**实验六 变治法**

1. **实验目的**

1．理解变治法的基本思想。

2．掌握运用变治法解决实际问题的方法。

3．通过实例，加深对变治法的理解和运用。

1. **实验要求**

1.熟悉相关的算法设计方法。

2.掌握相关的算法分析方法。

3.能有把算法转变为程序的能力。

1. **主要仪器及耗材**

计算机及相关软件

1. **基础实验**

1．考虑一下求n个数字构成的数组中最大元素和最小元素的问题。

a. 设计一个基于预排序的算法来解该问题并确定它的效率类型。

b. 比较以下三种算法的效率：（1）蛮力法，（2）基于预排序的算法，（3）分治算法（参考实验五第5题）。

**(1)蛮力法：直接对所有元素进行遍历找最大最小值。**

**#include<iostream>**

**#include<cstdio>**

**using namespace std;**

**int main()**

**{**

**int n;**

**cin>>n;**

**int a[n];**

**for(int i=0;i<n;i++)**

**cin>>a[i];**

**int minx=a[0],maxx=a[0];**

**for(int i=1;i<n;i++)**

**{**

**if(minx>a[i])**

**minx=a[i];**

**if(maxx<a[i])**

**maxx=a[i];**

**}**

**cout<<"min:"<<minx<<endl;**

**cout<<"max:"<<maxx<<endl;**

**return 0;**

**}**

**复杂度：线性遍历，O（n）**

1. **.预排序：直接先用STL的sort即可。**

**#include<iostream>**

**#include<cstdio>**

**#include<algorithm>**

**using namespace std;**

**int main()**

**{**

**int n;**

**cin>>n;**

**int a[n];**

**for(int i=0;i<n;i++)**

**cin>>a[i];**

**sort(a,a+n);**

**cout<<"min:"<<a[0]<<endl;**

**cout<<"max:"<<a[n-1]<<endl;**

**return 0;**

**}**

**复杂度：由于用到了快排，O（n\*Log(n)）**

1. **分治算法：如果A中只有2个数，那么比较1次就可以确定最大数与最小数。否则，将A划分成相等的两个子集A1与A2.用算法MaxMin递归地在A1和A2中找最大与最小.令a1,a2分别表示A1与A2中的最大数，b1与b2分别表示A1与A2中的最小数，那么max{a1,a2}与min{b1,b2}就是所需要的结果。**

**#include<iostream>**

**#include<cstdio>**

**using namespace std;**

**void MaxMin(int a[],int low,int high,int &maxe,int &mine) {**

**if(low==high) { //jin仅一个元素**

**maxe=a[low];**

**mine=a[low];**

**} else if(high-low==1) { //两个元素**

**maxe=max(a[low],a[high]);**

**mine=min(a[low],a[high]);**

**} else {**

**int mid=(low+high)/2;**

**int lmaxe,lmine;**

**MaxMin(a,low,mid,lmaxe,lmine);**

**int rmaxe,rmine;**

**MaxMin(a,mid+1,high,rmaxe,rmine);**

**maxe=max(lmaxe,rmaxe);**

**mine=min(lmine,rmine);**

**}**

**}**

**int main() {**

**int n;**

**scanf("%d",&n);**

**int a[n];**

**for(int i=0; i<n; i++)**

**cin>>a[i];**

**int maxe;**

**int mine;**

**MaxMin(a,0,n-1,maxe,mine);**

**cout<<"min:"<<a[0]<<endl;**

**cout<<"max:"<<a[n-1]<<endl;**

**}**

**分治法和蛮力法都是线性复杂度，所以**

**对三种算法做出比较：分治>蛮力法>预排序**

2. 查找变位词

a. 为这个问题设计一个高效的算法:在一个类似英语词典的大文件中找出变位词的所有集合。例如，eat,ate和tea属于同一个变位词集合。

b. 写一个程序实现该算法。

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define WORDMAX 100**

**int charcomp(const void \*x, const void \*y)**

**{**

**return \*(char \*)x - \*(char \*)y;**

**}**

**int main()**

**{**

**char word[WORDMAX], sig[WORDMAX];**

**while (scanf("%s", word) != EOF)**

**{**

**strcpy(sig, word);**

**qsort(sig, strlen(sig), sizeof(char), charcomp);**

**printf("%s %s\n", sig, word);**

**}**

**return 0;**

**}**

3. 写一个程序，为n个不同整数构成的一个给定列表构造一棵AVL树。

**节点类：**

**public class AVLNode<E> {  
  
  
 E element;  
 AVLNode<E> left;  
 AVLNode<E> right;  
 int height;  
  
 public AVLNode(E element) {  
 this(element, null, null);  
 }  
  
 public AVLNode(E element, AVLNode<E> left, AVLNode<E> right) {  
 this.element = element;  
 this.left = left;  
 this.right = right;  
 }  
  
