## JAVAFX算法

### JAVAFX新建窗口

**private** **void** checkBtn(){

//进入查询学生信息面板

**try** {

FXMLLoader loader = **new** FXMLLoader(getClass().getResource("/CheckPane.fxml"));

Parent conRoot = loader.load();

Stage conStage = **new** Stage();

AddController converController = loader.getController();

converController.init2(**this**, conStage);

conStage.setTitle("查询");

conStage.setScene(**new** Scene(conRoot));

conRoot.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

conStage.show();

} **catch**(Exception e) {

e.printStackTrace();

}

}

**public** **void** init(MenuController controller, Stage stage){

**this**.stage = stage;

}

## 文件操作

### 根据某属性搜索修改txt文件中储存的特定对象

**public** **static** **void** onAbsent(String Number) **throws** FileNotFoundException, Exception {//锟睫革拷学锟斤拷4

**int** i =0;

File Fl = **new** File("Number.txt");

File fl = **new** File("Students.txt");

**try** (Scanner input = **new** Scanner(Fl)) {

i = input.nextInt();

}

Student[] stu = **new** Student[i];

**try**(Scanner Input = **new** Scanner(fl)){

**for**(**int** j =0; j < i;j++){

stu[j] = **new** Student(Input.next(), Input.next(),

Input.next(), Input.nextInt(),Input.nextInt());

**if**(Number.compareTo(stu[j].GetNumber()) == 0){

stu[j] = **new** Student(stu[j].GetName(), stu[j].GetClass(),

stu[j].GetNumber(), stu[j].GetScore(),(stu[j].GetAbsent()+1));

}

}

}

**try**(PrintWriter output = **new** PrintWriter(Fl)){

output.print(i);

}

**if**(i == 0){

**try** (PrintWriter output = **new** PrintWriter(fl)) {

}

}

**else**{

**try** (PrintWriter output = **new** PrintWriter(fl)) {

**for**(**int** j = 0;j < i;j ++){

output.print(stu[j].GetName() + "\t");

output.print(stu[j].GetClass() + "\t");

output.print(stu[j].GetNumber() + "\t");

output.print(stu[j].GetScore() + "\t");

output.println(stu[j].GetAbsent());

}

}

}

}

### 复制文件方法1

@SuppressWarnings("resource")

**public** **void** copyfile(File oldfile,File newfile) **throws** IOException{//从stream处理改为了Channel处理，去除冗余空字节。

FileChannel inputChannel = **null**;

FileChannel outputChannel = **null**;

**try** {

inputChannel = **new** FileInputStream(oldfile).getChannel();

outputChannel = **new** FileOutputStream(newfile).getChannel();

outputChannel.transferFrom(inputChannel, 0, inputChannel.size());

} **finally** {

inputChannel.close();

outputChannel.close();

} ;

