**堆和二叉查找树**

一个数组,始终保持最大/最小的元素在a[0]单元,该数组可以追加元素,可以随时取走a[0]这个最大/最小元素.

解决思路:保证数组a[0]始终存放最大/最小元素,且数组整体分布特征满足一定条件,使得在数组元素发生变化时,在最少的时间内使得a[0]仍是最大/最小元素.

牵扯3个问题:

1. 给一个随机分布的无序数组,怎样把数组元素的位置调整成a[0]是最大元素,且初始化好数组元素分布特征.
2. 追加一个元素,扔保证a[0]最大/最小,且分布特征不变
3. 取走a[0],补充a[0]仍使a[0]最大/最小,且分布特征不变.

如果不维持数组元素的分布特征,每次数组动态变化,保持a[0]最大/最小,时间复杂度都是O(n).如果维持数组元素的分布特征,使得除了第一次初始化数组元素分布特征,时间复杂度是O(n),以后每次保持a[0]最大/最小,时间复杂度都是O(logn).

解决方案:

把数组映射成一棵完全二叉树,a[0]就是根节点,使得每个结点满足条件:以该结点为根的二叉树,左子树和右子树的所有结点都 <= 或 >= 根结点.

堆的特点

以根为基准把树劈成两半，左子树和右子树结点的大小没有必然关系。

建堆复杂度O（n）

(1)downAdjust函数: downAdjust(Type array,int i);假设a[i]结点的左子树右子树都是最堆,a[i]与左孩子右孩子关系还未调整,该函数就是调节a[i].

从完全二叉树最后一个有孩子结点的结点开始逆序往上一直到根结点,对每个结点执行downAdust,便成了堆.

插入O（logn）

插入就是追加到数组尾,即完全二叉树最底层增加一个结点,然后执行一次upAdjust函数即可.

删除O（logn）

删除就是取走a[0],从剩下元素拿来一个补充. 把完全二叉树最后一个结点放到a[0],数组元素个数减1.对a[0]执行downAdjust.

 /\*实现一个最大堆\*/

#include<stdio.h>

#include<stdlib.h>

/\*交换\*/

void swap(int \*a, int \*b) {

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

/\*向下调整\*/

void downAdjust(int \*a,unsigned i,unsigned length) {

unsigned max;

unsigned halfNum = length >> 1;

while (i <= halfNum) {

if (2 \* i + 1 <= length)

max = a[2 \* i] > a[2 \* i + 1] ? 2 \* i : 2 \* i + 1;

else

max = 2 \* i;

if (a[i] < a[max]) {

swap(&a[i], &a[max]);

i = max;

}

else

return;

}

}

/\*建堆\*/

void buildHeap(int \*a, unsigned int length) {

for (int i = length >> 1; i >= 1; --i) {

downAdjust(a, i, length);

}

}

/\*追加调整\*/

void addAdjust(int \*a, unsigned int length,int value) {

}

/\*删除调整\*/

void deleteAdjust(int \*a, unsigned int length) {

}

int main() {

int a[11] = { 10,1, 8, 7, 0, 19, 5, 2, 23, 4, 6 };

buildHeap(a, 10);

for (int i = 1; i <= 10; ++i) {

printf("%d ", a[i]);

}

system("pause");

return 0;

}

复习时，手画一棵查找树，亲自摸清查找树的各操作

二叉查找树的定义

一颗空树或该非空树满足：

1，若有左子树，左子树的所有节点小于根节点

2，若有右子树，右子树的所有节点大于根节点\*/

/\*二叉查找树的一些性质

1.查找某个关键字的比较次数最多是树的深度，

模拟过程是从树根往下沿着

某一路径走到叶子结点。查找性能取决于树的形态

2.同一组数，会因插入的顺序不同，生成不同形态的查找树。

①按照有序序列插入，生成单支树，最坏情况时间复杂度是n/2

②最好情况和折半查找判定树相同，时间复杂度是log2（n）

③随机输入情况下，时间复杂度是log（n）

3.中序遍历二叉树是个有序序列

4，删除一个结点，操作较复杂：

①待删结点p是叶子结点，待删结点的父节点的相应孩子指针域修改成空

②待删结点p只有一棵子树，待删结点的父节点的相应孩子指针域

修改成p->child

③待删结点有两棵子树，分两种情况：

p的左孩子有右孩子，最右的孩子的数据代替p的数据，删除最右孩子

p的左孩子没有右孩子，p的左孩子的数据代替p的数据，删除p的左孩子

总而言之，就是把以p为树根的树的最大结点的值代替p的值，删除之

最大结点其实是中序遍历时，p的直接前驱。

④删除结点是根结点时，要分开讨论，因为根结点没有父节点且会改变\*T

\*/

/\*详细笔记，关于查找2函数\*/

typedef struct node {

int data;

struct node \*lchild, \*rchild;