}**

**整个类：**

**public class MyAVLTree<E extends Comparable<E>> {  
  
 private MyAVLTree.AVLNode root;  
  
 public MyAVLTree() {  
 this.root = null;  
 }  
  
 public void insert(E x) {  
 root = insert(x, root);  
 }  
  
 public void remove(E x) {  
 remove(x, root);  
 }  
  
 public int height() {  
 return height(root);  
 }  
  
 */\*\*  
 \* 插入新数据  
 \*/* public AVLNode<E> insert(E x, AVLNode<E> t) {  
 if (t == null) {  
 return new AVLNode<E>(x);  
 }  
 *//先比较 是插左边还是插右边* int compareResult = x.compareTo(t.element);*//调用一个函数比较x和t.element，小于返回负数* if (compareResult < 0) {*//插到左子树上* t.left = insert(x, t.left);  
 *//插入之后要判断是否打破了平衡，因为插入的是左子树，  
 // 只有左子树才会打破平衡，用左子树的高减去右子树的高* if (height(t.left) - height(t.right) == 2) {  
 *//如果等于2，说明平衡被打破了，需要进行调整。就看选择什么方法调整* if (x.compareTo(t.left.element) < 0) {  
 *//如果x小于t的左子树的值，那么x会被插到t的左子树的左子树上，符合LL 用右旋转调整。* t = rightRotate(t);  
 } else {  
 *//如果x大于t的左子树的值，则会被插到t的左子树的右子树上，符合LR，用先左旋转后右旋转来矫正。* t = leftAndRightRotate(t);  
 }  
 }  
 } else if (compareResult > 0) {*//插到右子树上，逻辑和上面一样。* t.right = insert(x, t.right);  
 if (height(t.right) - height(t.left) == 2) {  
 if (x.compareTo(t.right.element) > 0) {  
 t = leftRotate(t);  
 } else {  
 t = rightAndLeftRotate(t);  
 }  
 }  
 } else {  
 *//已经有这个值了* }  
 t.height = Math.*max*(height(t.left), height(t.right)) + 1;  
 return t;  
 }  
  
 */\*\*  
 \* 删除数据  
 \*/* private AVLNode<E> remove(E x, AVLNode<E> t) {  
 if (t == null)  
 return null;  
 int compareResult = x.compareTo(t.element);  
 if (compareResult < 0) {  
 t.left = remove(x, t.left);  
 *//完了之后验证该子树是否平衡* if (t.right != null) { *//若右子树为空，则一定是平衡的，此时左子树相当对父节点深度最多为1, 所以只考虑右子树非空情况* if (t.left == null) { *//若左子树删除后为空，则需要判断右子树* if (height(t.right) - t.height == 2) {  
 AVLNode<E> k = t.right;  
 if (k.right != null) { *//右子树存在，按正常情况单旋转* t = leftRotate(t);  
 } else { *//否则是右左情况，双旋转* t = rightAndLeftRotate(t);  
 }  
 }  
 }  
 if (t.left!=null){ *//否则判断左右子树的高度差  
 //左子树自身也可能不平衡，故先平衡左子树，再考虑整体* AVLNode<E> k = t.left;  
 *//删除操作默认用右子树上最小节点补删除的节点  
 //k的左子树高度不低于k的右子树* if (k.right != null) {  
 if (height(k.left) - height(k.right) == 2) {  
 AVLNode<E> m = k.left;  
 if (m.left != null) { *//左子树存在，按正常情况单旋转* k = rightRotate(k);  
 } else { *//否则是左右情况，双旋转* k = leftAndRightRotate(k);  
 }  
 }  
 } else {  
 if (height(k.left) - k.height == 2) {  
 AVLNode<E> m = k.left;  
 if (m.left != null) { *//左子树存在，按正常情况单旋转* k = rightRotate(k);  
 } else { *//否则是左右情况，双旋转* k = leftAndRightRotate(k);  
 }  
 }  
 }  
 if (height(t.right) - height(t.left) == 2) {  
 *//右子树自身一定是平衡的，左右失衡的话单旋转可以解决问题* t = leftRotate(t);  
 }  
 }  
 }  
 *//完了之后更新height值* t.height = Math.*max*(height(t.left), height(t.right)) + 1;  
 } else if (compareResult > 0) {  
 t.right = remove(x, t.right);  
 *//下面验证子树是否平衡* if (t.left != null) { *//若左子树为空，则一定是平衡的，此时右子树相当对父节点深度最多为1* t = balanceChild(t);  
 }  
 *//完了之后更新height值* t.height = Math.*max*(height(t.left), height(t.right)) + 1;  
 } else if (t.left != null && t.right != null) {  
 *//默认用其右子树的最小数据代替该节点的数据并递归的删除那个节点* AVLNode<E> min = t.right;  
 while (min.left != null) {  
 min = min.left;  
 }  
*// t.element = findMin(t.right).element;* t.element = min.element;  
 t.right = remove(t.element, t.right);  
 t = balanceChild(t);  
 *//完了之后更新height值* t.height = Math.*max*(height(t.left), height(t.right)) + 1;  
 } else {  
 t = (t.left != null) ? t.left : t.right;  
 }  
 return t;  
 }  
  
