实习一

1. **求解约瑟夫环**

题目描述:

已知n个人(以编号1，2，3...n分别表示)围坐在一张圆桌周围。从编号为k的人开始报数，数到m的那个人出列;他的下一个人又从1开始报数，数到m的那个人又出列;依此规律重复下去，直到圆桌周围的人全部出列。

1 .用数组实现

思路:

假设有n个人，从1到n，放入一个ArrayList中，设置开始的编号k(设为0，为第一个人），加上需要报的数减1，即得到需要出列人的索引，要注意要出列的人是不是最后一个人，要特殊处理。当ArrayList为空，则完成。

时间复杂度O(n),空间复杂度O（n）

源代码:

import java.util.ArrayList;

public class Main3 {

public static void yesefu(int totalNum,int countNum) {

//初始化人数

ArrayList<Integer> start = new ArrayList<Integer>();

for(int i=1;i<=totalNum;i++) {

start.add(i);

}

//从第k个数开始计数

int k = 0;//从第一个人开始计数

while(start.size() > 0) {

k = k + countNum;

//第m人的索引位置

k = k % (start.size()) - 1;//因为索引是从0开始，进行取余是因为防止k大于圈中人的总数

//判断是否到队尾，即k的大小为start的大小，此时k为-1

if(k < 0) {

//如果到队尾了，则输出队尾元素

System.out.print(start.get(start.size()-1)+" ");

start.remove(start.size() - 1);

k = 0;

}else {

//否则，直接输出k对应的元素

System.out.print(start.get(k)+" ");

start.remove(k);

}

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

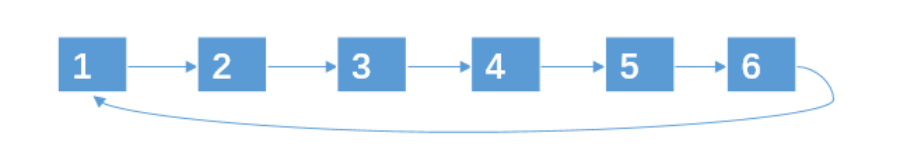
yesefu(10,3);

}

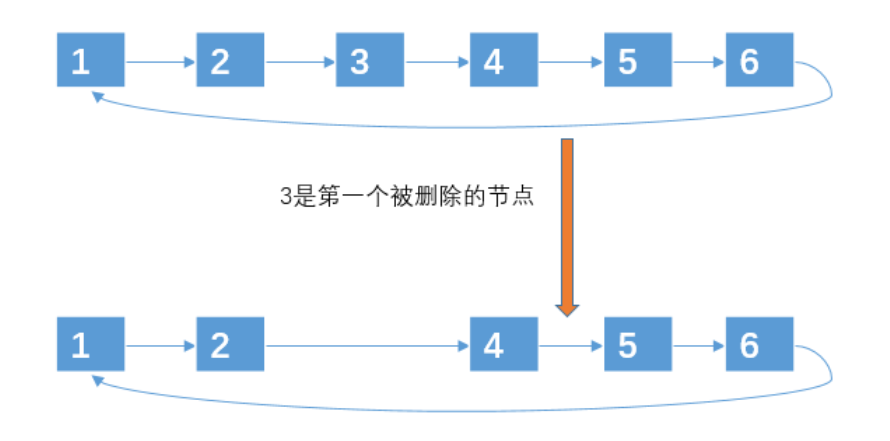
}

2. 用环形链表解决

* 1. 先创建一个环形链表来存放元素



* 1. 然后在遍历链表的同时做删除操作



这种做法的时间复杂度是 O(nm), 空间复杂度是 O(n)

源代码:

// 定义链表节点

class Node{

int date;

Node next;

public Node(int date) {

this.date = date;

}

}

public static int solve(int n, int m) {

if(m == 1 || n < 2)

return n;

// 创建环形链表

Node head = createLinkedList(n);

// 遍历删除

int count = 1;

Node cur = head;

Node pre = null;//前驱节点

while (head.next != head) {

// 删除节点

if (count == m) {

count = 1;

pre.next = cur.next;

cur = pre.next;

} else {

count++;

pre = cur;

cur = cur.next;

}

}

return head.date;

}

static Node createLinkedList(int n) {

Node head = new Node(1);

Node next = head;

for (int i = 2; i <= n; i++) {

Node tmp = new Node(i);

next.next = tmp;

next = next.next;