}

## 排序算法

### 二叉树排序

1. **public** **class** SortedBinTree <T **extends** Comparable>{
3. **static** **class** Node{
4. Object data;
5. Node parent;
6. Node left;
7. Node right;
9. **public** Node(Object data, Node parent, Node left, Node right) {
10. **this**.data = data;
11. **this**.parent = parent;
12. **this**.left = left;
13. **this**.right = right;
14. }
16. **public** String toString(){
17. **return** "[data=" + data + "]";
18. }
20. **public** **boolean** equals(Object obj){
21. **if**(**this** == obj){
22. **return** **true**;
23. }
25. **if**(obj.getClass() == Node.**class**){
26. Node target = (Node) obj;
27. **return** data.equals(target.data) && left == target.left
28. && right == target.right && parent == target.parent;
29. }
31. **return** **false**;
32. }
34. }
36. **private** Node root;
38. **public** SortedBinTree(){
39. root = **null**;
40. }
42. **public** SortedBinTree(T o){
43. root = **new** Node(o, **null**, **null**, **null**);
44. }
46. /\*\*
47. \* 添加节点
48. \* @param ele 新节点的元素
49. \*/
50. **public** **void** add(T ele){
51. **if**(root == **null**){
52. root = **new** Node(ele, **null**, **null**, **null**);
53. }
54. **else**{
55. Node current = root;
56. Node parent = **null**;
57. **int** cmp;
59. //搜索合适的叶子节点，以该叶子节点为父节点添加新节点
60. **do**{
61. parent = current;
62. cmp = ele.compareTo(current.data);
64. //如果新节点的值大于当前节点的值
65. **if**(cmp > 0){
66. //以当前节点的右子节点作为当前节点
67. current = current.right;
68. }**else**{
69. current = current.left;
70. }
71. }**while**(current != **null**);
73. //创建新节点
74. Node newNode = **new** Node(ele, parent, **null**, **null**);
76. //如果新节点的值大于父节点的值
77. **if**(cmp > 0){
78. parent.right = newNode;
79. }**else**{
80. parent.left = newNode;
81. }
82. }
83. }
85. /\*\*
86. \* 删除节点
87. \* @param ele
88. \*/
89. **public** **void** remove(T ele){
90. Node target = getNode(ele);
92. **if**(target == **null**){
93. **return**;
94. }
96. //左右子树都为空
97. **if**(target.left == **null** && target.right == **null**){
98. **if**(target == root){
99. root = **null**;
100. }
101. **else**{
102. //被删除节点是父节点的左子节点
103. **if**(target == target.parent.left){
104. //将target的父节点的left设为null
105. target.parent.left = **null**;
106. }**else**{
107. target.parent.right = **null**;
108. }
110. target.parent = **null**;
111. }
112. }
114. //左空右不空
115. **else** **if**(target.left == **null** && target.right != **null**){
116. **if**(target == root){
117. root = target.right;
118. }
119. **else**{
120. //被删除节点是父节点的左子节点
121. **if**(target == target.parent.left){
122. target.parent.left = target.right;
123. }
124. **else**{
125. target.parent.right = target.right;
126. }
128. //让target的右子树的parent指向target的parent
129. target.right.parent = target.parent;
130. }
131. }
133. **else** **if**(target.left != **null** && target.right == **null**){
134. **if**(target == root){
135. root = target.left;
136. }
137. **else**{
138. //被删除节点是父节点的左子节点
139. **if**(target == target.parent.left){
140. target.parent.left = target.left;
141. }
142. **else**{
143. target.parent.right = target.left;
144. }
146. //让target的右子树的parent指向target的parent
147. target.left.parent = target.parent;
148. }
149. }
151. //左右都不为空
152. **else**{
153. //leftMaxNode:target的左子树中值最大的节点
154. Node leftMaxNode = target.left;
156. //搜索target的左子树中值最大的节点
157. **while**(leftMaxNode.right != **null**){
158. leftMaxNode = leftMaxNode.right;
159. }
161. //从原来的子树中删除leftMaxNode节点
162. leftMaxNode.parent.right = **null**;
164. leftMaxNode.parent = target.parent;
166. **if**(target == target.parent.left){
167. target.parent.left = leftMaxNode;
168. }
169. **else**{
170. target.parent.right = leftMaxNode;
171. }
173. leftMaxNode.left = target.left;
174. leftMaxNode.right = target.right;
175. target.parent = target.left = target.right = **null**;
176. }
177. }
179. /\*\*
180. \* 根据指定值搜索节点
181. \* @param ele 指定值
182. \* @return 节点
183. \*/
184. **public** Node getNode(T ele){
185. //从根节点开始搜索
186. Node p = root;
187. **while**(p != **null**){
188. **int** cmp = ele.compareTo(p.data);
190. **if**(cmp < 0){
191. p = p.left;
192. }
193. **else** **if**(cmp > 0){
194. p = p.right;
195. }
196. **else**{
197. **return** p;
198. }
199. }
201. **return** **null**;
202. }
204. /\*\*
205. \* 广度优先遍历
206. \* @return
207. \*/
208. **public** List<Node> breadthFirst(){
209. Queue<Node> queue = **new** ArrayDeque<Node>();
210. List<Node> list = **new** ArrayList<Node>();
212. **if**(root != **null**){
213. queue.offer(root);
214. }
216. **while**(!queue.isEmpty()){
217. //将该队列的“队尾”元素添加到List中
218. list.add(queue.peek());
219. //弹出队尾节点
220. Node p = queue.poll();
222. //如果左子节点不为null，将它加入“队列”
223. **if**(p.left != **null**){
224. queue.offer(p.left);
225. }
227. **if**(p.right != **null**){
228. queue.offer(p.right);
229. }
230. }
232. **return** list;
233. }
235. /\*\*
236. \* 中序遍历
237. \* @return
238. \*/
239. **public** List<Node> inIterator(){
240. **return** inIterator(root);
241. }
243. **private** List<Node> inIterator(Node node){
244. List<Node> list = **new** ArrayList<Node>();
246. //递归处理左子树
247. **if**(node.left != **null**){
248. list.addAll(inIterator(node.left));
249. }
251. //处理根节点
252. list.add(node);
254. //递归处理右子树
255. **if**(node.right != **null**){
256. list.addAll(inIterator(node.right));
257. }
259. **return** list;
260. }
261. }