 private AVLNode<E> balanceChild(AVLNode<E> t) {  
 if (t.right == null) { *//若右子树删除后为空，则只需判断左子树与根的高度差* if (height(t.left) - t.height == 2) {  
 AVLNode<E> k = t.left;  
 if (k.left != null) {  
 t = rightRotate(t);  
 } else {  
 t = leftAndRightRotate(t);  
 }  
 }  
 } else { *//若右子树删除后非空，则判断左右子树的高度差  
 //右子树自身也可能不平衡，故先平衡右子树，再考虑整体* AVLNode<E> k = t.right;  
 *//删除操作默认用右子树上最小节点（靠左）补删除的节点* if (k.left != null) {  
 if (height(k.right) - height(k.left) == 2) {  
 AVLNode<E> m = k.right;  
 if (m.right != null) { *//右子树存在，按正常情况单旋转* k = leftRotate(k);  
 } else { *//否则是右左情况，双旋转* k = rightAndLeftRotate(k);  
 }  
 }  
 } else {  
 if (height(k.right) - k.height == 2) {  
 AVLNode<E> m = k.right;  
 if (m.right != null) { *//右子树存在，按正常情况单旋转* k = leftRotate(k);  
 } else { *//否则是右左情况，双旋转* k = rightAndLeftRotate(k);  
 }  
 }  
 }  
 *//左子树自身一定是平衡的，左右失衡的话单旋转可以解决问题* if (height(t.left) - height(t.right) == 2) {  
 t = rightRotate(t);  
 }  
 }  
 return t;  
 }  
  
  
 */\*\*  
 \* 右旋转  
 \*  
 \* @param t 需要调整的树  
 \* @return 调整后的树  
 \*/* private AVLNode<E> rightRotate(AVLNode<E> t) {  
 AVLNode newTree = t.left;  
 t.left = newTree.right;  
 newTree.right = t;  
 t.height = Math.*max*(height(t.left), height(t.right)) + 1;  
 newTree.height = Math.*max*(height(newTree.left), height(newTree.right)) + 1;  
 return newTree;  
 }  
  
 */\*\*  
 \* 左旋转  
 \*/* private AVLNode<E> leftRotate(AVLNode t) {  
 AVLNode<E> newTree = t.right;  
 t.right = newTree.left;  
 newTree.left = t;  
 t.height = Math.*max*(height(t.left), height(t.right)) + 1;  
 newTree.height = Math.*max*(height(newTree.left), height(newTree.right)) + 1;  
 return newTree;  
 }  
  
 */\*\*  
 \* 先左旋后右旋  
 \*/* private AVLNode<E> leftAndRightRotate(AVLNode<E> t) {  
 t.left = leftRotate(t.left);  
 return rightRotate(t);  
 }  
  
