想要一次通过测试几乎是不可能的，要通过不断调试与验证才能检查出错误，并定位出错地点。在编程过程中遇到问题是很常见的，我们要有独立解决问题的能力，比如看专业书籍或上网查找，不能总想着依赖老师或其他同学。

我的收获：

要学会给变量命名。这道题所涉及的存储变量，标志变量有很多，很复杂。好的名字可以见名知意，非常容易理解。命名的过程同时也是概念提取的过程。变量名宁可长一些说明清楚用途也不要用a、b、c之类的无意义的名称，除非是循环计数器中用i、j、k等。

尽可能自己解决问题。写这道题时因为很久没有使用C++的STL库，我遇到了一些语法问题。像这种比较细比较专的问题问他人是不合适的，尽量通过查阅书籍和相关文档解决。