### 冒泡排序

设数组的长度为N：   
（1）比较前后相邻的二个数据，如果前面数据大于后面的数据，就将这二个数据交换。

（2）这样对数组的第0个数据到N-1个数据进行一次遍历后，最大的一个数据就“沉”到数组第N-1个位置。

（3）N=N-1，如果N不为0就重复前面二步，否则排序完成。

/\*\*

\* 冒泡排序的第一种实现, 没有任何优化

\* @param a

\* @param n

\*/

public static void bubbleSort1(int [] a, int n){

int i, j;

for(i=0; i<n; i++){//表示n次排序过程。

for(j=1; j<n-i; j++){

if(a[j-1] > a[j]){//前面的数字大于后面的数字就交换

//交换a[j-1]和a[j]

int temp;

temp = a[j-1];

a[j-1] = a[j];

a[j]=temp;

}

}

}

}// end

下面开始考虑优化，如果对于一个本身有序的序列，或则序列后面一大部分都是有序的序列，上面的算法就会浪费很多的时间开销，这里设置一个标志flag，如果这一趟发生了交换，则为true，否则为false。明显如果有一趟没有发生交换，说明排序已经完成。

/\*\*

\* 设置一个标志，如果这一趟发生了交换，则为true，否则为false。明显如果有一趟没有发生交换，说明排序已经完成。

\* @param a

\* @param n

\*/

public static void bubbleSort2(int [] a, int n){

int j, k = n;

boolean flag = true;//发生了交换就为true, 没发生就为false，第一次判断时必须标志位true。

while (flag){

flag=false;//每次开始排序前，都设置flag为未排序过

for(j=1; j<k; j++){

if(a[j-1] > a[j]){//前面的数字大于后面的数字就交换

//交换a[j-1]和a[j]

int temp;

temp = a[j-1];

a[j-1] = a[j];

a[j]=temp;

//表示交换过数据;

flag = true;

}

}

k--;//减小一次排序的尾边界

}//end while

}//end

再进一步做优化。比如，现在有一个包含1000个数的数组，仅前面100个无序，后面900个都已排好序且都大于前面100个数字，那么在第一趟遍历后，最后发生交换的位置必定小于100，且这个位置之后的数据必定已经有序了，也就是这个位置以后的数据不需要再排序了，于是记录下这位置，第二次只要从数组头部遍历到这个位置就可以了。如果是对于上面的冒泡排序算法2来说，虽然也只排序100次，但是前面的100次排序每次都要对后面的900个数据进行比较，而对于现在的排序算法3，只需要有一次比较后面的900个数据，之后就会设置尾边界，保证后面的900个数据不再被排序。

public static void bubbleSort3(int [] a, int n){

int j , k;

int flag = n ;//flag来记录最后交换的位置，也就是排序的尾边界

while (flag > 0){//排序未结束标志

k = flag; //k 来记录遍历的尾边界

flag = 0;

for(j=1; j<k; j++){

if(a[j-1] > a[j]){//前面的数字大于后面的数字就交换

//交换a[j-1]和a[j]

int temp;

temp = a[j-1];

a[j-1] = a[j];

a[j]=temp;

//表示交换过数据;

flag = j;//记录最新的尾边界.

}

}

}

}}

### 插入排序

public static void insertSort(int[] a) {

int i, j, insertNote;// 要插入的数据

for (i = 1; i < a.length; i++) {

// 从数组的第二个元素开始循环将数组中的元素插入

insertNote = a[i];

// 设置数组中的第2个元素为第一次循环要插入的数据

j = i - 1;

while (j >= 0 && insertNote < a[j]) {

a[j + 1] = a[j];

// 如果要插入的元素小于第j个元素,就将第j个元素向后移动

j--;

}

a[j + 1] = insertNote;

// 直到要插入的元素不小于第j个元素,将insertNote插入到数组中

}

}

### 选择排序

public static void select\_sort(int[] arr){

if(arr == null || arr.length < 2){

return;