 */\*\*  
 \* 先右旋后左旋  
 \*/* private AVLNode<E> rightAndLeftRotate(AVLNode<E> t) {  
 t.right = rightRotate(t.right);  
 return leftRotate(t);  
 }  
  
  
 */\*\*  
 \* 获取指定树的高度  
 \*/* private int height(AVLNode<E> t) {  
 return t == null ? -1 : t.height;  
 }  
  
 public void printTree() {  
 printTree(root);  
 }  
  
 private void printTree(AVLNode<E> tree) {  
 if (tree == null) {  
 return;  
 }  
 System.*out*.print(tree.element + " ");  
 printTree(tree.left);  
 printTree(tree.right);  
  
 }  
  
 private static class AVLNode<E> {  
 E element;  
 AVLNode<E> left;  
 AVLNode<E> right;  
 int height;  
  
 public AVLNode(E element) {  
 this(element, null, null);  
 }  
  
 public AVLNode(E element, AVLNode<E> left, AVLNode<E> right) {  
 this.element = element;  
 this.left = left;  
 this.right = right;  
 }  
 }  
}**

**主函数：**

**import java.util.Scanner;  
  
public class AVL {  
  
 public static void main(String[] args) {  
  
 MyAVLTree<Integer> avlTree = new MyAVLTree<>();  
  
 Scanner num = new Scanner(System.*in*);*//输入的是节点数* int n = num.nextInt();  
  
 int avl[] = new int[n];  
 Scanner reader = new Scanner(System.*in*);  
 for (int i = 0; i < avl.length; i++) {  
 avl[i] = reader.nextInt();  
 }  
  
 for (int i = 0; i < avl.length; i++) {  
 avlTree.insert(avl[i]);  
 }  
  
  
 System.*out*.println("AVL树的先序遍历:");  
 avlTree.printTree();  
  
 }  
}**

4. 任选一种语言实现堆排序。

**#include<iostream>**

**#include<algorithm>**

**#define maxn 1001 //heap's size**

**using namespace std;**

**struct Heap {**

**int size; // number of elements in array**

**int \*array;**

**Heap() { //init**

**size = 0;**

**array = new int[maxn];**

**}**

**Heap(int n) { //init**

**size = 0;**

**array = new int[n];**

**}**

**~Heap() { //free memory**

**delete array;**

**}**

**bool empty() {**

**if(size != 0) return false;**

**return true;**

**}**

**void insert(int value) {**

**array[++size] = value;**

**int index = size;**

**while(index > 1) {**

**if(array[index] > array[index/2]) swap(array[index],array[index/2]);**

**index /= 2;**

**}**

**}**

**void del()**

**{**

**if(empty()) return;**

**swap(array[1],array[size--]);**

**int index = 1;**

**while(2\*index <= size)**

**{**

**int next = 2\*index;**

**if(next < size && array[next+1] > array[next]) next++;**

**if(array[index] < array[next])**

**{**

**swap(array[index],array[next]);**

**index = next;**

**} else break;**

**}**

**}**

**int max() {**

**if(empty()) return -1;**

**return array[1];**

**}**

**};**

**void buildHeap(int array[],int size) {**

**int i,tmp,index;**

**for(i = size/2; i >= 1; i--) {**

**tmp = array[i];**

**index = 2\*i;**

**while(index <= size) {**

**if(index < size && array[index+1] > array[index]) index++;**

**if(array[index] < tmp) break;**

**array[index/2] = array[index];**

**index \*= 2;**

**}**

**array[index/2] = tmp;**

**}**

**}**

**int main() {**

**int n,i,j,k;**

**cout << "input heap's size:";**

**cin >> n;**

**Heap H = Heap(n);**

**int\* array = new int[n];**

**for(i = 1; i <= n; i++) {**

**int tmp;**

**cin >> tmp;**

**array[i] = tmp;**

**H.insert(tmp);**

**}**

**buildHeap(array,n);**

**for(i = 1; i <= n; i++) {**

**cout << array[i] << " ";**

**}**

**cout << endl;**

**while(!H.empty()) {**

**cout << H.max() << endl;**

**H.del();**

**}**

**return 0;**

**};**

1. **注意事项**

1. 变治法的基于变换思想的技术，用来把问题变换成一种更容易解决的问题。

2. 变治法在解决实际问题的时候，要注意分析问题的特性。