}

// 头尾串联

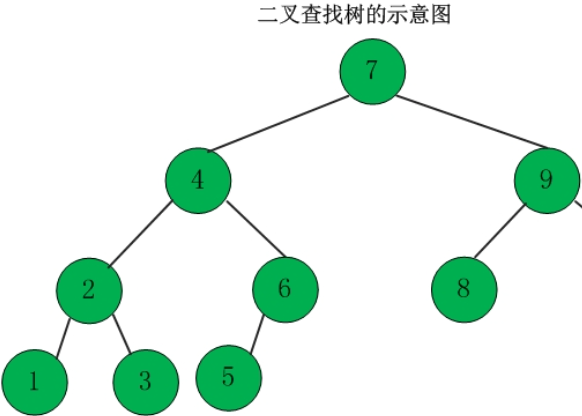
next.next = head;

return head;

}

**二. 二分搜索树**

二叉搜索树简介  
它是特殊的二叉树：对于二叉树，假设x为二叉树中的任意一个结点，x节点包含关键字key，节点x的key值记为key[x]。如果y是x的左子树中的一个结点，则key[y] <= key[x]；如果y是x的右子树的一个结点，则key[y] >= key[x]。那么，这棵树就是二叉查找树。如下图所示：



在二叉查找树中：  
(01) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；  
(02) 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；  
(03) 任意节点的左、右子树也分别为二叉查找树。  
(04) 没有键值相等的节点

1. **二叉查找树节点的定义**

public class BSTree<T extends Comparable<T>> {

private BSTNode<T> mRoot; // 根结点

public class BSTNode<T extends Comparable<T>> {

T key; // 关键字(键值)

BSTNode<T> left; // 左孩子

BSTNode<T> right; // 右孩子

BSTNode<T> parent; // 父结点

public BSTNode(T key, BSTNode<T> parent, BSTNode<T> left, BSTNode<T> right) {

this.key = key;

this.parent = parent;

this.left = left;

this.right = right;

}

}

......

}

BSTree是二叉树，它保护了二叉树的根节点mRoot；mRoot是BSTNode类型，而BSTNode是二叉查找树的节点，它是BSTree的内部类。BSTNode包含二叉查找树的几个基本信息：  
(01) key -- 它是关键字，是用来对二叉查找树的节点进行排序的。  
(02) left -- 它指向当前节点的左孩子。  
(03) right -- 它指向当前节点的右孩子。  
(04) parent -- 它指向当前节点的父结点。

**2 遍历**

这里讲解前序遍历、中序遍历、后序遍历3种方式。

2.1 前序遍历  
若二叉树非空，则执行以下操作：  
(01) 访问根结点；  
(02) 先序遍历左子树；  
(03) 先序遍历右子树。

前序遍历代码

private void preOrder(BSTNode<T> tree) {

if(tree != null) {

System.out.print(tree.key+" ");

preOrder(tree.left);

preOrder(tree.right);

}

}

public void preOrder() {

preOrder(mRoot);

}

2.2 中序遍历

若二叉树非空，则执行以下操作：  
(01) 中序遍历左子树；  
(02) 访问根结点；  
(03) 中序遍历右子树。

中序遍历代码

private void inOrder(BSTNode<T> tree) {

if(tree != null) {

inOrder(tree.left);

System.out.print(tree.key+" ");

inOrder(tree.right);

}

}

public void inOrder() {

inOrder(mRoot);

}

2.3 后序遍历

若二叉树非空，则执行以下操作：  
(01) 后序遍历左子树；  
(02) 后序遍历右子树；  
(03) 访问根结点。

后序遍历代码

private void postOrder(BSTNode<T> tree) {

if(tree != null)

{

postOrder(tree.left);

postOrder(tree.right);

System.out.print(tree.key+" ");

}

}

public void postOrder() {

postOrder(mRoot);

}

**3. 查找**

递归版本的代码

/\*

\* (递归实现)查找"二叉树x"中键值为key的节点

\*/

private BSTNode<T> search(BSTNode<T> x, T key) {

if (x==null)

return x;

int cmp = key.compareTo(x.key);

if (cmp < 0)

return search(x.left, key);

else if (cmp > 0)

return search(x.right, key);

else

return x;

}

public BSTNode<T> search(T key) {

return search(mRoot, key);

}

非递归版本的代码

/\*

\* (非递归实现)查找"二叉树x"中键值为key的节点

\*/

private BSTNode<T> iterativeSearch(BSTNode<T> x, T key) {

while (x!=null) {

int cmp = key.compareTo(x.key);

if (cmp < 0)

x = x.left;

else if (cmp > 0)

x = x.right;

else

return x;

}

return x;

}

public BSTNode<T> iterativeSearch(T key) {

return iterativeSearch(mRoot, key);

}  
**4. 最大值和最小值**

查找最大值的代码

/\*

\* 查找最大结点：返回tree为根结点的二叉树的最大结点。

\*/

private BSTNode<T> maximum(BSTNode<T> tree) {

if (tree == null)

return null;

while(tree.right != null)

tree = tree.right;

return tree;