}

int index;//定义下标

for(int i=0;i<arr.length;i++){

index = i;//初始下标为i

for(int j=i+1;j<arr.length;j++){

if(arr[j]<arr[index]){

index = j;

}

}

if(index != i){

int temp = arr[index];

arr[index] = arr[i];

arr[i] = temp;

}

}

}

### 快速排序

public static void sort(int a[], int low, int hight) {

int i, j, index;

if (low > hight) {

return;

}

i = low;

j = hight;

index = a[i]; // 用子表的第一个记录做基准

while (i < j) { // 从表的两端交替向中间扫描

while (i < j && a[j] >= index)

j--;

if (i < j)

a[i++] = a[j];// 用比基准小的记录替换低位记录

while (i < j && a[i] < index)

i++;

if (i < j) // 用比基准大的记录替换高位记录

a[j--] = a[i];

}

a[i] = index;// 将基准数值替换回 a[i]

sort(a, low, i - 1); // 对低子表进行递归排序

sort(a, i + 1, hight); // 对高子表进行递归排序

}

public static void quickSort(int a[]) {

sort(a, 0, a.length - 1);

}

public static void main(String[] args) {

int a[] = { 49, 38, 65, 97, 76, 13, 27, 49 };

quickSort(a);

System.out.println(Arrays.toString(a));

}

## 查找算法

### 顺序查找

public static int seqSearch( int[] a, int d){

for( int i=0; i< a.length; i++){

if( d== a[ i]){

return i+1;

}

}

return -1;

}

### 二分查找

1. /\*\*
2. \* 使用递归的二分查找
3. \*title:recursionBinarySearch
4. \*@param arr 有序数组
5. \*@param key 待查找关键字
6. \*@return 找到的位置
7. \*/
8. **public** **static** **int** recursionBinarySearch(**int**[] arr,**int** key,**int** low,**int** high){
10. **if**(key < arr[low] || key > arr[high] || low > high){
11. **return** -1;
12. }
14. **int** middle = (low + high) / 2;          //初始中间位置
15. **if**(arr[middle] > key){
16. //比关键字大则关键字在左区域
17. **return** recursionBinarySearch(arr, key, low, middle - 1);
18. }**else** **if**(arr[middle] < key){
19. //比关键字小则关键字在右区域
20. **return** recursionBinarySearch(arr, key, middle + 1, high);
21. }**else** {
22. **return** middle;
23. }
25. }

# 数据库

## Java通过JDBC连接数据库

auhor: Wang Fei

windows安装mysql

mysql仅提供32位程序，32位和64位系统均可安装 官网下载：link 按照提示，安装MySQL Server和MySQL Connectors-Connector/j

解决中文乱码

修改C:\ProgramData\MySQL\MySQL Server 8.0\my.ini文件 mysql下添加： default-character-set=utf8 mysqld下添加： default-set-server=utf8

管理员权限启动CMD或PowerShell net stop mysql80 net start mysql80 (服务名称可以通过“计算机管理”-“服务”查询)

导入数据

create database Student;

use Student;

source C:\Users\HickeyHsu\Desktop\Student.sql;

select \* from tstudent;

配置JDBC

下载安装JDBC

右键项目“Build Path”-“Add External Archives”添加JDBC包

编写连接对象代码

新建connector类

package application;

import java.sql.\*;

public class Connector {

public static void main(String[] args) {

Connection conn = null;

Statement stmt = null;

//System.out.println(a);

try {

Class.forName("com.mysql.cj.jdbc.Driver"); //线加载驱动

// System.out.println(888);

} catch (ClassNotFoundException e) {

System.out.println("找不到驱动程序类 ，加载驱动失败！");

e.printStackTrace();

}

String url = "jdbc:mysql://localhost:3306/Student?useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone=GMT"; //数据库名称

String username = "root"; //账号

String password = "123123"; //密码

try {

//连接数据库

System.out.println("连接数据库...");

conn = DriverManager.getConnection(url, username, password);

System.out.println("连接成功");

stmt = conn.createStatement();

String sql;

sql = "SELECT \* FROM tstudent";

ResultSet rs = stmt.executeQuery(sql);

// 展开结果集数据库

while(rs.next()){

// 通过字段检索

int id = rs.getInt("Sno");

String name = rs.getString("Sname");

// 输出数据

System.out.print("学号: " + id);

System.out.print(", 名字: " + name);

System.out.print("\n");

}

} catch (SQLException se) {

System.out.println("失败");

se.printStackTrace();

}

}

}