}BSTNode, \*BST;

/\*在二叉查找树中插入新结点\*/

/\*1，如果树空，即第一次插入结点，树根是新结点

2，树不空，新结点等于根，不必插入，返回错误信息

新结点小于根，插入左子树；大于根，插入右子树\*/

Status BSTInsert(BST \*T, int elem) {

if (NULL == \*T){

\*T = (BST)malloc(sizeof(BSTNode));

(\*T)->data = elem;

(\*T)->lchild = NULL;

(\*T)->rchild = NULL;

}

/\*以下三个if条件语句是互斥执行的\*/

if (elem == (\*T)->data)

return error;

else if (elem < (\*T)->data)

return BSTInsert(&(\*T)->lchild, elem);

else/\*递归必须有return，否则编译器VS13会给出警告，

请仔细思考递归\*/

return BSTInsert(&(\*T)->rchild, elem);

}

/\*调用插入函数，通过数组构建含有n个元素的查找树 \*/

void InputData(BST \*T, int \*a, int n) {

\*T = NULL;

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

BSTInsert(T, a[i]);

}

/\*利用递归查找关键字key，查找成功返回指针，失败返回空指针\*/

BSTNode\* SearchBST(BST T, int key) {

if (NULL == T)

return NULL;//递归结束条件（还可作为形参是空树的处理操作）

else {

if (key == T->data)

return T;//根是目标元素，返回

else if (key < T->data) {//小于根，找左子树

return SearchBST(T->lchild, key);

//【注意】必须要有return，否则编译器会给出警告

}

else {//大于根，找右子数

return SearchBST(T->rchild, key);

}/\*注意：左子树和右子树只找了一个。普通二叉树的查找，左右子树

都找了，注意二者判断条件的区别\*/

}

}

/\*非递归在二叉排序树查找关键字key,查找成功返回指针，失败返回空指针\*/

BSTNode\* SearchBST2(BST T, int key) {

BSTNode \*p = T;

while (NULL != p && key != p->data) {

if (key < p->data)

p = p->lchild;

else

p = p->rchild;

}

return p;

}

/\*删除一个结点，并接上她的左右子树\*/

Status DeleteNode(BST \*T, int key) {

BST p, q, maxchild, secmaxchild;

if (NULL == \*T)

return error;

p = \*T;

q = NULL;

while (NULL != p && key != p->data) {

q = p;

if (key < p->data)

p = p->lchild;

else

p = p->rchild;

}

if (p == \*T) {

printf("删除的是树根，需改变\*T，未编，结束函数\n");

return error;

}

if (NULL == p)

return error;

else {

/\*待删结点p是叶子结点，q是其父节点\*/

if (NULL == p->lchild && NULL == p->rchild) {

if (p == q->lchild)

q->lchild = NULL;

if (p == q->rchild)

q->rchild = NULL;

free(p); p = NULL;

}

/\*待删结点p只有一棵子树，q直接将其删除\*/

else if (NULL != p->lchild && NULL == p->rchild) {

if (p == q->lchild)

q->lchild = p->lchild;

if (p == q->rchild)

q->rchild = p->lchild;

free(p); p = NULL;

}

else if (NULL == p->lchild && NULL != p->rchild) {

if (p == q->lchild)

q->lchild = p->rchild;

if (p == q->rchild)

q->rchild = p->rchild;

free(p); p = NULL;

}

/\*待删结点p有两棵子树\*/

else if (NULL != p->lchild && NULL != p->rchild) {

secmaxchild = p;

maxchild = p->lchild;

while (NULL != maxchild->rchild) {

secmaxchild = maxchild;

maxchild = maxchild->rchild;

}

if (maxchild != p->lchild) {

p->data = maxchild->data;

if (maxchild->lchild != NULL) {

secmaxchild->rchild = maxchild->lchild;

}

}

else {

p->data = maxchild->data;

p->lchild = maxchild->lchild;

}

free(maxchild); maxchild = NULL;

}

return ok;

}

}