}

public T maximum() {

BSTNode<T> p = maximum(mRoot);

if (p != null)

return p.key;

return null;

}

查找最小值的代码

/\*

\* 查找最小结点：返回tree为根结点的二叉树的最小结点。

\*/

private BSTNode<T> minimum(BSTNode<T> tree) {

if (tree == null)

return null;

while(tree.left != null)

tree = tree.left;

return tree;

}

public T minimum() {

BSTNode<T> p = minimum(mRoot);

if (p != null)

return p.key;

return null;

}

**5. 前驱和后继**

节点的前驱：是该节点的左子树中的最大节点。  
节点的后继：是该节点的右子树中的最小节点。

查找前驱节点的代码

/\*

\* 找结点(x)的前驱结点。即，查找"二叉树中数据值小于该结点"的"最大结点"。

\*/

public BSTNode<T> predecessor(BSTNode<T> x) {

// 如果x存在左孩子，则"x的前驱结点"为 "以其左孩子为根的子树的最大结点"。

if (x.left != null)

return maximum(x.left);

// 如果x没有左孩子。则x有以下两种可能：

// (01) x是"一个右孩子"，则"x的前驱结点"为 "它的父结点"。

// (01) x是"一个左孩子"，则查找"x的最低的父结点，并且该父结点要具有右孩子"，找到的这个"最低的父结点"就是"x的前驱结点"。

BSTNode<T> y = x.parent;

while ((y!=null) && (x==y.left)) {

x = y;

y = y.parent;

}

return y;

}

查找后继节点的代码

/\*

\* 找结点(x)的后继结点。即，查找"二叉树中数据值大于该结点"的"最小结点"。

\*/

public BSTNode<T> successor(BSTNode<T> x) {

// 如果x存在右孩子，则"x的后继结点"为 "以其右孩子为根的子树的最小结点"。

if (x.right != null)

return minimum(x.right);

// 如果x没有右孩子。则x有以下两种可能：

// (01) x是"一个左孩子"，则"x的后继结点"为 "它的父结点"。

// (02) x是"一个右孩子"，则查找"x的最低的父结点，并且该父结点要具有左孩子"，找到的这个"最低的父结点"就是"x的后继结点"。

BSTNode<T> y = x.parent;

while ((y!=null) && (x==y.right)) {

x = y;

y = y.parent;

}

return y;

}

**6. 插入**

插入节点的代码

/\*

\* 将结点插入到二叉树中

\*

\* 参数说明：

\* tree 二叉树的

\* z 插入的结点

\*/

private void insert(BSTree<T> bst, BSTNode<T> z) {

int cmp;

BSTNode<T> y = null;

BSTNode<T> x = bst.mRoot;

// 查找z的插入位置

while (x != null) {

y = x;

cmp = z.key.compareTo(x.key);

if (cmp < 0)

x = x.left;

else

x = x.right;

}

z.parent = y;

if (y==null)

bst.mRoot = z;

else {

cmp = z.key.compareTo(y.key);

if (cmp < 0)

y.left = z;

else

y.right = z;

}

}

/\*

\* 新建结点(key)，并将其插入到二叉树中

\*

\* 参数说明：

\* tree 二叉树的根结点

\* key 插入结点的键值

\*/

public void insert(T key) {

BSTNode<T> z=new BSTNode<T>(key,null,null,null);

// 如果新建结点失败，则返回。

if (z != null)

insert(this, z);

}

**合并通用代码:**

public class BSTree<T extends Comparable<T>> {

private BSTNode<T> mRoot; // 根结点

public class BSTNode<T extends Comparable<T>> {

T key; // 关键字(键值)

BSTNode<T> left; // 左孩子

BSTNode<T> right; // 右孩子

BSTNode<T> parent; // 父结点

public BSTNode(T key, BSTNode<T> parent, BSTNode<T> left, BSTNode<T> right) {

this.key = key;

this.parent = parent;

this.left = left;

this.right = right;

}

public T getKey() {

return key;

}

public String toString() {

return "key:"+key;

}

}

public BSTree() {

mRoot=null;

}

/\*

\* 前序遍历"二叉树"

\*/

private void preOrder(BSTNode<T> tree) {

if(tree != null) {

System.out.print(tree.key+" ");

preOrder(tree.left);

preOrder(tree.right);

}

}

public void preOrder() {

preOrder(mRoot);

}

/\*

\* 中序遍历"二叉树"

\*/

private void inOrder(BSTNode<T> tree) {

if(tree != null) {

inOrder(tree.left);

System.out.print(tree.key+" ");

inOrder(tree.right);

}

}

public void inOrder() {

inOrder(mRoot);

}

/\*

\* 后序遍历"二叉树"

\*/

private void postOrder(BSTNode<T> tree) {

if(tree != null)

{

postOrder(tree.left);

postOrder(tree.right);

System.out.print(tree.key+" ");

}

}

public void postOrder() {

postOrder(mRoot);

}

/\*

\* (递归实现)查找"二叉树x"中键值为key的节点

\*/

private BSTNode<T> search(BSTNode<T> x, T key) {

if (x==null)

return x;

int cmp = key.compareTo(x.key);

if (cmp < 0)

return search(x.left, key);

else if (cmp > 0)

return search(x.right, key);

else

return x;

}

public BSTNode<T> search(T key) {

return search(mRoot, key);

}

/\*

\* (非递归实现)查找"二叉树x"中键值为key的节点

\*/

private BSTNode<T> iterativeSearch(BSTNode<T> x, T key) {

while (x!=null) {

int cmp = key.compareTo(x.key);

if (cmp < 0)

x = x.left;

else if (cmp > 0)

x = x.right;

else

return x;

}

return x;

}

public BSTNode<T> iterativeSearch(T key) {

return iterativeSearch(mRoot, key);

}

/\*

\* 查找最小结点：返回tree为根结点的二叉树的最小结点。

\*/

private BSTNode<T> minimum(BSTNode<T> tree) {

if (tree == null)

return null;

while(tree.left != null)

tree = tree.left;

return tree;

}

public T minimum() {

BSTNode<T> p = minimum(mRoot);

if (p != null)

return p.key;

return null;

}

/\*

\* 查找最大结点：返回tree为根结点的二叉树的最大结点。

\*/

private BSTNode<T> maximum(BSTNode<T> tree) {

if (tree == null)

return null;

while(tree.right != null)

tree = tree.right;

return tree;

}

public T maximum() {

BSTNode<T> p = maximum(mRoot);

if (p != null)

return p.key;

return null;

}

/\*

\* 找结点(x)的后继结点。即，查找"二叉树中数据值大于该结点"的"最小结点"。

\*/

public BSTNode<T> successor(BSTNode<T> x) {

// 如果x存在右孩子，则"x的后继结点"为 "以其右孩子为根的子树的最小结点"。

if (x.right != null)

return minimum(x.right);

// 如果x没有右孩子。则x有以下两种可能：

// (01) x是"一个左孩子"，则"x的后继结点"为 "它的父结点"。

// (02) x是"一个右孩子"，则查找"x的最低的父结点，并且该父结点要具有左孩子"，找到的这个"最低的父结点"就是"x的后继结点"。

BSTNode<T> y = x.parent;

while ((y!=null) && (x==y.right)) {

x = y;

y = y.parent;

}

return y;

}

/\*

\* 找结点(x)的前驱结点。即，查找"二叉树中数据值小于该结点"的"最大结点"。

\*/

public BSTNode<T> predecessor(BSTNode<T> x) {

// 如果x存在左孩子，则"x的前驱结点"为 "以其左孩子为根的子树的最大结点"。

if (x.left != null)

return maximum(x.left);

// 如果x没有左孩子。则x有以下两种可能：

// (01) x是"一个右孩子"，则"x的前驱结点"为 "它的父结点"。

// (01) x是"一个左孩子"，则查找"x的最低的父结点，并且该父结点要具有右孩子"，找到的这个"最低的父结点"就是"x的前驱结点"。

BSTNode<T> y = x.parent;

while ((y!=null) && (x==y.left)) {

x = y;

y = y.parent;

}

return y;

}

/\*

\* 将结点插入到二叉树中

\*

\* 参数说明：

\* tree 二叉树的

\* z 插入的结点

\*/

private void insert(BSTree<T> bst, BSTNode<T> z) {

int cmp;

BSTNode<T> y = null;

BSTNode<T> x = bst.mRoot;

// 查找z的插入位置

while (x != null) {

y = x;

cmp = z.key.compareTo(x.key);

if (cmp < 0)

x = x.left;

else

x = x.right;

}

z.parent = y;

if (y==null)

bst.mRoot = z;

else {

cmp = z.key.compareTo(y.key);

if (cmp < 0)

y.left = z;

else

y.right = z;

}

}

/\*

\* 新建结点(key)，并将其插入到二叉树中

\*

\* 参数说明：

\* tree 二叉树的根结点

\* key 插入结点的键值

\*/

public void insert(T key) {

BSTNode<T> z=new BSTNode<T>(key,null,null,null);

// 如果新建结点失败，则返回。

if (z != null)

insert(this, z);

}

}